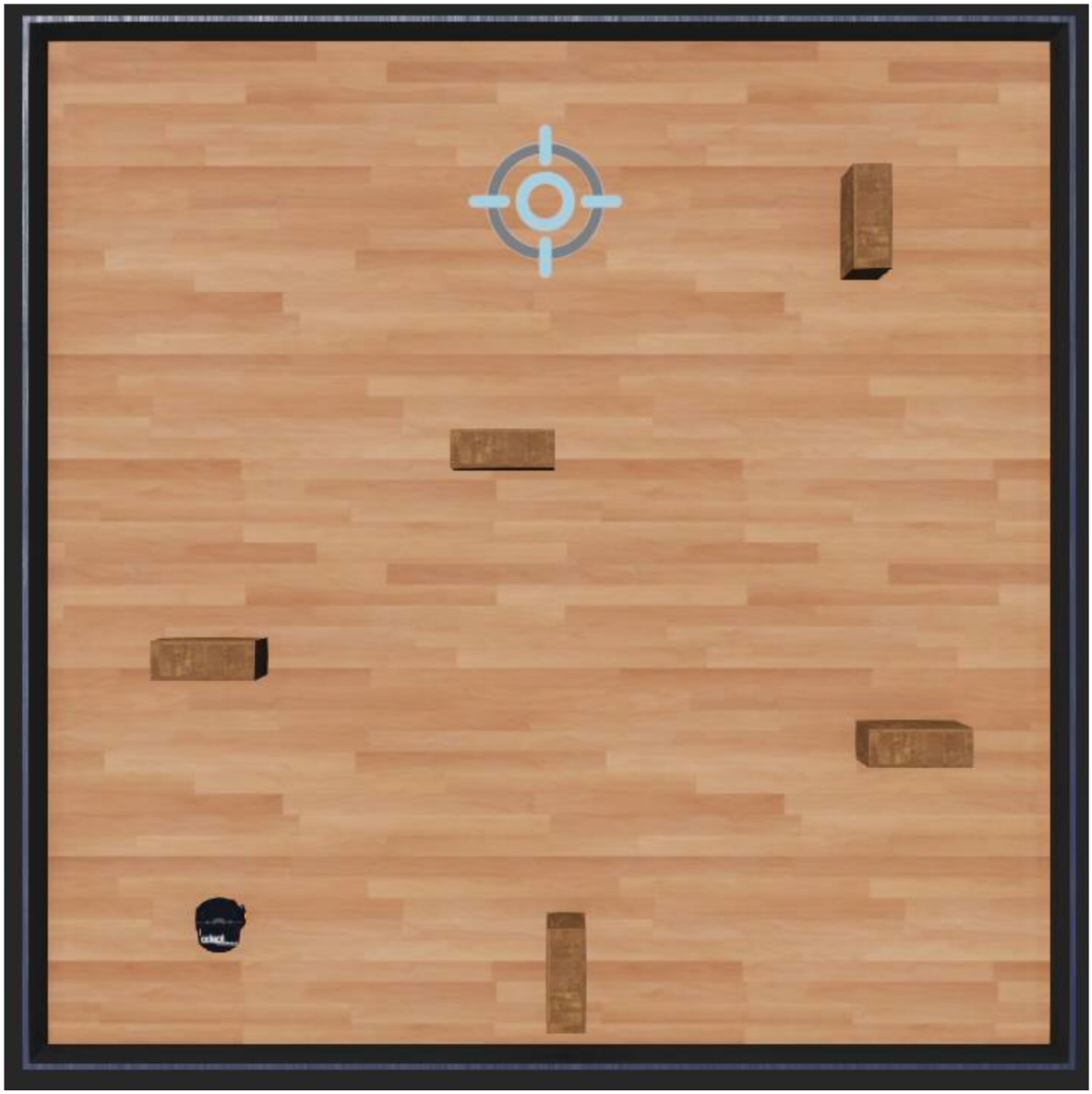
In this assignment, we aim to autonomously navigate a mobile robot with a differential drive (DX - Pioneer 3) in a static environment, as shown below, using the Webots simulator. The agent should be designed as a self-learning entity using a certain algorithm to plan its path efficiently, considering the starting and destination points, to reach the goal without colliding with obstacles.

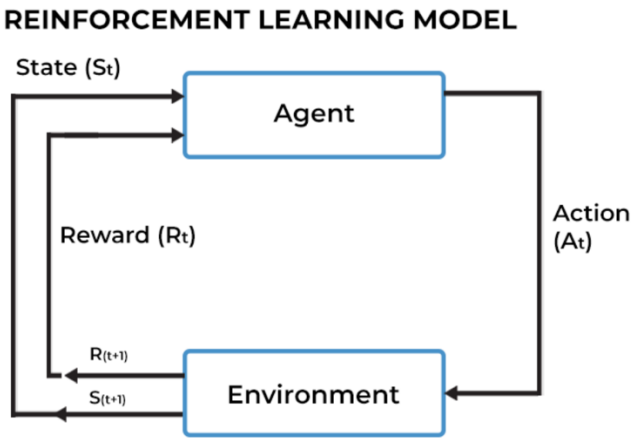


```python
# Add Webots controlling libraries
from controller import Robot
from controller import Supervisor

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

# 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

# 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")
```

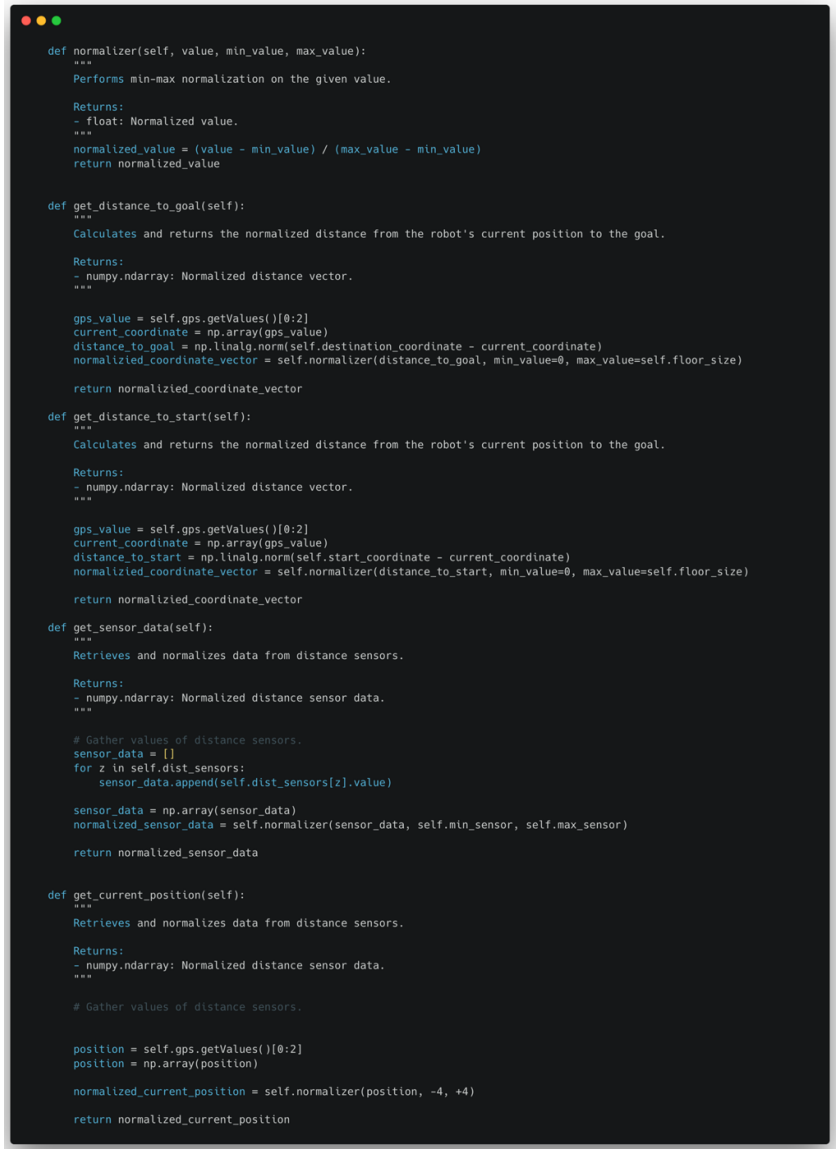
First, we need to import the necessary libraries into the robot's controller file to utilize their functions in solving the problem. We use the Supervisor library to access additional information about the agent's environment, making the training process easier. Next, we create an instance of the robot. To eliminate randomness in the variables mentioned in the code on each execution, we define a specific seed to generate random states. This allows us to observe the changes in the agent's learning solely based on our own hyperparameter modifications. This random state should follow a specific distribution on each run. We define a variable related to the device such that if CUDA is available, it will be used for processing the agent's computations. Otherwise, the CPU will be used.

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

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

        # General environment parameters
        self.max_speed = 1.5 # Maximum Angular speed in rad/s
        self.start_coordinate = np.array([-2.60, -2.96])
        self.destination_coordinate = np.array([-0.03, 2.72]) # Target (Goal) position
        self.reach_threshold = 0.06 # Distance threshold for considering the destination reached.
        obstacle_threshold = 0.1 # Threshold for considering proximity to obstacles.
        self.obstacle_threshold = 1 - obstacle_threshold
        self.floor_size = np.linalg.norm([8, 8])


        # Activate Devices
        #~~ 1) Wheel Sensors
        self.left_motor = robot.getDevice('left wheel')
        self.right_motor = robot.getDevice('right wheel')

        # Set the motors to rotate for ever
        self.left_motor.setPosition(float('inf'))
        self.right_motor.setPosition(float('inf'))

        # Zero out starting velocity
        self.left_motor.setVelocity(0.0)
        self.right_motor.setVelocity(0.0)

        #~~ 2) GPS Sensor
        sampling_period = 1 # in ms
        self.gps = robot.getDevice("gps")
        self.gps.enable(sampling_period)

        #~~ 3) Enable Touch Sensor
        self.touch = robot.getDevice("touch sensor")
        self.touch.enable(sampling_period)

        # List of all available sensors
        available_devices = list(robot.devices.keys())
        # Filter sensors name that contain 'so'
        filtered_list = [item for item in available_devices if 'so' in item and any(char.isdigit() for char in item)]
        filtered_list = sorted(filtered_list, key=lambda x: int(''.join(filter(str.isdigit, x))))

        #Space
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Box(low=0, high=1, shape=(5,), dtype=np.float32)
        self.max_steps = 500

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


        # Create dictionary from all available distance sensors and keep min and max of from total values
        self.max_sensor = 0
        self.min_sensor = 0
        self.dist_sensors = {}
        for i in filtered_list:
            self.dist_sensors[i] = robot.getDevice(i)
            self.dist_sensors[i].enable(sampling_period)
            self.max_sensor = max(self.dist_sensors[i].max_value, self.max_sensor)
            self.min_sensor = min(self.dist_sensors[i].min_value, self.min_sensor)
```

In the figure below, we can observe different components of a model or an agent that utilizes reinforcement learning:



In this section of the code, we aim to define and implement the agent's environment. Actions are input to the environment, and the reward and next state are output from it. The class related to the agent's environment or "Environment" must inherit from "Supervisor" because we need access to simulation environment information. Initially, we need to initialize variables such as maximum speed, destination threshold, obstacle distance threshold, environment size, etc., for problem-solving. Then, we specify the left and right motors, considering their positions as unlimited and their speeds as zero initially, so we can assign values to them when needed. Next, we identify and initialize the GPS, Touch, and Distance sensors. Using the Gymnasium library, we determine the number of discrete actions. There are three actions: moving straight, turning right, and turning left. Using the Box function, we determine the environment and the dimensions of the state vector. Based on the composition of the agent's state space, the agent's observation space will have dimensions of (5,). We also specify that the value of each element in the robot's observation space should be between 0 and 1. Therefore, we need to normalize the robot's observation space. Furthermore, we set the maximum number of steps the robot can take. In general, this piece of code sets the action space, observation space, and maximum number of stages for a reinforcement learning environment. The action space is discrete with three possible actions, and the observation space is continuous with five dimensions.



```
    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_to_goal(self):
        """
        Calculates and returns the normalized distance from the robot's current position to the goal.

        Returns:
        - numpy.ndarray: Normalized distance vector.
        """

        gps_value = self.gps.getValues()[0:2]
        current_coordinate = np.array(gps_value)
        distance_to_goal = np.linalg.norm(self.destination_coordinate - current_coordinate)
        normalizied_coordinate_vector = self.normalizer(distance_to_goal, min_value=0, max_value=self.floor_size)

        return normalizied_coordinate_vector

    def get_distance_to_start(self):
        """
        Calculates and returns the normalized distance from the robot's current position to the goal.

        Returns:
        - numpy.ndarray: Normalized distance vector.
        """

        gps_value = self.gps.getValues()[0:2]
        current_coordinate = np.array(gps_value)
        distance_to_start = np.linalg.norm(self.start_coordinate - current_coordinate)
        normalizied_coordinate_vector = self.normalizer(distance_to_start, min_value=0, max_value=self.floor_size)

        return normalizied_coordinate_vector

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

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

        # Gather values of distance sensors.
        sensor_data = []
        for z in self.dist_sensors:
            sensor_data.append(self.dist_sensors[z].value)

        sensor_data = np.array(sensor_data)
        normalized_sensor_data = self.normalizer(sensor_data, self.min_sensor, self.max_sensor)

        return normalized_sensor_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.


        position = self.gps.getValues()[0:2]
        position = np.array(position)

        normalized_current_position = self.normalizer(position, -4, +4)

        return normalized_current_position
```

The `normalizer()` function takes an input value along with its minimum and maximum possible values. It then normalizes the input value using the Min-Max normalization method to scale the values between 0 and 1. The normalized value is returned as the output.

The `get_distance_to_goal()` function calculates the current distance of the robot to the goal or destination based on the values obtained from the GPS sensor. It uses the Euclidean distance and returns the normalized value of the distance as the output.

The `get_distance_to_start()` function, similar to the previous function, calculates the current distance of the robot to the starting point or origin. It normalizes the distance and returns the normalized value as the output.

The `get_sensor_data()` function takes the distance information obtained from the distance sensors, normalizes the values, and returns them as the output.

The `get_current_position()` function uses the GPS sensor information to obtain the current position of the robot. It normalizes the position values and returns them as the output.

Please note that the specific implementation details of these functions are not provided. The descriptions above provide a general understanding of the purpose and behavior of each function.

```python
def get_observations(self):

    normalized_sensor_data = np.array(self.get_sensor_data(), dtype=np.float32)
    normalized_current_distance = np.array([self.get_distance_to_goal()], dtype=np.float32)
    normalized_current_position = np.array(self.get_current_position(), dtype=np.float32)

    state_vector = np.concatenate([normalized_current_distance, normalized_sensor_data, normalized_current_position], axis=0)

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


def step(self, action):

    self.apply_action(action)
    step_reward, done = self.get_reward()
    state = self.get_observations()
    # Time-based termination condition
    if (int(self.getTime()) + 1) % self.max_steps == 0:
        done = True
    none = 0
    return state, step_reward, done, none, {}


def get_reward(self):

    done = False
    reward = 0

    normalized_sensor_data = self.get_sensor_data()
    normalized_current_distance = self.get_distance_to_goal()
    normalized_start_distance = self.get_distance_to_start()

    normalized_current_distance *= 100 # The value is between 0 and 1. Multiply by 100 will make the function work better
    reach_threshold = self.reach_threshold * 100


     # (1) Reward according to distance
    if normalized_current_distance < 42:
        if normalized_current_distance < 10:
            growth_factor = 5
            A = 2.5
        elif normalized_current_distance < 25:
            growth_factor = 4
            A = 1.5
        elif normalized_current_distance < 37:
            growth_factor = 2.5
            A = 1.2
        else:
            growth_factor = 1.2
            A = 0.9
        reward += A * (1 - np.exp(-growth_factor * (1 / normalized_current_distance)))

    else:
        reward += -normalized_current_distance / 100


    # (2) Reward or punishment based on failure or completion of task
    check_collision = self.touch.value
    if normalized_current_distance < reach_threshold:
        # Reward for finishing the task
        done = True
        reward += 50
        print('+++ SOLVED +++')
    elif check_collision:
        # Punish if collision
        done = True
        reward -= 5


    # (3) Punish if close to obstacles
    elif np.any(normalized_sensor_data[normalized_sensor_data > self.obstacle_threshold]):
        reward -= 0.0001

    return reward, done
```

The function `get_observations()` is responsible for constructing the robot's observation space and returning the State Vector as its output. The robot's observation space includes information from distance sensors, including two dimensions, information about the distance from the robot to the destination (one dimension), and the current location of the robot (two dimensions). These

pieces of information are concatenated in the current moment, forming an observation vector representing the robot's state at that moment. The more comprehensive and rich this observable space is, the better it is for the robot.

By using the `reset()` function and the help of a supervisor, we can reset the environment and restore everything to the initial conditions, preparing the environment for the robot to start again in its initial state. Additionally, we should consider the basic time of the environment. Then, we can use the `get_observation()` function to obtain 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 punishment for that action and then calculate the robot's observations after performing that action to observe the changes in the environment. Next, we check if we have reached the maximum allowed number of steps for the robot. If we have reached it, the executed command will be issued.

In the `reward()` function, we aim to adjust the rewards and punishments related to the robot's actions according to our problem-solving objectives. The robot, based on the specified values in this function, will be either punished or rewarded for the action it has taken to solve the problem. Therefore, the values specified in this function should be adjusted as best as possible considering the solution and the nature of the problem.

First case: Distance to the destination
To differentiate between distances from the robot to the destination, we normalize its value by multiplying it by 100. For distances smaller than 42, we consider different values of A and growth_factor to determine the reward relative to the proximity of the robot to the destination. It is evident that these values will be higher for closer distances. Otherwise, if the robot is more than 42 units away from the destination, we penalize it. The rewards considered for the robot relative to different A and growth_factor values are calculated using the equation seen in the provided code snippet. With the help of this case, the robot learns that the closer it gets to the goal, the more reward it receives, and if it is far from the destination, it gets punished.

Second case: Arrival at the destination
If the robot reaches a specified threshold for reaching the destination, it has reached the goal and receives a reward of 50 units. The robot learns to recognize and reach the coordinates of the goal based on its current location in its observation space.

Third case: Collision with obstacles
If the robot collides with obstacles, it receives a penalty of 5 units. The robot learns to avoid obstacles.

Fourth case: Proximity to obstacles
If the robot gets too close to obstacles beyond a certain threshold, it is penalized.

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

        Returns:
        - None
        """
        self.left_motor.setPosition(float('inf'))
        self.right_motor.setPosition(float('inf'))

        if action == 0: # move forward
            # print("forward")
            self.left_motor.setVelocity(self.max_speed)
            self.right_motor.setVelocity(self.max_speed)
        elif action == 1: # turn right
            # print("right")
            self.left_motor.setVelocity(self.max_speed)
            self.right_motor.setVelocity(-self.max_speed)
        elif action == 2: # turn left
            # print("left")
            self.left_motor.setVelocity(-self.max_speed)
            self.right_motor.setVelocity(self.max_speed)

        robot.step(500)


        self.left_motor.setPosition(0)
        self.right_motor.setPosition(0)
        self.left_motor.setVelocity(0)
        self.right_motor.setVelocity(0)


class Agent_FUNCTION():
    def __init__(self, save_path, num_episodes):
        self.save_path = save_path
        self.num_episodes = num_episodes

        self.env = Environment()

        #PPO
        self.policy_network = PPO("MlpPolicy", self.env, verbose=1,
tensorboard_log="./results_tensorboard/")#.learn(total_timesteps=1000)


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

    def load(self):
        print(self.save_path+"PPO-Best")
        self.policy_network = PPO.load(self.save_path+"PPO-Best")
```

Using the `apply_action()` function, we intend to apply the desired action to the robot. These actions are discrete and consist of three cases: moving forward, rotating to the right, and rotating to the left.

To make the robot move forward, both wheels move at maximum speed in the same direction. To make the robot move to the right, the left motor moves forward at maximum speed, and the right motor moves backward at maximum speed. To move to the left, the motion of the motors is the opposite of the previous case.

Class related to the agent function, or Agent_Function:

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

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

In PPO, the agent tries to learn an optimal policy that maximizes the total cumulative reward for each action taken in any state. To achieve this, the algorithm uses a specific objective function called the "clipped surrogate objective" that allows policy updates to be performed in a way that restricts significant changes in the policy. This enables the agent to improve its performance in the environment while preventing excessive fluctuations during training.

In the first stage, the agent evaluates a new objective function by considering the difference in action probabilities between the new and old policies. In the second stage, using this objective function, the policy parameters are updated to enable the agent to achieve better performance in the given environment. PPO is a suitable algorithm for continuous action spaces and can handle continuous actions effectively.

In comparison to the Deep Q-Network (DQN) algorithm, which is a value-based algorithm, 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.

For example, in a project like this, if a robot needs to learn how to navigate in a dynamic environment to reach a specific goal, the PPO algorithm can be helpful. This algorithm uses the robot's past experiences in the environment to determine better policies for the robot's movement. These improvements can include changes in speed, direction, or how the robot responds to obstacles and different environmental conditions.
In summary, PPO utilizes the robot's past experiences in the environment to improve operational policies, enabling the robot to achieve optimal performance in dynamic environments and reach its goal.

The Proximal Policy Optimization (PPO) algorithm has significant improvements compared to TRPO and DQN algorithms. These improvements can include:

1. Efficiency: PPO has noticeable improvements in efficiency compared to TRPO. These improvements often lead to faster learning and more optimal utilization of computational resources.

2. Stability: PPO has greater stability in learning compared to TRPO and DQN. These improvements result in a reduction in the occurrence of sudden shocks during the learning process.

3. Experience Memory Usage: PPO can utilize experience memory to manage past experiences, which helps improve the efficiency and stability of the algorithm.

4. Compatibility with Continuous Action Problems: PPO is a suitable algorithm for problems with continuous action spaces and can effectively handle continuous actions. This demonstrates its flexibility and effectiveness in real-world problems.

Stable Baselines3 is a popular library for implementing reinforcement learning algorithms in Python. It includes various implementations of reinforcement learning algorithms such as PPO (Proximal Policy Optimization), A2C (Advantage Actor-Critic), DQN (Deep Q-Network), and more. Stable Baselines3 provides useful tools for training, evaluating, and using reinforcement learning models, and it can help you in the development and testing of reinforcement learning algorithms. In this project, we have attempted to use this library to benefit from the PPO algorithm.

In this class, we consider the policy learning network using the Stable Baselines3 library and the PPO algorithm. At the same time, we specify the path for storing the generated charts during the agent's training. This allows us to access the generated charts for analysis and examination of the training process after completing the training.

We use the save() and load() functions to save the trained agent's policy function after training and during agent testing.

```python
    def train(self) :
        start_time = time.time()
        reward_history = []
        best_score = -np.inf

        self.policy_network.learn(total_timesteps=(num_episodes*100))

        self.save()


    def test(self):

        state = self.env.reset()
        for episode in range(1, self.num_episodes+1):
            rewards = []
            # state = self.env.reset()
            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)
                # print("state: ", state)
                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)
                # print("reward: ", reward)
                ep_reward += reward

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

We start the training of the policy network, which is the PPO algorithm, using the train() function. We also specify the total number of time steps required for training. With this function, the agent gains experience in the environment and learns through reinforcement learning using PPO to update the best policy function for the robot. By using this updated policy function, the agent can achieve the best rewards during both training and testing phases.

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

```python
if __name__ == '__main__':
    # Configs
    save_path = './results/'
    train_mode = False
    num_episodes = 1000 if train_mode else 10


    env = Environment()

    agent = Agent_FUNCTION(save_path, num_episodes)


    if train_mode:
        # Initialize Training
        agent.train()
    else:
        # Load PPO
        agent.load()
        # Test
        agent.test()
```

Finally, we specify the path for saving the agent's policy function along with the number of episodes required for training and testing. Then, to perform training and testing of the agent in the environment, we can set the variable related to whether training or testing should be done.

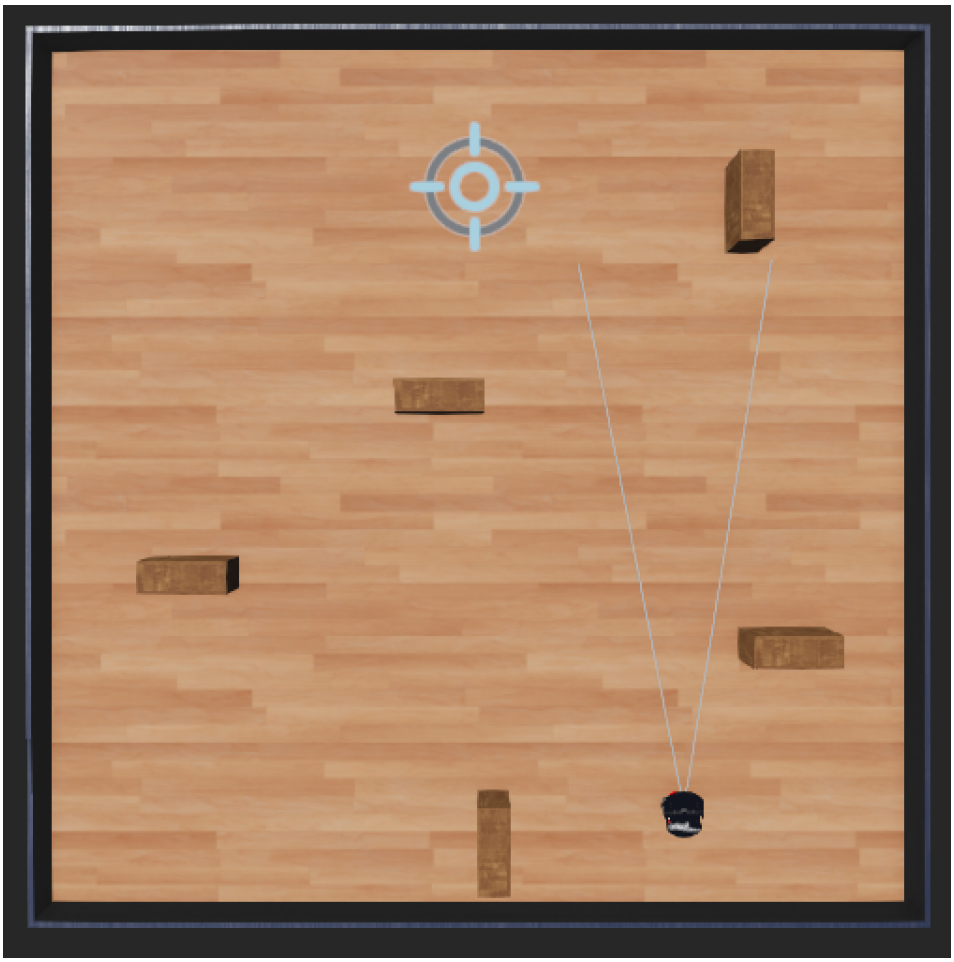
Result of agent testing:

```
./results/PPO-Best
+++ SOlVED +++
Episode 1: Score = 94.862
+++ SOlVED +++
Episode 2: Score = 86.788
+++ SOlVED +++
Episode 3: Score = 63.198
+++ SOlVED +++
Episode 4: Score = 88.877
+++ SOlVED +++
Episode 5: Score = 91.566
+++ SOlVED +++
Episode 6: Score = 87.651
+++ SOlVED +++
Episode 7: Score = 63.508
+++ SOlVED +++
Episode 8: Score = 113.697
+++ SOlVED +++
Episode 9: Score = 63.084
+++ SOlVED +++
Episode 10: Score = 99.268
INFO: 'my_controller' controller exited successfully.
```

As you can see in the above image, the agent achieved 10 successful reachings of the goal after learning in the provided environment as the problem statement.

2- Change the starting point to a different point in the environment of your choice and perform the test again (without retraining). In this case, the navigation may not necessarily be successful.

New Agent Position:



To investigate whether the robot can still reach the goal without retraining when the starting point is changed, we modify the robot's initial position to different coordinates.

Result of agent testing in the new position:

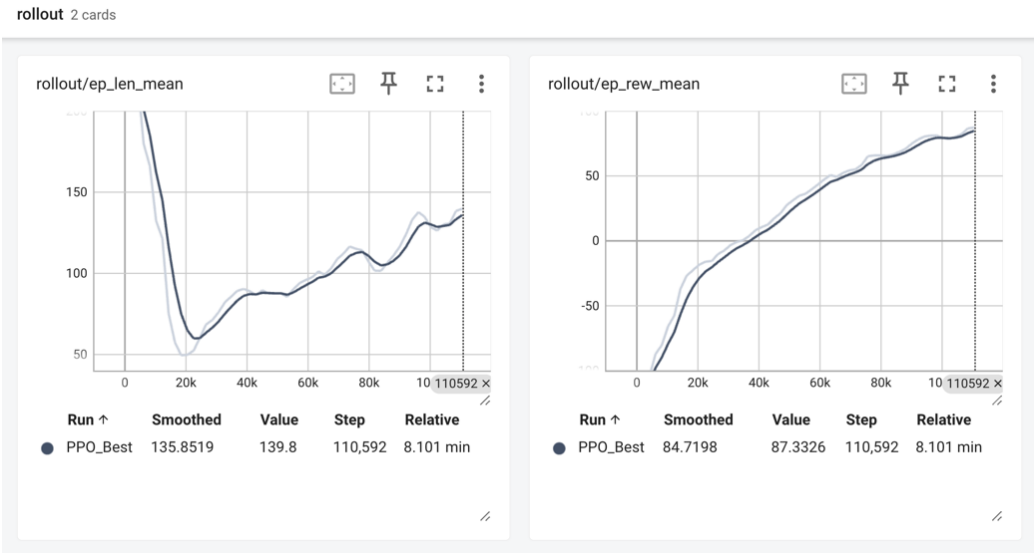
```
./results/PPO-Best
+++ SOlVED +++
Episode 1: Score = 114.387
+++ SOlVED +++
Episode 2: Score = 99.963
+++ SOlVED +++
Episode 3: Score = 77.537
+++ SOlVED +++
Episode 4: Score = 100.684
+++ SOlVED +++
Episode 5: Score = 96.940
+++ SOlVED +++
Episode 6: Score = 104.900
+++ SOlVED +++
Episode 7: Score = 76.467
+++ SOlVED +++
Episode 8: Score = 93.270
+++ SOlVED +++
Episode 9: Score = 113.247
+++ SOlVED +++
Episode 10: Score = 99.236
INFO: 'my_controller' controller exited successfully.
```

As you can see in the above image, the agent achieved 10 successful reachings of the goal even after the initial training without any retraining in the new starting position.

Reason for Success:

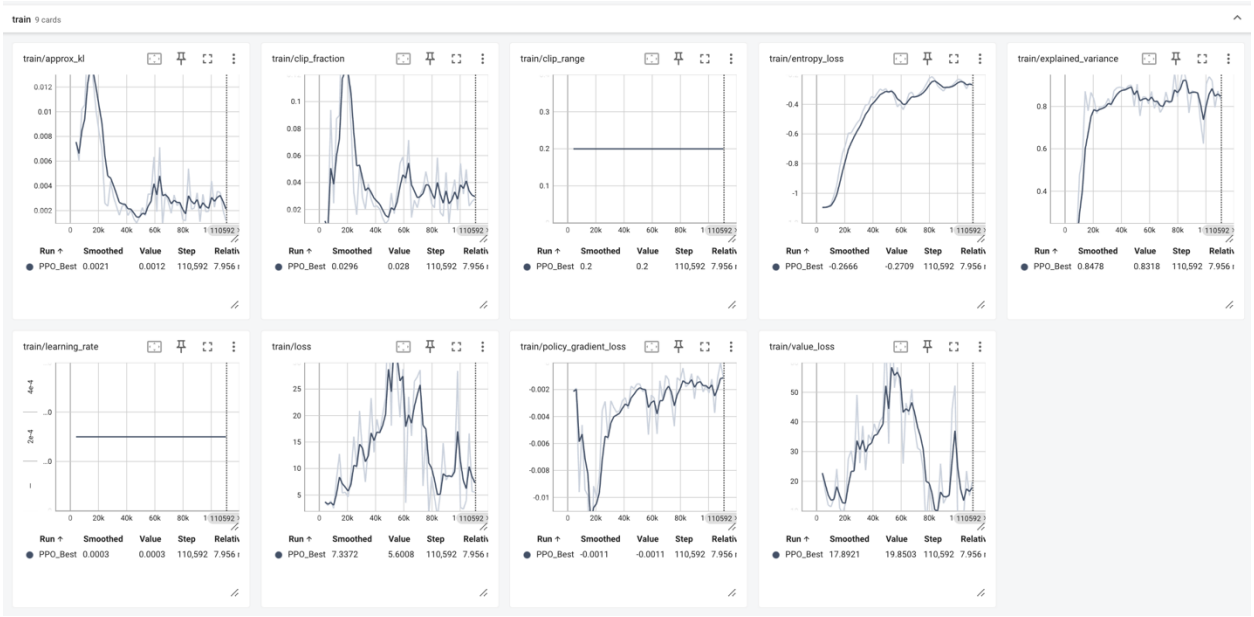
As mentioned in the observations section, the robot's observation space includes information from distance sensors, distance to the goal, and the robot's current position. During the training process, through exploration and receiving different rewards and penalties, the agent has learned how rewards are related to observations and how it should behave to get closer to the goal and ultimately reach it.
The information about the distance to the goal in the observation space and the rewards obtained during training have enabled the agent to learn the relationship between rewards along the trajectory and proximity to the goal.
The information about the distance to obstacles and penalties when approaching them during training has taught the robot to avoid obstacles during the testing phase.
The information about the current position in the observation space and the rewards obtained when reaching the goal during training allow the agent to learn the relationship between the current position and the goal position and understand the path it needs to follow to reach the goal.
Due to these learned relationships, when the starting position is changed, the robot can utilize the learned observations-rewards connections during each moment and evaluate them with the rewards and penalties during the testing phase, enabling it to reach the goal coordinates.

Analysis of the agent's training charts for questions 1 and 2:



The chart related to "ep_len_mean" represents the average length of each episode during the learning process, which has a direct correlation with avoiding collisions with obstacles. Throughout the training process, the agent has taken more steps per episode.

The chart related to "ep_rew_mean" shows the average rewards received during the training process. As observed, the agent starts off inexperienced and performs poorly, receiving penalties until it learns from its experiences to gain significant rewards over its lifespan. As seen in the chart, at the end of training, the agent has achieved high rewards while reaching its goal.



The "approx_kl" chart indicates the approximate average KL divergence between the old and new policy functions (for PPO). In other words, it estimates the level of changes occurring during

updates. As seen in the corresponding chart, this value decreases over time during the training process. This means that towards the end of training, the robot has reached its goal, and there is no need for further policy updates.

The "clip_fraction" chart represents the control level of the clip feature on policy updates, which decreases over time. This indicator shows that the agent hasn't experienced sudden changes during policy updates.

The value of "clip_range" remains constant throughout the training process.

The "entropy_loss" chart displays the difference between the learned policy function and a completely random policy function at each step during the training process. As observed in the chart, this value is increasing, indicating that the policy function has moved away from randomness and is becoming more purposeful in its training.

The "explained_variance" chart represents a fraction of the explained variance by the value function, as explained by the given function. The ideal value for this chart is one. As seen, we have approached this value during the training process.

The learning_rate value remains constant throughout the training process.

The "train/loss" chart represents the current total loss value at each step. The "value/loss" chart usually represents the error between the output of the value function and the Monte Carlo estimation. As observed, both charts initially increase due to the robot's exploration, but after gaining experience and learning during training, the values of the two errors decrease over the training process.

The "policy_gradient_loss" chart represents the current value of the policy gradient loss (its value doesn't have significant meaning). As seen, its value decreases over time during training.

After performing the training process and examining the generated charts to explain the training process, we can conclude that the training process has been conducted correctly. The agent has learned to avoid obstacles and walls to reach its goal by gaining experience and receiving rewards and penalties during the training process.

Extra Point: Perform obstacle avoidance path planning using only the provided GPS and Camera sensors, similar to the instructions given. It should be noted that the Pioneer 3-DX robot does not have a camera sensor, so you will need to manually add it. The camera's properties can be adjusted after adding it to the robot in Webots.

```python
# Add Webots controlling libraries
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 DQN
from stable_baselines3 import PPO
from stable_baselines3.common import results_plotter
from stable_baselines3.common.monitor import Monitor
from stable_baselines3.common.results_plotter import load_results, ts2xy
from stable_baselines3.common.callbacks import BaseCallback


# 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")
```

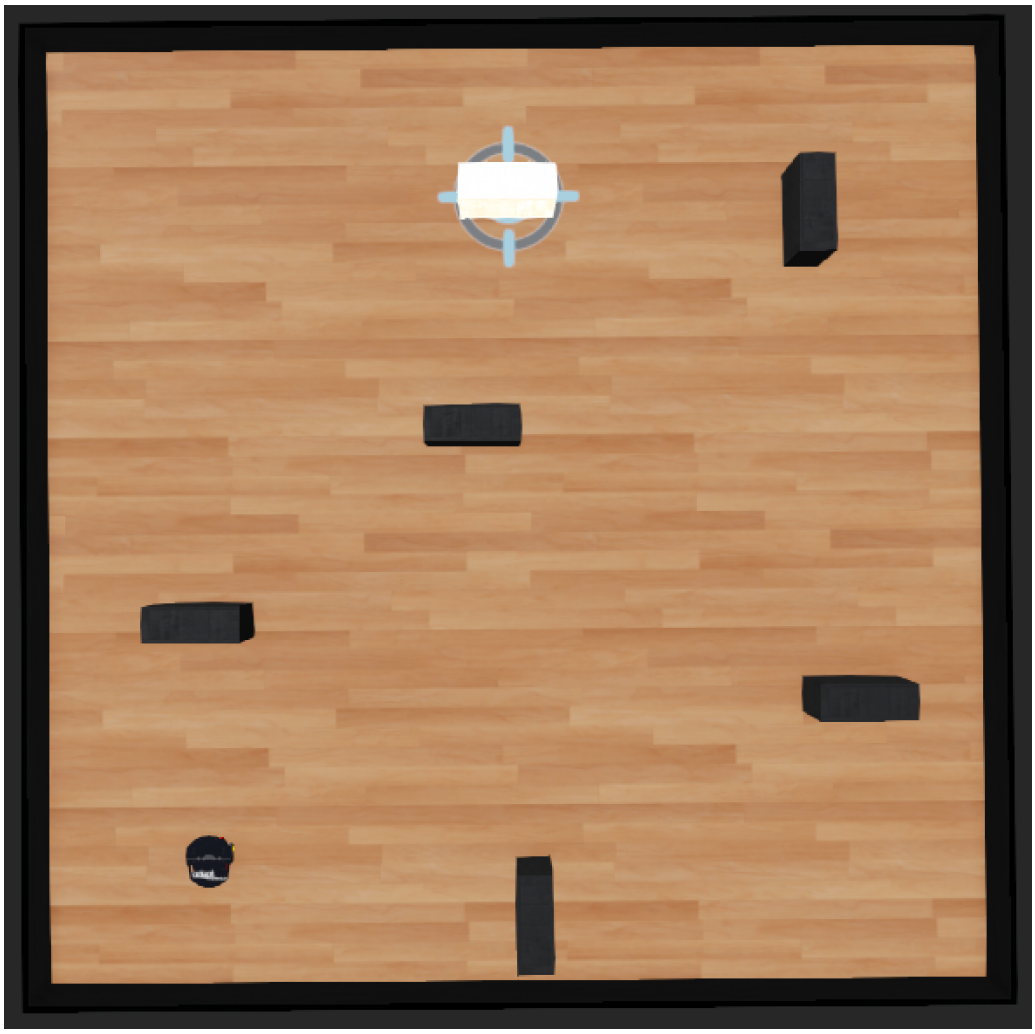
The code related to this section has been explained in the previous section.

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

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

        # General environment parameters
        self.max_speed = 1.5 # Maximum Angular speed in rad/s
        self.start_coordinate = np.array([-2.60, -2.96])
        self.destination_coordinate = np.array([-0.03, 2.72]) # Target (Goal) position
        self.reach_threshold = 0.08 # Distance threshold for considering the destination reached.
        obstacle_threshold = 0.1 # Threshold for considering proximity to obstacles.
        self.obstacle_threshold = 1 - obstacle_threshold
        self.floor_size = np.linalg.norm([8, 8])


        # Activate Devices
        #~~ 1) Wheel Sensors
        self.left_motor = robot.getDevice('left wheel')
        self.right_motor = robot.getDevice('right wheel')

        # Set the motors to rotate for ever
        self.left_motor.setPosition(float('inf'))
        self.right_motor.setPosition(float('inf'))

        # Zero out starting velocity
        self.left_motor.setVelocity(0.0)
        self.right_motor.setVelocity(0.0)

        #~~ 2) GPS Sensor
        sampling_period = 1 # in ms
        self.gps = robot.getDevice("gps")
        self.gps.enable(sampling_period)

        #~~ 3) Enable Camera
        sampling_period = 1 # in ms
        self.camera = robot.getDevice("camera")
        self.camera.enable(sampling_period)

        #~~ 4) Enable Touch Sensor
        self.touch = robot.getDevice("touch sensor")
        self.touch.enable(sampling_period)

        self.action_space = spaces.Discrete(3)

        self.observation_space = spaces.Box(low=0, high=1, shape=(2,), dtype=np.float32)
        # Reset
        self.simulationReset()
        self.simulationResetPhysics()
        super(Supervisor, self).step(int(self.getBasicTimeStep()))
        robot.step(200) # take some dummy steps in environment for initialization

        self.max_steps = 500
```

The only difference is that the robot's observation space has been reduced to 2D because we no longer use distance sensors. Instead, we will use the information collected by the camera sensor.

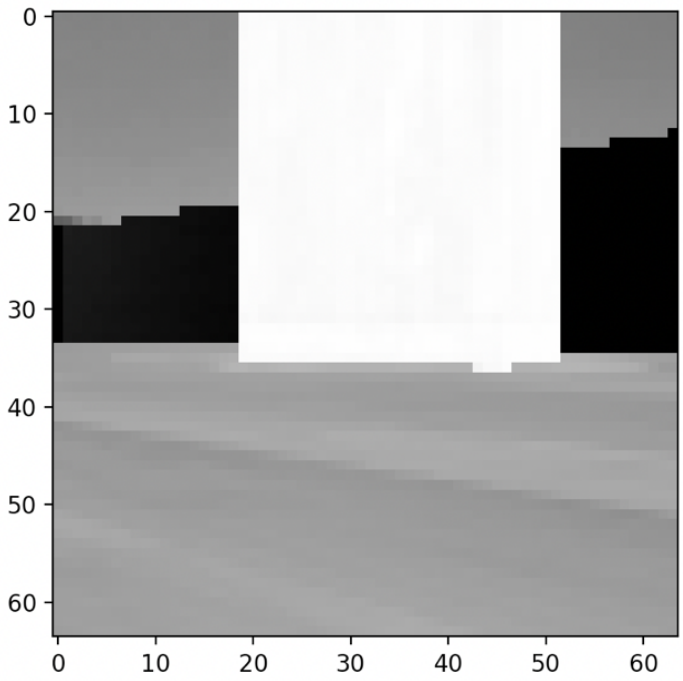
```python
def normalizer(self, value, min_value, max_value):

    normalized_value = (value - min_value) / (max_value - min_value)
    return normalized_value


def get_distance_to_goal(self):

    gps_value = self.gps.getValues()[0:2]
    current_coordinate = np.array(gps_value)
    distance_to_goal = np.linalg.norm(self.destination_coordinate - current_coordinate)
    normalizied_coordinate_vector = self.normalizer(distance_to_goal, min_value=0, max_value=self.floor_size)

    return normalizied_coordinate_vector



def get_camera_information(self):
    image = self.camera.getImageArray()
    image = np.array(image)

    # Convert image to a supported depth format (e.g., CV_8U)
    if image.dtype != np.uint8:
        image = image.astype(np.uint8)

    gray_image = cv.cvtColor(image, cv.COLOR_RGB2GRAY)
    # plt.imshow(gray_image, cmap='gray")

    plt.show()
    # Calculate histogram
    histogram = cv.calcHist([gray_image], [0], None, [256], [0, 256])


    # Calculate sum of frequencies before intensity 50 (obstacles)
    intensity_50 = 50
    frequencies_before_50 = histogram[:intensity_50].sum()

    # Calculate sum of additive values of histogram between intensities 240 to 255
    intensities_240_to_255 = histogram[240:256].sum()

    return frequencies_before_50, intensities_240_to_255


def get_distance_to_start(self):

    gps_value = self.gps.getValues()[0:2]
    current_coordinate = np.array(gps_value)
    distance_to_start = np.linalg.norm(self.start_coordinate - current_coordinate)
    normalizied_coordinate_vector = self.normalizer(distance_to_start, min_value=0, max_value=self.floor_size)

    return normalizied_coordinate_vector


def get_current_position(self):


    position = self.gps.getValues()[0:2]
    position = np.array(position)

    normalized_current_position = self.normalizer(position, -4, +4)

    return normalized_current_position
```

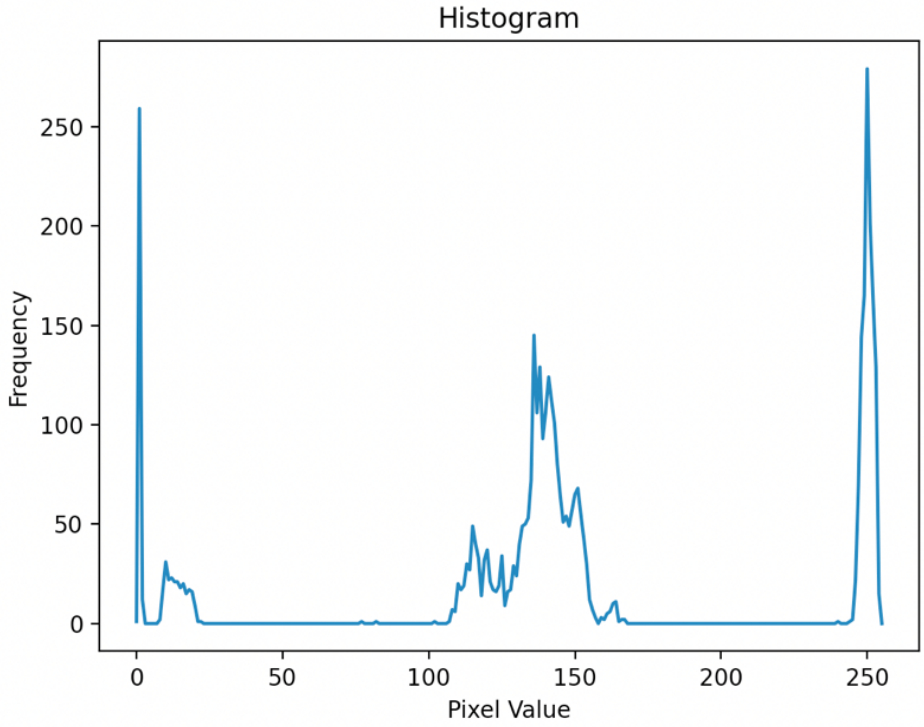
In the function `get_camera_information()`, we first receive the current captured image from the camera and convert it to the acceptable format of OpenCV. Then, for targeted processing, we convert the image to grayscale. To simplify the use of the camera for problem-solving, we modify the environment as follows:

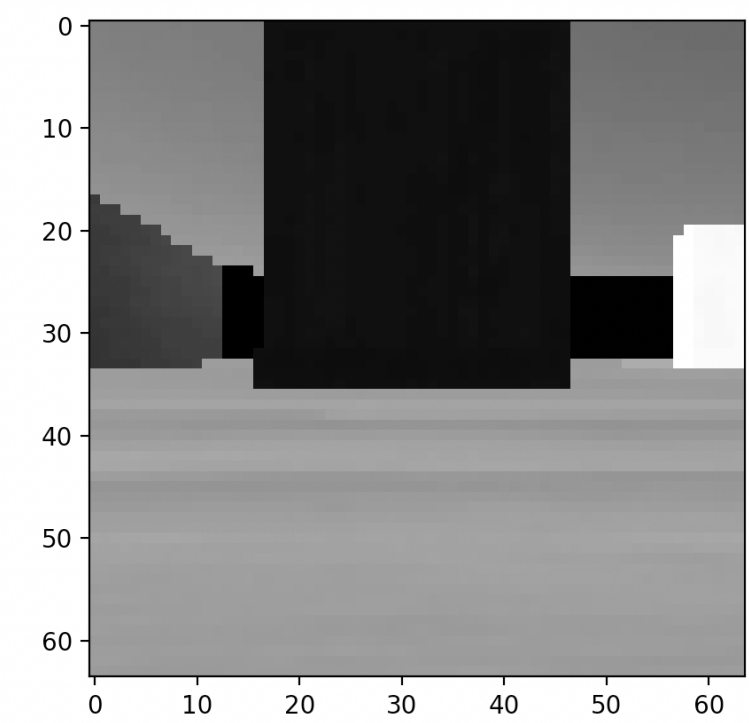


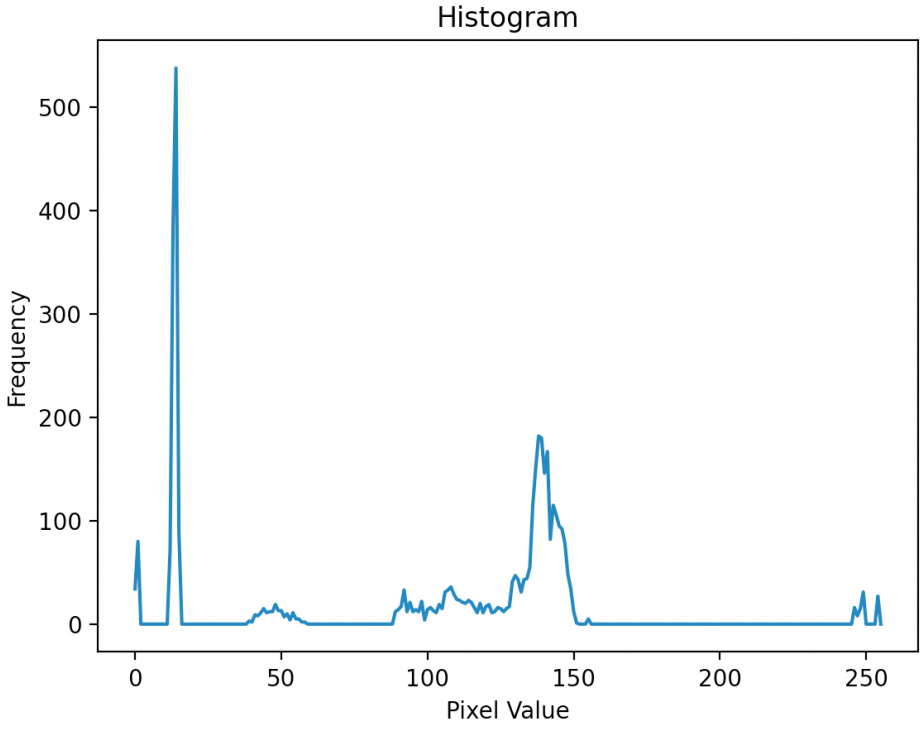
We have converted obstacles to black and the goal to white in the image. After receiving the camera image and converting it to grayscale, we analyze and extract features of the obstacles and goals. Then, we plot the histograms of these images.

The images of the goal and obstacles from the robot's perspective will be as follows:


0
10
20
30
40
50
60
0
10
20
30
40
50
60


Histogram
Frequency
250
200
150
100
50
0
0
50
100
150
200
250
Pixel Value


0
10
20
30
40
50
60
0
10
20
30
40
50
60


Histogram
Frequency
500
400
300
200
100
0
0
50
100
150
200
250
Pixel Value

By analyzing the above histograms, we observe that from the robot's perspective, the goal is visible in the intensity range of 240 to 255, while obstacles are visible in the intensity range of 0 to 50. We return the number of pixels within these ranges as the output of the function. The more pixels registered in these ranges, the closer we are to obstacles or goals, and vice versa.

```
    def get_observations(self):

        normalized_current_position = np.array(self.get_current_position(), dtype=np.float32)

        state_vector = np.concatenate([normalized_current_position], axis=0)

        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()
        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
        if (int(self.getTime()) + 1) % self.max_steps == 0:
            done = True
        none = 0
        return state, step_reward, done, none, {}
```

The code related to this section has been explained in the previous section. The only difference is that in the robot's observation space this time, only the robot's positional information is recorded because we no longer have information from distance sensors.

```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_current_distance = self.get_distance_to_goal()

        normalized_current_distance *= 100 # The value is between 0 and 1. Multiply by 100 will make the function work better
        reach_threshold = self.reach_threshold * 100

        distance_to_obstacles, distance_to_goal = self.get_camera_information()


        # [1] Reward according to distance
        if normalized_current_distance < 42:
            if normalized_current_distance < 2:
                growth_factor = 9
                A = 7
            elif normalized_current_distance < 5:
                growth_factor = 8
                A = 6
            elif normalized_current_distance < 10:
                growth_factor = 5
                A = 3.5
            elif normalized_current_distance < 25:
                growth_factor = 5
                A = 3.5
            elif normalized_current_distance < 37:
                growth_factor = 4.5
                A = 3.2
            else:
                growth_factor = 3.2
                A = 2.9
            reward += A * (1 - np.exp(-growth_factor * (1 / normalized_current_distance)))

        else:
            reward += -(normalized_current_distance / 100) * 2


        # [2] Punish if close to obstacles
        if distance_to_obstacles > 1700:

                if distance_to_obstacles > 4000:
                    reward-= 6
                elif distance_to_obstacles > 3000:
                    reward-= 4
                elif distance_to_obstacles > 2500:
                    reward-= 3
                elif distance_to_obstacles > 2000:
                    reward-= 1
        elif distance_to_obstacles < 1000:
                if distance_to_obstacles < 800:
                    reward+= 2
                elif distance_to_obstacles < 1000:
                    reward+= 1


        # [3] Reward if close to Goal
        if distance_to_goal > 0:

            if distance_to_goal > 1500:
                reward+= 11
            elif distance_to_goal > 1000:
                reward+= 9
            elif distance_to_goal > 700:
                reward+= 7
            elif distance_to_goal > 600:
                reward+= 6
            elif distance_to_goal > 500:
                reward+= 5


        # [4] Reward or punishment based on failure or completion of task
        check_collision = self.touch.value

        if normalized_current_distance < reach_threshold:
            # Reward for finishing the task
            done = True
            reward += 200
            print('+++ SOLVED +++')
        elif check_collision:
            # Punish if Collision
            done = True
            reward -= 10

        return reward, done


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

        Returns:
        - None
        """
        self.left_motor.setPosition(float('inf'))
        self.right_motor.setPosition(float('inf'))

        if action == 0: # move forward
            # print("forward")
            self.left_motor.setVelocity(self.max_speed)
            self.right_motor.setVelocity(self.max_speed)
        elif action == 1: # turn right
            # print("right")
            self.left_motor.setVelocity(self.max_speed)
            self.right_motor.setVelocity(-self.max_speed)
        elif action == 2: # turn left
            # print("left")
            self.left_motor.setVelocity(-self.max_speed)
            self.right_motor.setVelocity(self.max_speed)

        robot.step(500)


        self.left_motor.setPosition(0)
        self.right_motor.setPosition(0)
        self.left_motor.setVelocity(0)
        self.right_motor.setVelocity(0)


class Agent_FUNCTION():
    def __init__(self, save_path, num_episodes):
        self.save_path = save_path
        self.num_episodes = num_episodes


        self.env = Environment()

        #PPO
        self.policy_network = PPO("MlpPolicy", self.env, verbose=1, tensorboard_log="./results_tensorboard/")


    def save(self):
        print(self.save_path ,"best_model")
        self.policy_network.save(self.save_path + "best_model")

    def load(self):
        print(self.save_path+"best_model")
        self.policy_network = PPO.load(self.save_path+"best_model")


    def compute_returns(self, rewards):
        t_steps = torch.arange(len(rewards))
        discount_factors = torch.pow(self.gamma, t_steps).to(device)
        rewards = torch.tensor(rewards, dtype=torch.float32, device=device)
        returns = rewards * discount_factors
        returns = returns.flip(dims=(0,)).cumsum(dim=0).flip(dims=(0,))
        if self.baseline:
            mean_reward = torch.mean(rewards)
            returns -= mean_reward
        return returns

    def compute_loss(self, log_probs, returns):
        loss = []
        for log_prob, G in zip(log_probs, returns):
            loss.append(-log_prob * G)
        return torch.stack(loss).sum()
```

The code related to this section has been explained in the previous section. The differences that occurred in this section are as follows:
- Changes in the reward value related to proximity to the destination.
- Extraction of useful information from the camera sensor. In the camera information function, it was explained that the number of pixels related to the range of observing the goal and obstacles is returned as the function's output. The more pixels registered in these ranges, the closer we are to obstacles or goals. Based on this, we can consider rewards in a way that if the number of pixels related to the observation of the goal increases, it receives higher rewards. Conversely, if the number of pixels related to the observation of obstacles increases, we consider penalties for the agent. Accordingly, we have removed the rewards and penalties related to the distance sensors and replaced them with rewards and penalties related to the camera sensor so that the robot learns to avoid obstacles and move towards the goal.

```
    def train(self) :
        start_time = time.time()
        reward_history = []
        best_score = -np.inf

        self.policy_network.learn(total_timesteps=(num_episodes*100))
        self.save()

        self.env.reset()


    def test(self):

        state = self.env.reset()
        for episode in range(1, self.num_episodes+1):
            rewards = []
            # state = self.env.reset()
            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)
                # print("state: ", state)
                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)
                # print("reward: ", reward)
                ep_reward += reward

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


if __name__ == '__main__':
    # Configs
    save_path = './results/'
    train_mode = False
    num_episodes = 4000 if train_mode else 20


    env = Environment()

    agent = Agent_FUNCTION(save_path, num_episodes)


    if train_mode:
        # Initialize Training
        agent.train()
    else:
        # Load PPO
        agent.load()
        # Test
        agent.test()
```

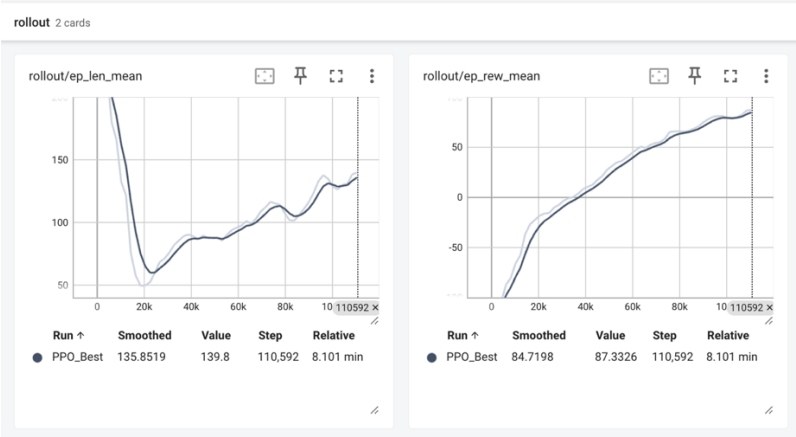
The code related to this section has been explained in the previous section, with the difference being that the number of episodes has been set to 4000, which corresponds to a total of 4,000,000 time steps for training.

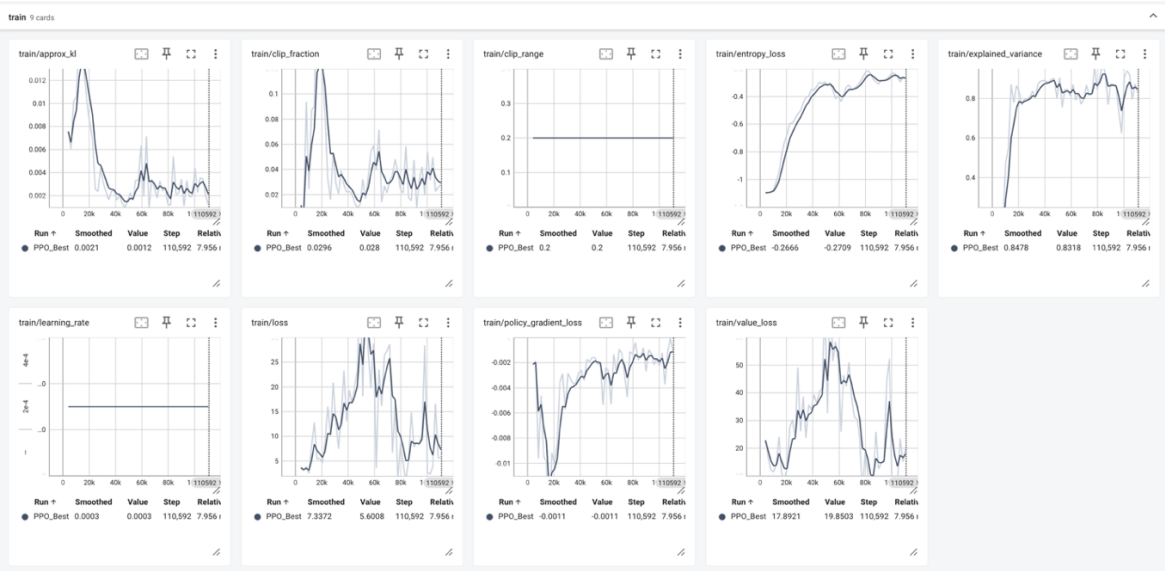
Result of the agent's test for the bonus question:

```
+++ SOlVED +++
Episode 1: Score = 865.431
Episode 2: Score = -3.790
+++ SOlVED +++
Episode 3: Score = 470.613
Episode 4: Score = 16.916
Episode 5: Score = -30.480
Episode 6: Score = -46.796
Episode 7: Score = -118.496
+++ SOlVED +++
Episode 8: Score = 631.411
Episode 9: Score = 66.848
Episode 10: Score = -82.350
+++ SOlVED +++
Episode 11: Score = 496.264
Episode 12: Score = 24.079
+++ SOlVED +++
Episode 13: Score = 605.160
Episode 14: Score = -41.812
Episode 15: Score = 29.859
+++ SOlVED +++
Episode 16: Score = 657.868
+++ SOlVED +++
Episode 17: Score = 694.132
+++ SOlVED +++
Episode 18: Score = 700.910
+++ SOlVED +++
Episode 19: Score = 780.998
Episode 20: Score = 81.448
INFO: 'my_controller' controller exited successfully.
```

As you can see in the above image, the agent was able to reach the goal successfully by using the camera sensor instead of distance sensors.

Analysis of the agent's training graph for the bonus question:





The explanations related to the graphs have been provided in the previous section. The graphs generated in this section also indicate that the training was conducted appropriately. Additionally, the result of the agent's test shows that using the camera instead of distance sensors was successful. The agent is able to avoid obstacles and move towards the goal by extracting useful information from the camera.