Testing

Actual **codes** @ on-policy SARSA

```
import gym
import numpy as np
import cv2

# Define the discretization parameters
env = gym.make('CartPole-v1')

# Convert continuous state to discrete state
upperBounds = env.observation_space.high
lowerBounds = env.observation_space.low
cartVelocityMin = -3
cartVelocityMax = 3
poleAngleVelocityMin = -10
poleAngleVelocityMax = 10
upperBounds[1] = cartVelocityMax
upperBounds[3] = poleAngleVelocityMax
lowerBounds[1] = cartVelocityMin
lowerBounds[3] = poleAngleVelocityMin

numberOfBinsPosition = 30
numberOfBinsVelocity = 30
numberOfBinsAngle = 30
numberOfBinsAngleVelocity = 30
DISCRET_NBR = [numberOfBinsPosition, numberOfBinsVelocity, numberOfBinsAngle, numberOfBinsAngleVelocity]


def convert_state_discrete(state):
    position = state[0]
    velocity = state[1]
    angle = state[2]
    angularVelocity = state[3]

    cartPositionBin = np.linspace(lowerBounds[0], upperBounds[0], DISCRET_NBR[0])
    cartVelocityBin = np.linspace(lowerBounds[1], upperBounds[1], DISCRET_NBR[1])
    poleAngleBin = np.linspace(lowerBounds[2], upperBounds[2], DISCRET_NBR[2])
    poleAngleVelocityBin = np.linspace(lowerBounds[3], upperBounds[3], DISCRET_NBR[3])

    indexPosition = np.maximum(np.digitize(position, cartPositionBin) - 1, 0)
    indexVelocity = np.maximum(np.digitize(velocity, cartVelocityBin) - 1, 0)
    indexAngle = np.maximum(np.digitize(angle, poleAngleBin) - 1, 0)
    indexAngularVelocity = np.maximum(np.digitize(angularVelocity, poleAngleVelocityBin) - 1, 0)

    return tuple([indexPosition, indexVelocity, indexAngle, indexAngularVelocity])


def generate_Q():
    return np.zeros(tuple(DISCRET_NBR) + (2,))


def sarsa(env, num_episodes, alpha, gamma, epsilon):
    num_actions = env.action_space.n
    num_states = tuple(DISCRET_NBR)

    Q = generate_Q()

    for episode in range(num_episodes):
        observation = env.reset()
        state = convert_state_discrete(observation)
        action = epsilon_greedy_policy(Q, state, epsilon)
        while True:
            # Take action and observe next state and reward
            next_state, reward, done, _ = env.step(action)
            next_state=convert_state_discrete(next_state)
            # Choose next action using epsilon-greedy policy
            next_action = epsilon_greedy_policy(Q, next_state, epsilon)

            # Update Q-value of current state-action pair
            Q[state][action] += alpha * (reward + gamma * Q[next_state][next_action] - Q[state][action])
```

```
            state = next_state
            action = next_action

            if done:
                break
    return Q
def epsilon_greedy_policy(Q, state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.randint(len(Q[state]))
    else:
        return np.argmax(Q[state])

# Test the learned policy
def test_policy(env, Q):
    observation = env.reset()
    state = convert_state_discrete(observation)
    done = False
    total_reward = 0

    frames = []  # List to store environment frames

    while not done:
        action = np.argmax(Q[state])
        state, reward, done, _ = env.step(action)
        state = convert_state_discrete(state)
        total_reward += reward

        # Save the rendered frame
        frame = env.render(mode='rgb_array')
        frames.append(frame)

    return total_reward, frames

if __name__ == '__main__':
    # Set hyperparameters
    num_episodes = 5000
    alpha = 0.1  # learning rate
    gamma = 0.99  # discount factor
    epsilon = 0.1  # exploration rate

    # Run SARSA algorithm
    Q = sarsa(env, num_episodes, alpha, gamma, epsilon)

    # Test the learned policy
    total_rewards = []
    all_frames = []

    for _ in range(100):
        reward, frames = test_policy(env, Q)
        total_rewards.append(reward)
        all_frames.extend(frames)

    average_reward = np.mean(total_rewards)
    print("Average reward over 100 test episodes:", average_reward)

    env.close()

    # Display the recorded frames as a video
    height, width, _ = all_frames[0].shape
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    video = cv2.VideoWriter('cartpole.mp4', fourcc, 30.0, (width, height))

    for frame in all_frames:
        video.write(frame)

    video.release()
```

```
 Average reward over 100 test episodes: 65.89
```

Actual **codes** @ off-policy Q learning

```
import gym
import numpy as np
import cv2

# Define the discretization parameters
env = gym.make('CartPole-v1')

# Convert continuous state to discrete state
upperBounds = env.observation_space.high
lowerBounds = env.observation_space.low
cartVelocityMin = -3
cartVelocityMax = 3
poleAngleVelocityMin = -10
poleAngleVelocityMax = 10
upperBounds[1] = cartVelocityMax
upperBounds[3] = poleAngleVelocityMax
lowerBounds[1] = cartVelocityMin
lowerBounds[3] = poleAngleVelocityMin

numberOfBinsPosition = 30
numberOfBinsVelocity = 30
numberOfBinsAngle = 30
numberOfBinsAngleVelocity = 30
DISCRET_NBR = [numberOfBinsPosition, numberOfBinsVelocity, numberOfBinsAngle, numberOfBinsAngleVelocity]


def convert_state_discrete(state):
    position = state[0]
    velocity = state[1]
    angle = state[2]
    angularVelocity = state[3]

    cartPositionBin = np.linspace(lowerBounds[0], upperBounds[0], DISCRET_NBR[0])
    cartVelocityBin = np.linspace(lowerBounds[1], upperBounds[1], DISCRET_NBR[1])
    poleAngleBin = np.linspace(lowerBounds[2], upperBounds[2], DISCRET_NBR[2])
    poleAngleVelocityBin = np.linspace(lowerBounds[3], upperBounds[3], DISCRET_NBR[3])

    indexPosition = np.maximum(np.digitize(position, cartPositionBin) - 1, 0)
    indexVelocity = np.maximum(np.digitize(velocity, cartVelocityBin) - 1, 0)
    indexAngle = np.maximum(np.digitize(angle, poleAngleBin) - 1, 0)
    indexAngularVelocity = np.maximum(np.digitize(angularVelocity, poleAngleVelocityBin) - 1, 0)

    return tuple([indexPosition, indexVelocity, indexAngle, indexAngularVelocity])


def generate_Q():
    return np.zeros(tuple(DISCRET_NBR) + (2,))


def q_learning(env, num_episodes, alpha, gamma, epsilon):
    num_actions = env.action_space.n
    num_states = tuple(DISCRET_NBR)

    Q = generate_Q()

    for episode in range(num_episodes):
        observation = env.reset()
        state = convert_state_discrete(observation)
        while True:
            # Choose action using epsilon-greedy policy
            action = epsilon_greedy_policy(Q, state, epsilon)

            # Take action and observe next state and reward
            next_state, reward, done, _ = env.step(action)
            next_state = convert_state_discrete(next_state)

            # Update Q-value of current state-action pair
            best_next_action = np.argmax(Q[next_state])
            Q[state][action] += alpha * (reward + gamma * Q[next_state][best_next_action] - Q[state][action])

            state = next_state

            if done:
                break
    return Q
def epsilon_greedy_policy(Q, state, epsilon):
    if np.random.rand() < epsilon:
```

```
            return np.random.randint(len(Q[state]))
        else:
            return np.argmax(Q[state])

    # Test the learned policy
    def test_policy(env, Q):
        observation = env.reset()
        state = convert_state_discrete(observation)
        done = False
        total_reward = 0

        frames = []  # List to store environment frames

        while not done:
            action = np.argmax(Q[state])
            state, reward, done, _ = env.step(action)
            state = convert_state_discrete(state)
            total_reward += reward

            # Save the rendered frame
            frame = env.render(mode='rgb_array')
            frames.append(frame)

        return total_reward, frames

    if __name__ == '__main__':
        # Set hyperparameters
        num_episodes = 5000
        alpha = 0.1  # learning rate
        gamma = 0.99  # discount factor
        epsilon = 0.1  # exploration rate

        # Run Q_learning
        Q = q_learning(env, num_episodes, alpha, gamma, epsilon)

        # Test the learned policy
        total_rewards = []
        all_frames = []

        for _ in range(100):
            reward, frames = test_policy(env, Q)
            total_rewards.append(reward)
            all_frames.extend(frames)

        average_reward = np.mean(total_rewards)
        print("Average reward over 100 test episodes:", average_reward)

        env.close()

        # Display the recorded frames as a video
        height, width, _ = all_frames[0].shape
        fourcc = cv2.VideoWriter_fourcc(*'mp4v')
        video = cv2.VideoWriter('Q-learning.mp4', fourcc, 30.0, (width, height))

        for frame in all_frames:
            video.write(frame)

        video.release()
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:317: DeprecationWarning: WARN: Initializing wrapper in old step API which returns or
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/wrappers/step_api_compatibility.py:39: DeprecationWarning: WARN: Initializing environment ir
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/core.py:43: DeprecationWarning: WARN: The argument mode in render method is deprecated; use
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
Average reward over 100 test episodes: 106.51
```

Actual **codes** @ Expected SARSA

```
import gym
import numpy as np
import cv2
```

```
# Define the discretization parameters
env = gym.make('CartPole-v1')

# Convert continuous state to discrete state
upperBounds = env.observation_space.high
lowerBounds = env.observation_space.low
cartVelocityMin = -3
cartVelocityMax = 3
poleAngleVelocityMin = -10
poleAngleVelocityMax = 10
upperBounds[1] = cartVelocityMax
upperBounds[3] = poleAngleVelocityMax
lowerBounds[1] = cartVelocityMin
lowerBounds[3] = poleAngleVelocityMin

numberOfBinsPosition = 30
numberOfBinsVelocity = 30
numberOfBinsAngle = 30
numberOfBinsAngleVelocity = 30
DISCRET_NBR = [numberOfBinsPosition, numberOfBinsVelocity, numberOfBinsAngle, numberOfBinsAngleVelocity]


def convert_state_discrete(state):
    position = state[0]
    velocity = state[1]
    angle = state[2]
    angularVelocity = state[3]

    cartPositionBin = np.linspace(lowerBounds[0], upperBounds[0], DISCRET_NBR[0])
    cartVelocityBin = np.linspace(lowerBounds[1], upperBounds[1], DISCRET_NBR[1])
    poleAngleBin = np.linspace(lowerBounds[2], upperBounds[2], DISCRET_NBR[2])
    poleAngleVelocityBin = np.linspace(lowerBounds[3], upperBounds[3], DISCRET_NBR[3])

    indexPosition = np.maximum(np.digitize(position, cartPositionBin) - 1, 0)
    indexVelocity = np.maximum(np.digitize(velocity, cartVelocityBin) - 1, 0)
    indexAngle = np.maximum(np.digitize(angle, poleAngleBin) - 1, 0)
    indexAngularVelocity = np.maximum(np.digitize(angularVelocity, poleAngleVelocityBin) - 1, 0)

    return tuple([indexPosition, indexVelocity, indexAngle, indexAngularVelocity])


def generate_Q():
    return np.zeros(tuple(DISCRET_NBR) + (2,))


def expected_sarsa(env, num_episodes, alpha, gamma, epsilon):
    num_actions = env.action_space.n
    num_states = tuple(DISCRET_NBR)

    Q = generate_Q()

    for episode in range(num_episodes):
        observation = env.reset()
        state = convert_state_discrete(observation)
        while True:
            # Choose action using epsilon-greedy policy
            action = epsilon_greedy_policy(Q, state, epsilon)

            # Take action and observe next state and reward
            next_state, reward, done, _ = env.step(action)
            next_state = convert_state_discrete(next_state)

            # Calculate expected Q-value of the next state
            next_action_probs = epsilon_greedy_probs(Q, next_state, epsilon)
            expected_next_q = np.sum(Q[next_state] * next_action_probs)

            # Update Q-value of the current state-action pair
            Q[state][action] += alpha * (reward + gamma * expected_next_q - Q[state][action])

            state = next_state

            if done:
                break
    return Q
```

```
def epsilon_greedy_policy(Q, state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.randint(len(Q[state]))
    else:
        return np.argmax(Q[state])


def epsilon_greedy_probs(Q, state, epsilon):
    num_actions = Q.shape[-1]
    greedy_action = np.argmax(Q[state])
    probs = np.ones(num_actions) * epsilon / num_actions
    probs[greedy_action] += (1.0 - epsilon)
    return probs



# Test the learned policy
def test_policy(env, Q):
    observation = env.reset()
    state = convert_state_discrete(observation)
    done = False
    total_reward = 0

    frames = []  # List to store environment frames

    while not done:
        action = np.argmax(Q[state])
        state, reward, done, _ = env.step(action)
        state = convert_state_discrete(state)
        total_reward += reward

        # Save the rendered frame
        frame = env.render(mode='rgb_array')
        frames.append(frame)

    return total_reward, frames


if __name__ == '__main__':
    # Set hyperparameters
    num_episodes = 5000
    alpha = 0.1  # learning rate
    gamma = 0.99  # discount factor
    epsilon = 0.1  # exploration rate

    # Run Expected SARSA
    Q = expected_sarsa(env, num_episodes, alpha, gamma, epsilon)

    # Test the learned policy
    total_rewards = []
    all_frames = []

    for _ in range(100):
        reward, frames = test_policy(env, Q)
        total_rewards.append(reward)
        all_frames.extend(frames)

    average_reward = np.mean(total_rewards)
    print("Average reward over 100 test episodes:", average_reward)

    env.close()

    # Display the recorded frames as a video
    height, width, _ = all_frames[0].shape
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    video = cv2.VideoWriter('Expected-SARSA.mp4', fourcc, 30.0, (width, height))

    for frame in all_frames:
        video.write(frame)

    video.release()
```

```
 Average reward over 100 test episodes: 86.92
```

✓ 2m 22s completed at 11:32 PM