Implementing an Agent with pysc2

Miguel Angel Navarro Mata

March 24, 2020

Contents

0.1 Introduction

In this document we are going to be reviewing the implementation of a Zerg Bot with PySC2 2.0 by Steven Brown. Before we start we need to ensure we have installed the following items in our environment:

StarCraft 2
Python 3.x.x
PySC2 2.0

0.2 Creating the basic Agent

First we need to import the following libraries

```
from pysc2.agents import base_agent
from pysc2.env import sc2_env
from pysc2.lib import actions, features
from absl import app
```

Now, we can create our agent class:

```
class ZergAgent(base_agent.BaseAgent):
    def step(self, obs):
        super(ZergAgent, self).step(obs)

        return actions.FUNCTIONS.no_op()
```

For now our class only has a method: step. This method it's where all of our decision making takes place. At the end of every step it returns an action. But for now it just return no action because it hasn't had been added the logic to take decisions yet.

0.3 Add the Run Code

Now we need to set up the environment to run it. For that we need to implement the following code, in which we specify that our agent will be runnning in the map **Simple64**.

```
def main(useless_argv):
agent = ZergAgent()

try:
    while True:
        with sc2_env.SC2Env(
             map_name="Simple64",
```

After this, we need to specify our players. In the following code we specify that the first player is our agent, and the agent's race is Zerg. Then we specify that the second player is a bot, using the game's internal AI, the bot race is random and the difficulty level is very easy.

```
players=[sc2_env.Agent(sc2_env.Race.zerg),
          sc2_env.Bot(sc2_env.Race.random,
          sc2_env.Difficulty.very_easy)],
```

Then, we specify the screen and minimap resolutions. These resolutions essentially determine how many "pixels" of data are in each feature layer, those layers include things like terrain height, visibility, and unit ownership.

```
agent_interface_format=features.AgentInterfaceFormat(
    feature_dimensions=features.Dimensions(screen=84,
    minimap=64)
),
```

We need to set the amout of "game steps" that pass before our bot will choose an action. By default is set to 8, but in this case we are going to be setting it to 16.

```
step_mul=16,
```

Then we set our legth of each game, we do that by setting the following parameter to the amount of minutes we want(NOTE: setting it to 0 makes the game run as long as necessary):

```
game_steps_per_episode=0,
```

The following parameter is optional, but it helps you to see all the the observation layers available to your bot.

```
visualize=True) as env:
```

The rest of the coding it's just about the looping, inputting the agent the current environment state, receiving an action and repeating until the game is finnished.

```
                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)
```

To this point our code should be able to run however it does not perform any actions since we haven't implemented yet the logic for it to take decisions about what actions to take. In the following sections we are going to be using the code constructed until this point and improving it by adding more actions.

0.4 Select a Drone

Before our agent can create any Zerling we need a Spawing Pool. And to build an Spawning Pool, we need to select a drone. So the first step for our agent is going to be select a drone.

We need to add the unit list to the module import:

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

Unit list allows you to retrieve unit types using a unit's name. We have enable the feature units in our main with the agent interface format parameter:

```
agent_interface_format=features.AgentInterfaceFormat(
    feature_dimensions=features.Dimensions(screen=84,
                                           minimap=64),
    use_feature_units=True
),
```

Now that we have enabled the units we can go back to our **ZergAgent** class to get a list of all the drones on the screen by the definition of the following utility method:

```
def get_units_by_type(self, obs, unit_type):
        return [unit for unit in obs.observation.feature_units
                if unit.unit_type == unit_type]
```

And we call it in our **step()** method to retrieve the drones on screen at the moment.

```
def step(self, obs):
    super(ZergAgent, self).step(obs)

    drones = self.get_units_by_type(obs, units.Zerg.Drone)
```

Now, we select a Drone:

```
if len(drones) > 0:
    drone = random.choice(drones)

    return actions.FUNCTIONS.select_point("select_all_type",
            (drone.x, drone.y))
```

The **select_all_type** parameter acts like CNTRL + click, so all Drones on the screen will be selected.

To this point our agent is able to select the drones, the next step will be implementing the code so this Drones can create *Spawning Pools.*

0.5 Build a Spawning Pool

Before starting the implementation for this step, we need to validate we have a Drone selected. So let's add this utility method to our agent class:

```
def unit_type_is_selected(self, obs, unit_type):
    if (len(obs.observation.single_select) > 0 and
        obs.observation.single_select[0].unit_type == unit_type):
      return True

    if (len(obs.observation.multi_select) > 0 and
        obs.observation.multi_select[0].unit_type == unit_type):
      return True

    return False
```

This code checks both the single and multi-selections to see if the first selected unit is the correct type.

So, modifying our step method to validate it, should look like this:

```
def step(self, obs):
    super(ZergAgent, self).step(obs)

    if self.unit_type_is_selected(obs, units.Zerg.Drone):
```

Next, we check if we can build a Spawning Pool. If we don't have enough minerals this may no tbe possible and will result in a crash. To check if we can perform an action, we need to add the following utility method to our agent class:

```
def can_do(self, obs, action):
    return action in obs.observation.available_actions
```

Once we can verify the actions we can perform at the moment, we can implement the code to create Spawning Pools.

```
if self.can_do(obs, actions.FUNCTIONS.Build_SpawningPool_screen.id):
    # Selects a random point on the screen.
    x = random.randint(0, 83)
    y = random.randint(0, 83)
    return actions.FUNCTIONS.Build_SpawningPool_screen("now", (x,y))
```

Can you see the problem in here? if we let the code as it is all our drones would become spawning pools. We need to modify our code to look like this:

```
spawning_pools = self.get_units_by_type(obs, units.Zerg.SpawningPool)
# If there's no spawning pool, it will build one.
if len(spawning_pools) == 0:
    # Check if we have a Drone selected.
    if self.unit_type_is_selected(obs, units.Zerg.Drone):
        # Check if we can build a Spawning Pool. If we don't have
        # enough minerals this may not be possible and will result
        # in a crash.
        if self.can_do(obs,
        actions.FUNCTIONS.Build_SpawningPool_screen.id):
            # Selects a random point on the screen.
            x = random.randint(0, 83)
            y = random.randint(0, 83)

            return actions.FUNCTIONS.Build_SpawningPool_screen("now",
                    (x,y))

    # List of all Drones on the screen.
    drones = self.get_units_by_type(obs, units.Zerg.Drone)

    if len(drones) > 0:
        drone=random.choice(drones)
        # The select_all_type parameter here acts like CTRL + click,
        # so all Drones on the screen will be selected.
        return actions.FUNCTIONS.select_point("select_all_type",
                                              (drone.x, drone.y))
```

Now we have a bot that will build a spawning poll if we don't have one already.

0.6 Build Zerlings

With our Spawning Pool building bot, we can build some Zerlings. The first step is to select all the Larva on the screen:

```
# Selects all Larva on the screen
larvae = self.get_units_by_type(obs, units.Zerg.Larva)
if len(larvae) > 0:
    larva = random.choice(larvae)

    return actions.FUNCTIONS.select_point("select_all_type",
            (larva.x, larva.y))
```

After selecting the larva, we can create some Zerlings. Paste the following code above the previous block:

```
# Creates some Zerlings.
if self.unit_type_is_selected(obs, units.Zerg.Larva):
    if self.can_do(obs, actions.FUNCTIONS.Train_Zergling_quick.id):
        return actions.FUNCTIONS.Train_Zergling_quick("now")
```

To this point our bot is able to create Zerlings, however it's not so effected because our limited supply. In order to make a more effective strategy we need Overlords.

0.7 Spawn More Overlords

We can spawn an Overlord whenever we have no free supply and Larva is selected, so we need to modify what we implemented in the previous step so it looks like this:

```
# Creates some Zerlings.
if self.unit_type_is_selected(obs, units.Zerg.Larva):
    #We can spawn an Overlord if we have no free supply
    free_supply = (obs.observation.player.food_cap -
                   obs.observation.player.food_used)

    if free_supply == 0:
        if self.can_do(obs, actions.FUNCTIONS.Train_Overlord_quick.id):
            return actions.FUNCTIONS.Train_Overlord_quick("now")

        if self.can_do(obs, actions.FUNCTIONS.Train_Zergling_quick.id):
            return actions.FUNCTIONS.Train_Zergling_quick("now")
```

If we check our bot now, we will find that it produces Zerlings. The next step will be making all those Zerlings attack.

0.8 Attack