RLlib: Abstractions for Distributed Reinforcement Learning

Eric Liang [*1] **Richard Liaw** [*1] **Philipp Moritz** [1] **Robert Nishihara** [1] **Roy Fox** [1] **Ken Goldberg** [1]
Joseph E. Gonzalez [1] **Michael I. Jordan** [1] **Ion Stoica** [1]

Abstract

Reinforcement learning (RL) algorithms involve the deep nesting of highly irregular computation patterns, each of which typically exhibits opportunities for distributed computation. We argue for distributing RL components in a composable way by adapting algorithms for top-down hierarchical control, thereby encapsulating parallelism and resource requirements within short-running compute tasks. We demonstrate the benefits of this principle through RLlib: a library that provides scalable software primitives for RL. These primitives enable a broad range of algorithms to be implemented with high performance, scalability, and substantial code reuse. RLlib is available as part of the open source Ray project [1].

1. Introduction

Advances in parallel computing and composition through symbolic differentiation have been fundamental to the recent success of deep learning. Today, there are a wide range of deep learning frameworks (Paszke et al., 2017; Abadi et al., 2016; Chen et al., 2016; Jia et al., 2014) that enable rapid innovation in neural network design and facilitate training at the scale necessary for progress in the field.

In contrast, while the reinforcement learning community enjoys the advances in systems and abstractions for deep learning, there has been comparatively less progress in the design of systems and abstractions that directly target reinforcement learning. Nonetheless, many of the challenges in reinforcement learning stem from the need to scale learning and simulation while also integrating a rapidly increasing range of algorithms and models. As a consequence, there is a fundamental need for composable parallel primitives to support research in reinforcement learning.

In the absence of a single dominant computational pattern (e.g., tensor algebra) or fundamental rules of composition (e.g., symbolic differentiation), the design and implementation of reinforcement learning algorithms can often be cumbersome, requiring RL researchers to directly reason about complex nested parallelism. Unlike typical operators in deep learning frameworks, individual components may require parallelism across a cluster (e.g., for rollouts), leverage neural networks implemented by deep learning frameworks, recursively invoke other components (e.g., model-based subtasks), or interface with black-box third-party simulators. In essence, the heterogeneous and distributed nature of many of these components poses a key challenge to reasoning about their parallel composition. Meanwhile, the main algorithms that connect these components are rapidly evolving and expose opportunities for parallelism at varying levels. Finally, RL algorithms manipulate substantial amounts of state (e.g., replay buffers and model parameters) that must be managed across multiple levels of parallelism and physical devices.

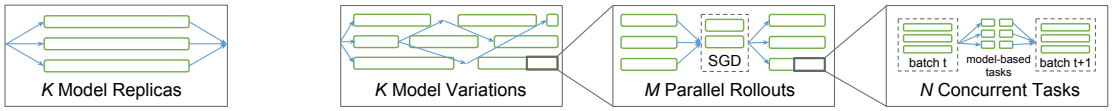
The substantial recent progress in RL algorithms and applications has resulted in a large and growing number of RL libraries (Caspi, 2017; Duan et al., 2016; Hafner et al., 2017; Hesse et al., 2017; Kostrikov, 2017; Schaarschmidt et al., 2017). While some of these are highly scalable, few enable the composition of components at scale. In large part, this is due to the fact that many of the frameworks used by these libraries rely on communication between long-running program replicas for distributed execution; e.g., MPI (Gropp et al., 1996), Distributed TensorFlow (Abadi et al., 2016), and parameter servers (Li et al., 2014)). As this programming model ignores component boundaries, it does not naturally encapsulate parallelism and resource requirements within individual components.[2] As a result, reusing these distributed components requires the insertion of appropriate control points in the program, a burdensome and error-prone process (Section 2). The absence of usable encapsulation hinders code reuse and leads to error prone reimplementation of mathematically complex and often highly stochastic algorithms. Even worse, in the distributed setting, often

[*]Equal contribution [1]University of California, Berkeley. Correspondence to: Eric Liang <ericliang@berkeley.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

[1]RLlib documentation can be found at http://rllib.io

[2]By *encapsulation*, we mean that individual components specify their own internal parallelism and resources requirements and can be used by other components that have no knowledge of these requirements.



(a) Deep Learning (b) Reinforcement Learning

Figure 1. In contrast with deep learning, RL algorithms leverage parallelism at multiple levels and physical devices. Here, we show an RL algorithm composing derivative-free optimization, policy evaluation, gradient-based optimization, and model-based planning (Table 2).

large parts of the distributed communication and execution must also be reimplemented with each new RL algorithm.

We believe that the ability to build scalable RL algorithms by composing and reusing existing components and implementations is essential for the rapid development and progress of the field.[3] Toward this end, we argue for structuring distributed RL components around the principles of logically centralized program control and parallelism encapsulation (Graefe & Davison, 1993; Pan et al., 2010). We built RLlib using these principles, and as a result were not only able to implement a broad range of state-of-the-art RL algorithms, but also to pull out scalable primitives that can be used to easily compose new algorithms.

1.1. Irregularity of RL training workloads

Modern RL algorithms are highly irregular in the computation patterns they create (Table 1), pushing the boundaries of computation models supported by popular distribution frameworks. This irregularity occurs at several levels:

1. The duration and resource requirements of tasks differ by orders of magnitude depending on the algorithm; e.g., A3C (Mnih et al., 2016) updates may take milliseconds, but other algorithms like PPO (Schulman et al., 2017) batch rollouts into much larger granularities.
2. Communication patterns vary, from synchronous to asynchronous gradient-based optimization, to having several types of asynchronous tasks in high-throughput off-policy algorithms such as Ape-X and IMPALA (Horgan et al., 2018; Espeholt et al., 2018).
3. Nested computations are generated by model-based hybrid algorithms (Table 2), hyperparameter tuning in conjunction with RL or DL training, or the combination of derivative-free and gradient-based optimization within a single algorithm (Silver et al., 2017).
4. RL algorithms often need to maintain and update substantial amounts of state including policy parameters, replay buffers, and even external simulators.

As a consequence, the developers have no choice but to use a hodgepodge of frameworks to implement their algorithms, including parameter servers, collective communication primitives in MPI-like frameworks, task queues, etc. For more complex algorithms, it is common to build custom distributed systems in which processes independently compute and coordinate among themselves with no central control (Figure 2(a)). While this approach can achieve high performance, the cost to develop and evaluate is large, not only due to the need to implement and debug distributed programs, but because composing these algorithms further complicates their implementation (Figure 3). Moreover, today's computation frameworks (e.g., Spark (Zaharia et al., 2010), MPI) typically assume regular computation patterns and have difficulty when sub-tasks have varying durations, resource requirements, or nesting.

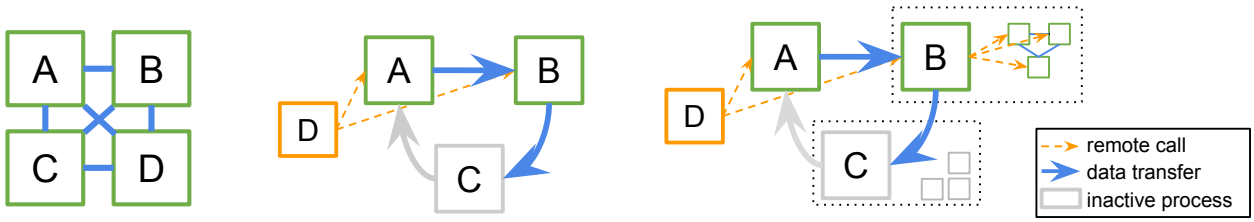
[3] We note that composability *without* scalability can trivially be achieved with a single-threaded library and that all of the difficulty lies in achieving these two objectives simultaneously.

Table 1. RL spans a broad range of computational demand.

Dimension	DQN/Laptop	IMPALA+PBT/Cluster
Task Duration	∼1ms	minutes
Task Compute	1 CPU	several CPUs and GPUs
Total Compute	1 CPU	hundreds of CPUs and GPUs
Nesting Depth	1 level	3+ levels
Process Memory	megabytes	hundreds of gigabytes
Execution	synchronous	async. and highly concurrent

1.2. Logically centralized control for distributed RL

It is desirable for a single programming model to capture all the requirements of RL training. This can be done without eschewing high-level frameworks that structure the computation. Our key insight is that for each distributed RL algorithm, an equivalent algorithm can be written that exhibits logically centralized program control (Figure 2(b)). That is, instead of having independently executing processes (**A**, **B**, **C**, **D** in Figure 2(a)) coordinate among themselves (e.g., through RPCs, shared memory, parameter servers, or collective communication), a single *driver program* (**D** in Figure 2(b) and 2(c)) can delegate algorithm sub-tasks to other processes to execute in parallel. In this paradigm, the worker processes **A**, **B**, and **C** passively hold state (e.g., policy or simulator state) but execute no computations until called by **D**. To support nested computations, we propose extending the centralized control model with *hierarchical delegation of control* (Figure 2(c)), which allows the worker processes (e.g., **B**, **C**) to further delegate work (e.g., simulations, gradient computation) to sub-workers of their own when executing tasks.



(a) Distributed Control (b) Logically Centralized Control (c) Hierarchical Control

Figure 2. Most RL algorithms today are written in a fully distributed style (a) where replicated processes independently compute and coordinate with each other according to their roles (if any). We propose a hierarchical control model (c), which extends (b) to support nesting in RL and hyperparameter tuning workloads, simplifying and unifying the programming models used for implementation.

Building on such a logically centralized and hierarchical control model has several important advantages. First, the equivalent algorithm is often easier to implement in practice, since the distributed control logic is entirely encapsulated in a single process rather than multiple processes executing concurrently. Second, the separation of algorithm components into sub-routines (e.g., do rollouts, compute gradients with respect to some policy loss), enables code reuse across different execution patterns. Sub-tasks that have different resource requirements (e.g., CPUs vs GPUs) can be placed on different machines, reducing compute costs as we show in Section 5. Finally, distributed algorithms written in this model can be seamlessly nested within each other, satisfying the parallelism encapsulation principle.

Logically centralized control models can be highly performant, our proposed hierarchical variant even more so. This is because the bulk of data transfer (blue arrows in Figure 2) between processes happens out of band of the driver, not passing through any central bottleneck. In fact many highly scalable distributed systems (Zaharia et al., 2010; Chang et al., 2008; Dean & Ghemawat, 2008) leverage centralized control in their design. Within a single differentiable tensor graph, frameworks like TensorFlow also implement logically centralized scheduling of tensor computations onto available physical devices. Our proposal extends this principle into the broader ML systems design space.

The contributions of this paper are as follows.

1. We propose a general and composable hierarchical programming model for RL training (Section 2).
2. We describe RLlib, our highly scalable RL library, and how it builds on the proposed model to provide scalable abstractions for a broad range of RL algorithms, enabling rapid development (Section 3).
3. We discuss how performance is achieved within the proposed model (Section 4), and show that RLlib meets or exceeds state-of-the-art performance for a wide variety of RL workloads (Section 5).

2. Hierarchical Parallel Task Model

As highlighted in Figure 3, parallelization of entire programs using frameworks like MPI (Gropp et al., 1996) and Distributed Tensorflow (Abadi et al., 2016) typically require explicit algorithm modifications to insert points of coordination when trying to compose two programs or components together. This limits the ability to rapidly prototype novel distributed RL applications. Though the example in Figure 3 is simple, new hyperparameter tuning algorithms for long-running training tasks; e.g., HyperBand, Population Based Training (PBT) (Li et al., 2016; Jaderberg et al., 2017) increasingly demand fine-grained control over training.

We propose building RL libraries with hierarchical and logically centralized control on top of flexible task-based programming models like Ray (Moritz et al., 2017). Task-based systems allow subroutines to be scheduled and executed asynchronously on worker processes, on a fine-grained basis, and for results to be retrieved or passed between processes.

2.1. Relation to existing distributed ML abstractions

Though typically formulated for distributed control, abstractions such as parameter servers and collective communication operations can also be used within a logically centralized control model. As an example, RLlib uses allreduce and parameter-servers in some of its policy optimizers (Figure 4), and we evaluate their performance in Section 5.

2.2. Ray implementation of hierarchical control

We note that, within a single machine, the proposed programming model can be implemented simply with thread-pools and shared memory, though it is desirable for the underlying framework to scale to larger clusters if needed.

We chose to build RLlib on top of the Ray framework, which allows Python tasks to be distributed across large clusters. Ray's distributed scheduler is a natural fit for the hierarchical control model, as nested computation can be implemented in Ray with no central task scheduling bottleneck.

```
if mpi.get_rank() <= m:
    grid = mpi.comm_world.split(0)
else:
    eval = mpi.comm_world.split(
        mpi.get_rank() % n)
...
if mpi.get_rank() == 0:
    grid.scatter(
        generate_hyperparams(), root=0)
    print(grid.gather(root=0))
elif 0 < mpi.get_rank() <= m:
    params = grid.scatter(None, root=0)
    eval.bcast(
        generate_model(params), root=0)
    results = eval.gather(
        result, root=0)
    grid.gather(results, root=0)
elif mpi.get_rank() > m:
    model = eval.bcast(None, root=0)
    result = rollout(model)
    eval.gather(result, root=0)
```

(a) Distributed Control

```
@ray.remote
def rollout(model):
    # perform a rollout and
    # return the result

@ray.remote
def evaluate(params):
    model = generate_model(params)
    results = [rollout.remote(model)
       for i in range(n)]
    return results

param_grid = generate_hyperparams()
print(ray.get([evaluate.remote(p)
   for p in param_grid]))
```

(b) Hierarchical Control

Figure 3. Composing a distributed hyperparameter search with a function that also requires distributed computation involves *complex nested parallel computation patterns*. With MPI (a), a new program must be written from scratch that mixes elements of both. With hierarchical control (b), components can remain unchanged and simply be invoked as remote tasks.

To implement a logically centralized control model, it is first necessary to have a mechanism to launch new processes and schedule tasks on them. Ray meets this requirement with *Ray actors*, which are Python classes that may be created in the cluster and accept remote method calls (i.e., tasks). Ray permits these actors to in turn launch more actors and schedule tasks on those actors as part of a method call, satisfying our need for hierarchical delegation as well.

For performance, Ray provides standard communication primitives such as `aggregate` and `broadcast`, and critically enables the *zero-copy* sharing of large data objects through a shared memory object store. As shown in Section 5, this enables the performance of RLlib algorithms. We further discuss framework performance in Section 4.

3. Abstractions for Reinforcement Learning

To leverage RLlib for distributed execution, algorithms must declare their policy π, experience postprocessor ρ, and loss L. These can be specified in any deep learning framework, including TensorFlow and PyTorch. RLlib provides *policy evaluators* and *policy optimizers* that implement strategies for distributed policy evaluation and training.

3.1. Defining the Policy Graph

RLlib's abstractions are as follows. The developer specifies a policy model π that maps the current observation o_t and (optional) RNN hidden state h_t to an action a_t and the next RNN state h_{t+1}. Any number of user-defined values y_t^i (e.g., value predictions, TD error) can also be returned:

$$\pi_\theta(o_t, h_t) \Rightarrow (a_t, h_{t+1}, y_t^1 \dots y_t^N) \quad (1)$$

Most algorithms will also specify a trajectory postprocessor ρ that transforms a batch $X_{t,K}$ of K $\{(o_t, h_t, a_t, h_{t+1}, y_t^1 \dots y_t^N, r_t, o_{t+1})\}$ tuples starting at t. Here r_t and o_{t+1} are the reward and new observation after taking an action. Example uses include advantage estimation (Schulman et al., 2015) and goal relabeling (Andrychowicz et al., 2017). To also support multi-agent environments, experience batches $X_{t,K}^p$ from the P other agents in the environment are also made accessible:

$$\rho_\theta(X_{t,K}, X_{t,K}^1 \dots X_{t,K}^P) \Rightarrow X_{post} \quad (2)$$

Gradient-based algorithms define a combined loss L that can be descended to improve the policy and auxiliary networks:

$$L(\theta; X) \Rightarrow loss \quad (3)$$

Finally, the developer can also specify any number of utility functions u^i to be called as needed during training to, e.g., return training statistics s, update target networks, or adjust annealing schedules:

$$u^1 \dots u^M(\theta) \Rightarrow (s, \theta_{update}) \quad (4)$$

To interface with RLlib, these algorithm functions should be defined in a *policy graph* class with the following methods:

```
abstract class rllib.PolicyGraph:
  def act(self, obs, h): action, h, y*
  def postprocess(self, batch, b*): batch
  def gradients(self, batch): grads
  def get_weights; def set_weights;
  def u*(self, args*)
```

3.2. Policy Evaluation

For collecting experiences, RLlib provides a `PolicyEvaluator` class that wraps a policy graph and environment to add a method to `sample()` experience batches. Policy evaluator instances can be created as Ray remote actors and *replicated* across a cluster for parallelism. To make their usage concrete, consider a minimal TensorFlow policy gradients implementation that extends the `rllib.TFPolicyGraph` helper template:

```
class PolicyGradient(TFPolicyGraph):
  def __init__(self, obs_space, act_space):
    self.obs, self.advantages = ...
    pi = FullyConnectedNetwork(self.obs)
    dist = rllib.action_dist(act_space, pi)
    self.act = dist.sample()
    self.loss = -tf.reduce_mean(
      dist.logp(self.act) * self.advantages)
  def postprocess(self, batch):
    return rllib.compute_advantages(batch)
```

From this policy graph definition, the developer can create a number of policy evaluator replicas `ev` and call `ev.sample.remote()` on each to collect experiences in parallel from environments. RLlib supports OpenAI Gym (Brockman et al., 2016), user-defined environments, and also batched simulators such as ELF (Tian et al., 2017):

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
avg_grad = aggregate(grads)
local_graph.apply(avg_grad)
weights = broadcast(
    local_graph.weights())
for ev in evaluators:
    ev.set_weights(weights)
```

(a) Allreduce

```
samples = concat([ev.sample()
    for ev in evaluators])
pin_in_local_gpu_memory(samples)
for _ in range(NUM_SGD_EPOCHS):
   local_g.apply(local_g.grad(samples)
weights = broadcast(local_g.weights())
for ev in evaluators:
    ev.set_weights(weights)
```

(b) Local Multi-GPU

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
    grad, ev, grads = wait(grads)
    local_graph.apply(grad)
    ev.set_weights(
        local_graph.get_weights())
    grads.append(ev.grad(ev.sample()))
```

(c) Asynchronous

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
    grad, ev, grads = wait(grads)
    for ps, g in split(grad, ps_shards):
        ps.push(g)
    ev.set_weights(concat(
        [ps.pull() for ps in ps_shards])
    grads.append(ev.grad(ev.sample()))
```

(d) Sharded Param-server

Figure 4. Pseudocode for four RLlib policy optimizer step methods. Each step() operates over a local policy graph and array of remote evaluator replicas. Ray remote calls are highlighted in orange; other Ray primitives in blue (Section 4). *Apply* is shorthand for updating weights. Minibatch code and helper functions omitted. The param server optimizer in RLlib also implements pipelining not shown here.

```
evaluators = [rllib.PolicyEvaluator.remote(
   env=SomeEnv, graph=PolicyGradient)
  for _ in range(10)]
print(ray.get([
   ev.sample.remote() for ev in evaluators]))
```

3.3. Policy Optimization

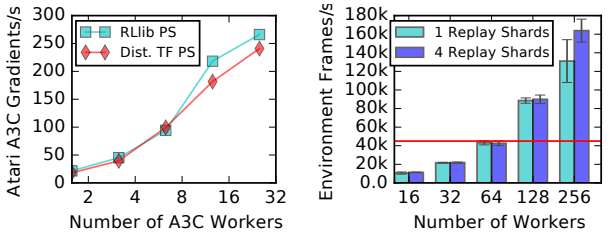
RLlib separates the implementation of algorithms into the declaration of the algorithm-specific *policy graph* and the choice of an algorithm-independent *policy optimizer*. The policy optimizer is responsible for the performance-critical tasks of distributed sampling, parameter updates, and managing replay buffers. To distribute the computation, the optimizer operates over a set of policy evaluator replicas.

To complete the example, the developer chooses a policy optimizer and creates it with references to existing evaluators. The async optimizer uses the evaluator actors to compute gradients in parallel on many CPUs (Figure 4(c)). Each `optimizer.step()` runs a round of remote tasks to improve the model. Between steps, policy graph replicas can be queried directly, e.g., to print out training statistics:

```
optimizer = rllib.AsyncPolicyOptimizer(
   graph=PolicyGradient, workers=evaluators)
while True:
   optimizer.step()
   print(optimizer.foreach_policy(
      lambda p: p.get_train_stats()))
```

Policy optimizers extend the well-known gradient-descent optimizer abstraction to the RL domain. A typical gradient-descent optimizer implements $step(L(\theta), X, \theta) \Rightarrow \theta_{opt}$. RLlib's policy optimizers instead operate over the local policy graph G and a set of remote evaluator replicas, i.e., $step(G, ev_1 \ldots ev_n, \theta) \Rightarrow \theta_{opt}$, capturing the sampling phase of RL as part of optimization (i.e., calling `sample()` on policy evaluators to produce new simulation data).

The policy optimizer abstraction has the following advantages. By separating execution strategy from policy and loss definitions, specialized optimizers can be swapped in to take advantage of available hardware or algorithm features without needing to change the rest of the algorithm. The policy graph class encapsulates interaction with the deep learning framework, allowing algorithm authors to avoid mixing distributed systems code with numerical computations, and enabling optimizer implementations to be improved and reused across different deep learning frameworks.



(a) Sharded Param. Server (b) Ape-X in RLlib

Figure 5. RLlib's centrally controlled policy optimizers match or exceed the performance of implementations in specialized systems. The RLlib parameter server optimizer using 8 internal shards is competitive with a Distributed TensorFlow implementation tested in similar conditions. RLlib's Ape-X policy optimizer scales to 160k frames per second with 256 workers at a frameskip of 4, more than matching a reference throughput of ∼45k fps at 256 workers, demonstrating that a single-threaded Python controller can efficiently scale to high throughputs.

As shown in Figure 4, by leveraging centralized control, policy optimizers succinctly capture a broad range of choices in RL optimization: synchronous vs asynchronous, allreduce vs parameter server, and use of GPUs vs CPUs. RLlib's policy optimizers provide performance comparable to optimized parameter server (Figure 5(a)) and MPI-based implementations (Section 5). Pulling out this optimizer abstraction is easy in a logically centralized control model since each policy optimizer has full control over the distributed computation it implements.

3.4. Completeness and Generality of Abstractions

We demonstrate the completeness of RLlib's abstractions by formulating the algorithm families listed in Table 2 within the API. When applicable, we also describe the concrete implementation in RLlib:

DQNs: DQNs use y^1 for storing TD error, implement n-step

Table 2. RLlib's policy optimizers and evaluators capture common components (Evaluation, Replay, Gradient-based Optimizer) within a logically centralized control model, and leverages Ray's hierarchical task model to support other distributed components.

Algorithm Family	Policy Evaluation	Replay Buffer	Gradient-Based Optimizer	Other Distributed Components
DQNs	X	X	X	
Policy Gradient	X		X	
Off-policy PG	X	X	X	
Model-Based/Hybrid	X		X	Model-Based Planning
Multi-Agent	X	X	X	
Evolutionary Methods	X			Derivative-Free Optimization
AlphaGo	X	X	X	MCTS, Derivative-Free Optimization

return calculation in ρ_θ, and the Q loss in L. Target updates are implemented in u^1, and setting the exploration ϵ in u^2.

DQN implementation: To support experience replay, RLlib's DQN uses a policy optimizer that saves collected samples in an embedded replay buffer. The user can alternatively use an asynchronous optimizer (Figure 4(c)). The target network is updated by calls to u^1 between optimizer steps.

Ape-X implementation: Ape-X (Horgan et al., 2018) is a variation of DQN that leverages distributed experience prioritization to scale to many hundreds of cores. To adapt our DQN implementation, we created policy evaluators with a distribution of ϵ values, and wrote a new high-throughput policy optimizer ($\sim$200 lines) that pipelines the sampling and transfer of data between replay buffer actors using Ray primitives. Our implementation scales nearly linearly up to 160k environment frames per second with 256 workers (Figure 5(b)), and the optimizer can compute gradients for $\sim$8.5k $80\times80\times4$ observations/s on a V100 GPU.

Policy Gradient / Off-policy PG: These algorithms store value predictions in y^1, implement advantage estimation using ρ_θ, and combine actor and critic losses in L.

PPO implementation: Since PPO's loss function permits multiple SGD passes over sample data, when there is sufficient GPU memory RLlib chooses a GPU-optimized policy optimizer (Figure 4(b)) that pins data into local GPU memory. In each iteration, the optimizer collects samples from evaluator replicas, performs multi-GPU optimization locally, and then broadcasts the new model weights.

A3C implementation: RLlib's A3C can use either the asynchronous (Figure 4(c)) or sharded parameter server policy optimizer (4(d)). These optimizers collect gradients from the policy evaluators to update the canonical copy of θ.

DDPG implementation: RLlib's DDPG uses the same replay policy optimizer as DQN. L includes both actor and critic losses. The user can also choose to use the Ape-X policy optimizer with DDPG.

Model-based / Hybrid: Model-based RL algorithms extend $\pi_\theta(o_t, h_t)$ to make decisions based on model rollouts, which can be parallelized using Ray. To update their environment models, the model loss can either be bundled with L, or the model trained separately (i.e., in parallel using Ray primitives) and its weights periodically updated via u^1.

Multi-Agent: Policy evaluators can run multiple policies at once in the same environment, producing batches of experience for each agent. Many multi-agent algorithms use a centralized critic or value function, which we support by allowing ρ_θ to collate experiences from multiple agents.

Evolutionary Methods: Derivative-free methods can be supported through non-gradient-based policy optimizers.

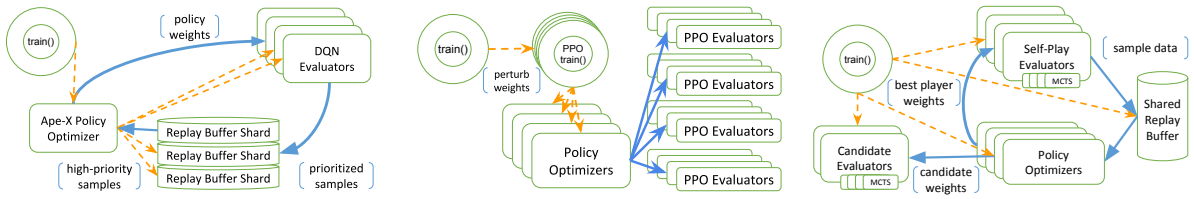
Evolution Strategies (ES) implementation: ES is a derivative-free optimization algorithm that scales well to clusters with thousands of CPUs. We were able to port a single-threaded implementation of ES to RLlib with only a few changes, and further scale it with an aggregation tree of actors (Figure 8(a)), suggesting that the hierarchical control model is both flexible and easy to adapt algorithms for.

PPO-ES experiment: We studied a hybrid algorithm that runs PPO updates in the inner loop of an ES optimization step that randomly perturbs the PPO models. The implementation took only $\sim$50 lines of code and did not require changes to PPO, showing the value of encapsulating parallelism. In our experiments, PPO-ES converged faster and to a higher reward than PPO on the Walker2d-v1 task. A similarly modified A3C-ES implementation solved PongDeterministic-v4 in 30% less time.

AlphaGo: We sketch how to scalably implement the AlphaGo Zero algorithm using a combination of Ray and RLlib abstractions. Pseudocode for the $\sim$70 line main algorithm loop is provided in the Supplementary Material.

1. *Logically centralized control of multiple distributed components*: AlphaGo Zero uses multiple distributed components: model optimizers, self-play evaluators, candidate model evaluators, and the shared replay buffer. These components are manageable as Ray actors under a top-level AlphaGo policy optimizer. Each optimizer step loops over actor statuses to process new results, routing data between actors and launching new actor instances.

2. *Shared replay buffer*: AlphaGo Zero stores the experi-



(a) Ape-X (b) PPO-ES (c) AlphaGo Zero

Figure 6. Complex RL architectures are easily captured within RLlib's hierarchical control model. Here blue lines denote data transfers, orange lines lighter overhead method calls. Each train() call encompasses a batch of remote calls between components.

ences from self-play evaluator instances in a shared replay buffer. This requires routing game results to the shared buffer, which is easily done by passing the result object references from actor to actor.

3. *Best player*: AlphaGo Zero tracks the current best model and only populates its replay buffer with self-play from that model. Candidate models must achieve a $\geq 55\%$ victory margin to replace the best model. Implementing this amounts to adding an `if` block in the main control loop.

4. *Monte-Carlo tree search*: MCTS (i.e., model-based planning) can be handled as a subroutine of the policy graph, and optionally parallelized as well using Ray.

HyperBand and Population Based Training: Ray includes distributed implementations of hyperparameter search algorithms such as HyperBand and PBT (Li et al., 2016; Jaderberg et al., 2017). We were able to use these to evaluate RLlib algorithms, which are themselves distributed, with the addition of ∼15 lines of code per algorithm. We note that these algorithms are non-trivial to integrate when using distributed control models due to the need to modify existing code to insert points of coordination (Figure 3). RLlib's use of short-running tasks avoids this problem, since control decisions can be easily made between tasks.

4. Framework Performance

In this section, we discuss properties of Ray (Moritz et al., 2017) and other optimizations critical to RLlib.

4.1. Single-node performance

Stateful computation: Tasks can share mutable state with other tasks through Ray actors. This is critical for tasks that operate on and mutate stateful objects like third-party simulators or neural network weights.

Shared memory object store: RL workloads involve sharing large quantities of data (e.g., rollouts and neural network weights). Ray supports this by allowing data objects to be passed directly between workers without any central bottleneck. In Ray, workers on the same machine can also read data objects through shared memory without copies.

Vectorization: RLlib can batch policy evaluation to improve hardware utilization (Figure 7), supports batched environments, and passes experience data between actors efficiently in columnar array format.

4.2. Distributed performance

Lightweight tasks: Remote call overheads in Ray are on the order of ∼200µs when scheduled on the same machine. When machine resources are saturated, tasks spill over to other nodes, increasing latencies to around ∼1ms. This enables parallel algorithms to scale seamlessly to multiple machines while preserving high single-node throughput.

Nested parallelism: Building RL algorithms by composing distributed components creates multiple levels of nested parallel calls (Figure 1). Since components make decisions that may affect downstream calls, the call graph is also inherently dynamic. Ray supports this by allowing any Python function or class method to be invoked remotely as a lightweight task. For example, `func.remote()` executes `func` remotely and immediately returns a placeholder result which can later be retrieved or passed to other tasks.

Resource awareness: Ray allows remote calls to specify resource requirements and utilizes a resource-aware scheduler to preserve component performance. Without this, distributed components can improperly allocate resources, causing algorithms to run inefficiently or fail.

Fault tolerance and straggler mitigation: Failure events become significant at scale (Barroso et al., 2013). RLlib leverages Ray's built-in fault tolerance mechanisms (Moritz et al., 2017), reducing costs with preemptible cloud compute instances (Amazon, 2011; Google, 2015). Similarly, stragglers can significantly impact the performance of distributed algorithms at scale (Dean & Barroso, 2013). RLlib supports straggler mitigation in a generic way via the `ray.wait()` primitive. For example, in PPO we use this to drop the slowest evaluator tasks, at the cost of some bias.

Data compression: RLlib uses the LZ4 algorithm to compress experience batches. For image observations, LZ4 reduces network traffic and memory usage by more than an order of magnitude, at a compression rate of ∼1 GB/s/core.

5. Evaluation

Sampling efficiency: Policy evaluation is an important building block for all RL algorithms. In Figure 7 we benchmark the scalability of gathering samples from policy evaluator actors. To avoid bottlenecks, we use four intermediate actors for aggregation. Pendulum-CPU reaches over 1.5 million actions/s running a small 64×64 fully connected network as the policy. Pong-GPU nears 200k actions/s on the DQN convolutional architecture (Mnih et al., 2015).

Large-scale tests: We evaluate the performance of RLlib on Evolution Strategies (ES), Proximal Policy Optimization (PPO), and A3C, comparing against specialized systems built *specifically for those algorithms* (OpenAI, 2017; Hesse et al., 2017; OpenAI, 2016) using Redis, OpenMPI, and Distributed TensorFlow. The same hyperparameters were used in all experiments. We used TensorFlow to define neural networks for the RLlib algorithms evaluated.

RLlib's ES implementation scales well on the Humanoid-v1 task to 8192 cores using AWS m4.16xl CPU instances (Amazon, 2017). With 8192 cores, we achieve a reward of 6000 in a median time of 3.7 minutes, which is over twice as fast as the best published result (Salimans et al., 2017). For PPO we evaluate on the same Humanoid-v1 task, starting with one p2.16xl GPU instance and adding m4.16xl instances to scale. This cost-efficient local policy optimizer (Table 3) outperformed the reference MPI implementation that required multiple expensive GPU instances to scale.

We ran RLlib's A3C on an x1.16xl machine and solved the PongDeterministic-v4 environment in 12 minutes using an asynchronous policy optimizer and 9 minutes using a sharded param-server optimizer, which matches the performance of a well-tuned baseline (OpenAI, 2016).

Multi-GPU: To better understand RLlib's advantage in the PPO experiment, we ran benchmarks on a p2.16xl instance comparing RLlib's local multi-GPU policy optimizer with one using an allreduce in Table 3. The fact that different strategies perform better under different conditions suggests that policy optimizers are a useful abstraction.

Policy Optimizer	Gradients computed on	Environment	SGD throughput
Allreduce-based	4 GPUs, Evaluators	Humanoid-v1	330k samples/s
		Pong-v0	23k samples/s
	16 GPUs, Evaluators	Humanoid-v1	**440k samples/s**
		Pong-v0	**100k samples/s**
Local Multi-GPU	4 GPUs, Driver	Humanoid-v1	**2.1M samples/s**
		Pong-v0	N/A (out of mem.)
	16 GPUs, Driver	Humanoid-v1	1.7M samples/s
		Pong-v0	**150k samples/s**

Table 3. A specialized multi-GPU policy optimizer outperforms distributed allreduce when data can fit entirely into GPU memory. This experiment was done for PPO with 64 Evaluator processes. The PPO batch size was 320k, The SGD batch size was 32k, and we used 20 SGD passes per PPO batch.

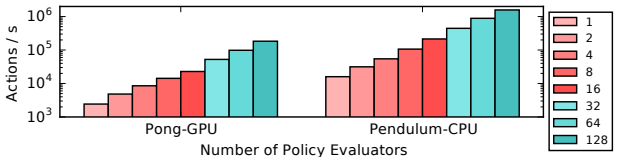
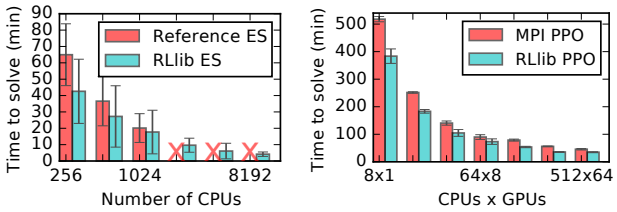


Figure 7. Policy evaluation throughput scales nearly linearly from 1 to 128 cores. PongNoFrameskip-v4 on GPU scales from 2.4k to ∼200k actions/s, and Pendulum-v0 on CPU from 15k to 1.5M actions/s. We use a single p3.16xl AWS instance to evaluate from 1-16 cores, and a cluster of four p3.16xl instances from 32-128 cores, spreading actors evenly across the cluster. Evaluators compute actions for 64 agents at a time, and share the GPUs on the machine.



(a) Evolution Strategies (b) PPO

Figure 8. The time required to achieve a reward of 6000 on the Humanoid-v1 task. RLlib implementations of ES and PPO outperform highly optimized reference optimizations.

6. Related work

There are many reinforcement learning libraries (Caspi, 2017; Duan et al., 2016; Hafner et al., 2017; Hesse et al., 2017; Kostrikov, 2017; Schaarschmidt et al., 2017). These often scale by creating long-running program replicas that each participate in coordinating the distributed computation as a whole, and as a result do not generalize well to complex architectures. RLlib instead uses a hierarchical control model with short-running tasks to let each component control its own distributed execution, enabling higher-level abstractions such as policy optimizers to be used for composing and scaling RL algorithms.

Outside of reinforcement learning, there has been a strong effort to explore composition and integration between different deep learning frameworks. ONNX (Microsoft, 2017), NNVM (DMLC, 2017), and Gluon (Gluon, 2017) sit between model specifications and hardware to provide cross-library optimizations. Deep learning libraries (Paszke et al., 2017; Abadi et al., 2016; Chen et al., 2016; Jia et al., 2014) provide support for the gradient-based optimization components that appear in RL algorithms.

7. Conclusion

RLlib is an open source library for reinforcement learning that leverages fine-grained nested parallelism to achieve state-of-the-art performance across a broad range of RL workloads. It offers both a collection of reference algorithms and scalable abstractions for easily composing new ones.

Acknowledgements

In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported in part by DHS Award HSHQDC-16-3-00083, and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Amazon. Scientific computing with EC2 spot instances. https://aws.amazon.com/ec2/spot/spot-and-science/, 2011.

Amazon. Amazon EC2 pricing. https://aws.amazon.com/ec2/pricing, 2017.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

Barroso, L. A., Clidaras, J., and Hölzle, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI gym, 2016.

Caspi, I. Reinforcement learning coach by Intel. https://github.com/NervanaSystems/coach, 2017.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

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.

Dean, J. and Barroso, L. A. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

DMLC. NNVM compiler: Open compiler for AI frameworks. http://www.tvmlang.org/2017/10/06/nnvm-compiler-announcement.html, 2017.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pp. 1329–1338, 2016.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

Gluon. The Gluon API specification. https://github.com/gluon-api/gluon-api, 2017.

Google. Preemptible virtual machines. https://cloud.google.com/preemptible-vms, 2015.

Graefe, G. and Davison, D. L. Encapsulation of parallelism and architecture-independence in extensible database query execution. *IEEE Transactions on Software Engineering*, 19(8):749–764, 1993.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22 (6):789–828, 1996.

Hafner, D., Davidson, J., and Vanhoucke, V. TensorFlow agents: Efficient batched reinforcement learning in TensorFlow. *arXiv preprint arXiv:1709.02878*, 2017.

Hesse, C., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. OpenAI baselines. https://github.com/openai/baselines, 2017.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. Distributed prioritized experience replay. *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1Dy---0Z.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Kostrikov, I. PyTorch implementation of advantage actor critic (A2C), proximal policy optimization (PPO)

and scalable trust-region method for deep reinforcement learning. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr, 2017.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. 2016.

Li, M., Andersen, D. G., Park, J. W., Smola, A., and Ahmed, A. Scaling distributed machine learning with the parameter server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, volume 583, pp. 598, 2014.

Microsoft. ONNX: Open neural network exchange format. https://onnx.ai, 2017.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. *arXiv preprint arXiv:1712.05889*, 2017.

OpenAI. Universe Starter Agent. https://github.com/openai/universe-starter-agent, 2016.

OpenAI. Evolution Strategies Starter Agent. https://github.com/openai/evolution-strategies-starter, 2017.

Pan, H., Hindman, B., and Asanović, K. Composing parallel software efficiently with Lithe. *ACM Sigplan Notices*, 45 (6):376–387, 2010.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Salimans, T., Ho, J., Chen, X., and Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Schaarschmidt, M., Kuhnle, A., and Fricke, K. TensorForce: A TensorFlow library for applied reinforcement learning. Web page, 2017. URL https://github.com/reinforceio/tensorforce.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of Go without human knowledge. 2017.

Tian, Y., Gong, Q., Shang, W., Wu, Y., and Zitnick, L. ELF: an extensive, lightweight and flexible research platform for real-time strategy games. *CoRR*, abs/1707.01067, 2017. URL http://arxiv.org/abs/1707.01067.

Zaharia, M., Chowdhury, N. M., Franklin, M., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. 2010.