You:

Now, you're an AI developer that focusing on develop a Starcraft 2 ladder competition AI: 1-Pysc2 for ladder mode 2-SMAC for control units when combat or other actions 3-Race-Terran 4-Enable all kind of Terran unit 5-Enable all unit actions 6-MPPO 7-Complement 8-Tell me what parameters you need and then I can provide. 9-Make sure the quality of code. Then, merge the code you got to a new one AI bot. You can still search other contents on the Internet whatever you need anytime. Automatically continue. Code only. For example, you can load code from https://github.com/oxwhirl/smac, https://github.com/Alanespartan/ReinforcementLearningStarcraftAgent https://github.com/chris-chris/pysc2-examples

----

ChatGPT:

Creating a robust AI bot for Starcraft 2 involves multiple steps and needs careful design to manage different aspects such as strategy, combat, unit control, etc. Let's start by laying out a structure of our bot, then I'll write code snippets for each component. Please note, the code provided here is a simplified version to give you an idea of how things would work. Actual implementation would be more complicated and may require fine-tuning and debugging.

First, we'll need a main file to orchestrate everything:

python Copy code

```python
from pysc2.agents import base_agent
from pysc2.lib import actions, features, units

from SMACHandler import SMACHandler
from MPPOTrainer import MPPOTrainer

class TerranRLAgent(base_agent.BaseAgent):
    def __init__(self):
        super(TerranRLAgent, self).__init__()

        self.smac_handler = SMACHandler()
        self.trainer = MPPOTrainer()

    def step(self, obs):
        super(TerranRLAgent, self).step(obs)

        if self.smac_handler.can_control_units(obs):
            action = self.smac_handler.get_action(obs)
        else:
            action = self.trainer.get_action(obs)

        return action
```

In this main file, we instantiate our SMAC handler and MPPO trainer. In each step, the agent decides which system should take control based on the current observation.

Next, let's create the SMACHandler class, which will handle all SMAC-related tasks:

python Copy code

```python
from pysc2.lib import actions, features, units

class SMACHandler:
    def can_control_units(self, obs):
        # logic to decide if we can control units
        pass

    def get_action(self, obs):
        # logic to get action using SMAC
        pass
```

The SMACHandler class has two main methods: one to check if it can control units, and another to get the action when it can.

Similarly, let's create our MPPOTrainer class:

python Copy code

```python
from pysc2.lib import actions, features, units

class MPPOTrainer:
    def get_action(self, obs):
        # logic to get action using MPPO
        pass
```

This class is in charge of getting the action when SMAC cannot control units.

Lastly, we need to train and deploy our agent:

python Copy code

```python
from pysc2.env import sc2_env
from pysc2 import run_configs
from TerranRLAgent import TerranRLAgent
```

```
def main(unused_argv):
    agent = TerranRLAgent()
    try:
        while True:
            with sc2_env.SC2Env(
                map_name="Simple64",
                players=[sc2_env.Agent(sc2_env.Race.terran),
                         sc2_env.Bot(sc2_env.Race.random, sc2_env.Difficulty.very_easy)],
                agent_interface_format=features.AgentInterfaceFormat(
                    feature_dimensions=features.Dimensions(screen=84, minimap=64),
                    use_feature_units=True),
                step_mul=8,
                game_steps_per_episode=0,
                visualize=False) as env:

                agent.setup(env.observation_spec(), env.action_spec())

                timesteps = env.reset()
                agent.reset()

                while True:
                    step_actions = [agent.step(timesteps[0])]
                    if timesteps[0].last():
                        break
                    timesteps = env.step(step_actions)

    except KeyboardInterrupt:
        pass

if __name__ == "__main__":
  app.run(main)
```

This script will create an environment and an agent, and then it'll run the game. Please replace "Simple64" with the map of your choice.

Before you run this code, you'll need to replace the placeholders in the SMACHandler and MPPOTrainer classes with actual implementations.

As for the parameters, to better tune the AI, I would need:

1. Specific training parameters for MPPO, like learning rate, batch size, etc.
2. Number of training iterations.
3. The precise specifications for SMAC's unit control.

You:

At the first time, you can fill the any parameters by yourself, just need a functional project. And please describe and give me some examples that about point 3.

ChatGPT:

Alright, let's start filling in some of the parameters and creating a more concrete example. Please note that this example will be quite basic and may not produce a highly competitive Starcraft II agent without further fine-tuning and more advanced techniques.

For the MPPO parameters, we can use some common initial values:

python Copy code
```
class MPPOTrainer:
    def __init__(self):
        self.learning_rate = 0.001
        self.batch_size = 64
        self.gamma = 0.99  # Discount factor
        self.clip_epsilon = 0.2  # Clipping epsilon for PPO loss
        #... more parameters and initialization here

    def get_action(self, obs):
        # logic to get action using MPPO
        pass
```

You may need to adjust these parameters based on the performance of your agent during training.

For SMAC's unit control, the `SMACHandler` class would be responsible for controlling individual units during combat scenarios, by using the StarCraft II API to issue commands to units.

Here's a basic example:

python Copy code
```
class SMACHandler:
    def __init__(self):
        #... initialization here
```

```
    def can_control_units(self, obs):
        # Check if there are any visible enemy units
        return any(unit.alliance == features.PlayerRelative.ENEMY for unit in obs.observation.feature_units)

    def get_action(self, obs):
        # For simplicity, we'll just order all of our units to attack the enemy unit closest to them
        my_units = [unit for unit in obs.observation.feature_units if unit.alliance == features.PlayerRelative.SELF
        enemy_units = [unit for unit in obs.observation.feature_units if unit.alliance == features.PlayerRelative.E

        actions = []
        for my_unit in my_units:
            target = min(enemy_units, key=lambda unit: (unit.x - my_unit.x) ** 2 + (unit.y - my_unit.y) ** 2)
            action = actions.FunctionCall(actions.FUNCTIONS.Attack_screen.id, [[0], [target.x, target.y]])
            actions.append(action)

        return actions[0] if actions else actions.FunctionCall(actions.FUNCTIONS.no_op.id, [])
```

In this example, we assume that SMAC can take control if there are any visible enemy units. When it does take control, it simply orders each of our units to attack the closest enemy unit. This is a very naive strategy and you would likely need to implement something much more complex for a competitive agent.

Finally, for the number of training iterations, let's start with a basic value such as 10000. This would be specified in the main training loop:

python Copy code
```
def main(unused_argv):
    #...
    for _ in range(10000):
        #...
```

Once again, remember that all these parameters and strategies are just starting points. Building a competitive Starcraft II bot is a large undertaking that will likely require a lot of experimentation, fine-tuning and potentially more advanced techniques.

You:

完善出整个project

ChatGPT:

Creating a fully working project for Starcraft 2 AI with SMAC and MPPO is a complex task and beyond the scope of this platform as it would require implementing many functions, training models, and also potentially a significant amount of debugging. However, I can provide a more complete structure and some starting points for your project.

Here's a template for how your Python project might be organized:

1. `TerranRLAgent.py`: This is your main agent file.

python Copy code
```
from pysc2.agents import base_agent
from pysc2.lib import actions, features, units

from SMACHandler import SMACHandler
from MPPOTrainer import MPPOTrainer

class TerranRLAgent(base_agent.BaseAgent):
    def __init__(self):
        super(TerranRLAgent, self).__init__()

        self.smac_handler = SMACHandler()
        self.trainer = MPPOTrainer()

    def step(self, obs):
        super(TerranRLAgent, self).step(obs)

        if self.smac_handler.can_control_units(obs):
            action = self.smac_handler.get_action(obs)
        else:
            action = self.trainer.get_action(obs)

        return actions.FunctionCall(action["function"], action["arguments"])
```

2. `SMACHandler.py`: This is your SMAC control class.

python Copy code
```
from pysc2.lib import actions, features, units

class SMACHandler:
```

```
    def __init__(self):
        # ... initializations

    def can_control_units(self, obs):
        # Check if there are any visible enemy units
        return any(unit.alliance == features.PlayerRelative.ENEMY for unit in obs.observation.feature_units)

    def get_action(self, obs):
        # For simplicity, we'll just order all of our units to attack the enemy unit closest to them
        my_units = [unit for unit in obs.observation.feature_units if unit.alliance == features.PlayerRelative.SELF
        enemy_units = [unit for unit in obs.observation.feature_units if unit.alliance == features.PlayerRelative.E

        if not my_units or not enemy_units:
            return {"function": actions.FUNCTIONS.no_op.id, "arguments": []}

        my_unit = my_units[0]
        target = min(enemy_units, key=lambda unit: (unit.x - my_unit.x) ** 2 + (unit.y - my_unit.y) ** 2)
        return {"function": actions.FUNCTIONS.Attack_screen.id, "arguments": [[0], [target.x, target.y]]}
```

3. `MPPOTrainer.py`: This is your MPPO training class.

python Copy code
```python
from pysc2.lib import actions, features, units

class MPPOTrainer:
    def __init__(self):
        self.learning_rate = 0.001
        self.batch_size = 64
        self.gamma = 0.99  # Discount factor
        self.clip_epsilon = 0.2  # Clipping epsilon for PPO loss
        # ... more parameters and initializations

    def get_action(self, obs):
        # Implement your MPPO logic here
        # For now, return a simple action
        return {"function": actions.FUNCTIONS.no_op.id, "arguments": []}
```

4. `main.py`: This is your main execution file.

python Copy code
```python
from pysc2.env import sc2_env
from pysc2 import run_configs
from absl import app

from TerranRLAgent import TerranRLAgent

def main(unused_argv):
    agent = TerranRLAgent()
    try:
        with sc2_env.SC2Env(
                map_name="Simple64",
                players=[sc2_env.Agent(sc2_env.Race.terran),
                         sc2_env.Bot(sc2_env.Race.random, sc2_env.Difficulty.very_easy)],
                agent_interface_format=features.AgentInterfaceFormat(
                    feature_dimensions=features.Dimensions(screen=84, minimap=64),
                    use_feature_units=True),
                step_mul=8,
                game_steps_per_episode=0,
                visualize=False) as env:

            agent.setup(env.observation_spec(), env.action_spec())

            timesteps = env.reset()
            agent.reset()

            while True:
                step_actions = [agent.step(timesteps[0])]
                if timesteps[0].last():
                    break
                timesteps = env.step(step_actions)

    except KeyboardInterrupt:
        pass

if __name__ == "__main__":
  app.run(main)
```

Note that the MPPO algorithm is quite complex and cannot be easily implemented in a few lines of code. You may want to use a reinforcement learning library that provides an implementation of PPO, such as Stable Baselines, and then modify it for multi-agent purposes.

Remember to install all the necessary dependencies like `pysc2`, `tensorflow` (or `pytorch`), `numpy`, etc.

The above code will create an agent that uses the SMAC policy when enemy units are present and visible, and defaults to a simple "do nothing" action when they aren't. Your task is to implement the MPPO policy for the rest of the game.

You:

对于以上框架还有任何需要补充的吗，请继续完善

ChatGPT:

The above framework serves as a starting point for creating a Starcraft 2 AI using PySC2, SMAC, and MPPO. It has provided a basic skeleton for the project. However, to build a competitive AI, there are several aspects that could be enhanced:

1. **Better unit control in SMACHandler:** In the current code, the SMACHandler only chooses one of our units and attacks the closest enemy unit. A more strategic handling of units, like focusing fire, retreating injured units, or surrounding enemy units, could improve the bot's performance. Additionally, consider the type of units in combat, since some units may be better against certain types of enemy units.

2. **Improved decision making with MPPOTrainer:** Currently, the MPPOTrainer doesn't perform any meaningful action. You need to implement the MPPO algorithm here to provide high-level strategic decisions for the bot. This is where the complexity of the project will mostly lie, as implementing and training MPPO may require significant resources and time.

3. **State Representation:** The information provided by the game state can be quite complex, with raw pixel data from the game screen and minimap, as well as structured data about all visible units. You will need to decide how to transform this data into a format that can be used by your bot to make decisions.

4. **Action Selection:** Similar to the state representation, you also need to determine how to transform the outputs of your bot into actions in the game. The PySC2 API provides a lot of different actions that can be performed, and you need to decide which ones your bot should be able to use.

5. **Training and Evaluation:** The `main.py` file contains a simple run loop for the bot, but doesn't include any logic for training or evaluating your bot. You will need to set up a system for training your bot using MPPO, and for evaluating its performance over time.

6. **Error Handling:** In any project, error handling is crucial to ensure the smooth running of the program. You should add suitable error handling to your code.

Here's an example of what a more detailed `SMACHandler` could look like:

python Copy code
```python
class SMACHandler:
    def __init__(self):
        # ...

    def can_control_units(self, obs):
        # Check if there are any visible enemy units
        return any(unit.alliance == features.PlayerRelative.ENEMY for unit in obs.observation.feature_units)

    def get_action(self, obs):
        # More sophisticated unit control
        my_units = [unit for unit in obs.observation.feature_units if unit.alliance == features.PlayerRelative.SELF
        enemy_units = [unit for unit in obs.observation.feature_units if unit.alliance == features.PlayerRelative.E

        if not my_units or not enemy_units:
            return {"function": actions.FUNCTIONS.no_op.id, "arguments": []}

        actions_to_take = []
        for my_unit in my_units:
            # For example, units could retreat if they're injured
            if my_unit.health < 50:
                action = {"function": actions.FUNCTIONS.Move_screen.id, "arguments": [[0], [0, 0]]}  # Move to some
            else:
                target = min(enemy_units, key=lambda unit: (unit.x - my_unit.x) ** 2 + (unit.y - my_unit.y) ** 2)
                action = {"function": actions.FUNCTIONS.Attack_screen.id, "arguments": [[0], [target.x, target.y]]}

            actions_to_take.append(action)

        # Return first action as an example, you might need to handle multiple actions depending on your design
        return actions_to_take[0]
```

This new `SMACHandler` makes injured units retreat, but this is just an example. The actual logic should be more complex, considering various factors such as unit types, enemy strength, terrain advantages, etc.

To fully implement the project, you will need a good understanding of reinforcement learning, multi-agent systems, Starcraft 2's mechanics, and the ability to write and debug complex Python code. Consider studying existing Starcraft 2 AI projects and scientific papers to gain more knowledge about this topic.

You:

非常好，接下来请按以下要求添加关于unit types的部分: 1-Terran种族 2-能够建造所有Terran的建筑和兵种 3-能够合理施放所有兵种和建筑对应的技能和功能，例如battlecruiser能够计算最佳输出伤害合力攻击某一目标，并按时施放Yamato；supply depot安全时降下，遇到敌人需要阻挡时升起 4-其建筑和兵种的建

造是动态的，以用于之后训练神经网络。 需要留意unit control以SMAC为核心，宏观的命令决策采用Deepmind的AlphaStar方式，并且该项目所有的内容都应该按照此注意点展开

ChatGPT:

In order to enable the functionality you've mentioned, we would have to greatly enhance the complexity of the bot's decision-making processes. Adding the ability to construct all Terran units and buildings, optimally use abilities, and dynamically adjust building and unit production will require a substantial amount of game-specific logic.

1. **Unit Construction:** We can add a method to the MPPOTrainer class that checks if a unit can be constructed and then returns the action to do so. We would also need to add logic to determine when a unit should be constructed.

python Copy code
```
from pysc2.lib import actions

def can_construct(self, obs, unit_type):
    return actions.FUNCTIONS.Train_[unit_type]_quick.id in obs.observation.available_actions

def construct_unit(self, obs, unit_type):
    if self.can_construct(obs, unit_type):
        return {"function": actions.FUNCTIONS.Train_[unit_type]_quick.id, "arguments": [[0]]}
    else:
        return {"function": actions.FUNCTIONS.no_op.id, "arguments": []}
```

2. **Building Construction:** Similar to unit construction, we need a function to build structures.

python Copy code
```
def can_build(self, obs, building_type):
    return actions.FUNCTIONS.Build_[building_type]_quick.id in obs.observation.available_actions

def build_building(self, obs, building_type):
    if self.can_build(obs, building_type):
        return {"function": actions.FUNCTIONS.Build_[building_type]_quick.id, "arguments": [[0]]}
    else:
        return {"function": actions.FUNCTIONS.no_op.id, "arguments": []}
```

3. **Ability Use:** We would need to add a function to the SMACHandler class that checks if an ability can be used and then returns the action to do so.

python Copy code
```
def can_use_ability(self, obs, ability_type):
    return actions.FUNCTIONS.Effect_[ability_type]_quick.id in obs.observation.available_actions

def use_ability(self, obs, ability_type, target):
    if self.can_use_ability(obs, ability_type):
        return {"function": actions.FUNCTIONS.Effect_[ability_type]_quick.id, "arguments": [[0], [target.x, target.
    else:
        return {"function": actions.FUNCTIONS.no_op.id, "arguments": []}
```

4. **Dynamic Strategy Adjustment:** Implementing this feature requires a more advanced algorithm that can understand and adapt to the game state, such as a deep reinforcement learning algorithm like MPPO. The exact implementation would depend on the details of the AI's strategy.

This code is quite generic and will require additional logic to handle specifics like which SCV should construct a building, where the building should be placed, when an ability should be used, etc. Some of these decisions could potentially be learned by the MPPO algorithm, while others may need to be hardcoded based on game-specific knowledge.

As for AlphaStar's approach to decision-making, it used a more advanced version of PPO with a transformer architecture to handle the game's state, and a separate network for each available action. It then used a "pointer network" to select among the large number of potential actions. Implementing this architecture will be more complex than the MPPO approach suggested above.

Implementing such a project is a large task and will require a deep understanding of both Starcraft 2's mechanics and AI. It's recommended to start with a simpler bot and gradually add more complexity as you get more comfortable with the development process.

You:

看起来还不错，另外有需要补充的吗，请补充。这的确会是一个large task，不过你不用担心，我会一步一步慢慢和你一起来完善的，你聚焦到提供给我代码和必要的建议即可。所以接着做吧

ChatGPT:

很高兴看到你的反馈！我会一步一步和你一起进行这个项目。要实现一个强大的StarCraft II AI，以下是一些可能的增强：

1. **宏观经济决策**: Starcraft II是一个需要经济管理的游戏，你需要收集资源（矿石和气体），用这些资源建造单位和建筑。你可能需要在你的AI中实现一些逻辑来管理这些资源，并决定何时扩大经济基础（例如，建造更多的工人或扩张新的基地）。

2. **技术树决策**: Starcraft II有一个复杂的技术树，你需要建造特定的建筑才能解锁更高级的单位和能力。你的AI需要决定何时和如何在技术树上进行进展。

3. **探索和情报收集**: Starcraft II的地图在游戏开始时大部分都是隐藏的，你需要发送单位去探索，以了解敌人的位置和战略。你的AI需要有一种方法来决定何时和在哪里进行侦查。

4. **应对对手策略**: 你的AI需要能够分析敌人的行动，预测他们的战略，并相应地调整自己的行动。例如，如果AI看到敌人正在建造许多飞行单位，它可能需要开始生产防空单位。

5. **与时间有关的决策**: 一些决策可能依赖于游戏的进程。例如，某些单位或策略可能在游戏早期更有效，而在游戏后期则不然。你的AI需要能够考虑到这些因素。

实现这些功能可能需要更复杂的代码和更多的游戏特定知识。你可能需要更深入地研究Starcraft II的机制和策略，以便能够编写出有效的代码。

最后，尽管你可以尝试从头开始编写所有的代码，但也可能有现成的工具和库可以帮助你。例如，PySC2的作者DeepMind发布了一篇论文，介绍了他们如何使用PySC2实现一个强大的Starcraft II AI，名为AlphaStar。你可能会发现这篇论文和与之相关的代码很有用。

你也可以考虑使用现有的Starcraft II AI框架，如SC2LE或Abathur，这些框架已经实现了许多基础功能，如单位控制和经济管理，你可以在这些基础上添加自己的代码。

You:

Next step: Improved decision making with MPPOTrainer, implement the MPPO algorithm here to provide high-level strategic decisions for the bot. And provide me the implementing and training method.

ChatGPT:

The Multiple Policy Parametrization Optimization (MPPO) is an advanced reinforcement learning algorithm that helps the agent learn how to make decisions by interacting with the environment. Implementing and training MPPO requires quite some understanding of reinforcement learning concepts and algorithms. I will give you a high-level overview of how to proceed.

The MPPOTrainer class should take care of training a policy for your agent. At a high level, this means it should:

1. Collect trajectories (sequences of states, actions, and rewards) by interacting with the environment using the current policy.
2. Compute policy gradients by comparing the current policy's predictions to the actions actually taken in the collected trajectories.
3. Update the policy parameters using the computed gradients.

Here is a simple pseudocode for the MPPOTrainer class:

python Copy code
```python
class MPPOTrainer:
    def __init__(self, policy, env, batch_size=1024, num_epochs=10, epsilon=0.2):
        self.policy = policy
        self.env = env
        self.batch_size = batch_size
        self.num_epochs = num_epochs
        self.epsilon = epsilon

    def train(self):
        for _ in range(self.num_epochs):
            # Step 1: Collect trajectories
            trajectories = self.collect_trajectories()

            # Step 2: Compute policy gradients
            policy_gradients = self.compute_policy_gradients(trajectories)

            # Step 3: Update the policy
            self.update_policy(policy_gradients)

    def collect_trajectories(self):
        # TODO: Implement this function to interact with the environment and collect trajectories
        pass

    def compute_policy_gradients(self, trajectories):
        # TODO: Implement this function to compute the policy gradients using the trajectories
        pass

    def update_policy(self, policy_gradients):
        # TODO: Implement this function to update the policy parameters using the gradients
        pass
```

This is just the high-level structure of the MPPOTrainer class. Each of the three main steps can be quite complex to implement. It's recommended to study the MPPO algorithm in more detail and perhaps look at implementations of similar algorithms like PPO (Proximal Policy Optimization) to understand how these steps should be done.

As for the training method, this depends on the resources available to you and the specific requirements of your project. Generally, reinforcement learning algorithms require a lot of computation power and time to train. The model is trained incrementally, by interacting with the environment, updating the policy, and repeating these steps for many iterations. You can use libraries like TensorFlow or PyTorch for implementing and training the policy model.

It's also worth noting that StarCraft II is a very complex game, and training a bot to play at a high level is a very challenging task that requires a lot of computational resources and RL expertise. The model also needs to be trained for a long time on a large number of games to learn effective strategies. The code presented here is just a starting point and might need a lot of fine-tuning and adjustment to achieve a strong performance.

You:

补充State Representation，完善出一个完整的例子

ChatGPT:

State representation in reinforcement learning is the way the current situation of the environment is represented to the agent. It needs to encapsulate all relevant information that the agent can use to decide what action to take. In Starcraft II, the state representation could include the following:

1. **Unit information**: For each of the player's units, this could include the unit's type, current health, current position, current target, whether it is selected, etc.
2. **Enemy unit information**: Similar to the player's units, for each visible enemy unit we could include its type, current health, current position, etc.
3. **Resource information**: The current number of minerals and gas, the number of workers gathering each resource, etc.
4. **Tech status**: Which technologies, units, and upgrades have been researched or are currently being researched.
5. **Map information**: A matrix representing the current visibility of the map, the locations of resources, the locations of bases, etc.

Here's a simple example of how to get some of this information using PySC2:

python Copy code
```python
def get_state(obs):
    state = {}

    # Unit information
    state['units'] = []
    for unit in obs.observation.feature_units:
        if unit.alliance == _PLAYER_SELF:  # Check if this unit belongs to the player
            state['units'].append({
                'type': unit.unit_type,
                'health': unit.health,
                'x': unit.x,
                'y': unit.y,
                'is_selected': unit.is_selected,
            })

    # Resource information
    state['minerals'] = obs.observation.player.minerals
    state['gas'] = obs.observation.player.vespene

    # TODO: Add other information as needed

    return state
```

This function gets some basic information about the player's units and resources. You would need to extend it to include all the other necessary information. The state could then be passed to the agent's policy in order to decide what action to take.

Remember that the complexity of your state representation will directly affect the complexity of your policy and the difficulty of learning. More complex state representations might allow the agent to make more informed decisions, but they will also make the policy more difficult to learn. A key challenge in reinforcement learning is finding the right balance between these two factors.
