Lecture 18
System Design for Distributed RL

Bolei Zhou
The Chinese University of Hong Kong

Today’s Outline

- Parallelism in distributed ML systems
- Development of the distributed RL systems
- Case studies on the system designs of AlphaGo, OpenAI Five, and AlphaStar

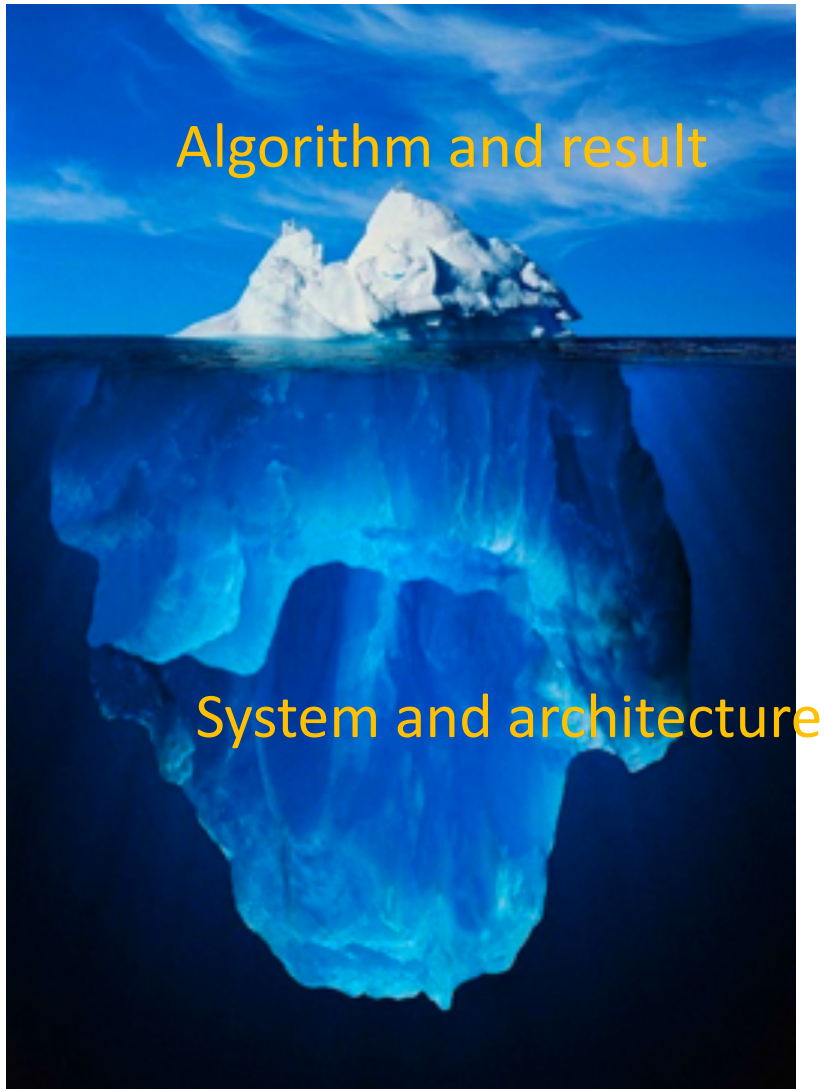
Distributed ML Systems

PhD/Master thesis project
Small data set

Scale up

Massive data set

System and architecture are the foundation

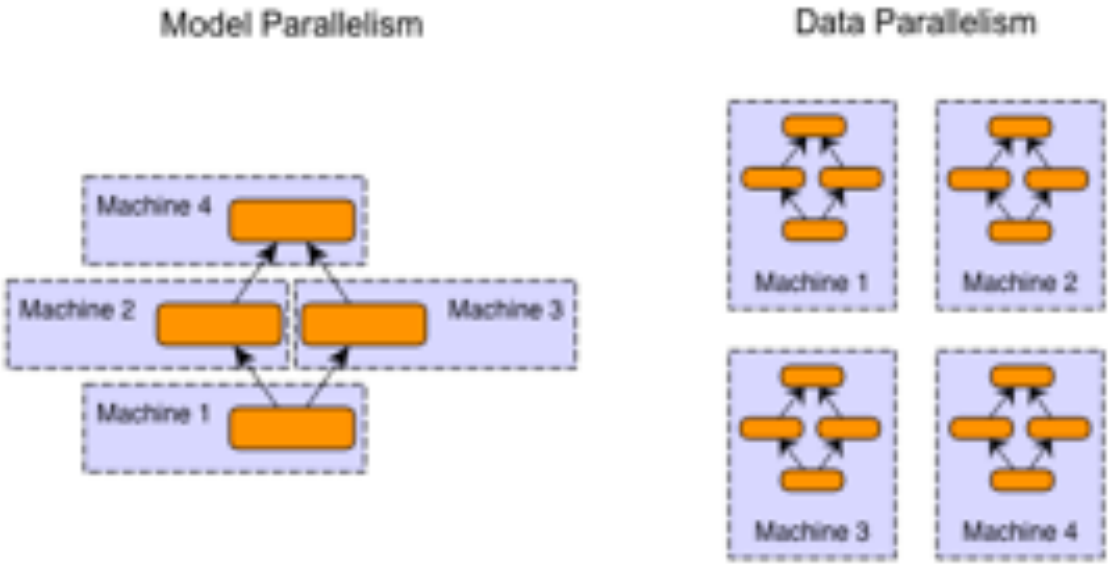


Properties of Distributed ML Systems

- **Consistency:** ensure the consensus of multiple nodes if they are simultaneously working toward one goal
- **Fault tolerance:** distribute workload to a cluster of 1000 nodes, what if one of the 1000 nodes crashes
- **Communication:** ML involves a lot of I/O, distributed file systems, CPU I/O, GPU I/O

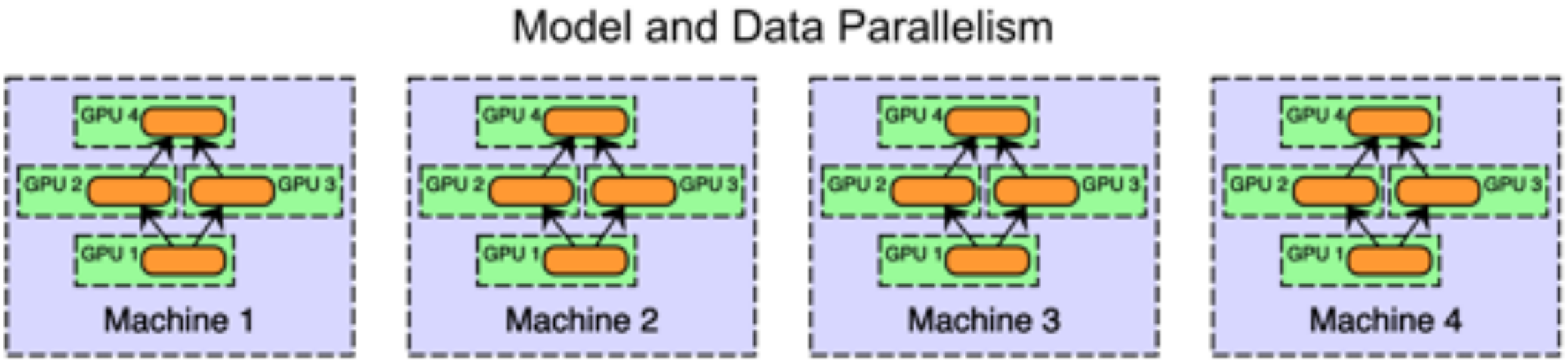
Parallelism in Distributed ML Systems

- **Model parallelism**: different machines are responsible for computation in different parts of a single network
- **Data parallelism**: different machine has a complete copy of the model, each machine gets a different portion of the data



Parallelism in Distributed ML Systems

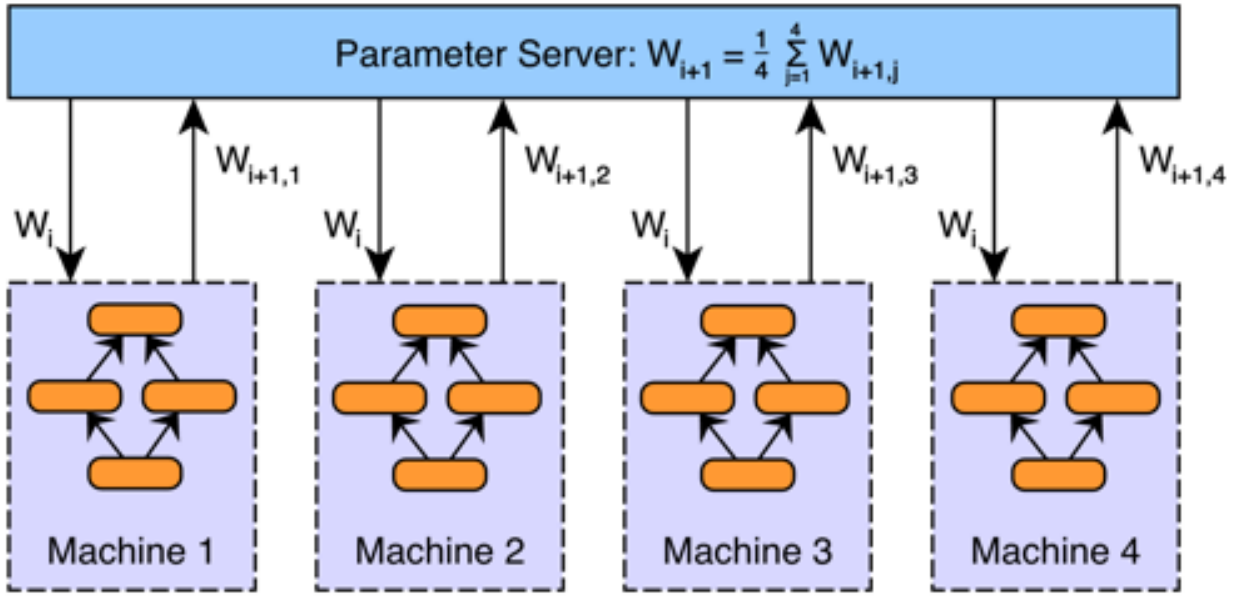
- A cluster of multi-GPU systems
- Model parallelism on GPUs, data parallelism between machines



Updating Model Parameters

Parameter averaging

- Distribute a copy of current parameters to each worker
- Train each worker on a subset of the data
- Set the global parameters to the average of the parameters from each worker
- Repeat

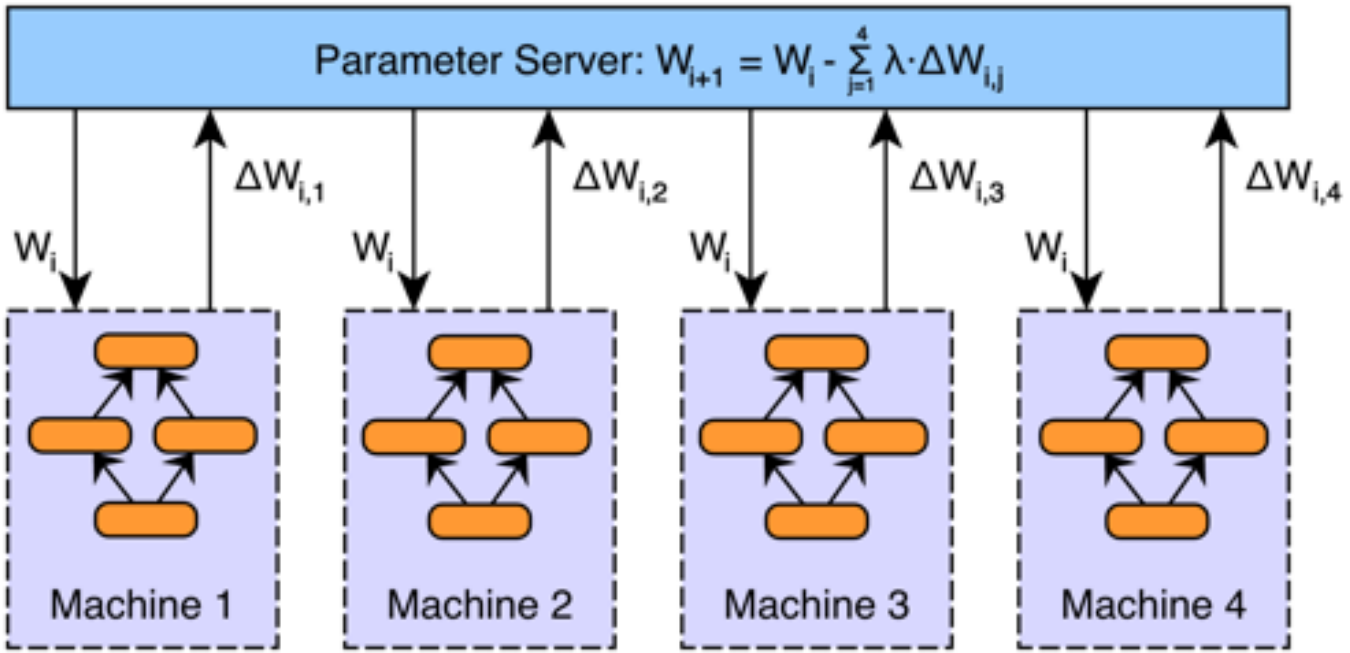


Updating Model Parameters

Distributed Stochastic Gradient Descend:

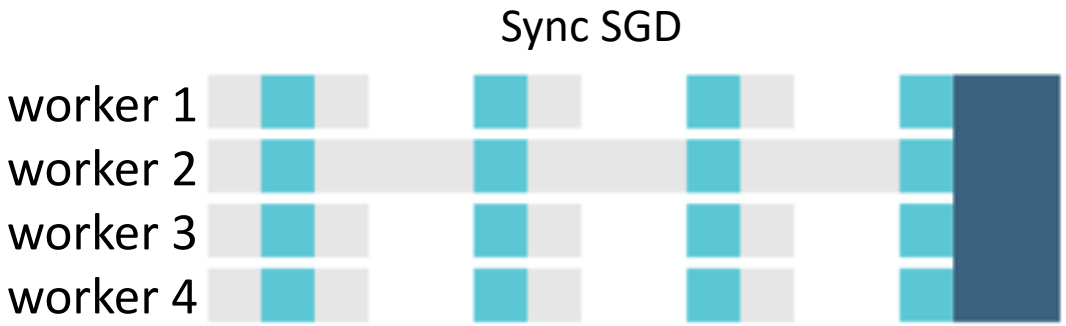
- Transfer updates (gradient) to the parameter server

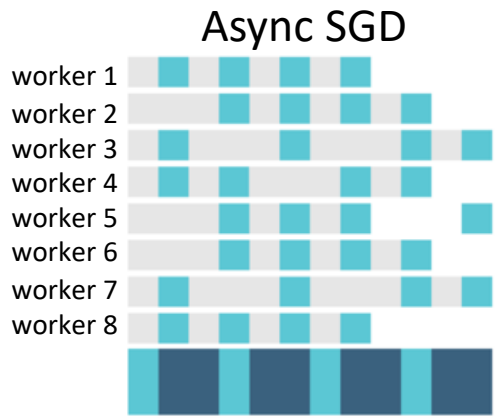
$$W_{i+1} = W_i - \lambda \sum_{j=1}^{N} \Delta W_{i,j}$$



Synchronous Update versus Asynchronous Update

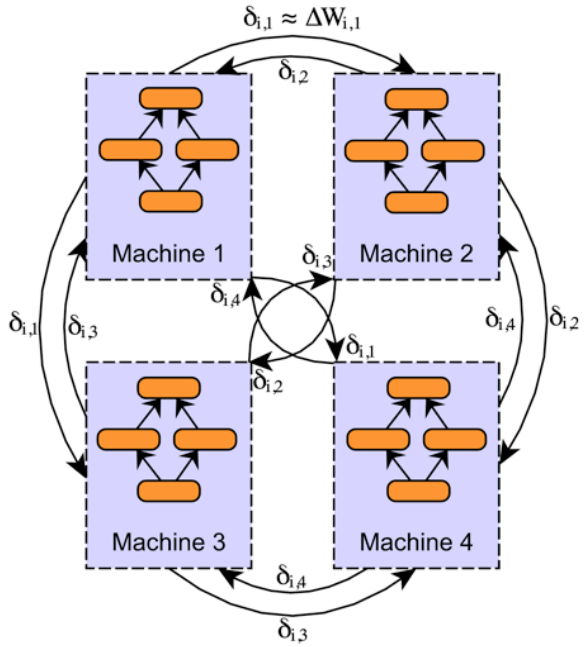
- Synchronous update:
  - The agent are trained in parallel, but the updates are collected and then enforced on the global network at one time
  - Has to wait until every agent finish returning the update in one episode
- Asynchronous update:
  - The global network is periodically updated by individual worker
  - The updates don’t happen simultaneously





Decentralized Asynchronous Stochastic Gradient Descend

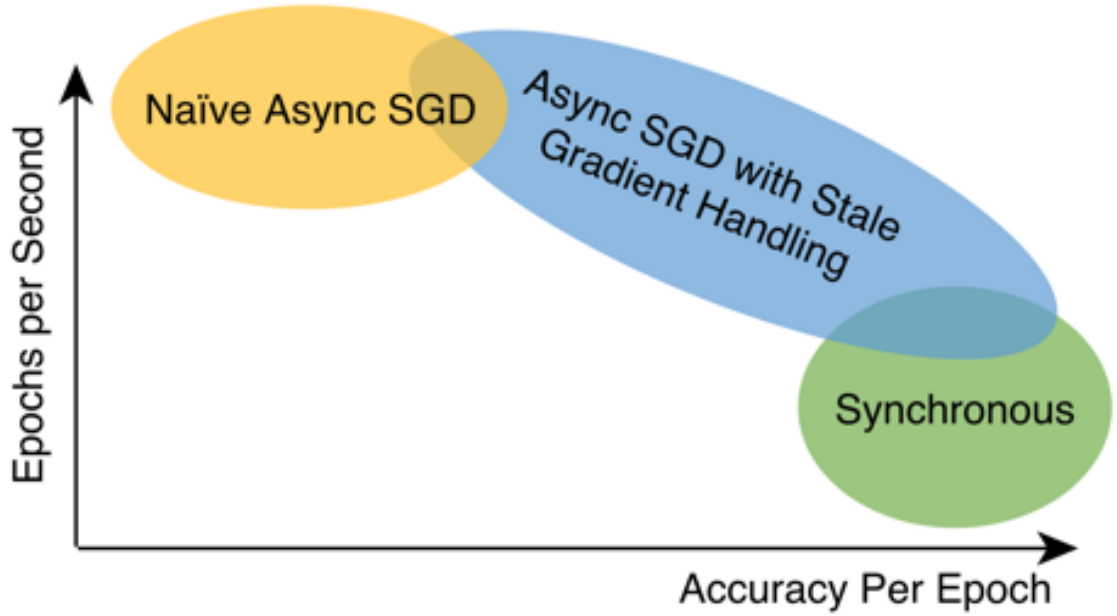
- In previous setup, the parameter server might be fragile
- No centralized parameter server is present
- Peer to peer communication to transmit compressed model updates

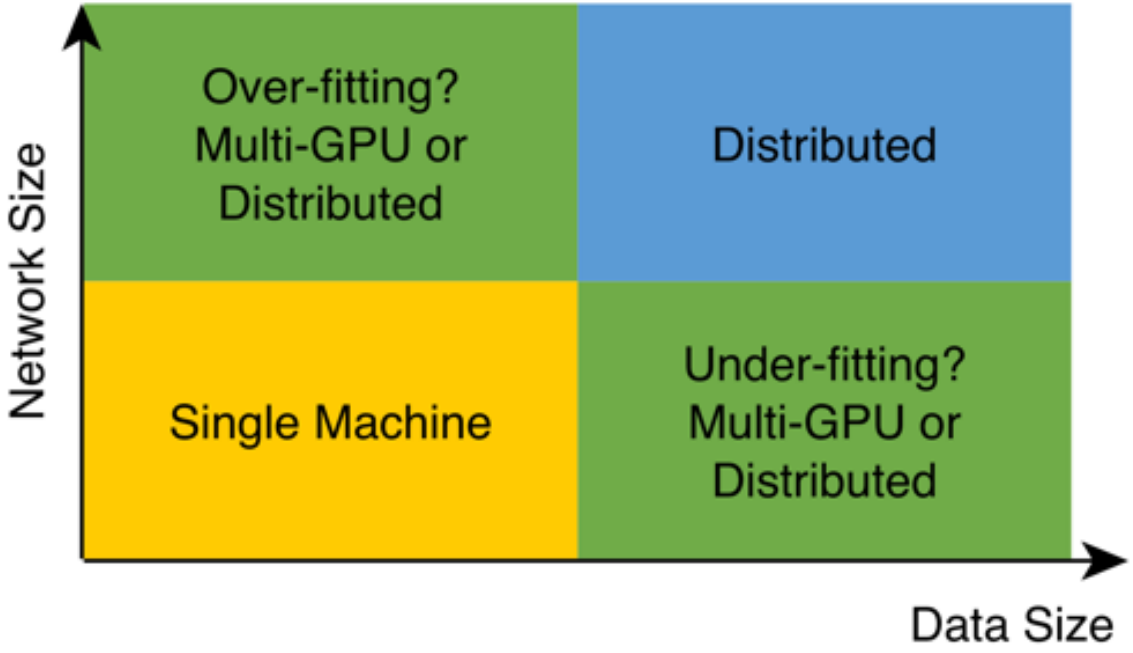


Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In Sixteenth Annual Conference of the International Speech Communication Association, 2015.

Parallelism in Distributed ML Systems

- Balance between parallelism and the performance





Hogwild: Lock-free asynchronous SGD

Synchronous stochastic gradient descent with locks

- Each thread draws a random example i from the training data.
  - Acquire a lock on the current state of parameters θ.
  - Thread reads θ.
  - Thread updates $\theta \leftarrow (\theta - \alpha \nabla L(f_\theta(x_i), y_i))$.
  - Release lock on θ.

Asynchronous stochastic gradient descent

Each thread draws a random example i from the training data.

- Thread reads current state of θ.
- Thread updates $\theta \leftarrow (\theta - \alpha \nabla L(f_\theta(x_i), y_i))$.

Take advantage of the sparsity in machine learning problems, theoretic analysis on the convergence

Feng et al. NIPS'11

https://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf

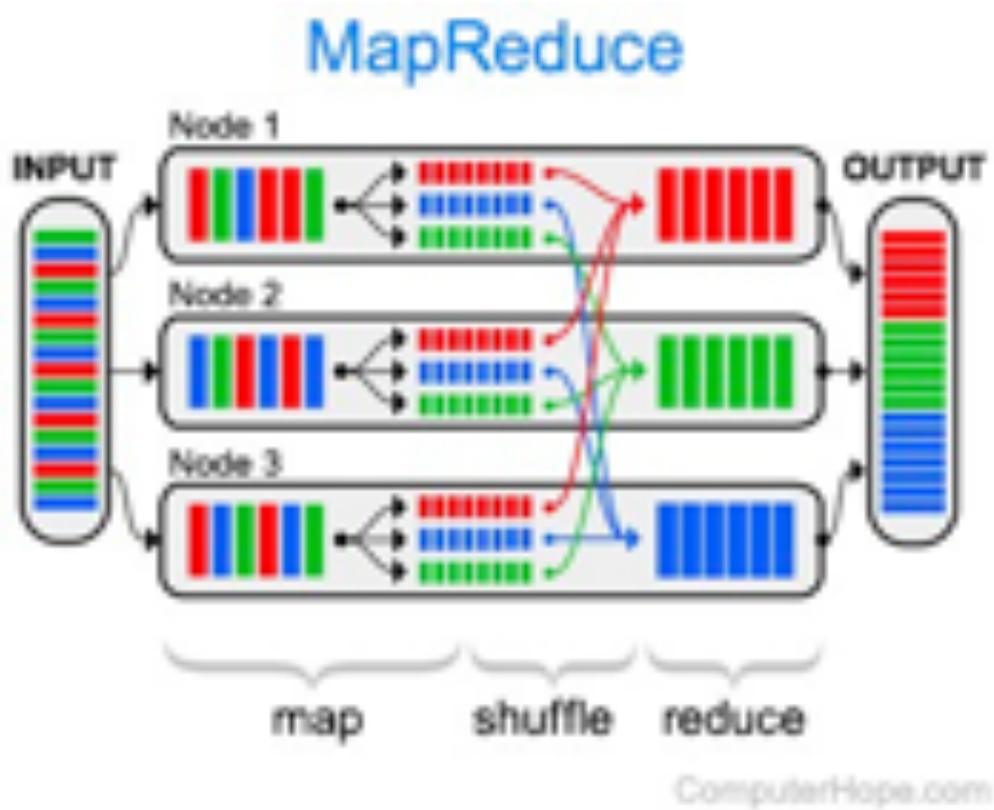
Implementation of Hogwild (asych SGD) in PyTorch

Multi-processing design in PyTorch

```
import torch.multiprocessing as mp
from model import MyModel

def train(model):
    # Construct data_loader, optimizer, etc.
    for data, labels in data_loader:
        optimizer.zero_grad()
        loss_fn(model(data), labels).backward()
        optimizer.step()  # This will update the shared parameters

if __name__ == '__main__':
    num_processes = 4
    model = MyModel()
    # NOTE: this is required for the ``fork`` method to work
    model.share_memory()
    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()
```

Case Study: MapReduce

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

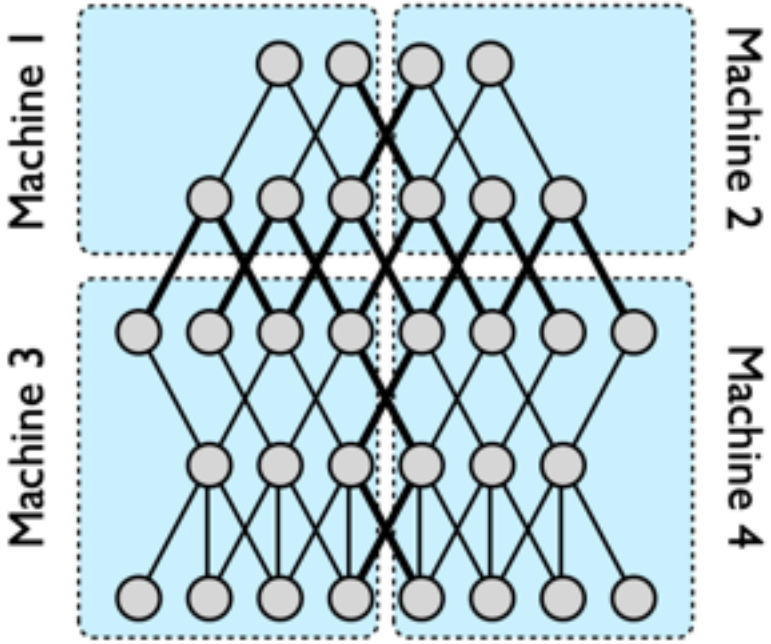


http://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf

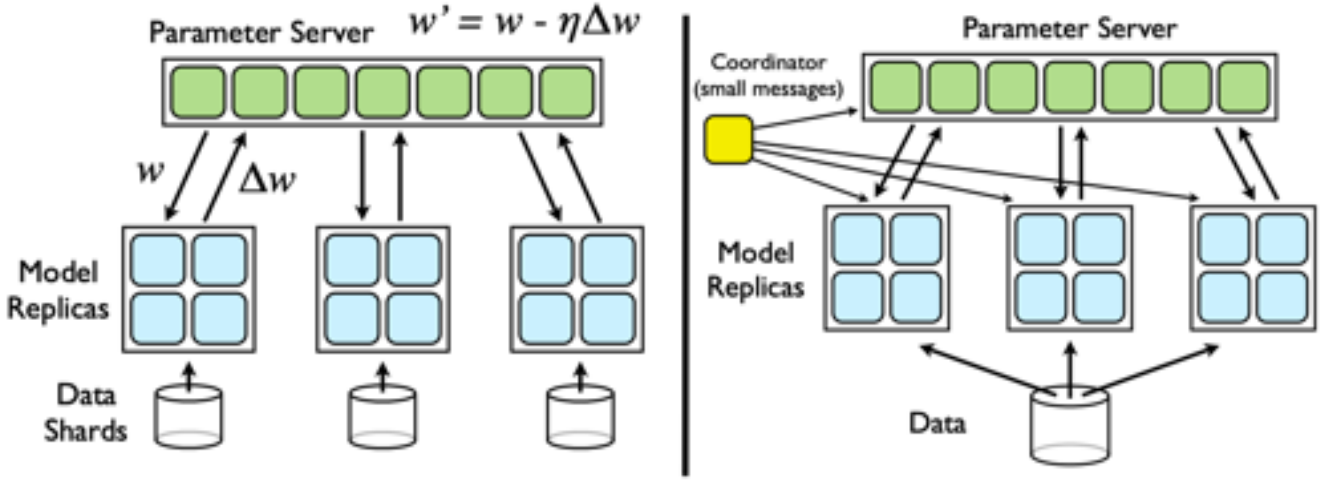
Case Study: DisBelief

- Distributed learning infrastructure used at Google (Jeff Dean et al. 2012)
- Hybrid of model and data parallelism across CPU cores

Model parallelism



Distributed optimization algorithms



Downpour SGD

Sandblaster L-BFGS

https://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

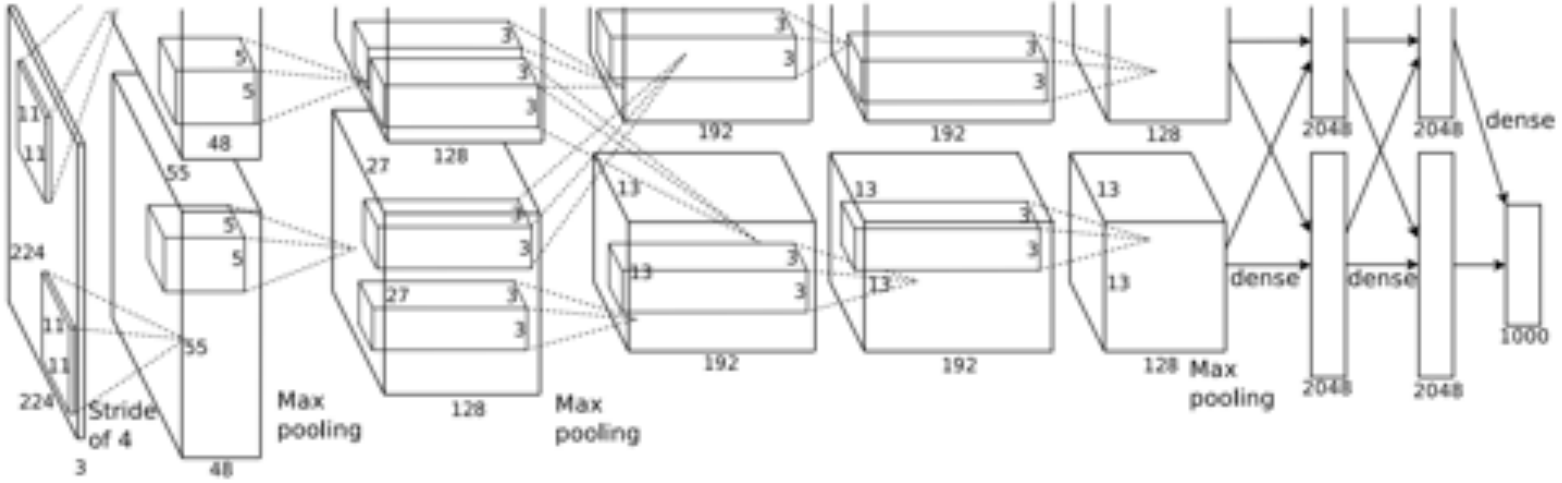
Fun facts about Jeff Dean

- Most badass engineer at Google
- Projects Jeff involved: MapReduce, DistBelief, LevelDB, BigTable, Tensorflow
- Fun gossips about Jeff:
  - Compilers don't warn Jeff Dean. Jeff Dean warns compilers.
  - Jeff Dean writes directly in binary. He then writes the source code as documentation for other developers.
  - Jeff Dean got promoted to level 11 in an employ system where max level is 10.

https://www.quora.com/What-are-all-the-Jeff-Dean-facts

Case Study: AlexNet

- Originally trained on 2 GTX 580 GPUs with 3 GB ram each only
- In 2013 I trained AlexNet on one Titan Black 6GB for two weeks



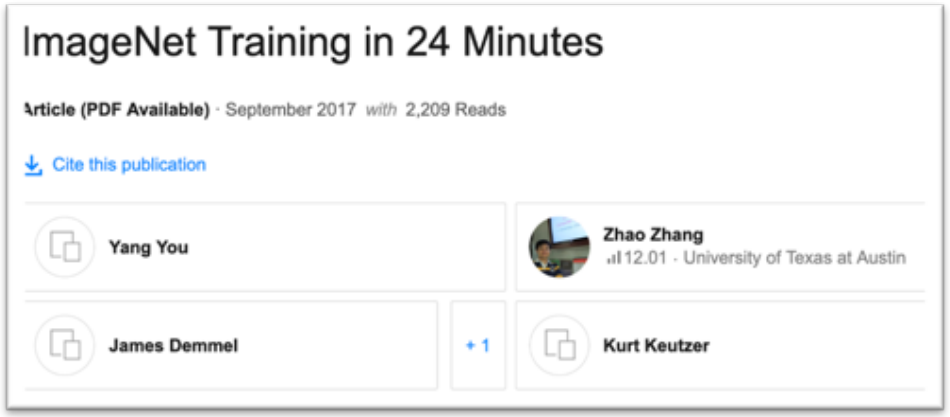
https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Case Study: AlexNet

Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Priya Goyal Piotr Dollár Ross Girshick Pieter Noordhuis
Lukasz Wesolowski Aapo Kyrola Andrew Tulloch Yangqing Jia Kaiming He

Facebook



ImageNet Training in 24 Minutes

Article (PDF Available) · September 2017 with 2,209 Reads

Cite this publication

Yang You

Zhao Zhang
12.01 · University of Texas at Austin

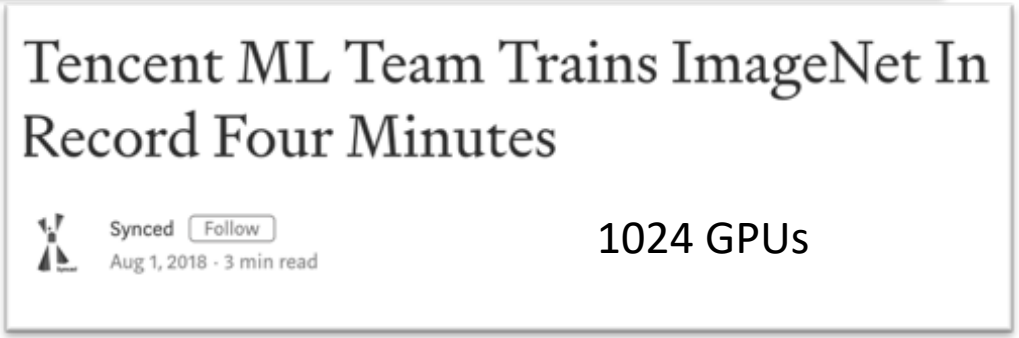
James Demmel

+ 1

Kurt Keutzer

Now anyone can train Imagenet in 18 minutes

Written: 10 Aug 2018 by *Jeremy Howard*



Tencent ML Team Trains ImageNet In Record Four Minutes

Synced Follow

Aug 1, 2018 · 3 min read

1024 GPUs

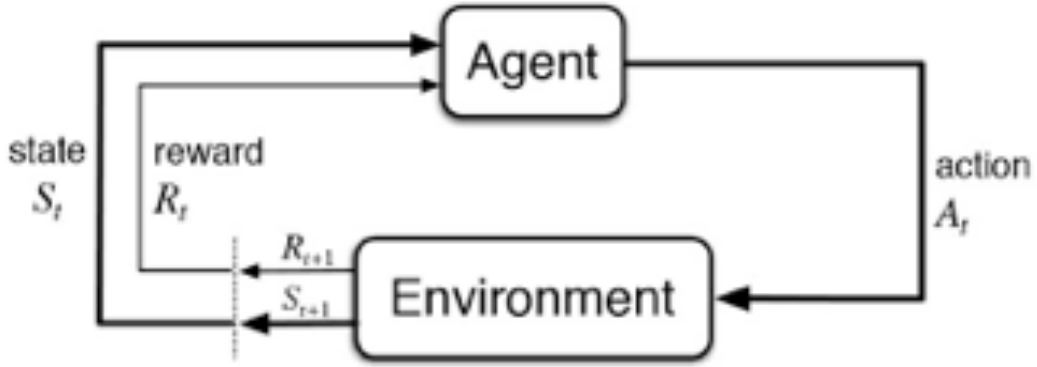
SenseTime Trains ImageNet/AlexNet In Record 1.5 minutes

Synced Follow

Feb 26 · 3 min read

512 Volta GPUs

Diagram of Reinforcement Learning



Single environment

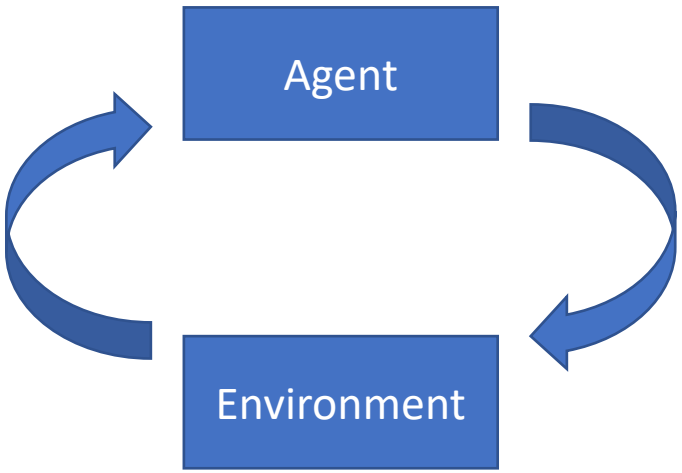
Single agent

```
import gym
env = gym.make("Breakout-v0")
observation = env.reset()
agent = load_agent()
for step in range(100):
    action = agent(observation)
    observation, reward, done, info = env.step(action)
```

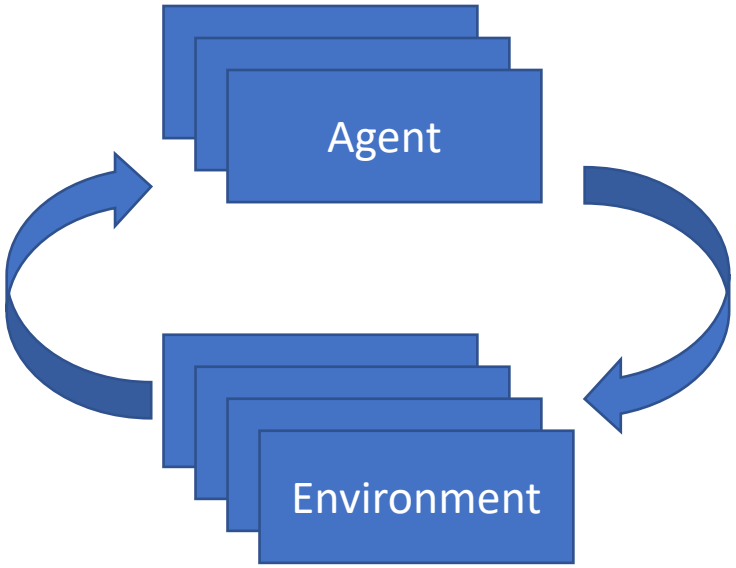
One episode at a time

Diagram of Reinforcement Learning

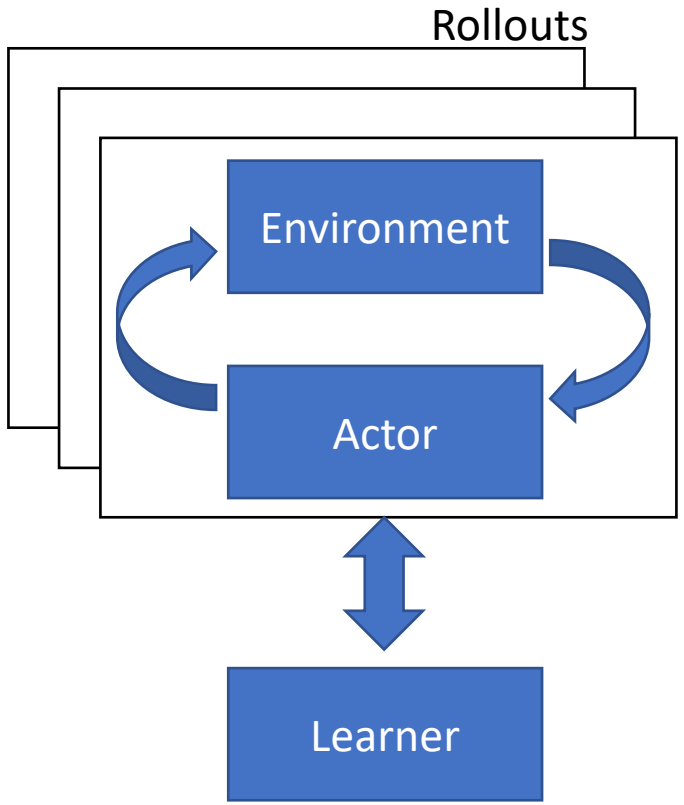
Standard RL



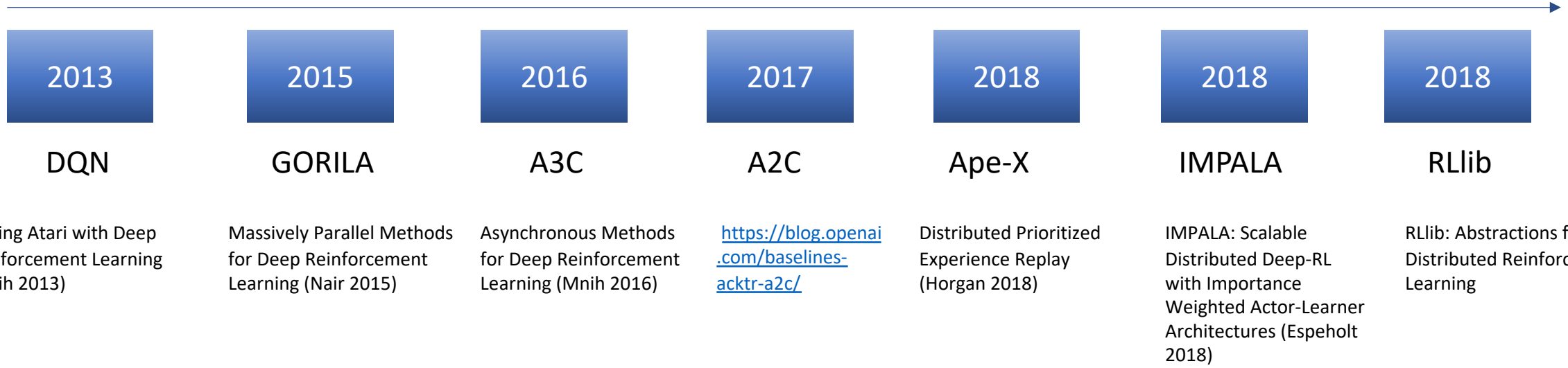
Distributed RL



Distributed RL implementation



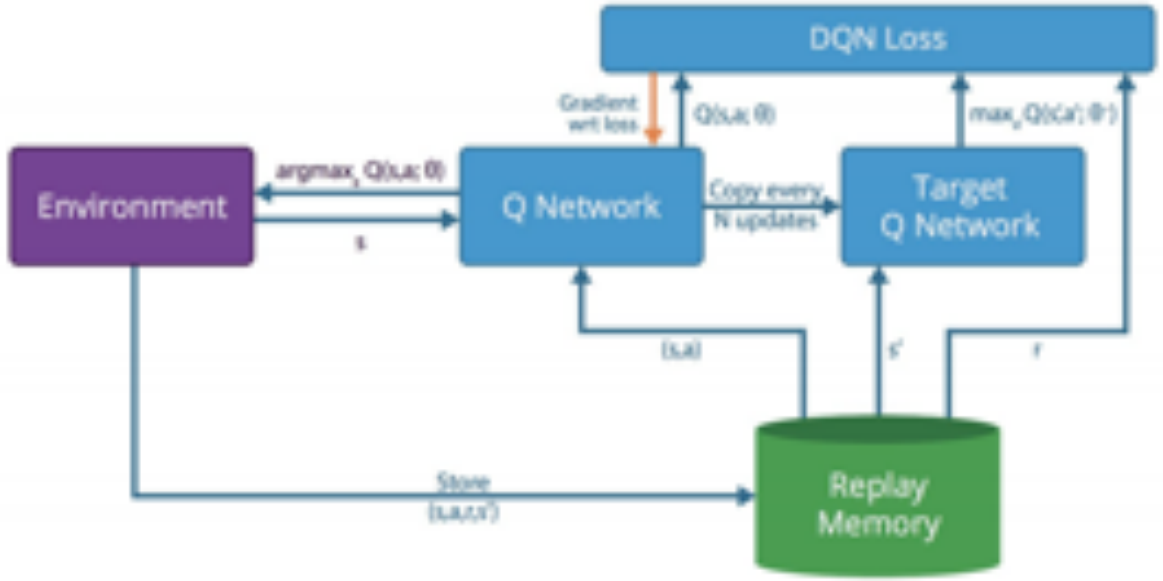
Development of Distributed RL Systems



2013: Deep Q Network

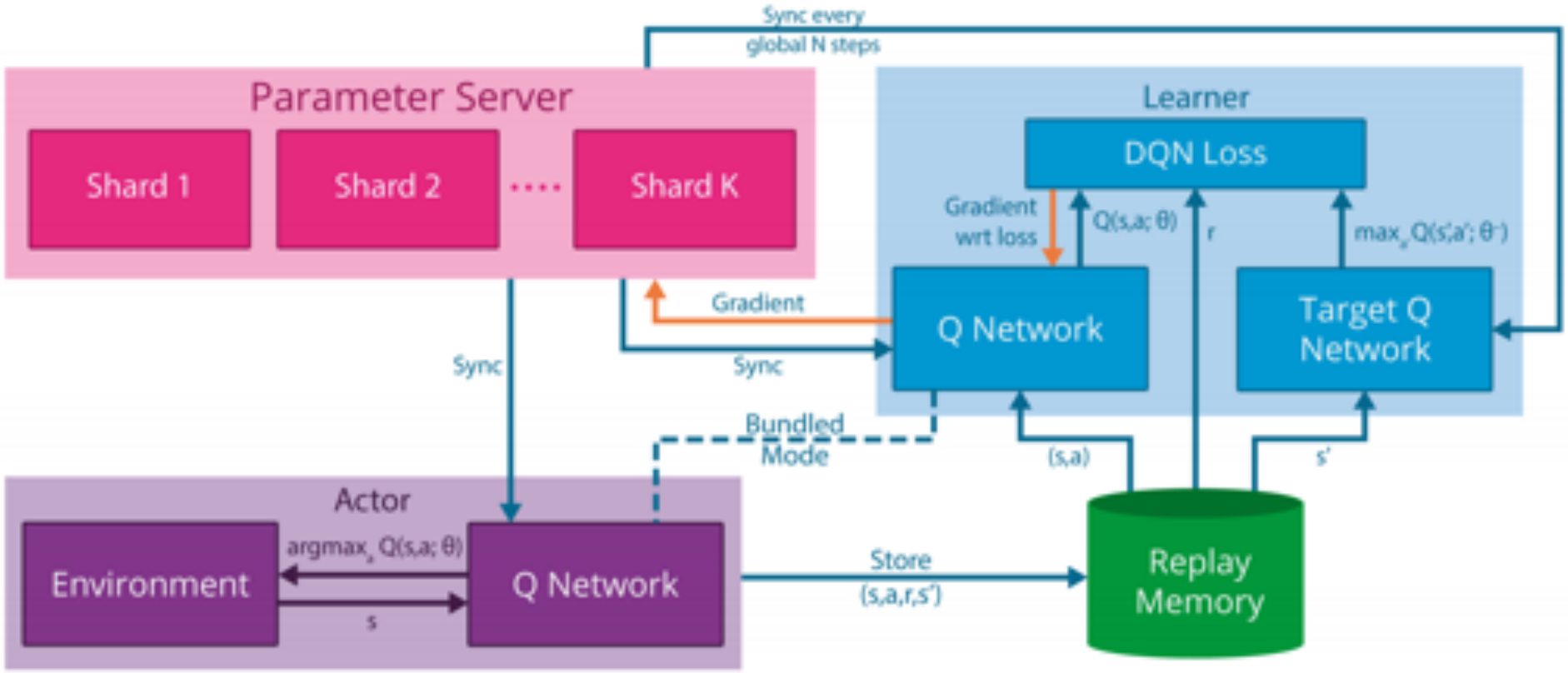
Drawback:

Very slow when single agent on a single machine interacts with a single environment



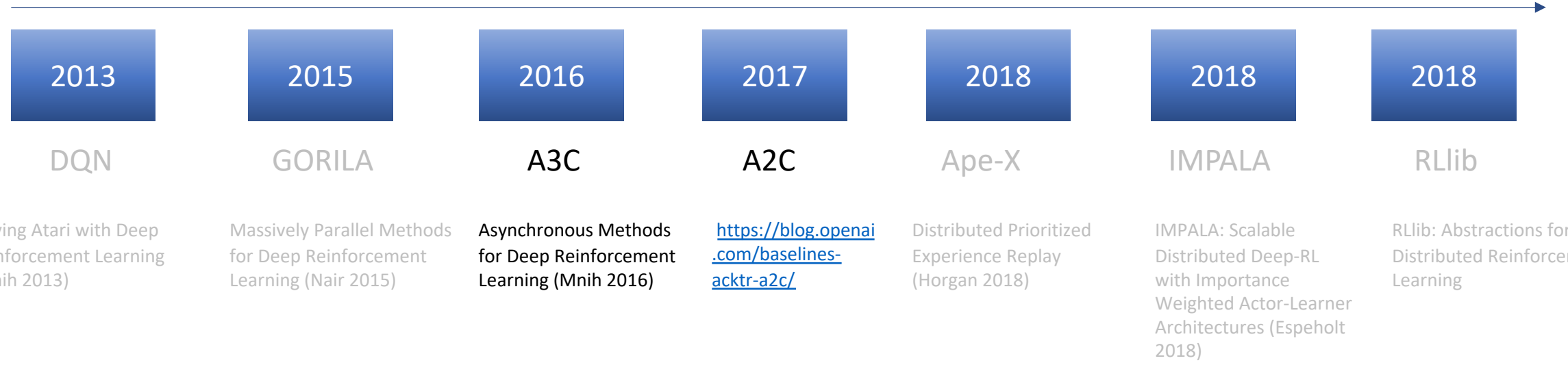

```
for i in range(T):
    s, a, s_1, r = evaluate()
    replay.store((s, a, s_1, r))

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

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

2015: General Reinforcement Learning Architecture (GORILA)



DeepMind. https://arxiv.org/pdf/1507.04296.pdf

Development of Distributed RL Systems



Review on Actor-Critic Methods

- Actor critic policy gradient
  - Actor: the policy function used to generate the action
  - Critic: the value function used to evaluate the reward of actions

Policy Update: $$\Delta\theta = \alpha * \nabla_\theta * (log\, \pi(S_t, A_t, \theta)) * R(t)$$

New update: $$\Delta\theta = \alpha * \nabla_\theta * (log\, \pi(S_t, A_t, \theta)) * Q(S_t, A_t)$$

Review on Actor-Critic Methods

- Basic Actor-Critic algorithm

1. Using a linear value function approximation: $Q_{\mathbf{w}}(s,a) = \psi(s,a)^T \mathbf{w}$
   1. Critic: update $\mathbf{w}$ by a linear TD(0)
   2. Actor: update θ by policy gradient

Algorithm 1 Simple QAC

1: **for** each step **do**
2: generate sample s, a, r, s', a' following π_θ
3: $\delta = r + \gamma Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s, a)$ #TD error
4: $\mathbf{w} \leftarrow \mathbf{w} + \beta \delta \psi(s, a)$
5: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_{\mathbf{w}}(s, a)$
6: **end for**

Review on Actor-Critic Methods

- Advantage function to stabilize learning
- Combing Q with baseline V: $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$
- Then the policy gradient becomes the form of **Advantage Actor Critic**:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}}[\nabla_{\theta} \log \pi_{\theta}(s,a) A^{\pi_{\theta}}(s,a)]$$

- Example code of AAC

```
for (log_prob, value), reward in zip(policy.saved_log_probs, rewards):
    advantage = reward - value
    policy_loss.append(- log_prob * advantage)          # policy gradient
    value_loss.append(F.smooth_l1_loss(value, reward)) # value function approximation
optimizer.zero_grad()
policy_loss = torch.stack(policy_loss).sum()
value_loss = torch.stack(value_loss).sum()
loss = policy_loss + value_loss
```

A3C: Asynchronous Advantage Actor Critic (A3C)

- From DeepMind researchers, ICML'16
- One of the widely used RL algorithms
- Multi-threaded **asynchronous** variants of one-step Sarsa, one-step Q-learning, n-step Q-learning, and advantage actor-critic.

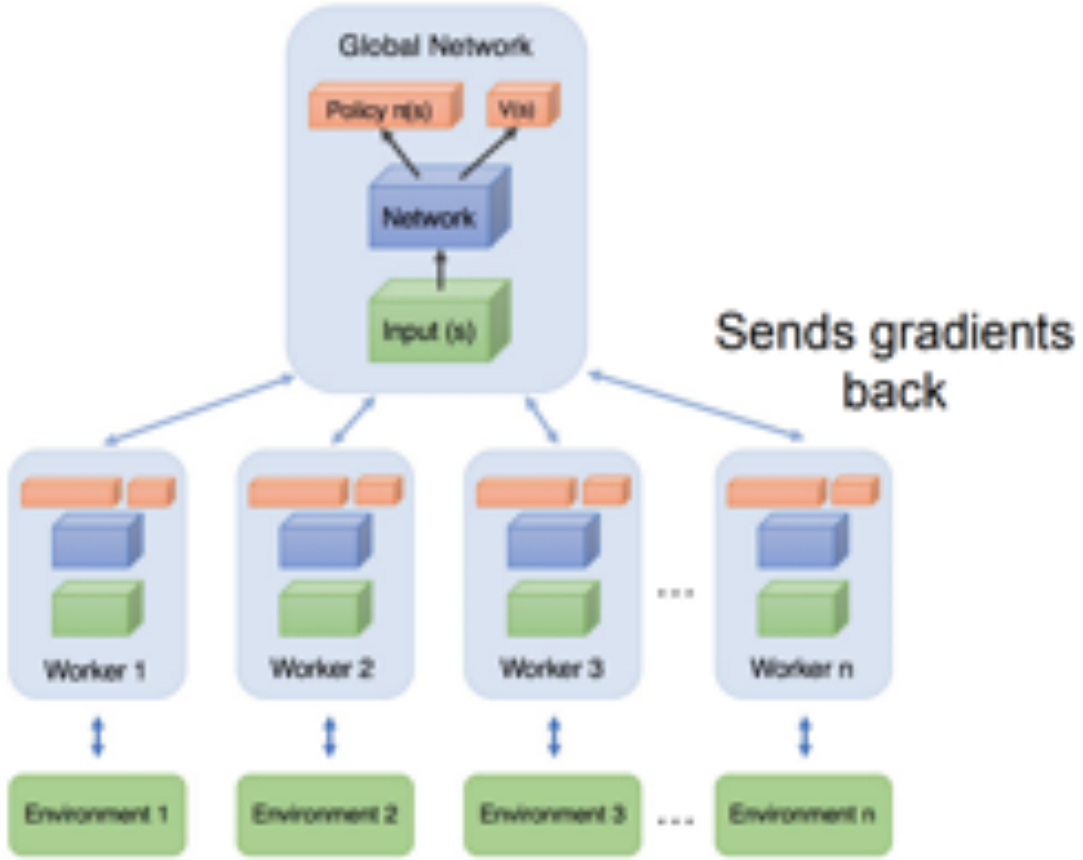
Mnih et al. Asynchronous Methods for Deep Reinforcement Learning. ICML 2016

A3C: Asynchronous Advantage Actor Critic (A3C)

- GORILA system design:
  - Actor-learners: separate machines
  - One parameter server
  - Replay memory
- A3C system design:
  - Actor learners: multiple CPU threads on a single machine
  - Rely on parallel actors to explore different parts of the environment
  - Diversity of the agents make it no need for reply memory to stabilize the training

A3C: 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()
```



Comparison to Variants of DQN and GORILA

- Faster updates
- Reply memory is removed
- Move 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%

Sample code for A3C

- https://github.com/ikostrikov/pytorch-a3c/blob/master/main.py
- https://github.com/greydanus/baby-a3c/blob/master/baby-a3c.py

```
torch.manual_seed(args.seed)
env = create_atari_env(args.env_name)
shared_model = ActorCritic(
    env.observation_space.shape[0], env.action_space)
shared_model.share_memory()

if args.no_shared:
    optimizer = None
else:
    optimizer = my_optim.SharedAdam(shared_model.parameters(), lr=args.lr)
    optimizer.share_memory()

processes = []

counter = mp.Value('i', 0)
lock = mp.Lock()

p = mp.Process(target=test, args=(args.num_processes, args, shared_model, counter))
p.start()
processes.append(p)

for rank in range(0, args.num_processes):
    p = mp.Process(target=train, args=(rank, args, shared_model, counter, lock, optimizer))
    p.start()
    processes.append(p)
for p in processes:
    p.join()
```

Why Asynchronism works in A3C?

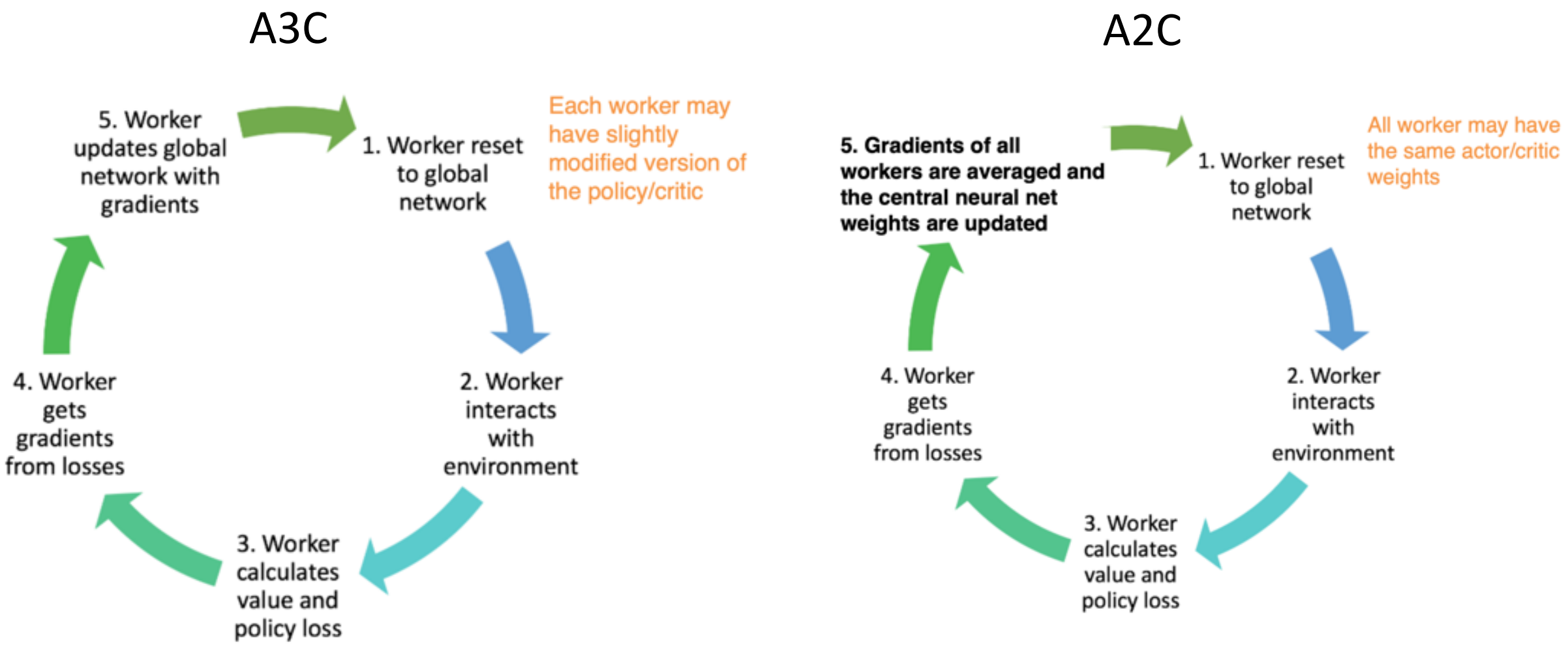
- Added noise would provide some regularization or exploration?
- Or if it was just an implementation detail that allowed for faster training with a CPU-based implementation?

2017: A2C

- Synchronous implementation: waits for each actor to finish its experience, then average the update over all the actors
- Advantage: more effectively use of GPUs (large batch sizes)
- A2C is more cost-effective than A3C when using a single GPU machine
  - A3C: Desktop computer with all CPU threads
  - A2C: Desktop computer with a GPU card and many CPU threads

https://openai.com/blog/baselines-acktr-a2c/

Comparison of A3C and A2C

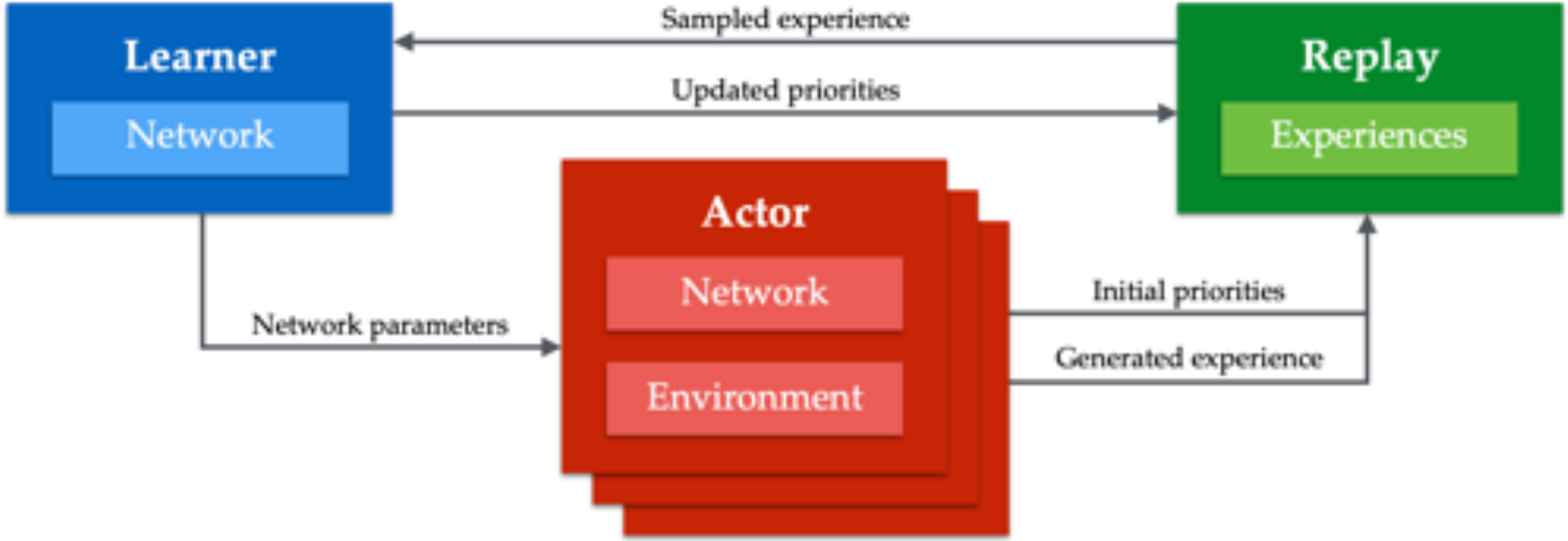


Sample code for A2C

- https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail

2018: Apex-X (Distributed Prioritized Experience Replay)

- A3C and A2C don’t scale very well across multiple machines
- Ape-X: Distributed DQN/DDPG



Horgan et al. Distributed Prioritized Experience Replay. ICLR 2018

2018: Apex-X (Distributed Prioritized Experience Replay)

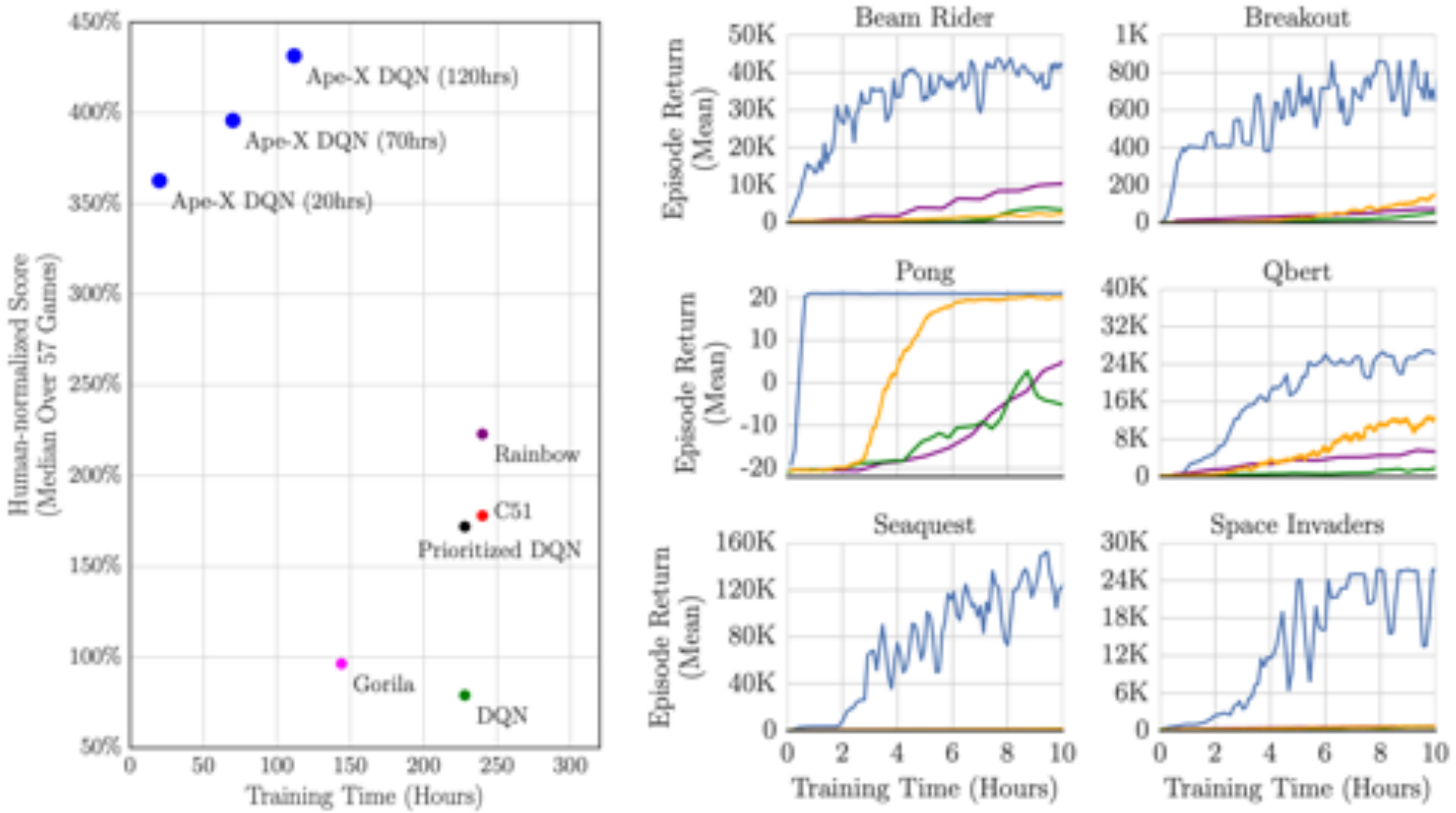
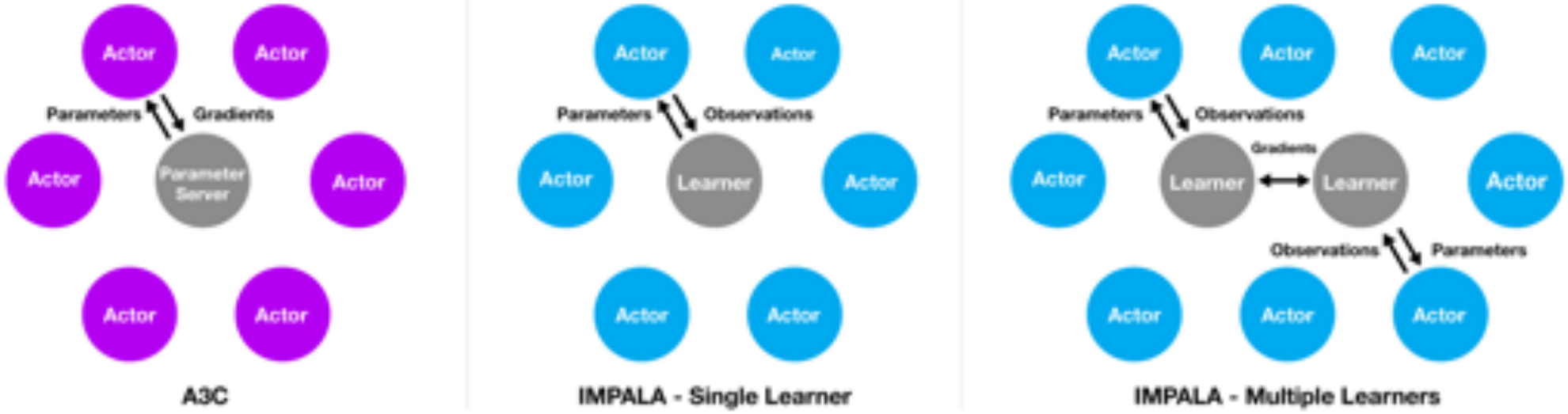


Figure 2: Left: Atari results aggregated across 57 games, evaluated from random no-op starts. Right: Atari training curves for selected games, against baselines. Blue: Ape-X DQN with 360 actors; Orange: A3C; Purple: Rainbow; Green: DQN. See appendix for longer runs over all games.

2018: IMPALA (Importance Weighted Actor-Learner Architecture)

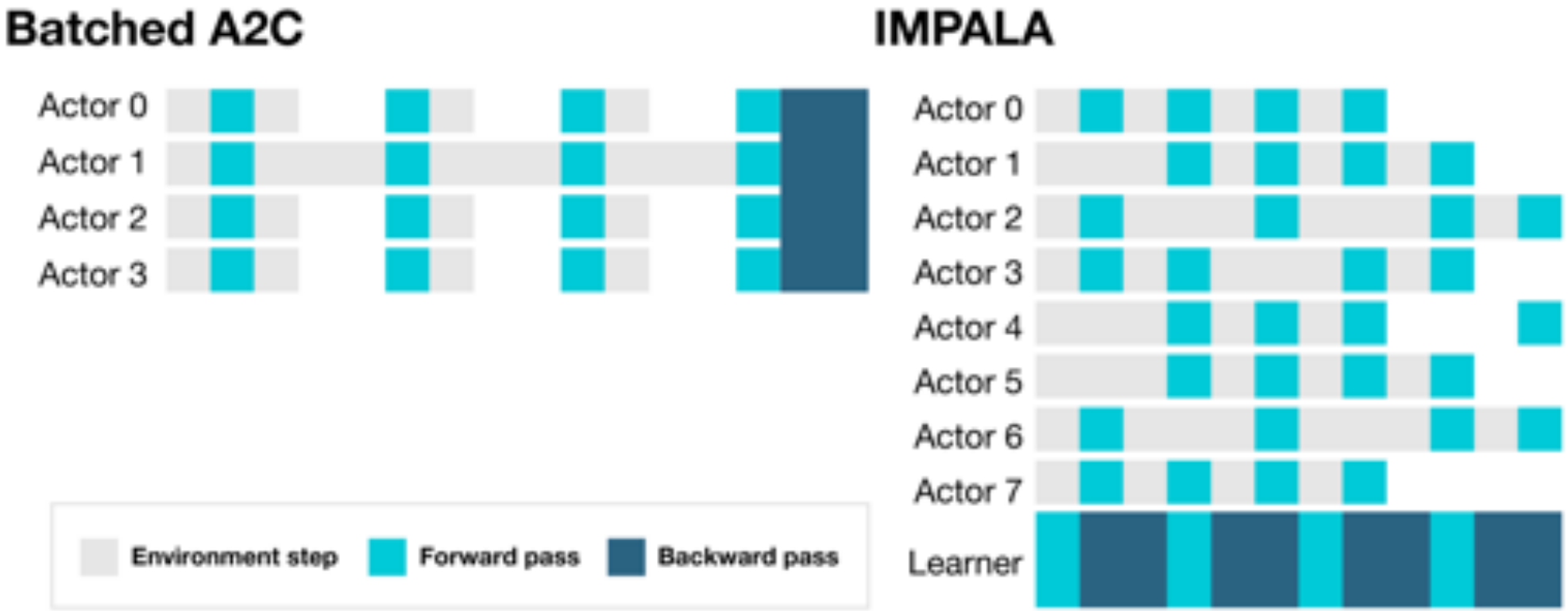
- Actors are used only to generate experience (no gradient)
- Completely independent actors and learners



https://arxiv.org/pdf/1802.01561.pdf
https://deepmind.com/blog/impala-scalable-distributed-deeprl-dmlab-30/

2018: IMPALA (Importance Weighted Actor-Learner Architecture)

- Learning is continuous with IMPALA, compared to A2C



2018: IMPALA (Importance Weighted Actor-Learner Architecture)

- Importance sampling to correct policy lag

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 value update

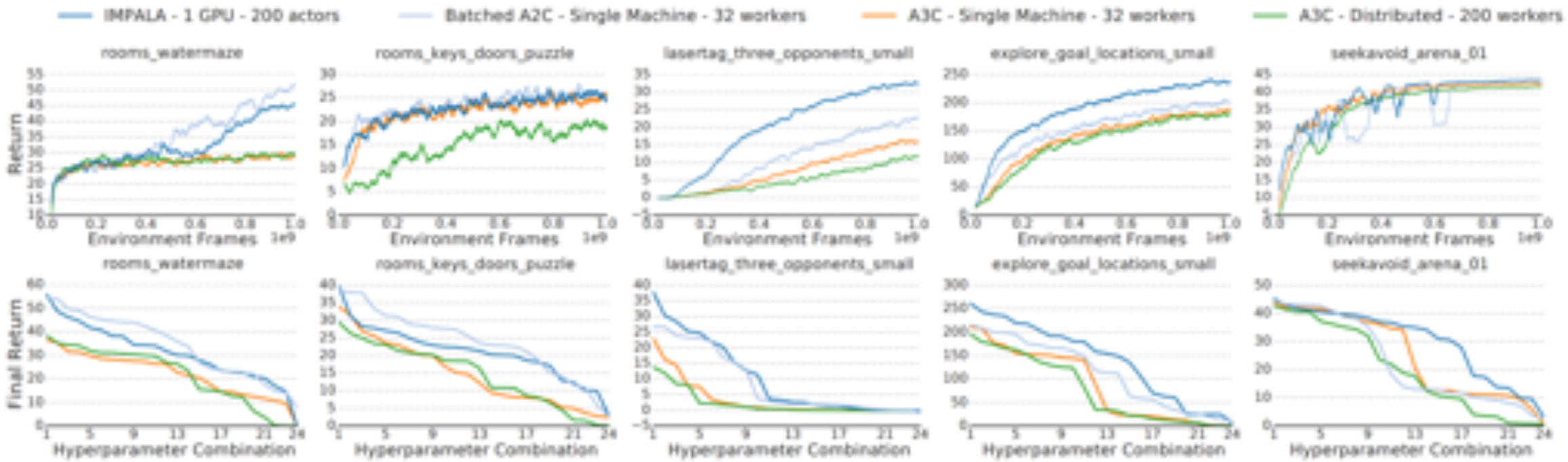
4.1. V-trace target

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:

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

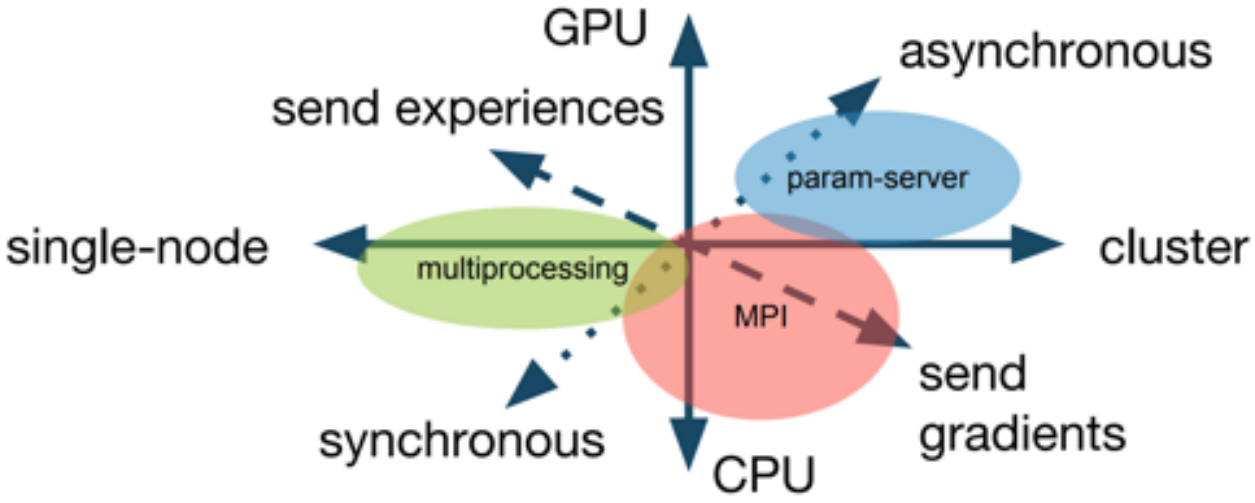
where $\delta_t V \stackrel{\text{def}}{=} \rho_t \big(r_t + \gamma V(x_{t+1}) - V(x_t) \big)$ is a temporal difference for V, and $\rho_t \stackrel{\text{def}}{=} \min \left(\bar{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)} \right)$ and $c_i \stackrel{\text{def}}{=} \min \left(\bar{c}, \frac{\pi(a_i|x_i)}{\mu(a_i|x_i)} \right)$ are truncated importance sampling (IS) weights (we make use of the notation $\prod_{i=s}^{t-1} c_i = 1$ for $s = t$). In addition we assume that the truncation levels are such that $\bar{\rho} \geq \bar{c}$.

2018: IMPALA (Importance Weighted Actor-Learner Architecture)



2018 RLlib: abstraction for distributed RL

- A huge variety of algorithms and distributed systems used to implement, but little reuse of components
- Abstractions for distributed reinforcement learning: Decompose RL algorithms into reusable components

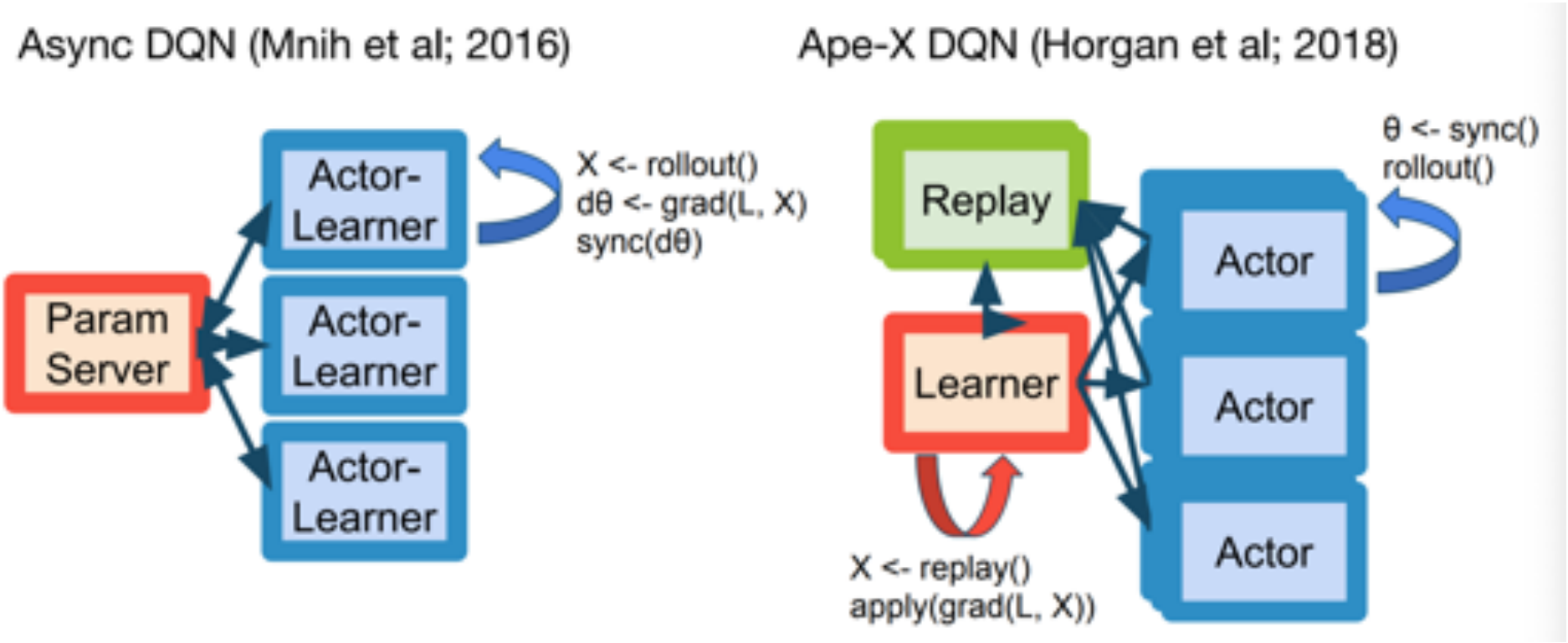


ICML'18. https://arxiv.org/pdf/1712.09381.pdf

https://ray.readthedocs.io/en/latest/rllib-algorithms.html

2018 RLlib: abstraction for distributed RL

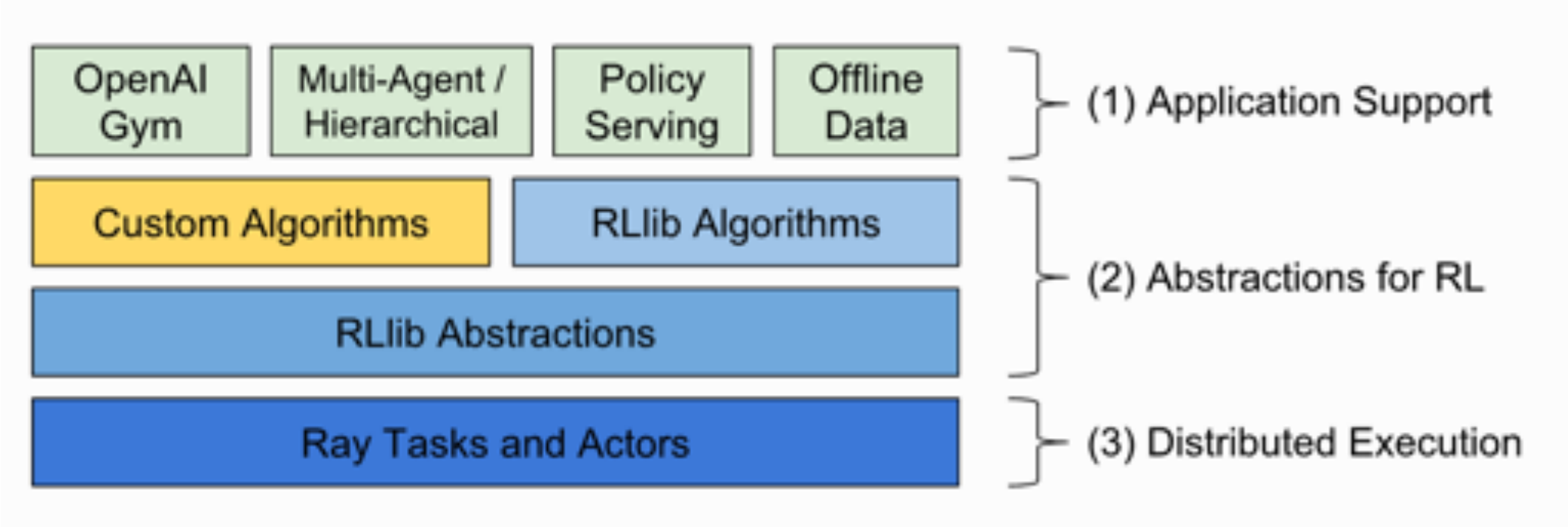
- Decompose the common components and reuse



ICML'18. https://arxiv.org/pdf/1712.09381.pdf https://ray.readthedocs.io/en/latest/rllib.html

2018 RLlib: abstraction for distributed RL

- Abstractions for distributed reinforcement learning in levels

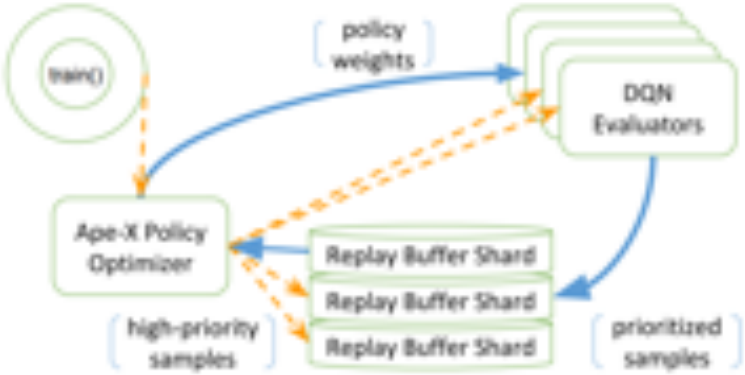


2018 RLlib: abstraction for distributed RL

- Complex RL architectures can be expressed in RLlib

Algorithms

- High-throughput architectures
  - Distributed Prioritized Experience Replay (Ape-X)
  - Importance Weighted Actor-Learner Architecture (IMPALA)
  - Asynchronous Proximal Policy Optimization (APPO)
- Gradient-based
  - Advantage Actor-Critic (A2C, A3C)
  - Deep Deterministic Policy Gradients (DDPG, TD3)
  - Deep Q Networks (DQN, Rainbow, Parametric DQN)
  - Policy Gradients
  - Proximal Policy Optimization (PPO)
- Derivative-free
  - Augmented Random Search (ARS)
  - Evolution Strategies
- Multi-agent specific
  - QMIX Monotonic Value Factorisation (QMIX, VDN, IQN)
- Offline



(a) Ape-X

(b) PPO-ES

(c) AlphaGo Zero

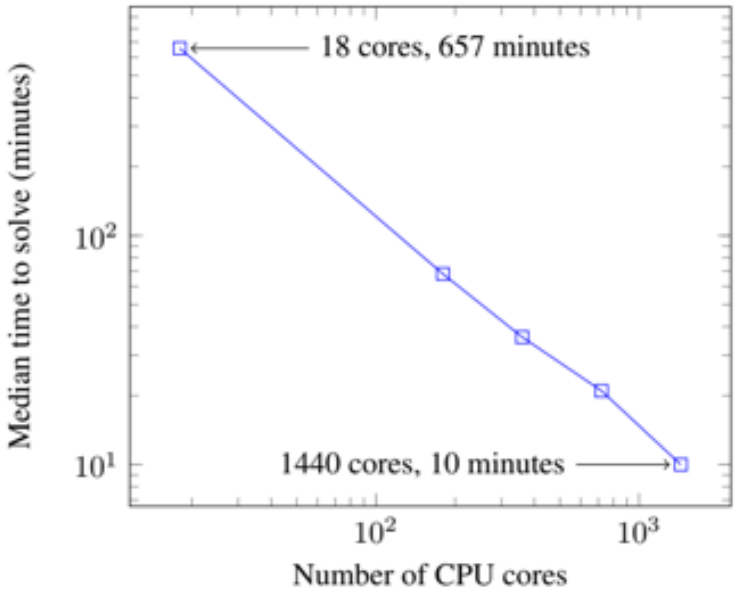
Some Other Parallelizable Algorithms: (Revisited) Evolution Strategies

- Highly parallelizable black-box optimization

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

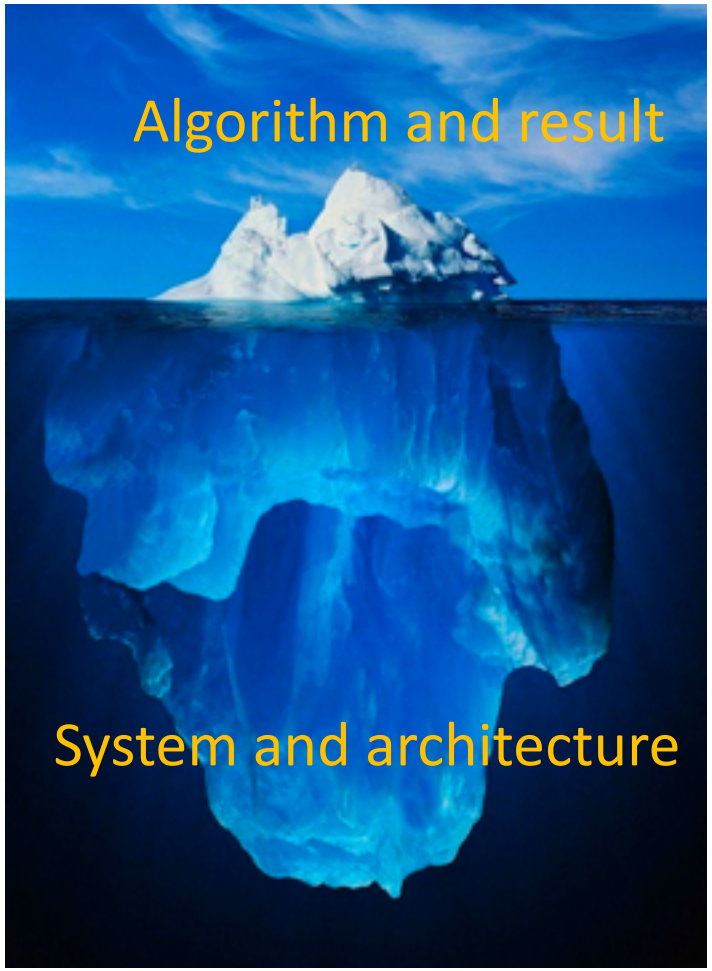
Linear speedup
1,440 CPU workers solve MuJoCo 3D humanoid in 10 min



https://openai.com/blog/evolution-strategies/ https://arxiv.org/pdf/1703.03864.pdf

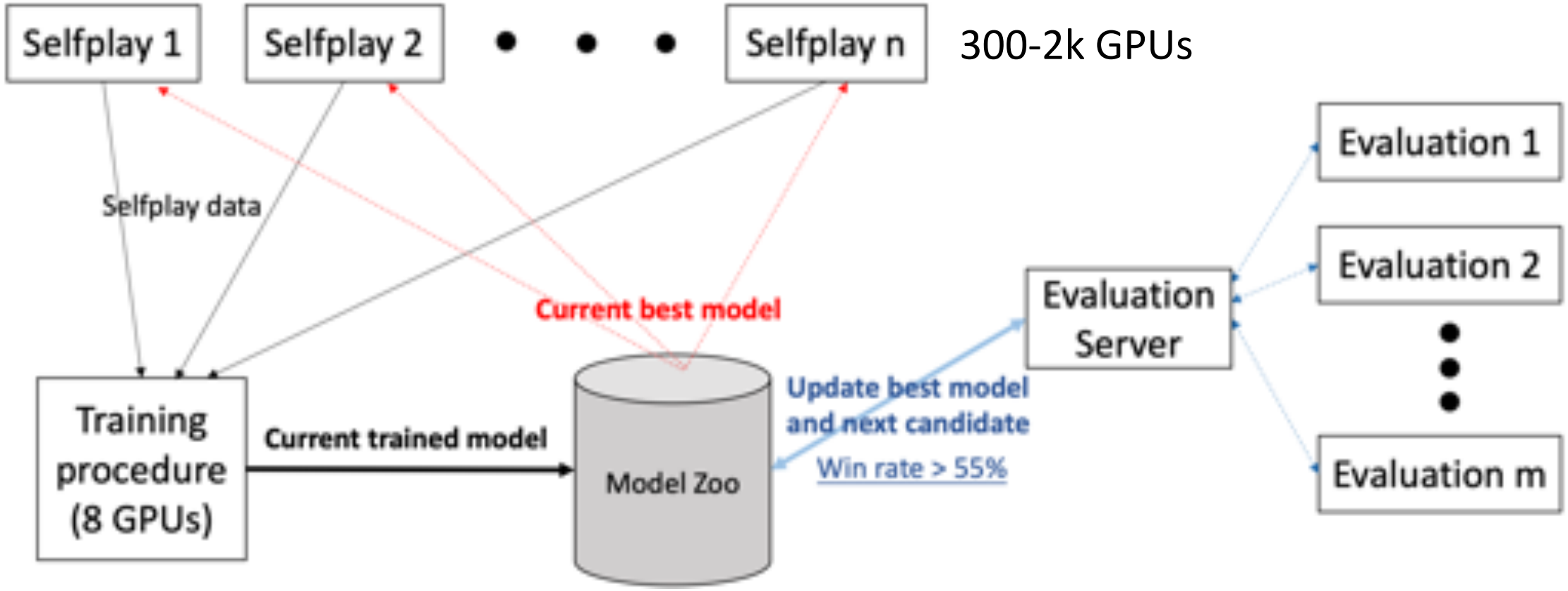
Case Study: AI for Modern Games

- AlphaGo for playing the Game of Go
- OpenAI Five for the Dota 2
- AlphaStar for playing the Starcraft2



System Design for AlphaGo Zero

- Reproducing AlphaGo Zero by Yuandong Tian at Facebook AI Research

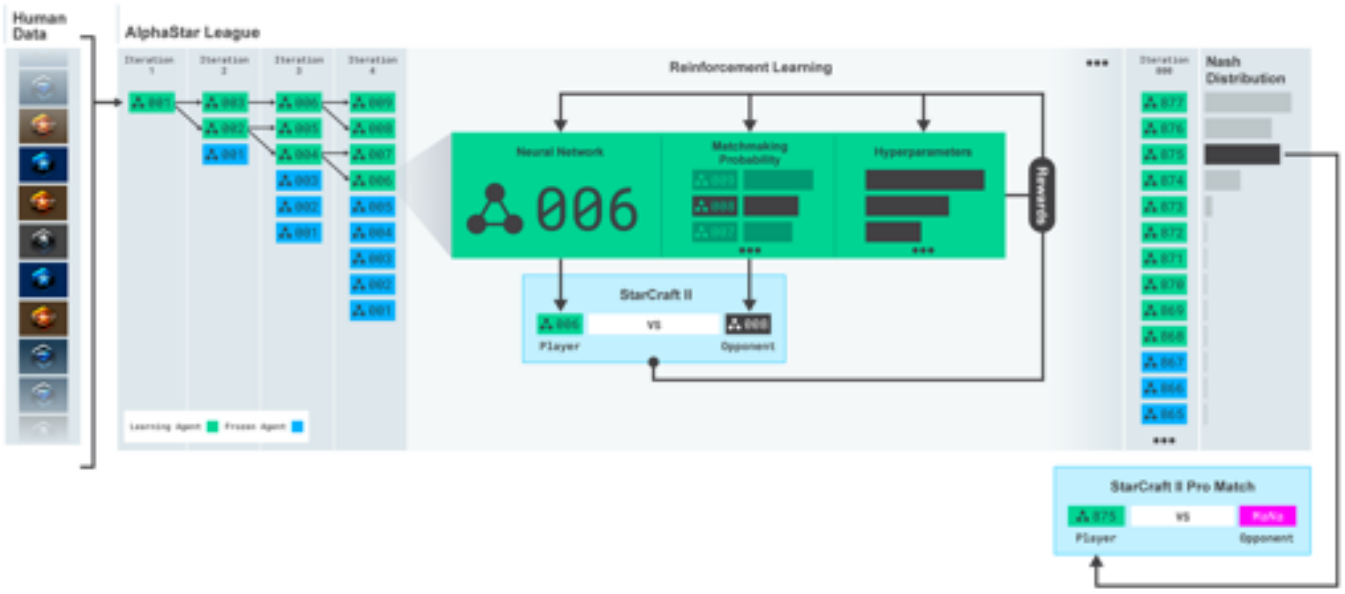


https://yuandong-tian.com/reproducibility.pdf

System Design for AlphaStar

- Population-based and multi-agent RL

AlphaStar training league



How AlphaStar is trained

AlphaStar's behaviour is generated by a deep neural network that receives input data from the raw game interface (a list of units and their properties), and outputs a sequence of instructions that constitute an action within the game. More specifically, the neural network architecture applies a transformer torso to the units (similar to relational deep reinforcement learning), combined with a deep LSTM core, an auto-regressive policy head with a pointer network, and a centralised value baseline. We believe that this advanced model will help with many other challenges in machine learning research that involve long-term sequence modelling and large output spaces such as translation, language modelling and visual representations.

https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/

System Design for OpenAI Five

- Dota2: a real-time strategy game played between two teams of five players with each player controlling a hero
- One of the most popular and complex games in the world, with annual $40M prize pool

System Design for OpenAI Five

- April 13, 2019: Live of OpenAI Five vs. Dota 2 world champions

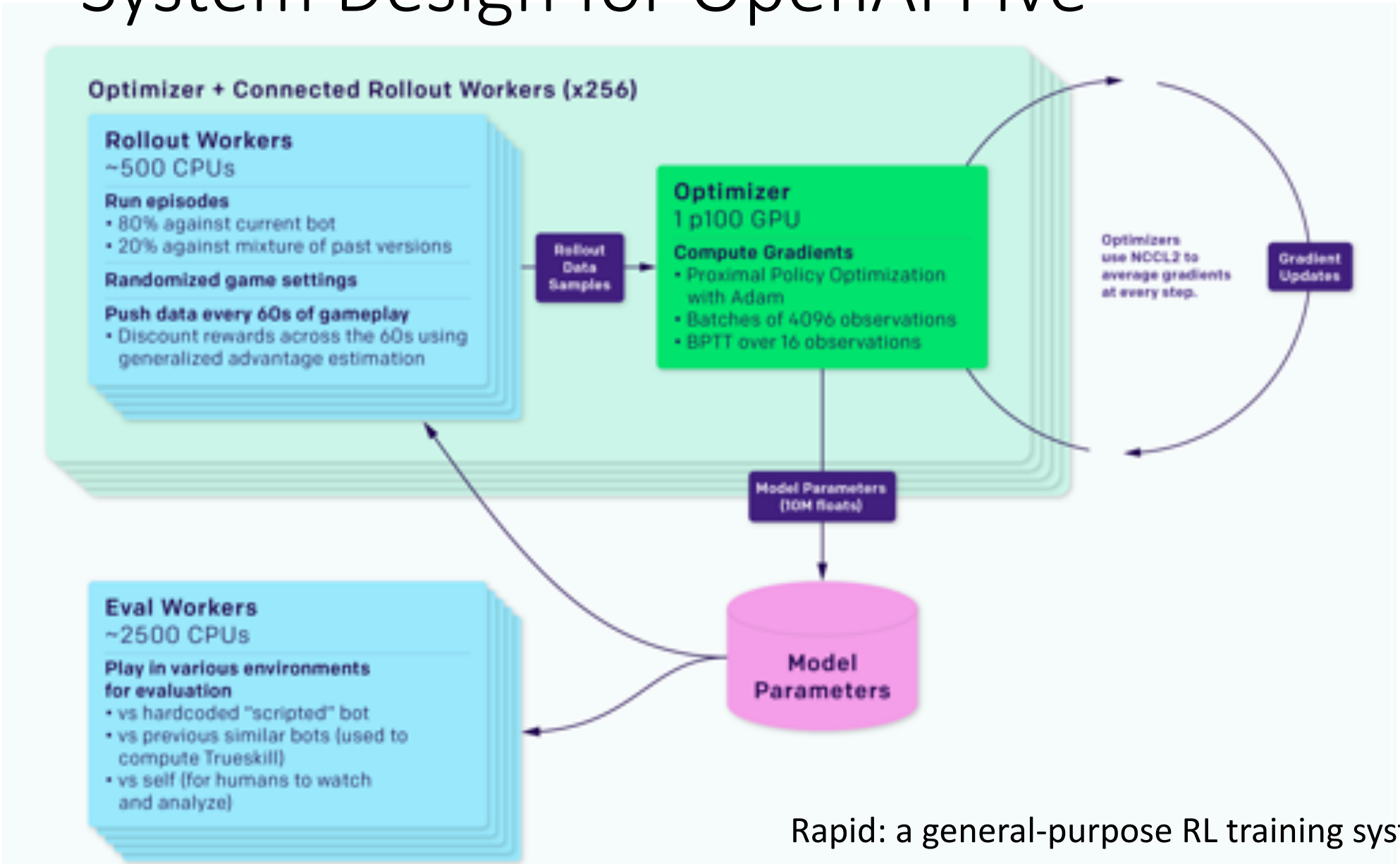
https://openai.com/blog/openai-five-finals/

System Design for OpenAI Five

- Massively-scaled version of Proximal Policy Optimization (PPO) with 256 GPUs and 128,000 CPU cores
- Model: each hero is a single-layer, 1024-unit LSTM

	OPENAI 1V1 BOT	OPENAI FIVE
CPUs	60,000 CPU cores on Azure	128,000 preemptible CPU cores on GCP
GPUs	256 K80 GPUs on Azure	256 P100 GPUs on GCP
Experience collected	~300 years per day	~180 years per day (~900 years per day counting each hero separately)
Size of observation	~3.3 kB	~36.8 kB
Observations per second of gameplay	10	7.5
Batch size	8,388,608 observations	1,048,576 observations
Batches per minute	~20	~60

System Design for OpenAI Five



Rapid: a general-purpose RL training system

Conclusion

- Computer system and architecture design becomes more and more important to machine learning
- Full-stack software-hardware systems + ML will drive future intelligent systems