```
import os
os.environ["KERAS_BACKEND"] = "tensorflow"

import keras
from keras import layers
import tensorflow as tf
import gymnasium as gym
import numpy as np
import random
from collections import deque
import matplotlib.pyplot as plt
```

```
# Create the CartPole-v1 environment
env = gym.make("CartPole-v1", render_mode="human")

num_states = env.observation_space.shape[0]
print("Size of State Space ->  {}".format(num_states))
num_actions = env.action_space.n
print("Size of Action Space ->  {}".format(num_actions))
```

```
Size of State Space ->  4
Size of Action Space ->  2
```

```
class ReplayBuffer:
    def __init__(self, buffer_capacity=100000, batch_size=64):
        self.buffer_capacity = buffer_capacity
        self.batch_size = batch_size
        self.buffer = deque(maxlen=self.buffer_capacity)

    def record(self, obs_tuple):
        self.buffer.append(obs_tuple)

    def sample(self):
        batch = random.sample(self.buffer, self.batch_size)
        state_batch, action_batch, reward_batch, next_state_batch, done_batch = zip(*batch)
        return np.array(state_batch), np.array(action_batch), np.array(reward_batch), np.array(next_state_batch), np.array(done_batch)

    def size(self):
        return len(self.buffer)
```

```
def build_q_network():
    inputs = layers.Input(shape=(num_states,))
    out = layers.Dense(24, activation="relu")(inputs)
    out = layers.Dense(24, activation="relu")(out)
    outputs = layers.Dense(num_actions, activation="linear")(out)
    model = keras.Model(inputs, outputs)
    return model
```

```
def policy(state, epsilon):
    if np.random.rand() <= epsilon:
        return random.choice(range(num_actions))
    q_values = q_network.predict(state)
    return np.argmax(q_values[0])
```

```
def update_q_network(state_batch, action_batch, reward_batch, next_state_batch, done_batch):
    q_values_next = target_q_network.predict(next_state_batch)
    q_values_next_max = np.amax(q_values_next, axis=1)
    targets = reward_batch + gamma * (1 - done_batch) * q_values_next_max

    q_values = q_network.predict(state_batch)
    for i in range(buffer.batch_size):
        q_values[i][action_batch[i]] = targets[i]

    q_network.train_on_batch(state_batch, q_values)
```

```
def update_target_network():
    target_q_network.set_weights(q_network.get_weights())
```

```
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.995
gamma = 0.99
tau = 0.005
learning_rate = 0.001
batch_size = 64
buffer_capacity = 50000
total_episodes = 100
target_update_steps = 50 # Update target network every 100 steps
```

```
# Initialize the Q-networks
q_network = build_q_network()
target_q_network = build_q_network()
target_q_network.set_weights(q_network.get_weights())

# Compile the Q-network
q_network.compile(optimizer=keras.optimizers.Adam(learning_rate), loss='mse')
```

```
# Initialize the replay buffer
buffer = ReplayBuffer(buffer_capacity, batch_size)

# Store reward history of each episode
ep_reward_list = []
avg_reward_list = []
```

```
# Training loop
step = 0
for ep in range(total_episodes):
    state, _ = env.reset()
    state = np.expand_dims(state, axis=0)
    episodic_reward = 0

    while True:
        action = policy(state, epsilon)
        next_state, reward, done, truncated, _ = env.step(action)
        next_state = np.expand_dims(next_state, axis=0)

        buffer.record((state, action, reward, next_state, done))
        episodic_reward += reward

        if buffer.size() >= batch_size:
            state_batch, action_batch, reward_batch, next_state_batch, done_batch = buffer.sample()
            state_batch = np.squeeze(state_batch, axis=1)
            next_state_batch = np.squeeze(next_state_batch, axis=1)
            update_q_network(state_batch, action_batch, reward_batch, next_state_batch, done_batch)

            if step % target_update_steps == 0:
                update_target_network()
            step += 1

        state = next_state

        if done or truncated:
            break

    ep_reward_list.append(episodic_reward)
    avg_reward = np.mean(ep_reward_list[-40:])
    print("Episode * {} * Avg Reward is ==> {}".format(ep, avg_reward))
    avg_reward_list.append(avg_reward)

    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

# Plotting graph
plt.plot(avg_reward_list)
plt.xlabel("Episode")
plt.ylabel("Avg. Episodic Reward")
plt.show()
```

Show hidden output

```
Start coding or generate with AI.
```