**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-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:**

Absolutely, you can extend the existing classes to handle different types of units, build structures, and train units. Here is how you can implement this functionality:

1. Add a list of commands for each unit in the `SMACHandler` class to handle. This list includes the available actions for the unit such as `attack`, `move`, `build_structure`, `train_unit`, etc. Remember to update this list dynamically as the game state changes (e.g., more actions become available as the player researches new technologies or builds new structures).

2. In the `get_action` method of `SMACHandler`, determine the appropriate action for each unit based on its type and the current game state.

Here's an example of how you could extend the `SMACHandler` class to handle different types of Terran units and their actions:

python Copy code
```python
class SMACHandler:
    def __init__(self):
        # ...
        self.commands = {
            units.Terran.SCV: [self.move, self.attack, self.build_structure, self.gather_resources],
            units.Terran.Marine: [self.move, self.attack],
            # ...
        }

    # ...

    def get_action(self, obs):
        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:
            if my_unit.health < 50:
                action = self.move(my_unit)  # Move to some safe position
            else:
                # Choose an action based on the unit type
                if my_unit.unit_type in self.commands:
                    for command in self.commands[my_unit.unit_type]:
                        if command.can_execute(my_unit, obs):
                            action = command.execute(my_unit, obs)
                            break
                else:
                    # Default to attacking
                    target = min(enemy_units, key=lambda unit: (unit.x - my_unit.x) ** 2 + (unit.y - my_unit.y) **
                    action = {"function": actions.FUNCTIONS.Attack_screen.id, "arguments": [[0], [target.x, target.

            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 is a simple example and real implementation will be much more complex. For instance, you may need a way to coordinate the actions of different units (e.g., don't send a single marine to attack an enemy base), decide where to build structures or move units to, manage the economy (e.g., when to build workers or gather resources), among other things.

Additionally, you will also need to implement the `move`, `attack`, `build_structure`, and `gather_resources` commands, along with any other commands you want your units to be able to execute. These commands should have `can_execute` and `execute` methods. The `can_execute` method checks if the command can be executed given the current game state and unit, and the `execute` method returns the action to be taken.

As for the AlphaStar approach for macro decision making, it's a much more complex topic. AlphaStar uses a separate neural network for each type of unit to make decisions, and trains these networks using a combination of supervised learning (from human replay data) and reinforcement learning. Implementing this from scratch would be a substantial task and is beyond the scope of this platform. However, I hope the above example gives you a good starting point for your project.

------------------