Twin Delayed DDPG (TD3): Previously, we have used DDPG but it has some issues like brittle hyperparameter and tuning. The drawback is that it dramatically estimates the Q-value as it leads to policy breaking because it exploits errors in the Q function. So here TD3 algorithm is used to overcome this issue.

Trick One: Clipped Double-Q Learning: It learns two Q-functions instead of one hence we call it a twin, and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Trick Two: Delayed Policy Updates: It updates the policy and target network less frequently than the Q-function

Trick Three: Target Policy Smoothing: It adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

The installation of OpenAI Box2D

We have to go to the official SWIG download website and install it. I installed it on Windows, extracted the zip file, and added it to the path.Go to This PC and then Environmental variables and set the path. At last run

swig -version
SWIG Version 4.2.1
Compiled with i686-w64-mingw32-g++ [i686-w64-mingw32]
Configured options: +pcre

It is successfully installed and added as path now, we need to install

pip install
https://download.lfd.uci.edu/pythonlibs/p3pwyuzl/Box2D-2.3.2-cp311-cp311-win_amd64.whl

To verify the installation

python -c "import Box2D"

It can also be installed using Miniconda and setting new gym environment It also works!

A program example of Bipedel walker:

```
import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# Set up the environment
env = gym.make('BipedalWalker-v3')
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

# Actor model definition using PyTorch
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()
        self.l1 = nn.Linear(state_dim, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, action_dim)
        self.max_action = max_action

    def forward(self, state):
        a = F.relu(self.l1(state))
        a = F.relu(self.l2(a))
        return self.max_action * torch.tanh(self.l3(a))

# Critic model definition using PyTorch
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.l1 = nn.Linear(state_dim + action_dim, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, 1)
```

```
    def forward(self, state, action):
        q = F.relu(self.l1(torch.cat([state, action], 1)))
        q = F.relu(self.l2(q))
        return self.l3(q)

# Replay Buffer
class ReplayBuffer:
    def __init__(self, max_size):
        self.buffer = []
        self.max_size = max_size
        self.ptr = 0

    def add(self, transition):
        if len(self.buffer) < self.max_size:
            self.buffer.append(transition)
        else:
            self.buffer[self.ptr] = transition
        self.ptr = (self.ptr + 1) % self.max_size

    def sample(self, batch_size):
        indices = np.random.randint(0, len(self.buffer),
size=batch_size)
        states, actions, rewards, next_states, dones =
zip(*[self.buffer[idx] for idx in indices])
        return np.array(states), np.array(actions), np.array(rewards),
np.array(next_states), np.array(dones)

# TD3 Agent using PyTorch
class TD3:
    def __init__(self, state_dim, action_dim, max_action):
        self.actor = Actor(state_dim, action_dim,
max_action).to(device)
        self.actor_target = Actor(state_dim, action_dim,
max_action).to(device)
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.actor_optimizer = optim.Adam(self.actor.parameters(),
lr=3e-4)

        self.critic1 = Critic(state_dim, action_dim).to(device)
        self.critic2 = Critic(state_dim, action_dim).to(device)
        self.critic1_target = Critic(state_dim, action_dim).to(device)
        self.critic2_target = Critic(state_dim, action_dim).to(device)
        self.critic1_target.load_state_dict(self.critic1.state_dict())
```

```
        self.critic2_target.load_state_dict(self.critic2.state_dict())
        self.critic_optimizer =
optim.Adam(list(self.critic1.parameters()) +
list(self.critic2.parameters()), lr=3e-4)

        self.max_action = max_action
        self.replay_buffer = ReplayBuffer(1_000_000)
        self.batch_size = 100
        self.gamma = 0.99
        self.tau = 0.005
        self.policy_noise = 0.2
        self.noise_clip = 0.5
        self.policy_freq = 2
        self.total_it = 0

    def select_action(self, state):
        state = torch.FloatTensor(np.array(state).reshape(1,
-1)).to(device)
        return self.actor(state).cpu().data.numpy().flatten()

    def train(self):
        if len(self.replay_buffer.buffer) < self.batch_size:
            return

        self.total_it += 1
        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(self.batch_size)

        state = torch.FloatTensor(states).to(device)
        action = torch.FloatTensor(actions).to(device)
        reward = torch.FloatTensor(rewards).to(device)
        next_state = torch.FloatTensor(next_states).to(device)
        done = torch.FloatTensor(dones).to(device).reshape(-1, 1)

        with torch.no_grad():
            noise = (torch.randn_like(action) *
self.policy_noise).clamp(-self.noise_clip, self.noise_clip)
            next_action = (self.actor_target(next_state) +
noise).clamp(-self.max_action, self.max_action)

            target_q1 = self.critic1_target(next_state, next_action)
            target_q2 = self.critic2_target(next_state, next_action)
            target_q = torch.min(target_q1, target_q2)
```

```
            target_q = reward.reshape(-1, 1) + ((1 - done) * self.gamma
* target_q).detach()

        current_q1 = self.critic1(state, action)
        current_q2 = self.critic2(state, action)
        critic_loss = F.mse_loss(current_q1, target_q) +
F.mse_loss(current_q2, target_q)

        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        if self.total_it % self.policy_freq == 0:
            actor_loss = -self.critic1(state, self.actor(state)).mean()

            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()

            for param, target_param in zip(self.critic1.parameters(),
self.critic1_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)

            for param, target_param in zip(self.critic2.parameters(),
self.critic2_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)

            for param, target_param in zip(self.actor.parameters(),
self.actor_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)

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

# Parameters
max_episodes = 100  # Reduced number of episodes
max_timesteps = 500  # Reduced number of timesteps per episode
exploration_noise = 0.1
log_interval = 10  # Log every 10 episodes
```

```python
# Initialize TD3 agent
td3_agent = TD3(state_dim, action_dim, max_action)

# Training loop
for episode in range(max_episodes):
    state, _ = env.reset()
    episode_reward = 0
    for t in range(max_timesteps):
        # Removed rendering for faster training
        action = td3_agent.select_action(state)
        action = action + np.random.normal(0, exploration_noise,
size=action_dim)
        action = action.clip(-max_action, max_action)

        next_state, reward, done, _, _ = env.step(action)
        td3_agent.replay_buffer.add((state, action, reward, next_state,
float(done)))

        state = next_state
        episode_reward += reward

        if done:
            break

        td3_agent.train()

    if episode % log_interval == 0:
        print(f'Episode: {episode}, Reward: {episode_reward}')

env.close()
```

Output:
Episode: 0, Reward: -108.37449958632448
Episode: 10, Reward: -99.65493046615883
Episode: 20, Reward: -109.01720450718528
Episode: 30, Reward: -98.54277702978354
Episode: 40, Reward: -98.6473104272563
Episode: 50, Reward: -140.0961763335989
Episode: 60, Reward: -110.87277882853644
Episode: 70, Reward: -103.18383347048673

Episode: 80, Reward: -105.02699768537056
Episode: 90, Reward: -106.03824049568841

Here the pygame is not visible, because I have disabled rendering for fast training of my model. If I want to see the rendering walker graphical representation I should enable the training option.

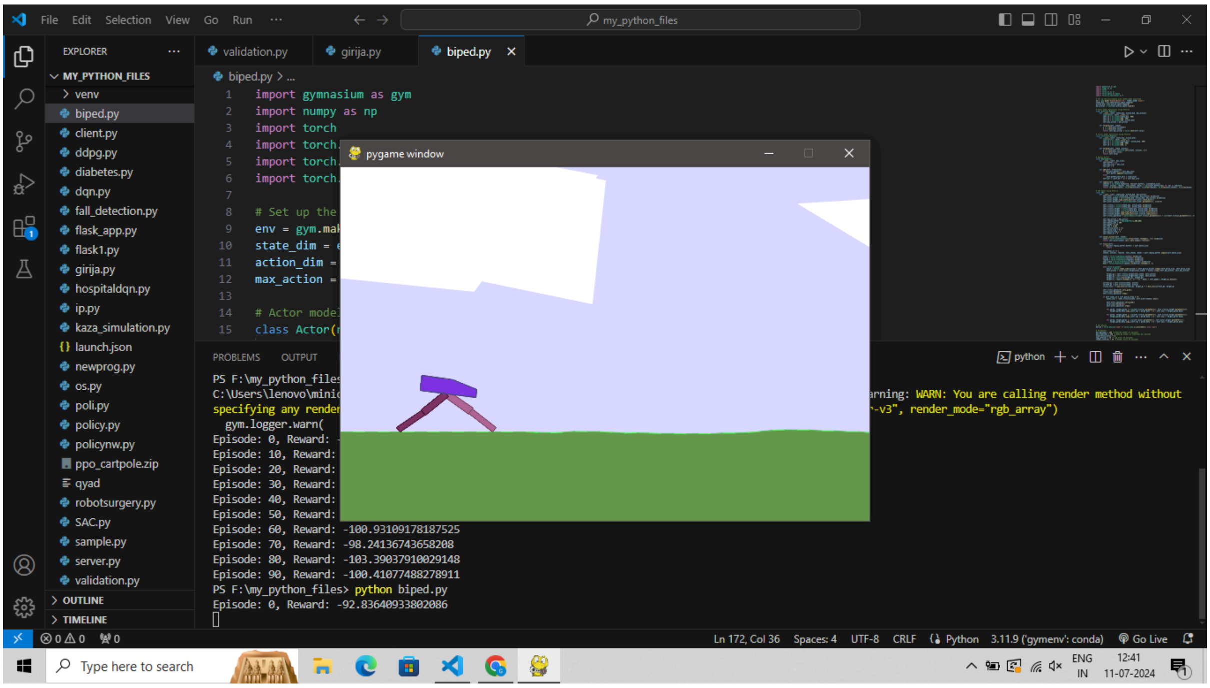
```
import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# Set up the environment with render_mode specified
env = gym.make('BipedalWalker-v3', render_mode='human')
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

# Actor model definition using PyTorch
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()
        self.l1 = nn.Linear(state_dim, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, action_dim)
        self.max_action = max_action

    def forward(self, state):
        a = F.relu(self.l1(state))
        a = F.relu(self.l2(a))
        return self.max_action * torch.tanh(self.l3(a))

# Critic model definition using PyTorch
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.l1 = nn.Linear(state_dim + action_dim, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, 1)
```

```
    def forward(self, state, action):
        q = F.relu(self.l1(torch.cat([state, action], 1)))
        q = F.relu(self.l2(q))
        return self.l3(q)

# Replay Buffer
class ReplayBuffer:
    def __init__(self, max_size):
        self.buffer = []
        self.max_size = max_size
        self.ptr = 0

    def add(self, transition):
        if len(self.buffer) < self.max_size:
            self.buffer.append(transition)
        else:
            self.buffer[self.ptr] = transition
        self.ptr = (self.ptr + 1) % self.max_size

    def sample(self, batch_size):
        indices = np.random.randint(0, len(self.buffer),
size=batch_size)
        states, actions, rewards, next_states, dones =
zip(*[self.buffer[idx] for idx in indices])
        return np.array(states), np.array(actions), np.array(rewards),
np.array(next_states), np.array(dones)

# TD3 Agent using PyTorch
class TD3:
    def __init__(self, state_dim, action_dim, max_action):
        self.actor = Actor(state_dim, action_dim,
max_action).to(device)
        self.actor_target = Actor(state_dim, action_dim,
max_action).to(device)
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.actor_optimizer = optim.Adam(self.actor.parameters(),
lr=3e-4)

        self.critic1 = Critic(state_dim, action_dim).to(device)
        self.critic2 = Critic(state_dim, action_dim).to(device)
        self.critic1_target = Critic(state_dim, action_dim).to(device)
        self.critic2_target = Critic(state_dim, action_dim).to(device)
        self.critic1_target.load_state_dict(self.critic1.state_dict())
```

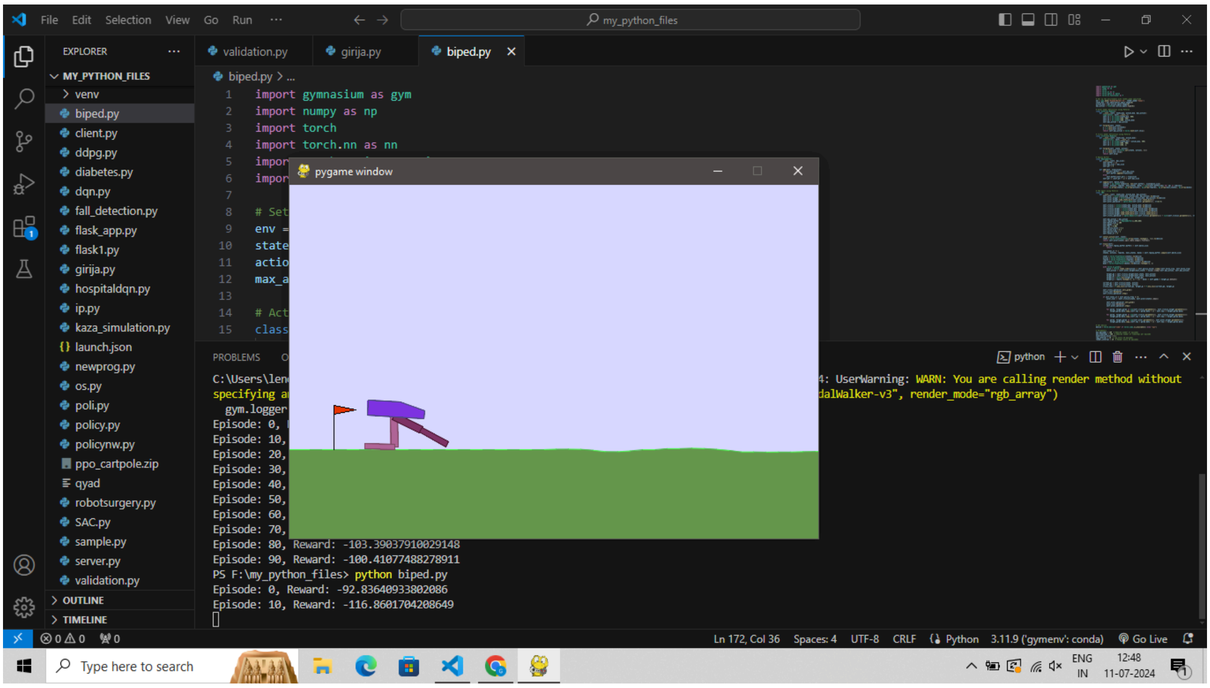
```
        self.critic2_target.load_state_dict(self.critic2.state_dict())
        self.critic_optimizer =
optim.Adam(list(self.critic1.parameters()) +
list(self.critic2.parameters()), lr=3e-4)

        self.max_action = max_action
        self.replay_buffer = ReplayBuffer(1_000_000)
        self.batch_size = 100
        self.gamma = 0.99
        self.tau = 0.005
        self.policy_noise = 0.2
        self.noise_clip = 0.5
        self.policy_freq = 2
        self.total_it = 0

    def select_action(self, state):
        state = torch.FloatTensor(np.array(state).reshape(1,
-1)).to(device)
        return self.actor(state).cpu().data.numpy().flatten()

    def train(self):
        if len(self.replay_buffer.buffer) < self.batch_size:
            return

        self.total_it += 1
        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(self.batch_size)

        state = torch.FloatTensor(states).to(device)
        action = torch.FloatTensor(actions).to(device)
        reward = torch.FloatTensor(rewards).to(device)
        next_state = torch.FloatTensor(next_states).to(device)
        done = torch.FloatTensor(dones).to(device).reshape(-1, 1)

        with torch.no_grad():
            noise = (torch.randn_like(action) *
self.policy_noise).clamp(-self.noise_clip, self.noise_clip)
            next_action = (self.actor_target(next_state) +
noise).clamp(-self.max_action, self.max_action)

            target_q1 = self.critic1_target(next_state, next_action)
            target_q2 = self.critic2_target(next_state, next_action)
            target_q = torch.min(target_q1, target_q2)
```

```
            target_q = reward.reshape(-1, 1) + ((1 - done) * self.gamma
* target_q).detach()

        current_q1 = self.critic1(state, action)
        current_q2 = self.critic2(state, action)
        critic_loss = F.mse_loss(current_q1, target_q) +
F.mse_loss(current_q2, target_q)

        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        if self.total_it % self.policy_freq == 0:
            actor_loss = -self.critic1(state, self.actor(state)).mean()

            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()

            for param, target_param in zip(self.critic1.parameters(),
self.critic1_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)

            for param, target_param in zip(self.critic2.parameters(),
self.critic2_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)

            for param, target_param in zip(self.actor.parameters(),
self.actor_target.parameters()):
                target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)

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

# Parameters
max_episodes = 100  # Reduced number of episodes
max_timesteps = 500  # Reduced number of timesteps per episode
exploration_noise = 0.1
log_interval = 10  # Log every 10 episodes
render_interval = 10  # Render every 10 episodes
```

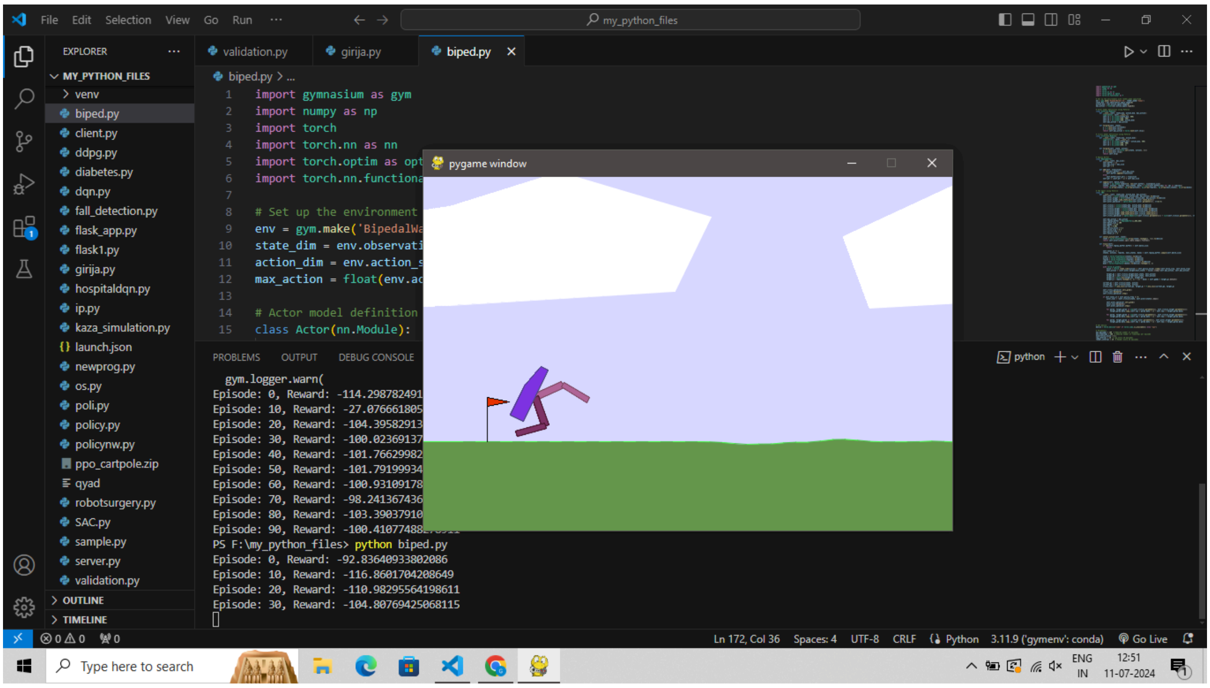
```python
# Initialize TD3 agent
td3_agent = TD3(state_dim, action_dim, max_action)

# Training loop
for episode in range(max_episodes):
    state, _ = env.reset()
    episode_reward = 0
    for t in range(max_timesteps):
        if episode % render_interval == 0:  # Render every
`render_interval` episodes
            env.render()

        action = td3_agent.select_action(state)
        action = action + np.random.normal(0, exploration_noise,
size=action_dim)
        action = action.clip(-max_action, max_action)

        next_state, reward, done, _, _ = env.step(action)
        td3_agent.replay_buffer.add((state, action, reward, next_state,
float(done)))

        state = next_state
        episode_reward += reward

        if done:
            break

        td3_agent.train()

    if episode % log_interval == 0:
        print(f'Episode: {episode}, Reward: {episode_reward}')

env.close()
```



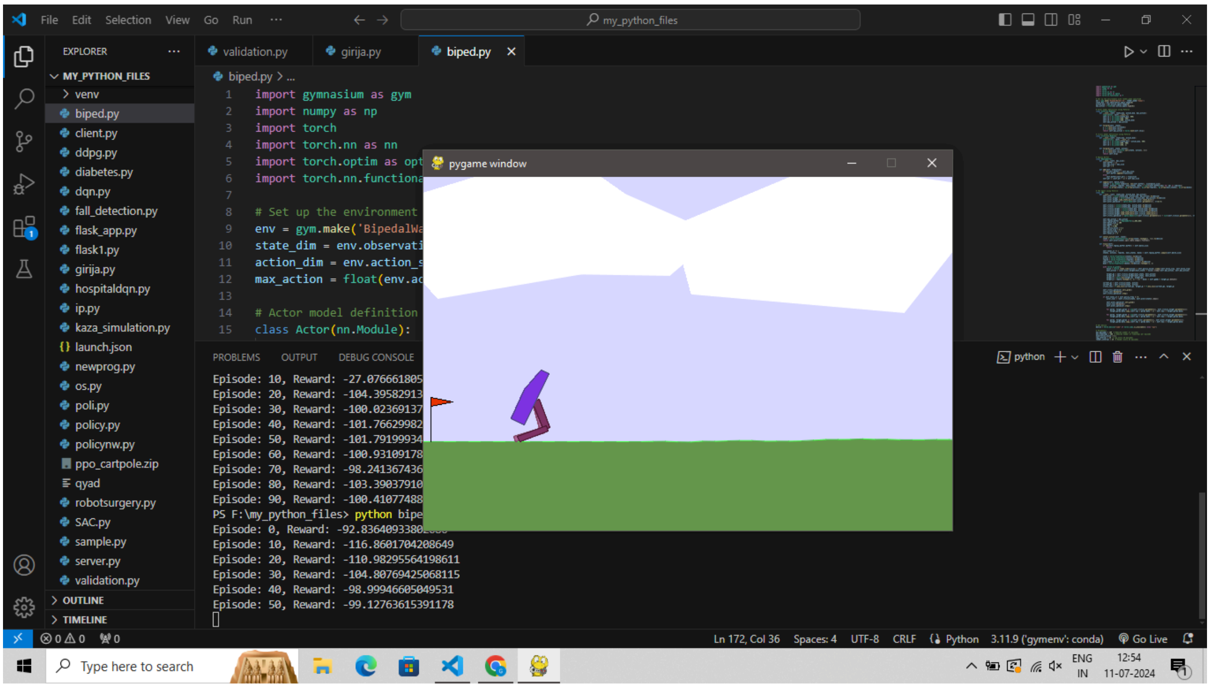
Reward 0



Reward 10



Reward 30



Reward 50

Here this code is implemented in Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm used to the “BipedelWalker-V3” environment.
State Dimension- state_dim is the size of the state space.
Acton Dimension- action_dim is the size of the action space.

Max Action- max_action is the maximum value for actions in the environment.
Tanh for the output layer to ensure actions are in the range [-max_action, max_action]
Input will be the concatenation of state and action vectors. The replay buffer stores transitions (state, action, reward,next_state, and done) to be sampled during training.
Circular Buffer - When it's full, it overwrites old experiences.

Again, actor and critic networks initialize the actor and two critic networks along with their target networks and optimizes and stores it in experience storage. And finally, training is done.