Factorized Deep Retrieval and Distributed TensorFlow Serving

Xinyang Yi, Yi-Fan Chen, Sukriti Ramesh, Vinu Rajashekhar, Lichan Hong, Noah Fiedel,
Nandini Seshadri, Lukasz Heldt, Xiang Wu, Ed H. Chi
Google Inc.
{xinyang,yifanchen,sukritiramesh,vinuraja,lichan,nfiedel,nands,heldt,wuxiang,edchi}@google.com

ABSTRACT

Many systems need to retrieve and score items from a huge catalog, such as recommender and search ranking systems. We call this the *deep retrieval* problem. In this paper, we focus on the design and implementation of a large-scale deep retrieval system for catalogs with hundreds of millions of items, utilizing three critical components: (1) distributed matrix factorization trainer, (2) approximate dot-products with hashing techniques, and (3) distributed serving in TensorFlow. This factorized deep retrieval system has been productionized at Google for a terabyte-byte-sized model, and is highly scalable, low-lantency, and adapts to long-tail and fresh content. Furthermore, the distributed serving capability is general and can serve other large-scale models in the future.

1 INTRODUCTION

Large scale machine-learned information retrieval, a.k.a., *deep retrieval*, systems have been widely used to support applications involved with query and item matching problems, e.g., search ranking and recommender systems. In production applications, these systems are typically required to serve real-time retrieval of hundreds of millions of items, and meanwhile, provide a good scoring function that can perform well not only for head items but also long-tail and fresh items. Matrix factorization (MF) is one of the most widely adopted machine-learning techniques for production retrieval systems. The output of MF consists of a large number of embeddings for query and items. Learning large sets of embeddings commonly suffers from folding [13], where spurious discoveries happen due to the subspace interleaving of unrelated items because of the scarcity of explicit negatives. The learning algorithm thus needs to be efficient in terms of training speed and effective in terms of addressing implicit feedback. The last modeling challenge, which is acute for applications where fresh content arrives in streaming fashion, is cold-starting queries and items without observed interactions.

Building a production-scale, efficient machine-learning serving system is non-trivial, and it becomes more challenging in the case of serving large embedding factors. The standard practice to host a model on a single machine is not scalable. For example, one of our production recommenders requires over 1TB RAM to load over 700 million embeddings. Solving this requires distributed model serving including complex capabilities such as serving-specific model conversion, auto-scaling, etc., as discussed in Section 4.

In this paper, by bridging together modeling and system thinking, we present a TensorFlow [1] based factorized deep retrieval platform. We developed this platform to generally serve production class deep retrieval. Our contributions are: 1) We extended the classic WALS factorization [6] to a hybrid approach where features and items are co-embedded into the same low-dimensional space, for addressing cold-start issues; 2) We developed a TensorFlow distributed WALS factorization trainer that can efficiently factorize a matrix with more than a billion rows and columns; 3) We designed and implemented a distributed serving system based on TensorFlow Serving [8]. We deployed the resulting 1.2 TB factorization model as a new candidate generator in a large-scale production recommendation system, improving the main platform metric significantly, while serving the first distributed (and largest to date) model with TensorFlow Serving.

2 PROBLEM AND MODELING

We consider a standard setup for retrieval problems where we have N queries and M items. Let $\Omega \in [N] \times M$ be a set of query-item pairs observed, and $l_{(i,j)}$ be the observed labels for $(i, j) \in \Omega$. Let $P \in \mathbb{R}^{N \times M}$ be the sparse matrix containing $l_{i,j}$. By matrix factorization, we aim to compute d-dimensional query and item embeddings $U \in \mathbb{R}^{N \times d}, V \in \mathbb{R}^{M \times d}$ such that $UV^T \sim P$. To address the aforementioned cold-start issues, we also consider the content features of both queries and items, which are represented by matrices $\Phi \in \{0,1\}^{N \times N_f}, \Psi \in \{0,1\}^{M \times M_f}$ [1]. Similarly, let Ω_1, Ω_2 denote the sets of the non-zero in Φ, Ψ. We aim to learn U, V together with feature embeddings $U_f \in \mathbb{R}^{N_f \times d}$, $V_f \in \mathbb{R}^{M_f \times d}$ by the loss [2]:

$$\mathcal{L}(U,V,U_f,V_f) = \|(UV^T - P)_\Omega\|_F^2 + \|(UU_f^T - \Phi)_{\Omega_1}\|_F^2 + \|(VV_f^T - \Psi)_{\Omega_2}\|_F^2$$
$$+\sigma\left(\|(UV^T)_{\Omega^c}\|_F^2 + \|(UU_f^T)_{\Omega_1^c}\|_F^2 + \|(VV_f^T)_{\Omega_2^c}\|_F^2\right).$$

The σ is a hyper-parameter used to capture the impact of all implicit negatives, and thus avoid folding. The concatenations

$$U^+ = \begin{bmatrix} U \\ V_f \end{bmatrix},\ V^+ = \begin{bmatrix} V \\ U_f \end{bmatrix},\ P^+ = \begin{bmatrix} P & \Phi \\ \Psi^T & \mathbf{0} \end{bmatrix},$$

then allow us to solve $\mathcal{L}(U^+, V^+)$ by applying Algorithm 1 directly.

Algorithm 1 WALS factorization

```
1: Input: Sparse matrix $P \in \mathbb{R}^{N \times M}$ with support $\Omega$, element-wise
   weight matrix with $W \in \mathbb{R}^{N \times M}$ with support $\Omega$, unobserverd
   weight $\sigma$, regularization $\lambda$, number of iterations $T$.
2: Loss: $\mathcal{L}(U,V) = \|[W \odot (UV^T - P)]_\Omega\|_F^2 + \sigma\|(UV^T)_{\Omega^c}\|_F^2 + \lambda(\|U\|_F^2 + \|V\|_F^2)$.
3: For $t = 0, 1, \ldots, T$
       $U^{(t)} \leftarrow \mathrm{argmin}_U\ \mathcal{L}(U, V^{(t-1)}), V^{(t)} \leftarrow \mathrm{argmin}_V\ \mathcal{L}(U^{(t)}, V)$.   (1)
```

WALS projection. A critical step in inference, for solving the cold-start issue, is *projection* that maps new queries and items into embeddings by using the content features and labels, if any. The projection is done by applying one step of WALS iteration on either rows or columns shown in (1).

[1]We assume all features have discrete values. Hence N_f, M_f denote the overall cardinalities. We set $\Phi_{i,k} = 1$, if query i has the k'th feature.

[2]To ease notation, we omit the element-wise weights on observed entries, and the ℓ_2 regularization on embeddings. See Algorithm 1 for a full WALS loss.

Fast similarity search. Low-latency online retrieval is based on a highly efficient similarity search system built on hashing techniques, e.g., [2, 4, 7, 9], for approximate maximum inner product search (MIPS) problems. Specifically, compact representation of high dimensional embeddings are built through quantization [5] and end-to-end learning of coarse and product quantizers [12].

3 DISTRIBUTED FACTORIZATION

We developed a WALS factorization training framework on TensorFlow (CPU based) [3] capable of factorizing large matrices with scale orders of magnitude greater than those reported in previous works [10, 11]. In one case, a $500M \times 500M$ matrix with 125 Billion observed entries were factorized, using 10 iterations, to 500-dimensional embeddings within 24 hours[3].

The training framework is designed to be highly scalable, strictly synchronized (to facilitate the alternating minimization procedure in WALS) and fault tolerant (facilitating the use of batch resources). The synchronization and recovery mechanisms can potentially serve other learning algorithms requiring strict synchronization.

Strict Synchronization Control. To implement the ALS algorithm exactly, the strict orchestration of switching of the sides of operation is needed. A synchronization control mechanism was developed on TensorFlow using simple communicating protocols between chief and workers to coordinate. The chief is responsible for observing the overall work progress and communicating with workers through the possession of *run-tokens*. Run-tokens determine whether a worker can process work or has halted. The global state is maintained using TensorFlow variables on parameter servers. The state consistency is complicated by the requirement of fault tolerance but we were able to achieve this entirely using existing TensorFlow components.

Fault Tolerance. It is important to allow training to recover from a previous state, if interrupted. The state variables used for strict synchronization mentioned previously require consistency, while native TensorFlow checkpointing is nonatomic and variable updating is nontransactional. We have designed the framework to orchestrate the synchronization of all workers before checkpointing models, and address these special challenges effectively to guarantee consistent checkpointed state.

Factorization Operations. We perform the factor updates by collecting alternating side factors on-the-fly. There are no constraints on the ordering in which the factors are updated. This allows PS tasks to be dynamically updated by workers where the sparse matrix rows/columns are streamed in. The efficient asynchronous reading of inputs and variables between parameter servers and workers yields high CPU utilization on factorization kernels.

4 DISTRIBUTED TENSORFLOW SERVING

To serve the factorization model in Section 1 requiring 1.2 TB peak memory in production, we rely on TensorFlow Serving [8], a flexible, high-performance serving system for machine-learning models, designed for production environments.

[3]Our trainer used 400 workers (each with 16 CPUs and 35 GB RAM) and 100 parameter servers (PS, each with 10 CPU and 80 GB RAM) running on batch resources where some workers are allowed to be preempted during training.

4.1 Remote SessionRun TensorFlow Op

TensorFlow graphs are executed by running a session, providing inputs and fetching specific outputs. To coordinate execution across TensorFlow graphs at serving time, we built a TensorFlow op called the Remote SessionRun Op (RSO), that can make remote calls to execute a TensorFlow graph hosted on other server(s). This is functionally equivalent to inlining one graph into another, but the actual computation is performed remotely.

Each instantiation of the op can be configured with relevant attributes, inputs and outputs. Op attributes are specified statically at model definition and export time, and include target address of the remote server, RPC failure handling semantics, etc. Op inputs include model-name, details about input tensors and output tensor names. Op outputs include the output tensors and the status, for error propagation, if needed.

Building the Export. The model-export is saved as a TensorFlow SavedModel. The WALS projection and assembling of sharded nearest items is saved as a single root model that includes RSO instantiations. The shards for fast embedding search are saved as n sub-models. To manage the multiple SavedModels atomically, the export code relies on a versioned directory structure with a specific sub-directory naming convention for the shards.

4.2 Serving System

The serving system is set up such that multiple servers load one or more sub-models of the distributed model. Communication between servers is performed using RSO instances (Section 4.1) that spawn RPC calls to TensorFlow graphs hosted on different servers.

Orchestration. An orchestrator job is set up to coordinate loading and availability management for the multiple sub-models, including first loading n sub-models, followed by the root model. Once the root model is available, the distributed model can begin receiving requests. Upon receiving a query, the root model execution begins, resulting in RPC calls to n sub-models, each of which respond with the set of candidates. The root model then performs the aggregation and returns the RPC response to the client. For the safe and deterministic execution of a distributed model, it is required that the model has a single point of entry for user queries – in this case, the root graph, which processes the queries, spawns RPC calls as needed, aggregates responses, and enforces error handling semantics based on the use-case.

Context Propagation. For distributed serving, it is imperative to propagate context relevant to the remote execution. For the instantiated RSOs, this is done through the TensorFlow runtime. Examples of context, include RPC deadlines to prevent infinite cycles and application IDs for ACLs and quotas, which are available as part of the original user query.

Accounting and Access Control. The distributed model described in Section 1 runs in a multi-tenant environment. To facilitate resource sharing, while isolating users from being adversely impacted by other user models at inference-time, the serving system could use RPC request originator or model-owner details for accounting. Such contextual information is propagated through the aforementioned context system. Similarly, with a simple access control configuration, the model-owner can control which users are allowed to manage and query the distributed model.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Commun. ACM* 51, 1 (Jan. 2008), 117–122. https://doi.org/10.1145/1327452.1327494

[3] Yi-Fan Chen, Ashish Agarwal, Rasmus Larsen, Walid Krichene, Karthik Lakshmanan, Chris Colby, Ahmed Taei, and John Anderson. 2016. Distributed TensorFlow WALS Trainer: Google Internal Design Report. (2016).

[4] Edith Cohen and David D. Lewis. 1997. Approximating Matrix Multiplication for Pattern Recognition Tasks. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 682–691. http://dl.acm.org/citation.cfm?id=314161.314415

[5] Ruiqi Guo, Sanjiv Kumar, Krzysztof Choromanski, and David Simcha. 2016. Quantization based Fast Inner Product Search. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Arthur Gretton and Christian C. Robert (Eds.), Vol. 51. PMLR, Cadiz, Spain, 482–490. http://proceedings.mlr.press/v51/guo16a.html

[6] Y. Hu, Y. Koren, and C. Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *2008 Eighth IEEE International Conference on Data Mining*. 263–272. https://doi.org/10.1109/ICDM.2008.22

[7] Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. 2012. Efficient Retrieval of Recommendations in a Matrix Factorization Framework. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM '12)*. ACM, New York, NY, USA, 535–544. https://doi.org/10.1145/2396761.2396831

[8] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *ArXiv e-prints* (Dec. 2017). arXiv:cs.DC/1712.06139

[9] Parikshit Ram and Alexander G. Gray. 2012. Maximum Inner-product Search Using Cone Trees. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*. ACM, New York, NY, USA, 931–939. https://doi.org/10.1145/2339530.2339677

[10] Sebastian Schelter, Venu Satuluri, and Reza Zadeh. 2014. Factorbird - a Parameter Server Approach to Distributed Matrix Factorization. *CoRR* abs/1411.0602 (2014). arXiv:1411.0602 http://arxiv.org/abs/1411.0602

[11] Wei Tan, Liangliang Cao, and Liana Fong. 2016. Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. ACM, New York, NY, USA, 219–230. https://doi.org/10.1145/2907294.2907297

[12] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix X Yu. 2017. Multiscale Quantization for Fast Similarity Search. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5749–5757. http://papers.nips.cc/paper/7157-multiscale-quantization-for-fast-similarity-search.pdf

[13] Doris Xin, Nicolas Mayoraz, Hubert Pham, Karthik Lakshmanan, and John R. Anderson. 2017. Folding: Why Good Models Sometimes Make Spurious Recommendations. In *Proceedings of the Eleventh ACM Conference on Recommender Systems (RecSys '17)*. ACM, New York, NY, USA, 201–209. https://doi.org/10.1145/3109859.3109911