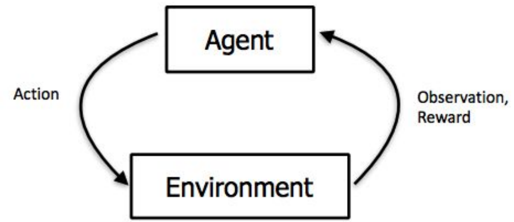
Distributed RL

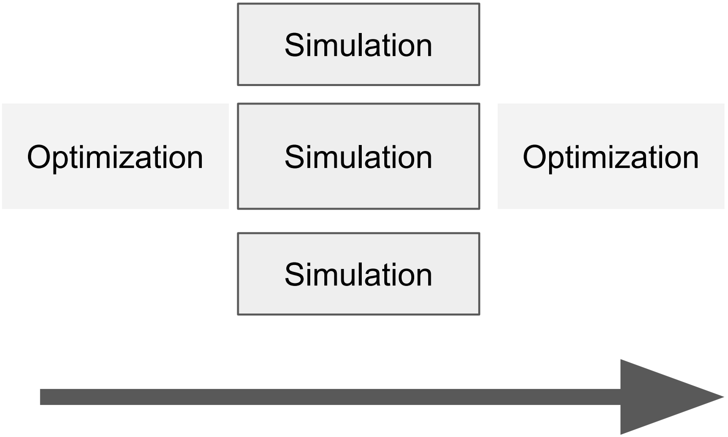
Richard Liaw

Common Computational Patterns for RL

Original



Batch Optimization

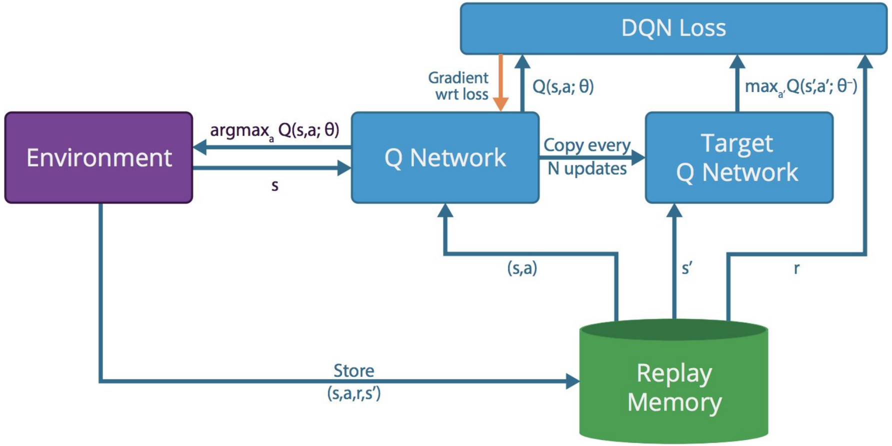


How can we **better utilize** our computational resources **to accelerate** RL progress?

History of large scale distributed RL

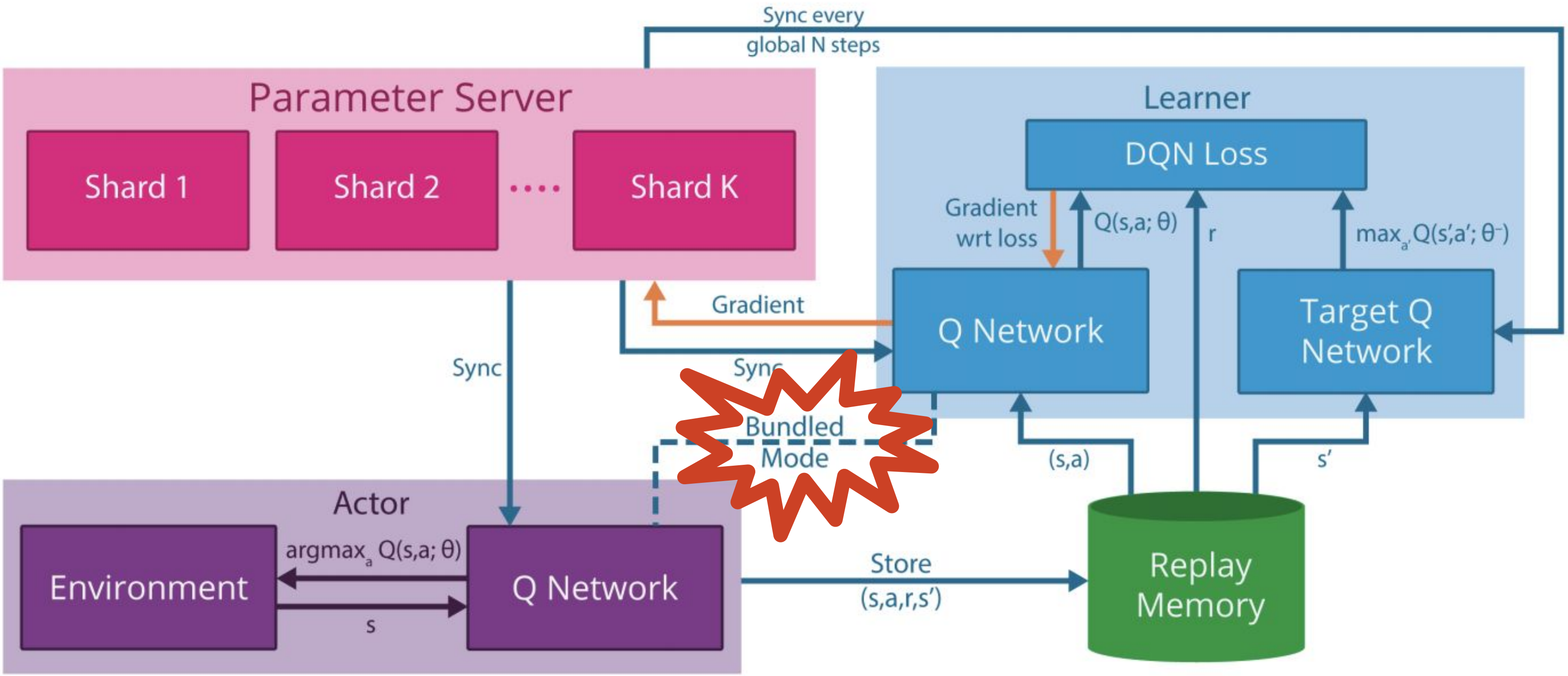
2013	2015	2016	2018	2018	2019
DQN	**GORILA**	**A3C**	**IMPALA**	**Ape-X**	**R2D3**
Playing Atari with Deep Reinforcement Learning (Mnih 2013)	Massively Parallel Methods for Deep Reinforcement Learning (Nair 2015)	Asynchronous Methods for Deep Reinforcement Learning (Mnih 2016)	IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures (Espeholt 2018)	Distributed Prioritized Experience Replay (Horgan 2018)	Making Efficient Use of Demonstrations to Solve Hard Exploration Problems (Le Paine 2019)

2013/2015: DQN

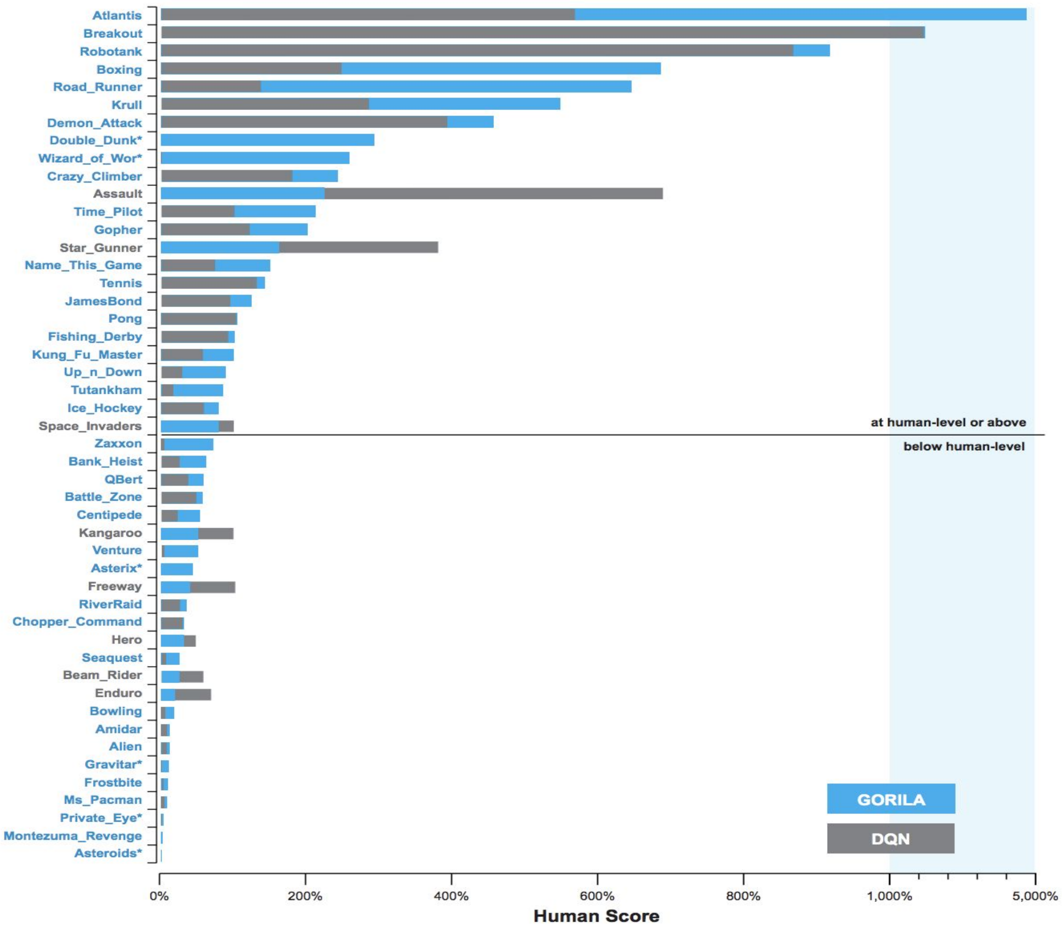



```
for i in range(T):
    s, a, s_1, r = evaluate()
    replay.store((s, a, s_1, r))

    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



History of large scale distributed RL

2013

DQN

Playing Atari with Deep Reinforcement Learning (Mnih 2013)

2015

GORILA

Massively Parallel Methods for Deep Reinforcement Learning (Nair 2015)

2016

A3C

Asynchronous Methods for Deep Reinforcement Learning (Mnih 2016)

2018

IMPALA

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

2018

Ape-X

Distributed Prioritized Experience Replay (Horgan 2018)

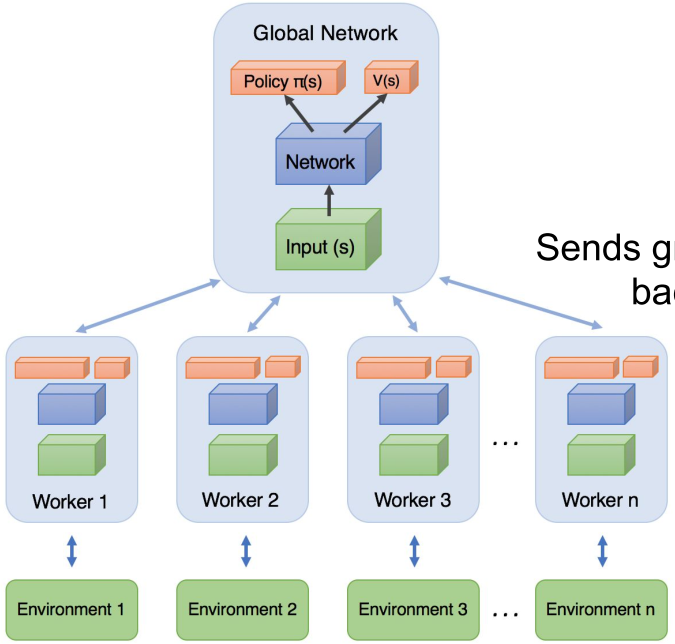
2019

R2D3

Making Efficient Use of Demonstrations to Solve Hard Exploration Problems (Le Paine 2019)

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

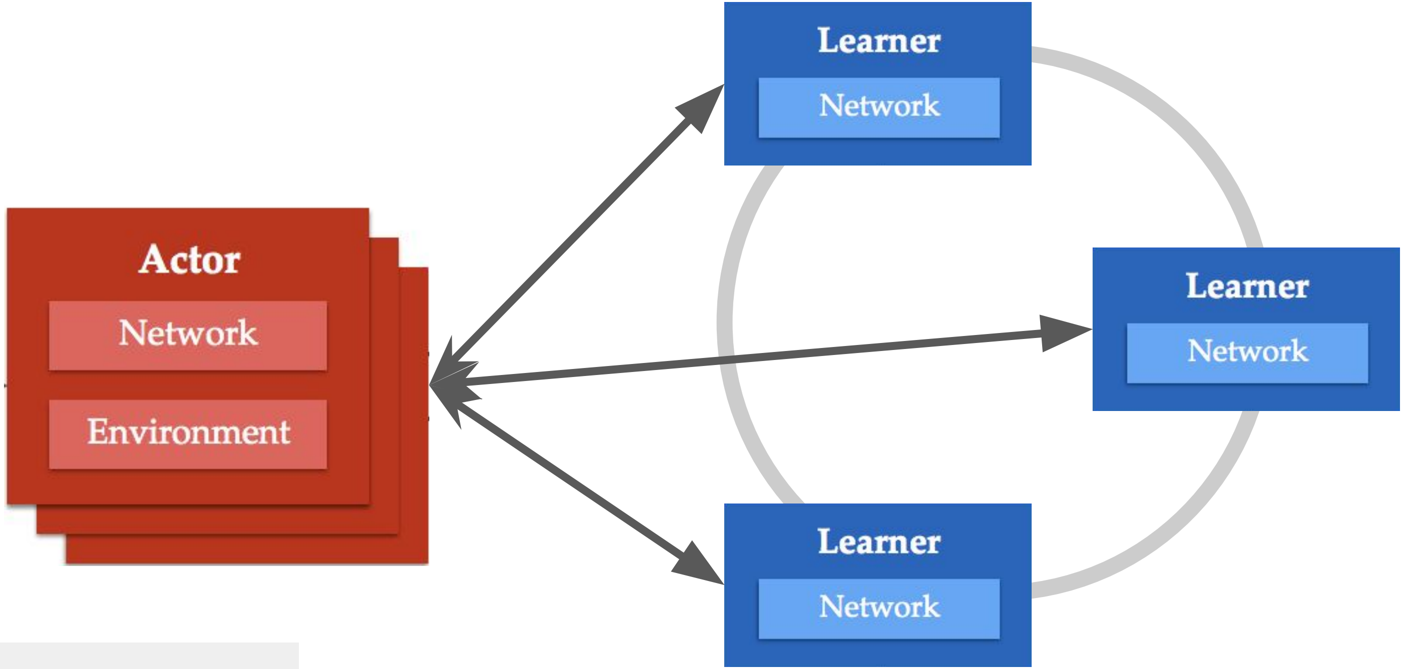
Changes to GORILA:

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%

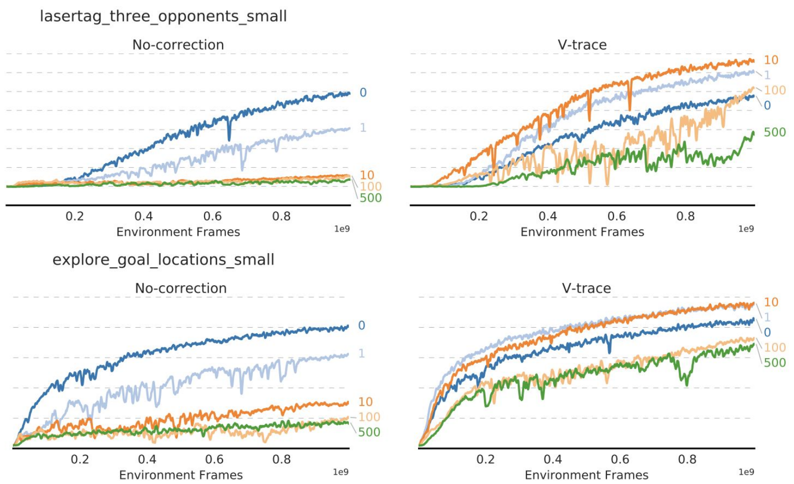
Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary

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:

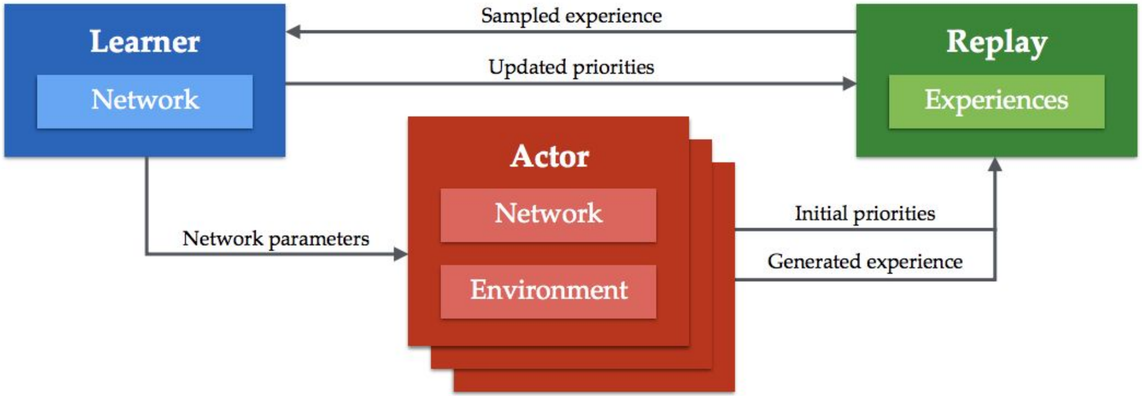
$$v_s \stackrel{\text{def}}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s}\Big(\prod_{i=s}^{t-1} c_i\Big)\delta_t V, \quad (1)$$

Ape-X/R2D2 (2018)

Scaling Off-Policy learning...

Ape-X:

1. Distributed DQN/DDPG/R2D2
2. Reintroduces replay
3. **Distributed Prioritization:** Unlike Prioritized DQN, initial priorities are not set to “max TD”



Ape-X Performance

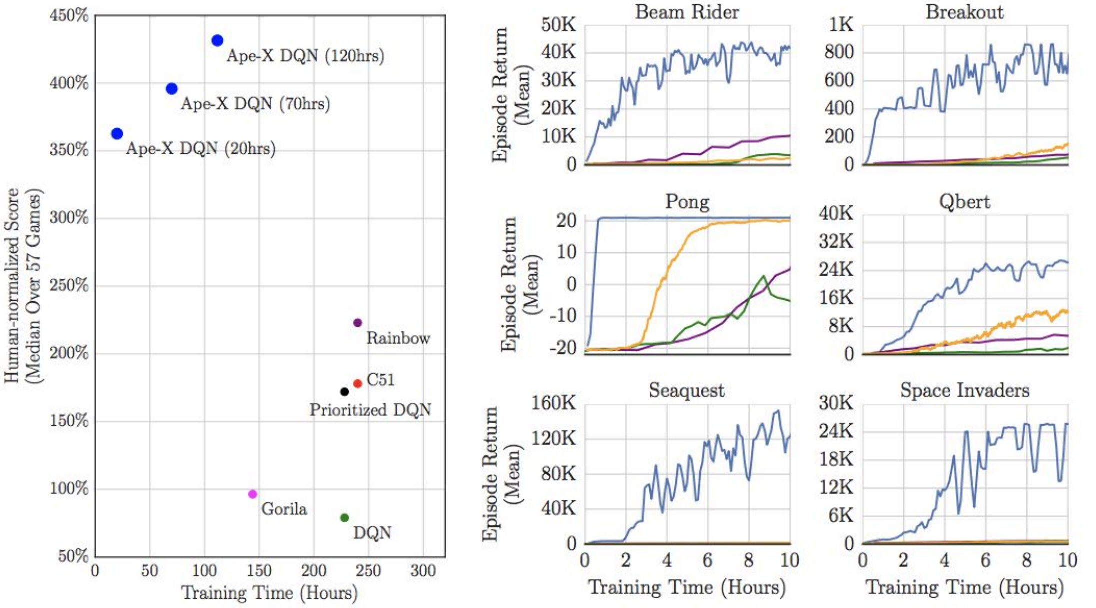
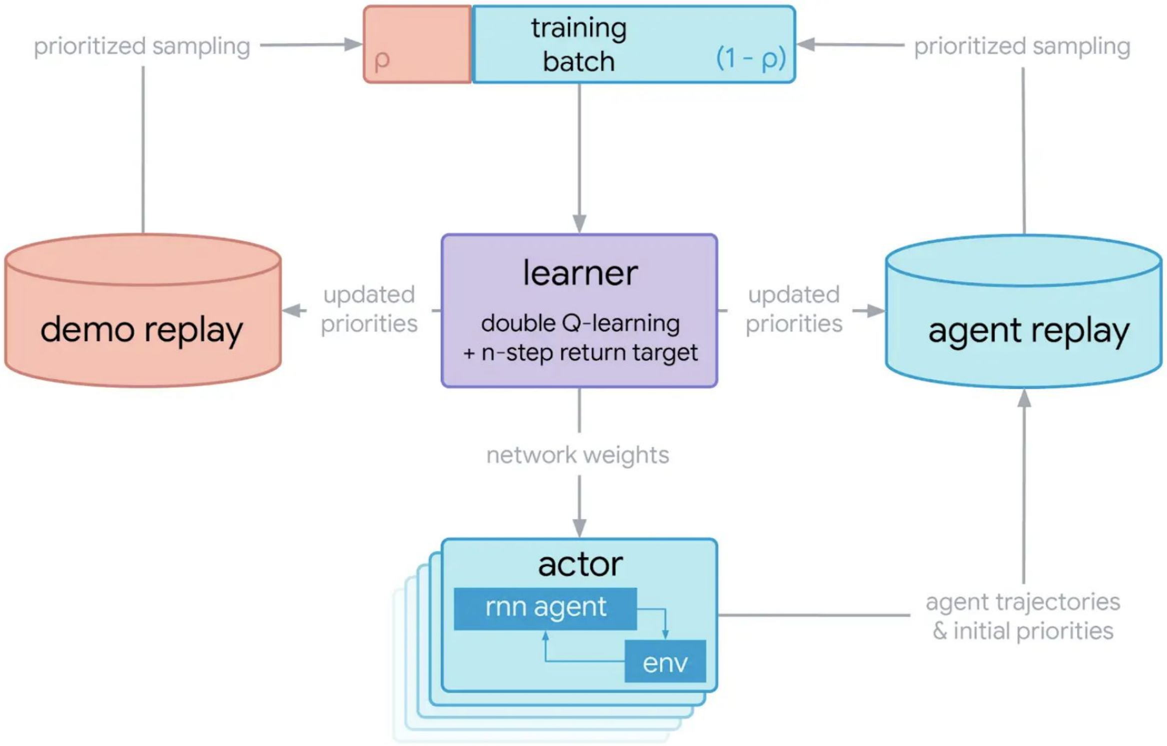


Figure 2: Left: Atari results aggregated across 57 games, evaluated from random no-op starts. Right: Atari training curves for selected games, against baselines. Blue: Ape-X DQN with 360 actors; Orange: A3C; Purple: Rainbow; Green: DQN. See appendix for longer runs over all games.

With Demonstrations: R2D3 (2019)



Other interesting distributed architectures

QT-Opt

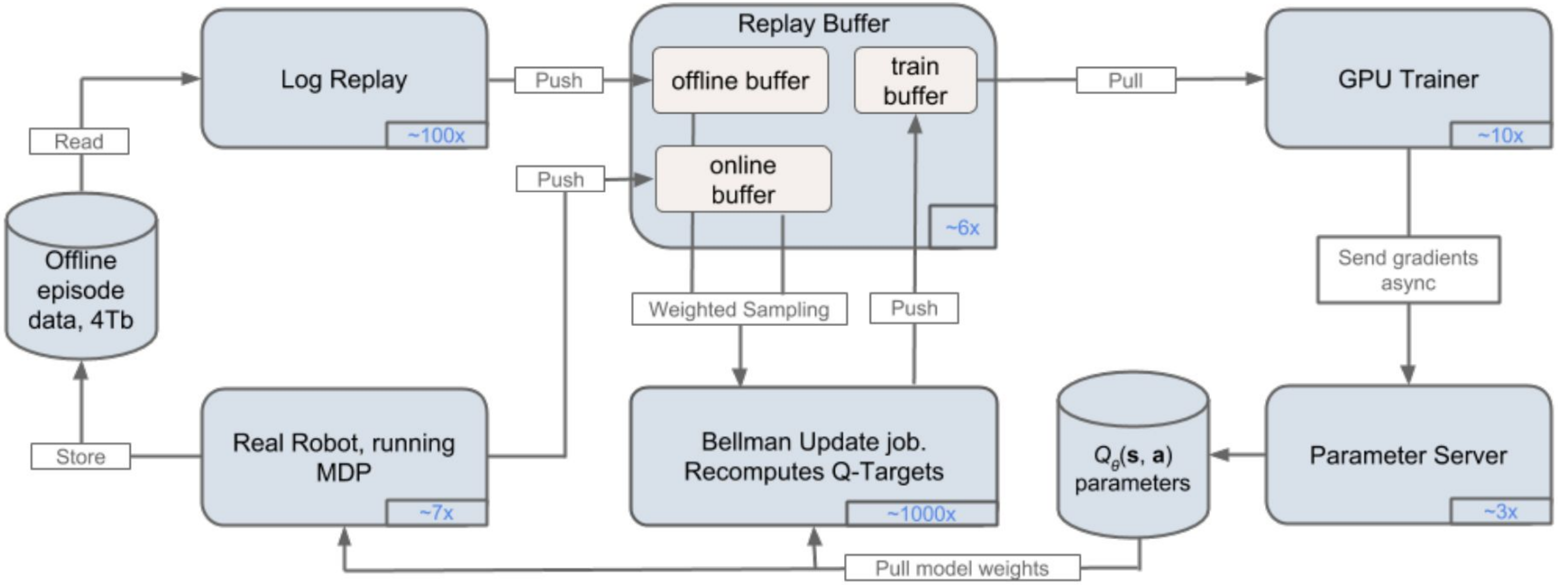
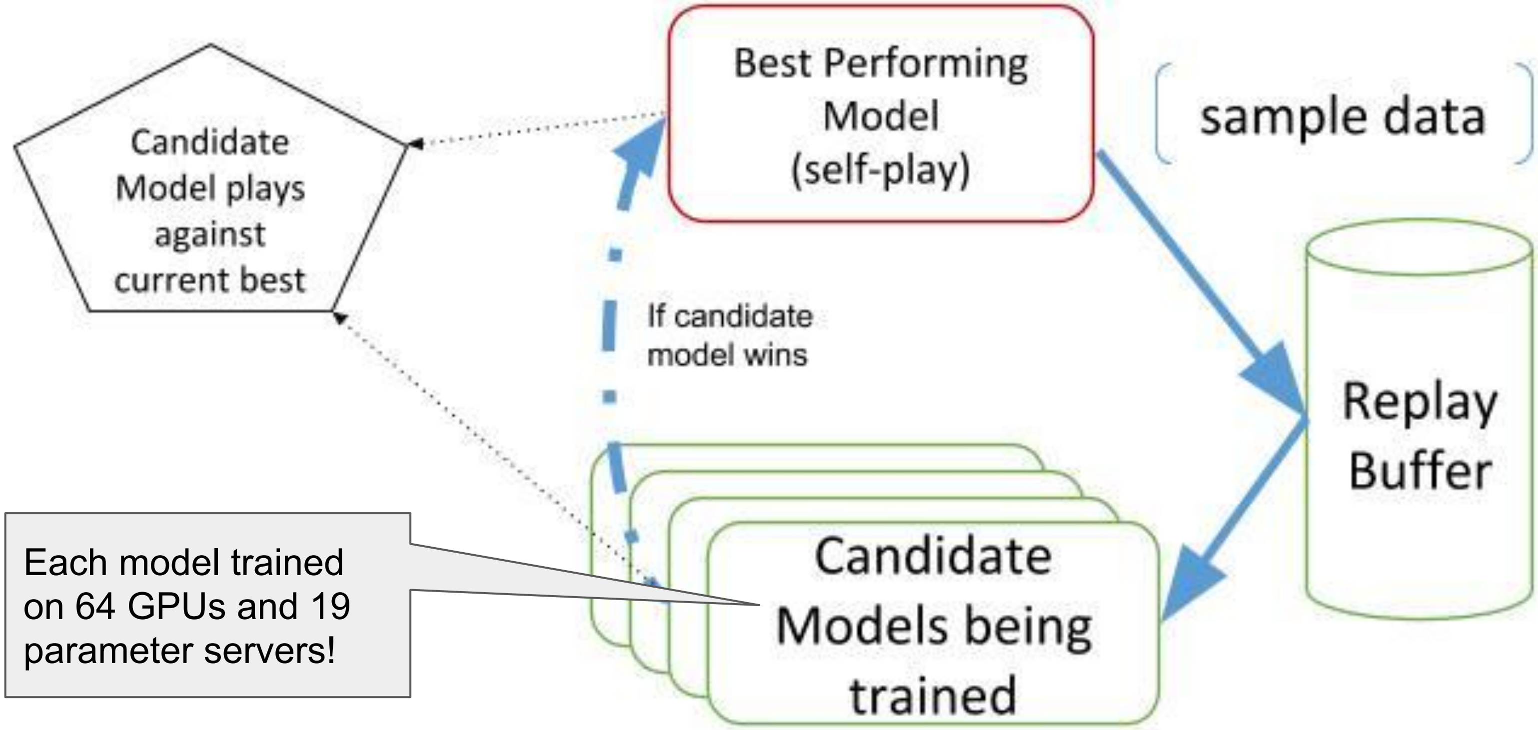


Figure 14: Architecture of the QT-Opt distributed reinforcement learning algorithm.

https://arxiv.org/pdf/1806.10293.pdf

AlphaZero



Each model trained on 64 GPUs and 19 parameter servers!

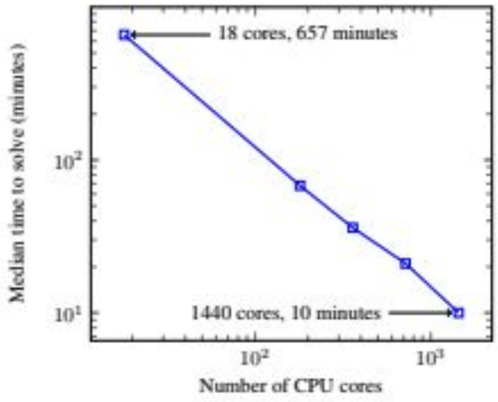
Evolution Strategies

Evolution Strategies as a Scalable Alternative to Reinforcement Learning

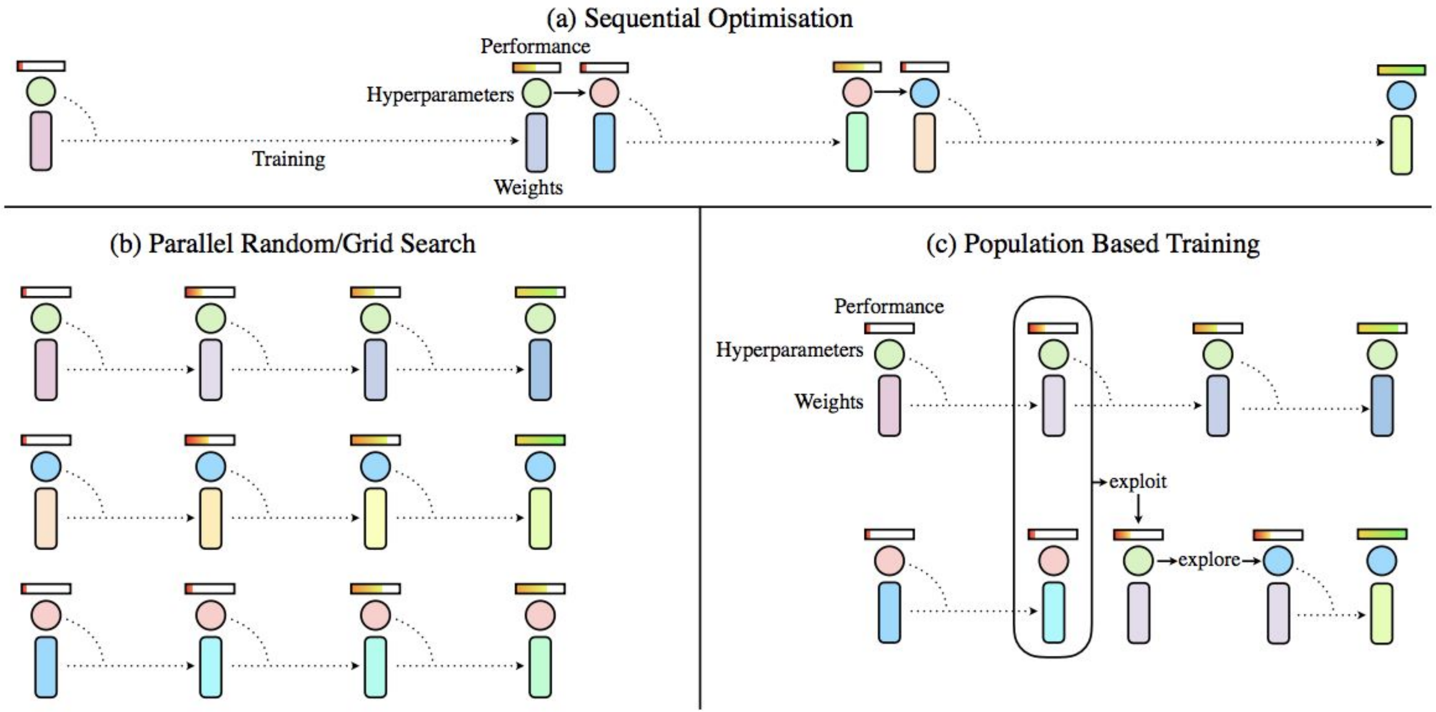
Tim Salimans **Jonathan Ho** **Xi Chen** **Szymon Sidor** **Ilya Sutskever**
OpenAI

Algorithm 2 Parallelized Evolution Strategies

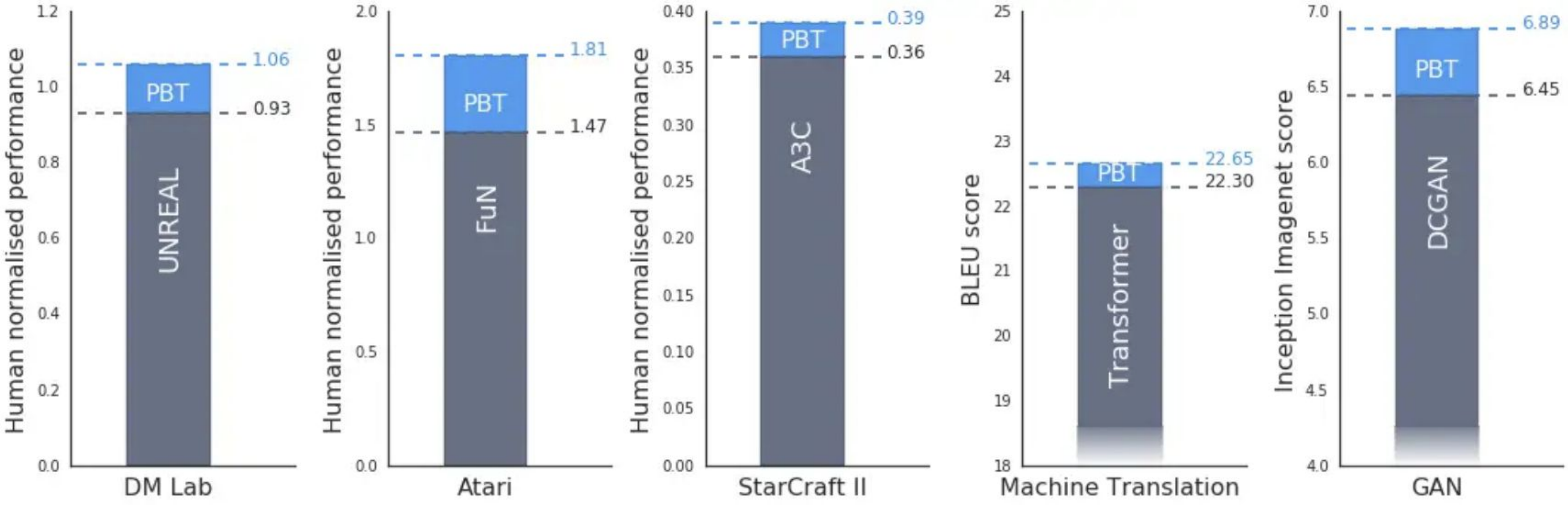
```
Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
Initialize: n workers with known random seeds, and initial parameters θ_0
for t = 0, 1, 2, ... do
  for each worker i = 1, ..., n do
    Sample ε_i ~ N(0, I)
    Compute returns F_i = F(θ_t + σ ε_i)
  end for
  Send all scalar returns F_i from each worker to every other worker
  for each worker i = 1, ..., n do
    Reconstruct all perturbations ε_j for j = 1, ..., n using known random seeds
    Set θ_{t+1} ← θ_t + α (1/(nσ)) Σ_{j=1}^{n} F_j ε_j
  end for
end for
```



Beyond RL: Population-based Training



Benefits of PBT



https://deepmind.com/blog/article/population-based-training-neural-networks

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

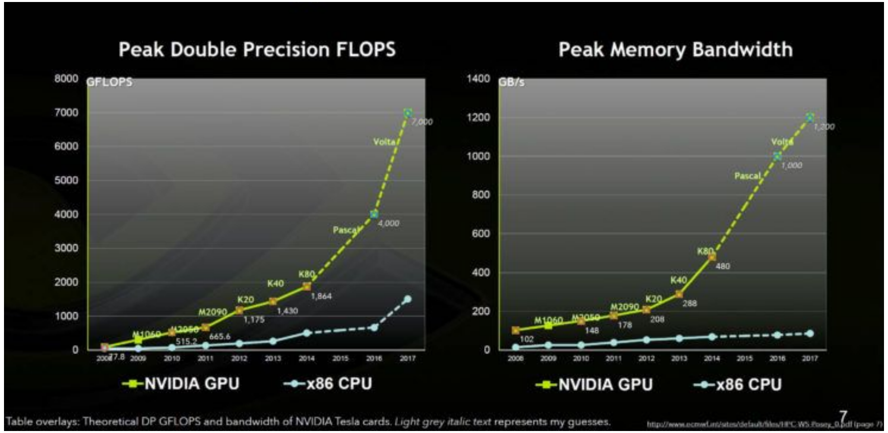


Fig. courtesy NVidia Inc.

CPU GPU TPU

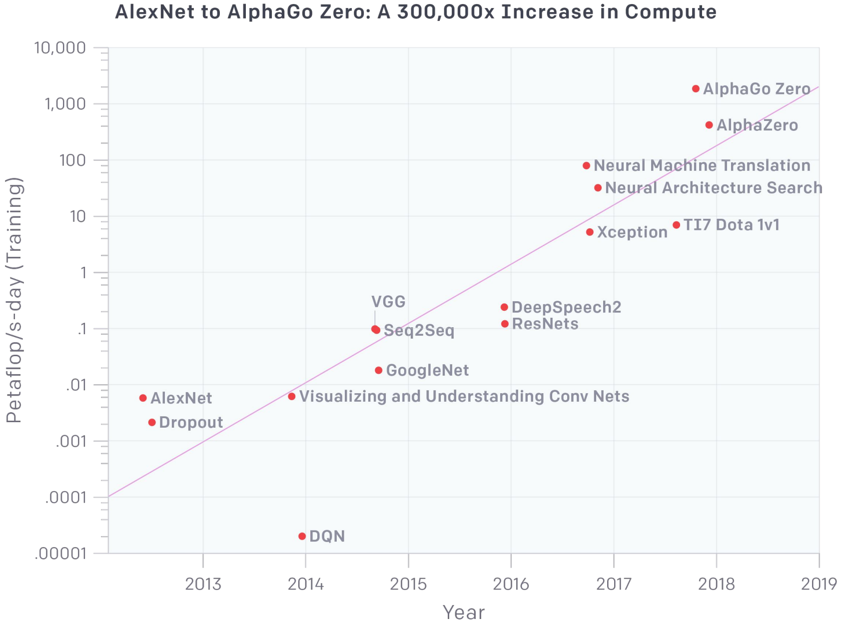
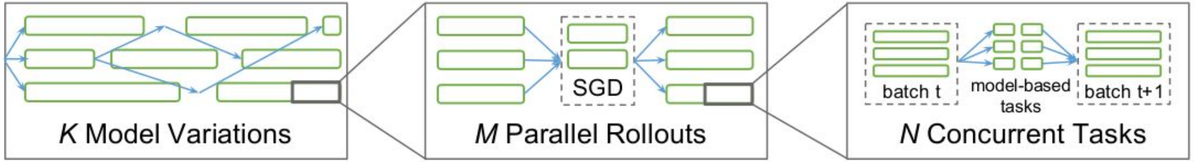


Fig. courtesy OpenAI

How do we leverage this hardware?

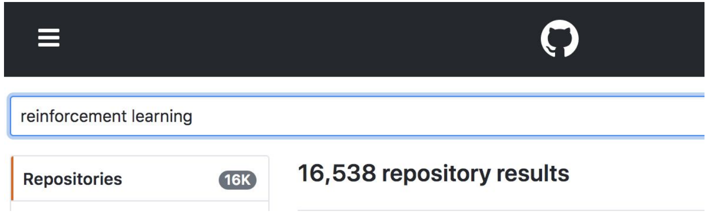
(a) Supervised Learning



(b) Reinforcement Learning

scalable abstractions for RL?

Systems for RL today



- Many implementations (16000+ repos on GitHub!)
  - how general are they (and do they scale)?

    PPO: multiprocessing, MPI

    AlphaZero: custom systems

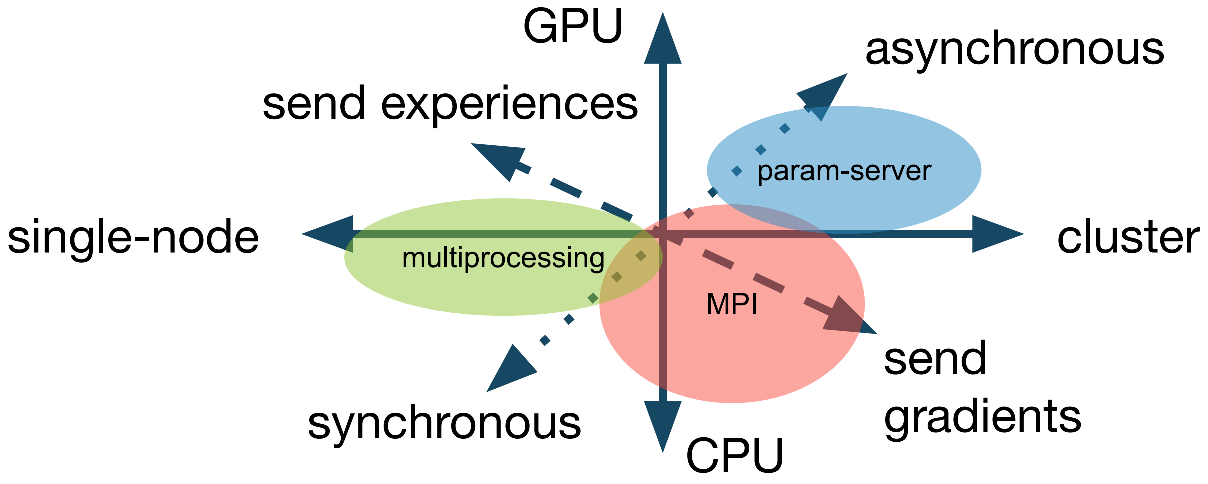
    Evolution Strategies: Redis

    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

Agent

Environment

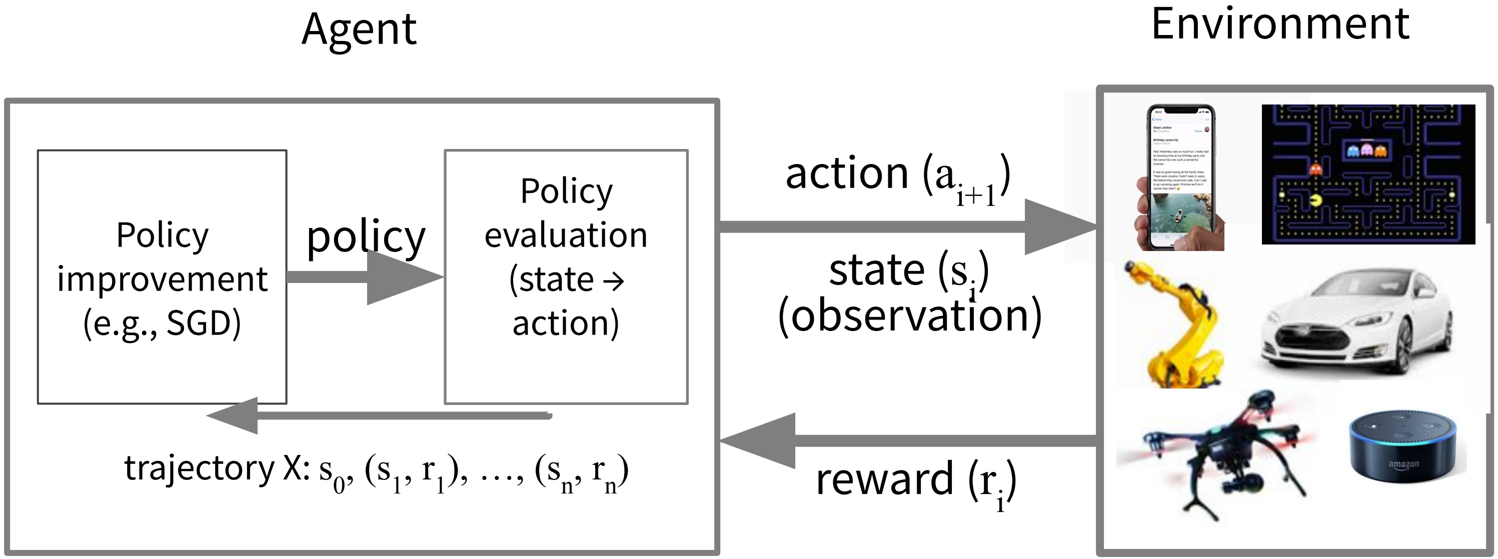
Policy:
state → action

action (a_{i+1})

state (s_i)
(observation)

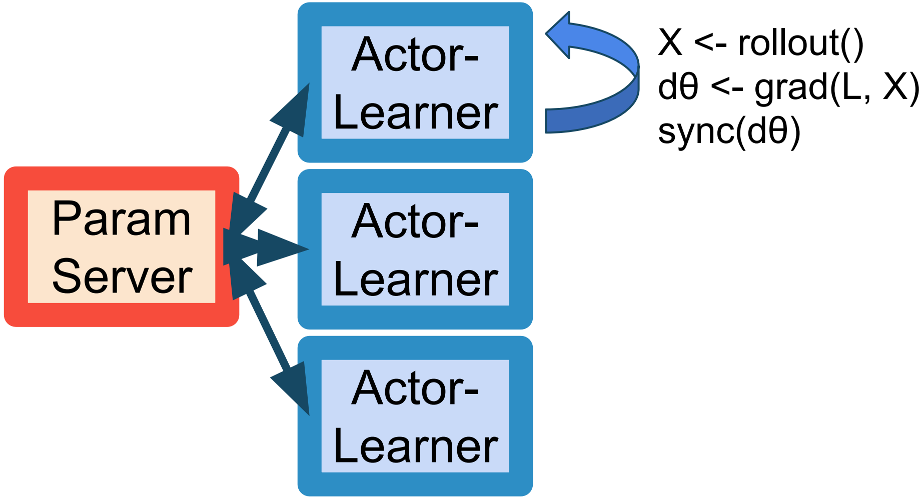
reward (r_i)

Structure of RL computations

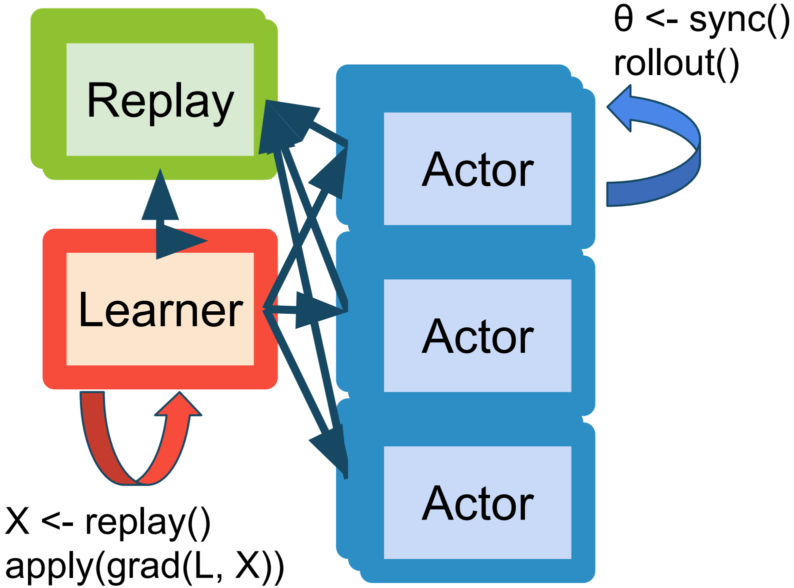


Many RL loop decompositions

Async DQN (Mnih et al; 2016)

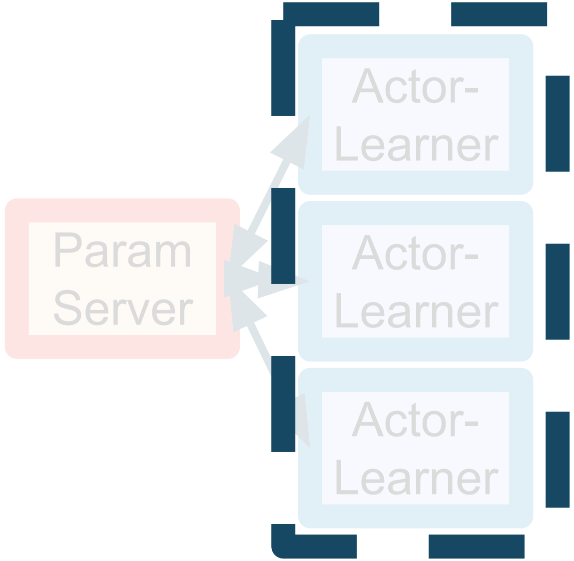


Ape-X DQN (Horgan et al; 2018)

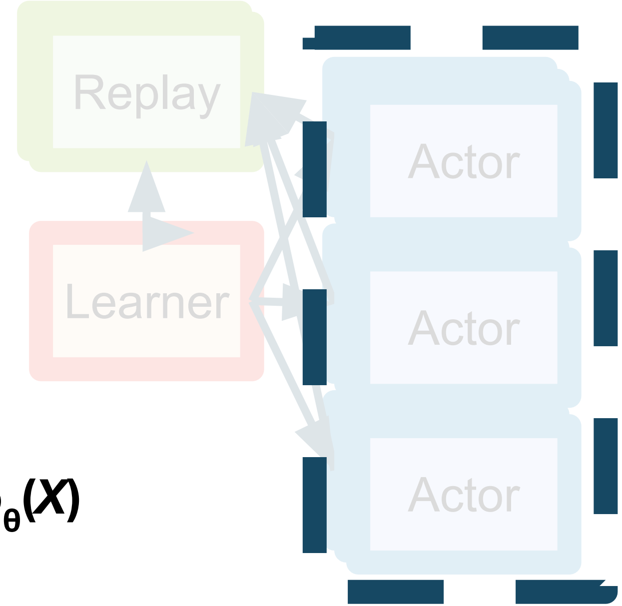


Common components

Async DQN (Mnih et al; 2016)



Ape-X DQN (Horgan et al; 2018)



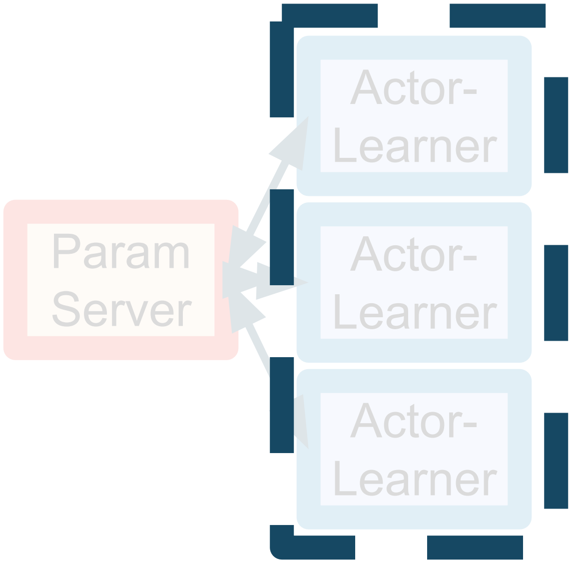
Policy $\pi_\theta(o_t)$

Trajectory postprocessor $\rho_\theta(X)$

Loss $L(\theta,X)$

Common components

Async DQN (Mnih et al; 2016)

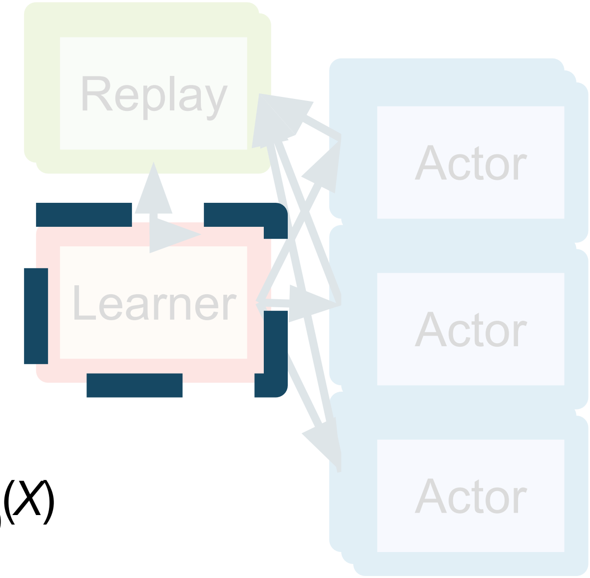


Policy $\pi_\theta(o_t)$

Trajectory postprocessor $\rho_\theta(X)$

Loss $L(\theta,X)$

Ape-X DQN (Horgan et al; 2018)



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.

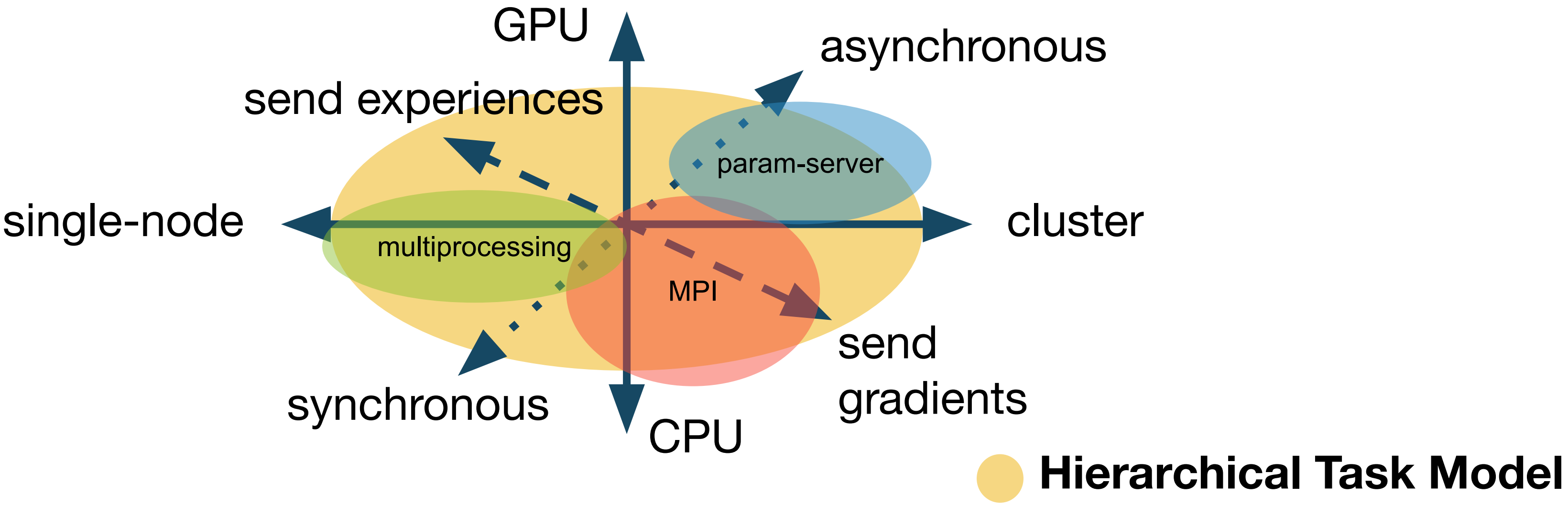
Requirements for a new system

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

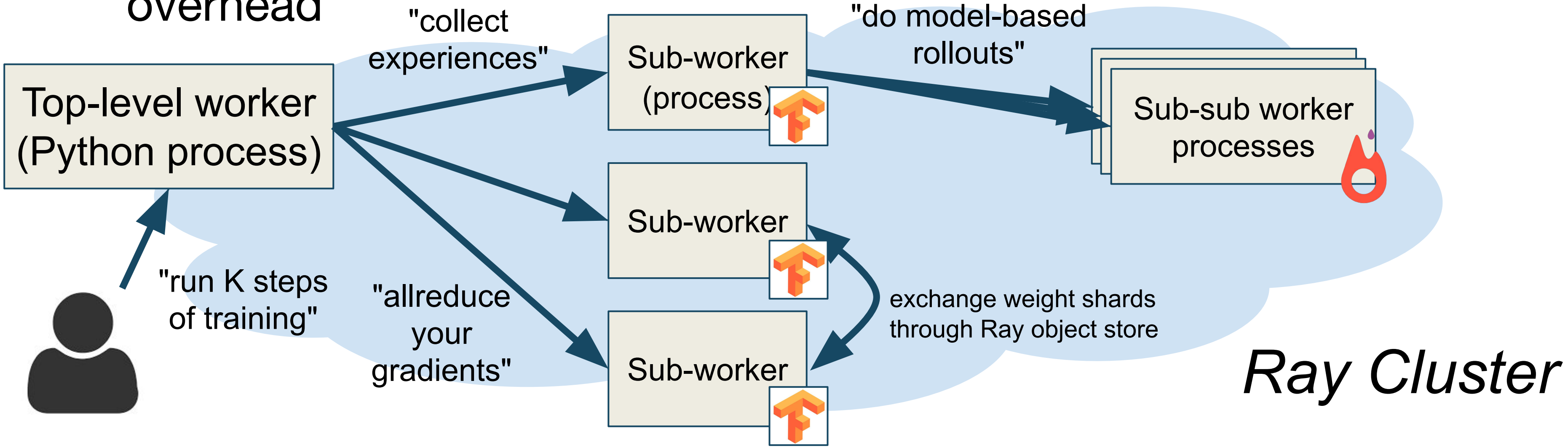
Ray System Substrate

- 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.)



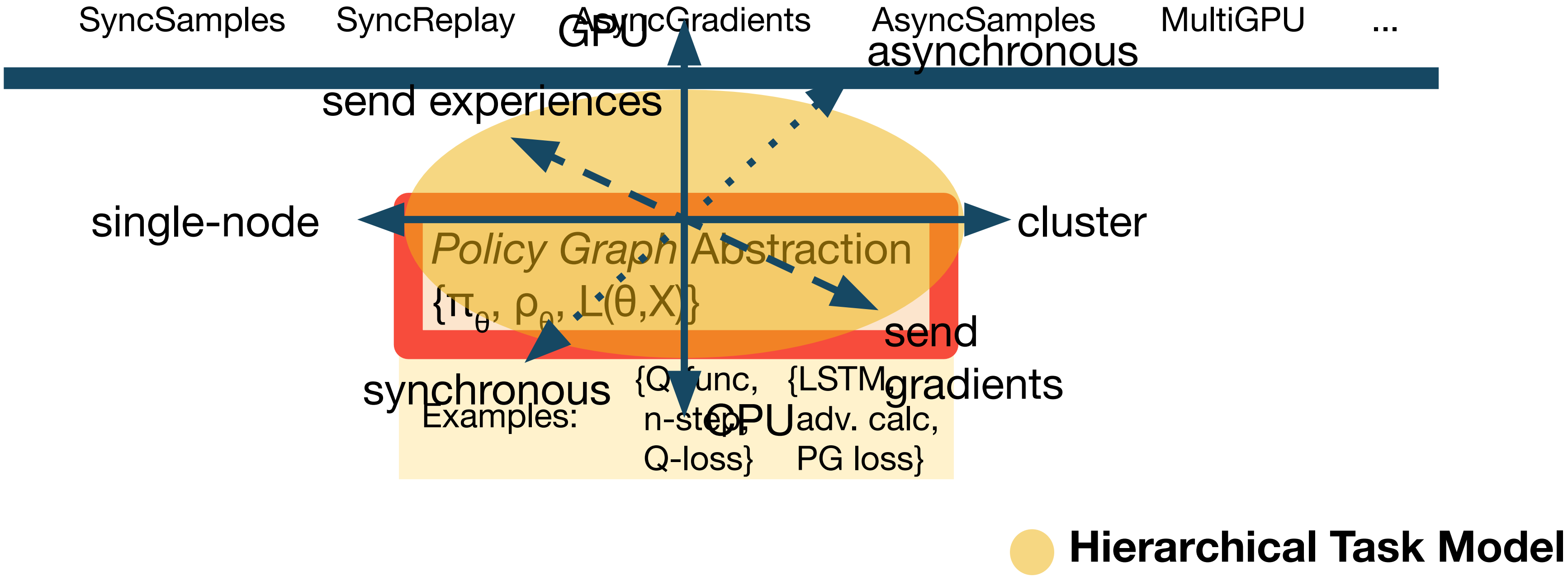
Hierarchical Parallel Task Model

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

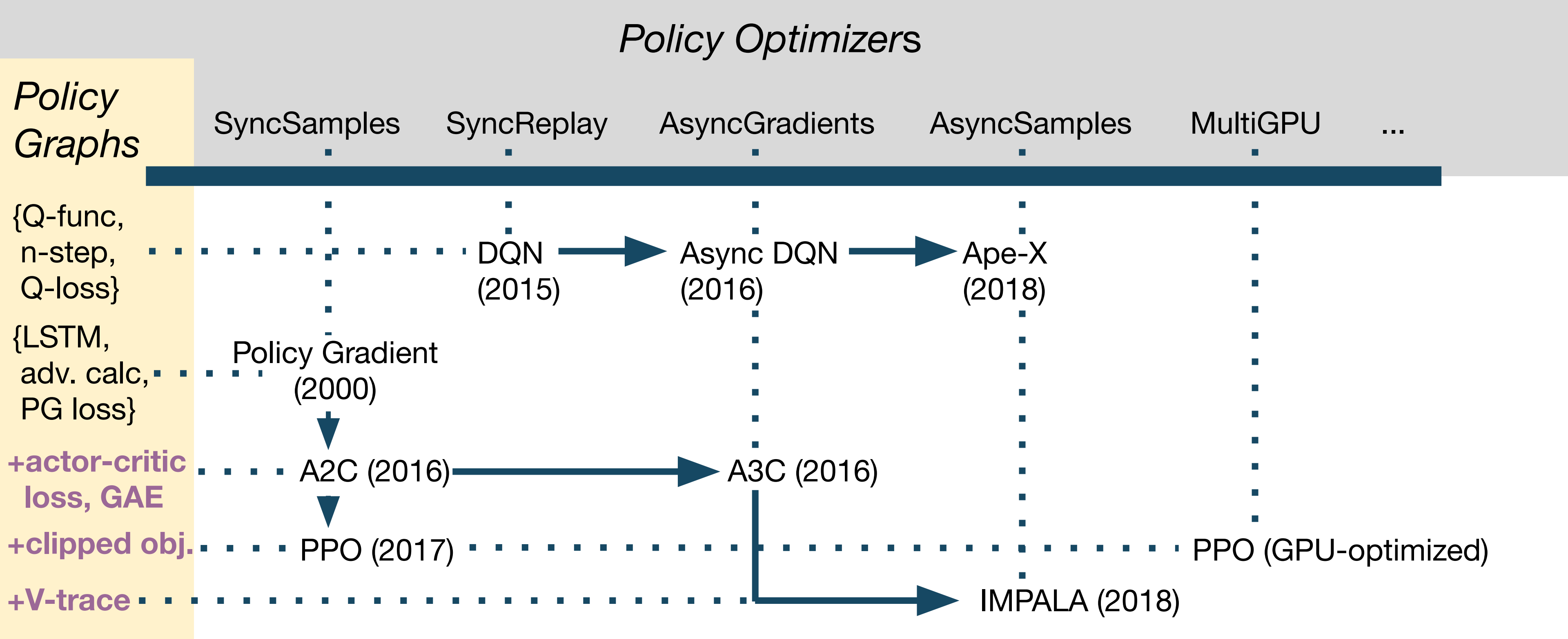


Unifying system enables RL Abstractions

Policy Optimizer Abstraction



RLlib Abstractions in Action



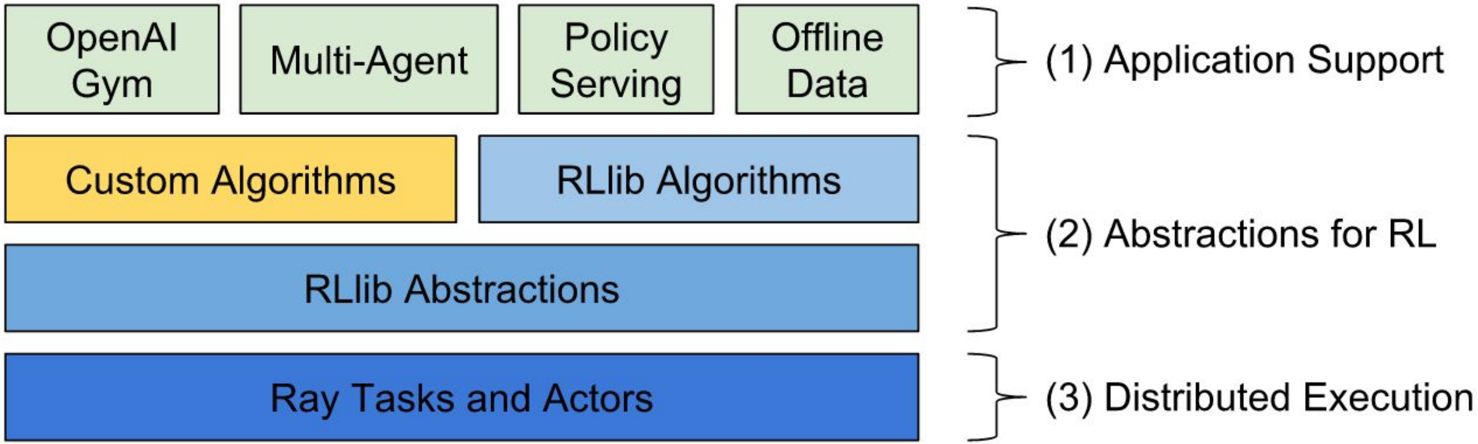
RLlib Reference Algorithms

- **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

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

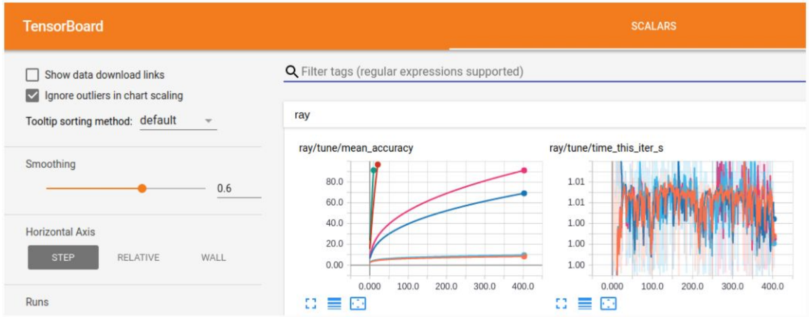
```
import ray
import ray.rllib.agents.ppo as ppo
from ray.tune.logger import pretty_print

ray.init()
config = ppo.DEFAULT_CONFIG.copy()
config["num_gpus"] = 0
config["num_workers"] = 1
config["eager"] = False
trainer = ppo.PPOTrainer(config=config, env="CartPole-v0")

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

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

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

Code example: hyperparam tuning

```
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
```



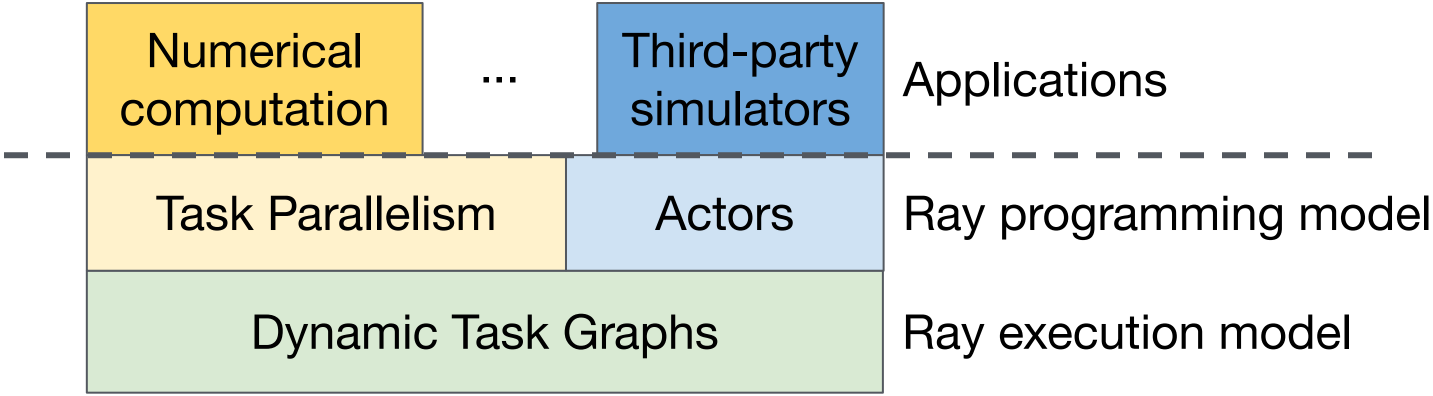
Summary: Ray and RLlib addresses challenges in providing scalable abstractions for reinforcement learning.

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