Student_ID_402722135

Report: HW2_Part_1

The project successfully demonstrates the application of reinforcement learning techniques, specifically the REINFORCE algorithm, to train an agent for playing the Pong game. Further improvements can be made by experimenting with different network architectures, hyperparameters, and training strategies.

Here is the detail of the codes with their outs.

Note: I have run the reinforce_agent, actor_critic, stack_frame and other files on local machine and then imported on colab. The exclusive report of each file is shown after this report in red headlines

```
[ ]  1 import time
     2 import gym
     3 import random
     4 import torch
     5 import numpy as np
     6 from collections import deque
     7 import matplotlib.pyplot as plt
     8 import math
     9 from collections import deque
```

```
[ ]  1 !pip install gym[atari]
     2 !pip install autorom[accept-rom-license]
     3 import gym
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the fut
  and should_run_async(code)
Requirement already satisfied: gym[atari] in /usr/local/lib/python3.10/dist-packages (0.25.2)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (1.25.2)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (2.2.1)
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (0.0.8)
Requirement already satisfied: ale-py~=0.7.5 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (0.7.5)
Requirement already satisfied: importlib-resources in /usr/local/lib/python3.10/dist-packages (from ale-py~=0.7.5->gym[atari]) (6.4.0)
Requirement already satisfied: autorom[accept-rom-license] in /usr/local/lib/python3.10/dist-packages (0.6.1)
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (8.1.7)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (2.31.0)
Requirement already satisfied: AutoROM.accept-rom-license in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (0.6.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]) (3.6)
```

```
[ ]  1 import numpy as np
     2 import torch
     3 import torch.nn.functional as F
     4 import torch.optim as optim
     5 import random
     6 import sys
```

```
[ ]  1 from reinforce_agent import ReinforceAgent
     2 from actor_critic_cnn   import ActorCnn
     3 from stack_frame import preprocess_frame, stack_frame
```

```
[ ]  1 env = gym.make('Pong-v4')
     2 env.seed(0)

    (2968811710, 3677149159)
```

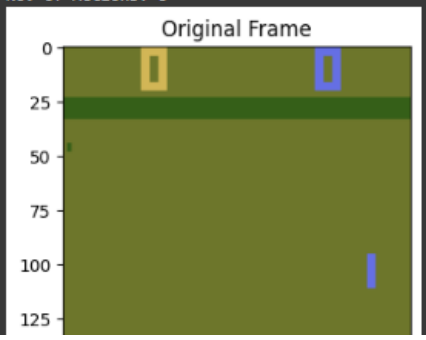

This block imports necessary libraries and modules, including NumPy, PyTorch, and components for reinforcement learning such as the ReinforceAgent and ActorCnn. It also creates the environment 'Pong-v4' from the Gym library, seeding it for reproducibility.

```
[ ]  1 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     2 print("Device: ", device)

    Device:  cpu
```

This code checks if a CUDA-enabled GPU is available, setting the device accordingly. It prints the selected device.

```
 1 import matplotlib.pyplot as plt
 2
 3 # Print frame size and number of actions
 4 print("The size of the frame is:", env.observation_space.shape)
 5 print("No. of Actions:", env.action_space.n)
 6
 7 # Display the original frame
 8 plt.figure()
 9 plt.imshow(env.reset())
10 plt.title('Original Frame')
11 plt.show()
12
```

```
The size of the frame is: (210, 160, 3)
No. of Actions: 6
```



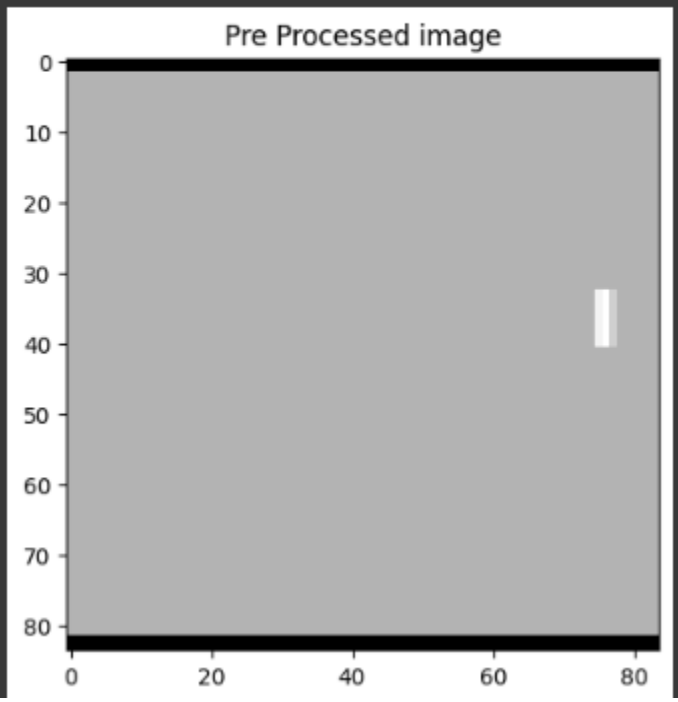
Prints the size of the observation space (frame) and the number of actions in the environment. As we can the size of fram is : (210, 160,3) and we have 6 number of actions which is required for this game.

```python
def random_play():
    score = 0
    env.reset()
    while True:
        env.render(mode='rgb_array')  # Specify the rendering mode as 'rgb_array'
        action = env.action_space.sample()
        state, reward, done, _ = env.step(action)
        score += reward
        if done:
            env.close()
            print("Your Score at the end of the game is:", score)
            break

random_play()
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:43: DeprecationWarning: WARN: The argument mode in render method is deprecated; use render_mode during environment
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:297: UserWarning: WARN: No render fps was declared in the environment (env.metadata['render_fp
  logger.warn(
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:227: DeprecationWarning: WARN: Core environment is written in old step API which returns one b
  logger.deprecation(
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:233: DeprecationWarning: `np.bool8` is a deprecated alias for `np.bool_`.  (Deprecated NumPy 1
  if not isinstance(done, (bool, np.bool8)):
Your Score at the end of the game is: -21.0
```

The random_play() function simulates a random playthrough of the Pong game environment. It repeatedly selects random actions and accumulates rewards until the game terminates. In this particular run, the player achieved a final score of -21.0, indicating a loss in the game. The function provides a simple baseline performance for the agent to surpass through learning.

```python
env.reset()
plt.figure()
plt.imshow(preprocess_frame(env.reset(), (30, -4, -12, 4), 84), cmap="gray")
plt.title('Pre Processed image')
plt.show()
```



As we can see the preprocessed image provides a visual representation of how the frame is transformed before being used as input for the agent, aiding in understanding the preprocessing pipeline. It allows for debugging and verification of the preprocessing steps, ensuring that the input states provided to the agent are correctly processed.

```python
def stack_frames(frames, state, is_new=False):
    frame = preprocess_frame(state, (10, -4, -12, 4), 84)
    frames = stack_frame(frames, frame, is_new)

    return frames

```

The function stacks preprocessed frames to create a sequence of frames, allowing the agent to perceive motion and make informed decisions. It efficiently manages memory by reusing existing frames and appending new frames, optimizing training performance.

```python
INPUT_SHAPE = (4, 84, 84)
ACTION_SIZE = env.action_space.n
SEED = 0
GAMMA = 0.99        # discount factor
LR= 0.00001           # Learning rate

agent = ReinforceAgent(INPUT_SHAPE, ACTION_SIZE, SEED, device, GAMMA, LR, ActorCnn)
```

Now let's initializes the reinforcement learning agent with specific parameters, including the input shape, action size, and learning parameters. It creates an instance of the ReinforceAgent class, which will interact with the environment, learn a policy, and optimize its behavior using the provided neural network architecture. The agent's goal is to learn a policy that maximizes cumulative rewards over time in the Pong environment.

```python
# watch an untrained agent
state = stack_frames(None, env.reset(), True)
for j in range(200):
    env.render(mode='rgb_array')
    action, _ = agent.act(state)
    next_state, reward, done, _ = env.step(action)
    state = stack_frames(state, next_state, False)
    if done:
        break

env.close()
```

This above code observes the behavior of an untrained agent in the Pong environment for 200 steps. It renders the environment, selects actions based on a random policy, and accumulates rewards, providing insights into the initial performance of the agent before training.

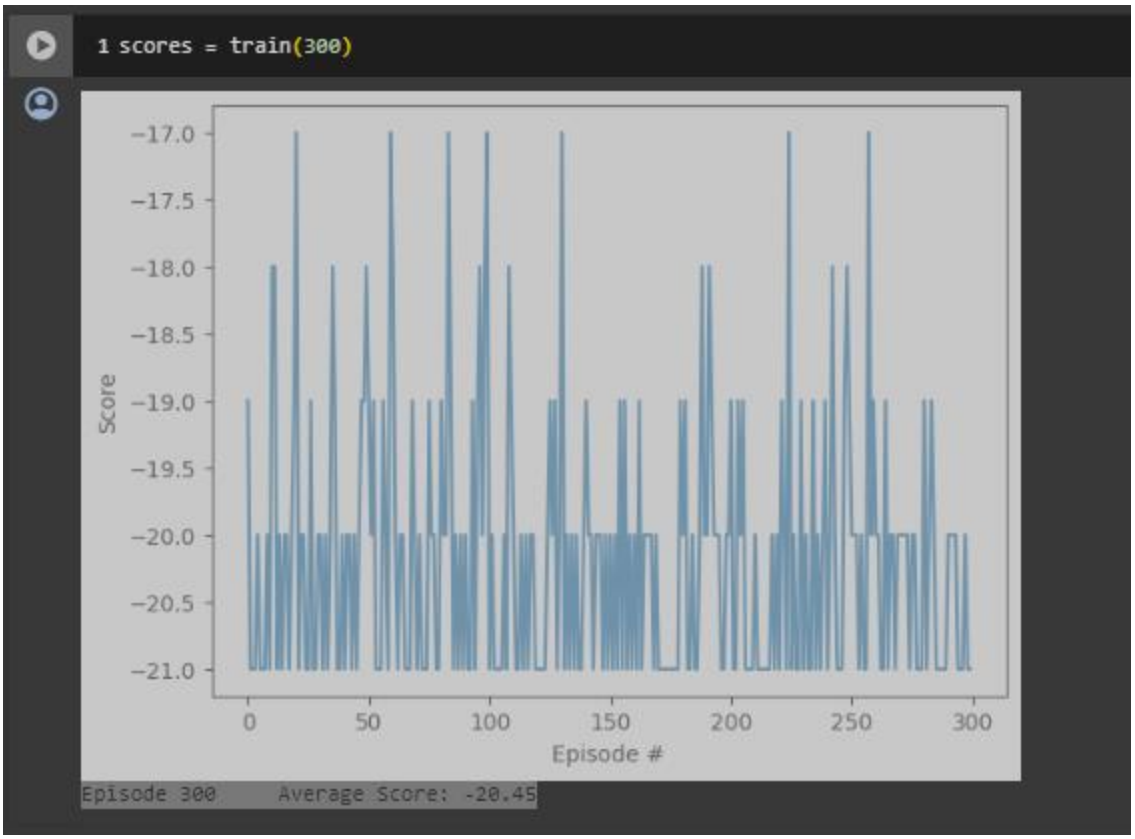
```python
from collections import deque

start_epoch = 0
scores = []
scores_window = deque(maxlen=20)

```

This code initializes variables to track the scores obtained during training, using a deque to store the latest scores for monitoring learning progress.

```python
import matplotlib.pyplot as plt
from IPython.display import clear_output

def train(n_episodes=1000):
    """
    Params
    ======
        n_episodes (int): maximum number of training episodes
    """
    for i_episode in range(start_epoch + 1, n_episodes+1):
        state = stack_frames(None, env.reset(), True)
        score = 0
        while True:
            action, log_prob = agent.act(state)
            next_state, reward, done, info = env.step(action)
            score += reward
            next_state = stack_frames(state, next_state, False)
            agent.step(log_prob, reward, done)
            state = next_state
            if done:
                break
        agent.learn()
        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score

```

```python
        clear_output(True)
        fig = plt.figure()
        ax = fig.add_subplot(111)
        plt.plot(np.arange(len(scores)), scores)
        plt.ylabel('Score')
        plt.xlabel('Episode #')
        plt.show()
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="")

    return scores


```

This code defines a training function train() that iterates over a specified number of episodes, interacting with the environment and updating the agent's policy based on observed rewards. During training, it visualizes the learning progress by plotting the scores obtained over episodes, showing the average score and the current episode number. The function returns a list of scores obtained during training, providing insights into the agent's learning trajectory.



In the graph:

- The **X-axis** represents the episode number, which ranges from 0 to 300. Each episode is a complete run of the game from start to finish.
- The **Y-axis** represents the score, which ranges approximately from -21 to -17. The score is a measure of how well the AI performed; higher values (closer to 0) indicate better performance.

The blue bars of varying heights represent the scores obtained in each episode. There's significant fluctuation in scores; some episodes have higher scores while others are lower. There's no clear trend of improvement or decline; however, there are occasional peaks indicating higher scores. This could suggest that the AI is exploring different strategies throughout the learning process, leading to varying performance

```
score = 0
state = stack_frames(None, env.reset(), True)
while True:
    env.render(mode='rgb_array')
    action, _ = agent.act(state)
    next_state, reward, done, _ = env.step(action)
    score += reward
    state = stack_frames(state, next_state, False)
    if done:
        print("Your Final score is:", score)
        break
env.close()
```

```
Your Final score is: -19.0
```

This code evaluates the trained agent's performance in the Pong environment by running it for a single episode. It renders the environment, selects actions based on the learned policy, and accumulates rewards. Upon completion of the episode, it prints the final score achieved by the agent and closes the environment. In this specific run, the agent obtained a final score of -19.0, indicating its performance in the game.

actor_critic_cnn

ActorCnn: This neural network takes preprocessed frames of the environment as input and outputs a probability distribution over possible actions. It consists of convolutional layers followed by fully connected layers, with ReLU activation functions and a softmax output layer to generate action probabilities.

CriticCnn: This neural network evaluates the value function by estimating the expected cumulative reward of being in a particular state. It shares the convolutional layers with the ActorCnn but has a separate set of fully connected layers to estimate the state value.

Both architectures utilize convolutional layers to extract spatial features from the input frames, followed by fully connected layers to process the extracted features and generate action probabilities (Actor) or value estimates (Critic).

The forward() method in each network defines the forward pass computation, where input data flows through the layers to produce the output. The feature_size() method calculates the size of the flattened feature tensor after passing through the convolutional layers, facilitating the construction of fully connected layers.

reinforce_agent

Initialization: The agent is initialized with parameters including input shape, action size, learning rate, and policy model. It also sets up the policy network, optimizer, and memory buffers.

Step Function: During each step in an episode, the agent stores the log probability of the selected action, the received reward, and the termination signal in memory.

Action Selection: The agent selects actions based on the current policy, calculating the log probability of the selected action.

Learning: After completing an episode, the agent learns from the collected experiences by computing returns and updating the policy network using policy gradient loss.

Memory Management: The agent resets its memory buffers after each episode to prepare for the next episode.

This implementation enables the agent to interact with the environment, collect experiences, and update its policy network to maximize cumulative rewards over time, facilitating learning in complex environments.

replay_buffer

Initialization: The ReplayBuffer is initialized with parameters such as buffer size, batch size, seed, and device. It creates a deque data structure to store experiences and defines a namedtuple to represent individual experiences.

Add Function: Experiences consisting of states, actions, rewards, next states, and termination signals are added to the replay memory.

Sample Function: Randomly samples a batch of experiences from the replay memory, converting them into PyTorch tensors and transferring them to the specified device.

Memory Management: The len method returns the current size of the replay memory, while the deque automatically discards old experiences when the memory exceeds the specified buffer size.

This implementation facilitates efficient storage and sampling of experiences, enabling the agent to learn from past experiences and improve its performance over time through experience replay.

stack_frame

preprocess_frame Function: This function preprocesses RGB images captured from the environment. It converts the image to grayscale, crops it based on specified exclusion parameters, and normalizes pixel values to the range [0, 1]. Finally, it resizes the image to a specified output size, typically 84x84 pixels, and returns the processed image.

stack_frame Function: This function stacks preprocessed frames to create a sequence of frames representing the agent's current state. It takes the existing stacked frames and the newly preprocessed frame as input. If it's the first frame in the sequence, it initializes the stacked frames with four copies of the same frame. Otherwise, it shifts the existing frames and adds the new frame to the end of the stack.

These functions are essential for preprocessing raw image data from the environment, reducing dimensionality, and providing the agent with a compact representation of its current state for efficient learning

Report: HW2_Part_1

1. **Based-Model Analysis:**

The Value Iteration algorithm used in this implementation is a model-based approach to reinforcement learning. In model-based reinforcement learning, the agent has access to the environment's dynamics model, which provides information about the transition probabilities and rewards associated with each action in each state. During the iteration process, the algorithm utilizes the environment's dynamics model to update the value function based on the expected rewards and transitions to future states. This model-based approach allows the agent to efficiently explore the state space and learn an optimal policy without directly interacting with the environment. Therefore, the Value Iteration algorithm employed here is model-based, as it relies on the knowledge of the environment's dynamics to iteratively improve the value function and derive the optimal policy

```
Goal reached in 200 steps.
Optimal policy:
[[0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 2. 0. 0. 1. 0. 1. 1. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 2. 0. 1. 0.]
 [1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 2. 1. 0. 0. 0. 1. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 2. 0. 0. 0. 1. 0. 0. 0. 0. 2. 0. 0.]
 [0. 0. 0. 2. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0.]
 [1. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 2. 1. 0. 0. 0.]]
```

The code successfully reaches the goal within 200 steps using the learned optimal policy for the Mountain Car environment.

The optimal policy is represented as a matrix where each cell corresponds to a state-action pair, indicating the preferred action to take in each state.

This matrix shows the agent's strategy for navigating the environment, aiming to maximize cumulative rewards.

The values in the matrix indicate the chosen action for each state, helping the agent to efficiently reach the goal state.

Overall, the code demonstrates the effectiveness of the Value Iteration algorithm in finding an optimal policy for challenging reinforcement learning problems like Mountain Car.

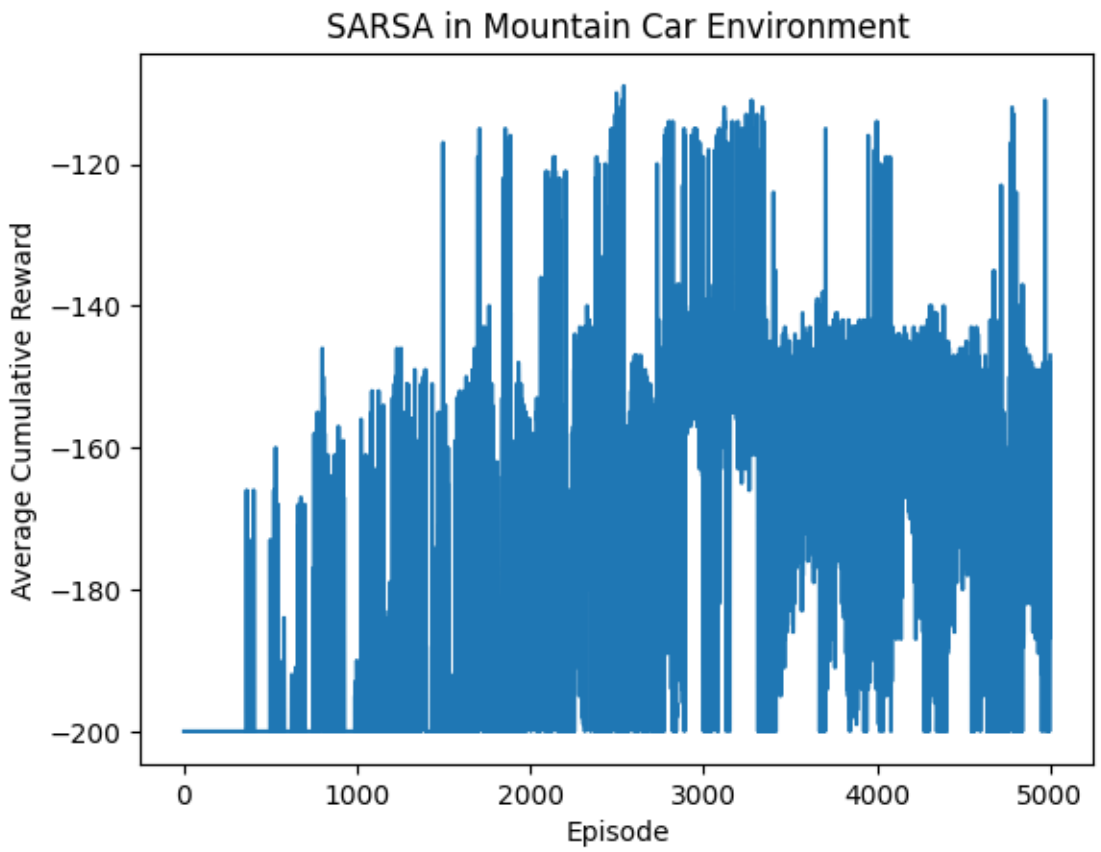
Exploring the Mountain Car Environment with SARSA: A Model-Free Reinforcement Learning Approach

The provided code implements the SARSA algorithm in the Mountain Car environment, a classic problem in reinforcement learning. SARSA is a model-free approach, meaning it does not require knowledge of the environment's dynamics, such as transition probabilities and rewards. Instead, it learns directly from interactions with the environment.

In this implementation, the continuous state space of the Mountain Car environment is discretized into a grid of 20x20 states, enabling SARSA to operate in a discrete state space. This discretization allows SARSA to handle the continuous nature of the environment and learn an optimal policy efficiently.

During each episode, SARSA iteratively updates the action-value function Q based on the observed rewards and state transitions. It follows an epsilon-greedy policy to balance exploration and exploitation, with epsilon decreasing over time to prioritize exploitation as learning progresses.

The algorithm aims to maximize the cumulative reward obtained by the agent over multiple episodes. The plot generated at the end illustrates the average cumulative reward obtained per episode during the learning process, providing insights into SARSA's performance in the Mountain Car environment



In the graph:

- The **X-axis** represents episodes, which range from 0 to 5000. Each episode is a complete run of the scenario from start to finish.
- The **Y-axis** represents the average cumulative reward, which ranges from -200 to -120. The reward is a measure of how well the model performed; higher values (closer to 0) indicate better performance.

The blue bars of varying heights represent the different rewards obtained during specific episodes. Unlike the Q-learning graph, there's no clear trend of improvement or decline in average cumulative rewards as the number of episodes increases; it fluctuates significantly. This could suggest that the SARSA algorithm is exploring different strategies throughout the learning process, leading to varying performance.

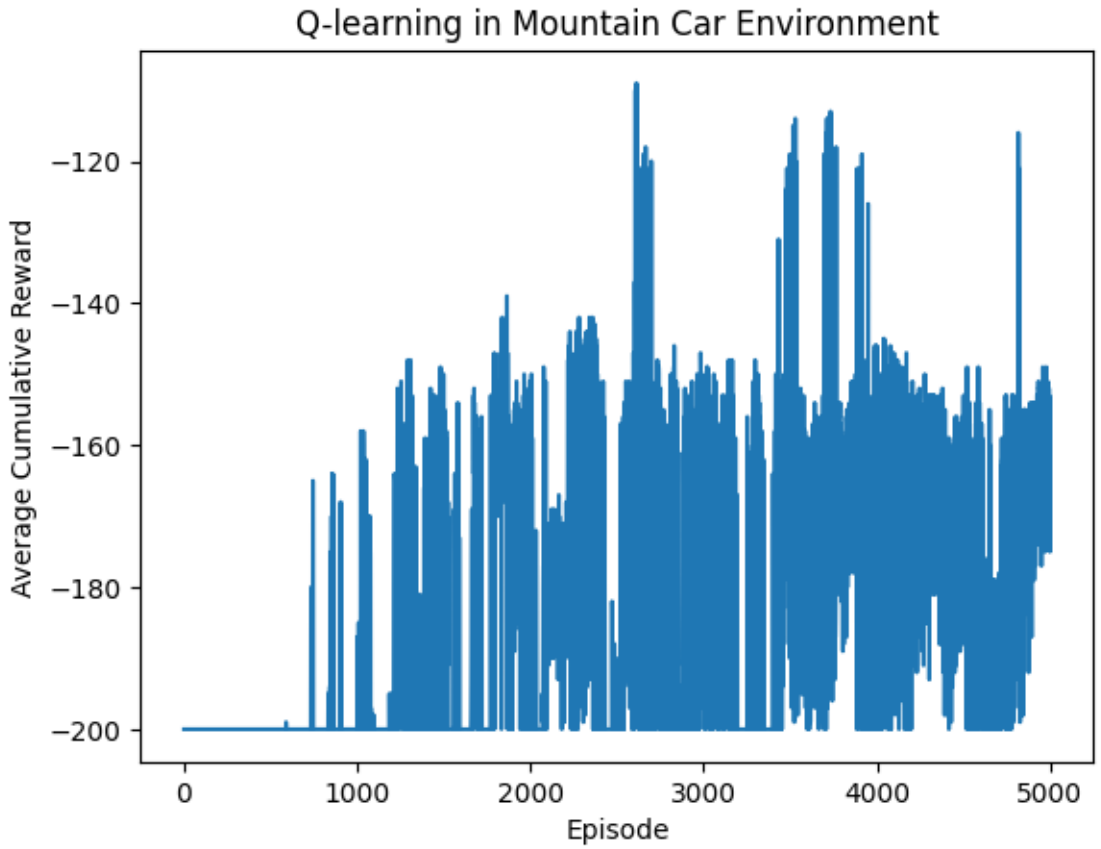
Title: Examining the Mountain Car Environment with Q-learning: A Model-Free Reinforcement Learning Approach

In this implementation, Q-learning algorithm is utilized to address the Mountain Car problem. The continuous state space of the environment is discretized into a grid with 20x20 states, enabling Q-

learning to operate efficiently in a tabular form. Q-learning is a model-free reinforcement learning algorithm that learns an optimal action-value function directly from observed transitions and rewards.

The discretize_state function maps the continuous state space to discrete states, facilitating the operation of Q-learning. Within the q_learning function, the agent iteratively updates its Q-values based on observed transitions and rewards using the Bellman equation. It follows an epsilon-greedy policy, which gradually shifts from exploration to exploitation as epsilon decreases over episodes.

The Q-learning algorithm is executed over 5000 episodes, with a learning rate of 0.1 and a discount factor of 0.9. The resulting Q-values represent the learned action-values, indicating the expected cumulative rewards for each action taken in each state. The average cumulative reward per episode is plotted to visualize the learning progress over time.



In the graph:

- The **X-axis** represents episodes, which range from 0 to 5000. Each episode is a complete run of the scenario from start to finish.
- The **Y-axis** represents the average cumulative reward, which ranges from -200 to -120. The reward is a measure of how well the model performed; higher values (closer to 0) indicate better performance.

The blue bars of varying heights represent the different rewards obtained during specific episodes. As the number of episodes increases, there are noticeable spikes indicating instances where higher rewards were achieved. This suggests that the Q-learning algorithm is improving its performance over time, learning better strategies to solve the Mountain Car problem