# Abmarl

***Release 0.1.0***

**Edward Rusu**

**Jun 08, 2021**

# CONTENTS

Abmarl is a package for developing agent-based simulations and training them with multiagent reinforcement learning. We provide an intuitive command line interface for training, visualizing, and analyzing agent behavior. We define an *Agent Based Simulation Interface* and *Simulation Managers*, which control which agents interact with the simulation at each step. We support *integration* with several popular simulation interfaces, including *gym.Env* and *MultiAgentEnv*.

Abmarl is a layer in the Reinforcement Learning stack that sits on top of RLlib. We leverage RLlib's framework for training agents and extend it to more easily support custom simulations, algorithms, and policies. We enable researchers to rapidly prototype RL experiments and simulation design and lower the barrier for pre-existing projects to prototype RL as a potential solution.

CHAPTER

ONE

# DESIGN

A reinforcement learning experiment in Abmarl contains two interacting components: a Simulation and a Trainer.

The Simulation contains agent(s) who can observe the state (or a substate) of the Simulation and whose actions affect the state of the simulation. The simulation is discrete in time, and at each time step agents can provide actions. The simulation also produces rewards for each agent that the Trainer can use to train optimal behaviors. The Agent-Simulation interaction produces state-action-reward tuples (SARs), which can be collected in *rollout fragments* and used to optimize agent behaviors.

The Trainer contains policies that map agents' observations to actions. Policies are one-to-many with agents, meaning that there can be multiple agents using the same policy. Policies may be heuristic (i.e. coded by the researcher) or trainable by the RL algorithm.

In Abmarl, the Simulation and Trainer are specified in a single Python configuration file. Once these components are set up, they are passed as parameters to RLlib's tune command, which will launch the RLlib application and begin the training process. The training process will save checkpoints to an output directory, from which the user can visualize and analyze results. The following diagram demonstrates this workflow.

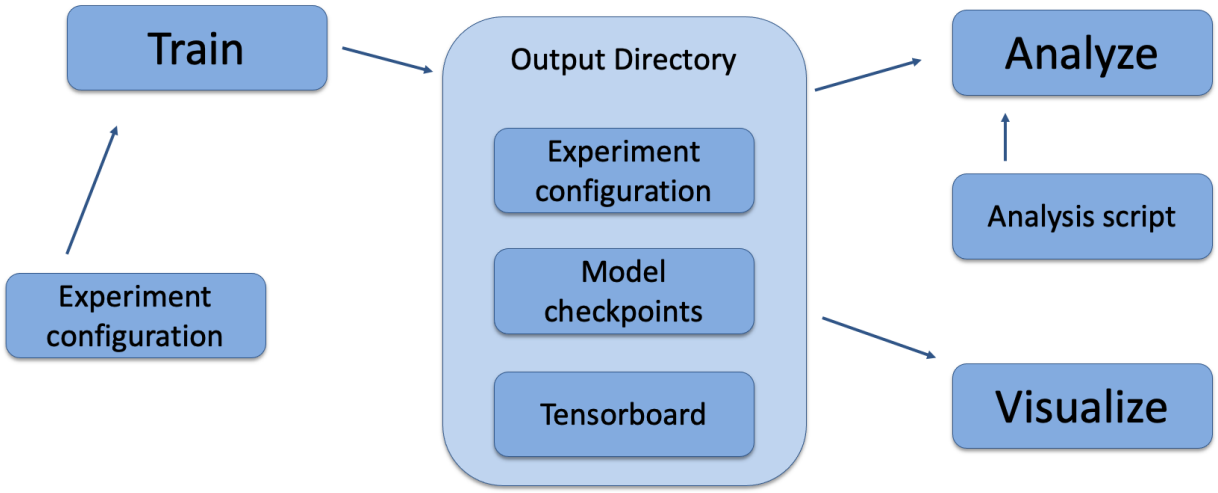



Fig. 1: Abmarl's usage workflow. An experiment configuration is used to train agents' behaviors. The policies and simulation are saved to an output directory. Behaviors can then be analyzed or visualized from the output directory.

## 1.1 Creating Agents and Simulations

Abmarl provides three interfaces for setting up an agent-based simulations.

### 1.1.1 Agent

First, we have *Agents*. An agent is an object with an observation and action space. Many practitioners may be accustomed to gym.Env's interface, which defines the observation and action space for the *simulation*. However, in heterogeneous multiagent settings, each *agent* can have different spaces; thus we assign these spaces to the agents and not the simulation.

An agent can be created like so:

```
from gym.spaces import Discrete, Box
from abmarl.sim import Agent
agent = Agent(
    id='agent0',
    observation_space=Box(-1, 1, (2,)),
    action_space=Discrete(3)
)
```

At this level, the Agent is basically a dataclass. We have left it open for our users to extend its features as they see fit.

### 1.1.2 Agent Based Simulation

Next, we define an *Agent Based Simulation*, or ABS for short, with the ususal `reset` and `step` functions that we are used to seeing in RL simulations. These functions, however, do not return anything; the state information must be obtained from the getters: `get_obs`, `get_reward`, `get_done`, `get_all_done`, and `get_info`. The getters take an agent's id as input and return the respective information from the simulation's state. The ABS also contains a dictionary of agents that "live" in the simulation.

An Agent Based Simulation can be created and used like so:

```
from abmarl.sim import Agent, AgentBasedSimulation
class MySim(AgentBasedSimulation):
    def __init__(self, agents=None, **kwargs):
        self.agents = agents
     ... # Implement the ABS interface

# Create a dictionary of agents
agents = {f'agent{i}': Agent(id=f'agent{i}', ...) for i in range(10)}
# Create the ABS with the agents
sim = MySim(agents=agents)
sim.reset()
# Get the observations
obs = {agent.id: sim.get_obs(agent.id) for agent in agents.values()}
# Take some random actions
sim.step({agent.id: agent.action_space.sample() for agent in agents.values()})
# See the reward for agent3
print(sim.get_reward('agent3'))
```

**Warning:** Implementations of AgentBasedSimulation should call `finalize` at the end of its `__init__`. Finalize ensures that all agents are configured and ready to be used for training.

**Note:** Instead of treating agents as dataclasses, we could have included the relevant information in the Agent Based Simulation with various dictionaries. For example, we could have `action_spaces` and `observation_spaces` that maps agents' ids to their action spaces and observation spaces, respectively. In Abmarl, we favor the dataclass approach and use it throughout the package and documentation.

### 1.1.3 Simulation Managers

The Agent Based Simulation interface does not specify an ordering for agents' interactions with the simulation. This is left open to give our users maximal flexibility. However, in order to interace with RLlib's learning library, we provide a *Simulation Manager* which specifies the output from `reset` and `step` as RLlib expects it. Specifically,

1. Agents that appear in the output dictionary will provide actions at the next step.
2. Agents that are done on this step will not provide actions on the next step.

Simulation managers are open-ended requiring only `reset` and `step` with output described above. For convenience, we have provided two managers: *Turn Based*, which implements turn-based games; and *All Step*, which has every non-done agent provide actions at each step.

Simluation Managers "wrap" simulations, and they can be used like so:

```
from abmarl.managers import AllStepManager
from abmarl.sim import AgentBasedSimulation, Agent
class MySim(AgentBasedSimulation):
    ... # Define some simulation

# Instatiate the simulation
sim = MySim(agents=...)
# Wrap the simulation with the simulation manager
sim = AllStepManager(sim)
# Get the observations for all agents
obs = sim.reset()
# Get simulation state for all non-done agents, regardless of which agents
# actually contribute an action.
obs, rewards, dones, infos = sim.step({'agent0': 4, 'agent2': [-1, 1]})
```

### 1.1.4 External Integration

In order to train agents in a Simulation Manager using RLlib, we must wrap the simulation with either a *GymWrapper* for single-agent simulations (i.e. only a single entry in the *agents* dict) or a *MultiAgentWrapper* for multiagent simulations.

## 1.2 Training with an Experiment Configuration

In order to run experiments, we must define a configuration file that specifies Simulation and Trainer parameters. Here is the configuration file from the *Corridor tutorial* that demonstrates a simple corridor simulation with multiple agents.

```
# Import the MultiCorridor ABS, a simulation manager, and the multiagent
# wrapper needed to connect to RLlib's trainers
from abmarl.sim.corridor import MultiCorridor
from abmarl.managers import TurnBasedManager
from abmarl.external import MultiAgentWrapper

# Create and wrap the simulation
# NOTE: The agents in `MultiCorridor` are all homogeneous, so this simulation
# just creates and stores the agents itself.
sim = MultiAgentWrapper(TurnBasedManager(MultiCorridor()))

# Register the simulation with RLlib
sim_name = "MultiCorridor"
from ray.tune.registry import register_env
register_env(sim_name, lambda sim_config: sim)

# Set up the policies. In this experiment, all agents are homogeneous,
# so we just use a single shared policy.
ref_agent = sim.unwrapped.agents['agent0']
policies = {
    'corridor': (None, ref_agent.observation_space, ref_agent.action_space, {})
}
def policy_mapping_fn(agent_id):
    return 'corridor'

# Experiment parameters
params = {
    'experiment': {
        'title': f'{sim_name}',
        'sim_creator': lambda config=None: sim,
    },
    'ray_tune': {
        'run_or_experiment': 'PG',
        'checkpoint_freq': 50,
        'checkpoint_at_end': True,
        'stop': {
            'episodes_total': 2000,
        },
        'verbose': 2,
        'config': {
            # --- simulation ---
            'env': sim_name,
            'horizon': 200,
            'env_config': {},
            # --- Multiagent ---
            'multiagent': {
                'policies': policies,
                'policy_mapping_fn': policy_mapping_fn,
```

(continues on next page)

(continued from previous page)

```
            },
            # --- Parallelism ---
            "num_workers": 7,
            "num_envs_per_worker": 1,
        },
    }
}
```

**Warning:** The simulation must be a *Simulation Manager* or an *External Wrapper* as described above.

**Note:** This example has `num_workers` set to 7 for a computer with 8 CPU's. You may need to adjust this for your computer to be *<cpu count> - 1*.

### 1.2.1 Experiment Parameters

The strucutre of the parameters dictionary is very important. It *must* have an *experiment* key which contains both the *title* of the experiment and the *sim_creator* function. This function should receive a config and, if appropriate, pass it to the simulation constructor. In the example configuration above, we just retrun the already-configured simulation. Without the title and simulation creator, Abmarl may not behave as expected.

The experiment parameters also contains information that will be passed directly to RLlib via the *ray_tune* parameter. See RLlib's documentation for a list of common configuration parameters.

### 1.2.2 Command Line

With the configuration file complete, we can utilize the command line interface to train our agents. We simply type `abmarl train multi_corridor_example.py`, where *multi_corridor_example.py* is the name of our configuration file. This will launch Abmarl, which will process the file and launch RLlib according to the specified parameters. This particular example should take 1-10 minutes to train, depending on your compute capabilities. You can view the performance in real time in tensorboard with `tensorboard --logdir ~/abmarl_results`.

## 1.3 Visualizing

We can visualize the agents' learned behavior with the `visualize` command, which takes as argument the output directory from the training session stored in `~/abmarl_results`. For example, the command

```
abmarl visualize ~/abmarl_results/MultiCorridor-2020-08-25_09-30/ -n 5 --record
```

will load the experiment (notice that the directory name is the experiment title from the configuration file appended with a timestamp) and display an animation of 5 episodes. The `--record` flag will save the animations as *.mp4* videos in the training directory.

## 1.4 Analyzing

The simulation and trainer can also be loaded into an analysis script for post-processing via the `analyze` command. The analysis script must implement the following *run* function. Below is an example that can serve as a starting point.

```
# Load the simulation and the trainer from the experiment as objects
def run(sim, trainer):
    """
    Analyze the behavior of your trained policies using the simulation and trainer
    from your RL experiment.

    Args:
        sim:
            Simulation Manager object from the experiment.
        trainer:
            Trainer that computes actions using the trained policies.
    """
    # Run the simulation with actions chosen from the trained policies
    policy_agent_mapping = trainer.config['multiagent']['policy_mapping_fn']
    for episode in range(100):
        print('Episode: {}'.format(episode))
        obs = sim.reset()
        done = {agent: False for agent in obs}
        while True: # Run until the episode ends
            # Get actions from policies
            joint_action = {}
            for agent_id, agent_obs in obs.items():
                if done[agent_id]: continue # Don't get actions for done agents
                policy_id = policy_agent_mapping(agent_id)
                action = trainer.compute_action(agent_obs, policy_id=policy_id)
                joint_action[agent_id] = action
            # Step the simulation
            obs, reward, done, info = sim.step(joint_action)
            if done['__all__']:
                break
```

Analysis can then be performed using the command line interface:

```
abmarl analyze ~/abmarl_results/MultiCorridor-2020-08-25_09-30/ my_analysis_script.py
```

See the *Predator Prey tutorial* for an example of analyzing trained agent behavior.

## 1.5 Running at scale with HPC

Abmarl also supports some functionality for training at scale. See the *magpie tutorial*, which provides a walkthrough for launching a training experiment on multiple compute nodes with slurm.

CHAPTER

TWO

# ABMARL HIGHLIGHTS

## 2.1 Emergent Collaborative and Competitive Behavior

In this experiment, we study how collaborative and competitive behaviors emerge among agents in a partially observable stochastic game. In our simulation, each agent occupies a square and can move around the map. Each agent can "attack" agents that are on a different "team"; the attacked agent loses its life and is removed from the simulation. Each agent can observe the state of the map in a region surrounding its location. It can see other agents and what team they're on as well as the edges of the map. The diagram below visuially depicts the agents' observation and action spaces.

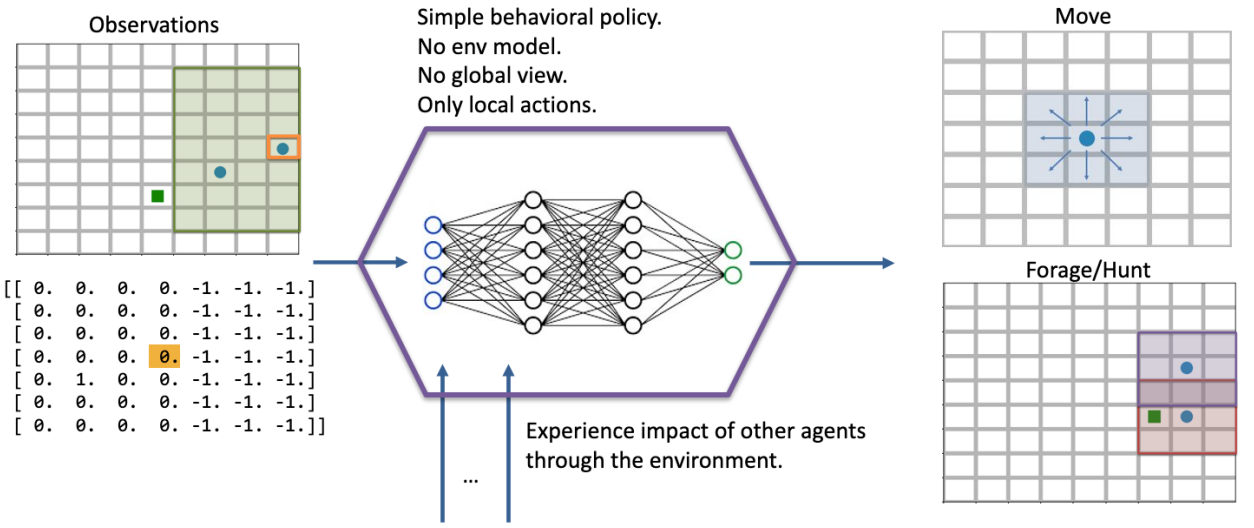



Fig. 1: Each agent has a partial observation of the map centered around its location. The green box shows the orange agent's observation of the map, and the matrix below it shows the actual observation. Each agent can choose to move or to "attack" another agent in one of the nearby squares. The policy is just a simple 2-layer MLP, each layer having 64 units. We don't apply any kind of specialized architecture that encourages collaboration or competition. Each agent is simple: they do not have a model of the simulation; they do not have a global view of the simulation; their actions are only local in both space and in agent interaction (they can only interact with one agent at a time). Yet, we will see efficient and complex strategies emerge, collaboration and competition from the common or conflicting interest among agents.

In the various examples below, each policy is a two-layer MLP, with 64 unites in each layer. We use RLlib's A2C Trainer with default parameters and train for two million episodes on a compute node with 72 CPUs, a process that takes 3-4 hours per experiment.

**Attention:** This page makes heavy use of animated graphics. It is best to read this content on our html site instead of our pdf manual.

### 2.1.1 Single Agent Foraging

We start by considering a single foraging agent whose objective is to move around the map collecting resource agents. The single forager can see up to three squares away, move up to one square away, and forage ("attack") resources up to one square away. The forager is rewarded for every resource it collects and given a small penalty for attempting to move off the map and an even smaller "entropy" penalty every time-step to encourage it to act quickly. At the beginning of every episode, the agents spawn at random locations in the map. Below is a video showing a typical full episode of the learned behavior and a brief analysis.

**Note:** From an Agent Based Modeling perspective, the resources are technically agents themselves. However, since they don't do or see anything, we tend not to call them agents in the text that follows.

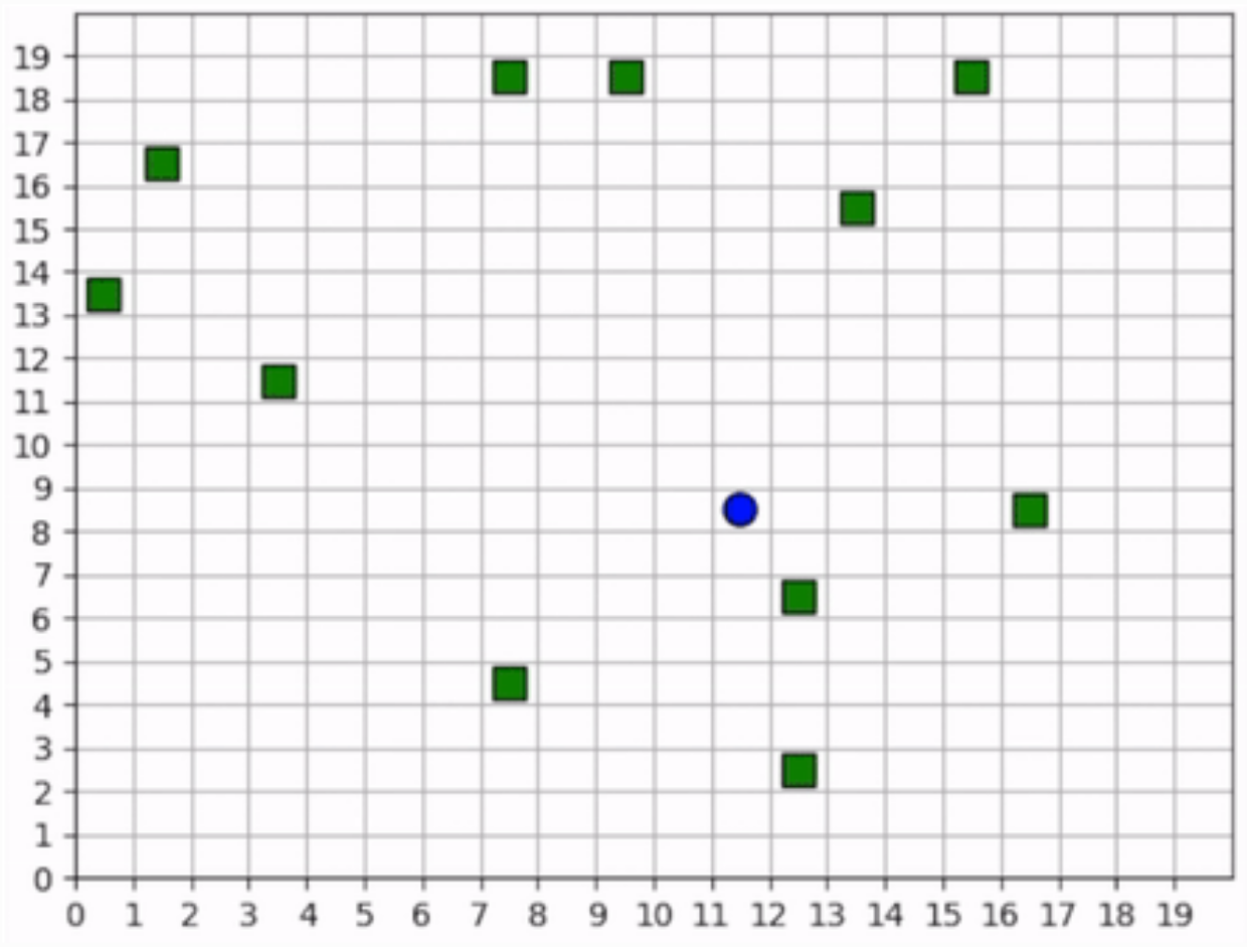



Fig. 2: A full episode showing the forager's learned strategy. The forager is the blue circle and the resources are the green squares. Notice how the forager bounces among resource clusters, greedily collecting all local resources before exploring the map for more.

### When it can see resources

The forager moves toward the closest resource that it observes and collects it. Note that the foraging range is 1 square: the forager rarely waits until it is directly over a resource; it usually forages as soon as it is within range. In some cases, the forager intelligently places itself in the middle of 2-3 resources in order to forage within the least number of moves. When the resources are near the edge of the map, it behaves with some inefficiency, likely due to the small penalty we give it for moving off the map, which results in an aversion towards the map edges. Below is a series of short video clips showing the foraging strategy.

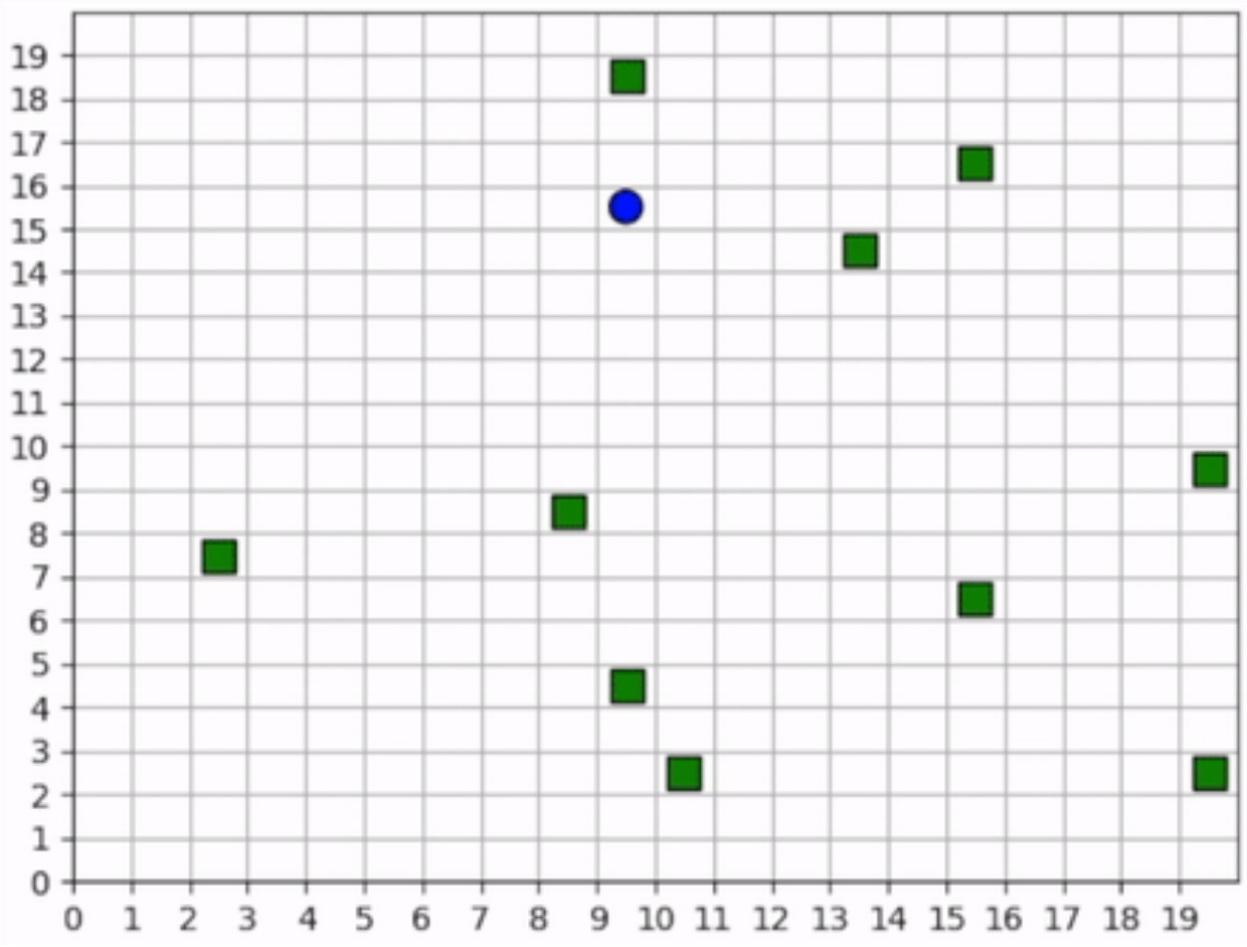



Fig. 3: The forager learns an effective foraging strategy, moving towards and collecting the nearest resources that it observes.

### When it cannot see resources

The forager's behavior when it is near resources is not surprising. But how does it behave when it cannot see any resources? The forager only sees that which is near it and does not have any information distinguishing one "deserted" area of the map from another. Recall, however, that it observes the edges of the map, and it uses this information to learn an effecive exploration strategy. In the video below, we can see that the forager learns to explore the map by moving along its edges in a clockwise direction, occasionally making random moves towards the middle of the map.

---

**Important:** We do not use any kind of heuristic or mixed policy. The exporation strategy *emerges* entirely from

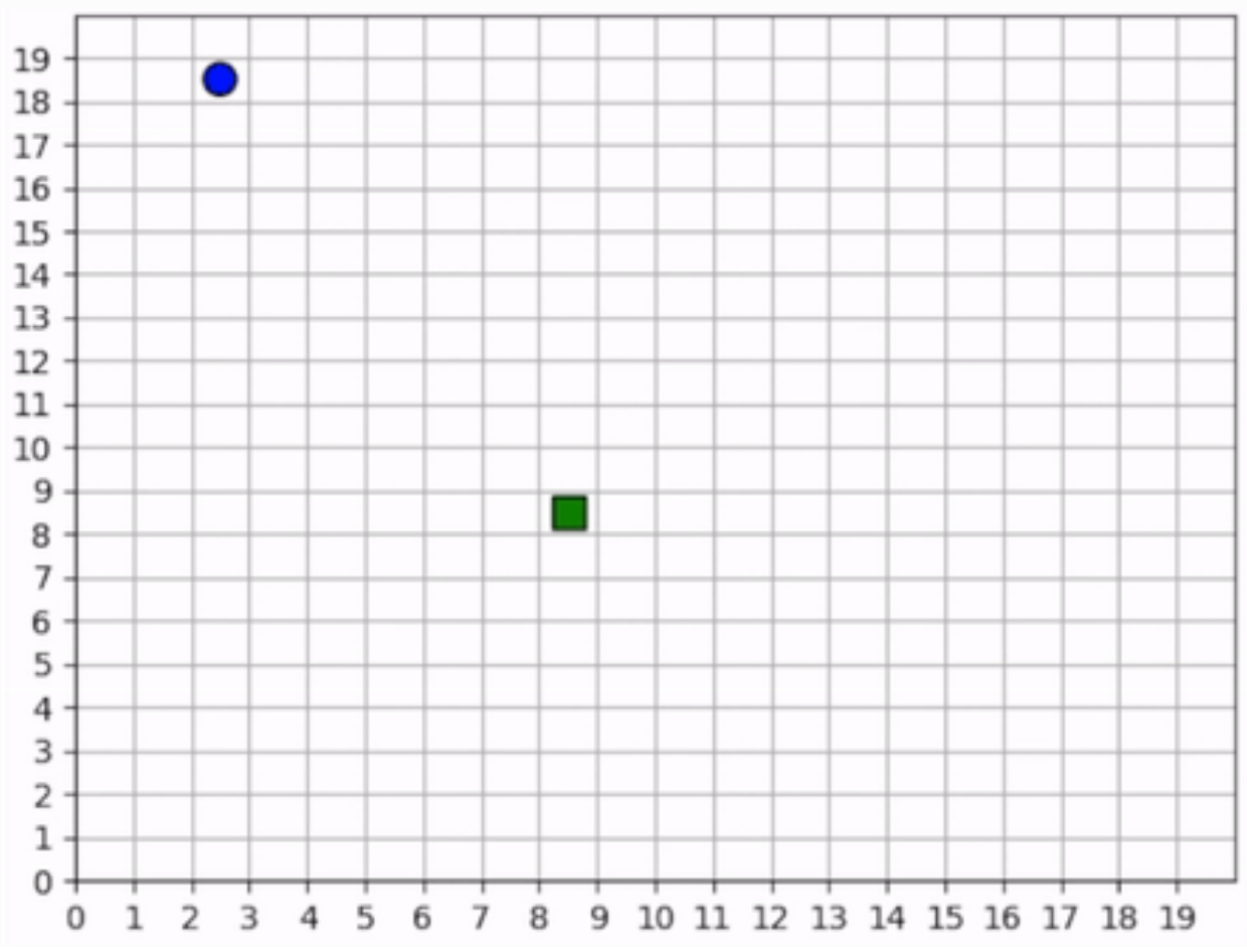



Fig. 4: The forager learns an effective exploration strategy, moving along the edge of the map in a clockwise direction.

reinforcement learning.

### 2.1.2 Multiple Agents Foraging

Having experimented with a single forager, let us now turn our attention to the strategies learned by multiple foragers interacting in the map at the same time. Each forager is homogeneous with each other as described above: they can all move up to one square away, observe up to three squares away, and are rewarded the same way. The observations include other foragers in addition to the resources and map edges. All agents share a single policy. Below is a brief analysis of the learned behaviors.

#### Cover and explore

Our reward schema implicitly encourages the foragers to collaborate because we give a small penalty to each one for taking too long. Thus, the faster they can collect all the resources, the less they are penalized. Furthermore, because each agent trains the same policy, there is no incentive for competitive behavior. An agent can afford to say, "I don't need to get the resource first. As long as one of us gets it quickly, then we all benefit". Therefore, the foragers learn to spread out to *cover* the map, maximizing the amount of squares that are observed.

In the video clips below, we see that the foragers avoid being within observation distance of one another. Typically, when two foragers get too close, they repel each other, each moving in opposite directions, ensuring that the space is *covered*. Furthermore, notice the dance-like exploration strategy. Similar to the single-agent case above, they learn to *explore* along the edges of the map in a clockwise direction. However, they're not as efficient as the single agent because they "repel" each other.

**Important:** We do not directly incentivize agents to keep their distance. No part of the reward schema directly deals with the agents' distances from each other. These strategies are *emergent*.

#### Breaking the pattern

When a forager observes a resource, it breaks its "cover and explore" strategy and moves directly for the resource. Even multiple foragers move towards the same resource. They have no reason to coordinate who will get it because, as we stated above, there is no incentive for competition, so no need to negotiate. If another forager gets there first, everyone benefits. The foragers learn to prioritize collecting the resources over keeping their distance from each other.

**Tip:** We should expect to see both of these strategies occuring at the same time within a simulation because while some agents are "covering and exploring", others are moving towards resources.

### 2.1.3 Introducing Hunters

So far, we have seen intelligent behaviors emerge in both single- and multi-forager scenarios; we even saw the emergence of collaborative behavior. In the following experiments, we explore competitive emergence by introducing hunters into the simulation. Like foragers, hunters can move up to one square away and observe other agents and map edges up to three squares away. Hunters, however, are more effective killers and can attack a forager up to two squares away. They are rewarded for successful kills, they are and penalized for bad moves and for taking too long, exactly the same way as foragers.

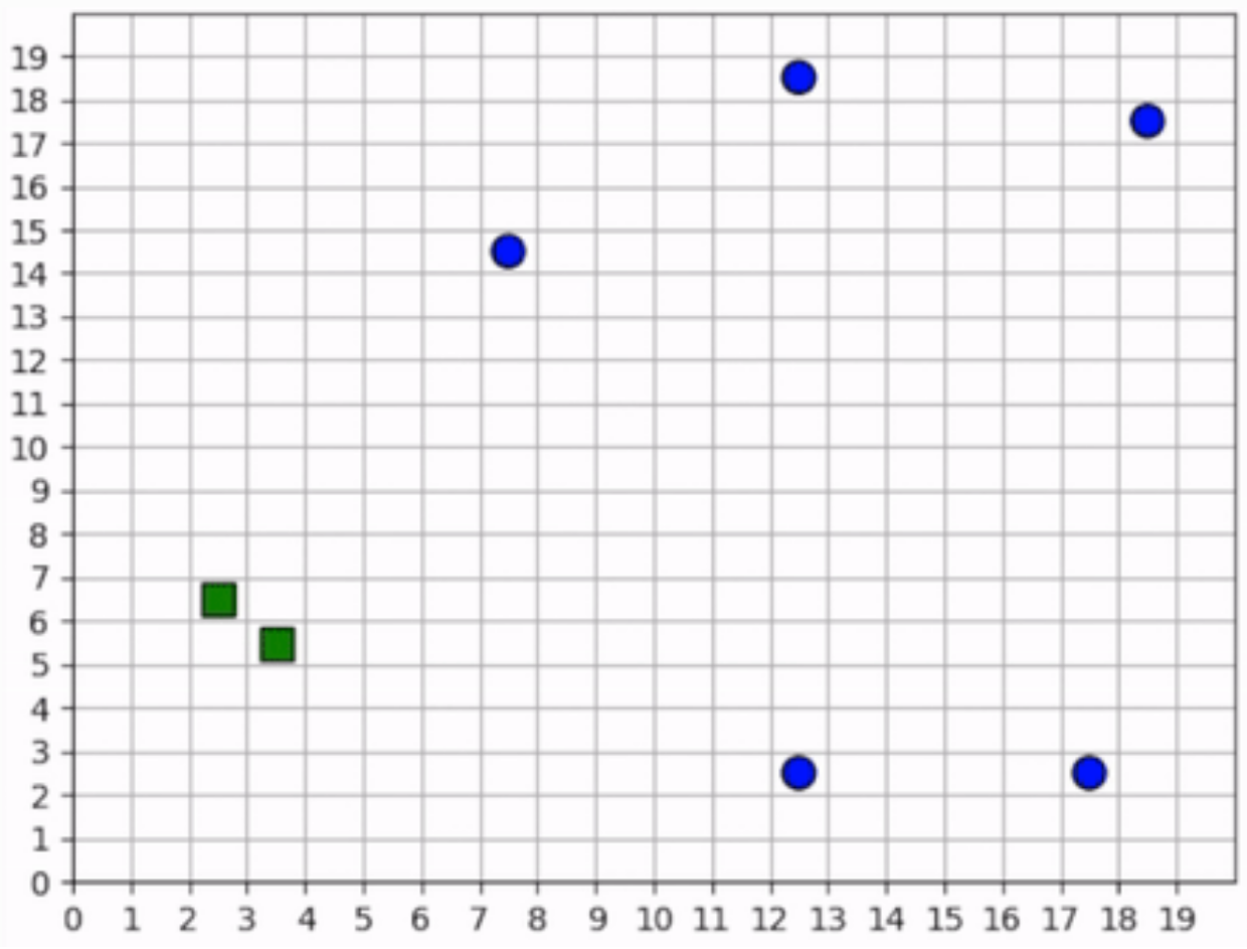



Fig. 5: The foragers cover the map by spreading out and explore it by traveling in a clockwise direction.

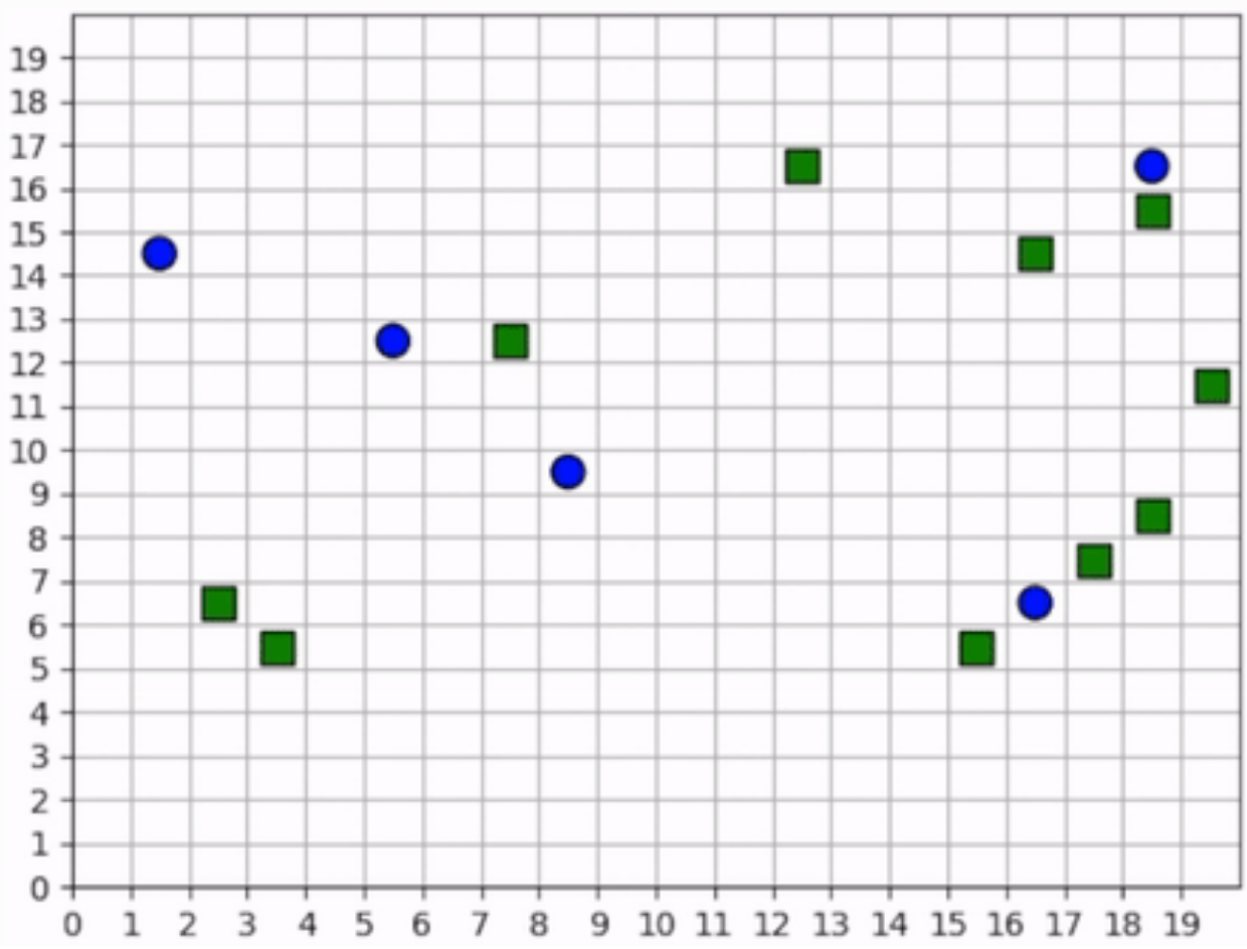
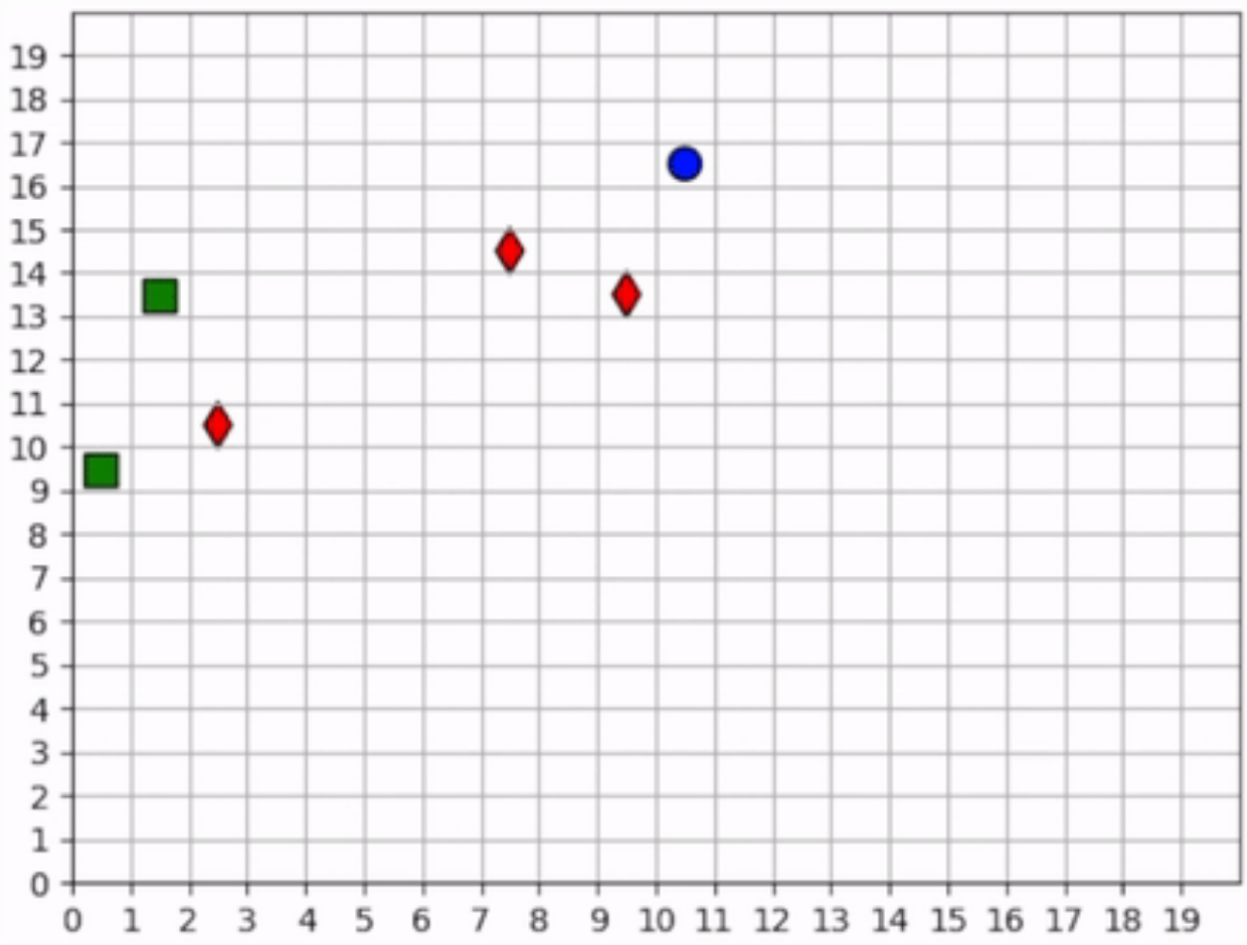



Fig. 6: The foragers move towards resources to forage, even when there are other foragers nearby.

However, the hunters and foragers have completely different objectives: a forager tries to clear the map of all *resources*, but a hunter tries to clear the map of all *foragers*. Therefore, we set up two policies. All the hunters will train the same policy, and all the foragers will train the same policy, and these policies will be distinct.

The learned behaviors among the two groups in this mixed collaborate-competitive simulation are tightly integrated, with multiple strategies appearing at the same time within a simulation. Therefore, in contrast to above, we will not show video clips that capture a single strategy; instead, we will show video clips that capture multiple strategies and attempt to describe them in detail.

### First Scenario



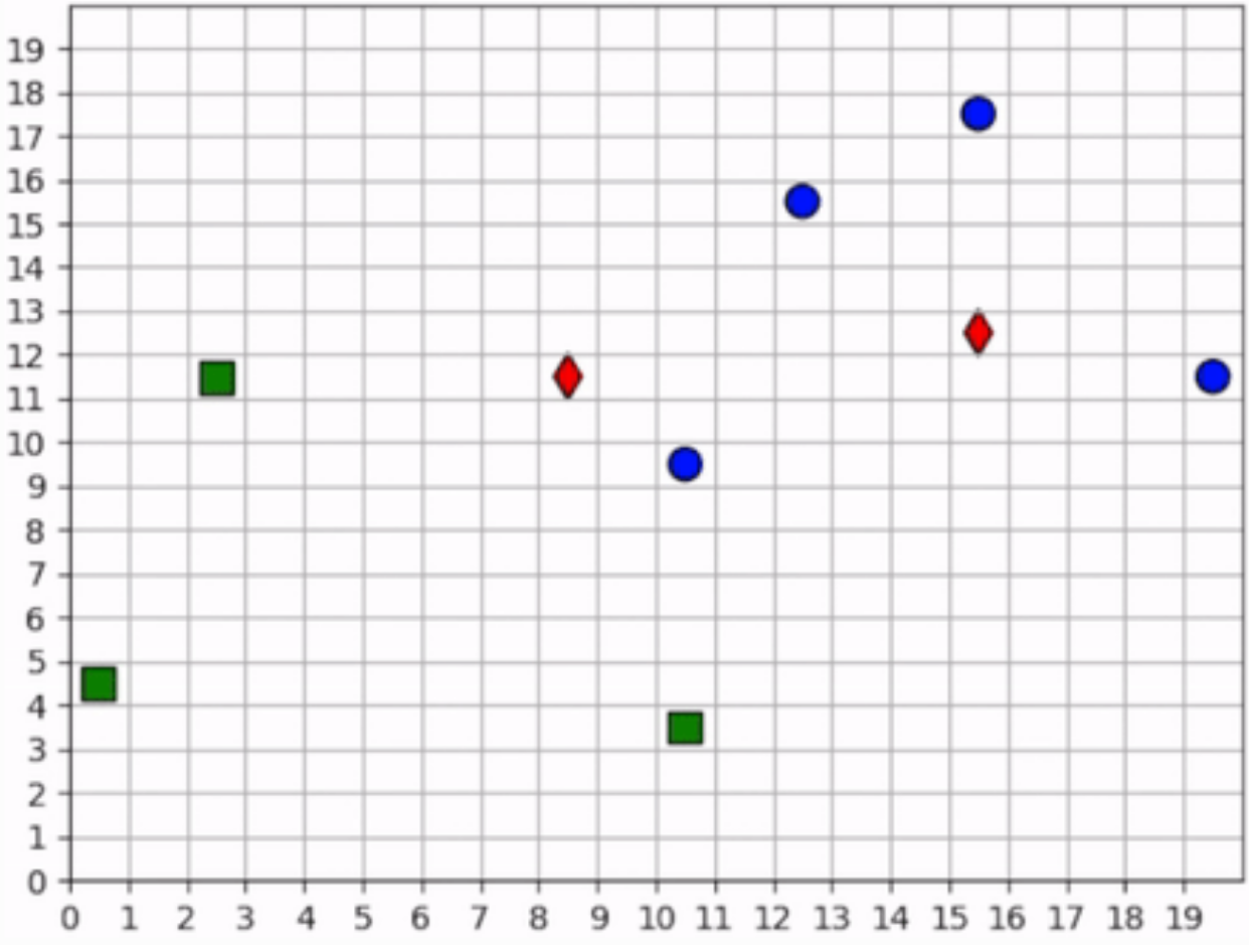

Two of the foragers spawn next to hunters and are killed immediately. Afterwards, the two hunters on the left do not observe any foragers for some time. They seem to have learned the *cover* strategy by spreading out, but they don't seem to have learned an effecient *explore* strategy since they mostly occupy the same region of the map for the duration of the simulation.

Three foragers remain at the bottom of the map. These foragers work together to collect all nearby resources. Just as they finish the resource cluster, a hunter moves within range and begins to chase them towards the bottom of the map. When they hit the edge, they split in two directions. The hunter kills one of them and then waits for one step, unsure about which forager to persue next. After one step, we see that it decides to persue the forager to the right.

Meanwhile, the forager to the left continues to run away, straight into the path of another hunter but also another resource. The forager could get away by running to the right, but it decides to collect the resource at the cost of its own life.

The last remaining forager has escaped the hunter and has conveniently found another cluster of resources, which it collects. A few frames later, it encounters the same hunter, and this time it is chased all the way across the map. It manages to evade the hunter and collect one final resource before encountering yet another hunter. At the end, we see both hunters chasing the forager to the top of the map, boxing it in and killing it.

### Second scenario




None of the foragers are under threat at the beginning of this scenario. They clear a cluster of resources before one of them wanders into the path of a hunter. The hunter gives chase, and the forager actually leads the hunter back to the group. This works to its benefit, however, as the hunter is repeatedly confused by the foragers exercising the *splitting* strategy. Meanwhile the second hunter has spotted a forager and joins the hunt. The two hunters together are able to split up the pack of foragers and systematically hunt them down. The last forager is chased into the corner and killed.

**Note:** Humorously, the first forager that was spotted is the one who manages to stay alive the longest.

CHAPTER

THREE

# INSTALLATION

## 3.1 User Installation

You can install abmarl via *pip*:

```
pip install abmarl
```

**Attention:** Upload the wheel to pip and confirm.

## 3.2 Developer Installation

To install Abmarl for development, first clone the repository and then install via pip's development mode:

```
git clone git@github.com:LLNL/Abmarl.git
cd abmarl
pip install -r requirements.txt
pip install -e . --no-deps
```

CHAPTER

# FOUR

# FULL TUTORIALS

We provide tutorials that demonstrate how to train, visualize, and analyze MARL policies.

## 4.1 MultiCorridor

MultiCorridor extends RLlib's simple corridor, wherein agents must learn to move to the right in a one-dimensonal corridor to reach the end. Our implementation provides the ability to instantiate multiple agents in the simulation and restricts agents from occupying the same square. Every agent is homogeneous: they all have the same action space, observation space, and objective function.

Fig. 1: Animation of agents moving left and right in a corridor until they reach the end.

### 4.1.1 Creating the MultiCorridor Simulation

#### The Agents in the Simulation

It's helpful to start by thinking about what we want the agents to learn and what information they will need in order to learn it. In this tutorial, we want to train agents that can reach the end of a one-dimensional corridor without bumping into each other. Therefore, agents should be able to move left, move right, and stay still. In order to move to the end of the corridor without bumping into each other, they will need to see their own position and if the squares near them are occupied. Finally, we need to decide how to reward the agents. There are many ways we can do this, and we should at least capture the following:

- The agent should be rewarded for reaching the end of the corridor.
- The agent should be penalized for bumping into other agents.
- The agent should be penalized for taking too long.

Since all our agents are homogeneous, we can create them in the Agent Based Simulation itself, like so:

```
from enum import IntEnum

from gym.spaces import Box, Discrete, MultiBinary
import numpy as np

from abmarl.sim import Agent, AgentBasedSimulation

class MultiCorridor(AgentBasedSimulation):

    class Actions(IntEnum): # The three actions each agent can take
        LEFT = 0
        STAY = 1
        RIGHT = 2

    def __init__(self, end=10, num_agents=5):
        self.end = end
        agents = {}
        for i in range(num_agents):
            agents[f'agent{i}'] = Agent(
                id=f'agent{i}',
                action_space=Discrete(3), # Move left, stay still, or move right
                observation_space={
                    'position': Box(0, self.end-1, (1,), np.int), # Observe your own␣
↪position
                    'left': MultiBinary(1), # Observe if the left square is occupied
                    'right': MultiBinary(1) # Observe if the right square is occupied
                }
            )
        self.agents = agents

        self.finalize()
```

Here, notice how the agents' *observation_space* is a *dict* rather than a *gym.space.Dict*. That's okay because our *Agent* class can convert a *dict of gym spaces* into a *Dict* when `finalize` is called at the end of `__init__`.

### Resetting the Simulation

At the beginning of each episode, we want the agents to be randomly positioned throughout the corridor without occupying the same squares. We must give each agent a position attribute at reset. We will also create a data structure that captures which agent is in which cell so that we don't have to do a search for nearby agents but can directly index the space. Finally, we must track the agents' rewards.

```
def reset(self, **kwargs):
    location_sample = np.random.choice(self.end-1, len(self.agents), False)
    # Track the squares themselves
    self.corridor = np.empty(self.end, dtype=object)
    # Track the position of the agents
    for i, agent in enumerate(self.agents.values()):
        agent.position = location_sample[i]
        self.corridor[location_sample[i]] = agent

    # Track the agents' rewards over multiple steps.
```

(continues on next page)

(continued from previous page)

```
    self.reward = {agent_id: 0 for agent_id in self.agents}
```

### Stepping the Simulation

The simulation is driven by the agents' actions because there are no other dynamics. Thus, the MultiCorridor Simulation only concerns itself with processing the agents' actions at each step. For each agent, we'll capture the following cases:

- An agent attempts to move to a space that is unoccupied.
- An agent attempts to move to a space that is already occupied.
- An agent attempts to move to the right-most space (the end) of the corridor.

```
def step(self, action_dict, **kwargs):
    for agent_id, action in action_dict.items():
        agent = self.agents[agent_id]
        if action == self.Actions.LEFT:
            if agent.position != 0 and self.corridor[agent.position-1] is None:
                # Good move, no extra penalty
                self.corridor[agent.position] = None
                agent.position -= 1
                self.corridor[agent.position] = agent
                self.reward[agent_id] -= 1 # Entropy penalty
            elif agent.position == 0: # Tried to move left from left-most square
                # Bad move, only acting agent is involved and should be penalized.
                self.reward[agent_id] -= 5 # Bad move
            else: # There was another agent to the left of me that I bumped into
                # Bad move involving two agents. Both are penalized
                self.reward[agent_id] -= 5 # Penalty for offending agent
                # Penalty for offended agent
                self.reward[self.corridor[agent.position-1].id] -= 2
        elif action == self.Actions.RIGHT:
            if self.corridor[agent.position + 1] is None:
                # Good move, but is the agent done?
                self.corridor[agent.position] = None
                agent.position += 1
                if agent.position == self.end-1:
                    # Agent has reached the end of the corridor!
                    self.reward[agent_id] += self.end ** 2
                else:
                # Good move, no extra penalty
                    self.corridor[agent.position] = agent
                    self.reward[agent_id] -= 1 # Entropy penalty
            else: # There was another agent to the right of me that I bumped into
                # Bad move involving two agents. Both are penalized
                self.reward[agent_id] -= 5 # Penalty for offending agent
                # Penalty for offended agent
                self.reward[self.corridor[agent.position+1].id] -= 2
        elif action == self.Actions.STAY:
            self.reward[agent_id] -= 1 # Entropy penalty
```

**Attention:** Our reward schema reveals a training dynamic that is not present in single-agent simulations: an agent's reward does not entirely depend on its own interaction with the simulation but can be affected by other agents' actions. In this case, agents are slightly penalized for being "bumped into" when other agents attempt to move onto their square, even though the "offended" agent did not directly cause the collision. This is discussed in MARL literature and captured in the way we have designed our Simulation Managers. In Abmarl, we favor capturing the rewards as part of the simulation's state and only "flushing" them once they rewards are asked for in `get_reward`.

**Note:** We have not needed to consider the order in which the simulation processes actions. Our simulation simply provides the capabilities to process *any* agent's action, and we can use *Simulation Managers* to impose an order. This shows the flexibility of our design. In this tutorial, we will use the *TurnBasedManager*, but we can use any *SimulationManager*.

### Querying Simulation State

The trainer needs to see how agents' actions impact the simulation's state. They do so via getters, which we define below.

```
def get_obs(self, agent_id, **kwargs):
    agent_position = self.agents[agent_id].position
    if agent_position == 0 or self.corridor[agent_position-1] is None:
        left = False
    else:
        left = True
    if agent_position == self.end-1 or self.corridor[agent_position+1] is None:
        right = False
    else:
        right = True
    return {
        'position': [agent_position],
        'left': [left],
        'right': [right],
    }

def get_done(self, agent_id, **kwargs):
    return self.agents[agent_id].position == self.end - 1

def get_all_done(self, **kwargs):
    for agent in self.agents.values():
        if agent.position != self.end - 1:
            return False
    return True

def get_reward(self, agent_id, **kwargs):
    agent_reward = self.reward[agent_id]
    self.reward[agent_id] = 0
    return agent_reward

def get_info(self, agent_id, **kwargs):
    return {}
```

#### Rendering for Visualization

Finally, it's often useful to be able to visualize a simulation as it steps through an episode. We can do this via the render funciton.

```
def render(self, *args, fig=None, **kwargs):
    draw_now = fig is None
    if draw_now:
        from matplotlib import pyplot as plt
        fig = plt.gcf()

    fig.clear()
    ax = fig.gca()
    ax.set(xlim=(-0.5, self.end + 0.5), ylim=(-0.5, 0.5))
    ax.set_xticks(np.arange(-0.5, self.end + 0.5, 1.))
    ax.scatter(np.array(
        [agent.position for agent in self.agents.values()]),
        np.zeros(len(self.agents)),
        marker='s', s=200, c='g'
    )

    if draw_now:
        plt.plot()
        plt.pause(1e-17)
```

### 4.1.2 Training the MultiCorridor Simulation

Now that we have created the simulation and agents, we can create a configuration file for training.

#### Simulation Setup

We'll start by setting up the simulation we have just built. Then we'll choose a Simulation Manager. Abmarl comes with two built-In managers: *TurnBasedManager*, where only a single agent takes a turn per step, and *AllStepManager*, where all non-done agents take a turn per step. For this experiment, we'll use the *TurnBasedManager*. Then, we'll wrap the simulation with our *MultiAgentWrapper*, which enables us to connect with RLlib. Finally, we'll register the simulation with RLlib.

```
# MultiCorridor is the simulation we created above
from abmarl.sim.corridor import MultiCorridor
from abmarl.managers import TurnBasedManager
# MultiAgentWrapper needed to connect with RLlib
from abmarl.external import MultiAgentWrapper

# Create an instance of the simulation and register it
sim = MultiAgentWrapper(TurnBasedManager(MultiCorridor()))
sim_name = "MultiCorridor"
from ray.tune.registry import register_env
register_env(sim_name, lambda sim_config: sim)
```

### Policy Setup

Now we want to create the policies and the policy mapping function in our multiagent experiment. Each agent in our simulation is homogeneous: they all have the same observation space, action space, and objective function. Thus, we can create a single policy and map all agents to that policy.

```
ref_agent = sim.unwrapped.agents['agent0']
policies = {
    'corridor': (None, ref_agent.observation_space, ref_agent.action_space, {})
}
def policy_mapping_fn(agent_id):
    return 'corridor'
```

### Experiment Parameters

Having setup the simulation and policies, we can now bundle all that information into a parameters dictionary that will be read by Abmarl and used to launch RLlib.

```
params = {
    'experiment': {
        'title': f'{sim_name}',
        'sim_creator': lambda config=None: sim,
    },
    'ray_tune': {
        'run_or_experiment': 'PG',
        'checkpoint_freq': 50,
        'checkpoint_at_end': True,
        'stop': {
            'episodes_total': 2000,
        },
        'verbose': 2,
        'config': {
            # --- Simulation ---
            'env': sim_name,
            'horizon': 200,
            'env_config': {},
            # --- Multiagent ---
            'multiagent': {
                'policies': policies,
                'policy_mapping_fn': policy_mapping_fn,
            },
            # --- Parallelism ---
            # Number of workers per experiment: int
            "num_workers": 7,
            # Number of simulations that each worker starts: int
            "num_envs_per_worker": 1, # This must be 1 because we are not "threadsafe"
        },
    }
}
```

#### Command Line interface

With the configuration file complete, we can utilize the command line interface to train our agents. We simply type `abmarl train multi_corridor_example.py`, where *multi_corridor_example.py* is the name of our configuration file. This will launch Abmarl, which will process the file and launch RLlib according to the specified parameters. This particular example should take 1-10 minutes to train, depending on your compute capabilities. You can view the performance in real time in tensorboard with `tensorboard --logdir ~/abmarl_results`.

#### Visualizing the Trained Behaviors

We can visualize the agents' learned behavior with the `visualize` command, which takes as argument the output directory from the training session stored in `~/abmarl_results`. For example, the command

```
abmarl visualize ~/abmarl_results/MultiCorridor-2020-08-25_09-30/ -n 5 --record
```

will load the experiment (notice that the directory name is the experiment title from the configuration file appended with a timestamp) and display an animation of 5 episodes. The `--record` flag will save the animations as *.mp4* videos in the training directory.

### 4.1.3 Extra Challenges

Having successfully trained a MARL experiment, we can further explore the agents' behaviors and the training process. Some ideas are:

- We could enhance the MultiCorridor Simulation so that the "target" cell is a different location in each episode.
- We could introduce heterogeneous agents with the ability to "jump over" other agents. With heterogeneous agents, we can nontrivially train multiple policies.
- We could study how the agents' behaviors differ if they are trained using the *AllStepManager*.
- We could create our own Simulation Manager so that if an agent causes a collision, it skips its next turn.
- We could do a parameter search over both simulation and algorithm parameters to study how the parameters affect the learned behaviors.
- We could analyze how often agents collide with one another and where those collisions most commonly occur.
- And much, much more!

As we attempt these extra challenges, we will experience one of Abmarl's strongest features: the ease with which we can modify our experiment file and launch another training job, going through the pipeline from experiment setup to behavior visualization and analysis!

## 4.2 PredatorPrey

PredatorPrey is a multiagent simulation useful for exploring competitve behaviors between groups of agents. Resources "grow" in a two-dimensional grid. Prey agents move around the grid harvesting resources, and predator agents move around the grid hunting the prey agents.

**Attention:** This tutorial requires seaborn for visualizing the resources. This can be easily added to your virtual environment with `pip install seaborn`.

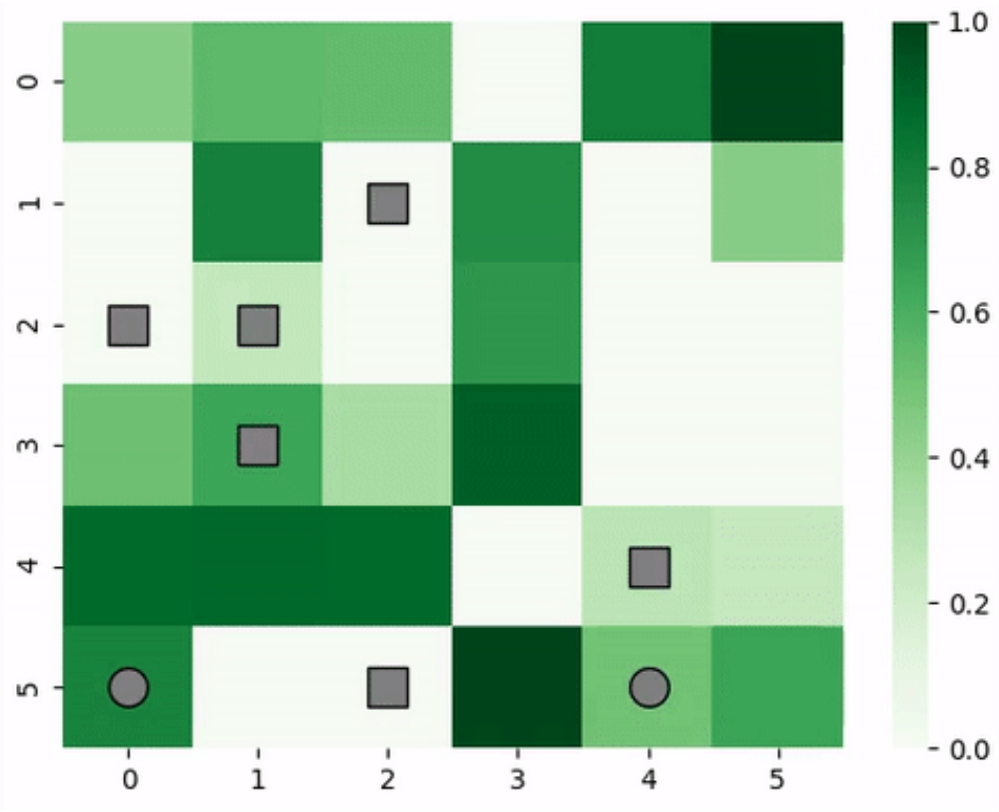



Fig. 2: Animation of predator and prey agents in a two-dimensional grid.

### 4.2.1 Creating the PredatorPrey Simulation

#### The Agents in the Simulation

In this tutorial, we will train predators to hunt prey by moving around the grid and attacking them when they are nearby. In order to learn this, they must be able to see a subset of the grid around their position, and they must be able to distinguish between other predators and prey. We will reward the predators as follows:

- The predator should be rewarded for successfully killing a prey.
- The predator should be penalized for trying to move off the edge of the grid.
- The predator should be penalized for taking too long.

Concurrently, we will train prey agents to harvest resources while attempting to avoid predators. To learn this, prey must be able to see a subset off the grid around them, both the resources available and any other agents. We will reward the prey as follows:

- The prey should be rewarded for harvesting resources.
- The prey should be penalized for trying to move off the edge of the grid.
- The prey should be penalized for getting eaten by a predator.
- The prey should be penalized for taking too long.

In order to accomodate this, we will create two types of Agents, one for Predators and one for Prey. Notice that all agents can move around and view a subset of the grid, so we'll capture this in a parent class and encode the distinction in the agents' respective child classes.

```
from abc import ABC, abstractmethod

from gym.spaces import Box, Discrete, Dict
import numpy as np

from abmarl.sim import PrincipleAgent, AgentBasedSimulation

class PredatorPreyAgent(PrincipleAgent, ABC):
    @abstractmethod
    def __init__(self, move=None, view=None, **kwargs):
        super().__init__(**kwargs)
        self.move = move
        self.view = view

    @property
    def configured(self):
        return super().configured and self.move is not None and self.view is not None

class Prey(PredatorPreyAgent):
    def __init__(self, harvest_amount=None, **kwargs):
        super().__init__(**kwargs)
        self.harvest_amount = harvest_amount

    @property
    def configured(self):
        return super().configured and self.harvest_amount is not None

    @property
```

(continues on next page)

(continued from previous page)

```
    def value(self):
        return 1

class Predator(PredatorPreyAgent):
    def __init__(self, attack=None, **kwargs):
        super().__init__(**kwargs)
        self.attack = attack

    @property
    def configured(self)
        return super().configured and self.attack is not None

    @property
    def value(self):
        return 2
```

#### The PredatorPrey Simulation

The PredatorPrey Simulation needs much detailed explanation, which we believe will distract from this tutorial. Suffice it to say that we have created a simulation that works with the above agents and captures our desired features. This simulation can be found in full in our repo.

### 4.2.2 Training the Predator Prey Simulation

With the PredatorPrey simulation and agents at hand, we can create a configuration file for training.

#### Simulation Setup

Setting up the PredatorPrey simulation requires us to explicity make agents and pass those to the simulation builder. Once we've done that, we can choose which *SimulationManager* to use. In this tutorial, we'll use the *AllStepManager*. Then, we'll wrap the simulation with our *MultiAgentWrapper*, which enables us to connect with RLlib. Finally, we'll register the simulation with RLlib.

#### Policy Setup

Next, we will create the policies and the policy mapping function. Because predators and prey are competitve, they must train separate polices from one another. Furthermore, since each prey is homogeneous with other prey and each predator with other predators, we can have them train the same policy. Thus, we will have two policies: one for predators and one for prey.

### Experiment Parameters

The last thing is to wrap all the parameters together into a single *params* dictionary. Below is the full configuration file:

```
# Setup the simulation
from abmarl.sim.predator_prey import PredatorPreySimulation, Predator, Prey
from abmarl.managers import AllStepManager

region = 6
predators = [Predator(id=f'predator{i}', attack=1) for i in range(2)]
prey = [Prey(id=f'prey{i}') for i in range(7)]
agents = predators + prey

sim_config = {
    'region': region,
    'max_steps': 200,
    'agents': agents,
}
sim_name = 'PredatorPrey'

from abmarl.external.rllib_multiagentenv_wrapper import MultiAgentWrapper
from ray.tune.registry import register_env
sim = MultiAgentWrapper(AllStepManager(PredatorPreySimulation.build(sim_config)))
agents = sim.unwrapped.agents
register_env(sim_name, lambda sim_config: sim)

# Set up policies
policies = {
    'predator': (None, agents['predator0'].observation_space, agents['predator0'].action_
↪space, {}),
    'prey': (None, agents['prey0'].observation_space, agents['prey0'].action_space, {})
}
def policy_mapping_fn(agent_id):
    if agent_id.startswith('prey'):
        return 'prey'
    else:
        return 'predator'

# Experiment parameters
params = {
    'experiment': {
        'title': '{}'.format('PredatorPrey'),
        'sim_creator': lambda config=None: sim,
    },
    'ray_tune': {
        'run_or_experiment': "PG",
        'checkpoint_freq': 50,
        'checkpoint_at_end': True,
        'stop': {
            'episodes_total': 20_000,
        },
        'verbose': 2,
```

(continues on next page)

(continued from previous page)

```
        'config': {
            # --- Simulation ---
            'env': sim_name,
            'env_config': sim_config,
            'horizon': 200,
            # --- Multiagent ---
            'multiagent': {
                'policies': policies,
                'policy_mapping_fn': policy_mapping_fn,
            },
            # "lr": 0.0001,
            # --- Parallelism ---
            # Number of workers per experiment: int
            "num_workers": 7,
            # Number of simulations that each worker starts: int
            "num_envs_per_worker": 1, # This must be 1 because we are not "threadsafe"
            # 'simple_optimizer': True,
            # "postprocess_inputs": True
        },
    }
}
```

### Using the Command Line

#### Training

With the configuration script complete, we can utilize the command line interface to train our predator. We simply type `abmarl train predator_prey_training.py`, where *predator_prey_training.py* is our configuration file. This will launch Abmarl, which will process the script and launch RLlib according to the specified parameters. This particular example should take about 10 minutes to train, depending on your compute capabilities. You can view the performance in real time in tensorboard with `tensorboard --logdir ~/abmarl_results`. We can find the rewards associated with the policies on the second page of tensorboard.

#### Visualizing

Having successfully trained predators to attack prey, we can vizualize the agents' learned behavior with the *visualize* command, which takes as argument the output directory from the training session stored in *~/abmarl_results*. For example, the command

```
abmarl visualize ~/abmarl_results/PredatorPrey-2020-08-25_09-30/ -n 5 --record
```

will load the training session (notice that the directory name is the experiment title from the configuration script appended with a timestamp) and display an animation of 5 episodes. The *–record* flag will save the animations as *.mp4* videos in the training directory.

### Analyzing

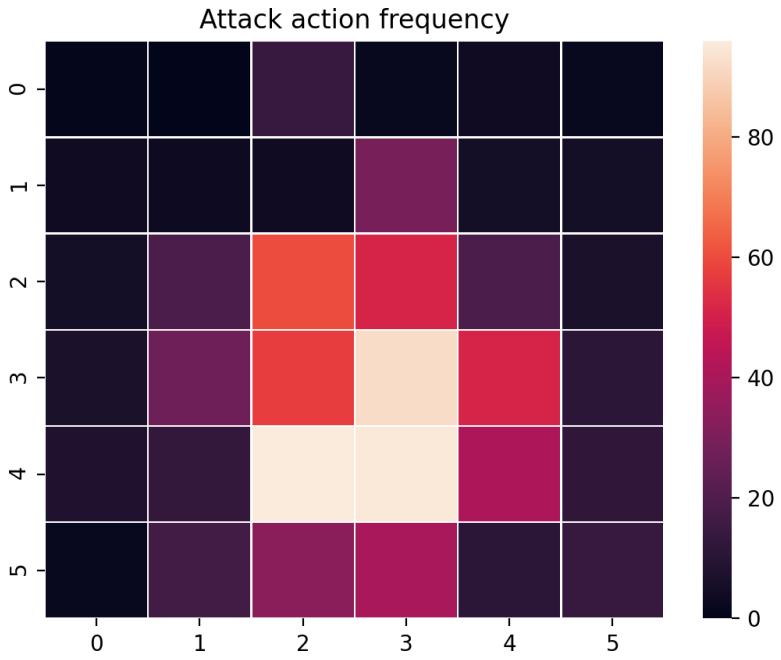
We can further investigate the learned behaviors using the *analyze* command along with an analysis script. Analysis scripts implement a *run* command which takes the Simulation and the Trainer as input arguments. We can define any script to further investigate the agents' behavior. In this example, we will craft a script that records how often a predator attacks from each grid square.

```
def run(sim, trainer):
    import numpy as np
    import seaborn as sns
    import matplotlib.pyplot as plt

    # Create a grid
    grid = np.zeros((sim.sim.region, sim.sim.region))
    attack = np.zeros((sim.sim.region, sim.sim.region))

    # Run the trained policy
    policy_agent_mapping = trainer.config['multiagent']['policy_mapping_fn']
    for episode in range(100): # Run 100 trajectories
        print('Episode: {}'.format(episode))
        obs = sim.reset()
        done = {agent: False for agent in obs}
        pox, poy = sim.agents['predator0'].position
        grid[pox, poy] += 1
        while True:
            joint_action = {}
            for agent_id, agent_obs in obs.items():
                if done[agent_id]: continue # Don't get actions for dead agents
                policy_id = policy_agent_mapping(agent_id)
                action = trainer.compute_action(agent_obs, policy_id=policy_id)
                joint_action[agent_id] = action
            obs, _, done, _ = sim.step(joint_action)
            pox, poy = sim.agents['predator0'].position
            grid[pox, poy] += 1
            if joint_action['predator0']['attack'] == 1: # This is the attack action
                attack[pox, poy] += 1
            if done['__all__']:
                break

    plt.figure(2)
    plt.title("Attack action frequency")
    ax = sns.heatmap(np.flipud(np.transpose(attack)), linewidth=0.5)

    plt.show()
```

We can run this analysis with

```
abmarl analyze ~/abmarl_results/PredatorPrey-2020-08-25_09-30/ movement_map.py
```

which renders the following image for us




The heatmap figures indicate that the predators spend most of their time attacking prey from the center of the map and rarely ventures to the corners.

---

**Note:** Creating the analysis script required some in-depth knowledge about the inner workings of the PredatorPrey Simulation. This will likely be needed when analyzing most simulation you work with.

---

### 4.2.3 Extra Challenges

Having successfully trained the predators to attack prey experiment, we can further explore the agents' behaviors and the training process. For example, you may have noticed that the prey agents didn't seem to learn anything. We may need to improve our reward schema for the prey or modify the way agents interact in the simulation. This is left open to exploration.

## 4.3 Magpie

The prospect of applying MuliAgent Reinforcement Learning algorithms on HPC systems is very attractive. As a first step, we demonstrate that abmarl can be used with magpie to create batch jobs for running on multiple compute nodes.

### 4.3.1 Installing Abmarl on HPC systems

Here we'll use conda to install on an HPC system:

- Create the conda virtual environment: *conda create –name abmarl*
- Activate it: *conda activate abmarl*
- Install pip installer: *conda install –name abmarl pip*
- Follow *installation instructions*

### 4.3.2 Usage

We demonstrate running the *PredatorPrey tutorial* using Mapgie.

#### make-runnable

Abmarl's command line interface provides the *make-runnable* subcommand that converts the configuration script into a runnable script and saves it to the same directory.

```
abmarl make-runnable predator_prey_training.py
```

This will create a file called *runnable_predator_prey_training.py*.

#### Magpie flag

The full use of *make-runnable* is seen when it is run with the `--magpie` flag. This will create a custom magpie script using magpie's ray default script as a starting point. This also adds the correct initialization parameters to *ray.init()* in the *runnable_* script. For example,

```
abmarl make-runnable predator_prey_training.py --magpie
```

will create the *runnable_* script with `ray.init(address=os.environ['MAGPIE_RAY_ADDRESS'])` and will create a magpie batch script that is setup to run this example. To launch the batch job, we simply run it from the command line:

```
sbatch -k --ip-isolate=yes PredatorPrey_magpie.sbatch-srun-ray
```

The script can be modified to adjust the job parameters, such as the number of compute nodes, the time limit for the job, etc. This can also be done through abmarl via the `-n` and `-t` options.

**Attention:** the *num_workers* parameter in the tune configuration is the number of processors to utilize per compute node, which is the different from the number of compute nodes you are requesting.

CHAPTER

FIVE

# ABMARL API SPECIFICATION

## 5.1 Abmarl Simulations

**class** abmarl.sim.**PrincipleAgent**(*id=None*, *seed=None*, ***kwargs*)
Principle Agent class for agents in a simulation.

**property configured**
All agents must have an id.

**finalize**(***kwargs*)

**property id**

**property seed**
Seed for random number generation.

**class** abmarl.sim.**ObservingAgent**(*observation_space=None*, ***kwargs*)
ObservingAgents can observe the state of the simulation.

The agent's observation must be *in* its observation space. The SimulationManager will send the observation to the Trainer, which will use it to produce actions.

**property configured**
Observing agents must have an observation space.

**finalize**(***kwargs*)
Wrap all the observation spaces with a Dict and seed it if the agent was created with a seed.

**property observation_space**

**class** abmarl.sim.**ActingAgent**(*action_space=None*, ***kwargs*)
ActingAgents can act in the simulation.

The Trainer will produce actions for the agents and send them to the SimulationManager, which will process those actions in its step function.

**property action_space**

**property configured**
Acting agents must have an action space.

**finalize**(***kwargs*)
Wrap all the action spaces with a Dict if applicable and seed it if the agent was created with a seed.

**class** abmarl.sim.**Agent**(*observation_space=None*, ***kwargs*)
Bases: *abmarl.sim.agent_based_simulation.ObservingAgent*, *abmarl.sim.agent_based_simulation.ActingAgent*

An Agent that can both observe and act.

**class** abmarl.sim.**AgentBasedSimulation**

AgentBasedSimulation interface.

Under this design model the observations, rewards, and done conditions of the agents is treated as part of the simulations internal state instead of as output from reset and step. Thus, it is the simulations responsibility to manage rewards and dones as part of its state (e.g. via self.rewards dictionary).

This interface supports both single- and multi-agent simulations by treating the single-agent simulation as a special case of the multi-agent, where there is only a single agent in the agents dictionary.

**property agents**

A dict that maps the Agent's id to the Agent object. An Agent must be an instance of PrincipleAgent. A multi-agent simulation is expected to have multiple entries in the dictionary, whereas a single-agent simulation should only have a single entry in the dictionary.

**finalize**()

Finalize the initialization process. At this point, every agent should be configured with action and observation spaces, which we convert into Dict spaces for interfacing with the trainer.

**abstract get_all_done**(***kwargs*)

Return the simulation's done status.

**abstract get_done**(*agent_id*, ***kwargs*)

Return the agent's done status.

**abstract get_info**(*agent_id*, ***kwargs*)

Return the agent's info.

**abstract get_obs**(*agent_id*, ***kwargs*)

Return the agent's observation.

**abstract get_reward**(*agent_id*, ***kwargs*)

Return the agent's reward.

**abstract render**(***kwargs*)

Render the simulation for vizualization.

**abstract reset**(***kwargs*)

Reset the simulation simulation to a start state, which may be randomly generated.

**abstract step**(*action*, ***kwargs*)

Step the simulation forward one discrete time-step. The action is a dictionary that contains the action of each agent in this time-step.

## 5.2 Abmarl Simulation Managers

**class** abmarl.managers.**SimulationManager**(*sim*)

Control interaction between Trainer and AgentBasedSimulation.

A Manager implmenents the reset and step API, by which it calls the AgentBasedSimulation API, using the getters within reset and step to accomplish the desired control flow.

**sim**

The AgentBasedSimulation.

**agents**

The agents that are in the AgentBasedSimulation.

**render**(***kwargs*)

**abstract reset**(***kwargs*)
Reset the simulation.

**Returns** The first observation of the agent(s).

**abstract step**(*action_dict*, ***kwargs*)
Step the simulation forward one discrete time-step.

**Parameters** **action_dict** – Dictionary mapping agent(s) to their actions in this time step.

**Returns**

The observations, rewards, done status, and info for the agent(s) whose actions we expect to receive next.

Note: We do not necessarily return anything for the agent whose actions we just received in this time-step. This behavior is defined by each Manager.

**class** abmarl.managers.**TurnBasedManager**(*sim*)
The TurnBasedManager allows agents to take turns. The order of the agents is stored and the obs of the first agent is returned at reset. Each step returns the info of the next agent "in line". Agents who are done are removed from this line. Once all the agents are done, the manager returns all done.

**reset**(***kwargs*)
Reset the simulation and return the observation of the first agent.

**step**(*action_dict*, ***kwargs*)
Assert that the incoming action does not come from an agent who is recorded as done. Step the simulation forward and return the observation, reward, done, and info of the next agent. If that next agent finished in this turn, then include the obs for the following agent, and so on until an agent is found that is not done. If all agents are done in this turn, then the wrapper returns all done.

**class** abmarl.managers.**AllStepManager**(*sim*)
The AllStepManager gets the observations of all agents at reset. At step, it gets the observations of all the agents that are not done. Once all the agents are done, the manager returns all done.

**reset**(***kwargs*)
Reset the simulation and return the observation of all the agents.

**step**(*action_dict*, ***kwargs*)
Assert that the incoming action does not come from an agent who is recorded as done. Step the simulation forward and return the observation, reward, done, and info of all the non-done agents, including the agents that were done in this step. If all agents are done in this turn, then the manager returns all done.

## 5.3 Abmarl External Integration

**class** abmarl.external.**GymWrapper**(*sim*)
Wrap an AgentBasedSimulation object with only a single agent to the gym.Env interface. This wrapper exposes the single agent's observation and action space directly in the simulation.

**render**(***kwargs*)
Forward render calls to the composed simulation.

**reset**(***kwargs*)
Return the observation from the single agent.

**step**(*action*, ***kwargs*)
Wrap the action by storing it in a dict that maps the agent's id to the action. Pass to sim.step. Return the observation, reward, done, and info from the single agent.

**property unwrapped**
Fall through all the wrappers and obtain the original, completely unwrapped simulation.

**class** abmarl.external.**MultiAgentWrapper**(*sim*)
Enable connection between SimulationManager and RLlib Trainer.

Wraps a SimulationManager and forwards all calls to the manager. This class is boilerplate and needed because RLlib checks that the simulation is an instance of MultiAgentEnv.

**sim**
The SimulationManager.

**render**(**args*, ***kwargs*)
See SimulationManager.

**reset**()
See SimulationManager.

**step**(*actions*)
See SimulationManager.

**property unwrapped**
Fall through all the wrappers to the SimulationManager.

**Returns** The wrapped SimulationManager.

# INDEX

## T



## U