

✓ Reinforcement Learning using SARSA in OpenAI Gym / Gymnasium

✓ Imports and Installs

```
!pip install gymnasium[atari] # for the gym library
!pip install gymnasium[accept-rom-license] #to add atari envs
!pip install ale-py # for atari envs
import gymnasium as gym
import numpy as np
import random
import time
from IPython.display import clear_output
```

 Show hidden output

✓ Step 1: Creating the Environment

```
env_name = "MountainCar-v0"
```

 /usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `tra` and `should_run_async(code)`

```
env = gym.make(env_name, render_mode="rgb_array")
```

✓ Step 2: Creating the Agent with SARSA Learning

These are the hyperparameters specific to the training within the environment and learning task. These are used within the Agent / Learner class and are similar to the training of a neural network.

```
#MAX_NUM_EPISODES = 1000
MAX_NUM_EPISODES = 25000 #100001
STEPS_PER_EPISODE = 200 # This is specific to MountainCar. May change with env
EPSILON_MIN = 0.005
max_num_steps = MAX_NUM_EPISODES * STEPS_PER_EPISODE
EPSILON_DECAY = 500 * EPSILON_MIN / max_num_steps
ALPHA = 0.1 # Learning rate
GAMMA = 0.9 #1.0 # Discount factor
NUM_DISCRETE_BINS = 30 # Number of bins to Discretize each observation dim

class SARSA_Learner(object):
    def __init__(self, env):
        self.obs_shape = env.observation_space.shape
        self.obs_high = env.observation_space.high
        self.obs_low = env.observation_space.low
        # printing all the obs_ for debugging
        #print(self.obs_high, 'obs high')
        #print(self.obs_low, 'obs low')
        self.obs_bins = NUM_DISCRETE_BINS # Number of bins to Discretize each observation dim
        self.bin_width = (self.obs_high - self.obs_low) / self.obs_bins
        self.action_shape = env.action_space.n
        # Create a multi-dimensional array (aka. Table) to represent the
        # Q-values
        self.Q = np.zeros((self.obs_bins + 1, self.obs_bins + 1,
                                self.action_shape)) # (51 x 51 x 3)
        self.alpha = ALPHA # Learning rate
        self.gamma = GAMMA # Discount factor
        self.epsilon = 0.9
        ### Unique to SARSA Method ###
        self.rewards = []
    ''' def discretize(self, obs):
        # Convert the observation to a tuple of integers using list comprehension
        # and unpack it using the * operator when indexing into self.Q
```

```

    return tuple([int(np.round((obs[i] - self.obs_low[i]) / self.bin_width[i])) for i in range(len(obs))])

def get_action(self, obs):
    discretized_obs = self.discretize(obs)
    # Epsilon-Greedy action selection
    if self.epsilon > EPSILON_MIN:
        self.epsilon -= EPSILON_DECAY
    if np.random.random() > self.epsilon:
        return np.argmax(self.Q[discretized_obs]) # unpack the tuple
    else: # Choose a random action
        return np.random.choice([a for a in range(self.action_shape)])
'''

def discretize(self, obs):
    #clipped_obs = np.clip(obs[0], self.obs_low, self.obs_high)
    #return tuple(((clipped_obs - self.obs_low) / self.bin_width).astype(int))
    #discretized_env = (self.obs_high - self.obs_low) / self.obs_bins
    #discretized_pos = int((clipped_obs[0] - self.obs_low[0]) / discretized_env[0])
    #discretized_vel = int((clipped_obs[1] - self.obs_low[1]) / discretized_env[1])
    #return discretized_pos, discretized_vel
    discrete_obs = (obs[0] - self.obs_low) / self.bin_width
    return tuple(np.clip(discrete_obs.astype(int), 0, self.obs_bins - 1))

def get_action(self, obs):
    discretized_obs = self.discretize(obs)
    # Epsilon-Greedy action selection
    if self.epsilon > EPSILON_MIN:
        self.epsilon -= EPSILON_DECAY
    if np.random.random() > self.epsilon:
        return np.argmax(self.Q[discretized_obs])
    else: # Choose a random action
        return np.random.choice([a for a in range(self.action_shape)])

def learn(self, obs, action, reward, next_obs, next_action):
    '''
    This is the SARSA learning method that uses the get_action function
    to retrieve the next state-action
    as input to updating the Q learning matrix, versus the
    max Q value of the next state as in Q Learning.
    '''
    discretized_obs = self.discretize(obs)
    discretized_next_obs = self.discretize(next_obs)
    # change self.Q[discretized_next_obs][next_action] to np.max(self.Q[discretized_next_obs])
    # if want to use Q learning
    td_target = reward + self.gamma * self.Q[discretized_next_obs][next_action]
    td_error = td_target - self.Q[discretized_obs][action]
    self.Q[discretized_obs][action] += self.alpha * td_error

def train(agent, env):
    best_reward = -float('inf')
    for episode in range(MAX_NUM_EPISODES):
        done = False
        obs = env.reset()
        action = agent.get_action(obs)
        ### printing the obs for debugging
        #print(obs, ' obs')
        #print(type(obs), ' obs type')
        #print(obs[0][0], ' obs[0]')
        #print(obs[0][1], ' obs[1]')
        #print(obs[1], ' obs[1]')
        #print(type(obs[0]), ' obs[0] type')
        #print(type(obs[1]), ' obs[1] type')
        total_reward = 0.0
        while not done:
            next_obs, reward, terminated, truncated, info = env.step(action)
            # retrieving the next action necessary for the SARSA learning method
            next_action = agent.get_action(next_obs)
            done = terminated or truncated
            agent.learn(obs, action, reward, next_obs, next_action)
            obs = next_obs
            action = next_action
            total_reward += reward
        if total_reward > best_reward:
            best_reward = total_reward
        print("Episode#{:} reward:{:} best_reward:{:} eps:{:}".format(episode,
                                                                    total_reward, best_reward, agent.epsilon))

# Return the trained policy

```

```
return np.argmax(agent.Q, axis=2)
```

```
def test(agent, env, policy):
    done = False
    obs = env.reset()
    total_reward = 0.0
    while not done:
        action = policy[agent.discretize(obs)]
        next_obs, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        obs = next_obs
        total_reward += reward
    return total_reward
```

 /usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `tra` and `should_run_async` (code)

✓ Step 3: Instantiating & Training the Learner / Agent

In this stage, the agent is still learning its' preference / policy. In the testing phase, the outcome of its' preferences will be monitored and visualized.

```
agent = SARSA_Learner(env)
learned_policy = train(agent, env)
```


✓ Step 4: Evaluating the Training by Recording the Agent within the Environment

```
# Uses the Gym Monitor wrapper to evaluate the agent and record video
# only one video will be saved
```

```
# video of the final episode with the episode trigger
env = gym.wrappers.RecordVideo(
    env, "./gym_monitor_output", episode_trigger=lambda x: x == 0)
```

```
test(agent, env, learned_policy)
```

```
env.close()
```

 /usr/local/lib/python3.10/dist-packages/gymnasium/wrappers/record_video.py:94: UserWarning: **WARN: Overwriting existing video**
 logger.warn(
 Moviepy - Building video /content/gym_monitor_output/rl-video-episode-26000.mp4.
 Moviepy - Writing video /content/gym_monitor_output/rl-video-episode-26000.mp4

 Moviepy - Done !
 Moviepy - video ready /content/gym_monitor_output/rl-video-episode-26000.mp4
 Moviepy - Building video /content/gym_monitor_output/rl-video-episode-0.mp4.
 Moviepy - Writing video /content/gym_monitor_output/rl-video-episode-0.mp4

 Moviepy - Done !
 Moviepy - video ready /content/gym_monitor_output/rl-video-episode-0.mp4

Start coding or [generate](#) with AI.

