Primjena neuronskih mreža u pronalasku rješenja igre

Brazdil, Vedran

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

Sveučilišni studij

PRIMJENA NEURONSKIH MREŽA U PRONALASKU RJEŠENJA IGRE

Diplomski rad

Vedran Brazdil

Osijek, 2018.

SADRŽAJ

1. UVOD

Tema diplomskog rada je "Primjena neuronskih mreža u pronalasku rješenja igre". U teorijskom dijelu opisan je program Unity3D pomoću kojeg je izrađena 3D igra, programski jezik koji je korišten (C#), agenti strojnog učenja te način kako ih koristiti. U seminaru potrebno je teorijski opisati i neuronske mreže te način njihove primjene u pronalasku rješenja igre. U praktičnom dijelu rada potrebno je napraviti simulaciju igre koja će u nekoliko generacija sama doći do traženog rješenja. Praktični dio završnog rada napravljen je koristeći osobno računalo s Windows 7 operacijskim sustavom. Osim Unity3D-a korišteni su i razni drugi programi i tehnologije koje će biti navedene kroz seminar. Neki od tih programa korištenih za izvršavanje zadatka su Anaconda i MonoDevelop (korišten za pisanje programskog koda).

1.1. Zadatak diplomskog rada

U seminaru potrebno je teorijski opisati neuronske mreže te način njihove primjene u pronalasku rješenja igre. U praktičnom dijelu rada potrebno je napraviti simulaciju igre koja će u nekoliko generacija sama doći do traženog rješenja.

Tehnologije: C#, Unity3D, Tensorflow, Python, Anaconda Prompt

2. UNITY 3D

Unity [8][1] (hr. jedinstvo) je cross-platforma, koju je razvio Unity Technologies, stvorena je za izradu igara. Koristi se za izradu igara za računala (PC), konzole, mobilne uređaje te web stranice. Prvi put predstavljen 2005 godine na Appleovoj svjetskoj konferenciji za programere. Tada samo za OS X, a do danas proširio se i na još nekoliko platformi. 2006. godine na WWDC-u, Apple Inc. pohvalio je Unity za najbolje korištenje grafike Mac OS X. Trenutno najnovija verzija Unitya, Unity 2018, ujedno i jedna od najpopularnijih i najboljih razvojnih platformi za stvaranje 3D i 2D igara. Poboljšana učinkovitost, čini teški posao glatkim i zanimljivim te uz višeplatformsku podršku ima više korisnika nego ikada prije. Unity donosi novu umjetničku moć.

Uz naglasak na prenosivost, razvoj cilja na sljedeće API-je: Direct3D na Windowsu i Xbox 360; OpenGL na Macu i Windowsu; OpenGL ES na Androidu i iOS-u; te vlasničke API-je na video igračkim konzolama. Unity dozvoljava rezolucijske postavke za svaku platformu koju podupire te pruža specifikaciju kompresije, refleksijsko mapiranje, podršku za „bump“ mapiranje, „parallax“ mapiranje, dinamično zasjenjivanje korištenjem mapa sjena, prostorno ekranski prostor ambijentalne okluzije (SSAO), „render-to-texture“ i puni zaslon poslije procesnih efekta. Unityev grafički mehanizam platformske raznolikosti može pružiti sjenčanje s raznim varijantama.

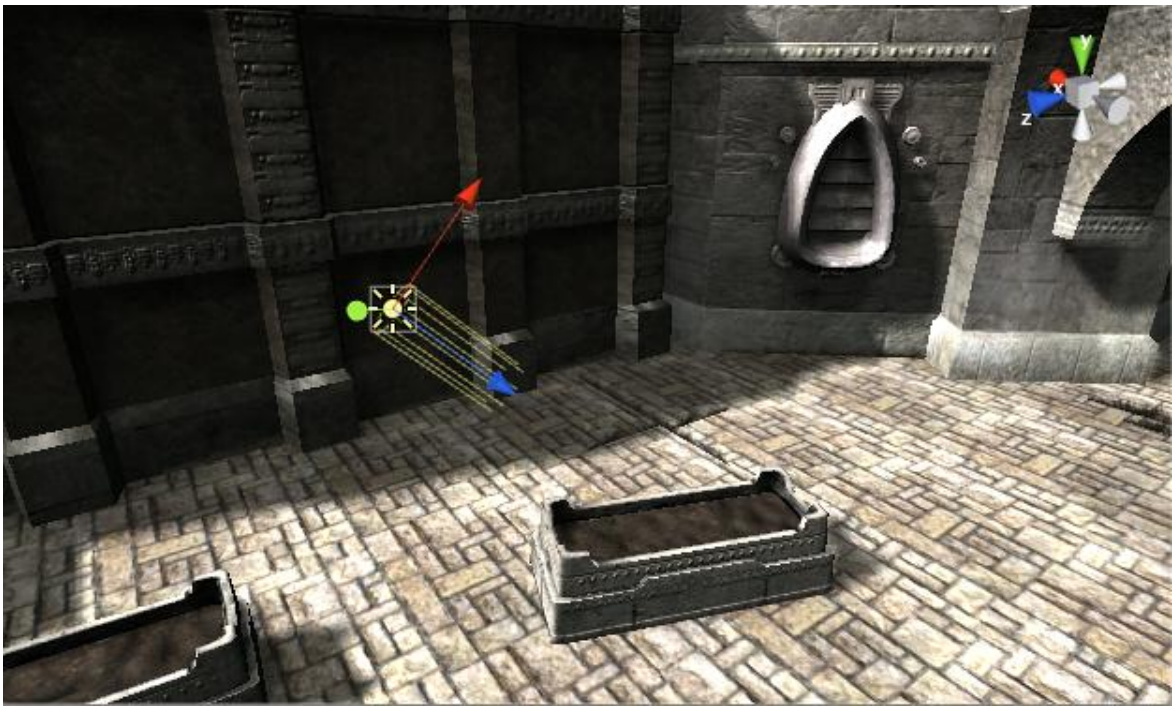
Unity je poznat po svojoj mogućnosti razvijanja igara na više platforma. Unutar projekta, programeri mogu isporučiti igru na mobilne uređaje, stolna računala (PC), web preglednike i konzole. Podržane platforme uključuju Android, iOS, OS X, Apple TV, BlackBerry 10, Linux, Nintendo 3DS line, PlayStation 3, PlayStation 4, PlayStation Vita, Unity Web Player, Wii, Wii U, Windows, Windows Phone 8, Xbox 360 i Xbox One. Unity Web Player je dodatak za preglednik (engl. *browser*) kojeg podržavaju samo Windows i OS X. Korištenje Unity Web Playera obustavljeno je u korist WebGL-a. Nintendo uz svaku dozvolu za programiranje daje i besplatnu kopiju samog programa te je osnovna oprema za stvaranje igara na Wii konzoli.

2.1. Grafika

Unity nudi iznenađujuću vizualnu točnost, snagu generiranja i ambijent. Unity, s obzirom na visoko razvijenu tehnologiju grafike koju koristi, omogućava programerima da igre naprave

onako kako su to zamislili. Razumijevanje grafike [6] ključno je za pravilno dodavanje i raspoređivanje elementa određene dubine u svoju igru.

Jedan od najbitnijih dijelova grafike u Unityju je osvjetljenje [5] koje daje velik udio na ambijent u igrama. Kako bi Unity izračunao sjenčanje 3D objekta, treba znati intenzitet, smjer i boju svjetlosti koja pada na taj 3D objekt. Izvor svjetlosti dobiva se tako da u scenu ubacimo objekt svjetlost. Najkorištenija vrsta osvjetljenja su usmjerena svjetla (engl. *directional lights*), koja omogućuju da svi objekti na sceni budu osvijetljeni. Svjetlost dolazi uvijek iz istog smjera. Usmjerena svjetla predstavljaju nekakav veliki izvor svjetlosti koji se nalazi izvan postojeće mape, kao što je prikazano na slici Sl. 2.1.1.



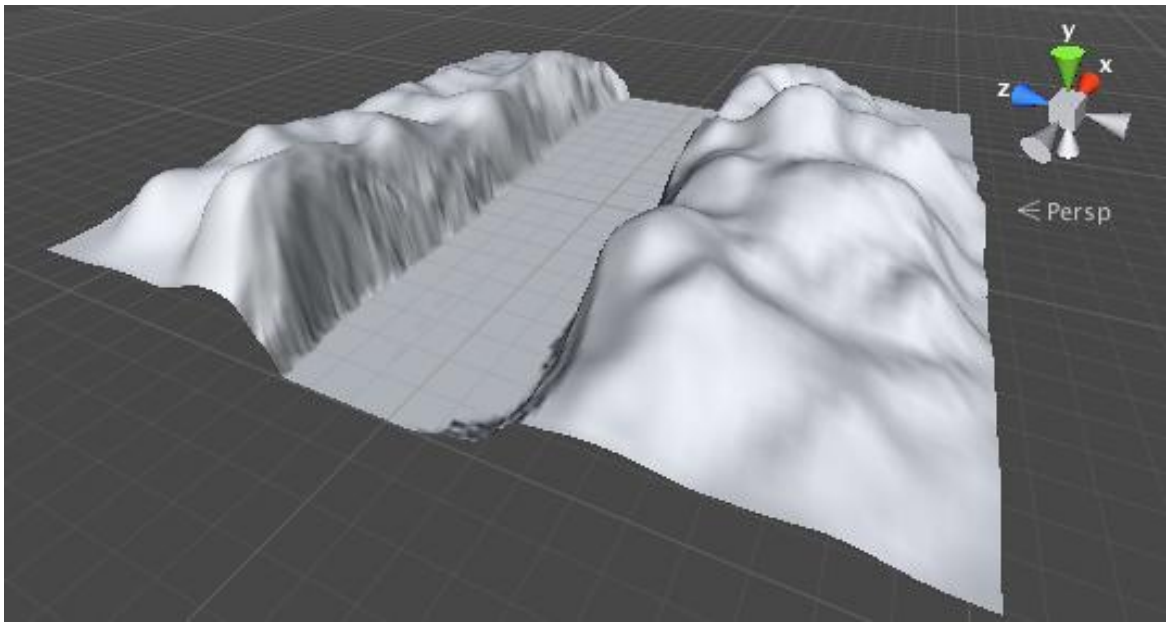
Sl. 2.1.1 Osvjetljenje scene [5]

Dodavanjem Unity svjetla u scenu stvaramo sjene koje s objekta padaju na drugi dio tog objekta ili na neki drugi objekt u prostoru. Na taj način, sjene dodaju određenu dubinu i realnost sceni zbog toga što izražavaju veličinu i položaj objekta koji bi inače izgledao ravno.

Unity scenu stvaramo slaganjem, organiziranjem i premetanjem objekta u trodimenzionalnom prostoru. Ekran korisnika je dvodimenzionalan pa zbog toga mora postojati način da bi se taj prizor uhvatio i „izravnao“ za prikaz na ekranu. Prikaz na ekranu postižemo uz pomoć objekta kamera [4].

Kamera je objekt koji stvara i definira pogled na prostor scene. Pozicija kamere u prostoru određuje točku gledišta, a osi objekta Z (naprijed) i Y (gore) određuju dubinu i domet gledanja, odnosno vrh ekrana. Kamera može isto tako definirati veličinu i oblik regije koja se nalazi unutar vidika. Kada namjestimo sve potrebne parametre, kamera može prikazivati što ona trenutno „vidi“, nama, na naš ekran. Ako se objekt kamere rotira i pomiče, pomicat će se i pogled kamere na našem ekranu isto kao što se pomiče i objekt. Kamera, naravno kao i samo ljudsko oko, u stvarnosti vidi svijet tako da su predmeti u daljini manji nego predmeti u blizini. Tehnika stvaranja takvog pogleda u igri je već poznata te se koristi u umjetnosti i grafici. Ta tehnika vrlo je važna za stvaranje realnih scena u igrama.

Važno je napomenuti kako Unity posjeduje sistem za stvaranje terena [7] koji omogućuje korisnicima da vrlo jednostavno i brzo dodaju velike krajolike u igru. Za vrijeme izvođenja, stvaranje terena vrlo je dobro optimizirano te pruža učinkovito renderiranje u samom izborniku. Postoji veliki izbor alata koji omogućuju vrlo lagano, brzo i raznoliko stvaranje terena za igre različitih vrsta.



Sl. 2.1.2 Stvaranje brda [7]

3. PROGRAMSKI JEZIK C#

C# [10] je višebrojno paradigmatski programski jezik. C# temelje različite programske discipline kao što su „strogo tipkane“, funkcionalne, imperativne, deklarativne, objektno-orijentirane, generičke i komponentno-orijentirane.

Microsoft je u okviru svoje inicijative .NET razvio C# programski jezik, koji je kasnije odobren kao standard (od strane Ecma (ECMA-334) i ISO (ISO/IEC 23270:2006)). On je jedan od programskih jezika koji je dizajniran da mu infrastruktura sliči uobičajenom jeziku (Common Language Infrastructure).

Anders Hejlsberg sa svojim timom nastavlja razvijati ovaj objektno-orijentiran programski jezik opće namjene. Najnovija verzija je C# 7.3, koja je izdana u 2018-oj godini zajedno uz Visual Studio 2017 verzije 15.7.2.

Ciljevi nastanka su:

- Namijenjen da bude opće namjene, jednostavan, moderan, objektno-orijentiran programski jezik.
- Uz svoju implementaciju osigurava potporu načelima programerskih inženjera kao što su provjere tipa i granica nizova, detekcija ne određenih varijabli koje se koriste u kodu te automatsko prikupljanje otpada.
- Programska robusnost, izdržljivost i programerska produktivnost su vrlo bitne.
- Namijenjen je za korištenje u razvoju softverskih komponenata prikladnih za primjenu u distribuiranim okruženjima.
- Prenosivost, koja je vrlo bitna za izvorni kod i programere, prednost je kod onih koji su već upoznati s programskim jezicima C i C++.
- Podrška za internacionalizaciju.
- Namijenjen je da bude pogodan za pisanje aplikacija sustava različitih veličina, koji mogu koristiti sofisticiran operacijski sustav ili nekog s posebnim funkcijama.

- C# aplikacije su namijenjene da budu ekonomične u pogledu memorije te ne zahtijevaju procesorsku snagu, no jezik nije namijenjen da se natječe s C ili nekim asemblerskim jezikom.

3.1. Sintaksa

Programski jezik C# ima sintaksu vrlo sličnu sintaksama ostalih C stilskih programskih jezika [11]. Neki od jezika sa sličnom sintaksom su C, C++ i Java.

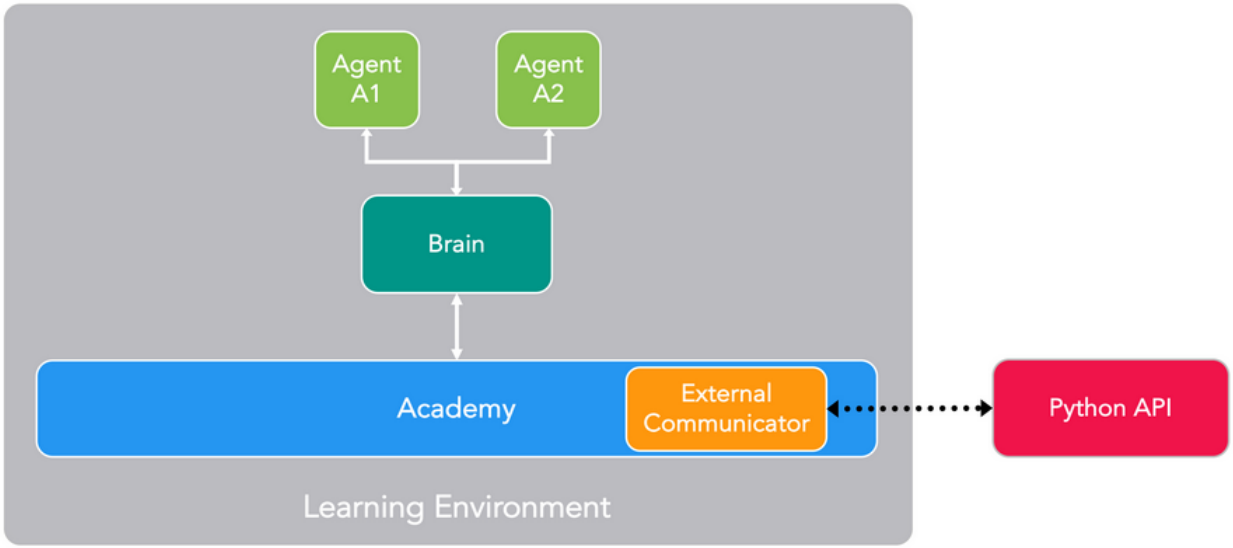
Slične stvari u sintaksi ovih jezika su:

- Točka-zarez („ ; “) se koristi kada označujemo kraj naredbe.
- Vitičaste zagrade („ { “ , „ } “) koriste se kod grupiranja naredbi u na primjer funkcije, klase ili u imenske prostore (engl. *namespace*).
- Uglate zagrade („ [“ , „] “) koriste se za deklariranje matrica i polja te da bi im dodali vrijednost, cijele matrice ili samo pod određenim indeksom.
- Varijablama se dodjeljuju vrijednosti koristeći znak jednakosti („ = “), a prilikom uspoređivanja koriste se dva uzastopna znaka jednakosti („ == “).

4. AGENTI STROJNOG UČENJA

Unity Alat Agenata Strojnog Učenja (engl. *Unity Machine Learning Agents Toolkit*), skraćeno ML-Agenti, je Unity dodatak otvorenog koda (engl. *open-source*) koji omogućuje igrama i simulacijama da služe kao okolina za treniranje inteligentnih agenata. Agenti se mogu trenirati korištenjem podržanog učenja (engl. *reinforcement learning*), imitacijskog učenja (engl. *imitation learning*), neuroevolucijom (engl. *neuroevolution*) ili nekih drugih metoda strojnog učenja kroz, jednostavan za korištenje, Python API. Isto tako ML-Agenti pružaju implementacije (temeljene na TensorFlowu) najnovijih algoritama koji omogućuju programerima igara da jednostavno uče inteligentne agente na 2D, 3D ili VR/AR vrstama igara. Ovi istrenirani agenti mogu se koristiti za različite stvari, uključujući kontroliranje ponašanja NPC-eva (likovi u igri koji se ne mogu kontrolirati ni igrati, engl. *non-player character*), automatiziranje testiranja igre pa čak i vrednovanje različitih odluka dizajnera prije nego se igra dovrši. ML-Agenti su obostrano korisni za programere igara i AI istraživače zbog pružanja svoje centralne platforme na kojoj se napredak AI razvitka može mjeriti i ocjenjivati, a zatim učiniti dostupnim velikom rasponu programera igara.

Agent je glumac koji može promatrati svoje okruženje i na temelju tih promatranja donijeti najbolju moguću odluku. Stvaranjem agenta u Unityju moguće je kroz proširenje klase „Agent“. Najvažniji aspekti stvaranja agenta, koji može uspješno učiti, su opservacije koje on prikuplja za učenje te nagrade koje dobiva na temelju svojih odluka (koje bi trebale težiti ispunjenju cilja).



Sl. 4.1 Okruženje treniranja [18]

Agent prosljeđuje svoje opservacije mozgu (engl. *the brain*). Mozak, tada, donosi odluke i prosljeđuje odabrane akcije natrag agentu. Agent mora izvršavati te proslijeđene akcije, na primjer kretanje naprijed ili nazad. Kako bismo naučili agenta korištenjem podržanog učenja, agent mora dobiti odgovarajuću nagradu uz svaku svoju akciju odnosno odluku. Nagrada se koristi kako bi se otkrila optimalna politika odlučivanja.

Važno je napomenuti da se nagrade ne dodjeljuju već istreniranom (naučenom) agentu ni prilikom imitacijskog učenja.

Klasa „Brain“ (mozak) izdvaja logiku donošenja odluka od samih agenta tako da se može koristiti isti mozak na više agenata. Donošenje odluka mozga ovisi o vrsti samog mozga. Vanjski mozak samo prenosi opservacije kroz agenta na neki od vanjskih procesa koji svoju odluku šalje natrag agentu, koji ju izvršava. Unutarnji mozak koristi već istreniranu politiku donošenja odluka kroz parametre (taj mozak neće naučiti ništa novo već samo koristiti naučeno). Ostale vrste mozgova nisu direktno vezane za trening, ali se mogu koristiti za dijelove treniranja.

4.1. Mozak

Mozak obuhvaća proces donošenja odluka. Objekti mozga moraju biti dijete Akademije (engl. *Academy*) unutar hijerarhije Unity scene. Svaki agent mora imati dodijeljen mozak, ali se isti mozak može dodijeliti više nego jednom agentu. Naravno, može se stvoriti više mozgova, koji će biti dodijeljeni jednom ili više agenata.

Predlaže se korištenje klase „Brain“ direktno a ne potklase. Ponašanje mozga određeno je njegovom vrstom. Agenti strojnog učenja (engl. *ML-Agents*) definiraju četiri vrste mozgova:

- Vanjski (engl. *External*) – Vanjska i unutarnja vrsta mozga pretežito rade zajedno. Ova vrsta koristi se prilikom treniranja agenta. To znači da ćemo koristiti nekakav vanjski proces koji će donositi odluke. U ovom slučaju koristit ćemo vanjski mozak koji će komunicirati s Python skriptom pomoću python „UnityEnvironment“ klase koja je uključena u python porciju samih agenta strojnog učenja SDK (ML-Agents SDK).

- Unutarnji (engl. *Internal*) – Korištenjem unutarnjih tipova mozga koristimo i već istreniranu politiku donošenja odluka.

- Istraživački (engl. *Heuristic*) – Ovaj tip namještamo prilikom ručnog kodiranja agentove logike kroz proširenje klase „Decision“.

- Igrač (engl. *Player*) – Namještamo vrstu igrač kada želimo osobno isprobati sve moguće akcije te ujedno testirati napisani kod.

Tijekom treninga koristit ćemo vrstu vanjskog mozga, a nakon treninga istrenirani model ćemo unijeti u Unity projekt te koristiti ga pomoću unutarnje vrste mozga.

Klasa mozak ima nekoliko glavnih svojstava koje se mogu namjestiti unutar inspekcijskog prozora u Unityju. Ta svojstva moraju odgovarati agentu koji koristi taj mozak. Na primjer veličina vektora opservacijskog prostora (engl. *Vector Observation Space Size*) mora odgovarati dužini vektora opservacija tog agenta.

4.2. Odluke

Opservacijsko-akcijsko-nagrađivački ciklus ponavlja se nakon promjenjivog broja simuliranih koraka (početna ne promijenjena vrijednost je jednom po koraku). Moguće je namjestiti agenta da stvara odluku na zahtjev. Donošenje odluka regularnim intervalima koraka je generalno najprikladnije za simulacije temeljene na fizici. Stvaranje odluka na zahtjev je generalno korisnije kod situacija gdje agenti reagiraju na specifične situacije ili kada reagiraju samo na određeno vrijeme.

Da bismo kontrolirali frekvenciju koraka stvaranja odluka, moramo namjestiti vrijednost frekvencije odluka (engl. *decision frequency*) za agenta unutar Unity inspekcijskog prozora. Agenti koji koriste mozak isti mozak mogu koristiti različite frekvencije. Kroz simulacijske korake, u kojima nije zahtijevana ni jedna odluka, agent prima istu akciju odabranu prošlom odlukom.

4.3. Opservacije

Da bi donio odluku, agent mora razmotriti svoje okruženje da bi odredio svoje trenutno stanje. Opservacija stanja može preuzeti tipove forme:

• Kontinuirani Vektor (engl. *Continuous Vector*) – vektor koji se sastoji od niza brojeva.

• Diskretni Vektor (engl. *Discrete Vector*) – indeks u tablici stanja (pretežito vrlo koristan kod jednostavnih okolina).

• Vizualne Opservacije (engl. *Visual Observations*) – jedna ili više kamera.

Korištenjem kontinuiranog i diskretnog vektora opservacijskog prostora agenta, potrebno je implementirati funkciju Agent.CollectObservations() koja stvara vektor stanja. Kada se koriste vizualne opservacije, potrebno je samo identificirati koja Unity kamera će prosljeđivati slike a Agent klasa će se pobrinuti za ostalo. Nije potrebno implementirati funkciju CollectObservations() ako agent koristi vizualnu opservaciju (osim ako uz nju koristi i bilo koju vektorsku opservaciju).

4.3.1. Kontinuirani vektorski opservacijski prostor

Za agente koji koriste kontinuiranu vrstu stanja prostora, potrebno je stvoriti vektor koji će reprezentirati agentove opservacije na svakom koraku simulacije. Klasa mozak poziva funkciju CollectObservations() svakog svog agenta. Svaka implementacija ove funkcije mora pozivati funkciju AddVectorObs kojom dodaje opservaciju u vektor.

Opservacijama moramo uključiti sve informacije potrebne agentu da izvrši svoj zadatak. Bez preciznih, relevantnih i potrebnih informacija, agent će učiti vrlo sporo ili neće naučiti uopće. Razuman pristup određivanju informacija koje bi trebale biti uključene je taj da razmislimo što je to potrebno da se izračuna analitičko rješenje ovog problema.

Vektor mora uvijek sadržavati isti broj elemenata te opservacije uvijek moraju biti na istom mjestu u nizu. Ako imamo okolinu u kojoj će neke informacije u određenim situacijama nedostajati, uvijek možemo umjesto njih upisati nulu, bitno je samo da polje ne ostane prazno. Isto tako možemo se koristiti metodom u kojoj na primjer nećemo provjeravati stanje svih protivnika u okolini, nego samo najbližih pet.

Kada namještamo mozak agenta unutar Unity editora, potrebno je namjestiti sljedeća svojstva kako bi se koristila kontinuirana vektorska opservacija:

- Veličina Prostora (engl. *Space Size*) – Veličina prostora mora se podudarati sa veličinom našeg vektora opservacija.
- Vrsta Prostora (engl. *Space Type*) – Vrstu prostora potrebno je namjestiti na kontinuiranu (engl. *Continuous*).
- Vrsta Mozga (engl. *Brain Type*) – Vrstu mozga potrebno je namjestiti na vanjsku (engl. *External*) tijekom treninga, a na unutarnju (engl. *Internal*) da bismo koristili istreniran model.

Vektor koji sadrži opservacije je obična lista decimalnih (float) brojeva. To znači da prilikom unosa neke opservacije, bez obzira na njenu vrstu podatka, moramo ju pretvoriti u decimalni broj. Cijeli brojevi se mogu dodati direktno u opservacijski vektor. Izričito je potrebno pretvoriti Boolean varijable u broj. To možemo postići pretvaranjem istine u 1, a laži u 0 ili obrnuto.

Kod unosa entiteta kao što su pozicije i rotacija, predlaže se unos dijelova njihovih komponenata individualno. Na primjer prvo dodamo poziciju na x osi, y osi pa zatim z, a ne cijeli vektor pozicije odjednom.

Kod unosa vrste podataka koji ovise o tipu ili su unikatni elementi, potrebno je svakome elementu pridodati određeni broj koji će ga označivati. Na primjer ako je potrebno u vektor unijeti opservaciju o tome koje oružje agent trenutno posjeduje, svako oružje će imati svoj unikatni broj te će se taj broj predavati kao opservacija.

Normalizacija

Za nabolje rezultate prilikom treniranja, potrebno je normalizirati komponente vektora unutar raspona od -1 do 1 ili od 0 do 1. Kada normaliziramo vrijednosti, PPO neuronska mreža (živčana mreža) može puno brže doći do određenih zaključaka. Naravno nije uvijek potrebno normalizirati na predložene raspone, ali se smatra kao najboljom praksom kod korištenja neuronskih mreža. Što je veći razmak između granica kod komponenata koje razmatramo, veća je šansa da će to utjecati na trening.

Postizanje normalizacije u rasponima između [0, 1] možemo se koristiti sljedećom formulom:

NormalVrijednost = (TrenutnaVrijednost - minVrijednost) / (maxVrijednost - minVrijednost)

Rotacije i kutovi svakako bi trebali biti normalizirani. Za kutova između 0 i 360 stupnjeva, mogu se koristiti sljedeće formule:

```
Quaternion rotation = transform.rotation;

Vector3 NormaliziranKut = rotation.eulerAngles / 180.0f - Vector3.one;  // za raspon [-1,1]

Vector3 NormaliziranKut = rotation.eulerAngles / 360.0f;  // za raspon [0,1]
```

4.4. Akcija

Akcija je instrukcija koju zadaje mozak agentu na izvršenje. Akcija je prenesena na agenta kao parametar kada Akademija (engl. *Academy*) pozove funkciju AgentAction(). Vrsta vektora akcijskog prostora može biti kontinuirana ili diskretna. Razlika je u tome što kontinuirana vrsta sadrži decimalne brojeve unutar nekog razmaka za svaku akciju, a diskretna sadrži točno određen broj za neku akciju. Na primjer, kod kontinuirane da bismo se kretali unutar 2D prostora možemo imati dvije akcije, naprijed i nazad kao jedna, a kao druga lijevo i desno odnosno svaka akcija za svoju dimenziju. Kod diskretne vrste trebali bi imati četiri akcije, svaka za jedan smjer, te bi predavao vrijednost od 0 do 4.

Ni mozak, a ni algoritam treniranje ne znaju što vrijednosti samih akcija znače. Algoritam za treniranje nasumično isprobava različite vrijednosti te nakon nekog vremena gleda kolika je nagrada određenih akcija u određenoj situaciji.

4.4.1. Kontinuirani akcijski prostor

Kada agent koristi mozak koji je namješten na kontinuirani akcijski prostor (engl. *the Continuous vector action space*), parametar akcija proslijeđen agentovoj AgentAction() funkciji je u nizu dužine iste kao i mozgov vektor akcijskog prostora. Individualne vrijednosti u nizu imaju značenja ona koja im mi odredimo. Ako dodamo element u niz kao brzinu kretanja agenta, na primjer, proces treninga će naučiti kontrolirati brzinu kroz ovaj parametar.

Pri normalnim postavkama, izlaz koji omogućuje PPO algoritam ograničuje vrijednosti funkcije vectorAction unutar [-1,1]. Ako želimo sami ograničiti izlaz, možemo sljedećom funkcijom:

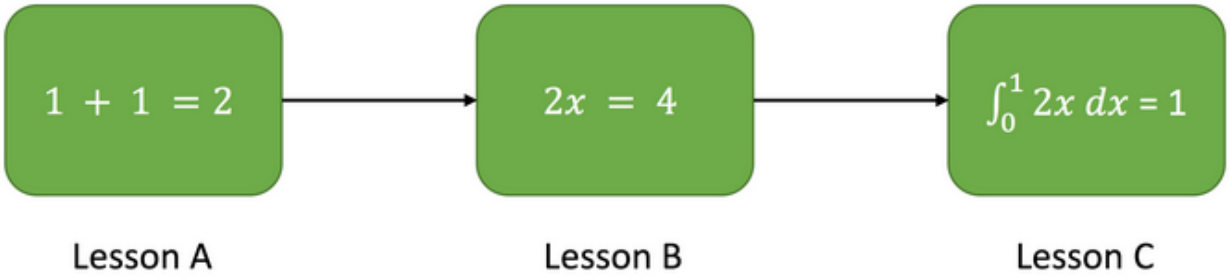
```
Izlaz_1 = Mathf.Clamp(act[1], -1, 1) * 100f;
```

4.5. Nagrade

U podržanom učenju, nagrada je signal koji agentu govori da je obavio nešto dobro. PPO algoritam podržanog učenja radi tako da optimizira agentove odluke tako da konačna nagrada bude veća nego prethodna. Što je bolji mehanizam nagrađivanja, bolje i brže će agent učiti.

Važno je napomenuti da se sistem nagrađivanja ne koristi kod unutarnjeg mozga koji koristi već istrenirani model. Nagrađivanje se koristi samo prilikom treniranja agenta.

Možda najbolji savjet je taj da se krene od vrlo jednostavnih zadataka pa sve do kompleksnijih. Generalno gledano, treba se nagrađivati rezultat, a ne one akcije za koje mi mislimo da bi dovele do određenog stanja. Da bismo razvili naš sustav nagrađivanja, možemo koristiti klasu Monitor koja nam prikazuje kumulativnu nagradu koju dobiva agent. Moguće je namjestiti vrstu mozga na Player te tako sami možemo isprobati određene situacije i gledati kolika će nam nagrada biti za određene odluke.



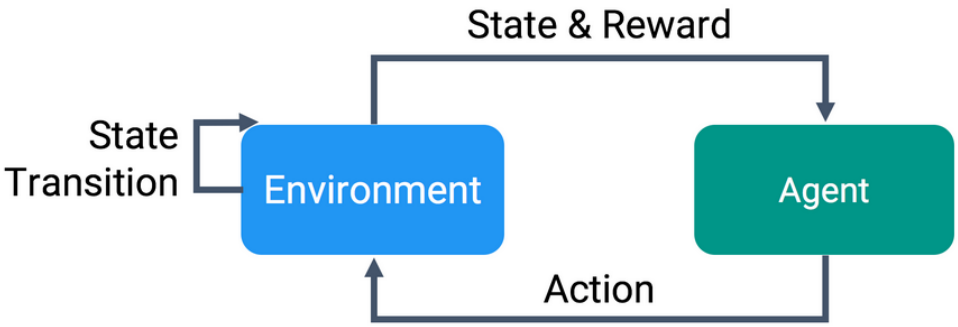
Sl. 4.5.1 Postupno pojačanje zadataka [18]

Dodjeljujemo nagrade agentu pozivanjem funkcije AddReward() unutar funkcije AgentAction(). Dodijeljena nagrada u svakom koraku bi trebala biti negdje u rasponu [-1,1]. Iako je moguće stavljati nagrade izvan raspona, nije preporučeno te će negativno utjecati na treniranje. Nagrada se resetira na nulu nakon svakog koraka.

5. PODRŽANO UČENJE

Podržano učenje (engl. *Reinforcement learning*) je tehnika umjetne inteligencije koja trenira agente da obavljaju zadatke kroz ponašanje koje se može nagraditi. Tijekom podržanog učenja, agent otkriva svoje okruženje, promatra stanje stvari te na temelju tih promatranja donosi odluke. Ako ta odluka vodi prema boljem stanju, agent dobiva pozitivnu nagradu. Suprotno tome ako odluka vodi prema manje željenom (lošijem) stanju, agent dobiva negativnu nagradu ili ne dobiva nagradu uopće. Dok agent uči kroz trening, podržano učenje mijenja njegov način donošenja odluka tako da on kroz neko vrijeme postiže sve veću ukupnu nagradu.

Agenti strojnog učenja koriste tehniku podržanog učenja koja se zove Proksimalna Optimizacija Pravila (engl. *Proximal Policy Optimization*), skraćeno PPO. PPO koristi neuronsku mrežu da približi idealnu funkciju koja povezuje agentove opservacije s najboljim mogućim odlukama (akcijama) u tom stanju. PPO algoritam agenata strojnog učenja implementiran je u TensorFlow i pokreće se odvojeno u Python procesu (komunikacija s pokrenutim Unity aplikacijama odvija se preko podnožja (engl. *soket*)).



Sl. 5.1 Podržano učenje [20]

5.1. Proksimalna Optimizacija Pravila - PPO

Jedan od najnovijih algoritama podržanog učenja je Proksimalna Optimizacija Pravila. Algoritam PPO se usporedno pokazao kao dovoljno dobar ili čak bolji nego ostali suvremeni algoritmi slične upotrebe uz to da je puno jednostavniji za podešavanje i implementiranje. Time je i PPO postao i norma algoritama za podržano učenje OpenAI zbog svoje jednostavnosti i dobre performanse.

Uz nadgledano učenje, možemo vrlo lagano implementirati funkcije koštanja (engl. *cost function*), pokrenuti stupanj spuštanja a svejedno biti vrlo sigurni da ćemo dobiti odlične

rezultate s relativno malim podešavanjima hiperparametara. Naravno, put do uspjeha u podržanom učenju nije toliko očit. Algoritmi sadržavaju vrlo puno pokretnih dijelova koji su teški za otklanjanje neispravnosti (engl. *debuging*) te zahtijevaju znatan napor u namještanju da bi se dobili dobri rezultati. PPO pogađa balans između jednostavnosti postavljanja i implementacije, kompleksnosti primjene, jednostavnosti podešavanja, pokušavajući stvoriti napredak svakim korakom koji će minimizirati funkciju koštanja osiguravajući pri tome razliku u odstupanju od prošlih pravila vrlo malu.

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\right]$$

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

Sl. 4.5.1 Formula PPO algoritma [19]

Potrebno je znati da prilikom treniranja agenata, kroz Unity agente strojnog učenja, nije potrebno znati koristiti PPO algoritam pa čak ni kako on točno radi. Na njega se može gledati kao jednu kutiju za treniranje. Ima ponešto parametara koji su vezani za učenje unutar Unityja te pythonove skripte no nije potrebno ulaziti duboko u znanje o PPO kako bi se agenti mogli uspješno trenirati.

5.2. TensorFlow

Kao što je već prije spomenuto, većina algoritama unutar ML-Agenata sadrže neku vrstu dubokog učenja (engl. *deep learning*). Specifičnije, implementacije su sagrađene na temelju biblioteke otvorenog koda TensorFlow. To znači da modeli proizvedeni kroz ML-Agente su u formatu razumljivom samo TensorFlowu.

TensorFlow je biblioteka otvorenog koda za izvođenje računanja pomoću grafova protoka podataka, to jest temeljni prikaz modela dubokog učenja. TensorFlow olakšava trening i zaključivanje na CPU-ima i GPU-ima unutar računala, servera ili mobilnog uređaja. Unutar ML-Agenata, dok učimo agenta ponašanju, stvaramo izlaz koji je TensorFlow model (s ekstenzijom „ .bytes“) podatka koji se zatim može implementirati u unutarnji mozak agenta. Ako nismo

napisali svoj novi algoritam i implementirali ga, TensorFlow je pretežito apstraktan te radi u pozadini scene. Korištenje TensorFlowa zahtjeva Unity 2017.1 ili noviji te Unity TensorFlow dodatak.

ML-Agenti omogućuju da se koristi već prije istrenirani TensorFlow graf unutar Unity igre. Ovo je moguće uz pomoć TensorFlowSharp projekta. Primarna korist ovog dodatka je ta da se TensorFlow model može trenirati korištenjem ML-Agenata, a sekundarna korist je ta da se može koristiti bilo koji TensorFlow model.

6. POSTAVLJANJE POTREBNIH PROGRAMA

Da bismo instalirali i koristili ML-Agente, potrebno je instalirati Unity, klonirati ML-Agents repozitorij te instalirati Python sa svim potrebnim dodatcima. U nastavku bit će objašnjeni koraci instalacije svakog potrebnog programa [15].

Kao prvi korak potrebno je skinuti najnoviju ili bilo koju verziju 2017 ili noviju Unityja. Verzije možemo pogledati i skinuti preko njihove stranice [9]. Kada je instalacija Unityja gotova, potrebno je klonirati ML-Agents GitHub repozitorij. Možemo ga klonirati sljedećom naredbom ili ručno s iste poveznice:

git clone https://github.com/Unity-Technologies/ml-agents.git

Direktorij unity-environment unutar ovog repozitorija sadrži sve Unity Assete potrebne za ML-Agents projekte. Direktorij python sadrži trening kod. Oba direktorija nalaze se u korijenu samog repozitorija.

6.1. Instaliranje Pythona

Kako bismo koristili ML-Agente, obavezan je Python 3 zajedno sa dodatnim potrebnim programima koji su navedeni u datoteci zahtjeva. Neki od dodatnih potrebnih programa su TensorFlow i Jupyter. Predlaže se skidanje i instaliranje Anaconde za Windows [13]. Korištenjem Anakonde, moguće je stvoriti i koristiti različito okruženje za različite verzije Pythona. U prethodnim verzijama mogao se koristiti i Python 2, no sada samo Python 3.

U Anakondi potrebno je stvoriti novo Conda okruženje koje će se koristiti uz ML-Agente. To znači da svi paketi koje instaliramo bit će povezani samo za ovo okruženje koje smo napravili. Bilo kaka instalacija neće utjecati na Python ili na druga okruženja. Prije svakog korištenja ML-Agenta, potrebno je aktiviranje stvorenog Conda okruženja.

Stvaranje Conda okruženja se postiže na sljedeći način. Prvo je potrebno otvoriti Anaconda Prompt koji smo instalirali, a zatim u njega napisati sljedeću naredbu:

conda create -n tfp3.5 python=3.5

Gdje je „tfp3.5“ naziv našeg okruženja, a python=3.5 određuje verziju pythona u našem okruženju. Moguće je koristiti i novije verzije Pythona, no ova verzija se ispostavila kao najbolja.

```
(base) C:\aaaaa>conda create -n tfp3.5 python=3.5
Solving environment: done

## Package Plan ##

  environment location: C:\ProgramData\Anaconda3\envs\tfp3.5

  added / updated specs:
    - python=3.5


The following NEW packages will be INSTALLED:

    certifi:        2018.1.18-py35_0
    pip:            9.0.3-py35_0
    python:         3.5.5-h0c2934d_1
    setuptools:     39.0.1-py35_0
    vc:             14-h0510ff6_3
    vs2015_runtime: 14.0.25123-3
    wheel:          0.30.0-py35h38a90bc_1
    wincertstore:   0.2-py35hfebbdb8_0
```

Sl. 6.1.1 Stvaranje Conda okruženja (1)

Zatim postoji mogućnost da dođe pitanje u vezi instaliranja novih paketa. Pritiskom na tipku „y“ pa zatim „enter“ prihvatit ćemo instalaciju paketa i nastaviti instalaciju (paketi su obavezni).

```
Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: -
SafetyError: The package for setuptools located at C:\ProgramData\Anaconda3\pkgs
\setuptools-39.0.1-py35_0
appears to be corrupted. The path 'Scripts/easy_install.exe'
has a sha256 mismatch.
  reported sha256: 993203a406e04936a07829b1f482fd27d739b640482e213f4c49ea1ee78a5
fcf
  actual sha256: 5db9bbc84aeeb447388ddc7103a5ca15184c683fb67c673aa30e894907aeac5
4

SafetyError: The package for wheel located at C:\ProgramData\Anaconda3\pkgs\whee
l-0.30.0-py35h38a90bc_1
appears to be corrupted. The path 'Scripts/wheel.exe'
has a sha256 mismatch.
  reported sha256: 993203a406e04936a07829b1f482fd27d739b640482e213f4c49ea1ee78a5
fcf
  actual sha256: ba17fec1d195753fde579cff262d8b127ed1f871ecb9f6dce6682db57f9f2f7
4



done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate tfp3.5
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

Sl. 6.1.2 Stvaranje Conda okruženja (2)

Naredbom conda activate tfp3.5 ulazimo u stvoreno okruženje, a naredbom conda deactivate izlazimo iz njega. Potrebno je biti spojen na Internet te sljedećim naredbama instalirati ML-Agente:

conda activate tfp3.5

git clone https://github.com/Unity-Technologies/ml-agents.git

cd C:\Direktorij_gdje_smo_klonirali_MLAgente\ml-agents\python

pip install .

Pip će zatim instalirati sve što je potrebno da se ML-Agenti mogu koristiti unutar ovog okruženja. Naravno uz dodatke Pythonu potreban je i TensorFlow, no o tome nešto kasnije.

6.2. Instaliranje Nvidia CUDA alata

Iako je moguće, za treniranje ML-Agenata nije potrebno koristiti GPU zato što se PPO algoritam svejedno neće puno ubrzati (no u budućnosti cilja se na treniranje pomoću GPU). U slučaju da se ne koristi treniranje pomoću GPU, ovo poglavlje se može preskočiti. Isto tako ako se koristi GPU treniranje, potrebno je provjeriti ako je grafička kartica koju planiramo koristiti na listi kompatibilnih [16]. Uvođenjem verzije ML-Agents v0.4, samo CUDA v9.0 i cuDNN v7.0.5 su podržane [17]. Nakon skidanja instalera, potrebno je instalirati alat CUDA koji sadrži GPU-bazirane biblioteke, alate za otklon greški i optimiziranje, C++ kompajler i sve potrebno da pokreće ML-Agente.

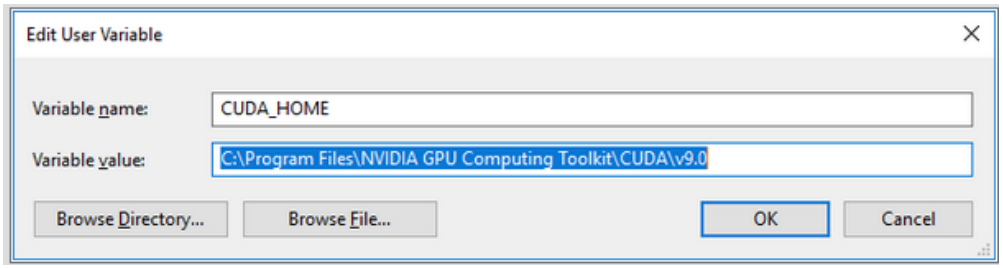
6.3. Instaliranje Nvidia cuDNN biblioteke

Ako se koristi GPU treniranje potrebno je skinuti i instalirati cuDNN biblioteku koju je napisala Nvidia [14]. Naravno, ako se ne koristi GPU treniranje, kao i prošlo poglavlje i ovo se može preskočiti. cuDNN je biblioteka koja služi za ubrzanje GPU prilikom rada s dubokim neuronskim mrežama. Prije skidanja same biblioteke, potrebno se registrirati na stranicu Nvidia Developer Program.

Potrebno je pripaziti, prilikom skidanja, na verziju biblioteke te na njenu kompatibilnost s CUDA alatom koji je instaliran. Nakon skidanja cuDNN podataka, potrebno je izvući podatke i staviti ih unutar CUDA direktorija. Potrebno je kopirati sva tri direktorija koji se nalaze u zip-u (bin, include i lib).

6.4. Namještanje varijabla okruženja

Kako bi omogućili računalu da pronalazi potrebne datoteke prilikom računala, trebamo napraviti jednu varijablu okruženja te dvije varijable puta. Da bismo dodali varijable okruženja, upisujemo „environment variables“ unutar tražilice Windowsa. Trebali bismo vidjeti opciju koja se zove „Edit the system environment variables“. Odabiranjem te opcije, dalje idemo na tipku „Environment Variables“. Klikom na „New“ dodajemo novu sistemsku varijablu. Za ime varijable potrebno je upisati „CUDA_HOME“, a za vrijednost treba upisati lokaciju direktorija CUDA alata.

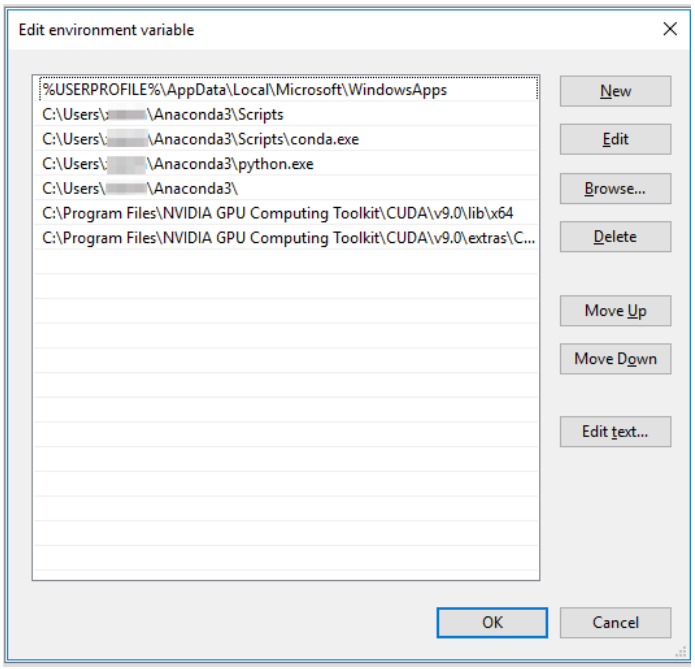


Sl. 6.4.1 Stvaranje nove varijable okruženja [15]

Dvije varijable puta postavljamo unutar istog „Environment Variables“ prozora, unutar druge „kutije“ koja se zove „System Variables“. Potrebno je pronaći varijablu koja se zove „Path“, označiti ju i kliknuti „Edit“. U otvorenu listu treba dodati dva direktorija. Ovisno o korištenoj verziji i mjestu instaliranja, otprilike izgledaju ovako:

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\lib\x64

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\extras\CUPTI\libx64



Sl. 6.4.2 Dodavanje puta u sistemske varijable [15]

6.5. Install TensorFlow GPU

Na kraju u oba slučaja, GPU i CPU treniranja, potrebno je instalirati TensorFlow koristeći pip. U slučaju da imamo instaliranu krivu verziju, možemo ju deinstalirati pomoću naredbe:

pip uninstall tensorflow

Ako koristimo GPU verziju treniranja TensorFlow instaliramo na sljedeći način. Potrebno je biti spojen na internet te je potrebno uključiti naše Conda okruženje. Zatim upisujemo sljedeću naredbu kako bi instalirali TensorFlow verzije 1.7.1:

pip install tensorflow-gpu==1.7.1

Kako bismo provjerili da li je TensorFlow dobro instaliran te dali on može identificirati GPU, potrebno je upisati unutar otvorenog Anaconda Prompta sljedeće:

import tensorflow as tf

sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))

Zatim bismo trebali dobiti nešto otprilike ovako „Found device 0 with properties ...“. Što bi značilo da je dobro instaliran.

Ako ipak koristimo GPU treniranje potrebno je instalirati TensorFlow na sljedeći način. Potrebno je biti spojen na internet te je potrebno uključiti naše Conda okruženje. Zatim upisujemo sljedeću naredbu kako bi instalirali TensorFlow verzije 1.7.1:

pip install tensorflow==1.7.1

U slučaju da posljednja verzija TensorFlowa ne radi kako treba, moguće je instalirati neku od prethodnih verzija. Na primjer može se u naredbu dodati poveznica odakle da pip instalira TensorFlow, kao na slici Sl 6.5.1.

```
(tfp3.5) C:\aaaaa>pip install https://storage.googleapis.com/tensorflow/windows/
cpu/tensorflow-0.12.0rc0-cp35-cp35m-win_amd64.whl
Collecting tensorflow==0.12.0rc0 from https://storage.googleapis.com/tensorflow/
windows/cpu/tensorflow-0.12.0rc0-cp35-cp35m-win_amd64.whl
  Downloading https://storage.googleapis.com/tensorflow/windows/cpu/tensorflow-0
.12.0rc0-cp35-cp35m-win_amd64.whl (12.2MB)
    100% |################################| 12.2MB 74kB/s
Requirement already satisfied: wheel>=0.26 in c:\programdata\anaconda3\envs\tfp3
.5\lib\site-packages (from tensorflow==0.12.0rc0)
Collecting six>=1.10.0 (from tensorflow==0.12.0rc0)
  Using cached six-1.11.0-py2.py3-none-any.whl
Collecting numpy>=1.11.0 (from tensorflow==0.12.0rc0)
  Using cached numpy-1.14.2-cp35-none-win_amd64.whl
Collecting protobuf==3.1.0 (from tensorflow==0.12.0rc0)
  Downloading protobuf-3.1.0-py2.py3-none-any.whl (339kB)
    100% |################################| 348kB 1.0MB/s
Requirement already satisfied: setuptools in c:\programdata\anaconda3\envs\tfp3.
5\lib\site-packages (from protobuf==3.1.0->tensorflow==0.12.0rc0)
Installing collected packages: six, numpy, protobuf, tensorflow
Successfully installed numpy-1.14.2 protobuf-3.1.0 six-1.11.0 tensorflow-0.12.0r
c0
```

Sl. 6.5.1 Instaliranje CPU TensorFlowa

Provjeru kod ovog načina izvodimo kao na slici Sl 6.5.2, naredbama:

python

import tensorflow as tf

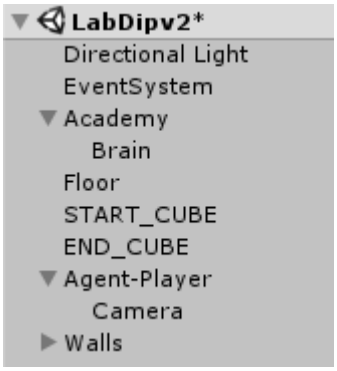
Nakon izvršenja naredbi, ne bismo trebali dobiti nikakvu grešku.

```
(tfp3.5) C:\aaaaa>python
Python 3.5.5 |Anaconda, Inc.| (default, Mar 12 2018, 17:44:09) [MSC v.1900 64 bi
t (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>>
```

Sl. 6.5.2 Provjera CPU TensorFlowa

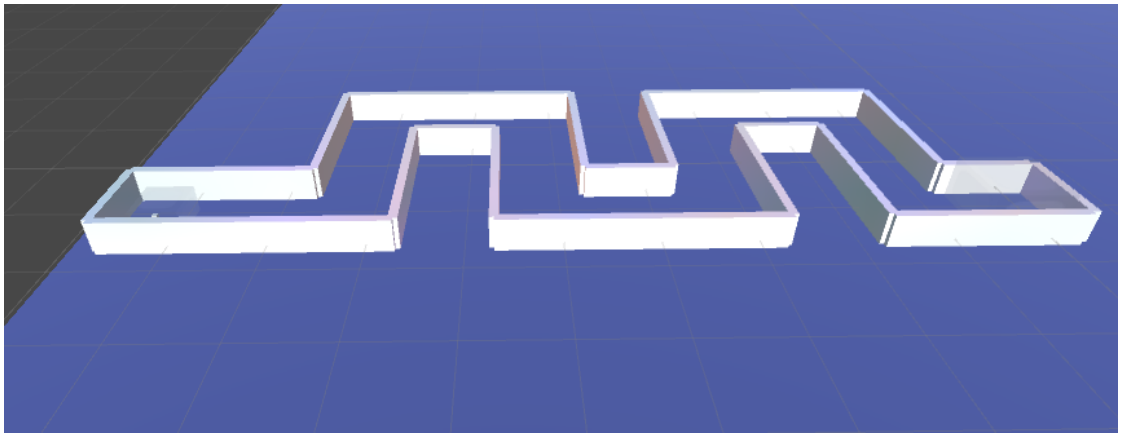
7. IZRADA IGRE U UNITY 3D

Zbog načina na koji funkcionira treniranje, potrebno napraviti dvije različite igre. Jedna će biti korištena samo za treniranje a druga će koristiti već istreniran mozak (opisana pod poglavljem „9.1 Zadnja verzija igre“). Ideja Unity igre za treniranje je jednostavan labirint, odnosno put okružen zidovima kroz koje agent mora sam proći. Igra je napravljena najjednostavnije moguće kako bi proces treniranja išao što brže. Igra se sastoji od objekta poda, zidova, početne točke, završne točke i agenta uz kojeg je obavezno imati akademiji i mozak.



Sl. 7.1 Hijerarhija scene

Objekt pod i objekt zid, jednostavni su 3D objekti koji sadržavaju „Box Collider“ i „Rigidbody“ koji omogućuju da agent ne propadne u pod te ne prolazi kroz zid. Jedina razlika je ta što ako se agent sudari s objektom zida, kreče od početka staze. Izgled zidova i poda možemo vidjeti na slici Sl 7.2.

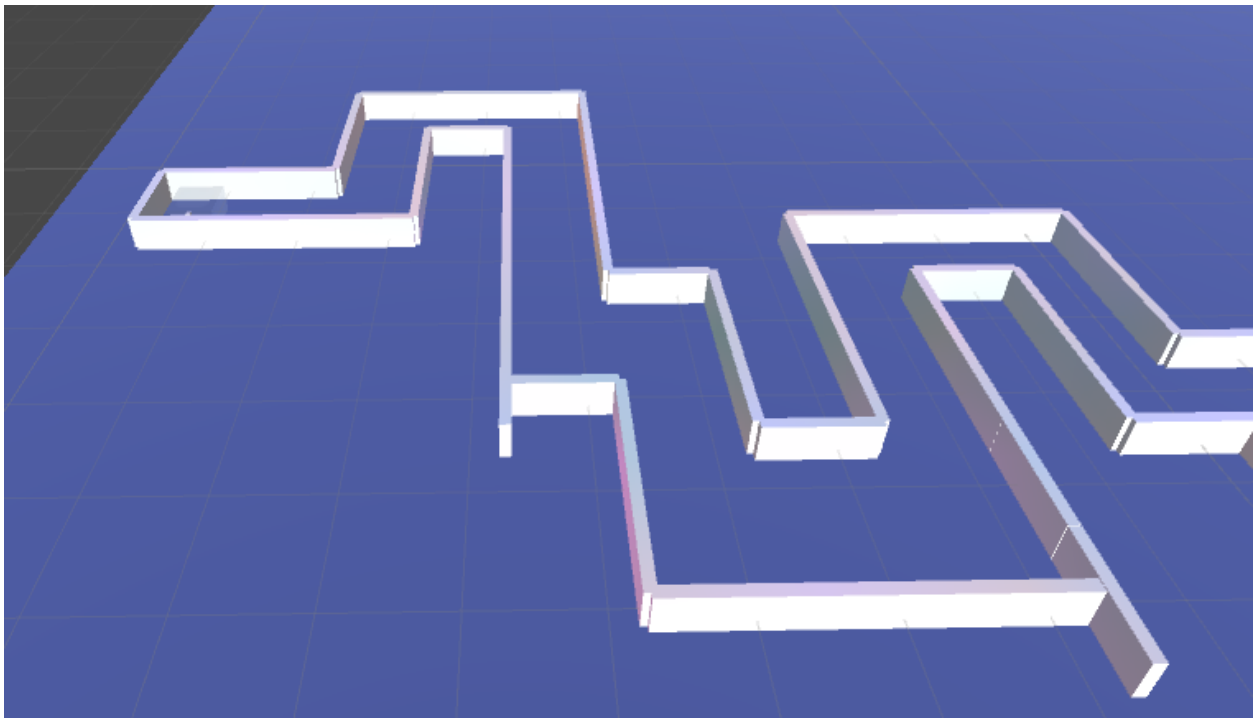


Sl. 7.2 Prva razina staze

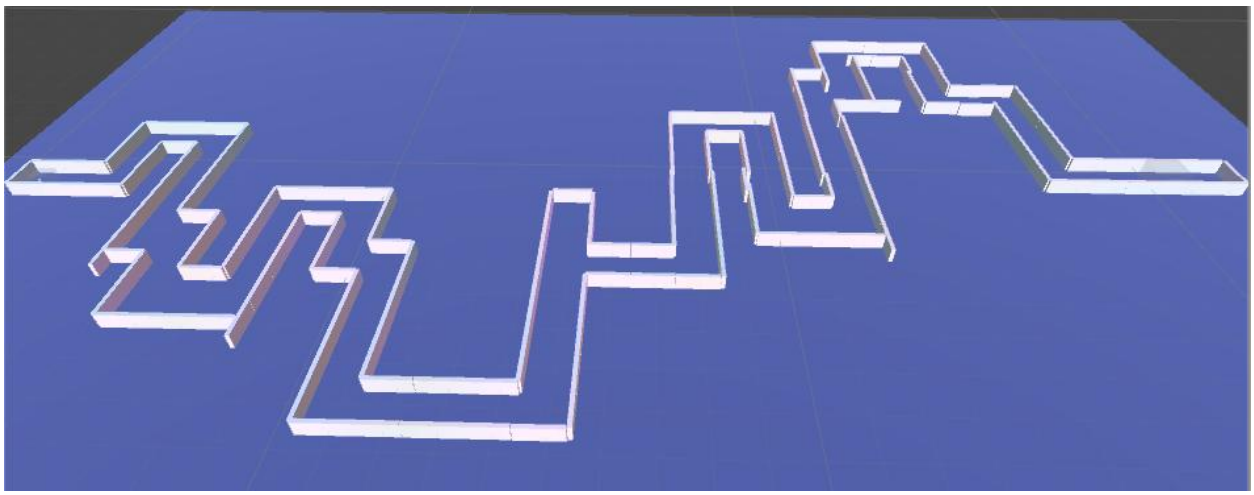
Početna točka i završna točka su 3D objekti koji naravno označavaju početak staze te cilj, odnosno kraj staze. Početna točka ne sadrži ni „Rigidbody“ ni „Box Collider“, ona služi samo za početnu lokaciju gdje se agent stvori, na početku igre te svakog sudara s nekim objektom (zidom

ili ciljem). Početna točka nalazi se u sloju (engl. *layer*) „TransparentFX“ što omogućuje da ju agent uopće ne vidi te ga ona ne ometa u njegovoj opservaciji okruženja. Nasuprot tome završna točka ima „Box Collider“ i „Rigidbody“ što omogućava agentu da zna gdje se ona nalazi te da se može s njome sudariti, odnosno doći u kontakt. Jednom kada agent dođe u kontakt sa završnom točkom, stvara se natrag na početku staze.

Ranije je spomenuto da je bitno agenta učiti postepeno. Iz tog razloga prva staza na kojoj je agent trenirao izgledala je kao na slici Sl. 7.2. Gdje je agent učio osnove. Naučio je što su ulazi u neuronsku mrežu koja ga kontrolira te što su izlazi. Zatim Agentov trening prelazi na nešto težu stazu, prikazanu na slici Sl. 7.3, a kasnije i na još težu stazu prikazanu slikom Sl. 7.4.



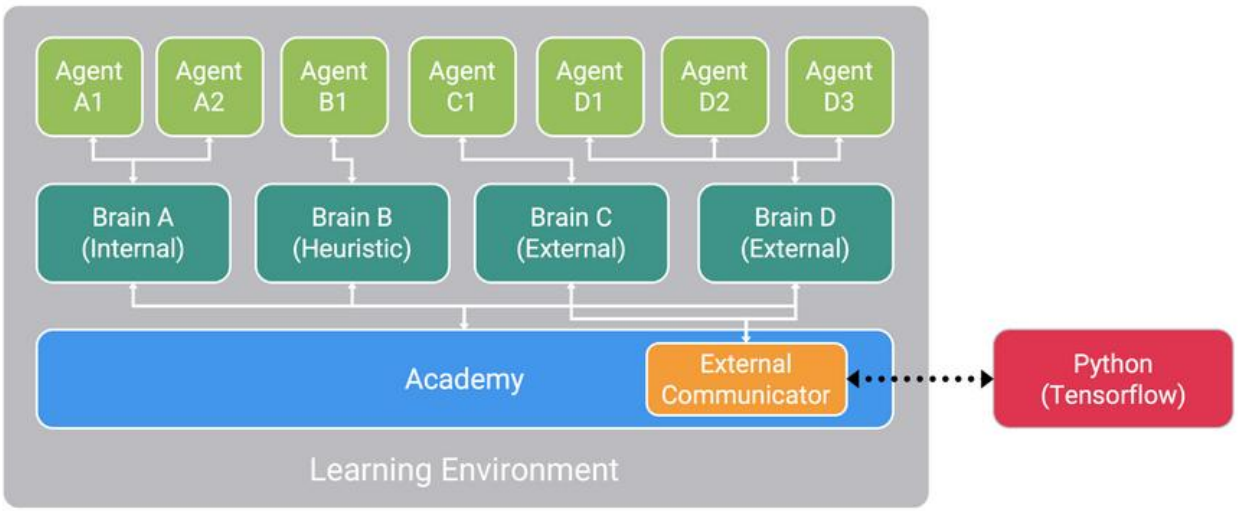
Sl. 7.3 Druga razina staze



Sl. 7.4 Treća razina staze

Da bismo trenirali i koristili ML-Agente u Unity sceni, potrebno je imati jednu Academy potklasu uz bilo koliko objekata mozga i bilo koliko Agent potklasa. Svaki mozak korišten u sceni mora biti spojen na „GameObject“ koji je dijete akademije unutar Unity hijerarhije scene. Agenti koji se instanciraju trebaju biti spojeni na „GameObject“ koji predstavlja tog agenta.

Potrebno je spojiti mozak svakome agentu. Agenti mogu dijeliti mozgove, odnosno moguće je isti mozak dati svakome agentu. Bez obzira na dodjeljivanje mozgova agentima, svaki agent će donositi svoje vlastite opservacije te donositi drugačije odluke, no ako se koristi ista vrsta mozga na više agenata, koristit će istu logiku donošenja odluka. Isto tako kod unutarnjih mozgova koristi će isti istrenirani TensorFlow model.



Sl. 7.5 Treniranje sa više agenata i više mozgova [20]

7.1. Akademija

Objekt akademija orkestrira i organizira agente i njihove procese donošenja odluke. Kao što je ranije spomenuto, samo jedan objekt akademije treba biti u sceni.

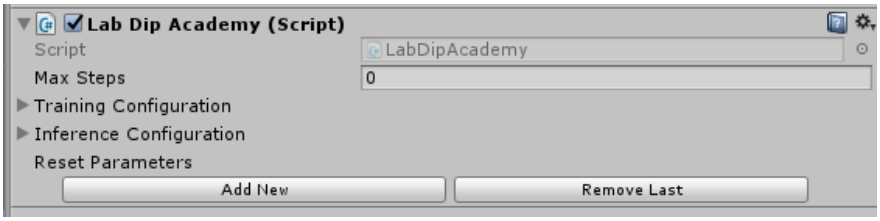
S obzirom na to da je klasa Academy apstraktna, potrebno je dodati potklasu te klase. Nakon stvaranja potklase možemo implementirati sljedeće funkcije:

• InitializeAcademy() – Priprema okruženje prvi put kada se igra uključi.

• AcademyReset() – Priprema okruženje i agente za sljedeću epizodu treniranja.

• AcademyStep() – Priprema okruženje za sljedeći simulacijski korak. Ova se funkcija poziva prije svake funkcije AgentAction() trenutnog koraka.

Bazna Academy klasa definira nekoliko bitnih svojstava koje se mogu namjestiti kroz Unity Editor Inspector. Za treniranje najbitnija je „Max Steps“, koja određuje koliko će svaka epizoda treniranja trajati. Jednom kada brojač koraka akademije dođe do namještene vrijednosti, poziva funkciju AcademyReset() kako bi krenuo s novom epizodom.



Sl. 7.1.1 Postavke objekta akademije

```
public class LabDipAcademy : Academy {

    // Use this for initialization
    public override void AcademyReset() {

    }

    // Update is called once per frame
    public override void AcademyStep() {

    }
}
```

Sl. 7.1.2 Skripta koju koristi akademija

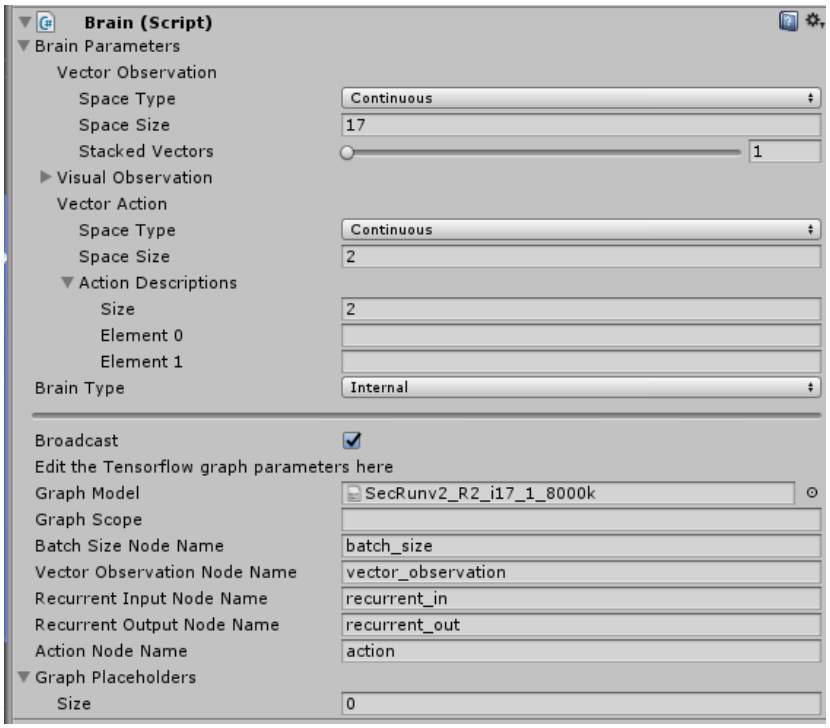
Zbog jednostavnosti treniranja i zadatka, nije bilo potrebno unositi dodatne funkcije u skriptu akademije, kao što je prikazano na slici Sl 7.1.2.

7.2. Mozak

Mozak oblaže proces donošenja odluka. Kao što je prikazano na slici Sl. 7.1, mozak je dijete akademije. Klasa Brain koristi se direktno, a ne kao potklasa. Ponašanje mozga ovisi o njegovoj vrsti, kao što je već prije spomenuto.

Klasa Brain ima par vrlo bitnih svojstava koja se mogu namjestiti kroz inspekcijski prozor, prikazan na slici Sl. 7.2.1. Svojstva je potrebno namjestiti da odgovaraju agentu koji ga koristi.

Svojstva mozga podijeljena su u dva djela. Prvi dio je konstantan bez obzira na vrstu mozga koja se koristi, a drugi ovisi o vrsti mozga. Parametar „Vector Observation“ (hr. vektor opservacija) predstavlja agentov vektor opservacija te mu veličina prostora mora biti jednaka broju opservacija koje agent predaje. Slično tome, „Vector Action“ (hr. vektor akcija) mora biti iste veličine kao i broj akcija koje agent može primiti i izvršiti.



Sl. 7.2.1 Svojstva mozga

Iz slike Sl. 7.2.1 možemo vidjeti da je vrsta mozga namještena na „Internal“ (hr. unutarnji), što znači već imamo istrenirani model kojeg koristimo. U slučaju kada se koristi vanjski mozak, nije potrebno unositi nikakva dodatna svojstva. Pod svojstvom „Graph Model“ dodajemo TensorFlow istrenirani model kojeg želimo koristiti.

7.3. Agent

Klasa Agent predstavlja glumca u sceni koji prikuplja opservacije i odrađuje akcije. Ta klasa je pretežito spojena na „GameObject“ koji u sceni predstavlja tog glumca. Svaki agent mora imati mozak.

Da bismo stvorili agenta, potrebno je produžiti klasu Agent i implementirati osnovne, potrebne funkcije u skriptu, kao što su：

- CollectObservations() – Prikuplja agentove opservacije okruženja.
- AgentAction() – Izvršava akciju koju mu je odredio agentov mozak te dodaje nagradu ovisno o trenutnom stanju.

Implementacija ovih funkcija određuje kako treba namjestiti svojstva mozga ovog agenta. Vrlo bitno je namjestiti kada je agent gotov sa zadatkom ili vremenski ograničiti njegovo ponašanje. Uvijek se može ručno namjestiti agentovo završavanje zadatka u funkciji AgentAction().

```
void Start()
{
    timeLeft = 110.0f;
    //CAMERA
    isLocalPlayer = true;
    if (isLocalPlayer) {
        mainCamera.gameObject.SetActive (false);
    }else {
        mainCamera.GetComponentInChildren<Camera>().gameObject.SetActive(false);
    }


    StartCube = GameObject.FindGameObjectWithTag("Respawn");
    EndCube = GameObject.FindGameObjectWithTag("goal");

    // Starting position
    playerStartDot = StartCube.transform.position;
    playerLastPos = playerStartDot;
    playerStartDotRotation = StartCube.transform.rotation;
    gameObject.transform.position = playerStartDot;
    gameObject.transform.rotation = playerStartDotRotation;
    // Starting speed
    gameObject.GetComponent<Rigidbody>().velocity = new Vector3(0f, 0f, 0f);
    distanceLast = Vector3.Distance(transform.position,EndCube.transform.position);
}
```

Sl. 7.3.1 Funkcija Start()

Agenta na početku igre, pomoću funkcije Start(), stvaramo na poziciju objekta početne točke („StartCube“). Agentu brzinu stavljamo na 0 te rotaciju prema smjeru u kojem se treba kretati, kako se ne bi zabio u zid istog trena. Postavljamo početno vrijeme odbrojavanja na 110 sekundi, kako se ne bi vrtio u beskonačnoj petlji kod treniranja.

```
public override void AgentAction(float[] vectorAction, string textAction)
{
    if (brain.brainParameters.vectorActionSpaceType == SpaceType.continuous)
    {
        //Moved in the right direction:
        float a1 = gameObject.transform.position.x;
        if (gameObject.transform.position.x < playerLastPos.x) {
            AddReward(0.25f);
        } else {
            AddReward(-0.4f);
        }
        playerLastPos = gameObject.transform.position;
```

Sl. 7.3.2 Funkcija AgentAction() (1)

Kao što je ranije spomenuto, funkcijom AgentAction() izvršavamo akcije kroz agenta te ocjenjujemo odluku mozga kroz sustav nagrađivanja. Iz slike Sl. 7.3.2 možemo vidjeti da se nagrađuje kretanje agenta u pravom smjeru, a negativno nagrađuje ako ide u krivom ili ako se nije pomaknuo uopće.

```
// INPUT and CLAMPING
float action_walk = 2f * Mathf.Clamp(vectorAction[0], 0f, 5f);
float action_rotate_y = 2f * Mathf.Clamp(vectorAction[1], -1f, 1f);
```

Sl. 7.3.3 Funkcija AgentAction() (2)

Na slici Sl. 7.3.3 možemo vidjeti moguće akcije koje agent izvršava. Akcijski vektor sastoji se od dva broja u nizu. Prvi broj govori o tome koliko će se kretati agent. Drugi broj govori u koju stranu će se agent okrenuti. Kretanje je moguće u intervalu [0, 5] što znači da agent može stajati u mjestu ili se kretati naprijed. Rotiranje, odnosno okretanje u stranu je moguće od -1 (lijevo) do 1 (desno).

```
// FRONT SIDE
Physics.Raycast(transform.position, transform.TransformDirection(Vector3.forward), out hitfINF, Mathf.Infinity);
RayTest = Physics.Raycast(transform.position, transform.TransformDirection(Vector3.forward), out hitf, 5f);
if (hitfINF.collider != null && hitfINF.collider.tag == "goal"){
    AddReward(1f);
    AddReward(1f);
} else if (RayTest) {
    AddReward(-0.6f);
} else {
    AddReward(0.9f);
}

// LEFT SIDE
RayTest = Physics.Raycast(transform.position, transform.TransformDirection(Vector3.left), out hitl, 3f);
if (hitl.collider != null && hitl.collider.tag == "goal") {
    AddReward(-0.1f);
} else if (RayTest) {
    AddReward(-0.2f);
} else {
    AddReward(0.4f);
}

// FRONT LEFT
Physics.Raycast(transform.position, transform.TransformDirection(new Vector3(-1, 0, 1)), out hitfl, 4f);


// RIGHT SIDE
RayTest = Physics.Raycast(transform.position, transform.TransformDirection(Vector3.right), out hitr, 3f);
if (hitr.collider != null && hitr.collider.tag == "goal") {
    AddReward(-0.1f);
} else if (RayTest) {
    AddReward(-0.2f);  //-4
} else {
    AddReward(0.4f);  //5
}

// FRONT RIGHT
Physics.Raycast(transform.position, transform.TransformDirection(new Vector3(1, 0, 1)), out hitfr, 4f);
```

Sl. 7.3.4 Funkcija AgentAction() (3)

Potrebno je agentu dati nagradu ako se kreće prema krajnjoj točki, u slučaju kada ju on sam i vidi. Nagrađivanje, negativno ili pozitivno, prikazano je na slici Sl. 7.3.4.

```
// No rotation, going forward
if (hitf.collider != null && hitf.collider.tag == "goal" && action_rotate_y != 0f) {
    AddReward (-1f);
    AddReward (-0.5f);
} else if (hitf.collider != null && hitf.collider.tag == "goal" && action_rotate_y == 0f) {
    AddReward (1f);
    AddReward (1f);
    AddReward (1f);
}
```

Sl. 7.3.5 Funkcija AgentAction() (4)

Kod rotacije bitno je provjeriti da li je krajnja točka ispred agenta ili možda zid. Ako je krajnja točka ispred, bilo kakvo rotiranje donosi negativnu nagradu. Suprotno tome, ako je krajnja točka ispred a rotacije nema, agent dobiva pozitivnu nagradu (kao u slici Sl. 7.3.5).

```
// Rotating
if (action_rotate_y != 0f) {
    if ((hitfl.collider != null && hitfl.collider.tag == "goal") && (hitfr.collider != null && hitfr.collider.tag == "goal")
        && (hitf.collider != null && hitf.collider.tag == "goal")){
        AddReward (-1f);
    }
    // TO RIGHT SIDE
    else if (action_rotate_y > 0f) {
        if ((hitr.collider != null && hitr.collider.tag != "goal") && (hitfr.collider != null && hitr.collider.tag == "goal")){
            AddReward (1f);
            AddReward (1f);
        }
        else if (hitr.collider != null && hitr.collider.tag == "goal"){
            AddReward (1f); //1.7
            //AddReward (0.5f);
        }
        else if ((hitl.collider != null && hitl.collider.tag == "goal") || (hitfl.collider != null && hitfl.collider.tag == "goal")){
            AddReward (-1f);
            AddReward (-1f); //skrece desno a lijevo je goal
        }
        else if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.left), 4f) == true
            && Physics.Raycast (transform.position, transform.TransformDirection (new Vector3 (-1, 0, 1)), 5f) == true
            && Physics.Raycast (transform.position, transform.TransformDirection (Vector3.forward), 3f) == true) {
            AddReward (0.8f);
            //Debug.Log ("NEMOZE LIJEVO PA SKRECE DESNO");
        }
        else if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.right), 4f) == true
            && Physics.Raycast (transform.position, transform.TransformDirection (new Vector3 (1, 0, 1)), 5f) == true) {
            AddReward (-0.8f);
            AddReward (-0.4f);
            //Debug.Log ("DIRA DESNO I SKRECE DESNO");
        }
        else if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.right), 5f) == true
            && Physics.Raycast (transform.position, transform.TransformDirection (new Vector3(1, 0, 1)), 5f) == false){
            AddReward(0.8f);
            //Debug.Log("DIRA SAMO DESNO ++++ SKRECE DESNO");
        }
    }
```

Sl. 7.3.6 Funkcija AgentAction() (5)

```
// TO LEFT SIDE
else if (action_rotate_y < 0f) {
    if ((hitl.collider != null && hitl.collider.tag != "goal") && (hitfl.collider != null && hitl.collider.tag == "goal")){
        AddReward (1f);
        AddReward (1f); //2.2
    }
    else if (hitl.collider != null && hitl.collider.tag == "goal"){
        AddReward (1f); //1.7
    }
    else if ((hitr.collider != null && hitr.collider.tag == "goal") || (hitfr.collider != null && hitfr.collider.tag == "goal")){
        AddReward (-1f);
        AddReward (-1f); //skrece lijevo a desno je goal
    }
    else if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.right), 4f) == true
        && Physics.Raycast (transform.position, transform.TransformDirection (new Vector3 (1, 0, 1)), 5f) == true
        && Physics.Raycast (transform.position, transform.TransformDirection (Vector3.forward), 3f) == true) {
            AddReward (0.8f);
            //Debug.Log ("NEMOZE DESNO PA SKRECE LIJEVO");
    }
    else if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.left), 4f) == true
        && Physics.Raycast (transform.position, transform.TransformDirection (new Vector3(-1, 0, 1)), 5f) == true){
        AddReward (-0.8f);
        AddReward (-0.4f);
        //Debug.Log("DIRA LIJEVO I SKRECE LIJEVO");
    }
    else if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.left), 5f) == true
        && Physics.Raycast (transform.position, transform.TransformDirection (new Vector3(-1, 0, 1)), 5f) == false){
        AddReward(0.8f);
        //Debug.Log("DIRA SAMO LIJEVO ++++ SKRECE LIJEVO");
    }
}
```

Sl. 7.3.7 Funkcija AgentAction() (6)

Kod rotiranja, vrlo je bitno provjeriti zašto se rotira. Ako je zid desno od agenta i naprijed-desno, rotacija u desno dobiva negativnu nagradu. Ako je zid desno, a naprijed-desno nije, rotacija se nagrađuje pozitivno. Isto vrijedi i za lijevu stranu, kao što je prikazano u slikama: Sl. 7.3.6 i Sl. 7.3.7.

```
// Walking Forward
if (action_walk > 0f)
{
    //gameObject.transform.position += gameObject.transform.forward * Time.deltaTime * action_walk * movementSpeed;
    AddReward(0.7f);
} // Standing in place, while there is wall in front
else if (action_walk == 0f && Physics.Raycast (transform.position, transform.TransformDirection (Vector3.forward), 3f) == true )
{
    AddReward(0.05f);
} // Stoped for no reason
else
{
    AddReward(-0.7f);
}
```

Sl. 7.3.8 Funkcija AgentAction() (7)

Svako kretanje nagrađuje se pozitivno zato što želimo da agent dođe do kraja staze što prije. Iako agent ima mogućnost stajanja u mjestu, ono donosi negativnu nagradu, osim ako je zid nedaleko ispred agenta (vidimo na slici Sl. 7.3.8).

```
// OUTPUT
gameObject.transform.Rotate(new Vector3(0, 1, 0), action_rotate_y);
gameObject.transform.position += gameObject.transform.forward * Time.deltaTime * action_walk * movementSpeed;
```

Sl. 7.3.9 Funkcija AgentAction() (8)

Na slici Sl. 7.3.9 vidimo naš izlaz neuronske mreže, odnosno akciju koju agent izvršava.

```
// going wrong way???
distance = Vector3.Distance(transform.position,EndCube.transform.position);
if (distance < distanceLast) {
    AddReward (0.7f);
    WrongWay = false;
    counterBAD = 0;
} else {
    if (WrongWay == true) {
        Debug.Log ("counter= " + counterBAD);
        counterBAD = counterBAD + 1;
        if (counterBAD > 220) {
            for (int c = 0; c <= counterBAD - 50; c++) {
                AddReward (-0.022f);
            }
        }
    }
    AddReward (-0.3f);
    WrongWay = true;
}
```

Sl. 7.3.10 Funkcija AgentAction() (9)

```
public override void AgentReset()
{
    timeLeft = 110.0f;
    gameObject.GetComponent<Rigidbody>().velocity = Vector3.zero;
    gameObject.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
    gameObject.transform.position = playerStartDot;
    gameObject.transform.rotation = playerStartDotRotation;
    playerLastPos = playerStartDot;
}
```

Sl. 7.3.11 Funkcija AgentReset()

Funkcija AgentReset(), koristi se kada se agent sudari sa zidom ili krajnjom točkom. Ta ga funkcija vraća na početnu poziciju sa početnom brzinom i rotacijom te se resetira njegov iznos nagrade.

7.4. Okolina

Okolina ML-Agentima može biti bilo koja scena izgrađena u Unityu. Unity scena pruža okruženje u kojem agent može učiti, trenirati, opservirati i djelovati. Onako kako je scena napravljena da služi prilikom učenja, puno će utjecati na postizanje cilja. Prilikom izrade okoline i same igre, važno je gledati na to da igra započne na sceni na kojoj želimo agenta trenirati. Također je bitno namjestiti da njome može upravljati vanjski mozak, ako planiramo trenirati agenta.

Unutar okoline agent donosi opservacije i zaključke koje dalje prenosi mozgu. Opservacije dodajemo kroz funkciju CollectObservations().

```
public override void CollectObservations()
{

    Vector3 normalizedRotation = gameObject.transform.rotation.eulerAngles / 360.0f;
    AddVectorObs(normalizedRotation.y);

    AddVectorObs(gameObject.transform.position.x);
    AddVectorObs(gameObject.transform.position.y);
    AddVectorObs(gameObject.transform.position.z);
    //AddVectorObs(gameObject.transform.GetComponent<Rigidbody>().velocity);
    AddVectorObs(gameObject.transform.GetComponent<Rigidbody>().velocity.x);
    AddVectorObs(gameObject.transform.GetComponent<Rigidbody>().velocity.y);
    AddVectorObs(gameObject.transform.GetComponent<Rigidbody>().velocity.z);
```

Sl. 7.4.1 Funkcija CollectObservations() (1)

U funkciji CollectObservations(), pozivamo funkciju AddVectorObs() kojoj predajemo parametar, odnosno opservaciju. Tu funkciju možemo pozvati koliko god puta želimo, ovisno o tome koliko opservacija nam je potrebno da agent izvrši zadatak. Nekad pretjerivanje nije dobro te agentu treba puno više vremena da nauči koristiti se tim opservacijama.

Kako bi agent došao do kraja staze, potrebno mu je dati sljedeće opservacije:

- Trenutno stanje rotacije po Y osi
- Trenutnu poziciju na X, Y i Z osi
- Trenutno stanje brzine na X, Y i Z osi
- Udaljenost najbližeg predmeta lijevo
- Vrstu najbližeg predmeta lijevo
- Udaljenost najbližeg predmeta lijevo-naprijed
- Vrstu najbližeg predmeta lijevo-naprijed
- Udaljenost najbližeg predmeta naprijed
- Vrstu najbližeg predmeta naprijed
- Udaljenost najbližeg predmeta desno-naprijed
- Vrstu najbližeg predmeta desno-naprijed
- Udaljenost najbližeg predmeta desno

- Vrstu najbližeg predmeta desno

```
// LEFT
Physics.Raycast(transform.position, transform.TransformDirection(Vector3.left), out hit1, Mathf.Infinity);
AddVectorObs(hit1.distance);
if (hit1.collider != null && hit1.collider.tag == "goal") { AddVectorObs (1f);
} else { AddVectorObs (0f); }

// LEFT FRONT
Physics.Raycast(transform.position, transform.TransformDirection(new Vector3(-1, 0, 1)), out hit2, Mathf.Infinity);
AddVectorObs(hit2.distance);
if (hit2.collider != null && hit2.collider.tag == "goal") { AddVectorObs (1f);
} else { AddVectorObs (0f); }


// FRONT
Physics.Raycast (transform.position, transform.TransformDirection (Vector3.forward), out hit3, Mathf.Infinity);
AddVectorObs(hit3.distance);
if (hit3.collider != null && hit3.collider.tag == "goal") { AddVectorObs (1f);
} else { AddVectorObs (0f); }

// RIGHT FRONT
Physics.Raycast(transform.position, transform.TransformDirection(new Vector3(1, 0, 1)), out hit4, Mathf.Infinity);
AddVectorObs(hit4.distance);
if (hit4.collider != null && hit4.collider.tag == "goal") { AddVectorObs (1f);
} else { AddVectorObs (0f); }


// RIGHT
Physics.Raycast(transform.position, transform.TransformDirection(Vector3.right), out hit5, Mathf.Infinity);
AddVectorObs(hit5.distance);
if (hit5.collider != null && hit5.collider.tag == "goal") { AddVectorObs (1f);
} else { AddVectorObs (0f); }
```

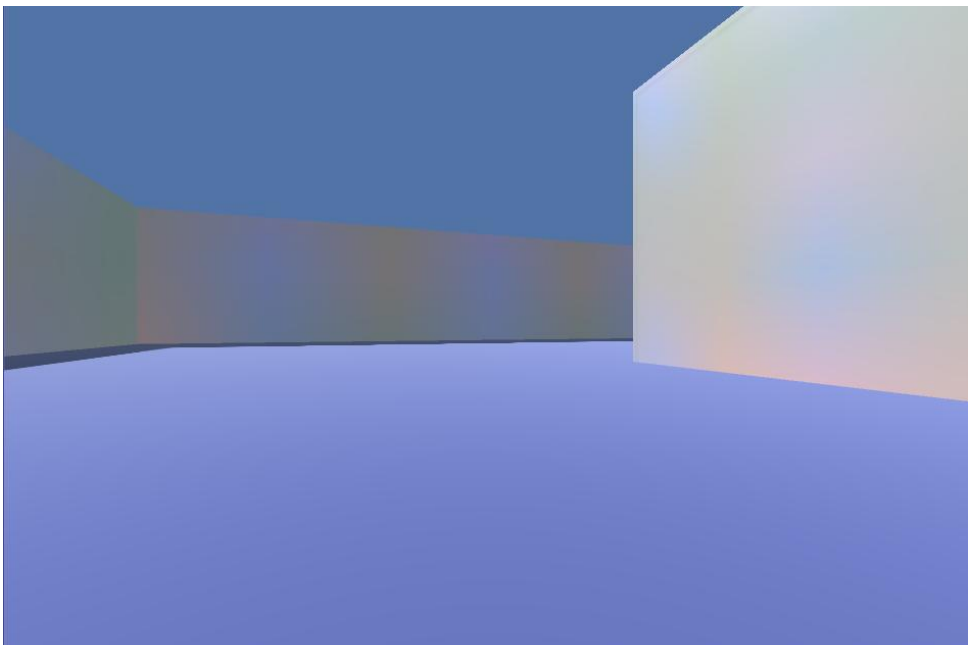
Sl. 7.4.2 Funkcija CollectObservations() (2)

```
void OnCollisionEnter(Collision colwall)
{

    if (colwall.gameObject.tag == "goal") {
        //gameObject.GetComponent<Rigidbody>().velocity = new Vector3(0f, 0f, 0f);
        for(float i=0; i<timeLeft;i++){
            AddReward (1f);
        }
        AddReward (1f);
        AddReward (1f);
        AddReward (1f);
        AddReward (1f);
        Done ();
        SetReward (1f);
        //Debug.Log("ENDED IT");
    }else if (colwall.gameObject.tag == "Wall" || colwall.gameObject.tag == "WallU") {
        //gameObject.GetComponent<Rigidbody>().velocity = new Vector3(0f, 0f, 0f);
        for(float i=0; i<timeLeft;i++){
            AddReward (-0.5f);
        }
        Done();
        SetReward(-1f);
        //Debug.Log("Hit the wall");
    }
}
```

Sl. 7.4.2 Funkcija OnCollisionEnter()

Agentova interakcija s okolinom je sudaranje sa zidom ili s krajnjom točkom. Sudaranje agenta s bilo kojim objektom provjeravamo funkcijom OnCollisionEnter(). U slučaju da se agent sudario u zid, dodjeljuje mu se negativna nagrada te se iznos agentove nagrade resetira na početak. Kada se agent sudari u krajnju točku, dobiva pozitivnu nagradu te također kreće iz početka s resetiranim iznosom nagrade. Resetiranje, odnosno kraj epizode pokrećemo funkcijom Done().



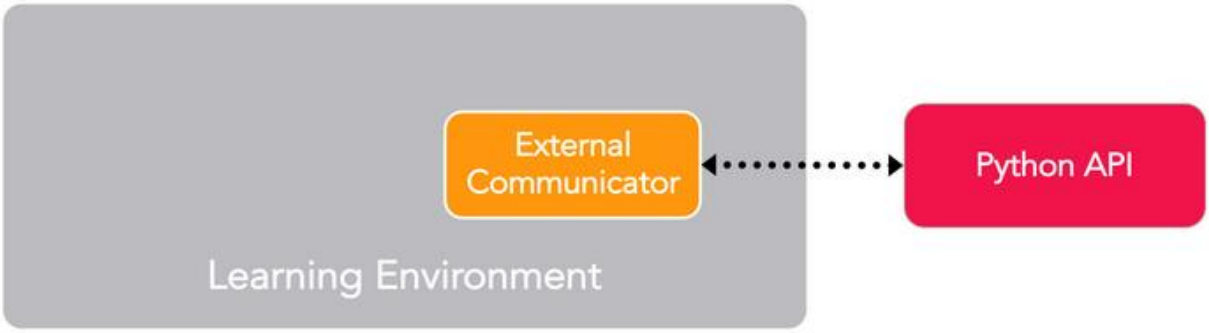
Sl. 7.4.3 Agentov pogled na okolinu

8. TRENIRANJE AGENATA STROJNOG UČENJA

ML-Agenti izvršavaju treniranje korištenjem vanjskog Python trening procesa. Tijekom treninga, ovaj vanjski proces komunicira s akademijom objekta u Unity sceni kako bi stvorio blokove agentovog iskustva. To iskustvo zatim postaje set za treniranje neuronske mreže koja se koristi za optimiziranje pravila agenta. U podržanom učenju, neuronska mreža optimizira pravila tako što maksimizira očekivane nagrade.

Izlaz procesa treniranje je podatak modela koja sadržava optimizirana pravila agenta. Taj podatak je TensorFlow model grafa podataka (engl. *data graph*) koji sadrži matematičke operacije i optimizirane težine odabrane kroz proces treniranja. Taj istrenirani model zatim možemo koristiti u našem projektu, ako namjestimo mozak na vrstu unutarnji.

Korištenjem Python skripte (programa), learn.py treniramo agente. Ovu skriptu možemo naći u direktoriju python unutar ML-Agents direktorija kojeg smo prethodno skinuli. Podatak kroz kojeg postavljamo svojstva, trainer_config.yaml, omogućava namještanje specifičnih hiperparametara koji se koriste kroz treniranje. Njega možemo mijenjati koristeći bilo koji program za izmjenu teksta.



Sl. 8.1 Komunikacija između Python API-a i stvorenog okruženja [18]

8.1. Treniranje kroz skriptu learn.py

Korištenjem Python skripte learn.py treniramo agente. Python skripta podržava treniranje podržanog učenja, učenje programa (engl. *curriculum learning*) i imitacijsko učenje (engl. *behavioral cloning imitation learning*).

Skriptu learn.py pokrećemo kroz komandnu liniju. Prije pokretanja moramo pokrenuti prethodno stvoreno okruženje koje ima instaliran TensorFlow i Python 3. Također bi bilo dobro provjeriti svojstva namještena u trainer_config.yaml datoteci koja kontroliraju treniranje.

Osnovna komanda za treniranje je:

python3 learn.py <env_ime> --run-id=<ID-ucenja> --train

gdje je:

- <env_ime> je ime koje uključuje cijeli put do naše Unity (.exe) datoteke kroz koju pokrećemo našu igru. Ako se <env_ime> ne preda kao parametar, trening će započeti unutar Editora.

- <ID-ucenja> je neobavezan identifikator koji će služiti kao naziv kada se modeli budu spremali. Ova opcija je vrlo korisna kod organiziranja, pogotovo ako se treniraju razni agenti na raznim okruženjima.

```
cd C:\ml-agents\python
activate tfp3.5
python learn.py FirstRunv4.exe --run-id=i10_4 --save-freq=10000 --train
```

Sl. 8.1.1 Primjer pokretanja procesa treniranja

Na slici Sl. 8.1.1 možemo vidjeti primjer pokretanja learn.py skripte. Korisni dodatci naredbi su „—save-freq“ kojim namještamo frekvenciju spremanja modela izraženu u stopama treniranja te „—load“ koji omogućuje da nastavimo treniranje modela kojeg smo već trenirali.

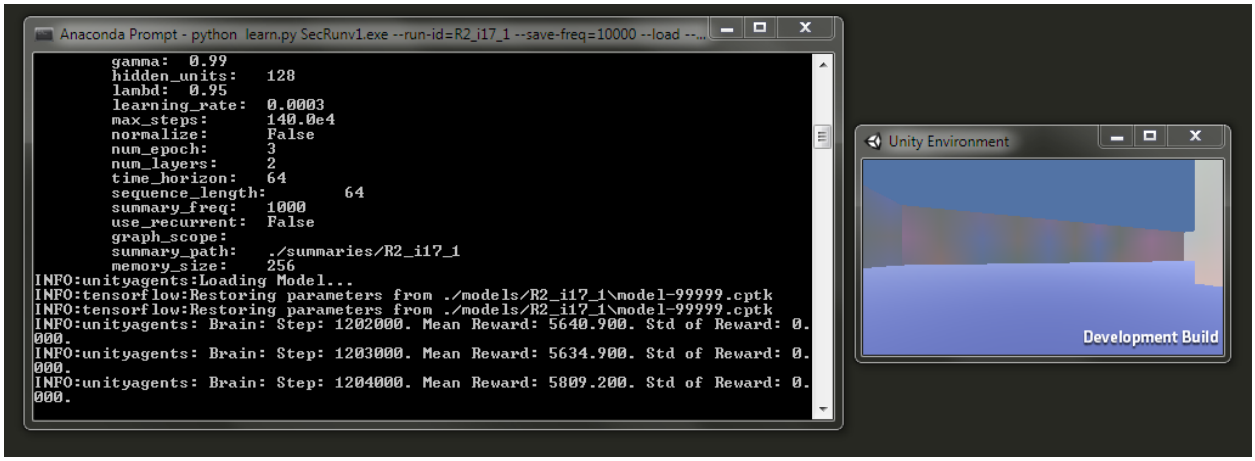
8.2. Simulacija i proces treniranja

Trening i proces simulacije izvršavaju se u koracima orkestriranim ML-Agent akademije (klase). Akademija radi s agentima i mozgovima u sceni kako bi prošli kroz simulaciju. Kada akademija dođe do kraja svojim maksimalnih koraka ili svi agenti u sceni izvrše funkciju Done(), epizoda treniranja je gotova.

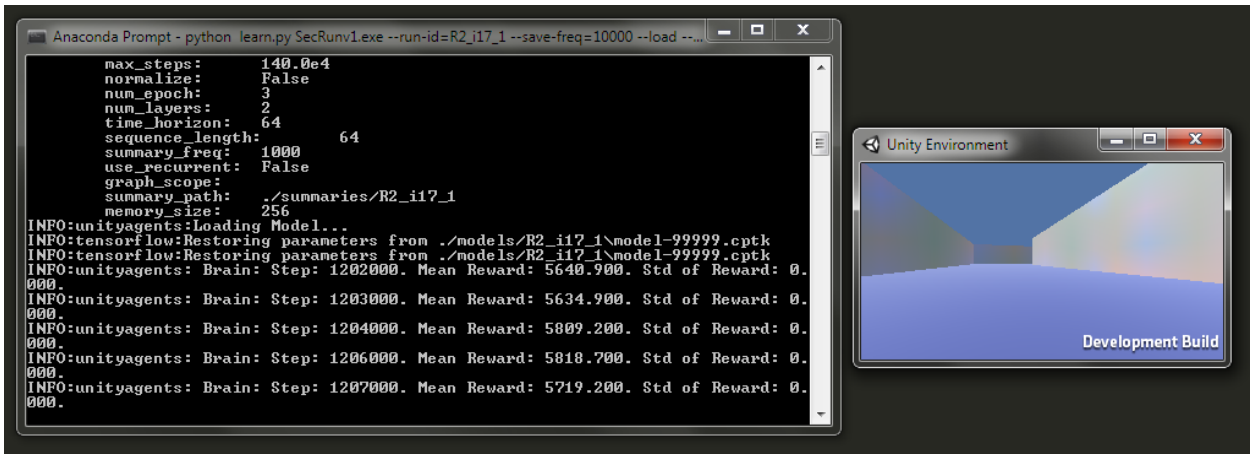
Tijekom treninga vanjski Python proces komunicira s akademijom te izvršava seriju epizoda dok prikuplja podatke i optimizira njegov model neuronske mreže. Vrsta mozga agenta određuje, sudjeluje li on u ovom procesu ili ne. Vanjski mozgovi izvršavaju ovu komunikaciju kako bi trenirali TensorFlow model.

ML-Agentova Academy klasa orkestrira agentove simulacije u sljedećoj petlji:

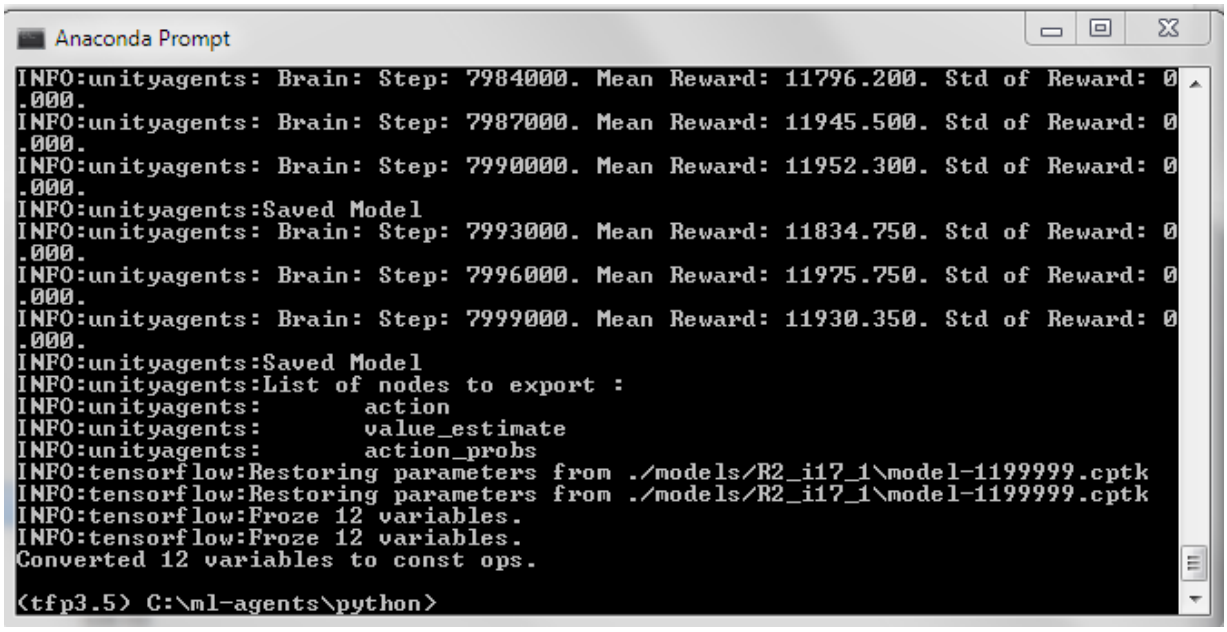
- Poziva funkciju AcademyReset() potklase Academy klase.
- Poziva funkciju AgentReset() za svakog agenta u sceni.
- Poziva funkciju CollectObservations() za svakog agenta u sceni.
- Koristi klasu Brain svakog agenta da odredi sljedeću akciju agentu.
- Poziva funkciju AcadamyAct() potklase Academy.
- Poziva funkciju AgentAction() za svakog agenta u sceni, prosljeđujući im odabrane akcije agentovog mozga.
- Poziva agentovu funkciju AgentOnDone() ako je ima te ako je označen kao gotov (engl *Done*). Opcionalno možemo namjestiti agenta da se restarta ako je gotov, čak i prije kraja epizode. U tom slučaju akademija poziva AgentReset() funkciju.
- Kada akademija dođe do „Max Step“ broja kojeg smo namjestili ili su svi agenti gotovi, započinje nova epizoda treniranja pozivanjem funkcije potklase Academy, AcademyReset().



Sl. 8.2.1 Proces treniranja (1)



Sl. 8.2.2 Proces treniranja (2)



Sl. 8.2.3 Kraj proces treniranja

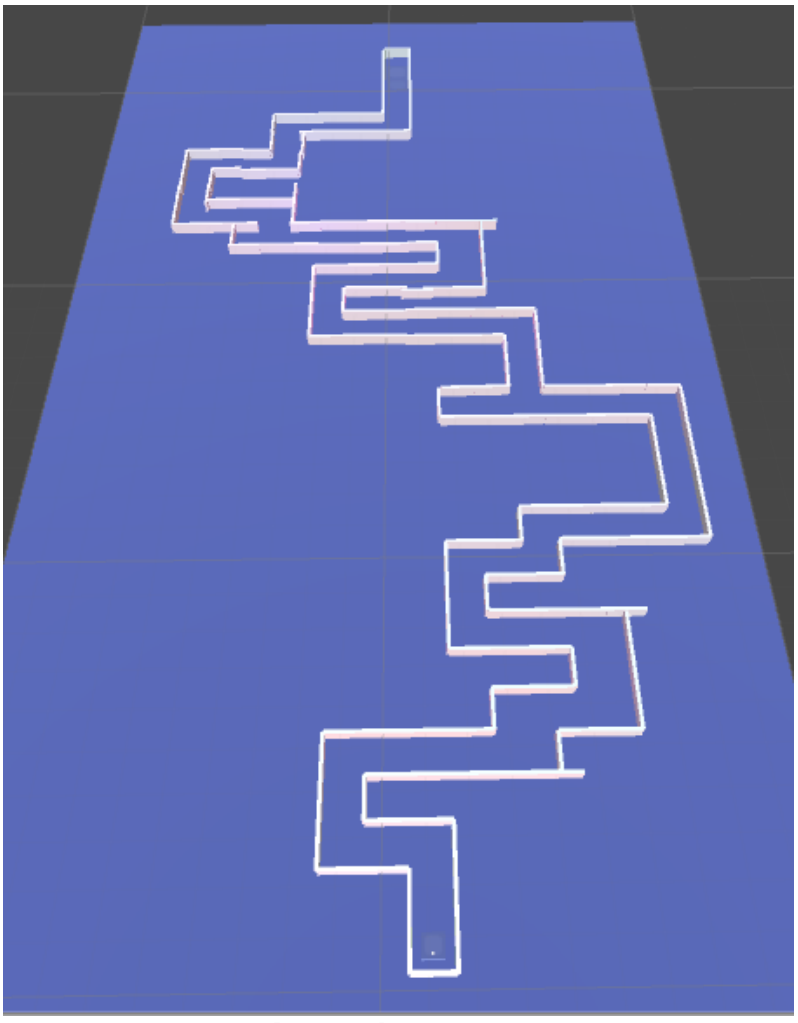
Za vrijeme treniranja moguće je gledati napredak agenta, na malom ekranu kojeg nam omogućuje Development Build kod izrade („buildanja“) igre, prikazan na slikama Sl. 8.2.1 i Sl. 8.2.2. Na istim slikama možemo vidjeti proces spremanja modela i ispisivanja određenog broja koraka kako bismo znali kako agent napreduje.

9. REZULTAT

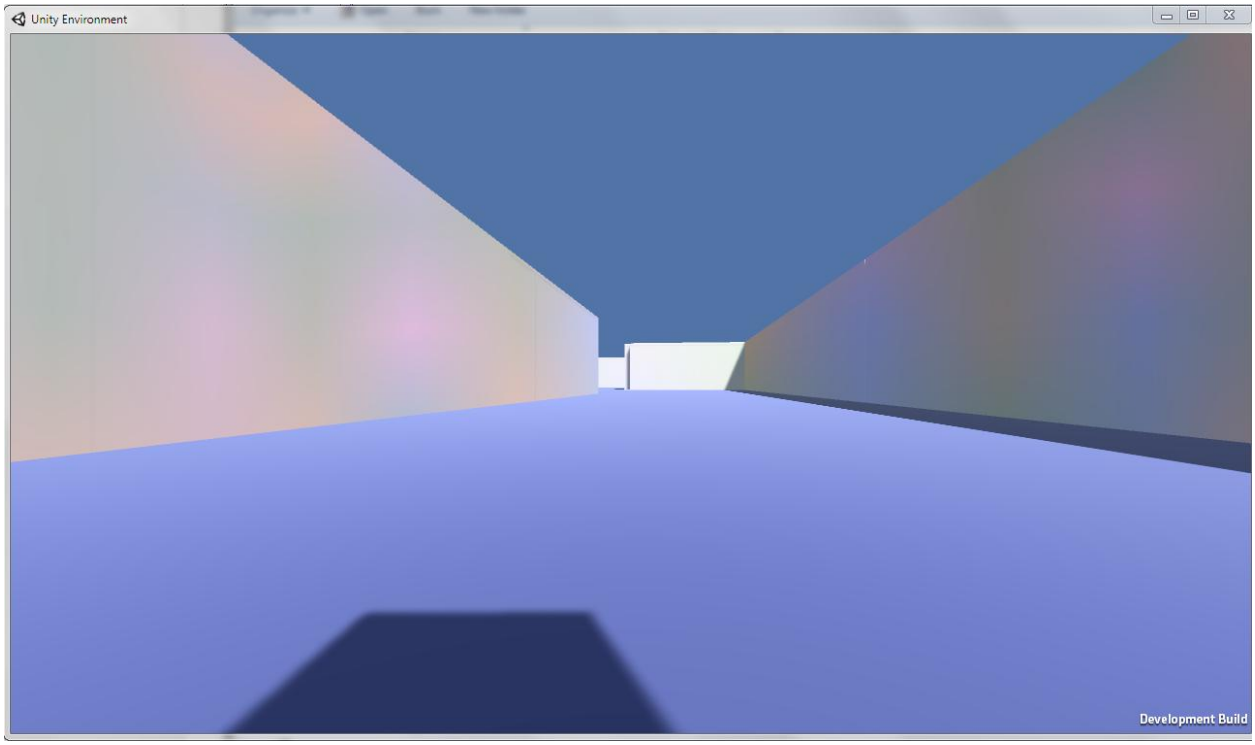
Kroz trening od 8 000 000 koraka, u trajanju od otprilike 40 sati, postignuti su sljedeći rezultati:

- Agent raspoznaje razliku između zida i krajnje točke puta
- Agent je naučio što znači pojedina opservacija
- Agent je naučio kako se koristiti svim mogućim akcijama koje on može izvršiti
- Agent je naučio kretati se kroz stazu, pri tome ne sudarajući se sa zidom.

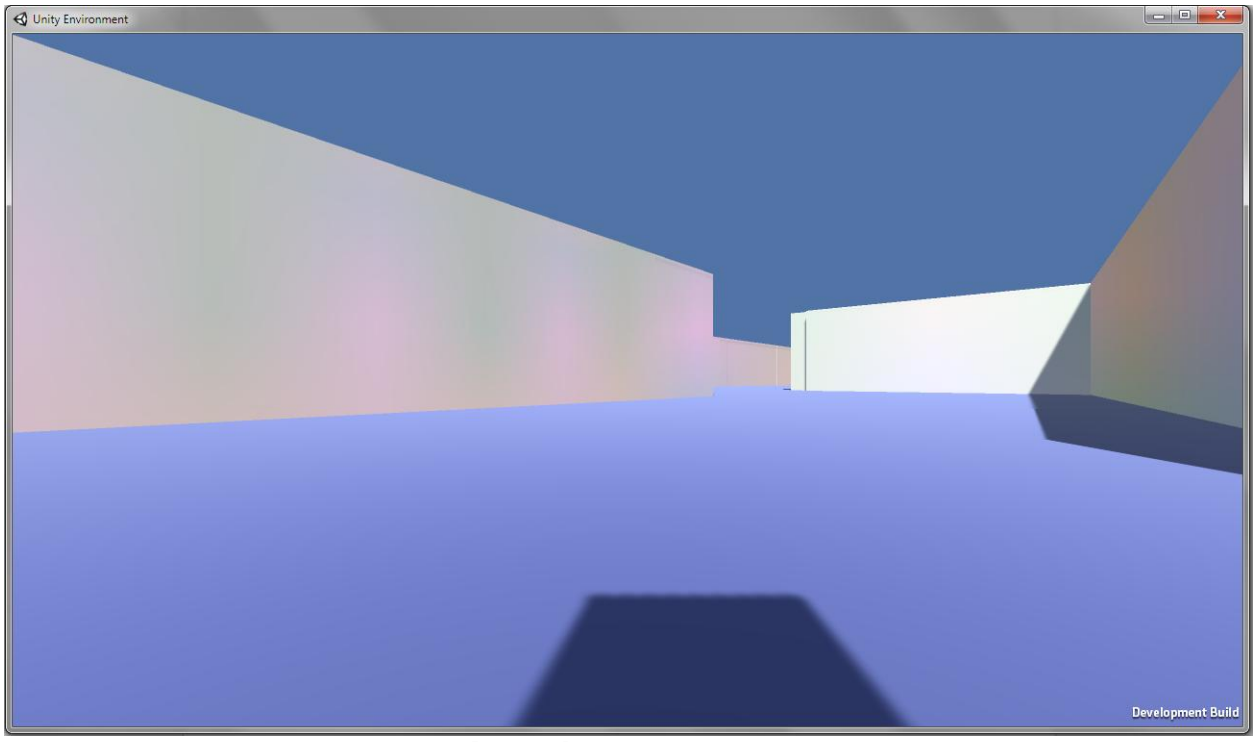
Agent je učio od lakše staze pa sve do najteže, na svakoj je proveo minimalno 10 sati treniranja. Staza koju je zadnju savladao prikazana je na slici Sl. 9.1.



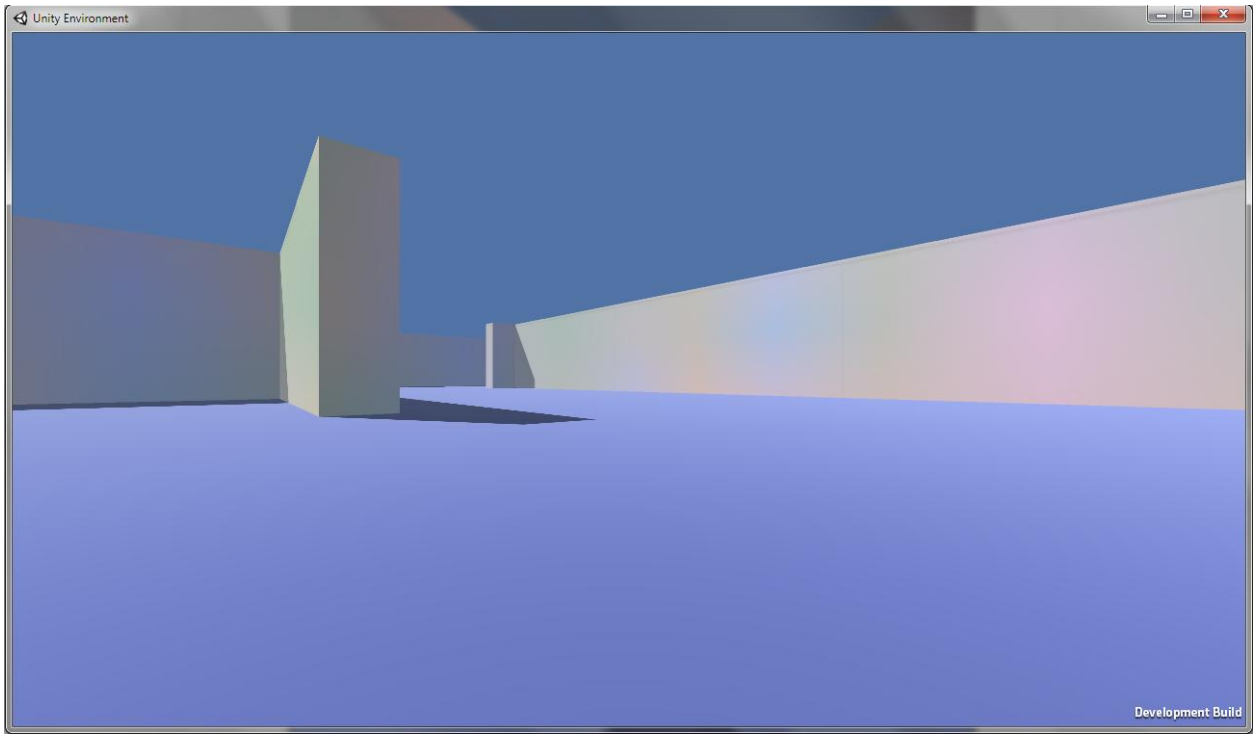
Sl. 9.1 Labirint (staza)



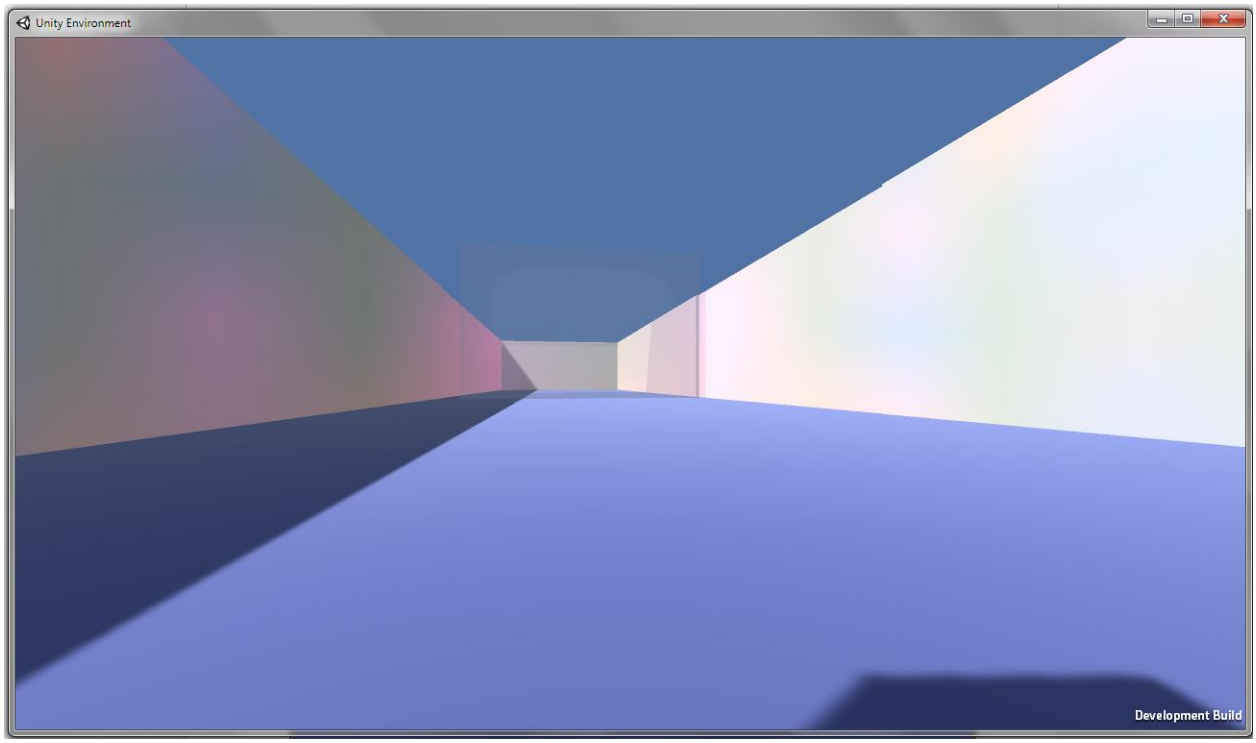
Sl. 9.2 Kretanje agenta kroz labirint



Sl. 9.3 Skretanje agenta (1)



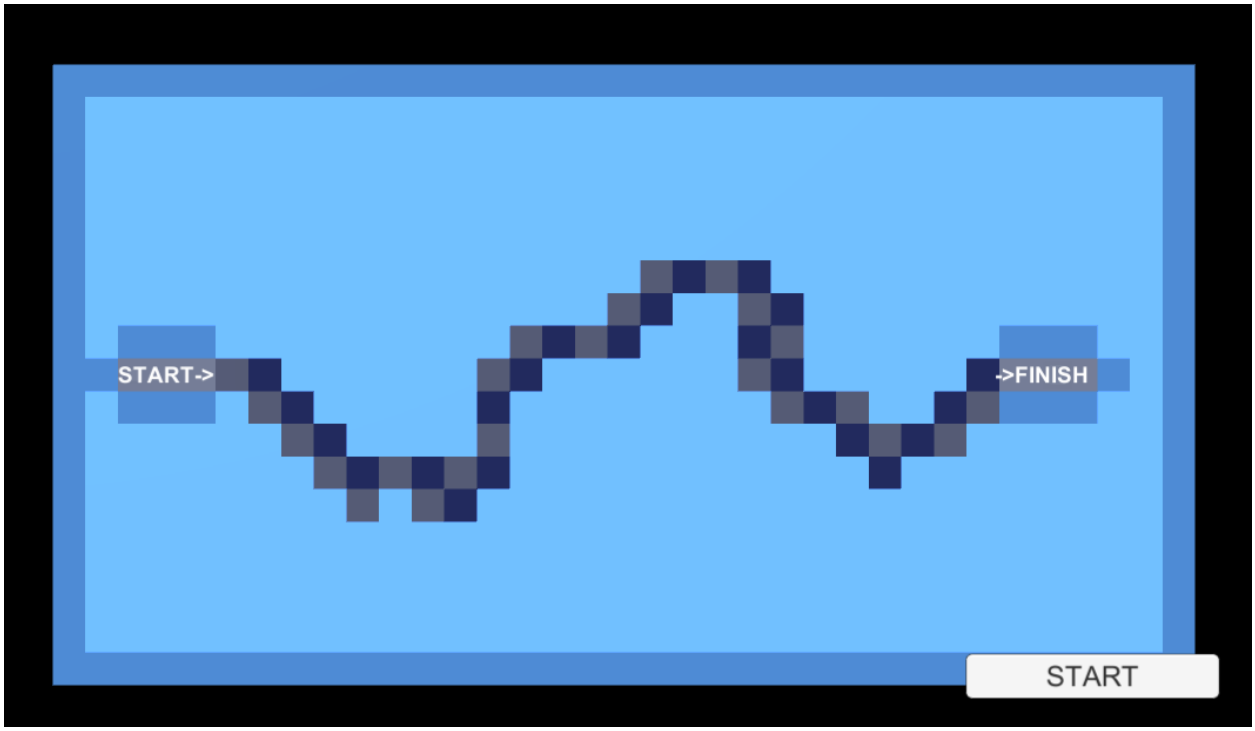
Sl. 9.4 Skretanje agenta (2)



Sl. 9.5 Dolazak agenta do kraja labirinta

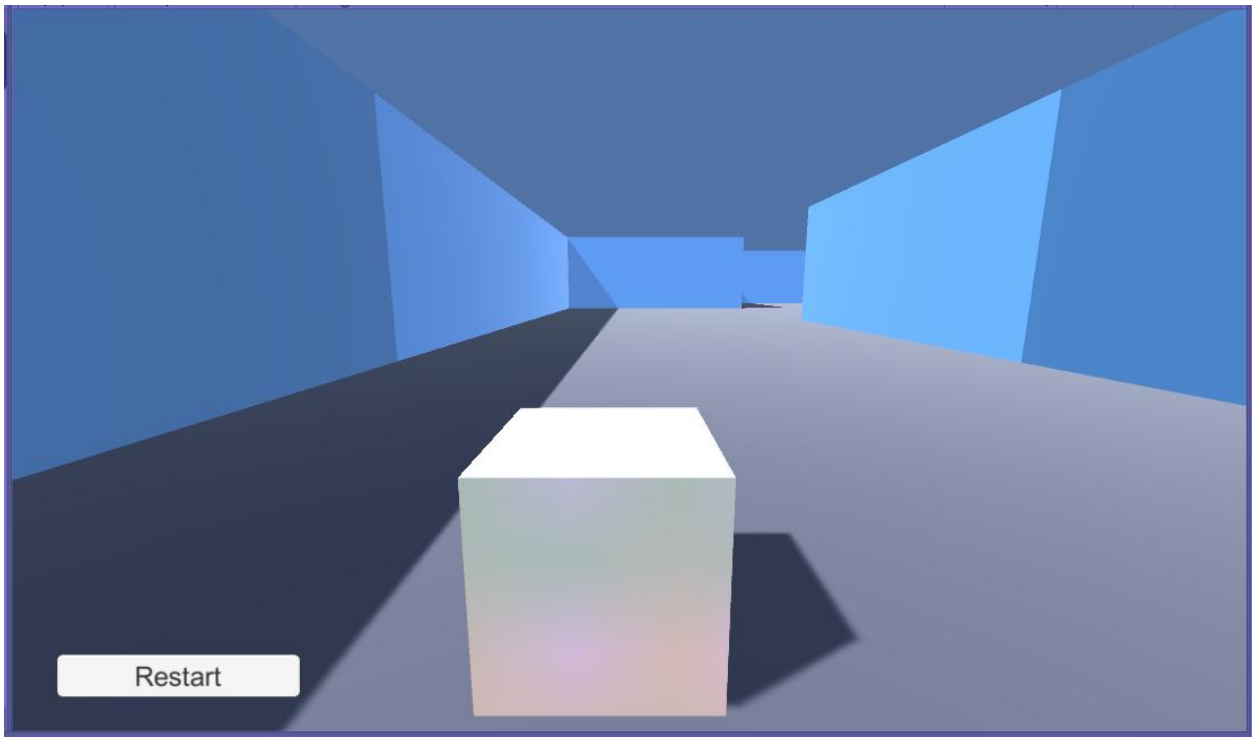
9.1. Zadnja verzija igre

Nakon što je agent naučio koristiti osnovne akcije i tako prolaziti osnovne putanje, izrađena je igra (aplikacija) u kojoj korisnik sam može zadati stazu koju će zatim agent prolaziti.



Sl. 9.1.1 Primjer napravljene staze

Pokretanjem igre dobivamo praznu površinu na kojoj crtamo stazu kojom će se agent kretati. Lijevim klikom crtamo, a desnim klikom miša brišemo crte. Potrebno je nacrtati neprekidnu krivulju od „START->“ do mjesta „->FINISH“. Pritiskom na tipku „START“ u donjem desnom kutu započinjemo kretanje agenta.



Sl. 9.1.2 Primjer pokrenute igre

Nakon pritiska tipke „START“, staza se generira te agent kreće svoje kretanje kroz nju. Isto kao i kod treniranja, ako se agent zabije u zid, kreče iz početka. Pritiskom na tipku „Restart“ vratit ćemo se na prethodni izbornik te ćemo moći ponovo crtati novu stazu.

Koristeći izrađenu igru, lakše smo mogli stvarati nove staze te time nastaviti treniranje na težim i zahtjevnijim stazama. Ukupno treniranje dovodi i više od 20 000 000 koraka što iznosi otprilike preko 100 sati treniranja.

10. ZAKLJUČAK

Kroz isprobavanja različitih opservacija i akcija koje dopuštamo agentu na izvršavanje, možemo zaključiti da više (opservacija ili akcija) ne znači nužno bolje. Uz više nepotrebnih opservacija, agentovo učenje se znatno usporava, a nekada uopće ni ne uspije naučiti kako kontrolirati moguće akcije. Vrlo bitno je agentu postupno pojačavati težinu zadatka. Treba napraviti stazu sa svim mogućim kombinacijama skretanja kako bi mogao naučiti kretati se na nasumično generiranim stazama. Vrlo je bitno postepeno isprobavanje i balansiranje mehanike nagrađivanja. Treba trenirati isti model malo po malo, uz izmjenu skripte i staze postepeno. Tako ćemo lakše uočiti velike promjene i znati gdje je neka promjena bila dobra ili loša. Agent čak i nakon 100 sati treniranja nije savršen. Ima puno grešaka kao što su slabije raspoznavanje krajnje točke od zida, problem kod izbora kada može skrenuti i lijevo i desno u isto vrijeme te problem s izlaskom iz beskonačnih petlji. Mogućnost da se ti problemi riješe mogući su dodavanjem još nekih opservacija no to bi dovelo do treniranja cijelog modela iz početka s ukupnim vremenom treniranja većim nego sada.

LITERATURA

[1] Unity online stranica, https://unity3d.com/ 25.4.2018.

[2] Unity online video zapisi, https://unity3d.com/learn/tutorials 25.4.2018.

[3] Unity online priručnik, http://docs.unity3d.com/Manual/index.html 26.4.2018.

[4] Unity o kamerama, http://docs.unity3d.com/Manual/CamerasOverview.html 1.5.2018.

[5] Unity o osvjetljenju, http://docs.unity3d.com/Manual/LightingOverview.html 1.5.2018.

[6] Unity o grafici, http://docs.unity3d.com/Manual/Graphics.html 1.5.2018.

[7] Unityev o stvaranju terena, http://docs.unity3d.com/Manual/script-Terrain.html 2.5.2018.

[8] Wikipedia, Unity 3D, https://en.wikipedia.org/wiki/Unity_(game_engine) 2.5.2018.

[9] Unity free download stranica, https://unity3d.com/get-unity/download/archive 10.5.2018.

[10] Osnove Programskog jezika C#, https://www.infoq.com/minibooks/emag-c-sharp-preview 12.5.2018.

[11] Wikipedia, C# sintaksa, https://en.wikipedia.org/wiki/C_Sharp_syntax 12.5.2018.

[12] ML-Agenti GitHub, https://github.com/Unity-Technologies/ml-agents.git 22.5.2018.

[13] Anaconda Prompt download, https://www.anaconda.com/download/#windows 22.5.2018.

[14] cuDNN download, https://developer.nvidia.com/cudnn 22.5.2018.

[15] ML-agents dokumenti o instaliranju za Windows OS, https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Installation-Windows.md 30.5.2018

[16] Nvidia lista CUDA podržavanja, https://developer.nvidia.com/cuda-gpus 2.5.2018.

[17] CUDA download, https://developer.nvidia.com/cuda-toolkit-archive 2.5.2018.

[18] Dokumenti o ML-Agents treniranju, https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md#running-example-training-npc-behaviors 10.6.2018.

[19] Osnove PPO algoritma, https://blog.openai.com/openai-baselines-ppo/ 14.6.2018

[20] Uvod u ML-Agente, https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/ 15.6.2018.

SAŽETAK

Unity 3D, jednostavan program za izradu igara, uz ML-Agents dodatak pruža nove usluge treniranja umjetne inteligencije da izvršava određene zadatke. Treniranje agenata ne ovisi samo o jačini CPU ili GPU, već ovisi i o okruženju koje stvorimo za agente te mehanici nagrađivanja koju napišemo u kodu. Agenti uče uz pomoć neuronske mreže kojoj mi određujemo broj ulaza, odnosno opservacija te broj izlaza, odnosno akcija koje agent izvršava. Vrlo je bitno agenta postepeno učiti od lakših pa sve do težih zadataka. Da bismo trenirali agente kroz Unity i ML-Agents dodatak, nije potrebno dodatno znanje o neuronskim mrežama zbog toga što nam funkcije neuronske mreže izvršava TensorFlow u pozadini, algoritmom PPO kojeg mi ne mijenjamo.

Ključne riječi: Unity 3D, Python, Agenti Strojnog Učenja, PPO

ABSTRACT

Unity 3D, simple program for creating games, with ML-Agents addition gives new services of training artificial intelligence to do the specific tasks. Training agents do not depend only on strength of CPU or GPU, but it depends more on the environment which we create for agents and also the mechanic of rewarding that we write in code. Agents learn throughout the help of neuron network whom we determine the number of inputs, also called observations and the number of outputs, also called actions that the agent needs to do. It is very important for the agent to be learnt step by step from very easy to very hard tasks. To train agents throughout Unity and ML-Agents, it is not necessary to know additional knowledge about neural networks because all functions of the neural network are done by TensorFlow in the background, by algorithm PPO which we do not change by hand.

Key words: Unity 3D, Python, Machine Learning Agents, PPO

ŽIVOTOPIS

Vedran Brazdil rođen je 01. Prosinca 1994. godine u Osijeku. Živi u Osijeku te se u istom mjestu obrazuje. Pohađao je Osnovnu školu Ljudevita Gaja u Osijeku od 2001. do 2009. Zatim nastavlja svoje obrazovanje u srednjoj Elektrotehničkoj i prometnoj školi Osijek, četverogodišnjeg smjera elektrotehničar. Završetkom srednje škole, uspješno upisuje preddiplomski sveučilišni studij računarstva 2013. godine. Sve ispite redovno polaže te završava preddiplomski sveučilišni studij 2016.g s temom završnog rada „Proceduralno generiranje 3D svijeta u Unity 3D“. Nakon preddiplomskog studija upisuje diplomski studij računarstva, smjer Informacijske i podatkovne znanosti. Praksu obavlja u Atos-u, u Osijeku, gdje od 7. mjeseca 2017. godine radi kao student.

Potpis:

_____