HPC assignment final

CS22B2009 PULAPA SRIHITHA

April 2025

1 Original Code :

```
import os
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Dense
import tensorflow_probability as tfp
import pybullet_envs
import gym
from tqdm import trange
import time

class ReplayBuffer:
    def __init__(self, max_size, input_shape, n_actions):
        self.m_size = max_size
        self.m_cntr = 0
        self.state_m = np.zeros((self.m_size, *input_shape))
        self.new_state_m = np.zeros((self.m_size, *input_shape))
        self.action_m = np.zeros((self.m_size, n_actions))
        self.reward_m = np.zeros(self.m_size)
        self.terminal_m = np.zeros(self.m_size, dtype=bool)

    def store(self, state, action, reward, state_, done):
        id = self.m_cntr % self.m_size
        self.state_m[id] = state
        self.new_state_m[id] = state_
        self.action_m[id] = action
        self.reward_m[id] = reward
        self.terminal_m[id] = done
        self.m_cntr += 1

    def sample(self, batch_size):
        max_m = min(self.m_cntr, self.m_size)
        batch = np.random.choice(max_m, batch_size)
        return (self.state_m[batch], self.action_m[batch], self.reward_m[batch],
                self.new_state_m[batch], self.terminal_m[batch])

class CriticNetwork(keras.Model):
    def __init__(self, n_actions, fc1_dims=256, fc2_dims=256, name='critic', chkpt_dir='tmp/
        super(CriticNetwork, self).__init__()
        self.fc1 = Dense(fc1_dims, activation='relu')
        self.fc2 = Dense(fc2_dims, activation='relu')
        self.q = Dense(1, activation=None)
        self.model_name = name
        self.checkpoint_file = os.path.join(chkpt_dir, name+'_sac')

    def call(self, state, action):
        x = self.fc1(tf.concat([state, action], axis=1))
        x = self.fc2(x)
        return self.q(x)

class ValueNetwork(keras.Model):
    def __init__(self, fc1_dims=256, fc2_dims=256, name='value', chkpt_dir='tmp/sac'):
        super(ValueNetwork, self).__init__()
        self.fc1 = Dense(fc1_dims, activation='relu')
        self.fc2 = Dense(fc2_dims, activation='relu')
        self.v = Dense(1, activation=None)
        self.model_name = name
        self.checkpoint_file = os.path.join(chkpt_dir, name+'_sac')

    def call(self, state):
        x = self.fc1(state)
        x = self.fc2(x)
        return self.v(x)

class ActorNetwork(keras.Model):
    def __init__(self, max_action, fc1_dims=256, fc2_dims=256, n_actions=2, name='actor', ch
        super(ActorNetwork, self).__init__()
        self.max_action = max_action
        self.noise = 1e-6
        self.fc1 = Dense(fc1_dims, activation='relu')
        self.fc2 = Dense(fc2_dims, activation='relu')
        self.mu = Dense(n_actions)
        self.sigma = Dense(n_actions)
        self.model_name = name
        self.checkpoint_file = os.path.join(chkpt_dir, name+'_sac')

    def call(self, state):
        x = self.fc1(state)
        x = self.fc2(x)
        mu = self.mu(x)
        sigma = tf.clip_by_value(self.sigma(x), self.noise, 1)
        return mu, sigma

    def sample_normal(self, state, reparameterize=True):
        mu, sigma = self.call(state)
        dist = tfp.distributions.Normal(mu, sigma)
        actions = dist.sample() if not reparameterize else dist.sample()
        action = tf.tanh(actions) * self.max_action
        log_probs = dist.log_prob(actions)
        log_probs -= tf.math.log(1 - tf.pow(action, 2) + self.noise)
        log_probs = tf.reduce_sum(log_probs, axis=1, keepdims=True)
        return action, log_probs

class Agent:
    def __init__(self, alpha=0.0003, beta=0.0003, input_dims=[8], env=None,
                 gamma=0.99, n_actions=2, max_size=1000000, tau=0.005,
                 layer1_size=256, layer2_size=256, batch_size=256, reward_scale=2):
        self.gamma = gamma
        self.tau = tau
        self.memory = ReplayBuffer(max_size, input_dims, n_actions)
        self.batch_size = batch_size
        self.n_actions = n_actions
        self.scale = reward_scale

        self.actor = ActorNetwork(n_actions=n_actions, name='actor', max_action=env.action_s
        self.critic_1 = CriticNetwork(n_actions=n_actions, name='critic_1')
        self.critic_2 = CriticNetwork(n_actions=n_actions, name='critic_2')
        self.value = ValueNetwork(name='value')
        self.target_value = ValueNetwork(name='target_value')

        self.actor.compile(optimizer=Adam(learning_rate=alpha))
        self.critic_1.compile(optimizer=Adam(learning_rate=beta))
        self.critic_2.compile(optimizer=Adam(learning_rate=beta))
        self.value.compile(optimizer=Adam(learning_rate=beta))
        self.target_value.compile(optimizer=Adam(learning_rate=beta))

        self.update_network_parameters(tau=1)

    def choose_action(self, observation):
        state = tf.convert_to_tensor([observation], dtype=tf.float32)
        actions, _ = self.actor.sample_normal(state, reparameterize=False)
        return actions[0].numpy()

    def remember(self, state, action, reward, new_state, done):
        self.memory.store(state, action, reward, new_state, done)

    def update_network_parameters(self, tau=None):
        tau = tau or self.tau
        weights = [w * tau + t * (1 - tau) for w, t in zip(self.value.get_weights(), self.ta
        self.target_value.set_weights(weights)

    def learn(self):
        if self.memory.m_cntr < self.batch_size:
            return

        state, action, reward, new_state, done = self.memory.sample(self.batch_size)

        states = tf.convert_to_tensor(state, dtype=tf.float32)
        actions = tf.convert_to_tensor(action, dtype=tf.float32)
        rewards = tf.convert_to_tensor(reward, dtype=tf.float32)
        states_ = tf.convert_to_tensor(new_state, dtype=tf.float32)
        dones = tf.convert_to_tensor(done, dtype=tf.float32)

        with tf.GradientTape() as tape:
            value = tf.squeeze(self.value(states), 1)
            value_ = tf.squeeze(self.target_value(states_), 1)
            new_actions, log_probs = self.actor.sample_normal(states)
            log_probs = tf.squeeze(log_probs, 1)
            q1 = tf.squeeze(self.critic_1(states, new_actions), 1)
            q2 = tf.squeeze(self.critic_2(states, new_actions), 1)
            critic_value = tf.minimum(q1, q2)
            value_target = critic_value - log_probs
            value_loss = 0.5 * keras.losses.MSE(value, value_target)
        grads = tape.gradient(value_loss, self.value.trainable_variables)
        self.value.optimizer.apply_gradients(zip(grads, self.value.trainable_variables))

        with tf.GradientTape() as tape:
            new_actions, log_probs = self.actor.sample_normal(states, reparameterize=True)
            log_probs = tf.squeeze(log_probs, 1)
            q1 = tf.squeeze(self.critic_1(states, new_actions), 1)
            q2 = tf.squeeze(self.critic_2(states, new_actions), 1)
            critic_value = tf.minimum(q1, q2)
            actor_loss = tf.reduce_mean(log_probs - critic_value)
        grads = tape.gradient(actor_loss, self.actor.trainable_variables)
        self.actor.optimizer.apply_gradients(zip(grads, self.actor.trainable_variables))

        with tf.GradientTape(persistent=True) as tape:
            q_hat = self.scale * rewards + self.gamma * value_ * (1 - dones)
            q1_old = tf.squeeze(self.critic_1(states, actions), 1)
            q2_old = tf.squeeze(self.critic_2(states, actions), 1)
            c1_loss = 0.5 * keras.losses.MSE(q1_old, q_hat)
            c2_loss = 0.5 * keras.losses.MSE(q2_old, q_hat)
        grads1 = tape.gradient(c1_loss, self.critic_1.trainable_variables)
        grads2 = tape.gradient(c2_loss, self.critic_2.trainable_variables)
        self.critic_1.optimizer.apply_gradients(zip(grads1, self.critic_1.trainable_variable
        self.critic_2.optimizer.apply_gradients(zip(grads2, self.critic_2.trainable_variable

        self.update_network_parameters()

def main():
    startt = time.time()
    env = gym.make('InvertedPendulumBulletEnv-v0')
    agent = Agent(input_dims=env.observation_space.shape, env=env, n_actions=env.action_spac
    n_games = 100
    best_score = -np.inf
    score_history = []

    for i in trange(n_games, desc="Training"):
        done = False
        observation = env.reset()
        score = 0
        while not done:
            action = agent.choose_action(observation)
            observation_, reward, done, info = env.step(action)
            agent.remember(observation, action, reward, observation_, done)
            agent.learn()
            score += reward
            observation = observation_
        score_history.append(score)
        avg_score = np.mean(score_history[-100:])
        if avg_score > best_score:
            best_score = avg_score
        if i % 100 == 0:
            print(f"\nEpisode {i}, Score: {score:.1f}, Avg Score: {avg_score:.1f}")
    endt = time.time()
    print("Total Execution Time:", endt - startt)

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

2 C code :

```
#include <stdlib.h>
#include <stdbool.h>
```

```c
#include <string.h>
#include<math.h>
#include<stdio.h>
#include<time.h>


typedef struct {
    int max_size;
    int counter;
    int input_shape;
    int n_actions;

    float *state_m;
    float *new_state_m;
    float *action_m;
    float *reward_m;
    bool *terminal_m;
} ReplayBuffer;

// Function to initialize the ReplayBuffer
ReplayBuffer* create_replay_buffer(int max_size, int input_shape, int n_actions) {
    ReplayBuffer *buffer = (ReplayBuffer*)malloc(sizeof(ReplayBuffer));
    buffer->max_size = max_size;
    buffer->counter = 0;
    buffer->input_shape = input_shape;
    buffer->n_actions = n_actions;

    buffer->state_m = (float*)calloc(max_size * input_shape, sizeof(float));
    buffer->new_state_m = (float*)calloc(max_size * input_shape, sizeof(float));
    buffer->action_m = (float*)calloc(max_size * n_actions, sizeof(float));
    buffer->reward_m = (float*)calloc(max_size, sizeof(float));
    buffer->terminal_m = (bool*)calloc(max_size, sizeof(bool));

    return buffer;
}

// Function to store a transition in the buffer
void store_transition(ReplayBuffer *buffer, float *state, float *action,
                      float reward, float *state_, bool done) {
    int id = buffer->counter % buffer->max_size;

    memcpy(&buffer->state_m[id * buffer->input_shape], state, buffer->input_shape * sizeof(f
    memcpy(&buffer->new_state_m[id * buffer->input_shape], state_, buffer->input_shape * siz
    memcpy(&buffer->action_m[id * buffer->n_actions], action, buffer->n_actions * sizeof(flo
    buffer->reward_m[id] = reward;
    buffer->terminal_m[id] = done;
    buffer->counter++;
}

// Function to sample a batch of transitions
void sample_batch(ReplayBuffer *buffer, int batch_size, float *states, float *actions,
                  float *rewards, float *states_, bool *dones) {
    int max_m = buffer->counter < buffer->max_size ? buffer->counter : buffer->max_size;

    for (int i = 0; i < batch_size; i++) {
        int index = rand() % max_m;

        memcpy(&states[i * buffer->input_shape], &buffer->state_m[index * buffer->input_shap
        memcpy(&states_[i * buffer->input_shape], &buffer->new_state_m[index * buffer->input
        memcpy(&actions[i * buffer->n_actions], &buffer->action_m[index * buffer->n_actions]

        rewards[i] = buffer->reward_m[index];
        dones[i] = buffer->terminal_m[index];
    }
}

// Function to free allocated memory
void free_replay_buffer(ReplayBuffer *buffer) {
    free(buffer->state_m);
    free(buffer->new_state_m);
    free(buffer->action_m);
    free(buffer->reward_m);
    free(buffer->terminal_m);
    free(buffer);
}

typedef struct {
    int input_size;
    int output_size;
    float *weights;
    float *bias;
} DenseLayer;

// Function to initialize a dense layer
DenseLayer* create_dense_layer(int input_size, int output_size) {
    DenseLayer *layer = (DenseLayer*)malloc(sizeof(DenseLayer));
    layer->input_size = input_size;
    layer->output_size = output_size;
    layer->weights = (float*)malloc(input_size * output_size * sizeof(float));
    layer->bias = (float*)malloc(output_size * sizeof(float));
    // Randomly initialize weights and biases (for simplicity, set to small values)
    for (int i = 0; i < input_size * output_size; i++) {
        layer->weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < output_size; i++) {
        layer->bias[i] = 0;
    }

    return layer;
}

// ReLU activation function
void relu_dense(float *array, int size) {
    for (int i = 0; i < size; i++) {
        if (array[i] < 0) {
            array[i] = 0;
        }
    }
}

// Forward pass through a dense layer
void forward_dense(DenseLayer *layer, float *input, float *output) {
    for (int i = 0; i < layer->output_size; i++) {
        output[i] = layer->bias[i];
        for (int j = 0; j < layer->input_size; j++) {
            output[i] += input[j] * layer->weights[i * layer->input_size + j];
        }
    }
    relu_dense(output, layer->output_size);
}

// Critic Network
typedef struct {
    DenseLayer *fc1;
    DenseLayer *fc2;
    DenseLayer *q;
} CriticNetwork;

// Initialize the Critic Network
CriticNetwork* create_critic_network(int state_dim, int action_dim) {
    CriticNetwork *net = (CriticNetwork*)malloc(sizeof(CriticNetwork));
    net->fc1 = create_dense_layer(state_dim + action_dim, 256);
    net->fc2 = create_dense_layer(256, 256);
    net->q = create_dense_layer(256, 1);
    return net;
}
```

```
// Forward pass for Critic Network
float forward_critic(CriticNetwork *net, float *state, float *action, int state_dim, int act
    float input[state_dim + action_dim];
    memcpy(input, state, state_dim * sizeof(float));
    memcpy(input + state_dim, action, action_dim * sizeof(float));

    float hidden1[256], hidden2[256], q_value[1];
    forward_dense(net->fc1, input, hidden1);
    forward_dense(net->fc2, hidden1, hidden2);
    forward_dense(net->q, hidden2, q_value);

    return q_value[0];
}

// Value Network
typedef struct {
    DenseLayer *fc1;
    DenseLayer *fc2;
    DenseLayer *v;
} ValueNetwork;

// Initialize the Value Network
ValueNetwork* create_value_network(int state_dim) {
    ValueNetwork *net = (ValueNetwork*)malloc(sizeof(ValueNetwork));
    net->fc1 = create_dense_layer(state_dim, 256);
    net->fc2 = create_dense_layer(256, 256);
    net->v = create_dense_layer(256, 1);
    return net;
}

// Forward pass for Value Network
float forward_value(ValueNetwork *net, float *state) {
    float hidden1[256], hidden2[256], v_value[1];
    forward_dense(net->fc1, state, hidden1);
    forward_dense(net->fc2, hidden1, hidden2);
    forward_dense(net->v, hidden2, v_value);

    return v_value[0];
}

// Free memory for a dense layer
void free_dense_layer(DenseLayer *layer) {
    free(layer->weights);
    free(layer->bias);
    free(layer);
```

```
}

// Free memory for the networks
void free_critic_network(CriticNetwork *net) {
    free_dense_layer(net->fc1);
    free_dense_layer(net->fc2);
    free_dense_layer(net->q);
    free(net);
}

void free_value_network(ValueNetwork *net) {
    free_dense_layer(net->fc1);
    free_dense_layer(net->fc2);
    free_dense_layer(net->v);
    free(net);
}


typedef struct {
    int fc1_dims;
    int fc2_dims;
    int n_actions;
    float max_action;
    float noise;

    float *fc1_weights;
    float *fc2_weights;
    float *mu_weights;
    float *sigma_weights;
} ActorNetwork;


// Function to initialize the ActorNetwork
ActorNetwork* create_actor_network(float max_action, int fc1_dims, int fc2_dims, int n_actic
    ActorNetwork *net = (ActorNetwork*)malloc(sizeof(ActorNetwork));
    net->fc1_dims = fc1_dims;
    net->fc2_dims = fc2_dims;
    net->n_actions = n_actions;
    net->max_action = max_action;
    net->noise = 1e-6;

    // Allocate memory for weights and initialize them
    net->fc1_weights = (float*)calloc(fc1_dims * fc2_dims, sizeof(float));
    net->fc2_weights = (float*)calloc(fc2_dims * n_actions, sizeof(float));
    net->mu_weights = (float*)calloc(n_actions, sizeof(float));
    net->sigma_weights = (float*)calloc(n_actions, sizeof(float));
```

```c
    // Initialize weights with small random values
    for (int i = 0; i < fc1_dims * fc2_dims; i++) {
        net->fc1_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < fc2_dims * n_actions; i++) {
        net->fc2_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < n_actions; i++) {
        net->mu_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
        net->sigma_weights[i] = 1.0; // Initial sigma set to 1 for exploration
    }

    return net;
}


// Simple ReLU activation function
void relu(float *input, int size) {
    for (int i = 0; i < size; i++) {
        if (input[i] < 0) input[i] = 0;
    }
}


// Forward pass
void forward(ActorNetwork *net, float *state, float *mu, float *sigma) {
    float fc1_output[net->fc2_dims];
    float fc2_output[net->n_actions];

    // Fully connected layer 1
    for (int i = 0; i < net->fc2_dims; i++) {
        fc1_output[i] = 0;
        for (int j = 0; j < net->fc1_dims; j++) {
            fc1_output[i] += state[j] * net->fc1_weights[i * net->fc1_dims + j];
        }
    }
    relu(fc1_output, net->fc2_dims);

    // Fully connected layer 2
    for (int i = 0; i < net->n_actions; i++) {
        fc2_output[i] = 0;
        for (int j = 0; j < net->fc2_dims; j++) {
            fc2_output[i] += fc1_output[j] * net->fc2_weights[i * net->fc2_dims + j];
        }
    }
    relu(fc2_output, net->n_actions);

    // Compute mu and sigma
    for (int i = 0; i < net->n_actions; i++) {
        mu[i] = net->max_action * tanh(fc2_output[i]);  // Bound actions
        sigma[i] = fmax(net->noise, fmin(net->sigma_weights[i], 1.0)); // Ensure sigma is po
    }
}


// Function to sample an action from a normal distribution
void sample_normal(ActorNetwork *net, float *state, float *action, float *log_probs) {
    float mu[net->n_actions];
    float sigma[net->n_actions];

    forward(net, state, mu, sigma);

    for (int i = 0; i < net->n_actions; i++) {
        // Sample from normal distribution using Box-Muller transform
        float epsilon = ((float)rand() / RAND_MAX) * 2 - 1;
        action[i] = mu[i] + sigma[i] * epsilon;

        // Compute log probability only if log_probs is not NULL
        if (log_probs != NULL) {
            float exponent = -0.5 * pow((action[i] - mu[i]) / sigma[i], 2);
            log_probs[i] = exponent - log(sigma[i]) - log(sqrt(2 * M_PI));
        }
    }
}



// Function to free memory
void free_actor_network(ActorNetwork *net) {
    free(net->fc1_weights);
    free(net->fc2_weights);
    free(net->mu_weights);
    free(net->sigma_weights);
    free(net);
}


typedef struct {
    float gamma;
    float tau;
    int batch_size;
```

```c
    int n_actions;
    float scale;

    ReplayBuffer *memory;
    ActorNetwork *actor;
    CriticNetwork *critic_1;
    CriticNetwork *critic_2;
    ValueNetwork *value;
    ValueNetwork *target_value;
} Agent;

// Function to create an agent
Agent* create_agent(float alpha, float beta, int input_dims, int n_actions,
                    int max_size, float tau, int layer1_size, int layer2_size,
                    int batch_size, float reward_scale) {
    Agent *agent = (Agent*)malloc(sizeof(Agent));
    agent->gamma = 0.99;
    agent->tau = tau;
    agent->batch_size = batch_size;
    agent->n_actions = n_actions;
    agent->scale = reward_scale;

    // Create networks
    agent->memory = create_replay_buffer(max_size, input_dims, n_actions);
    agent->actor = create_actor_network(1.0, layer1_size, layer2_size, n_actions);
    agent->critic_1 = create_critic_network(input_dims , n_actions);
    agent->critic_2 = create_critic_network(input_dims, n_actions);
    agent->value = create_value_network(input_dims);
    agent->target_value = create_value_network(input_dims);

    return agent;
}

// Function to choose action
void choose_action(Agent *agent, float *observation, float *action) {
    sample_normal(agent->actor, observation, action, NULL);
}

// Function to store experience
void remember(Agent *agent, float *state, float *action, float reward, float *new_state, boo
    store_transition(agent->memory, state, action, reward, new_state, done);
}

// Update target network parameters
void update_network_parameters(Agent *agent, float tau) {
    if (tau == 0) {
        tau = agent->tau;
    }

    for (int i = 0; i < 256; i++) {  // Assuming 256 weights
        agent->target_value->v->weights[i] = tau * agent->value->v->weights[i] +
                                             (1 - tau) * agent->target_value->v->weights[i];
    }
}


// Learning function
void learn(Agent *agent) {
    if (agent->memory->counter < agent->batch_size) {
        return;
    }

    float states[agent->batch_size * 8];  // Flattened array instead of 2D array
    float new_states[agent->batch_size * 8];
    float actions[agent->batch_size * agent->n_actions];
    float rewards[agent->batch_size];
    bool dones[agent->batch_size];

    sample_batch(agent->memory, agent->batch_size, states, actions, rewards, new_states, do

    update_network_parameters(agent, 0);
}

// Free memory
void free_agent(Agent *agent) {
    free_replay_buffer(agent->memory);
    free_actor_network(agent->actor);
    free_critic_network(agent->critic_1);
    free_critic_network(agent->critic_2);
    free_value_network(agent->value);
    free_value_network(agent->target_value);
    free(agent);
}

typedef struct {
    int observation_space_dim;
    int action_space_dim;
    float reward_range_min;
    float reward_range_max;
} Env;

Env* create_env() {
    Env *env = (Env*)malloc(sizeof(Env));
    env->observation_space_dim = 8; // Assuming 8 observations
    env->action_space_dim = 2;      // Assuming 2 actions
    env->reward_range_min = -100;
    env->reward_range_max = 100;
    return env;
}

void free_env(Env *env) {
    free(env);
}

// Placeholder function for resetting the environment
void reset_env(Env *env, float *observation) {
    for (int i = 0; i < env->observation_space_dim; i++) {
        observation[i] = ((float)rand() / RAND_MAX) * 2 - 1;  // Random values between -1 a
    }
}

// Placeholder function for stepping in the environment
void step_env(Env *env, float *action, float *observation, float *reward, bool *done) {
    for (int i = 0; i < env->observation_space_dim; i++) {
        observation[i] = ((float)rand() / RAND_MAX) * 2 - 1;
    }
    *reward = ((float)rand() / RAND_MAX) * (env->reward_range_max - env->reward_range_min) +
    *done = rand() % 10 == 0; // Randomly end episodes
}

int main() {
    clock_t start, end;
    double cpu_time_used;
    start = clock();

    Env *env = create_env();
    Agent *agent = create_agent(0.0003, 0.0003, env->observation_space_dim, env->action_spac
                                1000000, 0.005, 256, 256, 256, 2);

    int n_games = 10000;
    float best_score = env->reward_range_min;
    float score_history[100] = {0};
    int history_size = 0;

    for (int i = 0; i < n_games; i++) {
        float observation[env->observation_space_dim];
        reset_env(env, observation);
        bool done = false;
        float score = 0;

        while (!done) {
            float action[env->action_space_dim];
            choose_action(agent, observation, action);

            float new_observation[env->observation_space_dim];
            float reward;
            step_env(env, action, new_observation, &reward, &done);

            score += reward;
            remember(agent, observation, action, reward, new_observation, done);
            learn(agent);

            // Copy new observation to current observation
            memcpy(observation, new_observation, sizeof(new_observation));
        }

        // Store score
        if (history_size < 100) {
            score_history[history_size++] = score;
        } else {
            for (int j = 1; j < 100; j++) {
                score_history[j - 1] = score_history[j];
            }
            score_history[99] = score;
        }

        // Compute average score
        float avg_score = 0;
        for (int j = 0; j < history_size; j++) {
            avg_score += score_history[j];
        }
        avg_score /= history_size;

        // Update best score
        if (avg_score > best_score) {
            best_score = avg_score;
        }
        if(i%100 == 0)
            printf("Episode %d | Score: %.1f | Avg Score: %.1f\n", i, score, avg_score);
    }

    // Free memory
    free_agent(agent);
    free_env(env);
    end = clock();  // End time

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;  // Convert to seconds
    printf("Time taken: %f seconds\n", cpu_time_used);


    return 0;
}
```

3 Open Mp Code:

```
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include<math.h>
#include<stdio.h>
#include<time.h>


typedef struct {
    int max_size;
    int counter;
    int input_shape;
    int n_actions;

    float *state_m;
    float *new_state_m;
    float *action_m;
    float *reward_m;
    bool *terminal_m;
} ReplayBuffer;

// Function to initialize the ReplayBuffer
ReplayBuffer* create_replay_buffer(int max_size, int input_shape, int n_actions) {
    ReplayBuffer *buffer = (ReplayBuffer*)malloc(sizeof(ReplayBuffer));
    buffer->max_size = max_size;
    buffer->counter = 0;
    buffer->input_shape = input_shape;
    buffer->n_actions = n_actions;

    buffer->state_m = (float*)calloc(max_size * input_shape, sizeof(float));
    buffer->new_state_m = (float*)calloc(max_size * input_shape, sizeof(float));
    buffer->action_m = (float*)calloc(max_size * n_actions, sizeof(float));
    buffer->reward_m = (float*)calloc(max_size, sizeof(float));
    buffer->terminal_m = (bool*)calloc(max_size, sizeof(bool));

    return buffer;
}

// Function to store a transition in the buffer
void store_transition(ReplayBuffer *buffer, float *state, float *action,
                      float reward, float *state_, bool done) {
    int id = buffer->counter % buffer->max_size;

    memcpy(&buffer->state_m[id * buffer->input_shape], state, buffer->input_shape * sizeof(f
    memcpy(&buffer->new_state_m[id * buffer->input_shape], state_, buffer->input_shape * siz
    memcpy(&buffer->action_m[id * buffer->n_actions], action, buffer->n_actions * sizeof(flo
    buffer->reward_m[id] = reward;
    buffer->terminal_m[id] = done;

    buffer->counter++;
}

// Function to sample a batch of transitions
void sample_batch(ReplayBuffer *buffer, int batch_size, float *states, float *actions,
                  float *rewards, float *states_, bool *dones) {
    int max_m = buffer->counter < buffer->max_size ? buffer->counter : buffer->max_size;

    for (int i = 0; i < batch_size; i++) {
        int index = rand() % max_m;

        memcpy(&states[i * buffer->input_shape], &buffer->state_m[index * buffer->input_shap
        memcpy(&states_[i * buffer->input_shape], &buffer->new_state_m[index * buffer->input
        memcpy(&actions[i * buffer->n_actions], &buffer->action_m[index * buffer->n_actions]

        rewards[i] = buffer->reward_m[index];
        dones[i] = buffer->terminal_m[index];
    }
}

// Function to free allocated memory
void free_replay_buffer(ReplayBuffer *buffer) {
    free(buffer->state_m);
    free(buffer->new_state_m);
    free(buffer->action_m);
    free(buffer->reward_m);
    free(buffer->terminal_m);
    free(buffer);
}
```

```c
typedef struct {
    int input_size;
    int output_size;
    float *weights;
    float *bias;
} DenseLayer;

// Function to initialize a dense layer
DenseLayer* create_dense_layer(int input_size, int output_size) {
    DenseLayer *layer = (DenseLayer*)malloc(sizeof(DenseLayer));
    layer->input_size = input_size;
    layer->output_size = output_size;
    layer->weights = (float*)malloc(input_size * output_size * sizeof(float));
    layer->bias = (float*)malloc(output_size * sizeof(float));

    // Randomly initialize weights and biases (for simplicity, set to small values)
    for (int i = 0; i < input_size * output_size; i++) {
        layer->weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < output_size; i++) {
        layer->bias[i] = 0;
    }

    return layer;
}

// ReLU activation function
void relu_dense(float *array, int size) {
    for (int i = 0; i < size; i++) {
        if (array[i] < 0) {
            array[i] = 0;
        }
    }
}

// Forward pass through a dense layer (parallel)
void forward_dense(DenseLayer *layer, float *input, float *output) {
    #pragma omp parallel for
    for (int i = 0; i < layer->output_size; i++) {
        float sum = layer->bias[i];
        for (int j = 0; j < layer->input_size; j++) {
            sum += input[j] * layer->weights[i * layer->input_size + j];
        }
        output[i] = sum;
    }
    relu_dense(output, layer->output_size);
}


// Critic Network
typedef struct {
    DenseLayer *fc1;
    DenseLayer *fc2;
    DenseLayer *q;
} CriticNetwork;

// Initialize the Critic Network
CriticNetwork* create_critic_network(int state_dim, int action_dim) {
    CriticNetwork *net = (CriticNetwork*)malloc(sizeof(CriticNetwork));
    net->fc1 = create_dense_layer(state_dim + action_dim, 256);
    net->fc2 = create_dense_layer(256, 256);
    net->q = create_dense_layer(256, 1);
    return net;
}

// Forward pass for Critic Network
float forward_critic(CriticNetwork *net, float *state, float *action, int state_dim, int act
    float input[state_dim + action_dim];
    memcpy(input, state, state_dim * sizeof(float));
    memcpy(input + state_dim, action, action_dim * sizeof(float));

    float hidden1[256], hidden2[256], q_value[1];
    forward_dense(net->fc1, input, hidden1);
    forward_dense(net->fc2, hidden1, hidden2);
    forward_dense(net->q, hidden2, q_value);

    return q_value[0];
}

// Value Network
typedef struct {
    DenseLayer *fc1;
    DenseLayer *fc2;
    DenseLayer *v;
} ValueNetwork;

// Initialize the Value Network
ValueNetwork* create_value_network(int state_dim) {
    ValueNetwork *net = (ValueNetwork*)malloc(sizeof(ValueNetwork));
    net->fc1 = create_dense_layer(state_dim, 256);
    net->fc2 = create_dense_layer(256, 256);
    net->v = create_dense_layer(256, 1);
    return net;
}

// Forward pass for Value Network
float forward_value(ValueNetwork *net, float *state) {
    float hidden1[256], hidden2[256], v_value[1];
    forward_dense(net->fc1, state, hidden1);
    forward_dense(net->fc2, hidden1, hidden2);
    forward_dense(net->v, hidden2, v_value);

    return v_value[0];
}

// Free memory for a dense layer
void free_dense_layer(DenseLayer *layer) {
    free(layer->weights);
    free(layer->bias);
    free(layer);
}

// Free memory for the networks
void free_critic_network(CriticNetwork *net) {
    free_dense_layer(net->fc1);
    free_dense_layer(net->fc2);
    free_dense_layer(net->q);
    free(net);
}

void free_value_network(ValueNetwork *net) {
    free_dense_layer(net->fc1);
    free_dense_layer(net->fc2);
    free_dense_layer(net->v);
    free(net);
}


typedef struct {
    int fc1_dims;
    int fc2_dims;
    int n_actions;
    float max_action;
    float noise;

    float *fc1_weights;
    float *fc2_weights;
    float *mu_weights;
    float *sigma_weights;
} ActorNetwork;


// Function to initialize the ActorNetwork
ActorNetwork* create_actor_network(float max_action, int fc1_dims, int fc2_dims, int n_actio
    ActorNetwork *net = (ActorNetwork*)malloc(sizeof(ActorNetwork));
    net->fc1_dims = fc1_dims;
    net->fc2_dims = fc2_dims;
    net->n_actions = n_actions;
    net->max_action = max_action;
    net->noise = 1e-6;

    // Allocate memory for weights and initialize them
    net->fc1_weights = (float*)calloc(fc1_dims * fc2_dims, sizeof(float));
    net->fc2_weights = (float*)calloc(fc2_dims * n_actions, sizeof(float));
    net->mu_weights = (float*)calloc(n_actions, sizeof(float));
    net->sigma_weights = (float*)calloc(n_actions, sizeof(float));

    // Initialize weights with small random values
    for (int i = 0; i < fc1_dims * fc2_dims; i++) {
        net->fc1_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < fc2_dims * n_actions; i++) {
        net->fc2_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < n_actions; i++) {
        net->mu_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
        net->sigma_weights[i] = 1.0; // Initial sigma set to 1 for exploration
    }

    return net;
}


// Simple ReLU activation function
void relu(float *input, int size) {
    for (int i = 0; i < size; i++) {
        if (input[i] < 0) input[i] = 0;
    }
}


// Forward pass
```

```
void forward(ActorNetwork *net, float *state, float *mu, float *sigma) {
    float fc1_output[net->fc2_dims];
    float fc2_output[net->n_actions];

    // Fully connected layer 1
    #pragma omp parallel for
    for (int i = 0; i < net->fc2_dims; i++) {
        fc1_output[i] = 0;
        for (int j = 0; j < net->fc1_dims; j++) {
            fc1_output[i] += state[j] * net->fc1_weights[i * net->fc1_dims + j];
        }
    }
    relu(fc1_output, net->fc2_dims);

    // Fully connected layer 2
    #pragma omp parallel for
    for (int i = 0; i < net->n_actions; i++) {
        fc2_output[i] = 0;
        for (int j = 0; j < net->fc2_dims; j++) {
            fc2_output[i] += fc1_output[j] * net->fc2_weights[i * net->fc2_dims + j];
        }
    }
    relu(fc2_output, net->n_actions);

    // Compute mu and sigma
    for (int i = 0; i < net->n_actions; i++) {
        mu[i] = net->max_action * tanh(fc2_output[i]);  // Bound actions
        sigma[i] = fmax(net->noise, fmin(net->sigma_weights[i], 1.0)); // Ensure sigma is po
    }
}


// Function to sample an action from a normal distribution
void sample_normal(ActorNetwork *net, float *state, float *action, float *log_probs) {
    float mu[net->n_actions];
    float sigma[net->n_actions];

    forward(net, state, mu, sigma);

    for (int i = 0; i < net->n_actions; i++) {
        // Sample from normal distribution using Box-Muller transform
        float epsilon = ((float)rand() / RAND_MAX) * 2 - 1;
        action[i] = mu[i] + sigma[i] * epsilon;

        // Compute log probability only if log_probs is not NULL
        if (log_probs != NULL) {
                float exponent = -0.5 * pow((action[i] - mu[i]) / sigma[i], 2);
                log_probs[i] = exponent - log(sigma[i]) - log(sqrt(2 * M_PI));
            }
        }
    }



// Function to free memory
void free_actor_network(ActorNetwork *net) {
    free(net->fc1_weights);
    free(net->fc2_weights);
    free(net->mu_weights);
    free(net->sigma_weights);
    free(net);
}


typedef struct {
    float gamma;
    float tau;
    int batch_size;
    int n_actions;
    float scale;

    ReplayBuffer *memory;
    ActorNetwork *actor;
    CriticNetwork *critic_1;
    CriticNetwork *critic_2;
    ValueNetwork *value;
    ValueNetwork *target_value;
} Agent;

// Function to create an agent
Agent* create_agent(float alpha, float beta, int input_dims, int n_actions,
                    int max_size, float tau, int layer1_size, int layer2_size,
                    int batch_size, float reward_scale) {
    Agent *agent = (Agent*)malloc(sizeof(Agent));
    agent->gamma = 0.99;
    agent->tau = tau;
    agent->batch_size = batch_size;
    agent->n_actions = n_actions;
    agent->scale = reward_scale;

    // Create networks
    agent->memory = create_replay_buffer(max_size, input_dims, n_actions);
    agent->actor = create_actor_network(1.0, layer1_size, layer2_size, n_actions);
    agent->critic_1 = create_critic_network(input_dims , n_actions);
    agent->critic_2 = create_critic_network(input_dims, n_actions);
    agent->value = create_value_network(input_dims);
    agent->target_value = create_value_network(input_dims);

    return agent;
}

// Function to choose action
void choose_action(Agent *agent, float *observation, float *action) {
    sample_normal(agent->actor, observation, action, NULL);
}

// Function to store experience
void remember(Agent *agent, float *state, float *action, float reward, float *new_state, boo
    store_transition(agent->memory, state, action, reward, new_state, done);
}

// Update target network parameters
void update_network_parameters(Agent *agent, float tau) {
    if (tau == 0) {
        tau = agent->tau;
    }

    for (int i = 0; i < 256; i++) {  // Assuming 256 weights
        agent->target_value->v->weights[i] = tau * agent->value->v->weights[i] +
                                             (1 - tau) * agent->target_value->v->weights[i]
    }
}


// Learning function
void learn(Agent *agent) {
    if (agent->memory->counter < agent->batch_size) {
        return;
    }

    float states[agent->batch_size * 8];  // Flattened array instead of 2D array
    float new_states[agent->batch_size * 8];
    float actions[agent->batch_size * agent->n_actions];
    float rewards[agent->batch_size];
    bool dones[agent->batch_size];

    sample_batch(agent->memory, agent->batch_size, states, actions, rewards, new_states, do
```

```c
    update_network_parameters(agent, 0);
}

// Free memory
void free_agent(Agent *agent) {
    free_replay_buffer(agent->memory);
    free_actor_network(agent->actor);
    free_critic_network(agent->critic_1);
    free_critic_network(agent->critic_2);
    free_value_network(agent->value);
    free_value_network(agent->target_value);
    free(agent);
}

typedef struct {
    int observation_space_dim;
    int action_space_dim;
    float reward_range_min;
    float reward_range_max;
} Env;

Env* create_env() {
    Env *env = (Env*)malloc(sizeof(Env));
    env->observation_space_dim = 8; // Assuming 8 observations
    env->action_space_dim = 2;      // Assuming 2 actions
    env->reward_range_min = -100;
    env->reward_range_max = 100;
    return env;
}

void free_env(Env *env) {
    free(env);
}

// Placeholder function for resetting the environment
void reset_env(Env *env, float *observation) {
    for (int i = 0; i < env->observation_space_dim; i++) {
        observation[i] = ((float)rand() / RAND_MAX) * 2 - 1;  // Random values between -1 a
    }
}

// Placeholder function for stepping in the environment
void step_env(Env *env, float *action, float *observation, float *reward, bool *done) {
    for (int i = 0; i < env->observation_space_dim; i++) {
        observation[i] = ((float)rand() / RAND_MAX) * 2 - 1;
    }
    *reward = ((float)rand() / RAND_MAX) * (env->reward_range_max - env->reward_range_min) -
    *done = rand() % 10 == 0; // Randomly end episodes
}

int main() {
    clock_t start, end;
    double cpu_time_used;
    start = clock();

    Env *env = create_env();
    Agent *agent = create_agent(0.0003, 0.0003, env->observation_space_dim, env->action_spac
                                1000000, 0.005, 256, 256, 256, 2);

    int n_games = 10000;
    float best_score = env->reward_range_min;
    float score_history[100] = {0};
    int history_size = 0;

    for (int i = 0; i < n_games; i++) {
        float observation[env->observation_space_dim];
        reset_env(env, observation);

        bool done = false;
        float score = 0;

        while (!done) {
            float action[env->action_space_dim];
            choose_action(agent, observation, action);

            float new_observation[env->observation_space_dim];
            float reward;
            step_env(env, action, new_observation, &reward, &done);

            score += reward;
            remember(agent, observation, action, reward, new_observation, done);
            learn(agent);

            // Copy new observation to current observation
            memcpy(observation, new_observation, sizeof(new_observation));
        }

        // Store score
        if (history_size < 100) {
            score_history[history_size++] = score;
        } else {
            for (int j = 1; j < 100; j++) {
                score_history[j - 1] = score_history[j];
            }
            score_history[99] = score;
        }

        // Compute average score
        float avg_score = 0;
        for (int j = 0; j < history_size; j++) {
            avg_score += score_history[j];
        }
        avg_score /= history_size;

        // Update best score
        if (avg_score > best_score) {
            best_score = avg_score;
        }
        if(i%100 == 0)
            printf("Episode %d | Score: %.1f | Avg Score: %.1f\n", i, score, avg_score);
    }

    // Free memory
    free_agent(agent);
    free_env(env);
    end = clock();  // End time

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;  // Convert to seconds
    printf("Time taken: %f seconds\n", cpu_time_used);


    return 0;
}
```

4 Cuda Code:

```
import os
import time
import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import gym
from tqdm import trange
```

```python
from numba import cuda

# CUDA kernel to normalize rewards
@cuda.jit
def normalize_rewards_kernel(in_arr, out_arr):
    idx = cuda.grid(1)
    if idx < in_arr.size:
        out_arr[idx] = in_arr[idx] / 10.0

class ReplayBuffer:
    def __init__(self, max_size, input_shape, n_actions):
        self.mem_size = max_size
        self.mem_cntr = 0
        self.state_memory = np.zeros((self.mem_size, *input_shape), dtype=np.float32)
        self.new_state_memory = np.zeros((self.mem_size, *input_shape), dtype=np.float32)
        self.action_memory = np.zeros((self.mem_size, n_actions), dtype=np.float32)
        self.reward_memory = np.zeros(self.mem_size, dtype=np.float32)
        self.terminal_memory = np.zeros(self.mem_size, dtype=bool)

    def store_transition(self, state, action, reward, state_, done):
        index = self.mem_cntr % self.mem_size
        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.action_memory[index] = action
        self.reward_memory[index] = reward
        self.terminal_memory[index] = done
        self.mem_cntr += 1

    def sample_buffer(self, batch_size):
        max_mem = min(self.mem_cntr, self.mem_size)
        batch = np.random.choice(max_mem, batch_size)

        states = self.state_memory[batch]
        actions = self.action_memory[batch]
        rewards = self.reward_memory[batch]
        states_ = self.new_state_memory[batch]
        terminal = self.terminal_memory[batch]

        # Normalize rewards using CUDA
        d_input = cuda.to_device(rewards)
        d_output = cuda.device_array_like(d_input)
        threadsperblock = 128
        blockspergrid = (rewards.size + (threadsperblock - 1)) // threadsperblock
        normalize_rewards_kernel[blockspergrid, threadsperblock](d_input, d_output)
        rewards = d_output.copy_to_host()
        return states, actions, rewards, states_, terminal

class CriticNetwork(Model):
    def __init__(self, n_actions):
        super(CriticNetwork, self).__init__()
        self.fc1 = Dense(256, activation='relu')
        self.fc2 = Dense(256, activation='relu')
        self.q = Dense(1, activation=None)

    def call(self, state, action):
        x = tf.concat([state, action], axis=1)
        x = self.fc1(x)
        x = self.fc2(x)
        return self.q(x)

class ValueNetwork(Model):
    def __init__(self):
        super(ValueNetwork, self).__init__()
        self.fc1 = Dense(256, activation='relu')
        self.fc2 = Dense(256, activation='relu')
        self.v = Dense(1, activation=None)

    def call(self, state):
        x = self.fc1(state)
        x = self.fc2(x)
        return self.v(x)

class ActorNetwork(Model):
    def __init__(self, n_actions, max_action):
        super(ActorNetwork, self).__init__()
        self.fc1 = Dense(256, activation='relu')
        self.fc2 = Dense(256, activation='relu')
        self.mu = Dense(n_actions, activation=None)
        self.sigma = Dense(n_actions, activation='softplus')
        self.max_action = max_action
        self.reparam_noise = 1e-6

    def call(self, state):
        x = self.fc1(state)
        x = self.fc2(x)
        mu = self.mu(x)
        sigma = self.sigma(x) + self.reparam_noise
        return mu, sigma

    def sample_normal(self, state, reparameterize=True):
        mu, sigma = self.call(state)
        dist = tfp.distributions.Normal(mu, sigma)
        actions = dist.sample()
        action = tf.math.tanh(actions) * self.max_action
        log_prob = dist.log_prob(actions)
        log_prob -= tf.math.log(1 - tf.math.pow(action, 2) + 1e-6)
        log_prob = tf.reduce_sum(log_prob, axis=1, keepdims=True)
        return action, log_prob

class Agent:
    def __init__(self, alpha=0.0003, beta=0.0003, input_dims=[8], env=None,
                 gamma=0.99, n_actions=2, max_size=1000000, tau=0.005,
                 batch_size=256):
        self.gamma = gamma
        self.tau = tau
        self.memory = ReplayBuffer(max_size, input_dims, n_actions)
        self.batch_size = batch_size

        self.actor = ActorNetwork(n_actions, env.action_space.high[0])
        self.critic_1 = CriticNetwork(n_actions)
        self.critic_2 = CriticNetwork(n_actions)
        self.value = ValueNetwork()
        self.target_value = ValueNetwork()

        self.actor.compile(optimizer=Adam(learning_rate=alpha))
        self.critic_1.compile(optimizer=Adam(learning_rate=beta))
        self.critic_2.compile(optimizer=Adam(learning_rate=beta))
        self.value.compile(optimizer=Adam(learning_rate=beta))

        self.update_network_parameters(tau=1)

    def update_network_parameters(self, tau=None):
        if tau is None:
            tau = self.tau

        weights = []
        targets = self.target_value.weights
        for i, weight in enumerate(self.value.weights):
            weights.append(weight * tau + targets[i] * (1 - tau))
        self.target_value.set_weights(weights)

    def remember(self, state, action, reward, new_state, done):
        self.memory.store_transition(state, action, reward, new_state, done)

    def choose_action(self, observation):
        state = tf.convert_to_tensor(observation, dtype=tf.float32)
        state = tf.expand_dims(state, axis=0)
        actions, _ = self.actor.sample_normal(state, reparameterize=False)
        return actions[0].numpy()

    def learn(self):
        if self.memory.mem_cntr < self.batch_size:
            return

        state, action, reward, new_state, done = self.memory.sample_buffer(self.batch_size)

        state = tf.convert_to_tensor(state, dtype=tf.float32)
        action = tf.convert_to_tensor(action, dtype=tf.float32)
        reward = tf.convert_to_tensor(reward, dtype=tf.float32)
        new_state = tf.convert_to_tensor(new_state, dtype=tf.float32)
        done = tf.convert_to_tensor(done.astype(np.float32), dtype=tf.float32)

        with tf.GradientTape() as tape:
            value = tf.squeeze(self.value(state), 1)
            target_value = tf.squeeze(self.target_value(new_state), 1)
            target = reward + self.gamma * target_value * (1 - done)
            value_loss = 0.5 * tf.keras.losses.MSE(target, value)
        grads = tape.gradient(value_loss, self.value.trainable_variables)
        self.value.optimizer.apply_gradients(zip(grads, self.value.trainable_variables))

        with tf.GradientTape() as tape:
            new_actions, log_probs = self.actor.sample_normal(state, reparameterize=True)
            q1_new_policy = tf.squeeze(self.critic_1(state, new_actions), 1)
            q2_new_policy = tf.squeeze(self.critic_2(state, new_actions), 1)
            critic_value = tf.minimum(q1_new_policy, q2_new_policy)
            actor_loss = tf.reduce_mean(log_probs - critic_value)
        actor_grads = tape.gradient(actor_loss, self.actor.trainable_variables)
        self.actor.optimizer.apply_gradients(zip(actor_grads, self.actor.trainable_variables

        with tf.GradientTape(persistent=True) as tape:
            q1_old_policy = tf.squeeze(self.critic_1(state, action), 1)
            q2_old_policy = tf.squeeze(self.critic_2(state, action), 1)
            target = reward + self.gamma * target_value * (1 - done)
            critic_1_loss = 0.5 * tf.keras.losses.MSE(target, q1_old_policy)
            critic_2_loss = 0.5 * tf.keras.losses.MSE(target, q2_old_policy)
        critic_1_grads = tape.gradient(critic_1_loss, self.critic_1.trainable_variables)
        critic_2_grads = tape.gradient(critic_2_loss, self.critic_2.trainable_variables)
        self.critic_1.optimizer.apply_gradients(zip(critic_1_grads, self.critic_1.trainable_
        self.critic_2.optimizer.apply_gradients(zip(critic_2_grads, self.critic_2.trainable_

        self.update_network_parameters()

# GPU setup check
```

```python
def setup_gpu():
    try:
        cuda.select_device(0)
        print("GPU is available.")
    except:
        print("Using CPU only.")

setup_gpu()

start_time = time.time()

env = gym.make('InvertedPendulum-v4', render_mode=None, new_step_api=True)

gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError as e:
        print(e)

agent = Agent(input_dims=env.observation_space.shape, env=env,
              n_actions=env.action_space.shape[0])

n_games = 1000
score_history = []
best_score = -np.inf

for i in trange(n_games, desc="Training"):
    observation = env.reset()
    done = False
    score = 0

    while not done:
        action = agent.choose_action(observation)
        observation_, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated

        agent.remember(observation, action, reward, observation_, done)
        agent.learn()
        score += reward
        observation = observation_

    score_history.append(score)
    avg_score = np.mean(score_history[-100:])
    if avg_score > best_score:
        best_score = avg_score

    if i % 100 == 0:
        print(f"\nEpisode {i}, Score: {score:.2f}, Avg Score: {avg_score:.2f}")

print("Total Training Time:", time.time() - start_time)
```

5 MPI Code:

```
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include<math.h>
#include<stdio.h>
#include<time.h>
#include <mpi.h>



typedef struct {
    int max_size;
    int counter;
    int input_shape;
    int n_actions;

    float *state_m;
    float *new_state_m;
    float *action_m;
    float *reward_m;
    bool *terminal_m;
} ReplayBuffer;

// Function to initialize the ReplayBuffer
ReplayBuffer* create_replay_buffer(int max_size, int input_shape, int n_actions) {
    ReplayBuffer *buffer = (ReplayBuffer*)malloc(sizeof(ReplayBuffer));
    buffer->max_size = max_size;
    buffer->counter = 0;
    buffer->input_shape = input_shape;
    buffer->n_actions = n_actions;

    buffer->state_m = (float*)calloc(max_size * input_shape, sizeof(float));
    buffer->new_state_m = (float*)calloc(max_size * input_shape, sizeof(float));
    buffer->action_m = (float*)calloc(max_size * n_actions, sizeof(float));
    buffer->reward_m = (float*)calloc(max_size, sizeof(float));
    buffer->terminal_m = (bool*)calloc(max_size, sizeof(bool));

    return buffer;
}

// Function to store a transition in the buffer
void store_transition(ReplayBuffer *buffer, float *state, float *action,
                      float reward, float *state_, bool done) {
    int id = buffer->counter % buffer->max_size;

    memcpy(&buffer->state_m[id * buffer->input_shape], state, buffer->input_shape * sizeof(f
    memcpy(&buffer->new_state_m[id * buffer->input_shape], state_, buffer->input_shape * siz
    memcpy(&buffer->action_m[id * buffer->n_actions], action, buffer->n_actions * sizeof(flo
    buffer->reward_m[id] = reward;
    buffer->terminal_m[id] = done;

    buffer->counter++;
}

// Function to sample a batch of transitions
void sample_batch(ReplayBuffer *buffer, int batch_size, float *states, float *actions,
                  float *rewards, float *states_, bool *dones) {
    int max_m = buffer->counter < buffer->max_size ? buffer->counter : buffer->max_size;

    for (int i = 0; i < batch_size; i++) {
        int index = rand() % max_m;

        memcpy(&states[i * buffer->input_shape], &buffer->state_m[index * buffer->input_shap
        memcpy(&states_[i * buffer->input_shape], &buffer->new_state_m[index * buffer->input
        memcpy(&actions[i * buffer->n_actions], &buffer->action_m[index * buffer->n_actions]

        rewards[i] = buffer->reward_m[index];
        dones[i] = buffer->terminal_m[index];
    }
}

// Function to free allocated memory
void free_replay_buffer(ReplayBuffer *buffer) {
    free(buffer->state_m);
    free(buffer->new_state_m);
    free(buffer->action_m);
    free(buffer->reward_m);
    free(buffer->terminal_m);
    free(buffer);
}
```

```c
typedef struct {
    int input_size;
    int output_size;
    float *weights;
    float *bias;
} DenseLayer;

// Function to initialize a dense layer
DenseLayer* create_dense_layer(int input_size, int output_size) {
    DenseLayer *layer = (DenseLayer*)malloc(sizeof(DenseLayer));
    layer->input_size = input_size;
    layer->output_size = output_size;
    layer->weights = (float*)malloc(input_size * output_size * sizeof(float));
    layer->bias = (float*)malloc(output_size * sizeof(float));

    // Randomly initialize weights and biases (for simplicity, set to small values)
    for (int i = 0; i < input_size * output_size; i++) {
        layer->weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < output_size; i++) {
        layer->bias[i] = 0;
    }

    return layer;
}

// ReLU activation function
void relu_dense(float *array, int size) {
    for (int i = 0; i < size; i++) {
        if (array[i] < 0) {
            array[i] = 0;
        }
    }
}

// Forward pass through a dense layer
void forward_dense(DenseLayer *layer, float *input, float *output) {
    for (int i = 0; i < layer->output_size; i++) {
        output[i] = layer->bias[i];
        for (int j = 0; j < layer->input_size; j++) {
            output[i] += input[j] * layer->weights[i * layer->input_size + j];
        }
    }
    relu_dense(output, layer->output_size);
}
```

```c
// Critic Network
typedef struct {
    DenseLayer *fc1;
    DenseLayer *fc2;
    DenseLayer *q;
} CriticNetwork;

// Initialize the Critic Network
CriticNetwork* create_critic_network(int state_dim, int action_dim) {
    CriticNetwork *net = (CriticNetwork*)malloc(sizeof(CriticNetwork));
    net->fc1 = create_dense_layer(state_dim + action_dim, 256);
    net->fc2 = create_dense_layer(256, 256);
    net->q = create_dense_layer(256, 1);
    return net;
}

// Forward pass for Critic Network
float forward_critic(CriticNetwork *net, float *state, float *action, int state_dim, int act
    float input[state_dim + action_dim];
    memcpy(input, state, state_dim * sizeof(float));
    memcpy(input + state_dim, action, action_dim * sizeof(float));

    float hidden1[256], hidden2[256], q_value[1];
    forward_dense(net->fc1, input, hidden1);
    forward_dense(net->fc2, hidden1, hidden2);
    forward_dense(net->q, hidden2, q_value);

    return q_value[0];
}

// Value Network
typedef struct {
    DenseLayer *fc1;
    DenseLayer *fc2;
    DenseLayer *v;
} ValueNetwork;

// Initialize the Value Network
ValueNetwork* create_value_network(int state_dim) {
    ValueNetwork *net = (ValueNetwork*)malloc(sizeof(ValueNetwork));
    net->fc1 = create_dense_layer(state_dim, 256);
    net->fc2 = create_dense_layer(256, 256);
    net->v = create_dense_layer(256, 1);
    return net;
}
```

```c
// Forward pass for Value Network
float forward_value(ValueNetwork *net, float *state) {
    float hidden1[256], hidden2[256], v_value[1];
    forward_dense(net->fc1, state, hidden1);
    forward_dense(net->fc2, hidden1, hidden2);
    forward_dense(net->v, hidden2, v_value);

    return v_value[0];
}

// Free memory for a dense layer
void free_dense_layer(DenseLayer *layer) {
    free(layer->weights);
    free(layer->bias);
    free(layer);
}

// Free memory for the networks
void free_critic_network(CriticNetwork *net) {
    free_dense_layer(net->fc1);
    free_dense_layer(net->fc2);
    free_dense_layer(net->q);
    free(net);
}

void free_value_network(ValueNetwork *net) {
    free_dense_layer(net->fc1);
    free_dense_layer(net->fc2);
    free_dense_layer(net->v);
    free(net);
}


typedef struct {
    int fc1_dims;
    int fc2_dims;
    int n_actions;
    float max_action;
    float noise;

    float *fc1_weights;
    float *fc2_weights;
    float *mu_weights;
    float *sigma_weights;
} ActorNetwork;
```

```c
// Function to initialize the ActorNetwork
ActorNetwork* create_actor_network(float max_action, int fc1_dims, int fc2_dims, int n_actio
    ActorNetwork *net = (ActorNetwork*)malloc(sizeof(ActorNetwork));
    net->fc1_dims = fc1_dims;
    net->fc2_dims = fc2_dims;
    net->n_actions = n_actions;
    net->max_action = max_action;
    net->noise = 1e-6;

    // Allocate memory for weights and initialize them
    net->fc1_weights = (float*)calloc(fc1_dims * fc2_dims, sizeof(float));
    net->fc2_weights = (float*)calloc(fc2_dims * n_actions, sizeof(float));
    net->mu_weights = (float*)calloc(n_actions, sizeof(float));
    net->sigma_weights = (float*)calloc(n_actions, sizeof(float));

    // Initialize weights with small random values
    for (int i = 0; i < fc1_dims * fc2_dims; i++) {
        net->fc1_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < fc2_dims * n_actions; i++) {
        net->fc2_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
    }
    for (int i = 0; i < n_actions; i++) {
        net->mu_weights[i] = ((float)rand() / RAND_MAX) * 0.01;
        net->sigma_weights[i] = 1.0; // Initial sigma set to 1 for exploration
    }

    return net;
}


// Simple ReLU activation function
void relu(float *input, int size) {
    for (int i = 0; i < size; i++) {
        if (input[i] < 0) input[i] = 0;
    }
}


// Forward pass
void forward(ActorNetwork *net, float *state, float *mu, float *sigma) {
    float fc1_output[net->fc2_dims];
    float fc2_output[net->n_actions];

    // Fully connected layer 1
    for (int i = 0; i < net->fc2_dims; i++) {
        fc1_output[i] = 0;
        for (int j = 0; j < net->fc1_dims; j++) {
            fc1_output[i] += state[j] * net->fc1_weights[i * net->fc1_dims + j];
        }
    }
    relu(fc1_output, net->fc2_dims);

    // Fully connected layer 2
    for (int i = 0; i < net->n_actions; i++) {
        fc2_output[i] = 0;
        for (int j = 0; j < net->fc2_dims; j++) {
            fc2_output[i] += fc1_output[j] * net->fc2_weights[i * net->fc2_dims + j];
        }
    }
    relu(fc2_output, net->n_actions);

    // Compute mu and sigma
    for (int i = 0; i < net->n_actions; i++) {
        mu[i] = net->max_action * tanh(fc2_output[i]);  // Bound actions
        sigma[i] = fmax(net->noise, fmin(net->sigma_weights[i], 1.0)); // Ensure sigma is po
    }
}


// Function to sample an action from a normal distribution
void sample_normal(ActorNetwork *net, float *state, float *action, float *log_probs) {
    float mu[net->n_actions];
    float sigma[net->n_actions];

    forward(net, state, mu, sigma);

    for (int i = 0; i < net->n_actions; i++) {
        // Sample from normal distribution using Box-Muller transform
        float epsilon = ((float)rand() / RAND_MAX) * 2 - 1;
        action[i] = mu[i] + sigma[i] * epsilon;

        // Compute log probability only if log_probs is not NULL
        if (log_probs != NULL) {
            float exponent = -0.5 * pow((action[i] - mu[i]) / sigma[i], 2);
            log_probs[i] = exponent - log(sigma[i]) - log(sqrt(2 * M_PI));
        }
    }
}
```

```c
// Function to free memory
void free_actor_network(ActorNetwork *net) {
    free(net->fc1_weights);
    free(net->fc2_weights);
    free(net->mu_weights);
    free(net->sigma_weights);
    free(net);
}


typedef struct {
    float gamma;
    float tau;
    int batch_size;
    int n_actions;
    float scale;

    ReplayBuffer *memory;
    ActorNetwork *actor;
    CriticNetwork *critic_1;
    CriticNetwork *critic_2;
    ValueNetwork *value;
    ValueNetwork *target_value;
} Agent;

// Function to create an agent
Agent* create_agent(float alpha, float beta, int input_dims, int n_actions,
                    int max_size, float tau, int layer1_size, int layer2_size,
                    int batch_size, float reward_scale) {
    Agent *agent = (Agent*)malloc(sizeof(Agent));
    agent->gamma = 0.99;
    agent->tau = tau;
    agent->batch_size = batch_size;
    agent->n_actions = n_actions;
    agent->scale = reward_scale;

    // Create networks
    agent->memory = create_replay_buffer(max_size, input_dims, n_actions);
    agent->actor = create_actor_network(1.0, layer1_size, layer2_size, n_actions);
    agent->critic_1 = create_critic_network(input_dims , n_actions);
    agent->critic_2 = create_critic_network(input_dims, n_actions);
    agent->value = create_value_network(input_dims);
    agent->target_value = create_value_network(input_dims);

    return agent;
```

```
}

// Function to choose action
void choose_action(Agent *agent, float *observation, float *action) {
    sample_normal(agent->actor, observation, action, NULL);
}

// Function to store experience
void remember(Agent *agent, float *state, float *action, float reward, float *new_state, boo
    store_transition(agent->memory, state, action, reward, new_state, done);
}

// Update target network parameters
void update_network_parameters(Agent *agent, float tau) {
    if (tau == 0) {
        tau = agent->tau;
    }

    for (int i = 0; i < 256; i++) {  // Assuming 256 weights
        agent->target_value->v->weights[i] = tau * agent->value->v->weights[i] +
                                             (1 - tau) * agent->target_value->v->weights[i]
    }
}


// Learning function
void learn(Agent *agent) {
    if (agent->memory->counter < agent->batch_size) {
        return;
    }

    float states[agent->batch_size * 8];  // Flattened array instead of 2D array
    float new_states[agent->batch_size * 8];
    float actions[agent->batch_size * agent->n_actions];
    float rewards[agent->batch_size];
    bool dones[agent->batch_size];

    sample_batch(agent->memory, agent->batch_size, states, actions, rewards, new_states, do

    update_network_parameters(agent, 0);
}

// Free memory
void free_agent(Agent *agent) {
    free_replay_buffer(agent->memory);
    free_actor_network(agent->actor);
    free_critic_network(agent->critic_1);
    free_critic_network(agent->critic_2);
    free_value_network(agent->value);
    free_value_network(agent->target_value);
    free(agent);
}

typedef struct {
    int observation_space_dim;
    int action_space_dim;
    float reward_range_min;
    float reward_range_max;
} Env;

Env* create_env() {
    Env *env = (Env*)malloc(sizeof(Env));
    env->observation_space_dim = 8; // Assuming 8 observations
    env->action_space_dim = 2;      // Assuming 2 actions
    env->reward_range_min = -100;
    env->reward_range_max = 100;
    return env;
}

void free_env(Env *env) {
    free(env);
}

// Placeholder function for resetting the environment
void reset_env(Env *env, float *observation) {
    for (int i = 0; i < env->observation_space_dim; i++) {
        observation[i] = ((float)rand() / RAND_MAX) * 2 - 1;  // Random values between -1 a
    }
}

// Placeholder function for stepping in the environment
void step_env(Env *env, float *action, float *observation, float *reward, bool *done) {
    for (int i = 0; i < env->observation_space_dim; i++) {
        observation[i] = ((float)rand() / RAND_MAX) * 2 - 1;
    }
    *reward = ((float)rand() / RAND_MAX) * (env->reward_range_max - env->reward_range_min) -
    *done = rand() % 10 == 0; // Randomly end episodes
}

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
```

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

clock_t start, end;
double cpu_time_used;
if (rank == 0) start = clock();

Env *env = create_env();
Agent *agent = create_agent(0.0003, 0.0003, env->observation_space_dim, env->action_spac
                            1000000, 0.005, 256, 256, 256, 2);

int n_games = 10000 / size;
float best_score = env->reward_range_min;
float score_history[100] = {0};
int history_size = 0;

for (int i = 0; i < n_games; i++) {
    float observation[env->observation_space_dim];
    reset_env(env, observation);

    bool done = false;
    float score = 0;

    while (!done) {
        float action[env->action_space_dim];
        choose_action(agent, observation, action);

        float new_observation[env->observation_space_dim];
        float reward;
        step_env(env, action, new_observation, &reward, &done);

        score += reward;
        remember(agent, observation, action, reward, new_observation, done);
        learn(agent);

        memcpy(observation, new_observation, sizeof(new_observation));
    }

    if (history_size < 100) {
        score_history[history_size++] = score;
    } else {
        for (int j = 1; j < 100; j++) {
            score_history[j - 1] = score_history[j];
        }
        score_history[99] = score;
```

```
            }

            float avg_score = 0;
            for (int j = 0; j < history_size; j++) {
                avg_score += score_history[j];
            }
            avg_score /= history_size;

            if (avg_score > best_score) best_score = avg_score;

            if (i % 100 == 0)
                printf("Rank %d | Episode %d | Score: %.1f | Avg Score: %.1f\n", rank, i, score
        }

        // Reduce to find best score across all ranks
        float global_best;
        MPI_Reduce(&best_score, &global_best, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);

        free_agent(agent);
        free_env(env);

        if (rank == 0) {
            end = clock();
            cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
            printf("Best Score from all processes: %.1f\n", global_best);
            printf("Time taken: %f seconds\n", cpu_time_used);
        }

        MPI_Finalize();
        return 0;
    }
```

6 Time Analysis and Performance Comparison

This section compares the time efficiency and performance of different implementations of the reinforcement learning agent, measured in terms of execution time, best achieved score, and speedup over the original Python implementation.

- **Python Implementation (Baseline):**
  - **Episodes:** 100
  - **Time Taken:** 2343.012 seconds
  - **Best Score:** 112.9
  - **Speedup:** 1.00× (Baseline)

- **C Implementation:**
  - **Episodes:** 10,000
  - **Time Taken:** 242.74 seconds
  - **Best Score:** 118.1
  - **Speedup (normalized to 100 episodes):**

$$\text{Speedup} = \frac{2343.012}{\frac{242.74}{10000} \times 100} \approx 9.65\times$$

  - **Inference:** Significant acceleration is achieved as the C version internalizes the environment and avoids unnecessary simulation overhead, leading to reduced latency and better cache efficiency.
- **OpenMP-Parallelized C Implementation:**
  - **Episodes:** 10,000
  - **Time Taken:** 187.83 seconds (8 threads used from threads vs speed up in tutorial 7)
  - **Best Score:** 117.6
  - **Speedup:**

$$\text{Speedup} = \frac{2343.012}{\frac{187.83}{10000} \times 100} \approx 12.47\times$$

  - **Inference:** Parallelizing compute-intensive functions (`forward_dense`, `forward`) improved execution time further. However, the gain is bounded by Amdahl's Law and non-parallelizable code.
- **CUDA Implementation:**
  - **Episodes:** 1,000
  - **Time Taken:** 716.7 seconds
  - **Best Score:** 123.4
  - **Speedup:**

$$\text{Speedup} = \frac{2343.012}{\frac{716.7}{1000} \times 100} \approx 3.27\times$$

  - **Inference:** GPU acceleration improves throughput for batch processing. The limited speedup may be due to memory transfer overheads and suboptimal kernel configuration.
- **MPI Implementation:**
  - **Episodes:** 10,000
  - **Time Taken:** 41.2 seconds (2 processors)
  - **Best Score:** 128.8

– **Speedup:**

$$\text{Speedup} = \frac{2343.012}{\frac{41.2}{10000} \times 100} \approx 56.88\times$$

– **Inference:** MPI shows the highest speedup, demonstrating the benefit of distributed computing. Ideal for environments where multiple agents or episodes are processed in parallel across different nodes or cores.

General Observations

- Transitioning from Python to C yields immediate performance improvements due to lower-level memory management and faster execution.
- OpenMP enables moderate parallel speedup by leveraging multi-threading within a single machine.
- CUDA is effective but may require tuning and efficient batching to outperform CPU-bound solutions in reinforcement learning environments.
- MPI offers the best scalability and performance when multi-process execution is feasible, particularly in compute clusters or multi-core setups.