Određivanje kombinacije zaključane brave primjenom strojnog učenja

Kramar, Filip

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:930957

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-24**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

Određivanje kombinacije zaključane brave primjenom strojnog učenja

Završni rad

Filip Kramar

Osijek, 2022.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 31.08.2022.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime Pristupnika:	Filip Kramar
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	R 4371, 22.07.2019.
OIB Pristupnika:	39984804023
Mentor:	Izv. prof. dr. sc. Časlav Livada
Sumentor:	,
Sumentor iz tvrtke:	
Naslov završnog rada:	Određivanje kombinacije zaključane brave primjenom strojnog učenja
Znanstvena grana rada:	**Umjetna inteligencija (zn. polje računarstvo)**
Zadatak završnog rad:	U radu je potrebno opisati strojno učenje u Unity razvojnom okruženju te istu primijeniti na određivanje nasumične kombinacije brojeva. Odrediti složenost izvođenja algoritma promjenom duljine kombinacije potrebe na otključavanje brave. Tema rezervirana za: Filip Kramar
Prijedlog ocjene završnog rada:	Vrlo dobar (4)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 2 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	31.08.2022.
Datum potvrde ocjene od strane Odbora:	07.09.2022.

Potvrda mentora o predaji konačne verzije rada:	Mentor elektronički potpisao predaju konačne verzije.
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 08.09.2022.

Ime i prezime studenta:	Filip Kramar
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R 4371, 22.07.2019.
Turnitin podudaranje [%]:	12

Ovom izjavom izjavljujem da je rad pod nazivom**: Određivanje kombinacije zaključane brave primjenom strojnog učenja**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Časlav Livada

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1

1.1. Zadatak završnog rada 1

2. PREGLED PODRUČJA RADA 2

3. KORIŠTENI ALATI 3

3.1. Unity 3

3.2. Programski jezik C# 4

3.3. Unity *Machine Learning Agents* 4

3.3.1. Tipovi učenja 6

3.4. Tensorflow 8

4. IZRADA PROJEKTA 9

4.1. Izrada okruženja 9

4.2. Implementacija agenta 12

4.3. Testiranje agenta 19

4.4. Rezultati treniranja 21

5. ZAKLJUČAK 24

LITERATURA 25

SAŽETAK 27

ABSTRACT 28

ŽIVOTOPIS 29

1. UVOD

Strojno učenje vrsta je umjetne inteligencije koja računalima omogućuje učenje i bolje predviđanje ishoda bez eksplicitnog programiranja. Za vrijeme „učenja“ računala koriste algoritme koji ponavljaju isti zadatak više puta, ali proces se svaki put modificira sve dok se ne izvrši zadani zadatak. Zatim računalo traži uzorke u podacima kako bi kasnije mogao donijeti zaključke na temelju danih primjera i stečenih iskustava.

Strojno učenje postalo je jako važno u posljednjih par godina zbog mnoštva podataka s kojima se može susresti u svakodnevnom životu. Strojno učenje pomaže rješavati složene probleme koji koriste velike količine različitih podataka ljudi, lokacija, organizacija i problema. Također razvoj računalne tehnologije znatno je pomagao popularnosti strojnog učenja. Strojno učenje može riješiti brzo i razmjerno koje čovjek ne može.

Ovaj završni rad će istražiti alat za strojno učenje koji je dio Unity *game enginea*. Pomoću Unity *game enginea* mogu se kreirati virtualne simulacije te 2D i 3D igre u kojima se agenti mogu trenirati. Unity *Machine Learning Agents* (ML-Agents) koristi se za izradu zadatka ovog završnog rada. Zadatak agenta, koji će koristiti podržano učenje, je pronaći točnu kombinaciju brave koja se nakon svakog pronalaska ponovno nasumično generira.

U nastavku završnoga rada opisani su korišteni alati s naglaskom na alat za strojno učenje Unity *Machine Learning Agents*. U poglavlju nakon opisani su izrada projekta i rezultati.

1.1. Zadatak završnog rada

U radu je potrebno opisati strojno učenje u Unity razvojnom okruženju te isto primijeniti na određivanje nasumične kombinacije brojeva. Odrediti složenost izvođenja algoritma promjenom duljine kombinacije potrebne za otključavanje brave.

2. PREGLED PODRUČJA RADA

Umjetna inteligencija je dio računalne znanosti koja se neprestano razvija i usavršava, no uspješnost rješavanja problema ovisi o mogućnostima postojećih okolina te zbog toga zahtijeva stalan razvoj novih i boljih okolina što može zahtijevati puno novaca, vremena i znanja. Unity i alat ML-Agents omogućavaju stvaranja specifičnih okolina u svrhu istraživanja novih okolina i algoritama.

S obzirom na to da su Unity i alat ML-Agents besplatni, mogu ih koristiti studenti zainteresirani za strojno učenje. Tako su Mateo Bareš i David Hodak koristili strojno učenje u svojim projektima. Mateo Bareš je u svom diplomskom radu istrenirao agenta da izbjegne prepreke dok se kreće na cesti koja je nasumično generirana [1], dok je David Hodak naučio agenta naći skriveni objekt nasumično generiran u okolini [2].

Unity i ML-Agents dostupni su svim programerima koji žele isprobati strojno učenje, a ne samo studentima. Youtube korisnik Sumrak primjer je programera koji je stvorio okruženje u kojem je agent morao koristiti RayPerceptionSensorComponent3D za detekciju automobila kako bi prešao ulicu i izbjegao automobile [3].

Još jedan primjer je član Unity tima Hector Caballero koji je napravio igru u kojoj agent kontrolira auto i treba proći poligon bez sudara. S obzirom na složenost zadatka takvo okruženje predstavilo je pravi izazov za agenta [4].

Thomas Van Iseghem je napravio simulaciju parkirališta. U Unityju je simulirao parkiralište te naučio agenta da samostalno nađe mjesto na parkiralištu i parkira auto [5].

Iako nije koristio ML-Agents, Youtube korisnik Chrispresso uspio je naučiti agenta da prelazi određene razine u retro igri Super Mario Bros. Korišteni su genetski algoritmi i neuronska mreža, a agent je uspio naučiti iskorištavati greške u igri u svrhu bržeg prolaska razine [6].

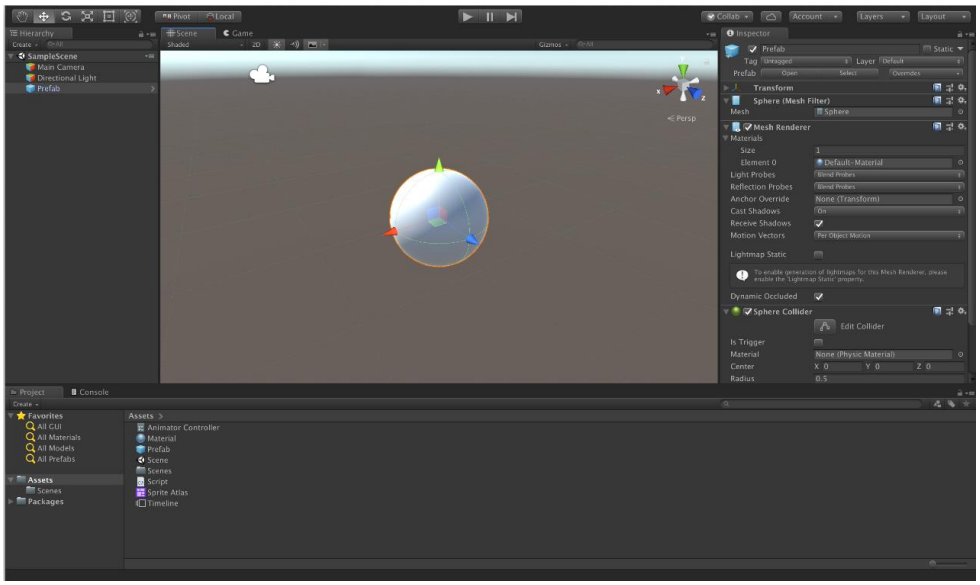
3. KORIŠTENI ALATI

3.1. Unity

Unity je najpopularniji svjetski *game engine*. Razvio ga je Unity Technologies u lipnju 2005. na Apple Worldwide Developers Conference kao pokretač igre za Mac OS X, no s vremenom se proširio na razne platforme za mobilne uređaje, osobna računala, virtualnu stvarnost i konzole. Koristi se za stvaranje trodimenzionalnih i dvodimenzionalnih igara, interaktivnih simulacija i slično. Program je napisan u C++ programskom jeziku, ali za razvijanje vlastitog sadržaja i korištenje raznih mogućnosti Unity game enginea, programeri pišu u C# programskom jeziku [7].

Na slici 3.1. nalazi se Unity sučelje u kojem se stvara sadržaj. Uvijek je prikazana jedna scena koja može predstavljati jednu razinu igre, okolinu simulacije i slično. Unutar scene mogu se stvarati razni jednostavni objekti pomoću kojih se manipulira komponentama kao što su C# skripte, RigidBody, veličina objekta, orijentacija objekta, položaj objekta i slično. Za kreiranje kompliciranijih objekata potrebno je koristiti program za modeliranje ili uzeti gotove komponente iz Unity assets storea [8].

Stručnjaci preporučuju Unity početnicima zbog njegove fleksibilnosti, pokrivanja razvoja raznih medija, jednostavnosti, velikom količinom gotovih dodataka i biblioteka. Jedan od dodataka je Unity ML-Agents za strojno učenje na kojem se ovaj rad temelji.



Sl.3.1.Unity uređivač

3.2. Programski jezik C#

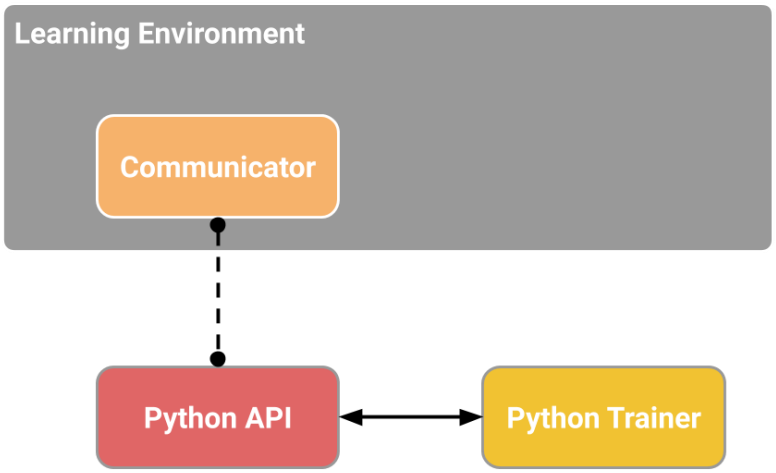
C# je objektno orijentirani programski jezik koji je jednostavan, moderan i univerzalno primjenjiv. Koristi se za razvoj web i desktop aplikacija, mobilnih aplikacija, igara i još mnogo toga. Sintaksa C# slična je sintaksi C, Javi i C++. što dozvoljava jednostavniju prilagodbu programerima stručnima u tim programskim jezicima. Pojednostavljuje kompleksnost C++ programskog jezika te omogućuje obilježja kao što su višenitnost, svojstva i događaje, uvjetnu kompilaciju, lambde i automatsko upravljanje memorijom.

Programski jezik C# temelji se na svojstvima enkapsulacije, nasljeđivanja i polimorfizma. Enkapsulacija znači spajanje podataka s metodama koje rade na tim podacima, ili ograničavanje izravnog pristupa nekim komponentama objekta. Klasa smije naslijediti najviše jednu roditeljsku klasu, ali neograničen broj sučelja. Kada klasa naslijedi neku klasu, klasa koja je naslijedila može koristiti varijable i metode roditeljske klase. Klase i metode ne moraju biti napisane određenim redoslijedom što omogućuje neograničen broj klasa, sučelja i slično. Pomoću polimorfizma definiramo više različitih, ali srodnih operacija tj. dozvoljava postojanje metode istih imena s različitim ulaznim parametrima [9].

Kao i Unity, C# ima puno biblioteka koji pružaju gotove funkcije koje se mogu koristiti u drugim programima i lako implementirati u druge projekte.

3.3. Unity *Machine Learning Agents*

Unity Machine Learning Agents Toolkit (ML-Agents) je projekt otvorenog koda koji pruža implementiranje najsuvremenijih algoritama strojnog učenja kako bi omogućili programerima igara da lako treniraju inteligentne agente za 2D, 3D i VR/AR igre. Za treniranje agenata može se također koristiti Python API pomoću učenja s pojačanjem, učenja imitacijom, neuroevolucije ili bilo koje druge metode [10].



Sl. 3.2. Glavne komponente ML-Agentsa [11]

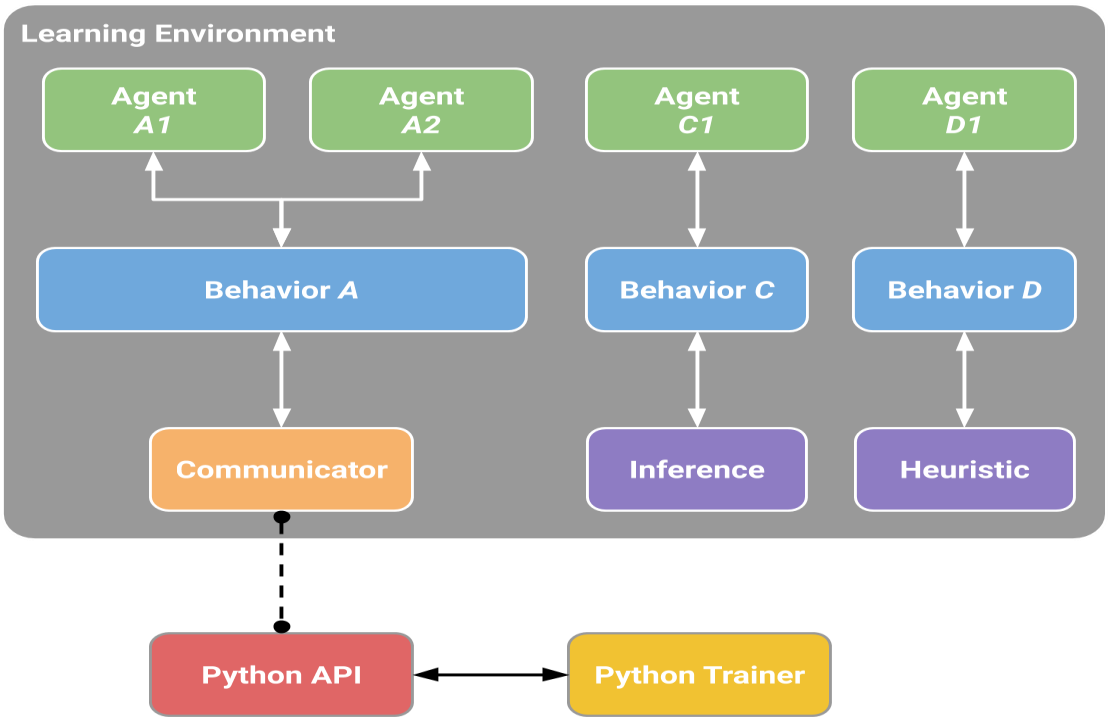
ML-Agents sastoji se od četiri glavnih komponenti (Slika 3.2.):

- **Okolina za učenje *(engl. Learning Enviroment)*** - sadrži scenu Unity i sve likove igre. U okolini agenti promatraju, djeluju i uče.
- **Python API** - sadrži Python sučelje niske razine za interakciju i manipuliranje okruženjem za učenje. Python API ne pripada Unity okruženju, već živi van njega i rukuje s Unityjem kroz komunikator.
- **Vanjski komunikator (engl. *Communicator*)** - dio okoline za učenje koji povezuje okolinu za učenje s Python API-jem niske razine
- **Python treneri (engl. *Python Trainer*)** - sadrže sve algoritme strojnog učenja koji omogućuju treniranje agenata

Okruženje za učenje sadrži dvije Unity komponente koje pridonose ograniziranju Unity scene, a to su:

- **Agenti (engl. Agents)** - priključeni su sa bilo kojim Unity *GameObject* likom unutar scene i bave se generiranjem svojih opažanja, izvođenjem primljenih radnji i dodjeljivanjem nagrada. Mora biti povezan s ponašanjem.
- **Ponašanja (engl. Behaviors)** - definiraju specifikacije agenta kao što je npr. broj koraka koje agent može napraviti u epizodi. Ponašanje se može smatrati funkcijom koja prima agentove nagrade i zapažanja te vraća akcije. Postoje tri vrste ponašanja: učenje, heuristika ili zaključivanje. Ponašanje učenja je nedefinirano, ali spremno za treniranje. Heurističko ponašanje (engl. *Heuristic*) je definirano čvrsto kodiranim skupom pravila implementiranih

u kodu. Ponašanje zaključivanja sadrži datoteku neuronske mreže. Nakon što se uvježba ponašanje učenja, ono postaje ponašanje zaključivanja.



Sl. 3.3.Primjer više ponašanja i agenata[11]

Okolina učenja može sadržavati više agenata i ponašanja u isto vrijeme (Slika 3.3.). Agenti mogu imati isto ponašanje, ako su njihova opažanja i akcije isti. [11].

3.3.1. Tipovi učenja

Da bi se trenirao agent unutar okruženja ML-Agents, tri entiteta moraju biti definirana u svakom trenutku igre;

- **Opažanja** – vrijednosti koje agent očitava iz okoline. Opažanja mogu biti numerička i/ili vizualna. Numerička opažanja mjere atribute okoline s agentove točke gledališta, a vizualna opažanja su slike koje proizvode kamere spojene na agenta koje predstavljaju ono što agent vidi u tom trenutku. Ovisno o kompliciranosti željene simulacije, opažanja će biti kontinuirana ili diskretna.
- **Akcije** – radnje koje agent može napraviti. Kao i opažanja, ovisno o kompliciranosti željene simulacije akcije mogu biti kontinuirane ili diskretne.

Signali nagrade - vrijednost koja pokazuje uspjeh agenta. Signali nagrade koriste se samo kada agent napravi neku akciju dobru ili lošu tj. hoće li dobiti pozitivnu ili negativnu nagradu. Treba

imati na umu da pomoću signala nagrade se priopćuju ciljevi zadatka agentu, te ih treba postaviti tako da se maksimiziranjem nagrade generira željeno ponašanje [11].

Podržano učenje

ML-Agents omogućuje implementaciju dvaju algoritama podržanog učenja *Proximal Policy Optimization* (PPO) i *Soft Actor Critic* (SAC). U ovom radu korišten je PPO algoritam zato što se pokazao općenitijim i stabilnijim u usporedbi s drugim algoritmima podržanog učenja. S PPO može se jednostavno implementirati funkcija troškova, pokrenuti gradijentni spust na njoj i biti vrlo siguran da će se dobiti izvrsni rezultati s relativno malim podešavanjem hiperparametara.

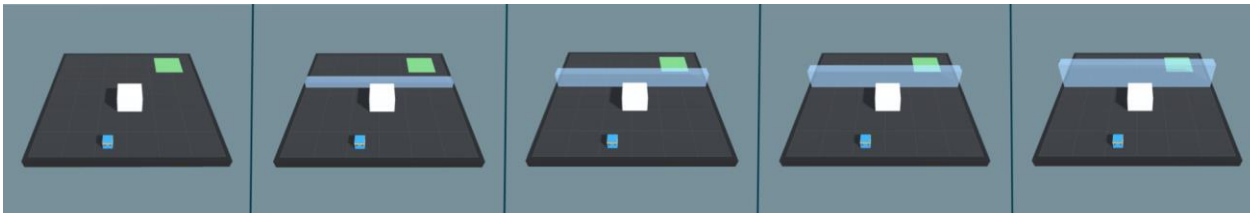
Za razliku od PPO-a, SAC može učiti iz iskustava prikupljenih u bilo kojem trenutku u prošlosti (eng. off-policy). Kako se iskustva prikupljaju, ona se stavljaju u međuspremnik za ponavljanje iskustva i nasumično izvlače tijekom treninga. Ovo čini SAC znatno učinkovitijim u pogledu uzorkovanja, često zahtijevajući 5-10 puta manje uzoraka za učenje istog zadatka kao PPO. Međutim, SAC obično zahtijeva više ažuriranja modela. SAC je dobar izbor za teža ili sporija okruženja [11].

Učenje imitacijom

Nekad je jednostavnije demonstrirati ponašanje koje želimo od agenta, umjesto da ga se pokušava naučiti korištenjem metoda pokušaja i pogreške. To se radi tako što se agentu daju primjeri zapažanja iz igre i akcije iz stvarnog svijeta pomoću kontrolera kako bi se vodilo agentovo ponašanje. Učenje imitacijom može se koristiti samostalno ili u kombinaciji s podržanim učenjem. Koristi se samostalno ako se agenta treba trenirati specifičan tip ponašanja [11].

Učenje s kurikulumom

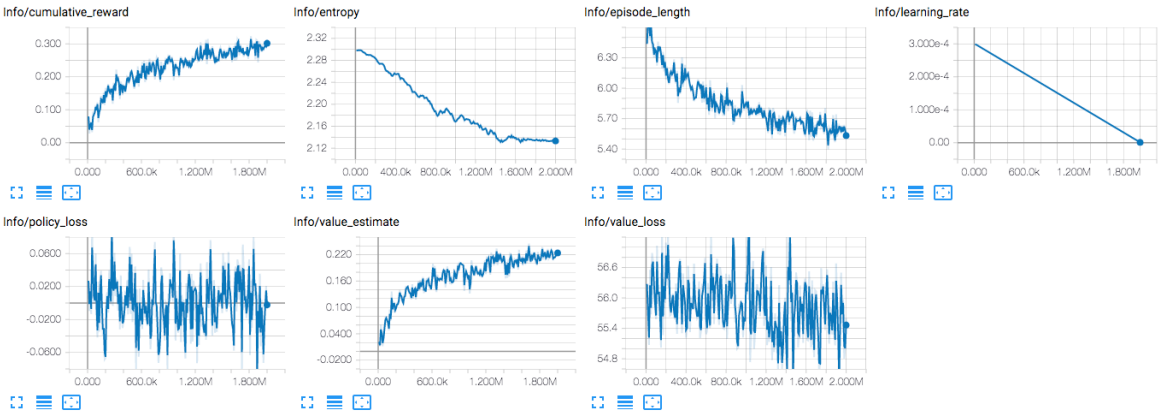
Učenje s kurikulumom najsličnije je ljudskom učenju. Agenta se prvo uči jednostavniji problemi koji se postepeno otežavaju. Na slici 3.4. nalazi se primjer učenja s kurikulumom. Prvo se agenta nauči doći do cilja bez zida i onda se postepeno povećava visina zida sve dok agent ne nauči riješiti i najteži slučaj [11].



Sl. 3.4. Primjer učenja s kurikulumom

3.4. Tensorflow

TensorFlow je besplatna biblioteka otvorenog koda za strojno učenje i umjetnu inteligenciju. Može se koristiti za razne zadatke, ali je namijenjen za duboko uvježbavanje neuronske mreže i zaključivanje. TensorFlow je razvijen od strane Google Brain tima kao pomoć u istraživanju i proizvodnji. Može se koristiti u raznim programskim jezicima kao što su Python, C++, Javascript, i Java. Izvodi se na procesorima i grafičkim procesorima. ML-Agents koristi TensorFlow za izgradnju modela kojeg dobijemo na kraju treniranja kao .onnx datoteku koja predstavlja model neuronske mreže agenta [12].



Sl. 3.5.Primjer TensorBoarda

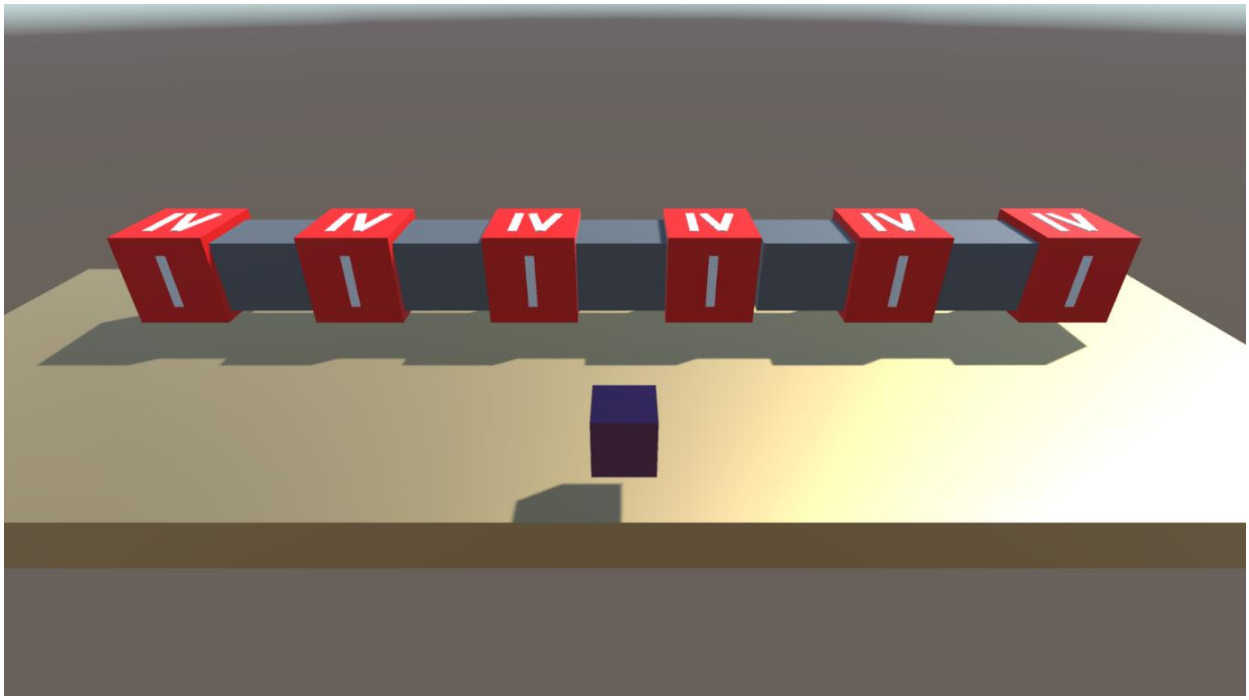
TensorBoard je Tensorflow alat koji služi za vizualizaciju određenih atributa tijekom treniranja. Izgled TensorBoarda ovisi o parametrima koje postavimo u konfiguracijskoj datoteci treniranja. Na slici 3.5. nalazi se primjer TensorBoard sučelja nakon treniranja [13].

4. IZRADA PROJEKTA

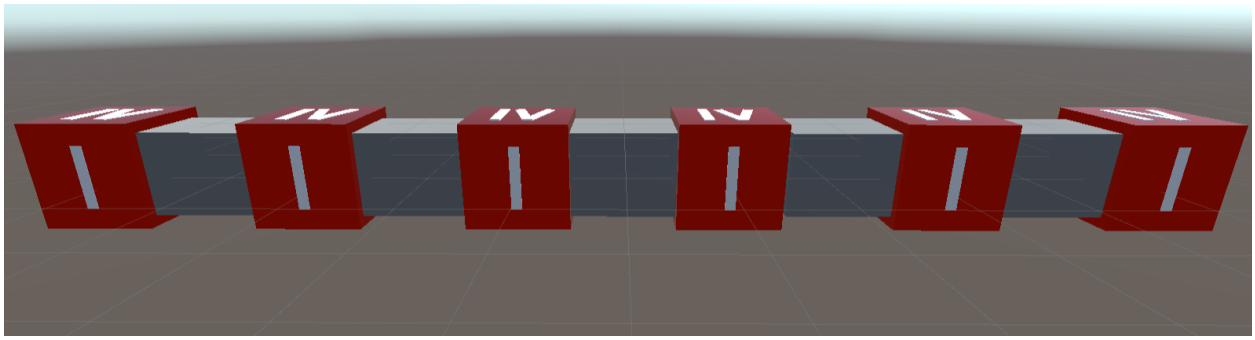
Projekt se sastoji od okruženja stvorenog u programskom okruženju Unity u kojem će biti testirane opcije ML-Agents alata i agenta kako bi se pronašla prava kombinacija brave.

4.1. Izrada okruženja

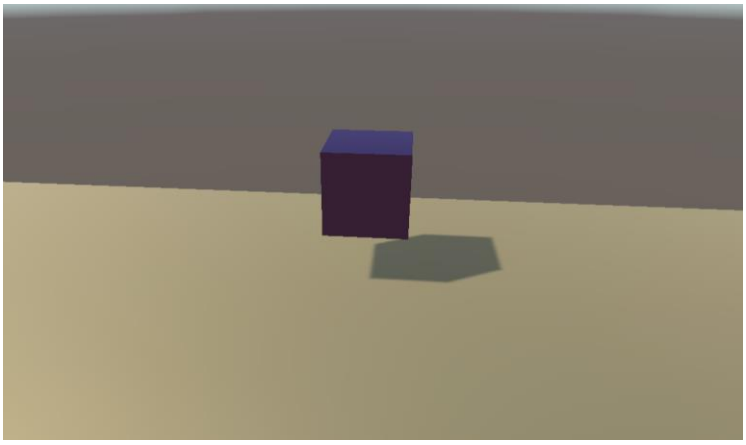
Na slici 4.1. nalazi se scena simulacije Unity uređivača. Scena je osmišljena kao platforma koja sadrži objekt koji predstavlja bravu (Slika 4.2.) i objekt koji predstavlja agenta (Slika 4.3.).Svi dijelovi okoline su napravljeni pomoću *GameObject* alata koji sadrži temeljne objekte u Unityju za izradu likova, rekvizita i krajolika.



Sl. 4.1.Izgled okoline u Unityu

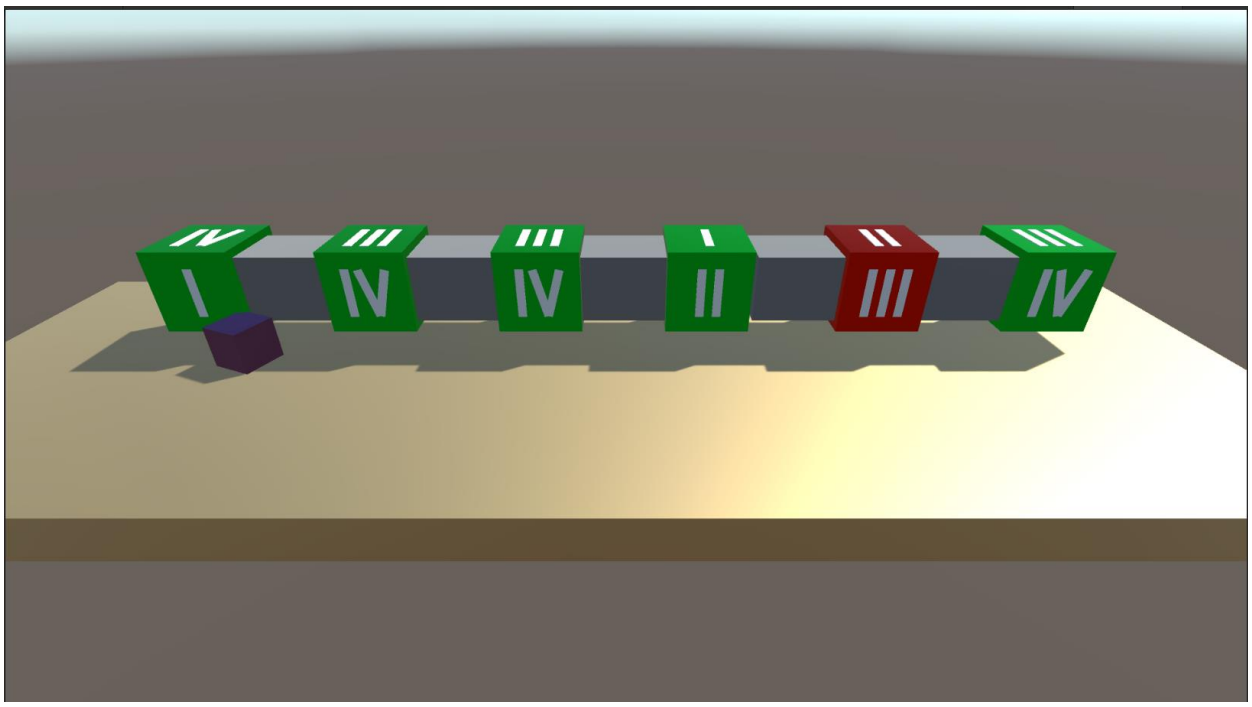


Sl. 4.2.Objekt koji predstavlja bravu



Sl. 4.3.Objekt koji predstavlja agenta

Na slikama 4.5. i 4.6. vidi se funkcionalnost brave. Agent mora doći u kontakt sa znamenkom kombinacije kako bi se okrenula i pokrenula događaj koji provjerava valjanost te znamenke s točnom kombinacijom. Ako je znamenka točna pokreće se novi događaj u kojem znamenka mijenja materijal u zelenu boju i ne može se više rotirati (Slika 4.4.). Kada agent završi epizodu, znamenka se restartira tj. vraća se u crvenu boju i može se opet rotirati.



Sl. 4.4.Okolina sa pet točnih znamenki i jednom netočnom

```
1:  public class Rotation : MonoBehaviour
2:  {
3:      [SerializeField] private Material correctMaterial;
4:      [SerializeField] private Material incorrectMaterial;
5:      [SerializeField] private GameObject objectToChange;
6:      private int CurrentNumber;
7:      private bool CanRotate;
        public static event Action<string, int> RotationDetected =
8:  delegate{ };
9:      void Start()
10:     {
11:         CurrentNumber = 1;
12:         CanRotate = true;
13:         FourPinAgent.Reset += ResetPin;
14:         FivePinAgent.Reset += ResetPin;
15:         SixPinAgent.Reset += ResetPin;
16:         FourPinAgent.CorrectNumber += ChangeMaterial;
17:         FivePinAgent.CorrectNumber += ChangeMaterial;
18:         SixPinAgent.CorrectNumber += ChangeMaterial;
19:     }
20:     void OnTriggerEnter()
21:     {
22:         if (this.CanRotate)
23:         {
24:             transform.Rotate(0f, 0f, 90f);
25:             CurrentNumber++;
26:             if (CurrentNumber > 4)
27:                 CurrentNumber = 1;
28:             RotationDetected(name, CurrentNumber);
29:         }
30:     }
31:     private void ResetPin()
32:     {
            objectToChange.GetComponent<MeshRenderer>().material =
33: incorrectMaterial;
34:         CanRotate = true;
35:     }
36:     private void ChangeMaterial(string PinName)
37:     {
38:         if (PinName == name)
39:         {
                objectToChange.GetComponent<MeshRenderer>().material =
40: correctMaterial;
41:             CanRotate = false;
42:        }
43:    }
```

Sl.4.5.Programski kod funkcionalnosti brave

```
44:  private void OnDestroy()
45:      {
46:           FourPinAgent.CorrectNumber -= ChangeMaterial;
47:           FourPinAgent.Reset -= ResetPin;
48:           FivePinAgent.CorrectNumber -= ChangeMaterial;
49:           FivePinAgent.Reset -= ResetPin;
50:           SixPinAgent.CorrectNumber -= ChangeMaterial;
51:           SixPinAgent.Reset -= ResetPin;
52:       }
53:  }
```

Sl. 4.6. Metoda OnDestroy klase brave

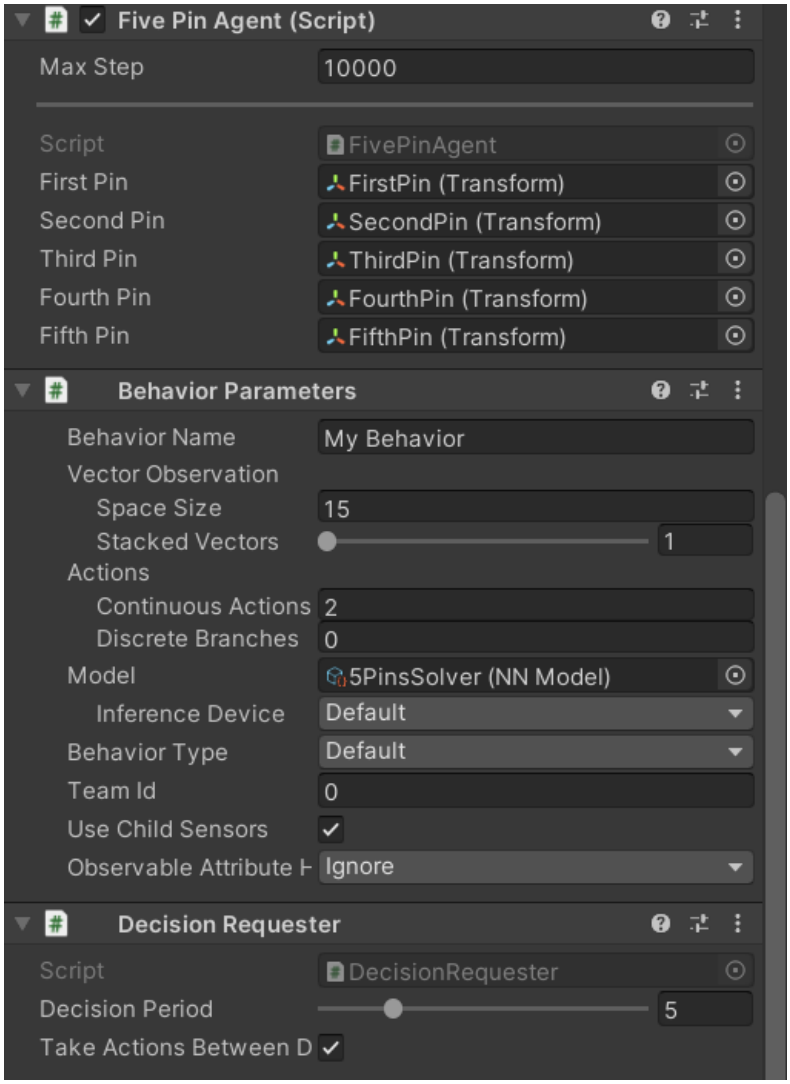
4.2. Implementacija agenta

Kako bi se implementirao agent, mora se izraditi posebna C# skripta. Klasa u skripti mora naslijediti unaprijed definiranu klasu *Agent* koja je dio paketa ML-Agents i prepisati metode. Metode koje se moraju prepisati su *CollectObservations* i *OnActionReceived*, a opcionalne za prepisati su *Heuristic* i *OnEpisodeBegin*. Pomoću *OnActionReceived* metode što će agent raditi svaki put kada donese odluku. Kako bi agent mogao donijeti odluke mora mu se dodati komponentu *Decision Requester* koja periodično traži donošenje odluke. Metoda *OnEpisodeBegin* postavlja vrijednosti podataka na početku svake epizode. *CollectObservations* je metoda pomoću koje agent skuplja opservacije za vrijeme treniranja tj. informacije koje će agent skupljati svaki korak iz okoline. *Heuristic* služi za testiranje okoline prije treniranja tako što igrač kontrolira agenta umjesto istreniranog modela neuronske mreže i provjerava funkcionalnost okoline. Kada se skriptu primjeni na objekt koji predstavlja objekt (Slika 4.3.) automatski će biti dodana i komponenta *Behavior Parameters* (Slika 4.7.).

Behavior Parameters sastoji se od:

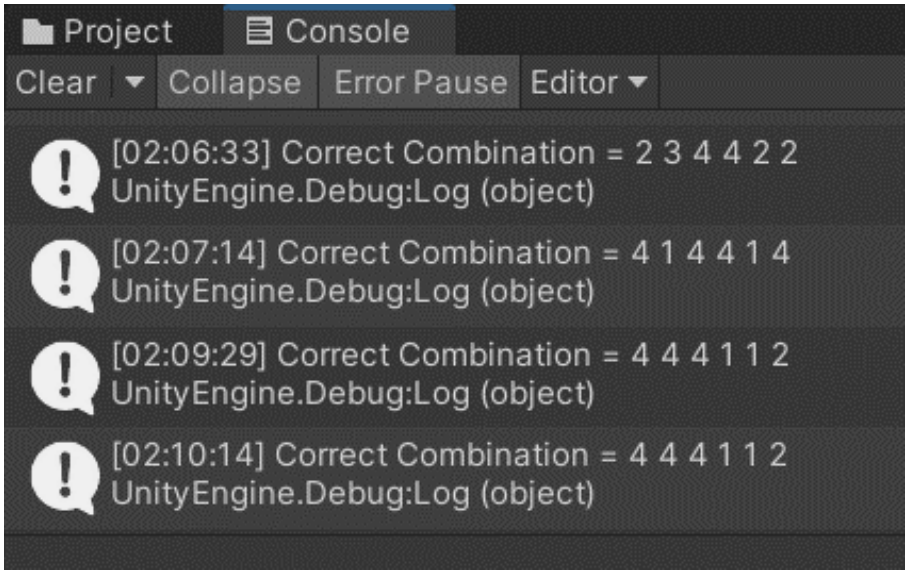
- *Vector Observation* – veličina u kojoj se definira koliko informacija agent prima iz okoline. U ovom slučaju je tri po znameki
- *Actions* – definira tipove vrijednosti koje agent generira i pomoću njih naš agent se kreće. U ovom slučaju agent definira dvije kontinuirane vrijednosti, jednu za x os i jednu za z os
- *Model* – sadrži datoteku neuronske mreže nastale treniranjem agenta. Parametar *Inference Device* ovisi o tom trenira li se pomoću procesora (*CPU*) ili grafičke procesorske jedinice (*GPU*)

- *Behaviour Type* – može biti *inference only* ili *heuristic only*. Ako je ponašanje na *heuristic only* onda igrač kontrolira agenta i testira agenta i okolinu, dok *inference only* koristi istrenirani model neuronske mreže kako bi kontrolirali agenta [14].



Sl. 4.7. Parametri komponenti agenta

U *CollectObservations* (Slike 4.9. i 4.10.) agent iz okoline opaža je li znamenka točna ili ne, ukoliko nije dobiva dodatno informaciju udaljenosti agenta do znamenke i ponavlja isti postupak za svaku znamenku. Metodom *OnActionReceived* (Slika 4.11.) definira se što će agent raditi s vrijednostima generiranima u *Actions* iz *Behaviour Parameters*. Generirane vrijednosti pridružuje novim vrijednostima x i z vrijednosti vektora agenta. Y vrijednost ne treba mijenjati zato što je y vrijednost konstantna. Ako tijekom kretanja agent padne sa platforme, dobiva negativnu nagradu kako bi ga odučili od takvog ponašanja i epizoda završava. Pomoću varijable *Max Step* postavlja se koliko koraka agent smije napraviti prije nego epizoda prisilno završi kako se ne bi naučio samo ne pasti s platforme. Prilikom kontakta s pinom aktivira se metoda *Check* (Slika 4.12.) koja provjerava koja znamenka je bila u kontaktu i je li broj prikazan isti kao točan broj. Agent dobiva pozitivnu nagradu ukoliko je prikazani broj točan. Ako je suma nagrada jednaka broju znamenke, epizoda završava tj. restartira se. Kako bi se otežalo agentu rješavanje zadatka prilikom svake epizode, kombinacija se nasumično generira u metodi *OnEpisodeBegin* (Slika 4.9.). Točna kombinacija se uvijek može provjeriti u porukama konzole (Slika 4.8.).



Sl. 4.8.Ispis točne kombinacije

```
1:  public class SixPinAgent : Agent
2:  {
3:      [SerializeField] private Transform firstPin;
4:      [SerializeField] private Transform secondPin;
5:      [SerializeField] private Transform thirdPin;
6:      [SerializeField] private Transform fourthPin;
7:      [SerializeField] private Transform fifthPin;
8:      [SerializeField] private Transform sixthPin;
9:      public static event System.Action<string> CorrectNumber =
    delegate { };
10:     public static event System.Action Reset = delegate { };
11:     private int[] CurrentCombination, CorrectCombination;
12:     private bool[] IsCorrect;
13:     void Start()
14:     {
15:         CurrentCombination = new int[] { 1, 1, 1, 1, 1,1 };
16:         Rotation.RotationDetected += Check;
17:     }
18:     public override void OnEpisodeBegin()
19:     {
20:         this.transform.localPosition = new Vector3(-1, 0f, 0.3f);
21:         IsCorrect = new bool[] { false, false, false, false,
    false,false };
22:         Reset();
23:         CorrectCombination = new int[] { Random.Range(1, 5),
    Random.Range(1, 5), Random.Range(1, 5), Random.Range(1, 5),
    Random.Range(1, 5),Random.Range(1, 5) };
24:         Debug.Log($"Correct Combination = {CorrectCombination[0]}
    {CorrectCombination[1]} {CorrectCombination[2]}
    {CorrectCombination[3] } {CorrectCombination[4]}
    {CorrectCombination[5]}");
25:     }
26:     public override void CollectObservations(VectorSensor sensor)
27:     {
28:         sensor.AddObservation(IsCorrect[0] ? 0 : 1);
29:         if (!IsCorrect[0])
30:         {
31:             Vector3 dirToThePin = (firstPin.transform.localPosition -
    this.transform.localPosition).normalized;
32:             sensor.AddObservation(dirToThePin.x);
33:             sensor.AddObservation(dirToThePin.z);
34:         }
35:         else
36:         {
37:             sensor.AddObservation(0f);
38:             sensor.AddObservation(0f);
39:         }
40:         sensor.AddObservation(IsCorrect[1] ? 0 : 1);
```

Sl. 4.9. Metode *OnEpisodeBegin i CollectObservations*

```
41:         if (!IsCorrect[1])
42:         {
43:             Vector3 dirToThePin = (secondPin.transform.localPosition - this.transform.localPosition).normalized;
44:             sensor.AddObservation(dirToThePin.x);
45:             sensor.AddObservation(dirToThePin.z);
46:         }
47:         else
48:         {
49:             sensor.AddObservation(0f);
50:             sensor.AddObservation(0f);
51:         }
52:         sensor.AddObservation(IsCorrect[2] ? 0 : 1);
53:         if (!IsCorrect[2])
54:         {
55:             Vector3 dirToThePin = (thirdPin.transform.localPosition - this.transform.localPosition).normalized;
56:             sensor.AddObservation(dirToThePin.x);
57:             sensor.AddObservation(dirToThePin.z);
58:         }
59:         else
60:         {
61:             sensor.AddObservation(0f);
62:             sensor.AddObservation(0f);
63:         }
64:         sensor.AddObservation(IsCorrect[3] ? 0 : 1);
65:         if (!IsCorrect[3])
66:         {
67:             Vector3 dirToThePin = (fourthPin.transform.localPosition - this.transform.localPosition).normalized;
68:             sensor.AddObservation(dirToThePin.x);
69:             sensor.AddObservation(dirToThePin.z);
70:         }
71:         else
72:         {
73:             sensor.AddObservation(0f);
74:             sensor.AddObservation(0f);
75:         }
76:         sensor.AddObservation(IsCorrect[4] ? 0 : 1);
77:         if (!IsCorrect[4])
78:         {
79:             Vector3 dirToThePin = (fifthPin.transform.localPosition - this.transform.localPosition).normalized;
80:             sensor.AddObservation(dirToThePin.x);
81:             sensor.AddObservation(dirToThePin.z);
82:         }
83:         else
```

Sl. 4.10.Nastavak metode *CollectObservations*

```
84:         {
85:             sensor.AddObservation(0f);
86:             sensor.AddObservation(0f);
87:         }
88:         sensor.AddObservation(IsCorrect[5] ? 0 : 1);
89:         if (!IsCorrect[5])
90:         {
91:             Vector3 dirToThePin = (sixthPin.transform.localPosition - this.transform.localPosition).normalized;
92:             sensor.AddObservation(dirToThePin.x);
93:             sensor.AddObservation(dirToThePin.z);
94:         }
95:         else
96:         {
97:             sensor.AddObservation(0f);
98:             sensor.AddObservation(0f);
99:         }
100:     }
101:     public override void OnActionReceived(ActionBuffers movements)
102:     {
103:         float xOrinentation = movements.ContinuousActions[1];
104:         float xOrientation = movements.ContinuousActions[0];
105:         float accelerationSpeed = 2f;
106:         this.transform.localPosition += new Vector3(xOrientation, 0, zOrinentation) * Time.deltaTime * accelerationSpeed;
107:         if (this.transform.localPosition.y < -1)
108:         {
109:             AddReward(-1f);
110:             EndEpisode();
111:         }
112:     }
113:     public override void Heuristic(in ActionBuffers movementsOut)
114:     {
115:         ActionSegment<float> continuousActionsOut = movementsOut.ContinuousActions;
116:         continuousActionsOut[0] = -Input.GetAxisRaw("Horizontal");
117:         continuousActionsOut[1] = Input.GetAxisRaw("Vertical");
118:     }
119:     private string SetName(int position)
120:     {
121:         switch (position)
122:         {
123:             case 0:
124:                 return "FirstPin";
125:                 break;
126:             case 1:
```

Sl. 4.11.Metode *OnActionReceived, Heuristic i SetName*

```
127:                 return "SecondPin";
128:                 break;
129:             case 2:
130:                 return "ThirdPin";
131:                 break;
132:             case 3:
133:                 return "FourthPin";
134:                 break;
135:             case 4:
136:                 return "FifthPin";
137:                 break;
138:             default:
139:                 return "SixthPin";
140:                 break;
141:         }
142:     }
143:     private void Check(string pinName, int pinNumber)
144:     {
145:         switch (pinName)
146:         {
147:             case "FirstPin":
148:                 CurrentCombination[0] = pinNumber;
149:                 CheckPin(0);
150:                 break;
151:             case "SecondPin":
152:                 CurrentCombination[1] = pinNumber;
153:                 CheckPin(1);
154:                 break;
155:             case "ThirdPin":
156:                 CurrentCombination[2] = pinNumber;
157:                 CheckPin(2);
158:                 break;
159:             case "FourthPin":
160:                 CurrentCombination[3] = pinNumber;
161:                 CheckPin(3);
162:                 break;
163:             case "FifthPin":
164:                 CurrentCombination[4] = pinNumber;
165:                 CheckPin(4);
166:                 break;
167:             case "SixthPin":
168:                 CurrentCombination[5] = pinNumber;
169:                 CheckPin(5);
170:                 break;
```

Sl. 4.12.Nastavak metode *SetName* i metoda *Check*

```
171:         private void CheckPin(int PinNumber)
172:         {
173:             if (CorrectCombination[PinNumber] ==
     CurrentCombination[PinNumber])
174:             {
175:                 CorrectNumber(SetName(PinNumber));
176:                 AddReward(1f);
177:                 IsCorrect[PinNumber] = true;
178:                 if (GetCumulativeReward() == 6f)
179:                     EndEpisode();
180:             }
181:         }
182:         private void OnDestroy()
183:         {
184:             Rotation.RotationDetected -= Check;
185:         }
186:     }
```

Sl. 4.13.Metode *CheckPin* i *OnDestroy*

4.3. Testiranje agenta

Za treniranje agenta koristi se Python okruženje u koje su prethodno instalirani dijelovi ML-Agents Python paketa koji omogućuju korištenje naredbe *mlagents-learn* i komunikaciju s agentom u svrhu generiranja iskustava. Na temelju tih iskustava izgrađena je neuronska mreža za optimizaciju pravila dana agentu. Kako bi se osiguralo uspješnije i stabilnije treniranje potrebno je postaviti parametre u .yaml datoteci. Na slici 4.14. nalazi se .yaml datoteka korištena za treniranje agenta.

Parametri na koje treba obratiti pozornost su:

- *trainer_type* - može biti `ppo`, `sac`, or `poca`. Ovisno koji tip trenera trener će se ponašati drukčije.
- *summary_freq* - broj iskustava koje agent treba prikupiti kako bi pokazao statistiku treniranja. Određuje kako će *TensorBoard* graf izgledati.
- *max_steps* - broj koraka napravljenih u okolini prije kraja treniranja.
- *time_horizon* - broj koraka agenta za prikupljanje iskustava prije nego se iskustva dodaju u spremnik iskustava. Ako agentov broj koraka dosegne ograničenje prije završetka epizode, očekivana ukupna nagrade procjenjuje se na temelju iz trenutnog statusa agenta.
- *num_layers* - broj skrivenih slojeva neuronske mreže. Što je problem kompleksniji to će više slojeva biti.

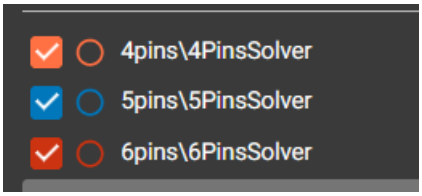
- *hiden_units* - broj članova u pojedinačnom sloju neuronske mreže. Ako opažanja možemo lagano mapirati akcijama ova vrijednost bit će mala.
- *learning_rate* - stopa učenja korištena za ažuriranje modula intrinzične znatiželje. Broj treba smanjiti ako su trening i gubitak znatiželje nestabilni.
- *batch_size* - broj iskustava u svakoj iteraciji postepenog spuštanja. Broj mora biti znatno manji od parametra *buffer_size*.
- buffer_size - broj iskustava koje agent treba prikupiti prije početka učenja ili ažuriranja modela. Broj treba biti višekratnik batch_size parametra.
- *beta* - ovaj parametar čini politiku „nasumičnom“ tj. što je ovaj parametar viši, to će agent više istraživati okolinu.
- *epsilon* – utječe na brzinu razvijanja politike za vrijeme treniranja. Služi kako se politika ne bi ažurirala drastično ako dođe do neuobičajenih iskustava.
- *num_epoch* - broj prolaza kroz međuspremnik iskustva prilikom izvođenja optimizacije gradijenta spuštanja. Što je veća veličina *batch_size* parametra, to je veća prihvatljiva veličina ovog parametra [15].

```
1: behaviors:
2:   6PinsSolver:
3:     trainer_type: ppo
4:     hyperparameters:
5:       batch_size: 1024
6:       buffer_size: 10240
7:       learning_rate: 0.0003
8:       beta: 0.005
9:       epsilon: 0.2
10       num_epoch: 3
11       learning_rate_schedule: linear
12     network_settings:
13       normalize: false
14       hidden_units: 128
15       num_layers: 2
16     max_steps: 500000
17     time_horizon: 64
18     summary_freq: 50000
```

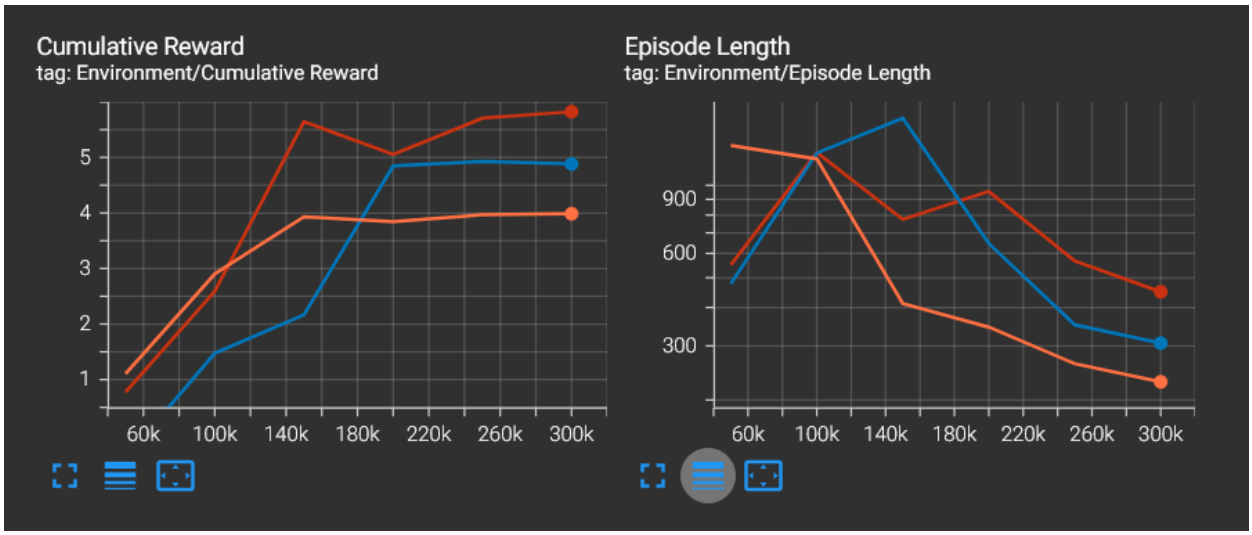
Sl. 4.14.Configuration.yaml datoteka

4.4. Rezultati treniranja

Rezultati se mogu prikazati pomoću TensorBoard-a vizualnih grafova. Grafove se može pratiti tijekom obuke agenta ili nakon obuke. Graf *Cumulative Reward* (Slika 4.16.) pokazuje srednju kumulativnu nagradu agenta nakon *summary_freq* koraka. Iz grafa se vidi kako agent u početku dobiva jako nisku srednju vrijednost kumulativnih nagrada te kako s vremenom ta vrijednost raste. Kako agent nikada neće biti savršen ova vrijednost nikada neće dostići maksimalnu vrijednost. Graf *Episode Length* (Slika 4.16.) prikazuje srednju vrijednost trajanja svake epizode okoline agenta. U početku agentu treba duže da riješi problem, ali što agent dulje uči, trajanje epizoda bi trebalo biti kraće. Zbog nasumičnog generiranja kombinacije brave ovaj graf nikad neće biti linearan.



Sl. 4.15 Legenda za vizualni prikaz rezultata treniranja



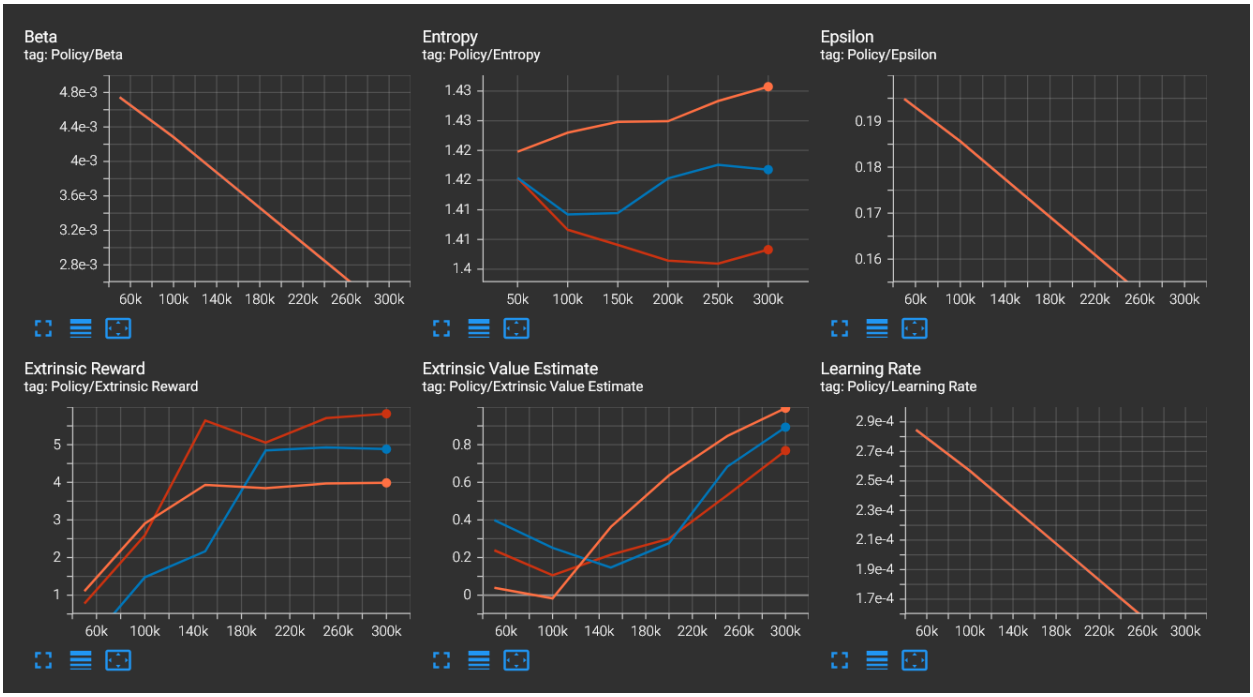
Sl. 4.16.Grafovi *Cumulative Reward i Episode Length*

Policy je kompleksan set pravila koja agentu govore kako se treba ponašati u različitim situacijama za vrijeme treniranja. *Policy Loss* graf (Slika 4.17.) prikazuje koliko se politika promijenila za vrijeme treniranja. Vrijednost bi se trebala smanjiti s uspješim treningom. Budući da ova okruženje sadrži nepredvidljivo ponašanje (slučajno generiranje kombinacije brave) donošenje odluka neće se olakšati s vremenom. *Value Loss* graf (Slika 4.17.) odnosi se na to koliko je predvidljiva

vrijednost svakog stanja. Vrijednost bi se trebala rasti kako agent uči, a zatim se smanjivati kako se nagrada stabilizira.

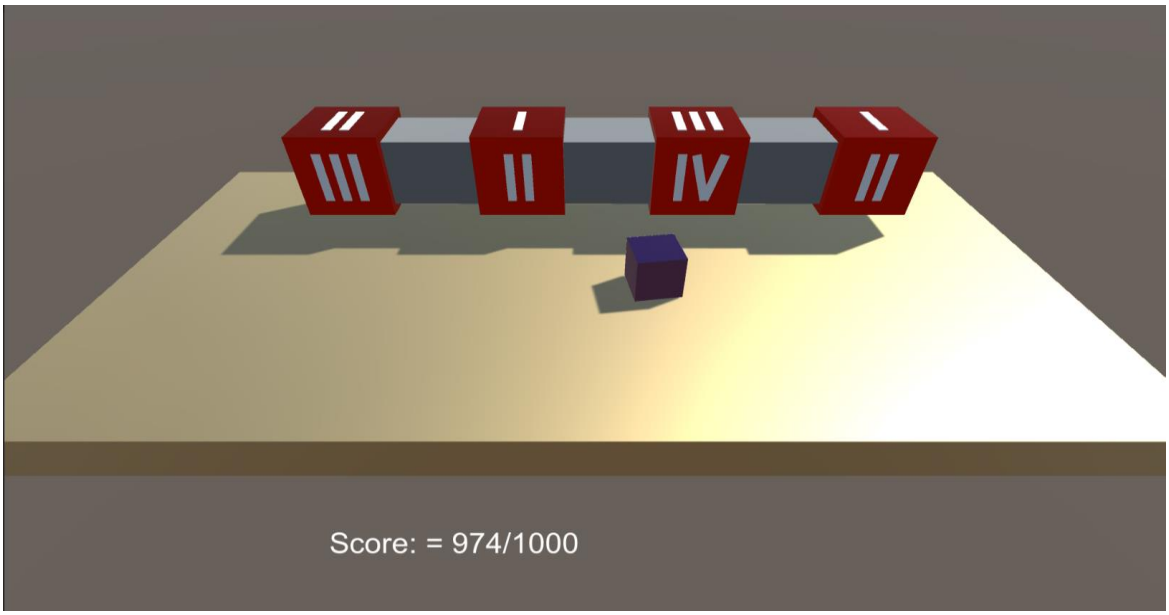
Sl. 4.17. Grafovi *Policy Loss* i *Value Loss*

Beta graf (Slika 4.18.) odnosi se na beta parametar iz configuration.yaml datoteke (Slika 4.14.) tj. opisuje koliko će agent istraživati okolinu. Vrijednost bi se trebala linearno smanjivati zato što kako agent uči ima manje potrebu za istraživanjem. *Entropy* graf (Slika 4.18.) govori o nasumičnnosti odluka agenta za vrijeme treniranja. Vrijednost bi trebala s vremenom postupno padati za vrijeme uspješnih treniranja. *Epsilon* graf (Slika 4.18.) vezan je uz epsilon graf iz *configuration.yaml* datoteke (Slika 4.14.). Vrijednost bi se trebala linearno opadati do pola početne vrijednosti. Graf *Extrinsic Reward* (Slika 4.18.) odgovara prosječnoj kumulativnoj nagradi dobivenoj od okruženja po epizodi. Izgleda isto kao *Cumulative Reward* zato što se koriste samo vanjske nagrade. *Extrinsic Value Estimate* graf (Slika 4.18.) pokazuje srednju vrijednost stanja na koje je agent naišao za vrijeme prošlog perioda. Vrijednost bi trebala postepeno rasti kada agent više ne istražuje toliko okolinu. Graf Learning Rate (Slika 4.18.) odnosi se na veličinu *learning_rate_schedule* parametar iz *configuration.yaml* datoteke (Slika 4.14.) [15]. Treniranje agenta za ovaj projekt trajalo je 300 tisuća koraka. Kombinaciji sa četiri znamenke bilo je potrebno 33 minuta, kombinaciji sa 5 znamenki bilo je potrebno 34 minute i kombinaciji sa 6 pinova bilo je potrebno 35 minuta kako bi se agent utrenirao pronaći točnu kombinaciju. Nakon što se agent utrenirao, izmjereno je prosječno vrijeme potrebno agenti za pronalazak točne kombinacije. Četveroznamenkastoj kombinaciji bilo je prosječno potrebno 14.2 sekundi, peteroznamenkastoj kombinaciji bilo je prosječno potrebno 17.6 sekundi ,a šesteroznamenkastoj kombinaciji bilo je prosječno potrebno 22.8 sekundi. Oko 180 tisuća koraka graf srednjih kumulativnih nagrada agenta počinje stagnirati.



Sl. 4.18.Vizualni prikaz rezultata treniranja

Osim TensorBoard-a rezultati treniranja agent mogu se prikazati stavljanjem modela neuronske mreže agenta u *Behaviour Parameters* (Slika 4.7.) i testiranjem uspješnosti rješavanja problema. U simulaciju je dodan Text objekt pomoću kojeg se može pratiti uspješnost agenta tj. je li agent uspio naći točnu kombinaciju brave ili ne. Na slici 4.19. prikazana je statistika simulacije. Od 1000 iteracija agent je točno riješio problem 974 puta. Postotak uspješnosti agenta je 97.4%. Može se zaključiti kako agent daje zadovoljavajuće rezultate i uspješno rješava problem.



Sl.4.19. Simulacijski prikaz rezultata treniranja

5. ZAKLJUČAK

U ovome završnom radu istražene su mogućnosti Unity programskog okruženja i dodatka za strojno učenje ML-Agents. U praktičnom dijelu rada agent je naučen naći točnu kombinaciju brave. Ovaj projekt pripada jednostavnijim problemima koje agent treba riješiti zato što agent treba paziti na samo par varijabli. Nasumičnost u generiranju kombinacija brava dodana je kako bi činila ovaj završni rad kompleksnijim. Agentu je bilo potrebno 300 tisuća koraka treninga da bi dao zadovoljavajuće rezultate.

Iako je ML-Agents jedan od starijih dodataka i dalje privlači veliku pažnju na sebe i ima nova izdanja. Omogućuje testiranje stvaranje simulacija strojnog učenja koje kasnije možemo implementirati na računala ili strojeve. Isto tako važno je napomenuti da su Unity i ML-Agents besplatni, i to omogućuje svakome tko je zainteresiran za strojno učenje da testira svoje projekte. ML-Agents će se bez sumnje nastaviti poboljšavati i pomoći u brzini procesa razvoja novih tehnologija strojnog učenja.

LITERATURA

[1]M. Bareš ,Strojno učenje u Unityu, Diplomski rad, Osijek: Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija, 2018.

[2]D. Hodak, Napredni mehanizmi percepcije u igri traženja na temelju dubokog učenja, Završni rad, Osijek: Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija, 2020.

[3]Sumrak,Cross-the-road-Unity-ML-Agents,https://github.com/Sumrak-s/Cross-the-road-Unity-ML-Agents, posjećeno 25.6.2022.

[4]H. Caballero, Unity Live Webinar Series - Building an Arcade Racer. Part 5: Machine Learning Agents,https://www.youtube.com/watch?v=8KMESIrCozY&ab_channel=Unity-ANZSEA, posjećeno 25.6.2022.

[5]T.Iseghem,AI-Parking-Unity,https://github.com/VanIseghemThomas/ai-parking-unity, posjećeno 25.6.2022.

[6]Chrispresso,AI-Learns-to-Play-Super-Mario-Bros!,https://chrispresso.io/AI_Learns_To_Play_SMB_Using_GA_And_NN,posjećeno 25.6.2022.

[7]Unity (game engine),https://en.wikipedia.org/wiki/Unity_(game_engine),posjećeno 27.6.2022.

[8]Beginner's Guide to Unity – Understanding Unity Editor and Its Interface, https://circuitstream.com/blog/beginners-guide-to-unity-understanding-unity-editor-and-its-interface/,posjećeno 27.6.2022.

[9]C# Tutorial, https://www.tutorialspoint.com/csharp/index.htm, posjećeno 27.6.2022.

[10]ML-Agents Github, https://github.com/Unity-Technologies/ml-agents,posjećeno 28.6.2022.

[11]ML-Agents-Overview,https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md ,posjećeno 28.6.2022.

[12]Tensorflow, https://www.tensorflow.org/,posjećeno 28.6.2022.

[13]TensorBoard, https://www.tensorflow.org/tensorboard,posjećeno 28.6.2022.

[14]Agents,https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md,posjećeno 24.6.2022.

[15]Using-TensorBoard,https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md ,posjećeno 24.06.2022.

SAŽETAK

Cilj ovoga završnog rada je demonstrirati sve sposobnosti dubinskog i strojnog učenja pomoću alata za izradu virtualnih simulacija. Unity je *game engine* koji se koristi za izradu trodimenzionalnih (3D) i dvodimenzionalnih (2D) igara, interaktivnih simulacija itd. Simulacija je napravljena s dodatkom ML-Agents koji omogućuje primjenu strojnog učenja i obuku agenata za obavljanje određenih zadataka. Simulacija je stvorena kao dio testiranja mogućnosti u kojem se agenta podučavaja korištenjem strojnog učenja uz podršku. Unutar određenog vremena, agent mora pronaći nasumično generiranu kombinaciju brave. Agent je uspješno savladao zadani zadatak.

Ključne riječi: ML-agents, strojno učenje, TensorFlow, podržano učenje, Unity

ABSTRACT

This final project aims to demonstrate all the potentials of deep learning and machine learning by employing tools for building virtual simulations. Three-dimensional (3D) and two-dimensional (2D) games, interactive simulations, and other experiences can all be made using the game engine Unity. The simulation was created with an add-on called ML-Agents, which enables the application of machine learning and the training of agents to carry out particular tasks. A simulation was created as part of the effort in which the agents are instructed using machine learning with support. Within a set amount of time, the agent has to figure out a randomly generated lock combination. The assigned task has been successfully completed by the agent.

Keywords: ML-agents, machine learning, TensorFlow, support learning, Unity

ŽIVOTOPIS

Filip Kramar rođen je 09.08.2000. u Osijeku. Od 2007. do 2015. pohađa OŠ Višnjevac. Nakon toga upisuje III. gimnaziju Osijek u Osijeku. Srednju školu završava 2019. te potom upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek koji trenutno pohađa.