Scalable Reinforcement Learning with RLlib

Eric Liang and Richard Liaw

#ray #rllib

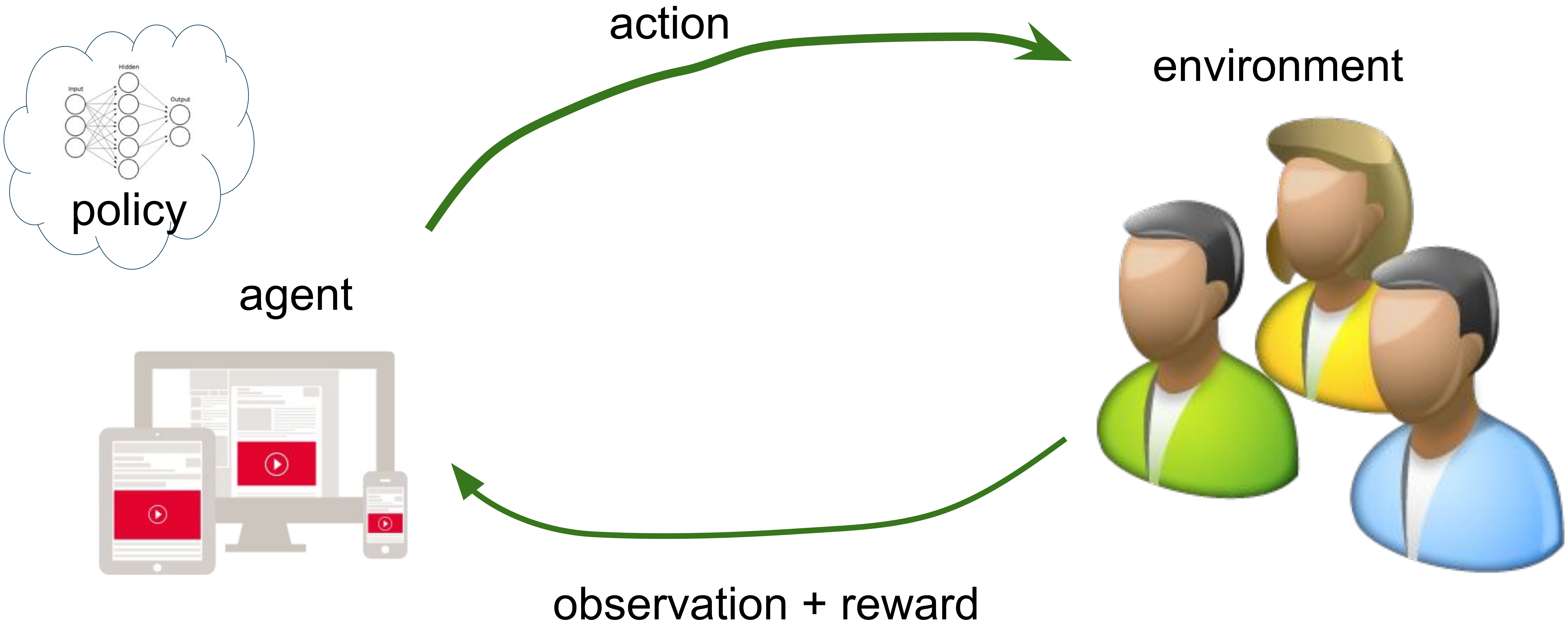
Talk overview

Reinforcement learning (RL)

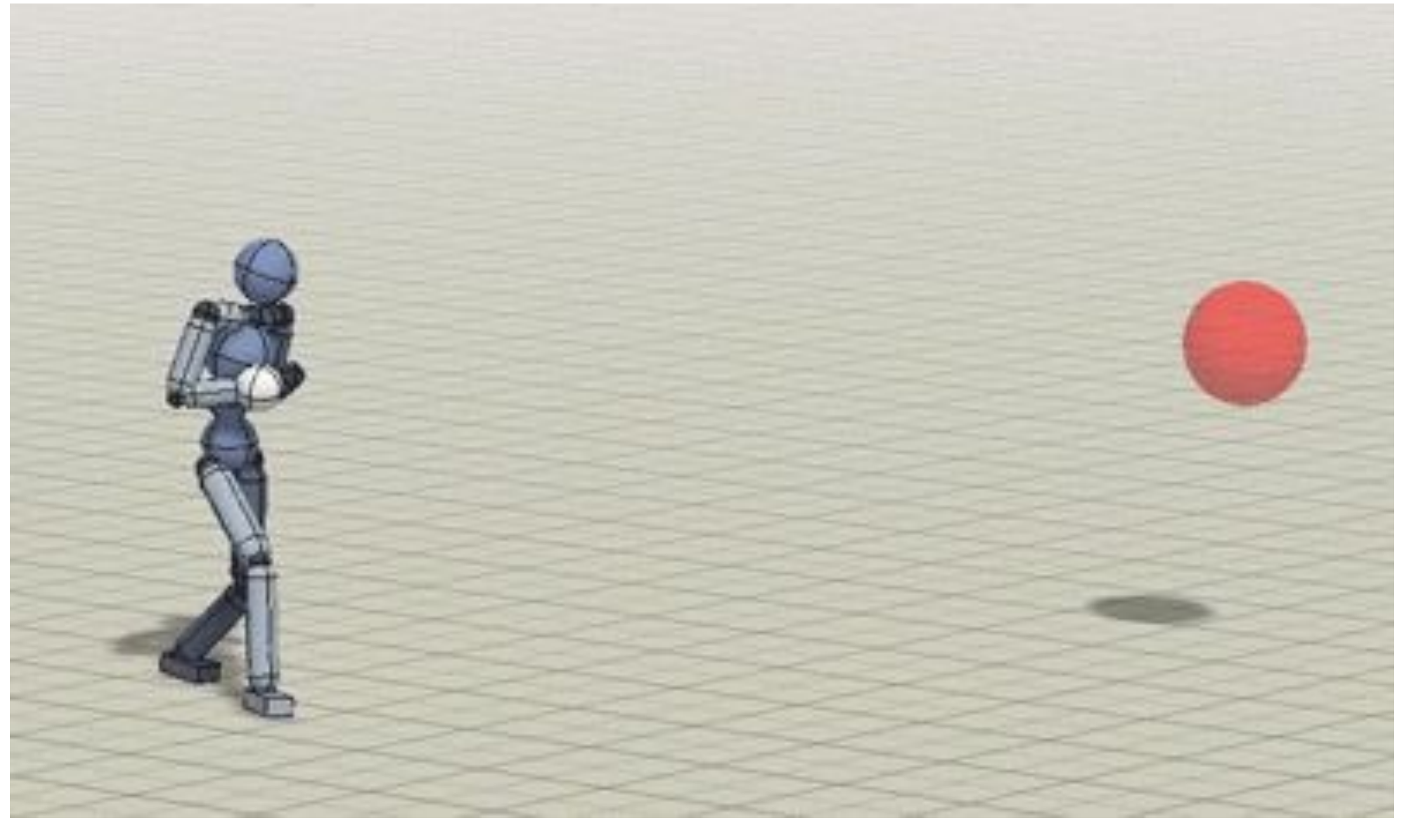
Leveraging **Ray** for distributed AI

RLlib and Abstractions for scalable RL

Reinforcement Learning is centered around **interaction**



Applications of RL



How do we improve RL?

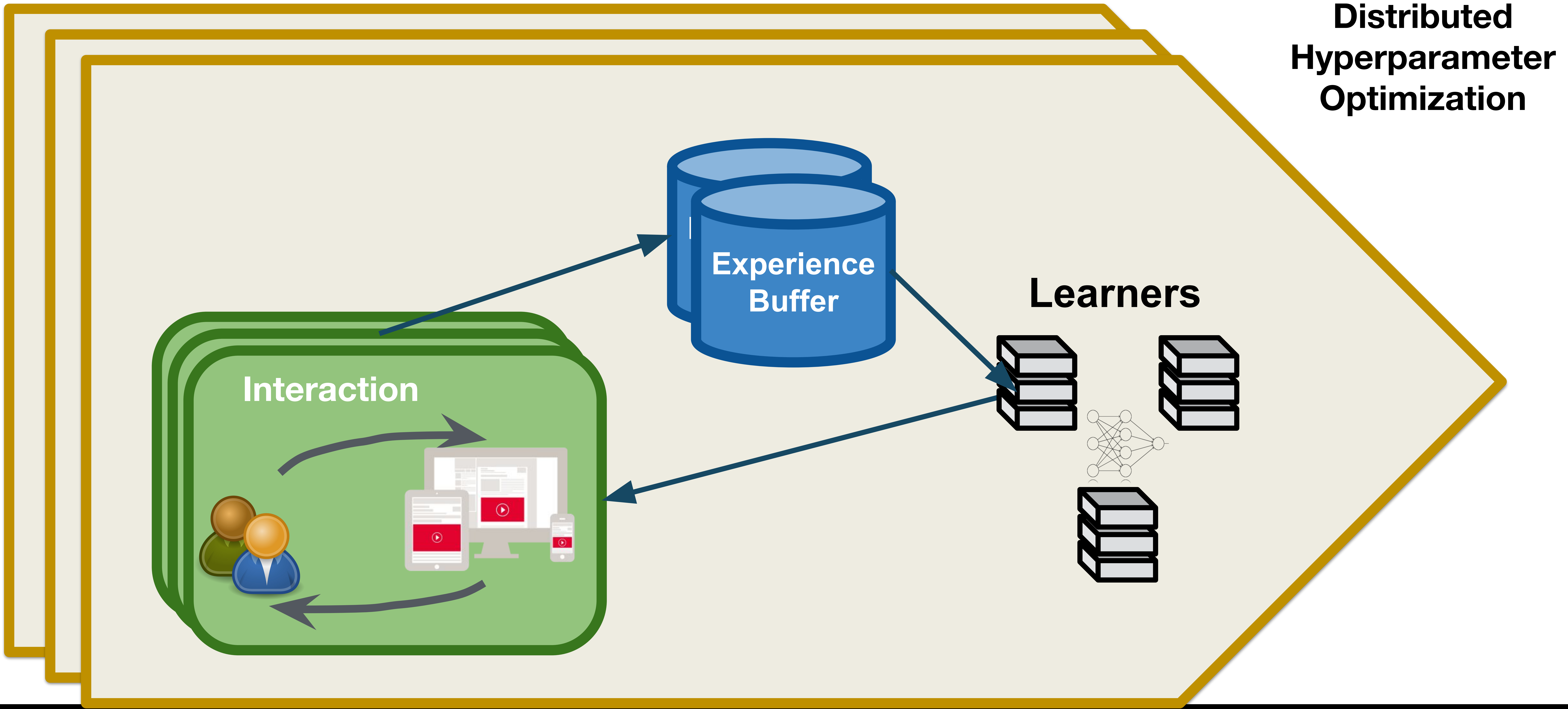
More data from interaction

+

More Compute

Improved Performance

Distributed RL



How to do distributed RL?

Abstractions for Reinforcement Learning

Distributed Execution Environment

Hardware

Ray

Provides task parallel API, actor API, and DataFrame API

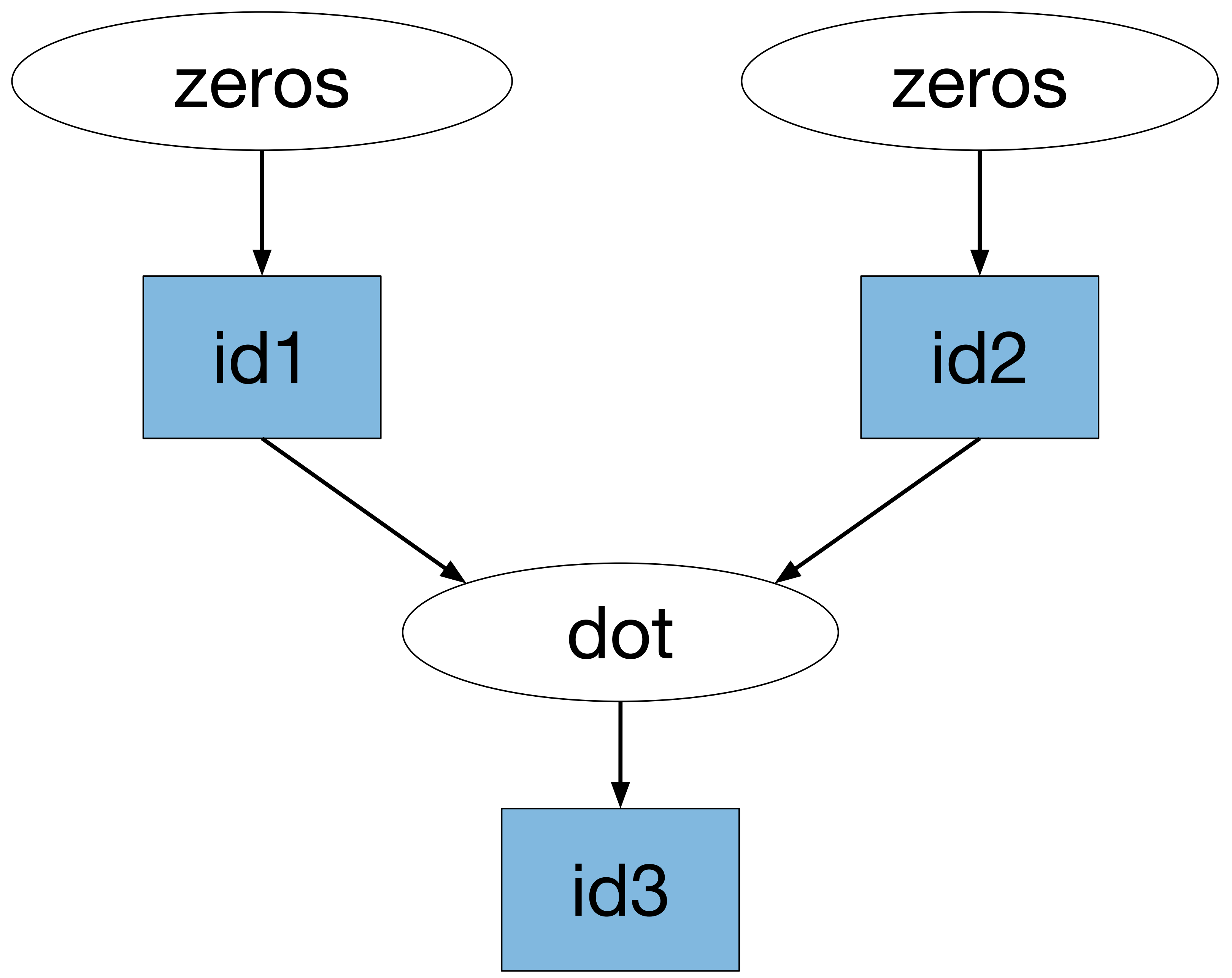
Abstractions for Reinforcement Learning

Distributed Execution Environment

Hardware

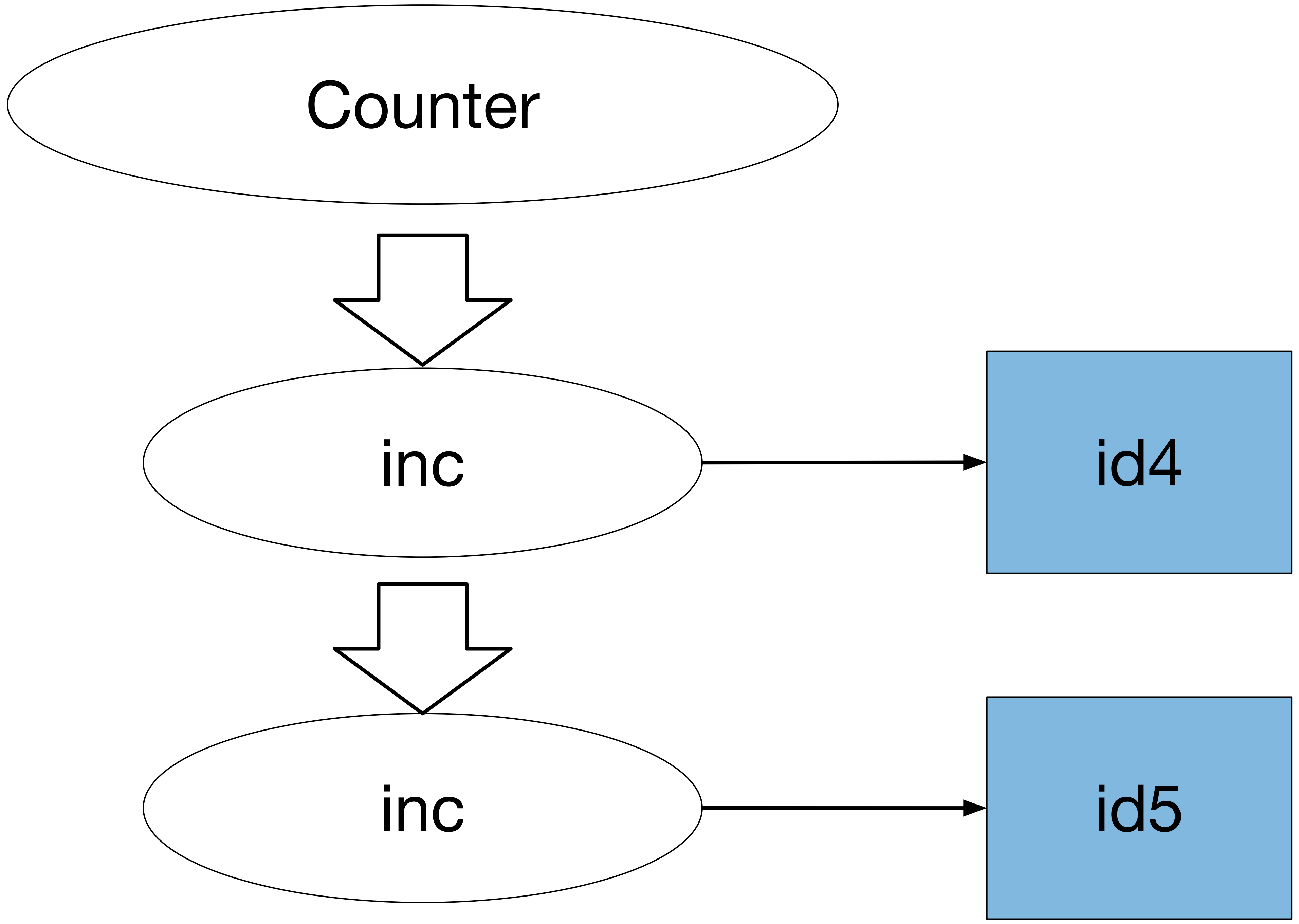
Ray provides a **Task parallel API**

```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
  return np.dot(a, b)

id1 = zeros.remote([5, 5])
id2 = zeros.remote([5, 5])
id3 = dot.remote(id1, id2)
result = ray.get(id3)
```

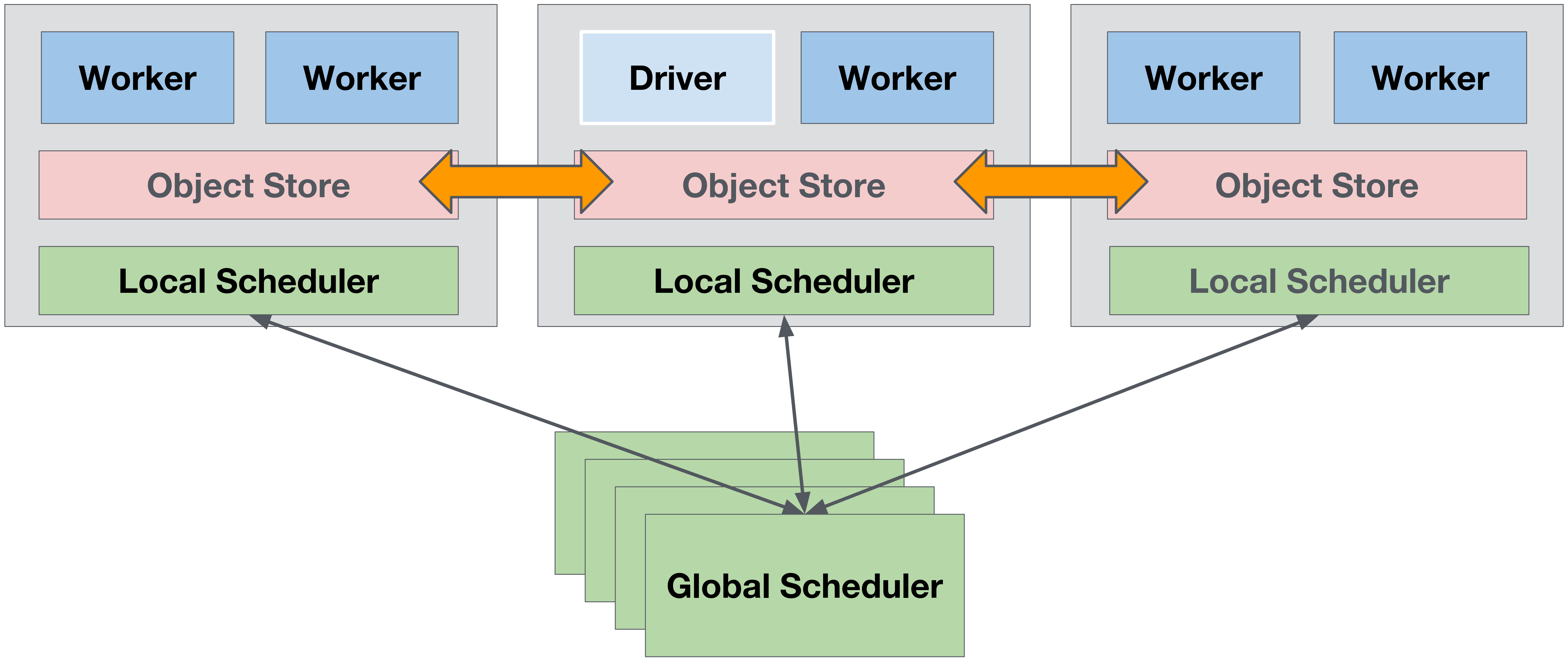


Ray also provides an **actor API**

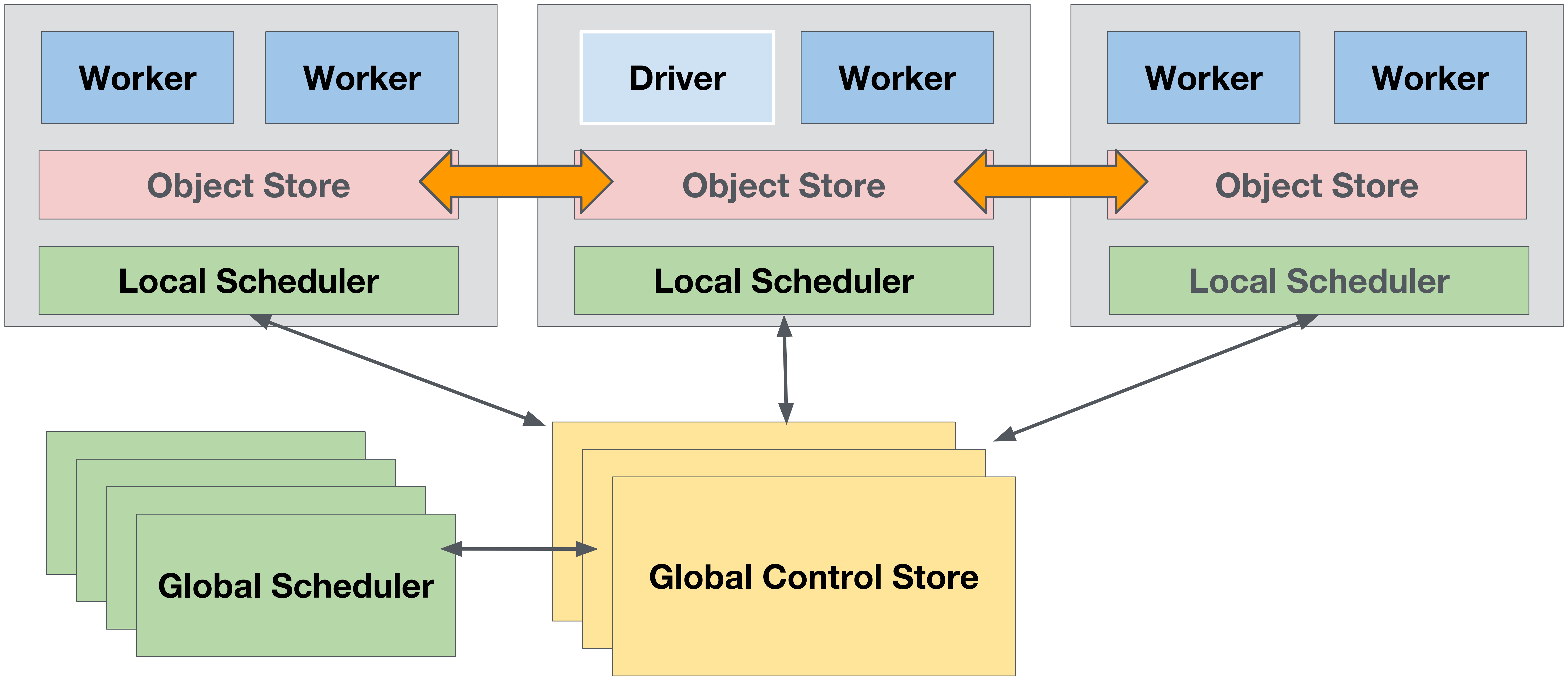
```python
@ray.remote(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter.remote()
id4 = c.inc.remote()
id5 = c.inc.remote()
result = ray.get([id4, id5])
```



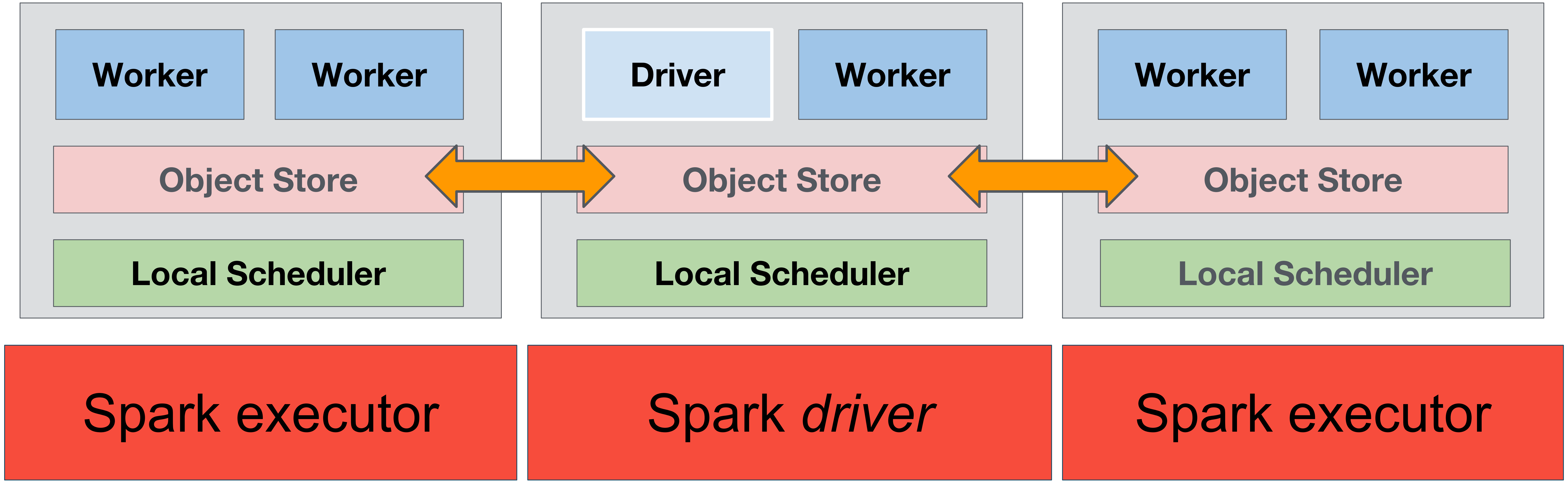
Ray Architecture Overview



Ray Architecture Overview



You can run Ray on Spark




```
$ pip install ray
> sc.parallelize(1 to 100).mapPartitions(_ =>
  "ray start --redis-address=DRIVER_ADDR"!!)
```

Ray Libraries

tune

ray sgd

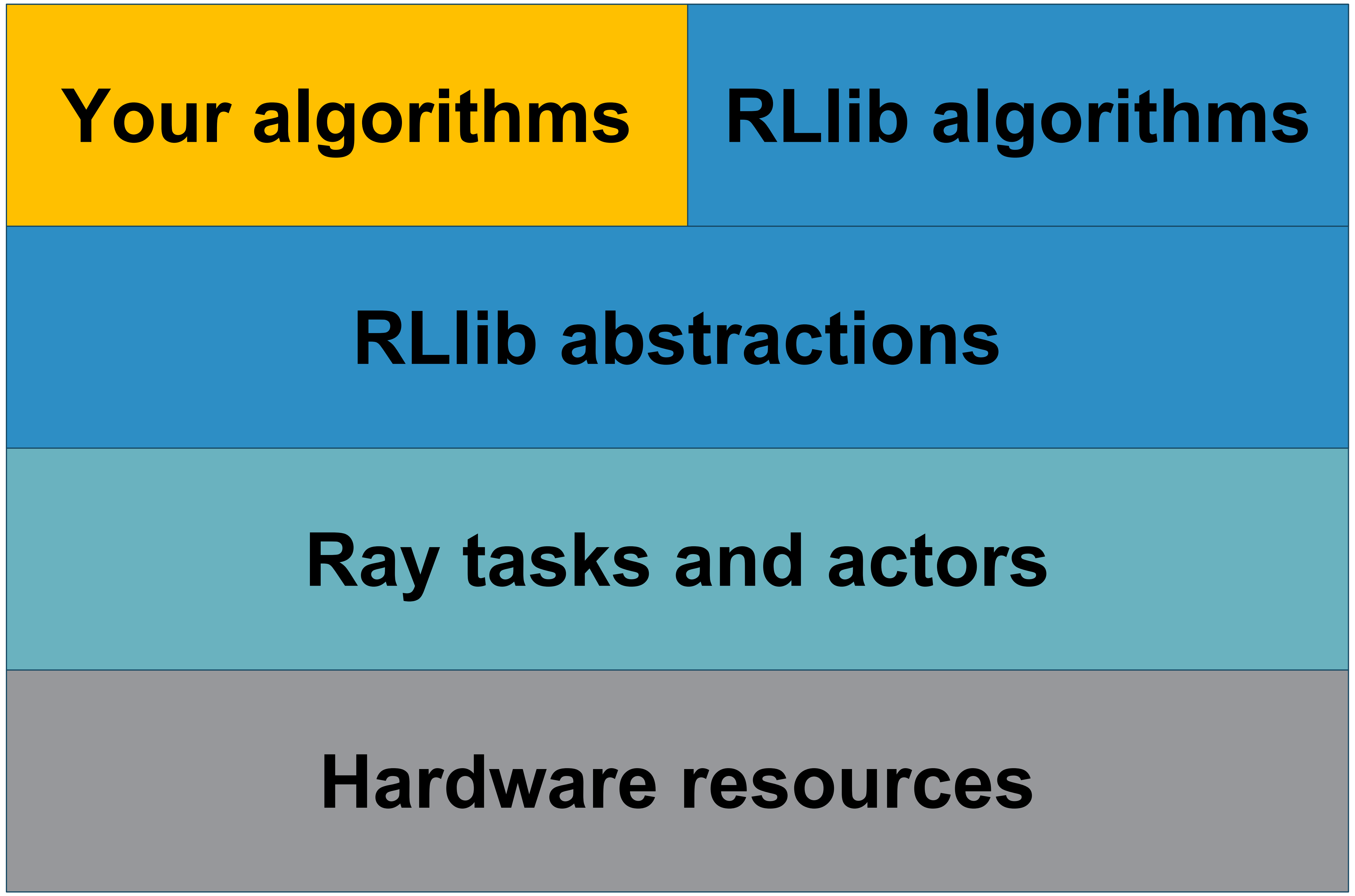
Pandas on Ray

rllib

Ray tasks and actors

Hardware resources

What is RLlib



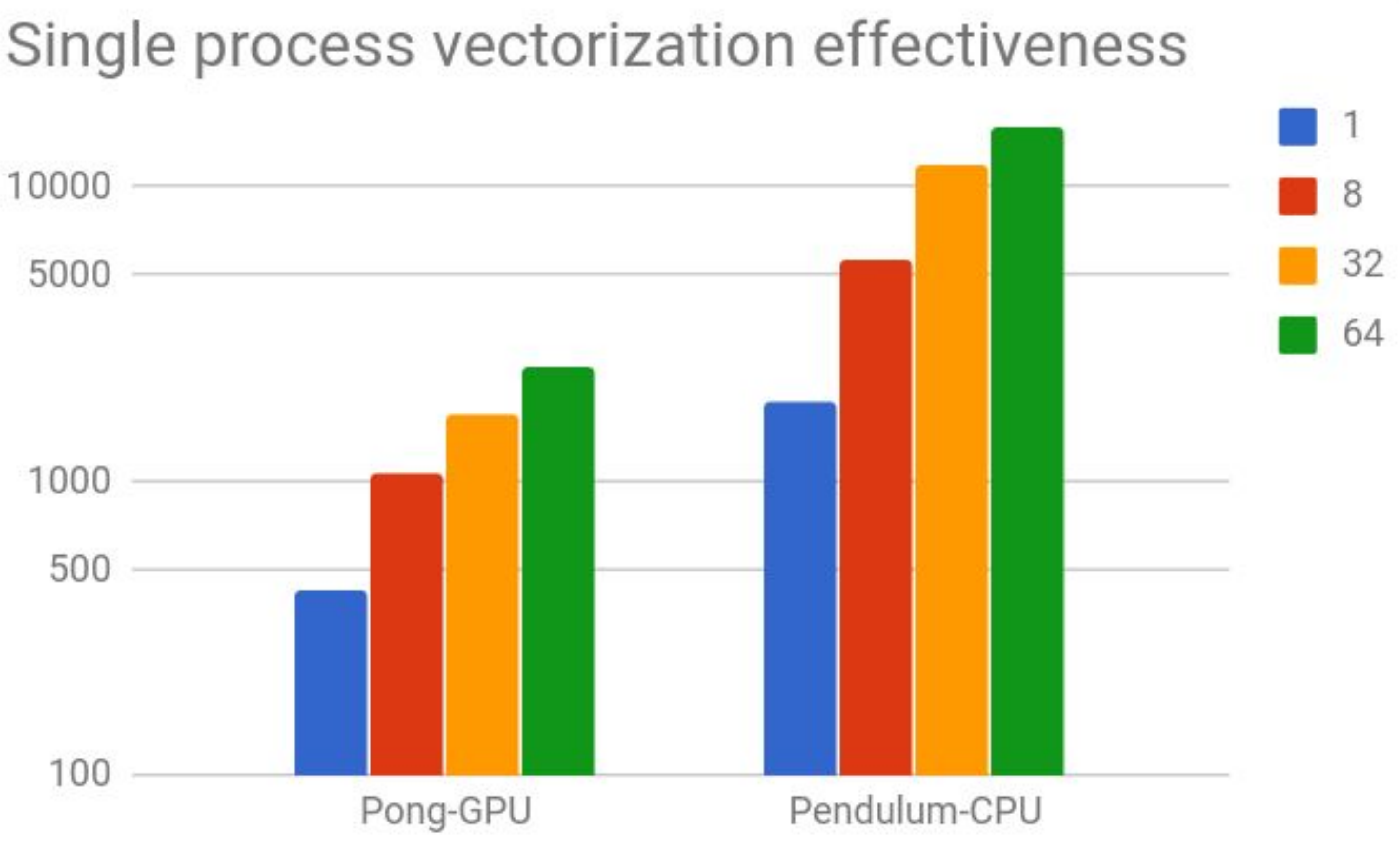
RLlib is easy to get started with

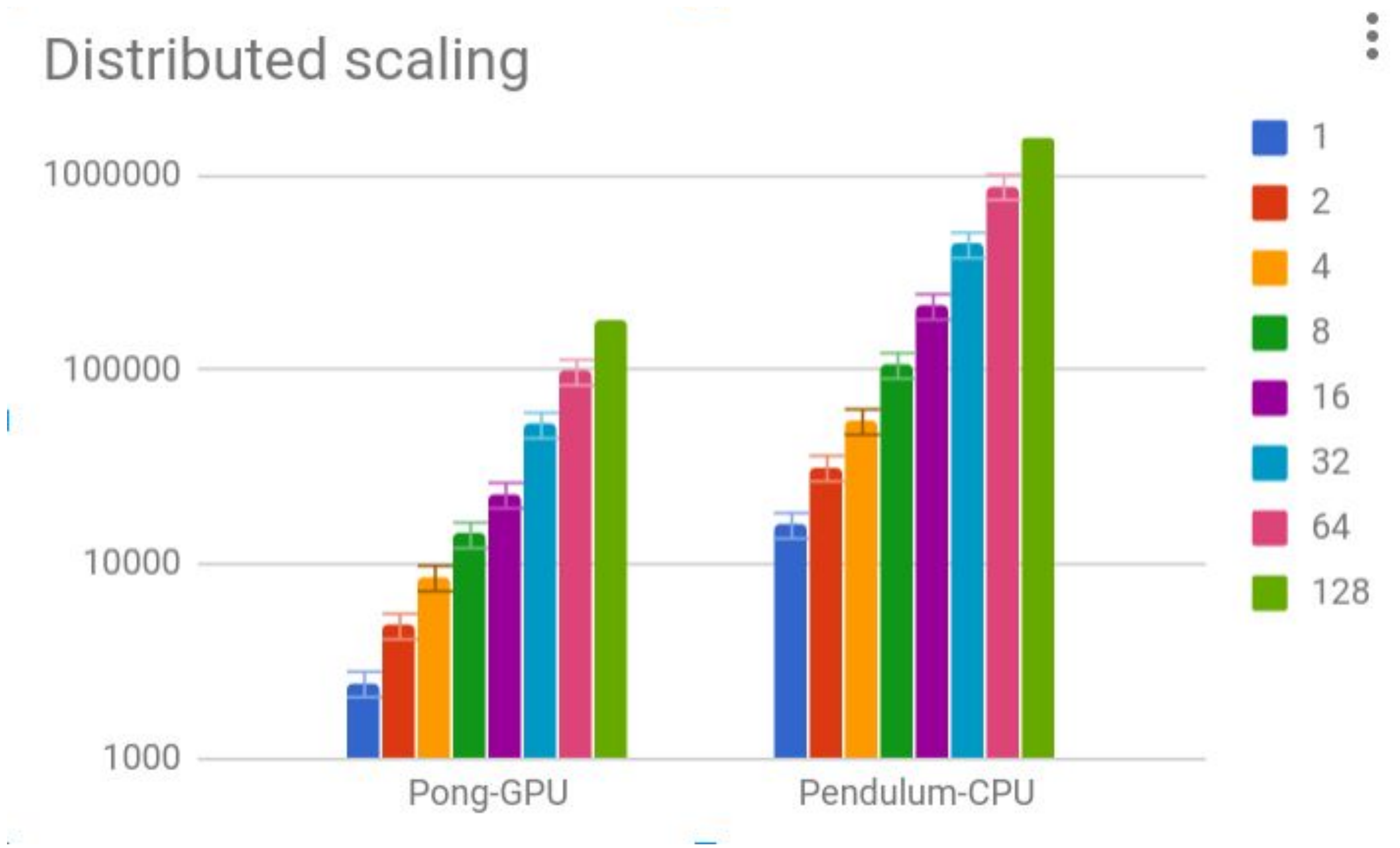
```
./train.py --env=CartPole-v0 --run=DQN
```

RLlib has a simple Python API

```
from ray.rllib.dqn import DQNAgent

env_creator = lambda config: my_env()
agent = DQNAgent(env_creator=creator)
while True:
    print(agent.train())
```

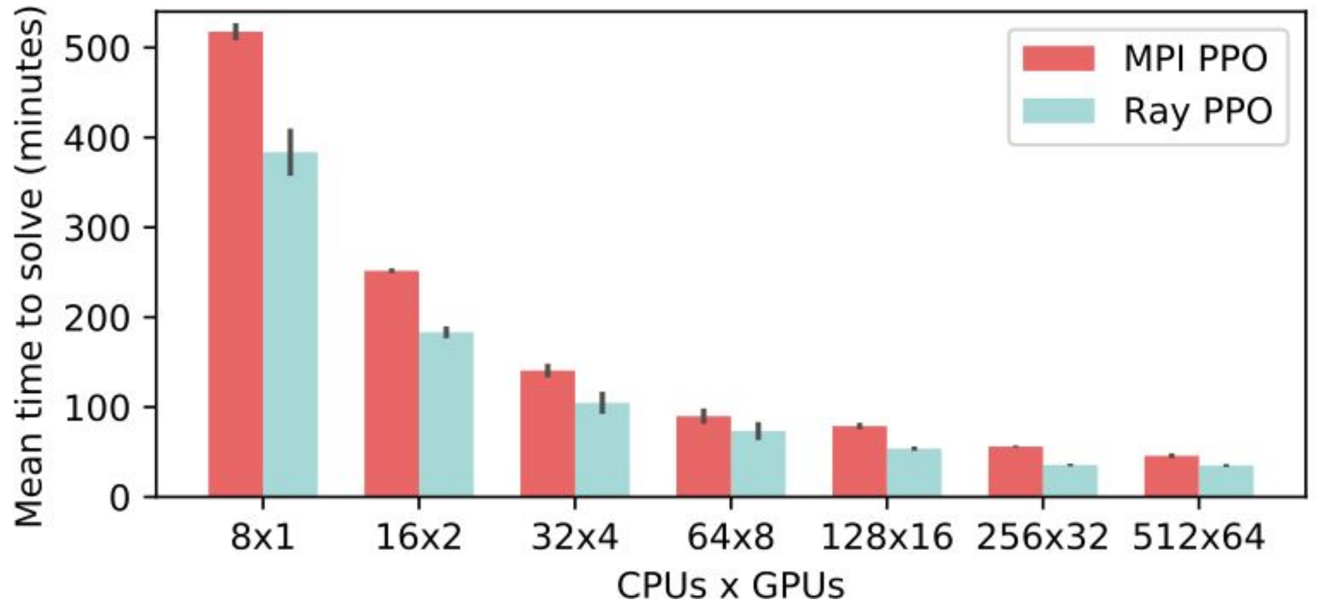
RLlib efficiently scales to multi-core and clusters



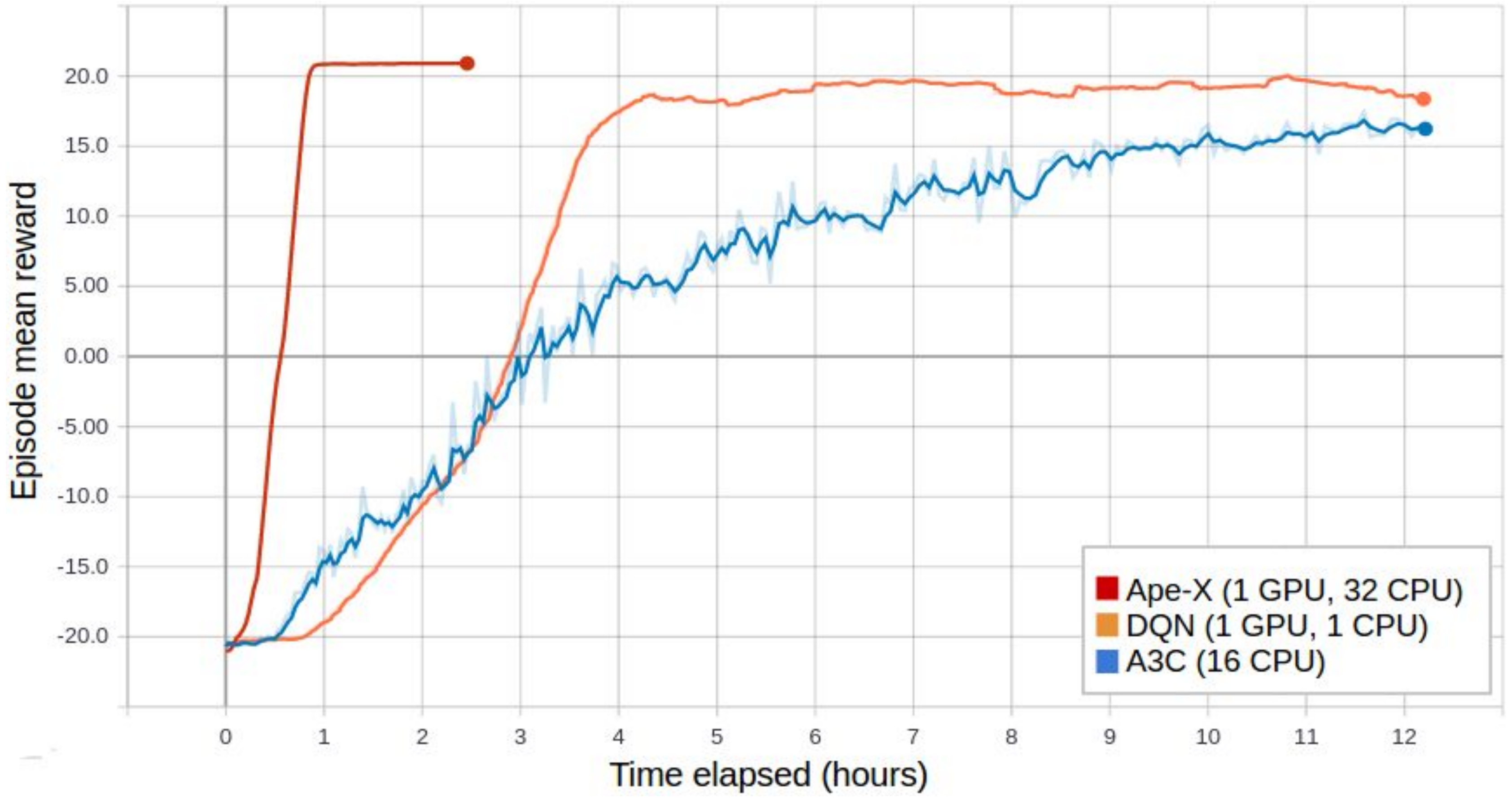


Unified framework for scalable RL

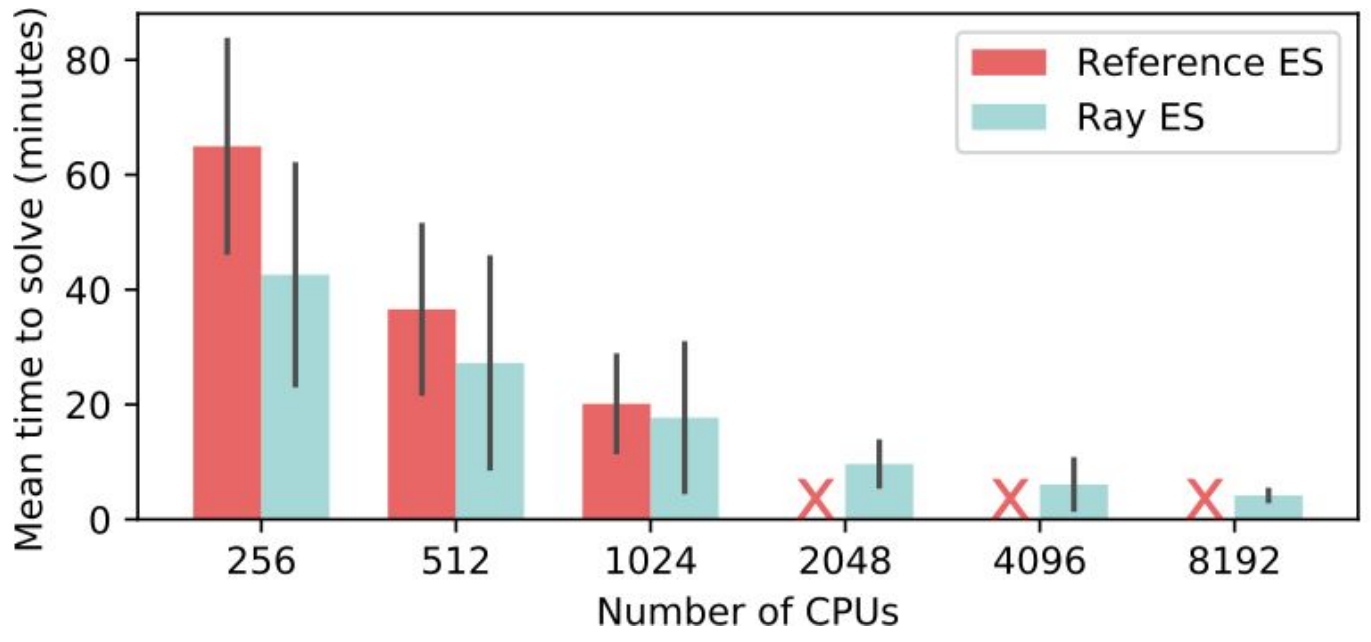
Distributed PPO
(vs OpenMPI)



Ape-X Distributed DQN, DDPG



Evolution Strategies
(vs Redis-based)



RLlib algorithms and optimizers

Current RLlib Algorithms:

Policy Gradients (PG)

Proximal Policy Optimization (PPO)

Asynchronous Advantage Actor-Critic (A3C)

Deep Q Networks (DQN)

Evolution Strategies (ES)

Deep Deterministic Policy Gradients (DDPG)

Ape-X Distributed Prioritized Experience Replay, including both DQN and DPG variants

work in progress: IMPALA

work in progress: TRPO

RLlib Policy Optimizers:

AsyncOptimizer

SyncLocalOptimizer

SyncLocalReplayOptimizer

LocalMultiGPUOptimizer

ApexOptimizer

all scale from laptop to clusters

Community Contributions

Alibaba Group

RLlib makes implementing algorithms simple

- Developer specifies policy, postprocessor, loss

Neural network
in TF / PyTorch / etc.

Python function

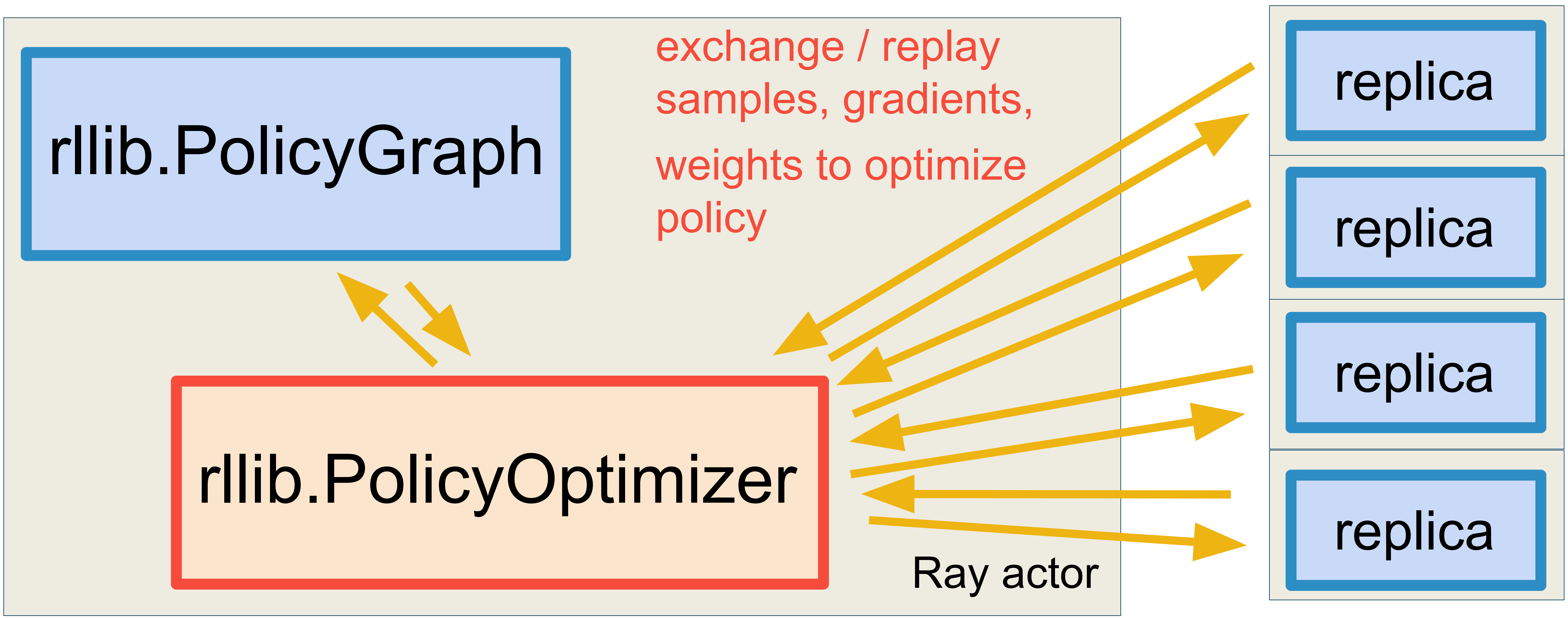
Tensor ops in
TF / Pytorch

class rllib.PolicyGraph

Scale RL algorithms with RLlib

- Use RLlib to define your learning algorithm
- Use RLlib to scale training to a cluster

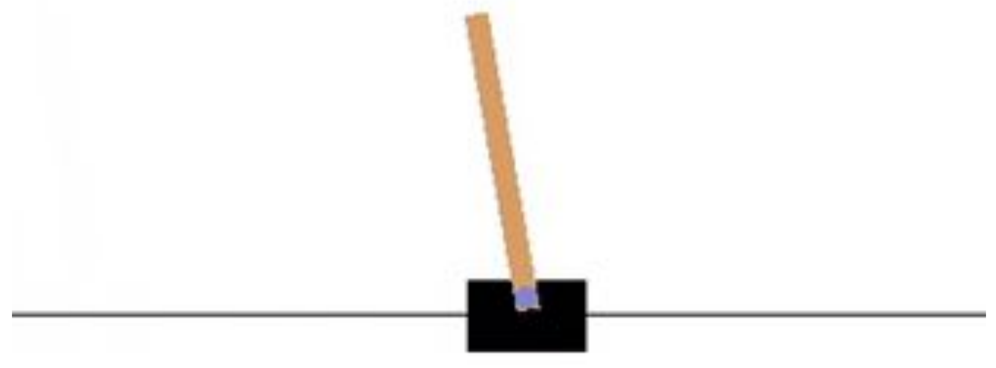
RLlib abstractions



RLlib example algorithms

1. Simple parallel policy gradient
2. Ape-X distributed experience prioritization

Example: Policy gradient



CartPole task: keep pole balanced on cart

1. Defining the policy network

policy def

```
network_out = FullyConnectedNetwork(obs, size=[64, 64])            # 2 outputs
action_distribution = CategoricalDistribution(network_out)          # e.g., P(LEFT) = 0.8, P(RIGHT) = 0.2
action_op = action_distribution.sample()                            # e.g., LEFT
```

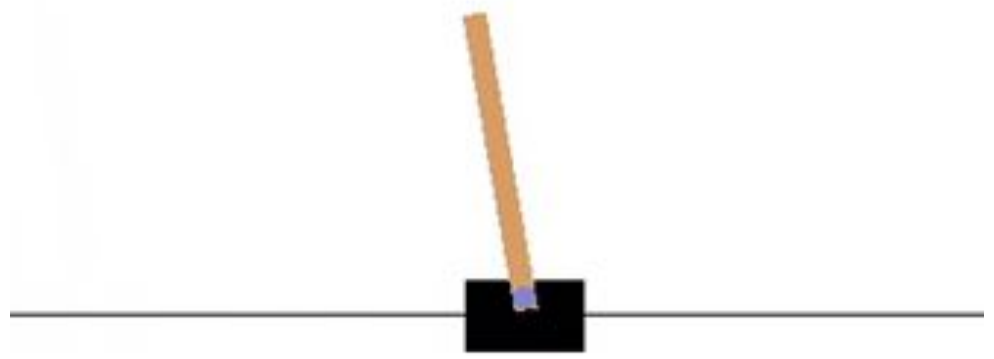
using policy

```
current_obs = env.reset()                                            # e.g., [1.2, -1.5]
action = session.run(action_op, feed_dict={obs: current_obs})        # returns LEFT or RIGHT
next_obs, reward, done = env.step(action)
```

sample data

```
experiences = [ ([1.2, -1.5], LEFT,  [1.1, -0.2], +1,  False),
                ([1.1, -0.2], RIGHT, [1.2, -0.8], +1,  False),
                ([1.2, -0.8], LEFT,  [1.1, -1.1], -10, True) ]   # batch of experiences
```

Example: Policy gradient



CartPole task: keep pole balanced on cart

2. Experience postprocessing

sample input

```
experiences_in = [ ([1.2, -1.5], LEFT,  [1.1, -0.2], +1,  False),
                   ([1.1, -0.2], RIGHT, [1.2, -0.8], +1,  False),
                   ([1.2, -0.8], LEFT,  [1.1, -1.1], -10, True) ]
```

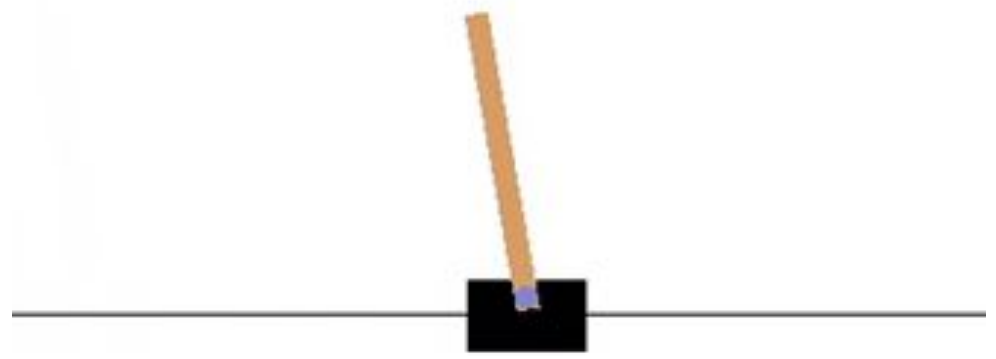
---

sample output

```
experiences_out = [ ([1.2, -1.5], LEFT,  [1.1, -0.2], -6.2, False),
                    ([1.1, -0.2], RIGHT, [1.2, -0.8], -8.0, False),
                    ([1.2, -0.8], LEFT,  [1.1, -1.1], -10,  True) ]
```

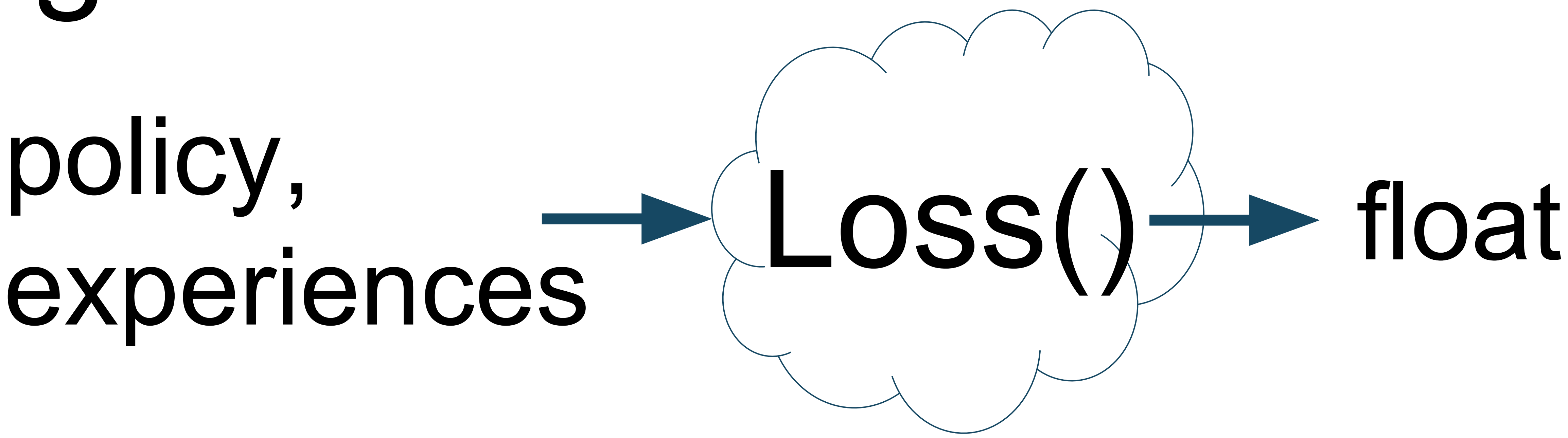
temporal discounting: propagate consequences of actions

Example: Policy gradient



CartPole task: keep pole balanced on cart

3. Defining the loss function




```
loss = -tf.reduce_mean(dist.logp(action) * advantages)
train_op = tf.train.GradientDescentOptimizer.minimize(loss)
```

Parallel Policy Gradient with RLlib

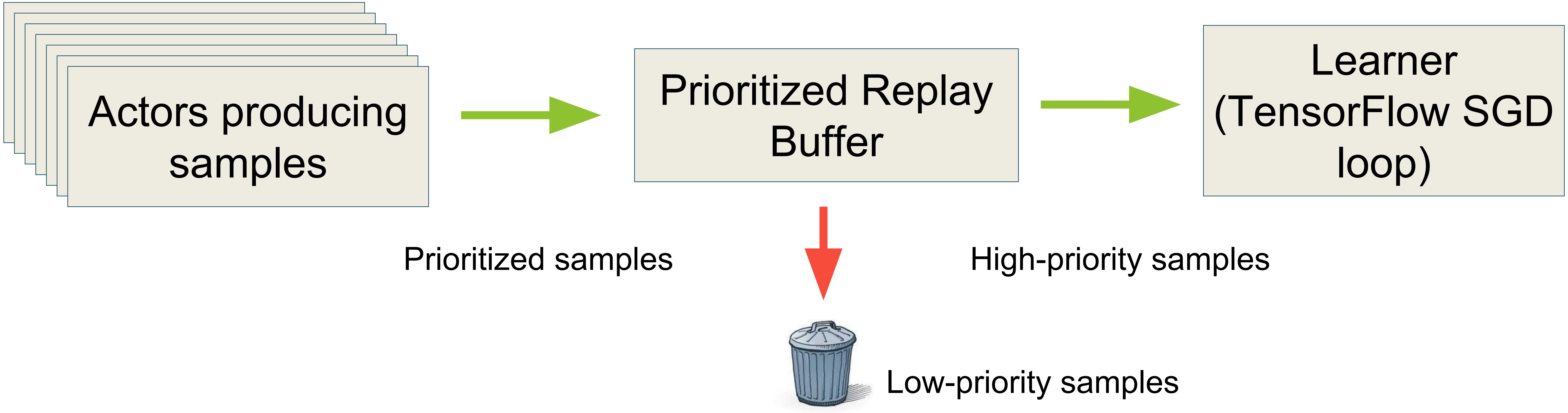
```python
class PolicyGradientGraph(rllib.TFPolicyGraph):
    def __init__(self, obs_space, action_space):
        self.obs, self.adv = tf.placeholder(), tf.placeholder()
        model = FullyConnectedNetwork(self.obs, size=[64, 64])
        dist = rllib.action_distribution(action_space, model)
        self.act = dist.sample()
        self.loss = -tf.reduce_mean(dist.logp(self.act) * self.adv)

    def postprocess(self, batch):
        return rllib.compute_advantages(batch)
```

Parallel Policy Gradient with RLlib

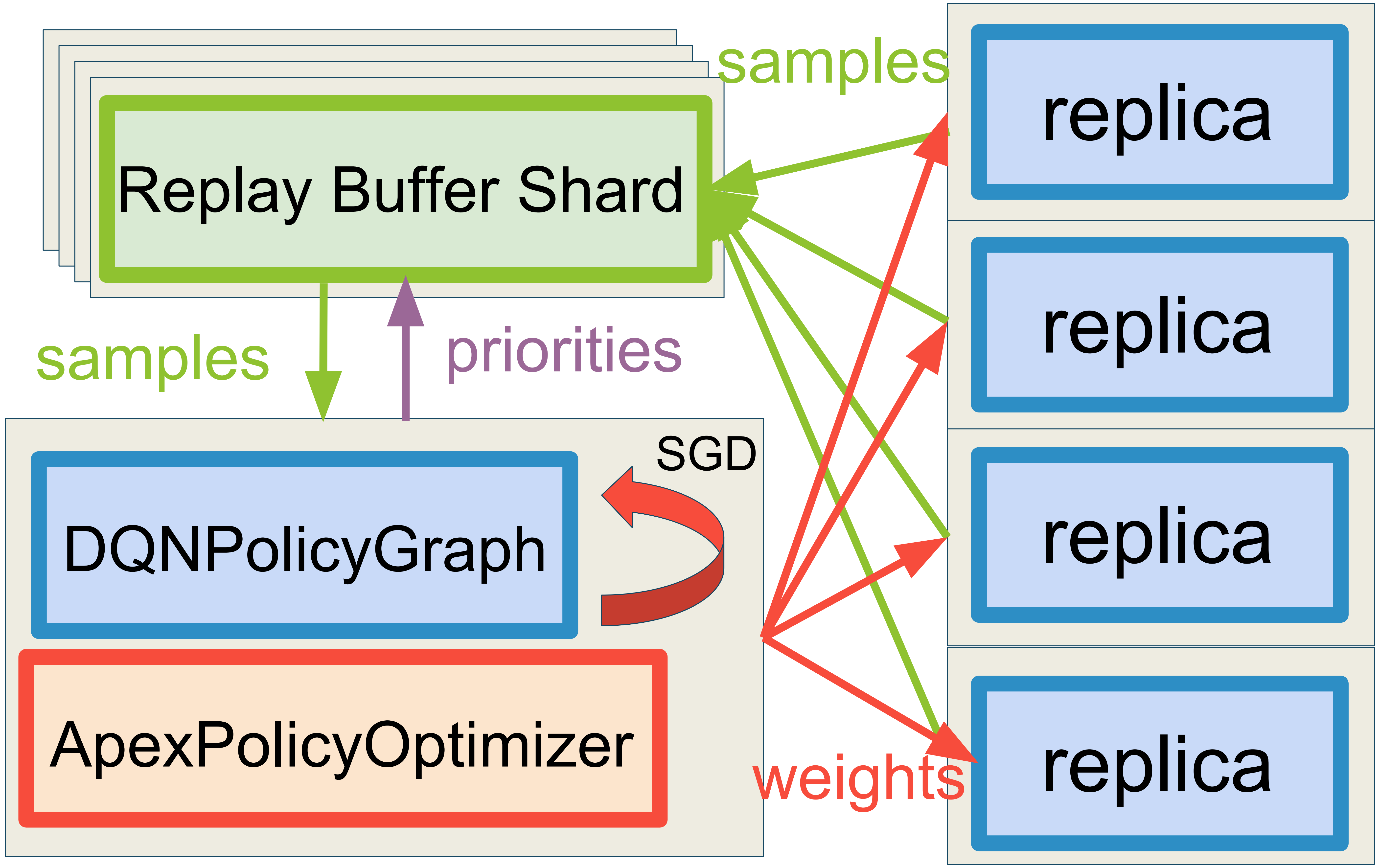
```python
# Setup distributed workers
workers = [rllib.PolicyEvaluator.remote(
    env="CartPole-v0", policy_graph=PolicyGradientGraph)
    for _ in range(10)]

# Choose policy optimizer
optimizer = rllib.AsyncPolicyOptimizer(workers)

# Training loop
while True:
    optimizer.step()
    print(optimizer.foreach_policy(lambda p: p.get_stats()))
```

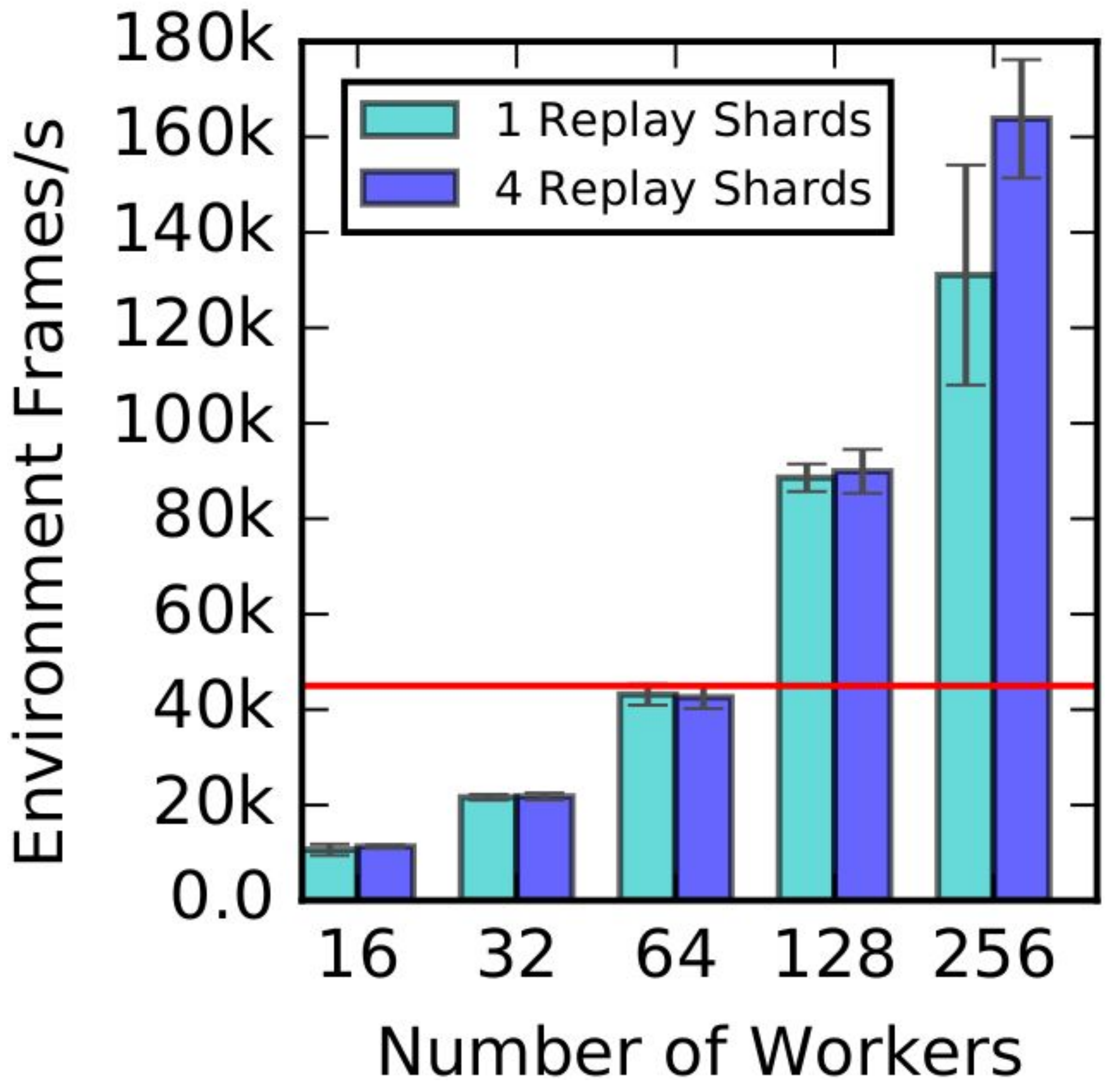
Example: Ape-X distributed DQN

Basic idea: prioritize important experiences



Example: Ape-X distributed DQN





<200 lines of Python

RLlib is a scalable framework for reinforcement learning

We're continuing to improve RLlib

Find us at github.com/ray-project/ray

Thank you!

{ekl, rliaw}@berkeley.edu

Performance: single-node

Policy graph abstraction => automatic optimizations

– Vectorization of policy execution (including support for sparse vector envs, e.g., ELF)
– Execution of multiple agents and policies can be fused together into one neural network evaluation

High-performance data exchange between processes

– Shared-memory object store between Ray actors and tasks
– Compress experience batches with LZ4 (~1GB/s/core)
– Column batch format for fast processing of experiences

Performance: distributed

Choice of policy optimizers for distributed execution

– Take advantage of differing hardware configurations (e.g., availability of GPUs, CPU vs GPU balance, large clusters)
– Easy to experiment with novel distributed algorithms

Leverage Ray's unified parallel and distributed execution:

– Lightweight tasks "spill over" to multiple nodes
– Asynchronous tasks enable pipelining of computation
– Object store enables efficient data transfers between actors bypassing the driver

How general is this formulation?

- Can define basic alg. with $\pi_\theta(o_t)$, $\rho_\theta(X)$, $L(\theta, X)$
- $\pi_\theta(o_t, h_t) \Rightarrow (a_t, h_{t+1}, y^1_t \ldots y^n_t)$ → **Recurrent policies, actor-critic methods**
- $\rho_\theta(X_{pre}, X_{pre}^{1\ldots k}) \Rightarrow X_{post}$ → **Multi-agent, Hindsight Experience Replay**
- $u^{1..m}(\theta) \Rightarrow (msg, \theta_{update})$ → **DQNs, distributed prioritization, model-based/hybrid algs (e.g. AlphaZero)**

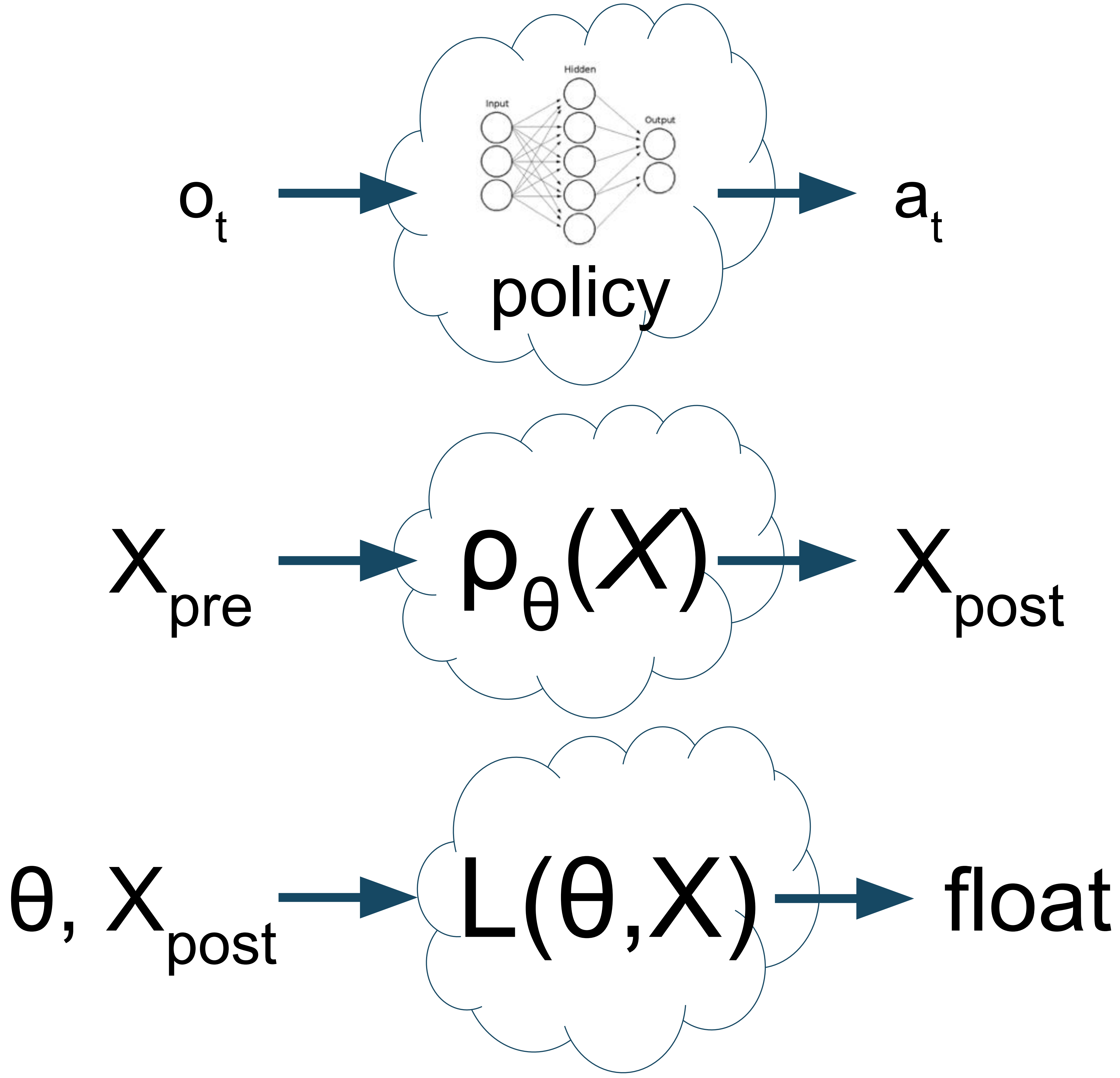
RLlib abstractions for algorithms

1. Policy $\pi_\theta(o_t) \Rightarrow a_t$

2. Experience postprocessing

   X = batch of (o_t, a_t, r_t, o_{t+1}) tuples

3. Loss function: improve π

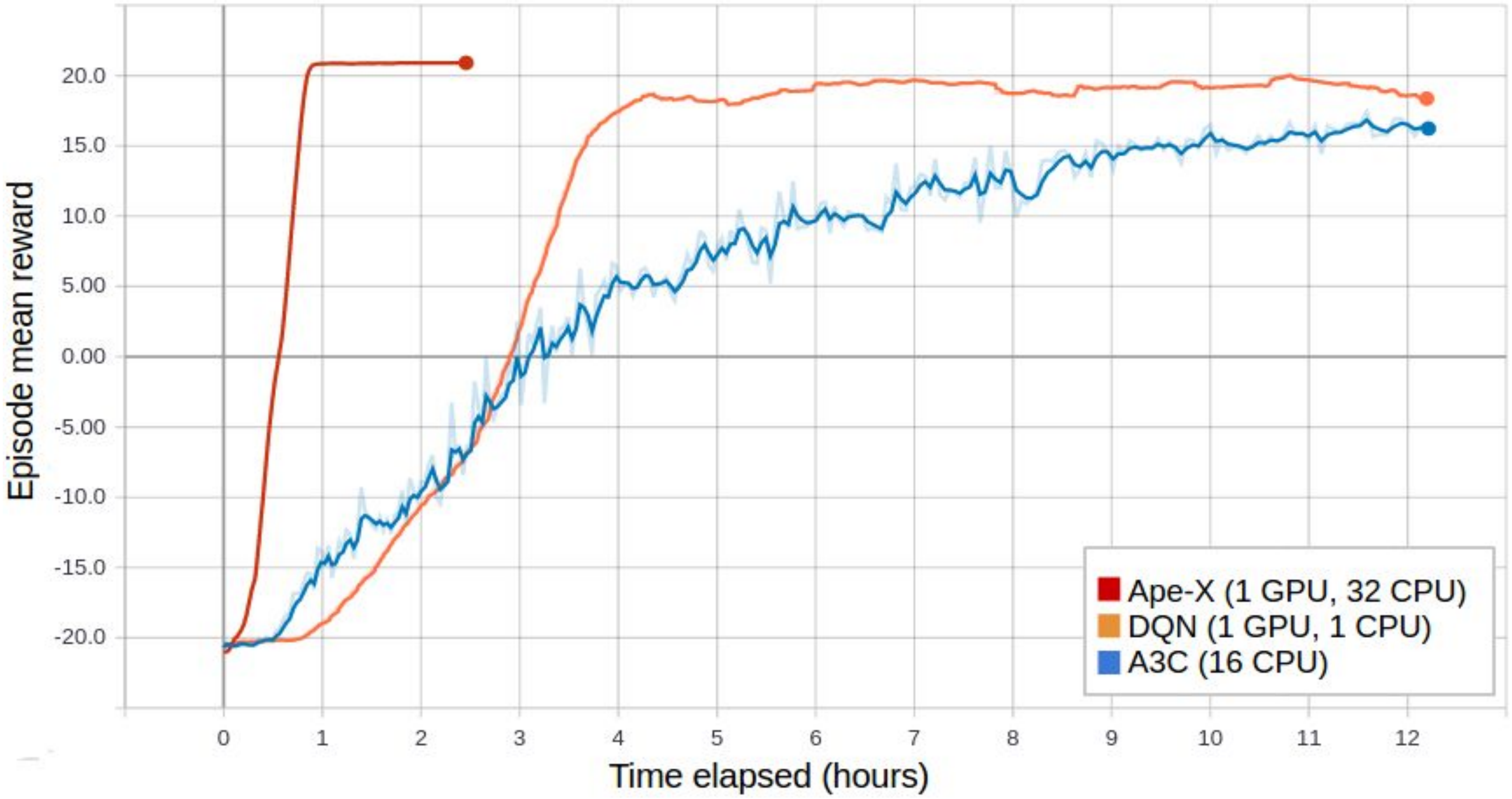


Case study: Ape-X distributed DQN

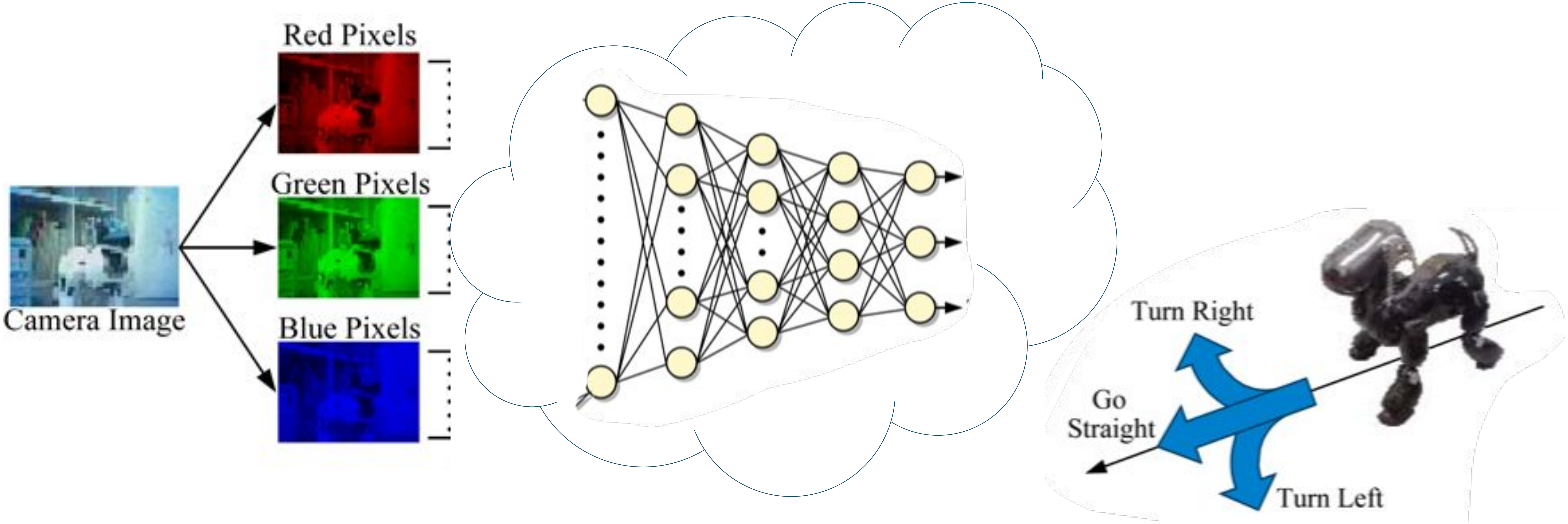
- Base algorithm: DQN
- $\rho_\theta(X) = \text{adjust\_step}(X) + \text{td\_error}(\theta, X)$
- $\pi_\theta(s) = \text{argmax}_a\, Q(s, a)$
- $u^1(\theta) = \text{assign}(\theta_{q\_target}, \theta_q)$

Compute Q error on workers ("distributed prioritization")

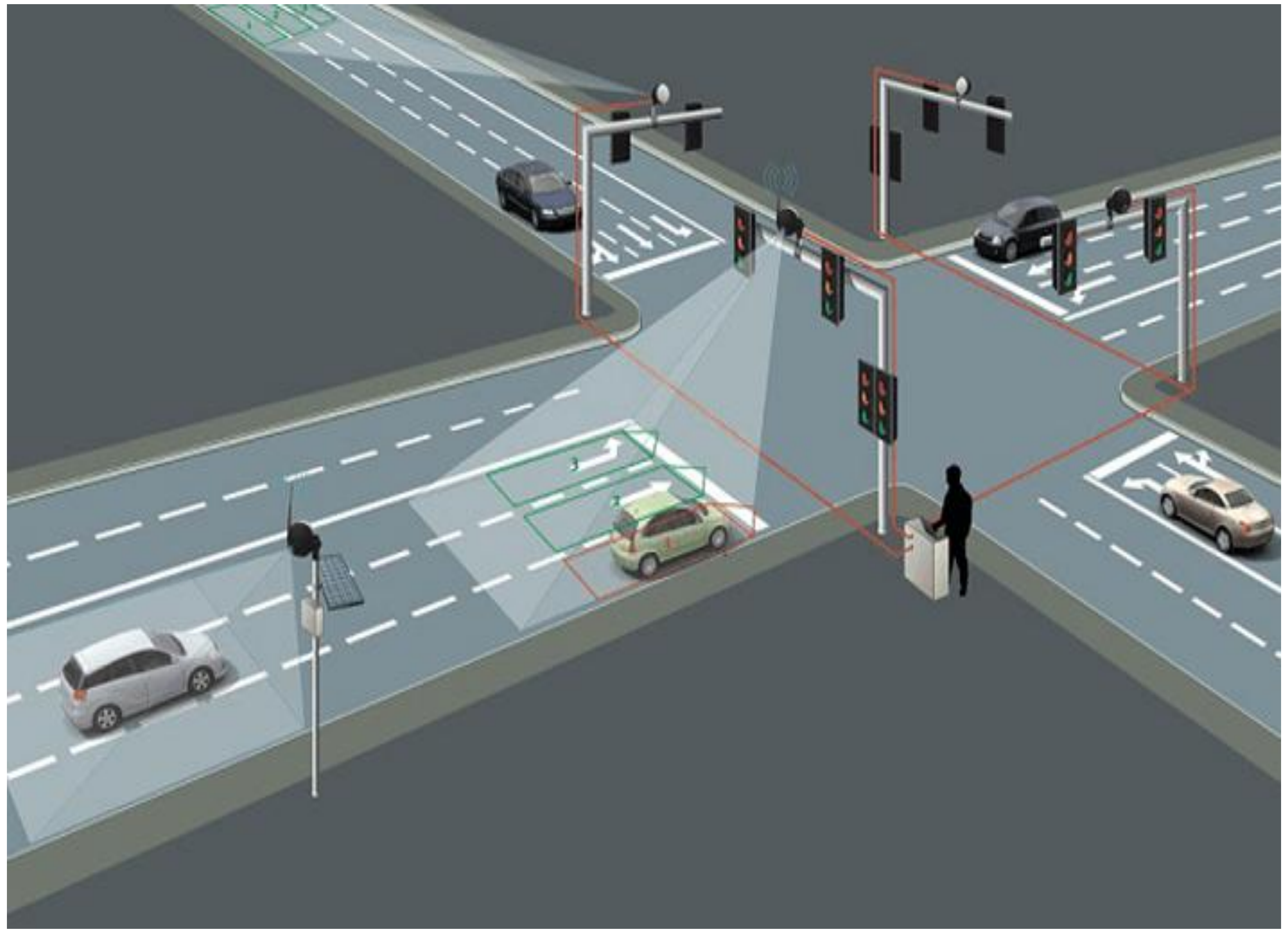
Case study: Ape-X distributed DQN



Deep Reinforcement Learning



Gathering more data



Simulation-based Learning