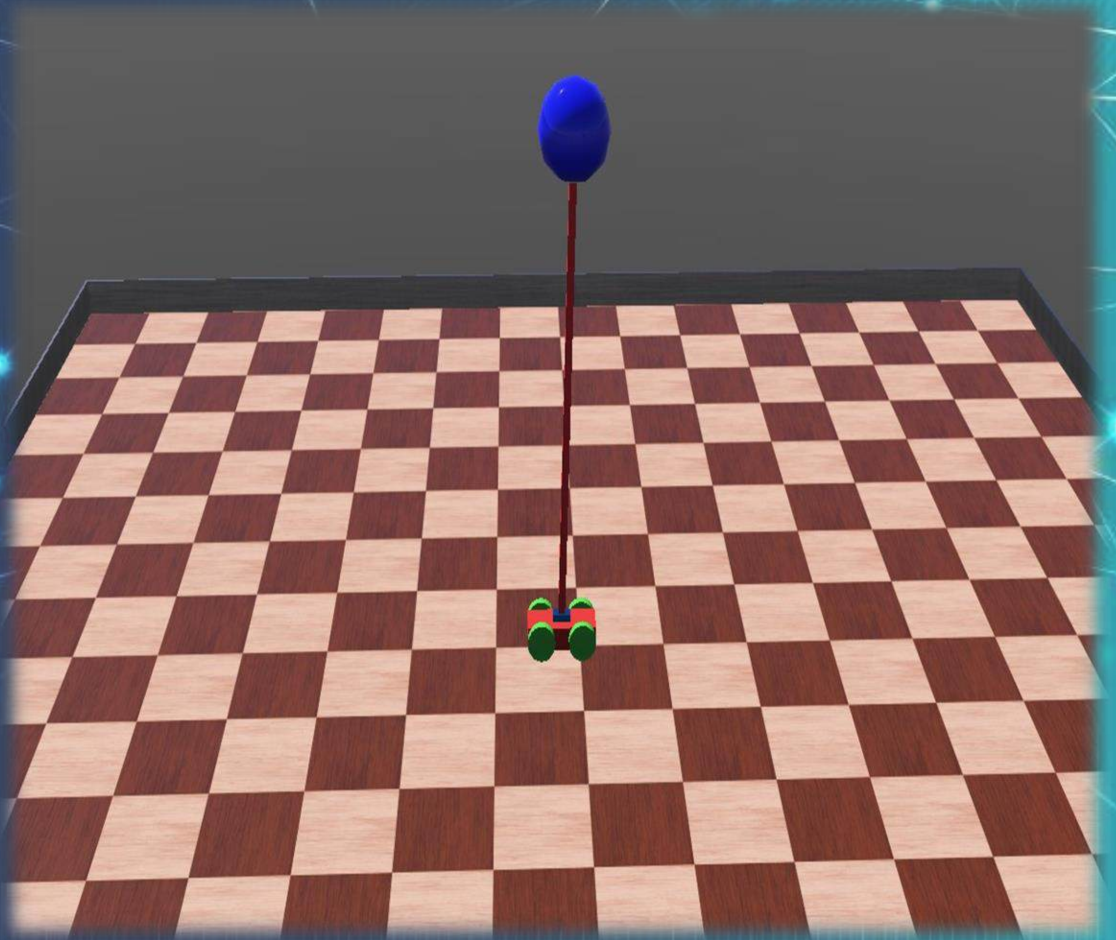
Cart-Pole Control System Optimization using Reinforcement Learning

Mars Task-2 (AI/ML)



Created by

Kris Keshav

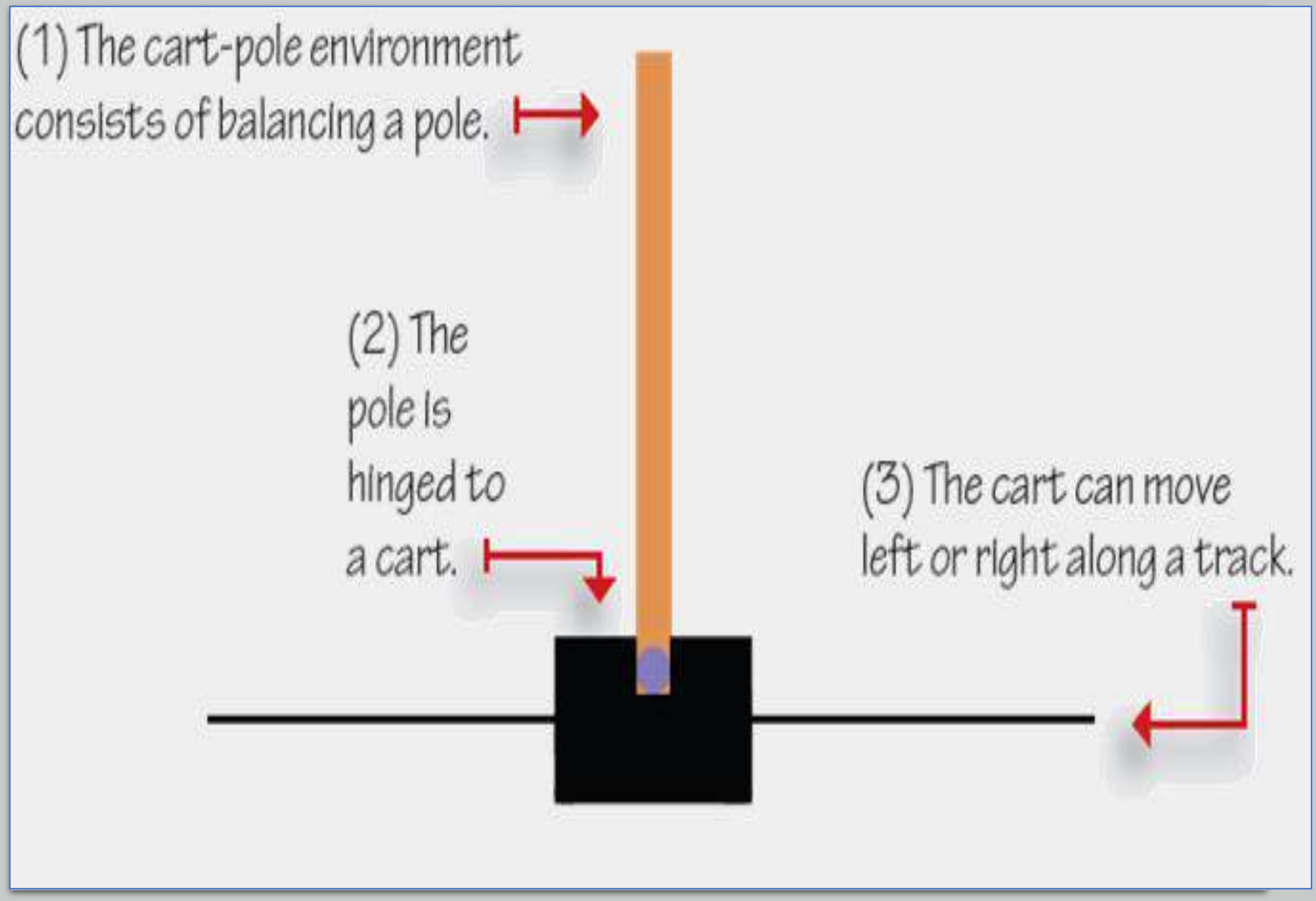
Arpit Pandey

Jheel Maheshwari

Mentor-

Vishal Sir

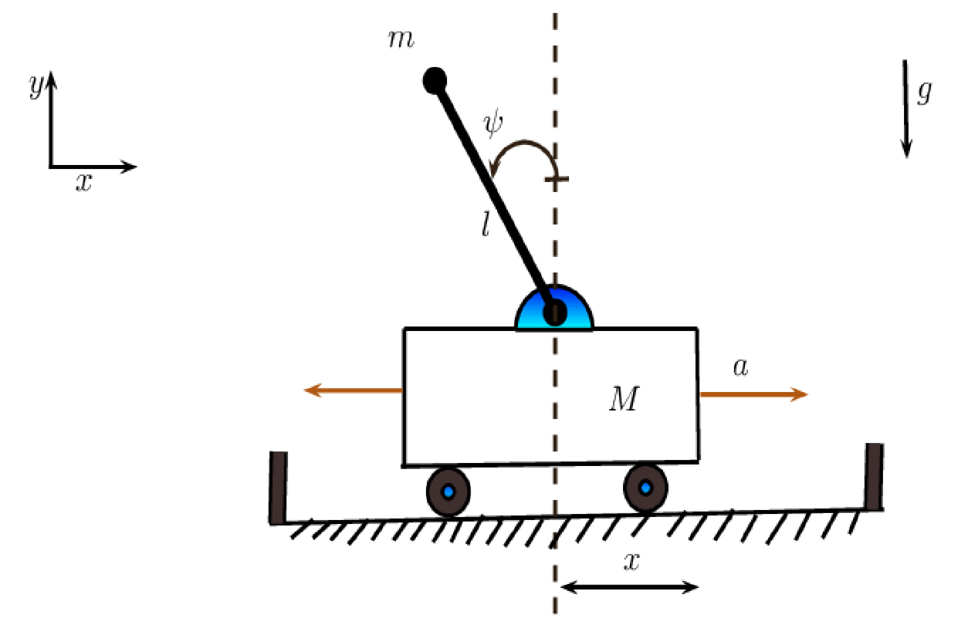
What is a Cart-Pole Environment?



Task Overview

Objective

Train a reinforcement learning agent to balance a cart-pole system using Proximal Policy Optimization (PPO) and neural networks.



Significance

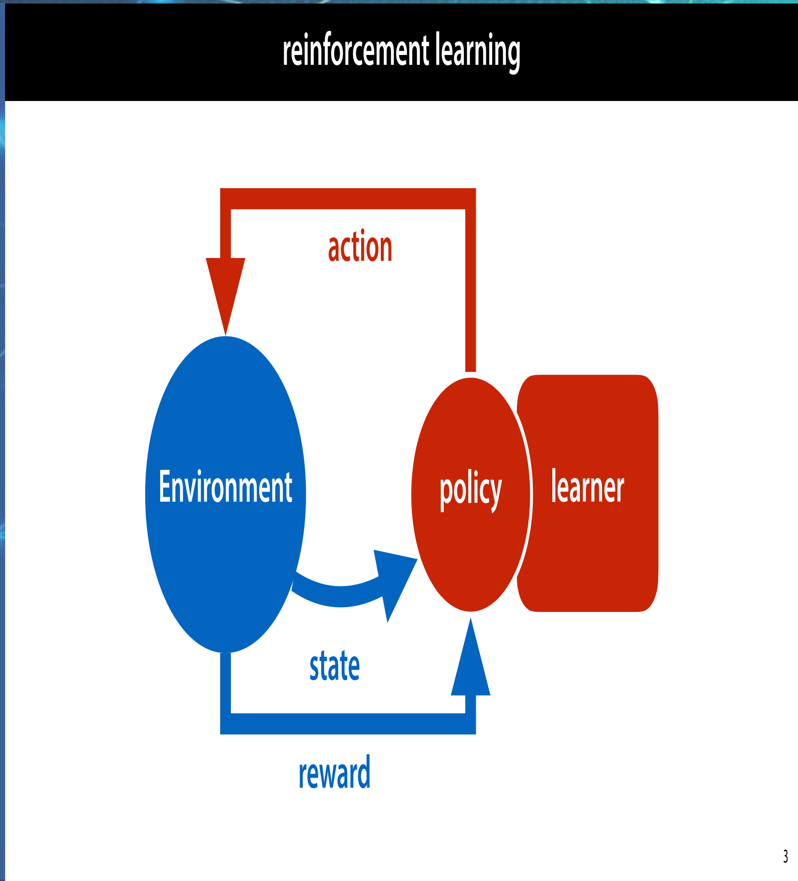
Contribute to autonomous vehicle technology and robotic control systems.

Scope

Implement a reinforcement learning algorithm for real-time control.

Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. The agent explores actions, receives feedback in the form of rewards or penalties, and adjusts its behavior based on past experiences. RL algorithms, such as Q-learning, enable agents to learn optimal strategies for complex tasks like game playing, robotics control, and resource management. RL balances exploration of new actions with exploitation of known strategies, making it suitable for dynamic decision-making in diverse real-world applications.



Proximal Policy Optimization(PPO)

PPO (Proximal Policy Optimization) is a reinforcement learning algorithm, introduced by OpenAI, that optimizes policies for continuous and discrete action spaces. It maximizes cumulative rewards by enforcing a "proximal policy update" constraint, preventing large updates for stability. PPO iteratively collects data, computes advantages, and optimizes the policy using stochastic gradient ascent. Its simplicity, efficiency, and stability make it popular for training agents in diverse domains, including robotics and games.

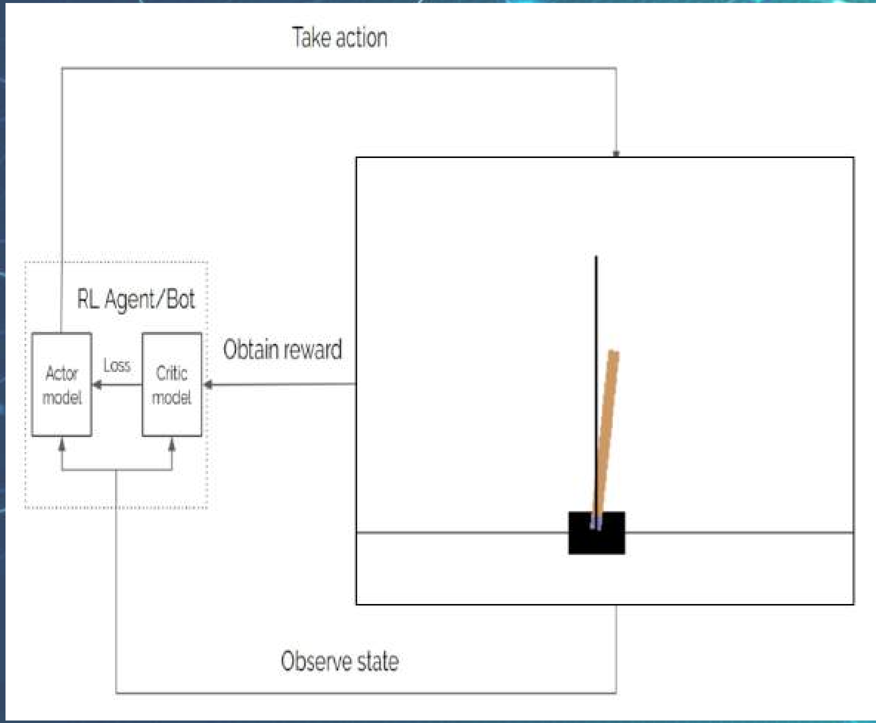
Application in our Task

Reinforcement learning powers our Cart Pole Control System optimization. It allows our agent to learn and adapt control policies by interacting with the environment, crucial for achieving stability and balance.

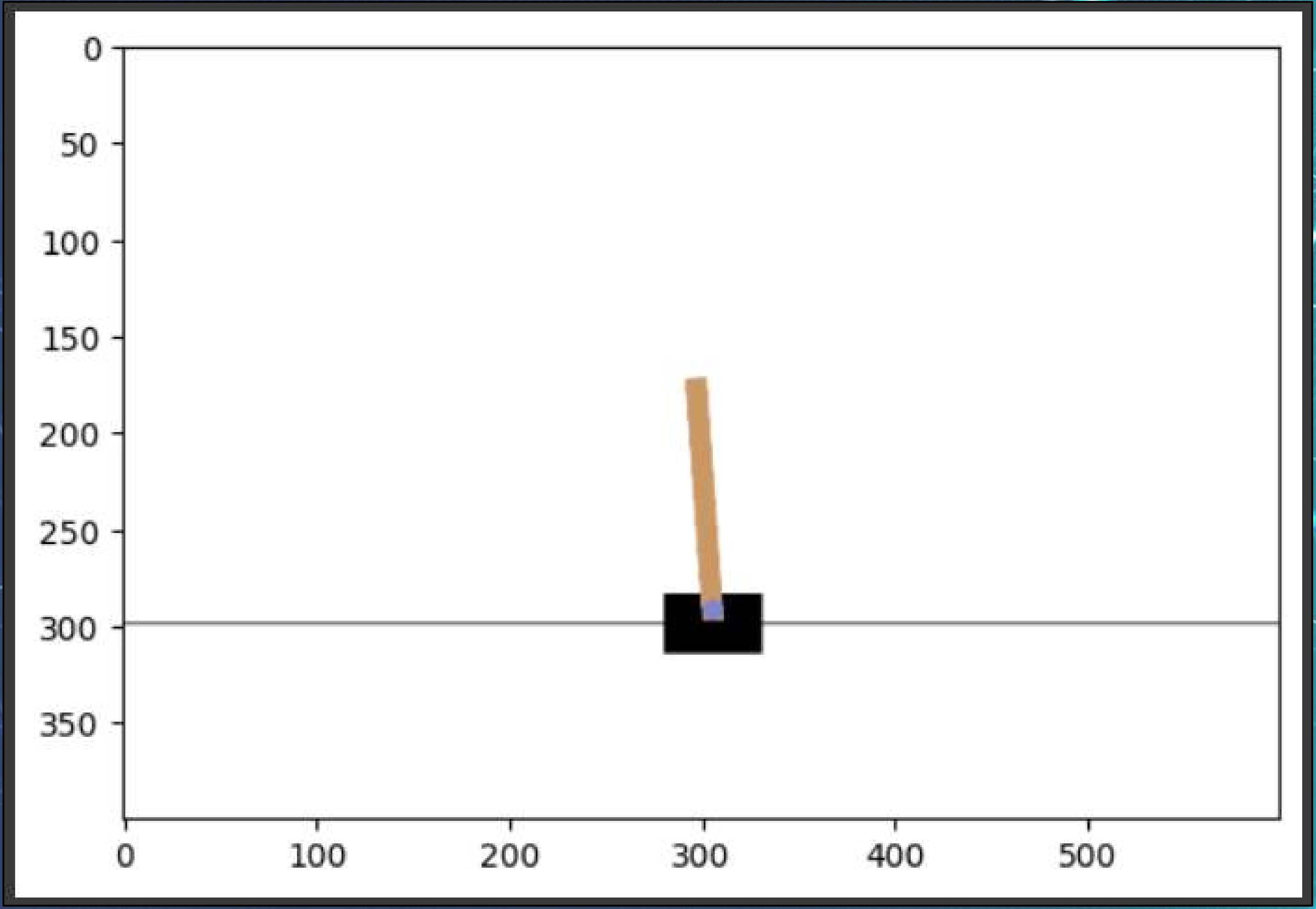
RL and PPO improve cart pole balancing by optimizing actions through policy iteration. Feedback (rewards/penalties) guides learning, while PPO's continuous action handling ensures smooth adjustments for stability.

RL balances exploration and exploitation, discovering effective strategies. PPO's stability and efficiency streamline training, requiring fewer samples.

RL with PPO offers a robust framework for efficient cart pole balancing.



Our Very First Try on Google Colab



Code and Working

- ***Setting up a virtual display and installing necessary dependencies***

```
[ ]  !sudo apt-get install xvfb
     !pip install xvfbwrapper
     !apt-get install x11-utils > /dev/null 2>&1
     !pip install pyglet > /dev/null 2>&1
     !apt-get install -y xvfb python-opengl > /dev/null 2>&1
     !pip install gym pyvirtualdisplay > /dev/null 2>&1
```

- ***Importing required libraries, creating and setting up the environment***

```
[ ]  import numpy as np
     import time
     import gym
     import matplotlib.pyplot as plt
     from IPython import display as ipythondisplay
     from pyvirtualdisplay import Display

     # Starting virtual display
     display = Display(visible=0, size=(400, 300))
     display.start()

     # Creating environment
     env = gym.make("CartPole-v1")

     # Defining parameters
     episodeNumber = 10000
     timeSteps = 100
```

[ ]

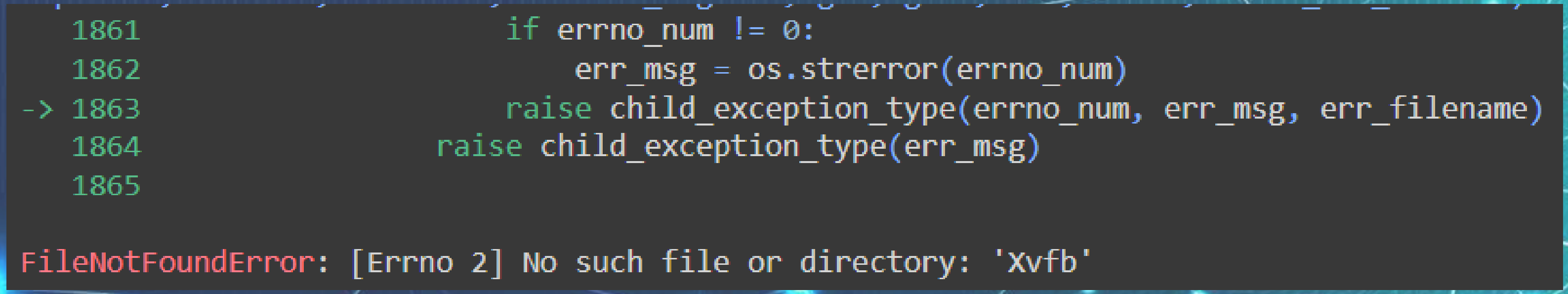
```
for episodeIndex in range(episodeNumber):
    initial_state = env.reset()
    print("Episode:", episodeIndex)

    episode_images = []  # Collect images for this episode

    for timeIndex in range(timeSteps):
        # Take a random action
        random_action = env.action_space.sample()
        observation, reward, done, info = env.step(random_action)

        # Render the environment and store the image
        screen = env.render(mode='rgb_array')
        episode_images.append(screen)

        if done:
            print("Episode terminated")
            break

    # Display the collected images for this episode
    for image in episode_images:
        plt.imshow(image)
        ipythondisplay.clear_output(wait=True)
        ipythondisplay.display(plt.gcf())
        time.sleep(0)

ipythondisplay.clear_output(wait=True)
env.close()
```

Running the cart-pole simulation

Each time, it randomly tries different actions to see how well it can balance the pole on the cart and then shows images of each attempt.

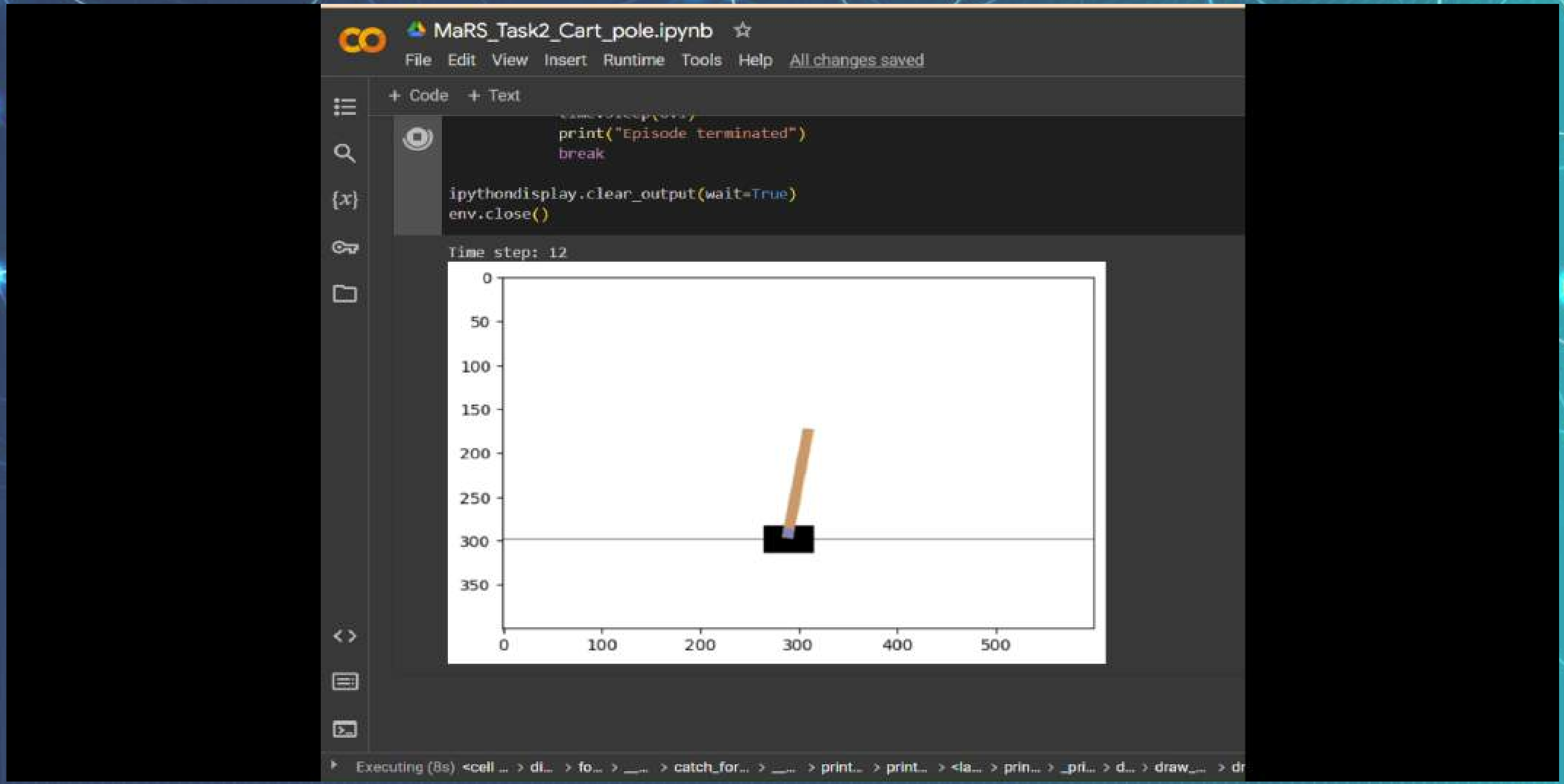
Challenges encountered

- At first we were not able to simulate the cart-pole even after successfully running the code
- We were getting errors like “ModuleNotFoundError”, ”FileNotFoundError”, etc.
- In first try the output image was flickering and was not still

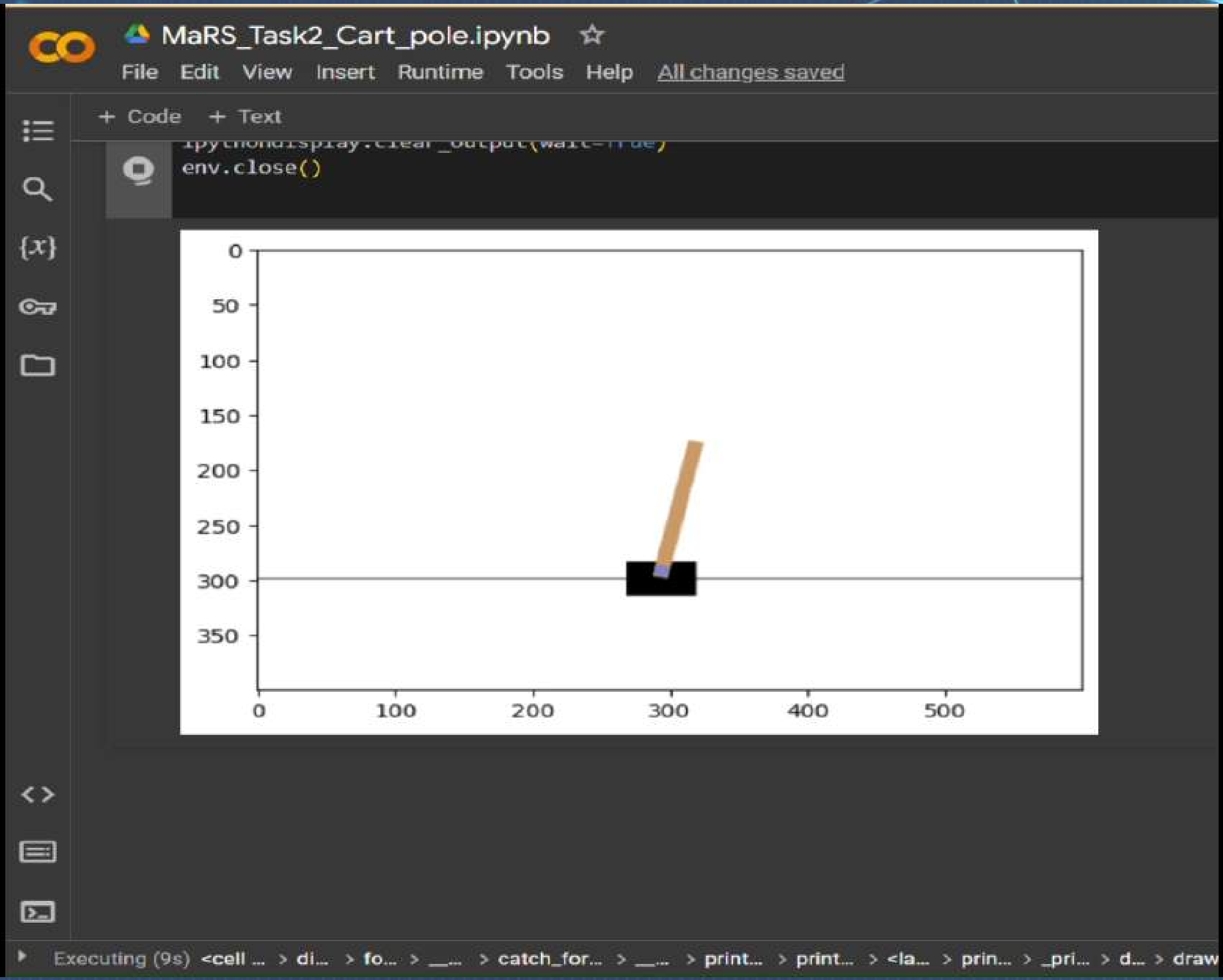


How we resolved

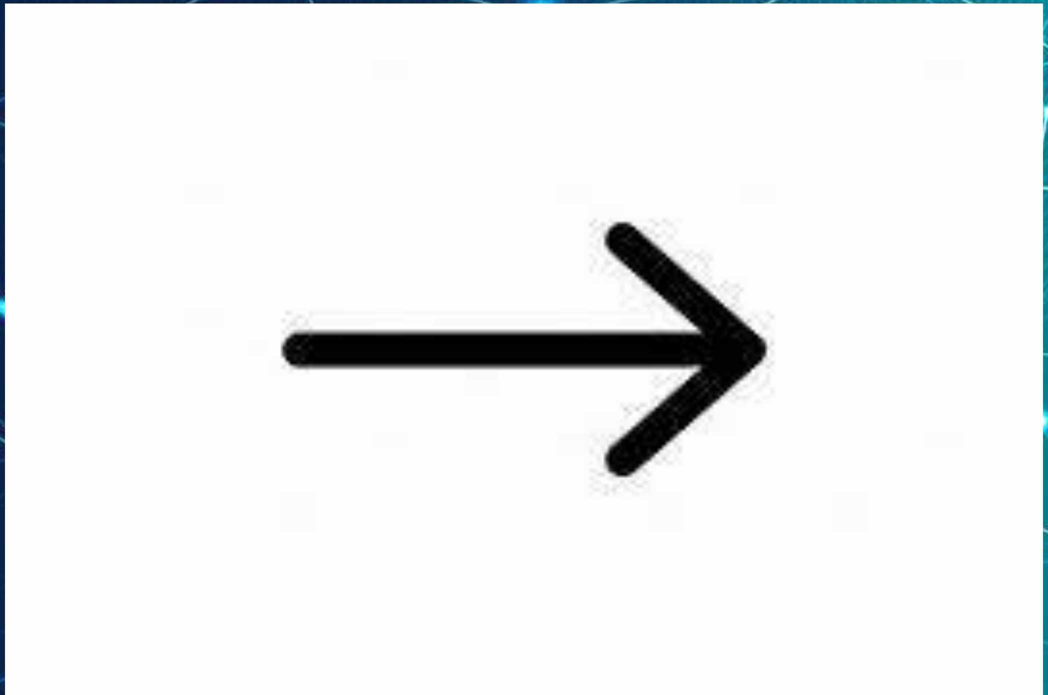
- Imported some files and modules to render Gym’s environments like Cart-Pole, Frozen-Lake, etc.
- Adjusted sleep times of each step and episode properly to avoid flickering of the output



FINAL OUTPUT



After trying on Google Colab, we switched to Webots for better simulation capabilities and a more comprehensive development.



System Design and Implementation: Switched To Webots

Software Used: Webots

Webots is a professional robot simulation software featuring a 3D physics-based environment, pre-built robot models, sensor and actuator simulation, programming interfaces in multiple languages, and integration with the Robot Operating System (ROS) for robotic development and testing.

Creating a new world in webots

Initially, we incorporated a rectangular arena into the virtual environment by following the necessary steps in Webots. Subsequently, we attempted to add a cart-pole robot to the arena but encountered difficulties in doing so. As a solution, we familiarized ourselves with the process of adding a robot to the environment using a text editor. After several attempts, we successfully added the robot to the environment using Notepad.

File Edit View

```
#VRML_SIM R2023b utf8

EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/backgrounds/
protos/TexturedBackground.proto"
EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/backgrounds/
protos/TexturedBackgroundLight.proto"
EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/floors/proto
s/RectangleArena.proto"

WorldInfo {
}
Viewpoint {
  orientation -0.3903399227006217 0.37880316161707106 0.8391322360003718
1.7747628103279394
  position 0.24026206039697126 -4.325655692517675 5.415307256893812
}
TexturedBackground {
}
TexturedBackgroundLight {
}
RectangleArena {
  floorSize 6 7
}
DEF ROBOT Robot {
  translation -0.08492170450638159 7.953734626431077e-08 0.03947485932600569
  rotation 0.9999999014943304 0.00031387860339814305 -0.00031383363713345224
1.5707964249031956
  children [
    DEF HINGE_COVER Solid {
      translation 0 0.03 -3.469446951953614e-18
      rotation 0 1 0 -1.5707953071795862
      children [
        Shape {
          appearance PBRAppearance {
            baseColor 0 0.6509803921568628 1
          }
          geometry Box {
            size 0.030000000000000002 0.019999999999999997 0.05
```

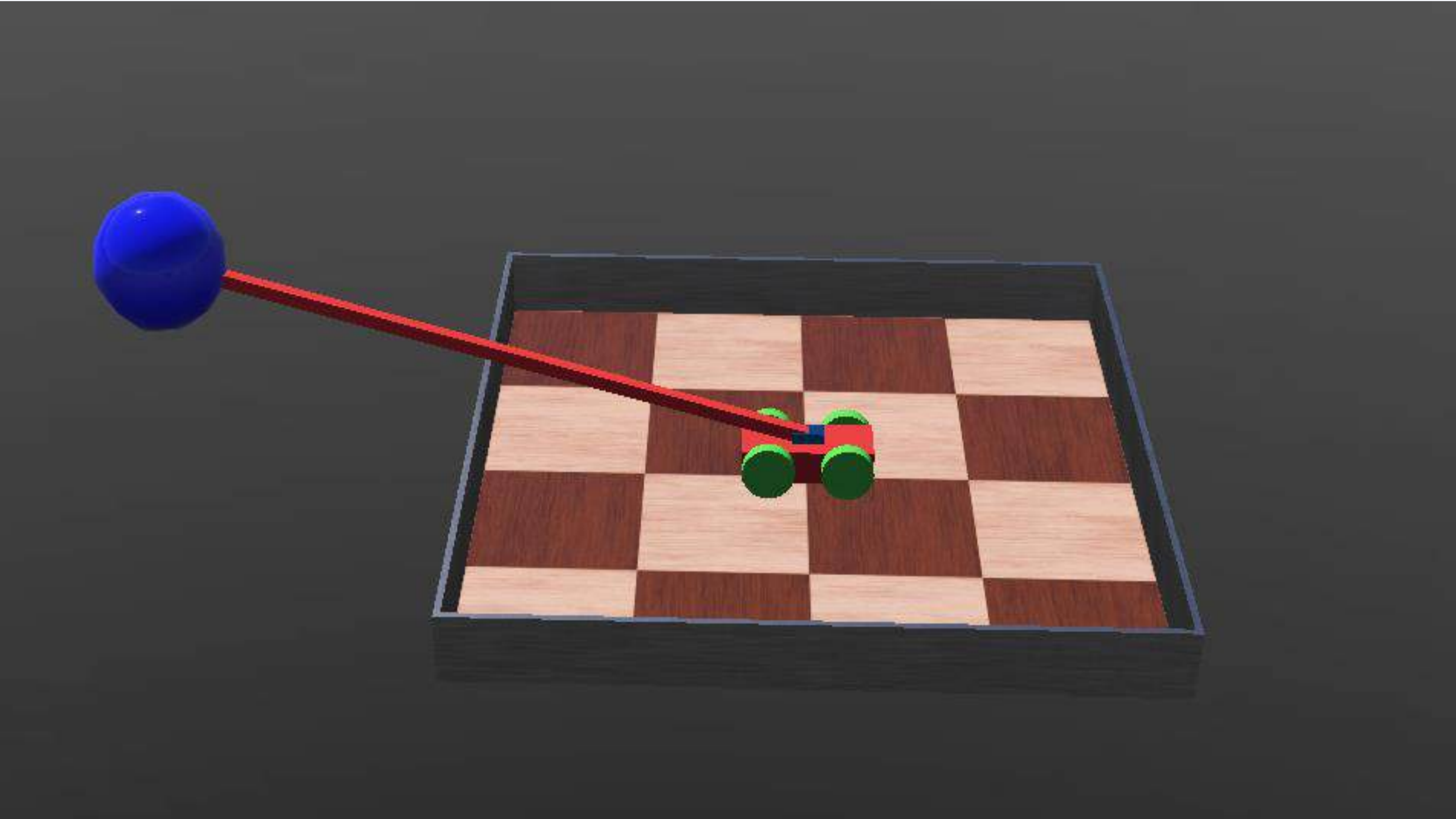
Ln 1, Col 1 | 6,983 characters | 80% | Unix (LF) | UTF-8

File Edit View

```
          }
          geometry Box {
            size 0.030000000000000002 0.019999999999999997 0.05
          }
        }
      ]
      name "hingeCover"
    }
    DEF BODY Shape {
      appearance PBRAppearance {
        baseColor 0.917647 0.145098 0.145098
        roughness 1
        metalness 0
      }
      geometry Box {
        size 0.2 0.05 0.08
      }
    }
    DEF WHEEL1 HingeJoint {
      jointParameters HingeJointParameters {
        position 1.3759088008343242e-08
        axis 0 0 1
        anchor 0.06 0 0.05
      }
      device [
        RotationalMotor {
          name "wheel1"
        }
      ]
      endPoint Solid {
        translation 0.060001004644634315 1.3392769197992635e-05 0.050000010543824816
        rotation 1.7670998430694134e-08 -1.7671157165212377e-08 0.9999999999999998
1.5708000567710334
        children [
          DEF WHEEL Shape {
            appearance PBRAppearance {
              baseColor 0.305882 0.898039 0.25098
              roughness 1
              metalness 0
            }
            geometry Cylinder {
              height 0.02
```

Ln 1, Col 1 | 6,017 characters | 80% | Unix (LF) | UTF-8

The code snippet defines a robot model with a body, wheels, a pole, and a hinge cover. The robot has rotational motors for its wheels and a position sensor for the pole. The model includes physics properties such as mass and density for different components.



```python
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical
from torch import from_numpy, no_grad, save, load, tensor, clamp
from torch import float as torch_float
from torch import long as torch_long
from torch import min as torch_min
from torch.utils.data.sampler import BatchSampler, SubsetRandomSampler
import numpy as np
from torch import manual_seed
from collections import namedtuple

Transition = namedtuple('Transition', ['state', 'action', 'a_log_prob', 'reward', 'next_state'])


class PPOAgent:
    """
    PPOAgent implements the PPO RL algorithm (https://arxiv.org/abs/1707.06347).
    It works with a set of discrete actions.
    It uses the Actor and Critic neural network classes defined below.
    """

    def __init__(self, number_of_inputs, number_of_actor_outputs, clip_param=0.2, max_grad_norm=0.5, ppo_update_iters=5,
                 batch_size=8, gamma=0.99, use_cuda=False, actor_lr=0.001, critic_lr=0.003, seed=None):
        super().__init__()
        if seed is not None:
            manual_seed(seed)

        # Hyper-parameters
        self.clip_param = clip_param
        self.max_grad_norm = max_grad_norm
        self.ppo_update_iters = ppo_update_iters
        self.batch_size = batch_size
        self.gamma = gamma
        self.use_cuda = use_cuda

        # models
        self.actor_net = Actor(number_of_inputs, number_of_actor_outputs)
        self.critic_net = Critic(number_of_inputs)
        if self.use_cuda:
            self.actor_net.cuda()
            self.critic_net.cuda()

        # Create the optimizers
        self.actor_optimizer = optim.Adam(self.actor_net.parameters(), actor_lr)
        self.critic_net_optimizer = optim.Adam(self.critic_net.parameters(), critic_lr)

        # Training stats
        self.buffer = []

    def work(self, agent_input, type_="simple"):
        """
        type_ == "simple"
            Implementation for a simple forward pass.
        type_ == "selectAction"
            Implementation for the forward pass, that returns a selected action according to the probability
            distribution and its probability.
        type_ == "selectActionMax"
            Implementation for the forward pass, that returns the max selected action.
        """
        agent_input = from_numpy(np.array(agent_input)).float().unsqueeze(0)  # Add batch dimension with unsqueeze
        if self.use_cuda:
            agent_input = agent_input.cuda()
        with no_grad():
            action_prob = self.actor_net(agent_input)

        if type_ == "simple":
            output = [action_prob[0][i].data.tolist() for i in range(len(action_prob[0]))]
            return output
        elif type_ == "selectAction":
            c = Categorical(action_prob)
            action = c.sample()
            return action.item(), action_prob[:, action.item()].item()
        elif type_ == "selectActionMax":
            return np.argmax(action_prob).item(), 1.0
        else:
            raise Exception("Wrong type in agent.work(), returning input")
```

This code creates a smart agent (PPOAgent) that learns to balance a pole on a cart. It uses two neural networks: one (Actor) decides what action to take, and the other (Critic) tells how good the situation is. The agent learns by trying actions, seeing what happens, and updating its networks based on the results. Over time, it gets better at balancing the pole.

```python
from deepbots.supervisor.controllers.robot_supervisor_env import RobotSupervisorEnv
from utilities import normalize_to_range
from PPO_agent import PPOAgent, Transition

from gym.spaces import Box, Discrete
import numpy as np


class CartpoleRobot(RobotSupervisorEnv):
    def __init__(self):
        super().__init__()
        # Define agent's observation space using Gym's Box, setting the lowest and highest possible values
        self.observation_space = Box(low=np.array([-0.4, -np.inf, -1.3, -np.inf]),
                                     high=np.array([0.4, np.inf, 1.3, np.inf]),
                                     dtype=np.float64)
        # Define agent's action space using Gym's Discrete
        self.action_space = Discrete(2)

        self.robot = self.getSelf()  # Grab the robot reference from the supervisor to access various robot methods
        self.position_sensor = self.getDevice("polePosSensor")
        self.position_sensor.enable(self.timestep)

        self.pole_endpoint = self.getFromDef("POLE_ENDPOINT")
        self.wheels = []
        for wheel_name in ['wheel1', 'wheel2', 'wheel3', 'wheel4']:
            wheel = self.getDevice(wheel_name)  # Get the wheel handle
            wheel.setPosition(float('inf'))  # Set starting position
            wheel.setVelocity(0.0)  # Zero out starting velocity
            self.wheels.append(wheel)
        self.steps_per_episode = 200  # Max number of steps per episode
        self.episode_score = 0  # Score accumulated during an episode
        self.episode_score_list = []  # A list to save all the episode scores, used to check if task is solved

    def get_observations(self):
        # Position on x-axis
        cart_position = normalize_to_range(self.robot.getPosition()[0], -0.4, 0.4, -1.0, 1.0)
        # Linear velocity on x-axis
        cart_velocity = normalize_to_range(self.robot.getVelocity()[0], -0.2, 0.2, -1.0, 1.0, clip=True)
        # Pole angle off vertical
        pole_angle = normalize_to_range(self.position_sensor.getValue(), -0.23, 0.23, -1.0, 1.0, clip=True)
        # Angular velocity y of endpoint
```

```python
        # Angular velocity y of endpoint
        endpoint_velocity = normalize_to_range(self.pole_endpoint.getVelocity()[4], -1.5, 1.5, -1.0, 1.0, clip=True)

        return [cart_position, cart_velocity, pole_angle, endpoint_velocity]

    def get_default_observation(self):
        # This method just returns a zero vector as a default observation
        return [0.0 for _ in range(self.observation_space.shape[0])]

    def get_reward(self, action=None):
        # Reward is +1 for every step the episode hasn't ended
        return 1

    def is_done(self):
        if self.episode_score > 195.0:
            return True

        pole_angle = round(self.position_sensor.getValue(), 2)
        if abs(pole_angle) > 0.261799388:  # more than 15 degrees off vertical (defined in radians)
            return True

        cart_position = round(self.robot.getPosition()[0], 2)  # Position on x-axis
        if abs(cart_position) > 0.39:
            return True

        return False

    def solved(self):
        if len(self.episode_score_list) > 100:  # Over 100 trials thus far
            if np.mean(self.episode_score_list[-100:]) > 195.0:  # Last 100 episodes' scores average value
                return True
        return False

    def get_info(self):
        return None

    def render(self, mode='human'):
        pass

    def apply_action(self, action):
        action = int(action[0])
```

1. Imports: The code starts with importing necessary modules and classes such as RobotSupervisorEnv from deepbots.supervisor.controllers.robot_supervisor_env, normalize_to_range from utilities, PPOAgent from PPO_agent, and various components from gym.

2. Class Definition: The CartpoleRobot class is defined, inheriting from RobotSupervisorEnv. It initializes the environment with observation and action spaces, sets up sensors and actuators for the robot, defines methods for getting observations, rewards, checking if an episode is done, checking if the task is solved, and applying actions to the robot.

3. Environment Setup: The environment is set up with observation and action spaces suitable for a cartpole problem. Sensors and actuators for the robot are initialized.

4. Observations: The get_observations method computes the observations based on the robot's state, such as position, velocity, pole angle, and endpoint velocity.

5. Reward and Done Conditions: The get_reward method defines the reward function, and the is_done method checks if an episode is done based on certain conditions such as the angle of the pole and the position of the cart.

6. Training Loop: The code then sets up a training loop using a Proximal Policy Optimization (PPO) agent. It iterates through episodes, taking actions based on the agent's policy, storing transitions, and training the agent.

```python
import numpy as np


def normalize_to_range(value, min_val, max_val, new_min, new_max, clip=False):
    """
    Normalize value to a specified new range by supplying the current range.

    :param value: value to be normalized
    :param min_val: value's min value, value ∈ [min_val, max_val]
    :param max_val: value's max value, value ∈ [min_val, max_val]
    :param new_min: normalized range min value
    :param new_max: normalized range max value
    :param clip: whether to clip normalized value to new range or not
    :return: normalized value ∈ [new_min, new_max]
    """
    value = float(value)
    min_val = float(min_val)
    max_val = float(max_val)
    new_min = float(new_min)
    new_max = float(new_max)

    if clip:
        return np.clip((new_max - new_min) / (max_val - min_val) * (value - max_val) + new_max, new_min, new_max)
    else:
        return (new_max - new_min) / (max_val - min_val) * (value - max_val) + new_max
```

This code provides a normalization function for fitting the internal parameters like cart position, velocity or pole angle and angular velocity.

These components are crucial for representing the state of the cart-pole system, and normalizing them can help the reinforcement learning algorithm learn more effectively by providing consistent input ranges.

Challenges Encountered

Console - All

```
  File "C:\Users\krish\AppData\Roaming\Python\Python312\site-packages\gym\__init__.py", line 13, in <module>
    from gym.envs import make, spec, register
  File "C:\Users\krish\AppData\Roaming\Python\Python312\site-packages\gym\envs\__init__.py", line 10, in <module>
    _load_env_plugins()
  File "C:\Users\krish\AppData\Roaming\Python\Python312\site-packages\gym\envs\registration.py", line 770, in load_env_plugins
    for plugin in metadata.entry_points().get(entry_point, []):
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'EntryPoints' object has no attribute 'get'
WARNING: 'my_controller_cart_pole' controller exited with status: 1.
```

This error occurs because the importlib-metadata package removed the deprecated entry point interfaces starting version 5.0. To solve this error, we changed importlib-metadata package to version 4.0.0.

```
Traceback (most recent call last):
  File "C:\Users\krish\OneDrive\Desktop\my_project\controllers\my_controller_cart_pole\my_controller_cart_pole.py", line 3, in <module>
    from utilities import normalize_to_range
ModuleNotFoundError: No module named 'utilities'
WARNING: 'my_controller_cart_pole' controller exited with status: 1.
```

- **Version compatibility issues**
- **Dependency conflict**

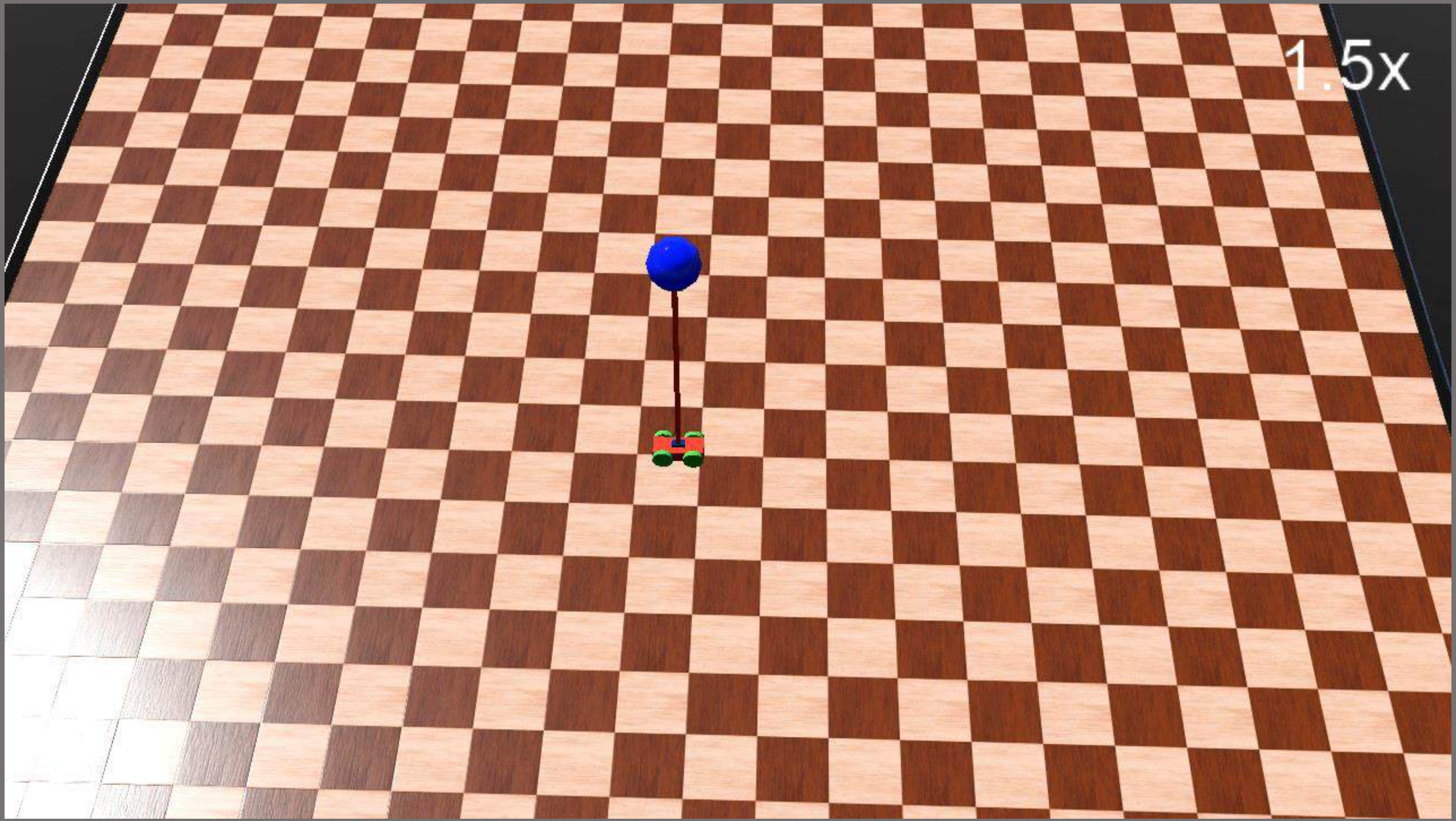
We were using deepbots version- 1.0.0.dev0 which requires gym 0.21 but we add gym 0.26.2 which is incompatible.

How we resolved ?

Uninstalling the latest version of gym (0.26.2) and switching to gym older version(0.21)

```
C:\Users\krish>pip install gym==0.21
Defaulting to user installation because normal site-packages is not writeable
Collecting gym==0.21
  Using cached gym-0.21.0.tar.gz (1.5 MB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... error
  error: subprocess-exited-with-error

  × Getting requirements to build wheel did not run successfully.
  │ exit code: 1
  ╰─> [1 lines of output]
      error in gym setup command: 'extras_require' must be a dictionary whose values are strings or lists of strings con
taining valid project/version requirement specifiers.
      [end of output]

  note: This error originates from a subprocess, and is likely not a problem with pip.
error: subprocess-exited-with-error

× Getting requirements to build wheel did not run successfully.
│ exit code: 1
╰─> See above for output.

note: This error originates from a subprocess, and is likely not a problem with pip.
```

Our simulation in Webots at beginning phase of training

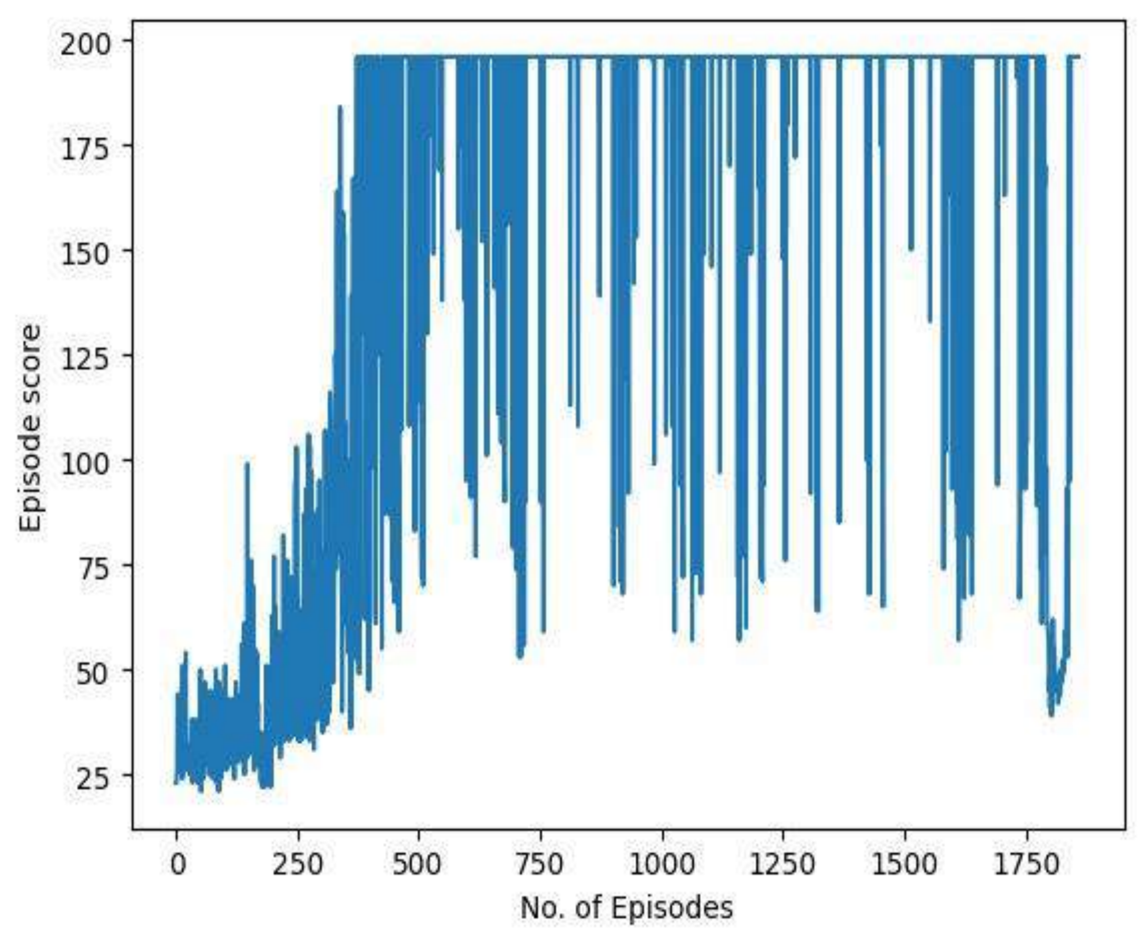


Our Simulation in Webots

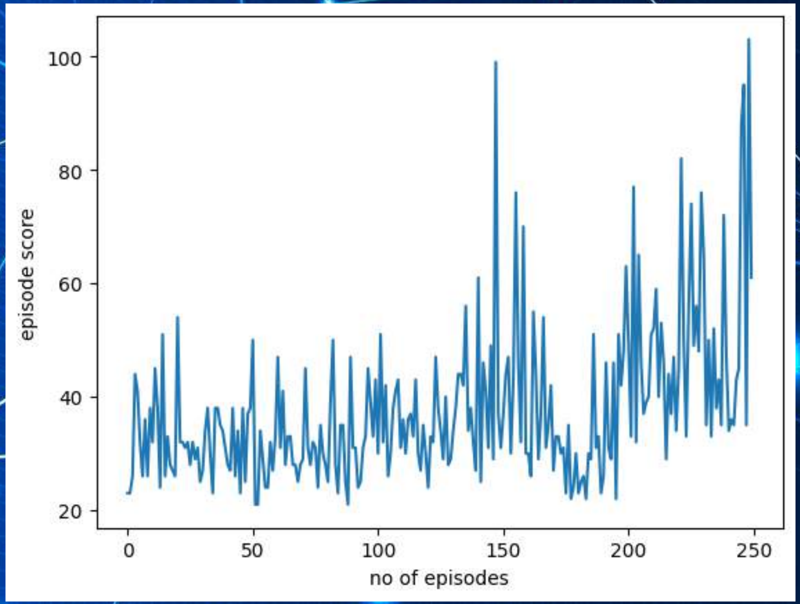
Results and Performance

Learning Curve

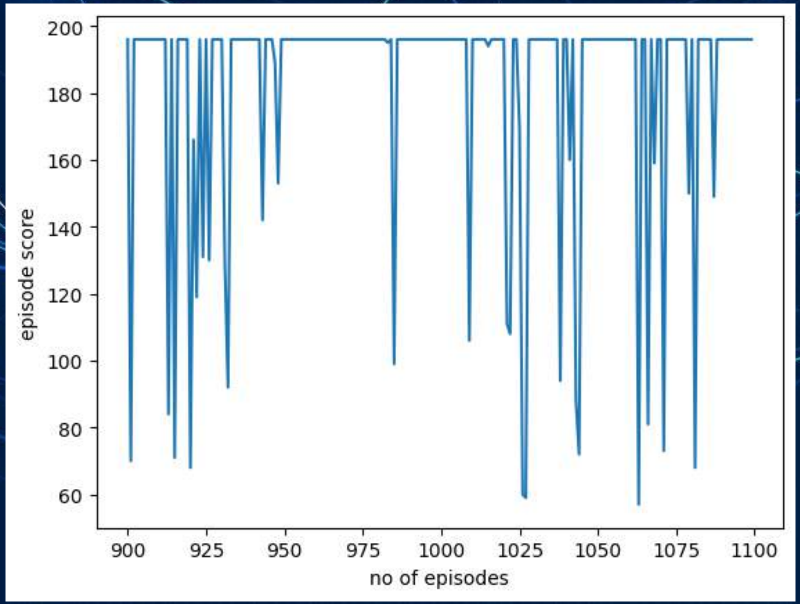
- *The videos and graph clearly depict the Cart-Pole agent's progress. Initially, it struggled to balance, yielding lower scores. However, over subsequent episodes, its performance improved steadily. While occasional dips occurred, the agent ultimately mastered cart-pole control by the end of the observation period.*



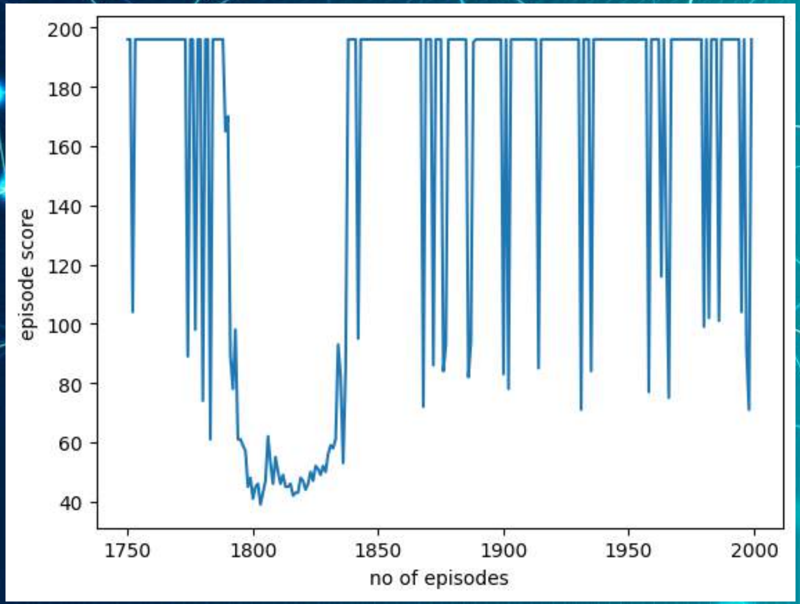
Phase Wise Scores



At the beginning of training



In the middle of training



At the end of training

Lessons Learned

Exploration-Exploitation

- Balancing exploration (trying new things) and exploitation (using what works) in reinforcement learning is important.
- Early on, encouraging exploration to discover strategies is required.
- Shifting towards exploitation as the agent learns to optimize performance.
- Adjusting exploration rates or strategies like **epsilon-greedy** for balance.

Real-World Adaptation

- Adjusting the model to handle changes in the environment.
- Techniques like **transfer learning** or continual learning are used.
- Improves the model's robustness and reliability.

Applications and Future Scopes

Enhancements

Implementing advanced algorithms like DDPG or SAC for improved stability and performance, integrating neural network architectures for better feature representation, and exploring meta-learning approaches for faster adaptation to new environments and tasks.

Applications

The cart-pole balancing system finds applications in various fields such as robotics, manufacturing, and aerospace. It can be used for stabilizing unmanned aerial vehicles (UAVs), balancing machinery on production lines, and enhancing the agility and stability of humanoid robots for tasks in industries like logistics and healthcare.


Thank You!