Reinforcement Learning

Assignment Report

Prashant Raj Shrestha
Oliver Obst | Yi Guo
31 October 2023

Final Report - 22027722

Introduction

Reinforcement learning is the process of learning through mistakes. To put it to action through the given code, several changes had to be made to answer the questions in the assessment.

The questions provides a robot with 3 wheel that can be moved around by setting each wheel's velocity between (-1 and 1). Similarly, there exists an orange ball (target) along with the robot in a 2-dimensional environment. The aim of the robot is to move and get to the orange ball for which the robot gets a reward of 10 points. However, for every step that the robot has not reached to goal, it gets penalised by a small value of 0.01 points.

Q1. Using a fixed target

1.1 Step Reduction and Action Listing

The robot is represented by the robot's pose which is given by (x, y, and θ). x and y signify the coordinates of the robot is the 2-dimensional environment while the θ represents the orientation of the robot. Similarly, the target is represented by ($target_x, target_y$) which represents the coordinates of the target.

To simplify the representation and to implement reinforcement learning, a function is created to convert the infinitesimal representation into discrete states. The constructed function is given by `getState(x, y, theta, target_pos)` which takes `x`, `y` and `theta` as the robot's position and orientation, and `target_pos` as the position of the target. The `dx` and `dy` variables are used to find the relative distance between the target and the robot along with the `angle` to calculate the orientation.

Using the position of the target from the robot's perspective, 4 unique states are constructed:

- **State 0:** Ball in front of robot
- **State 1:** Ball behind the robot
- **State 2**: Ball to the left of the robot
- **State 3:** Ball to the right of the robot

The code for the function is provided in the snippet below:

```
def getState(x, y, theta, target_pos):
        dx = target_pos[0] - x
        dy = target_pos[1] - y
        angle = math.degrees(math.atan2(dy, dx)) - theta
```

```
        # print(angle)
        if angle < -180:
                angle += 360
        elif angle > 180:
                angle -= 360

        if abs(angle) <= 45:
                state = 0 # ball in front of the robot
        elif abs(angle) >= 135:
                state = 1 # ball behind the robot
        elif angle < -45:
                state = 2 # ball to the left of the robot
        elif angle > 45 and angle < 135:
                state = 3 # ball to the right of the robot

        # print(state)
        return state
```

For creating a set of actions, the velocity of each wheel was mapped into a tuple and then permuted through 3 times (for all three wheels) to get the total list of actions. The code for the action listing is attached below

```
permutations_list = list(product((-0.5, 0.0, 0.5), repeat=3))
ACTIONS = [perm for perm in permutations_list]
```

1.2 Epsilon Greedy Action Selection and Q-Policy

As the first question works on the basis of Tabular Reinforcement, an ϵ-greedy action selection is used while a Q-table is constructed to store the policy. Initially set to 1, the ϵ value is gradually decayed to control the ration between exploration and exploitation.

```
EPSILON = max(EPSILON *( 1 - 0.02), 0.01)
```

A Q-table is constructed which stores values in (State, Action) pairs. As the program iterates through the steps, the Q-table is updated on the basis of the current state, action, obtained reward, and the estimated future reward.

```
# Initalise Q-table
Q = np.zeros((NUM_STATES, NUM_ACTIONS)) # done outside the while loop

# Update Q-table
next_state = getState(x, y, theta, target_pos)
next_action = np.argmax(Q[next_state])
```

```
Q[state, action] += ALPHA * (reward + GAMMA * Q[next_state, next_action] -
Q[state, action])
```

1.3 Reward Structure

The reward structure leans strongly towards positive reinforcement as the robot is rewarded highly when it meets the target (by 10 points). However, the question suggested that the robot to be penalised by 0.01 points at every step the target is not met. I have modified this part of the logic quite heavily by trying to make the movement linear as the target is stationary. Using the logic below, I have penalised excessive turning by a small amount to encourage the robot to move in a linear fashion.

```
elif step > 1 and action != previous_action:
        # Penalty for turning
        reward = -0.01
```

Another major modification to the reward structure is that, I have encourage the robot to converge faster by reward the steps that have decreased the distance between the target and the robot. In other words, if the robot gets closer to the target, it gets a small reward.

```
elif current_distance_to_target < distance_to_target:
        # Reward if the distance to the target is decreasing
        reward = 0.1
```

These additions improved the performance of the model significantly.

1.4 Training and Convergence

The model is trained until enough episodes are iterated over until the desired performance is achieved. The training and reward values are stored to create a training-reward plot which shows the training performance over time.

1.5 Testing

The test is carried out to measure the robot's performance under specific controlled conditions.

To conduct the testing, a modified version of the original file is used. The policy file is loaded onto the Q-table which was previously saved in the training loop. Thereafter, using the policy, 50 episodes are run to measure the robot's performance with the help of *Average Test Reward* and *Standard Deviation*. The ϵ is also set to 0, imposing exploitation on the robot instead of exploration.

```
# Loading the trained policy file
Q = np.load("./Trained Files/policy.npy")

# Testing
test_rewards = []
for episode in range(50):
        score = 0
        robot_pos = [random.randint(FIELD.left , FIELD.right),
random.randint(FIELD.top, FIELD.bottom), random.randint(0, 359)]

        target_pos = [random.randint(FIELD.left , FIELD.right),
random.randint(FIELD.top, FIELD.bottom)]
        steps = 1000
```

```
        for step in range(steps):
                current_state = getState(robot_pos[0], robot_pos[1],
robot_pos[2], target_pos)
                action = np.argmax(Q[current_state])
                omega_0 = ACTIONS[action][0]
                omega_1 = ACTIONS[action][1]
                omega_2 = ACTIONS[action][2]
                robot_pos_prime = update_pose(robot_pos[0], robot_pos[1],
robot_pos[2], omega_0, omega_1, omega_2, 0.1)

                reward, flag= rewardFunction(robot_pos_prime, target_pos)
                robot_pos = robot_pos_prime
                score += reward
                # if not flag:
                        # break
test_rewards.append(score)
```

1.6 Model Performance

Running the code above, the test-average outputs the cumulative reward for each episode. It is used as the measure of the training policy's performance. The standard deviation of the test-average is also used, which shows the variation in performance.

Using the code below, the measures were outputted for analysis

```
print("Average reward:", sum(test_rewards)/len(test_rewards))
print("Standard deviation:", np.std(test_rewards))
print("Testing rewards:", test_rewards)
print("Training rewards:", training_rewards)
print("Highest reward:", max(test_rewards))
```

```
Average reward: 51.29016449406043
Standard deviation: 221.7261139828191
Testing rewards: [ -3.4372480371761593, -5.336435861707259,
-3.2530984506832104, -2.4016044169927047, -1.9917151790841565,
-1.853065369409576, -4.269372669079555, -2.8079052976726953,
-1.8870244987587863, -5.4767141172030875, -11.812987255938657,
-7.843796770017554, -2.0588553777077654, -4.4724403435023445,
-3.3241601747917158, -3.2659820102451445, -2.0060272720949057,
-6.440147440765158, -10.679967002829503, -2.0499860733565987,
-1.9631397787001337, -4.882254801262283, -2.4711771823838715,
-7.4662121387989, -2.290218014943289, 1000, -7.309143819978339,
-2.9935580373411743, -9.6535562982012, -4.145614263683289,
-8.592167497068855, 767.1313907790023, -2.6130834492036783,
-1.9799140512016495, -2.812688616543688, -2.5568736561102505,
-1.737759752707237, -2.225246441217819, -6.371563035491016,
-4.503238710948073, -2.4913047112693323, -3.6377991110976002, 1000,
```

```
-7.160114660879809, -5.221181719647117, -3.5311643765368226,
-4.768295246055107, -1.498391534826723, -5.889590484284826,
-5.189381066583019 ]
Training rewards: [ 0.   38.63 34.22 ... 99.33 99.46 85.83]
Highest reward: 1000
```

From the output above, we can see that the average reward is 51.29 with a very high standard deviation of 221.72. Focusing on the average reward, a positive value indicates good model performance, however, the high standard deviation shows major inconsistencies in the model performance. This is evident in the shell output above which shows a lot of small negative values (highlighted by yellow) while scare counts of large positive values (highlighted by purple).

The highest reward is 1000 which is the maximum, so the model can perform extremely well in some situations.

Q2. Using a moving target

2.1 Step Reduction and Action Listing

Similarly, as was the case in the previous question, the robot is represented by the robot's pose which is given by (x, y, and θ). x and y signify the coordinates of the robot is the 2-dimensional environment while the θ represents the orientation of the robot. Similarly, the target is represented by ($target_x, target_y$) which represents the coordinates of the target. However, there is one more addition to the target's parameters – the target velocity.

The function for step reduction is created to convert the continuous representation into discrete states. The constructed function is given by `getState(x, y, theta, target_pos, target_velocity)` which takes `x`, `y` and `theta` as the robot's position and orientation, and `target_pos` and `target_velocity` as the position and velocity of the target respectively. The `relative_position_rotated` uses the `create_rotation` function to generate the relative velocity of the target with respect to the robot which is expressed in the robot's coordinate system.

The create rotation is given by

```
def create_rotation_matrix(theta):
        return np.array([[np.cos(-theta), -np.sin(-theta)],
        [np.sin(-theta), np.cos(-theta)]])
```

Using the variables below, a rather complicated set of states are formed which is given by the code below

```
def getState(x, y, theta, target_pos, target_velocity):
# Unpack the target position
```

```
x_target, y_target = target_pos

relative_position = np.array([x_target, y_target]) - np.array([x, y])
rotation_matrix = create_rotation_matrix(theta)
relative_position_rotated = np.dot(rotation_matrix, relative_position)

x_rel_rotated = relative_position_rotated[0]
y_rel_rotated = relative_position_rotated[1]

if y_rel_rotated > 0 and abs(x_rel_rotated) < ROBOT_RADIUS:
        state_position = 0
elif y_rel_rotated > 0 and x_rel_rotated < 0:
        state_position = 1
elif y_rel_rotated > 0 and x_rel_rotated >= 0:
        state_position = 2
elif y_rel_rotated <= 0 and x_rel_rotated < 0:
        state_position = 3
else: # y_rel_rotated <= 0 and x_rel_rotated >= 0
        state_position = 4

# Calculate horizontal_velocity
if target_velocity[0] >= 0.5:
        horizontal_velocity = 0
elif -0.5 <= target_velocity[0] < 0.5:
        horizontal_velocity = 1
elif target_velocity[0] < -0.5:
        horizontal_velocity = 2

# Calculate vertical_velocity
if target_velocity[1] >= 0.5:
        vertical_velocity = 0
elif -0.5 <= target_velocity[1] < 0.5:
        vertical_velocity = 1
elif target_velocity[1] < -0.5:
        vertical_velocity = 2

# Combine the position and velocity states
state = state_position * 9 + horizontal_velocity * 3 + vertical_velocity
return state
```

This code enabled both the robot and the moving target to be represented fully using three variables `state_position`, `horizontal_velocity`, and `vertical_velocity`. There are four states, three horizontal and vertical velocity states which permutes to 45 different states. It is a big larger than ideal but I opted to capture the fuller picture.

The constructed states are segregated by

- ***State Position***

    - **State Position 0:** The target is directly in front of the robot
    - **State Position 1:** The target is towards the left of the robot
    - **State Position 2:** The target is towards the right of the robot
    - **State Position 3:** The target is behind the robot
- ***Horizontal Velocity***
    - **Horizontal Velocity 0:** When the velocity of the target is greater than 0.5
    - **Horizontal Velocity 1:** When the velocity of the target is between -0.5 and 0.5
    - **Horizontal Velocity 2:** When the velocity of the target is less than -0.5
- ***Vertical Velocity***
    - **Vertical Velocity 0:** When the velocity of the target is greater than 0.5
    - **Vertical Velocity 1:** When the velocity of the target is between -0.5 and 0.5
    - **Vertical Velocity 2:** When the velocity of the target is less than -0.5

2.2 Epsilon Greedy Action Selection and Q-Policy

The action selection was the same as the first question and the same Q-table with (state, action) pairs were used.

2.3 Reward Structure

Similar to question 1, the reward structure has been slightly modified to account for the velocity of the target.

```
if step > 1000:
        # Start a new episode if the episode has timed out
        [x, y, theta], target_pos, target_vel, episode, step,
episode_reward = new_episode(episode, episode_reward)
        reward = 0

elif not FIELD.collidepoint(x, y):
        # Penalize the robot for going out of bounds
        [x, y, theta], target_pos, target_vel, episode, step,
episode_reward = new_episode(episode, episode_reward)
        score -= 1
        reward = -10

elif current_distance_to_target <= ROBOT_RADIUS:
        # Reward the robot for colliding with the target
        [x, y, theta], target_pos, target_vel, episode, step,
episode_reward = new_episode(episode, episode_reward)
        reward = 10
        score += 10

elif current_velocity_towards_target > previous_velocity_towards_target:
```

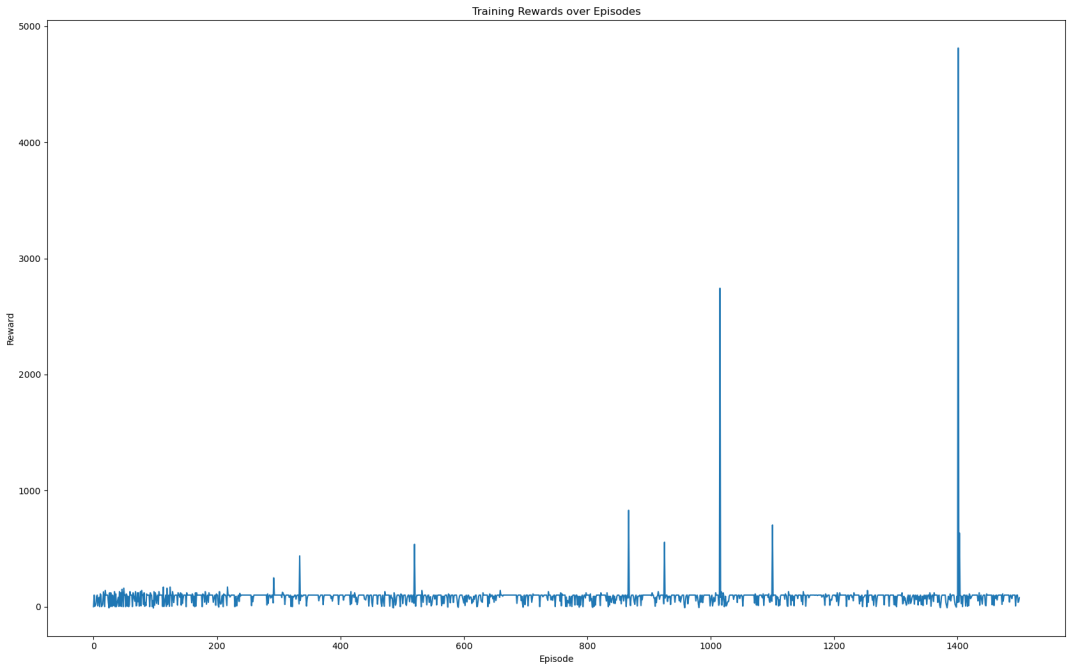
```
reward = 0.1
score += 0.01
```

Looking at the code snippet above, the first three conditions are the same as that on question 1. However, instead of measuring the distance between the robot and the target, the velocity difference is measure. This is done by using a function to account for the changes in velocity and update the new ones. The code for that is attached below

```
previous_distance_to_target = current_distance_to_target
previous_velocity_towards_target = current_velocity_towards_target
current_distance_to_target = distance([x, y], target_pos)

# Calculate the vector pointing from the robot to the target
target_vector = np.array([target_pos[0] - x, target_pos[1] - y])

# Normalize the target vector to get the unit vector
target_unit_vector = target_vector / np.linalg.norm(target_vector)

# Calculate the dot product of the robot's velocity vector and the target
unit vector
current_velocity_towards_target = np.dot(np.array([V_x_rotated,
V_y_rotated]), target_unit_vector)
```

2.4 Training and Convergence

The model is trained until enough episodes are iterated over until the desired performance is achieved. The training and reward values are stored to create a training-reward plot which shows the training performance over time. The plot is show below



This shows some leakage in rewards which can be contributing to errors in testing.

2.5 Testing

While testing, the following code was used to measure the average test rewards and standard deviation. The code is identical to that of question of except for the addition of a few line that contribute to the target velocity

```
test_rewards = []
previous_distance_to_target = 1000

for episode in range(50):
        score = 0
        robot_pos = [random.randint(FIELD.left , FIELD.right),
random.randint(FIELD.top, FIELD.bottom), random.randint(0, 359)]
        target_pos = [random.randint(FIELD.left , FIELD.right),
random.randint(FIELD.top, FIELD.bottom)]
        target_vel = [random.randint(-1, 1), random.randint(-1, 1)]

        steps = 1000
        for step in range(steps):
                current_state = getState(robot_pos[0], robot_pos[1],
robot_pos[2], target_pos, target_vel)
                action = np.argmax(Q[current_state])
                omega_0 = ACTIONS[action][0]
                omega_1 = ACTIONS[action][1]
                omega_2 = ACTIONS[action][2]
                robot_pos_prime = update_pose(robot_pos[0], robot_pos[1],
robot_pos[2], omega_0, omega_1, omega_2, 0.1)
```

```
                reward, current_distance_to_target=
rewardFunction(robot_pos_prime, target_pos, previous_distance_to_target)
                previous_distance_to_target = current_distance_to_target
                robot_pos = robot_pos_prime
                score += reward
                target_pos[0] += target_vel[0]
                target_pos[1] += target_vel[1]
        test_rewards.append(score)
```

2.6 Model Performance

Running the code above, the test-average outputs the cumulative reward for each episode. It is used as the measure of the training policy's performance. The standard deviation of the test-average is also used, which shows the variation in performance.

```
Average reward: -514.6837999999998
Standard deviation: 1969.4011298380938
Testing rewards: [ -323.59000000000106, -9.889999999999834,
11.229999999999592, -9.889999999999834, 19.589999999999065,
-9.999999999999831, 2.8700000000001626, 4.850000000000115,
-9.889999999999834, 3.860000000000128, 12.439999999999413,
-9.889999999999834, -9.889999999999834, -8.23999999999987,
15.189999999998996, -10000, -9.889999999999834, -9.889999999999834,
-9.889999999999834, -9.889999999999834, -9.889999999999834,
-9.889999999999834, 34.22000000000023, -9.889999999999834,
-9.889999999999834, 34.77000000000129, -9454.71, -8.569999999999864,
-9.889999999999834, 5.730000000000148, -5.5999999999998815,
-3.3999999999998933, 14.639999999999079, 5.29000000000015, -3306.59,
-9.889999999999834, 1.2200000000001539, -9.889999999999834,
-3.9499999999998905, -2595.1899999999996, 33.0100000000015,
-9.889999999999834, -9.889999999999834, 8.150000000000055,
-9.889999999999834, -9.889999999999834, 0.45000000000014073,
-6.149999999999881, -9.889999999999834, -8.019999999999875]
Training rewards: [  0.  100.    3.6 ...  37.   46.1  77. ]
Highest reward: 34.77000000000129
```

Looking at the results, the model performance was very poor. Th average reward is very low at -514.68 and extremely high standard deviation at 1969.4. This tells us that the model was poorly constructed and requires more work on the reward as well as state selection.

This is also backed by the leakage in the reward seen in the graph at heading 2.4.

Q3. Using Neural Network for Reinforcement Learning

3.1 Neural Network Architecture

The neural network that was constructed was made up of the following elements:

- **Input Layer:** The input size is set to 2 which takes the position of the robot (x, y)
- **Hidden Layers:** There are two hidden layers of size 256 and 128 respectively. The activation function is ReLU.
- **Output Layer:** The output layer has 8 outputs as there are 8 possible actions.

```
class Policy(nn.Module):
        def __init__(self, input_size, hidden_size, hidden_size2,
output_size):
                super(Policy, self).__init__()
                self.linear1 = nn.Linear(input_size, hidden_size)
                self.linear2 = nn.Linear(hidden_size, hidden_size2)
                self.linear3 = nn.Linear(hidden_size2, output_size)

        def forward(self, state):
                x = F.relu(self.linear1(state))
                x = F.relu(self.linear2(x))
                x = F.softmax(self.linear3(x), dim=-1)

                return x


# Creating the policy network
input_size = 2
hidden_size = 256
hidden_size2 = 128
actions = [0, 1, 2, 3, 4, 5, 6, 7]
output_size = len(actions)
policy = Policy(input_size, hidden_size, hidden_size2, output_size)
optimizer = optim.Adam(policy.parameters(), lr=0.01)
gamma = 0.99
```

For training, the learning rate is set of 0.01 using the Adam Optimiser while the Discount Factor γ is set is 0.99.

1. At the start of each training iteration, we reset the optimiser to prepare for training.
2. **Logits:** The policy network generates logits, which are numerical values representing the likelihood of taking different actions.
3. **Probability Distribution:** We convert these logits into a probability distribution by applying the softmax function, which helps us determine the probabilities of each action.
4. **Action Selection:** Actions are chosen based on this probability distribution, with actions having higher probabilities being more likely to be selected.
5. **Calculating Log Probabilities:** We calculate the natural logarithm of the probabilities of the actions that were chosen.
6. **Loss Calculation:** The loss is determined by taking the negative mean (average) of the log probabilities, which is then multiplied by the corresponding rewards obtained during

training.

7. **Backpropagation:** The calculated loss is used to perform backpropagation, adjusting the neural network's weights to improve its performance in future iterations.

3.2 Training Process

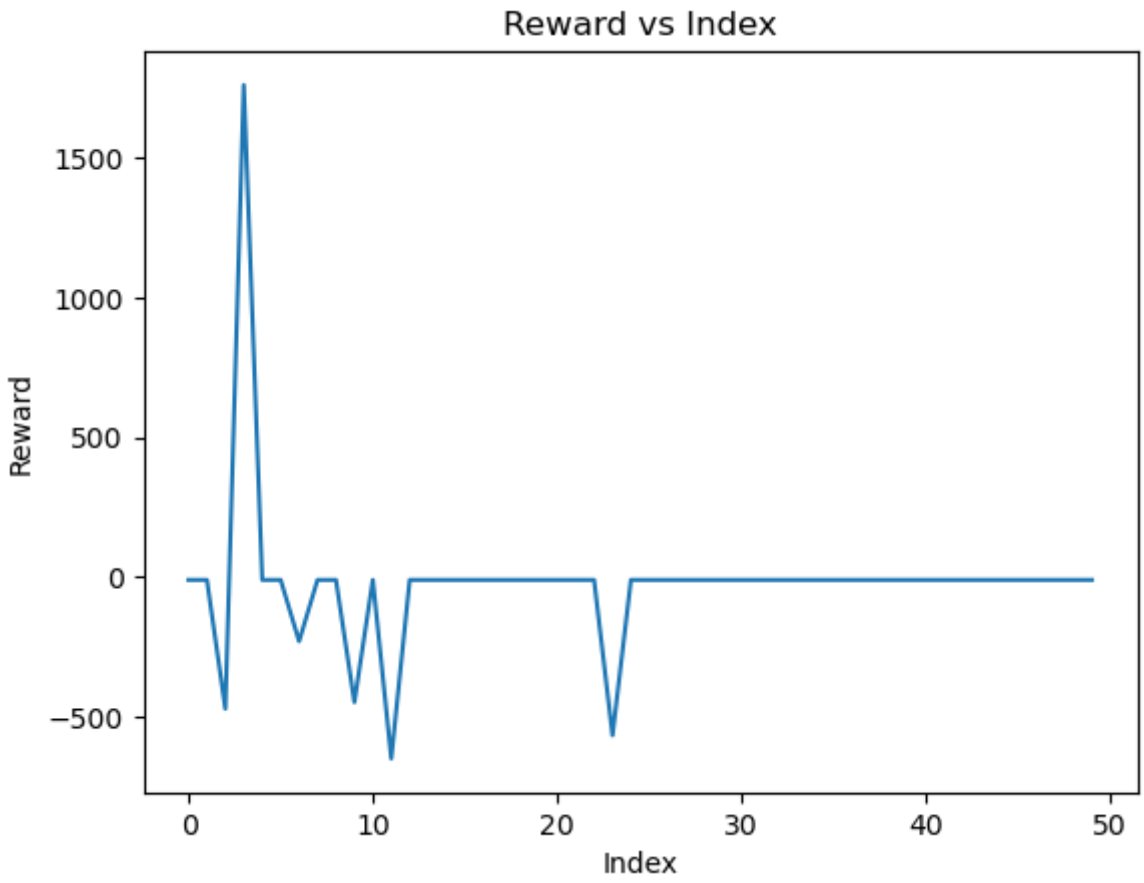
The training process is iterated every episode for the neural network to learn the patterns of the environment.

- **Policy Network Action Selection**:
  - The policy network (`policy`) takes the current state (robot position `x` and `y`) as input and generates action probabilities using a softmax function. An action is then sampled based on these probabilities.
- **Action Execution and Pose Update**:
  - The selected action is used to update the robot's position by modifying the control values (omega values).
  - The robot's pose is updated based on the control values using the `update_pose` function.
- **Reward Calculation**:
  - Rewards are calculated based on the updated robot's position and its proximity to the target.
  - If the robot goes out of bounds, it receives a negative reward.
  - If the robot gets closer to the target, it receives a positive reward.
  - If the robot reaches the target, it receives a higher positive reward.
- **Policy Gradient Training**:
  - The training process involves training the policy network using the calculated rewards and actions taken.
  - The log probability of the action is used to calculate the loss.
  - The loss is used for backpropagation to update the policy network's weights using the Adam optimiser

```
# RL Alrogithm
state = torch.FloatTensor([x, y])
action = policy(state)
action_distribution = torch.distributions.Categorical(action)
action = action_distribution.sample()

# Perform robot action
omega_0, omega_1, omega_2 = actionMatrix(action)

# Update robot pose
x_prime, y_prime, theta_prime = update_pose(x, y, theta, target_pos,
omega_0, omega_1, omega_2)
```

```
previous_distance_to_target = distance_to_target
distance_to_target = distance([x_prime, y_prime], target_pos)

# Calculate reward based on distance to target
if not FIELD.collidepoint(x, y):
        reward = -10
        score -= 10
        train_policy(state, action, reward_list)
        episode_reward += reward
        break

elif distance_to_target < previous_distance_to_target:
        reward += 0.1
        score += 0.01

elif distance_to_target <= ROBOT_RADIUS:
        reward = 10
        score += 10
        train_policy(state, action, reward_list)
        episode_reward += reward
        break

else:
        reward = -10
        score -= 0.01

# Train policy with the selected action and calculated reward
train_policy(state, action, reward_list)
episode_reward += reward

# Update robot position
x, y, theta = x_prime, y_prime, theta_prime
```

The training performance can also be seen in the plot of the reward against the index



3.3 Testing

Using a similar testing code structure, it has been modified to accommodate the Neural Network structure. The testing code is given by,

```
# Testing the agent
test_rewards = []
class Policy(nn.Module):
        def __init__(self, input_size, hidden_size, hidden_size2,
output_size):
                super(Policy, self).__init__()
                self.linear1 = nn.Linear(input_size, hidden_size)
                self.linear2 = nn.Linear(hidden_size, hidden_size2)
                self.linear3 = nn.Linear(hidden_size2, output_size)

        def forward(self, state):
                x = F.relu(self.linear1(state))
                x = F.relu(self.linear2(x))
                x = F.softmax(self.linear3(x), dim=-1)
                return x

input_size = 2
hidden_size = 256
hidden_size2 = 128
actions = [0, 1, 2, 3, 4, 5, 6, 7]
output_size = len(actions)
```

```
for episode in range(50):
        score = 0
        robot_pose = [random.randint(FIELD.left, FIELD.right),
random.randint(FIELD.top, FIELD.bottom), random.randint(0, 359)]
        target_pos = [random.randint(FIELD.left, FIELD.right),
random.randint(FIELD.top, FIELD.bottom)]
        target_vel = [random.uniform(0.2, 0.6), random.uniform(-0.6, 0.6)]

        step = 0
        while step < 1000:
                x, y, theta = robot_pose
                state = torch.FloatTensor([x, y])
                policy_network = Policy(input_size, hidden_size,
hidden_size2, output_size) # Initialize policy network
                policy_network.load_state_dict(torch.load("policy.pth"))
                policy_network.eval() # Set the network to evaluation mode
                action_probs = policy_network(state)
                action_distribution =
torch.distributions.Categorical(action_probs)
                action = action_distribution.sample()

                # Perform the action in the environment
                omega_0, omega_1, omega_2 = actionMatrix(action.item())
                robot_pose_prime = update_pose([x, y, theta], omega_0,
omega_1, omega_2, 0.1)
                target_pos_prime, target_vel_prime =
update_target_pose(target_pos, target_vel)
                reward = compute_reward(robot_pose_prime,
target_pos_prime)
                robot_pose = robot_pose_prime
                target_pos = target_pos_prime
                target_vel = target_vel_prime
                score += reward
                step += 1
        test_rewards.append(score)
```

Using this function, the neural network is initialised using the same structure used in training. However, instead of training, it is used for evaluation by using the `.eval()` method to set the neural network to evaluation mode. This is done after loading the state dictionary which was exported during the training loop. Finally, the reward is computed by calling the `compute_reward` function which is defined by,

```
def compute_reward(robot_pose, target_pos):
[x,y,theta]= robot_pose
current_distance_to_target = distance([x, y], target_pos)

if not FIELD.collidepoint(x, y):
        # Penalize the robot for going out of bounds
```

```
        return -1

elif current_distance_to_target < TARGET_RADIUS:
        # Reward the robot for reaching the target
        return 10

else:
        return -0.01
```

3.4 Model Performance

The performance was outputted using the code above

```
Test rewards: [ -9.999999999999831, -9.999999999999831,
-471.3399999999997, 1761.7700000000073, -9.999999999999831,
-9.999999999999831, -229.7799999999999, -9.999999999999831,
-9.999999999999831, -447.57999999999583, -9.999999999999831, -649.54,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -566.379999999997,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831, -9.999999999999831,
-9.999999999999831, -9.999999999999831]
Average test reward: -20.85699999999955
Standard deviation of test reward: 293.40959176039274
```

Looking at the average test reward, we can see a relatively low performing model. Similarly, there is a considerable amount of standard deviation as well.

While the performance is not the best, it can be considered acceptable which can be improved with more iterations and fine tuning the hyperparameters of the model.

Q4. Using Neural Network to Predict "Steps-to-Target"

4.1 Data Collection

The data used was collected from the problem in question 1. The robot was tasked to get to a target given the robot's position and orientation along with the target's position.

The data was collected and exported to a CSV file which will be the basis of this question's solution.

The dataset contains the `robot_position`, `target_position`, `episode`, and `steps` iterated over 5000 times providing us over 5000 data points to work with.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd

data = pd.read_csv('../Q4/data.csv')

data['robot_pos'] = data['robot_pos'].apply(lambda x: x[1:-1].split(','))
data['target_pos'] = data['target_pos'].apply(lambda x:
x[1:-1].split(','))

X_robot = np.vstack(data['robot_pos'].apply(lambda x: [float(i) for i in
x]))
X_target = np.vstack(data['target_pos'].apply(lambda x: [float(i) for i in
x]))

X = np.hstack((X_robot, X_target))
y = data['step'].to_numpy()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

The data was imported from CSV as a Pandas dataframe and then converted to a NumPy array after separating the X and Y variables. Once, the features and target variable was separated, the dataset was split in 80/20 ratio and normalised using the Standard Scaler.

4.2 Neural Network Building

The neural network was built using 2 hidden layers of varying sizes (128 and 64).

- **Input Layer:** The input takes the feature variables (X)
- **Hidden Layers:** There are two hidden layers of size 128 and 64 respectively. The activation function is ReLU.
- **Output Layer:** The output layer has a single output as this is a regression task.

Mean Squared Error (MSE) Loss Function and Adam Optimiser was used. The learning rate was set to 0.01.

```
## Creating a neural network
import torch
import torch.nn as nn
import torch.optim as optim

class Regression(nn.Module):
        def __init__(self, input_size, output_size):
                super(Regression, self).__init__()
                self.linear1 = nn.Linear(input_size, 128)
                self.linear2 = nn.Linear(128, 64)
                self.linear3 = nn.Linear(64, output_size)

        def forward(self, x):
                x = torch.relu(self.linear1(x))
                x = torch.relu(self.linear2(x))
                x = self.linear3(x)

                return x
```

The model was trained over 10000 epochs where the loss was computed and the weights were back propagated.

```
input_size = X_train.shape[1]
model = Regression(input_size, 1)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

## Training the model
epochs = 10000
for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        outputs = model(torch.tensor(X_train, dtype=torch.float32))
        loss = criterion(outputs, torch.tensor(y_train,
dtype=torch.float32).unsqueeze(1))
        loss.backward()
        optimizer.step()
        print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}')
```

4.3 Performance Evaluation

The performance of the model was evaluated using the test set using Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

The MAE was 227.85, which suggests that the model's average prediction is offset by about 227.85 units from the target. This denotes a mediocre performance from the model.

Similarly, the RMSE was 286.1, which suggest that the square of the difference in predicted and target values are off by 286.1.

Looking at the values, the model has mediocre performance when it comes to predicting the steps needed to reach the target for the simulation in Question 1. It looks like there's room from improvement but it's not far off the mark