Reinforcement Learning Course Final Project - VMAS Related

21311475 Xie Wenlong

1 Abstract

In this project, the author read the source code of VMAS and compiled and explained it; VMAS is a simulation environment for Multi-Agent Reinforcement Learning (MARL), its source code structure is clear and easy to understand. Subsequently, the author implemented multiple PPO algorithms under the multi-agent situation, including CPPO, MAPPO, IPPO and HetIPPO.

The project code is available at https://github.com/ZojkLoyn/MARL_2024_Autumn, which contains two branches: `my_job` and `torchrl`, respectively, the code implemented by the author using the `VMAS` library and the code implemented using the `torchrl` library.

2 VMAS Source Code Reading and Compilation

VMAS, which stands for Vectorized Multi Agent Simulator, is a simulation environment for Multi-Agent Reinforcement Learning (MARL). This report will introduce the source code structure of VMAS and provide detailed explanations of its key components.

2.1 Core Concepts

- **Environment:** The concept of an environment comes from the OpenAI Gym library, representing a Markov decision process. One environment contains `n_agents` agents, `num_envs` worlds, pre-set scenarios, observation spaces, rewards, termination conditions, and action spaces, etc.
- **World:** In a VMAS environment, each world represents a MARL environment. Each world contains `n_agents` agents, each agent has its own observation space, action space, reward function, etc. This design is to fully utilize the parallel computing capabilities of the GPU, each world runs independently and does not interfere with each other.
- **Agent:** Agent is the subject in the MARL environment. Each agent has its own state, observation space, action space, reward function, etc. It changes the state and affects the environment through actions, and obtains rewards from the environment. Each world has `n_agents` agents, which can also be understood as each agent has a copy in each world.
- **Action:** Action is the operation that the agent performs in the environment. Each action changes the state of the agent and affects the environment.
- **State:** State is the state in the environment. Each state contains the position, velocity, direction, etc. of the agent.
- **Observation:** Observation is the perception of the agent to the environment, which is to obtain the state information of one or more agents.
- **Reward:** Reward is the reward obtained by the agent from the environment. According to the agent's actions and the state of the environment, the agent can obtain different rewards.

2.2 File Structure

The `simulator` directory is the main code directory of the simulator, and the `core` file is the core of the code.

The `core` file contains the following classes:

2.2.1 Class `TorchVectorizeObject`

`TorchVectorizeObject` is the base class of all vectorizable objects in VMAS. Its `batch_dim` attribute represents the batch size, which is usually the number of worlds in a VMAS environment `num_envs`.

It contains the `to` method, which is used to transfer the object to the specified device.

2.2.2 Class `Shape`

The `Shape` class is used to represent the shape of an object.

There are subclasses `Box`, `Sphere`, and `Line`, which represent boxes, spheres, and lines, respectively.

2.2.3 Class `EntityState`

It is used to represent the state of an entity, with four attributes `pos`, `vel`, `ang_vel`, and `rot`, representing position, linear velocity, angular velocity, and rotation matrix, respectively.

The subclass `AgentState` specifically represents the state of an agent, with four attributes `c`, `force`, `torque`, used to represent communication language, force, and torque, respectively.

2.2.4 Class `Action`

It is used to represent the action of a single `Agent`, with two main attributes `u` and `c`, representing physical actions and communication actions, respectively.

2.2.5 Class `Entity`

It is used to represent an entity, with physical properties such as mass, and properties such as collision and mobility. In addition, it also has a `state` attribute representing position-related properties, with functions to set `state` related information.

There are two subclasses:

- `Landmark` represents a preset landmark, which is identical to `Entity`. It is mainly used to distinguish between `Agent` and `Landmark`.
- `Agent` represents an agent, with attributes describing the agent's ability to observe and feel the world, with constraints on force (which will be processed in `step` of the world), and dynamics-related and action-related information.

2.2.6 Other Files

- `Joint` file contains `Joint` joint-related content, representing how entities change their state according to actions.
- The `Dynamics` folder contains entity dynamics-related content, representing how entities change their force state according to actions.
- The `Environment` folder contains environment-related content, representing an environment, containing a scenario, functions for obtaining information, a `step` function for inputting actions and executing an environment update, and some rendering functions.
- The `Scenario` file contains the base class of the scenario, specifying that the scenario should have a world, with functions for creating a world, resetting the world in a specified environment, the agent's observation of the environment, the received reward, and processing actions. Specific preset scenarios are contained in the `Scenarios` folder.

2.3 Usage of Environment

VMAS's environment is similar to OpenAI Gym's environment and can be used to train agents. You only need to master a few basic concepts to use it, as shown below.

2.3.1 Initialize Environment

use `make_env` function to create an environment, with the following important parameters

- `scenario:` Scenario name, which can specify which preset scenario to use.
- `num_envs:` Number of worlds, i.e., number of parallel environments.
- `device:` Device, used to specify whether to use GPU or CPU.
- World-related information
  - `max_steps:` The maximum number of steps each world can execute.
  - `continuous_actions:` Whether to automatically reset after the world ends.
- Additional parameters of the scenario
  - `n_agents:` Number of agents. In some preset scenarios, the number of agents is fixed and cannot be specified.

2.3.2 Observation Space and Action Space

VMAS's environment has observation space and action space, respectively representing the agent's observation of the environment and the actions that the agent can execute.

For scenarios where all agents are equal, the observation space and action space of each agent are the same, so you only need to get the space of one agent. You can use `env.observation_space[0].shape[0]` to get the number of tensor dimensions of the observation space of a single agent in a world `observation_dim`; you can use `env.action_space[0].shape[0]` to get the number of tensor dimensions of the action space of a single agent in a world `action_dim`.

2.3.3 Reset and step

VMAS's environment supports reset and step, respectively used to initialize the environment and execute the agent's actions.

use `env.reset()` to reset the environment, returning an observation tensor `state`.

use `env.step(actions)` to input the action tensor `action` and iterate the environment, returning an observation tensor `next_state`, a reward tensor `reward`, a termination tensor `done`, and an information tensor `info`.

- action tensor `action` is used to represent the agent's actions in the world, with the format of agent index, world index, and action tensor, i.e. $\text{n\_agents}_{\text{tuple}} \times [\text{num\_envs}, \text{action\_dim}]_{\text{tensor}}$
- observation tensor `state` and `next_state` represent the agent's observation in the world, with the same format of agent index, world index, and observation tensor, i.e.

$\text{n\_agents}_{\text{tuple}} \times [\text{num\_envs}, \text{observation\_dim}]_{\text{tensor}}$

- reward tensor `reward` represents the agent's harvest in the world in this iteration, with the format of agent index, world index, i.e.

$$\text{n\_agents}_{\text{tuple}} \times [\text{num\_envs}]_{\text{tensor}}$$

- termination tensor `done` represents whether the world ends after the iteration, with the element type of `bool` and the format of world index

$$[\text{num\_envs}]_{\text{tensor}}$$

- information tensor `info` is used to record the specific information of the reward tensor, with different formats for different preset scenarios, such as the `Balance` scenario, which is agent index, `pos` reward and `ground` reward, i.e.

$$\text{n\_agents}_{\text{tuple}} \times \begin{cases} \text{"pos\_rew"} : [\text{num\_envs}]_{\text{tensor}} \\ \text{"ground\_rew"} : [\text{num\_envs}]_{\text{tensor}} \end{cases}$$

3 Proximal Policy Optimization

PPO stands for Proximal Policy Optimization, which is an agent training idea. By iteratively executing the agent's actions and collecting the agent's experiences, and then using these experiences to update the agent's policy.

3.1 Single Agent PPO

PPO algorithm is based on the Actor-Critic framework, where the Actor is the policy network and the Critic is the value network. The policy network is used to generate the agent's actions, and the value network is used to evalu-

ate the good or bad of the agent's actions. The PPO algorithm iteratively executes the agent's actions and collects the agent's experiences, and then uses these experiences to update the agent's policy.

The input of the Actor is the agent's observation, and the output is the agent's action. As shown in Figure 1.

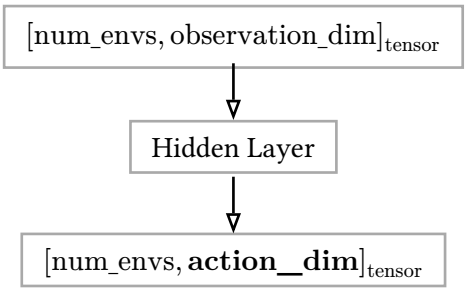

Figure 1: The structure of the Actor network in single agent

The input of the Critic is the agent's observation and action, and the output is the agent's action value. As shown in Figure 2.

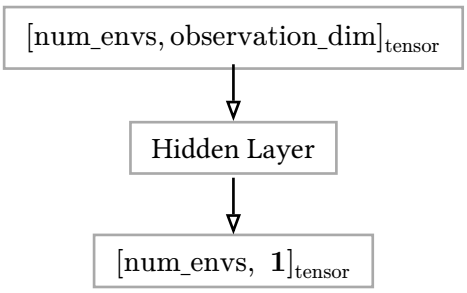

Figure 2: The structure of the Critic network in single agent

The algorithm's process is as follows:

1. Initialize the Actor and Critic networks and set the learning rate.
2. In the environment, the Actor decides the agent's action and collects the agent's experience. The experience includes the agent's observation, action, reward, and next state, that is, $\left(s_t, a_t, r_t, s_{\{t+1\}}\right)$.
3. Calculate the advantage function $A_t = r_t + \gamma V\left(s_{\{t+1\}}\right) - V(s_t)$ through the experience, where γ is the hyperparameter discount factor, and $V(s)$ is the value function that the Critic network needs to learn.
4. Train the Critic network through the advantage function, so that it can better estimate the action value.
5. Use the special objective function $r_{t(\theta)} = \frac{\pi_{\theta(a_t|s_t)}}{\pi_{\theta,\text{old}}(a_t|s_t)}$ combined with the clip loss function to train the Actor network, so that it can better estimate the action probability.
6. Repeat steps 3-5 until the predetermined training steps or convergence conditions are reached.
7. Repeat steps 2-6 until the predetermined training rounds or convergence conditions are reached.

3.2 Multiple Agent PPO

In the multi-agent case, the Actor network and Critic network will be more complex, responsible for handling tasks for multiple agents.

3.2.1 Multi-Agent Network Structure

For multi-agents, we will define the overall-network as a network whose input is the state of all agents and whose output is the information of all agents.

Its network structure is Figure 3.

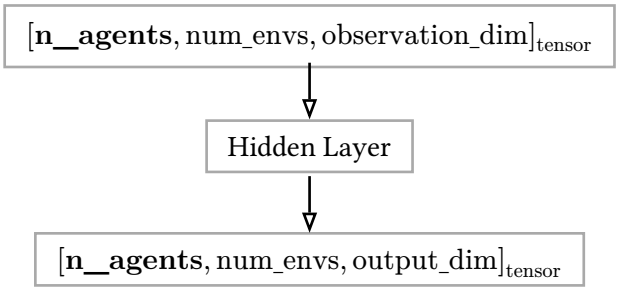

Figure 3: The structure of the overall-network in multiple agent

And the hidden layer is divided into three types according to the specific design of the distribution.

3.2.1.1 Centralized

In the centralized mode, the hidden layer can simultaneously obtain and process the state information of all agents and make decisions for all agents.

The num_env is regarded as the batch attribute, and the n_agents $*$ observation_dim is regarded as the feature attribute.

Its network structure is Figure 4.

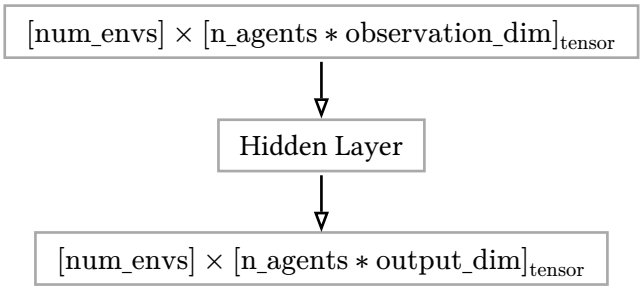

Figure 4: The structure of the centralized network

3.2.1.2 Shared Parameters

In the shared parameters, the hidden layer can only obtain and process the state information of a single agent and make decisions for a single agent. However, its parameters are shared by all agents.

The num_env $*$ n_agents is regarded as the batch attribute, and the observation_dim is regarded as the feature attribute.

Its network structure is Figure 5.

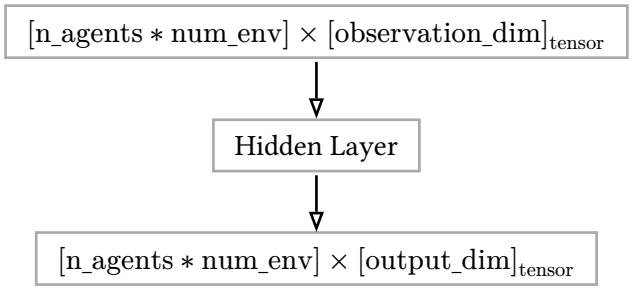

Figure 5: The structure of the shared parameters network

3.2.1.3 Independent Parameters

In the independent parameters, the hidden layer can only obtain and process the state information of a single agent and make decisions for a single agent. And each agent occupies a network independently.

The num_env is regarded as the batch attribute, and the observation_dim is regarded as the feature attribute.

Its network structure is Figure 6.

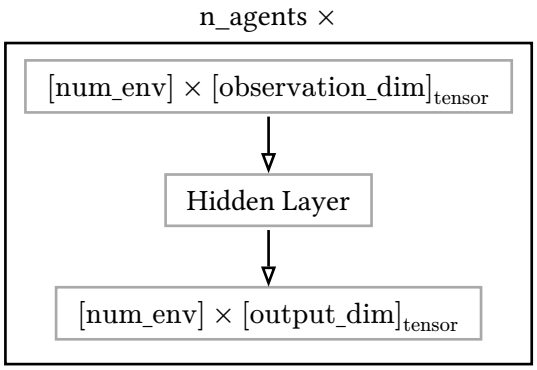

Figure 6: The structure of the independent parameters network

3.2.2 Multi-agent PPO Algorithm

The core idea of various multi-agent PPO algorithms is the same as that of the single-agent PPO algorithm. The difference lies in the design of the Actor network and the Critic network.

The following are the definitions of four multi-agent PPO algorithms mentioned in the VMAS paper, which depend on the design of the Actor network and the Critic network.

3.2.2.1 CPPO

- Actor: Centralized [Section 3.2.1.1]
- Critic: Centralized [Section 3.2.1.1]

3.2.2.2 MAPPO

- Actor: Independent [Section 3.2.1.3]
- Critic: Centralized [Section 3.2.1.1]

3.2.2.3 IPPO

- Actor: Shared Parameters [Section 3.2.1.2]
- Critic: Shared Parameters [Section 3.2.1.2]

3.2.2.4 HetIPPO

- Actor: Independent [Section 3.2.1.3]

- Critic: Independent [Section 3.2.1.3]

4 Project Implementation

In the project implementation, I mainly implemented the CPPO, MAPPO, IPPO, and HetIPPO algorithms for the Balance scenario.

The implementation code is located at https://github.com/ZojkLoyn/MARL_2024_Autumn. Among them, the `torchrl` branch uses the `TorchRL` library to implement, which can train a good result; and the `my_job` branch is implemented directly using the `VMAS` library, but there are unresolved bugs.

4.1 Design Details

There is a design detail that VMAS requires the output of the Actor network to be limited within the range of $[-1, 1]$; while the last layer of the `torchrl.MultiAgentMLP` network is a Linear Layer, which may output illegal values; therefore, an appropriate activation layer should be added after the network output layer. However, the effect of different activation layers is different.

- `ReLU` activation functions, whose value range is inconsistent with $[-1, 1]$; therefore, illegal values will still be produced.
- `Softmax` and other vector activation functions, which will affect the output of other actions; therefore, they are not applicable.
- `Tanh` and some activation functions, whose value range is $[-1, 1]$, but their output value distribution is uneven, and they cannot reach the boundary; therefore, they cannot produce boundary policies, and problems may easily occur.
- `HardTanh` , whose value range is $[-1, 1]$, and their output value distribution is uniform, and they can easily reach the boundary; therefore, they can produce boundary policies, and the effect is good.

4.2 Branch `my_job`

In the `my_job` branch, I use the `VMAS` library to simulate the training environment and use the `PyTorch` library to implement the Actor network and Critic network.

However, there is a bug in it, and the training has no effect, so it is only used as a proof of work here.

4.3 Branch `torchrl`

In the `torchrl` branch, I implemented the training environment simulation, neural network implementation, and training process using the `TorchRL` library.

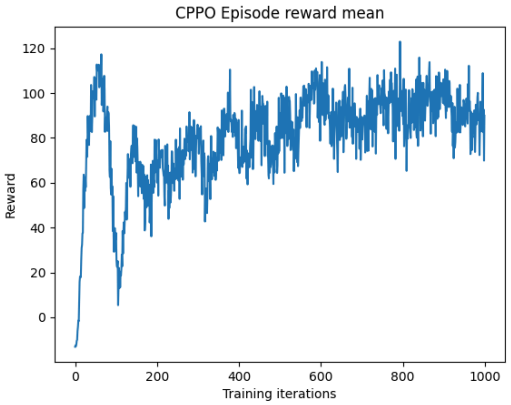

Figure 7: CPPO Training Results

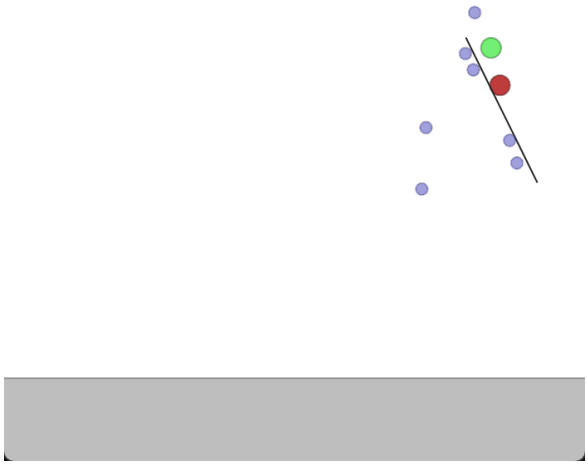

Figure 8: It still has room for improvement

It can train a good result within 1000 training iters. The training results are shown in Figure 7. However, it still has some problems, such as the deviation of the transport path and the target location, see Figure 8.

5 Conclusion

In this experiment, I implemented multiple MARL algorithms and verified them through experiments. The experimental results show that the PPO algorithm performs well in multi-agent tasks and can train a good result. However, my implementation still has some problems, such as the deviation of the transport path and the target location, which need further optimization.