Scalable Multi-Framework Multi-Tenant Lifecycle Management of Deep Learning Training Jobs

Scott Boag[1], **Parijat Dube**[1], **Benjamin Herta**[1], **Waldemar Hummer**[1], **Vatche Ishakian**[1,2],
K. R. Jayaram[1], **Michael Kalantar**[1], **Vinod Muthusamy**[1], **Priya Nagpurkar**[1], **Florian Rosenberg**[1,3]

[1] IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights
NY 10598, United States

[2] Bentley University
175 Forest Street, Waltham
MA 02452, United States

[3] Braintribe IT Technologies GmbH
Kandlgasse 19-21
1070 Vienna, Austria

Abstract

With the ongoing rise and phenomenal success of machine learning (ML), particularly deep learning, efficient training of large neural network models in scalable cloud infrastructures becomes a priority. ML workloads have traditionally been run in high-performance computing (HPC) environments, where users log in to dedicated machines and utilize the attached GPUs to run jobs that train models on huge datasets. Providing a similar user experience in a multi-tenant cloud environment comes with its own unique challenges regarding fault tolerance, performance, and security. We tackle these challenges and present a deep learning stack specifically designed for on-demand cloud environments. Based on a detailed discussion of the system architecture, we examine real usage data from internal users, and discuss performance experiments that illustrate the scalability of the system.

1 Introduction

Training large neural network models is very resource intensive, and even after exploiting parallelism and accelerators such as GPUs, a single training job can still take days [1]. Consequently, the cost of hardware is a barrier to entry. Cloud-based deep learning solutions that mitigate the high upfront investment in hardware are therefore gaining in popularity. Even when upfront cost is not a concern, there is a need for robust and scalable sharing of resources among the teams in an organization.

One reason for the popularity of deep learning in both industry and academia is deep-learning frameworks such as Tensorflow, PyTorch, Caffe, Torch, Theano, and MXNet. These frameworks reduce the effort and skillset required to design, train, and use deep learning models. As with programming languages, frameworks have their own strengths and users develop an affinity to one or more of them. To cater to such a diverse user population, a cloud-based DL solution should support a wide range of frameworks, and do this in a secure, scalable and fault-tolerant manner.

This paper introduces the Fabric for Deep Learning (FfDL, pronounced *fiddle*), a cloud-based deep learning stack used at IBM by AI researchers in areas including machine translation, computer vision, and healthcare. Building on our previous work [2], this paper makes three key contributions. In Section 2, we outline the key challenges faced in cloud-based deep learning systems, and how FfDL seeks to address them. Next in Section 3, we describe the architecture and selected implementation aspects. Section 5 then presents an evaluation of the performance characteristics and an analysis of the usage patterns observed in an internal deployment of the platform.

2 Challenges

Deep learning practitioners are accustomed to training their models in a high-performance computing (HPC) environment where they have access to a cluster of machines with specialized hardware,

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

including high speed interconnects and large, high throughput disk drives. A number of new concerns arise when trying to execute these workloads in a cloud environment where failures are more common and there is a tendency to provision more commodity hardware for economic reasons. This section presents a number of these challenges.

Resilience. Faults may occur at any hardware or software layer, including network switches, GPUs, and job scheduling components [3]. FfDL can survive restarts of components, detect and avoid faulty or misconfigured subsystems, and recover from certain classes of intermittent failures.

Scalability and elasticity. There is an expectation of virtually unlimited resources, and the infrastructure needs to scale to match possibly unpredictable workload patterns. FfDL is designed to scale with the workload. Microservices can be replicated to match the load, and hardware resources, including GPUs can be added and removed without disrupting running jobs.

Observability. HPC users are accustomed to being able to login to a machine, install their own scripts and tools, and run diagnostic tools to help debug their models. FfDL exposes a number of standard APIs that can be used to monitor the progress of a training job and track key metrics.

Security. A cloud service runs workloads from multiple users on shared infrastructure with enough isolation that malicious users cannot compromise one another. This is exacerbated in DL workloads since the service needs to execute user-provided code. FfDL supports a number of isolation techniques, at the process, container, and network layer to guard against unauthorized access while still running user code in a shared system deployment.

Distributed training. Many DL frameworks support distributed training, where the training is parallelized across multiple learner processes, often using custom communication protocols. FfDL supports framework agnostic distribution, by orchestrating the learner replicas and establishing the network communication among learners in a training job without compromising security of other jobs. Frameworks that do not have distribution built-in are modified so that they can be distributed too.

3 System Architecture

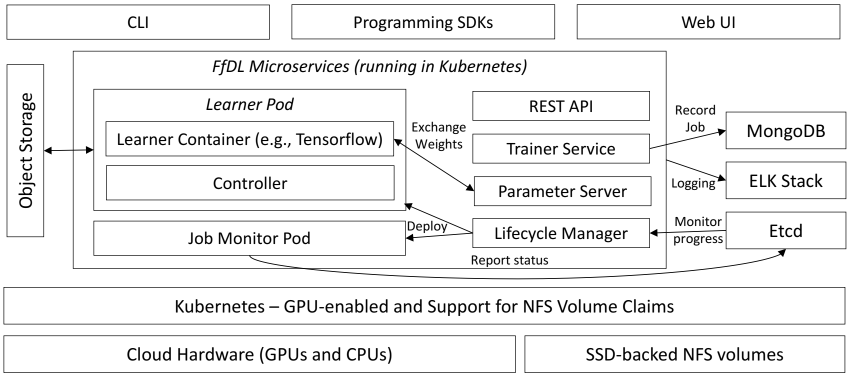


Figure 1: FfDL Architecture

Scalable and Fault-Tolerant Architecture. The FfDL platform has adopted a microservices architecture [4], to reduce coupling between components, keep each component simple and as stateless as possible, isolating component failures, and to allow each component to be developed, tested, deployed, scaled, and upgraded independently. The major components are illustrated in Figure 1.

REST API. The REST API microservice handles all incoming requests. The service also load balances requests and is responsible for authentication. Load balancing is implemented by registering the REST API service instances dynamically in a service registry.

Trainer. The Trainer service admits training job requests, persisting metadata and model input configuration in a database (MongoDB). It initiates job deployment, halting and (user-requested) job termination by calling the appropriate gRPC [5] methods on the Lifecycle Manager (LCM) microservice. The Trainer also assigns a unique identifier to each job, which is used by all other components to track the job.

Lifecycle Manager (LCM) and Learner Pods. The LCM deploys training jobs arriving from the Trainer, halting (pausing), and terminating training jobs. LCM uses the Kubernetes [6] cluster manager to deploy containerized training jobs. A training job is a set of interconnected Kubernetes pods, each containing one or more Docker containers. Jobs can be single node or distributed, and can co-ordinate using a peer-to-peer overlay network (for Caffe2) or using a parameter server (e.g., Caffe, Torch). The LCM determines the learner pods, parameter servers, and interconnections among them based on the job configuration, and calls on Kubernetes for deployment. For example, if a user creates a Caffe2 training job with 4 learners and two CPUs/GPUs per learner, the LCM creates five pods: one for each learner (called the *learner pod*), and one monitoring pod called the *job monitor*.

Kubernetes Cluster Manager. We have extended Kubernetes to support scheduling of GPUs. The microservices themselves are deployed as pods, and we rely on Kubernetes to manage this cluster of GPU-enabled machines effectively, to restart microservices when they crash, and to report the health of microservices. Training jobs are, by default, scheduled in FIFO order. We have also developed extensions to the Kubernetes scheduler to support job priorities. For each training job, the LCM uses its specification to request the set of required resources (e.g., GPUs, memory), and Kubernetes finds and provisions the nodes that satisfy the requirements. We are in the process of open sourcing our Kubernetes extensions, working with the community to potentially merge them into upstream.

Coordinating through etcd [7]. To coordinate the microservices and training jobs, we use the etcd distributed key-value store. It is used, among other things, to report the progress and status of learners in a training job. Failed microservice instances and training jobs are rescheduled by Kubernetes and the restarted instances continue where their predecessors left off using the information in etcd.

Checkpoint/Resume. The LCM also halts (pauses) the training job in response to a user request. Since the LCM is common to all jobs and learner frameworks in FfDL, supporting pause becomes a challenge, unless we modify the frameworks to export a pause API. Instead, we use etcd: the LCM creates a *halt* node in the job's etcd directory (`/jobs/training-xyz/halt`). The learner pod has a sidecar *controller* container which watches etcd and instructs the learner to halt and checkpoint its state to persistent storage. The controller then creates a *halted* node in etcd to report successful halt back to the LCM, which garbage collects the pods. When the user wants to resume the job, the LCM will redeploy the learner pods and the controller will instruct the learner to resume from the last checkpoint. Note that this makes use of the framework's support for checkpoint/resume.

Monitoring Jobs. The LCM deploys a *job monitor* pod per training job, responsible for monitoring the progress of all learners and reporting aggregated status to the Trainer. Communication between the job monitor and learners are via etcd: the Controller in the learner pod updates the current status of each learner to an etcd node (`/jobs/training-xyz/status`). The controller monitors the learner container in its pod, interprets its state and writes updates to etcd. If a single learner pod fails, the job monitor considers the entire job to have failed, but this can be relaxed. For non-failure status updates, the job monitor aggregates the updates from the learners, i.e., a job's status changes from, for example, DOWNLOADING only when all its learners have changed status to PROCESSING.

Handling Faults. FfDL has been designed so that failures are isolated in a component. If a learner fails, the job monitor detects this failure, and instructs the LCM to terminate itself and the learner pods. Users can inspect logs to diagnose the root cause for a failure and restart the job from a checkpoint. If the job monitor fails, it gets restarted by Kubernetes, picks up the state from etcd, and resumes normal operation. The LCM can be upgraded at runtime without disturbing existing jobs. If LCM crashes and gets restarted, jobs that were *in-flight* (i.e, in the middle of deployment) will have to be re-deployed by the (restarted) LCM, but existing jobs are not affected. We also deploy multiple replicas of the LCM, and also etcd itself is replicated, and all updates to etcd nodes are serializable.

Security. FfDL isolates the execution of a user's model by running the learner a separate Docker container with non-elevated privileges and in an isolated pod with network policies to prevent all incoming and outgoing network access from the pod. Input data and output results from the learner are communicated through a shared file system.

Plugin frameworks. Adding support for a new framework is as simple as building a Docker image that includes some custom scripts and the respective framework libraries. The LCM will instantiate pods with the correct Docker image.

4 Related Work

Scalable training of DL models remains a bottleneck in several important applications. Jin et. al [8] compare the training time of synchronous and asynchronous approaches to stochastic gradient descent (SGD) for image classification. The core finding is that asynchronous SGD converges faster up to 32 nodes, whereas synchronous SGD scales better between 32 and 128 nodes. While their contribution focuses on distributed algorithms (including network optimizations like gossipping), our work is primarily targeted at supporting multi-tenant environments and ensuring efficient, fair, and secure job resource allocation. The Hemingway tool [9] guides the selection of appropriate algorithms and cluster size to use for particular distributed jobs. Li et al. [10] discuss challenges associated with building a scalable ML service, including feature computation over global data, accounting for regional characteristics, as well as real-time serving of hundreds of thousands of models.

While there has been a lot of focus on securing multi-tenant services [11], there has been little attention paid to DL workloads in such an environment. DL workloads exhibit unique characteristics, including long running jobs that can last for days or weeks, the need to execute arbitrary user code on sensitive data, and the use of specialized hardware (GPUs) that still lack mature virtualization support. This paper is among the first we are aware of to address this aspect. Security and privacy are also key concerns when it comes to vulnerabilities in the models themselves. Tramer et al. [12] discuss attack vectors for ML cloud services, in particular model extraction where an attacker attempts to reverse-engineer a machine learning model by systematically probing an inferencing API. In the future we also envision cross-tenant optimizations for FfDL, such as joint backpropagation neural network learning [13], which will require additional security hardening in the platform.

The ModelDB system [14] manages machine learning models, letting data scientists quickly iterate on models, make results reproducible, and find insights faster. While ModelDB focuses on storing and querying model (meta-)data, our work focuses on efficient execution in the cloud. In the near future we also envision a more tight integration of FfDL with model databases like ModelDB. The SLAQ framework [15] explores quality-runtime tradeoffs across multiple jobs to maximize system-wide quality. The approach is based on predicting loss values that characterize the convergence behavior of a machine learning job at runtime. In our ongoing work we are also extending our platform to integrate predictive approaches. Another key success factor for optimization is prediction of job resource requirements as well as job performance [16, 17].

5 Evaluation

5.1 Scalability

Our experiments evaluate two aspects of performance: (1) scalability of the platform with the number of concurrent jobs, and (2) scalability of a training job with GPUs.

We developed an Apache JMeter based workload generator which submits concurrent jobs to a FfDL deployment on a 10 node cluster. A node in the cluster has 2 Nvidia Tesla K80s cards (each with two individually addressable GPUs), for a total of 40 GPUs. Each job involves training a Torch based neural network model for natural language classification. Table 1 summarizes the results with different number of concurrent jobs. The average end-to-end job completion time increased from 50.15 to 81.36 secs when going from 10 to 40 concurrent jobs. Thus, with a 4x increase in number of concurrent jobs, the average job completion time only increased by 62.2%. It is important to note that when the stress test was conducted, the cluster was in active use by other users, and hence the results tend to underestimate how the system would scale with a more controlled workload.

Given the two K80 cards per node, a single node training job can be configured to use up to four individual GPUs. Gradient exchange between GPUs after each iteration requires some communication overhead. We ran training jobs with varying GPUs and measured the achieved throughput (samples processed per second) and model accuracy. In particular, we trained a Caffe Alexnet model with an Imagenet-1K dataset [18] for 90 epochs and an effective batch size per learner of 256 images. The dataset is about 250 GB and has 1.3 million images. Figure 2 shows single node performance with 1,

# jobs	mean (s)	min (s)	max (s)	stdev
10	50.15	43.23	54.22	4.17
20	58.36	43.95	64.63	5.48
30	74.67	54.68	186.73	21.69
40	81.36	49.48	129.93	23.93

Table 1: Concurrent job completion times

2, and 4 GPUs. With 4 GPUs, the training finishes in about 40 hours compared to over 100 hours with a single GPU. The throughput with 1, 2, and 4 GPUs is about 313, 539, and 929 images per second, respectively. This constitutes a 2.97x increase going from 1 to 4 GPUs. The trained model's accuracy is very similar, between 57%-58%.

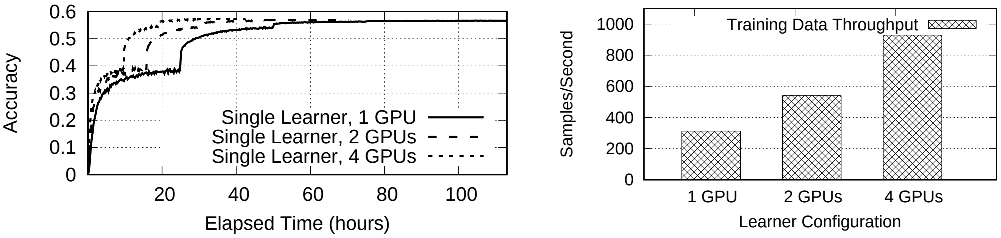
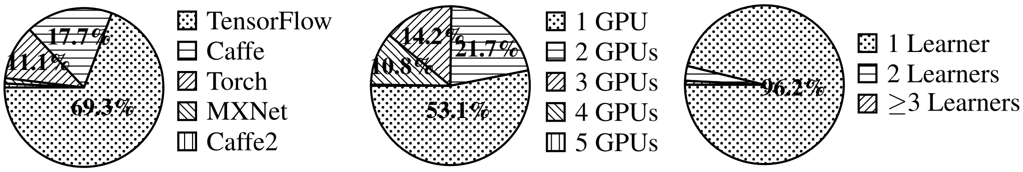


Figure 2: Accuracy and throughput with 1/2/4 GPUs for a single learner

In distributed training, a job can seamlessly exploit GPUs from multiple nodes, but then the gradient exchange must traverse the network. We evaluate a distributed job with 2 learners, each with 4 GPUs, and compare it to a single learner with 4 GPUs. For a fair comparison between the different configurations, we used same effective batch size of 2048 images for each run. With 8 GPUs we achieve a throughput of 1752 images per second compared to 1040 images with 4 GPUs, a factor of 1.7x increase. The 8 GPU job converges to 58% accuracy and the network communication overhead in 2 learner case is 18.76%. Observe that, to efficiently reap the benefits of distribution we need to work with large batch sizes so as to keep the compute to communication ratio high. However, large batch sizes may lead to slow convergence unless other model hyperparameters (e.g., learning rate, step size) are appropriately tuned. While the results are for a specific implementation of the parameter server, the insights are applicable to any parameter server based distributed learning. The network communication overhead gets more prominent when working with faster GPUs (such as NVIDIA Tesla P100 and V100), and innovative inter-GPU communication technologies will be needed [19].

5.2 Platform Usage

We report some interesting usage data for a total of 9498 training jobs that we recorded over a time window of 31 days. Note that this usage data reports only a fraction of the DL activity in the organization. Also, these numbers are from real (human) usage only, and do not include the test and performance training jobs we run automatically as part of our continuous integration pipeline.

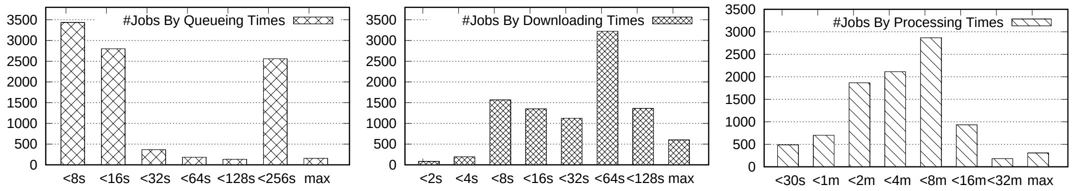


(a) Deep Learning Frameworks (b) GPUs Per Learner (c) Learners Per Job

Figure 3: Platform Usage - Job Characteristics

Figure 3 summarizes the key metrics in terms of job characteristics. As illustrated in Figure 3a, the vast majority of jobs (69.3%) are written in TensorFlow, followed by Caffe (17.7%) and Torch (11.1%). Only a small fraction of jobs use MXNet or Caffe2 as the underlying framework. Figure 3b depicts the GPUs per learner container. More than half of the learners (53.1%) use a single GPU, 21.7% use 2 GPUs, and around 25% of learners have 3 or more GPUs attached. Finally, Figure 3c shows the number of learners per jobs. In this dataset, the majority of jobs use only a single learner (96.2%), but we are recently seeing increasing usage of running multi-learner configurations.

Figure 4 shows job execution times, which roughly consist of job queueing (4a), downloading training data from the object store (4b), and actual processing time (4c). Queueing times are typically short, with most jobs being scheduled within 16 seconds, although we recorded some outliers during periods of high resource utilization. Downloading and processing time are highly dependent on the size of the training data set. The downloading times we recorded are often less than a minute, whereas processing



(a) Queueing Times (b) Downloading Times (c) Processing Times

Figure 4: Platform Usage - Job Execution Times

typically takes at least several minutes (with outliers in the order of hours). As can be seen from these metrics, FfDL is currently used primarily for smaller jobs and quick experimentation, whereas more computationally heavy workflows are currently being migrated from our HPC cluster environments to this new platform. We plan to report the findings of this migration effort in future work.

5.3 Lessons Learned

From a usability standpoint, our hypothesis was that users can get up to speed and train/serve models much faster with FfDL, compared to other platforms and building their own deep learning software stack. We ran beta programs in our organization and users were able to ramp up quickly (approx. 1 week); FfDL allows them to focus on their application code without worrying a lot about the training infrastructure. We also got valuable feedback for potential improvements, and have already improved the syntax of the configuration file twice, added new frameworks and features like log streaming.

We designed FfDL from the ground-up to be fault tolerant, and this proved invaluable in operations. The microservice-based design that we adopted has proven resilient to partial and complete machine failures, connectivity issues, etc. and has enabled us to upgrade single microservices independent of the others. We have also been able to upgrade FfDL microservices without disrupting running jobs, because of our use of a dedicated job monitor and because we store job status in etcd. The sidecar container pattern in the learner pod has proven very effective for isolating the learner process and plugging in cross-framework functionalities (data loading, log scraping, etc.). We experienced a range of different (intermittent) issues with our Kubernetes cluster, and the fault-tolerant architecture of FfDL allowed us to recover quickly and provide stable service. Although the platform is not in large scale production yet, our early results allow us to further tweak the performance and reliability.

Currently, training jobs (including resource requirements) are manually defined in a configuration file. For better usability we are working on an automated approach to determine the optimal resource allocation. Another challenge is the seamless integration of external components. While currently the training data needs to be manually uploaded to an object storage service, users would like to see this functionality built into the platform. Users were also asking for a way to get real time feedback and better debugging capabilities for training their models.

6 Conclusion

Training deep neural network models requires a highly tuned system with the right combination of software, drivers, compute, memory, network, and storage resources. FfDL offers a stack that abstracts away these concerns so data scientists can execute training jobs with their choice of DL framework at scale in the cloud. FfDL has been architected to offer properties such as resilience, scalability, multi-tenancy, and security without modifying the DL frameworks. Our users report that they are able to train their models much more quickly and conveniently, and we observe that the system has been able to scale and is ready for future growth.

For future work, we plan to better understand the usage patterns of our users and optimize the system for these workloads. We are also aware that model training is one step in the overall machine learning pipeline, and plan to better integrate with the end-to-end flow. With the rapid pace of innovation in the deep learning space, we envision that in the near future users will be working in fully integrated ML development and debugging environments, with interactive exploration of the problem space, and (semi-)automated recommendations on how to fine-tune their algorithms. This will open up an entirely new field of exciting new challenges for machine learning systems researchers.

References

[1] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1223–1231. 2012.

[2] Bishwaranjan Bhattacharjee, Scott Boag, Chandani Doshi, Parijat Dube, Ben Herta, Vatche Ishakian, K. R. Jayaram, Rania Khalaf, Avesh Krishna, Yu Bo Li, Vinod Muthusamy, Ruchir Puri, Yufei Ren, Florian Rosenberg, Seetharami R. Seelam, Yandong Wang, Jian Ming Zhang, and Li Zhang. IBM Deep Learning Service. *IBM Journal of Research and Development*, 61(4):10–1, 2017.

[3] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 193–204. ACM, 2010.

[4] Chris Richardson. *Pattern: Microservice Architecture*, 2017.

[5] gRPC: A high performance, open-source universal RPC framework.

[6] Google Inc. *Kubernetes: Production Grade Container Orchestration*, 2017.

[7] CoreOS. *The ETCD Distributed Key-Value Store*, 2017.

[8] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning? *arXiv preprint arXiv:1611.04581*, 2016.

[9] Xinghao Pan, Shivaram Venkataraman, Zizheng Tai, and Joseph Gonzalez. Hemingway: Modeling distributed optimization algorithms. *arXiv preprint arXiv:1702.05865*, 2017.

[10] Li Erran Li, Eric Chen, Jeremy Hermann, Pusheng Zhang, and Luming Wang. Scaling machine learning as a service. In *3rd International Conference on Predictive Applications and APIs*, volume 67 of *Proceedings of Machine Learning Research (PMLR)*, pages 14–29, 2017.

[11] James B.D. Joshi, Hassan Takabi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security and Privacy*, 8:24–31, 2010.

[12] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security Symposium*, pages 601–618, 2016.

[13] Jiawei Yuan and Shucheng Yu. Privacy preserving back-propagation neural network learning made practical with cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):212–221, 2014.

[14] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. ModelDB: A System for Machine Learning Model Management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

[15] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *ACM Symposium on Cloud Computing*, pages 390–404, 2017.

[16] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 363–378. USENIX Association, 2016.

[17] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *HotNets*, pages 50–56, 2016.

[18] 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.

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