root@DESKTOP-OPUMIU2:/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav# python3 test.py --policy orca --phase test --gpu

2024-11-05 21:52:36, INFO: Using device: cuda:0

2024-11-05 21:52:36, INFO: human number: 5

2024-11-05 21:52:36, INFO: Not randomize human's radius and preferred speed

2024-11-05 21:52:36, INFO: Training simulation: circle\_crossing, test simulation: circle\_crossing

2024-11-05 21:52:36, INFO: Square width: 10.0, circle width: 4.0

2024-11-05 21:52:36, INFO: ORCA agent buffer: 0.000000

2024-11-05 21:52:36, INFO: Agent is invisible and has holonomic kinematic constraint

2024-11-05 21:52:42, INFO: TEST has success rate: 0.43, collision rate: 0.57, nav time: 10.86, total reward: -0.0220

2024-11-05 21:52:42, INFO: Frequency of being in danger: 0.30 and average min separate distance in danger: 0.08

2024-11-05 21:52:42, INFO: Collision cases: 0 1 2 5 8 10 11 13 14 16 18 19 20 22 23 24 26 30 34 37 41 42 43 45 51 53 57 59 66 69 70 72 73 80 82 83 85 89 90 91 93 95 97 98 99 100 101 105 106 107 109 111 113 114 115 117 119 127 128 129 131 132 134 136 138 139 140 145 146 147 148 152 153 155 157 159 163 164 165 166 167 170 171 172 174 178 180 184 188 189 190 192 193 194 196 198 199 201 204 205 206 207 208 213 214 215 217 219 220 221 225 227 229 230 231 232 234 235 237 239 240 242 243 247 251 255 256 257 258 259 260 261 262 263 264 265 266 268 271 272 273 274 275 276 279 281 283 284 285 288 289 291 292 295 296 297 298 299 300 302 304 307 309 310 311 312 315 317 319 322 324 327 329 331 333 334 336 337 340 342 343 344 346 347 349 351 352 353 355 357 358 361 362 363 364 365 366 367 368 370 372 374 376 378 379 381 382 383 384 385 386 389 391 393 394 395 400 401 402 403 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 424 425 427 431 432 433 434 437 438 440 442 443 444 445 447 448 449 450 452 453 456 457 458 460 462 463 464 466 467 469 471 473 474 479 481 483 485 486 488 490 492 493 495 496 497 498

2024-11-05 21:52:42, INFO: Timeout cases: 118 168 224

root@DESKTOP-OPUMIU2:/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav# python3 test.py --policy sarl --model\_dir data/output --phase test --gpu

2024-11-05 21:53:52, INFO: Using device: cuda:0

2024-11-05 21:53:53, INFO: Policy: SARL w/ global state

2024-11-05 21:53:58, INFO: human number: 5

2024-11-05 21:53:58, INFO: Not randomize human's radius and preferred speed

2024-11-05 21:53:58, INFO: Training simulation: circle\_crossing, test simulation: circle\_crossing

2024-11-05 21:53:58, INFO: Square width: 10.0, circle width: 4.0

2024-11-05 21:53:58, INFO: Agent is invisible and has holonomic kinematic constraint

2024-11-06 01:59:35, INFO: TEST has success rate: 0.92, collision rate: 0.00, nav time: 12.43, total reward: 0.2538

2024-11-06 01:59:35, INFO: Frequency of being in danger: 0.01 and average min separate distance in danger: 0.13

2024-11-06 01:59:35, INFO: Collision cases: 419

2024-11-06 01:59:35, INFO: Timeout cases: 0 1 3 11 22 32 68 76 91 115 116 137 145 147 151 166 182 184 185 196 203 206 215 274 276 285 298 342 367 369 374 392 405 427 428 432 449 457 460 463 483

import logging

import gym

import matplotlib.lines as mlines

import numpy as np

import rvo2

from matplotlib import patches

from numpy.linalg import norm

from crowd\_sim.envs.utils.human import Human

from crowd\_sim.envs.utils.info import \*

from crowd\_sim.envs.utils.utils import point\_to\_segment\_dist

class CrowdSim(gym.Env):

metadata = {'render.modes': ['human']}

def \_\_init\_\_(self):

"""

Movement simulation for multiple robots and humans.

Robots are connected by a flexible hose.

Humans are controlled by an unknown and fixed policy.

Robots are controlled by a known and learnable policy.

"""

self.time\_limit = None

self.time\_step = None

self.robot1 = None # First robot

self.robot2 = None # Second robot

self.humans = None

self.global\_time = None

self.human\_times = None

# reward function

self.success\_reward = None

self.collision\_penalty = None

self.hose\_penalty = None # Penalty for violating hose constraints

self.discomfort\_dist = None

self.discomfort\_penalty\_factor = None

# simulation configuration

self.config = None

self.case\_capacity = None

self.case\_size = None

self.case\_counter = None

self.randomize\_attributes = None

self.train\_val\_sim = None

self.test\_sim = None

self.square\_width = None

self.circle\_radius = None

self.human\_num = None

# for visualization

self.states = None

self.action\_values = None

self.attention\_weights = None

def configure(self, config):

self.config = config

self.time\_limit = config.getint('env', 'time\_limit')

self.time\_step = config.getfloat('env', 'time\_step')

self.randomize\_attributes = config.getboolean('env', 'randomize\_attributes')

self.success\_reward = config.getfloat('reward', 'success\_reward')

self.collision\_penalty = config.getfloat('reward', 'collision\_penalty')

self.hose\_penalty = config.getfloat('reward', 'hose\_penalty') # New penalty

self.discomfort\_dist = config.getfloat('reward', 'discomfort\_dist')

self.discomfort\_penalty\_factor = config.getfloat('reward', 'discomfort\_penalty\_factor')

if self.config.get('humans', 'policy') == 'orca':

self.case\_capacity = {'train': np.iinfo(np.uint32).max - 2000, 'val': 1000, 'test': 1000}

self.case\_size = {'train': np.iinfo(np.uint32).max - 2000, 'val': config.getint('env', 'val\_size'),

'test': config.getint('env', 'test\_size')}

self.train\_val\_sim = config.get('sim', 'train\_val\_sim')

self.test\_sim = config.get('sim', 'test\_sim')

self.square\_width = config.getfloat('sim', 'square\_width')

self.circle\_radius = config.getfloat('sim', 'circle\_radius')

self.human\_num = config.getint('sim', 'human\_num')

else:

raise NotImplementedError

self.case\_counter = {'train': 0, 'test': 0, 'val': 0}

logging.info('Human number: {}'.format(self.human\_num))

if self.randomize\_attributes:

logging.info("Randomize human's radius and preferred speed")

else:

logging.info("Not randomize human's radius and preferred speed")

logging.info('Training simulation: {}, test simulation: {}'.format(self.train\_val\_sim, self.test\_sim))

logging.info('Square width: {}, circle width: {}'.format(self.square\_width, self.circle\_radius))

def set\_robot1(self, robot):

self.robot1 = robot

def set\_robot2(self, robot):

self.robot2 = robot

def generate\_robot(self, robot):

"""

Randomly assign positions to a robot within certain bounds.

"""

# Randomly assign positions within certain bounds

px = np.random.uniform(-self.square\_width / 2, self.square\_width / 2)

py = np.random.uniform(-self.square\_width / 2, self.square\_width / 2)

gx = np.random.uniform(-self.square\_width / 2, self.square\_width / 2)

gy = np.random.uniform(-self.square\_width / 2, self.square\_width / 2)

robot.set(px, py, gx, gy, 0, 0, 0)

return robot

def generate\_random\_robot\_position(self):

"""

Generate initial and goal positions for robots.

Robots are connected by a flexible hose of length 0.6 meters.

"""

# Generate positions for robot1

self.robot1 = self.generate\_robot(self.robot1)

# Generate positions for robot2, ensuring it's within hose length

while True:

self.robot2 = self.generate\_robot(self.robot2)

distance = norm((self.robot1.px - self.robot2.px, self.robot1.py - self.robot2.py))

if distance <= 0.6:

break

def reset(self, phase='test', test\_case=None):

"""

Set px, py, gx, gy, vx, vy, theta for robots and humans

"""

if self.robot1 is None or self.robot2 is None:

raise AttributeError('Robots have to be set!')

assert phase in ['train', 'val', 'test']

if test\_case is not None:

self.case\_counter[phase] = test\_case

self.global\_time = 0

if phase == 'test':

self.human\_times = [0] \* self.human\_num

else:

self.human\_times = [0] \* self.human\_num

self.generate\_random\_robot\_position()

if self.case\_counter[phase] >= 0:

if phase in ['train', 'val']:

human\_num = self.human\_num

self.generate\_random\_human\_position(human\_num=human\_num, rule=self.train\_val\_sim)

else:

self.generate\_random\_human\_position(human\_num=self.human\_num, rule=self.test\_sim)

# case\_counter is always between 0 and case\_size[phase]

self.case\_counter[phase] = (self.case\_counter[phase] + 1) % self.case\_size[phase]

else:

raise NotImplementedError

# Set time steps for all agents

for agent in [self.robot1, self.robot2] + self.humans:

agent.time\_step = self.time\_step

agent.policy.time\_step = self.time\_step

self.states = list()

if hasattr(self.robot1.policy, 'action\_values'):

self.action\_values = list()

if hasattr(self.robot1.policy, 'get\_attention\_weights'):

self.attention\_weights = list()

# Get current observation

if self.robot1.sensor == 'coordinates':

ob1 = [human.get\_observable\_state() for human in self.humans]

ob2 = [human.get\_observable\_state() for human in self.humans]

ob = [ob1, ob2]

elif self.robot1.sensor == 'RGB':

raise NotImplementedError

return ob

def step(self, actions):

"""

Compute actions for all agents, detect collision, update environment, and return (ob, reward, done, info)

:param actions: List of actions for each robot

"""

human\_actions = []

for human in self.humans:

# Observation for humans is always coordinates

ob = [other\_human.get\_observable\_state() for other\_human in self.humans if other\_human != human]

ob += [robot.get\_observable\_state() for robot in [self.robot1, self.robot2] if robot.visible]

human\_actions.append(human.act(ob))

# Collision detection

collision = False

hose\_violation = False

dmin = float('inf')

robots = [self.robot1, self.robot2]

for i, robot in enumerate(robots):

action = actions[i]

# Check collision with humans

for j, human in enumerate(self.humans):

px = human.px - robot.px

py = human.py - robot.py

if robot.kinematics == 'holonomic':

vx = human.vx - action.vx

vy = human.vy - action.vy

else:

vx = human.vx - action.v \* np.cos(action.r + robot.theta)

vy = human.vy - action.v \* np.sin(action.r + robot.theta)

ex = px + vx \* self.time\_step

ey = py + vy \* self.time\_step

# Closest distance between boundaries of two agents

closest\_dist = point\_to\_segment\_dist(px, py, ex, ey, 0, 0) - human.radius - robot.radius

if closest\_dist < 0:

collision = True

break

elif closest\_dist < dmin:

dmin = closest\_dist

if collision:

break

# Hose constraint enforcement

distance = norm((self.robot1.px - self.robot2.px, self.robot1.py - self.robot2.py))

if distance > 0.6:

hose\_violation = True

# Check if robots reach their goals

reaching\_goals = [norm(np.array(robot.compute\_position(actions[i], self.time\_step)) - np.array(robot.get\_goal\_position())) < robot.radius for i, robot in enumerate(robots)]

done = False

if self.global\_time >= self.time\_limit - 1:

reward = 0

done = True

info = Timeout()

elif collision:

reward = self.collision\_penalty

done = True

info = Collision()

elif all(reaching\_goals):

reward = self.success\_reward

done = True

info = ReachGoal()

elif hose\_violation:

reward = -self.hose\_penalty # Negative reward for hose violation

done = False

info = ConstraintViolation()

elif dmin < self.discomfort\_dist:

# Only penalize agent for getting too close

reward = (dmin - self.discomfort\_dist) \* self.discomfort\_penalty\_factor \* self.time\_step

done = False

info = Danger(dmin)

else:

reward = 0

done = False

info = Nothing()

# Update environment

self.states.append([self.robot1.get\_full\_state(), self.robot2.get\_full\_state()] +

[[human.get\_full\_state() for human in self.humans]])

if hasattr(self.robot1.policy, 'action\_values'):

self.action\_values.append([self.robot1.policy.action\_values, self.robot2.policy.action\_values])

if hasattr(self.robot1.policy, 'get\_attention\_weights'):

self.attention\_weights.append([self.robot1.policy.get\_attention\_weights(), self.robot2.policy.get\_attention\_weights()])

# Update all agents

self.robot1.step(actions[0])

self.robot2.step(actions[1])

for i, human\_action in enumerate(human\_actions):

self.humans[i].step(human\_action)

self.global\_time += self.time\_step

for i, human in enumerate(self.humans):

# Only record the first time the human reaches the goal

if self.human\_times[i] == 0 and human.reached\_destination():

self.human\_times[i] = self.global\_time

# Compute the observation

if self.robot1.sensor == 'coordinates':

ob1 = [human.get\_observable\_state() for human in self.humans]

ob2 = [human.get\_observable\_state() for human in self.humans]

ob = [ob1, ob2]

elif self.robot1.sensor == 'RGB':

raise NotImplementedError

return ob, reward, done, info

def generate\_random\_human\_position(self, human\_num, rule):

"""

Generate human positions according to certain rule.

:param human\_num:

:param rule:

:return:

"""

if rule == 'square\_crossing':

self.humans = []

for i in range(human\_num):

self.humans.append(self.generate\_square\_crossing\_human())

elif rule == 'circle\_crossing':

self.humans = []

for i in range(human\_num):

self.humans.append(self.generate\_circle\_crossing\_human())

else:

raise ValueError("Rule doesn't exist")

def generate\_circle\_crossing\_human(self):

human = Human(self.config, 'humans')

if self.randomize\_attributes:

human.sample\_random\_attributes()

while True:

angle = np.random.random() \* np.pi \* 2

# add some noise to simulate all the possible cases robot could meet with human

px\_noise = (np.random.random() - 0.5) \* human.v\_pref

py\_noise = (np.random.random() - 0.5) \* human.v\_pref

px = self.circle\_radius \* np.cos(angle) + px\_noise

py = self.circle\_radius \* np.sin(angle) + py\_noise

collide = False

for agent in [self.robot1, self.robot2] + self.humans:

min\_dist = human.radius + agent.radius + self.discomfort\_dist

if norm((px - agent.px, py - agent.py)) < min\_dist or \

norm((px - agent.gx, py - agent.gy)) < min\_dist:

collide = True

break

if not collide:

break

human.set(px, py, -px, -py, 0, 0, 0)

return human

def generate\_square\_crossing\_human(self):

human = Human(self.config, 'humans')

if self.randomize\_attributes:

human.sample\_random\_attributes()

if np.random.random() > 0.5:

sign = -1

else:

sign = 1

while True:

px = np.random.random() \* self.square\_width \* 0.5 \* sign

py = (np.random.random() - 0.5) \* self.square\_width

collide = False

for agent in [self.robot1, self.robot2] + self.humans:

if norm((px - agent.px, py - agent.py)) < human.radius + agent.radius + self.discomfort\_dist:

collide = True

break

if not collide:

break

while True:

gx = np.random.random() \* self.square\_width \* 0.5 \* -sign

gy = (np.random.random() - 0.5) \* self.square\_width

collide = False

for agent in [self.robot1, self.robot2] + self.humans:

if norm((gx - agent.gx, gy - agent.gy)) < human.radius + agent.radius + self.discomfort\_dist:

collide = True

break

if not collide:

break

human.set(px, py, gx, gy, 0, 0, 0)

return human

# Add any additional methods needed to support the new functionality

willzomer123: @workspace /fix

Workspace: ### Plan

1. Identify the problem in the provided code.

2. Fix the problem while maintaining the original functionality and style.

### Problem

The issue is that the

self.case\_capacity

and

self.case\_size

dictionaries are not properly aligned, which can lead to readability issues.

### Solution

Align the dictionary values properly for better readability.

```python

self.case\_capacity = {

'train': np.iinfo(np.uint32).max - 2000,

'val': 1000,

'test': 1000

}

self.case\_size = {

'train': np.iinfo(np.uint32).max - 2000,

'val': config.getint('env', 'val\_size'),

'test': config.getint('env', 'test\_size')

}

self.train\_val\_sim = config.get('sim', 'train\_val\_sim')

self.test\_sim = config.get('sim', 'test\_sim')

self.square\_width = config.getfloat('sim', 'square\_width')

self.circle\_radius = config.getfloat('sim', 'circle\_radius')

self.human\_num = config.getint('sim', 'human\_num')

else:

raise NotImplementedError

self.case\_counter = {

'train': 0,

'test': 0,

'val': 0

}

logging.info('Human number: {}'.format(self.human\_num))

if self.randomize\_attributes:

logging.info("Randomize human's radius and preferred speed")

else:

logging.info("Not randomize human's radius and preferred speed")

logging.info('Training simulation: {}, test simulation: {}'.format(self.train\_val\_sim, self.test\_sim))

logging.info('Square width: {}, circle width: {}'.format(self.square\_width, self.circle\_radius))

def set\_robot1(self, robot):

self.robot1 = robot

def set\_robot2(self, robot):

self.robot2 = robot

```

Similar code found with 2 license types

willzomer123: crowd\_sim.py", line 255, in step

vx = human.vx - action.vx

AttributeError: 'int' object has no attribute 'vx'

GitHub Copilot: ### Plan

1. Identify why

action

is an integer instead of an object with

vx

and

vy

attributes.

2. Ensure that the

actions

list contains action objects with the necessary attributes.

3. Modify the code that generates or assigns

actions

to use the correct action objects.

### Problem

The error occurs because

action

is an integer, so accessing

action.vx

raises an `AttributeError`. This suggests that

actions[i]

is an integer and not an action object with

vx

and

vy

attributes.

### Solution

Ensure that the

actions

list contains action objects with

vx

and

vy

attributes for each robot. If

actions

is currently a list of integers, update it to create action objects. For example:

```python

# Example of creating action objects with vx and vy attributes

actions = [Action(vx=..., vy=...), Action(vx=..., vy=...)]

```

Verify that wherever

actions

is defined or updated, it uses the correct action class. This should resolve the `AttributeError` by ensuring that

action

has the required attributes.

willzomer123: @/fix

GitHub Copilot: ### Plan

1. Identify where

actions

is defined before being passed to the

step

function.

2. Check if

actions

contains integers instead of action objects.

3. Modify the code to ensure

actions

is a list of action objects with `vx` and `vy` attributes.

### Solution

The error occurs because

action

is an integer, so accessing

action.vx

raises an `AttributeError`. This suggests that

actions[i]

is an integer rather than an action object.

Ensure that when you call the

step

function, the

actions

list contains action objects. Here's how you might define and use

actions

:

```python

# Define action objects for each robot

action1 = ActionXY(vx=desired\_vx1, vy=desired\_vy1)

action2 = ActionXY(vx=desired\_vx2, vy=desired\_vy2)

# Create a list of actions

actions = [action1, action2]

# Call the step function with the actions

env.step(actions)

```

Replace `desired\_vx1`, `desired\_vy1`, `desired\_vx2`, and `desired\_vy2` with the appropriate velocity values for your robots.

### Explanation

- \*\*Define Proper Actions\*\*: Make sure `ActionXY` (or the appropriate action class) is used to create action objects with `vx` and `vy` attributes.

- \*\*Pass Correct Actions to

step

\*\*: When calling `env.step(actions)`,

actions

should be a list of these action objects.

- \*\*Avoid Integers in Actions\*\*: If

actions

contains integers or incorrect types, update the code that generates

actions

to produce the correct action objects.

### Example

Here's an example of how to define and use actions:

```python

# Assuming you have an ActionXY class

class ActionXY:

def \_\_init\_\_(self, vx, vy):

self.vx = vx

self.vy = vy

# Desired velocities for each robot

desired\_vx1, desired\_vy1 = 1.0, 0.0

desired\_vx2, desired\_vy2 = 0.5, 0.5

# Create action objects

action1 = ActionXY(vx=desired\_vx1, vy=desired\_vy1)

action2 = ActionXY(vx=desired\_vx2, vy=desired\_vy2)

# List of actions

actions = [action1, action2]

# Call the step function

env.step(actions)

```

### Summary

By ensuring

actions

contains the correct action objects with `vx` and `vy` attributes, the `AttributeError` should be resolved.

willzomer123: File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/utils/explorer.py", line 60, in run\_k\_episodes

action1 = self.robot1.act(ob[0])

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/utils/robot.py", line 13, in act

action = self.policy.predict(state)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/policy/multi\_human\_rl.py", line 38, in predict

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(action)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/crowd\_sim.py", line 117, in onestep\_lookahead

return self.step(action, update=False)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/crowd\_sim.py", line 254, in step

vx = human.vx - action.vx

AttributeError: 'int' object has no attribute 'vx'

GitHub Copilot: ### Plan

1. \*\*Identify the Source of the Error\*\*: The `AttributeError` occurs because `action` is an integer and does not have the `vx` attribute.

2. \*\*Check the `predict` Method in `multi\_human\_rl.py`\*\*: Ensure that the `predict` method returns an action object with `vx` and `vy` attributes.

3. \*\*Modify the `predict` Method to Return the Correct Action Object\*\*.

### Solution

In `multi\_human\_rl.py`, update the `predict` method to return an action object with `vx` and `vy` attributes instead of an integer.

```python

# multi\_human\_rl.py

class MultiHumanRL(Policy):

def predict(self, state):

# Your model prediction logic here

# For example, if your model outputs action velocities:

predicted\_vx, predicted\_vy = self.model.predict(state)

# Create an action object with the predicted velocities

action = ActionXY(vx=predicted\_vx, vy=predicted\_vy)

return action

```

Ensure that:

- \*\*Model Outputs\*\*: The model's output (`predicted\_vx`, `predicted\_vy`) provides the necessary velocity components.

- \*\*Action Object\*\*: You use the appropriate action class (`ActionXY`) that has `vx` and `vy` attributes.

### Explanation

- \*\*Incorrect Return Value\*\*: The `predict` method was returning an integer, leading to the `AttributeError` when accessing `action.vx`.

- \*\*Correct Return Type\*\*: By returning an action object with `vx` and `vy`, the

step

function can access these attributes without error.

### Summary

By modifying the `predict` method to return an action object with the required attributes, the `AttributeError` can be resolved.

willzomer123: Traceback (most recent call last):

File "train.py", line 183, in <module>

main()

File "train.py", line 165, in main

explorer.run\_k\_episodes(env.case\_size['val'], 'val', episode=episode)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/utils/explorer.py", line 60, in run\_k\_episodes

action1 = self.robot1.act(ob[0])

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/utils/robot.py", line 13, in act

action = self.policy.predict(state)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/policy/multi\_human\_rl.py", line 38, in predict

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(action)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/crowd\_sim.py", line 117, in onestep\_lookahead

return self.step(action, update=False)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/crowd\_sim.py", line 255, in step

vx = human.vx - action.vx

AttributeError: 'int' object has no attribute 'vx'

GitHub Copilot: ### Plan

The error occurs because the

step

function expects a list of actions for all robots, but

onestep\_lookahead

is passing only a single action. We need to modify

onestep\_lookahead

to accept a single action and generate default actions for the other robots.

### Solution

\*\*In

crowd\_sim.py

\*\*, modify the

onestep\_lookahead

method:

```python

def onestep\_lookahead(self, action, robot\_index=0):

# Generate default actions for other robots

default\_action = ActionXY(0, 0) # Assuming stationary for simplicity

actions = []

for i in range(2): # Assuming there are two robots

if i == robot\_index:

actions.append(action)

else:

actions.append(default\_action)

return self.step(actions, update=False)

```

\*\*In

multi\_human\_rl.py

\*\*, update the

predict

method to pass the `robot\_index`:

```python

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(action, robot\_index=self.robot\_index)

```

Ensure that

self.robot\_index

is set appropriately when initializing the policy for each robot.

\*\*Set `robot\_index` in

robot.py

\*\* when assigning the policy:

```python

# In robot.py

class Robot(Agent):

def \_\_init\_\_(self, config, section, robot\_index):

super().\_\_init\_\_(config, section)

self.robot\_index = robot\_index

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

state = JointState(self.get\_full\_state(), ob)

action = self.policy.predict(state)

return action

```

\*\*Adjust robot initialization in

train.py

\*\*:

```python

# In train.py

robot1 = Robot(env\_config, 'robot', robot\_index=0)

robot2 = Robot(env\_config, 'robot', robot\_index=1)

```

### Summary

By modifying

onestep\_lookahead

to accept a single action and providing default actions for other robots, and by ensuring the correct `robot\_index` is used, we resolve the

AttributeError

.

willzomer123: Traceback (most recent call last):

File "train.py", line 183, in <module>

main()

File "train.py", line 165, in main

explorer.run\_k\_episodes(env.case\_size['val'], 'val', episode=episode)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/utils/explorer.py", line 60, in run\_k\_episodes

action1 = self.robot1.act(ob[0])

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/utils/robot.py", line 13, in act

action = self.policy.predict(state)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/policy/multi\_human\_rl.py", line 38, in predict

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(action)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/crowd\_sim.py", line 117, in onestep\_lookahead

return self.step(action, update=False)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/crowd\_sim.py", line 255, in step

vx = human.vx - action.vx

AttributeError: 'int' object has no attribute 'vx'

GitHub Copilot: ### Analysis

The error you're encountering indicates that an

int

object is being used where an action object with

vx

and

vy

attributes is expected. Specifically, in

crowd\_sim.py

, within the

step

function, the code attempts to access

action.vx

, but

action

is an integer.

Let's trace the error step by step:

1. \*\*In

train.py

\*\*:

- Line 165:

explorer.run\_k\_episodes(env.case\_size['val'], 'val', episode=episode)

2. \*\*In

explorer.py

\*\*:

- Line 60:

action1 = self.robot1.act(ob[0])

3. \*\*In

robot.py

\*\*:

- Line 13:

action = self.policy.predict(state)

4. \*\*In

multi\_human\_rl.py

\*\*:

- Line 38:

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(action)

5. \*\*In

crowd\_sim.py

\*\*:

- Line 117:

return self.step(action, update=False)

6. \*\*In

crowd\_sim.py

\*\*:

- Line 255:

vx = human.vx - action.vx

At \*\*Step 5\*\*,

self.step(action, update=False)

is called with

action

being an integer instead of an action object. This misalignment causes the AttributeError when trying to access

action.vx

.

### Root Cause

The primary issue lies in the

predict

method of

multi\_human\_rl.py

. Specifically, during the exploration (when

probability < self.epsilon

),

max\_action

is selected randomly from

self.action\_space

. If

self.action\_space

contains integers instead of action objects,

max\_action

becomes an integer.

### Solution

We need to ensure that

self.action\_space

contains action objects with

vx

and

vy

attributes. Here's how to fix it:

#### 1. Validate

build\_action\_space

Method

Ensure that the

build\_action\_space

method in your policy correctly populates

self.action\_space

with action objects, not integers.

```python

# In multi\_human\_rl.py or the appropriate policy file

def build\_action\_space(self, v\_pref):

self.action\_space = []

# Define possible speeds and orientations

speeds = [...] # e.g., np.linspace(0, v\_pref, num\_speed\_steps)

angles = [...] # e.g., np.linspace(-np.pi, np.pi, num\_angle\_steps)

for speed in speeds:

for angle in angles:

vx = speed \* np.cos(angle)

vy = speed \* np.sin(angle)

action = ActionXY(vx, vy)

self.action\_space.append(action)

```

#### 2. Check Exploration Phase in

predict

Method

In

multi\_human\_rl.py

, within the

predict

method, ensure that during the exploration phase, the randomly selected

max\_action

is an action object.

```python

def predict(self, state):

# ...

probability = np.random.random()

if self.phase == 'train' and probability < self.epsilon:

# Randomly select an action from the action space

max\_action = random.choice(self.action\_space)

else:

# Existing code for selecting the best action

# ...

```

#### 3. Ensure

action\_space

Contains Action Objects

Double-check that

self.action\_space

is a list of

ActionXY

or

ActionRot

objects, depending on your kinematics.

#### 4. Modify

onestep\_lookahead

in

crowd\_sim.py

Update the

onestep\_lookahead

method to handle a single action by constructing a list of actions for all robots.

```python

# In crowd\_sim.py

def onestep\_lookahead(self, action, robot\_index=0):

# Create a list of actions where actions for other robots are default or previous actions

actions = []

for i in range(2): # Assuming there are two robots

if i == robot\_index:

actions.append(action)

else:

# Use a default action or the previous action for the other robot

default\_action = ActionXY(0, 0)

actions.append(default\_action)

return self.step(actions, update=False)

```

#### 5. Pass

robot\_index

from

predict

Method

Modify the

predict

method to pass the

robot\_index

to

onestep\_lookahead

.

```python

# In multi\_human\_rl.py

def predict(self, state):

# ...

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(max\_action, robot\_index=self.robot\_index)

```

Ensure that

self.robot\_index

is properly set for each robot's policy.

#### 6. Set

robot\_index

for Each Robot

When initializing each robot and its policy, assign a unique

robot\_index

.

```python

# In robot.py

class Robot(Agent):

def \_\_init\_\_(self, config, section, robot\_index):

super().\_\_init\_\_(config, section)

self.robot\_index = robot\_index

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

state = JointState(self.get\_full\_state(), ob)

action = self.policy.predict(state)

return action

```

```python

# In train.py or wherever you initialize robots

robot1 = Robot(env\_config, 'robot', robot\_index=0)

robot2 = Robot(env\_config, 'robot', robot\_index=1)

```

#### 7. Verify Action Objects Throughout the Code

Ensure that all functions and methods that interact with actions are expecting and returning action objects with

vx

and

vy

.

### Summary

By ensuring that

self.action\_space

contains action objects and modifying the

onestep\_lookahead

method to accept and construct a list of actions for all robots, the

AttributeError

should be resolved.

### Additional Tips

- \*\*Logging\*\*: Add logging statements to verify the types of variables at different points in your code.

```python

# Example

logging.debug(f"Action type: {type(action)}, action: {action}")

```

- \*\*Type Checking\*\*: Use assertions to check types.

```python

assert isinstance(action, ActionXY), f"Expected action to be ActionXY, got {type(action)}"

```

- \*\*Testing\*\*: Write unit tests for your

build\_action\_space

and

predict

methods to ensure they behave as expected.

### Final Thoughts

The key is to ensure consistency in the types of objects being passed around, especially when dealing with actions in a multi-agent environment. By carefully tracing and validating each step, you can identify and fix issues related to object types and method expectations.

willzomer123: Traceback (most recent call last):

File "train.py", line 183, in <module>

main()

File "train.py", line 165, in main

explorer.run\_k\_episodes(env.case\_size['val'], 'val', episode=episode)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/utils/explorer.py", line 60, in run\_k\_episodes

action1 = self.robot1.act(ob[0])

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/utils/robot.py", line 13, in act

action = self.policy.predict(state)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/policy/multi\_human\_rl.py", line 38, in predict

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(action, robot\_index=self.robot\_index)

AttributeError: 'SARL' object has no attribute 'robot\_index'

GitHub Copilot: To resolve the

AttributeError

you're encountering, you need to modify the

predict

method in your

multi\_human\_rl.py

to accept

robot\_index

as a parameter. Here's how you can do it:

### Solution

\*\*1. Modify the

predict

Method in

multi\_human\_rl.py

:\*\*

Change the

predict

method to accept

robot\_index

as an argument and use it in the method.

```python

# In multi\_human\_rl.py

def predict(self, state, robot\_index=0):

# ... existing code ...

if self.query\_env:

next\_human\_states, reward, done, info = self.env.onestep\_lookahead(action, robot\_index=robot\_index)

else:

# ... existing code ...

# ... existing code ...

```

\*\*2. Update the

act

Method in

robot.py

:\*\*

When calling the

predict

method from the robot, pass the

robot\_index

attribute.

```python

# In robot.py

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

state = JointState(self.get\_full\_state(), ob)

action = self.policy.predict(state, robot\_index=self.robot\_index)

return action

```

\*\*3. Ensure Robots Have Unique

robot\_index

:\*\*

Make sure each robot has a unique

robot\_index

when they are initialized.

```python

# In train.py or wherever robots are initialized

robot1 = Robot(env\_config, 'robot', robot\_index=0)

robot2 = Robot(env\_config, 'robot', robot\_index=1)

```

\*\*4. Use the Same Policy Instance if Desired:\*\*

Since you're using the same policy instance for both robots, passing

robot\_index

as a parameter avoids conflicts.

\*\*Summary:\*\*

By modifying the

predict

method to accept

robot\_index

as a parameter and passing the robot's index when calling it, you eliminate the need for the

robot\_index

attribute in the

SARL

class. This should resolve the

AttributeError

and allow each robot to correctly use the

predict

method without interfering with each other.

willzomer123: Traceback (most recent call last):

File "train.py", line 183, in <module>

main()

File "train.py", line 131, in main

explorer.run\_k\_episodes(il\_episodes, 'train', update\_memory=True, imitation\_learning=True)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/utils/explorer.py", line 60, in run\_k\_episodes

action1 = self.robot1.act(ob[0])

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/utils/robot.py", line 13, in act

action = self.policy.predict(state, robot\_index=self.robot\_index)

TypeError: predict() got an unexpected keyword argument 'robot\_index'

GitHub Copilot: The error you're encountering occurs because during imitation learning, the

predict

method of the policy you're using does not accept the

robot\_index

parameter. Specifically, when using policies like \*\*ORCA\*\* during imitation learning, the

predict

method signature differs from that in your reinforcement learning policies.

### Analysis

Here's the error message:

```plaintext

TypeError: predict() got an unexpected keyword argument 'robot\_index'

```

This error indicates that the

predict

method is being called with an unexpected keyword argument,

robot\_index

. This happens in the

act

method of the

Robot

class:

```python

# robot.py

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

state = JointState(self.get\_full\_state(), ob)

action = self.policy.predict(state, robot\_index=self.robot\_index) # Error occurs here

return action

```

During imitation learning, you're using a different policy (e.g., \*\*ORCA\*\*) that does not accept

robot\_index

as a parameter in its

predict

method.

### Solution

To fix this issue, modify the

act

method in

robot.py

to check whether the

predict

method of the current policy accepts the

robot\_index

parameter. You can use the `inspect` module to examine the parameters of the

predict

method at runtime.

Here's how you can modify the

act

method:

```python

# robot.py

import inspect

class Robot(Agent):

def \_\_init\_\_(self, config, section, robot\_index):

super().\_\_init\_\_(config, section)

self.robot\_index = robot\_index

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

state = JointState(self.get\_full\_state(), ob)

# Check if 'robot\_index' is a parameter of the 'predict' method

if 'robot\_index' in inspect.signature(self.policy.predict).parameters:

action = self.policy.predict(state, robot\_index=self.robot\_index)

else:

action = self.policy.predict(state)

return action

```

### Explanation

- \*\*Import `inspect` Module\*\*: Allows you to inspect the signature of methods and check their parameters.

- \*\*Conditional Parameter Passing\*\*:

```python

if 'robot\_index' in inspect.signature(self.policy.predict).parameters:

action = self.policy.predict(state, robot\_index=self.robot\_index)

else:

action = self.policy.predict(state)

```

- \*\*If

robot\_index

Is Accepted\*\*: Pass

robot\_index

to the

predict

method (used for policies like \*\*SARL\*\*).

- \*\*If Not\*\*: Call

predict

without

robot\_index

(used for policies like \*\*ORCA\*\* during imitation learning).

### Alternative Solution

If you prefer not to use the `inspect` module, you can check the type or name of the policy:

```python

# robot.py

class Robot(Agent):

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

state = JointState(self.get\_full\_state(), ob)

# Check if the policy is an instance of MultiHumanRL

if isinstance(self.policy, MultiHumanRL):

action = self.policy.predict(state, robot\_index=self.robot\_index)

else:

action = self.policy.predict(state)

return action

```

However, using the `inspect` module provides a more flexible solution that doesn't require hardcoding policy types.

### Summary

By modifying the

act

method to conditionally include the

robot\_index

parameter based on whether the

predict

method accepts it, you ensure compatibility with different policies used during imitation learning and reinforcement learning phases.

### Additional Notes

- \*\*Ensure Consistency in Policy Assignments\*\*: When using different policies (e.g., ORCA for imitation learning and SARL for reinforcement learning), make sure to assign the appropriate policy to each robot at the correct phase.

- \*\*Check Policy Initialization\*\*: In your

train.py

, ensure that policies are properly initialized and assigned:

```python

# In train.py

# Initialize imitation learning policy

il\_policy\_name = train\_config.get('imitation\_learning', 'il\_policy')

il\_policy = policy\_factory[il\_policy\_name]()

il\_policy.configure(policy\_config)

il\_policy.set\_device(device)

# Assign imitation learning policy to robots during imitation learning

if imitation\_learning:

robot1.set\_policy(il\_policy)

robot2.set\_policy(il\_policy)

else:

robot1.set\_policy(policy)

robot2.set\_policy(policy)

```

- \*\*Switching Policies Between Phases\*\*: You may need to switch policies between imitation learning and reinforcement learning phases. Ensure that this switch is handled correctly in your training script.

### Testing

After making the changes, test your code to ensure that:

- The error

TypeError: predict() got an unexpected keyword argument 'robot\_index'

is resolved.

- Both imitation learning and reinforcement learning phases run without issues.

- The robots behave as expected during both phases.

### Conclusion

By adapting the

act

method to handle different policy interfaces, you allow your robots to work seamlessly with various policies, enhancing the flexibility and robustness of your code.

willzomer123: Now you have a preliminary understanding of the situation of this project. The goal of this project is to enable the robot to reach the target point as quickly as possible while avoiding multiple dynamic pedestrians with unknown future trajectories. The current code is for one robot to avoid multiple pedestrians, and I now hope to expand it to the situation where multiple robots avoid multiple pedestrians. The robot is set to be circular with a radius of 0.3m, and two robots are connected by a flexible hose that can be bent and is 0.6m long. In the process of reinforcement learning training, it is necessary to ensure that the straight-line distance between two robots is less than or equal to the length of the hose (0.6m), that is, the two robots are always connected by a hose, and neither hose nor the two robots can touch pedestrians. Please modify all the necessary code to meet the above requirements.

GitHub Copilot: ### Plan

1. \*\*Update

crowd\_sim.py

\*\*:

- Modify the environment to handle two robots connected by a flexible hose.

- Implement the hose length constraint in the

step

function.

- Update collision detection to include the hose and ensure it doesn't collide with pedestrians.

2. \*\*Update

info.py

\*\*:

- Add a new `HoseViolation` class to represent the hose length constraint violation.

3. \*\*Update

robot.py

\*\*:

- Modify the

Robot

class to handle multiple robots and their interactions.

- Add methods to access the other robot's state.

4. \*\*Update

explorer.py

\*\*:

- Modify the

run\_k\_episodes

method to handle multiple robots and the hose constraint.

- Handle the `HoseViolation` in the training loop.

5. \*\*Update

multi\_human\_rl.py

and `sarl.py`\*\*:

- Adjust the policy to consider the other robot's state.

- Modify the input to the neural network to include the other robot's state.

6. \*\*Update

state.py

\*\*:

- Modify the

JointState

class to include the other robot's state.

7. \*\*Update training scripts (`train.py`)\*\*:

- Ensure both robots are initialized and trained.

- Enforce the hose constraint during training.

### Code Modifications

#### 1. Update

crowd\_sim.py

```python

# crowd\_sim.py

from crowd\_sim.envs.utils.info import \*

from crowd\_sim.envs.utils.utils import point\_to\_segment\_dist

class CrowdSim(gym.Env):

# ... existing code ...

def step(self, actions, update=True):

"""

Compute actions for all agents, detect collision, update environment, and return (ob, reward, done, info)

:param actions: List of actions for each robot [action1, action2]

"""

human\_actions = []

for human in self.humans:

# Observation for humans is always coordinates

ob = [other\_human.get\_observable\_state() for other\_human in self.humans if other\_human != human]

ob += [robot.get\_observable\_state() for robot in [self.robot1, self.robot2] if robot.visible]

human\_actions.append(human.act(ob))

# Collision detection

collision = False

hose\_violation = False

dmin = float('inf')

robots = [self.robot1, self.robot2]

for i, robot in enumerate(robots):

action = actions[i]

# Check collision with humans

for human in self.humans:

px = human.px - robot.px

py = human.py - robot.py

dist = np.hypot(px, py) - human.radius - robot.radius

if dist < 0:

collision = True

break

elif dist < self.discomfort\_dist:

# Discomfort penalty can be added here

pass

if dist < dmin:

dmin = dist

if collision:

break

# Hose length constraint

robot\_distance = np.hypot(self.robot1.px - self.robot2.px, self.robot1.py - self.robot2.py)

if robot\_distance > 0.6:

hose\_violation = True

# Compute rewards

if collision:

reward = self.collision\_penalty

done = True

info = Collision()

elif hose\_violation:

reward = -0.5 # Penalty for violating hose constraint

done = False

info = HoseViolation()

else:

# Reward for making progress towards the goal can be added here

reward = 0

done = False

info = Nothing()

# Update agents if not looking ahead

if update:

# Update robot positions

for i, robot in enumerate(robots):

robot.step(actions[i])

# Update human positions

for i, human in enumerate(self.humans):

human.step(human\_actions[i])

self.global\_time += self.time\_step

# Get new observations

ob = self.\_get\_observation()

return ob, reward, done, info

def \_get\_observation(self):

if self.robot1.sensor == 'coordinates':

ob1 = [human.get\_observable\_state() for human in self.humans]

ob2 = [human.get\_observable\_state() for human in self.humans]

ob = [ob1, ob2]

elif self.robot1.sensor == 'RGB':

raise NotImplementedError

return ob

def set\_robot1(self, robot):

self.robot1 = robot

self.robot1.set\_env(self)

def set\_robot2(self, robot):

self.robot2 = robot

self.robot2.set\_env(self)

```

#### 2. Update

info.py

```python

# info.py

class HoseViolation(object):

def \_\_init\_\_(self):

pass

def \_\_str\_\_(self):

return 'Hose length constraint violated'

```

#### 3. Update

robot.py

```python

# robot.py

class Robot(Agent):

def \_\_init\_\_(self, config, section, robot\_index):

super().\_\_init\_\_(config, section)

self.robot\_index = robot\_index

self.env = None

def set\_env(self, env):

self.env = env

def get\_other\_robot\_state(self):

if self.robot\_index == 0:

return self.env.robot2.get\_full\_state()

else:

return self.env.robot1.get\_full\_state()

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

other\_robot\_state = self.get\_other\_robot\_state()

state = JointState(self.get\_full\_state(), other\_robot\_state, ob)

action = self.policy.predict(state)

return action

```

#### 4. Update

explorer.py

```python

# explorer.py

class Explorer(object):

def \_\_init\_\_(self, env, robot1, robot2, device, memory=None, gamma=None, target\_policy=None):

self.env = env

self.robot1 = robot1

self.robot2 = robot2

self.device = device

self.memory = memory

self.gamma = gamma

self.target\_policy = target\_policy

def run\_k\_episodes(self, k, phase, update\_memory=False, imitation\_learning=False, episode=None,

print\_failure=False):

self.robot1.policy.set\_phase(phase)

self.robot2.policy.set\_phase(phase)

hose\_violations = 0

# ... existing code ...

for i in range(k):

ob = self.env.reset(phase)

done = False

# ... existing code ...

while not done:

action1 = self.robot1.act(ob[0])

action2 = self.robot2.act(ob[1])

actions = [action1, action2]

ob, reward, done, info = self.env.step(actions)

# ... existing code ...

if isinstance(info, HoseViolation):

hose\_violations += 1

# You can handle additional logic here

# ... existing code ...

# After episodes, you can log hose violation statistics

logging.info('Hose violations: {}'.format(hose\_violations))

```

#### 5. Update

multi\_human\_rl.py

```python

# multi\_human\_rl.py

class MultiHumanRL(CADRL):

def predict(self, state):

"""

Modify the input state to include other robot's state

"""

if self.phase is None or self.device is None:

raise AttributeError('Phase, device attributes have to be set!')

# ... existing code ...

# Combine self state and other robot state

combined\_self\_state = state.self\_state + state.other\_robot\_state

# Transform state

state\_tensor = torch.cat([torch.Tensor([combined\_self\_state + human\_state]).to(self.device)

for human\_state in state.human\_states], dim=0)

# ... existing code ...

return max\_action

```

#### 6. Update

state.py

```python

# state.py

class JointState(object):

def \_\_init\_\_(self, self\_state, other\_robot\_state, human\_states):

self.self\_state = self\_state

self.other\_robot\_state = other\_robot\_state

self.human\_states = human\_states

```

#### 7. Update `train.py`

```python

# train.py

# Initialize robots

robot1 = Robot(env\_config, 'robot', robot\_index=0)

robot2 = Robot(env\_config, 'robot', robot\_index=1)

# Set policies

robot1.set\_policy(policy)

robot2.set\_policy(policy)

# Set robots in the environment

env.set\_robot1(robot1)

env.set\_robot2(robot2)

# Initialize explorer with both robots

explorer = Explorer(env, robot1, robot2, device, memory=memory, gamma=0.9)

```

#### 8. Update `sarl.py`

Modify the

ValueNetwork

to accept the combined self state.

```python

# sarl.py

class ValueNetwork(nn.Module):

def forward(self, state):

"""

Modify to handle combined self state (self + other robot)

"""

size = state.shape

combined\_self\_state = state[:, 0, :self.self\_state\_dim]

# ... existing code ...

```

---

Please make sure to test each component thoroughly to ensure that the hose constraint is correctly enforced and that the robots behave as expected in the simulation.

willzomer123: Traceback (most recent call last):

File "train.py", line 183, in <module>

main()

File "train.py", line 165, in main

explorer.run\_k\_episodes(env.case\_size['val'], 'val', episode=episode)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/utils/explorer.py", line 60, in run\_k\_episodes

action1 = self.robot1.act(ob[0])

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/utils/robot.py", line 17, in act

action = self.policy.predict(state, robot\_index=self.robot\_index)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/policy/multi\_human\_rl.py", line 43, in predict

batch\_next\_states = torch.cat([torch.Tensor([next\_self\_state + next\_human\_state]).to(self.device)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_nav/policy/multi\_human\_rl.py", line 43, in <listcomp>

batch\_next\_states = torch.cat([torch.Tensor([next\_self\_state + next\_human\_state]).to(self.device)

File "/home/zzy/catkin\_ws/src/CrowdNav/crowd\_sim/envs/utils/state.py", line 18, in \_\_add\_\_

return other + (self.px, self.py, self.vx, self.vy, self.radius, self.gx, self.gy, self.v\_pref, self.theta)

TypeError: can only concatenate list (not "tuple") to list

GitHub Copilot: ### Analysis

The error you're encountering is:

```

TypeError: can only concatenate list (not "tuple") to list

```

This error occurs when you attempt to concatenate a tuple to a list using the `+` operator, which is not allowed in Python. The traceback indicates that the issue arises in the

\_\_add\_\_

method of the

FullState

class in

state.py

:

```python

# state.py, line 18

def \_\_add\_\_(self, other):

return other + (self.px, self.py, self.vx, self.vy, self.radius, self.gx, self.gy, self.v\_pref, self.theta)

```

Here,

other

is a list, and `(self.px, self.py, ...)` is a tuple. Attempting to concatenate them directly results in the `TypeError`.

### Solution

To fix this error, you need to ensure that both operands of the `+` operator are of the same type (both lists or both tuples). In this case, you can convert the tuple to a list before concatenation.

\*\*Modify the

\_\_add\_\_

methods in

FullState

and

ObservableState

to return lists instead of tuples.\*\*

#### Update

FullState.\_\_add\_\_

:

```python

# state.py

class FullState(object):

# ... existing code ...

def \_\_add\_\_(self, other):

return other + [self.px, self.py, self.vx, self.vy, self.radius,

self.gx, self.gy, self.v\_pref, self.theta]

# ... existing code ...

```

#### Update

ObservableState.\_\_add\_\_

:

```python

# state.py

class ObservableState(object):

# ... existing code ...

def \_\_add\_\_(self, other):

return other + [self.px, self.py, self.vx, self.vy, self.radius]

# ... existing code ...

```

### Explanation

By converting the tuple `(self.px, self.py, ...)` to a list `[self.px, self.py, ...]`, you ensure that both

other

and the added values are lists. This allows for successful concatenation using the `+` operator.

### Test the Fix

After making these changes, rerun your code to see if the error is resolved. The `TypeError` should no longer occur, and your program should proceed to the next steps.

### Additional Verification

Ensure that any other parts of your code that rely on the

\_\_add\_\_

methods are compatible with this change. Since you're now consistently using lists, it should integrate smoothly with your existing list operations.

### Summary

- \*\*Issue\*\*: Cannot concatenate a list and a tuple using `+`.

- \*\*Cause\*\*: The

\_\_add\_\_

methods were returning tuples when

other

was a list.

- \*\*Solution\*\*: Modify the

\_\_add\_\_

methods to return lists by replacing tuples with lists.

### Final Note

This adjustment ensures type consistency in your concatenations and should fix the `TypeError` you're encountering. If you encounter any further issues or have additional questions, feel free to ask.