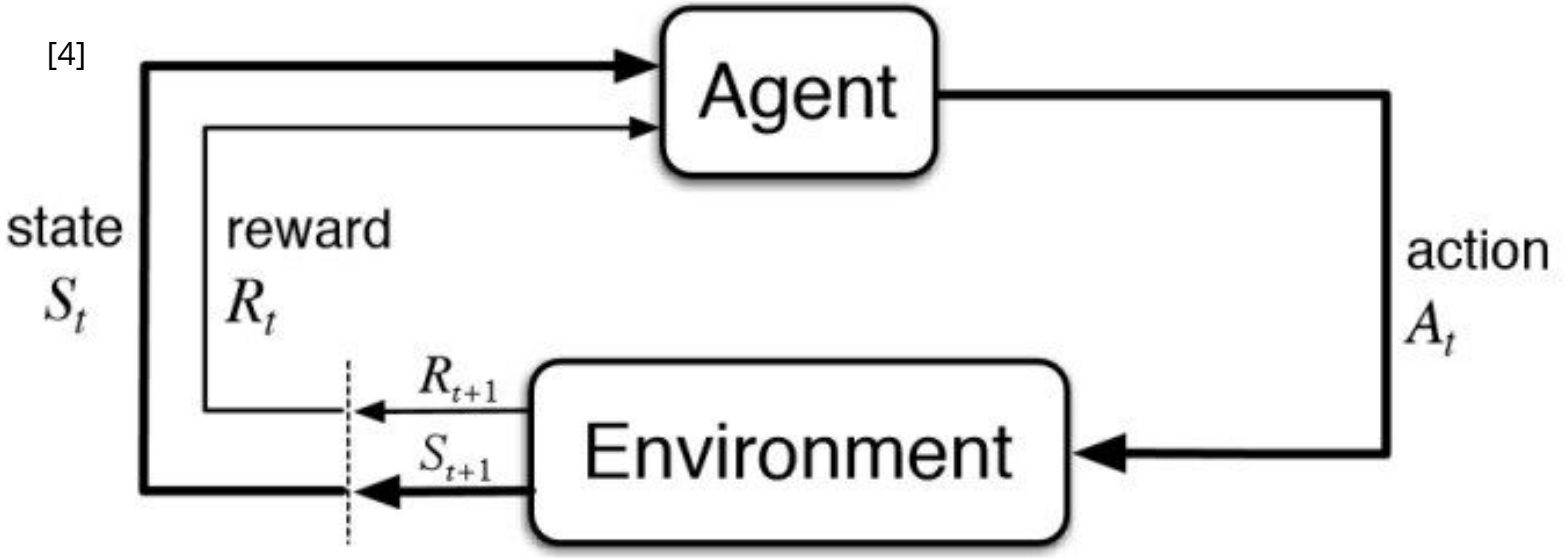
RLlib: Abstractions for Distributed Reinforcement Learning

Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica

R244 Presentation By: Vikash Singh November 14, 2018 Session 6

What is Reinforcement Learning (RL) ?

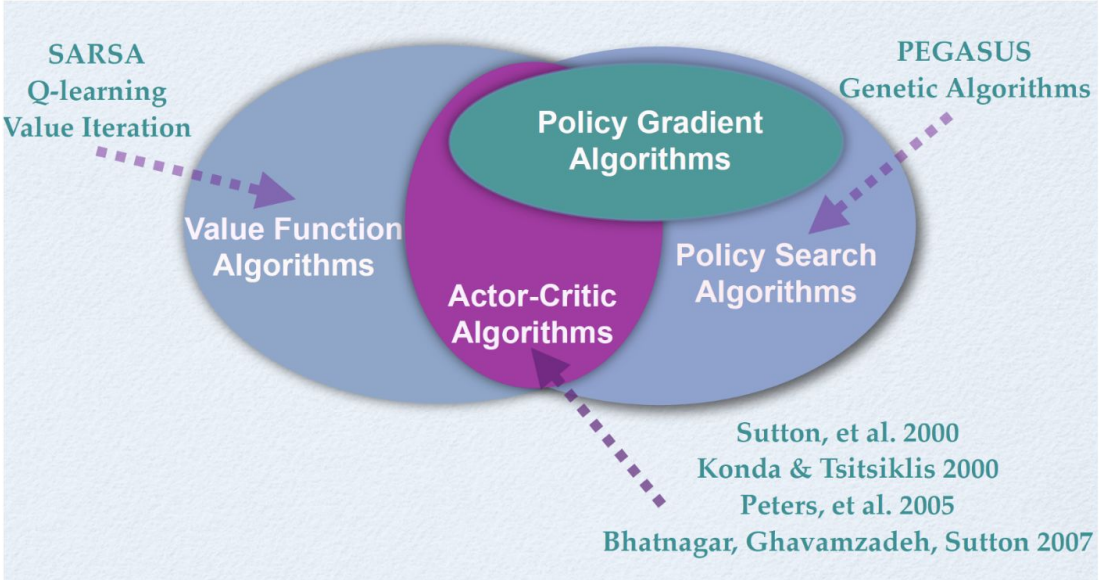
[4]



Understanding the Goal of RL

- **Policy:** Strategy used by the agent to determine which action to take given its current state
- **Goal:** Learn a policy to optimize long term reward

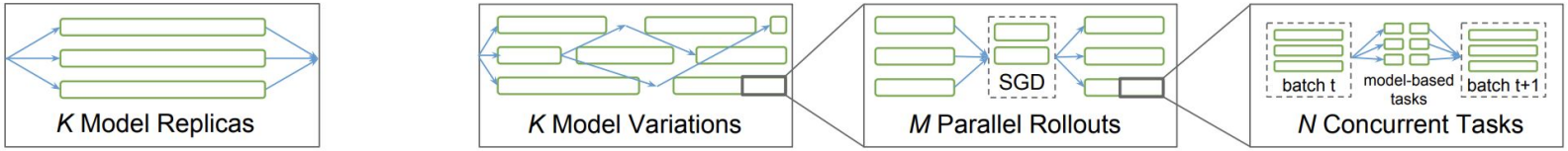
[2]



Problem with Distributed RL

- Absence of a single dominant computational pattern or rules of composition (e.g., symbolic differentiation)
- Many different heterogeneous components (deep neural nets, third party simulators)
- State must be managed across many levels of parallelism and devices
- People forced to build custom distributed systems to coordinate without central control!

Nested Parallelism in RL



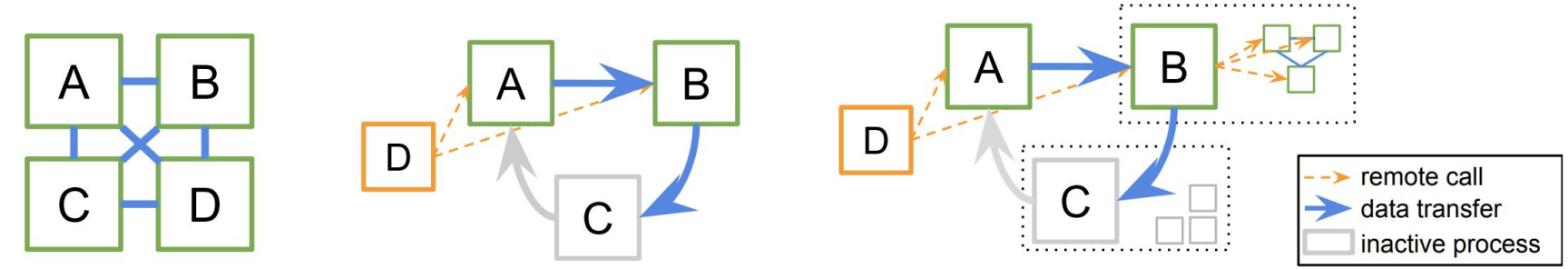
(a) Deep Learning (b) Reinforcement Learning

Opportunities for distributed computation in this nested structure! How to take advantage of this ?

RLlib: Scalable Software Primitives for RL

- Abstractions encapsulate parallelism and resource requirements
- Built on top of Ray[1] (task based system for distributed execution)
- Logically centralized top down hierarchical control
- Reuse of components for rapid prototyping, development of new RL algorithms

Hierarchical and Logically Centralized Control



(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.

Example: Distributed vs Hierarchical Control

```
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.

Abstractions for RL

- **Policy Graph**: define **policy** (could be neural network in TF, Pytorch), **postprocessor** (Python function) , and **loss**
- **Policy Evaluator:** wraps policy graph and environment to sample experience batches (can specify many replicas)
- **Policy Optimizer:** extend gradient descent to RL, operates closely with the policy evaluator

Advantages of Separating Optimization from Policy

- Specialized optimizers can be swapped in to take advantage of hardware without changing algorithm
- Policy graph encapsulates interaction with deep learning framework, avoid mixing deep learning with other components
- Rapidly change between different choices in RL optimization (synchronous vs. asynchronous, allreduce vs parameter server, use of GPUs and CPUs, etc)

Common Themes in RL Algorithm Families

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

Complex RL Architectures using RLlib

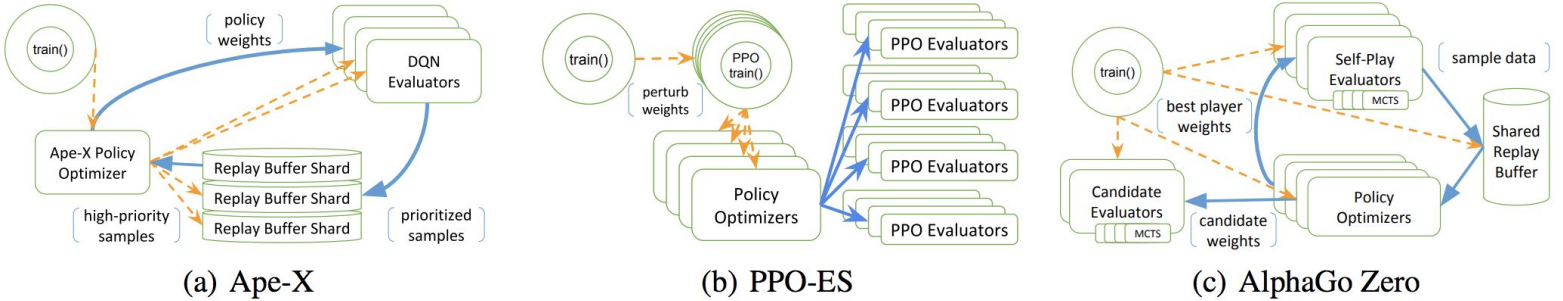
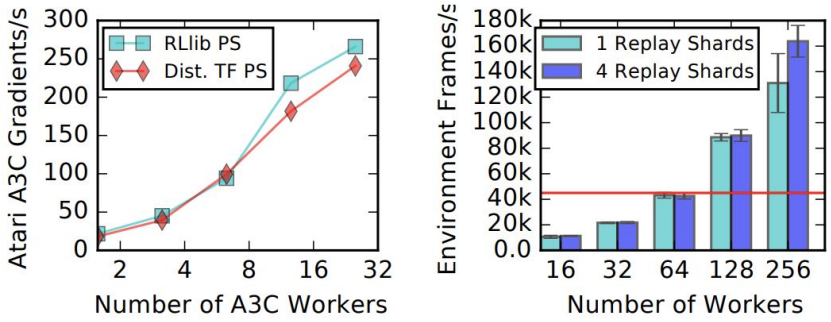


Figure 6. Complex RL architectures are easily captured within RLlib's hierarchical control model. Here blue lines denote data transfers, orange lines lighter overhead method calls. Each train() call encompasses a batch of remote calls between components.

RLlib vs Distributed TF Parameter Server



(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.

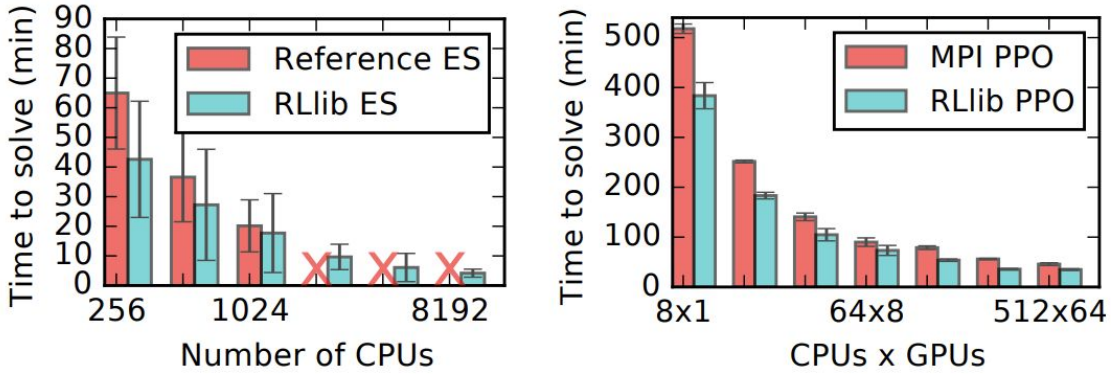
Key Questions:

- Can a centrally controlled policy optimizer compete in performance with an implementation in a specialized system like Distributed $TF_{[3]}$?
- Can a single threaded controller scale to large throughputs?

Scalability of Distributed Policy Evaluation

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.

More Performance Comparisons to Specialized Alternatives



(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.

Policy Optimizer Comparison in Multi-GPU Conditions

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.

Minor Criticism

- Comparisons could be more exhaustive to cover more RL strategies
- Abstractions may be potentially limiting for newer models that don’t align with this paradigm
- Unclear how involved developer needs to be in resource awareness to achieve optimal performance

Final Thoughts

- RLlib presents a useful set of abstractions that simplify the development of RL systems, while also ensuring scalability
- Successfully breaks down RL ‘hodgepodge’ of components into separate, reusable components
- Logically centralized hierarchical control with parallel encapsulation prevents messy errors from coordinating separate distributed components

References

1. Moritz, Philipp, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications." *arXiv preprint arXiv:1712.05889*(2017).
2. Seo, Jae Duk. "My Journey to Reinforcement Learning - Part 0: Introduction." Towards Data Science. April 06, 2018. Accessed November 06, 2018. https://towardsdatascience.com/my-journey-to-reinforcement-learning-part-0-introduction-1e3aec1ee5bf.
3. Vishnu, Abhinav, Charles Siegel, and Jeffrey Daily. "Distributed tensorflow with MPI." *arXiv preprint arXiv:1603.02339* (2016).
4. "KDnuggets." KDnuggets Analytics Big Data Data Mining and Data Science. Accessed November 06, 2018. https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html.