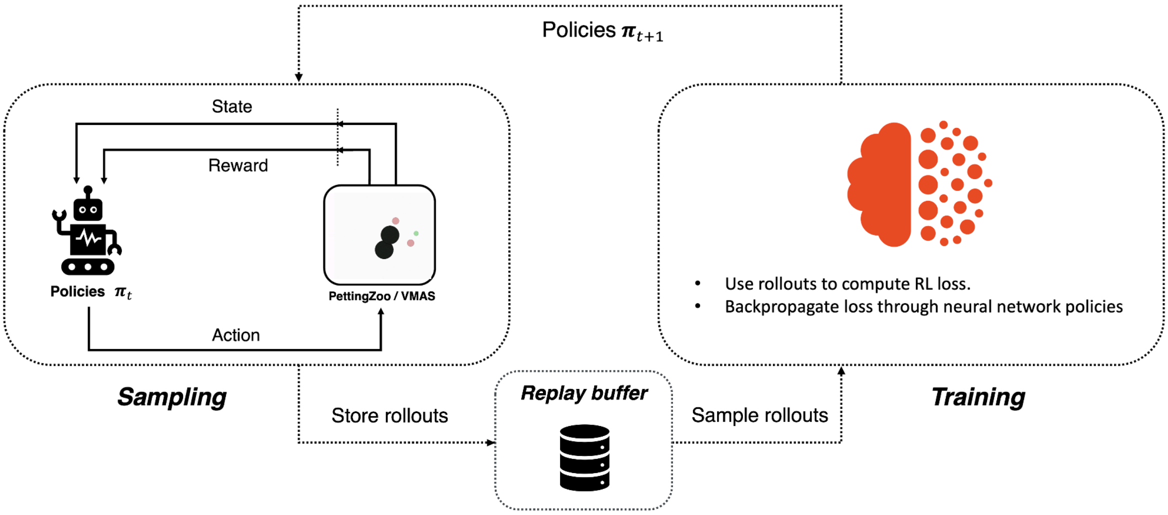
After the soft actor critic, another algorithm we will talk about is the deep deterministic policy gradient model, which is an off-policy actor-critic method.



It is used in environments with continuous state actions.
It can be taught with Deep Q- Learning with continuous state actions.

Key Equations:

1. Learning a Q function
2. Learning a policy

Learning a Q function:
Goal: Expected future reward for taking action in a given state.
Method: It uses a bellman's equation, a core principle in reinforcement learning to train the Q function.
Off-Policy Learning: It means it can learn from the previous experiences collected.

Learning a Policy Function:
Goal: Determines the action an agent should take in a given state to maximize the future reward.
Method: It employs an actor-network that represents the policy function. This network takes the state as input and outputs a continuous action.

Using the Q-function to improve the policy: These estimates of future rewards guide the policy update. According to the critic network, the actor-network is optimized to select actions with high Q-values.

The difference between DQN and DDPG

Feature	**DQN**	**DDPG**
Action Space	Discrete	Continuous
Q-function representation	Deep Neural Network	Deep Neural Network
Policy	Implicitly learned through Q-values	Explicitly learned through a separate policy network
Learning approach	On-policy	Off-policy

Before implementing these we need to install the following libraries
Pip install vmas
Pip install pettingzoo[mpe]==1.24.4
Pip install tqdm

First, we establish hyperparameters for use. Construct a multi-agent environment utilizing TorchRL's wrapper for either PettingZoo or VMAS. Following that, we formulate the policy and critic networks, discussing the effects of various choices on parameter sharing and critic centralization. Afterward, create the sampling collector and the replay buffer. In the end, we will execute our training loop and examine the outcomes.

Example program:

```
import copy
import tempfile

import torch
```

```
from matplotlib import pyplot as plt
from tensordict import TensorDictBase

from tensordict.nn import TensorDictModule, TensorDictSequential
from torch import multiprocessing

from torchrl.collectors import SyncDataCollector
from torchrl.data import LazyMemmapStorage, RandomSampler, ReplayBuffer

from torchrl.envs import (
    check_env_specs,
    ExplorationType,
    PettingZooEnv,
    RewardSum,
    set_exploration_type,
    TransformedEnv,
    VmasEnv,
)

from torchrl.modules import (
    AdditiveGaussianWrapper,
    MultiAgentMLP,
    ProbabilisticActor,
    TanhDelta,
)

from torchrl.objectives import DDPGLoss, SoftUpdate, ValueEstimators

from torchrl.record import CSVLogger, PixelRenderTransform,
VideoRecorder

from tqdm import tqdm

# Check if we're building the doc, in which case disable video
rendering
__sphinx_build__ = True
if __sphinx_build__:
    print("Sphinx build is defined.")
else:
    print("Sphinx build is not defined.")
try:
    is_sphinx = __sphinx_build__
```

```
except NameError:
    is_sphinx = False

# Seed
seed = 0
torch.manual_seed(seed)

# Devices
is_fork = multiprocessing.get_start_method() == "fork"
device = (
    torch.device(0)
    if torch.cuda.is_available() and not is_fork
    else torch.device("cpu")
)

# Sampling
frames_per_batch = 1_000  # Number of team frames collected per
sampling iteration
n_iters = 10  # Number of sampling and training iterations
total_frames = frames_per_batch * n_iters

# We will stop training the evaders after this many iterations,
# should be 0 <= iteration_when_stop_training_evaders <= n_iters
iteration_when_stop_training_evaders = n_iters // 2

# Replay buffer
memory_size = 1_000_000  # The replay buffer of each group can store
this many frames

# Training
n_optimiser_steps = 100  # Number of optimisation steps per training
iteration
train_batch_size = 128  # Number of frames trained in each optimiser
step
lr = 3e-4  # Learning rate
max_grad_norm = 1.0  # Maximum norm for the gradients

# DDPG
gamma = 0.99  # Discount factor
polyak_tau = 0.005  # Tau for the soft-update of the target network
max_steps = 100  # Environment steps before done

n_chasers = 2
```

```python
n_evaders = 1
n_obstacles = 2

use_vmas = True  # Set this to True for a great performance speedup

if not use_vmas:
    base_env = PettingZooEnv(
        task="simple_tag_v3",
        parallel=True,  # Use the Parallel version
        seed=seed,
        # Scenario specific
        continuous_actions=True,
        num_good=n_evaders,
        num_adversaries=n_chasers,
        num_obstacles=n_obstacles,
        max_cycles=max_steps,
    )
else:
    num_vmas_envs = (
        frames_per_batch // max_steps
    )  # Number of vectorized environments. frames_per_batch collection
will be divided among these environments
    base_env = VmasEnv(
        scenario="simple_tag",
        num_envs=num_vmas_envs,
        continuous_actions=True,
        max_steps=max_steps,
        device=device,
        seed=seed,
        # Scenario specific
        num_good_agents=n_evaders,
        num_adversaries=n_chasers,
        num_landmarks=n_obstacles,
    )

    print(f"group_map: {base_env.group_map}")
    print("action_spec:", base_env.full_action_spec)
    print("reward_spec:", base_env.full_reward_spec)
    print("done_spec:", base_env.full_done_spec)
    print("observation_spec:", base_env.observation_spec)
    print("action_keys:", base_env.action_keys)
    print("reward_keys:", base_env.reward_keys)
    print("done_keys:", base_env.done_keys)
```

```
env = TransformedEnv(
    base_env,
    RewardSum(
        in_keys=base_env.reward_keys,
        reset_keys=["_reset"] * len(base_env.group_map.keys()),
    ),
)
check_env_specs(env)
n_rollout_steps = 5
rollout = env.rollout(n_rollout_steps)
print(f"rollout of {n_rollout_steps} steps:", rollout)
print("Shape of the rollout TensorDict:", rollout.batch_size)

policy_modules = {}
for group, agents in env.group_map.items():
    share_parameters_policy = True  # Can change this based on the
group

    policy_net = MultiAgentMLP(
        n_agent_inputs=env.observation_spec[group,
"observation"].shape[
            -1
        ],  # n_obs_per_agent
        n_agent_outputs=env.full_action_spec[group, "action"].shape[
            -1
        ],  # n_actions_per_agents
        n_agents=len(agents),  # Number of agents in the group
        centralised=False,  # the policies are decentralised (i.e.,
each agent will act from its local observation)
        share_params=share_parameters_policy,
        device=device,
        depth=2,
        num_cells=256,
        activation_class=torch.nn.Tanh,
    )

    # Wrap the neural network in a
:class:`~tensordict.nn.TensorDictModule`.
    # This is simply a module that will read the ``in_keys`` from a
tensordict, feed them to the
    # neural networks, and write the
    # outputs in-place at the ``out_keys``.
```

```python
    policy_module = TensorDictModule(
        policy_net,
        in_keys=[(group, "observation")],
        out_keys=[(group, "param")],
    )  # We just name the input and output that the network will read and write to the input tensordict
    policy_modules[group] = policy_module

policies = {}
for group, _agents in env.group_map.items():
    policy = ProbabilisticActor(
        module=policy_modules[group],
        spec=env.full_action_spec[group, "action"],
        in_keys=[(group, "param")],
        out_keys=[(group, "action")],
        distribution_class=TanhDelta,
        distribution_kwargs={
            "min": env.full_action_spec[group, "action"].space.low,
            "max": env.full_action_spec[group, "action"].space.high,
        },
        return_log_prob=False,
    )
    policies[group] = policy

exploration_policies = {}
for group, _agents in env.group_map.items():
    exploration_policy = AdditiveGaussianWrapper(
        policies[group],
        annealing_num_steps=total_frames // 2,  # Number of frames after which sigma is sigma_end
        action_key=(group, "action"),
        sigma_init=0.9,  # Initial value of the sigma
        sigma_end=0.1,  # Final value of the sigma
    )
    exploration_policies[group] = exploration_policy

critics = {}
for group, agents in env.group_map.items():
    share_parameters_critic = True  # Can change for each group
    MADDPG = True  # IDDPG if False, can change for each group
```

```
    # This module applies the lambda function: reading the action and
observation entries for the group
    # and concatenating them in a new ``(group, "obs_action")`` entry
    cat_module = TensorDictModule(
        lambda obs, action: torch.cat([obs, action], dim=-1),
        in_keys=[(group, "observation"), (group, "action")],
        out_keys=[(group, "obs_action")],
    )

    critic_module = TensorDictModule(
        module=MultiAgentMLP(
            n_agent_inputs=env.observation_spec[group,
"observation"].shape[-1]
            + env.full_action_spec[group, "action"].shape[-1],
            n_agent_outputs=1,  # 1 value per agent
            n_agents=len(agents),
            centralised=MADDPG,
            share_params=share_parameters_critic,
            device=device,
            depth=2,
            num_cells=256,
            activation_class=torch.nn.Tanh,
        ),
        in_keys=[(group, "obs_action")],  # Read ``(group,
"obs_action")``
        out_keys=[
            (group, "state_action_value")
        ],  # Write ``(group, "state_action_value")``
    )

    critics[group] = TensorDictSequential(cat_module, critic_module)

reset_td = env.reset()
for group, _agents in env.group_map.items():
    print(
        f"Running value and policy for group '{group}':",
        critics[group](policies[group](reset_td)),
    )

# Put exploration policies from each group in a sequence
agents_exploration_policy =
TensorDictSequential(*exploration_policies.values())
```

```
collector = SyncDataCollector(
    env,
    agents_exploration_policy,
    device=device,
    frames_per_batch=frames_per_batch,
    total_frames=total_frames,
)

replay_buffers = {}
for group, _agents in env.group_map.items():
    replay_buffer = ReplayBuffer(
        storage=LazyMemmapStorage(
            memory_size, device=device
        ),  # We will store up to memory_size multi-agent transitions
        sampler=RandomSampler(),
        batch_size=train_batch_size,  # We will sample batches of this
size
    )
    replay_buffers[group] = replay_buffer

losses = {}
for group, _agents in env.group_map.items():
    loss_module = DDPGLoss(
        actor_network=policies[group],  # Use the non-explorative
policies
        value_network=critics[group],
        delay_value=True,  # Whether to use a target network for the
value
        loss_function="l2",
    )
    loss_module.set_keys(
        state_action_value=(group, "state_action_value"),
        reward=(group, "reward"),
        done=(group, "done"),
        terminated=(group, "terminated"),
    )
    loss_module.make_value_estimator(ValueEstimators.TD0, gamma=gamma)

    losses[group] = loss_module

target_updaters = {
    group: SoftUpdate(loss, tau=polyak_tau) for group, loss in
losses.items()
```

```
}

optimisers = {
    group: {
        "loss_actor": torch.optim.Adam(
            loss.actor_network_params.flatten_keys().values(), lr=lr
        ),
        "loss_value": torch.optim.Adam(
            loss.value_network_params.flatten_keys().values(), lr=lr
        ),
    }
    for group, loss in losses.items()
}

def process_batch(batch: TensorDictBase) -> TensorDictBase:
    """
    If the `(group, "terminated")` and `(group, "done")` keys are not
present, create them by expanding
    `"terminated"` and `"done"`.
    This is needed to present them with the same shape as the reward to
the loss.
    """
    for group in env.group_map.keys():
        keys = list(batch.keys(True, True))
        group_shape = batch.get_item_shape(group)
        nested_done_key = ("next", group, "done")
        nested_terminated_key = ("next", group, "terminated")
        if nested_done_key not in keys:
            batch.set(
                nested_done_key,
                batch.get(("next",
"done")).unsqueeze(-1).expand((*group_shape, 1)),
            )
        if nested_terminated_key not in keys:
            batch.set(
                nested_terminated_key,
                batch.get(("next", "terminated"))
                .unsqueeze(-1)
                .expand((*group_shape, 1)),
            )
    return batch

pbar = tqdm(
```

```
    total=n_iters,
    desc=", ".join(
        [f"episode_reward_mean_{group} = 0" for group in
env.group_map.keys()]
    ),
)
episode_reward_mean_map = {group: [] for group in env.group_map.keys()}
train_group_map = copy.deepcopy(env.group_map)

# Training/collection iterations
for iteration, batch in enumerate(collector):
    current_frames = batch.numel()
    batch = process_batch(batch)  # Util to expand done keys if needed

    # Loop over groups
    for group in train_group_map.keys():
        group_batch = batch.exclude(
            *[
                key
                for _group in env.group_map.keys()
                if _group != group
                for key in [_group, ("next", _group)]
            ]
        )  # Exclude data from other groups
        group_batch = group_batch.reshape(
            -1
        )  # This just affects the leading dimensions in batch_size of
the tensordict
        replay_buffers[group].extend(group_batch)

        for _ in range(n_optimiser_steps):
            subdata = replay_buffers[group].sample()
            loss_vals = losses[group](subdata)

            for loss_name in ["loss_actor", "loss_value"]:
                loss = loss_vals[loss_name]
                optimiser = optimisers[group][loss_name]

                loss.backward()

                # Optional
                params = optimiser.param_groups[0]["params"]
                torch.nn.utils.clip_grad_norm_(params, max_grad_norm)
```

```
                optimiser.step()
                optimiser.zero_grad()

            # Soft-update the target network
            target_updaters[group].step()

        # Exploration sigma anneal update
        exploration_policies[group].step(current_frames)

    # Stop training a certain group when a condition is met (e.g.,
number of training iterations)
    if iteration == iteration_when_stop_training_evaders:
        del train_group_map["agent"]

    # Logging
    for group in env.group_map.keys():
        episode_reward_mean = (
            batch.get(("next", group, "episode_reward"))[
                batch.get(("next", group, "done"))
            ]
            .mean()
            .item()
        )
        episode_reward_mean_map[group].append(episode_reward_mean)

    pbar.set_description(
        ", ".join(
            [
                f"episode_reward_mean_{group} =
{episode_reward_mean_map[group][-1]}"
                for group in env.group_map.keys()
            ]
        ),
        refresh=False,
    )
    pbar.update()

fig, axs = plt.subplots(2, 1)
for i, group in enumerate(env.group_map.keys()):
    axs[i].plot(episode_reward_mean_map[group], label=f"Episode reward
mean {group}")
    axs[i].set_ylabel("Reward")
```

```
    axs[i].axvline(
        x=iteration_when_stop_training_evaders,
        label="Agent (evader) stop training",
        color="orange",
    )
    axs[i].legend()
axs[-1].set_xlabel("Training iterations")
plt.show()

if use_vmas and not is_sphinx:
    # Replace tmpdir with any desired path where the video should be
saved
    with tempfile.TemporaryDirectory() as tmpdir:
        video_logger = CSVLogger("vmas_logs", tmpdir,
video_format="mp4")
        print("Creating rendering env")
        env_with_render = TransformedEnv(env.base_env,
env.transform.clone())
        env_with_render = env_with_render.append_transform(
            PixelRenderTransform(
                out_keys=["pixels"],
                # the np.ndarray has a negative stride and needs to be
copied before being cast to a tensor
                preproc=lambda x: x.copy(),
                as_non_tensor=True,
                # asking for array rather than on-screen rendering
                mode="rgb_array",
            )
        )
        env_with_render = env_with_render.append_transform(
            VideoRecorder(logger=video_logger, tag="vmas_rendered")
        )
        with set_exploration_type(ExplorationType.MODE):
            print("Rendering rollout...")
            env_with_render.rollout(100,
policy=agents_exploration_policy)
        print("Saving the video...")
        env_with_render.transform.dump()
        print("Saved! Saved directory tree:")
        video_logger.print_log_dir()
```

Output:

```
PS F:\my_python_files> python ddpg.py
Sphinx build is defined.
group_map: {'adversary': ['adversary_0', 'adversary_1'], 'agent':
['agent_0']}
action_spec: CompositeSpec(
    adversary: CompositeSpec(
        action: BoundedTensorSpec(
            shape=torch.Size([10, 2, 2]),
            space=ContinuousBox(
                low=Tensor(shape=torch.Size([10, 2, 2]), device=cpu,
dtype=torch.float32, contiguous=True),
                high=Tensor(shape=torch.Size([10, 2, 2]), device=cpu,
dtype=torch.float32, contiguous=True)),
            device=cpu,
            dtype=torch.float32,
            domain=continuous), device=cpu, shape=torch.Size([10,
2])),
    agent: CompositeSpec(
        action: BoundedTensorSpec(
            shape=torch.Size([10, 1, 2]),
            space=ContinuousBox(
                low=Tensor(shape=torch.Size([10, 1, 2]), device=cpu,
dtype=torch.float32, contiguous=True),
                high=Tensor(shape=torch.Size([10, 1, 2]), device=cpu,
dtype=torch.float32, contiguous=True)),
            device=cpu,
            dtype=torch.float32,
            domain=continuous), device=cpu, shape=torch.Size([10,
1])), device=cpu, shape=torch.Size([10]))
reward_spec: CompositeSpec(
    adversary: CompositeSpec(
        reward: UnboundedContinuousTensorSpec(
            shape=torch.Size([10, 2, 1]),
            space=None,
```

```
            device=cpu,
            dtype=torch.float32,
            domain=continuous), device=cpu, shape=torch.Size([10,
2])),
    agent: CompositeSpec(
        reward: UnboundedContinuousTensorSpec(
            shape=torch.Size([10, 1, 1]),
            space=None,
            device=cpu,
            dtype=torch.float32,
            domain=continuous), device=cpu, shape=torch.Size([10,
1])), device=cpu, shape=torch.Size([10]))
done_spec: CompositeSpec(
    done: DiscreteTensorSpec(
        shape=torch.Size([10, 1]),
        space=DiscreteBox(n=2),
        device=cpu,
        dtype=torch.bool,
        domain=discrete),
    terminated: DiscreteTensorSpec(
        shape=torch.Size([10, 1]),
        space=DiscreteBox(n=2),
        device=cpu,
        dtype=torch.bool,
        domain=discrete), device=cpu, shape=torch.Size([10]))
observation_spec: CompositeSpec(
    adversary: CompositeSpec(
        observation: UnboundedContinuousTensorSpec(
            shape=torch.Size([10, 2, 14]),
            space=None,
            device=cpu,
            dtype=torch.float32,
            domain=continuous), device=cpu, shape=torch.Size([10,
2])),
    agent: CompositeSpec(
```

```
        observation: UnboundedContinuousTensorSpec(
            shape=torch.Size([10, 1, 12]),
            space=None,
            device=cpu,
            dtype=torch.float32,
            domain=continuous), device=cpu, shape=torch.Size([10,
1])), device=cpu, shape=torch.Size([10]))
action_keys: [('adversary', 'action'), ('agent', 'action')]
reward_keys: [('adversary', 'reward'), ('agent', 'reward')]
done_keys: ['done', 'terminated']
2024-06-25 11:38:03,116 [torchrl][INFO] check_env_specs
succeeded!
rollout of 5 steps: TensorDict(
    fields={
        adversary: TensorDict(
            fields={
                action: Tensor(shape=torch.Size([10, 5, 2, 2]),
device=cpu, dtype=torch.float32, is_shared=False),
                episode_reward: Tensor(shape=torch.Size([10, 5, 2, 1]),
device=cpu, dtype=torch.float32, is_shared=False),
                observation: Tensor(shape=torch.Size([10, 5, 2, 14]),
device=cpu, dtype=torch.float32, is_shared=False)},
            batch_size=torch.Size([10, 5, 2]),
            device=cpu,
            is_shared=False),
        agent: TensorDict(
            fields={
                action: Tensor(shape=torch.Size([10, 5, 1, 2]),
device=cpu, dtype=torch.float32, is_shared=False),
                episode_reward: Tensor(shape=torch.Size([10, 5, 1, 1]),
device=cpu, dtype=torch.float32, is_shared=False),
                observation: Tensor(shape=torch.Size([10, 5, 1, 12]),
device=cpu, dtype=torch.float32, is_shared=False)},
            batch_size=torch.Size([10, 5, 1]),
            device=cpu,
```

```
        is_shared=False),
      done: Tensor(shape=torch.Size([10, 5, 1]), device=cpu,
dtype=torch.bool, is_shared=False),
      next: TensorDict(
        fields={
          adversary: TensorDict(
            fields={
              episode_reward: Tensor(shape=torch.Size([10, 5, 2,
1]), device=cpu, dtype=torch.float32, is_shared=False),
              observation: Tensor(shape=torch.Size([10, 5, 2, 14]),
device=cpu, dtype=torch.float32, is_shared=False),
              reward: Tensor(shape=torch.Size([10, 5, 2, 1]),
device=cpu, dtype=torch.float32, is_shared=False)},
            batch_size=torch.Size([10, 5, 2]),
            device=cpu,
            is_shared=False),
          agent: TensorDict(
            fields={
              episode_reward: Tensor(shape=torch.Size([10, 5, 1,
1]), device=cpu, dtype=torch.float32, is_shared=False),
              observation: Tensor(shape=torch.Size([10, 5, 1, 12]),
device=cpu, dtype=torch.float32, is_shared=False),
              reward: Tensor(shape=torch.Size([10, 5, 1, 1]),
device=cpu, dtype=torch.float32, is_shared=False)},
            batch_size=torch.Size([10, 5, 1]),
            device=cpu,
            is_shared=False),
          done: Tensor(shape=torch.Size([10, 5, 1]), device=cpu,
dtype=torch.bool, is_shared=False),
          terminated: Tensor(shape=torch.Size([10, 5, 1]),
device=cpu, dtype=torch.bool, is_shared=False)},
        batch_size=torch.Size([10, 5]),
        device=cpu,
        is_shared=False),
```

```
        terminated: Tensor(shape=torch.Size([10, 5, 1]), device=cpu,
dtype=torch.bool, is_shared=False)},
    batch_size=torch.Size([10, 5]),
    device=cpu,
    is_shared=False)
Shape of the rollout TensorDict: torch.Size([10, 5])
Running value and policy for group 'adversary': TensorDict(
    fields={
        adversary: TensorDict(
            fields={
                action: Tensor(shape=torch.Size([10, 2, 2]), device=cpu,
dtype=torch.float32, is_shared=False),
                episode_reward: Tensor(shape=torch.Size([10, 2, 1]),
device=cpu, dtype=torch.float32, is_shared=False),
                obs_action: Tensor(shape=torch.Size([10, 2, 16]),
device=cpu, dtype=torch.float32, is_shared=False),
                observation: Tensor(shape=torch.Size([10, 2, 14]),
device=cpu, dtype=torch.float32, is_shared=False),
                param: Tensor(shape=torch.Size([10, 2, 2]), device=cpu,
dtype=torch.float32, is_shared=False),
                state_action_value: Tensor(shape=torch.Size([10, 2, 1]),
device=cpu, dtype=torch.float32, is_shared=False)},
            batch_size=torch.Size([10, 2]),
            device=cpu,
            is_shared=False),
        agent: TensorDict(
            fields={
                episode_reward: Tensor(shape=torch.Size([10, 1, 1]),
device=cpu, dtype=torch.float32, is_shared=False),
                observation: Tensor(shape=torch.Size([10, 1, 12]),
device=cpu, dtype=torch.float32, is_shared=False)},
            batch_size=torch.Size([10, 1]),
            device=cpu,
            is_shared=False),
```

```
        done: Tensor(shape=torch.Size([10, 1]), device=cpu,
dtype=torch.bool, is_shared=False),
        terminated: Tensor(shape=torch.Size([10, 1]), device=cpu,
dtype=torch.bool, is_shared=False)},
    batch_size=torch.Size([10]),
    device=cpu,
    is_shared=False)
Running value and policy for group 'agent': TensorDict(
    fields={
        adversary: TensorDict(
            fields={
                action: Tensor(shape=torch.Size([10, 2, 2]), device=cpu,
dtype=torch.float32, is_shared=False),
                episode_reward: Tensor(shape=torch.Size([10, 2, 1]),
device=cpu, dtype=torch.float32, is_shared=False),
                obs_action: Tensor(shape=torch.Size([10, 2, 16]),
device=cpu, dtype=torch.float32, is_shared=False),
                observation: Tensor(shape=torch.Size([10, 2, 14]),
device=cpu, dtype=torch.float32, is_shared=False),
                param: Tensor(shape=torch.Size([10, 2, 2]), device=cpu,
dtype=torch.float32, is_shared=False),
                state_action_value: Tensor(shape=torch.Size([10, 2, 1]),
device=cpu, dtype=torch.float32, is_shared=False)},
            batch_size=torch.Size([10, 2]),
            device=cpu,
            is_shared=False),
        agent: TensorDict(
            fields={
                action: Tensor(shape=torch.Size([10, 1, 2]), device=cpu,
dtype=torch.float32, is_shared=False),
                episode_reward: Tensor(shape=torch.Size([10, 1, 1]),
device=cpu, dtype=torch.float32, is_shared=False),
                obs_action: Tensor(shape=torch.Size([10, 1, 14]),
device=cpu, dtype=torch.float32, is_shared=False),
```

            observation: Tensor(shape=torch.Size([10, 1, 12]), device=cpu, dtype=torch.float32, is_shared=False),
            param: Tensor(shape=torch.Size([10, 1, 2]), device=cpu, dtype=torch.float32, is_shared=False),
            state_action_value: Tensor(shape=torch.Size([10, 1, 1]), device=cpu, dtype=torch.float32, is_shared=False)},
         batch_size=torch.Size([10, 1]),
         device=cpu,
         is_shared=False),
      done: Tensor(shape=torch.Size([10, 1]), device=cpu, dtype=torch.bool, is_shared=False),
      terminated: Tensor(shape=torch.Size([10, 1]), device=cpu, dtype=torch.bool, is_shared=False)},
   batch_size=torch.Size([10]),
   device=cpu,
   is_shared=False)
episode_reward_mean_adversary = 1.0,
episode_reward_mean_agent = -1.0:
100%|██████████|
10/10 [03:42<00:00, 18.79s/it]

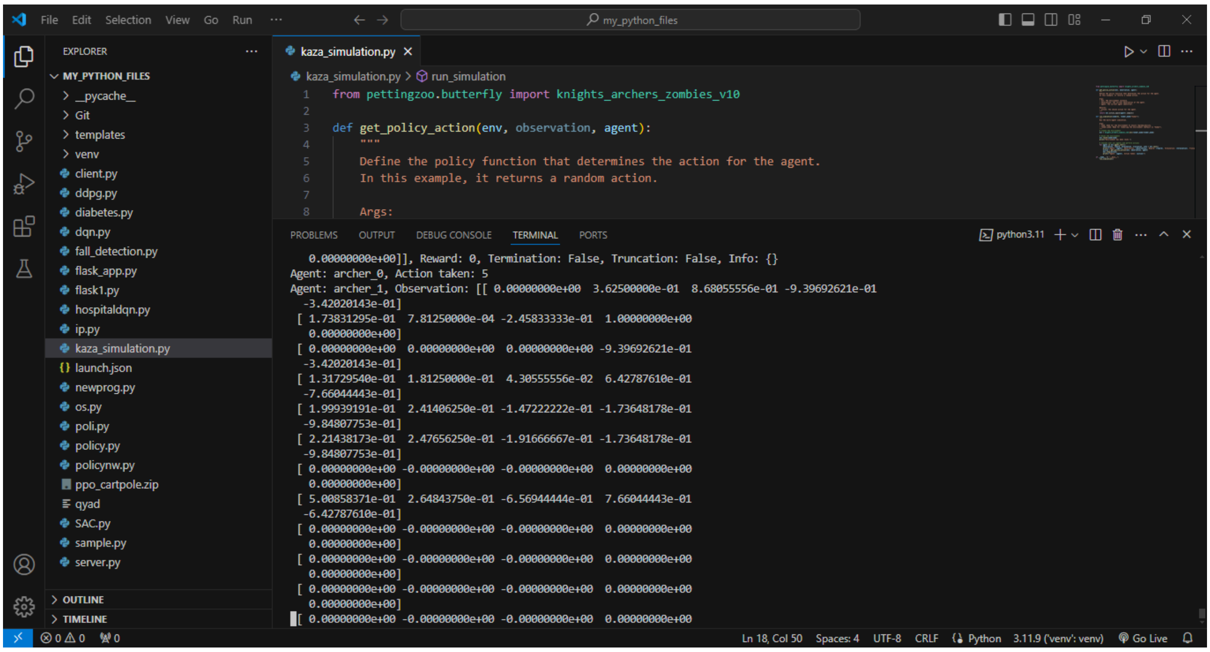
PettingZoo:
PettingZoo is a simple, pythonic interface capable of representing general multi-agent reinforcement learning (MARL) problems. PettingZoo includes a wide variety of reference environments, helpful utilities, and tools for creating your own custom environments.

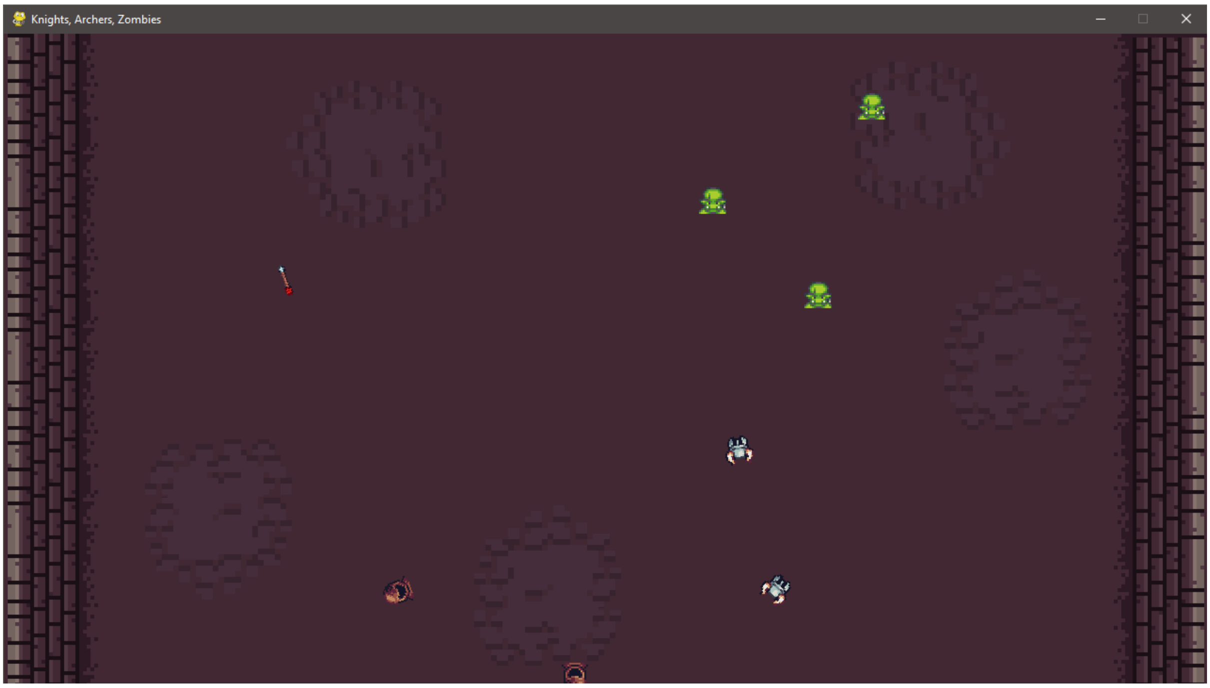
Example program kaza_simulation.py:

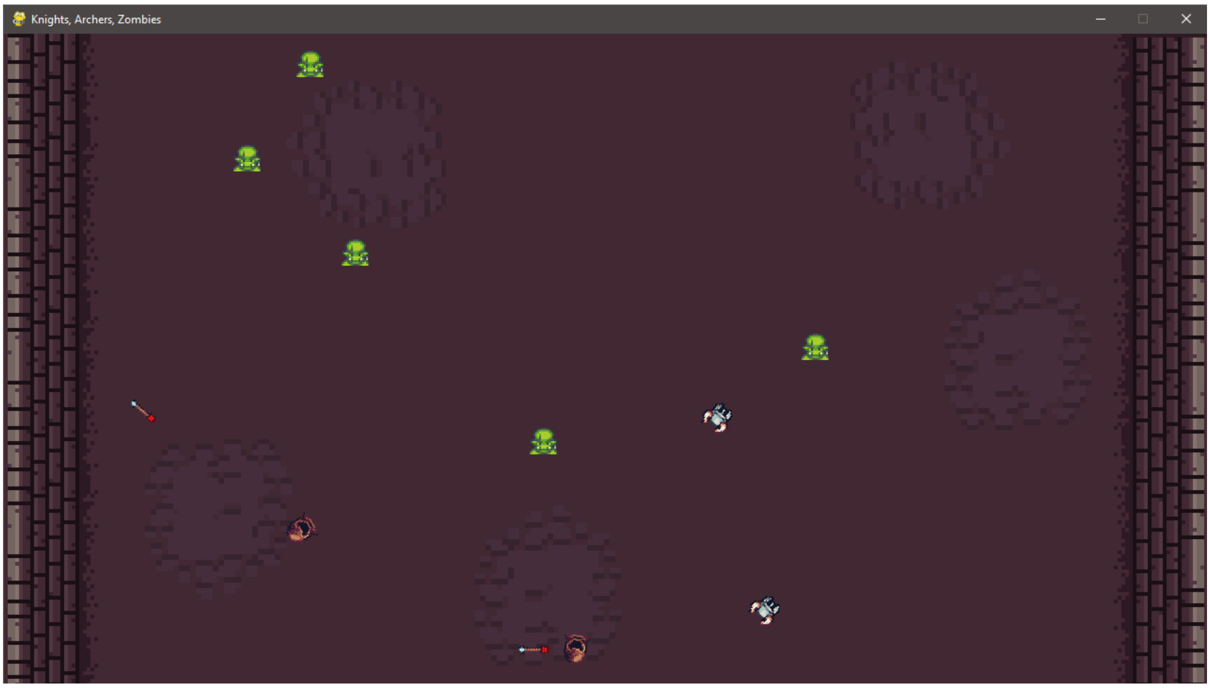
```
from pettingzoo.butterfly import knights_archers_zombies_v10

def get_policy_action(env, observation, agent):
    """
    Define the policy function that determines the action for the agent.
```

```
    In this example, it returns a random action.

    Args:
    - env: The environment instance.
    - observation: The current observation of the agent.
    - agent: The current agent identifier.

    Returns:
    - action: The chosen action for the agent.
    """
    return env.action_space(agent).sample()

def run_simulation(seed=42, render_mode="human"):
    """
    Run the multi-agent simulation.

    Args:
    - seed: Seed for the environment to ensure reproducibility.
    - render_mode: Mode for rendering the environment (default is
"human").
    """
    # Create the environment
    env = knights_archers_zombies_v10.env(render_mode=render_mode)

    # Reset the environment
    env.reset(seed=seed)
    print("Environment has been reset.")

    # Iterate over all agents and perform actions
    for agent in env.agent_iter():
        observation, reward, termination, truncation, info = env.last()
        print(f"Agent: {agent}, Observation: {observation}, Reward:
{reward}, Termination: {termination}, Truncation: {truncation}, Info:
{info}")
        action = get_policy_action(env, observation, agent)
        env.step(action)
        print(f"Agent: {agent}, Action taken: {action}")

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

Here, we need to install library of pettingzoo
Install pip pettingzoo







Here, in this game when the agent is dead, the only valid action is None.