1. **platform.ini**

[env:firebeetle32]

platform = espressif32

board = esp32doit-devkit-v1

framework = arduino

monitor\_speed = 115200

1. **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++;

    }

  }

}

1. **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

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.")

1. **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\*reward[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

1. **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()

1. **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

1. ***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}")

1. **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;

}

}

1. **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");

}

}

1. **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);

}

}

}