MINI PROJECT

AI GAME

Reinforcement Learning Game Suite

TEAM LEADER NAME – PRIYANSHI AGRAWAL

ROLL NO – 2301CS90

MEMBER 2 NAME – MANVITHA REDDY

ROLL NO – 2301CS29

MEMBER 3 NAME – SHIKSHA RAGINEE

ROLL NO – 2301AI13

MEMBER 4 NAME – GOMPA VASAVI

ROLL NO – 2301AI43

MEMBER 5 NAME – PREETI KUMARI

ROLL NO – 2301AI17

MEMBER 6 NAME – GURU SAI BHASKARI

ROLL NO – 2301AI19

Main.py

```python
import sys

def cliff_menu():
    print("\nCliff Climbing Game selected.")
    print("Choose algorithm:")
    print("1. SARSA")
    print("2. Q-Learning")
    algo_choice = input("Enter your choice (1 or 2): ").strip()

    if algo_choice == "1":
        import cliff_climbing
        return cliff_climbing.train_sarsa, cliff_climbing.test_sarsa
    elif algo_choice == "2":
        import cliff_climbing
        return cliff_climbing.train_q_learning, cliff_climbing.test_q_learning
    else:
        print("Invalid algorithm choice.")
        sys.exit()

def taxi_menu():
    print("\nTaxi Game selected.")
    import taxi
    return taxi.train_taxi, taxi.test_taxi
def blackjack_menu():
    print("\nBlackjack Game selected.")
    import blackjack
    return blackjack.train_blackjack, blackjack.test_blackjack
def mountaincar_menu():
    print("\nMountainCar Game selected.")
    import mountaincar
    return mountaincar.train_mountaincar, mountaincar.test_mountaincar

def main():
    print("Welcome to the Game Trainer & Tester!")
    print("Select an operation:")
    print("1. Train a game")
    print("2. Test a game")
    operation = input("Enter 1 or 2: ").strip()

    if operation not in ["1", "2"]:
        print("Invalid selection. Exiting.")
        sys.exit()

    print("\nSelect the game:")
    print("1. Cliff Climbing")
    print("2. Taxi")
    print("3. Blackjack")
    print("4. MountainCar")  # <-- add this line

    game_choice = input("Enter game choice (1, 2, 3 or 4): ").strip()

    # game_choice = input("Enter game choice (1, 2 or 3): ").strip()

    # if game_choice == "1":
    #     train_func, test_func = cliff_menu()
    # elif game_choice == "2":
    #     train_func, test_func = taxi_menu()
    # else:
    #     print("Invalid game choice. Exiting.")
    #     sys.exit()
    if game_choice == "1":
        train_func, test_func = cliff_menu()
    elif game_choice == "2":
        train_func, test_func = taxi_menu()
    elif game_choice == "3":
        train_func, test_func = blackjack_menu()
    elif game_choice == "4":
        train_func, test_func = mountaincar_menu()
    else:
        print("Invalid game choice. Exiting.")
        sys.exit()

    if operation == "1":
        print("\nStarting training...")
        train_func()
    else:
        print("\nStarting testing...")
        test_func()

if __name__ == "__main__":
    main()
```

Cliff_climbing.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
import os
from IPython.display import clear_output
import random
import pickle

#pip install gymnasium
import gymnasium as gym

Tabnine | Edit | Test | Explain | Document | Cody
def save_frames_as_gif(frames,episode, algorithm_type, path='./climbing_gif/', filename='gym_animation.gif'):
    #Mess with this to change frame size
    if not os.path.exists(path):
        bs.makedirs(path)

    print(frames[0].shape)
    plt.figure(figsize=(frames[0].shape[1] / 72.0, frames[0].shape[0] / 72.0), dpi=72)

    patch = plt.imshow(frames[0])
    plt.axis('off')
    plt.title(f"Run from episode {episode} {algorithm_type}")

    def animate(i):
        patch.set_data(frames[i])

    anim = animation.FuncAnimation(plt.gcf(), animate, frames = len(frames), interval=50)
    # anim.save(path + filename, writer='imagemagick', fps=30)

    anim.save(os.path.join(path, filename), writer='imagemagick', fps=30)

Tabnine | Edit | Test | Explain | Document | Cody
def train_sarsa():   #SARSA (On-Policy TD Control) for estimating Q
    env = gym.make('CliffWalking-v0',render_mode="rgb_array")
    # env = gym.make('CliffWalking-v0', render_mode="rgb_array", new_step_api=True)

    observation,info = env.reset()
    #print(env.action_space.n)
    #print(env.observation_space)
    action = env.action_space.sample()
    # initialize constants
    EPSILON = 0.01
    ALPHA = 0.50
    RENDER_AT_EPISODE = 10
    EPISODES = 100
    LEARNING_RATE = 0.99

    #initialize for each state and action
    episode_frames_SARSA = {}
    previous_state_action_reward = {}
    policy_given_state = {}
    Q_table = np.random.rand(env.observation_space.n,env.action_space.n) #THIS NEEDS TO BE A TABLE FOR EACH STATE AND ACTION
    Q_table[47] = np.zeros((1,env.action_space.n))

    #initialize the policy
    for interaction in range(Q_table.shape[0]):
            best_action = np.argmax(Q_table[interaction])
            policy_given_state[interaction] = np.ones((1,env.action_space.n))[0] * EPSILON/env.action_space.n
            policy_given_state[interaction][best_action] = 1-EPSILON+(EPSILON/env.action_space.n)
    #SANITY CHECK
    for row in policy_given_state.keys():
        #print(policy_given_state[row])
        np.random.choice([0,1,2,3],p=policy_given_state[row])
    time_step=0
    C = 0
    number_time_step = []
    for episode in range(EPISODES): #for each episode
        if episode%10_000 == 0:
            print(f"Currently on episode {episode}")
        frames=[]
        observation,info = env.reset() #initialize the environment
        #This is the epsilon greedy method
        if random.random() < EPSILON:
            action = random.randint(0,env.action_space.n-1)
        else:
            action = np.argmax(Q_table[observation])
        done=False
        while not done: #for each time step
            next_observation, reward, done, truncated,info = env.step(action) #take an action and get feedback from the environment

            if random.random() < EPSILON:
                next_action = random.randint(0,env.action_space.n-1)
            else:
                next_action = np.argmax(Q_table[next_observation])

            Q_table[observation][action] = Q_table[observation][action] + ALPHA*(reward + LEARNING_RATE*Q_table[next_observation][next_action] - Q_table[observation][action])
            observation = next_observation
            action = next_action
            time_step+=1
            if episode%RENDER_AT_EPISODE == 0:
                clear_output(wait=True)
                single_frame = env.render()
                frames.append(single_frame)
```

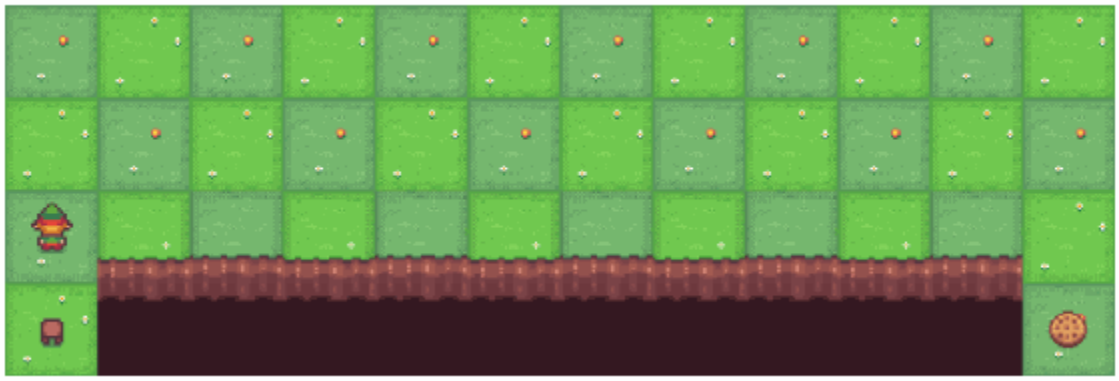
```python
        if episode%RENDER_AT_EPISODE ==0:
            frames.append(single_frame)
            frames.append(single_frame)
            frames.append(single_frame)
            frames.append(single_ (variable) episode: int
            episode_frames_SARSA[episode] = frames
            save_frames_as_gif(frames,episode,"SARSA",filename=f"SARSA episode {episode}.gif")

        number_time_step.append(time_step)
         # After training is complete
        with open('sarsa_q_table.pkl', 'wb') as f:
            pickle.dump(Q_table, f)

Tabnine | Edit | Test | Explain | Document | Cody
def test_sarsa():   # Final test for SARSA agent
    with open('sarsa_q_table.pkl', 'rb') as f:
       Q_table = pickle.load(f)
    env = gym.make('CliffWalking-v0', render_mode="rgb_array")
    observation, info = env.reset()
    done = False
    frames = []

    while not done:
        action = np.argmax(Q_table[observation])  # Use the trained SARSA Q-table
        observation, reward, done, truncated, info = env.step(action)
        frames.append(env.render())

    save_frames_as_gif(frames, episode="Final", algorithm_type="SARSA", filename="SARSA Final Test.gif")

Tabnine | Edit | Test | Explain | Document | Cody
def train_q_learning():    # THIS IS FOR Q-LEARNING
    env = gym.make('CliffWalking-v0',render_mode="rgb_array")

    ALPHA = 0.85
    EPSILON = 0.1
    RENDER_AT_EPISODE = 10
    EPISODES = 100
    LEARNING_RATE = 0.99
    alpha = np.linspace(0.01,0.99,10)
    learning_rate= np.linspace(0.01,0.99,10)

    episode_frames_Q = {}
    time_step=0
    sum_of_rewards = {}

    Q_table = np.zeros((env.observation_space.n,env.action_space.n)) #THIS NEEDS TO BE A TABLE FOR EACH STATE AND ACTION
    for episode in range(EPISODES): #for each episode
        observation,info = env.reset() #initialize the environment
        done=False
        rewards = 0
        frames=[]
        while not done: #for each time step
            # SELECT AN ACTION USING EPSILON GREEDY METHOD
            #if random.random() < EPSILON:
            #    action = random.randint(0,env.action_space.n-1)
            #else:
            action = np.argmax(Q_table[observation])

            next_observation, reward, done, truncated,info = env.step(action) #take an action and get feedback from the environment
            Q_table[observation][action] = Q_table[observation][action] + ALPHA*(reward + LEARNING_RATE*np.max(Q_table[next_observation]) - Q_table[observation][action])
            observation = next_observation
            time_step+=1
            rewards+=reward
            if episode%RENDER_AT_EPISODE == 0:
                clear_output(wait=True)
                single_frame = env.render()
                frames.append(single_frame)
                plt.imshow(single_frame)
                plt.show()
        sum_of_rewards[episode] = rewards
        if episode%RENDER_AT_EPISODE == 0:
            frames.append(single_frame)
            frames.append(single_frame)
            episode_frames_Q[episode] = frames
            save_frames_as_gif(frames,episode,"Q-Learning",filename=f"Q-Learning episode {episode}.gif")
    with open('sarsa_q_table.pkl', 'wb') as f:
        pickle.dump(Q_table, f)

Tabnine | Edit | Test | Explain | Document | Cody
def test_q_learning()   : # Final test for Q-Learning agent
    with open('sarsa_q_table.pkl', 'rb') as f:
       Q_table = pickle.load(f)
    env = gym.make('CliffWalking-v0', render_mode="rgb_array")
    observation, info = env.reset()
    done = False
    frames = []

    while not done:
        action = np.argmax(Q_table[observation])  # Use the trained Q-Learning Q-table
        observation, reward, done, truncated, info = env.step(action)
        frames.append(env.render())

    save_frames_as_gif(frames, episode="Final", algorithm_type="Q-Learning", filename="Q-Learning Final Test.gif")
```

GAME 1 – cliff climbing—sarsa algorithm

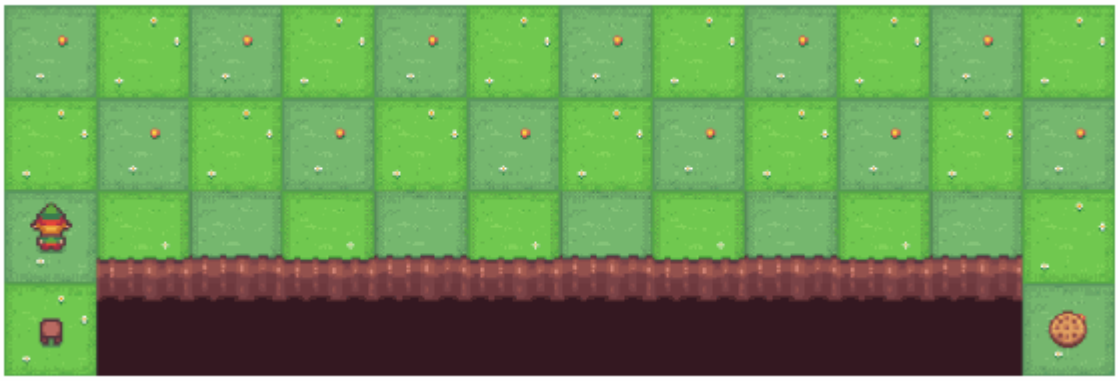
```
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 1

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 1

Cliff Climbing Game selected.
Choose algorithm:
1. SARSA
2. Q-Learning
Enter your choice (1 or 2): 1

Starting training...
Currently on episode 0
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 2

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 1

Cliff Climbing Game selected.
Choose algorithm:
1. SARSA
2. Q-Learning
Enter your choice (1 or 2): 1

Starting testing...
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
PS D:\AI game\aigame>
```

Result --

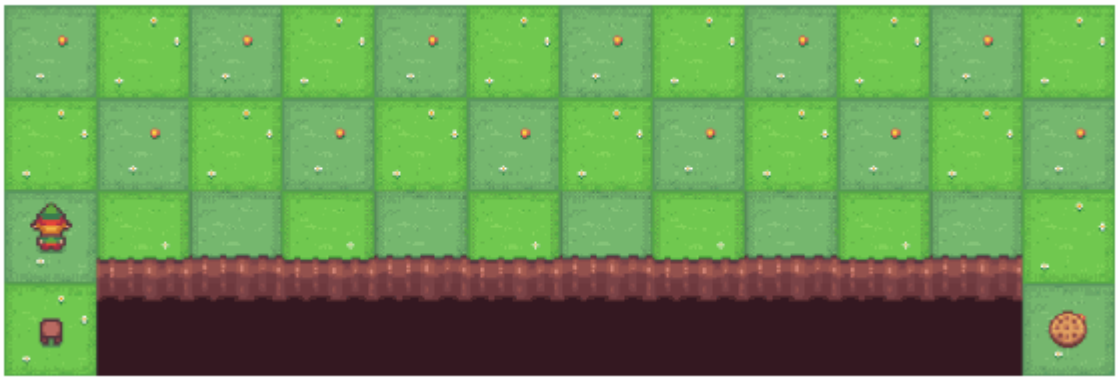
Run from episode 70 SARSA



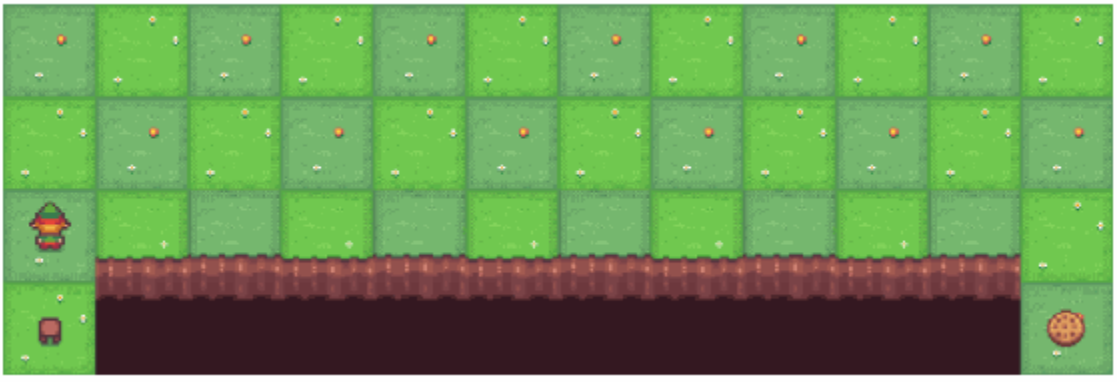
Run from episode 80 SARSA



Run from episode 90 SARSA



Run from episode Final SARSA

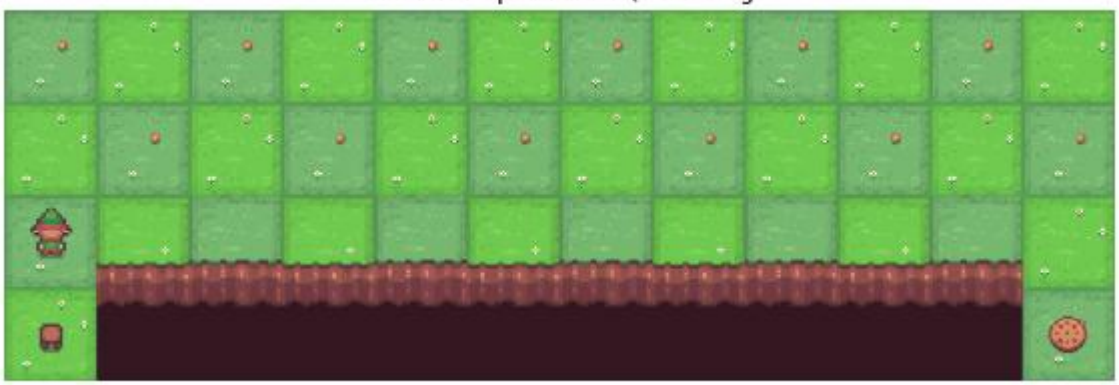


GA E 1 – cliff climbing—Q learning algorithm

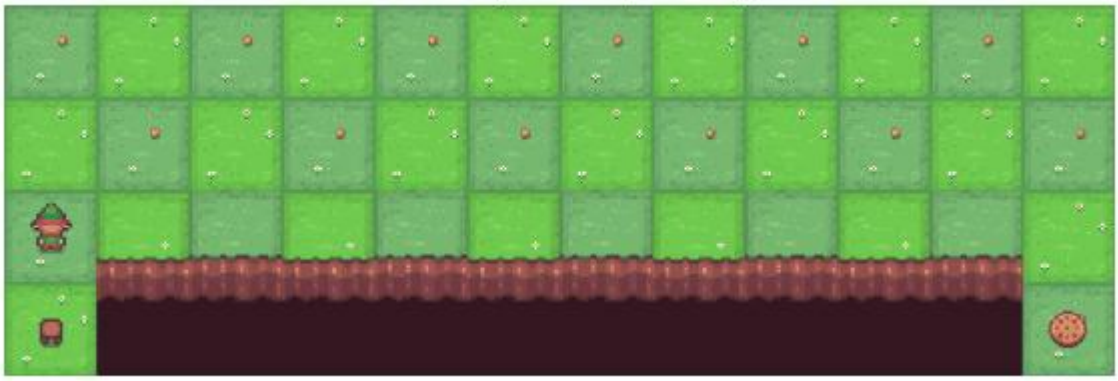
```
NameError: name 'matplotlib' is not defined
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 1

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 1

Cliff Climbing Game selected.
Choose algorithm:
1. SARSA
2. Q-Learning
Enter your choice (1 or 2): 2

Starting training...
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
D:\AI game\aigame\cliff_climbing.py:175: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 2

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 1

Cliff Climbing Game selected.
Choose algorithm:
1. SARSA
2. Q-Learning
Enter your choice (1 or 2): 2

Starting testing...
(240, 720, 3)
MovieWriter imagemagick unavailable; using Pillow instead.
```

RESULT—

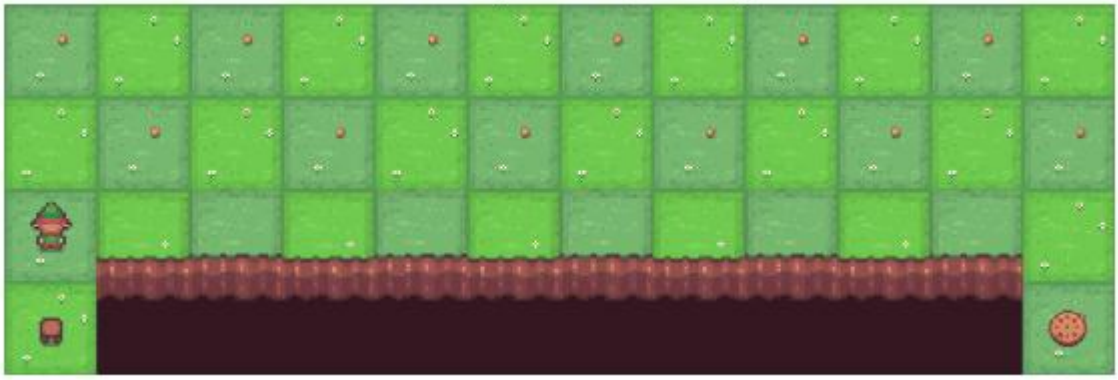
Run from episode 70 Q-Learning



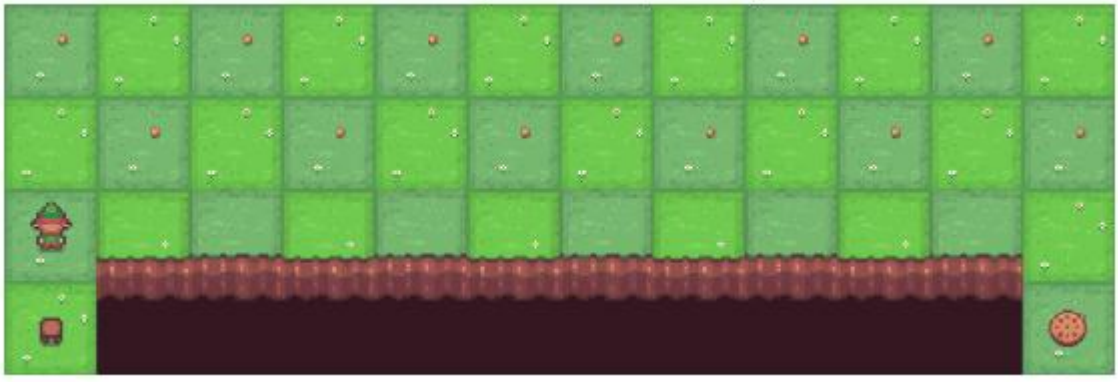
Run from episode 80 Q-Learning



Run from episode 90 Q-Learning



Run from episode Final Q-Learning



taxi.py-

```python
from matplotlib import animation
import os
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import pickle
Tabnine | Edit | Test | Explain | Document | Cody
def save_frames_as_gif(frames, episode, filename='taxi_test.gif', path='./taxi_gif/'):
    if not os.path.exists(path):
        os.makedirs(path)

    plt.figure(figsize=(frames[0].shape[1]/72.0, frames[0].shape[0]/72.0), dpi=72)
    patch = plt.imshow(frames[0])
    plt.axis('off')

    def animate(i):
        patch.set_data(frames[i])

    anim = animation.FuncAnimation(plt.gcf(), animate, frames=len(frames), interval=200)
    anim.save(os.path.join(path, filename), writer='pillow')
    plt.close()
```

```python
def train_taxi(episodes=15000):
    import gymnasium as gym
    import numpy as np
    import pickle

    env = gym.make('Taxi-v3')
    q = np.zeros((env.observation_space.n, env.action_space.n))

    learning_rate_a = 0.9
    discount_factor_g = 0.9
    epsilon = 1.0
    epsilon_decay_rate = 0.0001
    rng = np.random.default_rng()

    for i in range(episodes):
        state = env.reset()[0]
        terminated = False
        truncated = False

        while not terminated and not truncated:
            if rng.random() < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q[state, :])

            new_state, reward, terminated, truncated, _ = env.step(action)
            q[state, action] = q[state, action] + learning_rate_a * (
                reward + discount_factor_g * np.max(q[new_state, :]) - q[state, action]
            )
            state = new_state

        epsilon = max(epsilon - epsilon_decay_rate, 0)
        if epsilon == 0:
            learning_rate_a = 0.0001

    with open("taxi.pkl", "wb") as f:
        pickle.dump(q, f)

    env.close()
```

Tabnine | Edit | Test | Fix | Explain | Document | Cody

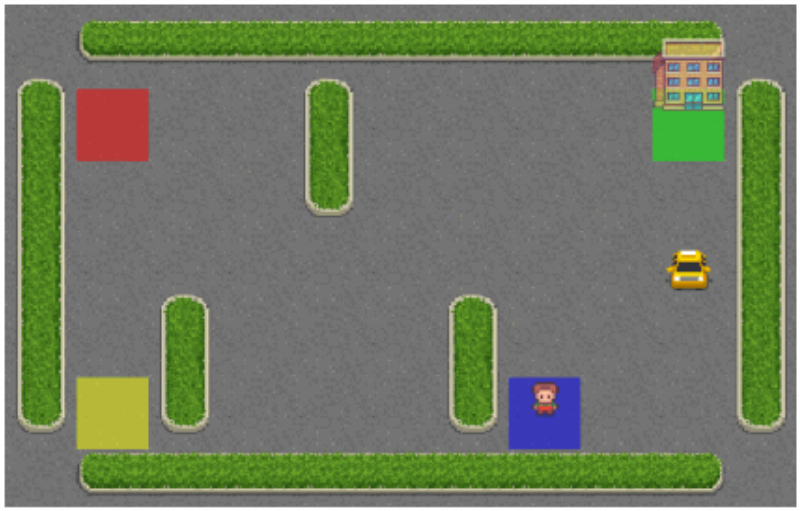
```python
def test_taxi(episodes=5):
    import gymnasium as gym
    import pickle
    import numpy as np

    env = gym.make('Taxi-v3', render_mode='rgb_array')

    with open("taxi.pkl", "rb") as f:
        q = pickle.load(f)

    for i in range(episodes):
        state = env.reset()[0]
        terminated = False
        truncated = False
        frames = []

        while not terminated and not truncated:
            action = np.argmax(q[state, :])
            state, reward, terminated, truncated, _ = env.step(action)
            frames.append(env.render())

        save_frames_as_gif(frames, i, filename=f"taxi_episode_{i}.gif")

    env.close()
```

GAME 2 – TAXI-

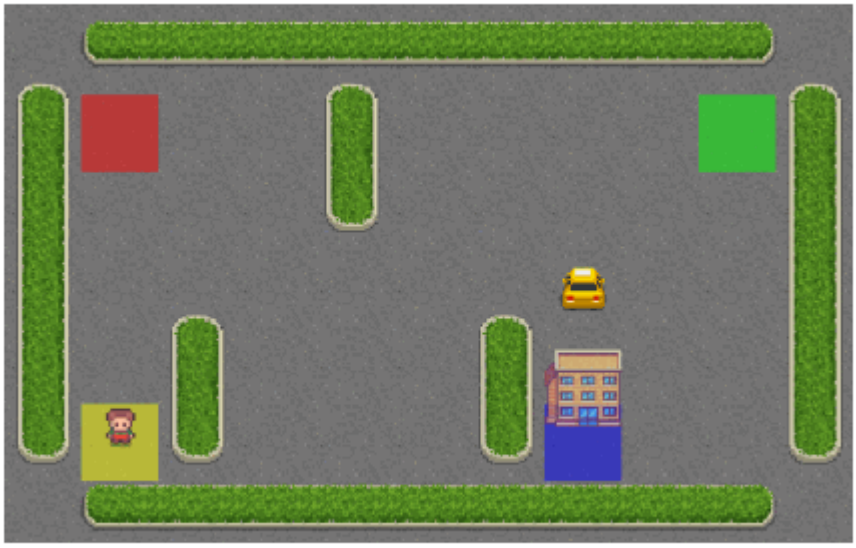
```
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 1

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 2

Taxi Game selected.

Starting training...
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 2

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 2

Taxi Game selected.

Starting testing...
```

RESULT--

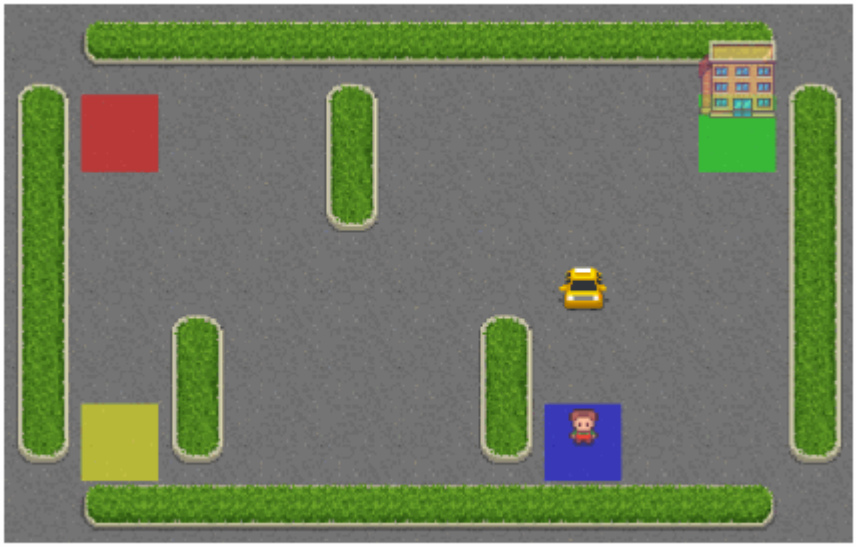
EPISODE = 2



EPISODE=3



EPISODE=4



blackjack.py—

```python
import matplotlib.pyplot as plt
from matplotlib import animation
from matplotlib.animation import PillowWriter
from collections import defaultdict
import numpy as np
from tqdm import tqdm
import gymnasium as gym
import pickle

env = gym.make("Blackjack-v1", sab=True)
Tabnine | Edit | Test | Explain | Document | Cody
def default_q_values():
    return np.zeros(env.action_space.n)
class BlackjackAgent:
    Tabnine | Edit | Test | Explain | Document | Cody
    def __init__(self, learning_rate, initial_epsilon, epsilon_decay, final_epsilon, discount_factor=0.95):
        # self.q_values = defaultdict(lambda: np.zeros(env.action_space.n))
        self.q_values = defaultdict(default_q_values)
        self.lr = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon
        self.training_error = []

    Tabnine | Edit | Test | Explain | Document | Cody
    def get_action(self, obs):
        if np.random.random() < self.epsilon:
            return env.action_space.sample()
        return int(np.argmax(self.q_values[obs]))

    Tabnine | Edit | Test | Explain | Document | Cody
    def update(self, obs, action, reward, terminated, next_obs):
        future_q_value = (not terminated) * np.max(self.q_values[next_obs])
        td = reward + self.discount_factor * future_q_value - self.q_values[obs][action]
        self.q_values[obs][action] += self.lr * td
        self.training_error.append(td)

    Tabnine | Edit | Test | Explain | Document | Cody
    def decay_epsilon(self):
        self.epsilon = max(self.final_epsilon, self.epsilon - self.epsilon_decay)


Tabnine | Edit | Test | Explain | Document | Cody
def train_blackjack(n_episodes=1000000):
    global agent
    agent = BlackjackAgent(
        learning_rate=0.1,
        initial_epsilon=1.0,
        epsilon_decay=1.0 / (n_episodes / 2),
        final_epsilon=0.1,
    )
    env = gym.make("Blackjack-v1", sab=True)
    env = gym.wrappers.RecordEpisodeStatistics(env, n_episodes)
    for episode in tqdm(range(n_episodes)):
        obs, info = env.reset()
        done = False
        while not done:
            action = agent.get_action(obs)
            next_obs, reward, terminated, truncated, info = env.step(action)
            agent.update(obs, action, reward, terminated, next_obs)
            done = terminated or truncated
            obs = next_obs
        agent.decay_epsilon()
    with open("blackjack_agent.pkl", "wb") as f:
        pickle.dump(agent, f)
    print("Training complete and agent saved.")


Tabnine | Edit | Test | Explain | Document | Cody
def test_blackjack_one_episode():
    try:
        with open("blackjack_agent.pkl", "rb") as f:
            agent = pickle.load(f)
    except FileNotFoundError:
        print("No trained agent found.")
        return

    agent.epsilon = 0.0
    env = gym.make("Blackjack-v1", sab=True, render_mode="rgb_array")
    obs, info = env.reset()
    done = False
    frames = []

    while not done:
        frames.append(env.render())
        action = agent.get_action(obs)
        obs, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated

    print("Final Reward:", reward)
    plt.imshow(frames[-1])
    plt.axis("off")
    plt.title(f"Final Reward: {reward}")
    plt.show()

    fig = plt.figure(figsize=(6, 4))
    img = plt.imshow(frames[0])
    plt.axis("off")

    def animate(i):
        img.set_data(frames[i])
        return [img]
```

```
    anim = animation.FuncAnimation(fig, animate, frames=len(frames), interval=500, blit=True)
    anim.save("blackjack_agent.gif", writer=PillowWriter(fps=2))
    print("Animation saved as blackjack_agent.gif")


Tabnine | Edit | Test | Explain | Document | Cody
def test_blackjack_multiple_episodes(num_test_episodes=5):
    try:
        with open("blackjack_agent.pkl", "rb") as f:
            agent = pickle.load(f)
    except FileNotFoundError:
        print("No trained agent found.")
        return

    agent.epsilon = 0.0
    for episode_num in range(1, num_test_episodes + 1):
        env = gym.make("Blackjack-v1", sab=True, render_mode="rgb_array")
        obs, info = env.reset()
        done = False
        frames = []
        print(f"\n--- Test Episode {episode_num} ---")
        print("Initial Observation:", obs)
        while not done:
            frames.append(env.render())
            action = agent.get_action(obs)
            print(f"Agent action: {'Hit' if action == 1 else 'Stick'}, Obs: {obs}")
            obs, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated
        print("Final Reward:", reward)

        fig = plt.figure(figsize=(6, 4))
        img = plt.imshow(frames[0])
        plt.axis("off")

        def animate(i):
            img.set_data(frames[i])
            return [img]

        anim = animation.FuncAnimation(fig, animate, frames=len(frames), interval=500, blit=True)
        anim.save(f"blackjack_ep{episode_num}.gif", writer=PillowWriter(fps=2))
        print(f"Saved GIF as blackjack_ep{episode_num}.gif")


Tabnine | Edit | Test | Explain | Document | Cody
def test_blackjack_statistics(n=1000):
    try:
        with open("blackjack_agent.pkl", "rb") as f:
            agent = pickle.load(f)
    except FileNotFoundError:
        print("No trained agent found.")
        return

    agent.epsilon = 0.0
    env = gym.make("Blackjack-v1", sab=True)
    wins, draws, losses = 0, 0, 0

    for _ in range(n):
        obs, info = env.reset()
        done = False
        while not done:
            action = agent.get_action(obs)
            obs, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated
        if reward == 1.0:
            wins += 1
        elif reward == 0.0:
            draws += 1
        else:
            losses += 1

    print(f"Wins: {wins}, Draws: {draws}, Losses: {losses}")

Tabnine | Edit | Test | Explain | Document | Cody
def test_blackjack():
    print("\nBlackjack Testing Options:")
    print("1. Run and visualize one episode")
    print("2. Run and visualize multiple episodes")
    print("3. Evaluate statistics over 1000 episodes")
    choice = input("Enter choice (1, 2, or 3): ").strip()

    if choice == "1":
        test_blackjack_one_episode()
    elif choice == "2":
        test_blackjack_multiple_episodes()
    elif choice == "3":
        test_blackjack_statistics()
    else:
        print("Invalid choice.")
```

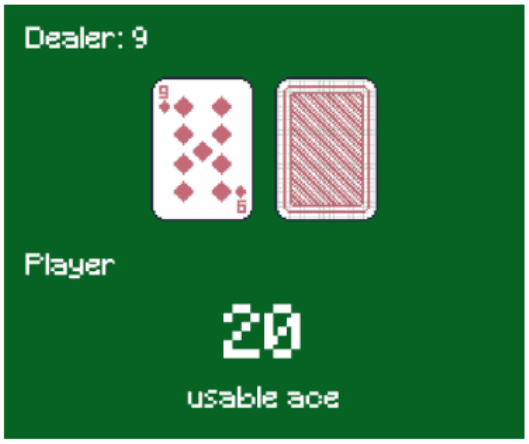
GAME 3-- BLACKJACK –

```
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 1

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 3

Blackjack Game selected.

Starting training...
100%|████████████████████████████████████████████████████████████| 1000000/1000000 [03:14<00:00, 5154.01it/s]
Training complete and agent saved.
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 2

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 3

Blackjack Game selected.

Starting testing...

Blackjack Testing Options:
1. Run and visualize one episode
2. Run and visualize multiple episodes
3. Evaluate statistics over 1000 episodes
Enter choice (1, 2, or 3): 1
Final Reward: 1.0
Animation saved as blackjack_agent.gif
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 2

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 3
```

```
Blackjack Game selected.

Starting testing...

Blackjack Testing Options:
1. Run and visualize one episode
2. Run and visualize multiple episodes
3. Evaluate statistics over 1000 episodes
Enter choice (1, 2, or 3): 2

--- Test Episode 1 ---
Initial Observation: (21, 10, 1)
Agent action: Stick, Obs: (21, 10, 1)
Final Reward: 1.0
Saved GIF as blackjack_ep1.gif

--- Test Episode 2 ---
Initial Observation: (7, 6, 0)
Agent action: Hit, Obs: (7, 6, 0)
Agent action: Hit, Obs: (11, 6, 0)
Agent action: Hit, Obs: (12, 6, 0)
Final Reward: -1.0
Saved GIF as blackjack_ep2.gif

--- Test Episode 3 ---
Initial Observation: (11, 8, 0)
Agent action: Hit, Obs: (11, 8, 0)
Agent action: Hit, Obs: (12, 8, 0)
Agent action: Stick, Obs: (21, 8, 0)
Final Reward: 1.0
Saved GIF as blackjack_ep3.gif

--- Test Episode 4 ---
Initial Observation: (11, 4, 0)
Agent action: Hit, Obs: (11, 4, 0)
Agent action: Stick, Obs: (21, 4, 0)
Final Reward: 1.0
Saved GIF as blackjack_ep4.gif

--- Test Episode 5 ---
Initial Observation: (16, 1, 0)
Agent action: Hit, Obs: (16, 1, 0)
Final Reward: -1.0
Saved GIF as blackjack_ep5.gif
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 2

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 3
```

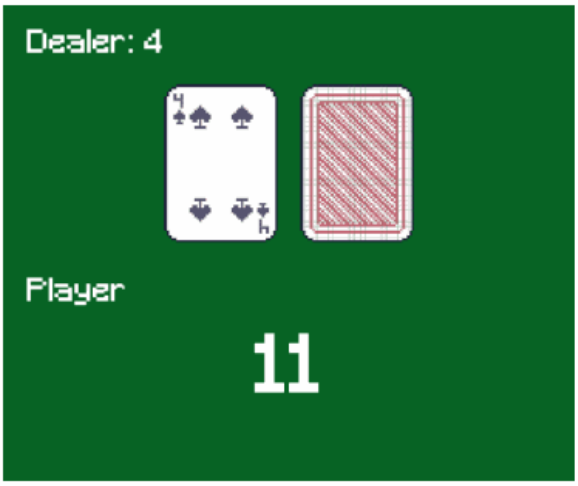
```
Blackjack Game selected.

Starting testing...

Blackjack Testing Options:
1. Run and visualize one episode
2. Run and visualize multiple episodes
3. Evaluate statistics over 1000 episodes
Enter choice (1, 2, or 3): 3
Wins: 432, Draws: 71, Losses: 497
```

RESULT-- FOR 1 EPISODE

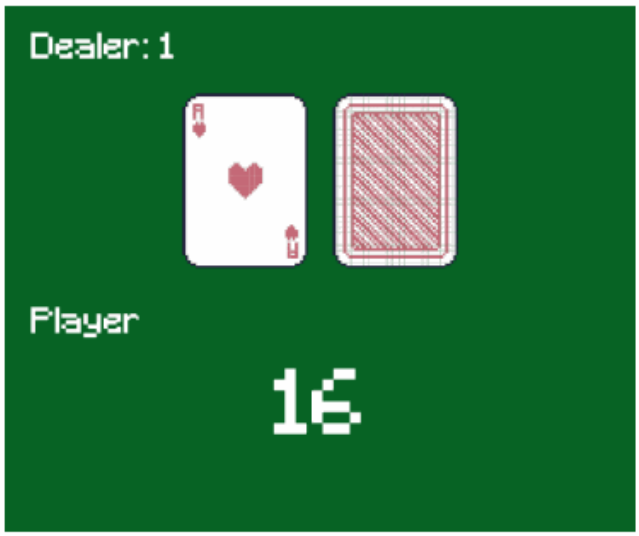


FOR MORE EPISODE—

EPISODE 4



EPISODE 5

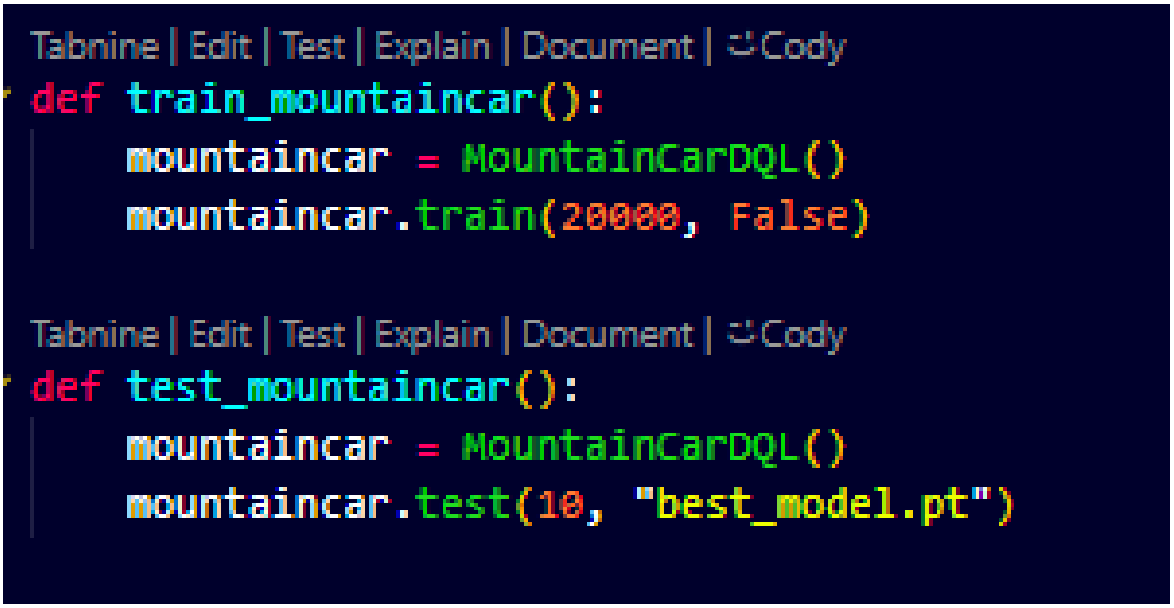


mountaincar.py---

```python
from matplotlib import animation
import os
import matplotlib.pyplot as plt
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
import random
import torch
from torch import nn
import torch.nn.functional as F
Tabnine | Edit | Test | Explain | Document | ↻Cody
def save_frames_as_gif(frames, filename='mountaincar_test.gif', path='./mountaincar_gifs/'):
    if not os.path.exists(path):
        os.makedirs(path)

    plt.figure(figsize=(frames[0].shape[1]/72.0, frames[0].shape[0]/72.0), dpi=72)
    patch = plt.imshow(frames[0])
    plt.axis('off')

    def animate(i):
        patch.set_data(frames[i])

    anim = animation.FuncAnimation(plt.gcf(), animate, frames=len(frames), interval=50)
    anim.save(os.path.join(path, filename), writer='pillow')
    plt.close()

# Define model
class DQN(nn.Module):
    Tabnine | Edit | Test | Explain | Document | ↻Cody
    def __init__(self, in_states, h1_nodes, out_actions):
        super().__init__()

        # Define network layers
        self.fc1 = nn.Linear(in_states, h1_nodes)   # first fully connected layer
        self.out = nn.Linear(h1_nodes, out_actions) # ouptut layer w

    Tabnine | Edit | Test | Explain | Document | ↻Cody
    def forward(self, x):
        x = F.relu(self.fc1(x)) # Apply rectified linear unit (ReLU) activation
        x = self.out(x)         # Calculate output
        return x

# Define memory for Experience Replay
class ReplayMemory():
    Tabnine | Edit | Test | Explain | Document | ↻Cody
    def __init__(self, maxlen):
        self.memory = deque([], maxlen=maxlen)

    Tabnine | Edit | Test | Explain | Document | ↻Cody
    def append(self, transition):
        self.memory.append(transition)

    Tabnine | Edit | Test | Explain | Document | ↻Cody
    def sample(self, sample_size):
        return random.sample(self.memory, sample_size)

    Tabnine | Edit | Test | Explain | Document | ↻Cody
    def __len__(self):
        return len(self.memory)

# MountainCar Deep Q-Learning
class MountainCarDQL():
    # Hyperparameters (adjustable)
    learning_rate_a = 0.01         # learning rate (alpha)
    discount_factor_g = 0.9         # discount rate (gamma)
    network_sync_rate = 50000          # number of steps the agent takes before syncing the policy and target network
    replay_memory_size = 100000       # size of replay memory
    mini_batch_size = 32            # size of the training data set sampled from the replay memory

    num_divisions = 20

    # Neural Network
    loss_fn = nn.MSELoss()          # NN Loss function. MSE=Mean Squared Error can be swapped to something else.
    optimizer = None                # NN Optimizer. Initialize later.


    # Train the environment
    Tabnine | Edit | Test | Explain | Document | ↻Cody
    def train(self, episodes, render=False):
        # Create FrozenLake instance
        env = gym.make('MountainCar-v0', render_mode='human' if render else None)
        num_states = env.observation_space.shape[0] # expecting 2: position & velocity
        num_actions = env.action_space.n

        # Divide position and velocity into segments
        self.pos_space = np.linspace(env.observation_space.low[0], env.observation_space.high[0], self.num_divisions)    # Between -1.2 and 0.6
        self.vel_space = np.linspace(env.observation_space.low[1], env.observation_space.high[1], self.num_divisions)    # Between -0.07 and 0.07

        epsilon = 1 # 1 = 100% random actions
        memory = ReplayMemory(self.replay_memory_size)

        # Create policy and target network. Number of nodes in the hidden layer can be adjusted.
        policy_dqn = DQN(in_states=num_states, h1_nodes=10, out_actions=num_actions)
        target_dqn = DQN(in_states=num_states, h1_nodes=10, out_actions=num_actions)

        # Make the target and policy networks the same (copy weights/biases from one network to the other)
        target_dqn.load_state_dict(policy_dqn.state_dict())

        # Policy network optimizer. "Adam" optimizer can be swapped to something else.
        self.optimizer = torch.optim.Adam(policy_dqn.parameters(), lr=self.learning_rate_a)
```

```python
        # List to keep track of rewards collected per episode. Initialize list to 0's.
        rewards_per_episode = []

        # List to keep track of epsilon decay
        epsilon_history = []

        # Track number of steps taken. Used for syncing policy => target network.
        step_count=0
        goal_reached=False
        best_rewards=-200

        for i in range(episodes):
            state = env.reset()[0]  # Initialize to state 0
            terminated = False      # True when agent falls in hole or reached goal

            rewards = 0

            # Agent navigates map until it falls into hole/reaches goal (terminated), or has taken 200 actions (truncated).
            while(not terminated and rewards>-1000):

                # Select action based on epsilon-greedy
                if random.random() < epsilon:
                    # select random action
                    action = env.action_space.sample() # actions: 0=left,1=idle,2=right
                else:
                    # select best action
                    with torch.no_grad():
                        action = policy_dqn(self.state_to_dqn_input(state)).argmax().item()

                # Execute action
                new_state,reward,terminated,truncated,_ = env.step(action)

                # Accumulate reward
                rewards += reward

                # Save experience into memory
                memory.append((state, action, new_state, reward, terminated))

                # Move to the next state
                state = new_state

                # Increment step counter
                step_count+=1

            # Keep track of the rewards collected per episode.
            rewards_per_episode.append(rewards)
            if(terminated):
                goal_reached = True

            # Graph training progress
            if(i!=0 and i%1000==0):
                print(f'Episode {i} Epsilon {epsilon}')

                # self.plot_progress(rewards_per_episode, epsilon_history)

            if rewards>best_rewards:
                best_rewards = rewards
                print(f'Best rewards so far: {best_rewards}')
                # Save policy
                torch.save(policy_dqn.state_dict(), f"mountaincar_dql_{i}.pt")
                torch.save(policy_dqn.state_dict(), "best_model.pt")

            # Check if enough experience has been collected
            if len(memory)>self.mini_batch_size and goal_reached:
                mini_batch = memory.sample(self.mini_batch_size)
                self.optimize(mini_batch, policy_dqn, target_dqn)

                # Decay epsilon
                epsilon = max(epsilon - 1/episodes, 0)
                epsilon_history.append(epsilon)

                # Copy policy network to target network after a certain number of steps
                if step_count > self.network_sync_rate:
                    target_dqn.load_state_dict(policy_dqn.state_dict())
                    step_count=0

        # Close environment
        env.close()
```

```python
    Tabnine | Edit | Test | Explain | Document | Cody
    def optimize(self, mini_batch, policy_dqn, target_dqn):

        current_q_list = []
        target_q_list = []

        for state, action, new_state, reward, terminated in mini_batch:

            if terminated:
                # Agent receive reward of 0 for reaching goal.
                # When in a terminated state, target q value should be set to the reward.
                target = torch.FloatTensor([reward])
            else:
                # Calculate target q value
                with torch.no_grad():
                    target = torch.FloatTensor(
                        reward + self.discount_factor_g * target_dqn(self.state_to_dqn_input(new_state)).max()
                    )

            # Get the current set of Q values
            current_q = policy_dqn(self.state_to_dqn_input(state))
            current_q_list.append(current_q)

            # Get the target set of Q values
            target_q = target_dqn(self.state_to_dqn_input(state))
            # Adjust the specific action to the target that was just calculated
            target_q[action] = target
            target_q_list.append(target_q)

        # Compute loss for the whole minibatch
        loss = self.loss_fn(torch.stack(current_q_list), torch.stack(target_q_list))

        # Optimize the model
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    Cody
    '''
    Converts a state (position, velocity) to tensor representation.
    Example:
    Input = (0.3, -0.03)
    Return = tensor([16, 6])
    '''
    Tabnine | Edit | Test | Explain | Document | Cody
    def state_to_dqn_input(self, state)->torch.Tensor:
        state_p = np.digitize(state[0], self.pos_space)
        state_v = np.digitize(state[1], self.vel_space)

        return torch.FloatTensor([state_p, state_v])

    # Run the environment with the learned policy
    Tabnine | Edit | Test | Explain | Document | Cody
    def test(self, episodes, model_filepath):
        # Create FrozenLake instance
        # env = gym.make('MountainCar-v0', render_mode='human')
        env = gym.make('MountainCar-v0', render_mode='rgb_array')
        num_states = env.observation_space.shape[0]
        num_actions = env.action_space.n

        self.pos_space = np.linspace(env.observation_space.low[0], env.observation_space.high[0], self.num_divisions)    # Between -1.2 and 0.6
        self.vel_space = np.linspace(env.observation_space.low[1], env.observation_space.high[1], self.num_divisions)    # Between -0.07 and 0.07

        # Load learned policy
        policy_dqn = DQN(in_states=num_states, h1_nodes=10, out_actions=num_actions)
        # policy_dqn.load_state_dict(torch.load(model_filepath))
        policy_dqn.load_state_dict(torch.load(model_filepath, weights_only=True))
        policy_dqn.eval()    # switch model to evaluation mode
          # Add this before the loop
        for i in range(episodes):
            frames = []
            state = env.reset()[0]  # Initialize to state 0
            terminated = False      # True when agent falls in hole or reached goal
            truncated = False       # True when agent takes more than 200 actions

            # Agent navigates map until it falls into a hole (terminated), reaches goal (terminated), or has taken 200 actions (truncated).
            while(not terminated and not truncated):
                # Select best action
                frame = env.render()
                frames.append(frame)

                # if hasattr(env, 'render'):
                #      frame = env.render()  # Get a rendered frame
                #      if frame is not None:
                #         frames.append(frame)
                with torch.no_grad():
                    action = policy_dqn(self.state_to_dqn_input(state)).argmax().item()

                # Execute action
                state,reward,terminated,truncated,_ = env.step(action)
            if len(frames) > 0:
                save_frames_as_gif(frames, filename=f"mountaincar_test_episode_{i}.gif")

        env.close()
```

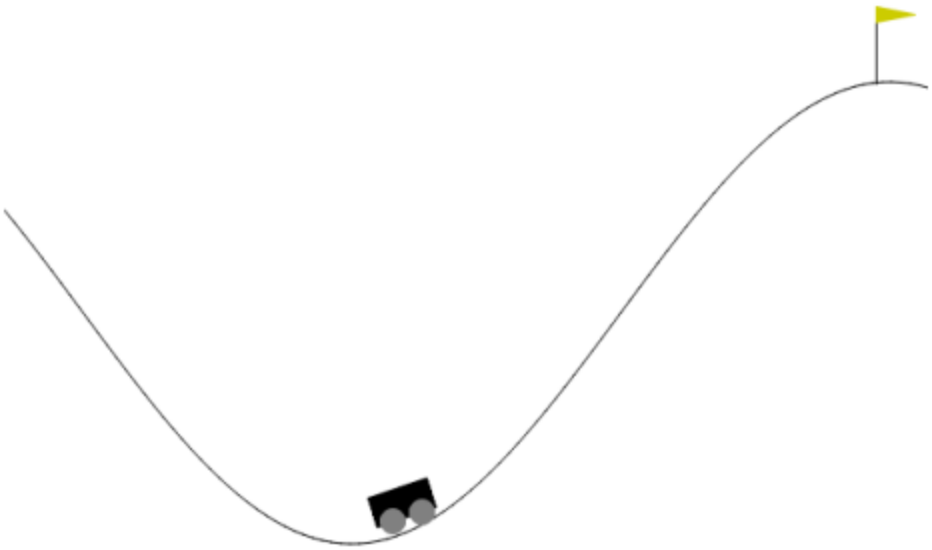

Tabnine | Edit | Test | Explain | Document | Cody
def train_mountaincar():
mountaincar = MountainCarDQL()
mountaincar.train(20000, False)
Tabnine | Edit | Test | Explain | Document | Cody
def test_mountaincar():
mountaincar = MountainCarDQL()
mountaincar.test(10, "best_model.pt")

GAME 3-- MOUNTAIN CAR–

```
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 1

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 4

MountainCar Game selected.

Starting training...
Episode 1000 Epsilon 1
Episode 2000 Epsilon 1
Episode 3000 Epsilon 0.9586000000000046
Episode 4000 Epsilon 0.9086000000000101
Episode 5000 Epsilon 0.8586000000000156
Episode 6000 Epsilon 0.8086000000000211
Episode 7000 Epsilon 0.7586000000000266
Episode 8000 Epsilon 0.7086000000000321
Episode 9000 Epsilon 0.6586000000000376
Episode 10000 Epsilon 0.6086000000000431
Episode 11000 Epsilon 0.5586000000000486
Best rewards so far: -194.0
Best rewards so far: -176.0
Episode 12000 Epsilon 0.5086000000000541
Best rewards so far: -175.0
Best rewards so far: -169.0
Episode 13000 Epsilon 0.4586000000000596
Episode 14000 Epsilon 0.40860000000006513
Best rewards so far: -164.0
Best rewards so far: -162.0
Episode 15000 Epsilon 0.35860000000007064
Episode 16000 Epsilon 0.30860000000007615
Best rewards so far: -144.0
Episode 17000 Epsilon 0.25860000000008165
Episode 18000 Epsilon 0.20860000000008716
Episode 19000 Epsilon 0.15860000000009267
PS D:\AI game\aigame> python main.py
Welcome to the Game Trainer & Tester!
Select an operation:
1. Train a game
2. Test a game
Enter 1 or 2: 2

Select the game:
1. Cliff Climbing
2. Taxi
3. Blackjack
4. MountainCar
Enter game choice (1, 2, 3 or 4): 4

MountainCar Game selected.

Starting testing...
PS D:\AI game\aigame>
```

RESULT-

EPOSODE -8



EPISODE - 9