Training with Low-precision Embedding Tables

Jian Zhang [1], Jiyan Yang [2], Hector Yuen[2]
[1]Stanford University
[2]Facebook, Inc.
`zjian@stanford.edu,{chocjy,hyz}@fb.com`

Abstract

Starting from the success of Glove and Word2Vec in natural language processing, continuous representations are widely deployed in many other domain of applications. These applications span over encoding textual information to modeling user and items in recommender systems, using embedding vectors to represent a large number of objects. As the cardinality of the object sets increases, the embedding components quickly become the bottleneck in training memory footprint. In this work, we focus on building a system to train continuous embeddings in low precision floating point representation. Specifically, our system performs SGD-style model updates in single precision arithmetics, casts the updated parameters using stochastic rounding and stores the parameters in half-precision floating point. Theoretically, we prove that for strongly convex objectives, our SGD-based training algorithm retains the same convergence rate up to constants. We also present a system-friendly implementation for faster random number generator that increases runtime performance by 30%. We deploy our training system to deep neural networks with low precision embedding tables for recommender systems on top of both public dataset Criteo and an internal dataset at Facebook. We empirically demonstrate that our half-precision floating point training system can achieve generalization performance matching that of single precision training system, with up to 2X memory saving and 1.2X faster training speed.

1 Introduction

Starting from the success of word embedding [Pennington et al., 2014, Mikolov et al., 2013] in natural language processing, continuous representations are widely deployed in many applications domains. Beyond embedding words in natural language, embedding-based approach has also demonstrated state-of-the-art performance in entity representation in recommender systems [Cheng et al., 2016b, Li et al., 2016, Wang et al., 2017, Lian et al., 2018] and knowledge-base constructions [Wu et al., 2018a]. In these applications, a large number of entities are encoded using embedding tables where each row vector represents a single entity; we call these matrices the embedding tables across our discussions in this paper. Because the size of the embedding tables quickly scales with the number of entities, training models on top of large embedding tables can imply significant amount of memory consumption; this can be a major system bottleneck under training memory budget.

To alleviate this problem, we work on memory-efficient training using full precision arithmetic on top of embedding tables stored in lower precision numbers, while the rest of model parameters are stored using normal bit-width. This setting is orthogonal to the one in mixed-precision training using low precision arithmetic [Micikevicius et al., 2017], where the model parameters are still stored in full precision. Training with low precision embedding can bring multiple benefits to both statistical modeling performance and system performance. On the modeling side, low precision embeddings can potentially imply larger model capacity with higher embedding dimensions under the same memory budget. On the system side, low precision embedding not only reduces memory consumption, but also accelerates the training process by communicating embedding vectors in low precision between memory and processor. In this paper, in contrast to [Ling et al., 2016, Li et al., 2017, Lin et al., 2016, Chen et al., 2017, Wu et al., 2018b, De Sa et al., 2018] where fixed-point training is discussed, we focus on low precision floating point embedding table training. This is because fixed-point representation may need extensive tuning to determine the hyperparameters for quantization, e.g., value truncation range. Floating-point, on the other hand, allows for problem and data specific deployment without these intensive quantization related tuning. Though low precision embedding appears appealing, achieving minimal generalization performance degradation comparing to training with full precision embeddings can be challenging. Naively, using stochastic gradient descent (SGD), one can round updated embedding vectors from full precision to the nearest low precision floating-point value before writing back to the memory. Empirically, we observe the deterministic nearest rounding approach can result in significant training and evaluation loss degradation. The importance of rounding scheme is also observed and discussed by Gupta et al. [2015], Ortiz et al. [2018].

In this paper, we work on training algorithms to match the generalization performance of low precision embeddings to that of full precision embeddings. To guide our algorithm design, we first investigate on why the deterministic nearest rounding results in significant generalization performance degradation for training with low precision embeddings. Inspired by our understandings, we then propose a system-friendly training algorithm for low precision embeddings to attain matching generalization performance to that of full precision embeddings. Specifically,

- Our analysis reveals that, in mini-batch SGD training with the deterministic nearest rounding, low precision floating-point embeddings lead to systematic bias when accumulating the model updates. The influence of this bias is especially severe in the typical case where the model update is relatively small comparing to the embedding parameters. In these situations, the convergence of SGD can significantly slow down due to the accumulated update bias.
- Motivated by our insight on the systematic model update bias, we propose to use a stochastic float-point rounding to achieve unbiased model updates. Specifically this unbiased update is achieved by drawing Bernoulli samples for each individual entry in the embedding table. To derive the low precision value written to the embedding table memory, the random sample determines whether to round up or down. Moreover, we derive theory that shows that the incurred additional variance has negligible effects in convergence.
- To allow for efficient real system implementation, we propose an approximated variant of our stochastic rounding approach. In in, we draw Bernoulli samples based on samples from discretized instead of continuous uniform distribution. This can be implemented using fast random bit sequence generation in a batch fashion. Thus it can be significantly faster than the naive implementation with sequential Bernoulli sampling for each individual entry in the embedding table.

We empirically demonstrate the quality of the approximated stochastic rounding approach described above in Section 4. Specifically, for Terabyte Criteo dataset we show that our implementation can attain matching generalization results as baseline model using full precision training. This generalization performance is achieved with approximately 50% less memory and 1.2X throughput. Furthermore, we demonstrate that by using 2X higher embedding dimensions under the same memory budget, models based on low precision embedding can achieve a significant log loss reduction.

Note that in [Ortiz et al., 2018], low-precision floating point training with stochastic rounding is also discussed. Compared to that, we discuss this approach from the optimization perspective and provide rigorous analysis for the convergence rate. We also provide ways to deploy the system efficiently.

The rest of the paper is organized as follows: In Section 2, we demonstrate the systematic update bias from nearest rounding, and discuss how to de-bias with stochastic float-point rounding. In Section 3 and Section 4, we present the proposed approximated stochastic rounding for efficient implementation and present empirical results, respectively. In the rest of the paper we use the notation FP32 and FP16 respectively for IEEE single precision and IEEE half precision for the simplicity in presentation.

2 Training Low-precision Embedding Tables with Stochastic Rounding

The goal in this paper is to train models using full precision arithmetics on top of embedding tables stored as low precision numbers, i.e., FP16, in memory, with generalization performance matching that of training with full precision embeddings. In this section, we first discuss preliminaries on the model update procedure in our setting, which can be characterized by the rounding operations in model updates. In Section 2.1, we empirically demonstrate that deterministic nearest rounding can result in significant performance degradation. To mitigate the performance degradation, we propose using stochastic floating-point rounding in Section 2.2 and analyze the convergence rate of stochastic gradient descent with this stochastic rounding in Section 2.3.

The model update procedure for low precision embeddings. The SGD-style model update for low precision embeddings closely resembles that for full precision embeddings; the only difference lies in the fact that the updated embedding table is cast to low precision before written to memory, as shown in Algorithm 1. We notice the conversion from FP32 to FP16 before writing to memory is lossy and can use different rounding options. For this conversion, one can cast using nearest rounding, i.e., casting the FP32 values to their nearest FP16-representable values. However, we empirically observe that nearest rounding can results in *significant degradation* of training and evaluation loss.

Algorithm 1 Model Update for FP16 Parameters

```
Require: Learning rate $\alpha$ in FP32.
Require: Parameter $w \in \mathbb{R}^d$ in FP16.
Require: Mini-batch gradient $g(\cdot) \in \mathbb{R}^d$ in FP32.
1: Set $\tilde{w} \leftarrow \textsc{HalfToFloat}(w)$
2: Compute $\hat{w} = \tilde{w} - \alpha \cdot g(\tilde{w})$          ▷ Using FP32 arithmetic.
3: Set $w \leftarrow \textsc{RoundToHalf}(\hat{w})$ ▷ Can use different rounding.
4: return $w$
```

2.1 Parameter update bias from nearest rounding

To understand the systematically biased model update accumulation using nearest rounding, we first use a concrete example where the bias is generated in a single update step. In Figure 1, we add a small model update $-\alpha g(\tilde{w}) \approx 4.5776367e^{-5}$ to a relatively large embedding parameter $\tilde{w} = 1.5$ using FP32 arithmetic, following the procedures in Algorithm 1. To add the small model update to the large parameter value, the significand of the model update first shifts right to match the exponent of the other number. After the two significands are added together, the FP32 result is then cast into FP16 using nearest rounding, only preserving the significand until the yellow cutoff bit in Figure 1. We observe the cut-off bit becomes 0 for the FP16 value after nearest rounding. This is a consequence of two facts: 1) the significand bits after the cut-off bit are all 0s for FP32 valued $\tilde{w}$ because it is cast from FP16 value w; 2) the most significant non-zero bit of the model update is multiple bits away after the cut-off bit.

Generalizing from this example, we notice that *when adding a small positive update in FP32 to a significantly larger positive parameter in FP16, nearest rounding can results in a systematically smaller value than the FP32 value before rounding; similarly, when adding a small negative update in FP32 to a large positive parameter in FP16, the FP16 result is systematically larger than the FP32 value*. This systematic bias occurs when the model updates become significantly smaller than the parameter values. We notice that this is a typical situation in training deep learning models, e.g., when using decayed learning rate or Adagrad optimizer, the model updates usually become smaller as the training progresses. The systematic bias described above can result in catastrophically biased accumulation of model updates. In Appendix A, we present an example which shows the accumulation of bias in one of our experiments.

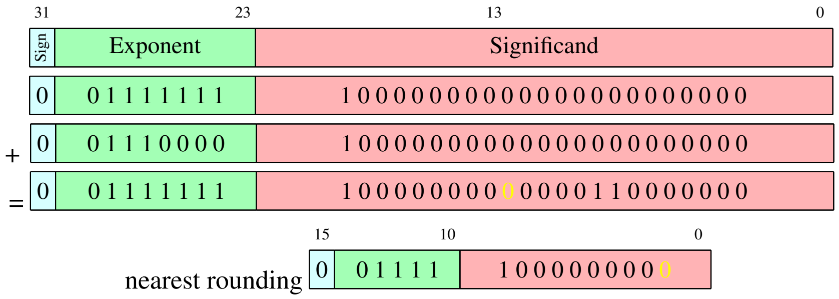

Figure 1: Nearest rounding can result in biased model updates. When adding a small positive model updates to a large parameter value, the FP16 value after nearest rounding can be systematically smaller than the FP32 value before rounding. Example is adding $4.5776367e^{-5}$ to 1.5.

2.2 Stochastic rounding

To make the model update accumulation unbiased, we propose to use *floating-point stochastic rounding*. As shown in Algorithm 1, for each parameter we update, we first perform updates in FP32 arithmetic to obtain $\hat{w}$. To perform the stochastic rounding, as shown in Algorithm 2, we compute the rounded-up value $\hat{w}^{\text{up}}$ (the smallest FP16 value that is larger than or equal to $\hat{w}$) as well as the rounded down values $\hat{w}^{\text{down}}$ (the largest FP16 value that is smaller than or equal to $\hat{w}$) for $\hat{w}$. We then draw a random number from a Bernoulli distribution with parameter $p = (\hat{w} - \hat{w}^{\text{down}})/(\hat{w}^{\text{up}} - \hat{w}^{\text{down}})$ to determine if we round $\tilde{w}$ up or down. That is, we let the rounded value be $\hat{w}^{\text{up}}$ with probability p and be $\hat{w}^{\text{down}}$ with probability $1 - p$. One can show that the output of this stochastic rounding is an unbiased estimate of FP32 value $\hat{w}$.

Note that it is possible to apply this algorithm with different optimizers that adjust learning rate differently, e.g., Adagrad [Duchi et al., 2011]. Next, we analyze the convergence property for the case where SGD is used.

Algorithm 2 Floating point stochastic rounding

```
Require: Updated model parameter $\hat{w}$ in FP32.
Require: The set $S$ of FP16 representable values
1: $\hat{w}^{\text{up}} = \min_{x \geq \hat{w}, x \in S} x$, $\hat{w}^{\text{down}} = \max_{x \leq \hat{w}, x \in S} x$
2: $p = (\hat{w} - \hat{w}^{\text{down}})/(\hat{w}^{\text{up}} - \hat{w}^{\text{down}})$
3: $Z \sim \text{Bernoulli}(p)$
4:
       $w = \begin{cases} \hat{w}^{\text{up}} & \text{if } Z = 1 \\ \hat{w}^{\text{down}} & \text{otherwise} \end{cases}$
5: return $w$
```

2.3 Theoretical guarantee

As mentioned above, one appealing property of low-precision training with stochastic rounding is that the effective gradient update it results in is still unbiased. However, stochastic rounding comes with a cost of higher variance in SGD. Nevertheless, in Lemma 2 we show that the variance incurred can be controlled by the gradient variance and the learning rate. In Theorem 1 we show how this affects the overall convergence of the SGD algorithm for strongly-convex problems. The proof can be found in Appendix D.

Theorem 1. *Let $F(w)$ be a function that is μ-strongly convex. Assume $F(w) = \mathbb{E}_i [f_i(w)]$ and $\mathbb{E}\left[\|\nabla f_i(\cdot)\|^2\right] \leq G^2$. If w is initialized in FP16 and SGD is used to update w with Algorithm 1 with stochastic rounding described in Algorithm 2, then we have*

$$\mathbb{E}\left[F(\bar{w}_T) - F(w^*)\right] \leq \frac{(1+2\epsilon)^2 G^2}{2\mu} \frac{1 + \log(T+1)}{T+1} + \frac{\epsilon}{2} G \|w^*\|, \tag{1}$$

where w^* is the optimal point, $\bar{w}_t = \frac{1}{T+1} \sum_{t=0}^{T} w_t$, and $\epsilon = 2^{-10}$ which is a fixed value based on FP16 representation. Learning rate in each iteration is chosen to be $\alpha_t = \frac{1+\epsilon t + 2\epsilon}{\mu(t+1)}$.

As can be seen, doing low-precision training with stochastic rounding preserves the same convergence rate up to constants. There is an unavoidable error $\frac{\epsilon}{2} G\|w^*\|$ as the final solution is represented using FP16. When low-precision training is not used, i.e., $\epsilon = 0$, we recover the standard bound.

Despite the promising theoretical property, it is important to derive high-performance implementation to make the approach scalable and efficient. We provide details regarding this in the next section.

3 System Implementation

In this section we describe ideas involved in finding a high performance implementation. One of the major challenges is the random number generator as stochastic rounding requires at least one random number per update for each embedding table entry. We need to ensure that this overhead is small compared to the rest of the computation. In this regard, we describe our main approach in Section 3.1. We also consider other ideas such as vectorization and prefetching. However, due to page limit, we leave the details in Appendix B. We present micro-benchmarking results for these improvements in Section 3.2. The implementation and experiments we present in this paper are done on Intel Xeon CPUs, but these concepts are easily extendible to other hardware architectures.

3.1 Random Number Generator

The use of stochastic rounding requires the ability to generate random numbers efficiently. A straightforward approach is to use the Bernoulli distribution implementation provided by the standard library. This is the implementation that generates pseudo-random numbers at a reasonable rate which can be used to determine the result of stochastic rounding for an element; so we are going to use as this a baseline for comparison.

We made a series of incremental changes which lead to increased performance. First, we make use of the Intel MKL libraries [Intel, 2009]. This library provides us the ability to generate a stream of random numbers in floating point format, and it gives 137X relative speedup compared to baseline.

Next, we observe that random floating point numbers are generated from a random stream of bits that get scaled into the desired range and converted into a floating point format. Making use of the random stream directly as part of stochastic rounding can bring some performance improvements by avoiding unnecessary conversions.

As shown in Figure 2, rounding can be thought as removing bits from 0 to 12 from a single precision float and rounding up or down depending on the value of bit 12. Stochastic rounding can be approximated by adding uniformly distributed bits from bit 0 to bit 12 and then doing round to zero when doing the conversion. That is, stochastic rounding can be thought of as adding 12 random bits to the least significant digits of the significand and performing a truncation.

In our setting we found that given the frequency of updates, not all the bits of the random number are required, we test by keeping only the most significant bits of the random number and find that using 8 bits is a good balance of performance and accuracy. The intuition is that for an entry that gets updated 2^n times we need at least n random bits. This results in having more throughput on the random stream since less bits per rounding are needed. Our micro-benchmarking results show that this approach 192X relative speedup compared to the baseline approach using the standard library.

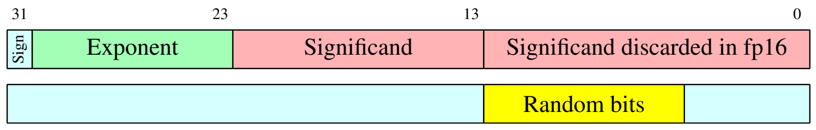

Figure 2: Stochastic rounding can be achieved by adding random bits starting from the discarded portion of the significand. Our empirical results show that 8 bits provide results equivalent to the single precision approach.

3.2 Micro-benchmarking results

In order to evaluate the approaches described above, we create a controlled experiment where we use an embedding table of 16 million rows each of which contains 64 elements in FP16, and updates are applied to 4 million randomly chosen rows from the original table. The framework used to implement it is Caffe2 [1], and we use Adagrad [Duchi et al., 2011] as the optimizer here. Auxiliary variables in Adagrad such as moment estimate are stored in the same data type as the embedding tables.

With all the improvements described above and in Appendix B, we are able to produce a faster implementation that is around 30% faster than the original one while saving roughly half the memory. Table 1 summarizes all the performance measurements as well as their relative computation speedup compared to single-precision implementation.

APPROACH NAME	THROUGHPUT (MITEMS/SEC)	BANDWIDTH (GB/S)	RELATIVE SPEEDUP
FP32 + SIMD	8.77	11.27	1
FP16 nearest rounding + SIMD	12.6	9.7	1.43
FP16 stochastic rounding	0.75	0.58	0.085
FP16 stochastic rounding + SIMD	3.01	2.31	0.34
FP16 stochastic rounding + MKL rng + SIMD	10.77	8.31	1.22
FP16 stochastic rounding + MKL rng + SIMD + `rsqrt`	11.5	8.9	1.31

Table 1: Performance measurements for different Adagrad implementations. The measurements are done on a table of 16 million rows with 64 elements per row and updates of 4 million random rows chosen from a uniform distribution. FP32 and FP16 are used as baselines. MKL rng is described in Section 3.1. Details of other improvements such as SIMD and `rsqrt` can be found in Appendix B.

4 Experiments

In this section we present the experimental results of the proposed approach. We validate our approach using a benchmark real dataset, the Terabyte Criteo data [2]. The Terabyte Criteo data is a click prediction dataset that has a size of 1.3TB, comprising more than 4.3 billion records. It is a common benchmark dataset for ranking applications.

This is a binary classification problem and we use DNN [LeCun et al., 2015] as our main model. For categorical features, we adopt a similar procedure as in [Wang et al., 2017] and [Cheng et al., 2016a] to transform them into dense vectors using embedding tables. The first dimension of the

[1]`https://pytorch.org/`

[2]`https://www.criteo.com/news/press-releases/2015/07/criteo-releases-industrys-largest-ever-dataset/`

embedding table is proportional to the cardinality of the categorical feature with a maximum of 50 million. The second dimension is the embedding size chosen by the user. We then concatenate dense embeddings of all the categorical features along with the rest of the input features. This is followed by 2 fully-connected layers with width 512. We use Adagrad [Duchi et al., 2011] as the optimizer with a batch size of 100 for training all the variables in the network. We tune hyper-parameters for the full-precision setting. We use a base learning rate of 0.015 for embedding tables and 0.005 for all of the rest of the parameters. More details regarding this setup can be found in Appendix C.

Based on the setup described above, the embedding tables account for a large memory consumption. Thus we perform low-precision training for all embedding tables with first dimension larger than 1000000. That is, each element in each of these embedding tables is initialized in FP16 and we apply Algorithm 1 with stochastic rounding described in Algorithm 2 to update the elements. The rest of the parameters are trained in full-precision. We train the model with the architecture described above with various the embedding size for the categorical features. We use full-precision training as a baseline. Results can be found in Table 2.

TRAINING APPROACH	EMBEDDING SIZE	LOG LOSS	MEMORY USAGE
FP16 with stochastic rounding	8	**0.12702**	**10.42 GB**
FP32	8	0.12704	18.59 GB
FP16 with stochastic rounding	16	**0.12662**	**18.84 GB**
FP32	16	0.12661	35.17 GB
FP16 with stochastic rounding	32	**0.12629**	**35.61 GB**
FP32	32	0.12625	68.27 GB
FP16 with stochastic rounding	64	**0.12606**	**69.23 GB**
FP32	64	0.12603	134.45 GB
FP16 with stochastic rounding	128	**0.12598**	**124.69 GB**

Table 2: Log loss and memory usage with different embedding size using both FP32 training (baseline) and FP16 training with stochastic rounding.

As can be seen in the table, FP16 training with stochastic rounding doesn't increase the log loss much. As expected, the memory usage is reduced by a half. The training speed of our approach is 1.2X faster than baseline for each fixed embedding size. The reduced memory budget can be used to double the embedding size for better prediction accuracy. This is validated by the experimental results as we can see that it generates more than 0.001 log loss gain after doing so with similar memory consumption compared to the baseline, which is considered as practically significant. We want to note that using nearest rounding generated undesirable results. For example, for embedding size of 64, the log loss is 0.12652 as opposed to 0.12606 using stochastic rounding. All results can be found in Appendix C.

Moreover, in order to test our approach with larger-scale experiments, we compare our method to baseline in one of the ranking applications at Facebook. This application needs to train a neural network model with billions of parameters on billions of records. Being able to reduce the memory usage of training while maintaining similar model quality and training speed is a challenging task.

Our experimental results show that low-precision training with stochastic rounding reduces total memory usage by 50% from hundreds of Gigabytes to half of it, while the model quality and training speed stay neutral. This proves the practicality of our approach in very large scale setting.

5 Conclusions and Future Work

Theoretically we show that low-precision floating point training with stochastic rounding inherits similar convergence rate as single-precision training and empirically we show that our efficient implementation yields similar generalization performance on Criteo dataset as well as a Facebook internal dataset with 2X memory reduction and 1.2X throughput.

In the future we want to make use of bigger embedding tables, perhaps different bucketing strategies. We would like to explore different hardware architectures since some of them support efficient half float operations. It would be interesting to explore if the same results can be achieved by using fewer bits during stochastic rounding, and if this scheme could be adopted in fixed-point formats.

It is worth noting that while adaptive precision representations are not supported in current DNN accelerators, such schemes can be prototyped efficiently in FPGAs and if there is a compelling enough use case they can be realized into specialized ASICs.

Acknowledgments. We would like to thank Wei Wen, Bor-Yiing Su, Jongsoo Park, Dheevatsa Mudigere, Misha Smelyanskiy, and Liang Xiong for the discussions and the support of this work.

References

Xi Chen, Xiaolin Hu, Hucheng Zhou, and Ningyi Xu. Fxpnet: Training a deep convolutional neural network in fixed-point representation. In *IJCNN*, 2017.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide and deep learning for recommender systems. In *RecSys*, 2016a.

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*, 2016b.

Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, 2015.

Intel. *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009. Santa Clara, USA. ISBN 630813-054US.

Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. Training quantized nets: A deeper understanding. In *Advances in Neural Information Processing Systems 30*. 2017.

Mu Li, Ziqi Liu, Alexander J. Smola, and Yu-Xiang Wang. Difacto: Distributed factorization machines. In *WSDM*, 2016.

Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining*, KDD '18, 2018.

Darryl Dexu Lin, Sachin Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *ICML*, 2016.

Shaoshi Ling, Yangqiu Song, and Dan Roth. Word embeddings with limited memory. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, 2016.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 2013.

Marc Ortiz, Adrián Cristal, Eduard Ayguadé, and Marc Casas. Low-precision floating-point schemes for neural network training. *CoRR*, abs/1804.05267, 2018.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014.

Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *ADKDD@KDD*, 2017.

Sen Wu, Luke Hsiao, Xiao Cheng, Braden Hancock, Theodoros Rekatsinas, Philip Levis, and Christopher Ré. Fonduer: Knowledge base construction from richly formatted data. In *Proceedings of the 2018 International Conference on Management of Data*, 2018a.

Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *CoRR*, abs/1802.04680, 2018b.

A Supplementary Experimental Results using Nearest Rounding

In Figure 3, we plot the value of a specific embedding parameter entry as a function of the iteration count for single-precision training and half-precision training with nearest rounding, respectively. In this plot we collect the FP32 model update sequence for one of our experiments; see Section 4 for more details about the setup. This update sequence is then accumulated using a FP32 floating point number and a FP16 floating point number with nearest rounding.

We can observe that, after certain iterations, the FP16 value is biased towards being larger than the FP32 value. We notice in this model update sequence, there are approximately 2x more negative updates than positive ones added to the positive entry. According to our analysis of the systematic bias in Section 2.1, the FP16 accumulation result is expected to be larger than the FP32 accumulation result due to large number of negative updates to the positive parameter. This severe problem from biased model update accumulation motivates us to use stochastic rounding to generate more accurate updates.

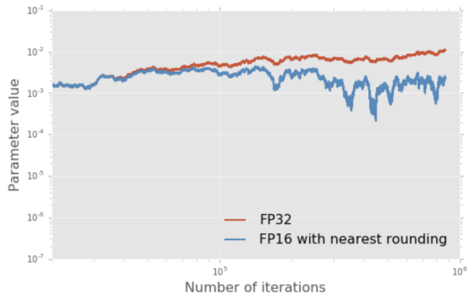

Figure 3: Evolution of a specific parameter value versus iteration number using single-precision training and half-precision training with nearest rounding.

B Supplementary Details of System Implementation

In this section we present the details of system implementation that is not covered by Section 3. Recall we plan to improve the system implementation from a few directions. Besides random number generator improvement, we discuss vectorization and prefetching, low-precision improvement in the next two subsections, respectively.

B.1 Vectorization and Prefetching

The bulk of the computation is implemented inside the Adagrad optimizer in the Caffe2 framework. To improve performance we make changes to the Adagrad Optimizer by explicitly vectorizing its implementation using SIMD intrinsics. On systems supporting the AVX2 instruction set can provide significant speedups. Most of the operations are successfully replaced with their SIMD equivalent except for parts involving operations with the random number generator part. The challenge is that the distribution generator has to be invoked with a parameter which is different per element, preventing us from vectorizing it.

A first optimization is to draw random numbers in the interval of $[0, 1]$ and compare them to p. These numbers can be pre-generated in a bigger batch size and used later, amortizing the fixed startup cost over a large set of elements. Experimental findings show that generating batches of 1024 random numbers provides a good balance of extra memory and performance.

Since the accesses to the table are sparse and random, it is hard for the hardware prefetching mechanism to predict the access pattern. To alleviate this issue we make use of software prefetching since we know the rows that will be accessed a-priori. We make use of the vprefetch0 instruction which gives guidance to the CPU about which would be the next region in memory to be be processed.

The prefetch call takes the memory address of the element to be prefetched as an argument. This has to be timed properly because if we put a very early row in the cache, it might already be in the cache, on the other hand if we prefetch too far away, this will result in still a cache miss and perhaps that future entry being flushed since more space is needed for useful calculations, in both cases rendering the operation useless. To compute this address to prefetch we use the following rule of thumb:

$$\frac{\text{row size in bytes} * \text{prefetch distance}}{\text{memory bandwidth}} >= \text{Memory latency}$$

This approach is used in the single-precision floating point version as well as the half-float one, the difference is that the width of the row of half floats is half the size of the row of floats, so the prefetch distance has to be adjusted accordingly.

B.2 Low Precision improvements

One of the most computationally expensive operations during the computation of Adagrad is the reciprocal square root. As a reminder, the update to the parameter θ is computed as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \tag{2}$$

where η is the learning rate, G is the second moment estimate and ϵ is a small value to avoid division by zero.

In some scenarios such as computer graphics applications it is possible to use a faster but less precise implementation to compute the reciprocal of a square root. One of these implementations is the `rsqrt` operator provided as a SIMD intrinsic which provides up to 11 bits of precision. Since we are using half floats we have an opportunity to take advantage of this form of computation since half float significands are only 10 bits; so we are well within the precision requirements for this data type. With optimization this we are able to surpass the runtime performance of the single precision floating point baseline.

C Supplementary Details of Experiments

Below we present more details regarding the dataset we use and the setup of the experiments.

Dataset description The Terabyte Criteo data is a click prediction dataset that has a size of 1.3TB, comprising more than 4.3 billion records. It has 13 integer features and 26 categorical features where each category has various cardinalities. It is taken from 24 days of click data made available by Criteo. Criteo dataset is a common benchmark dataset for ranking applications. We use the data belonging the first 23 days as training set and use the data from the last day as test set.

Experiment setup This is a binary classification problem and we use DNN [LeCun et al., 2015] as our main model. For categorical features, we adopt a similar procedure to transform them into dense vectors instead of doing naive one-hot encoding which may explode the input dimensionality for the subsequent fully-connected layers.

The dense embedding of each categorical feature is formulated in the following manner. For each categorical feature, we let x denote its one-hot encoded version, e.g., $x = (0, \ldots, 0, 1, 0, \ldots, 0)$ where the coordinate of 1 is corresponding to the activated level of this feature. We then let

$$\tilde{x} = Wx$$

as the dense representation of x where $W \in \mathbb{R}^{m \times d}$. In above, m is the cardinality of this categorical feature and d is the embedding dimension chosen. We call the variable W as embedding table which is jointly trained with the rest of the model. Similar approach is also described in [Wang et al., 2017] and [Cheng et al., 2016a].

As the raw value of each categorical feature is an ID, we use hashing to map each ID to an index range. We choose the hash size to be proportional to the cardinality of this categorical feature with a maximum of 50 million. We concatenate dense embeddings of all the categorical features along with the rest of the input features. This is followed by 2 fully-connected layers with width 512. We use Adagrad [Duchi et al., 2011] as the optimizer with a batch size of 100 for training all the variables in the network. We tune hyper-parameters for the full-precision setting. We use a base learning rate of 0.015 for embedding tables and 0.005 for all of the rest of the parameters.

Based on the setup described above, the embedding tables account for a large memory consumption. Thus we perform low-precision training for the embedding tables with first dimension larger than 1000000. That is, each element in each of these embedding tables is initialized in FP16 and we apply Algorithm 1 for updating the elements. During the forward pass, we use FP32 arithmetics. That is, after we fetch the rows in the embedding table corresponding to the activated IDs in the batch, we convert them to FP32 before doing summation.

Results We train the model with architecture described above with various the embedding size for the categorical features. We use full-precision training as a baseline. Results can be found in Table 3.

TRAINING APPROACH	EMBEDDING SIZE	LOG LOSS	MEMORY USAGE
FP16 with stochastic rounding	8	**0.12702**	**10.42 GB**
FP16 with nearest rounding	8	0.12769	10.42 GB
FP32	8	0.12704	18.59 GB
FP16 with stochastic rounding	16	**0.12662**	**18.84 GB**
FP16 with nearest rounding	16	0.12715	18.82 GB
FP32	16	0.12661	35.17 GB
FP16 with stochastic rounding	32	**0.12629**	**35.61 GB**
FP16 with nearest rounding	32	0.12674	35.70 GB
FP32	32	0.12625	68.27 GB
FP16 with stochastic rounding	64	**0.12606**	**69.23 GB**
FP16 with nearest rounding	64	0.12651	69.26 GB
FP32	64	0.12603	134.45 GB
FP16 with stochastic rounding	128	**0.12598**	**124.69 GB**

Table 3: Log loss and memory usage with different embedding size using both FP32 training (baseline), FP16 training with nearest rounding, and FP16 training with stochastic rounding.

D Proofs

D.1 Variance of Stochastic Rounding

Lemma 2. *For any FP16 representable number $a \in \mathbb{R}$, for any number $v \in \mathbb{R}$, let l and u be the largest FP16 representable number smaller than or equal to $a + v$ and smallest FP16 representable number greater than or equal to $a + v$, respectively. Let X be a random variable being l with probability $\frac{u-(a+v)}{u-l}$ and being u with probability $\frac{a+v-l}{u-l}$. We have*

$$\mathbb{E}\left[(X-(a+v))^2\right] \le \epsilon\left(|a|\cdot|v| + v^2\right) \tag{3}$$

where ϵ is a fixed value related to FP16 representation.

Proof. For simplicity, we denote $a + v$ by c. Based on the definition of x, it is not hard to see that

$$\mathbb{E}\left[(X-c)^2\right] = u(c-l) + cl - c^2 = (u-c)(c-l). \tag{4}$$

Case 1: When $u > l > 0$

By the definition of l and u, they must be two consecutive numbers that are FP16 representable. We have

$$u - l \le \epsilon_1 \cdot l \le \epsilon_2 \cdot u, \tag{5}$$

where ϵ is a fixed value related to FP16 representation, independent of the value being rounded.

Now we calculate ϵ. An FP16 number can be expressed as $s \cdot 2^e\left(1 + \sum_{i=1}^{10} 2^{-i} b_i\right)$. Where s is the sign, e is the exponent and b are the bits on the significand. As l and u are two consecutive numbers that are FP16 representable, their exponents can at most differ by 1.

We want to find ϵ to satisfy

$$u - l \le \epsilon \cdot l. \tag{6}$$

For the case of the same exponent, the smallest difference between the two significands is 2^{-10}. Thus ϵ has to satisfy

$$\frac{2^e \cdot 2^{-10}}{2^e \cdot \text{significand}_l} \leq \epsilon. \tag{7}$$

Since the range of significand$_l$ is $[1, 2)$, we set $\epsilon = 2^{-10}$.

For the case where the exponent is different, we have

$$\begin{aligned} l &= 2^e\Big(1 + \sum_{i=1}^{10} 2^{-i}\Big) && (8) \\ u &= 2^{e+1} && (9) \end{aligned}$$

In this case,

$$\begin{aligned} \frac{u-l}{l} &= \frac{2 - 1 - \sum_{i=1}^{10} 2^{-i}}{1 + \sum_{i=1}^{10} 2^{-i}} && (10) \\ &= \frac{2^{-10}}{2 - 2^{-10}} && (11) \\ &< 2^{-10}. && (12) \end{aligned}$$

Therefore, $\epsilon = 2^{-10}$ satisfies the bound.

Case 1.1: When $v \geq 0$

In this case since l is the largest FP16 representable number smaller than or equal to c and $a = c - l \leq c$, we have $a \leq l$. Therefore,

$$c - l \leq c - a = v. \tag{13}$$

By

Following Equation 4, we have

$$\begin{aligned} \mathbb{E}\left[(X-c)^2\right] &\leq (u-c)v && (14) \\ &\leq (u-l)v && (15) \\ &\leq \epsilon l v && (16) \\ &\leq \epsilon(a+v)v && (17) \\ &= \epsilon(av + v^2). && (18) \end{aligned}$$

Case 1.2: When $v < 0$

In this case since u is the smallest FP16 representable number greater than or equal to c and $a = c - v \geq c$, we have $a \geq u$. Therefore,

$$u - c \leq a - c = |v|. \tag{19}$$

Following Equation (4), we have

$$\begin{aligned} \mathbb{E}\left[(X-c)^2\right] &\leq |v| \cdot (c-l) && (20) \\ &\leq |v| \cdot (u-l) && (21) \\ &\leq \epsilon \cdot |v| \cdot u && (22) \\ &\leq \epsilon \cdot |v| \cdot a. && (23) \end{aligned}$$

Case 2: When $l < u < 0$

Again, l and u must be two consecutive numbers that are FP16 representable. We have

$$u - l \leq \epsilon \cdot |u| \leq \epsilon \cdot |v|. \tag{24}$$

The ϵ here is still 2^{-10} since the derivation of ϵ above is independent of the sign of u and l.

Case 2.1: When $v \geq 0$ Since the reasoning in **Case 1.1** is independent of the value of l and u, we can reuse Equation (13).

Also, notice that in this case $a \leq l$. Therefore, we have

$$|l| \leq |a|. \tag{25}$$

Following Equation 4, we have

$$\begin{aligned} \mathbb{E}\left[(X-c)^2\right] &\leq (u-c)v && (26)\\ &\leq (u-l)v && (27)\\ &\leq \epsilon|l|\cdot v && (28)\\ &\leq \epsilon|a|\cdot v. && (29)\end{aligned}$$

Case 2.2: When $v < 0$ Since the reasoning in **Case 1.2** is independent of the value of l and u, we can reuse Equation (19).

Also, notice that in this case $c \leq u$. Therefore, we have

$$|u| \leq |c|. \tag{30}$$

Following Equation (4), we have

$$\begin{aligned} \mathbb{E}\left[(X-c)^2\right] &\leq |v|\cdot(c-l) && (31)\\ &\leq |v|\cdot(u-l) && (32)\\ &\leq \epsilon\cdot|v|\cdot|u| && (33)\\ &\leq \epsilon\cdot|v|\cdot|c| && (34)\\ &= \epsilon\cdot|v|\cdot|a+v| && (35)\\ &\leq \epsilon\cdot(|v|\cdot|a|+v^2) && (36)\end{aligned}$$

This completes the proof.

□

D.2 Variance of Quantization Noise in SGD Updates

Lemma 3. *For $w_t, g_t, w^* \in \mathbb{R}^d$ and $\alpha_t \in \mathbb{R}^+$, let $r_t = Q(w_t - \alpha_t g_t) - (w_t - \alpha_t g_t)$ where $Q(\cdot)$ is the stochastic rounding function. Then we have*

$$\mathbb{E}\left[\|r_t\|^2\right] \leq \epsilon\cdot(\|w_t - w^*\|^2 + 2\alpha_t^2\|g_t\|^2 + \alpha_t\|w^*\|\|g_t\|^2). \tag{37}$$

Proof. For simplicity, we let

$$y = w_t - \alpha_t g_t. \tag{38}$$

We try to bound $\mathbb{E}\left[\|Q(y)-y\|^2\right]$. As we apply stochastic rounding on each coordinate of y, we analyze each $Q(y_i) - y_i$ for $i = 1, 2, \ldots, d$.

Since $y_i = w_{t,i} - \alpha_t g_{t,i}$, by setting $a = w_{t,i}$ and $v = -\alpha_t g_{t,i}$ in Lemma 2, we have

$$\mathbb{E}\left[(Q(y_i) - y_i)^2\right] \leq \epsilon\cdot\left(\alpha_t\cdot|w_{t,i}|\cdot|g_{t,i}| + \alpha_t^2\cdot g_{t,i}^2\right). \tag{39}$$

In above, $w_{t,i}$ denotes the i-th coordinate of w_t, and similar for $g_{t,i}$.

Then we have

$$\begin{aligned}
\mathbb{E}\left[\|r_t\|^2\right] &= \mathbb{E}\left[\|Q(y)-y\|^2\right] && (40)\\
&= \sum_{i=1}^{d}\mathbb{E}\left[(Q(y_i)-y_i)^2\right] && (41)\\
&\leq \epsilon\left(\sum_{i=1}^{d}|w_{t,i}|\cdot\alpha_t\cdot|g_{t,i}|+\sum_{i=1}^{d}\alpha_t^2\cdot g_{t,i}^2\right) && (42)\\
&\leq \epsilon\left(\sum_{i=1}^{d}|w_{t,i}-w_i^*|\cdot\alpha_t\cdot|g_{t,i}|+\sum_{i=1}^{d}|w_i^*|\cdot\alpha_t\cdot|g_{t,i}|+\sum_{i=1}^{d}\alpha_t^2\cdot g_{t,i}^2\right) && (43)\\
&:= \epsilon\cdot(A+B+C). && (44)
\end{aligned}$$

Using the fact that $ab \leq (a^2+b^2)/2 \leq a^2+b^2$, we have

$$\begin{aligned}
A &\leq \sum_{i=1}^{d}(w_{t,i}-w_i^*)^2+\alpha_t^2\sum_{i=1}^{d}g_{t,i}^2 && (45)\\
&= \|w_t-w^*\|^2+\alpha_t^2\|g_t\|^2. && (46)
\end{aligned}$$

Using Cauchy-Schwartz inequality, we have

$$\begin{aligned}
B &\leq \left(\sum_{i=1}^{d}(w_i^*)^2\right)^{\frac{1}{2}}\cdot\alpha_t\cdot\left(\sum_{i=1}^{d}g_{t,i}^2\right)^{\frac{1}{2}} && (47)\\
&= \|w^*\|\cdot\alpha_t\cdot\|g_t\|. && (48)
\end{aligned}$$

Finally, we have

$$C := \sum_{i=1}^{d}\alpha_t^2\cdot g_{t,i}^2 = \alpha_t^2\|g_t\|^2. \quad (49)$$

Combining all these, we have

$$\mathbb{E}\left[\|r_t\|^2\right] \leq \epsilon\|w_t-w^*\|^2+\epsilon\alpha_t^2\|g_t\|^2+\epsilon\alpha_t\|w^*\|\|g_t\|+\epsilon\alpha_t^2\|g_t\|^2. \quad (50)$$

This completes the proof.

□

D.3 Proof of Theorem 1

Proof. Recall the actual update rule is

$$w_{t+1} = Q(w_t-\alpha_t g_t) = w_t-\alpha g_t+r_t, \quad (51)$$

where $r_t = Q(w_t-\alpha_t g_t)-(w_t-\alpha_t g_t)$ and $Q(\cdot)$ is the stochastic rounding function.

Conditioned on w_t, taking the conditional expectation with respect to g_t, we have

$$\begin{aligned}
\mathbb{E}\left[\|w_{t+1}-w^*\|^2\right] &= \|w_t-w^*\|^2-2\mathbb{E}\left[\langle w_t-w^*,\alpha_t g_t-r_t\rangle\right]+\mathbb{E}\left[\|\alpha_t g_t-r_t\|^2\right] && (52)\\
&= \|w_t-w^*\|^2-2\alpha_t\langle w_t-w^*,\nabla F(w_t)\rangle\\
&\quad +\alpha_t^2\cdot\mathbb{E}\left[\|g_t\|^2\right]+\mathbb{E}\left[\|r_t\|^2\right] && (53)\\
&= \|w_t-w^*\|^2-2\alpha_t\langle w_t-w^*,\nabla F(w_t)\rangle\\
&\quad +\alpha_t^2\cdot\mathbb{E}\left[\|g_t\|^2\right]+\mathbb{E}_{g_t}\left[\mathbb{E}_Q\left[\|r_t\|^2|g_t\right]\right] && (54)\\
&\leq \|w_t-w^*\|^2-2\alpha_t\langle w_t-w^*,\nabla F(w_t)\rangle+\alpha_t^2\mathbb{E}\left[\|g_t\|^2\right]\\
&\quad +\epsilon\cdot\mathbb{E}\left[\|w_t-w^*\|^2+2\alpha_t^2\|g_t\|^2+\alpha_t\|w^*\|\|g_t\|\right] && (55)\\
&\leq (1+\epsilon)\|w_t-w^*\|^2-2\alpha_t\langle w_t-w^*,\nabla F(w_t)\rangle\\
&\quad +(1+2\epsilon)\alpha_t^2G^2+\epsilon\alpha_t G\|w^*\|. && (56)
\end{aligned}$$

In above, to obtain (53), we use the fact that $\mathbb{E}_{g_t}\left[\mathbb{E}_Q\left[\|r_t\||g_t\right]\right] = 0$. To obtain (55), we use Lemma 3. To obtain (56), we use the fact that $\left(\mathbb{E}\left[\|g_t\|\right]\right)^2 \le \mathbb{E}\left[\|g_t\|^2\right]$ and the assumption that $\mathbb{E}\left[\|g_t\|^2\right] \le G^2$.

Now taking the expectation with respect to w_t and using the strong convexity, we have

$$\mathbb{E}\left[\|w_{t+1} - w^*\|^2\right] \le (1+\epsilon-\alpha_t\mu)\mathbb{E}\left[\|w_t - w^*\|^2\right] + 2\alpha_t(F(w^*) - F(w_t)) + (1+2\epsilon)\alpha_t^2 G^2 + \epsilon\alpha_t G\|w^*\|. \tag{57}$$

Dividing both sides by $2\alpha_t$ and rearranging the terms, we have

$$\mathbb{E}\left[F(w_t)\right] - F(w^*) \le \frac{1+\epsilon-\alpha_t\mu}{2\alpha_t}\mathbb{E}\left[\|w_t - w^*\|^2\right] - \frac{1}{2\alpha_t}\mathbb{E}\left[\|w_{t+1} - w^*\|^2\right] + \frac{1+2\epsilon}{2}\alpha_t G^2 + \frac{\epsilon}{2}G\|w^*\|. \tag{58}$$

Letting $\alpha_t = \frac{1+\epsilon t+2\epsilon}{\mu(t+1)}$, we have

$$\mathbb{E}\left[F(w_t)\right] - F(w^*) \le \frac{\mu}{2}\frac{t-\epsilon}{1+\epsilon t+2\epsilon}\mathbb{E}\left[\|w_t - w^*\|^2\right] - \frac{\mu}{2}\frac{t+1}{1+\epsilon t+2\epsilon}\mathbb{E}\left[\|w_{t+1} - w^*\|^2\right] + \frac{1+2\epsilon}{2}\alpha_t G^2 + \frac{\epsilon}{2}G\|w^*\|. \tag{59}$$

In above, let $a(t) := \frac{\mu}{2}\frac{t-\epsilon}{1+\epsilon t+2\epsilon}$ and $b(t) := \frac{\mu}{2}\frac{t+1}{1+\epsilon t+2\epsilon}$. Notice that $a(0) < 0$, and $a(t+1) = \frac{\mu}{2}\frac{t+1-\epsilon}{1+\epsilon t+2\epsilon} < b(t)$. Letting $t = 0, 1, \ldots, T$ and summing up both sides, and dividing both sides by $T+1$, we have

$$\begin{aligned}
\mathbb{E}\left[F(\bar{w})\right] - F(w^*) &\le \frac{1}{T+1}\sum_{t=0}^{T}\mathbb{E}\left[F(w_t)\right] - F(w^*) && (60)\\
&\le \frac{1+2\epsilon}{2}\frac{\sum_{t=0}^{T}\alpha_t}{T+1}G^2 + \frac{\epsilon}{2}G\|w^*\| && (61)\\
&= \frac{(1+2\epsilon)G^2}{2\mu(T+1)}\sum_{t=0}^{T}\left(\frac{1+\epsilon}{t+1} - \epsilon\right) + \frac{\epsilon}{2}G\|w^*\| && (62)\\
&< \frac{(1+2\epsilon)^2 G^2}{2\mu}\frac{1+\log(T+1)}{T+1} + \frac{\epsilon}{2}G\|w^*\|, && (63)
\end{aligned}$$

where $\bar{w} = \frac{1}{T+1}\sum_{t=0}^{T} w_t$.

□