Real-Time Machine Learning: The Missing Pieces

Alexey Tumanov
Georgia Institute of Technology
43 PUBLICATIONS **2,879** CITATIONS
SEE PROFILE

Michael Jordan
University of California, Berkeley
955 PUBLICATIONS **208,721** CITATIONS
SEE PROFILE

Ion Stoica
University of California, Berkeley
448 PUBLICATIONS **86,828** CITATIONS
SEE PROFILE

Real-Time Machine Learning: The Missing Pieces

Robert Nishihara*, Philipp Moritz*, Stephanie Wang, Alexey Tumanov, William Paul,
Johann Schleier-Smith, Richard Liaw, Michael I. Jordan, Ion Stoica
UC Berkeley

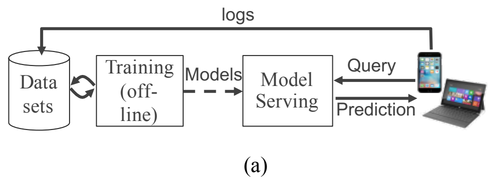
Abstract

Machine learning applications are increasingly deployed not only to serve predictions using static models, but also as tightly-integrated components of feedback loops involving dynamic, real-time decision making. These applications pose a new set of requirements, none of which are difficult to achieve in isolation, but the combination of which creates a challenge for existing distributed execution frameworks: computation with millisecond latency at high throughput, adaptive construction of arbitrary task graphs, and execution of heterogeneous kernels over diverse sets of resources. We assert that a new distributed execution framework is needed for such ML applications and propose a candidate approach with a proof-of-concept architecture that achieves a 63x performance improvement over a state-of-the-art execution framework for a representative application.

1 Introduction

The landscape of machine learning (ML) applications is undergoing a significant change. While ML has predominantly focused on training and serving predictions based on static models (Figure 1a), there is now a strong shift toward the tight integration of ML models in feedback loops. Indeed, ML applications are expanding from the supervised learning paradigm, in which static models are trained on offline data, to a broader paradigm, exemplified by reinforcement learning (RL), in which applications may operate in real environments, fuse and react to sensory data from numerous input streams, perform continuous micro-simulations, and close the loop by taking actions that affect the sensed environment (Figure 1b).

*equal contribution



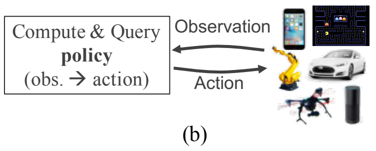
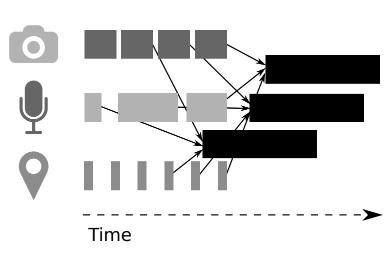


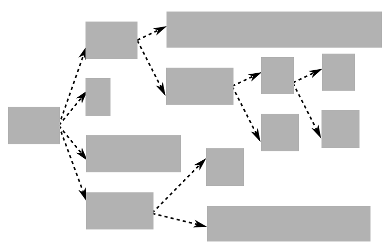
Figure 1: (a) Traditional ML pipeline (off-line training). (b) Example reinforcement learning pipeline: the system continously interacts with an environment to learn a policy, i.e., a mapping between observations and actions.

Since learning by interacting with the real world can be unsafe, impractical, or bandwidth-limited, many reinforcement learning systems rely heavily on **simulating physical or virtual environments**. Simulations may be used during training (e.g., to learn a neural network policy), and during deployment. In the latter case, we may constantly update the simulated environment as we interact with the real world and perform many simulations to figure out the next action (e.g., using online planning algorithms like Monte Carlo tree search). This requires the ability to perform simulations faster than real time.

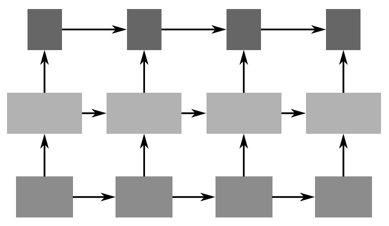
Such emerging applications require new levels of programming flexibility and performance. Meeting these requirements without losing the benefits of modern distributed execution frameworks (e.g., application-level fault tolerance) poses a significant challenge. Our own experience implementing ML and RL applications in Spark, MPI, and TensorFlow highlights some of these challenges



(a) multiple sensor inputs



(b) Monte Carlo tree search (MCTS)



(c) Recurrent Neural Network (RNN)

Figure 2: Example components of a real-time ML application: (a) online processing of streaming sensory data to model the environment, (b) dynamic graph construction for Monte Carlo tree search (here tasks are simulations exploring sequences of actions), and (c) heterogeneous tasks in recurrent neural networks. Different shades represent different types of tasks, and the task lengths represent their durations.

and gives rise to three groups of requirements for supporting these applications. *Though these requirements are critical for ML and RL applications, we believe they are broadly useful.*

Performance Requirements. Emerging ML applications have stringent latency and throughput requirements.

- **R1:** *Low latency*. The real-time, reactive, and interactive nature of emerging ML applications calls for fine-granularity task execution with millisecond end-to-end latency [8].
- **R2:** *High throughput*. The volume of micro-simulations required both for training [16] as well as for inference during deployment [19] necessitates support for high-throughput task execution on the order of millions of tasks per second.

Execution Model Requirements. Though many existing parallel execution systems [9, 21] have gotten great mileage out of identifying and optimizing for common computational patterns, emerging ML applications require far greater flexibility [10].

- **R3:** *Dynamic task creation*. RL primitives such as Monte Carlo tree search may generate new tasks during execution based on the results or the durations of other tasks.
- **R4:** *Heterogeneous tasks*. Deep learning primitives and RL simulations produce tasks with widely different execution times and resource requirements. Explicit system support for heterogeneity of tasks and resources is essential for RL applications.
- **R5:** *Arbitrary dataflow dependencies*. Similarly, deep learning primitives and RL simulations produce arbitrary and often fine-grained task dependencies (not restricted to bulk synchronous parallel).

Practical Requirements.

- **R6:** *Transparent fault tolerance*. Fault tolerance remains a key requirement for many deployment scenarios, and supporting it alongside high-throughput and non-deterministic tasks poses a challenge.
- **R7:** *Debuggability and Profiling*. Debugging and performance profiling are the most time-consuming aspects of writing any distributed application. This is especially true for ML and RL applications, which are often compute-intensive and stochastic.

Existing frameworks fall short of achieving one or more of these requirements (Section 5). We propose a flexible distributed programming model (Section 3.1) to enable **R3-R5**. In addition, we propose a system architecture to support this programming model and meet our performance requirements (**R1-R2**) without giving up key practical requirements (**R6-R7**). The proposed system architecture (Section 3.2) builds on two principal components: a logically-centralized control plane and a hybrid scheduler. The former enables stateless distributed components and lineage replay. The latter allocates resources in a bottom-up fashion, splitting locally-born work between node-level and cluster-level schedulers.

The result is millisecond-level performance on microbenchmarks and a 63x end-to-end speedup on a representative RL application over a bulk synchronous parallel (BSP) implementation.

2 Motivating Example

To motivate requirements **R1-R7**, consider a hypothetical application in which a physical robot attempts to achieve a goal in an unfamiliar real-world environment. Various sensors may fuse video and LIDAR input to build multiple candidate models of the robot's environment (Fig. 2a). The robot is then controlled in real time using actions informed by a recurrent neural network (RNN) *policy* (Fig. 2c), as well as by Monte Carlo tree search (MCTS) and other online planning algorithms (Fig. 2b). Using a physics simulator along with the most recent environment models, MCTS tries millions of action sequences in parallel, adaptively exploring the most promising ones.

The Application Requirements. Enabling these kinds of applications involves simultaneously solving a number of challenges. In this example, the latency requirements (**R1**) are stringent, as the robot must be controlled in real time. High task throughput (**R2**) is needed to support the online simulations for MCTS as well as the streaming sensory input.

Task heterogeneity (**R4**) is present on many scales: some tasks run physics simulators, others process diverse data streams, and some compute actions using RNN-based policies. Even similar tasks may exhibit substantial variability in duration. For example, the RNN consists of different functions for each "layer", each of which may require different amounts of computation. Or, in a task simulating the robot's actions, the simulation length may depend on whether the robot achieves its goal or not.

In addition to the heterogeneity of tasks, the dependencies between tasks can be complex (**R5**, Figs. 2a and 2c) and difficult to express as batched BSP stages.

Dynamic construction of tasks and their dependencies (**R3**) is critical. Simulations will adaptively use the most recent environment models as they become available, and MCTS may choose to launch more tasks exploring particular subtrees, depending on how promising they are or how fast the computation is. Thus, the dataflow graph must be constructed dynamically in order to allow the algorithm to adapt to real-time constraints and opportunities.

3 Proposed Solution

In this section, we outline a proposal for a distributed execution framework and a programming model satisfying requirements **R1-R7** for real-time ML applications.

3.1 API and Execution Model

In order to support the execution model requirements (**R3-R5**), we outline an API that allows arbitrary functions to be specified as remotely executable tasks, with dataflow dependencies between them.

1. Task creation is non-blocking. When a *task* is created, a *future* [4] representing the eventual return value of the task is returned immediately, and the task is executed asynchronously.

2. Arbitrary function invocation can be designated as a remote task, making it possible to support arbitrary execution kernels (**R4**). Task arguments can be either regular values or futures. When an argument is a future, the newly created task becomes dependent on the task that produces that future, enabling arbitrary DAG dependencies (**R5**).

3. Any task execution can create new tasks without blocking on their completion. Task throughput is therefore not limited by the bandwidth of any one worker (**R2**), and the computation graph is dynamically built (**R3**).

4. The actual return value of a task can be obtained by calling the `get` method on the corresponding future. This blocks until the task finishes executing.

5. The `wait` method takes a list of futures, a timeout, and a number of values. It returns the subset of futures whose tasks have completed when the timeout occurs or the requested number have completed.

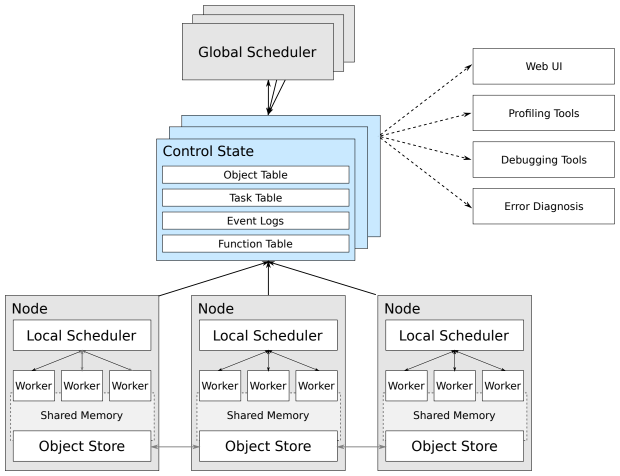


Figure 3: Proposed Architecture, with hybrid scheduling (Section 3.2.2) and a centralized control plane (Section 3.2.1).

The `wait` primitive allows developers to specify latency requirements (**R1**) with a timeout, accounting for arbitrarily sized tasks (**R4**). This is important for ML applications, in which a straggler task may produce negligible algorithmic improvement but block the entire computation. This primitive enhances our ability to dynamically modify the computation graph as a function of execution-time properties (**R3**).

To complement the fine-grained programming model, we propose using a dataflow execution model in which tasks become available for execution if and only if their dependencies have finished executing.

3.2 Proposed Architecture

Our proposed architecture consists of multiple *worker* processes running on each node in the cluster, one *local scheduler* per node, one or more *global schedulers* throughout the cluster, and an in-memory *object store* for sharing data between workers (see Figure 3).

The two principal architectural features that enable **R1-R7** are a *hybrid scheduler* and a *centralized control plane*.

3.2.1 Centralized Control State

As shown in Figure 3, our architecture relies on a logically-centralized control plane [13]. To realize this architecture, we use a database that provides both (1) storage for the system's control state, and (2) publish-subscribe functionality to enable various system components to communicate with each other.[1]

This design enables virtually any component of the system, except for the database, to be stateless. This means that as long as the database is fault-tolerant, we can recover from component failures by simply restarting the failed components. Furthermore, the database stores the computation lineage, which allows us to reconstruct lost data by replaying the computation [21]. As a result, this design is fault tolerant (**R6**). The database also makes it easy to write tools to profile and inspect the state of the system (**R7**).

To achieve the throughput requirement (**R2**), we shard the database. Since we require only exact matching operations and since the keys are computed as hashes, sharding is relatively straightforward. Our early experiments show that this design enables sub-millisecond scheduling latencies (**R1**).

3.2.2 Hybrid Scheduling

Our requirements for latency (**R1**), throughput (**R2**), and dynamic graph construction (**R3**) naturally motivate a hybrid scheduler in which local schedulers assign tasks to workers or delegate responsibility to one or more global schedulers.

Workers submit tasks to their local schedulers which decide to either assign the tasks to other workers on the same physical node or to "spill over" the tasks to a global scheduler. Global schedulers can then assign tasks to local schedulers based on global information about factors including object locality and resource availability.

Since tasks may create other tasks, schedulable work may come from any worker in the cluster. Enabling any local scheduler to handle locally generated work without involving a global scheduler improves low latency (**R1**), by avoiding communication overheads, and throughput (**R2**), by significantly reducing the global scheduler load. This hybrid scheduling scheme fits well with the recent trend toward large multicore servers [20].

[1]In our implementation we employ Redis [18], although many other fault-tolerant key-value stores could be used.

4 Feasibility

To demonstrate that these API and architectural proposals could in principle support requirements **R1-R7**, we provide some simple examples using the preliminary system design outlined in Section 3.

4.1 Latency Microbenchmarks

Using our prototype system, a task can be created, meaning that the task is submitted asynchronously for execution and a future is returned, in around 35μs. Once a task has finished executing, its return value can be retrieved in around 110μs. The end-to-end time, from submitting an empty task for execution to retrieving its return value, is around 290μs when the task is scheduled locally and 1ms when the task is scheduled on a remote node.

4.2 Reinforcement Learning

We implement a simple workload in which an RL agent is trained to play an Atari game. The workload alternates between stages in which actions are taken in parallel simulations and actions are computed in parallel on GPUs. Despite the BSP nature of the example, an implementation in Spark is **9x** slower than the single-threaded implementation due to system overhead. An implementation in our prototype is **7x** faster than the single-threaded version and **63x** faster than the Spark implementation.[2]

This example exhibits two key features. First, tasks are very small (around 7ms each), making low task overhead critical. Second, the tasks are heterogeneous in duration and in resource requirements (e.g., CPUs and GPUs).

This example is just one component of an RL workload, and would typically be used as a subroutine of a more sophisticated (non-BSP) workload. For example, using the `wait` primitive, we can adapt the example to process the simulation tasks in the order that they finish so as to better pipeline the simulation execution with the action computations on the GPU, or run the entire workload nested within a larger adaptive hyperparameter search. These changes are all straightforward using the API described in Section 3.1 and involve a few extra lines of code.

[2]In this comparison, the GPU model fitting could not be naturally parallelized on Spark, so the numbers are reported as if it had been perfectly parallelized with no overhead in Spark.

5 Related Work

Static dataflow systems [9, 21, 12, 14] are well-established in analytics and ML, but they require the dataflow graph to be specified upfront, e.g., by a driver program. Some, like MapReduce [9] and Spark [21], emphasize BSP execution, while others, like Dryad [12] and Naiad [14], support complex dependency structures (**R5**). Others, such as TensorFlow [1] and MXNet [6], are optimized for deep-learning workloads. However, none of these systems fully support the ability to dynamically extend the dataflow graph in response to both input data and task progress (**R3**).

Dynamic dataflow systems like CIEL [15] and Dask [17] support many of the same features as static dataflow systems, with additional support for dynamic task creation **(R3)**. These systems meet our execution model requirements (**R3-R5**). However, their architectural limitations, such as entirely centralized scheduling, are such that low latency (**R1**) must often be traded off with high throughput (**R2**) (e.g., via batching), whereas our applications require both.

Other systems like Open MPI [11] and actor-model variants Orleans [5] and Erlang [3] provide low-latency (**R1**) and high-throughput (**R2**) distributed computation. Though these systems do in principle provide primitives for supporting our execution model requirements (**R3-R5**) and have been used for ML [7, 2], much of the logic required for systems-level features, such as fault tolerance (**R6**) and locality-aware task scheduling, must be implemented at the application level.

6 Conclusion

Machine learning applications are evolving to require dynamic dataflow parallelism with millisecond latency and high throughput, posing a severe challenge for existing frameworks. We outline the requirements for supporting this emerging class of real-time ML applications, and we propose a programming model and architectural design to address the key requirements (**R1-R5**), without compromising existing requirements (**R6-R7**). Preliminary, proof-of-concept results confirm millisecond-level system overheads and meaningful speedups for a representative RL application.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 265–283.

[2] Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595* (2015).

[3] Armstrong, J., Virding, R., Wikström, C., and Williams, M. Concurrent programming in ERLANG.

[4] Baker, Jr., H. C., and Hewitt, C. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages* (New York, NY, USA, 1977), ACM, pp. 55–59.

[5] Bykov, S., Geller, A., Kliot, G., Larus, J. R., Pandya, R., and Thelin, J. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM, p. 16.

[6] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NIPS Workshop on Machine Learning Systems (LearningSys'16)* (2016).

[7] Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. Deep learning with COTS HPC systems. In *Proceedings of The 30th International Conference on Machine Learning* (2013), pp. 1337–1345.

[8] Crankshaw, D., Bailis, P., Gonzalez, J. E., Li, H., Zhang, Z., Franklin, M. J., Ghodsi, A., and Jordan, M. I. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. *arXiv preprint arXiv:1409.3809* (2014).

[9] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[10] Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)* (2016).

[11] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[12] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72.

[13] Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE 103*, 1 (2015), 14–76.

[14] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[15] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 113–126.

[16] Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., Maria, A. D., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., and Silver, D. Massively parallel methods for deep reinforcement learning, 2015.

[17] Rocklin, M. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference* (2015), K. Huff and J. Bergstra, Eds., pp. 130 – 136.

[18] Sanfilippo, S. Redis: An open source, in-memory data structure store. `https://redis.io/`, 2009.

[19] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of Go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[20] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev. 43*, 2 (Apr. 2009), 76–85.

[21] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache Spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.