دانشجو
نرگس رازی زاده
شماره دانشجویی: ۴۰۱۴۴۳۱۷۴

تمرین دوم

درس ربات‌های متحرک خودگردان
استاد درس جناب آقای دکتر آرمین سلیمی بدر

The Proximal Policy Optimization algorithm is a reinforcement learning method that guides the agent using learning from experiences gained from the environment. In this algorithm, the agent tries to learn an optimal policy that can take action in each situation to maximize the total total profit.
PPO is a policy-based algorithm, in the sense that it tries to optimize the agent's decision-making policy. A policy is a mapping that transforms the state of the environment into action possibilities. The goal of PPO is to update the policy in a way that improves the agent's performance in the environment.
The PPO method works on the basis of gradual updating of the policy. This update is divided into two phases. In the first step, the agent evaluates a new objective function using the difference between the probabilities of actions in the new and old policies. In the second step, using this objective function, the policy parameters are updated so that the agent can provide the best performance in the environment.
An important feature of PPO is that it uses a clip function to prevent large policy changes, allowing updates to be made more consistently.
The clip function in the mathematical sense is an operator that, by applying it to a value, keeps that value within a certain range. In other words, the clip function ensures that a variable value is within a certain range and if it goes out of this range, it sets its value to the minimum or maximum possible value in the range.
In the context of reinforcement learning algorithms, the clip function is commonly used to limit large changes in policy parameters or probability distributions. In this way, this function helps the updates made in each step to be limited and stable, preventing sudden increases or large fluctuations in parameter changes. This can prevent instability problems in the training of reinforcement learning algorithms.
So, as we said, the PPO algorithm is a policy optimization method that provides improvements over previous algorithms such as TRPO (Trust Region Policy Optimization). In summary, the goal of this algorithm is to optimize policy and improve agent performance in complex environments.

Main differences between PPO and DQN
DQN algorithm
The objective function:
DQN is a value-based algorithm and deals with the approximate optimization of the action value function (Q-function). Its purpose is to minimize the error in predicting Q-values.
Action space:
DQN is usually used for problems with discrete action space. This means that the possible actions are limited to a discrete set of numbers.
Using experience memory:

DQN uses an experience memory (Experience Replay) to manage past experiences. This memory stores experiences and extracts random categories during training to improve the use of past experiences.
PPO algorithm
The objective function:
PPO provides improvements over policy optimization algorithms such as TRPO. This algorithm also aims to optimize the policy, but with constraints and clips to prevent large policy changes at each update.
Action space:
PPO is suitable for problems with continuous action space and can be handled by continuous actions.
Using the clip:
One of the important features of PPO is the use of clips to limit changes in the policy. This restriction prevents sudden and unstable changes in education.
Conclusion
Ultimately, the choice between PPO and DQN depends on the nature of the problem. PPO is suitable for problems with continuous action space and DQN is usually used for problems with discrete action space. Each of these algorithms has its advantages and limitations, depending on the situation of the problem, choosing one of them may be more suitable. Finally, in this exercise, PPO has been chosen as a model.
The provided code contains 3 classes SaveOnBestTrainingRewardCallback, Environment and Agent_REINFORCE.
:SaveOnBestTrainingRewardCallback
This class is a callback that is used in the training process to save the model whenever the best training reward is obtained.
The 'on_step' method is called after each training step.
The 'on_training_end' method is called at the end of training.
In this particular implementation, the callback keeps track of the best training reward seen so far and saves the model whenever a new best reward is obtained.
:Environment
This class represents the RL environment in which an agent interacts and learns.
A class usually contains methods such as ``reset'' to reset the environment, ``step'' to perform an action and observe the next state, rewards and termination conditions, and other environment-specific utility methods.
The environment class is responsible for providing observations, rewards, and termination signals to the agent based on its actions.
:Agent
This class represents an agent that uses the "PPO" algorithm from the Stable Baselines 3 library to train the agent.
The agent class usually contains methods such as 'train' to start the training process, 'act' to select an action according to the current state, and other utility methods.
The agent class interacts with the environment, gathering experience to improve its performance over time.
These three classes work together to train an RL agent using the PPO algorithm in a given environment. "SaveOnBestTrainingRewardCallback" helps to save the best model, "Environment" class represents the RL environment, and "Agent" class trains the agent using the PPO algorithm.

```python
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:])
              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(f"Saving new best model to {self.save_path}")
                  self.model.save(self.save_path)

        return True
```

This code defines a callback class named "SaveOnBestTrainingRewardCallback" that is used to save the model during the training process based on the training reward. The callback checks the training reward periodically and saves the model if a new best reward is obtained.
The _on_step method is called after each training step. Checks if the current number of steps ('n_calls') is a multiple of 'check_freq' (training reward check frequency). If so, continues to calculate the average training reward over the last 100 episodes.
The average reward is calculated using the "ts2xy" and "load_results" functions.
If the average reward is greater than the previous best average reward, the model is saved in the "save_path" location.
In general, this callback periodically checks the training reward and saves the model if a new best reward is achieved. It provides a convenient way to track training progress and save the best model.

Environment class

The code for this class defines a custom environment class called "Environment" that implements the bot environment in the Webots simulator. This environment is used to train an agent to move towards a target position while avoiding obstacles.

Here is an overview of each function in the "Environment" class:

```python
def __init__(self, max_steps):
    super().__init__()

    # General environment parameters
    self.max_speed = 1.5 # Maximum Angular speed in rad/s
    self.destination_coordinate = np.array([-0.03, 2.72]) # Target (Goal) position
    self.reach_threshold = 0.065 # 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])
    self.max_steps = max_steps

    # 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))))

    self.action_space = spaces.Discrete(3)

    self.observation_space = spaces.Box(low=0, high=1, shape=(3,), 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

    # 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)
```

This constructor function initializes the various parameters and devices used in the environment. Adjusts wheel sensors, GPS sensor, touch sensor and distance sensors. It also defines the action space and observation space for the environment.

```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
```

This function performs min-max normalization on the given value and scales it between 0 and 1.

```
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
```

This function calculates and returns the normalized distance from the current position of the robot to the target. It uses GPS sensor values and destination coordinates to calculate the distance.

```
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
```

This function retrieves and normalizes data from distance sensors. Collects values from distance sensors and applies normalization to scale values between 0 and 1.

```
def get_observations(self):
    """
    Obtains and returns the normalized sensor data and current distance to the goal.

    Returns:
    - numpy.ndarray: State vector representing distance to goal and distance sensors value.
    """

    normalized_sensor_data = np.array(self.get_sensor_data(), dtype=np.float32)
    normalizied_current_coordinate = np.array([self.get_distance_to_goal()], dtype=np.float32)

    state_vector = np.concatenate([normalizied_current_coordinate, normalized_sensor_data], dtype=np.float32)

    return np.array(state_vector)
```

This function gets and returns the normalized sensor data and the current distance to the target. It concatenates the normalized sensor data and the normalized distance to the target into a 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(), {}
```

This function resets the environment and returns the initial observations. It calls the functions necessary to reset the simulation and physics and returns the initial state vector.

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

This function takes steps based on the given action in the environment. It applies the action to the bot engines using the "apply_action" function, calculates the set reward and flag using the "get_reward" function, and returns the new status, stage reward, set flag, and additional

information.

```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_sensor_data = self.get_sensor_data()
    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
    # (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 +++')
        # break
    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 above function calculates and returns the reward based on the current state. It calculates the reward based on the distance to the target, hitting obstacles and completing the task. The reward is accumulated based on various conditions and returned along with a designated flag that can indicate reaching a goal or hitting an obstacle.

```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
            self.left_motor.setVelocity(self.max_speed)
            self.right_motor.setVelocity(self.max_speed)
        elif action == 1: # turn right
            self.left_motor.setVelocity(self.max_speed)
            self.right_motor.setVelocity(-self.max_speed)
        elif action == 2: # turn 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)
```

The function above applies the specified action to the robot's motors. Adjusts the speed of the left and right motors based on the action: move forward, turn right, or turn left. It also performs some simulation steps to allow the robot to perform the action.

These functions work together to define the behavior of the environment and allow an agent to interact with the environment, receive observations, perform actions, and receive rewards based on its actions.

Agent class
This code defines a reinforcement learning agent class called "Agent". Here is an overview of each function in the class:

```python
    def __init__(self, save_path,load_path, num_episodes, max_steps):
        self.save_path = save_path
        self.load_path = load_path
        self.num_episodes = num_episodes
        self.max_steps = max_steps

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

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

The constructor function above initializes various parameters such as save and load paths for the trained model, the number of episodes to train/test, and the maximum number of steps per episode. It also sets up the environment and network used for training.

```python
    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()
        # Helper from the library
        results_plotter.plot_results(
            [log_dir], 1e5, results_plotter.X_TIMESTEPS, "PPO"
        )
        self.plot_results2(log_dir)
```

The train function trains the agent using the Proximal Policy Optimization (PPO) algorithm. It learns from the environment for a specified number of episodes and stores the best model based on the training reward using a callback. After training, it draws the learning curve and saves it as a PNG file.

```python
    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 test function tests the performance of the agent by running it in the environment for a certain number of parts. It collects the points obtained in each part and prints the points of each part. The agent takes actions based on the predictions of the policy network and updates the state accordingly.

These functions work together to define the training and testing methods for the reinforcement learning agent. The agent interacts with the environment, learns from the rewards obtained, and updates its policy network to improve its performance over time.

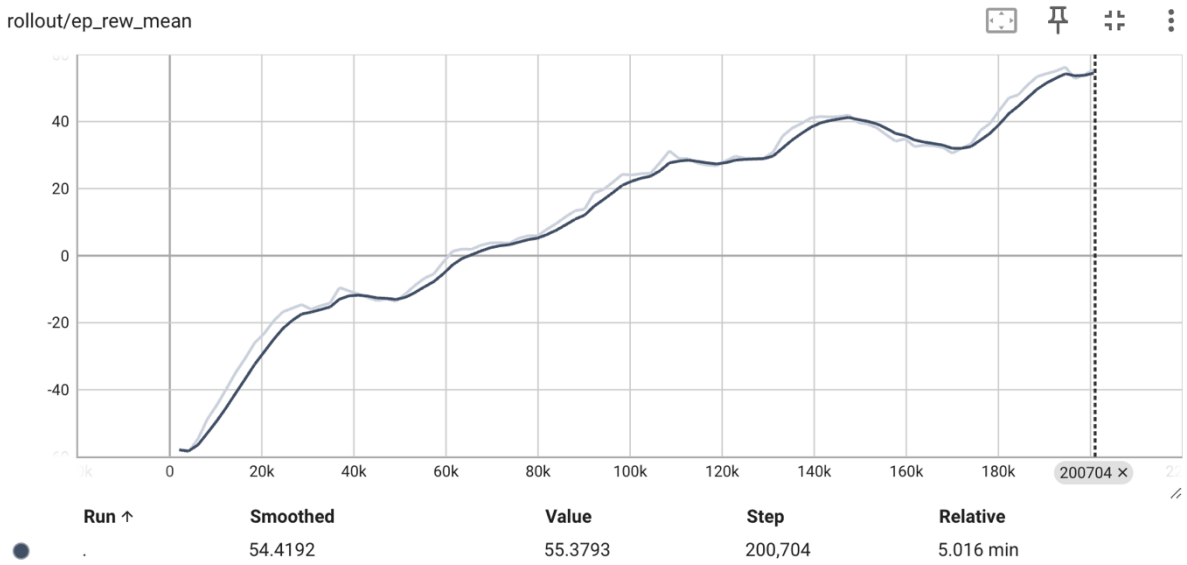
Example of training result

```
Num timesteps: 200000
Best mean reward: 56.23 - Last mean reward per episode: 54.68
+++ SOlVED +++
+++ SOlVED +++
+++ SOlVED +++
+++ SOlVED +++
+++ SOlVED +++
+++ SOlVED +++
+++ SOlVED +++
+++ SOlVED +++
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 84.5        |
|    ep_rew_mean          | 55.4        |
| time/                   |             |
|    fps                  | 659         |
|    iterations           | 98          |
|    time_elapsed         | 304         |
|    total_timesteps      | 200704      |
| train/                  |             |
|    approx_kl            | 0.0030681053 |
|    clip_fraction        | 0.0276      |
|    clip_range           | 0.2         |
|    entropy_loss         | -0.149      |
|    explained_variance   | 0.967       |
|    learning_rate        | 0.0003      |
|    loss                 | 0.401       |
|    n_updates            | 970         |
```

Sample test result

```
Using cpu device
Wrapping the env in a DummyVecEnv.
+++ SOlVED +++
Episode 1: Score = 59.103
+++ SOlVED +++
Episode 2: Score = 58.575
+++ SOlVED +++
Episode 3: Score = 59.468
+++ SOlVED +++
Episode 4: Score = 62.951
+++ SOlVED +++
Episode 5: Score = 58.352
+++ SOlVED +++
Episode 6: Score = 60.596
+++ SOlVED +++
Episode 7: Score = 61.089
+++ SOlVED +++
Episode 8: Score = 62.625
+++ SOlVED +++
Episode 9: Score = 59.192
+++ SOlVED +++
Episode 10: Score = 57.800
INFO: 'my_controller' controller exited successfully.
```

Charts



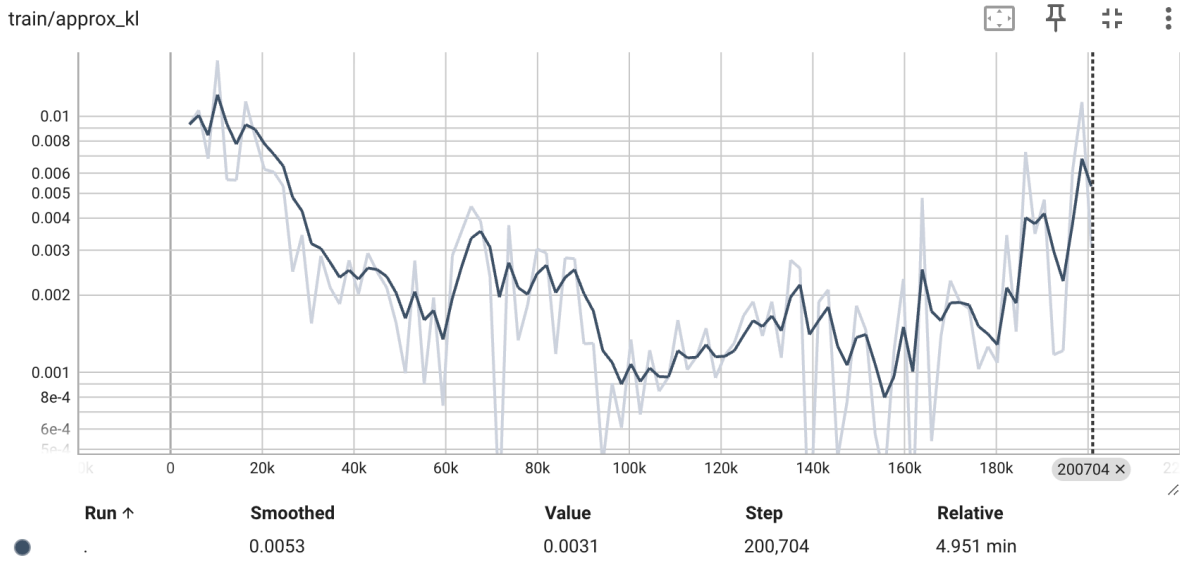
"ep_rew_mean" is a metric that measures the average reward of each episode in the supply phase of a reinforcement learning algorithm.

In reinforcement learning, an episode is a sequence of states, actions, and rewards that starts when the agent enters a new state and ends when an end state is reached or a maximum number of steps is exceeded. The propagation phase is the part of the learning process in which the agent interacts with the environment and gathers experiences (state, action, reward, next state) to learn from.

The "ep_rew_mean" metric calculates the average reward received by the agent across all episodes in the launch phase. It shows a measure of the agent's performance in terms of collecting rewards.

To calculate ep_rew_mean, you need to add up all rewards received by the agent in all episodes of the launch phase and divide that number by the total number of episodes.

A higher ep_rew_mean indicates that the agent receives more rewards on average, indicating that it is learning to make better decisions in the environment. However, it is important to note that a high ep_rew_mean does not necessarily mean that the agent is optimal or close to optimal. It is possible that the agent receives high rewards due to randomness or exploitation of certain aspects of the environment, rather than true learning.



In Proximal Policy Optimization (PPO), the metric "approx_kl" stands for "approximate Kullback-Leibler divergence" and is used to measure the difference between the current policy and the new policy.

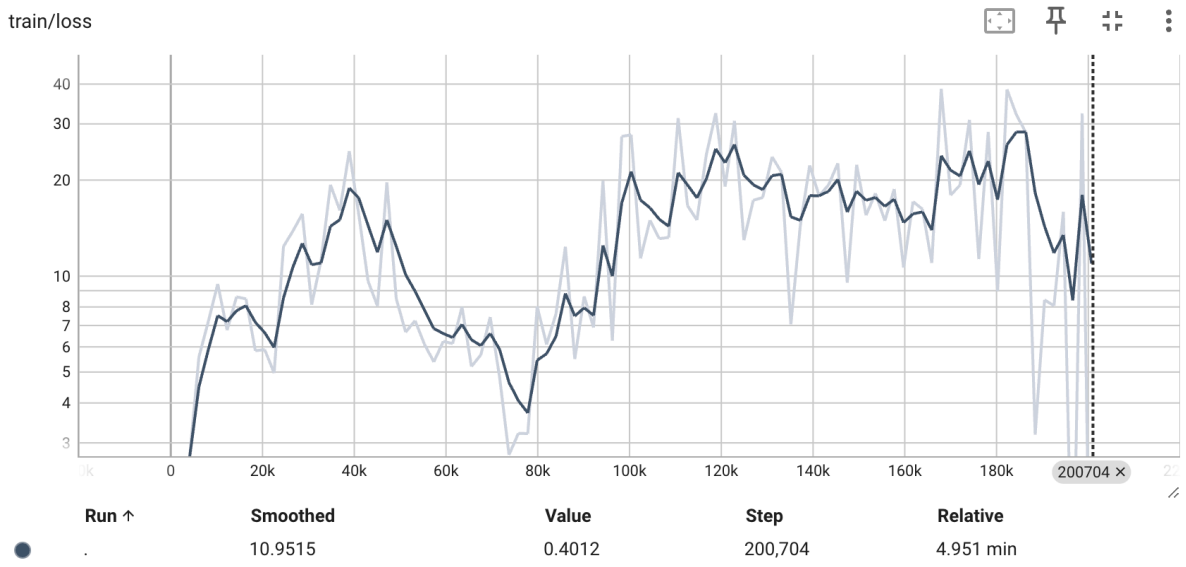
In PPO, the new policy is usually a predefined policy that specifies the desired behavior or strategy for the agent. On the other hand, the current policy is the policy that the agent has learned so far through interaction with the environment.

The Kullback-Leibler (KL) divergence is a measure of the difference between two probability distributions. In the context of PPO, it is used to compare the agent's policy distribution (i.e., the probability of taking each possible action in each state) with the distribution of the new policy.

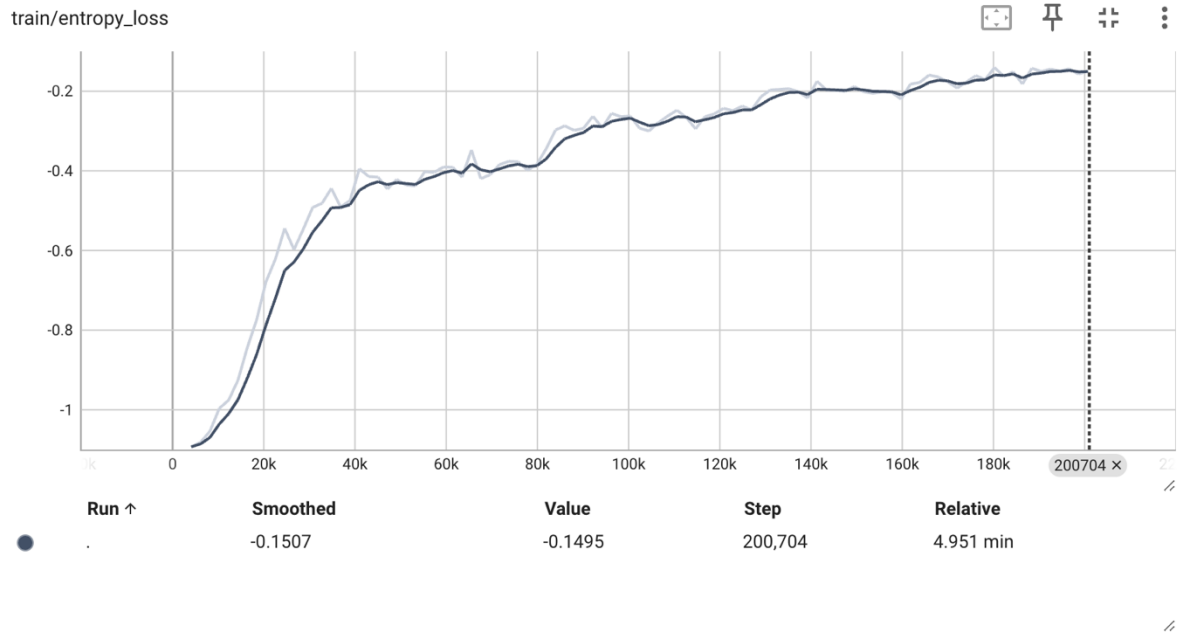
The goal of PPO is to minimize the value of approx_kl, which means that the agent learns to accept the new policy as its policy. By monitoring the value of approx_kl during training, we can evaluate how well the agent learns and whether it converges to the desired behavior.

In general, the approx_kl metric in PPO provides a way to measure the difference between the current policy and the new policy, and is an important component of the algorithm's

objective function. By minimizing the value of approx_kl, the agent learns to adopt the desired behavior and optimize its policy relative to the new policy.



In Proximal Policy Optimization (PPO), the loss function is a key component of the algorithm that measures the difference between the policy learned by the agent and the new policy. The value of the loss function is a measure of how well the policy performs compared to the new policy.



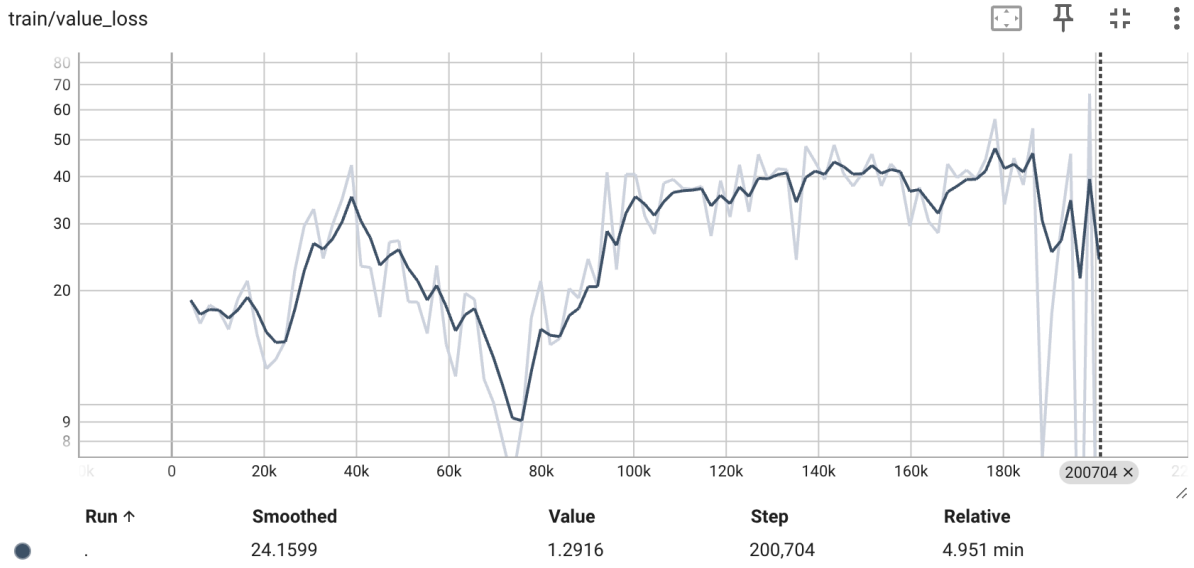
In the context of policy optimization (PPO), the entropy_loss metric measures the policy's deviation from a uniform distribution. In other words, it quantifies how different the policy is from a random policy, where all actions are equally likely.

Entropy is a measure of uncertainty or randomness of a system. In the case of PPO, policy entropy is a measure of how unpredictable the policy is. A policy with high entropy is more random, while a policy with low entropy is more deterministic.

Intuitively, a policy with low entropy loss is preferred because it indicates that the policy is less predictable and more exploratory. A policy that always selects an action in a given state is highly predictable and has high entropy, while a policy that randomly selects an action in a given state is less predictable and It has low entropy.

By minimizing entropy loss, the agent is encouraged to explore the environment efficiently and avoid getting stuck in local optima. A good policy should balance exploration and exploitation, and entropy loss by promoting exploration helps achieve this balance.

In summary, the entropy loss in PPO measures the degree to which the policy deviates from a uniform distribution and is used as a tuning term to encourage the policy to explore the environment effectively.



In the context of proximal policy optimization (PPO), the "value_loss" metric measures how well the policy estimates the value of the environment. In other words, it assesses how well the policy predicts the expected return or utility of taking a particular action in a particular situation.
By minimizing the loss of value, the agent learns an accurate estimate of the value of the environment, which is essential for making informed decisions and maximizing expected returns.

analysis
According to the videos attached in the training file, the robot finds the path to reach the goal well in 10 performed tests. This is because in the observations of the robot, the distance between the robot and the target is seen using GPS, and the robot realizes during training that when the distance between the robot and the target decreases, the robot receives a reward, and it learns this relationship well, and during testing. He also gets this connection in his

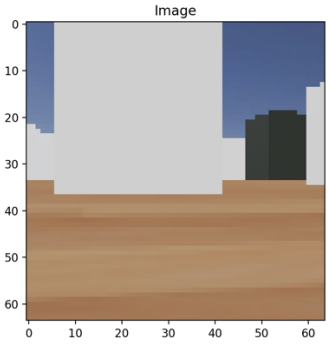
observations and knows that he will get a reward by reducing his distance to the target, so he knows in which direction to move to get the most points.
Now, by moving the robot to other places, the robot takes the previous approach and tries to avoid the obstacles in order not to be punished and tries to get closer to the target to receive a reward.
By moving the robot around and testing it, in most cases, it gets the answer in 10 episodes. In only one test, it reached 60% accuracy.

```
Using cpu device
Wrapping the env in a DummyVecEnv.
+++ SOlVED +++
Episode 1: Score = 84.533
INFO: Creating video...
Recording at 31.25 FPS, 1318359 bit/s.
Video encoding stage 1... (please wait)
+++ SOlVED +++
Episode 2: Score = 84.211
+++ SOlVED +++
Episode 3: Score = 85.118
+++ SOlVED +++
Episode 4: Score = 86.804
Episode 5: Score = -1.279
Episode 6: Score = -1.556
Episode 7: Score = -1.279
+++ SOlVED +++
Episode 8: Score = 88.555
+++ SOlVED +++
Episode 9: Score = 88.161
+++ SOlVED +++
Episode 10: Score = 89.278
Using cpu device
Wrapping the env in a DummyVecEnv.
+++ SOlVED +++
Episode 1: Score = 59.103
+++ SOlVED +++
Episode 2: Score = 58.575
+++ SOlVED +++
Episode 3: Score = 59.468
+++ SOlVED +++
Episode 4: Score = 62.951
+++ SOlVED +++
Episode 5: Score = 58.352
+++ SOlVED +++
Episode 6: Score = 60.596
+++ SOlVED +++
Episode 7: Score = 61.089
+++ SOlVED +++
Episode 8: Score = 62.625
+++ SOlVED +++
Episode 9: Score = 59.192
+++ SOlVED +++
Episode 10: Score = 57.800
INFO: 'my_controller' controller exited successfully.
```

Score question

In this part, we have to train the robot to reach the destination using camera and GPS. To use the camera in this exercise, we use the histogram of the image taken from the camera. First, we make the walls and obstacles white and place a box with the opposite color of the walls and obstacles at the destination. The color of the walls and obstacles are white and the color of the destination box is black. In this case, by taking the image from the camera and taking it to gray scale and drawing its Histogram, our work becomes somewhat easier because the destination box and other objects become more distinct.

Image taken from the camera:



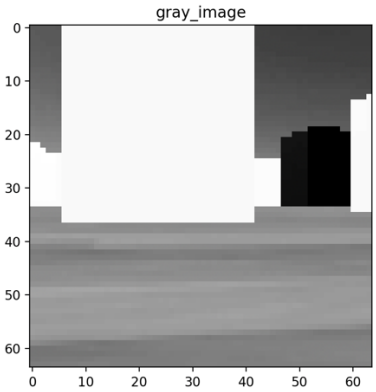
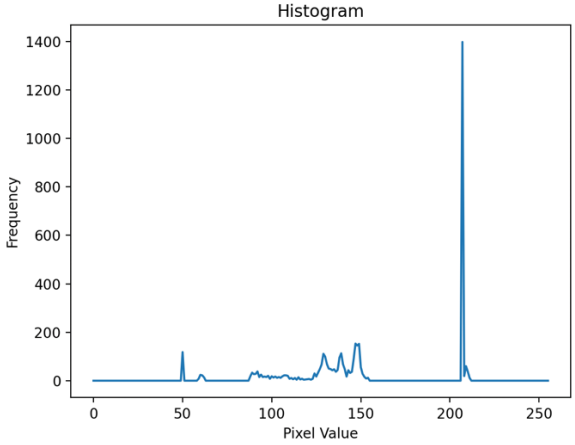
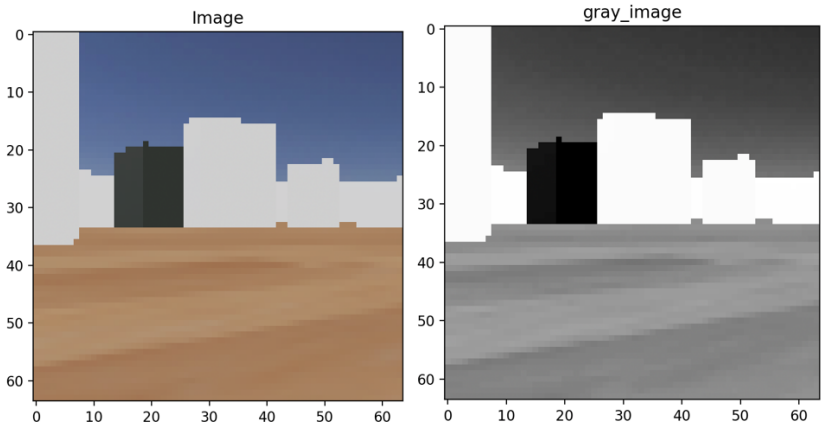
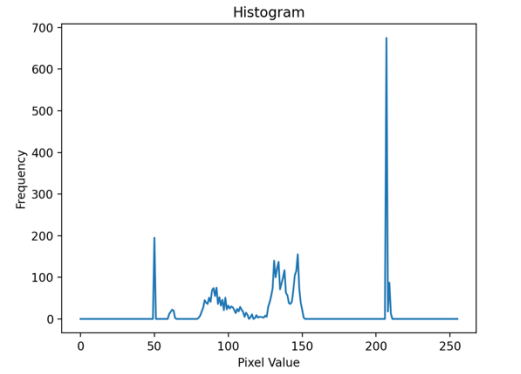


Image Histogram:



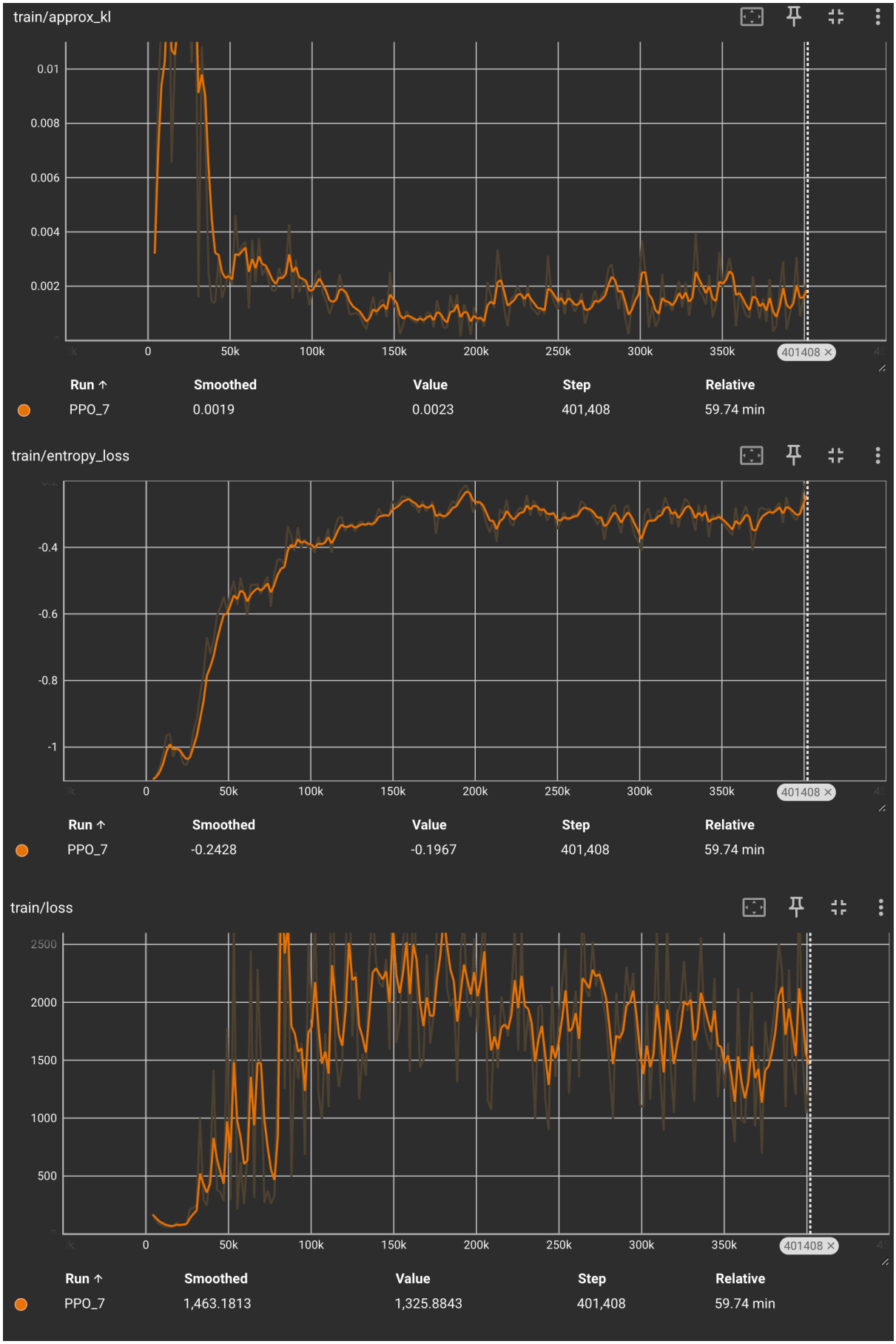


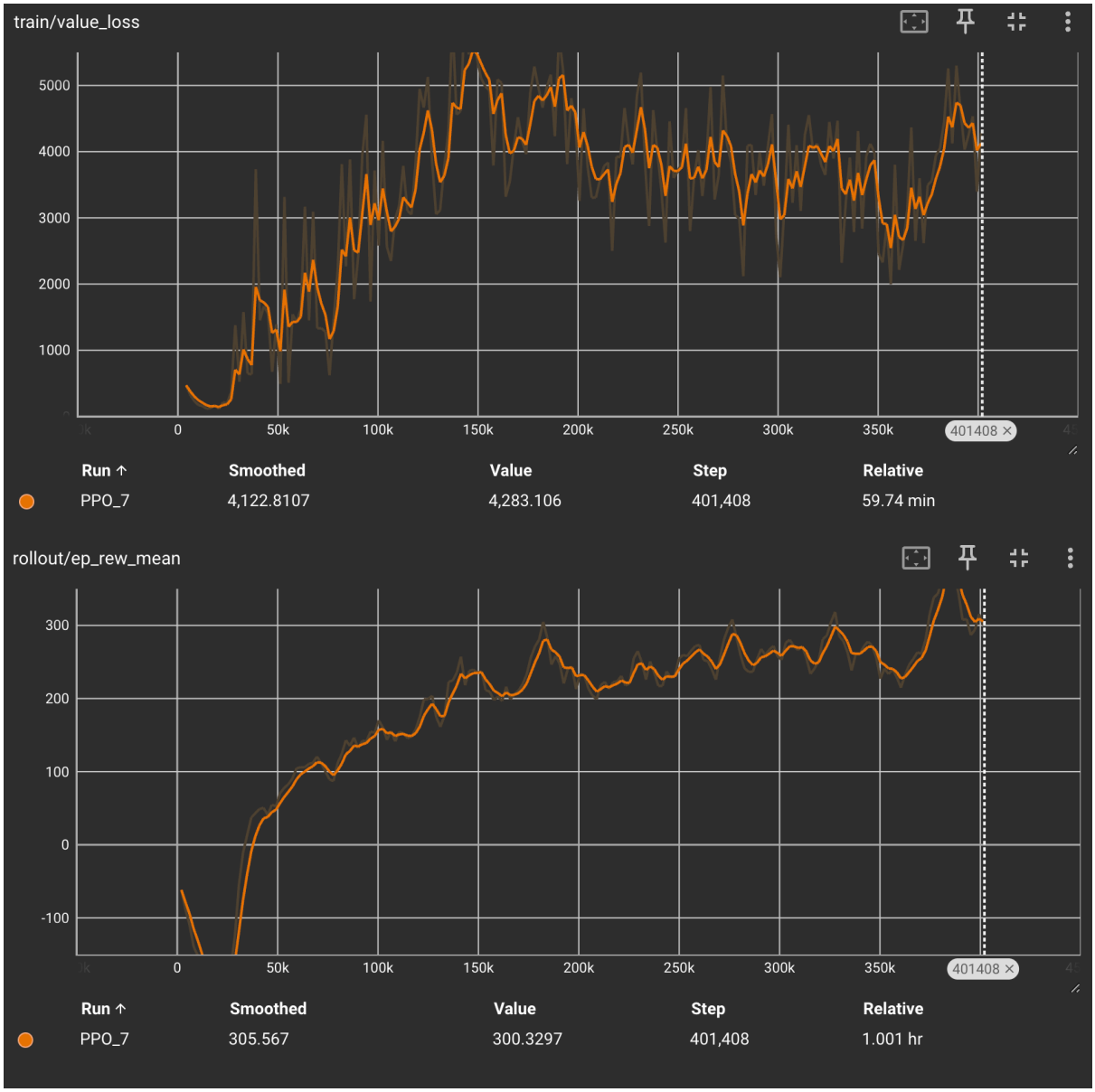


So, the sum of the histogram of the black destination box is calculated up to 50. And 200 onwards is for boxes and white walls.
Using these values, we determine the rewards and punishments of the robot.
PPO is used in this section as in the previous section.
Punishments and rewards are set in such a way that when you get too close and see a lot of white color (means getting close to walls and obstacles), a penalty will be given, and if you see black color (target), a reward will be given.

According to the implementation, we got the following results.

```
Using cpu device
Wrapping the env in a DummyVecEnv.
./results/best_model
+++ SOlVED +++
Episode 1: Score = 458.732
Episode 2: Score = 356.779
Episode 3: Score = 133.171
+++ SOlVED +++
Episode 4: Score = 415.026
Episode 5: Score = 561.000
Episode 6: Score = 231.797
Episode 7: Score = 349.671
Episode 8: Score = 385.343
Episode 9: Score = 175.661
Episode 10: Score = 459.162
INFO: 'my_controller' controller exited successfully.
```


train/approx_kl
0.01
0.008
0.006
0.004
0.002
0
50k
100k
150k
200k
250k
300k
350k
401408
Run
Smoothed
Value
Step
Relative
PPO_7
0.0019
0.0023
401,408
59.74 min
train/entropy_loss
-0.4
-0.6
-0.8
-1
0
50k
100k
150k
200k
250k
300k
350k
401408
Run
Smoothed
Value
Step
Relative
PPO_7
-0.2428
-0.1967
401,408
59.74 min
train/loss
2500
2000
1500
1000
500
0
50k
100k
150k
200k
250k
300k
350k
401408
Run
Smoothed
Value
Step
Relative
PPO_7
1,463.1813
1,325.8843
401,408
59.74 min


train/value_loss
5000
4000
3000
2000
1000
0
50k
100k
150k
200k
250k
300k
350k
401408
Run
Smoothed
Value
Step
Relative
PPO_7
4,122.8107
4,283.106
401,408
59.74 min
rollout/ep_rew_mean
300
200
100
0
-100
0
50k
100k
150k
200k
250k
300k
350k
401408
Run
Smoothed
Value
Step
Relative
PPO_7
305.567
300.3297
401,408
1.001 hr