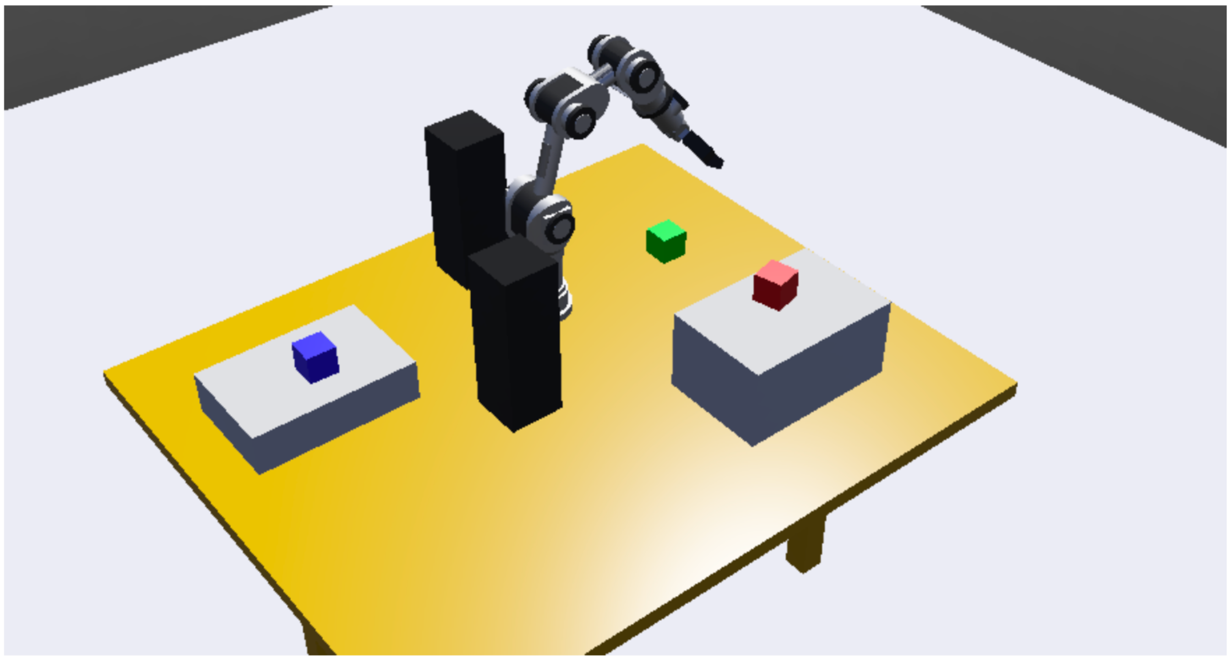
Problem:



Open the exercise file and open the ipr.wbt environment. This environment includes a robot called "Neuronics' IPR," three objects with red, green, and blue colors as targets, and two obstacle objects with yellow color. Train this robot using reinforcement learning so that initially it has no information about the positions of the target and obstacle objects, and it can only detect whether it has reached them or not upon collision with each of these objects (you can use sensors such as 2TouchSensor for this purpose). After training, this robot should move alternately towards the green, red, and blue objects in sequence (with minimal displacement and in the shortest time), and repeat this motion alternately forever. Additionally, the robot should not collide with any obstacles.

Several important points are mentioned in the question, which we will review initially:

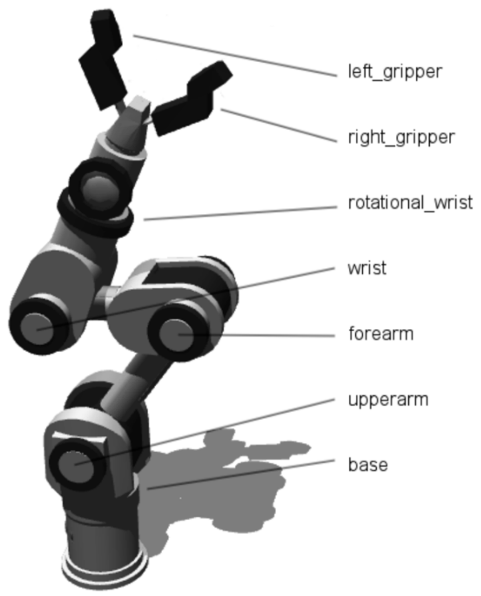
• The robot initially has no information about the positions of the target color cubes and obstacles, and we should not define their positions in the code for the robot.

• The robot should not collide with obstacles.

• The robot should move alternately towards the targets with minimal displacement and in the shortest time indefinitely.

Next, the solution approach is described step by step:

First, let's examine the robot. The robot uses seven different motors or arms for movement in space, which you can see in the image below:



IPR motors

Among these joints or arms, there are two arms corresponding to the right and left grippers. The IPR robot uses these joints to lift objects. However, since our goal is only to reach predefined targets, we don't need to change these joints during the robot's movement, and we can keep the robot's hands closed without altering the positions of those joints.

So, in the end, we have five joints that need to be initialized in the best possible way during the reinforcement learning process to achieve the desired problem formulation. Each of these joints will receive continuous values within specific ranges that are specific to themselves.

To solve this problem using a reinforcement learning approach, the state space or observation space along with the robot's action space must be defined in a specific manner:

After conducting the necessary investigations, we concluded that the robot's action space should be continuous and in ۵ dimensions. The specified value in each dimension continuously refers to the changeable position of each of the mentioned joints. By considering the robot's action space as continuous, we help it have smoother movements. Additionally, by achieving desirable learning, we can enable it to reach the goal with fewer movements, consume less energy, and avoid obstacles.

To define the robot's observation space, we can use the current position of each joint that can be changed along with the robot hand's spatial position. To obtain the current position of each joint, we can utilize sensors or encoders defined for each joint. To obtain the position of the robot's hand, we can use two approaches: solving direct kinematics or utilizing GPS. After the investigations conducted by the exercise-solving team, we concluded that using the embedded GPS in the robot's hand can be a better option.

Finally, after specifying the state and action spaces of the robot, we need to determine the reinforcement learning algorithm. Due to the continuous action space, we will use the PPO algorithm to solve this problem.

The Proximal Policy Optimization (PPO) algorithm is a reinforcement learning algorithm used for training artificial intelligence agents in dynamic environments. The main goal of this algorithm is to provide an optimization method for policy optimization, which offers improvements over previous algorithms such as Trust Region Policy Optimization (TRPO). PPO uses a clipping function to limit large changes in the policy, ensuring more stable updates and preventing sudden increases or oscillations in parameter changes.

In PPO, the agent aims to learn an optimal policy that maximizes the total cumulative reward in any given state. To achieve this, the algorithm utilizes a surrogate objective function called the "clipped surrogate objective," which allows policy updates while restricting significant changes in the policy. This enables the agent to improve its performance in the environment while avoiding excessive fluctuations during training.

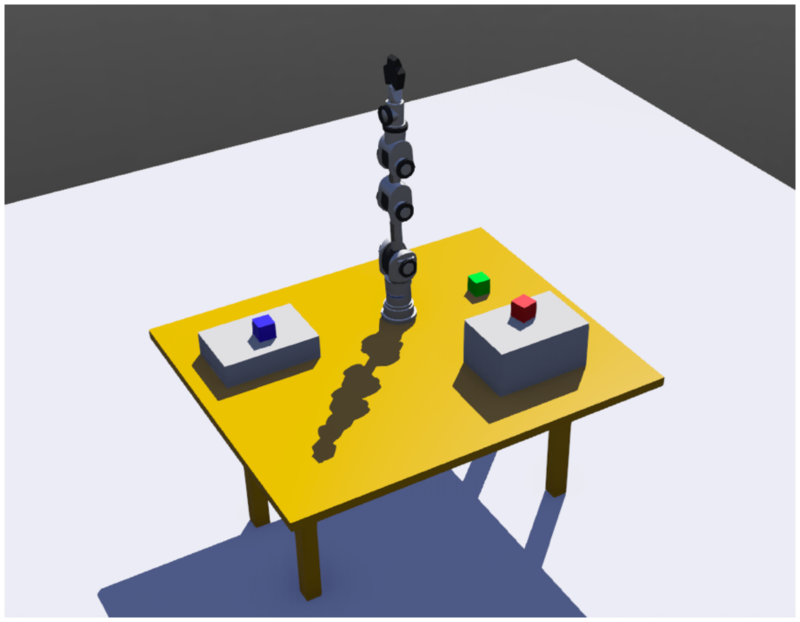
In the first stage, the agent evaluates a new objective function by comparing the probabilities of actions in the new and old policies. In the second stage, the policy parameters are updated based on this objective function to enable the agent to achieve better performance in the environment. PPO is a suitable algorithm for problems with continuous action spaces.

In comparison to the DQN algorithm, which is value-based, PPO is a policy-based algorithm that focuses on approximate policy optimization. Additionally, DQN is typically used for problems with discrete action spaces, while PPO is suitable for problems with continuous action spaces.

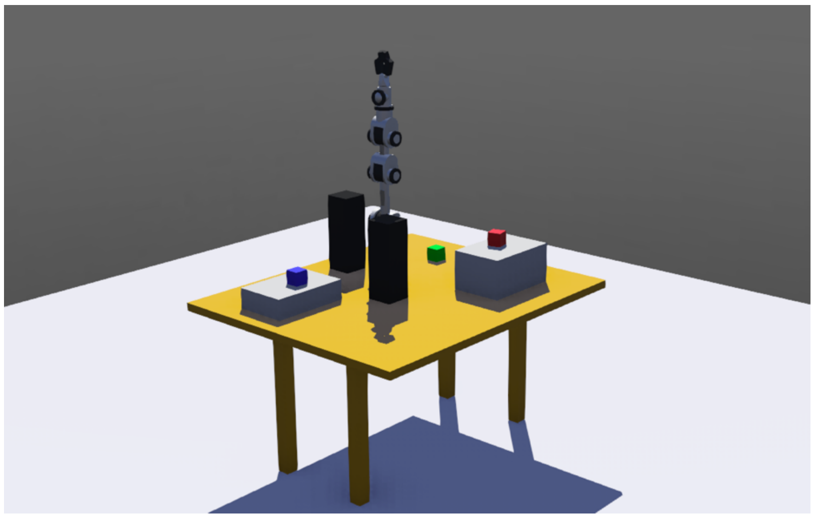
Stable Baselines۳ is a popular Python library for implementing reinforcement learning algorithms. It includes various implementations of reinforcement learning algorithms such as DQN, A۲C (Advantage Actor-Critic), PPO, and more. Stable Baselines۳ provides useful tools for training, evaluating, and utilizing reinforcement learning models, aiding in the development and testing of reinforcement learning algorithms. In this project, the aim is to utilize this library for the PPO algorithm.

In solving this problem, we intend to use a Curriculum Learning approach. In this approach, the robot carries out the learning process from easy to difficult, initially removing obstacles from the environment to learn how to identify and reach goals. Subsequently, after relative learning in an obstacle-free environment, obstacles are introduced to teach the robot how to navigate and avoid them.

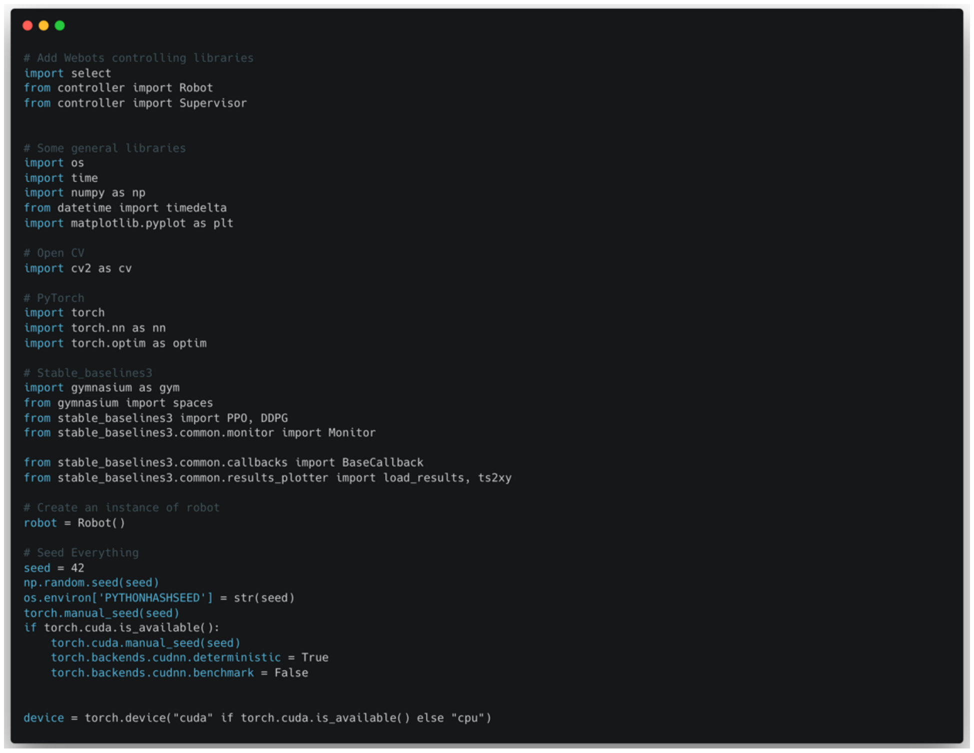
The robot's environment in the first stage is the training and learning of the robot to reach goals in easy conditions:



The robot's environment in the second stage involves completing the training and learning process of escaping obstacles in more challenging conditions:



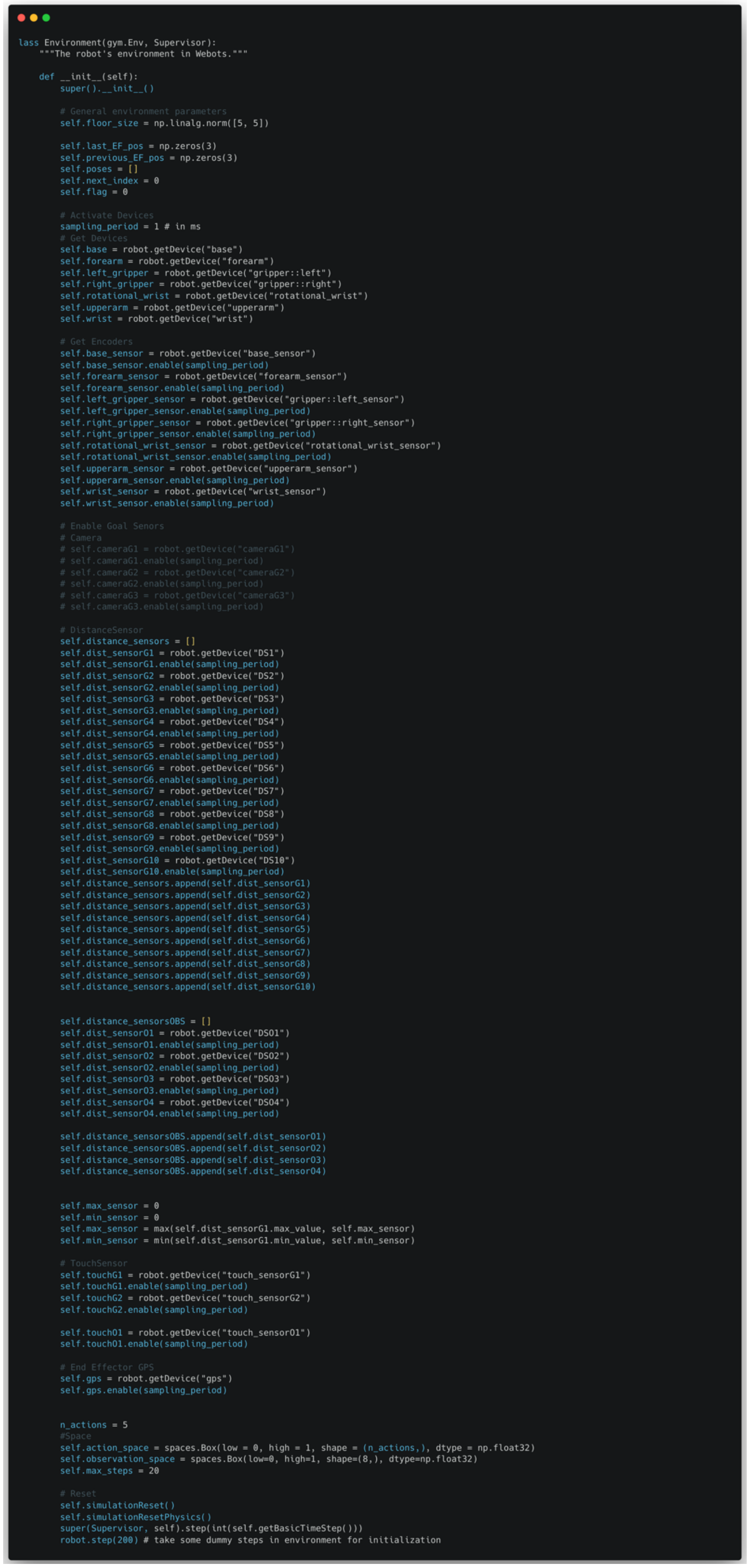
After getting familiar with the solution approach to this problem in the previous section, we describe the implemented code in more detail to examine each part thoroughly:



```python
# Add Webots controlling libraries
import select
from controller import Robot
from controller import Supervisor


# Some general libraries
import os
import time
import numpy as np
from datetime import timedelta
import matplotlib.pyplot as plt

# Open CV
import cv2 as cv

# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim

# Stable_baselines3
import gymnasium as gym
from gymnasium import spaces
from stable_baselines3 import PPO, DDPG
from stable_baselines3.common.monitor import Monitor

from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.results_plotter import load_results, ts2xy

# Create an instance of robot
robot = Robot()

# Seed Everything
seed = 42
np.random.seed(seed)
os.environ['PYTHONHASHSEED'] = str(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

First, we need to import the required libraries in the robot's controller file to utilize the functions available in them for implementing the problem's solution. We use the Supervisor library to access additional information about the agent's environment, which facilitates the training process. We create an instance of the robot.

To eliminate the randomness of the variables mentioned in the code in each run, we define a specific seed to generate the random state. This ensures that in each run, we observe the changes in the agent's learning solely based on our hyperparameter modifications. This random state follows a specific distribution in each run. We define a variable related to the device, so that if CUDA is available, it utilizes it; otherwise, it considers CPU for performing agent computations.



```
lass Environment(gym.Env, Supervisor):
    """The robot's environment in Webots."""

    def __init__(self):
        super().__init__()

        # General environment parameters
        self.floor_size = np.linalg.norm([5, 5])

        self.last_EF_pos = np.zeros(3)
        self.previous_EF_pos = np.zeros(3)
        self.poses = []
        self.next_index = 0
        self.flag = 0

        # Activate Devices
        sampling_period = 1 # in ms
        # Get Devices
        self.base = robot.getDevice("base")
        self.forearm = robot.getDevice("forearm")
        self.left_gripper = robot.getDevice("gripper::left")
        self.right_gripper = robot.getDevice("gripper::right")
        self.rotational_wrist = robot.getDevice("rotational_wrist")
        self.upperarm = robot.getDevice("upperarm")
        self.wrist = robot.getDevice("wrist")

        # Get Encoders
        self.base_sensor = robot.getDevice("base_sensor")
        self.base_sensor.enable(sampling_period)
        self.forearm_sensor = robot.getDevice("forearm_sensor")
        self.forearm_sensor.enable(sampling_period)
        self.left_gripper_sensor = robot.getDevice("gripper::left_sensor")
        self.left_gripper_sensor.enable(sampling_period)
        self.right_gripper_sensor = robot.getDevice("gripper::right_sensor")
        self.right_gripper_sensor.enable(sampling_period)
        self.rotational_wrist_sensor = robot.getDevice("rotational_wrist_sensor")
        self.rotational_wrist_sensor.enable(sampling_period)
        self.upperarm_sensor = robot.getDevice("upperarm_sensor")
        self.upperarm_sensor.enable(sampling_period)
        self.wrist_sensor = robot.getDevice("wrist_sensor")
        self.wrist_sensor.enable(sampling_period)

        # Enable Goal Senors
        # Camera
        # self.cameraG1 = robot.getDevice("cameraG1")
        # self.cameraG1.enable(sampling_period)
        # self.cameraG2 = robot.getDevice("cameraG2")
        # self.cameraG2.enable(sampling_period)
        # self.cameraG3 = robot.getDevice("cameraG3")
        # self.cameraG3.enable(sampling_period)

        # DistanceSensor
        self.distance_sensors = []
        self.dist_sensorG1 = robot.getDevice("DS1")
        self.dist_sensorG1.enable(sampling_period)
        self.dist_sensorG2 = robot.getDevice("DS2")
        self.dist_sensorG2.enable(sampling_period)
        self.dist_sensorG3 = robot.getDevice("DS3")
        self.dist_sensorG3.enable(sampling_period)
        self.dist_sensorG4 = robot.getDevice("DS4")
        self.dist_sensorG4.enable(sampling_period)
        self.dist_sensorG5 = robot.getDevice("DS5")
        self.dist_sensorG5.enable(sampling_period)
        self.dist_sensorG6 = robot.getDevice("DS6")
        self.dist_sensorG6.enable(sampling_period)
        self.dist_sensorG7 = robot.getDevice("DS7")
        self.dist_sensorG7.enable(sampling_period)
        self.dist_sensorG8 = robot.getDevice("DS8")
        self.dist_sensorG8.enable(sampling_period)
        self.dist_sensorG9 = robot.getDevice("DS9")
        self.dist_sensorG9.enable(sampling_period)
        self.dist_sensorG10 = robot.getDevice("DS10")
        self.dist_sensorG10.enable(sampling_period)
        self.distance_sensors.append(self.dist_sensorG1)
        self.distance_sensors.append(self.dist_sensorG2)
        self.distance_sensors.append(self.dist_sensorG3)
        self.distance_sensors.append(self.dist_sensorG4)
        self.distance_sensors.append(self.dist_sensorG5)
        self.distance_sensors.append(self.dist_sensorG6)
        self.distance_sensors.append(self.dist_sensorG7)
        self.distance_sensors.append(self.dist_sensorG8)
        self.distance_sensors.append(self.dist_sensorG9)
        self.distance_sensors.append(self.dist_sensorG10)


        self.distance_sensorsOBS = []
        self.dist_sensor01 = robot.getDevice("DS01")
        self.dist_sensor01.enable(sampling_period)
        self.dist_sensor02 = robot.getDevice("DS02")
        self.dist_sensor02.enable(sampling_period)
        self.dist_sensor03 = robot.getDevice("DS03")
        self.dist_sensor03.enable(sampling_period)
        self.dist_sensor04 = robot.getDevice("DS04")
        self.dist_sensor04.enable(sampling_period)

        self.distance_sensorsOBS.append(self.dist_sensor01)
        self.distance_sensorsOBS.append(self.dist_sensor02)
        self.distance_sensorsOBS.append(self.dist_sensor03)
        self.distance_sensorsOBS.append(self.dist_sensor04)


        self.max_sensor = 0
        self.min_sensor = 0
        self.max_sensor = max(self.dist_sensorG1.max_value, self.max_sensor)
        self.min_sensor = min(self.dist_sensorG1.min_value, self.min_sensor)

        # TouchSensor
        self.touchG1 = robot.getDevice("touch_sensorG1")
        self.touchG1.enable(sampling_period)
        self.touchG2 = robot.getDevice("touch_sensorG2")
        self.touchG2.enable(sampling_period)

        self.touch01 = robot.getDevice("touch_sensor01")
        self.touch01.enable(sampling_period)

        # End Effector GPS
        self.gps = robot.getDevice("gps")
        self.gps.enable(sampling_period)


        n_actions = 5
        #Space
        self.action_space = spaces.Box(low = 0, high = 1, shape = (n_actions,), dtype = np.float32)
        self.observation_space = spaces.Box(low=0, high=1, shape=(8,), dtype=np.float32)
        self.max_steps = 20

        # Reset
        self.simulationReset()
        self.simulationResetPhysics()
        super(Supervisor, self).step(int(self.getBasicTimeStep()))
        robot.step(200) # take some dummy steps in environment for initialization
```

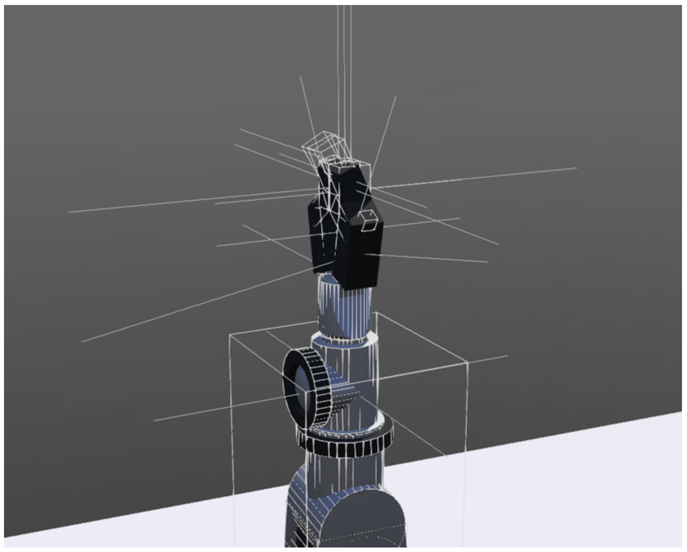
In this code snippet, the environment class attempts to initialize all variables related to the environment and the agent. These variables include the size of the field, definition of robot's motors or joints, definition of internal and external sensors of the robot, and their activation. As seen in the above code snippet, arrays and variables are initially considered to store the coordinates obtained by the robot during the learning stages, which will be used later. Then, we proceed to define the joints mentioned in the first part and discuss their sensors. After that, we define the camera (which is not used in this solution, but there may be possible solutions that utilize a camera) and the distance sensors related to measuring the distance to the target and obstacles. Finally, we define the collision sensors and GPS. All these sensors require activation after definition, which we activate by assigning them values for a short period of time.

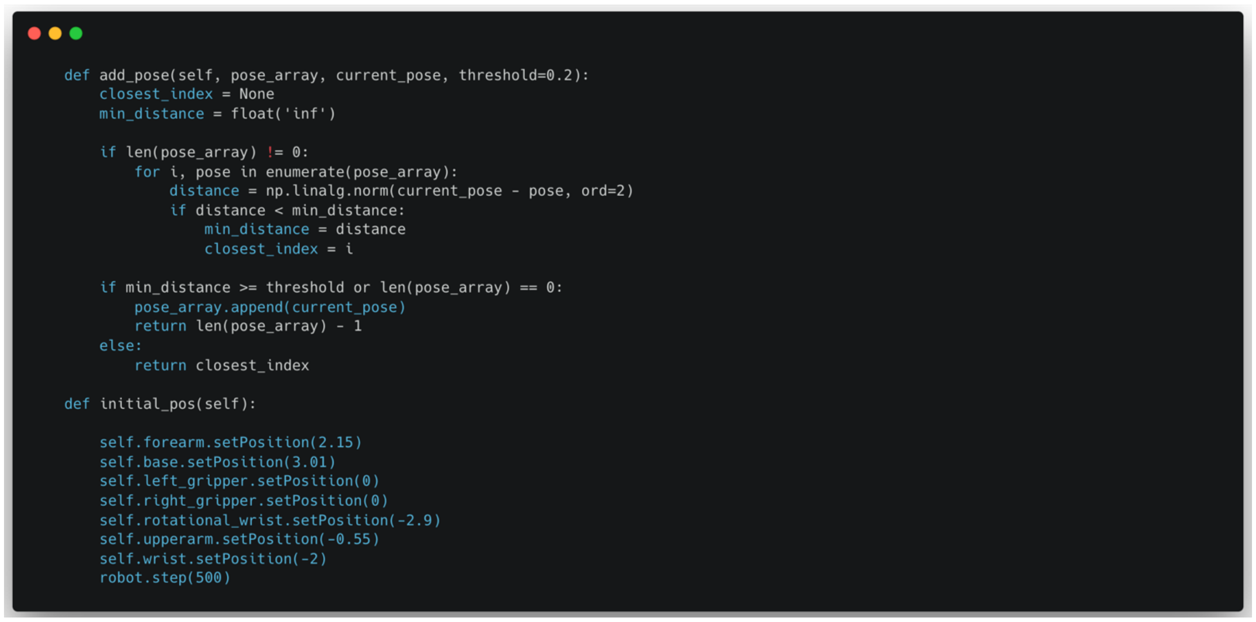
Finally, we define the action space and the observable space of the robot.

Robot's action space: As mentioned in the first part, the robot's action space involves continuously changing the positions of the 5 joints. Therefore, we define the robot's action space in 5 dimensions as continuous values ranging from 0 to 1. During execution, the value of each dimension corresponding to the positional state of a joint is scaled according to its range of values.

Robot's observable space: The observable space of the robot includes the current positions of these 5 joints along with 3 dimensions specifying the position of the robot's hand. Ultimately, the observable space of the robot is defined in 8 dimensions as continuous values.

Representation of external sensors of the robot:

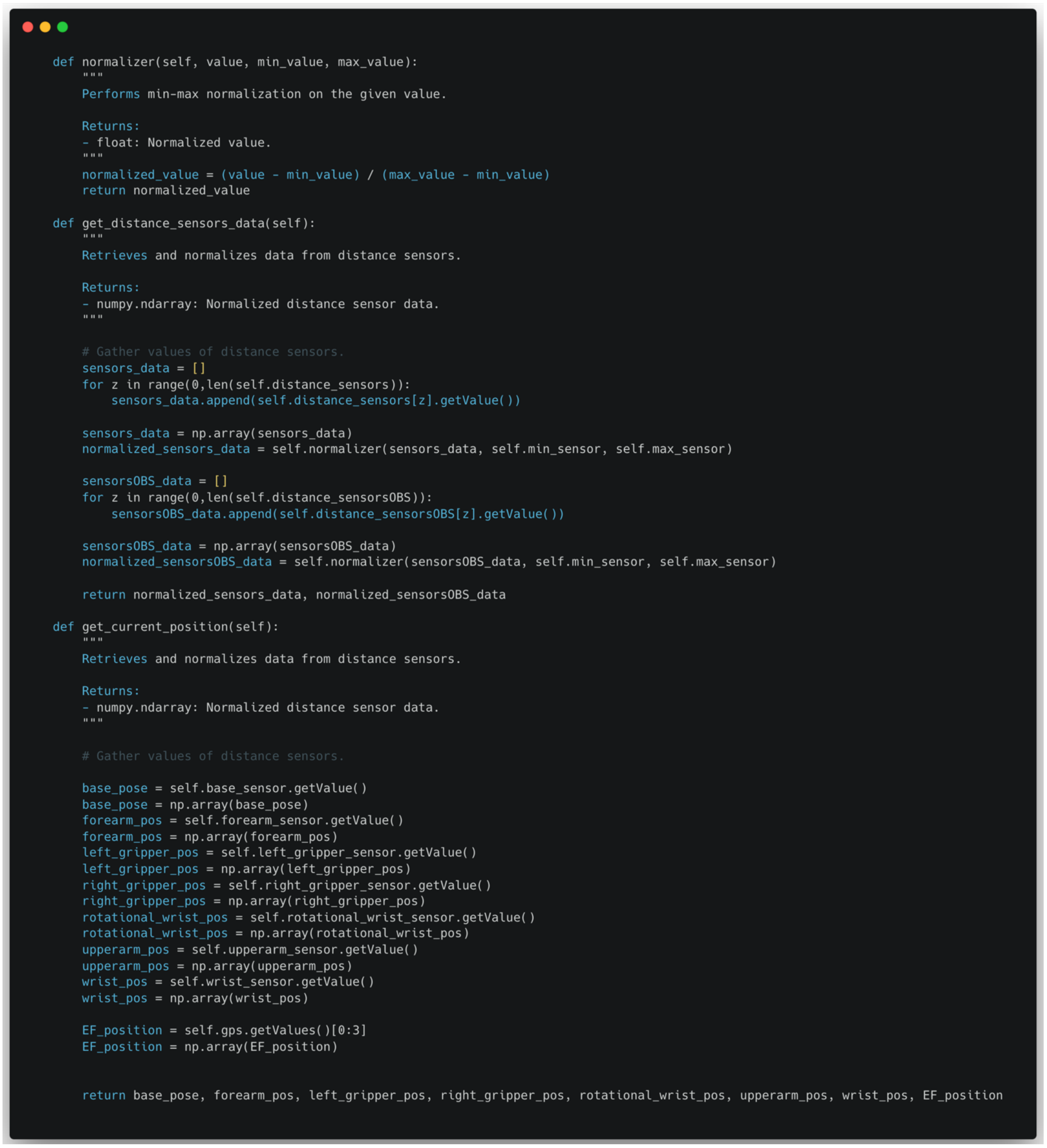




```python
def add_pose(self, pose_array, current_pose, threshold=0.2):
    closest_index = None
    min_distance = float('inf')

    if len(pose_array) != 0:
        for i, pose in enumerate(pose_array):
            distance = np.linalg.norm(current_pose - pose, ord=2)
            if distance < min_distance:
                min_distance = distance
                closest_index = i

    if min_distance >= threshold or len(pose_array) == 0:
        pose_array.append(current_pose)
        return len(pose_array) - 1
    else:
        return closest_index

def initial_pos(self):

    self.forearm.setPosition(2.15)
    self.base.setPosition(3.01)
    self.left_gripper.setPosition(0)
    self.right_gripper.setPosition(0)
    self.rotational_wrist.setPosition(-2.9)
    self.upperarm.setPosition(-0.55)
    self.wrist.setPosition(-2)
    robot.step(500)
```

Using the `add_pose` function, we measure the collision position of the robot's hand with the previous collision positions. If the current collision point is further away from the other points than a certain threshold, we define that collision point as a new target point and then return its index in the array of positions. If the current collision point is close to one of the recorded points in the array, meaning we collided with a previous target again, we return the index of the closest recorded collision point in the output. This function is used to create a loop in reaching the targets. You will see the use of this function in the rewards section for the robot.

Using the `initial_pos` function, we initialize the robot's joints to the fully upright position.

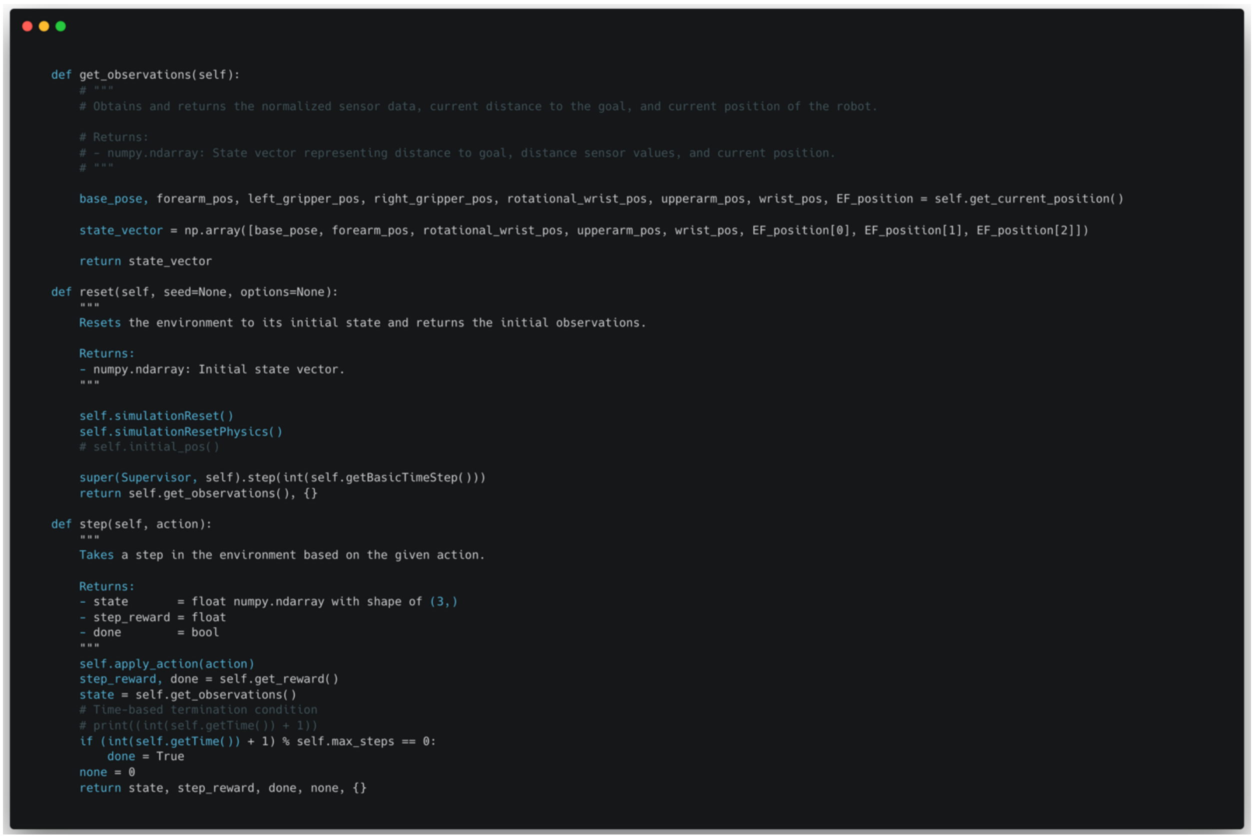


```python
def normalizer(self, value, min_value, max_value):
    """
    Performs min-max normalization on the given value.

    Returns:
    - float: Normalized value.
    """
    normalized_value = (value - min_value) / (max_value - min_value)
    return normalized_value

def get_distance_sensors_data(self):
    """
    Retrieves and normalizes data from distance sensors.

    Returns:
    - numpy.ndarray: Normalized distance sensor data.
    """

    # Gather values of distance sensors.
    sensors_data = []
    for z in range(0,len(self.distance_sensors)):
        sensors_data.append(self.distance_sensors[z].getValue())

    sensors_data = np.array(sensors_data)
    normalized_sensors_data = self.normalizer(sensors_data, self.min_sensor, self.max_sensor)

    sensorsOBS_data = []
    for z in range(0,len(self.distance_sensorsOBS)):
        sensorsOBS_data.append(self.distance_sensorsOBS[z].getValue())

    sensorsOBS_data = np.array(sensorsOBS_data)
    normalized_sensorsOBS_data = self.normalizer(sensorsOBS_data, self.min_sensor, self.max_sensor)

    return normalized_sensors_data, normalized_sensorsOBS_data

def get_current_position(self):
    """
    Retrieves and normalizes data from distance sensors.

    Returns:
    - numpy.ndarray: Normalized distance sensor data.
    """

    # Gather values of distance sensors.

    base_pose = self.base_sensor.getValue()
    base_pose = np.array(base_pose)
    forearm_pos = self.forearm_sensor.getValue()
    forearm_pos = np.array(forearm_pos)
    left_gripper_pos = self.left_gripper_sensor.getValue()
    left_gripper_pos = np.array(left_gripper_pos)
    right_gripper_pos = self.right_gripper_sensor.getValue()
    right_gripper_pos = np.array(right_gripper_pos)
    rotational_wrist_pos = self.rotational_wrist_sensor.getValue()
    rotational_wrist_pos = np.array(rotational_wrist_pos)
    upperarm_pos = self.upperarm_sensor.getValue()
    upperarm_pos = np.array(upperarm_pos)
    wrist_pos = self.wrist_sensor.getValue()
    wrist_pos = np.array(wrist_pos)

    EF_position = self.gps.getValues()[0:3]
    EF_position = np.array(EF_position)


    return base_pose, forearm_pos, left_gripper_pos, right_gripper_pos, rotational_wrist_pos, upperarm_pos, wrist_pos, EF_position
```

Using the `normalizer` function, we normalize the values of the distance sensors.

Using the `get_distance_sensors_data` function, we obtain the values received by the distance sensors related to the targets and obstacles. After normalization, we return the values of the distance to the target and obstacles separately in the output.

Using the `get_current_position` function, we obtain the current positions of each joint from their internal sensors. We also receive the spatial position of the robot's hand using the GPS sensor. Then, we return all the received positions in the current moment in the output



```python
def get_observations(self):
    # """
    # Obtains and returns the normalized sensor data, current distance to the goal, and current position of the robot.

    # Returns:
    # - numpy.ndarray: State vector representing distance to goal, distance sensor values, and current position.
    # """

    base_pose, forearm_pos, left_gripper_pos, right_gripper_pos, rotational_wrist_pos, upperarm_pos, wrist_pos, EF_position = self.get_current_position()

    state_vector = np.array([base_pose, forearm_pos, rotational_wrist_pos, upperarm_pos, wrist_pos, EF_position[0], EF_position[1], EF_position[2]])

    return state_vector

def reset(self, seed=None, options=None):
    """
    Resets the environment to its initial state and returns the initial observations.

    Returns:
    - numpy.ndarray: Initial state vector.
    """

    self.simulationReset()
    self.simulationResetPhysics()
    # self.initial_pos()

    super(Supervisor, self).step(int(self.getBasicTimeStep()))
    return self.get_observations(), {}

def step(self, action):
    """
    Takes a step in the environment based on the given action.

    Returns:
    - state       = float numpy.ndarray with shape of (3,)
    - step_reward = float
    - done        = bool
    """
    self.apply_action(action)
    step_reward, done = self.get_reward()
    state = self.get_observations()
    # Time-based termination condition
    # print((int(self.getTime()) + 1))
    if (int(self.getTime()) + 1) % self.max_steps == 0:
        done = True
    none = 0
    return state, step_reward, done, none, {}
```

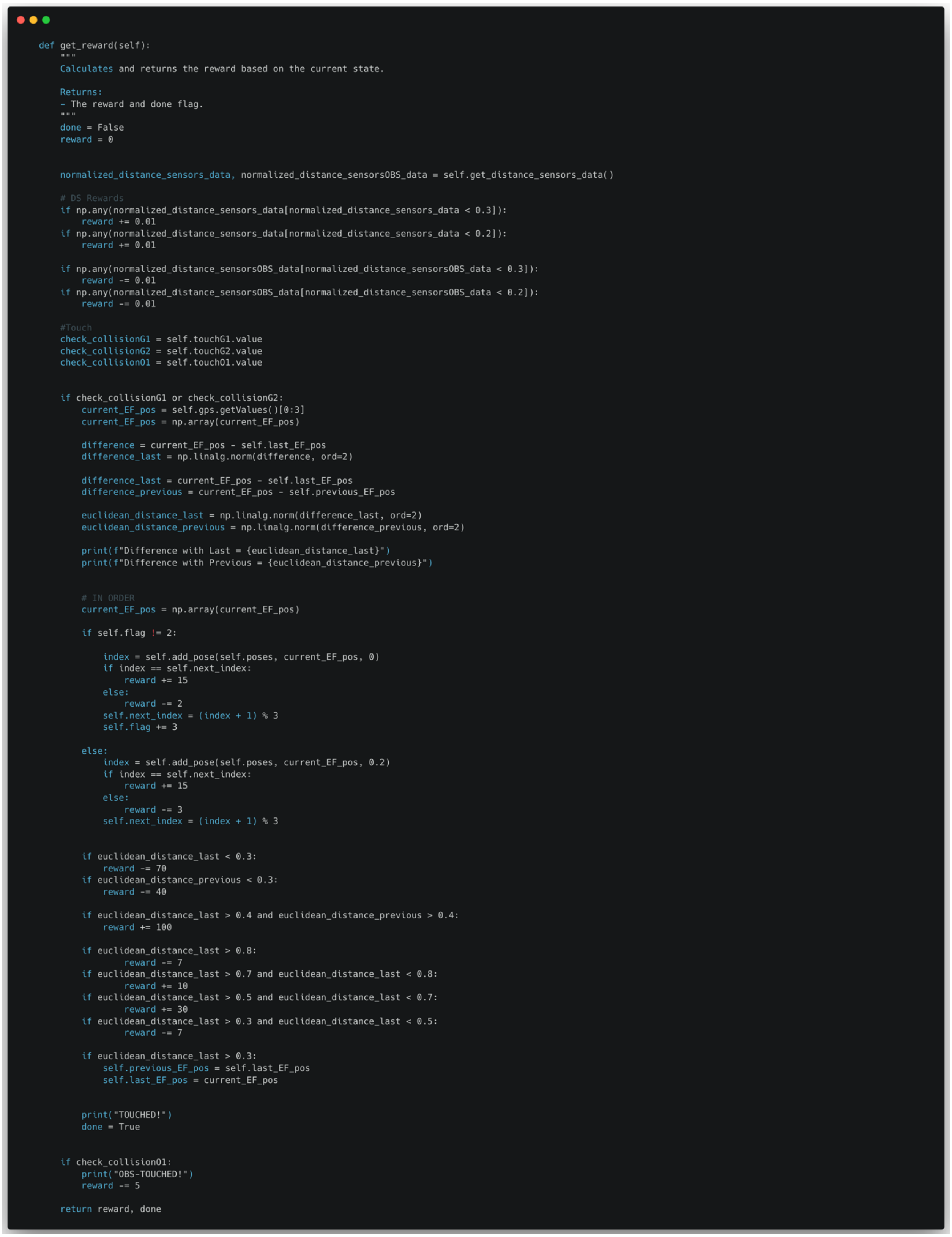
Using the `get_observation` function, we form the observable space of the robot in the output. Within this function, we use the `get_current_position` function to obtain the current position of the robot, which includes the positions of all joints and the spatial position of the robot's hand. We return all of them except for the hand joints in the output, as we mentioned earlier that these joints do not change and are not our target. Therefore, we do not include them in the observable and operational space of the robot.

Using the `reset` function with the help of the Supervisor, we can reset the environment and return everything to the initial conditions, preparing the environment for a fresh start of the robot in the initial conditions. We also consider the basic time of the environment. Then, we can use the `get_observation` function for the initial observation in the environment.

Using the `step` function, we apply the required action that the robot wants to perform. We calculate the reward or penalty for performing that action and then calculate the robot's observations after performing that action to observe the changes in the environment. Then, we check if we have reached the maximum number of allowed steps for the robot. If we have reached it, the executed command will be issued.

In the `reward` function, we aim to set the rewards and penalties for the robot's actions according to our problem-solving objectives. The robot, based on the specified values in this function, will either be penalized or rewarded for the task it has performed to solve the problem. Therefore, the values specified in this function should be set optimally considering the solution and the nature of the problem.

As discussed in the first part, this problem is solved using a progressive learning approach. In the first stage of training, obstacles were removed for the agent to learn reaching the goals. Eventually, with the reintroduction of obstacles, the agent previously trained in the first stage is loaded to observe the learning process with the help of rewards or penalties for avoiding or colliding with obstacles. In this section, we will examine the final rewards and penalties.



```python
def get_reward(self):
    """
    Calculates and returns the reward based on the current state.

    Returns:
    - The reward and done flag.
    """
    done = False
    reward = 0


    normalized_distance_sensors_data, normalized_distance_sensorsOBS_data = self.get_distance_sensors_data()

    # DS Rewards
    if np.any(normalized_distance_sensors_data[normalized_distance_sensors_data < 0.3]):
        reward += 0.01
    if np.any(normalized_distance_sensors_data[normalized_distance_sensors_data < 0.2]):
        reward += 0.01

    if np.any(normalized_distance_sensorsOBS_data[normalized_distance_sensorsOBS_data < 0.3]):
        reward -= 0.01
    if np.any(normalized_distance_sensorsOBS_data[normalized_distance_sensorsOBS_data < 0.2]):
        reward -= 0.01

    #Touch
    check_collisionG1 = self.touchG1.value
    check_collisionG2 = self.touchG2.value
    check_collisionO1 = self.touchO1.value


    if check_collisionG1 or check_collisionG2:
        current_EF_pos = self.gps.getValues()[0:3]
        current_EF_pos = np.array(current_EF_pos)

        difference = current_EF_pos - self.last_EF_pos
        difference_last = np.linalg.norm(difference, ord=2)

        difference_last = current_EF_pos - self.last_EF_pos
        difference_previous = current_EF_pos - self.previous_EF_pos

        euclidean_distance_last = np.linalg.norm(difference_last, ord=2)
        euclidean_distance_previous = np.linalg.norm(difference_previous, ord=2)

        print(f"Difference with Last = {euclidean_distance_last}")
        print(f"Difference with Previous = {euclidean_distance_previous}")


        # IN ORDER
        current_EF_pos = np.array(current_EF_pos)

        if self.flag != 2:

            index = self.add_pose(self.poses, current_EF_pos, 0)
            if index == self.next_index:
                reward += 15
            else:
                reward -= 2
            self.next_index = (index + 1) % 3
            self.flag += 3

        else:
            index = self.add_pose(self.poses, current_EF_pos, 0.2)
            if index == self.next_index:
                reward += 15
            else:
                reward -= 3
            self.next_index = (index + 1) % 3


        if euclidean_distance_last < 0.3:
            reward -= 70
        if euclidean_distance_previous < 0.3:
            reward -= 40

        if euclidean_distance_last > 0.4 and euclidean_distance_previous > 0.4:
            reward += 100

        if euclidean_distance_last > 0.8:
                reward -= 7
        if euclidean_distance_last > 0.7 and euclidean_distance_last < 0.8:
                reward += 10
        if euclidean_distance_last > 0.5 and euclidean_distance_last < 0.7:
                reward += 30
        if euclidean_distance_last > 0.3 and euclidean_distance_last < 0.5:
                reward -= 7

        if euclidean_distance_last > 0.3:
            self.previous_EF_pos = self.last_EF_pos
            self.last_EF_pos = current_EF_pos


        print("TOUCHED!")
        done = True


    if check_collisionO1:
        print("OBS-TOUCHED!")
        reward -= 5

    return reward, done
```

In the section related to the distance sensor:

Using the `get_distance_sensors_data` function, we measure the current distance between the agent and the targets or obstacles. Using predefined threshold values, we reward the robot if it is close to the targets and penalize it if it is close to the obstacles. It needs to be reiterated that the robot has learned to approach the targets in the first stage, so the embedded distance sensors in the robot's hands always indicate the distance between the robot's hands and the targets.

At the moment of the robot's hand colliding with the targets:

Initially, we calculate the Euclidean distance between the current collision and the previous two collisions to use them for punishment and reward purposes.

Collision with targets and forming a loop:

To encourage the agent to collide with targets by forming a loop, we enter the collision coordinates into the array of coordinates using the `add_pose` function. After colliding with a target, the index of that target is determined in the array. Finally, if the agent collides with a target in the next collision with coordinates at an index one higher, we reward the robot, and if the robot breaks out of the loop and collides with targets outside the loop, we penalize it.

In the next iteration, if the current collision distance is less than a certain threshold compared to the previous collision, we penalize the robot. Additionally, if the current collision distance with the previous two collisions is also less than a certain threshold, we penalize the robot again, but this penalty should be less severe than the previous case because repetitive collision with a target is worse for us. Therefore, we penalize the consecutive repetitive collisions more.

In the next section, if the current collision distance with the previous collision and the previous two collisions is greater than a certain threshold, we reward the robot well to encourage it to form a loop.

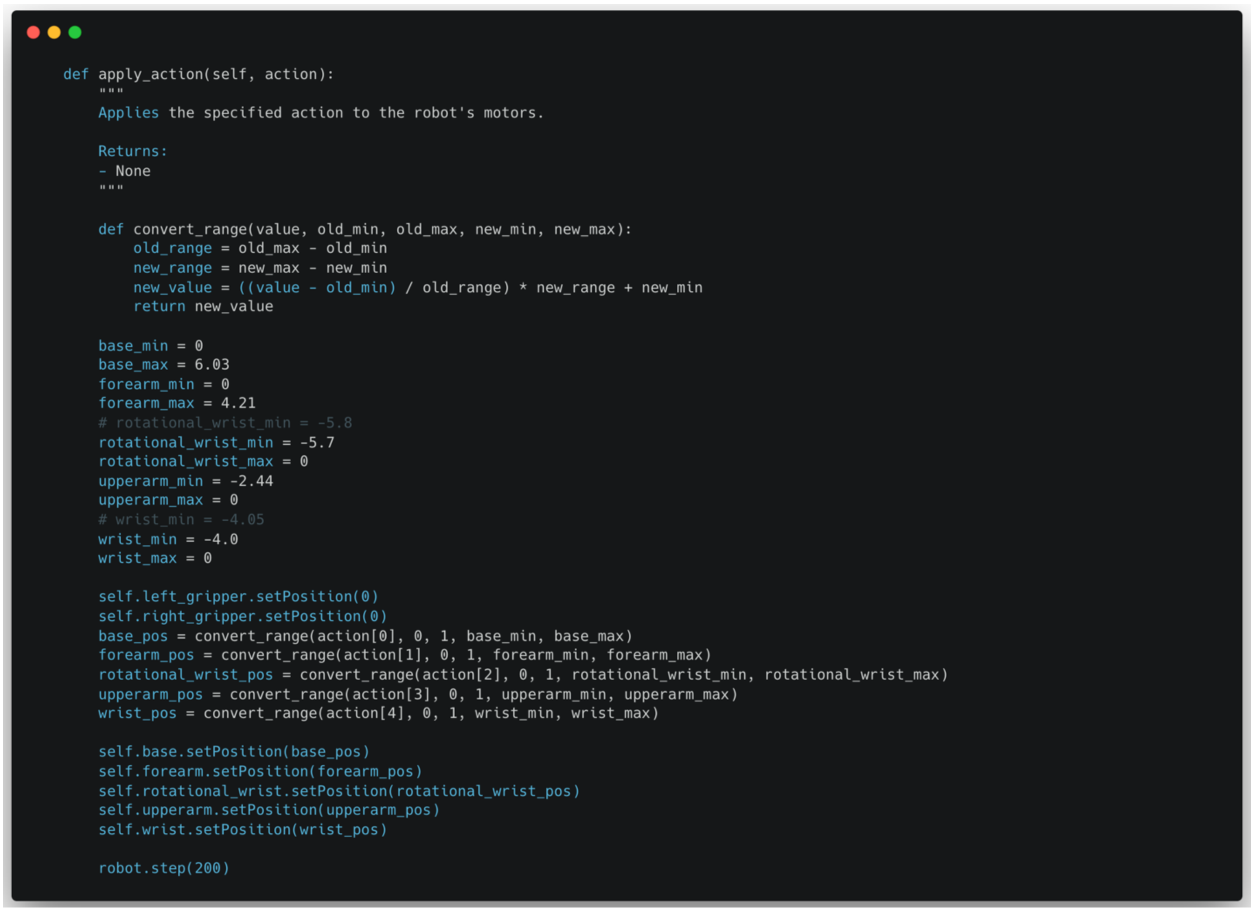
Finally, if the current collision distance between the robot and the previous collision is very small or very large, we penalize it. We don't want it to be a repetitive collision, nor do we want it to be too far away. But if the distance is average, we want to reward it to stay in the loop.

In the initial training stage in this section, to minimize the changes for the robot to reach the targets, we subtract the current state space from the previous state space. If the result of this subtraction is small, we reward the robot, and if it is large, we penalize it.

Collision with obstacles:

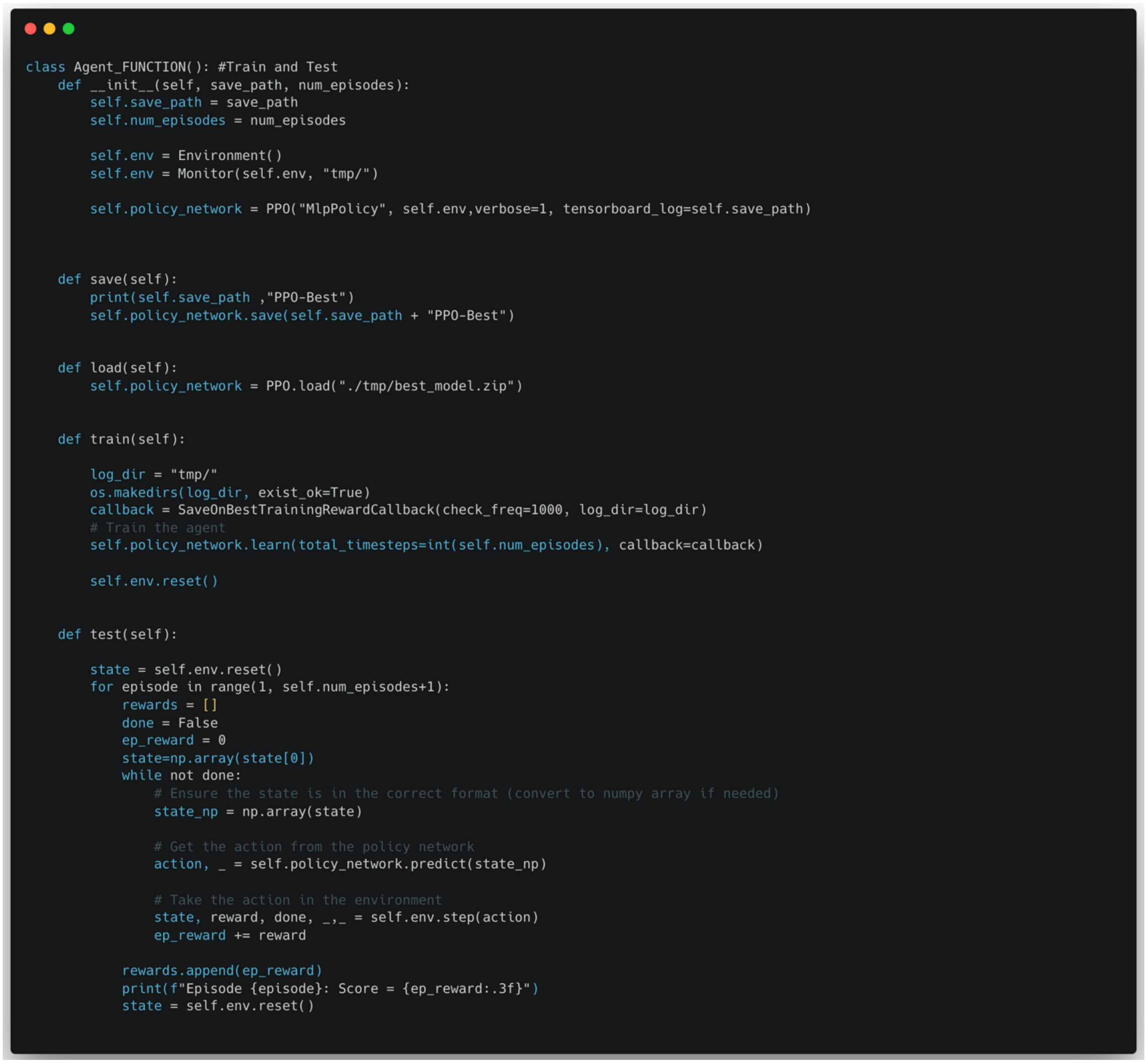
In the event of a collision with obstacles, we penalize the robot

.



```
def apply_action(self, action):
    """
    Applies the specified action to the robot's motors.

    Returns:
    - None
    """

    def convert_range(value, old_min, old_max, new_min, new_max):
        old_range = old_max - old_min
        new_range = new_max - new_min
        new_value = ((value - old_min) / old_range) * new_range + new_min
        return new_value

    base_min = 0
    base_max = 6.03
    forearm_min = 0
    forearm_max = 4.21
    # rotational_wrist_min = -5.8
    rotational_wrist_min = -5.7
    rotational_wrist_max = 0
    upperarm_min = -2.44
    upperarm_max = 0
    # wrist_min = -4.05
    wrist_min = -4.0
    wrist_max = 0

    self.left_gripper.setPosition(0)
    self.right_gripper.setPosition(0)
    base_pos = convert_range(action[0], 0, 1, base_min, base_max)
    forearm_pos = convert_range(action[1], 0, 1, forearm_min, forearm_max)
    rotational_wrist_pos = convert_range(action[2], 0, 1, rotational_wrist_min, rotational_wrist_max)
    upperarm_pos = convert_range(action[3], 0, 1, upperarm_min, upperarm_max)
    wrist_pos = convert_range(action[4], 0, 1, wrist_min, wrist_max)

    self.base.setPosition(base_pos)
    self.forearm.setPosition(forearm_pos)
    self.rotational_wrist.setPosition(rotational_wrist_pos)
    self.upperarm.setPosition(upperarm_pos)
    self.wrist.setPosition(wrist_pos)

    robot.step(200)
```

Using the `apply_action` function, we execute the predicted actions by the agent. In the previous section, we mentioned that in the action space of the robot, there are ۵ continuous dimensions ranging from ۰ to ۱. Each dimension corresponds to a position that each joint should change to. Therefore, after scaling each dimension within the range of joint positions using the `convert_range` function, we consider it as the position that the joint should change to.



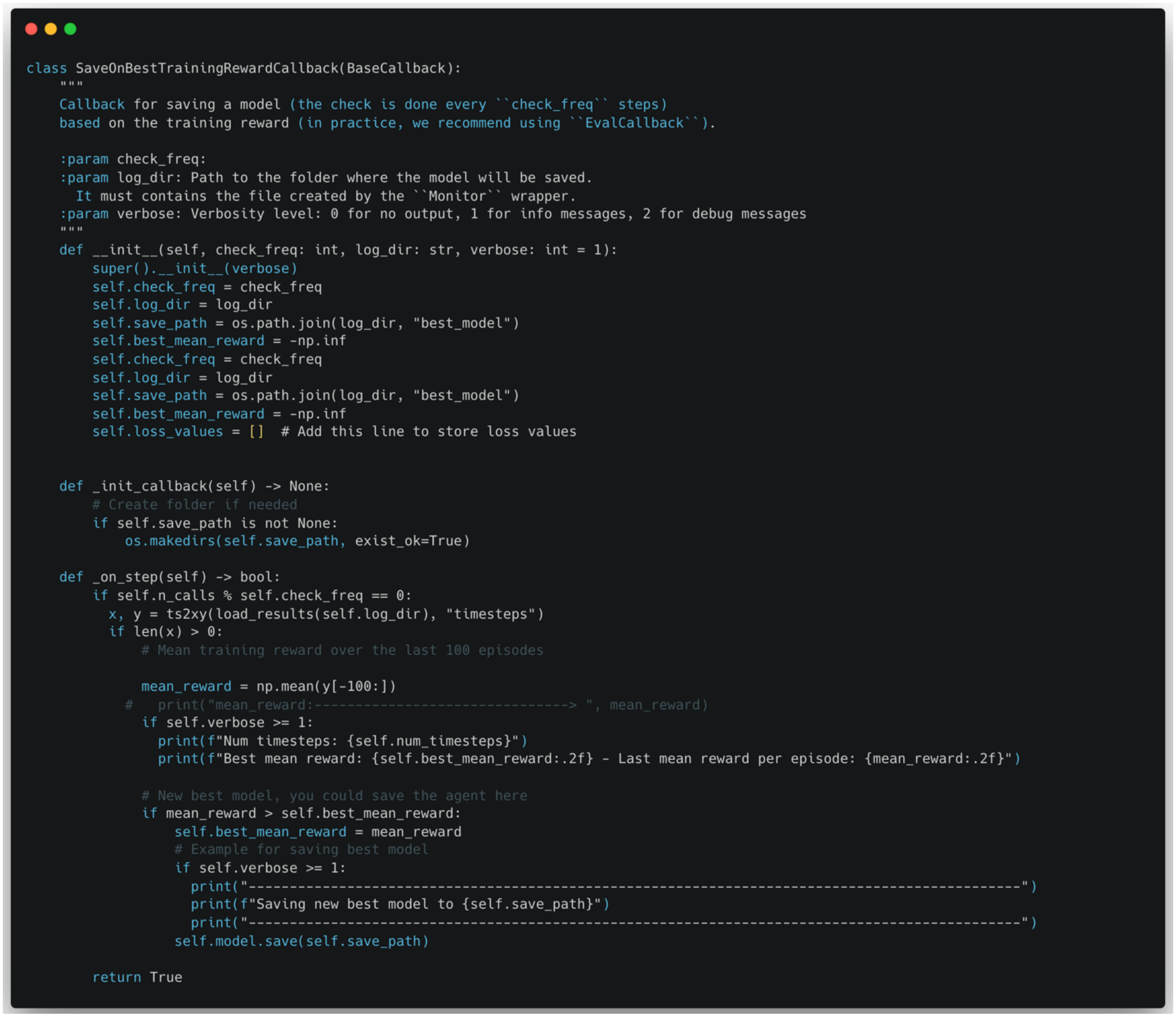
```
class Agent_FUNCTION(): #Train and Test
    def __init__(self, save_path, num_episodes):
        self.save_path = save_path
        self.num_episodes = num_episodes

        self.env = Environment()
        self.env = Monitor(self.env, "tmp/")

        self.policy_network = PPO("MlpPolicy", self.env,verbose=1, tensorboard_log=self.save_path)



    def save(self):
        print(self.save_path ,"PPO-Best")
        self.policy_network.save(self.save_path + "PPO-Best")


    def load(self):
        self.policy_network = PPO.load("./tmp/best_model.zip")


    def train(self):

        log_dir = "tmp/"
        os.makedirs(log_dir, exist_ok=True)
        callback = SaveOnBestTrainingRewardCallback(check_freq=1000, log_dir=log_dir)
        # Train the agent
        self.policy_network.learn(total_timesteps=int(self.num_episodes), callback=callback)

        self.env.reset()


    def test(self):

        state = self.env.reset()
        for episode in range(1, self.num_episodes+1):
            rewards = []
            done = False
            ep_reward = 0
            state=np.array(state[0])
            while not done:
                # Ensure the state is in the correct format (convert to numpy array if needed)
                state_np = np.array(state)

                # Get the action from the policy network
                action, _ = self.policy_network.predict(state_np)

                # Take the action in the environment
                state, reward, done, _,_ = self.env.step(action)
                ep_reward += reward

            rewards.append(ep_reward)
            print(f"Episode {episode}: Score = {ep_reward:.3f}")
            state = self.env.reset()
```

The class related to the agent's function, or Agent_FONCTION:

This class is implemented for decision-making by the robot based on its learning using reinforcement learning. In this class, the agent's policy function path along with the number of episodes is considered for learning or testing the agent. Initially, we introduce the agent's environment. To learn the robot based on its experiences, we use the PPO (Proximal Policy Optimization) reinforcement learning algorithm.

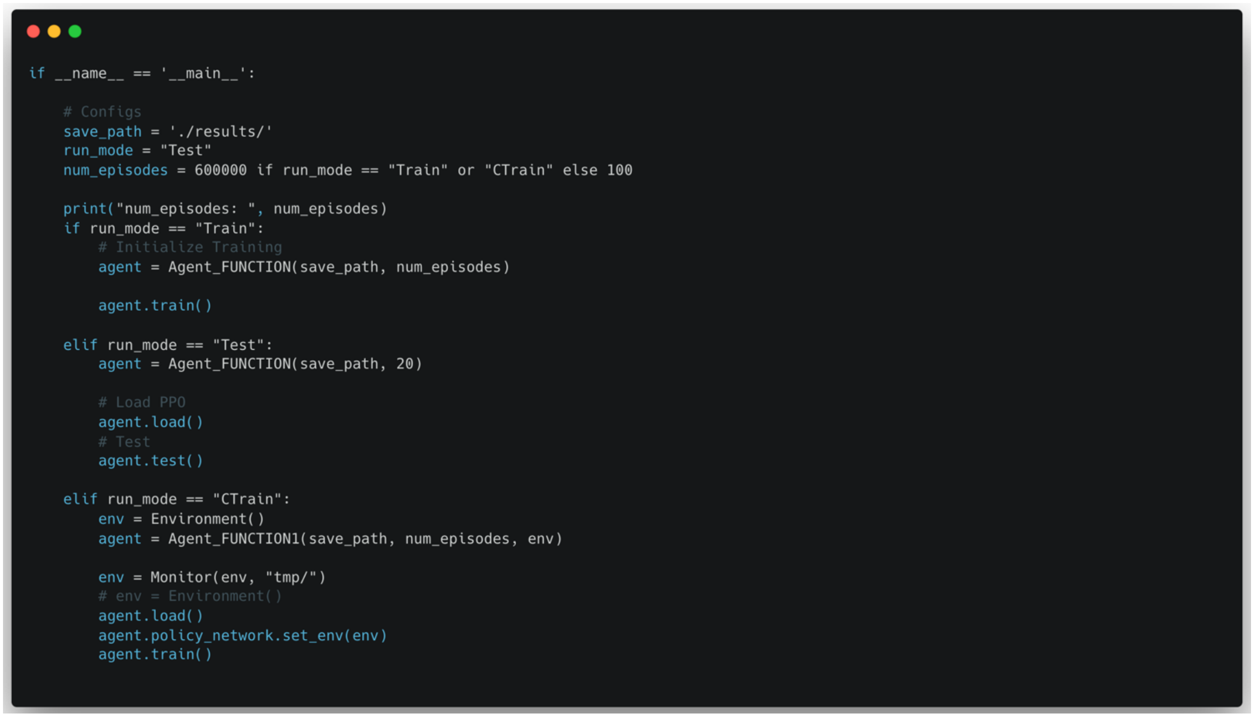
By using the `train` function, we start the learning process by training the policy network, which is the PPO algorithm. We also specify the total number of time steps required for learning. Using this function, the agent gains experience in the environment and learns through the PPO reinforcement learning algorithm to update the best policy function for the robot, so that it can achieve the best rewards during learning and testing.

Using the `test` function, the required actions at each step are predicted based on the learned policy function, and they are applied. Then, in each episode, the obtained reward value is calculated and applied.



```
class SaveOnBestTrainingRewardCallback(BaseCallback):
    """
    Callback for saving a model (the check is done every ``check_freq`` steps)
    based on the training reward (in practice, we recommend using ``EvalCallback``).

    :param check_freq:
    :param log_dir: Path to the folder where the model will be saved.
      It must contains the file created by the ``Monitor`` wrapper.
    :param verbose: Verbosity level: 0 for no output, 1 for info messages, 2 for debug messages
    """
    def __init__(self, check_freq: int, log_dir: str, verbose: int = 1):
        super().__init__(verbose)
        self.check_freq = check_freq
        self.log_dir = log_dir
        self.save_path = os.path.join(log_dir, "best_model")
        self.best_mean_reward = -np.inf
        self.check_freq = check_freq
        self.log_dir = log_dir
        self.save_path = os.path.join(log_dir, "best_model")
        self.best_mean_reward = -np.inf
        self.loss_values = []  # Add this line to store loss values


    def _init_callback(self) -> None:
        # Create folder if needed
        if self.save_path is not None:
            os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self) -> bool:
        if self.n_calls % self.check_freq == 0:
          x, y = ts2xy(load_results(self.log_dir), "timesteps")
          if len(x) > 0:
              # Mean training reward over the last 100 episodes

              mean_reward = np.mean(y[-100:])
            #   print("mean_reward:-------------------------------> ", mean_reward)
              if self.verbose >= 1:
                print(f"Num timesteps: {self.num_timesteps}")
                print(f"Best mean reward: {self.best_mean_reward:.2f} - Last mean reward per episode: {mean_reward:.2f}")

              # New best model, you could save the agent here
              if mean_reward > self.best_mean_reward:
                  self.best_mean_reward = mean_reward
                  # Example for saving best model
                  if self.verbose >= 1:
                    print("----------------------------------------------------------------------------------------------")
                    print(f"Saving new best model to {self.save_path}")
                    print("----------------------------------------------------------------------------------------------")
                  self.model.save(self.save_path)

        return True
```

Using the above code snippet that utilizes the Monitor library, we calculate the average rewards obtained in a specified number of recent episodes. If the average rewards received by the robot at each step increase, we save the model. This function helps us perform the learning process with a large number of episodes and ensure that the best model is saved during the learning process.



```python
if __name__ == '__main__':

    # Configs
    save_path = './results/'
    run_mode = "Test"
    num_episodes = 600000 if run_mode == "Train" or "CTrain" else 100

    print("num_episodes: ", num_episodes)
    if run_mode == "Train":
        # Initialize Training
        agent = Agent_FUNCTION(save_path, num_episodes)

        agent.train()

    elif run_mode == "Test":
        agent = Agent_FUNCTION(save_path, 20)

        # Load PPO
        agent.load()
        # Test
        agent.test()

    elif run_mode == "CTrain":
        env = Environment()
        agent = Agent_FUNCTION1(save_path, num_episodes, env)

        env = Monitor(env, "tmp/")
        # env = Environment()
        agent.load()
        agent.policy_network.set_env(env)
        agent.train()
```

In the main function, the desired number of episodes for learning and testing the agent is initialized.

Finally, we need to consider the execution mode of the code. In the first stage, the code is executed in the Train mode to let the agent learn to reach its goals. After a reasonable level of learning, we save the model. Then, in the second stage, we execute the code in the CTrain mode. In this mode, the saved model is loaded, and the learning process continues with the introduction of obstacles.

Finally, during the agent's testing phase, the code is executed in the Test mode