MARLlib

Release v0.1.0

ICLR2023

Apr 25, 2023

MARLLIB HANDBOOK

CHAPTER

ONE

INTRODUCTION

Multi-Agent Reinforcement Learning Library (MARLlib) is **a comprehensive Multi-Agent Reinforcement Learning algorithm library** based on Ray and one of its toolkits RLlib. It provides MARL research community with a unified platform for building, training, and evaluating MARL algorithms.

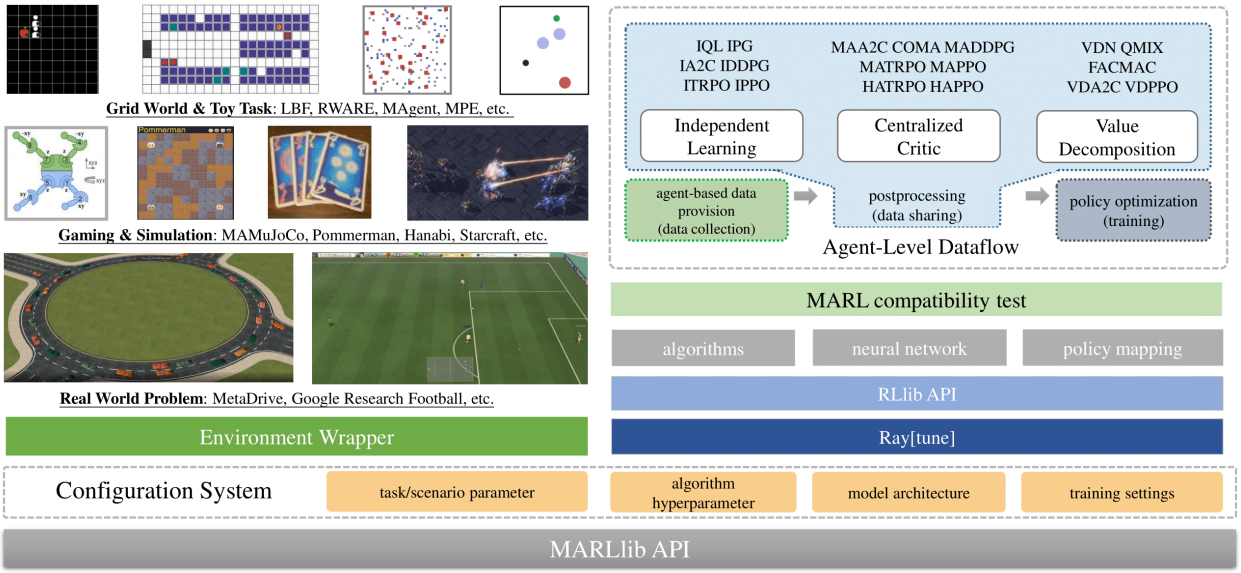


Fig. 1: Overview of the MARLlib architecture.

MARLlib offers several key features that make it stand out:

- MARLlib unifies diverse algorithm pipelines with agent-level distributed dataflow, allowing researchers to develop, test, and evaluate MARL algorithms across different tasks and environments.
- MARLlib supports all task modes, including cooperative, collaborative, competitive, and mixed. This makes it easier for researchers to train and evaluate MARL algorithms across a wide range of tasks.
- MARLlib provides a new interface that follows the structure of Gym, making it easier for researchers to work with multi-agent environments.
- MARLlib provides flexible and customizable parameter-sharing strategies, allowing researchers to optimize their algorithms for different tasks and environments.

Using MARLlib, you can take advantage of various benefits, such as:

- **Zero knowledge of MARL**: MARLlib provides 18 pre-built algorithms with an intuitive API, allowing researchers to start experimenting with MARL without prior knowledge of the field.

- **Support for all task modes**: MARLlib supports almost all multi-agent environments, making it easier for researchers to experiment with different task modes.
- **Customizable model architecture**: Researchers can choose their preferred model architecture from the model zoo, or build their own.
- **Customizable policy sharing**: MARLlib provides grouping options for policy sharing, or researchers can create their own.
- **Access to over a thousand released experiments**: Researchers can access over a thousand released experiments to see how other researchers have used MARLlib.

Before starting, please ensure you've installed the dependencies by following the *Installation*. The environment-specific description is maintained in *Environments*. *Quick Start* gives some basic examples.

CHAPTER

TWO

INSTALLATION

The installation of MARLlib has two parts: common installation and external environment installation. We've tested the installation on Python 3.8 with Ubuntu 18.04 and Ubuntu 20.04.

2.1 MARLlib Installation

We strongly recommend using conda to manage your dependencies and avoid version conflicts. Here we show the example of building python 3.8 based conda environment.

```
conda create -n marllib python=3.8
conda activate marllib
git clone https://github.com/Replicable-MARL/MARLlib.git
cd MARLlib
pip install --upgrade pip
pip install -r requirements.txt

# add patch files to MARLlib
python patch/add_patch.py -y
```

2.2 External Environments Requirements

External environments are not auto-integrated (except MPE). However, you can install them by following.

- our simplified guides.
- the official guide of each environment.

CHAPTER

THREE

FRAMEWORK

MARLlib is a software library designed to facilitate the development and evaluation of multi-agent reinforcement learning (MARL) algorithms. The library is built on top of Ray, a distributed computing framework, and RLlib, one of its toolkits. Specifically, MARLlib extends RLlib by incorporating 18 MARL algorithms and 10 multi-agent environments, providing a comprehensive testing bed for MARL research. One of the key features of MARLlib is its auto-adaptation capability, which allows for seamless execution of algorithms on various environments,including model architecture and interface. Additionally, MARLlib offers flexible customization options through straightforward configuration file modifications.

3.1 Architecture

In this part, we introduce the MARLlib training pipelines from three perspectives:

- agent and environment interaction
- data sampling and training workflow
- core components that form the whole pipeline

3.1.1 Environment Interface

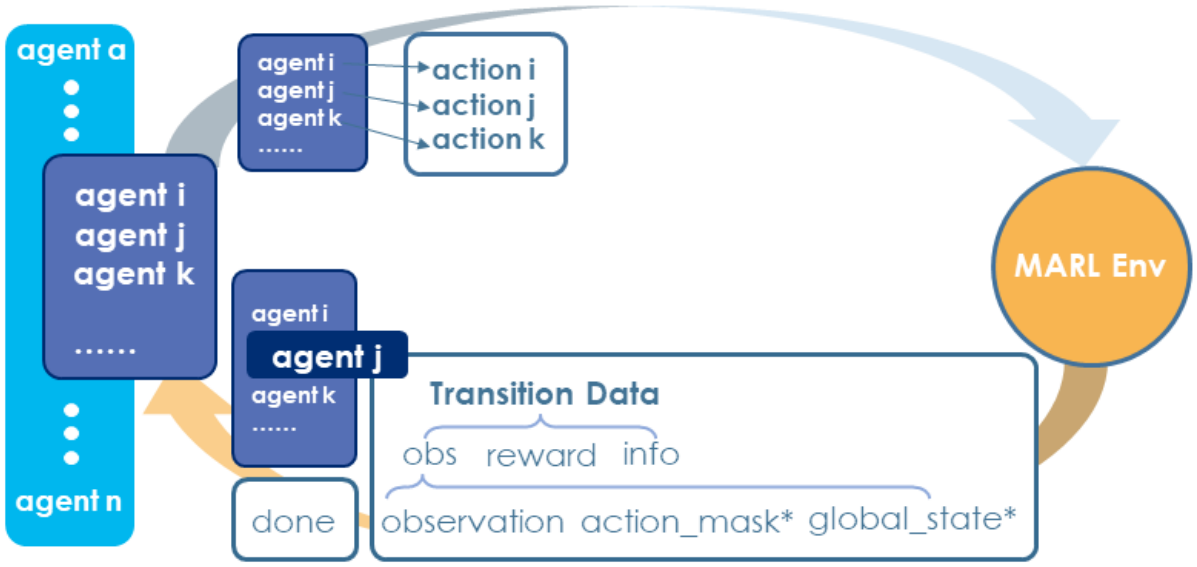


Fig. 1: Agent-Environment Interface in MARLlib

The environment interface in MARLlib enables the following abilities:

1. agent-agnostic: each agent has insulated data in the training stage
2. task-agnostic: diverse environments in one interface
3. asynchronous sampling: flexible agent-environment interaction mode

First, MARLlib treats MARL as the combination of single agent RL processes.

Second, MARLlib unifies all the ten environments into one abstract interface that helps the burden for algorithm design work. And the environment under this interface can be any instance, enabling multi-tasks / task agnostic learning.

Third, unlike most of the existing MARL framework that only supports synchronous interaction between agents and environments, MARLlib supports an asynchronous interacting style. This should be credited to RLlib's flexible data collecting mechanism as data of different agents can be collected and stored in both synchronous and asynchronous ways.

3.1.2 Workflow

Same as RLlib, MARLlib has two phases after launching the process.

Phase 1: Pre-learning

MARLlib commences the reinforcement learning process by instantiating the environment and the agent model. Subsequently, a mock batch is generated based on environment characteristics and fed into the sampling/training pipeline of the designated algorithm. Upon successful completion of the learning workflow with no encountered errors, MARLlib proceeds to the subsequent stage.

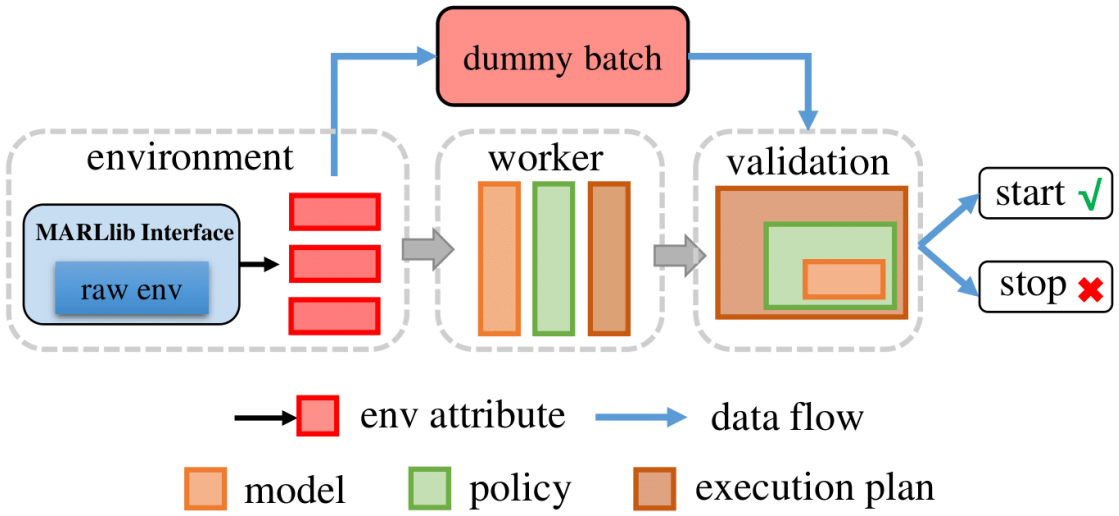


Fig. 2: Pre-learning Stage

Phase 2: Sampling & Training

Upon completion of the pre-learning stage, MARLlib assigns real jobs to the workers and the learner, and schedules these processes under the execution plan to initiate the learning process.

During a standard learning iteration, each worker interacts with its environment instance(s) using agent model(s) to sample data, which is then passed to the replay buffer. The replay buffer is initialized according to the algorithm and decides how the data are stored. For instance, for the on-policy algorithm, the buffer is a concatenation operation, while for the off-policy algorithm, the buffer is a FIFO queue.

Following this, a pre-defined policy mapping function distributes the collected data to different agents. Once all the data for one training iteration are fully collected, the learner begins to optimize the policy/policies using these data, and broadcasts the new model to each worker for the next sampling round.

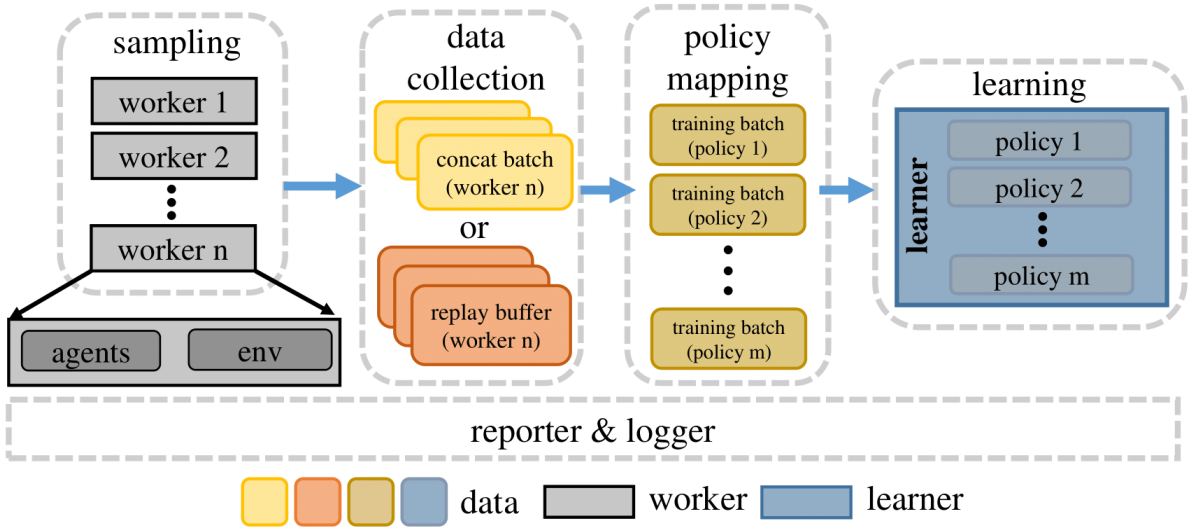
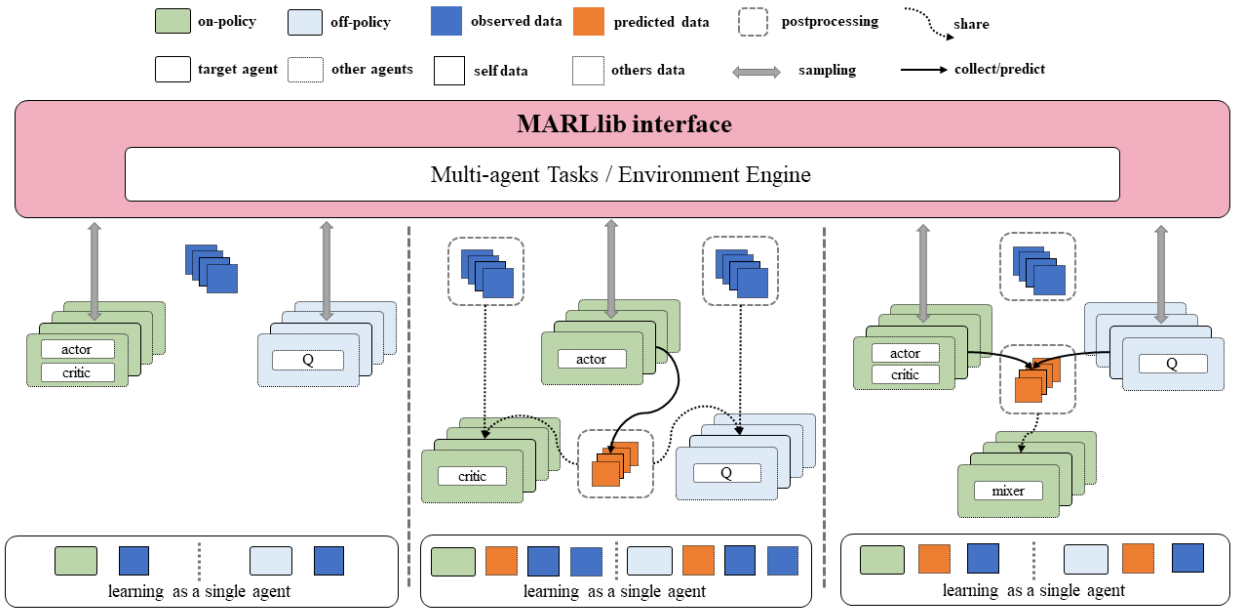


Fig. 3: Sampling & Training Stage

3.1.3 Algorithm Pipeline



(a) Independent Learning (b) Centralized Critic (c) Value Decomposition

Independent Learning

In MARLlib, implementing independent learning (left) is straightforward due to the availability of many algorithms provided by RLlib. To initiate training, one can select an algorithm from RLlib and apply it to the multi-agent environment with no additional effort compared to RLlib. Although independent learning in MARL does not require any data exchange, its performance is typically inferior to that of the centralized training strategy in most tasks.

Centralized Critic

Centralized critic learning is one of the two centralized training strategies in the CTDE framework supported by MARLlib. Under this approach, agents are required to share their information with each other after obtaining the policy output but before the critic value computation. This shared information includes individual observations, actions, and global state (if available).

The exchanged data is collected and stored as transition data during the sampling stage, where each transition data contains both self-collected data and exchanged data. These data are then utilized to optimize a centralized critic function along with a decentralized policy function. The implementation of information sharing is primarily done in the postprocessing function for on-policy algorithms. In the case of off-policy algorithms like MADDPG, additional data such as action value provided by other agents is collected before the data enters the training iteration batch.

Value Decomposition

In MARLlib, Value Decomposition (VD) is another category of centralized training strategies, differing from centralized critics in terms of the information agents are required to share. Specifically, only the predicted Q value or critic value needs to be shared among the agents, and additional data may be necessary depending on the algorithm used. For example, QMIX requires a global state to compute the mixing Q value.

The data collection and storage mechanism for VD is similar to that of centralized critics, with the agents collecting and storing transition data during the sampling stage. The joint Q learning methods (VDN, QMIX) are based on the original PyMARL, with only FACMAC, VDA2C, and VDPPO following the standard RLlib training pipeline among the five VD algorithms.

3.1.4 Key Component

Postprocessing Before Data Collection

MARL algorithms adopting the centralized training with decentralized execution (CTDE) paradigm necessitate the sharing of information among agents during the learning phase. In value decomposition algorithms such as QMIX, FACMAC, and VDA2C, the computation of the total Q or V value requires agents to provide their respective Q or V value estimation. Conversely, algorithms based on centralized criticism such as MADDPG, MAPPO, and HAPPO require agents to share their observation and action data to determine a centralized critic value. The postprocessing module is the ideal location for agents to exchange data with their peers. For centralized critics algorithms, agents may obtain additional information from other agents to calculate a centralized critic value. On the other hand, for value decomposition algorithms, agents must provide their predicted Q or V value to other agents. Additionally, the postprocessing module is also responsible for computing various learning targets using techniques such as GAE or N-step reward adjustment.

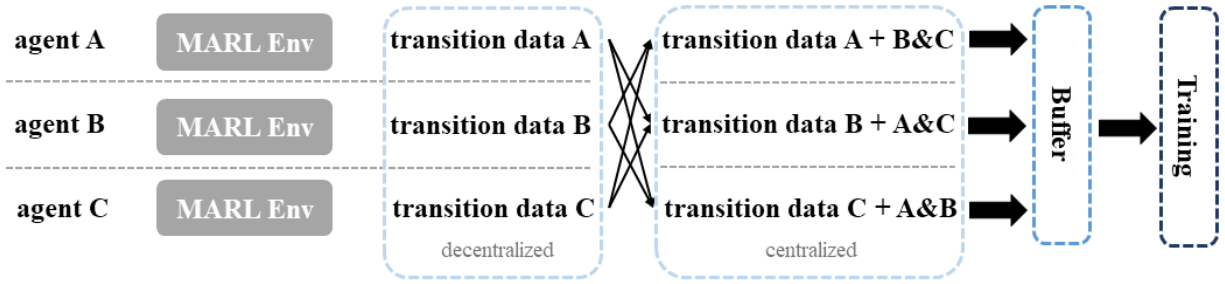


Fig. 4: Postprocessing Before Data Collection

Postprocessing Before Batch Learning

In the context of MARL algorithms, not all algorithms can leverage the postprocessing module. One such example is off-policy algorithms like MADDPG and FACMAC, which face the challenge of outdated data in the replay buffer that cannot be used for current training interactions. To address this challenge, an additional "before batch learning" function is implemented to accurately compute the Q or V value of the current model just before the sampled batch enters the training loop. This ensures that the data used for training is up-to-date and accurate, improving the training effectiveness.

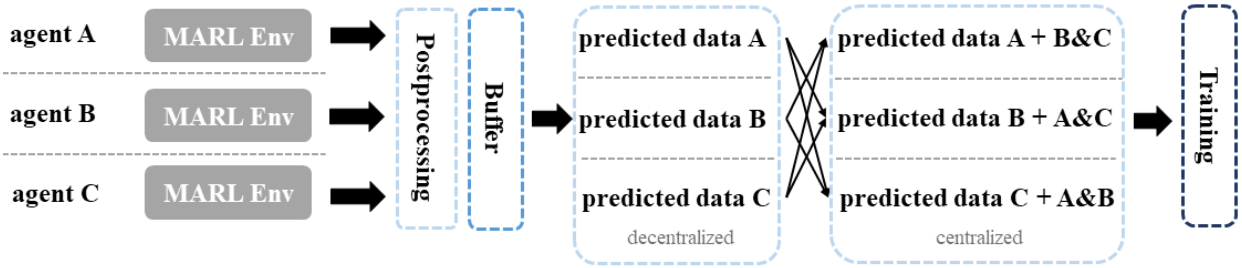


Fig. 5: Postprocessing Before Batch Learning

Centralized Value function

In the centralized critic agent model, the conventional value function based solely on an agent's self-observation is replaced with a centralized critic that can adapt to the algorithm's requirements. The centralized critic is responsible for processing information received from other agents and generating a centralized value as output.

Mixing Value function

In the value decomposition agent model, the original value function is retained, but a new mixing value function is introduced to obtain the overall mixing value. The mixing function is flexible and can be customized as per the user's requirements. Currently, the VDN and QMIX mixing functions are available. To modify the mixing value, the user can make changes to the model configuration file located at **marl/model/configs/mixer**.

Heterogeneous Optimization

In heterogeneous optimization, individual agent parameters are updated independently, and therefore, the policy function is not shared across different agents. However, according to the algorithm proof, updating the policies of agents sequentially and setting the values of the loss-related summons can lead to an incremental summation with any positive update.

To ensure the incremental monotonicity of the algorithm, a trust region is utilized to obtain suitable parameter updates, as is the case in the HATRPO algorithm. To accelerate the policy and critic update process while considering computational efficiency, the proximal policy optimization technique is employed in the HAPPO algorithm.

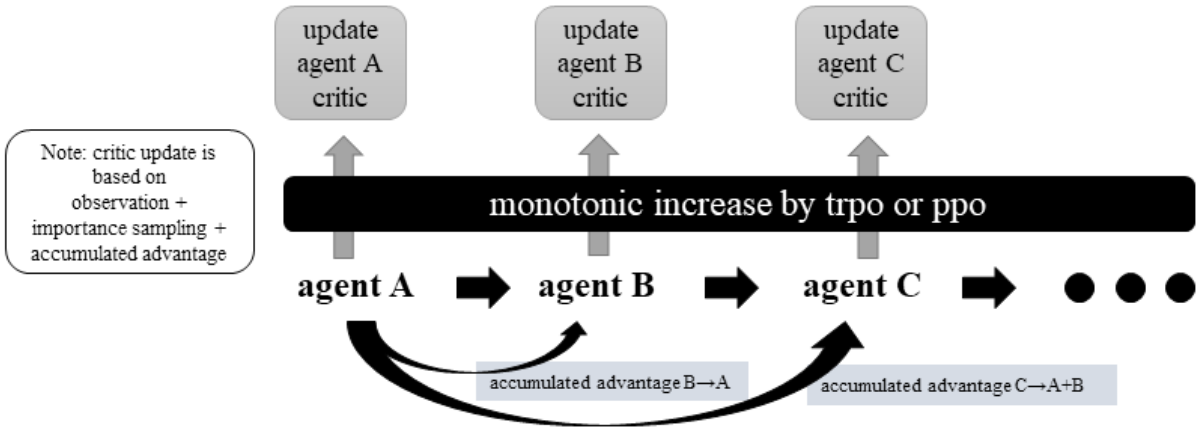


Fig. 6: Heterogeneous Agent Critic Optimization

Policy Mapping

Policy mapping plays a crucial role in standardizing the interface of the Multi-Agent Reinforcement Learning (MARL) environment. In MARLlib, policy mapping is implemented as a dictionary with a hierarchical structure. The top-level key represents the scenario name, the second-level key contains group information, and four additional keys (**description**, **team_prefix**, **all_agents_one_policy**, and **one_agent_one_policy**) are used to define various policy settings. The **team_prefix** key groups the agents based on their names, while the last two keys indicate whether a fully shared or no-sharing policy strategy is applicable for the given scenario. The policy mapping method is utilized to initialize and allocate policies to different agents, and each policy is trained using the data sampled only by the agents in its corresponding policy group.

For instance, consider a mixed mode scenario from MAgent, which can be represented using the following policy mapping:

```
"adversarial_pursuit": {
    "description": "one team attack, one team survive",
    "team_prefix": ("predator_", "prey_"),
    "all_agents_one_policy": False,
    "one_agent_one_policy": False,
},
```

3.2 Algorithms Checklist

3.2.1 Independent Learning

- *IQL: multi-agent version of D(R)QN.*
- IPG
- *IA2C: multi-agent version of A2C*
- *IDDPG: multi-agent version of DDPG*
- *ITRPO: multi-agent version of TRPO*
- *IPPO: multi-agent version of PPO*

3.2.2 Centralized Critic

- *MAA2C: A2C agent with a centralized critic*
- *COMA: MAA2C with Counterfactual Multi-Agent Policy Gradients*
- *MADDPG: DDPG agent with a centralized Q*
- *MATRPO: TRPO agent with a centralized critic*
- *MAPPO: PPO agent with a centralized critic*
- *HATRPO: Sequentially updating critic of MATRPO agents*
- *HAPPO: Sequentially updating critic of MAPPO agents*

3.2.3 Value Decomposition

- *VDN: mixing Q with value decomposition network*
- *QMIX: mixing Q with monotonic factorization*
- *FACMAC: mixing a bunch of DDPG agents*
- *VDA2C: mixing a bunch of A2C agents' critics*
- *VDPPO: mixing a bunch of PPO agents' critics*

3.3 Environment Checklist

Please refer to *Environments*

CHAPTER

FOUR

ENVIRONMENTS

Environment list of MARLlib, including installation and description.

Note: make sure you have read and completed the *Installation* part.

4.1 SMAC

StarCraft Multi-Agent Challenge (SMAC) is a multi-agent environment for collaborative multi-agent reinforcement learning (MARL) research based on Blizzard's StarCraft II RTS game. It focuses on decentralized micromanagement scenarios, where an individual RL agent controls each game unit.

Official Link: https://github.com/oxwhirl/smac

Original Learning Mode	Cooperative
MARLlib Learning Mode	Cooperative + Collaborative
Observability	Partial
Action Space	Discrete
Observation Space Dim	1D
Action Mask	Yes
Global State	Yes
Global State Space Dim	1D
Reward	Dense / Sparse
Agent-Env Interact Mode	Simultaneous

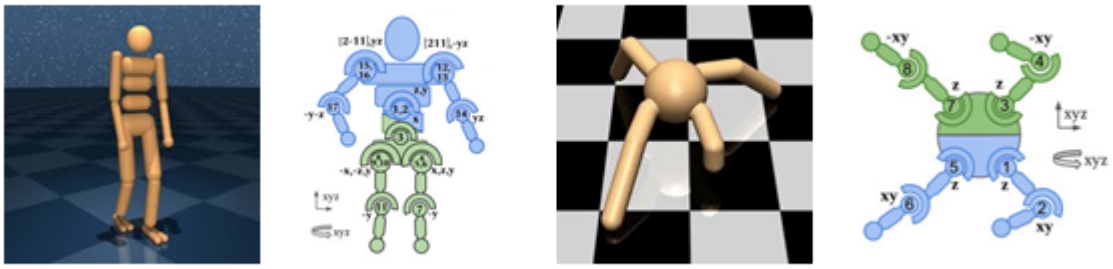
4.1.1 Installation

```
bash install_sc2.sh # https://github.com/oxwhirl/pymarl/blob/master/install_sc2.sh
pip3 install numpy scipy pyyaml matplotlib
pip3 install imageio
pip3 install tensorboard-logger
pip3 install pygame
pip3 install jsonpickle==0.9.6
pip3 install setuptools
pip3 install sacred

git clone https://github.com/oxwhirl/smac.git
cd smac
pip install .
```

Note: the location of the StarcraftII game directory should be pre-defined, or you can just follow the error log (when the process can not found the game's location) and put it in the right place.

4.2 MAMuJoCo



Multi-Agent Mujoco (MAMuJoCo) is an environment for continuous cooperative multi-agent robotic control. Based on the popular single-agent robotic MuJoCo control suite provides a wide variety of novel scenarios in which multiple agents within a single robot have to solve a task cooperatively.

Official Link: https://github.com/schroederdewitt/multiagent_mujoco

Original Learning Mode	Cooperative
MARLlib Learning Mode	Cooperative + Collaborative
Observability	Partial
Action Space	Continuous
Observation Space Dim	1D
Action Mask	No
Global State	Yes
Global State Space Dim	1D
Reward	Dense
Agent-Env Interact Mode	Simultaneous

4.2.1 Installation

```
mkdir /home/YourUserName/.mujoco
cd /home/YourUserName/.mujoco
wget https://roboti.us/download/mujoco200_linux.zip
unzip mujoco200_linux.zip
export LD_LIBRARY_PATH=/home/YourUserName/.mujoco/mujoco200/bin;
pip install mujoco-py==2.0.2.8

git clone https://github.com/schroederdewitt/multiagent_mujoco
cd multiagent_mujoco
mv multiagent_mujoco /home/YourPathTo/MARLlib/multiagent_mujoco

# optional
sudo apt-get install libosmesa6-dev # If you meet GCC error with exit status 1
pip install patchelf-wrapper
```

Note: To access the MuJoCo API, you may get a mjkey (free now) and put it under /home/YourUserName/.mujoco.

4.3 Google Research Football

Google Research Football (GRF) is a reinforcement learning environment where agents are trained to play football in an advanced, physics-based 3D simulator. It also provides support for multiplayer and multi-agent experiments.

Official Link: https://github.com/google-research/football

Original Learning Mode	Collaborative + Competitive
MARLlib Learning Mode	Cooperative + Collaborative
Observability	Full
Action Space	Discrete
Observation Space Dim	2D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Sparse
Agent-Env Interact Mode	Simultaneous

4.3.1 Installation

Google Research Football is somehow a bit tricky for installation. We wish you good luck.

```
sudo apt-get install git cmake build-essential libgl1-mesa-dev libsdl2-dev libsdl2-image-
↪dev libsdl2-ttf-dev libsdl2-gfx-dev libboost-all-dev libdirectfb-dev libst-dev mesa-
↪utils xvfb x11vnc python3-pip
python3 -m pip install --upgrade pip setuptools psutil wheel
```

We provide solutions (may work) for potential bugs

- Compiler error on /usr/lib/x86_64-linux-gnu/libGL.so
- apt-get, unmet dependencies, … "but it is not going to be installed"
- Errors related to Could NOT find Boost

4.4 MPE

Multi-particle Environments (MPE) are a set of communication-oriented environments where particle agents can (sometimes) move, communicate, see each other, push each other around, and interact with fixed landmarks.

Official Link: https://github.com/openai/multiagent-particle-envs

Our version: https://github.com/Farama-Foundation/PettingZoo/tree/master/pettingzoo/mpe

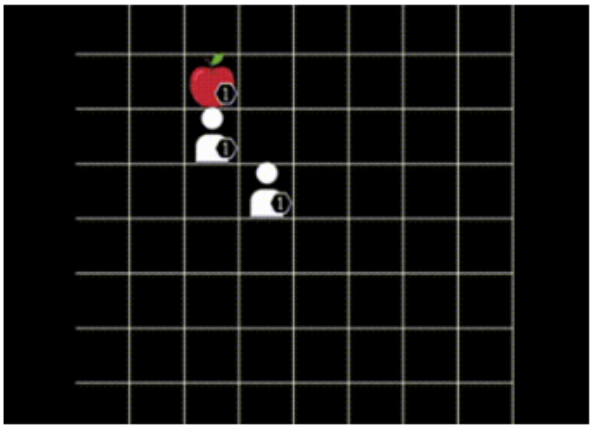
Original Learning Mode	Collaborative + Competitive
MARLlib Learning Mode	Cooperative + Collaborative + Competitive + Mixed
Observability	Full
Action Space	Discrete + Continuous
Observation Space Dim	1D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Dense
Agent-Env Interact Mode	Simultaneous / Asynchronous

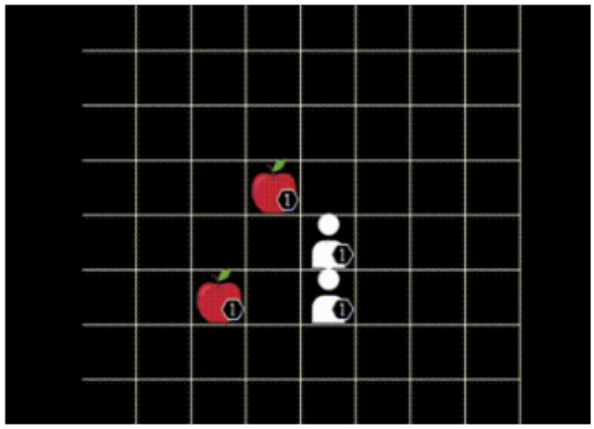
4.4.1 Installation

We use the pettingzoo version of MPE

```
pip install pettingzoo[mpe]
```

4.5 LBF





Level-based Foraging (LBF) is a mixed cooperative-competitive game that focuses on coordinating the agents involved. Agents navigate a grid world and collect food by cooperating with other agents if needed.

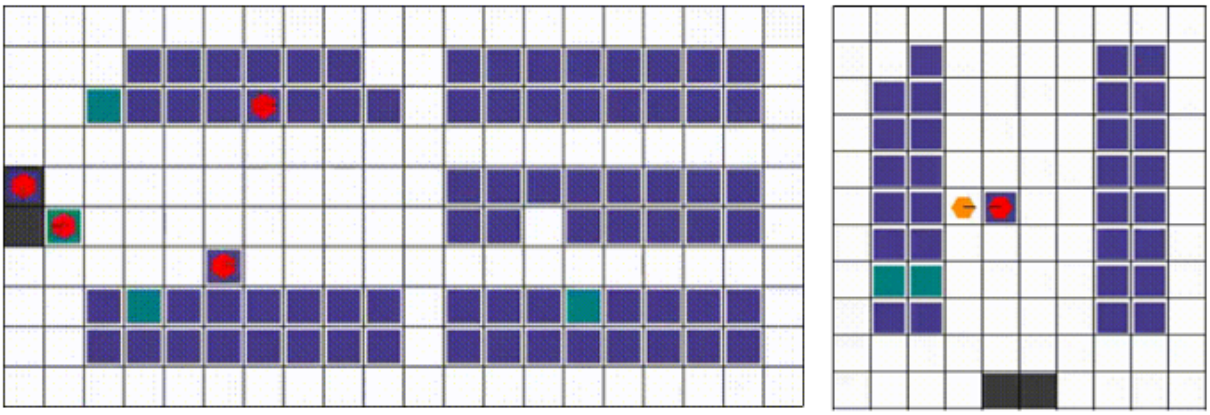
Official Link: https://github.com/semitable/lb-foraging

Original Learning Mode	Cooperative + Collaborative
MARLlib Learning Mode	Cooperative + Collaborative
Observability	Partial
Action Space	Discrete
Observation Space Dim	1D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Dense
Agent-Env Interact Mode	Simultaneous

4.5.1 Installation

```
pip install lbforaging==1.0.15
```

4.6 RWARE



Robot Warehouse (RWARE) simulates a warehouse with robots moving and delivering requested goods. Real-world applications inspire the simulator, in which robots pick up shelves and deliver them to a workstation.

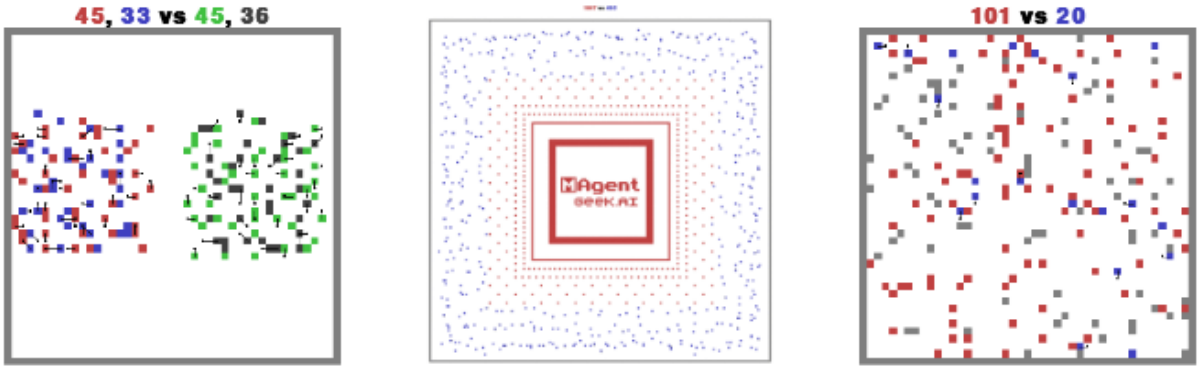
Official Link: https://github.com/semitable/robotic-warehouse

Original Learning Mode	Cooperative
MARLlib Learning Mode	Cooperative + Collaborative
Observability	Partial
Action Space	Discrete
Observation Space Dim	1D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Sparse
Agent-Env Interact Mode	Simultaneous

4.6.1 Installation

```
pip install rware==1.0.1
```

4.7 MAgent



MAgent is a set of environments where large numbers of pixel agents in a grid world interact in battles or other competitive scenarios.

Official Link: https://www.pettingzoo.ml/magent

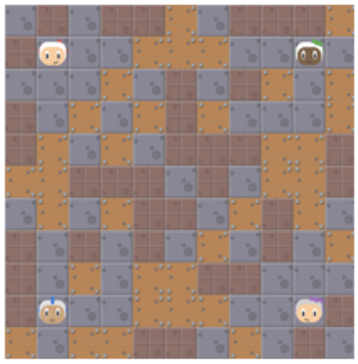
Our version: https://github.com/Farama-Foundation/PettingZoo/tree/master/pettingzoo/mpe

Original Learning Mode	Collaborative + Competitive
MARLlib Learning Mode	Collaborative + Competitive
Observability	Partial
Action Space	Discrete
Observation Space Dim	2D
Action Mask	No
Global State	MiniMap
Global State Space Dim	2D
Reward	Dense
Agent-Env Interact Mode	Simultaneous / Asynchronous

4.7.1 Installation

```
pip install pettingzoo[magent]
```

4.8 Pommerman


Pommerman is stylistically similar to Bomberman, the famous game from Nintendo. Pommerman's FFA is a simple but challenging setup for engaging adversarial research where coalitions are possible, and Team asks agents to be able to work with others to accomplish a shared but competitive goal.

Official Link: https://github.com/MultiAgentLearning/playground

Original Learning Mode	Collaborative + Competitive
MARLlib Learning Mode	Cooperative + Collaborative + Competitive + Mixed
Observability	Full
Action Space	Discrete
Observation Space Dim	2D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Sparse
Agent-Env Interact Mode	Simultaneous

4.8.1 Installation

```
git clone https://github.com/MultiAgentLearning/playground
cd playground
pip install .
cd /home/YourPathTo/MARLlib/patch
python add_patch.py --pommerman
pip install gym==0.21.0
```

4.9 MetaDrive


MetaDrive is a driving simulator that supports generating infinite scenes with various road maps and traffic settings to research generalizable RL. It provides accurate physics simulation and multiple sensory inputs, including Lidar, RGB images, top-down semantic maps, and first-person view images.

Official Link: https://github.com/decisionforce/metadrive

Original Learning Mode	Collaborative
MARLlib Learning Mode	Collaborative
Observability	Partial
Action Space	Continuous
Observation Space Dim	1D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Dense
Agent-Env Interact Mode	Simultaneous

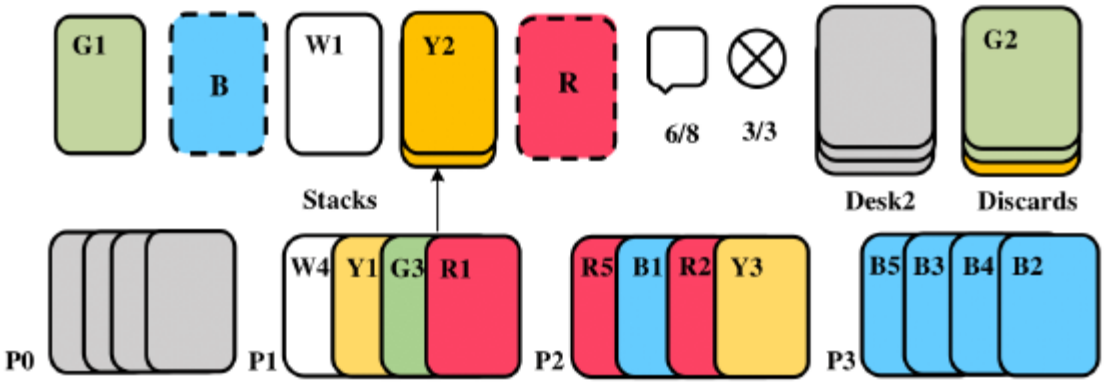
4.9.1 Installation

```
pip install metadrive-simulator==0.2.3
```

4.10 Hanabi

Hanabi is a cooperative card game created by French game designer Antoine Bauza. Players are aware of other players' cards but not their own and attempt to play a series of cards in a specific order to set off a simulated fireworks show.

Official Link: https://github.com/deepmind/hanabi-learning-environment



Original Learning Mode	Collaborative
MARLlib Learning Mode	Collaborative
Observability	Partial
Action Space	Discrete
Observation Space Dim	1D
Action Mask	Yes
Global State	Yes
Global State Space Dim	1D
Reward	Dense
Agent-Env Interact Mode	Asynchronous

4.10.1 Installation

From MAPPO official site

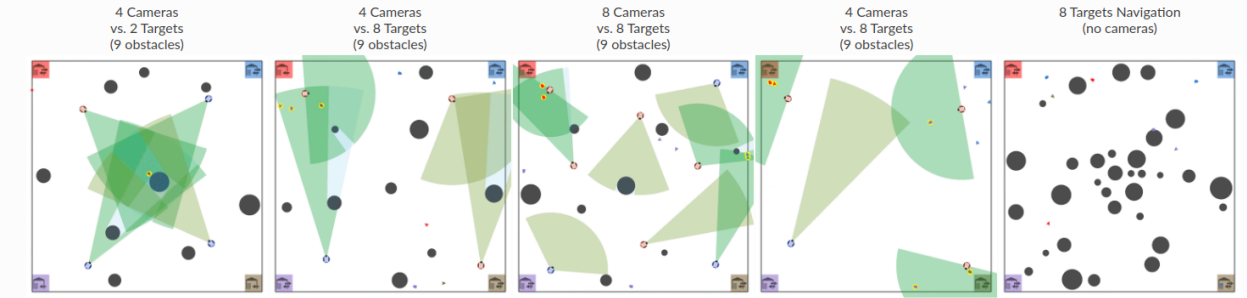
The environment code for Hanabi is developed from the open-source environment code but has been slightly modified to fit the algorithms used here. To install, execute the following:

```
pip install cffi
cd /home/YourPathTo/MARLlib/patch/hanabi
mkdir build
cd build
cmake ..
make -j
```

4.11 MATE

Multi-Agent Tracking Environment (MATE) is an asymmetric two-team zero-sum stochastic game with partial observations, and each team has multiple agents (multiplayer). Intra-team communications are allowed, but inter-team communications are prohibited. It is cooperative among teammates, but it is competitive among teams (opponents).

Official Link: https://github.com/XuehaiPan/mate



Original Learning Mode	Cooperative + Mixed
MARLlib Learning Mode	Cooperative + Mixed
Observability	Partial
Action Space	Discrete + Continuous
Observation Space Dim	1D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Dense
Agent-Env Interact Mode	Simultaneous

4.11.1 Installation

```
pip3 install git+https://github.com/XuehaiPan/mate.git#egg=mate
```

4.12 GoBigger

GoBigger is a game engine that offers an efficient and easy-to-use platform for agar-like game development. It provides a variety of interfaces specifically designed for game AI development. The game mechanics of GoBigger are similar to those of Agar, a popular massive multiplayer online action game developed by Matheus Valadares of Brazil. The objective of GoBigger is for players to navigate one or more circular balls across a map, consuming Food Balls and smaller balls to increase their size while avoiding larger balls that can consume them. Each player starts with a single ball, but can divide it into two when it reaches a certain size, giving them control over multiple balls. Official Link: https://github.com/opendilab/GoBigger

Original Learning Mode	Cooperative + Mixed
MARLlib Learning Mode	Cooperative + Mixed
Observability	Partial + Full
Action Space	Continuous
Observation Space Dim	1D
Action Mask	No
Global State	No
Global State Space Dim	/
Reward	Dense
Agent-Env Interact Mode	Simultaneous

4.12.1 Installation

```
conda install -c opendilab gobigger
```

CHAPTER

FIVE

QUICK START

If you have not installed MARLlib yet, please refer to *Installation* before running.

5.1 Configuration Overview

To start your MARL journey with MARLlib, you need to prepare all the configuration files to customize the whole learning pipeline. There are four configuration files that you need to ensure correctness for your training demand:

- scenario: specify your environment/task settings
- algorithm: finetune your algorithm hyperparameters
- model: customize the model architecture
- ray/rllib: changing the basic training settings

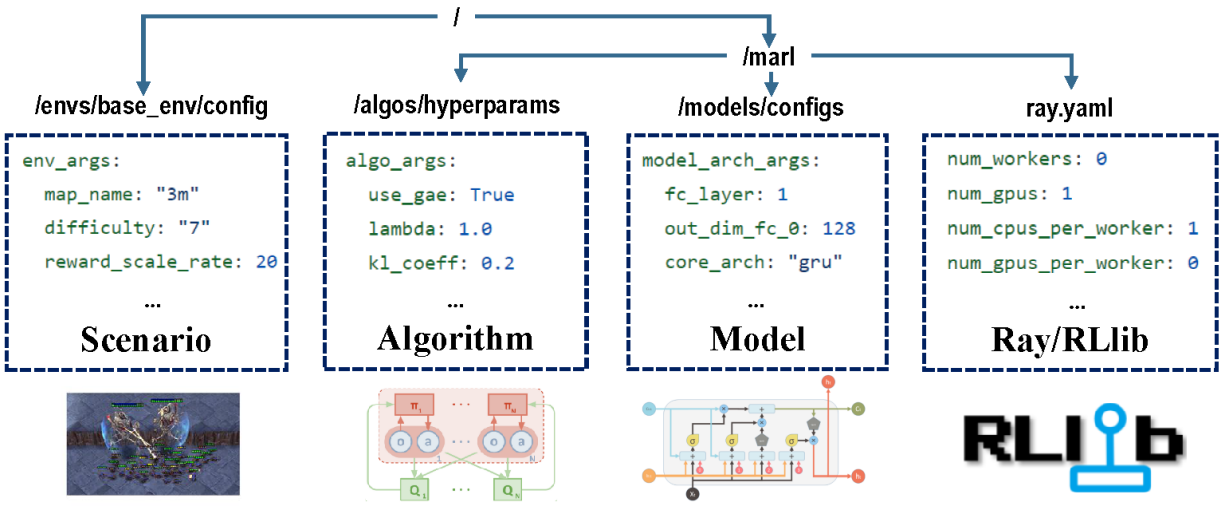


Fig. 1: Prepare all the configuration files to start your MARL journey

5.1.1 Scenario Configuration

MARLlib provides ten environments for you to conduct your experiment. You can follow the instruction in the *Environments* section to install them and change the corresponding configuration to customize the chosen task.

5.1.2 Algorithm Hyper-parameter

After setting up the environment, you need to visit the MARL algorithms' hyper-parameter directory. Each algorithm has different hyper-parameters that you can finetune. Most of the algorithms are sensitive to the environment settings. Therefore, you need to give a set of hyper-parameters that fit the current MARL task.

We provide a commonly used hyper-parameters directory, a test-only hyper-parameters directory, and a finetuned hyper-parameters sets for the three most used MARL environments, including SMAC , MPE, and MAMuJoCo

5.1.3 Model Architecture

Observation space varies with different environments. MARLlib automatically constructs the agent model to fit the diverse input shape, including: observation, global state, action mask, and additional information (e.g., minimap)

However, you can still customize your model in model's config. The supported architecture change includes:

- Observation/State Encoder: CNN, FC
- Multi-layers Perceptron: MLP
- Recurrent Neural Network: GRU, LSTM
- Q/Critic Value Mixer: VDN, QMIX

5.1.4 Ray/RLlib Running Options

Ray/RLlib provides a flexible multi-processing scheduling mechanism for MARLlib. You can modify the file of ray configuration to adjust sampling speed (worker number, CPU number), training speed (GPU acceleration), running mode (locally or distributed), parameter sharing strategy (all, group, individual), and stop condition (iteration, reward, timestep).

5.2 How to Customize

To modify the configuration settings in MARLlib, it is important to first understand the underlying configuration system.

5.2.1 Level of the configuration

There are three levels of configuration, listed here in order of priority from low to high:

- File-based configuration, which includes all the default `*.yaml` files.
- API-based customized configuration, which allows users to specify their own preferences, such as `{"core_arch": "mlp", "encode_layer": "128-256"}`.
- Command line arguments, such as `python xxx.py --ray_args.local_mode --env_args.difficulty=6 --algo_args.num_sgd_iter=6`.

If a parameter is set at multiple levels, the higher level configuration will take precedence over the lower level configuration.

5.2.2 Compatibility across different levels

It is important to ensure that hyperparameter choices are compatible across different levels. For example, the Multiple Particle Environments (MPE) support both discrete and continuous actions. To enable continuous action space settings, one can simply change the *continuous_actions* parameter in the mpe.yaml to **True**. It is important to pay attention to the corresponding setting when using the API-based approach or command line arguments, such as marl.make_env(xxxx, continuous_actions=True), where the argument name must match the one in `mpe.yaml` exactly.

5.3 Training

```
from marllib import marl
# prepare env
env = marl.make_env(environment_name="mpe", map_name="simple_spread")
# initialize algorithm with appointed hyper-parameters
mappo = marl.algos.mappo(hyperparam_source="mpe")
# build agent model based on env + algorithms + user preference
model = marl.build_model(env, mappo, {"core_arch": "mlp", "encode_layer": "128-256"})
# start training
mappo.fit(env, model, stop={"timesteps_total": 1000000}, checkpoint_freq=100, share_
↪policy="group")
```

5.3.1 prepare the `environment`

task mode	api example
cooperative	`marl.make_env(environment_name="mpe", map_name="simple_spread", force_coop=True)`
collaborative	`marl.make_env(environment_name="mpe", map_name="simple_spread")`
competitive	`marl.make_env(environment_name="mpe", map_name="simple_adversary")`
mixed	`marl.make_env(environment_name="mpe", map_name="simple_crypto")`

Most of the popular environments in MARL research are supported by MARLlib:

Env Name	Learning Mode	Observability	Action Space	Observations
LBF	cooperative + collaborative	Both	Discrete	1D
RWARE	cooperative	Partial	Discrete	1D
MPE	cooperative + collaborative + mixed	Both	Both	1D
SMAC	cooperative	Partial	Discrete	1D
MetaDrive	collaborative	Partial	Continuous	1D
MAgent	collaborative + mixed	Partial	Discrete	2D
Pommerman	collaborative + competitive + mixed	Both	Discrete	2D
MAMuJoCo	cooperative	Partial	Continuous	1D
Google Research Football	collaborative + mixed	Full	Discrete	2D
Hanabi	cooperative	Partial	Discrete	1D

Each environment has a readme file, standing as the instruction for this task, including env settings, installation, and important notes.

5.3.2 initialize the `algorithm`

running target	api example
train & finetune	`marl.algos.mappo(hyperparam_source=$ENV)`
develop & debug	`marl.algos.mappo(hyperparam_source="test")`
3rd party env	`marl.algos.mappo(hyperparam_source="common")`

Here is a chart describing the characteristics of each algorithm:

algorithm	support task mode	discrete action	continuous action	policy type
IQL: multi-agent version of D(R)QN.	all four	heavy_check_mark		off-policy
IPG	all four	heavy_check_mark	heavy_check_mark	on-policy
IA2C: multi-agent version of A2C	all four	heavy_check_mark	heavy_check_mark	on-policy
IDDPG: multi-agent version of DDPG	all four		heavy_check_mark	off-policy
ITRPO: multi-agent version of TRPO	all four	heavy_check_mark	heavy_check_mark	on-policy
IPPO: multi-agent version of PPO	all four	heavy_check_mark	heavy_check_mark	on-policy
COMA: MAA2C with Counterfactual Multi-Agent Policy Gradients	all four	heavy_check_mark		on-policy
MADDPG: DDPG agent with a centralized Q	all four		heavy_check_mark	off-policy
MAA2C: A2C agent with a centralized critic	all four	heavy_check_mark	heavy_check_mark	on-policy
MATRPO: TRPO agent with a centralized critic	all four	heavy_check_mark	heavy_check_mark	on-policy
MAPPO: PPO agent with a centralized critic	all four	heavy_check_mark	heavy_check_mark	on-policy
HATRPO: Sequentially updating critic of MATRPO agents	cooperative	heavy_check_mark	heavy_check_mark	on-policy
HAPPO: Sequentially updating critic of MAPPO agents	cooperative	heavy_check_mark	heavy_check_mark	on-policy
VDN: mixing Q with value decomposition network	cooperative	heavy_check_mark		off-policy
QMIX: mixing Q with monotonic factorization	cooperative	heavy_check_mark		off-policy
FACMAC: mixing a bunch of DDPG agents	cooperative		heavy_check_mark	off-policy
VDA2C: mixing a bunch of A2C agents' critics	cooperative	heavy_check_mark	heavy_check_mark	on-policy
VDPPO: mixing a bunch of PPO agents' critics	cooperative	heavy_check_mark	heavy_check_mark	on-policy

***all four**: cooperative collaborative competitive mixed

5.3.3 construct the agent `model`

model arch	api example
MLP	`marl.build_model(env, algo, {"core_arch": "mlp")`
GRU	`marl.build_model(env, algo, {"core_arch": "gru"})`
LSTM	`marl.build_model(env, algo, {"core_arch": "lstm"})`
encoder arch	`marl.build_model(env, algo, {"core_arch": "gru", "encode_layer": "128-256"})`

5.3.4 kick off the training `algo.fit`

setting	api example
train	`algo.fit(env, model)`
debug	`algo.fit(env, model, local_mode=True)`
stop condition	`algo.fit(env, model, stop={'episode_reward_mean': 2000, 'timesteps_total': 10000000})`
policy sharing	`algo.fit(env, model, share_policy='all') # or 'group' / 'individual'`
save model	`algo.fit(env, model, checkpoint_freq=100, checkpoint_end=True)`
GPU accelerate	`algo.fit(env, model, local_mode=False, num_gpus=1)`
CPU accelerate	`algo.fit(env, model, local_mode=False, num_workers=5)`

policy inference `algo.render`

setting	api example
render	`algo.render(env, model, local_mode=True, restore_path='path_to_model')`

By default, all the models will be saved at `/home/username/ray_results/experiment_name/checkpoint_xxxx`

5.4 Logging & Saving

MARLlib uses the default logger provided by Ray in **ray.tune.CLIReporter**. You can change the saved log location here.

CHAPTER

SIX

PART 1. SINGLE AGENT (DEEP) RL

There have been many great RL tutorials and open-sourced repos where you can find both the principle of different RL algorithms and the implementation details. In this part, we will quickly navigate from RL to DRL.

6.1 Reinforcement Learning (RL)

Reinforcement Learning focuses on goal-directed learning from interaction. The learning entity must discover which strategy produces the greatest reward by "trial and error".

6.2 Deep Reinforcement Learning (DRL)

Deep Reinforcement Learning (DRL) combines Reinforcement Learning and Deep Learning. It can solve a wide range of complex decision-making tasks previously out of reach for a machine to solve real-world problems with human-like intelligence.

6.2.1 Deep Learning (DL)

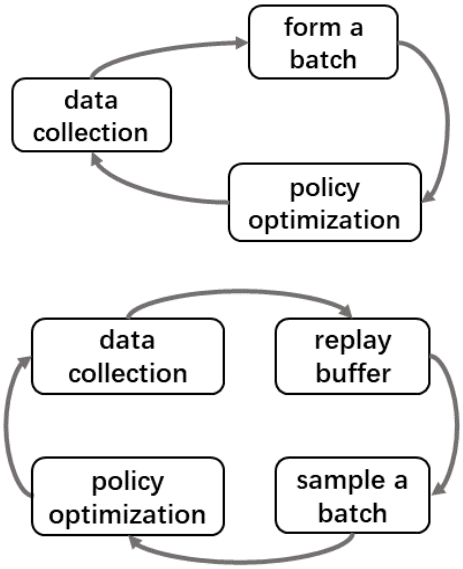
Deep learning can learn from a training set and then apply that learning to a new data set. Deep learning is well known for its function-fitting ability, which can infinitely approximate the optimal mapping function for a high-dimensional problem.

6.2.2 DL + RL

Deep neural networks enable RL with state representation and function approximation for value function, policy, etc. Deep reinforcement learning incorporates deep learning into the solution, allowing agents to make decisions from unstructured input data without manual engineering of the state space. You can find an instance of combining Q learning with Deep learning in *Deep (Recurrent) Q Learning: A Recap*.

6.2.3 Learning Cycle

On-policy (left) and **off-policy** (right) learning cycle:



- **data collection**: agent sends an action to the environment, environment returns some results, including observation, state, reward, etc.
- **form a batch**: policy optimization needs a batch of data from **data collection** to conduct stochastic gradient descent (SGD).
- **replay buffer**: data from **data collection** is sent to the replay buffer for future optimization use.
- **sample a batch**: sample a batch from **replay buffer** follow some rules.
- **policy optimization**: use the data batch to optimize the policy.

6.2.4 RL/DRL Algorithms

A comprehensive collection of RL/DRL algorithms from very old to very new can be found:

- Awesome Deep RL

6.3 Resources

A great RL resource guide includes all kinds of RL-related surveys, books, open-sourced repos, etc.

CHAPTER

SEVEN

PART 2. NAVIGATE FROM RL TO MARL

Reinforcement learning (RL), particularly deep RL, has achieved remarkable success in solving decision-making problems and has become crucial in various fields. One of these fields is multi-agent reinforcement learning (MARL), which is concerned with both the individual learning of an agent in terms of developing an effective strategy and the collective behavior of a group of agents. In MARL, agents need to coordinate with each other when sharing the same group target or adapt to changes when facing adversarial agents.

7.1 MARL: On the shoulder of RL

Multi-agent reinforcement learning (MARL) is an extension of reinforcement learning (RL) that focuses on analyzing group behavior while keeping the core component unchanged, i.e., how an agent optimizes its strategy to obtain higher rewards. Most existing MARL algorithms are built on well-known RL algorithms such as Q learning and Proximal Policy Optimization (PPO), making RL a strong tool for MARL. However, applying plain RL algorithms with no external signal from other agents is suboptimal. To understand the necessity of transitioning from RL to MARL for multi-agent tasks, it is essential to cover some basic concepts.

7.2 Partially Observable Markov Decision Process (POMDP)

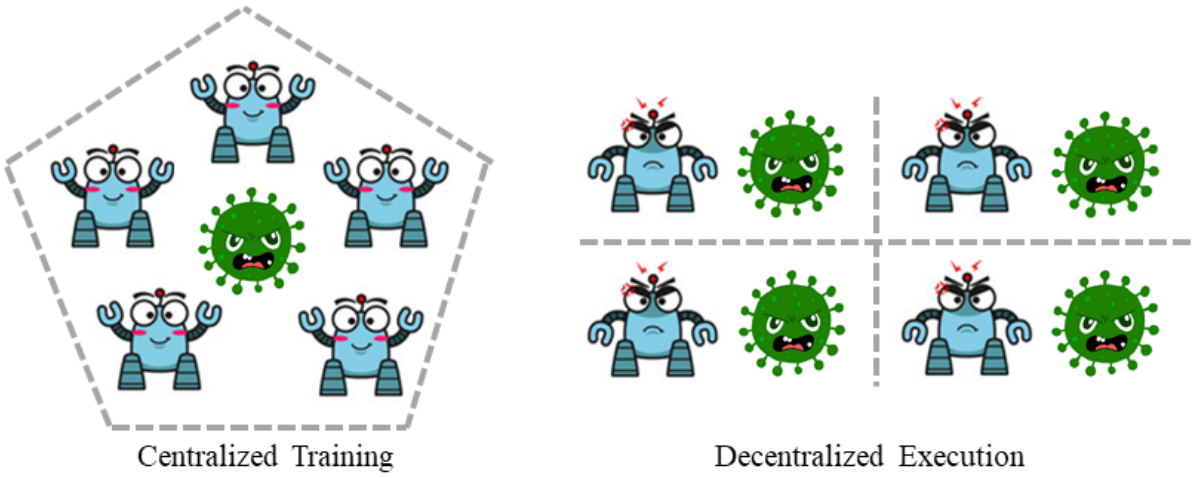
In a Partially Observable Markov Decision Process (POMDP), the system states are unobservable and probabilistically mapped to observations. The agent's access to the system state is limited, and taking the same action can result in different observations. The observation is, however, still dependent on the system state. Hence, the agent must learn or hold a belief about its observation and learn a policy that accounts for all possible states.

POMDP is a more general approach for modeling sequential decision-making problems, especially in multi-agent settings. For instance, in a complex cooperative task, a group of agents may not have access to the entire system information, making it impractical for a single agent to solve the task. Instead, a more practical approach is to provide each agent with local observations to make decisions based on the current situation.

The main challenge lies in helping the agent build its belief system such that its decision-making aligns with its teammates and the system state towards achieving the final goal. To tackle this challenge, one of the most commonly used

techniques is Centralized Training & Decentralized Execution (CTDE), which we will discuss next.

7.3 Centralized Training & Decentralized Execution (CTDE)



The CTDE framework, which stands for Centralized Training & Decentralized Execution, is a widely used approach in multi-agent reinforcement learning (MARL). In this setting, agents are trained together in a centralized manner where they can access all available information, including the global state, other agents' status, and rewards. However, during the execution stage, agents are forced to make decisions based on their local observations, without access to centralized information or communication.

CTDE strikes a balance between coordination learning and deployment cost, making it a popular framework in MARL. Since multi-agent tasks involve numerous agents, learning a policy that aligns with the group target requires incorporating extra information from other sources. Thus, centralized training is the preferred choice. However, after training, the delivery of centralized information is too costly during deployment, leading to delays in decision-making. Furthermore, centralized execution is insecure, as centralized information can be intercepted and manipulated during transmission.

Following the CTDE framework, there are two main branches of MARL algorithms:

- *Centralized Critic*
- *Value Decomposition*

Centralized critic-based algorithms are applicable to all types of multi-agent tasks, including cooperative, collaborative, competitive, and mixed. These algorithms use a centralized critic to approximate the state-action value function, which enables agents to learn a policy that considers the actions of other agents and the global state.

On the other hand, value decomposition-based algorithms can only be applied to cooperative and collaborative scenarios. These algorithms use a value decomposition technique to decompose the value function into individual value functions, one for each agent. The agents then learn their own policies based on their individual value functions. Since value decomposition-based algorithms do not use a centralized critic, they cannot be applied to competitive scenarios where the agents' objectives conflict.

7.4 Diversity: Task Mode, Interacting Style, and Additional Infomation

Multi-agent tasks exhibit a high degree of diversity compared to single-agent tasks. In single-agent settings, diversity is generally limited to the action space, reward function, and observation dimension. In contrast, multi-agent tasks feature a range of diversifying elements, including

1. **Task mode**: how agents work together.
2. **Interacting style**: in which order the agent interacts with the environment.
3. **Additional information**: depends on whether the environment provides it.

These complexities present unique challenges when comparing MARL algorithms. An algorithm that performs well in one multi-agent task may not perform as effectively in another, making it difficult to develop a unified benchmark for MARL.

7.5 The Future of MARL

It can be inferred that MARL provides a bridge between RL and real-world scenarios by enabling a group of agents to learn how to coordinate with each other, providing extra information to guide the evolution of strategies, and equipping agents with the ability to handle diverse tasks with a more general policy. The primary motivations behind MARL are to achieve these goals, as well as to progress towards the development of artificial general intelligence. MARL can now outperform humans in games like chess and MOBA, solve real-world tasks like vision+language-based navigation, help to design a better traffic system, etc.

MARL has been gaining increasing attention in both academic research and industrial applications. With the development of more sophisticated algorithms and hardware, MARL is becoming more practical and effective in solving complex real-world problems. The potential applications of MARL are numerous and span across many different fields, including transportation, logistics, robotics, finance, and more. As the technology continues to advance, we can expect to see even more exciting developments and innovations in the field of MARL in the coming years.

CHAPTER

EIGHT

PART 3. A COLLECTIVE SURVEY OF MARL

8.1 Tasks: Arenas of MARL

In the field of machine learning, it is widely accepted that the evaluation of a new idea necessitates the use of an appropriate dataset. The most effective approach is to employ a representative or widely accepted dataset, adhere to its established evaluation pipeline, and compare the performance of the new idea with other existing algorithms.

In the context of Multi-Agent Reinforcement Learning (MARL), a dataset corresponds to a collection of scenarios that comprise a single multi-agent task. Multi-agent tasks are customizable on a variety of aspects, such as the number of agents, map size, reward function, and unit status. This section provides a brief overview of the categories of multi-agent tasks, ranging from the simplest matrix game to real-world applications.

8.1.1 Matrix Problem and Grid World

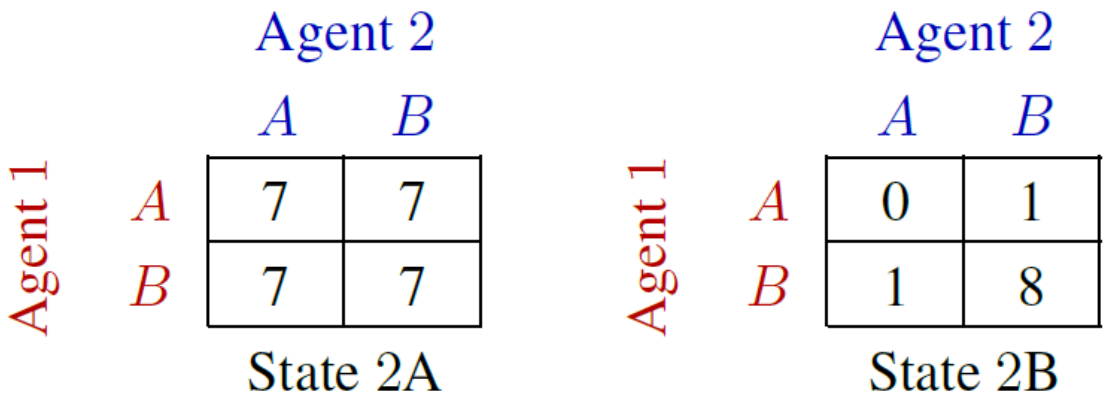


Fig. 1: Two-step game

The first option for evaluating a new idea in MARL involves using a matrix and grid world task.
One such example is the **Two-step Game**.

In this task, two agents act in turn to gain the highest team reward. The task is very straightforward:

1. two agents in the task
2. the observation is a short vector with a length four
3. two actions (A&B) to choose from

Despite the simple task setting, however, the game is still very challenging as one agent needs to coordinate with another agent to achieve the highest reward: the joint action with the highest reward is not a good option from the view of the first agent if it is not willing to cooperate with another agent. **Two-step Game** evaluates whether an agent has learned to cooperate by sacrificing its reward for a higher team reward.

As the value(reward) of the matrix can be customized, the number of matrix combinations (scenarios) that can be solved is a good measurement of the robustness of an algorithm in solving **cooperative-like** multi-agent tasks.

The grid world-based tasks are relatively more complicate than the matrix problem. A well-known grid world example in RL is frozen lake. For MARL, there are many grid-world-based tasks, including:

- *LBF*
- *RWARE*
- *MAgent*

Different tasks target different topics like mixed cooperative-competitive task mode, sparse reward in MARL, and many agents in one system.

8.1.2 Gaming and Physical Simulation

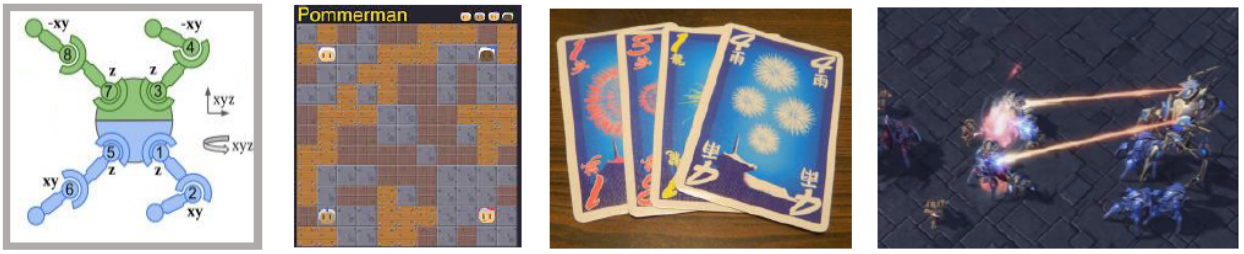


Fig. 2: Gaming & Simulation: MAMuJoCo, Pommerman, Hanabi, Starcraft, etc.

Recent advances in MARL research have shifted the focus towards video gaming and physical simulation, as a means to bridge the gap between simple matrix games and the high cost of sampling and training on real-world scenarios. This approach allows for algorithms to showcase their performance on more complex tasks with a more manageable cost. One of the most popular multi-agent tasks in MARL is StarCraft Multi-Agent Challenge(*SMAC*), which is for discrete control and cooperative task mode. For continuous control, the most used task is the multi-agent version of MuJoCo: (*MAMuJoCo*). To analyze the agent behavior of adversary agents, a typical task is *Pommerman*. Scenarios within one task can contain different task modes, like *MPE*, which simplifies the evaluation procedure of the algorithm's generalization ability within one task domain.

8.1.3 Towards Real-world Application

Fig. 3: Real World Problem: MetaDrive, Flatland, Google Research Football, etc.

Tasks that are real-world-problem oriented, including traffic system design(*MetaDrive*), football(*Google Research Football*), and auto driving, also benchmark recent years' MARL algorithms. These tasks can inspire the next generation of AI solutions. Although the tasks belonging to this categorization are of great significance to the real application, unluckily, fewer algorithms choose to be built on these tasks due to high complexity and standard evaluation procedure.

8.2 Methodology of MARL: Task First or Algorithm First

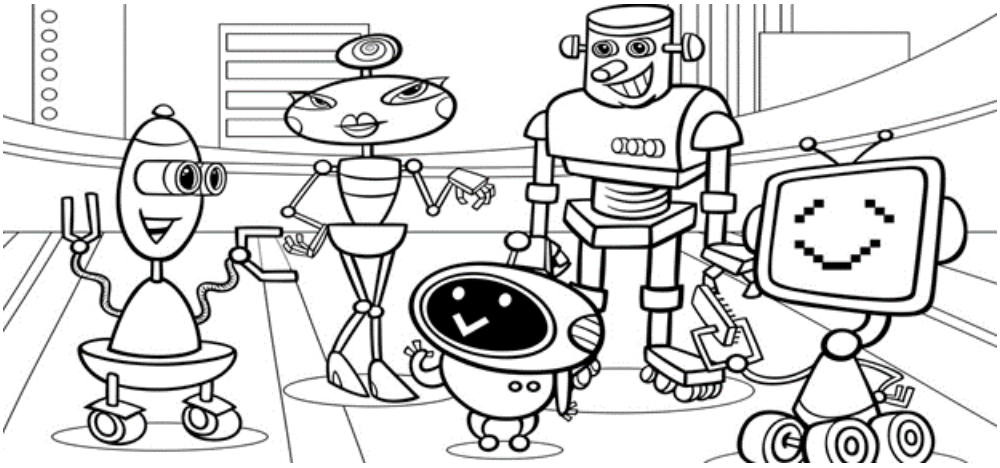
The current state of research on multi-agent reinforcement learning (MARL) is facing challenges regarding the diversity of multi-agent tasks and the categorization of MARL algorithms. These characteristics make it difficult to conduct a fair comparison of different algorithms and raise a question for researchers: should algorithms be developed for a specific task (task first) or for general tasks (algorithm first). This difficulty stems from the nature of multi-agent tasks, as well as the various learning styles and knowledge-sharing strategies.

Since the development of algorithms is closely related to the features of the task, there is a trade-off between the algorithm's ability to generalize on a broad topic and its expertise in a particular multi-agent task.

In the subsequent section, we will provide a brief introduction to how the environment is categorized based on the agents' relationship. We will then classify the algorithms based on their learning style and the connection between the learning style and the agents' relationship.

Finally, we will discuss the extension of MARL algorithms to become more general and applicable to real-world scenarios using knowledge-sharing techniques.

8.2.1 Agent Relationship



In multi-agent reinforcement learning (MARL), the learning of agents is regulated by the relationship among them. Two critical factors affecting the development of MARL algorithms are the working mode and agent similarity.

The first factor is the working mode, also referred to as the task mode, which describes how agents work and learn in a multi-agent task. For instance, a task can be Cooperative-like, where agents share the same goal. Alternatively, a task can be Competitive-like, where agents have different or adversary objectives.

The second factor is agent similarity. In a task with homogeneous agents, they prefer knowledge sharing with each other and learning as a team. Conversely, in a task with heterogeneous agents, they prefer to learn their policies separately.

Task Mode: Cooperative-like or Competitive-like

The task modes in multi-agent reinforcement learning can be broadly classified into two types: Cooperative-like, where agents tend to work as a team towards a shared goal, and Competitive-like, where agents have adversarial targets and exhibit aggression towards other agents.

Mode 1: Cooperative-like

The Cooperative-like task mode is prevalent in scenarios where agents are rewarded only when the team achieves a shared goal. This mode is considered a strict form of cooperation, where each agent cannot access its individual reward. In Cooperative tasks, agents must have a robust credit assignment mechanism to decompose the global reward and update their policies accordingly.

Environments contain **cooperative** scenarios:

- *SMAC*
- *MAMuJoCo*
- *Google Research Football*

- *MPE*
- *LBF*
- *RWARE*
- *Pommerman*

Another mode is **collaborative**, where agents can access individual rewards. Under this mode, the agents tend to work together, but the target varies between different agents. Sometimes individual rewards may cause some potential interest conflict. Collaborative task mode has less restriction and richer reward information for wilder algorithms development: il is a good solution for collaborative tasks, as each agent has been allocated an individual reward for doing a standard RL. *Centralized Critic* is a more robust algorithm family for collaborative tasks as the improved critic help agent coordinate using global information. *Value Decomposition*-based methods are still applicable for collaborative tasks as we can integrate all the individual rewards received into one (only the agents act simultaneously). **Cooperative** mode can also be transformed to **collaborative** as we can copy the global reward to each agent and treat them as an individual reward.

Environments contain **collaborative** scenarios:

- *SMAC*
- *MAMuJoCo*
- *Google Research Football*
- *MPE*
- *LBF*
- *RWARE*
- *Pommerman*
- *MAgent*
- *MetaDrive*
- *Hanabi*

Mode 2: Competitive-like

When agents have different targets in a task, especially when the targets are adversarial, the task can become much more complicated. An example of such a task is a zero-sum game, where the total reward is fixed, and any reward gained by one agent results in an equal loss for another agent. A specific example can be found in *MPE* that in scenarios like **simple_push**, agent ONE is trying to gain more reward by getting closer to its target location while agent TWO gains reward by pushing agent ONE away from the target location.

Moreover, the competitive-like mode can also be not so **pure competitive**. It can incorporate some cooperative agents' relationships. This type of work mode is referred to as **mixed** mode. A representative task of mixed mode is *MAgent*, where agents are divided into several groups. Agents in the same group need to attack the enemy group cooperatively.

Environments contain **competitive** or **mixed** scenarios:

- *MPE*
- *Pommerman*
- *MAgent*

Agents Type: Heterogeneous or Homogeneous

Two methods exist to solve the multi-agent problem, **heterogeneous** and **homogeneous**. Homogeneous agent affiliated with the environment holds the same policy. The policy gives out different actions based on the agent's observation. Heterogeneous methods require each agent to maintain its individual policy, which can accept different environment observation dimensions or output actions with diverse semantic meanings.

8.2.2 Learning Style

Categorizing MARL algorithms by their learning style provides an overview of the topics that researchers are currently interested in. The following are the three main classes:

- Independent Learning: This class applies single-agent RL directly to multi-agent settings without additional coordination mechanisms.
- Centralized Training Decentralized Execution: This class adds extra modules to the training pipeline to help agents learn coordinated behavior while keeping independently executed policies.
- Fully Centralized: In this class, agents are treated as a single agent with multiple actions to execute simultaneously, and the learning algorithm is designed accordingly.

Independent Learning

The core idea of Independent Learning is to extract an independent policy for one agent from the multi-agent system and train it using RL, ignoring other agents and system states. Based on this idea, if every agent learns its policy independently, we can obtain a set of policies that jointly solve the task.

Every RL algorithm can be extended to be MARL compatible, including:

- *IQL: multi-agent version of D(R)QN.*
- *IA2C: multi-agent version of A2C*
- *IDDPG: multi-agent version of DDPG*
- *IPPO: multi-agent version of PPO*
- *ITRPO: multi-agent version of TRPO*

However, independent learning always falls into the local-optimal, and performance degrades rapidly when the multi-agent tasks require a coordinated behavior among agents. This is primarily due to the low utilization of other agents' information and the system's global state.

Centralized Training Decentralized Execution (CTDE)

To enable agents to learn a coordinated behavior while keeping computation budget and optimization complexity low, various learning settings have been proposed in MARL research. Among these, the Centralized Training Decentralized Execution (CTDE) framework has garnered the most attention in recent years. We have introduced the CTDE framework earlier: *Centralized Training & Decentralized Execution (CTDE)*.

Within the CTDE framework, there are two main branches of algorithms: Centralized Critic (CC) and Value Decomposition (VD).

CC-based algorithms can handle general multi-agent tasks but have some restrictions on their architecture. On the other hand, VD-based algorithms are well-suited for solving cooperative-like tasks with a robust credit assignment mechanism, but they have limited applicability.

Type 1. Centralized Critic

CC is first used in MARL since the *MADDPG: DDPG agent with a centralized Q*. As the name indicated, a critic is a must in a CC-based algorithm, which excludes most Q-learning-based algorithms as they have no critic module. Only actor-critic algorithms like *MAA2C: A2C agent with a centralized critic* or actor-Q architecture like *MADDPG: DDPG agent with a centralized Q* fulfill this requirement.

For the training pipeline of CC, the critic is targeting finding a good mapping between the value function and the combination of system state and self-state. This way, the critic is updated regarding the system state and the local states. The policy is optimized using policy gradient according to GAE produced by the critic. The policy only takes the local states as input to conduct a decentralized execution.

The core idea of CC is to provide different information for critics and policy to update them differently. The critic is centralized as it utilizes all the system information to accurately estimate the whole multi-agent system. The policy is decentralized, but as the policy gradient comes from the centralized critic, it can learn a coordinated strategy.

A list of commonly seen CC algorithms:

- *MAA2C: A2C agent with a centralized critic*
- *COMA: MAA2C with Counterfactual Multi-Agent Policy Gradients*
- *MADDPG: DDPG agent with a centralized Q*
- *MATRPO: TRPO agent with a centralized critic*
- *MAPPO: PPO agent with a centralized critic*
- *HATRPO: Sequentially updating critic of MATRPO agents*
- *HAPPO: Sequentially updating critic of MAPPO agents*

Type 2. Value Decomposition

VD is introduced to MARL since the *VDN: mixing Q with value decomposition network*. The name **value decomposition** is based on the fact that the value function of each agent is updated by factorizing the global value function. Take the most used baseline algorithms of VD *VDN: mixing Q with value decomposition network* and *QMIX: mixing Q with monotonic factorization* for instance: VDN sums all the individual value functions to get the global function. QMIX mixes the individual value function and sets non-negative constraints on the mixing weight. The mixed global value function can then be optimized to follow standard RL. Finally, if learnable, backpropagated gradient updates all the individual value functions and the mixer.

Although VDN and QMIX are all off-policy algorithms, the value decomposition can be easily transferred to on-policy algorithms like *VDA2C: mixing a bunch of A2C agents' critics* and *VDPPO: mixing a bunch of PPO agents' critics*. Instead of decomposing the Q value function, on-policy VD algorithms decompose the critic value function. And using the decomposed individual critic function to update the policy function by policy gradient.

The pipeline of the VD algorithm is strictly CTDE. Global information like state and other agent status is only accessible in the mixing stage in order to maintain a decentralized policy or individual Q function.

A list of commonly seen VD algorithms:

- *VDN: mixing Q with value decomposition network*
- *QMIX: mixing Q with monotonic factorization*
- *FACMAC: mixing a bunch of DDPG agents*
- *VDA2C: mixing a bunch of A2C agents' critics*
- *VDPPO: mixing a bunch of PPO agents' critics*

Fully Centralized

A fully centralized method is a viable option when the number of agents and the action space are relatively small. The approach of the fully centralized algorithm to multi-agent tasks is straightforward: all agents and their action spaces are combined into one, and a standard RL pipeline is used to update the policy or Q-value function. For instance, a five-agent discrete control problem can be transformed into a single-agent multi-discrete control problem. Therefore, only a cooperative-like task mode is suitable for this approach, as it would be counterproductive to combine agents that are adversaries to each other.

Although few works focus on fully centralized MARL, it can still serve as a baseline for algorithms of CTDE and others.

8.2.3 Knowledge Sharing

In MARL, agents can share knowledge with others to learn faster or reuse knowledge from previous tasks to adapt quickly to new ones. This is based on the idea that different strategies may share a similar function, which exists across three levels in MARL: agent, scenario, and task.

At the agent level, knowledge sharing is targeted at increasing sample efficiency and improving learning speed. Sharing at the scenario level focuses on developing a multi-task MARL framework to handle multiple scenarios simultaneously within the same task domain. Task-level sharing is the most difficult, and it requires an algorithm to learn and generalize knowledge from one task domain and apply it to a new domain.

Agent Level

Agent-level knowledge sharing primarily focuses on two components: the replay buffer and model parameters. Typically, these two parts are linked, implying that if two agents share model parameters, they also share the replay buffer. However, there are some exceptions where only part of the model is shared. For example, in an actor-critic architecture, if only the critic is shared, the critic is updated with full data, while the policy is updated with the sampled data.

Sharing knowledge across agents can enhance the algorithm's performance by increasing sample efficiency, making it an essential technique in MARL. However, sharing knowledge is not always beneficial. In some cases, diverse individual policy sets are required, and sharing knowledge can significantly reduce this diversity. For example, adversary agents may not share knowledge to maintain competitiveness.

Scenario Level

Scenario-level multi-task MARL is a learning approach that focuses on developing a general policy that can be applied to multiple scenarios within the same task. Compared to task-level multi-task MARL, scenario-level multi-task MARL is more feasible as the learned strategies across different scenarios are more similar than different. For example, skills like hit and run are commonly used across different scenarios in SMAC, despite variations in unit type, agent number, and map terrain.

Recent research has demonstrated that scenario-level knowledge sharing can be achieved through a transformer-based architecture and a meta-learning approach. This holds promise for real-world applications where the working environment is subject to constant changes, requiring agents to quickly adapt to new scenarios.

Task Level

Task-level multi-task MARL aims to learn a self-contained and adaptable strategy without limitations on task mode, enabling agents to effectively reuse knowledge from previous tasks and learn new ones. Achieving task-level knowledge sharing requires agents to identify and apply common principles across different tasks. For example, when presented with a new cooperative task, agents can leverage their understanding of teamwork to quickly find effective solutions. This ability to understand and apply common sense and teamwork concepts is a critical component of human intelligence. Thus, achieving task-level knowledge sharing represents a significant milestone towards the development of artificial general intelligence.

CHAPTER

NINE

JOINT Q LEARNING FAMILY

9.1 Deep (Recurrent) Q Learning: A Recap

Vanilla Q Learning

In Q-learning, the goal is to learn the value of each state-action pair by iteratively updating a table of Q-values. Without the use of deep learning to approximate the Q-values, Q-learning is limited to a Q-table that stores all possible state-action pairs and their corresponding Q-values.

To update the Q-values in the table, the algorithm follows a two-step process. First, the agent selects an action using the epsilon-greedy exploration strategy, which involves choosing the best-known action or exploring a random action. The agent then transitions to the next state and receives a reward from the environment.

In the second step, the Q-value for the state-action pair is updated using the Bellman equation, which takes into account the immediate reward received and the expected future rewards. The updated Q-value is then stored in the Q-table. The process repeats until the Q-values converge or the desired performance is achieved.

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha * (r + \lambda * max_a(s^{'}, a^{'}))$$

Here s is the state. a is the action. $s^{'}$ is the next state. $a^{'}$ is the next action that yields the highest Q value α is the learning rate. λ is the discount factor.

Keeping iterating these two steps and updating the Q-table can converge the Q value. And the final Q value is the reward expectation of the action you choose based on the current state.

Deep Q Learning

Deep Q learning is a significant advancement in Q learning that allows us to approximate the Q value using a neural network. Instead of using a Q-table to store all possible state-action pairs and their corresponding Q values, a Q network is used to approximate the Q value. This is possible because we can encode the state to a feature vector and learn the mapping between the feature vector and the Q value. The Q function is designed to take the state as input and has a number of output dimensions corresponding to the number of possible actions. The max value of output nodes is selected as the Q value for the next state-action pair.

The Q network is updated using the Bellman equation. The optimization target is to minimize the minimum square error (MSE) between the current Q value estimation and the target Q estimation.

$$\phi_{k+1} = \arg\min_{\phi}(Q_\phi(s,a) - (r + \lambda * max_{a'} Q_{\phi_{tar}}(s^{'}, a^{'})))^2$$

The Q_{tag} network is updated every t timesteps copying the Q network.

DQN + Recurrent Neural Network(DRQN)

When dealing with a Partially Observable Markov Decision Process (POMDP), we may not have full access to the state information, and therefore, we need to record the history or trajectory information to assist in choosing the action. Recurrent Neural Networks (RNNs) are introduced to Deep Q Learning to handle POMDPs by encoding the history into the hidden state of the RNN.

In this case, the optimization target is changed to predicting the next Q-value given the current state and the history. We can use the RNN to encode the history of past states and actions, and then pass this encoded information along with the current state to the Q network to predict the Q-value. The goal is to minimize the difference between the predicted Q-value and the target Q-value obtained from the Bellman equation.

$$\phi_{k+1} = \arg\min_{\phi}(Q_\phi(o,h,a) - (r + \lambda * max_{a'} Q_{\phi_{tar}}(o^{'}, h^{'}, a^{'})))^2$$

Here o is the observation as we cannot access the state s. $o^{'}$ is the next observation. h is the hidden state(s) of the RNN. $h^{'}$ is the next hidden state(s) of the RNN.

You Should Know

Navigating from DQN to DRQN, you need to:

- replace the deep Q net's multi-layer perceptron(MLP) module with a recurrent module, e.g., GRU, LSTM.
- store the data in episode format. (while DQN has no such restriction)

9.2 IQL: multi-agent version of D(R)QN.

Quick Facts

- Independent Q Learning (IQL) is the natural extension of q learning under multi-agent settings.
- Agent architecture of IQL consists of one module: `Q`.
- IQL is applicable for cooperative, collaborative, competitive, and mixed task modes.

9.2.1 Workflow

In IQL, each agent follows a standard D(R)QN sampling/training pipeline.

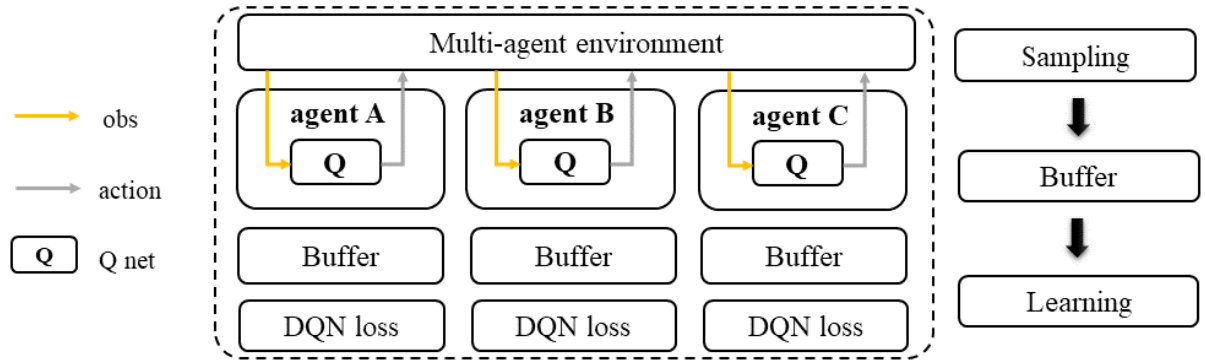


Fig. 1: Independent Q Learning (IQL)

9.2.2 Characteristic

action space

`discrete`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

`off-policy`	`stochastic`	`independent learning`

9.2.3 Insights

Preliminary

- *Deep (Recurrent) Q Learning: A Recap*

In Independent Q-Learning (IQL), each agent in a multi-agent system is treated as a single agent and uses its own collected data as input to conduct the standard DQN or DRQN learning procedure. This means that each agent learns its own Q-function independently without any information sharing among the other agents. However, knowledge sharing across agents is possible but optional in IQL.

Information Sharing

In the field of multi-agent learning, the term "information sharing" can be vague and unclear, so it's important to provide clarification. We can categorize information sharing into three types:

- real/sampled data: observation, action, etc.
- predicted data: Q/critic value, message for communication, etc.
- knowledge: experience replay buffer, model parameters, etc.

Traditionally, knowledge-level information sharing has been viewed as a "trick" and not considered a true form of information sharing in multi-agent learning. However, recent research has shown that knowledge sharing is actually crucial for achieving optimal performance. Therefore, we now consider knowledge sharing to be a valid form of information sharing in multi-agent learning.

9.2.4 Math Formulation

Standing at the view of a single agent, the mathematical formulation of IQL is the same as *Deep (Recurrent) Q Learning: A Recap*.

Note in multi-agent settings, all the agent models and buffer can be shared, including:

- replay buffer $\mathcal{D}$.
- Q function Q_ϕ.
- target Q function $Q_{\phi_{\text{targ}}}$.

9.2.5 Implementation

Our implementation of IQL is based on the vanilla implementation in RLlib, but we have made some additional improvements to ensure that its performance matches the official implementation. The differences between our implementation and the vanilla implementation can be found in the following:

- `episode_execution_plan`
- `EpisodeBasedReplayBuffer`
- `JointQLoss`
- `JointQPolicy`

Key hyperparameters location:

- `marl/algos/hyperparams/common/iql`
- `marl/algos/hyperparams/finetuned/env/iql`

9.3 VDN: mixing Q with value decomposition network

Quick Facts

- Value Decomposition Network(VDN) is one of the value decomposition versions of IQL.
- Agent architecture of VDN consists of one module: Q network.
- VDN is applicable for cooperative and collaborative task modes.

9.3.1 Workflow

In the VDN approach, each agent follows the same D(R)QN sampling pipeline as in other deep Q-learning methods. However, before entering the training loop, each agent shares its Q value and target Q value with other agents. During the training loop, the Q value and target Q value of the current agent and other agents are summed to obtain the Q_{tot} value. This summation allows each agent to incorporate the impact of other agents' actions on the environment and make more informed decisions.

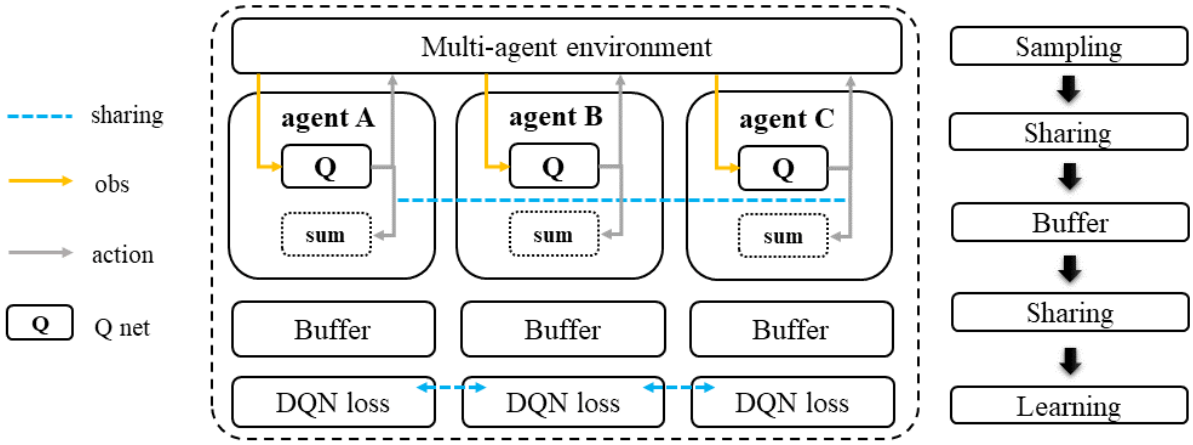


Fig. 2: Value Decomposition Network (VDN)

9.3.2 Characteristic

action space

discrete

task mode

cooperative	collaborative

taxonomy label

off-policy	stochastic	value decomposition

9.3.3 Algorithm Insights

Preliminary

- *IQL: multi-agent version of D(R)QN.*

Optimizing the joint policy of multiple agents with a single team reward can be a challenging task due to the large combined action and observation space. Value Decomposition Network (VDN) was introduced as a solution to this problem. The algorithm decomposes the joint Q value into the sum of individual Q values for each agent. This allows each agent to learn and optimize its own policy independently while still contributing to the team reward. In VDN, each agent follows a standard D(R)QN sampling pipeline and shares its Q value and target Q value with other agents before entering the training loop. The Q value and target Q value of the current agent and other agents are then summed in the training loop to get the total Q value.

- Each agent is still a standard Q, use self-observation as input and output the action logits(Q value).
- The Q values of all agents are added together for mixed Q value annotated as Q_{tot}
- Using standard DQN to optimize the Q net using Q_{tot} with the team reward r.
- The gradient each Q net received depends on the **contribution** of its Q value to the Q_{tot}:

The Q net that outputs a larger Q will be updated more; the smaller will be updated less.

The value decomposition version of IQL is also referred as **joint Q learning**(JointQ). These two names emphasize different aspects. Value decomposition focuses on how the team reward is divided to update the Q net, known as credit assignment. Joint Q learning shows how the optimization target Q_{tot} is got. As VDN is developed to address the cooperative multi-agent task, sharing the parameter is the primary option, which brings higher data efficiency and a smaller model size.

You Should Know:

VDN is the first algorithm that decomposes the joint value function for cooperative multi-agent tasks. However, simply summing the Q value across agents can lead to a reduced diversity of policy and can quickly get stuck in a local optimum, particularly when the Q network is shared across agents.

9.3.4 Math Formulation

VDN needs information sharing across agents. Here we bold the symbol (e.g., o to $\mathbf{o}$) to indicate that more than one agent information is contained.

Q sum: add all the Q values to get the total Q value

$$Q_{\phi}^{tot} = \sum_{i=1}^{n} Q_{\phi}^{i}$$

Q learning: every iteration get a better total Q value estimation, passing gradient to each Q function to update it.

$$L(\phi, \mathcal{D}) = \mathop{\mathrm{E}}_{\tau \sim \mathcal{D}} \left(Q_{\phi}^{tot} - \left(r + \gamma (1 - d) Q_{\phi_{targ}}^{tot'} \right) \right)^2$$

Here $\mathcal{D}$ is the replay buffer, which can be shared across agents. r is the reward. d is set to 1(True) when an episode ends else 0(False). γ is discount value. Q_{ϕ} is Q net, which can be shared across agents. $Q_{\phi_{\text{targ}}}$ is target Q net, which can be shared across agents.

9.3.5 Implementation

We use vanilla VDN implementation of RLlib, but with further improvement to ensure the performance is aligned with the official implementation. The differences between ours and vanilla VDN can be found in

- `episode_execution_plan`
- `EpisodeBasedReplayBuffer`
- `JointQLoss`
- `JointQPolicy`

Key hyperparameters location:

- `marl/algos/hyperparams/common/vdn`
- `marl/algos/hyperparams/finetuned/env/vdn`

9.4 QMIX: mixing Q with monotonic factorization

Quick Facts

- Monotonic Value Function Factorisation(QMIX) is one of the value decomposition versions of IQL.
- Agent architecture of QMIX consists of two modules: `Q` and `Mixer`.
- QMIX is applicable for cooperative and collaborative task modes.

9.4.1 Workflow

In QMIX, each agent follows a standard D(R)QN sampling pipeline and shares its Q-value and target Q-value with other agents before entering the training loop. During the training loop, the Q-value and target Q-value of the current agent and other agents are fed into the Mixer to obtain the overall Q-value of the team, denoted as Q_{tot}.

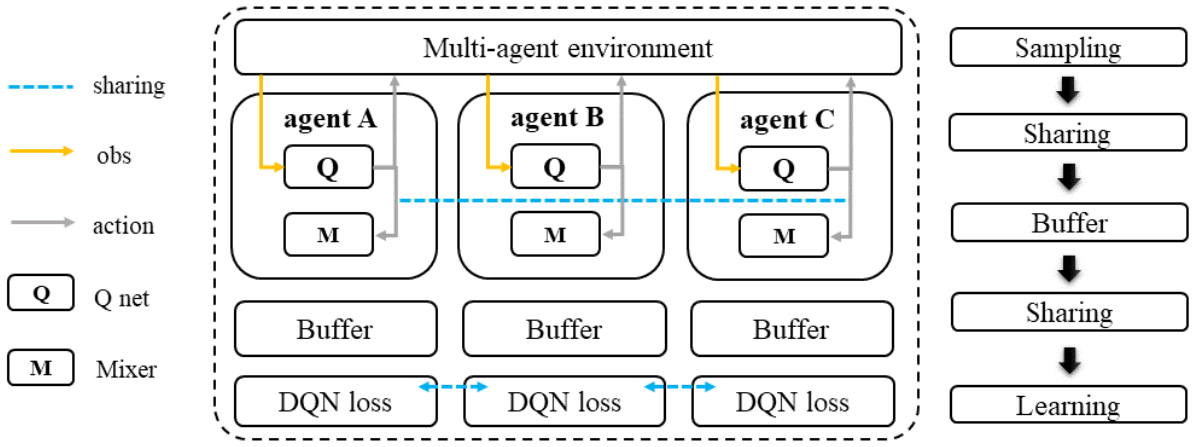


Fig. 3: Monotonic Value Function Factorisation (QMIX)

9.4.2 Characteristic

action space

discrete

task mode

cooperative	collaborative

taxonomy label

off-policy	stochastic	value decomposition

9.4.3 Algorithm Insights

Preliminary

- *IQL: multi-agent version of D(R)QN.*
- *VDN: mixing Q with value decomposition network*

VDN optimizes multiple agents' joint policy by a straightforward operation: sum all the rewards. However, this operation reduces the representation of the strategy because the full factorization is not necessary for extracted decentralized policies to be entirely consistent with the centralized counterpart.

Simply speaking, VDN force each agent to find the best action to satisfy the following equation:

$$\underset{\mathbf{u}}{\operatorname{argmax}}\, Q_{tot}(\boldsymbol{\tau}, \mathbf{u}) = \begin{pmatrix} \underset{u^1}{\operatorname{argmax}}\, Q_1(\tau^1, u^1) \\ \vdots \\ \underset{u^n}{\operatorname{argmax}}\, Q_n(\tau^n, u^n) \end{pmatrix}$$

QMIX claims that a larger family of monotonic functions is sufficient for factorization (value decomposition) but not necessary to satisfy the above equation The monotonic constraint can be written as:

$$\frac{\partial Q_{tot}}{\partial Q_a} \geq 0, \ \forall a \in A$$

With monotonic constraints, we need to introduce a feed-forward neural network that takes the agent network outputs as input and mixes them monotonically. To satisfy the monotonic constraint, the weights (but not the biases) of the mixing network are restricted to be non-negative.

This neural network is named **Mixer**.

The similarity of QMIX and VDN:

- Each agent is still a standard Q function, use self-observation as input and output the action logits(Q value).
- Using standard DQN to optimize the Q function using Q_{tot} with the team reward r.

Difference:

- Additional model **Mixer** is added into QMIX.
- The Q values of all agents are fed to the **Mixer** for getting Q_{tot}.
- The gradient each Q function received is backpropagated from the **Mixer**.

Similar to VDN, QMIX is only applicable to the cooperative multi-agent task. Sharing the parameter is the primary option, which brings higher data efficiency and smaller model size.

You Should Know:

Variants of QMIX are proposed, like WQMIX and Q-attention. However, in practice, a finetuned QMIX (RIIT) is all you need.

9.4.4 Math Formulation

QMIX needs information sharing across agents. Here we bold the symbol (e.g., s to $\mathbf{s}$) to indicate that more than one agent information is contained.

Q mixing: a learnable mixer computing the global Q value by mixing all the Q values.

$$Q_{tot}(\mathbf{a}, s; \boldsymbol{\phi}, \psi) = g_{\psi}\big(\text{`}\mathbf{s}, Q_{\phi_1}, Q_{\phi_2}, .., Q_{\phi_n}\big)$$

Q learning: every iteration get a better total global Q value estimation, passing gradient to both mixer and each Q function to update them.

$$L(\phi, \mathcal{D}) = \mathop{\mathrm{E}}_{\tau \sim \mathcal{D}}\left(Q_{\phi}^{tot} - \left(r + \gamma(1-d)Q_{\phi_{targ}}^{tot'}\right)\right)^2$$

Here $\mathcal{D}$ is the replay buffer, which can be shared across agents. r is the reward. d is set to 1(True) when an episode ends else 0(False). γ is discount value. Q_{ϕ} is Q function, which can be shared across agents. $Q_{\phi_{\text{targ}}}$ is target Q function, which can be shared across agents. g_{ψ} is mixing network. $g_{\psi_{\text{targ}}}$ is target mixing network.

9.4.5 Implementation

In our implementation of QMIX in RLlib, we have made some changes to improve its performance and make it consistent with the official implementation. These differences can be found in the code and configuration files used in our implementation, such as changes in the network architecture or the hyperparameters used during training. We have made these changes to ensure that our version of QMIX is capable of achieving similar or better results compared to the official implementation.

- `episode_execution_plan`
- `EpisodeBasedReplayBuffer`
- `JointQLoss`
- `JointQPolicy`

Key hyperparameters location:

- `marl/algos/hyperparams/common/qmix`
- `marl/algos/hyperparams/finetuned/env/qmix`

9.5 Read List

- Human-level control through deep reinforcement learning
- Deep Recurrent Q-learning for Partially Observable MDPs
- Value-Decomposition Networks For Cooperative Multi-Agent Learning
- QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning

CHAPTER

TEN

DEEP DETERMINISTIC POLICY GRADIENT FAMILY

10.1 Deep Deterministic Policy Gradient: A Recap

Preliminary

- Q-Learning & Deep Q Network(DQN)

Deep Deterministic Policy Gradient (DDPG) is a popular reinforcement learning algorithm that can handle continuous action spaces. DDPG concurrently learns a Q-function and a policy network using off-policy data and the Bellman equation. The Q-function estimates the expected cumulative reward for taking a specific action in a given state, and the policy network produces actions to maximize the Q-value.

To extend Q-learning to continuous action spaces, DDPG introduces an additional policy network, represented by the function $\mu(s;\theta)$, which takes the state as input and outputs a continuous action. The Q-value is estimated as $Q(s,a;\phi)$ using the state-action pair as input, where the Q-function is represented by ϕ, and a is obtained by applying the policy network on the state. By learning both the Q-function and the policy network concurrently, DDPG can handle continuous action spaces effectively.

Mathematical Form

Q learning:

$$L(\phi,\mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_\phi(s,a) - \left(r+\gamma(1-d)Q_{\phi_{\text{targ}}}(s',\mu_{\theta_{\text{targ}}}(s'))\right)\right)^2\right]$$

Policy learning:

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}}\left[Q_\phi(s,\mu_\theta(s))\right]$$

Here $\mathcal{D}$ is the replay buffer a is the action taken. r is the reward. s is the observation/state. s' is the next observation/state. d is set to 1 (True) when episode ends else 0 (False). γ is discount value. μ_θ is policy function. Q_ϕ is Q function. $\mu_{\theta_{\text{targ}}}$ is target policy function $Q_{\phi_{\text{targ}}}$ is target Q function.

You Should Know

Tricks like *gumble softmax* enables DDPG policy function to output categorical-like action distribution.

10.2 IDDPG: multi-agent version of DDPG

Quick Facts

- Independent deep deterministic policy gradient (IDDPG) is a natural extension of DDPG under multi-agent settings.
- An IDDPG agent architecture consists of two models: `policy` and `Q`.
- IDDPG is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary

- *Deep Deterministic Policy Gradient: A Recap*

10.2.1 Workflow

Each agent follows the standard DDPG learning pipeline, which concurrently learns a Q-function and a policy. The models and buffers used in the pipeline can be shared or separated according to the agents' group. This flexibility also applies to all algorithms in the DDPG family. For example, the models and buffers can be shared across all agents to allow for better coordination and faster learning, or they can be trained separately to allow for agents to learn individual strategies.

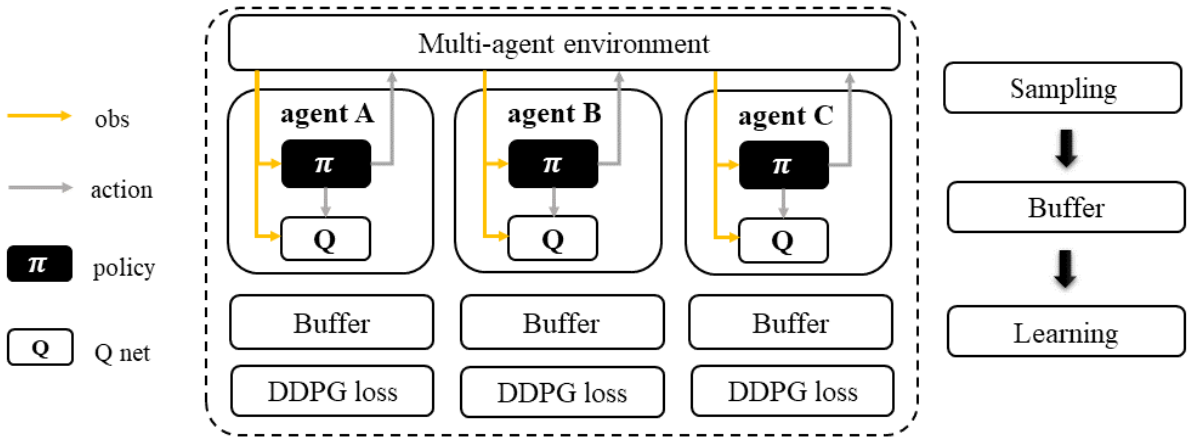


Fig. 1: Independent Deep Deterministic Policy Gradient (IDDPG)

10.2.2 Characteristic

action space

`continuous`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

`off-policy`	`deterministic`	`independent learning`

10.2.3 Insights

Independent Deep Deterministic Policy Gradient (IDDPG) is a version of the Deep Deterministic Policy Gradient (DDPG) algorithm designed for multi-agent scenarios. In IDDPG, each agent has its own DDPG algorithm that samples data and learns independently from other agents. Unlike other multi-agent algorithms, IDDPG does not require agents to share any information, including real or sampled data and predicted actions.

However, IDDPG does allow for optional knowledge sharing between agents. This means that agents can choose to share their models or other information with each other to improve their learning performance. Overall, IDDPG is a flexible and scalable algorithm that can be applied to a wide range of multi-agent scenarios.

Information Sharing

In the field of multi-agent learning, the term "information sharing" can be vague and unclear, so it's important to provide clarification. We can categorize information sharing into three types:

- real/sampled data: observation, action, etc.
- predicted data: Q/critic value, message for communication, etc.
- knowledge: experience replay buffer, model parameters, etc.

Traditionally, knowledge-level information sharing has been viewed as a "trick" and not considered a true form of information sharing in multi-agent learning. However, recent research has shown that knowledge sharing is actually crucial for achieving optimal performance. Therefore, we now consider knowledge sharing to be a valid form of information sharing in multi-agent learning.

10.2.4 Mathematical Form

Standing at the view of a single agent, the mathematical formulation of IDDPG is the same as DDPG: *Deep Deterministic Policy Gradient: A Recap*. , except that in MARL, agent usually has no access to the global state typically under partial observable setting. Therefore, we use o for local observation and s for the global state. We then rewrite the mathematical formulation of DDPG as:

Q learning: get a better Q function

$$L(\phi, \mathcal{D}) = \underset{(o,u,r,o',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(o,u) - \left(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(o', \mu_{\theta_{\text{targ}}}(o'))\right) \right)^2 \right]$$

Policy learning: maximize the Q function output by updating the policy function.

$$\max_\theta \underset{o\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(o, \mu_\theta(o)) \right]$$

Here $\mathcal{D}$ is the replay buffer a is the action taken. r is the reward. o is the local observation. o' is the next local observation. d is set to 1 (True) when episode ends else 0 (False). γ is discount value. μ_θ is policy function. Q_ϕ is Q function. $\mu_{\theta_{\text{targ}}}$ is target policy function $Q_{\phi_{\text{targ}}}$ is target Q function.

Note in multi-agent settings, all the agent models and buffer can be shared, including:

- replay buffer $\mathcal{D}$.
- policy function μ_θ.
- Q function Q_ϕ.
- target policy function $\mu_{\theta_{\text{targ}}}$.
- target Q function $Q_{\phi_{\text{targ}}}$.

10.2.5 Implementation

We extend the vanilla IDDPG of RLlib to be recurrent neural network(RNN) compatible. The main differences are:

- **model side: the agent model-related modules and functions are rewritten, including:**
  - `build_rnnddpg_models_and_action_dist`
  - `DDPG_RNN_TorchModel`
- **algorithm side: the sampling and training pipelines are rewritten, including:**
  - `episode_execution_plan`
  - `ddpg_actor_critic_loss`

Key hyperparameter location:

- `marl/algos/hyperparams/common/ddpg`
- `marl/algos/hyperparams/fintuned/env/ddpg`

Continuous Control Tasks

- **There is only a few MARL dataset focusing on continuous control. The popular three are:**
  - *MPE* (discrete+continuous)
  - *MAMuJoCo* (continuous only)
  - *MetaDrive* (continuous only)

10.3 MADDPG: DDPG agent with a centralized Q

Quick Facts

- Multi-agent deep deterministic policy gradient(MADDPG) is one of the extended version of *IDDPG: multi-agent version of DDPG*.
- Agent architecture of MADDPG consists of two models: `policy` and `Q`.
- MADDPG is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary

- *IDDPG: multi-agent version of DDPG*

10.3.1 Workflow

During the sampling stage, each agent in MADDPG follows the same DDPG learning pipeline to infer an action. However, instead of computing the Q-value based on its own action, each agent uses a centralized Q-function that takes all agents' actions as input to compute the Q-value. This requires sharing data among agents before storing it in the buffer.

During the learning stage, each agent predicts its next action using the target policy and shares it with other agents before entering the training loop. This is done to ensure that all agents use the same action for computing the Q-value in the centralized Q-function during the next sampling stage.

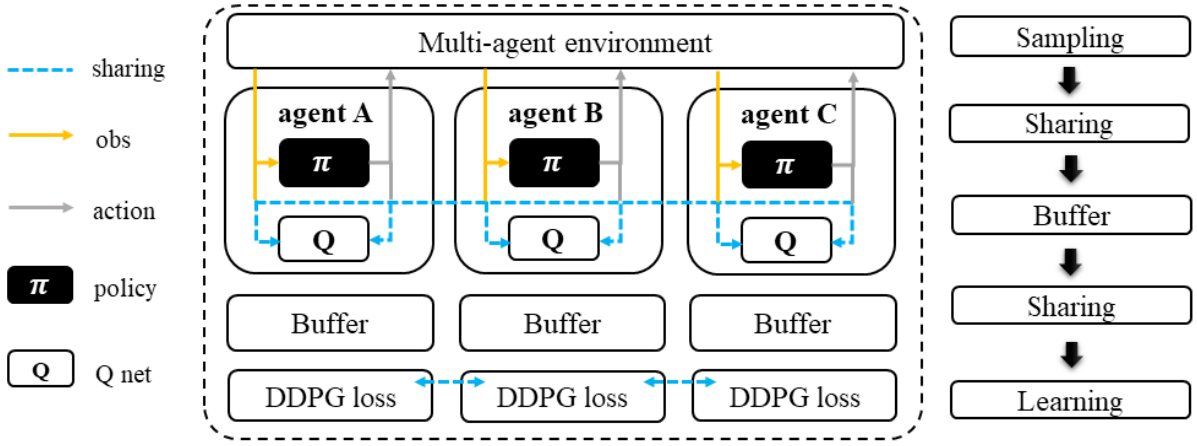


Fig. 2: Multi-agent Deep Deterministic Policy Gradient (MADDPG)

10.3.2 Characteristic

action space

`continuous`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

`off-policy`	`deterministic`

10.3.3 Insights

In multi-agent environments, traditional reinforcement learning algorithms like Q-Learning or policy gradient may not be effective due to various reasons such as changing policies during training, non-stationary environments, and high variance in coordination between agents. To address these challenges, Multi-agent Deep Deterministic Policy Gradient (MADDPG) was introduced. MADDPG extends the DDPG algorithm with a centralized Q function that takes observation and action from all agents, including other agents. The policy network $\mu(s)$ is parameterized by θ to produce action values, while the centralized Q value is calculated as $Q(\mathbf{s}, \mu(\mathbf{s}))$ and the Q network is parameterized by ϕ. It's important to note that the local observation is denoted by o in the policy network, while the joint observation/state is denoted by $\mathbf{s}$ in the centralized Q function, which includes information about opponents.

You Should Know

- MADDPG is a widely known research work that initiated the exploration of Multi-Agent Reinforcement Learning (MARL) under the paradigm of centralized training and decentralized execution (CTDE).
- Recent studies have shown that stochastic policy gradient methods can be directly employed in MARL with good performance, as exemplified by *IPPO: multi-agent version of PPO*
- However, MADDPG has been subject to criticism due to its unstable performance in practical scenarios.

10.3.4 Mathematical Form

In MADDPG, information sharing between agents is required. This includes the observation and actions of other agents, in addition to the agent's own observations. The shared information can be denoted using bold symbols, such as $\mathbf{u}$ to represent the actions of all agents.

Q learning: get a better centralized Q function

$$L(\phi, \mathcal{D}) = \underset{(o,s,\mathbf{u},r,o',s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(o, s, \mathbf{u}, r, o', s', d) - \left(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(o', s', \mu_{\theta_{\text{targ}}}(\mathbf{o}'))\right) \right)^2 \right]$$

Policy learning: maximize the Q function output by updating the policy function.

$$\max_\theta \underset{\mathbf{o},s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(o, s, \mu_\theta(\mathbf{o})) \right]$$

Here $\mathcal{D}$ is the replay buffer and can be shared across agents. $\mathbf{u}$ is an action set, including opponents. r is the reward. s is the observation/state set, including opponents. s' is the next observation/state set, including opponents. d is set to 1(True) when an episode ends else 0(False). γ is discount value. μ_θ is a policy function that can be shared across agents. Q_ϕ is Q function, which can be shared across agents. $\mu_{\theta_{\text{targ}}}$ is target policy function, which can be shared across agents. $Q_{\phi_{\text{targ}}}$ is target Q function, which can be shared across agents.

10.3.5 Implementation

To make DDPG compatible with recurrent neural networks (RNNs), we extend vanilla DDPG in RLlib. Then, we add a centralized sampling and training module to the original pipeline.

The main differences between Independent Deep Deterministic Policy Gradient (IDDPG) and MADDPG are:

- **model side: the agent model-related modules and functions are built in a centralized style:**
  - `build_maddpg_models_and_action_dist`
  - `MADDPG_RNN_TorchModel`

- **algorithm side: the sampling and training pipelines are built in a centralized style:**
  - `centralized_critic_q`
  - `central_critic_ddpg_loss`

Key hyperparameter location:

- `marl/algos/hyperparams/common/maddpg`
- `marl/algos/hyperparams/fintuned/env/maddpg`

You Should Know

- The policy inference procedure of MADDPG is kept the same as IDDPG.
- Some tricks like *gumble softmax* enables MADDPG to output categorical-like action distribution.

10.4 FACMAC: mixing a bunch of DDPG agents

Quick Facts

- Factored Multi-Agent Centralised Policy Gradients (FACMAC) is one of the extended version of *IDDPG: multi-agent version of DDPG*.
- Agent architecture of FACMAC consists of three models: `policy`, `Q`, and `mixer`.
- FACMAC is applicable for cooperative and collaborative task modes.

Preliminary:

- *IDDPG: multi-agent version of DDPG*
- *QMIX: mixing Q with monotonic factorization*

10.4.1 Workflow

Each agent in the FACMAC algorithm follows the standard DDPG learning pipeline to infer the action and calculate the Q value using the centralized Q function. However, in this stage, agents share data such as observations or states with each other before sending the sampled data to the buffer.

In the learning stage, each agent predicts its own Q value using the Q function, its next action using the target policy, and the next Q value using the target Q function. Then, each agent shares the predicted data with other agents before entering the training loop.

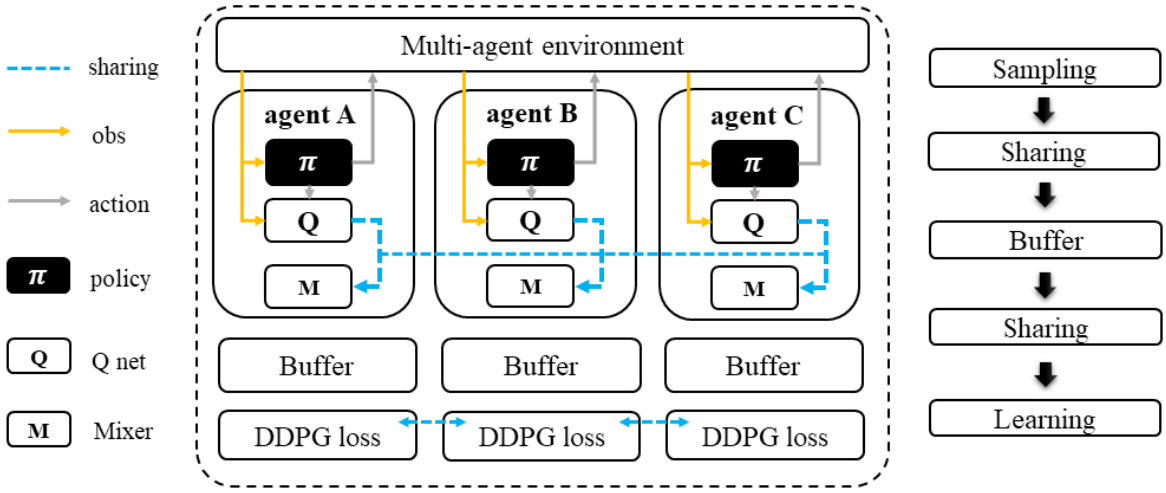


Fig. 3: Factored Multi-Agent Centralised Policy Gradients (FACMAC)

10.4.2 Characteristic

action space

`continuous`

task mode

`cooperative`	`collaborative`

taxonomy label

`off-policy`	`deterministic`	`value decomposition`

10.4.3 Insights

FACMAC is a variant of *IDDPG: multi-agent version of DDPG* in the value decomposition method and a counterpart of *MADDPG: DDPG agent with a centralized Q*. The main contribution of FACMAC is:

1. MARL's first value decomposition method can deal with a continuous control problem.
2. Proposed with a multi-agent benchmark *MAMuJoCo* that focuses on continuous control with heterogeneous agents.
3. It can also be applied to discrete action space with tricks like *gumble softmax* and keep robust performance

Compared to existing methods, FACMAC:

- outperforms MADDPG and other baselines in both discrete and continuous action tasks.
- scales better as the number of agents (and/or actions) and the complexity of the task increases.
- proves that factoring the critic can better take advantage of our centralized gradient estimator to optimize the agent policies when the number of agents and/or actions is large.

You Should Know

- Recent studies have shown that stochastic policy gradient methods are more stable and effective in handling MARL problems, such as the MAA2C algorithm mentioned in :ref:MAA2C. If you aim for improved performance, it is recommended to consider using stochastic policy gradient methods.
- The scope of applicable scenarios for FACMAC is relatively limited. This includes only cooperative tasks and continuous tasks without the use of any additional techniques or methods.

10.4.4 Mathematical Form

FAMAC needs information sharing across agents. Here we bold the symbol (e.g., u to $\mathbf{u}$) to indicate more than one agent information is contained.

Q mixing: using a learnable mixer to compute the global Q value.

$$Q_{tot}(\mathbf{u}, s; \boldsymbol{\phi}, \psi) = g_{\psi}\big(s, Q_{\phi_1}, Q_{\phi_2}, .., Q_{\phi_n}\big)$$

Q learning: get a better Q function and mixer function

$$L(\phi, \psi, \mathcal{D}) = \underset{(o,s,\mathbf{u},r,o's',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_{tot}(\mathbf{u}, o, s; \boldsymbol{\phi}, \psi) - (r + \gamma(1-d) Q_{tot}(\mathbf{u}', o', s'; \boldsymbol{\phi}_{\text{targ}}, \psi_{\text{targ}})) \right)^2 \right]$$

Policy learning: maximize the Q function output by updating the policy function.

$$\max_{\theta} \underset{o\sim\mathcal{D}}{\mathrm{E}} \left[Q_{\phi}(o, \mu_{\theta}(o)) \right]$$

Here $\mathcal{D}$ is the replay buffer, which can be shared across agents. $\mathbf{u}$ is an action set, including opponents. r is the reward. s is the observation/state set, including opponents. s' is the next observation/state set, including opponents. d is set to 1(True) when an episode ends else 0(False). γ is discount value. μ_{θ} is policy function, which can be shared across agents. Q_{ϕ} is Q function, which can be shared across agents. g_{ψ} is mixing network. $\mu_{\theta_{\text{targ}}}$ is target policy function, which can be shared across agents. $Q_{\phi_{\text{targ}}}$ is target Q function, which can be shared across agents. $g_{\psi_{\text{targ}}}$ is target mixing network.

10.4.5 Implementation

We have extended the vanilla DDPG algorithm of RLlib to be compatible with recurrent neural networks (RNNs). This RNN-compatible DDPG algorithm is further enhanced with a centralized sampling and training module. The main distinguishing features between IDDPG and MADDPG are as follows:

- **model side: the agent model-related modules and functions are built in a value decomposition style:**
  - `build_facmac_models_and_action_dist`
  - `FACMAC_RNN_TorchModel`
- **algorithm side: the sampling and training pipelines are built in a value decomposition style:**
  - `q_value_mixing`
  - `value_mixing_ddpg_loss`

Key hyperparameter location:

- `marl/algos/hyperparams/common/maddpg`

- `marl/algos/hyperparams/fintuned/env/maddpg`

You Should Know

- The policy inference procedure of FACMAC is kept the same as IDDPG.
- Tricks such as using *gumbel softmax* are employed to allow the FACMAC network to produce action distributions that resemble categorical distributions.

10.5 Read List

- Continuous Control with Deep Reinforcement Learning
- Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments
- FACMAC: Factored Multi-Agent Centralised Policy Gradients

CHAPTER

ELEVEN

ADVANCED ACTOR CRITIC FAMILY

11.1 Advanced Actor-Critic: A Recap

Preliminary:

- Vanilla Policy Gradient (PG)
- Monte Carlo Policy Gradients (REINFORCE)

You might be wondering why we need an advanced actor-critic (A2C) algorithm when we already have policy gradient method variants like REINFORCE. Well, the thing is, these methods are not always stable during training due to the large variance in the reward signal used to update the policy. To reduce this variance, we can introduce a baseline. A2C tackles this issue by using a critic value function, which is conditioned on the state, as the baseline. The difference between the Q value and the state value is then calculated as the advantage.

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

Now we need two functions Q and V to estimate A. Luckily we can do some transformations for the above equation. Recall the bellman optimality equation:

$$Q(s_t, a_t) = r_{t+1} + \lambda V(s_{t+1})$$

A can be written as:

$$A_t = r_{t+1} + \lambda V(s_{t+1}) - V(s_t)$$

In this way, only V is needed to estimate A Finally we use policy gradient to update the V function by:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A_t$$

The only thing left is how to update the parameters of the critic function:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

Here V is the critic function. ϕ is the parameters of the critic function. $\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory.

11.2 IA2C: multi-agent version of A2C

Quick Facts

- Independent advanced actor-critic (IA2C) is a natural extension of standard advanced actor-critic (A2C) in multi-agent settings.
- Agent architecture of IA2C consists of two modules: `policy` and `critic`.
- IA2C is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary:

- *Advanced Actor-Critic: A Recap*

11.2.1 Workflow

The IA2C algorithm employs a standard A2C sampling and training pipeline for each of its agents, making it a reliable baseline for all multi-agent reinforcement learning (MARL) tasks with consistent performance. It's worth noting that the buffer and agent models can be either shared or separately trained among agents, which is a feature that applies to all algorithms within the A2C family.

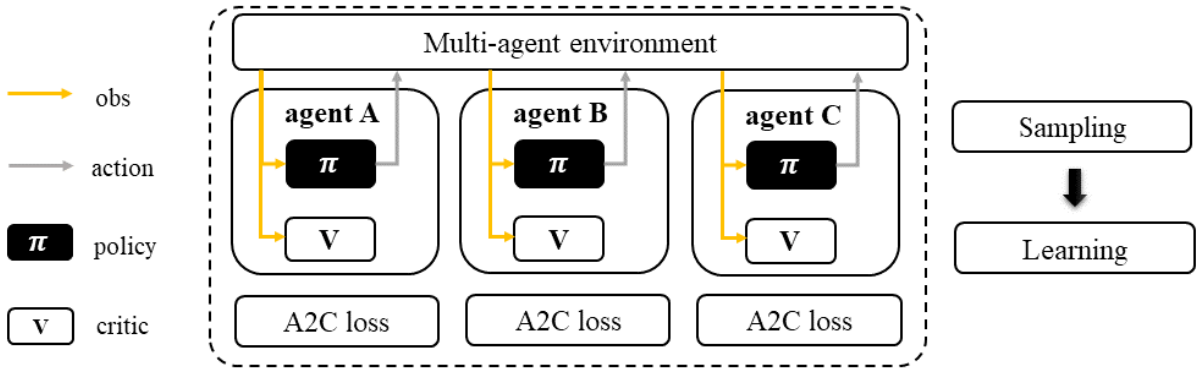


Fig. 1: Independent Advanced Actor-Critic (IA2C)

11.2.2 Characteristic

action space

`discrete`	`continuous`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

`on-policy`	`stochastic`	`independent learning`

11.2.3 Insights

IA2C is a straightforward adaptation of the standard A2C algorithm to multi-agent scenarios, where each agent acts as an A2C-based sampler and learner. Unlike some other multi-agent algorithms, IA2C does not require information sharing among agents to function effectively. However, the option to share knowledge among agents is available in IA2C.

Information Sharing

In the field of multi-agent learning, the term "information sharing" can be vague and unclear, so it's important to provide clarification. We can categorize information sharing into three types:

- real/sampled data: observation, action, etc.
- predicted data: Q/critic value, message for communication, etc.

- knowledge: experience replay buffer, model parameters, etc.

Traditionally, knowledge-level information sharing has been viewed as a "trick" and not considered a true form of information sharing in multi-agent learning. However, recent research has shown that knowledge sharing is actually crucial for achieving optimal performance. Therefore, we now consider knowledge sharing to be a valid form of information sharing in multi-agent learning.

11.2.4 Mathematical Form

When considering a single agent's perspective, the mathematical formulation of IA2C is similar to that of A2C, with the exception that in multi-agent reinforcement learning (MARL), agents often do not have access to the global state, especially under partial observable settings. In this case, we use :math:o to represent the local observation of the agent and :math:s to represent the global state. The mathematical formulation of IA2C can be rewritten as follows to account for this:

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(o_t) - \hat{R}_t \right)^2$$

Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = r_{t+1} + \lambda V_\phi(o_{t+1}) - V_\phi(o_t)$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(u_t|o_t) A_t$$

Note that in multi-agent settings, all the agent models can be shared, including:

- value function V_ϕ.
- policy function π_θ.

11.2.5 Implementation

We use vanilla A2C implementation of RLlib in IA2C.

Key hyperparameter location:

- `marl/algos/hyperparams/common/a2c`
- `marl/algos/hyperparams/fintuned/env/a2c`

11.3 MAA2C: A2C agent with a centralized critic

Quick Facts

- Multi-agent advanced actor-critic (MAA2C) is one of the extended versions of *IA2C: multi-agent version of A2C*.
- Agent architecture of MAA2C consists of two models: `policy` and `critic`.
- MAA2C is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary:

- *IA2C: multi-agent version of A2C*

11.3.1 Workflow

In the sampling stage, agents share information with others. The information includes others' observations and predicted actions. After collecting the necessary information from other agents, all agents follow the standard A2C training pipeline, except using the centralized critic value function to calculate the GAE and conduct the A2C critic learning procedure.

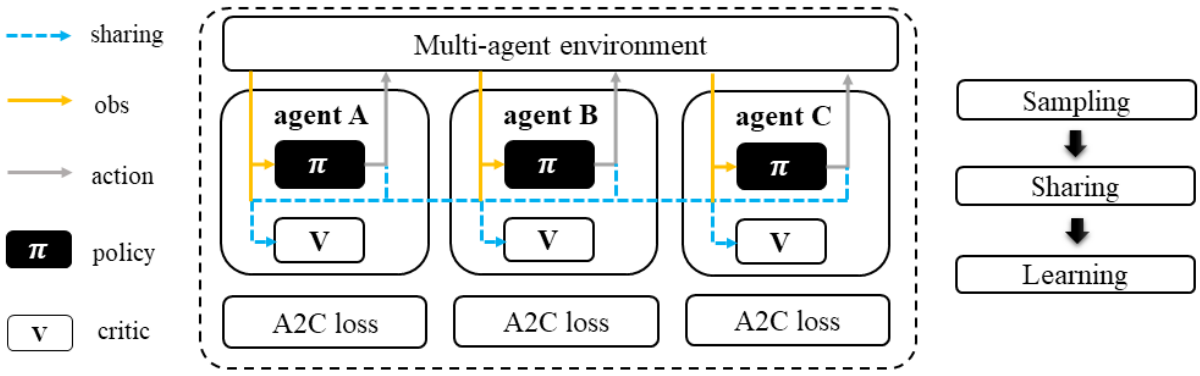


Fig. 2: Multi-agent Advanced Actor-Critic (MAA2C)

11.3.2 Characteristic

action space

`discrete`	`continuous`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

`on-policy`	`stochastic`	`centralized critic`

11.3.3 Insights

The use of a centralized critic has been shown to significantly improve the performance of multi-agent proximal policy optimization (MAPPO) in multi-agent reinforcement learning (MARL). This same architecture can also be applied to IA2C with similar success. Additionally, MAA2C can perform well in most scenarios, even without the use of a centralized critic. While there is no official MAA2C paper, we have implemented MAA2C in the same pipeline as MAPPO, using an advanced actor-critic loss function. Our implementation has shown promising results in various MARL tasks.

11.3.4 Mathematical Form

MAA2C needs information sharing across agents. Critic learning utilizes self-observation and global information, including state and actions. Here we bold the symbol (e.g., u to $\mathbf{u}$) to indicate that more than one agent information is contained.

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(o_t, s_t, \mathbf{u_t^-}) - \hat{R}_t \right)^2$$

Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = r_{t+1} + \lambda V_\phi(o_{t+1}, s_{t+1}, \mathbf{u_{t+1}^-}) - V_\phi(o_t, s_t, \mathbf{u_t^-})$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(u_t|o_t) A_t$$

Here $\mathcal{D}$ is the collected trajectories that can be shared across agents. R is the rewards-to-go. τ is the trajectory. A is the advantage. γ is discount value. λ is the weight value of GAE. o is the current agent local observation. u is the current agent action. $\mathbf{u}^-$ is the action set of all agents, except the current agent. s is the current agent global state. V_ϕ is the critic value function, which can be shared across agents. π_θ is the policy function, which can be shared across agents.

11.3.5 Implementation

Based on IA2C, we add centralized modules to implement MAA2C. The details can be found in:

- `centralized_critic_postprocessing`
- `central_critic_a2c_loss`
- `CC_RNN`

Key hyperparameter location:

- `marl/algos/hyperparams/common/maa2c`
- `marl/algos/hyperparams/fintuned/env/maa2c`

11.4 COMA: MAA2C with Counterfactual Multi-Agent Policy Gradients

Quick Facts

- Counterfactual multi-agent policy gradients (COMA) is based on MAA2C.
- Agent architecture of COMA consists of two models: `policy` and `Q`.
- COMA adopts a counterfactual baseline to marginalize a single agent's action's contribution.
- COMA is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary:

- *IA2C: multi-agent version of A2C*
- *MAA2C: A2C agent with a centralized critic*

11.4.1 Workflow

In the sampling stage, agents share information with each other, including their observations and predicted actions. Once the necessary information has been collected, all agents follow the standard A2C training pipeline. However, in order to update the policy, agents use the counterfactual multi-agent (COMA) loss function. Similar to MAA2C, the value function (critic) is centralized. This centralized critic enables agents to effectively learn from the collective experience of all agents, leading to improved performance in MARL tasks.

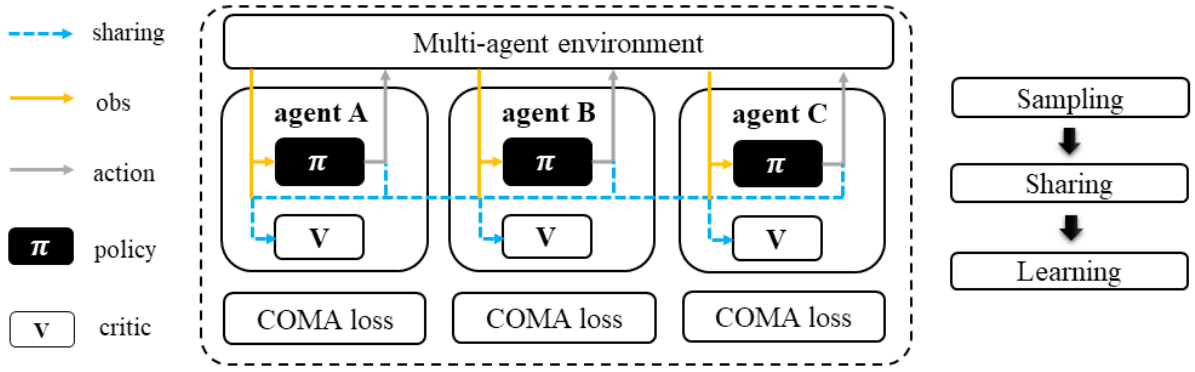


Fig. 3: Counterfactual Multi-Agent Policy Gradients (COMA)

11.4.2 Characteristic

action space

`discrete`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

on-policy	stochastic	centralized critic

11.4.3 Insights

Efficiently learning decentralized policies is an essential demand for modern AI systems. However, assigning credit to an agent becomes a significant challenge when only one global reward exists. COMA provides one solution for this problem:

1. COMA uses a counterfactual baseline that considers the actions of all agents except for the one whose credit is being assigned, making the computation of the credit assignment more effective.
2. COMA also utilizes a centralized Q value function, allowing for the efficient computation of the counterfactual baseline in a single forward pass.
3. By incorporating these techniques, COMA significantly improves average performance compared to other multi-agent actor-critic methods under decentralized execution and partial observability settings.

You Should Know

- While COMA is based on stochastic policy gradient methods, it has only been evaluated in the context of discrete action spaces. Extending this method to continuous action spaces can be challenging and may require additional techniques to compute the critic value. This is because continuous action spaces involve a potentially infinite number of actions, making it difficult to compute the critic value in a tractable manner.
- While COMA has shown promising results in improving average performance over other multi-agent actor-critic methods under decentralized execution and partial observability settings, it is worth noting that it may not always outperform other MARL methods in all tasks. It is important to carefully consider the specific task and setting when selecting an appropriate MARL method for a particular application.

11.4.4 Mathematical Form

COMA requires information sharing across agents. In particular, it uses Q learning which utilizes both self-observation and global information, including state and actions of other agents.

One unique feature of COMA is its use of a counterfactual baseline for advantage estimation, which is different from other algorithms in the A2C family. This allows for more accurate credit assignment to individual agents, even when there is only one global reward.

Q learning: every iteration gives a better Q function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(Q_\phi(o_t, s_t, u_t, (\mathbf{u_t}^-)) - \hat{R}_t \right)^2$$

Marginalized Advantage Estimation: how good are current action's Q value compared to the average Q value of the whole action space.

$$A_t = Q_\phi(o_t, s_t, u_t, \mathbf{a}^-) - \sum_{u_t} \pi(u_t|\tau) Q_\phi(o_t, s_t, u_t, (\mathbf{u_t}^-))$$

Policy learning:

$$L(o, s, a, \mathbf{a}^-, \theta) = \log \pi_\theta(a|s) A((o, s, a, \mathbf{a}^-)$$

Here $\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. A is the advantage. o is the current agent local observation. u is the current agent action. $\mathbf{u}^{-}$ is the action set of all agents, except the current agent. s is the global state. Q_ϕ is the Q function. π_θ is the policy function.

11.4.5 Implementation

Based on IA2C, we add the COMA loss function. The details can be found in:

- `centralized_critic_postprocessing`
- `central_critic_coma_loss`
- `CC_RNN`

Key hyperparameter location:

- `marl/algos/hyperparams/common/coma`
- `marl/algos/hyperparams/fintuned/env/coma`

11.5 VDA2C: mixing a bunch of A2C agents' critics

Quick Facts

- Value decomposition advanced actor-critic (VDA2C) is one of the extensions of *IA2C: multi-agent version of A2C*.
- Agent architecture of VDA2C consists of three modules: `policy`, `critic`, and `mixer`.
- VDA2C is proposed to solve cooperative and collaborative tasks only.

Preliminary:

- *IA2C: multi-agent version of A2C*
- *QMIX: mixing Q with monotonic factorization*

11.5.1 Workflow

During the sampling stage, agents exchange information with other agents, including their observations and predicted critic values. After gathering the required information, all agents follow the usual A2C training pipeline, with the exception of using a mixed critic value to calculate the Generalized Advantage Estimation (GAE) and perform the critic learning procedure for A2C.

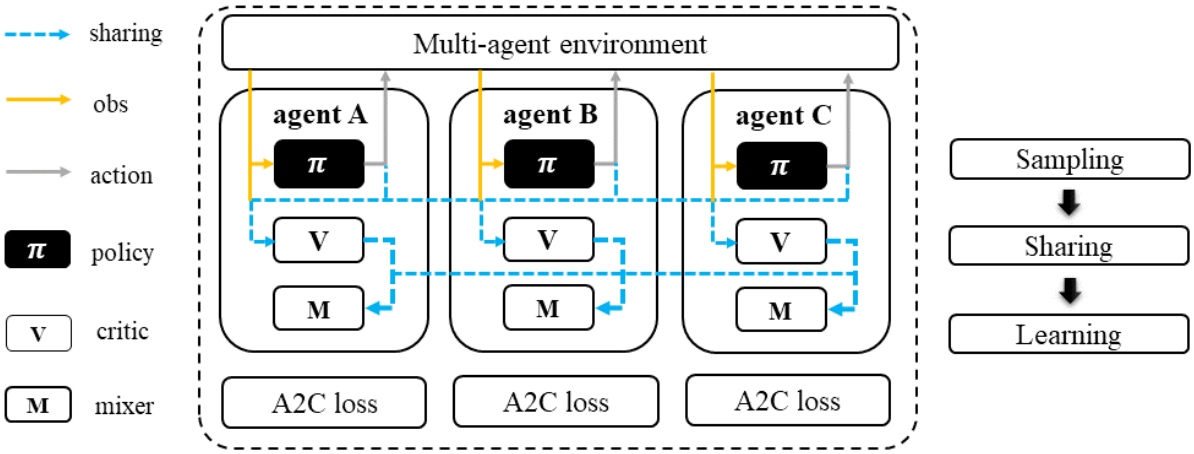


Fig. 4: Value Decomposition Advanced Actor-Critic (VDA2C)

11.5.2 Characteristic

action space

`discrete`	`continuous`

task mode

`cooperative`	`collaborative`

taxonomy label

`on-policy`	`stochastic`	`value decomposition`

11.5.3 Insights

To put it simply, VDA2C is an algorithm that focuses on assigning credit to different actions in multi-agent settings. It does this by using a value function, called the V function, which is different from the Q function used in other similar algorithms. VDA2C uses on-policy learning and is applicable to both discrete and continuous control problems. However, it may not be as efficient as other joint Q learning algorithms in terms of sampling.

11.5.4 Mathematical Form

VDA2C needs information sharing across agents. Therefore, the critic mixing utilizes both self-observation and other agents' observation. Here we bold the symbol (e.g., u to $\mathbf{u}$) to indicate that more than one agent information is contained.

Critic mixing:

$$V_{tot}(\mathbf{u}, s; \boldsymbol{\phi}, \psi) = g_{\psi}\left(s, V_{\phi_1}, V_{\phi_2}, .., V_{\phi_n}\right)$$

Mixed Critic learning: every iteration gives a better value function and a better mixing function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_{tot} - \hat{R}_t\right)^2$$

Advantage Estimation: how good are current joint action set regarding to the baseline critic value.

$$A_t = r_{t+1} + \lambda V_{tot}^{t+1} - V_{tot}^{t}$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(u_t|s_t) A_t$$

Here $\mathcal{D}$ is the collected trajectories that can be shared across agents. R is the rewards-to-go. τ is the trajectory. A is the advantage. γ is discount value. λ is the weight value of GAE. o is the current agent local observation. u is the current agent action. $\mathbf{u}^-$ is the action set of all agents, except the current agent. s is the current agent global state. V_ϕ is the critic value function, which can be shared across agents. π_θ is the policy function, which can be shared across agents. g_ψ is a mixing network, which must be shared across agents.

11.5.5 Implementation

Based on IA2C, we add mixing Q modules to implement VDA2C. The details can be found in:

- `value_mixing_postprocessing`
- `value_mix_actor_critic_loss`
- `VD_RNN`

Key hyperparameter location:

- `marl/algos/hyperparams/common/vda2c`
- `marl/algos/hyperparams/fintuned/env/vda2c`

11.6 Read List

- Advanced Actor-Critic Algorithms
- The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games
- Counterfactual Multi-Agent Policy Gradients
- Value-Decomposition Multi-Agent Actor-Critics

CHAPTER

TWELVE

TRUST REGION POLICY OPTIMIZATION FAMILY

12.1 Trust Region Policy Optimization: A Recap

Preliminary:

- Vanilla Policy Gradient (PG)

In reinforcement learning, finding the appropriate learning rate is essential for policy-gradient based methods. To address this issue, Trust Region Policy Optimization (TRPO) proposes that the updated policy should remain within a trust region. TRPO has four steps to optimize its policy function:

1. sample a set of trajectories.
2. estimate the advantages using any advantage estimation method (here we adopt General Advantage Estimation (GAE)).
3. solve the constrained optimization problem using conjugate gradient and update the policy by it.
4. fit the value function (critic).

And we repeat these steps to get the global maximum point of the policy function.

Mathematical Form

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

General Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

: computing the policy gradient using estimated advantage to update the policy function.

Policy learning step 1: estimate policy gradient

$$g_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} A_t$$

Policy learning step 2: Use the conjugate gradient algorithm to compute

$$x_k \approx H_k^{-1} g_k$$

Policy learning step 3

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{x_k^T H_k x_k}} x_k$$

Here $\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. V_ϕ is the critic function. A is the advantage. γ is discount value. λ is the weight value of GAE. a is the action. s is the observation/state. ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. π_θ is the policy function. H_k is the Hessian of the sample average KL-divergence. $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.

A more detailed explanation can be found in - Spinning Up: Trust Region Policy Optimisation

12.2 ITRPO: multi-agent version of TRPO

Quick Facts

- Independent trust region policy optimization (ITRPO) is a natural extension of standard trust region policy optimization (TRPO) in multi-agent settings.
- Agent architecture of ITRPO consists of two modules: `policy` and `critic`.
- ITRPO is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary:

- *Trust Region Policy Optimization: A Recap*

12.2.1 Workflow

In ITRPO, each agent follows a standard TRPO sampling/training pipeline. Note that buffer and agent models can be shared or separately training across agents. And this applies to all algorithms in TRPO family.

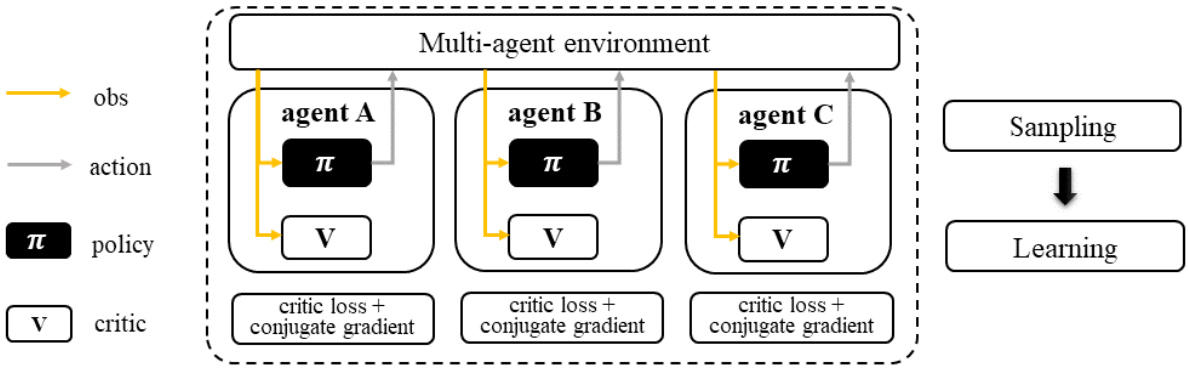


Fig. 1: Independent Trust Region Policy Optimization (ITRPO)

12.2.2 Characteristic

action space

`discrete`	`continuous`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

`on-policy`	`stochastic`	`independent learning`

12.2.3 Insights

ITRPO is a multi-agent version of TRPO, where each agent is a TRPO-based sampler and learner. ITRPO does not require information sharing between agents during training. However, knowledge sharing between agents is optional and can be implemented if desired.

Information Sharing

In the field of multi-agent learning, the term "information sharing" can be vague and unclear, so it's important to provide clarification. We can categorize information sharing into three types:

- real/sampled data: observation, action, etc.
- predicted data: Q/critic value, message for communication, etc.
- knowledge: experience replay buffer, model parameters, etc.

Traditionally, knowledge-level information sharing has been viewed as a "trick" and not considered a true form of information sharing in multi-agent learning. However, recent research has shown that knowledge sharing is actually crucial for achieving optimal performance. Therefore, we now consider knowledge sharing to be a valid form of information sharing in multi-agent learning.

12.2.4 Mathematical Form

Standing at the view of a single agent, the mathematical formulation of ITRPO is similiar as *Trust Region Policy Optimization: A Recap*, except that in MARL, agent usually has no access to the global state typically under partial observable setting. Therefore, we use o for local observation and :math:`s`for the global state. We then rewrite the mathematical formulation of TRPO as:

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(o_t) - \hat{R}_t \right)^2$$

General Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Policy learning step 1: estimate policy gradient

$$g_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(u_t|o_t)|_{\theta_k} A_t$$

Policy learning step 2 & 3 are the same as *Trust Region Policy Optimization: A Recap*.

$\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. V_ϕ is the critic function. A is the advantage. γ is discount value. λ is the weight value of GAE. u is the action. o is the local observation. ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. π_θ is the policy function.

Note that in multi-agent settings, all the agent models can be shared, including:

- critic function V_ϕ.
- policy function π_θ.

12.2.5 Implementation

We implement TRPO based on PPO training pipeline of RLlib. The detail can be found in:

- `TRPOTorchPolicy`
- `TRPOTrainer`

Key hyperparameter location:

- `marl/algos/hyperparams/common/trpo`
- `marl/algos/hyperparams/fintuned/env/trpo`

12.3 MATRPO: TRPO agent with a centralized critic

Quick Facts

- Multi-agent trust region policy optimization (MATRPO) is one of the extended version of *ITRPO: multi-agent version of TRPO*.
- Agent architecture of MATRPO consists of two models: `policy` and `critic`.
- MATRPO is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary:

- *ITRPO: multi-agent version of TRPO*

12.3.1 Workflow

During the sampling stage of MATRPO, agents share information such as observations and predicted actions with each other. Once each agent collects the necessary information from the others, they can begin the standard TRPO training pipeline. The only difference is that a centralized value function is used to calculate the Generalized Advantage Estimation (GAE) and conduct the TRPO policy learning and critic learning procedure. This allows the agents to take into account the actions and observations of their teammates when updating their policies.

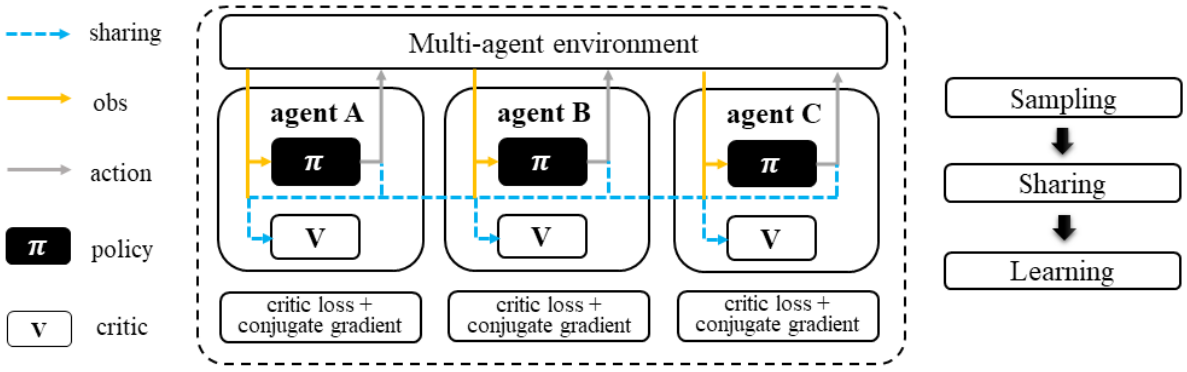


Fig. 2: Multi-agent Trust Region Policy Optimization (MATRPO)

12.3.2 Characteristic

action space

discrete	continuous

task mode

cooperative	collaborative	competitive	mixed

taxonomy label

on-policy	stochastic	centralized critic

12.3.3 Insights

MATRPO and *MAPPO: PPO agent with a centralized critic* are very alike of their features, only the decentralized policy is optimized in the TRPO manner in MATRPO instead of PPO manner.

12.3.4 Mathematical Form

MATRPO needs information sharing across agents. Critic learning utilizes self-observation and information other agents provide, including observation and actions. Here we bold the symbol (e.g., u to $\mathbf{u}$) to indicate more than one agent information is contained.

Critic learning: every iteration gives a better centralized value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(o_t, s_t, \mathbf{u_t}^-) - \hat{R}_t \right)^2$$

General Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Policy learning step 1: estimate policy gradient

$$g_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(u_t|o_t)|_{\theta_k} A_t$$

Policy learning step 2 & 3 are the same as *Trust Region Policy Optimization: A Recap*.

Here $\mathcal{D}$ is the collected trajectories that can be shared across agents. R is the rewards-to-go. τ is the trajectory. A is the advantage. γ is discount value. λ is the weight value of GAE. u is the current agent action. $\mathbf{u}^-$ is the action set of all agents, except the current agent. s is the global state. o is the local observation ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. V_ϕ is the value function, which can be shared across agents. π_θ is the policy function, which can be shared across agents.

12.3.5 Implementation

Based on ITRPO, we add centralized modules to implement MATRPO. The details can be found in:

- `centralized_critic_postprocessing`
- `centre_critic_trpo_loss_fn`
- `CC_RNN`

Key hyperparameter location:

- `marl/algos/hyperparams/common/matrpo`
- `marl/algos/hyperparams/fintuned/env/matrpo`

12.4 HATRPO: Sequentially updating critic of MATRPO agents

Quick Facts

- Heterogeneous-Agent Trust Region Policy Optimisation (HATRPO) algorithm is based on *MATRPO: TRPO agent with a centralized critic*.
- Agent architecture of HATRPO consists of three modules: `policy`, `critic`, and `sequential updating`.
- In HATRPO, agents have non-shared `policy` and shared `critic`.
- HATRPO is proposed to solve cooperative and collaborative tasks.

12.4.1 Workflow

HATRPO is a variant of TRPO in which each agent still shares information with others during the sampling stage, but the policies are updated sequentially rather than simultaneously. In the updating sequence, the next agent's advantage is computed using the current sampling importance and the former advantage, except for the first agent, whose advantage is the original advantage value.

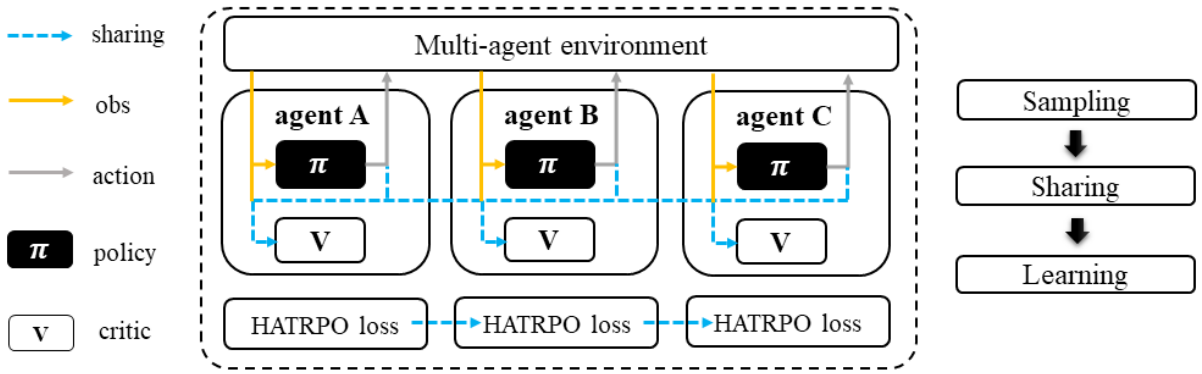


Fig. 3: Heterogeneous-Agent Trust Region Policy Optimization (HATRPO)

12.4.2 Characteristic

action space

discrete	continuous

task mode

cooperative	collaborative

taxonomy label

on-policy	stochastic	centralized critic

12.4.3 Insights

Preliminary

- *ITRPO: multi-agent version of TRPO*

The previous methods either hold the sharing parameters for different agents or lack the essential theoretical property of trust region learning, which is the monotonic improvement guarantee. This could lead to several issues when dealing with MARL problems. Such as:

1. If the parameters have to be shared, the methods could not apply to the occasions that different agents observe different dimensions.
2. Sharing parameters could suffer from an exponentially-worse suboptimal outcome.
3. although ITRPO/MATRPO can be practically applied in a non-parameter sharing way, it still lacks the essential theoretical property of trust region learning, which is the monotonic improvement guarantee.

The HATRPO paper proves that for Heterogeneous-Agent:

1. Theoretically-justified trust region learning framework in MARL.
2. HATRPO adopts the sequential update scheme, which saves the cost of maintaining a centralized critic for each agent in CTDE(centralized training with decentralized execution).

Some Interesting Facts

- A similar idea of the multi-agent sequential update was also discussed in dynamic programming, where artificial "in-between" states must be considered. On the contrary, HATRPO sequential update scheme is developed based on the paper proposed Lemma 1, which does not require any artificial assumptions and holds for any cooperative games
- Bertsekas (2019) requires maintaining a fixed order of updates that is pre-defined for the task, whereas the order in MATRPO is randomised at each iteration, which also offers desirable convergence property

---

12.4.4 Mathematical Form

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

Initial Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Advantage Estimation for m = 1: how good are current action regarding to the baseline critic value of the first chosen agent.

$$\mathbf{M}^{i_1}(s, \mathbf{u}) = \hat{A}_{s,\mathbf{u}}(s, \mathbf{u})$$

Advantage Estimation if m > 1: how good are current action regarding to the baseline critic value of the chosen agent except the first one.

$$\mathbf{M}^{i_{1:m}}(s, \mathbf{u}) = \frac{\bar{\pi}^{i_{1:m-1}}(u^{1:m-1}|o)}{\pi^{i_{1:m-1}}(u^{1:m-1}|o)} \mathbf{M}^{i_{1:m-1}}(s, \mathbf{u})$$

Estimate the gradient of the agent's maximisation objective.

$$\hat{\boldsymbol{g}}_k^{i_m} = \frac{1}{B} \sum_{b=1}^{B} \sum_{t=1}^{T} \nabla_{\theta_k^{i_m}} \log \pi_{\theta_k^{i_m}}^{i_m} \left(a_t^{i_m} \mid o_t^{i_m} \right) M^{i_{1:m}}(s_t, \boldsymbol{a}_t)$$

HessianoftheaverageKL-divergence

$$\hat{\boldsymbol{H}}_k^{i_m} = \frac{1}{BT} \sum_{b=1}^{B} \sum_{t=1}^{T} D_{\mathrm{KL}} \left(\pi_{\theta_k^{i_m}}^{i_m}\left(\cdot \mid o_t^{i_m}\right), \pi_{\theta^{i_m}}^{i_m}\left(\cdot \mid o_t^{i_m}\right) \right)$$

Use the conjugate gradient algorithm to compute the update direction.

$$\boldsymbol{x}_k^{i_m} \approx \left(\hat{\boldsymbol{H}}_k^{i_m} \right)^{-1} \boldsymbol{g}_k^{i_m}$$

Estimate the maximal step size allowing for meeting the KL-constraint

$$\hat{\beta}_k^{i_m} \approx \sqrt{\frac{2\delta}{\left(\hat{\boldsymbol{x}}_k^{i_m}\right)^T \hat{\boldsymbol{H}}_k^{i_m} \hat{\boldsymbol{x}}_k^{i_m}}}$$

Update agent 's policy by

$$\theta_{k+1}^{i_m} = \theta_k^{i_m} + \alpha^j \hat{\beta}_k^{i_m} \hat{\boldsymbol{x}}_k^{i_m}$$

Here $\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. A is the advantage. γ is discount value. λ is the weight value of GAE. u is the current agent action. $\mathbf{u}^-$ is the action set of all agents, except the current agent. s is the global state. o is the local information. ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. V_ϕ is the value function. π_θ is the policy function. B is batch size T is steps per episode $j \in 0, 1, \ldots, L$ is the smallest such which improves the sample loss by at least $\kappa\alpha^j \hat{\beta}_k^{i_m} \hat{\boldsymbol{x}}_k^{i_m} \cdot \hat{\boldsymbol{g}}_k^{i_m}$, found by the backtracking line search.

12.4.5 Implementation

Based on MATRPO, we add four components to implement HATRPO. The details can be found in:

- `hatrpo_loss_fn`
- `hatrpo_post_process`
- `TrustRegionUpdator`
- `HATRPOUpdator`

Key hyperparameter location:

- `marl/algos/hyperparams/common/hatrpo`
- `marl/algos/hyperparams/fintuned/env/hatrpo`

12.5 Read List

- Trust Region Policy Optimization Algorithms
- High-Dimensional Continuous Control Using Generalized Advantage Estimation
- Trust Region Policy Optimisation in Multi-Agent Reinforcement Learning

CHAPTER

THIRTEEN

PROXIMAL POLICY OPTIMIZATION FAMILY

13.1 Proximal Policy Optimization: A Recap

Preliminary:

- Vanilla Policy Gradient (PG)
- Trust Region Policy Optimization (TRPO)
- General Advantage Estimation (GAE)

Proximal Policy Optimization (PPO) is a simple first-order optimization algorithm for reinforcement learning. It is similar to another algorithm called Trust Region Policy Optimization (TRPO) but with a simpler implementation. The PPO algorithm defines the probability ratio between the new policy and the old policy as $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ where θ is the new policy and θ_k is the old policy. Instead of adding complicated KL constraints, PPO ensures that this policy ratio stays within a small interval between $1-\epsilon$ and $1+\epsilon$, where ϵ is a hyperparameter that controls the size of the interval. The objective function of PPO takes the minimum value between the original value and the clipped value, where the clipped value is obtained by multiplying the policy ratio by the advantage estimate and then clipping it to the interval $[1-\epsilon, 1+\epsilon]$.

There are two primary variants of PPO: PPO-Penalty and PPO-Clip. Here we only give the formulation of PPO-Clip, which is more common in practice. For PPO-penalty, please refer to Proximal Policy Optimization.

Mathematical Form

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

General Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right)$$

Here $\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. V_ϕ is the critic function. A is the advantage. γ is discount value. λ is the weight value of GAE. a is the action. s is the observation/state. ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. π_θ is the policy function.

13.2 IPPO: multi-agent version of PPO

Quick Facts

- Independent proximal policy optimization (IPPO) is a natural extension of standard proximal policy optimization (PPO) in multi-agent settings.
- Agent architecture of IPPO consists of two modules: `policy` and `critic`.
- IPPO is applicable for cooperative, collaborative, competitive, and mixed task modes.

Preliminary:

- *Proximal Policy Optimization: A Recap*

13.2.1 Workflow

In IPPO, each agent uses the standard PPO sampling/training pipeline, making it a versatile baseline for multi-agent reinforcement learning tasks with reliable performance. It's worth noting that the buffer and agent models can either be shared or trained separately across agents. This applies to all algorithms in the PPO family, not just IPPO.

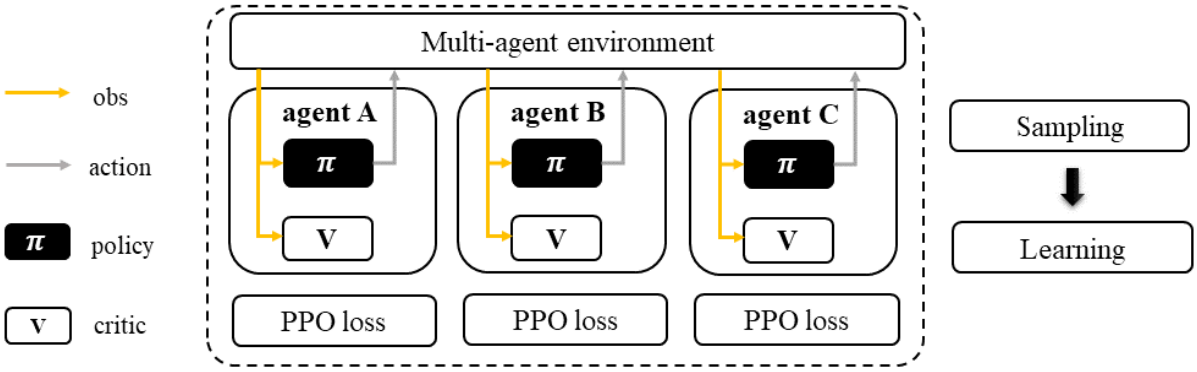


Fig. 1: Independent Proximal Policy Optimization (IPPO)

13.2.2 Characteristic

action space

`discrete`	`continuous`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

`on-policy`	`stochastic`	`independent learning`

13.2.3 Insights

In simpler terms, IPPO is a straightforward implementation of PPO for multi-agent reinforcement learning tasks. Each agent follows the same PPO sampling and training process, making it a versatile baseline for various MARL tasks. Unlike other MARL algorithms, IPPO does not require information sharing between agents, although it can still be optionally implemented for knowledge sharing.

Information Sharing

In the field of multi-agent learning, the term "information sharing" can be vague and unclear, so it's important to provide clarification. We can categorize information sharing into three types:

- real/sampled data: observation, action, etc.
- predicted data: Q/critic value, message for communication, etc.
- knowledge: experience replay buffer, model parameters, etc.

Traditionally, knowledge-level information sharing has been viewed as a "trick" and not considered a true form of information sharing in multi-agent learning. However, recent research has shown that knowledge sharing is actually crucial for achieving optimal performance. Therefore, we now consider knowledge sharing to be a valid form of information sharing in multi-agent learning.

13.2.4 Mathematical Form

Standing at the view of a single agent, the mathematical formulation of IPPO is similiar as *Proximal Policy Optimization: A Recap*, except that in MARL, agent usually has no access to the global state typically under partial observable setting. Therefore, we use o for local observation and :math:`s`for the global state. We then rewrite the mathematical formulation of PPO as:

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(o_t) - \hat{R}_t \right)^2$$

General Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$L(o, u, \theta_k, \theta) = \min \left(\frac{\pi_\theta(u|o)}{\pi_{\theta_k}(u|o)} A^{\pi_{\theta_k}}(o, u), \ \ \text{clip} \left(\frac{\pi_\theta(u|o)}{\pi_{\theta_k}(u|o)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(o, u) \right)$$

$\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. V_ϕ is the critic function. A is the advantage. γ is discount value. λ is the weight value of GAE. u is the action. o is the local observation. ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. π_θ is the policy function.

Note that in multi-agent settings, all the agent models can be shared, including:

- critic function V_ϕ.
- policy function π_θ.

13.2.5 Implementation

We have made some modifications to the IPPO algorithm, specifically to the way the stochastic gradient descent (SGD) iteration is implemented. Other than that, we use the vanilla PPO implementation provided by RLlib as the basis for IPPO. You can find more information about the modifications we made to the SGD iteration in our documentation.

- `MultiGPUTrainOneStep`
- `learn_on_loaded_batch`

Key hyperparameter location:

- `marl/algos/hyperparams/common/ppo`
- `marl/algos/hyperparams/fintuned/env/ppo`

13.3 MAPPO: PPO agent with a centralized critic

Quick Facts

- Multi-agent proximal policy optimization (MAPPO) is one of the extended version of *IPPO: multi-agent version of PPO*.
- Agent architecture of MAPPO consists of two models: `policy` and `critic`.
- MAPPO is proposed to solve cooperative tasks but is still applicable to collaborative, competitive, and mixed tasks.

Preliminary:

- *IPPO: multi-agent version of PPO*

13.3.1 Workflow

During the sampling stage in collaborative multi-agent reinforcement learning, agents need to communicate and share information with each other, such as observations and predicted actions. Once all the necessary information is collected, each agent follows the standard PPO training pipeline, but with the addition of a centralized value function for calculating the Generalized Advantage Estimation (GAE) and conducting the PPO critic learning procedure.

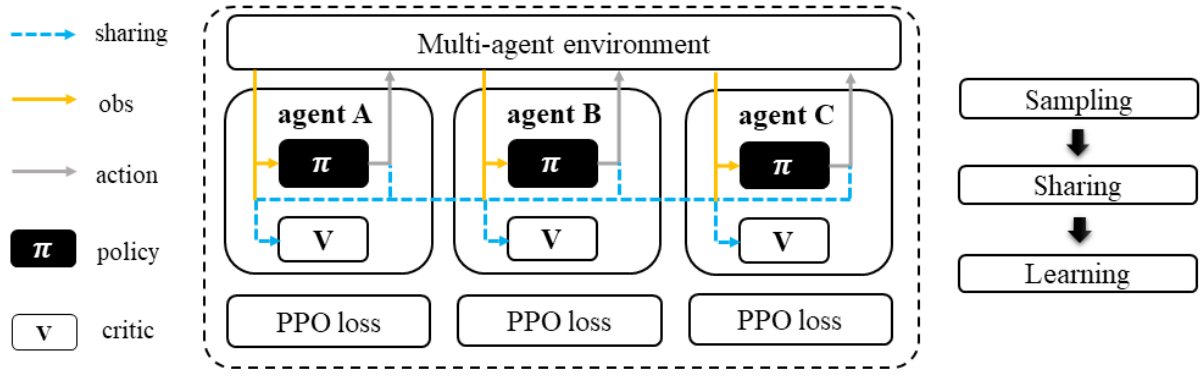


Fig. 2: Multi-agent Proximal Policy Optimization (MAPPO)

13.3.2 Characteristic

action space

`discrete`	`continuous`

task mode

`cooperative`	`collaborative`	`competitive`	`mixed`

taxonomy label

on-policy	stochastic	centralized critic

13.3.3 Insights

On-policy reinforcement learning algorithms are less sample efficient than their off-policy counterparts in MARL. The MAPPO algorithm overturn this consensus by experimentally proving that:

1. On-policy algorithms can achieve comparable performance to various off-policy methods.
2. MAPPO is a robust MARL algorithm for diverse cooperative tasks and can outperform SOTA off-policy methods in more challenging scenarios.
3. Formulating the input to the centralized value function is crucial for the final performance.

You Should Know

- MAPPO paper is done in cooperative settings. Nevertheless, it can be directly applied to competitive and mixed task modes. Moreover, the performance is still good.
- Sampling procedure of on-policy algorithms can be parallel conducted. Therefore, the actual time consuming for a comparable performance between MAPPO and off-policy algorithms is almost the same when we have enough sampling *workers*.
- Parameters are shared across agents. Not sharing these parameters will not incur any problems. Conversely, partly sharing these parameters(e.g., only sharing the critic) can help achieve better performance in some scenarios.

13.3.4 Mathematical Form

MAPPO needs information sharing across agents. Critic learning utilizes self-observation and information other agents provide, including observation and actions. Here we bold the symbol (e.g., u to $\mathbf{u}$) to indicate more than one agent information is contained.

Critic learning: every iteration gives a better centralized value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(o_t, s_t, \mathbf{u_t}^-) - \hat{R}_t \right)^2$$

General Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$L(o, s, u, \mathbf{u}^-, \theta_k, \theta) = \min \left(\frac{\pi_\theta(u|o)}{\pi_{\theta_k}(u|o)} A^{\pi_{\theta_k}}(o, s, \mathbf{u}^-), \ \ \text{clip} \left(\frac{\pi_\theta(u|o)}{\pi_{\theta_k}(u|o)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(o, s, \mathbf{u}^-) \right)$$

Here $\mathcal{D}$ is the collected trajectories that can be shared across agents. R is the rewards-to-go. τ is the trajectory. A is the advantage. γ is discount value. λ is the weight value of GAE. u is the current agent action. $\mathbf{u}^-$ is the action set of all agents, except the current agent. s is the global state. o is the local observation ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. V_ϕ is the value function, which can be shared across agents. π_θ is the policy function, which can be shared across agents.

13.3.5 Implementation

Based on IPPO, we add centralized modules to implement MAPPO. The details can be found in:

- `centralized_critic_postprocessing`
- `central_critic_ppo_loss`
- `CC_RNN`

Key hyperparameter location:

- `marl/algos/hyperparams/common/mappo`
- `marl/algos/hyperparams/fintuned/env/mappo`

13.4 VDPPO: mixing a bunch of PPO agents' critics

Quick Facts

- Value decomposition proximal policy optimization (VDPPO) is one of the extended version of *IPPO: multi-agent version of PPO*.
- Agent architecture of VDPPO consists of three modules: `policy`, `critic`, and `mixer`.
- VDPPO is proposed to solve cooperative and collaborative task modes.

Preliminary:

- *IPPO: multi-agent version of PPO*
- *QMIX: mixing Q with monotonic factorization*

13.4.1 Workflow

In the sampling stage, agents share information with others. The information includes others' observations and predicted critic value. After collecting the necessary information from other agents, all agents follow the standard PPO training pipeline, except for using the mixed critic value to calculate the GAE and conduct the PPO critic learning procedure.

13.4.2 Characteristic

action space

`discrete`	`continuous`

task mode

`cooperative`	`collaborative`

taxonomy label

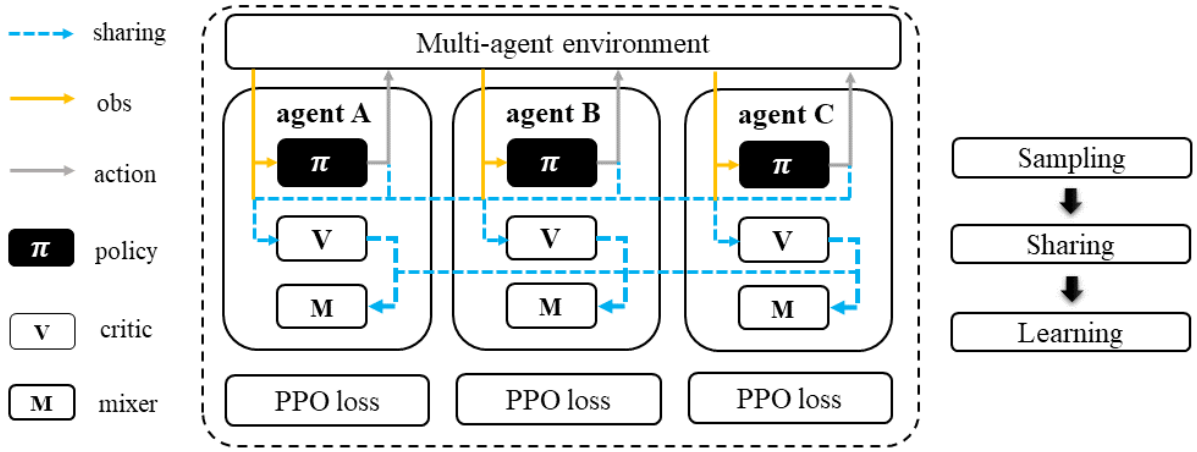


Fig. 3: Value Decomposition Proximal Policy Optimization (VDPPO)

on-policy	stochastic	value decomposition

13.4.3 Insights

VDPPO focuses on the credit assignment learning, which is similar to the joint Q learning family. VDPPO is easy to understand when you have basic idea of *QMIX: mixing Q with monotonic factorization* and *VDA2C: mixing a bunch of A2C agents' critics*.

13.4.4 Mathematical Form

VDPPO needs information sharing across agents. Therefore, the critic mixing utilizes both self-observation and other agents' observation. Here we bold the symbol (e.g., u to $\mathbf{u}$) to indicate more than one agent information is contained.

Critic mixing: a learnable mixer for computing the global value function.

$$V_{tot}(\mathbf{a}, s; \boldsymbol{\phi}, \psi) = g_\psi\left(s, V_{\phi_1}, V_{\phi_2}, .., V_{\phi_n}\right)$$

Critic learning: every iteration gives a better global value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_{tot}(\mathbf{u}, s; \boldsymbol{\phi}, \psi) - \hat{R}_t\right)^2$$

General Advantage Estimation: how good are current joint action set regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^{V_{tot}}$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$L(s, o, u, \mathbf{u}^-, \theta_k, \theta) = \min\left(\frac{\pi_\theta(u|o)}{\pi_{\theta_k}(u|o)} A^{\pi_{\theta_k}}(s, o, \mathbf{u}^-),\ \text{clip}\left(\frac{\pi_\theta(u|o)}{\pi_{\theta_k}(u|o)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, o, \mathbf{u}^-)\right)$$

Here $\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. A is the advantage. γ is discount value. λ is the weight value of GAE. u is the current agent action. $\mathbf{u}^-$ is the action set of all agents, except the current agent. s is the global state. o is the local observation. ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. V_ϕ is the value function. π_θ is the policy function. g_ψ is the mixer.

13.4.5 Implementation

Based on IPPO, we add the mixer to implement VDPPO. The details can be found in:

- `value_mixing_postprocessing`
- `value_mix_ppo_surrogate_loss`
- `VD_RNN`

Key hyperparameter location:

- `marl/algos/hyperparams/common/vdppo`
- `marl/algos/hyperparams/fintuned/env/vdppo`

13.5 HAPPO: Sequentially updating critic of MAPPO agents

Quick Facts

- Heterogeneous-Agent Proximal Policy Optimisation (HAPPO) algorithm is based on *MAPPO: PPO agent with a centralized critic*.
- Agent architecture of HAPPO consists of three modules: `policy`, `critic`, and `sequential updating`.
- In HAPPO, agents have non-shared `policy` and shared `critic`.
- HAPPO is proposed to solve cooperative and collaborative tasks.

13.5.1 Workflow

In the sampling stage, agents share information with others. The information includes others' observations and predicted actions. After collecting the necessary information from other agents, all agents follow the standard PPO training pipeline, except HAPPO would update each policy sequentially. In this updating sequence, the next agent's advantage is iterated by the current sampling importance and hte former advantage, except the first agent's advantage is the original advantae value.

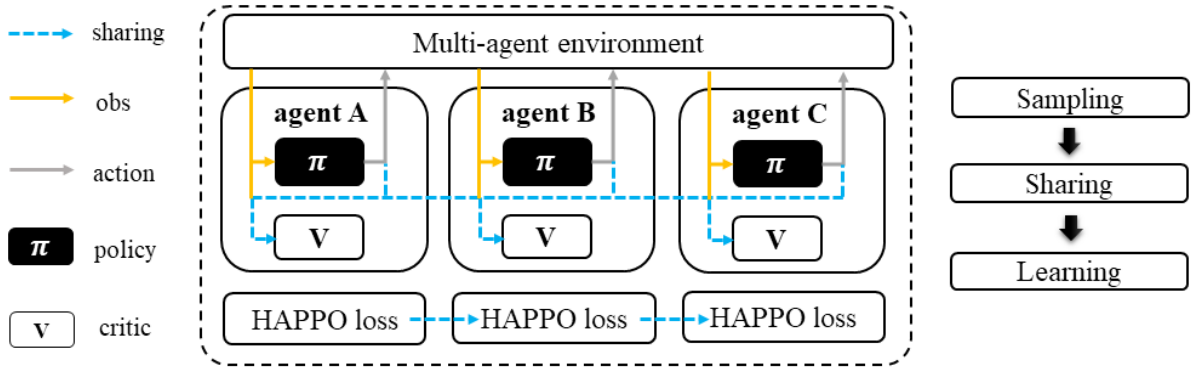


Fig. 4: Heterogeneous-Agent Proximal Policy Optimization (HAPPO)

13.5.2 Characteristic

action space

discrete	continuous

task mode

cooperative	collaborative

taxonomy label

on-policy	stochastic	centralized critic

13.5.3 Insights

Preliminary

- *IPPO: multi-agent version of PPO*

The previous methods either hold the sharing parameters for different agents or lack the essential theoretical property of trust region learning, which is the monotonic improvement guarantee. This could lead to several issues when dealing with MARL problems. Such as:

1. If the parameters have to be shared, the methods could not apply to the occasions that different agents observe different dimensions.
2. Sharing parameters could suffer from an exponentially-worse suboptimal outcome.
3. although IPPO/MAPPO can be practically applied in a non-parameter sharing way, it still lacks the essential theoretical property of trust region learning, which is the monotonic improvement guarantee.

The HAPPO paper proves that for Heterogeneous-Agent:

1. Theoretically-justified trust region learning framework in MARL.
2. HAPPO adopts the sequential update scheme, which saves the cost of maintaining a centralized critic for each agent in CTDE(centralized training with decentralized execution).

Some Interesting Facts

- A similar idea of the multi-agent sequential update was also discussed in dynamic programming, where artificial "in-between" states must be considered. On the contrary, HAPPO sequential update scheme is developed based on the paper proposed Lemma 1, which does not require any artificial assumptions and holds for any cooperative games
- Bertsekas (2019) requires maintaining a fixed order of updates that is pre-defined for the task, whereas the order in MAPPO is randomised at each iteration, which also offers desirable convergence property

---

13.5.4 Mathematical Form

Critic learning: every iteration gives a better value function.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

Initial Advantage Estimation: how good are current action regarding to the baseline critic value.

$$A_t = \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Advantage Estimation for m = 1: how good are current action regarding to the baseline critic value of the first chosen agent.

$$\mathbf{M}^{i_1}(s, \mathbf{u}) = \hat{A}_{s,\mathbf{u}}(s, \mathbf{u})$$

Advantage Estimation if m > 1: how good are current action regarding to the baseline critic value of the chosen agent except the first one.

$$\mathbf{M}^{i_{1:m}}(s, \mathbf{u}) = \frac{\bar{\pi}^{i_{1:m-1}}(u^{1:m-1}|o)}{\pi^{i_{1:m-1}}(u^{1:m-1}|o)} \mathbf{M}^{i_{1:m-1}}(s, \mathbf{u})$$

Policy learning: computing the policy gradient using estimated advantage to update the policy function.

$$\frac{1}{BT} \sum_{b=1}^{B} \sum_{t=0}^{T} \left[min \left(\frac{\pi_{\theta^{im}}^{i_m}(u^{i_m}|o)}{\pi_{\theta_k^{im}}^{i_m}(u^{i_m}|o)} M^{i_{1:m}}(s|u), clip \left(\frac{\pi_{\theta^{im}}^{i_m}(u^{i_m}|o)}{\pi_{\theta_k^{im}}^{i_m}(u^{i_m}|o)}, 1 \pm \epsilon \right) \right) M^{i_{1:m}}(s|u) \right]$$

Here $\mathcal{D}$ is the collected trajectories. R is the rewards-to-go. τ is the trajectory. A is the advantage. γ is discount value. λ is the weight value of GAE. u is the current agent action. $\mathbf{u}^-$ is the action set of all agents, except the current agent. s is the global state. o is the local information. ϵ is a hyperparameter controlling how far away the new policy is allowed to go from the old. V_ϕ is the value function. π_θ is the policy function. B is batch size T is steps per episode

13.5.5 Implementation

Based on MAPPO, we add three components to implement HAPPO. The details can be found in:

- `add_opponent_information_and_critical_vf`
- `happo_surrogate_loss`
- `add_all_agents_gae`

Key hyperparameter location:

- `marl/algos/hyperparams/common/happo`
- `marl/algos/hyperparams/fintuned/env/happo`

13.6 Read List

- High-Dimensional Continuous Control Using Generalized Advantage Estimation
- Proximal Policy Optimization Algorithms
- Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge?
- The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games
- Trust Region Policy Optimisation in Multi-Agent Reinforcement Learning

CHAPTER

FOURTEEN

AWESOME PAPER LIST

We collect most of the existing MARL algorithms based on the multi-agent environment they choose to conduct on, with tag to annotate the sub-topic.

[B] Basic [S] Information Sharing [RG] Behavior/Role Grouping [I] Imitation [G] Graph [E] Exploration [R] Robust [P] Reward Shaping [F] Offline [T] Tree Search [MT] Multi-task

14.1 MPE

- Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments **[B][2017]**
- Learning attentional communication for multi-agent cooperation **[S][2018]**
- learning when to communicate at scale in multiagent cooperative and competitive tasks **[S][2018]**
- Qtran: Learning to factorize with transformation for cooperative multi-agent reinforcement learning **[B][2019]**
- Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient **[R][2019]**
- Tarmac: Targeted multi-agent communication **[S][2019]**
- Learning Individually Inferred Communication for Multi-Agent Cooperation **[S][2020]**
- Multi-Agent Game Abstraction via Graph Attention Neural Network **[G+S][2020]**
- Promoting Coordination through Policy Regularization in Multi-Agent Deep Reinforcement Learning **[E][2020]**

- Robust Multi-Agent Reinforcement Learning with Model Uncertainty **[R][2020]**
- Shared Experience Actor-Critic for Multi-Agent Reinforcement Learning **[B][2020]**
- Weighted QMIX Expanding Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning **[B][2020]**
- Cooperative Exploration for Multi-Agent Deep Reinforcement Learning **[E][2021]**
- Multiagent Adversarial Collaborative Learning via Mean-Field Theory **[R][2021]**
- The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games **[B][2021]**
- Variational Automatic Curriculum Learning for Sparse-Reward Cooperative Multi-Agent Problems **[2021]**
- ToM2C: Target-oriented Multi-agent Communication and Cooperation with Theory of Mind **[2021]**
- Taming Communication and Sample Complexities in Decentralized Policy Evaluation for Cooperative Multi-Agent Reinforcement Learning **[2021]**
- SPD: Synergy Pattern Diversifying Oriented Unsupervised Multi-agent Reinforcement Learning **[2022]**
- Distributional Reward Estimation for Effective Multi-Agent Deep Reinforcement Learning **[2022]**

14.2 SMAC

- Value-Decomposition Networks For Cooperative Multi-Agent Learning **[B][2017]**
- Counterfactual Multi-Agent Policy Gradients **[B][2018]**
- Multi-Agent Common Knowledge Reinforcement Learning **[RG+S][2018]**
- QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning **[B][2018]**
- Efficient Communication in Multi-Agent Reinforcement Learning via Variance Based Control **[S][2019]**
- Exploration with Unreliable Intrinsic Reward in Multi-Agent Reinforcement Learning **[P+E][2019]**
- Learning nearly decomposable value functions via communication minimization **[S][2019]**
- Liir: Learning individual intrinsic reward in multi-agent reinforcement learning **[P][2019]**
- MAVEN: Multi-Agent Variational Exploration **[E][2019]**
- Adaptive learning A new decentralized reinforcement learning approach for cooperative multiagent systems **[B][2020]**
- Counterfactual Multi-Agent Reinforcement Learning with Graph Convolution Communication **[S+G][2020]**
- Deep implicit coordination graphs for multi-agent reinforcement learning **[G][2020]**
- DOP: Off-policy multi-agent decomposed policy gradients **[B][2020]**
- F2a2: Flexible fully-decentralized approximate actor-critic for cooperative multi-agent reinforcement learning **[B][2020]**
- From few to more Large-scale dynamic multiagent curriculum learning **[MT][2020]**
- Learning structured communication for multi-agent reinforcement learning **[S+G][2020]**
- Learning efficient multi-agent communication: An information bottleneck approach **[S][2020]**
- On the robustness of cooperative multi-agent reinforcement learning **[R][2020]**
- Qatten: A general framework for cooperative multiagent reinforcement learning **[B][2020]**
- Revisiting parameter sharing in multi-agent deep reinforcement learning **[RG][2020]**

- Qplex: Duplex dueling multi-agent q-learning **[B][2020]**
- ROMA: Multi-Agent Reinforcement Learning with Emergent Roles **[RG][2020]**
- Towards Understanding Cooperative Multi-Agent Q-Learning with Value Factorization **[B][2021]**
- Contrasting centralized and decentralized critics in multi-agent reinforcement learning **[B][2021]**
- Learning in nonzero-sum stochastic games with potentials **[B][2021]**
- Natural emergence of heterogeneous strategies in artificially intelligent competitive teams **[S+G][2021]**
- Rode: Learning roles to decompose multi-agent tasks **[RG][2021]**
- SMIX(): Enhancing Centralized Value Functions for Cooperative Multiagent Reinforcement Learning **[B][2021]**
- Tesseract: Tensorised Actors for Multi-Agent Reinforcement Learning **[B][2021]**
- The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games **[B][2021]**
- UPDeT: Universal Multi-agent Reinforcement Learning via Policy Decoupling with Transformers **[MT][2021]**
- Randomized Entity-wise Factorization for Multi-Agent Reinforcement Learning **[MT][2021]**
- Cooperative Multi-Agent Transfer Learning with Level-Adaptive Credit Assignment **[MT][2021]**
- Uneven: Universal value exploration for multi-agent reinforcement learning **[B][2021]**
- Value-decomposition multi-agent actor-critics **[B][2021]**
- RMIX: Learning Risk-Sensitive Policies for Cooperative Reinforcement Learning Agents **[2021]**
- Regularized Softmax Deep Multi-Agent Q-Learning **[2021]**
- Policy Regularization via Noisy Advantage Values for Cooperative Multi-agent Actor-Critic methods **[2021]**
- ALMA: Hierarchical Learning for Composite Multi-Agent Tasks **[2022]**
- PAC: Assisted Value Factorisation with Counterfactual Predictions in Multi-Agent Reinforcement Learning **[2022]**
- Rethinking Individual Global Max in Cooperative Multi-Agent Reinforcement Learning **[2022]**
- Surprise Minimizing Multi-Agent Learning with Energy-based Models **[2022]**
- Heterogeneous Skill Learning for Multi-agent Tasks **[2022]**
- SHAQ: Incorporating Shapley Value Theory into Multi-Agent Q-Learning **[2022]**
- Self-Organized Group for Cooperative Multi-agent Reinforcement Learning **[2022]**
- ResQ: A Residual Q Function-based Approach for Multi-Agent Reinforcement Learning Value Factorization **[2022]**
- Efficient Multi-agent Communication via Self-supervised Information Aggregation **[2022]**
- Episodic Multi-agent Reinforcement Learning with Curiosity-Driven Exploration **[2022]**
- CTDS: Centralized Teacher with Decentralized Student for Multi-Agent Reinforcement Learning **[2022]**

14.3 MAMuJoCo

- FACMAC: Factored Multi-Agent Centralised Policy Gradients **[B][2020]**
- Trust Region Policy Optimisation in Multi-Agent Reinforcement Learning **[B][2021]**
- A Game-Theoretic Approach to Multi-Agent Trust Region Optimization **[2021]**
- Settling the Variance of Multi-Agent Policy Gradients **[2021]**
- Graph-Assisted Predictive State Representations for Multi-Agent Partially Observable Systems **[2022]**
- Order Matters: Agent-by-agent Policy Optimization **[2023]**

14.4 Google Research Football

- Adaptive Inner-reward Shaping in Sparse Reward Games **[P][2020]**
- Factored action spaces in deep reinforcement learning **[B][2021]**
- Semantic Tracklets An Object-Centric Representation for Visual Multi-Agent Reinforcement Learning **[B][2021]**
- TiKick: Towards Playing Multi-agent Football Full Games from Single-agent Demonstrations **[F][2021]**
- Celebrating Diversity in Shared Multi-Agent Reinforcement Learning **[2021]**
- Mingling Foresight with Imagination: Model-Based Cooperative Multi-Agent Reinforcement Learning **[2022]**

14.5 Pommerman

- Using Monte Carlo Tree Search as a Demonstrator within Asynchronous Deep RL **[I+T][2018]**
- Accelerating Training in Pommerman with Imitation and Reinforcement Learning **[I][2019]**
- Agent Modeling as Auxiliary Task for Deep Reinforcement Learning **[S][2019]**
- Backplay: man muss immer umkehren **[I][2019]**
- Terminal Prediction as an Auxiliary Task for Deep Reinforcement Learning **[B][2019]**
- Adversarial Soft Advantage Fitting Imitation Learning without Policy Optimization **[B][2020]**
- Evolutionary Reinforcement Learning for Sample-Efficient Multiagent Coordination **[B][2020]**

14.6 LBF & RWARE

- Shared Experience Actor-Critic for Multi-Agent Reinforcement Learning **[B][2020]**
- Benchmarking Multi-Agent Deep Reinforcement Learning Algorithms in Cooperative Tasks **[B][2021]**
- Learning Altruistic Behaviors in Reinforcement Learning without External Rewards **[B][2021]**
- Scaling Multi-Agent Reinforcement Learning with Selective Parameter Sharing **[RG][2021]**
- LIGS: Learnable Intrinsic-Reward Generation Selection for Multi-Agent Learning **[2022]**

14.7 MetaDrive

- Learning to Simulate Self-Driven Particles System with Coordinated Policy Optimization **[B][2021]**
- Safe Driving via Expert Guided Policy Optimization **[I][2021]**

14.8 Hanabi

- Bayesian Action Decoder for Deep Multi-Agent Reinforcement Learning **[B][2019]**
- Re-determinizing MCTS in Hanabi **[S+T][2019]**
- Diverse Agents for Ad-Hoc Cooperation in Hanabi **[B][2019]**
- Joint Policy Search for Multi-agent Collaboration with Imperfect Information **[T][20209]**
- Off-Belief Learning **[B][2021]**
- The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games **[B][2021]**
- 2021 Trajectory Diversity for Zero-Shot Coordination **[B][2021]**

14.9 MAgent

- Mean field multi-agent reinforcement learning **[B][2018]**
- Graph convolutional reinforcement learning **[B][2018]**
- Factorized q-learning for large-scale multi-agent systems **[B][2019]**
- From few to more Large-scale dynamic multiagent curriculum learning **[MT][2020]**

14.10 Other Tasks

- Learning Fair Policies in Decentralized Cooperative Multi-Agent Reinforcement Learning **[2020]**
- Contrasting Centralized and Decentralized Critics in Multi-Agent Reinforcement Learning **[2021]**
- Learning to Ground Multi-Agent Communication with Autoencoders **[2021]**
- Latent Variable Sequential Set Transformers For Joint Multi-Agent Motion Prediction **[2021]**
- Learning to Share in Multi-Agent Reinforcement Learning **[2021]**
- Resilient Multi-Agent Reinforcement Learning with Adversarial Value Decomposition **[2021]**
- Multi-Agent MDP Homomorphic Networks **[2021]**
- Multi-Agent Reinforcement Learning for Active Voltage Control on Power Distribution Networks **[2021]**
- Multi-Agent Reinforcement Learning in Stochastic Networked Systems **[2021]**
- Mirror Learning: A Unifying Framework of Policy Optimisation **[2022]**
- E-MAPP: Efficient Multi-Agent Reinforcement Learning with Parallel Program Guidance **[2022]**
- Shield Decentralization for Safe Multi-Agent Reinforcement Learning **[2022]**
- Provably Efficient Offline Multi-agent Reinforcement Learning via Strategy-wise Bonus **[2022]**

- Asynchronous Actor-Critic for Multi-Agent Reinforcement Learning **[2022]**
- Near-Optimal Multi-Agent Learning for Safe Coverage Control **[2022]**
- Multi-agent Dynamic Algorithm Configuration **[2022]**

14.11 New Environments

- SMACv2: A New Benchmark for Cooperative Multi-Agent Reinforcement Learning **[2022]**
- MATE: Benchmarking Multi-Agent Reinforcement Learning in Distributed Target Coverage Control **[2022]**
- Nocturne: a scalable driving benchmark for bringing multi-agent learning one step closer to the real world **[2022]**
- GoBigger: A Scalable Platform for Cooperative-Competitive Multi-Agent Interactive Simulation **[2023]**

CHAPTER

FIFTEEN

EXISTING BENCHMARKS

We collect most of the existing MARL benchmarks here with brief introduction.

[B] Basic [S] Information Sharing [RG] Behavior/Role Grouping [I] Imitation [G] Graph [E] Exploration [R] Robust [P] Reward Shaping [F] Offline [T] Tree Search [MT] Multi-task

15.1 PyMARL

Github: https://github.com/oxwhirl/pymarl

PyMARL is the first and most well-known MARL library. All algorithms in PyMARL is built for SMAC, where agents learn to cooperate for a higher team reward. However, PyMARL has not been updated for a long time, and can not catch up with the recent progress. To address this, the extension versions of PyMARL are presented including PyMARL2 and EPyMARL.

15.2 PyMARL2

Github: https://github.com/hijkzzz/pymarl2

PyMARL2 focuses on credit assignment mechanism and provide a finetuned QMIX with state-of-art-performance on SMAC. The number of available algorithms increases to ten, with more code-level tricks incorporated.

15.3 EPyMARL

Github: https://github.com/uoe-agents/epymarl

EPyMARL is another extension for PyMARL that aims to present a comprehensive view on how to unify cooperative MARL algorithms. It first proposed the independent learning, value decomposition, and centralized critic categorization, but is restricted to cooperative algorithms. Nine algorithms are implemented in EPyMARL. Three more cooperative environments LBF, RWARE, and MPE are incorporated to evaluate the generalization of the algorithms.

15.4 MARL-Algorithms

Github: https://github.com/starry-sky6688/MARL-Algorithms

MARL-Algorithm is a library that covers broader topics compared to PyMARL including learning better credit assignment, communication-based learning, graph-based learning, and multi-task curriculum learning. Each topic has at least one algorithm, with nine implemented algorithms in total. The testing bed is limited to SMAC.

15.5 MAPPO benchmark

Github: https://github.com/marlbenchmark/on-policy

MAPPO benchmark is the official code base of MAPPO. It focuses on cooperative MARL and covers four environments. It aims at building a strong baseline and only contains MAPPO.

15.6 MAlib

Github: https://github.com/sjtu-marl/malib

MAlib is a recent library for population-based MARL which combines game-theory and MARL algorithm to solve multi-agent tasks in the scope of meta-game.

15.7 MARLlib

Please refer to *Introduction*.