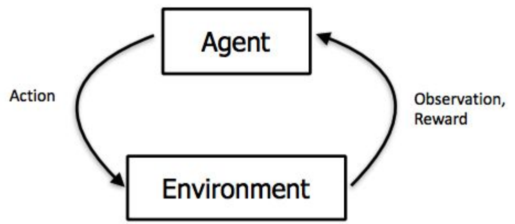
Distributed RL

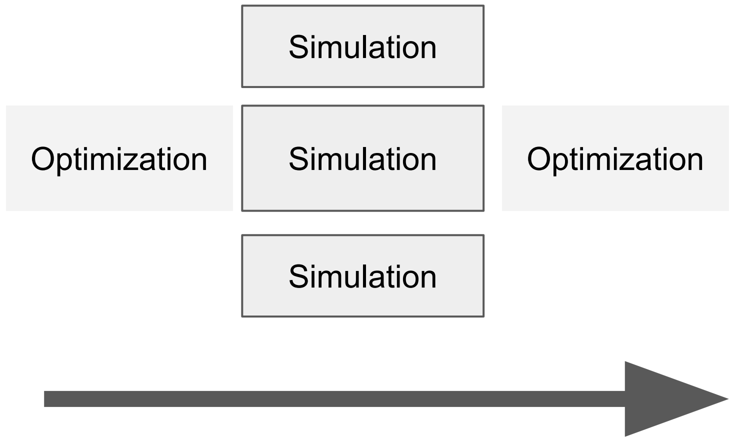
Richard Liaw, Eric Liang

Common Computational Patterns for RL

Original



Batch Optimization



How can we **better utilize** our computational resources **to accelerate** RL progress?

History of large scale distributed RL

2013 — 2015 — 2016 — 2018 — 2018 — ?

DQN

Playing Atari with Deep Reinforcement Learning (Mnih 2013)

GORILA

Massively Parallel Methods for Deep Reinforcement Learning (Nair 2015)

A3C

Asynchronous Methods for Deep Reinforcement Learning (Mnih 2016)

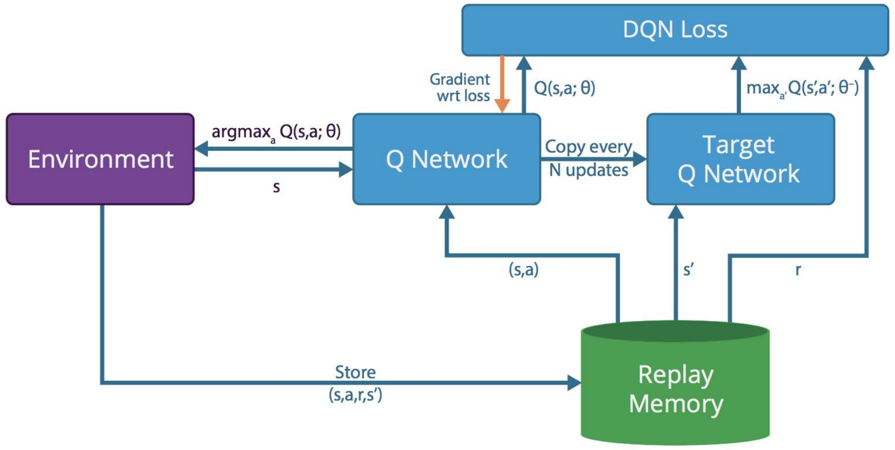
Ape-X

Distributed Prioritized Experience Replay (Horgan 2018)

IMPALA

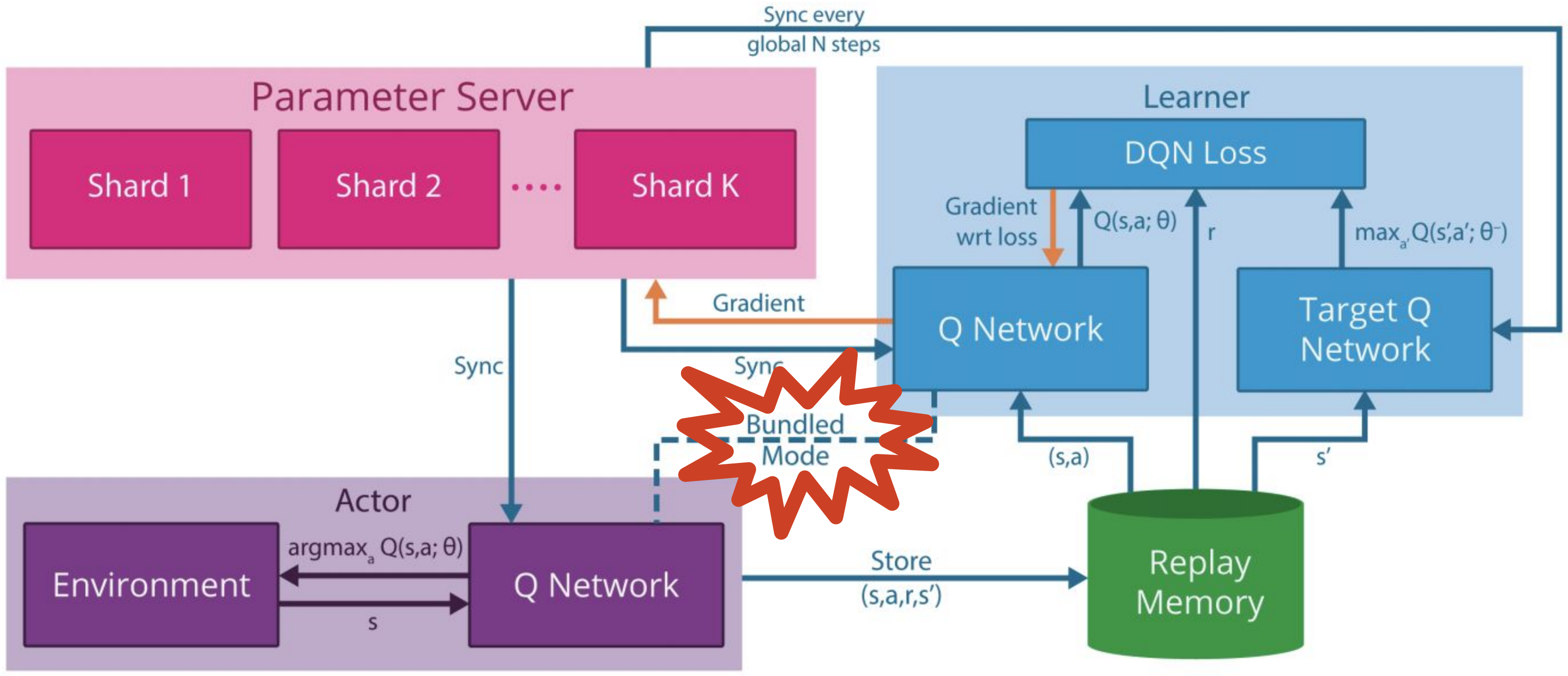
IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures (Espeholt 2018)

2013/2015: DQN

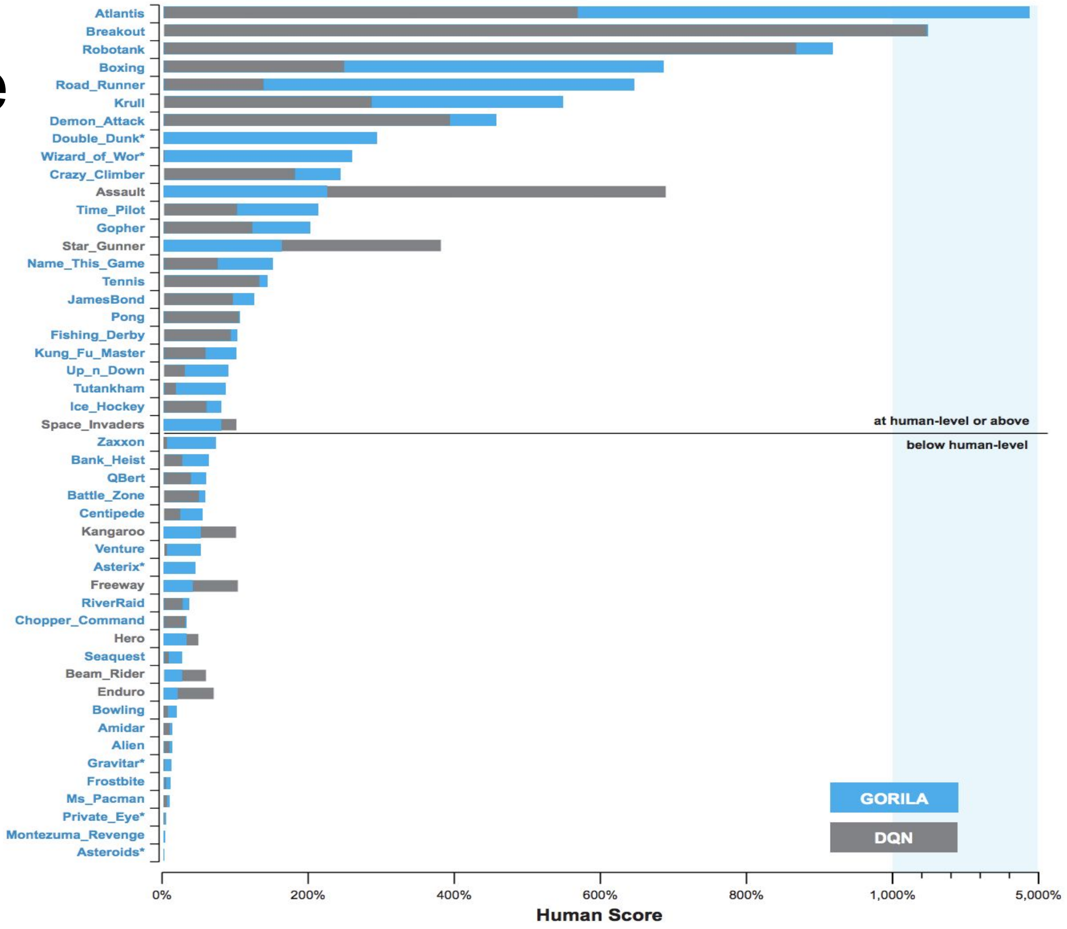


```
for i in range(T):
    s, a, s_1, r = evaluate()
    replay.store((s, a, s_1, r))

    minibatch = replay.sample()
    q_network.update(mini_batch)

    if should_update_target():
        q_network.sync_with(target_net)
```

2015: General Reinforcement Learning Architecture (GORILA)

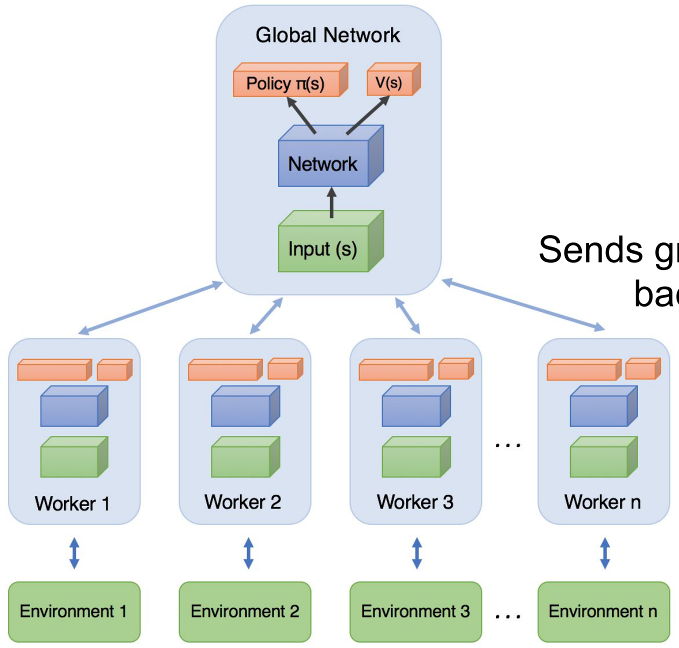


GORILA Performance



2016: Asynchronous Advantage Actor Critic (A3C)

```
# Each worker:

while True:
    sync_weights_from_master()

    for i in range(5):
        collect sample from env

    grad = compute_grad(samples)
    async_send_grad_to_master()
```



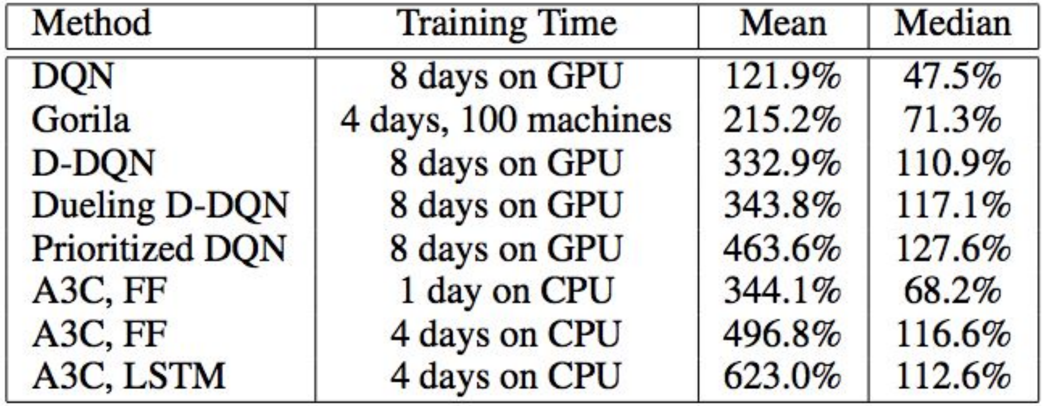
Sends gradients back

Each has different exploration -> more diverse samples!

A3C Performance

Changes to GORILA:

1. **Faster updates**
2. **Removes** the replay buffer
3. **Moves** to Actor-Critic (from Q learning)



Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

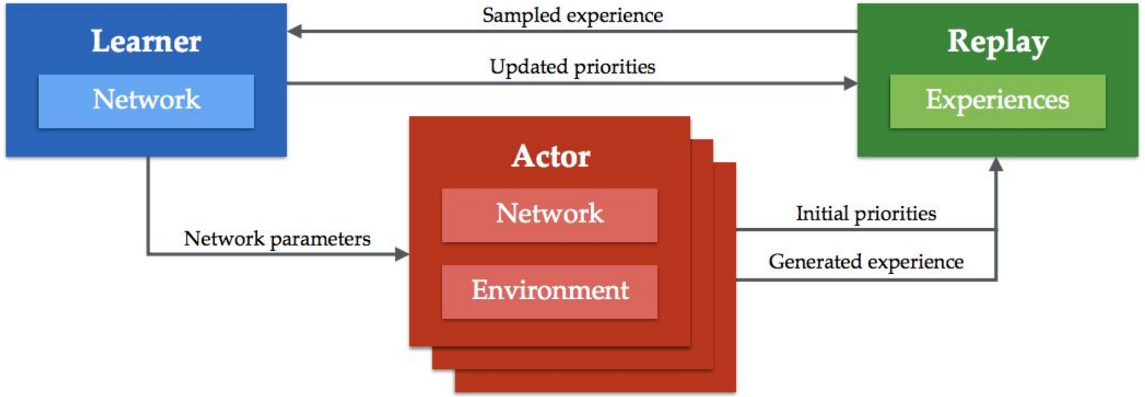
Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary

Distributed Prioritized Experience Replay (Ape-X)

A3C doesn't scale very well…

Ape-X:

1. Distributed DQN/DDPG
2. Reintroduces replay
3. **Distributed Prioritization:** Unlike Prioritized DQN, initial priorities are not set to "max TD"



Ape-X Performance

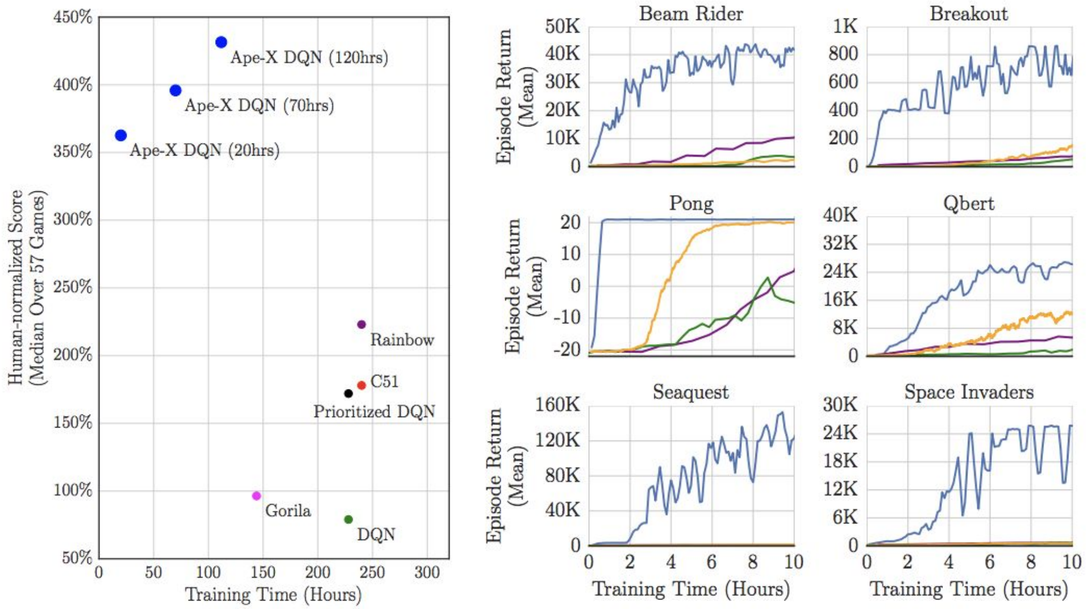
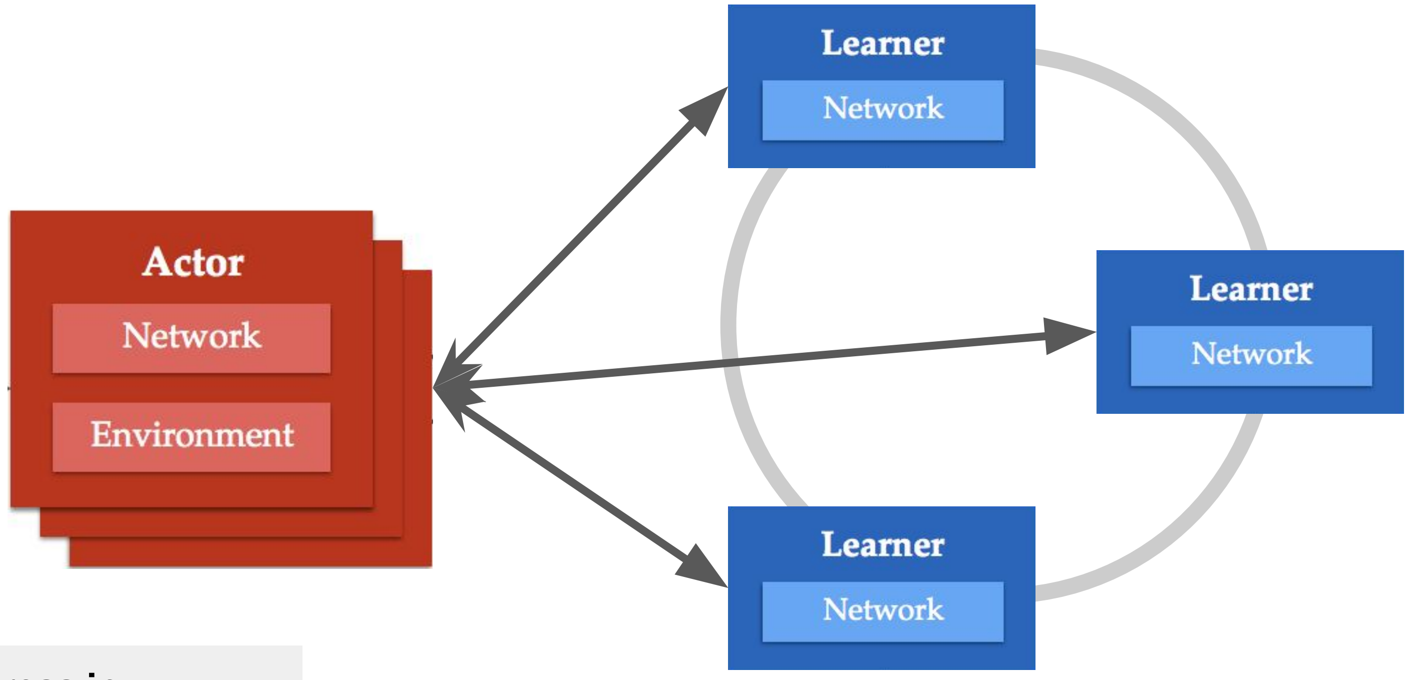


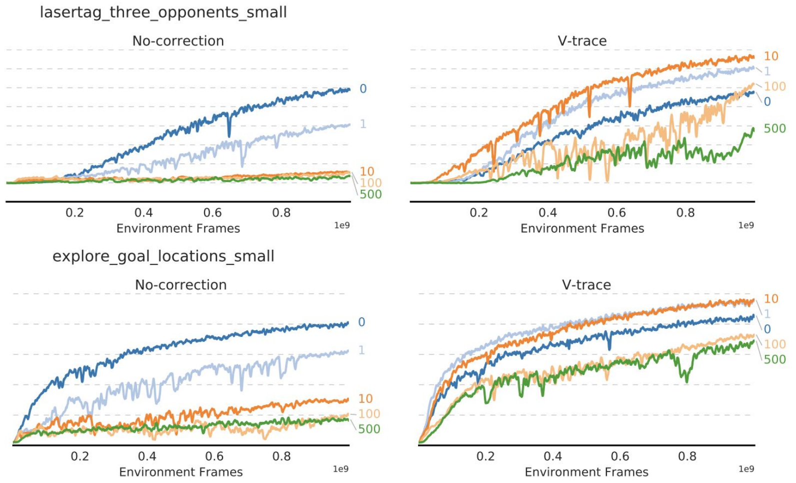
Figure 2: Left: Atari results aggregated across 57 games, evaluated from random no-op starts. Right: Atari training curves for selected games, against baselines. Blue: Ape-X DQN with 360 actors; Orange: A3C; Purple: Rainbow; Green: DQN. See appendix for longer runs over all games.

Importance Weighted Actor-Learner Architectures (IMPALA)



Motivated by progress in distributed deep learning!

How to correct for Policy Lag? Importance Sampling!

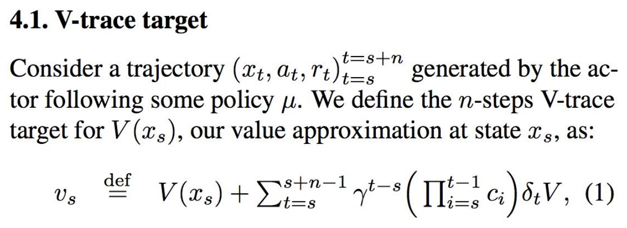


Given an actor-critic model:

1. Apply importance-sampling to policy gradient

$$\mathbb{E}_{a_s \sim \mu(\cdot|x_s)} \left[\frac{\pi_{\bar{\rho}}(a_s|x_s)}{\mu(a_s|x_s)} \nabla \log \pi_{\bar{\rho}}(a_s|x_s) q_s \middle| x_s \right]$$

2. Apply importance sampling to critic update

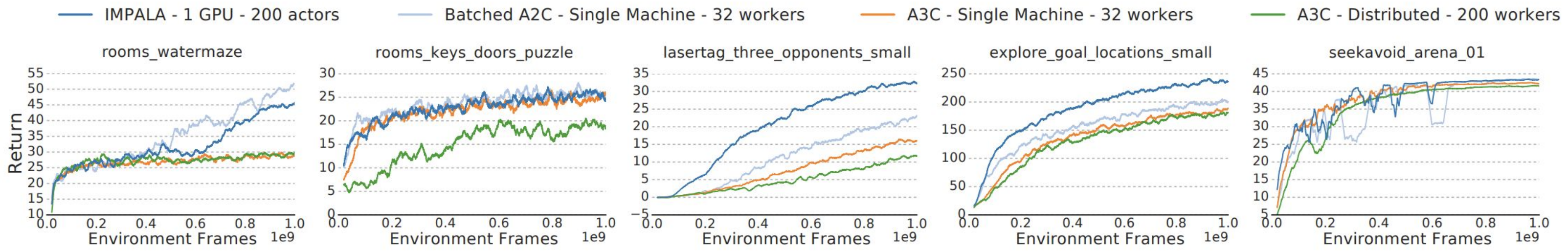


4.1. V-trace target

Consider a trajectory $(x_t, a_t, r_t)_{t=s}^{t=s+n}$ generated by the actor following some policy μ. We define the n-steps V-trace target for $V(x_s)$, our value approximation at state x_s, as:

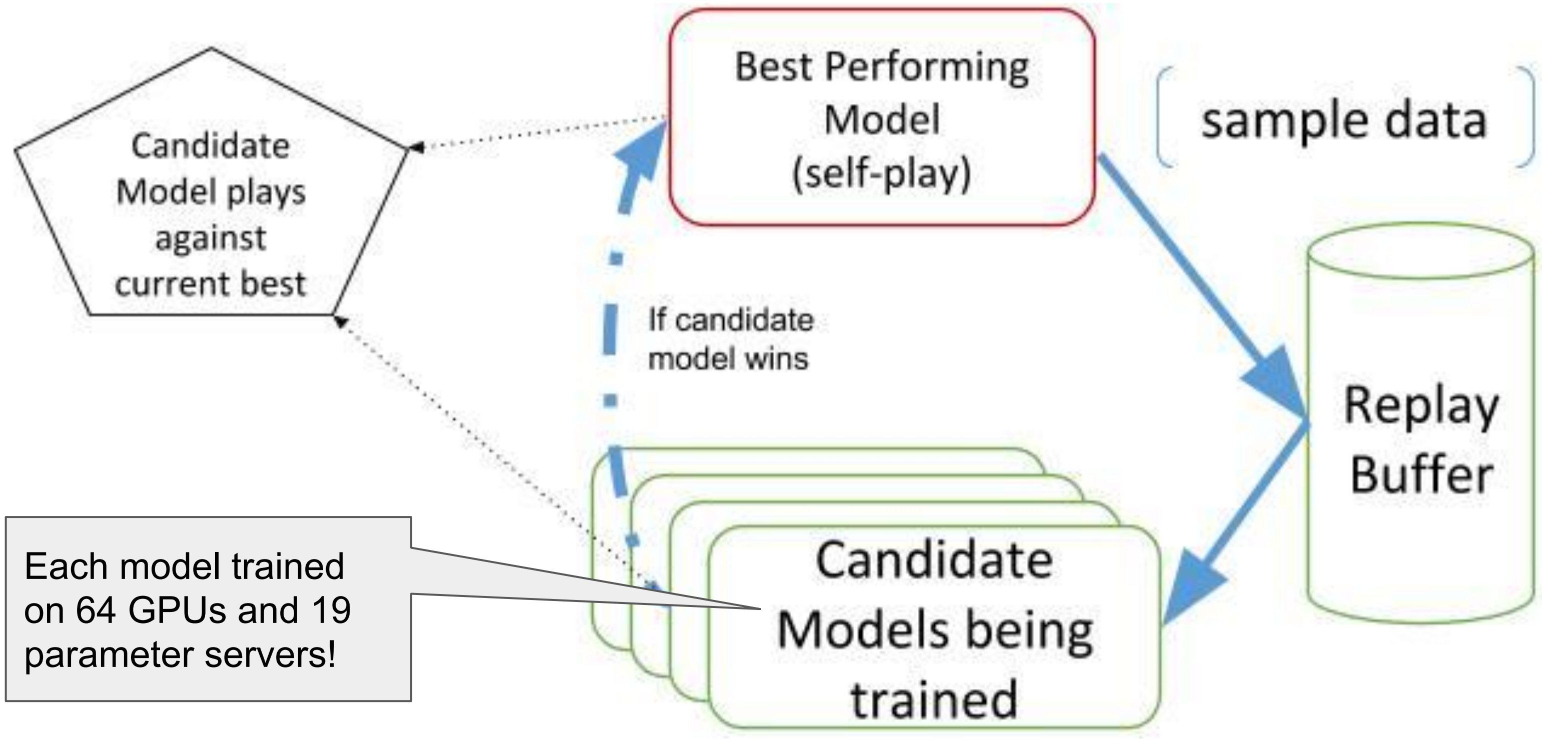
$$v_s \overset{\text{def}}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \Big(\prod_{i=s}^{t-1} c_i \Big) \delta_t V, \quad (1)$$

IMPALA Performance



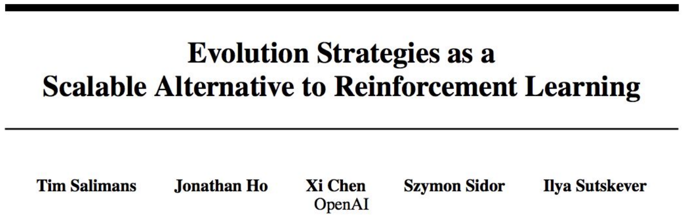
Other interesting distributed architectures

AlphaZero


Candidate Model plays against current best
Best Performing Model (self-play)
sample data
If candidate model wins
Replay Buffer
Candidate Models being trained

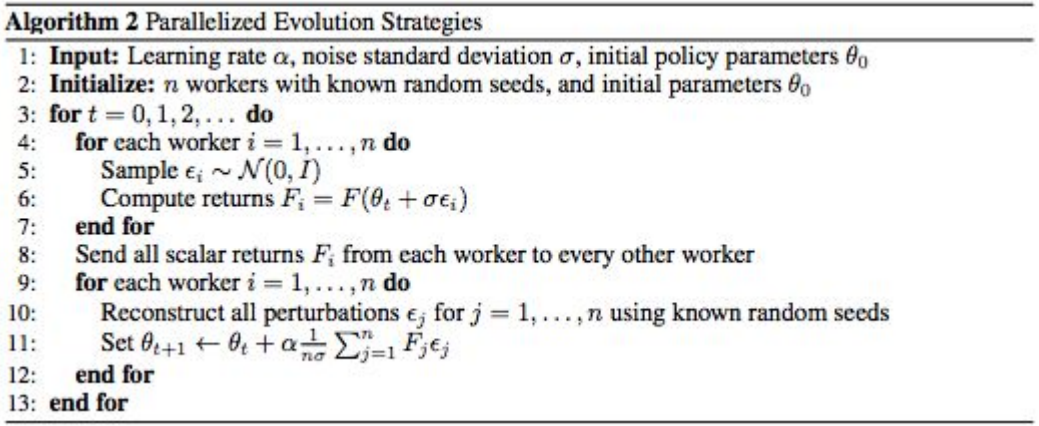
Each model trained on 64 GPUs and 19 parameter servers!

Evolution Strategies



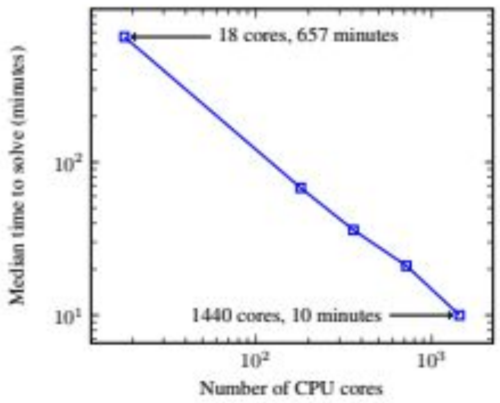
Evolution Strategies as a
Scalable Alternative to Reinforcement Learning

Tim Salimans **Jonathan Ho** **Xi Chen** **Szymon Sidor** **Ilya Sutskever**
OpenAI



Algorithm 2 Parallelized Evolution Strategies

```
 1: Input: Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
 2: Initialize: $n$ workers with known random seeds, and initial parameters $\theta_0$
 3: for $t = 0, 1, 2, \dots$ do
 4:   for each worker $i = 1, \dots, n$ do
 5:     Sample $\epsilon_i \sim \mathcal{N}(0, I)$
 6:     Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
 7:   end for
 8:   Send all scalar returns $F_i$ from each worker to every other worker
 9:   for each worker $i = 1, \dots, n$ do
10:     Reconstruct all perturbations $\epsilon_j$ for $j = 1, \dots, n$ using known random seeds
11:     Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
12:   end for
13: end for
```



RLlib: Abstractions for Distributed Reinforcement Learning (ICML'18)

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

RL research scales with compute

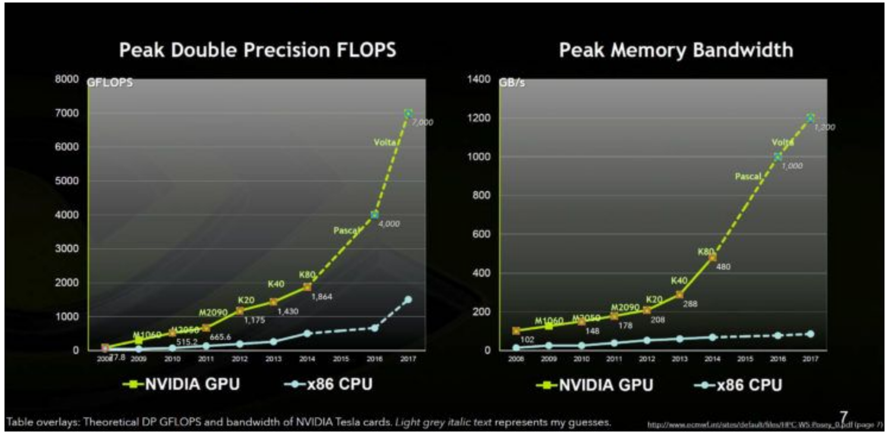


Fig. courtesy NVidia Inc.

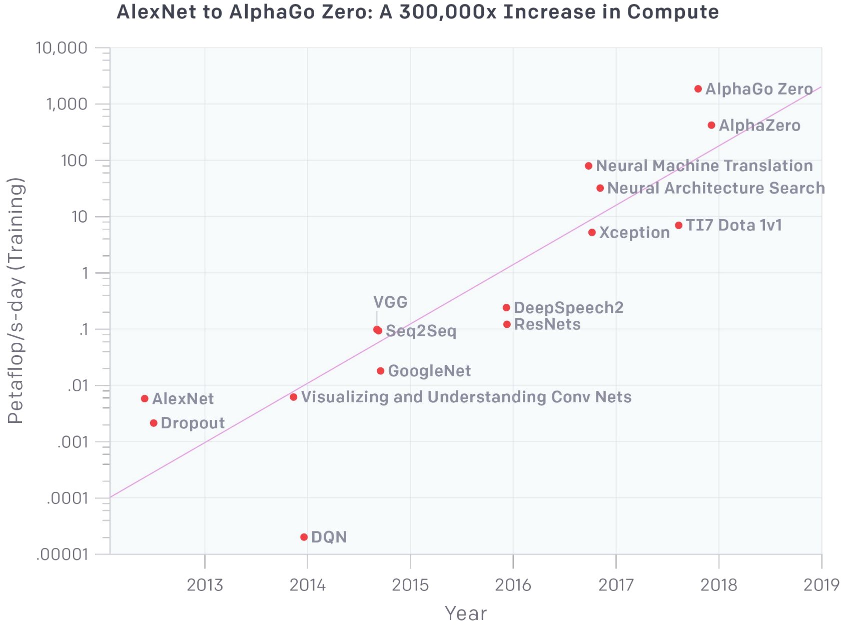


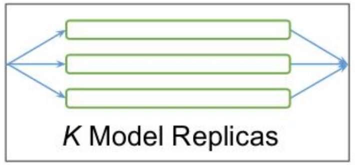
Fig. courtesy OpenAI

CPU

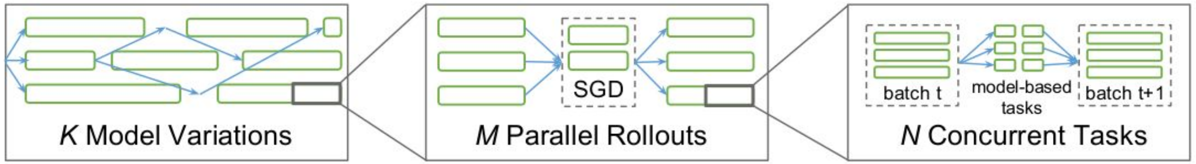
GPU

TPU

How do we leverage this hardware?



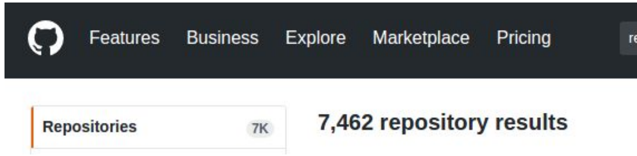
(a) Supervised Learning



(b) Reinforcement Learning

scalable abstractions for RL?

Systems for RL today



7,462 repository results

- Many implementations (7000+ repos on GitHub!)
  - how general are they (and do they scale)?

    PPO: multiprocessing, MPI

    AlphaZero: custom systems

    Evolution Strategies: Redis

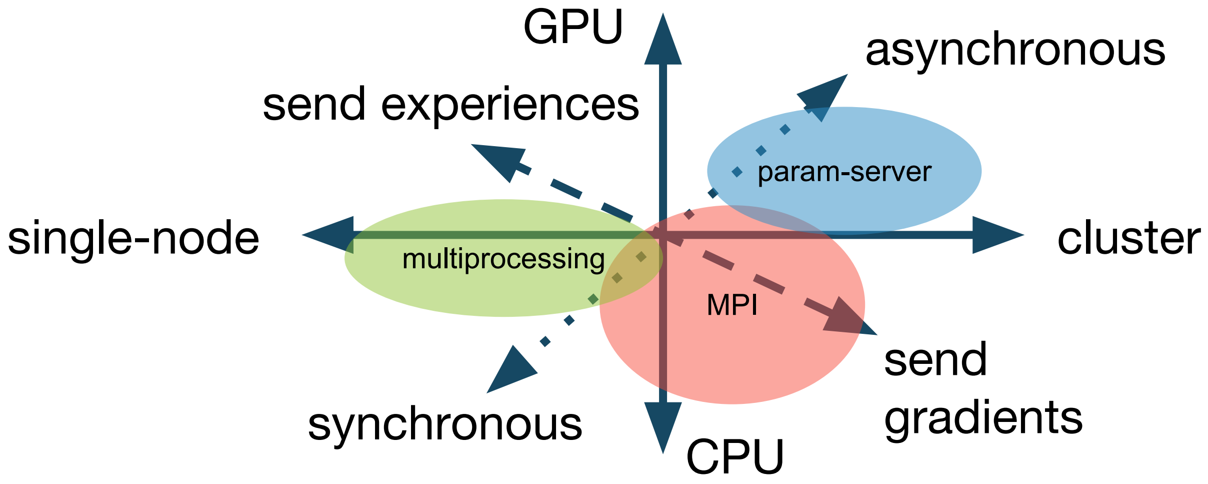
    IMPALA: Distributed TensorFlow

    A3C: shared memory, multiprocessing, TF

- Huge variety of algorithms and distributed systems used to implement, but little reuse of components

Challenges to reuse

1. Wide range of physical execution strategies for one "algorithm"



Challenges to reuse

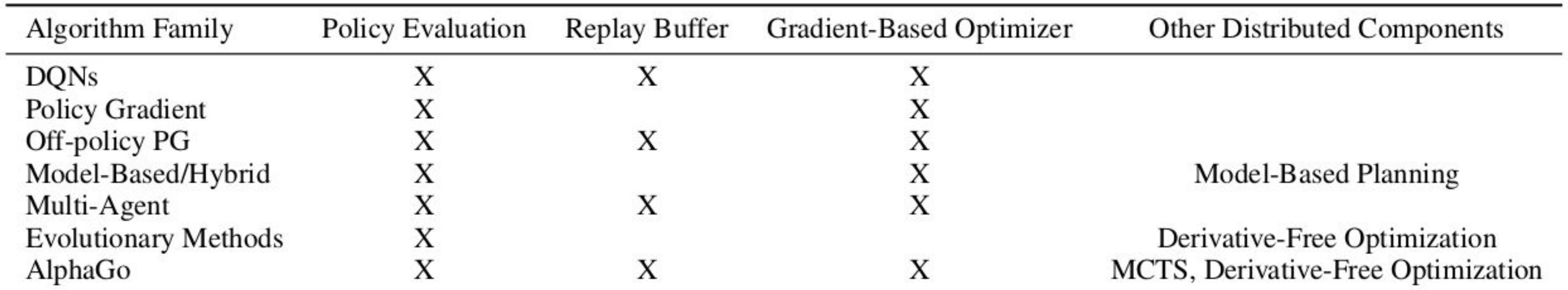
2. Tight coupling with deep learning frameworks

Different parallelism paradigms:

– Distributed TensorFlow vs TensorFlow + MPI?

Challenges to reuse

3. Large variety of algorithms with different structures



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

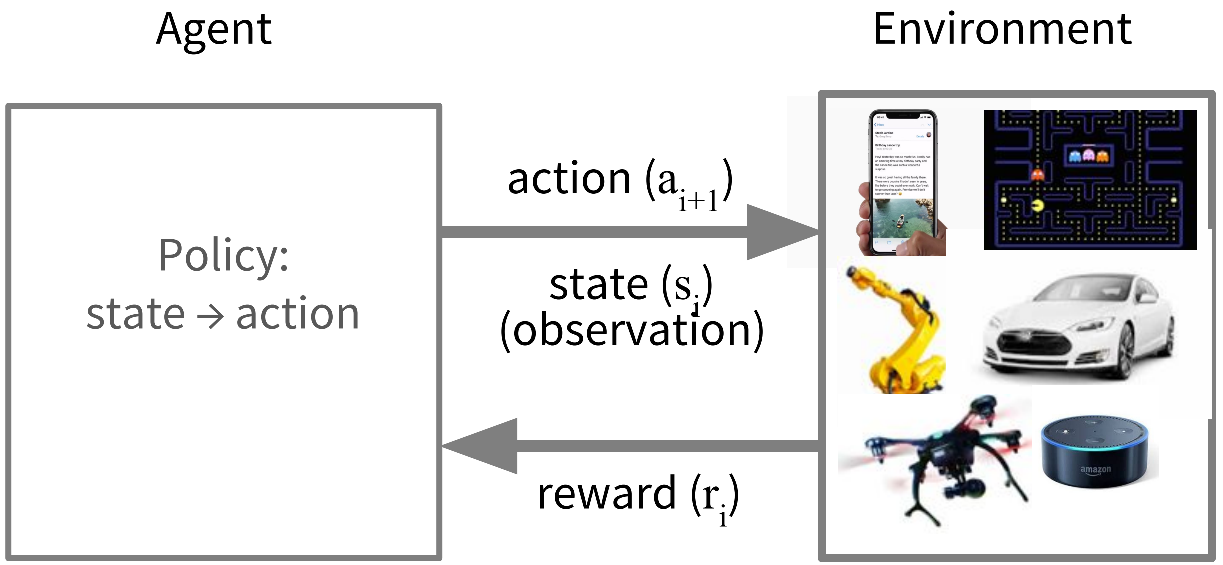
We need abstractions for RL

Good abstractions decompose RL algorithms into reusable components.

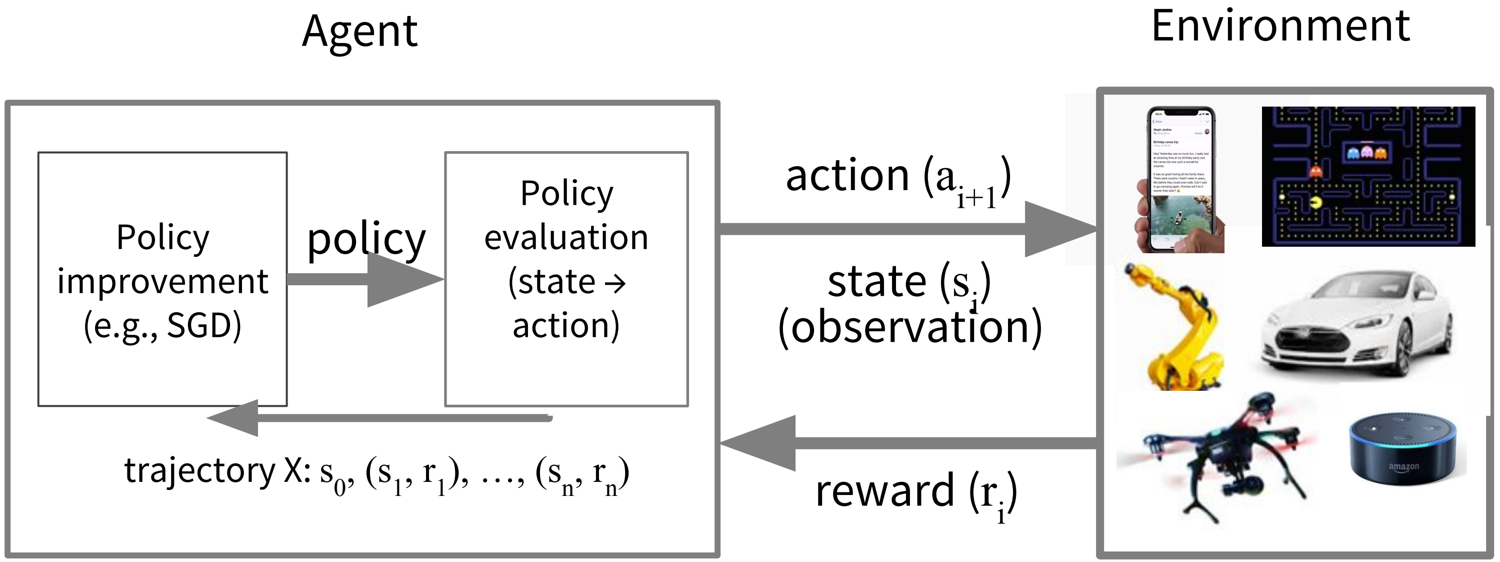
Goals:

- Code reuse across deep learning frameworks
- Scalable execution of algorithms
- Easily compare and reproduce algorithms

Structure of RL computations

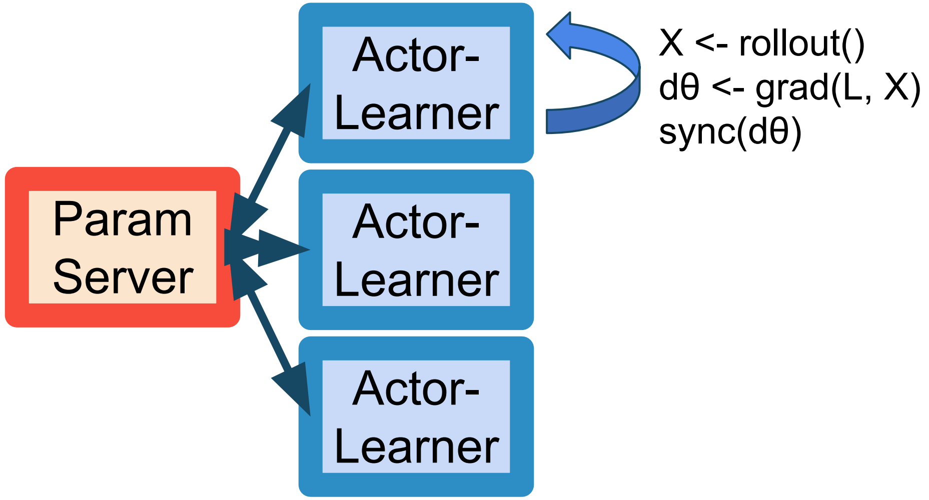


Structure of RL computations

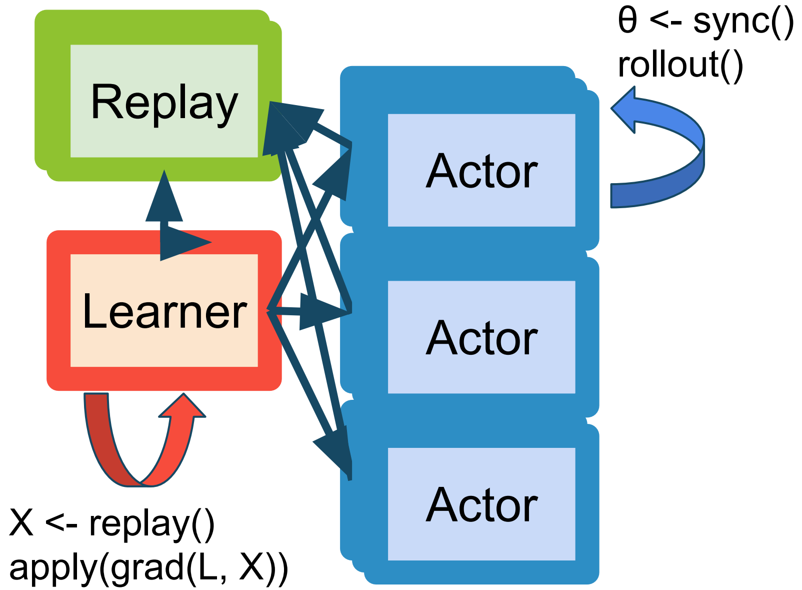

Agent
Environment
Policy improvement (e.g., SGD)
policy
Policy evaluation (state → action)
trajectory X: s_0, (s_1, r_1), …, (s_n, r_n)
action (a_{i+1})
state (s_i) (observation)
reward (r_i)

Many RL loop decompositions

Async DQN (Mnih et al; 2016)

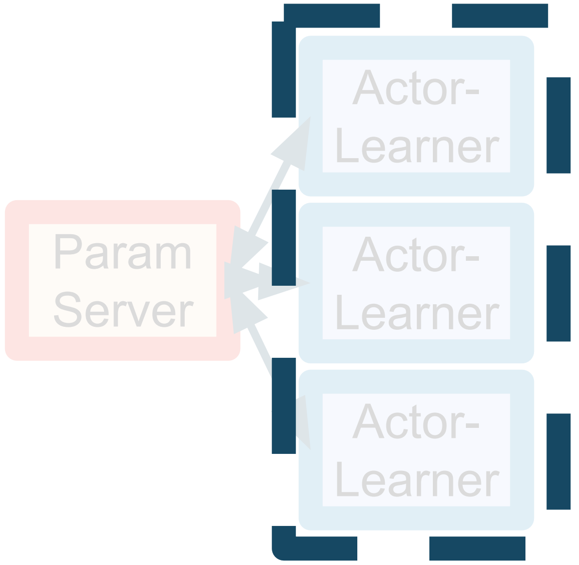


Ape-X DQN (Horgan et al; 2018)

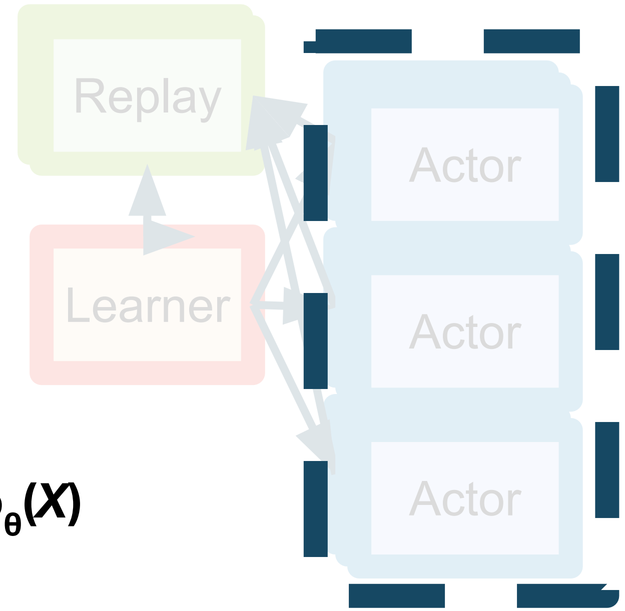


Common components

Async DQN (Mnih et al; 2016)



Ape-X DQN (Horgan et al; 2018)



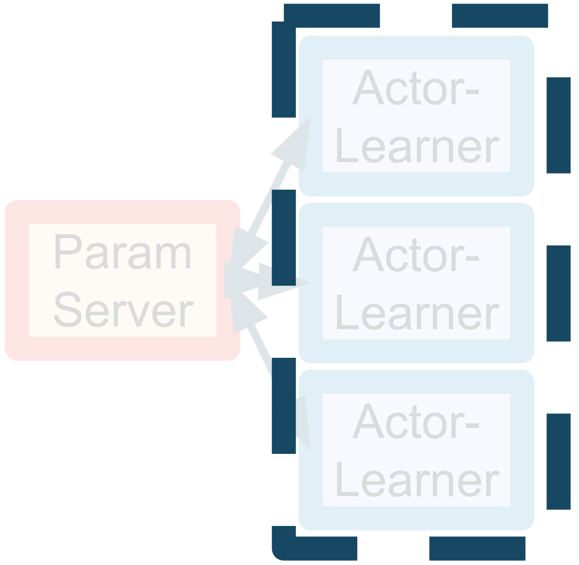
Policy $\pi_\theta(o_t)$

Trajectory postprocessor $\rho_\theta(X)$

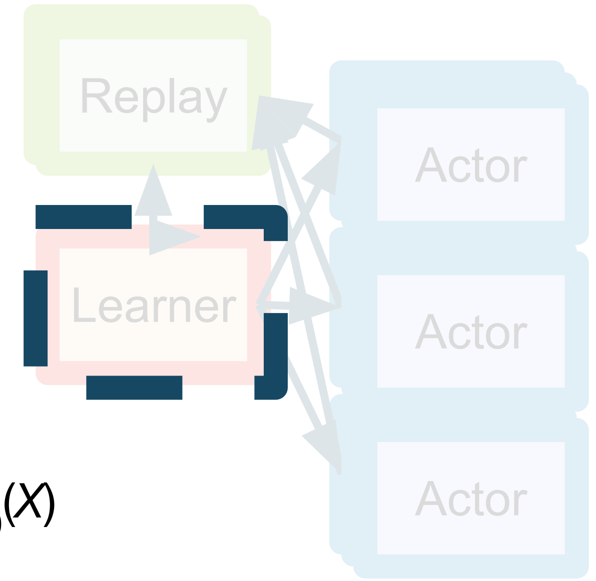
Loss $L(\theta,X)$

Common components

Async DQN (Mnih et al; 2016)



Ape-X DQN (Horgan et al; 2018)



Policy $\pi_\theta(o_t)$

Trajectory postprocessor $\rho_\theta(X)$

Loss $L(\theta,X)$

Structural differences

Async DQN (Mnih et al; 2016)

- Asynchronous optimization
- Replicated workers
- Single machine

Ape-X DQN (Horgan et al; 2018)

- Central learner
- Data queues between components
- Large replay buffers
- Scales to clusters

+ Population-Based Training (Jaderberg et al; 2017)

- Nested parallel computations
- Control decisions based on intermediate results

...and this is just one family!

→ No existing system can effectively meet all the varied demands of RL workloads.

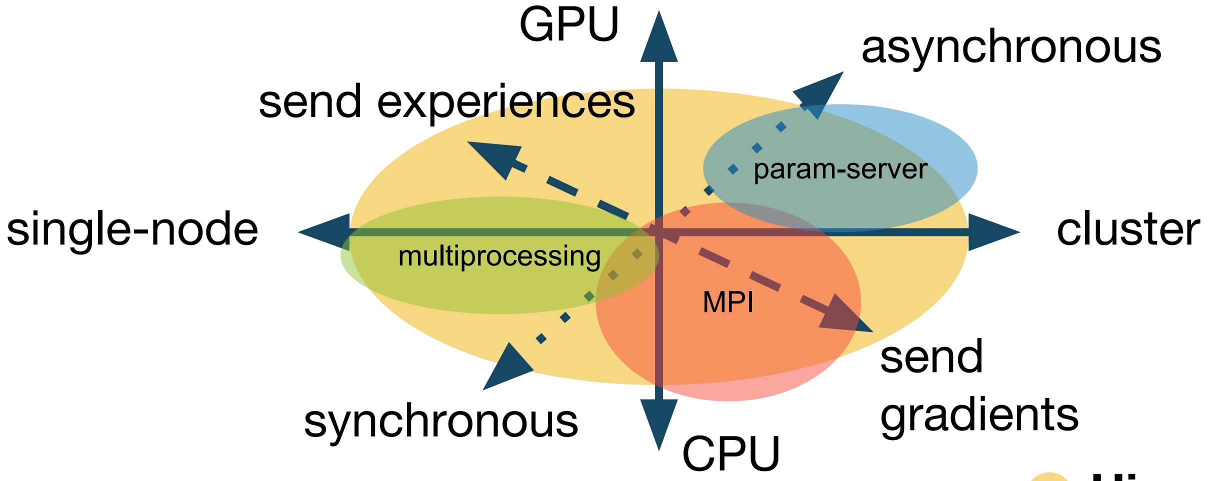
Requirements for a new system

Goal: Capture a broad range of RL workloads with high performance and substantial code reuse

1. Support stateful computations
    - e.g., simulators, neural nets, replay buffers
    - big data frameworks, e.g., Spark, are typically stateless
2. Support asynchrony
    - difficult to express in MPI, esp. nested parallelism
3. Allow easy composition of (distributed) components

Ray System Substrate

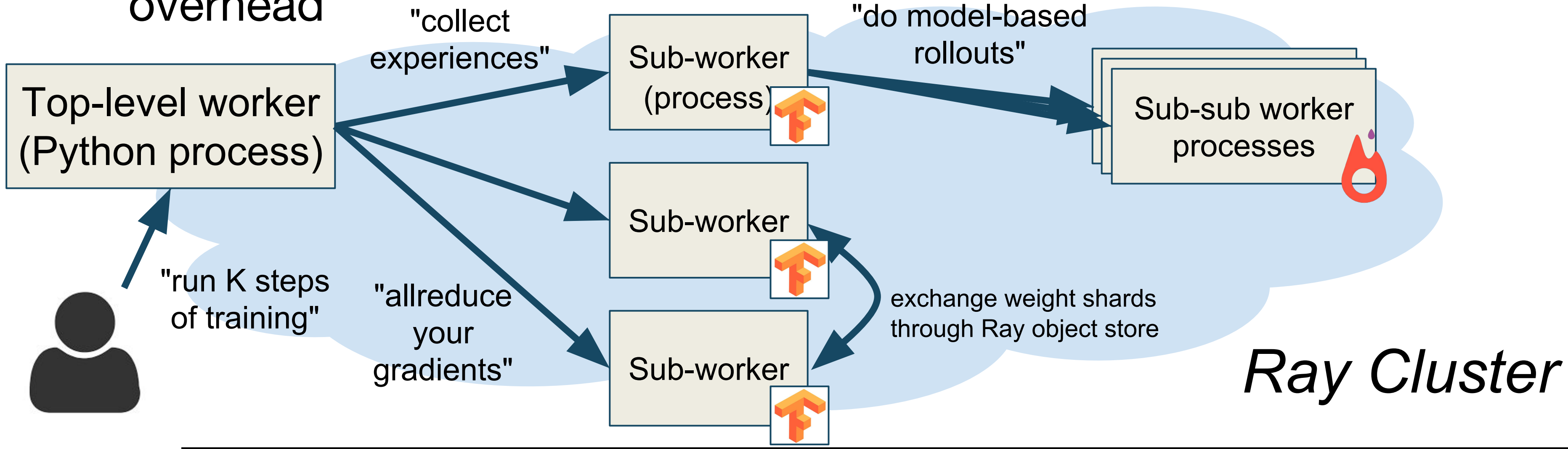
- RLlib builds on Ray to provide higher-level RL abstractions
- Hierarchical parallel task model with stateful workers
  - flexible enough to capture a broad range of RL workloads (vs specialized sys.)



● **Hierarchical Task Model**

Hierarchical Parallel Task Model

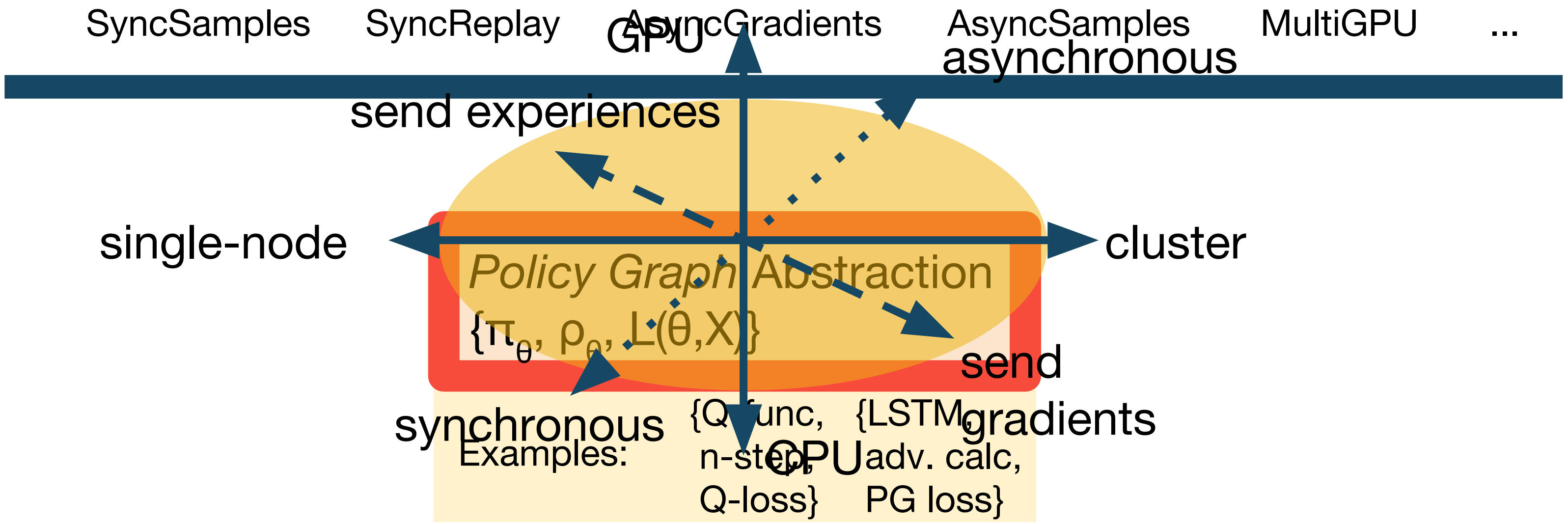
1. Create Python class instances in the cluster (stateful workers)
2. Schedule short-running tasks onto workers
   - Challenge: High performance: 1e6+ tasks/s, ~200us task overhead



Top-level worker (Python process)
"collect experiences"
Sub-worker (process)
"do model-based rollouts"
Sub-sub worker processes
Sub-worker
Sub-worker
"run K steps of training"
"allreduce your gradients"
exchange weight shards through Ray object store
Ray Cluster

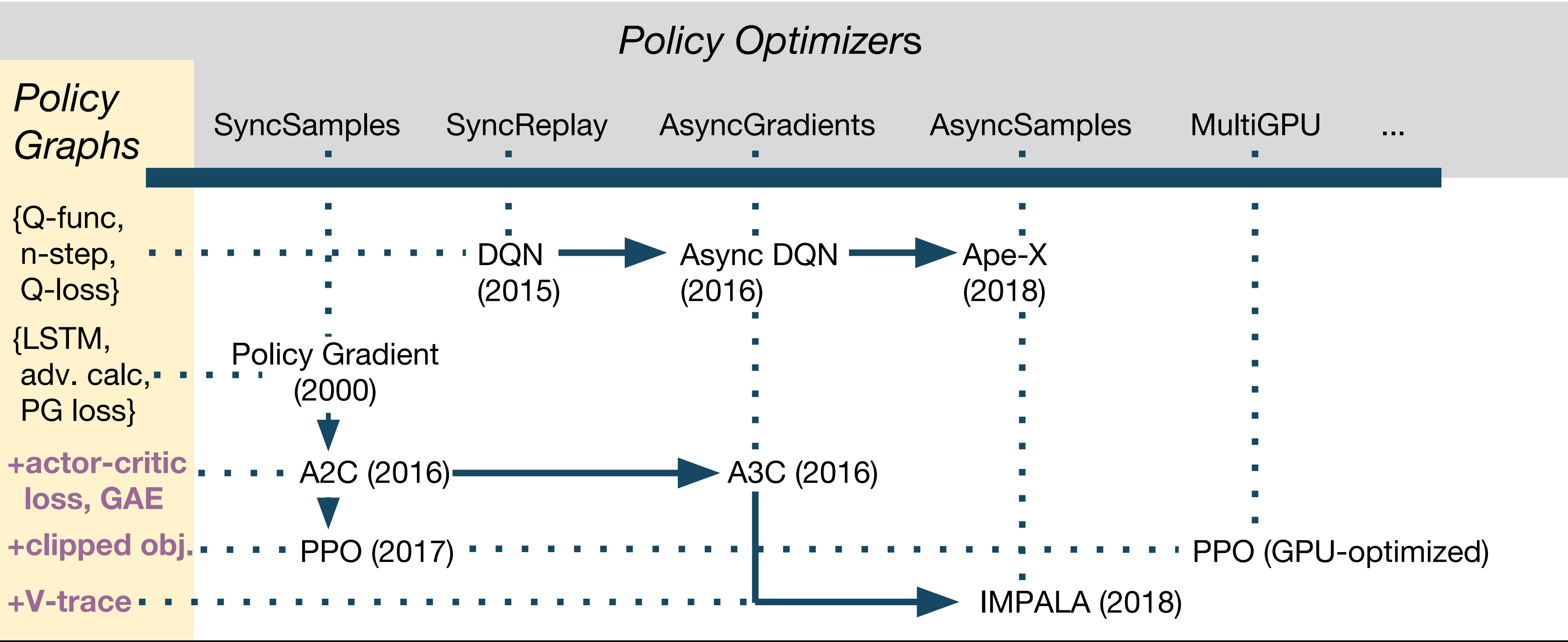
Unifying system enables RL Abstractions

Policy Optimizer Abstraction



● **Hierarchical Task Model**

RLlib Abstractions in Action



RLlib Reference Algorithms

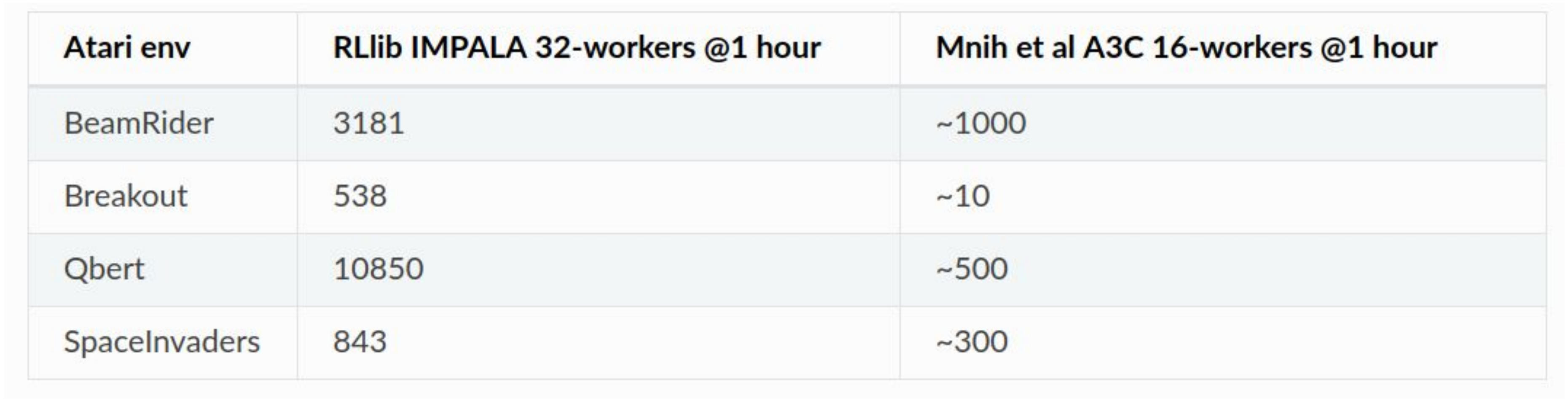
- **High-throughput architectures**
  - Distributed Prioritized Experience Replay (Ape-X)
  - Importance Weighted Actor-Learner Architecture (IMPALA)
- **Gradient-based**
  - Advantage Actor-Critic (A2C, A3C)
  - Deep Deterministic Policy Gradients (DDPG)
  - Deep Q Networks (DQN, Rainbow)
  - Policy Gradients
  - Proximal Policy Optimization (PPO)
- **Derivative-free**
  - Augmented Random Search (ARS)
  - Evolution Strategies

Community Contributions

FLOW Lab

Alibaba Group

RLlib Reference Algorithms

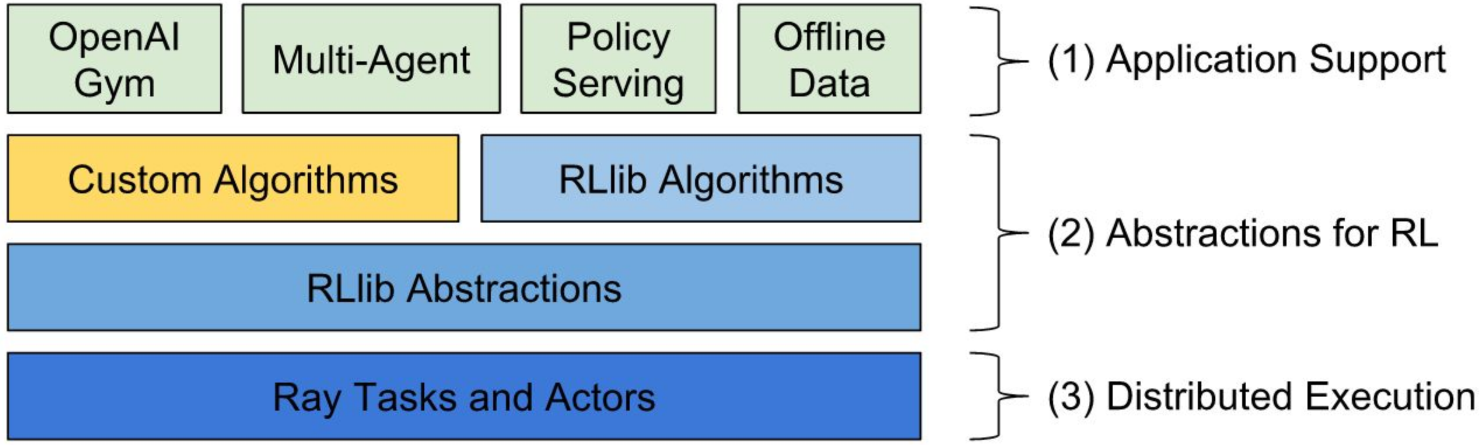


Atari env	RLlib IMPALA 32-workers @1 hour	Mnih et al A3C 16-workers @1 hour
BeamRider	3181	~1000
Breakout	538	~10
Qbert	10850	~500
SpaceInvaders	843	~300

1 GPU + 64 vCPUs (large single machine)

Scale your algorithms with RLlib

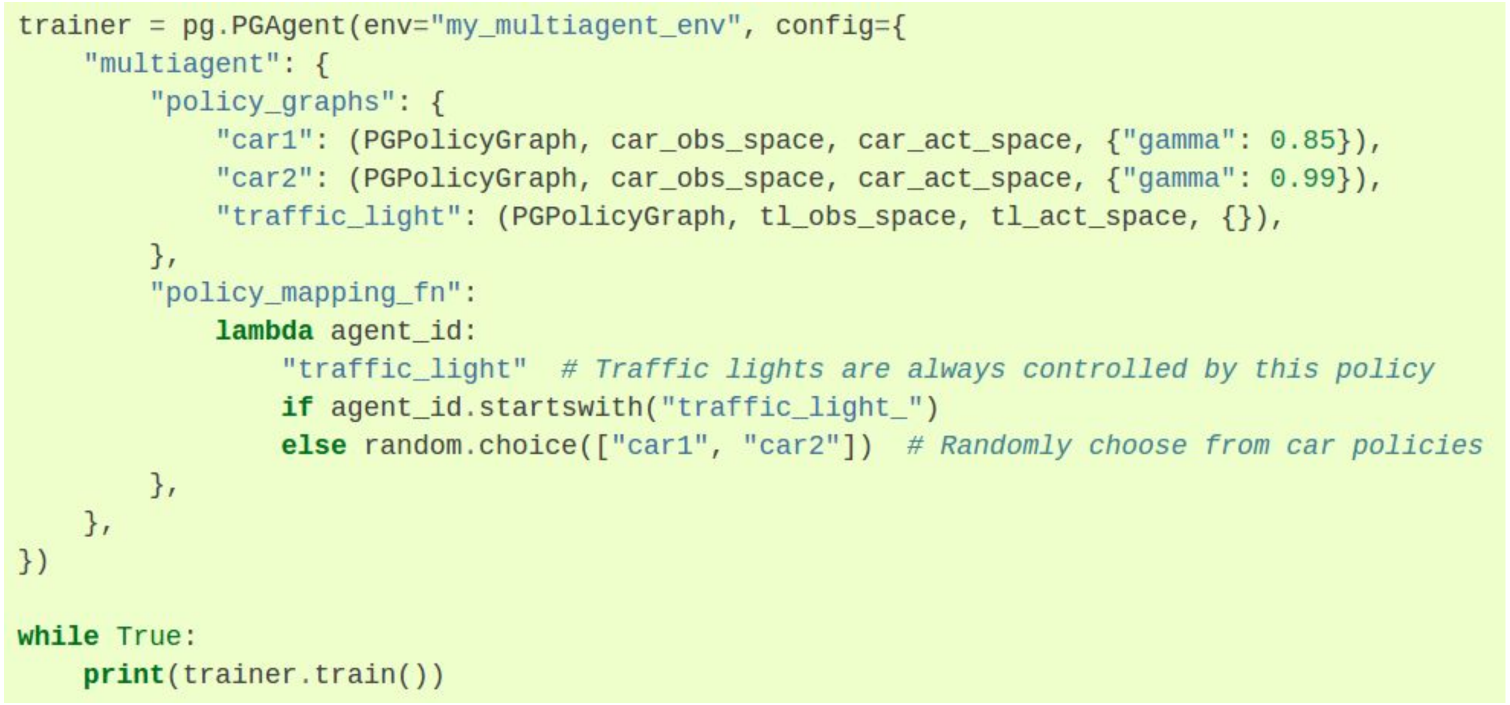
- Beyond a "collection of algorithms",
- RLlib's abstractions let you easily implement and scale new algorithms (multi-agent, novel losses, architectures, etc)



Code example: training PPO

```python
import ray
import ray.rllib.agents.ppo as ppo
from ray.tune.logger import pretty_print

ray.init()
config = ppo.DEFAULT_CONFIG.copy()
config["num_gpus"] = 0
config["num_workers"] = 1
agent = ppo.PPOAgent(config=config, env="CartPole-v0")

# Can optionally call agent.restore(path) to load a checkpoint.

for i in range(1000):
   # Perform one iteration of training the policy with PPO
   result = agent.train()
   print(pretty_print(result))

   if i % 100 == 0:
       checkpoint = agent.save()
       print("checkpoint saved at", checkpoint)
```

Code example: multi-agent RL



```python
trainer = pg.PGAgent(env="my_multiagent_env", config={
    "multiagent": {
        "policy_graphs": {
            "car1": (PGPolicyGraph, car_obs_space, car_act_space, {"gamma": 0.85}),
            "car2": (PGPolicyGraph, car_obs_space, car_act_space, {"gamma": 0.99}),
            "traffic_light": (PGPolicyGraph, tl_obs_space, tl_act_space, {}),
        },
        "policy_mapping_fn":
            lambda agent_id:
                "traffic_light"  # Traffic lights are always controlled by this policy
                if agent_id.startswith("traffic_light_")
                else random.choice(["car1", "car2"])  # Randomly choose from car policies
        },
    },
})

while True:
    print(trainer.train())
```

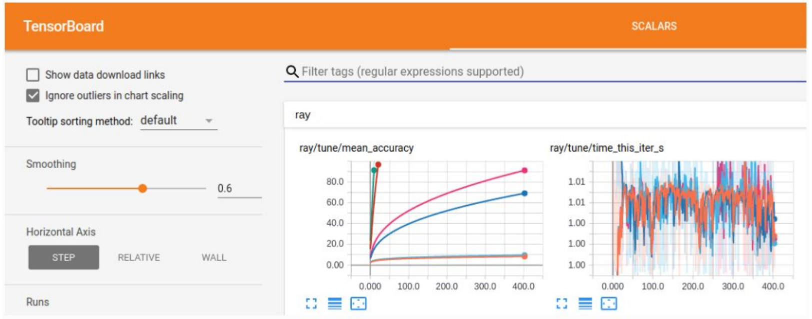
http://rllib.io

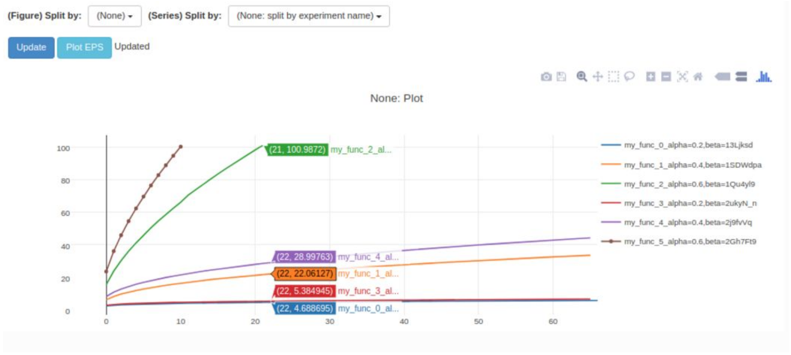
Code example: hyperparam tuning

```python
import ray
import ray.tune as tune

ray.init()
tune.run_experiments({
    "my_experiment": {
        "run": "PPO",
        "env": "CartPole-v0",
        "stop": {"episode_reward_mean": 200},
        "config": {
            "num_gpus": 0,
            "num_workers": 1,
            "sgd_stepsize": tune.grid_search([0.01, 0.001, 0.0001]),
        },
    },
})
```

Code example: hyperparam tuning

```
== Status ==
Using FIFO scheduling algorithm.
Resources requested: 4/4 CPUs, 0/0 GPUs
Result logdir: ~/ray_results/my_experiment
PENDING trials:
 - PPO_CartPole-v0_2_sgd_stepsize=0.0001:   PENDING
RUNNING trials:
 - PPO_CartPole-v0_0_sgd_stepsize=0.01:     RUNNING [pid=21940], 16 s, 4013 ts, 22 rew
 - PPO_CartPole-v0_1_sgd_stepsize=0.001:    RUNNING [pid=21942], 27 s, 8111 ts, 54.7 rew
```


TensorBoard
SCALARS
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Filter tags (regular expressions supported)
ray
ray/tune/mean_accuracy
ray/tune/time_this_iter_s



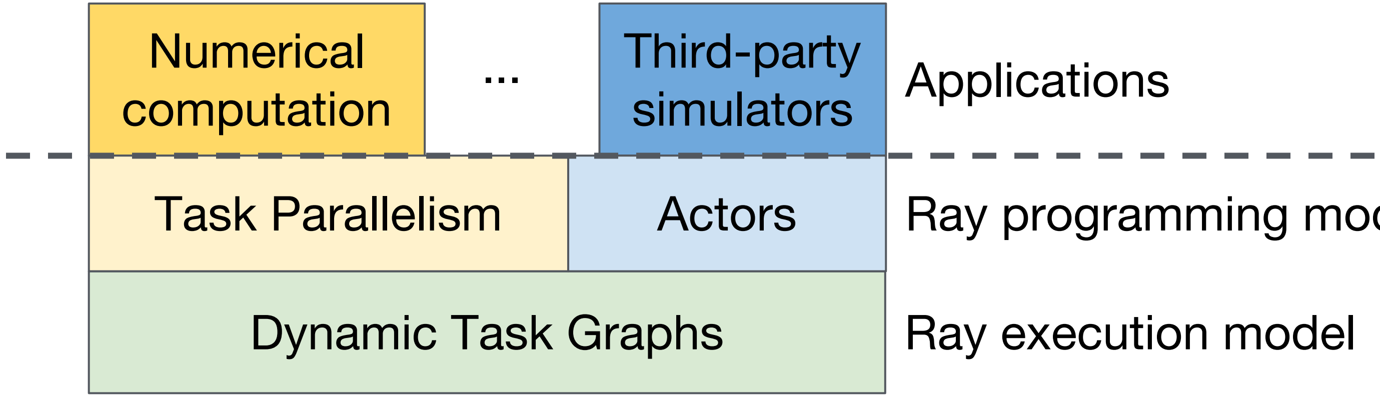
Summary: Ray and RLlib addresses challenges in providing scalable abstractions for reinforcement learning.

RLlib is open source and available at http://rllib.io

Thanks!

Ray distributed execution engine

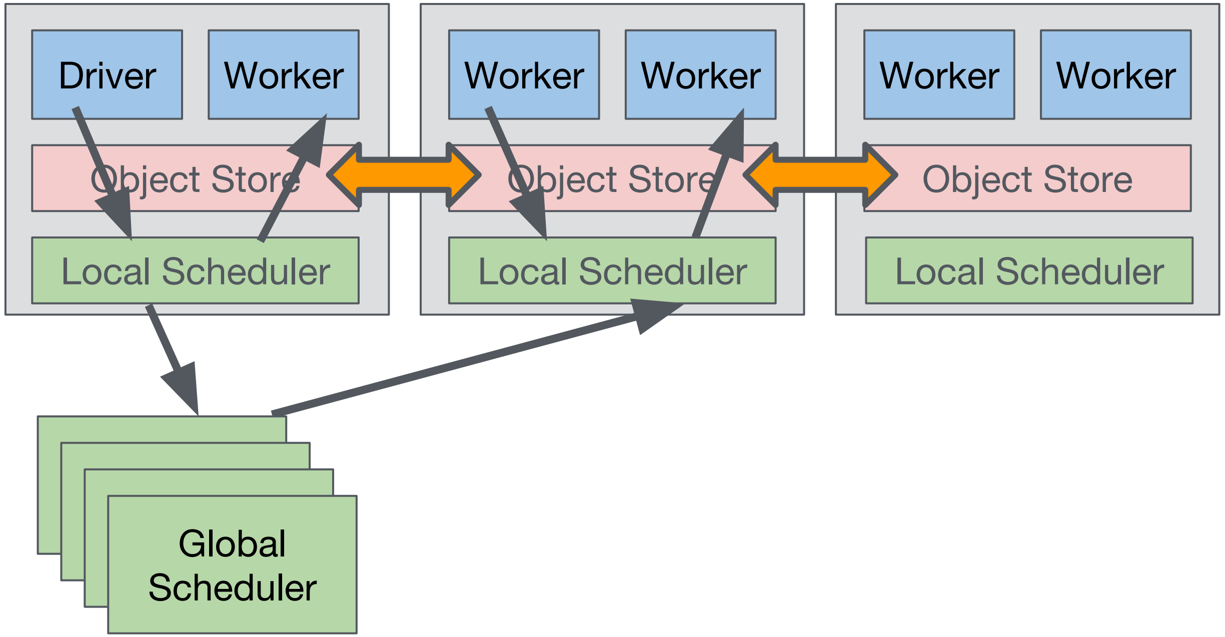
- Ray provides **Task parallel** and **Actor** APIs built on **dynamic task graphs**



- These APIs are used to build distributed **applications**, **libraries** and **systems**

Ray distributed scheduler

- Faster than Python multi-processing on a single node
- Competitive with MPI in many workloads


Driver
Worker
Worker
Worker
Worker
Worker
Object Store
Object Store
Object Store
Local Scheduler
Local Scheduler
Local Scheduler
Global Scheduler