Controlling a Toy Car around a Continuous Grid [Version 1]
Project Code: CCV1

CS60077-Reinforcement Learning

Pranav Mehrotra, 20CS10085

Problem Description:

- **Objective:** Guide the car along a circular track, ensuring it completes a full lap without veering off course from the starting point.
- **Tasks:**
  - Model the state space and reward system for the problem
  - Implement the Deep RL Algorithms as follows:
    - DQN
    - NPG
    - TRPO
    - PPO

Environment, State Space, and Reward Formulation:

- **Environment:**
  - A circular grid featuring a predetermined track, where the car must successfully navigate the entire lap. The environment is replicated through the pygame interface.
- **State Space:**
  - Each state is represented as an eight-element tuple (sensor1_val, sensor2_val, sensor3_val, sensor4_val, sensor5_val, car_x, car_y, car_angle). Here, (car_x, car_y) indicates the current position of the car on the grid, and car_angle represents the car's orientation relative to the horizontal, with values ranging from $-\pi$ to π.
  - During each time step, the car has the option to make a slight left turn, a slight right turn, or continue straight, with a constant speed. It's worth noting that the window coordinates are defined, with the top-left corner at (0, 0) and the bottom-right at (window_size-1, window_size-1).
  - To enhance training for Deep RL agents involving neural networks, two circular regions have been implemented concentric to the track. One is black-colored with a smaller radius, and the other is green-colored with a larger radius. These circles serve to confine and bounce back the car when it attempts to cross their boundaries. This measure is implemented to prevent the car from entering undesired states, ensuring effective training for the neural network-based agents.

(A) DQN Algorithm:

- **Configurations:**
    - Used a neural network with 1 input layer of size 8, two hidden layers of size 64 and 1 output layer of size 3.

    - **Other configurations:**

```
State_dim = 8,
 num_actions= 3,
 hidden_size=128,
 gamma=0.99,
 epsilon=0.3,
 epsilon_min=0.01,
 epsilon_decay=0.995,
 learning_rate=1e-4,
 tau=0.001,
 batch_size=64
, max_memory_size=50000

Few more in the code
```

(B) NPG Algorithm:

- **Configurations:**
    - Used a neural network with 1 input layer of size 8, two hidden layers and 1 output layer of size 3, followed by softmax layer

    - **Other configurations:**

```
gamma = 0.99
   hidden_size = 64
   actor_lr = 5e-4
```

```
  critic_lr = 3e-4
  batch_size = 5
  l2_rate = 1e-3
  max_kl = 1e-2
  max_episode = 10
```

(C) TRPO Algorithm:

- **Configurations:**
  - Used a neural network with 1 input layer of size 8, two hidden layers and 1 output layer of size 3, followed by a softmax layer for the **actor** component of the agent. Also used Droupout for robust learning.
  - Used a neural network with 1 input layer of size 8, two hidden layers and 1 output layer of size 1, for the **critic** component of the agent. Also used Droupout for robust learning.

  - **Other configurations:**

```
gamma=0.995,
 tau=0.97,
 l2_reg=0.001,
 max_kl=0.01,
 damping=0.1,
 seed=65431,
batch_size=14000,
 num_episodes=10,
 hidden_size=64,
 render=False
```

(D) PPO Algorithm:

- **Configurations:**
  - Used a neural network with 1 input layer of size 8, two hidden layers and 1 output layer of size 3, followed by a softmax layer for the **actor** component of the agent. Also used Droupout for robust learning.
  - Used a neural network with 1 input layer of size 8, two hidden layers and 1 output layer of size 1, for the **critic** component of the agent. Also used Droupout for robust learning.

- **Other configurations:**

```
state_dim = env.observation.shape[0]  # 8
  action_dim = env.num_actions  # 3
  render = False

  # training hyperparameters
  max_episode_len = 6000
  max_training_timestep = 210000

  # ppo hyperparameters
  update_timestep = max_episode_len
  K_epochs = 50
  eps_clip = 0.2
  gamma = 0.99
  lr_actor = 0.0003
  lr_critic = 0.001
  random_seed = 33985
```