Sustav protiv varanja implementacijom reinforcement learning agenata u Unityu

Mihael, Lukaš

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:614922

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihael Lukaš

Sustav protiv varanja implementacijom reinforcement learning agenata u Unityu

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihael Lukaš

Matični broj: 0016109639

Studij: Organizacija poslovnih sustava

Sustav protiv varanja implementacijom reinforcement learning agenata u Unityu

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Igor Tomičić

Varaždin, rujan 2021.

Mihael Lukaš

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad govori o sustavu protiv varanja te kako se implementacijom agenata pomoću podržanog učenja (eng. *Reinforcement Learning* – u nastavku RL) mogu raspoznati igrači od botova. Proučava se strojno učenje, njegove metode i načini na koje ih je moguće implementirati. Spominje se i sigurnost kao jedna od važnijih stavki prilikom igranja igara te na koji se način može varati i čime se igrači služe da bi dobili prednost nad svojim protivnicima.

Početni dio rada se bazira na teoriji i objašnjava što je to strojno učenje, koji se alati koriste prilikom treniranja, dok je drugi dio baziran na praktičnoj primjeni. Naime, izgradnjom vlastitog sustava protiv varanja, implementiranog u, također, vlastitu multiplayer igru, je bila namjera prikazati kako RL agent pronalazi botove i s kojom vjerojatnošću to može tvrditi.

Ključne riječi: sigurnost, umjetna inteligencija, infosec, reinforcement learning, agenti, igrice, gaming

Sadržaj

1. Uvod

Rad se bazira na temu sustava protiv varanja i to pomoću implementacije RL agenata. Cilj je prikazati teorijske osnove iz kojih se crpe znanja o tome kako bi se trebao implementirati agent koji samostalno uči te kako primjenjuje naučeno da bi detektirao bota.

Kako bi se došlo do podataka za izradu teorijskog dijela, korištena je literatura koja se sastoji od knjiga, internetskih stranica i znanstvenih članaka, a također je izgrađen i praktični dio u vidu vlastitog sustava protiv varanja koji se očituje u igri. Igra je u trećem licu i sastoji se od dva tima, gdje se upravlja likom kojem je cilj prenijeti pet kutija na drugu stranu prije nego što to napravi drugi tim u određenom vremenu. Za razvoj igre je korišten Unity, a implementiranje RL agenta je omogućeno pomoću biblioteke The Unity Machine Learning Agents Toolkit (u nastavku ML-Agents). Glavni cilj ovog projekta jest odrediti što sve agent mora proučavati unutar igre, na koji način mora donositi odluke te odrediti nagrade prema kojima će naučiti što treba raditi. Umjetna inteligencija, odnosno specifično RL, napreduje velikim koracima i koristi se u brojnim stvarnim primjerima. Računalo samostalno uči voziti automobil, pomaže pri trgovanju i financijama, obrađuje prirodni jezik, koristi se u zdravstvu, igrama, robotici i tako dalje.

Sadržaj rada se temelji na prikazu značajki i metoda strojnog učenja, kao što su nadzorno učenje, nenadzorno i RL. Osim toga, proučava se funkcioniranje neuronskih mreža jer su veoma važna stavka u strojnom učenju i imaju široku primjenu u različitim granama znanosti. Dana su objašnjenja o projektu te su doneseni zaključci.

2. Strojno učenje

Strojno učenje (eng. *Machine Learning*) je jednostavno objasnio Arthur Samuel 1959. godine rekavši da je to područje istraživanja koje omogućuje računalu učenje bez izričitog programiranja [1]. Iako su već postojale definicije i primjeri 50-ih godina o strojnom učenju, postao je popularan tek 90-ih godina, većinom zbog razvoja tehnologije. 1952. godine, Arthur Samuel, napravio je umjetnog igrača za igru na ploči *Dama*. To je strateška igra za dva igrača koji s dijagonalnim potezima osvaja protivničke figure tako da se preskoči preko njih. Program koji je napisao radio je na računalu IBM 701 te je čak prikazan na televiziji 1956. godine. Funkcionirao je tako da je imao stablo pretraživanja s trenutno postavljenim figurama i pomoću njega proučavao moguće poteze i odlučivao koji je najbolji potez. Glavna prepreka je bila ta što je trebalo puno vremena. Računala su imali male memorije pa je kao rješenje uveo dodatne parametre, kao što je *alpha-beta* granična vrijednost s kojima je ubrzavao proces pronalaženja optimalnog poteza. Potezi koje je radilo računalo nisu bili na profesionalnoj razini nego više na amaterskoj, pa računalo nije moglo pobijediti profesionalnog igrača. Neovisno o tome, ovaj program je za to vrijeme bio veliki korak prema strojnom učenju i umjetnoj inteligenciji [2].

Još jedna detaljnija definicija strojnog učenja, jest sljedeća: *Računalnom programu je rečeno da uči iz iskustva E s obzirom na neki zadatak T i neku mjeru performansi P, ako se njegova izvedba na T, mjereno s P, poboljšava s iskustvom E* [3]. Ovu definiciju kazao je Tom Mitchell 1998. godine unutar svoje knjige *Machine Learning*. Primjer je postojanje sentimenta u tekstu. Svakom se tekstu mogu dodijeliti tri stanja sentimenta, a to su pozitivni, neutralni i negativni. Primjer pozitivnog sentimenta je sljedeći tekst: *Danas je sunčano vrijeme i osjećam se odlično!*, jer osoba ima pozitivan stav i mišljenje. Ako pak postoji tekst: *Automobil mi se pokvario prilikom odlaska na posao.* Znači da je on negativnog sentimenta zato što osoba ima negativno mišljenje. Sve ostalo za što osoba nema određeno mišljenje je neutralni sentiment [4]. Prema definiciji Toma Mitchella, zadatak T je u ovom primjeru određivanje sentimenta. Da bi računalo samostalno odredilo sentiment, potreban mu je skup podataka (veliki broj tekstova) na kojem su već određeni sentimenti. Pomoću tog skupa podataka, računalo će učiti i dobiti iskustvo E. Nakon učenja, računalo s nekim postotkom uspješnosti može određivati sentiment. Koliko ispravno određuje sentiment, ovisi o mjeri učinkovitosti T prema kojoj će se analizirati rezultati i određivati koliko često računalo ispravno odredi sentiment.

U današnje vrijeme, strojno učenje se koristi gotovo svugdje, od surfanja po web preglednicima, prikaza oglasa, raspoznavanja objekata preko slika, pa sve do medicine, robotike i autonomne vožnje automobila. Teško je zamisliti svijet bez strojnog učenja.

Strojno učenje čine nekoliko metoda, a to su nadzorno učenje (eng. *supervised learning*), nenadzorno učenje (eng. *unsupervised learning*) i podržano učenje (RL).

2.1. Nadzorno učenje

Najčešće korištena metoda strojnog učenja je nadzorno učenje. Za njega je potrebno imati skup podataka (eng. *training set*) na kojem se trenira model. Odluku će donositi samo prema naučenim podacima, što pokazuje ovisnost o tome kakav je skup podataka. Cilj nadzornog učenja jest taj da se trenira model $y = f(x)$, tako da se predvidi izlaz y, baziran na ulazu x.

Problemi unutar nadzornog učenja su kategorizirani u probleme regresije i klasifikacije. U regresijskom se pokušavaju predvidjeti rezultati unutar kontinuiranog izlaza, što ustvari znači da se pokušavaju preslikati ulazne varijable u neku kontinuiranu funkciju i prema tome dobiti rezultate. Primjer regresije je određivanje koliko godina ima osoba prema njezinoj slici. Problemom klasifikacije se pokušavaju predvidjeti rezultati u diskretnom izlazu, odnosno preslikati ulazne varijable u diskretne kategorije. Primjer toga jest odrediti je li neki mail spam ili nije [5].

Ulazni podaci u algoritme se nazivaju svojstvima (eng. *feature*). To su glavni atributi na kojima se provodi treniranje modela. Obično su numeričkog tipa podataka, ali u nekim slučajevima mogu biti tekstovi ili čak grafikoni. Ako postoji više svojstava koja se uzimaju u obzir, gotovo svaki put ih je potrebno staviti na sličnu skalu. Primjer toga je određivanje vrijednosti stana. Na sljedećoj tablici se mogu vidjeti slično prodani stanovi, odnosno primjer skupa podataka s kojim bi se mogao trenirati model nadzornog učenja.

Tablica 1. Karakteristike stanova

No.	Kvadratura (m^2)	Cijena (eura)	Broj soba
1.	185	950.000	6
2.	120	1.000.000	4
3.	90	320.000	3
4.	30	250.000	3
5.	60	310.000	5

[autorski rad]

U ovome primjeru, svojstva su kvadratura, cijena i broj soba. Kako je cijena dosta veliki broj, a kvadratura i broj soba manji, bez skaliranja bi cijena imala puno veću važnost nego broj soba ili kvadratura. Da se izjednače te vrijednosti, odnosno da budu iste važnosti, potrebno je provesti skaliranje svojstava (eng. *feature scaling*). Da bi se skaliralo svojstvo, potrebno je uzeti maksimalnu vrijednost i podijeliti svaku vrijednost s njome. Maksimalna vrijednost kvadrature jest 185 i da bi se izračunala vrijednost za stan 3, uzme se njegova kvadratura i podijeli se s maksimalnom vrijednošću.

$$\frac{90}{185} = 0{,}486$$

Njihovim dijeljenjem dobije se vrijednost 0,486. Takvim računanjem uvijek će se dobiti vrijednost između broja 0 i 1. Kada bi se na taj način izračunalo skaliranje svojstava, tada bi izgledao ovako:

Tablica 2. Skaliranje svojstava

No.	Kvadratura (m^2)	Cijena (eura)	Broj soba
1.	1	0,95	1
2.	0,649	1	0,667
3.	0,486	0,32	0,5
4.	0,162	0,25	0,5
5.	0,324	0,31	0,833

[autorski rad]

S takvim svojstvima se dalje može trenirati model sa sigurnošću da je svako svojstvo identične važnosti [6].

Prilikom skaliranja svojstava, može se provesti i srednja normalizacija (eng. *mean normalization*) kod koje se želi izbjeći da najveća vrijednost bude 1. Ako bi se uzela cijena, tada bi se normalizacija napravila na način da se uzme srednja vrijednost svih podataka. Na velikom skupu podataka je to možda nemoguće, pa se može uzeti približna srednja vrijednost. U ovom slučaju bi to iznosilo 566.000 eura. Tada se uzme trenutna vrijednost (primjerice redni broj 3 koji iznosi 320.000), te se od njega oduzme srednja vrijednost i podijeli s maksimalnim brojem.

$$\frac{320.000 - 566.000}{1.000.000} = -0{,}246$$

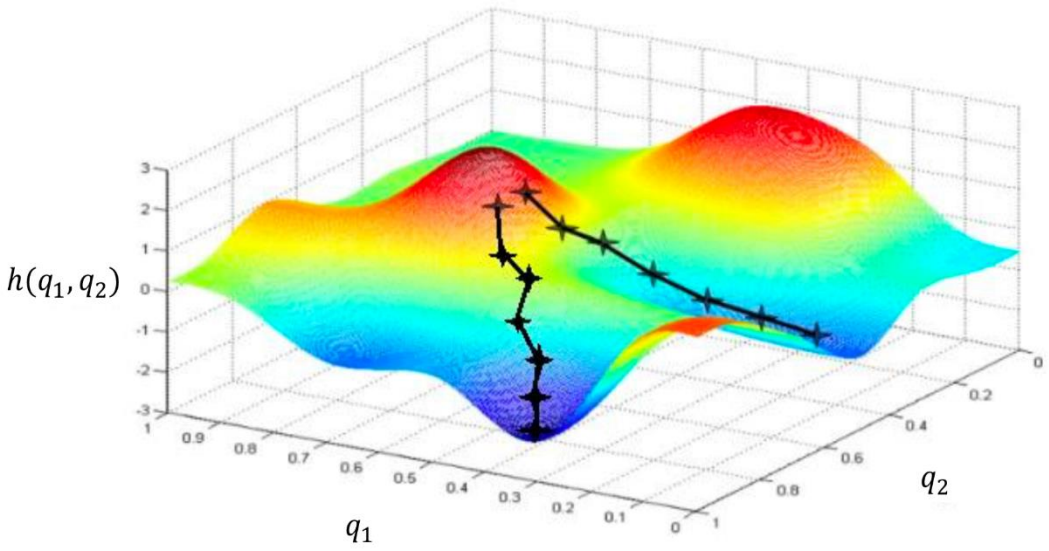
Tada je moguće da vrijednosti budu i negativne, odnosno ako se sve izračuna na taj način, dobit će se sljedeća tablica [6].

Tablica 3. Srednja normalizacija vrijednosti

No.	Kvadratura (m^2)	Cijena (eura)	Broj soba
Average	97	566.000	4,2
1.	0,476	0,384	0,3
2.	0,124	0,434	-0,033
3.	-0,038	-0,246	-0,2
4.	-0,362	-0,316	-0,2
5.	-0,2	-0,256	0,133

[autorski rad]

Nakon što su odabrana i uređena svojstva, može se prijeći na računanje funkcije troškova i gradijentskog spuštanja. Cilj nadzornog učenja je da se na dobivenom skupu podataka nauči funkcija *h* koja preslikava X u Y. Funkcija *h* se naziva hipoteza. Da bi se mjerila točnost hipoteze, potrebno je koristiti funkciju troškova. Funkcija troškova pokazuje koliko je model pogrešan u svojoj sposobnosti da procijeni odnos između X i Y. Radi toga, vrijednost te funkcije mora biti što manja, što rezultira tome da ju je potrebno minimizirati. Minimiziranje funkcije troškova obavlja se pomoću gradijentskog spuštanja [7]. Na sljedećoj slici je primjer prikaza gradijentskog spuštanja.



Slika 1. Gradijentsko spuštanje [8]

Parametri q_1 i q_2 su ustvari lokalni minimumi prema kojima se ide ako je ispravno napravljeno gradijentsko spuštanje. Na slici su prikazana dva moguća scenarija, gdje gradijentsko spuštanje ide prema q_1, a drugi (koji je pozicioniran do njega) ide prema q_2. Zbog toga što gradijentsko spuštanje ide prema bilo kojem minimumu, postoji mogućnost da neće doći do globalnog minimuma, nego samo do jednog od lokalnih minimuma. Iz tog razloga je potrebno podešavati parametre kao što su stopa učenja (eng. *learning rate*), količina svojstava, koliko su svojstva važna (smanjiti ih ili ih povećati) i slično. Kada gradijentsko spuštanje dođe na lokalni minimum, više neće mijenjati svoju poziciju. Također, što je bliži lokalnom minimumu, automatski će uzimati manje korake zbog derivacije unutar formule i potrebno ga je ponavljati sve dok se ne dođe na lokalni minimum. Ako se gradijentsko spuštanje koristi za linearnu regresiju, onda se ne mora paziti na to hoće li se spustiti do lokalnog ili globalnog minimuma jer ona ima samo jedan globalni minimum [7].

Ponekad gradijentsko spuštanje neće funkcionirati kako treba. Glavni problemi koji se mogu pojaviti su neispravna stopa učenja, prekomjerno prilagođavanje podacima (eng. *overfit*) i nedovoljno prilagođavanje podacima (eng. *underfit*) iz skupa podataka za treniranje. Problem koji se javlja kod neispravne stope učenja jest taj da gradijentsko spuštanje ne funkcionira, to jest da se ne spušta. Zbog toga treba pokušati uzeti manju vrijednost stope učenja i za testiranje uzeti nekoliko njih. Kada se pojavi overfit, treba pokušati smanjiti broj svojstava ili koristiti regularizaciju. A ako se pojavi underfit, moguća rješenja su povećati broj svojstava i/ili dodavati polinomska svojstva [9].

Regularizacija omogućuje da se ostave sva svojstva, ali smanjuje njihove vrijednosti, što ustvari znači da obeshrabruje učenje kompleksnijeg modela tako da se izbjegne rizik od prekomjernog prilagođavanja [10].

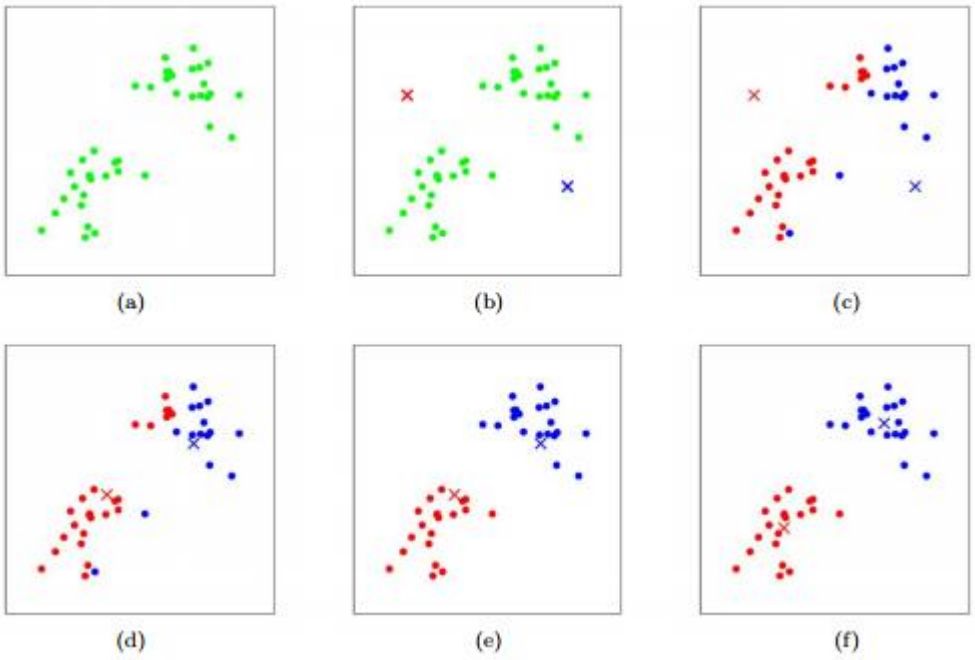
Unutar algoritma proksimalne optimizacije politike (eng. *Proximal Policy Optimization* – u nastavku PPO), koji se koristi u RL-u i koji će se koristiti unutar projektnog dijela, također se koristi gradijentsko spuštanje gdje je potrebno paziti na stopu učenja i probleme vezane za prekomjerno i nedovoljno prilagođavanje podacima.

2.2. Nenadzorno učenje

Ako postoji malo podataka o tome kakvi će biti izlazni rezultati ili se ne zna kako će uopće izgledati rezultati, može se koristiti nenadzorno učenje. U njemu algoritam analizira neoznačene klastere i pokušava ih organizirati u grupe tako da shvati koji su njihovi međusobni odnosi. Unutar nenadzornog učenja nema povratnih informacijama o predviđanju rezultata, nego se pokušavaju otkriti uzorci na određenom skupu podataka. Primjer toga je segmentacija korisnika, genetika, otkrivanje anomalija i slično [11].

Jedni od popularnijih algoritama unutar nenadzornog učenja su K-means, KNN (k-najbliži susjedi, eng. *k-nearest neighbors*), otkrivanje anomalija i neuronske mreže [12].

Na sljedećoj slici se može vidjeti kako funkcionira K-means algoritam.



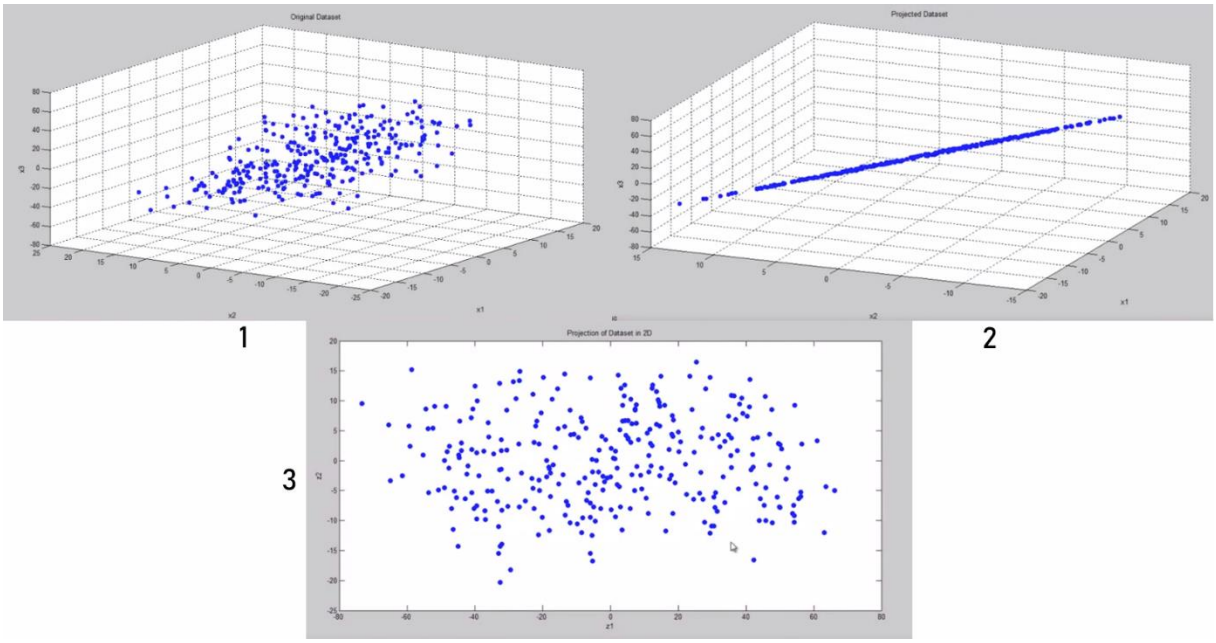
Slika 2. K-means algoritam [13]

Na a) slici je prikazan skup podataka kojeg je potrebno organizirati u dva klastera. Kada bi se K-means algoritam pokrenuo, nasumično bi se odabrala dva mjesta gdje će postaviti točke naziva *cluster centroid*. To se može vidjeti na slici b). To je iterativni algoritam i radit će samo dvije sljedeće stvari:

1. Dodjela klastera
2. Pomicanje centroidnog koraka

Dodjeljivanje klastera funkcionira na način da prolazi kroz svaki podatak (zeleno označene točke) i ovisno o tome koja je točka bliža centroidu, označit će se i može se vidjeti na slici c). Drugi korak jest pomicanje centroida tako da se izračuna prosječna lokacija svih podataka od jednog centroida i tada pomakne centroid prema njima. Ako se gleda slika c), prosječna lokacija crvenih točaka je prema dnu, dok prosječna lokacija plavih točaka je prema gore, što znači da će se centroidi pomaknuti prema tim lokacijama (slika d)). Nakon toga se ponovno dodjeljuju klasteri i pomiču centroidi, sve dok ne konvergiraju. Finalni rezultat gdje su centroidi konvergirali se može vidjeti na slici f), gdje je algoritam napravio dva klastera od neoznačenih podataka [13].

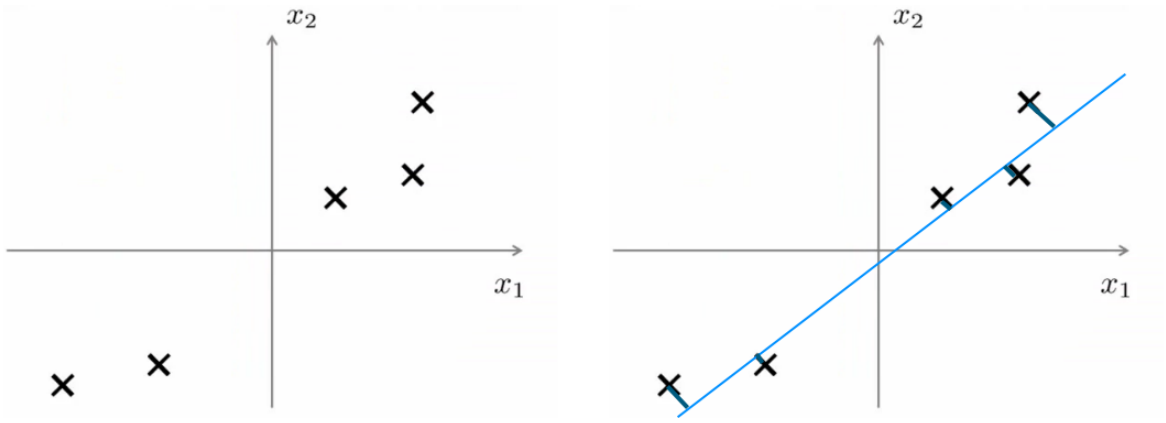
Problem koji se javlja prilikom nenadzornog učenja jest smanjenje dimenzionalnosti (eng. *dimensionality reduction*). Rješenje smanjenja dimenzionalnosti je moguće riješiti s kompresijom podataka, gdje se primjerice podaci prebacuju iz trodimenzionalnih u dvodimenzionalne. Primjer toga se može vidjeti na sljedećim slikama:



Slika 3. Smanjenje dimenzionalnosti (3D u 2D) [14]

Pretpostavka je da postoje podaci kao u podslici 1. Okretanjem osi, može se vidjeti da svi podaci leže u jednoj ravnini (podslika 2). Označi li se ta ravnina sa z_1 i z_2, tada se nju može prebaciti iz trodimenzionalne u dvodimenzionalne (podslika 3).

S time se bavi algoritam analiza glavnih komponenti (eng. *Principal Component Analysis*, skraćeno PCA). Prije nego što se započne s algoritmom, potrebno je provesti srednju normalizaciju i skaliranje svojstava. Još jednostavniji primjer bi bilo smanjivanje dimenzionalnosti iz 2D u 1D.



Slika 4. PCA [15]

Na slici 3 se s lijeve strane može vidjeti skup podataka s kojim se raspolaže u 2D-u. PCA pokušava pronaći manju dimenziju tako da je greška projekcija (prikazana desno na slici 3. s tamnoplavom bojom kod X-eva) minimalna. Nakon toga označi liniju koja ima minimalnu grešku projekcije i tada se skup podataka može prikazati na jednodimenzionalnoj površini (slika u nastavku).

Slika 5. 2D PCA [15]

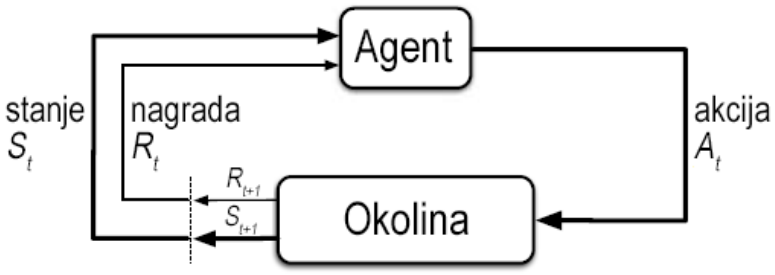
PCA zbog smanjenja dimenzionalnosti, omogućuje smanjenje memorije i prostora na disku te ubrzava algoritam učenja. Osim toga, omogućuje da se podaci lakše prikazuju grafički [15].

2.3. Podržano učenje

Podržano učenje, kao dio strojnog učenja, je treniranje modela da donosi određene akcije. Kada se osoba prvi put susretne s određenom igrom, primjerice nekom first-person shooter, tada će morati naučiti kontrolirati igrača tipkovnicom i mišem. Nakon nekoliko pokušaja, osoba će shvatiti kako pokretati igrača i ciljati. Ako još dulje nastavi igrati, tada više neće morati razmišljati o tome kako pomicati igrača, nego će razmišljati o nekim drugim svrhama igre. Cilj RL-a je točno to, imati inteligentnog agenta koji u okolini donosi akcije maksimizirajući nagradu. Agent predstavlja igrača kojim upravlja računalo. On ne zna koje akcije treba poduzimati, nego ih treba samostalno saznati i koristiti one koje mu daju najveće nagrade. Nagrada se obično promatra kao kumulativna, a može biti pozitivna i negativna. Ako agent napravi nekoliko akcija koje dovede do dobrog rezultata, dobit će pozitivnu nagradu, a inače negativnu. Pomoću kumulativne nagrade, odnosno zbroja svih pozitivnih i negativnih nagrada koje je agent dobio u nekom vremenskom roku, se shvaća poduzima li agent ispravne akcije ili ne [16].

Jedan od primjera je upravljanje zrakoplovom, gdje je cilj da agent nauči održavati zrakoplov u zraku. Agent predstavlja pilota, akcije koje može poduzeti su na kontrolnoj ploči zrakoplova, a okolina je prostor u kojem leti sa zrakoplovom. Ako se zrakoplov sruši na tlo, tada će agent dobiti negativnu nagradu. Ako uspije održavati zrakoplov, tj. donijeti određene akcije koje će održati zrakoplov u zraku, tada će dobiti pozitivnu nagradu. Nakon brojnih pokušaja i grešaka, agent će shvatiti koje akcije dovode do rušenja zrakoplova, a koje omogućuju da zrakoplov leti, pa će nakon nekog vremena naučiti letjeti.

Problem RL-a se nalazi u tome kako optimirati Markovljev proces odlučivanja (eng. *Markov Decision Processes – MDP*). Metoda koja nudi rješenje konačnih MDP se smatra da je RL metoda [16].



Slika 6. Interakcija između agenta i okoline [16]

Na slici su prikazani elementi RL, odnosno način funkcioniranja agenta i okoline. Svakim korakom *t* agent dobiva određeni prikaz stanja okoline (opservacija okoline), pri čemu radi određene akcije unutar okoline. Nakon toga dobiva nagradu iz okoline i nova stanja prema

kojima agent odlučuje koje akcije treba napraviti. Svakim korakom agent provodi preslikavanje iz stanja u vjerojatnosti odabira svake moguće radnje. To preslikavanje se naziva politika (eng. *policy*). RL metode specificiraju kako agenti mijenjaju svoju politiku kao rezultat svojeg iskustva. Cilj agentu, kao što je već rečeno, prema tome jest, maksimiziranje ukupne vrijednosti kumulativne nagrade.

Nekoliko spomenutih metoda, a koje rješavaju MDP, su dinamično programiranje (eng. *dynamic programming*), Monte Carlo metoda i učenje s vremenskom razlikom (eng. *Temporal-Difference Learning*).

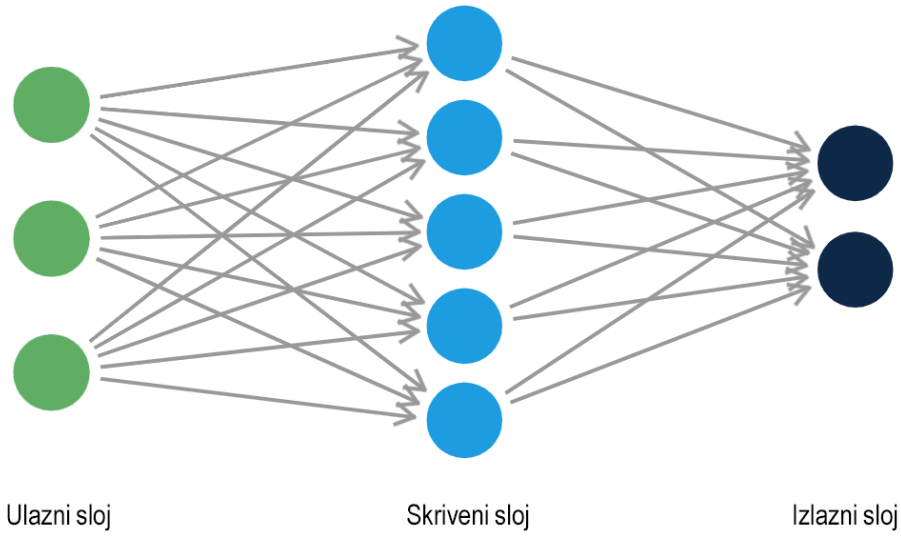
Dinamično programiranje (u nastavku DP) je skup algoritama koji se mogu koristiti za računanje optimalnih politika s obzirom na dani savršeni model okoline kao MDP. Da bi model odgovarao, mora biti potpun i točan prema okolini. Monte Carlo metode su jednostavne, ne trebaju model kao kod DP-a, ali nisu prikladne za računanje korak po korak. Učenje s vremenskom razlikom su najkompleksnije u odnosu na prošle dvije metode, ali mogu se inkrementirati korak po korak i ne trebaju imati model. Svaki od modela ima svoje pozitivne i negativne strane i, naravno, sve ovisi o tome o kojem se problemu radi [16].

Primjer RL-a u praksi je OpenAI. OpenAI je 2019. godine prvi AI program koji je uspio pobijediti svjetske esports prvake u igri Dota2. Dota 2 je multiplayer igra, žanra MOBA (eng. *multiplayer online battle arena*) gdje se dva tima natječu u osvajanju protivničke baze. U igri trenutno postoji 121 heroj i svaki od njih ima poseban dizajn, napade, jačine i slabosti. Osim heroja, unutar igre postoje podanici (eng. *minions*) kod svakog tima koji napadaju protivničke jedinice ili tvrđave. Njih nije moguće kontrolirati. Jedna runda traje otprilike 40 minuta te je igra veoma kompleksna, a svake godine se na svim natjecanjima zaradi skoro 40 milijuna američkih dolara [17]. Korištenjem RL, odnosno algoritma PPO koji će se također koristiti unutar ovog rada, su uspjeli pobijediti svjetske prvake. Agent mora opservirati brojne stvari u igri, što znači da računalo koje pokreće RL mora biti u mogućnosti podnijeti takve zahtjeve. OpenAI koristi Google Cloud Platformu s 128.000 CPU jezgri i 256 P100 GPU te mora opservirati oko 16.000 vrijednosti (u projektnom zadatku će agent opservirati samo 94 vrijednosti) u rasponu od 40-ak minuta (koliko runda traje). OpenAI nije igrao sa svim herojima, nego samo sa 17 i odlučili su izbjegavati korištenje predmeta kojima se mogu upravljati drugi heroji. Ono što je OpenAI napravio u jednom danu, za to isto bi čovjeku bilo potrebno oko 180 godina. Isprva je OpenAI igrao sam protiv sebe i na temelju toga učio kako igrati, a kasnije je igrao protiv pravih igrača koji su se prijavili putem *communitya*. Protiv stvarnih osoba, OpenAI je imao 99.4% stopu pobjede na preko 7.000 igara, a 2019. godine su se susreli s tada najboljim Dota2 timom, Team OG, i onda su imali preko 10.000 godina iskustva. OpenAI je pobijedio dvije od tri igre i s time postao prvi AI program koji je pobijedio svjetskog prvaka u Dota2 [18].

2.4. Neuronske mreže

Jedna od važnijih tema su i neuronske mreže koje se koriste u algoritmima strojnog učenja. Neuronske mreže se mogu povezati s ljudskim mozgom jer je prepun malih stanica zvanih neuroni. Neuron ima stanično tijelo te sadrži određeni broj ulaza koji se nazivaju dendritima. Dendriti primaju ulazne podatke s drugih lokacija. Neuron, također, ima izlaz zvan akson, a on služi za slanje signala drugim neuronima. Način na koji neuroni međusobno komuniciraju je putem malih impulsa električne energije. Dakle, pojednostavljeno rečeno, neuron je računska jedinica koja dobiva niz ulaza, nad njima vrši računanje, a zatim izlazne podatke šalje pomoću aksona do drugih čvorova ili do drugih neurona u mozgu [19].

Na sljedećoj slici se može vidjeti prikaz neuronske mreže.



Slika 7. Neuronske mreže [20]

Na slici se može vidjeti da postoje tri ulaza neuronske mreže koja su označena sa zelenom bojom. Oni predstavljaju dendrite. Tada neuron radi izračun i donosi neku vrijednost na svojem izlazu i ona je prikazana sa strelicom. Ta se strelica može poistovjetiti s aksonom u biološkom neuronu [19]. Neuronska mreža je ništa drugo nego skup takvih različitih neurona zajedno. Prvi sloj se naziva ulaznim slojem, dok se završni sloj naziva izlaznim slojem. Sve između ulaznog i izlaznog sloja jest skriveni sloj [21].

Umjetna neuronska mreža ima sposobnost učenja, a taj postupak podrazumijeva iterativno podešavanje težinskih faktora na osnovu pogreške izračunate vrijednosti modela i stvarne vrijednosti. Učenje se odvija prema algoritmu širenja unatrag (eng. *back propagation*). Kada informacija jednom prođe kroz neuronsku mrežu, dobije se vrijednost koja se kasnije uspoređuje sa stvarnom vrijednošću. Na temelju razlika između stvarne i izračunate

vrijednosti, korigiraju se težinski faktori. Kako se težinski faktori korigiraju, tako neuronska mreža uči predviđati stvarne vrijednosti i tako se smanjuju razlike između stvarnih i izračunatih vrijednosti. Također, prilikom učenja u neuronskim mrežama, potrebno je paziti na prekomjerno prilagođavanje podacima i na nedovoljno prilagođavanje podacima [21].

Primjer korištenja neuronskih mreža je prepoznavanje slika gdje je pomoću algoritma širenja unatrag moguće shvatiti radi li se o psu ili mački na slici. Potrebno je dati označene slike na kojima piše gdje se nalaze psi, a gdje mačke (spomenuto je prije kao nadzorno učenje). Tada će se na svakom neuronu napraviti određeni zadaci gdje će na kraju algoritam doći do zaključka radi li se o psu ili mački [22].

3. Unity

Engine za igre kreiran od strane Unity Technologies omogućava kreiranje 2D ili 3D igara. Osim toga omogućuje brojne platforme kao što su Windows, iOS, Android, WebGL i slično. Također, omogućuje razvoj proširene stvarnosti (eng. *augmented reality*) i virtualne stvarnosti (eng. *virtual reality*) [23]. Prva verzija je izašla 2005. godine, a moguće ga je koristiti besplatno ako se koristi individualna licenca i zarađuje manje od 100.000 dolara u godini dana. Podupire programske jezike C#, C++, JavaScript, Boo, Lua, ali najviše se koristi C#. Unutar ovog projekta, korišten je C# pomoću programa Visual Studio [24].

Prilikom otvaranja novog projekta, kreira se scena unutar koje je moguće stavljati 3D objekte. Svaka scena sadrži glavnu kameru i svjetlost te ih je moguće pomicati po želji. Svakom objektu koji se kreira moguće je dodijeliti komponente i, između ostalog, skripte. Unity već ima napravljene jednostavne objekte, kao što su kvadri, kugle, sfere i slično te sadrže default komponente bez dodatnog korisničke izmjene. Takav način kreiranja scene ubrzava cijeli proces kreiranja i olakšava korisniku rad unutar programa.

Ono što je potrebno znati za ovaj projekt jesu tipovi podataka koji se javljaju unutar Unitya. Svaki objekt u Unityu ima svoju poziciju u prostoru pod nazivom *Transform*. Podatak *Transform* sadrži poziciju, rotaciju i mjerilo objekta. Pozicija je ustvari trodimenzionalni vektor, oznake Vector3, koji se sastoji od tri realna broja (X, Y, Z). Rotacija je tip podatka Quaternion i također je trodimenzionalna i rotira se ovisno o osi (X, Y, Z). Ovi tipovi podataka su važni za izračunavanje opservacija unutar biblioteke ML-Agenta što je kasnije objašnjeno u radu.

3.1. ML-Agents

ML-Agents jest open-source biblioteka unutar Unitya koja omogućuje implementiranje agenata koji funkcioniraju pomoću RL-a baziranom na Pytorchu. Agenti se mogu trenirati na brojne načine i mogu se implementirati unutar 2D, 3D i VR/AR igara. Prema službenoj dokumentaciji postoje dva podržana algoritma RL-a. To su *Proximal Policy Optimization (PPO)*, *Soft Actor-Critic (SAC)* i MultiAgent Posthumous Credit Assignment (*MA-POCA)*. Po defaultu je algoritam PPO i taj algoritam se koristi unutar projektnog dijela rada [25]. Sve vezano uz PPO algoritam radi biblioteka ML-Agents i nije potrebno samostalno programirati te dijelove.

3.1.1. YAML datoteka

Jedino što je potrebno napraviti jest urediti parametre koji se koriste prilikom učenja. Zbog toga postoji YAML datoteka unutar konfiguracije ML-Agentsa. Primjer default YAML datoteke se nalazi unutar sljedećeg koda.

```
behaviors:
  Basic:
    trainer_type: ppo
    hyperparameters:
      batch_size: 32
      buffer_size: 2048
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 20
      num_layers: 1
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.9
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 500000
    time_horizon: 3
    summary_freq: 2000
```

Prvi element poslije naziva *behaviors* jest *Basic.* Taj naziv predstavlja trenutnog agenta, odnosno model na kojem agent uči. Svaki mora biti jedinstven i ne smije biti duplikata unutar Unitya. Nakon toga zadaje se algoritam koji se koristi, u ovom slučaju PPO te njegovi parametri. Neki parametri su zajednički svim algoritmima za učenje, ali njihove postavke, naravno, moraju biti različite. *Batch_size* određuje broj iskustava u svakoj iteraciji gradijentnog spuštanja. Što se tiče PPO algoritma, s kontinuiranim akcijama iznosi između 512 i 5120, a što se tiče diskretnih akcija, trebao bi iznositi između 32 i 512. Ova vrijednost je povezana s

buffer_size i trebala bi biti nekoliko puta manja od njega. *Buffer_size* je broj iskustava koji se moraju obuhvatiti prije ažuriranja modela. Tipična vrijednost za PPO jest između 2048 i 409600, gdje veći broj omogućuje stabilnije treniranje. Nakon toga dolazi *learning_rate* koji pokazuje koliko se gradient descent spušta svakim korakom. Preporučeno je da se smanji ako je trening nestabilan ili ako nagrada ne raste konstantno i da je njegova vrijednost između 0.00001 i 0.001. Parametar *beta* se koristi samo za algoritam PPO i pokazuje koliko su agentove akcije nasumične. On je važan parametar kojeg je potrebno pravilno konfigurirati da agent istraži što više okoline. Obično je njegova vrijednost između 0.0001 i 0.01. Sljedeći parametar jest *epsilon* (vrijednosti između 0.1 i 0.3), a utječe na to koliko se politika može razvijati tijekom treninga. Ako se *epsilon* postavi kao mala vrijednost, bit će stabilniji trening, ali i sporiji. Nadalje, *lambd* je parametar koji se koristi za regulariziranje funkcije troškova. Ako je zadan kao manja vrijednost, tada postoji mogućnost visoke pristranosti (eng. *high bias*), a ako je zadan kao viša vrijednost, onda postoji mogućnost visoke varijance (eng. *high variance*). Njegova vrijednost varira između 0.9 i 0.95. Nadalje, postoji parametar *num_epoch*, koji označava koliko je prolaza potrebno napraviti kroz međuspremnik iskustva (*buffer*) da bi se napravila optimizacija gradijentnog spuštanja. Smanjene vrijednosti će omogućiti stabilnije treniranje, ali i sporije.

Nakon toga su prikazani parametri za mrežne postavke (eng. *network settings*). Prvi parametar je *normalize* koji služi za potrebu normalizacije vektora opservacije. Preporučuje se njegovo korištenje prilikom kompleksnih i kontinuiranih akcija. Sljedeći parametar jest *hidden_units* koji označava koliko skrivenih jedinica postoji unutar neuronske mreže. Zadana vrijednost je 128. Parametar *num_layers* pokazuje broj skrivenih slojeva unutar neuronske mreže i njegova zadana vrijednost je 2, a zadnji parametar unutar mrežnih postavki jest *vis_encode_type* koji predstavlja tip enkodiranja za vizualne opservacije.

Nadalje, dolaze parametri vezani uz nagrade koje agent dobiva. Prvi elemenat jest *strength* koji pokazuje faktor s kojim se množi nagrada dobivena iz okoline (npr. scena iz Unitya). A drugi elemenat jest *gamma*, odnosno faktor popuštanja za buduće nagrade koje dolaze iz okoline. Njegova zadana vrijednost jest 0.99.

Sljedeća grupa parametara je vezana za trajanje treniranja i njegovu analizu. Tako, parametar *keep_checkpoints* je maksimalni broj koliko treba ostaviti kontrolnih točaka unutar modela. Nakon određenog broja koraka, spremaju se kontrolne točke, a taj određeni broj koraka se određuje parametrom *checkpoint_interval*. Sljedeći parametar jest *max_steps* koji pokazuje koliko je koraka moguće napraviti u treniranju. Nakon što broj koraka prilikom treniranja dođe do tog broja, zaustavit će se treniranje. Da bi agent mogao učiti, potrebno je dodavati njegovo iskustvo u međuspremnik iskustva. Koliko koraka ulazi u međuspremnik određuje varijabla *time_horizon* i njegova zadana vrijednost jest 64. I, zadnji parametar unutar

ove YAML datoteke, je *summary_freq* koji određuje broj iskustva koji trebaju biti skupljeni da bi se prikazala njegova statistika.

Osim prikazanih, postoji još puno parametara koji se mogu koristiti. Neki ovise točno o algoritmu koji se koristi, a neki služe za još detaljnije i kompleksnije treniranje [26].

3.1.2. Proksimalna optimizacija politike

PPO je jedan od najuspješnijih algoritama unutar RL metoda i korišten je unutar projektnog dijela rada. Algoritam koristi neuronsku mrežu za približavanje idealnoj funkciji koja mapira opservacije agenta u najbolju akciju koju agent može donijeti u određenom stanju. PPO algoritam je implementiran unutar Tensorflowa koji komunicira s Pythonovim procesima [27]. Detaljnije, taj algoritam služi za optimiziranje ažuriranja koraka unutar gradijentskog spuštanja, gdje se pokazao kao bolje rješenje od standardnog gradijentskog spuštanja. Omogućuje stabilnija i brža treniranja [28].

Algoritam PPO se koristi unutar ovog rada jer omogućuje stabilnije treniranje i lakše implementiranje pomoću YAML parametara te je najviše korišten unutar ML-Agentsa.

4. Sigurnost i varanje unutar računalnih igara

Multiplayer igre, gdje nekoliko igrača igra istovremeno preko mreže, bilo lokalne ili interneta. Igre više nemaju samo svojstvo zabave, već da se na njima napravi karijera, od streamera pa sve do profesionalnih igrača. Elektronički sport (*eSports*) postaje sve popularniji i igra sve više igrača koji žele dobiti prednost nad drugima nedopuštenim načinima.

Igrač može varati na brojne načine i to ne mora biti samo korištenjem nedopuštenih programa. Na primjer, igranje u dosluhu s drugim igračem da se ostvari prednost. Takvo igranje se vidjelo unutar igre StarCraft gdje se nekoliko igrača udruživalo da ostvare bolji rang. Osim u StarCraftu, slično je postojalo u igri League of Legends. Prije je bilo moguće u visokim rangovima igrati s prijateljima gdje tada taj tim ima veće šanse da pobijedi od drugog koji, na primjer, nema nijednog *premade* igrača. Trenutno je to zabranjeno u visokim rangovima i nije moguće igrati s prijateljem. Nadalje, moguće je varati izmjenom konfiguracija unutar igre. U multiplayer igrama to baš i nije moguće, ali ranije u singleplayer igrama su se mogle mijenjati konfiguracije da bi se dobila prednost unutar igre. U multiplayer igrama igrači ponekad znaju na kraju izaći iz igre ako gube. Neke igre to ne kažnjavaju, ali većinom u timskim igrama se mogu dobiti zabrane igranja i slično. Sve ovisi o tome kakva je igra i o čemu se radi. Igra kao *Counter Strike* ima predmete (eng. *skins*) koji se mogu koristiti prilikom igranja i svaki ima svoju vrijednost u novcu. Zato se neke osobe žele domoći tih predmeta kako bi ostvarile profit, pa se onda dolazi do varanja vezanog s virtualnom imovinom [29].

Postoji još puno mogućnosti varanja, ali unutar ovog rada se ističe varanje pomoću umjetne inteligencije (eng. *Artificial Intelligence*, skraćenica AI). Primjerice, cilj MMORPG (eng. *massively multiplayer online role-playing game*) igara jest da se dođe do što više razine da bi igrač bio što jači. U nekim igrama to može trajati po nekoliko stotina sati, ako ne i tisuća. Pretpostavka je da se nekom igraču ne da igrati toliko dugo, nego će napraviti AI bota koji će umjesto njega obavljati određene stvari. Skupljanje iskustva, razina, skupljanje oružja, skrivenih tajni i slično. Tako igrač može ostaviti upaljen program dugo vremena i ostvariti prednost. Naravno, onaj tko je radio igru, ne želi da netko s takvim načinom dobije prednost nad nekim drugim igračem koji igra samostalno, pa pokušava napraviti određene programe koji će to pratiti i pokušavati raspoznati koriste li se nedopuštena sredstva.

Zato postoji, primjerice, VAC (eng. *Valve Anti-Cheat System*), a predstavlja automatski sustav koji pronalazi korištenje nedopuštenih programa instaliranih na računalima korisnika. Korištenjem *third-party* programa koji pomažu igraču da ostvari prednost nad drugima će rezultirati zabranom igranja te igre. Također, to uključuje i izmjenu izvršnih datoteka igre. VAC je napravila kompanija Valve i podržava više od sto igara unutar platforme Steam [30]. Sličan

način sprečavanja varanja se može vidjeti kod ESL-a (eng. *Electronic Sports League*) gdje postoji ESEA client (eng. *E-Sports Entertainment Association League*) koji pruža igračima da skinu program i igraju Counter-Strike s prijateljima ili profesionalno. ESEA također ima implementiran sustav za detekciju varanja, ali također ima napravljeno rangiranje gdje svaki igrač ima određeni rang i tada može igrati samo s igračima koji imaju jednak rang [31].

5. Povezani radovi

Postoji nekoliko radova vezanih uz detekciju varanja unutar multiplayer igara pomoću strojnog učenja. Jedan od radova proučava samo logove koje igra radi, što znači da se ne pronalazi igrač koji vara na način da se zadire u njegovu privatnost, nego se proučavaju datoteke koje se zapisuju prilikom igranja na server. Oni koriste nadzorno učenje, ali prije nadzornog učenja moraju imati skup podataka na kojem se model mora trenirati. Također, prilikom čitanja logova, moraju izvući najbitnija svojstva koje će strojno učenje koristiti [32]. Unutar ovog projektnog dijela, gdje se koristi RL, to se ne mora raditi. Pomoću RL-a, agent će proučavati igrače i, pomoću određenih nagrada, shvatiti tko vara.

Sljedeći rad pokazuje detekciju varanja pomoću ishoda igara. Predložili su općeniti okvir za nekoliko tipova igara, gdje je potrebno znati trenutni rang igrača. Pretpostavlja se da igrač ima umjetno povećan rang i, pomoću izračuna vjerojatnosti, ga pokušavaju otkriti [33]. Kod igre *Courier* ne postoje rangovi i problem koji se ovdje rješava pomoću RL-a je složen na način da bude što više generaliziran i učinkovit za implementiranje unutar Unity igara.

Posljednji navedeni rad, veoma sličan ovome, jest pronalaženje botova unutar igre Quake II tako da se analiziraju njihove putanje. Rad se bazira na tome da igrač gotovo nikada neće koristiti sličnu ili identičnu putanju, dok bot hoće i pomoću tih postavki se pokušava pronaći razlika između igrača i bota [34]. Problem je sličan kao i unutar ovog rada, ali postoje neke razlike. Naime, učenjem RL agenta je moguće složiti da detektira stvarnog igrača kojemu pomaže umjetna inteligencija. Agent može shvatiti da igrač ima puno brže reakcije od drugih igrača, da igrač raspolaže podacima s kojima drugi igrači ne raspolažu i slično. Potrebno je pravilno postaviti problem i okolinu na kojoj bi agent mogao to naučiti. Unutar ovog rada se neće detektirati igrači koji koriste nedopuštena pomagala, ali bi se agent sasvim sigurno mogao prilagoditi tome da ih pronađe.

6. Implementacija sustava protiv varanja

Cilj ovog projekta je omogućiti računalu da samostalno odredi koristi li igrač umjetnu inteligenciju da dobije prednost nad drugim igračima. Puno igara ima problema s time gdje igrači osposobe bota da igra umjesto njih i na taj način dobivaju prednost nad drugim igračima, na primjer, skupljanje novaca u nekoj od igrica. Radi toga, napravila se igra u kontroliranoj okolini gdje će se implementirati RL agent koji će moći razlikovati bota od igrača.

6.1. Ideja i razvoj

Prilikom kreiranja početnih skica aplikacije i razmišljanja o implementiranju RL-a, odlučeno je da se samostalno napravi jednostavna igra unutar koje će se implementirati RL. Ukratko, postojao bi sigurnosni agent koji može raspoznavati bota od stvarnog igrača. Koraci unutar kreiranja projekta su bili:

- Izgradnja multiplayer igre
- Implementiranje agenta za sigurnost (ML-Agents)
- Treniranje agenta
- Prikaz rezultata

Poznato je da agentu, prilikom korištenja RL-a, treba veća količina vremena da nauči neke radnje. Da bi ispravno agent razlikovao bota od igrača, potrebno je obaviti treniranje gdje stvarna osoba igra zajedno s botovima. Ako jedna runda u igri traje oko minutu, a potrebno je odigrati nekoliko tisuća rundi da agent može ostvariti svoj cilj, osobi to ne bi bilo vremenski prihvatljivo. Jedino ako se radi o već nekoj poznatoj igri koju igra veliki broj ljudi, što tada ne bi bio problem. Zbog toga, jedan od ciljeva prilikom kreiranja igre jest da runde traju relativno kratko.

Ideja izgradnje igre je bila da se napravi jednostavna igra, bez kompliciranih logika koja će služiti kao dobar predložak. Ovakav koncept agenta može se implementirati u bilo koju igru. Iz toga je proizašla ideja da se napravi igrač koji bi nosio stvari s jedne strane na drugu. Da ne izgleda dosadno, napravljen je dizajn mape i dodano topništvo čije projektile je potrebno izbjegavati. Iz toga je nastala igra *Courier* gdje igrač izgleda kao veliki tic-tac i nosi kutije. Baš zbog toga što je glavni cilj igre prenositi kutije, dobila je naziv kurir. Glavni koraci prilikom kreiranja igre su bili sljedeći:

- Dizajniranje i kreiranje okoline (mape)
- Logika i fizika za kretanje igrača
- Topništvo

- Interakcija s objektima (projektili, voda, kutije)
- Implementacija multiplayera
- Implementacija botova

Prvo je kreirana singleplayer igra u kojoj su testirani pokreti igrača, nošenje kutija, rezultati, topništvo i voda. Isprva je na igraču bila CinemaMachine kamera koja omogućuje glatko praćenje igrača, bez da kamera prolazi kroz zidove i slično, ali nakon implementacije multiplayera, nije bilo moguće implementirati CinemaMachine zbog velikih zastajkivanja (eng. *lag*). Radi toga, nakon što je multiplayer ubačen u igru, složila se obična kamera koja prati igrača.

Igra se sastoji od serverskog i klijentskog dijela te je jedino testirana na Windows platformi i lokalnoj mreži. Više o serveru može se vidjeti u poglavlju Multiplayer.

Trenutno postoji nekoliko bugova, kao što je bacanje kutija kroz zidove, nepravilni prikaz kamere blizu zidova, nepostojanje tekstura i slično. Mogla bi se doraditi, možda i hoće u budućnosti, ali s obzirom na to da je igra rađena zbog implementacije RL-a, nije bilo potrebno.

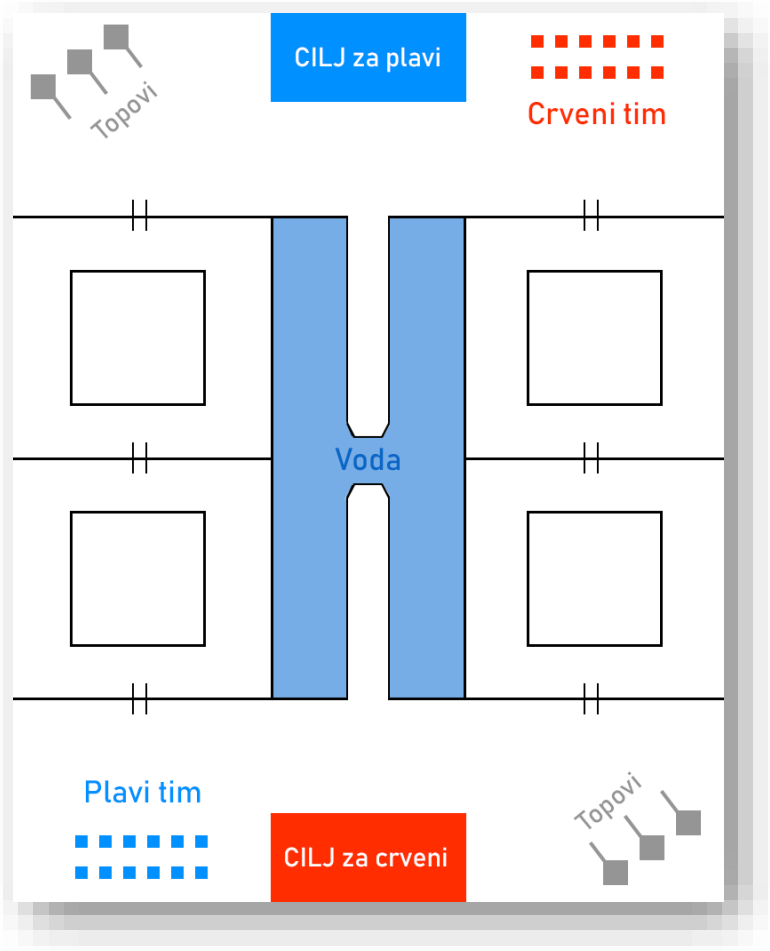
6.2. Korišteni programi

Engine koji pokreće igru jest Unity. Korišten je većinom zbog toga što podržava brojne biblioteke, može se koristiti programski jezik C# i omogućuje jednostavan export projekata na gotovo sve platforme. Verzija Unitya koja se koristi unutar projekta jest 2020.2.1f1. Razlog korištenja starije verzije Unitya jest ta da je paket ML-Agents isproban i testiran unutar te verzije, dok instalacijom nove verzije neke postavke nisu ispravno funkcionirale. Da bi se izbjegli određeni bugovi, ostavljeno je na 2020.2.1f1 verziji prilikom kreiranja projekta. Programski jezik C# koji se koristi s Unityem, pokreće se pomoću programa Visual Studio 2019 verzije 16.11.1, koja u vrijeme pisanja rada bila zadnja verzija Visual Studija.

Nakon toga, paket ML-Agents koristi stabilnu release verziju 18. Da bi ML-Agents mogao normalno funkcionirati, potreban mu je Python i Ptyhonova biblioteka PyTorch. Verzija Python jest 3.8.5, dok biblioteke PyTorch jest 1.9.0.. Također obje verzije su zadnje verzije tih izdanja prilikom pisanja rada. Za jednostavniju instalaciju paketa i korištenja Pythona, koristi se komandna linija programa Anaconda verzije 1.7.2.. Za pokretanje treniranja modela i startanja servera za Tensorboarda koristi se Anaconda. Tensorboard će se koristiti za prikaz i analizu rezultata, jer omogućuje odličan prikaz grafova i podataka. Verzija Tensorboarda jest 2.4.0..

6.3. 3D igra *Courier*

Courier je jednostavna multiplayer igra unutar koje postoje dva protivnička tima. Cilj igre jest prebaciti pet kutija iz svoje baze na suprotnu stranu prije protivničkog tima. Igrači se stvaraju na predodređenim pozicijama koje se nalaze pokraj kutija. Prilikom toga, cijelo vrijeme topništvo puca nasumično prema smjeru u kojem je okrenuta. Tri topa se nalaze na svakoj strani, gdje dva topa pucaju periodom od 0.4 sekunde, dok treći top puca periodom od 0.7 sekundi. Identični periodi su stavljeni na obje strane, što osigurava izjednačenost između timova. Neovisno gdje se nalazi topništvo, uvijek može pogoditi igrača iz bilo kojeg tima. Kada se igrači stvore na početnim pozicijama, ne može ih pogoditi topništvo. Nakon tri sekunde, projektil nestaje i postoji mogućnost da pogodi više igrača odjednom, odnosno ne nestaje nakon kolizije s drugim objektom.



Slika 8. Skica igre [autorski rad]

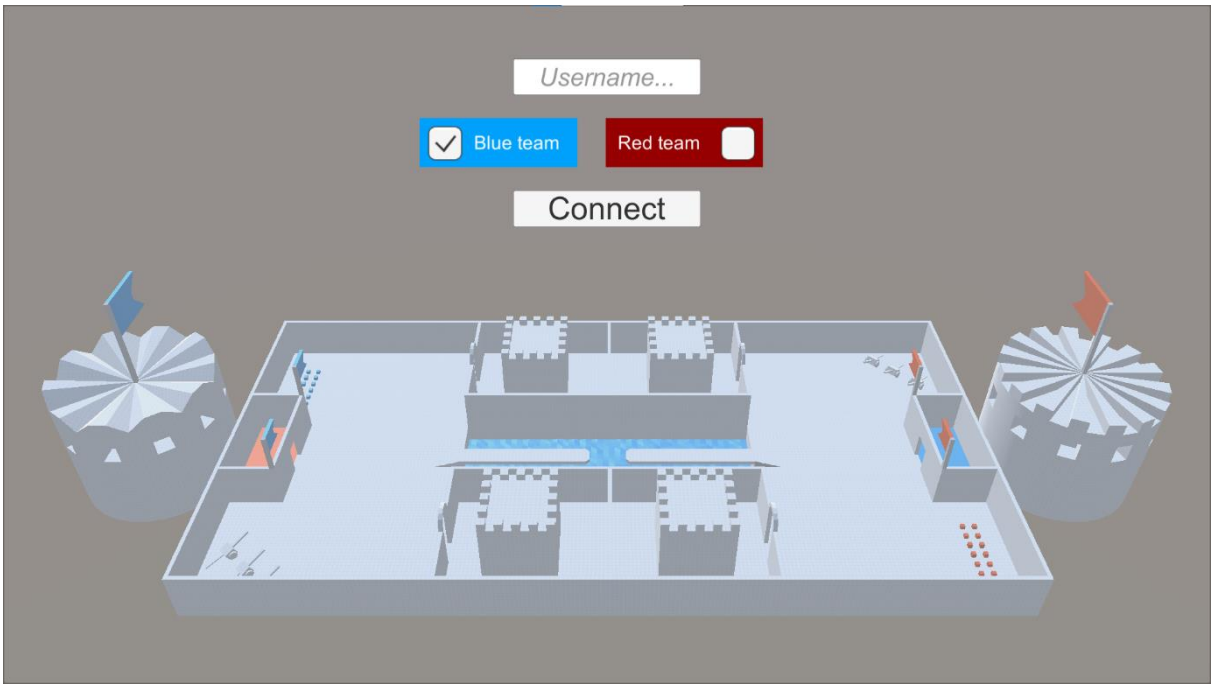
Unutar sredine mape, nalazi se voda, koja prilikom doticaja s igračem, vraća ga na početnu poziciju. Također, ako igrača pogodi projektil, vraća ga na početnu poziciju. Na desnoj i lijevoj strani postoje prolazi (označeni s dvije crtice na skici), te su zagrađeni sa zidovima što

omogućuje igraču da bude zaštićeniji od projektila. Prilikom korištenja mosta na sredini mape, potrebno je napraviti skok kako bi se prešla voda, prilikom čega igrača cijelo vrijeme gađa topništvo. Sigurniji, ali sporiji put, bi bio korištenje prolaza, dok brži, ali riskantniji put, bi bio preko vode.

Mapa je slagana pomoću biblioteka ProGrids i ProBuilder. ProGrids kreira gridove na cijeloj sceni i omogućava brzo i jednostavno postavljanje objekata na mapu. Grid na sceni nikada ne mijenja poziciju niti orijentaciju, što znači da se uvijek zna koliko su objekti udaljeni jedni od drugih i koliko se daleko pomiču [35]. ProBuilder zato omogućuje kreiranje bilo kakvih oblika. Potrebno je samo unijeti podatke kao što su visina i dužina i pomoću jednog gumba kreira željenu podlogu. Svaki dio oblika se može pomicati nakon što se objekt kreira, kao što je slaganje rubova, brisanje dijelova, dodavanje novih elemenata i slično. Također se može samo napraviti pola mape i zrcalno napraviti drugi dio mape, što je u ovoj igri i napravljeno. Osim toga, mogu se slagati prilagođene kolizije objekata, kompleksni tereni, strukture, trigeri i NavMeshevi [36].

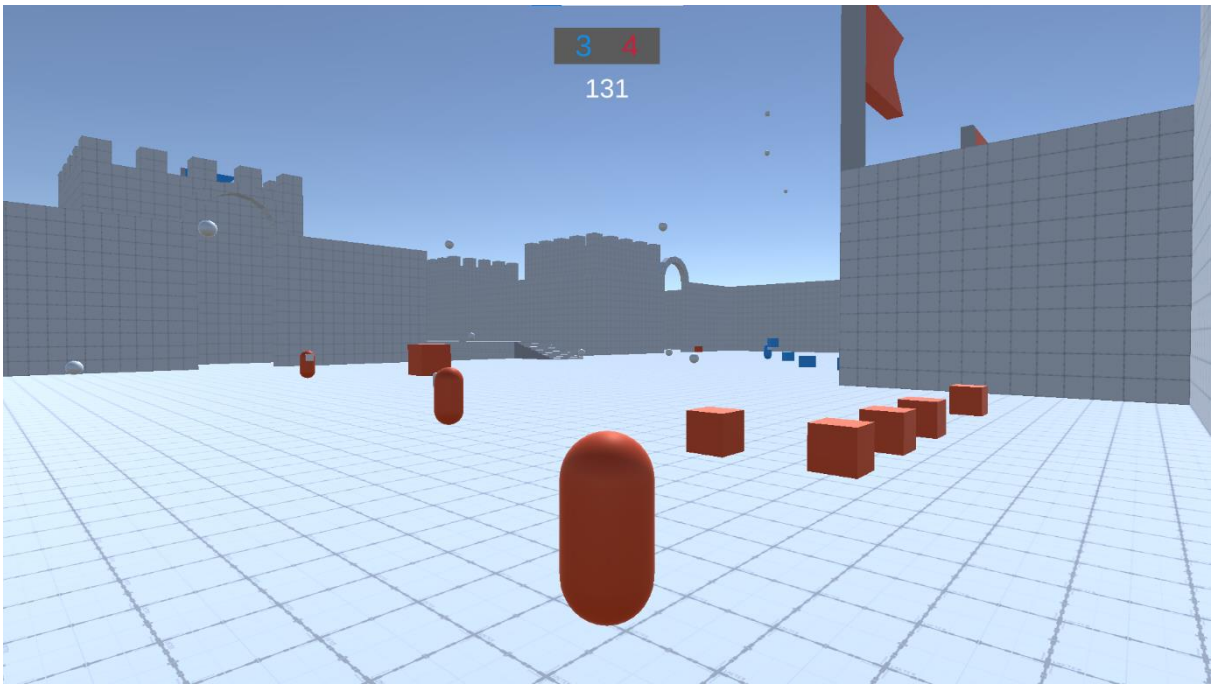
Svaki tim se sastoji od maksimalno pet igrača, a svaki igrač je ustvari jedna 3D kapsula koja na sebi ima mali kvadar da bi se znalo u kojem smjeru gleda igrač. Svaki igrač ima svoju masu i collidere što znači da ne mogu prolaziti kroz zidove, druge igrače, projektile, kutije i slično. Jedino kada igrač primi kutiju, kutiji se onesposobljava fizika (rigidbody), te se pojavljuje ispred igrača, što znači da je moguće da kutija propadne kroz zid ako igrač tako želi. Ispuštanjem kutije, vraća joj se fizika (rigidbody) te se tada collidea sa svim ostalim stvarima. Ovisno o tome kojom brzinom se igrač kreće, tu silu dobiva kutija prilikom ispuštanja, što simulira bacanjem kutije u daljinu. Ako slučajno ili namjerno kutija završi izvan mape, kroz dvije sekunde će se vratiti na svoju početnu poziciju i omogućiti drugom igraču da ju dalje nosi. Kutija s kojom se dobije bod za tim, ne može nitko podići niti pomaknuti. S time se onemogućava igraču da ponovno s istom kutijom dobiva bodove.

Otvaranjem igre, pojavljuje se meni gdje je potrebno upisati svoje korisničko ime te odabrati tim. Odmah se može vidjeti cijela mapa, baza od plavih je kod plavih zastava, dok baza od crvenih je kod crvenih zastava. Meni se može vidjeti na sljedećoj slici.



Slika 9. Meni prilikom otvaranja igre [autorski rad]

Korisničko ime je obavezno i ne može se pristupiti serveru bez njegovog unosa. Odabirom tima i klikom na gumb Connect, igrač se spaja na server i odmah pojavljuje unutar mape.



Slika 10. Igra *Courier* [autorski rad]

Na prethodnoj slici se može vidjeti da se gore u sredini nalaze rezultati, gdje plavi broj označava rezultat plavoga tima, dok crveni označava rezultat crvenog tima. Ispod toga se nalazi broj koji pokazuje koliko je sekundi ostalo do kraja runde. Svaka runda traje 185 sekundi

ili kraće, ako neki od timova ubaci pet kutija na pravilno mjesto. Nakon toga igra završava i svi se igrači vraćaju na početnu poziciju.

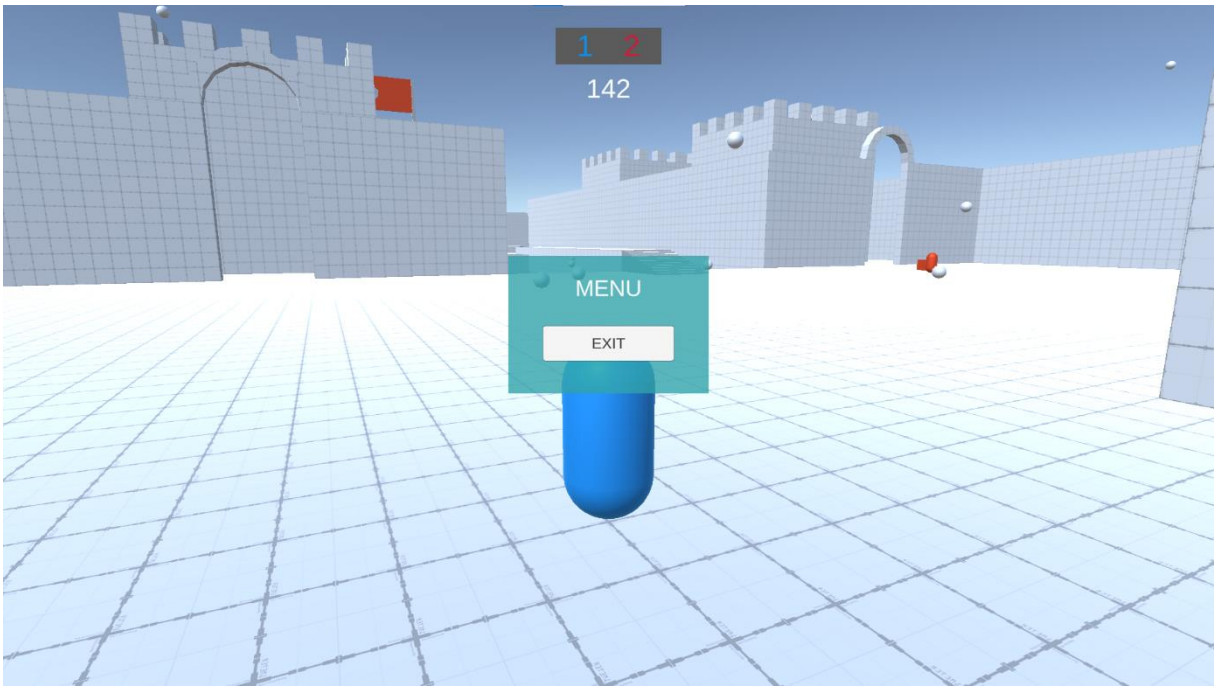
Ako igrača pogodi projektil ili ako padne u vodu, tada se vraća na početnu poziciju. Prilikom toga, igrač se ne može odmah kretati, nego mora čekati dvije sekunde da mu se pojavi tijelo. Nakon toga se aktiviraju kontrole i ponovno se moguće kretati s igračem. Također prilikom završetka runde, kada se svi igrači vraćaju na početnu poziciju, moraju pričekati dvije sekunde da im se pojavi tijelo i tek nakon toga se mogu početi kretati.

Cijelo vrijeme tijekom igranja, zapisuju se podaci igre u tekst datoteku na serveru, što omogućuje detaljniju povijest igre. Na primjer, pamti se datum i vrijeme pokretanja servera, tko i kada se spojio, kada je jedan od tima postavio kutiju i slično. Kada bi se ovaj projekt radio pomoću nadziranog učenja, tada bi se mogli koristiti logovi za detektiranje botova ili varanja.

6.3.1. Kontrole

Tipkovnica i miš su potrebni za igranje *Couriera*. W, A, S i D tipke se koriste za pokretanje igrača gore-dolje lijevo-desno. Lijevo-desno je ustvari strafeanje, dok je moguće okrenuti igrača mišem. Kretanje igrača je napravljeno s jednostavnom fizikom unutar Unitya. Na svakog igrača je postavljena C# skripta *Player.cs* koja dohvaća input od igrača (WASD tipke) i ovisno o tome koja je tipka stisnuta, igraču se računa smjer vektora te se njemu dodaje brzina tako da pomakne igrača. Prilikom toga, svaki put se provjerava je li igrač na podu, odnosno u doticaju s podom (mapom). Ako jest, tada može skočiti s razmakom na tipkovnici. Osim toga, igrač se može ubrzati (sprint), tako da drži tipku SHIFT prilikom korištenja WASD. Implementirano je da svaki igrač ima varijablu health koja iznosi 100, ali svaki put kada dotakne vodu ili kada ga pogodi projektil, oduzme mu se cijeli health. Trenutno sve rezultira vraćanjem na početnu poziciju, ali će lakše biti u budućnosti mijenjati jačinu ozljede ako se, na primjer, doda novi neprijatelj.

Ako igrač želi podići kutiju, potreban mu je lijevi klik miša. Podizanje kutija funkcionira na način da se prilikom svakog klika na mišu provjeri je li igrač dovoljno blizu kutije. Ako jest, tada će se kutija pojaviti ispred njega, a inače se neće dogoditi ništa.



Slika 11. Meni unutar igre [autorski rad]

Unutar igre postoji meni kojem se može pristupiti pomoću tipke ESC. Klikom na nju, otvara se mali meni koji omogućuje igraču da izađe iz igre, odnosno da se odspoji sa servera.

6.3.2. Multiplayer

Igra *Courier* se sastoji od servera i klijentskog dijela aplikacije. Unutar serverskog dijela, nalazi se sva logika igre kao što je kontroliranje igrača, botova, topništva, vode, rezultata i slično. Klijentski dio služi za prikaz podataka koje server šalje i za parsiranje koraka koje igrač da računalu. Server i klijentski dio cijelo vrijeme komuniciraju i razmjenjuju podatke o pozicijama svih igrača, njegovim karakteristikama kao što su health, tim, korisničko ime i kutija koju nosi te podaci o svim kutijama, projektilima i rezultatima. Cijeli multiplayer je implementiran pomoću tutorijala od Toma Weilanda koji su dostupni u literaturi [37].

Otvaranjem nove adrese ili VPN-a, postoji mogućnost igranja i preko javne IP adrese gdje bi se omogućilo da server i klijent ne moraju biti na istoj lokalnoj mreži.

6.3.3. Botovi

Da bi RL mogao razlikovati botove od igrača, potrebno je napraviti botove, odnosno ponavljajuće radnje igrača. Za implementaciju botova, koriste se stanja strojeva (eng. *state machines*), gdje ukratko bot ima određen broj stanja i završetkom jednog stanja, prelazi na drugo stanje. Svi prijelazi između stanja botova su prikazana u sljedećoj tablici.

Tablica 4. Prijelaz između stanja botova

No.	Trenutno stanje	Sljedeće stanje	Napravi prijelaz između stanja ako je zadovoljen uvjet
1.	Traži	Idi prema kutiji	Pronašao sam kutiju
2.	Idi prema kutiji	Traži	Ako si zaglavio više od pola sekunde
3.	Idi prema kutiji	Pokupi kutiju	Ako si došao dovoljno blizu kutije
4.	Pokupi kutiju	Traži	Ako nisi pokupio kutiju (netko drugi je uzeo kutiju)
5.	Pokupi kutiju	Idi na odredište	Ako si pokupio kutiju
6.	Idi na odredište	Ispusti kutiju	Ako si došao do odredišta
7.	Ispusti kutiju	Traži	Ako ne nosiš kutiju
8.	Iz bilo kojeg stanja	Mrtav	Ako je health nula ili manji od nule
9.	Mrtav	Traži	Ako je health veći od nule

[autorski rad]

Svaki bot ima šest stanja, a to su *Traži*, *Idi prema kutiji*, *Pokupi kutiju*, *Idi na odredište*, *Ispusti kutiju* i *Mrtav*. Kada se bot stvori na serveru, njegovo početno stanje jest *Traži*. Unutar tog stanja, bot traži sve kutije i odlučuje koja mu je najbliža. Nakon što je shvatio koja mu je najbliža, prelazi u sljedeće stanje (broj 1), a to je *Idi prema kutiji*, gdje se pomoću NavMeshAgent počinje kretati. NavMeshAgent funkcionira tako da se cijela karta *bakea* (ispeče) i prema tome što je ispečeno, NavMeshAgent može saznati gdje su zidovi, kako najbrže doći do nekog objekta i slično [38]. Prelazak između stanja broj 2 kaže da ako bot stoji više od pola sekunde na mjestu, odnosno ne radi ništa, da ponovno krene tražiti nove kutije. Ako dođe dovoljno blizu kutije, tada mu se stanje mijenja u *Pokupi kutiju*. Ponekad se dogodi da bot taman krene podići kutiju, ali prije toga netko drugi uzme njegovu kutiju, što onda rezultira blokiranju bota. Radi toga uvodi se prijelaz broj 4 gdje ako bot nije uspio pokupiti kutiju, da krene tražiti novu. Ako bot uspije pokupiti kutiju, tada prelazi u *Idi na odredište* gdje mora odnijeti kutiju do svojeg cilja tako da osvoji bod za svoj tim. Kada dođe do odredišta, prelazi u stanje ispuštanja kutije i nakon toga ponovno krene tražiti novu kutiju.

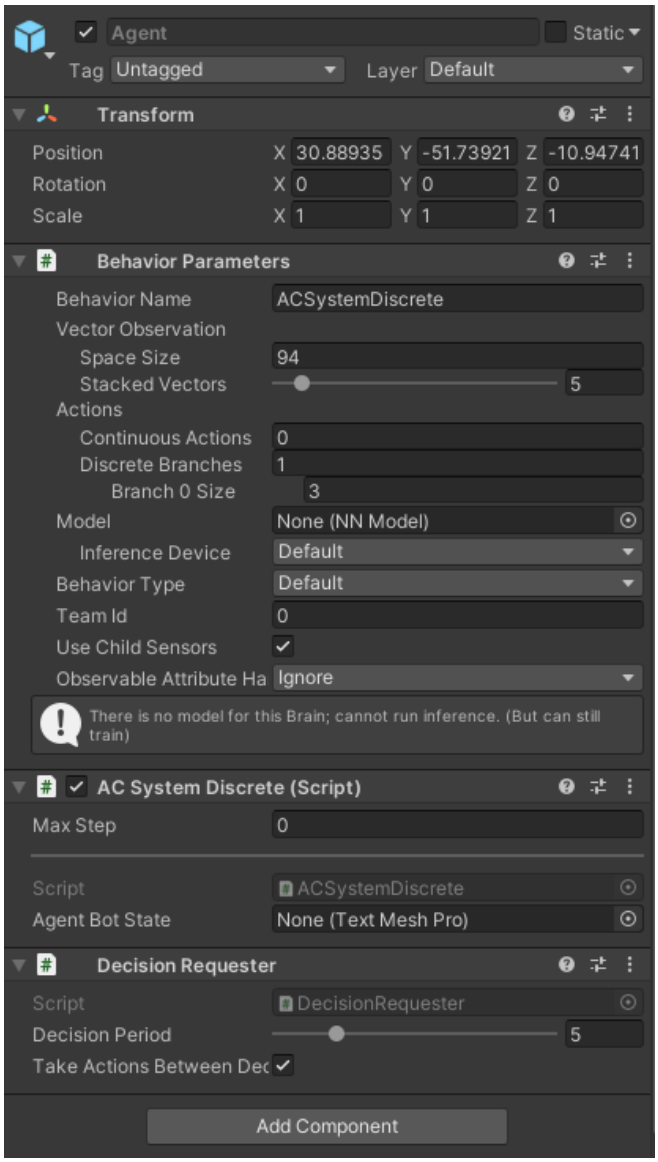
Osim promjene ovih nekoliko stanja, moguće je iz bilo kojeg stanja postati mrtav. Ako bot padne u vodu ili ako ga pogodi projektil, imat će nula ili manje healtha, što znači da je mrtav. Radi toga, prelazi u stanje *Mrtav* te se nakon nekoliko sekundi (dok mu se pojavi tijelo), ponovno vraća u stanje *Traži* i sve kreće ispočetka.

Trenutno ne postoji stanje koje govori što raditi ako ne postoji nijedna kutija, ali nije ni potrebno, jer je gotovo nemoguće da bot ne može doći do kutija. Čak ako i kutija ispadne iz mape ili padne u vodu, nakon nekoliko sekundi se ponovno pojavi na svome mjestu. Između ostalog, sveukupno postoji 12 kutija po jednom timu, a potrebno je samo njih pet ubaciti na odredište za pobjedu [39].

Osim ovog načina, botovi bi se mogli implementirati pomoću ML-Agentsa, gdje bi agent kao bot, samostalno shvatio što treba raditi. Ovaj način jedino zahtijeva veću količinu vremena, gdje je potrebno trenirati botove, pa moguće da će biti isprobano unutar novijih verzija igre.

6.4. Konfiguracija ML-Agentsa

Da bi ML-Agents funkcionirao unutar Unitya, potrebno je kreirati objekte koji će reprezentirati agente. Unutar projekta, kreira se onoliko agenata koliko je igrača, odnosno na svakog igrača postoji agent kojem će biti cilj da shvati je li igrač bot ili stvarni igrač. Na sljedećoj slici se mogu vidjeti karakteristike objekta Agenti unutar Unitya.



Slika 12. Postavke agenta (Unity)

Isprva je bilo zamišljeno da se kreira jedan god-mode agent koji će provjeravati sve igrače odjednom, ali zbog komplikacija vezanih uz skriptu *BehaviourParameters*, odlučeno je da se agent kreira prilikom kreiranja igrača. Jedan od razloga je bio parametar *VectorObservation → SpaceSize.* Unutar igre, uvijek se može spojiti neki novi igrač, što znači da će agent imati više opservacija nego što je imao prije njegovog spajanja. Jedno od rješenja je moglo biti dinamično mijenjanje elemenata, ali ML-Agents ne dopušta mijenjanje opservacija. Radi toga se agent kreira tek nakon što se igrač pojavi na serveru i tada će uvijek biti jednak broj opservacija. Na jednom igraču, agent opservira sljedeće stvari:

- Lokalna pozicija (Vector3)
- Lokalna rotacija po Y osi (float)
- Brzina igrača (Vector3)
- Pokupljena kutija (boolean)

- Pozicije kutija svog tima (12 kutija po timu, a od svake kutije lokalna pozicija – Vector3 i lokalna rotacija – Quaternion)
- Rezultat plavog tima (int)
- Rezultat crvenog tima (int)

To je sveukupno 94 opservacija koje jedan agent sakuplja prilikom proučavanja igrača. Svi agenti imaju zajednički *mozak*, što znači da uče jedan od drugog. Ono što nauči agent na jednom igraču, to će znati i agent na drugom igraču.

Botovi i igrači su implementirani na način da se mogu raspoznati unutar servera, ali agentu se šalju pokreti svih igrača, neovisno o tome je li bot ili ne, i prema tim podacima, agent mora samostalno naučiti razliku.

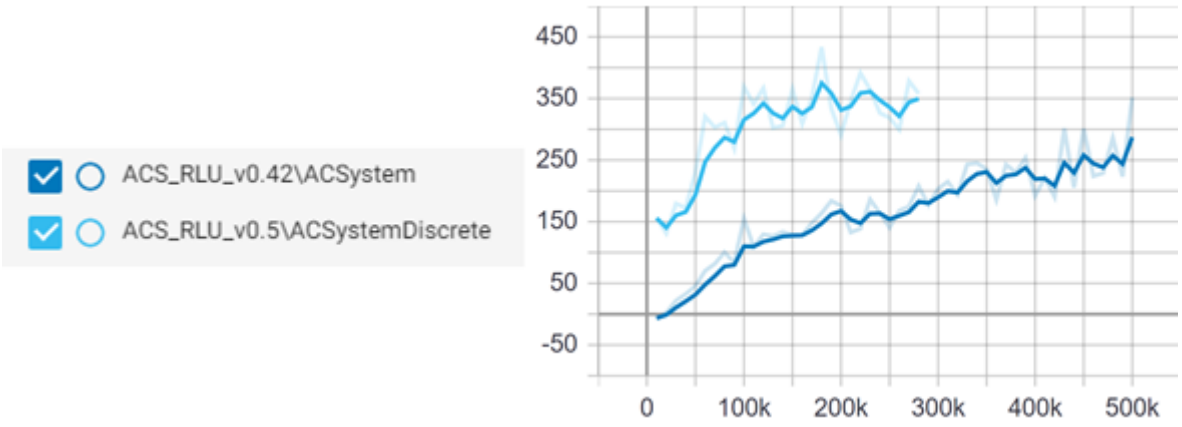
Agent donosi jednu diskretnu odluku u vrijednostima 0, 1 ili 2. Ako donese odluku 0, tada zaključuje da je igrač bot (istinit – eng. *true*). Odluka 1 govori da igrač nije niti bot niti stvaran igrač (neutralan – eng. *neutral*), a inače, ako donese odluku 2, tada zaključuje da je igrač stvarni igrač (lažan – eng. *false*). Iz razloga što se ne zna koji su stvarni igrači a koji botovi, može se agentu dodijeliti nagrada o donesenom zaključku. Zaključi li agent da je igrač stvarni igrač, dobit će pozitivnu nagradu (+1), a inače, ako zaključi da je stvarni igrač bot, tada će dobiti negativnu nagradu (-1). Ako agent odluči ostati pri neutralnom mišljenju, tada neće dobiti nagradu. Kod donošenja odluke kod bota, ako kaže da je bot bot, tada će dobiti pozitivnu nagradu (+1), a ako kaže da je bot stvarni igrač, tada će dobiti negativnu nagradu (-1). Isto kao i kod stvarnog igrača, ako agent ostane neutralan, neće dobiti niti pozitivnu niti negativnu nagradu.

Prilikom kreiranja agenta, važno je prije svega testirati radi li ispravno igra i može li agent normalno obavljati svoje dužnosti. Zbog toga postoji mogućnost implementiranja heuristike, gdje ustvari korisnik upravlja agentom tako da isproba njegove funkcionalnosti, kao što su na primjer, kretanje agenta po podlozi. Tek nakon što je agent potpuno ispravan, može se početi s njegovim treniranjem. Prilikom isprobavanja radi li sve ispravno, složeno je da F1 služi za pozitivno označavanje svih igrača, F2 za neutralno i F3 za negativno. Nakon isprobavanja, i uvjerenja da sve radi ispravno, krenulo se dalje.

Nakon svakog treniranja agenta, generira se ONNX file unutar kojeg se nalazi model naučenog agenta. Taj model se može dodati na agenta i tada će agent sve ono što je naučio pokazati u praksi. Ostale postavke *BehaviourParameters* su ostavljene po defaultu.

Postavlja se pitanje zašto koristiti diskretne akcije, a ne kontinuirane. Naime, kontinuirane akcije uvijek vraćaju decimalne vrijednosti između -1 i 1. Prilikom testiranja projekta, isprobane su i kontinuirane akcije, gdje ako agent donese odluku manju od nule, znači da igrač nije bot, a ako donese odluku veću od 0, tada je igrač bot. Inače ako zapiše 0,

tada je agent odredio da je igrač neutralan. Kada se koriste diskretne akcije, tada agent može samo odabrati jedan od tri broja, što znači da neutralna akcija unutar kontinuiranih akcija je manje moguća nego pozitivna ili negativna. Osim toga, testirano je učenje agenta s kontinuiranim i diskretnim akcijama čije su kumulativne nagrade prikazane na sljedećoj slici.



Slika 13. Usporedba kontinuiranih i diskretnih akcija (Tensorboard)

Tamnoplava krivulja prikazuje kontinuirane akcije, dok svijetloplava prikazuje diskretne. Može se odmah zaključiti da diskretne akcije puno brže skupe veću kumulativnu nagradu i da kontinuiranim akcijama treba više koraka da dostignu diskretne. Moguće je da bi kontinuirane akcije nakon nekog vremena došle na razinu diskretnih, ali vrijeme unutar ovog projekta je veoma bitno kao što je objašnjeno na početku petog poglavlja. Testiranje je rađeno na manjem broju koraka, samo radi usporedbe.

Nakon toga, potrebno je odrediti parametre YAML datoteke unutar koje se nalaze konfiguracija o treniranju. Više o tome je objašnjeno u sljedećem potpoglavlju. Nakon toga, sve konfiguracije su složene i može se krenuti na treniranje agenta.

6.4.1. Tensorboard

Tijekom i nakon treniranja je potrebno analizirati podatke da se može vidjeti radi li sve ispravno. Zbog toga se unutar projektnog dijela koristi program Tensorboard. On omogućuje kontinuirano promatranje statistike na modelu, odnosno agentu. Dok se pokrene učenje, ML-Agents sprema sve vrijednosti unutar datoteke *results* kojoj se može pristupiti preko Tensorboarda.

Tensorboard kreira lokalni server na određenom portu kojemu možemo pristupiti preko željenog web preglednika. Da bi se stvorio server, potrebno je napraviti sljedeće:

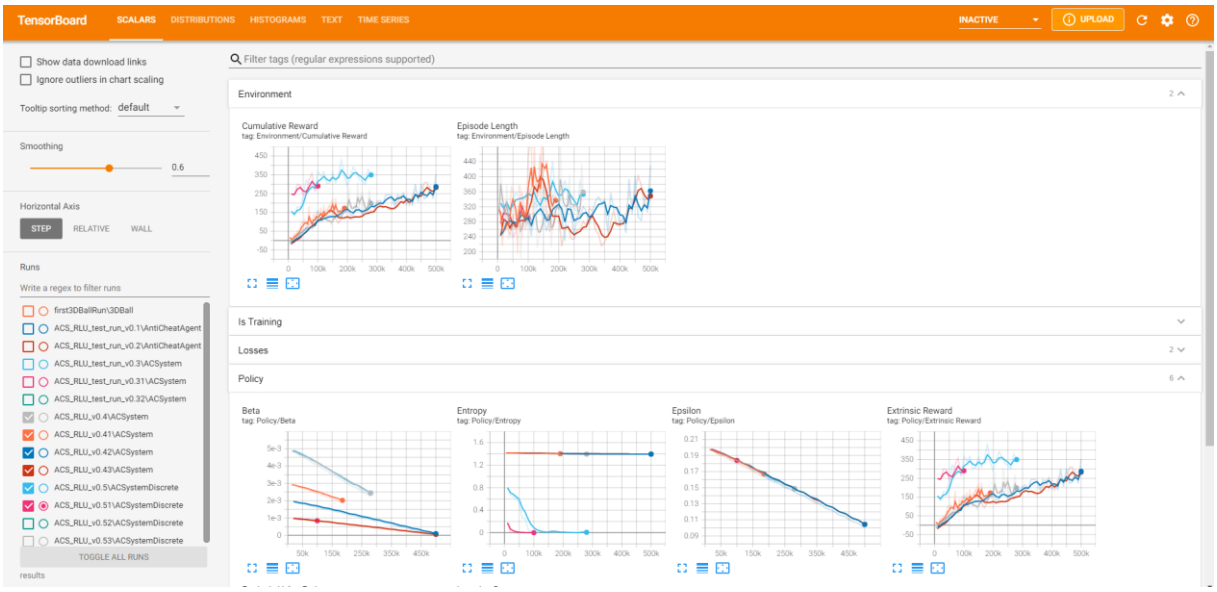
- Otvoriti naredbeni redak
- Pronaći datoteku unutar koje se nalazi ML-Agents paket

- Pokrenuti sljedeću komandu:

```
tensorboard --logdir results --port 6006
```

- U pregledniku otvoriti adresu http://localhost:6006/

Logdir unutar naredbe pokazuje na datoteku gdje se nalaze rezultati koje ML-Agents sprema, a port pokazuje na kojem će se portu kreirati server. Prema tome portu, otvara se adresa u web pregledniku i prikazuju svi modeli koji postoje unutar *results* datoteke [40]. Primjer izgleda aplikacije se može vidjeti na sljedećoj slici.



Slika 14. Prikaz Tensorboarda (Tensorboard)

Na vrhu je meni unutar kojeg je moguće otvoriti nekoliko prozora, odnosno vidjeti skalare, distribucije, histograme, tekstove i vremenske serije. Pod skalarima (glavni prozor) mogu se vidjeti podaci vezani za okolinu i treniranje. S lijeve strane se može označiti koji se sve modeli žele prikazati. Na slici su trenutno samo modeli od verzije 0.4, pa sve do 0.51. Unutar datoteke *results* su i testne verzije, ali one nisu prikazane na grafovima. Tako pod okolinom (eng. *Environment*) se mogu vidjeti kumulativne nagrade i duljina trajanja epizoda. Idealno je da kumulativne nagrade rastu postepeno bez naglih uspona i padova.

Potpuno narančasta krivulja (verzija v0.41) je verzija s kontinuiranim akcijama i vidi se da je dosta nagla prilikom svojih padova i uspona što je riješeno sa sljedećom verzijom (tamnoplava krivulja – v0.42) gdje krivulja polaganije raste.

Pod politikom (*policy*) se mogu vidjeti razni parametri. Zadnje dvije verzije (0.5 i 0.51) su verzije s diskretnim akcijama te se može vidjeti da puno brže imaju veću kumulativnu nagradu nego one s kontinuiranim akcijama. Ali, također, može se vidjeti da entropija pada veoma brzo s obzirom na kontinuirane akcije, pa agent možda neće dovoljno isprobavati svoje

akcije. S time se može zaključiti da verzija v0.51 treba još dodatne konfiguracije da bi funkcionirala pravilno.

6.4.2. Konfiguracija YAML datoteke

U nastavku se nalazi korištena YAML datoteka za treniranje modela.

```
behaviors:
  ACSystemDiscrete:
    trainer_type: ppo
    hyperparameters:
      batch_size: 32
      buffer_size: 10240
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    max_steps: 500000
    time_horizon: 64
    summary_freq: 10000
```

Tijekom testnog treniranja, korišteno je nekoliko konfiguracija. Glavni cilj je bio što više izravnati liniju kumulativne nagrade tako da nema pretjeranih padova niti uzdizanja i cijelo vrijeme se pratila entropija. Te dvije stvari su bile glavne prilikom optimiranja YAML datoteke. Testiranje je funkcioniralo na način da agent proučava samo botove, odnosno nisu igrale stvarne osobe. Agent je tada morao shvatiti da su oni botovi. Nakon nekoliko verzija YAML datoteke, napravljena je YAML datoteka koja je prikazana gore.

Kao što je već spomenuto u radu, koristi se PPO algoritam. Prilikom testiranja su se većinom mijenjali parametri *batch_size*, *buffer_size*, *learning_rate* i *beta*. Ostali parametri su ostali identični. Parametar *batch_size* za diskretne akcije prilikom korištenja PPO-a se pokazao da najbolje funkcionira, dok je na defaultnom minimumu, vrijednosti 32. Prema tome je određen *buffer_size* koji mora biti nekoliko puta veći od *batch_size*a. Korištenjem manjeg *buffer_sizea*, odnosno minimalni default 2048 za PPO, pokazalo se da agent jako loše uči tijekom koraka. Ispočetka obično krene s negativnom nagradom i tijekom vremena gotovo da ni ne prođe preko pozitivne nagrade. Zato je povećan *buffer_size* za 320 puta od *batch_size*a i to je omogućilo stabilnije treniranje. Prilikom mijenjanja tih parametara također se odmah mijenjao parametar *learning_rate* gdje su isprobani minimum i maksimum defaultnih verzija, ali odlučeno je za 0.0003. Testiranjem je zaključeno da parametar *beta* dobro radi dok je ostavljen na default vrijednosti, pa je tako i unutar ove YAML datoteke.

Postoji još mjesta za napredak, ali za ovaj jednostavan projekt je ova konfiguracija dovoljna. Naime, vjerojatno je još potrebno doraditi parametar *learning_rate* tako da se koraci unutar gradijentskog spuštanja malo stabilnije i bolje ažuriraju, ali pošto je u ovom radu također važna brzina učenja, *learning rate* je ostavljen na takvoj vrijednosti.

6.4.3. Pokretanje treniranja

Treniranje se obavlja pomoću Anaconda konzole unutar koje se pokreću python skripte vezane za ML-Agents. Da bi se pokrenulo treniranje, potrebno je koristiti naredbu *mlagents-learn* te odrediti YAML datoteku i naziv treniranja. Svako treniranje ima svoj jedinstveni ID s kojim se tada prikazuje na Tensorboardu. Kada se krene trenirati model, moguće ga je zaustaviti, ažurirati YAML datoteke ili neke druge parametre te nastaviti treniranje. ML-Agents ima dvadesetak neobaveznih parametara koji mogu ubrzati rad učenja. Na primjer, postoji argument *force* koji omogućuje da se izbrišu svi podaci ID-a treniranja i da se na njemu započne novo treniranje. Argument *resume* nastavlja već postojeće treniranje, što omogućuje da se trenirani model ne mora naučiti odjednom. Također, postoje argumenti za treniranje bez grafike, što omogućuje brže treniranje ako je kompleksan projekt u Unityu, kvaliteta nivoa, podaci o tensorflowu, torchu i ostalo.

Sljedećom naredbom se treniranje pokretalo:

```
mlagents-learn config/ppo/ACSystemDiscrete.yaml --run-id=ACS_RLU_v0.72
                                --time-scale=1
```

Nakon naredbe *mlagents-learn*, potrebno je navesti lokaciju YAML datoteke i dodati ID trenutne verzije modela. Ovakvom naredbom se prvi put pokretalo treniranje, a svaki sljedeći put se još dodavao argument *resume*. Na verziji 0.72 su agenti gdje igra jedna osoba protiv

jednog bota. Radi toga se koristi argument *time-scale* koji omogućuje normalan tok igre. Taj argument je ekvivalentan varijabli *Time.timeScale* u Unityu, što znači da određuje brzinu igre. Po defaultu je postavljen na 20 i igra bi unutar Unitya tekla prebrzo i osoba ju ne bi mogla igrati. Nakon što se pokrene treniranje, ML-Agents će tražiti da se započne scena u Unityu. Nakon započinjanja scene, odmah kreće treniranje te se pokazuju podaci vezani uz YAML datoteku. Prema YAML datoteci, svakih 10.000 koraka (*summary_freq*) će se pokazati sažetak treniranja, gdje će se unutar konzole prikazati koliko je vremena prošlo nakon zadnjeg sažetka, kolika je prosječna nagrada i koliko je prošlo koraka. Ako se želi zaustaviti treniranje, potrebno je zaustaviti igru u Unityu ili poslati signal u konzolnoj aplikaciji (CTRL+C) da se ugasi trenutni proces. Nakon toga se zaustavlja treniranje te se generira model vrste datoteke ONNX, koji se može ubaciti u skriptu (*Behavior Parameters*) agenta. Kada se ubaci, agent radi ono što je naučio prilikom treniranja u stvarnom vremenu.

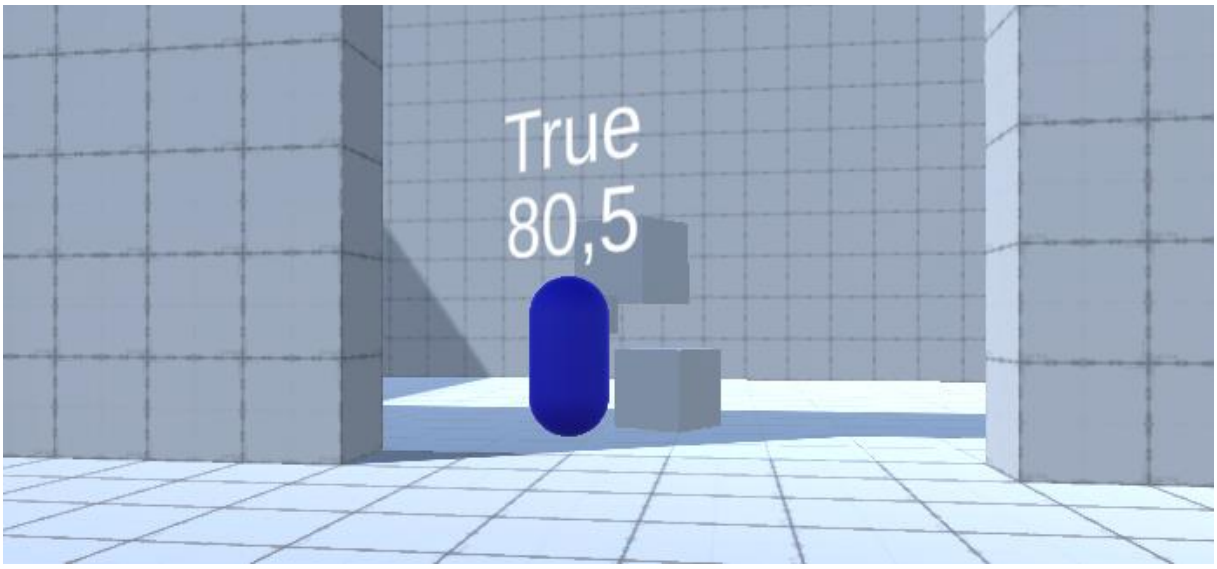
Testno treniranje agenata je isprva samo učilo na botovima. Na testiranju je igralo pet protiv pet botova i agent je morao shvatiti da su to botovi. Tek nakon dorade YAML datoteke i zadovoljavanja određenih parametara, napravljeno je treniranje gdje jedna osoba igra protiv jednog bota i onda kasnije dva protiv dva.

7. Rezultati

Prije nego što počne treniranje, potrebno je odrediti koliko agent donosi ispravnih odluka tako da se može zaključiti isplati li se koristiti RL unutar ovoga problema. Način na koji se izračunava postotak je taj da se uzme zadnjih 3.000 stanja agenta (pozitivan, negativan i neutralan) koja je dodijelio igraču te da se iz njih izračuna koliki je postotak pozitivnih stanja. Znači, ako je agent 2.700 puta označio igrača s pozitivnim stanjem od 3.000 puta, sljedećim izračunom se dobije koliki je postotak varanja tog igrača.

$$\frac{2.700}{3.000} * 100 = 90\%$$

Prema tome, vjerojatnost da igrač vara je 90% i gotovo sigurno se može zaključiti da je igrač bot. Također, ako je agent 200 puta označio igrača da vara od 3.000 puta, tada je vjerojatnost da igrač vara 6.67%. Postoci i stanje koje agent dodjeljuje igraču se prikazuju samo na serveru, što znači da klijent (igrač) to ne može vidjeti. Kako to izgleda na serveru, može se vidjeti na sljedećoj slici.



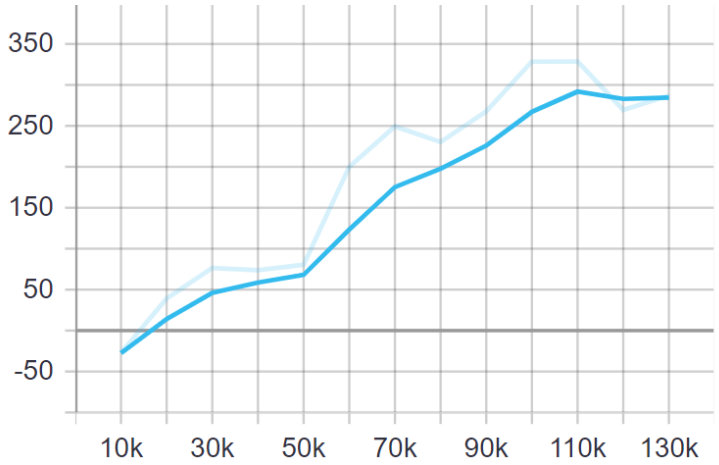
Slika 15. Postotak varanja

Stanje koje piše iznad igrača označava stanje koje je agent dodijelio igraču. U ovom slučaju *True*, što znači da je agent označio igrača kao bota, a ispod stanja piše ukupan postotak koji je izračunat od zadnjih 3.000 stanja. Taj postotak će cijelo vrijeme varirati i zbog toga će se uvesti računanje srednje vrijednosti postotka varanja kroz pet igara. Stanje koje agent da igraču će se uzimati svake sekunde i zapisivati u tekst datoteku na serveru. Iz te datoteke će se tada uzeti sve te vrijednosti i izračunati njihova prosječna vrijednost. Ta vrijednost će se uzimati kao reprezentativna i uspoređivati s ostalim vrijednostima.

Treniranje je započelo s 1v1 gdje postoji jedan bot u plavom timu i jedna osoba u crvenom timu i ono se može vidjeti u sljedećem poglavlju.

7.1. Treniranje 1v1

Verzija na kojoj je napravljen model je 0.72 i radi na YAML datoteci koja je bila prikazana unutar potpoglavlja *Pokretanje treniranja*. Na sljedećoj slici se može vidjeti kumulativna nagrada kroz 130.000 koraka.



Slika 16. Kumulativna nagrada treniranja 1v1 (Tensorboard)

Uvijek je bolje trenirati što dulje, ali prema ovim podacima se može vidjeti da agent, gotovo svakim korakom, povećava nagradu i da ima određeno znanje za razlikovanje bota od stvarnog igrača. U početku je treniranje krenulo s negativnom nagradom, gdje je agent nasumično isprobavao što bi trebao raditi. Svakih 10.000 koraka, nagrada se povećava stabilno. Treniranje je trajalo 1 sat 51 minutu i 17 sekundi.

Pokretanjem naučenog modela je napravljeno testiranje gdje se u pet igara uzima postotak varanja, pri čemu je osoba u početku u crvenom timu kao što je bilo u treniranju, a nakon tih pet igara, zamijenit će se pozicije osobe i bota, pa će tada osoba biti u plavom timu, a bot u crvenom. Kroz pet igara, gdje je igrač u crvenom timu, prosječna vrijednost njegovog varanja je bila 12,18%, dok je od bota bila 84,15%. Prema prosječnim vrijednostima se može zaključiti da agent može razlikovati bota od igrača u tom scenariju. Sljedeći scenarij je da je osoba unutar plavog tima, a bot unutar crvenog tima. Prosječna vrijednost varanja igrača u plavom timu je bila 71,92%, a bota u crvenom 6,34%.

Tablica 5. Treniranje 1v1

	Plavi tim		Crveni tim	
	Igrač	Postotak varanja	Igrač	Postotak varanja
Test 1	Bot	84,15%	Osoba	12,18%
Test 2	Osoba	71,92%	Bot	6,34%

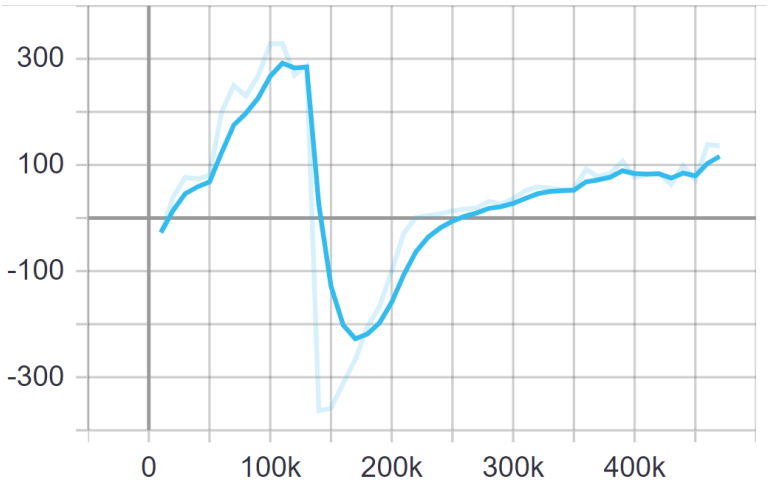
[autorski rad]

Iz toga se može zaključiti da agent ne radi ispravno, jer označuje igrača kao bota, a bota kao igrača. Razlog tome je to što agent proučava pozicije igrača, njihovu rotaciju, kutije i slično. Bot većinom koristi istu putanju od kutija do suprotne strane mape gdje ostavlja kutiju, dok se igrač iz crvenog tima većinom koristio središnjim mostom. Ono što je agent zaključio jest ako se krene kao plavi tim i kreće se lijevom stranom (gdje bot gotovo uvijek putuje), da je osoba bot. A inače, ako je osoba unutar crvenog tima i kreće se lijevom ili središnjom stranom, tada je stvarni igrač. Točno tako je zaključio agent prilikom izmjene strana timova. Treniranje s dva igrača, gdje je jedan stvaran a drugi nije, je premalo. Također, postoji premalo treniranja, jer je treniranje trajalo samo dva sata.

Radi toga, potrebno je promijeniti način treniranja, gdje će se u nastavku igrati 2v2, odnosno dva bota i dvije osobe.

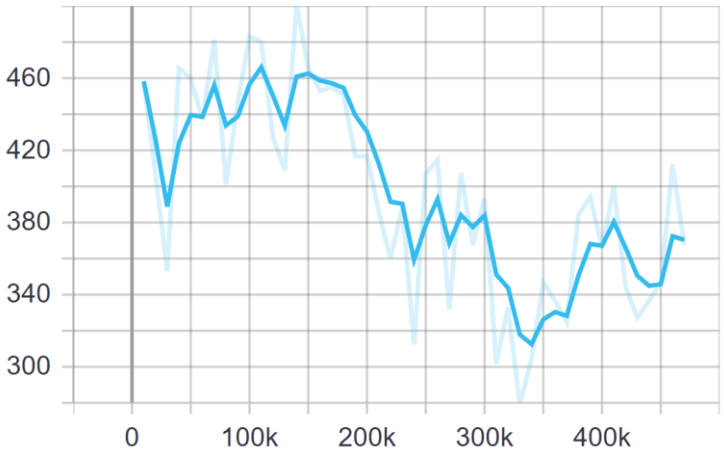
7.2. Treniranje 2v2

Unutar ovog treniranja, postoji jedna osoba u plavom timu zajedno s jednim botom i jedna osoba u crvenom timu s jednim botom. Prilikom treniranja 2v2 je korištena ista verzija modela (0.72), samo je nastavljeno treniranje od 1v1. Treniranje se može vidjeti na sljedećoj slici gdje se može primijetiti nastavak na prošlo treniranje.



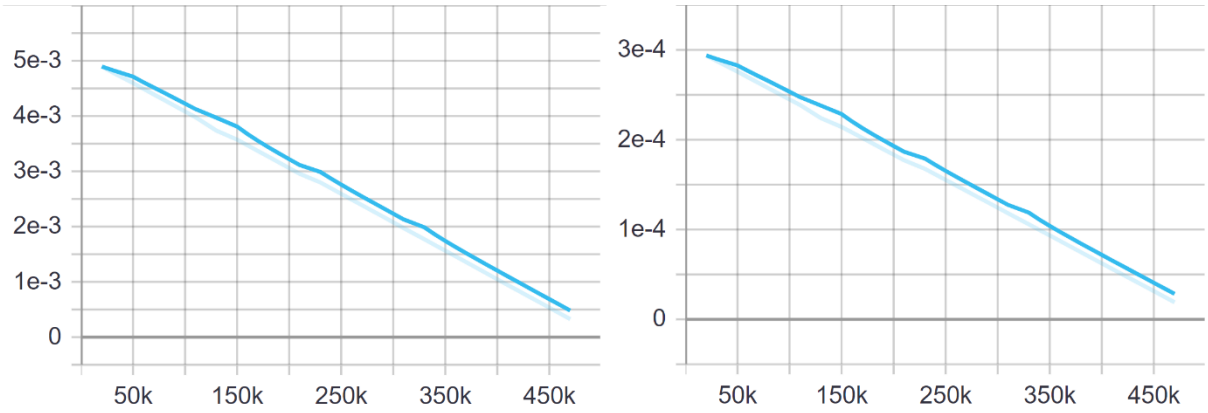
Slika 17. Kumulativna nagrada treniranja 2v2 (Tensorboard)

Prilikom nastavka treniranja na prošlo treniranje, kumulativna nagrada je pala na vrijednost od -227,5. Znači da agent nije točno znao što treba raditi jer se našao u novoj okolini gdje postoji više igrača. Nakon toga je kumulativno počelo rasti kontinuirano bez nekih naglih padova ili uzdizanja. Treniranje je trajalo 3 sata 26 minuta i 56 sekundi i napravljeno je 340.000 koraka. Trajanje epizoda, odnosno koliko je trebalo vremena da se sažetak prikaže, se može vidjeti na sljedećoj slici.



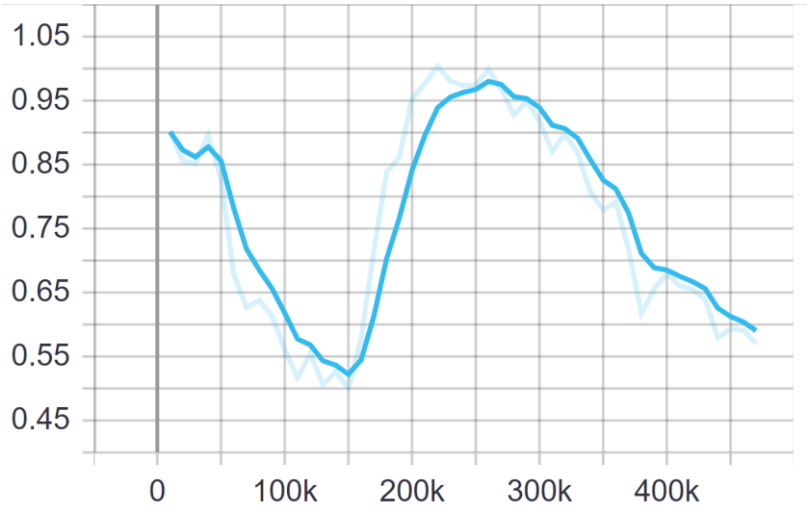
Slika 18. Vrijeme trajanja epizode (Tensorboard)

Može se primijetiti da nakon 130.000 koraka počinju epizode trajati sve kraće. Da postoji deset igrača na serveru, trajanje epizode bi trajalo još i kraće i vjerojatno bi se agent puno brže učio.



Slika 19. Parametar *beta* (lijevo) i *learning rate* (desno) (Tensorboard)

Parametri *beta* i *learning rate* su kontinuirano padali, što znači da je agent kontinuirano učio stvari unutar prvog i drugog treniranja. Koristio je svoje znanje iz prvog treniranja u drugom treniranju.



Slika 20. Entropija (Tensorboard)

Što se tiče nasumičnih akcija koje agent donosi, mogu se vidjeti na slici entropije. Unutar treniranja 1v1 agent je već znao odrediti bota od osobe u određenim timovima, a kasnije, dok je krenulo 2v2 treniranje, ponovno je morao donositi nasumičnije akcije tako da može pronaći uzorak u ponašanju osoba i bota. Entropija je od 250.000 koraka počela padati i može se zaključiti da agent više nije morao nasumično donositi akcije jer je već znao što treba raditi.

Nakon završetka 2v2 treniranja, napravljeni su testovi kao i unutar 1v1. Testiranje se odvijalo tako da je prvo jedna osoba u crvenom timu, dok je druga u plavom, gdje oba tima imaju po jednog bota te se na kraju igrači zamijene za timove. Rezultati su prikazani u sljedećoj tablici.

Tablica 6. Treniranje 2v2

	Plavi tim		Crveni tim	
	Igrač	Postotak varanja	Igrač	Postotak varanja
Test 1	Osoba 1	28,98%	Osoba 2	14,28%
	Bot 1	38,43%	Bot 2	67,78%
Test 2	Osoba 2	17,90%	Osoba 1	31,72%
	Bot 2	40,54%	Bot 1	62,61%

[autorski rad]

Iz tablice se može zaključiti da su oba testiranja gotovo ispravna, jedino što se tiče bota unutar plavog tima. Kako je unutar treniranja 1v1 bot igrao na plavoj strani, tako je teže učio, jer *learning rate* je padala kontinuirano. Agent je počeo označavati sve bolje tog bota i vjerojatno bi ga još bolje nastavio pronalaziti da je treniranje trajalo još neko vrijeme. Naime, unutar ovog treniranja, agent razlikuje osobe od botova jer je prosječna razlika između osobe i bota 29,12%. Da bi se razlikovali botovi od igrača unutar ove igre, može se samo reći da je igrač sigurno bot ako je iznad 60%, a da treba napraviti dodatne provjere ako je agent označio igrača s više od 35%.

Treniranje 2v2 je funkcioniralo puno bolje zato što postoji više igrača unutar igre i zato što je napravljeno dulje treniranje. Idealno bi bilo treniranje 5v5, gdje bi agent tada mogao proučiti gotovo sve putanje unutar okoline i shvatiti tko je bot, a tko stvaran igrač.

7.3. Prednosti

Unutar kratkog vremena treniranja je napravljeno da agent razlikuje bota od stvarnog igrača. Bilo je potrebno oko pet sati treniranja da bi se postigli ovi rezultati. Naravno, potrebno je puno više vremena da bi agent s višom vjerojatnošću mogao razlikovati bota od osobe. ML-Agents omogućuje jednostavno implementiranje RL agenta s mogućnostima praćenja podataka statistike, bez da se neki od algoritama moraju samostalno pisati i slično. Potrebno je samo postaviti unutar Unitya određene skripte, dodati opservacije, akcije i nagrade i može se krenuti s treniranjem agenta.

7.4. Nedostaci

Jedan od nedostataka jest taj da je potrebno puno vremena da bi agent mogao s visokom vjerojatnošću odrediti bota od osobe. Osim toga, kao što se moglo vidjeti unutar treniranja 1v1, potrebno je imati što više igrača i napraviti što više mogućnosti da bi agent mogao znati što treba činiti u određenim situacijama. Nadalje, ML-Agents postoji samo unutar Unitya. Naravno, moguće je implementirati iste algoritme unutar nekog drugog pogona igre, ali to bi bilo dugotrajno i trebalo bi imati dosta znanja o algoritmima i načinima kako ih implementirati te optimirati.

8. Zaključak

Strojno učenje je grana umjetne inteligencije koja omogućuje računalu da uči bez izričitog programiranja. Ono se dijeli na nadzorno učenje, nenadzorno učenje i RL. Nadzorno učenje omogućuje treniranje modela na označenim podacima gdje donosi zaključke o tome kako razlikovati podatke. Nenadzorno učenje koristi neoznačene podatke i pokušava ih organizirati u klastere. RL iliti podržano učenje, je agent koji dobiva stanja iz okoline, a poduzimanjem nekih akcija na okolini, ponovno dobiva nova stanja i nagrade pomoću kojih uči nešto napraviti. Cilj agenta je maksimiziranje nagrade, gdje se pozitivne nagrade ostvaruju ciljevima. OpenAI također koristi RL agenta i on je prvi AI program koji je uspio pobijediti svjetske prvake u Dota2. Osim RL-a, veoma su bitne neuronske mreže, koje simuliraju neurone u mozgu.

Sigurnost je jedna od važnijih stvari vezanih za igranje računalnih igara. Dobivanje prednosti nad drugim igračima je uvijek cilj multiplayer igre, ali kada to nije moguće, igrači se okreću nedopuštenim načinima dobivanja prednosti. Pronalaze načine kako da prevare sustav bez da budu kažnjeni. Jedan od tih načina je kreiranje umjetne inteligencije (botova) koji igraju umjesto igrača. Računalo može igrati danonoćno, dok se igrač nakon nekog vremena mora odmarati, a kada se odmara, ne može igrati igru u kojoj, na primjer, mora obaviti određene *questove*. Radi takvih primjera, unutar vlastite igre *Courier*, se implementirao sustav protiv varanja, koji pomoću RL-a, omogućuje pronalaženje botova. *Courier* je jednostavna multiplayer igra unutar koje postoje dva tima i gdje je potrebno prebaciti kutije s jedne strane na drugu izbjegavajući određene prepreke. U igri su implementirani botovi koji izvršavaju određena stanja. Da bi RL agent pronašao razliku između bota i stvarnog igrača, potrebno mu je dati određena stanja koja on proučava i prema tome odrediti je li igrač bot ili ne.

Početak treniranja agenta je funkcioniralo na način da je jedan stvarni igrač igrao protiv jednog bota. To je rezultiralo nepravilnim odlučivanjem agenta kada bi se zamijenili timovi. Radi toga, napravljeno je treniranje gdje postoje dva stvarna igrača, jedan u plavom, a drugi u crvenom timu. Svaki od timova je još imao po jednog bota. Nakon toga je agent trenirao nekoliko sati i uspio razlikovati bota od igrača s određenim postotkom neovisno o tome u kojem timu se nalaze igrači. Kada bi agent imao mogućnost treniranja nekoliko dana, čak i tjedana, mogao bi s većom vjerojatnošću odrediti bota od igrača. Multiplayer igre koje se igraju jako često (nekoliko igara po sat vremena svaki dan) bi imale veću šansu da agent ispravno nauči razlike između botova i osoba, pa čak i da pronađe osobe koje koriste umjetnu inteligenciju da bi dobile prednost nad drugima.

Popis literature

[1] M. Awad i R. Khanna, „Machine Learning. Efficient Learning Machines“, 2015. [Na internetu]. Dostupno: https://link.springer.com/chapter/10.1007/978-1-4302-5990-9_1. [pristupano 23.08.2021.]

[2] M. Lee, „Samuel's Checkers Player“, 2005. [Na internetu]. Dostupno: http://www.incompleteideas.net/book/ebook/node109.html. [pristupano 23.08. 2021.].

[3] J. Brownlee, „What is Machine Learning?“, 2013. [Na internetu]. Dostupno: https://machinelearningmastery.com/what-is-machine-learning/. [pristupano 24.08.2021.].

[4] R. Prabowo i M. Thelwall, „Sentiment analysis: A combined approach“, *Journal of Informetrics 3*, str. 143 -157, 2009. [Na internetu]. Dostupno: https://www.sciencedirect.com/science/article/abs/pii/S1751157709000108. [pristupano 25.08.2021.].

[5] C3.ai, „Supervised Learning“, bez dat. [Na internetu]. Dostupno: https://c3.ai/introduction-what-is-machine-learning/supervised-learning/. [pristupano 25.08.2021.].

[6] A. Bhandari, „Feature Scaling for Machine Learning: Understanding the Difference Between Normalization vs. Standardization“, 2020. [Na internetu]. Dostupno: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/. [pristupano 25.08.2021.].

[7] M. Conor, „Machine learning fundamentals (I): Cost functions and gradient descent“, 2017. [Na internetu]. Dostupno: https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220. [pristupano 25.08.2021.].

[8] K. Santos i S. Ojha, „Parallelizing Gradient Descent“, 2018. [Na internetu]. Dostupno: https://shashank-ojha.github.io/ParallelGradientDescent/Final%20Report.pdf. [pristupano 26.08.2021.].

[9] J. Brownlee, „Overfitting and Underfitting With Machine Learning Algorithms“, 2016. [Na internetu]. Dostupno: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/. [pristupano 26.08.2021.].

[10] P. Gupta, „Regularization in Machine Learning“, 2017. [Na internetu]. Dostupno: https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a. [pristupano 26.08.2021.].

[11] L. Heidmann, „Unsupervised Machine Learning: Use Cases & Examples“, 2020. [Na internetu]. Dostupno: https://blog.dataiku.com/unsupervised-machine-learning-use-cases-examples. [pristupano 26.08.2021.].

[12] JavaTPoint, „Unsupervised Machine Learning“, bez dat. [Na internetu]. Dostupno: https://www.javatpoint.com/unsupervised-machine-learning. [pristupano 30.08.2021.].

[13] C. Piech, „K Means“, 2013. [Na internetu]. Dostupno: https://stanford.edu/~cpiech/cs221/handouts/kmeans.html. [pristupano 26.08.2021.].

[14] A. Ng, „Motivation I: Data Compression“, bez dat. [Na internetu]. Dostupno: https://www.coursera.org/learn/machine-learning/lecture/0EJ6A/motivation-i-data-compression. [pristupano 26.08.2021.].

[15] A. Holehouse, „Dimensionality Reduction (PCA)“, bez dat. [Na internetu]. Dostupno: https://www.holehouse.org/mlclass/14_Dimensionality_Reduction.html. [pristupano 26.08.2021.].

[16] R. S. Sutton i A. G. Barto, Reinforcement Learning: An Introduction, London, England: A Bradford Book, 2016.

[17] OpenAI, „OpenAI Five“, 2018. [Na internetu]. Dostupno: https://openai.com/blog/openai-five/. [pristupano 10.09.2021.].

[18] C. Berner, G. Brockman, B. Chan i ostali, „Dota 2 with Large Scale Deep Reinforcement Learning“, 2021. [Na internetu]. Dostupno: https://arxiv.org/abs/1912.06680. [pristupano 10.09.2021.].

[19] D. Domijan, „Uvod u neuronske mreže“, *Metodički ogledi,* svez. 7, pp. 101-127, 2000. [Na internetu]. Dostupno: https://www.bib.irb.hr/225446. [pristupano 13.09.2021.].

[20] D. Ramos, „Real-Life and Business Applications of Neural Networks“, 2018. [Na internetu]. Dostupno: https://www.smartsheet.com/neural-network-applications. [pristupano 13.09.2021.].

[21] Ž. U. Andrijić, „Umjetne neuronske mreže“, *Osvježimo znanje,* svez. Dostupno: https://hrcak.srce.hr/file/322233, pp. 219-220, 2019. [pristupano 13.09.2021.].

[22] Altexsoft, „Image Recognition with Deep Neural Networks and its Use Cases“, 2019. [Na internetu]. Dostupno: https://www.altexsoft.com/blog/image-recognition-neural-networks-use-cases/. [pristupano 13.09.2021.].

[23] Unity, „Welcome to Unity“, bez dat. [Na internetu]. Dostupno: https://unity.com/our-company. [pristupano 31.08.2021.].

[24] I. Buckley, „7 Unity Game Development Languages to Learn: Which Is Best?“, 2019. [Na internetu]. Dostupno: https://www.makeuseof.com/tag/unity-game-development-languages/. [pristupano 31.08.2021.].

[25] ML-Agents, „ML-Agents Toolkit Overview“, 2021. [Na internetu]. Dostupno: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md. [pristupano 31.08.2021.].

[26] ML-Agents, „Training Configuration File“, 2021. [Na internetu]. Dostupno: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md. [pristupano 01.09.2021.].

[27] ML-Agents, „Training with Proximal Policy Optimization“, 2018. [Na internetu]. Dostupno: https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Training-PPO.md. [pristupano 16.09.2021.].

[28] J. Schulman, F. Wolski, P. Dhariwal, A. Radford i O. Klimov, „Proximal Policy Optimization Algorithms“, 2017. [Na internetu]. Dostupno: https://arxiv.org/abs/1707.06347. [pristupano 16.09.2021.].

[29] J. Yan i B. Randell, „A Systematic Classification of Cheating in Online Games“, 2005. [Na internetu]. Dostupno: https://dl.acm.org/doi/abs/10.1145/1103599.1103606. [pristupano 14.09.2021.].

[30] Steam, „Valve Anti-Cheat (VAC) System“, bez dat. [Na internetu]. Dostupno: https://help.steampowered.com/en/faqs/view/571A-97DA-70E9-FF74. [pristupano 14.09.2021.].

[31] ESEA, „ESEA Client“, bez dat. [Na internetu]. Dostupno: https://play.esea.net/client. [pristupano 14.09.2021.].

[32] H. Alayed, F. Frangoudes i C. Neuman, „Behavioral-Based Cheating Detection in Online First Person Shooters using Machine Learning Techniques“, 2013. [Na internetu]. Dostupno: https://ieeexplore.ieee.org/abstract/document/6633617 [pristupano 14.09.2021.].

[33] L. Chapel, D. Botvich i D. Malone, „Probabilistic Approaches to Cheating Detection in Online Games“, 2010. [Na internetu]. Dostupno: https://www.researchgate.net/publication/221157498_Probabilistic_Approaches_to_Cheating_Detection_in_Online_Games. [pristupano 14.09.2021.].

[34] H.-K. Pao i K.-T. Chen, „Game Bot Detection via Avatar Trajectory Analysis“, 2010. [Na internetu]. Dostupno: https://ieeexplore.ieee.org/document/5560779. [pristupano 14.09.2021.].

[35] Unity, „About ProGrids“, 2020. [Na internetu]. Dostupno: https://docs.unity3d.com/Packages/com.unity.progrids@3.0/manual/index.html. [pristupano 29.08.2021.].

[36] Unity, „ProBuilder“, bez dat. [Na internetu]. Dostupno: https://unity.com/features/probuilder. [pristupano 29.08.2021.].

[37] T. Weiland, „Connecting Unity Clients to a Dedicated Server | C# Networking Tutorial“, 2019. [Na internetu]. Dostupno: https://www.youtube.com/watch?v=uh8XaC0Y5MA&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5. [pristupano 29.08.2021.].

[38] Unity, „NavMesh Agent“, 2020. [Na internetu]. Dostupno: https://docs.unity3d.com/Manual/class-NavMeshAgent.html. [pristupano 29.08.2021.].

[39] J. Weimann, „Unity Bots with State Machines - Extensible State Machine / FSM“, 2020. [Na internetu]. Dostupno: https://game.courses/bots-ai-statemachines/.[pristupano 30.08.2021.].

[40] ML-Agents, „Using TensorBoard to Observe Training“, 2021. [Na internetu]. Dostupno: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md. [pristupano 12.09.2021.].

Popis slika

Popis tablica