Merlin: A GPU Accelerated Recommendation Framework

Even Oldridge*
Julio Perez
Ben Frederickson
Nicolas Koumchatzky
eoldridge@nvidia.com
jperez@nvidia.com
benf@nvidia.com
nkoumchatzky@nvidia.com
NVIDIA

Minseok Lee
Zehuan Wang
Lei Wu
Fan Yu
minseokl@nvidia.com
jwang@nvidia.com
davidwu@nvidia.com
fayu@nvidia.com
NVIDIA

Rick Zamora
Onur Yilmaz
Alec Gunny
Vinh Nguyen
rzamora@nvidia.com
oyilmaz@nvidia.com
agunny@nvidia.com
vinhn@nvidia.com
NVIDIA

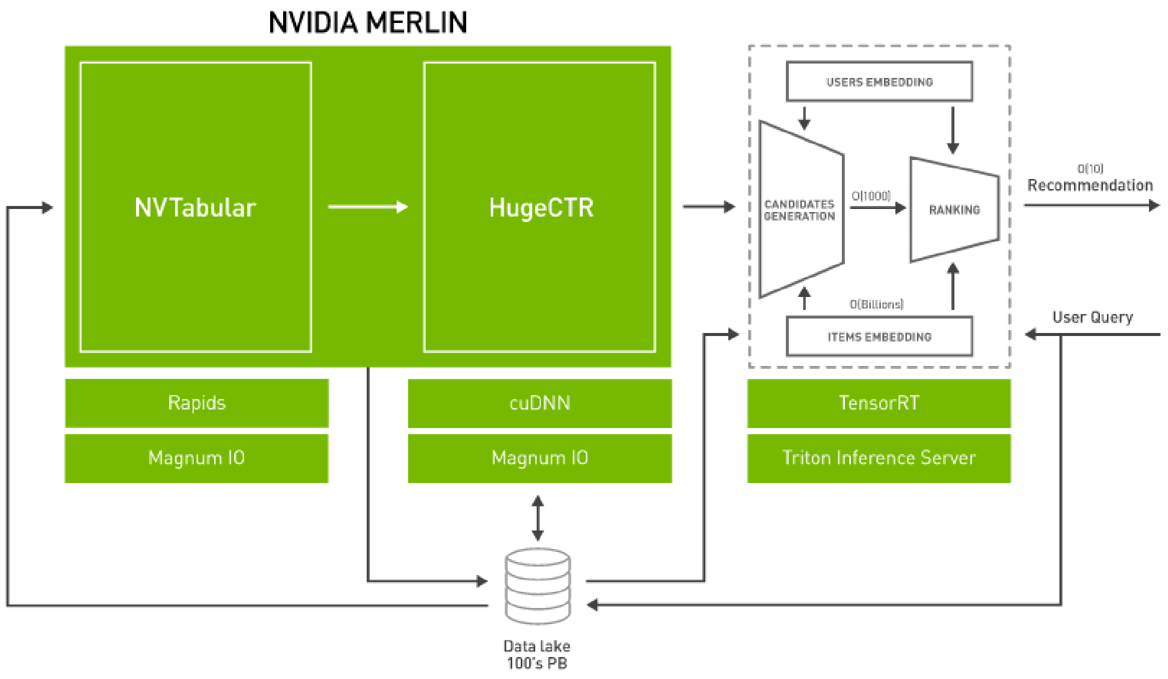

Figure 1: Merlin Recommender System Framework

ABSTRACT

The scale of recommender system datasets in industry has grown to the point where special thought must be taken in both the preparation of the data and in the training methods used in order to avoid performance issues that can slow down the total training iteration time by orders of magnitude. Extract Transform Load (ETL) and data preparation can take more time than training, leading to the adage that data scientists spend >75% of their time preparing data for modelling. These large datasets are required in order to enable deep learning (DL) based recommender systems which outperform traditional methods in industry settings where small differences in model performance can have a significant impact on the bottom line. Similarly, training DL recommenders whose embeddings scale beyond a single GPU requires significant expertise to implement efficiently.

In this paper we present Merlin, an open source Graphics Processing Unit (GPU) accelerated recommendation framework that scales to datasets and user/item combinations of arbitrary size. The framework provides fast feature engineering and preprocessing for operators common to recommendation datasets and high training throughput of several canonical deep learning based recommender

*Corresponding Author

IRS '20, August 22–27, 2020, San Diego, CA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6201-8/20/08…$15.00
https://doi.org/10.1145/3292500.3330823

models including Wide and Deep[3], Deep Cross Networks[14], DeepFM[6], and DLRM[8] to enable fast experimentation and production retraining. For production deployment Merlin also provides low latency, high-throughput inference. These components combine to provide an end to end framework for training and deploying deep learning recommender system models on the GPU that is both easy to use and highly performant. The Merlin framework is freely available and open source[10, 11].

CCS CONCEPTS

• **Information systems** → **Recommender systems**; Content analysis and feature selection; • **Computer systems organization** → Single instruction, multiple data.

KEYWORDS

Recommender Systems, Feature Engineering, Deep Learning, GPU Acceleration

ACM Reference Format:
Even Oldridge, Julio Perez, Ben Frederickson, Nicolas Koumchatzky, Minseok Lee, Zehuan Wang, Lei Wu, Fan Yu, Rick Zamora, Onur Yilmaz, Alec Gunny, and Vinh Nguyen. 2020. Merlin: A GPU Accelerated Recommendation Framework. In *IRS '20: International Symposium on Industrial Recommendation Systems,August 22–27, 2020, San Diego, CA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3292500.3330823

1 INTRODUCTION

Building recommender systems in industry is a complex process of data collection, data preparation and transformation, modelling, and production deployment. Each of these tasks requires significant knowledge and understanding to achieve both a high quality of recommendation and good performance when it comes to the total iteration time of training. Even amongst organizations with significant engineering and data science teams dedicated to each stage of the process, interaction between the different stages can play a major role in the time it takes to optimize feature engineering and training and to deploy a model into production.

In the initial experimentation phase, Extract-Transform-Load (ETL) operations prepare and export datasets for training, usually in the form of tabular data that can reach hundreds of TBs in scale. An example public dataset of this type is the Criteo Terabyte click logs dataset[4], which contains click logs of four billion interactions over a period of 24 days. Industry datasets can be orders of magnitude larger. During experimentation, data scientists and machine learning engineers use feature engineering, which creates new features by transforming existing ones, and preprocessing, which prepares the engineered features for consumption by the model, to transform the dataset into something ready for training. Training is then performed using DL frameworks such as TensorFlow[1], PyTorch[9], or our recommender specific training framework, HugeCTR[10]. For many companies this experimentation never ends, with models continually being updated and new features added to improve performance over time.

Once the models have been trained and evaluated offline, they can be moved into production for online evaluation, typically through A/B testing. The recommender system inference process involves selecting and ranking candidate items by the predicted probability that the user will interact with them. Selection of a subset of items is necessary for most commercial applications, with millions of items to choose from. The selection method is typically a highly efficient algorithm such as approximate nearest neighbors, random forest, or filtering based on user preferences and business rules. The DL recommender model then re-ranks the candidates and those with the highest predicted probability are presented to the user.

There are many challenges when training large-scale recommender systems:

- Huge datasets: Commercial recommender systems are trained on large datasets, often terabytes or more. At this scale, data ETL and preprocessing steps often take much more time than training the DL model.
- Complex preprocessing and feature engineering pipelines: Datasets need to be preprocessed and transformed into a suitable form to be used with DL models and frameworks. Feature engineering requires iteration and exploration and code complexity can be a significant challenge, particularly when moving a pipeline into production.
- Inefficient dataloading: For many models dataloading during training is a input bottleneck, leading to GPU underutilization. Increasing the batch size can help in that context but must be done carefully to avoid impacting the quality of recommendations.
- Extensive repeated experimentation: The whole data ETL, feature engineering, training and evaluation process must be repeated many times, potentially on many model architectures, requiring significant computational resources and time. Even after being deployed, recommender systems also require periodic retraining to account for new users, items and recent trends in order to maintain high accuracy over time. Lowering total iteration time is key to effective recommender system deployments.
- Huge embedding tables: Embedding is a universally employed technique to handle categorical variables, for both users and items as well as their associated side information. On large commercial systems, the user and item base can easily reach an order of hundreds of millions, requiring a large amount of memory compared to other types of DL layers. Unlike other DL operations, embedding lookup is memory bandwidth constrained. While the CPU generally offers a larger memory pool, it has much lower memory bandwidth and compute when compared to a GPU.
- Distributed training: Distributed training is continually setting new records in training DL models in the vision and natural language domains, as reflected by the MLPerf benchmark[7]. This concept is a still relatively new in recommender systems where the user and item embeddings that dominate these models can exceed GPU memory by orders of magnitude. Distributed training requires both model parallelism and data parallelism, making it hard to achieve high scale-out efficiency.

Even when training challenges have been overcome, the deployment of deep learning based recommenders to a production setting bring new issues that must also be resolved. Some notable challenges for deploying large-scale recommendation systems in production include:

- Real-time inference: For each query, the number of user-item pairs to score can be as large as a few thousand after candidate generation. The inference server must support high throughput and low latency to serve many users concurrently and meet the Service Level Agreement (SLA) requirements necessary in order to provide a positive user experience.
- Monitoring and retraining: Recommender systems operate in continuously changing environments: new users registering, new items becoming available, and emerging trends. To remain effective recommender systems need ongoing monitoring and retraining to ensure that recommendation quality doesn't decline. The inference server must also be able to concurrently deploy different versions of a model, and load/unload models on the fly to facilitate A/B testing or other evaluation methods.

These scale and diversity of these challenges make the training and deployment of deep learning based recommender systems into production the domain of large companies with big engineering teams who are able to overcome them. The benefit of deploying these large scale deep learning recommender systems is significant enough to warrant the effort, but this still remains a barrier to the small and even mid sized companies who would like to explore deep learning based recommendation.

The Merlin recommender system framework is an attempt to provide solutions and structure in the recommender system space. Its focus is on ease of use for each component library, simple interoperability between components, and acceleration on the GPU, scaling easily to multi-GPU and multi-Node solutions when required. Component libraries are also interoperable with other common frameworks and tools in the deep learning ecosystem for maximum flexibility in production settings where existing solutions require integration, allowing users of the framework to pick and chose the components that work best for them. In the next section we will introduce the Merlin framework and its component libraries.

2 MERLIN: A GPU ACCELERATED RECOMMENDATION FRAMEWORK

Merlin is an application framework and ecosystem created to facilitate all phases of recommender system development, from experimentation to production, accelerated on GPUs. Figure 1 shows an architectural diagram of Merlin, and its three core components are described below.

NVTabular: A collection of operators for high-speed, on-GPU preprocessing and feature engineering of tabular data with the capability of scaling to handle terabyte-scale datasets. The output of NVTabular[11] can be made available to a training framework such as HugeCTR, PyTorch, or TensorFlow at high throughput using NVTabular dataloader extensions, eliminating the input bottleneck commonly found in GPU recommender system training.

HugeCTR: HugeCTR is a highly efficient C++ recommender system dedicated training framework. It features multi-GPU and multi-node training, and supports model-parallel and data-parallel scaling, allowing users to implement embeddings that scale across multiple GPUs to utilize the entire memory space. HugeCTR covers common and recent recommender system architectures such as Wide and Deep (W&D), Deep Cross Network, and DeepFM, with Deep Learning Recommender Model (DLRM) support coming soon.

Triton Inference Server: Built using TensorRT[12], an SDK for high performance deep learning inference, Triton Inference Server[13] includes a DL inference optimizer and runtime that delivers low latency and high throughput. Triton provides a comprehensive, GPU-optimized inferencing solution, allowing models from a variety of backends to be served including HugeCTR, PyTorch, TensorFlow, TensorRT, and Open Neural Network Exchange (ONNX) runtime[2]. Triton Inference Server automatically manages and makes use of all the available GPUs and offers capability to serve multiple versions of a model and report various performance metrics, allowing for effective model monitoring and A/B testing.

In the next sections, we explore each of these components in detail.

3 NVTABULAR: FAST FEATURE ENGINEERING, PREPROCESSING AND DATALOADING

The time taken to perform feature engineering and preprocessing of recommender system datasets often exceeds the time it takes to train the model itself. As a concrete example, processing the Criteo Terabyte Click Logs dataset takes 5.5 days to complete using the original NumPy script, while training DLRM on the processed dataset takes less than an hour on a single NVIDIA V100 GPU.

NVTabular is a feature engineering and preprocessing library, designed to quickly and easily manipulate terabyte-scale datasets. It is especially suitable for recommender systems, which require a scalable way to process additional information, such as user and item metadata or contextual information. It provides a high-level abstraction to simplify code and accelerates computation on the GPU using the RAPIDS cuDF library. Using NVTabular, with just 10-20 lines of high-level API code, you can set up a data engineering pipeline and achieve up to 10X speedup compared to optimized CPU-based approaches while experiencing no dataset size limitations, regardless of the GPU/CPU memory capacity.

The total time taken to do ETL is a mix of the time to run the code, but also the time taken to write it. The RAPIDS team has done amazing work accelerating the Python data science ecosystem on GPU, providing acceleration through cuDF, Apache Spark 3.0[15], and Dask-cuDF[5]. NVTabular uses those accelerations but provides a higher-level API focused on recommender systems, which greatly simplifies code complexity while still providing the same level of performance. Figure 2 shows the positioning of NVTabular relative to other dataframe libraries.

The preprocessing workflow required to transform the 1-TB Criteo Ads dataset can be implemented with just 12 lines of code using NVTabular. Numerical and categorical columns are specified explicitly. An NVTabular workflow is defined and supplied with a set of train and validation files. Then, preprocessing operations are added to the workflow and data is persisted to disk. In comparison, custom-built processing codes, such as the NumPy-based data util in Facebook's DLRM implementation, can have 500-1000 lines of code for the same pipeline.

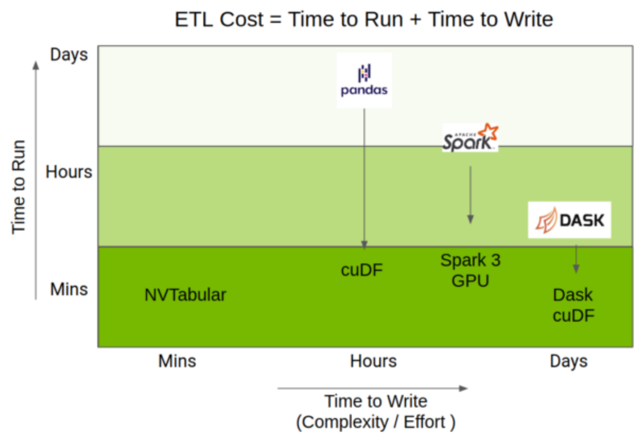

Figure 2: NVTabular positioning compared to other popular dataframe libraries

Figure 3 shows the relative performance of NVTabular to the original DLRM preprocessing script, and a Spark-optimized ETL process running on a single node cluster. Of note is the percentage of time taken up by training compared to the time taken in ETL. In the baseline cases, the ratio of ETL to training almost exactly matches the common adage that data scientists spend 75% of their time processing the data. With NVTabular, that relationship is flipped.

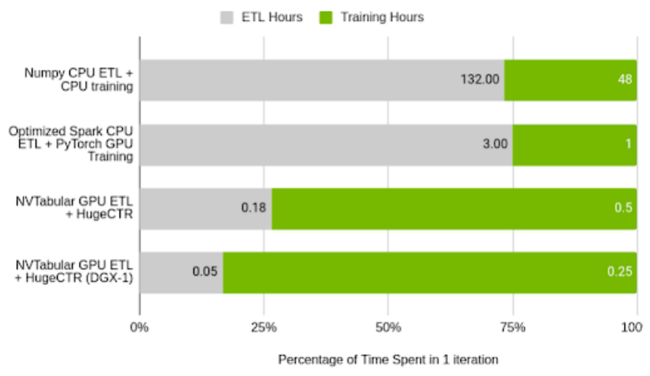

Figure 3: NVTabular Criteo comparison. GPU (Tesla V100 32 GB) vs. CPU (AWS r5d.24xl, 96 cores, 768 GB RAM)

The total time taken to process the dataset and train the model on a CPU is over a week using the original script. With significant effort, that can be reduced to four hours using Spark for ETL and training on a GPU. With NVTabular and HugeCTR, which we cover later in this post, you can accelerate iteration time to 40 minutes for a single GPU and 18 minutes on a DGX-1 cluster. In the latter case, the four-billion interaction dataset is processed in only three minutes.

4 HUGECTR: A RECOMMENDATION SPECIFIC TRAINING FRAMEWORK

HugeCTR is a highly efficient GPU framework designed for recommender model training, which targets both high performance and ease of use. It supports simple MLP models and also more sophisticated hybrid models such as W&D, Deep Cross Network, and DeepFM. We are also working on enabling DLRM with HugeCTR. The model details and hyperparameters can be specified easily in JSON format, allowing for quick selection from a range of common models.

Compared to other generic DL frameworks such as PyTorch and TensorFlow, HugeCTR is designed specifically to accelerate end to end training performance of large scale CTR models. Unlike the generic frameworks it explicitly prevents users from developing their model in a way that isn't optimal, constraining to optimal layer width and memory sizes in order to achieve significant performance benefits. To prevent data loading from becoming a major bottleneck in training, it implements a dedicated data reader which is inherently asynchronous and multi-threaded, so that the data transfer time overlaps with the GPU computation.

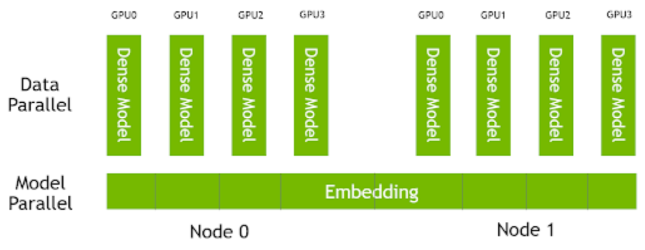

Figure 4: HugeCTR model and data parallelism architecture.

The embedding table in HugeCTR is model-parallel and distributed across all the GPUs in a cluster, which consists of multiple nodes and multiple GPUs. The dense component of these models is data-parallel, with one copy on each GPU (Figure 4). For high-speed and scalable inter and intra-node communication, HugeCTR uses NCCL. For cases where there are many input features, the HugeCTR embedding table can be segmented into multiple slots. The features that belong to the same slot are converted to the corresponding embedding vectors independently, and then reduced to a single embedding vector. It allows you to efficiently reduce the number of effective features within each slot to a manageable degree.

Figure 5 shows the training performance of a W&D network with HugeCTR on a single V100 GPU on the Criteo Terabyte Click Ads dataset, compared to TensorFlow on the same GPU and a dual 20-core Intel Xeon CPU E5-2698 v4. HugeCTR achieves a speedup of up to 54X over TensorFlow CPU, and 4X that of TensorFlow GPU. To reproduce the result, the Wide and Deep sample, including the instructions and the JSON model config file, is provided in the HugeCTR repo[10].

Figure 6 shows the strong scaling results of HugeCTR with a deeper W&D model on a DGX-1 for both the full-precision mode (FP32) and mixed-precision mode (FP16).

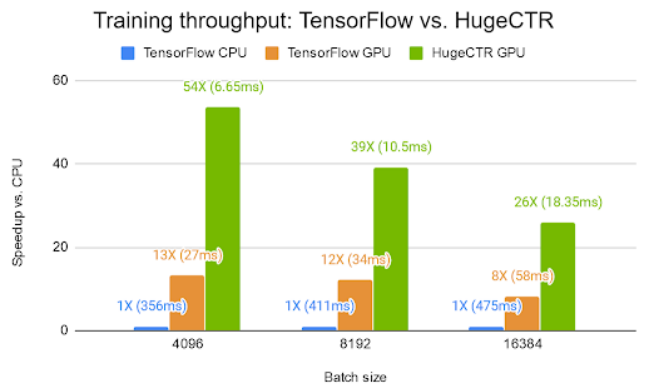

Figure 5: TensorFlow v2.0 CPU and GPU performance in comparison with HugeCTR v2.1 on a single V100 16-GB GPU. CPU: Dual 20-core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz (80 threads). Model: W&D, 2x1024 FC layers. Bars represent speedup factor vs. TensorFlow CPU. The higher, the better. Numbers in parentheses denote average time taken for one iteration.

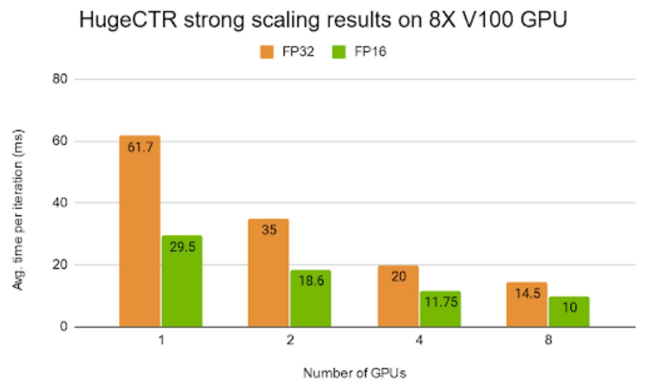

Figure 6: HugeCTR strong scaling results on 8X V100 16-GB GPU. Batch size: 16384. Model: W&D, 7x1024 FC layers.

5 TENSORRT & TRITON INFERENCE SERVER

TensorRT is an SDK for high performance DL inference which includes a DL inference optimizer and runtime that delivers low latency and high throughput for inference applications. TensorRT can accept trained neural networks from all DL frameworks using a common interface, the open neural network exchange format (ONNX). It automatically optimizes the network architecture using operations such as vertical and horizontal layer fusion and reduced precision operations (FP16, INT8) that leverage the high mixed-precision arithmetic throughput of the Tensor Cores on NVIDIA GPUs. TensorRT also automatically selects the best kernel based on the task at hand and the target GPU architecture. For further model-specific optimization, TensorRT is highly programmable and extensible, allowing you to insert your own plugin layers.

Triton Inference Server builds upon this SDK to provide a cloud-inferencing solution optimized for NVIDIA GPUs. The server provides an inference service via an HTTP or gRPC endpoint, allowing remote clients to request inferencing for any model being managed by the server. Triton Server can serve DL recommender models using several backends, including TensorFlow, PyTorch (TorchScript), ONNX runtime, and TensorRT runtime. With DLRM, we show how to deploy a pretrained PyTorch model with Triton, achieving a 9X reduction in latency on an A100 GPU compared to CPU, as shown in Figure 7.

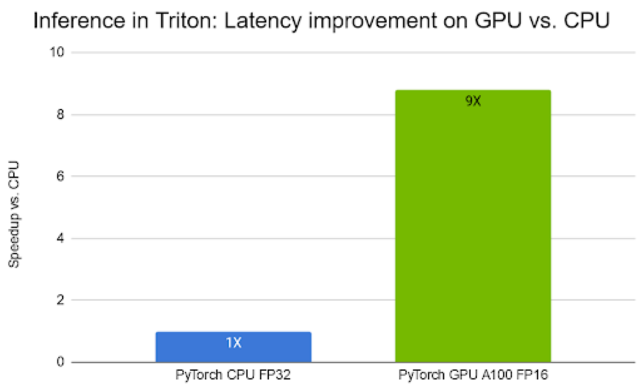

Figure 7: DLRM inference with Triton Inference Server. Bars represent the speedup factor for GPU vs. CPU. Batch size 2048. CPU: Dual Intel(R) Xeon(R) Platinum 8168 @2.7 GHz (96 threads). GPU: Tesla A100 40 GB. The higher, the better.

Triton contains optimizations specifically targeted at recommender systems. As an example the W&D model was upgraded by implementing a fused embedding lookup kernel to leverage the GPU high-memory bandwidth. Running in the Triton Server custom backend, the GPU W&D TensorRT inference pipeline provides up to 18X reduction in latency and 17.6X improvement in throughput compared to an equivalent CPU inference pipeline. All this is deployed using Triton Inference Server to provide production quality metrics and to ensure production robustness.

6 CONCLUSION

In this paper we present Merlin, an accelerated recommendation framework that scales on the GPU to datasets of arbitrary size. The framework provides fast feature engineering and preprocessing for operators common to recommendation datasets and high throughput training of several canonical deep learning based recommender models. It has been designed to enable fast experimentation and production retraining. For production deployment Merlin also provides low latency, high-throughput inference.

The three components of Merlin: NVTabular[11], HugeCTR[10], and Triton Inference Server[13], are open source and freely available. Together these components combine to provide an end to end framework for training and deploying deep learning recommender system models on the GPU that is both easy to use and highly performant. The framework is actively under development and welcomes issues and feature requests to help guide future development.

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, Dandelion 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. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[3] 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. 2016. Wide & Deep Learning for Recommender Systems. *CoRR* abs/1606.07792 (2016). arXiv:1606.07792 http://arxiv.org/abs/1606.07792

[4] Criteo. 2013. *Criteo Terabyte Click Logs dataset.* https://labs.criteo.com/2013/12/download-terabyte-click-logs/

[5] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* https://dask.org

[6] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. *CoRR* abs/1703.04247 (2017). arXiv:1703.04247 http://arxiv.org/abs/1703.04247

[7] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, G. Wei, and C. Wu. 2020. MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. *IEEE Micro* 40, 2 (2020), 8–16.

[8] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[9] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[10] HugeCTR Development Team. 2019. *HugeCTR: a high-efficiency GPU framework for Click-Through-Rate (CTR) estimation training.* https://github.com/NVIDIA/HugeCTR

[11] NVTabular Development Team. 2020. *NVTabular: GPU Accelerated ETL.* https://github.com/NVIDIA/NVTabular

[12] TensorRT Development Team. 2018. *TensorRT.* https://github.com/NVIDIA/tensorrt

[13] Triton Development Team. 2018. *Triton Inference Server.* https://github.com/NVIDIA/triton-inference-server

[14] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. *CoRR* abs/1708.05123 (2017). arXiv:1708.05123 http://arxiv.org/abs/1708.05123

[15] Matei Zaharia, Reynold Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache spark: A unified engine for big data processing. *Commun. ACM* 59 (11 2016), 56–65. https://doi.org/10.1145/2934664