conference
proceedings

2019 USENIX Conference on Operational Machine Learning

Santa Clara, CA, USA
May 20, 2019

Sponsored by
usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISBN 978-1-939133-00-7

Proceedings of the 2019 USENIX Conference on Operational Machine Learning
Santa Clara, CA, USA May 20, 2019

OpML '19 Sponsors

Gold Sponsor

Silver Sponsors

Bronze Sponsor

ISBN 978-1-939133-00-7

USENIX Supporters

USENIX Patrons
Bloomberg • Facebook • Google • Microsoft • NetApp

USENIX Benefactors
Amazon • Oracle • Two Sigma • VMware

USENIX Partners
BestVPN.com • Cisco Meraki • Teradactyl • TheBestVPN.com

Open Access Publishing Partner
PeerJ

USENIX Association

Proceedings of the 2019 USENIX Conference on Operational Machine Learning

May 20, 2019
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs

Bharath Ramsundar, *Computable*
Nisha Talagala, *Pyxeda AI*

Program Committee

Jeremy Barnes, *Element AI*
Fei Chen, *LinkedIn*
Mike Del Balso, *NewCo*
Sindhu Ghanta, *ParallelM*
Sean Grullon, *GSK Healthcare*
Neoklis Polyzotis, *Google*
Jennifer Prendki, *Figure8*
Suresh Raman, *Intuit*
Marius Seritan, *LinkedIn*
Sarah Sirajuddin, *Google*
Eno Thereska, *Amazon*
Boris Tvaroska, *Lenovo*
Todd Underwood, *Google*
Shivaram Venkataraman, *University of Wisconsin—Madison*
Martin Wicke, *Google*
Josh Wills, *Slack*

Steering Committee

Nitin Agrawal, *Samsung*
Eli Collins, *Accel Partners/Samba Nova*
Casey Henderson, *USENIX Association*
Robert Ober, *Nvidia*
Bharath Ramsundar, *Computable*
Jairam Ranganathan, *Uber*
D. Sculley, *Google*
Tal Shaked, *Lyft*
Swaminathan Sundararaman
Nisha Talagala, *Pyxeda AI*
Sandeep Uttamchandani, *Intuit*
Joel Young, *LinkedIn*

Message from the OpML '19 Program Co-Chairs

Welcome to OpML 2019!

We are very excited to launch the inaugural USENIX Conference on Operational Machine Learning (OpML). As Machine Learning (and its variants Deep Learning, Reinforcement Learning, etc.) make ever more inroads into every industry, new challenges have emerged regarding how to deploy, manage, and optimize these systems in production. We started OpML to provide a forum where practitioners, researchers, industry, and academia can gather to present, evaluate, and debate the problems, best practices, and latest cutting-edge technologies in this critical emerging field. Managing the ML production lifecycle is a necessity for wide-scale adoption and deployment of machine learning and deep learning across industries and for businesses to benefit from the core ML algorithms and research advances.

The conference has received strong interest, with 61 submissions spanning both academia and industry. Thanks to the hard work of our Program Committee, we have created an exciting program with thirty technical presentations, two keynotes, two panel discussions, and six tutorials. Each presentation and paper submission was evaluated by 3–5 PC members, with the final decisions made during a half-day online PC meeting.

We would like to thank the many people whose hard work made this conference possible. First and foremost, we would like to thank the authors for their incredible work and the submissions to OpML '19. Thanks to the Program Committee for their hard work in reviews and spirited discussion (Jeremy Barnes, Fei Chen, Mike Del Balso, Sindhu Ghanta, Sean Grullon, Neoklis Polyzotis, Jennifer Prendki, Suresh Raman, Marius Seritan, Sarah Sirajuddin, Eno Thereska, Boris Tvaroska, and Todd Underwood). Many thanks to Sindhu Ghanta for serving as tutorials chair and Swami Sundararaman for his many contributions during the early days of the conference. Thank you to Joel Young and Sandeep Uttamchandani for organizing the two panels. We would also like to thank the members of the steering committee for their guidance throughout the process (Nitin Agrawal, Eli Collins, Casey Henderson, Robert Ober, Jairam Ranganathan, D. Sculley, Tal Shaked, Swaminathan Sundararaman, Sandeep Uttamchandani, and Joel Young). Finally, we would like to thank Casey Henderson and Kurt Andersen of USENIX for their tremendous help and insight as we worked on this new conference, and all of the USENIX staff for their extraordinary level of support throughout the process.

We hope you enjoy the conference and proceedings!

Best Regards,
Bharath Ramsundar, *Computable*
Nisha Talagala, *Pyxeda AI*

OpML '19: 2019 USENIX Conference on Operational Machine Learning
May 20, 2019
Santa Clara, CA, USA

Production Experiences and Learnings

Handling Heterogeneity, Distribution, and Scale

Measuring and Diagnosing Production ML

Optimizing and Tuning

Solutions and Platforms

Opportunities and Challenges Of Machine Learning Accelerators In Production

Rajagopal Ananthanarayanan, Peter Brandt, Manasi Joshi, Maheswaran Sathiamoorthy, *Google, Inc.*[1]
{ananthr, pbrandt, manasi, nlogn}@google.com

Abstract

The rise of deep learning has resulted in tremendous demand for compute power, with the FLOPS required for leading machine learning (ML) research doubling roughly every 3.5 months since 2012 [1]. This increase in demand for compute has coincided with the end of Moore's Law [2].

As a result, major industry players such as NVIDIA, Intel, and Google have invested in ML accelerators that are purpose built for deep learning workloads.

ML accelerators present many opportunities and challenges in production environments. This paper discusses some high level observations from experience internally at Google.

1. Products Enabled by ML Accelerators

ML accelerators have had transformational impact on consumer tech products. Many of the recent AI-inspired products developed across the industry would not have been possible without gains in compute via ML accelerators. Notable examples at Google include computational photography features in Photos and Lens, breakthroughs in Translate quality, Gmail SmartCompose, and improvements to Search and Assistant [3, 4]. Similarly, ML accelerators have powered neural architecture search [15, 20] with hyperparameter exploration to pick the best of the breed of large set of models for a given task.

DeepMind's WaveNet model is particularly illustrative [5]. WaveNet enabled a dramatic jump in text-to-speech quality, which for the first time approached truly human-sounding voice. However, the initial version of this model took 1 second to generate just .02 seconds of audio. Through optimizations enabled by Google's TPU ML accelerators, it was possible to achieve a 1000X speed improvement in audio generation and to launch both in Assistant and as a Cloud product offering [6].

ML accelerators have led to the launch of new large scale compute products e.g. NVIDIA's DGX-2 2 petaFLOPS system and Google's TPU v3 pod 100 petaFLOPS system.

2. Overview of ML Accelerator Specialization

ML accelerator chips and the systems which incorporate them are characterized by a variety of specializations compared to general purpose CPUs and distributed systems [2, 13]. These specializations have led to order of magnitude gains in performance and cost [16], and in turn led to significant breakthroughs in AI research e.g. AmoebaNet [17], AlphaGo [18], and BERT [19].

Below, we summarize key specializations tailored to deep learning, encompassing both supervised and unsupervised learning with neural networks (NN) [10].

2.1. Instruction Sets

The main instructions for an ML accelerator implement linear algebra operations such as matrix multiplication and convolutions. Supported data types allow variable precision tailored for deep learning workloads such as bfloat16 [11] and quantized or low-precision arithmetic [2, 13], leading to advantages in memory use and power savings.

2.2. Memory Hierarchy

ML accelerator instructions operate over block-oriented data to fully utilize memory and computation capacity. The memory hierarchy consists of on-chip buffers, on-board high bandwidth memory to efficiently feed data, and host memory to hold state across multiple ML accelerators.

2.3. Host Systems and Networks

To enable access to file systems for input/output, debugging and development workflows, language runtimes, and general purpose computing stack, ML accelerators are connected to a host CPU system. Hosts connect to each other through networks such as Gigabit Ethernet.

ML accelerators connect to hosts through off-the-shelf networking such as PCIe. Accelerator boards also incorporate customized high speed interconnects that connect multiple cores on and across boards. This allows for fast synchronization of state, e.g. by using AllReduce.

3. Software Stack Design

Software stacks for ML accelerators generally strive to abstract away hardware complexity from developers. However, the supported ops, data types, and tensor shapes can differ greatly across hardware platforms and create limitations. These differences can render parts of a model's architecture unsuitable for accelerators. It can be challenging to adapt models trained on one platform, e.g. CPU, to run on a different platform for production inference e.g. TPUv1 or mobile devices. Many bespoke solutions exist, but a good general purpose approach remains an active area of research, including compilers and runtimes that abstract away hardware details [7]. In practice, API design emphasizing helpful error messages greatly improves developer experience and enables broader adoption.

[1]Alphabetical by last name

4. Developer Experience

Model developers want to accelerate training and inference across a variety of models. Below we summarize a few key considerations in porting these computations to accelerators.

4.1 Model Suitability and Decomposition

Operations used by a model must be implemented using the instruction set of the accelerator e.g. to launch CUDA kernels. For a modeler, it is a crucial first step to know which of a model's ops are not supported on the accelerator and whether alternatives exist. Beyond compatibility, it is also important to consider the suitability of ops to run the accelerator (e.g. matmuls) vs. the host CPU (e.g. I/O).

A common decomposition is to place the input ops on the host CPU, with its access to the operating system stack, including file systems, and feed the data to the accelerator. APIs such as tf.data enable this decomposition [25, 26].

4.2 Batch Sizes and Learning Rates

Large batch sizes help to fully exploit the data parallelism available in accelerators. However, increasing the batch size without additional tuning may increase the out-of-sample error [12]. Hyper-parameter tuning, and warm-up techniques where learning rate is slowly increased, may be necessary to obtain quality comparable to lower batch sizes.

4.3 Toolchain - Reproducibility, Performance, and Tests

For a model developer, A/B diff tools integrated into the workflow are essential to compare metrics around model convergence (e.g. accuracy, recall, per batch weight distribution at every step of training) and performance (e.g. latency, throughput, resource utilization). The diff tools can quantify model prediction equivalence between CPU and accelerator based models. Comparing two model versions both using accelerators is important to track benefits and trade offs between cost, speed, and utilization. Finally, continuous quality tests and performance benchmarks across models must be used to gate models rolling into production.

4.4 Other Considerations

Contemporary large models deployed on multiple computers use asynchronous stochastic gradient descent (SGD) [12] to remain efficient on loosely coupled clusters found in data centers. With dedicated high performance interconnects, an accelerator-based system can use Synchronous SGD, which can be beneficial in terms of accuracy [10].

Other performance optimizations include support for batching, switching between model versions, model multi-tenancy to drive higher throughput for inference, and optimizing lookups of variables in the model graph from host CPUs to reduce per query cost.

5. Production Deployment

5.1 System Design for Balanced I/O and Compute

It is important to pay attention to bottlenecks that can creep into various stages of the training pipeline. For example, we need to ensure that the CPU host(s) connected to the ML accelerators can perform data processing, shuffling, transformation, etc. at a high throughput. If any of these stages is slow, the entire pipeline will be slow.

5.2. Diverse Hardware Pool Utilization

Traditionally, large-scale data processing used algorithms such as MapReduce [8] on large clusters of fungible, commodity hardware sharing the x86 architecture [13].

Accelerators add significant heterogeneity into data center environments. Ensuring efficient use of diverse hardware pools to achieve maximum value for an organization is an open problem. Implementing a dominant resource fairness policy [14] works well in practice for some training workloads. However, complications arise while considering data and traffic proximity, inference latency, and query cost.

5.3. Resource Planning

Timelines for designing new hardware and deploying it to data centers often stretch over several years. Pricing and accessibility to resources can significantly determine scale of adoption and benefit to the organization. Forecasting the future ML compute needs is uncertain, and driven by research progress, e.g. new computationally intensive models such as BERT [19]. When trends are accurately reflected in ASIC design and data center planning, it can drive enormous performance improvements [13].

6. Future Work

Several trends point to areas that will be increasingly important for accelerators over the next few years.

Multitask learning [21, 22], where a single model performs multiple tasks (e.g. CTR prediction and user like prediction in a recommender system), is growing in popularity. Today, the number of tasks is limited and generally related. Models with orders of magnitude more tasks of greater diversity are an active area of research [23]. Such models will consume more data of wider variety, posing I/O challenges. It may be necessary to distill [24] such giant models and leverage ML accelerators to make production serving possible.

Transfer learning is a related approach where pre-trained models are fine tuned to create many different models for different tasks, often with significant quality wins. If this grows in prevalence, it will dramatically increase the need for ML software stacks to consistently run inference on

model architectures across diverse HW platforms, regardless of the HW platform used to train the models.

References

[1] AI and Compute. https://blog.openai.com/ai-and-compute/

[2] Jouppi, N. P., Young, C., Patil, N., & Patterson, D. (2018). A domain-specific architecture for deep neural networks. *Communications of the ACM*, 61(9), 50-59.

[3] Google I/O 2018 Keynote, Sundar Pichai https://www.youtube.com/watch?v=ogfYd705cRs

[4] Cloud TPU. https://cloud.google.com/tpu/

[5] Oord, A.V.D., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. and Kavukcuoglu, K., 2016. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

[6] Cloud Text to Speech. https://cloud.google.com/blog/products/gcp/introducing-cloud-text-to-speech-powered-by-deepmind-wavenet-technology

[7] Compilers for ML. https://www.c4ml.org/

[8] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.

[9] Jouppi, N.P. *et al* (2017), June. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*(pp. 1-12). IEEE.

[10] Ian Goodfellow, Yoshua Bengio, Aaron Courville, (2016) Deep Learning, The MIT Press.

[11] Bfloat16 format. https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

[12] Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*.

[13] Wang, E., Davis, J.J., Zhao, R., Ng, H.C., Niu, X., Luk, W., Cheung, P.Y. and Constantinides, G.A., 2019. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *arXiv preprint arXiv:1901.06955*.

[14] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., & Stoica, I. (2011, March). Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In NSDI (Vol. 11, No. 2011, pp. 24-24).

[15] Using Evolutionary AutoML to Discover Neural Network Architectures. https://ai.googleblog.com/2018/03/using-evolutionary-automl-to-discover.html

[16] DAWNBench v1 Deep Learning Benchmark Results. https://dawn.cs.stanford.edu/2018/04/30/dawnbench-v1-results/

[17] Real, E., Aggarwal, A., Huang, Y. and Le, Q.V. (2018). Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*.

[18] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. and Chen, Y., 2017. Mastering the game of go without human knowledge. *Nature*, *550*(7676), p.354.

[19] Devlin, J., Chang, M.W., Lee, K. and Toutanova, K., (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[20] Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

[21] Caruna, R. (1993). Multitask learning: A knowledge-based source of inductive bias. In *Machine Learning: Proceedings of the Tenth International Conference*, 41–48.

[22] Caruana, R. (1998). Multitask learning. In *Learning to learn*. Springer. 95–133.

[23] Kaiser, L., Gomez, A.N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L. and Uszkoreit, J., (2017). One model to learn them all. *arXiv preprint arXiv:1706.05137*.

[24] Hinton, G., Vinyals, O. and Dean, J., (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[25] Keras on TPUs in Colab. https://medium.com/tensorflow/tf-keras-on-tpus-on-colab-674367932aa0

[26] Data Input Pipeline Performance https://www.tensorflow.org/guide/performance/datasets

Accelerating Large Scale Deep Learning Inference through DeepCPU at Microsoft

Minjia Zhang, Samyam Rajbandari, Wenhan Wang, Elton Zheng,
Olatunji Ruwase, Jeff Rasley, Jason Li, Junhua Wang, Yuxiong He
Microsoft AI & Research
{minjiaz,samyamr, wenhanw, Elton.Zheng, olruwase, Jeff.Rasley, jasol, junhuaw, yuxhe}@microsoft.com

Abstract

The application of deep learning models presents significant improvement to many Microsoft services and products. In this paper, we introduce our experience and methodology of developing and applying the DeepCPU library for serving DL models in production at large scale with remarkable latency improvement and infrastructure cost reduction. We describe two ways to use the library, through customized optimization or framework integration, targeting different scenarios.

1 Introduction

Deep learning (DL) sits at the core of many essential products and services at Microsoft, such as web question and answering, web relevance ranking, advertising, language modeling, text translation, and conversational bot [2, 3, 4, 5, 7, 8]. Many of these services are deployed in large scale, supporting millions of users and billions of requests.

Such large scale DL inference faces *threefold* challenges to deploy a trained DL model at production. First, users expect to receive an inference result with low latency. The serving system needs to provide adequate quality of service, expressed as latency SLA (service level agreement), which is often a few milliseconds [11]. In practice, DL models are computationally expensive, incurring long latency, e.g., ranging from hundreds of milliseconds to seconds, that blocks their deployment [12, 23]. Second, when the volume of requests exceeds the capacity of a single server, the DL service must scale horizontally. An efficient serving system reduces the required replications and save thousands of machines and millions of cost. Finally, these constraints come together with restriction on the deployment infrastructure. In particular, it is strongly preferable to use existing commodity hardware, one main reason being the easier maintenance of the infrastructure and the agility of deployment.

To tackle these challenges, we foremost rely on a large amount of CPUs for serving DL models and adopt a co-development methodology called *SLT (scenario, library, and technique)* to make the best use of the CPU resource for business critical scenarios while accelerating the iteration cycle of deployment and optimization. In this paper, we present the SLT methodology and how it leads to DeepCPU, a DL inference library, which is deployed in production for many services on thousands of servers, and is tailored for DL scenarios with large number of users. We show two ways of applying DeepCPU, either through customized end-to-end optimized DL serving solution or low-level interface integration into frameworks such as TensorFlow [9] and ONNX [6]. Our evaluation on production models demonstrates the ability of the DeepCPU library that addresses latency SLA violation problem on a single server and also improves the throughput so that the DL service scales horizontally.

Scenarios	Services	Major components
Deep feature	Encoder model	GRU, Conv
	Embedding model	Stacked Bidir RNN, MLP, Attention
Web Q&A	MRC model A	Bidir RNN, Attention
	MRC model B	Bidir LSTM, Stacked LSTM, Conv, MLP, Attention
	MRC model C	Bidir GRU, Conv, MLP, Attention
Similarity ranking	Ranking model A	RNN encoder/decoder, Attention
	Ranking model B	GRU, Conv, MaxPool, Scoring
Query processing	Query rewriting	RNN encoder/decoder
	Query tagging	Stacked RNN

Table 1: DL scenarios and corresponding models.

2 Scenario, Library, and Technique (SLT)

This section highlights the SLT methodology. Section 2.1 describes DL inference scenarios that are of interest in our production. Section 2.2 introduces what DeepCPU library is. Section 2.3 shows our performance optimization techniques.

2.1 Major Deep Learning Scenarios

We start the SLT methodology with a bird's-eye view of some major Microsoft scenarios that leverage DL models from the standpoint of latency SLA and resource consumption. Table 1 shows some of the scenarios, services, and model components.

Deep feature uses a DL model to encode entities (e.g., text) into deep descriptors (i.e., vectors). The generated vectors are used for semantic understanding of downstream models.

Web Q&A addresses web question-and-answering scenario. It uses a machine reading comprehension model to generate a high quality answer based on the question in a query.

Similarity ranking reranks the top-N text passages for each query based on their semantic similarity to the query.

Query rewriting performs sequence-to-sequence rewriting to map a query to some other query (well corrected, altered, paraphrased) at runtime and uses this query to surface more and better documents for the query.

Query tagging identifies entities in the query to enable more precise matching with documents.

These are just a few examples. There are many more services that leverage DL models in various forms. These services often face challenges from latency, cost, or both. For example, for MRC models, latency is often a big challenge. MRC model A has serving latency of 200ms using TensorFlow [9] but requires to meet 10ms latency SLA for shipping.

DL services	Original Latency	Latency Target	Optimized Latency	Latency Reduction	Throughput Improvement
Encoder model	∼29ms	10ms	5.4ms	5X	5X
MRC model A	∼100ms	10ms	9ms	>10X	>10X
MRC model B	∼107ms	10ms	4.1ms	>20X	>50X
MRC model C	∼45ms for batch size 1	10ms	<8.5ms for batch size 20	11X	>100X
Ranking model A	10∼12ms for batch size 1	6ms	<6ms for batch size 33	>6X	>30X
Ranking model B	10ms for batch size 1	6ms	<5ms for batch size 150	>10X	>100X
Query rewriting	51ms	5ms	4ms	>10X	>3X
Query tagging	9∼16ms	3ms	0.95ms	10X	>10X
NMT model	29ms	10ms	5.8ms	5X	5X

Table 2: Optimization results with and without DeepCPU on production models.

For similarity ranking models, cost is often a big concern. Ranking model A takes 10ms to serve a query with batch size 1 on a single server, whereas the latency SLA is 5ms for batch size 150. This is not scalable because even a fan-out solution requires thousands of machines to serve the large volume of request traffic.

2.2 Highly Reusable Library

Table 1 also shows the DL components each model has. Those components are divided into three categories.

RNN family includes GRU/LSTM cell and sequence, unidirectional/bidirectional RNN, and stacked RNNs [10, 13].

Fundamental building blocks and common DL layers includes matrix-multiply kernels, high-way network [20], max pooling layer [16], Conv layer [15], MLP layer [18], etc.

DL layers for machine reading comprehension and conversation models includes variety of attention layers [14, 19], seq2seq decoding with beam search [21], etc.

We build DeepCPU, a library of these components as building blocks with customized optimization. We find that these components are highly reusable and allow faster implementation and decreased development cost to support new scenarios. As an example, it takes < 200 lines of C++ code for running a Seq2Seq model end-to-end with the library.

2.3 Performance Optimization Techniques

Not only we support the library, but we also offer optimization techniques to optimize different components. We perform three large categories of optimizations:

Intra-op optimizations. We provide i) more efficient matrix computation by combining Intel MKL [1] with customized cache-aware kernel computation to handle, large matrix computation, as well as small or tall-and-skinny matrix-multiplication. ii) optimized common activation functions using continued fraction expansion [22], efficient parallelization, and SIMD vectorization.

Inter-op optimizations. We perform operation fusion which fuses point-wise operation to avoid multiple scans of data and reduced data movement overhead.

Parallelism, scheduling, and affinity. The parallelism, load balancing, and scheduling order are also critical to the performance of DL optimization on multicore CPU. Existing frameworks such as TensorFlow are designed to handle generic DAG, which can lead to suboptimal parallelism decisions and cannot control per-op parallelism, while we consider the characteristics of the workload and perform global optimization by looking at model structure. We also pin application threads to physical cores and make DL computation NUMA-aware and socket-aware to avoid expensive context switching and cross-socket communication overhead.

3 How is DeepCPU Utilized?

DeepCPU is currently released as C++ SDK to first party users. There are two approaches to use the library.

Customized optimization. This approach requires rewriting the model runtime using the DeepCPU library. After then we tune the performance such as thread settings, targeting at obtaining the ultimately optimized performance, because at large scale, every possible bit of hardware optimization space leads to major improvements. This approach requires interaction with the model developer and requires some development efforts if the model changes drastically. To achieve improved performance with less development work, we also integrate DeepCPU into existing DL frameworks.

Framework integration. We replace frequently used and costly operators, such as LSTM, GRU, Conv2D, Attention, with DeepCPU 's high-performance implementations in TensorFlow runtime. This approach targets framework users directly, and it allows users to use existing frameworks to develop models while taking only a minimal amount of work to switch the operators to take the advantage of DeepCPU. Meanwhile, we are closely working with ONNX team to power ONNX runtime [6] with DeepCPU technology, which allows frameworks that support ONNX IR, such as PyTorch [17], to also benefit from DeepCPU.

For new scenarios and models, we often encourage to try the framework integration approach first, which allows fast deployment if that already gives satisfying performance results (e.g., meeting latency SLA). Otherwise, we apply customized optimization to further boost the performance.

4 Evaluation results

Table 2 shows a list of models we have optimized with DeepCPU, with both latency and throughput improvement in comparison with TensorFlow on a server with two 2.20 GHz Intel Xeon E5-2650 V4 processors, each of which has 12-core with 128GB RAM. Overall, we see 5–20 times latency improvement, which helps to change the model status from non–shippable to shippable. Meanwhile, we have achieved up to 100 times throughput improvement and cost reduction. These models have been running in production for the last two years on thousands of servers.

References

[1] Intel(R) Math Kernel Library. https://software.intel.com/en-us/mkl.

[2] Internet-Scale Deep Learning for Bing Image Search. https://blogs.bing.com/search-quality-insights/May-2018/Internet-Scale-Deep-Learning-for-Bing-Image-Search. Accessed: 27-January-2019.

[3] Machine Reading. https://www.ailab.microsoft.com/experiments/ef90706b-e822-4686-bbc4-94fd0bca5fc5. Accessed: 27-January-2019.

[4] Machine Reading at Scale – Transfer Learning for Large Text Corpuses. https://blogs.technet.microsoft.com/machinelearning/2018/10/17/machine-reading-at-scale-transfer-learning-for-large-text-corpuses/. Accessed: 27-January-2019.

[5] Microsoft is teaching systems to read, answer and even ask questions. https://blogs.microsoft.com/ai/microsoft-is-teaching-systems-to-read-answer-and-even-ask-questions/. Accessed: 27-January-2019.

[6] Open neural network exchange format (ONNX). https://github.com/onnx/onnx. Accessed: 27-January-2019.

[7] Towards More Intelligent Search: Deep Learning for Query Semantics. https://blogs.bing.com/search-quality-insights/May-2018/Towards-More-Intelligent-Search-Deep-Learning-for-Query-Semantics. Accessed: 27-January-2019.

[8] What's New in Deep Learning Research: Microsoft Wants Machines to Understand What They Read. https://medium.com/@jrodthoughts/whats-new-in-deep-learning-research-microsoft-wants-machines-to-understand-what-they-read-ebe61e1853a5. Accessed: 27-January-2019.

[9] Martín 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, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, 2016.

[10] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555, 2014.

[11] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication*, SIGCOMM '13, pages 159–170, 2013.

[12] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 31:1–31:15, 2018.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[14] Rudolf Kadlec, Martin Schmid, Ondrej Bajgar, and Jan Kleindienst. Text Understanding with the Attention Sum Reader Network. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL '16*, 2016.

[15] Yoon Kim. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*, pages 1746–1751, 2014.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012*, NIPS '12, pages 1106–1114, 2012.

[17] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration.(2017). *https://github.com/pytorch/pytorch*, 2017.

[18] Dennis W Ruck, Steven K Rogers, Matthew Kabrisky, Mark E Oxley, and Bruce W Suter. The multilayer perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks*, 1(4):296–298, 1990.

[19] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. arXiv preprint arXiv:1611.01603, 2016.

[20] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway Networks. arXiv preprint arXiv:1505.00387, 2015.

[21] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014*, NIPS '14, pages 3104–3112, 2014.

[22] AJ Van der Poorten. Continued fraction expansions of values of the exponential function and related fun with continued fractions. *Nieuw Archief voor Wiskunde*, 14:221–230, 1996.

[23] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, Boston, MA, 2018. USENIX Association.

MLOp Lifecycle Scheme for Vision-based Inspection Process in Manufacturing

Junsung Lim, Hoejoo Lee, Youngmin Won, Hunje Yeon
Samsung Research
{junsung.lim, hoejoo.lee, ymin.won, hunje.yeon}@samsung.com

Abstract

Recent advances in machine learning and the proliferation of edge computing have enabled manufacturing industry to integrate machine learning into its operation to boost productivity. In addition to building high performing machine learning models, stakeholders and infrastructures within the industry should be taken into an account in building an operational lifecycle. In this paper, a practical machine learning operation scheme to build the vision inspection process is proposed, which is mainly motivated from field experiences in applying the system in large scale corporate manufacturing plants. We evaluate our scheme in four defect inspection lines in production. The results show that deep neural network models outperform existing algorithms and the scheme is easily extensible to other manufacturing processes.

1 Introduction

Machine learning(ML) have begun to impact various industrial fields and manufacturing is no exception. Manufacturers, in preparation for the smart manufacturing era to come, aim to improve their competitiveness by adapting new technologies that excel product quality, cut down production cost and reduce lead time in production [7]. Manufacturing industry is an attractive field for ML Operations(MLOps) for number of reasons. First, a huge volume of data is generated, forming the foundation for source of learning. Secondly, trivial and repeated tasks in production process opens up opportunities for ML models. For instance, consider a defect inspection task in which product surfaces are visually checked for scratches by a human inspector. While the task itself is trivial, thus susceptible to human errors, it is difficult to express a good set of rules for scratch detection. Given the recent advancement in deep neural network(DNN), MLOps have become natural selection for such tasks.

MLOps in production is more than just training and running ML models. Despite large volume of raw data collected, it needs to be cleaned and labeled to use them as a ML training dataset. Test data are generated from multiple devices on network edge and thus running inference on a single server is infeasible due to high latency caused by data communication and inference. Also, a use of off-premise services is not proper as every manufacturing data is confidential and should be stored securely on premise. Last but not least, there are multiple stakeholders with different roles in production process and thus, require different tools at each stage in MLOp lifecycle.

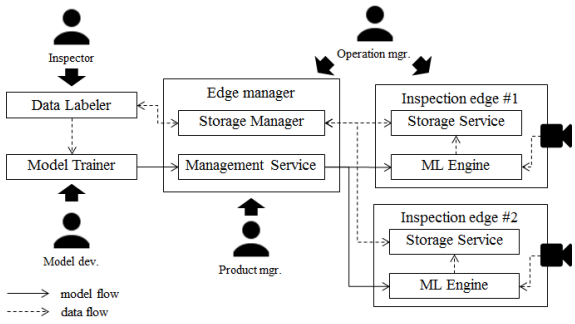


Figure 1: Overall system architecture of the proposed scheme with multiple stakeholders. A circular lifecycle is formed among the components.

In this paper, we propose a MLOp lifecycle scheme for vision-based inspection systems in manufacturing. Figure 1 describes overall architecture and components required for in-factory operations, ranging from data collection, ML model development and deployment on multiple edge devices. Based on the scheme, we developed a MLOp lifecycle solution called CruX. We have successfully set up CruX in Samsung Electronics' smartphone and home appliance plants for scratch, dent and missing part detection. Four DNN models of three different tasks(one-class adversarial net, multi-class classification, and object detection) are trained and deployed to a total of 158 edge devices for inspection. Compared to the existing rule-based algorithms, models achieved at least 32.8% improvement in defect detection accuracy and all inferences at edge took less than 2 seconds per image on CPU.

2 Related Work

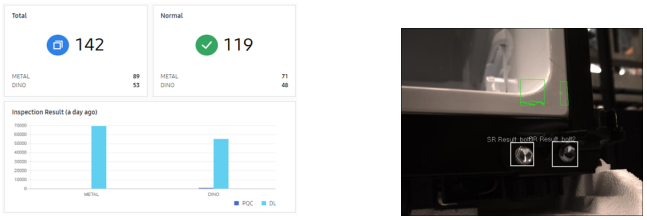
With the popularity of ML, a number of model versioning and serving solutions are available. Data Version Control [1], ModelDB [10] and ModelChimp [2] provide ML model and data pipeline versioning. These solutions, however, require model developers to control versions by either extending existing ML code or setting up extra infrastructure.TensorFlow Serving [8] is a solution to serve TensorFlow model, but it requires models to be accessible from its own file system, leaving the challenge of deploying the model across physically separated edge devices. Complex factors of real field requirements such as different stakeholders in the lifecycle, deployment needs, management and controllability of ML models on multiple edge devices, call for a new operational scheme in the manufacturing industry.

3 Proposed Scheme

We propose a MLOp lifecycle scheme for vision inspection systems, in which four key stakeholders and five components are identified and defined as shown in Figure. 1.

Raw image data are captured by camera which is usually located at the conveyor belt. While some of the images can be annotated by a non ML-domain expert (e.g. identify screw(s) from an image), some are not (e.g. classify scratches by type). Due to this reason, Data Labeler is designed and used by inspectors on site. An intuitive user experience is important as we do not want inspectors spending more time annotating than inspecting the product. Model developers use Model Trainer to train and test DNN models from annotated data. Model Trainer provides a function to train DNN models with different set of hyper-parameters to identify the best hyper-parameter set for the model. The trained model is then uploaded to Edge manager for configuration before deployment. We found this step to be important in production because no edge (or the inspected product) is the same. Model configurations, such as threshold, is adjusted per edge and deployed to edges under the supervision of operation manager. As the inspection continues, statistics are collected and visualized to the product manager.

All the components are modular but interconnected. This is important because it enables the process of training, deploying and running model possible through a single graphical user-interface without having to make any code-level changes.



(a) Edge manager (b) Inference result

Figure 2: (a) Manager monitors inspection status and deploys models to edges. (b) Inference result where detected objects(bolt) are located in white bounding boxes.

4 Evaluation

We implemented the proposed scheme called *CruX*, and applied in two different plants. Back-end components are developed in Python, Java and Go. Data are exchanged among the components using REST APIs and message queues. The proposed scheme supports three different DNN models, namely multi-class classification(ResNet50 [6]), one-class generative adversarial network(GAN [5]) and object detection(YOLO [9]). All are implemented with TensorFlow and fine-tuned from ImageNet [4] pretrained weights. Figure 2 shows a web-based GUI that is provided to end-users. Edge manager and Inspection edge run Windows 7 64bit with 8GB RAM, 2.60GHz CPU and no GPUs.

Table 1 shows the results on production lines. In prior to this, rule-based algorithms [3] are used to detect scratches, dents and missing parts. We noticed that the rule-based algorithms are very sensitive to small changes in data (e.g. image orientation and brightness) and difficult to update. On the other hand, DNN models showed higher defect detection accuracy, outperforming previous method by 32.8% 92.8%. All four production lines required inspection time to not exceed 3 seconds.

5 Conclusion

In this paper, we propose a MLOp scheme for vision inspection in manufacturing. We identify four key stakeholders and five components across in realizing MLOp lifecycle. We successfully applied it on four production fields of smartphone and home appliance plants. ML models trained and deployed by the scheme outperform existing inspection systems, and we aim to update the operation automated as the future work.

Table 1: Defect inspection results on four production lines (*: Defection detection accuracy).

Inspection area	Edges deployed	DNN model (Backbone)	DDA* improvement	Avg. inference time
Scratch (smartphone)	88	Multi-class (ResNet50)	32.8%	760 ms
Dent (smartphone)	52	One-class (GAN)	40.0%	998 ms
Missing part (refrigerator)	9	Object detection (YOLO)	92.8%	1416 ms
Missing part (washing machine)	9	Object detection (YOLO)	85.6%	1632 ms

References

[1] Data science version control system, 2019.

[2] Experiment tracking | modelchimp, 2019.

[3] Daniel Lélis Baggio. *Mastering OpenCV with practical computer vision projects*. Packt Publishing Ltd, 2012.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[5] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] Hyoung Seok Kang, Ju Yeon Lee, SangSu Choi, Hyun Kim, Jun Hee Park, Ji Yeon Son, Bo Hyun Kim, and Sang Do Noh. Smart manufacturing: Past research, present findings, and future directions. *International Journal of Precision Engineering and Manufacturing-Green Technology*, 3(1):111–128, 2016.

[8] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[9] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, July 2017.

[10] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Model db: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

Shooting the moving target: machine learning in cybersecurity

Ankit Arun
PatternEx
San Jose, CA

Ignacio Arnaldo
PatternEx
San Jose, CA

Abstract

We introduce a platform used to productionize machine learning models for detecting cyberthreats. To keep up with a diverse and ever evolving threat landscape, it is of paramount importance to seamlessly iterate over the two pillars of machine learning: data and models. To satisfy this requirement, the introduced platform is modular, extensible, and automates the continuous improvement of the detection models. The platform counts more than 1000 successful model deployments at over 30 production environments.

1 Introduction

The cybersecurity community is embracing machine learning (ML) to transition from a reactive to a predictive strategy for threat detection. In fact, most cyberthreats exhibit distinctive activity patterns, allowing practitioners to leverage ML to accurately identify attacks. However, while there is a plethora of research on detecting attacks using ML [1], the findings are rarely deployed in real-world solutions.

The limited adoption of ML in cybersecurity is explained by the following challenges: a) the diversity of the threat landscape [2] requires the creation and deployment of a large number of models; b) threats keep evolving to bypass defenses, requiring detection models to be frequently updated.

To alleviate model management effort and to simultaneously tackle the *moving target* problem, we present a scalable, extensible, and automated machine learning platform designed to keep the detection models deployed in production environments up to date. Our platform is designed to satisfy the following requirements:

1. To maintain and to enable the extension of the datasets required to retrain detection models. Each dataset (one per model) contains examples of a particular attack, as well as a representative sample of benign activity. In this paper, we refer to these datasets as "golden datasets".
2. To support modifications to the modeling strategy (namely the addition of new features), and to update the deployment logic accordingly.
3. To seamlessly deploy updated models in production.
4. To do the aforementioned points in minimal time.

2 Overview of our machine learning platform

Figure 1 shows a schematic representation of our platform. In the following, we briefly describe the different modules.

Golden dataset repository The golden datasets are stored in a repository accessed by threat researchers, data scientists, and ML engineers. The repository is stored in Amazon S3.

Configurable data pipelines To simplify and speed up both data ingestion and changes in the feature extraction logic, we have created a configurable and extensible log parsing and feature computation engine.

The parsing engine relies on Protocol buffers (`protobuf`) messages expressed in plain text to convert raw logs into a structured format. The *Log Parsing Engine* in Figure 1 shows a snippet of the `protobuf` message. The logic needed to extract the fields that make up the structured format is declared in `fields` blocks, each composed of the following parameters:
- `name`: the name of the extracted field
- `display_name`: the display name of the extracted field
- `data_type`: the type of extracted field
- `index`: the relative position of the raw data field(s) needed to extract the new field
- `definition`: the definition of the transformation required to extract the new field, declared as a SQL expression.

With this approach, edits to the extraction and transformation logic correspond to configuration changes rather than changes in the platform codebase. To achieve scalability, we rely on Spark jobs to perform the parsing and extraction logic.

In a similar way, features are also expressed as a `protobuf` messages (as shown in the *Feature Compute Engine* module in Figure 1). The extraction of the features is performed by

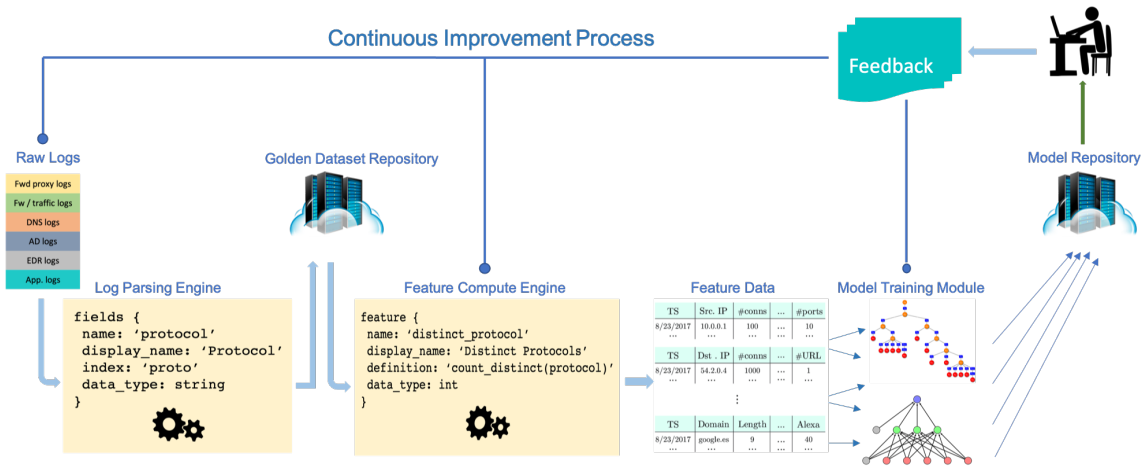


Figure 1: The presented machine learning platform implements a continuous improvement process based on end user feedback to maintain a wide range of cyberattack detection models deployed in production environments up to date.

a Spark job that reads the messages and applies the transformations indicated in the `definition` fields. Note that the definitions are again SQL expressions, and that changes to feature extraction logic (including the addition of new features) entail only the modification of the *feature messages*.

Model training and validation In a nutshell, this module retrieves the newly extracted features and trains machine learning models using the standard machine learning libraries *scikit-learn* and *TensorFlow*[1].

Model repository After training, the models are stored at a central location, making it a one stop shop for all the models.

Model distribution and deployment All the serviced environments share the same parsing and feature computation logic, and periodically pull the models from the repository. This way, the updated models are seamlessly deployed across all the production systems.

3 Continuous improvement process

The threat alerts generated by the deployed models are analyzed by the end users (security analysts or threat researchers working at the serviced environments). As shown in Figure 1, the end users provide feedback, triggering a new model improvement iteration. In the following, we describe the process that takes place when the feedback takes the form of a) new attack or benign examples, b) ideas for new features.

Extending the golden datasets Our threat research team and end users contribute new examples of malicious or benign activities to the existing golden datasets on an ongoing basis. Any time new raw data is contributed, the platform triggers all the steps shown, from left to right, in Figure 1: parsing of the new examples and extension of the appropriate golden dataset, feature extraction, and model retraining and backup in the model repository.

Modifying the modeling strategy We limit the modifications of the modeling to either the addition of new features or the modification of an existing one [2]. As explained in Section 2, in either case the required changes are limited to the edit of configuration files. Any time edits are performed to the feature definition files, the platform triggers the re-extraction of the features for the affected golden datasets, followed by the re-training and distribution of the impacted detection models.

4 Current state of the system

The presented platform currently supports the ingestion of 31 data sources, maintains 27 golden datasets, and counts 70 models readily available for distribution and deployment. As of the day of this writing, the platform has successfully performed more than 1000 model deployments, where each model is updated weekly.

References

[1] Heju Jiang, Jasvir Nagra, and Parvez Ahammad. Sok: Applying machine learning in security-a survey. *arXiv preprint arXiv:1611.03186*, 2016.

[2] MITRE. Adversarial Tactics, Techniques & Common Knowledge. https://attack.mitre.org, 2019.

[3] K. Veeramachaneni, I. Arnaldo, V. Korrapati, C. Bassias, and K. Li. *AI*^{2}: Training a Big Data Machine to Defend. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud*, pages 49–54, April 2016.

[1] We consider that the details of the modeling strategy are out of the scope of this paper. The interested reader is referred to [3]

[2] Motivation: end users are domain experts that are not well-versed in the advantages and drawbacks of the different ML models and training strategies.

Deep Learning Inference Service at Microsoft

Jonathan Soifer, Jason Li, Mingqin Li, Jeffrey Zhu, Yingnan Li, Yuxiong He,
Elton Zheng, Adi Oltean, Maya Mosyak, Chris Barnes, Thomas Liu, Junhua Wang
Microsoft

Abstract

This paper introduces the Deep Learning Inference Service, an online production service at Microsoft for ultra-low-latency deep neural network model inference. We present the system architecture and deep dive into core concepts such as intelligent model placement, heterogeneous resource management, resource isolation, and efficient routing. We also present production scale and performance numbers.

1 Introduction

Over the past couple of years, many services across Microsoft have adopted deep neural networks (DNN) to deliver novel capabilities. For example, the Bing search engine uses DNNs to improve search relevance by encoding user queries and web documents into semantic vectors, where the distance between vectors represents the similarity between query and document [6, 7, 9]. However, due to the computational complexity of DNNs, application-embedded inference and off-the-shelf micro-service offerings don't meet the necessary scale, performance, and efficiency requirements for many of Microsoft's critical production services. These services receive hundreds of thousands calls per second and are often constrained to single-digit millisecond latency budgets. DNNs authored across a spectrum of operating systems and frameworks must be provisioned efficiently on heterogeneous datacenter hardware, such as CPU, GPU, and FPGA. With rapid innovations in DNN architectures, the system must be extensible and agile by supporting fast model validation, deployment, and proper version control. Deep Learning Inference Service (DLIS) is a dedicated platform to address these requirements, and now serves as the inference backend for many teams across Microsoft such as web search, advertising, and Office intelligence. At present, DLIS is handling three million inference calls per second, served from tens of thousands of model instances, and deployed in more than 20 data centers world-wide.

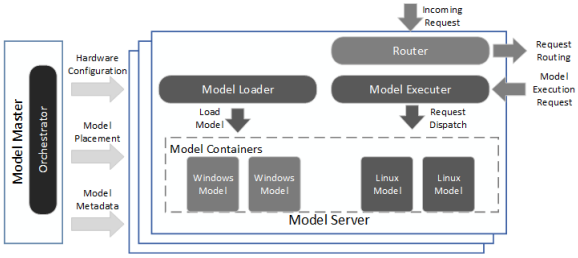


Figure 1: DLIS Architecture

2 System Overview

Figure 1 shows an overview of DLIS and its key components. Model Master (MM) is a singleton orchestrator responsible for intelligently provisioning model containers onto one or more servers by factoring in model requirements and hardware resources. Model Servers (MS) are the server unit and can number in the thousands. They have two roles: routing and model execution. MS receives an incoming request from a client and efficiently routes it to another MS hosting an instance of the requested model. The MS receiving the request from the routing server then executes the request with low-latency. These three core functionalities of provisioning, routing, and model execution will be discussed in detail in sections 3, 4, and 5. In addition to the features discussed in this paper, MS is flexible. It runs on both Windows and Linux and supports multiple orchestrators outside of MM. These include YARN and Kubernetes [1, 3].

3 Intelligent Model Placement

The performance of different DNN models varies across hardware. For example, convolutional neural network models are most performant on GPU, while recurrent neural network models often achieve lower latency on FPGA or CPU [5, 8, 10]. DLIS needs to understand different models' requirements and place them efficiently onto matching hardware. This neces-

sitates an intelligent model placement system in the Model Master.

Model Placement. MM has a global view of all servers and their respective hardware and resource availability, which includes CPU instruction sets, number of CPU cores, amount of memory, and number of GPUs, among others. MM is aware of a model's estimated resource usage through a validation test run prior to model deployment. To host an instance of a model, servers must satisfy the following constraints: they must meet the hardware requirements of the model, they must have available resources to host at least one instance, and they must be spread across a certain number of fault domains. Placement is multi-tenant and dynamic. Instances can be hosted with other instances of the same model or a different model. Further, MM reads resource usage at runtime and can decide to move instances to different servers at any time.

Diverse Hardware Management. Specialized hardware such as GPU and FPGA requires proper configuration and management. To support this, DLIS uses a special model called a machine configuration model (MCM). MCMs configure servers at regular intervals. For example, an MCM may run every ten minutes, installing a GPU driver, resetting GPU clock speed, and verifying overall GPU health.

4 Low-Latency Model Execution

DNNs are computationally complex. Different levels of optimization are required to achieve low-latency serving. DLIS supports both system- and model-level optimizations [10]. This section describes the system optimizations, while model optimizations are outside the scope of this paper.

Resource Isolation and Data Locality. For low-latency serving in a multi-tenant environment, data access is localized to take advantage of different cache layers, while resource isolation is used to ensure that model instances do not interfere with each other. To achieve this, MS isolates model instances in containers. Linux models are run in Docker containers, while Windows models are run in custom containers under job objects [2]. DLIS enforces resource isolation in the form of processor affinity, NUMA affinity (when hardware supports it), and memory restrictions. Processor affinity allows model-critical data to stay in the nearest processor caches. NUMA affinity guarantees that a model doesn't have to cross memory banks. Memory restrictions ensure that the model never needs to access disk. Together, they ensures that model instances localize data access with minimal interference from other instances.

Server-to-Model Communication. Container-based isolation leads to a need for efficient communication between server and model. To support this, Linux models are wrapped in custom infrastructure to enable efficient communication over UDP. Windows models are wrapped in custom infrastructure to enable efficient communication over a shared-memory queue. The shared-memory queue provides inter-process communication latencies of less than a few hundred microseconds.

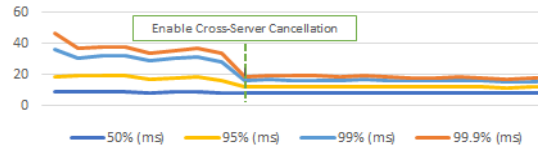


Figure 2: Latency Before and After Enabling Cross-Server Cancellation

5 Efficient Routing

Traffic patterns to model inference at Microsoft come with unique challenges. First, there is frequent burst traffic - many requests in the span of a few milliseconds. In extreme scenarios, each request may be a batch with hundreds of sub-requests. Such burst traffic can lead to many requests being enqueued on the same server. Next, tail model latency is often very near performance SLA. These challenges necessitate MS to route requests with minimal overhead.

Backup Requests and Cross-Server Cancellation. With frequent burst traffic, it is hard to accurately predict each server's load. To compensate, MS router supports backup requests which serves as a second chance if the first request has a risk of missing SLA. Backup requests can be either statically configured (for example, sending a backup request after 5ms) or dynamically configured (for example, sending a backup request at the 95th-percentile model latency). For many low-latency scenarios, backup requests alone are not enough. For example, say an SLA is 15ms, current 95th-percentile model latency is 10ms, and average model latency is 8ms. If backup requests are configured to send at 10ms, the request will almost certainly timeout. However, if the backup request is sent earlier (say at 2ms), the system's load will effectively be doubled. To solve this, MS router supports backup requests with cross-server cancellation [4]. In this mode, MS will send backup requests earlier. When a server dequeues the request, it will notify the other server to abandon that request. For our scenarios, backup requests at 2ms with cross-server cancellation provides the best latency improvement with the least amount of extra computation. With these optimizations, MS routing overhead is less than 1.5ms. Figure 2 shows the nearly 2x latency drop after cross-server cancellation is enabled for a model.

6 Conclusion

We have presented DLIS. It is serving millions of inference calls per second across tens of thousands of model instances. These models run on varying hardware with low overhead and are supporting many production Microsoft services.

References

[1] Apache hadoop yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. 2018.

[2] Docker. https://www.docker.com/. Accessed: 2019-2-12.

[3] Kubernetes. https://kubernetes.io/. Accessed: 2019-2-12.

[4] Jeffrey Dean. Achieving rapid response times in large online services. https://research.google.com/people/jeff/latency.html, 2012. Accessed: 2019-2-12.

[5] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[6] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 2333–2338. ACM, 2013.

[7] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 24(4):694–707, 2016.

[8] Daniel Strigl, Klaus Kofler, and Stefan Podlipnig. Performance and scalability of gpu-based convolutional neural networks. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 317–324, Washington, DC, USA, 2010. IEEE Computer Society.

[9] Hongfei Zhang, Xia Song, Chenyan Xiong, Corby Rosset, Paul Bennett, Nick Craswell, and Saurabh Tiwary. Generic intent representation in web search. In *submission*, 2019.

[10] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 951–965, Berkeley, CA, USA, 2018. USENIX Association.

Towards Taming the Resource and Data Heterogeneity in Federated Learning

Zheng Chai[1], Hannan Fayyaz[2], Zeshan Fayyaz[3], Ali Anwar[4],
Yi Zhou[4], Nathalie Baracaldo[4], Heiko Ludwig[4], Yue Cheng[1]

[1]*George Mason University,* [2]*York University,* [3]*Ryerson University,* [4]*IBM Research–Almaden*

1 Introduction

Traditionally, training machine learning models requires all data to be in the same place accessible to a trusted third party. However, privacy concerns and legislations such as General Data Protection Regulation (GDPR) [16] and Health Insurance Portability and Accountability Act (HIPAA) [14] inhibit transmitting data to a central place resulting in the impossibility of training machine learning models using this traditional technique. Despite these limitations, in some cases data owners would benefit from collaboratively training a model. To address this requirement, very recently *Federated Learning* (FL) has emerged as an alternative way to do collaborative model train models without sharing the training data [12] [17] [18].

In FL, each data owner, *party*, maintains its own data locally and engage in a collaborative learning procedure where only model updates are shared with an aggregator. Note that the aggregator does not have access to the data of any of the parties. Through FL, parties with relatively small datasets can learn more accurate models than they would if they had only used their own data. Examples of such scenario include a large number of individual parties providing personal data to smart phone apps and a relatively small number of competing companies within the same domain training a single model. A concrete scenario where FL has been used to collaboratively train models include Google's key board predictive model [6].

In these scenarios, parties may be very diverse. This diversity largely differentiates FL from traditional distributed learning systems such as [8,11] where a datacenter is available for careful management. Most of the times, the data parties involved in FL training have diversified training sets that may vary in size, computing power, and network bandwidth. These differences impact the FL process as we empirically demonstrate in our experimental section.

In the following, we first overview existing FL approaches. We show that stragglers are not considered by existing techniques. Then, through a preliminary study, we demonstrate the potential impact of stragglers on FL process and finally conclude with a discussion of the research problems.

2 Related Work

Existing FL approaches do not account for the resource and dataset heterogeneities [7,10,12], nor are they straggler-aware. In particular, there are two main approaches in training a FL model: *synchronous* and *asynchronous* FL.

In synchronous FL, a fixed number of data parties are queried in each learning epoch to ensure performance and data privacy. Recent synchronous FL algorithms focus on reducing the total training time without considering the straggler parties. For example, [12] proposes to reduce network communication costs by performing multiple SGD (stochastic gradient descent) updates locally and batching data parties. [7] reduces communication bandwidth consumption by structured and sketched updates. Moreover, [9] exploits randomized technique to reduce communication rounds. FedCS [13] proposes to solve data party selection issue via a deadline-based approach that filters out slowly-responding parties. However, FedCS does not consider how this approach effects the contributing factors of straggler parties in model training. Similarly, [19] proposes a FL algorithm for the use case of running FL on resource constrained devices. However, they do not aim to handle straggler parties and treat all parties as resource constrained. In contrast, we focus on scenarios where resource constrained devices are paired with high resource devices to perform FL.

Most asynchronous FL algorithms work only for convex loss and do not allow parties to drop-out. For instance, [15] provides performance guarantee only for convex loss functions with bounded delay assumption. Similarly, [3, 10] allow uniform sampling of the data parties and provide performance guarantee for convex loss functions. Furthermore, the comparison of synchronous and asynchronous methods of distributed gradient descent [4] suggest that FL should use the synchronous approach, because it is more efficient than the asynchronous approaches [12, 13].

3 Preliminary Study

We conduct an experimental study on AWS EC2 to quantify the impact of resource and dataset heterogeneity on training time of FL. We use a multi-party TensorFlow [2] setup to emulate a FL environment following the configuration settings used in [5], with δ as 0.001, ε as 8, and σ in the Gaussian mechanism as 1.0. We deploy 20 data parties to emulate a randomly picked 100-party FL environment, where each party is running inside of a Docker container. The training process

Test	# of Clients	# of CPUs	CPUs per Client
1	4	16	4
2	4	8	2
3	4	4	1
4	3	1	1/3
5	5	1	1/5

Table 1: Distribution of data parties and CPUs.

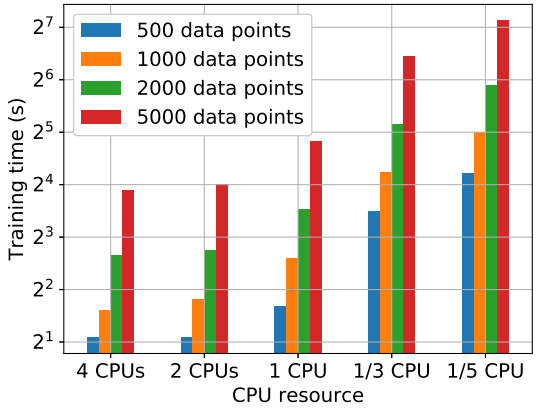


Figure 1: Per-epoch training time different CPU resources and different dataset sizes.

terminates until the accumulated privacy cost exceeds the privacy bound (δ). All the containerized parties are running on a single EC2 virtual machine (VM) instance of `m4.10xlarge` with 40 vCPUs and 160 GiB memory.

We train a CNN (Convolutional Neural Network) model on the MNIST dataset [1], which contains 60,000 28 grayscale images of ten handwritten digits. To emulate a realistic imbalanced party data distribution, we use **Non-IID** in data selection, where each party randomly selects 5 digit categories and then performs the image sampling from these 5 categories. The CNN model consists of two CNN layers and one MaxPooling layer. We use a filter size of 3 for the CNN layers and 2 for the MaxPooling layer. We also add two drop-out layers with a dropping out rate of 0.25 and 0.5, respectively. We use Adadelta for the optimizer, and accuracy as the training evaluation metric. We train the model with 8 learning epoches and measure the training time for each epoch.

Resource Heterogeneity First, we explore the impact of CPU resource heterogeneity on training time. Table 1 summarizes the parties and CPU resource distributions of 5 test groups. We reduce the total amount of CPU resources from Test 1 to 5, and within each test, each party gets an equal share of the available CPU resource. For example, in Test 1, 4 parties get allocated 16 CPU cores with 4 cores per party. Within each test group, we conduct 4 tests each with varied dataset size (sizing from 500 – 5000 data points). Figure 1 plot the average training time of one learning epoch across all data parties for each test. As shown, as the amount of CPU resources allocated to each party increases, the training time gets longer. Reducing the per-party CPU from 4 cores to 2 cores does not impact the training time much, since the CPU bottleneck is relieved with 4 CPU cores.

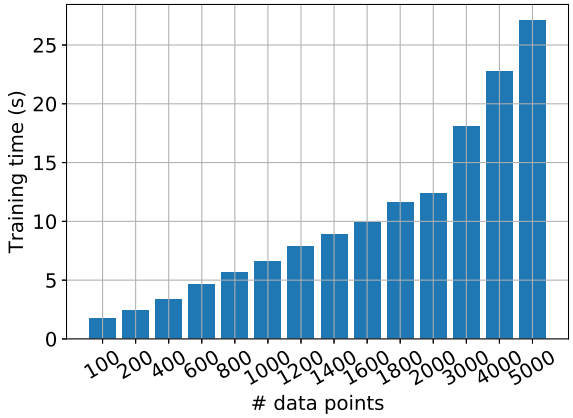


Figure 2: Per-epoch training time with different dataset sizes.

Data Heterogeneity We next quantify the impact of data heterogeneity on training time. We deploy 14 data parties, each owning a different dataset size (varying from 100–5000 data points) but with the same amount of CPU resources (i.e., 1 CPU core), to concurrently training the model. As shown in Figure 2, the training time gets linearly increased as the dataset size gets bigger. This demonstrates that data heterogeneity can significantly impact the FL system's training time.

4 Research Problems and Opportunities

Our preliminary results imply that the straggler issues can be severe under a complicated and heterogeneous FL environment. We believe that our paper will lead to discussions on the following aspects, which are the focus of our ongoing research:

P1: *How to classify parties based on their response time and then use this information for our advantage without affecting the FL process?* A naive solution can lead to misrepresentation of data, because resource constraints may be correlated with quantity/quality of data.

P2: *How to incorporate data of each party in the FL process without worrying about stragglers?* This problem is challenging because we need to make sure we do not over include or exclude certain data parties in FL process. We should be able to provide performance guarantee for general machine learning models and algorithms.

P3: *How to identify drop-out parties and mitigate the effect of drop-out data parties without affecting the ML process?* Existing approaches cannot identify drop-out parties dynamically during the FL process, and no effective method has been proposed to mitigate the information loss when drop-out happens.

Acknowledgments We thank the reviewers for their feedback. This work is sponsored in part by George Mason University, an AWS Cloud Research Grant, and a Google Cloud Platform Research Grant.

References

[1] THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[2] Martín 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, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[3] Inci M Baytas, Ming Yan, Anil K Jain, and Jiayu Zhou. Asynchronous multi-task learning. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 11–20. IEEE, 2016.

[4] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[5] Robin C Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. *arXiv preprint arXiv:1712.07557*, 2017.

[6] Edwin B Kaehler. Dynamic predictive keyboard, July 7 1992. US Patent 5,128,672.

[7] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[8] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *Cidr*, volume 1, pages 2–1, 2013.

[9] Guanghui Lan and Yi Zhou. An optimal randomized incremental gradient method. *Mathematical programming*, pages 1–49, 2017.

[10] Guanghui Lan and Yi Zhou. Random gradient extrapolation for distributed and stochastic optimization. *SIAM Journal on Optimization*, 28(4):2753–2782, 2018.

[11] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, 2014. USENIX Association.

[12] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

[13] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. *arXiv preprint arXiv:1804.08333*, 2018.

[14] Jacquelyn K O'herrin, Norman Fost, and Kenneth A Kudsk. Health insurance portability accountability act (hipaa) regulations: effect on medical record research. *Annals of surgery*, 239(6):772, 2004.

[15] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. Federated multi-task learning. In *Advances in Neural Information Processing Systems*, pages 4424–4434, 2017.

[16] Colin Tankard. What the gdpr means for businesses. *Network Security*, 2016(6):5–8, 2016.

[17] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, and Rui Zhang. A hybrid approach to privacy-preserving federated learning. *arXiv preprint arXiv:1812.03224*, 2018.

[18] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, and Rui Zhang. A hybrid trust model for distributed differential privacy. *Theory and Practice of Differential Privacy*, 2018.

[19] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K. Leung, Christian Makaya, Ting He, and Kevin Chan. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. *CoRR*, abs/1804.05271, 2018.

MPP: Model Performance Predictor

Sindhu Ghanta
ParallelM

Sriram Subramanian
ParallelM

Lior Khermosh
ParallelM

Harshil Shah
ParallelM

Yakov Goldberg
ParallelM

Swaminathan Sundararaman
ParallelM

Drew Roselli
ParallelM

Nisha Talagala
ParallelM

Abstract

Operations is a key challenge in the domain of machine learning pipeline deployments involving monitoring and management of real-time prediction quality. Typically, metrics like accuracy, RMSE etc., are used to track the performance of models in deployment. However, these metrics cannot be calculated in production due to the absence of labels. We propose using an ML algorithm, *Model Performance Predictor* (MPP), to track the performance of the models in deployment. We argue that an ensemble of such metrics can be used to create a score representing the prediction quality in production. This in turn facilitates formulation and customization of ML alerts, that can be escalated by an operations team to the data science team. Such a score automates monitoring and enables ML deployments at scale.

1 Introduction

Using machine learning models to extract insights from massive datasets is a widespread industry goal [10]. The training phase typically generates several models and the model with the best predictive performance is deployed to production. However, a model's performance in production depends on both the particular data it receives and the datasets originally used to train the model. Models perform optimally on different data distributions and vary in their capacities for generalization. Production datasets often vary with external factors [8, 14]. Whether rapid or gradual, these variations can require models to be updated or rolled back to maintain good predictive performance. Massive scale in production systems prohibits manual intervention or monitoring of such events, requiring in turn automated methods to detect, diagnose, and improve the quality of predictive performance. However, typical production scenarios do not have real-time labels, so popular metrics that compare predictions with labels cannot be used to assess real-time health.

We present a technique to track the predictive performance of the deployed models called: *Model Performance Predictor* (MPP). It tracks the predictive performance metric of the model. For (a) classification and (b) regression, we present an example that targets (a) accuracy and (b) RMSE respectively as the metric to track.

Detecting the applicability of an activity model to a different domain using another model was proposed in [15] using algorithm-specific information. Similar to our approach, an error dataset is used to train another model, but it is limited to a specific algorithm (random forest) and a unique domain. With a similar goal of detecting the confidence in predictions made by a machine learning algorithm, [3] proposed hedging the predictions using conformal predictors. A hold out set (in contrast to error set) is used to obtain a bound on the error probability. On the other hand, we present an approach that models the errors by using them as the labels. On similar lines, [12] presented a metric that tracks the divergence in data patterns between training and inference. We argue that an ensemble of such approaches can be customized to serve as a score, based on which alerts can be raised.

2 Model Performance Predictor

The goal of Model Performance Predictor (MPP) algorithm is to predict the predictive performance of the deployed algorithm on the test data. This algorithm is trained on the error dataset which consists of prediction errors made by the primary algorithm. In the training phase, the data is divided into training and validation datasets (apart from the test set). The training dataset is used to train the primary algorithm that will be deployed in production. Predictions made by this algorithm on the validation dataset generate the errors that are used as labels to train the MPP algorithm.

Figure 1 describes the structure of this framework. Labels of this error dataset are the errors in predictions made by the primary algorithm, and features could be a range of things depending on the application. They could simply be the primary algorithm features themselves, predictions from the primary algorithm, probability measures from the primary predictions or some algorithm-specific metrics, such as the number of

trees or variation in output from different trees in a Random Forest. Both primary and MPP algorithms make predictions on the test dataset. The primary algorithm focuses on the classification/regression task, while the MPP algorithm focuses on predicting the performance of the primary algorithm. We present MPP as a binary classification algorithm that predicts whether a prediction is correct (1) or incorrect (0).

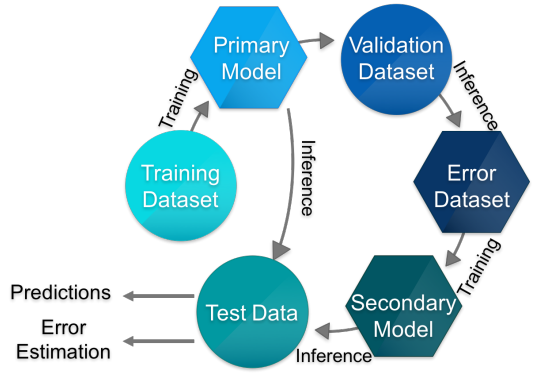


Figure 1: MPP algorithm flow

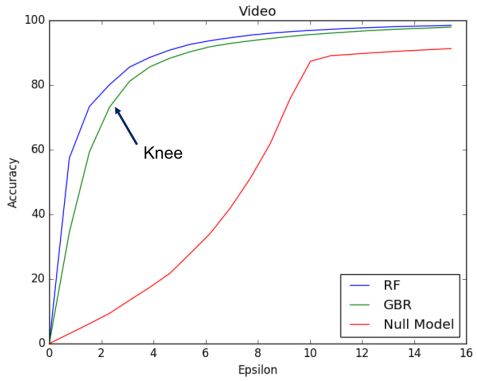


Figure 2: REC curve for the Video dataset . RF represents Random Forest; GBR represents Gradient Boosted Regression Tree

For regression problems, in order to calculate error analogously to how it is done in classification, we use a threshold (ε) on the absolute error of primary predictions to be within tolerable limits. For example, as long as error is within ±ε of the true value, it is considered an acceptable prediction (1). When the prediction of an algorithm is outside these bounds, it's considered an unacceptable (or incorrect) prediction (0). However, this threshold value is application specific and there is a need to detect a default value. To provide a default value, we use the null model concept introduced by [5]. Every regression problem has a null model and hence an REC curve associated with it. We detect the knee of this curve using the first convex dip in its double differential and choose the corresponding ε to be the default threshold value. An REC plot for the video dataset [11] is shown in Figure 2. We calculate this default threshold for all the regression experiments reported in Section 3.

Dataset	Primary Algorithm Error	MPP predicted accuracy	Absolute difference
Samsung [2]	0.92	0.92	0.00
Yelp [1]	0.95	0.95	0.00
Census [13]	0.78	0.63	0.15
Forest [6]	0.65	0.64	0.01
Letter [17]	0.71	0.6	0.11

Table 1: MPP's performance on classification datasets. Ideally, the primary algorithm accuracy and MPP's prediction should match.

Dataset	Primary Algorithm Error	MPP predicted accuracy	Absolute difference
Facebook [16]	0.56	0.56	0.00
Songs [4]	0.58	0.61	0.03
Blog [7]	0.73	0.71	0.02
Turbine [9]	0.51	0.85	0.34
Video [11]	0.59	0.72	0.13

Table 2: MPP's performance on regression datasets with default epsilon value. Ideally, the primary algorithm accuracy (generated by thresholding with default epsilon) and the MPP's prediction should match

3 Illustration

We illustrate the performance of this algorithm using 5 classification and regression datasets, listed in Table 1 and 2 respectively. Features used by the MPP algorithm for the purpose of these experiments are same as the features used by the primary algorithm. Ideally, the score presented by *MPP* algorithm should match the predictive performance of the primary algorithm. It can be seen from the tables that the MPP algorithm is able to track the performance of primary algorithm in most of the datasets.

4 Conclusion

We presented an approach *MPP* to track the predictive performance of a ML model in deployment. Such a score helps operations teams to create automated ML alerts and data scientists to get insights about the efficacy of deployed models in production. This helps both, the operations teams to monitor and manage the deployed ml model potentially preventing catastrophic predictions and the data scientists to get the information they need for further analysis of the production system.

References

[1] Yelp Dataset. https://www.yelp.com/dataset_challenge/, 2013.

[2] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L. Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. *21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN*, pages 1–15, 2013.

[3] Vineeth Balasubramanian, Shen-Shyang Ho, and Vladimir Vovk. *Conformal Prediction for Reliable Machine Learning: Theory, Adaptations and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[4] T. Bertin-Mahieux. UCI machine learning repository. http://archive.ics.uci.edu/ml/datasets/YearPredictionMSD, 2011.

[5] Jinbo Bi and Kristin P. Bennett. Regression error characteristic curves. In *Proceedings of the 20th International Conference on Machine Learning*, pages 43–50, 2003.

[6] Jock A. Blackard, Denis J. Dean, and Charles W. Anderson. UCI machine learning repository. https://archive.ics.uci.edu/ml/datasets/Covertype, 1998.

[7] K. Buza. Feedback prediction for blogs. in data analysis, machine learning and knowledge discovery. *Springer International Publishing*, pages 145–152, 2014.

[8] Piotr Cal and Michał Woźniak. Drift detection and model selection algorithms: Concept and experimental evaluation. In Emilio Corchado, Václav Snášel, Ajith Abraham, Michał Woźniak, Manuel Graña, and Sung-Bae Cho, editors, *Hybrid Artificial Intelligent Systems*, pages 558–568, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] Andrea Coraddu, Luca Oneto, Alessandro Ghio, Stefano Savio, Davide Anguita, and Massimo Figari. Machine learning approaches for improving condition?based maintenance of naval propulsion plants. *Journal of Engineering for the Maritime Environment*, –(–):–, 2014.

[10] A. Daecher. Internet of things: From sensing to doing. Wall Street Journal, 2016.

[11] T. Deneke, H. Haile, S. Lafond, and J. Lilius. Video transcoding time prediction for proactive load balancing. In *Multimedia and Expo (ICME), 2014 IEEE International Conference on*, pages 1–6, July 2014.

[12] Sindhu Ghanta, Sriram Subramanian, Lior Khermosh, Swaminathan Sundararaman, Harshil Shah, Yakov Goldberg, Drew Roselli, and Nisha Talagala. Ml health: Fitness tracking for production models. *arXiv:1902.02808*, 2019.

[13] Ronny Kohavi and Barry Becker. UCI machine learning repository. "https://archive.ics.uci.edu/ml/datasets/Census+Income, 1996.

[14] Osama A. Mahdi, Eric Pardede, and Jinli Cao. Combination of information entropy and ensemble classification for detecting concept drift in data stream. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '18, pages 13:1–13:5, New York, NY, USA, 2018. ACM.

[15] Robert P. Sheridan. Using random forest to model the domain applicability of another random forest model. *Journal of Chemical Information and Modeling*, 53(11):2837–2850, 2013.

[16] Kamaljot Singh. Facebook comment volume prediction. *International Journal of Simulation- Systems, Science and Technology- IJSSST V16*, January 2016.

[17] David J. Slate. UCI machine learning repository. https://archive.ics.uci.edu/ml/datasets/Letter+Recognition, 1991.

Low-latency Job Scheduling with Preemption for the Development of Deep Learning

Hidehito Yabuuchi *
The University of Tokyo
yabuuchi@os.ecc.u-tokyo.ac.jp

Daisuke Taniwaki Shingo Omura
Preferred Networks, Inc.
{dtaniwaki, omura}@preferred.jp

Abstract

Efficient job scheduling of *trial-and-error (TE)* jobs is a challenging problem in deep learning projects. Unfortunately, existing job schedulers to date do not feature well-balanced scheduling for the mixture of TE and *best-effort (BE)* jobs, or they can handle the mixture in limited situations at most. To fill in this niche, we present an algorithm that efficiently schedules both TE and BE jobs by selectively preempting the BE jobs that can be, when the time comes, resumed without much delay. In our simulation study with synthetic workloads, we were able to reduce the 95th percentile of the *slowdown rates* for the TE jobs in the standard FIFO strategy by 96.6% while compromising the median of the BE slowdown rates by only 18.0% and the 95th percentile by only 23.9%.

1 Introduction

Efficient job scheduling of clusters is in high demand these days, especially due to the recent explosive development of deep learning (DL) algorithms. One important type of jobs in the development of DL is *trial-and-error (TE)* jobs, in which the users conduct small-scale experiments on a trial basis for the debugging and the testing of prototype algorithms. In fact, for the private cluster at the authors' institution, TE jobs account for approximately 30% of all jobs in six months. Starting the TE jobs with low latency is critical because the users often want to monitor the learning curves of the prototypes immediately in order to save time for exploring numerous other options. The other jobs can be executed in the *best-effort (BE)* manner, but their delay should be minimized.

Unfortunately, most scheduling algorithms to date can handle the mixture of TE and BE jobs in certain situations at most. Big-C [2], a container-based preemptive job scheduler, does not handle multiplexing of GPUs. Optimus [5] and Gandiva [6] are efficient job schedulers for DL jobs, but they are only compatible with select DL frameworks. Reservation-based schedulers such as Hawk [4] reserve a separate portion of a cluster to guarantee the immediate scheduling for short jobs. Given highly diverse workload, however, it is often challenging to find the optimal reservation factor.

In this paper, we take the novel strategy of systematically suspending a selected set of BE jobs in favor of the TE jobs. Our proposed algorithm can handle any DL jobs that can be suspended, and it can be used in a variety of situations. We also take special care not to *neglect* the BE jobs. By selectively preempting the BE jobs for which the scheduler can re-schedule its execution in relatively short time, our algorithm makes sure not to greatly delay the BE jobs.

2 Proposed Preemption Algorithm

2.1 System Model

We built our preemption algorithm on the FIFO principle, which is widely used in production (e.g., Kubernetes [1]), so that we can easily integrate our algorithm into the existing frameworks. For simplicity, we assume that each job consists of a single task. Unlike big-data processing, a typical job that trains a DL model does not have multiple tasks.

When submitting a job, the users are asked to specify its type, either TE or BE, along with the types and the amount of the resource demanded for the job. When a TE job arrives at a job queue, one or more BE jobs are suspended to make room for the incoming TE job if the resource is insufficient. The preempted BE jobs are placed back on the top of the queue to observe the FIFO. Some jobs demand the time for suspension processing (e.g., storing data) before being suspended. We therefore allow a *grace period (GP)* of user-specified length for each suspension prompt. In this study, we propose an efficient rule for deciding which BE jobs shall be preempted.

2.2 Proposed Algorithm

Our algorithm is based on the following observations:

Minimizing the re-scheduling intervals. Since a preempted BE job is placed back on the top of the queue, it will be re-scheduled without much delay. However, if a BE

*Work done during an internship at Preferred Networks, Inc.

job that demands large resource is preempted without any consideration, other BE jobs waiting in the queue must wait until the scheduler secures a large room for the resumption of the preempted large BE job.

Minimizing the number of preemptions. On the other hand, preempting too small a BE job can also increase the overall slowdown of BE jobs. If a single preemption cannot make enough room for an incoming TE job, the scheduler has to preempt still another BE job. Many numbers of preemptions increase the total time loss incurred by the re-scheduling.

Minimizing the preemption-incurred time loss. It is also not preferable to preempt a BE job with too long a GP, because the length of GP affects the time until the execution of the incoming TE jobs.

Thus, we shall always preferentially preempt BE jobs with (1) small resource demand, (2) an ability to offer enough resource for the incoming TE job, and (3) short GPs. Our *Fitting Grace Period Preemption (FitGpp)* algorithm evaluates the following score for each BE job j:

$$\text{Score}(j) := \frac{\|D_j\|}{\max_{j\in\mathcal{J}} \|D_j\|} + s \times \frac{\text{GP}_j}{\max_{j\in\mathcal{J}} \text{GP}_j} \qquad (1)$$

where D_j is the vector of resource quantities demanded by the job j [1], and $\mathcal{J}$ is the set of all running BE jobs. The parameter s determines the importance of the GP relative to the resource demand. At all time, FitGpp preempts the BE job that solves:

$$\arg\min \left\{ \text{Score}(j) \mid D_{\text{TE}} \leq D_j + N \ \wedge \ \text{PC}_j < P \right\} \qquad (2)$$

where N is the amount of free resource of the node on which j is running, PC_j is the number of times that j has been preempted, and P is the maximum number of times a given BE job can be preempted, which guards the job against starvation.

Note that the FitGpp's criterion of preemption does not depend on the execution time of jobs, so that it is not affected by the algorithm's ability to estimate the execution time. This is an important advantage of FitGpp because the estimation is generally hard [3]. This is especially true for DL jobs, whose execution time are sensitive to the hyper-parameters.

3 Evaluation

Here we briefly describe our simulation study. The more comprehensive evaluation can be found in our report [7].

We evaluated our FitGpp algorithm in a simulated environment, which consisted of 84 nodes, each having 32 CPUs, 256 GB RAM, and 8 GPUs. We compared FitGpp against (non-preemptive) vanilla FIFO, *Longest Remaining Time Preemption (LRTP)*, and *RAND*. LRTP is the algorithm used in Big-C [2], and it preferentially preempts the job with the longest remaining execution time. RAND is an algorithm that preempts a randomly selected running BE job. We compared the performance of the algorithms based on the *slowdown rate* computed by the formula $1 + \frac{\text{WaitingTime}}{\text{ExecutionTime}}$.

In order to synthesize a realistic set of workloads, we analyzed a trace of the cluster at the authors' institution, which consists of over 50,000 jobs. We approximated the mixture of TE and BE jobs in the trace with a mixture of truncated normal distributions. For the lengths of GPs, we used a normal distribution with the mean of 3 min. We set the maximum preemption limit P to 1. We evaluated the algorithms on a set of 2^{19} jobs generated from the distributions with 30% of them being TE. In the simulation, the jobs were submitted at such a rate that the cluster load would be kept at 2.0 if they were scheduled by FIFO. Additional details are in Appendix A.

The results are given in Fig. 1. FitGpp with $s = 4.0$ was able to reduce the 95th percentile of the slowdown rates of the TE jobs by 96.6% relative to that of FIFO. Our algorithm increased the median of the slowdown rates of BE jobs by only 18.0% and the 95th percentile by only 23.9%.

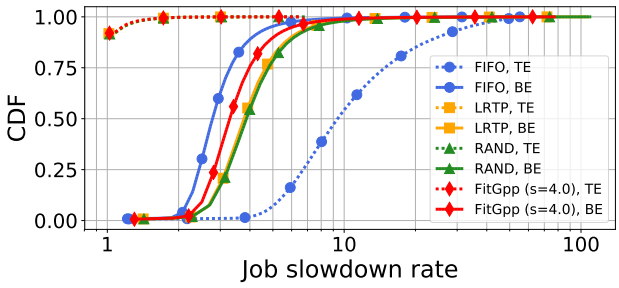


Figure 1: Job slowdown rates with synthetic workloads.

The superiority of FitGpp in this experiment was most likely due to its ability to shorten the intervals between preemptions and re-scheduling. In fact, the median of the intervals with FitGpp was almost half compared to that of LRTP and RAND, and the 95th percentile was 20% shorter than that of LRTP and 33% shorter than that of RAND. We shall also not forget that FitGpp makes an effort to reduce the total number of preemptions. When $P = 1$, it reduced the total number of preempted jobs to less than 7.0% relative to that of LRTP and RAND.

4 Conclusion

In this paper, we presented FitGpp, a preemption algorithm that reduces the latency of the TE jobs while controlling the slowdown of the BE jobs incurred by the preemption processes. Future directions include extending of this work to non-FIFO based setting and scheduling of multi-node jobs in distributed DL. Finally, the application of our algorithm is not necessarily limited to the scheduling of DL jobs. We shall be able to extend our algorithm to any type of workload that consists of a mixture of TE-like jobs and BE-like jobs.

[1] Each coordinate entry of D_j is the amount of a type of resource (e.g., CPU and RAM) relative to the capacity of the node.

Acknowledgments

We thank K. Uenishi, K. Fukuda, S. Maeda, and Y. Doi for fruitful discussion and reviewing this paper. We also thank M. Koyama for a help in the composition of the paper.

References

[1] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14:70–93, 2016.

[2] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 251–263, 2017.

[3] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of ACM Symposium of Cloud Computing conference (SoCC)*, pages 135–148, 2018.

[4] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of 2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[5] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of Thirteenth EuroSys Conference (EuroSys '18)*, 2018.

[6] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 515–610, 2018.

[7] Hidehito Yabuuchi, Daisuke Taniwaki, and Singo Omura. Low-latency job scheduling with preemption for the development of deep learning. *ArXiv e-prints*, arXiv:1902.01613v1 [cs.DC], 2019.

Appendix A Experimental Details

We simulated LongestRemainingTimePreemption (LRTP) algorithm on the assumption that it can perfectly estimate the execution time of each job. Both LRTP and RAND continue the preemption process until they can prepare enough resource for an incoming TE job. For the evaluation of RAND, we repeated the same experiment four times and report the average statistics.

In order to synthesize realistic workloads, we analyzed a trace of the cluster at the authors' institution. The trace consisted of approximately 50,000 jobs with about 30% of them being TE. Fig. 2 shows the brief statistics of the trace.

To create a realistic sequence of synthetic workloads, we approximated the empirical distributions of (1) the execution time, (2) the number of demanded CPUs, (3) the amount of demanded RAM, and (4) the number of demanded GPUs for both the TE jobs and the BE jobs with separate normal distributions, and artificially generated typical jobs from their truncated versions. The means of the fitted normal distributions for the execution time of the TE jobs and the BE jobs were respectively 5 min and 30 min. We truncated these distributions at 30 min and 24 hours, in this order.

For the lengths of GPs, we prepared the normal distribution with the mean of 3 min and truncated the distribution at 20 min. We set the length of GPs at such large values for the following three reasons: (1) typical DL jobs tend to accompany large data to store before the suspension, (2) the data often requires preprocessing step for the storage, such as serialization, and (3) we expect the developers of DL algorithms to specify long GPs because a prematurely suspended job is destined to fail.

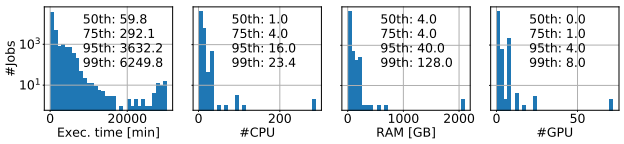


Figure 2: Statistics of jobs on the cluster at the authors' institution.

`tensorflow-tracing`: A Performance Tuning Framework for Production

Sayed Hadi Hashemi[+△], Paul Rausch, Benjamin Rabe[+△]
Kuan-Yen Chou[+], Simeng Liu[+△], Volodymyr Kindratenko[△], Roy H Campbell[+]

1 Introduction

The growing popularity of Deep Neural Networks (DNN) within the mainstream [8] has had a rapid transformative effect on clusters and data centers. DNN training jobs are becoming one of the largest tenants within clusters, and often take hours to weeks to complete; and even a slight performance improvement can save substantial runtime costs. Despite this fact, the DNN specific performance tuning tools are yet to keep up with the needs of the new changes in production environments.

On one hand, the existing application-agnostic resource-level tools such as top, Nvidia Nsight (for GPU utilization), IPM (for MPI network monitoring) are too limited to predict or explain the behavior and performance of a job accurately. In DNN applications, there exists a complex relationship among resources. Even though measuring coarse metrics such as bandwidth, latency, and GPU/CPU utilization can draw an overall picture of cluster performance, these metrics are not easily translatable to application-level metrics and do not provide actionable insights on how to handle performance bottlenecks.

On the other hand, the short list of application-aware tools, such as MLModelScope [6], TensorBoard [1], and `tf.RunOptions` [2], while able to provide actionable insights, are mainly designed for the need of application developers and are not intended for production use. Such tools require substantial modification to applications, and early planning as to what, when and how data should be collected.

In this article, we introduce `tensorflow-tracing` to fill the gap between these two classes of performance tuning tools. To achieve this goal, `tensorflow-tracing` addresses the following technical challenges:

- Collecting the application-level runtime metrics, such as the timing of each operation or the iteration time, needs explicitly expressed in the training job source code. To makes it possible to trace ML jobs without requiring any application modification, `tensorflow-tracing` *monkeypatches* the `tensorflow` library at the system level.
- Collecting some metrics is expensive and have a significant overhead on the runtime. `tensorflow-tracing` treats metrics differently; it collects low-overhead metrics automatically, while expensive ones are collected on demand through an admin interface.
- There is no easy way to exchange runtime metrics among users and admins — `tensorflow-tracing` facilities this through a portable file format and supporting tools to explore these metrics offline.

The `tensorflow-tracing` is publicly available under `Apache-2.0` license[1]. It supports native TensorFlow [3], Horovod [7], and IBM PowerAI [5] applications.

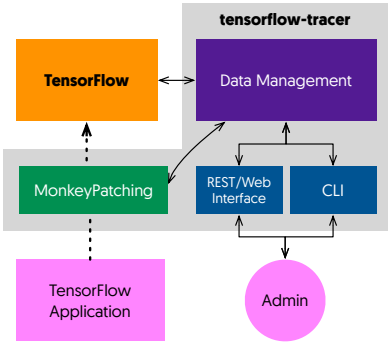


Figure 1: The architecture of `tensorflow-tracing`

[+]University of Illinois at Urbana-Champaign
[△]National Center for Supercomputing Applications
This material is based on work supported by the National Science Foundation under Grant No. 1725729.

[1] https://github.com/xldrx/tensorflow-tracer

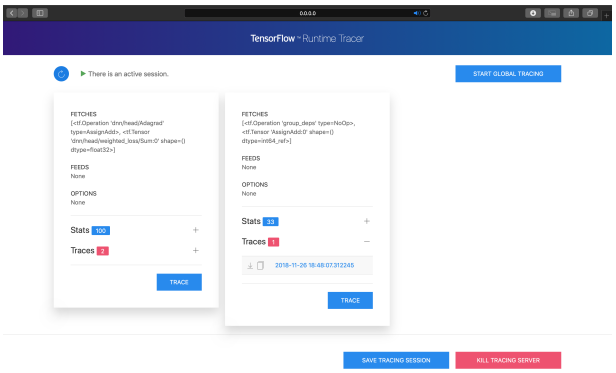


Figure 2: The main web interface of `tensorflow-tracing`. Each entry represents a separate task in the DNN session.

Figure 3: The timeline interface provides details of the execution of one iteration step. Each box represent an operation in the DataFlow DAG. There is a timeline for every resources on each machine. The trace is collected from `next_frame_sv2p` [4] model in `tensor2tensor` library [10].

2 Design and Implementation

Figure 1 shows the building blocks of `tensorflow-tracing`:

MonkeyPatching In order to provide tracing without code modification, `tensorflow-tracing` injects a proxy function to the `tensorflow` library using a *monkey-patching* scheme to intercepts the calls to certain functions and redirects them to the *Data Management* module. While *monkeypatching* the library at the system-level automatically enables tracing for any DNN application, `tensorflow-tracing` also supports per application patching.

Data Management This module is responsible for collecting profiling and tracing data as well as making online decisions as to whether a task should be traced[2]. This module is also responsible for serializing/deserializing tracing sessions from/to a file.

REST/Web Interface This interface is the main portal for interacting with the system. `tensorflow-tracing` starts a web server whenever an application is executed which is accessible either through a web browser or a REST API client (possibly from terminal). The interface provides two logical views:

1. *Main Interface* shows the list of tasks and their associated profiling/tracing data. This interface allows request tracing. (Figure 2)
2. *Timeline Interface* visualizes an instance of a task trace as a series of timelines, one for every resources (e.g. CPU, GPU, Network Interface) on each machine. Each box represent an operation in the DataFlow DAG of DNN application. (Figure 3)

CLI It loads a tracing sessions offline and enables exploring through a web interface.

[2] `MonitoredSession.run` function calls

3 `tensorflow-tracing` in action

Overhead We observe no performance hit on collecting low-overhead metrics such as iteration times, 'session.run' call names and frequencies. We observe less than 3% runtime overhead to iteration time when individual operations in a call are traced. CPU Memory requirements varies for different models. For example: an *Inception v3* [9] trace consumes 718KB while **next_frame_sv2p** [4] consumes 2.4MB.

Case Study We have used `tensorflow-tracing` on different workloads to find the performance issues on application, framework, and infrastructure level. For example, our work TicTac **??** addresses the communication timing issue we found in the tracing of a distributed TensorFlow job with parameter server.

4 Limitation and Future Work

The correctness of the `tensorflow-tracing`'s distributed traces relies on the precision of the clocks on the different machines. Currently it relies on external sources to synchronize the clocks.

`tensorflow-tracing` traces the network activities just in the user space. This will miss the events such as packet drops or queue latencies. We are planning to expand this capability by adding network stack events from kernel space.

References

[1] Tensorboard: Visualizing learning. https://www.tensorflow.org/guide/summaries_and_tensorboard, Accessed: February 15, 2019.

[2] tf.runoptions. https://www.tensorflow.org/api_docs/python/tf/RunOptions, Accessed: February 15, 2019.

[3] Martin 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, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Mohammad Babaeizadeh, Chelsea Finn, Dumitru Erhan, Roy Campbell, and Sergey Levine. Stochastic variational video prediction. 2018.

[5] Minsik Cho, Ulrich Finkler, Sameer Kumar, David Kung, Vaibhav Saxena, and Dheeraj Sreedhar. Powerai ddl. *arXiv preprint arXiv:1708.02188*, 2017.

[6] Abdul Dakkak, Cheng Li, Abhishek Srivastava, Jinjun Xiong, and Wen-Mei Hwu. MLModelScope: Evaluate and Measure ML Models within AI Pipelines. *arXiv preprint arXiv:1811.09737*, 2018.

[7] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[8] Svetlana Sicular and Kenneth Brant. Hype cycle for artificial intelligence, 2018, July 2018.

[9] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[10] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, et al. Tensor2tensor for neural machine translation. *arXiv preprint arXiv:1803.07416*, 2018.

Disdat: Bundle Data Management for Machine Learning Pipelines

Ken Yocum
Intuit, Inc.

Sean Rowan
Intuit, Inc.

Jonathan Lunt
Intuit, Inc.

Theodore M. Wong
23andMe, Inc.

Abstract

Modern machine learning pipelines can produce hundreds of data artifacts (such as features, models, and predictions) throughout their lifecycle. During that time, data scientists need to reproduce errors, update features, re-train on specific data, validate / inspect outputs, and share models and predictions. Doing so requires the ability to publish, discover, and version those artifacts.

This work introduces *Disdat*, a system to simplify ML pipelines by addressing these data management challenges. Disdat is built on two core data abstractions: *bundles* and *contexts*. A bundle is a versioned, typed, immutable collection of data. A context is a sharable set of bundles that can exist on local and cloud storage environments. Disdat provides a bundle management API that we use to extend an existing workflow system to produce and consume bundles. This bundle-based approach to data management has simplified both authoring and deployment of our ML pipelines.

1 Introduction

Managing data artifacts associated with ML pipelines remains challenging for data scientists, even with existing tools for code versioning, continuous deployment, and application container execution. The development and deployment lifecycle of a pipeline may create thousands of artifacts, including features, trained models, and predictions. At any point in time, the data science team may need to share inputs to reproduce errors, re-train on specific data, or validate model behavior.

Naming and storing data artifacts is frequently an ad-hoc and error-prone process in which data is managed per project, found via tribal knowledge, and shared by e-mail or instant messaging. This leads to significant data scatter across local computers (such as laptops) and cloud storage (such as AWS S3 [4]). Worse, data science team members often convolve naming and versioning. For example, where one expects a logical name like `financials` for a data set, one instead finds a taxonomy of names like `financials_v_1-20190520`.

We introduce *Disdat*, a system that leverages two practical abstractions–the *bundle* and *context*–to strike a balance between prescription and the need for data scientists to use the latest tools when authoring and deploying ML pipelines. The bundle is a named collection of files and literals, and is the unit at which data is produced, versioned, and consumed. The context is a view abstraction that gathers together one or more bundles, and assists with managing bundles across multiple locations. Bundles and contexts are minimally prescriptive in the same sense as high-level pipelining systems such as Luigi [9], Airflow [1], and Pinball [8] that encode dependencies between user-defined tasks.

Bundles and contexts in Disdat together support common data science activities. Conceptually, Disdat accomplishes for data what Docker does for application images. Bundles allow users to find the latest version of related pipeline data with a single "human" name instead of parsing ad-hoc names. Contexts facilitate simple sharing and synchronization of bundles between different users and across local and cloud storage locations through intuitive "push"/"pull" operations.

Disdat stands in contrast to existing systems for managing pipeline data artifacts. Many are closed, monolithic ecosystems, providing pipeline authoring, model versioning, deployment, feature storage, monitoring, and visualization. Examples include Palantir's Foundry [6], Facebook's FBLearner [3], and Uber's Michelangelo and PyML [10]. Perhaps closer in spirit to Disdat are MLFlow [2], Pachyderm [5], and DVC [7], which aim to version pipeline experiments to enable reproducibility.

Unlike prior approaches, Disdat treats bundles as first-class citizens. Where Pachyderm and DVC support `git`-like operations, Disdat eschews some version control concepts, such as branching and merging, whose semantics for ML artifacts remain an open question (e.g., merging ML model weights between branches). In addition, their units of data versioning are implementation specific; each Pachyderm container produces a single commit to a "repository", while DVC relies on an extant `git` repository to version DVC metadata. Like Disdat, the MLFlow API captures parameters and data outputs, but

users must still organize and manage their data.

The core of Disdat consists of an API to create and publish bundles in contexts. We use that API to instrument the Luigi pipelining system from Spotify [9], allowing data scientists to author pipelines that automatically produce bundles. By virtue of this design, Disdat pipelines automatically re-use prior results where possible and can publish new versions of data products on a cloud storage service.

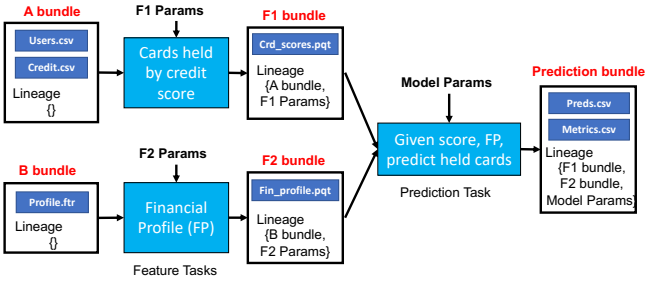


Figure 1: An ML pipeline with two featurization tasks feeding a predictive model and producing three output bundles.

2 Motivating Example

We motivate Disdat's design with a simple data processing scenario. Consider a financial services company wishing to predict credit card ownership among users. To do so, it creates a three-task ML pipeline shown in Figure 1. This pipeline featurizes the input data and applies a trained model to assign a likelihood of ownership to each user.

In general, ML pipelines consist of tasks that read and write one or more files. For example, the first featurization step ("F1") in Figure 1 reads two `.csv` files describing the user population and produces a Parquet `.pqt` feature file. The model task consumes the features to produce predictions and performance metrics. Data scientists may re-run individual tasks or the whole pipeline many times to explore features, fix bugs, and tune hyper-parameters.

Many challenges face the data scientist in managing the flow of data through this example pipeline. They must create a naming and file system directory scheme that disambiguates input, intermediate, and output files relating to different populations, which usually results in names like `users-popA.csv` and `users-popB.csv`. They often incorporate ad-hoc versioning to track updates to populations or pipeline code changes, which leads to clumsily embedded metadata such as `users-popA-20190520.csv` or `Crd_scores-with-low-score-cutoff.pqt`. Lastly, sharing and re-using data requires mechanisms to find artifacts across local and cloud locations as well as polices to define "latest" among multiple versions of the same artifact.

Disdat builds on *bundles* and *contexts* to address this challenge. Bundles organize collections of data items flowing through pipelines; thus, each task in Figure 1 produces a single bundle. A bundle is an immutable set of tags, lineage information, and named arrays. Each named array may store *scalar-typed* data, *file links*, or *pointers* to bundles. File links are references to files, such as POSIX filenames or S3 URLs.

In Figure 1, the "Prediction" bundle has one named array with two file links. When Disdat creates the bundle, it places the files and bundle metadata in the current context (on the local file system). A context serves as an organizational unit for bundles–the user decides whether the context represents a project, pipeline, or data in a test or deploy environment. Contexts hold any number of bundles and can exist at different *locations*–the local file system and a cloud storage service.

Disdat bundles provide three distinct names by which to distinguish data versions. These are a *human_name*, *processing_name*, and a UUID. The *human_name* indicates the logical data use; it supports data sharing among colleagues. In our example, the final output bundle may have *human_name* `card_predictions`. The *processing_name* is a unique string computed from the parameterized task; it allows a pipeline to re-use the most recent upstream task's output.

Note that each pipeline execution can produce bundles with the same *human_name*, but that differ by UUID and creation date. Thus synchronization between local and cloud locations is as simple as downloading bundles whose UUIDs are not present. This allows the data scientist to easily get the latest version either from a local context or one hosted on AWS S3.

3 Discussion

Disdat is a Python-based system consisting of an API for creating and managing bundles in contexts, a command-line interface, and an instrumented pipelining system. Disdat uses the API to extend Spotify's Luigi so that tasks transparently ingest bundles and produce bundles as output. In addition, Disdat can *dockerize* pipelines to run on container execution services like AWS Batch or AWS SageMaker.

At Intuit, we use Disdat for batch prediction pipelines and have found this approach valuable. Sometimes data scientists may not access raw data on their laptops or their laptop may have insufficient resources. During development, Disdat makes it easy to run that portion of a pipeline on the cloud and retrieve the output to test locally. Similarly, errors often occur during large-scale tests, and it is easy to find the set of input bundles that caused failures. Bundles have also simplified performance monitoring, as any data scientist may pull all versions of a pipeline's outputs for analysis (for example, in a notebook via the Disdat API).

4 Availability

Disdat is open-source (ASL 2.0) software available on github at `http://github.com/kyocum/disdat`.

References

[1] M. Beauchemin. Airflow. https://airflow.apache.org/, 2014.

[2] Databricks. Mlflow. https://mlflow.org/, 2019.

[3] Facebook. Introducing FBLearner Flow: Facebook's AI backbone. https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone/, May 2016.

[4] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing Google's datasets. *SIGMOD*, 2016.

[5] Pachyderm. Pachyderm. https://pachyderm.readthedocs.io/, 2019.

[6] Palantir. Foundry. https://www.palantir.com/palantir-foundry/, 2018.

[7] D. Petrov. Data version control. https://blog.dataversioncontrol.com/data-version-control-beta-release-iterative-machine-learning-a7faf7c8be67, May 2017.

[8] Pinterest. Pinball. https://github.com/pinterest/pinball, 2019.

[9] Spotify. Luigi. https://github.com/spotify/luigi.org, 2016.

[10] Uber. Michelangelo. https://eng.uber.com/michelangelo/, September 2017.

TonY: An Orchestrator for Distributed Machine Learning Jobs

Anthony Hsu, Keqiu Hu, Jonathan Hung, Arun Suresh, Zhe Zhang
LinkedIn
{ahsu,khu,jhung,asuresh,zezhang}@linkedin.com

Abstract

Training machine learning (ML) models on large datasets requires considerable computing power. To speed up training, it is typical to distribute training across several machines, often with specialized hardware like GPUs or TPUs. Managing a distributed training job is complex and requires dealing with resource contention, distributed configurations, monitoring, and fault tolerance. In this paper, we describe TonY, an open-source orchestrator for distributed ML jobs built at LinkedIn to address these challenges.

1 Introduction

The past couple of decades have seen an explosion in "Big Data" systems for storing and processing data. Some widely used systems include MapReduce [10], Hadoop Distributed File System [19], and Spark [21]. The scale of these systems has made it possible to store petabytes of data and do large-scale ML.

Many features on the LinkedIn website are powered by ML, including People You May Know, Job Recommendations, the News Feed, and Learning Recommendations. Many of these features are powered by ML techniques such as boosted decision trees [7] and generalized linear models [22].

To boost the accuracy of predictions, ML engineers have started experimenting with non-linear models such as neural networks [11] to capture more complex relationships in the data. Programming these neural networks in a generic language is tedious and error-prone. To address this, many frameworks have been created to simplify the construction and training of neural networks. These frameworks include DistBelief [9] and its successor TensorFlow [4], Theano [6], Caffe [13], PyTorch [17], and Keras [8].

An ML engineer will often begin model development by developing on a single machine. One popular tool is a "notebook" program such as Jupyter [15] or Zeppelin [1] that allows an ML engineer to interactively explore the data and test out fragments of their models. This works when experimenting on a sample of the data. However, to validate a new model, they generally need to train and test their model on the full dataset, which may be petabytes in size and would take too long to train on a single machine. To scale up their training, they need to divide the data across multiple machines and train in parallel [5].

Most ML frameworks provide APIs for doing distributed training. However, to make use of multiple machines, an ML engineer still has to copy their program to each host, set the appropriate environment variables and configurations for distributed training on each host, and then launch their training program on each host. This ad-hoc process faces several challenges:

- **Resource contention.** ML engineers sharing the same pool of unmanaged machines fight for the same memory, CPU, and GPU resources. Consequently, jobs may fail with out-of-memory exceptions or errors allocating GPUs.
- **Tedious and error-prone configuration.** Setting up a distributed training job requires copying configurations to all hosts and it is hard to verify and update these configurations.
- **Lack of monitoring.** While the job is running, it is difficult to monitor its global progress.
- **Lack of fault tolerance.** Transient errors are hard to debug and require manual restarts.

To address these challenges, we built and open-sourced TonY [12], an orchestrator that interacts with a cluster scheduler to launch and manage distributed training jobs.

2 Architecture

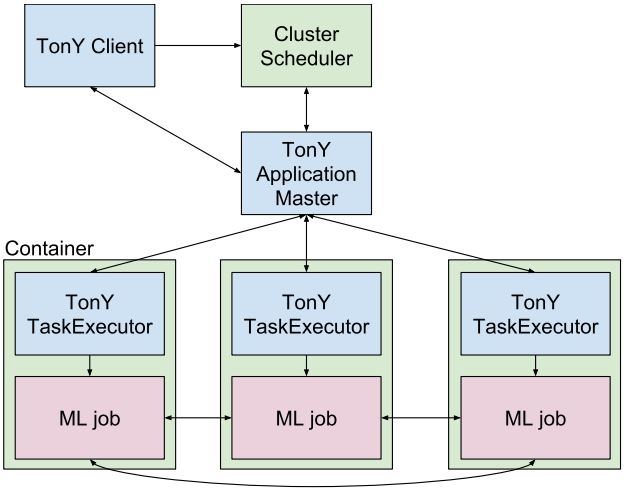


Figure 1: TonY's architecture.

TonY consists of a client for submitting jobs to a scheduler and an application that runs in the scheduler. Users use the client to submit their ML jobs, and the application handles allocating resources, setting up configurations, and launching the ML job in a distributed fashion. The client interface is generic and its implementation can support submitting to multiple schedulers. The scheduler implementation can be changed without requiring users to update their ML or client submission code.

For our initial implementation of TonY, we added support for running distributed TensorFlow jobs on Hadoop YARN (Yet Another Resource Negotiator) [20] (hence the name TonY), as these were the most commonly used ML framework and scheduler, respectively, at LinkedIn.

The overall architecture of TonY is presented in Figure 1. We present the client and cluster components of TonY in more detail in the following subsections.

2.1 TonY Client

The TonY client is the library users use to launch their distributed ML jobs. Users describe in an XML file the resources required by their job. For TensorFlow, this might include the number of worker and parameter server instances as well as how much memory and how many GPUs per instance. If needed, users can also specify additional configurations for the underlying scheduler. In the case of YARN, this might include specifying the queue [3] or node label [14] (e.g.: high-memory) to run on.

Users will also provide the path to their ML program as well as the virtual environment or Docker image [16] in which their program should run on the cluster. Additionally, users can specify properties such as model-specific hyperparameters, input data, and output location via command-line arguments passed to the TonY client.

Often, distributed ML jobs will be run as part of a larger workflow that includes data preprocessing and model deployment. To simplify integration into existing workflows, we built a TonY plugin for one such workflow manager, Azkaban [2], that lets users add distributed ML jobs in the same workflow alongside Spark, MapReduce, and other jobs.

2.2 TonY Cluster Application

When the user runs the TonY Client to submit their job, the client will package the user configurations, ML program, and virtual environment into an archive file that it submits to the cluster scheduler.

The TonY Client will launch a master program in the cluster scheduler. In our initial implementation supporting Hadoop's YARN scheduler, we launch a TonY ApplicationMaster (AM) in a YARN container. The AM then negotiates with YARN's ResourceManager (RM) to request all the other containers (e.g.: worker and parameter server tasks) needed by the ML job. The AM handles heterogeneous resource requests for different task types, such as requesting containers with GPUs for the worker tasks but requesting CPU-only containers for the parameter server tasks.

Once the task containers are allocated by the RM to the TonY AM, it then launches a TaskExecutor in each task container. This TaskExecutor will allocate a port for its task to run on and register this port with the AM. Upon receiving registration from all TaskExecutors, the AM will construct a global cluster spec that it will then send back to every TaskExecutor. Each TaskExecutor will then set the global cluster spec along with task-specific configuration in environment variables before spawning the ML job as a child process. Once all the ML jobs start up, they will communicate and coordinate with one another via the ML framework's distributed protocol (whether that be RPC, MPI, etc.), and the TaskExecutors will monitor the task processes and heartbeat back to the AM. When the task processes finish, the TaskExecutor will register the final exit status with the AM before terminating.

The TaskExecutor for the first worker task will also allocate a port for launching a visualization user interface such as TensorBoard for monitoring the running job. This also gets registered with TonY AM. This user interface URL, along with links to all the other task logs, is sent back to the TonY Client so that users can directly access the visualization UI and task logs from one place.

Finally, if any task fails, the TonY AM will automatically tear down the remaining tasks, request new task containers, setup a new global cluster spec, and relaunch the tasks. The ML tasks can then restore from the last checkpoint and continue training.

3 Discussion

Previously, ML engineers had to write ad-hoc scripts to launch distributed ML jobs on a pool of machines, with no resource guarantees or isolation from other users. Now, using TonY, users can configure their job once and rely on TonY to negotiate with a cluster scheduler for guaranteed resources.

The TonY master handles all the distributed setup and provides a central place to monitor and visualize the training job. It also ensures fault tolerance by restarting distributed jobs in case of transient task failures.

The master and TaskExecutor orchestration framework is also an ideal place to instrument the ML tasks and collect metrics about the tasks' performance and resource utilization. These statistics could be aggregated and analyzed in a UI such as Dr. Elephant [18] to suggest new settings for the ML jobs that would improve performance and resource utilization. We are currently implementing these new features in TonY and plan to discuss them more in future work.

References

[1] Apache zeppelin. https://zeppelin.apache.org/. Accessed: 2019-02-04.

[2] Azkaban: Open-source workflow manager. https://azkaban.github.io/. Accessed: 2019-02-04.

[3] Orgqueue for easy capacityscheduler queue configuration management. https://issues.apache.org/jira/browse/YARN-5734, 2016.

[4] Martin 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, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[5] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv preprint arXiv:1802.09941*, 2018.

[6] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *SciPy*, 2010.

[7] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, 2016.

[8] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] Jonathan Hung. Open sourcing tony: Native support of tensorflow on hadoop. https://engineering.linkedin.com/blog/2018/09/open-sourcing-tony--native-support-of-tensorflow-on-hadoop, 2018.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.

[14] Konstantinos Karanasos, Arun Suresh, and Chris Douglas. Advancements in yarn resource manager. *Encyclopedia of Big Data Technologies*, 2018.

[15] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[16] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[18] Akshay Rai. Open sourcing dr. elephant: Self-serve performance tuning for hadoop and spark. https://engineering.linkedin.com/blog/2016/04/dr-elephant-open-source-self-serve-performance-tuning-hadoop-spark, 2016.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *MSST*, 2010.

[20] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*, 2013.

[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[22] XianXing Zhang, Yitong Zhou, Yiming Ma, Bee-Chung Chen, Liang Zhang, and Deepak Agarwal. Glmix: Generalized linear mixed models for large-scale response prediction. In *KDD*, 2016.

Transfer Learning for Performance Modeling of Deep Neural Network Systems

Md Shahriar Iqbal
University of South Carolina

Lars Kotthoff
University of Wyoming

Pooyan Jamshidi
University of South Carolina

Abstract

Modern deep neural network (DNN) systems are highly configurable with large a number of options that significantly affect their non-functional behavior, for example inference time and energy consumption. Performance models allow to understand and predict the effects of such configuration options on system behavior, but are costly to build because of large configuration spaces. Performance models from one environment cannot be transferred directly to another; usually models are rebuilt from scratch for different environments, for example different hardware. Recently, transfer learning methods have been applied to reuse knowledge from performance models trained in one environment in another. In this paper, we perform an empirical study to understand the effectiveness of different transfer learning strategies for building performance models of DNN systems. Our results show that transferring information on the most influential configuration options and their interactions is an effective way of reducing the cost to build performance models in new environments.

1 Introduction

Deep neural networks (DNNs) are becoming increasingly complex, with an increased number of parameters to tune, and increased energy consumption for the deployed system [19]. Current state-of-the-art methods for tuning DNNs do not consider how a DNN is deployed in a system stack [1, 2, 19], and do therefore not consider energy consumption. Figure 1 shows a 4-level deployment environment of a DNN system where options and option interactions from each level contribute to energy consumption [10, 11, 14].

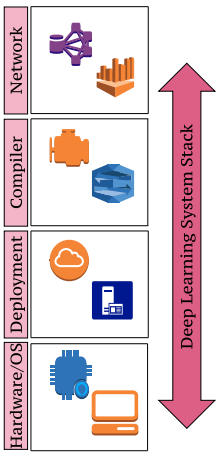


Figure 1: DNN System Stack

Performance models have been extensively used for understanding the behavior of configurable systems [5, 6, 16, 17, 22, 24]. However, constructing such models requires extensive experimentation because of large parameter spaces, complex interactions, and unknown constraints [23]. Such models are usually designed for fixed environments, i.e., fixed hardware and fixed workloads, and cannot be used directly when the environment changes. Repeating the process of building a performance model every time an environment change occurs is costly and slow. Several transfer learning approaches have been proposed to reuse information from performance models in a new environment for different configurable systems [4, 8–10]; however, to the best of our knowledge, no approach focuses specifically on DNNs in different environments. We consider the following research question:

How can we efficiently and effectively transfer information from a performance model of a DNN trained for one environment to another environment?

We perform an empirical study comparing different transfer learning strategies for performance models of DNNs for different environmental changes, e.g., different hardware and different workloads. We consider guided sampling (GS) [9], direct model transfer (DM) [21], linear model shift (LMS), and non-linear model shift (NMLS) [10]. We model the non-functional properties inference time and energy consumption in this paper and consider configuration options that affect these properties as the parameters we tune, i.e. hardware-level configuration options. Our results indicate that GS transfer learning outperforms next best learning method, NMLS, by 19.76% and 8.33% using regression trees (RT) and by 23.47% and 12.70% using neural networks (NN) for inference time and energy consumption, respectively. This enabled us to build performance models in new environments using only 2.44% of the configuration space to predict best configurations in our systems with comparable accuracy to the performance models built for the original environment.The difference between the lowest and highest energy consumption can be up to a factor of 20.

2 Methodology

We consider a pre-trained image recognition DNN system in 16 different environments: 2 different hardware platforms (Nvidia Jetson TX1, h_1, and Jetson TX2, h_2), 2 pre-trained models (Xception [3], m_1, and InceptionV3 [20], m_2) and 4 different image sizes (200×200, 400×400, 600×600, and 800×800, s_1 through s_4). In each environment, we evaluate the performance on the same 10 randomly selected images from the ILSVRC2017 [15] image recognition dataset.

The configuration space we consider is composed of the following hardware configuration options: (i) CPU status, (ii) CPU frequency, (iii) GPU frequency, and (iv) memory controller frequency. We evaluate a total of 46,080 configurations on the TX1 platform and 11,616 configurations on the TX2 platform, for a total experimental effort of $\approx$ 43.6 days of computational time across all 16 environments. We chose the TX1 and TX2 platforms due to their limited energy budget to better understand DNN system behavior with changing configuration options.

We construct performance models of the effect of configuration options on DNN system performance using these experimental data with RTs and NNs, which are frequently used in the literature to induce performance models [5, 12, 13, 16, 21]. We measure the performance of these models in terms of mean absolute percentage error, Err [18].

We implement **GS** using a step-wise regression technique with forward selection (FS) and backward elimination (BE). Each step of FS adds an interaction term to the regression

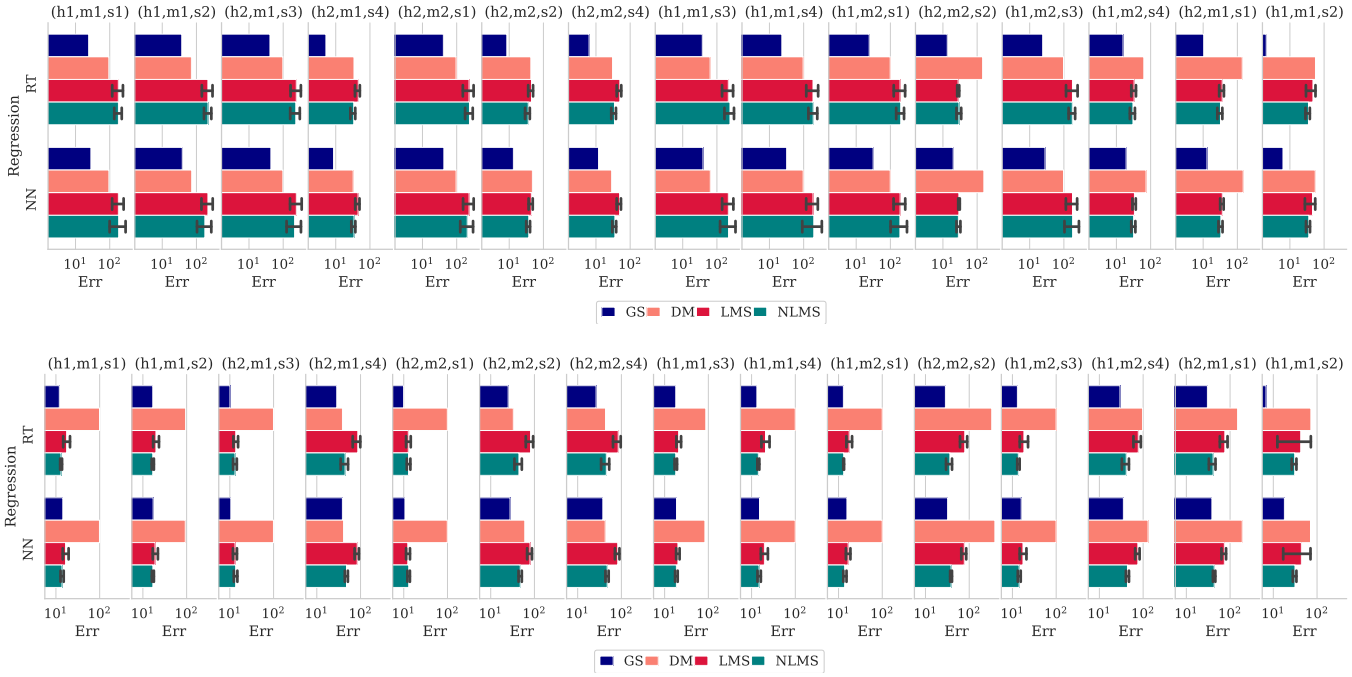


Figure 2: Comparison of prediction error of different transfer learning techniques (GS, DM, LMS, and NLMS) for performance models of DNN systems (Regression Tree and Neural Net) for inference time (top) and energy consumption (bottom). We consider 15 different target environments and show error bars for values aggregated over 10 predictions on a log scale.

model that increases the coefficient of determination, while BE removes an interaction term if its significance is below a threshold. We study the interaction terms of the final regression model; in particular, we exclude terms with coefficients that are less than 10^{-12}. These terms guide the sampling towards important configuration options and avoid wasting resources on evaluations that effect no change when building performance models in new environments. The **DM** transfer learning approach reuses a performance model built for one environment directly in a different environment. The **LMS** and **NMLS** transfer learning techniques learn a linear regression model and a non-linear random forest regression model, respectively, to translate the predictions from a performance model trained for one environment into predictions for a different environment. These transfer models are based on a small number of randomly-sampled configurations that are evaluated in both environments.

In our experiments, we select the TX2 platform with the InceptionV3 DNN and 600×600 images as the source environment to train the performance models for. We transfer these performance models to each of the remaining 15 target environments. The source code and data are available in an online appendix [7].

3 Results and Discussion

We present the results in Figure 2. They demonstrate that GS outperforms DM, LMS, and NMLS in each environment for both inference time and energy consumption. Average Err of the performance models induced using GS are 28.09% and 22.93% lower than DM, 25.64% and 21.59% lower than LMS, and 23.47% and 19.76% lower than NLMS for inference time using NN and RT, respectively. Similarly, they are 42.85% and 39.41% lower than DM, 20.52% and 13.19% lower than LMS, and 12.70% and 8.33% lower than NLMS for energy consumption for NN and RT, respectively. All of GS, LMS, and NLMS incurred the same cost (evaluation of 2.44% of the entire configuration space, $\approx$ 2.48 hours), while the cost for DM was zero as the performance model from the source environment is reused without modification in the target environment. For the DM and GS transfer learning techniques, an increase in computational effort of just 2.48 hours ($\approx$ 0.15% of the effort to train the original performance model) leads to an decrease of Err of 28.09% and 22.93% for inference time and 42.85% and 39.41% for energy consumption using NN and RT, consecutively.

If the environment change between source and target includes a hardware change, DM is more effective than LMS and NLMS for inference time modeling; however, for energy consumption, NLMS performs better than DM and LMS.

Guided sampling can help practitioners to quickly develop reliable performance models for new environments based on information they have obtained in the past to tune and optimize a system. Such performance models can guide practitioners to avoid invalid configurations and are useful for design space exploration to quickly find optimal configurations in new environments using influential configurations which typically practitioners miss. These models are also useful to learn the performance landscape of a system for performance debugging, and obtain a better understanding of how the configuration options affect performance in general. In future work, we will consider extending the configuration space with options from all 4 levels of the DNN system stack.

4 Acknowledgements

This work has been supported by AFRL and DARPA (FA8750-16-2-0042). Lars Kotthoff is supported by NSF grant #1813537.

References

[1] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. *arXiv:1710.05420*, 2017.

[2] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 367–379. IEEE Press, 2016.

[3] François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, pages 1610–02357, 2017.

[4] Daniel Geschwender, Frank Hutter, Lars Kotthoff, Yuri Malitsky, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm Configuration in the Cloud: A Feasibility Study. In *LION 8*, pages 41–44, February 2014.

[5] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 301–311. IEEE, 2013.

[6] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *In Proc. of Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[7] Md Shahriar Iqbal, editor. *Opml-DNNPerfModeling*. 2019. https://github.com/iqbal128855/OpML19-DNNPerfModeling.

[8] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 497–508. IEEE Press, 2017.

[9] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 71–82. ACM, 2018.

[10] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 31–41. IEEE Press, 2017.

[11] Irene Manotas, Lori Pollock, and James Clause. Seeds: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 503–514. ACM, 2014.

[12] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering (ASE)*, pages 1–31, 2017.

[13] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Using bad learners to find good configurations. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, ESEC/FSE 2017, pages 257–267, New York, NY, USA, 2017. ACM.

[14] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. 2016.

[15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[16] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 342–352. IEEE, November 2015.

[17] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 284–294. ACM, August 2015.

[18] P. M. Swamidass. Mape (mean absolute percentage error)mean absolute percentage error (mape). In *Encyclopedia of Production and Manufacturing Management*, pages 462–462, Boston, MA, 2000. Springer US.

[19] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[20] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 2818–2826, 2016.

[21] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proc. Int'l Conf. on Performance Engineering (ICPE)*, pages 39–50. ACM, 2017.

[22] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proc. of the Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1375–1382. ACM, 2015.

[23] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 307–319, New York, NY, USA, August 2015. ACM.

[24] Nezih Yigitbasi, Theodore L Willke, Guangdeng Liao, and Dick Epema. Towards machine learning-based auto-tuning of mapreduce. In *Proc. Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 11–20. IEEE, 2013.

KnowledgeNet: Disaggregated and Distributed Training and Serving of Deep Neural Networks

Saman Biookaghazadeh
Arizona State University

Yitao Chen
Arizona State University

Kaiqi Zhao
Arizona State University

Ming Zhao
Arizona State University

Abstract

Deep Neural Networks (DNNs) have a significant impact on numerous applications, such as video processing, virtual/augmented reality, and text processing. The ever-changing environment forces the DNN models to evolve, accordingly. Also, the transition from the cloud-only to edge-cloud paradigm has made the deployment and training of these models challenging. Addressing these challenges requires new methods and systems for continuous training and distribution of these models in a heterogeneous environment.

In this paper, we propose KnowledgeNet (KN), which is a new architectural technique for a simple disaggregation and distribution of the neural networks for both training and serving. Using KN, DNNs can be partitioned into multiple small blocks and be deployed on a distributed set of computational nodes. Also, KN utilizes the knowledge transfer technique to provide small scale models with high accuracy in edge scenarios with limited resources. Preliminary results show that our new method can ensure a state-of-the-art accuracy for a DNN model while being disaggregated among multiple workers. Also, by using knowledge transfer technique, we can compress the model by 62% for deployment, while maintaining the same accuracy.

1 Introduction

Deep Neural Networks (DNNs) have achieved tremendous accuracy improvements for various tasks, such as decision making, text processing, and video processing. Edge and cloud applications are adopting DNNs to assist users and other systems in better decision making. For example, a recent effort [1] is using neural networks on surveillance cameras on the roads to detect objects of interest. However, state-of-the-art DNN models are computationally heavy and need to continuously adopt to the environment.

Some related works have proposed new methods for distribution and acceleration of DNNs on distributed heterogeneous systems, for both training, and inference. One approach is distributed synchronous SGD algorithm [4], where each computing node executes the complete model on different batches of data. This method suffers from lack of scalability in the training process. Also, in the edge-cloud paradigm, the edge nodes are not powerful enough to take care of the whole model. Another method is *model-parallelism* [3, 9]. In this method, different layers of the model are distributed among several accelerators (on the same machine). Need to mention, the feasibility of such a model in a distributed edge-cloud environment with average connections speed is not yet evaluated.

Other related works have studied several methods to prepare DNN models for edge deployment. These methods can be broadly classified into four categories: (1) *Weight Sharing*, (2) *Quantization*, (3) *Pruning*, and (4) *Knowledge Transfer*. The weight sharing techniques [2, 5] reduce the memory occupied by the model by grouping weights and replacing them with a single value. The quantization techniques [5, 7] reduce the size of the model by shrinking the number of bits needed by the weights. The pruning techniques [5, 8, 10] reduce the complexity of a model significantly by removing weights or connections that produce a negligible response. Finally, with the knowledge transfer techniques, a small model is being supervised by a large model during the training to achieve a much higher accuracy. Unfortunately, all these methods (except knowledge transfer) are only feasible for inference scenarios.

2 Approach

Following the previous discussions, we propose KnowledgeNet (KN), which enables a disaggregated and distributed training/serving process while supporting heterogeneous environments, such as the edge-cloud paradigm. KN can disaggregate and deploy large DNN models on a set of heterogeneous processors to enable continuous training based on the user data and ever-changing environment.

The KN utilizes two specific methods to enable disaggregated and distributed model training and serving. First,

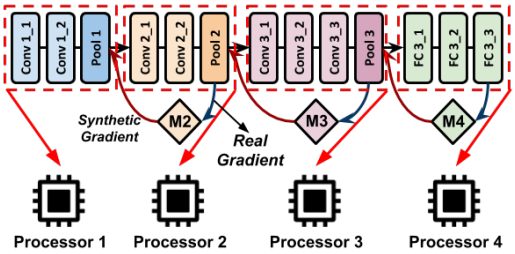


Figure 1: Representation of the model in the KN setting. Each dash box is being mapped onto a distinct processor. Also, the synthetic gradients are being generated asynchronously, using extra components (represented as M blocks).

KN can split a large DNN model into several small models (Figure 1). Each of these small models can be deployed on an independent processing node. The conventional training method suffers from the locking among the layers in a model, based upon their prior or next layer, during the forward and backward propagation. To alleviate this problem, we use the synthetic gradient method [6] to generate the target gradient for each section, asynchronously. Using synthetic gradients, each individual or set of layers can continue their progress, by adding a new module, which is responsible to generate synthetic gradients, approximating the true gradients in the conventional training model. As a result, the training process can be seamlessly offloaded onto a set of heterogeneous process without compromising the accuracy.

Second, while disaggregation can overcome the distribution problem, it may still need compression to be deployed on the edge devices. Edge devices are usually equipped with small processors with limited computational capabilities. We develop a new knowledge transfer (KT) technique in KN, which enables fine-grained supervision from the oracle model on the cloud and the model deployed on edge. Using this technique, the DNN model can be transformed into two equivalent models: (1) A large-scale oracle model on the cloud and (2) a small-scale counterpart model on the edge. Our novel KT technique provides state-of-the-art accuracy for the small-scale model, while receiving supervision from the oracle model. In the KN, unlike conventional KT techniques, where the only knowledge comes from only the final layer's loss, each section of the small model can constantly receive supervision from a specific section of the oracle model, in order to adopt the same representation.

3 Evaluation

In this section we provide preliminary evaluations on the feasibility of the KN, based on the its capability to maintain a state-of-the-art accuracy, while enabling distributed training over a set of heterogeneous devices.

Our experimental result for the synthetic gradient approach can achieve comparable accuracy as the conventional backpropagation approach. We use a simple four-layered model which consists of one convolutional layer (including max pooling layer) and three fully-connected layers with MNIST dataset for the evaluation. After training for 500K iterations, the backpropagation approach achieves 98.4% accuracy whereas the synthetic gradient approach achieves 97.7% accuracy.

Our knowledge transfer result shows that we can compress a model significantly and maintain the same accuracy by leveraging the knowledge from a large network. The teacher model is VGG16 and the student model is a network that is shorter than the teacher and consists of much fewer parameters (3.2M vs. 8.5M). After training for 100 epochs, the accuracy of teacher model and independent student model is 74.12% and 61.24%, respectively. The dependent student model that uses our proposed knowledge transfer method achieves almost comparable accuracy as the teacher model.

4 Conclusions and Future Work

The emerging trend of heterogeneous systems, both in the cloud-only and in the form of edge-cloud systems, necessitates rethinking the current methods for training and deploying deep learning models. Current available methods cannot enable efficient serving and continuous training of the complex models on the distributed heterogeneous systems. KN seeks to address the above challenges, through our novel disaggregation and distribution of the DNNs, and also our new layer-by-layer knowledge transfer techniques. Our preliminary results suggest our new method as a promising approach for training DNNs for the emerging heterogeneouscomputing paradigms.

References

[1] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *computer*, 50(10):58–67, 2017.

[2] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[3] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pages 1662–1670, 2018.

[4] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[5] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[6] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.

[7] Deepak Kadetotad, Sairam Arunachalam, Chaitali Chakrabarti, and Jae-sun Seo. Efficient memory compression in deep neural networks using coarse-grain sparsification for speech applications. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 78. ACM, 2016.

[8] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[9] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[10] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015.

Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform

Denis Baylor
Google Research

Kevin Haas
Google Research

Konstantinos (Gus) Katsiapis
Google Research

Sammy Leong
Google Research

Rose Liu
Google Research

Clemens Menwald
Google Research

Hui Miao
Google Research

Neoklis Polyzotis
Google Research

Mitchell Trott
Google Research

Martin Zinkevich
Google Research

Abstract

Large organizations rely increasingly on continuous ML pipelines in order to keep machine-learned models continuously up-to-date with respect to data. In this scenario, disruptions in the pipeline can increase model staleness and thus degrade the quality of downstream services supported by these models. In this paper we describe the operation of continuous pipelines in the Tensorflow Extended (TFX) platform that we developed and deployed at Google. We present the main mechanisms in TFX to support this type of pipelines in production and the lessons learned from the deployment of the platform internally at Google.

1 Introduction

The workflows and underlying systems for machine learning (ML) in production systems come in different shapes and sizes. One key distinction is that between one-off and continuous pipelines. One-off pipelines are initiated by engineers to produce ML models "on demand". In contrast, continuous pipelines are "always on": they ingest new data and produce newly updated models continuously. The expectation is that a "fresh" model should be pushed to serving as frequently and timely as possible in order to reflect the latest trends in the incoming traffic.

Generally speaking, any ML task whose underlying data domain is non-stationary can benefit from continuous training to keep models fresh. Failing to update models in non-stationary settings can lead to performance degradation. The frequency with which models need to be updated depends on the speed with which the underlying data evolves. We describe two characteristic examples:

- **Recommender Systems**: In recommendation systems the inventory of items that represent the corpus keeps expanding. As an example, in YouTube new videos are added every second of the day. The models that retrieve those items and rank them for users have to be updated as the corpus expands to make sure that the recommendations are fresh.

- **Perception Problems**: In many perception problems, label acquisition can be slow and costly, while the models themselves still have not converged. In these cases, it is beneficial to continuously update the model with new labeled training data, as long as the performance keeps improving with newly arriving labels.

The most extreme case of refreshing models is *online learning* [3] which updates a model with every received request, i.e. the serving model is the training model. However, in practice it is more common to update a model in batches to ensure production safety by validating the data and models before they are updated. At Google, many ML pipelines update models on an hourly or daily basis. This is often enough for the most common use-cases we will discuss below.

A key metric for continuous pipelines is model freshness, as a delay in generating a new model can negatively affect downstream services. Given that the arrival of new data is highly irregular, this necessitates a "reactive" architecture where the pipeline can detect the presence of new inputs and trigger the generation of a new model accordingly. This also implies that continuous pipelines cannot be implemented effectively as the repeated execution of one-off pipelines at scheduled intervals, e.g., every 24h: if new data appears slightly after the scheduled execution of the pipeline, it can take more than one interval to produce a fresh model which may be unacceptable in a production setting.

In this paper we describe how we implemented support for continuous pipelines in the TensorFlow Extended (TFX) platform [1]. TFX enables Google's engineers to reliably run ML in production and is used across hundreds of teams internally. The design of TFX is influenced by Google's use cases and our experience with its deployment. However, we believe that the abstractions and lessons learned are relevant for large-scale deployments of continuous ML pipelines in other environments and organizations.

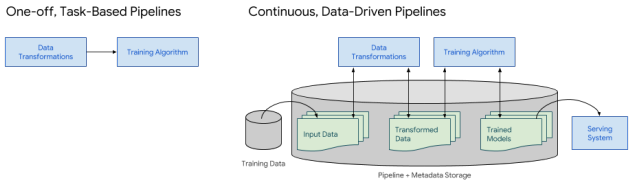


Figure 1: Continuous, data-driven pipelines need to be aware of artifacts, their properties, and lineage.

2 Continuous Pipelines in TFX

2.1 Maintaining State

Continuous pipelines need to maintain state in order to detect when new inputs appear and infer how they affect the generation of updated models. Moreover, this state can help the pipeline determine what results can be reused from previous runs. For instance, a pipeline that updates a deep learning model every hour needs to reinitialize (some of) the model's weights (also called warm-starting) from a previous run to avoid having to retrain over all data that has been accumulated up to this point. Similarly, model validation needs to retrieve the current production model in order to compare it against a new candidate model.

To manage this state, TFX introduces an ontology of artifacts which model the inputs and outputs of each pipeline component, e.g., data, statistics, models, analyses. Artifacts also have properties, e.g., a data artifact is characterized by its position in the timeline and the data split that it represents (e.g., training, testing, eval). Moreover, TFX maintains the lineage between artifacts.

2.2 Orchestration

Metadata about artifacts reflects the state of the pipeline and is recorded in a persistent store. The metadata store supports transactional updates, so that pipeline components can publish their output artifacts in a consistent fashion. Moreover, the store serves as the communication channel between components, e.g., the trainer can "listen" for the appearance of data artifacts and react accordingly. This pub/sub functionality, illustrated in Figure 1, forms the cornerstone of component execution and orchestration in TFX and enables several advanced properties. First, components can operate asynchronously at different iteration intervals, allowing fresh models to be produced as soon as possible. For instance, the trainer can generate a new model using the latest data and an old vocabulary, without having to wait for an updated vocabulary. The new model may still be better than the current model in production. Second, components can reuse results from previous runs if their inputs and configuration have not changed. Overall, this data-driven execution is essential for continuous pipelines and mostly absent from one-off pipelines.

2.3 Automated Validation

Any system that automatically generates new ML models must have validation safeguards in place before pushing a new model to production. Using human operators for these validation checks is prohibitively expensive and can slow down iteration cycles. Moreover, these safeguards need to apply at several points in the pipeline in order to catch different classes of errors before they propagate through the system. This implies more than just checking the quality of the updated model compared to the current production model. As an example, suppose that an error in the data leads to a suboptimal model. Whereas a model-validation check will prevent that model from being pushed to production, the trainer's checkpointed state might be affected by the corrupted data and thus propagate errors to any subsequent warm-started models.

TFX addresses these points by employing several validation checks at different stages of the pipeline. These checks ensure that models are trained on high-quality data (data validation [2][1]), are at least as good as or better than the current production model (model validation[2]), and are compatible with the deployment environment (serving infrastructure validation[3]).

3 Realizing One-Off, Task-Based Pipelines

TFX also supports one-off or task-based pipelines. The target audience is engineers who do not need the full power of continuous pipelines, or engineers who have set up a continuous pipeline but need to manually trigger execution of some components, e.g. experimenting with different model architectures while the input data remain unchanged.

Realizing one-off pipelines with a system that has been designed for continuous pipelines is technically straight forward, as a one-off run is just one iteration of a continuous pipeline without prior state. However, the mental model of task-based execution does not map to that of data-driven orchestration. Developers who are used to seeing jobs execute in sequence, as they were defined in a directed acyclic graph (DAG), are not accustomed to runs being triggered by the presence of a specific configuration of artifacts, as represented by the pipeline state.

As a solution, TFX introduces a framework that allows users to specify job dependency as they would in a task-based orchestration system. This also allows users of the open source version of TFX to orchestrate their TFX pipelines with task-based orchestration systems like Apache Airflow[4].

[1] Using TensorFlow Data Validation.
[2] Using TensorFlow Model Analysis for model validation.
[3] Using TensorFlow Serving.
[4] Details about the API that allows both modes of executions can only be added to this paper after March

References

[1] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1387–1395, New York, NY, USA, 2017. ACM.

[2] E. Breck, N. Polyzotis, S. Roy, S. Whang, and M. Zinkevich. Data validation for ML. In *To appear in Proceedings of SysML'19*.

[3] N. Cesa-Bianchi P. Auer and C. Gentile. Adaptive and self-confident on-line learning algorithms. In *Journal of Computer and System Sciences*, pages 48–75, 2002.

Katib: A Distributed General AutoML Platform on Kubernetes

Jinan Zhou Andrey Velichkevich Kirill Prosvirov Anubhav Garg
Cisco Systems, USA
{jinazhou, avelichk, kprosvir, anubhgar}@cisco.com

Yuji Oshima
NTT Software Innovation Center, Japan
yuuji.ooshima.fn@hco.ntt.co.jp

Debo Dutta
Cisco Systems, USA
dedutta@cisco.com

Abstract

Automatic Machine Learning (AutoML) is a powerful mechanism to design and tune models. We present Katib, a scalable Kubernetes-native general AutoML platform that can support a range of AutoML algorithms including both hyperparameter tuning and neural architecture search. The system is divided into separate components, encapsulated as microservices. Each micro-service operates within a Kubernetes pod and communicates with others via well-defined APIs, thus allowing flexible management and scalable deployment at a minimal cost. Together with a powerful user interface, Katib provides a universal platform for researchers as well as enterprises to try, compare and deploy their AutoML algorithms, on any Kubernetes platform.

1 Introduction

Automatic Machine Learning (AutoML) determines the optimal hyper-parameters or the neural network structure for a specific task. Thus it enables less technical users, and can discover state-of-art models that are almost as good as handcrafted ones ([21], [14], [16], [4], [10]). However, we have a long way before AutoML becomes mainstream. The first is the diversity of AutoML algorithms. Algorithms for hyperparameter tuning are generally different from those for neural architecture search (NAS). Even within NAS, different algorithms follow separate structural mechanisms. This diversity makes it difficult to reuse infrastructure and code, thus increasing the cost of deploying AutoML widely. The second problem is the prohibitive computational cost. The algorithm proposed by Zoph [23], for example, is expensive. This is a very active area of research.

To solve the first problem, we propose to build a general AutoML system. We show that it is possible to integrate both hyper-parameter tuning and NAS into one flexible framework. To help solve the second problem, we enable users to plug in their own optimized algorithms and we leverage micro-services and containers for scalability. With the help of Kubernetes [1], each component can be encapsulated inside a container as a micro-service.

Our contributions can be summarized as follows:

- We integrated various hyper-parameter tuning and neural architecture search algorithms into one single system.
- We standardized the interface to define and deploy AutoML workflows in Kubernetes based distributed systems.

The implementation of Katib is available at `https://github.com/kubeflow/katib`.

2 AutoML Workflows

AutoML algorithms share the common ground that they run in an iterative manner. The user first defines the search space, metrics target and maximum iterations. The algorithm searches for the optimal solution until the target metrics or the maximum number of iterations is reached. However, they may vary in terms of their internal mechanisms.

2.1 Hyperparameter Tuning

In hyperparameter tuning, we have a black-box function $f(\cdot)$ whose value is dependent on a set of parameters p. The goal is to find a $\hat{p}$ such that $f(p)$ can be minimized or maximized. In each iteration, a search algorithm service `Suggestion` will generate a set of candidate hyperparameters. The candidates are sent to `Trial` that provides training and validation services. The performance metrics are then collected by `Suggestion` to improve its next generation.

2.2 Neural Architecture Search

In neural architecture search (NAS), a neural network is represented by a directed acyclic graph (DAG) $G = \{V, E\}$, where a vertex V_i denotes the latent representation in i^{th} layer and a directed edge $E_k = (V_i, V_j)$ denotes an operation o_k whose input is V_i and output is given to V_j. The value of a vertex V_i depends on all the incoming edges:

$$V_i = g(\{o_k(V_j) | (V_i, V_j) \in E\})$$

$g(\cdot)$ is a function to combine all the inputs. It can vary in different algorithms. In [23] and [16], $g(\cdot)$ means concatenating along the depth dimension. In [4], [21] and [14], a weight is assigned to every edge so $g(\cdot)$ is naturally the weighted sum.

Extensive research in NAS has lead to enormous diversity of NAS solutions. In terms of the search objective, the algorithm may search for either the optimal network or the optimal cell. The former constructs the whole graph directly while the latter generates a subgraph G', and the whole graph is built

by duplicating the topology of G'. In terms of evolving strategy, some NAS algorithms adopt a generation approach while others use modification. With the generation approach, the algorithm will propose a new neural architecture in each iteration. With the modification approach, however, the algorithm will modify the current architecture by adding or deleting some parts of it instead of creating a brand-new candidate. Based on this categorization, the latest NAS algorithms can be summarized as follows:

Strategy	Search for Network	Search for Cell
Evolve by Generation	[23], [16], [20], [11], [2]	[23], [16], [24], [22], [13]
Evolve by Modification	[9], [4], [18], [7], [3]	[17], [21], [14], [10], [15], [6], [5]

Table 1: Summary of neural architecture search algorithms

Diverse as they are, those algorithms can be integrated into one system. Compared with hyperparameter tuning, NAS only needs one extra `ModelManager` service to store, construct and manipulate models. In each iteration, `Suggestion` provides the topology of the next candidate or the modification decisions of the previous architecture to `ModelManager`, which constructs the model and sends it to `Trial`. Then the model is evaluated and the performance metrics are fed back to `Suggestion`, starting a new iteration.

All these workflows can be summarized by Figure 1:

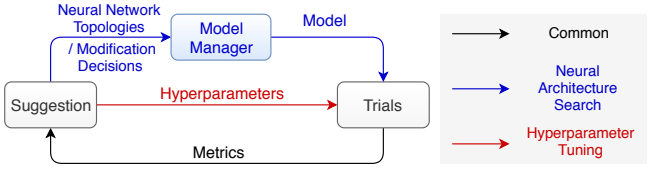


Figure 1: Summary of AutoML workflows

3 Katib System Design

We combine the requirements of these AutoML workflows and the ideas from Google's black-box optimization tool Vizier [8], with the design show in Figure 2. The user starts from defining an AutoML task with Katib's interface, the details of which can be found at `http://bit.ly/2E5B9pV`. A controller examines the task definition and spawns the necessary services. The data communication between different containers is managed by Vizier Core. The searching procedure follows exactly the workflow defined in Section 2.

Consider EnvelopeNet [10] as an example of non-standard NAS algorithm . In EnvelopeNet, the neural networks are updated by pruning and expanding *EnvelopeCells*, which are convolution blocks connected in parallel. And these modification decisions are based on *feature statistics* instead of validation accuracy. The `Suggestion` and training containers will be pre-built so the user only needs to specify the structure of *EnvelopeCells* and other necessary parameters in StudyJob yaml file. In each iteration, Vizier Core

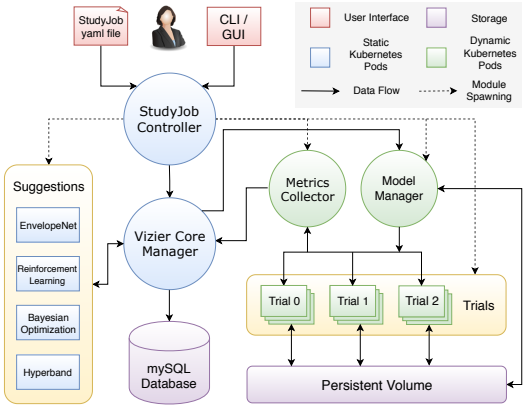


Figure 2: Design of Katib as a general AutoML system

Manager first requests one or more modification decisions from `Suggestion` and sends them to `ModelManager`. Then `ModelManager` calculates the current architectures, compiles the models into runnable objects, and sends them to `Trials`, which will carry out a truncated training process. Once finished, a `MetricsCollector` is spawned to parse *feature statistics* from the training logs. Finally, this information is fed back to `Suggestion` via the Vizier Core and a new iteration starts. During the process, all the model topologies and metrics are stored in a database and presented to the user. Katib is scalable. The controller can spawn multiple parallel `Trials` in each iteration to accelerate the search. These service components can be shared globally among all the users.

The initial version provides hyper-parameter tuning with Bayesian optimization [19], Hyperband [12], grid search and neural architecture search with reinforcement learning ([23]). The user can also deploy customized tasks by creating her own algorithm for the `Suggestion` and the training container for each `Trial`. We will add more algorithms such as EnvelopeNet [10] and integrate the support for advanced acceleration techniques such as parameter sharing [16].

4 Conclusions

This paper presents Katib, a distributed general AutoML system based on Kubernetes. The key idea is to abstract AutoML algorithms into functionally isolated components and containerize each component as a micro-service. With this extendable and scalable design, Katib can be a powerful tool for both advancing machine learning research and delivering turnkey AI solutions for enterprise users.

Acknowledgement

The authors would like to thank Cisco Systems for their support, especially Amit Saha for help with the paper.

References

[1] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, (3):81–

84, 2014.

[2] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

[3] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. AAAI, 2018.

[4] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[5] Yukang Chen, Qian Zhang, Chang Huang, Lisen Mu, Gaofeng Meng, and Xinggang Wang. Reinforced evolutionary neural architecture search. *arXiv preprint arXiv:1808.00193*, 2018.

[6] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 874–883. JMLR. org, 2017.

[7] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

[8] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[9] Haifeng Jin, Qingquan Song, and Xia Hu. Efficient neural architecture search with network morphism. *arXiv preprint arXiv:1806.10282*, 2018.

[10] Purushotham Kamath, Abhishek Singh, and Debo Dutta. Neural architecture construction using envelopenets. *arXiv preprint arXiv:1803.06744*, 2018.

[11] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric Xing. Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191*, 2018.

[12] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[13] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[14] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[15] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in Neural Information Processing Systems*, pages 7827–7838, 2018.

[16] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*, 2018.

[17] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

[18] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[19] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[20] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018.

[21] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[22] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2423–2432, 2018.

[23] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[24] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks

Anirban Bhattacharjee*, Yogesh Barve*, Shweta Khare, Shunxing Bao and Aniruddha Gokhale
Vanderbilt University, Nashville, TN, USA

Thomas Damiano
Lockheed Martin Advanced Technology Laboratories, Cherry Hill, NJ, USA

Abstract

With the proliferation of machine learning (ML) libraries and frameworks, and the programming languages that they use, along with operations of data loading, transformation, preparation and mining, ML model development is becoming a daunting task. Furthermore, with a plethora of cloud-based ML model development platforms, heterogeneity in hardware, increased focus on exploiting edge computing resources for low-latency prediction serving and often a lack of a complete understanding of resources required to execute ML workflows efficiently, ML model deployment demands expertise for managing the lifecycle of ML workflows efficiently and with minimal cost. To address these challenges, we propose an end-to-end data analytics, a serverless platform called *Stratum*. Stratum can deploy, schedule and dynamically manage data ingestion tools, live streaming apps, batch analytics tools, ML-as-a-service (for inference jobs), and visualization tools across the cloud-fog-edge spectrum. This paper describes the Stratum architecture highlighting the problems it resolves.

1 Introduction

With the increasing availability of data from a variety of sources, and significant improvements in hardware and networks that make Big Data computing easier and affordable, numerous machine learning (ML) libraries and frameworks (e.g., TensorFlow, Scikit Learn, PyTorch) have been designed in the recent past for predictive analytics. Video analysis, Object detection, Speech Recognition, Autonomous cars, Automated traffic signals, industrial robotics are examples of the many real-life applications that demand ML solutions as a part of their live stream analytics or in-depth batch analytics pipeline. However, writing code for data loading, transformation and pre-processing, and choosing the right ML algorithm for training the data and then evaluating the model and tuning the hyperparameters requires expertise. The significant promise of using predictive analytics to address a variety of problems of societal and environmental importance [3, 10] requires that ML model development be accessible even to novice users.

Further, there is substantial hype, particularly, with the use of hardware resources (e.g., GPUs, TPUs, FPGAs) along with cloud-offered infrastructure services. Dealing with this heterogeneity demands expertise in choosing the right hardware configuration that can enhance performance and minimize cost [11, 12], which is generally lacking in ML developers.

Consequently, the requirements for lifecycle management of predictive analytics are twofold:

1. ***Rapid ML model development framework,*** where the goal is to aid ML algorithm developers to build ML models using higher-level abstractions [8].
2. ***Rapid ML model deployment framework,*** where the goal is to aid developers to deploy and integrate the trained models for analytics on the target hardware and relieve the deployer from having to figure out the right configuration for their ML workflows on the infrastructure [4].

To that end, we propose a framework called *Stratum*, which addresses the development, deployment, and management lifecycle challenges of data analytics in a heterogeneous distributed environment across the cloud-fog-edge spectrum. In the rest of this paper, we present the vision behind Stratum, its key features and architectural details in Section 2, and application areas where Stratum will be useful.

2 Stratum Vision and Architecture

Figure 1 depicts the general architecture of how an analytics application can be deployed using Stratum using Model Driven Engineering [5]. We motivate an edge-cloud analytics use case scenario with a smart traffic management system. Traffic cameras collect traffic videos all the time, and rather than sending all the videos to the cloud, edge devices integrated with image recognition capabilities can procure useful insights such as traffic volume, speeding cars and traffic incidents. Based on data collected over a period of time, the traffic patterns and heavy traffic periods can be learned using batch analytics, which is a computationally intensive process that usually executes in the cloud. Finally, the intelligent traffic

*These authors contributed equally

control system typically resides in the fog nodes for real-time needs to dynamically adjust the signal timing of traffic lights based on the learned ML model and by analyzing real-time data using live analytics.

The Stratum deployment engine can deploy data ingestion tools, stream processing tools, batch analytics tool, machine learning platform, and framework on the target machine (bare metal and virtualized environments) as required. At the heart of Stratum, there is a domain-specific modeling language (DSML) that provides ML developers and deployers a user-interface with higher-level abstractions.

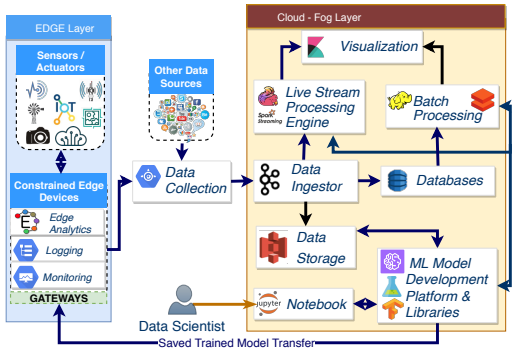


Figure 1: Generalized Representation of Applications Architecture in Stratum Metamodel

Using the DSML, the ML developer can create and evaluate their model using existing ML libraries and frameworks as shown in Figure. 2. Based on the user-defined evaluation strategy, Stratum can select the best model by evaluating a series of user-built models. Stratum can distribute each ML model on separate resources to speed up the training and evaluation phase. Moreover, a Jupyter notebook environment can be attached to our framework so that the auto-generated code by the Stratum DSML can be verified and modified by the expert user if needed.

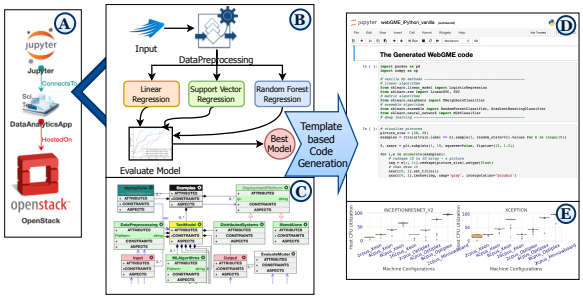


Figure 2: The user-defined hierarchical model (Blocks A and B) of ML model development framework in WebGME), the metamodel (partial) of Stratum (Block C), autogeneration of the ML code for subsequent deployment and execution (Block D), and performance monitoring tool (Block E).

Once the ML model is built and evaluated, the Stratum framework can save and profile it. Stratum supports a pluggable architecture, so the user-supplied specifications are parsed and transformed into deployment-level infrastructure-as-code [5–7]. Then the user's ML workflows are deployed on the appropriate machines across cloud-fog-edge, and Stratum's serverless execution platform allocates the necessary resources. A resource monitoring framework [1, 2] within Stratum keeps track of resource utilization and is responsible for triggering actions to elastically scale resources and migrate tasks, as needed, to meet the ML workflow's Quality of Services (QoS). The modeling concepts in Stratum DSML and code generation capabilities of the deployment/management engine are designed using the Web Generic Modeling Environment (WebGME) [9]. Both the DSML and engine are extensible, modularized and reusable.

3 Key Features and Benefits of Stratum

Stratum has been designed with the following key requirements in mind and hence supports the following features:

1. *Rapid Machine Learning (ML) model Development Framework:* The ML model development framework enables fast and flexible deployment of state-of-the-art ML capabilities. It provides a *ML Service Encapsulation* approach leveraging microservice and GPU-enabled containerization architecture and APIs abstracting common ML libraries and frameworks. It provides an easy-to-use scalable framework to build and evaluate ML models.
2. *Rapid Machine Learning (ML) model Deployment Framework:* Stratum provides intuitive and higher-level abstractions to hide the lower-level complexity of infrastructure deployment and management and provides an easy-to-use web-interface for the end users. The DSML generates "correct-by-construction" infrastructure code using constraint checkers before proceeding to actual deployment.
3. *Support for ML Model Transfer:* Stratum provides an intelligent way to transfer the trained model on the target machines (across the cloud-fog-edge spectrum) as an ML module for inference. ML module can be placed on the edge devices, or it can be placed on Cloud or Fog layer for live or in-depth analysis of data, which depends on user requirements and capacity analysis.
4. *Extensibility and Reusability:* Stratum is implemented in a modularized way, and each module is easy to reuse due to plug and play architecture. Similarly, new hardware support can be fused to Stratum in a standardized manner.

Availability

Stratum and its associated tooling are available via Github from https://github.com/doc-vu/Stratum.

Acknowledgments

This work was supported in part by NSF US Ignite CNS 1531079, AFOSR DDDAS FA9550-18-1-0126 and AFRL/Lockheed Martin StreamlinedML program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, AFOSR or AFRL.

References

[1] Y Barve, Shashank Shekhar, A Chhokra, Shweta Khare, Anirban Bhattacharjee, and Aniruddha Gokhale. Poster: Fecbench: An extensible framework for pinpointing sources of performance interference in the cloud-edge resource spectrum. In *Proceedings of the Third ACM/IEEE Symposium on Edge Computing*, 2018.

[2] Yogesh Barve, Shashank Shekhar, Shweta Khare, Anirban Bhattacharjee, and Aniruddha Gokhale. Upsara: A model-driven approach for performance analysis of cloud-hosted applications. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 1–10. IEEE, 2018.

[3] Bogdan Batrinca and Philip C Treleaven. Social media analytics: a survey of techniques, tools and platforms. *Ai & Society*, 30(1):89–116, 2015.

[4] Anirban Bhattacharjee, Yogesh Barve, Aniruddha Gokhale, and Takayuki Kuroda. A model-driven approach to automate the deployment and management of cloud services. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 109–114. IEEE, 2018.

[5] Anirban Bhattacharjee, Yogesh Barve, Aniruddha Gokhale, and Takayuki Kuroda. (wip) cloudcamp: Automating the deployment and management of cloud services. In *2018 IEEE International Conference on Services Computing (SCC)*, pages 237–240. IEEE, 2018.

[6] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[7] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[8] Markus Hofmann and Ralf Klinkenberg. *RapidMiner: Data mining use cases and business analytics applications*. CRC Press, 2013.

[9] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60, 2014.

[10] Maryam M Najafabadi, Flavio Villanustre, Taghi M Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. Deep learning applications and challenges in big data analytics. *Journal of Big Data*, 2(1):1, 2015.

[11] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[12] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404. ACM, 2017.