maro

MARO Team

Nov 05, 2020

INSTALLATION

1 Key Components **3**

2 Quick Start **5**

3 Contents **7**

3.1 Package . . . 7
3.2 Playground Docker Image . . . 8
3.3 Grass Cluster Provisioning on Azure . . . 9
3.4 K8S Cluster Provisioning on Azure . . . 12
3.5 Container Inventory Management (CIM) . . . 16
3.6 Bike Repositioning (Citi Bike) . . . 23
3.7 Multi Agent DQN for CIM . . . 32
3.8 Greedy Policy for Citi Bike . . . 36
3.9 Simulation Toolkit . . . 38
3.10 Data Model . . . 39
3.11 Event Buffer . . . 43
3.12 Business Engine . . . 44
3.13 RL Toolkit . . . 45
3.14 Distributed Toolkit . . . 48
3.15 Distributed Communication . . . 49
3.16 Distributed Orchestration . . . 52

Multi-Agent Resource Optimization (MARO) platform is an instance of Reinforcement learning as a Service (RaaS) for real-world resource optimization. It can be applied to many important industrial domains, such as container inventory management in logistics, bike repositioning in transportation, virtual machine provisioning in data centers, and asset management in finance. Besides Reinforcement Learning (RL), it also supports other planning/decision mechanisms, such as Operations Research.

CHAPTER

ONE

KEY COMPONENTS

- Simulation toolkit: it provides some predefined scenarios, and the reusable wheels for building new scenarios.
- RL toolkit: it provides a full-stack abstraction for RL, such as agent manager, agent, RL algorithms, learner, actor, and various shapers.
- Distributed toolkit: it provides distributed communication components, interface of user-defined functions for message auto-handling, cluster provision, and job orchestration.

CHAPTER

TWO

QUICK START

```
from maro.simulator import Env
from maro.simulator.scenarios.cim.common import Action

# Initialize an environment with a specific scenario, related topology.
# In Container Inventory Management, 1 tick means 1 day, here durations=100 means a␣
↪length of 100 days
env = Env(scenario="cim", topology="toy.5p_ssddd_l0.0", start_tick=0, durations=100)

# Query environment summary, which includes business instances, intra-instance␣
↪attributes, etc.
print(env.summary)

for ep in range(2):
    # Gym-like step function.
    metrics, decision_event, is_done = env.step(None)

    while not is_done:
        past_week_ticks = [
            x for x in range(decision_event.tick - 7, decision_event.tick)
        ]
        decision_port_idx = decision_event.port_idx
        intr_port_infos = ["booking", "empty", "shortage"]

        # Query the snapshot list of the environment to get the information of
        # the booking, empty container inventory, shortage of the decision port in␣
↪the past week.
        past_week_info = env.snapshot_list["ports"][
            past_week_ticks : decision_port_idx : intr_port_infos
        ]

        dummy_action = Action(
            vessel_idx=decision_event.vessel_idx,
            port_idx=decision_event.port_idx,
            quantity=0
        )

        # Drive environment with dummy action (no repositioning).
        metrics, decision_event, is_done = env.step(dummy_action)

    # Query environment business metrics at the end of an episode,
    # it is your optimized object (usually includes multi-target).
    print(f"ep: {ep}, environment metrics: {env.metrics}")
    env.reset()
```

CHAPTER

THREE

CONTENTS

3.1 Package

3.1.1 Install MARO from PyPI

- Max OS / Linux

```
pip install pymaro
```

- Windows

```
# Install torch first, if you don't have one.
pip install torch===1.6.0 torchvision===0.7.0 -f https://download.pytorch.org/whl/
↪torch_stable.html

pip install pymaro
```

3.1.2 Install MARO from Source (Editable Mode)

- Prerequisites
  - Python >= 3.6, < 3.8
  - C++ Compiler
    - Linux or Mac OS X: `gcc`
    - Windows: Build Tools for Visual Studio 2017
- Enable Virtual Environment
  - Mac OS / Linux

```
# If your environment is not clean, create a virtual environment firstly.
python -m venv maro_venv
source ./maro_venv/bin/activate
```

  - Windows

```
# If your environment is not clean, create a virtual environment firstly.
python -m venv maro_venv
.\maro_venv\Scripts\activate
```

- Install MARO

- Mac OS / Linux

```
# Install MARO from source.
bash scripts/install_maro.sh
```

- Windows

```
# Install MARO from source.
.\scripts\install_maro.bat
```

3.2 Playground Docker Image

3.2.1 Pull from Docker Hub

```
# Run playground container.
# Redis commander (GUI for redis) -> http://127.0.0.1:40009
# Local host docs -> http://127.0.0.1:40010
# Jupyter lab with maro -> http://127.0.0.1:40011
docker run -p 40009:40009 -p 40010:40010 -p 40011:40011 arthursjiang/maro:cpu
```

3.2.2 Run from Source

- Mac OS / Linux

```
# Build playground image.
bash ./scripts/build_playground.sh

# Run playground container.
# Redis commander (GUI for redis) -> http://127.0.0.1:40009
# Local host docs -> http://127.0.0.1:40010
# Jupyter lab with maro -> http://127.0.0.1:40011
docker run -p 40009:40009 -p 40010:40010 -p 40011:40011 maro/playground:cpu
```

- Windows

```
# Build playground image.
.\scripts\build_playground.bat

# Run playground container.
# Redis commander (GUI for redis) -> http://127.0.0.1:40009
# Local host docs -> http://127.0.0.1:40010
# Jupyter lab with maro -> http://127.0.0.1:40011
docker run -p 40009:40009 -p 40010:40010 -p 40011:40011 maro/playground:cpu
```

3.2.3 Major Services in Playground

Service	Description	Host
`Redis Commander`	Redis web GUI.	http://127.0.0.1:40009
`Read the Docs`	Local host docs.	http://127.0.0.1:40010
`Jupyter Lab`	Jupyter lab with MARO environment, examples, notebooks.	http://127.0.0.1:40011

(If you use other port mapping, remember to change the port number.)

3.2.4 Major Materials in Root Folder

Folder	Description
`examples`	Showcases of predefined scenarios.
`notebooks`	Quick-start tutorial.

(Those not mentioned in the table can be ignored.)

3.3 Grass Cluster Provisioning on Azure

With the following guide, you can build up a MARO cluster in grass mode on Azure and run your training job in a distributed environment.

3.3.1 Prerequisites

- Install the Azure CLI and login
- Install docker and Configure docker to make sure it can be managed as a non-root user

3.3.2 Cluster Management

- Create a cluster with a *deployment*

```
# Create a grass cluster with a grass-create deployment
maro grass create ./grass-azure-create.yml
```

- Scale the cluster

```
# Scale nodes with 'Standard_D4s_v3' specification to 2
maro grass node scale my_grass_cluster Standard_D4s_v3 2
```

  Check VM Size to see more node specifications.

- Delete the cluster

```
# Delete a grass cluster
maro grass delete my_grass_cluster
```

- Start/stop nodes to save costs

```
# Start 2 nodes with 'Standard_D4s_v3' specification
maro grass node start my_grass_cluster Standard_D4s_v3 2

# Stop 2 nodes with 'Standard_D4s_v3' specification
maro grass node stop my_grass_cluster Standard_D4s_v3 2
```

3.3.3 Run Job

- Push your training image

```
# Push image 'my_image' to the cluster
maro grass image push my_grass_cluster --image-name my_image
```

- Push your training data

```
# Push data under './my_training_data' to a relative path '/my_training_data' in␣
↪the cluster
# You can then assign your mapping location in the start-job deployment
maro grass data push my_grass_cluster ./my_training_data/* /my_training_data
```

- Start a training job with a *deployment*

```
# Start a training job with a start-job deployment
maro grass job start my_grass_cluster ./grass-start-job.yml
```

- Or, schedule batch jobs with a *deployment*

```
# Start a training schedule with a start-schedule deployment
maro grass schedule start my_grass_cluster ./grass-start-schedule.yml
```

- Get the logs of the job

```
# Get the logs of the job
maro grass job logs my_grass_cluster my_job_1
```

- List the current status of the job

```
# List the current status of the job
maro grass job list my_grass_cluster
```

- Stop a training job

```
# Stop a training job
maro grass job stop my_job_1
```

3.3.4 Sample Deployments

grass-azure-create

```yaml
mode: grass
name: my_grass_cluster

cloud:
  infra: azure
  location: eastus
  resource_group: my_grass_resource_group
  subscription: my_subscription

user:
  admin_public_key: "{ssh public key with 'ssh-rsa' prefix}"
  admin_username: admin

master:
  node_size: Standard_D2s_v3
```

grass-start-job

```yaml
mode: grass
name: my_job_1

allocation:
  mode: single-metric-balanced
  metric: cpu

components:
  actor:
    command: "bash {project root}/my_training_data/job_1/actor.sh"
    image: my_image
    mount:
      target: "{project root}"
    num: 5
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
  learner:
    command: "bash {project root}/my_training_data/job_1/learner.sh"
    image: my_image
    mount:
      target: "{project root}"
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
```

grass-start-schedule

```yaml
mode: grass
name: my_schedule_1

allocation:
  mode: single-metric-balanced
  metric: cpu

job_names:
  - my_job_2
  - my_job_3
  - my_job_4
  - my_job_5

components:
  actor:
    command: "bash {project root}/my_training_data/job_1/actor.sh"
    image: my_image
    mount:
      target: "{project root}"
    num: 5
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
  learner:
    command: "bash {project root}/my_training_data/job_1/learner.sh"
    image: my_image
    mount:
      target: "{project root}"
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
```

3.4 K8S Cluster Provisioning on Azure

With the following guide, you can build up a MARO cluster in k8s mode on Azure and run your training job in a distributed environment.

3.4.1 Prerequisites

- Install the Azure CLI and login
- Install and set up kubectl
- Install docker and Configure docker to make sure it can be managed as a non-root user
- Download AzCopy, then move the AzCopy executable to /bin folder or add the directory location of the AzCopy executable to your system path:

```
# Take AzCopy version 10.6.0 as an example

# Linux
tar xvf ./azcopy_linux_amd64_10.6.0.tar.gz; cp ./azcopy_linux_amd64_10.6.0/azcopy /
↪usr/local/bin

# MacOS (may required MacOS Security & Privacy setting)
unzip ./azcopy_darwin_amd64_10.6.0.zip; cp ./azcopy_darwin_amd64_10.6.0/azcopy /usr/
↪local/bin

# Windows
# 1. Unzip ./azcopy_windows_amd64_10.6.0.zip
# 2. Add the path of ./azcopy_windows_amd64_10.6.0 folder to your Environment␣
↪Variables
# Ref: https://superuser.com/questions/949560/how-do-i-set-system-environment-
↪variables-in-windows-10
```

3.4.2 Cluster Management

- Create a cluster with a *deployment*

```
# Create a k8s cluster
maro k8s create ./k8s-azure-create.yml
```

- Scale the cluster

```
# Scale nodes with 'Standard_D4s_v3' specification to 2
maro k8s node scale my_k8s_cluster Standard_D4s_v3 2
```

  Check VM Size to see more node specifications.

- Delete the cluster

```
# Delete a k8s cluster
maro k8s delete my_k8s_cluster
```

3.4.3 Run Job

- Push your training image

```
# Push image 'my_image' to the cluster
maro k8s image push my_k8s_cluster --image-name my_image
```

- Push your training data

```
# Push data under './my_training_data' to a relative path '/my_training_data' in␣
↪the cluster
# You can then assign your mapping location in the start-job deployment
maro k8s data push my_k8s_cluster ./my_training_data/* /my_training_data
```

- Start a training job with a *deployment*

```
# Start a training job with a start-job deployment
maro k8s job start my_k8s_cluster ./k8s-start-job.yml
```

- Or, schedule batch jobs with a *deployment*

```
# Start a training schedule with a start-schedule deployment
maro k8s schedule start my_k8s123_cluster ./k8s-start-schedule.yml
```

- Get the logs of the job

```
# Logs will be exported to current directory
maro k8s job logs my_k8s_cluster my_job_1
```

- List the current status of the job

```
# List current status of jobs
maro k8s job list my_k8s_cluster my_job_1
```

- Stop a training job

```
# Stop a training job
maro k8s job stop my_k8s_cluster my_job_1
```

3.4.4 Sample Deployments

k8s-azure-create

```
mode: k8s
name: my_k8s_cluster

cloud:
  infra: azure
  location: eastus
  resource_group: my_k8s_resource_group
  subscription: my_subscription

user:
  admin_public_key: "{ssh public key with 'ssh-rsa' prefix}"
  admin_username: admin

master:
  node_size: Standard_D2s_v3
```

k8s-start-job

```
mode: k8s
name: my_job_1

components:
  actor:
    command: ["bash", "{project root}/my_training_data/actor.sh"]
    image: my_image
    mount:
      target: "{project root}"
    num: 5
    resources:
```

(continues on next page)

(continued from previous page)

```
      cpu: 2
      gpu: 0
      memory: 2048m
  learner:
    command: ["bash", "{project root}/my_training_data/learner.sh"]
    image: my_image
    mount:
      target: "{project root}"
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
```

k8s-start-schedule

```
mode: k8s
name: my_schedule_1

job_names:
  - my_job_2
  - my_job_3
  - my_job_4
  - my_job_5

components:
  actor:
    command: ["bash", "{project root}/my_training_data/actor.sh"]
    image: my_image
    mount:
      target: "{project root}"
    num: 5
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
  learner:
    command: ["bash", "{project root}/my_training_data/learner.sh"]
    image: my_image
    mount:
      target: "{project root}"
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
```

3.5 Container Inventory Management (CIM)

The Container Inventory Management (CIM) scenario simulates a common problem of container shipping in marine transportation. Imagine an international market: The goods are packed in containers and shipped by vessels from the exporting country to the importing country. As a result of the imbalanced global trade, the volume of available containers in different ports may not match their needs. In other words, some ports will have excess containers while some ports may be in short. Therefore, We can use the excess capacity on vessels to reposition empty containers to alleviate this imbalance.

3.5.1 Resource Flow

In this scenario, the **container** is the central resource. Two events will trigger the movement of the container:

- The first one is the order, which will lead to the transportation of goods from the source port to the destination port.
- The second one is the repositioning operation. It is used to rebalance the container distribution worldwide.

Order

To simulate a real market, we assume that there will be a certain number of orders from some ports to other ports every day. And the total order number of each day is generated according to a predefined distribution. These orders are then allocated to each export port in a relatively fixed proportion, and each export port will have a relatively fixed number of import ports as customers. The order distribution and the proportion of order allocation are specified in the topology and can be customized based on different requirements.

An order will trigger a life cycle of a container, as shown in the figure above, a life cycle is defined as follows:

- Once an order is generated, an empty container of the corresponding export port (source port) will be released to the shipper.
- The shipper will fill the container with cargo which turns it into a laden and then return it to the port after a few days.
- Loading occurs when the vessel arrives at this port.
- After several days of sailing, the vessel will finally arrive at the corresponding import port (destination port) where the discharging of the laden happens.
- Then the laden will be released to the consignee, and the consignee will take out the cargo in it, which turns it into an empty container again.
- Finally, the consignee returns it as an available container for the import port in a few days.

Container Repositioning

As mentioned above, to rebalance the container distribution, the agent in each port will decide how to reposition the empty containers every time a vessel arrives at the port. The decision consists of two parts:

- Whether to take a `discharge` operation or a `load` operation;
- The number of containers to discharge/load.

The export-oriented ports (e.g. the ports in China) show a clearly high demand feature, and usually require additional supply of empty containers. These ports will tend to discharge empty containers from the vessel if feasible. While the import-oriented ports have a significant surplus feature, that usually receive many empty container from the consignee. So the imported-oriented ports will tend to load the surplus empty containers into the vessel if there is free capacity.

The specific quantity to operate for a `discharge` action is limited by the remaining space in the port and the total number of empty containers in the vessel. Similarly, a `load` action is limited by the remaining space in the vessel and the total number of empty containers in the port. Of course, a good decision will not only consider the self future supply and demand situation, but also the needs and situation of the upstream and downstream ports.

3.5.2 Topologies

To provide an exploration road map from easy to difficult, two kinds of topologies are designed and provided in CIM scenario. Toy topologies provide simplified environments for algorithm debugging and will show some typical relationships between ports to users. We hope these will provide users with some insights to know more and deeper about this scenario. While the global topologies are based on the real-world data, and are bigger and more complicated to present the real problem.

Toy Topologies

(In these topologies, the solid lines indicate the service route (voyage) among ports, while the dashed lines indicate the container flow triggered by orders.)

toy.4p_ssdd_l0.D: There are four ports in this topology. According to the orders, D1 and D2 are simple demanders (the port requiring additional empty containers) while S2 is a simple supplier (the port with surplus empty containers). Although S1 is a simple destination port, it's at the intersection of two service routes, which makes it a special port in this topology. To achieve the global optimum, S1 must learn to distinguish the service routes and take service route specific repositioning operations.

toy.5p_ssddd_l0.D: There are five ports in this topology. According to the orders, D1 and D2 are simple demanders while S1 and S2 are simple suppliers. As a port in the intersection of service routes, although the supply and demand of port T1 can reach a state of self-balancing, it still plays an important role for the global optimum. The best repositioning policy for port T1 is to transfer the surplus empty containers from the left service route to the right one. Also, the port D1 and D2 should learn to discharge only the quantity they need and leave the surplus ones to other ports.

toy.6p_sssbdd_l0.D: There are six ports in this topology. Similar to toy.5p_ssddd, in this topology, there are simple demanders D1 and D2, simple suppliers S1 and S2, and self-balancing ports T1 and T2. More difficult than in toy.5p_ssddd, more transfers should be taken to reposition the surplus empty containers from the left most service route to the right most one, which means a multi-steps solution that involving more ports is required.

Global Topologies

global_trade.22p_l0.D: This is a topology designed based on the real-world data. The order generation model in this topology is built based on the trade data from WTO. According to the query results in WTO from January 2019 to October 2019, The ports with large trade volume are selected, and the proportion of each port as the source of orders is directly proportional to the export volume of the country it belongs to, while the proportion as the destination is proportional to the import volume. In this scenario, there are much more ports, much more service routes. And most ports no longer have a simple supply/demand feature. The cooperation among ports is much more complex and it is difficult to find an efficient repositioning policy manually.

(To make it clearer, the figure above only shows the service routes among ports.)

Naive Baseline

Below are the final environment metrics of the method *no repositioning* and *random repositioning* in different topologies. For each experiment, we setup the environment and test for a duration of 1120 ticks (days).

No Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.4p_ssdd_l0.0	2,240,000	2,190,000	0
toy.4p_ssdd_l0.1	2,240,000	2,190,000	0
toy.4p_ssdd_l0.2	2,240,000	2,190,000	0
toy.4p_ssdd_l0.3	2,239,460	2,189,460	0
toy.4p_ssdd_l0.4	2,244,068	2,194,068	0
toy.4p_ssdd_l0.5	2,244,068	2,194,068	0
toy.4p_ssdd_l0.6	2,244,068	2,194,068	0
toy.4p_ssdd_l0.7	2,244,068	2,194,068	0
toy.4p_ssdd_l0.8	2,241,716	2,191,716	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.5p_ssddd_l0.0	2,240,000	2,140,000	0
toy.5p_ssddd_l0.1	2,240,000	2,140,000	0
toy.5p_ssddd_l0.2	2,240,000	2,140,000	0
toy.5p_ssddd_l0.3	2,239,460	2,139,460	0
toy.5p_ssddd_l0.4	2,244,068	2,144,068	0
toy.5p_ssddd_l0.5	2,244,068	2,144,068	0
toy.5p_ssddd_l0.6	2,244,068	2,144,068	0
toy.5p_ssddd_l0.7	2,244,068	2,144,068	0
toy.5p_ssddd_l0.8	2,241,716	2,141,716	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.6p_sssbdd_l0.0	2,240,000	2,087,000	0
toy.6p_sssbdd_l0.1	2,240,000	2,087,000	0
toy.6p_sssbdd_l0.2	2,240,000	2,087,000	0
toy.6p_sssbdd_l0.3	2,239,460	2,086,460	0
toy.6p_sssbdd_l0.4	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.5	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.6	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.7	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.8	2,241,716	2,088,716	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
global_trade.22p_l0.0	2,240,000	1,028,481	0
global_trade.22p_l0.1	2,240,000	1,081,935	0
global_trade.22p_l0.2	2,240,000	1,083,358	0
global_trade.22p_l0.3	2,239,460	1,085,212	0
global_trade.22p_l0.4	2,244,068	1,089,628	0
global_trade.22p_l0.5	2,244,068	1,102,913	0
global_trade.22p_l0.6	2,244,068	1,122,092	0
global_trade.22p_l0.7	2,244,068	1,162,108	0
global_trade.22p_l0.8	2,241,716	1,161,714	0

Random Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.4p_ssdd_l0.0	2,240,000	1,497,138 ± 30,423	4,185,080 ± 185,140
toy.4p_ssdd_l0.1	2,240,000	1,623,710 ± 36,421	2,018,360 ± 36,700
toy.4p_ssdd_l0.2	2,240,000	1,501,466 ± 48,566	2,145,180 ± 90,300
toy.4p_ssdd_l0.3	2,239,460	1,577,011 ± 35,109	2,098,500 ± 35,120
toy.4p_ssdd_l0.4	2,244,068	1,501,835 ± 103,196	2,180,480 ± 33,020
toy.4p_ssdd_l0.5	2,244,068	1,546,227 ± 81,107	2,077,320 ± 113,740
toy.4p_ssdd_l0.6	2,244,068	1,578,863 ± 127,815	2,220,720 ± 106,660
toy.4p_ssdd_l0.7	2,244,068	1,519,495 ± 60,555	2,441,480 ± 79,700
toy.4p_ssdd_l0.8	2,241,716	1,603,063 ± 109,149	2,518,920 ± 193,200

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.5p_ssddd_l0.0	2,240,000	1,371,021 ± 34,619	3,966,120 ± 138,960
toy.5p_ssddd_l0.1	2,240,000	1,720,068 ± 18,939	1,550,280 ± 25,600
toy.5p_ssddd_l0.2	2,240,000	1,716,435 ± 15,499	1,496,860 ± 31,260
toy.5p_ssddd_l0.3	2,239,460	1,700,456 ± 26,510	1,586,640 ± 11,500
toy.5p_ssddd_l0.4	2,244,068	1,663,139 ± 34,244	1,594,160 ± 103,040
toy.5p_ssddd_l0.5	2,244,068	1,681,519 ± 107,863	1,635,360 ± 61,880
toy.5p_ssddd_l0.6	2,244,068	1,660,330 ± 38,318	1,630,060 ± 81,580
toy.5p_ssddd_l0.7	2,244,068	1,709,022 ± 31,440	1,854,340 ± 167,080
toy.5p_ssddd_l0.8	2,241,716	1,763,950 ± 73,935	1,858,420 ± 60,680

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.6p_sssbdd_l0.0	2,240,000	1,529,774 ± 73,104	3,989,560 ± 232,740
toy.6p_sssbdd_l0.1	2,240,000	1,736,385 ± 16,736	1,122,120 ± 28,960
toy.6p_sssbdd_l0.2	2,240,000	1,765,945 ± 4,680	1,052,520 ± 44,020
toy.6p_sssbdd_l0.3	2,239,460	1,811,987 ± 15,436	998,740 ± 69,680
toy.6p_sssbdd_l0.4	2,244,068	1,783,362 ± 39,122	1,059,860 ± 49,100
toy.6p_sssbdd_l0.5	2,244,068	1,755,551 ± 44,855	1,101,100 ± 55,180
toy.6p_sssbdd_l0.6	2,244,068	1,830,504 ± 10,690	1,141,660 ± 10,520
toy.6p_sssbdd_l0.7	2,244,068	1,742,129 ± 23,910	1,311,420 ± 64,560
toy.6p_sssbdd_l0.8	2,241,716	1,761,283 ± 22,338	1,336,540 ± 30,020

Topology	Total Requirement	Resource Shortage	Repositioning Number
global_trade.22p_l0.0	2,240,000	1,010,009 ± 20,942	548,240 ± 14,600
global_trade.22p_l0.1	2,240,000	1,027,395 ± 19,183	188,160 ± 12,940
global_trade.22p_l0.2	2,240,000	1,035,851 ± 4,352	181,240 ± 5,240
global_trade.22p_l0.3	2,239,460	1,032,480 ± 1,332	190,220 ± 8,920
global_trade.22p_l0.4	2,244,068	1,034,412 ± 11,689	186,080 ± 6,280
global_trade.22p_l0.5	2,244,068	1,042,869 ± 16,146	188,720 ± 7,880
global_trade.22p_l0.6	2,244,068	1,096,502 ± 26,896	302,280 ± 27,540
global_trade.22p_l0.7	2,244,068	1,144,981 ± 5,355	283,520 ± 25,700
global_trade.22p_l0.8	2,241,716	1,154,184 ± 7,043	270,960 ± 2,240

3.5.3 Quick Start

Data Preparation

To start a simulation in CIM scenario, no extra data processing is needed. You can just specify the scenario and the topology when initialize an environment and enjoy your exploration in this scenario.

Environment Interface

Before starting interaction with the environment, we need to know the definition of `DecisionEvent` and `Action` in CIM scenario first. Besides, you can query the environment snapshot list to get more detailed information for the decision making.

DecisionEvent

Once the environment need the agent's response to promote the simulation, it will throw an `DecisionEvent`. In the scenario of CIM, the information of each `DecisionEvent` is listed as below:

- **tick** (int): The corresponding tick.
- **port_idx** (int): The id of the port/agent that needs to respond to the environment.
- **vessel_idx** (int): The id of the vessel/operation object of the port/agent.
- **action_scope** (ActionScope): ActionScope has two attributes:
  - `load` indicates the maximum quantity that can be loaded from the port the vessel.
  - `discharge` indicates the maximum quantity that can be discharged from the vessel to the port.
- **early_discharge** (int): When the available capacity in the vessel is not enough to load the ladens, some of the empty containers in the vessel will be early discharged to free the space. The quantity of empty containers that have been early discharged due to the laden loading is recorded in this field.

Action

Once we get a `DecisionEvent` from the environment, we should respond with an `Action`. Valid `Action` could be:

- `None`, which means do nothing.
- A valid `Action` instance, including:
  - **vessel_idx** (int): The id of the vessel/operation object of the port/agent.
  - **port_idx** (int): The id of the port/agent that take this action.
  - **quantity** (int): The sign of this value denotes different meanings:
    - Positive quantity means discharging empty containers from vessel to port.
    - Negative quantity means loading empty containers from port to vessel.

Example

Here we will show you a simple example of interaction with the environment in random mode, we hope this could help you learn how to use the environment interfaces:

```
from maro.simulator import Env
from maro.simulator.scenarios.cim.common import Action, DecisionEvent

import random

# Initialize an environment of CIM scenario, with a specific topology.
# In Container Inventory Management, 1 tick means 1 day, durations=100 here indicates␣
↪a length of 100 days.
```

(continues on next page)

(continued from previous page)

```
env = Env(scenario="cim", topology="toy.5p_ssddd_l0.0", start_tick=0, durations=100)

# Query for the environment summary, the business instances and intra-instance␣
↪attributes
# will be listed in the output for your reference.
print(env.summary)

metrics: object = None
decision_event: DecisionEvent = None
is_done: bool = False
action: Action = None

num_episode = 2
for ep in range(num_episode):
    # Gym-like step function.
    metrics, decision_event, is_done = env.step(None)

    while not is_done:
        past_week_ticks = [
            x for x in range(decision_event.tick - 7, decision_event.tick)
        ]
        decision_port_idx = decision_event.port_idx
        intr_port_infos = ["booking", "empty", "shortage"]

        # Query the snapshot list of this environment to get the information of
        # the booking, empty, shortage of the decision port in the past week.
        past_week_info = env.snapshot_list["ports"][
            past_week_ticks : decision_port_idx : intr_port_infos
        ]

        # Generate a random Action according to the action_scope in DecisionEvent.
        random_quantity = random.randint(
            -decision_event.action_scope.load,
            decision_event.action_scope.discharge
        )
        action = Action(
            vessel_idx=decision_event.vessel_idx,
            port_idx=decision_event.port_idx,
            quantity=random_quantity
        )

        # Drive the environment with the random action.
        metrics, decision_event, is_done = env.step(action)

    # Query for the environment business metrics at the end of each episode,
    # it is usually users' optimized object in CIM scenario (usually includes multi-
↪target).
    print(f"ep: {ep}, environment metrics: {env.metrics}")
    env.reset()
```

Jump to this notebook for a quick experience.

3.6 Bike Repositioning (Citi Bike)

The Citi Bike scenario simulates the bike repositioning problem triggered by the one-way bike trips based on the public trip data from Citi Bike.

> Citi Bike is New York City's bike share system, which consists of a fleet of bikes that are locked into a network of docking stations throughout the city. The bikes can be unlocked from one station and returned to any other station in the system, making them ideal for one-way trips. People use bike share to commute to work or school, run errands, get to appointments or social engagements, and more.

Since the demand for bikes and empty docks is dynamically changed during a day, and the bike flow between two stations are not equal in a same period, some stations suffer from severe bike shortages, while some have too much bikes and too few empty docks. In such a situation, the bike repositioning is essential to balance the bike's supply and demand. A good bike repositioning can not only meet the needs in the stations with heavy ride demand but also free the stations that do not have enough empty docks. Also, in the long run, a good bike repositioning can improve the bike useability, empower citizens' daily life, and reduce the carbon emission.

3.6.1 Resource Flow

In this scenario, the **bike** is the central resource. Two events will trigger the movement of the bike:

- The first one is the trip requirement, which may cause the bike transfer from the source station to the destination station;
- The second one is the repositioning operation. It is used to rebalance the bike distribution among stations.

Bike Trip

In the citi bike scenario in MARO, the trip generation and the corresponding bike flow is defined as follows:

- Given a fixed time interval, for each specific source-destination station pair, a trip requirement will arise according to a predefined distribution or the real trip data. It depends on the chosen topology.
- If there are enough available bikes in the source station of the trip requirement, the required bike(s) will be unlocked and assigned to this trip. Otherwise, a shortage will be recorded in the source station.
- The trip duration is read from the trip log if real trip data is used. Otherwise, the duration will be sampled from a specific random distribution.
- At the end of the trip, the bike will be returned to the destination station. But if the destination does not have enough available docks, the bike will be returned to the nearest station with available docks.

Bike Repositioning

As for the repositioning operation, the simulator in MARO will regularly check the remaining bikes in each station and compare it with a predefined low watermark and high watermark. If the bike inventory is lower than the low watermark, the station will generate a `Demand` event to request the supply of bikes from its neighboring stations. Similarly, if the bike inventory is higher than the high watermark, the station will generate a `Supply` event to transport excess bikes to its neighboring stations. The low watermark and the high watermark is specified in the topology and can be customized based on different requirements.

The target station candidates of the `Supply` and `Demand` events are selected by a predefined multi-layer filter in this scenario:

1. The distance between the caller station and the neighboring stations will be used to filter and get a specific number of stations;

2. The number of available bikes at each candidate station will be used to further filter on the candidate stations. For a `Supply` event, the stations with less bikes will be kept, while for a `Demand` event, the stations with more bikes will be kept;
3. The future trip requirement of the target station will be the last filter. For a `Supply` event, the stations with more future trip requirement will be left in the final station candidate set, while the stations with less future trip requirement will be left for `Demand` event.

The size of the candidate sets in each filter level is specified in the topology and can be customized based on different requirements.

Once the target station candidate is filtered, the `action scope` for each candidate will also be calculated in the simulator and return to the decision agent together with some other information in the *DecisionEvent*. For a `Supply` event, the bike inventory of the caller station and the number of available docks of the target station candidates will be attached. On the contrary, for a `Demand` event, the number of available docks of the source station and the bike inventory of the target station candidates will be attached.

Based on the given target station candidates and the corresponding `action scope`, the decision agent of the caller station should decide how many bikes to transfer to/request from the target station. We call a pair of `(target station, bike number)` a repositioning action. After an action taken, the destination station should wait for a certain period to get the bikes available for trip requirement. The action lead time is sampled from a predefined distribution.

3.6.2 Topologies

To provide an exploration road map from easy to difficult, two kinds of topologies are designed and provided in Citi Bike scenario. Toy topologies provide a super simplified environment for algorithm debugging, while the real topologies with real data from Citi Bike historical trips can present the real problem to users.

Toy Topologies

In toy topology, the generation of the trip requirements follows a stable pattern as introduced above. The detailed trip demand pattern are listed as below. And we hope that these toy topologies can provide you with some insights about this scenario.

toy.3s_4t: There are three stations in this topology. Every two minutes, there will be a trip requirement from S2 to S3 and a trip requirement from S3 to S2. At the same time, every two minutes, the system will generate trip requirement from S1 to S3 and from S1 to S2 with a fixed probability (80% and 20%, respectively). In this topology, the traffic flow between S2 and S3 is always equal, but station S1 is a super consumer with only bikes flow out. So the best repositioning policy in this topology is to reposition bikes from S2 and S3 to S1. It requires the active request action of S1 or the proactive transfer action of S2 and S3.

toy.4s_4t: There are four stations in this topology. According to the global trip demand, there are more returned bikes than leaving bikes in station S1 and S3, while more leaving bikes than returned bikes in station S2 and S4. So the best repositioning policy in this topology is to reposition the excess bikes from S1 and S3 to S2 and S4. Furthermore, the cooperation between these stations is also necessary since only a proper allocation can lead to a globally optimal solution.

toy.5s_6t: There are five stations in this topology. Although trip demand is more complex than the other two topologies above, we can still find that station S1 is a self-balancing station, station S2 and S5 have more returned bikes, and station S3 and S4 have more leaving bikes. Just like in topology toy.4s_4t, the best repositioning policy is to reposition excess bikes from S2 and S5 to S3 and S4 coordinately.

Real Topologies

ny.YYYYMM: Different from the stable generation model in the toy topologies, the trip requirement in the topology ny.YYYYMM is generated based on the real trip data from Citi Bike, which includes the source station, the destination station, and the duration of each trip. Besides, the total number of available bikes in this kind of topologies is counted from the real trip data of the specific month. Weighted by the the latest capacity of each stations, the available bikes are allocated to each station, which constitutes the initial bike inventory of each station. In this series of topologies, the definition of the bike flow and the trigger mechanism of repositioning actions are the same as those in the toy topologies. We provide this series of topologies to better simulate the actual Citi Bike scenario.

Naive Baseline

Below are the final environment metrics of the method *no repositioning* and *random repositioning* in different topologies. For each experiment, we setup the environment and test for a duration of 1 week.

No Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.3s_4t	15,118	8,233	0
toy.4s_4t	9,976	7,048	0
toy.5s_6t	16,341	9,231	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201801	48,089	2,688	0
ny.201802	126,374	8,814	0
ny.201803	138,952	10,942	0
ny.201804	161,443	10,349	0
ny.201805	323,375	29,081	0
ny.201806	305,971	26,412	0
ny.201807	254,715	19,669	0
ny.201808	302,589	26,352	0
ny.201809	313,002	28,472	0
ny.201810	339,268	24,109	0
ny.201811	263,227	21,485	0
ny.201812	209,102	15,876	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201901	161,474	10,775	0
ny.201902	187,354	12,593	0
ny.201903	148,371	7,193	0
ny.201904	280,852	16,906	0
ny.201905	287,290	27,213	0
ny.201906	379,415	33,968	0
ny.201907	309,365	21,105	0
ny.201908	371,969	33,703	0
ny.201909	344,847	24,528	0
ny.201910	351,855	29,544	0
ny.201911	324,327	29,489	0
ny.201912	184,015	14,205	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.202001	169,304	12,449	0
ny.202002	206,105	14,794	0
ny.202003	235,986	15,436	0
ny.202004	91,810	2,348	0
ny.202005	169,412	5,231	0
ny.202006	197,883	7,608	0

Random Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.3s_4t	15,154	8,422 $\pm$ 11	449 $\pm$ 22
toy.4s_4t	10,186	4,371 $\pm$ 72	3,392 $\pm$ 83
toy.5s_6t	16,171	7,513 $\pm$ 40	3,242 $\pm$ 71

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201801	48,089	6,693 ± 138	445,996 ± 6,756
ny.201802	126,374	21,418 ± 120	446,564 ± 3,505
ny.201803	138,952	22,121 ± 272	448,259 ± 1,831
ny.201804	161,443	22,201 ± 194	453,705 ± 3,697
ny.201805	323,375	54,365 ± 538	470,771 ± 5,337
ny.201806	305,971	49,876 ± 1,091	481,443 ± 6,981
ny.201807	254,715	46,199 ± 204	483,788 ± 982
ny.201808	302,589	53,679 ± 433	485,137 ± 2,557
ny.201809	313,002	61,432 ± 75	474,851 ± 2,908
ny.201810	339,268	64,269 ± 600	461,928 ± 1,018
ny.201811	263,227	40,440 ± 239	467,050 ± 6,595
ny.201812	209,102	26,067 ± 234	457,173 ± 6,444

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201901	161,474	19,295 ± 155	444,445 ± 2,287
ny.201902	187,354	23,875 ± 282	456,888 ± 362
ny.201903	148,371	12,451 ± 312	409,226 ± 5,392
ny.201904	280,852	29,591 ± 170	464,671 ± 6,148
ny.201905	287,290	44,199 ± 542	485,077 ± 6,140
ny.201906	379,415	51,396 ± 256	503,503 ± 4,742
ny.201907	309,365	33,861 ± 643	500,443 ± 4,314
ny.201908	371,969	51,319 ± 417	516,684 ± 1,400
ny.201909	344,847	34,532 ± 466	476,965 ± 3,932
ny.201910	351,855	37,828 ± 502	496,135 ± 4,167
ny.201911	324,327	34,745 ± 427	484,599 ± 8,771
ny.201912	184,015	20,119 ± 110	437,311 ± 5,936

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.202001	169,304	17,152 ± 218	476,821 ± 1,052
ny.202002	206,105	24,223 ± 209	480,012 ± 1,547
ny.202003	235,986	23,749 ± 654	458,536 ± 1,457
ny.202004	91,810	3,349 ± 48	326,817 ± 3.131
ny.202005	169,412	10,177 ± 216	378,038 ± 2,429
ny.202006	197,883	11,741 ± 170	349,932 ± 4,375

3.6.3 Quick Start

Data Preparation

To start the simulation of Citi Bike scenario, users have two options for the data preparation:

- If the topology data is not generated in advance, the system will automatically download and process the relevant data when the environment is created. The data will be stored in a temporary folder and be automatically deleted after the experiment.
- Before creating the environment, users can also manually download and generate relevant data. This approach will save you a lot of time if you need to conduct several experiments on the same topology. Therefore, we encourage you to generate the relevant data manually first.

The following is the introduction to related commands:

Environment List Command

The data environment `list` command is used to list the environments that need the data files generated before the simulation.

```
maro env data list

scenario: citi_bike, topology: ny.201801
scenario: citi_bike, topology: ny.201802
scenario: citi_bike, topology: ny.201803
scenario: citi_bike, topology: ny.201804
scenario: citi_bike, topology: ny.201805
scenario: citi_bike, topology: ny.201806
...
```

Generate Command

The data `generate` command is used to automatically download and build the specified predefined scenario and topology data files for the simulation. Currently, there are three arguments for the data `generate` command:

- `-s`: required, used to specify the predefined scenario. Valid scenarios are listed in the result of *environment list command*.
- `-t`: required, used to specify the predefined topology. Valid topologies are listed in the result of *environment list command*.
- `-f`: optional, if set, to force to re-download and re-generate the data files and overwrite the already existing ones.

```
maro env data generate -s citi_bike -t ny.201802

The data files for citi_bike-ny201802 will then be downloaded and deployed to ~/.maro/
↪data/citibike/_build/ny201802.
```

For the example above, the directory structure should be like:

```
|-- ~/.maro
    |-- data
    |   |-- citi_bike
```

(continues on next page)

(continued from previous page)

```
    |       |-- .build          # bin data file
    |           |-- [topology]  # topology
    |       |-- .source
    |           |-- .download   # original data file
    |           |-- .clean      # cleaned data file
    |-- temp                    # download temp file
```

Build Command

The data `build` command is used to build the CSV data files to binary data files that the simulator needs. Currently, there are three arguments for the data `build` command:

- `--meta`: required, used to specify the path of the meta file. The source columns that to be converted and the data type of each columns should be specified in the meta file.
- `--file`: required, used to specify the path of the source CSV data file(s). If multiple source CSV data files are needed, you can list all the full paths of the source files in a specific file and use a `@` symbol to specify it.
- `--output`: required, used to specify the path of the target binary file.

```
maro data build --meta ~/.maro/data/citibike/meta/trips.yml --file ~/.maro/data/
↪citibike/source/_clean/ny201801/trip.csv --output ~/.maro/data/citibike/_build/
↪ny201801/trip.bin
```

Environment Interface

Before starting interaction with the environment, we need to know the definition of `DecisionEvent` and `Action` in Citi Bike scenario first. Besides, you can query the environment snapshot list to get more detailed information for the decision making.

DecisionEvent

Once the environment need the agent's response to reposition bikes, it will throw an `DecisionEvent`. In the scenario of Citi Bike, the information of each `DecisionEvent` is listed as below:

- **station_idx** (int): The id of the station/agent that needs to respond to the environment.
- **tick** (int): The corresponding tick.
- **frame_index** (int): The corresponding frame index, that is the index of the corresponding snapshot in the environment snapshot list.
- **type** (DecisionType): The decision type of this decision event. In Citi Bike scenario, there are 2 types:
  - `Supply` indicates there is too many bikes in the corresponding station, so it is better to reposition some of them to other stations.
  - `Demand` indicates there is no enough bikes in the corresponding station, so it is better to reposition bikes from other stations.
- **action_scope** (dict): A dictionary that maintains the information for calculating the valid action scope:
  - The key of these item indicate the station/agent ids.
  - The meaning of the value differs for different decision type:

* If the decision type is `Supply`, the value of the station itself means its bike inventory at that moment, while the value of other target stations means the number of their empty docks.
* If the decision type is `Demand`, the value of the station itself means the number of its empty docks, while the value of other target stations means their bike inventory.

Action

Once we get a `DecisionEvent` from the environment, we should respond with an `Action`. Valid `Action` could be:

- `None`, which means do nothing.
- A valid `Action` instance, including:
  - **from_station_idx** (int): The id of the source station of the bike transportation.
  - **to_station_idx** (int): The id of the destination station of the bike transportation.
  - **number** (int): The quantity of the bike transportation.

Example

Here we will show you a simple example of interaction with the environment in random mode, we hope this could help you learn how to use the environment interfaces:

```
from maro.simulator import Env
from maro.simulator.scenarios.citi_bike.common import Action, DecisionEvent,␣
↪DecisionType

import random

# Initialize an environment of Citi Bike scenario, with a specific topology.
# In CitiBike, 1 tick means 1 minute, durations=1440 here indicates a length of 1 day.
# In CitiBike, one snapshot will be maintained every snapshot_resolution ticks,
# snapshot_resolution=30 here indicates 1 snapshot per 30 minutes.
env = Env(scenario="citi_bike", topology="toy.3s_4t", start_tick=0, durations=1440,␣
↪snapshot_resolution=30)

# Query for the environment summary, the business instances and intra-instance␣
↪attributes
# will be listed in the output for your reference.
print(env.summary)

metrics: object = None
decision_event: DecisionEvent = None
is_done: bool = False
action: Action = None

num_episode = 2
for ep in range(num_episode):
    # Gym-like step function.
    metrics, decision_event, is_done = env.step(None)

    while not is_done:
        past_2hour_frames = [
            x for x in range(decision_event.frame_index - 4, decision_event.frame_
↪index)
```

(continues on next page)

(continued from previous page)

```
        ]
        decision_station_idx = decision_event.station_idx
        intr_station_infos = ["trip_requirement", "bikes", "shortage"]

        # Query the snapshot list of this environment to get the information of
        # the trip requirements, bikes, shortage of the decision station in the past␣
↪2 hours.
        past_2hour_info = env.snapshot_list["stations"][
            past_2hour_frames : decision_station_idx : intr_station_infos
        ]

        if decision_event.type == DecisionType.Supply:
            # Supply: the value of the station itself means the bike inventory.
            self_bike_inventory = decision_event.action_scope[decision_event.station_
↪idx]
            # Supply: the value of other stations means the quantity of empty docks.
            target_idx_dock_tuple_list = [
                (k, v) for k, v in decision_event.action_scope.items() if k !=␣
↪decision_event.station_idx
            ]
            # Randomly choose a target station weighted by the quantity of empty␣
↪docks.
            target_idx, target_dock = random.choices(
                target_idx_dock_tuple_list,
                weights=[item[1] for item in target_idx_dock_tuple_list],
                k=1
            )[0]
            # Generate the corresponding random Action.
            action = Action(
                from_station_idx=decision_event.station_idx,
                to_station_idx=target_idx,
                number=random.randint(0, min(self_bike_inventory, target_dock))
            )

        elif decision_event.type == DecisionType.Demand:
            # Demand: the value of the station itself means the quantity of empty␣
↪docks.
            self_available_dock = decision_event.action_scope[decision_event.station_
↪idx]
            # Demand: the value of other stations means their bike inventory.
            target_idx_inventory_tuple_list = [
                (k, v) for k, v in decision_event.action_scope.items() if k !=␣
↪decision_event.station_idx
            ]
            # Randomly choose a target station weighted by the bike inventory.
            target_idx, target_inventory = random.choices(
                target_idx_inventory_tuple_list,
                weights=[item[1] for item in target_idx_inventory_tuple_list],
                k=1
            )[0]
            # Generate the corresponding random Action.
            action = Action(
                from_station_idx=target_idx,
                to_station_idx=decision_event.station_idx,
                number=random.randint(0, min(self_available_dock, target_inventory))
            )
```

(continues on next page)

(continued from previous page)

```
        else:
            action = None

        # Drive the environment with the random action.
        metrics, decision_event, is_done = env.step(action)

    # Query for the environment business metrics at the end of each episode,
    # it is usually users' optimized object (usually includes multi-target).
    print(f"ep: {ep}, environment metrics: {env.metrics}")
    env.reset()
```

Jump to this notebook for a quick experience.

3.7 Multi Agent DQN for CIM

This example demonstrates how to use MARO's reinforcement learning (RL) toolkit to solve the CIM problem. It is formalized as a multi-agent reinforcement learning problem, where each port acts as a decision agent. The agents take actions independently, e.g., loading containers to vessels or discharging containers from vessels.

3.7.1 State Shaper

State shaper converts the environment observation to the model input state which includes temporal and spatial information. For this scenario, the model input state includes:

- Temporal information, including the past week's information of ports and vessels, such as shortage on port and

remaining space on vessel. - Spatial information, including related downstream port features.

```
class CIMStateShaper(StateShaper):
    ...
    def __call__(self, decision_event, snapshot_list):
        tick, port_idx, vessel_idx = decision_event.tick, decision_event.port_idx,␣
↪decision_event.vessel_idx
        ticks = [tick - rt for rt in range(self._look_back - 1)]
        future_port_idx_list = snapshot_list["vessels"][tick : vessel_idx : 'future_
↪stop_list'].astype('int')
        port_features = snapshot_list["ports"][ticks : [port_idx] + list(future_port_
↪idx_list) : self._port_attributes]
        vessel_features = snapshot_list["vessels"][tick : vessel_idx : self._vessel_
↪attributes]
        state = np.concatenate((port_features, vessel_features))
        return str(port_idx), state
```

3.7.2 Action Shaper

Action shaper is used to convert an agent's model output to an environment executable action. For this specific scenario, the action space consists of integers from -10 to 10, with -10 indicating loading 100% of the containers in the current inventory to the vessel and 10 indicating discharging 100% of the containers on the vessel to the port.

```
class CIMActionShaper(ActionShaper):
    ...
    def __call__(self, model_action, decision_event, snapshot_list):
        scope = decision_event.action_scope
        tick = decision_event.tick
        port_idx = decision_event.port_idx
        vessel_idx = decision_event.vessel_idx

        port_empty = snapshot_list["ports"][tick: port_idx: ["empty", "full", "on_
↪shipper", "on_consignee"]][0]
        vessel_remaining_space = snapshot_list["vessels"][tick: vessel_idx: ["empty",
↪"full", "remaining_space"]][2]
        early_discharge = snapshot_list["vessels"][tick:vessel_idx: "early_discharge
↪"][0]
        assert 0 <= model_action < len(self._action_space)

        if model_action < self._zero_action_index:
            actual_action = max(round(self._action_space[model_action] * port_empty),␣
↪-vessel_remaining_space)
        elif model_action > self._zero_action_index:
            plan_action = self._action_space[model_action] * (scope.discharge + early_
↪discharge) - early_discharge
            actual_action = round(plan_action) if plan_action > 0 else round(self._
↪action_space[model_action] * scope.discharge)
        else:
            actual_action = 0

        return Action(vessel_idx, port_idx, actual_action)
```

3.7.3 Experience Shaper

Experience shaper is used to convert an episode trajectory to trainable experiences for RL agents. For this specific scenario, the reward is a linear combination of fulfillment and shortage in a limited time window.

```
class TruncatedExperienceShaper(ExperienceShaper):
    ...
    def __call__(self, trajectory, snapshot_list):
        experiences_by_agent = {}
        for i in range(len(trajectory) - 1):
            transition = trajectory[i]
            agent_id = transition["agent_id"]
            if agent_id not in experiences_by_agent:
                experiences_by_agent[agent_id] = defaultdict(list)
            experiences = experiences_by_agent[agent_id]
            experiences["state"].append(transition["state"])
            experiences["action"].append(transition["action"])
            experiences["reward"].append(self._compute_reward(transition["event"],␣
↪snapshot_list))
            experiences["next_state"].append(trajectory[i + 1]["state"])
```

(continues on next page)

(continued from previous page)

```
        return experiences_by_agent

    def _compute_reward(self, decision_event, snapshot_list):
        start_tick = decision_event.tick + 1
        end_tick = decision_event.tick + self._time_window
        ticks = list(range(start_tick, end_tick))

        # calculate tc reward
        future_fulfillment = snapshot_list["ports"][ticks::"fulfillment"]
        future_shortage = snapshot_list["ports"][ticks::"shortage"]
        decay_list = [self._time_decay_factor ** i for i in range(end_tick - start_
↪tick)
                      for _ in range(future_fulfillment.shape[0] // (end_tick - start_
↪tick))]

        tot_fulfillment = np.dot(future_fulfillment, decay_list)
        tot_shortage = np.dot(future_shortage, decay_list)

        return np.float(self._fulfillment_factor * tot_fulfillment - self._shortage_
↪factor * tot_shortage)
```

3.7.4 Agent

Agent is a combination of (RL) algorithm, experience pool, and a set of parameters that governs the training loop. For this scenario, the agent is the abstraction of a port. We choose DQN as our underlying learning algorithm with a TD-error-based sampling mechanism.

3.7.5 Agent Manager

Agent manager is an agent assembler and isolates the complexities of the environment and algorithm. For this scenario, It will load the DQN algorithm and an experience pool for each agent.

```
class DQNAgentManager(AbsAgentManager):
    def _assemble(self, agent_dict):
        set_seeds(config.agents.seed)
        num_actions = config.agents.algorithm.num_actions
        for agent_id in self._agent_id_list:
            eval_model = LearningModel(decision_layers=MLPDecisionLayers(name=f'
↪{agent_id}.policy',
                                                                         input_
↪dim=self._state_shaper.dim,
                                                                         output_
↪dim=num_actions,
                                                                         **config.
↪agents.algorithm.model)
                                       )

            algorithm = DQN(model_dict={"eval": eval_model},
                            optimizer_opt=(RMSprop, config.agents.algorithm.
↪optimizer),
                            loss_func_dict={"eval": smooth_l1_loss},
                            hyper_params=DQNHyperParams(**config.agents.algorithm.
↪hyper_parameters,
```

(continues on next page)

(continued from previous page)

```
                                                    num_actions=num_actions))

            experience_pool = ColumnBasedStore(**config.agents.experience_pool)
            agent_dict[agent_id] = CIMAgent(name=agent_id, algorithm=algorithm,␣
↪experience_pool=experience_pool,
                                            **config.agents.training_loop_parameters)
```

3.7.6 Main Loop with Actor and Learner (Single Process)

This single-process workflow of a learning policy's interaction with a MARO environment is comprised of: - Initializing an environment with specific scenario and topology parameters. - Defining scenario-specific components, e.g. shapers. - Creating an agent manager, which assembles underlying agents. - Creating an actor and a learner to start the training process in which the agent manager interacts with the environment for collecting experiences and updating policies.

3.7.7 Main Loop with Actor and Learner (Distributed/Multi-process)

We demonstrate a single-learner and multi-actor topology where the learner drives the program by telling remote actors to perform roll-out tasks and using the results they sent back to improve the policies. The workflow usually involves launching a learner process and an actor process separately. Because training occurs on the learner side and inference occurs on the actor side, we need to create appropriate agent managers on both sides.

On the actor side, the agent manager must be equipped with all shapers as well as an explorer. Thus, The code for creating an environment and an agent manager on the actor side is similar to that for the single-host version, except that it is necessary to set the AgentMode to AgentMode.INFERENCE. As in the single-process version, the environment and the agent manager are wrapped in a SimpleActor instance. To make the actor a distributed worker, we need to further wrap it in an ActorWorker instance. Finally, we launch the worker and it starts to listen to roll-out requests from the learner. The following code snippet shows the creation of an actor worker with a simple (local) actor wrapped inside.

```
agent_manager = DQNAgentManager(name="cim_remote_actor",
                                agent_id_list=agent_id_list,
                                mode=AgentMode.INFERENCE,
                                state_shaper=state_shaper,
                                action_shaper=action_shaper,
                                experience_shaper=experience_shaper,
                                explorer=explorer)
proxy_params = {"group_name": config.distributed.group_name,
                "expected_peers": config.distributed.actor.peer,
                "redis_address": (config.distributed.redis.host_name, config.
↪distributed.redis.port)
                }
actor_worker = ActorWorker(local_actor=SimpleActor(env=env, inference_agents=agent_
↪manager),
                           proxy_params=proxy_params)
actor_worker.launch()
```

On the learner side, an agent manager in AgentMode.TRAIN mode is required. However, it is not necessary to create shapers for an agent manager in AgentMode.TRAIN mode (although a state shaper is created in this example so that the model input dimension can be readily accessed). Instead of creating an actor, we create an actor proxy and wrap it inside the learner. This proxy serves as the communication interface for the learner and is responsible for sending roll-out requests to remote actor processes and receiving results. Calling the train method executes the usual training

loop except that the actual roll-out is performed remotely. The code snippet below shows the creation of a learner with an actor proxy wrapped inside.

```
agent_manager = DQNAgentManager(name="cim_remote_learner", agent_id_list=agent_id_
↪list, mode=AgentMode.TRAIN,
                                state_shaper=state_shaper, explorer=explorer)

proxy_params = {"group_name": config.distributed.group_name,
                "expected_peers": config.distributed.learner.peer,
                "redis_address": (config.distributed.redis.host_name, config.
↪distributed.redis.port)
                }
learner = SimpleLearner(trainable_agents=agent_manager,
                        actor=ActorProxy(proxy_params=proxy_params),
                        logger=Logger("distributed_cim_learner", auto_
↪timestamp=False))
learner.train(total_episodes=config.general.total_training_episodes)
```

Note: All related code snippets are supported in maro playground.

3.8 Greedy Policy for Citi Bike

In this example we demonstrate using a simple greedy policy for Citi Bike, which is a real-world bike repositioning scenario.

3.8.1 Greedy Policy

Our greedy policy is simple: if the event type is supply, the policy will make the current station send as many bikes as possible to one of k stations with the most empty docks. If the event type is demand, the policy will make the current station request as many bikes as possible from one of k stations with the most bikes. We use a heap data structure to find the top k supply/demand candidates from the action scope associated with each decision event.

```
class GreedyPolicy:
    ...
    def choose_action(self, decision_event: DecisionEvent):
        if decision_event.type == DecisionType.Supply:
            """
            Find k target stations with the most empty slots, randomly choose one of␣
↪them and send as many bikes to
            it as allowed by the action scope
            """
            top_k_demands = []
            for demand_candidate, available_docks in decision_event.action_scope.
↪items():
                if demand_candidate == decision_event.station_idx:
                    continue

                heapq.heappush(top_k_demands, (available_docks, demand_candidate))
                if len(top_k_demands) > self._demand_top_k:
                    heapq.heappop(top_k_demands)

            max_reposition, target_station_idx = random.choice(top_k_demands)
```

(continues on next page)

(continued from previous page)

```
            action = Action(decision_event.station_idx, target_station_idx, max_
↪reposition)
        else:
            """
            Find k source stations with the most bikes, randomly choose one of them␣
↪and request as many bikes from
            it as allowed by the action scope.
            """
            top_k_supplies = []
            for supply_candidate, available_bikes in decision_event.action_scope.
↪items():
                if supply_candidate == decision_event.station_idx:
                    continue

                heapq.heappush(top_k_supplies, (available_bikes, supply_candidate))
                if len(top_k_supplies) > self._supply_top_k:
                    heapq.heappop(top_k_supplies)

            max_reposition, source_idx = random.choice(top_k_supplies)
            action = Action(source_idx, decision_event.station_idx, max_reposition)

        return action
```

3.8.2 Interaction with the Greedy Policy

This environment is driven by real trip history data from Citi Bike.

```
env = Env(scenario=config.env.scenario, topology=config.env.topology, start_
↪tick=config.env.start_tick,
          durations=config.env.durations, snapshot_resolution=config.env.resolution)

if config.env.seed is not None:
    env.set_seed(config.env.seed)

policy = GreedyPolicy(config.agent.supply_top_k, config.agent.demand_top_k)
metrics, decision_event, done = env.step(None)
while not done:
    metrics, decision_event, done = env.step(policy.choose_action(decision_event))

env.reset()
```

Note: All related code snippets are supported in maro playground.

3.9 Simulation Toolkit

MARO simulation toolkit provides some predefined environments, such as CIM, Citi Bike, etc. It also provides some critical and reusable wheels for easily building new scenarios, with a high-performance guarantee and uniformed programming paradigm, including Data Model, Event Buffer, and Business Engine.

3.9.1 Overview

The key features of MARO simulation toolkit:

- Event-driven, which is friendly to business logging data, and more in line with real-world scenarios;
- High execution performance;
- Low development cost for new environments;
- Convenient interface for spatial-temporal data accessing, which is friendly to decision-related information querying.

In order to achieve the key features listed above, we choose Python as the frontend language. In the meantime, to avoid Python GIL problem and to achieve the better performance of memory allocation and cache hitting, we choose Cython as the backend language. Furthermore, we provide a declarative Data Model to auto-generate the related data class wrapper for easy underlying memory access, a Snapshot List slicing interface to quickly accessing data from the spatial-temporal and intra-node perspective. Considering the variously complex scenarios, we decide to build up an event-oriented simulation system, which is not only suitable to feed existing business logging data but also convenient to inject random events. New scenarios can be implemented in the pluggable Business Engine based on the Event Buffer supporting. The following figure illustrates the overall architecture of the MARO simulation system.

3.9.2 Environment

Well-designed topologies are provided in each predefined scenario (i.e. CIM, Citi Bike). You can specify both the scenario and the topology when initializing an environment. To distinguish the complicated problems in the real world, we provide various topologies with different difficulties for the user to do a step-by-step exploration. In general, the interface of environments is Gym-like, which use `step` to trigger the environment to apply `actions` from `agent(s)`. Furthermore, we concern more about the decision mode of multi-agent/hierarchical-agent and how to conveniently query data on the levels of spatial, temporal, and intra-node (resource holder). The following shows a typical environment interaction workflow and code snippet.

```
from maro.simulator import Env
from maro.simulator.scenarios.cim.common import Action

# Initialize an environment with a specific scenario, related topology.
# In Container Inventory Management, 1 tick means 1 day, here durations=100 means a␣
↪length of 100 days
env = Env(scenario="cim", topology="toy.5p_ssddd_l0.0", start_tick=0, durations=100)

# Query environment summary, which includes business instances, intra-instance␣
↪attributes, etc.
print(env.summary)

for ep in range(2):
    # Gym-like step function.
```

(continues on next page)

(continued from previous page)

```
    metrics, decision_event, is_done = env.step(None)

    while not is_done:
        past_week_ticks = [
            x for x in range(decision_event.tick - 7, decision_event.tick)
        ]
        decision_port_idx = decision_event.port_idx
        intr_port_infos = ["booking", "empty", "shortage"]

        # Query the snapshot list of the environment to get the information of
        # the booking, empty container inventory, shortage of the decision port in␣
↪the past week.
        past_week_info = env.snapshot_list["ports"][
            past_week_ticks : decision_port_idx : intr_port_infos
        ]

        dummy_action = Action(
            vessel_idx=decision_event.vessel_idx,
            port_idx=decision_event.port_idx,
            quantity=0
        )

        # Drive environment with dummy action (no repositioning).
        metrics, decision_event, is_done = env.step(dummy_action)

    # Query environment business metrics at the end of an episode,
    # it is your optimized object (usually includes multi-target).
    print(f"ep: {ep}, environment metrics: {env.metrics}")
    env.reset()
```

3.9.3 Critical Components

These components are the foundation of the whole MARO simulation system and can be easily reused to build up various real-world business scenarios with good performance and uniformed pattern. You can find more details in Data Model, Event Buffer, and Business Engine.

3.10 Data Model

The data model of MARO provides a declarative interface. We choose Python as the frontend language for saving development cost, and we take C as the backend language for improving the execution reference. What's more, the backend store is a pluggable design, user can choose different backend implementation based on their real performance requirement and device limitation.

3.10.1 Key Concepts

As shown in the figure above, there are some key concepts in the data model:

- **Node** is the abstraction of the resource holder, which is usually the major business instance of the scenario (i.e. vessels and ports in CIM scenario). A node usually has various attributes to present the business nature.
- **(Slot) Attribute** is the abstraction of business properties for the scenarios-specific resource holder (node). The attributes of a node could be declared as different data types based on the real requirements. Furthermore, for each attribute, a `slot` feature is provided to support the fixed-size array. The `slot` number can indicate the attribute values (e.g. the three different container types in CIM scenario) or the detailed categories (e.g. the ten specific products in the *Use Case* below). By default, the `slot` value is one.
- **Frame** is the collection of all nodes in the environment. The historical frames present the aggregated state of the environment during a specific period, while the current frame hosts the latest state of the environment at the current time point.
- **Snapshot List** is the dumped frames based on a pre-defined resolution. It captures the aggregated changes of the environment between the dump points.

3.10.2 Use Case

- Below is the declaration of a retail frame, which includes warehouse and store nodes.

```
from maro.backends.frame import node, NodeAttribute, NodeBase, FrameNode,␣
↪FrameBase

TOTAL_PRODUCT_CATEGORIES = 10
TOTAL_STORES = 8
TOTAL_WAREHOUSES = 2
TOTAL_SNAPSHOT = 100


@node("warehouse")
class Warehouse(NodeBase):
    inventories = NodeAttribute("i", TOTAL_PRODUCT_CATEGORIES)
    shortages = NodeAttribute("i", TOTAL_PRODUCT_CATEGORIES)

    def __init__(self):
        self._init_inventories = [100 * (i + 1) for i in range(TOTAL_PRODUCT_
↪CATEGORIES)]
        self._init_shortages = [0] * TOTAL_PRODUCT_CATEGORIES

    def reset(self):
        self.inventories[:] = self._init_inventories
        self.shortages[:] = self._init_shortages


@node("store")
class Store(NodeBase):
    inventories = NodeAttribute("i", TOTAL_PRODUCT_CATEGORIES)
    shortages = NodeAttribute("i", TOTAL_PRODUCT_CATEGORIES)
    sales = NodeAttribute("i", TOTAL_PRODUCT_CATEGORIES)

    def __init__(self):
        self._init_inventories = [10 * (i + 1) for i in range(TOTAL_PRODUCT_
↪CATEGORIES)]
```

(continues on next page)

(continued from previous page)

```
        self._init_shortages = [0] * TOTAL_PRODUCT_CATEGORIES
        self._init_sales = [0] * TOTAL_PRODUCT_CATEGORIES

    def reset(self):
        self.inventories[:] = self._init_inventories
        self.shortages[:] = self._init_shortages
        self.sales[:] = self._init_sales


class RetailFrame(FrameBase):
    warehouses = FrameNode(Warehouse, TOTAL_WAREHOUSES)
    stores = FrameNode(Store, TOTAL_STORES)

    def __init__(self):
        # If your actual frame number was more than the total snapshot number,␣
↪the old snapshots would be rolling replaced.
        super().__init__(enable_snapshot=True, total_snapshot=TOTAL_SNAPSHOT)
```

- The operations on the retail frame.

```
retail_frame = RetailFrame()

# Fulfill the initialization values to the backend memory.
for store in retail_frame.stores:
    store.reset()

# Fulfill the initialization values to the backend memory.
for warehouse in retail_frame.warehouses:
    warehouse.reset()

# Take a snapshot of the first tick frame.
retail_frame.take_snapshot(0)
snapshot_list = retail_frame.snapshots
print(f"Max snapshot list capacity: {len(snapshot_list)}")

# Query sales, inventory information of all stores at first tick, len(snapshot_
↪list["store"]) equals to TOTAL_STORES.
all_stores_info = snapshot_list["store"][0::["sales", "inventories"]].
↪reshape(TOTAL_STORES, -1)
print(f"All stores information at first tick (numpy array): {all_stores_info}")

# Query shortage information of first store at first tick.
first_store_shortage = snapshot_list["store"][0:0:"shortages"]
print(f"First store shortages at first tick (numpy array): {first_store_shortage}
↪")

# Query inventory information of all warehouses at first tick, len(snapshot_list[
↪"warehouse"]) equals to TOTAL_WAREHOUSES.
all_warehouses_info = snapshot_list["warehouse"][0::"inventories"].reshape(TOTAL_
↪WAREHOUSES, -1)
print(f"All warehouses information at first tick (numpy array): {all_warehouses_
↪info}")

# Add fake shortages to first store.
retail_frame.stores[0].shortages[:] = [i + 1 for i in range(TOTAL_PRODUCT_
↪CATEGORIES)]
retail_frame.take_snapshot(1)
```

(continues on next page)

(continued from previous page)

```
# Query shortage information of first and second store at first and second tick.
store_shortage_history = snapshot_list["store"][[0, 1]: [0, 1]: "shortages"].
↪reshape(2, -1)
print(f"First and second store shortage history at the first and second tick␣
↪(numpy array): {store_shortage_history}")
```

3.10.3 Supported Attribute Data Type

All supported data types for the attribute of the node:

Attribute Data Type	C Type	Range
i2	int16_t	-32,768 .. 32,767
i, i4	int32_t	-2,147,483,648 .. 2,147,483,647
i8	int64_t	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
f	float	-3.4E38 .. 3.4E38
d	double	-1.7E308 .. 1.7E308

3.10.4 Advanced Features

For better data access, we also provide some advanced features, including:

- **Attribute value change handler**: It is a hook function for the value change event on a specific attribute. The member function with the `_on_{attribute_name}_changed` naming pattern will be automatically invoked when the related attribute value changed. Below is the example code:

```
from maro.backends.frame import node, NodeBase, NodeAttribute

@node("test_node")
class TestNode(NodeBase):
    test_attribute = NodeAttribute("i")

    def _on_test_attribute_changed(self, value: int):
        pass
```

- **Snapshot list slicing**: It provides a slicing interface for querying temporal (frame), spatial (node), intra-node (attribute) information. Both a single index and an index list are supported for querying specific frame(s), node(s), and attribute(s), while the empty means querying all. The return value is a flattened 1-dimension NumPy array, which aligns with the slicing order as below:

```
snapshot_list = env.snapshot_list

# Get max size of snapshots (in memory).
print(f"Max snapshot size: {len(snapshot_list)}")

# Get snapshots of a specific node type.
test_nodes_snapshots = snapshot_list["test_nodes"]

# Get node instance amount.
print(f"Number of test_nodes in the frame: {len(test_nodes_snapshots)}")
```

(continues on next page)

(continued from previous page)

```
# Query one attribute on all frames and nodes.
states = test_nodes_snapshots[::"int_attribute"]

# Query two attributes on all frames and nodes.
states = test_nodes_snapshots[::["int_attribute", "float_attribute"]]

# Query one attribute on all frame and the first node.
states = test_nodes_snapshots[:0:"int_attribute"]

# Query attribute by node index list.
states = test_nodes_snapshots[:[0, 1, 2]:"int_attribute"]

# Query one attribute on the first frame and the first node.
states = test_nodes_snapshots[0:0:"int_attribute"]

# Query attribute by frame index list.
states = test_nodes_snapshots[[0, 1, 2]: 0: "int_attribute"]
```

3.11 Event Buffer

Event buffer is the execution engine of the MARO simulation system. Various business scenarios are constructed by different business event series and the related handler functions. To implement a scenario-specific environment, users need to provide the definitions of business events and related handler functions. After that, the input events of the event buffer will be automatically processed based on their priority. The priority of an event is first determined by its declared execution tick. Events of the same tick will be executed according to the FIFO rule. Currently, only a single-thread version event buffer is provided.

```
# Initialize an event buffer.
event_buffer: EventBuffer = EventBuffer()

# Insert a mock event.
event_buffer.insert_event(event)

# Execute events at a specific tick.
executed_events = event_buffer.execute(tick)
```

3.11.1 Event Category

To simplify the implementation of the business logic, MARO provides two kinds of basic event types, which can be used to construct various event execution pattern:

- **Atom event** is an event without any dependence. An atom event will be immediately popped out from the event buffer after execution.
- **Cascade event** is a series of events with dependencies. An internal event queue is hosted for the sub-events of each cascade event. Only after all sub-events executed, a cascade event can be popped out from the event buffer. Furthermore, the cascade event is chainable, which means that events in the internal event queue can also be cascade events.

3.11.2 Event Format

We provide a general-purpose event format for all the scenarios in MARO. A legal event generally contains the following properties:

- **tick** (int): The execution tick of this event.
- **event_type** (int): The type of this event. It is a customized field, the default value is 0 (PREDEFINE_EVENT_ACTION).
- **source** (str): The id/code of the event generator (not implemented yet).
- **target** (str): The id/code of the event receiver (not implemented yet).
- **payload** (Object): It is used to transfer sufficient information to the event handler.
- **tag** (EventCategory): The tag of the *event category*. Valid values include ATOM and CASCADE.
- **immediate_event_list** (list): The sub-event queue of the cascade event. Atom event does not have this field.
- **state** (EventState): The state of this event. Valid values include PENDING, EXECUTING and FINISHED. The figure below indicates the state changing of an event:

3.12 Business Engine

MARO provides a pluggable mechanism to isolate the business logic and the simulation kernel parts. For different business scenarios, specific business engines should be bind to it.

The business engine is responsible for defining:

- **Business instance**. Generally, the business instances are the resource holders in the business logic. For example:
  - The ports and vessels in the CIM scenario;
  - The stations in the Citi Bike scenario;
- **Business event**. Since it is closely related to the actual business logic, the business events are reproduced based on the existing business log data or generated according to the predefined business rules. For example:
  - The receiving of customers' orders, the vessel departure, etc in the CIM scenario;
  - The receiving of the customers' trip requirements, the finishing of a trip, etc in the Citi Bike scenario;
- **Decision event**. It is the external interactive interface for the resource optimization. The environment returns the decision events to the decision agent to trigger the agent's actions. For example:
  - The empty container repositioning operations that triggered by a vessel arrival event in the CIM scenario;
  - The bike repositioning operations that triggered when there are too many or too few bikes remained in the station in Citi Bike scenario;
- **Optimize metrics**. There are often multiple goals in real business scenarios, these metrics should be defined so as to be recorded to evaluate the repositioning. For example:
  - The shortage that used to measure the number of un-satisfied requirements;
  - The cost of the repositioning operations;

As shown in the figure below, the simulator will load the scenario-specific business engine and convert the filled business time series data into the corresponding business events. After that, the entire simulation system would be driven by these business events. In short, with the uniformed business engine interface, the simulation of different business scenarios is only based on the pluggable business engine (scenario-specific).

Generally, the business time series data is read from the historical log or generated by a data generation model. Currently, for topologies in Citi Bike scenario, data processing is needed before starting the simulation. You can find the brief introduction of the data processing command in Data Processing.

3.13 RL Toolkit

MARO provides a full-stack abstraction for reinforcement learning (RL), which empowers users to easily apply pre-defined and customized components to different scenarios in a scalable way. The main abstractions include *Learner, Actor*, *Agent Manager*, *Agent*, *Algorithm*, *State Shaper, Action Shaper, Experience Shaper*, etc.

3.13.1 Learner and Actor

- **Learner** is the abstraction of the learnable policy. It is responsible for learning a qualified policy to improve the business optimized object.

```
# Train function of learner.
def train(self, total_episodes):
    for current_ep in range(total_episodes):
        models = self._trainable_agents.get_models()
        performance, experiences = self._actor.roll_out(models=models,
                                                         epsilons=self._trainable_
↪agents.explorer.epsilons,
                                                         seed=self._seed)

        self._trainable_agents.store_experiences(experiences)
        self._trainable_agents.train()
        self._trainable_agents.update_epsilon(performance)
```

- **Actor** is the abstraction of experience collection. It is responsible for interacting with the environment and collecting experience. The experiences collected during interaction will be used for the training of the learners.

```
# Rollout function of actor.
def roll_out(self, models=None, epsilons=None, seed: int = None):
    self._env.set_seed(seed)

    # Assign epsilon
    if epsilons is not None:
        self._inference_agents.explorer.epsilons = epsilons

    # Load models
    if models is not None:
        self._inference_agents.load_models(models)

    metrics, decision_event, is_done = self._env.step(None)

    while not is_done:
        action = self._inference_agents.choose_action(decision_event, self._env.
↪snapshot_list)
        metrics, decision_event, is_done = self._env.step(action)
        self._inference_agents.on_env_feedback(metrics)
```

(continues on next page)

(continued from previous page)

```
    experiences = self._inference_agents.post_process(self._env.snapshot_list)
    performance = self._env.metrics
    self._env.reset()

    return {'local': performance}, experiences
```

3.13.2 Agent Manager

The agent manager provides a unified interactive interface with the environment for RL agent(s). From the actor's perspective, it isolates the complex dependencies of the various homogeneous/heterogeneous agents, so that the whole agent manager will behave just like a single agent. Besides that, the agent manager also plays the role of an agent assembler. It can assemble different RL agents according to the actual requirements, such as whether to share the underlying model, whether to share the experience pool, etc.

```
def assemble_agents(self, config):
    # Initialize experience pool instance.
    ...
    # Construct underlying learning model and related RL algorithm.
    ...
    for agent_id in self._agent_id_list:
        # Assemble your agent here, load experience pool, RL algorithms, etc.
        # You can control the experience pool and learning model sharing pattern,␣
↪based on different assembling strategy.
        self._agent_dict[agent_id] = Agent(...)
```

Furthermore, to well serve the distributed algorithm (scalable), the agent manager provides two kinds of working modes, which can be applied in different distributed components, such as inference mode in actor, training mode in learner.

- In **inference mode**, the agent manager is responsible to access and shape the environment state for the related agent, convert the model action to an executable environment action, and finally generate experiences from the interaction trajectory.
- In **training mode**, the agent manager will optimize the underlying model of the related agent(s), based on the collected experiences from in the inference mode.

3.13.3 Agent

An agent is a combination of (RL) algorithm, experience pool, and a set of non-algorithm-specific parameters (algorithm-specific parameters are managed by the algorithm module). Non-algorithm-specific parameters are used to manage experience storage, sampling strategies, and training strategies. Since all kinds of scenario-specific stuff will be handled by the agent manager, the agent is scenario agnostic.

```
class Agent(object):
    def __init__(self, name: str, algorithm: Algorithm, experience_pool: SimpleStore,␣
↪params: AgentParameters):
        """
            RL agent class. It's a sandbox for the RL algorithm, scenarios specific␣
↪details will be excluded out.
            We focus on the abstraction algorithm development here.
```

(continues on next page)

(continued from previous page)

```
            Environment observation and decision events will be converted to a␣
↪uniformed format before calling in.
            And the output will be converted to an environment executable format␣
↪before return back to the environment.
            Its key responsibility is optimizing policy based on interaction with the␣
↪environment.

            Args:
                name (str): The name of Agent.
                algorithm: A concrete algorithm instance that inherits from␣
↪AbstractAlgorithm. This is the centerpiece
                           of the Agent class and is responsible for the most␣
↪important tasks of an agent: choosing
                           actions and optimizing models.
                experience_pool (SimpleStore): A data store that stores experiences␣
↪generated by the experience shaper.
                params: A collection of hyper-parameters associated with the model␣
↪training loop.
        """
        ...
```

Under the management of the agent manager:

- In **inference mode**, given the shaped model state as input, the agent will output a model action (then the agent manager will shape it into an executable environment action). Also, at the end of each episode, the agent will fill the shaped experiences into the experience pool.
- In **training mode**, the agent will train and update its model with the experiences sampled from its experience pool.

3.13.4 Algorithm

The algorithm is the kernel abstraction of the RL formulation for a real-world problem. The model architecture, loss function, optimizer, and internal model update strategy are designed and parameterized here. In this module, two predefined interfaces must be implemented:

- `choose_action` is used to make a decision based on a provided model state.
- `train_on_batch` is used to trigger training and the policy update from external.

```
class Algorithm(object):
    def __init__(self, model_dict: dict, optimizer_opt: Union[dict, tuple], loss_func_
↪dict: dict, hyper_params):
        """
            It's the abstraction of RL algorithm, which provides a uniformed policy␣
↪interface, such choose_action, train_on_batch.
            We also provide some predefined RL algorithm based on it, such DQN, A2C,␣
↪etc. User can inherit form it to customized their own algorithms.

            Args:
                model_dict (dict): underlying models for the algorithm (e.g., for A2C,
                                   model_dict = {"actor": ..., "critic": ...})
                optimizer_opt (tuple or dict): tuple or dict of tuples of (optimizer_
↪class, optimizer_params) associated
                                               with the models in model_dict. If it␣
↪is a tuple, the optimizer to be
```

(continues on next page)

(continued from previous page)

```
                                                instantiated applies to all trainable␣
↪parameters from model_dict. If it
                                                is a dict, the optimizer will be␣
↪applied to the related model with the same key.
                loss_func_dict (dict): loss function types associated with the models␣
↪in model_dict.
                hyper_params: algorithm-specific hyper-parameter set.
        """
        ...
```

3.13.5 Shapers

MARO uses shapers to isolate business-related details and the algorithm modelings. It provides a clean interactive surface for RL agent(s). The followings are the three usually used shapers in RL formulations:

- **State shaper**: Given a decision event, the state shaper will extract relevant temporal-spatial information from the environment (snapshot list) for the decision agent. The output usually follows a format that can be directly inputted to the underlying algorithm.
- **Action shaper**: Once the agent outputs a decision action, the agent manager will call the action shaper to convert it into an executable environment action. Then, the executable environment action will be sent to the environment's `step` function to wake the sleeping environment.
- **Experience shaper**: At the end of each episode, the experience shaper will convert the agent's interaction trajectory to formatted learnable experiences, which usually contain the fields of `state`, `action`, and `reward`. For the storage of experiences, MARO use in-memory KV store. It can not only provide an extensible experience interface but also give the full control of constructing the algorithm-specific experience to users. As for the reward, since there are multiple optimized business metrics in a real-world business scenario, and the key performance index varies for different needs, how to calculate a simple scalar reward is not reasonable for a fixed pattern. So we left the reward definition to the end-user, and we only provide the raw business metrics in MARO. You can pass a reward function (e.g., a lambda) that directly calculates a reward based on these business metrics, or implement a helper method within the class. We recommend the latter one for complicated reward computations that require information from the environment trajectory and longer historical information (from the environment snapshot list). The actual shaping logic is encapsulated in the `_shape()` method, which converts the entire transition trajectory to experiences. By default, we provide a `k-step return` experience shaper for general usage, but for better performance, you need to carefully design this part according to your scenario and needs.

3.14 Distributed Toolkit

MARO distributed toolkit provides a unified, fast, and infrastructure-independent interface to support RL distributed training.

As shown in the overall architecture diagram above, MARO distributed toolkit follows a message-passing pattern that the cooperation between different components is based on the messages sending and receiving. A typical master/worker distributed program usually contains the following steps:

1. The master component will send tasks(w/ or w/o data) to the worker components;
2. The worker components will finish the tasks in their local computing environments or the local devices;
3. The worker components return the computed results to the master component.

According to the actual needs, the communication mode between master and worker components can be synchronous or asynchronous.

3.14.1 Key Components

There are two key components in the distributed toolkit:

- **Communication**: It provides the general message passing interfaces, such as `(i)send`, `receive`, `(i)broadcast`, `(i)scatter`, etc. The communication component use a replaceable communication protocol driver to adopt different communication protocol stack (e.g. TCP/IP, InfiniBand%20is%20a%20computer,both%20among%20and%20within%20computers.)). Check the distributed communication to get more details.
- **Orchestration**: It primarily provides a unified interface for cluster management and job management on different infrastructures. Check the distributed orchestration to get more details.

3.15 Distributed Communication

The distributed communication component provides a general message passing mechanism, which is used to build various communication topologies in the distributed environment. Besides the essential communication primitive supporting, it also provides the functions of peer discovering, fault recovering (partially), conditional event auto-dispatching, etc.

3.15.1 Proxy

Providing an implementation of the communication primitives, proxy is the primary entity of the communication component. Proxy provides a uniformed communication interface, the underlying driver is pluggable based on the real requirements. Currently, we use ZeroMQ as the default choice. Proxy also provides support for peer discovering based on Redis.

Message

Message is designed for general purpose, it is used to package the communication content between components. The main attributes of a message instance include:

- `tag`: A customized attribute, it can be used to implement the auto-dispatching logic with a *conditional event register table*.
- `source`: The alias of the message sender.
- `destination`: The alias of the message receiver.
- `payload`: A Python object for remote function call.
- `session_id` (auto-generated): UUID of a specific session, one session may include multiple messages.
- `message_id` (auto-generated): UUID of a specific message.

```
from maro.communication import Message

message = Message(tag="check_in",
                  source="worker_001",
                  destination="master",
                  payload="")
```

Session Message

We provide two kinds of predefined session types for common distributed scenarios:

- **Task Session**: It is used to describe a computing task sent from master to worker. Three stages are included:
  1. The master sends the task request(s) to the worker(s);
  2. Once the worker(s) receiving the task(s), the worker(s) start to finish the task(s);
  3. The worker(s) return the computing result(s) to the master.
- **Notification Session**: It is used for information syncing and only includes two stages:
  1. The sender sends out the notification message;
  2. The receiver(s) receive the notification message.

The stages of each session are maintained internally by the proxy.

```
from maro.communication import SessionMessage, SessionType

task_message = SessionMessage(tag="sum",
                              source="master",
                              destination="worker_001",
                              payload=[0, 1, 2, ...],
                              session_type=SessionType.TASK)

notification_message = SessionMessage(tag="check_out",
                                      source="worker_001",
                                      destination="master",
                                      payload="",
                                      session_type=SessionType.NOTIFICATION)
```

Communication Primitives

Proxy provides a set of general-purpose communication primitives that support both blocking and non-blocking cases. These primitives are decoupled from the underlying implementation of the communication driver (protocol). The main primitives are listed below:

- `send`: Unicast. It is a blocking, one-to-one sending mode. It will watch and collect the reply message from the remote peer.
- `isend`: The non-blocking version of the `send`. A message session ID will be immediately returned, which can be used by `receive_by_id`.
- `scatter`: An advanced version of `send`. Is is used to send message(s) to peer(s) and watch and collect reply message(s) from the peer(s). `scatter` is not a real multi-cast, each message will go through the full TCP/IP stack (ZeroMQ driver). If the message you want to send is completely same and you want better performance, use the `broadcast` interface instead.

- `iscatter`: The non-blocking version of the `scatter`. The related messages session ID(s) will be returned immediately, which can be used by `receive_by_id`.
- `broadcast`: A blocking function call which is used to broadcast the message to all subscribers, it will watch and collect all subscribers' reply messages.
- `ibroadcast`: The non-blocking version of the `broadcast`. The related messages session IDs will be returned immediately, which can be used by `receive_by_id`.
- `receive`: It is used to continually receive the message.
- `receive_by_id`: It only receives the message(s) with the given session ID(s).

3.15.2 Conditional Event Register Table

The conditional event register table provides a message auto-despatching mechanism. By registering the `conditional event` and related `handler function` to the register table, the handler function will be automatically executed with the received messages when the event conditions are met.

`Conditional event` is used to declare the required message group for auto-triggering the related handler function. The unit event is the minimal component in the conditional event, it follows a three-stage format: *source*:*tag*:`amount`.

- `source`: It is used to declare the required message source. The longest-prefix matching is supported.
  - `*` is used to present any sources.
- `tag`: The `tag` attribute of the message instance.
  - `*` is used to present any tags.
- `amount`: The required message instance amount. Both a absolute integer and a relative percentage are valid for this field.
  - `%` is used to represent the relative percentages, such as 60%, 10%, etc.

```
unit_event_abs = "worker:update:10"

unit_event_rel = "worker:update:60%"
```

To support more complex business logic, we provide two operations: `AND` and `OR` to combine unit events up:

- `AND`: Valid for multiple unit events and combined unit events. The combined event condition is met if all the conditions of the sub-events are met.
- `OR`: Valid for multiple unit events and combined unit events. The combined event condition is met if any sub-event meets the condition.

```
combined_event_and = ("worker_01:update:2",
                      "worker_02:update:3",
                      "AND")

combined_event_or = ("worker_03:update:1",
                     "worker_04:update:5",
                     "OR")

combined_event_mix = (("worker_01:update:2", "worker_02:update:3", "AND"),
                      "worker_03:update:1",
                      "OR")
```

`Handler function` is a user-defined callback function that is bind to a specific conditional event. When the condition of the event is met, the related messages will be sent to the handler function for its execution.

```
# A common handler function signature
def handler(that, proxy, messages):
    """
        Conditional event handler function.

        Args:
            that: local instance reference, which needs to be operated.
            proxy: the proxy reference for remote communication.
            messages: received messages.
    """
    pass
```

3.15.3 Distributed Decorator

Distributed decorator is a helper for generating a distributed worker class from a local functional class.

```
from maro.communication import dist, Proxy

# Initialize proxy instance for remote communication.
proxy = Proxy(group_name="master-worker",
              component_type="worker",
              expected_peers=[("master", 1)])

# Declare the trigger condition of rollout event.
rollout_event = "master:rollout:1"

# Implement rollout event handler logic.
def on_rollout(that, proxy, messages):
    pass

# Assemble event-handler directory.
handler_dict = {rollout_event: on_rollout}

# Convert a local functional class to a distributed one.
@dist(proxy, handler_dict)
class Worker:
    def __init__(self):
        pass
```

3.16 Distributed Orchestration

MARO provides easy-to-use CLI commands to provision and manage training clusters on cloud computing service like Azure. These CLI commands can also be used to schedule the training jobs with the specified resource requirements. In MARO, all training job related components are dockerized for easy deployment and resource allocation. It provides a unified abstraction/interface for different orchestration framework (e.g. *Grass*, *Kubernetes*).

3.16.1 Grass

Grass is a self-designed, development purpose orchestration framework. It can be confidently applied to small/middle size cluster (< 200 nodes). The design goal of Grass is to speed up the distributed algorithm prototype development. It has the following advantages:

- Fast deployment in a small cluster.
- Fine-grained resource management.
- Lightweight, no other dependencies are required.

In the Grass mode:

- All VMs will be deployed in the same virtual network for a faster, more stable connection and larger bandwidth. Please note that the maximum number of VMs is limited by the available dedicated IP addresses.
- It is a centralized topology, the master node will host Redis service for peer discovering, Fluentd service for log collecting, SMB service for file sharing.
- On each VM, the probe (worker) agent is used to track the computing resources and detect abnormal events.

Check Grass Cluster Provisioning on Azure to get how to use it.

3.16.2 Kubernetes

MARO also supports Kubernetes (k8s) as an orchestration option. With this widely used framework, you can easily build up your training cluster with hundreds and thousands of nodes. It has the following advantages:

- Higher durability.
- Better scalability.

In the Kubernetes mode:

- The dockerized job component runs in Kubernetes pod, and each pod only hosts one component.
- All Kubernetes pods are registered into the same virtual network using Container Network Interface(CNI).

Check K8S Cluster Provisioning on Azure to get how to use it.