Distributed RL

Joash Lee

Pan Liangming

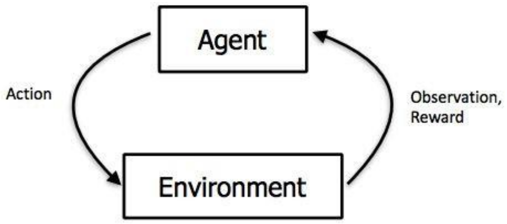
Vicky Feliren

Lesson Objectives

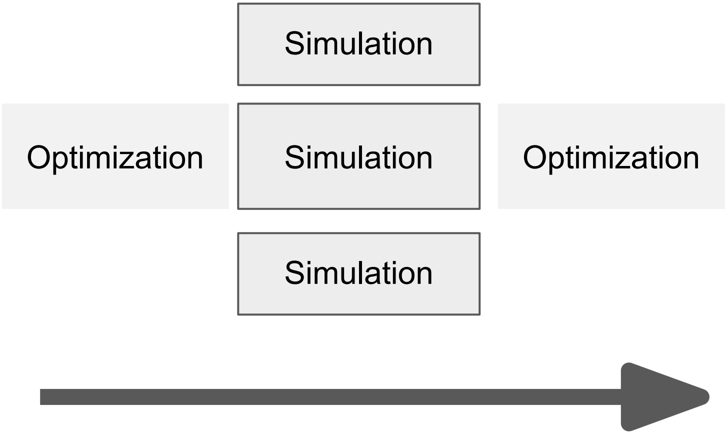
1. Why parallelise?
2. Understand how the computation of standard RL algorithms can be distributed to decrease wall-clock training time.
3. How these distributed RL algorithms can be modularised.
4. How modularised distributed RL algorithms can be implemented on real systems - case study: RLlib
5. Examples on using RLlib

Common Computational Patterns for RL

Original

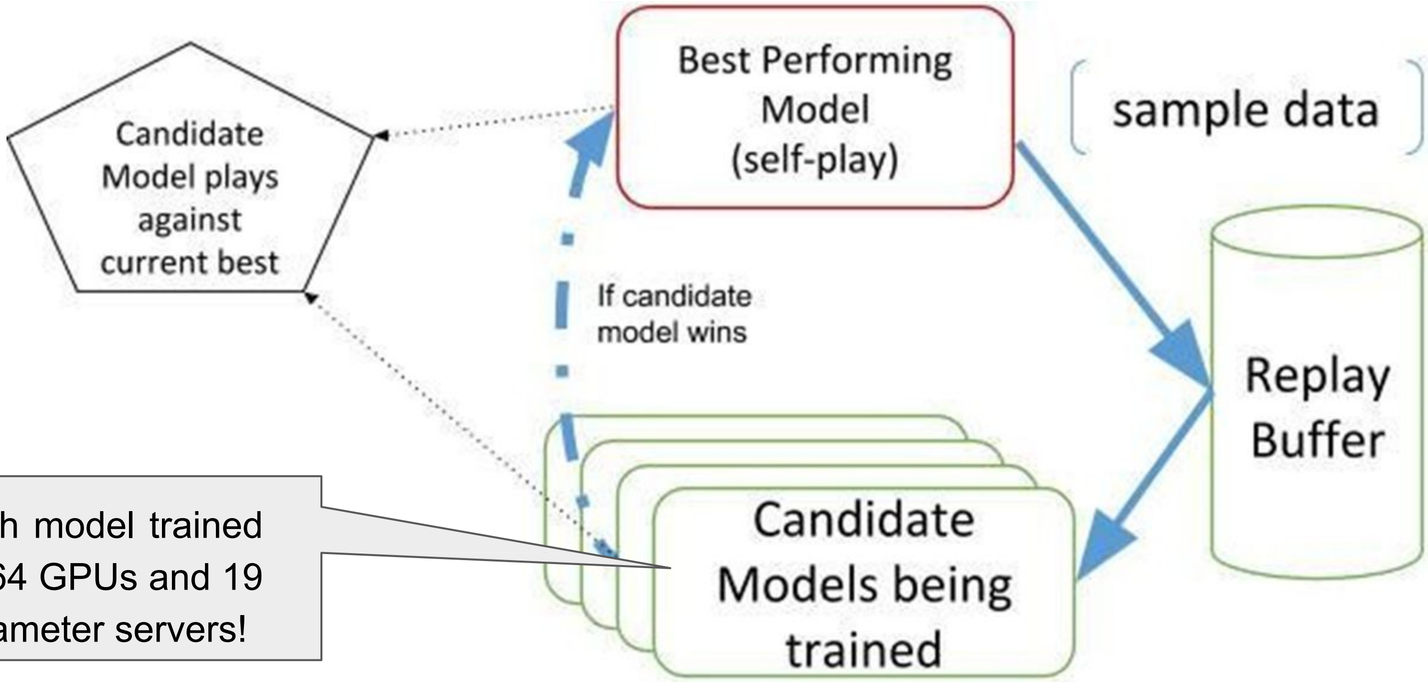


Batch Optimization



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

AlphaZero



Each model trained on 64 GPUs and 19 parameter servers!

Lesson Objectives

1. Why parallelise?
2. **Understand how the computation of standard RL algorithms can be distributed to decrease wall-clock training time.**
3. How these distributed RL algorithms can be modularised.
4. How modularised distributed RL algorithms can be implemented on real systems - case study: RLlib
5. Examples on using RLlib

History of large scale distributed RL

2013

DQ
N

Playing Atari with Deep Reinforcement Learning (Mnih 2013)

Q-Learnin
g

2015

GORIL
A

Massively Parallel Methods for Deep Reinforcement Learning (Nair 2015)

Policy Gradient

2016

A3C

Asynchronous Methods for Deep Reinforcement Learning

(Mnih 2016)

Q-Learnin
g

2018

Ape-X

Distributed Prioritized Experience Replay (Horgan 2018)

Policy Gradient

2018

IMPAL
A

IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures

(Espeholt 2018)

?

History of large scale distributed RL

2013 — 2015 — 2016 — 2018 — 2018 — ?

**DQ
N**
Playing Atari with Deep Reinforcement Learning
(Mnih 2013)

**GORIL
A**
Massively Parallel Methods for Deep Reinforcement Learning
(Nair 2015)

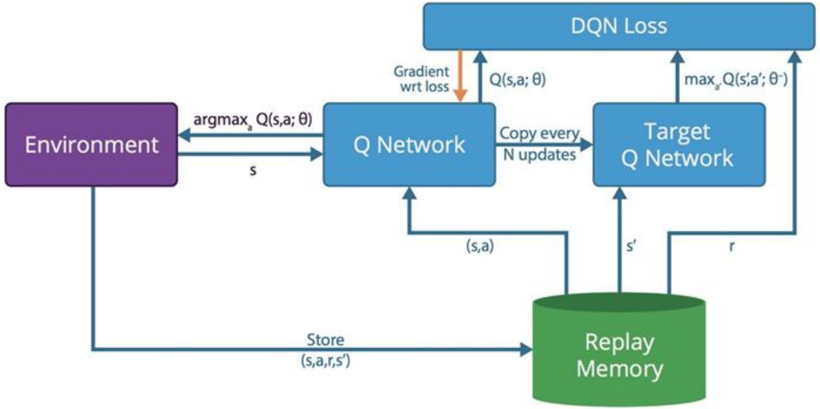
A3C
Asynchronous Methods for Deep Reinforcement Learning
(Mnih 2016)

Ape-X
Distributed Prioritized Experience Replay
(Horgan 2018)

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

2013/2015: DQN

•



1. (s_i, a_i, s_i', r_i) = env.step(a_i) $N\times$
   Store (s_i, a_i, s_i', r_i) in $\boldsymbol{B}$
2. Sample batch (s_j, a_j, s_j', r_j) from $\boldsymbol{B}$ $K\times$
   Update Q network
3. Update target network parameters: $\phi' \leftarrow \phi$

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., . . . Hassabis, D. (2015). Human-level Control Through Deep Reinforcement Learning. *Nature*

History of large scale distributed RL

2013 — 2015 — 2016 — 2018 — 2018 — ?

DQ
N
Playing Atari with Deep Reinforcement Learning
(Mnih 2013)

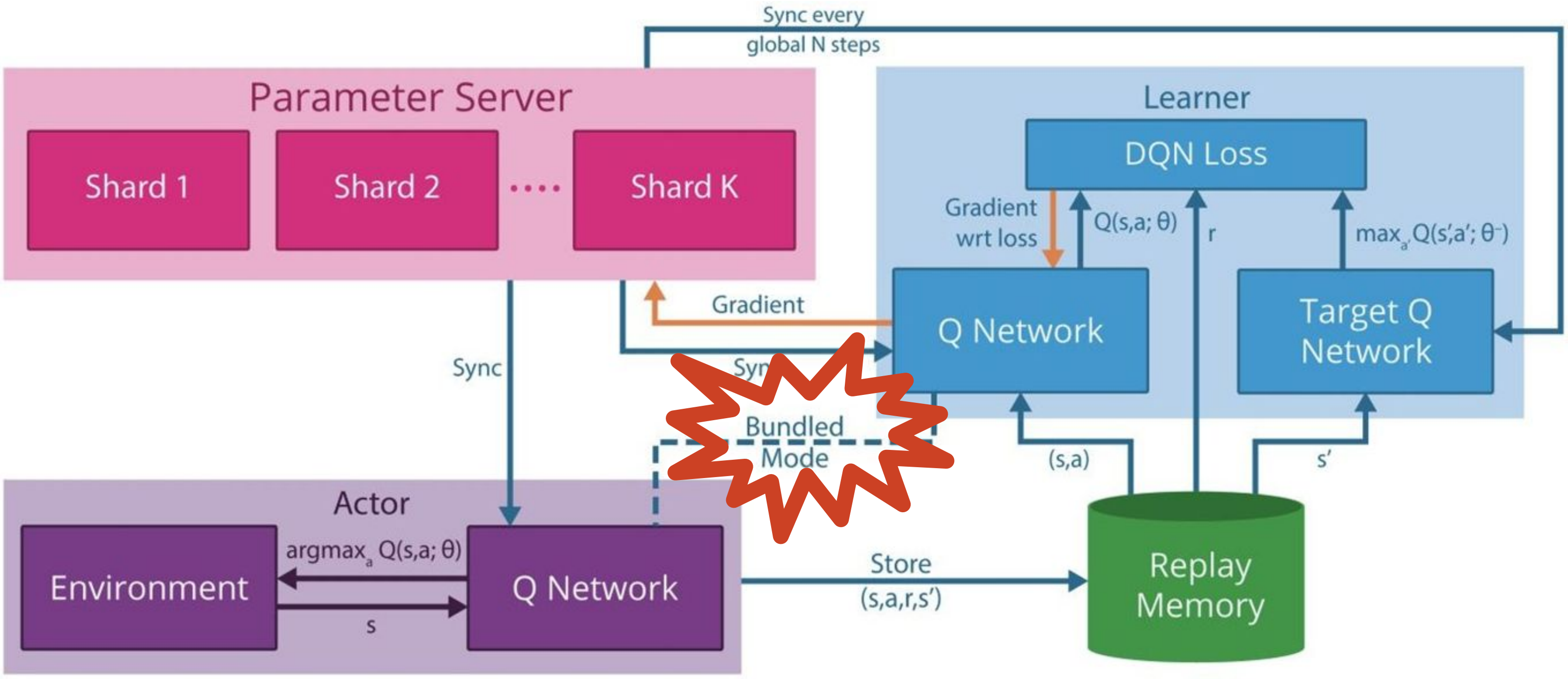
GORIL
A
Massively Parallel Methods for Deep Reinforcement Learning
(Nair 2015)

A3C
Asynchronous Methods for Deep Reinforcement Learning
(Mnih 2016)

Ape-X
Distributed Prioritized Experience Replay
(Horgan 2018)

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

2015: General Reinforcement Learning Architecture (GORILA)



Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., . . . Petersen, S. (2015). Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*.

2015: General Reinforcement Learning Architecture (GORILA)

- **Standard DQN**

  1. (s_i, a_i, s_i', r_i) = env.step(a_i)
     Store (s_i, a_i, s_i', r_i) in $\boldsymbol{B}$

  2. Sample batch $\left(s_j, a_j, s_j', r_j\right)$ from $\boldsymbol{B}$
     Update Q network

  3. Update target network parameters: $\phi' \leftarrow \phi$

- **Distributed DQN**

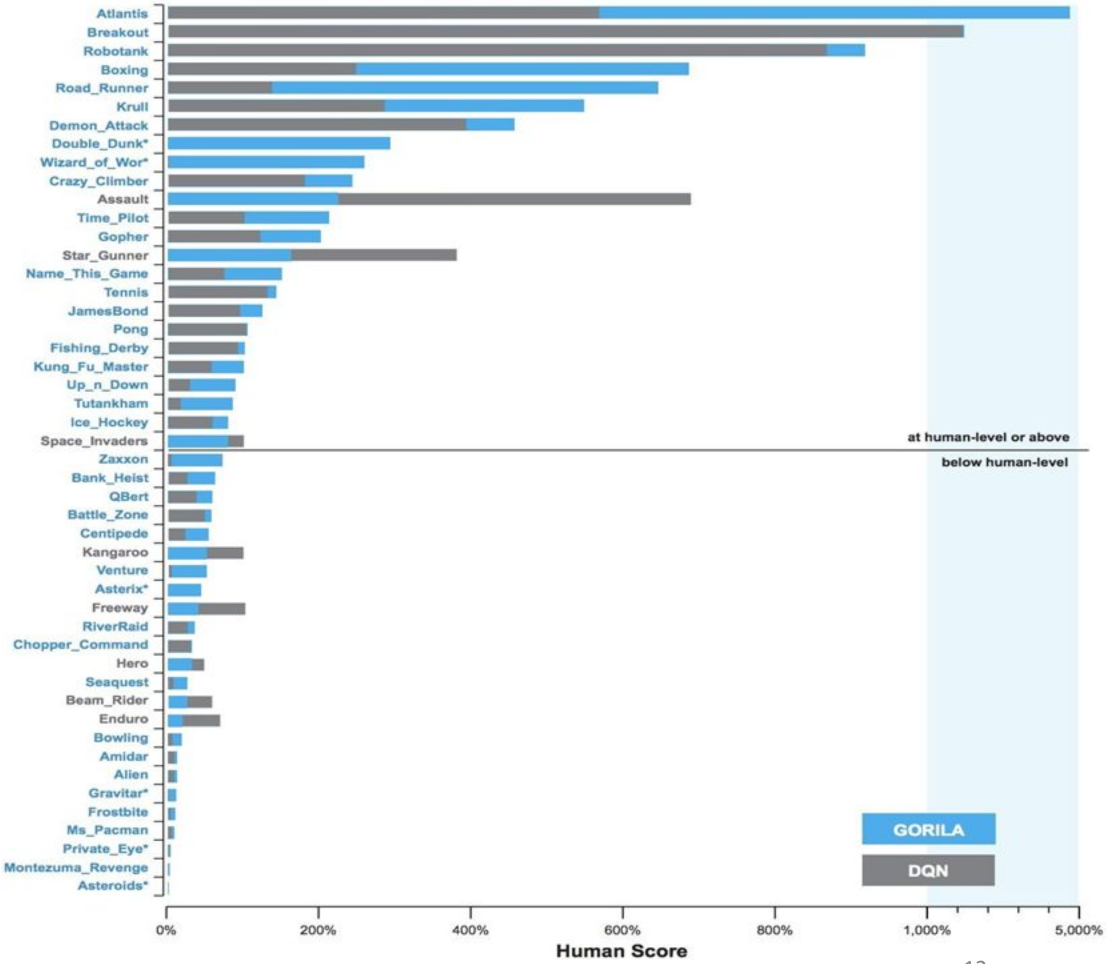
  Actor 1. (s_i, a_i, s_i', r_i) = env.step(a_i)
     Store (s_i, a_i, s_i', r_i) in $\boldsymbol{B}$

  Learner 2. Sample batch $\left(s_j, a_j, s_j', r_j\right)$ from $\boldsymbol{B}$
     Update θ with θ^+ from parameter server
     Calculate gradients w.r.t. θ
     Send gradients to parameter server

  Parameter Server 3. Update Q network

  Learner 4. Update target network parameters θ^- with θ^+ from the parameter server every N steps

GORILA Performance



History of large scale distributed RL

2013 — 2015 — 2016 — 2018 — 2018 — ?

DQ
N
Playing Atari with Deep Reinforcement Learning
(Mnih 2013)

GORIL
A
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)

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

Prioritised Experience Replay

Standard DQN

1. (s_i, a_i, s_i', r_i) = env.step(a_i)
   Store (s_i, a_i, s_i', r_i) in $\boldsymbol{B}$

2. Sample K transitions (s_j, a_j, s_j', r_j) **uniformly**
   from $\boldsymbol{B}$
   Update Q network

3. Update target network parameters: $\phi' \leftarrow \phi$

Prioritised Experience Replay

1. (s_t, a_t, s_t', r_t) = env.step(a_t)
   Store(s_t, a_t, s_t', r_t) in $\boldsymbol{B}$ with max priority $p_t = \max_{i<t} p_i$

2. Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$ from $\boldsymbol{B}$
   Compute TD error δ_j
   Update transition probability $p_j \leftarrow |\delta_j|$

   $K \times$

3. Update Q network

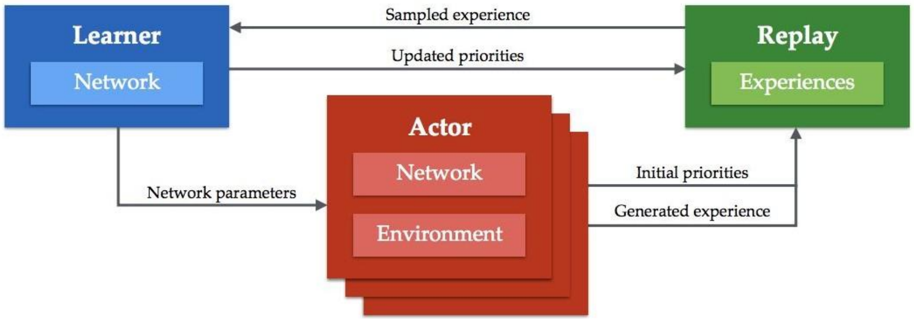
4. Update target network parameters: $\phi' \leftarrow \phi$

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). *Prioritized Experience Replay*.

Distributed Prioritized Experience Replay (Ape-X)

Prioritised Experience Replay

1. (s_i, a_i, s_i', r_i) = env.step(a_i)



3. Update Q network

4. Update target network parameters: $\phi' \leftarrow \phi$

Ape-X

Actors

- (s_i, a_i, s_i', r_i) = env.step(a_i)
- Compute TD error δ_j
  Update transition probability $p_j \leftarrow |\delta_j|$
- Store (s_i, a_i, s_i', r_i) in $\boldsymbol{B}$

Learners

- Update θ with θ^+ from parameter server
- Sample transition (s_j, a_j, s_j', r_j) from $\boldsymbol{B}$
- Calculate gradients w.r.t. θ
- Update parameters θ of Q-network
- Compute TD error δ_j
  Update transition probability $p_j \leftarrow |\delta_j|$
- Update target network parameters θ^-
  with θ^+ from the parameter server every N steps

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., & Silver, D. (2018). Distributed Prioritized Experience Replay

Distributed Prioritized Experience Replay (Ape-X)

Gorila

Actors

- (s_i, a_i, s_i', r_i) = env.step(a_i)
  Store (s_i, a_i, s_i', r_i) in $\boldsymbol{B}$

Learners

- Update θ with θ^+ from parameter server
- Sample batch (s_j, a_j, s_j', r_j) from $\boldsymbol{B}$
- Calculate gradients w.r.t. θ
- Send gradients to parameter server
- Update target network parameters θ^- with θ^+ from the parameter server every N steps

Parameter Server

- Update parameters θ of Q network

Ape-X

Actors

- (s_i, a_i, s_i', r_i) = env.step(a_i)
- Compute TD error δ_j
  Update transition probability $p_j \leftarrow |\delta_j|$
- Store (s_i, a_i, s_i', r_i) in $\boldsymbol{B}$

Learners

- Update θ with θ^+ from parameter server
- Sample transition (s_j, a_j, s_j', r_j) from $\boldsymbol{B}$
- Calculate gradients w.r.t. θ
- Update parameters θ of Q-network
- Compute TD error δ_j
  Update transition probability $p_j \leftarrow |\delta_j|$
- Update target network parameters θ^- with θ^+ from the parameter server every N steps

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., & Silver, D. (2018). Distributed Prioritized Experience Replay

Ape-X Performance

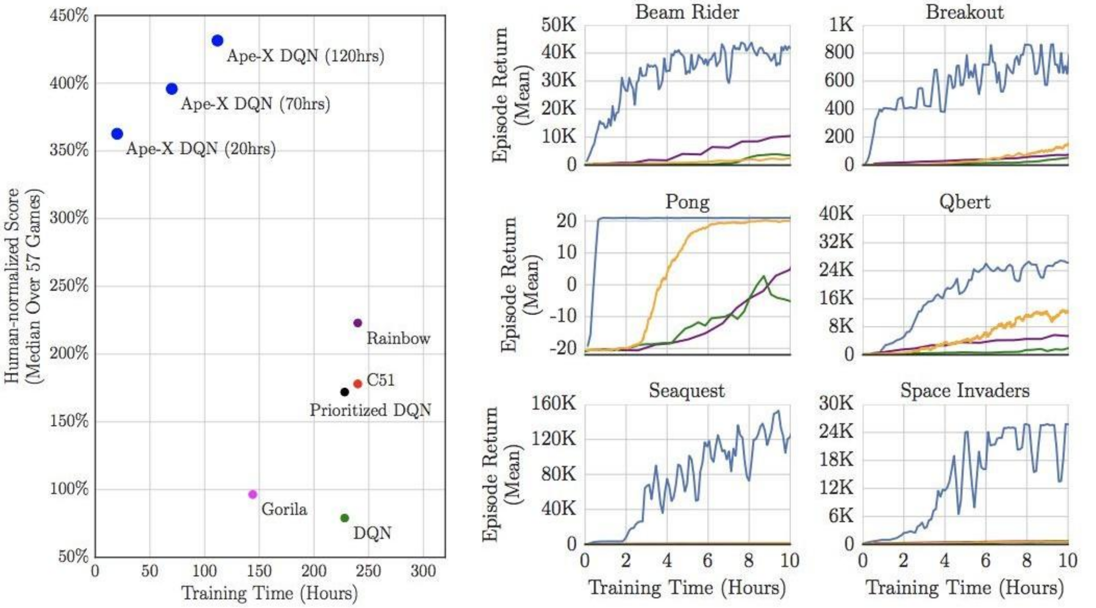


Figure 2: Left: Atari results aggregated across 57 games, evaluated from random no-op starts. Right: Atari training curves for selected games, against baselines. Blue: Ape-X DQN with 360 actors; Orange: A3C; Purple: Rainbow; Green: DQN. See appendix for longer runs over all games.

History of large scale distributed RL

2013 — 2015 — 2016 — 2018 — 2018 — ?

DQ
N
Playing Atari with Deep Reinforcement Learning
(Mnih 2013)

GORIL
A
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)

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

Recap: Online actor-critic

-

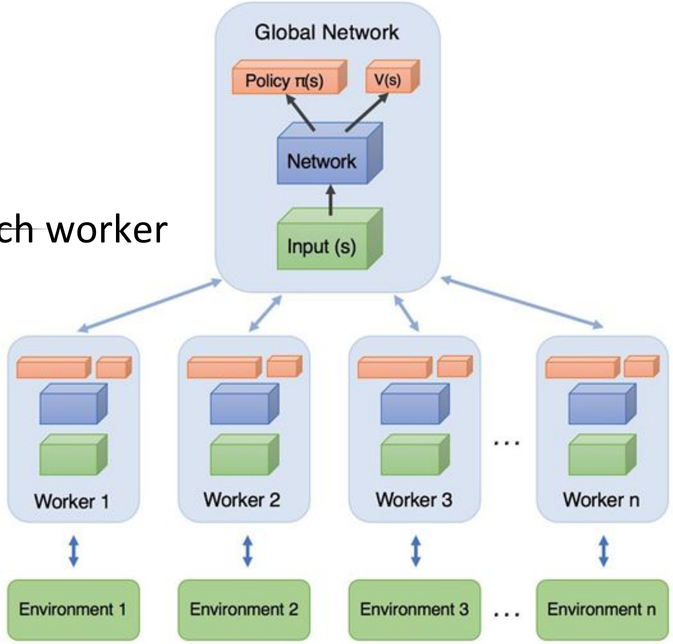
1. Take action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
2. Update $\hat{V}_\phi^\pi$ using target $r + \hat{V}_\phi^\pi(s')$
3. Evaluate $\hat{A}^\pi(s, a) = r(s, a) + \gamma \hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s)$
4. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) \hat{A}^\pi(s, a)$
5. Update policy
   $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Asynchronous advantage actor-critic (A3C)

1. Sync weights θ and ϕ from master
2. Take action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
3. Compute gradient of $\hat{V}_\phi^\pi$ using target $r + \hat{V}_\phi^\pi(s')$
4. Evaluate $\hat{A}^\pi(s,a) = r(s,a) + \gamma \hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s)$
5. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) \hat{A}^\pi(s,a)$

Each worker

1. Update policy: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
2. Update $\hat{V}_\phi^\pi$



Each has different exploration -> more diverse samples!

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., . . . Kavukcuoglu, K. (2016). *Asynchronous Methods for Deep Reinforcement Learning*. Paper presented at the Proceedings of The 33rd International Conference on Machine Learning

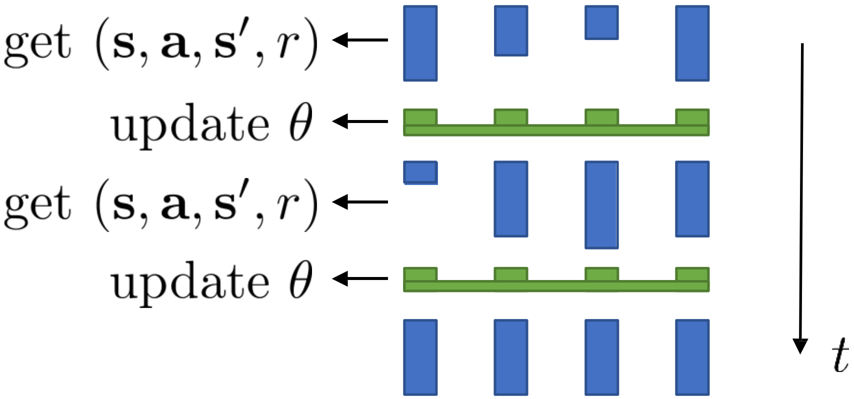
Asynchronous advantage actor-critic (A3C)

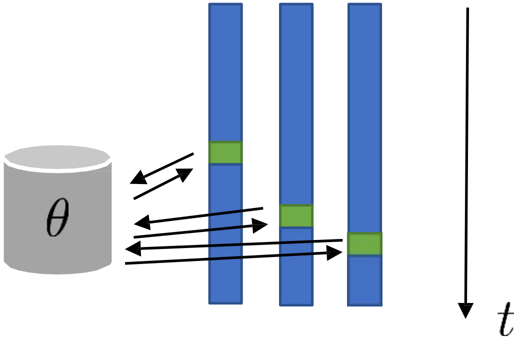
A2C

- can lead to low GPU utilisation due to rendering time variance within a batch

A3C

- decouples acting from learning





Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., . . . Kavukcuoglu, K. (2016). *Asynchronous Methods for Deep Reinforcement Learning*. Paper presented at the Proceedings of The 33rd International Conference on Machine Learning

Asynchronous advantage actor-critic (A3C)

- Some extra features:
- n-step estimation: $\hat{A}^{\pi}(s,a) = \sum_{i=0}^{k-1} \gamma^{i}\, r(s_t, a_t) + \gamma^{k} \hat{V}_{\phi}^{\pi}(s_{t+k}) - \hat{V}_{\phi}^{\pi}(s_t)$
- Entropy of the policy π_θ was added to the objective function to improve exploration:

$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) \hat{A}^{\pi}(s,a) + \beta \nabla_\theta H\big(\pi_\theta(s)\big)$$

A3C Performance

Changes to GORILA:

1. **Faster updates**
2. **No** replay buffer
3. **Actor-critic**

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

History of large scale distributed RL

2013 — 2015 — 2016 — 2018 — 2018 — ?

**DQ
N**

Playing Atari with Deep Reinforcement Learning
(Mnih 2013)

**GORIL
A**

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)

**IMPAL
A**

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

Importance Weighted Actor-Learner Architectures (IMPALA)

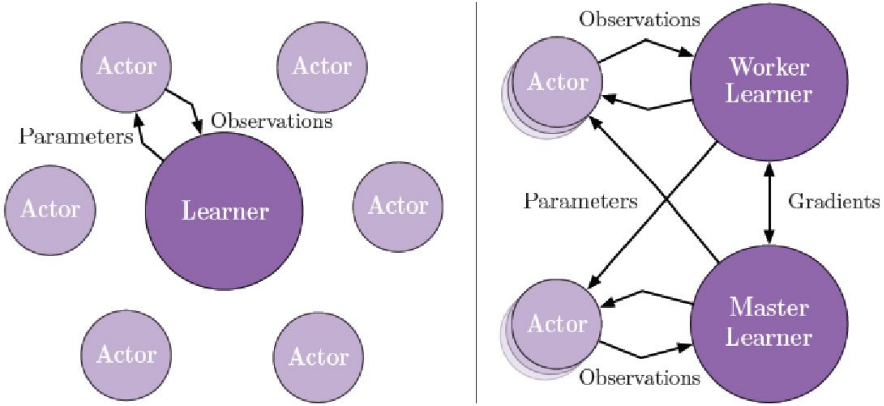


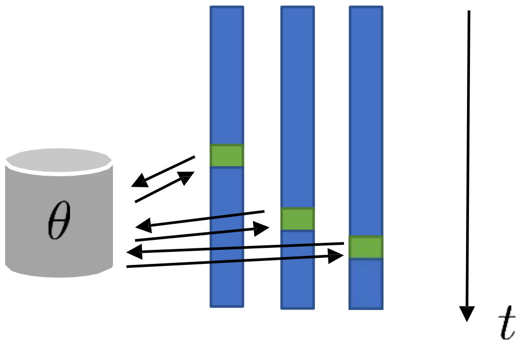
Figure 1. **Left: Single Learner.** Each *actor* generates trajectories and sends them via a queue to the *learner*. Before starting the next trajectory, *actor* retrieves the latest policy parameters from *learner*. **Right: Multiple Synchronous Learners.** Policy parameters are distributed across multiple *learners* that work synchronously.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., . . . Kavukcuoglu, K. (2018). *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. Paper presented at the Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research. http://proceedings.mlr.press

How to correct for Policy Lag? Importance Sampling!

Shortcoming of A3C:

- *Policy-lag*



Apply importance sampling:

1. to policy gradient

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

2. to critic update

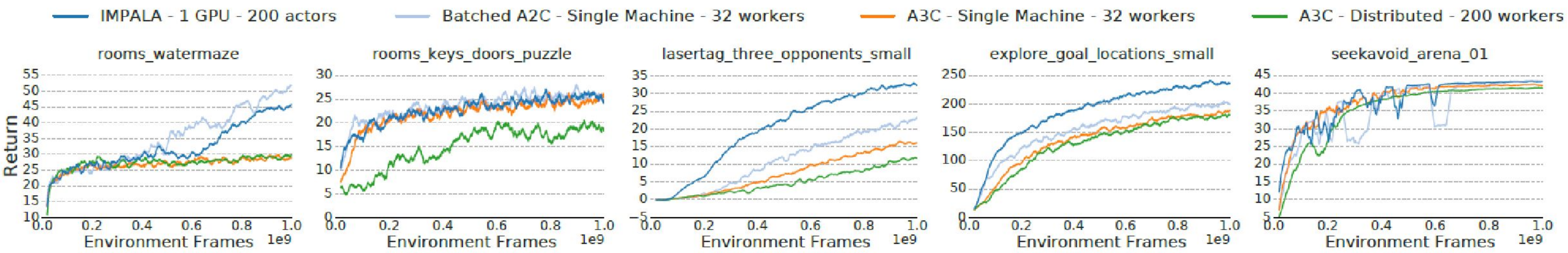
4.1. V-trace target

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:

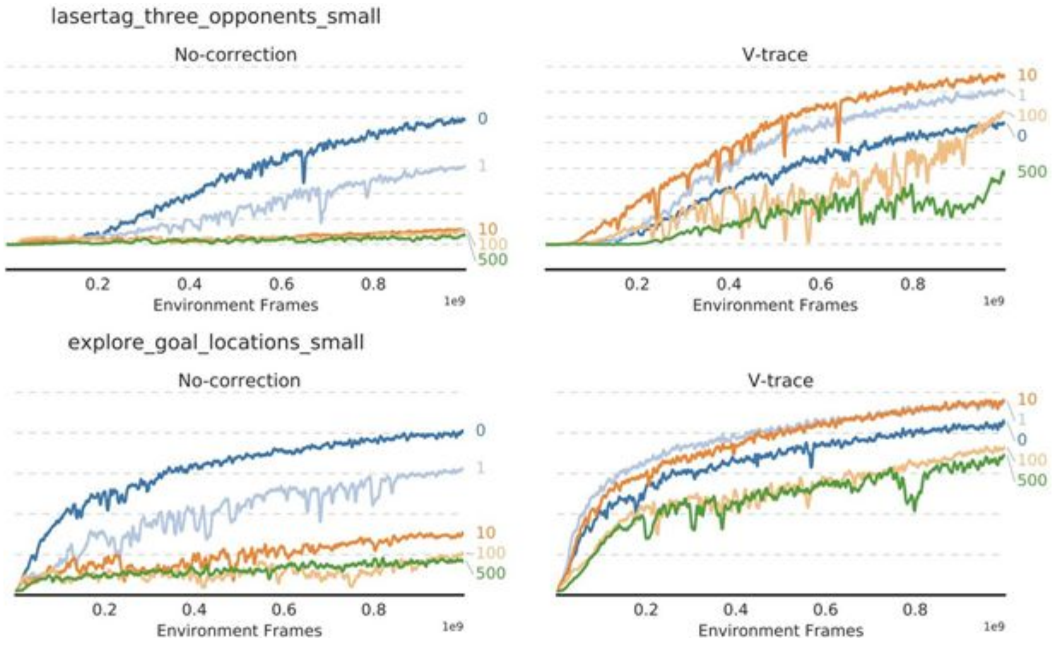
$$v_s \stackrel{\text{def}}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \Big(\prod_{i=s}^{t-1} c_i\Big) \delta_t V, \quad (1)$$

IMPALA - Performance

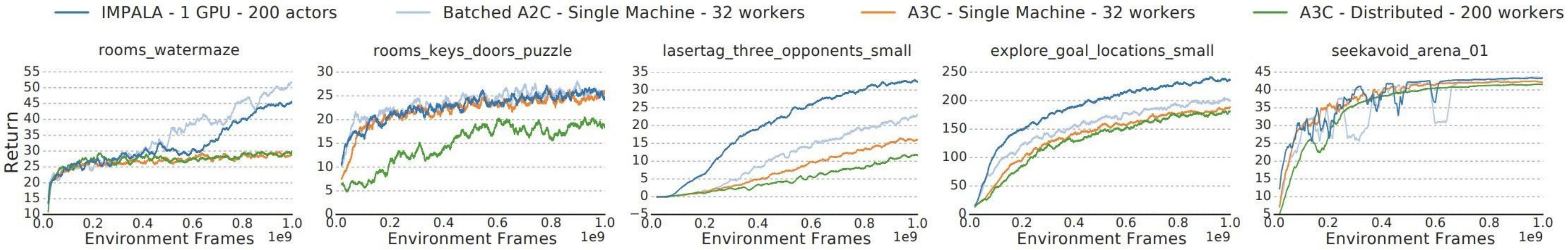
A comparison between IMPALA, A3C and batched A2C



IMPALA - Performance



IMPALA Performance



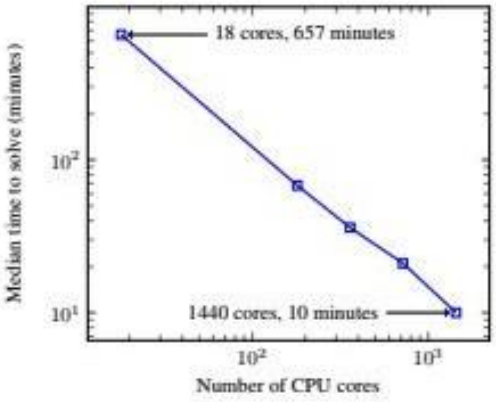
Evolution Strategies

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 $t = 0, 1, 2, \ldots$ **do**
  for each worker $i = 1, \ldots, n$ **do**
    Sample $\epsilon_i \sim \mathcal{N}(0, I)$
    Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
  end for
  Send all scalar returns F_i from each worker to every other worker
  for each worker $i = 1, \ldots, n$ **do**
    Reconstruct all perturbations ϵ_j for $j = 1, \ldots, n$ using known random seeds
    Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
  end for
end for



Summary

Algorithm	Policy Evaluation	Gradient-based optimizer	CPU	GPU	Replay Buffer	Prioritised Replay	Parameter Server	Importance Sampling
DQN	X	X	1	1	X			
Gorila	X	X			X		X	
Ape-X	X	X			X	X		
A3C	X	X	many	0				
Impala	X	X	many					X

Lesson Objectives

1. Why parallelise?
2. Understand how the computation of standard RL algorithms can be distributed to decrease wall-clock training time.
3. **How these distributed RL algorithms can be modularised.**
4. **How modularised distributed RL algorithms can be implemented on real systems - case study: RLlib**
5. Examples on using RLlib

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

http://rllib.io

RL research scales with compute

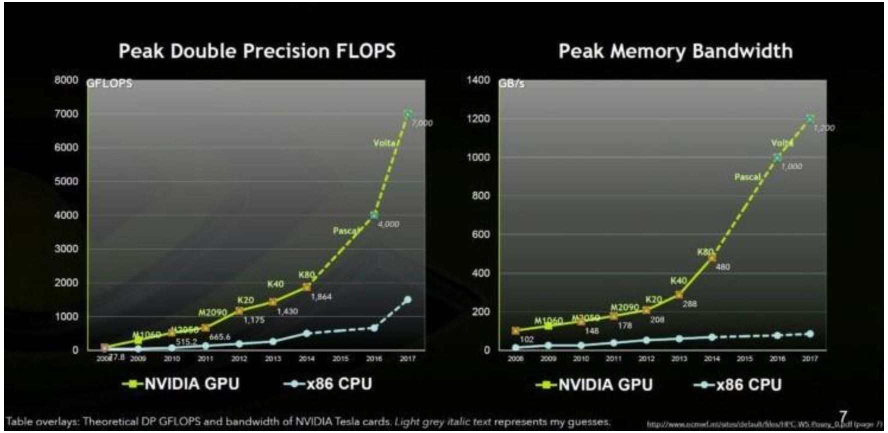


Fig. courtesy NVidia Inc.

CPU

GPU

TPU

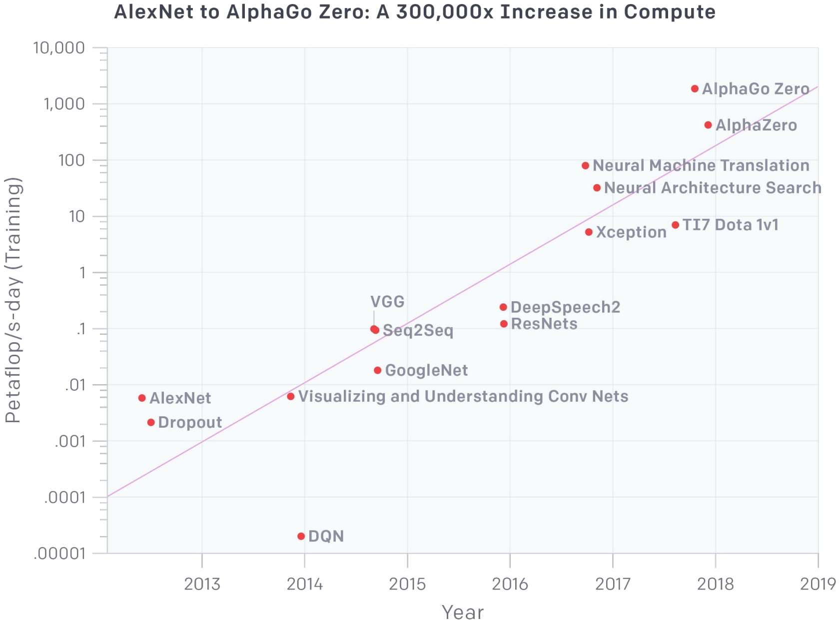
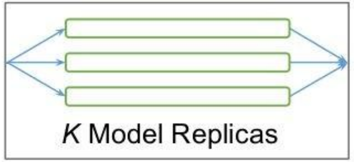
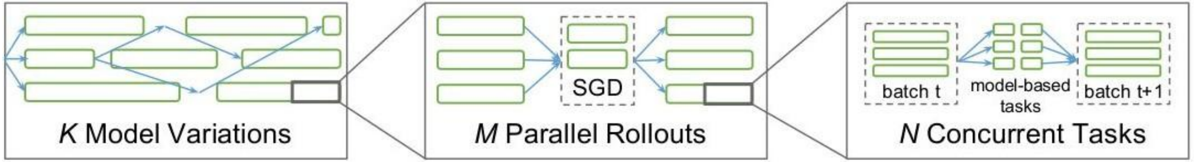


Fig. courtesy OpenAI

How do we leverage this hardware?



(a) Supervised Learning



(b) Reinforcement Learning

scalable abstractions for RL?

Systems for RL today

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

    PPO: multiprocessing, MPI

    AlphaZero: custom systems

    Evolution Strategies: Redis

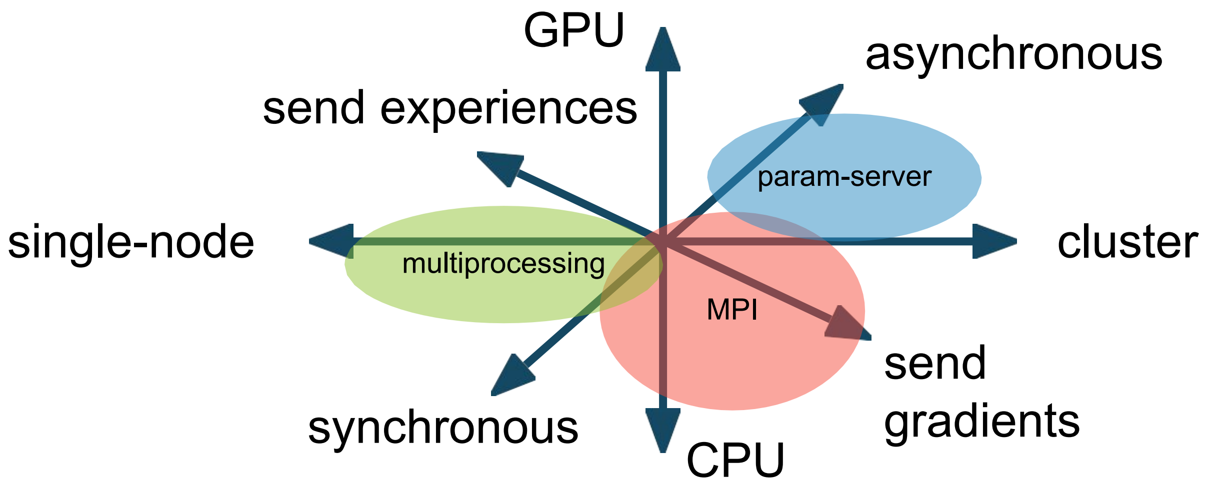
    IMPALA: Distributed TensorFlow

    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

[redacted]

We need abstractions for RL

Good abstractions decompose RL algorithms into reusable components.

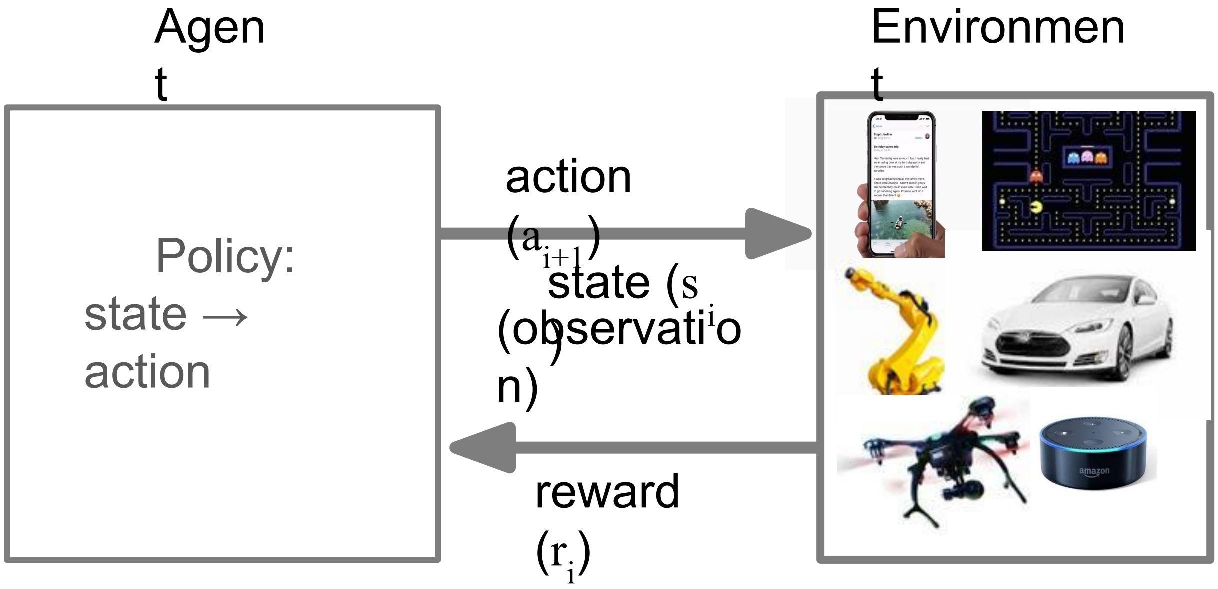
Goals:

– Code reuse across deep learning frameworks

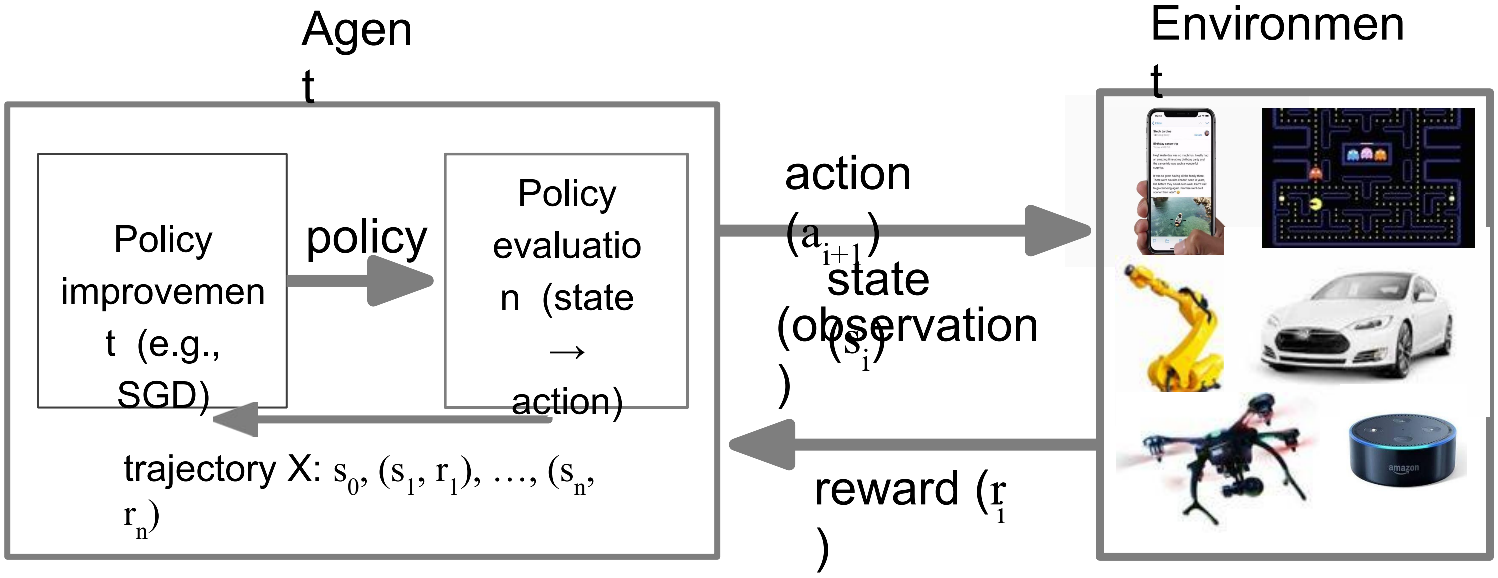
– Scalable execution of algorithms

– Easily compare and reproduce algorithms

Structure of RL computations

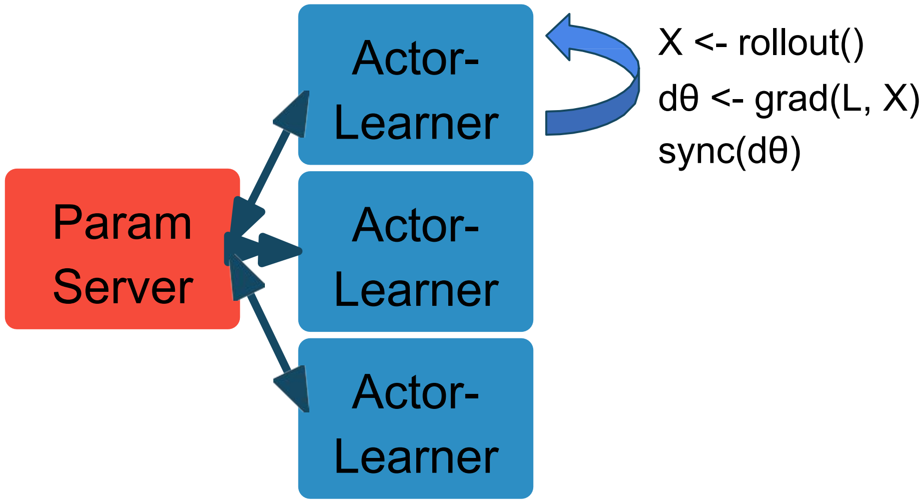


Structure of RL computations

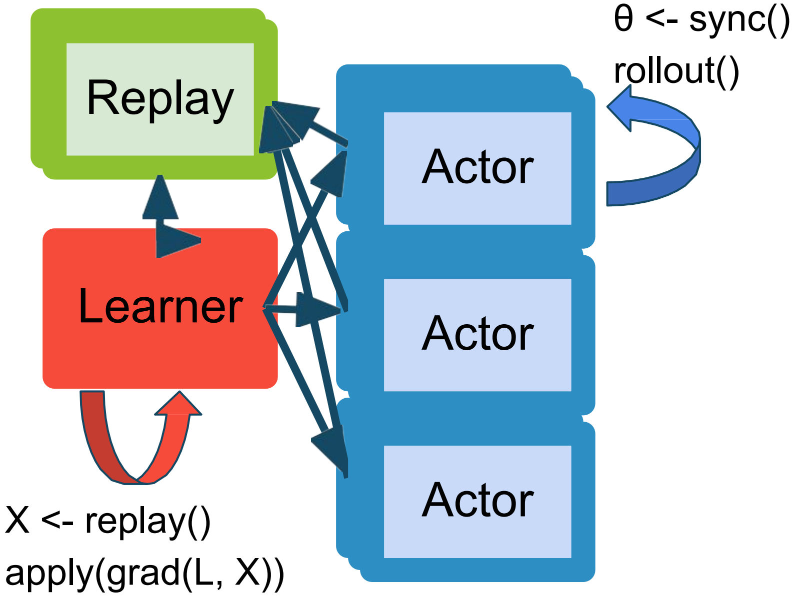


Many RL loop decompositions

Async DQN (Mnih et al; 2016)

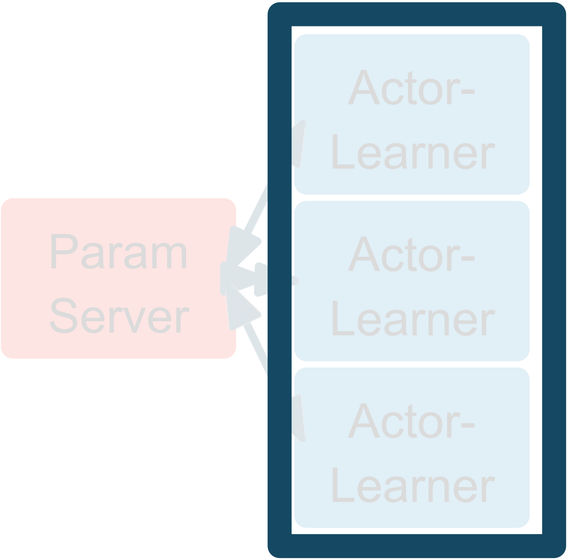


Ape-X DQN (Horgan et al; 2018)

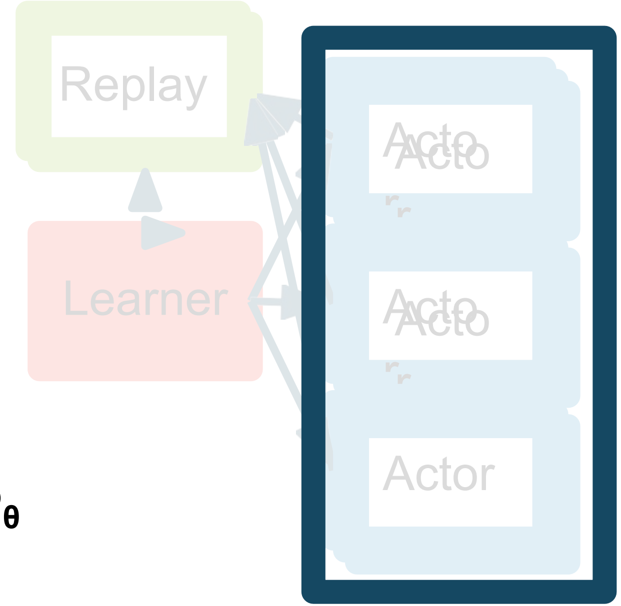


Common components

Async DQN (Mnih et al; 2016)



Ape-X DQN (Horgan et al; 2018)



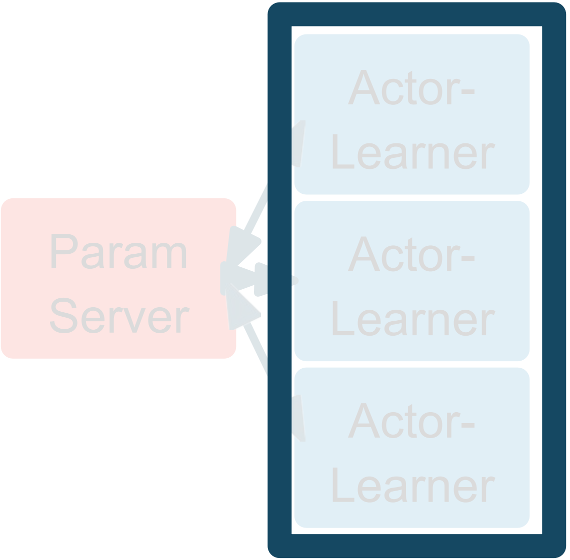
Policy $\pi_\theta(o_t)$

Trajectory postprocessor $\rho_\theta(X)$

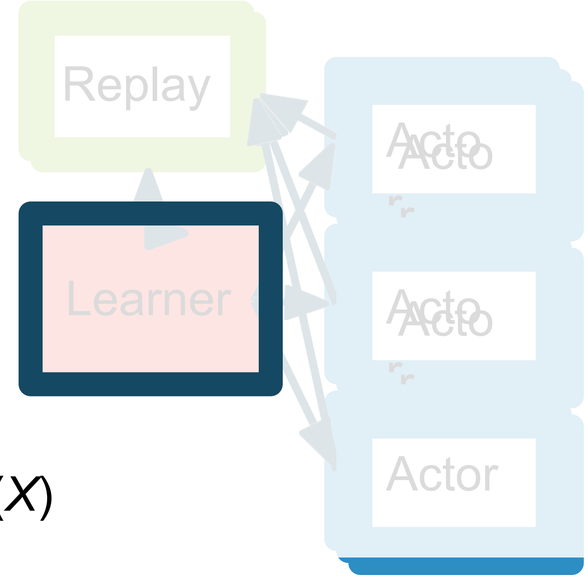
Loss $L(\theta, X)$

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 $\boldsymbol{L(\theta,X)}$

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

...and this is just one family!

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

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

- Nested parallel computations
- Control decisions based on intermediate results

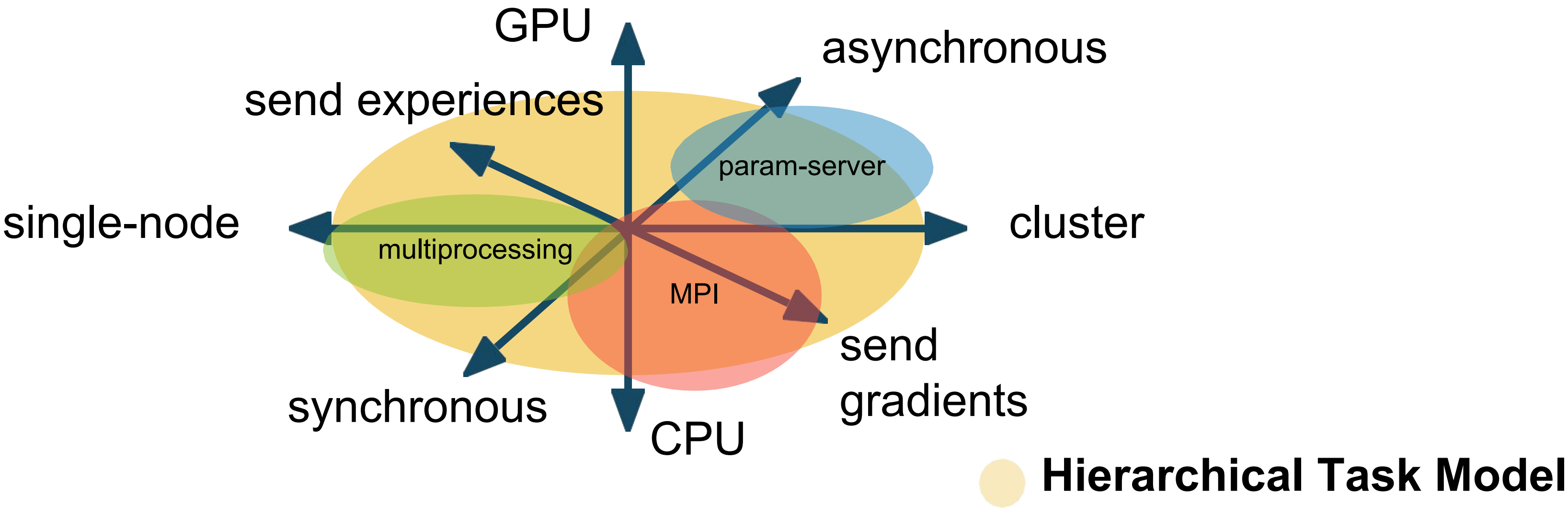
Requirements for a new system

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

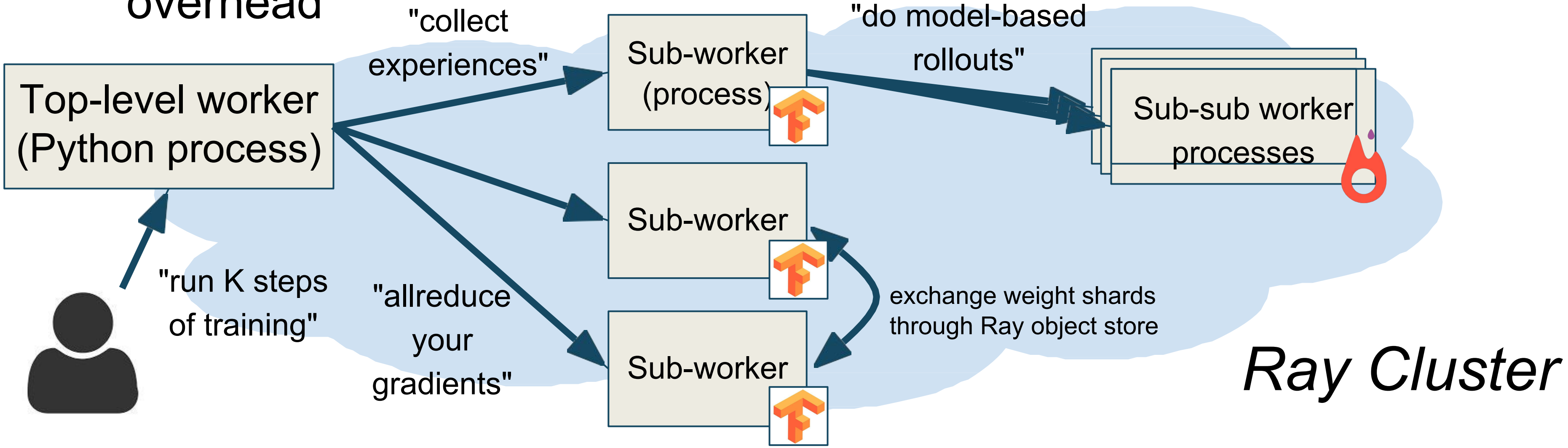
Ray System Substrate

- 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.)

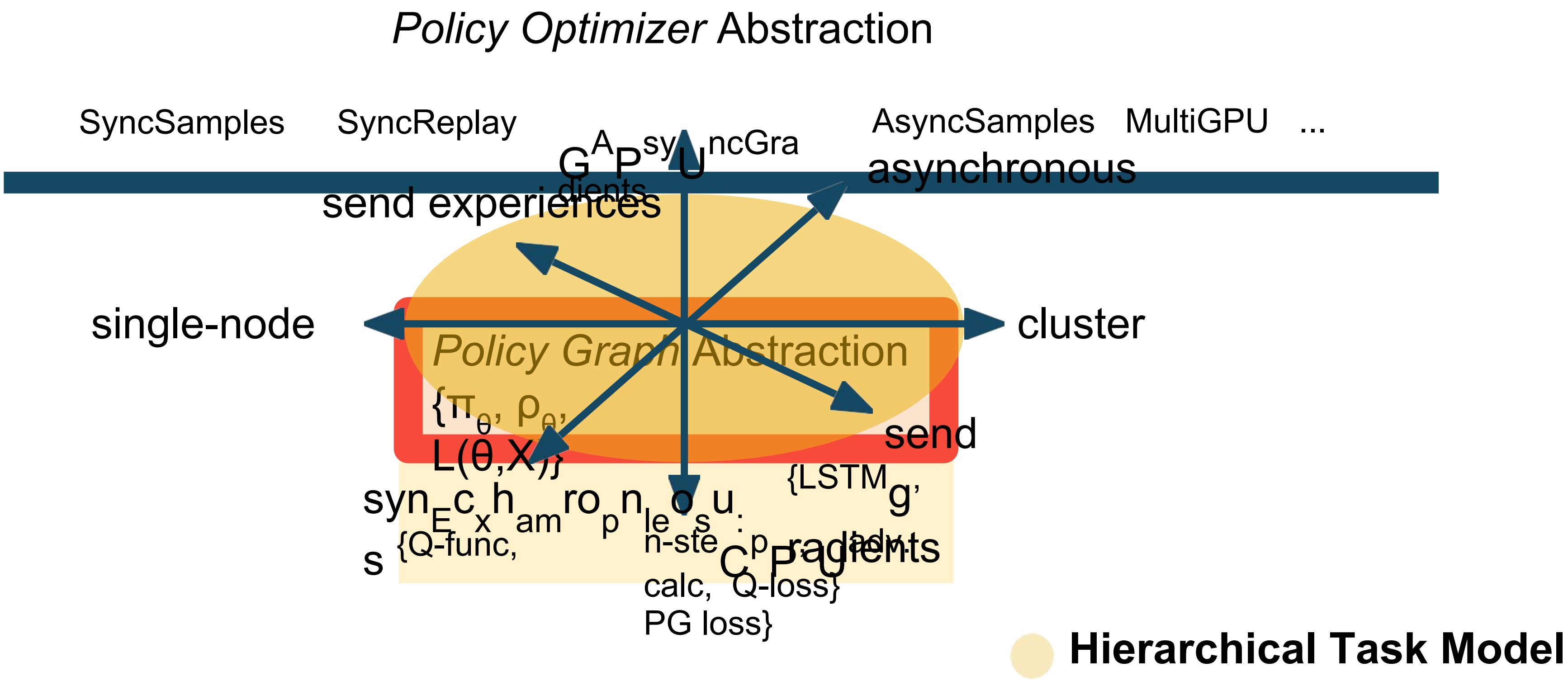


Hierarchical Parallel Task Model

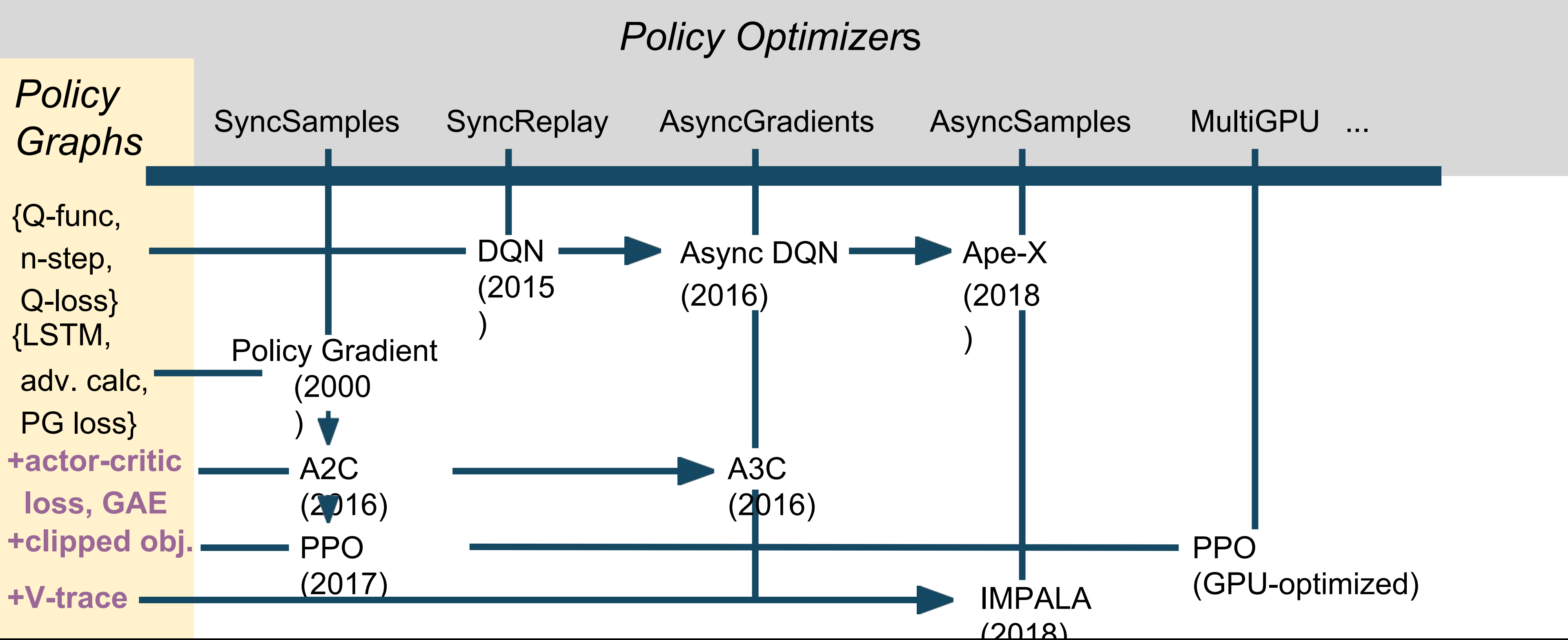
1. Create Python class instances in the cluster (stateful workers)
2. Schedule short-running tasks onto workers
   – Challenge: High performance: 1e6+ tasks/s, ~200us task overhead



Unifying system enables RL Abstractions



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

FLOW Lab

Community Contributions

Alibaba Group

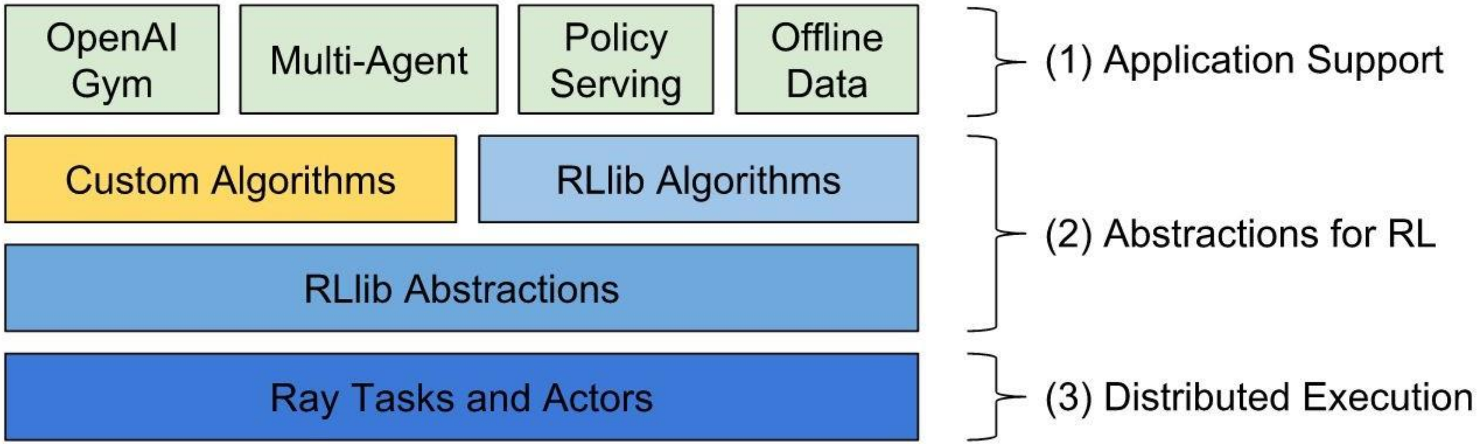
RLlib Reference Algorithms

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

Tutorial on google Colab:
https://drive.google.com/open?id=1pvE7KvnhYR0Ynqt0J0fzYSmkjlLg64Qq

```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

```
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())
```

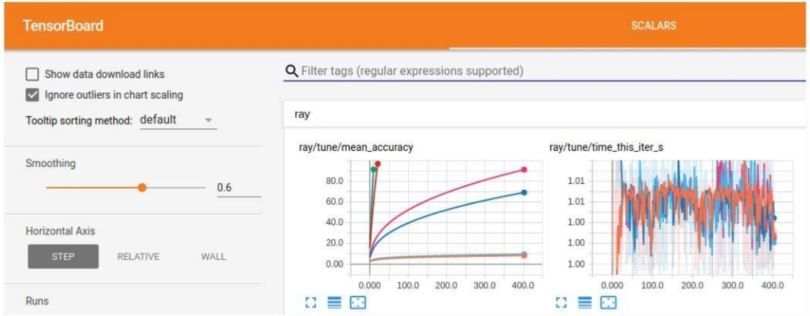
Code example: hyperparam tuning

```
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
```



http://rllib.io

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

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

Thanks!