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

Agent

Environment

Action

Observation, Reward

Batch Optimization

Optimization

Simulation

Simulation

Simulation

Optimization

How can we better utilize our computational resources to accelerate RL progress?
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
DQN
Playing Atari with Deep Reinforcement Learning (Mnih 2013)

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

2016
Policy Gradient
A3C
Asynchronous Methods for Deep Reinforcement Learning (Mnih 2016)

2018
Q-Learning
Ape-X
Distributed Prioritized Experience Replay (Horgan 2018)

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

?
History of large scale distributed RL

- **2013**: DQN
  Playing Atari with Deep Reinforcement Learning (Mnih 2013)

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

- **2016**: A3C
  Asynchronous Methods for Deep Reinforcement Learning (Mnih 2016)

- **2018**: Ape-X
  Distributed Prioritized Experience Replay (Horgan 2018)

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

- ?
2013/2015: DQN

1. \((s_i, a_i, s'_i, r_i) = \text{env.step}(a_i)\)
   Store \((s_i, a_i, s'_i, r_i)\) in \(B\)
2. Sample batch \((s_j, a_j, s'_j, r_j)\) from \(B\)
   Update Q network
3. Update target network parameters: \(\phi' \leftarrow \phi\)

History of large scale distributed RL

- 2013: DQN
  Playing Atari with Deep Reinforcement Learning (Mnih 2013)

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

- 2016: A3C
  Asynchronous Methods for Deep Reinforcement Learning (Mnih 2016)

- 2018: Ape-X
  Distributed Prioritized Experience Replay (Horgan 2018)

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

- ?
2015: General Reinforcement Learning Architecture (GORILA)

2015: General Reinforcement Learning Architecture (GORILA)

- **Standard DQN**
  1. \((s_i, a_i, s'_i, r_i) = \text{env.step}(a_i)\) 
     Store \((s_i, a_i, s'_i, r_i)\) in \(B\)
  2. Sample batch \((s_j, a_j, s'_j, r_j)\) from \(B\) 
     Update \(Q\) network
  3. Update target network parameters: \(\phi' \leftarrow \phi\)

- **Distributed DQN**
  1. \((s_i, a_i, s'_i, r_i) = \text{env.step}(a_i)\) 
     Store \((s_i, a_i, s'_i, r_i)\) in \(B\)
  2. Sample batch \((s_j, a_j, s'_j, r_j)\) from \(B\) 
     Update \(\theta\) with \(\theta^+\) from parameter server 
     Calculate gradients w.r.t. \(\theta\) 
     Send gradients to parameter server 
  3. Update \(Q\) network 
  4. Update target network parameters \(\theta^-\) 
     with \(\theta^+\) from the parameter server every \(N\) steps
GORILA Performance
History of large scale distributed RL

- **DQN** (2013)
  Playing Atari with Deep Reinforcement Learning (Mnih 2013)

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

- **A3C** (2016)
  Asynchronous Methods for Deep Reinforcement Learning (Mnih 2016)

- **Ape-X** (2018)
  Distributed Prioritized Experience Replay (Horgan 2018)

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

- **?** (2018?)
  (Note: The year is missing for this entry.)
Prioritised Experience Replay

**Standard DQN**

1. \((s_i, a_i, s'_i, r_i) = \text{env.step}(a_i)\)
   Store \((s_i, a_i, s'_i, r_i)\) in \(B\)

2. Sample \(K\) transitions \((s_j, a_j, s'_j, r_j)\) **uniformly** from \(B\)
   Update \(Q\) network

3. Update target network parameters: \(\phi' \leftarrow \phi\)

**Prioritised Experience Replay**

1. \((s_t, a_t, s'_t, r_t) = \text{env.step}(a_t)\)
   Store \((s_t, a_t, s'_t, r_t)\) in \(B\) 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 \(B\)
   Compute TD error \(\delta_j\)
   Update transition probability \(p_j \leftarrow |\delta_j|\)

3. Update \(Q\) network

4. Update target network parameters: \(\phi' \leftarrow \phi\)

Distributed Prioritized Experience Replay (Ape-X)

Prioritised Experience Replay

- Sampled experience
- Updated priorities
- Generated experience

Ape-X

**Actors**
- \((s_i, a_i, s'_i, r_i) = \text{env.step}(a_i)\)
- Compute TD error \(\delta_i\)
  - Update transition probability \(p_j \leftarrow |\delta_j|\)
- Store \((s_i, a_i, s'_i, r_i)\) in \(B\)

**Learners**
- Update \(\theta\) with \(\theta^+\) from parameter server
- Sample transition \((s_j, a_j, s'_j, r_j)\) from \(B\)
- Calculate gradients w.r.t. \(\theta\)
- Update parameters \(\theta\) of \(Q\)-network
- Compute TD error \(\delta_j\)
  - Update transition probability \(p_j \leftarrow |\delta_j|\)
- Update target network parameters \(\theta^-\) with \(\theta^+\) from the parameter server every \(N\) steps

Distributed Prioritized Experience Replay (Ape-X)

**Gorila**

**Actors**
- \((s_i, a_i, s'_i, r_i) = \text{env.step}(a_i)\)
  - Store \((s_i, a_i, s'_i, r_i)\) in \(B\)

**Learners**
- Update \(\theta\) with \(\theta^+\) from parameter server
- Sample batch \((s_j, a_j, s'_j, r_j)\) from \(B\)
- Calculate gradients w.r.t. \(\theta\)
- Send gradients to parameter server
- Update target network parameters \(\theta^-\) with \(\theta^+\) from the parameter server every \(N\) steps

**Parameter Server**
- Update parameters \(\theta\) of Q network

**Ape-X**

**Actors**
- \((s_i, a_i, s'_i, r_i) = \text{env.step}(a_i)\)
- Compute TD error \(\delta_j\)
  - Update transition probability \(p_j \leftarrow |\delta_j|\)
- Store \((s_i, a_i, s'_i, r_i)\) in \(B\)

**Learners**
- Update \(\theta\) with \(\theta^+\) from parameter server
- Sample transition \((s_j, a_j, s'_j, r_j)\) from \(B\)
- Calculate gradients w.r.t. \(\theta\)
- Update parameters \(\theta\) of Q-network
- Compute TD error \(\delta_j\)
  - Update transition probability \(p_j \leftarrow |\delta_j|\)
- Update target network parameters \(\theta^-\)
  - with \(\theta^+\) from the parameter server every \(N\) steps

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**: DQN
  - Playing Atari with Deep Reinforcement Learning
  - (Mnih 2013)

- **2015**: GORIL
  - 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)

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

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

Each has different exploration -> more diverse samples!

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

Asynchronous advantage actor-critic (A3C)

Some extra features:

• n-step estimation: \( \hat{A}^\pi(s, a) = \sum_{i=0}^{k-1} \gamma^i r(s_t, a_t) + \gamma^k \hat{V}_\phi(s_{t+k}) - \hat{V}_\phi(s_t) \)

• Entropy of the policy \( \pi_\theta \) was added to the objective function to improve exploration:

\[
\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) \hat{A}^\pi(s, a) + \beta \nabla_\theta H(\pi_\theta(s))
\]
A3C Performance

Changes to GORILA:

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

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

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** DQN
  Playing Atari with Deep Reinforcement Learning (Mnih 2013)

- **2015** GORIL
  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)

- **2018** IMPALA
  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.
How to correct for Policy Lag? Importance Sampling!

Shortcoming of A3C:
• Policy-lag

Apply importance sampling:

1. To policy gradient

\[
\mathbb{E}_{a_s \sim \mu(\cdot | x_s)} \left[ \pi \hat{\rho}(a_s | x_s) \nabla \log \pi \hat{\rho}(a_s | x_s) q_s | x_s \right]
\]

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 \(\mu\). We define the \(n\)-steps V-trace target for \(V(x_s)\), our value approximation at state \(x_s\), as:

\[
v_s \overset{\text{def}}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \left( \prod_{i=s}^{t-1} c_i \right) \delta_t V, \quad (1)
\]
IMPALA - Performance

A comparison between IMPALA, A3C and batched A2C
IMPALA - Performance
IMPALA Performance
Evolution Strategies

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 $t = 0, 1, 2, \ldots$ do
4:   for each worker $i = 1, \ldots, n$ do
5:     Sample $\epsilon_i \sim N(0, I)$
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 $i = 1, \ldots, n$ do
10:  Reconstruct all perturbations $\epsilon_j$ for $j = 1, \ldots, n$ 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
## Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Policy Evaluation</th>
<th>Gradient-based optimizer</th>
<th>CPU</th>
<th>GPU</th>
<th>Replay Buffer</th>
<th>Prioritised Replay</th>
<th>Parameter Server</th>
<th>Importance Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gorila</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Ape-X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3C</td>
<td>X</td>
<td>X</td>
<td>many</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impala</td>
<td>X</td>
<td>X</td>
<td>many</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
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
RL research scales with compute

http://rllib.io
How do we leverage this hardware?

(a) Supervised Learning
(b) Reinforcement Learning

scalable abstractions for RL?

http://rllib.io
Systems for RL today

• Many implementations (7000+ repos on GitHub!)
  – how general are they (and do they scale)?
    PPO: multiprocessing, MPI
    Evolution Strategies: Redis
    AlphaZero: custom systems
    IMPALA: Distributed TensorFlow
    A3C: shared memory, multiprocessing, TF

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

http://rllib.io
Challenges to reuse

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

http://rllib.io
Challenges to reuse

2. Tight coupling with deep learning frameworks

Different parallelism paradigms:
  – Distributed TensorFlow vs TensorFlow + MPI?

http://rllib.io
Challenges to reuse

3. Large variety of algorithms with different structures

<table>
<thead>
<tr>
<th>Algorithm Family</th>
<th>Policy Evaluation</th>
<th>Replay Buffer</th>
<th>Gradient-Based Optimizer</th>
<th>Other Distributed Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQNs</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Policy Gradient</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Off-policy PG</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Model-Based/Hybrid</td>
<td>X</td>
<td></td>
<td>X</td>
<td>Model-Based Planning</td>
</tr>
<tr>
<td>Multi-Agent</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Derivative-Free Optimization</td>
</tr>
<tr>
<td>Evolutionary Methods</td>
<td>X</td>
<td></td>
<td></td>
<td>MCTS, Derivative-Free Optimization</td>
</tr>
<tr>
<td>AlphaGo</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
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

Agnent

Policy:
state →
action

Environment

state (s)
(observation)

reward (r_i)

action (a_{i+1})

http://rllib.io
Structure of RL computations

Agen

Policy improvement (e.g., SGD)

Policy evaluation (state $\rightarrow$ action)

trajectory $X$: $s_0, (s_1, r_1), \ldots, (s_n, r_n)$

Environmen

action ($a_{i+1}$)

state (observation $s_i$)

reward ($r_i$)

http://rllib.io
Many RL loop decompositions

Async DQN (Mnih et al; 2016)  Ape-X DQN (Horgan et al; 2018)

\[ X \leftarrow \text{rollout()} \]
\[ d\theta \leftarrow \text{grad}(L, X) \]
\[ \text{sync}(d\theta) \]

\[ \theta \leftarrow \text{sync()} \]
\[ \text{rollout()} \]

\[ X \leftarrow \text{replay()} \]
\[ \text{apply(grad}(L, X)) \]

http://rllib.io
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 $L(\theta, X)$

http://rllib.io
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 \( L(\theta,X) \)

http://rllib.io
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.)

http://rllib.io
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)

Sub-worker (process)

Sub-worker

Sub-sub worker processes

"collect experiences"

"do model-based rollouts"

"allreduce your gradients"

"run K steps of training"

exchange weight shards through Ray object store

http://rllib.io
Unifying system enables RL Abstractions

*Policy Optimizer Abstraction*

SyncSamples → SyncReplay → send experiences → single-node

Cluster → AsyncSamples → MultiGPU ... asynchronous

*Policy Graph Abstraction*

Send experiences → single-node → cluster

Hierarchical Task Model

http://rllib.io
RLlib Abstractions in Action

Policy Optimizers

SyncSamples  SyncReplay  AsyncGradients  AsyncSamples  MultiGPU  ...

{Q-func, n-step, Q-loss}  {LSTM, adv. calc, PG loss}  +actor-critic loss, GAE  +clipped obj.  +V-trace

Policy Gradients


PPO (2017)  PPO (GPU-optimized)

http://rllib.io
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

[Link to RLlib](http://rllib.io)
## RLlib Reference Algorithms

<table>
<thead>
<tr>
<th>Atari env</th>
<th>RLlib IMPALA 32-workers @1 hour</th>
<th>Mnih et al A3C 16-workers @1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeamRider</td>
<td>3181</td>
<td>~1000</td>
</tr>
<tr>
<td>Breakout</td>
<td>538</td>
<td>~10</td>
</tr>
<tr>
<td>Qbert</td>
<td>10850</td>
<td>~500</td>
</tr>
<tr>
<td>SpaceInvaders</td>
<td>843</td>
<td>~300</td>
</tr>
</tbody>
</table>

1 GPU + 64 vCPUs (large single machine)

[http://rllib.io](http://rllib.io)
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)

- Application Support
- Abstractions for RL
- Distributed Execution

http://rllib.io
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()
    pretty_print(result)

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

Tutorial on google Colab: [https://drive.google.com/open?id=1pvE7KvnhYR0Ynqt0J0fzYSmkjILg64Qg](https://drive.google.com/open?id=1pvE7KvnhYR0Ynqt0J0fzYSmkjILg64Qg)
Code example: multi-agent RL

def main():
    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]),
        },
    },
})
```

http://rllib.io
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](http://rllib.io)

Thanks!