RLlib Flow
Distributed Reinforcement Learning is a Dataflow Problem

Eric Liang*, Zhanghao Wu*, Michael Luo, Sven Mika, Joseph E. Gonzalez, Ion Stoica

UC Berkeley, Anyscale
Deep Reinforcement Learning

- Reinforcement learning can be defined in high-level update equations.
- The implementation have remained quite low-level, i.e. at the level of message passing.
## Needs of RL Researchers

<table>
<thead>
<tr>
<th>Library</th>
<th>Distribution Scheme</th>
<th>Generality</th>
<th>Programmability</th>
<th>#Algo</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLGraph</td>
<td>Pluggable</td>
<td>General Purpose</td>
<td>Low-level / Pluggable</td>
<td>10+</td>
</tr>
<tr>
<td>Deepmind Acme</td>
<td>Actors + Reverb</td>
<td>Async Actor-Learner</td>
<td>Limited</td>
<td>10+</td>
</tr>
<tr>
<td>Intel Coach</td>
<td>Actor + NFS</td>
<td>Async Actor-Learner</td>
<td>Limited</td>
<td>30+</td>
</tr>
<tr>
<td>RLlib</td>
<td>Ray Actors</td>
<td>General Purpose</td>
<td>Flexible, but Low-level</td>
<td>20+</td>
</tr>
<tr>
<td>RLlib Flow</td>
<td>Actor / Dataflow</td>
<td>General Purpose</td>
<td>Flexible and High-level</td>
<td>20+</td>
</tr>
</tbody>
</table>

- RL practitioners are typically **not system engineers**
- RL algorithms should be **customizable** in various ways
# launch gradients computation tasks

```python
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients.remote(worker.sample.remote())
    pending_gradients[future] = worker
```

# asynchronously gather gradients and apply

```python
while pending_gradients:
    wait_results = ray.wait(pending_gradients.keys(), num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]
    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)
```

# launch gradient computation again

```python
future = worker.compute_gradients.remote(worker.sample.remote())
pending_gradients[future] = worker
```

---

**RL Implementation Remains Low Level**

**Data Flow**

**Worker Management**

**Execution Logic**

**A3C Implementation in RLlib**
# launch gradients computation tasks
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients.remote(worker.sample.remote())
    pending_gradients[future] = worker

# asynchronously gather gradients and apply
while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]
    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    # apply gradients
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)

# launch gradient computation again
future = worker.compute_gradients.remote(worker.sample.remote())
pending_gradients[future] = worker

A3C Implementation in RLlib
# launch gradients computation tasks

```python
pending_gradients = dict()

for worker in remote_workers:
    weights = local_worker.get_weights()
    future = worker.compute_gradients.remote(worker.sample.remote())
    pending_gradients[future] = worker

# asynchronously gather gradients and apply

while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]

    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    # apply gradients
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)

    # launch gradient computation again
    future = worker.compute_gradients.remote(worker.sample.remote())
    pending_gradients[future] = worker
```

---

**Data Flow**

**Worker Management**

**Execution Logic**

---

**RL Implementation Remains Low Level**

**A3C Implementation in RLlib**
# launch gradients computation tasks

```python
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients.remote(worker.sample.remote())
    pending_gradients[future] = worker
```

# asynchronously gather gradients and apply

```python
while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]
    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)
```

# launch gradient computation again

```python
future = worker.compute_gradients.remote(worker.sample.remote())
pending_gradients[future] = worker
```

## RL Implementation Remains Low Level

- **Data Flow**
- **Worker Management**
- **Execution Logic**

Hard to read, customize and optimize

A3C Implementation in RLlib
Complex Algorithms for RL

- Complex algorithms possible but require low-level code
  - Ape-X: 250 lines of Python
  - IMPALA: 694 lines of Python

How can we reduce the lines of code required to define a new distributed algorithm?
Multi-Agent Use Cases

● From the systems perspective, multi-agent training often does not impact distributed execution

● Exceptions:
  ○ Training agents different optimization frequencies
  ○ Training agents with different distributed algorithms

How can we support **composing existing RL algorithms** without requiring a rewrite?
Reinforcement Learning Basics

- RL is more like **data analytics** than supervised learning.
- We can view RL training as **dataflow**
Dataflow of Synchronous Training Loop

- Bulk synchronous algorithms like A2C, PPO.
Dataflow of Asynchronous Training

- Small change for *async* optimization (A3C)

Parallel Rollouts → Compute Gradients → Apply Gradients → Report Metrics

Remove Sync Barrier

Update Weights
Dataflow of Distributed Prioritized DQN

- Mixed async dataflow (Ape-X), with fine-grained updates
(a) Dataflow Operators for RL

From Actors

Parlter[T]

Actor

Send Message

(a) Creation & Message Passing
Dataflow Operators for RL

Parallel Apply  Sequential Apply

(b) Transformation
Dataflow Operators for RL

Async Gather (No Barrier)

Bulk Sync Gather (Full Barrier)

(c) Sequencing
A Dataflow Programming Model for Distributed RL

(d) Concurrency
A Dataflow Programming Model for Distributed RL

(a) Creation & Message Passing
- From Actors
- Send Message

(b) Transformation
- Parallel Apply
- Sequential Apply

(c) Sequencing
- Async Gather (No Barrier)
- Bulk Sync Gather (Full Barrier)

(d) Concurrency
- Duplicate
- Union
- Async Union
Implementation over Distributed Actor Framework

- Two separate modules: A general purpose **parallel iterator library**; a collection of RL specific **dataflow operators**

**RLlib Flow**
- RLlib Flow Operators (1118 lines of code)
- Parallel Iterator Library (1241 lines of code)
- Distributed Actor System

©2021 RiSELab
Evaluation: Revisiting A3C

(b) Transformation (Parallel Apply)
(c) Sequencing (Async Gather)
(b) Transformation (Sequential Apply)

(a) Creation (From Actor)
Update Weights
(a) Message Passing (Send Message)
Evaluation: A3C Comparison

A3C Implementation in RLlib Flow

1. # type: List[RolloutActor]
2. workers = create_rollout_workers()
3. # type: Iter[Gradients]
4. grads = ParallelRollouts(workers)
5. \texttt{par_for_each}(\texttt{ComputeGradients()})
6. \texttt{.gather_async()}
7. # type: Iter[TrainStats]
8. apply_op = grads
9. \texttt{for_each}(\texttt{ApplyGradients(workers)})
10. # type: Iter[Metrics]
11. return ReportMetrics(apply_op, workers)

A3C Implementation in Previous RLlib

1. # launch gradients computation tasks
2. pending_gradients = dict()
3. for worker in remote_workers:
4. \texttt{worker.set_weights.remote(\texttt{weights})}
5. future = worker.compute_gradients\texttt{.remote(\texttt{worker.sample.remote()})}
6. pending_gradients[future] = worker
7. # asynchronously gather gradients and apply
8. while pending_gradients:
9. wait_results = ray\texttt{.wait(}
10. pending_gradients\texttt{.keys()}
11. num_returns=1)
12. ready_list = wait_results[0]
13. future = ready_list[0]
14. gradient, info = ray\texttt{.get(future)}
15. worker = pending_gradients.pop(future)
16. # apply gradients
17. local_worker\texttt{.apply_gradients(gradient)}
18. weights = local_worker\texttt{.get_weights()}
19. worker.set_weights\texttt{.remote(\texttt{weights})}
20. # launch gradient computation again
21. future = worker.compute_gradients\texttt{.remote(\texttt{worker.sample.remote()})}
22. pending_gradients[future] = worker
Evaluation: Revisiting Ape-X

(a) Creation (From Actor)

(b) Transformation (Sequential Apply)

(c) Sequencing (Async Gather)

(d) Concurrency (Asynyc Union)

- Parallel Rollouts
- Store to Buffer
- Update Weights
- Update Priorities
- Optimize Policy
- Async Union
- Report Metrics

(a) Message Passing (Send Message)
Evaluation: Readability (Ape-X)

```python
workers = create_rollout_workers()
replay_buffer = create_replay_actors()
rollouts = ParallelRollouts(workers).gather_async()

store_op = rollouts.for_each(StoreToBuffer(replay_buffer)).for_each(UpdateWeights(workers))
replay_op = ParallelReplay(replay_buffer).gather_async().for_each(UpdatePriorities(workers)).for_each(TrainOneStep(workers))

return ReportMetrics(Union(store_op, replay_op), workers)
```
Evaluation: Readability (Ape-X)

- Previous implementation:
Evaluation: Composing Multiple Workflows

DQN Sub-Flow

- Replay from Buffer (batch_size=32)
- Optimize (policy="DQN")
- Update Target Network
- Union
- Async Union
- Report Metrics

PPO Sub-Flow

- Select Experiences (policy="PPO")
- Concatenate (batch_size=4096)
- Optimize (policy="PPO")
- Weight Updates
- Parallel Rollouts

(d) Concurrency (Duplicate)
(d) Concurrency (Union)
(d) Concurrency (Asnc Union)
# type: List[RolloutActor]
workers = create_rollout_workers()

# type: Iter[Rollout], Iter[Rollout]
r1, r2 = ParallelRollouts(workers).split()

# type: Iter[TrainStats], Iter[TrainStats]
ppo_op = ppo_plan(
    Select(r1, policy="PPO"), workers)

dqn_op = dqn_plan(
    Select(r2, policy="DQN"), workers)

# type: Iter[Metrics]
return ReportMetrics(
    Union(ppo_op, dqn_op), workers)
## Evaluation: Readability

- Lines of code saved for RLlib algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>RLlib</th>
<th>RLlib Flow</th>
<th>+shared</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3C</td>
<td>87</td>
<td>11</td>
<td>52</td>
<td>1.6-9.6×</td>
</tr>
<tr>
<td>A2C</td>
<td>154</td>
<td>25</td>
<td>50</td>
<td>3.1-6.1×</td>
</tr>
<tr>
<td>DQN</td>
<td>239</td>
<td>87</td>
<td>139</td>
<td>1.7-2.7×</td>
</tr>
<tr>
<td>PPO</td>
<td>386</td>
<td>79</td>
<td>225</td>
<td>1.7-4.8×</td>
</tr>
<tr>
<td>Ape-X</td>
<td>250</td>
<td>126</td>
<td>216</td>
<td>1.1-1.9×</td>
</tr>
<tr>
<td>IMPALA</td>
<td>694</td>
<td>89</td>
<td>362</td>
<td>1.9-7.8×</td>
</tr>
<tr>
<td>MAML</td>
<td>370*</td>
<td>136</td>
<td>136</td>
<td>2.7×</td>
</tr>
</tbody>
</table>
Performance against RLlib

The abstraction of RLlib Flow does not introduce overhead
Reinforcement Learning vs Data Streaming

- Asynchronous Dependencies (pink): no deterministic ordering
- Message Passing (pink dotted): update upstream operator state
- Consistency and Durability: less strict requirements
Performance against Spark Streaming

- **Lower-overhead** than streaming frameworks -- take advantage of RL requirements vs. data processing
RLlib Flow

Distributed Reinforcement Learning is a Dataflow Problem

RLlib Flow Operators:
1. Creation & Message Passing
2. Transformation
3. Sequencing
4. Concurrency

Lines of Code

RLlib
RLlib Flow
+shared
Ratio
A3C 87 11 52 1.6-9.6×
A2C 154 25 50 3.1-6.1×
DQN 239 87 139 1.7-2.7×
PPO 386 79 225 1.7-4.8×
Ape-X 250 126 216 1.1-1.9×
IMPALA 694 89 362 1.9-7.8×
MAML 370* 136 136 2.7×

Lines of Code

Comparison to Spark

Architecture of RLlib Flow

DQNSub-Flow

PPO Sub-Flow

Dataflow of A3C

Dataflow of Ape-X

Dataflow of Multi-Agent

Correspondence to:
Eric Liang<ericliang@berkeley.edu>,
Ion Stoica<istoica@berkeley.edu>