Strojno učenje u Unityu

Bareš, Mateo

Master's thesis / Diplomski rad

2018

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:200:284605

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

Diplomski studij računarstva

Strojno učenje u Unityju

Diplomski rad

Mateo Bareš

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 24.09.2018.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Mateo Bareš
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina	D 886 R, 19.09.2017.
OIB studenta:	72779015464
Mentor:	Doc.dr.sc. Časlav Livada
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva:	Izv. prof. dr. sc. Krešimir Nenadić
Naslov diplomskog rada:	Strojno učenje u Unityu
Znanstvena grana rada:	**Obradba informacija (zn. polje računarstvo)**
Zadatak diplomskog rada:	U radu je potrebno opisati način primjene strojnog učenja u razvojnom okruženju Unity. Potrebno je isto primijeniti u svrhu dolaska do rješenja igre. Oko stila igre dogovorit ćemo se na konzultacijama.
Prijedlog ocjene pismenog dijela ispita (diplomskog	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 2 razina
Datum prijedloga ocjene mentora:	24.09.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 28.09.2018.

Ime i prezime studenta:	Mateo Bareš
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D 886 R, 19.09.2017.
Ephorus podudaranje [%]:	7

Ovom izjavom izjavljujem da je rad pod nazivom**: Strojno učenje u Unityu**

izrađen pod vodstvom mentora Doc.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

Strojno učenje je grana umjetne inteligencije koja se bavi algoritmima pomoću kojih računala „uče“ obavljati određene zadatke na dostupnim podacima bez eksplicitnog programiranja. U zadnjih nekoliko godina strojno učenje doživjelo je nagli razvoj i raširilo upotrebu u specijaliziranim sustavima, ali i u tehnologijama i uređajima kojima se svakodnevno koristimo. Međutim testiranje strojnog učenja za situacije kao što su specijalizirani roboti koji obavljaju kompleksne zadatke koje je teško ili gotovo nemoguće programirati može biti dugotrajno i skupo. U ovom diplomskom radu istražiti će se mogućnosti novog alata za strojno učenje u Unity *game engineu* koji omogućuje izradu virtualne simulacije sa realnom fizikom. Korištenjem navedenog alata napraviti će se projekt kojim će se prikazati njegove mogućnosti i primjena. Unity Machine Learning Agents je SDK (*Software Development Kit*) koji će se koristiti za izradu projektnog zadataka. Projektni zadatak će prikazati scenarij u kojem agent mora doći na određeno mjesto prolazeći nasumično generiranu cestu. Na cesti će biti generirane crvene kugle koje će predstavljati prepreke koje agent mora zaobići. Agent će koristiti učenje s potporom.

U nastavku rada su ukratko opisane tehnologije koje se koriste sa naglaskom na ML-Agents. Nakon toga su objašnjeni elementi strojnog učenja bitni za izradu projekta. U sljedećem poglavlju je opisana izrada projekta, a na kraju su prikazani i komentirani rezultati rada.

1.1. Zadatak završnog rada

U ovom diplomskom radu potrebno je istražiti, objasniti i pokazati primjenu strojnog učenja u Unity *game engineu* korištenjem nove tehnologije Unity Machine Learning Agents. Također u radu je potrebno dati pozadinu strojnog učenja koja je bitna za razumijevanje projektnog zadatka.

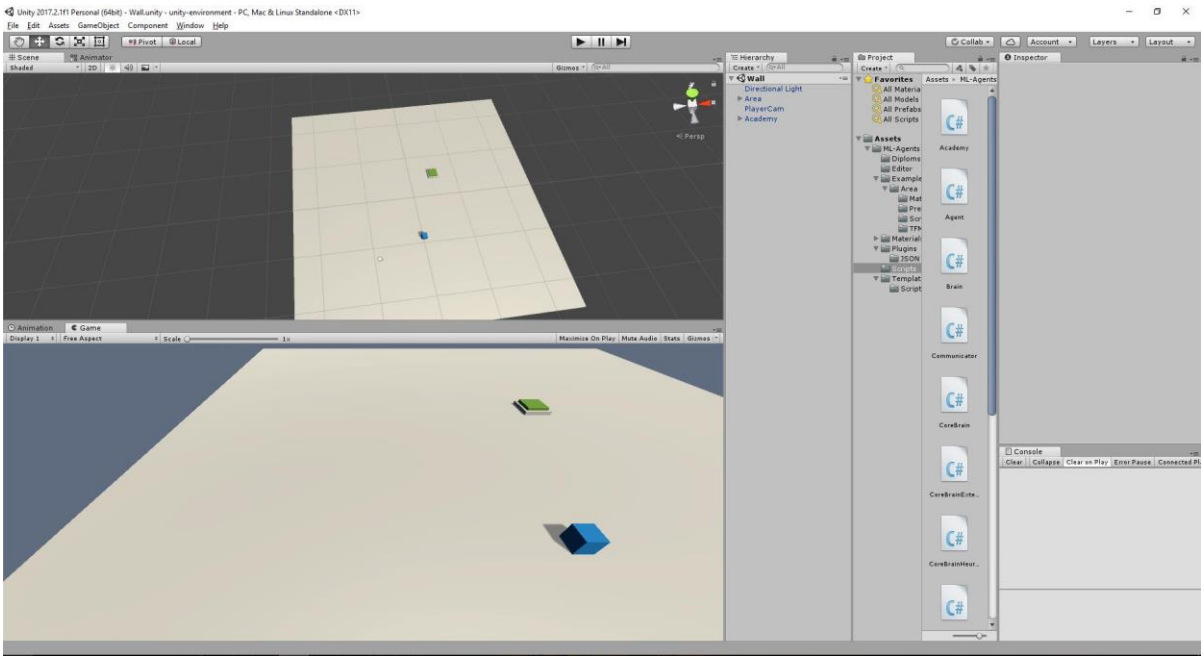
2. KORIŠTENE TEHNOLOGIJE I ALATI

2.1 Unity

Unity je *game engine* koji se koristi za razvoj 2D i 3D videoigara, simulacija i interaktivnog sadržaja. Unity podržava i razvoj aplikacija virtualne i proširene stvarnosti i treniranje agenata pomoću strojnog učenja što je i obrađeno u ovom radu. Unity je razvila tvrtka Unity Technologies 2005. godine. Sadržaji izrađeni i pokretani u Unityju mogu se izvoditi na više od 25 platformi.

Unity je razvijen u C++ programskom jeziku, a API kojim se programeri koriste za razvijanje sadržaja i korištenje raznih mogućnosti Unityja je pisan u C# programskom jeziku. C# programski jezik se koristi i za pisanje skripti kojima se određuje djelovanje agenata i okoline u kreiranom sadržaju. Za pokretanje realistične fizike visokih performansi Unity podržava i koristi Box2D i NVIDIA PhysX fizičke pokretače. [1]

Unity uređivač je sučelje Unityja u kojemu se izrađuje sadržaj. Organiziran je tako da uvijek prikazuje samo jednu scenu koja može predstavljati razinu neke videoigre, izbornik, okolinu simulacije i slično. Unutar scene se nalaze objekti kojima se manipulira raznim komponentama od kojih su najbitnije C# skripte koje određuju složenija djelovanja tih objekata. Uređivač podržava razne slikovne, audio, video i tekstualne formate. [2]



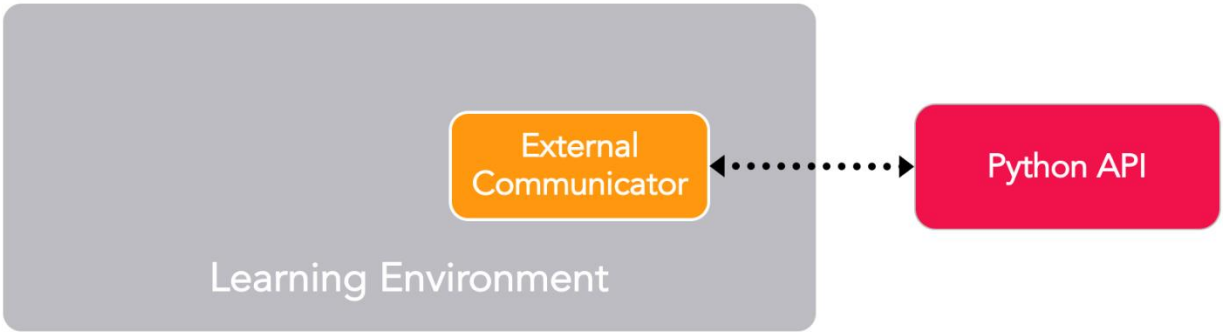
Sl. 2.1. Izgled sučelja Unity uređivača

Slika 2.1. prikazuje sučelje Unity uređivača. Na prikazu se vidi scena na kojoj se trenutno radi i prikaz scene unutar igre ispod nje, a sa desne strane je hijerarhija objekata u sceni i projektno stablo. [3]

2.2 Unity Machine Learning Agents

Unity ML-Agents je *software development kit* (SDK) koji omogućuje strojno učenje inteligentnih agenata koji mogu biti trenirani dubokim učenjem s potporom, evolucijskim strategijama ili drugim metodama strojnog učenja pomoću jednostavnog korištenja Python API-ja. Također ML-Agents pruža implementacije najsuvremenijih algoritama za jednostavno treniranje inteligentnih agenata. Ti agenti mogu biti korišteni u više svrha i obostrano su korisni za razvojne programere igara i istraživače umjetne inteligencije. ML-Agents pruža središnju platformu na kojoj se napredak u umjetnoj inteligenciji može procijeniti u Unityjevom bogatom okruženju i zatim postati dostupan široj istraživačkoj zajednici.

ML-Agents se sastoji od tri glavne komponente prikazane na slici 2.2.



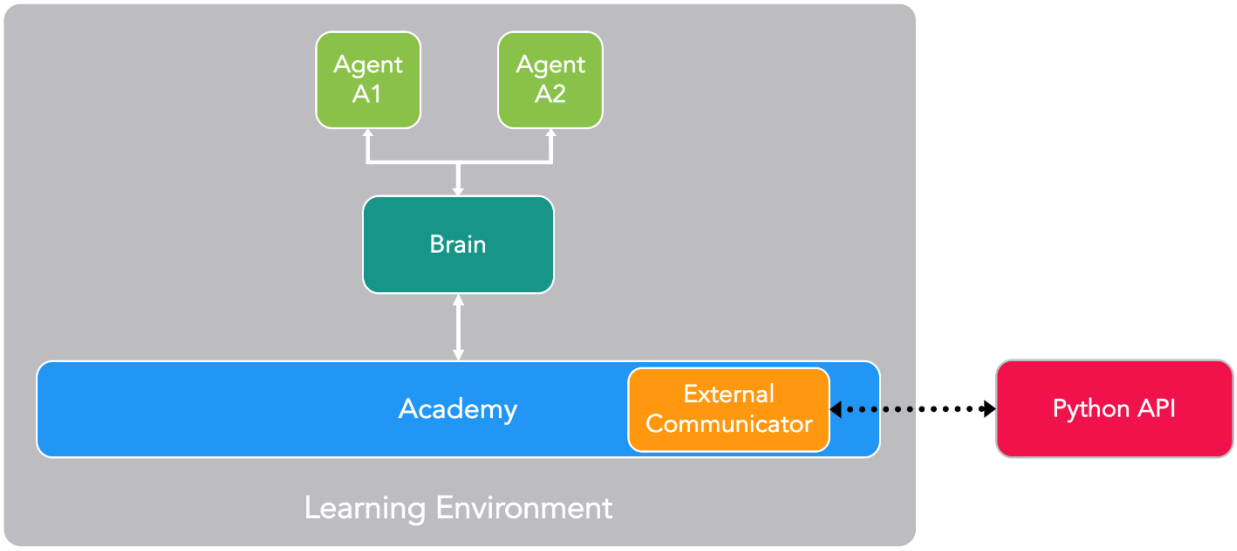
Sl. 2.2. Pojednostavljeni blok dijagram ML-Agentsa [4]

- **Okolina učenja** – sadrži Unity scenu i sve sudionike.
- **Python API** – sadrži sve algoritme strojnog učenja koji se koriste za trening. Za razliku od okoline učenja, Python API nije dio Unityja, nego postoji izvan i komunicira s Unityjem preko vanjskog komunikatora.
- **Vanjski komunikator** – povezuje okolinu učenja s Python API-jem. Postoji unutar okruženja učenja.

Okolina učenja sadrži dodatne tri komponente koje pomažu organizirati Unity scenu:

- **Agenti** – koji su dodani na Unity GameObject (bilo koji objekt unutar scene) i rukovodi generaciju njegovih opažanja, poduzimanja akcija i dodjeljivanje nagrade (pozitivne / negativne) kada je to prikladno. Svaki agent je povezan s točno jednim mozgom.
- **Mozgovi** – koji obuhvaćaju logiku odlučivanja agenta. Mozak je ono što sadrži pravila za svakog agenta i određuje koje akcije agent treba poduzeti u svakom slučaju. Konkretnije, to je komponenta koja prima opažanja i nagrade od agenta i vraća akciju.
- **Akademija** – koja orkestrira opažanje i proces donošenja odluka. U okviru Akademije može se odrediti nekoliko parametara na razini okoline, kao što su kvaliteta prikazivanja i brzina kojom se okoliš provodi. Vanjski komunikator živi unutar akademije.

Svaka okolina učenja će uvijek imati jednu globalnu akademiju i jednog agenta za svakog lika u sceni. Dok svaki agent mora biti povezan s mozgom, moguće je da agenti koji imaju slična opažanja i akcije mogu biti povezani na isti mozak. Na slici 2.3. prikazan je blok dijagram scene u kojoj postoje dva agenta koji su povezani na isti mozak jer njihov prostor opažanja i akcije su slične. To ne znači da će oba agenta imati ista opažanja i iste akcije. Mozak definira prostor svih mogućih opažanja i akcija, dok agenti povezani na njega mogu imati svaki svoja, jedinstvena opažanja i akcije.



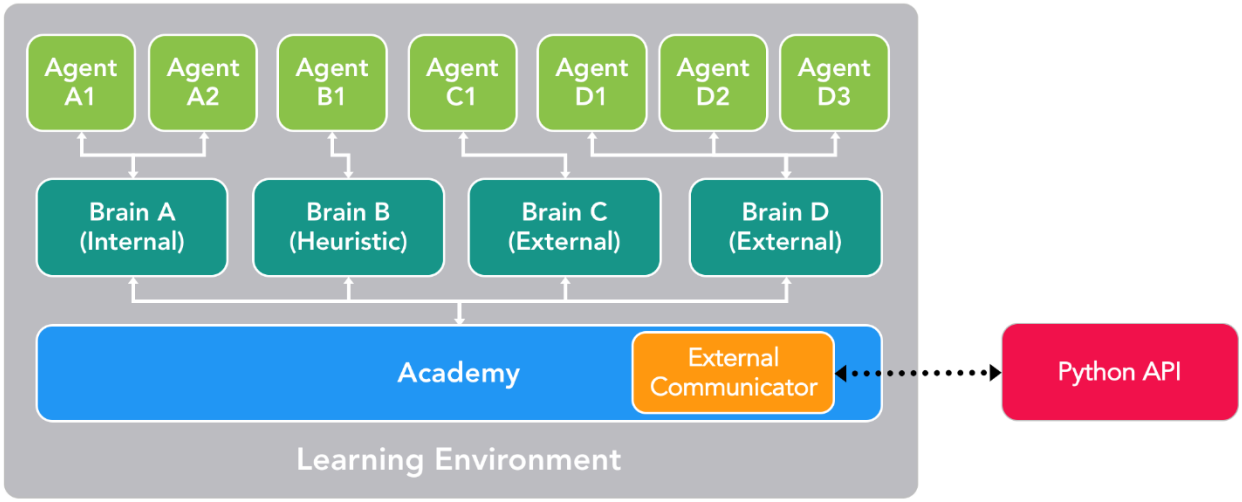
Sl. 2.3. Blok dijagram primjera scene u ML-Agentsu [4]

Postoje četiri različita tipa mozga koji omogućuju širok opseg scenarija treninga i zaključivanja:

- **Eksterni** – gdje se odluke donose pomoću Python API-ja. Opažanja i nagrade koje je prikupio mozak prosljeđuju se Python API-ju putem vanjskog komunikatora. Python API zatim vraća odgovarajuću radnju koju agent mora poduzeti.
- **Interni** – gdje se odluke donose pomoću ugrađenog TensorFlow modela. Ugrađeni TensorFlow model predstavlja naučena pravila i mozak izravno koristi model da bi odredio akciju za svakog agenta.
- **Igračev** – gdje se odluke donose koristeći pravi ulaz sa tipkovnice i igraćeg kontrolera. Ljudski igrač kontrolira agenta i opažanja i nagrade koje je prikupio mozak ne koriste se za kontrolu agenta.
- **Heuristički** – gdje se odluke donose pomoću ukodiranog ponašanja. To je slično tome kako je većina ponašanja likova u igrama trenutačno definirana i može biti korisna za ispravljanje pogrešaka ili uspoređivanje načina na koji agent s ukodiranim pravilima uspoređuje s agentom čije je ponašanje naučeno.

Kako je opisano čini se da vanjski komunikator i Python API iskorištava samo eksterni mozak, ali moguće je konfigurirati interne, igračeve i heurističke mozgove da također šalju opažanja, nagrade i akcije Python API-ju preko vanjskog komunikatora (značajka koja se naziva emitiranje).

Na slici 2.4. može se vidjeti primjer kako bi mogla biti konfigurirana scena sa više agenata sa različitim mozgovima.



Sl. 2.4. Blok dijagram primjera scene u ML-Agentsu sa više agenata sa različitim mozgovima [4]

S obzirom na fleksibilnost ML-agenata, postoji nekoliko načina na koje se trening i zaključivanje mogu nastaviti.

Treniranje agenta u ML-Agents okolini, koristeći učenje s potporom, zahtijeva tri entiteta u svakom trenutku simulacije:

- **Opažanja** – što agent percipira o okolini, mogu biti numerička i/ili vizualna. Numerička opažanja mjere atribute okoline s gledišta agenta. Opažanja mogu biti diskretna ili kontinuirana, ovisno o složenosti simulacije i agenta. Za najinteresantnija okruženja, agent će zahtijevati nekoliko neprekidnih numeričkih opažanja, a za jednostavna okruženja s malim brojem jedinstvenih konfiguracija, dovoljno će biti diskretna opažanja. Vizualna opažanja, s druge strane, slike su generirane iz kamere pričvršćene za agenta i predstavljaju ono što agent u tom trenutku vidi. Ne treba miješati opažanje agenta sa stanjem okoline. Stanje okoline predstavlja informacije o cijeloj sceni. Opažanja agenata, međutim, sadrže samo informacije kojih je agent svjestan i tipično je podskup stanja okoline.
- **Akcije** – koje akcije agent može poduzeti. Slično kao i opažanja, akcije mogu biti kontinuirane ili diskretne ovisno o složenosti okoline i agenta.
- **Signali nagrade** – skalarna vrijednost koja ukazuje na to koliko dobro agent djeluje. Signal nagrade ne mora biti pružan svakog trenutka, nego samo onda kada agent izvrši akciju koja je dobra ili loša. Nagradni signal je način na koji se ciljevi zadatka komuniciraju agentu pa ih treba postaviti na način da maksimiziranje nagrade generira željeno optimalno ponašanje.

Nakon definiranja navedena tri entiteta (građevnih blokova zadataka učenja s potporom), treniranje agentovog ponašanja može započeti. To se postiže simulacijom okoline za mnoge pokuse u kojima agent s vremenom uči što je optimalno djelovanje koje se poduzima za svako zapažanje tako da maksimizira svoju nagradu. U terminima učenja s potporom, ponašanje koje se nauči naziva se pravilom, što je u osnovi (optimalno) mapiranje zapažanja u akciju.

2.2.1 Tipovi učenja

Ugrađeni trening i zaključivanje

ML-Agents dolazi sa nekoliko implementacija najsuvremenijih algoritama za treniranje inteligentnih agenata. U ovom načinu, tip mozga je postavljen na eksterni tijekom treninga i interni tijekom zaključka. Tijekom treninga svi agenti na sceni šalju svoja opažanja Python API-ju putem vanjskog komunikatora (to je djelovanje eksternih mozgova). Python API obrađuje ta opažanja i šalje natrag aktivnosti za svakog agenta. Tijekom treninga ove su radnje uglavnom istraživačke kako bi se Python API mogao naučiti najboljim pravilima za svakog agenta. Kada završi obuka, moguće je izvesti naučenu politiku svakog agenta. Budući da su sve naše implementacije temeljene na TensorFlowu, naučena pravila su samo TensorFlow datoteka modela. Tada tijekom faze zaključivanja prebacimo tip mozga u interni i uključimo TensorFlow model generiran iz faze treniranja. Sada, tijekom faze zaključivanja, agenti i dalje nastavljaju generirati svoja zapažanja, ali umjesto ih šalju Python API-ju, oni će se unijeti u svoj (interni, ugrađeni) model kako bi generirali optimalnu akciju za svakog agenta koji bi trebao biti u svakoj točki na vrijeme.

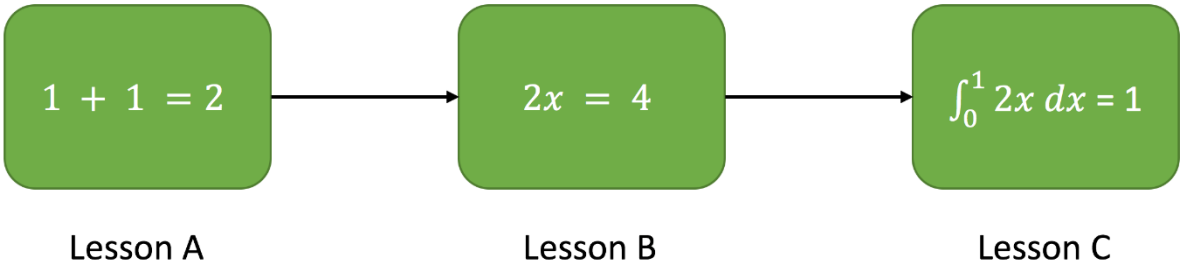
Prilagođeni trening i zaključivanje

U prethodnom načinu, eksterni tip mozga je korišten za obuku da generira TensorFlow model koji interni mozak može razumjeti i koristiti. Međutim, svaki korisnik ML-Agentsa može iskoristiti svoje algoritme za obuku i zaključke. U tom će slučaju tip mozga biti postavljen na vanjski kako za obuku tako i za faze izvođenja i ponašanje svih agenata na sceni bit će kontrolirano unutar Pythona.

Učenje s kurikulumom

Ovaj način je proširenje ugrađenog treninga i zaključivanja, a posebno je koristan pri treniranju složenih ponašanja za složene okoline. Učenje s kurikulumom je način osposobljavanja modela strojnog učenja gdje se teži aspekti problema postupno uvode na takav način da je model uvijek

optimalno izazvan. Ova ideja je već dugo prisutna i to je način na koji ljudi obično uče. Na primjer u osnovnoj školi se prvo uči aritmetika pa tek nakon toga algebra. Vještine i znanja naučena u ranijim temama pružaju temelj za kasnije lekcije. Isti princip može se primijeniti na strojno učenje, gdje trening na jednostavnijim zadacima može osigurati temelj za teže zadatke u budućnosti. Jednostavan primjer kurikuluma učenja matematike je prikazan na slici 2.5.



Sl. 2.5. Primjer kurikuluma matematike, lekcije idu od lakše prema težoj [4]

Kod učenja s potporom, signal učenja je povremeno zaprimljena nagrada tijekom obuke. Polazna točka kada se obučava agenta za obavljanje ovog zadatka bit će slučajna pravila. Ta početna pravila će agenta kretati u krug, a vrlo rijetko ili uopće neće postići nagradu za složene okoline. Tako pojednostavljenjem okoline na početku obuke dopuštamo agentu da brzo ažurira slučajna pravila na neka smislenija koja se sukcesivno poboljšavaju kako okolina postupno povećava složenost. ML-Agents podržavaju postavljanje prilagođenih parametara okoline unutar akademije. To omogućava dinamičko prilagođavanje elemenata okoline koji se odnose na poteškoće ili složenost na temelju napretka treninga.

Učenje imitacijom

Često je intuitivnije jednostavno pokazati ponašanje koje želimo da agent obavlja, a ne pokušavati ga naučiti metodom pokušaja-pogreške. Na primjer, umjesto treninga agenta postavljanjem svoje funkcije nagrađivanja, ovaj način omogućuje stvaranje pravih primjera od kontrolora scene o tome kako se agent treba ponašati. Točnije, u ovom načinu rada tip mozga tijekom treninga postavljen je na igračev i sve aktivnosti koje se provode s kontrolorom (uz opažanje agenata) bit će zabilježene i poslane Python API-ju. Algoritam učenja imitacijom će zatim koristiti ove parove opažanja i akcija od ljudskog igrača kako bi naučio pravila.

2.2.2 Dodatne značajke

- **Donošenje odluka na zahtjev** – moguće je da agenti zatraže odluku samo kada je to potrebno, a ne u svakom koraku okoline. To omogućuje treniranje igara na potez, igara gdje agenti moraju reagirati na događaje i igara gdje agenti mogu poduzimati akcije sa varijabilnim trajanjem.
- **Agenti s poboljšanom memorijom** – u nekim scenarijima, agenti moraju naučiti pamtiti prošlost kako bi se donijela najbolja odluka. Kada agent ima samo djelomično zapažanje okoline, praćenje prošlih opažanja može pomoći agentu da uči. Zbog toga postoji implementacija duge kratkotrajne memorije (LSTM) u trenerima koji omogućuju agentu da sprema memoriju za buduće korake.
- **Praćenje agentovog postupka odlučivanja** – budući da je komunikaciju u ML-Agentsu dvosmjerna, postoji agent Monitor klasa u Unityju koja prikazuje aspekte treniranog agenta, poput percepcije agenta o tome koliko dobro radi (tzv. procjena vrijednosti) unutar same Unity okoline. Iskorištavajući Unity kao alat za vizualizaciju i pružajući te rezultate u realnom vremenu, istraživači i razvojni programeri lakše će ispraviti ponašanje agenta.
- **Složena vizualna opažanja** - za razliku od drugih platformi, gdje opažanje agenata može biti ograničeno na samo jedan vektor ili sliku, ML-Agents omogućuju upotrebu više kamera za opažanje po agentu. To omogućuje agentima da nauče integrirati informacije iz više vizualnih izvora. To može biti korisno u nekoliko scenarija poput osposobljavanja samovoznog automobila koji zahtijeva više kamera s različitim vidovima ili navigacijsko sredstvo koje bi moglo trebati integrirati vizualne snimke iz zraka i iz prvog lica.
- **Emitiranje** – vanjski mozak šalje opažanja za sve svoje agente na Python API prema zadanim postavkama. Ovo je korisno za trening ili zaključivanje. Emitiranje je značajka koja se može omogućiti za ostala tri načina rada (igračev, interni, heuristički), gdje se i promatranja i akcije agenata šalju i Python API-u (usprkos činjenici da agentom nije upravljao Python API). Ovu značajku iskorištava učenje imitacijom, gdje se opažanja i akcije za igračev tip mozga koriste za učenje pravila agenata kroz demonstraciju. Međutim, to bi također moglo biti korisno za heuristički i unutarnji mozak, osobito kada se otklanjaju pogreške u ponašanju agenata.
- **Docker postavljanje (eksperimentalno)** – postavljanje ML-Agentsa bez instalacije Pythona i TensorFlowa
- **Trening u oblaku koristeći Amazon Web Services**
- **Trening u oblaku koristeći Microsoft Azure** [4]

2.3 C#

C# je objektno orijentirani jezik koji programerima omogućuje izradu raznih sigurnih i robusnih aplikacija koje se pokreću na .NET Frameworku. Koristi se za stvaranje Windows klijentskih aplikacija, XML web usluga, distribuiranih komponenti, aplikacija klijent-poslužitelj, aplikacija baze podataka i još mnogo toga. Visual C # nudi napredni program za uređivanje koda, praktične alate za dizajn korisničkog sučelja, integrirani *debugger* i mnoge druge alate kako bi se olakšao razvoj aplikacija na temelju C # jezika i .NET Frameworka. Sintaksa jezika ima mnoge sličnosti sa C, C++ i Java programskim jezicima. C# sintaksa pojednostavljuje kompleksnost C++ i pruža moćne značajke kao što su *nullable* vrijednosni tipovi, enumeracije, delegiranje, lambda ekspresije i direktni pristup memoriji. C# podržava generičke metode i tipove, koje pružaju povećanu sigurnost tipova i performanse. Podržava i iteratore koji omogućavaju objektima kolekcijskih klasa da definiraju prilagođena iteracijska ponašanja jednostavna za korištenje klijentskim kodom.

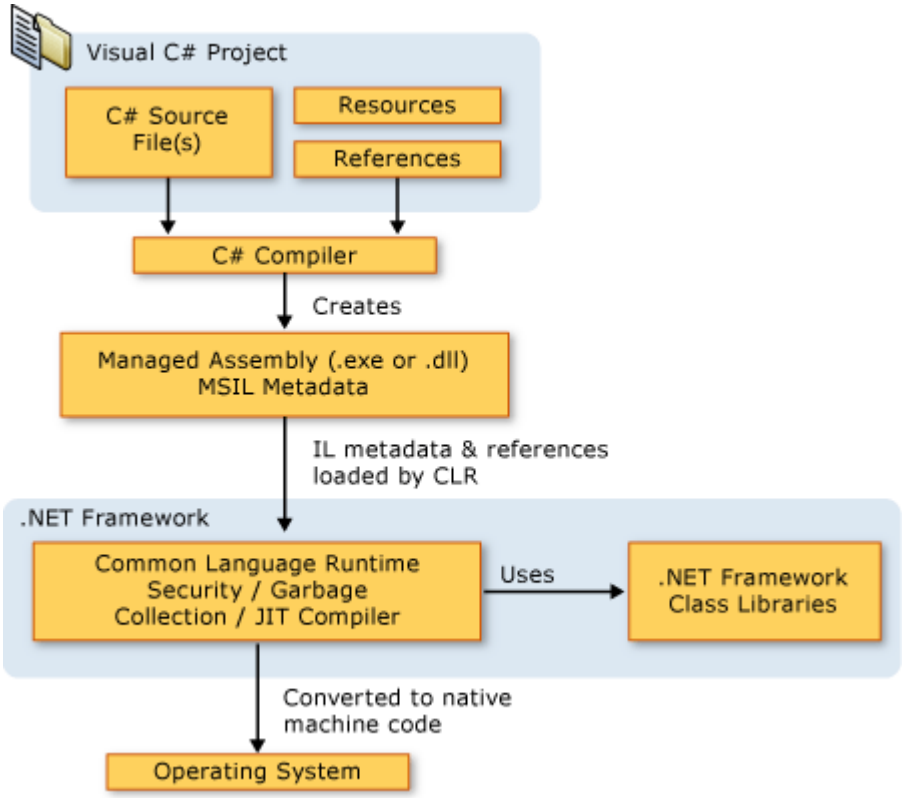
C# podržava koncepte učahurivanja ili enkapsulacije, nasljeđivanja i višeobličja ili polimorfizma. Sve varijable i metode su učahurene unutar definicije klasa. Klasa može direktno nasljeđivati samo jednu roditeljsku klasu, ali može implementirati bilo koji broj sučelja.

C# programi se izvode na .NET Frameworku, integralnoj komponenti Windows sustava koja uključuje virtualni sustav izvršenja koji se naziva *common language runtime* (CLR) i jedinstveni skup biblioteka klasa. CLE je Microsoftova komercijalna implementacija *common language infrastucturea* (CLI), međunarodnog standarda koji je osnova za stvaranje okruženja za izvršenje i razvoj u kojima jezici i biblioteke neprimjetno rade zajedno.

Izvorni kod koji je napisan u C # sastavljen je u *intermediate langauge* (IL) koji je u skladu s CLI specifikacijom. IL kod i resursi, kao što su bitmape i nizovi, pohranjeni su na disku u izvršnoj datoteci zvanoj *assembly*, obično s ekstenzijom .exe ili .dll. *Assembly* sadrži manifest koji pruža informacije o vrstama, verziji, kulturi i sigurnosnim zahtjevima *assemblyja*.

Kada se izvršava C # program, *assembly* je učitan u CLR, koji može poduzeti različite radnje na temelju podataka iz manifesta. Zatim, ako se zadovolje sigurnosni zahtjevi, CLR izvršava *just in time* (JIT) kompilaciju za pretvaranje IL kodova u izvorne strojne instrukcije. CLR također pruža i druge usluge vezane uz automatsko sakupljanje smeća, upravljanje iznimkama i upravljanje resursima. Kod koji izvršava CLR ponekad se naziva "upravljanim kodom", za razliku od "neupravljanog koda" koji se sastavlja na strojni jezik koji cilja određeni sustav. Dijagram na slici

2.2 prikazuje *compile-time* i *run-time* veze C# izvornih datoteka, .NET Framework biblioteka klasa, *assemblyja* i CLR-a. [2]



Sl. 2.6. .NET arhitektura [5]

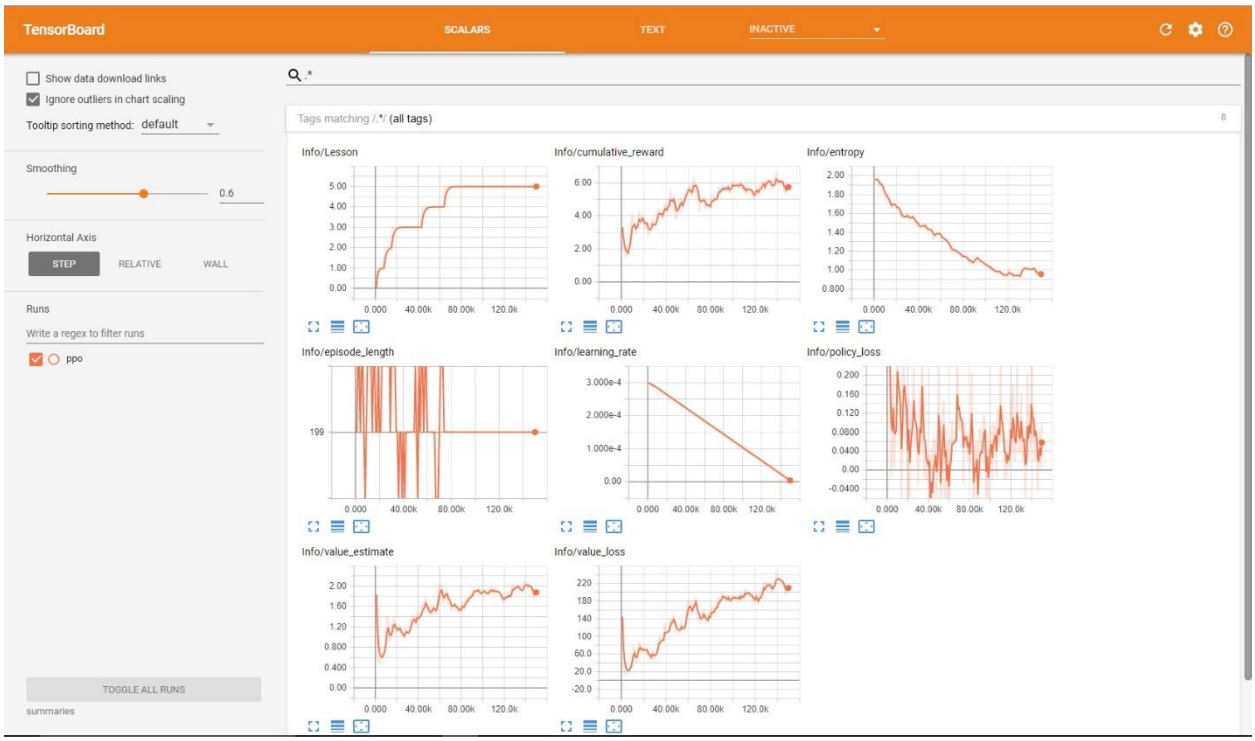
Jezična interoperabilnost ključna je značajka .NET okvira. S obzirom da je IL kod koji proizvodi C# prevodilac u skladu s *Common Product Specification* (CTS), IL kod generiran iz C # može komunicirati s kodom generiranim iz .NET verzija Visual Basic, Visual C ++ ili bilo kojeg od više od 20 drugih CTS-kompatibilnih jezika. Jedna cjelina može sadržavati više modula napisanih na različitim .NET jezicima, a tipovi se mogu međusobno referencirati kao da su pisani na istom jeziku.

Osim *run time* usluga, .NET Framework također uključuje veliku biblioteku od preko 4000 klasa organiziranih u *namespaceove* koji pružaju široku paletu korisnih funkcija za sve, od unosa i izlaza datoteke do manipulacije nizom do XML parsiranja, do kontrola Windows Forms sustava, Tipična aplikacija C# opsežno koristi .NET Framework biblioteku klasa za rukovanje uobičajenih "*plumbing*" poslova. [5]

2.4 Tensorflow

TensorFlow je open-source biblioteka korištena za numeričko računanje visokih performansi. Razvili su je istraživači i inženjeri iz Google Brain tima u Googleovoj AI organizaciji 2015. godine. Biblioteka dolazi sa snažnom podrškom za strojno učenje. TensorFlow se može izvoditi na procesorima, grafičkim procesorima i specifičnim *tensor processing unitima* (TPU) koji su aplikacijski specifični integrirani krugovi proizvedeni od strane Googlea za strojno učenje neuronskim mrežama. Unutar ML-Agenta, kada se trenira ponašanje agenta, izlaz je datoteka TensorFlow modela koja se zatim može ugraditi u interni mozak.

TensorBoard je vizualizacijski alat unutar TensorFlowa koji omogućuje vizualizaciju određenih atributa agenta (npr. nagradu) tijekom treninga. To može biti korisno i u građenju intuicije za različite atribute modela i u postavljanju optimalnih vrijednosti u Unity okolini. Na slici 2.7. prikazano je sučelje TensorBoarda na kojem se mogu vidjeti grafovi koji prikazuje razne parametre učenja kroz vrijeme. [6]



Sl. 2.7. Web sučelje TensorBoarda

Jedan od nedostataka TensorFlowa je da ne pruža izvorni C# API, a skripte za interni mozak u Unityju su pisane u C#. Zbog toga se koristi TensorFlowSharp bilioteka koja pruža .NET vezanje TensorFlowu. Kada je izgrađena Unity okolina koja sadrži interni mozak, zaključivanje se izvodi putem TensorFlowSharpa.

3. STROJNO UČENJE

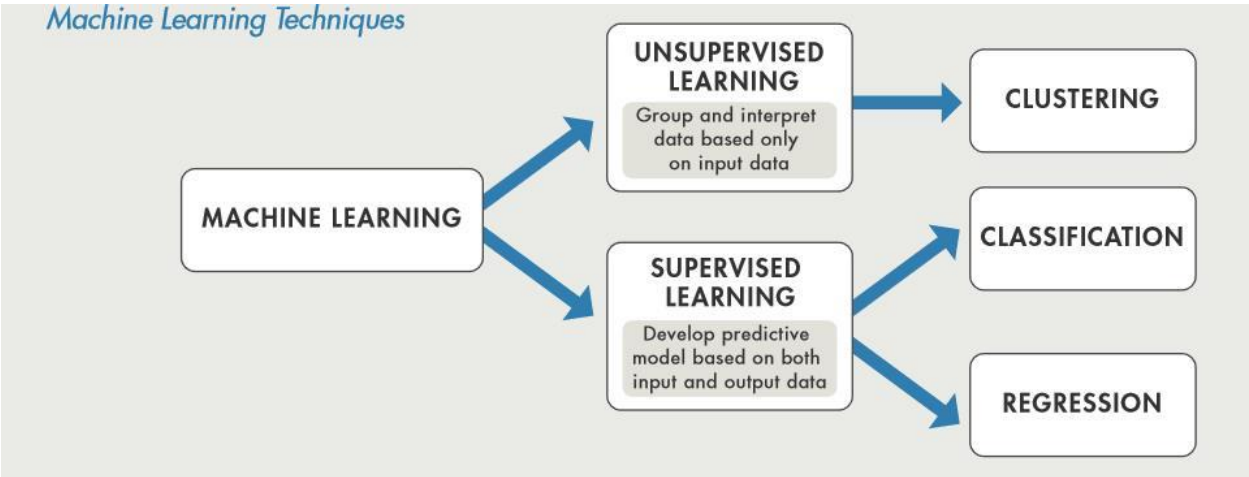
Strojno učenje odnosi se na programiranje računala da mogu „učiti“ iz unosa koji su im dostupni. A sam pojam učenja odnosi se na proces pretvaranja iskustva u stručnost ili znanje. Uneseni podaci u algoritmu učenja su trening podaci koji predstavljaju iskustvo, a izlaz algoritma je neko znanje, koje uglavnom ima formu novog računalnog programa koji može obaviti neki zadatak.

Strojno učenje koristi se za izradu programa koji su previše kompleksni za programiranje i programa koji trebaju biti adaptivni.

Kompleksni zadaci za programiranje uključuju zadatke koje ljudi obavljaju gotovo svakodnevno, ali ne mogu ih precizno opisati da bi definirali program i zadatke koji su izvan granica ljudskih sposobnosti. Na primjer vožnja automobila, raspoznavanje govora i slika su zadaci koje ljudi svakodnevno obavljaju, ali način na koji se oni obavljaju nije dovoljno razrađen te se puno bolji rezultati dobivaju programima koji uče iz iskustva kada su izloženi dovoljnom broju primjera. S druge strane analiza i prikupljanje znanja iz velikog broja podataka kao što su astronomski ili medicinski podaci je previše kompleksna za čovjeka, dok za računala sa velikom, gotovo beskonačnom memorijom i sve većom brzinom, otkrivanje značajnih uzoraka otvara nove horizonte.

Većina programa su ograničeni u adaptivnosti i ne mijenjaju se previše od trenutka kada su instalirani, no djelovanje programa koji imaju sposobnost učenja ovisi o ulaznim podacima te su takvi programi sami po sebi adaptivni. [7]

Strojno učenje može se podijeliti na nadzirano i nenadzirano učenje i na učenje s potporom. Nadzirano učenje uzima poznati set ulaznih podataka i odziva na te podatke i trenira model da generira razumna predviđanja na nove podatke.



Sl. 3.1. Podjela strojnog učenja [8]

3.1. Nadzirano učenje

Nadzirano učenje trenira model na poznatim ulaznim i izlaznim podacima, što znači da program može vršiti predviđanja izlaznih podataka temeljena na prošlim rezultatima. Nadzirano učenje koristi klasifikacijske i regresijske tehnike za razvijanje predviđajućih modela. Klasifikacijske tehnike predviđaju diskretne odzive, odnosno one kategoriziraju ulazne podatke. Regresijske tehnike predviđaju kontinuirane odzive, na primjer temperaturu okoline.

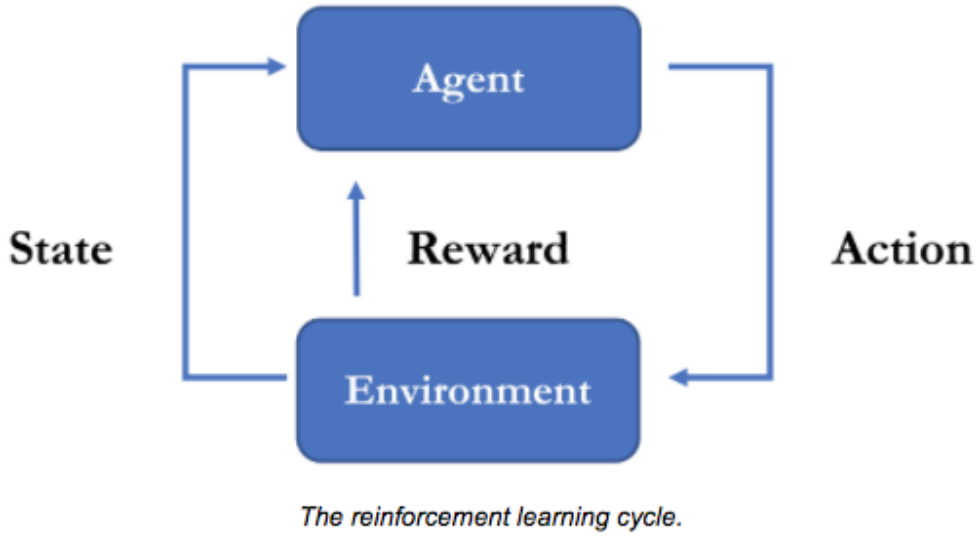
3.2. Nenadzirano učenje

Nenadzirano učenje grupira i pronalazi skrivene uzorke i strukture znajući samo ulazne podatke. Koristi se za skiciranje zaključaka iz skupova podataka koji se sastoje od ulaza bez poznatih odziva. Grupiranje je najčešća nenadzirana tehnika, koristi se za nalaženje skrivenih uzoraka ili grupa u podacima. [8]

3.3. Strojno učenje s potporom

Strojno učenje s potporom može se promatrati kao oblik učenja za sekvencijalno odlučivanje koje je obično povezano s kontroliranjem robota. To je tehnika umjetne inteligencije koja trenira agenta

da obavlja zadatke nagrađivanjem poželjnog ponašanja. Cilj ovog tipa učenja je naučiti pravila, koja su u suštini pridruživanje određene akcije određenim opažanjima. Tijekom učenja agent istražuje okolinu, promatra stanje stvari oko sebe i na temelju tih opažanja poduzima akciju. Opažanje je ono što agent može izmjeriti u svojoj okolini, a akcija u svom najnižem obliku je promjena konfiguracije agenta. Zadnja komponenta učenja s potporom je signal nagrade. Kada se agent trenira, pružaju mu se nagrade koje ukazuju koliko on dobro radi na dovršavanju zadatka. Agent na početku ne zna cilj njegovog treninga, ali ga uči jer dobiva veliku pozitivnu nagradu kada djeluje akcijom koja pridonosi obavljanju zadatka. Činjenica da su nagrade rijetke (nisu pružane u svakom koraku) temeljna je karakteristika ovog tipa učenja i upravo zato učenje dobrih pravila može biti teško i dugotrajno za složene okoline. Proces učenja s potporom prikazan je na slici 3.2. [9]



Sl. 3.2. Ciklus učenja s potporom [9]

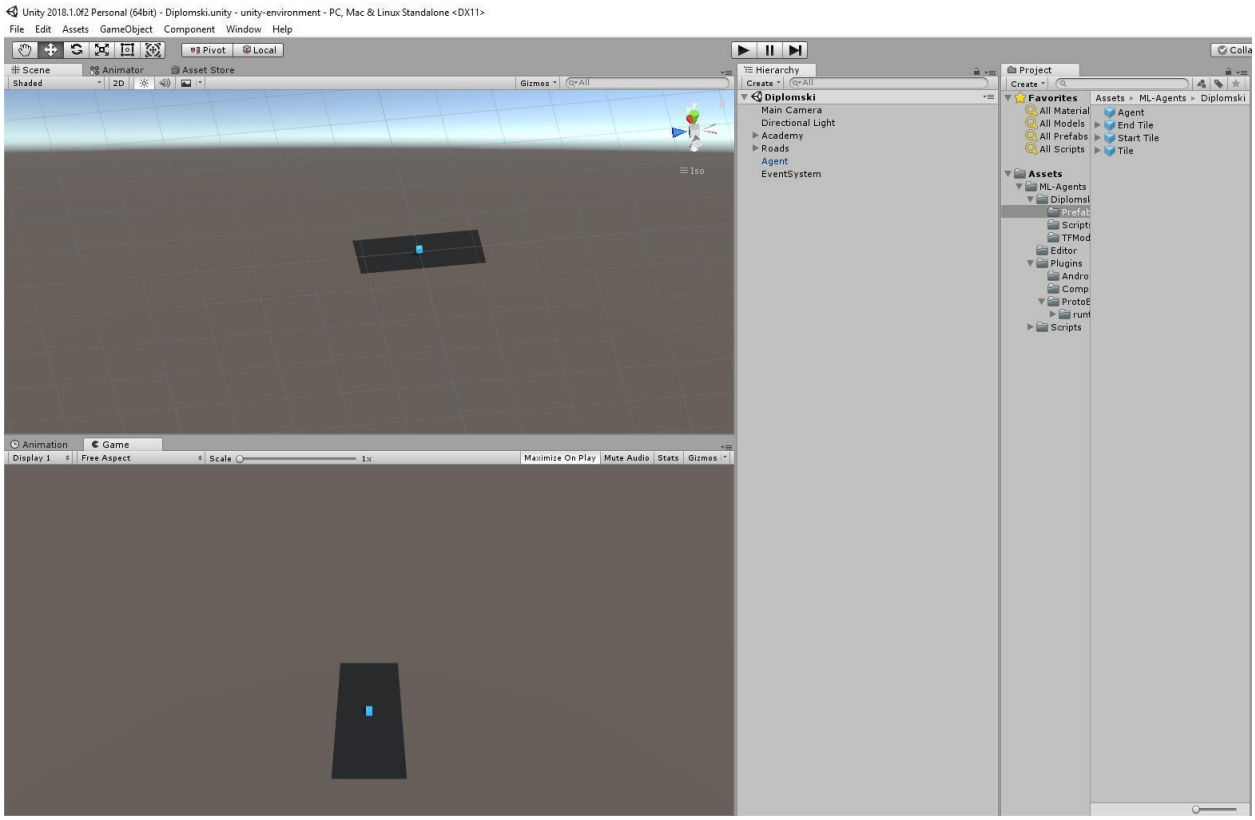
4. IZRADA PROJEKTA

Projekt koji će prikazati mogućnosti Unity-a i ML-Agenta za korištenje algoritama strojnog učenja sastojat će se od:

- okoline - u ovom projektu okolina će nasumično generirana cesta sa preprekama
- akademije – jedinstveni objekt koji mora postojati na sceni, sadrži mozgove kao objekte djece, u njoj se definiraju neki parametri treniranja
- mozga – u ovom slučaju postojati će samo jedan mozak, on je odgovoran za odlučivanje o djelovanju agenta
- agenta - inteligentni agent koji uči željeno ponašanje [10]

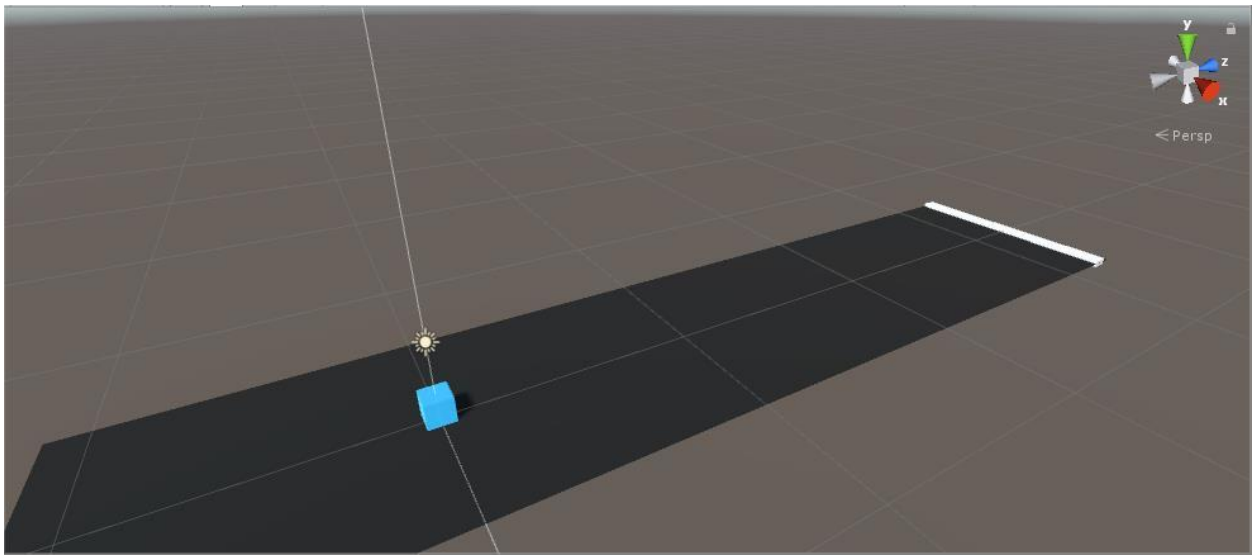
4.1. Kreiranje okoline

Na slici 4.1. prikazana je scena simulacije unutar Unity uređivača. Unutar scene je startna ploča ceste na koju se kod svake iteracije učenja dodaju ostale ploče da bi stvorile cestu. Plava kocka predstavlja agenta. U gornjem desnom kutu su brojači koji nadziru kroz koliko crvenih i zelenih kugli je agent prošao.



Sl. 4.1. Scena simulacije u Unityju

Cilj do kojeg agent treba doći prikazan je bijelim kvadrom koji se uvijek nalazi na zadnjoj ploči generirane ceste. Cilj uvijek ima istu poziciju na zadnjoj ploči ceste, ali pošto je cesta nasumično generirana cilj ima nasumičnu lokaciju u globalnom prostoru. Na slici 4.2. prikazan je cilj.



Sl. 4.2. Cilj

Dolazak na lokaciju cilja bez dodirivanja ijedne prepreke je glavni zadatak agenta u ovom projektu. Na slici 4.3. prikazan je C# kod cilja u kojemu postoji samo jednostavna ugrađena *callback* funkciju koja detektira sudar objekta agenta i objekta cilja i kada se on dogodi agent dobiva veliku nagradu i označava se da je završio iteraciju. Uz to dodaje se i nagrada ovisno o vremenu koje je preostalo za završavanje iteracije. Što je više vremena ostalo to je veća nagrada i time se potiče agenta da što brže obavi svoj zadatak.

```
using UnityEngine;

public class Goal : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        MyAgent agent = other.gameObject.GetComponent<MyAgent>();
        if (agent != null) {
            agent.AddReward(agent.timer * 0.005f);
            agent.AddReward(1f);
            agent.Done();
        }
    }
}
```

Sl. 4.3. Kod cilja

Da bi se otežao zadatak i svelo učenje na općeniti slučaj cesta kojom agent mora proći nasumično je generirana svaku iteraciju. Ako agent pređe granice ceste on pada i iteracija se smatra „neuspješnom“, odnosno agent dobiva veliku negativnu nagradu. Broj ploča od kojih se stvara cesta je nasumično odabran unutar postavljenih granica. Isto tako kut svake ploče se nasumično određuje unutar postavljenih granica. Na slikama 4.4. i 4.5. prikazani je C# kod klase Road u kojemu postoji metoda CreateRoad za stvaranje ceste. U toj metodi se pomoću metode CreateTile stvaraju ploče sa nasumičnim kutom dobivenim iz metode GetRandomEulerAngles. Da bi se svaka ploča preklapala sa zadnjom kreiranom koristi se metoda GetPosition koja vraća poziciju na koju treba postaviti novu ploču ceste. Zadnja stvorena ploča je posebna jer jedina sadržava objekt cilja. U klasi Road postoji još i metoda Reset koja se poziva u svakoj novoj iteraciji. Metoda Reset uništava sve objekte ploča ceste u okolini te ih ponovno nasumično stvar pozivajući metodu CreateRoad.

```
using UnityEngine;

public class Road : MonoBehaviour
{
    public GameObject startTile;
    public GameObject roadTile;
    public GameObject endTile;
    public float angleMin;
    public float angleMax;
    public int tileCountMin;
    public int tileCountMax;

    private GameObject previousTile;

    void Awake()
    {
        CreateRoad();
    }

    public void CreateRoad()
    {
        startTile.transform.eulerAngles = GetRandomEulerAngles(angleMin, angleMax);
        previousTile = startTile;
        int count = Random.Range(tileCountMin, tileCountMax + 1);
        for (int i = 0; i < count; i++) {
            GameObject newTile;
            if (i == count - 1) {
                newTile = CreateTile(endTile);
            }  else {
                newTile = CreateTile(roadTile);
            }
            previousTile = newTile;
        }
    }

    private GameObject CreateTile(GameObject tile)
    {
        GameObject newTile = Instantiate(tile, GetPosition(), Quaternion.identity,
transform);
        newTile.transform.eulerAngles = GetRandomEulerAngles(angleMin, angleMax);
        return newTile;
    }

    private Vector3 GetRandomEulerAngles(float min, float max)
    {
        Vector3 eulerAngles = Vector3.zero;
        eulerAngles.y = Random.Range(angleMin, angleMax);
        return eulerAngles;
    }
```

Sl. 4.4. Kod ceste 1. dio

```
    private Vector3 GetPosition()
    {
        Vector3 position = previousTile.transform.position;
        position.z += 15;
        float previousTileEulerY = previousTile.transform.eulerAngles.y > angleMax ?
previousTile.transform.eulerAngles.y - 360 : previousTile.transform.eulerAngles.y;
        position.x += previousTileEulerY < 0 ? previousTileEulerY / 10 :
previousTileEulerY / 10;
        return position;
    }

    public void Reset()
    {
        for (int i = 0; i < transform.childCount; i++) {
            Destroy(transform.GetChild(i).gameObject);
        }
        CreateRoad();
    }
}
```

Sl. 4.5. Kod ceste 2. dio

Cesta je od sastavljena od ploča koje na sebi mogu sadržavati crvenu kuglu. Crvena kugla predstavlja prepreku koju agent mora zaobići i ako prođe kroz nju iteracija neuspješno završava i agent dobiva veliku negativnu nagradu. Na slici 4.6. prikazan je kod klase RoadTile sa samo jednom metodom Start koja se pokreće samo jednom na početku životnog ciklusa objekta. U Start metodi se ovisno o postavljenoj šansi stvaranja prepreke i dobivenoj nasumičnoj vrijednosti stvara ili ne stvara prepreka na ploči ceste. Ako se prepreka stvori njena lokacija na ploči se nasumično određuje unutar postavljenih granica.

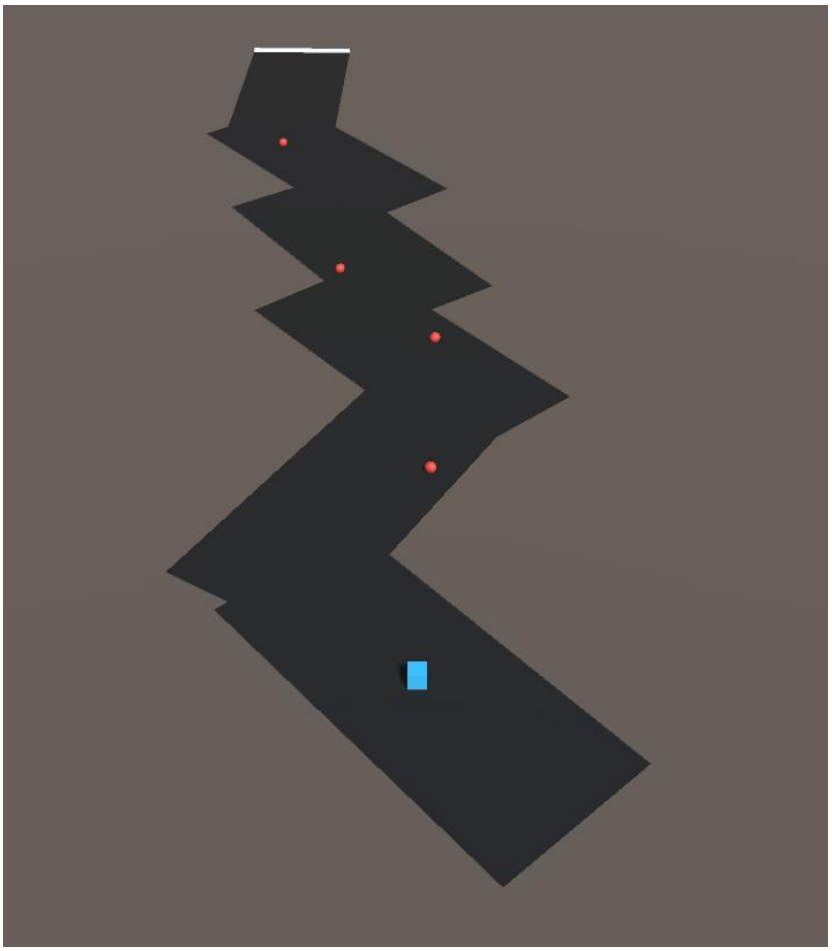
```
using UnityEngine;

public class RoadTile : MonoBehaviour
{
    public GameObject obstacle;
    public float maxBallPositionX;
    public float maxBallPositionZ;
    [Range(0f, 1f)]
    public float obstacleSpawnChance;

    private void Start()
    {
        if (Random.Range(0f, 1f) <= obstacleSpawnChance) {
            obstacle.SetActive(true);
            obstacle.transform.localPosition = new Vector3(Random.Range(-
maxBallPositionX, maxBallPositionX), 0.375f, Random.Range(-maxBallPositionZ,
maxBallPositionZ));
        }
```

Sl. 4.6. Kod ploče ceste

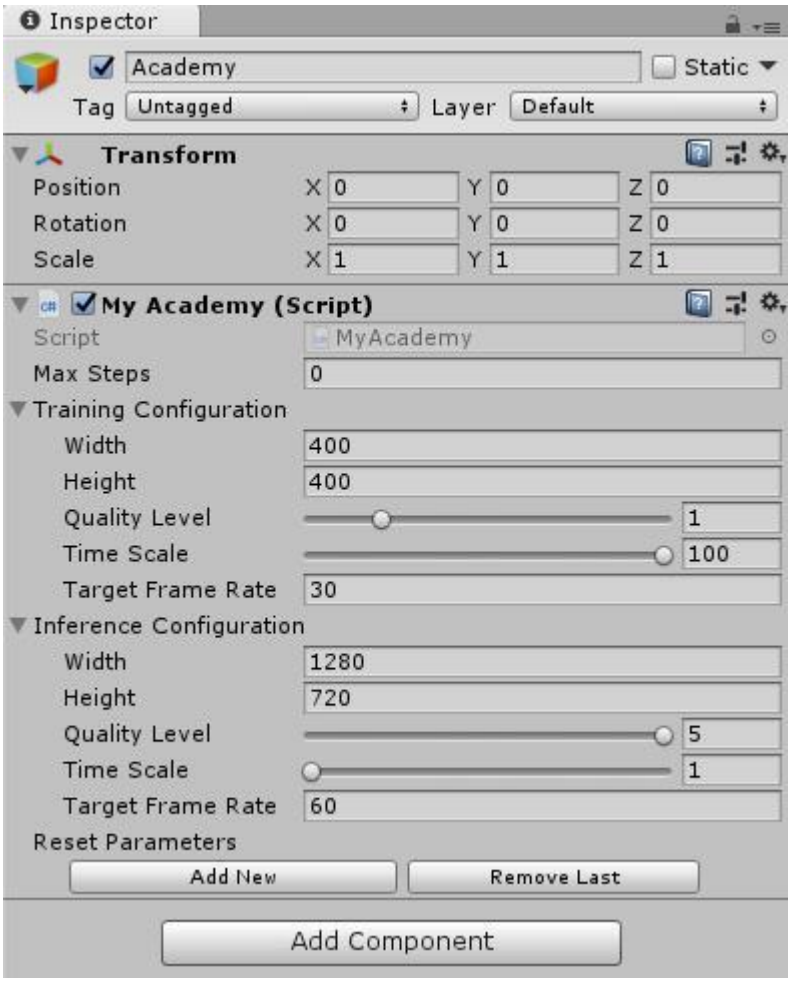
Na slici 4.7. prikazana je okolina sa generiranom cestom u jednoj iteraciji simulacije.



Sl. 4.7. Prikaz generirane ceste

4.2. Implementacija akademije

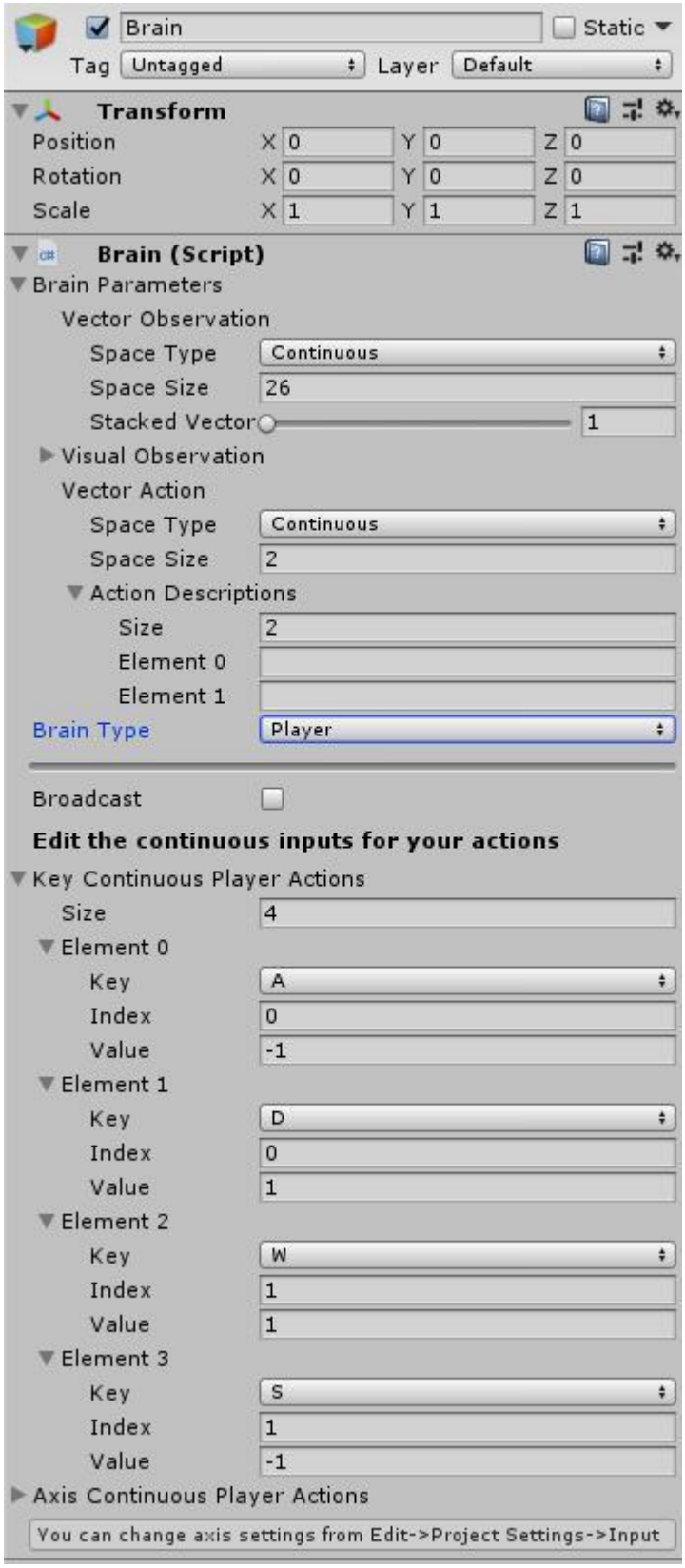
Objekt akademije orkestrira svim agentima i mozgovima na sceni. Svaka scena koja sadrži agenta mora sadržavati i jednu akademiju. Metodama akademije se može inicijalizirati okolinu nakon što se scena učita, resetirati okolinu i promijeniti atribute objekata u svakom koraku simulacije. Implementiranje tih metoda akademije nije obavezno i u ovom projektu se ne koriste. [11] Na slici 4.8. prikazane su komponente objekta akademije i svojstva komponente akademije. U svojstvima akademije može se namjestiti maksimalan broj koraka do resetiranja okoline. Isto tako mogu se podešavati konfiguracije treninga i zaključivanja koje se odnose na kvalitetu iscrtavanja i brzinu *enginea*. Zadnje svojstvo akademije su korisnički definirani parametri.



Sl. 4.8. Svojstva komponente akademije

4.3. Implementacija mozga

Mozak obuhvaća proces donošenja odluka. U sceni mozak mora bit objekt dijete objekta akademije. Svakom agentu mora biti dodijeljen mozak. Mozak ne zahtijeva korisnikovu implementaciju metoda već se direktno koristi ugrađeni mozak. Tijekom treniranja mozak se postavlja na eksterni, a nakon što je model istreniran, u fazi zaključivanja mozak se postavlja na interni. Kad se testira scena i agentove akcije, mozak se postavi na igračev i korisnik pridruži kontrole se tipkovnice agentovima akcijama te može direktno upravljati agentom [12]. Na slici 4.9. prikazane su komponente objekta mozga i svojstva komponente mozga. U mozgu se postavlja veličina i tip vektora opažanja, vizualnih opažanja i akcija. Tip akcija i opažanja može biti diskretan i kontinuiran.

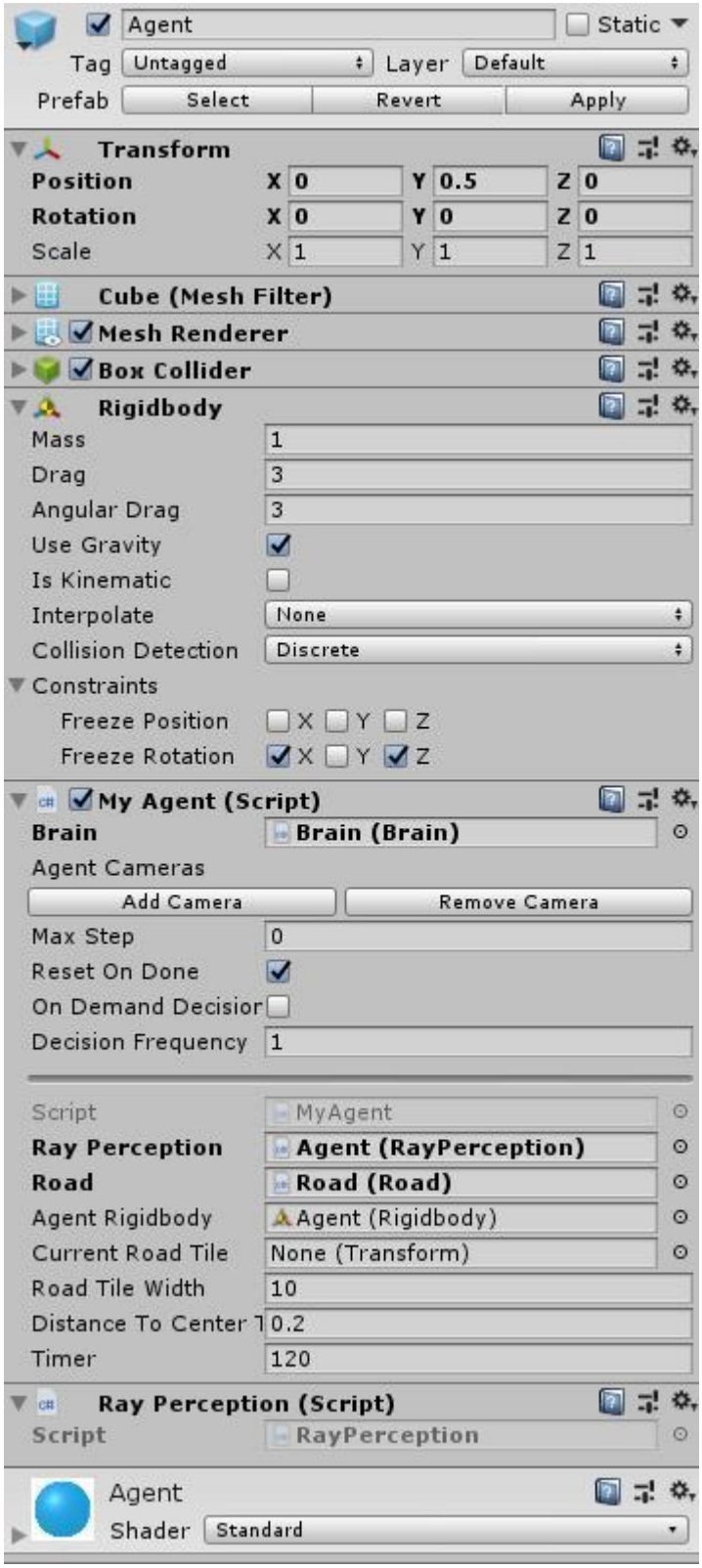


Sl. 4.9. Svojstva komponente mozga

4.4. Implementacija agenta

Agent je sudionik simulacije kojeg se trenira da obavlja određeni zadatak. On može opažati okolinu i djelovati akcijama ovisno o tim opažanjima. Najvažniji aspekti stvaranja agenta su opažanja koja agent skuplja i nagrade koju su mu dodijeljene za procjenjivanje vrijednosti trenutnog stanja u ovisnosti o obavljanju zadatka. Za implementiranje agenta potrebno je implementirati vlastitu klasu koja nasljeđuje od ugrađene, predefinirane klase agenta. Obavezno je implementirati metode CollectObservations i AgentAction. CollectObservations metoda služi za prikupljanje opažanja i u njoj treba definirati koje stavke u okolini su agentu bitne i čije stanje treba prikupljati. AgentAction metoda definira akcije koje agent čini u svakom koraku simulacije. Akcije i nagrade za pojedine akcije su definirane u agentu, ali mozak odlučuje koju akciju će agent izvesti. Potrebno je i definirati na koji način agent završava obavljanje zadatka ili postaviti maksimalni broj koraka koji agent može napraviti u jednoj iteraciji. [13]

Na slici 4.10. prikazane su komponente objekta agenta i svojstva komponente agenta. U svojstvima agenta mora se pridružiti jedan mozak. Postoji i opcija dodavanja kamera za vizualna zapažanja, definiranje maksimalnog broja koraka agenata u iteraciji i opcije za odlučivanje.



Sl. 4.10. Svojstva komponente agenta

Na slikama 4.11., 4.12., 4.13. i 4.14. prikazan je C# kod korisnički definirane klase agenta. U metodi CollectObservations agent u opažanja dodaje svoj relativni položaj u x i z osi u ovisnosti

o položaju cilja, svoju brzinu u x i z osi, svoju udaljenost od centra ploče na kojoj se trenutno nalazi i lokalnu percepciju prostora ispred sebe koristeći metodu Percieve iz RayPerception klase. Svako opažanje je normalizirano što znači da su vrijednosti unutar intervala [-1, 1]. Udaljenost agenta od centra trenutne ploče ceste se dobiva tako da agent detektira ulazak u svaku novu ploču ceste te tu ploču postavlja kao trenutnu. Svaka ploča ceste sadrži dva objekta koji predstavljaju lijevi i desni rubnik ceste. Pomoću metoda DetectLeftCurbPoint i DetectRightCurbPoint, odnosno metoda DetectFrontCurbPoint i DetectBackCurbPoint dobivaju se najbliže točke oba rubnika u ovisnosti o agentu. Udaljenosti agenta i točaka rubnika se oduzimaju i dijele sa širinom ploče zbog normalizacije. Metoda Perceive iz RayPerception klase detektira objekte unutar postavljenog radijusa i kutova oko agenta. Ako detektira objekt čija oznaka je jednaka oznaci koja je predana metodi vraća 1.

U metodi AgentAction prvo se provjerava je li isteklo postavljeno maksimalno vrijeme obavljanja zadatka, ako je iteracija se završava i agent dobiva veliku negativnu nagradu. Nakon toga provjerava se je li se udaljenost agenta od cilja smanjila u odnosu na prošlu za određenu razliku. Ako je onda se agentu dodjeljuje mala pozitivna nagrada, a ako je i pritom njegova udaljenost od centra trenutne ploče unutar zadanih granica dobiva i veću nagradu. Isto tako se provjerava je li agent prešao rubnike ploče i pao sa ceste, u tom slučaju bi dobio veliku negativnu nagradu i završila bi trenutna iteracija. Nakon toga ovisno o odluci mozga pozivom Move metode agent se miče po x i z osi.

U *callback* metodi OnTriggerEnter se provjerava je li agent dotaknuo prepreku i ako je iteracija se smatra neuspješnom i završava te agent dobiva veliku negativnu nagradu.

```
using UnityEngine;
using MLAgents;

public class MyAgent : Agent
{
    public RayPerception rayPerception;
    public Road road;
    public Rigidbody agentRigidbody;
    public Transform currentRoadTile;
    public float roadTileWidth;
    public float distanceToCenterThresh = 0.2f;
    public float timer;

    private Transform goal;
    private Vector3 startPosition;
    private Vector3 startRotation;
    private Vector3 leftCurbPoint;
    private Vector3 rightCurbPoint;
    private Vector3 movementForce;
    private Vector3 relativePositionToGoal;
    private float distanceToCenter;
    private float previousDistance = float.MaxValue;
    private float xPointRelativeGoalPositionMin;
    private float zPointRelativeGoalPositionMin;
    private float rewardDistance;
    private float startingTimer;
    private float maxSpeed = 13;
    private bool isInCenter;

    private void Start()
    {
        goal = FindObjectOfType<Goal>().transform;
        startPosition = transform.position;
        startRotation = transform.eulerAngles;
        InitializeRelativeGoalPointsMinimum();
        startingTimer = timer;
        rewardDistance = Vector3.Distance(transform.position, goal.position) / 12;
    }

    private void Update()
    {
        timer -= Time.deltaTime;
    }

    public override void AgentReset()
    {
        road.Reset();
        transform.position = startPosition;
        transform.eulerAngles = startRotation;
        agentRigidbody.angularVelocity = Vector3.zero;
        agentRigidbody.velocity = Vector3.zero;
        goal = FindObjectOfType<Goal>().transform;
        InitializeRelativeGoalPointsMinimum();
        isInCenter = true;
        previousDistance = float.MaxValue;
        timer = startingTimer;
    }
```

Sl. 4.11. Kod korisnički definirane klase agenta 1. dio

```
public override void CollectObservations()
    {
        relativePositionToGoal = GetRelativePosition(goal, transform.position);
        relativePositionToGoal.x = Map(relativePositionToGoal.x,
xPointRelativeGoalPositionMin, 0, -1, 1);
        relativePositionToGoal.z = Map(relativePositionToGoal.z,
zPointRelativeGoalPositionMin, 0, -1, 1);

        AddVectorObs(relativePositionToGoal.x);
        AddVectorObs(relativePositionToGoal.z);

        DetectLeftCurbPoint();
        DetectRightCurbPoint();
        distanceToCenter = (Vector3.Distance(transform.position, leftCurbPoint) -
Vector3.Distance(transform.position, rightCurbPoint)) / roadTileWidth;

        AddVectorObs(distanceToCenter);

        float rayDistance = 10f;
        float[] rayAngles = { 20f, 90f, 160f, 45f, 135f, 70f, 110f };
        string[] detectableObjects = { "redBall"};
        AddVectorObs(rayPerception.Perceive(rayDistance, rayAngles,
detectableObjects, 0f, 0f));

        AddVectorObs(agentRigidbody.velocity.z / maxSpeed);
        AddVectorObs(agentRigidbody.velocity.x / maxSpeed);
    }

    public override void AgentAction(float[] vectorAction, string textAction)
    {
        if (timer <= 0) {
            AddReward(-1f);
            Done();
        }
        CheckIfInCenter();
        CheckIfDistanceToGoalHasReduced();
        CheckIfAgentHasFallenOff();

        Move(vectorAction[0], vectorAction[1]);
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Road") {
            currentRoadTile = collision.transform;
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "redBall") {
            AddReward(-1.0f);
            Done();
        }
    }
```

Sl. 4.12. Kod korisnički definirane klase agenta 2. dio

```
private void CheckIfInCenter()
    {
        if (Mathf.Abs(distanceToCenter) <= distanceToCenterThresh) {
            isInCenter = true;
        } else if (Mathf.Abs(distanceToCenter) > distanceToCenterThresh) {
            isInCenter = false;
        }
    }

    private void CheckIfDistanceToGoalHasReduced()
    {
        float distanceToTarget = Vector3.Distance(transform.position, goal.position);

        if (distanceToTarget < previousDistance - rewardDistance) {
            AddReward(0.1f);
            Debug.Log("Adding 0.1 reward for getting closer to the goal");
            if (isInCenter) {
                AddReward(0.15f);
                Debug.Log("Adding 0.15 reward for staying in center");
            }
            previousDistance = distanceToTarget;
        }
    }

    private void CheckIfAgentHasFallenOff()
    {
        if (transform.position.y < 0.49) {
            AddReward(timer * -0.005f);
            AddReward(-1.0f);
            Done();
        }
    }

    private void Move(float xForce, float zForce)
    {
        movementForce = Vector3.zero;
        movementForce.x = Mathf.Clamp(xForce, -1f, 1f);
        movementForce.z = Mathf.Clamp(zForce, -1f, 1f);
        agentRigidbody.AddForce(movementForce, ForceMode.Impulse);
    }

    private void DetectLeftCurbPoint()
    {
        RaycastHit raycastHit = new RaycastHit();
        if (Physics.Raycast(transform.position, Vector3.left, out raycastHit)) {
            if (raycastHit.collider != null && currentRoadTile != null &&
raycastHit.collider.gameObject.transform == currentRoadTile.GetChild(0)) {
                leftCurbPoint = raycastHit.point;
            }
        } else {
            DetectFrontCurbPoint();
        }
    }
```

Sl. 4.13. Kod korisnički definirane klase agenta 3. dio

```
    private void DetectRightCurbPoint()
    {
        RaycastHit raycastHit = new RaycastHit();
        if (Physics.Raycast(transform.position, Vector3.right, out raycastHit)) {
            if (raycastHit.collider != null && currentRoadTile != null &&
raycastHit.collider.gameObject.transform == currentRoadTile.GetChild(1)) {
                rightCurbPoint = raycastHit.point;
            }
        } else {
            DetectBackCurbPoint();
        }
    }

    private void DetectFrontCurbPoint()
    {
        RaycastHit raycastHit = new RaycastHit();
        Physics.Raycast(transform.position, Vector3.forward, out raycastHit);
        if (raycastHit.collider != null && currentRoadTile != null &&
raycastHit.collider.gameObject.transform == currentRoadTile.GetChild(0)) {
            leftCurbPoint = raycastHit.point;
        }
    }

    private void DetectBackCurbPoint()
    {
        RaycastHit raycastHit = new RaycastHit();
        Physics.Raycast(transform.position, Vector3.back, out raycastHit);
        if (raycastHit.collider != null && currentRoadTile != null &&
raycastHit.collider.gameObject.transform == currentRoadTile.GetChild(1)) {
            rightCurbPoint = raycastHit.point;
        }
    }

    private Vector3 GetRelativePosition(Transform origin, Vector3 position)
    {
        Vector3 distance = position - origin.position;
        Vector3 relativePosition = Vector3.zero;
        relativePosition.x = Vector3.Dot(distance, origin.right.normalized);
        relativePosition.y = Vector3.Dot(distance, origin.up.normalized);
        relativePosition.z = Vector3.Dot(distance, origin.forward.normalized);

        return relativePosition;
    }

    private void InitializeRelativeGoalPointsMinimum()
    {
        Vector3 relativePositionToGoal = GetRelativePosition(goal,
transform.position);
        xPointRelativeGoalPositionMin = relativePositionToGoal.x;
        zPointRelativeGoalPositionMin = relativePositionToGoal.z;
    }

    private float Map(float value, float from1, float to1, float from2, float to2)
    {
        return (value - from1) / (to1 - from1) * (to2 - from2) + from2;
    }
}
```

Sl. 4.14. Kod korisnički definirane klase agenta 4. dio

4.5. Treniranje agenta

Treniranje agenta odvija se u eksternom Python procesu. Eksterni proces komunicira s objektom akademije unutar scene da bi generirao blok iskustava agenta. Ta iskustva postaju trening set za neuronsku mrežu koji se koriste za optimizaciju pravila koje agent stvara. U ovom slučaju se koristi učenje potporom i neuronska mreža optimizira pravila tako što maksimizira očekivane nagrade.

Za stabilno i uspješno učenje potrebno je postaviti hiperparametre. Hiperparametri koji se postavljaju u .yaml konfiguracijskoj datoteci prikazani su na slici 4.15. Hiperparametri su sljedeći:

- **batch_size** – broj iskustava koji se koristi za jednu iteraciju ažuriranja gradijentnog spusta, treba biti djelitelj *buffer_sizea*. Tipične vrijednosti za kontinuirani prostor akcija 512 – 5120, a za diskretni prostor akcija 32 – 512.
- **beta** – odgovara snazi pravilnosti entropije, koja čini pravila više slučajnim, osigurava da agent ispravno istraži okolinu tijekom treninga. Tipične vrijednosti su $1e^{-4}$ – $1e^{-2}$.
- **buffer_size** – odgovara tome koliko iskustava treba biti prikupljeno prije nego što se model ažurira, treba biti višekratnik *batch_sizea*. Tipične vrijednosti su 2048-409600.
- **epsilon –** odgovara prihvatljivom pragu divergencije između starih i novih pravila tijekom ažuriranja gradijentnog spusta. Tipične vrijednosti 0,1 – 0,3.
- **gamma** – odgovara faktoru smanjenja za buduće nagrade. To se može smatrati koliko daleko u budućnost agent treba brinuti o mogućim nagradama. Tipične vrijednosti su 0,8 – 0,995.
- **hidden_units** – odgovara broju jedinica koje se nalaze u svakom potpuno povezanom sloju neuronske mreže. Tipične vrijednosti su 32 – 512.
- **lambd –** odgovara lambda parametru koji se koristi za računanje GAE (*Generalized Advantage Estimate*), može se smatrati koliko se agent oslanja na svoju trenutačnu procjenu vrijednosti pri računanju ažurirane procjene vrijednosti. Tipične vrijednosti su 0,9 – 0,95.
- **learning_rate –** odgovara jačini svakog koraka ažuriranja gradijentnog spusta. Tipične vrijednosti $1e^{-5}$ – $1e^{-3}$
- **max_steps** – odgovara maksimalnom broju koraka koji simulacija može odraditi tijekom treninga; tipične vrijednosti $5e^{5}$ – $1e^{7}$.

- **memory_size** – koristi se samo kada je *use_recurrent* postavljen na *true*. Odgovara veličini polja brojeva sa pomičnim zarezom koji se koriste za spremanje skrivenih stanja povratne neuronske mreže. Treba biti višekratnik broja 4. Tipične vrijednosti su 64 – 512.
- **normalize** – odgovara tome primjenjuje li se normalizacija na ulaze vektorskih zapažanja.
- **num_epochs** – broj prolazaka kroz spremnik iskustava tijekom gradijentnog spusta. Tipične vrijednosti su 3 – 10.
- **num_layers** – broj skrivenih slojeva koji su prisutni nakon ulaznih vrijednosti zapažanja. Tipične vrijednosti su 1 – 3.
- **time_horizon** – odgovara tome koliko koraka prikupljanja iskustva po agentu treba obaviti prije dodavanja u spremnik iskustva, kada je postignuta granica prije kraja epizode, procjena vrijednosti se koristi za predviđanje ukupne očekivane nagrade od trenutačnog stanja agenta. Tipične vrijednosti su 32 – 2048.
- **sequence_length** - koristi se samo kada je *use_recurrent* postavljen na *true*. Odgovara dužini sekvenci iskustva koje prolaze kroz mrežu tijekom treninga. Tipične vrijednosti su 4 – 128.
- **curiosity_strength** – koristi se kada je *use_curiosity* postavljen na *true*. Odgovara veličini intrinzične nagrade koju generira intrinzični modul znatiželje. Tipične vrijednosti su 0,1 – 0,001.
- **curiosity_enc_size** – odgovara veličini skrivenog sloja koji se koristi za kodiranje promatranja unutar intrinzičnog modula znatiželje. Tipične vrijednosti su 64 – 256. [14]

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 2.0e6
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    use_curiosity: false
    curiosity_strength: 0.01
    curiosity_enc_size: 128
```

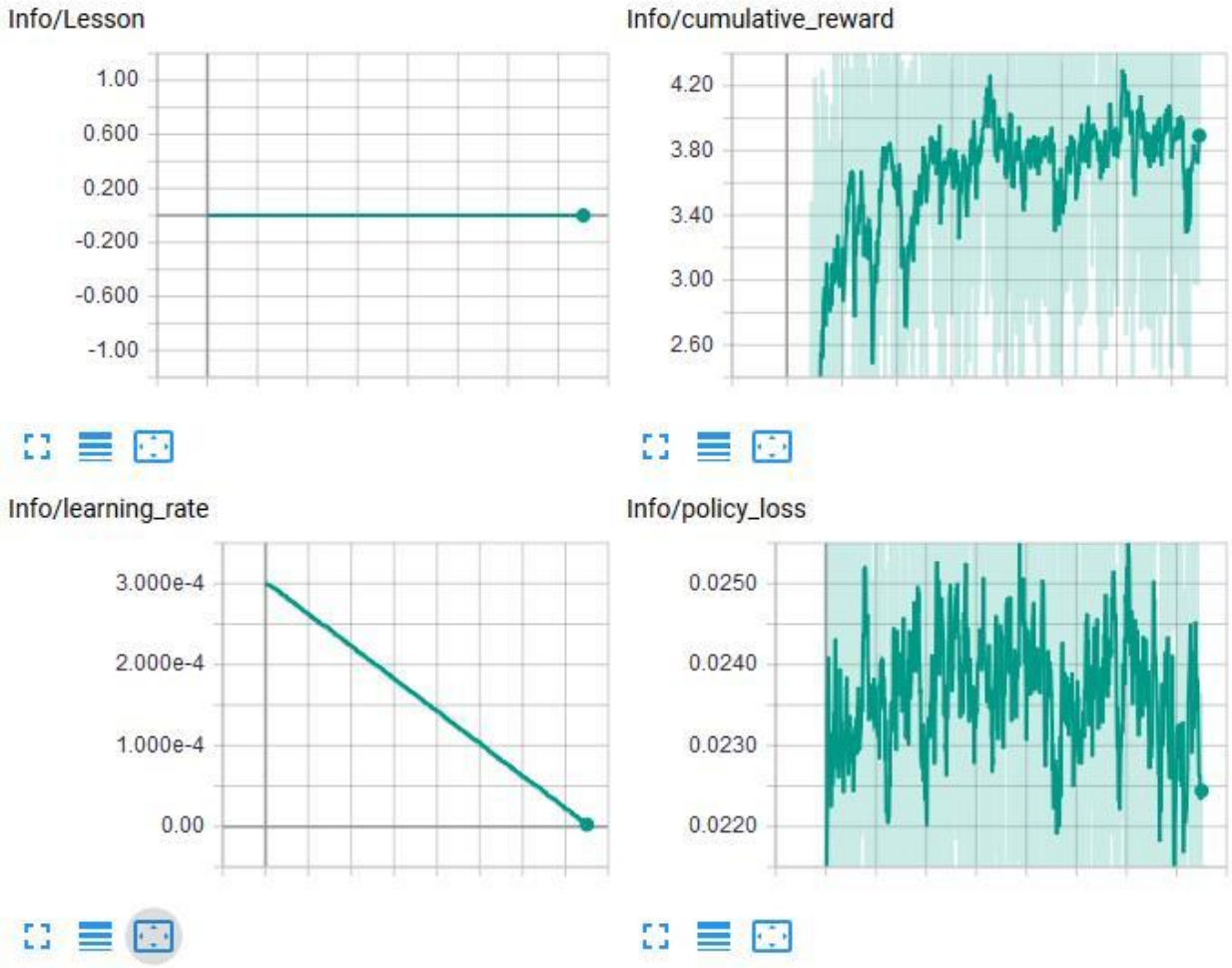
Sl. 4.15. Konfiguracijska datoteka za treniranje

4.6. Rezultati treniranja

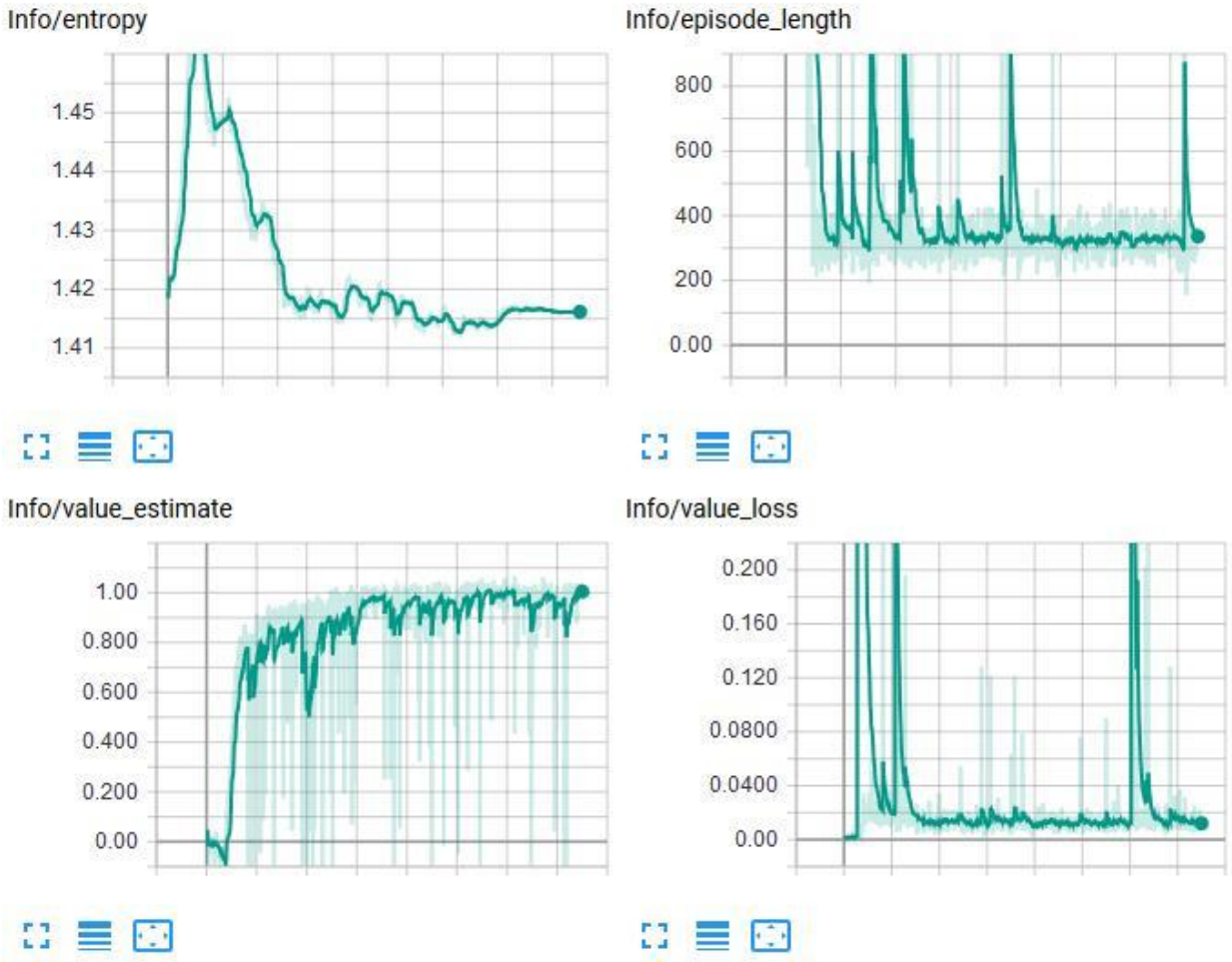
Tijekom treninga spremaju se različiti podaci koji tvore rezime cijelog treninga. Te podatke moguće je vizualizirati uporabom TensorBoard alata. Vizualizacija se može pratiti tijekom samog treniranja jer se podaci ažuriraju periodično ovisno o postavljenom *summary_freq* parametru unutar konfiguracijske datoteke.

Na slikama 4.16. i 4.17. prikazane su vizualizacije rezultata treniranja agenta za projekt ovog rada. Lesson graf se može zanemariti jer on ima značenje kada se koristi učenje s kurikulumom. Graf *cumulative_reward* prikazuje prosječnu kumulativnu vrijednost nagrade u svakoj iteraciji treninga. U uspješnom treningu trebala bi se povećavati kao što je u ovom slučaju. Graf *learning_rate* uvijek linearno opada i počinje od vrijednosti postavljene u konfiguracijskom parametru. Graf *policy_loss* prikazuje koliko se naučena pravila mijenjaju tijekom treninga, normalno je da oscilira, ali vrijednosti bi trebale biti ispod 1. Graf *entropy* pokazuje koliko su nasumične odluke agenta tijekom treninga, vrijednosti bi trebale opadati tijekom uspješnog treniranja kao što je u ovom slučaju. Graf *episode_length* prikazuje duljinu trajanja iteracije treninga. Graf *value_estimate* prikazuje koliku nagradu u budućnosti agent predviđa ovisno o svom trenutnom stanju, vrijednosti bi trebale rasti zajedno sa vrijednostima kumulativne nagrade. Posljednji graf *value_loss* prikazuje

koliko dobro model predviđa vrijednost svakog stanja, tijekom uspješnog učenja treba opadati kada se kumulativna nagrada stabilizira.



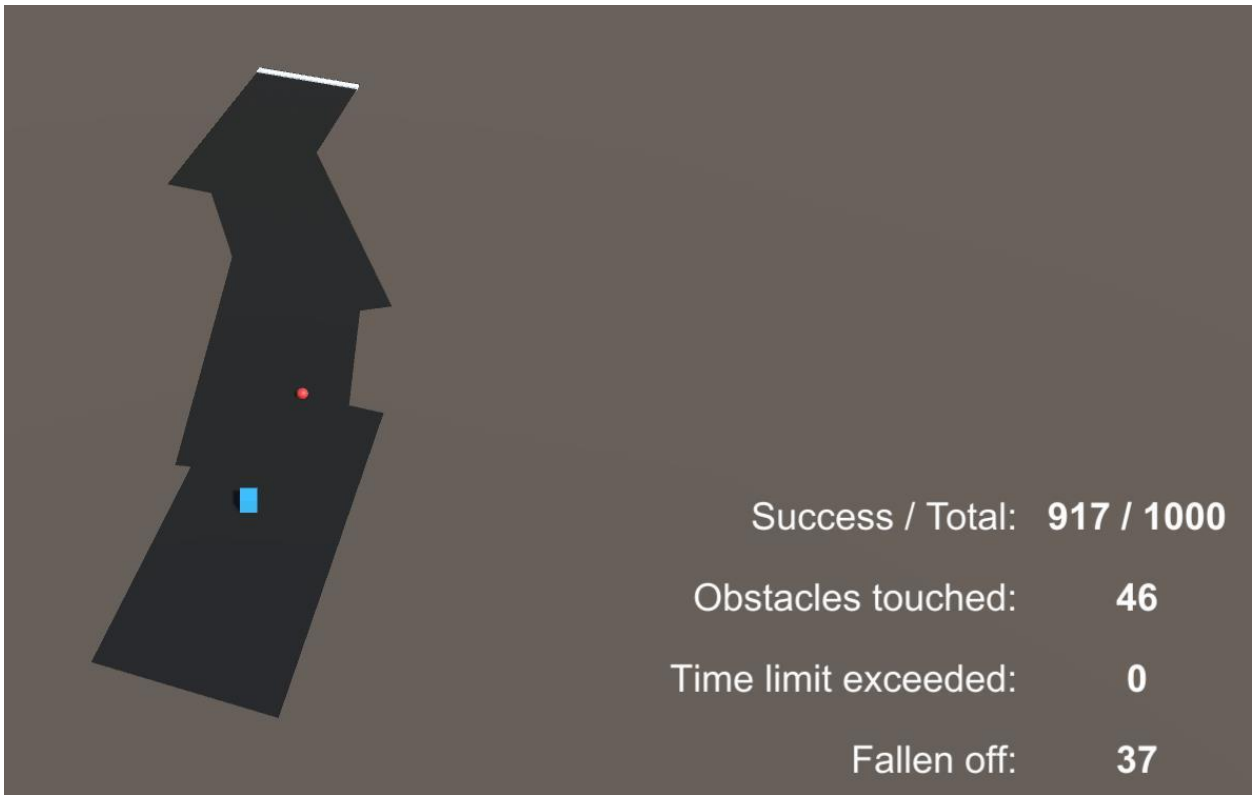
Sl. 4.16. Vizualizacija rezultata treniranja - 1.dio



Sl. 4.17. Vizualizacija rezultata treniranja - 2.dio

Nakon treniranja dodan je prikaz statistika koji prati uspješnost naučenih pravila treniranog modela. Na slici 4.18. prikazane su statistike pokrenute simulacije koja koristi interni mozak sa istreniranim modelom. U 1000 ukupnih iteracija simulacije 917 prolaza su uspješno odrađena, 46 puta je agent dotaknuo prepreku, a 37 puta je agent pao, odnosno prekoračio granice ceste. Ni u jednom slučaju agent nije prekoračio postavljeno maksimalno vrijeme obavljanja zadatka.

Od 8,3% neuspješnih iteracija 55,4% neuspjeha uzrokovano dodirivanjem prepreka, a ostalih 44,6% prekoračenjem granica ceste. Zaključuje se da je model malo uspješnije naučio pratiti cestu, nego izbjegavati prepreke. Uspješnost agenta iznosi 91,7% i može se zaključiti da iako nije savršen, istrenirani model prikazuje zadovoljavajući rezultat u smislu uspješnog obavljanja zadatka.



Sl. 4.18. Statistike istreniranog modela

5. ZAKLJUČAK

U ovom diplomskom radu istražene su i prikazane mogućnosti strojnog učenja u Unity *game engineu*. ML-Agents je novi *plugin* za Unity koji je korišten za izradu simulacije u kojoj je agent treniran učenjem sa potporom. U izrađenom projektu agent prolaziti nasumično generiranom cestom bez da izađe izvan njenih granica, isto tako agent uči izbjegavati prepreke, u ovom slučaju prikazane crvenim kuglama. ML-Agents je tek u beta fazi i još se razvija i kontinuirano mijenja, zbog toga njegove mogućnosti još nisu dovoljno velike i jake za rješavanje kompleksnih problema. Projekt ovog rada bi spadao u jednostavnije probleme gdje inteligentni agent mora paziti samo na nekolicinu varijabli, ali i dalje je bilo potrebno fino ugađanje hiperparametara i vrijednosti i trenutaka dobivanja nagrada. Za treniranje agenta u ovom projektu bilo je potrebno 15 milijuna koraka treninga za zadovoljavajuće rezultate. Za kompleksnije zadatke bilo bi potrebno puno više koraka i mnogo jače računalo da bi se treniranje odradilo u normalnom vremenu. Unatoč trenutnim poteškoćama i nedostacima, ovim projektom je prikazana mogućnost da se korištenjem besplatnog Unityja može izraditi simulacija strojnog učenja koja kasnije može biti implementirana na razna računala i strojeve u fizičkom svijetu. U budućnosti kada se ML-Agents dodatno razvije ovakva virtualna simulacija može uštediti novac i ubrzati proces razvoja novih tehnologija.

LITERATURA

[1] Unity

https://unity3d.com/unity, 20.6.2018.

[2] Unity editor

https://unity3d.com/unity/editor, 20.6.2018.

[3] Unity-Fast Facts

https://unity3d.com/public-relations, 20.6.2018.

[4] ML-Agents Overview

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md, 26.6.2018..

[5] Introduction to C# language and .NET framework

https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework, 24.6.2018.

[6] Background: TensorFlow

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Background-TensorFlow.md, 26.6.2018..

[7] Understanding Machine Learning: From Theory to Algorithms

http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning, 24.6.2018.

[8] Introducing Machine Learning

https://www.mathworks.com/content/dam/mathworks/tag-team/Objects/i/88174_92991v00_machine_learning_section1_ebook.pdf, 24.6.2018.

[9] Background: Machine Learning

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Background-Machine-Learning.md, 26.6.2018.

[10] Learning Environment Design

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design.md, 28.6.2018.

[11] ML-Agents Academy

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Academy.md, 28.6.2018.

[12] ML-Agents Brains

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Brains.md, 28.6.2018.

[13] ML-Agents Agent

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md, 28.6.2018.

[14] Training-PPO

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md, 20.9.2018.

SAŽETAK

U ovom diplomskom radu prikazuju se teorijske i praktične mogućnosti Unityj-a u izradi simulacije za učenje inteligentnih agenata algoritmima strojnog učenje. Unity je *game engine* koji se koristi za razvoj 2D i 3D videoigara, simulacija i interaktivnog sadržaja, a ML-Agents je plugin za Unity koji omogućuje izradu simulacija u kojima se koristi strojno učenje za treniranje inteligentnih agenata u obavljanju određenih zadataka. Strojno učenje obuhvaća metode koje omogućuje računalu da „uči“ obavljati određeni zadatak pomoću unesenih podataka, a bez eksplicitnog programiranja. U sklopu rada izrađen je projekt u kojem inteligentni agent uči korištenjem strojnog učenja s potporom. Zadatak agenta je prolaziti nasumično generiranom cestom i doći do njenog kraja. Na cesti se stvaraju prepreke koje agent mora zaobići, isto tako agent mora obaviti zadatak unutar određenog vremena i ne smije izaći iz granica ceste.

Ključne riječi: strojno učenje, učenje s potporom, Unity, ML-agents, inteligentni agent, C#, simulacija, TensorFlow

ABSTRACT

Machine learning in Unity

This graduate thesis presents theoretical and practical possibilites of Unity in making a simulation for traning intelligent agents with machine learning algorithms. Unity is a game engine which is used for developing 2D and 3D games, simulations and interactive content. ML-Agents is a Unity plugin which enables creating simulations that use machine learning algorithms to train intelligent agents in performing certain tasks. Machine learning encompasses methods that enable a computer to „learn“ to perform a particual task using input data, without explicit programming. Within the thesis project is created in which intelligent agent learns by using reinforcement machine learning. Agent has a task to navigate the randomly generated road and get to its end. Obstacles are created on the road which agent has to avoid, also agent has to complete the task within a certain time and must not leave the road boundaries.

Keywords: machine learning, reinforcement learning, Unity, ML-Agents, intelligent agent, C#, simulation, TensorFlow

ŽIVOTOPIS

Mateo Bareš je rođen 4.8.1994 u Osijeku. Od 2001. do 2009. pohađa OŠ Grigor Vitez. Nakon toga upisuje Isusovačku klasičnu gimnaziju u Osijeku koju završava 2013. godine. Iste godine redovno upisuje preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku koji završava 2016. godine. Nakon toga iste godine na istom fakultetu redovno upisuje diplomski studij računarstva gdje je sada student 2. godine.

Potpis:

_____