Distributed C++-Python embedding for fast predictions and fast prototyping

Georgios Varisteas
University of Luxembourg
georgios.varisteas@uni.lu

Tigran Avanesov
OlaMobile
tigran.avanesov@olamobile.com

Radu State
University of Luxembourg
radu.state@uni.lu

ABSTRACT

Python has evolved to become the most popular language for data science. It sports state-of-the-art libraries for analytics and machine learning, like Sci-Kit Learn. However, Python lacks the computational performance that a industrial system requires for high frequency real time predictions.

Building upon a year long research project heavily based on SciKit Learn (sklearn), we faced performance issues in deploying to production. Replacing sklearn with a better performing framework would require re-evaluating and tuning hyperparameters from scratch. Instead we developed a python embedding in a C++ based server application that increased performance by up to 20x, achieving linear scalability up to a point of convergence. Our implementation was done for mainstream cost effective hardware, which means we observed similar performance gains on small as well as large systems, from a laptop to an Amazon EC2 instance to a high-end server.

ACM Reference Format:
Georgios Varisteas, Tigran Avanesov, and Radu State. 2018. Distributed C++-Python embedding for fast predictions and fast prototyping. In *Second Workshop on Distributed Infrastructures for Deep Learning (DIDL '18), December 10–11, 2018, Rennes, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3286490.3286560

1 INTRODUCTION

Machine learning techniques have seen widespread adoption recently. So much so that even small companies without specialized engineers have sought to integrate predictive classifiers in their infrastructure. Python has emerged as one of the de facto platforms for machine learning, sporting several state-of-the-art frameworks like Tensorflow, PyTorch, Keras, Sci-Kit Learn, and others. Python allows for very fast prototyping relative to other platforms, has simple syntax and gradual learning curve, but it suffers from mediocre execution performance.

Python's lacking performance is due to multiple factors, including overhead from being interpreted and dynamically typed. Horizontal scaling via parallelism could improve performance, however only multiprocessing can produce true gains. For example, Python's multi-threading suffers severe performance penalties due to the global interpreter lock (GIL) [2].

In this paper we describe a simple technique for delegating time consuming preprocessing tasks to a C++ based component, embedding python for the machine learning parts; this scheme enables both high execution performance and fast prototyping. Our architecture is modular and can be used as is for a variety of models.

We have deployed this platform in programmatic display advertising for ranking available ad campaigns based on user attributes. When a user clicks a banner there is opportunity to optimize the target url, called the landing page, based on the user's profile. The available time window is 200ms. The list of potential campaigns is rapidly malleable, thus one classifier per campaign is required, predicting the potential for conversion given a user feature vector. These classifiers where build using Sci-Kit Learn [14]. Our first exclusively Python based prototype managed to predict on 100 models per 200ms. Our proposed implementation design achieved prediction on up to 500 models per available hardware thread, and almost linear scalability till up to a point for performance convergence.

2 RELATED WORK

Recently a lot of research is conducted on improving the performance of predictive systems utilizing Machine Learning. Such examples is binary inference optimization by NVIDIA [13]; it optimizes models built with a variety of frameworks and converts them into binary executable engines. TensorRT is such an optimizer [5, 12]. It performs fusion of either sequential layers, or layers sharing the same data input. However,

ACM ISBN 978-1-4503-6119-4/18/12…$15.00
https://doi.org/10.1145/3286490.3286560

such tools support a constrained set of features, which could prove suboptimal in certain cases.

Tensorflow [1] is a well known deep learning framework. Tensorflow Serving [1] is a production tool for optimally serving deep learning models. As with most other deep learning tools, Tensorflow-Serving's performance gains are dependent on the availability of expensive state-of-the-art GPU based hardware infrastructure. Such solutions are not always affordable for small applications of predictive systems.

Several other projects have targeted optimized deployment of Machine Learning and Deep Learning applications using distributed systems and parallelism Most of these solutions have targeted high scale systems and offer competitive performance at the cost of development and equipment cost. Li et. al. propose a parameter server for distributing the load and improving scalability [11]. MXNet [3] is a multi-language machine learning development library, which combines the benefits of multiple established frameworks and is designed for distributed deployment. MLbase [8] is a development framework with simplified APIs. Finally Lee et.al. [10] propose STRANDS, a novel parallelization runtime for improved distributed machine learning.

3 IMPLEMENTATION

The platform consists of 2 main components. The primary C++ process and the Python workers. The primary process includes the *request server*, a *request buffer*, the *prediction handlers*, and the in-memory *classifier database*. The *Python workers* are implemented via multiprocessing each with access to the classifier database in inter-process shared memory. Figure 1 depicts the overall design.

3.1 Request Server

We implemented a typical high performance REST server for handling incoming requests. This server will accept multiple simultaneous requests, filter for correctness, decompose into fine grained tasks which are then placed on the request buffer. It will also gather the predictions, package into a single JSON structure which will be returned to the client.

A single request can be translated to a set of predictions over N number of classifiers. Thus N tasks will be generated. In the end the array of predictions will sorted on probability, trimmed to a configurable length, converted into a JSON array, and transmitted back to the client.

3.2 Request Buffer

The request buffer acts as a *bag of tasks* for the Prediction Handlers. Since the depth of the task tree is always 1, tasks will not recursively generate other tasks, it made sense to use a single centralized buffer instead of other techniques like work dealing. It also simplified handling of the resulting predictions. A single buffer allows for results to be stored in the same data structures used for the initial task, which the buffer can keep track of. Then once all tasks have completed, the buffer can asynchronously notify the request handler to produce the reply to the client.

3.3 Prediction Handlers and Python Workers

Prediction handlers are software threads which act as an interface between the C++ part and the Python Workers. For each spawned handler, there is a corresponding Python Worker. The prediction handler thread will block while a worker is processing a task. Communication between the two components is performed via Unix pipes. The data transmitted include a classifier id and the feature vector.

The execution of the task consists of the following steps:

(1) Preprocessing of the feature vector. For example encoding categorical data into binary vectors
(2) Acquire the corresponding classifier, get it from the buffer if already preloaded, or load it from disk
(3) Perform the prediction.
(4) Transmitted the result back to the prediction handler via the same pipe.

Upon receiving a prediction result the *Prediction Handler* will store it onto the original task's structure and mark the task as completed.

Python Workers can apply two different types of parallelism to scale performance. They can apply the same input feature vector to multiple classifiers (MISD), apply different feature vectors to the same classifier (SIMD), or both simultaneously (MIMD) [4]. Our production deployed version of the system is utilizing both strategies. The server is capable of handling multiple requests at the same time, however all requests are processed in sequence by a single pool of workers. Thus the parallelism model per request is MISD, but due to interleaved requests each worker is applying SIMD, which overall equates to MIMD.

3.3.1 *Custom OneHotEncoding.*

Our test data have been all categorical. A common way to use such data is *One Hot Encoding* [6, 15], which converts the feature vector into a binary vector. For each feature F a binary vector V_F is constructed with length equal to the amount of potential values of that feature. Thus each element corresponds to a single value. This is done for all features. When encoding the index corresponding to the given value is set to 1 while everything else is 0. The final feature vector is concatenation of those vectors $V := \bigcup V_{F_i}$.

The typical implementation of the above algorithm initializes each vector V_F with zeros, and uses the value itself as

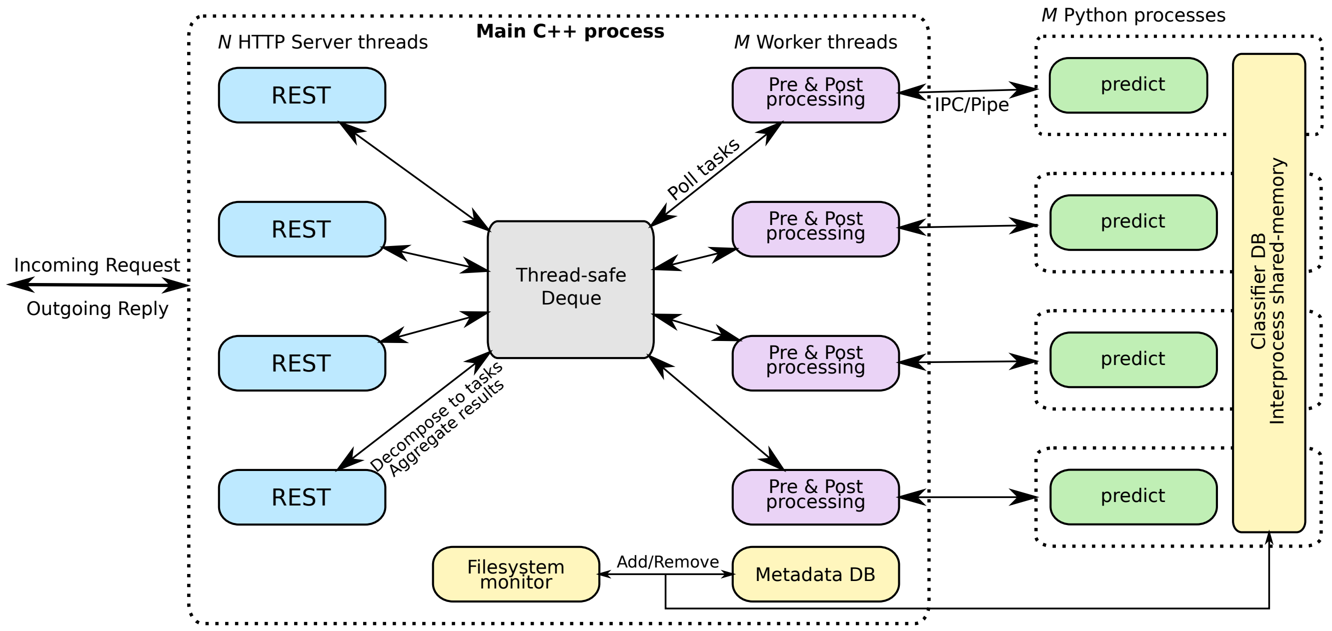


Figure 1: Predictor diagram

the index for the element set to one. This is very fast. However, when the pre-encoded value is a very large id – in the hundreds of thousands or even millions – it can create very sparse vectors with most elements never to be set to one. A solution would be to compact the values to a zero-based consecutive indexing. On the one hand, this would benefit memory requirements and prediction accuracy. On the other hand, such mapping would not be linear and would require a lookup for encoding. Moreover, if a request includes a feature value not observed during training, it would not be possible to be encoded.

If a value has not been observed during training, it makes sense that the model would not know how to handle it anyway. It would be like facing a novel situation not experienced during a training. Following that logic, our compacted mapping includes one extra column called *other*. All previously unobserved values are encoded as *other*.

3.4 Continuous Classifier Training

The classifier database is an in-memory key-value storage. The amount of available classifiers can be in the thousands and potentially of multiple MB in size. To optimize memory usage, the DB is complemented with a fixed size buffer for preloaded classifiers. On insert into a full buffer, the LFU replacement policy [9] is applied.

The database is comprised by 2 independent implementations, one in C++ and one in Python. However there is a unified mechanism for inserting and removing entries, maintaining the same keys.

The C++ entries include metadata used for preprocessing the feature vectors. The Python entries include the classifier itself and also the custom OneHotEncoder which has been initialized for the observed feature values of each classifier.

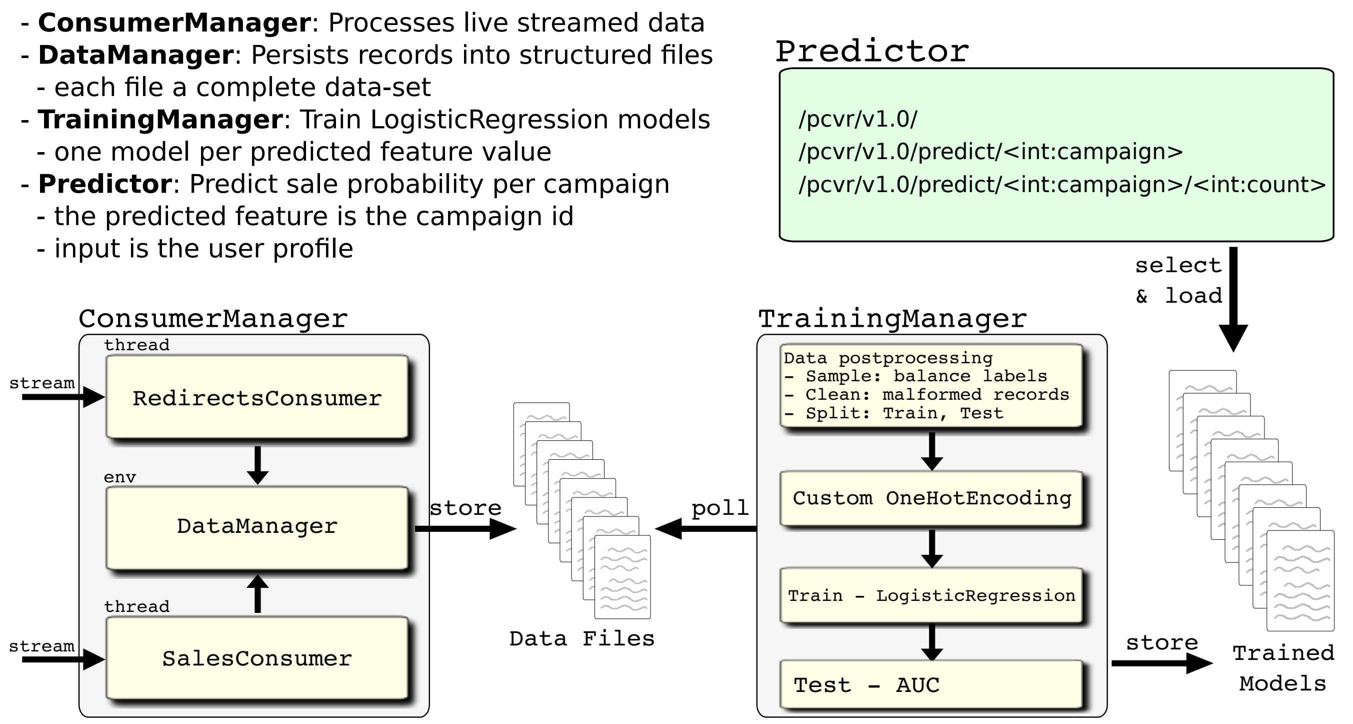
To optimize production behavior an automated filesystem monitor was also implemented to automatically detect persisted classifiers that are added or removed. This is important since the proposed system is coupled with a dynamic trainer which continuously trains new classifiers from a stream of live data. Figure 2 outlines the pipeline of the trainer.

Python workers are each a separate process, thus the Python component of the classifier DB lies in interprocess shared memory.

4 EVALUATION

We evaluated our implementation in isolation on a dedicated server, in order to avoid interference. In all experiments, secondary parameters were set in such a way that wouldn't affect performance. For example the buffer size for preloaded classifier was always large enough to fit all available classifiers. The host system had 64 Hardware Threads of Intel® Xeon® CPU E5-4650 v4 and 352GB of RAM.

For all experiments there is 1:1 mapping between Python workers and Hardware threads; albeit placement of threads and processes is left to the default OS scheduler. All performance results are the median of 20 repetitions. Load distribution plots present the actual results of the repetition with the corresponding median execution time.



- **ConsumerManager**: Processes live streamed data
- **DataManager**: Persists records into structured files
  - each file a complete data-set
- **TrainingManager**: Train LogisticRegression models
  - one model per predicted feature value
- **Predictor**: Predict sale probability per campaign
  - the predicted feature is the campaign id
  - input is the user profile

Figure 2: Pipeline for data preparation and classifier training

4.1 Performance

The first experiment is a simple comparison of execution time. We developed and tested 3 different implementations: the initial solely Python based prototype, ii) a Python-based multi-threaded version, iii) the C++-Python embedding with multiprocessing.

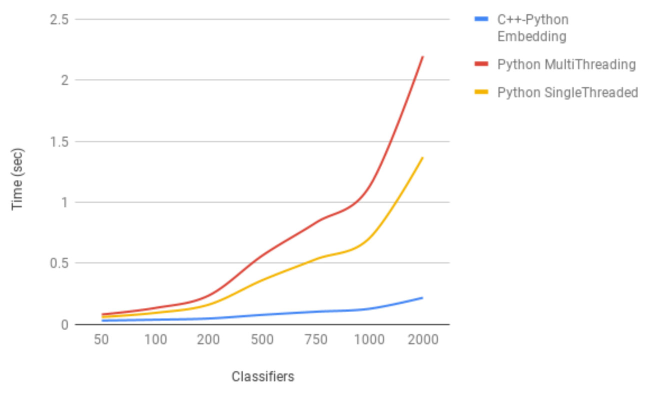


Figure 3: Total execution time predicting on 2000 classifiers for 3 different implementations

Figure 3 shows the execution time required to predict on various amount of classifiers for each version. Both Python versions perform multiple times worse than the C++-Python version.

It is interesting to note the steep of the each curve. For the Python versions, the execution time increases exponentially relative the size of the workload. Our proposed solution scales much better, showing that the cost of switching to Python is constant and independent.

4.2 Scalability

We evaluated the scalability of our solution by executing in isolation the same workload while varying the amount of Python Workers.

Figure 4 shows the execution time of predicting on 2000 classifiers. The input feature vector – the user's profile – is fixed. The execution time converges after 24 cores to 110ms; this is the minimum execution time our platform achieved on our test hardware. This time was achieved with multiple amounts of Python Workers above 24.

The prediction tasks are independent, thus one would expect continuous speedup improvement. There are 3 suspects for hindering the scalability of our implementation

- *The centralized queue for task distribution is synchronized with locks which might be creating a staircase effect*

  We measured the time spent waiting on the lock for each thread at all scales. The total time wait time was

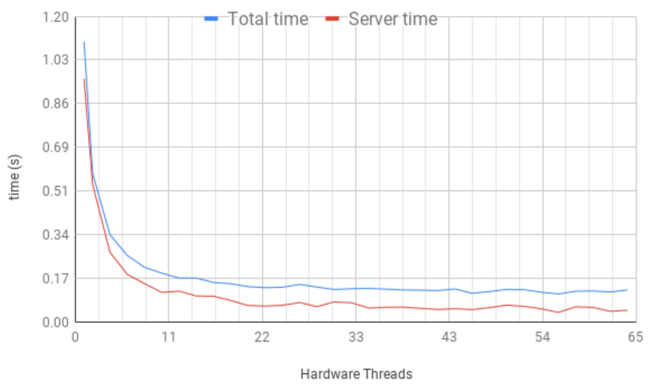


Figure 4: Execution time predicting on 2000 classifiers over varying Python-Worker Count with 1:1 mapping with Hardware Threads. Total time is measured externally by the client; Server time is measured within the Python Worker

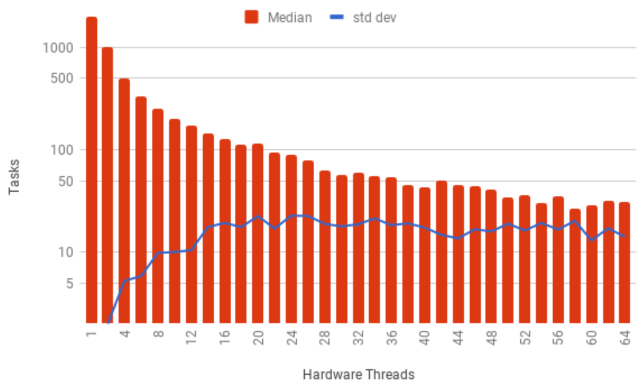


Figure 5: Median and Standard Deviation of number of tasks per Python Worker

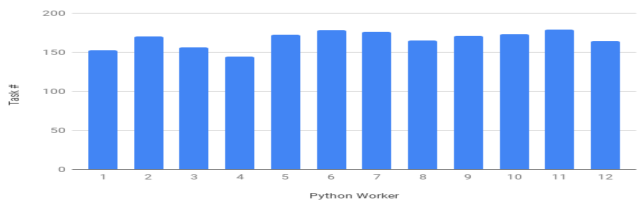


Figure 6: Distribution of tasks among 12 Python Workers with 1:1 mapping to Hardware Threads

consistently bellow 1ms. Hense the central queue is not the reason

- *The load distribution might not be uniform*

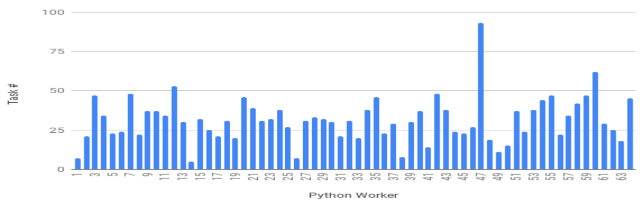


Figure 7: Distribution of tasks among 64 Python Workers with 1:1 mapping to Hardware Threads

Indeed the variance of the distribution of Tasks among workers gets very high at higher scales. Fig. 5 shows the median amount of Tasks executed by each worker, over the total amount of Python Workers; it also includes the standard deviation which at very high scales is almost as much as the median.
Fig. 7 shows the exact number of Tasks executed by each worker with 64 Python Workers. The distribution is quite divergent between workers. Fig. 6 shows a much more uniform distribution when using only 12 workers. Since the task queue is centralized, several workers might experience NUMA effects. The time required for a single prediction is around 1ms. Hence, the non-uniform cache transfer latency can potentially be significant relative to the tiny processing time between attempts.

- *The pre- and post- processing cost increases proportionally to the scale*
  The amount of data structures allocated is independent to the amount of workers. Furthermore, since each worker executes one task at a time, the same amount of post processing is performed independently of the number of workers.

One point of improvement would be the work distribution mechanism. Non-negligible time is spent allocating new tasks for each worker; packaging prediction tasks in batches would improve performance by reducing dynamic allocations. Using batches workers could run uninterrupted for a longer period of time. This scenario would indeed improve performance overall; however, it would impose a hard limit to the system's capacity for handling concurrent requests.

An alternative solution is the work dealing scheme [7] using a private task queue for each worker. This solution would improve decentralization and load uniformity. However it would burden the busiest thread (the request handler thread) extending the delay between pre-processing and task execution.

5 CONCLUSIONS

This work proposes a design which combines the speed of C++ for the computationally expensive tasks, with the ease of development of Python for the more experimental Machine Learning tasks. It is a simple to implement solution with widely known components and tools, that provides great performance and scalability benefits.

The future work of this project is firstly extensive testing in production environment with high traffic and fluctuating picks. Moreover, we are testing novel scheduling paradigms for the work distribution aspects to further improve scalability and concurrency.

6 ACKNOWLEDGMENTS

The authors would like to thank OlaMobile Luxembourg for providing access to infrastructure and live data. This project was funded by Fonds National de la Recherche Luxembourg, project number 11822390.

REFERENCES

[1] MartÃŋn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, and Google Brain. 2016. TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (2016), 265–284. https://doi.org/10.1038/nn.3331

[2] David Beazley. 2010. Understanding the Python GIL. *PyCON Python Conference. Atlanta, Georgia* C (2010), 1–62. http://dabeaz.com/python/UnderstandingGIL.pdf

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. 1–6. https://doi.org/10.1145/2532637

[4] Michael J Flynn and Kevin W Rudd. 1996. Parallel architectures. *ACM Comput. Surv.* 28, 1 (1996), 67–70. https://doi.org/10.1145/234313.234345

[5] Allison Gray, Chris Gottbrath, Ryan Olson, and Shashank Prasanna. 2017. Deploying deep neural networks with nvidia tensorrt.

[6] Xinran He, Stuart Bowers, Joaquin QuiÃśonero Candela, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, and Ralf Herbrich. 2014. Practical Lessons from Predicting Clicks on Ads at Facebook. *Proceedings of 20th ACM SIGKDD Conference on Knowledge Discovery and Data Mining - ADKDD'14* (2014), 1–9. https://doi.org/10.1145/2648584.2648589

[7] Danny Hendler. 2002. Work Dealing [Extended Abstract]. *Work* (2002), 164–172.

[8] T Kraska, A Talwalkar, J Duchi, R Griffith, M Franklin, and M Jordan. 2013. MLbase : A Distributed Machine-learning System. In *6th Biennial Conference on Innovative Data Systems Research (CIDRâĂŽ13)*.

[9] Donghee Lee, Jongmoo Choi, and Sam H Noh. 1996. LRFU (Least Recently / Frequently Used) Replacement 1 Introduction. *IEEE Trans. Comput.* 50, 12 (1996), 1352–1361.

[10] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. 2014. On Model Parallelization and Scheduling Strategies for Distributed Machine Learning. In *Nips*. 1–9.

[11] Mu Li. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 2014 International Conference on Big Data Science and Computing - BigDataScience '14*. 1–1. https://doi.org/10.1145/2640087.2644155

[12] S Migacz. 2017. 8-bit inference with TensorRT. In *GPU Technology Conference*.

[13] The Next and Platform Weekly. 2017. Nvidia Pushes Deep Learning Inference With New Pascal GPUs. *Next Platform, September* (2017), 1–6.

[14] Fabian Pedregosa, GaÃńl Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and ÃĽdouard Duchesnay. 2012. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2012), 2825–2830. https://doi.org/10.1007/s13398-014-0173-7.2

[15] Jun Wang, Weinan Zhang, and Shuai Yuan. 2016. Display Advertising with Real-Time Bidding (RTB) and Behavioural Targeting. *arXiv preprint arXiv:1610.03013* (2016). http://arxiv.org/abs/1610.03013