

SADRŽAJ

1. Uvod	1
2. Tehnike potpornog učenja	2
2.1. Potporno učenje	2
2.2. Q-učenje	2
2.3. Duboko Q-učenje	3
2.4. Duplo duboko Q-učenje	5
2.5. Dugo kratkotrajna memorija	5
3. Popularne igre, njihove složenosti i poznati programski agenti	6
3.1. Složenost igara	6
3.2. Šah	6
3.2.1. Opis igre	6
3.2.2. Složenost igre	6
3.2.3. Poznati programski agenti	7
3.3. Go	7
3.3.1. Opis igre	7
3.3.2. Složenost igre	8
3.3.3. Poznati programski agenti	8
3.4. Balansiranje štapa	8
3.4.1. Opis igre	8
3.4.2. Složenost igre	9
3.4.3. Poznati programski agenti	9
3.5. Atari videoigre	9
3.5.1. Opis igre	9
3.5.2. Složenost igre	10
3.5.3. Poznati programski agenti	10
3.6. DOTA 2	10

3.6.1. Opis igre	10
3.6.2. Složenost igre	13
3.6.3. Poznati programski agenti	13
3.7. Pojednostavljena verzija DOTA 2 igre	13
3.7.1. Opis igre	13
3.7.2. Složenost igre	14
3.8. Sažetak složenosti navedenih igara	15
4. Izrada modela potpornog učenja	16
4.1. Odabir tehnike potpornog učenja	16
4.2. Okruženje i tehnologije	17
4.3. Implementacija modela programskog agenta	17
4.4. Analiza rezultata	20
5. Projekt Breezy	22
5.1. Integracija	22
5.2. Instalacija	22
5.2.1. Zahtjevi	22
5.2.2. Igra DOTA 2	23
5.2.3. Breezy Addon	25
5.2.4. Breezy Server	25
6. Prilagodba modela na igru DOTA 2	27
6.1. Komunikacija s Breezy sučeljem	27
6.2. Predobrada podataka	28
6.3. Modifikacija agenta	31
6.4. Zapisivanje rezultata	33
7. Rezultati	35
8. Zaključak	36
9. Addendum	37
Literatura	45

1. Uvod

Videoigre su vrsta igara koje se igraju na računalu ili igačoj konzoli. Igrač bira akcije slanjem signala programu videoigre; najčešće putem tipkovnice, miša i/ili igačeg kontrolera. Odabir akcije igač temelji na trenutnom stanju igre koje može biti preneseno značajkama u igri (poput trenutne brzine kretanja), prikazom kadra na monitoru ili stanjem memorije u računalu. Videoigre stvaraju zanimljivu domenu za istraživanja u području računalne znanosti. One pružaju sintetička okruženja u kojima se pojavljuju složena ponašanja, ali u kojima se percepcija i djelovanje mogu u potpunosti kontrolirati, pa se tako radi o kvalitetnim okolinama za eksperimentiranje s inteligentnim programskim agentima.

Programski agent (engl. *software robot*) je program koji se pokreće paralelno s igrom. Svrha takvog programa jest igrati videoigru umjesto čovjeka. Razlog razvijanja programskih agenata može biti npr. omogućavanje igranja igre za dva ili više igrača jednom korisniku ili pokušaj postizanja nadljudskih rezultata u igri. Postoje mnoge tehnike izrade programskih agenata, a najrasprostranjenija je upotreba stabla odluke (engl. *decision trees*). Stabla odluke su vrlo predvidiva što je odlično ako je prostor stanja igre mali pa je moguće konstruirati tzv. *savršen model*; agenta koji će u svakom stanju donijeti najbolju odluku. Prostor stanja popularnih videoigara je toliko velik da je nemoguće napisati model stabla odluke koji bi mogao biti pohranjen na današnje memorijske sustave, a kad bi to i bilo moguće vrijeme potrebno za izračun najbolje akcije u svakom trenutku bi značajno nadišlo *vijek trajanja* igre ako ne i čovječanstva. Za takve slučajeve trenutno najuspješniji pristup jest koristiti potporno učenje (engl. *reinforcement learning*). Potpornim učenjem programski agenti mogu naučiti prepoznati sličnosti situacija u kojima se nalaze i odabrati najbitnije značajke trenutnog stanja na temelju kojih će odabirati akcije. U ovom radu dat će se osvrt na najpopularnije tehnike potpornog učenja, složenosti poznatih društvenih igara i videoigara te navesti primjeri poznatih programskih agenata koji su dosad razvijeni. Nadalje, bit će dokumentirana izrada programskog agenta na pojednostavljenom modelu popularne videoigre *DOTA 2* uz upotrebu sučelja *Project Breezy*.

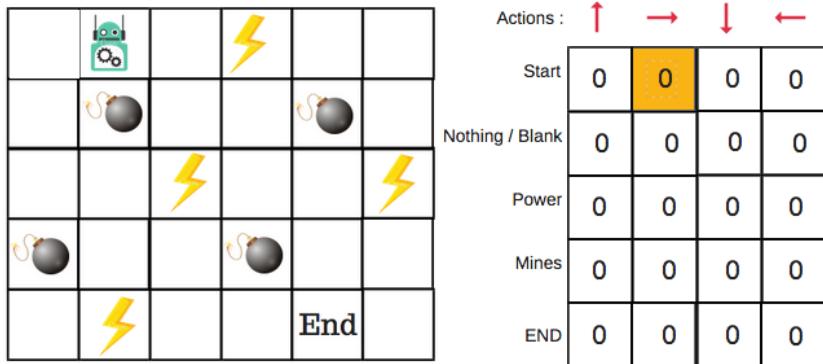
2. Tehnike potpornog učenja

2.1. Potporno učenje

Potporno učenje je dio umjetne inteligencije vezan uz izradu programskih agenata. Cilj tehnika potpornog učenja je konstruirati funkciju za odabir akcije na temelju stanja u kojem se agent trenutno nalazi i stanja kojima je u njega došao. Skup značajki koje opisuju jedno stanje nazivamo *prostor stanja* (engl. *observation space*), a skup svih akcija koje je u bilo kojem trenutku igre moguće odigrati *prostor akcija* (engl. *action space*). Izraz *prostor stanja* se u hrvatskom jeziku također koristi za skup svih stanja u koje je moguće doći, no zbog značajne razlike u značenju na temelju konteksta bi trebalo biti jasno o čemu se radi. Model potpornog učenja se često naziva *učenje temeljeno na nagradama* jer ga u procesu učenja, dok model istražuje prostor stanja i radi nasumične akcije, sustav kumulativno nagrađuje u skladu s ishodima igre. U zadnjih par godina je mnogo napretka postignuto u razvitku metoda potpornog učenja. U nastavku su dane neke od najpopularnijih tehnika.

2.2. Q-učenje

Q-učenje je tehnika potpornog učenja u kojoj svakom stanju u kojem se model može naći pridružimo svakoj akciji skalar koji nazivamo *Q-vrijednost*. Q-vrijednost je veličina koja određuje *dobrotu* (engl. *fitness*) akcije u nekom stanju. Tada se problem izrade programskog agenta pojednostavljuje jer više nije cilj konstruirati funkciju koja će odabrati optimalnu akciju nego je cilj naći optimalne Q-vrijednosti u svakom stanju za svaku akciju. Q-vrijednosti akcija u svakom stanju možemo predstaviti tablicom u kojoj stupci predstavljaju moguća stanja u kojima se model može naći, a redci sve akcije koje agent može odigrati. Na Slici 2.1 je prikazana igra u 2d prostoru i pripadajuća početna Q-tablica u kojoj je svaka Q-vrijednost jednaka 0.



Slika 2.1: Q-tablica od ADL (2018)

Q-vrijednost u svakom koraku se računa pomoću sljedeće formule Tutek (2018):

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')] \quad (2.1)$$

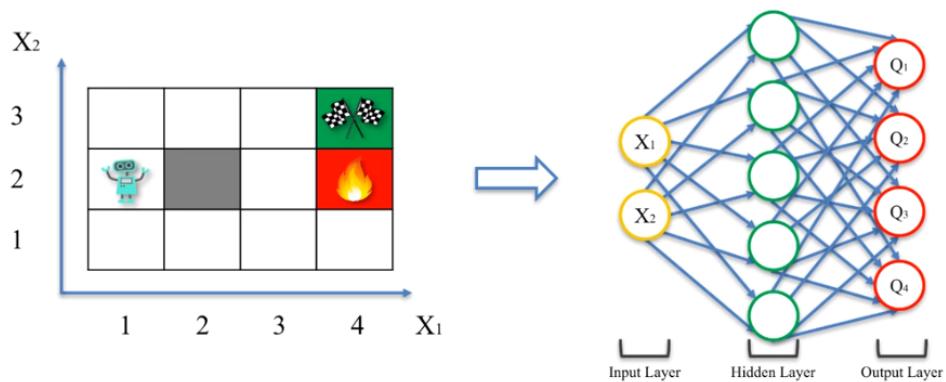
gdje je s stanje prije odabira akcije, a akcija koju je programski agent odigrao, s' stanje u koje je igra prešla nakon izvođenja akcije, R funkcija nagrade (engl. *reward*), a T funkcija prijelaza (engl. *transition*). Ovakvo mijenjanje Q-vrijednosti sliči stohastičkom gradijentnom spustu jer tijekom treniranja model radi mnogo nasumičnih akcija što osigurava istraživanje prostora stanja van Hasselt et al. (2015).

Funkciju odabira akcije u svakom stanju možemo definirati na više načina, a zovemo je *politika odabira* (engl. *exploration policy*). Najjednostavnija politika odabira je tzv. *argmax* politika; uvijek odabiremo akciju koja ima najveću Q-vrijednost. Za testiranje modela ovo je ujedno i optimalna politika van Hasselt et al. (2015), uz pretpostavku da su u modelu postignute optimalne Q-vrijednosti. *Boltzmannova* politika odabira pak nalaže da veličina Q-vrijednosti označava vjerojatnost odabira pojedine akcije. Ta politika se pokazala dobrom u stohastičkim sustavima gdje prijelaz iz jednog stanja u drugo nije moguće deterministički odrediti.

2.3. Duboko Q-učenje

Duboko Q-učenje ekstrapolira nad idejom Q-učenja, ali umjesto pohranjivanja Q-vrijednosti u tablicu se Q-vrijednosti akcija izračunavaju uz pomoć neuronske mreže. Neuronska mreža je funkcija iz \mathbb{R}^n u \mathbb{R}^m . Ulaz neuronske mreže je vektor n značajki (prostor stanja) gdje n predstavlja broj značajki koje opisuju trenutno stanje. Izlaz neuronske mreže je vektor m značajki (prostor akcija) gdje m predstavlja maksimalni broj akcija koji možemo odigrati u bilo kojem trenutku unutar igre. Izlazni vektor

je polje Q-vrijednosti. Prednost dubokog Q-učenja jest zauzimanje manje memorije jer se problem svodi na pronalaženje optimalnih težina neurona u skrivenom sloju neuronske mreže. Ta metoda radi jer programski agenti mogu naučiti prepoznati sličnosti situacija u kojima se nalaze i odabrati najbitnije značajke trenutnog stanja na temelju kojih će odabratи akciju. Arhitektura mreže jest proizvoljna, no u mnogim radovima je ideja imati jedan skriveni sloj veličine t gdje je t skalar između m i n kada je stanje reprezentirano konkretnim informacijama iz igre, no ako je stanje reprezentirano vrijednostima piksela na kadru ekrana onda je mreža često kompleksnija. Prikaz jedne takve neuronske mreže možemo vidjeti na Slici 2.2.



Slika 2.2: Duboka Q mreža sa mc.ai

Mijenjanje težina skrivenog sloja neuronske mreže se radi pomoću algoritma rasprostiranja (prenošenja) unatrag (engl. *backpropagation*). U svakom koraku podesimo samo Q-vrijednost akcije koje je programski agent odlučio odigrati:

$$Q_{k+1} \leftarrow R_{k+1} + \gamma \max_{a'} Q_k(s', a') \quad (2.2)$$

Dva važna poboljšanja koja su predložili Mnih et al. (2015) su korištenje neuronske mreže za dobivanje Q-vrijednosti te metodu *ponavljanja iskustva* (engl. *experience replay*) van Hasselt et al. (2015). Ponavljanje iskustva, Lin (1992), je tehniku kojom programski agent tijekom igranja igre umjesto podešavanja težina nakon prelaska iz jednog u drugo stanje spremi informacije u memoriju te svakih x koraka nasumičnim redoslijedom uzima informacije prijelaza iz memorije i uči neuronsku mrežu.

2.4. Duplo duboko Q-učenje

U običnom dubokom Q-učenju jedna neuronska mreža se koristi i za odabir akcije i za evaluaciju dobrote pojedine akcije u procesu učenja. To je loše jer se može dogoditi da mreža repetitivno bude nagrađivana za odabir akcije koja donosi suboptimalnu pozitivnu nagradu čime se Q-vrijednost te akcije poveća preko optimalne vrijednosti (engl. *overestimation*). Kako bi smanjili preoptimističnost algoritma van Hasselt et al. (2015) predlažu uvođenje još jedne neuronske mreže jednake arhitekture kao i originalna mreža. Jedna mreža se koristi za odabir akcije, a druga za evaluaciju dobrote akcije. Svakih θ koraka težine u mrežama se sinkroniziraju. Ovime preoptimističnost ne nestaje, ali se događa samo netom nakon sinkronizacije težina neuronskih mreža. Metoda ponavljanja iskustva značajno smanjuje vrijeme treniranja programskog agenta te omogućava postizanje boljih rezultata jer se smanjuje ovisnost uzastopnih stanja.

2.5. Dugo kratkotrajna memorija

Problem kod dosad navedenih tehnika potpornog učenja jest što odabiru akciju samo na temelju trenutnog stanja (ili zadnjih n stanja). Za postizanje nadljudskih sposobnosti u mnogim igrama programski agenti moraju naučiti ostvarivati dugoročne ciljeve korištenjem strategija. Metodom dugo kratkotrajne memorije (engl. *Long short-term memory*) programski agenti mogu koristiti arbitarno duge nizove informacija iz prošlih stanja. Iz tog razloga nije moguće treniranje neuronske mreže slučajnim redoslijedom akcija koristeći memoriju iskustva što kao posljedicu ima dug period treniranja OpenAI et al. (2019). Dugo kratkotrajna memorija je poopćenje metode korištenja povratnih neuronskih mreža (engl. *recurrent neural networks*) koja smanjuje problem teženja gradijenata u 0 ili u beskonačnost Hochreiter i Schmidhuber (1997). Neuroni dugo kratkotrajne memorije su značajno kompleksniji od perceptronova. Sadrže stanicu (engl. *cell*) i tri vrata (engl. *gate*); ulazna vrata (engl. *input gate*), izlazna vrata (engl. *output gate*) i vrata zaborava (engl. *forget gate*). Stanica pamti vrijednosti preko temporalnih nizova arbitrarne duljine, a vrata reguliraju protok informacija u i iz stanice. Osim u kreiranju programskih agenata za složene videoigre metoda dugo kratkotrajne memorije se koristi u prepoznavanju rukopisa, govora i detekciji anomalija u prometu.

3. Popularne igre, njihove složenosti i poznati programski agenti

3.1. Složenost igara

Uspješnost izrade programskog agenta za neku igru ponajprije ovisi o kompleksnosti igre odnosno njenoj složenosti. Složenost igre pojednostavljeno možemo prikazati s tri vrijednosti; veličina prostora stanja, veličina prostora akcija te duljina trajanja igre. Većina igri imaju fiksan prostor stanja, količina mogućih akcija ovisi o trenutnom stanju, a duljina trajanja igre varira s obzirom na odabране akcije. Prostor stanja možemo predstaviti vektorom s n značajki. Količinu mogućih akcija možemo predstaviti skalarom koji će predstavljati srednju vrijednost broja akcija kojih možemo poduzeti u svim trenucima igre. Duljinu trajanja igre možemo predstaviti srednjim brojem poteza koji stignemo odigrati do završetka igre.

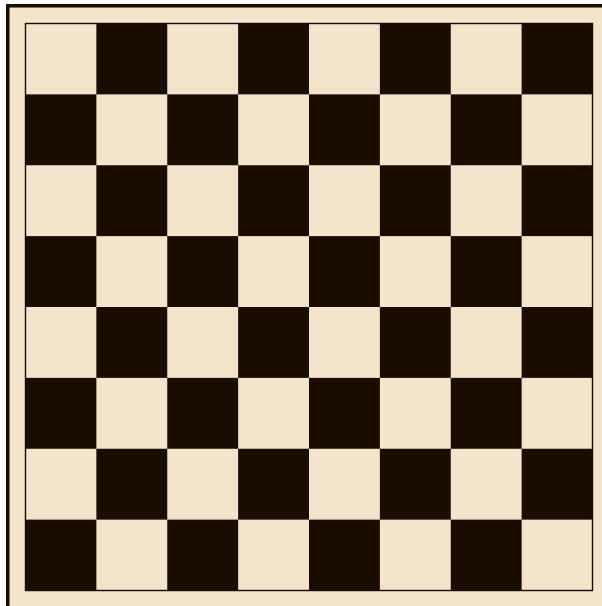
3.2. Šah

3.2.1. Opis igre

Šah je društvena igra za dva igrača koja se igra na kvadratnoj ploči od 64 polja prikazanoj na Slici 3.1. Igrači imaju šesnaest bijelih, odnosno crnih figura, a figure mogu biti jednog od šest različitih tipova; pijun, top, skakač, lovac, kraljica i kralj. Figure se kreću u raznim smjerovima, a cilj igre je *zarobiti* protivnikovog kralja.

3.2.2. Složenost igre

Prostor stanja šaha možemo prikazati vektorom od 64 značajke gdje svaka značajka određuje boju i tip figure koja se nalazi na pojedinom polju. U svakom potezu u prosjeku moguće je odigrati 40 različitih akcija, a maksimalno 218. Prosječno trajanje



Slika 3.1: Ploča za šah sa howtofixx.com

partije šaha jest između 40 i 70 poteza.

3.2.3. Poznati programske agenci

Šah je jedna od najpopularnijih igara za treniranje softverskih agenata. Dosad najbolji programski agenti koristili su kombinacije sofisticiranih algoritama pretraživanja stanja, domenskih strategija i ručno izrađenih evaluacijskih funkcija koje su podešavali ljudski eksperti. No od nedavno samotrenirajući modeli potpornog učenja kao AlphaZero u potpunosti dominiraju šahovsku scenu Silver et al. (2018). AlphaZero jest programski agent koji je sposoban naučiti mnogo igara i postići nadljudske rezultate igrajući partije sam protiv sebe. Nije mu potrebno nikakvo domensko znanje osim pravila igre.

3.3. Go

3.3.1. Opis igre

Go je društvena igra za dva igrača koja se igra na kvadratnoj ploči od 361 polja (19×19) koja se naziva *goban*. Polja su sjecišta linija za razliku od ploče za šah gdje polje označava svaka ćelija u mreži. Igrači koriste kamenčiće; bijele odnosno crne boje. Cilj igre je posjedovati što više teritorija okruženih svojim kamenčićima.

3.3.2. Složenost igre

Prostor stanja igre go možemo prikazati vektorom od 361 značajke gdje svaka značajka određuje boju kamenčića na polju. U svakom potezu u prosjeku moguće je odigrati 250 akcija, a maksimalno 361. Prosječna duljina trajanja partije igre go jest 150 poteza.

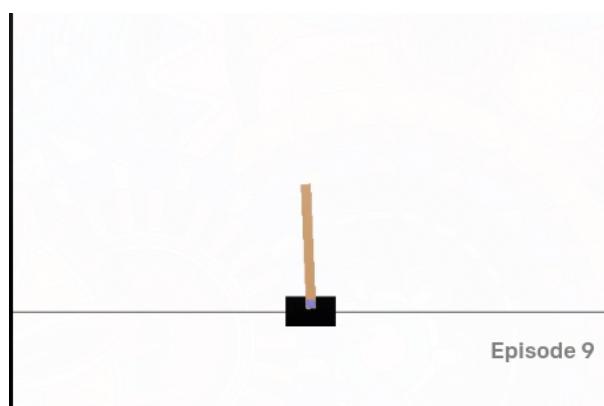
3.3.3. Poznati programski agenti

2015. AlphaGo je prvi programski agent koji je pobijedio ljudskog experta bez hendi-kepa na 19×19 ploči. Otada su se razvili AlphaGo Master, AlphaGo Zero i AlphaZero naveden ranije.

3.4. Balansiranje štapa

3.4.1. Opis igre

Ekvivalent prikazu ispisa *Hello World!* pri učenju novog programskog jezika u okviru učenja metoda potpornog učenja jest rješavanje problema balansiranja štapa (engl. *cartpole*) Brockman et al. (2016). Okruženje je dostupno na gym.openai.com. Cilj igre jest balansirati štap koji стоји вертикално на помičnom autiću. Okruženje se nalazi u 2d prostoru. Na Slici 3.2 prikazan je zaslon igre. Crnim pravokutnikom je prikazan autić koji se može kretati lijevo desno. Sivim pravokutnikom je prikazano dno štapa, a smeđim izduženim pravokutnikom ostatak štapa.



Slika 3.2: Zaslon igre balansiranja štapa

3.4.2. Složenost igre

Prostor stanja je vektor veličine 4 značajke; brzina kretanja autića, pozicija autića u odnosu na sredinu ekrana, nagib štapa i kutna brzina u vrhu štapa. U svakom trenutku moguće je odigrati 2 akcije; povećati silu uljevo ili povećati silu udesno. Cilj je što duže balansirati štap bez da autić izađe iz ekrana. Kada se dostigne 200 koraka u osnovnoj inačici igre, odnosno 500 koraka u naprednoj bez da se štap nagnuo preko 15 stupnjeva smatra se da je igrač pobijedio.

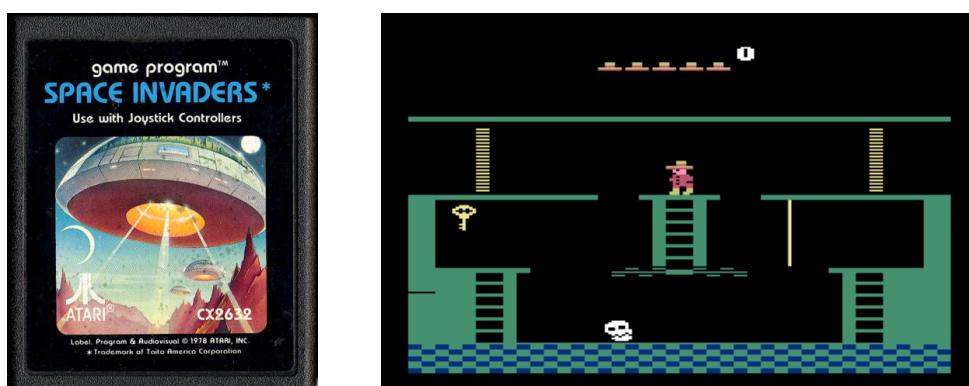
3.4.3. Poznati programski agenti

Za ovu igru ne postoje poznati programski agenti jer je igra vrlo jednostavna. Moguće je primijeniti bilo koju tehniku učenja iz poglavlja *Tehnike potpornog učenja* i ako se model dovoljno dugo uči trebao bi moći pobijediti igru u 95% slučajeva. Za potrebe ovog rada razvijen je model duplog dubokog Q-učenja koji je uspješno testiran na ovoj igri, više o tome u poglavlju *Izrada modela potpornog učenja*.

3.5. Atari videoigre

3.5.1. Opis igre

Atari igre su videoigre za nekoć popularnu konzolu Atari. Većina igara su vrlo jednostavne, a prostor stanja i akcija im je značajno manji od popularnih videoigara današnjice. Mnoge su dostupne na gym.openai.com. Imaju rezoluciju 210×160 piksela, a svaki piksel je u boji prikazanoj kao vektor 3 značajke.



Slika 3.3: Naslovница igre *Space Invaders* od Breininger (2018) i prikaz zaslona igre *Montezuma's Revenge* od Scott (2013)

3.5.2. Složenost igre

Prostor stanja Atari videoigara se može prikazati na dva načina. Jedan način je kadrom ekrana. Svaki piksel na ekranu prikazan je RGB bojom. Tako prostor stanja može biti vektor značajki veličine $210 \times 160 \times 3$ tj. 100800. Drugi način prikaza prostora stanja je radnom memorijom dok se igra odvija. U prosjeku moguće je odigrati 5 poteza u nekom stanju, a maksimalno 18 poteza. Prosječna duljina trajanja igre jest 5000 poteza.

3.5.3. Poznati programske agenci

Postoje mnogi radovi na temu izrade modela potpornog učenja za Atari videoigre. Najpoznatiji su Mnih et al. (2013, 2015); van Hasselt et al. (2015); Kulkarni et al. (2016). Originalni rad koristi metodu dubokog Q-učenja koristeći zadnja 4 kadra ekrana kao ulaz neuronske mreže. Umjesto izbora slučajnih akcija u procesu učenja koriste se akcije ljudskih eksperata radi bržeg postizanja dobrih rezultata. To značajno ubrzava proces učenja no smanjuje vjerojatnost pretraživanja prostora stanja te je vjerojatno da je doseg takvog programskog agenta manji Silver et al. (2018). Neovisno o tome sva 4 rada su uspješno postigla nadljudske sposobnosti na nekim igrama. Od originalnog rada su uvedena sljedeća poboljšanja: igranje iz iskustva, promjena modela za pristup duplim dubokim Q-učenjem i pristup hijerarhijskim dubokim Q-učenjem koje koristi hijerarhijsku duboku neuronsku mrežu (engl. *hierarchical deep neural network*).

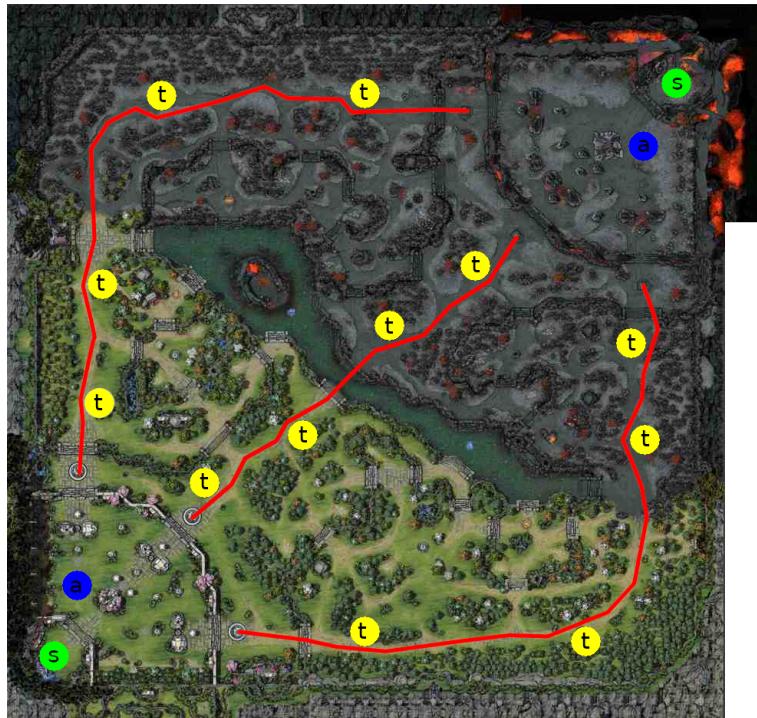
3.6. DOTA 2

DOTA 2 je videoigra za više igrača u kojoj, putem interneta, igrači pokušavaju osigurati pobjedu u virtualnoj arenici (engl. *MOBA - Multiplayer Online Battle Arena*). Firma Valve Corporation izdala je igru 2013. godine. Od 2016. godine pa do danas DOTA 2 je jedna od najpopularnijih igara na svijetu s preko pola milijuna stalnih igrača.

3.6.1. Opis igre

Dva tima po 5 igrača se natječu unutar jedne arene tko će prije uništiti glavnu građevinu (engl. *Ancient*) drugog tima. Slika 3.4 prikazuje tlocrt arene koji igrači u svakom trenutku vide u donjem lijevom kutu ekrana. Mjesto stvaranja (engl. *spawn point*) jest lokacija u arenici gdje se igrač stvori na početku igre te svaki puta kada izgubi život. Arena je podijeljena na 3 trake (engl. *lane*); gornja traka (engl. *top*), srednja

traka (engl. *mid*) i donja traka (engl. *bottom*). Na svakoj traci postoje po 4 tornja (engl. *tower*), dva tornja pripadaju jednom timu, a druga dva drugom timu.



Slika 3.4: DOTA 2 Minimap

█ lanes	█ towers
█ ancients	█ spawn points

Heroji

Svaki igrač upravlja jednim od 119 različitih *heroja*. Ono što im je svima zajedničko jest da imaju ograničenu količinu života (engl. *health points*) te ograničenu količinu energije (engl. *mana points*). Herojima se upravlja u stvarnom vremenu, a svaki od njih ima jedinstvenu pasivnu sposobnost i 4 različite akcije koje može odigrati. Heroj se u bilo kojem trenutku može kretati u dvodimenzionalnom svijetu, napraviti običan napad koji radi malu količinu štete protivnicima ili iskoristiti akciju. Svaka akcija heroja košta neku količinu energije da bi je odigrao te većina akcija ozljeđuje druge (protivnikove) heroje za određenu količinu životnih bodova. Kada heroju životni bodovi padnu na 0 on umire te se ponovno stvara na početku svoje strane arene.

Ratnici

Prije početka igre igrači imaju jednu minutu da kupe potrebite predmete (engl. *items*) te se pripreme za nadolazeću bitku. Nakon prve minute pripreme, igra kreće, a početak igre označava stvaranje prvog vala ratnika (engl. *creeps*) na svakoj traci. Ratnici su računalom upravljeni agenti koji se stvaraju svakih 30 sekundi u valovima. Jedan val ratnika sastoji se od nekoliko ratnika koji napadaju mačem i jednog ratnika koji može napadati na daljinu. Ratnici su napravljeni tako da prate put trake na kojoj se stvore, a jedino što ih može odvratiti od puta jest bilo koja vrsta neprijatelja. Čim se u vidokrugu ratnika nađe neprijatelj, ratnik će ga pratiti i napadati dok god neprijatelj ne bude uništen ili mu nestane iz vidokruga. Heroji mogu napadati i protivnikove i vlastite ratnike, a onaj heroj koji je zadao fatalni udarac (engl. *last hit*) ratniku je nagrađen zlatom.

Sučelje igre



Slika 3.5: Zaslon igre

Slika 3.5 prikazuje tipičnu scenu iz igre. Sučelje koje igrač vidi tijekom igre sastoji se većinom od vlastitog vidokruga arene koja je prikazana iz trodimenzionalne izometrične perspektive. Na zaslonu se nalaze dva heroja, jedan iznad kojeg je zeleni pravokutnik i drugi iznad kojeg je crveni pravokutnik. Oba pravokutnika predstavljaju količinu života koju heroj trenutno posjeduje. Između dva heroja nalaze se crveni i zeleni ratnici, iznad njih se također nalaze pokazatelji količine života. Heroje od ratnika

možemo razaznati veličinom likova i prisustvom male slikice uz količinu života iznad njihovog lika. U gornjem lijevom kutu nalaze se tipke za promjenu postavki igre te po-koja informacija o vlastitom heroju (npr. ukupan broj ubojstava (engl. *kills*) i ukupan broj smrti (engl. *deaths*)). U gornjem dijelu ekrana nalaze se slike svih heroja koji su trenutno u igri, a u samoj sredini nalazi se vrijeme trajanja bitke. U donjem lijevom kutu nalazi se mali prikaz tlocrta cijele arene. U donjem desnom kutu igrač može vi-djeti količinu zlata kojeg trenutno ima. Na dnu ekrana (s lijeva na desno) nalaze se: slika i ime heroja, akcije i sposobnosti, količina života i energije te predmeti koje je igrač kupio (9 slobodnih pravokutnika na slici).

3.6.2. Složenost igre

Prostor stanja možemo prikazati s 16000 značajki koje predstavljaju unutrašnje stanje same igre. Npr. unutra se nalaze pozicije svih heroja, drveća i ratnika, njihovi životni bodovi, akcije, predmeti itd. Prosječan broj akcija koje je moguće odigrati u pojedinom trenutku jest između 8000 i 80000 OpenAI et al. (2019). Prosječna duljina trajanja igre jest 20000 poteza.

3.6.3. Poznati programski agenti

13.4.2019. *OpenAI Five* agent je postao prvi sustav umjetne inteligencije koji je uspio pobijediti tim ljudskih eksperata u aktualnoj videoigri. Programski agent kontrolira ci-jeli tim heroja od 5 igrača koji međusobno surađuju. Koristi tehniku dugo kratkotrajne memorije kombiniranu s pažljivo primijenjenim domenskim znanjem. Za ovakav us-pjeh na igri čiji je prostor stanja nezamislivo velik agentu je bilo potrebno preko 10000 godina igranja.

3.7. Pojednostavljena verzija DOTA 2 igre

3.7.1. Opis igre

Pojednostavljena verzija DOTA 2 igre (koristeći sučelje Breezy) se igra u dva igrača, jedan protiv drugoga. Oba igrača igraju istog heroja te igra prestaje kada jedan igrač pobijedi drugoga dva puta unutar jednog meča. Od 119 heroja koje DOTA 2 ima moguće je odabrati samo jednog. Vektor značajki prostora stanja dan je tablicom 9.2 koja se nalazi u zadnjem poglavljju na stranici 39.

Tablica 3.1: Moguće akcije

Index	Opis akcije
0	Ne radi ništa
1–8	Akcije kretanja
9–12	Akcije odlaska po pojačanja
13	Napadni protivnikovog heroja
14	Napadni protivnikov toranj
15–19	Napadni protivnikovog ratnika
20–24	Napadni prijateljskog ratnika
25	Ozljedi sve [-50, 450]m ispred sebe
26	Ozljedi sve [200, 700]m ispred sebe
27	Ozljedi sve [450, 950]m ispred sebe
28	Ozljedi sve oko sebe u krugu od 1350m
29	Povrati 400 životnih bodova

Skraćena verzija prostora akcija dana je tablicom 3.1, a originalna verzija, tablica 9.1, se nalazi u zadnjem poglavlju na stranici 38.

Programski agenti se ne moraju brinuti o kupovanju predmeta niti poboljšavanju vlastitih akcija. Te se stvari odvijaju automatski kako bi se smanjila kompleksnost igre. Uvjet pobjede postiže se rušenjem jednog protivnikovog tornja ili ubojstvom protivnikovog heroja 2 puta. Glavne značajke u kojima jedan heroj može dobiti značajnu prednost nad protivnikom jest efektivno skupljanje zlata, umanjivanje zlata protivniku, ubijanje protivnika i rušenje tornja.

3.7.2. Složenost igre

Sučelje Breezy eksponira vektor od 310 značajki koji opisuje trenutno stanje igre. Za prostor stanja bez domenskog znanja moguće je uzeti vektor od 310 značajki, no tako bi proces učenja bio bespotrebno dug. Kako bi smanjili trajanje procesa učenja i povećali kvalitetu naučenih strategija ćemo iz vektora od 310 značajki izračunati 50 bitnih informacija. Prostor stanja će predstavljati 50 bitnih informacija iz zadnja 4 stanja što čini vektor od 200 značajki. Maksimalno 30 akcija moguće je odigrati u danom trenutku. U prosjeku se stigne odigrati 130 akcija po igri.

3.8. Sažetak složenosti navedenih igara

Tablica 3.2: Usporedba navedenih igara i programske agenata primjenjenih na njima

	Šah	Go	Balansiranje štapa	Atari igre	Dota 2	Breezy Dota 2
Prostor stanja	64	361	4	100800	16000	200
Prostor akcija	64	361	2	18	80000	30
Broj poteza	70	150	200	5000	20000	130
Potpuno znanje	✓	✓	✓			
Determinizam	✓	✓	✓			
Domensko znanje					✓	✓
Metoda učenja	RL	RL	DDQN	DDQN, HQN	LSTM	DDQN
Vrijeme treninga	4 sata	40 dana	12 sati	1 tjedan	10 mjeseci	10 dana
Sklopoljje	5000 TPUs	5000 TPUs	1 GPU	1 GPU	1500 GPUs	1 GPU

Vrlo je teško direktno usporediti igre i njihovu složenost odnosno predvidjeti uspješnost programskog agenta samo na temelju brojeva u tablici. Bitno je naglasiti kručajne razlike u konceptima izvođenja navedenih igara. Šah, go i balansiranje štapa su jedine navedene igre u kojima agent ima potpuno znanje o stanju igre. Ujedno, te igre su u potpunosti determinističke. U Atari igrama, igri DOTA 2 i većini ostalih videoigara igrači imaju parcijalno znanje o trenutnom stanju igre, a mnoga ponašanja su opisana stohastičkim procesima. Značajke složenosti koje odjeljuju popularne igre današnjice od jednostavnih videoigara prošlog stoljeća čine znatno veću razliku u kompleksnosti nego navedena dva faktora. Na primjer, u većini današnjih videoigra igrač nikad nije ‘na potezu’, igre se odvijaju u stvarnom vremenu što znači da je broj akcija u sekundi ograničen samo brzinom odabira akcije igrača, brzinom komunikacije igrača i računala na kojem se igra odvija (najčešće putem interneta) i brzinom obrade informacije same igre. To ograničava vrijeme razmišljanja igrača. Kako bismo drastično smanjili vrijeme treniranja agenata potpornog učenja na modernim videoigramama agentu je potrebno domensko znanje o samoj igri. Kao posljedicu navedena metoda ima da agenti neće jednako pretraživati prostor stanja te je vjerojatnost dosezanja optimalne politike smanjena.

4. Izrada modela potpornog učenja

4.1. Odabir tehnike potpornog učenja

Izrada modela programskog agenta je često neovisna o igri na kojoj se agent pokreće, no odabir igre drastično ograničava odabir tehnike potpornog učenja koja će biti najefektivnija u ograničenom vremenu učenja. Za potrebe ovog rada 10 dana učenja je razumna količina vremena u kojoj bi agent trebao vidljivo napredovati. Nije za očekivati da će agent postići nadljudske rezultate u tako malo vremena neovisno o tehnički potpornog učenja koja se odabere. Igra na kojoj će programski agent biti testiran jest *pojednostavljena verzija igre DOTA 2* navedena u prošlom poglavljju. Sučelje Breezy će biti korišteno za dobivanje informacija o trenutnom stanju igre. Igra se igra u stvarnom vremenu što znači da broj akcija kojih agent stigne odigrati u jednoj sekundi ovisi o brzini odabira akcija i prijenosa tih informacija igri. U prosjeku agent stigne odigrati između 3 i 8 akcija u jednoj sekundi u stvarnom vremenu. Prosječno trajanje jedne igre jest 10 minuta vremena igre, što su 2 minute u stvarnom vremenu. Vrijeme potrebno za pokretanje nove igre je 15 sekundi. To znači da će agent u najgorem slučaju moći odigrati $10 \times 24 \times 60 \times 3 \times (2 \times 60 - 15) \div 2 \approx 2\text{milijuna}$ akcija. To je otprije jedan red veličine manje akcija nego što su Mnih et al. (2015) i van Hasselt et al. (2015) dopustili programskim agentima da treniraju na Atari igrama. Oni su trenirali agenta temeljenog na tehnički duplog dubokog učenja na 50 milijuna koraka, no u oba rada je vidljiv pomak dosega programskega agenta već nakon 100 tisuća koraka. Za razliku od agagenta dugotrajne memorije kojeg su koristili OpenAI et al. (2019) koji se pokretao na tisuće grafičkih kartica u periodu od nekoliko mjeseci što je u potpunosti neusporedivo. Iz tog razloga odlučili smo se na implementaciju agagenta duplog dubokog učenja u kombinaciji s ručno izrađenim metodama radi ubrzavanja napretka programskega agagenta.

4.2. Okruženje i tehnologije

Za izradu programskog agenta izabran je jezik python radi dobre podrške sučelja za potrebe dubokog učenja i postojanja biblioteke gotovih igri na kojima agent može biti testiran. Verzija pythona korištena za pokretanje programskog agenta je *python 3.8.2*. Biblioteka koja je korištena za potrebe izrade modela dubokog učenja jest *tensorflow 2.2.0-rc4* koja koristi grafičku karticu. Grafička kartica na kojoj se pokretao programski agent je *Nvidia GeForce GTX 1060 6GB*. Probna igra za testiranje modela potpornog učenja je igra *balansiranje štapa* navedena u prošlom poglavljiju iz biblioteke *gym 0.17.1*.

4.3. Implementacija modela programskog agenta

Uvjet pobjede igre balansiranja štapa zahtjeva barem 95 pobjeda u 100 igara tijekom faze testiranja. Svakim korakom kojim agent ne izgubi igru nagrađen je za 1 bod. Maksimalni broj bodova koji agent može postići je 500. Uvjet prestanka učenja je dostizanje 475 bodova kao prosjek zadnjih 100 partija.

```
max_steps = 500
win_tolerance = 0.05

...
while last_100_games_avg_score < max_steps*(1 - win_tolerance):
    ...
```

Dvije mreže istih arhitektura su postavljene prije početka učenja; *q_network* i *t_network* (engl. *target network*). Arhitektura mreže se sastoji od 4 ulaza, 2 izlaza i jednog skrivenog sloja od 16 neurona. Aktivacijska funkcija je tangens hiperbolni, a za optimizator se koristi *Adam*. Funkcija gubitka je funkcija srednje kvadratne pogreške (engl. *mean square error*). Zbog mogućnosti zaustavljanja programa i ponovnog pokretanja sa zapamćenim težinama moguće je specificirati ime datoteke u kojoj će biti spremljene težine skrivenog sloja mreže i iste učitane pri pokretanju programa.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```

```

def create_network(filename=''):
    network = Sequential()
    network.add(Dense(16, input_dim=4, activation='tanh'))
    network.add(Dense(2, activation='linear'))
    if os.path.isfile(filename):
        network.load_weights(filename)
    network.compile(loss='mse', optimizer=Adam(lr=0.001))
    return network

weights_path = 'q_network.h5'
q_network = create_network(filename=weights_path)
t_network = create_network()

epsilon = 1 if training else 0
epsilon_min = 0.001
epsilon_decay = 0.999

...
if np.random.uniform() > epsilon:
    q_values = predict(q_network, state)
    action = np.argmax(q_values)
else:
    action = env.action_space.sample()

...
if epsilon > epsilon_min:
    epsilon *= epsilon_decay

```

Kodnim isječkom prikazan je način odabira akcije. Ako je slučajni broj na intervalu $[0, 1]$ veći od ϵ onda mreža odabire akciju, inače je akcija nasumično odabrana.

Veličina memorije iskustva je 2048, a nakon svake partije igre se mreža uči na 32 nasumična iskustva iz memorije iskustva.

```

batch_size = 2**5
memory = deque(maxlen=2**11)
discount_rate = 0.96

...
batch = random.sample(memory, batch_size)
for state, action, reward, state_ in batch:
    future_reward = discount_rate *
                    np.amax(predict(t_network, state_))

    q_value = reward * (1 + future_reward)
    target = predict(q_network, state)
    target[action] = q_value
    q_network.fit(to_array(state), to_array(target), epochs=1)

```

Kodni isječak pokazuje kako se Q-vrijednosti mijenjaju u procesu učenja koristeći memoriju iskustva. Iz memorije iskustva se izabiru 32 nasumična uzorka. Svaki uzorak sadrži trenutno stanje, akciju koju je Q-mreža odabrala, nagradu i stanje u koje je igra prešla. Za svaki uzorak se računa buduća nagrada umanjena za neki postotak, u ovom slučaju 4%. Varijabla *q_value* predstavlja željenu Q-vrijednost akcije koju je Q-mreža odabrala. Povratnim propagiranjem se podešavaju težine mreže kako bi bolje aproksimirali željenu Q-vrijednost. Važno je za napomenuti da se Q-vrijednosti drugih akcija ne podešavaju.

Svakih 1000 koraka se težine skrivenog sloja mreža sinkroniziraju.

```

steps = 0
sync_target_steps = 1000

...
if steps % sync_target_steps == 0:
    t_network.set_weights(q_network.get_weights())

steps += 1

```

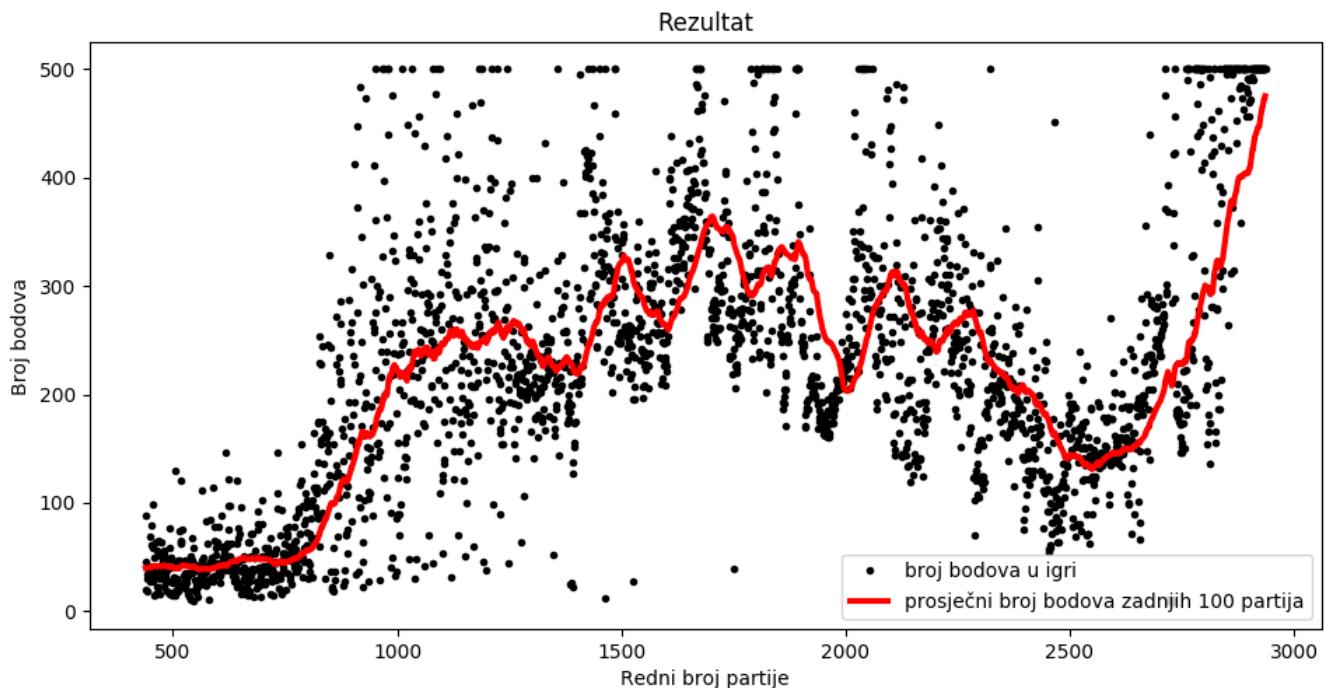
Nakon svake igre program ispisuje redni broj igre, postignut rezultat, epsilon vrijednost i prosječni rezultat zadnjih 100 igri.

```
f' run: {games_played:4d}, score: {env._elapsed_steps:3d}, '
f' epsilon: {epsilon:.4f}, avg: {running_sum/len(scores):3.2f}'
```

U elektronskim dodacima ovom radu se nalazi puni izvorni kod.

4.4. Analiza rezultata

Proces učenja agenta trajao je svega nekoliko sati i završio je uspješno. Uvjet zauzavljanja bio je dosezanje prosječnog rezultata u zadnjih 100 partija od 475, što je 5 postotno odstupanje od 500. Na slici 4.1 nalazi se prikaz rezultata tijekom procesa učenja.



Slika 4.1: Prikaz rezultata u procesu učenja na igri balansiranja štapa koristeći metodu duplog dubokog učenja

Sa slike možemo primijetiti kako agent uči balansirati štap relativno brzo no onda razlog gubljenja postane izlazak s ekrana. Kada agent shvati da se mora fokusirati na ostanak unutar ekrana štap mu počne ponovo padati. Od tog trenutka nadalje agent shvati da je cilj ostati što bliže sredini ekrana no s obzirom na to da se previše kreće štap mu i dalje često padne. Kad agent shvati da je cilj minimizirati kutnu brzinu u vrhu štapa onda krene rapidno učiti kao što je vidljivo od koraka 2500 nadalje. Da agent nije ograničen prestankom igre na 500 koraka nastavio bi dosezati tisuće koraka bez prestanka. Od tog trenutka imalo bi smisla povećati veličinu iskustvene memorije.

Puni izvorni kod za dobivanje danog grafa dan je u nastavku:

```
from re import.findall
import matplotlib.pyplot as plt
```

```

with open('cartpole-v1-training.txt') as f:
    e, s, _, a = zip(*[map(float,.findall(r'\d+\.\?\d*', line))
                        for line in f.read().split('\n')])

```

```

plt.title('Rezultat')
plt.xlabel('Redni broj partije')
plt.ylabel('Broj bodova')
plt.plot(e, s, 'k.', label='broj bodova u igri')
plt.plot(e, a, 'r', linewidth='3',
         label='prosječni broj bodova zadnjih 100 partija')
plt.legend()
plt.show()

```

U procesu testiranja agent je pobijedio 100 partija za redom. Nagrada tijekom procesa učenja ovisila je samo o tome je li nakon tog koraka igra bila gotova ili se nastavila. Dakle agent nije imao nikakvog znanja o samoj igri. S obzirom na to da je ova igra deterministički određena zakonima fizike; umjesto igranja slučajnih poteza tijekom testiranja mogli smo odabirati analitički najbolju akciju. Tada bi agent drastično brže naučio balansirati štap.

5. Projekt Breezy

Projekt Breezy je programsko sučelje za videoigru *DOTA 2* koje omogućava softverskim agentima da dobivaju informacije te izvode akcije u stvarnom vremenu unutar igre. Projekt je osmišljen za pojednostavljenu verziju originalne igre objašnjenu u prošlom poglavlju. Sučelje nudi vektor od 310 različitih informacija koje programski agent može koristiti tijekom igre kako bi donio odluku što sljedeće odigrati. S druge strane broj mogućih akcija koje može odigrati u bilo kojem trenutku jest 29. Više informacija dostupno je na https://web.cs.dal.ca/~dota2/?page_id=353.

5.1. Integracija

Sučelje radi kao dva odvojena sustava; kao zaseban server za komunikaciju sa softverskim agentom te kao dodatak (engl. *add-on*) igri *DOTA 2*. Dodatak igri služi kao potprogram unutar same igre koji radi akcije koje mu je poslao server. Server od dodatka prima podatke o trenutnom stanju igre te ih proslijeđuje softverskom agentu koji nakon donošenja odluke koju akciju će napraviti serveru šalje odgovor. Komunikacija se odvija putem http zahtjeva. U programu softverskog agenta potrebno je implementirati tri rute (engl. *route*); *connect*, *relay*, *update*. *Connect* rutom programski agent javlja serveru da je spremjan za početak igre. *Relay* rutu server poziva kada programski agent mora donijeti odluku koju akciju će napraviti, ovdje programski agent prima trenutno stanje igre u obliku *json* formata. *Update* ruta se poziva između mečeva kako bi programski agent imao više vremena za učenje ako mu je to potrebno.

5.2. Instalacija

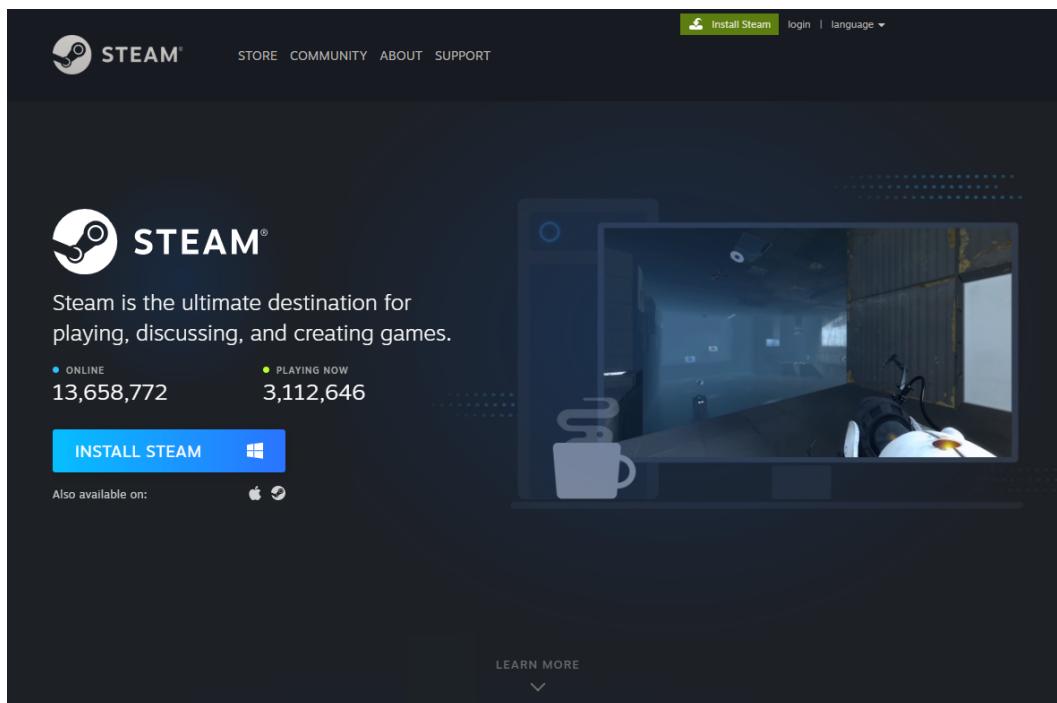
5.2.1. Zahtjevi

Prije instalacije sučelja Breezy potrebno je naglasiti da je moguće koristiti samo operacijski sustav Windows za pokretanje igre i sučelja, no softverski agent se može po-

kretati na odvojenom računalu na bilo kojem operacijskom sustavu. Računalo koje će pokretati igru mora imati program *java* 8. Izvršnu datoteku za instalaciju programa *java* 8 moguće je skinuti sa <https://adoptopenjdk.net/> ili <https://oracle.com/java/technologies/javase/javase-jdk8-downloads.html>.

5.2.2. Igra DOTA 2

Za početak potrebno je instalirati platformu Steam, to je moguće napraviti na sljedećem linku store.steampowered.com/about pritiskom na gumb *INSTALL STEAM* kako je prikazano na Slici 5.1.

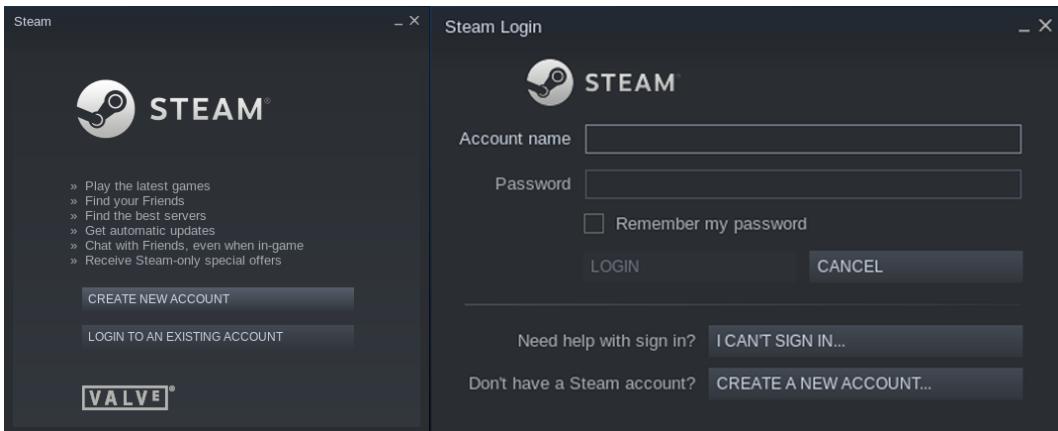


Slika 5.1: Instalacija platforme Steam

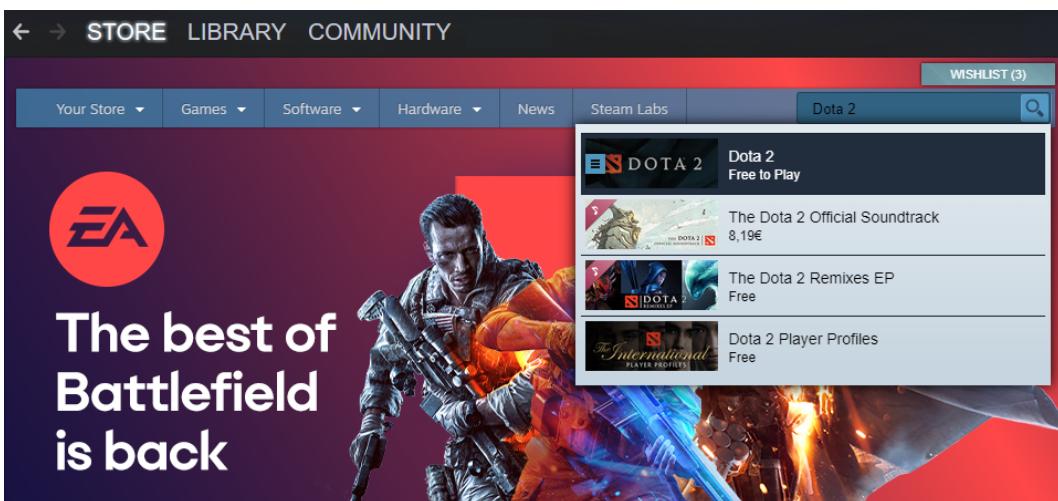
Nakon instalacije platforme, na istoj je potrebno napraviti korisnički račun. To se može napraviti pri prvom pokretanju programa Steam na računalu pritiskom na gumb *create new account* kako je prikazano na Slici 5.2.

U ovom trenutku se treba prijaviti u platformu korisničkim računom i instalirati besplatnu videoigru DOTA 2; nakon prijave na početnom ekranu programa Steam u gornjem lijevom kutu mora biti odabran *STORE* izbornik, a u gornjem dijelu programa postoji tražilica. U tražilicu je potrebno upisati ime igre, u ovom slučaju *DOTA 2* i pritisnuti na istu u padajućem izborniku kako je prikazano na Slici 5.3.

Program će se prebaciti na novi ekran otkuda je moguće instalirati igru pritiskom na zeleni gumb *Play Game*. Nakon instalacije, ostao je još jedan korak. Potrebno

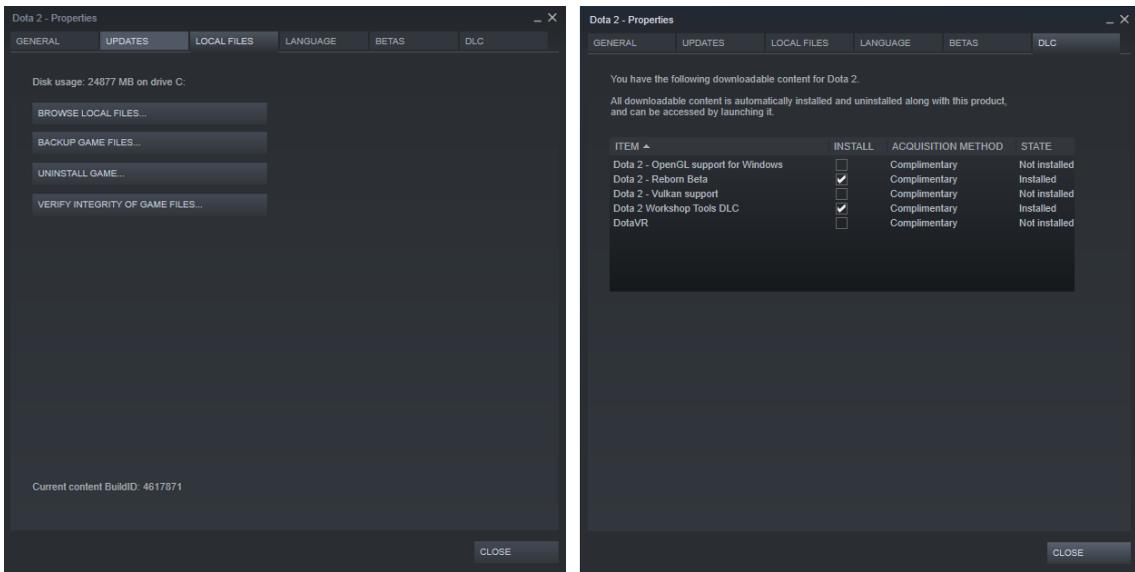


Slika 5.2: Pokretanje programa Steam



Slika 5.3: Pretraživanje igre na platformi Steam

je omogućiti sučelju Breezy pristup programskim alatima igre. To je moguće učiniti ravno iz izbornika *LIBRARY*, pritiskom desnog klika miša na igru DOTA 2. U padajućem izborniku potrebno je odabratи *Properties* nakon čega će se otvoriti iskočni prozor. U prozoru je potrebno odabrati karticu *DLC* na vrhu prozorčića. U listi dodataka za igru potrebno je staviti kvačicu na elemente *Dota 2 — Reborn Beta* i *Dota 2 Workshop tools DLC* te potom odabratи tipku *Close*. Slika 5.4 prikazuje navedeni postupak. Ovaj postupak će pokrenuti preuzimanje dodatnih datoteka za igru. Čim preuzimanje bude završeno igra je spremna za sinkroniziranje sa sučeljem Breezy!



Slika 5.4: Preuzimanje programskih alata za igru DOTA 2

5.2.3. Breezy Addon

Nakon instalacije igre je potrebno instalirati dodatak za igru putem kojeg će programski agent moći slati akcije. To je moguće napraviti putem platforme Steam ili manualno. Putem platforme Steam je potrebno odabratи *LIBRARY* izbornik na gornjem dijelu ekrana, odabratи igru DOTA 2 s izbornika na lijevoj strani te pritisnuti na zatamnjeni mali gumb *WORKSHOP* koji se nalazi negdje na sredini ekrana kako je prikazano na Slici 5.5. To bi trebalo otvoriti novi ekran u programu koji ima tražilicu. U tražilicu je potrebno upisati *Breezy Addon* te odabirom na isti stisnuti gumb *SUBSCRIBE*. Ako se dodatak skida manualno to je moguće učiniti putem sljedećeg dva linka:

<https://web.cs.dal.ca/~dota2/wp-content/uploads/2020/01/BreezyAddon.zip>

<https://gitlab.com/LivinBreezy/project-breezy/>

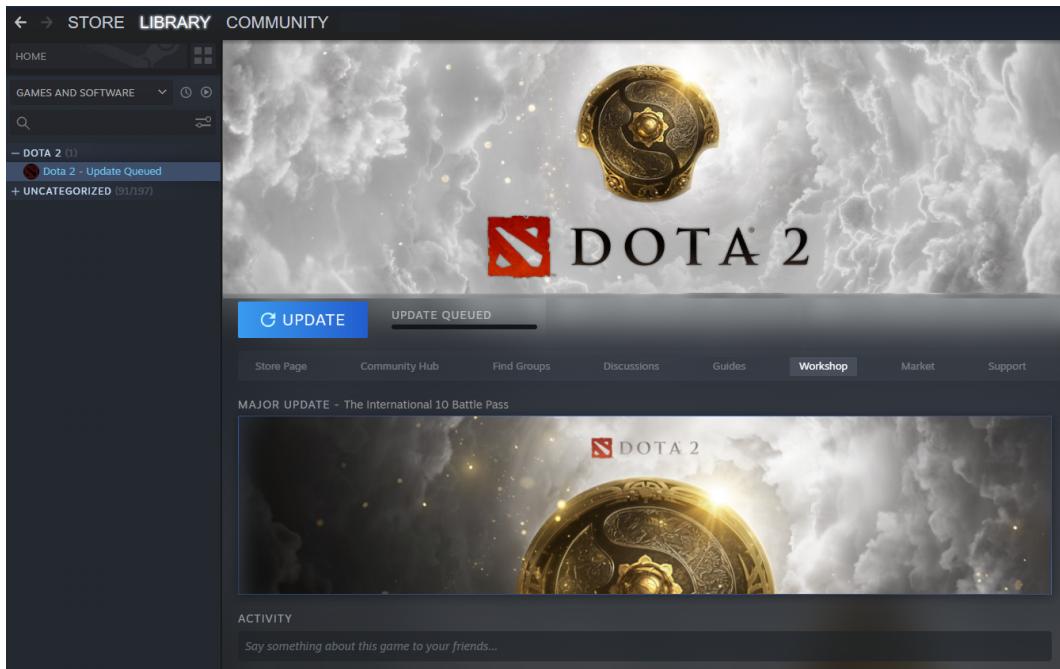
5.2.4. Breezy Server

Sada samo preostaje instalirati Breezy server sa:

<https://gitlab.com/LivinBreezy/project-breezy/>

Server je moguće spremiti bilo gdje na računalu. Nakon otpakiravanja zip datoteke servera potrebno je otvoriti *PowerShell* prozor na toj lokaciji i upisati sljedeću komandu kako bi se server preveo u izvršni kod:

```
.\gradlew.bat shadowjar
```



Slika 5.5: Steam WORKSHOP

Nakon instalacije server je moguće pokrenuti duplim klikom na datoteku *start.bat* ili komandom:

```
java -jar .\build\libs\server-1.0-SNAPSHOT-all.jar  
-conf conf.json
```

6. Prilagodba modela na igru DOTA 2

6.1. Komunikacija s Breezy sučeljem

Kao što je navedeno u poglavlju *Projekt Breezy* programski agent mora implementirati 3 metode za mogućnost komunikacije; *connect*, *update* i *relay*. U programskom jeziku python postoji biblioteka *flask* koja je vrlo pogodna za takve potrebe:

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/connect', methods=['GET'])
def connect():
    return {'response': 'the agent is ready'}


@app.route('/update', methods=['POST'])
def update():
    """ request.json -> post game information """
    ...
    return {'fitness': fitness(request.json)}


@app.route('/relay', methods=['POST'])
def relay():
    """ request.json = [int/float]*310 """
    ...
    return {'actionCode': action}
```

Agent uspostavlja konekciju s Breezy sučeljem navodeći svoje ime i broj igri koji želi odigrati.

```
if __name__ == '__main__':
    # tell breezy server to start the games
    url = f'http://{breezy_ip}:{breezy_port}/run/'
    data = json.dumps({'agent': 'medo', 'size': 100_000})
    response = requests.post(url=url, data=data)
    # start the agent server
    app.run(host=agent_ip, port=agent_port, debug=True)
```

6.2. Predobrada podataka

Nakon uspostavljanja veze sa serverom igra kreće. Pozivom metode *relay* sučelje Breezy prosljeđuje vektor od 310 značajki koje se nalaze u tablici 9.2 i čeka odgovor od agenta. Agent na temelju dobivenih informacija mora odlučiti koju će akciju odabrat. Poželjno je što više smanjiti prostor stanja kako bi agentu trebalo manje vremena za postizanje dobrih rezultata. Mnoge značajke iz prostora stanja su nebitne, invalidne ili je iz njih teško zaključiti koja bi akcija trebala biti odabrana. Primjer značajki iz kojih je teško izlučiti informaciju su npr. koordinate objekata. Svaki objekt u igri ima koordinatu *x* i *y*. Iz tih koordinata agent bi trebao naučiti udaljenosti do objekata i na temelju njih odabrat akciju. Kako bi smanjili vrijeme treniranja agentu će biti predane udaljenosti do objekata. Udaljenost mora biti predočena posebno po *x*, posebno po *y* koordinati jer inače agent ne bi mogao znati kako se približiti ili udaljiti od predmeta.

```
from collections import namedtuple

Creep = namedtuple('creep', 'hp x y xy')
Tower = namedtuple('tower', 'hp x y xy radius')
Stats = namedtuple('stats', 'last_hits denies time salve')
Hero = namedtuple('hero', 'x y xy lvl maxhp hp mana dmg
target abilities')
Ability = namedtuple('ability', 'lvl cost dmg cooldown
radius distance from_ to')
State = namedtuple('state', 'me opp good_tower bad_tower
stats creeps allies enemies')
```

Za razumljiviji kod i smanjenu mogućnost pogreške vektor od 310 značajki prebačen je u pripadajuće strukture podataka. Funkcija *get_entities* prima vektor od 310 značajki i vraća objekte serijalizirane u gore navedene strukture podataka. Varijable *me* i *opp* (*opponent*) predstavljaju *heroje* odnosno glavne likove u igri. Varijable *allies* i *enemies* sadrže informacije o ratnicima koje pripadaju timu agenta i neprijateljskog igrača.

```
state = get_entities(request.json)
me, opp, good_tower, bad_tower, stats, \
creeps, allies, enemies = state
```

Najbitnije informacije potrebne za donošenje odluka u igri su udaljenosti između objekata. 13 različitih relevantnih objekata postoje u igri čiju udaljenost od vlastitog heroja agent mora znati; udaljenost protivnika, prijateljskog tornja, protivnikovog tornja i svih ratnika u igri kojih je 10. Zato definiramo prostor stanja od 26 značajki koje predstavljaju 13 skalara udaljenosti u *x* smjeru i 13 skalara u *y* smjeru. Nadalje, bitno je znati udaljenosti prijateljskih ratnika od protivnikovog tornja, to je još 10 značajki. Osim udaljenosti agentu su potrebne informacije o životnim bodovima svih likova u igri i energiji heroja. Ukupno to čini vektor od 50 značajki koje opisuju jedno stanje u igri. Radi jednolikog uzimanja u obzir svih značajki iste moraju biti svedene na sličnu domenu vrijednosti. Iz tog razloga se sve veličine dijele pripadajućim koeficijentom kako bi se dobio broj na intervalu [0, 1].

```
current_observation = [
    *[abs(me.x - e.x) / d for e in
        [opp, good_tower, bad_tower] + creeps],
    *[abs(me.y - e.y) / d for e in
        [opp, good_tower, bad_tower] + creeps],
    *[abs(creep.x - bad_tower.x) / d for creep in allies],
    *[abs(creep.y - bad_tower.y) / d for creep in allies],
    *[creep.hp / me.dmg for creep in creeps],
    me.hp / me.maxhp,
    me.mana / me.abilities[0].cost,
    opp.hp / opp.maxhp,
    abs(opp.hp - me.hp) / opp.maxhp
]
```

Kako bi agent mogao odabirati akcije na temelju događaja koji su se nedavno dogodili, a ne samo onih koji se neposredno događaju prostor stanja sastoji se od opisa zadnjih 4 stanja kao što preporučuju van Hasselt et al. (2015).

```

from collections import deque

last4states = deque(maxlen=4)

...
last4states.append((current_observation, state))

observations, states = list(zip(*last4states))
observation_ = [i for j in observations for i in j]
r = reward(*states[-2:])
agent.memory.append(observation, action, r, observation_)

```

Struktura *deque* je dvostrani red ograničene veličine. Varijabla *last4states* sadrži zadnja 4 vektora prostora stanja. Svaki vektor prostora stanja ima 50 značajki, dakle ukupna veličina prostora stanja za odabir odluke je 200. Svakim korakom stanja se stavljuju u memoriju iskustva iz koje će agent kasnije učiti. Uz vektor prostora stanja u memoriju se stavlja nagrada odigravanja zadnje akcije. Način na koji vrednujemo odabir akcije u potpunosti oblikuje odabir budućih odluka. Igra ne pruža jednostavan način provjere dobrote stanja.

```

def reward(state1, state2):
    if state2.me.hp == -1: # I am dead
        return -1

    lh = state1.stats.last_hits - state2.stats.last_hits
    d = state1.stats.denies - state2.stats.denies
    dmg_done = state1.opp.hp - state2.opp.hp
    hp_lost = state1.me.hp - state2.me.hp
    time = state2.stats.time
    return ((lh + d) * 1000 + dmg_done - hp_lost + time) \
        / 1000

```

Iznos funkcije nagrade za odabranu akciju direktno utječe na strategiju koju će agent naučiti. Cilj agenta je upravo maksimizirati vrijednost funkcije nagrade odabirom najbolje akcije u svakom trenutku. Kada heroj agenta umre onda je agent kažnjen s negativnom nagradom -0.5 . U protivnom nagrada za odabir akcije ovisi o količini štete koju je agent taj potez nanio protivnicima i koliko je životnih bodova izgubio. Nadalje nagrada se povećava za količinu vremena kojeg je heroj agenta proveo u igri.

6.3. Modifikacija agenta

Promjenom igre potrebno je podesiti hiperparametre o kojima je više rečeno u poglavljima *Tehnike potpornog učenja* i *Izrada modela potpornog učenja*.

```
epsilon = 1
epsilon_min = 0.001
games = 500
game = 0
batch_size = 64
iterations = games * batch_size
epsilon_decay = (epsilon - epsilon_min) / iterations
replay_memory_size = 500_000
step = 0
sync_target_steps = 10_000
discount_rate = 0.99

memory = deque(maxlen=replay_memory_size)
```

Model programskog agenta strukturiran je u klasu *Agent*, no zbog smanjenja opsežnosti koda u radu ugniježđenost koda je smanjena. Arhitektura mreže se sastoji od jednog ulaznog, izlaznog i skrivenog sloja. Veličina ulaznog sloja je 200 značajki, 50 značajki po trenutku, 4 trenutka unazad. Veličina izlaznog sloja je 30, za svaku akciju koju agent može odigrati, a veličina skrivenog sloja je 128.

```
def create_network(filename=''):
    network = Sequential()
    network.add(Dense(128, input_dim=200, activation='tanh'))
    network.add(Dense(30, activation='linear'))
    if os.path.isfile(filename):
        network.load_weights(filename)

    network.compile(loss='mse', optimizer=Adam(lr=0.00025))
    return network
```

```
q_network = create_network()
t_network = create_network()
```

Svakih $sync_target_steps$ koraka se težine skrivenih slojeva $t_network$ i $q_network$ sinkroniziraju. Slučajne akcije se rade s vjerojatnošću $epsilon$ koji se linearno smanjuje do $epsilon_min$. Metoda $predict_$ prima vektor prostora stanja od 200 značajki i popis svih legalnih akcija koje agent može odigrati u trenutnom stanju. Ako je nasumični broj manji ili jednak $epsilon$ onda agent predviđa nasumičnu legalnu akciju. Inače agent odabire akciju koristeći $q_network$ mrežu tako da izabire legalnu akciju najveće Q-vrijednosti.

```
def predict_(state, legal_actions):
    if step % sync_target_steps == 0:
        t_network.set_weights(q_network.get_weights())

    step += 1
    if epsilon > epsilon_min:
        epsilon -= epsilon_decay

    if np.random.uniform() <= epsilon:
        return random.choice(legal_actions)

    q_values = predict(q_network, state)
    actions = sorted(range(len(q_values)),
                     key=lambda i: q_values[i], reverse=True)

    for action in actions:
        if action in legal_actions:
            return action
```

U svakoj igri agent stigne odigrati otprilike 130 akcija. Nakon svake igre agent podešava težine mreže. Svakih 50 igara se težine mreža spremaju u datoteku na disku.

```
def learn():
    if game % 50 == 0:
        q_network.save_weights(f'q_network{game}.h5')
    game += 1
    if len(memory) < batch_size:
        return
```

```

batch = random.sample(memory, batch_size)
for state, action, reward, state_ in batch:
    future_reward = discount_rate * \
        np.amax(predict(t_network, state_))
    q_value = reward * (1 + future_reward)
    target = predict(q_network, state)
    target[action] = q_value
    q_network.fit(to_array(state), to_array(target),
                  epochs=1, verbose=0)

```

6.4. Zapisivanje rezultata

Nakon svake igre potrebno je izračunati koliko je agent dobro odigrao igru. Između dvije uzastopne igre Breezy server poziva agentovu metodu *update* te agentu šalje rezultat igre. Rezultat igre je *json* datoteka sljedećeg oblika:

```

request.json = {
    'id': 'db56098d-824a-46a5-a138-ccf79a3059f3',
    'size': 1,
    'startTime': 'Thu Apr 02 09:03:35 CEST 2020',
    'endTime': 'Thu Apr 02 09:04:35 CEST 2020',
    'duration': 59552,
    'status': 'DONE', # or 'WAITING'
    'progress': 1,
    'gameIds': ['09:04:037942'],
    'winner': 'Dire',
    'direKills': 2,
    'radiantKills': 0,
    'deaths': 2
}

```

Na temelju tih podataka funkcija *fitness* računa koliko je agent dobro odigrao igru. U obzir se uzimaju i podaci koje je server proslijedio funkciji *update* i podaci o zadnjem stanju u kojem je agent bio prije završetka igre. Najbitniji podaci su je li agent pobijedio, je li uspio ubiti protivnikovog heroja, koliko dugo je igra trajala te koliko je ratnika uspio ubiti.

```

def fitness(result):
    duration = last4states[-1][1].stats.time
    win = result['winner'] == 'Radiant'
    kills = result['radiantKills']
    deaths = result['deaths']
    last_hits = last4states[-1][1].stats.last_hits
    denies = last4states[-1][1].stats.denies
    return 2000 * win + 500 * (2 + kills - deaths) + \
           duration / 60 + (denies + last_hits) * 100

```

Nakon izračunavanja dobrote odigrane igre podaci se zapisuju u datoteku te prosljeđuju natrag Breezy serveru.

```

response = {'fitness': fitness(request.json)}

print(agent.step)

with open('results.txt', 'a') as f:
    f.write(f"{{agent.game}, {response['fitness']} }\n")

return response

```

7. Rezultati

U 10 dana učenja model nije postigao značajno bolje rezultate (manje od 10%) od agenta koji radi nasumične akcije. Za to postoji mnogo razloga.

Najveći problem jest definicija funkcije nagrade. Agent je bio nagrađivan za akcije koje su pozitivno utjecale na razvoj igre samo ako je to bilo vidljivo odmah u sljedećem stanju. Količina takvih akcija je iznimno mala, otprilike 1.5%. Jedan način smanjenja problema jest nagrađivanje agenta prosječnom dobivenom nagradom u zadnjih n akcija ili jednostavno produženje trajanja vremena treniranja.

Drugi veliki problem jest početna razlika u iskustvu agenta i protivnika. Protivnik pobjeđuje nešto manje od 50% aktivnih ljudskih igrača. Da bi agent što brže napredovao potrebno je imati mogućnost progresivnog otežavanja igre.

Zbog aspekta odabira akcija u stvarnom vremenu programski agent je u značajno nepovoljnijem položaju od protivnika. Protivnik ima mogućnost odabira otprilike osam akcija u jednoj sekundi igre, s obzirom na to da je igra ubrzana za klijenta; agent ima mogućnost odabira osam akcija u jednoj sekundi stvarnog vremena, odnosno jedne akcije u jednoj sekundi igre. To znači da protivnik može napraviti 8 akcija za svaku akciju koju poduzme programski agent. S takvim nedostatkom upitno je bi li ljudski ekspert uopće mogao pobijediti. Nadalje, upravo zbog odabira akcija u stvarnom vremenu programski agent uvijek odabire akciju na temelju zakašnjelih informacija, u igri kao DOTA 2 stanje igre se može drastično promijeniti u milisekundama, a kamoli u sekundama.

Odabir tehnike potpornog učenja nije najbolji. Zbog učenja akcija nasumičnim redoslijedom radi ubrzanja procesa učenja agent ne može naučiti postizati dugoročne ciljeve jer je korelacija uzastopnih učenih stanja ništavna.

8. Zaključak

Iz razloga navedenih u prošlom poglavlju programski agent nije postigao zavidne rezultate. Usprkos tome, projekt nije bio uzaludan. Istražene su najpopularnije tehnike potpornog učenja i njihov razvoj. Objasnjen je postupak usporedbe složenosti okruženja za primjenu tehnika umjetne inteligencije. Uspješno je implementiran model duplog dubokog Q-učenja što se vidi iz rezultata postignutih na igri balansiranja štapa. Pokazano je kako pripremiti sirove podatke iz igre za model te je cijeli postupak popraćen kodom.

Razvoj programskih agenata nadljudskih sposobnosti za složene videoigre današnjice je još uvijek izvan dosega pojedinaca. Iako je agentima vodećih pionira potpornog učenja potrebno mnogo više domenskog znanja i vremena treniranja nego što je bilo predviđeno ovim radom algoritmi i metode koje smo koristili su komparabilni. Za daljnji rad najbolje bi bilo izabrati igru koja direktno podržava izradu programskih agenata i nativno nudi sučelje za komunikaciju s igrom. Osim ideja za poboljšanje navedenih u prošlom poglavlju navodimo još neke ideje u nastavku koje bi znatno poboljšale kvalitetu treniranja programskog agenta; mogućnost paralelizacije na više računala, preuzimanje težina najboljeg agenta prema prosjeku rezultata zadnjih n partija te duže vrijeme treniranja.

9. Addendum

U ovom poglavlju se nalaze originalne tablice.

Tablica 9.1: Prostor akcija

N	Akcija
0	Do Nothing
1	Move North
2	Move Northeast
3	Move East
4	Move Southeast
5	Move South
6	Move Southwest
7	Move West
8	Move Northwest
9	Get Top Bounty Rune (River, Beside Roshan)
10	Get Bottom Bounty Rune (River, Near Dire Secret Shop)
11	Get Top Powerup Rune
12	Get Bottom Powerup Rune
13	Attack Enemy Hero
14	Attack Enemy Tower
15	Attack Creep 0 (Nearest Enemy)
16	Attack Creep 1
17	Attack Creep 2
18	Attack Creep 3
19	Attack Creep 4 (Farthest Enemy)
20	Attack Creep 5 (Nearest Friendly)
21	Attack Creep 6
22	Attack Creep 7
23	Attack Creep 8
24	Attack Creep 9 (Farthest Friendly)
25	Cast Shadowraze 1 (Short)
26	Cast Shadowraze 2 (Medium)
27	Cast Shadowraze 3 (Long)
28	Cast Requiem of Souls (Ultimate)
29	Use Healing Salve

Tablica 9.2: Prostor stanja

N	Opis	N	Opis
0	team	24	last hits
1	level	25	denies
2	health	26	location 1
3	max health	27	location 2
4	health regen	28	facing
5	mana	29	vision range
6	max mana	30	opp team
7	mana regen	31	opp level
8	base move speed	32	opp health
9	current move speed	33	opp max health
10	base damage	34	opp health regen
11	damage variance	35	opp mana
12	attack damage	36	opp max mana
13	attack range buffer	37	opp mana regen
14	attack speed	38	opp base move speed
15	seconds per attack	39	opp curr move speed
16	attack animation point	40	opp base damage
17	last attack time	41	opp damage variance
18	attack target	42	opp attack damage
19	strength	43	opp attack range buffer
20	agility	44	opp attack speed
21	intellect	45	opp seconds per attack
22	gold	46	opp attack animation point
23	net worth	47	opp last attack time
		48	opp attack target
		49	opp strength
		50	opp agility
		51	opp intellect

N	Opis	N	Opis
52	opp location 1	80	power 2 location 2
53	opp location 2	81	power 1 type
54	opp facing	82	power 1 status
55	opp vision range	83	power 2 type
56	dota time	84	power 2 status
57	good tower team	85	good creep 1 team
58	good tower health	86	good creep 1 health
59	good tower max health	87	good creep 1 max health
60	good tower attack damage	88	good creep 1 health regen
61	good tower attack range buffer	89	good creep 1 base move speed
62	good tower attack speed	90	good creep 1 current move speed
63	bad tower team	91	good creep 1 base damage
64	bad tower health	92	good creep 1 damage variance
65	bad tower max health	93	good creep 1 attack damage
66	bad tower attack damage	94	good creep 1 attack range buffer
67	bad tower attack range buffer	95	good creep 1 attack speed
68	bad tower attack speed	96	good creep 1 seconds per attack
69	bounty 1 location 1	97	good creep 1 location 1
70	bounty 1 location 2	98	good creep 1 location 2
71	bounty 2 location 1	99	good creep 2 team
72	bounty 2 location 2	100	good creep 2 health
73	bounty 3 location 1	101	good creep 2 max health
74	bounty 3 location 2	102	good creep 2 health regen
75	bounty 4 location 1	103	good creep 2 base move speed
76	bounty 4 location 2	104	good creep 2 current move speed
77	power 1 location 1	105	good creep 2 base damage
78	power 1 location 2	106	good creep 2 damage variance
79	power 2 location 1	107	good creep 2 attack damage

N	Opis	N	Opis
108	good creep 2 attack range buffer	136	good creep 4 attack range buffer
109	good creep 2 attack speed	137	good creep 4 attack speed
110	good creep 2 seconds per attack	138	good creep 4 seconds per attack
111	good creep 2 location 1	139	good creep 4 location 1
112	good creep 2 location 2	140	good creep 4 location 2
113	good creep 3 team	141	good creep 5 team
114	good creep 3 health	142	good creep 5 health
115	good creep 3 max health	143	good creep 5 max health
116	good creep 3 health regen	144	good creep 5 health regen
117	good creep 3 base move speed	145	good creep 5 base move speed
118	good creep 3 current move speed	146	good creep 5 current move speed
119	good creep 3 base damage	147	good creep 5 base damage
120	good creep 3 damage variance	148	good creep 5 damage variance
121	good creep 3 attack damage	149	good creep 5 attack damage
122	good creep 3 attack range buffer	150	good creep 5 attack range buffer
123	good creep 3 attack speed	151	good creep 5 attack speed
124	good creep 3 seconds per attack	152	good creep 5 seconds per attack
125	good creep 3 location 1	153	good creep 5 location 1
126	good creep 3 location 2	154	good creep 5 location 2
127	good creep 4 team	155	bad creep 1 team
128	good creep 4 health	156	bad creep 1 health
129	good creep 4 max health	157	bad creep 1 max health
130	good creep 4 health regen	158	bad creep 1 health regen
131	good creep 4 base move speed	159	bad creep 1 base move speed
132	good creep 4 current move speed	160	bad creep 1 current move speed
133	good creep 4 base damage	161	bad creep 1 base damage
134	good creep 4 damage variance	162	bad creep 1 damage variance
135	good creep 4 attack damage	163	bad creep 1 attack damage

N	Opis	N	Opis
164	bad creep 1 attack range buffer	192	bad creep 3 attack range buffer
165	bad creep 1 attack speed	193	bad creep 3 attack speed
166	bad creep 1 seconds per attack	194	bad creep 3 seconds per attack
167	bad creep 1 location 1	195	bad creep 3 location 1
168	bad creep 1 location 2	196	bad creep 3 location 2
169	bad creep 2 team	197	bad creep 4 team
170	bad creep 2 health	198	bad creep 4 health
171	bad creep 2 max health	199	bad creep 4 max health
172	bad creep 2 health regen	200	bad creep 4 health regen
173	bad creep 2 base move speed	201	bad creep 4 base move speed
174	bad creep 2 current move speed	202	bad creep 4 current move speed
175	bad creep 2 base damage	203	bad creep 4 base damage
176	bad creep 2 damage variance	204	bad creep 4 damage variance
177	bad creep 2 attack damage	205	bad creep 4 attack damage
178	bad creep 2 attack range buffer	206	bad creep 4 attack range buffer
179	bad creep 2 attack speed	207	bad creep 4 attack speed
180	bad creep 2 seconds per attack	208	bad creep 4 seconds per attack
181	bad creep 2 location 1	209	bad creep 4 location 1
182	bad creep 2 location 2	210	bad creep 4 location 2
183	bad creep 3 team	211	bad creep 5 team
184	bad creep 3 health	212	bad creep 5 health
185	bad creep 3 max health	213	bad creep 5 max health
186	bad creep 3 health regen	214	bad creep 5 health regen
187	bad creep 3 base move speed	215	bad creep 5 base move speed
188	bad creep 3 current move speed	216	bad creep 5 current move speed
189	bad creep 3 base damage	217	bad creep 5 base damage
190	bad creep 3 damage variance	218	bad creep 5 damage variance
191	bad creep 3 attack damage	219	bad creep 5 attack damage

N	Opis	N	Opis
220	bad creep 5 attack range buffer	246	ability 4 level
221	bad creep 5 attack speed	247	ability 4 mana cost
222	bad creep 5 seconds per attack	248	ability 4 ability damage
223	bad creep 5 location 1	249	ability 4 cast range
224	bad creep 5 location 2	250	ability 4 cooldown time remaining
225	ability 1 level	251	ability 4 target type
226	ability 1 mana cost	252	ability 4 behavior
227	ability 1 ability damage	253	ability 5 level
228	ability 1 cast range	254	ability 5 mana cost
229	ability 1 cooldown time remaining	255	ability 5 ability damage
230	ability 1 target type	256	ability 5 cast range
231	ability 1 behavior	257	ability 5 cooldown time remaining
232	ability 2 level	258	ability 5 target type
233	ability 2 mana cost	259	ability 5 behavior
234	ability 2 ability damage	260	ability 6 level
235	ability 2 cast range	261	ability 6 mana cost
236	ability 2 cooldown time remaining	262	ability 6 ability damage
237	ability 2 target type	263	ability 6 cast range
238	ability 2 behavior	264	ability 6 cooldown time remaining
239	ability 3 level	265	ability 6 target type
240	ability 3 mana cost	266	ability 6 behavior
241	ability 3 ability damage	267	opp ability 1 level
242	ability 3 cast range	268	opp ability 1 mana cost
243	ability 3 cooldown time remaining	269	opp ability 1 ability damage
244	ability 3 target type	270	opp ability 1 cast range
245	ability 3 behavior	271	opp ability 1 cooldown time remaining

N	Opis	N	Opis
272	opp ability 1 target type	298	opp ability 5 cast range
273	opp ability 1 behavior	299	opp ability 5 cooldown time remaining
274	opp ability 2 level	300	opp ability 5 target type
275	opp ability 2 mana cost	301	opp ability 5 behavior
276	opp ability 2 ability damage	302	opp ability 6 level
277	opp ability 2 cast range	303	opp ability 6 mana cost
278	opp ability 2 cooldown time remaining	304	opp ability 6 ability damage
279	opp ability 2 target type	305	opp ability 6 cast range
280	opp ability 2 behavior	306	opp ability 6 cooldown time remaining
281	opp ability 3 level	307	opp ability 6 target type
282	opp ability 3 mana cost	308	opp ability 6 behavior
283	opp ability 3 ability damage	309	item flask
284	opp ability 3 cast range		
285	opp ability 3 cooldown time remaining		
286	opp ability 3 target type		
287	opp ability 3 behavior		
288	opp ability 4 level		
289	opp ability 4 mana cost		
290	opp ability 4 ability damage		
291	opp ability 4 cast range		
292	opp ability 4 cooldown time remaining		
293	opp ability 4 target type		
294	opp ability 4 behavior		
295	opp ability 5 level		
296	opp ability 5 mana cost		
297	opp ability 5 ability damage		

LITERATURA

ADL. q table, 2018. URL <https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/>. [Online, accessed Jun 14, 2020].

Jason Breininger. Space invaders (atari vcs/2600), 2018. URL <http://www.oldschoolgamermagazine.com/space-invaders-atari-vcs-2600/>. [Online, accessed Jun 14, 2020].

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, i Wojciech Zaremba. Openai gym, 2016.

Sepp Hochreiter i Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

howtofixx.com. chess board, 2020. URL <https://www.howtofixx.com/chess-game-printable-template/>. [Online, accessed Jun 12, 2020].

Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, i Joshua B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *CoRR*, abs/1604.06057, 2016. URL <http://arxiv.org/abs/1604.06057>.

L. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach Learn*, 8, 293-321, 1992. URL <https://doi.org/10.1007/BF00992699>.

mc.ai. Dqn, 2018. URL <https://mc.ai/deep-q-learning-intuition/>. [Online, accessed Jun 14, 2020].

V. Mnih, K. Kavukcuoglu, i D. Silver. Human-level control through deep reinforcement learning. *Nature*, 518, 529-533, 2015. URL <https://doi.org/10.1038/nature14236>.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, i Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL <http://arxiv.org/abs/1312.5602>.

OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, i Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

Jason Scott. Atari 2600: Montezuma's revenge - featuring panama joe, 2013. URL https://archive.org/details/atari_2600_montezumas_revvenge_-featuring_panama_joe_1984_parker_brothers_robert_. [Online, accessed Jun 14, 2020].

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharrshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, i Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362 (6419):1140–1144, 2018. ISSN 0036-8075. doi: 10.1126/science.aar6404. URL <https://science.sciencemag.org/content/362/6419/1140>.

Martin Tutek. Potporno učenje, 2018. URL https://www.fer.unizg.hr/_download/repository/UI-11-PotpornoUcenje.pdf. [Online, accessed Jun 14, 2020].

Hado van Hasselt, Arthur Guez, i David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL <http://arxiv.org/abs/1509.06461>.

Razvoj programskih agenata za složene videoigre

Sažetak

Kako umjetna inteligencija postaje sve popularnija nove metode potpornog učenja se neprestano razvijaju. Istraživanje je fokusirano na povećanje apstrakcije zadržavajući ili poboljšavajući postignut rezultat i/ili vrijeme treniranja prošlih agenata. Nada-lje, biblioteke strojnog učenja programskih jezika omogućuju izradu agenata umjetne inteligencije s lakoćom. Mogu li pojedinci razviti vlastite modele potpornog učenja bez mnogo domenskog znanja i trenirati agente na tipičnom igračem računalu u razumnom vremenu te nadigrati ugrađene agente igara koji koriste metode pretraživanja prostora stanja i stabla odluka? U ovom radu istražujemo tehnike potpornog učenja, složenost igara te je dan opis izrade agenta potpornog učenja za pojednostavljenu verziju igre DOTA 2 koristeći programsko sučelje *Project Breezy* za interakciju s igrom.

Ključne riječi: DOTA, DOTA2, Breezy, SF, Shadow Fiend, 1v1, potporno učenje, tensorflow

Development of software robots for complex video games

Abstract

As artificial intelligence is becoming increasingly more popular new methods and techniques of applying reinforcement learning are constantly emerging. Research focuses on abstraction improvement while retaining or even outperforming scores and/or learning speed in previously made agents. Furthermore, machine learning libraries for programming languages make it possible to build AI agents with ease. Can individuals build their own RL models without much domain knowledge and train agents on typical gaming hardware in a reasonable time frame to outperform pathfinding or decision tree agents found in most video games? In this paper we explore RL techniques, game complexity and build a RL agent for a simplified version of a popular video game DOTA 2 using *Project Breezy* to interact with the game client.

Keywords: DOTA, DOTA2, Breezy, SF, Shadow Fiend, 1v1, reinforcement learning, tensorflow