Eötvös Loránd Tudományegyetem

Informatikai Kar

Egyetemi-Vállalati Együttműködési Intézet

Mesterséges Intelligencia Tanszék

Környezet mobil robotok csapatainak megerősítéses tanulásának

megvalósításához

Szerző: Témavezető:

Vörös Döme Gulyás László Csaba Dr.

Programtervező informatikus BSc. Egyetemi Docens, PhD

Budapest, 2024

A képen szöveg, képernyőkép, Betűtípus, levél látható

Automatikusan generált leírás

Tartalom

[**1. Bevezetés** 6](#_Toc196374385)

[**2. Felhasználói dokumentáció** 8](#_Toc196374386)

[**2.1. Rendszerkövetelmények** 8](#_Toc196374387)

[**2.1.1. Hardware követelmények** 8](#_Toc196374388)

[**2.1.2. Program beszerzése** 9](#_Toc196374389)

[**2.2. Telepítés** 9](#_Toc196374390)

[**2.2.1. Program kicsomagolása** 10](#_Toc196374391)

[**2.2.2. Docker telepítése** 10](#_Toc196374392)

[**2.2.3. A program telepítése** 12](#_Toc196374393)

[**2.3. Indítás** 14](#_Toc196374394)

[**2.3.1. A program funkciójának beállítása** 14](#_Toc196374395)

[**2.3.2. Program leállítása** 15](#_Toc196374396)

[**2.3.3. Program törlése** 15](#_Toc196374397)

[**2.4. A program funkciói** 16](#_Toc196374398)

[**2.4.1. Új modell tanítása** 16](#_Toc196374399)

[**2.4.2. Tanítás folytatása** 18](#_Toc196374400)

[**2.4.3. Tanítási adatok kiírása** 20](#_Toc196374401)

[**2.5. Konfig és a program funkcióinak konfigfájljainak felépítése** 22](#_Toc196374402)

[**2.5.1. Új modell tanítása konfigfájljának felépítése** 23](#_Toc196374403)

[**2.5.2. Tanítás folytatása konfigfájljának felépítése** 24](#_Toc196374404)

[**2.5.3. Tanítási adatok kiírása konfigfájljának felépítése** 25](#_Toc196374405)

[**2.6. A létrehozott nyersanyagok és annak kezelése** 25](#_Toc196374406)

[**2.7. A program logolása és annak értelmezése** 27](#_Toc196374407)

[**3.** **Fejlesztői dokumentáció** 29](#_Toc196374408)

[**3.1. Forráskód beszerzése** 29](#_Toc196374409)

[**3.2. A forrás könyvtárszerkezete** 29](#_Toc196374410)

[**3.3. Függőségek** 31](#_Toc196374411)

[**3.3.1. A függőségek beszerzése** 32](#_Toc196374412)

[**3.3.2. A függőségek frissítése** 32](#_Toc196374413)

[**3.4. A forrásmappa felépítése** 32](#_Toc196374414)

[**3.4.1. Fejlesztői konténer felépítése** 33](#_Toc196374415)

[**3.4.2. Fejlesztői Docker konténer felépítése** 35](#_Toc196374416)

[**3.4.3. Környezet felépítése** 37](#_Toc196374417)

[**3.4.4. Robot és a környezet kapcsolata, kommunikációja** 39](#_Toc196374418)

[**3.5. Követelmény-specifikáció** 39](#_Toc196374419)

[**3.5.1. Követelményelemzés** 39](#_Toc196374420)

[**3.5.2. Megvalósíthatósági terv** 40](#_Toc196374421)

[**3.5.3. Nem funkcionális követelmények** 41](#_Toc196374422)

[**3.5.4. Funkcionális követelmények** 41](#_Toc196374423)

[**3.6. Felhasználói esetek** 42](#_Toc196374424)

[**3.6.1. Új modell tanítása felhasználói esetei** 42](#_Toc196374425)

[**3.6.2. Tanítás folytatása felhasználói esetei** 43](#_Toc196374426)

[**3.6.3. Tanítási adatok kiírása felhasználói esetei** 43](#_Toc196374427)

[**3.7. Rendszerarchitektúra** 44](#_Toc196374428)

[**3.7.1. Programnyelv és fejlesztői környezetek** 44](#_Toc196374429)

[**3.7.2. Főbb komponensei a programnak** 44](#_Toc196374430)

[**3.8. A fejlesztői környezet** 45](#_Toc196374431)

[**3.9. Tesztkörnyezet** 46](#_Toc196374432)

[**3.10. A tesztelési terv** 46](#_Toc196374433)

[**3.10.1. Manuális tesztelés** 46](#_Toc196374434)

[**3.10.2. Automatikus tesztelés** 47](#_Toc196374435)

[**3.10.3. Tesztelési eredmények** 51](#_Toc196374436)

[**3.10.4. Tesztek részletes leírása** 52](#_Toc196374437)

[**3.11. Verziókezelés** 53](#_Toc196374438)

[**3.12. A program logolása** 54](#_Toc196374439)

[**3.13. A program konfigolása** 55](#_Toc196374440)

[**4.** **Összefoglalás** 57](#_Toc196374441)

[**4.1. További fejlesztési lehetőségek** 57](#_Toc196374442)

[**5.** **Irodalomjegyzék** 58](#_Toc196374443)

# **1. Bevezetés**

Manapság az egyik legnépszerűbb téma a mesterséges intelligencia, viszont a legtöbb embernek nagyon kevés tudása van arról, hogy mennyi különböző helyen lehet alkalmazni a mesterséges intelligenciát. A legtöbb ember úgy ismeri, mint az eszköz, amely megír neked szövegeket vagy képeket generál, viszont nagyon sok tudományágban már majdnem elengedhetetlen a mesterséges intelligencia. Például az orvostudományban, ahol mesterséges intelligenciát tanítanak arra, hogy különböző betegségeket fedezzen fel képek alapján, például a rákkutatásban nagyon hasznos tud lenni [1], mivel már nagyobb valószínűséggel fogja ő felfedezni a képen a problémát, mint egy ember. A mindennapi életünkben nem is vesszük észre, de vannak olyan helyek, ahol mesterséges intelligenciát használnak évek óta, csak nem gondoltunk róla, hogyan működnek eddig, ezért nem is tudtuk, hogy ezt a technológiát használják a cégek. Ilyen például a különböző online vásárlási platformok vagy streaming szolgáltatások ajánlórendszerei [2], vagy például az email szolgáltatód ahogyan szűri a spam emaileket, ezáltal megelőzve, hogy a kártékony levelek elárasszák a felhasználó postaládáját [3].

A saját véleményem szerint a legizgalmasabb ága a mesterséges intelligenciának a robotika. A tény, hogy robotokat meg tudunk tanítani akármilyen feladatra, az szimplán lenyűgöző. Pár száz évvel ezelőtt el se tudták volna az emberek képzelni, amit manapság a robotok probléma nélkül meg tudnak oldani. A tény, hogy már vannak olyan gyárak, ahol például majdnem teljesen robotok végeznek el minden fizikai feladatot, az elképesztő a számomra. A robotika egyik legnépszerűbb ága, amikor egy adott robotot megkérnek arra, hogy teljesítsen egy feladatot, és ő azt magától megtanulja. Ezt nevezik megerősítéses tanításnak. A lényege a módszernek az, hogy a robot egy ideig végzi a feladatát, majd utána rájön, hogy mit is csinált, és az alapján ad magának jutalmat vagy büntetést, ahhoz képest, hogy mennyire volt közel a cég eléréséhez. Egy idő után pedig a sok visszajelzés után már tudni fogja, hogyan és mikor kell mozogni.

Ennek a fajta tanításnak sok fajta módja van. Például vannak különböző szoftverek, amelyek ilyen szimulálásra vannak kitalálva, ilyen például a Gazebo is, amelyet a későbbiekben fogok használni. Viszont elég programozói tudással akármilyen környezetben tudjuk ezt a technológiát használni, például akár videójátékok esetén is. A robotot beletesszük a kedvenc videójátékunkba és megmondjuk neki, mit kell tudnia csinálni és szépen lassan megtanulja magától, hogyan kell ezt használni. Például ezek már annyira népszerűek, hogy az egyik könyvtár a Stable Baselines 3 már támogatja is, hogy régi Atari játékokat tanítsunk megerősítéses tanítás segítségével [4], viszont akár az interneten nagyon sok példát találunk modern játékoknál is, ahol ezt a tanítást alkalmazzák.

Mégis azt gondolom, hogy a robotok tanítása egy nagyon komplikált feladat. A szakdolgozatom írása során nagyon sok mindent tanultam a témával kapcsolatban és nagyon nem egyszerű téma és teljesen megértem, hogy ha valaki feladná a tanulását, mivel neki túl komplikált. A célom a dolgozattal az lenne, hogy át tudjak adni egy környezetet, amely egy belépőpont lehet valakinek, aki nem teljesen ért még a témához, viszont szeretne többet tanulni vele kapcsolatban.

# **2. Felhasználói dokumentáció**

Ebben a fejezetben szeretném bemutatni, hogy a felhasználónak milyen rendszerrel kell rendelkeznie, hogy a program megfelelően működjön. Ezen kívül bemutatom, hogy a felhasználó hogyan tudja telepíteni, elindítani és használni a programot. Azon kívül bemutatom a programnak a felépítését, funkcióit és működését.

# **2.1. Rendszerkövetelmények**

## **2.1.1. Hardware követelmények**

A program futtatásához Ubuntu Linux rendszerre lesz szüksége a felhasználónak. Alapvetően erre azért van szükség, mivel a komplikált környezeti rendszert csak Ubuntu-ban lehetett biztosítani, ezzel mindig garantálva a folyamatos és rendeltetésszerű futást. Tesztelésem során arra jutottam, hogy a virtuális gépek nem elég erősek egy ilyen szoftver futtatásához. Minimum hardware követelménye a Gazebo alkalmazásnak van, ezek a következőek [5]:

* Processzor (CPU) — Quad Core Intel i5 vagy vele megegyező erejű processzor
* Memória (RAM) — 4 GB vagy több
* Videókártya (GPU) — Minimum egy dedikált videó kártya, legalább 1 GB rammal

Tárhely szempontból annyira tárhelyre van szükségünk amennyi helyet a Docker által létrehozott image fog foglalni. Általában a Docker image ilyen 15-16 GB helyet fog használni, azért én ajánlom, hogy 25 GB helyet hagyjunk szabadon a programnak.



*1. ábra: Docker által létrehozott image mérete*

A Gazebo verzió, amit ebben a szoftverben használunk az csak CPU-t használ, ezzel biztosítva, hogy minél több eszközön lehessen használni a programot, még ha a teljesítmény kicsit gyengül is miatta. Természetesen minél jobb hardware-el dolgozunk, annál jobb lesz a program teljesítménye is, de alapvető teszteléseim során arra jutottam, hogy egy belépőszintű, munkára használatos laptop elegendő lehet a kitűzött feladatok elvégzésére.

## **2.1.2. Program beszerzése**

A programot az általam használt verziókezelőből lehet letölteni, a Github-ról. A következő linkről telepíthető a program:

[*https://github.com/RedDome/MLSzakdoga.git*](https://github.com/RedDome/MLSzakdoga.git)

A linkre rákattintva bekerülünk az egész program könyvtárába, ebben minden megtalálható: wiki, képek, videók, fejlesztői környezet. Nekünk ezután a jobb oldalon lévő kék színű code gombra kell rákattintanunk, ott a legalján lesz egy lehetőség, ahol az lesz kiírva, hogy **„Download ZIP”**, erre rákattintva letöltjük a programot.



*2. ábra: Github letöltési útmutató*

# **2.2. Telepítés**

Ebben a fejezetben bemutatom, hogy a felhasználó hogyan tudja kicsomagolni és telepíteni a letöltött zip projekt fájlt. Ezen kívül bemutatom a Docker telepítését és konfigurálását, amely elengedhetetlen része a programnak.

## **2.2.1. Program kicsomagolása**

Miután beszereztük a programot, azután a tömörített fájlt helyezzük át egy általunk meghatározott könyvtárba. Miután ez megtörtént a következő lépés, amit tennünk kell, hogy kicsomagoljuk a könyvtárat. Ezt egy arra alkalmas szoftver segítségével fogjuk tudni megtenni, mint például a **WinRar** [6] vagy a **7Zip** [7]. Az általunk választott tömörítővel a kitömörítés ide opció használatával kicsomagoljuk az MLSzakdoga.zip fájlt. A kicsomagolt mappában fogjuk megtalálni a programot, beleértve a felhasználói és fejlesztői környezetet is. A két környezet abban fog különbözni, hogy másik Dockerfile-t fogunk használni, ami szabályozza, mit fogunk látni a programból.



*3. ábra: A kicsomagolt MLSzakdoga mappa*

## **2.2.2. Docker telepítése**

A Dockerfile egy olyan fájl a Docker szoftverhez, amely képes arra, hogy virtuális környezetet telepítsünk, ezzel segítsünk a felhasználónak, hogy ne kelljen csomagokat telepítenie, hanem csak ennek telepítésével egy teljes működő alkalmazást kap és nem kell azzal foglalkoznia, hogy minden kicsi függőséget feltelepítsen.

A **Docker** **Linux** rendszerben való futáshoz egy **Ubuntu** rendszerre lesz szükségünk, abból is a 24.10, 24.04, 22.04, 20.04es rendszer ajánlott a Docker problémamentes telepítéséhez. Az első lépés a letöltéshez a Docker hivatalos GPG (Digitális kulcs, amely biztosítja az adatok bizalmasságát és hitelességét) kulcsának hozzáadása:

|  |
| --- |
| # Docker hivatalos GPG kulcsának hozzáadása:  *sudo apt-get update*  *sudo apt-get install ca-certificates curl*  *sudo install -m 0755 -d /etc/apt/keyrings*  *sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc*  *sudo chmod a+r /etc/apt/keyrings/docker.asc* |

Utána a következő lépés a **Docker** adattár hozzáadása a forrásainkhoz, hogy le tudjuk tölteni a megfelelő **Docker** csomagokat:

|  |
| --- |
| # Docker adattár hozzáadása a forrásainkhoz:  *echo \*  *"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \*  *$(. /etc/os-release && echo "${UBUNTU\_CODENAME:-$VERSION\_CODENAME}") stable" | \*  *sudo tee /etc/apt/sources.list.d/docker.list > /dev/null*  *sudo apt-get update* |

A következő lépés a hivatalos csomagok letöltése lesz, ez a lépés több percig is eltarthat:

|  |
| --- |
| # Docker telepítése (több percig is eltarthat):  *sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin* |

Utolsó lépésként, ha mindent jól csináltunk a telepítéssel, ezután teszteljük le a **Docker** környezetet, a következő paranccsal:

|  |
| --- |
| # Docker tesztelése/ellenőrzése:  *sudo docker run hello-world* |

A képen szöveg, elektronika, képernyőkép, multimédia látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

*4. ábra: Docker Hello-World image futása terminálban*

## **2.2.3. A program telepítése**

Miután kicsomagoltunk és beléptünk a MLSzakdoga mappába, ezután menjünk bele a deploy nevezetű mappába, ugyanis ebben találhatóak a felhasználói környezethez használatos fájlok.

Ezt a mappát nyissuk meg Terminal-ban.



*5. ábra: Mappa Terminálban való megnyitása Ubuntun*

Amint megnyitottuk akkor pedig a **./build.sh** parancs futattásával tudjuk telepíteni a programot. Elsőre nagyon sok minden fog kiíródni a képernyőre és elsőre ijesztő lehet olyan ember számára, aki még nem foglalkozott Dockerimage-l korábban. A programot csak hagyjuk telepíteni, nagyjából 10-20 percet vesz igénybe a program telepítése. Ha a képen látható kódsor megegyezik azon, amit a saját képernyőnkön látunk, akkor sikeres volt a telepítés. Az első telepítés után már nem is ajánlott többször telepíteni (ha nem módosítunk a programon valamit), de ha mégis ezt tenné a felhasználó, akkor pillanatok alatt fel fog újra telepedni.



*6. ábra: Sikeres ./build.sh futtatás*

# **2.3. Indítás**



*7. ábra: Elindított program teljes képernyőn*

A program elindításához lépjünk be a deploy mappába a MLSzakdogában belül (Ha a telepítés után olvassuk ezt a részt, akkor már valószínűleg benne is vagyunk a mappában!). A program a következő paranccsal indítható:

|  |
| --- |
| # Program elindítása:  *./run.sh paraméterek* |

Miután elindítottuk a programot a megfelelő paraméterekkel, a felhasználónak várnia kell egy 15 másodpercet mire elindul a Gazebo program és utána még egy extra 10 másodpercet mire a funkció is elindul utána, és már láthatjuk is a szoftveren belül, hogy működik.

## **2.3.1. A program funkciójának beállítása**

Amikor paraméterek nélkül indítjuk el a programot, akkor nem működni a program és egy „*Rossz futtatás! Megfelelő futattás: ./run.sh Learn / ./run.sh Continue / ./run.sh SaveData*” hibát fog dobni a program. Mint ahogyan a hiba is írja, meg kell adni paraméterként milyen funkciót akarunk használni, itt leírom melyik funkció mit takar, de ezeket egy későbbi fejezetekben fogom jobban kifejteni:

Learn – Új modell tanítása

Continue – Tanítás folytatása

SaveData – Tanítási adatok kiírása

Ezen kívül a felhasználó mégegy paramétert megadhat: A WITHLOG paramétert a funkció után, ennek az lesz az eredménye, hogy a docker úgy fog elindulni, hogy kiír nekünk információkat, például látni fogjuk tanítás közben, hogy hol tart a program.

Például, ha a felhasználó szeretné elindítani a Learn funkciót logolással, akkor azt így teheti meg:

|  |
| --- |
| # Learn elindítása logolással:  *./run.sh Learn WITHLOG* |

## **2.3.2. Program leállítása**

A program mivel csak egy docker konténer, amely fut, ezért a leállítása nagyon egyszerű, a következő paranccsal lehet ezt megtenni:

|  |
| --- |
| # Program leállítása:  *./stop.sh* |

Arra kell figyelni a leállításnál, hogy pillanatokon belül ki fog lépni a szoftverből, ezért mindenképpen csak azután kapcsoljuk ki a programot, miután végeztünk a teendőnkkel!



*8. ábra: Sikeres leállítása a programnak*

## **2.3.3. Program törlése**

A program törlése olyan egyszerű, mint a leállítása:

|  |
| --- |
| # Program törlése:  *./delete.sh* |

A program ezután a telepítés funkcióval újra letölthető, viszont megint megugrik majd a várakozási idő, mivel újra kell telepítenie minden csomagot a Dockernek.



*9. ábra: Sikeres törlése a programnak*

# **2.4. A program funkciói**

A következő fejezetben szeretném részletesen bemutatni, hogy a felhasználó milyen funkciókat fog tudni használni a programban és azon kívül.

## **2.4.1. Új modell tanítása**

Ez a legfontosabb funkciója az egész programnak. A programnak az lesz a lényege, hogy a Gazeboban létrejött robotnak megmondjuk, hogy hova kell mozognia, és utána jutalmat adunk neki, ahogyan teljesített, és ezt addig csináljuk, amíg beállítottuk.

Amikor elindítjuk ezt a funkciót, akkor először a program a resources mappában létre fogja hozni a megfelelő mappákat, amelyekben majd a megfelelő adatokat fogja tárolni. Ezután a létrehozza a robotnak a logikáját és a feladatát, amit át fog neki adni. Ekkor kezd el mozogni a Gazeboban belül a robot, ezt a felhasználó valós időben nézheti, nyomon követheti, hogyan tart a robot.

Ha az alapértelmezett beállításokat fogjuk használni, akkor 1000 lépés után mentenie kell egyet a programnak. Ilyenkor a resources/models mappában létre fog jönni, egy 1000.zip nevű modell, ez minden 1000.-ik lépés után növekedni fog, amíg el nem érjük a végső lépésszámot. A resources/log mappában pedig a tanítással kapcsolatos nyers adatok fognak lementődni, amelyet a Tanítási adatok kiírása funkcióban fogunk majd tudni felhasználni.

A funkcióhoz megadott konfig fájlban lévő beállítások már garantálják a megfelelő működést, de egy későbbi fejezetben, megmutatom melyik beállítás mit jelent, ha a felhasználó szeretné ezeket módosítani.

Amikor a program elérte a megfelelő lépésszámot, akkor kilép a program.

A program végzése közben a felhasználónak van lehetősége felvenni a képernyőjét, ezt az Ubuntu rendszeren a vezérlőpultban találjuk meg. Utána pedig kiválasztjuk a képernyőfelvétel funkciót, és kijelöljük mekkora képernyőt lehessen felvenni, és utána a *Screencasts* mappában fogjuk tudni megtalálni a felvett videónkat.



*10. ábra: Robotunk tanulás közben*

**

*11. ábra: Egy 10000 lépéses tanítás közben létrejött adatok*

## **2.4.2. Tanítás folytatása**

Habár ez a funkció nagyon hasonlít az új modell tanítása funkcióra, teljesen más célt szolgál. Tudjuk, hogy egy ilyen tanítás órákig is eltarthat, és nem minden felhasználó tudja órákig ott hagyni a számítógépét, ezért a funkciónak az lenne a lényege, hogy egy lementett modellt, ott tudjunk folytatni, ahol „abbahagytuk”.

**FONTOS!!** Amikor ezt a funkciót először szeretnénk futtatni, akkor a *config* mappában lévő *CONTINUE\_DEFAULT\_CONFIG.yaml*-ben a **ModelPath** értéket meg kell adni, különben nem fog működni a program! Erről és a konfigfájlok felépítéséről egy későbbi fejezetben fogok írni. Ezt a funkciót csak akkor használjuk, ha már legalább egyszer használtuk az Új modell tanítása funkciót.

Ez a folytatás úgy történik, hogy a .zip végződésű lementett modellek, amelyek a tanítás közben jöttek létre, azokat be tudjuk olvasni, és a Stable Baselines segítségével, be tudjuk onnan újra olvasni az adatait, és csinál nekünk egy olyan szimulációt, amely ugyanúgy működött, mint ahol az előzőt abbahagytuk, viszont már okosabb lesz, ezáltal rögtön látható lesz a fejlődése.

A robot ezután megint elkezd mozogni a Gazebo alkalmazáson belül, és ha az alapértelmezett beállítást használjuk, akkor minden 1000 lépés után le fog menteni egy kész modellt a resources mappába, miközben a logjait, pedig a log mappába fogja létrehozni, ahonnan később fel tudjuk dolgozni az adatait.

A megfelelő lépések elvégzése után a program ki fog lépni.

A program végzése közben a felhasználónak van lehetősége felvenni a képernyőjét, ezt az Ubuntu rendszeren a vezérlőpultban találjuk meg. Utána pedig kiválasztjuk a képernyőfelvétel funkciót, és kijelöljük mekkora képernyőt lehessen felvenni, és utána a *Screencasts* mappában fogjuk tudni megtalálni a felvett videónkat.

A többi funkcióhoz hasonlóan, itt is a megfelelő konfig fájl módosításával, lehet beállítani a funkcióit, ezekről egy későbbi fejezetben fogok részletesebben írni.



*12. ábra: Egy 5000 lépéses folytatás közben létrejött adatok*

## **2.4.3. Tanítási adatok kiírása**

Ez a funkció nagyon különböző a többihez képest, ez lesz az egyetlen funkció, amely nem használja a Gazebo programot. Ez a funkció nem is fogja elindítani a szimulátort.

**FONTOS!!** Amikor ezt a funkciót először szeretnénk futtatni, akkor a *config* mappában lévő *SAVE\_DATA\_DEFAULT\_CONFIG.yaml*-ben a **LogFolder** értéket meg kell adni, különben nem fog működni a program! Erről és a konfigfájlok felépítéséről egy későbbi fejezetben fogok írni. Ezt a funkciót csak akkor használjuk, ha már legalább egyszer használtuk az Új modell tanítása funkciót.

Ez a program az alapból Tensorboard által kreált adatokkal foglalkozik. Amikor a felhasználó tanítja a programot, akkor minden egyes alkalommal amikor lement a modellt, akkor készülni fog egy log fájl is, amelynek a szerepe az, hogy minden egyes tulajdonságváltozást feljegyez, később ebből tud egy böngésző alapú megoldást adni nekünk, amelyben látjuk, hogy a robotunk hogyan fejlődött az idő múlásával.

Amikor elindítjuk ezt a funkciót, akkor a megadott mappában lévő nyers log fájlokat összegyűjti a program, és ki fogja őket válogatni lépésszám, érték, érték neve alapján. Ami után ezzel végzett a program, utána az érték nevek alapján újra formálja a létrejött adatok, hogy ezzel egy modern adatbázis kezelő alkalmazás is könnyedén meg tudja jeleníteni az adatot. Ezt az új létrehozott fájlt a processData mappában fogjuk találni a resources-on belül.



*13. ábra: Kimentett adatok kinézete egy .csv fájlban beolvasva egy arra alkalmas szoftverben*

Ezen felül még van egy másik megoldás is, amit tehet a felhasználó, az egy online Tensorboard felület, amely a nyers log fájlokból, különböző gráfokat létrehozni nekünk, tökéletes lehet azoknak, akik valamilyen kutatómunka miatt használják a programot.

A következő módon lehet elindítani ezt az online felületet:

|  |
| --- |
| # Tensorboard elindítása  *tensorboard --logdir=resources/logs/LOG\_HELYE* |

Ezután a képernyőn megjelenő porton megtekinthető a felület.



*14. ábra: Online Tensorboard felület kinézete*

# **2.5. Konfig és a program funkcióinak konfigfájljainak felépítése**

A legtöbb modern programnak szép felhasználó felülete van, amely segíti a felhasználónak a program használatát, viszont vannak olyan programok, amelyek a hordozhatóságra és a kicsi méretre pályáznak, azokban a programokban nem mindenhol biztos, hogy megtalálható lesz egy felhasználó felület. Helyette jobban preferálják a konfig alapú megoldásokat, amelyek könnyebb személyre szabhatóságot biztosítanak.

A program konfig fájljainknak .yaml végződése van, amely az egyik legnépszerűbb kiterjesztés az ilyen megvalósítás szempontjából. Úgy kell értelmezni az ilyen fájlokat, mint egy alkalmazásnál egy beállítások menüpont, csak itt egy fájlban vannak, amelyet maga a program fog később feldolgozni. Minél egyszerűbb láttam el őket, hogy könnyen lehessen értelmezni, mit is szeretnénk átírni. A következő alfejezetekben szeretném bemutatni az összes alfunkciónak a konfig fájljait, amelyet a programban tudunk használni, és bemutatni, hogy melyik soruk mit jelent.

## **2.5.1. Új modell tanítása konfigfájljának felépítése**



*15. ábra: Új modell tanítása konfigfájlja*

**FunctionName** – Ebben adjuk meg a funkciónak a nevét, ezt semmiképpen ne írjuk át! A learn az új modell tanítása funkciónak a neve

**FunctionProperties** – Az adott funkcióhoz tartó extra beállítások

**XGoalPosition** – X pozíció, amelyet a robotnak el kell érnie a tanítás során

**YGoalPosition** – Y pozíció, amelyet a robotnak el kell érnie a tanítás során

**LearningModel** – Tanítási modell, amellyel tanuljon a program. Jelenleg támogatott a modellek:

* *A2C*
* *PPO*
* *DQN*
* *SAC*
* *TD3*

Ennek a módosítását csak akkor ajánlom, ha ért hozzá a felhasználó!

**Length** – Tanítás hossza, mennyi ezer lépésig jusson el a robot

**SaveDataAfterFinished** – Amikor a tanítás végére értünk, akkor a program a tanitásí statisztikákat írja-e ki egy fájlba. Ha ez az érték igaz lesz, akkor a Tanítási adatok kiírása funkció fog lefutni

**SaveDataProperties** – Ha le akarjuk menteni az adatokat, akkor itt tudjuk beállítani hozzá a dolgokat

**CsvFilePath** – A .csv végződésű fájl útvonala, ahova ki tudjuk menteni az adatokat

## **2.5.2. Tanítás folytatása konfigfájljának felépítése**



*16. ábra: Tanítás folytatása funkció konfigfájlja*

**FunctionName** – Ebben adjuk meg a funkciónak a nevét, ezt semmiképpen ne írjuk át! A Continue a Tanítása folytatása funkciónak a neve.

**FunctionProperties** – Az adott funkcióhoz tartó extra beállítások

**ModelPath** – A modell útvonala, amelyet szeretnénk tovább tanítani

**XGoalPosition** – X pozíció, amelyet a robotnak el kell érnie a tanítás során

**YGoalPosition** – Y pozíció, amelyet a robotnak el kell érnie a tanítás során

**LearningModel** – Tanítási modell, amellyel tanuljon a program. Jelenleg támogatott a modellek:

* *A2C*
* *PPO*
* *DQN*
* *SAC*
* *TD3*

Ennek a módosítását csak akkor ajánlom, ha ért hozzá a felhasználó!

**Length** – Tanítás hossza, mennyi ezer lépésig jusson el a robot

**SaveDataAfterFinished** – Amikor a tanítás végére értünk, akkor a program a tanitásí statisztikákat írja-e ki egy fájlba. Ha ez az érték igaz lesz, akkor a Tanítási adatok kiírása funkció fog lefutni

**SaveDataProperties** – Ha le akarjuk menteni az adatokat, akkor itt tudjuk beállítani hozzá a dolgokat

**CsvFilePath** – A csv végződésű fájl útvonala, ahova ki tudjuk menteni az adatokat

## **2.5.3. Tanítási adatok kiírása konfigfájljának felépítése**



*17. ábra: Tanítási adatok kiírása konfigfájlja*

Ezt a funkciót csak magában hívjuk meg, amikor a másik programoknál kimentjük a dolgokat, akkor nem fogja ezt a config fájlt használni!

**FunctionName** – Ebben adjuk meg a funkciónak a nevét, ezt semmiképpen ne írjuk át! A SaveData a Tanítási adatok kiírása funkciónak a neve.

**FunctionProperties** – Az adott funkcióhoz tartó extra beállítások

**LogFolder** – A logmappának az útvonala, ahol a tanítás során a program létrehozta a log fájlokat

**CsvFilePath** – A csv végződésű fájl útvonala, ahova ki tudjuk menteni az adatokat

# **2.6. A létrehozott nyersanyagok és annak kezelése**

A program minden funkciója során valamilyen adat fog létrejönni, ebben a fejezetben szeretném ezeket bemutatni, és hogy miket tudunk velük kezelni.



*18. ábra: Resources mappa és annak felépítése*

A resources mappa, amelyben az összes adatot tároljuk, amely létrejön a program közben, azt 3 mappára tudjuk felosztani.

Az első a models mappa, amelyben láthatjuk, hogy napokra vannak felosztva a mappa nevek, ez arra szolgál, hogy könnyen meg tudjuk találni, hogy melyik nap mit csináltunk a programban. A dátum utáni szám arra szolgál, hogy a mai nap a hányadik mappát hozza létre a program, így könnyebben megtaláljuk a fájljainkat, akkor is, ha egy nap többször használjuk a programunkat. A mappákon belül találhatóak a lementett modellek .zip formátumban. Ezeket a modelleket tudjuk majd beolvasni a Tanítás folytatása funkcióban.

A második a logs mappa, amelyben ugyanolyan mapparendszer található, mint a models mappa esetében. Ezekben a mappákban, a tanítás közben létrejött értékváltozásokat tároljuk, amelyeket utána a Tanítási adatok kiírása funkció folyamán fogunk tudni felhasználni, azért, hogy utána az adatokat vagy meg tudjuk tekinteni a Tensorboard által létrehozott webhelyen, vagy saját gépünkre kimenteni, egy adatbázis fájlban, ahonnan bármit tehetünk majd vele.

A harmadik a proccessData mappa lesz, amelyben a Tanítási adatok kiírása funkció során, ha nem a weblapos megoldást válaszottuk, akkor csv fájlokat fogunk találni, amelyekben a kiírt adatok találhatóak. A .csv fájlokban az adatokat vesszők választják el egymástól, egy nagyon népszerű adatbázis formátum, amely a legtöbb adatbázis szoftverrel, mint mondjuk az excellel is tökéletesen működik.

# **2.7. A program logolása és annak értelmezése**

A logolás egy olyan kifejezés, hogy a programban található kódban el vannak rejtve olyan parancsok, ahol szeretnénk, hogy a programunk kiírjon valamilyen információt, lehet ez hiba vagy esetleg csak egy üzenet is. A legtöbbször ezt a konzolunkra tudjuk kiírni, viszont a legtöbb modern programban már egy külön fájlban írják ki ezeket a dolgokat, mivel egy komplexebb program esetén több 1000 sor log is létrejöhet, akár rövid időn belül is.

A mi programunk is egy fájlban fog logolni, a projekt törzsében található lesz az *app.log* nevű fájl, ebben találjuk a program alatt létrehozott összes logot. A felhasználónak itt csak a hibákat fogom megemlíteni, mivel a többit úgy érzem, hogy nem szükséges tudnia a felhasználónak, és ezeket csak a fejlesztői részen fogom jobban kifejteni.



*19. ábra: Kilogolt hiba kinézete*

A képen látható a logból kapott hibaüzetet. A kódban úgy van megoldva, hogy minél érthetőbben leírja, hogy a mi a program hibája, és hogyan lehetne megoldani a hibát, a legtöbb hiba valószínűleg a konfig részen fog megtörténni, a nem megfelelő átírása során. A log sor elején a dátumot láthatjuk, ezzel tudjuk igazolni, hogy igen mostanában történt a hiba, ezáltal ezt kell kijavítanom. A következő rész ERROR-t kell, hogy írjon, a logban a legtöbb sor INFO taget fog kapni, viszont ez a felhasználónak nem fontos adatok, hanem inkább a fejlesztőnek hasznos. Utána a log megadja, hogy a melyik mappában, fájlban, függvényben, sorban található a hiba, ez megint nem fontos infó a felhasználónak, de egy fejlesztőnek nagyon hasznos lehet valami hiba esetén. Az utolsó rész maga az üzenet, amit a program át akar adni, ez a legfontosabb a felhasználónak, mert itt fogja leírni, hogy mi a probléma, és mi lehet egy esetleges megoldás.

Habár a logolás nem egy olyan dolog, amely a legtöbb felhasználónak fontos lehet, viszont úgy érzem, hogy nagyon fontos lehet mindenkinek, aki szeretné jobban megérteni, mi történik a háttérben.

# **Fejlesztői dokumentáció**

A fejlesztői dokumentációban megismerhetjük a forráskód felépítését, a függőségeket, amelyek a program működéséhez elengedhetetlenek, illetve a programnak a belső működéseibe belemegy részletesebben a dolgozat, mint például a logolás vagy a konfigurálás. Ezen felül bemutatom a fejlesztői és tesztelési környezet beállítását és helyes használatát, a szoftver követelmény-specifikációját, és a dolgozat verziókezelését.

# **3.1. Forráskód beszerzése**

A projekt forráskódját mellékeltem a dolgozathoz, *VÖRÖSDÖME\_QK8IUC\_SZAKDOLGOZAT.zip* néven, de az általam használt verziókezelőből is letölthető az alábbi paranccsal:

1. Nyissuk meg az eszközünkön a terminált
2. Telepítsük fel a GIT csomagot

|  |
| --- |
| # Git csomag telepítése  *sudo apt update*  *sudo apt install git* |

1. Ezzel a paranccsal töltsük le a csomagot:

|  |
| --- |
| # Git projekt klónolása  *git clone https://github.com/RedDome/MLSzakdoga.git* |

A kapott mappánk a MLSzakdoga mappa lesz, amelyben minden megtalálható a program megfelelő működéséhez. A forráskód beszerzése után a következő fejezetben szeretném bemutatni a forráskód könyvtárszerkezetét.

# **3.2. A forrás könyvtárszerkezete**

Amint kicsomagoljuk a MLSzakdoga könyvtárat, akkor a mappában számos almappa mellett találni fogunk fájlokat is, amelyek a fejlesztői környezet megfelelő működéséhez elengedhetetlenek.

A képen szöveg, képernyőkép, Betűtípus, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

*20. ábra: MLSzakdoga fájlrendszere*

A MLSzakdoga gyökerében lévő fájlok leírása:

* .gitignore: GIT verziókezelőhöz használt fájl, ebben megadom azokat a fájlokat, amiket nem kell verziókövetni, mint például a létrejött adatok.
* Dockerfile: Docker kép létrejöttéért felelős, ebben töltjük be a megfelelő csomagokat és függőségeket, ezáltal egy könnyű és konzisztens környezetet tudunk biztosítani.
* entrypoint.sh: A fájl, amelyet a Dockerfile az kezdőpontjának fog beállítani, vagyis, hogy ez a fájl fusson le miután sikeresen létrehozta a Docker képet, ebben a fájlban megadunk különböző ROS-al és Gazebo-val kapcsolatos beállításokat, amelyek a megfelelő futáshoz kellenek.
* runprogram.py: Innen indul el a program, mind a fejlesztői, mind a felhasználói elindításnál.
* setup\_env.sh: A Gazebo és világa beállítását végzi el. A fájlban létrehozzuk a megfelelő könyvtárakat, amelyeket a környezet felállításához használni kell, utána a megfelelő fájlokat forrásoljuk. Ezeket a lépéseket ellenőrizzük is, ezáltal biztosítva, hogy megfelelően létrejöttek. Ezután létrehozzuk a turtlebot3 robotot, amelynek tulajdonságokat és logikát adunk. Ezután a világnak az adatait tudjuk megadni, milyen talajon működjön, milyen világítások és akadályok legyenek rajta. Az utolsó lépésként pedig a robotnak megadjuk a megfelelő szenzorokat és lézereket, amelyek biztosítják a mozgását és látását.
* start\_ros.sh: A roscore program elindításáért felelős, azután pedig innen indul el a Gazebo program.
* test\_env.sh: Azt fogjuk ebben a fájlban tesztelni, hogy a megfelelő szenzorok elérhetőek-e, létrejöttek-e a megfelelő módon.

Ezen kívül még megtalálhatóak mappák is a MLSzakdogán belül, ezeket is szeretném röviden bemutatni:

* .devcontainer: A Visual Studio Code Dev Container működéséhez szükséges mappa
* .vscode: A Visual Studio Code Dev Container működéséhez szükséges mappa
* catkin\_ws: A környezet felállítása közben létrejött fájlokat tartalmazza, amiket fejlesztői célokból kivezettem a forrásba, ezáltal könnyen módosíthatóak, és követhetőek.
* config: Itt találhatóak az alapértelmezett konfig fájlok, amelyekkel az alkalmazás különböző verziói elindíthatóak.
* deploy: A felhasználó alkalmazás ebben a mappában található, hasonlói a felépítése, mint a fejlesztői alkalmazásnak, csak könnyebben használható, így biztosítva a felhasználónak a környezetbarát megoldást.
* docs: Ebben találhatóak különböző képek, videók, dokumentumok, amelyek a programot mutatják be, segítik jobban megérteni a funkcióit.
* src: Ebben találhatóak a fájlok amelyek az alkalmazást működtetik.

# **3.3. Függőségek**

A program megfelelő működéséhez különböző külső könyvtárakat használ a program. Ezek a könyvtárak a következőek:

* **Stable Baselines 3**: Egy python könyvtár, amely megerősítéses tanulási algoritmusokat tartalmaz. Célja, hogy egyszerűsítse a tanulási algoritmusok használatát és fejlesztését. Több fajta algoritmus támogat, mint például a PPO (Proximal Policy Optimization), A2C (Advantage Actor Critic) vagy a DQN (Deep Q Learning).
* **ROS (Robot Operating System)**: Nyílt forráskódú keretrendszer, amelyet olyan alkalmazások fejlesztésére használnak, ahol robotokat alkalmazunk. Habár a nevéből arra gondolnánk, hogy ez egy operációs rendszer, ez valójában eszközök és könyvtárak gyűjteménye, amelyek a robotok programozását segítik.
* **Gazebo**: Fejlett robot szimulációs szoftver, feladata a robotok valós környezetekben való szimulálása. A Gazebo szoros integrációval működik a ROS-al, gyakran használják együtt őket a robotika világában.
* **Tensorflow**: A Google Brain Team által fejlesztett könyvtár, amely számos gépi tanulás és mesterséges intelligenciával kapcsolatos feladathoz használható. Az egyik legnépszerűbb mélytanulási keretrendszer, amely elérhető manapság.
* **Loguru**: A python alapvető logolásának újragondolása, a meglévő funkciók mellett még többet tesz bele a készítő és egyszerűbb használatával a projektben könnyebben lehet konzisztens logolást biztosítani.

## **3.3.1. A függőségek beszerzése**

Ezen függőségek alapvetően telepítve vannak a forráskód megfelelő futtatása során, és a tesztelésekkel pedig ellenőrizve van, hogy minden a megfelelő módon működik-e, ezért a felhasználónak a beszerzéssel kapcsolatban nincsen semmilyen teendője.

## **3.3.2. A függőségek frissítése**

A Gazebo és a ROS szoros integrációja miatt, nagyon nehéz a függőségek frissítése anélkül, hogy el ne rontsuk a másik függőségek funkcionalitását. Ezért habár van frissítési lehetőség, nem ajánlott a megfelelő működés érdekében.

# **3.4. A forrásmappa felépítése**

A következő alfejezetekben szeretném bemutatni, hogy a fejlesztői környezet milyen részekből áll, és ezekbe kicsit konkrétabban belemenni, hogy mi is történik ezekben a mappákban, és hogyan kötődnek hozzá a program működéséhez.

## **3.4.1. Fejlesztői konténer felépítése**



*21. ábra: Fejlesztői konténer működés közben*

A fejlesztői konténernek a célja az, hogy egy olyan fejlesztői konténer jöjjön létre, amelyben, ha bármi probléma történne, akkor csak teljesen újra lehessen indítani, nem fogja elrontani a fejlesztőnek a gépét semmilyen szinten, nem fog benne semmilyen kárt se hagyni, a konténer biztosít egy „sandbox” környezetet.

A fejlesztői konténer felépítését 2 részre lehet szétosztani:

1. .devcontainer mappa: Megtalálható benne a *devcontainer.json*, amelyben tudjuk beállítani a Docker fájlunk elérhetőségét. Azon kívül beállíthatóak a futási parancsok a Docker fájlhoz, a Python fájlok helyéi és a port, amelyen a Visual Studio kommunikálni fog.

A képen szöveg, képernyőkép, szoftver látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

*22. ábra: devcontainer.json felépítése*

1. .vscode mappa: Mivel a fejlesztői konténer működéséhez elengedhetetlen a Visual Studio, ezért ebben a mappában a Docker és a Visual Studio közötti kommunikáció történik. A *launch.json* fájl azért felel, hogy amikor elindítjuk a programot, akkor milyen fájl induljon el, és annak a környezeti beállításait, amíg a *tasks.json* file azokért a folyamatokért felel, amit a program indítása előtt el kell végeznie a környezetnek.



*23. ábra: launch.json felépítése*

**

*24. ábra: tasks.json felépítése*

## **3.4.2. Fejlesztői Docker konténer felépítése**

A fejlesztői Docker fájl, habár megegyezik a felhasználói Docker fájllal, azért van a projektben, hogy a fejlesztő úgy tudjon módosítani a programon, hogy egy biztosan működő programrésze maradhasson. Ez az elsődleges szerepe a 2 Docker fájlos felépítésnek.

A Docker első sorában először meg van adva milyen alapot használunk a konténerünkhöz, egy hivatalos ROS Noetic Docker képet használunk, amely egy Ubuntura épülő kép, amelyben benne van a Gazebo 11 is, ez a verzió, amit a program is használni fog. A következő részben beállítunk különböző dolgokat, mint például, hogy a gépünkön megjelenjen a Gazebo, vagy a turtlebot3-nak a használt modelljét. Utána telepíteni fogjuk a megfelelő csomagokat. A program legvégén beállítunk még extra beállításokat és utána megadjuk az entrypoint-ot, amelyet már korábbi fejezetben definiáltam, hogyan működik.



*25. ábra: Dockerfile felépítése*

## **3.4.3. Környezet felépítése**

A szakdolgozat egyik legnehezebb feladata a saját környezet létrehozása volt, és ennek felépítése. A környezet alapjai a Gazeboban használt alapkörnyezetnek a kiegészítése. A környezet meghívásánál megadjuk a kezdő, és célpoziciót, mielőtt feliratkozunk a különböző szenzorokra. Az init függvény végénél még beállítunk pár adatot, hogy a robot megfelelően működhessen. A szenzoroknak mind vannak funkciói, amikre fel kell iratkoznia a környezetnek, és ezeket is hívja meg a robot működés közben.



*26. ábra: A szenzorok függvényeinek definiálása*

Az alap Gazebo-s környezetnek 3 nagyon fontos függvénye van, amelyet mindenképpen saját igényeink szerint újra kell alakítani:

1. Step függvény: a robot mozgását tudjuk irányítani 3 különböző irányban. A függvény végén megnézzük, hogy a robot poziciója megegyezik-e a célpozícióval.

*27. ábra: A lépés függvénynek definiálása*

1. Reward függvény: Jutalmazás a szerepe, ahhoz képest, hogy a robot merre helyezkedik el a célponttól. Ha a célponttól messze helyezkedik el, falnak ütközne vagy túl közel lenne a falhoz, akkor is a robot negatív jutalmat kap.



*28. ábra: A jutalom függvénynek definiálása*

1. Reset függvény: Összesen annyi szerepe van, hogy a robot minden fontos adatát újraindítsa.



*29. ábra: Az alapállapot függvénynek definiálása*

## **3.4.4. Robot és a környezet kapcsolata, kommunikációja**

Az egyik legfontosabb feladat a robot és a környezet közötti megfelelő kommunikáció felállítása volt. Ezt sikerült is teljesíteni, az alábbi ábrán látható is melyik funkció hogyan kommunikál a robottal.



*30. ábra: Robot és a környezet közötti működési kapcsolat*

# **3.5. Követelmény-specifikáció**

Ebben a fejezetben szeretném bemutatni a dolgozat elején megírt követelmény-specifikációt, amelynek tervei alapján készült a dolgozat.

## **3.5.1. Követelményelemzés**

Az alapfeladat mindenképpen az lenne, hogy egy ilyen komplex mesterséges intelligencia programnak, mint a mobil robot megerősítéses tanítása, annak adjak egy olyat keretet, amely segítségével a legtöbb nem hozzáértő ember is tudja használni probléma nélkül. Korábbi kutatásaim alapján, úgy véltem, hogy Linuxon sokkal több segítség elérhető, mint a többi operációs rendszeren ezért mindenképpen abban a környezetben szeretném megoldani a feladatot. Mivel egy fontos része a feladatnak egy környezet létrehozása, amely sok emberhez eljuthat, ezért, szeretném a lehető legegyszerűbben kezelhetőre megcsinálni a programot. Mivel nagyon fontos a programnak a hordozhatóság, ezért nem szeretnék felhasználói felületet beletenni és helyette konfig alapú működést szeretnék létrehozni. A legfontosabb része a tanítás lesz, ahol Gazebo pályán szeretném megvalósítani a tanítást. A felhasználónak lehetősége lesz nézni élőben, ahogyan a program működik, és nézhető élőben, ahogyan tanul a robot. A következő funkció szerepe egy olyan funkció lenne, ahol a tanítást lehetne folytatni, így megadva a lehetőségét, hogy bármikor felkapható legyen a program. A harmadik funkciónak a lényege az lenne, hogy a tanítással szerzett adatokat feldolgozza, hogyan feldolgozható állapotba tegye őket. A végleges funkciója a szoftvernek az lenne, hogy a felhasználónak vegyen fel a szoftver egy videót, amin látszódik a robot mozgása, ezzel a megfelelő kutatáshoz, lesz videós anyagja is a kutatónak.

A felhasználónak legyen lehetősége a konfig alapú rendszerben a beállítások könnyű módosítására, ennek könnyű és gyors feldolgozására.

A programot a Linux környezeten kívül (22.04 Ubuntu), Python programozási nyelvben szeretném megvalósítani. Kutatásaim alapján arra jutottam, hogy a Linux + Python a legnépszerűbb környezet mesterséges intelligencia alkalmazások megvalósítására. A konfig alapú működést .yaml kiterjedésű fájlok fogják biztosítani. A mesterséges intelligencia résznél pedig a Turtlebot oldalán található útmutatót fogom használni kiinduló pontban, annak segítségével szeretném összekötni azt a projektet a sajátommal, esetlegesen módosítani azon, ha a program megköveteli. A korábbi egyetemes tanulmányaimat is át fogom nézni, mi szállítható át belőle ebbe az új projektbe.

Az alkalmazás célja, hogy gyors, megbízható és környezetbarát legyen. Célja, hogy minél kisebb mérete miatt könnyen szállítható legyen minden féle rendszerre, erőforrástól függetlenül.

## **3.5.2. Megvalósíthatósági terv**

- Humán erőforrás: 1 tervező/fejlesztő (250 emberóra)

- Hardver erőforrás: 1 fejlesztői, tesztelői számítógép (Ubuntu 22.04)

- Szoftver erőforrások: fejlesztőkörnyezet (Visual Studio), verziókövető (Github)

- Üzemeltetés: nem kell biztosítani

- Karbantartás: nem kell biztosítani

- Megvalósítás időtartama: Összesen 250 emberóra

## **3.5.3. Nem funkcionális követelmények**

* Hatékonyság: A program terhelés jelent a processzorra és memóriára, mentés esetén a háttértárat is terhelheti. A program igényel hálózati kapcsolatot. A program gyorsan működik, a tanítás és megjelenítés részei a programnak időigényesek lehetnek. A program hatékony működéséhez egy jól felszerelt rendszer ajánlott.
* Megbízhatóság: Szabványos használat esetén nem fordul elő hibajelenség, és nem jelenik meg hibaüzenet. Ha valamilyen adat megsérül arról a program logolni fog a megfelelő helyre, ahol a felhasználó megtudhatja mi a hiba és hogy mit tegyen.
* Biztonság: A program csak a telepítés idejére fog internet kapcsolatot igényelni, a program offline működése közben az adatok biztonságban vannak.
* Működési: Hosszabb használati idő, akár 5-6 óra is, gyakori használat várható
* Fejlesztési: Python nyelv, Linux rendszerben megvalósítva

## **3.5.4. Funkcionális követelmények**

Alapprogram:

* Tanítás lehetősége
* Tanítás folytatása
* Adatok megformázása
* Képernyő felvétele

Adatok megformázása:

* Nyers adatok kigyűjtése, feldolgozása
* Feldolgozható formátumba alakítása

Képernyő felvétele:

* Program működése közben létrejövő videó
* Kutatásokhoz tökéletes

# **3.6. Felhasználói esetek**

A program 3 részre bontható funkcionalitás szempontjából: Új modell tanítása, Tanítás folytatása és Tanítási adatok kiírása. A következő alfejezetekben szeretném ezeknek a funkcióknak bemutatni a felhasználói eseteit és azoknak diagrammjait.

## **3.6.1. Új modell tanítása felhasználói esetei**

Ennek a funkciónak a futása előtt van lehetőségünk beállítani a konfigban található beállításokat. Ezután ezt a program betölti és elindítja a tanítást. Időközönként a program menteni fog, és ha be van kapcsolva a mentés funkció, akkor a tanítás vége után feldolgozza a keletkezett adatokat. A legvégén pedig kilép a program.



*31. ábra: Új modell tanítása felhasználói esetei diagrammon ábrázolva*

## **3.6.2. Tanítás folytatása felhasználói esetei**

Ennek a funkciónak a futása előtt van lehetőségünk beállítani a konfigban található beállításokat. Ezután ezt a program betölti és mielőtt elindítja a tanítást, azelőtt betölti a folytatandó modellt. Időközönként a program menteni fog, és ha be van kapcsolva a mentés funkció, akkor a tanítás vége után feldolgozza a keletkezett adatokat. A legvégén pedig kilép a program.



*32. ábra: Tanítás folytatása felhasználói esetei diagrammon ábrázolva*

## **3.6.3. Tanítási adatok kiírása felhasználói esetei**

Ebben a funkcióban, miután beállítottuk a megfelelő konfig beállításokat, azután a program a megadott mappában lévő log fájlokat fel fogja dolgozni és ki fogja belőlük nyerni a fontos adatokat. Ezután a program a már létrejött adatokat átalakítja, hogy jobban értelmezhető legyen a felhasználó számára. Ezután leáll a program.



*33. ábra: Tanítási adatok kiírása felhasználói esetei diagrammon ábrázolva*

# **3.7. Rendszerarchitektúra**

A következő részben a rendszerben használt programnyelvet és főbb komponenseket szeretném bemutatni.

## **3.7.1. Programnyelv és fejlesztői környezetek**

A program Python programozási nyelvet választottam, ez a nyelv támogatta legjobban, azt a fajta dolgozatot, amit létre akartam hozni. A fejlesztői környezet a legfrissebb Visual Studio Code volt, amelyben Dev Containerben történt a fejlesztés. A Visual Studio Code-ban használtam a Remote Containers és Python csomagokat is a könnyebb használat érdekében.

A környezet felállításához Docker fájlokat és különböző Bourne shell scripteket használok.

## **3.7.2. Főbb komponensei a programnak**

A programnak több főbb komponense van, ezeket szeretném bemutatni a következő sorokban:

* **runprogram.py**: A Docker indítása utáni kiinduló pont, itt dől el milyen konfig fog betöltődni a programban.
* **sharedValues.py**: Ebben a fájlban találhatóak olyan értékek, amelyeket a program minden része használni fog
* **processConfigFile.py**: A konfig betöltéséért és feldolgozásáért felelős fájl
* **startFunction.py:** A megfelelő funkcióhoz tartozó metódus elindításáért felel
* **createDirectories.py:** A *resources* és tanulási funkciók során létrejövő mappákat hozza létre
* **trainGazebo.py:** Az új modell tanítása funkció implementációja
* **continueTrainingGazebo.py:** A tanítás folytatása funkció implementációja
* **saveDataFromTensorboardFiles.py:** Tanítási adatok kiírása funkciónak implementációja

# **3.8. A fejlesztői környezet**

Miután kicsomagoltuk a MLSzakdoga.zip-et, ezeket a lépéseket kell tennünk mielőtt el tudjuk indítani a fejlesztői környezetet:

1. Visual Studio Code telepítése: [8]
2. Megjelenítő képernyő beállítása a terminálban:

|  |
| --- |
| # X szerver konfigurálása  *xhost +local:docker* |

Ezzel a beállítással azt fogjuk elérni, hogy amikor a Dockeren belül elindul az alkalmazás, akkor a mi képernyőnkön megjelenjen meg a Gazebo. Amikor megnyitjuk először a kódot a Visual Studio Codeban, akkor a jobb alsó sarokban meg fog jelenni az opció, hogy beállítsuk a fejlesztői környezetet:



*34. ábra: Visual Studio PopUp amiben elindítható a Dev Container*

Miközben a fejlesztői konténer beállítása történik, közben a Docker kép le fog töltődni, ezáltal az első elindításkor sokat kell majd várnunk mire összeáll a program. Ezután végre készen állunk a program elindításához.

Az alkalmazást az F5 gomb megnyomásával tudjuk majd elindítani. Ezután összeáll a Gazebo környezet és elindul a világunkkal és a robotunkkal. Ezután a Visual Studio Code ki fog írni egy figyelmeztetést, mivel a Gazebo elindítja, egy folytonos feladatot, ezért az alkalmazásunk maga nem fog tudni elindulni magától.



*35. ábra: Visual Studio Code Warning üzenet*

A felhasználónak a Debug Anyway gombot kell megnyomnia és akkor fog elindulni a *runprogram.py* fájl, és pár másodperccel később már látni fogjuk a terminálban és a Gazeboban, hogy elindult a kívánt funkció.

Ha esetleg hibába ütközünk miközben elindítjuk a programot, bizonyosodjunk meg róla, hogy rendesen beállítottuk a megjelenítő képernyőt és utána építsük újra a konténert. Ezután már normálisan el kéne indulnia a programnak.

# **3.9. Tesztkörnyezet**

A tesztkörnyezet a fejlesztői környezetben lesz megtalálható. Egybe van importálva a src fájlban található kódokkal, amelyen belül a tests mappán belül találhatóak meg a tesztek. A tesztek nagy része, mivel a Gazebot fogja tesztelni, ezért a futásuk előtt el kell indítani a Gazebo-t.

# **3.10. A tesztelési terv**

Bármelyik szoftverfejlesztés egyik legfontosabb feltétele a megfelelő tesztelés. Egy megfelelően tesztelt program nagyon sok későbbi fejfájástól meg tudja kímélni a fejlesztőt. Az alábbi részen szeretném bemutatni a teszteléseket, amelyeket a programon elvégeztem.

## **3.10.1. Manuális tesztelés**

A manuális tesztelés minden elkészült funkció után készült. Minden elkészült funkcióról készült egy „user story” leírás, amelynek a feladata, hogy leírja a funkció milyen szerepet végez el a programon belül. A programfordítás után a program futása közben különböző check-ekkel tesztelem, hogy teljesül-e a „user story”-ban meghatározott szerepe a funkciónak. Ha ez nem teljesül, akkor a kódban hibajavítással javítom ki a hibát és ezt addig ismétlem, amíg az elvárt eredmény nem következik be. Minden funkció, ami a programban található az alapos manuális tesztelési folyamaton esett át, ezért bárki, aki szeretné a szoftvert fejlesztői szempontból megközelíteni a jövőben, annak már nem kell majd aggódnia a manuális tesztelés miatt, amíg az eredeti funkciólistát megtartja.

## **3.10.2. Automatikus tesztelés**

A projekt automatikus tesztelését a Python unittest tesztelő rendszer segítségével hoztam létre. Ez a rendszer alapvetően a TestCase osztályra épül, ez teszi lehetőve számunkra, hogy több tesztesetet tudjunk definiálni. A TestCase osztály metódusai is elérhetőek ebben a rendszerben, mint például az assertEqual, ahol két különböző értéket nézünk meg, hogy megegyeznek-e, vagy példaul az assertRaises, ahol pont a helytelen működést tudjuk azzal tesztelni, hogy megnézzük, hogy a teszt hibás megoldással tér-e vissza.

A mi projektünknél az automatikus tesztelés, 3 különböző dolgokat fog tesztelni, ezekről szeretnék röviden írni.

1, ROS környezet tesztelése:

Ezekben a tesztekben a saját általunk használt ROS környezetet teszteljük a Gazebo segítségével. A lényege az a teszteknek, hogy kisebb mozgásokra és értékváltozásokra reagál-e megfelelően a program.



*36. ábra: customGazeboEnvironmentTest.py felépítése*

2, Konfig beolvasás tesztelése:

Az ilyen fajta tesztekben a program konfigbeolvasási funkcióit tesztelem, hogy ha a megfelelő értékek hiányoznak, akkor kilép a program ahogyan annak kell, vagy ha esetleg olyan érték hiányzik, amelyik pótolható, akkor a megfelelő helyettesítést használja. A végén pedig van egy pár tesztelés, ahol tökéletesen beolvas minden értéket, minden megfelelően működik.



*37. ábra: gazeboProgramTest.py felépítése*

3, Program tesztelése:

Az utolsó tesztelési csoportban, a Gazeboban belül teszteljünk a különböző funkciókat.



*38. ábra: processConfigFileTest.py felépítése*

A unittest *setUp* metódusa azt teszi lehetővé, hogy a tesztfájl kezdete előtt egy teljesen új környezet jöjjön létre, ezzel biztosítva, hogy az előző tesztfutás nem rontotta el a környezetet a következő tesztfutás számára. Azért is fontos mert ha esetleg valamikor az egyik tesztfutás helytelen lenne, akkor ne történjen meg az, hogy a következő tesztfutás mint dominók, úgy bukjanak utána, hanem minden tesztfutás egy külön kis környezetben, ezzel biztosítva van, hogy nem tudják befolyásolni egymást. A környezet lebontásáért a *tearDown* metódus felel, ezáltal nem marad lezáratlanul az objektum. A tesztek közben naplózás segítségével leírom, merre tart a tesztfájl, ezáltal a futása során tudjuk követni éppen melyik tesztnél tart. Ez azért kell, mert mint mondjuk a „Google Test”-nél nincsen vizuális kiírás a legvégén kívül, ezért ennyiben akartam segíteni a fejlesztőnek.

A környezetet úgy tudjuk elindítani, hogy nyitunk a Visual Studio Code környezetünkben egy új terminált, és utána belemegyünk a megfelelő könyvtárba:

|  |
| --- |
| # Automatikus tesztelés elindítása  *cd src/tests*  # Attól függ a felhasználó melyiket szeretné futtatni  *python3 customGazeboEnvinronmentTest.py/gazeboProgramTest.py/processConfigFileTest.py* |

A terminál figyelésével látszódik melyik teszt indult el, és minden esetleges hibát kiír a rendszer, ezután elmondja, hogy mennyi tesztet futatott le a program és utána kilép.

## **3.10.3. Tesztelési eredmények**



*39. ábra: Helyes tesztfutás*



*40. ábra: Helytelen tesztfutás*

A manuális tesztelés során megbizonyosodtam róla, hogy minden egyes funkció az elvárt módon fog szerepelni, akár a saját gépemen, akkor egy teljesen új számítógépen, és ugyanazt az eredményt érem el, ha felhasználói módban indítom el a programot vagy ha fejlesztői módban végzem el ugyanazt a feladatot.

Az automatikus tesztelés is tökéletes működik a saját rendszeremen és bármelyik rendszeren, ahol próbáltam. A tesztek gyorsan és precízen lefutnak és mindenhol az elvárt eredményt adják vissza. Azok a mappák és fájlok, amiket létrehoz működés közben a tesztelés törlésre kerül, ezzel is megspórolva tárhelyet.

Ha esetlegesen megbukik valamelyik teszt, akkor látható lesz, hogy melyik érték nem lesz megfelelő, és hogy mit kell változtatni, hogy újra megfelelő legyen a teszt, viszont ha a fejlesztő nem módosít a teszteken vagy a program belső működésén, akkor ez nem fog megtörténni.

## **3.10.4. Tesztek részletes leírása**

A három csoport alapján szeretném bemutatni milyen tesztek találhatóak meg az automatikus tesztelésben:

1, ROS környezet:

* test\_CheckEnv: A környezet ellenőrzéséért felelős teszt
* test\_CheckEnvInitValues: A környezet alapértelmezett értékeinek vizsgálata
* test\_ChangeGoalPosition: Ha a környezetben módosítjuk a végcélt, akkor a program ezt érzékeli
* test\_ResetEnv: Környezet újraindításának tesztelése, értékek visszaállnak-e, az alapértelmezett állapotukra
* test\_Reward: A reward funkció lefut-e helyesen
* test\_Odom: Odom megfelelően működik-e

2, Konfig beolvasás:

* test\_emptyYamlConfigTest: Üres Yaml fájl esetén hibát kap-e a program
* test\_emptyFunctionNameConfig: Üres FunctionName esetén hibát kap-e a program
* test\_emptyLogFolderConfig: Üres LogFolder esetén hibát kap-e a program
* test\_emptyModelPathConfig: Üres ModelPath esetén hibát kap-e a program
* test\_usingDefaultValuesConfig: Ha üresek az értékek, amelyek helyettesíthetőek, akkor megfelelően pótolja őket
* test\_usingCsvFilePathDefaultValueConfig: Ha üres a csvFilePath, akkor helyesen helyettesíti a program
* test\_goodLearnWithSaveConfig: Helyes Learn konfig beolvasása, a Save funkcióval
* test\_goodLearnWithoutSaveConfig: Helyes Learn konfig beolvasása, a Save funkció nélkül
* test\_goodContinueWithSaveConfig: Helyes Continue konfig beolvasása, a Save funkcióval
* test\_goodContinueWithoutSaveConfig: Helyes Continue konfig beolvasása, a Save funkció nélkül
* test\_goodSaveDataConfig: Helyes SaveData konfig beolvasása

3, Program:

* test\_learnFunctionWithSave: Learn funkció megfelelően működik, és a végén lementi az adatokat a program
* test\_saveDataFunction: SaveData funkció megfelelően működik

# **3.11. Verziókezelés**

A projekt verziókezelése a Github webes, GIT verziókezelőjén valósult meg. Ennek az volt a nagy előnye, hogy nem csak lokálisan, hanem bárhonnan el tudtam érni a projektemet. Nem beszélve a biztonsági mentésról, amit biztosít ez a weblap. Nagy előnye volt még az oldalnak, hogy a konzulensnek sokkal könnyebben el tudtam küldeni a munkám állapotját. A Visual Studio Code alapvetően támogatja a verziókezelőket, főleg a GIT alapúakat. A szoftveren belül lehet megnézni minden kódot, amit feltöltöttél a felületre, miközben ad arra lehetőséget, hogy megnézd milyen módon írtad át a kódodat ahhoz képest, ahogy a verziókezelőben van feltüntetve. Lehetséges a Visual Studio Code-on belül commitolni, pusholni és minden parancsot megcsinálni, amelyet terminálból csinálnál alapvetően, megkönnyítve ezzel a fejlesztőknek a feladatát.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

*41. ábra: A projekt „Github” oldala*

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

*42. ábra: A Visual Studio Code verziókezelési felülete*

# **3.12. A program logolása**

Minden program esetén elengedhetetlen a megfelelő logolás, amelyet teljesül itt is. A resources mappában található log mappában lesz található *app.log* nevű fájl, ebben lesz található az összes log sor, amelyet a program létrehozott nekünk. A Python által biztosított logolási könyvtárban voltak olyan problémák, amelyek miatt amellett döntöttem, hogy egy másik logolási könyvtárat fogok használni. A loguru [9] nevű nagyon népszerű logolási könyvtárat használtam a program során. A loggert nagyon könnyű beállítani, csak 2 sor kóddal már be is lehet állítani. Utána pedig minden egyes fájlban, ahol használni akarjuk, ott csak meg be kell importálni az osztályt, és a main metódusban definiált tulajdonságokat meg fogja jegyezni, így az egész programon keresztül azonos beállításokkal fog futni a logolás.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

*43. ábra: Loguru beállítása ennyire egyszerű a programunkban*

# **3.13. A program konfigolása**

A programban úgy éreztem, hogy a konfig beolvasás biztosításával könnyebben hordozható lesz a program, és könnyebb lesz a használata, személyre szabása. A program indulásakor meg tudjuk adni milyen konfig fájlt szeretnénk használni, alapvetően biztosítok a felhasználónak 4 alapbeállítást, aminek az előnye az, hogy aki nem ért hozzá, annak semmit se kell rajta módosítania rajta, hanem csak elindítja a megfelelő funkció konfiggal és már működik is a program. Ezek adatok feldolgozása a *processConfigFile.py* fájlban fognak megtörténni.



*44. ábra: processConfigFile.py kinézete*

A fájl elején megadok alapértékeket, így a hibás indítás esetén is, legalább valami el fog tudni indulni, minden hibáról logolni fog a program és megtekinthető lesz a log fájlban. Ezután szétszedem, hogy funkciókhoz képest, mely értékeket állítsa be a program. Miután sikeresen beolvassa a fájlokat, azután a program a megfelelő funkciót el fogja indítani.

# **Összefoglalás**

A program célja egy olyan program létrehozása volt, amely a felhasználónak lehetőséget ad arra, hogy könnyen tudjon tanítani robotokat a Gazebo program segítségével. Véleményem szerint az tökéletesen sikerült, és mindenfajta korosztálynak és tudással rendelkező embernek is létrehoztam egy könnyen használható programot. A fejlesztőknek is létrehoztam egy olyan alapot, amelyet nagyon könnyen lehet tovább fejleszteni és a megfelelő fejlesztői környezet létrehozásával, egy könnyű környezetet kap képhez a fejlesztő, ahol a megfelelő tesztelésnek és dokumentálásnak köszönhetően, könnyen megtudja érteni a program működését erről az oldalról is.

# **4.1. További fejlesztési lehetőségek**

Jelenleg a komplikált környezet miatt, a program csak Linux rendszerben működik, ezt mindenképpen a jövőben megcsinálnám platformfüggetlenre, ezzel minél több embernek eljuttatva a programot.

Ezen kívül a programon belül is voltak gondolataim, amiket nem tudtam kidolgozni, például lehetőség arra, hogy a felhasználó a saját robotját/világját használja a programon belül, esetlegesen egy világkreáló, amiben a felhasználó tud magának csinálni világokat, ahhoz képest, hogy milyen tanulást szeretne elvárni a robottól.

A programnak a későbbiekben lehetne adni egy GPU támogatott változatot is, mivel ez nagyon felemelné a teljesítményt, és így a felhasználó kevesebb idő alatt többet tudna elérni a tanulásban.

Egy képernyőfelvevő funkció is lehetne egy későbbi fejlesztés része, amely szorosan együttműködik a programmal, és így könnyen használható lenne a felhasználónak.

# **Irodalomjegyzék**

1. „Applied machine learning in cancer research: A systematic review for patient diagnosis, classification and prognosis” [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC8523813/> [Hozzáférés dátuma: 2024. 11. 21.]
2. „8 ways Amazon is using generative AI to make life easier” [Online]. Available: <https://www.aboutamazon.com/news/innovation-at-amazon/how-amazon-uses-generative-ai> [Hozzáférés dátuma: 2024. 11. 21.]
3. „Email that keeps your private information safe.” [Online]. Available: <https://safety.google/gmail/> [Hozzáférés dátuma: 2024. 11. 21.]
4. „Stable Baselines3 – Train on Atari Games” [Online]. Availaible: <https://colab.research.google.com/github/Stable-Baselines-Team/rl-colab-notebooks/blob/sb3/atari_games.ipynb> [Hozzáférés dátuma: 2024. 11. 22.]
5. „System Requirements for Gazebo: A Comprehensive Guide” [Online]. Availaible: <https://magnetica42.rssing.com/chan-80216447/article16.html> [Hozzáférés dátuma: 2025. 03. 04.]
6. „WinRAR download free and support: WinRAR” [Online]. Availaible: <https://www.win-rar.com> [Hozzáférés dátuma: 2025. 03. 28.]
7. „7-Zip” [Online]. Availaible: <https://www.7-zip.org/> s[Hozzáférés dátuma: 2025. 03. 28.]
8. „Visual Studio Code” [Online]. Availaible: <https://code.visualstudio.com/docs/setup/linux> [Hozzáférés dátuma: 2025. 04. 16.]
9. „Loguru – Python logging made (stupidly) simple” [Online]. Availaible:

<https://github.com/Delgan/loguru> [Hozzáférés dátuma: 2025. 03. 04.]