1. platform.ini

```
[env:firebeetle32]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
```

2. AP_server_version.cpp

```
#include <Arduino.h>
#include <WiFi.h>

const char* ssid = "MyESP32AP";
const char* password = "12345678";
WiFiServer server(8080);
HardwareSerial MySerial(2); // second port
const int bufSize = 30;
char buf[bufSize];
int buf_index = 0;
String cmd_str;
WiFiClient client;

void command();
void getHip_INFO();
void handleCommandsTask(void* pvParameters);
void getAndSendHipInfoTask(void* pvParameters);

void setup() {
  // Init
  Serial.begin(115200);
  delay(1000);
  IPAddress IP(192, 168, 4, 1);
  IPAddress gateway(192, 168, 4, 1);
  IPAddress subnet(255, 255, 255, 0);
  WiFi.softAPConfig(IP, gateway, subnet);
  //Connecting
  WiFi.softAP(ssid, password);
  Serial.print("AP IP address: ");
  Serial.println(WiFi.softAPIP());
  server.begin();
```

```
  MySerial.begin(115200, SERIAL_8N1, 16, 17);
  Serial.println("Serial at pin is ready!");

  xTaskCreatePinnedToCore(handleCommandsTask, "HandleCommands", 10000, NULL, 1,
NULL, 0); // 核心 0
  xTaskCreatePinnedToCore(getAndSendHipInfoTask, "GetAndSendHipInfo", 10000, NULL,
1, NULL, 1); // 核心 1
  Serial.println("Multi-core ready!");
}

void loop() {
  if (!client || !client.connected()) {
    client = server.available();
    if (client) {
      Serial.println("Client connected!");
    }
  }
  vTaskDelay(pdMS_TO_TICKS(1000));
}

void handleCommandsTask(void* pvParameters) {
  while (true) {
    if (client && client.connected()) {
      command();
    }
    delay(10); // 避免 CPU 佔用率過高
  }
}

void getAndSendHipInfoTask(void* pvParameters) {
  while (true) {
    if (client && client.connected()) {
      getHip_INFO();
    }
    delay(10); // 避免 CPU 佔用率過高
  }
}
```

```
void command() {
    while (client.available()) {
        char c = client.read();
        if (c == '\0') {
            if (cmd_str.length() > 0) {
                Serial.print("received data from PC: ");
                MySerial.print(cmd_str);
                Serial.println(cmd_str);
                // client.write((const uint8_t *)cmd_str.c_str(), cmd_str.length());
                cmd_str = "";
            }
            }
        else {
            cmd_str += c;
        }
    }
}

void getHip_INFO() {
  const int bufferSize = 1024;
  static char buffer[bufferSize];
  static int index = 0;

  while (MySerial.available()) {
    buffer[index] = MySerial.read();
    if (buffer[index] == '\n' || index == bufferSize - 2) {
      buffer[index + 1] = '\0';
      // Serial.println("From HIP: ");
      Serial.println(buffer);
      client.println(buffer);
      index = 0;
      break;
    } else {
      index++;
    }
  }
}
```

3. client_order.py:

```
import socket
import numpy as np
from EMG import emg_nonasync

def analysis(data):
    result = []
    if data.startswith("X"):
        parts = data[1:].strip().split()
        count = 0
        for part in parts:
            if count == 9:
                break
            clean_part = ''.join(filter(lambda x: x in '0123456789.-', part))
            if clean_part and clean_part != '-' and not clean_part.endswith('.'):
                try:
                    result.append(float(clean_part))
                    count += 1
                except ValueError as e:
                    print(f"Error converting '{clean_part}' to float: {e}")
                    continue

        if len(result) == 9:
            return np.array(result), True
    # print(f"Failed to analyze data: {data}")
    return np.zeros(9), False

def FREEX_CMD(sock, mode1="E", value1="0", mode2="E", value2="0"):
    cmd_str = f"X {mode1} {value1} {mode2} {value2}\r\n\0"
    cmd_bytes = cmd_str.encode('ascii')
    try:
        sock.send(cmd_bytes)
    except Exception as e:
        FREEX_CMD(sock, "E", "0", "E", "0")
        print(f"Error when sending: {e}")

def connect_FREEX(host='192.168.4.1', port=8080):
```

```
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    print(f"Successfully connected to {host}:{port}")
    return sock

def read_line(sock):
    try:
        data = sock.recv(1024)
        if not data:
            return None
        data = data.decode('ascii').rstrip('\r\n\0')
        return data
    except Exception as e:
        FREEX_CMD(sock, "E", "0", "E", "0")
        print(f"Error when reading_line: {e}")
        return None

def get_INFO(sock, uri, bp_parameter, nt_parameter, lp_parameter):
    while True:
        info = read_line(sock)
        if info is None or info == "":
            FREEX_CMD(sock, "E", "0", "E", "0")
            print("stucking in EXO data failed")
            continue
        # print("raw_data: ", info)
        analyzed_data, is_analyzed = analysis(info)
        if is_analyzed:
            break
        else:
            FREEX_CMD(sock, "E", "0", "E", "0")
    # print("analyzed: ", analyzed_data)
    # analyzed_data = np.random.rand(9)
    # emg
    emg_observation, bp_parameter, nt_parameter, lp_parameter =
emg_nonasync.read_specific_data_from_websocket(uri ,bp_parameter, nt_parameter,
lp_parameter)

    return analyzed_data, emg_observation, bp_parameter, nt_parameter, lp_parameter
```

```python
def if_not_safe(limit, angle, speed):
    # if (angle >= limit and speed > 0) or (angle <= -limit and speed < 0):
    if (angle >= limit) or (angle <= -limit):
        return True
    else:
        return False

last_action_was_zero = False
left_disabled = False
right_disabled = False

def send_action_to_exoskeleton_speed(writer, action, state):
    global last_action_was_zero
    action[0] *= 10000  # Scale the action for the right side
    action[1] *= 10000  # Scale the action for the left side
    LIMIT = 10
    CURRENT_LIMIT = 50000
    R_angle, L_angle = state[0], state[3]
    R_current, L_current = state[2], state[5]
    current_action_is_zero = all(a == 0 for a in action)

    if current_action_is_zero and last_action_was_zero:
        return
    # print(f"action: {action}, angle: {R_angle}, {L_angle}, current: {R_current},
{L_current}")
    check_R = if_not_safe(LIMIT, R_angle, action[0])
    check_L = if_not_safe(LIMIT, L_angle, action[1])

    if check_R and check_L:
        # print("both actions aborted due to safety")
        FREEX_CMD(writer, "E", "0", "E", "0")
    elif check_R:
        # print("Right action aborted due to safety")
        FREEX_CMD(writer, "E", "0", 'C', f"{action[1]}" if not check_L else "0")
    elif check_L:
        # print("Left action aborted due to safety")
        FREEX_CMD(writer, 'C', f"{action[0]}" if not check_R else "0", "E", "0")
```

```
    else:
        FREEX_CMD(writer, 'C', f"{action[0]}", 'C', f"{action[1]}")

    last_action_was_zero = current_action_is_zero
    print("----------------------------")
def send_action_to_exoskeleton(writer, action, state, control_type='speed'):
    if control_type == 'speed':
        return send_action_to_exoskeleton_speed(writer, action, state)
    elif control_type == 'disable':
        pass
    else:
        raise ValueError("Unknown control_type specified.")
```

4. emg_nonasync.py

```
import numpy as np
from scipy.signal import butter, lfilter, iirnotch, lfilter_zi
import websocket
import json

def read_specific_data_from_websocket(uri, bp_parameter, nt_parameter,
lp_parameter):
        try:
            ws = websocket.WebSocket()
            ws.connect(uri)
            while True:
                data = ws.recv()
                emg_array, bp_parameter, nt_parameter, lp_parameter =
process_data_from_websocket(data, bp_parameter, nt_parameter, lp_parameter)
                if emg_array.shape[0] != 0:
                    return emg_array, bp_parameter, nt_parameter, lp_parameter
        except Exception as e:
            print(f"WebSocket error: {e}")
            pass
        # finally:
        #     ws.close()

def process_data_from_websocket(data, bp_parameter, nt_parameter, lp_parameter):
```

```
    emg_values = np.zeros((8,50))
    j = 0
    try:
        data_dict = json.loads(data)
        if "contents" in data_dict:
            # 提取 serial_number 和 eeg 的值
            serial_numbers_eegs = [(item['serial_number'][0], item['eeg']) for item
in data_dict['contents']]
            # 輸出結果
            for serial_number, eeg in serial_numbers_eegs:
                # print(f"Serial Number: {serial_number}, EEG: {eeg}")
                for i in range(8):
                    emg_values[i,j] = eeg[i]      # 最新的 50 筆 emg 資料
                j+=1
            try:
                emg_array = np.empty((8, 50))
                for k in range(8):
                    #print("check2",emg_values[k],bp_parameter[k], nt_parameter[k],
lp_parameter[k])
                    emg_array[k], bp_parameter[k], nt_parameter[k], lp_parameter[k] =
process_emg_signal(emg_values[k],bp_parameter[k], nt_parameter[k], lp_parameter[k])
                    #print("check5",emg_values[k],bp_parameter[k], nt_parameter[k],
lp_parameter[k])
                return emg_array, bp_parameter, nt_parameter, lp_parameter
            except Exception as e:
                print(f"處理信號時發生錯誤: {e}")
                return np.array([]), bp_parameter, nt_parameter, lp_parameter
    except json.JSONDecodeError:
        print("Failed to decode JSON from WebSocket")
    except Exception as e:
        # print(f"Error processing data from WebSocket: {e}")
        return np.array([]), bp_parameter, nt_parameter, lp_parameter

# 帶通濾波器設計
def bandpass_filter(data, lowcut, highcut, fs, bp_filter_state, order=4):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
```

```
    b, a = butter(order, [low, high], btype='band')
    if bp_filter_state.all() == 0:
        bp_filter_state = lfilter_zi(b, a)
        #print("check4", bp_filter_state)
    y, bp_filter_state = lfilter(b, a, data, zi=bp_filter_state)
    return y, bp_filter_state

# 陷波濾波器設計
def notch_filter(data, notch_freq, fs, notch_filter_state, quality_factor=30):
    nyq = 0.5 * fs
    freq = notch_freq / nyq
    b, a = iirnotch(freq, quality_factor)
    if notch_filter_state.all() == 0:
        notch_filter_state = lfilter_zi(b, a)
        #print("check5", notch_filter_state)
    y, notch_filter_state = lfilter(b, a, data, zi=notch_filter_state)
    return y, notch_filter_state

# 全波整流
def full_wave_rectification(data):
    return np.abs(data)

# 低通濾波器設計（提取包絡）
def lowpass_filter(data, cutoff, fs, lp_filter_state, order=4):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    if lp_filter_state.all() == 0:
        lp_filter_state = lfilter_zi(b, a)
        #print("check8", lp_filter_state)
    y, lp_filter_state = lfilter(b, a, data, zi=lp_filter_state)
    return y, lp_filter_state

# 即時信號處理函數
def process_emg_signal(data, bp_parameter, nt_parameter, lp_parameter, fs=1000):
    # 帶通濾波
    bandpassed, bp_parameter = bandpass_filter(data, 20, 450, fs, bp_parameter)
    # 50Hz 陷波濾波
```

```
    notch_filtered, nt_parameter = notch_filter(bandpassed, 50, fs, nt_parameter)
    # 全波整流
    rectified = full_wave_rectification(notch_filtered)
    # 低通濾波提取包絡
    enveloped, lp_parameter = lowpass_filter(rectified, 10, fs, lp_parameter)

    return enveloped, bp_parameter, nt_parameter, lp_parameter
# 以下為肌力回饋
def calculate_emg_level(data, initial_max_min_rms_values, times,
ta=20,rf=40,bf=25,Ga=15):
    #前 1 秒為暖機
    if times <= 1000:
        return 0, initial_max_min_rms_values
    # 使用第 1 秒到第 10 秒的資料來確定初始的最小、最大 RMS 值
    elif 1000 < times <= 5000:
        for i in range(8):
            rms_values = data[i]
            if initial_max_min_rms_values[i][0] == 0 or rms_values >
initial_max_min_rms_values[i][0]:
                initial_max_min_rms_values[i][0] = rms_values
            elif initial_max_min_rms_values[i][1] == 0 or rms_values <
initial_max_min_rms_values[i][1]:
                initial_max_min_rms_values[i][1] = rms_values
        return 0, initial_max_min_rms_values
    #每 0.05 秒傳出 reward 值
    else:
        reward = np.zeros(8)
        y = 0
        for i in range(8):
            rms_values = data[i]
            reward[i] = map_to_levels(rms_values, initial_max_min_rms_values[i])
        y =
ta*reward[0]+rf*reward[1]+bf*reward[2]+Ga*reward[3]+ta*reward[4]+rf*reward[5]+bf*rew
ard[6]+Ga*reward[7]
        print("Total: ",y/200,"Reward: ",reward)
        return y/200, initial_max_min_rms_values

def calculate_rms(signal):
```

```
    "計算訊號的 RMS 值。"
    return np.sqrt(np.mean(signal**2))

def map_to_levels(value, max_min_rms_values):
    """將值映射到超出 5 到-5 級的線性值上，基於放鬆閾值和初始最大 RMS 值，
    但在上下限內分為 5 到-5 十個等級區間。"""
    # 計算每個等級的值範圍大小
    try:
        level_range = (max_min_rms_values[0] - max_min_rms_values[1]) / 10

        if value <= max_min_rms_values[1]:
            # 計算低於 min_rms_values 的值應映射到哪個級別
            level_diff = (max_min_rms_values[1] - value) / level_range
            return 5 + round(level_diff)
        elif value >= max_min_rms_values[0]:
            # 計算高於 max_rms_values 的值應映射到哪個級別
            level_diff = (value - max_min_rms_values[0]) / level_range
            return -5 - round(level_diff)
        else:
            # 線性映射到 5 到-5
            normalized_value = (value - max_min_rms_values[1]) /
(max_min_rms_values[0] - max_min_rms_values[1])
            return int(round(normalized_value * (-10))) + 5
    except Exception as e:
        print(f"計算 reward 發生錯誤: {e}, return 0")
        return 0
```

5. Env.py

```
from wifi_streaming import client_order
from EMG import emg_nonasync
import asyncio
import gym
from gym import spaces
import numpy as np
from tensorboardX import SummaryWriter
import time
import keyboard
```

```
channel_names = [
    'Tibialis_anterior_right',  # 通道 1: 右腿脛前肌
    'Rectus Femoris_right',     # 通道 2: 右腿股直肌
    'Biceps_femoris_right',     # 通道 3: 右腿股二頭肌
    'Gastrocnemius_right',      # 通道 4: 右腿腓腸肌
    'Tibialis_anterior_left',   # 通道 5: 左腿脛前肌
    'Rectus Femoris_left',      # 通道 6: 左腿股直肌
    'Biceps_femoris_left',      # 通道 7: 左腿股二頭肌
    'Gastrocnemius_left'        # 通道 8: 左腿腓腸肌
]


class ExoskeletonEnv(gym.Env):
    metadata = {'render.modes': ['human']}

    def __init__(self, log_writer , device='cpu', host='192.168.4.1', url=
"ws://localhost:31278/ws", port=8080):
        super(ExoskeletonEnv, self).__init__()
        self.device = device
        self.host = host
        self.port = port
        self.uri = url
        self.observation = np.zeros(9)
        self.emg_observation = np.zeros(8)
        self.filtered_emg_observation = np.zeros((8,50))
        self.bp_parameter = np.zeros((8,8))
        self.nt_parameter = np.zeros((8,2))
        self.lp_parameter = np.zeros((8,4))
        self.initial_max_min_rms_values = np.zeros((8,2))
        self.current_step = 0
        self.init_time = 0
        self.reward = 0
        self.sock = client_order.connect_FREEX(self.host, self.port)
        self.observation_space = spaces.Box(low=-np.inf, high=np.inf, shape=(15,),
dtype=np.float32)
        self.action_space = spaces.Box(low=-1, high=1, shape=(2,), dtype=np.float32)
        self.log_writer = log_writer

    def step(self, action):
```

```
        # 改回用 send_action_to_exoskeleton_speed 函數
        self.observation, self.filtered_emg_observation, self.bp_parameter,
self.nt_parameter, self.lp_parameter = client_order.get_INFO(self.sock,
self.uri ,self.bp_parameter, self.nt_parameter, self.lp_parameter)
        #window.update_plot(self.filtered_emg_observation[0])
        self.emg_observation = np.sqrt(np.mean(self.filtered_emg_observation**2,
axis=1))

        client_order.send_action_to_exoskeleton(self.sock, action,
self.observation ,"speed")
        self.reward = self.calculate_reward()
        done = self.check_if_done(self.observation)
        self.current_step += 1
        self.render()
        return np.concatenate([self.observation, self.emg_observation], axis=0),
self.reward, done, {}

    def reset(self, is_recording=True):
        client_order.FREEX_CMD(self.sock, "E", "0", "E", "0")
        time.sleep(1)
        if self.sock is not None:
            self.sock.close()
            self.sock = None
        print("disconnect")
        self.sock= client_order.connect_FREEX(self.host, self.port)
        print("re-connected")
        time.sleep(2)
        client_order.FREEX_CMD(self.sock, "A", "0000", "A", "0000")
        print("reset to angle, be relaxed")
        time.sleep(2)
        client_order.FREEX_CMD(self.sock, "E", "0", "E", "0")
        time.sleep(2)
        self.emg_observation = np.zeros(8)
        self.filtered_emg_observation = np.zeros((8,50))
        self.bp_parameter = np.zeros((8,8))
        self.nt_parameter = np.zeros((8,2))
        self.lp_parameter = np.zeros((8,4))
        if is_recording:
```

```
            self.recoding_for_power_level()
        else:
            self.observation, self.filtered_emg_observation, self.bp_parameter,
self.nt_parameter, self.lp_parameter = client_order.get_INFO(self.sock,
self.uri ,self.bp_parameter, self.nt_parameter, self.lp_parameter)
            self.emg_observation = np.sqrt(np.mean(self.filtered_emg_observation**2,
axis=1))
        print("first data recv")
        return np.concatenate([self.observation, self.emg_observation],
axis=0)  #self.emg_observation 的格式
        # return np.zeros(15)

    def recoding_for_power_level(self):
        input("Press Enter to Reset Muscle Power Level, Please walk naturally for
about 10 seconds...")
        self.initial_max_min_rms_values = np.zeros((8,2))
        self.init_time = 0
        while self.init_time <= 5000:
            self.init_time = self.init_time + 50  #len(new_emg_observation)
            self.observation, self.filtered_emg_observation, self.bp_parameter,
self.nt_parameter, self.lp_parameter = client_order.get_INFO(self.sock,
self.uri ,self.bp_parameter, self.nt_parameter, self.lp_parameter)
            self.emg_observation = np.sqrt(np.mean(self.filtered_emg_observation**2,
axis=1))
            self.calculate_reward()
            if self.init_time % 1000 == 0:
                print("Countdown: ",10 - int(round(self.init_time/1000)))

    def calculate_reward(self):
        reward, self.initial_max_min_rms_values =
emg_nonasync.calculate_emg_level(self.emg_observation,
self.initial_max_min_rms_values, self.init_time)
        return reward

    def check_if_done(self, observation):
        # Implement logic to check if the episode is done
        return False
```

```
    def render(self, mode='human', close=False):
        self.log_writer.add_scalars('Joint/Angle', {'Joint1': self.observation[0],
'Joint2': self.observation[3]}, self.current_step)
        self.log_writer.add_scalars('Joint/Velocity', {'Joint1':
self.observation[1], 'Joint2': self.observation[4]}, self.current_step)
        self.log_writer.add_scalars('Joint/Current', {'Joint1': self.observation[2],
'Joint2': self.observation[5]}, self.current_step)
        self.log_writer.add_scalars('IMU', {'Roll': self.observation[6], 'Pitch':
self.observation[7], 'Yaw':self.observation[8]}, self.current_step)
        self.log_writer.add_scalar('Reward', self.reward, self.current_step)
        filtered_emg_step = self.current_step*50
        for i in range(self.emg_observation.shape[0]):
            for j in range(50):
                self.log_writer.add_scalar(f'Filtered_EMG/{channel_names[i]}',
self.filtered_emg_observation[i][j], filtered_emg_step+j)
            self.log_writer.add_scalar(f'sqrted EMG/Channel_{channel_names[i]}',
self.emg_observation[i], self.current_step)

    def close(self):
        print("closing")
        client_order.FREEX_CMD(self.sock, "A", "0", "A", "0")
        time.sleep(2)
        client_order.FREEX_CMD(self.sock, "E", "0", "E", "0")
        time.sleep(0.05)
        self.sock.close()
        self.log_writer.close()
```

6. models.py

```
import ptan
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
HID_SIZE = 20

class DDPGActor(nn.Module):
```

```
    def __init__(self, obs_size, act_size):
        super(DDPGActor, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),  # 17 features to hidden layer with 20
neurons
            nn.Tanh(),          # tanh activation function for hidden layer
            nn.Linear(20, act_size),   # Hidden layer to 2 output values
        )

    def forward(self, x):
        return self.net(x)

class D4PGCritic(nn.Module):
    def __init__(self, obs_size, act_size,
                 n_atoms, v_min, v_max):
        super(D4PGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )

        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, n_atoms)
        )

        delta = (v_max - v_min) / (n_atoms - 1)
        self.register_buffer("supports", torch.arange(
            v_min, v_max + delta, delta))

    def forward(self, x, a):
        obs = self.obs_net(x)
        return self.out_net(torch.cat([obs, a], dim=1))

    def distr_to_q(self, distr):
```

```
        weights = F.softmax(distr, dim=1) * self.supports
        res = weights.sum(dim=1)
        return res.unsqueeze(dim=-1)

class AgentD4PG(ptan.experience.BaseAgent):
    """
    Agent implementing noisy agent
    """
    def __init__(self, net, device="cpu", epsilon=0.3):
        self.net = net
        self.device = device
        self.epsilon = epsilon

    def __call__(self, states, agent_states):
        states_v = ptan.agent.float32_preprocessor(states)
        states_v = states_v.to(self.device)
        mu_v = self.net(states_v)
        actions = mu_v.data.cpu().numpy()
        actions += self.epsilon * np.random.normal(
            size=actions.shape)
        actions = np.clip(actions, -1, 1)
        return actions, agent_states

def unpack_batch(batch, device="cpu"):
    states, actions, rewards, dones, last_states = [], [], [], [], []
    for exp in batch:
        states.append(exp.state)
        actions.append(exp.action)
        rewards.append(exp.reward)
        dones.append(exp.last_state is None)
        if exp.last_state is None:
            last_states.append(exp.state)
        else:
            last_states.append(exp.last_state)
    states_v = ptan.agent.float32_preprocessor(states).to(device)
    actions_v = ptan.agent.float32_preprocessor(actions).to(device)
    rewards_v = ptan.agent.float32_preprocessor(rewards).to(device)
    last_states_v = ptan.agent.float32_preprocessor(last_states).to(device)
```

```
    dones_t = torch.BoolTensor(dones).to(device)
    return states_v, actions_v, rewards_v, dones_t, last_states_v
```

7. *d4pg_train_sync.py*

```
import os
import ptan
import time
from wifi_streaming import Env
from RL import models
import argparse
from tensorboardX import SummaryWriter
import numpy as np
import threading
from pynput import keyboard
from wifi_streaming import client_order

import torch
import torch.optim as optim
import torch.nn.functional as F

GAMMA = 0.99
BATCH_SIZE = 64
LEARNING_RATE = 1e-3
MOMENTUM = 0.9
REPLAY_SIZE = 100000
REPLAY_INITIAL = 10
REWARD_STEPS = 5 # 3~10

OBSERVATION_DIMS = 9+8
ACTION_DIMS = 2

TEST_ITERS = 160 # determines when training stop for a while
MAX_STEPS_FOR_TEST = 10

Vmax = 10
Vmin = -10
N_ATOMS = 51
DELTA_Z = (Vmax - Vmin) / (N_ATOMS - 1)
```

```
def find_best_model(base_path, subdir):
    """
    Searches for the best model within a specified directory.

    Parameters:
        base_path (str): The base path where models are stored.
        subdir (str): The subdirectory to search for the best model.

    Returns:
        tuple: Contains the path of the best model and its corresponding reward.
               Returns (None, float('-inf')) if no model is found.
    """
    best_reward = float('-inf')  # Initialize the best reward to negative infinity
    best_model_path = None  # Initialize the best model path to None
    search_path = os.path.join(base_path, subdir)  # Full path to search in

    for file in os.listdir(search_path):  # Iterate through each file in the
directory
        if file.startswith("best_") and file.endswith(".dat"):  # Check if file name
matches the pattern
            try:
                reward_str = file.split('_')[1]  # Extract the reward value from the
file name
                reward = float(reward_str)  # Convert the reward string to float
                if reward > best_reward:  # Update best reward and model path if a
better reward is found
                    best_reward = reward
                    best_model_path = os.path.join(search_path, file)
            except ValueError:
                pass  # Ignore files where the reward value cannot be converted to
float

    return best_model_path, best_reward  # Return the best model path and its reward

def test_net(net, env, count=10, device="cpu"):
    rewards = 0.0
    steps = 0
```

```
    obs = env.reset(is_recording=False)
    # while True:
    #         obs_v = ptan.agent.float32_preprocessor([obs]).to(device)
    #         mu_v = net(obs_v)
    #         action = mu_v.squeeze(dim=0).data.cpu().numpy()
    #         action = np.clip(action, -1, 1)
    #         obs, reward, done, _ = env.step(action)
    #         rewards += reward
    #         steps += 1
    #         if done or steps >= MAX_STEPS_FOR_TEST:
    #             print("net test1 finished")
    #             client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
    #             break
    # time.sleep(1)
    for i in range(count-1):
        steps = 0
        rewards = 0.0
        # obs = env.reset(is_recording=False)
        while True:
            obs_v = ptan.agent.float32_preprocessor([obs]).to(device)
            mu_v = net(obs_v)
            action = mu_v.squeeze(dim=0).data.cpu().numpy()
            action = np.clip(action, -1, 1)
            obs, reward, done, _ = env.step(action)
            rewards += reward
            steps += 1
            if done or steps >= MAX_STEPS_FOR_TEST:
                client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
                print(f"net test{i+2} finished")
                break
        time.sleep(1)
    return rewards / count, steps / count


def distr_projection(next_distr_v, rewards_v, dones_mask_t,
                     gamma, device="cpu"):
    # since we can't really computing tensor on cuda with numpy
    next_distr = next_distr_v.data.cpu().numpy()
    rewards = rewards_v.data.cpu().numpy()
```

```
    dones_mask = dones_mask_t.cpu().numpy().astype(np.bool_)
    batch_size = len(rewards)
    proj_distr = np.zeros((batch_size, N_ATOMS), dtype=np.float32)

    for atom in range(N_ATOMS):
        tz_j = np.minimum(Vmax, np.maximum(
            Vmin, rewards + (Vmin + atom * DELTA_Z) * gamma))
        b_j = (tz_j - Vmin) / DELTA_Z
        l = np.floor(b_j).astype(np.int64)
        u = np.ceil(b_j).astype(np.int64)
        eq_mask = u == l
        proj_distr[eq_mask, l[eq_mask]] += \
            next_distr[eq_mask, atom]
        ne_mask = u != l
        proj_distr[ne_mask, l[ne_mask]] += \
            next_distr[ne_mask, atom] * (u - b_j)[ne_mask]
        proj_distr[ne_mask, u[ne_mask]] += \
            next_distr[ne_mask, atom] * (b_j - l)[ne_mask]

    if dones_mask.any():
        proj_distr[dones_mask] = 0.0
        tz_j = np.minimum(Vmax, np.maximum(
            Vmin, rewards[dones_mask]))
        b_j = (tz_j - Vmin) / DELTA_Z
        l = np.floor(b_j).astype(np.int64)
        u = np.ceil(b_j).astype(np.int64)
        eq_mask = u == l
        eq_dones = dones_mask.copy()
        eq_dones[dones_mask] = eq_mask
        if eq_dones.any():
            proj_distr[eq_dones, l[eq_mask]] = 1.0
        ne_mask = u != l
        ne_dones = dones_mask.copy()
        ne_dones[dones_mask] = ne_mask
        if ne_dones.any():
            proj_distr[ne_dones, l[ne_mask]] = (u - b_j)[ne_mask]
            proj_distr[ne_dones, u[ne_mask]] = (b_j - l)[ne_mask]
    return torch.FloatTensor(proj_distr).to(device)
```

```
stop_event = threading.Event()
def on_press(key):
    try:
        if key.char == 'q':
            stop_event.set()
    except AttributeError:
        pass
def start_listening():
    listener = keyboard.Listener(on_press=on_press)
    listener.start()

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--cuda", default=False, action='store_true', help='Enable
CUDA')
    parser.add_argument("-n", "--name", required=True, help="Name of the run")
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")
    start_listening()
    save_path = os.path.join("saves", "d4pg-" + args.name)
    actor_subdir = "actor"
    critic_subdir = "critic"
    os.makedirs(os.path.join(save_path, actor_subdir), exist_ok=True)
    os.makedirs(os.path.join(save_path, critic_subdir), exist_ok=True)

    act_net = models.DDPGActor(OBSERVATION_DIMS, ACTION_DIMS).to(device)
    crt_net = models.D4PGCritic(OBSERVATION_DIMS, ACTION_DIMS, N_ATOMS, Vmin,
Vmax).to(device)

    best_actor_model_path, best_actor_reward = find_best_model(save_path,
actor_subdir)
    best_critic_model_path, best_critic_reward = find_best_model(save_path,
critic_subdir)

    if best_actor_model_path:
        print(f"best actor : {best_actor_model_path}, reward : {best_actor_reward}")
    else:
```

```
        print("No actor NN")

    if best_critic_model_path:
        print(f"best critic：{best_critic_model_path},reward：{best_critic_reward}")
    else:
        print("No critic NN")

    print(act_net)
    print(crt_net)
    tgt_act_net = ptan.agent.TargetNet(act_net)
    tgt_crt_net = ptan.agent.TargetNet(crt_net)

    writer = SummaryWriter(comment="-d4pg_" + args.name)
    env = Env.ExoskeletonEnv(log_writer=writer)
    agent = models.AgentD4PG(act_net, device=device)
    exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent, gamma=GAMMA,
steps_count=REWARD_STEPS)
    buffer = ptan.experience.ExperienceReplayBuffer(exp_source,
buffer_size=REPLAY_SIZE)
    act_opt = optim.SGD(act_net.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)
    crt_opt = optim.Adam(crt_net.parameters(), lr=LEARNING_RATE)

    frame_idx = 0
    best_reward = None
    training_stopped_early = False
    with ptan.common.utils.RewardTracker(writer) as tracker:
        with ptan.common.utils.TBMeanTracker(writer, batch_size=10) as tb_tracker:
            while True:
                if stop_event.is_set():
                    print("Training stopped by user.")
                    training_stopped_early = True
                    if best_reward is not None:
                        current_model_name = "best_%+.3f_%d.dat" % (best_reward,
frame_idx)
                    else:
                        print("you stopped training before any best reward was
achieved.")
                    current_model_path = os.path.join(save_path, current_model_name)
```

```
                    torch.save(act_net.state_dict(), current_model_path)
                    print(f"Current model saved to {current_model_path}")
                    break

                frame_idx += 1
                buffer.populate(1)
                rewards_steps = exp_source.pop_rewards_steps()
                if rewards_steps:
                    rewards, steps = zip(*rewards_steps)
                    tb_tracker.track("episode_steps", steps[0], frame_idx)
                    tracker.reward(rewards[0], frame_idx)

                if len(buffer) < REPLAY_INITIAL:
                    continue
                if len(buffer) == REPLAY_INITIAL:
                    print("Initialization of the buffer is finished, start
training...")
                    client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
                    input("Press Enter to continue...")

                batch = buffer.sample(BATCH_SIZE)
                states_v, actions_v, rewards_v, \
                dones_mask, last_states_v = \
                    models.unpack_batch(batch, device)

                # train critic
                crt_opt.zero_grad()
                crt_distr_v = crt_net(states_v, actions_v)
                last_act_v = tgt_act_net.target_model(
                    last_states_v)
                last_distr_v = F.softmax(
                    tgt_crt_net.target_model(
                        last_states_v, last_act_v), dim=1)
                proj_distr_v = distr_projection(
                    last_distr_v, rewards_v, dones_mask,
                    gamma=GAMMA**REWARD_STEPS, device=device)
                prob_dist_v = -F.log_softmax(
                    crt_distr_v, dim=1) * proj_distr_v
```

```
                critic_loss_v = prob_dist_v.sum(dim=1).mean()
                critic_loss_v.backward()
                crt_opt.step()
                tb_tracker.track("loss_critic", critic_loss_v, frame_idx)

                # train actor
                act_opt.zero_grad()
                cur_actions_v = act_net(states_v)
                crt_distr_v = crt_net(states_v, cur_actions_v)
                actor_loss_v = -crt_net.distr_to_q(crt_distr_v)
                actor_loss_v = actor_loss_v.mean()
                actor_loss_v.backward()
                act_opt.step()
                tb_tracker.track("loss_actor", actor_loss_v,
                                 frame_idx)

                tgt_act_net.alpha_sync(alpha=1 - 1e-3)
                tgt_crt_net.alpha_sync(alpha=1 - 1e-3)

                if frame_idx % TEST_ITERS == 0:
                    client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
                    print("Please prepare for a test phase by changing the
exoskeleton user, if desired.")
                    # input("Press Enter to continue after the user has been changed
and is ready...")
                    ts = time.time()
                    rewards, steps = test_net(act_net, env, count=4, device=device)
                    print("Test done in %.2f sec, reward %.3f, steps %d" % (
                        time.time() - ts, rewards, steps))
                    writer.add_scalar("test_reward", rewards, frame_idx)
                    writer.add_scalar("test_steps", steps, frame_idx)
                    if best_reward is None or best_reward < rewards:
                        if best_reward is not None:
                            print("Best reward updated: %.3f -> %.3f" % (best_reward,
rewards))
                            name = "best_%+.3f_%d.dat" % (rewards, frame_idx)
                            actor_model_path = os.path.join(save_path, "actor", name)
```

```
                            critic_model_path = os.path.join(save_path, "critic",
name)
                            torch.save(act_net.state_dict(), actor_model_path)
                            torch.save(crt_net.state_dict(), critic_model_path)
                        best_reward = rewards
                time.sleep(0.01)
    # except KeyboardInterrupt:
        # print("Training interrupted by keyboard.")

    # finally:
    if best_reward is None:
        print("No best reward achieved during the training.")
    elif training_stopped_early:
        print(f"Training stopped, Best reward achieved: {best_reward:.3f}")
    try:
        env.close()
    except Exception as e:
        print(f"Error while closing resources: {e}")
```

8. SceneController.cpp

```
using System.Collections;
using System.Collections.Generic;
using System.Reflection.Emit;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.XR.Interaction.Toolkit;

[RequireComponent(typeof(ARPlaneManager))]

public class SceneController : MonoBehaviour
{
    [SerializeField]
    private InputActionReference _togglePlanesAction;

    [SerializeField]
    private InputActionReference _leftActivateAction;
```

```
[SerializeField]
private InputActionReference _deleteCharacterAction;

[SerializeField]
private InputActionReference _rightActivateAction;

[SerializeField]
private XRRayInteractor _leftRayInteractor;

[SerializeField]
private GameObject _walker;

[SerializeField]
private GameObject _prefab;

private ARPlaneManager _planeManager;
private ARAnchorManager _anchorManager;
private bool _isVisible = true;
private int _numPlanesAddedOccurred = 0;

private List<ARAnchor> _anchors = new List<ARAnchor>();

private GameObject _currentPrefabInstance; // To keep track of the current instantiated prefab

// Start is called before the first frame update
void Start()
{
    Debug.Log("-> SceneController::Start()");

    _planeManager = GetComponent<ARPlaneManager>();

    if (_planeManager is null)
    {
        Debug.LogError("-> Can't find 'ARPlaneManager' :(");
    }

    _anchorManager = GetComponent<ARAnchorManager>();
```

```
        if (_anchorManager == null)
        {
            Debug.LogError("-> Can't find 'ARAnchorManager'! :(");
        }


        _togglePlanesAction.action.performed += OnTogglePlanesAction;
        _planeManager.planesChanged += OnPlanesChanged;
        _anchorManager.anchorsChanged += OnAnchorsChanged;
        _leftActivateAction.action.performed += OnLeftActivateAction;
        _rightActivateAction.action.performed += OnRightActivateAction;
        _deleteCharacterAction.action.performed += OnDeleteCharacterAction;
    }

    private void OnAnchorsChanged(ARAnchorsChangedEventArgs args)
    {
        // remove any anchors that have been removed outside our control, such as during a session
reset
        foreach (var removedAnchor in args.removed)
        {
            _anchors.Remove(removedAnchor);
            Destroy(removedAnchor.gameObject);
        }
    }

    private void OnLeftActivateAction(InputAction.CallbackContext obj)
    {
        CheckIfRayHitsCollider();
    }

    private void CheckIfRayHitsCollider()
    {
        // Check if the left ray interactor hits something
        if (_leftRayInteractor.TryGetCurrent3DRaycastHit(out RaycastHit hit))
        {
            foreach (var plane in _planeManager.trackables)
            {
                string log = $"ARPlane {plane.trackableId.ToString()}";
```

```
            // Assuming plane.extents represents the bounds of the plane
            string label = plane.classification.ToString();
            if (hit.transform.name == log && label == "Floor")
            {
                // If the hit plane is classified as a floor
                Debug.Log("-> Hit detected on the floor! :-) - name: " + hit.transform.name);

                // If there's already a prefab instance, destroy it
                if (_currentPrefabInstance != null)
                {
                    Destroy(_currentPrefabInstance);
                }

                // Instantiate the prefab at the hit location with the correct upright rotation
                _currentPrefabInstance = Instantiate(_prefab, hit.point, Quaternion.identity);

                //// Add an ARAnchor to the instantiated prefab
                //if (_currentPrefabInstance.GetComponent<ARAnchor>() == null)
                //{
                //    ARAnchor anchor = _currentPrefabInstance.AddComponent<ARAnchor>();
                //    if (anchor != null)
                //    {
                //        Debug.Log("-> CreateAnchoredObject() - anchor added!");
                //        _anchors.Add(anchor);
                //    }
                //    else
                //    {
                //        Debug.LogError("-> CreateAnchoredObject() - anchor is null!");
                //    }
                //}
                break;
            }
        }
    }
    else
    {
        Debug.Log("-> No hit detected!");
    }
```

```
}

private void OnDeleteCharacterAction(InputAction.CallbackContext obj)
{
    if (_currentPrefabInstance != null)
    {
        Debug.Log("Destroying character instance.");
        _currentPrefabInstance.SetActive(false);
    }
    else
    {
        Debug.Log("-> No character!");
    }
}

private void OnRightActivateAction(InputAction.CallbackContext obj)
{
    SpawnGrabbableCube();
}

private void SpawnGrabbableCube()
{
    Debug.Log("--> SceneController::SpawnGrabbableCube()");

    Vector3 spawnPosition;

    // Iterate through each plane found in the scene...
    foreach (var plane in _planeManager.trackables)
    {
        // Detect if the plane is a table, if so, spawn a cube on it
        if (plane.classification == PlaneClassification.Floor)
        {
            spawnPosition = plane.transform.position;
            spawnPosition.y += 0.3f; // Raise the cube a bit above the plane
            Instantiate(_walker, spawnPosition, Quaternion.identity);
        }
    }
}
```

```
// Update is called once per frame
void Update()
{

}

private void OnTogglePlanesAction(InputAction.CallbackContext obj)
{
    _isVisible = !_isVisible;
    float fillAlpha = _isVisible ? 0.3f : 0f;
    float lineAlpha = _isVisible ? 1.0f : 0f;

    Debug.Log("-> OnTogglePlanesAction() - trackables.count: " + _planeManager.trackables.count);

    foreach (var plane in _planeManager.trackables)
    {
        SetPlaneAlpha(plane, fillAlpha, lineAlpha);
    }
}

private void SetPlaneAlpha(ARPlane plane, float fillAlpha, float lineAlpha)
{
    var meshRenderer = plane.GetComponentInChildren<MeshRenderer>();
    var lineRenderer = plane.GetComponentInChildren<LineRenderer>();

    if (meshRenderer != null)
    {
        Color color = meshRenderer.material.color;
        color.a = fillAlpha;
        meshRenderer.material.color = color;
    }

    if (lineRenderer != null)
    {
        // Get the current start and end colors
```

```
        Color startColor = lineRenderer.startColor;
        Color endColor = lineRenderer.endColor;

        // Set the alpha component
        startColor.a = lineAlpha;
        endColor.a = lineAlpha;

        // Apply the new colors with updated alpha
        lineRenderer.startColor = startColor;
        lineRenderer.endColor = endColor;
    }
}

private void OnPlanesChanged(ARPlanesChangedEventArgs args)
{
    if (args.added.Count > 0)
    {
        _numPlanesAddedOccurred++;

        foreach (var plane in _planeManager.trackables)
        {
            PrintPlaneLabel(plane);
        }

        Debug.Log("--> Number of planes: " + _planeManager.trackables.count);
        Debug.Log("--> Num Planes Added Occurred:" + _numPlanesAddedOccurred);
    }
}

private void PrintPlaneLabel(ARPlane plane)
{
    string label = plane.classification.ToString();
    string log = $"Plane ID: {plane.trackableId}, Label: {label}";
    Debug.Log(log);
}

void OnDestroy()
{
```

```
        Debug.Log("--> SceneController::OnDestroy()");
        _togglePlanesAction.action.performed -= OnTogglePlanesAction;
        _planeManager.planesChanged -= OnPlanesChanged;
        _anchorManager.anchorsChanged -= OnAnchorsChanged;
        _leftActivateAction.action.performed -= OnLeftActivateAction;
        _rightActivateAction.action.performed -= OnRightActivateAction;
        _deleteCharacterAction.action.performed -= OnDeleteCharacterAction;
    }

}
```

9. PlayerAnimationController.cpp

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.Animations;
using UnityEngine.XR.Interaction.Toolkit;


public class PlayerController : MonoBehaviour
{
    public InputActionReference toggleWalkActionReference;
    public Animator animator;


    private void Start()
    {
        animator.ResetTrigger("ToggleWalk");
    }

    private void OnEnable()
    {
        toggleWalkActionReference.action.performed += OnToggleWalkPerformed;
        toggleWalkActionReference.action.Enable();
    }

    private void OnDisable()
    {
```

```
        toggleWalkActionReference.action.performed -= OnToggleWalkPerformed;
        toggleWalkActionReference.action.Disable();
    }

    private void OnToggleWalkPerformed(InputAction.CallbackContext context)
    {
        animator.SetTrigger("ToggleWalk");
    }
}
```

10. CharacterMovement_walk.cpp

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
public class CharacterMovement : MonoBehaviour
{
    public Animator animator;
    public float speed = 1.0f;
    private Rigidbody rb;

    private void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    private void FixedUpdate()
    {
        if (animator.GetCurrentAnimatorStateInfo(0).IsName("Walk"))
        {
            rb.MovePosition(transform.position + transform.forward * speed * Time.fixedDeltaTime);
        }
    }
}
```