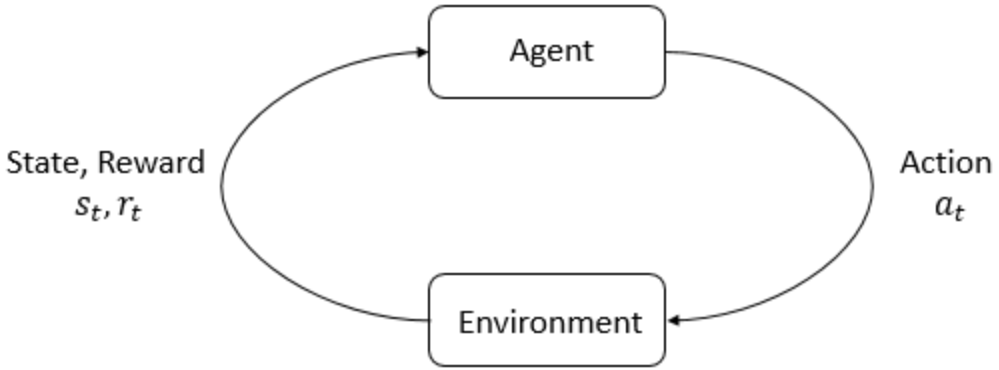
MSRL: Distributed Reinforcement Learning with Dataflow Fragments

Review

✳: Figure from paper /codebase

Motivation

Reinforcement Learning



Source: OpenAI SpinningUp

Reinforcement Learning (Distributed, Multi-agent)

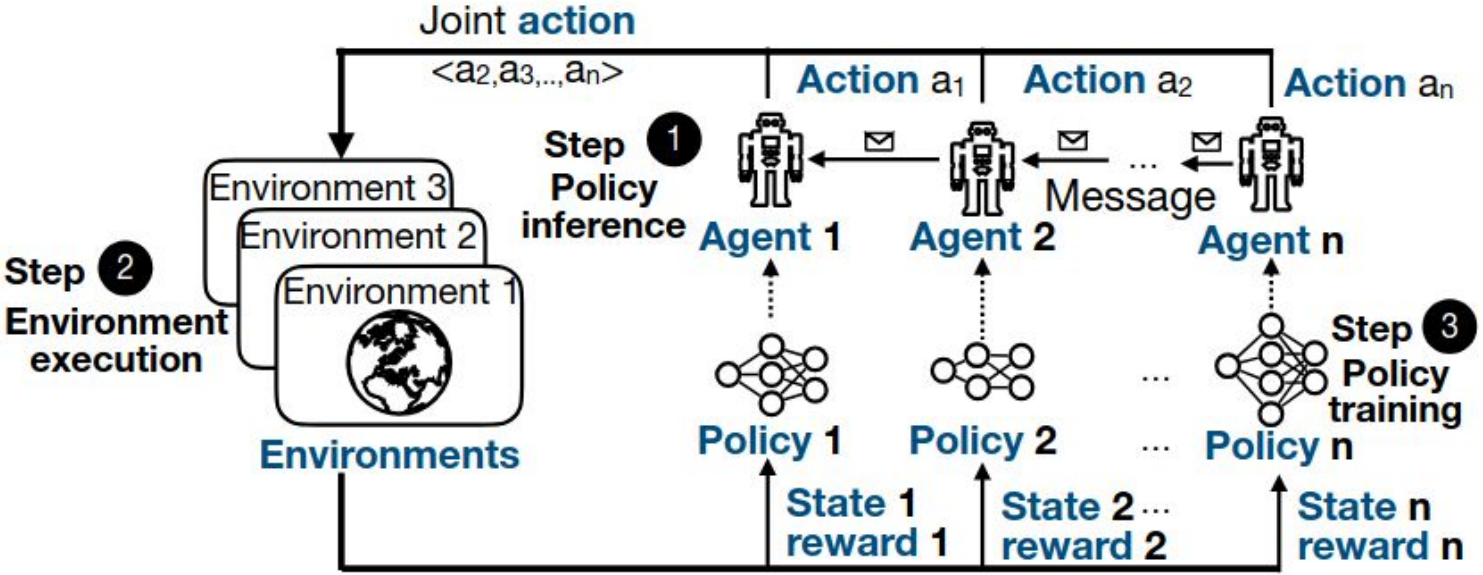


Fig. 1: RL training loop with multiple agents

Heterogenous Training

PPO, DQN, Dreamer, MAPPO, MuZero …

Heterogeneous deployment

A3C, IMPALA, self-play, CPU environments, GPU inference, distributed agents …

Existing Solutions

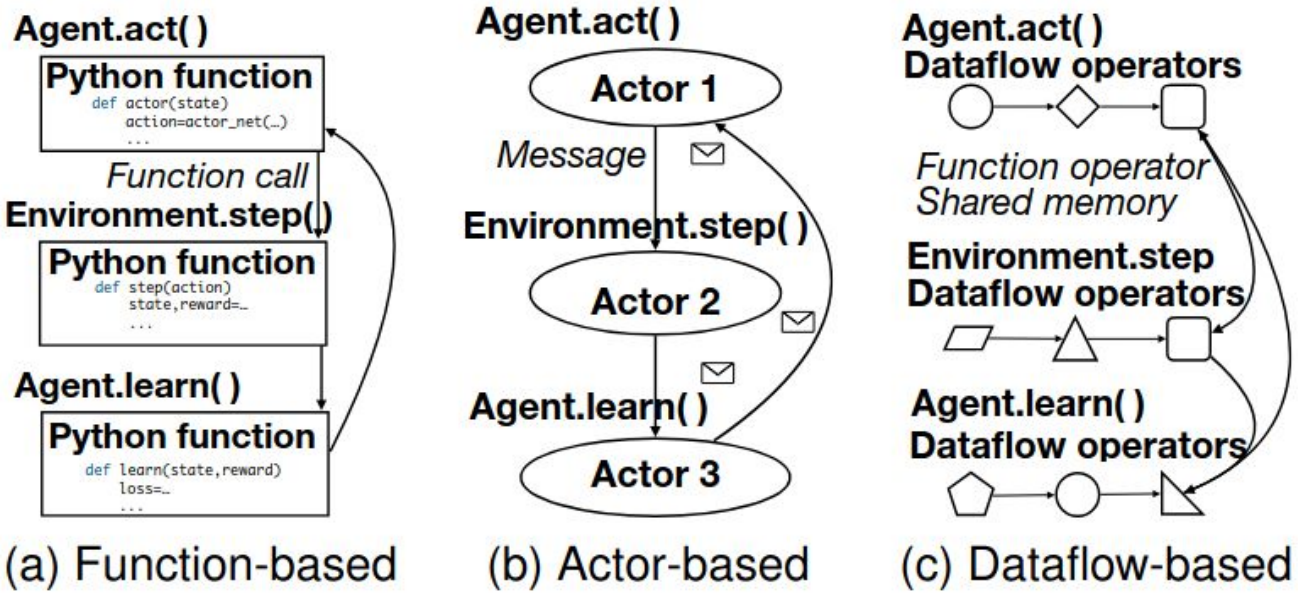


Fig. 2: Types of RL system designs

Existing Solutions (Function)

Example: CleanRL

Few abstractions

Direct implementation

Fixed distribution

Limited acceleration

```
# ALGO LOGIC: training.
if global_step > args.learning_starts:
    if global_step % args.train_frequency == 0:
        data = rb.sample(args.batch_size)
        with torch.no_grad():
            target_max, _ = target_network(data.next_observations).max(dim=1)
            td_target = data.rewards.flatten() + args.gamma * target_max * (1 - data.dones.flatten())
        old_val = q_network(data.observations).gather(1, data.actions).squeeze()
        loss = F.mse_loss(td_target, old_val)

        if global_step % 100 == 0:
            writer.add_scalar("losses/td_loss", loss, global_step)
            writer.add_scalar("losses/q_values", old_val.mean().item(), global_step)
            print("SPS:", int(global_step / (time.time() - start_time)))
            writer.add_scalar("charts/SPS", int(global_step / (time.time() - start_time)), global_step)

        # optimize the model
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # update target network
    if global_step % args.target_network_frequency == 0:
        for target_network_param, q_network_param in zip(target_network.parameters(), q_network.parameters()):
            target_network_param.data.copy_(
                args.tau * q_network_param.data + (1.0 - args.tau) * target_network_param.data
            )
```

Existing Solution (Agent)

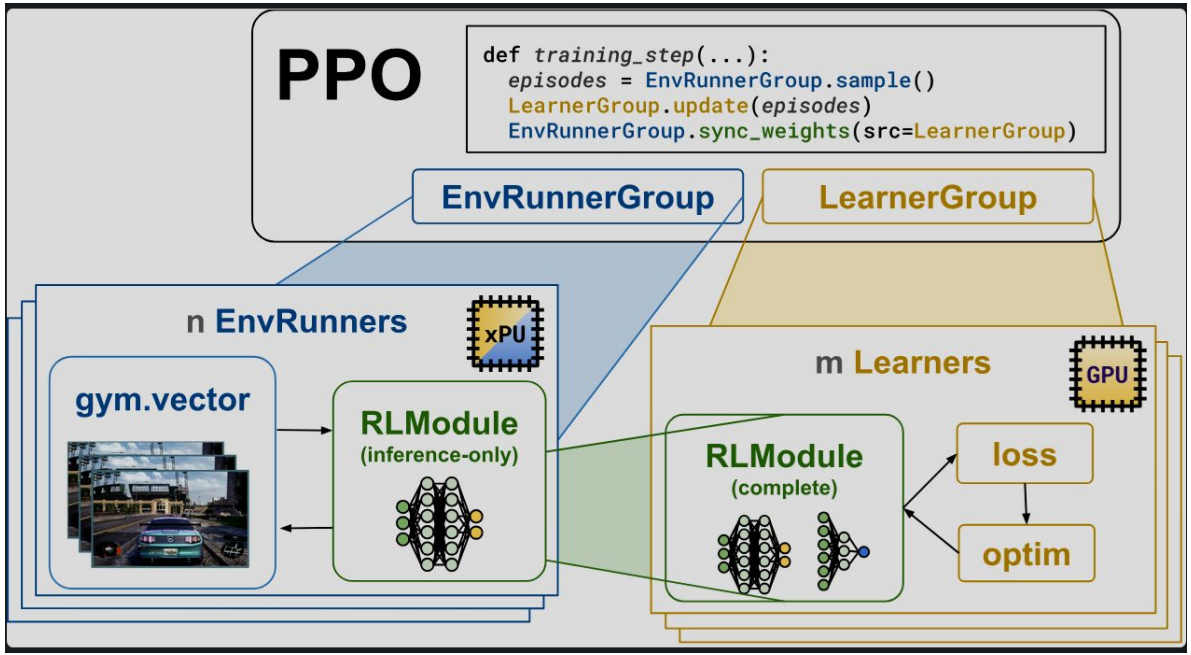
Example: RLlib

Wrapper layer on top of Ray

Distributed agents with message passing

Difficult to reason with low level API

Difficult to optimize

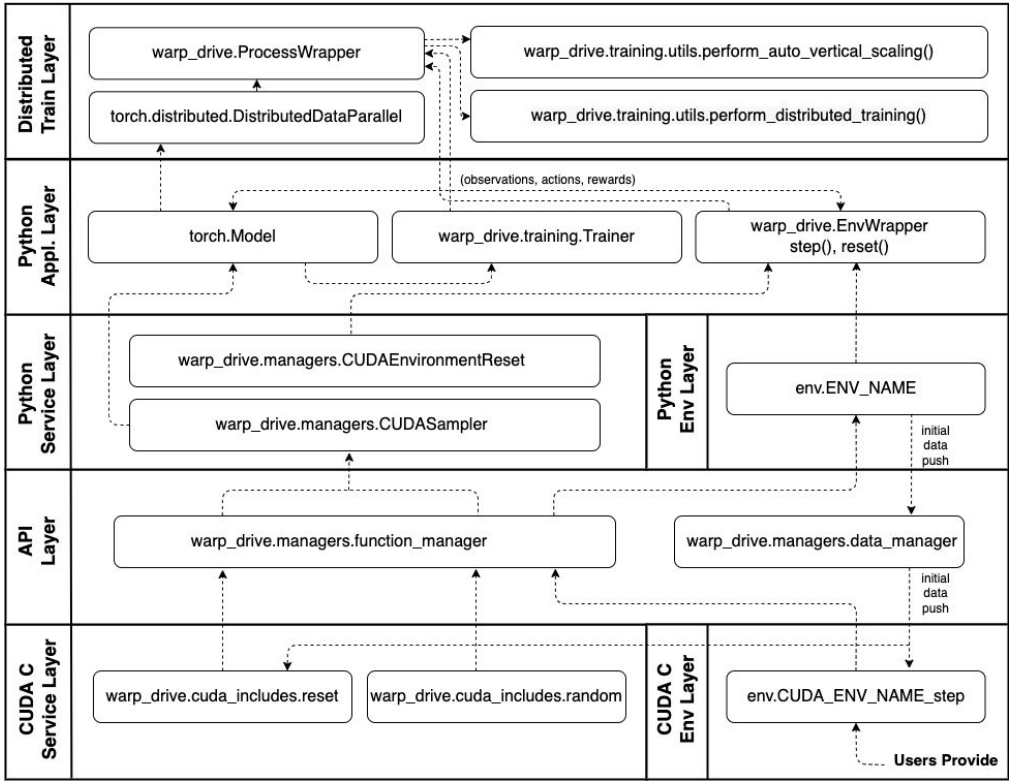


Existing Solution (Dataflow)

Example: WarpDrive

Write code using bespoke dataflow implementations

Fixed distribution



Contribution: MindSporeRL

Compilation

Distribution Policy

A high level API decouples RL logic from deployment.

Component APIs specify algorithmic components with natural boundaries (actors, learners, trainers)

Interaction APIs offer RL-specific functionality (replay buffers)

The Python AST in inspected and compiled into a fragmented dataflow graph (FDG).

Nodes are higher level encapsulations of potentially data-parallel components, with defined communication interfaces.

At runtime, a **coordinator** generates and dispatches the FDG from the source code and a **distribution policy**.

A **fragment optimizer** fuses fragment instances sharing an execution backend before submission.

MSRL API

Type	API	Description
Component	`Agent, Actor, Learner, Trainer`	Abstract classes for components
	`Actor.act(...)`	Trajectory collection
	`Learner.learn(...)`	DNN policy training
	`Trainer.train(...)`	RL training loop
	`MSRL.agent_act(...)`	Invoke actor
	`MSRL.agent_learn(...)`	Invoke learner
	`MSRL.env_step(...)`	Execute environment
	`MSRL.env_reset()`	Reset environment
Interaction	`MSRL.replay_buffer_insert(...)`	Store trajectories in buffer
	`MSRL.replay_buffer_sample()`	Sample trajectories from buffer

Tab. 2: MSRL APIs

MSRL API

Agent: actors and learners.

Actor: Trajectory collection by interaction with the environment.

Learner: DNN update logic.

Trainer: RL Training loop.

Fragmented Dataflow Graph

A dual-layer dataflow graph.

Nodes are **fragments**: higher level abstraction of a component. Each fragment can have a lower level bespoke dataflow implementation. Fragment allocation supports computation of different devices (and optimisation)

Two dimensions of optimisation affects device utilisation

- Fragment granularity
- Fragment co-location

```python
def create_fragment(self, frag_file=None) -> list:
    """Main function to create fragment."""
    # 1 read source code and transform to ast module. `train.py` and `template.py`
    ast_source = self.generate_ast_from_py(self.src_file)
    ast_target = self.generate_ast_from_py(self.template)
    # 2 Split source func `train_one_episode` in Class Trainer and insert to ast_target.
    ast_target = self.split_trainer_to_fragment(ast_target, ast_source)
    # 3 Get parameter list from distribution policy adn insert communication kernels to ast_target.
    parameter_list = self.interface_parser()
    ast_target = self.insert_communication_states(
        ast_target, ast_source, parameter_list, self.policy
    )
    # 4 Unparse the ast module to python code, and create the module of Actor and Learner.
    if frag_file is None:
        frag_name = self.save_fragment(ast_target)
    else:
        frag_name = frag_file.strip(".py")
        print(f"Import fragment from file: {frag_name}.")
    fragment_module = importlib.import_module(frag_name)
    actor = getattr(fragment_module, "Actor")
    learner = getattr(fragment_module, "Learner")
    # 5 Create the fragments list across to the policy topology.
    fragment_list = []
    if list(self.policy.topology.keys())[0] == "Actor":
        for _ in range(self.worker_num - 1):
            fragment_list.append(actor)
            fragment_list.append(learner)
    else:
        fragment_list.append(learner)
        for _ in range(self.worker_num - 1):
            fragment_list.append(actor)
    print("Fragments generated: ", fragment_list)
    self.clean_files([self.template])
    return fragment_list
```

Fragmented Dataflow Graph

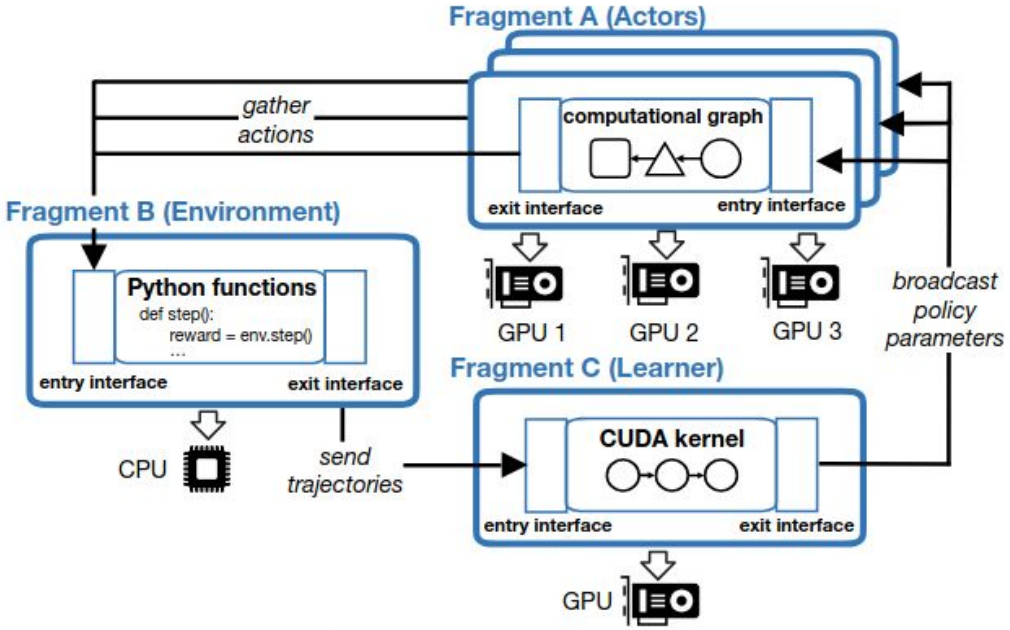


Fig. 3: Fragmented dataflow graph

Distribution Policy

A distribution policy governs how MSRL distributes and executes a given RL policy.

Allocation, replication, and co-location of fragments.

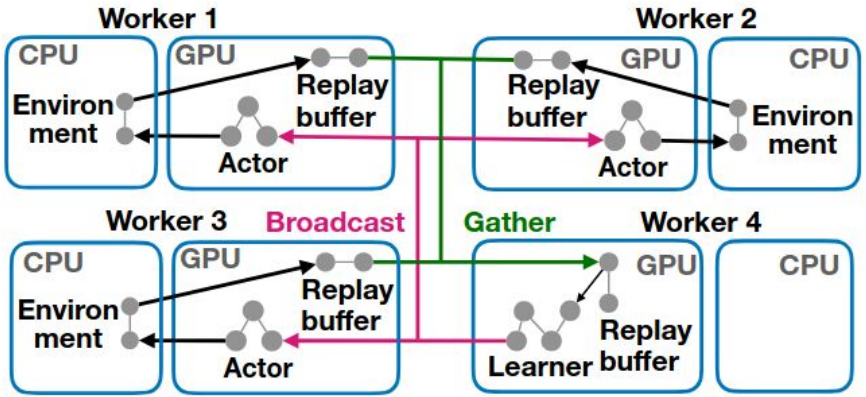
Advantage: Since no single distribution policy is optimal in all cases, the choice provides greater execution flexibility.

After distribution, the **fragment optimiser** can transform the FDG AST before submission to the DNN executor engine, such as batching tensors.

Computationally expensive environment,
small DNN

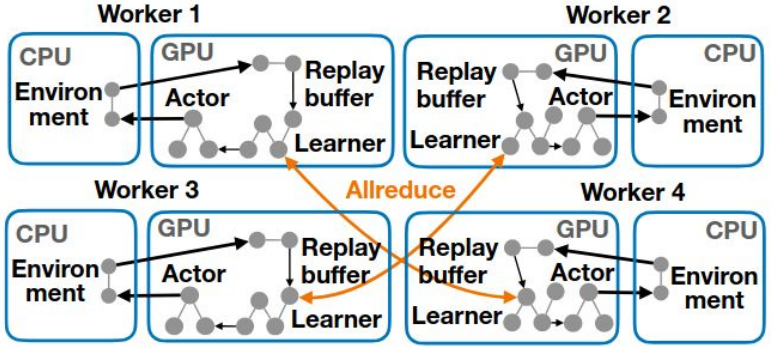
DP-SingleLearnerCoarse

replicate: *actor*,*env* split: *learner* e.g., Acme [18], Sebulba [16]



DP-MultiLearner

replicate: fused *actor/learner*, *env*



Tab. 3: Sample distribution policies with deployments

Lots of data, can't fit onto a single GPU

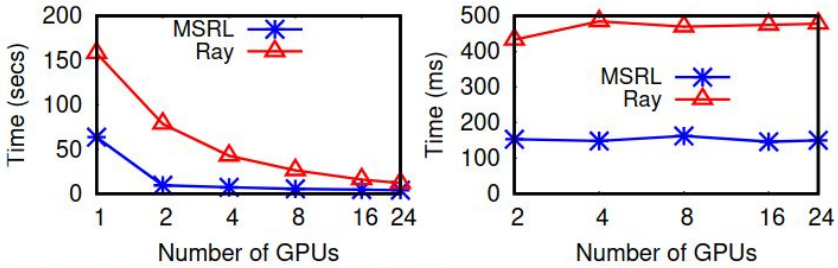
Experiments (Ray)

a)

Proximal Policy Optimisation (PPO), 320 environments distributed evenly among actors. A single learner trains the DNN.

Time dominated by environment computation.

Speedup: FDG Optimisation performs **fusion** , where multiple policy inference is combined into a single step on the GPU. In contrast, Ray performs inference sequentially.



(a) Episode time vs. GPUs (PPO) (b) Episode time vs. GPUs (A3C)

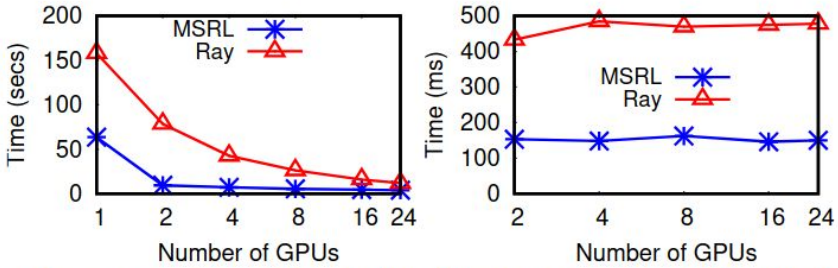
Fig. 6: Performance comparison with Ray

Experiments (Ray)

b)

A3C. Multiple copies of agents execute their own copy of the environment.

Speedup: the distribution policy can exploit customize DL engine asynchronous send / receive operations to avoid data copies between CPU and GPU. i.e. compared to Ray, the communication policy between nodes is more flexible.



(a) Episode time vs. GPUs (PPO) (b) Episode time vs. GPUs (A3C)

Fig. 6: Performance comparison with Ray

Experiments (WarpDrive)

GPU only training on *tag* environment.

Speedup from improved compiler optimisation compared to handwritten CUDA code.

Furthermore, fragments can scale to multiple GPUs (WarpDrive cannot).

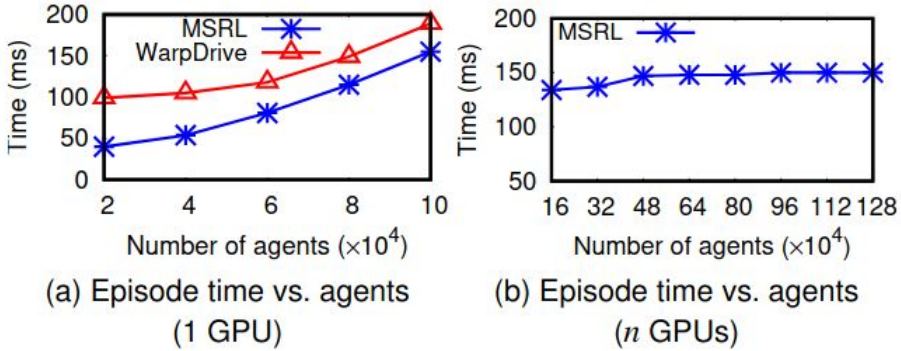


Fig. 7: Performance comparison with WarpDrive (PPO)

Experiments (Distribution)

Dependent on the algorithm and configuration bottleneck, different distribution policies work well.

Communication overhead and parallelism properties are unique.

Potential for automatic optimisation of DP.

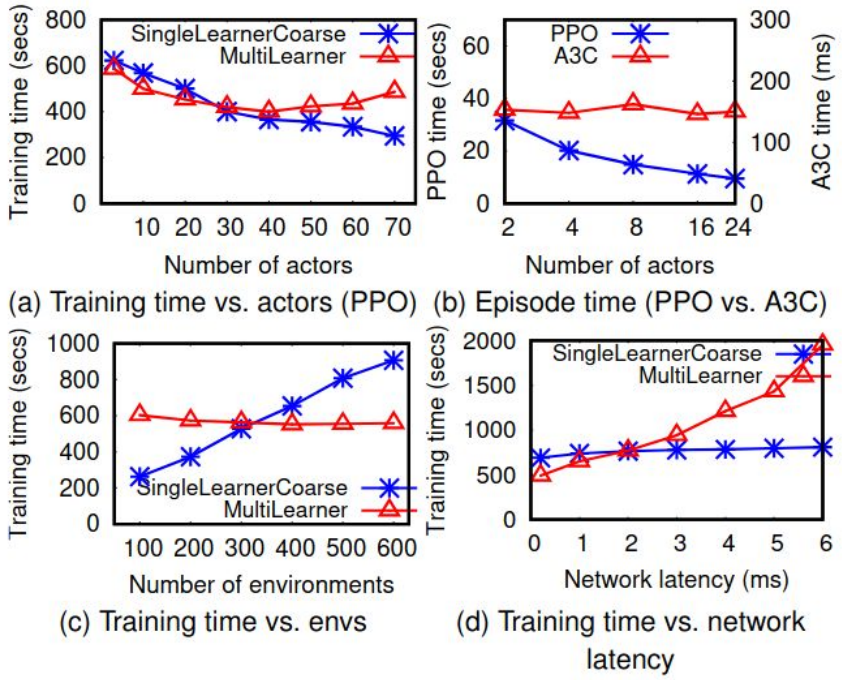
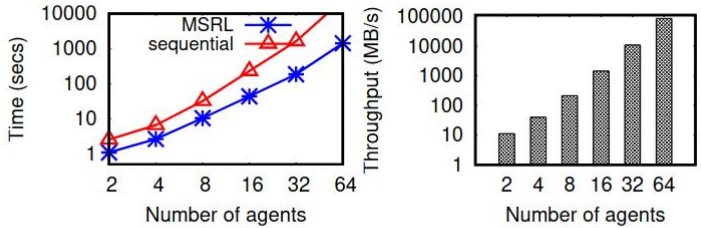


Fig. 8: Impact of parameters on distribution policies

Experiments (Scalability)

MSRL approach does not bottleneck with increased number of data-intensive agents. It is able to take advantage of distributed training resources.



(a) Training time per episode (b) Training throughput

Fig. 10: Scalability with agent count (MAPPO)

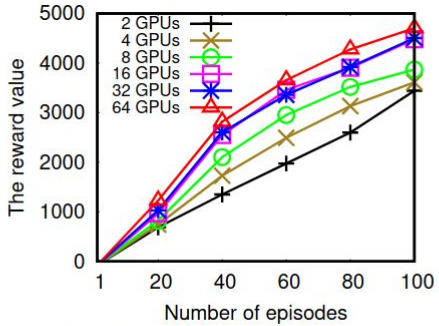


Fig. 11: Statistical efficiency with environment count (PPO)

Opinion

Expressivity

High-level API sufficiently covers a large range of algorithms. However, control over fragments is limited by the API boundaries and compiler.

But this is a **training framework …**

Does it cover:

- Safety checks?
- RLHF? (Probably not, MindSpore has an independent library)
- Custom data sources / replay buffer storage?
- Logging / metric integration?
- Configuration management?

Algorithm

Algorithm	RL Version	Action Space		Device			Example Environment
		Discrete	Continuous	CPU	GPU	Ascend	
DQN	>= 0.1	✓	/	✓	✓	✓	CartPole-v0
PPO	>= 0.1	/	✓	✓	✓	✓	HalfCheetah-v2
AC	>= 0.1	✓	/	✓	✓	✓	CartPole-v0
A2C	>= 0.2	✓	/	✓	✓	✓	CartPole-v0
DDPG	>= 0.3	/	✓	✓	✓	✓	HalfCheetah-v2
QMIX	>= 0.5	✓	/	✓	✓	✓	SMAC, Simple Spread
SAC	>= 0.5	/	✓	✓	✓	✓	HalfCheetah-v2
TD3	>= 0.6	/	✓	✓	✓	✓	HalfCheetah-v2
C51	>= 0.6	✓	/	✓	✓	✓	CartPole-v0
A3C	>= 0.6	✓	/	/	✓	✓	CartPole-v0
CQL	>= 0.6	/	✓	✓	✓	✓	Hopper-v0
MAPPO	>= 0.6	✓	/	✓	✓	✓	Simple Spread
GAIL	>= 0.6	/	✓	✓	✓	✓	HalfCheetah-v2
MCTS	>= 0.6	✓	/	✓	✓	/	Tic-Tac-Toe
AWAC	>= 0.6	/	✓	✓	✓	✓	Ant-v2
Dreamer	>= 0.6	/	✓	/	✓	✓	Walker-walk
IQL	>= 0.6	/	✓	✓	✓	✓	Walker2d-v2
MADDPG	>= 0.6	✓	/	✓	✓	✓	simple_spread
Double DQN	>= 0.6	✓	/	✓	✓	✓	CartPole-v0
Policy Gradient	>= 0.6	✓	/	✓	✓	✓	CartPole-v0
Dueling DQN	>= 0.6	✓	/	✓	✓	✓	CartPole-v0

Training Backend

- As a training paradigm, much of the empirical improvements actually comes from reliance on **MindSpore** DNN framework
- Optimisations and distributions only work with shared MindSpore dataflow normalisation framework as underlying.
- Did not see examples of “containerised” fragments - would development of such fragments actually be easy?

Engineering and Community

MSRL advertises as a practical training framework, not just a research project.

Limited mentions in both Chinese and Western communities.

Last version update January. No significant changes since. No active issues or pull requests.

Implementation likely not industrial quality. For example, there is only a limited test suite.

Summary

Strengths:

- Decouples RL logic and deployment via the FDG computation model
- Enables flexibility and further optimisation

Weaknesses:

- Expressivity does not cover non-RL modules
- Limited uptake by the community / as an industrial tool
- Requires single backend for majority of optimisations
- Experiments shown in paper cover only small subset of scenarios, no large scale benchmarking found as in other similar systems