Ray RLlib: A Composable and Scalable Reinforcement Learning Library

Eric Liang*, **Richard Liaw***, **Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, Ion Stoica**

University of California, Berkeley

Abstract

Reinforcement learning (RL) algorithms involve the deep nesting of distinct components, where each component typically exhibits opportunities for distributed computation. Current RL libraries offer parallelism at the level of the entire program, coupling all the components together and making existing implementations difficult to extend, combine, and reuse. We argue for building composable RL components by encapsulating parallelism and resource requirements within individual components, which can be achieved by building on top of a flexible task-based programming model. We demonstrate this principle by building Ray RLlib on top of Ray [40] and show that we can implement a wide range of state-of-the-art algorithms by composing and reusing a handful of standard components. This composability does not come at the cost of performance — in our experiments, RLlib matches or exceeds the performance of highly optimized reference implementations. Ray RLlib is available as part of Ray at https://github.com/ray-project/ray/.

1 Introduction

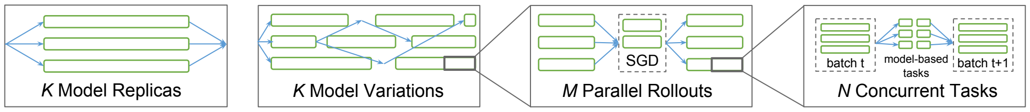
Advances in parallel computing and composition through symbolic differentiation were fundamental to the recent success of deep learning. Today, there are a wide range of deep learning frameworks [1, 11, 17, 31] that enable rapid innovation in neural network design and facilitate training at the scale 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 for reinforcement learning. Nonetheless, many of the challenges in reinforcement learning stem from the need to scale learning and simulation while also integrating a rapidly evolving space 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, use a neural network defined by a deep learning framework, recursively issue calls to other components, 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 different physical devices.

*Equal contribution

The substantial recent progress in RL algorithms and applications has resulted in a large and growing number of RL libraries [16, 19, 26, 28, 32, 48]. 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 [24], Distributed TensorFlow [11], and parameter servers [34]). As this programming model ignores component boundaries, it does not naturally encapsulate parallelism and resource requirements within individual components.[1] 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). As a result of this lack of encapsulation, most reinforcement learning algorithms are implemented from scratch. This includes the recreation not only of the RL components but also of the underlying distributed communication and execution logic.



(a) Deep Learning (b) Reinforcement Learning

Figure 1: In contrast with deep learning, reinforcement learning 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 1).

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.[2] Toward this end, we argue for structuring distributed RL components around the principle of parallelism encapsulation [23, 44]. This would enable one parallel or distributed component to call another, possibly in parallel, *without any modifications to the called component* (see Figure 2 for an example).

In contrast with existing distributed frameworks for implementing large-scale RL algorithms, we argue that a **flexible task-based programming model** is much more natural, as it permits this type of encapsulation. We demonstrate this by building RLlib and providing a set of reusable components that can be used to build a wide variety of the RL algorithms that have been proposed. We also highlight the benefits of encapsulation by providing a pluggable SGD module that allows different SGD strategies (allreduce, local synchronous, and asynchronous) to be swapped in independently of the rest of the algorithm.

1.1 The Importance of Composition

A reinforcement learning library with composable components would enable the rapid development and prototyping of new algorithms using high-performance components, similar to the way in which deep learning frameworks enable the rapid development of new neural network architectures. Many RL algorithms already require the use of many components. For example, a component that implements policy evaluation by running parallel rollouts over many CPUs could be called by a component that implements data-parallel distributed stochastic gradient descent on multiple GPUs. This entire setup could be run inside of a component that runs hyperparameter search as an outer loop. RL algorithms exhibit this pattern at multiple levels as shown in Figure 1.

As natural as this is, the actual implementation of such a scheme is typically done from scratch without the reuse of existing distributed components, and the end product is typically tightly coupled to a one-off message passing scheme crafted for the algorithm at hand.

A library exhibiting parallelism encapsulation would allow the nesting of components without modifying the called components. For example, the hyperparameter search component would invoke multiple copies of the gradient-based optimizer without touching the optimizer's internals.

As another example, a caller of the gradient-based optimizer may wish to obtain its status, pause the optimizer, checkpoint its state, or adjust its parameters. Traditional message passing frameworks like

[1]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.

[2]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.

MPI require explicit algorithm modification to insert these points of coordination. In contrast, with components that encapsulate parallelism and resource requirements, the caller could make these decisions without modification to the downstream components. See Figure 2 for an example. Ultimately, these design principles make it easier to reuse, extend, and combine existing RL components.

```
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) MPI

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

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

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

(b) Flexible task-based model

Figure 2: 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 the hyperparameter search and the function evaluation. With a flexible task-based model (b), components can remain unchanged and simply be invoked as tasks.

Beyond reusability, ease of development is critical for the fast design and evaluation of novel distributed RL applications such as AlphaGo [53, 54] and Neural Architecture Search [15, 62], currently implemented as specialized distributed systems. We demonstrate this ourselves in RLlib by implementing a novel hybrid of Proximal Policy Optimization [52] and Evolution Strategies [47] with only a few lines of code (Section 2.3.2), which was only possible due to RLlib's composition-oriented design.

1.2 Survey of Components

Modern RL algorithms span a large and complex design space. However, many components of these algorithms appear in many different algorithms. To orient the reader, we include a list of some of the more familiar RL algorithm components in Table 1.

Table 1: RLlib provides scalable abstractions that capture commonly used components (Evaluation, Replay, Gradient-based Optimizer), and leverages Ray's system primitives to provide support for other distributed components. We provide an extended discussion of these families in Appendix A.

Algorithm Family	Policy Evaluation	Replay Buffer	Gradient Based Optimization	Other Distributed Components
Policy Gradient [61, 50, 52, 55, 37]	✓		✓	
DQNs [38, 49, 57, 59, 14, 29, 41]	✓	✓	✓	
Off-policy PG [35, 25, 58, 30]	✓	✓	✓	
Model-Based/Hybrid [60, 45, 42]	✓		✓	Model-Based Planning
Multi-Agent [36, 20, 39, 46, 21]	✓	✓	✓	
Evolutionary Methods [47, 27]	✓			Derivative-Free Optimization
AlphaGo [53, 54]	✓	✓	✓	MCTS, Derivative-Free Optimization

2 Proposal: A Task-Based Design Principle

As highlighted in Section 1, parallelization of entire programs using frameworks like MPI [24] and Distributed Tensorflow [11] typically require explicit algorithm modifications to insert points of coordination when trying to compose two programs or components together (Figure 2). This limits the ability to rapidly prototype novel distributed RL applications.

We propose building RL libraries on top of *flexible task-based programming models* like Ray [40]. Task-based systems allow functions to be scheduled and executed asynchronously on remote machines and for results to be retrieved at a later time. In Section 3, we elaborate on the properties that make Ray a highly-performant fit for building composable RL components.

2.1 Ray Primitives

There are a few primitives required for such a task-based programming model. First, to allow tasks launched by the program to in turn launch more tasks, the framework must provide a flexible remote call API. Second, to provide the best performance, it is necessary to support stateful remote classes. Finally, the framework must support efficient communication between remote tasks and classes.

Remote Functions and Classes: Ray meets the first requirements by providing a remote call abstraction for Python functions and classes. Any Python function may be run as a remote task, and any Python class may be constructed remotely (this makes all class method calls remote tasks).

Efficient Communication: Ray provides standard communication primitives such as *aggregate* and *broadcast*, and critically enables the *zero-copy* sharing of large data objects through shared memory.

2.2 RLlib APIs

Ray RLlib provides scalable abstractions for performance-critical tasks such as distributed sample collection and SGD. Following the platform-application design pattern [56], RLlib's APIs are general and work with multiple deep learning frameworks, including TensorFlow and PyTorch.

2.2.1 Encapsulating Parallelism in Training

The composability advantages of the task-based model come from the relatively short duration of tasks compared to the overall program. To understand this, consider a training task that runs continuously until completion — it would present no opportunity for the caller of the task to adjust training except to kill the task. Each RLlib algorithm defines a **train**$(alg) \rightarrow result$ method that runs one logical iteration of training. Between iterations, control decisions may be made, e.g. whether to continue training, checkpoint, adjust parameters, or stop. Iterations typically make calls to multiple remote classes (i.e. Python objects created on other machines by Ray, also known as Ray *actors*) that retain state between calls for efficiency.

2.2.2 Pluggable SGD within the Task Model

RLlib includes an RL optimizer module that manages the common and performance-critical tasks of distributed sampling and SGD. This takes the form of a number of built-in SGD strategies that include the choice between synchronous and asynchronous execution (Figure 3). The user can specify a SGD strategy, or the RLlib optimizer can choose one automatically.

For gradient-based algorithms to leverage these optimizer strategies, RLlib requires those algorithms to implement an **Evaluator** abstraction. Evaluators are classes that define the following methods, where X is a vector of experiences, θ is the model parameters, and ev is the evaluator:

$$\textbf{sample}(ev) \rightarrow X; \;\; \textbf{grad}(ev, X) \rightarrow \nabla_\theta; \;\; \textbf{weights}(ev) \rightarrow \theta; \;\; \textbf{set\_weights}(ev, \theta) \rightarrow ev;$$

Any object that implements these interfaces is an Evaluator. A typical SGD programming API implements $step(L(\theta), X, \theta) \rightarrow \theta_{opt}$. RLlib's optimizers take as input a local Evaluator and array of remote Evaluators instead, i.e. $step(E_{local}, E_{1..n}, \theta) \rightarrow \theta_{opt}$.

To make the role of Evaluators concrete, consider the following example implementation of **train** for the PPO algorithm [52], also shown graphically in Figure 4(a). Algorithm initialization creates an appropriate RL optimizer (Figure 3) and many PPO-specific Evaluators for parallelism, each of which includes a replica of the PPO policy model and loss. Each train call runs a number of optimizer steps:

```
class ExamplePPO(Algorithm):
    def __init__(self):
        # setup optimizer, PPO evaluators
    def train(self):
        for i in range(10):
            self.optimizer.step(self.local_evaluator, self.evaluators)
        # return statistics
```

The Evaluator abstraction has the following advantages. Firstly, the optimizer can make decisions based on locality information from Evaluators, as described in Appendix C. Secondly, this API encapsulates the choice of SGD strategy, enabling authors to e.g. choose between synchronous and asynchronous implementations as outlined in Figure 3, without needing to modify algorithm code. Finally, the Evaluator interface entirely encapsulates the model graph and loss functions, allowing SGD implementations to be improved and reused across different deep learning frameworks.

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

(a) Allreduce SGD

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

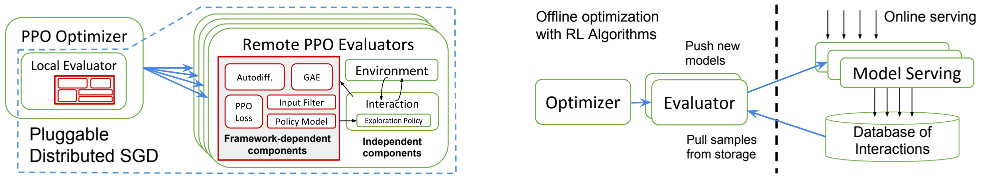
(b) Local Multi-GPU SGD

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

(c) Asynchronous SGD

Figure 3: Pseudocode for three RLlib optimizer strategies. Each takes as input an array of remote Evaluators and a local Evaluator. Ray remote calls are highlighted in orange; other Ray primitives in blue (Section 3). *Apply* is shorthand for updating weights. Minibatch code is omitted.

All of RLlib's gradient-based algorithms use this pluggable SGD interface. In Section 4, we show that Ray's SGD strategies provide performance comparable to highly optimized parameter server and MPI-based [24] implementations. Pulling out a distributed SGD abstraction is easy in a task-based model since the SGD component's parallelism can be entirely encapsulated.



(a) Evaluators in RLlib's PPO (b) Using Evaluators with Offline Data

Figure 4: Using the Evaluator API. Evaluators encapsulate modules defined by deep learning frameworks, for example, the model, loss function, and autodifferentiation, shown within the red PPO Evaluator block, as well as interaction with the environment. Evaluators can be used both for online training and for interfacing with offline replay data (Appendix E.1).

2.3 API Completeness and Generality

2.3.1 Composing Popular Algorithms with RLlib

We demonstrate the composability of the proposed components by implementing the following algorithms as part of RLlib:

Proximal Policy Optimization (PPO): RLlib's PPO implementation (Figure 4(a)) starts by creating a number of remote Evaluator processes in the cluster, which encapsulate the PPO policy and loss. Since PPO's loss function permits multiple SGD passes over sample data, when there is sufficient GPU memory available RLlib chooses a faster SGD strategy (Table 2) that pins data into local GPU memory. In each iteration, the optimizer collects samples from Evaluators, performs optimization locally, and then broadcasts the new model weights.

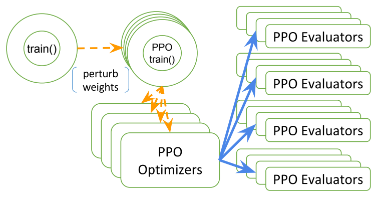
A3C: RLlib's A3C implementation uses the asynchronous SGD strategy (Figure 3). The strategy requests gradients from A3C Evaluators, sends weights in each request, and continuously updates the local model. This resembles a parameter server model with an inversion of control that encapsulates the worker parallelism. To minimize request-to-gradient latency, A3C Evaluators queue sample batches in a separate internal thread.

DQN: Since DQN is much more sample efficient that PPO, it is typically run with only a single Evaluator, wrapped in a replay buffer. In this case, RLlib chooses an embedded same-process SGD optimizer to minimize data transfer overheads. The DQN Evaluator implements an additional DQN-specific method to set the target network weights.

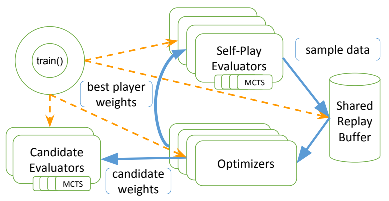
Evolution Strategies (ES): ES is a derivative-free optimization algorithm that scales well to clusters with thousands of CPUs. We were able to port a reference single-threaded implementation of ES to RLlib with only a few changes, suggesting that Ray's primitives (Section 3) generalize well, and that the flexible task-based programming model imposes minimal burden on developers.

2.3.2 Generalization to Complex Architectures

PPO-ES: We tried implementing a new RL algorithm that runs PPO updates in the inner loop of an ES optimization step that randomly perturbs the PPO models. Within an hour, we were able to deploy to a small cluster for evaluation. The implementation took only $\sim$50 lines of code and did not require modifying the PPO implementation, showing the value of encapsulation. In our experiments (Figure 7 in Appendix E.2), PPO-ES outperformed base PPO, converging faster and to a higher reward on the Walker2d-v1 task. A similarly modified A3C-ES implementation solved PongDeterministic-v4 in 30% less time than the baseline.



(a) PPO-ES



(b) AlphaGo Zero

Figure 5: RLlib diagrams for PPO-ES and AlphaGo Zero. Blue lines denote data transfers, orange lines lighter overhead method calls. Each train() call encompasses a batch of calls between components.

AlphaGo Zero: We show how AlphaGo could be implemented as a top-level RLlib algorithm. RLlib pseudocode for the $\sim$70 line main algorithm loop is provided in the Appendix E.3. Though our design is bare-bones, it is functionally complete and can leverage Ray's automatic checkpointing for fault-tolerance (Section 3). We discuss notable differences from prior RL algorithms:

1. **Top-down control of multiple distributed components**: AlphaGo Zero uses multiple distributed components that run concurrently: model optimizers, self-play evaluators, candidate model evaluators, and the shared replay buffer. In our RLlib implementation, these are managed as stateful components under the training task. Each task loops over component statuses to process new results, routing data between components and launching new component replicas as needed.
2. **Shared replay buffer**: Unlike DQN, which wraps a single Evaluator to add replay, AlphaGo Zero shares a replay buffer among many self-play Evaluator instances. This required routing game results to the shared buffer, which took $\sim$7 additional lines of code in the train function.
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. Supporting this required $\sim$10 lines of code.
4. **Monte-carlo tree search**: MCTS is added as a sub-routine of each self-play Evaluator. Since Ray supports parallelism encapsulation, discussed in Section 3, MCTS can also easily be distributed.

Hyperparameter Search Integration: Ray also provides a distributed hyperparameter search library independent of RLlib, which efficiently evaluates trials with early stopping algorithms [33, 22]. We were able to use this to evaluate RLlib algorithms, which are themselves distributed, with the addition of $\sim$5 lines of code per algorithm to return RL-specific training results (Appendix E.4). This suggests that Ray's primitives are well suited for interoperability between distributed components.

3 The Programming and Execution Model

In this section, we discuss some of the properties of Ray [40] that enable the implementation of scalable and composable RL algorithms in RLlib.

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 with *parallelism encapsulation*, which allows 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.

Lightweight tasks: Remote call overheads in Ray are on the order of hundreds of microseconds[3] when scheduled on the same machine. When machine resources are saturated, tasks spill over to other nodes in the Ray cluster, increasing latencies to around one millisecond. This enables parallel algorithms to scale seamlessly to multiple machines while preserving high single-node throughput.

Resource Awareness: RL algorithms have components with varying resource requirements, utilizing both GPUs and CPUs. Without *resource requirement encapsulation* in Ray, distributed components can improperly allocate resources especially during nested parallel calls which can cause algorithms to fail. Ray allows remote calls to specify resource requirements and utilizes a resource-aware scheduler to preserve component performance.

Stateful computation: Tasks can share mutable state with other tasks. This is critical for tasks that operate on and mutate stateful objects like third-party simulators or neural network weights. Tasks operating on the same mutable state are encapsulated as method invocations of a remote class (i.e. Ray actor). A class `cls` can be instantiated remotely via `c = cls.remote()`, and tasks operating on the remote class can be invoked via `c.method.remote()`. All current RLlib algorithms use remote classes due to the greater performance.

High-performance data sharing: RL workloads involve sharing large quantities of data (e.g., rollouts and neural network weights) between workers. Ray supports this by allowing data objects to be passed directly between workers without passing through any central bottleneck. Worker processes on the same machine can also *read data objects through shared memory without copies*. This enables the performance of RLlib algorithms (Section 4).

Straggler mitigation: Stragglers can significantly impact the performance of distributed algorithms at scale [18]. RLlib supports straggler mitigation in a generic way via the `ray.wait` primitive. For example, in PPO we use this to drop the slowest few Evaluator tasks, at the cost of some bias.

Fault tolerance: Similarly, failure events become significant at scale [13]. RLlib enables application-level checkpointing between tasks and also leverages Ray's built-in fault tolerance mechanisms [40], allowing the use of cheaper preemptible cloud compute instances [2, 3].

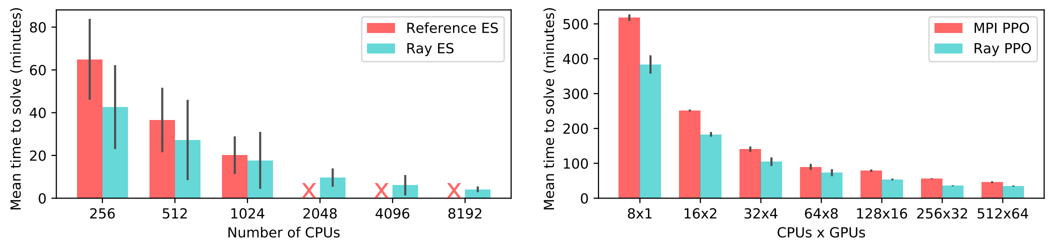
4 Evaluation

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* [7, 28, 4] using Redis, OpenMPI, and Distributed TensorFlow. The same hyperparameters were used in all experiments (Appendix D). 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 [5]. 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 [47]. 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 SGD strategy (Table 2) outperformed the reference MPI implementation that required multiple expensive GPU instances to scale.

We also ran RLlib's A3C on a 16-CPU machine and solved the PongDeterministic-v4 environment in 12 minutes (not shown), which is within 20% of a well-tuned baseline [4]. This slight performance

[3]This includes submitting the task, scheduling it, executing it, storing the result, and retrieving the result.



(a) Evolution Strategies (b) PPO

Figure 6: 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.

gap is due to an extra data conversion incurred by RLlib's async SGD compared to Distributed TensorFlow, which we are working to remove.

Finally, to better understand RLlib's advantage over the baseline in the PPO experiment, we ran microbenchmarks on a p2.16xl instance comparing RLlib's local multi-GPU SGD strategy with one using an allreduce in Table 2. The fact that different SGD strategies perform better under different conditions demonstrates the value of a pluggable abstraction in which different implementations can be swapped in. An extended discussion of pluggable SGD can be found in Appendix C.

Table 2: A specialized multi-GPU SGD strategy 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.

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

5 Related work

There are many distributed reinforcement learning libraries [16, 19, 26, 28, 32, 48]. These often distribute computation by coupling the RL algorithm logic to an architecture relying on communication between long-running program replicas. As discussed above, this coupling inherently prevents components from encapsulating their internal parallelism and resource requirements. RLlib proposes using a task-based programming model to let each component control its own resources and degree of parallelism, enabling the easy composition and reuse of components.

Outside of reinforcement learning, there has been a strong effort to explore composition and integration between different deep learning frameworks. ONNX [10], NNVM [9], and Gluon [8] sit between model specifications and hardware to provide cross-library optimizations. Deep learning libraries [1, 11, 17, 31] themselves are a complementary effort, as they provide thorough support for the gradient-based optimization components that appear in RL algorithms.

More broadly, recent efforts to tackle cross-library integration in the data science community include Weld [43], which provides an intermediate representation and compiler optimizations for popular data science libraries, and Apache Arrow [6], which provides a common memory layout for fast serialization and data transfer between platforms.

References

[1] PyTorch: Tensors and dynamic neural networks in python with strong gpu acceleration. `http://pytorch.org`.

[2] Scientific computing with EC2 spot instances. `https://aws.amazon.com/blogs/aws/scientific-computing-with-ec2-spot-instances`, 2011.

[3] Preemptible virtual machines. `https://cloud.google.com/preemptible-vms`, 2015.

[4] Universe starter agent. `https://github.com/openai/universe-starter-agent`, 2016.

[5] Amazon EC2 pricing. `https://aws.amazon.com/ec2/pricing`, 2017.

[6] Apache Arrow: Powering columnar in-memory analytics. `https://arrow.apache.org`, 2017.

[7] Evolution strategies starter agent. `https://github.com/openai/evolution-strategies-starter`, 2017.

[8] The Gluon API specification. `https://github.com/gluon-api/gluon-api`, 2017.

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

[10] ONNX: Open neural network exchange format. `https://onnx.ai`, 2017.

[11] 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).

[12] Abadi, M., Isard, M., and Murray, D. G. A computational model for TensorFlow: an introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (2017), ACM, pp. 1–7.

[13] 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 (2013), 1–154.

[14] Bellemare, M. G., Dabney, W., and Munos, R. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887* (2017).

[15] Bello, I., Zoph, B., Vasudevan, V., and Le, Q. Neural optimizer search with reinforcement learning.

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

[17] 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).

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

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

[20] Foerster, J., Assael, Y. M., de Freitas, N., and Whiteson, S. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems* (2016), pp. 2137–2145.

[21] Foerster, J., Nardelli, N., Farquhar, G., Torr, P., Kohli, P., Whiteson, S., et al. Stabilising experience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887* (2017).

[22] Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. Google Vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), ACM, pp. 1487–1495.

[23] GRAEFE, G., AND DAVISON, D. L. Encapsulation of parallelism and architecture-independence in extensible database query execution. *IEEE Transactions on Software Engineering 19*, 8 (1993), 749–764.

[24] 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 (1996), 789–828.

[25] GU, S., LILLICRAP, T., GHAHRAMANI, Z., TURNER, R. E., AND LEVINE, S. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv preprint arXiv:1611.02247* (2016).

[26] HAFNER, D., DAVIDSON, J., AND VANHOUCKE, V. TensorFlow agents: Efficient batched reinforcement learning in TensorFlow. *arXiv preprint arXiv:1709.02878* (2017).

[27] HANSEN, N., AND OSTERMEIER, A. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on* (1996), IEEE, pp. 312–317.

[28] HESSE, C., PLAPPERT, M., RADFORD, A., SCHULMAN, J., SIDOR, S., AND WU, Y. OpenAI baselines. `https://github.com/openai/baselines`, 2017.

[29] HESSEL, M., MODAYIL, J., VAN HASSELT, H., SCHAUL, T., OSTROVSKI, G., DABNEY, W., HORGAN, D., PIOT, B., AZAR, M., AND SILVER, D. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298* (2017).

[30] JADERBERG, M., MNIH, V., CZARNECKI, W. M., SCHAUL, T., LEIBO, J. Z., SILVER, D., AND KAVUKCUOGLU, K. Reinforcement learning with unsupervised auxiliary tasks. *International Conference on Learning Representations (ICLR)* (2017).

[31] 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).

[32] 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.

[33] LI, L., JAMIESON, K., DESALVO, G., ROSTAMIZADEH, A., AND TALWALKAR, A. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization.

[34] LI, M., ANDERSEN, D. G., AND PARK, J. W. Scaling distributed machine learning with the parameter server.

[35] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[36] LOWE, R., WU, Y., TAMAR, A., HARB, J., ABBEEL, P., AND MORDATCH, I. Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv preprint arXiv:1706.02275* (2017).

[37] 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* (2016), pp. 1928–1937.

[38] 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 (2015), 529–533.

[39] MORDATCH, I., AND ABBEEL, P. Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908* (2017).

[40] 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).

[41] NAIR, A., SRINIVASAN, P., BLACKWELL, S., ALCICEK, C., FEARON, R., DE MARIA, A., PANNEERSHELVAM, V., SULEYMAN, M., BEATTIE, C., PETERSEN, S., ET AL. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296* (2015).

[42] Oh, J., Singh, S., and Lee, H. Value prediction network. *arXiv preprint arXiv:1707.03497* (2017).

[43] Palkar, S., Thomas, J. J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S., Zaharia, M., and InfoLab, S. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)* (2017).

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

[45] Pascanu, R., Li, Y., Vinyals, O., Heess, N., Buesing, L., Racanière, S., Reichert, D., Weber, T., Wierstra, D., and Battaglia, P. Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170* (2017).

[46] Peng, P., Yuan, Q., Wen, Y., Yang, Y., Tang, Z., Long, H., and Wang, J. Multiagent bidirectionally-coordinated nets for learning to play StarCraft combat games. *arXiv preprint arXiv:1703.10069* (2017).

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

[48] Schaarschmidt, M., Kuhnle, A., and Fricke, K. TensorForce: A TensorFlow library for applied reinforcement learning. Web page, 2017.

[49] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[50] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)* (2015), pp. 1889–1897.

[51] Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *International Conference of Learning Representations (ICLR)* (2016).

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

[53] 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.

[54] 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.

[55] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 1998.

[56] Tiwana, A. *Platform Ecosystems: Aligning Architecture, Governance, and Strategy*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.

[57] Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double Q-learning. In *AAAI* (2016), pp. 2094–2100.

[58] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224* (2016).

[59] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

[60] Weber, T., Racanière, S., Reichert, D. P., Buesing, L., Guez, A., Rezende, D. J., Badia, A. P., Vinyals, O., Heess, N., Li, Y., et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203* (2017).

[61] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning 8*, 3-4 (1992), 229–256.

[62] Zoph, B., and Le, Q. V. Neural architecture search with reinforcement learning.

A Algorithm Families

Policy Gradients: This family of algorithms such as Vanilla Policy Gradient [61], TRPO [50, 51], PPO [52] commonly utilize policy evaluation and gradient-based optimization in two separate phases, abiding to the generalized policy iteration model [55]. Some algorithms such as A3C [37] execute the two simultaneously.

Deep Q-Networks: Most DQN variants [14, 29, 38, 49, 57, 59] use a gradient-based optimizer to update the policy, but they additionally utilize a replay buffer as an off-policy data source.

Policy Gradient with Off Policy: These algorithms, such as DDPG [35], Q-Prop [25], ACER [58], UNREAL [30] are extensions of an existing policy gradient method such as A3C that utilize an off policy data source such as a replay buffer to augment the training process.

Model-Based: This includes Model-Based combined with Model-Free learning. This family of algorithms, including I2A [60], IBP [45], Value Prediction Networks [42], augments policy evaluation as a planning subroutine to inform the policy during a rollout.

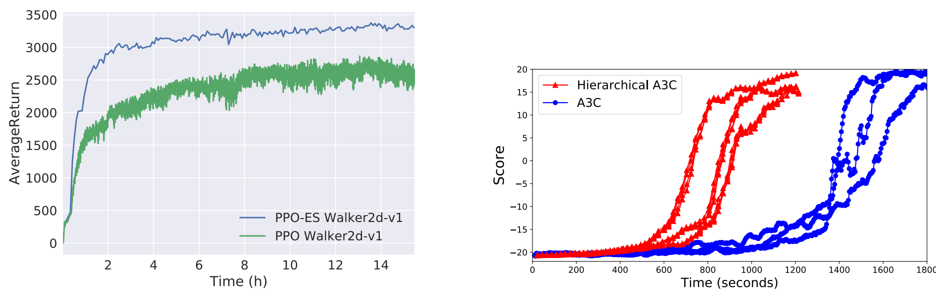
Multi-Agent RL: Modern approaches in this area [36, 20, 39, 46] tend to use gradient-based optimizers along with parallel policy evaluation from each agent to collect data and update the policy. Experience replay [21] has also been considered in this setting.

Evolutionary Methods: Methods such as Evolutionary Strategies [47] and Covariance Matrix Adaptation [27] tend to rely on Derivative Free Optimization and Monte Carlo methods instead of Gradient Based Optimization.

In addition to these broader areas of RL, we also consider the design of large-scale RL systems:

AlphaGo: AlphaGo Zero [54] utilizes multiple parameter servers feeding from a replay buffer, updated from a self-play agent evaluating the best current policy.

B Evolutionary Composition Case Study



(a) PPO-ES (b) A3C-ES

Figure 7: Evolutionary variants of PPO and A3C outperform their base implementations on Walker2d-v1 and PongDeterministic-v4 environments respectively. PPO-ES also converged to a higher reward.

C Choosing the right SGD strategy

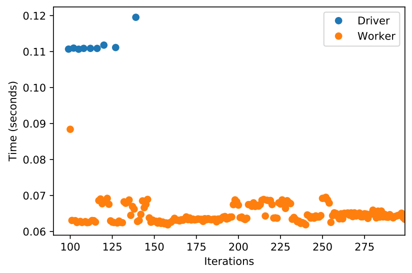
Deep learning frameworks perform many internal optimizations such as operator fusion [9, 12]. However, in many RL workloads data transfers across non-differentiable graph boundaries (e.g. during simulation) incur significant overheads. In the cluster setting, the right physical strategy for RL optimization can vary depending on factors including:

1. **Resource availability**: It may be cost prohibitive to provision GPU devices on each node in a large compute cluster. In this case, it may makes sense to send sample data to nodes with GPUs. Alternatively, a large number of CPUs may be just as fast, e.g. as in A3C.

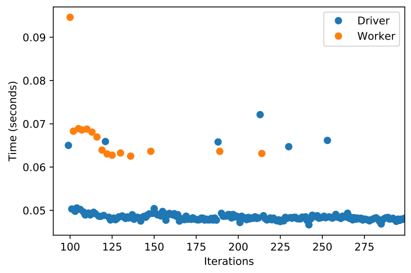
2. **Relative data sizes**: If sample data is small relative to the model, it may be more efficient to send samples to the driver for SGD. Conversely if samples are large, network traffic is minimized by keeping them local to worker processes.

3. **Algorithm**: Algorithm-specific details can often be exploited. For example, PPO's loss function permits multiple SGD passes. Table 2 shows how this can be exploited in certain cases to amortize CPU-to-GPU copy costs, increasing SGD throughput by 6x.

RLlib chooses an initial SGD strategy based on simple algorithm and environment-specific heuristics. Thanks to the task-based programming model, it can also adaptively switch strategies based on runtime statistics (Figure 8).



(a) Gradient latency history, CPU-only



(b) Gradient latency history with GPU driver

Figure 8: RLlib can adaptively identify the best SGD strategy, switching to the optimal choice within a few iterations. Here the optimal strategy for A3C (computing gradients on the driver vs on remote workers) changes depending on whether the driver has access to a GPU.

D Evaluation Hyperparameters

D.1 Hyperparameters for Evolution Strategies

Hyperparameter	Value
Noise standard deviation	0.02
Adam stepsize	0.01
L2 coefficient	0.005
Episodes per batch	5000

D.2 Hyperparameters for Proximal Policy Optimization

Hyperparameter	Value
Timesteps per batch	320000
SGD minibatch size	32768
SGD epochs per iteration	20
Adam stepsize	1e-4
PPO clip param	0.2
GAE parameter (λ)	0.95
Discount (γ)	0.995

E Case Study Pseudocode

E.1 Offline Data Evaluator

```
class OfflineEvaluator(Evaluator):
    """Example of how an Evaluator can bridge RL training and serving."""

    def __init__(self, config):
        self.dbi = ...  # setup connection to data store
        self.srv_client = ...  # setup RPC client to online serving system
        self.model = self.srv_client.get_current_policy()  # some policy

    def sample(self):
        """Pulls samples from the offline database of experience.

        The query may need to be parameterized by the current policy to support
        training of on-policy models.
        """

        return self.dbi.query(...)

    def set_weights(self, weights):
        """Pushes a new model to the online serving system.

        Note that after setting local weights, this just suggests a model for
        deployment; actual deployment would be governed by the serving system.
        """

        self.model.set_weights(weights)
        self.srv_client.suggest_model(self.model)

    def grad(self, samples):
        return self.model.grad(samples)

    def weights(self):
        return self.model.weights()
```

E.2 PPO-ES

```
class PPOES(Algorithm):
    """Simple nesting of PPO within an ES loop that perturbs weights."""

    def __init__(self, env_creator, config):
        Algorithm.__init__(self, env_creator, config)
        self.variants = [
            ray.remote(ppo.PPOAgent).remote(self.env_creator, config)
            for _ in range(config["num_variants"])]

    def train(self):
        scores = {}
        pending = {r.train.remote(): r for r in self.variants}
        start = time.time()
        best_result = None
        timesteps = 0
        while pending:
            [result], _ = ray.wait(list(pending.keys()))
            agent = pending[result]
            del pending[result]
            result = ray.get(result)
            timesteps += result.timesteps_this_iter
            if (not best_result or result.episode_reward_mean >
                    best_result.episode_reward_mean):
                best_result = result
            scores[agent] = result.episode_reward_mean
            if time.time() - start < self.config["es_iteration_time"]:
                pending[agent.train.remote()] = agent

        by_score = sorted(
            [(score, agent) for (agent, score) in scores.items()])
        print("Agent rankings: {}".format(by_score))

        # Now clone the top half and add some random weight perturbations.
        ranked = [agent for (_, agent) in by_score]
        mid = len(ranked) // 2
        top_half, bottom_half = ranked[mid:], ranked[:mid]
        for winner, loser in zip(top_half, bottom_half):
            if random.random() > 0.5:
                loser.set_weights.remote(winner.get_weights.remote())
                perturb_weights.remote(loser, self.config["es_noise_stdev"])

        return best_result._replace(timesteps_this_iter=timesteps)


@ray.remote
def perturb_weights(model, noise_stdev):
    weights = ray.get(model.get_weights.remote())
    perturbation = noise_stdev * np.random.randn(len(weights))
    model.set_weights.remote(weights + perturbation)
```

E.3 AlphaGo Zero

```
class AlphaGoZero(Algorithm):
    """Outline of a top-level Ray RL algorithm for AlphaGo Zero.

    This models the various distributed components in AlphaGo Zero as Ray
    actors. The train() call processes results from these components, routing
    data and launching new component actors as needed.

    Helper classes:
        AlphaGoEvaluator - Evaluator for self-play, holding the AlphaGo model.
        AlphaGoOptimizer - Optimizer wrapping SyncLocalOptimizer.
        SharedReplayBuffer - Evaluator backed by a replay buffer.

    Helper functions:
        elo_estimate - Estimates the strength of the given model.
        compare_models - Compares two models, returning the victory margin.
    """

    def __init__(self):
        self.best_model = AlphaGoEvaluator()
        self.replay_buffer = SharedReplayBuffer.remote()

        # AlphaGoOptimizers pull from the shared replay buffer to improve their
        # model. Note that AlphaGo Zero has many concurrent optimizers.
        self.optimizers = [
            AlphaGoOptimizer.remote(self.replay_buffer) for _ in range(20)]
        self.optimizer_tasks = set()
        weights = self.best_model.get_weights()
        # Initialize and kick off optimizer tasks
        for opt in self.optimizers:
            opt.set_weights.remote(weights)
            self.optimizer_tasks.append(opt.step.remote())
        # Self-play evaluators constantly evaluate the current best model and
        # add samples into the shared replay buffer.
        self.self_play_evaluators = [
            AlphaGoEvaluator.remote() for _ in range(20)]
        self.self_play_tasks = set()
        for ev in evaluators:
            self.self_play_tasks.append(ev.sample.remote())

        # When optimizers finish an optimization round, their model is compared
        # with the current best model to see if it can replace it. This set
        # tracks those concurrent comparison tasks.
        self.compare_tasks = set()

    def train(self):
        """Implements the train() method required for all Ray RL algorithms."""

        result = elo_estimate.remote(self.best_model)
        start = time.time()

        # Return when enough time has passed and our new ELO estimation
        # finishes (the model will always be slightly ahead of the ELO).
        while time.time() - start < 60 or not ray.wait(result, timeout=0.0):
            self._step()

        return result

    def _step(self):
        """One step of the event loop that coordinates AlphaGo components."""

        # For simplicity, assume ray.wait() also returns the worker that ran
        # the task. As of Ray 0.2 you would need to track this in a dict.
        result_id, result_worker, _ = ray.wait(
            self.optimizer_tasks + self.self_play_tasks + self.compare_tasks)

        if result_id in self.optimizer_tasks:
            # Launch a compare task to see if we can beat the best model
            self.compare_tasks.append(
                compare_models.remote(best_model, result_id))
            # Continue optimization on this worker
            self.optimizer_tasks.append(result_worker.step.remote())
            self.optimizer_tasks.remove(result_id)

        if result_id in self.self_play_tasks:
            # Tell the replay buffer to incorporate the new rollouts
            self.replay_buffer.add_samples.remote(result_id)
            # Continue self play on this worker
            self.self_play_tasks.append(result_worker.sample.remote())
            self.self_play_tasks.remove(result_id)

        if result_id in self.compare_tasks:
            self.compare_tasks.remove(result_id)
            result = ray.get(result_id)
            # If it beats the current best model, broadcast new weights to
            # all evaluators and optimizers.
            if result.win_ratio >= 0.55:
                self.best_model = ray.get(result.model)
                weights = ray.put(self.best_model.get_weights())
                for ev in self.self_play_evaluators:
                    ev.set_weights.remote(weights)
                for opt in self.optimizers:
                    opt.set_weights.remote(weights)
                ray.kill(self.compare_tasks)
                self.compare_tasks = []


@ray.remote
class AlphaGoEvaluator(Evaluator):
    def sample(self):
        # return self-play rollouts based on current policy. Note that
        # self-play itself may be parallelized, launching further remote tasks.

    # standard update methods around the current policy
    def grad(self, samples): ...
    def weights(self): ....
    def set_weights(self, weights): ...


@ray.remote
class AlphaGoOptimizer(SyncLocalOptimizer):
    def __init__(self, replay_buffer):
        self.evaluator = AlphaGoEvaluator()
        SyncLocalOptimizer.__init__(self, self.evaluator, [replay_buffer])

    def set_weights(self, weights):
        self.evaluator.set_weights(weights)


@ray.remote
class SharedReplayBuffer(Evaluator):
    def add_samples(self, samples):
        # adds the given samples to the internal buffer

    def sample(self):
        # return batch of experience from internal buffer

    # don’t need to implement these since AlphaGoOptimizer never calls them
    def grad(self, samples): pass
    def weights(self): pass
    def set_weights(self, weights): pass


@ray.remote
def compare_models(current_best, candidate):
    # compares the models, returning the candidate win ratio


@ray.remote
def elo_estimate(model):
    # returns an ELO estimate of the given model
```

E.4 Hyperparam Search Integration

```
# For each trial, Ray’s hyperparam tuner creates a remote instance of the
# algorithm class, which in turn may create other remote processes as needed
# for parallelization, e.g. PPO may create many remote evaluator processes.
# This enables seamless evaluation of parallel algorithms.
class ExampleAlgorithm(Algorithm):
```

```
# result format for hyperparam search integration
from ray.tune.result import TrainingResult
...

def __init__(self, env_creator, config):
    ...  # Hyperparameters are passed in the config.

def train():
    ...
    # To integrate with the Ray hyperparam tuner, algorithms must
    # implement a method that returns a training result object.
    return TrainingResult(
        episode_reward_mean=returns.mean(),
        episode_len_mean=lengths.mean(),
        timesteps_this_iter=lengths.sum())
```