Maze

enliteAI GmbH

Feb 23, 2021

CONTENTS

1 Getting Started | | 1

2 Spotlights 209

3 Documentation Overview 211

4 Indices and tables 333

Index 335

CHAPTER

ONE

GETTING STARTED | GITHUB_LOGO1.P |

1.1 Installation

To install Maze with pip, run:

```
pip install maze-rl
```

Note: Pip does not install PyTorch, you need to make sure it is available in your Python environment.

If you want to use RLLib it in combination with Maze, optionally install it with

```
pip install ray[rllib] tensorflow
```

To install the bleeding-edge development version from github, first clone the repo.

```
git clone https://github.com/enlite-ai/maze.git
cd maze
```

Finally, install the project with pip in development mode and you are good to go and ready to start developing.

```
pip install -e .
```

Alternatively you can install with pip directly from the GitHub repository

```
pip install git+https://github.com/enlite-ai/maze.git
```

1.2 A First Example

This example shows how to train and rollout a policy for the CartPole environment with A2C. It also gives a small glimpse into the Maze framework.

1.2.1 Training and Rollouts

To *train a policy* with the synchronous advantage actor-critic (*A2C*), run:

```
maze-run -cn conf_train env.name=CartPole-v0 algorithm=a2c
```

All outputs of the training run including model weights will be stored in `outputs/<exp-dir>`.

To *perform rollouts* for evaluating the trained policy, run:

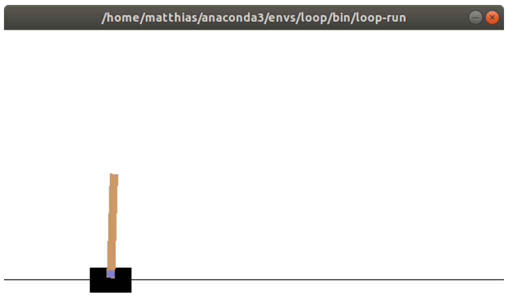
```
maze-run env.name=CartPole-v0 policy=torch_policy input_dir=outputs/<exp-dir>
```

This performs 50 rollouts and prints the resulting performance statistics to the command line:

```
 step|path                                                                 |        ␣
↪  value
=====|=====================================================================|====================
    1|rollout_stats  DiscreteActionEvents  action    substep_0/action      | [len:7900,
↪ :0.5]
    1|rollout_stats  BaseEnvEvents         reward    median_step_count     |        ␣
↪157.500
    1|rollout_stats  BaseEnvEvents         reward    mean_step_count       |        ␣
↪158.000
    1|rollout_stats  BaseEnvEvents         reward    total_step_count      |       ␣
↪7900.000
    1|rollout_stats  BaseEnvEvents         reward    total_episode_count   |        ␣
↪ 50.000
    1|rollout_stats  BaseEnvEvents         reward    episode_count         |        ␣
↪ 50.000
    1|rollout_stats  BaseEnvEvents         reward    std                   |        ␣
↪ 31.843
    1|rollout_stats  BaseEnvEvents         reward    mean                  |        ␣
↪158.000
    1|rollout_stats  BaseEnvEvents         reward    min                   |        ␣
↪ 83.000
    1|rollout_stats  BaseEnvEvents         reward    max                   |        ␣
↪200.000
```

To see the policy directly in action you can also perform sequential rollouts with rendering:

```
maze-run env.name=CartPole-v0 policy=torch_policy input_dir=outputs/<exp-dir> \
runner=sequential runner.render=true
```

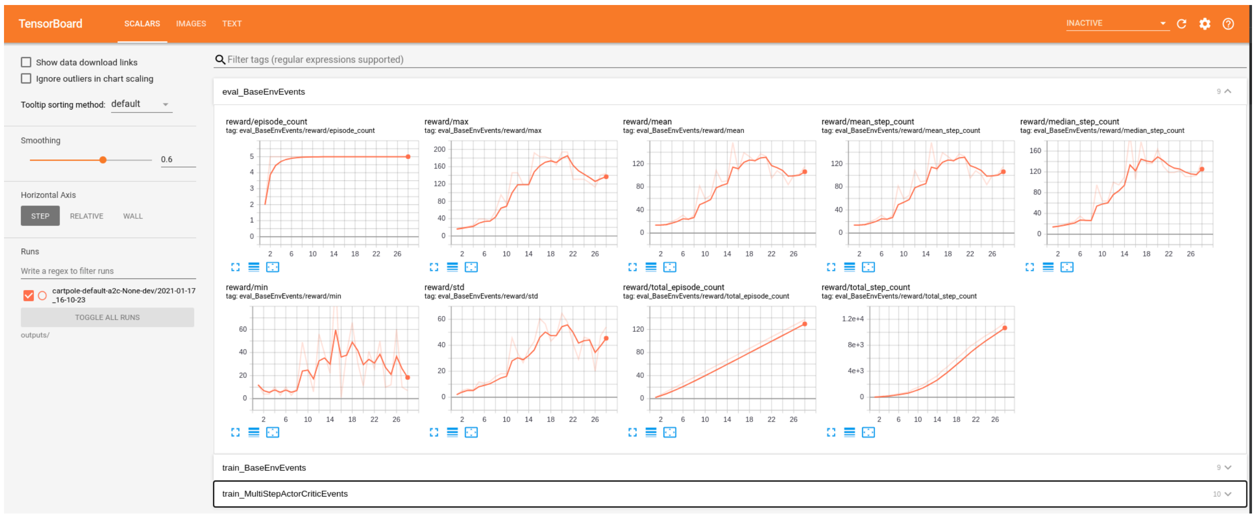


1.2.2 Tensorboard

To *watch the training progress with Tensorboard* start it by running:

```
tensorboard --logdir outputs/
```

and view it with your browser at http://localhost:6006/.



1.2.3 Training Outputs

For easier reproducibility Maze writes the full *configuration compiled with Hydra* to the command line an preserves it in the *TEXT* tab of Tensorboard along with the original training command.

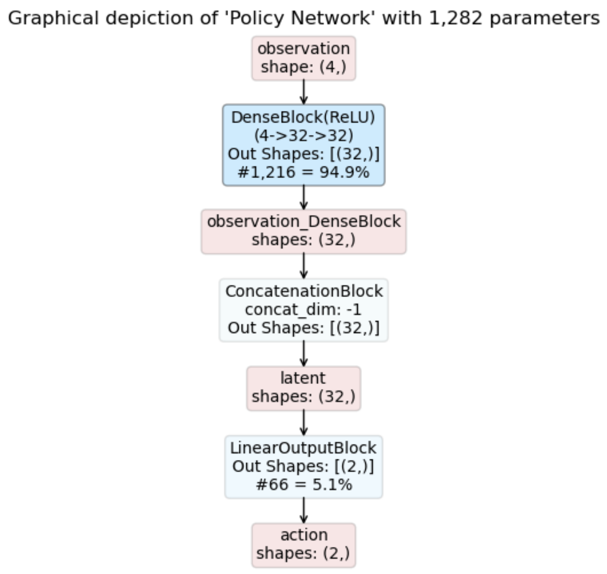
```
algorithm:
  device: cpu
  entropy_coef: 0.0
  gae_lambda: 1.0
  gamma: 0.98
  lr: 0.0005
  max_grad_norm: 0.0
  n_rollout_steps: 20
  policy_loss_coef: 1.0
  value_loss_coef: 0.5
env:
```

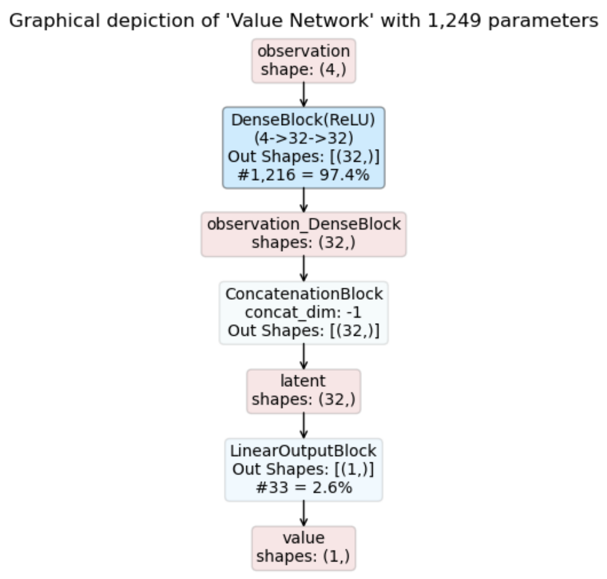
(continues on next page)

(continued from previous page)

```
  env: CartPole-v0
  type: maze.core.wrappers.maze_gym_env_wrapper.GymMazeEnv
log_base_dir: outputs
model:
  type: maze.perception.models.template_model_composer.TemplateModelComposer
  distribution_mapper_config: {}
  model_builder:
    type: maze.perception.builders.ConcatModelBuilder
    modality_config:
      feature:
        block_params:
          hidden_units: [32, 32]
          non_lin: torch.nn.SELU
        block_type: dense
      hidden: {}
      recurrence: {}
    observation_modality_mapping:
      observation: feature
  critics:
    type: maze.perception.models.critics.StateCriticComposer
...
```

You will also find PDFs showing the *inference graphs of the policy and critic networks* in the experiment output directory. This turns out to be extremely useful when playing around with model architectures or when returning to experiments at a later stage.


Graphical depiction of 'Policy Network' with 1,282 parameters
observation shape: (4,)
DenseBlock(ReLU) (4->32->32) Out Shapes: [(32,)] #1,216 = 94.9%
observation_DenseBlock shapes: (32,)
ConcatenationBlock concat_dim: -1 Out Shapes: [(32,)]
latent shapes: (32,)
LinearOutputBlock Out Shapes: [(2,)] #66 = 5.1%
action shapes: (2,)



1.3 Maze - Step by Step

This tutorial provides a step by step guide explaining how to implement your own Maze environment and get the best out of its features. We will do this based on the online version of the *Guillotine 2D Cutting Stock Problem* as it is still relatively simple but exhibits the required problem structure to introduce all relevant Maze concepts.

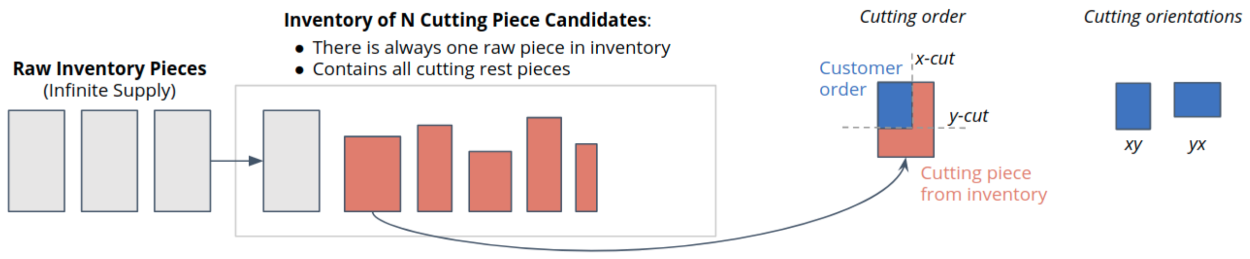
Before diving into this tutorial we recommend to read up on the *Maze Environment Hierarchy*. You can of course also do this along the way following the provided links to explanations of the required concepts when we get there.

The remainder of this tutorial is structured as follows:

1.3.1 Cutting-2D Problem Specification

This page introduces the problem we would like to address with a Deep Reinforcement Learning agent: an online version of the *Guillotine 2D Cutting Stock Problem*.

Description of Problem:

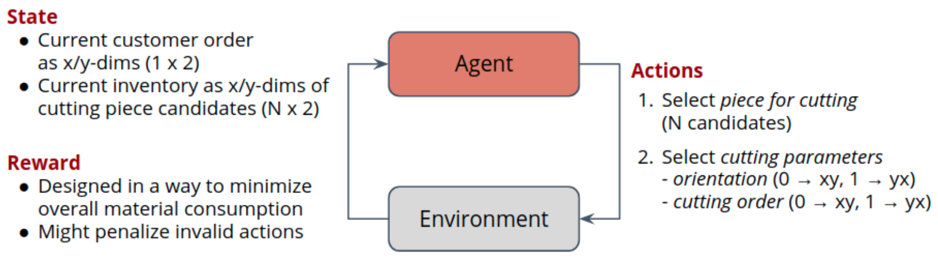

Inventory of N Cutting Piece Candidates:
• There is always one raw piece in inventory
• Contains all cutting rest pieces
Raw Inventory Pieces
(Infinite Supply)
Cutting order
Customer order
x-cut
y-cut
Cutting piece from inventory
Cutting orientations
xy
yx

- In each step there is one new incoming customer order generated according to a certain demand pattern.
- This customer order has to be fulfilled by cutting the exact x/y-dimensions from a set of available candidate pieces in the inventory.
- A new raw piece is transferred to the inventory every time the current raw piece in inventory is used to cut and deliver a customer order.

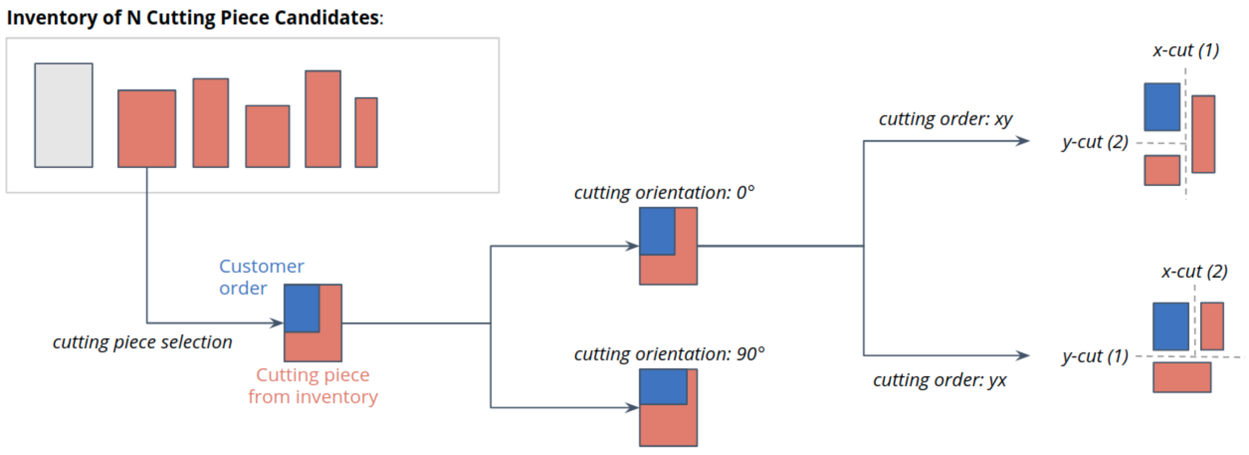
- The goal is to use as few raw pieces as possible throughout the episode, which can be achieved by following a clever cutting policy.

Agent-Environment Interaction Loop:

To make the problem more explicit from an RL perspective we formulate it according to the agent-environment interaction loop shown below.


State
Current customer order as x/y-dims (1 x 2)
Current inventory as x/y-dims of cutting piece candidates (N x 2)
Reward
Designed in a way to minimize overall material consumption
Might penalize invalid actions
Agent
Environment
Actions
1. Select piece for cutting (N candidates)
2. Select cutting parameters
- orientation (0 → xy, 1 → yx)
- cutting order (0 → xy, 1 → yx)

- The *State* contains the dimensions of the currently pending customer orders and all pieces on inventory.
- The *Reward* is specified to discourage the usage of raw inventory pieces.
- The *Action* is a joint action consisting of the following components (see image below for details):
  - Action a_0: Cutting piece selection (decides which piece from inventory to use for cutting)
  - Action a_1: Cutting orientation selection (decides the orientation of the customer)
  - Action a_2: Cutting order selection (decides which cut to take first; x or y)



Given this description of the problem we will now proceed with implementing a corresponding simulation.

1.3.2 Implementing the CoreEnv

The complete code for this part of the tutorial can be found here

```
# file structure
- cutting_2d
    - main.py
    - env
        - core_env.py
        - inventory.py
        - maze_state.py
        - maze_action.py
```

Page Overview

- *CoreEnv*
- *Environment Components*
- *MazeState and MazeAction*
- *Test Script*

CoreEnv

The first component we need to implement is the *Core Environment* which defines the main mechanics and functionality of the environment.

For this example we will call it `Cutting2DCoreEnvironment`. As for any other Gym environment we need to implement several methods according to the `CoreEnv` interface. We will start with the very basic components and add more and more features (complexity) throughout this tutorial:

- `step()`: Implements the cutting mechanics.
- `reset()`: Resets the environment as well as the piece inventory.
- `seed()`: Sets the random state of the environment for reproducibility.
- `close()`: Can be used for cleanup.
- `get_maze_state()`: Returns the current MazeState of the environment.

You can find the implementation of the basic version of the `Cutting2DCoreEnvironment` below.

Listing 1: env/core_env.py

```python
from typing import Union, Tuple, Dict, Any

import numpy as np

from maze.core.env.core_env import CoreEnv
from maze.core.utils.seeding import set_random_states

from .maze_state import Cutting2DMazeState
from .maze_action import Cutting2DMazeAction
from .inventory import Inventory


class Cutting2DCoreEnvironment(CoreEnv):
    """Environment for cutting 2D pieces based on the customer demand. Works as␣
↪follows:
     - Keeps inventory of 2D pieces available for cutting and fulfilling the demand.
     - Produces a new demand for one piece in every step (here a static demand).
     - The agent should decide which piece from inventory to cut (and how) to fulfill␣
↪the given demand.
     - What remains from the cut piece is put back in inventory.
     - All the time, one raw (full-size) piece is available in inventory.
       (If it gets cut, it is replenished in the next step.)
     - Rewards are calculated to motivate the agent to consume as few raw pieces as␣
↪possible.
     - If inventory gets full, the oldest pieces get discarded.
```

(continues on next page)

(continued from previous page)

```
    :param max_pieces_in_inventory: Size of the inventory.
    :param raw_piece_size: Size of a fresh raw (= full-size) piece.
    :param static_demand: Order to issue in each step.
    """

    def __init__(self, max_pieces_in_inventory: int, raw_piece_size: (int, int),
                 static_demand: (int, int)):
        super().__init__()

        self.max_pieces_in_inventory = max_pieces_in_inventory
        self.raw_piece_size = tuple(raw_piece_size)
        self.current_demand = static_demand

        # setup environment
        self._setup_env()

    def _setup_env(self):
        """Setup environment."""
        self.inventory = Inventory(self.max_pieces_in_inventory, self.raw_piece_size)
        self.inventory.replenish_piece()

    def step(self, maze_action: Cutting2DMazeAction) \
            -> Tuple[Cutting2DMazeState, np.array, bool, Dict[Any, Any]]:
        """Summary of the step (simplified, not necessarily respecting the actual␣
↪order in the code):
        1. Check if the selected piece to cut is valid (i.e. in inventory, large␣
↪enough etc.)
        2. Attempt the cutting
        3. Replenish a fresh piece if needed and return an appropriate reward

        :param maze_action: Cutting MazeAction to take.
        :return: maze_state, reward, done, info
        """

        info, reward = {}, 0
        replenishment_needed = False

        # check if valid piece id was selected
        if maze_action.piece_id >= self.inventory.size():
            info['error'] = 'piece_id_out_of_bounds'
        # perform cutting
        else:
            piece_to_cut = self.inventory.pieces[maze_action.piece_id]

            # attempt the cut
            if self.inventory.cut(maze_action, self.current_demand):
                info['msg'] = "valid_cut"
                replenishment_needed = piece_to_cut == self.raw_piece_size
            else:
                # assign a negative reward for invalid cutting attempts
                info['error'] = "invalid_cut"
                reward = -2

        # check if replenishment is required
        if replenishment_needed:
            self.inventory.replenish_piece()
```

(continues on next page)

(continued from previous page)

```
            # assign negative reward if a piece has to be replenished
            reward = -1

        # compile env state
        maze_state = self.get_maze_state()

        return maze_state, reward, False, info

    def get_maze_state(self) -> Cutting2DMazeState:
        """Returns the current Cutting2DMazeState of the environment."""
        return Cutting2DMazeState(self.inventory.pieces, self.max_pieces_in_inventory,
                                  self.current_demand, self.raw_piece_size)

    def reset(self) -> Cutting2DMazeState:
        """Resets the environment to initial state."""
        self._setup_env()
        return self.get_maze_state()

    def close(self):
        """No additional cleanup necessary."""

    def seed(self, seed: int) -> None:
        """Seed random state of environment."""
        set_random_states(seed)

    # --- lets ignore everything below this line for now ---

    def get_renderer(self) -> Any:
        pass

    def get_serializable_components(self) -> Dict[str, Any]:
        pass

    def is_actor_done(self) -> bool:
        pass

    def actor_id(self) -> Tuple[Union[str, int], int]:
        pass
```

Environment Components

To keep the implementation of the core environment short and clean we introduces a dedicated `Inventory` class providing functionality for:

- maintaining the inventory of available cutting pieces
- replenishing new *raw inventory pieces* if required
- the cutting logic of the environment

Listing 2: env/inventory.py

```
from .maze_action import Cutting2DMazeAction


class Inventory:
```

(continues on next page)

(continued from previous page)

```
    """Holds the inventory of 2D pieces and performs cutting.
    :param max_pieces_in_inventory: Size of the inventory. If full, the oldest pieces␣
↪get discarded.
    :param raw_piece_size: Size of a fresh raw (= full-size) piece.
    """

    def __init__(self, max_pieces_in_inventory: int, raw_piece_size: (int, int)):
        self.max_pieces_in_inventory = max_pieces_in_inventory
        self.raw_piece_size = raw_piece_size

        self.pieces = []

    # == Inventory management ==

    def is_full(self) -> bool:
        """Checks weather all slots in the inventory are in use."""
        return len(self.pieces) == self.max_pieces_in_inventory

    def store_piece(self, piece: (int, int)) -> None:
        """Store the given piece.
        :param piece: Piece to store.
        """
        # If we would run out of storage space, discard the oldest piece first
        if self.is_full():
            self.pieces.pop(0)
        self.pieces.append(piece)

    def replenish_piece(self) -> None:
        """Add a fresh raw piece to inventory."""
        self.store_piece(self.raw_piece_size)

    # == Cutting ==

    def cut(self, maze_action: Cutting2DMazeAction, ordered_piece: (int, int)) ->␣
↪bool:
        """Attempt to perform the cutting. Remains of the cut piece are put back to␣
↪inventory.

        :param maze_action: the cutting maze_action to perform
        :param ordered_piece: Dimensions of the piece that we should produce
        :return True if the cutting was successful, False on error.
        """
        if maze_action.rotate:
            ordered_piece = ordered_piece[::-1]

        # Check the piece ID is valid
        if maze_action.piece_id >= len(self.pieces):
            return False

        # Check whether the cut is possible
        if any([ordered_piece[dim] > available_size for dim, available_size
                in enumerate(self.pieces[maze_action.piece_id])]):
            return False

        # Perform the cut
        cutting_order = [1, 0] if maze_action.reverse_cutting_order else [0, 1]
        piece_to_cut = list(self.pieces.pop(maze_action.piece_id))
```

(continues on next page)

(continued from previous page)

```
        for dim in cutting_order:
            residual = piece_to_cut.copy()
            residual[dim] = piece_to_cut[dim] - ordered_piece[dim]
            piece_to_cut[dim] = ordered_piece[dim]
            if residual[dim] > 0:
                self.store_piece(tuple(residual))

        return True

    # == State representation ==

    def size(self) -> int:
        """Current size of the inventory."""
        return len(self.pieces)
```

MazeState and MazeAction

As motivated and explained in more detail in our tutorial on *Customizing Core and Maze Envs* CoreEnvs rely on MazeState and MazeAction objects for interacting with an agent.

For the present case this is a `Cutting2DMazeState`

Listing 3: env/maze_state.py

```
class Cutting2DMazeState:
    """Cutting 2D environment MazeState representation.
    :param inventory: A list of pieces in inventory.
    :param max_pieces_in_inventory: Max number of pieces in inventory (inventory␣
↪size).
    :param current_demand: Piece that should be produced in the next step.
    :param raw_piece_size: Size of a raw piece.
    """

    def __init__(self, inventory: [(int, int)], max_pieces_in_inventory: int,
                 current_demand: (int, int), raw_piece_size: (int, int)):
        self.inventory = inventory.copy()
        self.max_pieces_in_inventory = max_pieces_in_inventory
        self.current_demand = current_demand
        self.raw_piece_size = raw_piece_size
```

and a `Cutting2DMazeAction` defining which inventory piece to cut in which cutting order and orientation.

Listing 4: env/maze_action.py

```
class Cutting2DMazeAction:
    """Environment cutting MazeAction object.
    :param piece_id: ID of the piece to cut.
    :param rotate: Whether to rotate the ordered piece.
    :param reverse_cutting_order: Whether to cut along Y axis first (not X first as␣
↪normal).
    """

    def __init__(self, piece_id: int, rotate: bool, reverse_cutting_order: bool):
        self.piece_id = piece_id
        self.rotate = rotate
        self.reverse_cutting_order = reverse_cutting_order
```

These two classes are utilized in the *CoreEnv code above*.

Test Script

The following snippet will instantiate the environment and run it for 15 steps.

Listing 5: main.py

```
""" Test script CoreEnv """
from tutorials.tutorial_maze_env.part01_core_env.env.core_env import␣
↪Cutting2DCoreEnvironment
from tutorials.tutorial_maze_env.part01_core_env.env.maze_action import␣
↪Cutting2DMazeAction


def main():
    # init and reset core environment
    core_env = Cutting2DCoreEnvironment(max_pieces_in_inventory=200, raw_piece_
↪size=[100, 100],
                                        static_demand=(30, 15))
    maze_state = core_env.reset()
    # run interaction loop
    for i in range(15):
        # create cutting maze_action
        maze_action = Cutting2DMazeAction(piece_id=0, rotate=False, reverse_cutting_
↪order=False)
        # take actual environment step
        maze_state, reward, done, info = core_env.step(maze_action)
        print(f"reward {reward} | done {done} | info {info}")


if __name__ == "__main__":
    """ main """
    main()
```

When running the script you should get the following command line output:

```
reward -1 | done False | info {'msg': 'valid_cut'}
reward 0 | done False | info {'msg': 'valid_cut'}
reward 0 | done False | info {'msg': 'valid_cut'}
...
```

1.3.3 Adding a Renderer

The complete code for this part of the tutorial can be found here

```
# file structure
- cutting_2d
    - main.py  # modified
    - env
        - core_env.py  # modified
        - inventory.py
        - maze_state.py
        - maze_action.py
        - renderer.py  # new
```

Page Overview

- *Renderer*
- *Updating the CoreEnv*
- *Test Script*

Renderer

To check whether our implementation of the environment works as expected and to later on observe how trained agents behave we add a `Renderer` as a next step in this tutorial.

For implementing the renderer we rely on matplotlib to ensure that it is compatible with the *Maze Rollout Visualization Tools*.

The `Cutting2DRenderer` will show the selected piece (the MazeAction) on the left, along with the current MazeState of the inventory on the right as shown *here*.

Listing 6: env/renderer.py

```
from typing import Tuple, Optional

import numpy as np
import matplotlib.patches as patches
import matplotlib.pyplot as plt

from maze.core.annotations import override
from maze.core.log_events.step_event_log import StepEventLog
from maze.core.rendering.renderer import Renderer
from .maze_action import Cutting2DMazeAction
from .maze_state import Cutting2DMazeState


class Cutting2DRenderer(Renderer):
    """Rendering class for the 2D cutting env.

    The ``Cutting2DRenderer`` will show the selected piece (the maze_action) on the␣
↪left,
    plus the current state of the inventory on the right
    """

    @override(Renderer)
    def render(self, maze_state: Cutting2DMazeState, maze_action:␣
↪Optional[Cutting2DMazeAction], events: StepEventLog) -> None:
        """
        Render maze_state and maze_action of the cutting 2D env.

        :param maze_state: MazeState to render
        :param maze_action: MazeAction to render
        :param events: Events logged during the step (not used)
        """

        plt.figure("Cutting 2D", figsize=(8, 4))
        plt.clf()
```

(continues on next page)

(continued from previous page)

```
        # The maze_action taken

        plt.subplot(121, aspect='equal')
        if maze_action is not None:
            self._plot_maze_action(maze_action, "MazeAction", maze_state)
        else:
            self._add_title("MazeAction (none)")

        # The inventory state
        plt.subplot(122, aspect='equal')
        self._plot_inventory(maze_state, maze_action)

        plt.tight_layout()
        plt.draw()
        plt.pause(0.1)

    def _plot_maze_action(self, maze_action: Cutting2DMazeAction, title: str, maze_
↪state: Cutting2DMazeState):
        piece_to_cut = maze_state.inventory[maze_action.piece_id]
        if maze_action.rotate:
            piece_to_cut = piece_to_cut[::-1]

        plt.xlim([0, maze_state.raw_piece_size[0]])
        plt.ylim([0, maze_state.raw_piece_size[1]])

        self._draw_piece(piece_to_cut)
        self._draw_piece(maze_state.current_demand, highlight=True)
        self._draw_cutting_lines(maze_state.current_demand, piece_to_cut, maze_action.
↪reverse_cutting_order)
        self._add_title(title)

    def _plot_inventory(self, maze_state: Cutting2DMazeState, maze_action:␣
↪Cutting2DMazeAction):

        # plot inventory pieces
        inventory_piece_dims = np.vstack(maze_state.inventory)
        inventory_piece_dims = np.sort(inventory_piece_dims, axis=1)
        plt.plot(inventory_piece_dims[:, 0], inventory_piece_dims[:, 1], "ko",
                 alpha=0.5, label="inventory pieces")
        # plot current demand
        current_demand = sorted(maze_state.current_demand)
        plt.plot(current_demand[0], current_demand[1], "o",
                 color=(0.7, 0.2, 0.2), alpha=0.75, label="current demand")
        # plot maze_action
        piece_to_cut = inventory_piece_dims[maze_action.piece_id]
        plt.plot(piece_to_cut[0], piece_to_cut[1], "bo",
                 alpha=0.75, label="cutting inventory piece")
        plt.grid()
        plt.legend()
        plt.axis("equal")
        self._add_title("Inventory Pieces")

    @staticmethod
    def _draw_piece(piece: Tuple[int, int], highlight: bool = False):
        plt.gca().add_patch(patches.Rectangle((0, 0), piece[0], piece[1],
                                              facecolor=(0.7, 0.2, 0.2) if highlight␣
↪else (0.8, 0.8, 0.8)))
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def _add_title(title: str):
        plt.title(title, fontdict=dict(fontsize=16, fontweight='bold',␣
↪horizontalalignment='left'), loc='left')

    @staticmethod
    def _draw_cutting_lines(ordered_piece: Tuple[int, int], piece_to_cut: Tuple[int,␣
↪int], reverse_cutting_order: bool):
        """Draw the cutting lines.

        :param ordered_piece: Size of the ordered piece
        :param piece_to_cut: Piece which we are cutting
        :param reverse_cutting_order: If we should cut along Y axis first (instead of␣
↪X first)
        """

        if reverse_cutting_order:
            h_x = (0, piece_to_cut[0])
            h_y = (ordered_piece[1], ordered_piece[1])
            v_x = (ordered_piece[0], ordered_piece[0])
            v_y = (0, ordered_piece[1])
        else:
            h_x = (0, ordered_piece[0])
            h_y = (ordered_piece[1], ordered_piece[1])
            v_x = (ordered_piece[0], ordered_piece[0])
            v_y = (0, piece_to_cut[1])

        plt.plot(h_x, h_y, color='black', linestyle="--")
        plt.plot(v_x, v_y, color='black', linestyle="--")
```

Updating the CoreEnv

To make use of the renderer we simple have to instantiate it in the constructor of the CoreEnv and make it accessible via the *get_renderer()* method.

Listing 7: env/core_env.py

```
from .renderer import Cutting2DRenderer
...


class Cutting2DCoreEnvironment(CoreEnv):

    def __init__(self, max_pieces_in_inventory: int, raw_piece_size: (int, int),␣
↪static_demand: (int, int)):
        super().__init__()

        # initialize rendering
        self.renderer = Cutting2DRenderer()
        ...

    def get_renderer(self) -> Cutting2DRenderer:
        """Cutting 2D renderer module."""
        return self.renderer
```

Test Script

The following snippet will instantiate the environment and run it for 15 steps.

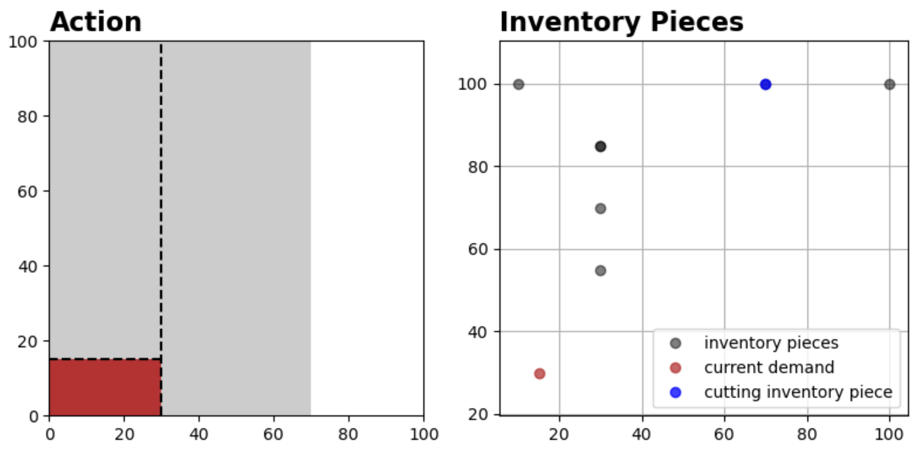
Listing 8: main.py

```python
""" Test script CoreEnv """
from tutorials.tutorial_maze_env.part02_renderer.env.core_env import␣
↪Cutting2DCoreEnvironment
from tutorials.tutorial_maze_env.part02_renderer.env.maze_action import␣
↪Cutting2DMazeAction


def main():
    # init and reset core environment
    core_env = Cutting2DCoreEnvironment(max_pieces_in_inventory=200, raw_piece_
↪size=[100, 100],
                                        static_demand=(30, 15))
    maze_state = core_env.reset()
    # run interaction loop
    for i in range(15):
        # create cutting maze_action
        maze_action = Cutting2DMazeAction(piece_id=0, rotate=False, reverse_cutting_
↪order=False)

        # render current state along with next maze_action
        core_env.renderer.render(maze_state, maze_action, None)

        # take actual environment step
        maze_state, reward, done, info = core_env.step(maze_action)
        print(f"reward {reward} | done {done} | info {info}")


if __name__ == "__main__":
    """ main """
    main()
```

When running the script you should get the following command line output:

```
reward -1 | done False | info {'msg': 'valid_cut'}
reward 0 | done False | info {'msg': 'valid_cut'}
reward 0 | done False | info {'msg': 'valid_cut'}
...
```

and a rendering of the current MazeState and MazeAction in each time step similar to the image shown below:



The dashed line represents the cutting configuration specified with the MazeAction.

1.3.4 Implementing the MazeEnv

The complete code for this part of the tutorial can be found here

```
# file structure
- cutting_2d
    - main.py  # modified
    - env
        - core_env.py  # modified
        - inventory.py
        - maze_state.py
        - maze_action.py
        - renderer.py
        - maze_env.py  # new
    - space_interfaces
        - dict_action_conversion.py  # new
        - dict_observation_conversion.py  # new
```

Page Overview

- *MazeEnv*
- *ObservationConversionInterface*
- *ActionConversionInterface*
- *Updating the CoreEnv*
- *Test Script*

MazeEnv

The *MazeEnv* wraps the CoreEnvs as a Gym-style environment in a reusable form, by utilizing the *interfaces (mappings)* from the MazeState to the observation and from the MazeAction to the action. After implementing the MazeEnv we will be ready to perform our first training run. To learn more about the usability and advantages of this concept you can follow up on *Customizing Core and Maze Envs*.

In the remainder of this part of the tutorial we will implement the `Cutting2DEnvironment` (MazeEnv) as well as a *corresponding set of interfaces*.

Listing 9: env/maze_env.py

```python
from maze.core.env.core_env import CoreEnv
from maze.core.env.maze_env import MazeEnv
from maze.core.env.action_conversion import ActionConversionInterface
from maze.core.env.observation_conversion import ObservationConversionInterface

from .core_env import Cutting2DCoreEnvironment
from ..space_interfaces.dict_observation_conversion import ObservationConversion
from ..space_interfaces.dict_action_conversion import ActionConversion


class Cutting2DEnvironment(MazeEnv[Cutting2DCoreEnvironment]):
    """Maze environment for 2d cutting.

    :param core_env: The underlying core environment.
    :param action_conversion: A action conversion interfaces.
    :param observation_conversion: An observation conversion interface.
    """

    def __init__(self,
                 core_env: CoreEnv,
                 action_conversion: ActionConversionInterface,
                 observation_conversion: ObservationConversionInterface):
        super().__init__(core_env=core_env,
                         action_conversion_dict={0: action_conversion},
                         observation_conversion_dict={0: observation_conversion})


def maze_env_factory(max_pieces_in_inventory: int, raw_piece_size: (int, int),
                     static_demand: (int, int)) -> Cutting2DEnvironment:
    """Convenience factory function that compiles a trainable maze environment.
    (for argument details see: Cutting2DCoreEnvironment)
    """

    # init core environment
    core_env = Cutting2DCoreEnvironment(max_pieces_in_inventory=max_pieces_in_
↪inventory,
                                        raw_piece_size=raw_piece_size,
                                        static_demand=static_demand)

    # init maze environment including observation and action interfaces
    action_conversion = ActionConversion(max_pieces_in_inventory=max_pieces_in_
↪inventory)
    observation_conversion = ObservationConversion(raw_piece_size=raw_piece_size,
                                                   max_pieces_in_inventory=max_pieces_
↪in_inventory)
    return Cutting2DEnvironment(core_env, action_conversion, observation_conversion)
```

The MazeEnv is instantiated with the underlying CoreEnv and the two interfaces for MazeStates and MazeActions. For convenience we also add a `maze_env_factory` to instantiate the MazeEnv from the original environment parameter set. This will be useful in the next part of the tutorial where we will train an agent based on this environment.

ObservationConversionInterface

The *ObservationConversionInterface* converts CoreEnv MazeState objects into machine readable Gym-style observations and defines the respective Gym observation space. In the present cases the observation is defined as a dictionary with the following structure:

- *inventory*: 2d array representing all pieces currently in inventory
- *inventory_size*: count of pieces currently in inventory
- *order*: 2d vector representing the customer order (current demand)

Listing 10: space_interfaces/dict_observation_conversion.py

```
import numpy as np
from typing import Dict
from gym import spaces

from maze.core.annotations import override
from maze.core.env.observation_conversion import ObservationConversionInterface
from ..env.maze_state import Cutting2DMazeState


class ObservationConversion(ObservationConversionInterface):
    """Cutting 2d environment state to dictionary observation.

    :param max_pieces_in_inventory: Size of the inventory. If inventory gets full,␣
↪the oldest pieces get discarded.
    :param raw_piece_size: Size of a fresh raw (= full-size) piece
    """

    def __init__(self, raw_piece_size: (int, int), max_pieces_in_inventory: int):
        self.max_pieces_in_inventory = max_pieces_in_inventory
        self.raw_piece_size = raw_piece_size

    @override(ObservationConversionInterface)
    def maze_to_space(self, maze_state: Cutting2DMazeState) -> Dict[str, np.ndarray]:
        """Converts core environment state to a machine readable agent observation."""

        # Convert inventory to numpy array and stretch it to full size (filling with␣
↪zeros)
        inventory_state = maze_state.inventory
        inventory_state += [(0, 0)] * (self.max_pieces_in_inventory - len(maze_state.
↪inventory))

        # Compile dict space observation
        return {'inventory': np.asarray(inventory_state, dtype=np.float32),
                'inventory_size': np.asarray([len(maze_state.inventory)], dtype=np.
↪float32),
                'ordered_piece': np.asarray(maze_state.current_demand, dtype=np.
↪float32)}

    @override(ObservationConversionInterface)
    def space_to_maze(self, observation: Dict[str, np.ndarray]) -> Cutting2DMazeState:
```

(continues on next page)

(continued from previous page)

```
        """Converts agent observation to core environment state (not required for␣
↪this example)."""
        raise NotImplementedError

    @override(ObservationConversionInterface)
    def space(self) -> spaces.Dict:
        """Return the Gym dict observation space based on the given params.

        :return: Gym space object
            - inventory: max_pieces_in_inventory x 2 (x/y-dimensions of pieces in␣
↪inventory)
            - inventory_size: scalar number of pieces in inventory
            - ordered_piece: 2d vector holding x/y-dimension of customer ordered piece
        """
        return spaces.Dict({
            'inventory': spaces.Box(low=np.zeros((self.max_pieces_in_inventory, 2),␣
↪dtype=np.float32),
                                    high=np.vstack([[self.raw_piece_size[0] + 1, self.
↪raw_piece_size[1] + 1]] *
                                                   self.max_pieces_in_inventory).
↪astype(np.float32),
                                    dtype=np.float32),
            'inventory_size': spaces.Box(low=np.float32(0), high=self.max_pieces_in_
↪inventory + 1,
                                         shape=(1,), dtype=np.float32),
            'ordered_piece': spaces.Box(low=np.float32(0), high=np.float32(max(self.
↪raw_piece_size) + 1),
                                         shape=(2,), dtype=np.float32)
        })
```

ActionConversionInterface

The *ActionConversionInterface* converts agent actions into CoreEnv MazeAction objects and defines the respective Gym action space. In the present cases the action is defined as a dictionary with the following structure:

- *piece_idx*: id of the inventory piece that should be used for cutting
- *rotation*: defines whether to rotate the piece for cutting or not
- *order*: defines the cutting order (*xy* vs. *yx*)

Listing 11: space_interfaces/dict_action_conversion.py

```
from typing import Dict
from gym import spaces
from maze.core.env.action_conversion import ActionConversionInterface

from ..env.maze_action import Cutting2DMazeAction
from ..env.maze_state import Cutting2DMazeState


class ActionConversion(ActionConversionInterface):
    """Converts agent actions to actual environment maze_actions.

    :param max_pieces_in_inventory: Size of the inventory
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, max_pieces_in_inventory: int):
        self.max_pieces_in_inventory = max_pieces_in_inventory

    def space_to_maze(self, action: Dict[str, int], maze_state: Cutting2DMazeState) ->
↪ Cutting2DMazeAction:
        """Converts agent dictionary action to environment MazeAction object."""
        return Cutting2DMazeAction(piece_id=action["piece_idx"],
                                   rotate=bool(action["cut_rotation"]),
                                   reverse_cutting_order=bool(action["cut_order"]))

    def maze_to_space(self, maze_action: Cutting2DMazeAction) -> Dict[str, int]:
        """Converts environment MazeAction object to agent dictionary action."""
        return {"piece_idx": maze_action.piece_id,
                "cut_rotation": int(maze_action.rotate),
                "cut_order": int(maze_action.reverse_cutting_order)}

    def space(self) -> spaces.Dict:
        """Returns Gym dict action space."""
        return spaces.Dict({
            "piece_idx": spaces.Discrete(self.max_pieces_in_inventory),  # Which␣
↪piece should be cut
            "cut_rotation": spaces.Discrete(2),  # Rotate: (yes / no)
            "cut_order": spaces.Discrete(2)      # Cutting order: (xy / yx)
        })
```

Updating the CoreEnv

For the sake of completeness we also show two more minor modifications required in the CoreEnv, which are not too important for this tutorial at the moment. In short, the *StructuredEnv* interface supports interaction patterns beyond standard Gym environments to model for example hierarchical or multi-agent RL problems. We will get back to this in our more advanced tutorials.

The code below defines that the current version of the environment requires only **one actor** (id 0) with a **single policy** (id 0) that is **never done**.

Listing 12: env/core_env.py

```
class Cutting2DCoreEnvironment(CoreEnv):

    ...

    def is_actor_done(self) -> bool:
        """Returns True if the just stepped actor is done, which is different to the␣
↪done flag of the environment."""
        return False

    def actor_id(self) -> Tuple[Union[str, int], int]:
        """Returns the currently executed actor along with the policy id. The id is␣
↪unique only with
        respect to the policies (every policy has its own actor 0).
        Note that identities of done actors can not be reused in the same rollout.

        :return: The current actor, as tuple (policy id, actor number).
        """
```

(continues on next page)

(continued from previous page)

```
        return 0, 0

    ...
```

Test Script

The following snippet will instantiate the environment and run it for 15 steps.

Note that (compared to the *previous example*) we are now:

- working with observations and actions instead of MazeStates and MazeActions
- able to sample actions from the action_space object

Listing 13: main.py

```
""" Test script CoreEnv """
from tutorials.tutorial_maze_env.part03_maze_env.env.maze_env import maze_env_factory


def main():
    # init maze environment including observation and action interfaces
    env = maze_env_factory(max_pieces_in_inventory=10,
                           raw_piece_size=[100, 100],
                           static_demand=(30, 15))

    # reset environment
    obs = env.reset()
    # run interaction loop
    for i in range(15):
        # sample random action
        action = env.action_space.sample()

        # take actual environment step
        obs, reward, done, info = env.step(action)
        print(f"reward {reward} | done {done} | info {info}")


if __name__ == "__main__":
    """ main """
    main()
```

```
reward -1 | done False | info {'msg': 'valid_cut'}
reward 0 | done False | info {'msg': 'valid_cut'}
reward 0 | done False | info {'msg': 'valid_cut'}
reward 0 | done False | info {'error': 'piece_id_out_of_bounds'}
reward 0 | done False | info {'error': 'piece_id_out_of_bounds'}
...
```

1.3.5 Training the MazeEnv

The complete code for this part of the tutorial can be found here

```
# file structure
- cutting_2d
    - conf
        - env
            - tutorial_cutting_2d_basic.yaml
        - model
            - tutorial_cutting_2d_basic.yaml
        - wrappers
            - tutorial_cutting_2d_basic.yaml
```

Page Overview

- *Hydra Configuration*
- *Training an Agent*

Hydra Configuration

The entire Maze workflow is boosted by the *Hydra configuration system*. To be able to perform our first training run via the Maze CLI we have to add a few more config files. Going into the very details of the config structure is for now beyond the scope of this tutorial. However, we still provide some information on the parts relevant for this example.

The config file for the `maze_env_factory` looks as follows:

Listing 14: conf/env/tutorial_cutting_2d_basic.yaml

```yaml
# @package env
type: tutorials.tutorial_maze_env.part03_maze_env.env.maze_env.maze_env_factory

# parametrizes the core environment
max_pieces_in_inventory: 200
raw_piece_size: [100, 100]
static_demand: [30, 15]
```

Additionally, we also provide a wrapper config but refer to *Customizing Environments with Wrappers* for details.

Listing 15: conf/wrappers/tutorial_cutting_2d_basic.yaml

```yaml
# @package wrappers

# limits the maximum number of time steps of an episode
TimeLimitWrapper:
  max_episode_steps: 200

# flattens the dictionary observations to work with DenseLayers
PreProcessingWrapper:
    pre_processor_mapping:
        - observation: inventory
          type: maze.preprocessors.FlattenPreProcessor
          keep_original: false
```

(continues on next page)

(continued from previous page)

```
          config:
            num_flatten_dims: 2
```

To learn more about the model config in `conf/env_model/tutorial_cutting_2d_basic.yaml` you can visit the *introduction on how to work with template models*.

Training an Agent

Once the config is set up we are good to go to start our first training run (in the cmd below with the PPO algorithm):

```
maze-run -cn conf_train env=tutorial_cutting_2d_basic wrappers=tutorial_cutting_2d_
↪basic \
model=tutorial_cutting_2d_basic algorithm=ppo
```

Running the trainer should print a command line output similar to the one shown below.

```
 step|path                                                                           |  ␣
↪             value
=====|===========================================================================|===================
   12|train     MultiStepActorCritic..time_epoch           ······················|  ␣
↪            24.333
   12|train     MultiStepActorCritic..time_rollout         ······················|  ␣
↪             0.754
   12|train     MultiStepActorCritic..learning_rate        ······················|  ␣
↪             0.000
   12|train     MultiStepActorCritic..policy_loss          0                     |  ␣
↪            -0.016
   12|train     MultiStepActorCritic..policy_grad_norm     0                     |  ␣
↪             0.015
   12|train     MultiStepActorCritic..policy_entropy       0                     |  ␣
↪             0.686
   12|train     MultiStepActorCritic..critic_value         0                     |  ␣
↪           -56.659
   12|train     MultiStepActorCritic..critic_value_loss    0                     |  ␣
↪            33.026
   12|train     MultiStepActorCritic..critic_grad_norm     0                     |  ␣
↪             0.500
   12|train     MultiStepActorCritic..time_update          ······················|  ␣
↪             1.205
   12|train     DiscreteActionEvents  action               substep_0/order       |  ␣
↪[len:8000, :0.5]
   12|train     DiscreteActionEvents  action               substep_0/piece_idx   |␣
↪[len:8000, :169.2]
   12|train     DiscreteActionEvents  action               substep_0/rotation    |  ␣
↪[len:8000, :1.0]
   12|train     BaseEnvEvents         reward               median_step_count     |  ␣
↪           200.000
   12|train     BaseEnvEvents         reward               mean_step_count       |  ␣
↪           200.000
   12|train     BaseEnvEvents         reward               total_step_count      |  ␣
↪         96000.000
   12|train     BaseEnvEvents         reward               total_episode_count   |  ␣
↪           480.000
   12|train     BaseEnvEvents         reward               episode_count         |  ␣
↪            40.000
```

(continues on next page)

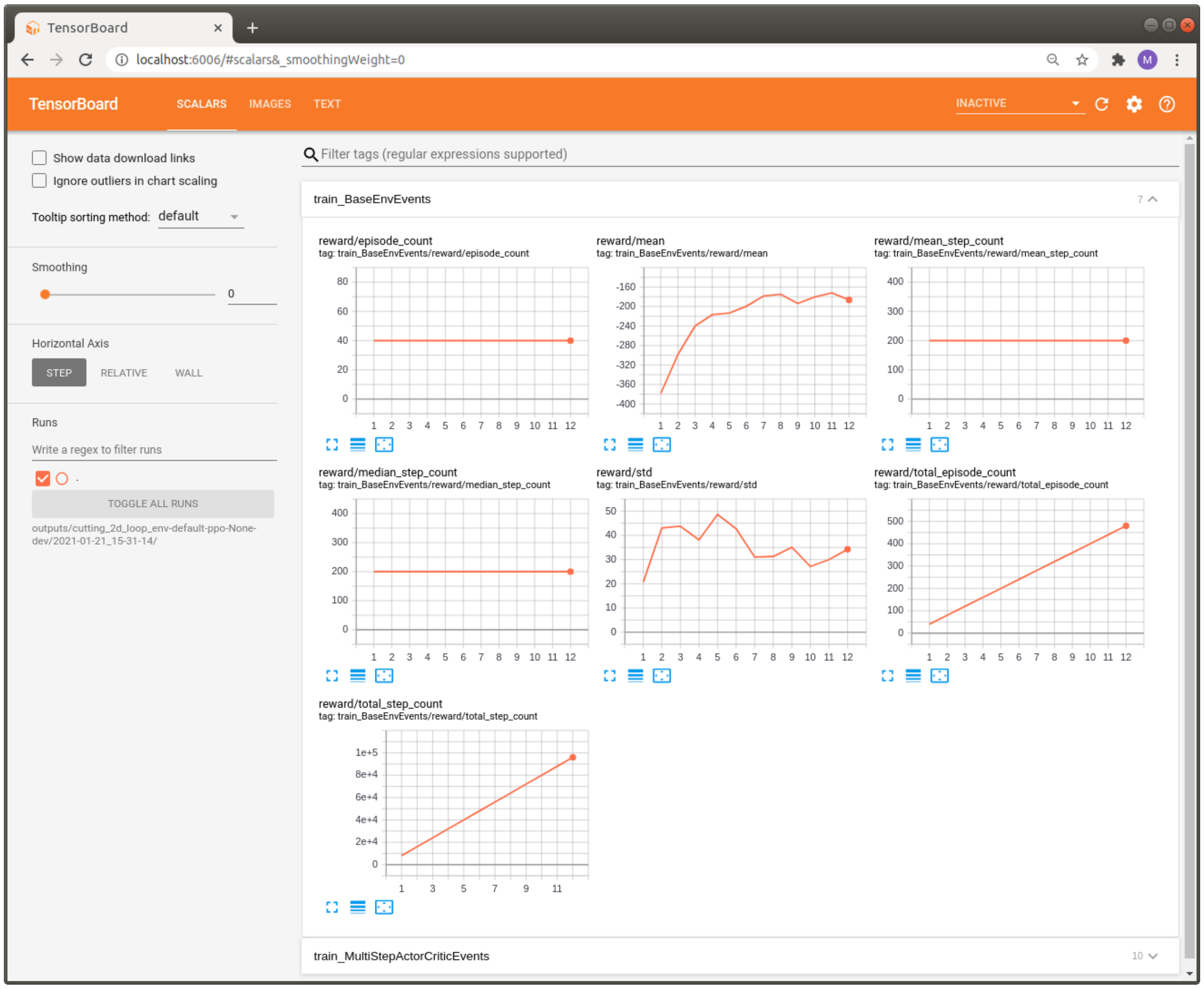
(continued from previous page)

```
   12|train      BaseEnvEvents          reward             std                  |  ␣
↪           34.248
   12|train      BaseEnvEvents          reward             mean                 |  ␣
↪         -186.450
   12|train      BaseEnvEvents          reward             min                  |  ␣
↪         -259.000
   12|train      BaseEnvEvents          reward             max                  |  ␣
↪         -130.000
```

To get a nicer view on these numbers we can also *take a look at the stats with Tensorboard*.

```
tensorboard --logdir outputs
```

You can view it with your browser at http://localhost:6006/.


localhost:6006/#scalars&_smoothingWeight=0

Show data download links

Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0

Horizontal Axis: STEP

outputs/cutting_2d_loop_env-default-ppo-None-dev/2021-01-21_15-31-14/

TensorBoard
SCALARS
IMAGES
TEXT
INACTIVE
Filter tags (regular expressions supported)
RELATIVE
WALL
Runs
Write a regex to filter runs
TOGGLE ALL RUNS
train_BaseEnvEvents
reward/episode_count
tag: train_BaseEnvEvents/reward/episode_count
reward/mean
tag: train_BaseEnvEvents/reward/mean
reward/mean_step_count
tag: train_BaseEnvEvents/reward/mean_step_count
reward/median_step_count
tag: train_BaseEnvEvents/reward/median_step_count
reward/std
tag: train_BaseEnvEvents/reward/std
reward/total_episode_count
tag: train_BaseEnvEvents/reward/total_episode_count
reward/total_step_count
tag: train_BaseEnvEvents/reward/total_step_count
train_MultiStepActorCriticEvents

For now we can only inspect standard metrics such as *reward statistics* or *mean_step_counts* per episode. Unfortunately, this is not too informative with respect to the cutting problem we are currently addressing. In the next part we will show how to make logging much more informative by *introducing events and KPIs*.

1.3.6 Adding Events and KPIs

The complete code for this part of the tutorial can be found here

```
# file structure
- cutting_2d
    - main.py  # modified
    - env
        - core_env.py  # modified
        - inventory.py  # modified
        - maze_state.py
        - maze_action.py
        - renderer.py
        - maze_env.py
        - events.py  # new
        - kpi_calculator.py  # new
    - space_interfaces
        - dict_action_conversion.py
        - dict_observation_conversion.py
```

Page Overview

- *Events*
- *KPI Calculator*
- *Updating CoreEnv and Inventory*
- *Test Script*

Events

In the *previous section* we have trained the initial version of our cutting environment and already learned how we can watch the training process with *commandline and Tensorboard logging*. However, watching only standard metrics such as *reward* or *episode step count* is not always too informative with respect to the agents behaviour and the problem at hand.

For example we might be interested in how often an agent selects an invalid cutting piece or specifies and invalid cutting setting. To tackle this issue and to enable better inspection and logging tools we introduce an *event system* that will be also reused in the *reward customization section* of this tutorial.

In particular, we introduce two event types related to the cutting process as well as inventory management. For each event we can define which statistics are computed at which stage of the aggregation process (*event*, *step*, *epoch*) via event decorators:

- `@define_step_stats(len)`: Events e_i are collected as a list of events $\{e_i\}$. The `len` function counts how often such an event occurred in the current environment step $Stats_{Step} = |\{e_i\}|$.
- `@define_episode_stats(sum)`: Defines how the S step statistics should be aggregated to episode statistics by simply summing them up: $Stats_{Episode} = \sum^S Stats_{Step}$
- `@define_epoch_stats(np.mean, output_name="mean_episode_total")`: A training epoch consists of N episodes. This decorator defines that epoch statistics should be the average of the contained episodes: $Stats_{Epoch} = (\sum^N Stats_{Episode})/N$

Below we will see that theses statistics will now be considered by the logging system as *InventoryEvents* and *CuttingEvents*. For more details on event decorators and the underlying working principles we refer to the dedicated section on *event and KPI logging*.

Listing 16: env/events.py

```
from abc import ABC

import numpy as np
from maze.core.log_stats.event_decorators import define_step_stats, define_episode_
↪stats, define_epoch_stats


class CuttingEvents(ABC):
    """Events related to the cutting process."""

    @define_epoch_stats(np.mean, output_name="mean_episode_total")
    @define_episode_stats(sum)
    @define_step_stats(len)
    def invalid_piece_selected(self):
        """An invalid piece is selected for cutting."""

    @define_epoch_stats(np.mean, output_name="mean_episode_total")
    @define_episode_stats(sum)
    @define_step_stats(len)
    def valid_cut(self, current_demand: (int, int), piece_to_cut: (int, int), raw_
↪piece_size: (int, int),
                  cutting_area: float):
        """A valid cut was performed."""

    @define_epoch_stats(np.mean, output_name="mean_episode_total")
    @define_episode_stats(sum)
    @define_step_stats(len)
    def invalid_cut(self, current_demand: (int, int), piece_to_cut: (int, int), raw_
↪piece_size: (int, int)):
        """Invalid cutting parameters have been specified."""


class InventoryEvents(ABC):
    """Events related to inventory management."""

    @define_epoch_stats(np.mean, output_name="mean_episode_total")
    @define_episode_stats(sum)
    @define_step_stats(len)
    def piece_discarded(self, piece: (int, int)):
        """The inventory is full and a piece has been discarded."""

    @define_epoch_stats(np.mean, input_name="step_mean", output_name="step_mean")
    @define_epoch_stats(max, input_name="step_max", output_name="step_max")
    @define_episode_stats(np.mean, output_name="step_mean")
    @define_episode_stats(max, output_name="step_max")
    @define_step_stats(None)
    def pieces_in_inventory(self, value: int):
        """Reports the count of pieces currently in the inventory."""

    @define_epoch_stats(np.mean, output_name="mean_episode_total")
    @define_episode_stats(sum)
    @define_step_stats(len)
    def piece_replenished(self):
        """A new raw cutting piece has been replenished."""
```

KPI Calculator

The goal of the cutting 2d environment is to learn a cutting policy that requires as little as possible raw inventory pieces for fulfilling upcoming customer demand. This metric is exactly what we *define as the KPI* to watch and optimize, e.g. the **raw_piece_usage_per_step**.

As you will see *below* the logging system considers such KPIs and prints statistics of these along with the remaining *BaseEnvEvents*.

Listing 17: env/kpi_calculator.py

```
from typing import Dict

from maze.core.env.maze_state import MazeStateType
from maze.core.log_events.kpi_calculator import KpiCalculator
from maze.core.log_events.episode_event_log import EpisodeEventLog
from .events import InventoryEvents


class Cutting2dKpiCalculator(KpiCalculator):
    """KPIs for 2D cutting environment.
    The following KPIs are available: Raw pieces used per step
    """

    def calculate_kpis(self, episode_event_log: EpisodeEventLog, last_maze_state:␣
↪MazeStateType) -> Dict[str, float]:
        """Calculates the KPIs at the end of episode."""

        # get overall step count of episode
        step_count = len(episode_event_log.step_event_logs)
        # count raw inventory piece replenishment events
        raw_piece_usage = 0
        for _ in episode_event_log.query_events(InventoryEvents.piece_replenished):
            raw_piece_usage += 1
        # compute step normalized raw piece usage
        return {"raw_piece_usage_per_step": raw_piece_usage / step_count}
```

Updating CoreEnv and Inventory

There are also a few changes we have to make in the CoreEnvironment:

- initialize the Publisher-Subscriber and the KPI Calculator
- creating the event topics for cutting and inventory events when setting up the environment
- instead of writing relevant events into the info dictionary in the step function we can now trigger the respective events.

Listing 18: env/core_env.py

```
...
from maze.core.events.pubsub import Pubsub
from .events import CuttingEvents, InventoryEvents
from .kpi_calculator import Cutting2dKpiCalculator


class Cutting2DCoreEnvironment(CoreEnv):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, max_pieces_in_inventory: int, raw_piece_size: (int, int),␣
↪static_demand: (int, int)):
        super().__init__()

        ...

        # init pubsub for event to reward routing
        self.pubsub = Pubsub(self.context.event_service)

        # KPIs calculation
        self.kpi_calculator = Cutting2dKpiCalculator()

    def _setup_env(self):
        """Setup environment."""
        inventory_events = self.pubsub.create_event_topic(InventoryEvents)
        self.inventory = Inventory(self.max_pieces_in_inventory, self.raw_piece_size,␣
↪inventory_events)
        self.inventory.replenish_piece()

        self.cutting_events = self.pubsub.create_event_topic(CuttingEvents)

    def step(self, maze_action: Cutting2DMazeAction) -> Tuple[Cutting2DMazeState, np.
↪array, bool, Dict[Any, Any]]:
        """Summary of the step (simplified, not necessarily respecting the actual␣
↪order in the code):
        1. Check if the selected piece to cut is valid (i.e. in inventory, large␣
↪enough etc.)
        2. Attempt the cutting
        3. Replenish a fresh piece if needed and return an appropriate reward

        :param maze_action: Cutting MazeAction to take.
        :return: maze_state, reward, done, info
        """

        info, reward = {}, 0
        replenishment_needed = False

        # check if valid piece id was selected
        if maze_action.piece_id >= self.inventory.size():
            self.cutting_events.invalid_piece_selected()
        # perform cutting
        else:
            piece_to_cut = self.inventory.pieces[maze_action.piece_id]

            # attempt the cut
            if self.inventory.cut(maze_action, self.current_demand):
                self.cutting_events.valid_cut(current_demand=self.current_demand,␣
↪piece_to_cut=piece_to_cut,
                                              raw_piece_size=self.raw_piece_size)
                replenishment_needed = piece_to_cut == self.raw_piece_size
            else:
                # assign a negative reward for invalid cutting attempts
                self.cutting_events.invalid_cut(current_demand=self.current_demand,␣
↪piece_to_cut=piece_to_cut,
                                                raw_piece_size=self.raw_piece_size)
                reward = -2
```

(continues on next page)

(continued from previous page)

```
        # check if replenishment is required
        if replenishment_needed:
            self.inventory.replenish_piece()
            # assign negative reward if a piece has to be replenished
            reward = -1

        # step execution finished, write step statistics
        self.inventory.log_step_statistics()

        # compile env state
        maze_state = self.get_maze_state()

        return maze_state, reward, False, info

    def get_kpi_calculator(self) -> Cutting2dKpiCalculator:
        """KPIs are supported."""
        return self.kpi_calculator
```

For the inventory we proceed analogously and also trigger the respective events.

Listing 19: env/inventory.py

```
...
from .events import InventoryEvents


class Inventory:
    """Holds the inventory of 2D pieces and performs cutting.
    :param max_pieces_in_inventory: Size of the inventory. If full, the oldest pieces␣
↪get discarded.
    :param raw_piece_size: Size of a fresh raw (= full-size) piece.
    :param inventory_events: Inventory event dispatch proxy.
    """

    def __init__(self, max_pieces_in_inventory: int, raw_piece_size: (int, int),
                 inventory_events: InventoryEvents):
        ...

        self.inventory_events = inventory_events

    def store_piece(self, piece: (int, int)) -> None:
        """Store the given piece.
        :param piece: Piece to store.
        """
        # If we would run out of storage space, discard the oldest piece first
        if self.is_full():
            self.pieces.pop(0)
            self.inventory_events.piece_discarded(piece=piece)

        self.pieces.append(piece)

    def replenish_piece(self) -> None:
        """Add a fresh raw piece to inventory."""
        self.store_piece(self.raw_piece_size)
        self.inventory_events.piece_replenished()
```

(continues on next page)

(continued from previous page)

```
    def log_step_statistics(self):
        """Log inventory statistics once per step"""
        self.inventory_events.pieces_in_inventory(self.size())
```

Test Script

The following snippet will instantiate the environment and run it for 15 steps.

To get access to event and KPI logging we need to wrap the environment with the *LogStatsWrapper*. To simplify the statistics logging setup we rely on the *SimpleStatsLoggingSetup* helper class.

Listing 20: main.py

```
""" Test script CoreEnv """
from maze.utils.log_stats_utils import SimpleStatsLoggingSetup
from maze.core.wrappers.log_stats_wrapper import LogStatsWrapper
from tutorials.tutorial_maze_env.part04_events.env.maze_env import maze_env_factory


def main():
    # init maze environment including observation and action interfaces
    env = maze_env_factory(max_pieces_in_inventory=200,
                           raw_piece_size=[100, 100],
                           static_demand=(30, 15))

    # wrap environment with logging wrapper
    env = LogStatsWrapper(env, logging_prefix="main")

    # register a console writer and connect the writer to the statistics logging␣
↪system
    with SimpleStatsLoggingSetup(env):
        # reset environment
        obs = env.reset()
        # run interaction loop
        for i in range(15):
            # sample random action
            action = env.action_space.sample()

            # take actual environment step
            obs, reward, done, info = env.step(action)


if __name__ == "__main__":
    """ main """
    main()
```

When running the script you will get an output as shown below. Note that statistics of both, events and KPIs, are printed along with default *reward* or *action* statistics.

```
 step|path                                                                    |    ␣
↪           value
=====|========================================================================|====================
    1|main    DiscreteActionEvents  action                substep_0/order      |    ␣
↪[len:15, :0.5]
    1|main    DiscreteActionEvents  action                substep_0/piece_idx  |   ␣
↪[len:15, :82.3]
```

(continues on next page)

(continued from previous page)

```
    1|main    DiscreteActionEvents  action                substep_0/rotation    |     ␣
↪[len:15, :0.7]
    1|main    BaseEnvEvents         reward                median_step_count     |     ␣
↪          15.000
    1|main    BaseEnvEvents         reward                mean_step_count       |     ␣
↪          15.000
    1|main    BaseEnvEvents         reward                total_step_count      |     ␣
↪          15.000
    1|main    BaseEnvEvents         reward                total_episode_count   |     ␣
↪           1.000
    1|main    BaseEnvEvents         reward                episode_count         |     ␣
↪           1.000
    1|main    BaseEnvEvents         reward                std                   |     ␣
↪           0.000
    1|main    BaseEnvEvents         reward                mean                  |     ␣
↪         -29.000
    1|main    BaseEnvEvents         reward                min                   |     ␣
↪         -29.000
    1|main    BaseEnvEvents         reward                max                   |     ␣
↪         -29.000
    1|main    InventoryEvents       piece_replenished     mean_episode_total    |     ␣
↪           3.000
    1|main    InventoryEvents       pieces_in_inventory   step_max              |     ␣
↪         200.000
    1|main    InventoryEvents       pieces_in_inventory   step_mean             |     ␣
↪         200.000
    1|main    CuttingEvents         invalid_cut           mean_episode_total    |     ␣
↪          14.000
    1|main    InventoryEvents       piece_discarded       mean_episode_total    |     ␣
↪           2.000
    1|main    CuttingEvents         valid_cut             mean_episode_total    |     ␣
↪           1.000
    1|main    BaseEnvEvents         kpi                   max/raw_piece_usage_..|     ␣
↪           0.000
    1|main    BaseEnvEvents         kpi                   min/raw_piece_usage_..|     ␣
↪           0.000
    1|main    BaseEnvEvents         kpi                   std/raw_piece_usage_..|     ␣
↪           0.000
    1|main    BaseEnvEvents         kpi                   mean/raw_piece_usage..|     ␣
↪           0.000
```

1.3.7 Training with Events and KPIs

The complete code for this part of the tutorial can be found here

```
# file structure
- cutting_2d
    - conf
        - env
            - tutorial_cutting_2d_events.yaml
        - model
            - tutorial_cutting_2d_events.yaml
        - wrappers
            - tutorial_cutting_2d_events.yaml
```

Page Overview

- *Hydra Configuration*
- *Training an Agent*

Hydra Configuration

The entire structure of this example is identical to the one on *training the MazeEnv*. Everything regarding the event systems was already changed in the *section on adding events and KPIs* and the trainers will consider these changes implicitly.

Training an Agent

To retrain the agent on the environment extended with event and KPI logging, run:

```
maze-run -cn conf_train env=tutorial_cutting_2d_events wrappers=tutorial_cutting_2d_
↪events \
model=tutorial_cutting_2d_events algorithm=ppo
```

Running the trainer should print an extended command line output similar to the one shown below. In addition to base events we now also get a statistics log of *CuttingEvents*, *InventoryEvents* and KPIs.

```
 step|path                                                                       |  ␣
↪              value
=====|===========================================================================|===================
    6|train     MultiStepActorCritic..time_epoch           ......................|  ␣
↪           24.548
    6|train     MultiStepActorCritic..time_rollout         ......................|  ␣
↪            0.762
    6|train     MultiStepActorCritic..learning_rate        ......................|  ␣
↪            0.000
    6|train     MultiStepActorCritic..policy_loss          0                     |  ␣
↪           -0.020
    6|train     MultiStepActorCritic..policy_grad_norm     0                     |  ␣
↪            0.013
    6|train     MultiStepActorCritic..policy_entropy       0                     |  ␣
↪            0.760
    6|train     MultiStepActorCritic..critic_value         0                     |  ␣
↪          -49.238
    6|train     MultiStepActorCritic..critic_value_loss    0                     |  ␣
↪           50.175
    6|train     MultiStepActorCritic..critic_grad_norm     0                     |  ␣
↪            0.500
    6|train     MultiStepActorCritic..time_update          ......................|  ␣
↪            1.210
    6|train     DiscreteActionEvents  action               substep_0/order       |  ␣
↪[len:8000, :0.0]
    6|train     DiscreteActionEvents  action               substep_0/piece_idx   |␣
↪[len:8000, :174.2]
    6|train     DiscreteActionEvents  action               substep_0/rotation    |  ␣
↪[len:8000, :1.0]
    6|train     BaseEnvEvents         reward               median_step_count     |  ␣
↪          200.000
```

(continues on next page)

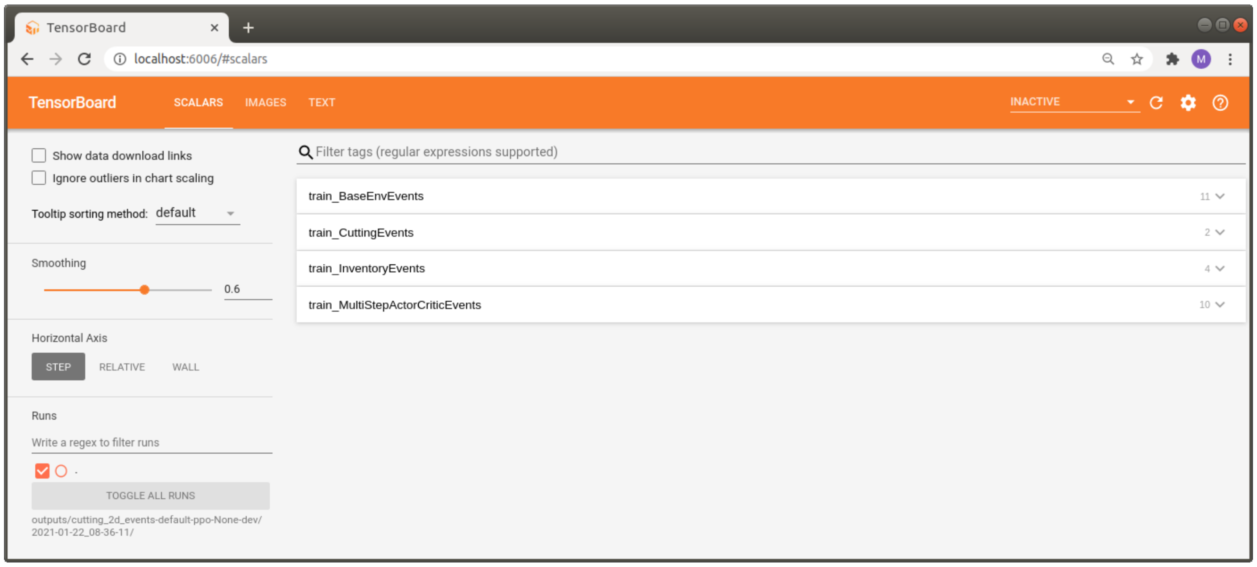
(continued from previous page)

```
    6|train     BaseEnvEvents         reward               mean_step_count      |  ␣
↪           200.000
    6|train     BaseEnvEvents         reward               total_step_count     |  ␣
↪         48000.000
    6|train     BaseEnvEvents         reward               total_episode_count  |  ␣
↪           240.000
    6|train     BaseEnvEvents         reward               episode_count        |  ␣
↪            40.000
    6|train     BaseEnvEvents         reward               std                  |  ␣
↪            38.427
    6|train     BaseEnvEvents         reward               mean                 |  ␣
↪          -182.175
    6|train     BaseEnvEvents         reward               min                  |  ␣
↪          -323.000
    6|train     BaseEnvEvents         reward               max                  |  ␣
↪          -119.000
    6|train     InventoryEvents       piece_replenished    mean_episode_total   |  ␣
↪            15.325
    6|train     InventoryEvents       piece_discarded      mean_episode_total   |  ␣
↪            67.400
    6|train     InventoryEvents       pieces_in_inventory  step_max             |  ␣
↪           200.000
    6|train     InventoryEvents       pieces_in_inventory  step_mean            |  ␣
↪           200.000
    6|train     CuttingEvents         valid_cut            mean_episode_total   |  ␣
↪           116.075
    6|train     CuttingEvents         invalid_cut          mean_episode_total   |  ␣
↪            83.925
    6|train     BaseEnvEvents         kpi                  max/raw_piece_usage_..|  ␣
↪             0.135
    6|train     BaseEnvEvents         kpi                  min/raw_piece_usage_..|  ␣
↪             0.020
    6|train     BaseEnvEvents         kpi                  std/raw_piece_usage_..|  ␣
↪             0.028
    6|train     BaseEnvEvents         kpi                  mean/raw_piece_usage..|  ␣
↪             0.077
```

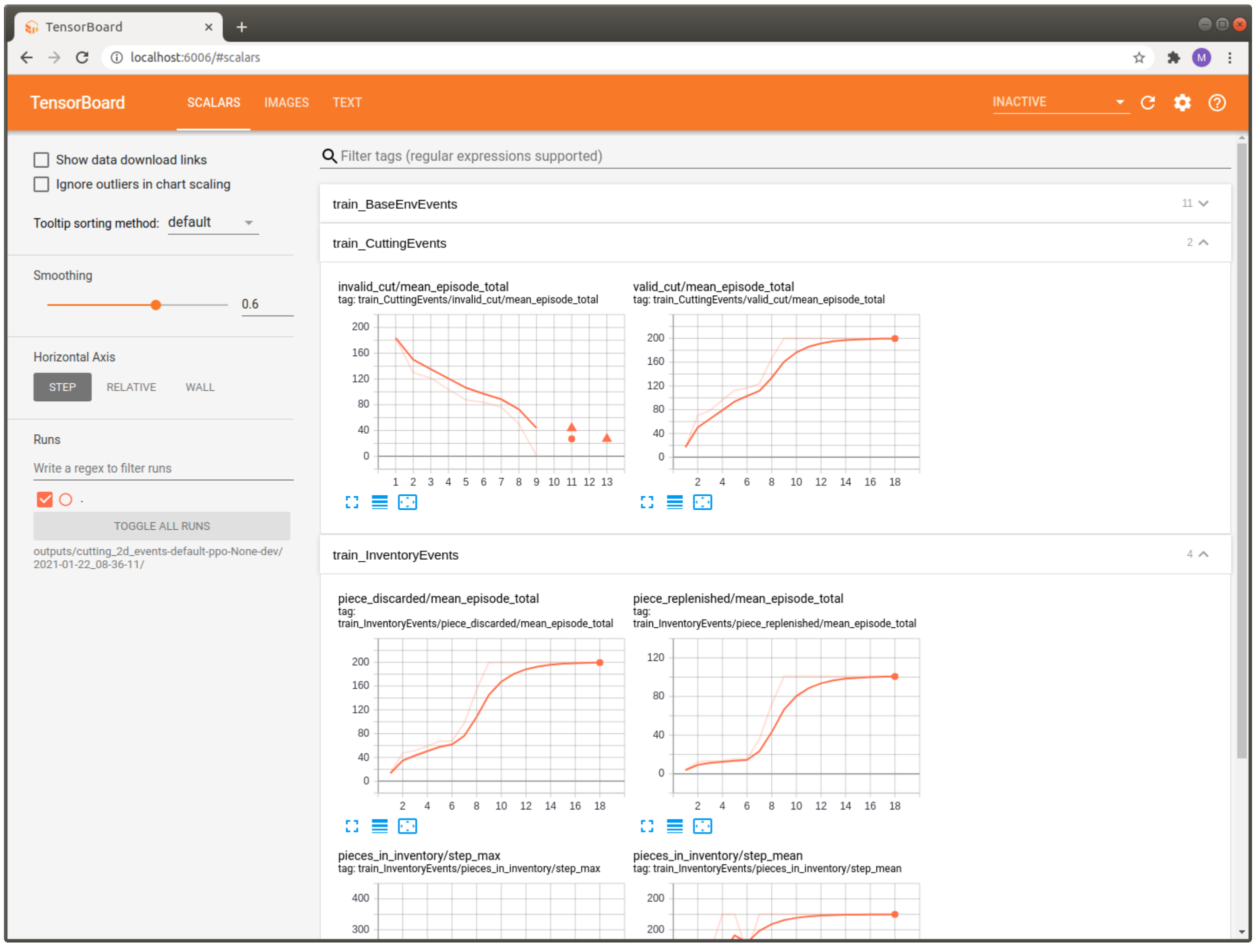
Of course these changes are also reflected in the *Tensorboard log* which you can again view with your browser at http://localhost:6006/.

```
tensorboard --logdir outputs
```

As you can see we now have the two additional sections *train_CuttingEvents* and *train_InventoryEvents* available.


TensorBoard
localhost:6006/#scalars
TensorBoard
SCALARS
IMAGES
TEXT
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
TOGGLE ALL RUNS
outputs/cutting_2d_events-default-ppo-None-dev/2021-01-22_08-36-11/
Filter tags (regular expressions supported)
train_BaseEnvEvents 11
train_CuttingEvents 2
train_InventoryEvents 4
train_MultiStepActorCriticEvents 10

A closer look at these events reveals that the agent actually starts to learn something meaning full as the number of invalid cuts decreases which of course implies that the number of valid cuts increases and we are able to full fill the current customer demand.


TensorBoard
localhost:6006/#scalars
TensorBoard
SCALARS
IMAGES
TEXT
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
TOGGLE ALL RUNS
outputs/cutting_2d_events-default-ppo-None-dev/2021-01-22_08-36-11/
Filter tags (regular expressions supported)
train_BaseEnvEvents 11
train_CuttingEvents 2
invalid_cut/mean_episode_total
tag: train_CuttingEvents/invalid_cut/mean_episode_total
valid_cut/mean_episode_total
tag: train_CuttingEvents/valid_cut/mean_episode_total
train_InventoryEvents 4
piece_discarded/mean_episode_total
tag: train_InventoryEvents/piece_discarded/mean_episode_total
piece_replenished/mean_episode_total
tag: train_InventoryEvents/piece_replenished/mean_episode_total
pieces_in_inventory/step_max
tag: train_InventoryEvents/pieces_in_inventory/step_max
pieces_in_inventory/step_mean
tag: train_InventoryEvents/pieces_in_inventory/step_mean

1.3.8 Adding Reward Customization

The complete code for this part of the tutorial can be found here

```
# file structure
- cutting_2d
    - main.py  # modified
    - env
        - core_env.py  # modified
        - inventory.py
        - maze_state.py
        - maze_action.py
        - renderer.py
        - maze_env.py  # modified
        - events.py
        - kpi_calculator.py
    - space_interfaces
        - dict_action_conversion.py
        - dict_observation_conversion.py
    - reward
        - default_reward.py  # new
```

Page Overview

- *Reward*
- *Updating the Core- and MazeEnv*
- *Where to Go Next*

Reward

In this part of the tutorial we introduce how to reuse the event system for *reward shaping and customization* via the `RewardAggregatorInterface`.

The `DefaultRewardAggregator` does the following:

- Requests the required event interfaces via `get_interfaces` (here *CuttingEvents* and *InventoryEvents*).
- Collects rewards and penalties according to relevant events.
- Aggregates the individual event rewards and penalties to a single scalar reward signal.

Note that this reward aggregator can have any form as long as it provides a scalar reward function that can be used for training. This gives a lot of flexibility in shaping rewards without the need to change the actual implementation of the environment (*more on this topic*).

Listing 21: reward/default_reward.py

```python
from abc import abstractmethod
from typing import List

from maze.core.env.reward import RewardAggregatorInterface

from ..env.events import CuttingEvents, InventoryEvents


```

(continues on next page)

(continued from previous page)

```
class CuttingRewardAggregator(RewardAggregatorInterface):
    """Interface for cutting reward aggregators."""

    @abstractmethod
    def collect_rewards(self) -> List[float]:
        """Assign rewards and penalties according to respective events.
        :return: List of individual event rewards.
        """


class DefaultRewardAggregator(CuttingRewardAggregator):
    """Default reward scheme for the 2D cutting env.

    :param invalid_action_penalty: Negative reward assigned for an invalid cutting␣
↪specification.
    :param raw_piece_usage_penalty: Negative reward assigned for starting a new raw␣
↪inventory piece.
    """

    def __init__(self, invalid_action_penalty: float, raw_piece_usage_penalty: float):
        super().__init__()
        self.invalid_action_penalty = invalid_action_penalty
        self.raw_piece_usage_penalty = raw_piece_usage_penalty

    def get_interfaces(self):
        """Specification of the event interfaces this subscriber wants to receive␣
↪events from.
        Every subscriber must implement this configuration method.
        :return: A list of interface classes"""
        return [CuttingEvents, InventoryEvents]

    def collect_rewards(self) -> List[float]:
        """Assign rewards and penalties according to respective events.
        :return: List of individual event rewards.
        """

        rewards: List[float] = []

        # penalty for starting a new raw inventory piece
        for _ in self.query_events(InventoryEvents.piece_replenished):
            rewards.append(self.raw_piece_usage_penalty)

        # penalty for selecting an invalid piece for cutting
        for _ in self.query_events(CuttingEvents.invalid_piece_selected):
            rewards.append(self.invalid_action_penalty)

        # penalty for specifying invalid cutting parameters
        for _ in self.query_events(CuttingEvents.invalid_cut):
            rewards.append(self.invalid_action_penalty)

        return rewards

    @classmethod
    def to_scalar_reward(cls, reward: List[float]) -> float:
        """Aggregate sub-rewards to scalar reward.

        This method is useful for example in a multi-agent setting
```

(continues on next page)

(continued from previous page)

```
        where we could sum over multiple actors to assign a joint reward.

        :param: reward: The aggregated reward (e.g. per-agent reward for multi-agent␣
↪RL settings).
        :return: The scalar reward returned by the environment.
        """
        return sum(reward)
```

Updating the Core- and MazeEnv

We also have to make a few modifications in the `CoreEnv`:

- Initialize the reward aggregator in the constructor.
- Instead of accumulating reward in the if-else branches of the `step` function we summarize it only once at the end. The conversion to a scalar is performed in the `step` function of the *MazeEnv*.

Listing 22: env/core_env.py

```
...
from ..reward.default_reward import CuttingRewardAggregator


class Cutting2DCoreEnvironment(CoreEnv):
    """Environment for cutting 2D pieces based on the customer demand. Works as␣
↪follows:
    ...
    :param reward_aggregator: Either an instantiated aggregator or a configuration␣
↪dictionary.
    """

    def __init__(self, max_pieces_in_inventory: int, raw_piece_size: (int, int),␣
↪static_demand: (int, int),
                 reward_aggregator: CuttingRewardAggregator):
        super().__init__()

        ...

        # init reward and register it with pubsub
        self.reward_aggregator = reward_aggregator
        self.pubsub.register_subscriber(self.reward_aggregator)

    def step(self, maze_action: Cutting2DMazeAction) -> Tuple[Cutting2DMazeState, np.
↪array, bool, Dict[Any, Any]]:
        """Summary of the step (simplified, not necessarily respecting the actual␣
↪order in the code):
        1. Check if the selected piece to cut is valid (i.e. in inventory, large␣
↪enough etc.)
        2. Attempt the cutting
        3. Replenish a fresh piece if needed and return an appropriate reward

        :param maze_action: Cutting maze_action to take.
        :return: state, reward, done, info
        """

        info = {}
```

(continues on next page)

(continued from previous page)

```
        replenishment_needed = False

        # check if valid piece id was selected
        if maze_action.piece_id >= self.inventory.size():
            self.cutting_events.invalid_piece_selected()
        # perform cutting
        else:
            piece_to_cut = self.inventory.pieces[maze_action.piece_id]

            # attempt the cut
            if self.inventory.cut(maze_action, self.current_demand):
                self.cutting_events.valid_cut(current_demand=self.current_demand,␣
↪piece_to_cut=piece_to_cut,
                                              raw_piece_size=self.raw_piece_size)
                replenishment_needed = piece_to_cut == self.raw_piece_size
            else:
                # assign a negative reward for invalid cutting attempts
                self.cutting_events.invalid_cut(current_demand=self.current_demand,␣
↪piece_to_cut=piece_to_cut,
                                                raw_piece_size=self.raw_piece_size)

        # check if replenishment is required
        if replenishment_needed:
            self.inventory.replenish_piece()
            # assign negative reward if a piece has to be replenished

        # step execution finished, write step statistics
        self.inventory.log_step_statistics()

        # aggregate reward from events
        reward = self.reward_aggregator.collect_rewards()

        # compile env state
        maze_state = self.get_maze_state()

        return maze_state, reward, False, info
```

Finally, we update the `maze_env_factory` function for instantiating the trainable `MazeEnv` and we are all set up for training with event based, customized rewards.

Listing 23: env/maze_env.py

```
...


def maze_env_factory(max_pieces_in_inventory: int, raw_piece_size: (int, int),
                     static_demand: (int, int)) -> Cutting2DEnvironment:
    """Convenience factory function that compiles a trainable maze environment.
    (for argument details see: Cutting2DCoreEnvironment)
    """

    # init reward aggregator
    reward_aggregator = DefaultRewardAggregator(invalid_action_penalty=-2, raw_piece_
↪usage_penalty=-1)

    # init core environment
```

(continues on next page)

(continued from previous page)

```
    core_env = Cutting2DCoreEnvironment(max_pieces_in_inventory=max_pieces_in_
↪inventory,
                                        raw_piece_size=raw_piece_size,
                                        static_demand=static_demand,
                                        reward_aggregator=reward_aggregator)

    # init maze environment including observation and action interfaces
    action_conversion = ActionConversion(max_pieces_in_inventory=max_pieces_in_
↪inventory)
    observation_conversion = ObservationConversion(raw_piece_size=raw_piece_size,
                                                   max_pieces_in_inventory=max_pieces_
↪in_inventory)
    return Cutting2DEnvironment(core_env, action_conversion, observation_conversion)
```

Where to Go Next

As the reward is implemented via a reward aggregator that is methodologically identical to the initial version there is no need to retain the model for now. However, we highly recommend to proceed with the more advanced tutorial on *Structured Environments and Action Masking*.

1.4 API Documentation

This page provides an overview of the Maze API documentation

1.4.1 Environment Interfaces

This page contains the reference documentation for environment interfaces.

maze.core.env

Environment interfaces:

BaseEnv	Interface definition for reinforcement learning environments defining the minimum required functionality for being considered an environment.
StructuredEnv	Interface for environments with sub-step structure, which is generally enough to cover multi-step, hierarchical and multi-agent environments.
CoreEnv	Interface definition for core environments forming the basis for actual RL trainable environments.
StructuredEnvSpacesMixin	This interface complements the StructuredEnv by action and observation spaces.
MazeEnv	Base class for (gym style) environments wrapping a core environment and defining state and execution interfaces.
RenderEnvMixin	Interface for rendering functionality in environments (compatible with gym env).

continues on next page

Table 1 – continued from previous page

`RecordableEnvMixin`	This interface provides a standard way of exposing internal MazeState and MazeAction objects for trajectory data recording.
`SerializableEnvMixin`	This interface provides a standard way of exposing environment components whose state should be serialized together with the environment state object when for example recording trajectory data.
`TimeEnvMixin`	This interface provides a standard way of exposing environment time to external components and wrappers.
`EventEnvMixin`	This interface provides a standard way of attaching environment events to the log statistics system.
`SimulatedEnvMixin`	Environment interface for simulated environments.

BaseEnv

class `maze.core.env.base_env.`**`BaseEnv`**

Interface definition for reinforcement learning environments defining the minimum required functionality for being considered an environment.

abstract close() → None

Performs any necessary cleanup.

abstract reset() → Any

Resets the environment and returns the initial state.

Returns the initial state after resetting.

abstract seed(*seed: Any*) → None

Sets the seed for this environment.

Commonly an integer is sufficient to seed the random number generator(s), but more expressive env-specific seed structured are also supported.

Param seed: the seed integer initializing the random number generator or an env-specific seed structure.

abstract step(*action: Any*) → Tuple[Any, Any, bool, Dict[Any, Any]]

Environment step function.

Parameters **`action`** – the selected action to take.

Returns state, reward, done, info

StructuredEnv

class `maze.core.env.structured_env.`**`StructuredEnv`**

Interface for environments with sub-step structure, which is generally enough to cover multi-step, hierarchical and multi-agent environments.

This environment can continuously create and destroy a previously unknown, unlimited number of actors during the course of an episode. Every actor is associated with one of the available policies.

The lifecycle of the environment is decoupled from the lifecycle of the actors. The interaction loop should continue, until the environment as a whole is set to done, which is returned as usual by the step() function. Individual actors might end earlier, which can be queried by the is_actor_done() method.

Pseudo-code of the interaction loop:

# start a new episode observation = env.reset()

while not done: # find out which actor is next to act (dictated by the env) sub_step_key, actor_id = env.actor_id()

# obtain the next action from the policy action = sample_from_policy(observation, sub_step_key, actor_id)

# step the env observation, reward, done, info = env.step(action)

# optionally use is_actor_done() to find out if the actor was terminated (relevant during training)

`abstract actor_id``()` → Tuple[Union[str, int], int]
Returns the current sub step key along with the currently executed actor.

The env must decide the actor in *`reset()`* and *`step()`*. In between these calls the return is constant per convention and *`actor_id()`* can be called arbitrarily.

Notes: * The id is unique only with respect to the sub step (every sub step may have its own actor 0). * Identities of done actors can not be reused in the same rollout.

Returns The current actor, as tuple (sub step key, actor number).

`abstract is_actor_done``()` → bool
Returns True if the just stepped actor is done, which is different to the done flag of the environment.

Like for *`actor_id()`*, the env updates this flag in *`reset()`* and *`step()`*.

Returns True if the actor is done.

CoreEnv

`class` `maze.core.env.core_env.`**`CoreEnv`**
Interface definition for core environments forming the basis for actual RL trainable environments.

`abstract actor_id``()` → Tuple[Union[str, int], int]
Returns the currently executed actor along with the policy id. The id is unique only with respect to the policies (every policy has its own actor 0).

Note that identities of done actors can not be reused in the same rollout.

Returns The current actor, as tuple (policy id, actor number).

`abstract close``()` → None
Performs any necessary cleanup.

`get_kpi_calculator``()` → Optional[*maze.core.log_events.kpi_calculator.KpiCalculator*]
By default, Core Envs do not have to support KPIs.

`abstract get_maze_state``()` → Any
Return current state of the environment.

:return The same state as returned by reset().

`abstract get_renderer``()` → *maze.core.rendering.renderer.Renderer*
Return renderer instance that can be used to render the env.

:return Renderer instance

`abstract get_serializable_components``()` → Dict[str, Any]
List components that should be serialized as part of trajectory data.

`get_step_events``()` → Iterable[*maze.core.events.event_record.EventRecord*]
Get all events recorded in the current step from the EventService.

:return An iterable of the recorded events.

abstract is_actor_done() → bool
Returns True if the just stepped actor is done, which is different to the done flag of the environment.

Returns True if the actor is done.

abstract reset() → Any
Reset the environment and return initial state.

Returns The initial state after resetting.

abstract seed(*seed: int*) → None
Sets the seed for this environment's random number generator(s).

Param seed: the seed integer initializing the random number generator.

abstract step(*maze_action: Any*) → Tuple[Any, Union[float, numpy.ndarray, Any], bool, Dict[Any, Any]]
Environment step function.

Parameters **maze_action** – Environment MazeAction to take.

Returns state, reward, done, info

StructuredEnvSpacesMixin

class maze.core.env.structured_env_spaces_mixin.**StructuredEnvSpacesMixin**
This interface complements the StructuredEnv by action and observation spaces.

StructuredEnv defines the logic and is usually implemented in the core env. In order to make it a complete, trainable env, the space definitions from this class are needed.

abstract property action_space
The currently active gym action space.

abstract property action_spaces_dict
A dictionary of gym action spaces, with policy IDs as keys.

abstract property observation_space
The currently active gym observation space.

abstract property observation_spaces_dict
A dictionary of gym observation spaces, with policy IDs as keys.

MazeEnv

class maze.core.env.maze_env.**MazeEnv**(**args*, ***kwds*)
Base class for (gym style) environments wrapping a core environment and defining state and execution interfaces. The aim of this class is to provide reusable functionality across different gym environments. This functionality comprises for example the reset-function, the step-function or the render-function.

Parameters

- **core_env** – Core environment.
- **action_conversion_dict** – A dictionary with action conversion interface implementation and policy names as keys.
- **observation_conversion_dict** – A dictionary with observation conversion interface implementation and policy names as keys.

property action_conversion
Return the action conversion mapping for the current policy.

action_conversion_dict
The action conversion mapping used by this env.

property action_space
Keep this env compatible with the gym interface by returning the action space of the current policy.

property action_spaces_dict
Policy action spaces as dict.

actor_id() → Tuple[Union[str, int], int]
forward call to `self.core_env`

close() → None
forward call to `self.core_env`

core_env
wrapped `CoreEnv`

get_env_time() → int
Return ID of the current core env step as env time.

get_episode_id() → str
Return the ID of current episode (the ID changes on env reset).

get_kpi_calculator() → Optional[*maze.core.log_events.kpi_calculator.KpiCalculator*]
forward call to `self.core_env`

get_maze_action() → Any
Return last MazeAction object for trajectory recording.

get_maze_state() → Any
Return current State object for the core env for trajectory recording.

get_observation_and_action_dicts(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]
Convert MazeState and MazeAction back into observations and actions using the space conversion interfaces.

Parameters

- **maze_state** – State of the environment
- **maze_action** – MazeAction (the one following the state given as the first param)
- **first_step_in_episode** – True if this is the first step in the episode.

Returns observation and action dictionaries (keys are substep_ids)

get_renderer() → *maze.core.rendering.renderer.Renderer*
Return the renderer exposed by the underlying core env.

get_step_events() → Iterable[*maze.core.events.event_record.EventRecord*]
forward call to `self.core_env`

is_actor_done() → bool
forward call to `self.core_env`

maze_env
direct access to the maze env (useful to bypass the wrapper hierarchy)

metadata
Only there to be compatible with gym.core.Env

property observation_conversion
Return the state to observation mapping for the current policy.

observation_conversion_dict
The observation conversion mapping used by this env.

property observation_space
Keep this env compatible with the gym interface by returning the observation space of the current policy.

property observation_spaces_dict
Policy observation spaces as dict.

reset() → Any
Resets the environment and returns the initial observation.

Returns the initial observation after resetting.

reward_range
A tuple (reward min value, reward max value) to be compatible with gym.core.Env

seed(*seed: Any*) → None
forward call to `self.core_env`

spec
Only there to be compatible with gym.core.Env

step(*action: Any*) → Tuple[Any, float, bool, Dict[Any, Any]]
Take environment step (see `CoreEnv.step` for details).

Parameters **action** – the action the agent wants to take.

Returns observation, reward, done, info

RenderEnvMixin

class maze.core.env.render_env_mixin.**RenderEnvMixin**
Interface for rendering functionality in environments (compatible with gym env).

Intended to be combined with :obj:'~maze.core.env.base_env.BaseEnv` and potentially other environment interfaces by multiple inheritance. e.g. *class MyEnv(BaseEnv, RenderEnvMixin)*.

abstract render(*mode: str = 'human'*) → None
Render current state of the environment.

Param mode: the render mode.

RecordableEnvMixin

class maze.core.env.recordable_env_mixin.**RecordableEnvMixin**
This interface provides a standard way of exposing internal MazeState and MazeAction objects for trajectory data recording.

abstract get_episode_id() → str
Get ID of the current episode. Usually a UUID converted to a string, but can be a custom string as well.

Returns Episode ID string

abstract get_maze_action() → Any
Return the last MazeAction taken in the environment

Returns Last MazeAction object.

abstract get_maze_state() → Any
Return current state of the environment.

Returns Current environment state object.

abstract get_renderer() → *maze.core.rendering.renderer.Renderer*
Return renderer that can be used to render the recorded trajectory data.

Returns Renderer instance.

SerializableEnvMixin

class maze.core.env.serializable_env_mixin.**SerializableEnvMixin**
This interface provides a standard way of exposing environment components whose state should be serialized together with the environment state object when for example recording trajectory data.

Implement this interface if there are additional components in the env besides state that should be serialized.

abstract get_serializable_components() → Dict[str, Any]
Return all modules that should be serialized as part of the env besides state.

Important notes:

- All returned modules should support serialization using pickle. For most objects, this is possible out-of-the-box without any special changes. However, there are some notable exceptions like event interfaces – if any of the modules (or their attributes) keeps reference to an abstract object like events interface, the *__getstate__* method will need to be overriden to exclude these from pickling.

Returns Dict in the format of { "serializable_module_name": serializable_module }

TimeEnvMixin

class maze.core.env.time_env_mixin.**TimeEnvMixin**
This interface provides a standard way of exposing environment time to external components and wrappers. e.g. for event logging.

abstract get_env_time() → int

Returns Internal environment time represented as integer.

EventEnvMixin

class maze.core.env.event_env_mixin.**EventEnvMixin**
This interface provides a standard way of attaching environment events to the log statistics system.

Implement this interface in the environment to activate the statistics support.

abstract get_kpi_calculator() → Optional[*maze.core.log_events.kpi_calculator.KpiCalculator*]
If available, return an instance of a KPI calculator that can be used to calculate KPIs from events at the end of episode.

:return KPI calculator or None if KPIs are not supported in this env.

abstract get_step_events() → Iterable[*maze.core.events.event_record.EventRecord*]
Retrieve all recorded events of the current environment step.

SimulatedEnvMixin

class maze.core.env.simulated_env_mixin.**SimulatedEnvMixin**
Environment interface for simulated environments.

The main addition to StructuredEnv is the clone method, which resets the simulation to the given env state. This interface is used by Monte Carlo Tree Search.

abstract clone_from(*maze_state: Any*) → None
Clone an environment by resetting the simulation to its current state.

step_without_observation(*action: Dict[str, Union[int, numpy.ndarray]]*) → Tuple[Any, bool, Dict[Any, Any]]
Environment step function that does not return any observation.

This method can be significantly faster than the full step function in cases with expensive state to observation mappings.

Parameters **action** – the selected action to take.

Returns reward, done, info

Interfaces for additional components:

ObservationConversionInterface	Interface specifying the conversion of abstract environment state to the gym-compatible observation.
ActionConversionInterface	Interface specifying the conversion of agent actions to actual environment MazeActions.
MazeStateType	Internal indicator of special typing constructs.
MazeActionType	Internal indicator of special typing constructs.
RewardAggregatorInterface	Event aggregation object for reward customization and shaping.
EnvironmentContext	This class keeps track of services that can be employed by all objects of the agent-environment loop.

ObservationConversionInterface

class maze.core.env.observation_conversion.**ObservationConversionInterface**
Interface specifying the conversion of abstract environment state to the gym-compatible observation.

abstract maze_to_space(*maze_state: Any*) → Dict[str, numpy.ndarray]
Converts core environment state to a machine readable agent observation.

space() → gym.spaces.Dict
Returns respective Gym observation space.

space_to_maze(*observation: Dict[str, numpy.ndarray]*) → Any
Converts agent observation to core environment state. (This is most like not possible for most observation observation_conversion)

ActionConversionInterface

class maze.core.env.action_conversion.**ActionConversionInterface**
Interface specifying the conversion of agent actions to actual environment MazeActions.

maze_to_space(*maze_action: Any*) → Dict[str, numpy.ndarray]
Converts environment MazeAction to agent action.

Parameters **maze_action** – the environment MazeAction.

Returns the agent action.

noop_action()
Converts environment MazeAction to agent action.

Returns the noop action.

abstract space() → gym.spaces.Dict
Returns respective gym action space.

abstract space_to_maze(*action: Dict[str, numpy.ndarray], maze_state: Any*) → Any
Converts agent action to environment MazeAction.

Parameters

- **action** – the agent action.
- **maze_state** – the environment state.

Returns the environment MazeAction.

MazeStateType

maze.core.env.maze_state.**MazeStateType**

MazeActionType

maze.core.env.maze_action.**MazeActionType**

RewardAggregatorInterface

class maze.core.env.reward.**RewardAggregatorInterface**
Event aggregation object for reward customization and shaping.

abstract classmethod to_scalar_reward(*reward: Any*) → float
Aggregate sub-rewards to scalar reward.

This method is useful for example in a multi-agent setting where we could sum over multiple actors to assign a joint reward.

Param reward: The aggregated reward (e.g. per-agent reward for multi-agent RL settings).

Returns The scalar reward returned by the environment.

EnvironmentContext

class maze.core.env.environment_context.**EnvironmentContext**

This class keeps track of services that can be employed by all objects of the agent-environment loop.

Currently the context is populated by

- Event service: Acts as backend of the PubSub service, collects all events from the env. The event service can also be directly facilitated by components outside the environment (e.g. agent, heuristics, state/observation mapping)
- Episode ID: Generates and keeps track of the current episode ID Episode IDs are used for connecting logged statistics, events and recorded trajectory data together, making analysis and drill-down across these different levels possible.
- Step ID: Tracks ID of the core env step we are currently in. Helps wrappers recognize core env steps in multi-step scenarios.

property episode_id

Get the episode ID.

Episode ID is a UUID generated in a lazy manner, ensuring that if the ID is not needed, the potentially costly random UUID generation is avoided. Once generated, it stays the same for the entire episode and then is reset.

Returns Episode UUID as string

increment_env_step() → None

This must be called after the env step execution, to notify the services about the start of a new step.

reset_env_episode()

This must be called when resetting the environment, to notify the context about the start of a new episode.

1.4.2 Environment Wrappers

This page contains the reference documentation for environment wrappers. *Here* you can find a more extensive write up on how to work with these.

Overview

- *Interfaces and Utilities*
- *Built-in Wrappers*
- *Observation Pre-Processing Wrapper*
- *Observation Normalization Wrapper*
- *Gym Environment Wrapper*

Interfaces and Utilities

These are the wrapper interfaces, base classes and interfaces:

`Wrapper`	A transparent environment Wrapper that works with any manifestation of `BaseEnv`.

Wrapper

class `maze.core.wrappers.wrapper.`**`Wrapper`**(**args*, ***kwds*)

A transparent environment Wrapper that works with any manifestation of `BaseEnv`. It is intended as drop-in replacement for gym.core.Wrapper.

Gym Wrappers elegantly expose methods and attributes of all nested envs. However wrapping destroys the class hierarchy, querying the base classes is not straight-forward. This environment wrapper fixes the behaviour of isinstance() for arbitrarily nested wrappers.

Suppose we want to check the base class:

class MyGymWrapper(Wrapper[gym.Env]): ...

# construct an env and wrap it env = MyEnv() env = MyGymWrapper(env)

# this assertion fails assert isinstance(env, MyEnv) == True

TypingWrapper makes *isinstance()* work as intuitively expected:

# this time use MyWrapper, which is derived from this Wrapper class env = MyEnv() env = MyWrapper(env)

# now the assertions hold assert isinstance(env, MyEnv) == True assert isinstance(env, MyWrapper) == True

Note:

gym.core.Wrapper assumes the existence of certain attributes (action_space, observation_space, reward_range, metadata) and duplicates these attributes. This behaviour is unnecessary, because __getattr__ makes these members of the inner environment transparently available anyway.

`get_observation_and_action_dicts`(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]

Convert MazeState and MazeAction back into raw action and observation.

This method is mostly used when working with trajectory data, e.g. for imitation learning. As part of trajectory data, MazeState and MazeActions are recorded. For imitation learning, they then need to be converted to raw observations and actions in the desired format (i.e. using all the required wrappers etc.)

The conversion is done by first transforming the MazeState and MazeAction using the space interfaces in MazeEnv, and then running them through the entire wrapper stack ("back up").

Both the MazeState and the MazeAction on top of it are converted as part of this single method, as some wrappers (mostly multi-step ones) need them both together (to be able to split them into observations and actions taken in different sub-steps). If you are not using multi-step wrappers, you don't need to convert both MazeState and MazeAction, you can pass in just one of them. Not all wrappers have to support this though.

See below for an example implementation.

Note: The conversion of MazeState to observation is in the "natural" direction, how it takes place when stepping the env. This is not true for the MazeAction to action conversion – when stepping the env, actions are converted to MazeActions, whereas here the MazeAction needs to be converted back into the "raw" action (i.e. in reverse direction).

(!) Attention: In case that there are some stateful wrappers in the wrapper stack (e.g. a wrapper stacking observations from previous steps), you should ensure that (1) the first_step_in_episode flag is passed to this function correctly and (2) that all states and MazeActions are converted in order – as they happened during the recorded episode.

Parameters

- **`maze_state`** – MazeState to convert. If none, only MazeAction will be converted (not all wrappers support this).
- **`maze_action`** – MazeAction (the one following the state given as the first param). If none, only MazeState will be converted (not all wrappers support this, some need both).
- **`first_step_in_episode`** – True if this is the first step in the episode. Serves to notify stateful wrappers (e.g. observation stacking) that they should reset their state.

Returns observation and action dictionaries (keys are IDs of sub-steps)

`classmethod wrap`(*env: T*, ***kwargs*) → Union[T, WrapperType]

Creation method providing appropriate type hints. Preferred method to construct the wrapper compared to calling the class constructor directly. :param env: The environment to be wrapped :param kwargs: Arguments to be passed on to wrapper's constructor. :return A newly created wrapper instance. Since we want to allow sub-classes to use .wrap() without having to reimplement them and still facilitate proper typing hints, we use a generic to represent the type of cls. See https://stackoverflow.com/questions/39205527/can-you-annotate-return-type-when-value-is-instance-of-cls/39205612#39205612 on why/how to use this to indicate that an instance of cls is returned.

Types of Wrappers:

`ObservationWrapper`	A Wrapper with typing support modifying the environments observation.
`ActionWrapper`	A Wrapper with typing support modifying the agents action.
`RewardWrapper`	A Wrapper with typing support modifying the reward before passed to the agent.
`WrapperRegistry`	Handles dynamic registration of Wrapper sub-classes.

ObservationWrapper

class `maze.core.wrappers.wrapper.`**`ObservationWrapper`**(**args*, ***kwds*)

A Wrapper with typing support modifying the environments observation.

`get_observation_and_action_dicts`(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]

Convert the observations, keep actions the same.

`abstract observation`(*observation: Any*) → Any

Observation mapping method.

`reset`() → Any

Intercept `BaseEnv.reset` and map observation.

step (*action*) → Tuple[Any, Any, bool, Dict[Any, Any]]
Intercept `BaseEnv.step` and map observation.

ActionWrapper

class `maze.core.wrappers.wrapper.`**ActionWrapper** (**args*, ***kwds*)
A Wrapper with typing support modifying the agents action.

abstract action (*action: Any*) → Any
Abstract action mapping method.

get_observation_and_action_dicts (*maze_state: Optional[Any], maze_action: Optional[Any], first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]
Reverse the actions, keep the observations the same.

abstract reverse_action (*action: Any*) → Any
Abstract action reverse mapping method.

step (*action*) → Tuple[Any, Any, bool, Dict[Any, Any]]
Intercept `BaseEnv.step` and map action.

RewardWrapper

class `maze.core.wrappers.wrapper.`**RewardWrapper** (**args*, ***kwds*)
A Wrapper with typing support modifying the reward before passed to the agent.

get_observation_and_action_dicts (*maze_state: Optional[Any], maze_action: Optional[Any], first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]
Keep both actions and observation the same.

abstract reward (*reward: Any*) → Any
Reward mapping method.

step (*action*) → Tuple[Any, Any, bool, Dict[Any, Any]]
Intercept `BaseEnv.step` and map rewards.

WrapperRegistry

class `maze.core.wrappers.wrapper_registry.`**WrapperRegistry** (**args*, ***kwds*)
Handles dynamic registration of Wrapper sub-classes.

wrap_from_config (*env: T, wrapper_config: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]]*) → Union[*maze.core.wrappers.wrapper.Wrapper*, T]
Wraps environment in wrappers specified in wrapper_config.

Parameters

- **env** – Environment to wrap.
- **wrapper_config** – Wrapper specification.

Returns Wrapped environment of type Wrapper.

Built-in Wrappers

Below you find the reference documentation for environment wrappers.

General Wrappers:

LogStatsWrapper	A statistics logging wrapper for *BaseEnv*.
ObservationLoggingWrapper	A observation logging wrapper for *BaseEnv*.
TimeLimitWrapper	Wrapper to limit the environment step count, equivalent to gym.wrappers.time_limit.
RandomResetWrapper	A wrapper skipping the first few steps by taking random actions.
SortedSpacesWrapper	This class wraps a given StructuredEnvSpacesMixin env to ensure that all observation- and action-spaces are sorted alphabetically.
NoDictSpacesWrapper	Wraps observations and actions by replacing dictionary spaces with the sole contained sub-space.

LogStatsWrapper

class maze.core.wrappers.log_stats_wrapper.**LogStatsWrapper**(**args*, ***kwds*)

A statistics logging wrapper for *BaseEnv*.

Parameters **env** – The environment to wrap.

close()

Close the stats rendering figure if needed.

get_observation_and_action_dicts(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]

Keep both actions and observation the same.

get_stats(*level: maze.core.log_stats.log_stats.LogStatsLevel*) → *maze.core.log_stats.log_stats.LogStatsAggregator*

Implementation of the LogStatsEnv interface, return the statistics aggregator.

get_stats_value(*event: Callable*, *level: maze.core.log_stats.log_stats.LogStatsLevel*, *name: Optional[str] = None*) → Union[int, float, numpy.ndarray, dict]

Implementation of the LogStatsEnv interface, obtain the value from the cached aggregator statistics.

render_stats(*event_name: str = 'BaseEnvEvents.reward'*, *metric_name: str = 'value'*, *aggregation_func: Optional[Union[str, Callable]] = None*, *group_by: str = None*, *post_processing_func: Optional[Union[str, Callable]] = 'cumsum'*)

Render statistics from the currently running episode.

Rendering is based on event logs. You can select arbitrary events from those dispatched by the currently running environment.

Parameters

- **event_name** – Name of the even the even log corresponds to
- **metric_name** – Metric to use (one of the event attributes, e.g. "n_items" – depends on the event type)
- **aggregation_func** – Optionally, specifies how to aggregate the metric on step level, i.e. when there are multiple same events dispatched during the same step.

- **group_by** – Optionally, another of event attributes to group by on the step level (e.g. “product_id”)
- **post_processing_func** – Optionally, a function to post-process the data (“cumsum” is often used)

reset() → Any
Reset the environment and trigger the episode statistics calculation of the previous run.

step(*action: Any*) → Tuple[Any, Any, bool, Dict[Any, Any]]
Collect the rewards for the logging statistics

classmethod wrap(*env: T, logging_prefix: Optional[str] = None*) → Union[T, *maze.core.log_stats.log_stats_env.LogStatsEnv*]
Creation method providing appropriate type hints. Preferred method to construct the wrapper compared to calling the class constructor directly.

Parameters

- **env** – The environment to be wrapped.
- **logging_prefix** – The episode statistics is connected to the logging system with this tagging prefix. If None, no logging happens.

:return A newly created wrapper instance.

write_epoch_stats()
Implementation of the LogStatsEnv interface, call reduce on the episode aggregator.

ObservationLoggingWrapper

class maze.core.wrappers.observation_logging_wrapper.**ObservationLoggingWrapper**(**args, **kwds*)
A observation logging wrapper for `BaseEnv`.

Parameters **env** – The environment to wrap.

get_observation_and_action_dicts(*maze_state: Optional[Any], maze_action: Optional[Any], first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]
Keep both actions and observation the same.

step(*action: Any*) → Tuple[Any, Any, bool, Dict[Any, Any]]
Create the observation logs on every step

TimeLimitWrapper

class maze.core.wrappers.time_limit_wrapper.**TimeLimitWrapper**(**args, **kwds*)
Wrapper to limit the environment step count, equivalent to gym.wrappers.time_limit.

Additionally to the gym wrapper, this one supports adjusting the limit after construction.

close() → None
forward call to the inner env

get_observation_and_action_dicts(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]

This wrapper does not modify observations and actions.

reset() → Any

Override BaseEnv.reset to reset the step count.

seed(*seed: int*) → None

forward call to the inner env

set_max_episode_steps(*max_episode_steps: int*) → None

Set the step limit.

Parameters **max_episode_steps** – The environment step() function sets the done flag if this step limit is reached. If 0, the step limit is disabled.

step(*action: Any*) → Tuple[Any, Any, bool, Dict[Any, Any]]

Override BaseEnv.step and set done if the step limit is reached.

RandomResetWrapper

class maze.core.wrappers.random_reset_wrapper.**RandomResetWrapper**(**args*, ***kwds*)

A wrapper skipping the first few steps by taking random actions. This is useful for skipping irrelevant initial parts of a trajectory or for introducing randomness in the training process.

Parameters

- **env** – Environment/wrapper to wrap.
- **min_skip_steps** – Minimum number of steps to skip.
- **max_skip_steps** – Maximum number of steps to skip.

get_observation_and_action_dicts(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]

This wrapper does not modify observations and actions.

reset() → Any

Override BaseEnv.reset to reset the step count.

SortedSpacesWrapper

class maze.core.wrappers.sorted_spaces_wrapper.**SortedSpacesWrapper**(**args*, ***kwds*)

This class wraps a given StructuredEnvSpacesMixin env to ensure that all observation- and action-spaces are sorted alphabetically. This is required that Maze custom action distributions and observation processing are in line with RLLib's internal processing pipeline.

property action_space

The currently active gym action space.

get_observation_and_action_dicts(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]
This wrapper does not modify observations and actions.

property observation_space
Keep this env compatible with the gym interface by returning the observation space of the current policy.

NoDictSpacesWrapper

class `maze.core.wrappers.no_dict_spaces_wrapper.`**NoDictSpacesWrapper**(**args*, ***kwds*)
Wraps observations and actions by replacing dictionary spaces with the sole contained sub-space. This wrapper is for example required when working with external frameworks not supporting dictionary spaces.

action(*action: numpy.ndarray*) → Dict[str, numpy.ndarray]
Implementation of `ActionWrapper` interface.

property action_space
The currently active gym action space.

property action_spaces_dict
A dictionary of gym action spaces, with policy IDs as keys.

get_observation_and_action_dicts(*maze_state: Optional[Any]*, *maze_action: Optional[Any]*, *first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]
Convert the observations, reverse the actions.

observation(*observation: Any*) → Any
Implementation of `ObservationWrapper` interface.

property observation_space
The currently active gym observation space.

property observation_spaces_dict
A dictionary of gym observation spaces, with policy IDs as keys.

reset() → Any
Intercept `BaseEnv.reset` and map observation.

reverse_action(*action: Dict[str, numpy.ndarray]*) → numpy.ndarray
Implementation of `ActionWrapper` interface.

step(*action*) → Tuple[Any, Any, bool, Dict[Any, Any]]
Intercept `BaseEnv.step` and map observation.

ObservationWrappers:

`DictObservationWrapper`	Wraps a single observation into a dictionary space.
`ObservationLoggingWrapper`	A observation logging wrapper for `BaseEnv`.
`ObservationStackWrapper`	An wrapper stacking the observations of multiple subsequent time steps.
`NoDictObservationWrapper`	Wraps observations by replacing the dictionary observation space with the sole contained sub-space.

DictObservationWrapper

class maze.core.wrappers.dict_observation_wrapper.**DictObservationWrapper**(**args, **kwds*)

Wraps a single observation into a dictionary space.

observation(*observation: numpy.ndarray*)

Implementation of *ObservationWrapper* interface.

ObservationStackWrapper

class maze.core.wrappers.observation_stack_wrapper.**ObservationStackWrapper**(**args, **kwds*)

An wrapper stacking the observations of multiple subsequent time steps.

Provides functionality for:

- selecting which observations to stack
- how many past observations should be stacked
- stacking deltas with the current step observation (instead of the observations itself)

Parameters

- **env** – Environment/wrapper to wrap.
- **stack_config** – The observation stacking configuration.

  observation: The name (key) of the respective observation keep_original: Bool, indicates weather to keep or remove the original observation from the dictionary. tag: Optional[str], tag to add to observation (e.g. stacked) delta: Bool, if true deltas are stacked to the previous observation stack_steps: Int, number of past steps to be stacked

get_observation_and_action_dicts(*maze_state: Optional[Any], maze_action: Optional[Any], first_step_in_episode: bool*) → Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int, str], Any]]]

If this is the first step in an episode, reset the observation stack.

observation(*observation: Dict[str, numpy.ndarray]*) → Dict[str, numpy.ndarray]

Stack observations.

Parameters observation – The observation to be stacked.

Returns The sacked observation.

reset() → Dict[str, numpy.ndarray]

Intercept ObservationWrapper.reset and map observation.

NoDictObservationWrapper

class maze.core.wrappers.no_dict_observation_wrapper.**NoDictObservationWrapper**(**args, **kwds*)

Wraps observations by replacing the dictionary observation space with the sole contained sub-space. This wrapper is for example required when working with external frameworks not supporting dictionary observation spaces.

observation(*observation: Any*) → Any
Implementation of *ObservationWrapper* interface.

property observation_space
The currently active gym observation space.

property observation_spaces_dict
A dictionary of gym observation spaces, with policy IDs as keys.

ActionWrappers:

DictActionWrapper	Wraps either a single action space or a tuple action space into dictionary space.
NoDictActionWrapper	Wraps actions by replacing the dictionary action space with the sole contained sub-space.
SplitActionsWrapper	Splits an actions into separate ones.
DiscretizeActionsWrapper	The DiscretizeActionsWrapper provides functionality for discretizing individual continuous actions into discrete

DictActionWrapper

class maze.core.wrappers.dict_action_wrapper.**DictActionWrapper**(**args, **kwds*)

Wraps either a single action space or a tuple action space into dictionary space.

Parameters **env** – The environment to wrap.

action(*action: Dict[str, numpy.ndarray]*) → Union[numpy.ndarray, Tuple[numpy.ndarray]]
Implementation of *ActionWrapper* interface.

reverse_action(*action: Union[numpy.ndarray, Tuple[numpy.ndarray]]*) → Dict[str, numpy.ndarray]
Implementation of *ActionWrapper* interface.

NoDictActionWrapper

class maze.core.wrappers.no_dict_action_wrapper.**NoDictActionWrapper**(**args, **kwds*)

Wraps actions by replacing the dictionary action space with the sole contained sub-space. This wrapper is for example required when working with external frameworks not supporting dictionary action spaces.

action(*action: numpy.ndarray*) → Dict[str, numpy.ndarray]
Implementation of *ActionWrapper* interface.

property action_space
The currently active gym action space.

property action_spaces_dict
A dictionary of gym action spaces, with policy IDs as keys.

reverse_action(*action: Dict[str, numpy.ndarray]*) → numpy.ndarray
Implementation of `ActionWrapper` interface.

SplitActionsWrapper

class maze.core.wrappers.split_actions_wrapper.**SplitActionsWrapper**(**args, **kwds*)

Splits an actions into separate ones.

An example is given by the LunarLanderContinuous-v2 env. Here we have a box action spaces with shape (2,) such that dimension 0 is the up/down action and dimension 1 is the left/right action. Now if we would like to split this action correspondingly we can wrap the env with the following config:

split_config:

action:

action_up: indices: [0]

action_side: indices: [1]

Now the actions as well as the action space is consists of two actions (action_up/action_side).

Parameters

- **env** – Environment/wrapper to wrap.
- **split_config** – The action splitting configuration.

action(*action: Dict[str, numpy.ndarray]*) → Dict[str, numpy.ndarray]
Implementation of `ActionWrapper` interface.

property action_space
The currently active gym action space.

property action_spaces_dict
A dictionary of gym action spaces, with policy IDs as keys.

reverse_action(*action: Dict[str, numpy.ndarray]*) → Dict[str, numpy.ndarray]
Implementation of `ActionWrapper` interface.

DiscretizeActionsWrapper

class maze.core.wrappers.discretize_actions_wrapper.**DiscretizeActionsWrapper**(**args, **kwds*)

The DiscretizeActionsWrapper provides functionality for discretizing individual continuous actions into discrete ones.

An example is given by having a continuous action called 'action_up' with space: gym.spaces.Box(shape=(5,), low=[-1,-1,-1,-1,-1], high=[1,1,1,1,1]

discretization_config:

action_up: num_bins: 5 low: [-1, 0, 0.5, 0, 0] high: 1

Now the action space will be split where each of the 5 continuous values of the box spaces are split evenly within the ranges of (-1,1),(0,1),(0.5,1), (0,1), (0,1) respectively.

Parameters

- **env** – Environment/wrapper to wrap.

- **discretization_config** – The action discretization configuration.

action (*action: Dict[str, numpy.ndarray]*) → Dict[str, numpy.ndarray]
Implementation of `ActionWrapper` interface.

property action_space
The currently active gym action space.

property action_spaces_dict
A dictionary of gym action spaces, with policy IDs as keys.

reverse_action (*action: Dict[str, numpy.ndarray]*) → Dict[str, numpy.ndarray]
Implementation of `ActionWrapper` interface.

RewardWrappers:

`RewardScalingWrapper`	Scales original step reward by a multiplicative scaling factor.
`RewardClippingWrapper`	Clips original step reward to range [min, max].

RewardScalingWrapper

class maze.core.wrappers.reward_scaling_wrapper.**RewardScalingWrapper** (**args, **kwds*)
Scales original step reward by a multiplicative scaling factor.

Parameters

- **env** – The underlying environment.
- **scale** – Multiplicative reward scaling factor.

reward (*reward: float*) → float
Scales the original reward.

Parameters **reward** – The original reward.

Returns The scaled reward.

RewardClippingWrapper

class maze.core.wrappers.reward_clipping_wrapper.**RewardClippingWrapper** (**args, **kwds*)
Clips original step reward to range [min, max].

Parameters

- **env** – The underlying environment.
- **min_val** – Minimum allowed reward value.
- **max_val** – Maximum allowed reward value.

reward (*reward: float*) → float
Clips the original reward.

Parameters **reward** – The original reward.

Returns The clipped reward.

Observation Pre-Processing Wrapper

Below you find the reference documentation for observation pre-processing. *Here* you can find a more extensive write up on how to work with the observation pre-processing package.

These are interfaces and components required for observation pre-processing:

PreProcessingWrapper	An observation pre-processing wrapper.
PreProcessor	Interface for observation pre-processors.

PreProcessingWrapper

class maze.core.wrappers.observation_preprocessing.preprocessing_wrapper.**PreProcessingWrapp**

An observation pre-processing wrapper. It provides functionality for:

- pre-processing observations (flattening, one-hot encoding, . . .)
- adopting the observation spaces accordingly

Parameters

- **env** – Environment/wrapper to wrap.
- **pre_processor_mapping** – The pre-processing configuration. Example mappings can be found in our reference documentation.

observation(*observation: Any*) → Any

Pre-processes observations.

Parameters **observation** – The observation to be pre-processed.

Returns The pre-processed observation.

PreProcessor

class maze.core.wrappers.observation_preprocessing.preprocessors.base.**PreProcessor**(*observatio*
gym.spaces
***kwargs*)

Interface for observation pre-processors. Pre-processors implementing this interface can be used in combination with the *PreProcessingWrapper*.

Parameters

- **observation_space** – The observation space to pre-process.
- **kwargs** – Arguments to be passed on to preprocessor's constructor.

abstract process(*observation: numpy.ndarray*) → numpy.ndarray

Pre-processes the observation.

Parameters **observation** – The observation to pre-process.

Returns The pre-processed observation.

abstract processed_shape() → Tuple[int, . . .]

Computes the observation's shape after pre-processing.

Returns The resulting shape.

abstract processed_space() → gym.spaces.Box
Modifies the given observation space according to the respective pre-processor.

Returns The updated observation space.

tag() → str
Returns a tag identifying the pre-processed feature.

Returns The pre-processor's tag.

These are the available built-in **maze.pre_processors** compatible with the PreProcessingWrapper:

FlattenPreProcessor	An array flattening pre-processor.
OneHotPreProcessor	An one-hot encoding pre-processor for categorical features.
ResizeImgPreProcessor	An image resizing pre-processor.
TransposePreProcessor	An array transposition pre-processor.
UnSqueezePreProcessor	An un-squeeze pre-processor.
Rgb2GrayPreProcessor	An rgb-to-gray-scale conversion pre-processor.

FlattenPreProcessor

class maze.core.wrappers.observation_preprocessing.preprocessors.flatten.**FlattenPreProcess**

An array flattening pre-processor.

Parameters

- **observation_space** – The observation space to pre-process.
- **num_flatten_dims** – The number of dimensions to flatten out (from right).

process(*observation: numpy.ndarray*) → numpy.ndarray
implementation of *PreProcessor* interface

processed_shape() → Tuple[int, . . .]
implementation of *PreProcessor* interface

processed_space() → gym.spaces.Box
implementation of *PreProcessor* interface

OneHotPreProcessor

class maze.core.wrappers.observation_preprocessing.preprocessors.one_hot.**OneHotPreProcesso**

An one-hot encoding pre-processor for categorical features.

process(*observation: numpy.ndarray*) → numpy.ndarray
implementation of *PreProcessor* interface

processed_shape() → Tuple[int, . . .]
implementation of *PreProcessor* interface

processed_space() → gym.spaces.Box
implementation of *PreProcessor* interface

ResizeImgPreProcessor

class maze.core.wrappers.observation_preprocessing.preprocessors.resize_img.**ResizeImgPrePro**

An image resizing pre-processor.

Parameters

- **observation_space** – The observation space to pre-process.
- **target_size** – Target size of resized image.
- **transpose** – Transpose rgb channel is required (should be last dimension such as [96, 96, 3]).

process (*observation: numpy.ndarray*) → numpy.ndarray
implementation of *PreProcessor* interface

processed_shape () → Tuple[int, . . .]
implementation of *PreProcessor* interface

processed_space () → gym.spaces.Box
implementation of *PreProcessor* interface

TransposePreProcessor

class maze.core.wrappers.observation_preprocessing.preprocessors.transpose.**TransposePrePro**

An array transposition pre-processor.

Parameters

- **observation_space** – The observation space to pre-process.
- **axes** – The num ordering of the axes of the input array.

process (*observation: numpy.ndarray*) → numpy.ndarray
implementation of *PreProcessor* interface

processed_shape () → Tuple[int, . . .]
implementation of *PreProcessor* interface

processed_space () → gym.spaces.Box
implementation of *PreProcessor* interface

UnSqueezePreProcessor

class maze.core.wrappers.observation_preprocessing.preprocessors.unsqueeze.**UnSqueezePrePro**

An un-squeeze pre-processor.

Parameters

- **observation_space** – The observation space to pre-process.
- **dim** – Index where to add an additional dimension.

process (*observation: numpy.ndarray*) → numpy.ndarray
implementation of *PreProcessor* interface

processed_shape () → Tuple[int, . . .]
implementation of *PreProcessor* interface

processed_space () → gym.spaces.Box
implementation of *PreProcessor* interface

Rgb2GrayPreProcessor

class maze.core.wrappers.observation_preprocessing.preprocessors.rgb2gray.**Rgb2GrayPreProces**

An rgb-to-gray-scale conversion pre-processor.

Parameters

- **observation_space** – The observation space to pre-process.
- **rgb_dim** – Dimension of the rgb channels.

process (*observation: numpy.ndarray*) → numpy.ndarray
implementation of *PreProcessor* interface

processed_shape () → Tuple[int, . . .]
implementation of *PreProcessor* interface

processed_space () → gym.spaces.Box
implementation of *PreProcessor* interface

Observation Normalization Wrapper

Below you find the reference documentation for observation normalization. *Here* you can find a more extensive write up on how to work with the observation normalization package.

These are interfaces and utility functions required for observation normalization:

ObservationNormalizationWrapper	An observation normalization wrapper.
ObservationNormalizationStrategy	Abstract base class for normalization strategies.
obtain_normalization_statistics	Obtain the normalization statistics of a given environment.

continues on next page

Table 11 – continued from previous page

estimate_observation_normalization_stat	Helper function estimating normalization statistics.
make_normalized_env_factory	Wrap an existing env factory to assign the passed normalization statistics.

ObservationNormalizationWrapper

class maze.core.wrappers.observation_normalization.observation_normalization_wrapper.**Obser**

An observation normalization wrapper. It provides functionality for:

- normalizing observations according to specified normalization strategies
- clipping observations according to specified min and max values
- estimating normalization statistics from observations collecting by interacting with the environment
- manually overwriting the observation normalization parameters

The current implementation assumes that observation space is **always** a Dict (even if just a Dict-wrapped Box).

Parameters

- **env** – Environment/wrapper to wrap.
- **default_strategy** – The default observation normalization strategy.
- **default_statistics** – Manual default normalization statistics.
- **statistics_dump** – Path to a pickle file dump of normalization statistics.
- **sampling_policy** – The sampling policy for estimating the statistics.
- **exclude** – List of observation keys to exclude from normalization.
- **manual_config** – Additional manual configuration options.

dump_statistics() → None

Dump statistics to file.

estimate_statistics() → None

Estimates and sets the observation statistics from collected observations.

get_statistics() → Dict[str, Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]]

Returns the normalization statistics of the respective normalization strategy. :return: The normalization statistics for all sub steps and all dictionary observations.

observation(*observation: Any*) → Any

Collect observations for statistics computation or normalize them.

Parameters **observation** – The observation to be normalized.

Returns The normalized observation.

classmethod register_new_observation_normalization_strategy(*containing_submodule: Any*)

Registers a new observation normalization strategy.

Parameters **containing_submodule** – Add all classes implementing ObservationNormalizationStrategy by walking the module recursively.

set_normalization_statistics(*stats: Dict[str, Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]]*) → None

Apply existing normalization statistics.

Parameters **`stats`** – The statistics dict

`set_observation_collection`(*status: bool*) → None

Activate / deactivate observation collection.

Parameters **`status`** – If True observations are collected for statistics computation. If False observations are normalized with the provided statistics

ObservationNormalizationStrategy

class `maze.core.wrappers.observation_normalization.normalization_strategies.base.`**`Observatio`**

Abstract base class for normalization strategies.

Provides functionality for:

- normalizing gym.Box observations as well as for normalizing the originally defined observation space.
- setting and getting the currently employed normalization statistics.
- interface definition for estimating the normalization statistics from a list of observations
- interface definition for normalizing a given gym.Box (np.ndarray) observation

Parameters

- **`observation_space`** – The observations space to be normalized.
- **`clip_range`** – The minimum and maximum value allowed for an observation.
- **`axis`** – Defines the axis along which to compute normalization statistics

abstract **`estimate_stats`**(*observations: List[numpy.ndarray]*) → Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]

Estimate observation statistics from collected observations.

Parameters **`observations`** – A lists of observations.

`get_statistics`() → Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]

Get normalization statistics.

Returns The normalization statistics.

`is_initialized`() → bool

Checks if the normalization strategy is fully initialized.

Returns True if fully initialized and ready to normalize; else False.

normalize_and_process_value(*value: numpy.ndarray*) → numpy.ndarray
Normalizes and post-processes the actual observation (see also: normalize_value).

Parameters **value** – Observation value to be normalized.

Returns Normalized and processed observation.

abstract normalize_value(*value: numpy.ndarray*) → numpy.ndarray
Normalizes the actual observation value with provided statistics. The type and shape of value and statistics have to match.

Parameters **value** – Observation to be normalized.

Returns Normalized observation.

normalized_space() → gym.spaces.Box
Normalizes extrema (low and high) in the observation space with respect to the given statistics. (e.g. it sets the maximum value of a Box space to the maximum in the respective observation)

Returns Observation space with extrema adjusted w.r.t. statistics and normalization strategy.

set_statistics(*stats: Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]*) → None
Set normalization statistics.

Parameters **stats** – A dictionary containing the respective observation normalization statistics.

obtain_normalization_statistics

class maze.core.wrappers.observation_normalization.observation_normalization_utils.**obtain_n**

Obtain the normalization statistics of a given environment.

- Returns None, if the ObservationNormalizationWrapper is not implemented
- Returns the loaded statistics, if available
- Runs the estimation and returns the newly calculated statistics, if not loaded previously

Parameters

- **env** – Environment with applied ObservationNormalizationWrapper (function returns None immediately if this is not the case.
- **n_samples** – Number of samples (=steps) to collect normalization statistics at the beginning of the training.

Returns The normalization statistics or None if the ObservationNormalizationWrapper is not implemented by the env.

estimate_observation_normalization_statistics

`class maze.core.wrappers.observation_normalization.observation_normalization_utils.estimate`

Helper function estimating normalization statistics. :param env: The observation normalization wrapped environment. :param n_samples: The number of samples to take for statistics computation.

make_normalized_env_factory

`class maze.core.wrappers.observation_normalization.observation_normalization_utils.make_nor`

Wrap an existing env factory to assign the passed normalization statistics.

Parameters

- **env_factory** – The existing env factory
- **normalization_statistics** – The normalization statistics that should be applied to the env

Returns The wrapped env factory

These are the available built-in **maze.normalization_strategies** compatible with the ObservationNormalizationWrapper:

MeanZeroStdOneObservationNormalizationStrategy	Normalizes observations to have zero mean and standard deviation one.
RangeZeroOneObservationNormalizationStrategy	Normalizes observations to value range [0, 1].

MeanZeroStdOneObservationNormalizationStrategy

class `maze.core.wrappers.observation_normalization.normalization_strategies.mean_zero_std_o`

Normalizes observations to have zero mean and standard deviation one.

The strategy first subtracts the observation mean followed by a division with the standard deviation. Depending on the original distribution of the input observations this yields a standard Normal.

estimate_stats(*observations: List[numpy.ndarray]*) → Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]
Implementation of *ObservationNormalizationStrategy* interface.

normalize_value(*value: numpy.ndarray*) → numpy.ndarray
Implementation of *ObservationNormalizationStrategy* interface.

RangeZeroOneObservationNormalizationStrategy

class `maze.core.wrappers.observation_normalization.normalization_strategies.range_zero_one`

Normalizes observations to value range [0, 1].

The strategy subtracts in a first step the minimum observed value to shift the lowest value after normalization to zero. In a subsequent step we divide the observation with the maximum of the previous step yielding observations in the range [0, 1].

estimate_stats(*observations: List[numpy.ndarray]*) → Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]
Implementation of *ObservationNormalizationStrategy* interface.

normalize_value(*value: numpy.ndarray*) → numpy.ndarray
Implementation of *ObservationNormalizationStrategy* interface.

Gym Environment Wrapper

Below you find the reference documentation for wrapping gym environments. *Here* you can find a more extensive write up on how to integrate Gym environments within Maze.

These are the contained components:

`GymMazeEnv`	Wraps a Gym env into a Maze environment.
`make_gym_maze_env`	Initializes a `GymMazeEnv` by registered Gym env name (id).
`GymCoreEnv`	Wraps a Gym environment into a maze core environment.
`GymRenderer`	A Maze-compatible Gym renderer.
`GymRewardAggregator`	A dummy reward aggregation object simply repeating the environment's original reward.
`GymObservationConversion`	A dummy conversion interface asserting that the observation is packed into a dictionary space.
`GymActionConversion`	A dummy conversion interface asserting that the action is packed into a dictionary space.

GymMazeEnv

class `maze.core.wrappers.maze_gym_env_wrapper.`**`GymMazeEnv`**(**args*, ***kwds*)

Wraps a Gym env into a Maze environment.

Example: *env = GymMazeEnv(env="CartPole-v0")*

Parameters **`env`** – The gym environment to wrap or the environment id.

`clone_from`(*maze_state: Any*) → None

Reset this gym environment to the given state by creating a deep copy of the *env.state* instance variable

make_gym_maze_env

class `maze.core.wrappers.maze_gym_env_wrapper.`**`make_gym_maze_env`**(*name: str*)

Initializes a `GymMazeEnv` by registered Gym env name (id).

Parameters **`name`** – The name (id) of a registered Gym environment.

Returns The instantiated environment.

GymCoreEnv

class `maze.core.wrappers.maze_gym_env_wrapper.`**`GymCoreEnv`**(*env: gym.Env*)

Wraps a Gym environment into a maze core environment.

Parameters **`env`** – The Gym environment.

`actor_id`() → Tuple[Union[str, int], int]

Intercept `CoreEnv.actor_id`

`close`() → None

Intercept `CoreEnv.close`

get_maze_state() → Any
Intercept `CoreEnv.get_maze_state`

get_renderer() → *maze.core.rendering.renderer.Renderer*
Intercept `CoreEnv.get_renderer`

get_serializable_components() → Dict[str, Any]
Intercept `CoreEnv.get_serializable_components`

is_actor_done() → bool
Intercept `CoreEnv.is_actor_done`

reset() → Any
Intercept `CoreEnv.reset`

seed(*seed: int*) → None
Intercept `CoreEnv.seed`

step(*maze_action: Any*) → Tuple[Any, Union[float, numpy.ndarray, Any], bool, Dict[Any, Any]]
Intercept `CoreEnv.step`

GymRenderer

class `maze.core.wrappers.maze_gym_env_wrapper.`**GymRenderer**(*env: gym.Env*)
A Maze-compatible Gym renderer.

render(*maze_state: Any*, *maze_action: Optional[Any]*, *events:* maze.core.log_events.step_event_log.StepEventLog, ***kwargs*) → None
Render the current state of the environment.

GymRewardAggregator

class `maze.core.wrappers.maze_gym_env_wrapper.`**GymRewardAggregator**
A dummy reward aggregation object simply repeating the environment's original reward.

get_interfaces() → List[Type[abc.ABC]]
Nothing to do here

classmethod to_scalar_reward(*reward: float*) → float
Nothing to do here for this env.

Param reward: already a scalar reward

Returns the same scalar reward

GymObservationConversion

class `maze.core.wrappers.maze_gym_env_wrapper.`**GymObservationConversion**(*env: gym.Env*)
A dummy conversion interface asserting that the observation is packed into a dictionary space.

Parameters **env** – Gym environment.

maze_to_space(*maze_state: Any*) → Any
Converts core environment state to agent observation.

space() → gym.Space
Returns respective gym observation space.

space_to_maze (*observation: Dict[str, numpy.ndarray]*) → Any
Converts agent observation to core environment state. (This is most like not possible for most observation observation_conversion)

GymActionConversion

class maze.core.wrappers.maze_gym_env_wrapper.**GymActionConversion** (*env: gym.Env*)
A dummy conversion interface asserting that the action is packed into a dictionary space.

Parameters **env** – Gym environment.

maze_to_space (*maze_action: Any*) → Dict[str, numpy.ndarray]
Converts environment MazeAction to agent action.

Parameters **maze_action** – the environment MazeAction.

Returns the agent action.

space () → gym.spaces.Dict
Returns respective gym action space.

space_to_maze (*action: Dict[str, numpy.ndarray], maze_state: Any*) → Any
Converts agent action to environment MazeAction.

Parameters

- **action** – the agent action.
- **maze_state** – the environment state.

Returns the environment MazeAction.

1.4.3 Event System, Logging & Statistics

This page contains the reference documentation for the event and logging system.

Overview

- *Event System*
- *Event Logging*
- *Statistics Logging*

Event System

These are interfaces, classes and utility functions of the event system:

Subscriber	Event aggregation object.
Pubsub	Implementation of a message broker (Pubsub stands for publish and subscribe).
event_topic_factory	Constructs a proxy instance of the event interface, as required by EventService and LogStatsAggregator.

continues on next page

Table 14 – continued from previous page

EventScope	Base class for all services that integrate with the event system and therefore use EventService as their backend.
EventService	Manages the recording of event invocations and provides simple event routing functionality.
EventCollection	A collection of EventRecord instances that can be queried by event specification.
EventRecord	This auxiliary class is used to record calls to the event interface

Subscriber

class maze.core.events.pubsub.**Subscriber**
Event aggregation object.

abstract get_interfaces() → List[Type[abc.ABC]]
Specification of the event interfaces this subscriber wants to receive events from. Every subscriber must implement this configuration method.

Returns A list of interface classes

notify_event(*event:* maze.core.events.event_record.EventRecord)
Notify the subscriber of a new event occurrence.

Parameters **event** – the event

Returns None

query_events(*event_spec: Union[Callable, Iterable[Callable]]*) → Iterable
Return all events collected at the current env step matching one or more given event types. The event types are specified by the interface member function object itself.

Event calls are recorded as EventRecord, an object providing access to the passed arguments of the event method.

Parameters **event_spec** – Specifies the event type by the interface member function. Can either be a single event type specification or a list of specifications.

Returns An iterable to the event objects.

reset()
Reset event aggregation.

Returns None

Pubsub

class maze.core.events.pubsub.**Pubsub**(*event_collector:* maze.core.events.event_service.EventService)
Implementation of a message broker (Pubsub stands for publish and subscribe).

create_event_topic(*interface_class: Type[T]*) → T
Returns a proxy instance of the event interface, which the publisher can use to publish events. Behind the scenes every event invocation is serialized as EventRecord object and then routed to the registered subscribers.

Parameters **interface_class** – The class object of an abstract interface that defines the events as methods.

Returns A proxy object, dynamically derived from the passed *interface_class*. This class is intended to be used by the publisher to trigger events.

interface_to_subscribers: Dict[Type[T], List[*Subscriber*]]
map of interface class to the list of subscribed receivers

notify_event(*event:* maze.core.events.event_record.EventRecord) → None
Notify about a new event. This is invoked by the EventService.

:param event The event to be added.

notify_next_step()
Resets the aggregated events of all registered subscribers

Returns None

register_subscriber(*new_subscriber:* maze.core.events.pubsub.Subscriber)

Register a subscriber to receive events from certain published interfaces, specified by Subscriber.get_interfaces()

Parameters **new_subscriber** – the subscriber to be registered

Returns None

subscribers: List[*Subscriber*]
all registered subscribers

event_topic_factory

`class` maze.core.events.event_topic_factory.**event_topic_factory**(*interface_class: Type[T], fn_notify_event: Callable[[*maze.core.events.event_record *None]*)

Constructs a proxy instance of the event interface, as required by EventService and LogStatsAggregator.

Parameters

- **interface_class** – The class object of an abstract interface that defines the events as methods.
- **fn_notify_event** – The proxy forwards all method invocations to fn_notify_event

Returns A proxy object, dynamically derived from the passed *interface_class*.

EventScope

`class` maze.core.events.event_service.**EventScope**
Base class for all services that integrate with the event system and therefore use EventService as their backend.

Currently PubSub is the only concrete implementation.

abstract notify_event(*event:* maze.core.events.event_record.EventRecord) → None
Called on all event occurrences, if the respective event interface is registered for this scope.

Parameters **event** – the event

Returns None

abstract notify_next_step() → None
Notification after the env step execution about the start of a new step.

EventService

class maze.core.events.event_service.**EventService**
Manages the recording of event invocations and provides simple event routing functionality. There is one EventService instance in every agent-environment loop, provided by the AgentEnvironmentContext.

Within the environment the richer routing functionality provided by PubSub should be utilized, rather than directly interacting with this class.

class TopicInfo(*interface_class: Type[T]*, *scope:* maze.core.events.event_service.EventScope, *proxy: T*)
internal class to keep track of the topic state, including the collected events

create_event_topic(*interface_class: Type[T]*, *scope:* maze.core.events.event_service.EventScope *= None*) → T
Create a proxy instance of the event interface, which can be used conveniently to publish events. Returns an existing proxy, if it has been created before.

Parameters

- **interface_class** – The class object of an abstract interface that defines the events as methods.
- **scope** – Every event topic can be bound to a single scope, e.g. a certain PubSub instance, to ensure that all events of the topic *interface_class* will be received by this PubSub instance.

Returns A proxy object, dynamically derived from the passed *interface_class*, that can be used to trigger events.

iterate_event_records() → Generator[*maze.core.events.event_record.EventRecord*, None, None]
A generator to iterate all collected events

notify_event(*event:* maze.core.events.event_record.EventRecord) → None
Notify the event service about a new event. This is invoked by the event topic proxies.

:param event The event to be added.

notify_next_step()
Notify this service after the env step execution about the start of a new step. This should only be called by the AgentEnvironmentContext.

Clears all collected events and notifies all registered scopes.

EventCollection

class maze.core.events.event_collection.**EventCollection**(*events: Iterable[*maze.core.events.event_record.EventRecord*] = ()*)
A collection of EventRecord instances that can be queried by event specification.

append(*event:* maze.core.events.event_record.EventRecord)
Append a new event record to the collection.

extend(*event_list: Iterable[*maze.core.events.event_record.EventRecord*]*)
Extends self.events with a list of new event records.

query_events (*event_spec: Union[Callable, Iterable[Callable]]*) → Iterable

Return all events collected at the current env step matching one or more given event types. The event types are specified by the interface member function object itself.

Event calls are recorded as EventRecord, an object providing access to the passed arguments of the event method.

Parameters **event_spec** – Specifies the event type by the interface member function. Can either be a single event type specification or a list of specifications.

Returns An iterable to the event objects.

EventRecord

class maze.core.events.event_record.**EventRecord** (*interface_class: Type[abc.ABC], interface_method: Callable, attributes: dict*)

This auxiliary class is used to record calls to the event interface

Event Logging

These are the components of the event system:

StepEventLog	Logs all events dispatched by the environment during one step.
EpisodeEventLog	Keeps logs of all events dispatched by an environment during one episode.
KpiCalculator	Interface for calculating KPI metrics.
LogEventsWriterRegistry	Handles registration of event log writers.
LogEventsWriter	Interface for modules writing out the event log data.
LogEventsWriterTSV	Writes event logs into TSV files.
EventRow	Represents one row into the output file for the *LogEventsWriterTSV*.
SimpleEventLoggingSetup	Simple setup for logging of environment events with all their attributes.
ObservationEvents	Event topic class with logging statistics based only on observations, therefore applicable to any valid reinforcement learning environment.
DiscreteActionEvents	Event topic class with logging statistics based only on discrete (categorical) actions, therefore applicable to any valid reinforcement learning environment.
ContinuousActionEvents	Event topic class with logging statistics based only on continuous actions (box spaces), therefore applicable to any valid reinforcement learning environment.
create_categorical_plot	Checks the type of value and calls the correct plotting function accordingly.
create_histogram	Creates simple matplotlib histogram of value.
create_relative_bar_plot	Counts the categories in value and prepares a relative bar plot of these.
create_violin_distribution	Creates simple matplotlib violin plot of value.

StepEventLog

class maze.core.log_events.step_event_log.**StepEventLog**(*env_time: int, events: Optional[maze.core.events.event_collection.EventColle = None*)

Logs all events dispatched by the environment during one step.

Parameters

- **env_time** – Internal time of the environment, if available. Otherwise Step ID.
- **events** – Events dispatched by an environment during one particular step.

append(*event: maze.core.events.event_record.EventRecord*)

Append a new event record to the step log.

extend(*event_list: Iterable[maze.core.events.event_record.EventRecord]*)

Append a list of events record to the step log.

EpisodeEventLog

class maze.core.log_events.episode_event_log.**EpisodeEventLog**(*episode_id: str*)

Keeps logs of all events dispatched by an environment during one episode.

Parameters **episode_id** – ID of the episode the events belong to

query_events(*event_spec: Union[Callable, Iterable[Callable]]*) → Iterable

Query events across the whole episode.

Parameters **event_spec** – Specification of events to query

Returns List of events from this episode that

KpiCalculator

class maze.core.log_events.kpi_calculator.**KpiCalculator**

Interface for calculating KPI metrics. If available, is called by statistics wrapper at the end of each episode.

abstract calculate_kpis(*episode_event_log: maze.core.log_events.episode_event_log.EpisodeEventLog, last_maze_state: Any*) → Dict[str, float]

Compute KPIs for the current episode.

This is expected to be called once at the end of the episode, if statistics logging is enabled.

Parameters

- **episode_event_log** – Log of events recorded during the past episode.
- **last_maze_state** – State of the environment at the end of the episode

Returns Values of KPI metrics in the format {kpi_name: kpi_value}

LogEventsWriterRegistry

class maze.core.log_events.log_events_writer_registry.**LogEventsWriterRegistry**

Handles registration of event log writers.

Registered writers will be forwarded episode event log data at the end of each episode.

classmethod record_event_logs(*episode_event_log:* maze.core.log_events.episode_event_log.EpisodeEventLog) → None

Write event log data through all registered event log writers.

Parameters **episode_event_log** – Log of recorded environment events.

classmethod register_writer(*writer:* maze.core.log_events.log_events_writer.LogEventsWriter) → None

Register a writer. Each writer will receive all globally recorded event logs.

Parameters **writer** – Event log writer to register.

LogEventsWriter

class maze.core.log_events.log_events_writer.**LogEventsWriter**

Interface for modules writing out the event log data.

Implement this interface for any custom event data logging.

abstract write(*episode_event_log:* maze.core.log_events.episode_event_log.EpisodeEventLog) → None

Write out provided episode data (into a file, DB etc.)

Parameters **episode_event_log** – Log of the episode events.

LogEventsWriterTSV

class maze.core.log_events.log_events_writer_tsv.**LogEventsWriterTSV**(*log_dir: Union[str, pathlib.Path] = PosixPath('event_logs')*)

Writes event logs into TSV files. Each event type has its own file. Each event record has associated episode ID and step number.

Parameters **log_dir** – Where event logs should be logged.

write(*episode_event_log:* maze.core.log_events.episode_event_log.EpisodeEventLog) → None

Write out provided episode data in to TSV files.

EventRow

class maze.core.log_events.log_events_writer_tsv.**EventRow**(*episode_id: str, env_time: Optional[int], attributes: dict*)

Represents one row into the output file for the *LogEventsWriterTSV*.

The purpose of this class is to keep event record attributes together with its episode and step IDs.

Parameters

- **episode_id** – ID of the episode the event was generated in
- **env_time** – What time the event was generated in (either internal env time, or ID of the step)
- **attributes** – Event attributes dict

SimpleEventLoggingSetup

class maze.core.log_events.log_events_utils.**SimpleEventLoggingSetup**(*env:* maze.core.wrappers.log_stats_wr

Simple setup for logging of environment events with all their attributes.

Events will be logged into CSV files in "event_logs" directory.

Parameters **env** – Env to log events from (needs to be wrapper in LogEventsWrapper)

ObservationEvents

class maze.core.log_events.observation_events.**ObservationEvents**

Event topic class with logging statistics based only on observations, therefore applicable to any valid reinforcement learning environment.

observation_original(*step_key: str, name: str, value: int*)

observation seen and dimensionality of observation space

observation_processed(*step_key: str, name: str, value: int*)

observation seen and dimensionality of observation space

DiscreteActionEvents

class maze.core.log_events.action_events.**DiscreteActionEvents**

Event topic class with logging statistics based only on discrete (categorical) actions, therefore applicable to any valid reinforcement learning environment.

action(*substep: str, name: str, value: int, action_dim: int*)

action taken and dimensionality of discrete action space

ContinuousActionEvents

class `maze.core.log_events.action_events.`**`ContinuousActionEvents`**

Event topic class with logging statistics based only on continuous actions (box spaces), therefore applicable to any valid reinforcement learning environment.

`action`(*substep: str*, *name: str*, *value: int*)

action taken and shape of box action space

create_categorical_plot

class `maze.core.log_events.log_create_figure_functions.`**`create_categorical_plot`**(*value: Union[List[Tuple int]], List[int], List[float]]*)

Checks the type of value and calls the correct plotting function accordingly.

Parameters **`value`** – Output of an reducer function

Returns plt.figure that contains a bar plot

create_histogram

class `maze.core.log_events.log_create_figure_functions.`**`create_histogram`**(*value*)

Creates simple matplotlib histogram of value.

Parameters **`value`** – output of an event

Returns plt.figure that contains a bar plot

create_relative_bar_plot

class `maze.core.log_events.log_create_figure_functions.`**`create_relative_bar_plot`**(*value: List[Tuple[int, int]]*)

Counts the categories in value and prepares a relative bar plot of these.

Parameters **`value`** – List of Tuples of (action, action_dim)

Returns plt.figure that contains a bar plot

create_violin_distribution

class `maze.core.log_events.log_create_figure_functions.`**`create_violin_distribution`**(*value: List[numpy.r*

Creates simple matplotlib violin plot of value.

Parameters **`value`** – output of an event (expected to be a list of numpy vectors)

Returns plt.figure that contains a bar plot

Statistics Logging

These are the components of the statistics logging system:

LogStatsEnv	Interface to access logging statistics generated by the environment.
LogStatsWriterConsole	Log statistics writer implementation for the console, mainly for debugging purposes.
LogStatsWriterTensorboard	Log statistics writer implementation for Tensorboard.
LogStatsLevel	Log statistics aggregation levels.
LogStatsConsumer	An interface to receive log statistics.
LogStatsAggregator	Complements the event system by providing aggregation functionality.
LogStatsWriter	A minimal interface concrete log statistics writers must implement.
GlobalLogState	Internal class that encapsulates the global state of the logging system.
LogStatsLogger	Auxiliary class returned by get_stats_logger.
register_log_stats_writer	Set the concrete writer implementation that will receive all successive statistics logging.
log_stats	Helper function.
increment_log_step	Notifies the logging system that the current step is finished.
get_stats_logger	Creates an object that can be used to pipe LogStatAggregator instances with the logging writers.
define_step_stats	Event method decorator, defines a new step statistics calculation for this event.
define_episode_stats	Event method decorator, defines a new episode statistics calculation for this event.
define_epoch_stats	Event method decorator, defines a new epoch statistics calculation for this event.
define_stats_grouping	Event method decorator, defines a grouping of all calculated statistics by an attribute.
define_plot	Event method decorator, defines a plot.
histogram	the histogram reducer function
LogStatsValue	Basic data structure for log statistics
LogStatsGroup	Basic data structure for log statistics
LogStatsKey	Basic data structure for log statistics
LogStats	Basic data structure for log statistics

LogStatsEnv

class maze.core.log_stats.log_stats_env.**LogStatsEnv**

Interface to access logging statistics generated by the environment. Most envs won't implement this directly, but use the LogStatsWrapper to implement this functionality.

abstract get_stats(*level:* maze.core.log_stats.log_stats.LogStatsLevel) → *maze.core.log_stats.log_stats.LogStatsAggregator*

Get access to the statistics aggregator, which can be used to receive the latest statistics dictionary (via *last_stats()*) or to register log consumers.

Parameters level – The statistics aggregation level (LogStatsLevel.STEP, LogStatsLevel.EPISODE or LogStatsLevel.EPOCH)

Returns The statistics aggregator

abstract get_stats_value(*event: Callable*, *level: maze.core.log_stats.log_stats.LogStatsLevel*, *name: Optional[str] = None*) → Union[int, float, numpy.ndarray, dict]

Obtain a single value from the statistics dict.

Parameters

- **event** – The event interface method of the value in question
- **level** – The statistics aggregation level (LogStatsLevel.STEP, LogStatsLevel.EPISODE or LogStatsLevel.EPOCH)
- **name** – The *output_name* of the statistics in case it has been specified in `maze.core.log_stats.event_decorators.define_epoch_stats()`

Returns The statistics dictionary

abstract write_epoch_stats() → None

Use this if you do not want to wait for the next `maze.core.log_stats.log_stats.increment_log_step()` call to update the epoch statistics.

This can be useful in a training scenario to get the results of the evaluation rollouts immediately, instead of waiting for the next log step increment to occur.

LogStatsWriterConsole

class maze.core.log_stats.log_stats_writer_console.**LogStatsWriterConsole**

Log statistics writer implementation for the console, mainly for debugging purposes. Creates table-like console text in a fixed width layout.

write(*path: str*, *step: int*, *stats: Dict[maze.core.log_stats.log_stats.LogStatsKey, Union[int, float, numpy.ndarray, dict]]*) → None

see LogStatsWriter.write

LogStatsWriterTensorboard

class maze.core.log_stats.log_stats_writer_tensorboard.**LogStatsWriterTensorboard**(*log_dir: str*, *tensorboard_render_*
bool)

Log statistics writer implementation for Tensorboard. :param log_dir: log_dir for TensorBoard :param tensorboard_render_figure: Indicates whether to visualize the actions taken in TensorBoard.

on_log_step_increment()

Hooked into increment_log_step, called by the logging system immediately before the increment.

write(*path: str*, *step: int*, *stats: Dict[maze.core.log_stats.log_stats.LogStatsKey, Union[int, float, numpy.ndarray, dict]]*) → None

LogStatsWriter.write implementation

LogStatsLevel

class maze.core.log_stats.log_stats.**LogStatsLevel**(*value*)

Log statistics aggregation levels.

EPISODE = 2

aggregator receives step statistics and produces episode statistics

EPOCH = 3

aggregator receives episode statistics and produces epoch statistics

STEP = 1

aggregator receives individual events and produces step statistics

LogStatsConsumer

class maze.core.log_stats.log_stats.**LogStatsConsumer**

An interface to receive log statistics. Implemented by the LogStatAggregator class to receive logs from the subjacent aggregator.

abstract receive(*stat: Dict[*maze.core.log_stats.log_stats.LogStatsKey*, Union[*int*,* float*,* numpy.ndarray*,* dict*]]*)

Receive a statistics object from an aggregator. This might be called multiple times in case we consume statistics from multiple LogStatAggregator objects.

Parameters **stat** – The statistics dictionary

LogStatsAggregator

class maze.core.log_stats.log_stats.**LogStatsAggregator**(*level:* maze.core.log_stats.log_stats.LogStatsLevel, **consumers:* maze.core.log_stats.log_stats.LogStatsConsumer)

Complements the event system by providing aggregation functionality. How the events are aggregated is specified by the event interface decorators (see event_decorators.py).

Note that the statistics calculation for episode aggregators will automatically be triggered on every *increment_log_step()* call.

add_event(*event_record:* maze.core.events.event_record.EventRecord) → None

Add a recorded event to this aggregator.

The aggregator only keeps track of event/attributes with relevant statistics decoration, everything else is filtered out immediately.

Parameters **event_record** –

add_value(*event: Callable*, *value: Union[*int*,* float*,* numpy.ndarray*,* dict*]*, *name:* str *= None*, *group: Tuple[Union[*str*,* int*], . . .] = None*) → None

Add a single value to this aggregator.

Parameters

- **event** – The event interface method the given value belongs to
- **value** – The value to add
- **name** – There may be multiple statistics per event, the name is used to refer to a specific statistics built from the event records

- **group** – The group identifier if this event is grouped

create_event_topic(*interface_class: Type[T]*) → T

Provide an event topic proxy analogous to the event proxies provided by EventSystem/PubSub. But in contrast to the event system, this can be used to inject statistics also on the step, episode and epoch level.

Note that different LogStatsLevel result in different behaviour of the returned event topic proxies!

Parameters **interface_class** – The class object of an abstract interface that defines the events as methods.

Returns A proxy object, dynamically derived from the passed *interface_class*, that can be used to trigger events.

last_stats: Optional[*LogStats*]

keep track of the previous statistics, required e.g. for cumulative statistics

last_stats_step: Optional[int]

step number of the last statistics calculation

receive(*stats: Dict[maze.core.log_stats.log_stats.LogStatsKey, Union[int, float, numpy.ndarray, dict]]*) → None

Receive statistics from the subjacent aggregation level

Parameters **stats** – The statistics dictionary

reduce() → Dict[*maze.core.log_stats.log_stats.LogStatsKey*, Union[int, float, numpy.ndarray, dict]]

Consume the aggregated values by - calculating the statistics - sending the statistics to the consumers - resetting the values for the next aggregation step

:return Returns the statistics object. The same object has been sent to the consumers.

register_consumer(*consumer:* maze.core.log_stats.log_stats.LogStatsConsumer) → None

Register a new consumer to receive the readily calculated statistics of this aggregator.

Parameters **consumer** – The consumer to add

LogStatsWriter

class maze.core.log_stats.log_stats.**LogStatsWriter**

A minimal interface concrete log statistics writers must implement.

abstract write(*path: Optional[str], step: int, stats: Dict[maze.core.log_stats.log_stats.LogStatsKey, Union[int, float, numpy.ndarray, dict]]*) → None

Write the passed statistics dictionary to the log.

Parameters

- **path** – This can be a path-like string to organize the log into different sections.
- **step** – The step number associated with the passed statistics
- **stats** – The statistics dictionary

:return None

GlobalLogState

class maze.core.log_stats.log_stats.**GlobalLogState**

Internal class that encapsulates the global state of the logging system.

hook_on_log_step: List[Callable] = [<bound method LogStatsAggregator._hook_on_log_step

list of functions called on increment_log_step()

LogStatsLogger

class maze.core.log_stats.log_stats.**LogStatsLogger**(*path: Optional[str]*)

Auxiliary class returned by get_stats_logger.

receive(*stat: Dict[maze.core.log_stats.log_stats.LogStatsKey, Union[int, float, numpy.ndarray, dict]]*) → None

Implementation of LogStatsConsumer interface

register_log_stats_writer

class maze.core.log_stats.log_stats.**register_log_stats_writer**(*writer: maze.core.log_stats.log_stats.LogStatsWri*

Set the concrete writer implementation that will receive all successive statistics logging.

Parameters **writer** – The writer to be used by the logging system

log_stats

class maze.core.log_stats.log_stats.**log_stats**(*stats: Dict[maze.core.log_stats.log_stats.LogStatsKey, Union[int, float, numpy.ndarray, dict]], path: Optional[str]*)

Helper function.

Parameters

- **stats** – The statistics dictionary
- **path** – This can be a path-like string to organize the log into different sections.

increment_log_step

class maze.core.log_stats.log_stats.**increment_log_step**

Notifies the logging system that the current step is finished.

get_stats_logger

class maze.core.log_stats.log_stats.**get_stats_logger**(*path: Optional[str] = None*)

Creates an object that can be used to pipe LogStatAggregator instances with the logging writers.

Example usage: >>> logger = get_stats_logger("eval") >>> aggregator = LogStatsAggregator(LogStatsLevel.STEP, logger) >>> aggregator.reduce() # calculate the statistics and sent it to the registered logging writers

Parameters **path** – The optional path to prefix the logging tags

Returns

define_step_stats

class maze.core.log_stats.event_decorators.**define_step_stats**(*reduce_function: Optional[Callable], input_name: Optional[str] = None, output_name: Optional[str] = None, group_by: Optional[str] = None, cumulative: bool = False*)

Event method decorator, defines a new step statistics calculation for this event.

Input all events in a single step (and side-loaded step statistics, see 'reduce_function' set to None)

Output step statistics

Parameters

- **reduce_function** – A function that takes a list of values and returns the calculated statistics. In the special case that we do not want to calculate the statistics from events, but have the statistics result already available for the current step, the reduce_function can be set to None. Then the event can be invoked at most once per step to side-load the result. This is very useful to log state information, e.g. the inventory size.
- **input_name** – The name of the event attribute (=keyword attribute), whose values are to be passed to the reduce function. Can be omitted, for which there are two reasons
  - no naming necessary, there is only a single event attribute
  - we want all event attributes to be passed to the reduce_function as dictionaries (or our reduce function does not care, e.g. counting the number of events with *len*)
- **output_name** – The name of the statistics, how it should be passed to the logger resp. the following aggregation stage. Will be the same as input_name if None is provided.
- **group_by** – If there are multiple groups defined for an event, per default the statistics is collected at the 'cell' level. This option allows to project the statistics onto a single group (e.g. for inventory statistics we might define location and product as groups, but prefer to monitor statistics grouped only per product, regardless of location and vice versa)
- **cumulative** – Enable cumulative statistics

Returns The decorator function

define_episode_stats

class maze.core.log_stats.event_decorators.**define_episode_stats**(*reduce_function: Callable, input_name: Optional[str] = None, output_name: Optional[str] = None, group_by: Optional[str] = None, cumulative: bool = False*)

Event method decorator, defines a new episode statistics calculation for this event.

Input all step statistics in the current episode and side-loaded episode statistics, see 'reduce_function' set to None

Output episode statistics

Parameters

- **reduce_function** – A function that takes a list of values and returns the calculated statistics
- **input_name** – The name of the step statistics, whose values are to be passed to the reduce function. Can be omitted, for which there are two reasons
  - no naming necessary, there is only a single step statistics available
  - we want all step statistics to be passed to the reduce_function as dictionaries (or our reduce function does not care, e.g. counting with *len*)
- **output_name** – The name of the generated episode statistics, how it should be passed to the logger respective the following aggregation stage. Will be the same as input_name if None is provided.
- **group_by** – If there are multiple groups defined for an event, per default the statistics is collected at the 'cell' level. This option allows to project the statistics onto a single group (e.g. for inventory statistics we might define location and product as groups, but prefer to monitor statistics grouped only per product, regardless of location and vice versa)
- **cumulative** – Enable cumulative statistics

Returns The decorator function

define_epoch_stats

class `maze.core.log_stats.event_decorators.`**`define_epoch_stats`**(*reduce_function: Callable, input_name: Optional[str] = None, output_name: Optional[str] = None, group_by: Optional[str] = None, cumulative: bool = False*)

Event method decorator, defines a new epoch statistics calculation for this event.

Input All episode statistics of the current epoch and side-loaded epoch statistics, see 'reduce_function' set to None

Output Epoch statistics

Parameters

- **reduce_function** – A function that takes a list of values and returns the calculated statistics
- **input_name** – The name of the event attribute (=keyword attribute), whose values are to be passed to the reduce function. Can be omitted, for which there are two reasons
  - no naming necessary, there is only a single epoch statistics available
  - we want all episode statistics to be passed to the reduce_function as dictionaries (or our reduce function does not care, e.g. counting with *len*)
- **output_name** – The name of the generated epoch statistics, how it should be passed to the logger respective the following aggregation stage. Will be the same as input_name if None is provided.
- **group_by** – If there are multiple groups defined for an event, per default the statistics is collected at the 'cell' level. This option allows to project the statistics onto a single group (e.g. for inventory statistics we might define location and product as groups, but prefer to monitor statistics grouped only per product, regardless of location and vice versa)
- **cumulative** – Enable cumulative statistics

Returns The decorator function

define_stats_grouping

class `maze.core.log_stats.event_decorators.`**`define_stats_grouping`**(**group_by: str*)

Event method decorator, defines a grouping of all calculated statistics by an attribute.

Parameters **group_by** – Name of the event attribute(s)

Returns The decorator function

define_plot

class `maze.core.log_stats.event_decorators.`**`define_plot`**(*create_figure_function: Callable, input_name: str = None*)

Event method decorator, defines a plot. :return: The decorator function

histogram

class `maze.core.log_stats.reducer_functions.`**`histogram`**(*values: Union[List[Union[float, int]], List[List[Union[float, int]]], List[collections.abc.ValuesView]]*)

the histogram reducer function We decided to return the full list, rather then binning the values (e.g. collections.Counter), so that float values are supported as well.

Parameters **`values`** – A list of values collected by the event system

Returns returns the same list of values so that a histogram can then be build from it

LogStatsValue

`maze.core.log_stats.log_stats.`**`LogStatsValue`**

LogStatsGroup

`maze.core.log_stats.log_stats.`**`LogStatsGroup`**

LogStatsKey

class `maze.core.log_stats.log_stats.`**`LogStatsKey`**(*event, output_name, group*)

Basic data structure for log statistics

property event

Alias for field number 0

property group

Alias for field number 2

property output_name

Alias for field number 1

LogStats

`maze.core.log_stats.log_stats.LogStats`

1.4.4 Rendering

These are interfaces, classes and utility functions for the rendering system:

Renderer	Interface for renderers of individual environments.
StepStatsRenderer	Simple statistics rendering based on episode event logs.
EventStatsRenderer	Renders customizable statistics on top of event logs.
NotebookEventLogsViewer	Event logs viewer for Jupyter Notebooks, built using ipython widgets.
NotebookTrajectoryViewer	Trajectory viewer for Jupyter Notebooks, built using ipython widgets.
KeyboardControlledTrajectoryViewer	Render trajectory data with the possibility to browse back and forward through the episode steps using keyboard.
RendererArg	Interface for classes exposing arguments available at renderers.
IntRangeArg	Represents an argument which can take on a value of integer in a particular range.
OptionsArrayArg	Represents an argument where a single value can be chosen from an array of allowed options.

Renderer

class maze.core.rendering.renderer.**Renderer**

Interface for renderers of individual environments.

Renders state of one particular step – based only on current state.

static arguments() → List[*maze.core.rendering.renderer_args.RendererArg*]

List the additional arguments that the renderer supports (beyond maze_state and maze_action), if any.

Exposing available argument options like this makes it possible to create appropriate user controls when controlling the renderer in interactive settings (e.g., using widgets in a Jupyter Notebook).

Note:

Note that the types and names of arguments returned are expected to be the same across all possible env configurations. What can change are the available values, which then are expected to stay the same for a whole episode.

Example of this are drivers in a vehicle routing env. If you would like to display a detail of the driver, a driver_id argument can be exposed. It will always be named *driver_id* and be of the same type, but across different episodes, the number of drivers (i.e. allowed range of the argument) might differ. It will always stay fixed during a whole episode though.

Returns List of renderer argument objects.

abstract render(*maze_state: Any*, *maze_action: Optional[Any]*, *events:* maze.core.log_events.step_event_log.StepEventLog, ***kwargs*) → None

Render the current state as a matplotlib figure.

Note that the maze_action is optional – it is None for the last (terminal) state in the episode!

Parameters

- **maze_state** – State to render.
- **maze_action** – MazeAction to render. Should be the MazeAction derived from the state to render (provided above)
- **events** – Events dispatched by the env during the last step (i.e. when the given state was produced)
- **kwargs** – Any additional arguments that the renderer accepts and exposes

StepStatsRenderer

class maze.core.rendering.step_stats_renderer.**StepStatsRenderer**

Simple statistics rendering based on episode event logs.

Suitable e.g. for ad-hoc plotting of statistics for the current episode during rollout.

static render_stats(*episode_event_log: maze.core.log_events.episode_event_log.EpisodeEventLog, event_name: str = 'BaseEnvEvents.reward', group_by: Optional[str] = None, aggregation_func: Optional[Callable] = None, metric: str = 'value', cumulative: bool = True*)

Queries the event log for the given events, then aggregates them and plots them according to the provided options. By default, a cumulative reward is plotted.

Parameters

- **episode_event_log** – The episode event log to draw events from.
- **event_name** – Name of the event to plot.
- **group_by** – Attribute of the event that the events should be grouped by when aggregating.
- **aggregation_func** – Function to aggregate the metric with.
- **metric** – The metric to plot.
- **cumulative** – If true, a cumulative sum of the metric is performed (after aggregation).

EventStatsRenderer

class maze.core.rendering.events_stats_renderer.**EventStatsRenderer**

Renders customizable statistics on top of event logs.

This renderer provides a central rendering functionality for event log data. Elementary customizability is offered (e.g. simple aggregation etc.). For more complex operations with the data, it is advised to work with the TSV event logs directly.

AGGREGATION_FUNCS = ['mean', 'sum', 'min', 'max', 'count']

Aggregation functions to offer to the user. Recognized as strings by pandas.

POST_PROCESSING_FUNCS = ['cumsum']

Post-processing functions to offer to the user. Recognized as strings by pandas.

close()

Close the stats figure if one has been created.

render_current_episode_stats(*episode_event_log: maze.core.log_events.episode_event_log.EpisodeEventLog, event_name: str = 'BaseEnvEvents.reward', metric_name: str = 'value', aggregation_func: Optional[Union[str, Callable]] = None, group_by: str = None, post_processing_func: Optional[Union[str, Callable]] = 'cumsum'*)

Render event stats from episode log of currently running episode.

Creates a new figure if needed.

Parameters

- **episode_event_log** – Episode event log to render events from
- **event_name** – Name of the even the even log corresponds to
- **metric_name** – Metric to use (one of the event attributes, e.g. “n_items” – depends on the event type)
- **aggregation_func** – Optionally, specifies how to aggregate the metric on step level, i.e. when there are multiple same events dispatched during the same step.
- **group_by** – Optionally, another of event attributes to group by on the step level (e.g. “product_id”)
- **post_processing_func** – Optionally, a function to post-process the data (“cumsum” is often used)

static render_timeline_stat(*df: pandas.DataFrame, event_name: str = 'BaseEnvEvents.reward', metric_name: str = 'value', aggregation_func: Optional[Union[str, Callable]] = None, group_by: str = None, post_processing_func: Optional[Union[str, Callable]] = 'cumsum'*)

Render event statistics from a data frame according to the supplied options.

Does not create a figure, renders into the currently active ax.

Parameters

- **df** – Event log to render statistics from
- **event_name** – Name of the even the even log corresponds to
- **metric_name** – Metric to use (one of the event attributes, e.g. “n_items” – depends on the event type)
- **aggregation_func** – Optionally, specifies how to aggregate the metric on step level, i.e. when there are multiple same events dispatched during the same step.
- **group_by** – Optionally, another of event attributes to group by on the step level (e.g. “product_id”)
- **post_processing_func** – Optionally, a function to post-process the data (“cumsum” is often used)

NotebookEventLogsViewer

class maze.core.rendering.notebook_event_logs_viewer.**NotebookEventLogsViewer**(*event_logs_dir_path*
Union[str, path-lib.Path])

Event logs viewer for Jupyter Notebooks, built using ipython widgets.

Usage: Inside a Jupyter Notebook, initialize the viewer with path to event logs directory, then call *build()*.

This viewer offers elementary rendering functionality for event logs collected during environment rollout. Event logs are expected to be passed in as a set of TSV with filenames corresponding to event names, the default format as written out using the *LogEventsWriterTSV*.

The logs will be passed and a set of widgets will be shown, offering options on which event to display and what event attribute to use as a metric.

Statics are always aggregated on episode level – the timeline displays mean value along with standard deviation (displayed as a ribbon).

Optionally, events can be grouped by another attribute (e.g., distribution center ID in multi-echelon inventory environment) and aggregated on step level – this way, we can show e.g. mean value of items stored by each distribution center across all the episodes. This can be configured using the widgets as well.

Param event_logs_dir_path: Path to directory where the event logs are stored.

build() → None
Build the interactive viewer. Expected to be called in a Jupyter notebook after initialization.

render(*event_path*, ***kwargs*) → None
Refresh the rendered view. Called by ipywidgets on widget update.

update_column_options(*metadata: Dict[str, Any]*)
Called by ipywidgets interact module when the user selects a different event to display. Loads attributes available for this event and updates the metric and group_by widget options accordingly.

NotebookTrajectoryViewer

class maze.core.rendering.notebook_trajectory_viewer.**NotebookTrajectoryViewer**(*episode_record:*
maze.core.trajecto

Trajectory viewer for Jupyter Notebooks, built using ipython widgets.

Displays trajectory data for the given episode as an interactive view, where the step ID and any additional arguments exposed by the renderer can be interactively set. The data is rendered using the renderer recorded in the episode record.

Parameters **episode_record** – Trajectory data to render.

build() → None
Build and show the interactive widgets.

Builds all the widgets (one for step ID, then one for each additional argument accepted by the renderer) and activates them using the interact function. Expected to be called from a cell in a Jupyter notebook.

render(*step_id*, ***kwargs*) → None
Render the view for the given step ID, with the given additional parameters.

Usually, this method is not called directly – it is expected to be called by ipython widgets.

Parameters

- **step_id** – ID of the step to display

- **kwargs** – Any additional arguments the renderer accepts

KeyboardControlledTrajectoryViewer

class maze.core.rendering.keyboard_controlled_trajectory_viewer.**KeyboardControlledTrajecto**

Render trajectory data with the possibility to browse back and forward through the episode steps using keyboard.

This is the simplest form of interactive rendering of episode trajectory, useful for example when rendering the trajectory ad hoc while the environment is still running. For more comfortable rendering of trajectory data inside of a Jupyter Notebook, use `NotebookTrajectoryViewer`.

Note:

The keyboard controls might not work well when run outside of terminal.

If running this through PyCharm, the "Emulate terminal in output console" options in Run/Debug configurations needs to be set to true, otherwise the keys will not be picked up correctly and this run loop will crash.

Also, the console needs to be active dor the keys to be picked up.

render()

Run the interactive rendering loop. Waits for user input (right or left arrow), updates the step index accordingly and triggers the redraw through the renderer.

RendererArg

class maze.core.rendering.renderer_args.**RendererArg**(*name: str*, *title: str*)

Interface for classes exposing arguments available at renderers.

Example such argument: ID of a distribution center that we want to render a detail of.

Subclasses of this class should also define how to convert these argument definitions into interactive controls that can be displayed to the user (currently only ipython widgets, though more types of controls can be added in the future).

Parameters

- **name** – Name of the argument as it should be passed to the renderer
- **title** – Name of the argument as it should be displayed to the user

abstract create_widget()

Build an ipython widget that can be used to control the value of this argument by the user.

IntRangeArg

class maze.core.rendering.renderer_args.**IntRangeArg**(*name: str*, *title: str*, *min_value: int*, *max_value: int*)

Represents an argument which can take on a value of integer in a particular range.

Parameters

- **min_value** – Min allowed value
- **max_value** – Max allowed value

create_widget()

Build an int slider widget.

OptionsArrayArg

class maze.core.rendering.renderer_args.**OptionsArrayArg**(*name: str*, *title: str*, *options: List[Union[Any, Tuple[str, Any]]]*)

Represents an argument where a single value can be chosen from an array of allowed options.

Parameters **options** – Array of allowed options. Either just a simple array of allowed argument values, or an array of tuples, each in the form of *(value_displayed_to_the_user, value_passed_to_renderer)*

create_widget()

Build a dropdown widget.

1.4.5 Trajectory Recorder

These are interfaces, classes and utility functions for recording trajectory data:

TrajectoryWriterRegistry	Handles registration of trajectory data writers.
StepRecord	Keeps trajectory data for one step.
EpisodeRecord	Records and keeps trajectory record data for a complete episode.
TrajectoryWriter	Interface for modules serializing the trajectory data.
TrajectoryWriterFile	Simple trajectory data writer.
SimpleTrajectoryRecordingSetup	Simple setup for trajectory data recording.
MonitoringSetup	Simple setup for environment monitoring.

TrajectoryWriterRegistry

class maze.core.trajectory_recorder.trajectory_writer_registry.**TrajectoryWriterRegistry**

Handles registration of trajectory data writers. Registered writers will be forwarded episode trajectory data at the end of each episode.

classmethod record_trajectory_data(*episode_record:* maze.core.trajectory_recorder.episode_record.EpisodeReco
→ None

Record trajectory data through all registered trajectory data writers.

Parameters **episode_record** – Record of episode trajectory data to log.

classmethod register_writer(*writer:* maze.core.trajectory_recorder.trajectory_writer.TrajectoryWriter)
→ None

Register a writer. Each writer will receive all globally recorded trajectory data.

Parameters **`writer`** – Trajectory writer to register.

StepRecord

class `maze.core.trajectory_recorder.step_record.`**`StepRecord`**(*maze_state: Any, maze_action: Optional[Any], step_event_log:* maze.core.log_events.step_event_log.StepEven *reward: Union[float, numpy.ndarray, Any], done: Optional[bool], info: Optional[Dict], serializable_components: Dict[str, Any]*)

Keeps trajectory data for one step. Note: It should be ensured that the components are not going to change after assigning them to the step record (e.g. by copying the relevant ones, especially state and the serializable components).

Parameters

- **`maze_state`** – Current MazeState of the env.
- **`maze_action`** – Last MazeAction taken by the agent.
- **`step_event_log`** – Log of events dispatched by the env during the last step.
- **`reward`** – Reward as returned by the environment (either scalar or distributed reward)
- **`done`** – Dictionary indicating whether the environment or particular agents are done
- **`info`** – Dictionary with any other supplementary information provided by the env
- **`serializable_components`** – dict of all serializable components as provided by the env - e.g. { “demand_generator” : demand_generator_object }

property env_time
Internal time of environment (if available) that this record belongs to.

property step_id
ID of the step this record belongs to.

EpisodeRecord

class `maze.core.trajectory_recorder.episode_record.`**`EpisodeRecord`**(*episode_id: str, renderer: Optional[*maze.core.rendering.renderer.R *= None*)

Records and keeps trajectory record data for a complete episode.

Parameters

- **`episode_id`** – ID of the episode. Can be used to link trajectory data from event logs and other sources.

- **`renderer`** – Where available, the renderer object should be associated to the episode record. This ensures correct configuration of the renderer (with respect to env configuration for this episode), and makes it easier to instantiate the correct renderer for displaying the trajectory data.

TrajectoryWriter

class `maze.core.trajectory_recorder.trajectory_writer.`**`TrajectoryWriter`**

Interface for modules serializing the trajectory data.

Implement this interface for any custom trajectory data logging.

abstract write(*episode_record: maze.core.trajectory_recorder.episode_record.EpisodeRecord*) → None

Write out provided episode data (into a file, DB etc.)

Parameters **`episode_record`** – Record of the episode trajectory data.

TrajectoryWriterFile

class `maze.core.trajectory_recorder.trajectory_writer_file.`**`TrajectoryWriterFile`**(*log_dir: Union[str, pathlib.Path] = PosixPath('trajectory*

Simple trajectory data writer. Serializes trajectory data for each episode into a separate file using Pickle.

Suitable for smaller scale rollouts or debugging.

Parameters **`log_dir`** – Where trajectory data should be logged.

write(*episode_record: maze.core.trajectory_recorder.episode_record.EpisodeRecord*) → None

Write episode trajectory data to a file using pickle.

Parameters **`episode_record`** – Episode trajectory data

SimpleTrajectoryRecordingSetup

class `maze.core.trajectory_recorder.trajectory_utils.`**`SimpleTrajectoryRecordingSetup`**(*env*)

Simple setup for trajectory data recording.

Trajectory data will be serialized into *trajectory_data* directory, one file per episode.

MonitoringSetup

class `maze.core.trajectory_recorder.monitoring_setup.`**`MonitoringSetup`**(*env: T, log_dir: str = '.'*)

Simple setup for environment monitoring.

Logs the following data:

- Epoch statistics (console + Tensorboard format)

- Environment events (TSV files, one per event type)
- Trajectory data

1.4.6 General and Rollout Runners

This page contains the reference documentation for all kinds of runners.

Overview

- *General Runners*
- *Rollout Runners*

General Runners

These are the basic interfaces, classes and utility functions of runners:

`Runner`	Runner interface for running Maze from CLI.
`maze_run`	Run a CLI task based on the provided configuration.

Runner

class maze.runner.**Runner**

Runner interface for running Maze from CLI.

This class will be instantiated from the config obtained from hydra (cfg.runner). Then, the run method will be called, being supplied the whole hydra config (cfg).

abstract run(*cfg: omegaconf.DictConfig*) → None

Perform the run.

Parameters **cfg** – Config of the hydra job.

maze_run

class maze.maze_cli.**maze_run**(*cfg: omegaconf.DictConfig*)

Run a CLI task based on the provided configuration.

A runner object is instantiated according to the config (cfg.runner) and it is then handed the whole configuration object (cfg). Runners can perform various tasks such as rollouts, trainings etc.

Parameters **cfg** – Hydra configuration for the rollout.

Rollout Runners

These are interfaces, classes and utility functions for rollout runners:

Here can find the documentation for training runners.

`RolloutRunner`	General abstract class for rollout runners.
`SequentialRolloutRunner`	Runs rollout in the local process.
`ParallelRolloutRunner`	Runs rollout in multiple processes in parallel.
`ParallelRolloutWorker`	Class encapsulating functionality performed in worker processes.
`EpisodeRecorder`	Keeps the statistics and event logs from the last episode so that it can then be shipped to the main process.
`EpisodeStatsReport`	Tuple for passing episode stats from workers to the main process.
`ExceptionReport`	Tuple for passing error reports from the workers to the main process.

RolloutRunner

class `maze.core.rollout.rollout_runner.`**`RolloutRunner`**(*n_episodes: int, max_episode_steps: int, record_trajectory: bool, record_event_logs: bool*)

General abstract class for rollout runners.

Offers general structure, plus a couple of helper methods for env instantiation and performing the rollout.

Parameters

- **`n_episodes`** – Count of episodes to run
- **`max_episode_steps`** – Count of steps to run in each episode (if environment returns done, the episode will be finished earlier though)
- **`record_trajectory`** – Whether to record trajectory data
- **`record_event_logs`** – Whether to record event logs

static init_env_and_agent(*env_config: omegaconf.DictConfig, wrappers_config: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]], max_episode_steps: int, agent_config: omegaconf.DictConfig, input_dir: str) -> (<class 'maze.core.env.base_env.BaseEnv'>, <class 'maze.core.agent.policy.Policy'>)*)

Build the environment (including wrappers) and agent according to given configuration.

Parameters

- **`env_config`** – Environment config
- **`wrappers_config`** – Wrapper config
- **`max_episode_steps`** – Max number of steps per episode to limit the env for
- **`agent_config`** – Policies config
- **`input_dir`** – Directory to load the model from

Returns Tuple of (instantiated environment, instantiated agent)

run(*cfg: omegaconf.DictConfig*) → None

Parse the supplied Hydra config and perform the run.

static run_interaction_maze(*env:* maze.core.env.structured_env.StructuredEnv, *agent:* maze.core.agent.policy.Policy, *n_episodes: int*, *render: bool = False*, *episode_end_callback: Callable = None*) → None

Helper function for running the agent-environment interaction loop for specified number of steps and episodes.

Parameters

- **env** – Environment to run
- **agent** – Agent to use
- **n_episodes** – Count of episodes to perform
- **render** – Whether to render the environment after every step
- **episode_end_callback** – If supplied, this will be executed after each episode to notify the observer

abstract run_with(*env: Union[None, str, Mapping[str, Any], Any]*, *wrappers: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]]*, *agent: Union[None, str, Mapping[str, Any], Any]*) → None

Run the rollout with the given env, wrappers and agent configuration. A helper method to make rollouts easily runnable also directly from python, without building the hydra config object.

Note that this method is designed to run only once – if you call it from python directly (and not using Hydra from command line as is the main use case), you should respect this. Otherwise, you might get weird behavior especially from the statistics and events logging system, as the rollout runners register their own stats and event writers (so you might get duplicate stats) and order of operations sometimes matters (especially with parallel rollouts, where we do not want to carry the writers into child processes).

Parameters

- **env** – Env config or object
- **wrappers** – Wrappers config (see `WrapperRegistry`)
- **agent** – Agent config or object

SequentialRolloutRunner

class maze.core.rollout.sequential_rollout_runner.**SequentialRolloutRunner**(*n_episodes: int*, *max_episode_steps: int*, *record_trajectory: bool*, *record_event_logs: bool*, *render: bool*)

Runs rollout in the local process. Useful for short rollouts or debugging.

Trajectory, event logs and stats are recorded into the working directory managed by hydra (provided that the relevant wrappers are present.)

Parameters

- **`n_episodes`** – Count of episodes to run
- **`max_episode_steps`** – Count of steps to run in each episode (if environment returns done, the episode will be finished earlier though)
- **`record_trajectory`** – Whether to record trajectory data
- **`record_event_logs`** – Whether to record event logs

`run_with`(*env: Union[None, str, Mapping[str, Any], Any], wrappers: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]], agent: Union[None, str, Mapping[str, Any], Any]*)

Run the rollout sequentially in the main process.

`update_progress`()

Called on episode end to update a simple progress indicator.

ParallelRolloutRunner

class `maze.core.rollout.parallel_rollout_runner.`**`ParallelRolloutRunner`**(*n_episodes: int, max_episode_steps: int, n_processes: int, record_trajectory: bool, record_event_logs: bool*)

Runs rollout in multiple processes in parallel.

Both agent and environment are run in multiple instances across multiple processes. While this greatly speeds up the rollout, the memory consumption might be high for large environments and agents.

Trajectory recording, event logging, as well as stats logging are supported. Trajectory logging happens in the child processes. Event logs and stats are shipped back to the main process so that they can be handled together there. This allows monitoring of progress and calculation of summary stats across all the processes.

(Note that the relevant wrappers need to be present in the config for the trajectory/event/stats logging to work. Data are logged into the working directory managed by hydra.)

In case of early rollout termination using a keyboard interrupt, data for all episodes completed till that point will be preserved (= written out). Graceful shutdown will be attempted, including calculation of statistics across the episodes completed before the rollout was terminated.

Parameters

- **`n_episodes`** – Count of episodes to run
- **`max_episode_steps`** – Count of steps to run in each episode (if environment returns done, the episode will be finished earlier though)
- **`n_processes`** – Count of processes to spread the rollout across.
- **`record_trajectory`** – Whether to record trajectory data
- **`record_event_logs`** – Whether to record event logs

run_with(*env: Union[None, str, Mapping[str, Any], Any], wrappers: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]], agent: Union[None, str, Mapping[str, Any], Any]*)
Run the parallel rollout in multiple worker processes.

ParallelRolloutWorker

class maze.core.rollout.parallel_rollout_runner.**ParallelRolloutWorker**
Class encapsulating functionality performed in worker processes.

static run(*env_config: omegaconf.DictConfig, wrapper_config: omegaconf.DictConfig, agent_config: omegaconf.DictConfig, n_episodes: int, max_episode_steps: int, record_trajectory: bool, input_directory: str, reporting_queue: multiprocessing.context.BaseContext.Queue*) → None
Build the environment and run the rollout for the specified number of episodes.

Parameters

- **env_config** – Hydra configuration of the environment to instantiate.
- **wrapper_config** – Hydra configuration of environment wrappers.
- **agent_config** – Hydra configuration of agent's policies.
- **n_episodes** – Number of episodes to run (in total, will be split across processes)
- **max_episode_steps** – Max number of steps per episode to perform (episode might end earlier if env returns done)
- **record_trajectory** – Whether to record trajectory data.
- **input_directory** – Directory to load the model from.
- **reporting_queue** – Queue for passing the stats and event logs back to the main process after each episode

EpisodeRecorder

class maze.core.rollout.parallel_rollout_runner.**EpisodeRecorder**
Keeps the statistics and event logs from the last episode so that it can then be shipped to the main process.

get_last_episode_data() → Tuple[Dict[*maze.core.log_stats.log_stats.LogStatsKey*, Union[int, float, numpy.ndarray, dict]], *maze.core.log_events.episode_event_log.EpisodeEventLog*]
Get the stats and event log from the last episode.

Returns Tuple of (episode stats, event log)

receive(*stat: Dict[*maze.core.log_stats.log_stats.LogStatsKey*, Union[int, float, numpy.ndarray, dict]]*) → None
Receive the statistics from the env and store them.

write(*episode_event_log:* maze.core.log_events.episode_event_log.EpisodeEventLog) → None
Receive the event logs from the env and store them.

EpisodeStatsReport

class `maze.core.rollout.parallel_rollout_runner.`**`EpisodeStatsReport`**(*stats*, *event_log*)

Tuple for passing episode stats from workers to the main process.

property event_log
Alias for field number 1

property stats
Alias for field number 0

ExceptionReport

class `maze.core.rollout.parallel_rollout_runner.`**`ExceptionReport`**(*exception*, *traceback*)

Tuple for passing error reports from the workers to the main process.

property exception
Alias for field number 0

property traceback
Alias for field number 1

1.4.7 Policies, Critics and Agents

This page contains the reference documentation for policies, critics and agents.

maze.core.agent

Policies:

FlatPolicy	Generic flat policy interface.
Policy	Structured policy class designed to work with structured environments.
TorchPolicy	Encapsulates multiple torch policies along with a distribution mapper for training and rollouts in structured environments.
DefaultPolicy	Encapsulates one or more policies identified by policy IDs.
RandomPolicy	Implements a random structured policy.
DummyCartPolePolicy	Dummy structured policy for the CartPole env.
SerializedTorchPolicy	Structured policy used for rollouts of trained models.

FlatPolicy

class `maze.core.agent.flat_policy.`**`FlatPolicy`**

Generic flat policy interface.

abstract compute_action(*observation: Dict[str, numpy.ndarray], deterministic: bool*) → Dict[str, Union[int, numpy.ndarray]]

Pick the next action based on the current observation.

Parameters

- **`observation`** – Current observation of the environment
- **`deterministic`** – Specify if the action should be computed deterministically

Returns Next action to take

abstract compute_top_action_candidates(*observation: Dict[str, numpy.ndarray], num_candidates: int*) → Tuple[Sequence[Dict[str, Union[int, numpy.ndarray]]], Sequence[float]]

Get the top :num_candidates actions as well as the probabilities, q-values, .. leading to the decision.

Parameters

- **`observation`** – Current observation of the environment
- **`num_candidates`** – The number of actions that should be returned

Returns a tuple of sequences, where the first sequence corresponds to the possible actions, the other sequence to the associated probabilities

Policy

class `maze.core.agent.policy.`**`Policy`**

Structured policy class designed to work with structured environments. (see *`StructuredEnv`*).

It encapsulates policies and queries them for actions according to the provided policy ID.

abstract compute_action(*observation: Dict[str, numpy.ndarray], maze_state: Optional[Any], policy_id: Union[str, int] = None, deterministic: bool = False*) → Dict[str, Union[int, numpy.ndarray]]

Query a policy that corresponds to the given ID for action.

Parameters

- **`observation`** – Current observation of the environment
- **`maze_state`** – Current state representation of the environment (only provided if *needs_state()* returns True)
- **`policy_id`** – ID of the policy to query (does not have to be provided if policies dict contains only 1 policy)
- **`deterministic`** – Specify if the action should be computed deterministically

Returns Next action to take

abstract compute_top_action_candidates(*observation: Dict[str, numpy.ndarray], num_candidates: int, maze_state: Optional[Any], policy_id: Union[str, int] = None, deterministic: bool = False*) → Tuple[Sequence[Dict[str, Union[int, numpy.ndarray]]], Sequence[float]]

Get the top :num_candidates actions as well as the probabilities, q-values, .. leading to the decision.

Parameters

- **observation** – Current observation of the environment
- **num_candidates** – The number of actions that should be returned
- **maze_state** – Current state representation of the environment (only provided if *needs_state()* returns True)
- **policy_id** – ID of the policy to query (does not have to be provided if policies dict contains only 1 policy)
- **deterministic** – Specify if the action should be computed deterministically

Returns a tuple of sequences, where the first sequence corresponds to the possible actions, the other sequence to the associated scores (e.g, probabilities or Q-values).

abstract needs_state() → bool

The policy implementation declares if it operates solely on observations (needs_state returns False) or if it also requires the state object in order to compute the action.

Note that requiring the state object comes with performance implications, especially in multi-node distributed workloads, where both objects would need to be transferred over the network.

TorchPolicy

class maze.core.agent.torch_policy.**TorchPolicy**(*networks: Mapping[Union[str, int], torch.nn.Module], distribution_mapper: maze.distributions.distribution_mapper.DistributionMapper, device: str*)

Encapsulates multiple torch policies along with a distribution mapper for training and rollouts in structured environments.

Parameters

- **networks** – Mapping of policy networks to encapsulate
- **distribution_mapper** – Distribution mapper associated with the policy mapping.
- **device** – Device the policy should be located on (cpu or cuda)

compute_action(*observation: Dict[str, numpy.ndarray], maze_state: Optional[Any] = None, policy_id: Union[str, int] = None, deterministic: bool = False*) → Dict[str, Union[int, numpy.ndarray]]

implementation of `Policy`

compute_action_distribution(*observation: Any, policy_id: Union[str, int] = None*) → Any

Query the policy corresponding to the given ID for the action distribution.

compute_action_logits_entropy_dist(*policy_id: Union[str, int], observation: Dict[Union[str, int], torch.Tensor], deterministic: bool, temperature: float*) → Tuple[Dict[str, torch.Tensor], Dict[str, torch.Tensor], torch.Tensor, *maze.distributions.dict.DictProbabilityDistribution*]

Compute action for the given observation and policy ID and return it together with the logits.

Parameters

- **`policy_id`** – ID of the policy to query (does not have to be provided if policies dict contain only 1 policy
- **`observation`** – Current observation of the environment
- **`deterministic`** – Specify if the action should be computed deterministically
- **`temperature`** – Controls the sampling behaviour. * 1.0 corresponds to unmodified sampling * smaller than 1.0 concentrates the action distribution towards deterministic sampling

Returns Tuple of (action, logits_dict, entropy, prob_dist)

`compute_action_with_logits`(*observation: Any*, *policy_id: Union[str, int] = None*, *deterministic: bool = False*) → Tuple[Any, Dict[str, torch.Tensor]]

Compute action for the given observation and policy ID and return it together with the logits.

Parameters

- **`observation`** – Current observation of the environment
- **`policy_id`** – ID of the policy to query (does not have to be provided if policies dict contain only 1 policy
- **`deterministic`** – Specify if the action should be computed deterministically

Returns Tuple of (action, logits_dict)

`compute_logits_dict`(*observation: Any*, *policy_id: Union[str, int] = None*) → Dict[str, torch.Tensor]

Get the logits for the given observation and policy ID.

Parameters

- **`observation`** – Observation to return probability distribution for
- **`policy_id`** – Policy ID this observation corresponds to

Returns Logits dictionary

`compute_top_action_candidates`(*observation: Dict[str, numpy.ndarray]*, *num_candidates: int*, *maze_state: Optional[Any] = None*, *policy_id: Union[str, int] = None*, *deterministic: bool = False*) → Tuple[Sequence[Dict[str, Union[int, numpy.ndarray]]], Sequence[float]]

implementation of `Policy`

`eval`() → None

implementation of `TorchModel`

`load_state_dict`(*state_dict: Dict*) → None

implementation of `TorchModel`

`logits_dict_to_distribution`(*logits_dict: Dict[str, torch.Tensor]*, *temperature: float = 1.0*)

Helper function for creation of a dict probability distribution from the given logits dictionary.

Parameters

- **`logits_dict`** – A logits dictionary [action_head: action_logits] to parameterize the distribution from.

- **`temperature`** – Controls the sampling behaviour. * 1.0 corresponds to unmodified sampling * smaller than 1.0 concentrates the action distribution towards deterministic sampling

Returns (DictProbabilityDistribution) the respective instance of a DictProbabilityDistribution.

`needs_state`() → bool
This policy does not require the state() object to compute the action.

`parameters`() → List[torch.Tensor]
implementation of *`TorchModel`*

`state_dict`() → Dict
implementation of *`TorchModel`*

`to`(*device: str*) → None
implementation of *`TorchModel`*

`train`() → None
implementation of *`TorchModel`*

DefaultPolicy

class `maze.core.agent.default_policy.`**`DefaultPolicy`**(*policies: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]]*)

Encapsulates one or more policies identified by policy IDs.

Parameters **`policies`** – Dict of policy IDs and corresponding policies.

`compute_action`(*observation: Dict[str, numpy.ndarray], maze_state: Optional[Any] = None, policy_id: Union[str, int] = None, deterministic: bool = False*) → Dict[str, Union[int, numpy.ndarray]]
implementation of *`Policy`* interface

`compute_top_action_candidates`(*observation: Dict[str, numpy.ndarray], num_candidates: int, maze_state: Optional[Any] = None, policy_id: Union[str, int] = None, deterministic: bool = False*) → Tuple[Sequence[Dict[str, Union[int, numpy.ndarray]]], Sequence[float]]
implementation of *`Policy`* interface

`needs_state`() → bool
This policy does not require the state() object to compute the action.

RandomPolicy

class `maze.core.agent.random_policy.`**`RandomPolicy`**(*action_spaces_dict: Dict[Union[str, int], gym.spaces.Space]*)

Implements a random structured policy.

Parameters **`action_spaces_dict`** – The action_spaces dict from the env

`compute_action`(*observation: Dict[str, numpy.ndarray], maze_state: Optional[Any], policy_id: Union[str, int] = None, deterministic: bool = False*) → Dict[str, Union[int, numpy.ndarray]]
Query a policy that corresponds to the given ID for action.

Parameters

- **`observation`** – Current observation of the environment
- **`maze_state`** – Current state of the environment (will always be None as *needs_state()* returns False)
- **`policy_id`** – ID of the policy to query (does not have to be provided if policies dict contain only 1 policy
- **`deterministic`** – Specify if the action should be computed deterministically

Returns Next action to take

`compute_top_action_candidates`(*observation: Dict[str, numpy.ndarray]*, *num_candidates: int*, *maze_state: Optional[Any] = None*, *policy_id: Union[str, int] = None*, *deterministic: bool = False*) → Tuple[Sequence[Dict[str, Union[int, numpy.ndarray]]], Sequence[float]]

implementation of `Policy` interface

`needs_state`() → bool

This policy does not require the state() object to compute the action.

DummyCartPolePolicy

`class` `maze.core.agent.dummy_cartpole_policy.`**`DummyCartPolePolicy`**

Dummy structured policy for the CartPole env.

Useful mainly for showcase of the config scheme and for testing.

`compute_action`(*observation: Dict[str, numpy.ndarray]*, *maze_state: Optional[Any] = None*, *policy_id: Union[str, int] = None*, *deterministic: bool = False*) → Dict[str, Union[int, numpy.ndarray]]

Sample an action.

`compute_top_action_candidates`(*observation: Dict[str, numpy.ndarray]*, *num_candidates: int*, *maze_state: Optional[Any] = None*, *policy_id: Union[str, int] = None*, *deterministic: bool = False*) → Tuple[Sequence[Dict[str, Union[int, numpy.ndarray]]], Sequence[float]]

implementation of `Policy` interface

`needs_state`() → bool

This policy does not require the state() object to compute the action.

SerializedTorchPolicy

`class` `maze.core.agent.serialized_torch_policy.`**`SerializedTorchPolicy`**(*model: Union[omegaconf.DictConfig, Dict]*, *state_dict_file: str*, *spaces_dict_file: str*, *device: str*)

Structured policy used for rollouts of trained models.

Will build the models based on the model composer and spaces config and set the state of individual policies according to the state dict dump.

Policies are set to eval mode by default.

Parameters

- **`model`** – Model composer configuration
- **`state_dict_file`** – Path to dumped state dictionaries of the trained policies
- **`spaces_dict_file`** – Path to dumped spaces configuration (action and observation spaces of the env the policy was trained on, used for model initialization)

Critics:

`StateCritic`	Structured state critic class designed to work with structured environments.
`TorchStateCritic`	Encapsulates multiple torch state critics for training in structured environments.
`TorchSharedStateCritic`	One critic is shared across all sub-steps or actors (default to use for standard gym-style environments).
`TorchStepStateCritic`	Each sub-step or actor gets its individual critic.
`TorchDeltaStateCritic`	First sub step gets a regular critic, subsequent sub-steps predict a delta w.r.t.
`StateActionCritic`	Structured state action critic class designed to work with structured environments.
`TorchStateActionCritic`	Encapsulates multiple torch state action critics for training in structured environments.
`TorchSharedStateActionCritic`	One critic is shared across all sub-steps or actors (default to use for standard gym-style environments).
`TorchStepStateActionCritic`	Each sub-step or actor gets its individual critic.

StateCritic

class `maze.core.agent.state_critic.`**`StateCritic`**

Structured state critic class designed to work with structured environments. (see *`StructuredEnv`*).

It encapsulates state critic and queries them for values according to the provided policy ID.

abstract predict_value(*observation: Dict[str, numpy.ndarray], critic_id: Union[int, str]*) → torch.Tensor

Query a critic that corresponds to the given ID for the state value.

Parameters

- **`observation`** – Current observation of the environment
- **`critic_id`** – The critic id to query

Returns The value for this observation

abstract predict_values(*observations: Dict[Union[str, int], Dict[str, numpy.ndarray]]*) → Tuple[Dict[Union[str, int], torch.Tensor], Dict[Union[str, int], torch.Tensor]]

Query a critic that corresponds to the given ID for the state value.

Parameters **`observations`** – Current observation of the environment

Returns Tuple containing the values and detached values

TorchStateCritic

class maze.core.agent.torch_state_critic.**TorchStateCritic**(*networks: Mapping[Union[str, int], torch.nn.Module], num_policies: int, device: str*)

Encapsulates multiple torch state critics for training in structured environments.

Parameters

- **networks** – Mapping of value functions (critic) to encapsulate.
- **num_policies** – The number of corresponding policies.
- **device** – Device the policy should be located on (cpu or cuda)

bootstrap_returns(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]], rews: numpy.ndarray, dones: numpy.ndarray, gamma: float, gae_lambda: float*) → Tuple[Dict[Union[str, int], torch.Tensor], Dict[Union[str, int], torch.Tensor], Dict[Union[str, int], torch.Tensor]]

Bootstrap returns using the value function.

Useful for example to implement PPO or A2C.

Parameters

- **observations** – Sub-step observations as tensor dictionary.
- **rews** – Array holding the per step rewards.
- **dones** – Array indicating if a step is a done step.
- **gamma** – Discounting factor
- **gae_lambda** – Bias vs variance trade of factor for Generalized Advantage Estimator (GAE)

Returns Tuple containing the computed returns, the predicted values and the detached predicted values.

compute_return(*gamma: float, gae_lambda: float, rewards: numpy.ndarray, values: torch.Tensor, dones: numpy.ndarray, deltas: torch.Tensor = None*) → torch.Tensor

Compute bootstrapped return from rewards and estimated values.

Parameters

- **gamma** – Discounting factor
- **gae_lambda** – Bias vs variance trade of factor for Generalized Advantage Estimator (GAE)
- **rewards** – Step rewards with shape (n_steps, n_workers)
- **values** – Predicted values with shape (n_steps, n_workers)
- **dones** – Step dones with shape (n_steps, n_workers)
- **deltas** – Predicted value deltas to previous sub-step with shape (n_steps, n_workers)

Returns Per time step returns.

property device

implementation of *TorchModel*

eval() → None
implementation of *TorchModel*

load_state_dict(*state_dict: Dict*) → None
implementation of *TorchModel*

abstract property num_critics
Returns the number of critic networks. :return: Number of critic networks.

parameters() → List[torch.Tensor]
implementation of *TorchModel*

predict_value(*observation: Dict[str, numpy.ndarray], critic_id: Union[int, str]*) → torch.Tensor
implementation of *StateCritic*

state_dict() → Dict
implementation of *TorchModel*

to(*device: str*) → None
implementation of *TorchModel*

train() → None
implementation of *TorchModel*

TorchSharedStateCritic

class maze.core.agent.torch_state_critic.**TorchSharedStateCritic**(*networks: Mapping[Union[str, int], torch.nn.Module], num_policies: int, device: str*)

One critic is shared across all sub-steps or actors (default to use for standard gym-style environments). Can be instantiated via the *SharedStateCriticComposer*.

property num_critics
implementation of *TorchStateCritic*

predict_values(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]]*) → Tuple[Dict[Union[str, int], torch.Tensor], Dict[Union[str, int], torch.Tensor]]
implementation of *StateCritic*

TorchStepStateCritic

class maze.core.agent.torch_state_critic.**TorchStepStateCritic**(*networks: Mapping[Union[str, int], torch.nn.Module], num_policies: int, device: str*)

Each sub-step or actor gets its individual critic. Can be instantiated via the *StepStateCriticComposer*.

property num_critics
implementation of *TorchStateCritic*

predict_values(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]]*) → Tuple[Dict[Union[str, int], torch.Tensor], Dict[Union[str, int], torch.Tensor]]
implementation of `StateCritic`

TorchDeltaStateCritic

class maze.core.agent.torch_state_critic.**TorchDeltaStateCritic**(*networks: Mapping[Union[str, int], torch.nn.Module], num_policies: int, device: str*)

First sub step gets a regular critic, subsequent sub-steps predict a delta w.r.t. to the previous critic. Can be instantiated via the `DeltaStateCriticComposer`.

property num_critics
implementation of `TorchStateCritic`

predict_value(*observation: Dict[str, numpy.ndarray], critic_id: Union[int, str]*) → torch.Tensor
Predictions depend on previous sub-steps, thus this method is not supported in the delta state critic.

predict_values(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]]*) → Tuple[Dict[Union[str, int], torch.Tensor], Dict[Union[str, int], torch.Tensor]]
implementation of `StateCritic`

StateActionCritic

class maze.core.agent.state_action_critic.**StateActionCritic**

Structured state action critic class designed to work with structured environments. (see `StructuredEnv`).

It encapsulates state critic and queries them for values according to the provided policy ID.

abstract predict_q_values(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]], actions: Dict[Union[str, int], Dict[str, torch.Tensor]], gather_output: bool*) → Dict[Union[str, int], List[Union[torch.Tensor, Dict[str, torch.Tensor]]]]
Predict the Q value based on the observations and actions.

Parameters

- **observations** – The observation for the current step.
- **actions** – The action performed at the current step.
- **gather_output** – Specify whether to gather the output in the discrete setting.

Returns A list of tensors holding the predicted q value for each critic.

TorchStateActionCritic

class maze.core.agent.torch_state_action_critic.**TorchStateActionCritic**(*networks: Mapping[Union[str, int], torch.nn.Module], num_policies: int, device: str, only_discrete_spaces: Dict[Union[str, int], bool], action_spaces_dict: Dict[Union[str, int], gym.spaces.Dict]*)

Encapsulates multiple torch state action critics for training in structured environments.

Parameters

- **networks** – Mapping of value functions (critic) to encapsulate.
- **num_policies** – The number of corresponding policies.
- **device** – Device the policy should be located on (cpu or cuda)
- **only_discrete_spaces** – A dict specifying if the action spaces w.r.t. the step only hold discrete action spaces.

compute_state_action_value_step(*observation: Dict[str, torch.Tensor], action: Dict[str, torch.Tensor], critic_id: Union[str, int, tuple]*) → List[torch.Tensor]

Predict the value with specified step_key, step_observation and action.

Parameters

- **observation** – The observation for the current step.
- **action** – The action performed at the current step.
- **critic_id** – The current step key of the multi-step env.

Returns A list of tensors holding the predicted q value for each critic.

compute_state_action_values_step(*observation: Dict[str, torch.Tensor], critic_id: Union[str, int, tuple]*) → List[Dict[str, torch.Tensor]]

Predict the value with specified step_key, step_observation and action for discrete actions only.

Parameters

- **observation** – The observation for the current step.
- **critic_id** – The current step key of the multi-step env.

Returns A list of dicts holding the predicted q value for each action w.r.t. to the critic.

property device

implementation of `TorchModel`

eval() → None
implementation of *TorchModel*

load_state_dict(*state_dict: Dict*) → None
implementation of *TorchModel*

abstract property num_critics
Returns the number of critic networks. :return: Number of critic networks.

parameters() → List[torch.Tensor]
implementation of *TorchModel*

per_critic_parameters() → List[List[torch.Tensor]]
Retrieve all trainable critic parameters (to be assigned to optimizers). :return: List of lists holding all parameters for the base critic corresponding to number of critic per step.

abstract predict_next_q_values(*next_observations: Dict[Union[str, int], Dict[str, torch.Tensor]], next_actions: Dict[Union[str, int], Dict[str, torch.Tensor]], next_actions_logits: Dict[Union[str, int], Dict[str, torch.Tensor]], next_actions_log_probs: Dict[Union[str, int], Dict[str, torch.Tensor]], alpha: torch.Tensor*) → Dict[Union[str, int], Union[torch.Tensor, Dict[str, torch.Tensor]]]
Predict the target q value for the next step. $V(st) := E_{at}[Q(st, at)log((at|st))]$.

Parameters

- **next_observations** – The next observations.
- **next_actions** – The next actions sampled from the policy.
- **next_actions_logits** – The logits of the next actions (only relevantt for the discrete case).
- **next_actions_log_probs** – The log probabilities of the actions.
- **alpha** – The alpha, or entropy coefficient.

Returns A dict w.r.t. the step holding tensors representing the predicted next q value

abstract predict_q_values(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]], actions: Dict[Union[str, int], Dict[str, torch.Tensor]], gather_output: bool*) → Dict[Union[str, int], List[Union[torch.Tensor, Dict[str, torch.Tensor]]]]
implementation of *StateActionCritic*

re_init_networks() → None
Reinitialize all parameters of the network.

state_dict() → Dict
implementation of *TorchModel*

to(*device: str*) → None
implementation of *TorchModel*

train() → None
implementation of *TorchModel*

update_target_weights(*tau: float*) → None
Preform a soft or hard update depending on the tau value chosen. tau==1 results in a hard update

Parameters **tau** – Parameter weighting the soft update of the target network.

TorchSharedStateActionCritic

class maze.core.agent.torch_state_action_critic.**TorchSharedStateActionCritic**(*networks: Mapping[Union[str, int], torch.nn.Module], num_policies: int, device: str, only_discrete_space Dict[Union[str, int], bool], action_spaces_dict: Dict[Union[str, int], gym.spaces.Dict]*)

One critic is shared across all sub-steps or actors (default to use for standard gym-style environments). Can be instantiated via the *SharedStateActionCriticComposer*.

property num_critics

implementation of *TorchStateActionCritic*

predict_next_q_values(*next_observations: Dict[Union[str, int], Dict[str, torch.Tensor]], next_actions: Dict[Union[str, int], Dict[str, torch.Tensor]], next_actions_logits: Dict[Union[str, int], Dict[str, torch.Tensor]], next_actions_log_probs: Dict[Union[str, int], Dict[str, torch.Tensor]], alpha: torch.Tensor*) → Dict[Union[str, int], Union[torch.Tensor, Dict[str, torch.Tensor]]]

implementation of *TorchStateActionCritic*

predict_q_values(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]], actions: Dict[Union[str, int], Dict[str, torch.Tensor]], gather_output: bool*) → Dict[Union[str, int], List[Union[torch.Tensor, Dict[str, torch.Tensor]]]]

implementation of *TorchStateActionCritic*

TorchStepStateActionCritic

class maze.core.agent.torch_state_action_critic.**TorchStepStateActionCritic**(*networks: Mapping[Union[str, int], torch.nn.Module]*, *num_policies: int*, *device: str*, *only_discrete_spaces: Dict[Union[str, int], bool]*, *action_spaces_dict: Dict[Union[str, int], gym.spaces.Dict]*)

Each sub-step or actor gets its individual critic. Can be instantiated via the *StepStateActionCriticComposer*.

property num_critics

implementation of *TorchStateActionCritic*

predict_next_q_values(*next_observations: Dict[Union[str, int], Dict[str, torch.Tensor]]*, *next_actions: Dict[Union[str, int], Dict[str, torch.Tensor]]*, *next_actions_logits: Dict[Union[str, int], Dict[str, torch.Tensor]]*, *next_actions_log_probs: Dict[Union[str, int], Dict[str, torch.Tensor]]*, *alpha: torch.Tensor*) → Dict[Union[str, int], Union[torch.Tensor, Dict[str, torch.Tensor]]]

implementation of *TorchStateActionCritic*

predict_q_values(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]]*, *actions: Dict[Union[str, int], Dict[str, torch.Tensor]]*, *gather_output: bool*) → Dict[Union[str, int], List[Union[torch.Tensor, Dict[str, torch.Tensor]]]]

implementation of *TorchStateActionCritic*

Models:

TorchModel	Base class for any torch model.
TorchActorCritic	Encapsulates a structured torch policy and critic for training actor-critic algorithms in structured environments.

TorchModel

class maze.core.agent.torch_model.**TorchModel**(*device: str*)

Base class for any torch model.

Parameters **device** – Device the networks should be located on (cpu or cuda)

property device

Returns the device the networks are located on.

abstract eval() → None

Set all networks to eval mode.

abstract load_state_dict(*state_dict: Dict*) → None

Set state dict of all encapsulated networks. :param state_dict: The torch state dictionary.

property num_params

Returns overall number of network parameters.

abstract parameters() → List[torch.Tensor]

Returns all parameters of all networks.

abstract state_dict() → Dict

Return state dict composed of state dicts of all encapsulated networks.

abstract to(*device: str*) → None

Move all networks to the specified device. :param device: The target device.

abstract train() → None

Set all networks to training mode.

TorchActorCritic

class maze.core.agent.torch_actor_critic.**TorchActorCritic**(*policy: maze.core.agent.torch_policy.TorchPolicy, critic: Union[maze.core.agent.torch_state_critic.TorchS maze.core.agent.torch_state_action_critic.TorchS device: str*)

Encapsulates a structured torch policy and critic for training actor-critic algorithms in structured environments.

Parameters

- **policy** – A structured torch policy for training in structured environments.
- **critic** – A structured torch critic for training in structured environments.
- **device** – Device the model (networks) should be located on (cpu or cuda)

property device

implementation of *TorchModel*

eval() → None

implementation of *TorchModel*

load_state_dict(*state_dict: Dict*) → None

implementation of *TorchModel*

parameters() → List[torch.Tensor]

implementation of *TorchModel*

state_dict() → Dict
implementation of *TorchModel*

to(*device: str*)
implementation of *TorchModel*

train() → None
implementation of *TorchModel*

1.4.8 Agent Integration

This page contains the reference documentation for the Maze agent integration components.

AgentExecution	Executes the provided policies in an Agent Integration setting.
AgentIntegration	Encapsulates an agent, space interfaces and a stack of wrappers, to make the agent's MazeActions accessible to an external env.
ActionCandidates	Action object for encapsulation of multiple action objects along with their respective probabilities.
MazeActionCandidates	MazeAction object for encapsulation of multiple MazeAction objects along with their respective probabilities.
ActionConversionCandidatesInterface	Wrapper for action conversion interface when working with multiple candidate actions/MazeActions.
ExternalCoreEnvRewardAggregator	Reward aggregator for summing up rewards that come as iterables from external env.
ExternalCoreEnv	Acts as a CoreEnv in the env stack in agent integration scenario.

AgentExecution

class maze.core.agent_integration.agent_execution.**AgentExecution**(*env: maze.core.agent_integration.external_*
policy: maze.core.agent.policy.Policy, *roll-out_done_event: threading.Event*, *num_candidates: int*)

Executes the provided policies in an Agent Integration setting.

Policies are executed until the rollout_done event is set, indicating that the rollout has been finished. Then, a final reset is sent and execution stops. Expected to be run on a separate thread alongside the agent integration running on the main thread.

Parameters

- **env** – Environment to step.
- **policy** – Structured policy working with structured environments.
- **rollout_done_event** – event indicating that the rollout has been finished.

run_rollout_maze()
Step the environment until the rollout is done.

AgentIntegration

class maze.core.agent_integration.agent_integration.**AgentIntegration**(*policy: maze.core.agent.policy.Policy, action_conversions: Dict[Union[str, int], maze.core.env.action_conversion, observation_conversions: Dict[Union[str, int], maze.core.env.observation_conv num_candidates: int = 1, wrapper_types: Optional[List[Type[maze.core.wra = None, wrapper_kwargs: Optional[List[Dict[str, Any]]] = None, renderer: Optional[maze.core.rendering.rend = None*)

Encapsulates an agent, space interfaces and a stack of wrappers, to make the agent's MazeActions accessible to an external env.

External env should supply states to agent integration object, and can query it for agent MazeActions. The agent with the supplied policy (or multiple policies) is run on a separate thread.

Note that the two threads (main thread running this wrapper and the second thread running the agent, wrappers etc.) never run in parallel, i.e. one is always suspended. This is enforced using the queues. Either the main thread runs and the agent thread is waiting for the state to be passed from the main thread, or the agent thread is running (computing the MazeAction) and the main thread is waiting until the MazeAction is passed back (then, the second thread is suspended again until the next state is passed in via the queue).

Queues have max size of one, enforcing that one step can be taken at a time.

Parameters

- **policy** – Structured policy working with structured environments. When querying for

MazeAction, it can be specified what policy should be run (using the actor_id parameter, first part of which corresponds to the policy_id).

- **`action_conversions`** – Action conversion interfaces for the respective policies.
- **`observation_conversions`** – Observation interfaces for the respective policies.
- **`num_candidates`** – Number of MazeAction candidates to get from the policy. If greater than 1, will return multiple MazeActions wrapped in `MazeActionCandidates`
- **`wrapper_types`** – Which wrappers should be run as part of the agent's stack.
- **`wrapper_kwargs`** – Optional arguments to pass to the given wrappers on instantiation.

`finish_rollout`(*maze_state: Any, reward: Union[float, numpy.ndarray, Any], done: bool, info: Dict[Any, Any], events: Optional[List[*maze.core.events.event_record.EventRecord*]] = None*)

Should be called when the rollout is finished. While this has no effect on the provided MazeActions, it passes an env reset call through the wrapper stack, enabling the wrappers to do any work they normally do at the end of an episode (like write trajectory data).

Parameters

- **`maze_state`** – Final state of the rollout
- **`reward`** – Reward for the previous step (can be null in initial step)
- **`done`** – Whether the external environment is done
- **`info`** – Info dictionary
- **`events`** – List of events to be recorded for this step (mainly useful for statistics and event logs)

`get_maze_action`(*maze_state: Any, reward: Union[None, float, numpy.ndarray, Any], done: bool, info: Union[None, Dict[Any, Any]], events: Optional[List[*maze.core.events.event_record.EventRecord*]] = None, actor_id: Tuple[Union[str, int], int] = 0, 0*)

Query the agent for MazeAction derived from the given state.

Passes the state etc. to the agent's thread, where it is integrated into an ordinary env rollout loop. In the first step, an env reset call is propagated through the env wrapper stack on agent's thread.

Parameters

- **`maze_state`** – Current state of the environment.
- **`reward`** – Reward for the previous step (can be null in initial step)
- **`done`** – Whether the external environment is done
- **`info`** – Info dictionary
- **`events`** – List of events to be recorded for this step (mainly useful for statistics and event logs)
- **`actor_id`** – Optional ID of the actor to run next (comprised of policy_id and agent_id)

Returns MazeAction from the agent

ActionCandidates

class maze.core.agent_integration.maze_action_candidates.**ActionCandidates**(*candidates_and_probabi Tuple[Sequence[Any], Sequence[float]]*)

Action object for encapsulation of multiple action objects along with their respective probabilities. Useful when getting multiple candidate actions from a policy.

Parameters **candidates_and_probabilities** – a tuple of sequences, where the first sequence corresponds to the possible actions, the other sequence to the associated probabilities

MazeActionCandidates

class maze.core.agent_integration.maze_action_candidates.**MazeActionCandidates**(*candidates: Sequence[Any], probabilities: Sequence[float]*)

MazeAction object for encapsulation of multiple MazeAction objects along with their respective probabilities. Useful when working with multiple candidate MazeActions from a policy.

Parameters

- **candidates** – Candidate MazeActions
- **probabilities** – Respective probabilities

ActionConversionCandidatesInterface

class maze.core.agent_integration.maze_action_candidates.**ActionConversionCandidatesInterfa**

Wrapper for action conversion interface when working with multiple candidate actions/MazeActions.

Wraps an action_conversion interface. When action is passed in, uses the wrapped interface to convert all action candidates to respective MazeActions separately.

Parameters **action_conversion** – Underlying interface to apply to each candidate

space() → gym.spaces.space.Space

Return the space defined by the underlying action conversion interface.

space_to_maze(*action:* maze.core.agent_integration.maze_action_candidates.ActionCandidates, *maze_state: Any*) → *maze.core.agent_integration.maze_action_candidates.MazeActionCandidates*

Convert an action candidates object (containing multiple candidate actions) into corresponding MazeAction candidates object.

Parameters

- **action** – Action candidates object, encapsulating multiple actions.
- **maze_state** – Current state of the environment.

Returns MazeAction candidate object, encapsulating multiple MazeActions.

ExternalCoreEnvRewardAggregator

class maze.core.agent_integration.external_core_env.**ExternalCoreEnvRewardAggregator**

Reward aggregator for summing up rewards that come as iterables from external env. Scalar rewards are just passed through.

get_interfaces() → List[Type[abc.ABC]]

No event interfaces required.

classmethod to_scalar_reward(*reward: Any*) → float

Sum up reward if iterable, otherwise just pass through.

ExternalCoreEnv

class maze.core.agent_integration.external_core_env.**ExternalCoreEnv**(*state_queue: queue.Queue, maze_action_queue: queue.Queue, rollout_done_event: threading.Event, renderer: Optional[*maze.core.rendering.rende

Acts as a CoreEnv in the env stack in agent integration scenario.

Designed to be run on a separate thread, alongside the agent integration running on the main thread.

Hence, the control flow is: External env (like a Unity env) controlling the agent integration object, which in turn controls this external core env, which controls the execution of rollout loop by suspending it until the next state is available from the agent integration object.

Wrappers of this env and the agents acting on top of it see it as ordinary CoreEnv, but no actual logic happens here – instead, states and associated info are obtained from the agent integration running on the main thread, and executions produced by the agents are passed back to the agent integration.

During the step function, the execution of this thread is suspended while waiting for the next state from the agent integration.

Parameters

- **state_queue** – Queue this core env uses to get states from agent integration object
- **maze_action_queue** – Queue this core env uses to pass executions back to agent integration object
- **rollout_done_event** – Set by the agent integration object. Used for detection of the end of rollout period.
- **renderer** – If available, what renderer should be associated with the state data (for rendering, plus to be serialized with trajectory data)

actor_id() → Tuple[Union[str, int], int]

Current actor ID set by the agent integration.

close() → None
: No cleanup required.

get_kpi_calculator() → Optional[*maze.core.log_events.kpi_calculator.KpiCalculator*]
: No KPI calculator available.

get_maze_state() → Any
: Return the last state obtained from the external env through agent integration.

get_renderer() → Optional[*maze.core.rendering.renderer.Renderer*]
: Renderer provided by the agent integration. Might be None if not available.

get_serializable_components() → Dict[str, Any]
: No components required.

get_step_events() → Iterable[*maze.core.events.event_record.EventRecord*]
: Return events provided by the agent integration.

is_actor_done() → bool
: Whether last actor is done, as set by the agent integration.

reset() → Any
: Reset is expected to be run twice – at the beginning and end of external env rollout.

  At the beginning, thread execution is suspended until the initial state is available.

  At the end of the rollout, just the last state is returned, as there the reset serves the only purpose of notifying the wrappers to do their processing of the previous episode. (Also, no more states are available from the external env at this point.

seed(*seed: int*) → None
: No seed required – all operation handled by external env.

set_actor_id(*new_value: Tuple[Union[str, int], int]*)
: Hook for the agent integration to set actor_id before querying execution.

set_is_actor_done(*new_value: bool*)
: Hook for the agent integration to set the actor_done flag before querying execution.

step(*maze_action: Any*) → Tuple[Any, Union[float, numpy.ndarray, Any], bool, Dict[Any, Any]]
: Relays the execution back to the agent integration. Then suspends thread execution until the next state is provided by agent integration.

1.4.9 Perception Module

This page contains the reference documentation of *Maze Perception Module*.

Overview

- *maze.perception.blocks*
- *maze.perception.builders*
- *maze.perception.models*
- *maze.perception.perception_utils*
- *maze.perception.weight_init*

maze.perception.blocks

These are basic neural network building blocks and interfaces:

PerceptionBlock	Interface for all perception blocks.
ShapeNormalizationBlock	Perception block normalizing the input and de-normalizing the output tensor dimensions.
InferenceBlock	An inference block combining multiple perception blocks into one prediction module.
InferenceGraph	Models a perception module inference graph.

PerceptionBlock

class maze.perception.blocks.base.**PerceptionBlock**(**args: Any*, ***kwargs: Any*)

Interface for all perception blocks. Perception blocks provide a mapping of M input tensors to N output tensors. Both input and output tensors are stored in a dictionary with unique keys.

Parameters

- **in_keys** – Keys identifying the input tensors.
- **out_keys** – Keys identifying the output tensors.
- **in_shapes** – List of input shapes.

abstract forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]

Forward pass of perception block.

Parameters **block_input** – The block's input dictionary.

Returns The block's output dictionary.

get_num_of_parameters() → int

Calculates the total number of parameters in the model :return: The total number of parameters

out_shapes() → List[Sequence[int]]

Returns the perception block's output shape.

Returns a list of output shapes.

ShapeNormalizationBlock

class maze.perception.blocks.shape_normalization.**ShapeNormalizationBlock**(**args: Any*, ***kwargs: Any*)

Perception block normalizing the input and de-normalizing the output tensor dimensions.

Examples where this interface needs to be implemented are Dense Layers (batch-dim, feature-dim) or Convolution Blocks (batch-dim, feature-dim, row-dim, column-dim)

Parameters

- **in_keys** – Keys identifying the input tensors.
- **out_keys** – Keys identifying the output tensors.
- **in_shapes** – List of input shapes.

- **`in_num_dims`** – Required number of dimensions for corresponding input.
- **`out_num_dims`** – Required number of dimensions for corresponding output.

`forward`(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *`PerceptionBlock`* interface

`abstract normalized_forward`(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
Shape normalized forward pass called in the actual forward pass of this block.

Parameters **`block_input`** – The block's shape normalized input dictionary.

Returns The block's shape normalized output dictionary.

InferenceBlock

`class` `maze.perception.blocks.inference.`**`InferenceBlock`**(**args: Any*, ***kwargs: Any*)

An inference block combining multiple perception blocks into one prediction module.

Conditions on using the InferenceBlock object:

1. All keys of the perception_blocks dictionary have to be unique
2. All out_keys used when creating the blocks have to be unique
3. All block keys in the perception_blocks dict have to sub-strings of all their corresponding out_keys
4. The given in_keys should be a subset of the inputs of the computational graph
5. The given out_keys should be a subset of the outputs of the computational graph

Parameters

- **`in_keys`** – Keys identifying the input tensors.
- **`out_keys`** – Keys identifying the output tensors.
- **`in_shapes`** – List of input shapes.
- **`perception_blocks`** – Dictionary of perception blocks.

`forward`(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *`PerceptionBlock`* interface

InferenceGraph

`class` `maze.perception.blocks.inference.`**`InferenceGraph`**(*inference_block: maze.perception.blocks.inference.InferenceBlock*)

Models a perception module inference graph.

Conditions on using the InferenceGraph object:

1. All keys of the perception_blocks dictionary have to be unique
2. All out_keys used when creating the blocks have to be unique
3. All out_keys of a given block have to be sub-strings of the blocks key in the perception_blocks dict

Parameters **`inference_block`** – An inference perception block to build the graph for.

save(*name: str*, *save_path: str*) → str

Construct the network and save it as a pdf.

Parameters

- **name** – The name of the network to be drawn (used in the tile only).
- **save_path** – The path the figure should be saved.

Returns Returns the full absolute save path of the pdf.

show(*name: str*, *block_execution: bool*) → None

Construct the graph and show it.

Parameters

- **name** – The name of the network to be drawn (used in the tile only).
- **block_execution** – Specify whether the execution should be blocked.

Feed Forward: these are built-in feed forward building blocks:

`DenseBlock`	A block containing multiple subsequent dense layers.
`VGGConvolutionBlock`	A block containing multiple subsequent vgg style convolutions.
`StridedConvolutionBlock`	A block containing multiple subsequent strided convolution layers.
`GraphConvBlock`	A block containing multiple subsequent graph convolution stacks.
`GraphAttentionBlock`	A block containing multiple subsequent graph (multi-head) attention stacks.

DenseBlock

class maze.perception.blocks.feed_forward.dense.**DenseBlock**(**args: Any*, ***kwargs: Any*)

A block containing multiple subsequent dense layers. The block expects the input tensors to have the from (batch-dim, feature-dim).

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_units** – List containing the number of hidden units for hidden layers.
- **non_lin** – The non-linearity to apply after each layer.

build_layer_dict() → collections.OrderedDict

Compiles a block-specific dictionary of network layers. This could be overwritten by derived layers (e.g. to get a 'BatchNormalizedDenseBlock').

Returns Ordered dictionary of torch modules [str, nn.Module]

normalized_forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]

implementation of `ShapeNormalizationBlock` interface

VGGConvolutionBlock

class `maze.perception.blocks.feed_forward.vgg_conv.`**`VGGConvolutionBlock`**(**args: Any*, ***kwargs: Any*)

A block containing multiple subsequent vgg style convolutions.

One convolution stack consists of two subsequent 3x3 convolution layers followed by 2x2 max pooling. The block expects the input tensors to have the from (batch-dim, channel-dim, row-dim, column-dim).

Parameters

- **`in_keys`** – One key identifying the input tensors.
- **`out_keys`** – One key identifying the output tensors.
- **`in_shapes`** – List of input shapes.
- **`hidden_channels`** – List containing the number of hidden channels for hidden layers.
- **`non_lin`** – The non-linearity to apply after each layer.

`build_layer_dict`() → collections.OrderedDict

Compiles a block-specific dictionary of network layers. This could be overwritten by derived layers (e.g. to get a 'BatchNormalizedConvolutionBlock').

Returns Ordered dictionary of torch modules [str, nn.Module]

`normalized_forward`(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]

implementation of *`ShapeNormalizationBlock`* interface

StridedConvolutionBlock

class `maze.perception.blocks.feed_forward.strided_conv.`**`StridedConvolutionBlock`**(**args: Any*, ***kwargs: Any*)

A block containing multiple subsequent strided convolution layers.

One layer consists of a single strided convolution followed by an activation function. The block expects the input tensors to have the from (batch-dim, channel-dim, row-dim, column-dim).

Parameters

- **`in_keys`** – One key identifying the input tensors.
- **`out_keys`** – One key identifying the output tensors.
- **`in_shapes`** – List of input shapes.
- **`hidden_channels`** – List containing the number of hidden channels for hidden layers.
- **`hidden_kernels`** – List containing the size of the convolving kernels.
- **`non_lin`** – The non-linearity to apply after each layer.
- **`convolution_dimension`** – Dimension of the convolution to use [1, 2, 3]
- **`hidden_strides`** – List containing the strides of the convolutions.
- **`hidden_dilations`** – List containing the spacing between kernel elements.
- **`hidden_padding`** – List containing the padding added to both sides of the input

- **padding_mode** – 'zeros', 'reflect', 'replicate' or 'circular'.

build_layer_dict() → collections.OrderedDict
Compiles a block-specific dictionary of network layers. This could be overwritten by derived layers (e.g. to get a 'BatchNormalizedConvolutionBlock').

Returns Ordered dictionary of torch modules [str, nn.Module]

normalized_forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *ShapeNormalizationBlock* interface

GraphConvBlock

class maze.perception.blocks.feed_forward.graph_conv.**GraphConvBlock**(**args: Any, **kwargs: Any*)
A block containing multiple subsequent graph convolution stacks.

One convolution stack consists of one graph convolution in addition to an activation layer. The block expects the input tensors to have the form: - Feature matrix: first in_key: (batch-dim, num-of-nodes, feature-dim) - Adjacency matrix: second in_key: (batch-dim, num-of-nodes, num-of-nodes) (also symmetric) And returns a tensor of the form (batch-dim, num-of-nodes, feature-out-dim).

Parameters

- **in_keys** – Two keys identifying the feature matrix and adjacency matrix respectively.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_features** – List containing the number of hidden features for hidden layers.
- **bias** – Specify if a bias should be used at each layer (can be list or single).
- **non_lins** – The non-linearity/ies to apply after each layer (the same in all layers, or a list corresponding to each layer).
- **node_self_importance** – Specify how important a given node is to itself (default should be 1).
- **trainable_node_self_importance** – Specify if the node_self_importance should be a constant or a trainable parameter with init value :param node_self_importance.
- **preprocess_adj** – Specify whether to preprocess the adjacency, that is compute: adj^ := D^bar^(-1/2) A^bar D^bar^(-1/2) in every forward pass for the whole bach. If this is set to false, the already normalized adj^ is expected as an input. Here A^bar := A + I_n * :param self_importance_scalar, and D^bar_ii := sum_j A^bar_ij.

build_layer_dict() → collections.OrderedDict
Compiles a block-specific dictionary of network layers.

This could be overwritten by derived layers (e.g. to get a 'BatchNormalizedConvolutionBlock').

Returns Ordered dictionary of torch modules [str, nn.Module].

normalized_forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *ShapeNormalizationBlock* interface

GraphAttentionBlock

class maze.perception.blocks.feed_forward.graph_attention.**GraphAttentionBlock**(**args: Any*, ***kwargs: Any*)

A block containing multiple subsequent graph (multi-head) attention stacks.

One convolution stack consists of one graph multi-head attention in addition to an activation layer. The block expects the input tensors to have the form:

- Feature matrix: first in_key: (batch-dim, num-of-nodes, feature-dim)
- Adjacency matrix: second in_key: (batch-dim, num-of-nodes, num-of-nodes) (also symmetric)

And returns a tensor of the form (batch-dim, num-of-nodes, feature-out-dim).

Parameters

- **in_keys** – Two keys identifying the feature matrix and adjacency matrix respectively.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_features** – List containing the number of hidden features for hidden layers.
- **non_lins** – The non-linearity/ies to apply after each layer (the same in all layers, or a list corresponding to each layer).
- **n_heads** – The number of heads each stack should have. (default suggestion 8)
- **attention_alpha** – Specify the negative slope of the leakyReLU in each of the attention layers. parameter with init value :param node_self_importance. (default suggestion 0.2)
- **avg_last_head_attentions** – Specify whether to average the outputs from the attention head in the last layer of the attention stack. (default suggestion True or n_heads=0 in the last layer)
- **attention_dropout** – Specify the dropout to be within the layers applied on the computed attention.

build_layer_dict() → collections.OrderedDict

Compiles a block-specific dictionary of network layers.

This could be overwritten by derived layers (e.g. to get a 'BatchNormalizedConvolutionBlock').

Returns Ordered dictionary of torch modules [str, nn.Module].

normalized_forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]

implementation of *ShapeNormalizationBlock* interface

Recurrent: these are built-in recurrent building blocks:

LSTMBlock	A block containing multiple subsequent LSTM layers followed by a final time-distributed dense layer with explicit non-linearity.

LSTMBlock

class maze.perception.blocks.recurrent.lstm.**LSTMBlock**(**args: Any*, ***kwargs: Any*)

A block containing multiple subsequent LSTM layers followed by a final time-distributed dense layer with explicit non-linearity.

The block expects the input tensors to have the from (batch-dim, time-dim, feature-dim).

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_size** – The number of features in the hidden state.
- **num_layers** – Number of recurrent layers.
- **bidirectional** – If True, becomes a bidirectional LSTM.
- **non_lin** – The non-linearity to apply after the final layer.

normalized_forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]

implementation of *ShapeNormalizationBlock* interface

General: these are build-in general purpose building blocks:

FlattenBlock	A flattening block.
CorrelationBlock	A feature correlation block.
ConcatenationBlock	A feature concatenation block.
FunctionalBlock	A block applying a custom callable.
GlobalAveragePoolingBlock	A global average pooling block.
MaskedGlobalPoolingBlock	A block applying masked global pooling.
MultiIndexSlicingBlock	A multi-index-slicing block.
RepeatToMatchBlock	A repeat-to-match block.
SelfAttentionConvBlock	Implementation of a self-attention block as described by reference: https://arxiv.org/abs/1805.08318
SelfAttentionSeqBlock	Implementation of a self-attention block as described by reference: https://arxiv.org/abs/1706.03762
SliceBlock	A slicing block.
ActionMaskingBlock	An action masking block.

FlattenBlock

class maze.perception.blocks.general.flatten.**FlattenBlock**(**args: Any*, ***kwargs: Any*)

A flattening block.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **num_flatten_dims** – the number of dimensions to flatten out (from right).

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of `PerceptionBlock` interface

CorrelationBlock

class maze.perception.blocks.general.correlation.**CorrelationBlock**(**args: Any, **kwargs: Any*)

A feature correlation block.

This block takes two feature representation as an input and correlates them along the last dimension. If the blocks do not have the same number of dimensions additional 1d-dimensions are added to allow for broadcasting.

Parameters

- **in_keys** – Keys identifying the input tensors.
- **out_keys** – Keys identifying the output tensors.
- **in_shapes** – List of input shapes.
- **reduce** – If True a sum reduction as applied along dim=-1.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of `PerceptionBlock` interface

ConcatenationBlock

class maze.perception.blocks.general.concat.**ConcatenationBlock**(**args: Any, **kwargs: Any*)

A feature concatenation block.

Parameters

- **in_keys** – Keys identifying the input tensors.
- **out_keys** – Keys identifying the output tensors.
- **in_shapes** – List of input shapes.
- **concat_dim** – The index of the concatenation dimension.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of `PerceptionBlock` interface

FunctionalBlock

class maze.perception.blocks.general.functional.**FunctionalBlock**(**args: Any, **kwargs: Any*)

A block applying a custom callable. It processes a tensor or sequence of tensors and returns a tensor or sequence of tensors. If the callable has more than one argument the names of the arguments of the function declaration have to match the in_keys of the tensors.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.

- **in_shapes** – List of input shapes.
- **func** – A simple callable taking a tensor or a sequence of tensors and returning a tensor or a sequence of tensors.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
Forward pass through the block, applying the callable to the input.

Parameters **block_input** – The block's input dictionary.

Returns The block's output dictionary.

GlobalAveragePoolingBlock

class maze.perception.blocks.general.gap.**GlobalAveragePoolingBlock**(**args: Any*, ***kwargs: Any*)
A global average pooling block. The block expects the input tensors to have the from (batch-dim, channel-dim, row-dim, column-dim).

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.

normalized_forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *ShapeNormalizationBlock* interface

MaskedGlobalPoolingBlock

class maze.perception.blocks.general.masked_global_pooling.**MaskedGlobalPoolingBlock**(**args: Any*, ***kwargs Any*)
A block applying masked global pooling. Pooling is applied wrt the mask (in_keys[1]) and the selected pooling function. That is, in the forward pass the input tensor 1 is iterated over in the first 2 dimensions, where the elements are selected based on the mask, before applying the pooling function.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **pooling_func** – Options: {'mean'}. So far only mean pooling is supported.
- **pooling_dim** – The dimension(s) along which the pooling functions get applied.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
Forward pass through the block, iterating over the first 2 dimensions and pooling the rest in dim=0.

Parameters **block_input** – The block's input dictionary.

Returns The block's output dictionary.

MultiIndexSlicingBlock

class maze.perception.blocks.general.multi_index_slicing.**MultiIndexSlicingBlock**(**args: Any*, ***kwargs: Any*)

A multi-index-slicing block. This can be used rather than the short hand tensor[. . . ,[a,b]] where [a,b] is the list given as selection indices.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **select_dim** – The dimension to slice from.
- **select_idxs** – The index or indices to select.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]

Forward pass, slicing the input tensor as defined by the selection_dim and select_idxs.

Parameters **block_input** – The block's input dictionary.

Returns The block's output dictionary.

RepeatToMatchBlock

class maze.perception.blocks.general.repeat_to_match.**RepeatToMatchBlock**(**args: Any*, ***kwargs: Any*)

A repeat-to-match block. This blocks takes two tensors and a dimension index as an input. Then when it's forward method is called, it matches the specified the dimension (with :param repeat_at_idx) of the first tensor with the specified dimension (:param repeat_at_idx) of the second tensor. This is done by repeating the first tensor n times in dimension in dimension :param repeat_at_idx. Here n = tensor_1.shape[repeat_at_idx] - tensor_0.shape[repeat_at_idx]. As a constraint the first tensor has to satisfy the following condition: tensor_0[repeat_at_idx] == 1

Parameters

- **in_keys** – The keys identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **repeat_at_idx** – Specify the dimension that should be matched between the tensors.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]

Forward pass, repeat the first tensor to match the second one in the given dimension

Parameters **block_input** – The block's input dictionary.

Returns The block's output dictionary.

SelfAttentionConvBlock

class maze.perception.blocks.general.self_attention_conv.**SelfAttentionConvBlock**(**args: Any*, ***kwargs: Any*)

Implementation of a self-attention block as described by reference: https://arxiv.org/abs/1805.08318

This block can then be used for 2d data (images), to compute the self attention. If two out_keys are given, the actual attention is returned from the forward pass with the second out_key. Otherwise only the computed self-attention is returned

Parameters

- **in_keys** – Keys identifying the input tensors. First key is self_attention output, second optional key is attention mask.
- **out_keys** – Keys identifying the output tensors. First key is self-attention output, second optional key is attention map.
- **in_shapes** – List of input shapes.
- **embed_dim** – The embedding dimensionality, which should be an even fraction of the input channels.
- **add_input_to_output** – Specifies weather the computed self attention is added to the input and returned.
- **bias** – Specify weather to use a bias in the projections.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *PerceptionBlock* interface

SelfAttentionSeqBlock

class maze.perception.blocks.general.self_attention_seq.**SelfAttentionSeqBlock**(**args: Any*, ***kwargs: Any*)

Implementation of a self-attention block as described by reference: https://arxiv.org/abs/1706.03762

Within this block the torch nn.MuliheadAttention is used to model the self attention. This block can then be used for 1d data as well as sequential data, where the embedding dimensionality has to be equal to the last dimension of the input.

Parameters

- **in_keys** – Keys identifying the input tensors. First key is self_attention output, second optional key is attention mask.
- **out_keys** – Keys identifying the output tensors. First key is self-attention output, second optional key is attention map.
- **in_shapes** – List of input shapes.
- **embed_dim** – Embedding dimension of the model (has to be equal to the last dimension of the input).
- **num_heads** – Parallel attention heads.
- **dropout** – A dropout layer on attn_output_weights.

- **add_input_to_output** – Specifies weather the computed self attention is added to the input and returned.
- **bias** – Add bias as module parameter.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *PerceptionBlock* interface

SliceBlock

class maze.perception.blocks.general.slice.**SliceBlock**(**args: Any*, ***kwargs: Any*)
A slicing block. This is for example useful for slicing the last time step in recurrent blocks by its index.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **slice_dim** – The dimension to slice from.
- **slice_idx** – The index to slice.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *PerceptionBlock* interface

ActionMaskingBlock

class maze.perception.blocks.general.action_masking.**ActionMaskingBlock**(**args: Any*, ***kwargs: Any*)
An action masking block.

The block takes two keys as input where the first key contains the logits tensor and the second key contains the binary mask tensor. Masking is performed by adding the smallest possible float32 number to the logits where the corresponding mask value is False (0.0).

Parameters

- **in_keys** – Keys identifying the input tensors.
- **out_keys** – Keys identifying the output tensors.
- **in_shapes** – List of input shapes.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *PerceptionBlock* interface

Joint: these are build in joint building blocks combining multiple perception blocks:

FlattenDenseBlock	A block containing a flattening stage followed by a dense layer block.
VGGConvolutionDenseBlock	A block containing multiple subsequent vgg style convolution stacks followed by flattening and a dense layer block.

continues on next page

Table 29 – continued from previous page

VGGConvolutionGAPBlock	A block containing multiple subsequent vgg style convolution stacks followed by global average pooling.
StridedConvolutionDenseBlock	A block containing multiple subsequent strided convolutions followed by flattening and a dense layer block.
LSTMLastStepBlock	A block containing a LSTM perception block followed by a Slicing Block keeping only the output of the final time step.

FlattenDenseBlock

class maze.perception.blocks.joint_blocks.flatten_dense.**FlattenDenseBlock**(**args: Any, **kwargs: Any*)

A block containing a flattening stage followed by a dense layer block.

For details on flattening see *FlattenBlock*. For details on dense layers see *DenseBlock*.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **num_flatten_dims** – the number of dimensions to flatten out (from right).
- **hidden_units** – List containing the number of hidden units for hidden layers.
- **non_lin** – The non-linearity to apply after each layer.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *ShapeNormalizationBlock* interface

VGGConvolutionDenseBlock

class maze.perception.blocks.joint_blocks.vgg_conv_dense.**VGGConvolutionDenseBlock**(**args: Any, **kwargs: Any*)

A block containing multiple subsequent vgg style convolution stacks followed by flattening and a dense layer block.

For details on the convolution part see *VGGConvolutionBlock*. For details on flattening see *FlattenBlock*. For details on dense layers see *DenseBlock*.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_channels** – List containing the number of hidden channels for hidden layers.
- **hidden_units** – List containing the number of hidden units for hidden layers.

- **non_lin** – The non-linearity to apply after each layer.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *ShapeNormalizationBlock* interface

VGGConvolutionGAPBlock

class maze.perception.blocks.joint_blocks.vgg_conv_gap.**VGGConvolutionGAPBlock**(**args: Any, **kwargs: Any*)

A block containing multiple subsequent vgg style convolution stacks followed by global average pooling.

For details on the convolution part see *VGGConvolutionBlock*. For details on gap see *GlobalAveragePoolingBlock*.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_channels** – List containing the number of hidden channels for hidden layers.
- **non_lin** – The non-linearity to apply after each layer.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *ShapeNormalizationBlock* interface

StridedConvolutionDenseBlock

class maze.perception.blocks.joint_blocks.strided_conv_dense.**StridedConvolutionDenseBlock**(

A block containing multiple subsequent strided convolutions followed by flattening and a dense layer block.

For details on the convolution part see *StridedConvolutionBlock*. For details on flattening see *FlattenBlock*. For details on dense layers see *DenseBlock*.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_channels** – List containing the number of hidden channels for hidden layers.
- **hidden_kernels** – List containing the size of the convolving kernels.
- **convolution_dimension** – Dimension of the convolution to use [1, 2, 3]
- **hidden_strides** – List containing the strides of the convolutions.
- **hidden_dilations** – List containing the spacing between kernel elements.
- **hidden_padding** – List containing the padding added to both sides of the input
- **padding_mode** – 'zeros', 'reflect', 'replicate' or 'circular'.

- **hidden_units** – List containing the number of hidden units for hidden layers.
- **non_lin** – The non-linearity to apply after each layer.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of `ShapeNormalizationBlock` interface

LSTMLastStepBlock

class maze.perception.blocks.joint_blocks.lstm_last_step.**LSTMLastStepBlock**(**args: Any, **kwargs: Any*)

A block containing a LSTM perception block followed by a Slicing Block keeping only the output of the final time step.

For details on flattening see `LSTMBlock`. For details on dense layers see `SliceBlock`.

Parameters

- **in_keys** – One key identifying the input tensors.
- **out_keys** – One key identifying the output tensors.
- **in_shapes** – List of input shapes.
- **hidden_size** – The number of features in the hidden state.
- **num_layers** – Number of recurrent layers.
- **bidirectional** – If True, becomes a bidirectional LSTM.
- **non_lin** – The non-linearity to apply after the final layer.

forward(*block_input: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of `PerceptionBlock` interface

maze.perception.builders

These are template model builders:

`BaseModelBuilder`	Base class for perception default model builders.
`ConcatModelBuilder`	A model builder that first processes individual observations, concatenates the resulting latent spaces and then processes this concatenated output to action and value outputs.

BaseModelBuilder

class maze.perception.builders.base.**BaseModelBuilder**(*modality_config: Dict[str, Union[str, Dict[str, Any]]], observation_modality_mapping: Dict[str, str]*)

Base class for perception default model builders.

Param modality_config: dictionary mapping perception modalities to blocks and block config parameters.

Parameters **`observation_modality_mapping`** – A mapping of observation keys to perception modalities.

`abstract from_observation_space`(*observation_space: gym.spaces.Dict*) → *maze.perception.blocks.inference.InferenceBlock*

Compiles an inference graph for a given observation space.

Only observations which are contained in the self.observation_modalities dictionary are considered.

Parameters **`observation_space`** – The respective observation space.

Returns the resulting inference block.

`classmethod to_recurrent_gym_space`(*observation_space: gym.spaces.Dict*, *rnn_steps: int*) → gym.spaces.Dict

Converts the given observation space to a recurrent space.

Parameters

- **`observation_space`** – The respective observation space.
- **`rnn_steps`** – Number of recurrent time steps.

Returns The rnn modified dictionary observation space.

ConcatModelBuilder

`class` `maze.perception.builders.concat.`**`ConcatModelBuilder`**(*modality_config: Dict[str, Union[str, Dict[str, Any]]]*, *observation_modality_mapping: Dict[str, str]*)

A model builder that first processes individual observations, concatenates the resulting latent spaces and then processes this concatenated output to action and value outputs.

Each input observation is first processed with the specified perception block. The required feature dimensionality after this step is 1D! In a next step the latent representations of the previous step are concatenated along the last dimension and once more processed with a `DenseBlock`.

Param modality_config: dictionary mapping perception modalities to blocks and block config parameters.

Parameters **`observation_modality_mapping`** – A mapping of observation keys to perception modalities.

`from_observation_space`(*observation_space: gym.spaces.Dict*) → *maze.perception.blocks.inference.InferenceBlock*

implementation of `BaseModelBuilder` interface

maze.perception.models

These are model composers and components:

BaseModelComposer	Abstract baseclass and interface definitions for model composers.
TemplateModelComposer	Composes template models from configs.
CustomModelComposer	Composes models from explicit model definitions.
SpacesConfig	Represents configuration of environment spaces (action & observation) used for model config.

BaseModelComposer

class maze.perception.models.model_composer.**BaseModelComposer**(*action_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], observation_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], distribution_mapper_config: Union[None, str, Mapping[str, Any], Any]*)

Abstract baseclass and interface definitions for model composers.

Model composers encapsulate the set of policy and critic networks along with the distribution mapper.

Parameters

- **action_spaces_dict** – Dict of sub-step id to action space.
- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **distribution_mapper_config** – Distribution mapper configuration.

abstract classmethod check_model_config(*model_config: Union[None, str, Mapping[str, Any], Any]*) → None

Asserts the provided model config for consistency. :param model_config: The model config to check.

abstract property critic

The critic model.

property distribution_mapper

The DistributionMapper, mapping the action heads to distributions.

abstract property policy

Policy networks.

save_models() → None

Save the policies and critics as pdfs.

TemplateModelComposer

class maze.perception.models.template_model_composer.**TemplateModelComposer**(*action_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], observation_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], distribution_mapper_config: Union[None, str, Mapping[str, Any], Any], model_builder: Union[None, str, Mapping[str, Any], Any, Type[*maze.perception.l *policy: Union[None, str, Mapping[str, Any], Any], critic: Union[None, str, Mapping[str, Any], Any]*)

Composes template models from configs.

Parameters

- **action_spaces_dict** – Dict of sub-step id to action space.
- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **distribution_mapper_config** – Distribution mapper configuration.

- **model_builder** – The model builder (template) to create the model from.
- **policy** – Specifies the policy type as a configType. E.g. {'type': maze.perception.models.policies.ProbabilisticPolicyComposer} specifies a probabilistic policy.
- **critic** – Specifies the critic type as a configType. E.g. {'type': maze.perception.models.critics.StateCriticComposer} specifies the single step state critic.

classmethod check_model_config(*model_config: Union[None, str, Mapping[str, Any], Any]*) → None

Asserts the provided model config for consistency. :param model_config: The model config to check.

property critic

Implementation of the BaseModelComposer interface, returns the value networks.

property policy

Implementation of the BaseModelComposer interface, returns the policy networks.

template_perception_net(*observation_space: gym.spaces.Dict*) → *maze.perception.blocks.inference.InferenceBlock*

Compiles a template perception network for a given observation space.

Parameters **observation_space** – The observation space tp build the model for.

Returns A Perception Inference Block.

template_policy_net(*observation_space: gym.spaces.Dict, action_space: gym.spaces.Dict*) → *maze.perception.blocks.inference.InferenceBlock*

Compiles a template policy network.

Parameters

- **observation_space** – The input observations for the perception network.
- **action_space** – The action space that defines the network action heads.

Returns A policy network (actor) InferenceBlock.

template_q_value_net(*observation_space: Optional[gym.spaces.Dict], action_space: gym.spaces.Dict, only_discrete_spaces: bool, perception_net: Optional[maze.perception.blocks.inference.InferenceBlock] = None*) → *maze.perception.blocks.inference.InferenceBlock*

Compiles a template state action (Q) value network.

Parameters

- **observation_space** – The input observations for the perception network.
- **action_space** – The action space that defines the network action heads.
- **perception_net** – A initial network to continue from. (e.g. useful for shared weights. Model building continues from the key 'latent'.)
- **only_discrete_spaces** – A dict specifying if the action spaces w.r.t. the step only hold discrete action spaces.

Returns A q value network (critic) InferenceBlock.

template_value_net(*observation_space: Optional[gym.spaces.Dict], perception_net: Optional[maze.perception.blocks.inference.InferenceBlock] = None*) → *maze.perception.blocks.inference.InferenceBlock*

Compiles a template value network.

Parameters

- **`observation_space`** – The input observations for the perception network.
- **`perception_net`** – A initial network to continue from. (e.g. useful for shared weights. Model building continues from the key 'latent'.)

Returns A value network (critic) InferenceBlock.

CustomModelComposer

class `maze.perception.models.custom_model_composer.`**`CustomModelComposer`**(*action_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], observation_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], distribution_mapper_config: Union[None, str, Mapping[str, Any], Any], policy: Union[None, str, Mapping[str, Any], Any], critic: Union[None, str, Mapping[str, Any], Any]*)

Composes models from explicit model definitions.

Parameters

- **`action_spaces_dict`** – Dict of sub-step id to action space.
- **`observation_spaces_dict`** – Dict of sub-step id to observation space.
- **`distribution_mapper_config`** – Distribution mapper configuration.
- **`policy`** – Mapping of sub-step keys to models.

- **`critic`** – Configuration for the critic composer.

`classmethod check_model_config`(*model_config: Union[None, str, Mapping[str, Any], Any]*) → None

Asserts the provided model config for consistency. :param model_config: The model config to check.

`property critic`

Return the critic networks.

`property policy`

Return the policy networks.

SpacesConfig

`class` `maze.perception.models.space_config.`**`SpacesConfig`**(*action_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], observation_spaces_dict: Dict[Union[str, int], gym.spaces.Dict]*)

Represents configuration of environment spaces (action & observation) used for model config.

Spaces config are needed (together with model config and dumped state dict) when loading a trained policy for rollout.

`classmethod load`(*in_file_path: str*) → *maze.perception.models.space_config.SpacesConfig*

Load a saved spaces config from a file.

Parameters **`in_file_path`** – Where to load the spaces config from.

Returns Loaded spaces config object

`save`(*dump_file_path: str*) → None

Save the spaces config to a file.

Parameters **`dump_file_path`** – Where to save the spaces config.

These are **maze.perception.models.policies**

`BasePolicyComposer`	Interface for policy (actor) network composers.
`ProbabilisticPolicyComposer`	Composes networks for probabilistic policies.

BasePolicyComposer

class maze.perception.models.policies.base_policy_composer.**BasePolicyComposer**(*action_spaces_dic Dict[Union[str, int], gym.spaces.Dict], observation_spaces_dict: Dict[Union[str, int], gym.spaces.Dict], distribution_mapper:* maze.distributions

Interface for policy (actor) network composers.

Parameters

- **action_spaces_dict** – Dict of sub-step id to action space.
- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **distribution_mapper** – The distribution mapper.

abstract property policy
The policy object

ProbabilisticPolicyComposer

class maze.perception.models.policies.probabilistic_policy_composer.**ProbabilisticPolicyComp**

Composes networks for probabilistic policies.

Parameters

- **action_spaces_dict** – Dict of sub-step id to action space.
- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **distribution_mapper** – The distribution mapper.
- **networks** – Policy networks as defined in the config (either list or dictionary of object params and type).

property policy
implementation of *BasePolicyComposer*

There are **maze.perception.models.critics**

BaseStateCriticComposer	Interface for critic (value function) network composers.
SharedStateCriticComposer	One critic is shared across all sub-steps or actors (default to use for standard gym-style environments).
StepStateCriticComposer	Each sub-step or actor gets its individual critic.

continues on next page

Table 33 – continued from previous page

DeltaStateCriticComposer	First sub step gets a regular critic, subsequent sub-steps predict a delta w.r.t.
StateCriticComposer	alias of *maze.perception.models.critics.step_state_critic_composer.StepStateCriticComposer*
BaseStateActionCriticComposer	Interface for state action (Q) critic network composers.
SharedStateActionCriticComposer	One critic is shared across all sub-steps or actors (default to use for standard gym-style environments).
StepStateActionCriticComposer	Each sub-step or actor gets its individual critic.
StateActionCriticComposer	alias of *maze.perception.models.critics.step_state_action_critic_composer.StepStateActionCriticComposer*

BaseStateCriticComposer

class maze.perception.models.critics.base_state_critic_composer.**BaseStateCriticComposer**(*obs*
Dic
int]
gym

Interface for critic (value function) network composers.

Parameters **observation_spaces_dict** – Dict of sub-step id to observation space.

abstract property critic
value networks

SharedStateCriticComposer

class maze.perception.models.critics.shared_state_critic_composer.**SharedStateCriticCompose**

One critic is shared across all sub-steps or actors (default to use for standard gym-style environments).

Instantiates a *TorchSharedStateCritic*.

Parameters

- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **networks** – The single, shared critic network as defined in the config.

property critic
implementation of *BaseStateCriticComposer*

StepStateCriticComposer

class maze.perception.models.critics.step_state_critic_composer.**StepStateCriticComposer**(*obs*
Dic
int]
gym
net-
wor
Uni
str,
Ma
ping
Any
Any
Ma
ping
Uni
str,
Ma
ping
Any
Any

Each sub-step or actor gets its individual critic.

Instantiates a *TorchStepStateCritic*.

Parameters

- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **networks** – Critics networks as defined in the config (either list or dictionary of object params and type).

property critic
implementation of *BaseStateCriticComposer*

DeltaStateCriticComposer

class maze.perception.models.critics.delta_state_critic_composer.**DeltaStateCriticComposer**(

First sub step gets a regular critic, subsequent sub-steps predict a delta w.r.t. to the previous critic.

Instantiates a *TorchDeltaStateCritic*.

Parameters

- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **networks** – The single, shared critic network as defined in the config.

property critic
implementation of *BaseStateCriticComposer*

StateCriticComposer

maze.perception.models.critics.**StateCriticComposer**
alias of *maze.perception.models.critics.step_state_critic_composer.StepStateCriticComposer*

BaseStateActionCriticComposer

class maze.perception.models.critics.base_state_action_critic_composer.**BaseStateActionCrit**

Interface for state action (Q) critic network composers.

Parameters

- **`observation_spaces_dict`** – Dict of sub-step id to observation space.
- **`action_spaces_dict`** – Dict of sub-step id to action space.

`abstract property critic`
value networks

SharedStateActionCriticComposer

`class` `maze.perception.models.critics.shared_state_action_critics_composer.`**`SharedStateActio`**

One critic is shared across all sub-steps or actors (default to use for standard gym-style environments).

Instantiates a *`TorchSharedStateActionCritic`*.

Parameters

- **`observation_spaces_dict`** – Dict of sub-step id to observation space.
- **`action_spaces_dict`** – Dict of sub-step id to action space.
- **`networks`** – Critics networks as defined in the config (either list or dictionary of object params and type).

`property critic`
implementation of *`BaseStateActionCriticComposer`*

StepStateActionCriticComposer

class maze.perception.models.critics.step_state_action_critic_composer.**StepStateActionCrit**

Each sub-step or actor gets its individual critic.

Instantiates a *TorchStepStateActionCritic*.

Parameters

- **observation_spaces_dict** – Dict of sub-step id to observation space.
- **action_spaces_dict** – Dict of sub-step id to action space.
- **networks** – Critics networks as defined in the config (either list or dictionary of object params and type).

property critic
implementation of *BaseStateActionCriticComposer*

StateActionCriticComposer

maze.perception.models.critics.**StateActionCriticComposer**
alias of *maze.perception.models.critics.step_state_action_critic_composer.StepStateActionCriticComposer*

These are **maze.perception.models.build_in** models

FlattenConcatBaseNet	Base flatten and concatenation model for policies and critics.
FlattenConcatPolicyNet	Flatten and concatenation policy model.
FlattenConcatStateValueNet	Flatten and concatenation state value model.

FlattenConcatBaseNet

class maze.perception.models.built_in.flatten_concat.**FlattenConcatBaseNet**(**args: Any*, ***kwargs: Any*)

Base flatten and concatenation model for policies and critics.

Parameters

- **obs_shapes** – Dictionary mapping of observation names to shapes.
- **hidden_units** – Dictionary mapping of action names to shapes.
- **non_lin** – The non-linearity to apply.

FlattenConcatPolicyNet

class maze.perception.models.built_in.flatten_concat.**FlattenConcatPolicyNet**(**args: Any*, ***kwargs: Any*)

Flatten and concatenation policy model.

Parameters

- **obs_shapes** – Dictionary mapping of observation names to shapes.
- **action_logits_shapes** – Dictionary mapping of observation names to shapes.
- **hidden_units** – Dictionary mapping of action names to shapes.
- **non_lin** – The non-linearity to apply.

forward(*x*)
forward pass.

FlattenConcatStateValueNet

class maze.perception.models.built_in.flatten_concat.**FlattenConcatStateValueNet**(**args: Any*, ***kwargs: Any*)

Flatten and concatenation state value model.

Parameters

- **obs_shapes** – Dictionary mapping of observation names to shapes.
- **hidden_units** – Dictionary mapping of action names to shapes.
- **non_lin** – The non-linearity to apply.

forward(*x*)
forward pass.

maze.perception.perception_utils

These are some helper functions when working with the perception module:

observation_spaces_to_in_shapes	Convert an observation space to the input shapes for the neural networks
flat_structured_observations	Compiles a flat dict from a structured observation nested dictionary.
convert_to_torch	Converts any struct to torch.Tensors.
convert_to_numpy	Convert torch to np

observation_spaces_to_in_shapes

class maze.perception.perception_utils.**observation_spaces_to_in_shapes**(*observation_spaces: Dict[Union[int, str], gym.spaces.Dict]*)

Convert an observation space to the input shapes for the neural networks

Parameters **observation_spaces** – the observation spaces of a structured Env

Returns the same structure but all the gym spaces are converted to tuples

flat_structured_observations

class maze.perception.perception_utils.**flat_structured_observations**(*structured_obs: Dict[Union[str, int], Dict[str, torch.Tensor]]*)

Compiles a flat dict from a structured observation nested dictionary.

Param The structured dictionary of observations.

Returns The flattened observation dictionary.

convert_to_torch

class maze.perception.perception_utils.**convert_to_torch**(*stats: Any, device: Optional[str], cast: Optional[torch.dtype], in_place: Union[bool, str]*)

Converts any struct to torch.Tensors.

Parameters

- **stats** – Any (possibly nested) struct, the values in which will be converted and returned as a new struct with all leaves converted to torch tensors.
- **device** – 'cpu' or 'cuda', or None if it should stay the same
- **cast** – the type the element should be cast to, or None if it should stay the same
- **in_place** – specify if the operation should be done in_place, can be bool or 'try'

Returns A new struct with the same structure as *stats*, but with all values converted to torch Tensor types.

convert_to_numpy

class maze.perception.perception_utils.**convert_to_numpy**(*stats: Any*, *cast: Optional[numpy.dtype]*, *in_place: Union[bool, str]*)

Convert torch to np

Parameters

- **stats** – Any (possibly nested) struct, the values in which will be converted and returned as a new struct with all leaves converted to torch tensors.
- **cast** – if the element should also be casted to a specific type
- **in_place** – specify if the operation should be done in_palce, can be bool or 'try'

Returns A new struct with the same structure as *stats*, but with all values converted to torch Tensor types. can be bool or 'try'

maze.perception.weight_init

These are some helper functions for initializing model weights:

make_module_init_normc	Compiles normc weight initialization function initializing module weights with normc_initializer and biases with zeros.
compute_sigmoid_bias	Compute the bias value for a sigmoid activation function such as in multi-binary action spaces (Bernoulli distributions).

make_module_init_normc

class maze.perception.weight_init.**make_module_init_normc**(*std: float = 1.0*)

Compiles normc weight initialization function initializing module weights with normc_initializer and biases with zeros.

Parameters **std** – The standard deviation.

Returns The module initialization function.

compute_sigmoid_bias

class maze.perception.weight_init.**compute_sigmoid_bias**(*probability: float*)

Compute the bias value for a sigmoid activation function such as in multi-binary action spaces (Bernoulli distributions).

Parameters **probability** – The desired selection probability.

Returns The respective bias value.

1.4.10 Action Spaces and Distributions Module

This page contains the reference documentation of *Maze Action Spaces and Distributions Module*.

These are interfaces, classes and utility functions:

ProbabilityDistribution	Base class for all probability distributions.
TorchProbabilityDistribution	Base class for wrapping Torch probability distributions.
DistributionMapper	Provides a mapping of spaces and action heads to the respective probability distributions to be used.
atanh	Computes the arc-tangent hyperbolic.
tensor_clamp	Clamping with tensor and broadcast support.

ProbabilityDistribution

class maze.distributions.distribution.**ProbabilityDistribution**

Base class for all probability distributions.

deterministic_sample() → Any

Draw a deterministic sample from the probability distribution.

Returns deterministic sample tensor.

entropy() → Any

Calculate the entropy of the probability distribution.

Returns entropy tensor.

kl(*other:* maze.distributions.distribution.ProbabilityDistribution) → Any

Calculates the Kullback-Leibler between self and the other probability distribution.

Parameters **other** – ([float]) the distribution to compare with.

Returns kl tensor.

log_prob(*actions: Any*) → Any

Returns the the log likelihood of the provided actions.

actions: the actions. :return: log likelihood tensor.

neg_log_prob(*actions: Any*) → Any

Returns the the negative log likelihood of the provided actions.

Parameters **actions** – the actions.

Returns negative log likelihood tensor.

sample() → Any

Draw a sample from the probability distribution.

Returns stochastic sample tensor.

TorchProbabilityDistribution

class `maze.distributions.torch_dist.`**`TorchProbabilityDistribution`**(**args, **kwds*)

Base class for wrapping Torch probability distributions.

Parameters

- **`dist`** – The torch probability distribution.
- **`action_space`** – The gym action space.

`entropy`() → torch.Tensor
implementation of *`ProbabilityDistribution`* interface

`kl`(*other: maze.distributions.torch_dist.TorchProbabilityDistribution*) → torch.Tensor
implementation of *`ProbabilityDistribution`* interface

`log_prob`(*actions: torch.Tensor*) → torch.Tensor
implementation of *`ProbabilityDistribution`* interface

abstract classmethod `required_logits_shape`(*action_space: gym.spaces.Space*) → Sequence[int]
Returns the required shape for the corresponding neural network logits output.

Parameters **`action_space`** – The respective action space to compute logits for.

Returns The required logits shape.

`sample`() → torch.Tensor
implementation of *`ProbabilityDistribution`* interface

DistributionMapper

class `maze.distributions.distribution_mapper.`**`DistributionMapper`**(*action_space: gym.spaces.Dict, distribution_mapper_config: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]]*)

Provides a mapping of spaces and action heads to the respective probability distributions to be used.

This ensures full flexibility for specifying different distributions to the same gym action space type. (e.g. One gym.spaces.Box space could be modeled with a Beta another one with a DiagonalGaussian distribution.) It allows to add and register arbitrary custom distributions.

Parameters

- **`action_space`** – The dictionary action space.

- **distribution_mapper_config** – A Distribution mapper configuration (for details see the docs).

action_head_distribution(*action_head: str, logits: torch.Tensor, temperature: float*) → *maze.distributions.torch_dist.TorchProbabilityDistribution*

Creates a probability distribution for a given action head.

Parameters

- **action_head** – The name of the action head (action dictionary key).
- **logits** – the logits to parameterize the distribution from
- **temperature** – Controls the sampling behaviour * 1.0 corresponds to unmodified sampling * smaller than 1.0 concentrates the action distribution towards deterministic sampling

Returns (ProbabilityDistribution) the appropriate instance of a ProbabilityDistribution

logits_dict_to_distribution(*logits_dict: Dict[str, torch.Tensor], temperature: float*) → *maze.distributions.dict.DictProbabilityDistribution*

Creates a dictionary probability distribution for a given logits dictionary.

Parameters

- **logits_dict** – A logits dictionary [action_head: action_logits] to parameterize the distribution from.
- **temperature** – Controls the sampling behaviour. * 1.0 corresponds to unmodified sampling * smaller than 1.0 concentrates the action distribution towards deterministic sampling

Returns (DictProbabilityDistribution) the respective instance of a DictProbabilityDistribution.

required_logits_shape(*action_head: str*) → Sequence[int]

Returns the required logits shape (network output shape) for a given action head.

Parameters **action_head** – The name of the action head (action dictionary key).

Returns The required logits shape.

atanh

class maze.distributions.utils.**atanh**(*x: torch.Tensor*)

Computes the arc-tangent hyperbolic.

Parameters **x** – The input tensor.

Returns The arc-tangent hyperbolic of x.

tensor_clamp

class maze.distributions.utils.**tensor_clamp**(*x: torch.Tensor, t_min: torch.Tensor, t_max: torch.Tensor*)

Clamping with tensor and broadcast support.

Parameters

- **x** – the tensor to clamp.
- **t_min** – the minimum values.
- **t_max** – the maximum values.

Returns the clamped tensor.

These are built-in Torch probability distributions:

CategoricalProbabilityDistribution	Categorical Torch probability distribution.
BernoulliProbabilityDistribution	Bernoulli Torch probability distribution for multi-binary action spaces.
DiagonalGaussianProbabilityDistribution	Diagonal Gaussian (Normal) Torch probability distribution.
SquashedGaussianProbabilityDistribution	Tanh-squashed diagonal Gaussian (Normal) Torch probability distribution.
BetaProbabilityDistribution	Beta Torch probability distribution.

CategoricalProbabilityDistribution

class maze.distributions.categorical.**CategoricalProbabilityDistribution**(**args*, ***kwds*)

Categorical Torch probability distribution.

Parameters **logits** – the action selection logits.

deterministic_sample()
implementation of *ProbabilityDistribution* interface

log_prob(*actions: torch.Tensor*) → torch.Tensor
implementation of *ProbabilityDistribution* interface

classmethod required_logits_shape(*action_space: gym.spaces.Discrete*) → Sequence[int]
implementation of *TorchProbabilityDistribution* interface

BernoulliProbabilityDistribution

class maze.distributions.bernoulli.**BernoulliProbabilityDistribution**(**args*, ***kwds*)

Bernoulli Torch probability distribution for multi-binary action spaces.

Parameters

- **logits** – the action selection logits.
- **action_space** – The gym action space.
- **temperature** – The distribution temperature parameter.

deterministic_sample() → torch.Tensor
implementation of *ProbabilityDistribution* interface

entropy() → torch.Tensor
implementation of *ProbabilityDistribution* interface

kl(*other:* maze.distributions.torch_dist.TorchProbabilityDistribution) → torch.Tensor
implementation of *ProbabilityDistribution* interface

log_prob(*actions: torch.Tensor*) → torch.Tensor
implementation of *ProbabilityDistribution* interface

classmethod required_logits_shape(*action_space: gym.spaces.MultiBinary*) → Sequence[int]
implementation of *TorchProbabilityDistribution* interface

DiagonalGaussianProbabilityDistribution

class maze.distributions.gaussian.**DiagonalGaussianProbabilityDistribution**(**args, **kwds*)

Diagonal Gaussian (Normal) Torch probability distribution.

Parameters **logits** – The logits for both mean and standard deviation.

deterministic_sample() → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

entropy() → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

kl(*other: maze.distributions.torch_dist.TorchProbabilityDistribution*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

log_prob(*actions: torch.Tensor*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

classmethod required_logits_shape(*action_space: gym.spaces.Space*) → Sequence[int]
implementation of *TorchProbabilityDistribution* interface

SquashedGaussianProbabilityDistribution

class maze.distributions.squashed_gaussian.**SquashedGaussianProbabilityDistribution**(**args, **kwds*)

Tanh-squashed diagonal Gaussian (Normal) Torch probability distribution.

Parameters

- **logits** – the logits for both mean and standard deviation.
- **action_space** – the underlying gym.spaces action space.

deterministic_sample()
implementation of *TorchProbabilityDistribution* interface

entropy() → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

kl(*other: maze.distributions.torch_dist.TorchProbabilityDistribution*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

log_prob(*actions: torch.Tensor*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

classmethod required_logits_shape(*action_space: gym.spaces.Space*) → Sequence[int]
implementation of *TorchProbabilityDistribution* interface

sample()
implementation of *TorchProbabilityDistribution* interface

BetaProbabilityDistribution

class maze.distributions.beta.**BetaProbabilityDistribution**(**args*, ***kwds*)

Beta Torch probability distribution.

Parameters

- **logits** – the logits for both mean and standard deviation.
- **action_space** – the underlying gym.spaces action space.

deterministic_sample() → torch.Tensor

implementation of *TorchProbabilityDistribution* interface

entropy() → torch.Tensor

implementation of *TorchProbabilityDistribution* interface

kl(*other:* maze.distributions.torch_dist.TorchProbabilityDistribution) → torch.Tensor

implementation of *TorchProbabilityDistribution* interface

log_prob(*actions: torch.Tensor*) → torch.Tensor

implementation of *TorchProbabilityDistribution* interface

classmethod required_logits_shape(*action_space: gym.spaces.Space*) → Sequence[int]

implementation of *TorchProbabilityDistribution* interface

sample() → torch.Tensor

implementation of *TorchProbabilityDistribution* interface

These are combined probability distributions:

MultiCategoricalProbabilityDistribution	Multi-categorical probability distribution.
DictProbabilityDistribution	Dictionary probability distribution.

MultiCategoricalProbabilityDistribution

class maze.distributions.multi_categorical.**MultiCategoricalProbabilityDistribution**(*logits: torch.Tenso ac- tion_space gym.spaces tem- per- a- ture: float*)

Multi-categorical probability distribution.

The respective functions either return aggregated properties across the sub-distributions using a reduce_fun such as mean or sum.

Parameters **logits** – The concatenated action selection logits for all sub spaces.

deterministic_sample() → Dict[str, torch.Tensor]

implementation of *TorchProbabilityDistribution* interface

entropy(*reduce_fun: callable = torch.mean*) → torch.Tensor

implementation of *TorchProbabilityDistribution* interface

kl(*other: maze.distributions.multi_categorical.MultiCategoricalProbabilityDistribution, reduce_fun: callable = torch.mean*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

log_prob(*actions: List[torch.Tensor]*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

neg_log_prob(*actions: List[torch.Tensor]*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

classmethod required_logits_shape(*action_space: gym.spaces.MultiDiscrete*) → Sequence[int]
implementation of *TorchProbabilityDistribution* interface

sample() → List[torch.Tensor]
implementation of *TorchProbabilityDistribution* interface

DictProbabilityDistribution

class maze.distributions.dict.**DictProbabilityDistribution**(*distribution_dict: Dict[str, maze.distributions.distribution.ProbabilityDistrib*

Dictionary probability distribution.

The respective functions either return

- the per key distribution properties or
- aggregate the properties across the sub-distributions using a reduce_fun such as mean or sum.

Parameters distribution_dict – dictionary holding sub-probability distributions.

deterministic_sample() → Dict[str, torch.Tensor]
implementation of *TorchProbabilityDistribution* interface

entropy(*reduce_fun: callable = torch.mean*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

kl(*other: maze.distributions.dict.DictProbabilityDistribution, reduce_fun: callable = torch.mean*) → torch.Tensor
implementation of *TorchProbabilityDistribution* interface

log_prob(*actions: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *TorchProbabilityDistribution* interface

neg_log_prob(*actions: Dict[str, torch.Tensor]*) → Dict[str, torch.Tensor]
implementation of *TorchProbabilityDistribution* interface

sample() → Dict[str, torch.Tensor]
implementation of *TorchProbabilityDistribution* interface

1.4.11 Core Utilities

These are general interfaces, classes and utility functions:

override	Annotation for documenting method overrides.
unused	Function to annotate unused variables.
set_random_states	Set random states of all random generators used in the framework.
flat_structured_space	Compiles a flat gym.spaces.Dict space from a structured environment space.
flat_structured_shapes	Flatten a dict of shape dicts to a single dict
read_config	Read YAML file into a dict
list_to_dict	Convert lists to int-indexed dicts.
EnvFactory	Helper class to instantiate an environment from configuration with the help of the Registry.
make_env_from_hydra	Create an environment instance from the hydra configuration, given the overrides.
Registry	Supports the creation of different modules that can be plugged into the environments (like demand generators or reward schemes) and can instantiate them from parameters read from config files.

override

class maze.core.annotations.**override**(*cls: Type*)

Annotation for documenting method overrides.

Parameters **cls** – The superclass that provides the overridden method. If this cls does not actually have the method, an error is raised.

unused

class maze.core.annotations.**unused**(**args*)

Function to annotate unused variables. Also disables the 'unused parameter/value' inspection warning.

set_random_states

class maze.core.utils.seeding.**set_random_states**(*seed: int*)

Set random states of all random generators used in the framework.

Param seed: the seed integer initializing the random number generators.

flat_structured_space

class `maze.core.utils.structured_env_utils.`**`flat_structured_space`**(*structured_space_dict: Dict[Union[int, str], gym.spaces.Dict]*)

Compiles a flat gym.spaces.Dict space from a structured environment space. :param: The structured dictionary spaces. :return: The flattened dictionary space.

flat_structured_shapes

class `maze.core.utils.structured_env_utils.`**`flat_structured_shapes`**(*shapes: Dict[Union[int, str], Dict[str, Sequence[int]]]*)

Flatten a dict of shape dicts to a single dict

Parameters **`shapes`** – Collection of shape dict.

Returns Flat shape dict.

read_config

class `maze.core.utils.config_utils.`**`read_config`**(*path: Union[pathlib.Path, str]*)

Read YAML file into a dict

Parameters **`path`** – Path of the file to read

Returns Dict with the YAML file contents

list_to_dict

class `maze.core.utils.config_utils.`**`list_to_dict`**(*list_or_dict: Union[list, Mapping]*)

Convert lists to int-indexed dicts.

Code is simplified by supporting only one universal data structure instead of implementing code paths for lists and dicts separately.

Parameters **`list_or_dict`** – The list to convert to dict. If it is already a dict, the dict is returned without modification.

Returns The passed list as dict.

EnvFactory

class `maze.core.utils.config_utils.`**`EnvFactory`**(*env: Union[None, str, Mapping[str, Any], Any], wrappers: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]]*)

Helper class to instantiate an environment from configuration with the help of the Registry.

Parameters

- **`env`** – environment configuration
- **`wrappers`** – collection of wrappers as configuration

make_env_from_hydra

class `maze.core.utils.config_utils.`**`make_env_from_hydra`**(*config_module: str, config_name: str = None, **hydra_overrides: str*)

Create an environment instance from the hydra configuration, given the overrides. :param config_module: Python module path of the hydra configuration package :param config_name: Name of the defaults configuration yaml within *config_module* :param hydra_overrides: Overrides as kwargs, e.g. env="cartpole", configuration="test" :return: The newly instantiated environment

Registry

class `maze.core.utils.registry.`**`Registry`**(**args, **kwds*)

Supports the creation of different modules that can be plugged into the environments (like demand generators or reward schemes) and can instantiate them from parameters read from config files.

Parameters

- **`root_module`** – Starting point for search for suitable classes to be registered.
- **`base_type`** – A common interface (parent class) of the registered types (e.g. DemandGenerator)

`arg_to_collection`(*arg: Union[List[Union[None, str, Mapping[str, Any], Any]], Mapping[str, Union[None, str, Mapping[str, Any], Any]]], **kwargs*) → Dict[Union[str, int], BaseType]

Instantiates objects specified in a list or dictionary.

`arg_to_obj`(*arg: Union[None, str, Mapping[str, Any], Any], config: Mapping[str, Any] = None, **kwargs*) → BaseType

Converts arg (usually passed to constructor of an env) to an instantiated class.

- If arg is already instantiated, just returns it
- If arg is a string, then construct an instance according to given type_registry and config parameters
- If arg is a dict-like configuration, construct a new instance. The type is identified by the reserved attribute `type` and the remaining attributes are passed to the constructor.

Parameters

- **`arg`** – Either - an instantiated object inheriting from base_type - a string, e.g. 'static', usually together with the `config` argument - a dict-like configuration, specifying the type name in the reserved attribute `type` together with the constructor arguments. (e.g. `{ 'class': 'static_demand', 'constructor_argument': 1, ... }`)
- **`config`** – Config to pass to the constructor if arg is a string.
- **`kwargs`** – Additional arguments that are merged with the configuration dictionary (useful to sideload objects that can not conveniently be specified in the config, e.g. a shared RandomState)

Returns arg if already instantiated, new object otherwise (see the *build_obj* method above)

classmethod build_obj(*klass_or_callable: Union[Type[BaseType], Callable], instance_or_config: Union[None, str, Mapping[str, Any], Any] = None, **kwargs*) → BaseType

Given a class, init an instance of that class with given keyword arguments.

Parameters

- **klass_or_callable** – Class to instantiate, or alternatively a function returning instance of the registry base type class.
- **instance_or_config** – Either already an actual instance of klass or keyword arguments to provide
- **kwargs** – Additional arguments that are merged with the configuration dictionary (useful to sideload objects that can not conveniently be specified in the config, e.g. a shared RandomState)

Returns Instance of the given class

classmethod callable_from_path(*path_string: str*) → Callable[[. . .], Any]

Attempt to import a callable from the specified path.

Parameters **path_string** – Path to the callable to import.

Returns Imported callable.

class_type_from_module_name(*module_name: Union[str, Type[BaseType]]*) → Type[BaseType]

Import the given module and lookup the class from the module with the correct base type.

The implementation expects exactly one matching class per module. A ValueError is returned otherwise. If the module name is not valid, a ModuleNotFoundError is triggered.

Parameters **module_name** – Absolute module path (e.g. `maze_envs.logistics.content_based_replenishment.env.maze_env`)

Returns The one and only class from the given module that derives from base_type.

collect_modules(*root_module: Any, base_type: Type[BaseType]*)

Populates a registry dictionary, by walking the specified root module.

Parameters

- **root_module** – Starting point for search for suitable classes to be registered.
- **base_type** – Class restriction. Registered classes/modules have to be of type klass.

Returns A dictionary with class name -> class type for all registered valid sub-classes.

1.4.12 Utilities

A collection of smaller auxiliary functions and classes:

SimpleStatsLoggingSetup	Helper class to simplify the statistics logging setup.

SimpleStatsLoggingSetup

class maze.utils.log_stats_utils.**SimpleStatsLoggingSetup**(*env: maze.core.log_stats.log_stats_env.LogStatsEnv, log_dir: str = None*)

Helper class to simplify the statistics logging setup. All statistics defined for the given env are sent to a console writer.

Limitation: It can only handle a single environment.

1.4.13 Trainers and Training Runners

This page contains the reference documentation for trainers and training runners:

Overview

- *General*
- *Trainers*
  - *Actor-Critics (AC)*
  - *Evolutionary Strategies (ES)*
  - *Imitation Learning (IL) and Learning from Demonstrations (LfD)*
- *Utilities*

General

These are general interfaces, classes and utility functions for trainers and training runners:

`Trainer`	Interface for trainers.
`TrainingRunner`	Base class for training runner implementations.
`TrainConfig`	Top-level configuration structure.
`ModelConfig`	Model configuration structure.
`AlgorithmConfig`	Base class for all specific algorithm configurations.
`ModelSelectionBase`	Base class for model selection strategies.
`BestModelSelection`	Best model selection strategy.

Trainer

class maze.train.trainers.common.trainer.**Trainer**

Interface for trainers.

abstract load_state(*file_path: Union[str, BinaryIO]*) → None

Load state from file. This is required for resuming training or model fine tuning with different parameters.

Parameters **file_path** – Path from where to load the state.

TrainingRunner

class maze.train.trainers.common.training_runner.**TrainingRunner**(*state_dict_dump_file: str, spaces_config_dump_file: str, normalization_samples: int*)

Base class for training runner implementations.

normalization_samples: int
Number of samples (=steps) to collect normalization statistics at the beginning of the training.

run(*cfg: omegaconf.DictConfig*) → None
While this method is designed to be overriden by individual subclasses, it provides some functionality that is useful in general:

- Building the env factory for env + wrappers
- Estimating normalization statistics from the env
- If successfully estimated, wrapping the env factory so that envs are already built with the statistics
- Building the model composer from model config and env spaces config
- Serializing the env spaces configuration (so that the model composer can be re-loaded for future rollout)
- Initializing logging setup

Parameters `cfg` – Full Hydra run job config

spaces_config_dump_file: str
Where to save the env spaces configuration (output directory handled by hydra)

state_dict_dump_file: str
Where to save the best model (output directory handled by hydra)

TrainConfig

class maze.train.trainers.common.training_runner.**TrainConfig**(*env: omegaconf.DictConfig, model: maze.train.trainers.common.training_runner, algorithm: maze.train.trainers.common.training_runner, runner: maze.runner.Runner*)

Top-level configuration structure.

The structured configuration support of hydra is limited currently (1.0-rc2).

E.g.

- Merging different configuration files did not work as expected (e.g. algorithm and env-algorithm)
- Although the entry-point expects a TrainConfig object, it just receives a DictConfig, which can cause unexpected behaviour.

Note that due to this limitations, this class merely acts as type hinting mechanism. Behind the scenes we receive raw DictConfig objects and either need to invoke the `Registry` functionality or `hydra.utils.instantiate` to instantiated objects of specific types where required.

ModelConfig

class `maze.train.trainers.common.training_runner.`**`ModelConfig`**(*policies: Dict[Any, Any], critics: Optional[Dict[Any, Any]], distribution_mapper: Dict[Any, Any]*)

Model configuration structure.

As with TrainConfig this class enables type hinting, but is not actually instantiated.

AlgorithmConfig

class `maze.train.trainers.common.training_runner.`**`AlgorithmConfig`**

Base class for all specific algorithm configurations.

ModelSelectionBase

class `maze.train.trainers.common.model_selection.model_selection_base.`**`ModelSelectionBase`**

Base class for model selection strategies.

`update`(*reward: float*) → None

Receives a new evaluation result from the model.

Parameters **`reward`** – mean evaluation reward

BestModelSelection

class `maze.train.trainers.common.model_selection.best_model_selection.`**`BestModelSelection`**(*du Op tio mo Op tio*

Best model selection strategy.

Parameters

- **`dump_file`** – Specifies the file path to dump the policy state for the best reward.
- **`model`** – The model to be dumped.

`update`(*reward: float*) → None

Implementation of ModelSelection.update().

Trainers

These are interfaces, classes and utility functions for built-in trainers:

Actor-Critics (AC)

MultiStepActorCritic	Base class for multi step actor critic.
MultiStepActorCriticEvents	Event interface, defining statistics emitted by the A2CTrainer.
MultiStepA2C	Multi step advantage actor critic.
A2CAlgorithmConfig	Algorithm parameters for multi-step A2C model.
MultiStepPPO	Multi step Proximal Policy Optimization.
PPOAlgorithmConfig	Algorithm parameters for multi-step PPO model.
MultiStepIMPALA	Multi step advantage actor critic.
ImpalaAlgorithmConfig	Algorithm parameters for Impala.
MultiStepIMPALAEvents	Events specific for the impala algorithm, in order to record and analyse it's behaviour in more detail
ImpalaLearner	Learner agent for Impala.
batch_outputs_time_major	Batch the collected output in time major format
log_probs_from_logits_and_actions_and_s	Computes action log-probs from policy logits, actions and acton_spaces.
from_logits	V-trace for softmax policies.
from_importance_weights	V-trace from log importance weights.
get_log_rhos	With the selected log_probs for multi-discrete actions of behavior and target policies we compute the log_rhos for calculating the vtrace.

MultiStepActorCritic

class maze.train.trainers.common.actor_critic.actor_critic_trainer.**MultiStepActorCritic**(*alg
Uni
maz
env
Uni
maz
maz
maz
eva
[<c
'ma
<cl
'ma
<cl
'ma
<cl
'ma
mod
maz
mod
Op-
tion
ini-
tial
Op-
tion
=
Nor*

Base class for multi step actor critic.

Parameters

- **algorithm_config** – Algorithm parameters.
- **env** – Distributed structured environment
- **eval_env** – Evaluation distributed structured environment
- **model** – Structured torch actor critic model.
- **initial_state** – path to initial state (policy weights, critic weights, optimizer state)
- **model_selection** – Optional model selection class, receives model evaluation results.

evaluate(*deterministic: bool, repeats: int*) → None
Perform evaluation on eval env.

Parameters

- **deterministic** – deterministic or stochastic action sampling (selection)
- **repeats** – number of evaluation episodes to average over

load_state(*file_path: Union[str, BinaryIO]*) → None
implementation of `Trainer`

load_state_dict(*state_dict: Dict*) → None
Set the model and optimizer state. :param state_dict: The state dict.

train() → None
Train policy using the synchronous advantage actor critic.

MultiStepActorCriticEvents

class maze.train.trainers.common.actor_critic.actor_critic_events.**MultiStepActorCriticEvent**
Event interface, defining statistics emitted by the A2CTrainer.

critic_grad_norm(*critic_id: int*, *value: float*)
gradient norm of the step critic

critic_value(*critic_id: int*, *value: float*)
critic value of the step critic

critic_value_loss(*critic_id: int*, *value: float*)
optimization loss of the step critic

learning_rate(*value: float*)
optimizer learning rate

policy_entropy(*step_id: int*, *value: float*)
entropy of the step policies

policy_grad_norm(*step_id: int*, *value: float*)
gradient norm of the step policies

policy_loss(*step_id: int*, *value: float*)
optimization loss of the step policy

time_epoch(*value: float*)
time required for epoch

time_rollout(*value: float*)
time required for rollout

time_update(*value: float*)
time required for update

MultiStepA2C

class maze.train.trainers.a2c.a2c_trainer.**MultiStepA2C**(*algorithm_config: maze.train.trainers.a2c.a2c_algorithm_config.A2CA env: Union[maze.train.parallelization.distributed_env.dist maze.core.env.structured_env.StructuredEnv, maze.core.env.structured_env_spaces_mixin.Structur maze.core.log_stats.log_stats_env.LogStatsEnv], eval_env: [<class 'maze.train.parallelization.distributed_env.distributed <class 'maze.core.env.structured_env.StructuredEnv'>, <class 'maze.core.env.structured_env_spaces_mixin.Structur <class 'maze.core.log_stats.log_stats_env.LogStatsEnv'>], model: maze.core.agent.torch_actor_critic.TorchActorCritic, model_selection: Optional[maze.train.trainers.common.model_selection. initial_state: Optional[str] = None*)

Multi step advantage actor critic.

Parameters

- **algorithm_config** – Algorithm parameters.
- **env** – Distributed structured environment
- **eval_env** – Evaluation distributed structured environment
- **model** – Structured torch actor critic model.
- **initial_state** – path to initial state (policy weights, critic weights, optimizer state)
- **model_selection** – Optional model selection class, receives model evaluation results.

A2CAlgorithmConfig

class maze.train.trainers.a2c.a2c_algorithm_config.**A2CAlgorithmConfig**(*n_epochs: int, epoch_length: int, deterministic_eval: bool, eval_repeats: int, patience: int, critic_burn_in_epochs: int, n_rollout_steps: int, lr: float, gamma: float, gae_lambda: float, policy_loss_coef: float, value_loss_coef: float, entropy_coef: float, max_grad_norm: float, device: str*)

Algorithm parameters for multi-step A2C model.

critic_burn_in_epochs: int
Number of critic (value function) burn in epochs

deterministic_eval: bool
run evaluation in deterministic mode (argmax-policy)

device: str
Either “cpu” or “cuda”

entropy_coef: float
weight of entropy loss

epoch_length: int
number of updates per epoch

eval_repeats: int

number of evaluation trials

gae_lambda: float
bias vs variance trade of factor for GAE

gamma: float
discounting factor

lr: float
learning rate

max_grad_norm: float
The maximum allowed gradient norm during training

n_epochs: int
number of epochs to train

n_rollout_steps: int
Number of steps taken for each rollout

patience: int
number of steps used for early stopping

policy_loss_coef: float
weight of policy loss

value_loss_coef: float
weight of value loss

MultiStepPPO

class maze.train.trainers.ppo.ppo_trainer.**MultiStepPPO**(*algorithm_config: maze.train.trainers.ppo.ppo_algorithm_config.PPOA env: Union[maze.train.parallelization.distributed_env.dist maze.core.env.structured_env.StructuredEnv, maze.core.env.structured_env_spaces_mixin.Structur maze.core.log_stats.log_stats_env.LogStatsEnv], eval_env: [<class 'maze.train.parallelization.distributed_env.distributed <class 'maze.core.env.structured_env.StructuredEnv'>, <class 'maze.core.env.structured_env_spaces_mixin.Structur <class 'maze.core.log_stats.log_stats_env.LogStatsEnv'>], model: maze.core.agent.torch_actor_critic.TorchActorCritic, model_selection: Optional[maze.train.trainers.common.model_selection.b initial_state: Optional[str] = None*)

Multi step Proximal Policy Optimization.

Parameters

- **algorithm_config** – Algorithm parameters.
- **env** – Distributed structured environment

- **`eval_env`** – Evaluation distributed structured environment
- **`model`** – Structured torch actor critic model.
- **`initial_state`** – path to initial state (policy weights, critic weights, optimizer state)
- **`model_selection`** – Optional model selection class, receives model evaluation results.

PPOAlgorithmConfig

class `maze.train.trainers.ppo.ppo_algorithm_config.`**`PPOAlgorithmConfig`**(*n_epochs: int, epoch_length: int, deterministic_eval: bool, eval_repeats: int, patience: int, critic_burn_in_epochs: int, n_rollout_steps: int, lr: float, gamma: float, gae_lambda: float, policy_loss_coef: float, value_loss_coef: float, entropy_coef: float, max_grad_norm: float, device: str, batch_size: int, n_optimization_epochs: int, clip_range: float*)

Algorithm parameters for multi-step PPO model.

batch_size: int
The batch size used for policy and value updates

clip_range: float
Clipping parameter of surrogate loss

critic_burn_in_epochs: int
Number of critic (value function) burn in epochs

deterministic_eval: bool
run evaluation in deterministic mode (argmax-policy)

device: str
Either "cpu" or "cuda"

entropy_coef: float
weight of entropy loss

epoch_length: int
number of updates per epoch

eval_repeats: int
number of evaluation trials

gae_lambda: float
bias vs variance trade of factor for GAE

gamma: float
discounting factor

lr: float
learning rate

max_grad_norm: float
The maximum allowed gradient norm during training

n_epochs: int
number of epochs to train

n_optimization_epochs: int
Number of epochs for for policy and value optimization

n_rollout_steps: int
Number of steps taken for each rollout

patience: int
number of steps used for early stopping

policy_loss_coef: float
weight of policy loss

value_loss_coef: float
weight of value loss

MultiStepIMPALA

class maze.train.trainers.impala.impala_trainer.**MultiStepIMPALA**(*model:* maze.core.agent.torch_actor_critic.Torc *rollout_actors:* maze.train.parallelization.distributed_ac *eval_env: Union[*maze.train.parallelization.distrib maze.core.env.structured_env.Structured maze.core.env.structured_env_spaces_m maze.core.log_stats.log_stats_env.LogS *options:* maze.train.trainers.impala.impala_algor

Multi step advantage actor critic.

Parameters

- **model** – Structured policy to train
- **rollout_actors** – Distributed actors for collection of training rollouts
- **eval_env** – Env to run evaluation on
- **options** – Algorithm options

evaluate(*deterministic: bool, repeats: int*) → None

Perform evaluation on eval env.

Parameters

- **deterministic** – deterministic or stochastic action sampling (selection)
- **repeats** – number of evaluation episodes to average over

load_state(*file_path: Union[str, BinaryIO]*) → None

implementation of `Trainer`

load_state_dict(*state_dict: Dict*) → None

Set the model and optimizer state. :param state_dict: The state dict.

train(*n_epochs: int, epoch_length: int, deterministic_eval: bool, eval_repeats: int, patience: Optional[int], model_selection: Optional[*maze.train.trainers.common.model_selection.best_model_selection.BestModelSelection*]*) → None

Train function that wraps normal train function in order to close all processes properly

Parameters

- **n_epochs** – number of epochs to train.
- **epoch_length** – number of updates per epoch.
- **deterministic_eval** – run evaluation in deterministic mode (argmax-policy)
- **eval_repeats** – number of evaluation trials
- **patience** – number of steps used for early stopping
- **model_selection** – Optional model selection class, receives model evaluation results

train_async(*n_epochs: int, epoch_length: int, deterministic_eval: bool, eval_repeats: int, patience: Optional[int], model_selection: Optional[maze.train.trainers.common.model_selection.best_model_selection.BestModelSelection]*) → None

Train policy using the synchronous advantage actor critic.

Parameters

- **n_epochs** – number of epochs to train.
- **epoch_length** – number of updates per epoch.
- **deterministic_eval** – run evaluation in deterministic mode (argmax-policy)
- **eval_repeats** – number of evaluation trials
- **patience** – number of steps used for early stopping
- **model_selection** – Optional model selection class, receives model evaluation results

ImpalaAlgorithmConfig

class maze.train.trainers.impala.impala_algorithm_config.**ImpalaAlgorithmConfig**(*n_epochs: int, epoch_length: int, deterministic_eval: bool, eval_repeats: int, eval_concurrenc int, queue_out_of_sy float, patience: int, n_rollout_steps: int = 50, actors_batch_size: int = 2, num_actors: int = 2, lr: float = 0.0002, gamma: float = 0.98, policy_loss_coef: float = 1.0, value_loss_coef: float = 0.5, entropy_coef: float = 0.00025, max_grad_norm. float = 0,*

actors_batch_size: `int` = 2
: number of actors to combine to one batch

deterministic_eval: `bool`
: run evaluation in deterministic mode (argmax-policy)

device: `str` = 'cpu'
: Device of the learner (either cpu or cuda). Note that the actors collecting rollouts are always run on CPU.

entropy_coef: `float` = 0.00025
: coefficient of the entropy used in the loss calculation

epoch_length: `int`
: number of updates per epoch

eval_concurrency: `int`
: Number of concurrently executed evaluation environments.

eval_repeats: `int`
: number of evaluation trials

gamma: `float` = 0.98
: discount factor

lr: `float` = 0.0002
: learning rate

max_grad_norm: `float` = 0
: max grad norm for gradient clipping, ignored if value==0

n_epochs: `int`
: number of epochs to train

n_rollout_steps: `int` = 50
: number of rolloutstep of each epoch substep

num_actors: `int` = 2
: number of actors to be run

patience: `int`
: number of steps used for early stopping

policy_loss_coef: `float` = 1.0
: coefficient of the policy used in the loss calculation

queue_out_of_sync_factor: `float`
: this factor multiplied by the actor_batch_size gives the size of the queue for the agents output collected by the learner. Therefor if the all rollouts computed can be at most (queue_out_of_sync_factor + num_agents/actor_batch_size) out of sync with learner policy

reward_clipping: `str` = 'abs_one'
: the type of reward clipping to be used, options 'abs_one', 'soft_asymmetric', 'None'

value_loss_coef: `float` = 0.5
: coefficient of the value used in the loss calculation

vtrace_clip_pg_rho_threshold: `float` = 1.0
: A scalar float32 tensor with the clipping threshold on rho_s in rho_s delta log pi(a|x) (r + gamma v_{s+1} - V(x_sfrom_importance_weights)). If None, no clipping is applied.

vtrace_clip_rho_threshold: `float` = 1.0
: A scalar float32 tensor with the clipping threshold for importance weights (rho) when calculating the baseline targets (vs). rho^bar in the paper. If None, no clipping is applied.

MultiStepIMPALAEvents

class maze.train.trainers.impala.impala_events.**MultiStepIMPALAEvents**

Events specific for the impala algorithm, in order to record and analyse it's behaviour in more detail

critic_grad_norm(*critic_key: Union[int, str], value: float*)

Record the critic gradient norm

Parameters

- **critic_key** – the key of the critic
- **value** – the value

critic_value(*critic_key: Union[int, str], value: float*)

Record the critic value

Parameters

- **critic_key** – the key of the critic
- **value** – the value

critic_value_loss(*critic_key: [<class 'int'>, <class 'str'>], value: float*)

Record the critic value loss

Parameters

- **critic_key** – the key of the critic
- **value** – the value

estimated_queue_sizes(*before: int, after: int*)

Record the estimated queue size before and after the collection of the actors output

Parameters

- **before** – the estimated queue size before collection
- **after** – the estimated queue size after collection

policy_entropy(*step_key: Union[int, str], value: float*)

Record the policy entropy

Parameters

- **step_key** – the step_key of the multi-step env
- **value** – the value

policy_grad_norm(*step_key: Union[int, str], value: float*)

Record the gradient norm

Parameters

- **step_key** – the step_key of the multi-step env
- **value** – the value

policy_loss(*step_key: Union[int, str], value: float*)

Record the policy loss

Parameters

- **step_key** – the step_key of the multi-step env
- **value** – the value

time_backprob(*time: float*, *percent: float*)
Record the total time it took the learner to backprob the loss + relative per to total update time

Parameters

- **time** – the absolute time it took for the computation
- **percent** – the relative percentage this computation took w.r.t. to one update

time_collecting_actors(*time: float*, *percent: float*)
Record the total time it took the learner to collect the actors output + relative per to total update time

Parameters

- **time** – the absolute time it took for the computation
- **percent** – the relative percentage this computation took w.r.t. to one update

time_dequeuing_actors(*time: float*, *percent: float*)
Record the time it took to dequeue the actors output from the synced queue + relative per to total update time

Parameters

- **time** – the absolute time it took for the computation
- **percent** – the relative percentage this computation took w.r.t. to one update

time_learner_rollout(*time: float*, *percent: float*)

Record the total time it took the learner to compute the logits on the agents output

- relative per to total update time

Parameters

- **time** – the absolute time it took for the computation
- **percent** – the relative percentage this computation took w.r.t. to one update

time_loss_computation(*time: float*, *percent: float*)
Record the total time it took the learner compute the loss + relative per to total update time

Parameters

- **time** – the absolute time it took for the computation
- **percent** – the relative percentage this computation took w.r.t. to one update

ImpalaLearner

class maze.train.trainers.impala.impala_learner.**ImpalaLearner**(*eval_env: Union[*maze.train.parallelization.distribute
maze.core.env.structured_env.StructuredEn
maze.core.env.structured_env_spaces_mix
maze.core.log_stats.log_stats_env.LogStat
model: maze.core.agent.torch_actor_critic.TorchA
n_rollout_steps: int)

Learner agent for Impala. The agent only exists once (in the main thread) and is in charge of doing the loss computation as computing and backpropagating the gradients. Furthermore it holds critic network in contrast to the actors.

evaluate(*deterministic: bool*, *repeats: int*) → None

Perform evaluation on eval env.

Parameters

- **deterministic** – deterministic or stochastic action sampling (selection)
- **repeats** – number of evaluation episodes to average over

learner_rollout_on_agent_output(*actors_output: maze.train.parallelization.distributed_actors.actor.AgentOutput*) → maze.train.trainers.impala.impala_learner.LearnerOutput

Compute the values and the action logits using the learners network parameters and the actors rollouts.
Thus we never step through an env here.

Parameters **actors_output** – The collected and batched actors output, including the env_outputs such as observations and actions

Returns A LearnerOutput names tuple consisting of (values, detached_values, actions_logits, n_critics)

batch_outputs_time_major

class maze.train.trainers.impala.impala_batching.**batch_outputs_time_major**(*actor_outputs: List[maze.train.paralleli* *learner_device: str*)

Batch the collected output in time major format

Parameters

- **actor_outputs** – A list of actor outputs (e.g. rollouts consisting of observations, actions_taken, infos, action_logtis, rewards and dones)
- **learner_device** – the device ('cpu' or 'cuda') of the learner

Returns An ActorOutput Named tuple where the the list of input rollouts has been batched in the second dim.

log_probs_from_logits_and_actions_and_spaces

class maze.train.trainers.impala.impala_vtrace.**log_probs_from_logits_and_actions_and_spaces**

Computes action log-probs from policy logits, actions and acton_spaces.

In the notation used throughout documentation and comments, T refers to the time dimension ranging from 0 to T-1. B refers to the batch size and NUM_ACTIONS refers to the number of actions.

Parameters

- **policy_logits** – A list (w.r.t. the substeps of the env) of dicts (w.r.t. the actions) of tensors of un-normalized log-probabilities (shape list[dict[str,[T, B, NUM_ACTIONS]]])
- **actions** – An list (w.r.t. the substeps of the env) of dicts (w.r.t. the actions) of tensors (list[dict[str,[T, B]]])
- **action_spaces** – A list (w.r.t. the substeps of the env) of the action spaces
- **distribution_mapper** – A distribution mapper providing a mapping of action heads to distributions.

Returns A list (w.r.t. the substeps of the env) of dicts (w.r.t. the actions) of tensors of shape [T, B] corresponding to the sampling log probability of the chosen action w.r.t. the policy. And a list (w.r.t. the substeps of the env) of DictProbability distributions corresponding to the step-action-distributions.

from_logits

class maze.train.trainers.impala.impala_vtrace.**from_logits**(*behaviour_policy_logits: Mapping[Union[str, int], Dict[str, torch.Tensor]], target_policy_logits: Mapping[Union[str, int], Dict[str, torch.Tensor]], actions: Mapping[Union[str, int], Dict[str, torch.Tensor]], action_spaces: Mapping[Union[str, int], gym.spaces.Space], distribution_mapper:* maze.distributions.distribution_mapper.Distribu *discounts: torch.Tensor, rewards: torch.Tensor, values: Mapping[Union[str, int], torch.Tensor], bootstrap_value: Mapping[Union[str, int], torch.Tensor], clip_rho_threshold: Optional[float], clip_pg_rho_threshold: Optional[float], device: Optional[str]*)

V-trace for softmax policies.

Calculates V-trace actor critic targets for softmax polices as described in

"IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures" by Espeholt, Soyer, Munos et al.

Target policy refers to the policy we are interested in improving and behaviour policy refers to the policy that generated the given rewards and actions.

In the notation used throughout documentation and comments, T refers to the time dimension ranging from 0 to T-1. B refers to the batch size and ACTION_SPACE refers to the list of numbers each representing a number of actions.

Parameters

- **behaviour_policy_logits** – A list (w.r.t. the substeps of the env) of dict (w.r.t. the actions) of tensors of un-normalized log-probabilities (shape list[dict[str,[T, B, NUM_ACTIONS]]])
- **target_policy_logits** – A list (w.r.t. the substeps of the env) of dict (w.r.t. the actions) of tensors of un-normalized log-probabilities (shape list[dict[str,[T, B, NUM_ACTIONS]]])
- **actions** – An list (w.r.t. the substeps of the env) of dicts (w.r.t. the actions) with actions sampled from the behavior policy. (list[dict[str,[T, B]]])

- **action_spaces** – A list (w.r.t. the substeps of the env) of the action spaces
- **distribution_mapper** – A distribution mapper providing a mapping of action heads to distributions.
- **discounts** – A float32 tensor of shape [T, B] with the discount encountered when following the behavior policy.
- **rewards** – A float32 tensor of shape [T, B] with the rewards generated by following the behavior policy.
- **values** – A list (w.r.t. the substeps of the env) of float32 tensors of shape [T, B] with the value function estimates wrt. the target policy.
- **bootstrap_value** – A list (w.r.t. the substeps of the env) of float32 tensors of shape [B] with the value function estimate at time T.
- **clip_rho_threshold** – A scalar float32 tensor with the clipping threshold for importance weights (rho) when calculating the baseline targets (vs). rho^bar in the paper.
- **clip_pg_rho_threshold** – A scalar float32 tensor with the clipping threshold on rho_s in: rho_s delta log pi(a|x) (r + gamma v_{s+1} - V(x_s)).
- **device** – the device the results should be sent to before returning it

Returns A *VTraceFromLogitsReturns* namedtuple with the following fields: vs: A list (w.r.t. the substeps of the env) of float32 tensors of shape [T, B]. Can be used as target to train a baseline (V(x_t) - vs_t)^2. pg_advantages: A list (w.r.t. the substeps of the env) of float32 tensors of shape [T, B]. Can be used as an estimate of the advantage in the calculation of policy gradients. log_rhos: A list (w.r.t. the substeps of the env) of float32 tensors of shape [T, B] containing the log importance sampling weights (log rhos). behaviour_action_log_probs: A list (w.r.t. the substeps of the env) of float32 tensors of shape [T, B] containing the behaviour policy action log probabilities (log mu(a_t)). target_action_log_probs: A list (w.r.t. the substeps of the env) of float32 tensors of shape [T, B] containing target policy action probabilities (log pi(a_t)). target_step_action_dists: A list (w.r.t. the substeps of the env) of the action probability distributions w.r.t. to the target policy

from_importance_weights

class maze.train.trainers.impala.impala_vtrace.**from_importance_weights**(*log_rhos: torch.Tensor*, *discounts: torch.Tensor*, *rewards: torch.Tensor*, *values: torch.Tensor*, *bootstrap_value: torch.Tensor*, *clip_rho_threshold: Optional[float]*, *clip_pg_rho_threshold: Optional[float]*)

V-trace from log importance weights.

Calculates V-trace actor critic targets as described in

"IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures" by Espeholt, Soyer, Munos et al.

In the notation used throughout documentation and comments, T refers to the time dimension ranging from 0 to T-1. B refers to the batch size. This code also supports the case where all tensors have the same number of additional dimensions, e.g., *rewards* is [T, B, C], *values* is [T, B, C], *bootstrap_value* is [B, C].

Parameters

- **log_rhos** – A float32 tensor of shape [T, B] representing the log importance sampling weights, i.e. log(target_policy(a) / behaviour_policy(a)). V-trace performs operations on rhos in log-space for numerical stability.
- **discounts** – A float32 tensor of shape [T, B] with discounts encountered when following the behaviour policy.
- **rewards** – A float32 tensor of shape [T, B] containing rewards generated by following the behaviour policy.
- **values** – A float32 tensor of shape [T, B] with the value function estimates wrt. the target policy.
- **bootstrap_value** – A float32 of shape [B] with the value function estimate at time T.
- **clip_rho_threshold** – A scalar float32 tensor with the clipping threshold for importance weights (rho) when calculating the baseline targets (vs). rho^bar in the paper. If None, no clipping is applied.
- **clip_pg_rho_threshold** – A scalar float32 tensor with the clipping threshold on rho_s in rho_s delta log pi(a|x) (r + gamma v_{s+1} - V(x_sfrom_importance_weights)). If None, no clipping is applied.

Returns A VTraceReturns namedtuple (vs, pg_advantages) where: vs: A float32 tensor of shape [T, B]. Can be used as target to train a baseline (V(x_t) - vs_t)^2. pg_advantages: A float32 tensor of shape [T, B]. Can be used as the advantage in the calculation of policy gradients.

get_log_rhos

class maze.train.trainers.impala.impala_vtrace.**get_log_rhos**(*target_action_log_probs: Mapping[Union[str, int], Dict[str, torch.Tensor]], behaviour_action_log_probs: Mapping[Union[str, int], Dict[str, torch.Tensor]]*)

With the selected log_probs for multi-discrete actions of behavior and target policies we compute the log_rhos for calculating the vtrace.

Parameters

- **target_action_log_probs** – A list (w.r.t. the substeps of the env) of dicts (w.r.t. the actions) of tensors of shape [T, B] corresponding to the sampling log probability of the chosen action w.r.t. the target policy.
- **behaviour_action_log_probs** – A list (w.r.t. the substeps of the env) of dicts (w.r.t. the actions) of tensors of shape [T, B] corresponding to the sampling log probability of the chosen action w.r.t. the behaviour policy.

Returns a list (w.r.t. the substeps of the env) of tensors, where each tensor is of the shape [T,B]

Evolutionary Strategies (ES)

ESTrainer	Trainer class for OpenAI Evolution Strategies.
ESAlgorithmConfig	Algorithm parameters for evolution strategies model.
ESEvents	Event interface, defining statistics emitted by the ES-Trainer.
ESMasterRunner	Baseclass of ES training master runners (serves as basis for dev and other runners).
ESDevRunner	Runner config for single-threaded training, based on ESDummyDistributedRollouts.
SharedNoiseTable	A fixed length vector of deterministically generated pseudo-random floats.
Optimizer	Abstract baseclass of an optimizer to be used with ES.
SGD	Stochastic gradient descent with momentum
Adam	Adam optimizer
ESRolloutResult	Result structure for distributed rollouts.
ESDummyDistributedRollouts	Implementation of the ES distribution by running the rollouts synchronously in the same process.
ESDistributedRollouts	Abstract base class of ES rollout distribution.
ESAbortException	This exception is raised if the current rollout is intentionally aborted.
ESRolloutWorkerWrapper	The rollout generation is bound to a single worker environment by implementing it as a Wrapper class.
get_flat_parameters	Get the parameters of all sub-policies as a single flat vector.

continues on next page

Table 44 – continued from previous page

set_flat_parameters	Overwrite the parameters of all sub-policies by a single flat vector.

ESTrainer

class maze.train.trainers.es.es_trainer.**ESTrainer**(*algorithm_config:* maze.train.trainers.es.es_algorithm_config.ESAlgorithmCon *policy:* maze.core.agent.torch_policy.TorchPolicy, *shared_noise:* maze.train.trainers.es.es_shared_noise_table.SharedNoiseTal *normalization_stats: Optional[Dict[str, Tuple[numpy.ndarray, numpy.ndarray]]]*)

Trainer class for OpenAI Evolution Strategies.

Parameters

- **algorithm_config** – Algorithm parameters.
- **policy** – Multi-step policy encapsulating the policy networks
- **shared_noise** – The noise table, with the same content for every worker and the master.
- **normalization_stats** – Normalization statistics as calculated by the NormalizeObservationWrapper.

load_state(*file_path: Union[str, BinaryIO]*) → None
implementation of *Trainer*

load_state_dict(*state_dict: Dict*) → None
Set the model and optimizer state. :param state_dict: The state dict.

train(*distributed_rollouts:* maze.train.trainers.es.distributed.es_distributed_rollouts.ESDistributedRollouts, *model_selection: Optional[*maze.train.trainers.common.model_selection.model_selection_base.ModelSelectionBase*]*) → None
Run the ES training loop.

Parameters

- **distributed_rollouts** – The distribution interface for experience collection.
- **model_selection** – Optional model selection class, receives model evaluation results.

ESAlgorithmConfig

class maze.train.trainers.es.es_algorithm_config.**ESAlgorithmConfig**(*n_rollouts_per_update: int*, *n_timesteps_per_update: int*, *max_epochs: int*, *max_steps: int*, *optimizer: Any*, *l2_penalty: float*, *noise_stddev: float*)

Algorithm parameters for evolution strategies model.

l2_penalty: float
L2 weight regularization coefficient.

max_epochs: int
The number of epochs to train before termination. Pass 0 to train indefinitely.

max_steps: int
Limit the episode rollouts to a maximum number of steps. Set to 0 to disable this option.

n_rollouts_per_update: int
Minimum number of episode rollouts per training iteration (=epoch).

n_timesteps_per_update: int
Minimum number of cumulative env steps per training iteration (=epoch). The training iteration is only finished, once the given number of episodes AND the given number of steps has been reached. One of the two parameters can be set to 0.

noise_stddev: float
The scaling factor of the random noise applied during training.

optimizer: Any
The optimizer to use to update the policy based on the sampled gradient.

ESEvents

class maze.train.trainers.es.es_events.**ESEvents**
Event interface, defining statistics emitted by the ESTrainer.

policy_grad_norm(*policy_id: int*, *value: float*)
gradient norm of the step policies

policy_norm(*policy_id: int*, *value: float*)
l2 norm of the step policy parameters

real_time(*value: float*)
elapsed real time per iteration (=epoch)

update_ratio(*value: float*)
norm(optimizer step) / norm(all parameters)

ESMasterRunner

class maze.train.trainers.es.es_runners.**ESMasterRunner**(*state_dict_dump_file: str, spaces_config_dump_file: str, normalization_samples: int, shared_noise_table_size: int*)

Baseclass of ES training master runners (serves as basis for dev and other runners).

abstract create_distributed_rollouts(*env: Union[maze.core.env.structured_env.StructuredEnv, maze.core.env.structured_env_spaces_mixin.StructuredEnvSpacesMixin], shared_noise: maze.train.trainers.es.es_shared_noise_table.SharedNoiseT* → *maze.train.trainers.es.distributed.es_distributed_rollouts.ESDistributed*

Abstract method, derived runners like ESDevRunner return an appropriate rollout generator.

Parameters

- **env** – the one and only environment
- **shared_noise** – noise table to be shared by all workers

Returns a newly instantiated rollout generator

run(*cfg: omegaconf.DictConfig*) → None

Run the training master node.

shared_noise_table_size: int

Number of float values in the deterministically generated pseudo-random table (250.000.000 x 32bit floats = 1GB)

ESDevRunner

class maze.train.trainers.es.es_runners.**ESDevRunner**(*state_dict_dump_file: str, spaces_config_dump_file: str, normalization_samples: int, shared_noise_table_size: int, n_eval_rollouts: int*)

Runner config for single-threaded training, based on ESDummyDistributedRollouts.

create_distributed_rollouts(*env: Union[maze.core.env.structured_env.StructuredEnv, maze.core.env.structured_env_spaces_mixin.StructuredEnvSpacesMixin], shared_noise: maze.train.trainers.es.es_shared_noise_table.SharedNoiseTable*) → *maze.train.trainers.es.distributed.es_distributed_rollouts.ESDistributedRollouts*

use single-threaded rollout generation

n_eval_rollouts: int

Fixed number of evaluation runs per epoch.

SharedNoiseTable

class maze.train.trainers.es.es_shared_noise_table.**SharedNoiseTable**(*count: int = 250000000*)

A fixed length vector of deterministically generated pseudo-random floats.

This enables a communication strategy for the distributed training, that allows to transfer noise table indices instead of full gradient vectors.

Parameters **count** – Number of float values in the fixed length table (250.000.000 x 32bit floats = 1GB)

get(*i: int, dim: int*) → numpy.ndarray

Get the pseudo-random sequence at table index i.

Parameters

- **i** – start index within the table
- **dim** – desired vector length

:return A noise vector with length dim

sample_index(*random_state: numpy.random.RandomState*) → int

Sample a random index within the table, taking into account the size of the noise vector.

Parameters **random_state** – random generator to be used

Returns A noise index to be passed to *maze.train.trainers.es.es_shared_noise_table.SharedNoiseTable.get()*.

Optimizer

class maze.train.trainers.es.optimizers.base_optimizer.**Optimizer**

Abstract baseclass of an optimizer to be used with ES.

setup(*policy:* maze.core.agent.torch_policy.TorchPolicy) → None

Two-stage construction to enable construction from config-files.

Parameters **policy** – ES policy network to optimize

update(*global_gradient: numpy.ndarray*) → float

Execute one update step.

Parameters **global_gradient** – A flat gradient vector

:return update ratio = norm(optimizer step) / norm(theta)

SGD

class maze.train.trainers.es.optimizers.sgd.**SGD**(*step_size: float, momentum: float = 0.9*)

Stochastic gradient descent with momentum

setup(*policy:* maze.core.agent.torch_policy.TorchPolicy) → None

prepare optimizer for training

Adam

class maze.train.trainers.es.optimizers.adam.**Adam**(*step_size*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*)

Adam optimizer

setup(*policy: maze.core.agent.torch_policy.TorchPolicy*) → None

prepare optimizer for training

ESRolloutResult

class maze.train.trainers.es.distributed.es_distributed_rollouts.**ESRolloutResult**(*is_eval: bool*)

Result structure for distributed rollouts.

ESDummyDistributedRollouts

class maze.train.trainers.es.distributed.es_dummy_distributed_rollouts.**ESDummyDistributedRo**

Implementation of the ES distribution by running the rollouts synchronously in the same process.

generate_rollouts(*policy: maze.core.agent.torch_policy.TorchPolicy*, *max_steps: Optional[int]*, *noise_stddev: float*, *normalization_stats: Dict[str, Dict[str, Union[numpy.ndarray, float, int, Iterable[Union[float, int]]]]]*) → Generator[*maze.train.trainers.es.distributed.es_distributed_rollouts.ESRolloutResult*, None, None]

First execute a fixed number of eval rollouts and then continue with producing training samples.

ESDistributedRollouts

class maze.train.trainers.es.distributed.es_distributed_rollouts.**ESDistributedRollouts**

Abstract base class of ES rollout distribution.

abstract generate_rollouts(*policy: maze.core.agent.torch_policy.TorchPolicy*, *max_steps: Optional[int]*, *noise_stddev: float*, *normalization_stats: Dict[str, Tuple[numpy.ndarray, numpy.ndarray]]*) → Generator[*maze.train.trainers.es.distributed.es_distributed_rollouts.ESRolloutResult*, None, None]

Declare a new rollout task and start producing results that can be obtained from the returned generator.

Note that different distribution strategies have different ways of balancing evaluation and training rollouts.

Parameters

- **policy** – Multi-step policy encapsulating the policy networks
- **max_steps** – Optionally limit the rollout to a number of environment steps (horizon).
- **noise_stddev** – The standard deviation of the applied parameter noise.
- **normalization_stats** – Normalization statistics as calculated by the NormalizeObservationWrapper.

ESAbortException

class maze.train.trainers.es.distributed.es_rollout_wrapper.**ESAbortException**

This exception is raised if the current rollout is intentionally aborted.

ESRolloutWorkerWrapper

class maze.train.trainers.es.distributed.es_rollout_wrapper.**ESRolloutWorkerWrapper**(**args*, ***kwds*)

The rollout generation is bound to a single worker environment by implementing it as a Wrapper class.

clear_abort()

Clear the abort flag.

generate_evaluation(*policy: maze.core.agent.torch_policy.TorchPolicy*) → *maze.train.trainers.es.distributed.es_distributed_rollouts.ESRolloutResult*

Generate a single evaluation rollout.

Parameters **policy** – Multi-step policy encapsulating the policy networks

:return A result set with a single evaluation rollout

generate_training(*policy: maze.core.agent.torch_policy.TorchPolicy*, *noise_stddev: float*) → *maze.train.trainers.es.distributed.es_distributed_rollouts.ESRolloutResult*

Generate a single training sample, consisting of two rollouts, obtained by adding and subtracting the same random perturbation vector from the policy.

Parameters

- **policy** – Multi-step policy encapsulating the policy networks.
- **noise_stddev** – The standard deviation of the applied parameter noise.

:return A result set with a pair of rollouts generated by adding/subtracting the perturbations
(antithetic sampling)

rollout(*policy: maze.core.agent.torch_policy.TorchPolicy*) → None

Use the passed policy to step the environment until it is done.

This method does not return any results, query the episode statistics instead to process the results.

Parameters **policy** – Multi-step policy encapsulating the policy networks

set_abort()

Abort the rollout (intended to be called from a thread).

get_flat_parameters

class maze.train.trainers.es.es_utils.**get_flat_parameters**(*policy: maze.core.agent.torch_policy.TorchPolicy*)

Get the parameters of all sub-policies as a single flat vector.

Parameters **policy** – source policy

Returns flattened parameters

set_flat_parameters

class maze.train.trainers.es.es_utils.**set_flat_parameters**(*policy: maze.core.agent.torch_policy.TorchPolicy, flat_params: torch.Tensor*)

Overwrite the parameters of all sub-policies by a single flat vector.

Parameters

- **policy** – target policy
- **flat_params** – concatenated vector

Imitation Learning (IL) and Learning from Demonstrations (LfD)

ImitationEvents	Event interface defining statistics emitted by the imitation learning trainers.
ImitationEvaluator	Abstract interface for imitation learning evaluation.
ImitationRunner	Dev runner for imitation learning.
ParallelLoadedImitationDataset	A version of the in-memory dataset that loads all data in parallel.
DataLoadWorker	Data loading worker used to map states to actual observations.
InMemoryImitationDataSet	Trajectory data set for imitation learning.
BCTrainer	Trainer for behavioral cloning learning.
BCAlgorithmConfig	Algorithm parameters for behavioral cloning.
BCEvaluator	Evaluates a given policy on validation data.
BCLoss	Loss function for behavioral cloning.

ImitationEvents

class maze.train.trainers.imitation.imitation_events.**ImitationEvents**

Event interface defining statistics emitted by the imitation learning trainers.

box_mean_abs_deviation(*step_id: Union[str, int], subspace_name: str, value: int*)
Mean absolute deviation for box (continuous) subspaces.

discrete_accuracy(*step_id: Union[str, int], subspace_name: str, value: int*)
Accuracy for discrete (categorical) subspaces.

multi_binary_accuracy(*step_id: Union[str, int], subspace_name: str, value: int*)
Accuracy for multi-binary subspaces.

policy_entropy(*step_id: Union[str, int], value: float*)
Entropy of the step policies.

policy_grad_norm(*step_id: Union[str, int], value: float*)
Gradient norm of the step policies.

policy_l2_norm(*step_id: Union[str, int], value: float*)
L2 norm of the step policies.

policy_loss(*step_id: Union[str, int], value: float*)
Optimization loss of the step policy.

ImitationEvaluator

class maze.train.trainers.imitation.imitation_evaluator.**ImitationEvaluator**

Abstract interface for imitation learning evaluation.

evaluate(*policy: maze.core.agent.torch_policy.TorchPolicy*) → None

Evaluate given policy (results are stored in stat logs) and dump the model if the reward improved.

Parameters **policy** – Policy to evaluate

ImitationRunner

class maze.train.trainers.imitation.imitation_runners.**ImitationRunner**(*state_dict_dump_file: str, spaces_config_dump_file: str, normalization_samples: int, dataset: omegaconf.DictConfig*)

Dev runner for imitation learning.

Loads the given trajectory data and trains a policy on top of it using supervised learning.

dataset: omegaconf.DictConfig

Specify the Dataset class used to load the trajectory data for training

run(*cfg: omegaconf.DictConfig*) → None

Run the training master node.

ParallelLoadedImitationDataset

class maze.train.trainers.imitation.parallel_loaded_im_data_set.**ParallelLoadedImitationData**

A version of the in-memory dataset that loads all data in parallel.

This significantly speeds up data loading in cases where conversion of MazeStates and MazeActions into actions and observations is demanding.

Parameters

- **trajectory_data_dir** – See the parent class.
- **env_factory** – See the parent class.
- **n_workers** – Number of worker processes to load data in.

DataLoadWorker

class maze.train.trainers.imitation.parallel_loaded_im_data_set.**DataLoadWorker**

Data loading worker used to map states to actual observations.

static run(*env_factory: Callable*, *trajectory_file_paths: List[Union[pathlib.Path, str]]*, *reporting_queue: multiprocessing.context.BaseContext.Queue*) → None

Load trajectory data from the provided trajectory file paths. Report exceptions to the main process.

Parameters

- **env_factory** – Function for creating an environment for MazeState and MazeAction conversion.
- **trajectory_file_paths** – Which trajectory data files should this worker load and process.
- **reporting_queue** – Queue for reporting loaded data and exceptions back to the main process.

InMemoryImitationDataSet

class maze.train.trainers.imitation.in_memory_data_set.**InMemoryImitationDataSet**(**args: Any*, ***kwargs: Any*)

Trajectory data set for imitation learning.

Loads all data on initialization and then keeps it in memory.

Parameters

- **trajectory_data_dir** – The directory where the trajectory data are stored.
- **env_factory** – Function for creating an environment for state and action conversion. For Maze envs, the environment configuration (i.e. space interfaces, wrappers etc.) determines the format of the actions and observations that will be derived from the recorded MazeActions and MazeStates (e.g. multi-step observations/actions etc.).

static get_trajectory_files(*trajectory_data_dir: str*) → List[pathlib.Path]

List pickle files ("pkl" suffix, used for trajectory data storage by default) in the given directory.

Parameters **trajectory_data_dir** – Where to look for the trajectory records (= pickle files).

Returns A list of available pkl files in the given directory.

static load_episode_record(*env: maze.core.env.structured_env.StructuredEnv*, *episode_record: maze.core.trajectory_recorder.episode_record.EpisodeRecord*) → Tuple[List[Dict[Union[int, str], Any]], List[Dict[Union[int, str], Any]]]

Convert an episode trajectory record into an array of observations and actions using the given env.

Parameters

- **env** – Env to use for conversion of MazeStates and MazeActions into observations and actions
- **episode_record** – Episode record to load

Returns Loaded observations and actions. I.e., a tuple (observation_list, action_list). Each of the lists contains observation/action dictionaries, with keys corresponding to IDs of structured sub-steps. (I.e., the dictionary will have just one entry for non-structured scenarios.)

random_split(*lengths: Sequence[int], generator: torch.Generator = torch.default_generator*) → List[torch.utils.data.dataset.Subset]

Randomly split the dataset into non-overlapping new datasets of given lengths.

The split is based on episodes – samples from the same episode will end up in the same subset. Based on the available episode lengths, this might result in subsets of slightly different lengths than specified.

Optionally fix the generator for reproducible results, e.g.:

self.random_split([3, 7], generator=torch.Generator().manual_seed(42))

Parameters

- **lengths** – lengths of splits to be produced (best effort, the result might differ based on available episode lengths
- **generator** – Generator used for the random permutation.

Returns A list of the data subsets, each with size roughly (!) corresponding to what was specified by lengths.

BCTrainer

class maze.train.trainers.imitation.bc_trainer.**BCTrainer**(*data_loader: torch.utils.data.dataloader.DataLoader, policy: maze.core.agent.torch_policy.TorchPolicy, optimizer: torch.optim.optimizer.Optimizer, loss: maze.train.trainers.imitation.bc_loss.BCLoss, train_stats: maze.core.log_stats.log_stats.LogStatsAggregator = <maze.core.log_stats.log_stats.LogStatsAggregat object>, imitation_events: maze.train.trainers.imitation.imitation_events.Imi = <abc.ImitationEventsProxy object>*)

Trainer for behavioral cloning learning.

Runs training on top of provided trajectory data and rolls out the policy using the provided evaluator.

In structured (multi-step) envs, all policies are trained simultaneously based on the substep actions and observation present in the trajectory data.

data_loader: torch.utils.data.dataloader.DataLoader

Data loader for loading trajectory data.

imitation_events: ***maze.train.trainers.imitation.imitation_events.ImitationEvents*** **= <a**

Imitation-specific training events

load_state(*file_path: Union[str, BinaryIO]*) → None
implementation of `Trainer`

load_state_dict(*state_dict: Dict*) → None
Set the model and optimizer state. :param state_dict: The state dict.

loss: *maze.train.trainers.imitation.bc_loss.BCLoss*
Class providing the training loss function.

optimizer: **torch.optim.optimizer.Optimizer**
Optimizer to use

policy: *maze.core.agent.torch_policy.TorchPolicy*
Structured policy to train.

train(*n_epochs: int, evaluator: maze.train.trainers.imitation.imitation_evaluator.ImitationEvaluator, eval_every_k_iterations: int = None*) → None
Run training.

Parameters

- **n_epochs** – How many epochs to train for
- **evaluator** – Evaluator to use for evaluation rollouts
- **eval_every_k_iterations** – Number of iterations after which to run evaluation (in addition to evaluations at the end of each epoch, which are run automatically). If set to None, evaluations will run on epoch end only.

train_stats: *maze.core.log_stats.log_stats.LogStatsAggregator* **= <maze.core.log_stats.**
Training statistics

BCAlgorithmConfig

class maze.train.trainers.imitation.bc_algorithm_config.**BCAlgorithmConfig**(*device: str, batch_size: int, n_eval_workers: int, validation_percentage: float, n_epochs: int, eval_every_k_iterations: int, n_eval_episodes: int, max_episode_steps: int, optimizer: Any*)
Algorithm parameters for behavioral cloning.

batch_size: int
Batch size for training

device: str
Either “cpu” or “cuda”

eval_every_k_iterations: int
Number of iterations after which to run evaluation (in addition to evaluations at the end of each epoch, which are run automatically). If set to None, evaluations will run on epoch end only.

max_episode_steps: int
Max number of steps per episode to run during each evaluation rollout

n_epochs: int
number of epochs to train

n_eval_episodes: int
Number of episodes to run during each evaluation rollout

n_eval_workers: int
Number of workers to perform evaluation runs in. If set to 1, evaluation is performed in the main process.

optimizer: Any
The optimizer to use to update the policy.

validation_percentage: float
Percentage of the data used for validation.

BCEvaluator

class maze.train.trainers.imitation.bc_evaluator.**BCEvaluator**(*loss:* maze.train.trainers.imitation.bc_loss.BCLos *model_selection:* *Op-tional[*maze.train.trainers.common.model_s *data_loader:* *torch.utils.data.DataLoader*)

Evaluates a given policy on validation data.

Parameters

- **data_loader** – The data used for evaluation.
- **loss** – Loss function to be used.
- **model_selection** – Model selection interface that will be notified of the recorded rewards.

evaluate(*policy:* maze.core.agent.torch_policy.TorchPolicy) → None
Evaluate given policy (results are stored in stat logs) and dump the model if the reward improved.

Parameters **policy** – Policy to evaluate

BCLoss

class maze.train.trainers.imitation.bc_loss.**BCLoss**(*action_spaces_dict: Dict[Union[int, str], gym.spaces.Dict], loss_discrete: torch.nn.Module = torch.nn.CrossEntropyLoss, loss_box: torch.nn.Module = torch.nn.MSELoss, loss_multi_binary: torch.nn.Module = torch.nn.functional.binary_cross_entropy_with_logits*)

Loss function for behavioral cloning.

action_spaces_dict: Dict[Union[int, str], gym.spaces.Dict]
Action space we are training on (used to determine appropriate loss functions)

calculate_loss(*policy: maze.core.agent.torch_policy.TorchPolicy, observation_dict: Dict[Union[int, str], Any], action_dict: Dict[Union[int, str], Any], events: maze.train.trainers.imitation.imitation_events.ImitationEvents*) → torch.Tensor
Calculate and return the training loss for one step (= multiple sub-steps in structured scenarios).

Parameters

- **policy** – Structured policy to evaluate
- **observation_dict** – Dictionary with observations identified by substep ID
- **action_dict** – Dictionary with actions identified by substep ID
- **events** –

Returns Total loss

Utilities

stack_numpy_dict_list	Stack list of dictionaries holding numpy arrays as values.
unstack_numpy_list_dict	Inverse of *stack_numpy_dict_list()*.
compute_gradient_norm	Computes the cumulative gradient norm of all provided parameters.
stack_torch_dict_list	Stack list of dictionaries holding torch tensors as values.

stack_numpy_dict_list

class maze.train.utils.train_utils.**stack_numpy_dict_list**(*dict_list: List[Dict[str, numpy.ndarray]], expand: bool = False*)

Stack list of dictionaries holding numpy arrays as values.

Parameters

- **dict_list** – A list of identical dictionaries to be stacked, e.g. [{a: 1}, {a: 2}]
- **expand** – If True the values are expended by one dimension at dimension zero.

Returns The list entries as a stacked dictionary, e.g. {a : [1, 2]}

unstack_numpy_list_dict

class maze.train.utils.train_utils.**unstack_numpy_list_dict**(*list_dict: Dict[str, numpy.ndarray]*)

Inverse of `stack_numpy_dict_list()`.

Converts a dict of stacked lists (e.g. {a : [1, 2]}) into a list of dicts (e.g. [{a: 1}, {a: 2}]).

Parameters **list_dict** – Dict of stacked lists, e.g. {a : [1, 2]}

Returns List of dicts, e.g. [{a: 1}, {a: 2}]

compute_gradient_norm

class maze.train.utils.train_utils.**compute_gradient_norm**(*params: Iterable[torch.Tensor]*)

Computes the cumulative gradient norm of all provided parameters.

Parameters **params** – Iterable over model parameters.

Returns The cumulative gradient norm.

stack_torch_dict_list

class maze.train.utils.train_utils.**stack_torch_dict_list**(*dict_list: List[Dict[str, Union[torch.Tensor, numpy.ndarray]]], expand: bool = False, dim: int = 0*)

Stack list of dictionaries holding torch tensors as values.

Similar to `stack_numpy_dict_list()`, but for tensors.

Parameters

- **dict_list** – A list of identical dictionaries to be stacked.
- **expand** – If True the values are expended by one dimension at dimension :param dim.
- **dim** – The dimension in which to stack/concat the lists.

Returns The list entries as a stacked dictionary.

1.4.14 Parallelization

This page contains the reference documentation for the parallelization module.

`ObservationAggregator`	Observation aggregator used in distributed training for aggregating observations of multiple instances of the same environment.
`BaseWorker`	This class holds a policy as well an env in order to step through the env, by producing action from the policy and recoding the rollout to be processed by the learner.
`BaseWorkerOutput`	Base class for outputs generated by the agent.

ObservationAggregator

class `maze.train.parallelization.observation_aggregator.`**`ObservationAggregator`**

Observation aggregator used in distributed training for aggregating observations of multiple instances of the same environment.

abstract aggregate() → Any

This function aggregates the collected list of observations.

reset(*observations: List[Any] = None*) → None

Reset aggregator.

Parameters **`observations`** – a list of observations.

BaseWorker

class `maze.train.parallelization.base_worker.`**`BaseWorker`**

This class holds a policy as well an env in order to step through the env, by producing action from the policy and recoding the rollout to be processed by the learner.

abstract rollout() → Union[Tuple[*maze.train.parallelization.base_worker.BaseWorkerOutput*, List], Tuple[numpy.ndarray, List]]

Interface to performs an agent rollout, that is sample actions, step through the env for a maximum of n_rollout_steps and collect data.

Returns This rollout as an ActorOutput or array of ActorOutputs

abstract update_policy(*state_dict: Dict*) → None

Update the policy with the given state dict.

Parameters **`state_dict`** – State dict to load.

BaseWorkerOutput

class `maze.train.parallelization.base_worker.`**`BaseWorkerOutput`**(*observations: Dict[Union[str, int], Dict[str, torch.Tensor]], actions_taken: Dict[Union[str, int], Dict[str, torch.Tensor]], rewards: torch.Tensor, dones: torch.Tensor, infos: List[Any]*)

Base class for outputs generated by the agent.

Parameters

- **`observations`** – Observations collected during the rollout.
- **`actions_taken`** – Actions taken during the rollout.
- **`rewards`** – Rewards collected during the rollout.
- **`dones`** – Dones collected during the rollout.

- **infos** – Infos collected during the rollout.

static get_dict_dict_obj_attr_names() → List[str]

Retrieve the attribute names of the actor output fields that have dict dict structure.

Returns A list of all attributes having a dict-dict structure.

static get_list_obj_attr_names() → List[str]

Retrieve the attribute names of the actor output fields that have list structure.

Returns A list of all attributes having a list structure.

static get_tensor_obj_attr_names() → List[str]

Retrieve the attribute names of the actor output fields that have tensor structure.

Returns A list of all attributes having a tensor structure.

to(*device: str*) → None

Cast all elements to the given device.

Parameters **device** – The device to put the output on (cpu or cuda).

Distributed Environments

These are interfaces, classes and utility functions for distributed environments:

BaseDistributedEnv	Abstract base class for distributed environments.
DummyStructuredDistributedEnv	Creates a simple wrapper for multiple environments, calling each environment in sequence on the current Python process.

BaseDistributedEnv

class maze.train.parallelization.distributed_env.distributed_env.**BaseDistributedEnv**(*num_envs*
int)

Abstract base class for distributed environments.

Param num_envs: the number of distributed environments.

abstract reset()

Reset all the environments and return respective observations in env-aggregated form.

Returns observations in env-aggregated form.

abstract seed(*seed: int = None*) → None

Sets the seed for this distributed env's random number generator(s) and its contained parallel envs.

abstract step(*actions: Iterable[Any]*) → Tuple[Dict[str, numpy.ndarray], numpy.ndarray, numpy.ndarray, Iterable[Dict[Any, Any]]]

Step the environments with the given actions.

Parameters **actions** – the list of actions for the respective envs.

Returns observations, rewards, dones, information-dicts all in env-aggregated form.

DummyStructuredDistributedEnv

class maze.train.parallelization.distributed_env.dummy_distributed_env.**DummyStructuredDist**

Creates a simple wrapper for multiple environments, calling each environment in sequence on the current Python process. This is useful for computationally simple environment such as `cartpole-v1`, as the overhead of multiprocess or multi-thread outweighs the environment computation time. This can also be used for RL methods that require a vectorized environment, but that you want a single environments to train with.

Parameters **env_factories** – A list of functions that will create the environments (each callable returns a *MultiStepEnvironment* instance when called).

property action_space
implementation of *StructuredEnvSpacesMixin* interface

property action_spaces_dict
Return the action space of one of the distributed envs.

actor_id() → List[Tuple[Union[str, int], int]]
Return the actor id tuples of all envs in a list.

close() → None
BaseDistributedEnv implementation

get_stats(*level: maze.core.log_stats.log_stats.LogStatsLevel = <LogStatsLevel.EPOCH: 3>*) → *maze.core.log_stats.log_stats.LogStatsAggregator*
Returns the aggregator of the individual episode statistics emitted by the parallel envs.

Parameters **level** – Must be set to *LogStatsLevel.EPOCH*, step or episode statistics are not propagated

get_stats_value(*event: Callable, level:* maze.core.log_stats.log_stats.LogStatsLevel, *name: Optional[str] = None*) → Union[int, float, numpy.ndarray, dict]
Obtain a single value from the epoch statistics dict.

Parameters

- **event** – The event interface method of the value in question.
- **name** – The *output_name* of the statistics in case it has been specified in *maze.core.log_stats.event_decorators.define_epoch_stats()*
- **level** – Must be set to *LogStatsLevel.EPOCH*, step or episode statistics are not propagated.

is_actor_done() → numpy.ndarray
Return the done flags of all actors in a list.

property observation_space
implementation of *StructuredEnvSpacesMixin* interface

property observation_spaces_dict
Return the observation space of one of the distributed envs.

reset() → Dict[str, numpy.ndarray]
BaseDistributedEnv implementation

seed(*seed: int = None*) → None
BaseDistributedEnv implementation

step(*actions: List[Any]*) → Tuple[Dict[str, numpy.ndarray], numpy.ndarray, numpy.ndarray, Iterable[Dict[Any, Any]]]
Step the environments with the given actions.

Parameters **actions** – the list of actions for the respective envs.

Returns observations, rewards, dones, information-dicts all in env-aggregated form.

write_epoch_stats()
Trigger the epoch statistics generation.

Distributed Actors

These are interfaces, classes and utility functions for distributed actors:

ActorAgent	Steps through a given environment and records rollouts.
BaseDistributedActors	The base class for all distributed actors.

ActorAgent

class maze.train.parallelization.distributed_actors.actor.**ActorAgent**(*env_factory: Callable, policy: maze.core.agent.torch_policy.To n_rollout_steps: int*)

Steps through a given environment and records rollouts. Designed to be used in distributed rollouts.

rollout() → maze.train.parallelization.distributed_actors.actor.AgentOutput_w_stats
Performs a agent rollout, that is sample actions and step through the env for a maximum of n_rollout_steps. This rollout (observations, rewards, dones, infos, actions_taken, actions_logits) is returned

Returns This rollout (observations, rewards, dones, infos, actions_taken, actions_logits) as an ActorOutput named tuple

update_policy(*state_dict: Dict*) → NoReturn
Update the policy with the given state dict.

Parameters **state_dict** – State dict to load.

BaseDistributedActors

class maze.train.parallelization.distributed_actors.distributed_actors.**BaseDistributedActo**

The base class for all distributed actors.

Distributed actors run rollouts independently. Rollouts are recorded and made available in batches to be used during training. When a new policy version is made available, it is distributed to all actors.

Parameters

- **env_factory** – Factory function for envs to run rollouts on
- **policy** – Structured policy to sample actions from
- **n_rollout_steps** – Number of rollouts steps to record in one rollout
- **n_actors** – Number of distributed actors to run simultaneously
- **batch_size** – Size of the batch the rollouts are collected in

abstract broadcast_updated_policy(*state_dict: Dict*) → None

Broadcast the newest version of the policy to the actors.

Parameters **state_dict** – State of the new policy version to broadcast.

abstract collect_outputs(*learner_device: str*) → Tuple[maze.train.parallelization.distributed_actors.actor.AgentOutp
float, float, float]

Collect *self.batch_size* actor outputs from the queue and return them batched where the first dim is time and the second is the batch size.

Parameters **learner_device** – the device of the learner

Returns A tuple of (1) batched version of ActorOutputs, (2) queue size before de-queueing, (3) queue size after dequeueing, and (4) the time it took to dequeue the outputs

get_epoch_stats_aggregator() → *maze.core.log_stats.log_stats.LogStatsAggregator*

Return the collected epoch stats aggregator

get_stats_value(*event: Callable*, *level:* maze.core.log_stats.log_stats.LogStatsLevel, *name: Optional[str] = None*) → Union[int, float, numpy.ndarray, dict]

Obtain a single value from the epoch statistics dict.

Parameters

- **event** – The event interface method of the value in question.
- **name** – The *output_name* of the statistics in case it has been specified in *maze.core.log_stats.event_decorators.define_epoch_stats()*

- **`level`** – Must be set to *LogStatsLevel.EPOCH*, step or episode statistics are not propagated.

`abstract start``()` → None
Start all distributed actors

`abstract stop``()` → None
Stop all distributed actors

- For installing Maze just follow the *installation instructions*.
- To see Maze in action check out *a first example*.
- For a more applied introduction visit the *step by step tutorial*.

You can also find an extensive overview of Maze in the *table of contents* as well as the *API documentation*.

CHAPTER

TWO

SPOTLIGHTS

Below we list of some of Maze's key features. The list is far from exhaustive but none the less a nice starting point to dive into the framework.

- Configure your applications and experiments with the *Hydra config system* .
- Design and visualize your policy and value networks with the *Perception Module*.
- *Pre-process* and *normalize* your observations without writing boiler plate code.
- Stick to your favourite tools and trainers by *combining Maze with other RL frameworks*.
- Although Maze supports more complex *environment structures* you can of course still *integrate existing Gym environments* .
- Scale your training runs with Ray and Kubernetes .

Warning: This is a preliminary, non-stable release of Maze. It is not yet complete and not all of our interfaces have settled yet. Hence, there might be some breaking changes on our way towards the first stable release.

This project is powered by | *Any questions or feedback, just get in touch*

CHAPTER
THREE

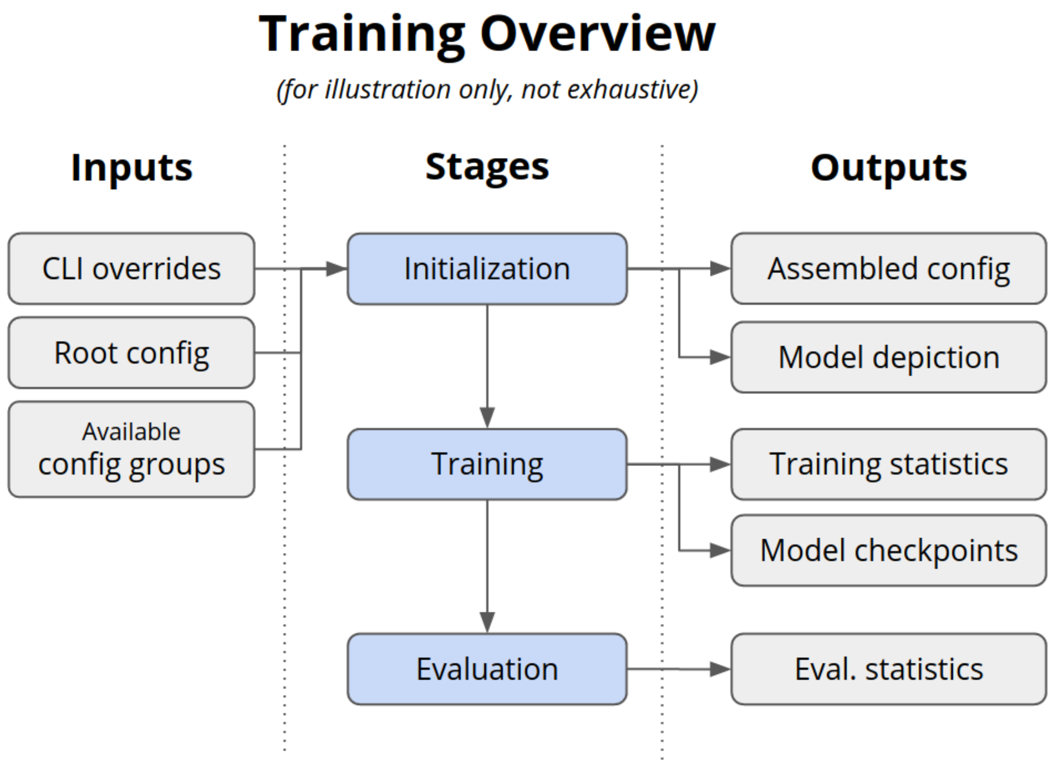
DOCUMENTATION OVERVIEW

Below you find an overview of the general Maze framework documentation, which is beyond the *API documentation*. The listed pages motivate and explain the underlying concepts but most importantly also provide code snippets and minimum working examples to quickly get you started.

3.1 Training

Here, we show how to train a policy on a standard Gym or custom environment using algorithms and models from Maze. This guide focuses on the main mechanics of Maze training runs, plus also gives some pointers on how to customize the training with custom environments (using the tutorial Maze 2D-cutting environment as an example), models, etc.

The figure below shows a conceptual overview of the Maze training workflow.



On this page:

- *The first example* demostrates training with the default settings. The main purpose is to show how the Maze training pipeline works in general.
- *The second example* explains how you can customize training on standard Gym and Maze environments (for which configuration files are already provided by Maze).
- The following section then explains what you need to *customize training for your own project*, including custom components and configuration files.
- Finally, the last section shows *how to launch training directly from Python* (avoiding the CLI).

In order to fully understand the configuration mechanisms used here, you should familiarize yourself with how *Maze makes use of the Hydra configuration framework*.

3.1.1 Example 1: Your First Training Run

We can train a policy from scratch on the Cartpole environment with default settings using the following command:

```
$ maze-run -cn conf_train env=gym_env env.name=CartPole-v0
```

The `-cn conf_train` argument specifies that we would like to use `conf_train.yaml` as our root config file. This is needed as by default, configuration for *rollouts* is used.

Furthermore, we specify that `gym_env` configuration should be used, with `CartPole-v0` as the Gym environment name. (For more information on how to read and customize the default configuration files, see *Hydra overview*.)

Such a training run consists of these main stages, loaded based on the default configuration provided by Maze:

1. The full configuration is assembled via Hydra based on the config files available, the defaults set in root config, and the overrides you provide via CLI (see *Hydra overview<hydra-overview>* to understand more about this process).
2. Hydra creates the output directory where all output files will be stored.
3. The full configuration of the job is logged: (1) to standard output, (2) as a text entry to your Tensorboard logs, and (3) as a YAML file in the output directory.
4. If the observation normalization wrappers is present, *observation normalization* statistics are collected and stored (note that no wrappers are applied by default).
5. Policies and critics are initialized and their graphical depictions saved.
6. The training starts, statistics are displayed in console and stored to a Tensorboard file, and current best model versions are saved (by default to `state_dict.pt` file).
7. Once the training is done, final evaluation runs are performed and final model versions saved. (When the training is done depends on the *training runner*. Usually, this is specified using the `runner.n_epochs` argument, but the training can also end with early stopping if there is no more improvement).

As the job is running, you should see the statistics from the training and evaluation runs printed in your console, as mentioned in the 6. step:

```
...
********** Iteration 3 **********
 step|path                                                                                     ␣
↪        |               value
=====|==================================================================================|======
    4|eval                  DiscreteActionEvents  action                  substep_0/
↪action      |     [len:281, :0.5]
    4|eval                  BaseEnvEvents         reward                  median_step_
↪count      |               18.500
```

(continues on next page)

(continued from previous page)

```
   4|eval                  BaseEnvEvents          reward                 mean_step_
↪count       |              28.100
   4|eval                  BaseEnvEvents          reward                 total_step_
↪count       |            928.000
   4|eval                  BaseEnvEvents          reward                 total_episode_
↪count    |             40.000
   4|eval                  BaseEnvEvents          reward                 episode_count␣
↪         |             10.000
   4|eval                  BaseEnvEvents          reward                 std           ␣
↪         |             16.447
   4|eval                  BaseEnvEvents          reward                 mean          ␣
↪         |             28.100
   4|eval                  BaseEnvEvents          reward                 min           ␣
↪         |             16.000
   4|eval                  BaseEnvEvents          reward                 max           ␣
↪         |             66.000
-> new overall best model 28.10000!
...
```

This main structure remains similar for all environment and training configurations.

3.1.2 Example 2: Customizing with Provided Components

When your Maze job is launched using `maze-run` from the CLI, the following happens under the hood:

1. A job configuration is assembled by putting available configuration files together with the overrides you specify as arguments to the run command. More on that can be found in *configuration documentation page*, specifically in *Hydra overview*.
2. The complete assembled configuration is handed over to the *Maze runner* specified in the configuration (in the `runner` group). This runner then launches and manages the training (or any other) job.

The common points for customizing the training run correspond to the configuration groups listed in the training root config file, namely:

- Environment (`env` configuration group), configuring which environment the training runs on, as well as customizing any other inner configuration of the environment, if available (like raw piece size in 2D cuttting environment)
- Training algorithm (`algorithm` configuration group), specifying the algorithm used and configuration for it
- Model (`model` configuration group), specifying how the models for policies and (optionally) critics should be assembled
- Runner (`runner` configuration group), specifying options for how the training is run (e.g. locally, in development mode, or using Ray on a Kubernetes cluster). The runner is also the main object responsible for administering the whole training run (and runners are thus specific to individual algorithms used).

Maze provides a host of configuration files useful for working with standard Gym environments and environments provided by Maze (such as the 2D cutting environment). Hence, to use these, it suffices to supply appropriate *overrides*, without writing any additional configuration files.

By default, the `gym_env` configuration is used, which allows us to specify the Gym env that we would like to instantiate:

```
$ maze-run -cn conf_train env=gym_env env.name=LunarLander-v2
```

With appropriate overrides, we can also include vector observation model and wrappers (providing normalization):

```
$ maze-run -cn conf_train env=gym_env env.name=LunarLander-v0 wrappers=gym_pixel_env␣
↪model=gym_pixel_env
```

Alternatively, we could use the *tutorial Cutting 2D environment*:

```
$ maze-run -cn conf_train env=tutorial_cutting_2d_struct_masked \
  wrappers=tutorial_cutting_2d model=tutorial_cutting_2d_struct_masked
```

Further, by default, the algorithm used is Evolution Strategies (the implementation is provided by Maze). To use a different algorithm, e.g. PPO with a shared critic, we just need to add the appropriate overrides:

```
$ maze-run -cn conf_train algorithm=ppo env=tutorial_cutting_2d_struct_masked \
  wrappers=tutorial_cutting_2d model=tutorial_cutting_2d_struct_masked
```

To see all the configuration files available out-of-the-box, check out the `maze/conf` package.

3.1.3 Training in Your Custom Project

While the default environments and configurations are nice to get started quickly or test different approaches in standard scenarios, the primary focus of Maze are fully custom environments and models solving real-world problems (which are of course much more fun as well!).

The best place to start with a custom environment is the *Maze step by step* tutorial (mentioned already in the previous section) showing how to implement a custom Maze environment from scratch, along with respective configuration files (see also *Hydra: Your Own Configuration Files*).

Then, you can easily launch your environment by supplying your own configuration file (here we use one from the tutorial):

```
$ maze-run -cn conf_train env=tutorial_cutting_2d_struct_masked \
  wrappers=tutorial_cutting_2d model=tutorial_cutting_2d_struct_masked
```

For links to more customization options (like building custom models with *Maze Perception Module*), check out the *Where to Go Next* section.

While customizing other configuration groups listed in the previous section (e.g., `algorithm`, `runner`) is not needed as often, all of these can be customized in an analogous way (i.e., implement your own components that plug into the framework instead of the default ones, and then add your own config to be able to configure them from the command line).

3.1.4 Plain Python Training

In most use cases, it will probably be more convenient to launch training directly from the CLI and just implement your custom components (wrappers, environments, models, etc.) as needed. However, it is definitely possible to launch training also from Python, and the inner architecture of Maze should be sufficiently modular to allow you to extract just the parts that you want.

Because each of the algorithms included in Maze has slightly different needs, the usage will likely slightly differ. However, regardless of which algorithm you intend to use, the *`TrainingRunner`* subclasses offer good examples of what components you will need for launching training directly from Python.

Specifically, you'll need to concentrate on the `run` method, which takes as an argument the full assembled hydra configuration (which is printed to the command line every time you launch a job).

Usually, the run method does roughly the following:

- Instantiates the environment and policy components (some of this functionality is provided by the shared *TrainingRunner* superclass, as a large part of that is common for all training runners)
- Assembles the policy and critics into a structured policy
- Instantiates the trainer and any other components needed for training
- Launches the training

For example, this is the run method taken directly from the evolution strategies runner:

```
    @override(TrainingRunner)
    def run(self, cfg: DictConfig) -> None:
        """Run the training master node."""
        super().run(cfg)
        env = self.env_factory()

        # --- init the shared noise table ---
        print("********** Init Shared Noise Table **********")
        shared_noise = SharedNoiseTable(count=self.shared_noise_table_size)

        # --- initialize policies ---
        policy = TorchPolicy(networks=self.model_composer.policy.networks,
                             distribution_mapper=self.model_composer.distribution_
↪mapper,
                             device="cpu")

        print("********** Trainer Setup **********")
        trainer = ESTrainer(algorithm_config=cfg.algorithm,
                            policy=policy,
                            shared_noise=shared_noise,
                            normalization_stats=self.normalization_statistics)

        # initialize model from input_dir
        self._init_trainer_from_input_dir(trainer=trainer, state_dict_dump_file=self.
↪state_dict_dump_file,
                                          input_dir=cfg.input_dir)

        model_selection = BestModelSelection(dump_file=self.state_dict_dump_file,␣
↪model=policy)

        print("********** Run Trainer **********")
        # run with pseudo-distribution, without worker processes
        trainer.train(self.create_distributed_rollouts(env=env, shared_noise=shared_
↪noise),
                      model_selection=model_selection)
```

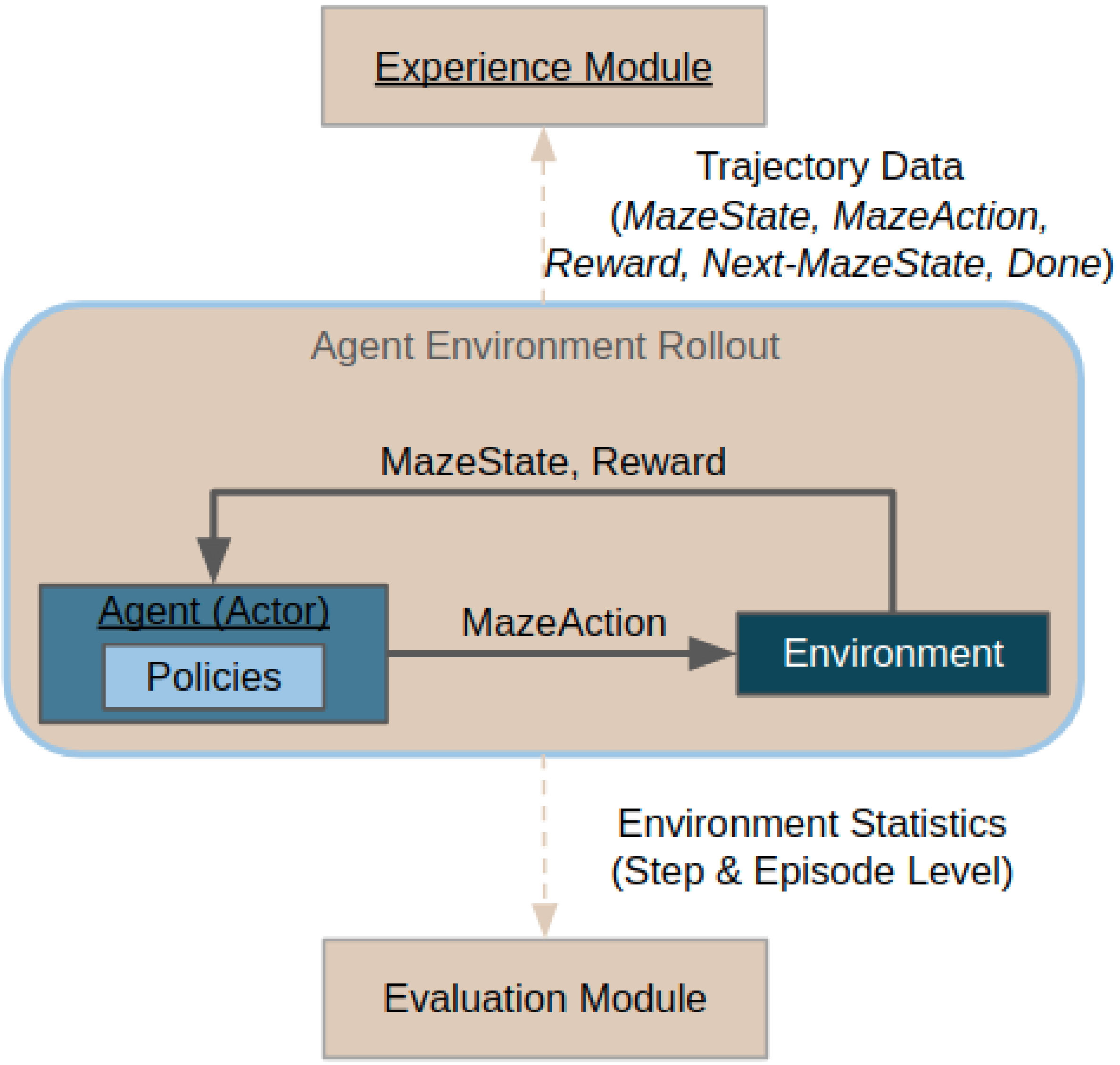
3.1.5 Where to Go Next

- After training, you might want to *roll out* the trained policy to further evaluate it or record the actions taken.
- To create a custom Maze environment, you might want to review *Maze environment hierarchy* and *creating a Maze environment from scratch*.
- To build custom Maze models, have a look at the *Maze Perception Module*.
- To better understand how to configure custom environments and other components of your project, you might want to review the more advanced parts of *configuration with Hydra*.

3.2 Rollouts

During rollouts, the agent interacts with a given environment, issuing actions obtained from a given policy (be it a heuristic or a trained policy).

Usually, the purpose of rollouts is either evaluation (or even deployment) of a given policy in a given environment, or collection of trajectory data. Collected trajectory data can later be used for further learning (e.g. *imitation learning*) or for inspecting the policy behavior more closely using *trajectory viewers*.



On this page:

- *The First Rollout* demostrates the main mechanics of running a rollout.
- *Rollout Runner Configuration* explains how to configure the rollout runners.
- *Environment and Agent Configuration* shows how to configure different environments and agents.
- Finally, *Plain Python Configuration* shows how to run rollouts without the CLI.

3.2.1 The First Rollout

Rollouts can be run from the command line, using the `maze-run` command. Rollout configuration (`conf_rollout`) is used by default. Hence, to run your first rollout, it suffices to execute:

```
$ maze-run env=gym_env env.name=CartPole-v0
```

This runs a rollout of a random policy on `cartpole` environment. Statistics from the rollout are printed to the console, and trajectory data with event logs are stored in the output directory automatically configured by Hydra.

Alternatively, we might configure the rollouts to run just one episode in sequential mode and render the env (but more on that and other configuration options below):

```
$ maze-run env=gym_env env.name=CartPole-v0 runner=sequential runner.n_episodes=1␣
↪runner.render=true
```

3.2.2 Rollout Runner Configuration

Rollouts are run by rollout *runners*, which are agent- and environment-agnostic (for configuring environments and agents, see the following section).

By default, rollouts are run in multiple processes in parallel (as can be seen in the rollout configuration file, which lists `runner:  parallel` in the defaults), and are handled by the ParallelRolloutRunner.

Alternatively, rollouts can be run sequentially in a single process by opting for the `sequential` runner configuration:

```
$ maze-run env=gym_env env.name=CartPole-v0 runner=sequential
```

This is mainly useful when running a single episode only or for debugging, as sequential rollouts are much slower.

The available configuration options for both scenarios are listed in the Hydra runner package (`conf/runners/`).

These are the parameters for `parallel` rollout runner:

```
# @package _group_
type: maze.core.rollout.parallel_rollout_runner.ParallelRolloutRunner

# Number of processes to run the rollouts in concurrently
n_processes: 5

# Total number of episodes to run
n_episodes: 50

# Max steps per episode to perform
max_episode_steps: 200

# If true, trajectory data will be recorded and stored in :code:`trajectory_data`␣
↪directory
record_trajectory: true

# If true, event logs will be recorded and stored in `event_logs_directory
record_event_logs: true

# (Note that the default output directory is handled by Hydra)
```

Using these parameters, we can modify the rollout to e.g. be run only in 3 processes, and be comprised of 100 episodes, each of max 50 steps:

```
$ maze-run env=gym_env env.name=CartPole-v0 runner.n_processes=3 \
  runner.n_episodes=100 runner.max_episode_steps=10
```

(Alternatively, you can create your own configuration file that you will then supply to the `maze-run` command as described in Hydra primer section).

3.2.3 Environment and Policy Configuration

Environment and policy are configured using the `env`, resp. `policy` Hydra packages. Rollout runners are environment- and agent-agnostic, and will attempt to instantiate the type specified in the config files using Maze Registry.

Environment is expected to conform to the `StructuredEnv` interface and agent to the `StructuredPolicy` interface.

For agents, there are the following example config files:

- `agent/random_policy.yaml` for instantiating a class that conforms to the `StructuredPolicy` interface directly
- `agent/cutting_2d_greedy_policy` (in `maze-envs/logistics`) for wrapping (potentially multiple) flat policies into a structured policy
- `agent/torch_policy` (in `maze/train`) for loading and rolling out a policy trained using the Maze framework

Hence, after training a policy on the *tutorial Cutting 2D environment*:

```
$ maze-run -cn conf_train env=tutorial_cutting_2d_struct_masked
  wrappers=tutorial_cutting_2d model=tutorial_cutting_2d_struct_masked
```

We can roll it out using:

```
$ maze-run policy=torch_policy env=tutorial_cutting_2d_struct_masked␣
↪wrappers=tutorial_cutting_2d \
  model=tutorial_cutting_2d_struct_masked input_dir=outputs/[training-output-dir]
```

Note that for this to work, the `training-output-dir` parameter must be set to the output directory of the training run (the model state dict and other configuration will be loaded from there).

3.2.4 Plain Python Configuration

Rollout runners are primarily designed to support running through Hydra from command line. That being said, you can of course instantiate and use the runners directly in Python if you have some special needs.

```
from maze.core.agent.dummy_cartpole_policy import DummyCartPolePolicy
from maze.core.rollout.sequential_rollout_runner import SequentialRolloutRunner
from maze.core.wrappers.maze_gym_env_wrapper import GymMazeEnv

# Instantiate an example environment and agent
env = GymMazeEnv("CartPole-v0")
agent = DummyCartPolePolicy()

# Run a sequential rollout with rendering
# (including an example wrapper the environment will be wrapped in)
sequential = SequentialRolloutRunner(
```

(continues on next page)

(continued from previous page)

```
    n_episodes=10,
    max_episode_steps=100,
    record_trajectory=True,
    record_event_logs=True,
    render=True)
sequential.run_with(env=env, wrappers={"ObservationLoggingWrapper": {}}, agent=agent)
```

Using the snippet above, you can run a rollout on any agent and environment directly from Python (parallel rollouts can be run similarly).

However, note that the rollout runners are currently designed to be run only once (which is their main use case for runs initiated from the command line). Running them repeatedly might cause some issues especially with statistics and event logging, as the runners initiate new writers every time (so you might get duplicate outputs) and some of these operations are order-sensitive (especially for the parallel rollouts where some state might be carried over to child processes).

3.2.5 Where to Go Next

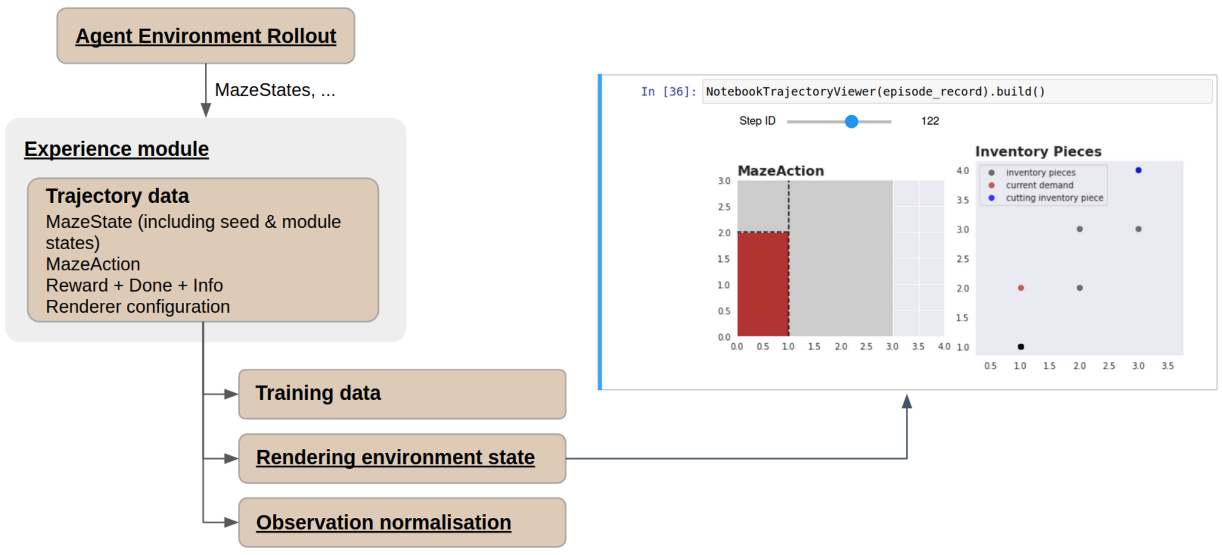
If you collected trajectory data during the rollout, you might want to:

- Visualize the collected rollout data in a *trajectory viewer notebook*
- Use the collected data for *imitation learning*

3.3 Collecting and Visualizing Rollouts

While the *Event System* provides an overview of notable events happening during an episode through *statistics and event logs*, it is often needed to dig deeper and visualize the full environment state at a given time step.

With the Maze Trajectory Viewer, it is possible replay past episodes from collected trajectory data in a Jupyter Notebook.



```
In [36]: NotebookTrajectoryViewer(episode_record).build()
```

3.3.1 Requirements

Note: Rollouts visualization in a notebook is not currently available for Gym environments.

The trajectory viewer notebook requires the environment to implement a Maze-compatible `Renderer` based on matplotlib. The tutorial 2D cutting environment serves as a perfect example – see the *Adding a Renderer* section to understand how to implement one.

Unfortunately, Maze does not yet support rendering from trajectory data for standard Gym environments. For such environments, you can render only during the rollout itself by setting the corresponding option on the sequential renderer (i.e., provide the following overrides for rollouts: `runner=sequential runner.render=true`).

3.3.2 Trajectory Data Collection

When using a compliant environment, past trajectories can be rendered directly from the trajectory data. These are usually collected using the rollout runners via CLI.

To simply collect trajectory data of a heuristic policy on the *tutorial Cutting 2D environment*, run:

```
$ maze-run env=tutorial_cutting_2d_flat policy=tutorial_cutting_2d_greedy_policy
```

Alternatively (and closer to a real training setting), you might want to first *train* an RL policy on the tutorial 2D cutting environment:

```
$ maze-run -cn conf_train env=tutorial_cutting_2d_struct_masked
  wrappers=tutorial_cutting_2d model=tutorial_cutting_2d_struct_masked
```

and then *roll it out* to collect the trajectory data (make sure to substitute the `input_dir` value for your actual training output directory):

```
$ maze-run policy=torch_policy env=tutorial_cutting_2d_struct_masked␣
↪wrappers=tutorial_cutting_2d \
  model=tutorial_cutting_2d_struct_masked input_dir=outputs/[training-output-dir]
```

Once the rollout has run, take note of the outputs directory created by Hydra, where the trajectory data will be logged – by default inside the `trajectory_data` subdirectory, one pickle file per episode (identified by a UUID generated for each episode).

(Whether trajectory data is recorded during a rollout is set using the `runner.record_trajectory` flag, which is on by default.)

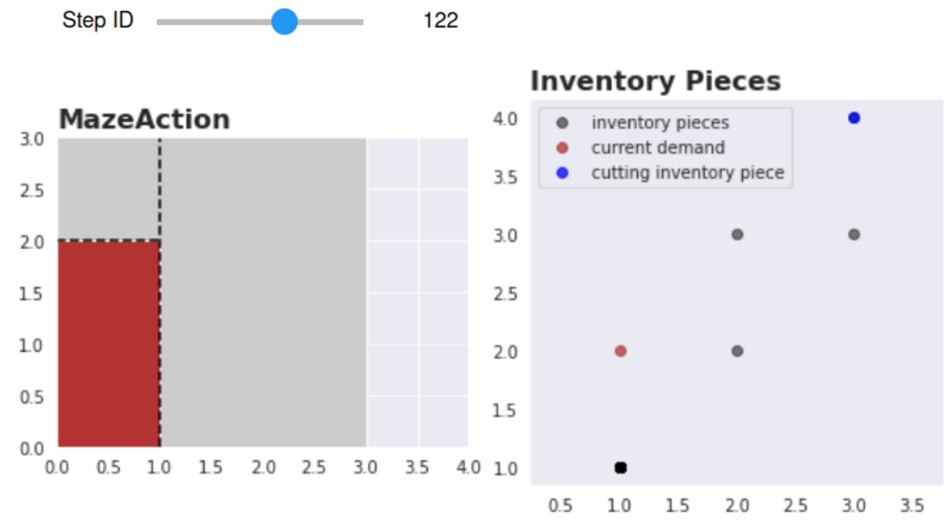
3.3.3 Trajectory Visualization

Maze includes a Jupyter Notebook in `evaluation/viewer.ipynb` that will guide you through the process. You only need to supply a path to the outputs directory where you trajectory data reside. The renderer will be automatically built from the trajectory data.

(Note that the notebook also lists example trajectory data in case you do not have any on hand.)

Once an episode is selected and loaded, it is possible to skim back and forward in time using the notebook widgets slider (controllable by mouse or keyboard).

```
In [36]: NotebookTrajectoryViewer(episode_record).build()
```

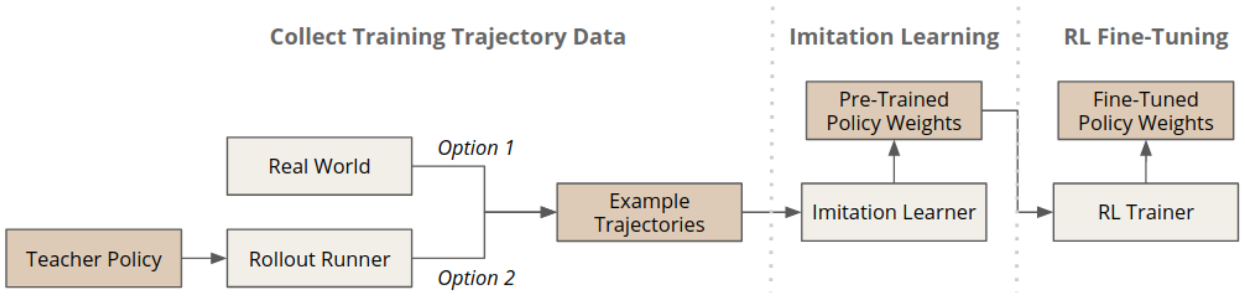


3.3.4 Where to Go Next

- To understand in more detail how to train a policy and then roll it out to collect trajectory data, check out *Trainings* and *Rollouts*.
- Rendering and reviewing each time step in detail comes with a lot of overhead. In case you just want to see and easily compare notable events that happened across different episodes, you might want to review the *Event system* and how it is used to log *statistics*, *KPIs*, and *raw events*.

3.4 Imitation Learning and Fine-Tuning

Imitation learning refers to the task of learning a policy by imitating the behaviour of an existing teacher policy usually represented as a fixed set of example trajectories. In some scenarios we might even have direct access to the actual teacher policy itself allowing us to generate as many training trajectories as required. Imitation learning is especially useful for initializing a policy to quick-start an actual training by interaction run or for settings where no training environment is available at all (e.g., offline RL).



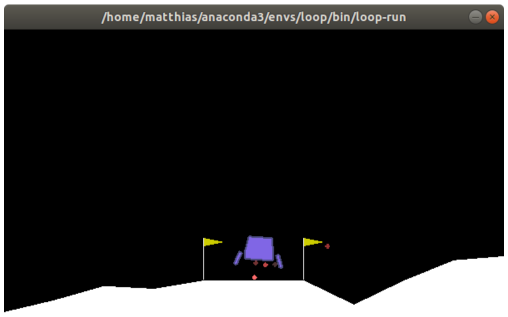
Overview:

- *Collect Training Trajectory Data*
- *Learn from Example Trajectories*
- *Fine-Tune a Pre-Trained Policy*
- *Where to Go Next*

3.4.1 Collect Training Trajectory Data

This section explains how to rollout a policy for collecting example trajectories. As the training trajectories might be already available (e.g., collected in practice) this step is optional.

As an example environment we pick the discrete version of the LunarLander environment as it already provides a heuristic policy which we can use to collect or training trajectories for imitation learning.



But first let's check if the policy actually does something meaningful by running a few rendering rollouts:

```
maze-run env.name=LunarLander-v2 policy=lunar_lander_heuristics \
runner=sequential runner.render=true runner.n_episodes=3
```

Hopefully this looks good and we can continue with actually collecting example trajectories for imitation learning.

The command bellow performs 3 rollouts of the heuristic policy and records them to the output directory.

```
maze-run env.name=LunarLander-v2 policy=lunar_lander_heuristics runner.n_episodes=3
```

You will get the following output summarizing the statistics of the rollouts.

```
 step|path                                                                    |         ␣
↪  value
=====|=======================================================================|================
    1|rollout_stats    DiscreteActionEvents  action|  substep_0/action      |[len:583,
↪ :1.2]
    1|rollout_stats    BaseEnvEvents         reward|  median_step_count     |         ␣
↪200.000
    1|rollout_stats    BaseEnvEvents         reward|  mean_step_count       |         ␣
↪194.333
    1|rollout_stats    BaseEnvEvents         reward|  total_step_count      |         ␣
↪583.000
    1|rollout_stats    BaseEnvEvents         reward|  total_episode_count   |         ␣
↪  3.000
    1|rollout_stats    BaseEnvEvents         reward|  episode_count         |         ␣
↪  3.000
    1|rollout_stats    BaseEnvEvents         reward|  std                   |         ␣
↪ 51.350
```

(continues on next page)

(continued from previous page)

```
   1|rollout_stats    BaseEnvEvents         reward|  mean                      |        ␣
↪190.116
   1|rollout_stats    BaseEnvEvents         reward|  min                       |        ␣
↪121.352
   1|rollout_stats    BaseEnvEvents         reward|  max                       |        ␣
↪244.720
```

The trajectories will be dumped similar to the file structure shown below.

```
- outputs/<experiment_path>
    - maze_cli.log
    - event_logs
    - trajectory_data
        - 00653455-d7e2-4737-a82b-d6d1bfce12f7.pkl
        - ...
```

The pickle files contain the distinct episodes recorded as a sequences of `StepRecord` objects each keeping the trajectory data for one step (state, action, reward, . . .).

3.4.2 Learn from Example Trajectories

Given the trajectories recorded in the previous step we now train a policy with *behavioral cloning*, a simple version of imitation learning.

To do so we simply provide the trajectory data as an argument and run:

```
maze-run -cn conf_train env.name=LunarLander-v2 model=vector_obs wrappers=vector_obs \
algorithm=bc runner.validation_percentage=50 \
runner.dataset.trajectory_data_dir=<absolute_experiment_path>/trajectory_data
```

```
...
********** Epoch 24: Iteration 1500 **********
 step|path                                                                   |   ␣
↪value
=====|=======================================================================|=========
   96|train     ImitationEvents       discrete_accuracy     0/action         |    0.
↪948
   96|train     ImitationEvents       policy_loss           0                |    0.
↪150
   96|train     ImitationEvents       policy_entropy        0                |    0.
↪209
   96|train     ImitationEvents       policy_l2_norm        0                |   42.
↪416
   96|train     ImitationEvents       policy_grad_norm      0                |    0.
↪870
 step|path                                                                   |   ␣
↪value
=====|=======================================================================|=========
   96|eval      ImitationEvents       discrete_accuracy     0/action         |    0.
↪947
   96|eval      ImitationEvents       policy_loss           0                |    0.
↪152
   96|eval      ImitationEvents       policy_entropy        0                |    0.
↪207
-> new overall best model -0.15179!
```

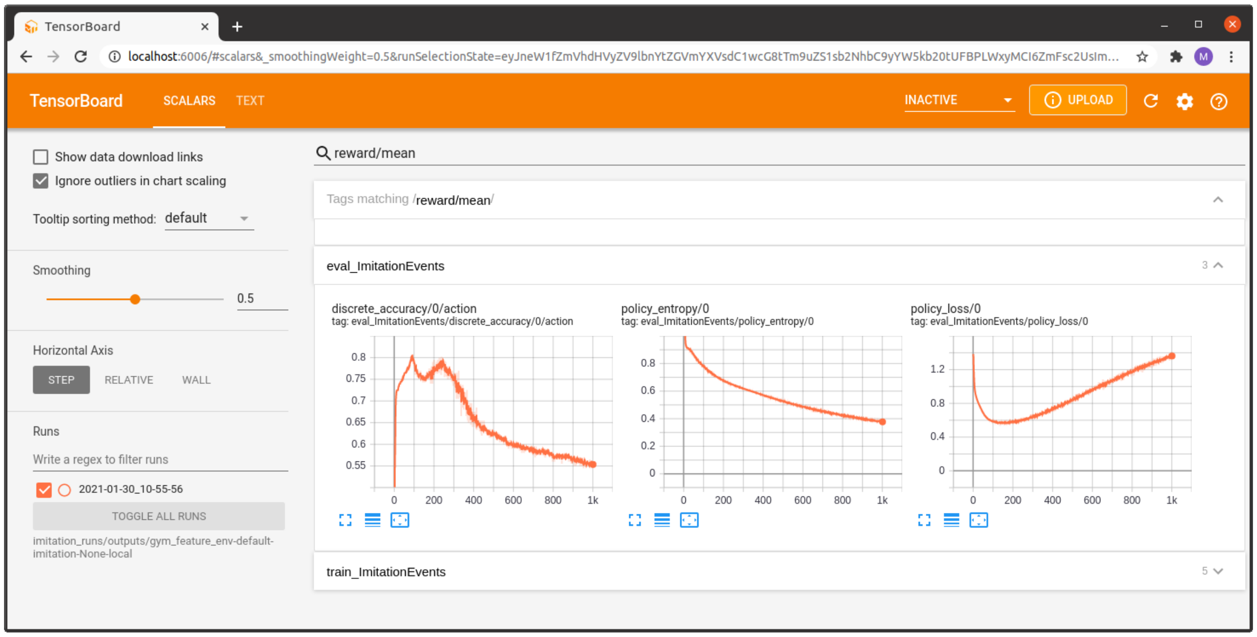
(continues on next page)

(continued from previous page)

```
...
```

As with all trainers, we can watch the training progress with Tensorboard.

```
tensorboard --logdir outputs/
```



localhost:6006/#scalars&_smoothingWeight=0.5&runSelectionState=eyJneW1fZmVhdHVyZV9lbnYtZGVmYXVsdC1wcG8tTm9uZS1sb2NhbC9yYW5kb20tUFBPLWxyMC6ZmFsc2UsIm...

☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.5
Horizontal Axis: STEP
☑ 2021-01-30_10-55-56
imitation_runs/outputs/gym_feature_env-default-imitation-None-local

TensorBoard
SCALARS
TEXT
INACTIVE
UPLOAD
reward/mean
Tags matching /reward/mean/
RELATIVE
WALL
Runs
Write a regex to filter runs
TOGGLE ALL RUNS
eval_ImitationEvents
discrete_accuracy/0/action
tag: eval_ImitationEvents/discrete_accuracy/0/action
policy_entropy/0
tag: eval_ImitationEvents/policy_entropy/0
policy_loss/0
tag: eval_ImitationEvents/policy_loss/0
train_ImitationEvents

Once training is complete we can check how the behaviourally cloned policy performs in action.

```
maze-run env.name=LunarLander-v2 model=vector_obs wrappers=vector_obs \
policy=torch_policy input_dir=outputs/<imitation-learning-experiment>
```

```
 step|path                                                            |         ␣
↪ value
=====|================================================================|==================
    1|rollout_stats    DiscreteActionEvents  action    substep_0/action    |[len:8033,
↪ :1.2]
    1|rollout_stats    BaseEnvEvents         reward    median_step_count   |         ␣
↪186.000
    1|rollout_stats    BaseEnvEvents         reward    mean_step_count     |         ␣
↪160.660
    1|rollout_stats    BaseEnvEvents         reward    total_step_count    |        ␣
↪8033.000
    1|rollout_stats    BaseEnvEvents         reward    total_episode_count |         ␣
↪ 50.000
    1|rollout_stats    BaseEnvEvents         reward    episode_count       |         ␣
↪ 50.000
    1|rollout_stats    BaseEnvEvents         reward    std                 |         ␣
↪111.266
    1|rollout_stats    BaseEnvEvents         reward    mean                |         ␣
↪101.243
    1|rollout_stats    BaseEnvEvents         reward    min                 |         -
↪164.563
    1|rollout_stats    BaseEnvEvents         reward    max                 |         ␣
↪282.895
```

With a mean reward of 101 this already looks like a promising starting point for RL fine-tuning.

3.4.3 Fine-Tune a Pre-Trained Policy

In the last section we show how to fine-tune the pre-trained policy with a model-free RL learner such as *PPO*. It is basically a standard PPO training run initialized with the imitation learning output.

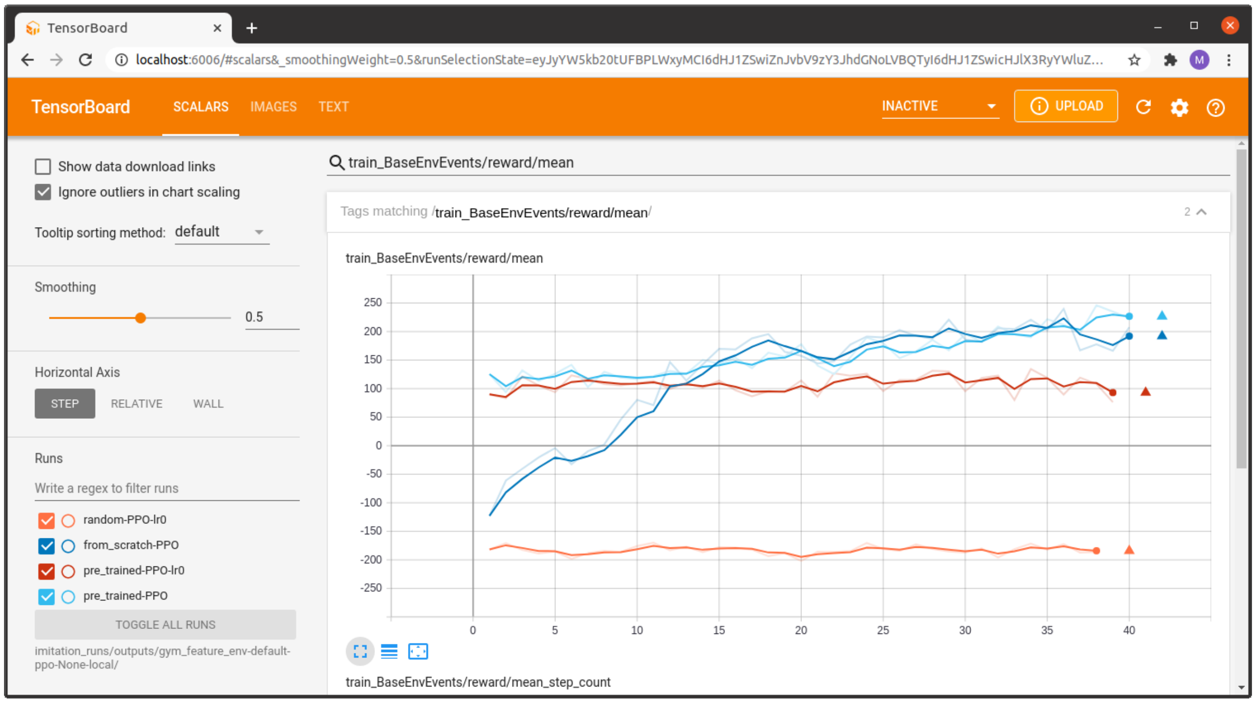
```
maze-run -cn conf_train env.name=LunarLander-v2 model=vector_obs critic=default_state␣
↪wrappers=vector_obs \
algorithm=ppo runner.eval_repeats=100 runner.critic_burn_in_epochs=10 \
input_dir=outputs/<imitation-learning-experiment>
```

Once training started we can observe the progress with Tensorboard (for the sake of clarity of this example we renamed the experiment directories for the screenshot below).

The Tensorboard log below compares the following experiments:

- a randomly initialized policy trained with learning rate 0.0 (random-PPO-lr0)
- a behavioural cloning pre-trained policy trained with learning rate 0.0 (pre_trained-PPO-lr0)
- a randomly initialized policy trained with PPO (from_scratch-PPO)
- a behavioural cloning pre-trained policy trained with PPO (pre_trained-PPO)

We also included training runs with a learning rate of 0.0 to get a feeling for the performance of the initial performance of the two models (randomly initialized vs. pre-trained).


localhost:6006/#scalars&_smoothingWeight=0.5&runSelectionState=eyJyYW5kb20tUFBPLWxyMCI6dHJ1ZSwiZnJvbV9zY3JhdGNoLVBQTyI6dHJ1ZSwicHJlX3RyYWluZ...

Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.5
Horizontal Axis: STEP
Runs
random-PPO-lr0
from_scratch-PPO
pre_trained-PPO-lr0
pre_trained-PPO
imitation_runs/outputs/gym_feature_env-default-ppo-None-local/

train_BaseEnvEvents/reward/mean
Tags matching /train_BaseEnvEvents/reward/mean/
train_BaseEnvEvents/reward/mean
train_BaseEnvEvents/reward/mean_step_count

As expected, we see that PPO fine-tuning of the pretrained model starts at an initially much higher reward level compared to the model trained entirely from scratch.

Although this is a quite simple example it is still a nice showcase for the usefulness of this two-stage learning paradigm. For scenarios with delayed and/or sparse rewards following this principle is often crucial to get the RL trainer to start learning at all.

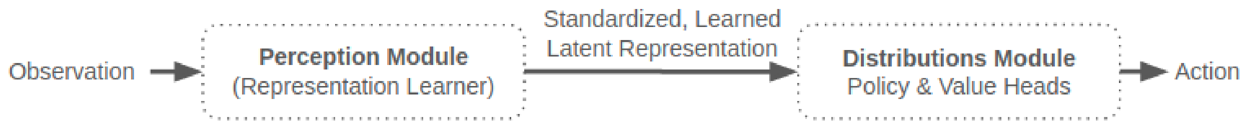
3.4.4 Where to Go Next

- You can find more details on *training* and *rollouts* on the dedicated pages.
- You can also read up on how to *visualize recorded rollouts*.
- For further details on the learning algorithms you can visit the *Trainers* page.

3.5 Introducing the Perception Module

One of the key ingredients for successfully training RL agents in complex environments is their combination with powerful representation learners; in our case PyTorch-based neural networks. These enable the agent to perceive all kinds of observations (e.g. images, audio waves, sensor data, . . .), unlocking the full potential of the underlying RL-based learning systems.

Maze supports neural network building blocks via the *Perception Module*, which is responsible for transforming raw observations into standardized, learned latent representations. These representations are then utilized by the *Action Spaces and Distributions Module* to yield policy as well as critic outputs.



This page provides a general introduction into the Perception Module (which we recommend to read, of course). However, you can also start using the module right away and jump to the *template* or *custom models* section.

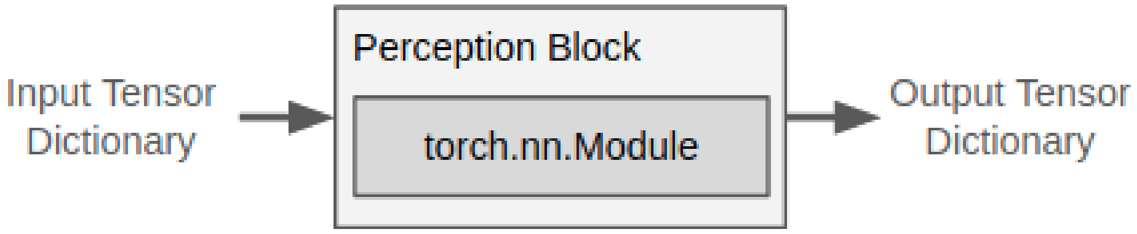
3.5.1 List of Features

Below we list the key features and design choices of the perception module:

- Based on PyTorch.
- Supports dictionary observation spaces.
- Provides a large variety of *neural network building blocks* and model styles for customizing policy and value networks:
  - feed forward: dense, convolution, graph convolution and attention, . . .
  - recurrent: LSTM, last-step-LSTM, . . .
  - general purpose: action and observation masking, self-attention, concatenation, slicing, . . .
- Provides shape inference allowing to derive custom models directly from observation space definitions.
- Allows for environment specific customization of existing network templates per yaml configuration.
- Definition of complex networks explicitly in Python using Maze perception blocks and/or PyTorch.
- Generates detailed visualizations of policy and value networks (model graphs) containing the perception building blocks as well as all intermediate representation produced.
- Can be easily extended with custom network components if necessary.

3.5.2 Perception Blocks

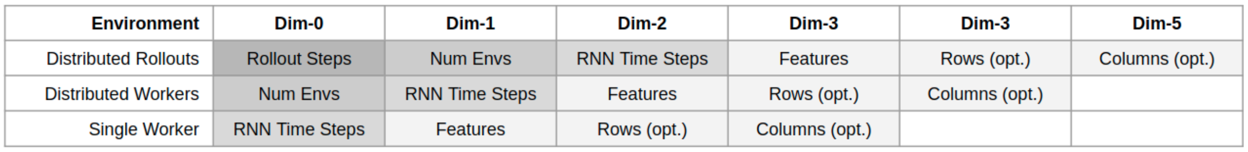
Perception blocks are components for composing models such as policy and value networks within Maze. They implement PyTorch's nn.Module interface and encapsulate neural network functionality into distinct, reusable units. In order to handle all our requirements (listed in the motivation below), every perception block expects a tensor dictionary as input and produce a tensor dictionary again as an output.



Maze already supports a number of *built-in neural network building blocks* which are, like all other components, *easily extendable*.

Motivation: Maze introduces perception blocks to extend PyTorch's nn.Module with **shape inference** to support the following features:

1. To derive, generate and customize *template models* directly from observation and action space definitions.
2. To *visualize models* and how these process observations to ultimately arrive at an action or value prediction.
3. To seamlessly apply models at different stages of the RL development processes without the need for extensive input reshaping regardless if we perform a distributed training using parallel rollout workers or if we deploy a single agent in production. The figure below shows a few examples of such scenarios.

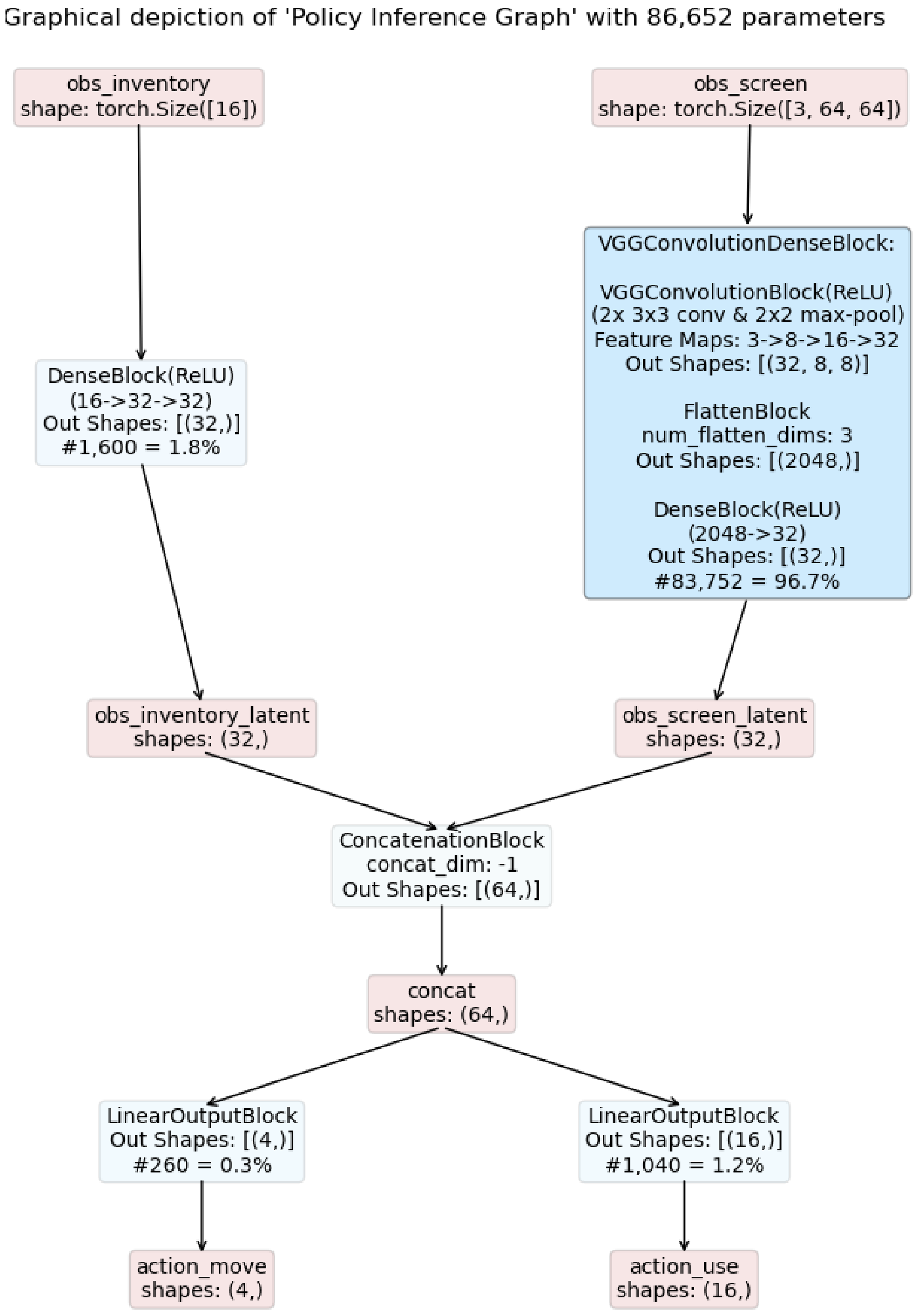


Environment	Dim-0	Dim-1	Dim-2	Dim-3	Dim-3	Dim-5
Distributed Rollouts	Rollout Steps	Num Envs	RNN Time Steps	Features	Rows (opt.)	Columns (opt.)
Distributed Workers	Num Envs	RNN Time Steps	Features	Rows (opt.)	Columns (opt.)	
Single Worker	RNN Time Steps	Features	Rows (opt.)	Columns (opt.)		

3.5.3 Inference Blocks

The InferenceBlock, a special perception block, combines multiple perception blocks into one prediction module. This is convenient and allows us to easily reuse semantically connected parts of our models but also enables us to derive and visualize inference graphs of these models. This is feasible as perception blocks operate with input and output tensor dictionaries, which can be easily linked to an inference graph.

The figure below shows a simple example of how such a graph can look like.



Details:

- The model depicted in the figure above takes two observations as inputs:

  - *obs_inventory* : a 16-dimensional feature vector
  - *obs_screen* : a 64 x 64 RGB image
- *obs_inventory* is processed by a DenseBlock resulting in a 32-dimensional latent representation.
- *obs_screen* is processed by a VGG-style model resulting in a 32-dimensional latent representation.
- Next, these two representations are concatenated into a joint representation with dimension 64.
- Finally we have two LinearOutputBlocks yielding the logits for two distinct action heads:
  - *action_move* : a *categorical action* deciding to move [*UP, DOWN, LEFT, RIGHT*],
  - *action_use* : a *multi-binary action* deciding which item to use from inventory.

Comments on visualization: Blue boxes are blocks, while red ones are tensors. The color depth of blocks (blue) indicates the number of the parameters relative to the total number of parameters.

3.5.4 Model Composers

Model Composers, as the name suggest, compose the models and as such bring all components of the perception module together under one roof. In particular, they hold:

- Definitions of observation and actions spaces.
- All defined models, that is, policies (multiple ones in multi-step scenarios) and critics (multiple ones in multi-step scenarios depending on the critic type).
- The *Distribution Mapper*, mapping (possible custom) probability distributions to action spaces.

Maze supports *different types of model composers* and we will show how to work with *template* and *custom models* in detail later on.

3.5.5 Implementing Custom Perception Blocks

In case you would like to implement and use custom components when designing your models you can add new blocks by implementing:

- The PerceptionBlock interface common for all perception blocks.
- The ShapeNormalizationBlock interface normalizing the input and de-normalizing the output tensor dimensions if required for your block (optional).
- The respective forward pass of your block.

The code-snippet below shows a simple toy-example block, wrapping a linear layer into a Maze perception block.

```python
"""Contains a single linear layer block."""
import builtins
from typing import Union, List, Sequence, Dict

import torch
from torch import nn as nn

from maze.core.annotations import override
from maze.perception.blocks.shape_normalization import ShapeNormalizationBlock

Number = Union[builtins.int, builtins.float, builtins.bool]
```

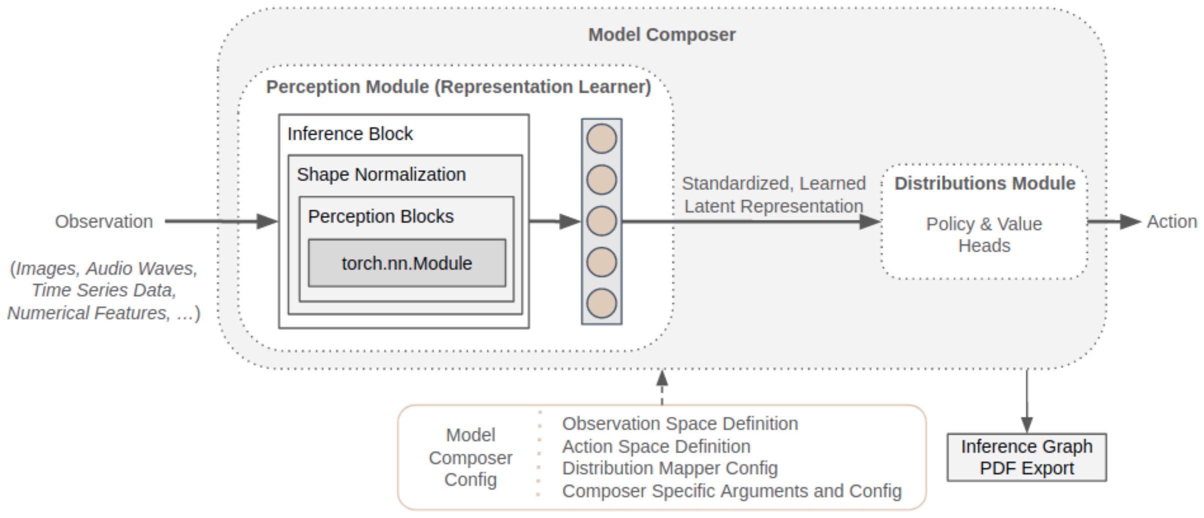
(continues on next page)

(continued from previous page)

```
class MyLinearBlock(ShapeNormalizationBlock):
    """A linear output block holding a single linear layer.

    :param in_keys: One key identifying the input tensors.
    :param out_keys: One key identifying the output tensors.
    :param in_shapes: List of input shapes.
    :param output_units: Count of output units.
    """

    def __init__(self,
                 in_keys: Union[str, List[str]],
                 out_keys: Union[str, List[str]],
                 in_shapes: Union[Sequence[int], List[Sequence[int]]],
                 output_units: int):
        super().__init__(in_keys=in_keys, out_keys=out_keys, in_shapes=in_shapes, in_
↪num_dims=2, out_num_dims=2)

        self.input_units = self.in_shapes[0][-1]
        self.output_units = output_units

        # initialize the linear layer
        self.net = nn.Linear(self.input_units, self.output_units)

    @override(ShapeNormalizationBlock)
    def normalized_forward(self, block_input: Dict[str, torch.Tensor]) -> Dict[str,␣
↪torch.Tensor]:
        """implementation of :class:`~maze.perception.blocks.shape_normalization.
↪ShapeNormalizationBlock` interface
        """
        # extract the input tensor of the first (and here only) input key
        input_tensor = block_input[self.in_keys[0]]
        # apply the linear layer
        output_tensor = self.net(input_tensor)
        # return the output tensor as a tensor dictionary
        return {self.out_keys[0]: output_tensor}

    def __repr__(self):
        """This is the text shown in the graph visualization."""
        txt = self.__class__.__name__
        txt += f"\nOut Shapes: {self.out_shapes()}"
        return txt
```

3.5.6 The Bigger Picture

The figure below shows how the components introduced in the perception module relate to each other.


Model Composer
Perception Module (Representation Learner)
Inference Block
Shape Normalization
Perception Blocks
torch.nn.Module
Observation
(Images, Audio Waves, Time Series Data, Numerical Features, ...)
Standardized, Learned Latent Representation
Distributions Module
Policy & Value Heads
Action
Model Composer Config
Observation Space Definition
Action Space Definition
Distribution Mapper Config
Composer Specific Arguments and Config
Inference Graph PDF Export

3.5.7 Where to Go Next

- For further details please see the *reference documentation*.
- Action Spaces and Distributions
- Working with template models
- Working with custom models
- Pre-processing and observation normalization

3.6 Action Spaces and Distributions

In response to the states perceived and rewards received, RL agents interact with their environment by taking appropriate actions. Depending on the problem at hand there are *different types of actions* an agent must be able to deal with (e.g. categorical, binary, continuous, . . .).

To support this requirement Maze introduces the Distribution Module which builds on top of the *Perception Module* allowing to fully customize which probability distributions to link with certain action spaces or even individual action heads.

3.6.1 List of Features

The distribution module provides the following key features:

- Supports flat dictionary action spaces (nested dict spaces are not yet supported)
- Supports a variety of different *action spaces and probability distributions*
- Supports customization of which probability distribution to use for which action space or head
- Supports action masking in combination with the perception module
- Allows to add and register new probability distributions whenever required

3.6.2 Action Spaces and Probability Distributions

Maze so far supports the following action space - probability distribution combinations.

Action Space	Available Distributions
Discrete	Categorical (default)
Multi-Discrete	Multi-Categorical (default)
Multi-binary	Bernoulli (default)
Box (Continuous)	Diagonal-Gaussian (default), Beta, Squashed-Gaussian
Dict	DictProbabilityDistribution (default)

The DictProbabilityDistribution combines any of the other action spaces and distributions into a joint action space in case you agent has to interact with the environment via different action space types at the same time.

Note that the table above does not always follow a one-to-one mapping. In case of a Box (Continuous) action space for example you can choose between a Diagonal-Gaussian distribution in case of an unbounded action space or a Beta or a Squashed-Gaussian distribution in case of a bounded action space. In other cases you might even want to *add additional probability distributions* according to the nature of the environment you are facing.

To allow for easy customization of the links between action spaces and distributions Maze introduces the **DistributionMapper** for which we show usage examples below.

3.6.3 Example 1: Mapping Action Spaces to Distributions

Adding the snippet below to your model config specifies the following:

- Use Beta distributions for all Box action spaces.
- All other action spaces behave as specified in the *defaults*.

```
# @package model
distribution_mapper_config:
  - action_space: gym.spaces.Box
    distribution: maze.distributions.beta.BetaProbabilityDistribution
```

3.6.4 Example 2: Mapping Actions to Distributions

Adding the snippet below to your model config specifies the following:

- Use Beta distributions for all Box action spaces.
- Use a Squashed-Gaussian distributions for the action with key “special_action”.
- All other action spaces behave as specified in the *defaults*.

```
# @package model
distribution_mapper_config:
  - action_space: gym.spaces.Box
    distribution: maze.distributions.beta.BetaProbabilityDistribution
  - action_head: special_action
    distribution: maze.distributions.squashed_gaussian.
↪SquashedGaussianProbabilityDistribution
```

When specifying custom behaviour for distinct action heads make sure to add them below the more general action space configurations (e.g. get more specific from top to bottom).

3.6.5 Example 3: Using Custom Distributions

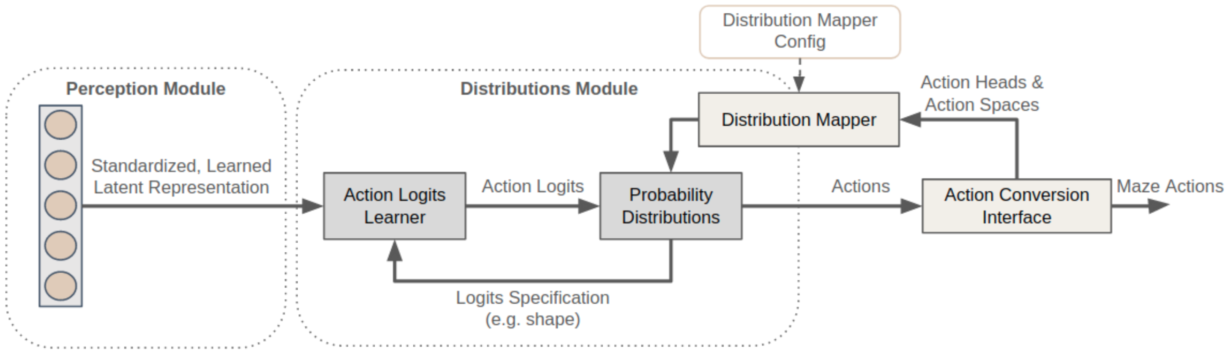
In case the probability distributions contained in Maze are not sufficient for your use case you can of course add additional custom probability distributions.

```
# @package model
distribution_mapper_config:
  - action_space: gym.spaces.Discrete
    distribution: my_package.maze_extentions.distributions.
↪CustomCategoricalProbabilityDistribution
```

The example above defines to use a **CustomCategoricalProbabilityDistribution** for all discrete action spaces. When adding a new distribution you (1) have to implement the **ProbabilityDistribution** interface and (2) make sure that it is accessible within your python path. Besides that you only have to provide the reference path of the probability distribution you would like to use.

3.6.6 The Bigger Picture

The figure below relates the distribution module with the overall workflow.


Distribution Mapper Config
Perception Module
Distributions Module
Distribution Mapper
Action Heads & Action Spaces
Standardized, Learned Latent Representation
Action Logits Learner
Action Logits
Probability Distributions
Actions
Action Conversion Interface
Maze Actions
Logits Specification (e.g. shape)

The distribution mapper takes the (dictionary) action space as an input and links the action spaces with the respective probability distributions specified in the config. Action logits are learned on top of the representation produced by the perception module where each probability distribution specifies its expected logits shape.

3.6.7 Where to Go Next

- For further details please see the *reference documentation*.
- Processing raw observations with the Maze Perception Module.
- Customizing models with Hydra.

3.7 Working with Template Models

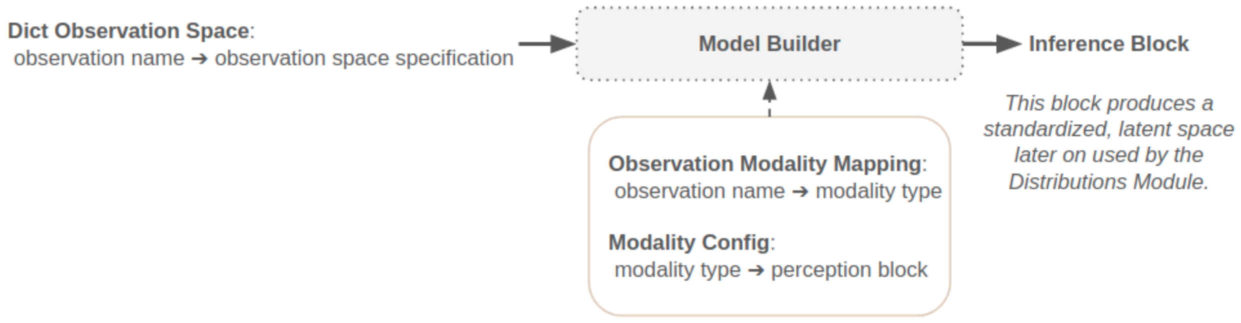
The Maze template *model composer* allows us to compose policy and value networks directly from an environment's observation and action space specification according to a selected model template and a corresponding model config. The central part of a template model composer is the *Model Builder* holding an *Inference Block* template (architecture template), which is then instantiated according to the config.

Next, we will introduce the general working principles. However, you can of course directly jump to the examples below to see how to build a *feed forward* as well as a *recurrent policy network* using the ConcatModelBuilder or check out how to work with simple *single observation and action environments*.

3.7.1 List of Features

A template model supports the following features:

- Works with dictionary observation spaces.
- Maps individual observations to modalities via the **Observation Modality Mapping**.
- Allows to individually assign *Perception Blocks* to modalities via the **Modality Config**.
- Allows to pick *architecture templates* defining the underlying modal structure via **Maze Model Builders**.
- Cooperates with the *Distributions Module* supporting customization of action and value outputs.


Dict Observation Space:
observation name ➔ observation space specification
Model Builder
Inference Block
This block produces a standardized, latent space later on used by the Distributions Module.
Observation Modality Mapping:
observation name ➔ modality type
Modality Config:
modality type ➔ perception block

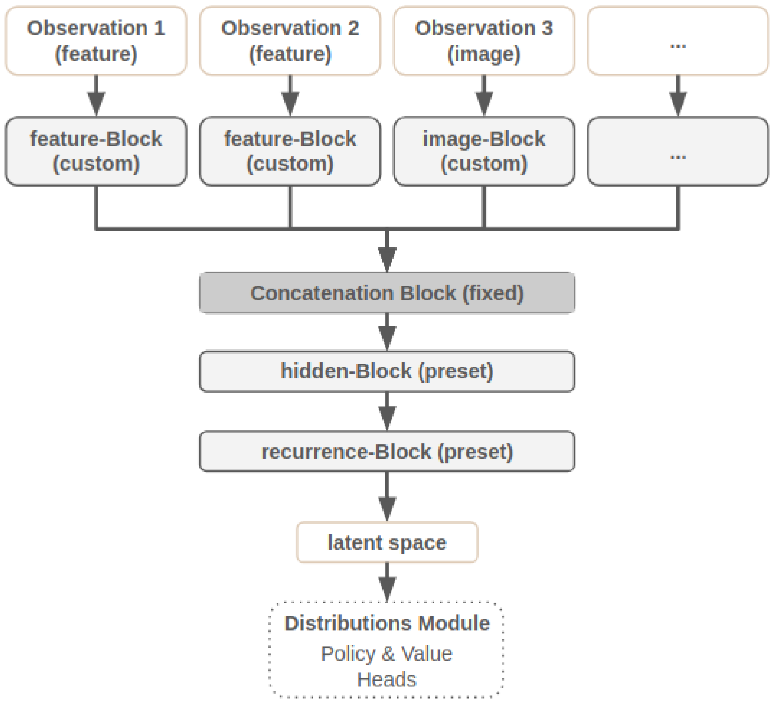
Note: Maze so far does not support "end-to-end" default behaviour but instead provides config templates, which can be adopted to the respective needs. We opted for this route as complete defaults might lead to unintended and non-transparent results.

3.7.2 Model Builders (Architecture Templates)

This section lists and describes the available Model Builder architectures templates. Before we describe the builder instances in more detail we provide some information on the available block types:

- **Fixed**: these blocks are fixed and are applied by the model builder per default.
- **Preset**: these blocks are predefined for the respective model builder. They are basically place holders for which you can specify the perception blocks they should hold.
- **Custom**: these blocks are introduced by the user for processing the respective observation modalities (types) such as features or images.

ConcatModelBuilder (*Reference Documentation*)



Model builder details:

- Processes the individual observations with **modality blocks** (custom).
- Joins the respective modality hidden representations via a **concatenation block** (fixed).
- The resulting representation is then further processed by the **hidden** and **recurrence block** (preset).

3.7.3 Example 1: Feed Forward Models

In this example we utilize the *ConcatModelBuilder* to compose a feed forward actor-critic model processing two observations for predicting two actions and one critic (value) output.

Observation Space:

- *observation_inventory* : a 16-dimensional feature vector
- *observation_screen* : a 64 x 64 RGB image

Action Space:

- *action_move* : a *categorical action* with four options deciding to move [*UP, DOWN, LEFT, RIGHT*]
- *action_use* : a 16-dimensional *multi-binary action* deciding which item to use from inventory

The model config is defined as:

```
# @package model
type: maze.perception.models.default_model_composer.DefaultModelComposer

# specify distribution mapping
# (here we use a default distribution mapping)
distribution_mapper_config: {}

# specifies the architecture of default models
```

(continues on next page)

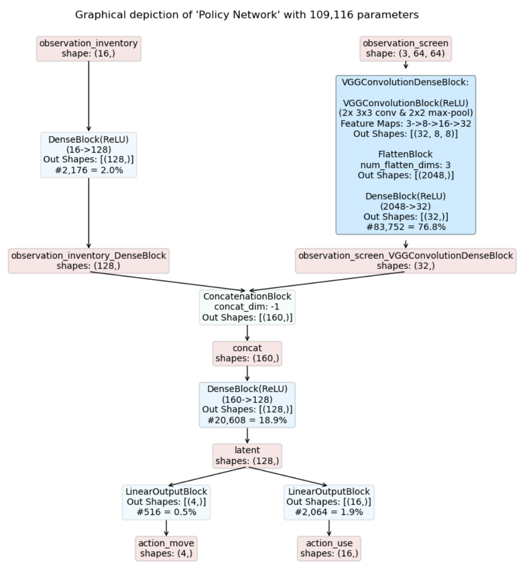
(continued from previous page)

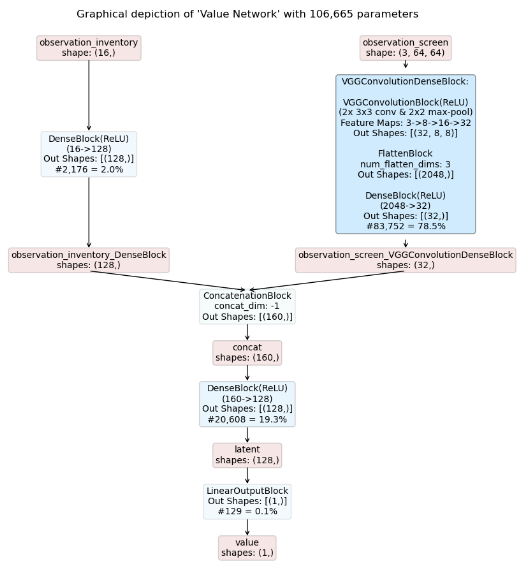
```yaml
model_builder:
  type: maze.perception.builders.ConcatModelBuilder

  # specifies the modality type of each observation
  observation_modality_mapping:
    observation_inventory: feature
    observation_screen: image

  # specifies with which block to process a modality
  modality_config:
    # modality processing
    feature:
      block_type: maze.perception.blocks.DenseBlock
      block_params:
        hidden_units: [32, 32]
        non_lin: torch.nn.ReLU
    image:
      block_type: maze.perception.blocks.VGGConvolutionDenseBlock
      block_params:
        hidden_channels: [8, 16, 32]
        hidden_units: [32]
        non_lin: torch.nn.ReLU
    # preserved keys for the model builder
    hidden:
      block_type: maze.perception.blocks.DenseBlock
      block_params:
        hidden_units: [128]
        non_lin: torch.nn.ReLU
    recurrence: {}

# select policy type
policy:
  type: maze.perception.models.policies.ProbabilisticPolicyComposer

# select critic type
critic:
  type: maze.perception.models.critics.StateCriticComposer
```

Details:

- Models are composed by the Maze *TemplateModelComposer*.
- No specific action space and probability distribution overrides are specified.
- The model is based on the ConcatModelBuilder architecture template.
- Observation *observation_inventory* is mapped to the user specified custom modality *feature*.
- Observation *observation_screen* is mapped to the user specified custom modality *image*.
- Modality Config:
  - Modalities of type *feature* are processed with a DenseBlock.
  - Modalities of type *image* are processed with a VGGConvolutionDenseBlock.
  - The concatenated joint spaces is processed with another DenseBlock.
  - No recurrence is employed.

The resulting inference graphs for an actor-critic model are shown below:


Graphical depiction of 'Policy Network' with 109,116 parameters
observation_inventory shape: (16,)
observation_screen shape: (3, 64, 64)
VGGConvolutionDenseBlock: VGGConvolutionBlock(ReLU) (2x 3x3 conv & 2x2 max-pool) Feature Maps: 3->8->16->32 Out Shapes: [(32, 8, 8)] FlattenBlock num_flatten_dims: 3 Out Shapes: [(2048,)] DenseBlock(ReLU) (2048->32) Out Shapes: [(32,)] #83,752 = 76.8%
DenseBlock(ReLU) (16->128) Out Shapes: [(128,)] #2,176 = 2.0%
observation_inventory_DenseBlock shapes: (128,)
observation_screen_VGGConvolutionDenseBlock shapes: (32,)
ConcatenationBlock concat_dim: -1 Out Shapes: [(160,)]
concat shapes: (160,)
DenseBlock(ReLU) (160->128) Out Shapes: [(128,)] #20,608 = 18.9%
latent shapes: (128,)
LinearOutputBlock Out Shapes: [(4,)] #516 = 0.5%
LinearOutputBlock Out Shapes: [(16,)] #2,064 = 1.9%
action_move shapes: (4,)
action_use shapes: (16,)


Graphical depiction of 'Value Network' with 106,665 parameters
observation_inventory shape: (16,)
observation_screen shape: (3, 64, 64)
VGGConvolutionDenseBlock: VGGConvolutionBlock(ReLU) (2x 3x3 conv & 2x2 max-pool) Feature Maps: 3->8->16->32 Out Shapes: [(32, 8, 8)] FlattenBlock num_flatten_dims: 3 Out Shapes: [(2048,)] DenseBlock(ReLU) (2048->32) Out Shapes: [(32,)] #83,752 = 78.5%
DenseBlock(ReLU) (16->128) Out Shapes: [(128,)] #2,176 = 2.0%
observation_inventory_DenseBlock shapes: (128,)
observation_screen_VGGConvolutionDenseBlock shapes: (32,)
ConcatenationBlock concat_dim: -1 Out Shapes: [(160,)]
concat shapes: (160,)
DenseBlock(ReLU) (160->128) Out Shapes: [(128,)] #20,608 = 19.3%
latent shapes: (128,)
LinearOutputBlock Out Shapes: [(1,)] #129 = 0.1%
value shapes: (1,)

3.7.4 Example 2: Recurrent Models

In this example we utilize the *ConcatModelBuilder* to compose a recurrent actor-critic model for the *the previous example*.

```
# @package model
type: maze.perception.models.default_model_composer.DefaultModelComposer

# specify distribution mapping
# (here we use a default distribution mapping)
distribution_mapper_config: {}

```

(continues on next page)

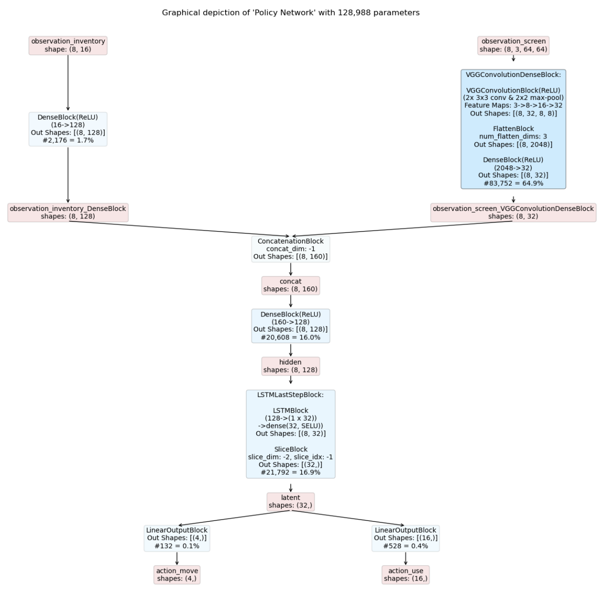
(continued from previous page)

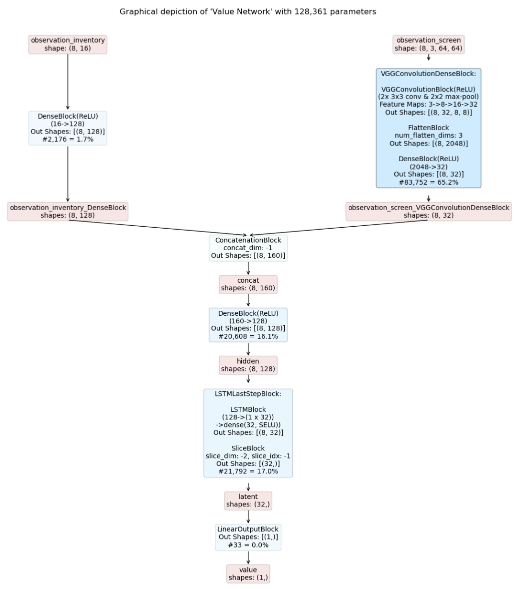
```yaml
# specifies the architecture of default models
model_builder:
  type: maze.perception.builders.ConcatModelBuilder

  # specifies the modality type of each observation
  observation_modality_mapping:
    observation_inventory: feature
    observation_screen: image

  # specifies with which block to process a modality
  modality_config:
    # modality processing
    feature:
      block_type: maze.perception.blocks.DenseBlock
      block_params:
        hidden_units: [32, 32]
        non_lin: torch.nn.ReLU
    image:
      block_type: maze.perception.blocks.VGGConvolutionDenseBlock
      block_params:
        hidden_channels: [8, 16, 32]
        hidden_units: [32]
        non_lin: torch.nn.ReLU
    # preserved keys for the model builder
    hidden:
      block_type: maze.perception.blocks.DenseBlock
      block_params:
        hidden_units: [128]
        non_lin: torch.nn.ReLU
    recurrence:
      block_type: maze.perception.blocks.LSTMLastStepBlock
      block_params:
        hidden_size: 32
        num_layers: 1
        bidirectional: False
        non_lin: torch.nn.SELU

# select policy type
policy:
  type: maze.perception.models.policies.ProbabilisticPolicyComposer

# select critic type
critic:
  type: maze.perception.models.critics.StateCriticComposer
```

Details:

- The main part of the model is identical to the example above.
- However, the example adds an additional recurrent block (LSTMLastStepBlock) considering not only the present but also the *k* previous time steps for its action and value predictions.

The resulting inference graphs for a recurrent actor-critic model are shown below:


Graphical depiction of 'Policy Network' with 128,988 parameters
observation_inventory shape: (8, 16)
observation_screen shape: (8, 3, 64, 64)
DenseBlock(ReLU) (16->128) Out Shapes: [(8, 128)] #2,176 = 1.7%
VGGConvolutionDenseBlock:
VGGConvolutionBlock(ReLU) (2x 3x3 conv & 2x2 max-pool) Feature Maps: 3->8->16->32 Out Shapes: [(8, 32, 8, 8)]
FlattenBlock num_flatten_dims: 3 Out Shapes: [(8, 2048)]
DenseBlock(ReLU) (2048->32) Out Shapes: [(8, 32)] #83,752 = 64.9%
observation_inventory_DenseBlock shapes: (8, 128)
observation_screen_VGGConvolutionDenseBlock shapes: (8, 32)
ConcatenationBlock concat_dim: -1 Out Shapes: [(8, 160)]
concat shapes: (8, 160)
DenseBlock(ReLU) (160->128) Out Shapes: [(8, 128)] #20,608 = 16.0%
hidden shapes: (8, 128)
LSTMLastStepBlock:
LSTMBlock (128->(1 x 32)) ->dense(32, SELU)) Out Shapes: [(8, 32)]
SliceBlock slice_dim: -2, slice_idx: -1 Out Shapes: [(32,)] #21,792 = 16.9%
latent shapes: (32,)
LinearOutputBlock Out Shapes: [(4,)] #132 = 0.1%
LinearOutputBlock Out Shapes: [(16,)] #528 = 0.4%
action_move shapes: (4,)
action_use shapes: (16,)


Graphical depiction of 'Value Network' with 128,361 parameters
observation_inventory shape: (8, 16)
observation_screen shape: (8, 3, 64, 64)
DenseBlock(ReLU) (16->128) Out Shapes: [(8, 128)] #2,176 = 1.7%
VGGConvolutionDenseBlock:
VGGConvolutionBlock(ReLU) (2x 3x3 conv & 2x2 max-pool) Feature Maps: 3->8->16->32 Out Shapes: [(8, 32, 8, 8)]
FlattenBlock num_flatten_dims: 3 Out Shapes: [(8, 2048)]
DenseBlock(ReLU) (2048->32) Out Shapes: [(8, 32)] #83,752 = 65.2%
observation_inventory_DenseBlock shapes: (8, 128)
observation_screen_VGGConvolutionDenseBlock shapes: (8, 32)
ConcatenationBlock concat_dim: -1 Out Shapes: [(8, 160)]
concat shapes: (8, 160)
DenseBlock(ReLU) (160->128) Out Shapes: [(8, 128)] #20,608 = 16.1%
hidden shapes: (8, 128)
LSTMLastStepBlock:
LSTMBlock (128->(1 x 32)) ->dense(32, SELU)) Out Shapes: [(8, 32)]
SliceBlock slice_dim: -2, slice_idx: -1 Out Shapes: [(32,)] #21,792 = 17.0%
latent shapes: (32,)
LinearOutputBlock Out Shapes: [(1,)] #33 = 0.0%
value shapes: (1,)

3.7.5 Example 3: Single Observation and Action Models

Even though designed for more complex models which process multiple observations and prediction multiple actions at the same time you can of course also compose models for simpler use cases.

In this example we utilize the *ConcatModelBuilder* to compose an actor-critic model for OpenAI Gym's CartPole Env. CartPole has an observation space with dimensionality four and a discrete action spaces with two options.

The model config is defined as:

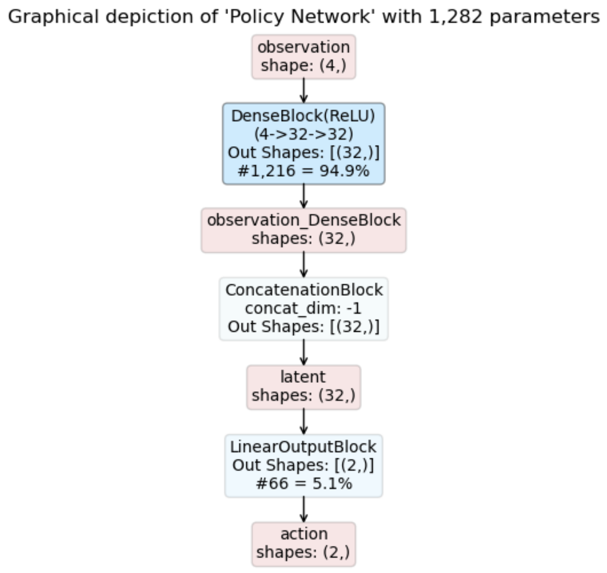
```
# @package model
type: maze.perception.models.default_model_composer.DefaultModelComposer
```

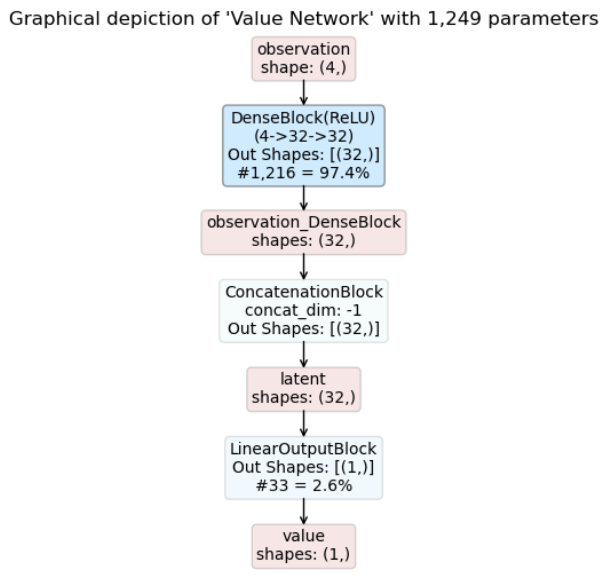
(continues on next page)

(continued from previous page)

```
# specify distribution mapping
# (here we use a default distribution mapping)
distribution_mapper_config: {}

# specifies the architecture of default models
model_builder:
  type: maze.perception.builders.ConcatModelBuilder

  # specifies the modality type of each observation
  observation_modality_mapping:
    observation: feature

  # specifies with which block to process a modality
  modality_config:
    # modality processing
    feature:
      block_type: maze.perception.blocks.DenseBlock
      block_params:
        hidden_units: [32, 32]
        non_lin: torch.nn.ReLU
    # preserved keys for the model builder
    hidden: {}
    recurrence: {}

# select policy type
policy:
  type: maze.perception.models.policies.ProbabilisticPolicyComposer

# select critic type
critic:
  type: maze.perception.models.critics.StateCriticComposer
```

The resulting inference graphs for an actor-critic model are shown below:


Graphical depiction of 'Policy Network' with 1,282 parameters
observation shape: (4,)
DenseBlock(ReLU) (4->32->32) Out Shapes: [(32,)] #1,216 = 94.9%
observation_DenseBlock shapes: (32,)
ConcatenationBlock concat_dim: -1 Out Shapes: [(32,)]
latent shapes: (32,)
LinearOutputBlock Out Shapes: [(2,)] #66 = 5.1%
action shapes: (2,)



Details:

- When there is only one observation, as for the present example, concatenation acts simply as an identity mapping of the previous output tensor (in this case *observation_DenseBlock*).

3.7.6 Where to Go Next

- You can read up on our general introduction to the *Perception Module*.
- Here we explain how to define and work with *custom models* in case the template models are not sufficient.

3.8 Working with Custom Models

The Maze custom *model composer* enables us to explicitly specify application specific models directly in Python. Models can be either written with Maze perception blocks or with plain PyTorch as long as they inherit from Pytorch's nn.Model.

As such models can be easily created, and even existing models from previous work or well known papers can be easily reused with minor adjustments. However, we recommend to create models using the predefined perception blocks in order to speed up writing as well as to take full advantage of features such as shape inference and graphical rendering of the models.

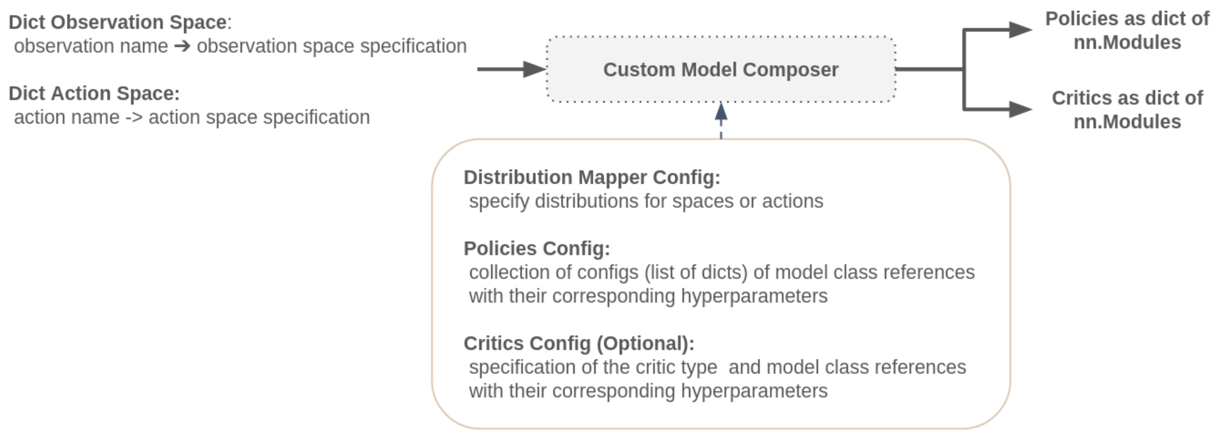
On this page we will first go over the features as well as general working principles. Afterwards we will demonstrate the custom model composer with three examples:

- A simple *feed forward model* for cartpole.
- A more *complex recurrent network* example.
- The cartpole example again but this time using *plain PyTorch* (that is, no Maze-Perception Blocks).

3.8.1 List of Features

The custom model composer supports the following features:

- Specify complex models directly in Python.
- Supports shape inference and shape checks for a given observation space when relying on Maze perception blocks.
- Reuse existing PyTorch nn.Models with minor modifications.
- Stores a graphical rendering of the networks if the inference block is utilized.


Dict Observation Space:
observation name ➔ observation space specification
Dict Action Space:
action name -> action space specification
Custom Model Composer
Policies as dict of nn.Modules
Critics as dict of nn.Modules
Distribution Mapper Config:
specify distributions for spaces or actions
Policies Config:
collection of configs (list of dicts) of model class references with their corresponding hyperparameters
Critics Config (Optional):
specification of the critic type and model class references with their corresponding hyperparameters

Note: All model composers have the single purpose of composing, testing and visualizing the models in code or from a config file. After all models have been created and retrieved the model composer will have served its purpose and is deleted.

3.8.2 The Custom Models Signature (on Action and Observation Shapes)

As previously mentioned the constraints we impose on any model used in conjunction with the custom model composer are twofold: Firstly the network class has to inherit from PyTorch's nn.Model in order to inherit all network specific methods and properties such as the *forward* method. Additionally a given network class has to have specified constructor arguments depending on the type of network.

Policy Networks must have the constructor arguments *obs_shapes* and *action_logits_shapes*. When the models are built in the constructor of the custom model composer these two arguments are passed to the constructor of the model in addition to any other arbitrary arguments specified. As the name suggests *obs_shapes* is a dictionary mapping observation names to their corresponding shapes represented as a sequence of integers. Similarly *action_logits_shapes* is a dictionary that maps action names to their corresponding action *distribution logits shapes*. (These shapes are also represented as a sequence of integers.) Both, observation and action logits shapes are inferred in the model composer utilizing the *observation_spaces_dict*, *action_spaces_dict* and *distribution_mapper*.

Critic Networks require only the constructor argument *obs_shapes*. Any other constructor argument is free for the user to specify.

To summarize the constraints we impose on custom models:

- Policy Networks:
  - inherit from nn.Model

- constructor arguments: *obs_shapes* and *action_logits_shapes*

- Critic Networks:
  - inherit from nn.Model
  - constructor arguments: *obs_shapes*

3.8.3 Example 1: Simple Custom Networks with Perception Blocks

Even though designed for more complex models that process multiple observations and predict multiple actions at the same time you can also compose models for simpler use cases, of course.

In this example we utilize the custom model composer in combination with the perception blocks to compose an actor-critic model for OpenAI Gym's CartPole Env using a single dense block in each network. CartPole has an observation space with dimensionality four and a discrete action space with two options.

The policy model can then be defined as:

```python
"""Shows how to use the custom model composer to build a custom policy network."""
from collections import OrderedDict
from typing import Dict, Union, Sequence, List

import numpy as np
import torch
import torch.nn as nn

from maze.perception.blocks.feed_forward.dense import DenseBlock
from maze.perception.blocks.inference import InferenceBlock
from maze.perception.blocks.output.linear import LinearOutputBlock
from maze.perception.weight_init import make_module_init_normc


class CustomCarpolePolicyNet(nn.Module):
    """Simple feed forward policy network.

    :param obs_shapes: The shapes of all observations as a dict.
    :param action_logits_shapes: The shapes of all actions as a dict structure.
    :param non_lin: The nonlinear activation to be used.
    :param hidden_units: A list of units per hidden layer.
    """

    def __init__(self, obs_shapes: Dict[str, Sequence[int]], action_logits_shapes:␣
↪Dict[str, Sequence[int]],
                 non_lin: Union[str, type(nn.Module)], hidden_units: List[int]):
        super().__init__()

        # Maze relies on dictionaries to represent the inference graph
        self.perception_dict = OrderedDict()

        # build latent embedding block
        self.perception_dict['latent'] = DenseBlock(
            in_keys='observation', out_keys='latent', in_shapes=obs_shapes[
↪'observation'],
            hidden_units=hidden_units,non_lin=non_lin)

        # build action head
        self.perception_dict['action'] = LinearOutputBlock(
```

(continues on next page)

(continued from previous page)

```
            in_keys='latent', out_keys='action', in_shapes=self.perception_dict[
↪'latent'].out_shapes(),
            output_units=int(np.prod(action_logits_shapes["action"])))

        # build inference block
        self.perception_net = InferenceBlock(
            in_keys='observation', out_keys='action', in_shapes=obs_shapes[
↪'observation'],
            perception_blocks=self.perception_dict)

        # apply weight init
        self.perception_net.apply(make_module_init_normc(1.0))
        self.perception_dict['action'].apply(make_module_init_normc(0.01))

    def forward(self, in_tensor_dict: Dict[str, torch.Tensor]) -> Dict[str, torch.
↪Tensor]:
        """Compute forward pass through the network.

        :param in_tensor_dict: Input tensor dict.
        :return: The computed output of the network.
        """
        return self.perception_net(in_tensor_dict)
```

And the critic model as:

```
"""Shows how to use the custom model composer to build a custom value network."""
from collections import OrderedDict
from typing import Dict, Union, Sequence, List

import torch
import torch.nn as nn

from maze.perception.blocks.feed_forward.dense import DenseBlock
from maze.perception.blocks.inference import InferenceBlock
from maze.perception.blocks.output.linear import LinearOutputBlock
from maze.perception.weight_init import make_module_init_normc


class CustomCarpoleCriticNet(nn.Module):
    """Simple feed forward critic network.

    :param obs_shapes: The shapes of all observations as a dict.
    :param non_lin: The nonlinear activation to be used.
    :param hidden_units: A list of units per hidden layer.
    """

    def __init__(self, obs_shapes: Dict[str, Sequence[int]], non_lin: Union[str,␣
↪type(nn.Module)],
                 hidden_units: List[int]):
        super().__init__()

        # Maze relies on dictionaries to represent the inference graph
        self.perception_dict = OrderedDict()

        # build latent embedding block
        self.perception_dict['latent'] = DenseBlock(
            in_keys='observation', out_keys='latent', in_shapes=obs_shapes[
↪'observation'], hidden_units=hidden_units,
```

(continues on next page)

(continued from previous page)

```
            non_lin=non_lin)

        # build action head
        self.perception_dict['value'] = LinearOutputBlock(
            in_keys='latent', out_keys='value', in_shapes=self.perception_dict['latent
↪'].out_shapes(), output_units=1)

        # build inference block
        self.perception_net = InferenceBlock(
            in_keys='observation', out_keys='value', in_shapes=obs_shapes['observation
↪'],
            perception_blocks=self.perception_dict)

        # apply weight init
        self.perception_net.apply(make_module_init_normc(1.0))
        self.perception_dict['value'].apply(make_module_init_normc(0.01))

    def forward(self, in_tensor_dict: Dict[str, torch.Tensor]) -> Dict[str, torch.
↪Tensor]:
        """Compute forward pass through the network.

        :param in_tensor_dict: Input tensor dict.
        :return: The computed output of the network.
        """
        return self.perception_net(in_tensor_dict)
```

An example config for the model composer could then look like this:

```
# @package model

# specify the custom model composer by reference
type: maze.perception.models.custom_model_composer.CustomModelComposer

# Specify distribution mapping
# (here we use a default distribution mapping)
distribution_mapper_config: {}

policy:
  # first specify the policy type
  type: maze.perception.models.policies.ProbabilisticPolicyComposer
  # specify the policy network(s) we would like to use, by reference
  networks:
  - type: docs.source.policy_and_value_networks.code_snippets.custom_cartpole_policy_
↪net.CustomCarpolePolicyNet
    # specify the parameters of our model
    non_lin: torch.nn.ReLU
    hidden_units: [16, 32]

critic:
  # first specify the critic type (here a state value critic)
  type: maze.perception.models.critics.StateCriticComposer
  # specify the critic network(s) we would like to use, by reference
  networks:
    - type: docs.source.policy_and_value_networks.code_snippets.custom_cartpole_
↪critic_net.CustomCarpoleCriticNet
      # specify the parameters of our model
      non_lin: torch.nn.ReLU
```

(continues on next page)

(continued from previous page)

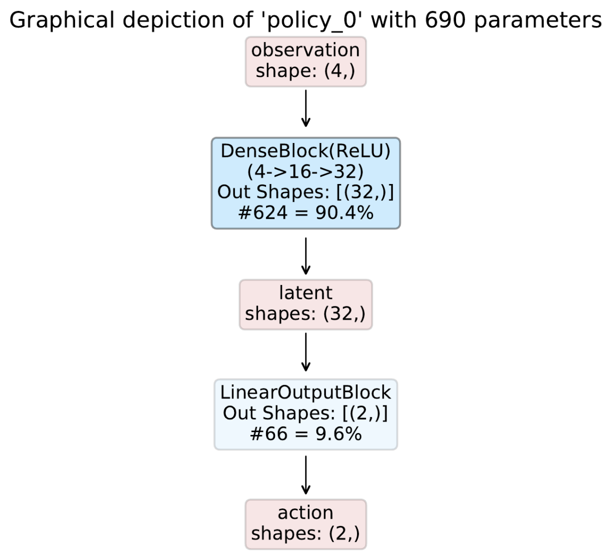
```
      hidden_units: [16, 32]
```

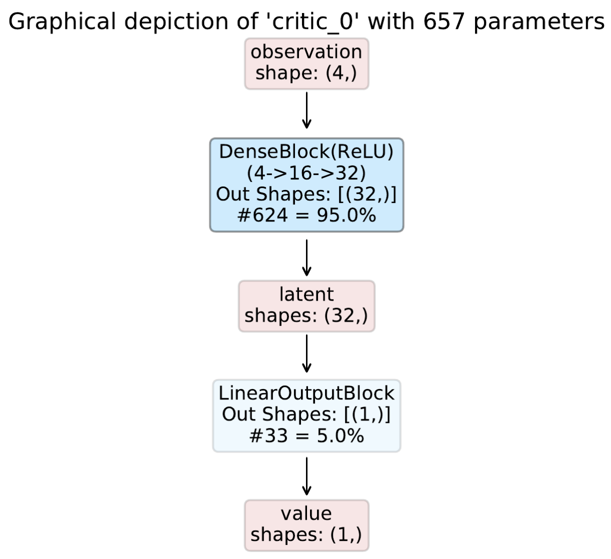
Details:

- Models are composed by the *CustomModelComposer*.
- No specific action space and probability distribution overrides are specified.
- Since we are in a single step environment we only have one policy. Additionally we specify the constructor arguments we defined in the python code above.
- For critics we specify the type to be single-step since we are working with a single step environment. Furthermore, the network and its constructor arguments are specified.

Note: Although mentioned previously, we want to point out the constructor arguments of the two models again: the policy network has the required arguments *obs_shapes* and *action_logits_shapes* in addition to the custom arguments *non_lin* and *hidden_units*. The critic network has only the required argument *obs_shapes* and the same custom arguments as the policy network.

The resulting inference graphs for a recurrent actor-critic model are shown below:


Graphical depiction of 'policy_0' with 690 parameters
observation shape: (4,)
DenseBlock(ReLU) (4->16->32) Out Shapes: [(32,)] #624 = 90.4%
latent shapes: (32,)
LinearOutputBlock Out Shapes: [(2,)] #66 = 9.6%
action shapes: (2,)



3.8.4 Example 2: More Complex Custom Networks with Perception Blocks

Now we will consider the more complex example used in the examples of the *template model composer*.

The observation space is defined as:

- *observation_screen* : a 64 x 64 RGB image
- *observation_inventory* : a 16-dimensional feature vector

The action space is defined as:

- *action_move* : a *categorical action* with four options deciding to move [*UP, DOWN, LEFT, RIGHT*]
- *action_use* : a 16-dimensional *multi-binary action* deciding which item to use from inventory

Since we are interested in building a policy and critic network, where both networks should have the same embedding structure we can create a *base* or *latent space* template:

```
"""Shows how to use the custom model composer to build a complex custom embedding␣
↪networks."""
from collections import OrderedDict
from typing import Dict, Union, Sequence, List

import torch.nn as nn

from maze.perception.blocks.feed_forward.dense import DenseBlock
from maze.perception.blocks.general.concat import ConcatenationBlock
from maze.perception.blocks.joint_blocks.lstm_last_step import LSTMLastStepBlock
from maze.perception.blocks.joint_blocks.vgg_conv_dense import␣
↪VGGConvolutionDenseBlock


class CustomComplexLatentNet:
    """Simple feed forward policy network.

    :param obs_shapes: The shapes of all observations as a dict.
    :param non_lin: The nonlinear activation to be used.
    :param hidden_units: A list of units per hidden layer.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self, obs_shapes: Dict[str, Sequence[int]],
                 non_lin: Union[str, type(nn.Module)], hidden_units: List[int]):
        self.obs_shapes = obs_shapes

        # Maze relies on dictionaries to represent the inference graph
        self.perception_dict = OrderedDict()

        # build latent feature embedding block
        self.perception_dict['latent_inventory'] = DenseBlock(
            in_keys='observation_inventory', out_keys='latent_inventory', in_
↪shapes=obs_shapes['observation_inventory'],
            hidden_units=[128], non_lin=non_lin)

        # build latent pixel embedding block
        self.perception_dict['latent_screen'] = VGGConvolutionDenseBlock(
            in_keys='observation_screen', out_keys='latent_screen', in_shapes=obs_
↪shapes['observation_screen'],
            non_lin=non_lin, hidden_channels=[8, 16, 32], hidden_units=[32])

        # Concatenate latent features
        self.perception_dict['latent_concat'] = ConcatenationBlock(
            in_keys=['latent_inventory', 'latent_screen'], out_keys='latent_concat',
            in_shapes=self.perception_dict['latent_inventory'].out_shapes() +
            self.perception_dict['latent_screen'].out_shapes(), concat_dim=-1)

        # Add latent dense block
        self.perception_dict['latent_dense'] = DenseBlock(
            in_keys='latent_concat', out_keys='latent_dense', hidden_units=hidden_
↪units, non_lin=non_lin,
            in_shapes=self.perception_dict['latent_concat'].out_shapes()
        )

        # Add recurrent block
        self.perception_dict['latent'] = LSTMLastStepBlock(
            in_keys='latent_dense', out_keys='latent', in_shapes=self.perception_dict[
↪'latent_dense'].out_shapes(),
            hidden_size=32, num_layers=1, bidirectional=False, non_lin=non_lin
        )
```

Now using the template we can create the policy:

```
"""Shows how to use the custom model composer to build a complex custom policy␣
↪networks."""
from typing import Dict, Union, Sequence, List

import numpy as np
import torch
import torch.nn as nn

from docs.source.policy_and_value_networks.code_snippets.custom_complex_latent_net␣
↪import \
    CustomComplexLatentNet
from maze.perception.blocks.inference import InferenceBlock
from maze.perception.blocks.output.linear import LinearOutputBlock
from maze.perception.weight_init import make_module_init_normc
```

(continues on next page)

(continued from previous page)

```


class CustomComplexPolicyNet(nn.Module, CustomComplexLatentNet):
    """Simple feed forward policy network.

    :param obs_shapes: The shapes of all observations as a dict.
    :param action_logits_shapes: The shapes of all actions as a dict structure.
    :param non_lin: The nonlinear activation to be used.
    :param hidden_units: A list of units per hidden layer.
    """

    def __init__(self, obs_shapes: Dict[str, Sequence[int]], action_logits_shapes:␣
↪Dict[str, Sequence[int]],
                 non_lin: Union[str, type(nn.Module)], hidden_units: List[int]):
        nn.Module.__init__(self)
        CustomComplexLatentNet.__init__(self, obs_shapes, non_lin, hidden_units)

        # build action heads
        for action_key, action_shape in action_logits_shapes.items():
            self.perception_dict[action_key] = LinearOutputBlock(
                in_keys='latent', out_keys=action_key, in_shapes=self.perception_dict[
↪'latent'].out_shapes(),
                output_units=int(np.prod(action_shape)))

        # build inference block
        in_keys = list(self.obs_shapes.keys())
        self.perception_net = InferenceBlock(
            in_keys=in_keys, out_keys=list(action_logits_shapes.keys()), perception_
↪blocks=self.perception_dict,
            in_shapes=[self.obs_shapes[key] for key in in_keys])

        # apply weight init
        self.perception_net.apply(make_module_init_normc(1.0))
        for action_key in action_logits_shapes.keys():
            self.perception_dict[action_key].apply(make_module_init_normc(0.01))

    def forward(self, in_tensor_dict: Dict[str, torch.Tensor]) -> Dict[str, torch.
↪Tensor]:
        """Compute forward pass through the network.

        :param in_tensor_dict: Input tensor dict.
        :return: The computed output of the network.
        """
        return self.perception_net(in_tensor_dict)
```

And the critic:

```
"""Shows how to use the custom model composer to build a complex custom value␣
↪networks."""
from typing import Dict, Union, Sequence, List

import torch
import torch.nn as nn

from docs.source.policy_and_value_networks.code_snippets.custom_complex_latent_net␣
↪import \
    CustomComplexLatentNet
```

(continues on next page)

(continued from previous page)

```
from maze.perception.blocks.inference import InferenceBlock
from maze.perception.blocks.output.linear import LinearOutputBlock
from maze.perception.weight_init import make_module_init_normc


class CustomComplexCriticNet(nn.Module, CustomComplexLatentNet):
    """Simple feed forward policy network.

    :param obs_shapes: The shapes of all observations as a dict.
    :param non_lin: The nonlinear activation to be used.
    :param hidden_units: A list of units per hidden layer.
    """

    def __init__(self, obs_shapes: Dict[str, Sequence[int]],
                 non_lin: Union[str, type(nn.Module)], hidden_units: List[int]):
        nn.Module.__init__(self)
        CustomComplexLatentNet.__init__(self, obs_shapes, non_lin, hidden_units)

        # build action heads
        self.perception_dict['value'] = LinearOutputBlock(
            in_keys='latent', out_keys='value', in_shapes=self.perception_dict['latent
↪'].out_shapes(),
            output_units=1)

        # build inference block
        in_keys = list(self.obs_shapes.keys())
        self.perception_net = InferenceBlock(
            in_keys=in_keys, out_keys='value', in_shapes=[self.obs_shapes[key] for␣
↪key in in_keys],
            perception_blocks=self.perception_dict)

        # apply weight init
        self.perception_net.apply(make_module_init_normc(1.0))
        self.perception_dict['value'].apply(make_module_init_normc(0.01))

    def forward(self, in_tensor_dict: Dict[str, torch.Tensor]) -> Dict[str, torch.
↪Tensor]:
        """Compute forward pass through the network.

        :param in_tensor_dict: Input tensor dict.
        :return: The computed output of the network.
        """
        return self.perception_net(in_tensor_dict)
```

An example config for the model composer could then look like this:

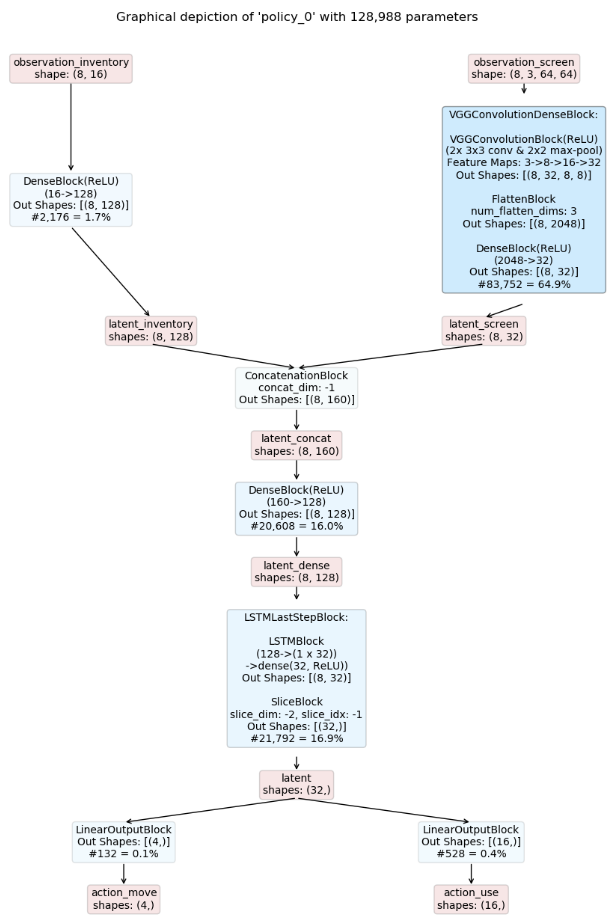
```
# @package model

# specify the custom model composer by reference
type: maze.perception.models.custom_model_composer.CustomModelComposer

# Specify distribution mapping
# (here we use a default distribution mapping)
distribution_mapper_config: {}

policy:
  type: maze.perception.models.policies.ProbabilisticPolicyComposer
```

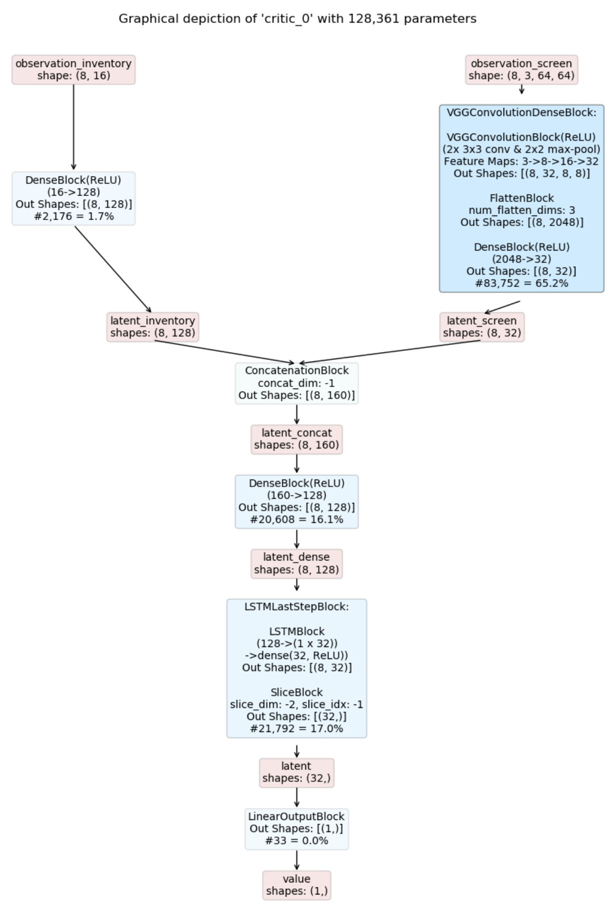
(continues on next page)

(continued from previous page)

```
  networks:
  # specify the policy network we would like to use, by reference
  - type: docs.source.policy_and_value_networks.code_snippets.custom_complex_policy_
↪net.CustomComplexPolicyNet
    # specify the parameters of our model
    non_lin: torch.nn.ReLU
    hidden_units: [128]

critic:
  # first specify the critic type (single step in this example)
  type: maze.perception.models.critics.StateCriticComposer
  networks:
    # specify the critic we would like to use, by reference
    - type: docs.source.policy_and_value_networks.code_snippets.custom_complex_critic_
↪net.CustomComplexCriticNet
      # specify the parameters of our model
      non_lin: torch.nn.ReLU
      hidden_units: [128]
```

The resulting inference graphs for a recurrent actor-critic model are shown below. Note that the models are identical except for the output layers due to the shared base model.


Graphical depiction of 'policy_0' with 128,988 parameters
observation_inventory shape: (8, 16)
observation_screen shape: (8, 3, 64, 64)
VGGConvolutionDenseBlock:
VGGConvolutionBlock(ReLU) (2x 3x3 conv & 2x2 max-pool) Feature Maps: 3->8->16->32 Out Shapes: [(8, 32, 8, 8)]
FlattenBlock num_flatten_dims: 3 Out Shapes: [(8, 2048)]
DenseBlock(ReLU) (2048->32) Out Shapes: [(8, 32)] #83,752 = 64.9%
DenseBlock(ReLU) (16->128) Out Shapes: [(8, 128)] #2,176 = 1.7%
latent_inventory shapes: (8, 128)
latent_screen shapes: (8, 32)
ConcatenationBlock concat_dim: -1 Out Shapes: [(8, 160)]
latent_concat shapes: (8, 160)
DenseBlock(ReLU) (160->128) Out Shapes: [(8, 128)] #20,608 = 16.0%
latent_dense shapes: (8, 128)
LSTMLastStepBlock:
LSTMBlock (128->(1 x 32)) ->dense(32, ReLU)) Out Shapes: [(8, 32)]
SliceBlock slice_dim: -2, slice_idx: -1 Out Shapes: [(32,)] #21,792 = 16.9%
latent shapes: (32,)
LinearOutputBlock Out Shapes: [(4,)] #132 = 0.1%
LinearOutputBlock Out Shapes: [(16,)] #528 = 0.4%
action_move shapes: (4,)
action_use shapes: (16,)


Graphical depiction of 'critic_0' with 128,361 parameters
observation_inventory shape: (8, 16)
observation_screen shape: (8, 3, 64, 64)
VGGConvolutionDenseBlock:
VGGConvolutionBlock(ReLU) (2x 3x3 conv & 2x2 max-pool) Feature Maps: 3->8->16->32 Out Shapes: [(8, 32, 8, 8)]
FlattenBlock num_flatten_dims: 3 Out Shapes: [(8, 2048)]
DenseBlock(ReLU) (2048->32) Out Shapes: [(8, 32)] #83,752 = 65.2%
DenseBlock(ReLU) (16->128) Out Shapes: [(8, 128)] #2,176 = 1.7%
latent_inventory shapes: (8, 128)
latent_screen shapes: (8, 32)
ConcatenationBlock concat_dim: -1 Out Shapes: [(8, 160)]
latent_concat shapes: (8, 160)
DenseBlock(ReLU) (160->128) Out Shapes: [(8, 128)] #20,608 = 16.1%
latent_dense shapes: (8, 128)
LSTMLastStepBlock:
LSTMBlock (128->(1 x 32)) ->dense(32, ReLU)) Out Shapes: [(8, 32)]
SliceBlock slice_dim: -2, slice_idx: -1 Out Shapes: [(32,)] #21,792 = 17.0%
latent shapes: (32,)
LinearOutputBlock Out Shapes: [(1,)] #33 = 0.0%
value shapes: (1,)

3.8.5 Example 3: Custom Networks with (plain PyTorch) Python

Finally, let's have a look at how we can create a custom model without using any Maze-Perception Components. As already mentioned, we still have to specify the constructor arguments *obs_shapes* and *action_logits_shapes* but do not need to use them. Considering again OpenAI Gym's CartPole Env the models could look like this:

The policy model:

```python
"""Shows how to create a custom cartpole model using no maze perception components."""
from typing import Dict, Sequence

import torch
import torch.nn as nn
import torch.nn.functional as F


class CustomPlainCartpolePolicyNet(nn.Module):
    """Simple feed forward policy network.

    :param obs_shapes: The shapes of all observations as a dict.
    :param action_logits_shapes: The shapes of all actions as a dict structure.
    :param hidden_layer_0: The number of units in layer 0.
    :param hidden_layer_1: The number of units in layer 1.
    :param use_bias: Specify whether to use a bias in the linear layers.
```

(continues on next page)

(continued from previous page)

```
    """
    def __init__(self, obs_shapes: Dict[str, Sequence[int]], action_logits_shapes:␣
↪Dict[str, Sequence[int]],
                 hidden_layer_0: int, hidden_layer_1: int, use_bias: bool):
        nn.Module.__init__(self)

        self.observation_name = list(obs_shapes.keys())[0]
        self.action_name = list(action_logits_shapes.keys())[0]

        self.l0 = nn.Linear(4, hidden_layer_0, bias=use_bias)
        self.l1 = nn.Linear(hidden_layer_0, hidden_layer_1, bias=use_bias)
        self.l2 = nn.Linear(hidden_layer_1, 2, bias=use_bias)

    def reset_parameters(self) -> None:
        """Reset the parameters of the Model"""

        self.l0.reset_parameters()
        self.l1.reset_parameters()
        self.l1.reset_parameters()

    def forward(self, in_tensor_dict: Dict[str, torch.Tensor]) -> Dict[str, torch.
↪Tensor]:
        """Compute forward pass through the network.

        :param in_tensor_dict: Input tensor dict.
        :return: The computed output of the network.
        """
        # Retrieve the observation tensor from the input dict
        xx_tensor = in_tensor_dict[self.observation_name]

        # Compute the forward pass thorough the network
        xx_tensor = F.relu(self.l0(xx_tensor))
        xx_tensor = F.relu(self.l1(xx_tensor))
        xx_tensor = self.l2(xx_tensor)

        # Create the output dictionary with the computed model output
        out = dict({self.action_name: xx_tensor})
        return out
```

And the critic model as:

```
"""Shows how to create a custom cartpole model using no maze perception components."""
from typing import Dict, Sequence

import torch
import torch.nn as nn
import torch.nn.functional as F


class CustomPlainCartpoleCriticNet(nn.Module):
    """Simple feed forward critic network.

    :param obs_shapes: The shapes of all observations as a dict.
    :param hidden_layer_0: The number of units in layer 0.
    :param hidden_layer_1: The number of units in layer 1.
    :param use_bias: Specify whether to use a bias in the linear layers.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, obs_shapes: Dict[str, Sequence[int]],
                 hidden_layer_0: int, hidden_layer_1: int, use_bias: bool):
        nn.Module.__init__(self)

        self.observation_name = list(obs_shapes.keys())[0]

        self.l0 = nn.Linear(4, hidden_layer_0, bias=use_bias)
        self.l1 = nn.Linear(hidden_layer_0, hidden_layer_1, bias=use_bias)
        self.l2 = nn.Linear(hidden_layer_1, 1, bias=use_bias)

    def reset_parameters(self) -> None:
        """Reset the parameters of the Model"""

        self.l0.reset_parameters()
        self.l1.reset_parameters()
        self.l1.reset_parameters()

    def forward(self, in_tensor_dict: Dict[str, torch.Tensor]) -> Dict[str, torch.
↪Tensor]:
        """Compute forward pass through the network.

        :param in_tensor_dict: Input tensor dict.
        :return: The computed output of the network.
        """
        # Retrieve the observation tensor from the input dict
        xx_tensor = in_tensor_dict[self.observation_name]

        # Compute the forward pass thorough the network
        xx_tensor = F.relu(self.l0(xx_tensor))
        xx_tensor = F.relu(self.l1(xx_tensor))
        xx_tensor = self.l2(xx_tensor)

        # Create the output dictionary with the computed model output
        out = dict({'value': xx_tensor})
        return out
```

An example config for the model composer could then look like this:

```
# @package model

# specify the custom model composer by reference
type: maze.perception.models.custom_model_composer.CustomModelComposer

# Specify distribution mapping
# (here we use a default distribution mapping)
distribution_mapper_config: {}

policy:
  # first specify the policy type
  type: maze.perception.models.policies.ProbabilisticPolicyComposer
  # specify the policy network(s) we would like to use, by reference
  networks:
  - type: docs.source.policy_and_value_networks.code_snippets.custom_plain_cartpole_
↪policy_net.CustomPlainCartpolePolicyNet
    # specify the parameters of our model
    hidden_layer_0: 16
    hidden_layer_1: 32
```

(continues on next page)

(continued from previous page)

```
    use_bias: True


critic:
  # first specify the critic type (here a state value critic)
  type: maze.perception.models.critics.StateCriticComposer
  # specify the critic network(s) we would like to use, by reference
  networks:
    - type: docs.source.policy_and_value_networks.code_snippets.custom_plain_cartpole_
↪critic_net.CustomPlainCartpoleCriticNet
      # specify the parameters of our model
      hidden_layer_0: 16
      hidden_layer_1: 32
      use_bias: True
```

Note: Since we do not use the *inference block* in this example, no visual representation of the model can be rendered.

3.8.6 Where to Go Next

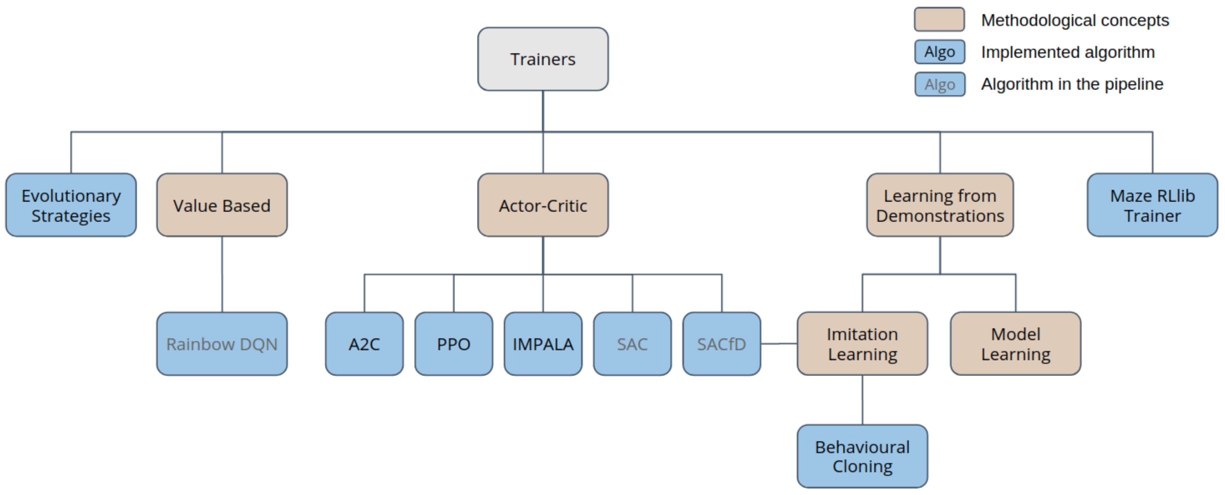
- You can read up on our general introduction to the *Perception Module*.
- We explain how to use the *template model builder* in case the you just want to get started with training.

3.9 Maze Trainers

Trainers are the central components of the Maze framework when it comes to optimizing policies using different RL algorithms. To be more specific, Trainers and TrainingRunners are responsible for the following tasks:

- manage the model types (actor networks, state-critics, state-action-critic, . . .),
- manage agent environment interaction and trajectory data generation,
- compute the loss (specific to the algorithm used),
- update the weights in order to decrease the loss and increase the performance,
- collect and log training statistics,
- manage model checkpoints and the training process (e.g., early stopping).

The figure below provides an overview of the currently supported Trainers.



This page gives a general (high-level) overview of the Trainers and corresponding algorithms supported by the Maze framework. For more details especially on the implementation please refer to the *API documentation on Trainers*. For more details on the training workflow and how to start trainings using the Hydra config system please refer to the *training section*.

Overview

- *Supported Spaces*
- *Advantage Actor-Critic (A2C)*
- *Proximal Policy Optimization (PPO)*
- *Importance Weighted Actor-Learner Architecture (IMPALA)*
- *Behavioural Cloning (BC)*
- *Evolutionary Strategies (ES)*
- *Maze RLlib Trainer*
- *Where to Go Next*

3.9.1 Supported Spaces

If not stated otherwise, Maze Trainers support dictionary spaces for both observations and actions.

If the environment you are working with does not yet interact via dictionary spaces simply wrap it with the built-in `DictActionWrapper` for actions and `DictObservationWrapper` for observations. In case of standard Gym environments just use the `GymMazeEnv`.

3.9.2 Advantage Actor-Critic (A2C)

A2C is a synchronous version of the originally proposed Asynchronous Advantage Actor-Critic (A3C). As a policy gradient method it maintains a probabilistic policy, computing action selection probabilities, as well as a critic, predicting the state value function. By setting the number of rollout steps as well as the number of parallel environments one can control the batch size used for updating the policy and value function in each iteration.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937).

Example

```
maze-run -cn conf_train env.name=CartPole-v0 algorithm=a2c model=vector_obs␣
↪critic=default_state
```

Algorithm Parameters | *A2CAlgorithmConfig*

```
# @package algorithm

# number of epochs to train
n_epochs: 0

# number of updates per epoch
epoch_length: 25

# run evaluation in deterministic mode (argmax-policy)
deterministic_eval: false

# number of evaluation trials
eval_repeats: 2

# number of steps used for early stopping
patience: 15

# number of critic (value function) burn in epochs
critic_burn_in_epochs: 0

# Number of steps taken for each rollout
n_rollout_steps: 100

# learning rate
lr: 0.0005

# discounting factor
gamma: 0.98

# bias vs variance trade of factor for Generalized Advantage Estimator (GAE)
gae_lambda: 1.0

# weight of policy loss
policy_loss_coef: 1.0

# weight of value loss
value_loss_coef: 0.5

# weight of entropy loss
entropy_coef: 0.00025
```

(continues on next page)

(continued from previous page)

```
# The maximum allowed gradient norm during training
max_grad_norm: 0.0

# Either "cpu" or "cuda"
device: cpu
```

Runner Parameters | `ACRunner`

```
# @package runner
type: "maze.train.trainers.common.actor_critic.actor_critic_runners.ACDevRunner"

# model class used for policy and critic updates
trainer_class: maze.train.trainers.a2c.a2c_trainer.MultiStepA2C

# Number of concurrently executed environments
concurrency: 2

# Path to initial state (can contain: policy weights, critic weights, optimizer state)
initial_state_dict: ~
```

```
# @package runner
type: "maze.train.trainers.common.actor_critic.actor_critic_runners.ACLocalRunner"

# model class used for policy and critic updates
trainer_class: maze.train.trainers.a2c.a2c_trainer.MultiStepA2C

# Number of concurrently executed environments
concurrency: 8

# Path to initial state (can contain: policy weights, critic weights, optimizer state)
initial_state_dict: ~
```

3.9.3 Proximal Policy Optimization (PPO)

The PPO algorithm belongs to the class of actor-critic style policy gradient methods. It optimizes a "surrogate" objective function adopted from trust region methods. As such, it alternates between generating trajectory data via agent rollouts from the environment and optimizing the objective function by means of a stochastic mini-batch gradient ascent.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Example

```
maze-run -cn conf_train env.name=CartPole-v0 algorithm=ppo model=vector_obs␣
↪critic=default_state
```

Algorithm Parameters | *PPOAlgorithmConfig*

```
# @package algorithm

# number of epochs to train
n_epochs: 0
```

(continues on next page)

(continued from previous page)

```
# number of updates per epoch
epoch_length: 25

# run evaluation in deterministic mode (argmax-policy)
deterministic_eval: false

# number of evaluation trials
eval_repeats: 2

# number of steps used for early stopping
patience: 15

# number of critic (value function) burn in epochs
critic_burn_in_epochs: 0

# Number of steps taken for each rollout
n_rollout_steps: 100

# learning rate
lr: 0.00025

# discounting factor
gamma: 0.98

# bias vs variance trade of factor for Generalized Advantage Estimator (GAE)
gae_lambda: 1.0

# weight of policy loss
policy_loss_coef: 1.0

# weight of value loss
value_loss_coef: 0.5

# weight of entropy loss
entropy_coef: 0.00025

# The maximum allowed gradient norm during training
max_grad_norm: 0.0

# Either "cpu" or "cuda"
device: cpu

# The batch size used for policy and value updates
batch_size: 100

# Number of epochs for for policy and value optimization
n_optimization_epochs: 4

# Clipping parameter of surrogate loss
clip_range: 0.2
```

Runner Parameters | `ACRunner`

```
# @package runner
type: "maze.train.trainers.common.actor_critic.actor_critic_runners.ACDevRunner"

# model class used for policy and critic updates
```

(continues on next page)

(continued from previous page)

```
trainer_class: maze.train.trainers.ppo.ppo_trainer.MultiStepPPO

# Number of concurrently executed environments
concurrency: 8

# Path to initial state (can contain: policy weights, critic weights, optimizer state)
initial_state_dict: ~
```

```
# @package runner
type: "maze.train.trainers.common.actor_critic.actor_critic_runners.ACLocalRunner"

# model class used for policy and critic updates
trainer_class: maze.train.trainers.ppo.ppo_trainer.MultiStepPPO

# Number of concurrently executed environments
concurrency: 8

# Path to initial state (can contain: policy weights, critic weights, optimizer state)
initial_state_dict: ~
```

3.9.4 Importance Weighted Actor-Learner Architecture (IMPALA)

IMPALA is a RL algorithm able to scale to a very large number of machines. Multiple workers collect trajectories (sequences of states, actions and rewards), which are communicated to a learner responsible for updating the policy by utilizing stochastic mini-batch gradient decent and the proposed V-trace correction algorithm. By decoupling roll-outs (interactions with the environment) and policy updates the algorithm is considered off-policy and asynchronous, making it very suitable for compute-intense environments.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., & Kavukcuoglu, K. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. arXiv preprint arXiv:1802.01561.

Example

```
maze-run -cn conf_train env.name=CartPole-v0 algorithm=impala model=vector_obs␣
↪critic=default_state
```

Algorithm Parameters | *ImpalaAlgorithmConfig*

```
# @package algorithm

# number of epochs to train
n_epochs: 0

# number of updates per epoch
epoch_length: 25

# run evaluation in deterministic mode (argmax-policy)
deterministic_eval: false

# number of evaluation trials
eval_repeats: 2

# number of evaluation envs
eval_concurrency: 2
```

(continues on next page)

(continued from previous page)

```
# number of steps used for early stopping
patience: 15

# this factor multiplied by the actor_batch_size gives the size of the queue for
# the agents output collected by the learner. Therefor if the all rollouts computed␣
↪can be at most
# (queue_out_of_sync_factor + num_agents/actor_batch_size) out of sync with learner␣
↪policy
queue_out_of_sync_factor: 1

# number of rolloutstep of each epoch substep
n_rollout_steps: 100

# number of actors to combine to one batch
actors_batch_size: 8

# number of actors to be run
num_actors: 8

# learning rate
lr: 0.0002

# discount factor
gamma: 0.98

# coefficient of the policy used in the loss calculation
policy_loss_coef: 1.0

# coefficient of the value used in the loss calculation
value_loss_coef: 0.5

# coefficient of the entropy used in the loss calculation
entropy_coef: 0.00025

# max grad norm for gradient clipping, ignored if value==0
max_grad_norm: 0

# A scalar float32 tensor with the clipping threshold for importance weights
# (rho) when calculating the baseline targets (vs). rho^bar in the paper. If None, no␣
↪clipping is applied.
vtrace_clip_rho_threshold: 1.0

# A scalar float32 tensor with the clipping threshold on rho_s in
# \rho_s \delta log \pi(a|x) (r + \gamma v_{s+1} - V(x_sfrom_importance_weights)). If␣
↪None, no clipping is
# applied.
vtrace_clip_pg_rho_threshold: 1.0

# the type of reward clipping to be used, options 'abs_one', 'soft_asymmetric', 'None'
reward_clipping: "None"

# Device of the learner (either cpu or cuda)
# Note that the actors collecting rollouts are always run on CPU.
device: "cpu"
```

Runner Parameters | `ImpalaRunner`

```
# @package runner
type: "maze.train.trainers.impala.impala_runners.ImpalaDevRunner"
```

```
# @package runner
type: "maze.train.trainers.impala.impala_runners.ImpalaLocalRunner"

# type of startmethod used for multiprocessing: 'forkserver', 'spawn', 'fork', 'dummy'
start_method: forkserver
```

3.9.5 Behavioural Cloning (BC)

Behavioural cloning is a simple imitation learning algorithm, that infers the behaviour of a "hidden" policy by imitating the actions produced for a given observation in a supervised learning setting. As such, it requires a set of training (example) trajectories collected prior to training.

Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2017). Imitation learning: A survey of learning methods. ACM Computing Surveys (CSUR), 50(2), 1-35.

Example: *Imitation Learning and Fine-Tuning*

Algorithm Parameters | `BCAlgorithmConfig`

```
# @package algorithm

# Number of epochs to train for
n_epochs: 1000

# Number of iterations after which to run evaluation (in addition to evaluations at␣
↪the end of
# each epoch, which are run automatically). If set to None, evaluations will run on␣
↪epoch end only.
eval_every_k_iterations: 500

# Number of episodes to run during each evaluation rollout
n_eval_episodes: 10

# Optimizer used to update the policy
optimizer:
  type: torch.optim.Adam
  lr: 0.001

# Device to train on
device: cuda

# Batch size
batch_size: 100

# Number of workers to run evaluation in
n_eval_workers: 4

# Percentage of the data used for validation.
validation_percentage: 20
```

Runner Parameters | `ImitationRunner`

```
# @package runner
type: "maze.train.trainers.imitation.ImitationRunner"

# Specify the Dataset class used to load the trajectory data for training
dataset:
  type: maze.train.trainers.imitation.in_memory_data_set.InMemoryImitationDataSet
  trajectory_data_dir: trajectory_data
```

```
# @package runner
type: "maze.train.trainers.imitation.ImitationRunner"

# Specify the Dataset class used to load the trajectory data for training
dataset:
  type: maze.train.trainers.imitation.in_memory_data_set.InMemoryImitationDataSet
  trajectory_data_dir: trajectory_data
```

3.9.6 Evolutionary Strategies (ES)

Evolutionary strategies is a black box optimization algorithm that utilizes direct policy search and can be very efficiently parallelized. Advantages of this methods include being invariant to action frequencies as well as delayed rewards. Further, it shows tolerance for extremely long time horizons, since it does need to compute or approximate a temporally discounted value function. However, it is considered less sample efficient then actual RL algorithms.

Salimans, T., Ho, J., Chen, X., Sidor, S., & Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864.

Example

```
maze-run -cn conf_train env.name=CartPole-v0 algorithm=es model=vector_obs
```

Algorithm Parameters | *ESAlgorithmConfig*

```
# @package algorithm

# Minimum number of episode rollouts per training iteration (=epoch)
n_rollouts_per_update: 10

# Minimum number of cumulative env steps per training iteration (=epoch).
# The training iteration is only finished, once the given number of episodes
# AND the given number of steps has been reached. One of the two parameters
# can be set to 0.
n_timesteps_per_update: 0

# The number of epochs to train before termination. Pass 0 to train indefinitely
max_epochs: 0

# Limit the episode rollouts to a maximum number of steps. Set to 0 to disable this␣
↪option.
max_steps: 0

# The optimizer to use to update the policy based on the sampled gradient.
optimizer:
  type: maze.train.trainers.es.optimizers.adam.Adam
  step_size: 0.01

# L2 weight regularization coefficient.
```

(continues on next page)

(continued from previous page)

```
l2_penalty: 0.005

# The scaling factor of the random noise applied during training.
noise_stddev: 0.02
```

Runner Parameters | *ESMasterRunner*

```
# @package runner
type: "maze.train.trainers.es.ESDevRunner"

# Fixed number of evaluation runs per epoch.
n_eval_rollouts: 10

# Number of float values in the deterministically generated pseudo-random table
shared_noise_table_size: 100000000
```

3.9.7 Maze RLlib Trainer

Finally, the Maze framework also contains an RLlib trainer class. This special class of trainers wraps all necessary and convenient Maze components into RLlib compatible objects such that Ray-RLlib can be reused to train Maze policies and critics. This enables us to train *Maze Models* with *Maze action distributions* in *Maze environments* with almost all RLlib algorithms.

Example and Details: *Maze RLlib Runner*

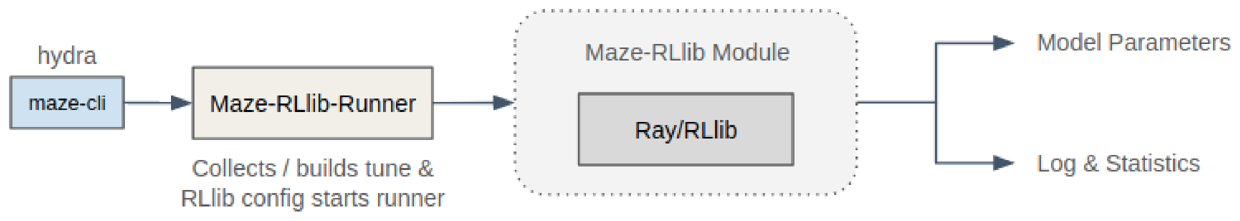
3.9.8 Where to Go Next

- You can read up on our general introduction to the *Maze Training Workflow*.
- To build and use custom Maze models please refer to *Maze Perception Module*.
- You can also look up the supported *Action Spaces and Distributions Module*

3.10 Maze RLlib Runner

The RLlib Runner allows to use RLlib Trainers in combination with Maze models and environments. Ray-RLlib is one of the most popular RL frameworks (algorithm collections) within the scientific community but also when it comes to practical relevance. It already comprises an extensive and tuned collection of various different RL training algorithms. To gain access to RLlib's algorithm collection while still having access to all of practical Maze features we introduce the Maze Rllib Module. It basically wraps *Maze models* (including our extensive Perception Module), *Maze environments* (including wrappers) as well as the customizable *Maze action distributions*. It further allows us to use the Maze hydra cmd-line interfaces together with RLlib while at the same time using the well optimized algorithms from RLlib.

This page gives an overview of the RLlib module and provides examples on how to apply it.



3.10.1 List of Features

- Use Maze environments, models and action distributes in conjunction with RLlib algorithms.
- Make full use of the Maze environment customization utils (wrappers, pre-processing, . . .).
- Use the hydra cmd-line interface to start training runs.
- Models trained with the Maze RLlib Runner are fully compatible with the remaining framework (except when using the default RLlib models).

3.10.2 Example 1: Training with Maze-RLlib and Hydra

Using RLlib algorithms with Maze and Hydra works analogously to *starting training* with native *Maze Trainers*. To train the CartPole environment with RLlib's PPO, run:

```
$ maze-run -cn conf_rllib env.name=CartPole-v0 rllib/algorithm=ppo
```

Here the `-cn conf_rllib` argument specifies to use the `conf_rllib.yaml` (available in `maze-rllib`) package, as our root config file. It specifies the way how to use RLlib trainers within Maze. (For more on root configuration files, see *Hydra overview*.)

3.10.3 Example 2: Overwriting Training Parameters

Similar to *native Maze trainers*, the parametrization of RLlib training runs is also done via Hydra. The main parameters for customizing training and are:

- Environment (`env` configuration group), configuring which environment the training runs on, this stays the same as in maze-train for example.
- Algorithm (`rllib/algorithm` configuration group), specifies the algorithm and its configuration (*all supported algorithms*).
- Model (`model` configuration group), specifying how the models for policies and (optionally) critics should be assembled, this also stays the same as in maze-train.
- Runner (`rllib/runner` configuration group), specifies how training is run (e.g. locally, in development mode). The runner is also the main object responsible for administering the whole training run.. The runner is also the main object responsible for administering the whole training run.

To train with a different algorithm we simply have to specify the `rllib/algorithm` parameter:

```
$ maze-run -cn conf_rllib env.name=CartPole-v0 rllib/algorithm=a3c
```

Furthermore, we have full access to the algorithm hyper parameters defined by RLlib and can overwrite them. E.g., to change the learning rate and rollout fragment length, execute

```
$ maze-run -cn conf_rllib env.name=CartPole-v0 rllib/algorithm=a3c \
  algorithm.config.lr=0.001 algorithm.config.rollout_fragment_length=50
```

3.10.4 Example 3: Training with RLlib's Default Models

Finally, it is also possible to utilize the RLlib default model builder by specifying `model=rllib`. This will load the rllib default model and parameters, which can again be customized via Hydra:

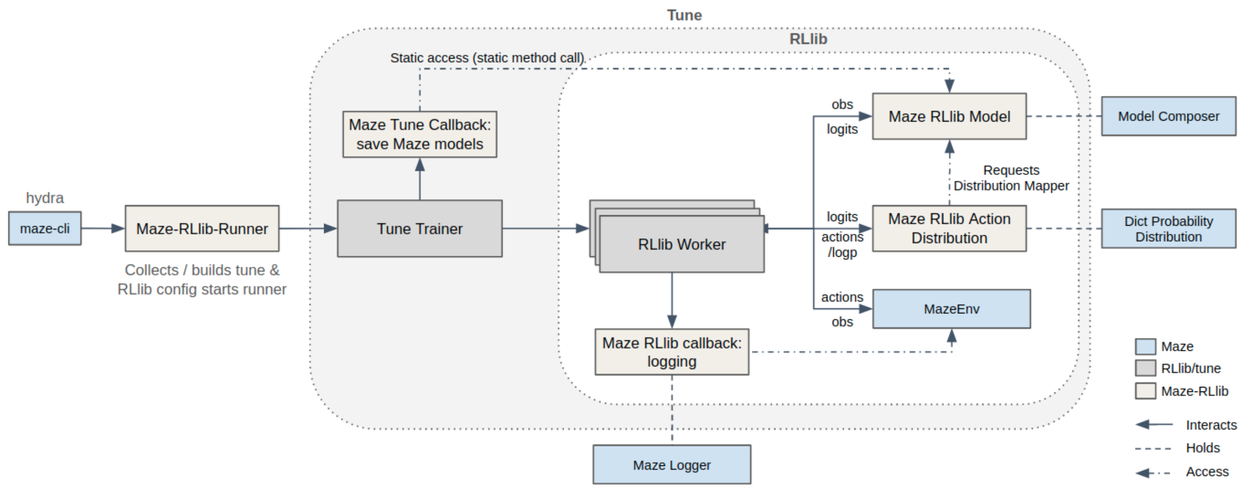
```
$ maze-run -cn conf_rllib env.name=CartPole-v0 model=rllib \
  model.fcnet_hiddens=[128,128] model.vf_share_layers=False
```

3.10.5 Supported Algorithms

- Advantage Actor-Critic (A2C, A3C)
- Deep Deterministic Policy Gradients (DDPG, TD3)
- Distributed Prioritized Experience Replay (Ape-X)
- Deep Q Networks (DQN, Rainbow, Parametric DQN)
- Importance Weighted Actor-Learner Architecture (IMPALA)
- Model-Agnostic Meta-Learning (MAML)
- Policy Gradients (PG)
- Proximal Policy Optimization (PPO)
- Asynchronous Proximal Policy Optimization (APPO)

3.10.6 The Bigger Picture

The figure below shows an overview of how the RLlib Module connects to the different Maze components in more detail:


Tune
RLlib
Static access (static method call)
Maze Tune Callback: save Maze models
hydra
maze-cli
Maze-RLlib-Runner
Collects / builds tune & RLlib config starts runner
Tune Trainer
RLlib Worker
obs
logits
Maze RLlib Model
Model Composer
Requests Distribution Mapper
logits
actions /logp
Maze RLlib Action Distribution
Dict Probability Distribution
actions
obs
MazeEnv
Maze RLlib callback: logging
Maze Logger
Maze
RLlib/tune
Maze-RLlib
Interacts
Holds
Access

3.10.7 Good to Know

Tip: Using the the argument `rllib/runner=dev` starts ray in local mode, by default sets the number workers to 1 and increases the log level (resulting in more information being printed). This is especially useful for debugging.

Tip: When *watching the training progress* of RLlib training runs with Tensorboard make sure to start Tensorboard with `--reload_multifile true` as both Maze and RLlib will dump an event log.

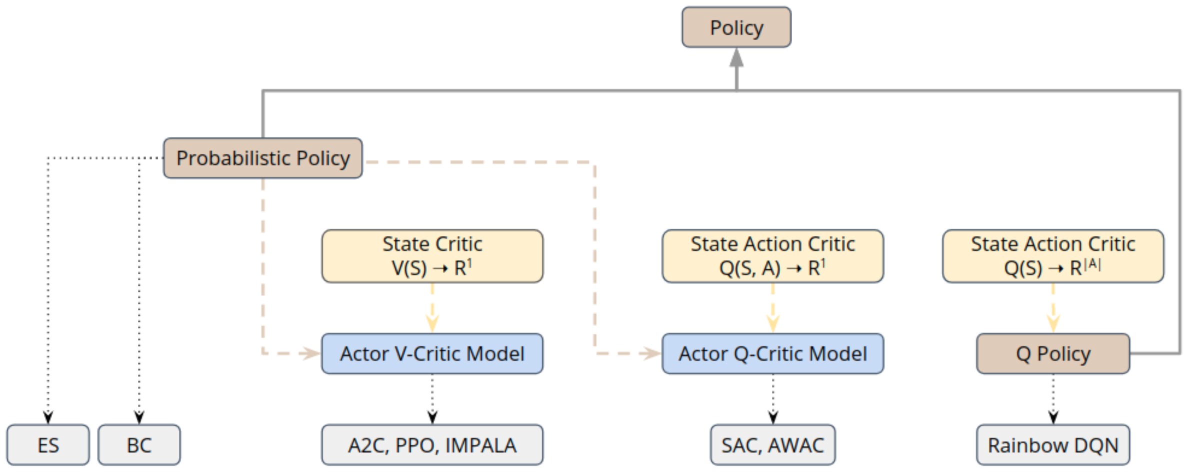
3.10.8 Where to Go Next

- After training, you might want to *rollout* the trained policy to further evaluate it or record the actions taken.
- To create a custom Maze environment, you might want to review *Maze environment hierarchy* and *creating a Maze environment from scratch*.
- To build and use custom Maze models please refer to *Maze Perception Module*.
- For more details on Hydra and how to use it go to *configuration with Hydra*.
- You can read up on our general introduction to the *Maze training workflow*.

3.11 Policies, Critics and Agents

Depending on the domain and task we are working on we rely on *different trainers* (algorithm classes) to appropriately address the problem at hand. This in turn implies different requirements for the respective *models* and contained policy and value estimators.

The figure below provides a conceptual overview of the model classes contained in Maze and relates them to compatible algorithm classes and trainers.



Note that all policy and critics are compatible with *Structured Environments*.

3.11.1 Policies (Actors)

An agent holds one or more policies and acts (selects actions) according to these policies. Each policy consists of one ore more policy networks. This might be for example required in (1) multi-agent RL settings where each agents acts according to its distinct policy network or (2) when working with auto-regressive action spaces or *multi-step environments*.

In case of *Policy Gradient Methods*, such as the actor-critic learners A2C or PPO, we rely on a **probabilistic policy** defining a conditional action selection probability distribution $\pi(a|s)$ given the current State s.

In case of value-based methods, such as DQN, the **Q-policy** is defined via the state-action value function $Q(s, a)$ (e.g, by selecting the action with highest Q value: $\text{argmax}_a Q(s, a)$).

3.11.2 Value Functions (Critics)

Maze so far supports two different kinds of critics. A standard **state critic** represented via a scalar value function $V(S)$ and a **state-action critic** represented either via a scalar state-action value function $Q(S, A)$ or its vectorized equivalent $Q(S)$ predicting the state-action values for all actions at once.

Analogously to policies each critic holds one or more value networks depending on the current usage scenario we are in (auto-regressive, multi-step, multi-agent, . . .). The table below provides an overview of the different critics styles.

State Critic $V(S)$	
TorchStepStateCritic	Each sub-step or actor gets its individual state critic.
TorchSharedStateCritic	One state critic is shared across all sub-steps or actors.
State-Action Critic $Q(S, A)$	
TorchStepStateActionCritic	Each sub-step or actor gets its individual state-action critic.
TorchSharedStateActionCritic	One state-action critic is shared across all sub-steps or actors.

3.11.3 Actor-Critics

To conveniently work with algorithms such as A2C, PPO, IMPALA or SAC we provide a *TorchActorCritic* model to unifying the different policy and critic types into one model.

3.11.4 Where to Go Next

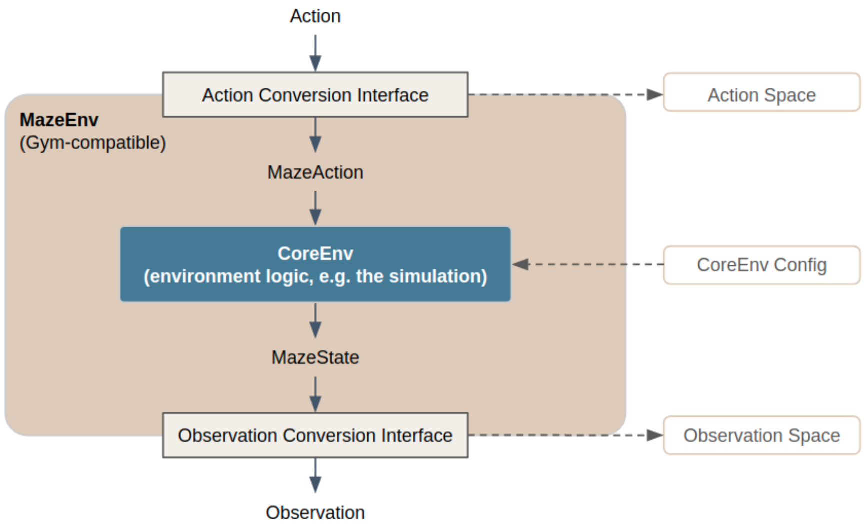
- For further details please see the *reference documentation*.
- To see how to actually implement policy and critic networks see the *Perception Module*.
- You can see the list of available *probability distributions* for probabilistic policies.
- You can also follow up on the available *Maze trainers*.

3.12 Maze Environment Hierarchy

When working with an environment, it is desirable to maintain some modularity in order to be able to, for example, test different configurations of action and observation spaces, modify or record rollouts, or turn an existing flat environment into a structured one.

This page explains how Maze achieves such modularity by breaking down the Maze environment into smaller components and utilizing wrappers. It also provides a high-level overview of what you need to do to use a new or existing custom environment with Maze. (You can find guidance on that at the end of each section.)

For more references on the individual components or on how to write a new environment from scratch, see the *Where to Go Next section* at the end.



The following sections describe the main components:

- *Core environment*, which implements the main environment mechanics, and works with MazeState and MazeAction objects.
- *Observation- and ActionConversionInterfaces* which turn MazeState and MazeAction objects (custom to the core environment) into actions and observations (instances of Gym-compatible spaces which can be fed into a model).
- *Maze env*, which encapsulates the core environment and implements functionality common to all environments above it (e.g. manages observation and action conversion).
- *Wrappers*, which add a great degree of flexibility by allowing you to encapsulate the environment and observe or modify its behavior.
- *Structured environment interface*, which Maze uses to model more complex scenarios such as multi-step (auto-regressive), multi-agent or hierarchical settings.

Here, we explain what parts a Maze environment is composed of and how to apply wrappers.

3.12.1 Core Environment

Core environment implements the main mechanics and functionality of the environment. Its interface is compatible with the Gym environment interface with functions such as `step` and `reset`.

The `step` function of the core environment takes an MazeAction object and returns a MazeState object. There are no strict requirements on how these objects should look – their structure is dependent on the needs of the core environment. However, note that these objects should be serializable, so that they can be easily recorded as part of trajectory data and inspected later.

Besides the Gym interface, core environment interface also contains a couple of hooks that make it easy to support various features of maze, like recording trajectory of your rollouts and then replaying these in a Jupyter notebook. These method include, e.g., `get_renderer()` and `get_serializable_components()`. You don't have to use these if you don't need them (e.g. just return an empty dictionary to `get_serializable_components()` if there are no additional components you would like to serialize with trajectory data) – but then, some features of Maze might not be available.

If you want to use a new or existing environment with Maze, core environment is where you should start. Implement the core environment interface in your environment or encapsulate your environment in an core environment subclass.

3.12.2 Gym-Space Interfaces

Observation- and ActionConversionInterfaces translate MazeState and MazeAction objects (custom to the core environment) into actions and observations (instances of Gym-compatible spaces, i.e., usually a dictionary of numpy arrays which can be fed into a model) and vice versa.

It makes sense to extract this functionality in a separate objects, as format of actions and observations often needs to be swapped (to allow for different trained policies or heuristics). Treating space interfaces as separate objects encapsulates their configuration and separates it from the core environment functionality (which does not need to be changed when only, e.g., the format of the action space is being changed).

If you are creating a new Maze environment, you will need to implement at least one pair of interfaces – one for conversion of MazeStates into observations that your models can later consume, and other one for converting the actions produced by the model to the MazeActions your environment works with.

For more information on the space interfaces and how to customize your environment with them, refer to *Customizing Core and Maze Environments*.

3.12.3 Maze Environment

Maze environment encapsulates the core environment together with the space interfaces. Here, the functionality shared across all core environments is implemented – like management of the space interfaces, support for statistics and logging, and else.

Maze environment is the smallest unit that an RL agent can interact with, as it encapsulates the core functionality implemented by the core environment, space interfaces that translate the MazeState and MazeAction so that the model can understand it, and support for other optional features of Maze that you can add (like statistics logging).

If you are creating a new environment, you will likely not need to think of the Maze environment class much, as it is mostly concerned with functionalities shared across all Maze environments. You will still need to subclass it to have a distinct Maze environment class for your environment, but usually it is enough to override the initializer, there is no need to modify any of its other functionalities.

3.12.4 Wrappers

(This section provides an overview. See also *Wrappers* for more details.)

Wrappers are a very flexible way how to modify behavior of an environment. As the name implies, a wrapper encapsulates the whole environment in it. This means that the wrapper has complete control over the behavior of the environment and can modify it as suited.

Note also that another wrapper can also be applied to an already wrapped environment. In this case, each method call (such as `step`) will traverse through the whole wrapper stack, from the outer-most wrapper to the Maze env, with each wrapper being able to intercept and modify the call.

Maze provides superclasses for commonly used wrapper types:

- **ObservationWrapper** can manipulate the observation before it reaches the agent. Observation wrappers are used for example for *observation normalization wrapper* or masking. Usually, this is the most common type of wrapper used.
- **RewardWrapper** can manipulate the reward before it reaches the model.
- **ActionWrapper** can manipulate the action the model produced before it is converted using ActionConversionInterface in Maze environment.
- **Wrapper** is the common superclass of all the wrappers listed above. It can be subclassed directly if you need to provide some more elaborate functionality, like turning your flat environment into a *structured multi-step one*

If you are creating a new Maze environment, wrappers are optional. Unless you have some very special needs, the wide variety of wrappers provided by Maze (like *observation normalization wrapper* or *trajectory recording wrapper*) should work with any Maze environment out of the box. However, you might need to implement a custom wrapper if you want to modify the behavior of your environment in some more customized manner, like turning your flat environment into a *structured multi-step one*.

For more information on wrappers and customization, see *Wrappers*.

3.12.5 Structured Environments

Loop uses the `StructuredEnv` concept to model more complex settings, such as multi-step (auto-regressive), multi-agent or hierarchical settings.

While such settings can indeed be quite complex, the `StructuredEnv` interface itself is rather simple under the hood. In summary, during each step in the environment:

1. The agent needs to ask which policy should act next. The environment exposes this using the `actor_id` method.
2. The agent then should evaluate the observation using the policy corresponding to the current actor ID, and issue the desired action using the `step` function in a usual Gym-like manner.

Note that the Actor ID, which identifies the currently active policy, is composed of (1) the sub-step key and (2) the index of the current actor in scope of this sub-step (as in some settings, there might be multiple actors per sub-step key).

Maze uses the `StructuredEnv` interface in all settings by default, and other Maze components like `TorchPolicy` support it (and make it convenient to work with) out of the box.

3.12.6 Where to Go Next

After understanding how Maze environment hierarchy works, you might want to:

- See how *Hydra configuration* works and how *environments can be customized* through it
- See more about how to *customize an existing environment with wrappers*
- Get more information on how to write a new *Maze environment from scratch*
- See how Maze environments *dispatch events* to facilitate statistics collection and other forms of logging
- Understand how *policies and agents* are structured

Also, note that the classes described above (like Core environment and Maze environment) themselves implement a set of interfaces that facilitate some of Maze functions, like **EventEnvMixin** interfacing the *Event system* or **RenderEnvMixin** facilitating *rendering*. You will likely not need to work with these directly, and hence they are not described here in detail. However, if you need to know more about these, head to the **reference documentation**.

3.13 Maze Event System

The Maze event system is a convenient way how to monitor notable events happening in the environment as the agent is interacting with it. This page explains the motivation behind it, gives an overview of how it is used in Maze (pointing to other relevant sections), and briefly explains how it works under the hood.

3.13.1 Motivation

Standard metrics such as reward and step count provide a high-level overview of how an agent is doing in an environment, but don't provide more detailed information on the actual behavior.

On the other hand, visualizing or otherwise inspecting the full environment state gives very detailed information on the behavior in some particular time frame, but is difficult to compare and aggregate across episodes or training runs.

In Maze, event system fills the space between – providing more information about environment mechanics than just watching the reward, while making it easy to log, understand, and compare it across episodes and rollouts.

3.13.2 What is an event?

As the name suggests, an event is something notable that happens during the agent-environment interaction loop. For example, when the inventory is full in the example 2D cutting env, a piece will be discarded and the corresponding event will be fired:

```
self.inventory_events.piece_discarded(piece=(50, 30))
```

As can be seen above, events carry a descriptive name, encapsulate the details (like the dimensions of the discarded piece), and are part of a topic (like "inventory events").

While there are some general events that apply to all environments (like reward-related events or KPIs), in general, environments declare their own topics and events as they see fit.

To understand how to declare and integrate custom events into your environment, see the *adding events and KPIs* tutorial.

3.13.3 How are events used in Maze?

There are three main things events are used for throughout Maze:

1. **Reward aggregation.** Reward aggregators declare which events they desire to observe, and then calculate the reward on top of them. This makes it possible to keep reward aggregators decoupled from the environment, which means they can be configured and changed easily. (Check out *reward aggregation* and the *tutorial* for more information.)
2. **Statistics and KPIs.** Event declarations can be annotated using decorators which specify how they should be aggregated on different levels (i.e., step, episode, and epoch). The statistics system then aggregates the events into statistics during trainings and rollouts, and displays these statistics in Tensorboard and console. This makes it possible to understand the agent's behavior much better than if only high-level statistics such as reward and step count were observed. (For more information, see how statistics are *logged* and *calculated*.)
3. **Raw event data logging.** Events and their details are logged in CSV format, which makes them easy to access and analyze later via any custom tools. (While the CSV format should be suitable for most data-analysis tools out there, it is also possible to extend the logging functionality via custom writers if needed.)

For any other custom needs, it is possible to plug into the event system directly through the `Pubsub` or `EventEnvMixin` interfaces.

3.13.4 PubSub: Dispatching and Observing Events

Each core environment maintains its own `Pubsub` message broker (stands for publisher-subscriber). Using the broker, it is possible to register event topics (created as described in the *tutorial*), register subscribers (which will then collect the dispatched events), and dispatch events themselves.

```python
# In a core env (which maintains a pubsub broker)

# Create a topic
inventory_events = self.pubsub.create_event_topic(InventoryEvents)

# Register a subscriber (can be a reward aggregator
# or any other class implementing the Subscriber interface)
self.pubsub.register_subscriber(my_subscriber)

# Dispatch an event
inventory_events.piece_discarded(piece=(50, 10))
```

Note that the subscriber must implement the `Subscriber` interface and declare which events it want to be notified about. This pattern is used by `RewardAggregators`, and the *tutorial on adding reward aggregation* is also a good place to start for any other custom needs.

3.13.5 EventEnvMixin Interface: Querying Events

Core environment also records all events dispatched during the last time step and makes them accessible using the `EventEnvMixin` interface. If you only need to query events dispatched during the last timestep, this option might be more lightweight than registering with the `Pubsub` message broker.

```
env.get_step_events()
```

To see the interface in action, you might want to check out the `LogStatsWrapper`, which uses this interface to query events for *aggregation*.

3.13.6 Where to Go Next

After understanding the main concepts of the event system, you might want to:

- See how *reward aggregation* works and how to *implement it in an environment from scratch*
- Check out the *statistics logging* in Tensorboard and console
- Review how the *events and KPI* aggregation works

3.14 Configuration with Hydra

Here, we explain the configuration scheme of the Maze framework, which explain how to configure your environment and other components using YAML files, run your experiments via CLI, and customize the runs via CLI overrides.

The Maze framework utilizes the Hydra configuration framework. These pages aim to give you a quick overview of how Maze uses Hydra and what its capabilities are, so that you can get up to speed configuring Maze quickly without much prior Hydra knowledge. It also points to relevant parts of Hydra docs if you would like to go deeper.

3.14.1 Hydra: Overview

The motivation behind using Hydra is primarily to:

- Keep separate components (e.g., environment, policy) in individual YAML files which are easier to understand
- Run multiple experiments with different components (like using two different environment configurations, or training with PPO vs. A2C) without duplicating the whole config file
- Make components/values different from the defaults immediately visible (with, e.g., `maze-run runner=sequential`)

Below, the core concepts of Hydra as Maze uses it are described:

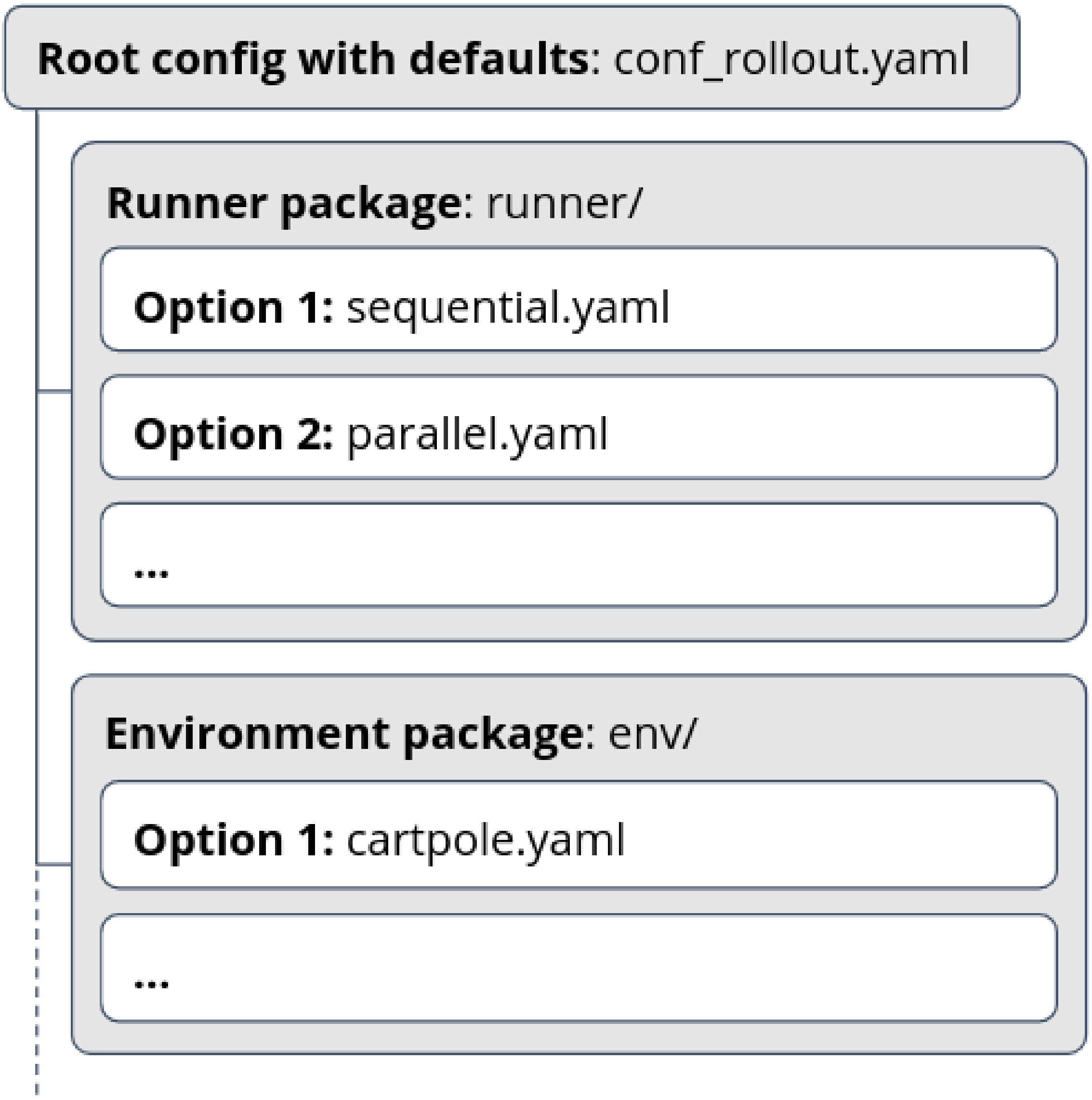
- *Introduction* explains the core concepts of assembling configuration with Hydra
- *Config Root & Defaults* explains how the root config file works and how default components are specified
- *Overrides* show how you can easily customize the config parameters without duplicating the config file, and have Hydra assemble the config and log it for you
- *Output Directory* shows how Hydra creates separate directories for your runs automatically. It is a bit separated from the previous concepts but still important for running your jobs.
- *Runner concept* section explains how the Hydra config is handled by Maze to launch various kinds of jobs (like rollout or train runs) with different configurations

Introduction

Hydra is a configuration framework that, in short, allows you to:

1. Break down your configuration into multiple smaller YAML files (representing individual components)
2. Launch your job through CLI providing overrides for individual components or values and have Hydra assemble the config for you

Ad (1): For illustrative purposes, this is an example of how your Hydra config structure can look like:


Root config with defaults: conf_rollout.yaml
Runner package: runner/
Option 1: sequential.yaml
Option 2: parallel.yaml
...
Environment package: env/
Option 1: cartpole.yaml
...

Ad (2): With the structure above, you could then launch your jobs with specified components (again, this is only for illustrative purposes):

```
$ maze-run runner=parallel
```

Or, you can even override any individual value anywhere in the config like this:

```
$ maze-run runner=parallel runner.n_processes=10
```

You can also review the basic example and composition example at Hydra docs.

Configuration Root, Groups and Defaults

The starting place for a Hydra config is the **root configuration file**. It lists (1) the individual **configuration groups** that you would like to use along with their defaults, and (2) any other configuration that is universal. A simple root config file might look like this (all of these examples are snippets taken from `maze` config, shortened for brevity):

```
# These are the individual config components with their defaults
defaults:
  - runner: parallel
  - env: cartpole
  - wrappers: default
    optional: true
  # ...

# Other values that are universally applicable (still can be changed with overrides)
log_base_dir: outputs

# ...
```

The snippet `runner:  parallel` tells Hydra to look for a file `runner/parallel.yaml` and transplant its contents under the `runner:` key. (If `optional:  true` is specified, Hydra does not raise an error when such a config file cannot be found.)

Hence, if the `runner/parallel.yaml` file looks like this:

```
n_processes: 5
n_episodes: 50
# ...
```

the final assembled config would look like this:

```
runner:
n_processes: 5
  n_episodes: 50
  # ...
env:
  # ...
```

Overrides

When running your job through a command line, you can customize individual bits of your configuration via command-line arguments.

As briefly demonstrated above, you can override individual defaults in the config file. For example, when running a Maze rollout, the default runner is `parallel`, but you could specify the sequential runner instead:

```
$ maze-run runner=sequential
```

Besides overriding specifying the config components, you can also override individual values in the config:

```
$ maze-run runner=sequential runner.max_episode_steps=1000
```

There is also more advanced syntax for adding/removing config keys and other patterns – for this, you can consult Hydra docs regarding basic overrides and extended override syntax.

Output Directory

Hydra also by default handles the output directory for each job you run.

By default, `outputs` is used as the base output directory and a new subdirectory is create inside for each run. Here, Hydra also logs the configuration for the current job in the `.hydra` subdirectory, so that you can always get back to it.

You can override the hydra output directory as follows:

```
$ maze-run hydra.run.dir=my_dir
```

More on the output directory setting can be found in Hydra docs: output/working directory and customizing working directory pattern.

Maze Runner Concept

In Maze, the `maze-run` command (that you have seen above already) is the single central utility for launching all sorts of jobs, like training or rollouts.

Under the hood, when you launch such a job, the following happens:

1. Maze checks the `runner` part of the Hydra configuration that was passed through the command. And instantiates a runner object from it (subclass of *`Runner`*).

   (The `runner` component of the configuration always specifies the Runner class to be instantiated, along with any other arguments it needs at initialization.)

2. Maze then calls the `run` method on the instantiated runner and passes it the whole config, as obtained from Hydra.

This enables the `maze-run` command to keep a lot of variability without much coupling of the individual functionalities. For example, rollouts are run through subclasses of *`RolloutRunner`* and trainings through subclasses of *`TrainingRunner`*.

You are also free to create your own subclasses for rollouts, trainings or any completely different use cases.

Where to Go Next

After understanding the basics of how Maze uses Hydra, you might want to:

- Try *running a rollout* using Hydra configuration through command line to put these ideas into action
- *Create custom Hydra configuration files* for your project
- Understand the *advanced concepts of Hydra*

3.14.2 Hydra: Your Own Configuration Files

We encourage you to add custom config files in your own project. These will make it easy for you to launch different versions of your environments and agents with different parameters.

To be able to use custom configuration files, you first need to *create your config module and add it to the Hydra search path*. Then, you can either create just *your own config modules* (e.g., when you just need to customize the environment config), or *create your own root config file if you have more custom needs*.

Step 1: Custom Config Module in Hydra Search Path

For this, first, create a module where your config will reside (let's say `your_project.conf`) and place an `__init__.py` file in there.

Then, add this config module to the Hydra search path by creating the following Hydra plugin (substitute `your_project.conf` with your actual config module path):

```python
# Inside your project in: hydra_plugins/add_custom_config_to_search_path.py

"""Hydra plugin to register additional config packages in the search path."""
from hydra.core.config_search_path import ConfigSearchPath
from hydra.plugins.search_path_plugin import SearchPathPlugin


class AddCustomConfigToSearchPathPlugin(SearchPathPlugin):
    """Hydra plugin to register additional config packages in the search path."""

    def manipulate_search_path(self, search_path: ConfigSearchPath) -> None:
        """Add custom config to search path (part of SearchPathPlugin interface)."""
        search_path.append("project", "pkg://your_project.conf")
```

Now, you can add additional root config files as well as individual components into your config package.

For more information on search path customization, check Config Search Path and SearchPathPlugins in Hydra docs.

Step 2a: Custom Config Components

If what you are after is only providing custom options for some of the components Maze configuration uses (e.g., a custom environment configuration), then it suffices to add these into the relevant directory in your config module and you are good to go.

For example, if you want a custom configuration for the Gym Car Racing env, you might do:

```yaml
# In your_project/conf/env/car_racing.yaml:

# @package env
type: maze.core.wrappers.maze_gym_env_wrapper.GymMazeEnv
env: "CarRacing-v0"
```

Then, you might call `maze-run` with the `env=car_racing` override and it will load the configuration from your file.

Depending on your needs, you can mix-and-match your custom configurations with configurations provided by Maze (e.g. use a custom `env` configuration while using a `wrappers` or `models` configuration provided by Maze).

Step 2b: Custom Root Config

If you need more customization, you will likely need to define your own root config. This is usually useful for custom projects, as it allows you to create custom defaults for the individual config groups.

We suggest you start by copying one of the root configs already available in Maze (like `conf_rollout` or `conf_train`, depending on what you need), and then adding more required keys or removing those that are not needed. However, it is also not difficult to start from scratch if you know what you need.

Once you create your root config file (let's say `your_project/conf/my_own_conf.yaml`), it suffices to point Hydra to it via the argument `-cn my_own_conf`, so your command would look like this (for illustrative purposes):

```
$ maze-run -cn my_own_conf
```

Then, all the defaults and available components that Hydra will look for depend on what you specified in your new root config.

For an overview of root config, check out *config root & defaults*.

Step 3: Custom Runners (Optional)

If you want to launch different types of jobs than what Maze provides by default, like implementing a custom training algorithm or deployment scenario that you would like to run via the CLI, you will benefit from creating a custom `Runner`.

You can subclass the desired class in the runner hierarchy (like the `TrainingRunner` if you are implementing a new training scheme, or the general `Runner` for some more general concept). Then, just create a custom config file for the `runner` config group that configures your new class, and you are good to go.

Where to Go Next

After understanding how custom configuration is done, you might want to:

- Review the *Hydra overview* to see how you should structure your custom configuration
- Read about the *advanced concepts of Hydra*

3.14.3 Hydra: Advanced Concepts

This page features a collection of more advanced Hydra features which are used throughout the framework.

Interpolation

Hydra is based on OmegaConf and supports interpolation.

Interpolation allows us to reference and reuse a value defined elsewhere in the configuration, without repeating it. For example:

```
original:
  value: 1  # We want to reference this value elsewhere
some:
  other:
    structure: ${original.value}  # Reference
```

A (somewhat limited) form of interpolation is used also in specializations described below.

Specializations

Specializations are parts of config that depend on multiple components. For example, your wrapper configuration might depend on both the environment chosen (e.g., `gym_pixel_env` or `gym_feature_env`) and your model (e.g., `default` or `rnn`) – if using an RNN, you might want to include *ObservationStackWrapper*, but its configuration also depends on the environment used.

Then, specializations come to the rescue. In your root config file, you can include a specialization like this (for illustrative purposes):

```
defaults:
  - env: gym_pixel_env
  - model: default
  - env_model: ${defaults.1.env}-${defaults.2.model}
    optional: true
```

Then, when you run this configuration with `env=gym_pixel_env` and `model=rnn`, Hydra will look into the `env_model` directory for configuration named `gym_pixel_env-rnn.yaml`. This allows you to capture the dependencies between these two components easily without having to specify more overrides.

Specializations are well explained in Hydra docs here.

Where to Go Next

After understanding advanced Hydra configuration, you might want to:

- *Create custom Hydra configuration files* for your project
- Review the root configurations available in the Maze framework (as they are a good basis for your custom configurations)

3.15 Environment Rendering

In cases when reviewing the statistics and event logs provided by the *event system* does not provide enough insight, rendering the environment state in a particular time step is helpful.

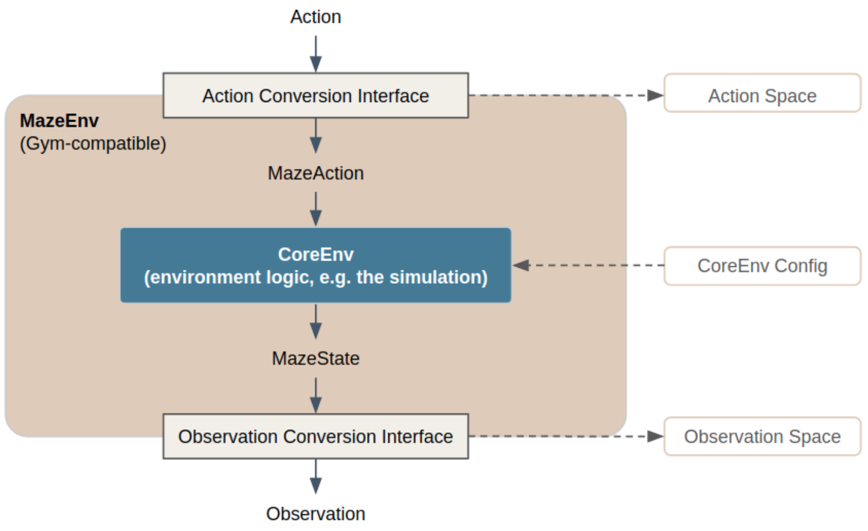
Maze supports two rendering modes:

1. **Rendering online during the rollout.** This is possible simply using the sequential rollout runner for a rollout, and setting the rendering flag to true using the following overrides: `runner=sequential runner.render=true`.
2. **Rendering offline, in a Jupyter notebook, from trajectory data collected earlier.** For environments which provide a Maze-compatible render, *rollouts* can be rendered and browsed retroactively. Review *collecting and visualizing rollouts* for more details. (Unfortunately, this mode is not yet supported for ordinary Gym envs – unless a custom Maze-compatible renderer is provided.)

3.16 Customizing Core and Maze Envs

Whenever simulations reach a certain level of complexity or (ideally) already exist, but have been developed for other purposes than the RL scenario, the Gym-style environment interfaces might not be sufficient anymore to meet all technical requirements (e.g., the state is too complex to be represented as a simple Gym-style numpy array). In case of existing simulations it probably was not even taken into account at all and we have to deal with simulation specific interfaces and objects.

To cope with such situations Maze introduces a few additional concepts which are summarized in the figure below. Before we continue with some practical *examples* emphasizing why this structure is useful for environment customization and convenient experimentation, we first describe the concepts and components in a bit more detail. You can also find these components in the *reference documentation*.



Observation- and ActionConversionInterfaces:

Maze introduces *MazeStates* and *MazeActions*, extending *Observations* and *Actions* (represented as numerical arrays) to simulation specific generic objects. This grants more freedom in choosing appropriate environment-specific representations to separate the data model from the numerical representation, which in turn greatly simplifies the development and readability of environment and engineered baseline agent implementations.

- **Action:** the Gym-style, machine readable action.
- **MazeAction:** the simulation specific representation of the action (e.g., an arbitrary Python object).
- **ActionConversionInterface:** maps agent actions to environment (simulation) specific MazeActions and defines the respective Gym action space.
- **Observation:** the Gym-style, machine readable observation (e.g., a numpy array).
- **MazeState:** the simulation specific representation of the observation (e.g. an arbitrary Python object).
- **ObservationConversionInterface:** maps simulation MazeStates to Gym-style observations and defines the respective Gym observation space.

Core and Maze Environments:

The same distinction is carried out for environments.

- **CoreEnv:** this is the central environment, which could be also seen as the simulation, forming the basis for actual, RL trainable environments. CoreEnvs accept *MazeAction* objects as input and yield *MazeState* objects as response.

- **CoreEnv Config:** configuration parameters for the CoreEnvironment (the simulation).
- **MazeEnv:** wraps the CoreEnvs as a Gym-style environment in a reusable form, by utilizing the interfaces (mappings) from the MazeState to the observations space and from the MazeAction to the action space.

3.16.1 List of Features

Introducing the concepts outlined above allows the following:

- Implement and maintain observations and actions as arbitrarily complex, simulation specific objects (MazeStates and MazeActions). In many cases sticking to Gym spaces gets quite cumbersome and makes the development processes unnecessarily complex.
- Easily experiment with different observation and action spaces simply by switching the Observation- and ActionConversionInterface.
- Train agents based on existing 3rd party simulations (environments) by implementing the Observation- and ActionConversionInterfaces (of course this also requires to have a Python API available).
- Easy configuration of the CoreEnv (simulation).

3.16.2 Example: Core- and MazeEnv Configuration

The config snippet below shows an example environment configuration for the built-in cutting-2d environment.

```
# @package env
type: maze_envs.logistics.cutting_2d.env.maze_env.Cutting2DEnvironment

# parametrizes the core environment (simulation)
core_env:
  max_pieces_in_inventory: 1000
  raw_piece_size: [100, 100]
  demand_generator:
    type: mixed_periodic
    n_raw_pieces: 3
    m_demanded_pieces: 10
    rotate: True
  # defines how rewards are computed
  reward_aggregator:
    type: maze_envs.logistics.cutting_2d.reward.default.DefaultRewardAggregator

# defines the conversion of actions to executions
action_conversion
  - type: maze_envs.logistics.cutting_2d.space_interfaces.action_conversion.dict.
↪ActionConversion
    max_pieces_in_inventory: 1000

# defines the conversion of states to observations
observation_conversion:
  - type: maze_envs.logistics.cutting_2d.space_interfaces.observation_conversion.dict.
↪ObservationConversion
    max_pieces_in_inventory: 1000
    raw_piece_size: [100, 100]
```

The config defines:

- which MazeEnv to use,

- the parametrization of the CoreEnv including *reward computation*,
- how MazeStates are converted to observations and
- how actions are converted to MazeActions.

All components together compose a concrete RL problem instance as a trainable environment. In particular, whenever you would like to experiment with specific aspects of your RL problem (e.g. tweak the observation space) you only have to exchange the respective part of your environment configuration.

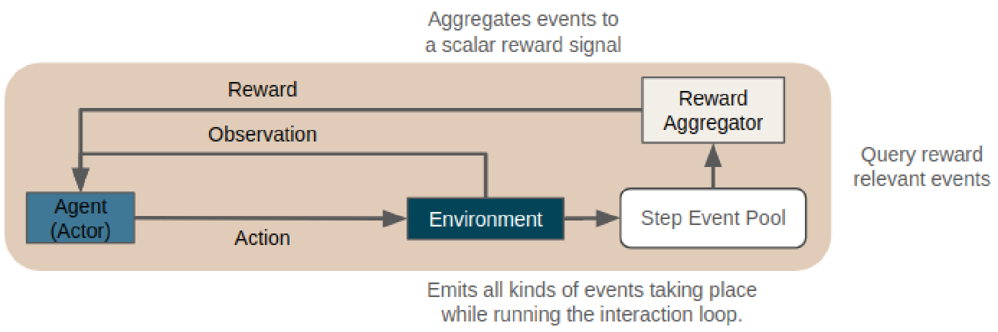
Note: As showing concrete implementations of a CoreEnv or the Observation- and ActionConversionInterfaces is beyond the scope of this page we refer to the *Maze - step by step tutorial* for details.

3.16.3 Where to Go Next

- You might want to get a bigger picture of the *Maze environment hierarchy*.
- Learn how to customize with *environment wrappers*.
- Learn about *reward customization and shaping*.
- See the special wrappers for *observation pre-processing* and *observation normalization*.

3.17 Customizing / Shaping Rewards

In a reinforcement learning problem the overall goal is defined via an appropriate reward signal. In particular, reward is attributed to certain, problem specific key events. During the training process the agent then has to discover a policy (behaviour) that maximizes the cumulative future reward over time. In case of a meaningful reward signal such a policy will be able to successfully address the decision problem at hand.



From a technical perspective, reward customization in Maze is based on the general *event system* (which also serves other purposes) and is implemented via `RewardAggregators`. In summary, after each step, the reward aggregator gets access to all the events the environment dispatched during the step (e.g., a new item was replenished to inventory), and can then calculate arbitrary rewards based on these events. This means it is possible to modify and shape the reward signal based on different events and their characteristics by plugging in different reward aggregators without further modifying the environment.

Below we show how to get started with reward customization by *configuring the CoreEnv* and by *implementing a custom reward*.

3.17.1 List of Features

Maze event-based reward computation allows the following:

- Easy experimentation with different reward signals.
- Implementation of custom rewards without the need to modify the original environment (simulation).
- Computing rewards based on multiple components of the environment as well as global events.
- Combining multiple different objectives into one multi-objective reward signal.
- Computation of multiple rewards in the same env, each based on a different set of components (multi agent).

3.17.2 Configuring the CoreEnv

The following config snippet shows how to specify reward computation for a CoreEnv via the field `reward_aggregator`. You only have to set the reference path of the RewardAggregator and reward computation will be carried out accordingly in all experiments based on this config.

For further details on the remaining entries of this config you can read up on how to *customize Core- and MazeEnvs*.

```yaml
# @package env
type: maze_envs.logistics.cutting_2d.env.maze_env.Cutting2DEnvironment

# parametrizes the core environment (simulation)
core_env:
  max_pieces_in_inventory: 1000
  raw_piece_size: [100, 100]
  demand_generator:
    type: mixed_periodic
    n_raw_pieces: 3
    m_demanded_pieces: 10
    rotate: True
  # defines how rewards are computed
  reward_aggregator:
    type: maze_envs.logistics.cutting_2d.reward.default.DefaultRewardAggregator

# defines the conversion of actions to executions
action_conversion
  - type: maze_envs.logistics.cutting_2d.space_interfaces.action_conversion.dict.
↪ActionConversion
    max_pieces_in_inventory: 1000

# defines the conversion of states to observations
observation_conversion:
  - type: maze_envs.logistics.cutting_2d.space_interfaces.observation_conversion.dict.
↪ObservationConversion
    max_pieces_in_inventory: 1000
    raw_piece_size: [100, 100]
```

3.17.3 Implementing a Custom Reward

This section contains a concrete implementation of a reward aggregator for the built-in cutting environment.

In summary, the reward aggregator first declares which events it is interested in (the get_interfaces method). At the end of the step, after all the events have been accumulated, the reward aggregator is asked to calculate the reward (the summarize_reward method). This is the core of the reward computation – you can see how the events are queried and the reward assembled based on their values.

```
"""Assigns negative reward for relying on raw pieces for delivering an order."""
from typing import List

from maze.core.annotations import override
from maze.core.events.pubsub import Subscriber
from maze_envs.logistics.cutting_2d.env.events import InventoryEvents
from maze.core.env.reward import RewardAggregatorInterface


class RawPieceUsageRewardAggregator(RewardAggregatorInterface):
    """
    Reward scheme for the 2D cutting env penalizing raw piece usage.

    :param reward_scale: Reward scaling factor.
    """
    def __init__(self, reward_scale: float):
        super().__init__()
        self.reward_scale = reward_scale

    @override(Subscriber)
    def get_interfaces(self) -> List:
        """Specification of the event interfaces this subscriber wants to receive␣
↪events from.
        Every subscriber must implement this configuration method.

        :return: A list of interface classes.
        """
        return [InventoryEvents]

    def summarize_reward(self) -> float:
        """Summarize reward based on the orders and pieces to cut.

        :return: the summarized scalar reward.
        """

        # iterate replenishment events and assign reward accordingly
        reward = 0.0
        for _ in self.query_events(InventoryEvents.piece_replenished):
            reward -= 1.0

        # rescale reward with provided factor
        reward *= self.reward_scale

        return reward

    @classmethod
    @override(RewardAggregatorInterface)
    def to_scalar_reward(cls, reward: float) -> float:
        """Nothing to do here for this env as the reward is already a scalar.
```

(continues on next page)

(continued from previous page)

```
        This method is useful for example in a multi-agent setting
        where we could sum over multiple actors to assign a joint reward.

        :param reward: Here already a scalar reward.
        :return: The scalar reward returned by the environment.
        """
        return reward
```

When adding a new reward aggregator you (1) have to implement the `RewardAggregatorInterface` and (2) make sure that it is accessible within your Python path.

Besides that you only have to provide the reference path of the `reward_aggregator` to use:

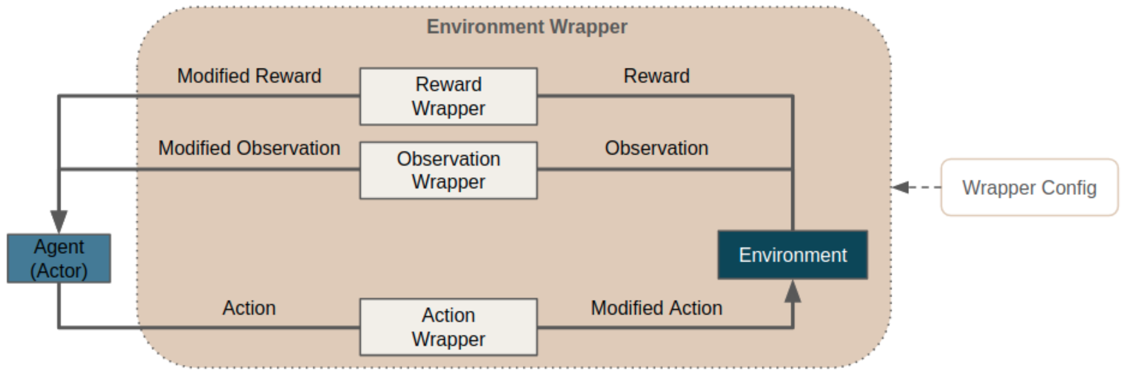
```
reward_aggregator:
    type: my_project.custom_reward.RawPieceUsageRewardAggregator
    reward_scale: 0.1
```

3.17.4 Where to Go Next

- Additional options for customizing environments can be found under the entry "*Environment Customization*" in the sidebar.
- For further technical details we highly recommend to read up on the *Maze event system*.
- To see another application of the event system you can read up on the *Maze logging system*.

3.18 Environment Wrappers

Environment wrappers are an elegant way to modify and customize environments for RL training and experimentation. As the name already suggests, they wrap an existing environment and allow to modify different parts of the agent-environment interaction loop including observations, actions, the reward or any other internals of the environment itself.



To gain access to the functionality of Maze environment wrappers you simply have to add a wrapper stack in your hydra configuration. To get started just copy one of our *hydra config snippets* below or *use it directly within Python*.

Note: Wrappers have been already introduced in OpenAi's Gym and elegantly expose methods and attributes of all nested envs. However, wrapping destroys the class hierarchy, querying the base classes is not straight-forward. Maze environment wrappers fix the behaviour of isinstance() for arbitrarily nested wrappers.

3.18.1 List of Features

Maze environment wrappers allows the following:

- Easy customization of environments: (observations, actions, reward, internals)
- Convenient development of concepts such as *observation pre-processing* and *observation normalization*.
- Preserves the class hierarchy of nested environments.

3.18.2 Example 1: Customizing Environments with Wrappers

To make use of Maze environment wrappers just add a config snippet as listed below.

```
# @package wrappers
RandomResetWrapper:
  min_skip_steps: 0
  max_skip_steps: 100
TimeLimitWrapper:
  max_episode_steps: 1000
```

Details:

- It applies the specified wrappers in the defined order from top to bottom.
- Adds a RandomResetWrapper randomly skipping the first 0 to 100 frames
- Adds a TimeLimitWrapper restricting the maximum temporal horizon of the environment

3.18.3 Example 2: Using Custom Wrappers

In case the *built-in wrappers* provided with Maze are not sufficient for your use case you can of course implement and add additional custom wrappers.

```
# @package wrappers
my_project.wrappers.custom_wrapper.CustomObserverWrapper:
  parameter_1: 0.5
  parameter_2: 1000
```

When adding a new environment wrappers you (1) have to implement the **Wrapper** interface and (2) make sure that it is accessible within your Python path. Besides that you only have to provide the reference path of the wrapper to use, plus any parameters the wrapper initializer needs.

3.18.4 Example 3: Plain Python Configuration

If you are not working with the Maze command line interface but still want to use wrappers directly within Python you can start with the code snippet below.

```python
"""Contains an example showing how to add wrappers."""
from maze.core.wrappers.random_reset_wrapper import RandomResetWrapper
from maze.core.wrappers.time_limit_wrapper import TimeLimitWrapper

# instantiate the environment
env = ...

# apply wrappers
env = RandomResetWrapper.wrap(env, min_skip_steps=0, max_skip_steps=100)
env = TimeLimitWrapper.wrap(env, max_episode_steps=1000)
```

3.18.5 Built-in Wrappers

Maze already comes with built-in environment wrappers. You can find a list and further details on the functionality of the respective wrappers in the *reference documentation*.

For the following wrappers we also provide a more extensive documentation:

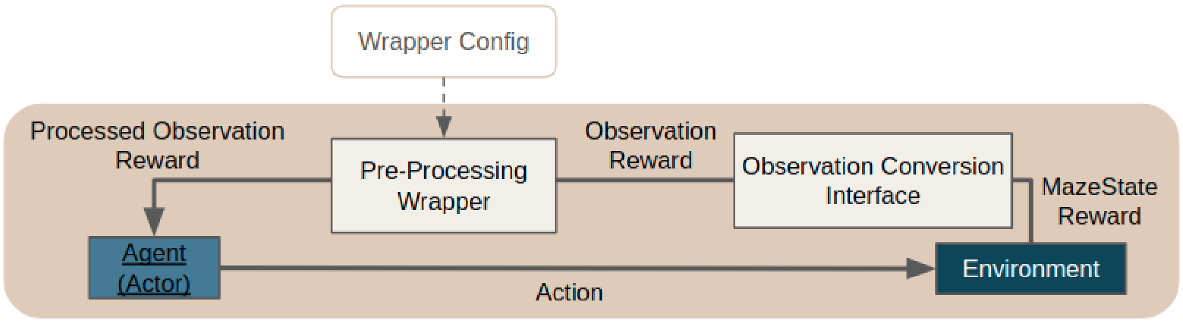
- *Observation Pre-Processing*
- *Observation Normalization*
- *Observation Logging*

3.18.6 Where to Go Next

- For further details please see the *reference documentation*.
- Special wrapper for *observation pre-processing* and *observation normalization*.
- You might also want to read up on the *Maze environment hierarchy*.

3.19 Observation Pre-Processing

Sometimes it is required to pre-process or modify observations before passing them through our policy or value networks. This might be for example the conversion of an three channel RGB image to a single channel grayscale image or the one-hot encoding of a categorical observation such as the current month into a feature vector of length 12. Maze supports observation pre-processing via the *PreProcessingWrapper*.



This means to gain access to observation pre-processing and to the features listed below you simply have to add the *PreProcessingWrapper* to your wrapper stack in your Hydra configuration.

To get started you can also just copy one of our *Hydra config snippets* or *use it directly from Python*.

3.19.1 List of Features

Maze observation pre-processing supports:

- Gym dictionary observation spaces
- *Individual pre-processors* for all sub-observations of these dictionary spaces
- *Cascaded pre-processing pipelines* for a single observation (e.g. first convert an image to grayscale before inserting an additional dimension from the left for CNN processing)
- The option to keep both, the original as well as the pre-processed observation
- Implicit update of affected observation spaces according to the pre-processor functionality

3.19.2 Example 1: Observation Specific Pre-Processors

This example adds pre-processing to two observations (*rgb_image* and *categorical_feature*) contained in a dictionary observation space.

```yaml
# @package wrappers
PreProcessingWrapper:
    pre_processor_mapping:
        - observation: rgb_image
          type: maze.preprocessors.Rgb2GrayPreProcessor
          keep_original: true
          config:
            num_flatten_dims: 2
        - observation: categorical_feature
          type: maze.preprocessors.OneHotPreProcessor
          keep_original: false
          config: {}
```

Details:

- Adds a gray scale converted version of observation *rgb_image* to the observation space but also keeps the original observation.
- Replaces the observation *categorical_feature* with an one-hot encoded version and drops the original observation.
- Observations space after pre-processing: {*rgb_image*, *rgb_image-rgb2gray*, *categorical_feature-one_hot*})

3.19.3 Example 2: Cascaded Pre-Processing

This example shows how to apply multiple pre-processor in sequence to a single observation.

```yaml
# @package wrappers
PreProcessingWrapper:
  pre_processor_mapping:
    - observation: rgb_image
      type: maze.preprocessors.Rgb2GrayPreProcessor
      keep_original: false
      config:
        rgb_dim: -1
    - observation: rgb_image-rgb2gray
      type: maze.preprocessors.ResizeImgPreProcessor
      keep_original: false
      config:
        target_size: [96, 96]
        transpose: false
    - observation: rgb_image-rgb2gray-resize_img
      type: maze.preprocessors.UnSqueezePreProcessor
      keep_original: false
      config:
        dim: -3
```

Details:

- Converts observation *rgb_image* into a gray scale image, then scales this gray scale image to size 96 x 96 pixel and finally inserts an additional dimension at index -3 to prepare the observation for CNN processing.
- None of the intermediate observations is kept as we are only interested in the final result here.
- Observations space after pre-processing: {*rgb_image-rgb2gray-resize_img*}).

3.19.4 Example 3: Using Custom Pre-Processors

In case the built-in pre-processors provided with Maze are not sufficient for your use case you can of course implement and add additional custom processors.

```yaml
# @package wrappers
PreProcessingWrapper:
    pre_processor_mapping:
        - observation: rgb_image
          type: my_project.preprocessors.custom.CustomPreProcessor
          keep_original: true
          config:
            num_flatten_dims: 2
```

When adding a new pre-processor you (1) have to implement the **PreProcessor** interface and (2) make sure that it is accessible within your Python path. Besides that you only have to provide the reference path of the pre-processor to use.

Observations will be tagged with the filename of your custom preprocessor (e.g. *rgb_image* -> *rgb_image-custom*).

3.19.5 Example 4: Plain Python Configuration

If you are not working with the Maze command line interface but still want to reuse observation pre-processing directly within Python you can start with the code snippet below.

```
"""Contains an example showing how to use observation pre-processing directly from␣
↪python."""
from maze.core.wrappers.maze_gym_env_wrapper import GymMazeEnv
from maze.core.wrappers.observation_preprocessing.preprocessing_wrapper import␣
↪PreProcessingWrapper

# this is the pre-processor config as a python dict
config = {
    "pre_processor_mapping": [
        {"observation": "observation",
         "type": "maze.preprocessors.Rgb2GrayPreProcessor",
         "keep_original": False,
         "config": {"rgb_dim": -1}},
    ]
}

# instantiate a maze environment
env = GymMazeEnv("CarRacing-v0")

# wrap the environment for observation pre-processing
env = PreProcessingWrapper.wrap(env, pre_processor_mapping=config["pre_processor_
↪mapping"])

# after this step the training env yields pre-processed observations
pre_processed_obs = env.reset()
```

3.19.6 Built-in Pre-Processors

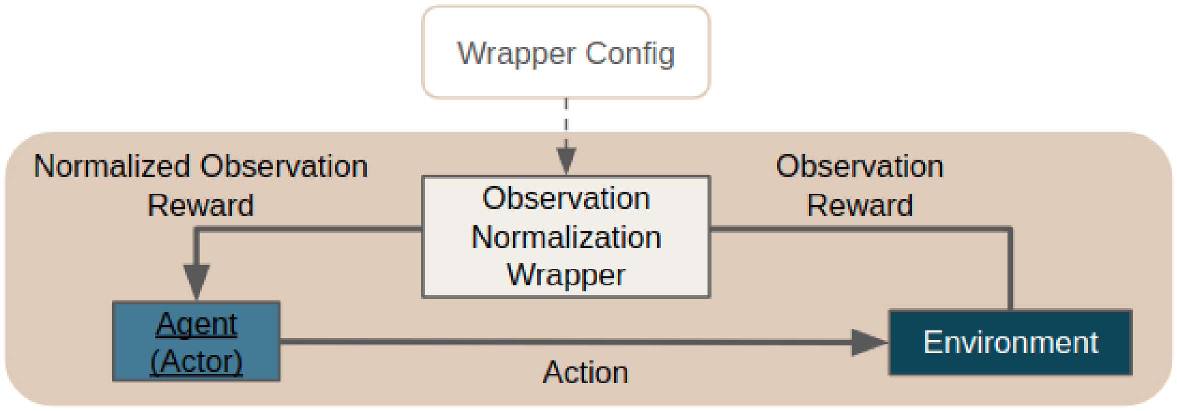
Maze already provides built-in pre-processors. You can find a list and further details on the functionality of the respective processors in the *reference documentation*.

3.19.7 Where to Go Next

- After pre-processing your observations you might also want to normalize them for efficient neural network processing using the *ObservationNormalizationWrapper*.
- Learn about more general *environment wrappers*.

3.20 Observation Normalization

For efficient RL training it is crucial that the inputs (e.g. observations) to our models (e.g. policy and value networks) follow a certain distribution and exhibit values living within a certain range. To ensure this precondition Maze provides general and customizable functionality for normalizing the observations returned by the respective environments via the *ObservationNormalizationWrapper*.



This means to gain access to observation normalization and to the features listed below you simply have to add the *ObservationNormalizationWrapper* to your wrapper stack in your Hydra configuration.

To get started you can also just copy one of our *Hydra config snippets* below or *use it directly within Python*.

3.20.1 List of Features

So far observation normalization supports:

- Different *normalization strategies* ([*mean-zero-std-one*, *range[0, 1]*, . . .)
- *Estimating normalization statistics from observations* collected by interacting with the respective environment (prior to training)
- Providing an action sampling policy for collecting these normalization statistics
- *Manually specification* of normalization statistics in case they are know beforehand
- *Excluding observations* such as action masks from normalization
- Preserving these statistics for continuing a training run, subsequent rollouts or deploying an agent
- Gym dictionary observation spaces
- Extendability with *custom observation normalization strategies* on the fly

As not all of the features listed above might be required right from the beginning you can find Hydra config examples with increasing complexity below.

3.20.2 Example 1: Normalization with Estimated Statistics

This example applies default observation normalization to all observations with statistics automatically estimated via sampling.

```
# @package wrappers
ObservationNormalizationWrapper:
    # default behaviour
    default_strategy: maze.normalization_strategies.
↪MeanZeroStdOneObservationNormalizationStrategy
    default_strategy_config:
        clip_range: [~, ~]
        axis: ~
    default_statistics: ~
```

(continues on next page)

(continued from previous page)

```
    statistics_dump: statistics.pkl
    sampling_policy:
        type: maze.core.agent.random_policy.RandomPolicy
    exclude: ~
    manual_config: ~
```

Details:

- Applies *mean zero - standard deviation one normalization* **to all observations** contained in the dictionary observation space
- Does not clip observations after normalization
- Does not compute individual normalization statistics along different axis of the observation vector / matrix
- Dumps the normalization statistics to the file "*statistics.pkl*"
- Estimates the required statistics from observations collected via random sampling
- Does not exclude any observations from normalization
- Does not provide any normalization statistics manually

3.20.3 Example 2: Normalization with Manual Statistics

In this example, we manually specify both the default normalization strategy and its corresponding default statistics. This is useful, e.g., when working with RGB pixel observation spaces. However, it requires to know the normalization statistics beforehand.

```
# @package wrappers
ObservationNormalizationWrapper:
    # default behaviour
    default_strategy: maze.normalization_strategies.
↪RangeZeroOneObservationNormalizationStrategy
    default_strategy_config:
        clip_range: [0, 1]
        axis: ~
    default_statistics:
        min: 0
        max: 255
    statistics_dump: statistics.pkl
    sampling_policy:
        type: maze.core.agent.random_policy.RandomPolicy
    exclude: ~
    manual_config: ~
```

Details:

- Add *range-zero-one* normalization with manually set statistics to all observations
- Clips the normalized observation to range [0, 1] in case something goes wrong. (As this example expects RGB pixel observations to have values between 0 and 255 this should not have an effect.)
- Subtracts 0 from each value contained in the observation vector / matrix and then divides it by 255
- The remaining settings do not have an effect here

3.20.4 Example 3: Custom Normalization and Excluding Observations

This advanced example shows how to utilize the full feature set of observation normalization. For explanations please see the comments and details below.

```
# @package wrappers
ObservationNormalizationWrapper:
    # default behaviour
    default_strategy: maze.normalization_strategies.
↪MeanZeroStdOneObservationNormalizationStrategy
    default_strategy_config:
        clip_range: [~, ~]
        axis: ~
    default_statistics: ~
    statistics_dump: statistics.pkl
    sampling_policy:
        type: maze.core.agent.random_policy.RandomPolicy
    # observation with key action_mask gets excluded from normalization
    exclude: [action_mask]
    manual_config:
        # observation pixel_image uses manually specified normalization statistics
        pixel_image:
          strategy: maze.normalization_strategies.
↪RangeZeroOneObservationNormalizationStrategy
          strategy_config:
            clip_range: [0, 1]
            axis: ~
          statistics:
            min: 0
            max: 255
        # observation feature_vector estimates normalization statistics via sampling
        feature_vector:
          strategy: maze.normalization_strategies.
↪MeanZeroStdOneObservationNormalizationStrategy
          strategy_config:
            clip_range: [-3, 3]
            # normalization statistics are computed along the first axis
            axis: [0]
```

Details:

- The default behaviour for observations without manual config is identical to example 1
- observation pixel_image: behaves as the default in example 2
- observation feature_vector:
  - By setting axis to [0] in the *strategy_config* each element in the observation gets normalized with an element-wise mean and standard deviation.
  - Why? A feature_vector has shape (d,). After collecting N observations for computing the normalization statistics we arrive at a stacked feature_vector-matrix with shape (N, 10). By computing the normalization statistics along axis [0] we get normalization statistics with shape (d,) again which can be applied in an elementwise fashion.
  - Additionally each element in the vector is clipped to range [-3, 3].
- **Note**, that even though a manual config is provided for some observations you can still decide if you would like to use predefined manual statistics or estimate them from sampled observations.

3.20.5 Example 4: Using Custom Normalization Strategies

In case the normalization strategies provided with Maze are not sufficient for your use case you can of course implement and add your own strategies.

```
# @package wrappers
ObservationNormalizationWrapper:
    # default behaviour
    default_strategy: my_project.normalization_strategies.custom.
↪CustomObservationNormalizationStrategy
    default_strategy_config:
        clip_range: [~, ~]
        axis: ~
    default_statistics: ~
    statistics_dump: statistics.pkl
    sampling_policy:
        type: maze.core.agent.random_policy.RandomPolicy
    exclude: ~
    manual_config: ~
```

When adding a new normalization strategy you (1) have to implement the **ObservationNormalizationStrategy** interface and (2) make sure that it is accessible within your Python path. Besides that you only have to provide the reference path of the pre-processor to use.

3.20.6 Example 5: Plain Python Configuration

If you are not working with the Maze command line interface but still want to reuse observation normalization directly within Python you can start with the code snippet below. It shows how to:

- instantiate an observation normalized environment
- estimate normalization statistics via sampling
- reuse the estimated statistics for normalization for subsequent tasks such as training or rollouts

```
"""Contains an example showing how to use observation normalization directly from␣
↪python."""
from maze.core.agent.random_policy import RandomPolicy
from maze.core.wrappers.maze_gym_env_wrapper import GymMazeEnv
from maze.core.wrappers.observation_normalization.observation_normalization_wrapper␣
↪import \
    ObservationNormalizationWrapper
from maze.core.wrappers.observation_normalization.observation_normalization_utils␣
↪import \
    obtain_normalization_statistics

# instantiate a maze environment
env = GymMazeEnv("CartPole-v0")

# this is the normalization config as a python dict
normalization_config = {
    "default_strategy": "maze.normalization_strategies.
↪MeanZeroStdOneObservationNormalizationStrategy",
    "default_strategy_config": {"clip_range": (None, None), "axis": 0},
    "default_statistics": None,
    "statistics_dump": "statistics.pkl",
    "sampling_policy": RandomPolicy(env.action_spaces_dict),
```

(continues on next page)

(continued from previous page)

```
    "exclude": None,
    "manual_config": None
}

# 1. PREPARATION: first we estimate normalization statistics
# ----------------------------------------------------------

# wrap the environment for observation normalization
env = ObservationNormalizationWrapper.wrap(env, **normalization_config)

# before we can start working with normalized observations
# we need to estimate the normalization statistics
normalization_statistics = obtain_normalization_statistics(env, n_samples=1000)

# 2. APPLICATION (training, rollout, deployment)
# ----------------------------------------------

# instantiate a maze environment
training_env = GymMazeEnv("CartPole-v0")
# wrap the environment for observation normalization
training_env = ObservationNormalizationWrapper.wrap(training_env, **normalization_
↪config)

# reuse the estimated the statistics in our training environment(s)
training_env.set_normalization_statistics(normalization_statistics)

# after this step the training env yields normalized observations
normalized_obs = training_env.reset()
```

3.20.7 Built-in Normalization Strategies

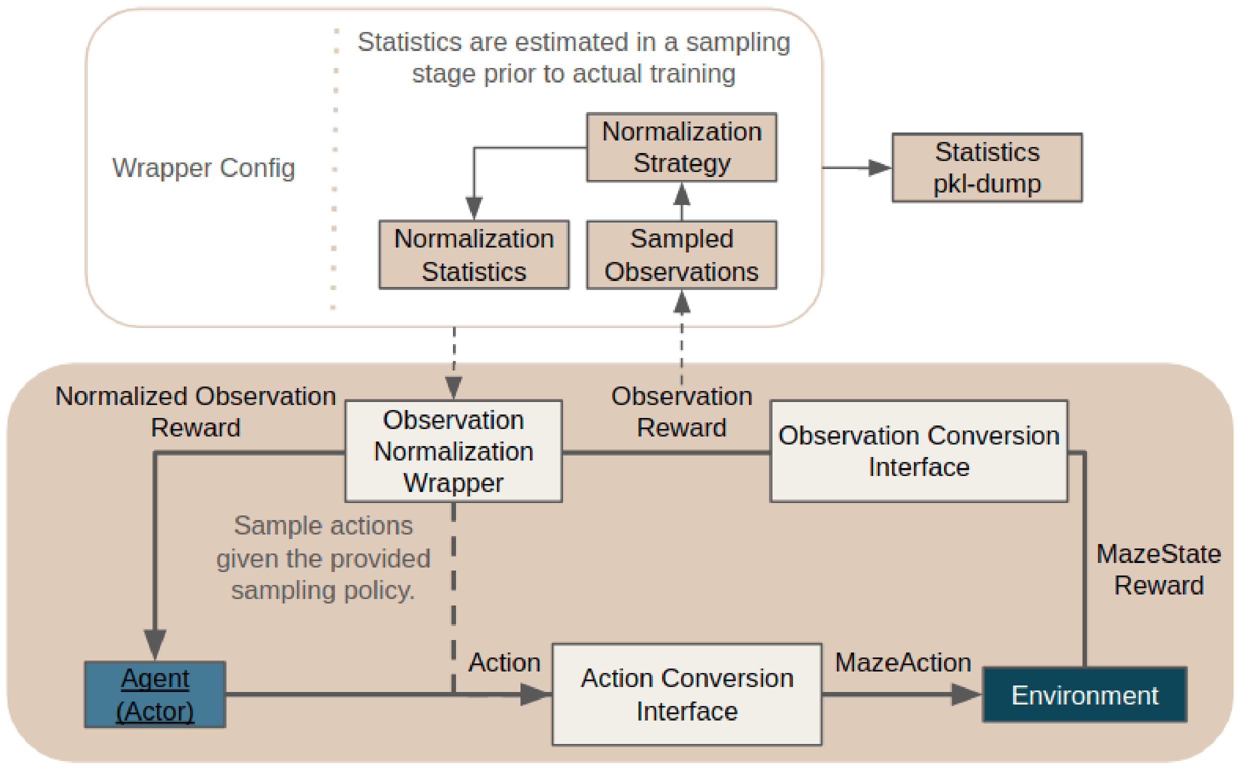
Normalization strategies simply specify the way how input observations are normalized.

Maze already comes with built-in normalization strategies. You can find a list and further details on the functionality of the respective strategies in the *reference documentation*.

3.20.8 The Bigger Picture

The figure below shows how observation normalization is embedded in the overall interaction loop and set the involved components into context.

It is located in between the `ObservationConversionInterface` (which converts environment MazeStates into machine readable observations) and the agent.


Statistics are estimated in a sampling stage prior to actual training
Wrapper Config
Normalization Strategy
Normalization Statistics
Sampled Observations
Statistics pkl-dump
Normalized Observation Reward
Observation Normalization Wrapper
Observation Reward
Observation Conversion Interface
Sample actions given the provided sampling policy.
MazeState Reward
Agent (Actor)
Action
Action Conversion Interface
MazeAction
Environment

According to the *sampling_policy* specified in the config the wrapper collects observations from the interaction loop and uses these to estimate the normalization statistics given the provided normalization strategies.

The statistics get dumped to the pickle file specified in the config for subsequent rollouts or deploying the agent.

If normalization statistics are known beforehand this stage can be skipped by simply providing the statistics manually in the wrapper config.

3.20.9 Where to Go Next

- Before normalizing your observations you first might want to pre-process them with the *PreProcessingWrapper*.
- Learn about more general *environment wrappers*.

3.21 Tricks of the Trade

This page contains a short list of tips and best practices that have been quite useful in our work over the last couple of years and will hopefully also make it easier for you to train your agents. However, you should be aware that not each item below will work in each and every application scenario. Nonetheless, if you are stuck most of them are certainly worth to give a try!

Note: Below you find a subjective and certainly not complete collection of RL tips and tricks that will hopefully continue to grow over time. However, if you stumble upon something crucial that is missing from the list, which you would like to share with the RL community and us do not hesitate to get in touch and discuss with us!

Overview

- *Learning and Optimization*
- *Models and Networks*
- *Observations*

3.21.1 Learning and Optimization

✓ **Action Masking**

Use action masking whenever possible! This can be crucial as it has the potential to drastically reduce the exploration space of your problem, which usually leads to a reduced learning time and better overall results. In some cases action masking also mitigates the need for reward shaping as invalid actions are excluded from sampling and there is no need to penalize them with negative rewards any more. If you want to learn more we recommend to check out the tutorial on *structured environments and action masking*.

✓ **Reward Scaling and Shaping**

Make sure that your step rewards are in a reasonable range (e.g., [-1, 1]) not spanning various orders of magnitude. If these conditions are not fulfilled you might want to apply reward scaling or clipping (see `RewardScalingWrapper`, `RewardClippingWrapper`) or *manually shape your reward*.

✓ **Reward and Key Performance Indicator (KPI) Monitoring**

When optimizing multi-target objectives (e.g., a weighted sum of sub-rewards) consider to monitor the contributing rewards on an individual basis. Event though the overall reward appears to not improve anymore it might still be the case that the contributing sub-rewards change or fluctuate in the background. This indicates that the policy and in turn the behaviour of your agent is still changing. In such settings we recommend to watch the learning progress by *monitoring KPIs*.

3.21.2 Models and Networks

✓ **Network Design**

Design use case and task specific custom network architectures whenever required. In a straight forward case this might be a CNN when processing image observations but it could also be a Graph Convolution Network (GCN) when working with graph or grid observations. To do so, you might want to check out the *Perception Module*, the built-in *network building blocks* as well as the section on *how to work with custom models*.

Further, you might want to consider *behavioural cloning (BC)* to design and tweak

- the network architectures
- the observations that are fed into these models

This requires that an imitation learning dataset fulfilling the pre-conditions for supervised learning is available. If so, incorporating BC into the model and observation design process can save a lot of time and compute as you are now training in a supervised learning setting. *Intuition*: If a network architecture, given the corresponding observations, is able to fit an offline trajectory dataset (without severe over-fitting) it might also be a good choice for actual RL training. If this is relevant to you, you can follow up on how to *employ imitation learning with Maze*.

✓ **Continuous Action Spaces**

When facing bounded continuous action spaces use `Squashed Gaussian` or `Beta` probability distributions for your action heads instead of an unbounded Gaussian. This avoids action clipping and limits the space of explorable actions to valid regions. You can learn in the section about *distributions and acton heads* how you can easily switch between different probability distributions using the `DistributionMapper`.

✓ **Action Head Biasing**

If you would like to incorporate prior knowledge about the selection frequency of certain actions you could consider to bias the output layers of these action heads towards the expected sampling distribution after randomly initializing the weights of your networks (e.g., `compute_sigmoid_bias`).

3.21.3 Observations

✓ **Observation Normalization**

For efficient RL training it is crucial that the inputs (e.g. observations) to our models (e.g. policy and value networks) follow a certain distribution and exhibit values within certain ranges. To ensure this precondition consider to normalize your observations before actual training by either:

- manually specifying normalization statistics (e.g, divide by 255 for uint8 RGB image observations)
- compute statistics from observations sampled by interacting with the environment

As this is a recurring, boilerplate code heavy task, Maze already provides *built-in customizable functionality for normalizing the observations*.

✓ **Observation Pre-Processing**

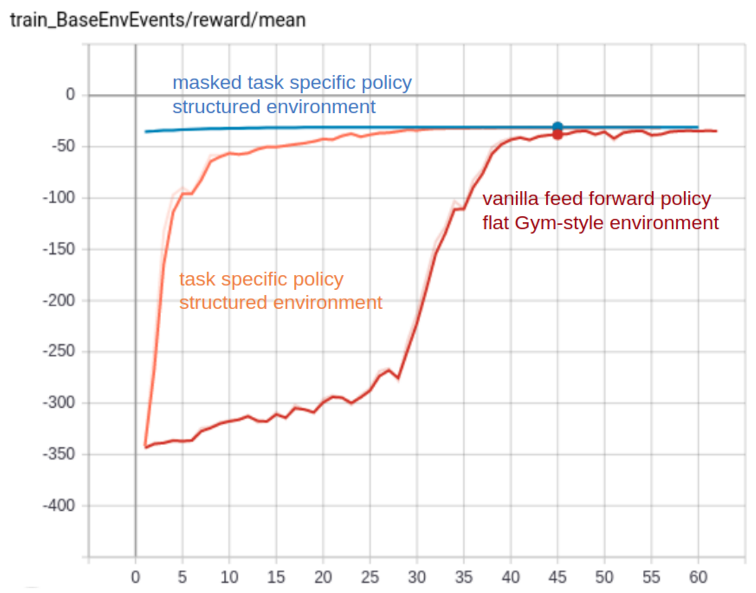
When feeding categorical observations to your models consider to convert them to their one-hot encoded vectorized counterparts. This representation is better suited for neural network processing and a common practice for example in Natural Language Processing (NLP). In Maze you can achieve this via *observation pre-processing* and the `OneHotPreProcessor`.

3.22 Structured Environments and Action Masking

This tutorial provides a step by step guide explaining how to implement a decision problem as a structured environment and how to train an agent for such a `StructuredEnv` with a structured *Maze Trainer*. The examples are again based on the online version of the *Guillotine 2D Cutting Stock Problem* which is a perfect fit for introducing the underlying concepts.

In particular, we will see how to evolve the performance of an RL agent by going through the following stages:

1. Flat Gym-style environment with vanilla feed forward models
2. Structured environment (e.g., with hierarchical sub-steps) with task specific policy networks
3. Structured environment (e.g., with hierarchical sub-steps) with masked task specific policy networks



Before diving into this tutorial we recommend to familiarize yourself with the basic *Maze - step by step tutorial*.

The remainder of this tutorial is structured as follows:

3.22.1 Turning a "flat" MazeEnv into a StructuredEnv

In this part of the tutorial we will learn how to reformulate an RL problem in order to turn it from a "flat" Gym-style environment into a structured environment.

The complete code for this part of the tutorial can be found here

```
# relevant files
- cutting_2d
    - main.py
    - env
        - struct_env.py
```

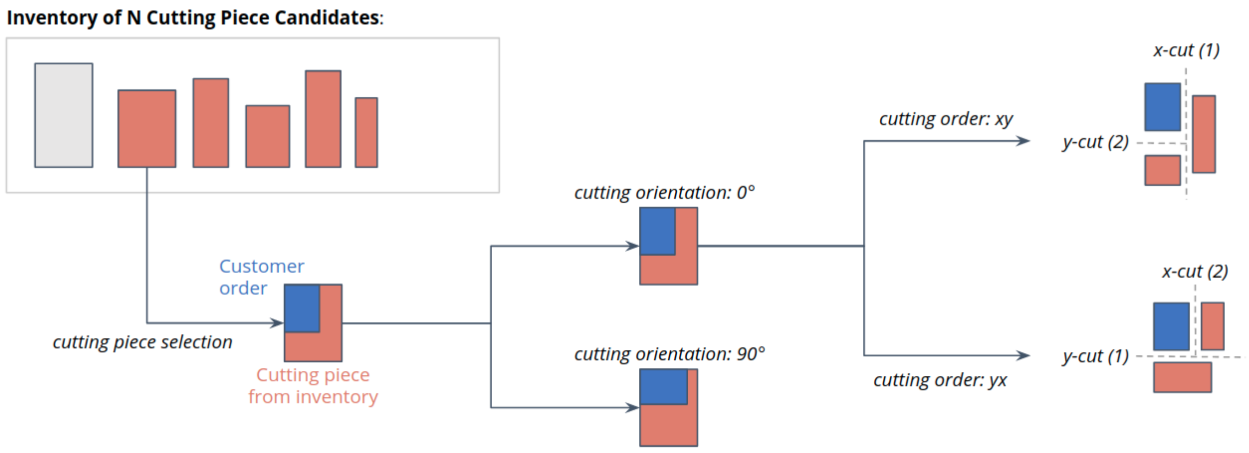
Page Overview

- *Analyzing the Problem Structure*
- *Implementing the Structured Environment*
- *Test Script*

Analyzing the Problem Structure

Before we start implementing the structured environment lets first *revisit the cutting 2D problem*. In particular, we put our attention to the joint action space consisting of the following components:

- Action a_0: cutting piece selection (decides which piece from inventory to use for cutting)
- Action a_1: cutting orientation selection (decides on the orientation of the cut)
- Action a_2: cutting order selection (decides which cut to take first; x or y)



Analysis of Action Space and Problem:

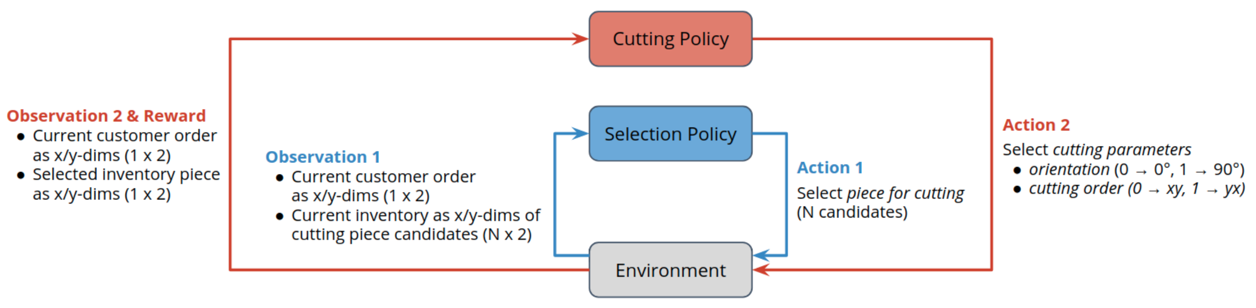
- We are facing a combinatorial action space with $O(N \cdot 2 \cdot 2)$ possible actions the agent has to choose from in each step. N is the maximum number of pieces stored in the inventory.
- Sampling from this joint action space might result in invalid cutting configurations. This is because the three sub-actions are treated independently from each other. For the problem at hand this is obviously not the case.
- It would be much more intuitive to sample the sub-actions sequentially and conditioned on each other. (E.g., it seems to be easier to decide on the cutting order and orientation once we know the piece we will cut from.)

Implementing the Structured Environment

We now address the issues discovered in the previous section and re-formulate the cutting 2D problem as a `StructuredEnv` with the following two sub-steps:

- **Select cutting piece** from inventory given inventory state and customer order.
- Select cutting configuration (**cutting order** and **cutting orientation**) given customer order and inventory cutting piece selected in the previous sub-step.

This could be also described with the modified agent environment interaction loop shown in the figure below. Note that the both observation and action space differ between the selection and the cutting sub-step. For the present example, reward is only granted once the cutting sub-step (i.e., the second step) is complete.


Cutting Policy
Selection Policy
Environment
Observation 2 & Reward
• Current customer order as x/y-dims (1 x 2)
• Selected inventory piece as x/y-dims (1 x 2)
Observation 1
• Current customer order as x/y-dims (1 x 2)
• Current inventory as x/y-dims of cutting piece candidates (N x 2)
Action 1
Select piece for cutting (N candidates)
Action 2
Select cutting parameters
• orientation (0 → 0°, 1 → 90°)
• cutting order (0 → xy, 1 → yx)

Note: Conceptually structured environments and conditional sub-steps are related to auto-regressive action spaces where subsequent actions are sampled conditioned on their predecessors. [e.g. DeepMind (2019), "Grandmaster level in StarCraft II using multi-agent reinforcement learning."]

The code for the `StructuredCutting2DEnvironment` below implements exactly this interaction pattern.

Listing 1: env/struct_env.py

```
from copy import deepcopy
from typing import Dict, Any, Union, Tuple, Optional, List

import gym
import numpy as np
from maze.core.env.maze_action import MazeActionType
from maze.core.env.maze_env import MazeEnv
from maze.core.env.maze_state import MazeStateType
from maze.core.env.structured_env import StructuredEnv
from maze.core.env.structured_env_spaces_mixin import StructuredEnvSpacesMixin
from maze.core.wrappers.wrapper import Wrapper
from .maze_env import maze_env_factory


class StructuredCutting2DEnvironment(Wrapper[MazeEnv], StructuredEnv,␣
↪StructuredEnvSpacesMixin):
    """Structured environment version of the cutting 2D environment.
    The environment alternates between the two sub-steps:

    - Select cutting piece
    - Select cutting configuration (cutting order and cutting orientation)

    :param maze_env: The "flat" cutting 2D environment to wrap.
    """

    def __init__(self, maze_env: MazeEnv):
        Wrapper.__init__(self, maze_env)

        # define sub-step action spaces
        self._action_spaces_dict = {
            0: gym.spaces.Dict({"piece_idx": maze_env.action_space["piece_idx"]}),
            1: gym.spaces.Dict({"cut_rotation": maze_env.action_space["cut_rotation"],
                                "cut_order": maze_env.action_space["cut_order"]})
        }

        # define sub-step observation spaces
        flat_space = maze_env.observation_space
```

(continues on next page)

(continued from previous page)

```
        self._observation_spaces_dict = {
            0: flat_space,
            1: gym.spaces.Dict({"selected_piece": flat_space["ordered_piece"],
                                "ordered_piece": flat_space["ordered_piece"]})
        }

        self._flat_obs = None
        self._action_0 = None
        self._sub_step_key = 0

    def step(self, action):
        """Generic step function alternating between the two sub-steps.
        :return: obs, rew, done, info
        """
        # sub-step: Select cutting piece
        if self._sub_step_key == 0:
            sub_step_result = self._selection_step(action)
        # sub-step: Select cutting configuration
        elif self._sub_step_key == 1:
            sub_step_result = self._cutting_step(action)
        else:
            raise ValueError("Sub-step id {} not allowed for this environment!".
↪format(self._sub_step_key))

        # alternate step index
        self._sub_step_key = np.mod(self._sub_step_key + 1, 2)

        return sub_step_result

    def reset(self) -> Any:
        """Resets the environment and returns the initial state.
        :return: The initial state after resetting.
        """
        self._flat_obs = self.env.reset()
        self._flat_obs["ordered_piece"] = self._flat_obs["ordered_piece"]

        self._sub_step_key = 0
        return self._obs_selection_step(self._flat_obs)

    @staticmethod
    def _obs_selection_step(flat_obs: Dict[str, np.array]) -> Dict[str, np.array]:
        """Formats initial observation / observation available for the first sub-step.
↪"""
        return deepcopy(flat_obs)

    @staticmethod
    def _obs_cutting_step(flat_obs: Dict[str, np.array], selected_piece_idx: int) ->␣
↪Dict[str, np.array]:
        """Formats observation available for the second sub-step."""
        return {"selected_piece": flat_obs["inventory"][selected_piece_idx],
                "ordered_piece": flat_obs["ordered_piece"]}

    def _selection_step(self, action: Dict[str, int]) -> Tuple[Dict[str, np.ndarray],␣
↪float, bool, Dict]:
        """Cutting piece selection step."""
        self._action_0 = action
        obs = self._obs_cutting_step(self._flat_obs, action["piece_idx"])
```

(continues on next page)

(continued from previous page)

```
        return obs, 0.0, False, {}

    def _cutting_step(self, action: Dict[str, int]) -> Tuple[Dict[str, np.ndarray],␣
↪float, bool, Dict]:
        """Cutting rotation and cutting order selection step."""
        flat_action = {"piece_idx": self._action_0["piece_idx"],
                       "cut_rotation": action["cut_rotation"],
                       "cut_order": action["cut_order"]}

        self._flat_obs, rew, done, info = self.env.step(flat_action)
        self._flat_obs["ordered_piece"] = self._flat_obs["ordered_piece"]

        return self._obs_selection_step(self._flat_obs), rew, done, info

    def actor_id(self) -> Tuple[Union[str, int], int]:
        """Returns the currently executed actor along with the policy id. The id is␣
↪unique only with
        respect to the policies (every policy has its own actor 0).
        Note that identities of done actors can not be reused in the same rollout.

        :return: The current actor, as tuple (policy id, actor number).
        """
        return self._sub_step_key, 0

    def is_actor_done(self) -> bool:
        """Returns True if the just stepped actor is done, which is different to the␣
↪done flag of the environment."""
        return False

    @property
    def action_space(self) -> gym.spaces.Dict:
        """Implementation of :class:`~maze.core.env.structured_env_spaces_mixin.
↪StructuredEnvSpacesMixin` interface."""
        return self._action_spaces_dict[self._sub_step_key]

    @property
    def observation_space(self) -> gym.spaces.Dict:
        """Implementation of :class:`~maze.core.env.structured_env_spaces_mixin.
↪StructuredEnvSpacesMixin` interface."""
        return self._observation_spaces_dict[self._sub_step_key]

    @property
    def action_spaces_dict(self) -> Dict[Union[int, str], gym.spaces.Dict]:
        """Implementation of :class:`~maze.core.env.structured_env_spaces_mixin.
↪StructuredEnvSpacesMixin` interface."""
        return self._action_spaces_dict

    @property
    def observation_spaces_dict(self) -> Dict[Union[int, str], gym.spaces.Dict]:
        """Implementation of :class:`~maze.core.env.structured_env_spaces_mixin.
↪StructuredEnvSpacesMixin` interface."""
        return self._observation_spaces_dict

    def seed(self, seed: int = None) -> None:
        """Sets the seed for this environment's random number generator(s).
        :param: seed: the seed integer initializing the random number generator.
        """
```

(continues on next page)

(continued from previous page)

```
        self.env.seed(seed)

    def close(self) -> None:
        """Performs any necessary cleanup."""
        self.env.close()

    def get_observation_and_action_dicts(self, maze_state: MazeStateType, maze_
↪action: MazeActionType,
                                         first_step_in_episode: bool) \
            -> Tuple[Optional[Dict[Union[int, str], Any]], Optional[Dict[Union[int,␣
↪str], Any]]]:
        """Convert the flat action and MazeAction from Maze env into the structured␣
↪ones.

        Note that both MazeState and MazeAction needs to be supplied together,␣
↪otherwise actions/observations for the
        individual sub-steps cannot be produced.
        """
        assert maze_state is not None and maze_action is not None,\
            "This wrapper needs both MazeState and MazeAction for the conversion (as␣
↪there are multiple sub-steps)."
        observation_dict, action_dict = self.env.get_observation_and_action_
↪dicts(maze_state, maze_action,
                                                                                   ␣
↪first_step_in_episode)
        assert len(observation_dict.items()) == 1 and len(action_dict.items()) == 1,
↪"wrapped env should be single-step"

        flat_action = list(action_dict.values())[0]
        flat_obs = list(observation_dict.values())[0]

        flat_obs["ordered_piece"] = flat_obs["ordered_piece"]

        obs_dict = {
            0: self._obs_selection_step(flat_obs),
            1: self._obs_cutting_step(flat_obs, flat_action["piece_idx"])
        }

        act_dict = {
            0: {k: flat_action[k] for k in ["piece_idx"]},
            1: {k: flat_action[k] for k in ["cut_rotation", "cut_order"]}
        }

        return obs_dict, act_dict


def struct_env_factory(max_pieces_in_inventory: int, raw_piece_size: Tuple[int, int],
                       static_demand: List[Tuple[int, int]]) ->␣
↪StructuredCutting2DEnvironment:
    """Convenience factory function that compiles a trainable structured environment.
    (for argument details see: Cutting2DEnvironment)
    """

    # init maze environment including observation and action interfaces
    env = maze_env_factory(max_pieces_in_inventory=max_pieces_in_inventory,
                           raw_piece_size=raw_piece_size,
                           static_demand=static_demand)
```

(continues on next page)

(continued from previous page)

```
    # convert flat to structured environment
    return StructuredCutting2DEnvironment(env)
```

Test Script

The following snippet first instantiates the structured environment and then performs one cycle of the structured agent environment interaction loop.

Listing 2: main.py

```
""" Test script CoreEnv """
from tutorials.tutorial_maze_env.part06_struct_env.env.struct_env import struct_env_
↪factory


def main():
    # init maze environment including observation and action interfaces
    struct_env = struct_env_factory(max_pieces_in_inventory=200,
                                    raw_piece_size=(100, 100),
                                    static_demand=[(30, 15)])

    # reset env
    obs_step1 = struct_env.reset()

    print("action_space 1:      ", struct_env.action_space)
    print("observation_space 1:", struct_env.observation_space)
    print("observation 1:       ", obs_step1.keys())

    # take first env step
    action_1 = struct_env.action_space.sample()
    obs_step2, rew, done, info = struct_env.step(action=action_1)

    print("action_space 2:      ", struct_env.action_space)
    print("observation_space 2:", struct_env.observation_space)
    print("observation 2:       ", obs_step2.keys())

    # take second env step
    action_2 = struct_env.action_space.sample()
    obs_step1 = struct_env.step(action=action_2)


if __name__ == "__main__":
    """ main """
    main()
```

Running the script will print the following output. Note that the observation and action spaces alternate from sub-step to sub-step.

```
action_space 1:      Dict(piece_idx:Discrete(200))
observation_space 1: Dict(inventory:Box(200, 2), inventory_size:Box(1,), order:Box(2,
↪))
observation 1:       dict_keys(['inventory', 'inventory_size', 'order'])
action_space 2:      Dict(order:Discrete(2), rotation:Discrete(2))
observation_space 2: Dict(order:Box(2,), selected_piece:Box(1, 2))
```

(continues on next page)

(continued from previous page)

```
observation 2:        dict_keys(['selected_piece', 'order'])
```

In the next part of this tutorial we will train an agent on this structured environment.

3.22.2 Training the Structured Environment

In this part of the tutorial we will learn how to train an agent with a *Maze trainer* implicitly supporting a Structured Environment. We will also design a policy network architecture matching the task at hand.

The complete code for this part of the tutorial can be found here

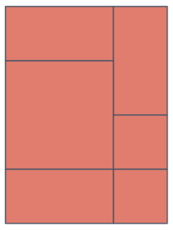
```
# relevant files
- cutting_2d
    - conf
        - env
            - tutorial_cutting_2d_flat.yaml
            - tutorial_cutting_2d_struct.yaml
        - model
            - tutorial_cutting_2d_flat.yaml
            - tutorial_cutting_2d_struct.yaml
        - wrappers
            - tutorial_cutting_2d.yaml
    - models
        - actor.py
        - critic.py
```

Page Overview

- *A Simple Problem Setting*
- *Task-Specific Actor-Critic Model*
- *Multi-Step Training*

A Simple Problem Setting

To emphasize the effects of action masking throughout this tutorial we devise a simple problem instance of the cutting 2d environment with the following properties:



Raw Piece Size:
[4, 3]

Static Demand:
[[1, 2], [1, 2], [1, 1], [2, 2], [1, 2], [1, 1]]

Episode length:
180 steps

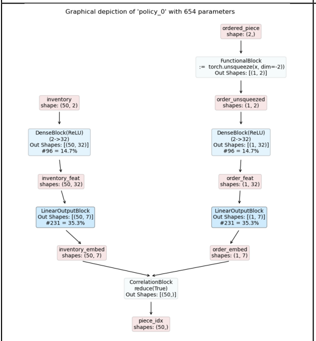
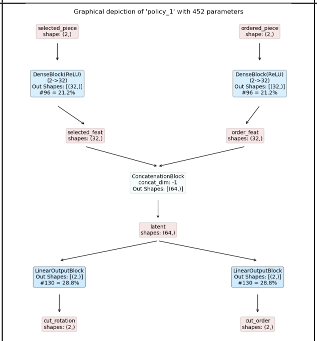
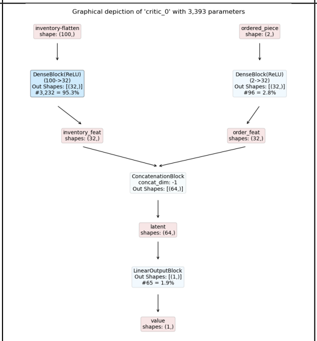
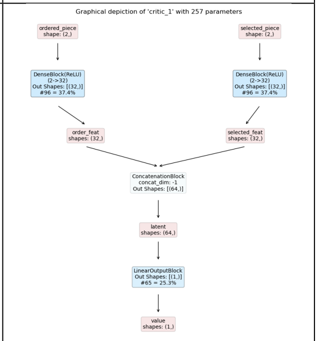
⇒ optimal: 30 + 1 pieces

Given the raw piece size and the items in the static demand (appear in an alternating fashion) we can cut six customer orders from one raw inventory piece. When limiting the episode length to 180 time steps the optimal solution with respect to new raw pieces from inventory is 31 (30 + 1 because the environment adds a new piece whenever the current one has been cut).

Task-Specific Actor-Critic Model

For this advanced tutorial we make use of Maze *custom models* to compose an actor-critic architecture that is geared towards the respective sub-tasks. Our structured environment requires two policies, one for piece selection and one for cutting parametrization. For each of the two sub-step policies we also build a distinct state critic (see `StepStateCriticComposer` for details). Note that it would be also possible to employ a `SharedStateCriticComposer` used to compute the advantages for both policies.

The images below show the for network architectures (click to view in large). For further details on how to build the models we refer to the accompanying repository and the section on *how to work with custom models*.

Piece Selection Policy	Cutting Policy	Piece Selection Critic	Cutting Critic
 Graphical depiction of 'policy_0' with 654 parameters ordered_piece shape: (2,) FunctionalBlock := torch.unsqueeze(x, dim=-2)) Out Shapes: [(1, 2)] inventory shape: (50, 2) order_unsqueezed shapes: (1, 2) DenseBlock(ReLU) (2->32) Out Shapes: [(50, 32)] #96 = 14.7% DenseBlock(ReLU) (2->32) Out Shapes: [(1, 32)] #96 = 14.7% inventory_feat shapes: (50, 32) order_feat shapes: (1, 32) LinearOutputBlock Out Shapes: [(50, 7)] #231 = 35.3% LinearOutputBlock Out Shapes: [(1, 7)] #231 = 35.3% inventory_embed shapes: (50, 7) order_embed shapes: (1, 7) CorrelationBlock reduce(True) Out Shapes: [(50,)] piece_idx shapes: (50,)	 Graphical depiction of 'policy_1' with 452 parameters selected_piece shape: (2,) ordered_piece shape: (2,) DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 21.2% DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 21.2% selected_feat shapes: (32,) order_feat shapes: (32,) ConcatenationBlock concat_dim: -1 Out Shapes: [(64,)] latent shapes: (64,) LinearOutputBlock Out Shapes: [(2,)] #130 = 28.8% LinearOutputBlock Out Shapes: [(2,)] #130 = 28.8% cut_rotation shapes: (2,) cut_order shapes: (2,)	 Graphical depiction of 'critic_0' with 3,393 parameters inventory-flatten shape: (100,) ordered_piece shape: (2,) DenseBlock(ReLU) (100->32) Out Shapes: [(32,)] #3,232 = 95.3% DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 2.8% inventory_feat shapes: (32,) order_feat shapes: (32,) ConcatenationBlock concat_dim: -1 Out Shapes: [(64,)] latent shapes: (64,) LinearOutputBlock Out Shapes: [(1,)] #65 = 1.9% value shapes: (1,)	 Graphical depiction of 'critic_1' with 257 parameters ordered_piece shape: (2,) selected_piece shape: (2,) DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 37.4% DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 37.4% order_feat shapes: (32,) selected_feat shapes: (32,) ConcatenationBlock concat_dim: -1 Out Shapes: [(64,)] latent shapes: (64,) LinearOutputBlock Out Shapes: [(1,)] #65 = 25.3% value shapes: (1,)

Some notes on the models:

- The selection policy takes the current *inventory* and the *ordered piece* as input and predicts a selection probability (*piece_idx*) for each inventory option.
- The cutting policy takes the *ordered piece* and the *selected piece* (previous step) as input and predicts *cutting rotation* and *cutting order*.
- The critic models have an analogous structure but predict the state-value instead of action logits.

Multi-Step Training

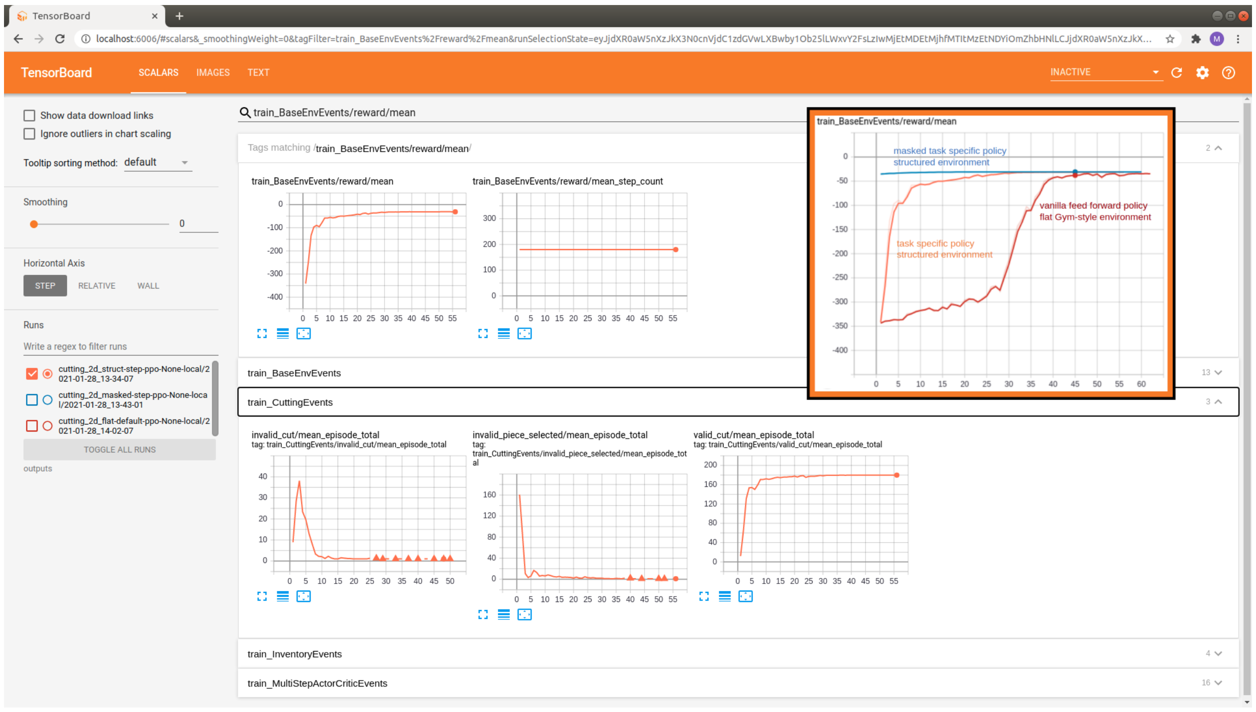
Given the models designed in the previous section we are now ready to train our first agent on a Structured Environment. We already mentioned that *Maze trainers* directly support the training of Structured Environments such as the `StructuredCutting2DEnvironment` implemented in the previous part of this tutorial.

To start training a cutting policy with the PPO trainer, run:

```
maze-run -cn conf_train env=tutorial_cutting_2d_struct wrappers=tutorial_cutting_2d \
model=tutorial_cutting_2d_struct algorithm=ppo
```

As usual, we can watch the training progress with Tensorboard.

```
tensorboard --logdir outputs
```


TensorBoard
SCALARS
IMAGES
TEXT
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0
Horizontal Axis
STEP
RELATIVE
WALL
Runs
cutting_2d_struct-step-ppo-None-local/2021-01-28_13-34-07
cutting_2d_masked-step-ppo-None-local/2021-01-28_13-43-01
cutting_2d_flat-default-ppo-None-local/2021-01-28_14-02-07
TOGGLE ALL RUNS
train_BaseEnvEvents/reward/mean
train_BaseEnvEvents/reward/mean_step_count
masked task specific policy structured environment
vanilla feed forward policy flat Gym-style environment
task specific policy structured environment
train_BaseEnvEvents
train_CuttingEvents
invalid_cut/mean_episode_total
invalid_piece_selected/mean_episode_total
valid_cut/mean_episode_total
train_InventoryEvents
train_MultiStepActorCriticEvents

We can see that the reward slowly approaches the optimum. Note that the performance of this agent is already much better than the vanilla Gym-style model we employed in the *introductory tutorial* (compare evolution of rewards above).

However, the event logs also reveal that the agent initially samples many invalid actions (e.g, *invalid_cut* and *invalid_piece_selected*). This is sample inefficient and slows down the learning progress.

Next, we will further improve the agent by avoiding sampling of these invalid choices via action masking.

3.22.3 Adding Step-Conditional Action Masking

In this part of the tutorial we will learn how to substantially increase the sample efficiency of our agents by adding *sub-step conditional action masking* to the structured environment.

The complete code for this part of the tutorial can be found here

```
# relevant files
- cutting_2d
    - main.py
    - env
        - struct_env_masked.py
```

Page Overview

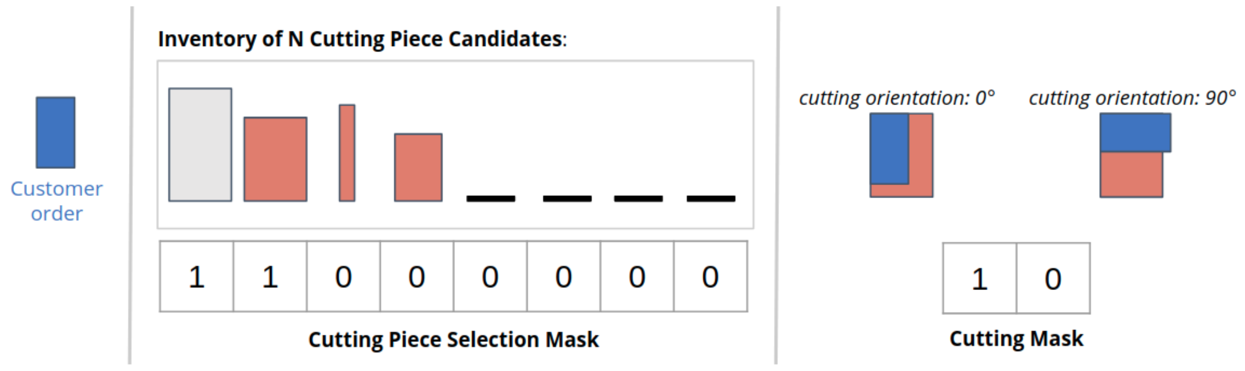
- *Masked Structured Environment*
- *Test Script*

In particular, we will add two different masks:

- **Inventory_mask**: allows to only select cutting pieces from inventory slots actually holding a piece that would allow to fulfill the customer order.

- **Rotation_mask**: allows to only specify valid cutting rotations (e.g., the ordered piece fits into the cutting piece from inventory). Note that providing this mask is only possible once the cutting piece has been selected in the first sub-step - hence the name *step-conditional masking*.

The figure below provides a sketch of the two masks.



Only the first two inventory pieces are able to fit the customer order. The four rightmost inventory slots do not hold a piece at all and are also masked. When rotating the piece by 90° for cutting the customer order would not fit into the selected inventory piece which is why we can simply mask this option.

Masked Structured Environment

One way to incorporate the two masks in our structured environment is to simply inherit from the initial version and extend it by the following changes:

- Add the two masks to the observation spaces (e.g., `inventory_mask` and `cutting_mask`)
- Compute the actual mask for the two sub-steps in the respective functions (e.g., `_obs_selection_step` and `_obs_cutting_step`).

Listing 3: env/struct_env_masked.py

```
from copy import deepcopy
from typing import Dict, List, Tuple

import gym
import numpy as np
from tutorials.tutorial_maze_env.part06_struct_env.env.maze_env import maze_env_
↪factory
from tutorials.tutorial_maze_env.part06_struct_env.env.struct_env import␣
↪StructuredCutting2DEnvironment
from maze.core.env.maze_env import MazeEnv


class MaskedStructuredCutting2DEnvironment(StructuredCutting2DEnvironment):
    """Structured environment version of the cutting 2D environment.
    The environment alternates between the two sub-steps:

    - Select cutting piece
    - Select cutting configuration (cutting order and cutting orientation)

    :param maze_env: The "flat" cutting 2D environment to wrap.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, maze_env: MazeEnv):
        super().__init__(maze_env)

        # add masks to observation spaces
        max_inventory = self.observation_conversion.max_pieces_in_inventory
        self._observation_spaces_dict[0].spaces["inventory_mask"] = \
            gym.spaces.Box(low=np.float32(0), high=np.float32(1), shape=(max_
↪inventory,), dtype=np.float32)

        self._observation_spaces_dict[1].spaces["cutting_mask"] = \
            gym.spaces.Box(low=np.float32(0), high=np.float32(1), shape=(2,),␣
↪dtype=np.float32)

    @staticmethod
    def _obs_selection_step(flat_obs: Dict[str, np.array]) -> Dict[str, np.array]:
        """Formats initial observation / observation available for the first sub-step.
↪"""
        observation = deepcopy(flat_obs)

        # prepare inventory mask
        sorted_order = np.sort(observation["ordered_piece"].flatten())
        sorted_inventory = np.sort(observation["inventory"], axis=1)

        observation["inventory_mask"] = np.all(observation["inventory"] > 0, axis=1).
↪astype(np.float32)
        for i in np.nonzero(observation["inventory_mask"])[0]:
            # exclude pieces which do not fit
            observation["inventory_mask"][i] = np.all(sorted_order <= sorted_
↪inventory[i])

        return observation

    @staticmethod
    def _obs_cutting_step(flat_obs: Dict[str, np.array], selected_piece_idx: int) ->␣
↪Dict[str, np.array]:
        """Formats observation available for the second sub-step."""

        selected_piece = flat_obs["inventory"][selected_piece_idx]
        ordered_piece = flat_obs["ordered_piece"]

        # prepare cutting action mask
        cutting_mask = np.zeros((2,), dtype=np.float32)

        selected_piece = selected_piece.squeeze()
        if np.all(flat_obs["ordered_piece"] <= selected_piece):
            cutting_mask[0] = 1.0

        if np.all(flat_obs["ordered_piece"][::-1] <= selected_piece):
            cutting_mask[1] = 1.0

        return {"selected_piece": selected_piece,
                "ordered_piece": ordered_piece,
                "cutting_mask": cutting_mask}


def struct_env_factory(max_pieces_in_inventory: int, raw_piece_size: Tuple[int, int],
                       static_demand: List[Tuple[int, int]]) ->␣
↪StructuredCutting2DEnvironment:
```

(continues on next page)

(continued from previous page)

```
    """Convenience factory function that compiles a trainable structured environment.
    (for argument details see: Cutting2DEnvironment)
    """

    # init maze environment including observation and action interfaces
    env = maze_env_factory(max_pieces_in_inventory=max_pieces_in_inventory,
                           raw_piece_size=raw_piece_size,
                           static_demand=static_demand)

    # convert flat to structured environment
    return MaskedStructuredCutting2DEnvironment(env)
```

Test Script

When re-running the *main script* of the previous section with the masked version of the structured environment we now get the following output:

```
action_space 1:      Dict(piece_idx:Discrete(200))
observation_space 1: Dict(inventory:Box(200, 2), inventory_size:Box(1,), ordered_
↪piece:Box(2,), inventory_mask:Box(200,))
observation 1:       dict_keys(['inventory', 'inventory_size', 'ordered_piece',
↪'inventory_mask'])
action_space 2:      Dict(cut_order:Discrete(2), cut_rotation:Discrete(2))
observation_space 2: Dict(ordered_piece:Box(2,), selected_piece:Box(2,), cutting_
↪mask:Box(2,))
observation 2:       dict_keys(['selected_piece', 'ordered_piece', 'cutting_mask'])
```

As expected, both masks are contained in the respective observations and spaces. In the next section we will utilize these masks to enhance the sample efficiency ouf our trainers.

3.22.4 Training with Action Masking

In this part of the tutorial we will retrain the environment with step-conditional action masking activated and benchmark it with the initial, unmasked version.

The complete code for this part of the tutorial can be found here

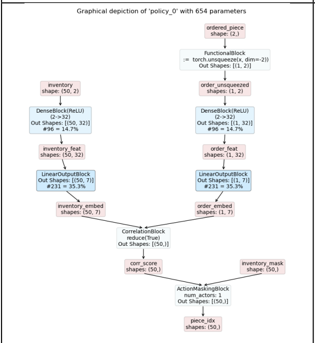
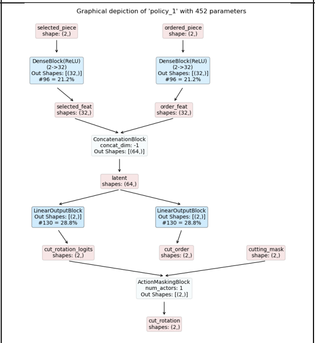
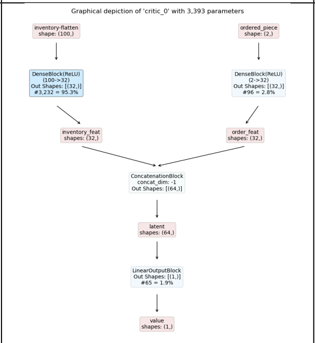
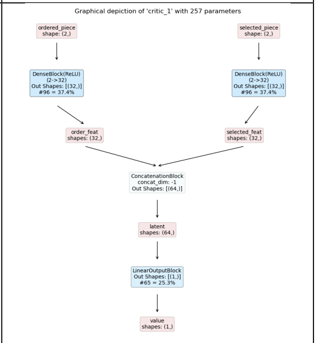
```
# relevant files
- cutting_2d
    - conf
        - env
            - tutorial_cutting_2d_flat.yaml
            - tutorial_cutting_2d_struct.yaml
            - tutorial_cutting_2d_struct_masked.yaml
        - model
            - tutorial_cutting_2d_flat.yaml
            - tutorial_cutting_2d_struct.yaml
            - tutorial_cutting_2d_struct_masked.yaml
        - wrappers
            - tutorial_cutting_2d.yaml
    - models
        - actor.py
        - critic.py
```

Page Overview

- *Masked Policy Models*
- *Retraining with Masking*
- *In Depth Inspection of Learning Progress*

Masked Policy Models

Before we can retrain the masked version of the structured environment we first need to specify how the masks are employed within the models. For this purpose we extend the two policy models with an `ActionMaskingBlock` applied to the respective logits. The resulting models are shown below:

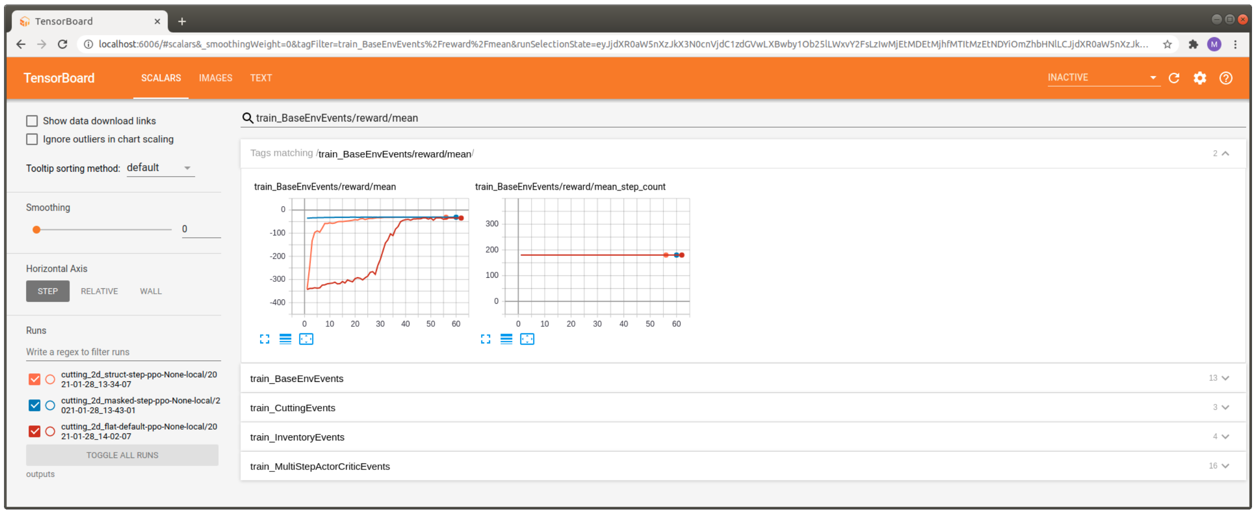
Masked Piece Selection Policy	Masked Cutting Policy	Piece Selection Critic	Cutting Critic
 Graphical depiction of 'policy_0' with 654 parameters ordered_piece shape: (2,) FunctionalBlock := torch.unsqueeze(x, dim=-2)) Out Shapes: [(1, 2)] inventory shape: (50, 2) order_unsqueezed shapes: (1, 2) DenseBlock(ReLU) (2->32) Out Shapes: [(50, 32)] #96 = 14.7% DenseBlock(ReLU) (2->32) Out Shapes: [(1, 32)] #96 = 14.7% inventory_feat shapes: (50, 32) order_feat shapes: (1, 32) LinearOutputBlock Out Shapes: [(50, 7)] #231 = 35.3% LinearOutputBlock Out Shapes: [(1, 7)] #231 = 35.3% inventory_embed shapes: (50, 7) order_embed shapes: (1, 7) CorrelationBlock reduce(True) Out Shapes: [(50,)] corr_score shapes: (50,) inventory_mask shape: (50,) ActionMaskingBlock num_actors: 1 Out Shapes: [(50,)] piece_idx shapes: (50,)	 Graphical depiction of 'policy_1' with 452 parameters selected_piece shape: (2,) ordered_piece shape: (2,) DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 21.2% DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 21.2% selected_feat shapes: (32,) order_feat shapes: (32,) ConcatenationBlock concat_dim: -1 Out Shapes: [(64,)] latent shapes: (64,) LinearOutputBlock Out Shapes: [(2,)] #130 = 28.8% LinearOutputBlock Out Shapes: [(2,)] #130 = 28.8% cut_rotation_logits shapes: (2,) cut_order shapes: (2,) cutting_mask shape: (2,) ActionMaskingBlock num_actors: 1 Out Shapes: [(2,)] cut_rotation shapes: (2,)	 Graphical depiction of 'critic_0' with 3,393 parameters inventory-flatten shape: (100,) ordered_piece shape: (2,) DenseBlock(ReLU) (100->32) Out Shapes: [(32,)] #3,232 = 95.3% DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 2.8% inventory_feat shapes: (32,) order_feat shapes: (32,) ConcatenationBlock concat_dim: -1 Out Shapes: [(64,)] latent shapes: (64,) LinearOutputBlock Out Shapes: [(1,)] #65 = 1.9% value shapes: (1,)	 Graphical depiction of 'critic_1' with 257 parameters ordered_piece shape: (2,) selected_piece shape: (2,) DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 37.4% DenseBlock(ReLU) (2->32) Out Shapes: [(32,)] #96 = 37.4% order_feat shapes: (32,) selected_feat shapes: (32,) ConcatenationBlock concat_dim: -1 Out Shapes: [(64,)] latent shapes: (64,) LinearOutputBlock Out Shapes: [(1,)] #65 = 25.3% value shapes: (1,)

Retraining with Masking

```
maze-run -cn conf_train env=tutorial_cutting_2d_struct_masked wrappers=tutorial_
↪cutting_2d \
model=tutorial_cutting_2d_struct_masked algorithm=ppo
```

Once training has finished we can again inspect the progress with Tensorboard. To get a better feeling for the effect of action masking we benchmark the following versions of the environment:

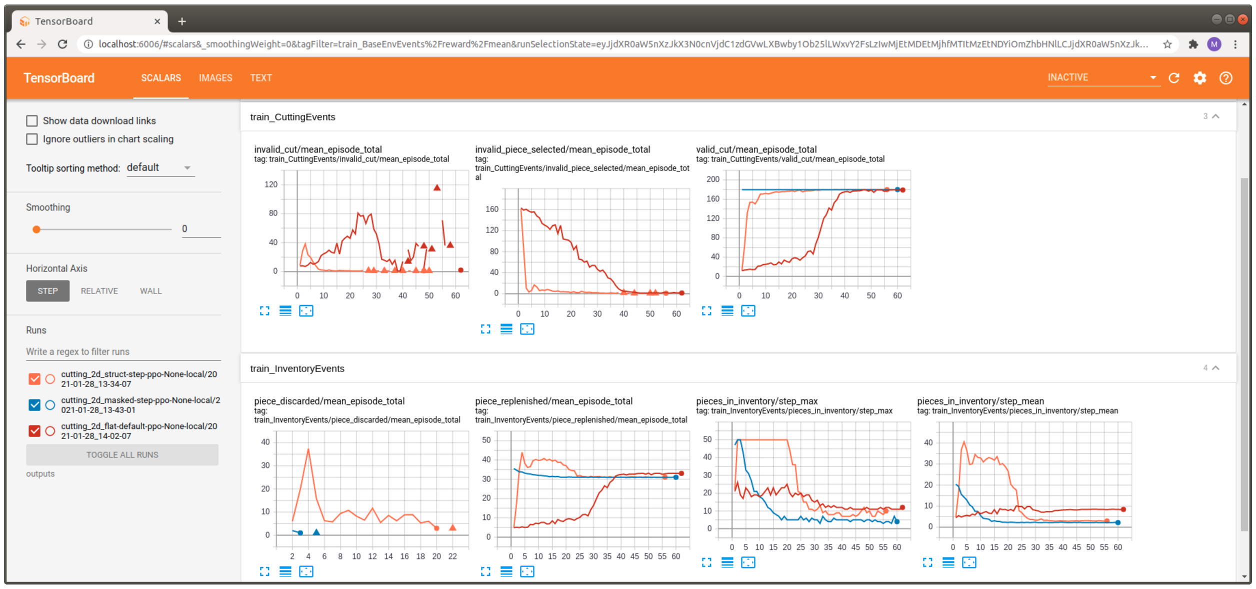
- Flat Gym-style environment with vanilla feed forward models (red)
- Structured Environment (e.g., with hierarchical sub-steps) with task specific policy networks (orange)
- Structured Environment (e.g., with hierarchical sub-steps) with masked, task specific policy networks (blue)


TensorBoard
SCALARS
IMAGES
TEXT
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
cutting_2d_struct-step-ppo-None-local/2021-01-28_13-34-07
cutting_2d_masked-step-ppo-None-local/2021-01-28_13-43-01
cutting_2d_flat-default-ppo-None-local/2021-01-28_14-02-07
TOGGLE ALL RUNS
outputs
train_BaseEnvEvents/reward/mean
Tags matching /train_BaseEnvEvents/reward/mean/
train_BaseEnvEvents/reward/mean
train_BaseEnvEvents/reward/mean_step_count
train_BaseEnvEvents
train_CuttingEvents
train_InventoryEvents
train_MultiStepActorCriticEvents

First of all we can observe a massive increase in learning speed when activating action masking. In fact the reward of the masked model starts at an much higher initial value. We can also observe a substantial improvement when switching from the vanilla feed forward Gym-style example (red) to the structured environment using task specific custom models (orange).

In Depth Inspection of Learning Progress

In this section we make use of *Maze Event Logging System* to learn more about the learning progress and behaviour of the respective versions.


TensorBoard
SCALARS
IMAGES
TEXT
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
cutting_2d_struct-step-ppo-None-local/2021-01-28_13-34-07
cutting_2d_masked-step-ppo-None-local/2021-01-28_13-43-01
cutting_2d_flat-default-ppo-None-local/2021-01-28_14-02-07
TOGGLE ALL RUNS
outputs
train_CuttingEvents
invalid_cut/mean_episode_total
tag: train_CuttingEvents/invalid_cut/mean_episode_total
invalid_piece_selected/mean_episode_total
tag: train_CuttingEvents/invalid_piece_selected/mean_episode_total
valid_cut/mean_episode_total
tag: train_CuttingEvents/valid_cut/mean_episode_total
train_InventoryEvents
piece_discarded/mean_episode_total
tag: train_InventoryEvents/piece_discarded/mean_episode_total
piece_replenished/mean_episode_total
tag: train_InventoryEvents/piece_replenished/mean_episode_total
pieces_in_inventory/step_max
tag: train_InventoryEvents/pieces_in_inventory/step_max
pieces_in_inventory/step_mean
tag: train_InventoryEvents/pieces_in_inventory/step_mean

- When looking at the cutting events we see that the agent utilizing action masking only performs valid cutting attempts right from the beginning of the training process. Avoiding the part where the agent has to learn via reward shaping which cuts are actually possible allows it to focus on learning how to cut efficiently. For the two other versions exactly the latter is the case.
- The same observation holds for the piece selection policy where again a lot of invalid attempts take place for the two unmasked versions.
- Finally, when looking at the inventory statistics we can see that the masked agent keeps very few pieces in

inventory (*pieces in inventory*) which is why it never has to discard any piece (*pieces discarded*) that might be required to fulfill upcoming customer orders.

Turning a "flat" MazeEnv into a StructuredEnv We will reformulate the problem from a "flat" Gym-style environment into a structured environment.

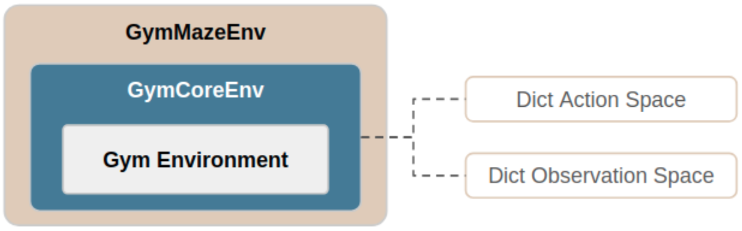
Training the Structured Environment We will train the structured environment with a Maze Trainer.

Adding Step-Conditional Action Masking We will learn how to substantially increase the sample efficiency by adding step-conditional action masking.

Training with Action Masking We will retrain the structured environment with step-conditional action masking activated and benchmark it with the initial version environment.

3.23 Integrating an Existing Gym Environment

Maze supports a seamless integration of existing OpenAI Gym environments. To get full Maze feature support for Gym environments we first have to transform them into Maze environments. This page shows how this is easily accomplished via the *GymMazeEnv*.



A Gym environment is transformed into a `GymMazeEnv` by:

- Wrapping the Gym environment into a `GymCoreEnv`.
- This requires transforming the observation and action spaces into a dictionary spaces via the `GymObservationConversion` and `GymActionConversion` interfaces.
- Finally, the GymCoreEnv is packed into a `GymMazeEnv` which is fully compatible with all other Maze components and modules.

To get a better understanding of the overall structure please refer to the *Maze environment hierarchy*.

3.23.1 Instantiating a Gym Environment as a Maze Environment

The config snippet below shows how to instantiate an existing, already registered Gym environment as a GymMazeEnv referenced by its environment name (here *CartPole-v0*).

```
# @package env
type: maze.core.wrappers.maze_gym_env_wrapper.make_gym_maze_env
name: "CartPole-v0"
```

To achieve the same result directly with plain Python you can start with the code snippet below.

```
from maze.core.wrappers.maze_gym_env_wrapper import GymMazeEnv
env = GymMazeEnv(env="CartPole-v0")
```

In case your environment is not yet registered with Gym you can also directly instantiate the Gym environment before passing it to the *GymMazeEnv*. This might be useful in case you already have your own custom Gym environments implemented.

```
import gym
from maze.core.wrappers.maze_gym_env_wrapper import GymMazeEnv
gym_env = gym.make("CartPole-v0")
env = GymMazeEnv(env=gym_env)
```

3.23.2 Where to Go Next

- For further details please see the *reference documentation*.
- Next you might be interested in how to *train an agent for your environment*.
- You might also want to read up on the *Maze environment hierarchy* for the bigger picture.

3.24 Combining Maze with other RL Frameworks

This tutorial explains how to use general Maze features in combination with existing RL frameworks. In particular, we will apply *observation normalization* before optimizing a policy with the stable-baselines3 A2C trainer. When adding new features to Maze we put a strong emphasis on reusablity to allow you to make use of as much of these features as possible but still give you the opportunity to stick to the optimization framework you are most comfortable or familiar with.

Since RLlib already has a dedicated spot within Maze we rely on stable-baselines3 for this tutorial. However, it is important to note that the examples below will also work with any other Python-based RL framework compatible with Gym environments.

We provide two different versions showing how to arrive at an observation normalized environment. The first one is written in *plain Python* where the second reproduces the Python example with a *Hydra configuration*.

Note: Although, this tutorial explains how to reuse observation normalization there is of course no limitation to this sole feature. So if you find this useful we definitely recommend you to browse through our *Environment Customization* section in the sidebar.

3.24.1 Reusing Environment Customization Features

The basis for this tutorial is the official getting started snippet of stable-baselines showing how to train and run A2C on a CartPole environment. We added a few comments to make things a bit more explicit.

If you would like to run this example yourself make sure to install stable-baselines3 first.

```
"""
Getting started example from:
https://stable-baselines3.readthedocs.io/en/master/guide/quickstart.html
"""

import gym
from stable_baselines3 import A2C

# ENV INSTANTIATION
# -----------------
env = gym.make('CartPole-v0')
```

(continues on next page)

(continued from previous page)

```
# TRAINING AND ROLLOUT
# --------------------

model = A2C('MlpPolicy', env, verbose=1)
model.learn(total_timesteps=10000)

obs = env.reset()
for i in range(1000):
    action, _state = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
      obs = env.reset()
```

Below you find exactly the same example but with an observation normalized environment. The following modifications compared to the example above are required:

- Instantiate a GymMazeEnv instead of a standard Gym environment
- Wrap the environment with the ObservationNormalizationWrapper
- Estimate normalization statistics from actual environment interactions

As you might already have experienced, re-coding these steps for different environments and experiments can get quite cumbersome. The wrapper also dumps the estimated statistics in a file (*statistics.pkl*) to reuse them later on for agent deployment.

```
"""
Contains an example showing how to train
an observation normalized maze environment with stable-baselines.
"""

from maze.core.agent.random_policy import RandomPolicy
from maze.core.wrappers.maze_gym_env_wrapper import GymMazeEnv
from maze.core.wrappers.no_dict_spaces_wrapper import NoDictSpacesWrapper
from maze.core.wrappers.observation_normalization.observation_normalization_utils␣
↪import \
    obtain_normalization_statistics
from maze.core.wrappers.observation_normalization.observation_normalization_wrapper␣
↪import \
    ObservationNormalizationWrapper

from stable_baselines3 import A2C

# ENV INSTANTIATION: a GymMazeEnv instead of a gym.Env
# ----------------------------------------------------
env = GymMazeEnv('CartPole-v0')

# OBSERVATION NORMALIZATION
# -------------------------

# we wrap the environment with the ObservationNormalizationWrapper
# (you can find details on this in the section on observation normalization)
env = ObservationNormalizationWrapper(
    env=env,
    default_strategy="maze.normalization_strategies.
↪MeanZeroStdOneObservationNormalizationStrategy",
    default_strategy_config={"clip_range": (None, None), "axis": 0},
```

(continues on next page)

(continued from previous page)

```
    default_statistics=None, statistics_dump="statistics.pkl",
    sampling_policy=RandomPolicy(env.action_spaces_dict),
    exclude=None, manual_config=None)

# next we estimate the normalization statistics by
# (1) collecting observations by randomly sampling 1000 transitions from the␣
↪environment
# (2) computing the statistics according to the define normalization strategy
normalization_statistics = obtain_normalization_statistics(env, n_samples=1000)
env.set_normalization_statistics(normalization_statistics)

# after this step all observations returned by the environment will be normalized

# stable-baselines does not support dict spaces so we have to remove them
env = NoDictSpacesWrapper(env)

# TRAINING AND ROLLOUT (remains unchanged)
# ----------------------------------------

model = A2C('MlpPolicy', env, verbose=1)
model.learn(total_timesteps=10000)

obs = env.reset()
for i in range(1000):
    action, _state = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
        obs = env.reset()
```

3.24.2 Reusing the Hydra Configuration System

This example is identical to the the previous one but instead of instantiated everything *directly from Python* it utilizes the *Hydra configuration system*.

```
"""
Contains an example showing how to train an observation normalized maze environment
instantiated from a hydra config with stable-baselines.
"""

from maze.core.utils.config_utils import make_env_from_hydra
from maze.core.wrappers.no_dict_spaces_wrapper import NoDictSpacesWrapper
from maze.core.wrappers.observation_normalization.observation_normalization_utils␣
↪import \
    obtain_normalization_statistics

from stable_baselines3 import A2C

# ENV INSTANTIATION: from hydra config file
# -----------------------------------------
env = make_env_from_hydra("conf")

# OBSERVATION NORMALIZATION
# -------------------------
```

(continues on next page)

(continued from previous page)

```
# next we estimate the normalization statistics by
# (1) collecting observations by randomly sampling 1000 transitions from the␣
↪environment
# (2) computing the statistics according to the define normalization strategy
normalization_statistics = obtain_normalization_statistics(env, n_samples=1000)
env.set_normalization_statistics(normalization_statistics)

# stable-baselines does not support dict spaces so we have to remove them
env = NoDictSpacesWrapper(env)

# TRAINING AND ROLLOUT (remains unchanged)
# ----------------------------------------

model = A2C('MlpPolicy', env, verbose=1)
model.learn(total_timesteps=10000)

obs = env.reset()
for i in range(1000):
    action, _state = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    env.render()
    if done:
        obs = env.reset()
```

This is the corresponding hydra config:

```
# @package _global_

# defines environment to instantiate
env:
  type: maze.core.wrappers.maze_gym_env_wrapper.GymMazeEnv
  env: "CartPole-v0"

# defines wrappers to apply
wrappers:
  # Observation Normalization Wrapper
  ObservationNormalizationWrapper:
    default_strategy: maze.normalization_strategies.
↪MeanZeroStdOneObservationNormalizationStrategy
    default_strategy_config:
      clip_range: [~, ~]
      axis: 0
    default_statistics: ~
    statistics_dump: statistics.pkl
    sampling_policy:
      type: maze.core.agent.random_policy.RandomPolicy
    exclude: ~
    manual_config: ~
```

3.24.3 Where to Go Next

- You can learn more about the *Hydra configuration system*.
- As *observation normalization* is not the scope of this section we recommend to read up on this in the dedicated section.
- You might be also interested in *observation pre-processing* and the remaining environment customization options (see sidebar *Environment Customization*).
- You can also check out the built-in Maze Trainers with full dictionary space support for observations and actions.
- You can also make use of the full *Maze environment hierarchy*.

3.25 Rollout and Training Examples

Run a rollout to test an environment with random action sampling:

```
maze-run -cn conf_rollout env.name=CartPole-v1 policy=random_policy
```

Run a rollout and render the state of the environment:

```
maze-run -cn conf_rollout env.name=CartPole-v1 policy=random_policy \
runner=sequential runner.render=true
```

Train a policy with evolutionary strategies (ES):

```
maze-run -cn conf_train env.name=CartPole-v1 algorithm=es model=vector_obs
```

Train a policy with with an actor-critic trainer such as A2C:

```
maze-run -cn conf_train env.name=CartPole-v1 algorithm=a2c \
model=vector_obs critic=default_state
```

Run a rollout of a policy, trained with the command above:

```
maze-run -cn conf_rollout env.name=CartPole-v1 model=vector_obs \
policy=torch_policy input_dir=outputs/<experiment-dir>
```

3.26 Tensorboard and Command Line Logging

This page gives a brief overview of the Tensorboard and command line logging facilities of Maze. We will show examples based on the cutting-2D Maze environment to make things a bit more interesting.

To understand the underlying concepts we recommend to read the sections on *event and KPI logging* as well as on the *Maze event system*.

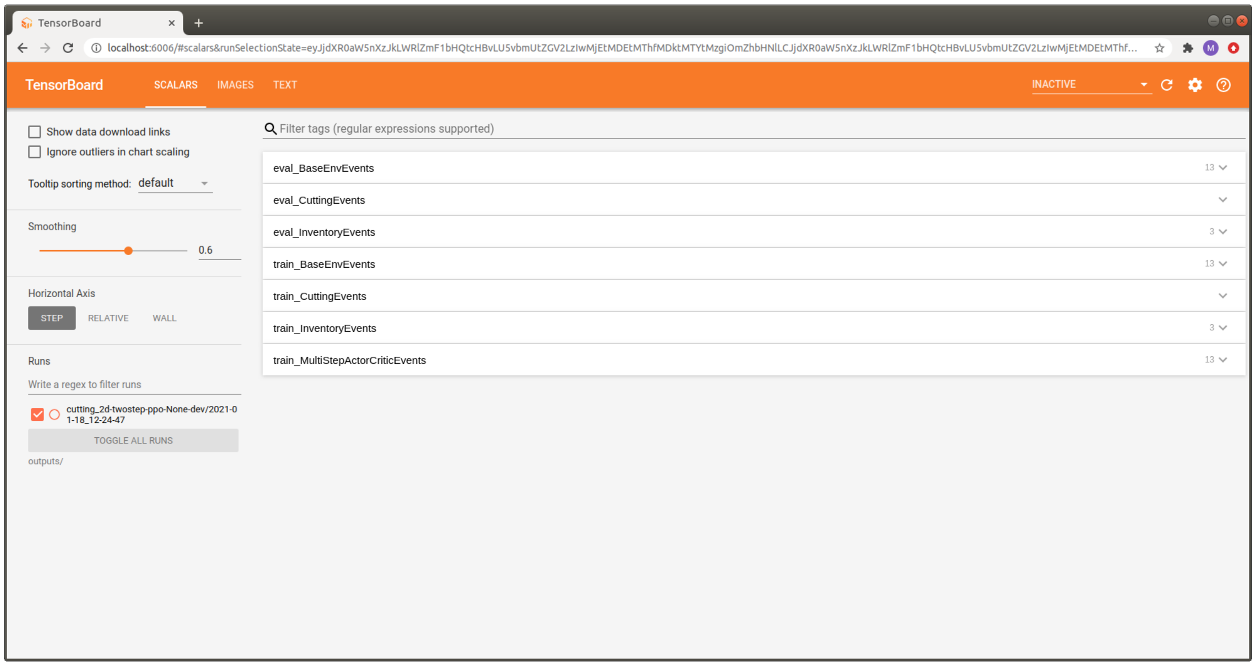
3.26.1 Tensorboard Logging

To watch the training progress with Tensorboard start it by running:

```
tensorboard --logdir outputs/
```

and view it with your browser at http://localhost:6006/.

You will get an output similar to the one shown in the image below.



Tooltip sorting method: default

Smoothing 0.6

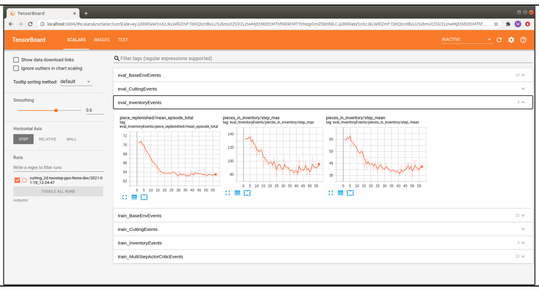
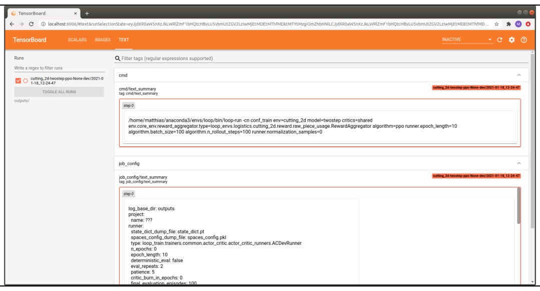
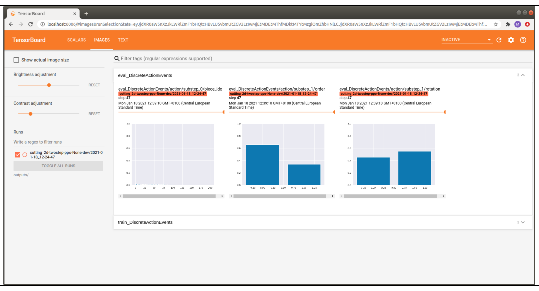
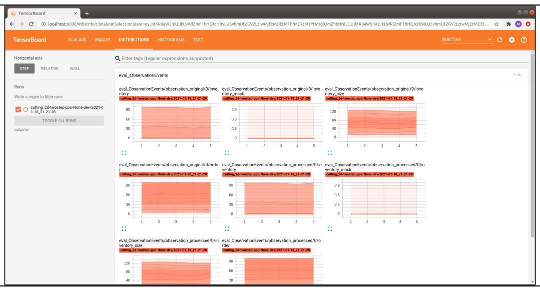
cutting_2d-twostep-ppo-None-dev/2021-01-18_12-24-47

eval_BaseEnvEvents 13
eval_CuttingEvents
eval_InventoryEvents 3
train_BaseEnvEvents 13
train_CuttingEvents
train_InventoryEvents 3
train_MultiStepActorCriticEvents 13

To keep everything organized and avoid having to browse through tons of pages we group the contained items into semantically connected sections:

- Since Maze allows you to use different environments for training and evaluation, each logging section has a *train_* or *eval_* prefix to show if the corresponding stats were logged as part of the training or the evaluation environment.
- The BaseEnvEvents sections (i.e., *eval_BaseEnvEvents* and *train_BaseEnvEvents* contain general statistics such as rewards or step counts. This section is always present, independent of the environment used.
- Other sections are specific to the environment used. In the example above, these are the *CuttingEvents* and the *InventoryEvents*.
- In addition, we get one additional section containing stats of the trainer used, called *train_NameOfTrainerEvents*. It contains statistics such as policy loss, gradient norm or value loss. This section is not present for the evaluation environment.

The gallery below shows some additional useful examples and features of the Maze Tensorboard log (click the images to display them in large).

Logging of component specific events in the *SCALARS* tab. (Useful for understanding the environment)	Logging of the training command and the complete hydra job config in the *TEXT* tab. (Useful for reproducing experiments)
	
Logging of action sampling statistics in the *IMAGE* tab. (Useful for understanding the agent's behaviour)	*Logging of observation distributions* in the *DISTRIBUTIONS* and *HISTOGRAMS* tab. (Useful for analysing observations)
	

```
/home/matthias/anaconda3/envs/loop/bin/loop-run -cn conf_train env=cutting_2d model=twostep critics=shared
env.core_env.reward_aggregator.type=loop_envs.logistics.cutting_2d.reward.raw_piece_usage.RewardAggregator algorithm=ppo runner.epoch_length=10
algorithm.batch_size=100 algorithm.n_rollout_steps=100 runner.normalization_samples=0
```

```
log_base_dir: outputs
project:
  name: ???
runner:
  state_dict_dump_file: state_dict.pt
  spaces_config_dump_file: spaces_config.pkl
  type: loop_train.trainers.common.actor_critic.actor_critic_runners.ACDevRunner
  n_epochs: 0
  epoch_length: 10
  deterministic_eval: false
  eval_repeats: 2
  patience: 5
  critic_burn_in_epochs: 0
  final_evaluation_episodes: 100
```

3.26.2 Command Line Logging

Whenever you start a training run you will also get a command line output similar to the one shown below. Analogously to the Tensorboard log, Maze distinguishes between *train* and *eval* outputs and groups the items into semantically connected output blocks.

```
 step|path                                                                   |  ␣
↪           value
=====|=======================================================================|===================
    1|train    MultiStepActorCritic..time_rollout            ......................|  ␣
↪           1.091
    1|train    MultiStepActorCritic..learning_rate           ......................|  ␣
↪           0.000
    1|train    MultiStepActorCritic..policy_loss             0                     |  ␣
↪          -0.000
    1|train    MultiStepActorCritic..policy_grad_norm        0                     |  ␣
↪           0.001
    1|train    MultiStepActorCritic..policy_entropy          0                     |  ␣
↪           1.593
    1|train    MultiStepActorCritic..policy_loss             1                     |  ␣
↪          -0.000
    1|train    MultiStepActorCritic..policy_grad_norm        1                     |  ␣
↪           0.008
```

(continues on next page)

(continued from previous page)

```
    1|train     MultiStepActorCritic..policy_entropy        1                     |  ␣
↪           0.295
    1|train     MultiStepActorCritic..critic_value          0                     |  ␣
↪          -0.199
    1|train     MultiStepActorCritic..critic_value_loss     0                     |  ␣
↪         116.708
    1|train     MultiStepActorCritic..critic_grad_norm      0                     |  ␣
↪           0.500
    1|train     MultiStepActorCritic..time_update           ......................|  ␣
↪           1.642
    1|train     DiscreteActionEvents  action                substep_0/piece_idx   | ␣
↪[len:4000, :54.8]
    1|train     BaseEnvEvents         reward                median_step_count     |  ␣
↪         200.000
    1|train     BaseEnvEvents         reward                mean_step_count       |  ␣
↪         200.000
    1|train     BaseEnvEvents         reward                total_step_count      |  ␣
↪        4000.000
    1|train     BaseEnvEvents         reward                total_episode_count   |  ␣
↪          20.000
    1|train     BaseEnvEvents         reward                episode_count         |  ␣
↪          20.000
    1|train     BaseEnvEvents         reward                std                   |  ␣
↪           1.465
    1|train     BaseEnvEvents         reward                mean                  |  ␣
↪         -71.950
    1|train     BaseEnvEvents         reward                min                   |  ␣
↪         -75.000
    1|train     BaseEnvEvents         reward                max                   |  ␣
↪         -70.000
    1|train     DiscreteActionEvents  action                substep_1/order       |  ␣
↪[len:4000, :0.5]
    1|train     DiscreteActionEvents  action                substep_1/rotation    |  ␣
↪[len:4000, :0.5]
    1|train     InventoryEvents       piece_replenished     mean_episode_total    |  ␣
↪          71.950
    1|train     InventoryEvents       pieces_in_inventory   step_max              |  ␣
↪         163.000
    1|train     InventoryEvents       pieces_in_inventory   step_mean             |  ␣
↪          69.946
    1|train     CuttingEvents         valid_cut             mean_episode_total    |  ␣
↪         200.000
    1|train     BaseEnvEvents         kpi                   max/raw_piece_usage_..|  ␣
↪           0.375
    1|train     BaseEnvEvents         kpi                   min/raw_piece_usage_..|  ␣
↪           0.350
    1|train     BaseEnvEvents         kpi                   std/raw_piece_usage_..|  ␣
↪           0.007
    1|train     BaseEnvEvents         kpi                   mean/raw_piece_usage..|  ␣
↪           0.360
Time required for epoch: 19.43s
Update epoch - 1
 step|path                                                                        |  ␣
↪           value
=====|============================================================================|===================
    2|eval      DiscreteActionEvents  action                substep_0/piece_idx   |  ␣
↪[len:800, :53.2]
```

(continues on next page)

(continued from previous page)

```
   2|eval       BaseEnvEvents          reward                median_step_count      |  ␣
↪          200.000
   2|eval       BaseEnvEvents          reward                mean_step_count        |  ␣
↪          200.000
   2|eval       BaseEnvEvents          reward                total_step_count       |  ␣
↪         1600.000
   2|eval       BaseEnvEvents          reward                total_episode_count    |  ␣
↪            8.000
   2|eval       BaseEnvEvents          reward                episode_count          |  ␣
↪            4.000
   2|eval       BaseEnvEvents          reward                std                    |  ␣
↪            1.414
   2|eval       BaseEnvEvents          reward                mean                   |  ␣
↪          -71.000
   2|eval       BaseEnvEvents          reward                min                    |  ␣
↪          -73.000
   2|eval       BaseEnvEvents          reward                max                    |  ␣
↪          -69.000
   2|eval       DiscreteActionEvents   action                substep_1/order        |  ␣
↪ [len:800, :0.5]
   2|eval       DiscreteActionEvents   action                substep_1/rotation     |  ␣
↪ [len:800, :0.5]
   2|eval       InventoryEvents        piece_replenished     mean_episode_total     |  ␣
↪           71.000
   2|eval       InventoryEvents        pieces_in_inventory   step_max               |  ␣
↪          145.000
   2|eval       InventoryEvents        pieces_in_inventory   step_mean              |  ␣
↪           68.031
   2|eval       CuttingEvents          valid_cut             mean_episode_total     |  ␣
↪          200.000
   2|eval       BaseEnvEvents          kpi                   max/raw_piece_usage_..|  ␣
↪            0.365
   2|eval       BaseEnvEvents          kpi                   min/raw_piece_usage_..|  ␣
↪            0.345
   2|eval       BaseEnvEvents          kpi                   std/raw_piece_usage_..|  ␣
↪            0.007
   2|eval       BaseEnvEvents          kpi                   mean/raw_piece_usage..|  ␣
↪            0.355
```

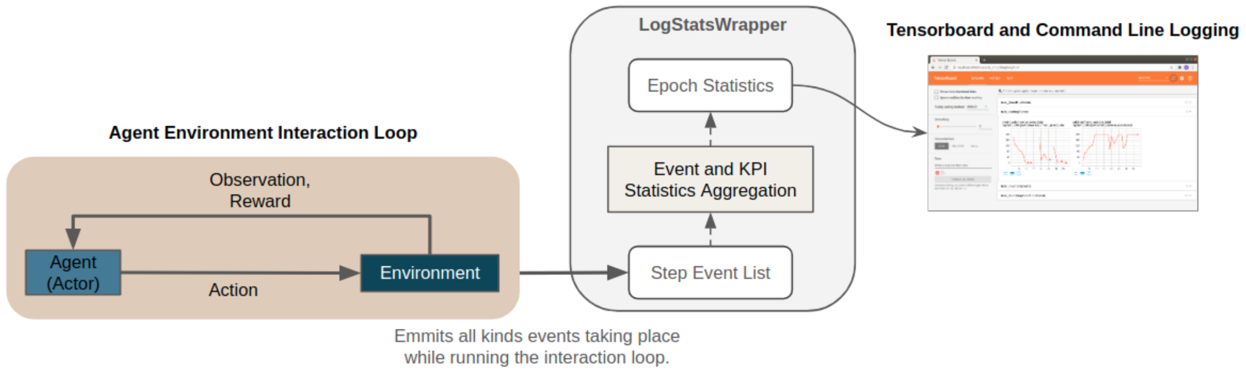
3.26.3 Where to Go Next

- For further details please see the *reference documentation*.
- For the bigger picture we refer to *event and KPI logging* as well as the *Maze event system*.
- You might be also interested in *observation distribution logging* and *action distribution logging*.

3.27 Event and KPI Logging

Monitoring only standard metrics such as reward or episode step count is not always sufficiently informative about the agent's behaviour and the problem at hand. To tackle this issue and to enable better inspection and logging tools for both, agents and environments, we introduce an event and key performance indicator (KPI) logging system. It is based on the more general *event system* and allows us to log and monitor environment specific metrics.

The figure below shows a conceptual overview of the logging system. In the remainder of this page we will go through the components in more detail.


Agent Environment Interaction Loop
Observation, Reward
Agent (Actor)
Action
Environment
LogStatsWrapper
Epoch Statistics
Event and KPI Statistics Aggregation
Step Event List
Tensorboard and Command Line Logging
Emmits all kinds events taking place while running the interaction loop.

3.27.1 Events

In this section we describe the event logging system from an usage perspective. To understand how this is embedded in the broader context of a Maze environment we refer to the *environments and KPI section* of our step by step tutorial as well as the dedicated section on the underlying *event system*.

In general, events can be define for any component involved in the RL process (e.g., environments, agents, . . .). They get fired by the respective component whenever they occur during the agent environment interaction loop. For logging, events are collected and aggregated via the `LogStatsWrapper`.

To provide full flexibility Maze allows to customize which statistics are computed at which stage of the aggregation process via event decorators (*step*, *episode*, *epoch*). The code snipped below contains an example for an event called `invalid_piece_selected` borrowed from the *cutting 2D tutorial*.

```
class CuttingEvents(ABC):
    """Events related to the cutting process."""

    @define_epoch_stats(np.mean, output_name="mean_episode_total")
    @define_episode_stats(sum)
    @define_step_stats(len)
    def invalid_piece_selected(self):
        """An invalid piece is selected for cutting."""
```

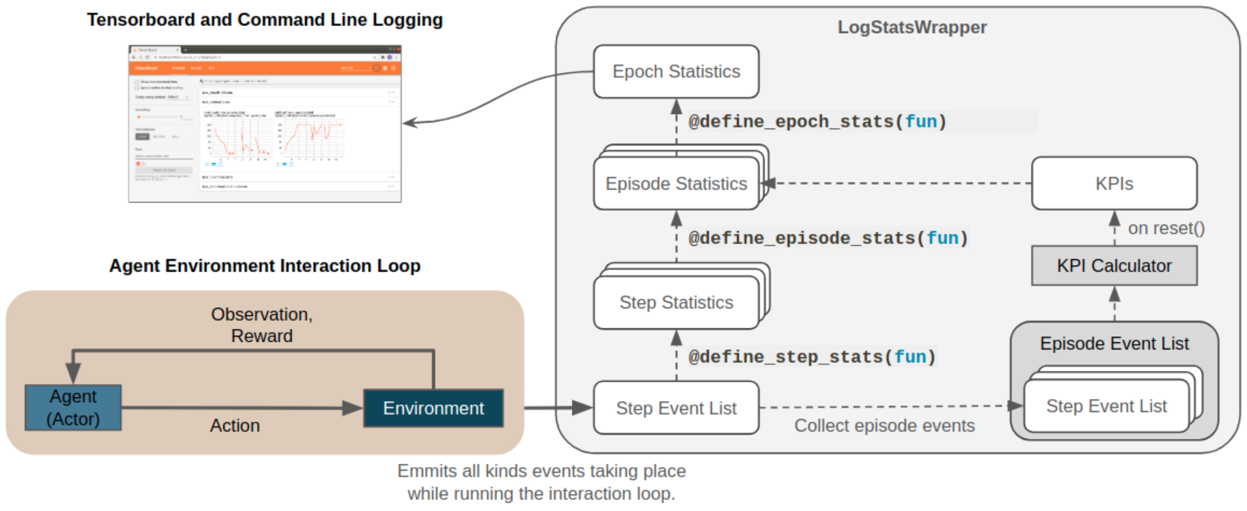
The snippet defines the following statistics aggregation hierarchy:

Step Statistics [`@define_step_stats(len)`]: in each environment step events e_i are collected as lists of events $\{e_i\}$. The function `len` associated with the decorator counts how often such an event occurred in the current step $Stats_{Step} = |\{e_i\}|$ (e.g., length of `invalid_piece_selected` event list).

Episode Statistics [`@define_episode_stats(sum)`]: defines how the S step statistics should be aggregated to episode statistics (e.g., by simply summing them up: $Stats_{Episode} = \sum^S Stats_{Step}$)

Epoch Statistics [`@define_epoch_stats(np.mean, output_name="mean_episode_total")`]: a training epoch consists of N episodes. This stage defines how these N episode statistics are averaged to epoch statistics (e.g., the mean of the contained episodes: $Stats_{Epoch} = (\sum^{N} Stats_{Episode})/N$).

The figure below provides a visual summary of the entire event statistics aggregation hierarchy as well as its relation to KPIs which will be explained in the next section. In *Tensorboard and on the command line* these events get then logged in dedicated sections (e.g., as *CuttingEvents*).



3.27.2 Key Performance Indicators (KPIs)

In applied RL settings the reward is not always the target metric we aim at optimizing from an economical perspective. Sometimes *rewards are heavily shaped* to get the agent to learn the right behaviour. This makes it hard to interpret for humans. For such cases Maze supports computing and logging of additional Key Performance Indicators (KPIs) along with the reward via the `KpiCalculator` implemented as a part of the `CoreEnv` (as reward KPIs are logged as *BaseEnvEvents*).

KPIs are in contrast to events computed in an aggregated form at the end of an episode triggered by the `reset()` method of the `LogStatsWrapper`. This is why we can compute them in a normalized fashion (e.g., dived by the total number of steps in an episode). Conceptually KPIs life on the same level as episode statistics in the logging hierarchy (see figure above).

For further details on how to implement a concrete KPI calculator we refer to the *KPI section* of our tutorial.

3.27.3 Plain Python Configuration

When working with the *CLI and Hydra configs* all components necessary for logging are automatically instantiated under the hood. In case you would like to test or run your logging setup directly from Python you can start with the snippet below.

```python
from docs.tutorial_maze_env.part04_events.env.maze_env import maze_env_factory
from maze.utils.log_stats_utils import SimpleStatsLoggingSetup
from maze.core.wrappers.log_stats_wrapper import LogStatsWrapper

# init maze environment
env = maze_env_factory(max_pieces_in_inventory=200, raw_piece_size=[100, 100],
                       static_demand=(30, 15))

# wrap environment with logging wrapper
env = LogStatsWrapper(env, logging_prefix="main")
```

(continues on next page)

(continued from previous page)

```
# register a console writer and connect the writer to the statistics logging system
with SimpleStatsLoggingSetup(env):
    # reset environment and run interaction loop
    obs = env.reset()
    for i in range(15):
        action = env.action_space.sample()
        obs, reward, done, info = env.step(action)
```

To get access to event and KPI logging we need to wrap the environment with the `LogStatsWrapper`. To simplify the statistics logging setup we rely on the `SimpleStatsLoggingSetup` helper class.

When running the script you will get an output as shown below. Note that statistics of both, events and KPIs, are printed along with default *reward* or *action* statistics.

```
 step|path                                                                        |     ␣
↪           value
=====|============================================================================|=====================
    1|main     DiscreteActionEvents  action                substep_0/order        |     ␣
↪[len:15, :0.5]
    1|main     DiscreteActionEvents  action                substep_0/piece_idx    |    ␣
↪[len:15, :82.3]
    1|main     DiscreteActionEvents  action                substep_0/rotation     |     ␣
↪[len:15, :0.7]
    1|main     BaseEnvEvents         reward                median_step_count      |     ␣
↪          15.000
    1|main     BaseEnvEvents         reward                mean_step_count        |     ␣
↪          15.000
    1|main     BaseEnvEvents         reward                total_step_count       |     ␣
↪          15.000
    1|main     BaseEnvEvents         reward                total_episode_count    |     ␣
↪           1.000
    1|main     BaseEnvEvents         reward                episode_count          |     ␣
↪           1.000
    1|main     BaseEnvEvents         reward                std                    |     ␣
↪           0.000
    1|main     BaseEnvEvents         reward                mean                   |     ␣
↪         -29.000
    1|main     BaseEnvEvents         reward                min                    |     ␣
↪         -29.000
    1|main     BaseEnvEvents         reward                max                    |     ␣
↪         -29.000
    1|main     InventoryEvents       piece_replenished     mean_episode_total     |     ␣
↪           3.000
    1|main     InventoryEvents       pieces_in_inventory   step_max               |     ␣
↪         200.000
    1|main     InventoryEvents       pieces_in_inventory   step_mean              |     ␣
↪         200.000
    1|main     CuttingEvents         invalid_cut           mean_episode_total     |     ␣
↪          14.000
    1|main     InventoryEvents       piece_discarded       mean_episode_total     |     ␣
↪           2.000
    1|main     CuttingEvents         valid_cut             mean_episode_total     |     ␣
↪           1.000
    1|main     BaseEnvEvents         kpi                   max/raw_piece_usage_..|     ␣
↪           0.000
    1|main     BaseEnvEvents         kpi                   min/raw_piece_usage_..|     ␣
↪           0.000
```

(continues on next page)

(continued from previous page)

```
   1|main    BaseEnvEvents         kpi                     std/raw_piece_usage_..|    ␣
↪          0.000
   1|main    BaseEnvEvents         kpi                     mean/raw_piece_usage..|    ␣
↪          0.000
```

3.27.4 Where to Go Next

- You can learn more about the general *event system*.
- For a more implementation oriented summary you can visit the *events and KPI section* of our tutorial.
- To see another application of the event system you can read up on *reward customization and shaping*.

3.28 Action Distribution Visualization

There are situations where it turns out to be extremely useful to watch the evolution of an agent's sampling behaviour throughout the training process. Looking at the action sampling distribution often provides a first intuition about the agent's behaviour without the need to look at individual rollouts.

However, most importantly it is a great debugging tool immediately revealing if:

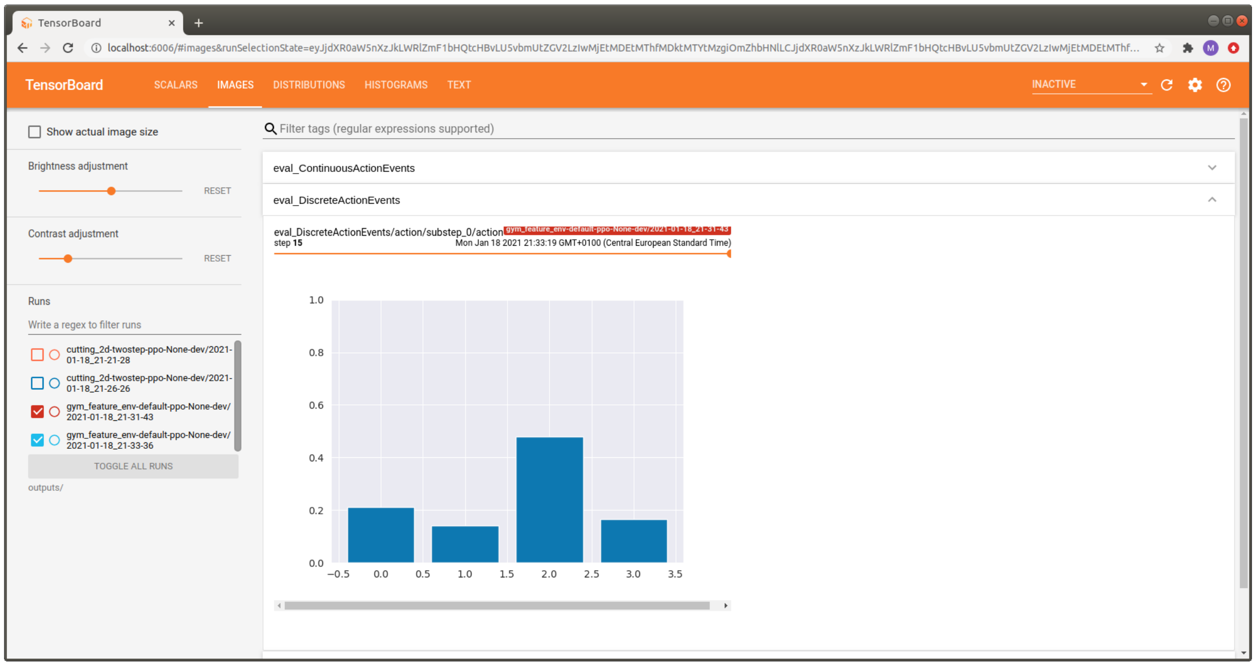
- the weights of the policy collapsed during training (e.g., the agent starts sampling always the same actions even though this does not make sense for the environment at hand).
- observations are properly normalized and the weights of the policy are initialized accordingly to result in a healthy initial sampling behaviour of the untrained model (e.g., each discrete action is taken a similar number of times when starting training).
- biasing the weights of the policy output layer results in the expected sampling behaviour (e.g., initially sampling an action twice as often as the remaining ones).
- the agents actually starts learning (i.e., the sampling distributions changes throughout the training epochs).

Maze visualizes action sampling distributions on a per-epoch basis in the *IMAGES* tab of Tensorboard. By using the slider above the images you can step through the training epochs and see how the sampling distribution evolves over time.

3.28.1 Discrete and Multi Binary Actions

Each *action space* has a dedicated visualization assigned. Discrete and multi-binary action spaces are visualized via histograms. The example below shows an action sampling distribution for the discrete version of LunarLander-v2. The indices on the x-axis correspond to the available actions:

- Action a_0 - do nothing
- Action a_1 - fire left orientation engine
- Action a_2 - fire main engine
- Action a_3 - fire right orientation engine

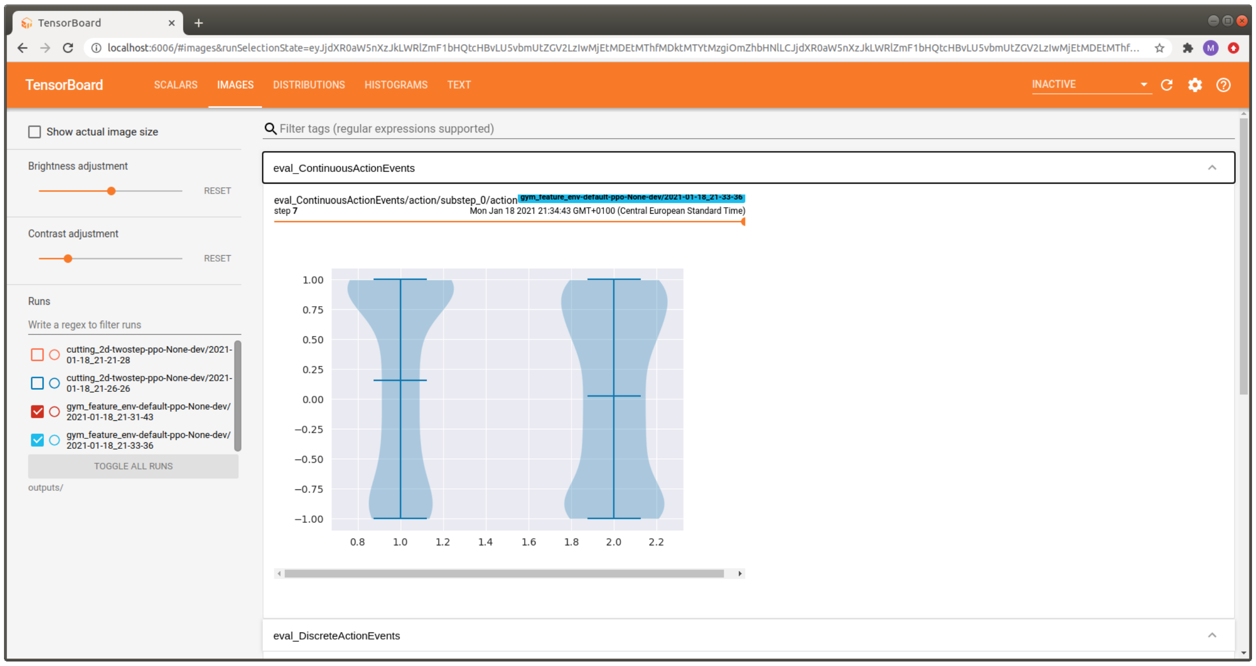

TensorBoard
SCALARS
IMAGES
DISTRIBUTIONS
HISTOGRAMS
TEXT
INACTIVE
Show actual image size
Brightness adjustment
RESET
Contrast adjustment
RESET
Runs
Write a regex to filter runs
cutting_2d-twostep-ppo-None-dev/2021-01-18_21-21-28
cutting_2d-twostep-ppo-None-dev/2021-01-18_21-26-26
gym_feature_env-default-ppo-None-dev/2021-01-18_21-31-43
gym_feature_env-default-ppo-None-dev/2021-01-18_21-33-36
TOGGLE ALL RUNS
outputs/
Filter tags (regular expressions supported)
eval_ContinuousActionEvents
eval_DiscreteActionEvents
eval_DiscreteActionEvents/action/substep_0/action
gym_feature_env-default-ppo-None-dev/2021-01-18_21-31-43
step 15
Mon Jan 18 2021 21:33:19 GMT+0100 (Central European Standard Time)
1.0
0.8
0.6
0.4
0.2
0.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

We can see that action a_2 (fire main engine) is taken most often, which is reasonable for this environment.

3.28.2 Continuous Actions

Continuous actions (Box spaces) are visualized via violin plots. The example below shows an action sampling distribution for LunarLanderContinuous-v2. The indices on the x-axis correspond to the available actions:

- Action a_1 - controls the main engine:
  - $a_1 \in [-1, 0]$: off
  - $a_1 \in (0, 1]$ throttle from 50% to 100% power (can't work with less than 50%).
- Action a_2 - controls the orientation engines:
  - $a_2 \in [-1.0, -0.5]$: fire left engine
  - $a_2 \in [0.5, 1.0]$: fire right engine
  - $a_2 \in (-0.5, 0.5)$: off


TensorBoard
SCALARS
IMAGES
DISTRIBUTIONS
HISTOGRAMS
TEXT
INACTIVE
Show actual image size
Brightness adjustment
RESET
Contrast adjustment
RESET
Runs
Write a regex to filter runs
cutting_2d-twostep-ppo-None-dev/2021-01-18_21-21-28
cutting_2d-twostep-ppo-None-dev/2021-01-18_21-26-26
gym_feature_env-default-ppo-None-dev/2021-01-18_21-31-43
gym_feature_env-default-ppo-None-dev/2021-01-18_21-33-36
TOGGLE ALL RUNS
outputs/
Filter tags (regular expressions supported)
eval_ContinuousActionEvents
eval_ContinuousActionEvents/action/substep_0/action gym_feature_env-default-ppo-None-dev/2021-01-18_21-33-36
step 7
Mon Jan 18 2021 21:34:43 GMT+0100 (Central European Standard Time)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
eval_DiscreteActionEvents

For the first action, corresponding to the main engine, values closer to 1.0 are sampled more often which is similar to the discrete case above.

3.28.3 Where to Go Next

- You might be also interested in *logging observation distributions*.

3.29 Observation Distribution Visualization

Maze provides the option to watch the evolution of value ranges of observations throughout the training process. This is especially useful for debugging your experiments and training runs as it reveals if:

- observations stay within an expected value range.
- observation normalization is applied correctly.
- observations drift as the agent's behaviour evolves throughout training.

3.29.1 Activating Observation Logging

To activate observation logging you only have to add the `ObservationLoggingWrapper` to your environment wrapper stack in your yaml config:

```yaml
# @package wrappers
ObservationLoggingWrapper: {}
```

If you are using plain Python you can start with the code snippet below.

```python
from maze.core.wrappers.maze_gym_env_wrapper import GymMazeEnv
from maze.core.wrappers.observation_logging_wrapper import ObservationLoggingWrapper

```

(continues on next page)

(continued from previous page)

```
env = GymMazeEnv(env="CartPole-v0")
env = ObservationLoggingWrapper(env)
```

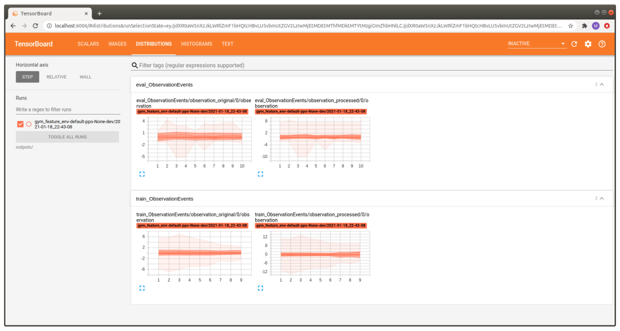
For both cases observations will be logged and distribution plots will be added to Tensorboard.

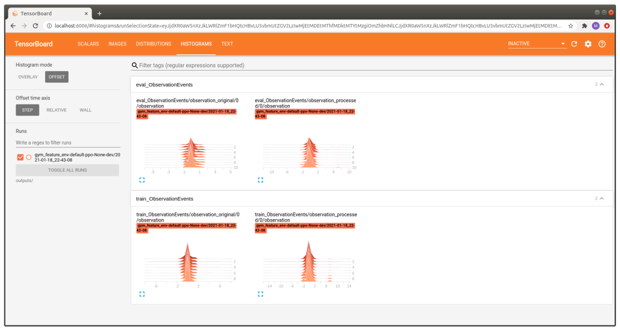
Warning: We support observation logging as an opt-in feature via a dedicated wrapper and recommend to use it only for debugging and inspection purposes. Once everything is on track and training works as expected we suggest to remove the wrapper again especially when dealing with environments with large observations. If you forget to remove it training might get slow and the memory consumption of Tensorboard might explode!

3.29.2 Tensorboard Examples

Maze visualizes observations on a per-epoch basis in the *DISTRIBUTIONS* and *HISTOGRAMS* tab of Tensorboard. By using the slider above the images you can step through the training epochs and see how the observation distribution evolves over time.

Below you see an example for both versions (just click the figure to view it in large).





Note that two different versions of the observation distribution are logged:

- *observation_original:* distribution of the original observation returned by the environment.
- *observation_processed:* distribution of the observation after processing (e.g. *pre-processing* or *normalization*).

This is especially useful to verify if the applied observation processing steps yield the expected result.

3.29.3 Where to Go Next

- You might be also interested in *logging action distributions*.
- You can learn more about *observation pre-processing* and *observation normalization*.

3.30 Runner Concept

In Maze, Runners are the entity responsible for launching and administering any job you start from a command line (like training or rollouts). They interpret the configuration and make sure the appropriate elements (models, trainers, etc.) are created, configured, and launched.

For a more detailed description of the runner concept, see *Hydra overview*. If you need to write custom runners for your project, see the *documentation for custom configuration*.

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

A2CAlgorithmConfig (*class in maze.train.trainers.a2c.a2c_algorithm_config*), 173

action() (*maze.core.log_events.action_events.ContinuousActionEvents method*), 80

action() (*maze.core.log_events.action_events.DiscreteActionEvents method*), 79

action() (*maze.core.wrappers.dict_action_wrapper.DictActionWrapper method*), 58

action() (*maze.core.wrappers.discretize_actions_wrapper.DiscretizeActionsWrapper method*), 60

action() (*maze.core.wrappers.no_dict_action_wrapper.NoDictActionWrapper method*), 58

action() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWrapper method*), 56

action() (*maze.core.wrappers.split_actions_wrapper.SplitActionsWrapper method*), 59

action() (*maze.core.wrappers.wrapper.ActionWrapper method*), 52

action_conversion() (*maze.core.env.maze_env.MazeEnv property*), 43

action_conversion_dict (*maze.core.env.maze_env.MazeEnv attribute*), 44

action_head_distribution() (*maze.distributions.distribution_mapper.DistributionMapper method*), 157

action_space() (*maze.core.env.maze_env.MazeEnv property*), 44

action_space() (*maze.core.env.structured_env_spaces_mixin.StructuredEnvSpacesMixin property*), 43

action_space() (*maze.core.wrappers.discretize_actions_wrapper.DiscretizeActionsWrapper property*), 60

action_space() (*maze.core.wrappers.no_dict_action_wrapper.NoDictActionWrapper property*), 58

action_space() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWrapper property*), 56

action_space() (*maze.core.wrappers.sorted_spaces_wrapper.SortedSpacesWrapper property*), 55

action_space() (*maze.core.wrappers.split_actions_wrapper.SplitActionsWrapper property*), 59

action_space() (*maze.train.parallelization.distributed_env.dummy_dis property*), 205

action_spaces_dict (*maze.train.trainers.imitation.bc_loss.BCLoss attribute*), 201

action_spaces_dict() (*maze.core.env.maze_env.MazeEnv property*), 44

action_spaces_dict() (*maze.core.env.structured_env_spaces_mixin.StructuredEnvSpace property*), 43

action_spaces_dict() (*maze.core.wrappers.discretize_actions_wrapper.DiscretizeActior property*), 60

action_spaces_dict() (*maze.core.wrappers.no_dict_action_wrapper.NoDictActionWrap property*), 58

action_spaces_dict() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWra property*), 56

action_spaces_dict() (*maze.core.wrappers.split_actions_wrapper.SplitActionsWrapper property*), 59

action_spaces_dict() (*maze.train.parallelization.distributed_env.dummy_distributed_en property*), 205

ActionCandidates (*class in maze.core.agent_integration.maze_action_candidates*), 121

ActionConversionCandidatesInterface (*class in maze.core.agent_integration.maze_action_candidates*), 121

ActionConversionInterface (*class in maze.core.env.action_conversion*), 48

ActionMaskingBlock (*class in maze.perception.blocks.general.action_masking*), 135

ActionWrapper (*class in maze.core.wrappers.wrapper*), 52

actor_id() (*maze.core.agent_integration.external_core_env.ExternalCo method*), 122 actor_id() (*maze.core.env.core_env.CoreEnv method*), 42

actor_id() (*maze.core.env.maze_env.MazeEnv method*), 44

actor_id() (*maze.core.env.structured_env.StructuredEnv method*), 42

actor_id() (*maze.core.wrappers.maze_gym_env_wrapper.GymCoreEnv method*), 70

actor_id() (*maze.train.parallelization.distributed_env.dummy_distributed_env.DummyStructuredDistributedEnv method*), 205

ActorAgent (*class in maze.train.parallelization.distributed_actors.actor*), 206

actors_batch_size (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 179

Adam (*class in maze.train.trainers.es.optimizers.adam*), 193

add_event() (*maze.core.log_stats.log_stats.LogStatsAggregator method*), 83

add_value() (*maze.core.log_stats.log_stats.LogStatsAggregator method*), 83

AgentExecution (*class in maze.core.agent_integration.agent_execution*), 118

AgentIntegration (*class in maze.core.agent_integration.agent_integration*), 119

aggregate() (*maze.train.parallelization.observation_aggregator.ObservationAggregator method*), 203

AGGREGATION_FUNCS (*maze.core.rendering.events_stats_renderer.EventStatsRenderer attribute*), 91

AlgorithmConfig (*class in maze.train.trainers.common.training_runner*), 168

append() (*maze.core.events.event_collection.EventCollection method*), 75

append() (*maze.core.log_events.step_event_log.StepEventLog method*), 77

arg_to_collection() (*maze.core.utils.registry.Registry method*), 164

arg_to_obj() (*maze.core.utils.registry.Registry method*), 164

arguments() (*maze.core.rendering.renderer.Renderer static method*), 90

atanh (*class in maze.distributions.utils*), 157

B

BaseDistributedActors (*class in maze.train.parallelization.distributed_actors.distributed_actors*), 207

BaseDistributedEnv (*class in maze.train.parallelization.distributed_env.distributed_env*), 204

BaseEnv (*class in maze.core.env.base_env*), 41

BaseModelBuilder (*class in maze.perception.builders.base*), 138

BaseModelComposer (*class in maze.perception.models.model_composer*), 140

BasePolicyComposer (*class in maze.perception.models.policies.base_policy_composer*), 145

BaseStateActionCriticComposer (*class in maze.perception.models.critics.base_state_action_critic_compose*
149

BaseStateCriticComposer (*class in maze.perception.models.critics.base_state_critic_composer*), 147

BaseWorker (*class in maze.train.parallelization.base_worker*), 203

BaseWorkerOutput (*class in maze.train.parallelization.base_worker*), 203

batch_outputs_time_major (*class in maze.train.trainers.impala.impala_batching*), 183

batch_size (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgor*
attribute), 199

batch_size (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorith*
attribute), 175

BCAlgorithmConfig (*class in maze.train.trainers.imitation.bc_algorithm_config*), 199

BCEvaluator (*class in maze.train.trainers.imitation.bc_evaluator*), 200

BCLoss (*class in maze.train.trainers.imitation.bc_loss*), 201

BCTrainer (*class in maze.train.trainers.imitation.bc_trainer*), 198

BernoulliProbabilityDistribution (*class in maze.distributions.bernoulli*), 158

BestModelSelection (*class in maze.train.trainers.common.model_selection.best_model_selectio*
168

BetaProbabilityDistribution (*class in maze.distributions.beta*), 160

bootstrap_returns() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 110

box_mean_abs_deviation() (*maze.train.trainers.imitation.imitation_events.ImitationEvents method*), 195

broadcast_updated_policy() (*maze.train.parallelization.distributed_actors.distributed_actors.BaseDistributedActors method*), 207
build() (*maze.core.rendering.notebook_event_logs_viewer.NotebookEventLogsViewer method*), 93
build() (*maze.core.rendering.notebook_trajectory_viewer.NotebookTrajectoryViewer method*), 93
build_layer_dict() (*maze.perception.blocks.feed_forward.dense.DenseBlock method*), 126
build_layer_dict() (*maze.perception.blocks.feed_forward.graph_attention.GraphAttentionBlock method*), 129
build_layer_dict() (*maze.perception.blocks.feed_forward.graph_conv.GraphConvBlock method*), 128
build_layer_dict() (*maze.perception.blocks.feed_forward.strided_conv.StridedConvolutionBlock method*), 128
build_layer_dict() (*maze.perception.blocks.feed_forward.vgg_conv.VGGConvolutionBlock method*), 127
build_obj() (*maze.core.utils.registry.Registry class method*), 164

C

calculate_kpis() (*maze.core.log_events.kpi_calculator.KpiCalculator method*), 77
calculate_loss() (*maze.train.trainers.imitation.bc_loss.BCLoss method*), 201
callable_from_path() (*maze.core.utils.registry.Registry class method*), 165
CategoricalProbabilityDistribution (*class in maze.distributions.categorical*), 158
check_model_config() (*maze.perception.models.custom_model_composer.CustomModelComposer class method*), 144
check_model_config() (*maze.perception.models.model_composer.BaseModelComposer class method*), 140
check_model_config() (*maze.perception.models.template_model_composer.TemplateModelComposer class method*), 142
class_type_from_module_name() (*maze.core.utils.registry.Registry method*), 165
clear_abort() (*maze.train.trainers.es.distributed.es_rollout_wrapper.ESRolloutWorkerWrapper method*), 194
clip_range (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
clone_from() (*maze.core.env.simulated_env_mixin.SimulatedEnvMixin method*), 47
clone_from() (*maze.core.wrappers.maze_gym_env_wrapper.GymMaze method*), 70
close() (*maze.core.agent_integration.external_core_env.ExternalCoreEn method*), 122
close() (*maze.core.env.base_env.BaseEnv method*), 41
close() (*maze.core.env.core_env.CoreEnv method*), 42
close() (*maze.core.env.maze_env.MazeEnv method*), 44
close() (*maze.core.rendering.events_stats_renderer.EventStatsRenderer method*), 91
close() (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper method*), 53
close() (*maze.core.wrappers.maze_gym_env_wrapper.GymCoreEnv method*), 70
close() (*maze.core.wrappers.time_limit_wrapper.TimeLimitWrapper method*), 54
close() (*maze.train.parallelization.distributed_env.dummy_distributed_e method*), 205
collect_modules() (*maze.core.utils.registry.Registry method*), 165
collect_outputs() (*maze.train.parallelization.distributed_actors.distributed_actors.I method*), 207
compute_action() (*maze.core.agent.default_policy.DefaultPolicy method*), 107
compute_action() (*maze.core.agent.dummy_cartpole_policy.DummyC method*), 108
compute_action() (*maze.core.agent.flat_policy.FlatPolicy method*), 104
compute_action() (*maze.core.agent.policy.Policy method*), 104
compute_action() (*maze.core.agent.random_policy.RandomPolicy method*), 107
compute_action() (*maze.core.agent.torch_policy.TorchPolicy method*), 105
compute_action_distribution() (*maze.core.agent.torch_policy.TorchPolicy method*), 105
compute_action_logits_entropy_dist() (*maze.core.agent.torch_policy.TorchPolicy method*), 105
compute_action_with_logits() (*maze.core.agent.torch_policy.TorchPolicy method*), 106
compute_gradient_norm (*class in maze.train.utils.train_utils*), 202
compute_logits_dict() (*maze.core.agent.torch_policy.TorchPolicy method*), 106
compute_return() (*maze.core.agent.torch_state_critic.TorchStateCriti method*), 110
compute_sigmoid_bias (*class in maze.perception.weight_init*), 155

`compute_state_action_value_step()` (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic method*), 113

`compute_state_action_values_step()` (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic method*), 113

`compute_top_action_candidates()` (*maze.core.agent.default_policy.DefaultPolicy method*), 107

`compute_top_action_candidates()` (*maze.core.agent.dummy_cartpole_policy.DummyCartPolePolicy method*), 108

`compute_top_action_candidates()` (*maze.core.agent.flat_policy.FlatPolicy method*), 104

`compute_top_action_candidates()` (*maze.core.agent.policy.Policy method*), 104

`compute_top_action_candidates()` (*maze.core.agent.random_policy.RandomPolicy method*), 108

`compute_top_action_candidates()` (*maze.core.agent.torch_policy.TorchPolicy method*), 106

`ConcatenationBlock` (*class in maze.perception.blocks.general.concat*), 131

`ConcatModelBuilder` (*class in maze.perception.builders.concat*), 139

`ContinuousActionEvents` (*class in maze.core.log_events.action_events*), 80

`convert_to_numpy` (*class in maze.perception.perception_utils*), 154

`convert_to_torch` (*class in maze.perception.perception_utils*), 153

`core_env` (*maze.core.env.maze_env.MazeEnv attribute*), 44

`CoreEnv` (*class in maze.core.env.core_env*), 42

`CorrelationBlock` (*class in maze.perception.blocks.general.correlation*), 131

`create_categorical_plot` (*class in maze.core.log_events.log_create_figure_functions*), 80

`create_distributed_rollouts()` (*maze.train.trainers.es.es_runners.ESDevRunner method*), 191

`create_distributed_rollouts()` (*maze.train.trainers.es.es_runners.ESMasterRunner method*), 191

`create_event_topic()` (*maze.core.events.event_service.EventService method*), 75

`create_event_topic()` (*maze.core.events.pubsub.Pubsub method*), 75

`create_event_topic()` (*maze.core.log_stats.log_stats.LogStatsAggregator method*), 84

`create_histogram` (*class in maze.core.log_events.log_create_figure_functions*), 80

`create_relative_bar_plot` (*class in maze.core.log_events.log_create_figure_functions*), 80

`create_violin_distribution` (*class in maze.core.log_events.log_create_figure_functions*), 80

`create_widget()` (*maze.core.rendering.renderer_args.IntRangeArg method*), 95

`create_widget()` (*maze.core.rendering.renderer_args.OptionsArrayAr method*), 95

`create_widget()` (*maze.core.rendering.renderer_args.RendererArg method*), 94

`critic()` (*maze.perception.models.critics.base_state_action_critic_comp property*), 150

`critic()` (*maze.perception.models.critics.base_state_critic_composer.Ba property*), 147

`critic()` (*maze.perception.models.critics.delta_state_critic_composer.D property*), 149

`critic()` (*maze.perception.models.critics.shared_state_action_critics_co property*), 150

`critic()` (*maze.perception.models.critics.shared_state_critic_composer. property*), 147

`critic()` (*maze.perception.models.critics.step_state_action_critic_comp property*), 151

`critic()` (*maze.perception.models.critics.step_state_critic_composer.Ste property*), 148

`critic()` (*maze.perception.models.custom_model_composer.CustomMod property*), 144

`critic()` (*maze.perception.models.model_composer.BaseModelCompose property*), 140

`critic()` (*maze.perception.models.template_model_composer.TemplateM property*), 142

`critic_burn_in_epochs` (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmCon attribute*), 173

`critic_burn_in_epochs` (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmCo attribute*), 176

`critic_grad_norm()` (*maze.train.trainers.common.actor_critic.actor_critic_events.Mu method*), 171

`critic_grad_norm()` (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEve method*), 181

`critic_value()` (*maze.train.trainers.common.actor_critic.actor_critic method*), 171

critic_value() (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEvents method*), 181
critic_value_loss() (*maze.train.trainers.common.actor_critic.actor_critic_events.MultiStepActorCriticEvents method*), 171
critic_value_loss() (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEvents method*), 181
CustomModelComposer (*class in maze.perception.models.custom_model_composer*), 143

D

data_loader (*maze.train.trainers.imitation.bc_trainer.BCTrainer attribute*), 198
DataLoadWorker (*class in maze.train.trainers.imitation.parallel_loaded_im_data_set*), 197
dataset (*maze.train.trainers.imitation.imitation_runners.ImitationRunner attribute*), 196
DefaultPolicy (*class in maze.core.agent.default_policy*), 107
define_episode_stats (*class in maze.core.log_stats.event_decorators*), 87
define_epoch_stats (*class in maze.core.log_stats.event_decorators*), 88
define_plot (*class in maze.core.log_stats.event_decorators*), 89
define_stats_grouping (*class in maze.core.log_stats.event_decorators*), 88
define_step_stats (*class in maze.core.log_stats.event_decorators*), 86
DeltaStateCriticComposer (*class in maze.perception.models.critics.delta_state_critic_composer*), 149
DenseBlock (*class in maze.perception.blocks.feed_forward.dense*), 126
deterministic_eval (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 173
deterministic_eval (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
deterministic_eval (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
deterministic_sample() (*maze.distributions.bernoulli.BernoulliProbabilityDistribution method*), 158
deterministic_sample() (*maze.distributions.beta.BetaProbabilityDistribution method*), 160
deterministic_sample() (*maze.distributions.categorical.CategoricalProbabilityDistributio method*), 158
deterministic_sample() (*maze.distributions.dict.DictProbabilityDistribution method*), 161
deterministic_sample() (*maze.distributions.distribution.ProbabilityDistribution method*), 155
deterministic_sample() (*maze.distributions.gaussian.DiagonalGaussianProbabilityDistri method*), 159
deterministic_sample() (*maze.distributions.multi_categorical.MultiCategoricalProbabilit method*), 160
deterministic_sample() (*maze.distributions.squashed_gaussian.SquashedGaussianProbab method*), 159
device (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmCon attribute*), 173
device (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithmC attribute*), 200
device (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgo attribute*), 180
device (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmCon attribute*), 176
device() (*maze.core.agent.torch_actor_critic.TorchActorCritic property*), 117
device() (*maze.core.agent.torch_model.TorchModel property*), 117
device() (*maze.core.agent.torch_state_action_critic.TorchStateActionCr property*), 113
device() (*maze.core.agent.torch_state_critic.TorchStateCritic property*), 110
DiagonalGaussianProbabilityDistribution (*class in maze.distributions.gaussian*), 159
DictActionWrapper (*class in maze.core.wrappers.dict_action_wrapper*), 58
DictObservationWrapper (*class in maze.core.wrappers.dict_observation_wrapper*), 57
DictProbabilityDistribution (*class in maze.distributions.dict*), 161
discrete_accuracy() (*maze.train.trainers.imitation.imitation_events.ImitationEvents method*), 195
DiscreteActionEvents (*class in maze.core.log_events.action_events*), 79
DiscretizeActionsWrapper (*class in maze.core.wrappers.discretize_actions_wrapper*), 59
distribution_mapper() (*maze.perception.models.model_composer.BaseModelComposer property*), 140

DistributionMapper (*class in maze.distributions.distribution_mapper*), 156

DummyCartPolePolicy (*class in maze.core.agent.dummy_cartpole_policy*), 108

DummyStructuredDistributedEnv (*class in maze.train.parallelization.distributed_env.dummy_distributed_env*), 205

dump_statistics() (*maze.core.wrappers.observation_normalization.observation_normalization_wrapper.ObservationNormalizationWrapper method*), 65

E

entropy() (*maze.distributions.bernoulli.BernoulliProbabilityDistribution method*), 158

entropy() (*maze.distributions.beta.BetaProbabilityDistribution method*), 160

entropy() (*maze.distributions.dict.DictProbabilityDistribution method*), 161

entropy() (*maze.distributions.distribution.ProbabilityDistribution method*), 155

entropy() (*maze.distributions.gaussian.DiagonalGaussianProbabilityDistribution method*), 159

entropy() (*maze.distributions.multi_categorical.MultiCategoricalProbabilityDistribution method*), 160

entropy() (*maze.distributions.squashed_gaussian.SquashedGaussianProbabilityDistribution method*), 159

entropy() (*maze.distributions.torch_dist.TorchProbabilityDistribution method*), 156

entropy_coef (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 173

entropy_coef (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180

entropy_coef (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176

env_time() (*maze.core.trajectory_recorder.step_record.StepRecord property*), 96

EnvFactory (*class in maze.core.utils.config_utils*), 163

EnvironmentContext (*class in maze.core.env.environment_context*), 49

EPISODE (*maze.core.log_stats.log_stats.LogStatsLevel attribute*), 83

episode_id() (*maze.core.env.environment_context.EnvironmentContext property*), 49

EpisodeEventLog (*class in maze.core.log_events.episode_event_log*), 77

EpisodeRecord (*class in maze.core.trajectory_recorder.episode_record*), 96

EpisodeRecorder (*class in maze.core.rollout.parallel_rollout_runner*), 102

EpisodeStatsReport (*class in maze.core.rollout.parallel_rollout_runner*), 103

EPOCH (*maze.core.log_stats.log_stats.LogStatsLevel attribute*), 83

epoch_length (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgor attribute*), 173

epoch_length (*maze.train.trainers.impala.impala_algorithm_config.Imp attribute*), 180

epoch_length (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgo attribute*), 176

ESAbortException (*class in maze.train.trainers.es.distributed.es_rollout_wrapper*), 194

ESAlgorithmConfig (*class in maze.train.trainers.es.es_algorithm_config*), 190

ESDevRunner (*class in maze.train.trainers.es.es_runners*), 191

ESDistributedRollouts (*class in maze.train.trainers.es.distributed.es_distributed_rollouts*), 193

ESDummyDistributedRollouts (*class in maze.train.trainers.es.distributed.es_dummy_distributed_rollouts*), 193

ESEvents (*class in maze.train.trainers.es.es_events*), 190

ESMasterRunner (*class in maze.train.trainers.es.es_runners*), 191

ESRolloutResult (*class in maze.train.trainers.es.distributed.es_distributed_rollouts*), 193

ESRolloutWorkerWrapper (*class in maze.train.trainers.es.distributed.es_rollout_wrapper*), 194

estimate_observation_normalization_statistics (*class in maze.core.wrappers.observation_normalization.observat*), 68

estimate_statistics() (*maze.core.wrappers.observation_normalization.observation_nor method*), 65

estimate_stats() (*maze.core.wrappers.observation_normalization.no method*), 66

estimate_stats() (*maze.core.wrappers.observation_normalization.no method*), 69

estimate_stats() (*maze.core.wrappers.observation_normalization.no method*), 69

estimated_queue_sizes() (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEve method*), 181

ESTrainer (*class in maze.train.trainers.es.es_trainer*), 189

eval() (*maze.core.agent.torch_actor_critic.TorchActorCritic method*), 117

eval() (*maze.core.agent.torch_model.TorchModel method*), 117
eval() (*maze.core.agent.torch_policy.TorchPolicy method*), 106
eval() (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic method*), 114
eval() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 110
eval_concurrency (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
eval_every_k_iterations (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithmConfig attribute*), 200
eval_repeats (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 173
eval_repeats (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
eval_repeats (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
evaluate() (*maze.train.trainers.common.actor_critic.actor_critic_trainer.MultiStepActorCritic method*), 170
evaluate() (*maze.train.trainers.imitation.bc_evaluator.BCEvaluator method*), 200
evaluate() (*maze.train.trainers.imitation.imitation_evaluator.ImitationEvaluator method*), 196
evaluate() (*maze.train.trainers.impala.impala_learner.ImpalaLearner method*), 182
evaluate() (*maze.train.trainers.impala.impala_trainer.MultiStepIMPALA method*), 177
event() (*maze.core.log_stats.log_stats.LogStatsKey property*), 89
event_log() (*maze.core.rollout.parallel_rollout_runner.EpisodeStatsReport property*), 103
event_topic_factory (*class in maze.core.events.event_topic_factory*), 74
EventCollection (*class in maze.core.events.event_collection*), 75
EventEnvMixin (*class in maze.core.env.event_env_mixin*), 46
EventRecord (*class in maze.core.events.event_record*), 76
EventRow (*class in maze.core.log_events.log_events_writer_tsv*), 79
EventScope (*class in maze.core.events.event_service*), 74
EventService (*class in maze.core.events.event_service*), 75
EventService.TopicInfo (*class in maze.core.events.event_service*), 75
EventStatsRenderer (*class in maze.core.rendering.events_stats_renderer*), 91
exception() (*maze.core.rollout.parallel_rollout_runner.ExceptionReport property*), 103
ExceptionReport (*class in maze.core.rollout.parallel_rollout_runner*), 103
extend() (*maze.core.events.event_collection.EventCollection method*), 75
extend() (*maze.core.log_events.step_event_log.StepEventLog method*), 77
ExternalCoreEnv (*class in maze.core.agent_integration.external_core_env*), 122
ExternalCoreEnvRewardAggregator (*class in maze.core.agent_integration.external_core_env*), 122

F

finish_rollout() (*maze.core.agent_integration.agent_integration.Ag method*), 120
flat_structured_observations (*class in maze.perception.perception_utils*), 153
flat_structured_shapes (*class in maze.core.utils.structured_env_utils*), 163
flat_structured_space (*class in maze.core.utils.structured_env_utils*), 163
FlatPolicy (*class in maze.core.agent.flat_policy*), 104
FlattenBlock (*class in maze.perception.blocks.general.flatten*), 130
FlattenConcatBaseNet (*class in maze.perception.models.built_in.flatten_concat*), 152
FlattenConcatPolicyNet (*class in maze.perception.models.built_in.flatten_concat*), 152
FlattenConcatStateValueNet (*class in maze.perception.models.built_in.flatten_concat*), 152
FlattenDenseBlock (*class in maze.perception.blocks.joint_blocks.flatten_dense*), 136
FlattenPreProcessor (*class in maze.core.wrappers.observation_preprocessing.preprocessors.fla* 62
forward() (*maze.perception.blocks.base.PerceptionBlock method*), 124
forward() (*maze.perception.blocks.general.action_masking.ActionMask method*), 135
forward() (*maze.perception.blocks.general.concat.ConcatenationBlock method*), 131
forward() (*maze.perception.blocks.general.correlation.CorrelationBlock method*), 131
forward() (*maze.perception.blocks.general.flatten.FlattenBlock method*), 130
forward() (*maze.perception.blocks.general.functional.FunctionalBlock method*), 132
forward() (*maze.perception.blocks.general.masked_global_pooling.Mas method*), 132

forward() (*maze.perception.blocks.general.multi_index_slicing.MultiIndexSlicingBlock method*), 133

forward() (*maze.perception.blocks.general.repeat_to_match.RepeatToMatchBlock method*), 133

forward() (*maze.perception.blocks.general.self_attention_conv.SelfAttentionConvBlock method*), 134

forward() (*maze.perception.blocks.general.self_attention_seq.SelfAttentionSeqBlock method*), 135

forward() (*maze.perception.blocks.general.slice.SliceBlock method*), 135

forward() (*maze.perception.blocks.inference.InferenceBlock method*), 125

forward() (*maze.perception.blocks.joint_blocks.flatten_dense.FlattenDenseBlock method*), 136

forward() (*maze.perception.blocks.joint_blocks.lstm_last_step.LSTMLastStepBlock method*), 138

forward() (*maze.perception.blocks.joint_blocks.strided_conv_dense.StridedConvolutionDenseBlock method*), 138

forward() (*maze.perception.blocks.joint_blocks.vgg_conv_dense.VGGConvolutionDenseBlock method*), 137

forward() (*maze.perception.blocks.joint_blocks.vgg_conv_gap.VGGConvolutionGAPBlock method*), 137

forward() (*maze.perception.blocks.shape_normalization.ShapeNormalizationBlock method*), 125

forward() (*maze.perception.models.built_in.flatten_concat.FlattenConcatPolicyNet method*), 152

forward() (*maze.perception.models.built_in.flatten_concat.FlattenConcatStateValueNet method*), 152

from_importance_weights (*class in maze.train.trainers.impala.impala_vtrace*), 187

from_logits (*class in maze.train.trainers.impala.impala_vtrace*), 185

from_observation_space() (*maze.perception.builders.base.BaseModelBuilder method*), 139

from_observation_space() (*maze.perception.builders.concat.ConcatModelBuilder method*), 139

FunctionalBlock (*class in maze.perception.blocks.general.functional*), 131

G

gae_lambda (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 174

gae_lambda (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176

gamma (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 174

gamma (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180

gamma (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConf attribute*), 176

generate_evaluation() (*maze.train.trainers.es.distributed.es_rollout_wrapper.ESRolloutW method*), 194

generate_rollouts() (*maze.train.trainers.es.distributed.es_distributed_rollouts.ESDist method*), 193

generate_rollouts() (*maze.train.trainers.es.distributed.es_dummy_distributed_rollout method*), 193

generate_training() (*maze.train.trainers.es.distributed.es_rollout_wrapper.ESRolloutW method*), 194

get() (*maze.train.trainers.es.es_shared_noise_table.SharedNoiseTable method*), 192

get_dict_obj_attr_names() (*maze.train.parallelization.base_worker.BaseWorkerOutput method*), 204

get_env_time() (*maze.core.env.maze_env.MazeEnv method*), 44

get_env_time() (*maze.core.env.time_env_mixin.TimeEnvMixin method*), 46

get_episode_id() (*maze.core.env.maze_env.MazeEnv method*), 44

get_episode_id() (*maze.core.env.recordable_env_mixin.RecordableE method*), 45

get_epoch_stats_aggregator() (*maze.train.parallelization.distributed_actors.distributed_actors. method*), 207

get_flat_parameters (*class in maze.train.trainers.es.es_utils*), 194

get_interfaces() (*maze.core.agent_integration.external_core_env.Ex method*), 122

get_interfaces() (*maze.core.events.pubsub.Subscriber method*), 73

get_interfaces() (*maze.core.wrappers.maze_gym_env_wrapper.Gym method*), 71

get_kpi_calculator() (*maze.core.agent_integration.external_core_env.ExternalCoreEnv method*), 123

get_kpi_calculator() (*maze.core.env.core_env.CoreEnv method*), 42

get_kpi_calculator() (*maze.core.env.event_env_mixin.EventEnvMixin method*), 46

get_kpi_calculator() (*maze.core.env.maze_env.MazeEnv method*), 44

get_last_episode_data() (*maze.core.rollout.parallel_rollout_runner.EpisodeRecorder method*), 102

get_list_obj_attr_names() (*maze.train.parallelization.base_worker.BaseWorkerOutput*

static method), 204
get_log_rhos (*class in maze.train.trainers.impala.impala_vtrace*), 188
get_maze_action() (*maze.core.agent_integration.agent_integration.AgentIntegration method*), 120
get_maze_action() (*maze.core.env.maze_env.MazeEnv method*), 44
get_maze_action() (*maze.core.env.recordable_env_mixin.RecordableEnvMixin method*), 45
get_maze_state() (*maze.core.agent_integration.external_core_env.ExternalCoreEnv method*), 123
get_maze_state() (*maze.core.env.core_env.CoreEnv method*), 42
get_maze_state() (*maze.core.env.maze_env.MazeEnv method*), 44
get_maze_state() (*maze.core.env.recordable_env_mixin.RecordableEnvMixin method*), 46
get_maze_state() (*maze.core.wrappers.maze_gym_env_wrapper.GymCoreEnv method*), 70
get_num_of_parameters() (*maze.perception.blocks.base.PerceptionBlock method*), 124
get_observation_and_action_dicts() (*maze.core.env.maze_env.MazeEnv method*), 44
get_observation_and_action_dicts() (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper method*), 53
get_observation_and_action_dicts() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWrapper method*), 56
get_observation_and_action_dicts() (*maze.core.wrappers.observation_logging_wrapper.ObservationLoggingWrapper method*), 54
get_observation_and_action_dicts() (*maze.core.wrappers.observation_stack_wrapper.ObservationStackWrapper method*), 57
get_observation_and_action_dicts() (*maze.core.wrappers.random_reset_wrapper.RandomResetWrapper method*), 55
get_observation_and_action_dicts() (*maze.core.wrappers.sorted_spaces_wrapper.SortedSpacesWrapper method*), 55
get_observation_and_action_dicts() (*maze.core.wrappers.time_limit_wrapper.TimeLimitWrapper method*), 54
get_observation_and_action_dicts() (*maze.core.wrappers.wrapper.ActionWrapper method*), 52
get_observation_and_action_dicts() (*maze.core.wrappers.wrapper.ObservationWrapper method*), 51
get_observation_and_action_dicts() (*maze.core.wrappers.wrapper.RewardWrapper method*), 52
get_observation_and_action_dicts() (*maze.core.wrappers.wrapper.Wrapper method*), 50
get_renderer() (*maze.core.agent_integration.external_core_env.Exter method*), 123
get_renderer() (*maze.core.env.core_env.CoreEnv method*), 42
get_renderer() (*maze.core.env.maze_env.MazeEnv method*), 44
get_renderer() (*maze.core.env.recordable_env_mixin.RecordableEnvM method*), 46
get_renderer() (*maze.core.wrappers.maze_gym_env_wrapper.GymCo method*), 71
get_serializable_components() (*maze.core.agent_integration.external_core_env.ExternalCoreEnv method*), 123
get_serializable_components() (*maze.core.env.core_env.CoreEnv method*), 42
get_serializable_components() (*maze.core.env.serializable_env_mixin.SerializableEnvMixin method*), 46
get_serializable_components() (*maze.core.wrappers.maze_gym_env_wrapper.GymCoreEnv method*), 71
get_statistics() (*maze.core.wrappers.observation_normalization.no method*), 66
get_statistics() (*maze.core.wrappers.observation_normalization.ob method*), 65
get_stats() (*maze.core.log_stats.log_stats_env.LogStatsEnv method*), 81
get_stats() (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper method*), 53
get_stats() (*maze.train.parallelization.distributed_env.dummy_distribu method*), 205
get_stats_logger (*class in maze.core.log_stats.log_stats*), 86
get_stats_value() (*maze.core.log_stats.log_stats_env.LogStatsEnv method*), 82
get_stats_value() (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper method*), 53
get_stats_value() (*maze.train.parallelization.distributed_actors.distributed_actors.I method*), 207
get_stats_value() (*maze.train.parallelization.distributed_env.dummy_distributed_en method*), 205
get_step_events() (*maze.core.agent_integration.external_core_env.ExternalCoreEnv*

method), 123

get_step_events() (*maze.core.env.core_env.CoreEnv method*), 42

get_step_events() (*maze.core.env.event_env_mixin.EventEnvMixin method*), 46

get_step_events() (*maze.core.env.maze_env.MazeEnv method*), 44

get_tensor_obj_attr_names() (*maze.train.parallelization.base_worker.BaseWorkerOutput static method*), 204

get_trajectory_files() (*maze.train.trainers.imitation.in_memory_data_set.InMemoryImitationDataSet static method*), 197

GlobalAveragePoolingBlock (*class in maze.perception.blocks.general.gap*), 132

GlobalLogState (*class in maze.core.log_stats.log_stats*), 85

GraphAttentionBlock (*class in maze.perception.blocks.feed_forward.graph_attention*), 129

GraphConvBlock (*class in maze.perception.blocks.feed_forward.graph_conv*), 128

group() (*maze.core.log_stats.log_stats.LogStatsKey property*), 89

GymActionConversion (*class in maze.core.wrappers.maze_gym_env_wrapper*), 72

GymCoreEnv (*class in maze.core.wrappers.maze_gym_env_wrapper*), 70

GymMazeEnv (*class in maze.core.wrappers.maze_gym_env_wrapper*), 70

GymObservationConversion (*class in maze.core.wrappers.maze_gym_env_wrapper*), 71

GymRenderer (*class in maze.core.wrappers.maze_gym_env_wrapper*), 71

GymRewardAggregator (*class in maze.core.wrappers.maze_gym_env_wrapper*), 71

H

histogram (*class in maze.core.log_stats.reducer_functions*), 89

hook_on_log_step (*maze.core.log_stats.log_stats.GlobalLogState attribute*), 85

I

imitation_events (*maze.train.trainers.imitation.bc_trainer.BCTrainer attribute*), 198

ImitationEvaluator (*class in maze.train.trainers.imitation.imitation_evaluator*), 196

ImitationEvents (*class in maze.train.trainers.imitation.imitation_events*), 195

ImitationRunner (*class in maze.train.trainers.imitation.imitation_runners*), 196

ImpalaAlgorithmConfig (*class in maze.train.trainers.impala.impala_algorithm_config*), 179

ImpalaLearner (*class in maze.train.trainers.impala.impala_learner*), 182

increment_env_step() (*maze.core.env.environment_context.EnvironmentContext method*), 49

increment_log_step (*class in maze.core.log_stats.log_stats*), 85

InferenceBlock (*class in maze.perception.blocks.inference*), 125

InferenceGraph (*class in maze.perception.blocks.inference*), 125

init_env_and_agent() (*maze.core.rollout.rollout_runner.RolloutRunner static method*), 99

InMemoryImitationDataSet (*class in maze.train.trainers.imitation.in_memory_data_set*), 197

interface_to_subscribers (*maze.core.events.pubsub.Pubsub attribute*), 74

IntRangeArg (*class in maze.core.rendering.renderer_args*), 95

is_actor_done() (*maze.core.agent_integration.external_core_env.Exte method*), 123

is_actor_done() (*maze.core.env.core_env.CoreEnv method*), 43

is_actor_done() (*maze.core.env.maze_env.MazeEnv method*), 44

is_actor_done() (*maze.core.env.structured_env.StructuredEnv method*), 42

is_actor_done() (*maze.core.wrappers.maze_gym_env_wrapper.GymC method*), 71

is_actor_done() (*maze.train.parallelization.distributed_env.dummy_d method*), 205

is_initialized() (*maze.core.wrappers.observation_normalization.no method*), 66

iterate_event_records() (*maze.core.events.event_service.EventService method*), 75

K

KeyboardControlledTrajectoryViewer (*class in maze.core.rendering.keyboard_controlled_trajectory_viewer*), 94

kl() (*maze.distributions.bernoulli.BernoulliProbabilityDistribution method*), 158

kl() (*maze.distributions.beta.BetaProbabilityDistribution method*), 160

kl() (*maze.distributions.dict.DictProbabilityDistribution method*), 161

kl() (*maze.distributions.distribution.ProbabilityDistribution method*), 155

kl() (*maze.distributions.gaussian.DiagonalGaussianProbabilityDistribution method*), 159

kl() (*maze.distributions.multi_categorical.MultiCategoricalProbabilityDistribution method*), 160

kl() (*maze.distributions.squashed_gaussian.SquashedGaussianProbabilityDistribution method*), 159

kl() (*maze.distributions.torch_dist.TorchProbabilityDistribution method*), 156

KpiCalculator (*class in maze.core.log_events.kpi_calculator*), 77

L

l2_penalty (*maze.train.trainers.es.es_algorithm_config.ESAlgorithmConfig attribute*), 190

last_stats (*maze.core.log_stats.log_stats.LogStatsAggregator attribute*), 84

last_stats_step (*maze.core.log_stats.log_stats.LogStatsAggregator attribute*), 84

learner_rollout_on_agent_output() (*maze.train.trainers.impala.impala_learner.ImpalaLearner method*), 183

learning_rate() (*maze.train.trainers.common.actor_critic.actor_critic_events.MultiStepActorCriticEvents method*), 171

list_to_dict (*class in maze.core.utils.config_utils*), 163

load() (*maze.perception.models.space_config.SpacesConfig class method*), 144

load_episode_record() (*maze.train.trainers.imitation.in_memory_data_set.InMemoryImitationDataSet static method*), 197

load_state() (*maze.train.trainers.common.actor_critic.actor_critic_trainer.MultiStepActorCritic method*), 170

load_state() (*maze.train.trainers.common.trainer.Trainer method*), 166

load_state() (*maze.train.trainers.es.es_trainer.ESTrainer method*), 189

load_state() (*maze.train.trainers.imitation.bc_trainer.BCTrainer method*), 198

load_state() (*maze.train.trainers.impala.impala_trainer.MultiStepIMPALA method*), 177

load_state_dict() (*maze.core.agent.torch_actor_critic.TorchActorCritic method*), 117

load_state_dict() (*maze.core.agent.torch_model.TorchModel method*), 117

load_state_dict() (*maze.core.agent.torch_policy.TorchPolicy method*), 106

load_state_dict() (*maze.core.agent.torch_state_action_critic.TorchStateActionCriti method*), 114

load_state_dict() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 111

load_state_dict() (*maze.train.trainers.common.actor_critic.actor_critic_trainer.Mu method*), 170

load_state_dict() (*maze.train.trainers.es.es_trainer.ESTrainer method*), 189

load_state_dict() (*maze.train.trainers.imitation.bc_trainer.BCTrainer method*), 199

load_state_dict() (*maze.train.trainers.impala.impala_trainer.MultiStepIMPALA method*), 177

log_prob() (*maze.distributions.bernoulli.BernoulliProbabilityDistributi method*), 158

log_prob() (*maze.distributions.beta.BetaProbabilityDistribution method*), 160

log_prob() (*maze.distributions.categorical.CategoricalProbabilityDistr method*), 158

log_prob() (*maze.distributions.dict.DictProbabilityDistribution method*), 161

log_prob() (*maze.distributions.distribution.ProbabilityDistribution method*), 155

log_prob() (*maze.distributions.gaussian.DiagonalGaussianProbabilityI method*), 159

log_prob() (*maze.distributions.multi_categorical.MultiCategoricalProb method*), 161

log_prob() (*maze.distributions.squashed_gaussian.SquashedGaussianP method*), 159

log_prob() (*maze.distributions.torch_dist.TorchProbabilityDistribution method*), 156

log_probs_from_logits_and_actions_and_spaces (*class in maze.train.trainers.impala.impala_vtrace*), 184

log_stats (*class in maze.core.log_stats.log_stats*), 85

LogEventsWriter (*class in maze.core.log_events.log_events_writer*), 78

LogEventsWriterRegistry (*class in maze.core.log_events.log_events_writer_registry*), 78

LogEventsWriterTSV (*class in*

maze.core.log_events.log_events_writer_tsv), 78
logits_dict_to_distribution() (*maze.core.agent.torch_policy.TorchPolicy method*), 106
logits_dict_to_distribution() (*maze.distributions.distribution_mapper.DistributionMapper method*), 157
LogStats (*in module maze.core.log_stats.log_stats*), 90
LogStatsAggregator (*class in maze.core.log_stats.log_stats*), 83
LogStatsConsumer (*class in maze.core.log_stats.log_stats*), 83
LogStatsEnv (*class in maze.core.log_stats.log_stats_env*), 81
LogStatsGroup (*in module maze.core.log_stats.log_stats*), 89
LogStatsKey (*class in maze.core.log_stats.log_stats*), 89
LogStatsLevel (*class in maze.core.log_stats.log_stats*), 83
LogStatsLogger (*class in maze.core.log_stats.log_stats*), 85
LogStatsValue (*in module maze.core.log_stats.log_stats*), 89
LogStatsWrapper (*class in maze.core.wrappers.log_stats_wrapper*), 53
LogStatsWriter (*class in maze.core.log_stats.log_stats*), 84
LogStatsWriterConsole (*class in maze.core.log_stats.log_stats_writer_console*), 82
LogStatsWriterTensorboard (*class in maze.core.log_stats.log_stats_writer_tensorboard*), 82
loss (*maze.train.trainers.imitation.bc_trainer.BCTrainer attribute*), 199
lr (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 174
lr (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
lr (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
LSTMBlock (*class in maze.perception.blocks.recurrent.lstm*), 130
LSTMLastStepBlock (*class in maze.perception.blocks.joint_blocks.lstm_last_step*), 138

M

make_env_from_hydra (*class in maze.core.utils.config_utils*), 164
make_gym_maze_env (*class in maze.core.wrappers.maze_gym_env_wrapper*), 70
make_module_init_normc (*class in maze.perception.weight_init*), 154
make_normalized_env_factory (*class in maze.core.wrappers.observation_normalization.observation_norr*), 68
MaskedGlobalPoolingBlock (*class in maze.perception.blocks.general.masked_global_pooling*), 132
max_episode_steps (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithm attribute*), 200
max_epochs (*maze.train.trainers.es.es_algorithm_config.ESAlgorithmCo attribute*), 190
max_grad_norm (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlg attribute*), 174
max_grad_norm (*maze.train.trainers.impala.impala_algorithm_config.Ir attribute*), 180
max_grad_norm (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlg attribute*), 176
max_steps (*maze.train.trainers.es.es_algorithm_config.ESAlgorithmCon attribute*), 190
maze_env (*maze.core.env.maze_env.MazeEnv attribute*), 44
maze_run (*class in maze.maze_cli*), 98
maze_to_space() (*maze.core.env.action_conversion.ActionConversion method*), 48
maze_to_space() (*maze.core.env.observation_conversion.Observation method*), 47
maze_to_space() (*maze.core.wrappers.maze_gym_env_wrapper.GymA method*), 72
maze_to_space() (*maze.core.wrappers.maze_gym_env_wrapper.GymC method*), 71
MazeActionCandidates (*class in maze.core.agent_integration.maze_action_candidates*), 121
MazeActionType (*in module maze.core.env.maze_action*), 48
MazeEnv (*class in maze.core.env.maze_env*), 43
MazeStateType (*in module maze.core.env.maze_state*), 48
MeanZeroStdOneObservationNormalizationStrategy (*class in maze.core.wrappers.observation_normalization.normaliz*), 69
metadata (*maze.core.env.maze_env.MazeEnv attribute*), 44
ModelConfig (*class in maze.train.trainers.common.training_runner*), 168
ModelSelectionBase (*class in maze.train.trainers.common.model_selection.model_selection_ba*), 168
MonitoringSetup (*class in*

maze.core.trajectory_recorder.monitoring_setup), 97
multi_binary_accuracy() (*maze.train.trainers.imitation.imitation_events.ImitationEvents method*), 195
MultiCategoricalProbabilityDistribution (*class in maze.distributions.multi_categorical*), 160
MultiIndexSlicingBlock (*class in maze.perception.blocks.general.multi_index_slicing*), 133
MultiStepA2C (*class in maze.train.trainers.a2c.a2c_trainer*), 172
MultiStepActorCritic (*class in maze.train.trainers.common.actor_critic.actor_critic_trainer*), 170
MultiStepActorCriticEvents (*class in maze.train.trainers.common.actor_critic.actor_critic_events*), 171
MultiStepIMPALA (*class in maze.train.trainers.impala.impala_trainer*), 177
MultiStepIMPALAEvents (*class in maze.train.trainers.impala.impala_events*), 181
MultiStepPPO (*class in maze.train.trainers.ppo.ppo_trainer*), 174

N

n_epochs (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 174
n_epochs (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithmConfig attribute*), 200
n_epochs (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
n_epochs (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
n_eval_episodes (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithmConfig attribute*), 200
n_eval_rollouts (*maze.train.trainers.es.es_runners.ESDevRunner attribute*), 191
n_eval_workers (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithmConfig attribute*), 200
n_optimization_epochs (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
n_rollout_steps (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 174
n_rollout_steps (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
n_rollout_steps (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
n_rollouts_per_update (*maze.train.trainers.es.es_algorithm_config.ESAlgorithmConfig attribute*), 190
n_timesteps_per_update (*maze.train.trainers.es.es_algorithm_config.ESAlgorithmConfig attribute*), 190
needs_state() (*maze.core.agent.default_policy.DefaultPolicy method*), 107
needs_state() (*maze.core.agent.dummy_cartpole_policy.DummyCartP method*), 108
needs_state() (*maze.core.agent.policy.Policy method*), 105
needs_state() (*maze.core.agent.random_policy.RandomPolicy method*), 108
needs_state() (*maze.core.agent.torch_policy.TorchPolicy method*), 107
neg_log_prob() (*maze.distributions.dict.DictProbabilityDistribution method*), 161
neg_log_prob() (*maze.distributions.distribution.ProbabilityDistributio method*), 155
neg_log_prob() (*maze.distributions.multi_categorical.MultiCategorica method*), 161
NoDictActionWrapper (*class in maze.core.wrappers.no_dict_action_wrapper*), 58
NoDictObservationWrapper (*class in maze.core.wrappers.no_dict_observation_wrapper*), 58
NoDictSpacesWrapper (*class in maze.core.wrappers.no_dict_spaces_wrapper*), 56
noise_stddev (*maze.train.trainers.es.es_algorithm_config.ESAlgorithm attribute*), 190
noop_action() (*maze.core.env.action_conversion.ActionConversionInte method*), 48
normalization_samples (*maze.train.trainers.common.training_runner.TrainingRunner attribute*), 167
normalize_and_process_value() (*maze.core.wrappers.observation_normalization.normalization_s method*), 66
normalize_value() (*maze.core.wrappers.observation_normalization.normalization_s method*), 67
normalize_value() (*maze.core.wrappers.observation_normalization.normalization_s method*), 69
normalize_value() (*maze.core.wrappers.observation_normalization.normalization_s method*), 69
normalized_forward() (*maze.perception.blocks.feed_forward.dense.DenseBlock method*), 126
normalized_forward() (*maze.perception.blocks.feed_forward.graph_attention.GraphAtte method*), 129

normalized_forward() (*maze.perception.blocks.feed_forward.graph_conv.GraphConvBlock method*), 128
normalized_forward() (*maze.perception.blocks.feed_forward.strided_conv.StridedConvolutionBlock method*), 128
normalized_forward() (*maze.perception.blocks.feed_forward.vgg_conv.VGGConvolutionBlock method*), 127
normalized_forward() (*maze.perception.blocks.general.gap.GlobalAveragePoolingBlock method*), 132
normalized_forward() (*maze.perception.blocks.recurrent.lstm.LSTMBlock method*), 130
normalized_forward() (*maze.perception.blocks.shape_normalization.ShapeNormalizationBlock method*), 125
normalized_space() (*maze.core.wrappers.observation_normalization.normalization_strategies.base.ObservationNormalizationStrategy method*), 67
NotebookEventLogsViewer (*class in maze.core.rendering.notebook_event_logs_viewer*), 93
NotebookTrajectoryViewer (*class in maze.core.rendering.notebook_trajectory_viewer*), 93
notify_event() (*maze.core.events.event_service.EventScope method*), 74
notify_event() (*maze.core.events.event_service.EventService method*), 75
notify_event() (*maze.core.events.pubsub.Pubsub method*), 74
notify_event() (*maze.core.events.pubsub.Subscriber method*), 73
notify_next_step() (*maze.core.events.event_service.EventScope method*), 74
notify_next_step() (*maze.core.events.event_service.EventService method*), 75
notify_next_step() (*maze.core.events.pubsub.Pubsub method*), 74
num_actors (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
num_critics() (*maze.core.agent.torch_state_action_critic.TorchSharedStateActionCritic property*), 115
num_critics() (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic property*), 114
num_critics() (*maze.core.agent.torch_state_action_critic.TorchStepStateActionCritic property*), 116
num_critics() (*maze.core.agent.torch_state_critic.TorchDeltaStateCritic property*), 112
num_critics() (*maze.core.agent.torch_state_critic.TorchSharedStateCri property*), 111
num_critics() (*maze.core.agent.torch_state_critic.TorchStateCritic property*), 111
num_critics() (*maze.core.agent.torch_state_critic.TorchStepStateCriti property*), 111
num_params() (*maze.core.agent.torch_model.TorchModel property*), 117

O

observation() (*maze.core.wrappers.dict_observation_wrapper.DictOb method*), 57
observation() (*maze.core.wrappers.no_dict_observation_wrapper.NoI method*), 58
observation() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictS method*), 56
observation() (*maze.core.wrappers.observation_normalization.observ method*), 65
observation() (*maze.core.wrappers.observation_preprocessing.prepro method*), 61
observation() (*maze.core.wrappers.observation_stack_wrapper.Obser method*), 57
observation() (*maze.core.wrappers.wrapper.ObservationWrapper method*), 51
observation_conversion() (*maze.core.env.maze_env.MazeEnv property*), 45
observation_conversion_dict (*maze.core.env.maze_env.MazeEnv attribute*), 45
observation_original() (*maze.core.log_events.observation_events.ObservationEvents method*), 79
observation_processed() (*maze.core.log_events.observation_events.ObservationEvents method*), 79
observation_space() (*maze.core.env.maze_env.MazeEnv property*), 45
observation_space() (*maze.core.env.structured_env_spaces_mixin.StructuredEnvSpace property*), 43
observation_space() (*maze.core.wrappers.no_dict_observation_wrapper.NoDictObser property*), 58
observation_space() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWra property*), 56
observation_space() (*maze.core.wrappers.sorted_spaces_wrapper.SortedSpacesWrapp property*), 56
observation_space() (*maze.train.parallelization.distributed_env.dummy_distributed_en property*), 205

observation_spaces_dict() (*maze.core.env.maze_env.MazeEnv property*), 45
observation_spaces_dict() (*maze.core.env.structured_env_spaces_mixin.StructuredEnvSpacesMixin property*), 43
observation_spaces_dict() (*maze.core.wrappers.no_dict_observation_wrapper.NoDictObservationWrapper property*), 58
observation_spaces_dict() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWrapper property*), 56
observation_spaces_dict() (*maze.train.parallelization.distributed_env.dummy_distributed_env.DummyStructuredDistributedEnv property*), 205
observation_spaces_to_in_shapes (*class in maze.perception.perception_utils*), 153
ObservationAggregator (*class in maze.train.parallelization.observation_aggregator*), 203
ObservationConversionInterface (*class in maze.core.env.observation_conversion*), 47
ObservationEvents (*class in maze.core.log_events.observation_events*), 79
ObservationLoggingWrapper (*class in maze.core.wrappers.observation_logging_wrapper*), 54
ObservationNormalizationStrategy (*class in maze.core.wrappers.observation_normalization.normalization_strategies.base*), 66
ObservationNormalizationWrapper (*class in maze.core.wrappers.observation_normalization.observation_normalization_wrapper*), 65
ObservationStackWrapper (*class in maze.core.wrappers.observation_stack_wrapper*), 57
ObservationWrapper (*class in maze.core.wrappers.wrapper*), 51
obtain_normalization_statistics (*class in maze.core.wrappers.observation_normalization.observation_normalization_utils*), 67
on_log_step_increment() (*maze.core.log_stats.log_stats_writer_tensorboard.LogStatsWriterTensorboard method*), 82
OneHotPreProcessor (*class in maze.core.wrappers.observation_preprocessing.preprocessors.one_hot*), 62
Optimizer (*class in maze.train.trainers.es.optimizers.base_optimizer*), 192
optimizer (*maze.train.trainers.es.es_algorithm_config.ESAlgorithmConfig attribute*), 190
optimizer (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithmConfig attribute*), 200
optimizer (*maze.train.trainers.imitation.bc_trainer.BCTrainer attribute*), 199
OptionsArrayArg (*class in maze.rendering.renderer_args*), 95
out_shapes() (*maze.perception.blocks.base.PerceptionBlock method*), 124
output_name() (*maze.core.log_stats.log_stats.LogStatsKey property*), 89
override (*class in maze.core.annotations*), 162

P

ParallelLoadedImitationDataset (*class in maze.train.trainers.imitation.parallel_loaded_im_data_set*), 196
ParallelRolloutRunner (*class in maze.core.rollout.parallel_rollout_runner*), 101
ParallelRolloutWorker (*class in maze.core.rollout.parallel_rollout_runner*), 102
parameters() (*maze.core.agent.torch_actor_critic.TorchActorCritic method*), 117
parameters() (*maze.core.agent.torch_model.TorchModel method*), 117
parameters() (*maze.core.agent.torch_policy.TorchPolicy method*), 107
parameters() (*maze.core.agent.torch_state_action_critic.TorchStateAct method*), 114
parameters() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 111
patience (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmC attribute*), 174
patience (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAl attribute*), 180
patience (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithm attribute*), 176
per_critic_parameters() (*maze.core.agent.torch_state_action_critic.TorchStateActionCriti method*), 114
PerceptionBlock (*class in maze.perception.blocks.base*), 124
Policy (*class in maze.core.agent.policy*), 104
policy (*maze.train.trainers.imitation.bc_trainer.BCTrainer attribute*), 199
policy() (*maze.perception.models.custom_model_composer.CustomMod property*), 144
policy() (*maze.perception.models.model_composer.BaseModelCompose property*), 140
policy() (*maze.perception.models.policies.base_policy_composer.BaseP property*), 145
policy() (*maze.perception.models.policies.probabilistic_policy_compose property*), 146

policy() (*maze.perception.models.template_model_composer.TemplateModelComposer property*), 142
policy_entropy() (*maze.train.trainers.common.actor_critic.actor_critic_events.MultiStepActorCriticEvents method*), 171
policy_entropy() (*maze.train.trainers.imitation.imitation_events.ImitationEvents method*), 195
policy_entropy() (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEvents method*), 181
policy_grad_norm() (*maze.train.trainers.common.actor_critic.actor_critic_events.MultiStepActorCriticEvents method*), 171
policy_grad_norm() (*maze.train.trainers.es.es_events.ESEvents method*), 190
policy_grad_norm() (*maze.train.trainers.imitation.imitation_events.ImitationEvents method*), 195
policy_grad_norm() (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEvents method*), 181
policy_l2_norm() (*maze.train.trainers.imitation.imitation_events.ImitationEvents method*), 195
policy_loss() (*maze.train.trainers.common.actor_critic.actor_critic_events.MultiStepActorCriticEvents method*), 171
policy_loss() (*maze.train.trainers.imitation.imitation_events.ImitationEvents method*), 195
policy_loss() (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEvents method*), 181
policy_loss_coef (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig attribute*), 174
policy_loss_coef (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180
policy_loss_coef (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig attribute*), 176
policy_norm() (*maze.train.trainers.es.es_events.ESEvents method*), 190
POST_PROCESSING_FUNCS (*maze.core.rendering.events_stats_renderer.EventStatsRenderer attribute*), 91
PPOAlgorithmConfig (*class in maze.train.trainers.ppo.ppo_algorithm_config*), 175
predict_next_q_values() (*maze.core.agent.torch_state_action_critic.TorchSharedStateActionCritic method*), 115
predict_next_q_values() (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic method*), 114
predict_next_q_values() (*maze.core.agent.torch_state_action_critic.TorchStepStateActionCritic method*), 116
predict_q_values() (*maze.core.agent.state_action_critic.StateActionCritic method*), 112
predict_q_values() (*maze.core.agent.torch_state_action_critic.TorchSharedStateActio method*), 115
predict_q_values() (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic method*), 114
predict_q_values() (*maze.core.agent.torch_state_action_critic.TorchStepStateActionC method*), 116
predict_value() (*maze.core.agent.state_critic.StateCritic method*), 109
predict_value() (*maze.core.agent.torch_state_critic.TorchDeltaStateC method*), 112
predict_value() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 111
predict_values() (*maze.core.agent.state_critic.StateCritic method*), 109
predict_values() (*maze.core.agent.torch_state_critic.TorchDeltaStat method*), 112
predict_values() (*maze.core.agent.torch_state_critic.TorchSharedSta method*), 111
predict_values() (*maze.core.agent.torch_state_critic.TorchStepState method*), 111
PreProcessingWrapper (*class in maze.core.wrappers.observation_preprocessing.preprocessing_wr*), 61
PreProcessor (*class in maze.core.wrappers.observation_preprocessing.preprocessors.ba*), 61
ProbabilisticPolicyComposer (*class in maze.perception.models.policies.probabilistic_policy_composer*), 146
ProbabilityDistribution (*class in maze.distributions.distribution*), 155
process() (*maze.core.wrappers.observation_preprocessing.preprocesso method*), 61
process() (*maze.core.wrappers.observation_preprocessing.preprocesso method*), 62
process() (*maze.core.wrappers.observation_preprocessing.preprocesso method*), 62
process() (*maze.core.wrappers.observation_preprocessing.preprocesso method*), 63
process() (*maze.core.wrappers.observation_preprocessing.preprocesso method*), 64
process() (*maze.core.wrappers.observation_preprocessing.preprocesso method*), 63
process() (*maze.core.wrappers.observation_preprocessing.preprocesso method*), 64
processed_shape() (*maze.core.wrappers.observation_preprocessing.preprocessors.ba method*), 61
processed_shape() (*maze.core.wrappers.observation_preprocessing.preprocessors.fla method*), 62

processed_shape() (*maze.core.wrappers.observation_preprocessing.preprocessors.one_hot.OneHotPreProcessor method*), 62

processed_shape() (*maze.core.wrappers.observation_preprocessing.preprocessors.resize_img.ResizeImgPreProcessor method*), 63

processed_shape() (*maze.core.wrappers.observation_preprocessing.preprocessors.rgb2gray.Rgb2GrayPreProcessor method*), 64

processed_shape() (*maze.core.wrappers.observation_preprocessing.preprocessors.transpose.TransposePreProcessor method*), 63

processed_shape() (*maze.core.wrappers.observation_preprocessing.preprocessors.unsqueeze.UnSqueezePreProcessor method*), 64

processed_space() (*maze.core.wrappers.observation_preprocessing.preprocessors.base.PreProcessor method*), 61

processed_space() (*maze.core.wrappers.observation_preprocessing.preprocessors.flatten.FlattenPreProcessor method*), 62

processed_space() (*maze.core.wrappers.observation_preprocessing.preprocessors.one_hot.OneHotPreProcessor method*), 62

processed_space() (*maze.core.wrappers.observation_preprocessing.preprocessors.resize_img.ResizeImgPreProcessor method*), 63

processed_space() (*maze.core.wrappers.observation_preprocessing.preprocessors.rgb2gray.Rgb2GrayPreProcessor method*), 64

processed_space() (*maze.core.wrappers.observation_preprocessing.preprocessors.transpose.TransposePreProcessor method*), 63

processed_space() (*maze.core.wrappers.observation_preprocessing.preprocessors.unsqueeze.UnSqueezePreProcessor method*), 64

Pubsub (*class in maze.core.events.pubsub*), 73

Q

query_events() (*maze.core.events.event_collection.EventCollection method*), 75

query_events() (*maze.core.events.pubsub.Subscriber method*), 73

query_events() (*maze.core.log_events.episode_event_log.EpisodeEventLog method*), 77

queue_out_of_sync_factor (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig attribute*), 180

R

random_split() (*maze.train.trainers.imitation.in_memory_data_set.InMemoryImitationDataSet method*), 198

RandomPolicy (*class in maze.core.agent.random_policy*), 107

RandomResetWrapper (*class in maze.core.wrappers.random_reset_wrapper*), 55

RangeZeroOneObservationNormalizationStrategy (*class in maze.core.wrappers.observation_normalization.normaliz*), 69

re_init_networks() (*maze.core.agent.torch_state_action_critic.TorchStateActionCriti method*), 114

read_config (*class in maze.core.utils.config_utils*), 163

real_time() (*maze.train.trainers.es.es_events.ESEvents method*), 190

receive() (*maze.core.log_stats.log_stats.LogStatsAggregator method*), 84

receive() (*maze.core.log_stats.log_stats.LogStatsConsumer method*), 83

receive() (*maze.core.log_stats.log_stats.LogStatsLogger method*), 85

receive() (*maze.core.rollout.parallel_rollout_runner.EpisodeRecorder method*), 102

record_event_logs() (*maze.core.log_events.log_events_writer_registry.LogEventsWrite class method*), 78

record_trajectory_data() (*maze.core.trajectory_recorder.trajectory_writer_registry.Trajecto class method*), 95

RecordableEnvMixin (*class in maze.core.env.recordable_env_mixin*), 45

reduce() (*maze.core.log_stats.log_stats.LogStatsAggregator method*), 84

register_consumer() (*maze.core.log_stats.log_stats.LogStatsAggregator method*), 84

register_log_stats_writer() (*class in maze.core.log_stats.log_stats*), 85

register_new_observation_normalization_strategy() (*maze.core.wrappers.observation_normalization.observation_nor class method*), 65

register_subscriber() (*maze.core.events.pubsub.Pubsub method*), 74

register_writer() (*maze.core.log_events.log_events_writer_registry.LogEventsWrite class method*), 78

register_writer() (*maze.core.trajectory_recorder.trajectory_writer_registry.Trajecto class method*), 95

Registry (*class in maze.core.utils.registry*), 164

render() (*maze.core.env.render_env_mixin.RenderEnvMixin method*), 45

render() (*maze.core.rendering.keyboard_controlled_trajectory_viewer.K method*), 94

render() (*maze.core.rendering.notebook_event_logs_viewer.NotebookEv*

method), 93
render() (*maze.core.rendering.notebook_trajectory_viewer.NotebookTrajectoryViewer method*), 93
render() (*maze.core.rendering.renderer.Renderer method*), 90
render() (*maze.core.wrappers.maze_gym_env_wrapper.GymRenderer method*), 71
render_current_episode_stats() (*maze.core.rendering.events_stats_renderer.EventStatsRenderer method*), 91
render_stats() (*maze.core.rendering.step_stats_renderer.StepStatsRenderer static method*), 91
render_stats() (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper method*), 53
render_timeline_stat() (*maze.core.rendering.events_stats_renderer.EventStatsRenderer static method*), 92
RenderEnvMixin (*class in maze.core.env.render_env_mixin*), 45
Renderer (*class in maze.core.rendering.renderer*), 90
RendererArg (*class in maze.core.rendering.renderer_args*), 94
RepeatToMatchBlock (*class in maze.perception.blocks.general.repeat_to_match*), 133
required_logits_shape() (*maze.distributions.bernoulli.BernoulliProbabilityDistribution class method*), 158
required_logits_shape() (*maze.distributions.beta.BetaProbabilityDistribution class method*), 160
required_logits_shape() (*maze.distributions.categorical.CategoricalProbabilityDistribution class method*), 158
required_logits_shape() (*maze.distributions.distribution_mapper.DistributionMapper method*), 157
required_logits_shape() (*maze.distributions.gaussian.DiagonalGaussianProbabilityDistribution class method*), 159
required_logits_shape() (*maze.distributions.multi_categorical.MultiCategoricalProbabilityDistribution class method*), 161
required_logits_shape() (*maze.distributions.squashed_gaussian.SquashedGaussianProbabilityDistribution class method*), 159
required_logits_shape() (*maze.distributions.torch_dist.TorchProbabilityDistribution class method*), 156
reset() (*maze.core.agent_integration.external_core_env.ExternalCoreEnv method*), 123
reset() (*maze.core.env.base_env.BaseEnv method*), 41
reset() (*maze.core.env.core_env.CoreEnv method*), 43
reset() (*maze.core.env.maze_env.MazeEnv method*), 45
reset() (*maze.core.events.pubsub.Subscriber method*), 73
reset() (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper method*), 54
reset() (*maze.core.wrappers.maze_gym_env_wrapper.GymCoreEnv method*), 71
reset() (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWra method*), 56
reset() (*maze.core.wrappers.observation_stack_wrapper.ObservationSta method*), 57
reset() (*maze.core.wrappers.random_reset_wrapper.RandomResetWrap method*), 55
reset() (*maze.core.wrappers.time_limit_wrapper.TimeLimitWrapper method*), 55
reset() (*maze.core.wrappers.wrapper.ObservationWrapper method*), 51
reset() (*maze.train.parallelization.distributed_env.distributed_env.Base method*), 204
reset() (*maze.train.parallelization.distributed_env.dummy_distributed_e method*), 205
reset() (*maze.train.parallelization.observation_aggregator.Observation method*), 203
reset_env_episode() (*maze.core.env.environment_context.EnvironmentContext method*), 49
ResizingPreProcessor (*class in maze.core.wrappers.observation_preprocessing.preprocessors.res*), 63
reverse_action() (*maze.core.wrappers.dict_action_wrapper.DictActi method*), 58
reverse_action() (*maze.core.wrappers.discretize_actions_wrapper.D method*), 60
reverse_action() (*maze.core.wrappers.no_dict_action_wrapper.NoD method*), 58
reverse_action() (*maze.core.wrappers.no_dict_spaces_wrapper.NoD method*), 56
reverse_action() (*maze.core.wrappers.split_actions_wrapper.SplitAc method*), 59
reverse_action() (*maze.core.wrappers.wrapper.ActionWrapper method*), 52
reward() (*maze.core.wrappers.reward_clipping_wrapper.RewardClippin method*), 60
reward() (*maze.core.wrappers.reward_scaling_wrapper.RewardScaling method*), 60
reward() (*maze.core.wrappers.wrapper.RewardWrapper method*), 52
reward_clipping (*maze.train.trainers.impala.impala_algorithm_confi attribute*), 180
reward_range (*maze.core.env.maze_env.MazeEnv attribute*), 45
RewardAggregatorInterface (*class in maze.core.env.reward*), 48
RewardClippingWrapper (*class in*

maze.core.wrappers.reward_clipping_wrapper), 60
RewardScalingWrapper (*class in maze.core.wrappers.reward_scaling_wrapper*), 60
RewardWrapper (*class in maze.core.wrappers.wrapper*), 52
Rgb2GrayPreProcessor (*class in maze.core.wrappers.observation_preprocessing.preprocessors.rgb2gray*), 64
rollout() (*maze.train.parallelization.base_worker.BaseWorker method*), 203
rollout() (*maze.train.parallelization.distributed_actors.actor.ActorAgent method*), 206
rollout() (*maze.train.trainers.es.distributed.es_rollout_wrapper.ESRolloutWorkerWrapper method*), 194
RolloutRunner (*class in maze.core.rollout.rollout_runner*), 99
run() (*maze.core.rollout.parallel_rollout_runner.ParallelRolloutWorker static method*), 102
run() (*maze.core.rollout.rollout_runner.RolloutRunner method*), 99
run() (*maze.runner.Runner method*), 98
run() (*maze.train.trainers.common.training_runner.TrainingRunner method*), 167
run() (*maze.train.trainers.es.es_runners.ESMasterRunner method*), 191
run() (*maze.train.trainers.imitation.imitation_runners.ImitationRunner method*), 196
run() (*maze.train.trainers.imitation.parallel_loaded_im_data_set.DataLoadWorker static method*), 197
run_interaction_maze() (*maze.core.rollout.rollout_runner.RolloutRunner static method*), 100
run_rollout_maze() (*maze.core.agent_integration.agent_execution.AgentExecution method*), 118
run_with() (*maze.core.rollout.parallel_rollout_runner.ParallelRolloutRunner method*), 101
run_with() (*maze.core.rollout.rollout_runner.RolloutRunner method*), 100
run_with() (*maze.core.rollout.sequential_rollout_runner.SequentialRolloutRunner method*), 101
Runner (*class in maze.runner*), 98

S

sample() (*maze.distributions.beta.BetaProbabilityDistribution method*), 160
sample() (*maze.distributions.dict.DictProbabilityDistribution method*), 161
sample() (*maze.distributions.distribution.ProbabilityDistribution method*), 155
sample() (*maze.distributions.multi_categorical.MultiCategoricalProbabilityDistribution method*), 161
sample() (*maze.distributions.squashed_gaussian.SquashedGaussianProb method*), 159
sample() (*maze.distributions.torch_dist.TorchProbabilityDistribution method*), 156
sample_index() (*maze.train.trainers.es.es_shared_noise_table.SharedN method*), 192
save() (*maze.perception.blocks.inference.InferenceGraph method*), 126
save() (*maze.perception.models.space_config.SpacesConfig method*), 144
save_models() (*maze.perception.models.model_composer.BaseModelC method*), 140
seed() (*maze.core.agent_integration.external_core_env.ExternalCoreEnv method*), 123
seed() (*maze.core.env.base_env.BaseEnv method*), 41
seed() (*maze.core.env.core_env.CoreEnv method*), 43
seed() (*maze.core.env.maze_env.MazeEnv method*), 45
seed() (*maze.core.wrappers.maze_gym_env_wrapper.GymCoreEnv method*), 71
seed() (*maze.core.wrappers.time_limit_wrapper.TimeLimitWrapper method*), 55
seed() (*maze.train.parallelization.distributed_env.distributed_env.BaseDi method*), 204
seed() (*maze.train.parallelization.distributed_env.dummy_distributed_en method*), 206
SelfAttentionConvBlock (*class in maze.perception.blocks.general.self_attention_conv*), 134
SelfAttentionSeqBlock (*class in maze.perception.blocks.general.self_attention_seq*), 134
SequentialRolloutRunner (*class in maze.core.rollout.sequential_rollout_runner*), 100
SerializableEnvMixin (*class in maze.core.env.serializable_env_mixin*), 46
SerializedTorchPolicy (*class in maze.core.agent.serialized_torch_policy*), 108
set_abort() (*maze.train.trainers.es.distributed.es_rollout_wrapper.ESR method*), 194
set_actor_id() (*maze.core.agent_integration.external_core_env.Exter method*), 123
set_flat_parameters (*class in maze.train.trainers.es.es_utils*), 195
set_is_actor_done() (*maze.core.agent_integration.external_core_env.ExternalCoreEnv method*), 123
set_max_episode_steps() (*maze.core.wrappers.time_limit_wrapper.TimeLimitWrapper method*), 55
set_normalization_statistics() (*maze.core.wrappers.observation_normalization.observation_nor method*), 65

set_observation_collection() (*maze.core.wrappers.observation_normalization.observation_normalization_wrapper.ObservationNormalizationWrapper method*), 66
set_random_states (*class in maze.core.utils.seeding*), 162
set_statistics() (*maze.core.wrappers.observation_normalization.normalization_strategies.base.ObservationNormalizationStrategy method*), 67
setup() (*maze.train.trainers.es.optimizers.adam.Adam method*), 193
setup() (*maze.train.trainers.es.optimizers.base_optimizer.Optimizer method*), 192
setup() (*maze.train.trainers.es.optimizers.sgd.SGD method*), 192
SGD (*class in maze.train.trainers.es.optimizers.sgd*), 192
ShapeNormalizationBlock (*class in maze.perception.blocks.shape_normalization*), 124
shared_noise_table_size (*maze.train.trainers.es.es_runners.ESMasterRunner attribute*), 191
SharedNoiseTable (*class in maze.train.trainers.es.es_shared_noise_table*), 192
SharedStateActionCriticComposer (*class in maze.perception.models.critics.shared_state_action_critics_composer*), 150
SharedStateCriticComposer (*class in maze.perception.models.critics.shared_state_critic_composer*), 147
show() (*maze.perception.blocks.inference.InferenceGraph method*), 126
SimpleEventLoggingSetup (*class in maze.core.log_events.log_events_utils*), 79
SimpleStatsLoggingSetup (*class in maze.utils.log_stats_utils*), 166
SimpleTrajectoryRecordingSetup (*class in maze.core.trajectory_recorder.trajectory_utils*), 97
SimulatedEnvMixin (*class in maze.core.env.simulated_env_mixin*), 47
SliceBlock (*class in maze.perception.blocks.general.slice*), 135
SortedSpacesWrapper (*class in maze.core.wrappers.sorted_spaces_wrapper*), 55
space() (*maze.core.agent_integration.maze_action_candidates.ActionConversionCandidatesInterface method*), 121
space() (*maze.core.env.action_conversion.ActionConversionInterface method*), 48
space() (*maze.core.env.observation_conversion.ObservationConversionInterface method*), 47
space() (*maze.core.wrappers.maze_gym_env_wrapper.GymActionConversion method*), 72
space() (*maze.core.wrappers.maze_gym_env_wrapper.GymObservationConversion method*), 71
space_to_maze() (*maze.core.agent_integration.maze_action_candidat method*), 121
space_to_maze() (*maze.core.env.action_conversion.ActionConversionI method*), 48
space_to_maze() (*maze.core.env.observation_conversion.ObservationConversionInterface method*), 47
space_to_maze() (*maze.core.wrappers.maze_gym_env_wrapper.GymA method*), 72
space_to_maze() (*maze.core.wrappers.maze_gym_env_wrapper.GymO method*), 71
spaces_config_dump_file (*maze.train.trainers.common.training_runner.TrainingRunner attribute*), 167
SpacesConfig (*class in maze.perception.models.space_config*), 144
spec (*maze.core.env.maze_env.MazeEnv attribute*), 45
SplitActionsWrapper (*class in maze.core.wrappers.split_actions_wrapper*), 59
SquashedGaussianProbabilityDistribution (*class in maze.distributions.squashed_gaussian*), 159
stack_numpy_dict_list (*class in maze.train.utils.train_utils*), 201
stack_torch_dict_list (*class in maze.train.utils.train_utils*), 202
start() (*maze.train.parallelization.distributed_actors.distributed_actors method*), 208
state_dict() (*maze.core.agent.torch_actor_critic.TorchActorCritic method*), 117
state_dict() (*maze.core.agent.torch_model.TorchModel method*), 117
state_dict() (*maze.core.agent.torch_policy.TorchPolicy method*), 107
state_dict() (*maze.core.agent.torch_state_action_critic.TorchStateAct method*), 114
state_dict() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 111
state_dict_dump_file (*maze.train.trainers.common.training_runner.TrainingRunner attribute*), 167
StateActionCritic (*class in maze.core.agent.state_action_critic*), 112
StateActionCriticComposer (*in module maze.perception.models.critics*), 151
StateCritic (*class in maze.core.agent.state_critic*), 109
StateCriticComposer (*in module maze.perception.models.critics*), 149
stats() (*maze.core.rollout.parallel_rollout_runner.EpisodeStatsReport property*), 103
STEP (*maze.core.log_stats.log_stats.LogStatsLevel attribute*), 83

`step()` (*maze.core.agent_integration.external_core_env.ExternalCoreEnv method*), 123
`step()` (*maze.core.env.base_env.BaseEnv method*), 41
`step()` (*maze.core.env.core_env.CoreEnv method*), 43
`step()` (*maze.core.env.maze_env.MazeEnv method*), 45
`step()` (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper method*), 54
`step()` (*maze.core.wrappers.maze_gym_env_wrapper.GymCoreEnv method*), 71
`step()` (*maze.core.wrappers.no_dict_spaces_wrapper.NoDictSpacesWrapper method*), 56
`step()` (*maze.core.wrappers.observation_logging_wrapper.ObservationLoggingWrapper method*), 54
`step()` (*maze.core.wrappers.time_limit_wrapper.TimeLimitWrapper method*), 55
`step()` (*maze.core.wrappers.wrapper.ActionWrapper method*), 52
`step()` (*maze.core.wrappers.wrapper.ObservationWrapper method*), 51
`step()` (*maze.core.wrappers.wrapper.RewardWrapper method*), 52
`step()` (*maze.train.parallelization.distributed_env.distributed_env.BaseDistributedEnv method*), 204
`step()` (*maze.train.parallelization.distributed_env.dummy_distributed_env.DummyStructuredDistributedEnv method*), 206
`step_id()` (*maze.core.trajectory_recorder.step_record.StepRecord property*), 96
`step_without_observation()` (*maze.core.env.simulated_env_mixin.SimulatedEnvMixin method*), 47
`StepEventLog` (*class in maze.core.log_events.step_event_log*), 77
`StepRecord` (*class in maze.core.trajectory_recorder.step_record*), 96
`StepStateActionCriticComposer` (*class in maze.perception.models.critics.step_state_action_critic_composer*), 151
`StepStateCriticComposer` (*class in maze.perception.models.critics.step_state_critic_composer*), 148
`StepStatsRenderer` (*class in maze.core.rendering.step_stats_renderer*), 91
`stop()` (*maze.train.parallelization.distributed_actors.distributed_actors.BaseDistributedActors method*), 208
`StridedConvolutionBlock` (*class in maze.perception.blocks.feed_forward.strided_conv*), 127
`StridedConvolutionDenseBlock` (*class in maze.perception.blocks.joint_blocks.strided_conv_dense*), 137
`StructuredEnv` (*class in maze.core.env.structured_env*), 41
`StructuredEnvSpacesMixin` (*class in maze.core.env.structured_env_spaces_mixin*), 43
`Subscriber` (*class in maze.core.events.pubsub*), 73
`subscribers` (*maze.core.events.pubsub.Pubsub attribute*), 74

T

`tag()` (*maze.core.wrappers.observation_preprocessing.preprocessors.base method*), 62
`template_perception_net()` (*maze.perception.models.template_model_composer.TemplateMod method*), 142
`template_policy_net()` (*maze.perception.models.template_model_composer.TemplateMod method*), 142
`template_q_value_net()` (*maze.perception.models.template_model_composer.TemplateMod method*), 142
`template_value_net()` (*maze.perception.models.template_model_composer.TemplateMod method*), 142
`TemplateModelComposer` (*class in maze.perception.models.template_model_composer*), 141
`tensor_clamp` (*class in maze.distributions.utils*), 157
`time_backprob()` (*maze.train.trainers.impala.impala_events.MultiStep method*), 181
`time_collecting_actors()` (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEve method*), 182
`time_dequeuing_actors()` (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEve method*), 182
`time_epoch()` (*maze.train.trainers.common.actor_critic.actor_critic_ev method*), 171
`time_learner_rollout()` (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEve method*), 182
`time_loss_computation()` (*maze.train.trainers.impala.impala_events.MultiStepIMPALAEve method*), 182
`time_rollout()` (*maze.train.trainers.common.actor_critic.actor_critic_ method*), 171
`time_update()` (*maze.train.trainers.common.actor_critic.actor_critic_e method*), 171
`TimeEnvMixin` (*class in maze.core.env.time_env_mixin*), 46
`TimeLimitWrapper` (*class in maze.core.wrappers.time_limit_wrapper*), 54
`to()` (*maze.core.agent.torch_actor_critic.TorchActorCritic method*), 118
`to()` (*maze.core.agent.torch_model.TorchModel method*), 117

to() (*maze.core.agent.torch_policy.TorchPolicy method*), 107
to() (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic method*), 114
to() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 111
to() (*maze.train.parallelization.base_worker.BaseWorkerOutput method*), 204
to_recurrent_gym_space() (*maze.perception.builders.base.BaseModelBuilder class method*), 139
to_scalar_reward() (*maze.core.agent_integration.external_core_env.ExternalCoreEnvRewardAggregator class method*), 122
to_scalar_reward() (*maze.core.env.reward.RewardAggregatorInterface class method*), 48
to_scalar_reward() (*maze.core.wrappers.maze_gym_env_wrapper.GymRewardAggregator class method*), 71
TorchActorCritic (*class in maze.core.agent.torch_actor_critic*), 117
TorchDeltaStateCritic (*class in maze.core.agent.torch_state_critic*), 112
TorchModel (*class in maze.core.agent.torch_model*), 117
TorchPolicy (*class in maze.core.agent.torch_policy*), 105
TorchProbabilityDistribution (*class in maze.distributions.torch_dist*), 156
TorchSharedStateActionCritic (*class in maze.core.agent.torch_state_action_critic*), 115
TorchSharedStateCritic (*class in maze.core.agent.torch_state_critic*), 111
TorchStateActionCritic (*class in maze.core.agent.torch_state_action_critic*), 113
TorchStateCritic (*class in maze.core.agent.torch_state_critic*), 110
TorchStepStateActionCritic (*class in maze.core.agent.torch_state_action_critic*), 116
TorchStepStateCritic (*class in maze.core.agent.torch_state_critic*), 111
traceback() (*maze.core.rollout.parallel_rollout_runner.ExceptionReport property*), 103
train() (*maze.core.agent.torch_actor_critic.TorchActorCritic method*), 118
train() (*maze.core.agent.torch_model.TorchModel method*), 117
train() (*maze.core.agent.torch_policy.TorchPolicy method*), 107
train() (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic method*), 114
train() (*maze.core.agent.torch_state_critic.TorchStateCritic method*), 111
train() (*maze.train.trainers.common.actor_critic.actor_critic_trainer.M method*), 170
train() (*maze.train.trainers.es.es_trainer.ESTrainer method*), 189
train() (*maze.train.trainers.imitation.bc_trainer.BCTrainer method*), 199
train() (*maze.train.trainers.impala.impala_trainer.MultiStepIMPALA method*), 177
train_async() (*maze.train.trainers.impala.impala_trainer.MultiStepIM method*), 177
train_stats (*maze.train.trainers.imitation.bc_trainer.BCTrainer attribute*), 199
TrainConfig (*class in maze.train.trainers.common.training_runner*), 167
Trainer (*class in maze.train.trainers.common.trainer*), 166
TrainingRunner (*class in maze.train.trainers.common.training_runner*), 167
TrajectoryWriter (*class in maze.core.trajectory_recorder.trajectory_writer*), 97
TrajectoryWriterFile (*class in maze.core.trajectory_recorder.trajectory_writer_file*), 97
TrajectoryWriterRegistry (*class in maze.core.trajectory_recorder.trajectory_writer_registry*), 95
TransposePreProcessor (*class in maze.core.wrappers.observation_preprocessing.preprocessors.tra*), 63

U

UnSqueezePreProcessor (*class in maze.core.wrappers.observation_preprocessing.preprocessors.un*), 64
unstack_numpy_list_dict (*class in maze.train.utils.train_utils*), 202
unused (*class in maze.core.annotations*), 162
update() (*maze.train.trainers.common.model_selection.best_model_sele method*), 168
update() (*maze.train.trainers.common.model_selection.model_selection_ method*), 168
update() (*maze.train.trainers.es.optimizers.base_optimizer.Optimizer method*), 192
update_column_options() (*maze.core.rendering.notebook_event_logs_viewer.NotebookEven method*), 93
update_policy() (*maze.train.parallelization.base_worker.BaseWorker method*), 203

`update_policy()` (*maze.train.parallelization.distributed_actors.u* *method*), 206

`update_progress()` (*maze.core.rollout.sequential_rollout_runner.SequentialRolloutRunner* *method*), 101

`update_ratio()` (*maze.train.trainers.es.es_events.ESEvents* *method*), 190

`update_target_weights()` (*maze.core.agent.torch_state_action_critic.TorchStateActionCritic* *method*), 114

V

`validation_percentage` (*maze.train.trainers.imitation.bc_algorithm_config.BCAlgorithmConfig* *attribute*), 200

`value_loss_coef` (*maze.train.trainers.a2c.a2c_algorithm_config.A2CAlgorithmConfig* *attribute*), 174

`value_loss_coef` (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig* *attribute*), 180

`value_loss_coef` (*maze.train.trainers.ppo.ppo_algorithm_config.PPOAlgorithmConfig* *attribute*), 176

`VGGConvolutionBlock` (*class in maze.perception.blocks.feed_forward.vgg_conv*), 127

`VGGConvolutionDenseBlock` (*class in maze.perception.blocks.joint_blocks.vgg_conv_dense*), 136

`VGGConvolutionGAPBlock` (*class in maze.perception.blocks.joint_blocks.vgg_conv_gap*), 137

`vtrace_clip_pg_rho_threshold` (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig* *attribute*), 180

`vtrace_clip_rho_threshold` (*maze.train.trainers.impala.impala_algorithm_config.ImpalaAlgorithmConfig* *attribute*), 180

W

`wrap()` (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper* *class method*), 54

`wrap()` (*maze.core.wrappers.wrapper.Wrapper* *class method*), 51

`wrap_from_config()` (*maze.core.wrappers.wrapper_registry.WrapperRegistry* *method*), 52

`Wrapper` (*class in maze.core.wrappers.wrapper*), 50

`WrapperRegistry` (*class in maze.core.wrappers.wrapper_registry*), 52

`write()` (*maze.core.log_events.log_events_writer.LogEventsWriter* *method*), 78

`write()` (*maze.core.log_events.log_events_writer_tsv.LogEventsWriterTSV* *method*), 78

`write()` (*maze.core.log_stats.log_stats.LogStatsWriter* *method*), 84

`write()` (*maze.core.log_stats.log_stats_writer_console.LogStatsWriterCo* *method*), 82

`write()` (*maze.core.log_stats.log_stats_writer_tensorboard.LogStatsWrit* *method*), 82

`write()` (*maze.core.rollout.parallel_rollout_runner.EpisodeRecorder* *method*), 102

`write()` (*maze.core.trajectory_recorder.trajectory_writer.TrajectoryWrite* *method*), 97

`write()` (*maze.core.trajectory_recorder.trajectory_writer_file.TrajectoryW* *method*), 97

`write_epoch_stats()` (*maze.core.log_stats.log_stats_env.LogStatsEnv* *method*), 82

`write_epoch_stats()` (*maze.core.wrappers.log_stats_wrapper.LogStatsWrapper* *method*), 54

`write_epoch_stats()` (*maze.train.parallelization.distributed_env.dummy_distributed_en* *method*), 206