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](#_Toc194435985)

[**2. Felhasználói dokumentáció** 8](#_Toc194435986)

[**2.1. Rendszerkövetelmények** 8](#_Toc194435987)

[**2.1.1. Hardware követelmények** 8](#_Toc194435988)

[**2.1.2. Program beszerzése** 8](#_Toc194435989)

[**2.2. Telepítés** 9](#_Toc194435990)

[**2.2.1. Docker telepítése** 10](#_Toc194435991)

[**2.3. Indítás** 11](#_Toc194435992)

[**2.4. A program funkciói** 12](#_Toc194435993)

[**2.4.1. Új modell tanítása** 12](#_Toc194435994)

[**2.4.2. Modell szimulálása/képernyő felvétele** 13](#_Toc194435995)

[**2.4.3. Tanítás folytatása** 14](#_Toc194435996)

[**2.4.4. Tanítási adatok kiírása** 14](#_Toc194435997)

[**2.5. Konfig és a program funkcióinak konfigfájljainak felépítése** 16](#_Toc194435998)

[**2.5.1. Új modell tanítása konfigfájljának felépítése** 17](#_Toc194435999)

[**2.5.2. Modell szimulálása/képernyő felvétele konfigfájljának felépítése** 18](#_Toc194436000)

[**2.5.3. Tanítás folytatása konfigfájljának felépítése** 18](#_Toc194436001)

[**2.5.4. Tanítási adatok kiírása konfigfájljának felépítése** 19](#_Toc194436002)

[**2.6. A létrehozott nyersanyagok és annak kezelése** 19](#_Toc194436003)

[**2.7. A program logolása és annak értelmezése** 21](#_Toc194436004)

[**3.** **Fejlesztői dokumentáció** 23](#_Toc194436005)

[**3.1. Forráskód beszerzése** 23](#_Toc194436006)

[**3.2. A forrás könyvtárszerkezete** 23](#_Toc194436007)

[**3.3. Függőségek** 24](#_Toc194436008)

[**3.3.1. A függőségek beszerzése** 25](#_Toc194436009)

[**3.3.2. A függőségek frissítése** 25](#_Toc194436010)

[**3.4. A forrásmappa felépítése** 25](#_Toc194436011)

[**3.4.1. Fejlesztői konténer felépítése** 25](#_Toc194436012)

[**3.4.2. Fejlesztői docker konténer felépítése** 26](#_Toc194436013)

[**3.4.3. Környezet felépítése** 27](#_Toc194436014)

[**3.4.4. Robot és a környezet kapcsolata, kommunikációja** 30](#_Toc194436015)

[**3.4.5. Konfig felépítése** 30](#_Toc194436016)

[**3.5. Követelmény-specifikáció** 31](#_Toc194436017)

[**3.5.1. Követelményelemzés** 31](#_Toc194436018)

[**3.5.2. Megvalósíthatósági terv** 32](#_Toc194436019)

[**3.5.3. Nem funkcionális követelmények** 32](#_Toc194436020)

[**3.5.4. Funkcionális követelmények** 33](#_Toc194436021)

[**3.6. Use Case-k** 33](#_Toc194436022)

[**3.7. Rendszerterv** 33](#_Toc194436023)

[**3.8. A fejlesztői környezet** 33](#_Toc194436024)

[**3.9. Tesztkörnyezet** 34](#_Toc194436025)

[**3.10. A tesztelési terv** 34](#_Toc194436026)

[**3.10.1. Manuális tesztelés** 34](#_Toc194436027)

[**3.10.2. Automatikus tesztelés** 35](#_Toc194436028)

[**3.10.3. Tesztelési eredmények** 37](#_Toc194436029)

[**3.10.4. Tesztek részletes leírása** 38](#_Toc194436030)

[**3.11. Verziókezelés** 38](#_Toc194436031)

[**3.12. A program logolása** 39](#_Toc194436032)

[**3.13. A program konfigolása** 40](#_Toc194436033)

[**3.14. Tanítási módszerek és annak különbségei** 40](#_Toc194436034)

[**3.15. A resources mappában létrejött adatok részletes leírása** 40](#_Toc194436035)

[**4.** **Összefoglalás** 41](#_Toc194436036)

[**4.1. További fejlesztési lehetőségek** 41](#_Toc194436037)

[**5.** **Irodalomjegyzék** 42](#_Toc194436038)

# **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. 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* |

Miután elindítottuk a programot, 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 és világának beállítása**

Amikor paraméterek nélkül indítjuk el a programot, akkor alapértelmezett beállításokat fog használni a szoftver.

## **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**

Ebben a funkcióban a felhasználó elindít egy új tanítást, a megadott robottal, a megadott világban. A funkció elindítása előtt lehetőség lesz arra, hogy eldöntsük, mennyi ideig legyen menjen a tanítás. TODO mennyi idő ennyi vagy annyi tanítás? Amikor elindul a tanítás, át tudunk lépni a Gazebo ablakra, amelyen láthatjuk valós időben a tanítást.



TODO. ábra: Robotunk tanulás közben

A program a tanítás közben jelezni fogja a felhasználó számára, hogy hol tart a tanítás, hol mentett modell-t, és még mennyi ideig tart a program. A tanítás befejezése után, a program ki fog lépni. Hogy miket lehet kezdeni a megtanított modellekkel és logokkal, azt a 2.6-os fejezetben tudja elolvasni.

## **2.4.2. Modell szimulálása/képernyő felvétele**

Ez a funkció arra szolgál, hogy amikor a felhasználó szeretne egy korábban lementett modellet, megtekinteni vagy esetleg felvételt készíteni róla, akkor ezt a funkciót ajánlom, mert ezzel szépen lehet demonstrálni, hogy a modellünk mennyit fejlődött és hogy milyen okos lett. A funkció elindítása előtt, a megfelelő konfig fájlban meg kell adni a szimulálásra szánt modell helyét, és hogy mennyi ideig szeretnénk hogy fusson a szimuláció. A konfig második részében meg lehet adni a felvétellel kapcsolatos beállítások, mint például a minősége, hossza, formátuma amiben kimentse a kód, és hasonló beállítások. Ezeket tényleg csak azoknak ajánlom, hogy átírják, akik értenek hozzá.

Miután a felhasználó beleírt a konfigba a megfelelő adatokat és elindította a programot, azután el fog indulni egy a Gazebo szoftver, és ebben el fog kezdeni mozogni a robot, onnan folytatva, ahol abbahagyta. Innentől a felhasználónak nincsen semmi más teendője, csak annyi hogy hátraül és nézi a robotját, ahogyan mozog. Amikor elérjük a megfelelő határidőt, akkor a program le fogja menteni a videófájlt, amelyet a resources/video mappában meg is fog tudni találni, és onnan megtekinthető lesz.

// TODO kép felvett videóról

## **2.4.3. Tanítás folytatása**

Habár ez a funkció nagyon hasonlít az előző 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”.

Az előző funckiókhoz hasonlóan, a funkció konfig fájljában tudjuk beállítani a megfelelő dolgokat, erről egy korábbi fejezetben írtam. A konfig kitöltése és a program elindítása után, a modell betöltésre kerül egy új Gazebo szimulátorban, és utána határozatlan vagy előre meghatározott ideig folytatja a modell a tanulást. Ha határozatlan ideig tanít a modell, akkor a usernek lesz lehetősége leállítani a modell-t, amíg ha határozott ideig tanul, akkor a beállított idő végén kilép a program. A tanításhoz megfelelően, itt is készülni fog lementett modellek és logok amiket a resources mappában fog tudni a felhasználó megtalálni.

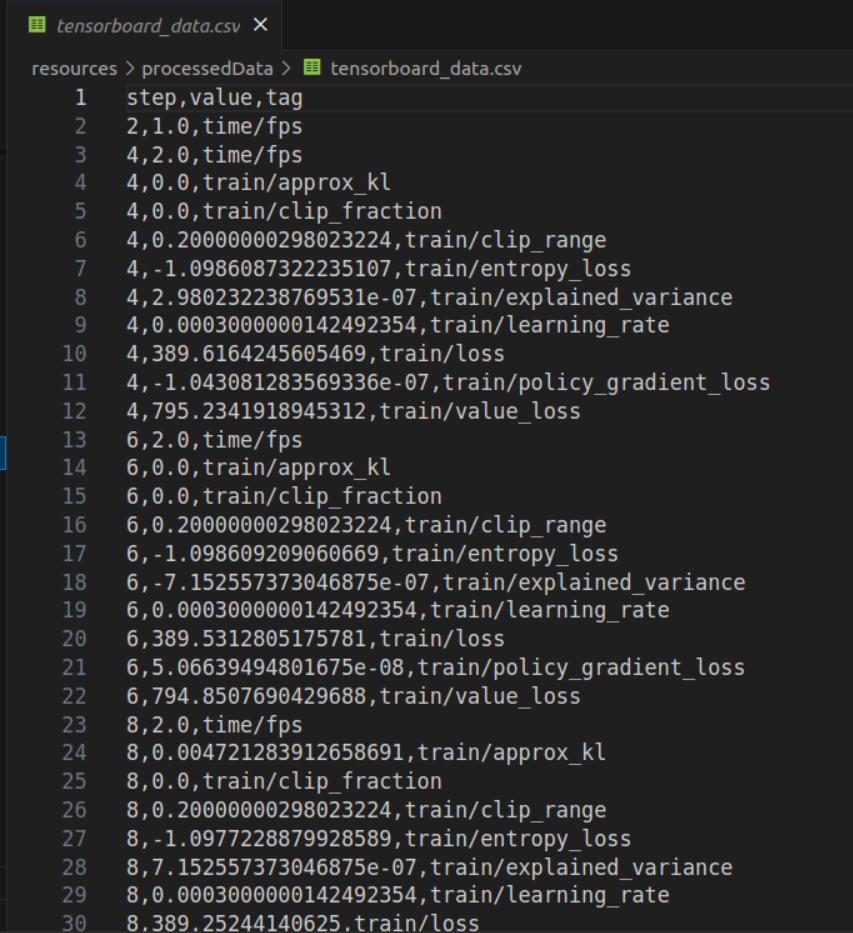
## **2.4.4. Tanítási adatok kiírása**

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



TODO. ábra: Online Tensorboard felület kinézete

Erre is lehetőséget ad a funkció, viszont inkább azoknak ajánlom a program használtat, aki az sok adattal saját maga szeretne csinálni valamit a felhasználó. A konfigban megadott log mappából, csinálni fog egy csv (Comma-Separated values) fájlt, amelyből utána ki lehet szedni a számunkra megfelelő adatot.



TODO. ábra: Kimentett adatok kinézete egy .csv fájlban

# **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**



TODO. 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 automatikusan böngészőben megjelenítse a tanitásí statisztikákat vagy írjak-e ki őket 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**



TODO. 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 automatikusan böngészőben megjelenítse a tanitásí statisztikákat vagy írjak-e ki őket 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**



TODO. 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. A fejlesztői dokumentálás részben, el fogom magyarázni részletesebben, hogy a létrejött fájlok mit tudnak és mire valóak, ha a felhasználó kíváncsi erre, akkor javaslom, hogy az ehhez tartózó //TODO fejezetet is olvassa el.



TODO. ábra: Resources mappa és annak felépítése

A resources mappa, amelyben az összes adatot tároljuk, amely létrejön a program közben, azt 5 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. Ennek a fájlnak a felépítését a fejlesztői dokumentációban fogom elmagyarázni részletesen //TODO, azt a fejezetet ajánlom elolvasni.

A negyedik a video mappa lesz, amelyben a Modell szimulálása/Képernyő felvétele funkció során létrejött videófájlokat találjuk meg. A konfig fájlban megadható milyen legyen a videóformátumja, de a legtöbb formátum könnyen felismerhető a számítógépünk által, de például a VLC nevű program [8] tökéletesen le tudja majdnem az összes videóformátumot játszani, probléma nélkül.

Az utolsó a logs mappa lesz, amelyben a program során létrejött logolás lesz megtalálható, erről a következő fejezetben fogok részletesen írni.

# **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 resources mappában található log mappában található lesz az appLog nevű fájl, ebben találjuk a program alatt minden minden 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.

// logról kép, hibáról leginkább

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 valószínüleg 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 kéne hogy írjon, a logban a legtöbb sor INFO tag-et 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 könytárat, akkor a mappában számos almappa mellett találni fogunk sima fájlokat is, amelyek a fejlesztői környezet megfelelő működéséhez elengedhetetlenek. A Dockerfile a docker image létrejöttéért felelős, ebben töltjük be a megfelelő csomagokat és függőségeket, ezáltal egy könnyű és konzisztens környezetet tudunk biztosítani. Az entrypoint.sh file lesz a fájl, amelyet a Dockerfile az entrypointjának fog beállítani, vagyis, hogy ez a fájl fusson le miután sikeresen létrehozta a docker image-t, ebben a fájlban beállítunk különböző ROS-al és gazebo-val kapcsolatos beállításokat, amelyek a megfelelő futáshoz kellenek. A setup\_env.sh file 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. A start\_ros.sh fájlban a roscore program indul el, amivel a node-ok indulnak el, amelyek összekötik a ros-t a környezettel és a robottal. A test\_env.sh pedig a környezet teszteléséért felel, amivel biztosítjuk a tökéletes működést. TODO start\_ros, test\_env-ről többet írni

A fejlesztői környezethez tartozik 2 mappa is, amely a fejlesztői konténerben lévő fejlesztést biztosítja. A fejlesztői konténer garantálja, hogy a program minden egyes alkalommal ugyanúgy működik, akármilyen gépet használunk. Ez a komplikált függőségek és könyvtárak miatt szükséges. A .devcontainer mappában lévő devcontainer.json-ban különböző beállításokat teszünk, mint például a főmappa helye, vagy a kódnak a gyökere. A .vscode mappában 2 fájl található, az első fájl amelynek a neve launch.json, amelyben megadjuk melyik fájl induljon el először a programban, miután összeáll a környezet, amíg ebben a fájlban beállítunk egy pár környezeti változót. A tasks.json fájlban a különböző feladatokat definiáljuk, amiket el kell végeznie a visual studio code-nak mielőtt elindulhatna a program.

A catkin\_ws mappában a környezet felállítása közben létrejött fájlokat tartalmazza, amiket fejlesztői célokból kivezettem a forrásba, ezáltal könnyen módosíthatóak, és követhetőek.

A docs mappában különböző fotók, videók és gráfok találhatóak, amelyek a program működését mutatja be. Ezek a képek megtalálhatóak lesznek ebben a szakdolgozatban.

# **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**



*TODO. 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.

*TODO. 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.



*TODO. ábra: launch.json felépítése*

**

*TODO. 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.



TODO. 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.



TODO. 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.

TODO. 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.



TODO. ábra: A jutalom függvénynek definiálása

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



TODO. 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.



TODO. 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. Use Case-k**

TODO Use Case főmenü kreálás

TODO Program közbeni use case diagramm

# **3.7. Rendszerterv**

# **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: [9]
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:



*TODO. 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.



*TODO. 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.

# **3.9. Tesztkörnyezet**

A tesztkörnyezet a fejlesztői környezetben megtalálható lesz egyből. Egybe van importálva a source fájlokkal, amelyen belül a tests mappán belül találhatóak meg a tesztek. Mivel a teszthez különböző könyvtárakat használunk, ezért csak a saját környezetünket lehet tesztelni és azt is csak limitált módon. Mivel a GUI része a programnak egyáltalán nem fontos, ezért a manuális tesztelés helyett a fejlesztői környezetet lehet használni tesztelésre.

# **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 tudjuk 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, a saját környezetünket fogja tesztelni, mivel itt történnek azok a hívások, amelyek tesztelhetőek lesznek, a program többi részét a manuális teszteléssel alaposan át tudjuk nézni. Ezt a környezetet először a teszt elején létre kell hozni, ezt a unittest setUp metódusában fogjuk tudni megtenni, ez 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 feljesztőnek.



TODO. ábra: test.py fájl felépítése

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

cd src/tests

Ezután a python3 test.py parancs futtatásával fog elindulni a program, 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.



TODO. ábra: test.py fájl futása közben bukott teszt

## **3.10.3. Tesztelési eredmények**



TODO. ábra: test.py fájl helyes lefutása

Ha a környezet megfelelően működik, akkor minden egyes automatikus tesztnek jónak kell lennie. Ez elengedhetetlen a program megfelelő működésével, mivel minden egyes funkció a környezettel kommunikál. A programon belüli funkciókat manuális teszteléssel lehet megnézni, de ha a fejlesztő nem írt át még semmit se a kódban, akkor a manuális tesztelés elengedhető, mivel minden egyes funkció, amely a kiadott verzióban található, az alapos manuális tesztelésen esett át.

## **3.10.4. Tesztek részletes leírása**

// TODO

# **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

*TODO. ábra: A projekt „Github” oldala*

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

*TODO. 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 [10] 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.

*TODO. 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.



*TODO. 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 és létrehoztam egy olyan alapot, amelyet nagyon könnyen lehet tovább fejleszteni és a megfelelő fejlesztői környezet létrehozásával, pedig egy könnyű környezetet kap képhez a fejlesztő, ahol a megfelelő tesztelésnek és dokumentálásnak köszönhetően, könnyen megtudja érteni a program működését erről az oldalról is. TODO

# **4.1. További fejlesztési lehetőségek**

Jelenleg a program a komplikált környezet miatt, 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 user 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 teljesítmény, és így a felhasználó kevesebb idő alatt többet tudna elérni a tanulásban.

# **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. „Official download of VLC media player, the best Open Source player - VideoLAN” [Online]. Availaible: <https://www.videolan.org/vlc/> [Hozzáférés dátuma: 2025. 03. 04.]
9. „Visual Studio Code” [Online]. Availaible: <https://code.visualstudio.com/docs/setup/linux> [Hozzáférés dátuma: 2025. 04. 16.]
10. „Loguru – Python logging made (stupidly) simple” [Online]. Availaible:

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