Common Computational Patterns for RL

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

for $i$ in range(T):
    $s, a, s_{-1}, r = \text{evaluate}()$
    replay.store($s, a, s_{-1}, r$)

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

    if should_update_target():
        q_network.sync_with(target_net)
2015: General Reinforcement Learning Architecture (GORILA)
GORILA Performance
2016: Asynchronous Advantage Actor Critic (A3C)

# Each worker:

while True:
    sync_weights_from_master()

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

    grad = compute_grad(samples)
    async_send_grad_to_master()

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)

<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...
Distributed Prioritized Experience Replay (Ape-X)

A3C doesn’t scale very well…

**Ape-X:**
1. Distributed DQN/DDPG
2. Reintroduces replay
3. **Distributed Prioritization:** Unlike Prioritized DQN, initial priorities are not set to “max TD”
Figure 2: Left: Atari results aggregated across 57 games, evaluated from random no-op starts. Right: Atari training curves for selected games, against baselines. Blue: Ape-X DQN with 360 actors; Orange: A3C; Purple: Rainbow; Green: DQN. See appendix for longer runs over all games.
Importance Weighted Actor-Learner Architectures (IMPALA)

Motivated by progress in distributed deep learning!
How to correct for Policy Lag? Importance Sampling!

Given an actor-critic model:

1. Apply importance-sampling to policy gradient

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

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

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

---

**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

---

Median time to solve (minutes) vs. Number of CPU cores

- 18 cores, 657 minutes
- 1440 cores, 10 minutes
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
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
    Evolution Strategies: Redis
    A3C: shared memory, multiprocessing, TF
    AlphaZero: custom systems
    IMPALA: Distributed TensorFlow

• 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

<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, MCTS,</td>
</tr>
<tr>
<td>Evolutionary Methods</td>
<td>X</td>
<td></td>
<td>X</td>
<td></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

Agent

Policy: state $\rightarrow$ action

Environment

action ($a_{i+1}$)

state ($s_i$) (observation)

reward ($r_i$)
Structure of RL computations

Agent

Environment

Policy improvement (e.g., SGD)

Policy evaluation (state → action)

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

action \( (a_{i+1}) \)

state \( (s_i) \) (observation)

reward \( (r_i) \)
Many RL loop decompositions

Async DQN (Mnih et al; 2016)

Actor-Learner

Actor-Learner

Actor-Learner

Param Server

X <- rollout()
dθ <- grad(L, X)
sync(dθ)

θ <- sync()

rollout()

Ape-X DQN (Horgan et al; 2018)

Replay

Learner

Actor

Actor

Actor

X <- replay()
apply(grad(L, X))

θ <- sync()
rollout()
Common components

Async DQN (Mnih et al; 2016)

- Param Server
- Actor-Learner
- Actor-Learner
- Actor-Learner

Policy $\pi_\theta(o_t)$
Trajectory postprocessor $\rho_\theta(X)$
Loss $L(\theta,X)$

Ape-X DQN (Horgan et al; 2018)

- Replay
- Learner
- Actor
- Actor
- Actor

http://rllib.io
Common components

Async DQN (Mnih et al; 2016)

Actor-Learner
Actor-Learner
Actor-Learner

Policy $\pi_\theta(o_t)$
Trajectory postprocessor $\rho_\theta(X)$
Loss $L(\theta,X)$

Ape-X DQN (Horgan et al; 2018)

Replay
Learner
Actor
Actor
Actor
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

- Top-level worker (Python process)
- Sub-worker (process)
- Sub-sub worker processes

"collect experiences"
"do model-based rollouts"
exchange weight shards through Ray object store

"run K steps of training"
"allreduce your gradients"
Unifying system enables RL Abstractions

**Policy Optimizer Abstraction**

- SyncSamples
- SyncReplay
- AsyncGradients
- AsyncSamples
- MultiGPU
- asynchronous

**Policy Graph Abstraction**

- \( \pi_\theta, \rho_\theta, L(\theta, X) \)
- \{Q-func, n-step, Q-loss\}
- \{LSTM, adv. calc, PG loss\}

Examples:

- Hierarchical Task Model

single-node

- send experiences

cluster

- send gradients

Hierarchical Task Model
RLlib Abstractions in Action

Policy Optimizers

Policy Graphs

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

SyncSamples  →  SyncReplay  →  AsyncGradients  →  AsyncSamples  →  MultiGPU  →  ...

- DQN (2015)  
- Async DQN (2016)  
- Ape-X (2018)  
- A2C (2016)  
- A3C (2016)  
- PPO (2017)  
- IMPALA (2018)  
- PPO (GPU-optimized)
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

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)
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)

- OpenAI Gym
- Multi-Agent
- Policy Serving
- Offline Data
- Custom Algorithms
- RLlib Algorithms
- RLlib Abstractions
- Ray Tasks and Actors

(1) Application Support
(2) Abstractions for RL
(3) Distributed Execution
import ray
import ray.rllib.agents.ppo as ppo
from ray.tune.logger import pretty_print

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

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

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

    if i % 100 == 0:
        checkpoint = agent.save()
        print("checkpoint saved at", checkpoint)
Code example: multi-agent RL

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

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

http://rllib.io
Code example: hyperparam tuning

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

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!
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 multiprocessing on a single node
- Competitive with MPI in many workloads