A Bag of Tricks for Scaling CPU-based Deep FFMs to more than 300m Predictions per Second

Blaž Škrlj[1,*], Benjamin Ben-Shalom[1], Grega Gašperšič[1], Adi Schwartz[1], Ramzi Hoseisi[1], Naama Ziporin[1], Davorin Kopič[1] and Andraž Tori[1]

[1]Outbrain Inc.

Abstract

Field-aware Factorization Machines (FFMs) have emerged as a powerful model for click-through rate prediction, particularly excelling in capturing complex feature interactions. In this work, we present an in-depth analysis of our in-house, Rust-based Deep FFM implementation, and detail its deployment on a CPU-only, multi-data-center scale. We overview key optimizations devised for both training and inference, demonstrated by previously unpublished benchmark results in efficient model search and online training. Further, we detail an in-house weight quantization that resulted in more than an order of magnitude reduction in bandwidth footprint related to weight transfers across data-centres. We disclose the engine and associated techniques under an open-source license to contribute to the broader machine learning community. This paper showcases one of the first successful CPU-only deployments of Deep FFMs at such scale, marking a significant stride in practical, low-footprint click-through rate prediction methodologies.

Keywords

Data Stream Mining, Factorization Machines, Online Learning, Scalable Machine Learning

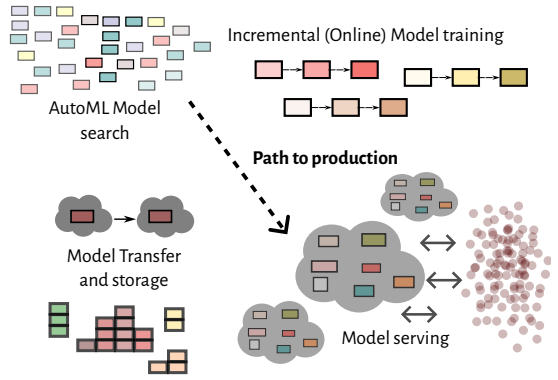

Figure 1: Overview of the key topics discussed in this paper. Performance optimizations that span model search (AutoML), online model training, storage, transfer and serving are discussed.

1. Introduction

Design and development of machine learning approaches for the domain of *recommendation systems* revolves around the interplay between scalability and approximation capability of classification and regression algorithms. Currently, many deployed recommendation engines rely on factorization machine-based approaches; this is mostly due to good trade-offs when it comes to scalability, maintainability and data scientists' involvement in building such models. Even though contemporary recommenders started to increasingly rely on language model-based techniques [1], utilizing factorization machines remains *de facto* solution for large-scale "screening" of candidates that are to be served. Such candidates can include from unseen items (online stores), to movie recommendations, to ads [2, 3]. Scalability of factorization machines enables creation of real-time systems that handle hundreds of millions of requests in predictable and maintainable manner. In recent years, two main branches of methods have emerged. Approaches based on frameworks such as TensorFlow [4] and PyTorch [5] enabled construction of highly expressive architectures that often require specialized hardware for efficient productization [6, 7, 8, 9]. CPU-only, single instance – single pass alternatives are fewer, and revolve around highly optimized C++ or Rust-based approaches that exploit consumer hardware as much as possible. The latter is the main focus of this paper (overview in Figure 1).

2. Fwumious Wabbit (FW) - an overview

We proceed with a discussion of Fwumious Wabbit (FW), an in-house, Rust-based factorization machine-based system currently used in production for large-scale recommendation[1].

2.1. Origins of FW and Vowpal Wabbit (VW)

The FW derives from Vowpal Wabbit (VW) [10], a high-performance, scalable open-source ML system recognized for its efficiency on large datasets [2]. While VW primarily uses logistic regression for tasks like click-through rate prediction, it lacks readily available advanced extensions found in the domain of factorization machines. One of the more expressive variations of factorization machines are the Field-aware Factorization Machines (FFMs), described in detail in the works of Juan et al. [11, 12]. Building on this foundation, we enhanced the FFM architecture by integrating elements of deep learning. Specifically, a multi-layer perceptron (MLP)-like structure in conjunction with the traditional FFM (and logistic regression) components. The architecture's computational complexity, a notable challenge, contributes to its rarity in existing benchmarks. When implemented in standard frameworks like TensorFlow, the architecture struggles to scale effectively for practical use.

Despite these challenges, our deep learning-extended FFM method demonstrated significant performance gains over other tested algorithms in internal assessments. However, scaling this method was not straightforward. It was

AdKDD Workshop 2024
*Corresponding author.
bskrlj@outbrain.com (B. Škrlj)
0000-0002-9916-8756 (B. Škrlj)

[1]The engine with main implementations discussed in this paper is freely available as https://github.com/outbrain/fwumious_wabbit.
[2]https://vowpalwabbit.org/

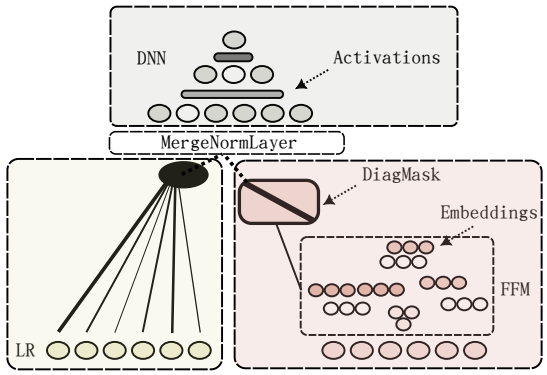

Figure 2: Architecture of implemented CPU-based DeepFFMs. Main blocks are the neural network (gray), logistic (yellow) and FFM (red) ones.

only through invoking BLAS [13], that we achieved critical performance enhancements, allowing for practical full-scale deployment[3]. An overview of the architecture is shown in Figure 2. . Key parts of the architecture are

$$\text{LR}(w,x) = \sum_{j}^{n} w_j \cdot x_j + b; \text{FFM}(w,x) = \sum_{j_1=1}^{n} \sum_{j_2=j_1+1}^{n} (w_{j_1,f_2} \cdot w_{j_2,f_1}) \cdot x_{j_1} x_{j_2}.$$

Neural part (matrix form),

$$\text{FFNN}(\mathbf{W}_{1,2,\ldots,n}, \mathbf{X}) = a_n(\ldots a_2(a_1(\mathbf{X} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2)\ldots) \cdot \mathbf{W}_n,$$

takes as input both FFM and LR's outputs, i.e.

$$\text{DFFM}(\mathbf{W}_{1,2,\ldots,n}, \mathbf{w}_b, \mathbf{w}_c, \mathbf{x}) = \text{FFNN}(\mathbf{W}_{1,2,\ldots,n}, \mathit{MergeNormLayer}(\text{LR}(\mathbf{w}_b, x), \mathit{DiagMask}(\text{FFM}(\mathbf{w}_c, x))).$$

Here, *MergeNormLayer* represents the operator that combines outputs of FFM and LR parts and applies normalization. Further, *DiagMask* represents diagonal mask of FFM space, inducing half smaller number of combinations requiring down-stream processing[4].

2.2. Criteo, Avazu and KDD2012 - a benchmark and stability analysis

Even though we evaluated FW extensively on internal data sets (and online, in A/B tests), where it showed consistent dominance, results on published data sets such as Criteo are also of relevance for dissemination of engines' behavior and overall performance. In this section we overview a benchmark we conducted to assess general behavior of VW and FW. We also implemented DCNv2 [14, 15], a Tensorflow-based strong baseline[5]. For considered data sets (Criteo[6], Avazu[7] and KDD2012[8]), log transform of continuous features was conducted and no additional data pruning (rare values etc.) was conducted (as is done in our system)[9]. The hyperparameters considered include power of t, learning rates for different types of blocks (ffm, lr), regularization amount (L2 norm, VW). For DCNv2 we considered different learning rates, cross layer numbers, dropout rates and beta parameters. Results of the benchmark are summarized in Figure 3. For each data set, algorithms considered are visualized as AUC scores computed in a rolling window of 30k instances[10].

The trace in each plot represents the average performance (95% CI), and light-gray regions represent model evaluations that were out-of-distribution – this aspect is particularly relevant for understanding **stability** of different approaches and their sensitivity to hyperparameter configurations. For example, we observed that adding deep layers to VW models in most cases resulted in worse performance. Carefully tuned VW hyperparameters yielded sufficient performance, however, indicate potentially cumbersome model search (when considering new use cases/data) in practice. Similar behavior was observed for DCNv2. The dotted black lines represent the overall best single-window performance, and performance on a given data set's test set[11] Overall, initial phases of learning revealed VW's capability to adapt with less data, the DeepFFMs dominate after enough data is seen by the engines. Superior performance was observed by DCNv2 on Criteo, yet not other data sets (all features considered). The benchmark demonstrates that progressively more complex architectures tend to result in better modeling capabilities, and with them, better AUCs in this benchmark. In terms of **runtime**, on the same hardware, Criteo data set could be processed on average in 32min by VW, and 31min by FW (linear model vs. DeepFFM). Deep VW variations took substantially longer, around 65min on average (batch size of 2k). This result indicates that FW enables more powerful models with same time bounds for training. The DCNv2 (CPU) baseline was 30%-50% slower compared to DeepFFM runs. These tatistics were obtained based on tens of thousands of runs that represented different algorithm configurations (both hyperparameters and field specifications). Being CPU-based, the described approaches enable seamless scaling to commodity hardware, resulting in lower training and inference costs in practice.

3. FW in practice: Service Architecture overview

This section aims to facilitate understanding of subsequently discussed optimizations that were put in place to enable scaling of Deep FFMs. The implemented FW contains both training and inference logic. The training logic is relevant for incrementally training **more than a hundred models**, online, every *n* minutes (depends on the model). Training jobs are separate deployments that automatically query for relevant chunks of data, download, update based on existing weights and send the weights to the serving layer. Serving layer on-the-fly reconstructs the final inference weights via a patching mechanism discussed in Section 6, and exposes the weights as part of the serving service that handles millions of requests with new data. Based on the effect of predictions, data is streamed back to the system as training

[3] https://github.com/outbrain/fwumious_wabbit/blob/main/src/block_neural.rs

[4] See https://github.com/outbrain/fwumious_wabbit/blob/main/src/regressor.rs for more details.

[5] Unique hash was assigned to each value for this baseline for ease of implementation.

[6] https://www.kaggle.com/c/criteo-display-ad-challenge

[7] https://www.kaggle.com/c/avazu-ctr-prediction/data

[8] https://www.kaggle.com/c/kddcup2012-track2

[9] Such minimal pre-processing is within reach of a regular production.

[10] RIG and Log-loss scores are aligned with AUC-based results, hence only these are reported for readability purposes

[11] for KDD, we took last 2m instances to capture apparent variability in data better, other data sets are split as reported in their origin publications.

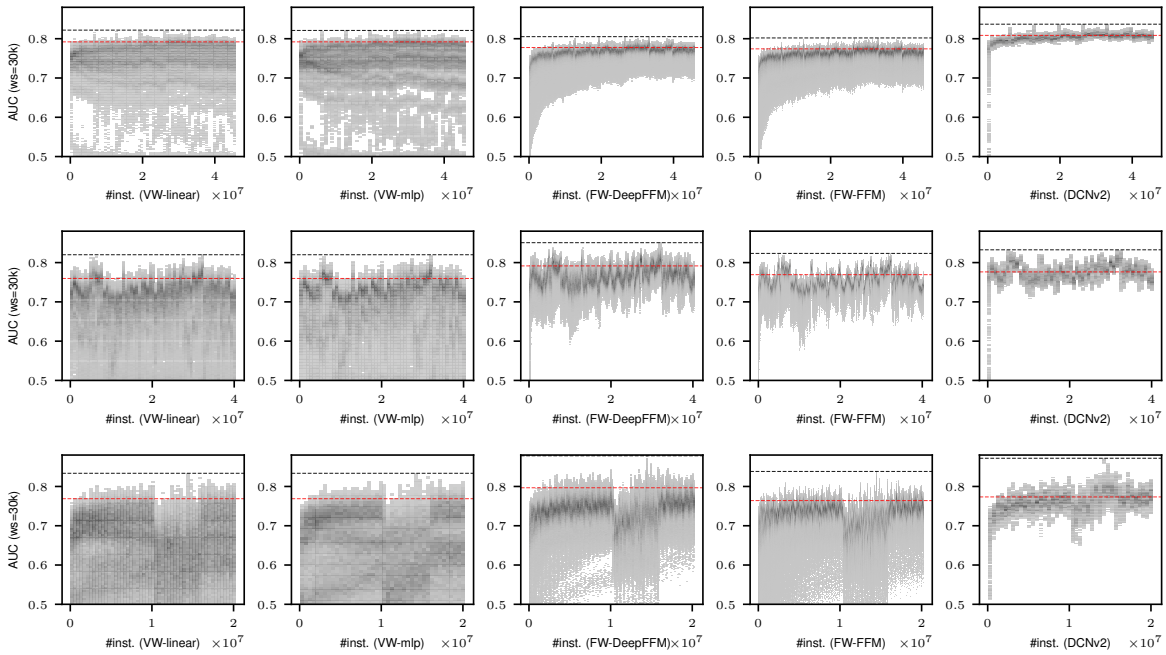

Figure 3: Visualization of overall performance of different algorithms (single-pass) across different benchmark data sets (top-down: Criteo, Avazu, kddcup2012. Visualizations show traces of all trained models (per engine).

Table 1
Stability analysis and overall performance. Rows with max test set performance highlighted.

Avazu (window=30k)

algo	avg	median	max	std	min	test
VW-linear	0.6832	0.7016	0.8200	0.0668	0.4664	0.7596
VW-mlp	0.6755	0.6984	0.8200	0.0748	0.4664	0.7596
FW-DeepFFM	0.7648	0.7654	0.8507	0.0243	0.4764	0.7916
FW-FFM	0.7524	0.7524	0.8234	0.0227	0.4816	0.7693
DCNv2	0.7750	0.7745	0.8326	0.0202	0.5005	0.7763

Criteo (window=30k)

algo	avg	median	max	std	min	test
VW-linear	0.7340	0.7460	0.8219	0.0556	0.4768	0.7920
VW-mlp	0.7247	0.7425	0.8211	0.0670	0.4768	0.7920
FW-DeepFFM	0.7655	0.7689	0.8053	0.0179	0.4796	0.7803
FW-FFM	0.7578	0.7621	0.8020	0.0198	0.4682	0.7742
DCNv2	0.8042	0.8052	0.8370	0.0118	0.4958	0.8085

KDDCup2012 (window=30k)

algo	avg	median	max	std	min	test
VW-linear	0.6333	0.6419	0.8336	0.0807	0.3430	0.7688
VW-mlp	0.6309	0.6402	0.8336	0.0869	0.3759	0.7688
FW-DeepFFM	0.7323	0.7400	0.8781	0.0414	0.3687	0.7967
FW-FFM	0.7228	0.7318	0.8382	0.0391	0.3651	0.7641
DCNv2	0.7589	0.7610	0.8718	0.0301	0.4792	0.7734

data (a feedback loop). The training jobs are Python-based services that interact with the binary via process invocations. Serving binds the inference capabilities with the serving (Java) service directly via a foreign function interface (ffi)[12]. The architecture enables separation of concerns – training jobs are separate to inference jobs, albeit at the cost of needing to send the updated weight data between services; this is one of the key performance bottlenecks that was addressed in this work. An overview of the scope of this paper is shown in Figure 1.

4. Model training improvements

We next discuss main improvements implemented at the level of training jobs and offline research.

4.1. Speeding up model warm-up phase

Model warm-up corresponds to a phase in model training where model starts with past data, and "catches up" with present data as fast as possible. We identified efficient data pre-fetching as a crucial optimization for speeding up this process. By implementing async learning cycles, multiple rounds of "future" data can be downloaded upfront, making sure the learning engine has constant influx of data. Data pre-fetch in practice results in up to **4x faster pre-warming**. Within the cloud environment where the jobs are deployed, we can control machine "taints", i.e. signatures that determine their hardware profile. Pre-warm jobs have dedicated taints, which in practice results in machines that are newer and stronger.

4.2. Hogwild-based training

An optimization that significantly improved model pre-warm time is the previously reported Hogwild-based model training[16], implemented also for Fwumious framework (as part of this work). Here, weight overlaps/overrides are allowed as the trade off for multi-threaded updates. By tuning Hogwild capacity to tainted machines, we observed multi-fold speedups in model warm-up. In practice, the times for bigger models went from multiple weeks to days, and in most cases around a day of training (to catch up). Weight degradation due to Hogwild was A/B tested and does not appear to cause any noticeable RPM drops. Summary of Howgild-based training compared to control (no such training) is shown in Table 2. Utilization of hogwild has shown substantial benefits also when utilized during online training (e.g., every 5min), and enabled of scaling of 100% bigger models. To the best of our knowledge, this is one of the first demonstrations of consistent Hogwild-based training improvements for Deep FFMs.

[12]https://github.com/outbrain/fwumious_wabbit/blob/main/src/lib.rs

Table 2
Impact of Hogwild-based training.

Implementation	Warmup time (same period)
FW-deepFFM-control	8d
FW-deepFFM-hogwild	23h (48 threads)
Implementation	Online training (same period)
FW-deepFFM-control	20m
FW-deepFFM-hogwild	4m (4 threads)

4.3. Sparse weight updates

The next discussed optimization is related to how gradients are accounted for during model optimization itself. We observed that deep layers, albeit being parameter-wise in minority compared to FFM part, take up considerable amount of time during optimization. To remedy this shortcoming, we identified an optimization opportunity that is a combination of activation function used in most models, $f(x) = \max(x, 0)$, and the specific implementation of FW. By realizing that we can identify *zero global gradient* scenarios upfront, prior to updating any weights, we could skip whole branches of computation with no impact on learning. The performance (speed) of training however, was across-the-board improved by 30% for most models, and for deeper ones by **up to 3x**, see Table 3 for more details. We observed that at most two hidden layers were feasible for production, hence any further speedups than observed 30% were not feasible in practice. This optimization was possible due to ReLU's nature; this activation maps weights to zeros, effectively enabling identification of compute branches that need to be skipped during updates.

5. Model serving improvements

We proceed our discussion with an overview of CPU-based model inference via **context caching**. A considerable optimization we observed could take place in our system is *context caching*. Each request can be separated into context and candidates. For all candidates in the request, the context is the same, even though the recommended content's features differ – this implies part of the feature space is very consistent for each candidate batch. To exploit this property, a dedicated serving-level caching scheme was put in place. FW at this point does an additional pass only with the context part, where it identifies and caches frequent parts of the context. On subsequent candidate passes it reuses this information on-the fly instead of re-calculating it for each context-candidate pair. Deployment impact of context caching is shown in Figure 4[13]. We next discuss **(SIMD) Instruction-aware forward pass**. Another optimization that is particular to inference is proper exploitation of SIMD intrinsics. These hardware instruction level optimizations, however, needed to be carefully implemented as the space of serving hardware is not homogeneous, meaning that on-the-fly instruction detection, and subsequent utilization of appropriate binary needed to be put in place. **SIMD intrinsics** were successfully used to speed up forward pass

[13] https://github.com/outbrain/fwumious_wabbit/blob/main/src/radix_tree.rs

Table 3
Speedups observed due to sparse weight updates.

#Hidden layers	1	2	3	4
Speedup (sparse updates)	1.3x	1.8x	2.4x	3.5x

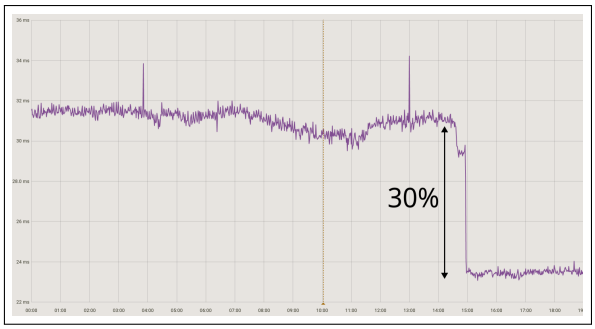

Figure 4: Impact of context caching on inference time.

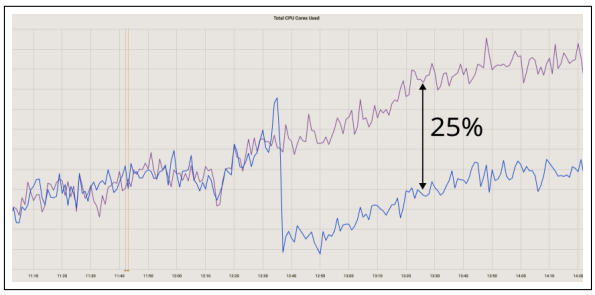

Figure 5: Relative impact of SIMD-enabled (blue, after drop) vs. SIMD-disabled (purple) FW in production (inference).

(inference) with no loss in RPM performance, and resulted in a consistent 20% speedup for all serving[14]. Real-life example of deployed SIMD-based FW vs. the control (no SIMD) is shown in Figure 5. Up to 25% faster inference (and with it lower resource utilization) were observed.

6. Storage and transfer optimization

As discussed in previous sections, training and serving jobs are separated. This separation of concerns, albeit easier to maintain, contributes to a major drawback: weight sending across the network. Model weights need to be constantly updated, which incurs substantial bandwidth costs. For example, **hundreds of live models** that take up to 10G of memory (per update) are constantly transferred across the network, resulting in a substantial bandwidth overhead to ensure low-latency online serving.

Model patching. The first improvement we implemented is the concept of *model patching*. This process is inspired by application of software patches (in general), albeit tailored to internal structure of FW's weights. Each trained model consists of training weights and the optimizer's weights. The latter are not required for actual inference, which immediately reduces the required space by half. Further, each subsequent inference weights update (inference weights can be multiple GB) first computes *model diff* – byte-level difference between old and new weights. This is possible due to a consistent memory-level structure of weight files. The diffs are compressed, sent to the serving layer, unpacked and applied to previous weights file to obtain the new set of weights (inference). This process takes tens of seconds, however, further reduces memory footprint on the network by more than 100% (less than a GB of updates per model after patching Deep FFMs).

First, instead of storing absolute indices of bytes that

[14] https://github.com/outbrain/fwumious_wabbit/blob/main/src/block_ffm.rs

change, *relative* locations are stored, resulting in a considerable storage saving. Next, small integers denoting these differences are stored as a custom integer type – instead of storing whole ints, compressed versions (small ints are impacted the most) are stored, leading to further improvements[15]. As patcher works at the level of bytes, we also successfully tested it for internal Tensorflow-based flows (reduced bandwidth for sending models). **Weight Quantization**. Inspired by recent weight quantization advancements in the field of large language models [17, 18], we implemented a **variation of 16b weight quantization** that, when combined with the byte-level patching mechanism, offered considerable bandwidth and model storage improvements. The quantization algorithm was designed to account for the following use-case specific properties. First, by ensuring consistently small weight patches, the quantization ensures consistently smaller network load. Second, the quantization and **dequantization** procedures must be fast, as they need to happen within a designated time window after each training round (procedure has tens of seconds at most at its disposal for full weight space). Finally, the algorithm needs to be able to dynamically select viable weight ranges, as we observed considerable variation in weight update sizes based on e.g., time of the day (traffic amount). The final version of the algorithm can be summarized as follows. For each online model update (e.g., **5min window**), weights are first traversed to obtain the minimum and maximum values (weights). These statistics are required to dynamically determine the range of relevant weight bins, as the amount of possible values for 16b representation is small (around 65k). Let $W = \{w_1, w_2, \dots, w_n | w_i \in \mathbb{R}\}$ denote the set of all (n) weights and $b_{\max}$ denote the number of possible weight buckets. Once the minimum and maximum are obtained, the bucket size is computed as

$$\text{BUCKET}_s = \frac{\max(W).\text{round}(\alpha) - \min(W).\text{round}(\beta)}{b_{\max}}.$$

Note that minimum and maximum are *rounded* to α and β decimals. This consideration stems from empirical results that indicated that considering full precision bounds results in less stable patch sizes [16]. When constraining minimum and maximum to certain precision, behavior stabilized whilst preserving performance and online behavior. In the second pass, weights are **quantized** – for each weight, its 16b representation is computed and stored. This results in computing

$$((w_i - \min(W)/\text{BUCKET}_s).\text{round}().\text{castTo16b}().\text{convertToBytes}(),$$

i.e. a set of bytes that represent a certain weight bucket. Bytes are stored in FW weight format and re-used during inference. An important detail also concerns *metadata* required to perform this type of quantization; the original weights file is enriched with a header that contains the bucket size and weight minimum – these two properties are sufficient for efficient weight reconstruction when/where relevant[17]. Results on a representative CTR model are shown in Table 4. Metrics of interest are time to produce patch and the final patch/weight update's size. Patching and quantization result in up to 30x smaller model updates.

[15] https://github.com/outbrain/fwumious_wabbit/blob/main/weight_patcher

[16] (quantization output tended to fluctuate more)

[17] https://github.com/outbrain/fwumious_wabbit/blob/main/src/quantization.rs

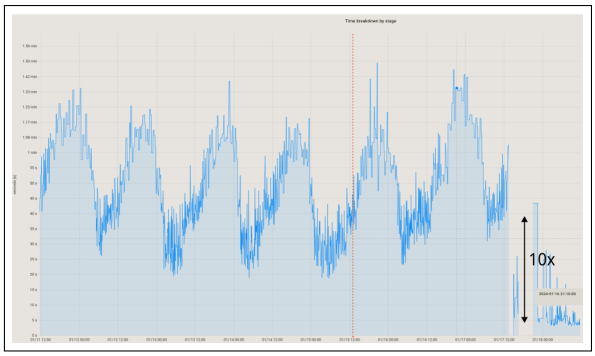

Figure 6: Speedup observed when jointly using quantization and model patching (as opposed to just patching).

Table 4
Impact of model quantization on the global production CTR model.

Weight processing	Avg. time spent	Update file size
no procecssing (baseline)	/	100%
fw-quantization	2s	50%
fw-patcher	45s	30±5%
fw-patcher + fw-quantization	**8s**	**3±2%**

Note that weight patching and quantization on their own already at least halve the size of weights that are used in serving and production. Further, by combining the two approaches, we observed a non-linear improvement in patch sizes – around **10x smaller updates** are regularly produced. The quantized patches-based model showed small lifts in and online A/B against control with no quantization applied, considerably reducing network bandwidth required with a small positive business impact (+0.15% RPM). Speedup in a real-life production system due to compound effect of quantization and patching can be observed in Figure 6. Rightmost part of the plot represents total time spent patching and computing quantized weights.

7. Conclusions and open problems

In this paper, we presented a collection of implementation details for scaling CPU-based DeepFFMs to operate at a multi-data-center scale, capable of handling hundreds of millions of predictions per second. We delved into both the offline and online components of our system. In the offline phase, we covered the complete workflow, including model architecture, enhancements to system warm-up processes, and bandwidth optimization strategies. Within the online phase, we describe two novel modifications to the inference layer that have yielded significant speed improvements. Our main algorithms, concepts, and performance benchmarks were discussed in detail, open-source implementations of key components were made freely available. The implementation is extensible to other FFM-based variants. As further work, on the inference side, implementing quantization techniques could accelerate the forward pass by using integer-based operations [19]. Improved weight sharing and memory mapping could offer training improvements.

References

[1] J. Zhang, K. Bao, Y. Zhang, W. Wang, F. Feng, X. He, Is chatgpt fair for recommendation? evaluating fairness in large language model recommendation, in: Proceed-

ings of the 17th ACM Conference on Recommender Systems, 2023, pp. 993–999.

[2] S. Zhang, Y. Tay, L. Yao, A. Sun, C. Zhang, Deep learning for recommender systems, in: Recommender Systems Handbook, Springer, 2021, pp. 173–210.

[3] Y. Deldjoo, M. Schedl, P. Cremonesi, G. Pasi, Recommender systems leveraging multimedia content, ACM Computing Surveys (CSUR) 53 (2020) 1–38.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/, software available from tensorflow.org.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[6] W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, M. Zhang, J. Tang, Autoint: Automatic feature interaction learning via self-attentive neural networks, in: Proceedings of the 28th ACM international conference on information and knowledge management, 2019, pp. 1161–1170.

[7] J. Lian, X. Zhou, F. Zhang, Z. Chen, X. Xie, G. Sun, xdeepfm: Combining explicit and implicit feature interactions for recommender systems, in: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining, 2018, pp. 1754–1763.

[8] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, et al., Wide & deep learning for recommender systems, in: Proceedings of the 1st workshop on deep learning for recommender systems, 2016, pp. 7–10.

[9] H. Guo, R. Tang, Y. Ye, Z. Li, X. He, Deepfm: a factorization-machine based neural network for ctr prediction, arXiv preprint arXiv:1703.04247 (2017).

[10] A. Bietti, A. Agarwal, J. Langford, A contextual bandit bake-off, arXiv:1802.04064v3 [stat.ML], 2018. URL: https://www.microsoft.com/en-us/research/publication/a-contextual-bandit-bake-off-2/.

[11] Y. Juan, D. Lefortier, O. Chapelle, Field-aware factorization machines in a real-world online advertising system, in: Proceedings of the 26th International Conference on World Wide Web Companion, 2017, pp. 680–688.

[12] Y. Juan, Y. Zhuang, W.-S. Chin, C.-J. Lin, Field-aware factorization machines for ctr prediction, in: Proceedings of the 10th ACM conference on recommender systems, 2016, pp. 43–50.

[13] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al., An updated set of basic linear algebra subprograms (blas), ACM Transactions on Mathematical Software 28 (2002) 135–151.

[14] R. Wang, R. Shivanna, D. Cheng, S. Jain, D. Lin, L. Hong, E. Chi, Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems, in: Proceedings of the web conference 2021, 2021, pp. 1785–1797.

[15] W. Shen, Deepctr: Easy-to-use,modular and extendible package of deep-learning based ctr models, https://github.com/shenweichen/deepctr, 2017.

[16] B. Recht, C. Re, S. Wright, F. Niu, Hogwild!: A lock-free approach to parallelizing stochastic gradient descent, Advances in neural information processing systems 24 (2011).

[17] B. Rokh, A. Azarpeyvand, A. Khanteymoori, A comprehensive survey on model quantization for deep neural networks, arXiv preprint arXiv:2205.07877 (2022).

[18] H. Bai, L. Hou, L. Shang, X. Jiang, I. King, M. R. Lyu, Towards efficient post-training quantization of pre-trained language models, Advances in Neural Information Processing Systems 35 (2022) 1405–1418.

[19] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 2704–2713.