Duboko poticano učenje i njegova primjena u implementaciji umjetnog igrača strateške videoigre

Hinić, Leon Hrid

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Leon Hrid Hinić

DUBOKO POTICANO UČENJE I NJEGOVA PRIMJENA U IMPLEMENTACIJI UMJETNOG IGRAČA STRATEŠKE VIDEOIGRE

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Leon Hrid Hinić

Matični broj: 44922/16-R

Studij: Informacijski sustavi

DUBOKO POTICANO UČENJE I NJEGOVA PRIMJENA U IMPLEMENTACIJI UMJETNOG IGRAČA STRATEŠKE VIDEOIGRE

DIPLOMSKI RAD

Mentor :

Dr. sc. Bogdan Okreša Đurić

Varaždin, siječanj 2024.

Leon Hrid Hinić

Izjava o izvornosti

Izjavljujem da je moj 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

Osnova rada je koncept dubokog poticanog učenja, pristupa implementacije strojnog učenja koji kombinira elemente poticanog i dubokog strojnog učenja, te njegovu primjenu u domeni video igara. Praktični dio rada obuhvaća implementiranje umjetnog inteligentnog igrača strateške videoigre koji koristi samostalno razvijen model dubokog poticanog učenja.

Ključne riječi: umjetna inteligencija; duboko učenje; reinforcement learning; poticano učenje; videoigre; agent;

Sadržaj

1. Uvod

Rad započinje pregledom povijesti i teorijskih osnova umjetne inteligencije i strojnog učenja. Stavlja se fokus na metode i pristupe strojnog učenja, te se dublje istražuje pristup koji se koristi za agente koji igraju video igre, a to je poticano učenje. Nakon poticanog učenja fokus se stavlja na duboko učenje i implementacije dubokog učenja kao što su duboke neuronske mreže, te tako dolazimo do dubokog poticanog učenja.

Duboko poticano učenje kombinira arhitekturu dubokog učenja i principe poticanog učenja kako bi omogućilo agentima učenje isključivo iz interakcije s okolinom kako bi donosili odluke i poduzimali radnje koje utječu na okolinu. Kombinacija ova dva pristupa omogućava agentima da nauče kako doći do zadanog cilja, bez ikakvih pred-definiranih uputa. U ovom kombiniranom pristupu, agent uči politiku ili strategiju primanjem povratnih informacija u obliku nagrade (ili kazne) na temelju svojih postupaka, odnosno interakcije s okolinom. Duboke neuronske mreže koriste se za predstavljanje agentove politike, dok informacije koje ulaze u prvi sloj neuronske mreže dolaze iz okoline.

Prikazan je i objašnjen sav kod implementacije dva agenta koji koriste duboko poticano učenje, konkretnije duboko Q-učenje da bi igrali video igru StarCraft II. Jedan od agenata igra mini igru posebno stvorenu za eksperimentiranje s agentima dok drugi agent igra istu vrstu igre kao i ljudski igrači, samo na nešto smanjenoj mapi.

2. Metode i tehnike rada

Glavna istraživačka metoda u ovom radu bila je izrada agenata koji igraju video igru *StarCraft 2* u programskom jeziku Python.

2.1. Python

Prema službenoj stranici, Python je jasan i moćan objektno orijentirani programski jezik, usporediv s jezicima kao što su Perl, Ruby, Scheme ili Java [1].

Dolazi s velikom standardnom bibliotekom koja podržava mnoge uobičajene programerske zadatke kao što je povezivanje s web poslužiteljima, pretraživanje teksta s regularnim izrazima, čitanje i mijenjanje datoteka. Lako se proširuje dodavanjem novih modula implementiranih u prevedenom jeziku kao što je C ili C++, što znači da korištenje funkcionalnosti iz dodataka pruža znatno bolje performanse u odnosu na implementiranje istih u vlastitim projektima u Pythonu.

Python je besplatan za korištenje i uključivanje u aplikaciju. Python se također može slobodno mijenjati i dalje distribuirati jer iako je jezik zaštićen autorskim pravima, dostupan je pod licencom otvorenog koda. [1].

Uz uobičajene standardne tipove podataka i podatkovnih struktura, Python podržava objektno orijentirano programiranje s klasama i višestrukim nasljeđivanjem. Kod se također može grupirati u module i pakete. Jezik podržava podizanje i hvatanje iznimaka, bez ikakvih dodataka. Tipovi podataka su strogo i dinamički tipizirani. Miješanje nekompatibilnih tipova (npr. pokušaj dodavanja niza i broja) uzrokuje pokretanje iznimke, tako da se pogreške prije uočavaju. Python sadrži napredne značajke programiranja kao što su generatori i razumijevanje popisa (eng. *list comprehension*). Pythonovo automatsko upravljanje memorijom oslobađa vas potrebe za ručnom dodjelom i oslobađanjem memorije u vašem kodu. [1].

2.2. TensorFlow

Glavni Python dodatak korišten za izradu modela pomoću kojeg agenti uče je Tensorflow. TensorFlow olakšava početnicima i stručnjacima stvaranje modela strojnog učenja za stolna računala, mobilne uređaje, web i oblak [2]. Iako Tensorflow nudi već gotove agente za duboko poticano učenje, te funkcionalnosti dodatka nisu korištene u ovom radu, korištene su samo osnovne funkcionalnosti za izgradnju dubokih neuralnih mreža.

TensorFlow je sustav strojnog učenja koji radi u velikim razmjerima i u heterogenim okruženjima. TensorFlow koristi grafove protoka podataka za predstavljanje izračuna, zajedničkog stanja i operacija koje mijenjaju to stanje. Preslikava čvorove grafa toka podataka na mnogo strojeva u grupe i unutar stroja na više računalnih uređaja, uključujući višejezgrene CPU-ove, GPU-ove opće namjene i posebno dizajnirane ASIC-ove poznate kao Tensor Processing Units (TPU). Ova arhitektura daje fleksibilnost razvojnom programeru aplikacija: dok je u prethod-

nim dizajnima "parametarskog poslužitelja" upravljanje dijeljenim stanjem ugrađeno u sustav. TensorFlow podržava različite aplikacije, s fokusom na obuku i zaključivanje na dubokim neuronskim mrežama [3].

Uz Tensorflow također je korišten dodatak koji je i njegova ovisnost (eng. *dependency*) pod imenom *Numpy*. Uglavnom je korišten za funkcije pseudoslučajnog odabira i manipulacije listama u Pythonu.

2.3. PySC2

PySC2 [4] je DeepMindova Python komponenta StarCraft II okruženja za učenje (SC2LE). Izlaže API za strojno učenje StarCraft II tvrtke Blizzard Entertainment kao Python RL okruženje.

3. Umjetna inteligencija i strojno učenje

Prije razumijevanja kompleksnog pojma kao što je duboko poticano učenje, potrebno je prvo razjasniti pojmove kao što su umjetna inteligencija, inteligentni agent i strojno učenje, kakve sve vrste strojnog učenja postoje, razliku između klasičnog i dubokog strojnog učenja, te u konačnici nešto više o poticanom i dubokom poticanom učenju.

3.1. Umjetna inteligencija

Russell i Norvig [5] navode da povijesno gledano, istraživači su slijedili nekoliko različitih verzija umjetne inteligencije. Neki su definirali inteligenciju u smislu vjernosti ljudskoj izvedbi, dok drugi preferiraju apstraktnu, formalnu definiciju inteligencije koja se naziva racionalnost - slobodno rečeno, raditi "pravu stvar", promatranu kroz kombinaciju matematike i inženjerstva, te se povezuje sa statistikom, teorijom upravljanja i ekonomijom.

Umjetna inteligencija je grana znanosti koja se bavi proučavanjem stvaranja inteligentnih agenata. [6, str. 1]. U ovom kontekstu, pojam agent obuhvaća bilo koji entitet koji je sposoban biti u interakciji s okolinom. Inteligentne agente karakterizira sposobnost donošenja odluka koje se smatraju inteligentnima, što uključuje izvršavanje koraka usmjerenih prema postizanju vlastitih ciljeva, uzimajući u obzir okolinu, učenje iz iskustava, te prilagodbu promjenama u okolini i ciljevima. [6, str. 1]

Glavni znanstveni cilj umjetne inteligencije jest dokučiti principe koji leže u osnovi inteligentnog ponašanja, kako u prirodnim tako i u umjetnim sustavima [6, str. 1]. U počecima razvoja ove discipline, to je obuhvaćalo razmatranje aspekata poput zaključivanja, reprezentacije znanja, planiranja i učenja. Međutim, u suvremenim istraživanjima, fokus se sve više usmjerava prema rješavanju specifičnih zadataka, kao što su obrada prirodnog jezika, percepcija okoline te sposobnost kretanja i manipulacije objektima [6, str. 1].

Većina inteligentnih agenata je dizajnirana za rješavanje konkretnih problema unutar pojedinih strogo definiranih domena i nisu sposobni za rješavanje probleme izvan njih. Ova ograničenost nije slučajna, nego je rezultat trenutnog stanja tehnologije i dostupnih metoda. Ovakva vrsta umjetne inteligencije često se naziva "slabom umjetnom inteligencijom" (eng. *Weak AI*) [7]. Umjetna inteligencija još nije razvijena do te razine da bi jedan agent mogao primijeniti svoje sposobnosti na više različitih domena ili čak na jednu veoma široku domenu. Od samih početaka razvoja umjetne inteligencije postoji težnja prema razvoju takve hipotetske umjetne inteligencije koja bi bila sposobna za različite zadatke, te se takva umjetna inteligencija naziva "jaka umjetna inteligencija" (eng. *Strong AI*) ili "Opća umjetna inteligencija" (eng. *Artificial general intelligence*, kraće AGI).

AGI se definira kao opću umjetnu inteligenciju, teorijski računalni program koji bi mogao izvršavati intelektualne zadatke na razini čovjeka ili čak nadmašiti ljudske sposobnosti [7]. Ovo se odnosi na sve zadatke koje čovjek može rješavati, jer su sustavi koji se temelje na umjetnoj inteligenciji već nadmašili čovjekove sposobnosti u određenim domenama. Jedna od tih domena su video igre, što je detaljnije obrađeno u kasnijim dijelovima ovog rada [8].

Neapolitan i Jiang [9] kao prvi program umjetne inteligencije navode "Logičkog Teoretičara" (eng. *Logic Theorist*) koji su razvili Allen Newell i Herbert Simon tijekom razdoblja 1955. - 1956. Ovaj program je bio osmišljen kako bi simulirao sposobnosti ljudskog razmišljanja pri rješavanju problema. Logički teoretičar je postigao značajan uspjeh u dokazivanju logičkih teorema, često pronalazeći znatno kraće dokaze za određene tvrdnje.

3.2. Strojno učenje

Pojam strojnog učenja povezuje se s umjetnom inteligencijom i često se pogrešno koriste kao međusobno zamjenjivi izrazi. Između ta dva pojma postoji hijerarhijski odnos, gdje je umjetna inteligencija nadređeni i širi pojam u usporedbi sa strojnim učenjem. Strojno učenje predstavlja samo jednu komponentu ili dio umjetne inteligencije [10].

Agent uči ako poboljša svoju izvedbu nakon promatranja svijeta. Učenje može varirati od trivijalnog, kao što je zapisivanje popisa za kupovinu, do dubokog, kao kada je Albert Einstein izveo novu teoriju svemira. Kada je agent računalo, to nazivamo strojnim učenjem: računalo promatra neke podatke, gradi model na temelju podataka strojnog učenja i koristi model kao hipotezu o svijetu i dio softvera koji može riješiti probleme [5, str. 651]

Murphy [11, str. 1] daje definiciju strojnog učenja kao skup metoda koje su u stanju automatski prepoznati obrasce u podacima i potom koristiti te identificirane obrasce za predviđanje budućih podataka ili za donošenje različitih vrsta odluka u uvjetima nesigurnosti. To uključuje planiranje kako prikupiti dodatne podatke radi poboljšanja učenja.

3.2.1. Pristupi strojnom učenju

Postoje različite podjele domene koja se naziva strojno učenje. Svaki od tih pristupa se dalje dijeli na manje, ali znatno različite pod-domene, te se te domene mogu kombinirati jedne s drugima, i s različitim arhitekturama modela učenja kao što je duboko učenje, o kojem će biti više riječi u kasnijim poglavljima.

3.2.1.1. Nadzirano učenje

U pristupu učenju pod nadzorom ili prediktivnom učenju, glavni cilj je naučiti model preslikavanja između ulaznih podataka x i odgovarajućih izlaznih vrijednosti y, koristeći označeni skup ulazno-izlaznih parova $D = \{(xi, yi)\}_{i=1}^{N}$ [11, str. 2]. Ulazni podaci x_i nazivaju se značajkama, atributima ili kovarijablima, dok se izlazne vrijednosti y_i nazivaju varijablama odgovora (eng. *response variable*).

Zamislite da želimo izgraditi sustav koji može klasificirati slike kao one koje sadrže, recimo, kuću, automobil, osobu ili kućnog ljubimca. Prvo prikupljamo veliki skup podataka slika kuća, automobila, ljudi i kućnih ljubimaca, svaka označena svojom kategorijom. Tijekom obuke stroju se prikazuje slika te stroj proizvodi izlaz u obliku vektora rezultata, po jedan za svaku kategoriju. Želimo da željena kategorija ima najvišu ocjenu od svih kategorija, ali to se

vjerojatno neće dogoditi prije treninga. Izračunavamo objektivnu funkciju koja mjeri pogrešku (ili udaljenost) između izlaznih rezultata i željenog uzorka rezultata. Stroj zatim modificira svoje unutarnje podesive parametre kako bi smanjio ovu pogrešku. Ovi podesivi parametri, koji se često nazivaju težinama, stvarni su brojevi koji se mogu vidjeti kao 'kvačice' koje definiraju ulazno-izlaznu funkciju stroja [12].

3.2.1.2. Nenadzirano učenje

Sljedeća značajna kategorija strojnog učenja je opisno ili nenadgledano učenje.

Algoritmi učenja bez nadzora uče isključivo iz neoznačenih ulaza x, koji su često dostupniji u izobilju od označenih primjera. Algoritmi za nenadzirano učenje obično proizvode generativne modele, koji mogu proizvesti realističan tekst, slike, audio i video, umjesto jednostavnog predviđanja oznaka za takve podatke [5, str. 776].

U ovom pristupu, imamo samo ulazne podatke, $D = \{x_i\}_{i=1}^{N}$, a cilj je identificirati uzorke u podacima [11, str. 2]. Ovo se također naziva proces otkrivanja znanja. Ovaj tip problema je manje precizno definiran jer ne postoji unaprijed specificiran oblik uzoraka koje treba pronaći, i način mjerenja pogreške nije očit.

U učenju bez nadzora agent uči obrasce u unosu bez ikakve eksplicitne povratne informacije. Najčešći zadatak učenja bez nadzora je klasteriranje: otkrivanje potencijalno korisnih klastera ulaznih primjera. Na primjer, kada se pokažu milijuni slika preuzetih s interneta, sustav računalnog vida može identificirati veliku skupinu sličnih slika koje bi govornik hrvatskog jezika nazvao "mačkama" [5, str. 653].

Russell i Norvig [5, str. 695] daju primjer u učenju bez nadzora, gdje taksi vozač postupno razvija koncept "dobrih prometnih dana" i "loših prometnih dana" bez prethodno označenih primjera.

3.3. Poticano učenje

Iako po definiciji poticano učenje spada u podskup učenja bez nadzora, zbog količine istraživanja koje se fokusiraju na ovu metodu, pogotovo u domeni videoigara, u ovom radu je izdvojeno kao zasebna kategorija.

U poticanom učenju agent uči iz niza poticaja: nagrada i kazni. Na primjer, na kraju šahovske partije agentu se kaže da je pobijedio (nagrada) ili izgubio (kazna). Na agentu je da odluči koje su radnje uzrokovale poticaje koji su bili najodgovorniji za to te da promijeni svoje radnje, kako bi ciljao na više nagrada u budućnosti [5].

Sutton i Barto [13, str. 1] opisuju poticano učenje kao proces učenja kako postupiti kako bi se maksimizirala vrijednost nagradnog signala, povezujući stanja i akcije. Ključna karakteristika ovog pristupa je da agent koji koristi ovu metodu nema unaprijed definiranu informaciju o tome koliku nagradu donosi svaka pojedina akcija, agent mora samostalno otkriti kroz iskustvo. Drugi značajan aspekt ovog pristupa je da izvođenje određenih akcija u većini slučajeva može

utjecati na nagrade za sve akcije koje slijede nakon toga.

Agent mora biti sposoban percipirati stanje (ili barem dio stanja) okoline i donositi akcije koje imaju utjecaj na to stanje, što znači da ima sposobnost mijenjanja stanja okoline. Također, agent mora imati ciljeve koji su ostvarivi u kontekstu te iste okoline. Navedene komponente poticanog učenja, prepoznavanje trenutnog stanja, izvođenje akcija koje mijenjaju to stanje i postizanje određenih ciljeva, promatraju se kao Markovljev proces odlučivanja (eng. *Markov decision process*) [5, str. 653].

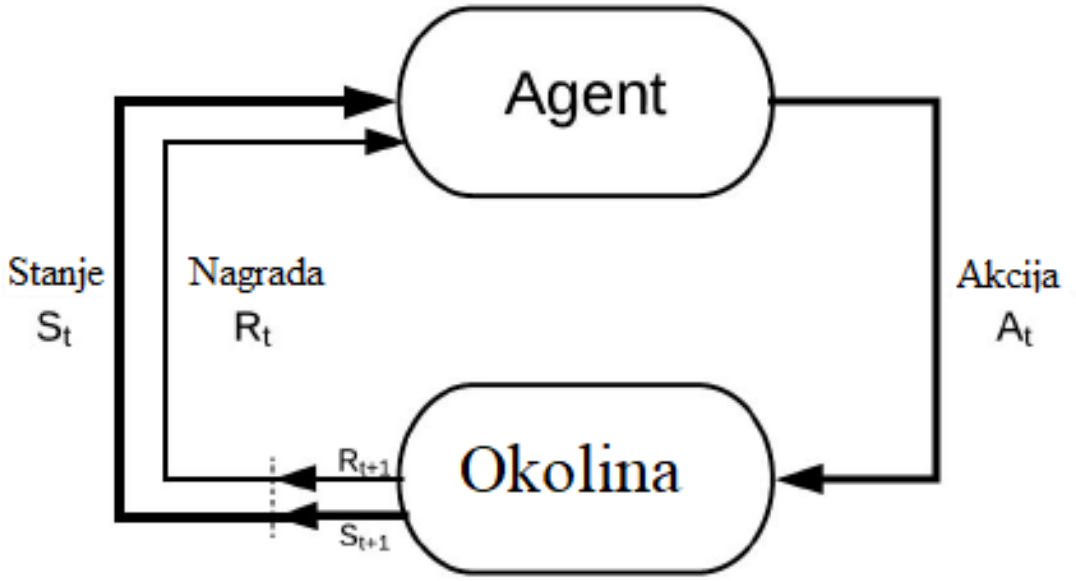
Uz agenta i okolinu, mogu se prepoznati četiri glavna podskupa sustava poticanog učenja: politika, nagradni signal, funkcija vrijednosti i, po želji, model okruženja [13, str. 6]. Politika definira način ponašanja agenta u određenom trenutku. Politiku agenta možemo objasniti kao ono što je agent naučio, koju akciju izvesti ovisno o trenutnom stanju agenta i okoline. Različite vrste poticanog učenja na različite načine dolaze do politike agenta, dok je neke vrste uopće ne koriste, kao što je i Q-učenje. Nagradni signal je ono što, nakon svakog diskretnog koraka, okolina vraća agentu u obliku broja koji prikazuje dobivenu nagradu za prošlu akciju [13, str. 6].

Prema Suttonu i Bartou [13, str. 6], u okviru poticanog učenja možemo prepoznati četiri ključna komponente: politiku, nagradni signal, funkciju vrijednosti i, opcionalno, model okoline.

- Politika definira način ponašanja agenta u određenom trenutku. Možemo je opisati kao strategiju koju je agent naučio kako bi odabirao akcije na temelju trenutnog stanja agenta i okoline. Različite metode poticanog učenja pristupaju definiranju politike na različite načine, dok neke vrste izravno prate akcije bez eksplicitnog oblikovanja politike.
- Nagradni signal predstavlja informaciju koju agent prima od okoline nakon svakog koraka. Ovaj signal je numerička vrijednost koja ocjenjuje ponašanje agenta u odnosu na zadane ciljeve. Nagradni signal igra ključnu ulogu u procesu učenja jer pomaže agentu razumjeti koje akcije su poželjne, a koje nisu.
- Funkcija vrijednosti povezuje stanja sa njihovom očekivanom budućom nagradom. Ovo je važna komponenta koja pomaže agentu procijeniti dugoročnu korisnost akcija. Postoji funkcija vrijednosti za svako stanje ili par stanja i akcija, ovisno o konkretnoj primjeni.
- Model okoline je aproksimacija stvarne okoline koja omogućava agentu da simulira kako će se okolina ponašati u različitim situacijama. Ova komponenta nije prisutna u svim metodama poticanog učenja, te ovisno o prisutnosti modela razlikujemo metode poticanog učenja na bazi modela (eng. *model-based*) i metode bez modela (eng. *model-free*).

Sve ove komponente zajedno čine osnovnu strukturu poticanog učenja, omogućujući agentu da uči i prilagođava svoje ponašanje kako bi maksimizirao ukupnu nagradu u okolini.

Na sljedećoj slici (slika 1) prikazano je međudjelovanje između agenta i okruženja u poticanom učenju.



Slika 1: Međudjelovanje agenta i okoline [14]

Agent izvršava akciju (eng. *Action*) A_t, na što okolina (eng. *Environment*) odgovara promjenom stanja (eng. *State*), odnosno pružanjem informacije o trenutnom stanju S_t, te dodjeljuje nagradu za tu akciju (koja je najčešće u numeričkom obliku) R_t.

Osim tih triju varijabli, postoji i četvrta komponenta, koja je stanje u koje okolina prelazi nakon izvršenja akcije. Sve te četiri komponente zajedno čine uređenu četvorku koja se promatra kao Markovljev proces odlučivanja koji se treba riješiti, odnosno kao uređena četvorka (S_t, A_t, R_t, S_{t+1}).

Markovljevi procesi odlučivanja (MDP), poznati i kao stohastičko dinamičko programiranje, su matematički modeli koji su se primarno koristili za modeliranje i rješavanje problema dinamičkog donošenja odluka u višekratnim razdobljima u stohastičkim okolnostima [15, str. 1].

Markovljev proces odlučivanja opisuje se uređenom četvorkom (S, A, P_a, R_a) [15, str. 1]. U kontekstu poticanog učenja, S predstavlja skup svih mogućih stanja koja okolina i agent mogu zauzeti, dok A obuhvaća skup svih akcija koje agent može izvršiti. Oznaka P_a, koja se ponekad naziva i $P_a(s, s')$, označava vjerojatnost da će izvođenje akcije a u stanju s rezultirati prijelazom u stanje s'. S druge strane, R_a ili preciznije $R_a(s, s')$ predstavlja numeričku vrijednost nagrade koju agent prima izvršavajući akciju a i prelazeći iz stanja s u stanje s'. Za razliku od uobičajene primjene u matematici, u kontekstu strojnog učenja, odnosno u poticanom učenju, Markovljev proces odlučivanja se promatra kao diskretan proces, što znači da se interakcija između agenta i okoline promatra u diskretnim vremenskim koracima.

Cilj poticanog učenja je maksimizirati očekivani zbroj nagrada. Razlikuje se od "samo rješavanja MDP-a" jer agentu nije dan MDP kao problem za rješavanje; agent je u MDP-u. Ono možda ne poznaje prijelazni model ili funkciju nagrađivanja i mora djelovati kako bi naučilo više. Zamislite da igrate novu igru čija pravila ne znate; nakon stotinjak poteza, sudac vam kaže "izgubili ste". To je ukratko poticano učenje. Točke gledišta dizajnera AI sustava, davanje signala nagrade agentu obično je mnogo lakše nego davanje označenih primjera kako se treba ponašati [5].

Prvo, funkcija nagrađivanja često je vrlo jednostavna i laka za specificiranje, zahtijeva samo nekoliko redaka koda da bi šahovskom agentu rekla je li pobijedio ili izgubio partiju ili da bi rekla agentu za automobilske utrke da je pobijedio ili izgubio utrku ili se srušio. Drugo, nije potrebno biti stručnjak, sposoban ponuditi ispravnu akciju u svakoj situaciji, kao što bi bio slučaj prilikom primijene nadziranog učenja.

Russell i Norvig [5] objašnjavaju kako se pokazalo da malo stručnosti može puno pomoći prilikom implementacije poticanog učenja. Na primjer nagrade za pobjedu/poraz za šah i utrke auta - su ono što nazivamo rijetkim nagradama (eng. *sparse*), jer se agentu u velikoj većini stanja ne daje nikakav informativni signal nagrade. U igrama kao što su tenis i kriket, možemo jednostavno osigurati dodatne nagrade za svaki osvojeni bod. U automobilskim utrkama agenta je moguće nagraditi za napredovanje po stazi u pravom smjeru. Kad se uči puzati, svaki pokret naprijed je uspjeh. Ove među-nagrade čine učenje mnogo lakšim [5].

Dok god se agentu može dati ispravan signal nagrade, poticano učenje pruža vrlo općenit način za izgradnju AI sustava. To posebno vrijedi za simulirana okruženja, gdje ne nedostaje prilika za stjecanje iskustva, u koje spada i domena primjene na video igre kao što je i prikazano u praktičnom dijelu ovog rada.

Jedan od izazova specifičnih za metodu poticanog učenja, a koji se ne pojavljuje u drugim oblicima strojnog učenja, je pronalaženje ravnoteže između istraživanja nepoznatih stanja i iskorištavanja već poznatih informacija. Kako bi maksimizirao ukupnu nagradu, agent mora preferirati akcije koje je već ranije isprobao i utvrdio da su efikasne u generiranju nagrade, ali kako bi pronašao te efikasne akcije, mora povremeno eksperimentirati s novim akcijama koje do tada nije isprobao.

Russell i Norvig [5] navode da je osmišljeno doslovno stotine različitih algoritama za poticano učenje, a mnogi od njih mogu kao alate koristiti širok raspon metoda učenja. Pristupe kategorizira na sljedeći način:

- Poticano učenje temeljeno na modelu: U ovim pristupima agent koristi prijelazni model okoline kako bi pomogao u tumačenju signala nagrađivanja i donošenju odluka o tome kako djelovati. Model može biti inicijalno nepoznat, u kojem slučaju agent uči model promatrajući učinke njegovih radnji, ili on može već biti poznat - na primjer, šahovski program može znati pravila šaha čak i ako ne zna kako birati dobre poteze. U djelomično vidljivim okruženjima, prijelazni model također je koristan za procjenu stanja. Sustavi poticanog učenje temeljeni na modelu često uče funkciju korisnosti $U(s)$, definiranu u smislu zbroja nagrada od stanja nadalje.

- Poticano učenje bez modela: U ovim pristupima agent niti zna niti uči prijelazni model za okolinu. Umjesto toga, on uči izravniji prikaz o tome kako se ponašati. Ovo dolazi u jednoj od dvije varijante:

- – Pretraživanje politike: Agent uči politiku $\pi(s)$ koja izravno preslikava stanja u akcije.
  – Učenje akcijske korisnosti: Najčešći oblik učenja korisnih radnji je Q-učenje, gdje agent uči Q-funkciju ili kvalitetu funkcija, $Q(s, a)$, koja označava zbroj nagrada od stanja s nadalje ako se poduzme radnja a. S obzirom na Q-funkciju, agent može odabrati što će učiniti u s pronalaženjem akcije s najvećom Q-vrijednošću [5].

3.4. Q-učenje

Q-učenje je metoda poticanog učenja bez modela [16]. Q-učenje se temelji na zapisivanju numeričkih vrijednosti u tablicu, koje se nazivaju Q-vrijednosti. Na temelju tih vrijednosti, agent odabire sljedeću akciju za izvršavanje. Nakon svakog diskretnog koraka, okolina pruža povratnu informaciju agentu, koji potom obrađuje te informacije putem ažuriranja vrijednosti u tablici Q-vrijednosti.

"Q" u Q-učenju označava kvalitetu (eng. *quality*). U kontekstu Q-učenja, kvaliteta predstavlja koliko je određena akcija korisna u ostvarivanju budućih nagrada [14].

Metoda Q-učenja izbjegava potrebu za modelom učenjem funkcije korisnosti akcije $Q(s, a)$ umjesto funkcije korisnosti $U(s)$. $Q(s, a)$ označava očekivanu ukupnu diskontiranu nagradu ako agent poduzme radnju a u s i nakon toga djeluje optimalno. Poznavanje Q-funkcije omogućuje agentu optimalno djelovanje jednostavnim odabirom maksimalnog $Q(s, a)$, bez potrebe za gledanjem unaprijed na temelju prijelaznog modela. [5]

Prema Sutton i Barto [13, str. 131] Q vrijednost se određuje na sljedeći način:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma maxQ(S_{t+1}, a) - Q(S_t, A_t)].$$

$Q(S_t, A_t)$ predstavlja vrijednost u tablici za izvršenu akciju A_t u stanju S_t, a R_{t+1} označava nagradu koju agent prima od okoline za izvršenu akciju. Varijabla α predstavlja koeficijent učenja, dok γ predstavlja koeficijent smanjenja buduće nagrade, što znači da svaka sljedeća akcija donosi manju nagradu jer agentu obično treba više vremena da postigne isti stanje odnosno cilj. $maxQ(S_{t+1}, a)$ predstavlja maksimalnu vrijednost sljedećeg stanja zapisanog u tablici koju agent može postići odabirom određene akcije.

Izraz unutar uglatih zagrada, odnosno $[R_{t+1} + \gamma \max Q(S_{t+1}, a) - Q(S_t, A_t)]$, potječe iz prethodne metode strojnog učenja koja se naziva "učenje po vremenskoj razlici" (eng. *Temporal difference learning*), često skraćeno kao TD. TD je vrsta učenja koja se razvila kao sinteza dviju prethodnih pristupa: Monte Carlo učenja i dinamičkog programiranja.

Kao što objašnjavaju Sutton i Barto [13, str. 119], TD metode, poput Monte Carlo metoda, mogu izravno učiti iz stvarnog iskustva bez potrebe za modeliranjem dinamike okoline. TD metode ažuriraju procjene djelomično na temelju drugih naučenih procjena, što znači da agent ne mora čekati kraj epizode da bi ažurirao svoje procjene (Q-vrijednosti). Ovo omogu-

ćava agentu da uči nakon svake izvršene akcije, bez potrebe za prethodnim znanjem o okolini. U ovom kontekstu, epizoda se definira kao niz stanja od početnog stanja S_{t_0} do trenutka kada se igra zaustavi, odnosno kada se postigne završno stanje.

Prema Morvanu [17], koraci algoritma za Q-učenje su:

Inicijaliziraj $Q(S_t, A_t)$ proizvoljno

Ponavljaj (za svaku epizodu):

Inicijaliziraj početno stanje S_{t_0}

Ponavljaj (za svaki korak u epizodi):

Izaberi akciju A_t koristeći politiku odabira prema Q (npr. ϵ - greedy)

Izvrši akciju A_t, promotri R_t i S_{t+1}

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max Q(S_{t+1}, a) - Q(S_t, A_t)]$

$S_t \leftarrow S_{t+1}$

Dok S_t nije konačan

Inicijalizacija početnog stanja $Q(S_t, A_t)$ je nije nužna, budući da će već u prvom koraku biti zamijenjena stvarnim vrijednostima koje agent dobiva od okoline. U određenim situacijama, ovaj korak nije nužan, te je moguće dodavati nove vrijednosti u Q-tablicu tek kada agent prvi put dođe u određeno stanje. Način definiranja jednog koraka unutar epizode varira ovisno o potrebama okoline koju agent pokušava savladati. U primjerima implementacije u ovom radu, korak će biti definiran na isti način kao što je prethodno opisano u koracima algoritma. Svaki korak sastoji se od izvršavanja akcije i promatranja nagrade te stanja koje okolina poprima nakon izvršenja te akcije.

Na početku igre, agent nema nikakvih saznanja o svom okruženju. U svakom sljedećem koraku, agent bira akciju koja će ga dovesti u stanje s najvećom očekivanom nagradom. U slučaju Q-učenja, agent preferira akciju s najvećom Q-vrijednošću. Ako ne postoji jedna konkretna najbolja akcija, agent će koristiti pseudoslučajni odabir akcije, što predstavlja politiku agenta koji koristi Q-učenje.

Pseudoslučajni odabir akcija ima ključnu ulogu u Q-učenju, čak i kada agent zna za postojanje akcija koje vode do veće nagrade. Pomoću pseudoslučajnog odabira se realizira rješenje prethodno spomenutog problema neodlučnosti između istraživanja nepoznatih stanja i korištenja već poznatih informacija. Bez pseudoslučajnog odabira, agent ima potencijal konstanto birati prvi nizu akcija koji mu je donio početnu nagradu (zato što bi ta nagrada bila veća od one prethodne koja je nepostojeća), iako taj niz vjerojatno ne bi bio optimalan za postizanje stvarnog cilja koji agent nastoji postići.

Učenje se odvija nakon izvršenja svake akcije i dobivanja povratne informacije u obliku nagrade i informacije o statusu okoline s kojom je agent u interakciji. Na temelju tih informacija izračunavaju se Q vrijednosti i zapisuju se u Q tablicu. U novijoj literaturi se zbog toga ova metoda naziva tablično Q-učenje (za razliku od dubokog o kojem će biti više govora kasnije)

Q-učenje suočava se s dva ključna problema. Prvi problem je da se agenti koji koriste Q-učenje teško prilagođavaju povećanju broja mogućih stanja i akcija, što rezultira znatno sporijim učenjem. Ovaj problem će biti detaljnije prikazan kroz prvi praktični primjer u ovom radu. Drugi problem poznat je kao pristranost maksimizacije (eng. *maximization bias*).

Do pristranosti maksimizacije dolazi u situaciji u kojoj postoji mnogo akcija a u određenom stanju s, čije stvarne vrijednosti $q(s, a)$ iznose nula, ali procijenjene vrijednosti $Q(s, a)$ su nesigurne i nekih su iznad nule, dok su drugi ispod nule. Maksimalna stvarna vrijednost je nula, no maksimalna procijenjena vrijednost je pozitivna, što uzrokuje pozitivnu pristranost.

Russell i Norvig navode da je važan dio jednadžbe je ono što ona ne sadrži: Q-agent za učenje ne treba model prijelaza, P(s'|s,a), ni za učenje ni za odabir akcije. Kao što je navedeno na početku poglavlja, metode bez modela mogu se primijeniti čak i u vrlo složenim domenama jer ne treba dati niti naučiti model. S druge strane, agent koji koristi Q učenje nema načina da gleda u budućnost, tako da može imati poteškoća kada su nagrade rijetke i moraju se konstruirati dugi nizovi akcija da bi se do njih došlo [5].

Q učenje ima bliskog srodnika koji se zove SARSA (za stanje, akciju, nagradu/*reward*, stanje, akciju). Pravilo ažuriranja za SARSA vrlo je slično Q-učenju, osim što se SARSA ažurira s Q-vrijednošću radnje a' koja je stvarno poduzeta [5].

Q-learning je algoritam učenja bez politike (eng. *off-policy*) jer uči Q-vrijednosti koje odgovaraju na pitanje "Koliko bi ova radnja vrijedila u ovom stanju, pod pretpostavkom da prestanem koristiti bilo koju politiku koju sada koristim i počnem djelovati u skladu s politika koja bira najbolju akciju (prema mojim procjenama)?" SARSA je algoritam na temelju politike: uči Q-vrijednosti koje odgovaraju na pitanje "Koliko bi ova radnja vrijedila u ovom stanju, pod pretpostavkom da se držim svoje politike?" Q-learning je fleksibilniji od SARSA-e, u smislu da Q-learning agent može naučiti kako se dobro ponašati kada je pod kontrolom širokog spektra politika istraživanja. S druge strane, SARSA je prikladan ako cjelokupnu politiku čak i djelomično kontroliraju drugi agenti ili programi, u kojem slučaju je bolje naučiti Q-funkciju za ono što će se zapravo dogoditi, a ne ono što bi se dogodilo da agent mora odabrati procijenjene najbolje akcije [5].

4. Duboko učenje

Duboko učenje je podskup strojnog učenja koji prema Deng, Yu i dr. [18] temelji na algoritmima za učenje višestrukih razina reprezentacije u svrhu modeliranja složenih odnosa među podacima. Značajke i koncepti više razine definirani su prema onima niže razine, a takva se hijerarhija značajki naziva dubokom arhitekturom. [18, str. 199]

Riječ duboko u sintagmi duboko učenje odnosi se na broj slojeva kroz koje se podaci transformiraju. Sustavi dubokog učenja imaju značajnu dubinu dodjele kredita (eng. *credit assignment path*) ili skraćeno CAP. CAP je lanac transformacija od ulaza do izlaza. CAP-ovi opisuju potencijalno uzročne veze između ulaza i izlaza. Za neuronsku mrežu u kojoj se aktivacija odvija samo u jednom smjeru (eng. *feed forward newtork*), dubina CAP-ova jednaka je dubini mreže odnosno broju skrivenih slojeva plus jedan (jer je izlazni sloj također parametriziran). Za rekurentne neuronske mreže, u kojima se signal može širiti kroz sloj više puta, CAP dubina je potencijalno neograničena. [19]

Dva su ključna aspekta zajednička među različitim opisima dubokog učenja na visokoj razini: modeli koji se sastoje od više slojeva ili faza nelinearne obrade informacija i metode za nadzirano ili nenadzirano učenje predstavljanja značajki na sukcesivno višim, apstraktnijim slojevima. Duboko učenje nalazi se u sjecištima istraživačkih područja neuronskih mreža, umjetne inteligencije, grafičkog modeliranja, optimizacije, prepoznavanja uzoraka i obrade signala.

Prema Deng, Yu i dr. [18, str. 199], drastično povećane mogućnosti obrade čipova, značajno povećana veličina podataka koji se koriste za obuku i nedavni napredak u istraživanju strojnog učenja i obrade signala/informacija omogućuju metodama dubokog učenja da učinkovito iskoriste složene, kompozicijske nelinearne funkcije, da nauče distribuirane i hijerarhijske reprezentacije značajki, te da učinkovito koriste i označene i neoznačene skupove podatka.

Učenje iz prikazanog (eng. *representation learning*) je skup metoda koje omogućuju da stroj uči direktno iz sirovih podataka i da automatski otkriva prikaze potrebne za detekciju ili klasifikaciju.

Metode dubokog učenja su metode učenja predstavljanja s višestrukim razinama predstavljanja, dobivene sastavljanjem jednostavnih, ali nelinearnih modula koji svaki transformiraju prikaz na jednoj razini (počevši od sirovog unosa) u prikaz na višoj, malo apstraktnijoj razini. Uz sastav dovoljno takvih transformacija, mogu se naučiti vrlo složene funkcije [12].

Ove razine predstavljanja obično se nazivaju slojevi. Slojevi se nizu jedan za drugim od ulaza do izlaza, te se svi slojevi osim ulaza i izlaza nazivaju se skriveni slojevi. Dvije vrste slojeva korištene u praktičnom dijelu ovog rada su gusti (eng. *dense*) i konvolucijski (eng. *convolutional*).

Za zadatke klasifikacije, viši slojevi reprezentacije pojačavaju aspekte ulaza koji su važni za diskriminaciju i potiskuju irelevantne varijacije. Slika, na primjer, dolazi u obliku niza vrijednosti piksela, a naučene značajke u prvom sloju reprezentacije obično predstavljaju prisutnost ili odsutnost rubova na određenim orijentacijama i lokacijama na slici. Drugi sloj obično otkriva motive uočavanjem određenih rasporeda rubova, bez obzira na male varijacije u položajima

rubova. Treći sloj može sastavljati motive u veće kombinacije koje odgovaraju dijelovima poznatih objekata, a sljedeći slojevi detektirali bi objekte kao kombinacije tih dijelova. Ključni aspekt dubinskog učenja je da ove slojeve značajki nisu dizajnirali ljudski inženjeri: one se uče iz podataka pomoću postupka učenja opće namjene [12].

Prema LeCun, Bengio i Hintonu [12], duboko učenje čini veliki napredak u rješavanju problema koji su godinama odolijevali najboljim pokušajima zajednice umjetne inteligencije. Pokazalo se da je vrlo dobar u otkrivanju zamršenih struktura u višedimenzionalnim podacima i stoga je primjenjiv u mnogim područjima znanosti, poslovanja i vlade. Osim što je oborio rekorde u prepoznavanju slika i govora, potukao je i druge tehnike strojnog učenja u predviđanju aktivnosti potencijalnih molekula lijekova, analizi podataka akceleratora čestica, rekonstrukciji moždanih krugova i predviđanju učinaka mutacija u nekodirajućoj DNK na ekspresiju gena i bolesti. Što je možda još iznenađujuće, dubinsko učenje je proizvelo izuzetno obećavajuće rezultate za razne zadatke u razumijevanju prirodnog jezika, posebno klasifikaciju tema, analizu osjećaja, odgovaranje na pitanja i prevođenje jezika [12].

4.1. Vrste mreža dubokog učenja

4.1.1. Mreža dubokih uvjerenja

Mreža dubokih uvjerenja (eng. *deep belief network*): probabilistički generativni modeli sastavljeni od više slojeva stohastičkih, skrivenih varijabli. Gornja dva sloja imaju neusmjerene, simetrične veze između sebe. Niži slojevi primaju usmjerene veze odozgo prema dolje od sloja iznad [18].

4.1.2. Boltzmannov stroj

Boltzmannov stroj (eng. *Boltzmann machine*): mreža simetrično povezanih jedinica sličnih neuronima koje donose stohastičke odluke hoće li biti uključene ili isključene [18].

4.1.3. Ograničeni Boltzmannov stroj

Ograničeni Boltzmannov stroj (eng. *restricted Boltzmann machine*): posebna vrsta BM-a koja se sastoji od sloja vidljivih jedinica i sloja skrivenih jedinica bez veza vidljivo-vidljivo ili skriveno-skriveno [18].

4.1.4. Duboka neuronska mreža

Duboka neuronska mreža (eng. *deep neural network* ili DNN): višeslojni perceptron s mnogo skrivenih slojeva, čije su težine potpuno povezane i često (iako ne uvijek) se inicijaliziraju korištenjem nenadzirane ili nadzirane tehnike preduvježbavanja [18].

U domeni videoigara ovo je daleko najčešće korištena vrsta mreže dubokog učenja, te je također i metoda koja je korištena za izradu praktičnog dijela ovog rada.

Slijedeće dvije vrste dubokih neuronskih mreža korištene su u praktičnom dijelu ovog rada.

4.1.4.1. Unaprijed usmjerena mreža

Unaprijed usmjerena (eng. *feedforward*) mreža, kao što ime sugerira, ima veze samo u jednom smjeru - to jest, ona tvori usmjereni aciklički graf s određenim ulaznim i izlaznim čvorovima. Svaki čvor izračunava funkciju svojih ulaza i prosljeđuje rezultat svojim nasljednicima u mreži. Informacije teku kroz mrežu od ulaznih čvorova do izlaznih čvorova i nema petlji [5].

U praktičnom dijelu ovog rada, ova vrsta korištena je u slučaju gdje agent dobiva znatno pojednostavljen opis okoline kao ulaz i ima mali broj akcija na odabir (koji su zapravo izlazi i feedforward mreže).

4.1.5. Konvolucijska mreža

Konvolucijska neuronska mreža (eng. *convolutional neural network*) je ona koja sadrži prostorno lokalne veze, barem u ranim slojevima, i ima uzorke težine koji se repliciraju preko jedinica u svakom sloju. Uzorak težine koji se replicira preko više lokalnih regija naziva se jezgra jezgre, a postupak primjene jezgre na piksele slike (ili na prostorno organizirane jedinice u sljedećem sloju) naziva se konvolucija [5].

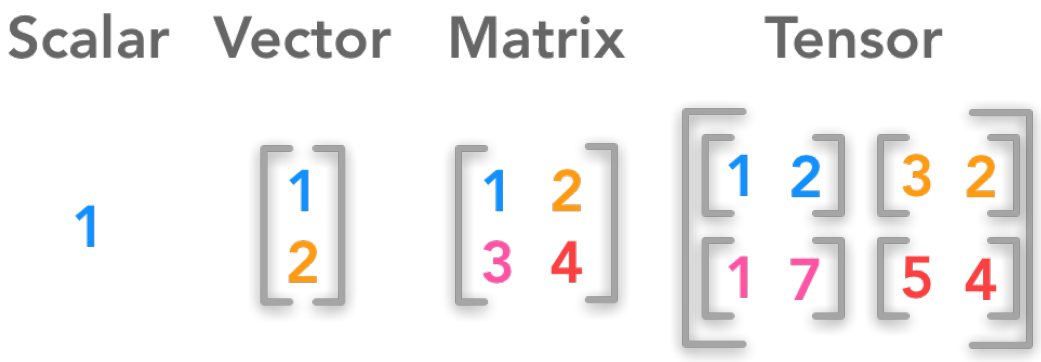
Ova vrsta mreže uglavnom se koristi za situacije u kojima agent pokušava naučiti direktno iz prikaza slike (uglavnom reprezentirani kao pikseli na ekranu), te je za istu svrhu korišten i u praktičnom dijelu.

4.2. Tenzor

Jedan od pojmova koji se često spominje u dubokom učenju je tenzor.

Vasilescu i Terzopoulos [20] navode da dok su matrice linearni operatori definirani nad vektorskim prostorom, ove generalizacije, koje se nazivaju tenzori, definiraju multilinearne operatore nad skupom vektorskih prostora. Dakle, multilinearna algebra, algebra tenzora višeg reda, uključuje linearnu algebru i matrice/vektore/skalare kao poseban slučaj. Multilinearna algebra služi kao objedinjujući matematički okvir prikladan za rješavanje niza izazovnih problema u znanosti o slici i vizualnom računalstvu.

Ova definicija odnosi se na matematičko razumijevanje pojma tenzor, iako zapravo opisuje pojam koji je lakše intuitivno razumjeti, barem na način na koji se on koristi u dubokom učenju. Ova intuitivna definicija lakše se objašnjava praktičnim primjerom, u praktičnom smislu tenzor je poopćenje vektora, odnosno tenzor je skup skupova vektora promatran kao jedinstvena struktura. Svaki vektor je ujedno i tenzor reda jedan, dok su čak i skalari tenzori reda nula. Ovaj odnos opisan je na slici 2.



Slika 2: Tenzori reda 0, 1, 2 i 3 [21]

Iako su svi elementi na slici zapravo tenzori, opisivanje tenzora reda manjeg od tri kao tenzor nepotrebno dovodi do zamršenja zato što ti elementi već imaju svoje dobro poznate nazive.

Kako se gotovo sve reprezentacije podataka u dubokom učenju prikazuju kao vektori, lako je uvidjeti zašto tenzori imaju široku upotrebu o ovoj domeni.

Duboko učenje moguće je kombinirati sa sve tri do sada spomenute paradigme strojnog učenja. Kako se praktični dio ovog rada odnosi na izradu agenta strateške video igre, najpogodnija paradigma za kombiniranje sa dubokim učenjem je poticano učenje. Rezultantna kombinacija naziva se duboko poticano učenje (eng. *deep reinforcement learning*).

5. Duboko poticano učenje

Ova domena je vrlo eksperimentalna, i uglavnom se napretci pokazuju kroz praktične primjere. U slučaju dubokog poticanog učenja, to je rad istraživača Google DeepMinda koji su u istom pokazali algoritam koji nazivaju duboko Q-učenje (eng. *deep Q-learning* ili skraćeno DQN) i njegovu upotrebu na različitim video igrama dostupnima za Atari 2600 platformu. Većinu koncepata koje DQN uvodi koriste se i u budućim naprednijim metodama, a sam DQN korišten je za izradu praktičnog dijela ovog rada.

Sva zasebnost ove kategorije strojnog učenja zasniva se u metodama implementacije, teoretski ona sadržava sve opisano u prijašnjim poglavljima duboko učenje i poticano učenje.

Mnih, Kavukcuoglu, Silver i dr. [22] navode da su namjeravali stvoriti jedinstveni algoritam koji bi bio u stanju razviti širok raspon kompetencija na raznolikom nizu izazovnih zadataka - što je središnji cilj opće umjetne inteligencije koji je izmicao prethodnim naporima. Kako bi to postigli, razvili smo novog agenta, duboku Q-mrežu (DQN), koja je u stanju kombinirati poticano učenje s klasom umjetne neuronske mreže poznate kao duboke neuronske mreže. Napredak u dubokim neuronskim mrežama, u kojima se nekoliko slojeva čvorova koristi za izgradnju sve apstraktnijih prikaza podataka, omogućio je umjetnim neuronskim mrežama učenje koncepata kao što su kategorije objekata izravno iz sirovih senzorskih podataka. Koristimo jednu posebno uspješnu arhitekturu, duboku konvolucijsku mrežu, koja koristi hijerarhijske slojeve popločanih konvolucijskih filtera za oponašanje učinaka receptivnih polja, iskorištavajući tako lokalne prostorne korelacije prisutne u slikama, i izgradnja otpornosti na prirodne transformacije kao što su promjene gledišta ili razmjera. Razmatramo zadatke u kojima agent stupa u interakciju s okolinom kroz niz opažanja, akcija i nagrada. Cilj agenta je odabrati akcije na način koji maksimizira kumulativnu buduću nagradu. Formalnije, koristimo duboku konvolucijsku neuronsku mrežu za aproksimaciju optimalne funkcije radnje-vrijednosti

$$Q(s,a) = max \sum_{i=t}^{\pi} [r_t + \alpha r_{t+1} + \alpha^2 r_{t+2} + ... | s_t = s, a_t = a, \pi]$$

koja predstavlja maksimalnu sumu nagrada r_t umanjenu za α u svakom vremenskom koraku t, postignuto politikom $\pi = P(a|s)$ nakon opservacije s i poduzimanja akcije a. [22]

Učenje u kontekstu dubokih neuralnih mreža odnosi se na promjenu načina aktivacije čvorova, izmjenom težina dodijeljenih vezama između neurona. Ako bi se učenje odvijalo u svakom koraku, ova metoda bila bi manje uspješna od klasičnog poticanog učenja, zato što podaci nemaju normalnu distribuciju.

Mnih, Badia, Mirza i dr. [23] navode da je redoslijed promatranih podataka na koje nailazi online RL agent je nestacionaran, a on-line ažuriranja RL-a snažno su korelirana. Pohranjivanjem agentovih podataka u memoriju za ponavljanje iskustva, podaci se mogu grupirati ili nasumično dogoditi u različitim vremenskim koracima. Agregiranje preko memorije na ovaj način smanjuje nestacionarnost i dekorelira ažuriranja, ali u isto vrijeme ograničava metode na algoritme poticanog učenja bez politike. Duboki RL algoritmi koji se temelje na ponavljanju

iskustva postigli su uspjeh u izazovnim domenama kao što je Atari 2600. Međutim, ponavljanje iskustva ima nekoliko nedostataka: koristi više memorije i računanja u svakom koraku interakcije, i zahtijeva algoritme učenja bez politike [23].

Prema Mnih, Kavukcuoglu, Silver i dr. [22], u određenim igrama DQN može otkriti relativno dugoročnu strategiju. Unatoč tome, igre koje zahtijevaju od agenta da nauči strategiju koja uzima u obzir veliki broj koraka unaprijed još uvijek predstavljaju veliki izazov za sve postojeće agente uključujući DQN. Demonstrirali su da ista arhitektura može uspješno naučiti kontrolne politike u nizu različitih okruženja sa vrlo minimalnim predznanjem, primajući samo piksele i rezultat igre kao ulaze, i koristeći isti algoritam, mrežnu arhitekturu i hiperparametre na svakoj igri, upućen samo u unose koje bi ljudski igrač imao.

Implementacija DQN algoritma spomenuta u ovom radu koristi konvolucijsku duboku neuralnu mrežu, odnosno barem neki od skrivenih slojeva implementirani su kao konvolucijski.

6. Praktični primjer: Izrada agenta koji igra stratešku videoigru

U ovom poglavlju prikazan je razvoj dva agenta koji igraju video igru Starcraft 2 i uče putem dubokog Q-učenja.

6.1. StarCraft 2

StarCraft II je znanstveno-fantastična videoigra strategije u stvarnom vremenu koju je razvila i objavila tvrtka Blizzard Entertainment [24].

Igra se bazira na ratu između tri rase, Terran, Protoss i Zerg. Svaka rasa ima zasebne građevine i jedinice koje može izgraditi, te shodno tome i različite načine i strategije kojima dolaze do pobjede. StarCraft II se odvija u stvarnom vremenu što znači da igrači trebaju istovremeno izvršavati veliki broj radnji kao što su izgradnja vlastite baze, izviđanje protivničke baze, te u kasnijim stadijima igre čak i borbu protiv neprijateljske vojske [25].

PySC2 službena dokumentacija nudi mogućnost preuzimanja velikog broja različitih mapa namijenjenih za nešto što se nazivaju mini igre (eng. *mini games*). Mini-igre su dizajnirane za jednog igrača, fiksne duljine i vježbaju različite aspekte igre. Pružaju agentu nagradu koja agentu daje do znanja koliko dobro radi. Rezultat bi trebao razlikovati loše agente od dobrih agenata.

Prvi agent implementiran u ovom radu izgrađen je da radi i testiran na dvije mini igre, iako bi u teoriji bio primjenjiv na sve mini igre koje su rješive korištenjem samo jedne akcije, pomicanjem jedinica po ekranu. Smanjenjem izbora akcija na samo jednu, i fokusiranjem na širi spektar ulaza ovaj agent znatno brže uči i može pokazati svoje sposobnosti već u nekoliko odigranih epizoda.

Drugi agent je proširena verzija Agenta jedan spomenutog u završnom radu [25]. Koristi isti umanjeni pogled, samo umjesto klasičnog (tabličnoga) Q učenja koristi duboko.

Svaki agent sastoji se od dvije Python skripte. U svakoj sekciji ovog poglavlja prva skripta koja će biti objašnjena je skripta pod nazivom Run.py, koja je zapravo glavna skripta koja je zadužena za pokretanje igre i meta podatke potrebne za to pokretanja. Run.py također inicijalizira objekte iz klase definirane u drugoj skripti Brain.py, u kojoj su implementirani i agent koji igra igru, i duboka neuronska mreža zadužena za učenje.

6.2. Mini igra

U ovoj sekciji opisana je implementacija agenta koji igra mini igru. Agent je ograničen na samo jednu akciju te se pravo odlučivanje zapravo fokusira na to gdje na ekranu agent treba kliknuti (poslati jedinice) a ne na koju od mnogih akcija odabrati.

6.2.1. Run.py

```
from absl import app

from pysc2.env import sc2_env
from pysc2.env.run_loop import run_loop

from Brain import Agent
```

Run započinje uvozom dodataka. Modul *app* iz dodatka *absl* služi za pokretanje aplikacije. Slijede dva uvoza *pysc2* dodatka koji se koriste za postavljanje meta podataka dok je *run loop* wrapper koji je zadužen za izvođenje svih potrebnih pod rutina da bi agent zapravo igrao igru.

```
def main(unused_argv):

    mapList = ["CollectMineralShards", "MoveToBeacon"]

    agentList = [Agent()]

    with sc2_env.SC2Env(
         map_name = mapList[1],
         players = [sc2_env.Agent(sc2_env.Race.terran)],
         agent_interface_format = sc2_env.parse_agent_interface_format(
             feature_screen=84,feature_minimap=64),
         step_mul = 16,
         game_steps_per_episode = 0,
         visualize = True
        ) as env:
            run_loop(agentList, env)
```

Main funkcija definira tri stvari, mapu koja će se igrati, listu agenata koji će igrati tu mapu i meta podatke potrebne za pokretanje te igre, te se u konačnici ta igra pokreće putem naredbe *run_loop*. U ovom primjeru prikazana je konfiguracija u kojoj se igra mini igra pod nazivom *MoveToBeacon*, a izbor druge mini igre odvio bi se zamjenom indeksa unutar *mapList[1]*.

Sljedeći blok postavlja okruženje koristeći klasu *SC2Env*. Parametri proslijeđeni konstruktoru konfiguriraju različite aspekte okruženja, kao što je mini igra koja će se igrati, koji su igrači (u ovom slučaju, jedan agent koji je uvezen iz koda koji je opisan kasnije u radu), format sučelja agenta koji definira veličine koje su agentu dostupne, *step_mul* koji opisuje koliko je agentu dozvoljeno ubrzanje u kontekstu izvršenja akcija u odnosu na stvarno vrijeme, ograničenje koraka unutar jedne epizode (game_steps_per_episode = 0 označava da nema ograničenja, igra se dok se ne završi), i hoće li se igra vizualizirali što je u ovom slučaju postavljeno na True.

Ova vizualizacija nije stvarni prikaz kakav bi ljudski igrači imali, nego pojednostavljeni koji omogućava ljudskim korisnicima da prate znatno ubrzane akcije agenta.

Naredba *with* ovdje se koristi kako bi se osiguralo ispravno postavljanje i čišćenje re-

sursa povezanih s okolinom.

```
if __name__ == "__main__":
    app.run(main)
```

Ovaj dio koda provjerava izvodi li se skripta kao glavni program, i ako je, poziva prethodno definiranu funkciju *main*.

6.2.2. Brain.py

Brain.py

```
import numpy as np
import tensorflow as tf
import os

from collections import deque

from pysc2.agents import base_agent
from pysc2.lib.actions import FUNCTIONS

SCREEN_SIZE = [84, 84]

MAIN_NETWORK_NAME = "MainNetwork"
TRAINING_NETWORK_NAME = "TrainingNetwork"

SAVE_PATH = "./checkpoints/minigame.ckpt"
INDEX_PATH = SAVE_PATH + ".index"

END_STEP_IDENTIFIER = 2
```

Ovaj dio koda predstavlja uvoz dodataka *numpy* i *tensorflow* čija je upotreba prethodno opisana, uvoze se *os* koji se koristi kasnije za manipulaciju putanjama, *deque* koji se koristi za stvaranje red za čekanje, te *base_agent* i *FUNCTIONS* iz *pysc2* od kojih se prvi koristi za stvaranje agenta komptabilinog sa okružjem, a drugi za pozivanje akcija koje agent može izvršavati.

Nakon uvoza dolazi definiranje konstanti koje će se koristiti u daljnjim dijelovima koda.

```
class Network(object):
    def __init__(self, name):

        tf.compat.v1.disable_eager_execution()

        self.learningRate = 0.001
        self.name = name

        with tf.compat.v1.variable_scope(self.name):
            self.episode = tf.Variable(
```

```
        initial_value = 0,
        trainable = False,
        name = "episode"
        )

    self.step = tf.Variable(
        initial_value = 0,
        trainable = False,
        name = "step"
        )
```

U ovom dijelu koda definiran je konstruktor klase *Network* koji uzima parametar imena za postavljanje naziva instance. Slijedi definicija dvije varijable *epizoda* i *korak*, inicijalizirane su vrijednostima 0. Obje varijable imaju *trainable* postavljeno na *False* što znači da se ne mogu trenirati, one se zapravo koriste za praćenje na trenutne epizodi i koraka u treningu mreže. Varijabla learningRate označava stopu učenja za mrežu koja će kasnije biti proslijeđena u algoritam korišten za optimizaciju.

Ova mreže koristi grafnu verziju implementacije, cijela mreža se izgradi kao graf prilikom inicijalizacije objekta klase *Network* prije bilo kojeg korištenja mreže za treniranje.

```
    self.inputs = tf.compat.v1.placeholder(
        dtype = tf.int32,
        shape = [None, 84, 84],
        name = "inputs"
        )


    self.targets = tf.compat.v1.placeholder(
        dtype = tf.float32,
        shape = [None],
        name = "targets"
        )

    self.incrementEpisode = tf.compat.v1.assign(
        ref = self.episode,
        value = self.episode + 1,
        name = "incrementEpisode"
        )

    self.transposed = tf.transpose(
        a = self.inputs,
        perm = [0, 2, 1],
        name = "transpose"
        )

    self.oneHotEncoded = tf.one_hot(
        indices = self.transposed,
        depth = 5,
        axis = -1,
        name = "oneHotEncoded"
```

```
    )

self.embed = tf.compat.v1.layers.conv2d(
    inputs = self.oneHotEncoded,
    filters = 1,
    kernel_size = [1, 1],
    strides = [1, 1],
    padding = "SAME",
    name = "embed"
    )

self.convolutional = tf.compat.v1.layers.conv2d(
    inputs = self.embed,
    filters = 20,
    kernel_size = [3, 3],
    strides = [1, 1],
    padding = "SAME",
    name = "convolutional"
    )

self.activationConvolutional = tf.nn.relu(
    features = self.convolutional,
    name = "activationConvolutional"
    )

self.output = tf.compat.v1.layers.conv2d(
    inputs = self.activationConvolutional,
    filters = 1,
    kernel_size = [1, 1],
    strides = [1, 1],
    padding = "SAME",
    name = "output"
    )

self.flatten = tf.compat.v1.layers.flatten(self.output, name = "flatten"
    )

self.actions = tf.compat.v1.placeholder(
    dtype = tf.float32,
    shape = [None, 84 * 84],
    name = "actions"
    )

self.prediction = tf.reduce_sum(
    input_tensor = tf.multiply(self.flatten, self.actions),
    axis = 1,
    name = "prediction"
    )

self.loss = tf.reduce_mean(
    input_tensor = tf.square(self.targets - self.prediction),
    name = "loss"
    )
```

```
self.optimizer = tf.compat.v1.train.AdamOptimizer(self.learningRate).
    minimize(self.loss, global_step = self.step)

self.saver = tf.compat.v1.train.Saver()
```

Ovaj dio koda je nastavak i uključuje rezervirana mjesta (eng. *placeholder*) za ulaze oblika [*None*, 84, 84] i ciljeve. Slijedi definicija operacije *incrementEpisode* koja povećava broj epizoda.

Ulazni podaci se transponiraju zato što okolina pruža koordinate u obrnutom redoslijedu od onog uglavnom korištenog (prva koordianta je y umjesto x). Nakon transponiranja odvija se korak koji se naziva *one hot encoding*. Ova metoda se koristi kada imamo numeričke kategoričke podatke, odnosno ne postoji uređeni odnos među brojevima. Kao što je prethodno objašnjeno ulazi se sastoje od matrice veličine 84 x 84, od kojih svaki element poprima vrijednosti od 0 do 4, ali to samo opisuje vrstu (kategoriju) jedinica na ekranu, a ne nekakav odnos između elementa (npr. 1 < 2 ne bi vrijedilo u ovom slučaju). Da umjetna neuronska mreža ne bi donijela krivu odluku i percipirala ovakve podatke kao stvarne vrijednosti koristi se one hot encoding da bi se podaci pretvorili u vektore koji imaju 0 na svim mjestima osim na jednom koji predstavlja tu kategoričku vrijednost. Ovaj koncept lakše je promatrati na jednostavnom primjeru prikaza 3 boje [26]

crvena - [1, 0, 0]

zelena - [0, 1, 0]

plava - [0, 0, 1]

U slučaju ovog rada one hot encoding se odvija na dubini (eng. *depth*) 5.

Sljedeći sloj je *self.embed* koji je zadužen za smanjenje dimenzionalnosti i poboljšanje efikasnosti mreže. Čak i niskodimenzionalna ugrađivanja mogu sadržavati mnogo informacija o relativno velikoj zakrpi slike. 1x1 konvolucije koriste se za izračunavanje smanjenja prije skupih 3x3 i 5x5 konvolucija. Osim što se koriste kao redukcije, također uključuju upotrebu ispravljene linearne aktivacije što ih čini dvostrukom namjenom. [27]

Nakon ovog sloja slijedi spomenuti "skuplji" sloj sa 3x3 konvolucijom *self.convolutional*. Između ovog i slijedećeg sloja *self.output* koji je zapravo skup Q vrijednosti po svakoj gomili, definiran je još jedan konstrukt *self.activationConvolutional* koji opisuje način aktivacije između dva prethodno spomenuta sloja.

Izlazne Q vrijednosti se izravnavanju u *self.flatten* da bi se mogle množiti sa *self.actions* i koristiti za izračunavanje predkicije (*self.prediction*) i gubitka (*self.loss*), odnosno izračunava se srednja kvadratna greška (eng. *mean square error*), kao što je opisano u originalnim DQN radovima [22] [28].

U ovom primjeru kao optimizator koje se koristi za minimiziranje gubitka odabran je optimizator *Adam*. Osim algoritma Adam isproban je i algoritam RSMProp kao u [22], bez primjetne razlike u performansi mreže.

```
class Agent(base_agent.BaseAgent):
    def __init__(self):
        super(Agent, self).__init__()

        self.epsilonMin = 0.01
        self.epsilonMax = 1.0
        self.epsilonDecaySteps = 10000

        self.trainFrequency = 250 # 500, 1000
        self.batchSize = 20
        self.discountFactor = 0.95

        tf.compat.v1.reset_default_graph()

        self.network = Network(MAIN_NETWORK_NAME)
        self.trainingNetwork = Network(TRAINING_NETWORK_NAME)

        self.memory = deque(maxlen = 10000)

        self.lastState = None
        self.lastAction = None

        self.sess = tf.compat.v1.Session()
        if os.path.isfile(INDEX_PATH):
            self.network.saver.restore(self.sess, SAVE_PATH)
            self.updateNetworks()

        else:
            self.sess.run(fetches = tf.compat.v1.global_variables_initializer())
```

Prošli isječak koda predstavlja početak definicije klase *Agent* koja nasljeđuje *BaseAgent* prethodno uvezen iz dodatka *pysc2*. Slijede definicije parametara čija će se funkcija opisati prilikom njihovog korištenja kasnije u kodu. Važno je napomenuti dio gdje se agentovo pamćenje (eng. *memory*) definira kao red čekanja duljine 10000. Prema Mnih, Kavukcuoglu, Silver i dr. [28] primarni razlog za nasumično uzorkovanje prošlih podataka je razbijanje korelacija u podacima.

```
    def step(self, obs):
        self.steps += 1

        if obs.step_type == END_STEP_IDENTIFIER:
            self.sess.run(fetches = self.network.incrementEpisode)
            self.network.saver.save(self.sess, SAVE_PATH)

        if FUNCTIONS.Move_screen.id not in obs.observation.available_actions:
            return FUNCTIONS.select_army("select")

        else:
            state = obs.observation.feature_screen.player_relative
```

```
x, y = self.chooseAction(state)

if (len(self.memory) > self.batchSize):
    states, actions, targets = self.getBatch()

    self.sess.run(
        fetches = [self.network.loss, self.network.optimizer],
        feed_dict = {
            self.network.inputs: states,
            self.network.actions: actions,
            self.network.targets: targets
            }
        )
if self.steps % self.trainFrequency == 0:
    self.updateNetworks()

if self.lastState is not None:
    self.memory.append((self.lastState, self.lastAction, obs.reward,
        state))

self.lastState = state
self.lastAction = np.ravel_multi_index((x, y), SCREEN_SIZE)

return FUNCTIONS.Attack_screen("now", (x, y))
```

Funkcija *step* je zapravo nadjačavanje (eng. *override*) funkcije istog imena koja je naslijeđena iz klase *BaseAgent*. Ona se izvodi u svakom koraku interakcije s okolinom. Započinje povećanjem brojača koraka za 1, provjerom je li postignut zadnji korak epizode, te u slučaju da je, poveća broj epizode spremljen unutar objekta klase *Network* i sprema cijeli mrežu kao graf na prethodno specificiranu putanju.

Nakon toga provjerava je li trenutno prvi korak odnosno je li potrebno označiti jedinice da bi se iste moglo pomicati svrhu rješavanja mini igre.

U nastavku se na temelju trenutnog stanja donosi sljedeća akcija. Trenutno stanje je matrica veličine piksela na ekranu (84 x 84), te svaki element matrice ima jedan od 5 mogućih brojčanih vrijednosti, koje označavaju što se nalazi na tom pikselu (npr. agentove jedinice su sve označene istim brojem, dok su neutralni objekti označeni drugim).

Kako je jedna od korištenih optimizacija "učenje u gomili" (eng. *batch learning*) slijedi provjera ima li agent trenutno dovoljno podataka u pamćenju odnosno sadrži li broj veći od *batchSize* varijable koja predstavlja veličinu gomile.

U slučaju da je *batch* dovoljno velik odvija se korak učenja u kojem agent dohvaća *batch* funkcijom *getBatch()* koja će biti naknadno objašnjena, te prosljeđuje dobivene podatke u mrežu.

Valja napomenuti drugu optimizaciju učenja, uvođenje druge mreže čije se težine ažuriraju rjeđe.

Prema Mnih, Kavukcuoglu, Silver i dr. [22], svakim C ažuriranjem kloniramo mrežu Q kako bismo dobili ciljnu mrežu Q' i koristimo Q' za generiranje Q-ciljeva učenja y_j za sljedeća

C ažuriranja Q. Ova izmjena čini algoritam stabilnijim u usporedbi sa standardnim online Q-učenjem, gdje ažuriranje koje povećava $Q(s_t, a_t)$ često također povećava $Q(s_{t+1}, a)$ za sve a i stoga također povećava cilj y_j, što može dovesti do oscilacija ili odstupanja politike [22].

Da bi pojednostavili učenje i spremanje akcija koristi se naredba *ravel_multi_index*, ona pretvara koordinate matrice iz formata (stupac, red) u (broj), gdje je broj zapravo odmak elementa od početka matrice u memoriji računala (kao da se višedimenzionalna matrica promatra kao jednodimenzionalna).

Za učenjem slijedi korak dodavanja trenutne situacije u agentovo sjećanje te izvršavanje akcije pomicanja jedinica po ekranu.

```
def getBatch(self):
    indices = np.random.choice(
        np.arange(len(self.memory)),
        size = self.batchSize,
        replace = False
        )

    batch = [self.memory[i] for i in indices]

    states = np.array(
        [each[0] for each in batch]
        )
    actions = np.array(
        [each[1] for each in batch]
        )
    rewards = np.array(
        [each[2] for each in batch]
        )
    nextStates = np.array(
        [each[3] for each in batch]
        )

    actions = np.eye(84 * 84)[actions]

    predictedOutputs = self.sess.run(
        fetches = self.trainingNetwork.output,
        feed_dict = {self.trainingNetwork.inputs: nextStates}
        )

    targets = [rewards[i] + self.discountFactor * np.max(predictedOutputs[i])
        for i in range(self.batchSize)]

    return states, actions, targets
```

Ova metoda pod nazivom *getBatch*, služi za dohvaćanje jedne gomile (*batch*) iskustava iz pamćenja agenta i pripremu podataka koji će se koristiti za treniranje neuronske mreže.

Na početku se pseudoslučajno odabiru indeksi (*indices*) te se potom vade potrebni sve zasebne informacije iz agentovog sjećanja, a to su stanja, akcije, nagrade i buduća stanja.

Akcije ovdje također prolaze kroz prethodno opisanu tehniku *one hot encoding* iz istog razloga, predstavljaju koordinate piksela na ekranu, i nemaju nekog stvarnog odnosa i usporedbe.

U ovom koraku se računaju TD ciljevi koji će biti proslijeđeni mreži u svrhu učenja, ovo je optimizacija za olakšavanje računanja same mreže.

Dio formule na koje se odnose ovi ciljevi je:

$r + \lambda max_{a'} Q(s', a'; \theta_i^-)$

Gdje r predstavlja nagradu (*rewards[i]*), λ koeficijent smanjenja vrijednosti (*self.discountFactor*), i $max_{a'} Q(s', a'; \theta_i^-)$ maksimalnu Q vrijednost koju mreža predviđa u stanju s' i akcijom a'.

Za izračunavanje odnosno predviđanje Q vrijednosti koristi se *self.trainingNetwork*, ona mreža čije se težine rjeđe ažuriraju.

```
def reset(self):
    self.lastState = None
    self.lastAction = None

    episode = self.network.episode.eval(session = self.sess)
    step = self.network.step.eval(session = self.sess)
    print("Episode␣", episode)
    print("Step␣", step)
```

Metoda *reset* je još jedan override iz baznog agenta te se automatski izvršava na početku svake epizode treninga, u ovom slučaju to se odvija prilikom pokretanja svake runde mini igre.

Posljednje agentovo zapamćeno stanje i akcija se brišu postavljanjem na *None*, te se dohvaćaju epizoda i korak spremljeni u Network mreži da bi se korisniku mogli potom ispisati u konzolu prilikom izvršavanja.

```
def updateNetworks(self):
    mainNetworkVariables = tf.compat.v1.get_collection(
        key = tf.compat.v1.GraphKeys.TRAINABLE_VARIABLES,
        scope = MAIN_NETWORK_NAME
        )
    trainingNetworkVariables = tf.compat.v1.get_collection(
        key = tf.compat.v1.GraphKeys.TRAINABLE_VARIABLES,
        scope = TRAINING_NETWORK_NAME
        )

    variablesToUpdate = []
    for mainVariable, trainingVariable in zip(mainNetworkVariables,
        trainingNetworkVariables):
        variablesToUpdate.append(trainingVariable.assign(mainVariable))
```

```
        self.sess.run(fetches = variablesToUpdate)
```

Metoda *updateNetworks* služi za sinkronizaciju parametara između glavne (eng. *main*) neuronske mreže i mreže za obuku (eng. *training*). Dohvaća varijable (kao elemente prethodno konstruiranog grafa) koje se mogu trenirati iz obje mreže, prepisuje podatke iz *main* mreže u *training* mrežu, stavlja imena prepisanih varijabli u listu, te prosljeđuje tu listu *run()* metodi koja izvršava ažuriranje grafa odnosno mreže.

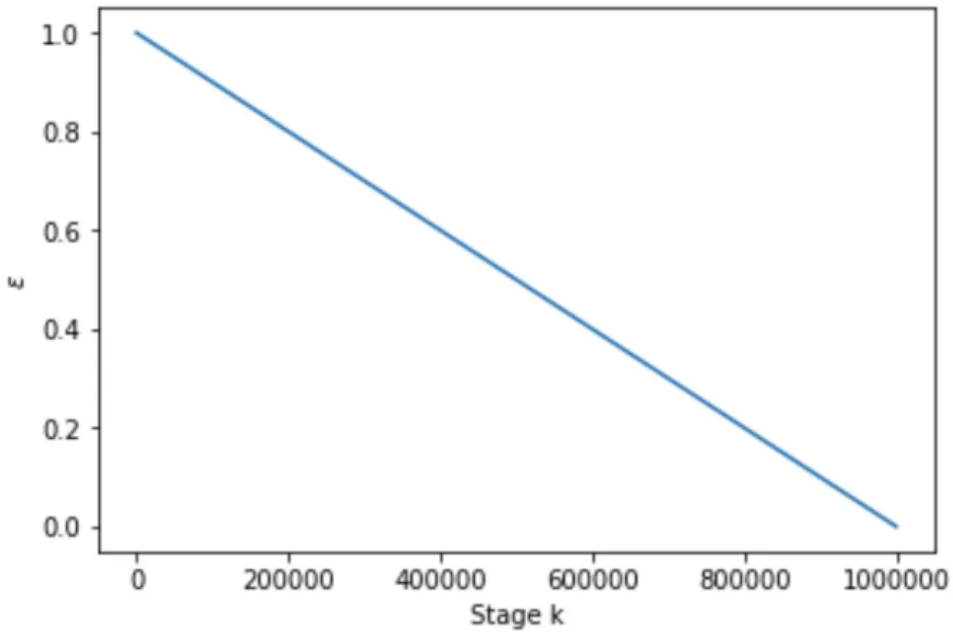
```
def chooseAction(self, state):
    step = self.network.step.eval(session = self.sess)
    epsilonDelta = ((self.epsilonMax - self.epsilonMin) / self.epsilonDecaySteps
        ) * step

    epsilon = max(
        self.epsilonMin,
        self.epsilonMax - epsilonDelta
        )

    if epsilon > np.random.rand():
        x, y = np.random.randint(84, size = 2)

    else:
        inputs = np.expand_dims(state, 0)

        qValues = self.sess.run(
            fetches = self.network.flatten,
            feed_dict = {self.network.inputs: inputs}
            )

        maxIndex = np.argmax(qValues)
        x, y = np.unravel_index(maxIndex, SCREEN_SIZE)

    return x, y
```

Metoda *selectAction* implementira strategiju istraživanja i eksploatacije za odabir, posebno koristeći strategiju epsilon-pohlepe.

Epsilon-greedy strategija omogućuje agentu da balansira između istraživanja i iskorištavanja znanja, postupno preferirajući eksploataciju tijekom vremena kako epsilon vrijednost opada. Ova je metoda ključna za osiguravanje da agent adekvatno istražuje svoje okruženje tijekom obuke dok prelazi na više deterministične radnje kako sve dalje ide igra. Ova metoda i njeni parametri opisani su u [22].

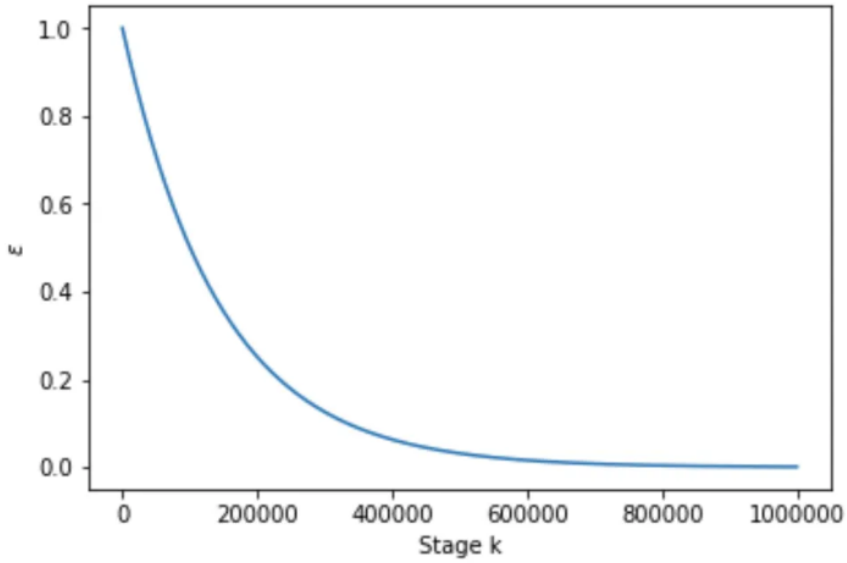
U koraku gdje $step = self.epsilonDecaySteps$, $self.epsilonDecaySteps * step$ dio se poništava odnosno agent od tog koraka najdalje koristi najmanji definirani koeficijent istraživanja, odnosno maksimalno preferira iskorištavanje već stečenoga znanja.

Prema Candidate postoje razne strategije od kojih je linearno smanjenje samo jedna. U ovom radu ostale metode neće biti dublje opisane ali će biti navedeni i prikazani grafovi ponašanja epsilona ovisno o koraku.



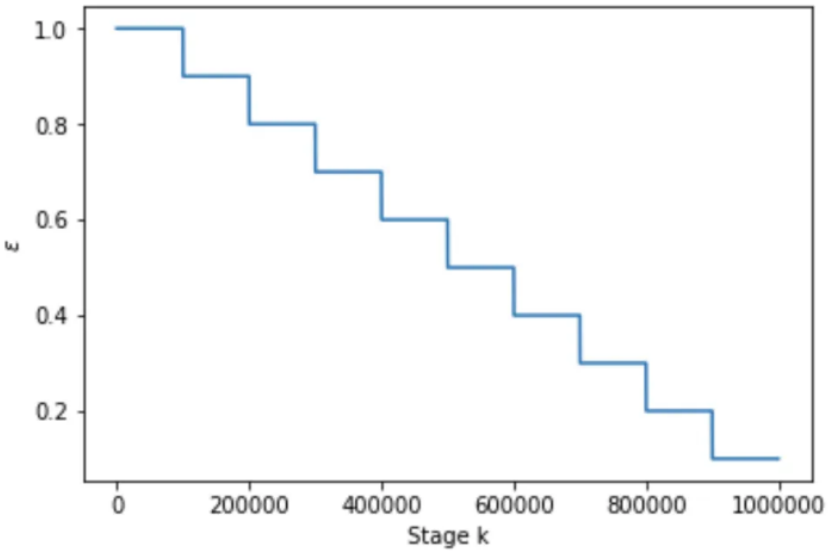
Slika 3: Linearno smanjenje epsilona [29]

Metoda sa slike 3 korištena u implementaciji agenata u ovom radu, epsilon se linearno smanjuje u svakom koraku (*step*).



Slika 4: Eksponencijalno smanjenje epsilona [29]

Sa grafa na slici 4 vidljivo je kako se epsilon sve brže smanjuje kako se povećava broj poduzetih koraka.



Slika 5: Diskretno smanjenje epsilona [29]

Zadnja metoda opisana u [29] je diskretno smanjenje epsilona. Epsilon zadržava istu vrijednost već broj koraka te se smanjuje samo u određenim trenucima.

Ovo je rješenje problema istraživanja naspram iskorištavanja prethodno opisanog u poglavlju poticano učenje.

6.3. Glavna igra

Agent koji igra glavnu igru nema sposobnosti zaključivanja na temelju piksela na mapi. Djelovanje ovog agenta usmjereno je na razumijevanje uzročno posljedičnih veza između stranih akcija koje agent može izvoditi (koje su iste kao i one koje bi ljudski igrač mogao koristiti), i stanja koje mu okruženje pruža. Valja navesti da je stanje dodatno obrađeno prije slanja mreži, na način da se agentu češće pruža povratna informacija, a ne samo na kraju igre.

6.3.1. Run.py

```
from absl import app

from pysc2.env import sc2_env
from pysc2.env.run_loop import run_loop

from BrainNew import Agent as Agent
```

Run.py ovog agenta započinje slično kao i prošlog, uz zamjenu *Brain* u *BrainNew* skriptu kako bi se koristio novi agent.

```
def main(unused_argv):
    mapList = ["Simple64"]
    agentList = [Agent()]

    with sc2_env.SC2Env(
         map_name = mapList[0],
         players = [sc2_env.Agent(sc2_env.Race.terran), sc2_env.Bot(sc2_env.Race.
             zerg, sc2_env.Difficulty.very_easy)],
         agent_interface_format = sc2_env.parse_agent_interface_format(
             feature_screen=84,feature_minimap=64),
         step_mul = 16,
         game_steps_per_episode = 0,
         visualize = True
        ) as env:
            run_loop(agentList, env)
```

U *main* funkciji se nalazi nekoliko izmjena iako je zapravo slično prethodno opisanoj skripti. Glavna razlika se nalazi u dva parametra proslijeđena metodi *SC2Env*. Prvi je *map_name* kojim se sada prosljeđuje "Simple64" što označava mapu koja je slična mapama na kojima igraju ljudski igrači, sa smanjenom veličinom. Drugi je *players* koji sada sadržava definicije dva igrača, prvi igrač koji je ujedno i naš agent igra rasu pod nazivom *Terran* dok će njegov protivnik igrati rasu *Zerg*, te se težina na kojoj će agent igrati postavlja na vrlo laku razinu (eng. *very easy*).

```
if __name__ == "__main__":
  app.run(main)
```

Kao i u prošloj skripti, ovaj dio koda provjerava izvodi li se skripta kao glavni program, i ako je, poziva prethodno definiranu i opisanu funkciju *main*.

6.3.2. Brain.py

```
import numpy as np
import tensorflow as tf
import os
import random

from collections import deque

from pysc2.agents import base_agent
from pysc2.lib import features, actions
```

Ovaj dio koda vezan za uvoze vrlo je sličan prošloj implementaciji. Koristi zamijenjene dodatke za pseudoslučajan odabir (*random*) i na drugačiji način uvozi akcije i značajke ekrana.

```
SCREEN_SIZE = [84, 84]

MAIN_NETWORK_NAME = "MainNetwork"
TRAINING_NETWORK_NAME = "TrainingNetwork"

SAVE_PATH = "./checkpoints/main.ckpt"
INDEX_PATH = SAVE_PATH + ".index"

Unit_Terran_Commandcenter = 18
Unit_Terran_Scv = 45
Unit_Terran_Supply_Depot = 19
Unit_Terran_Barracks = 21
```

Definicije korisnih konstanti se također razlikuju, sprema se na drugu putanju da bi se nesmetano moglo izvoditi učenje oba agenta.

Također se definiraju konstante koje opisuju jedinice koje će se koristiti kasnije.

```
Action_No_Action = actions.FUNCTIONS.no_op.id
Action_Select_Point = actions.FUNCTIONS.select_point.id
Action_Build_Supply_Depot = actions.FUNCTIONS.Build_SupplyDepot_screen.id
Action_Build_Barracks = actions.FUNCTIONS.Build_Barracks_screen.id
Action_Train_Marine = actions.FUNCTIONS.Train_Marine_quick.id
Action_Select_Army = actions.FUNCTIONS.select_army.id
Action_Attack_Minimap = actions.FUNCTIONS.Attack_minimap.id
```

Ovaj blok koda definira akcije koje će agent izvršavati da bi olakšali prikaz koda u kasnijim kompleksnijim dijelovima.

```
ACTION_DO_NOTHING = 'donothing'
ACTION_SELECT_SCV = 'selectscv'
ACTION_BUILD_SUPPLY_DEPOT = 'buildsupplydepot'
ACTION_BUILD_BARRACKS = 'buildbarracks'
ACTION_SELECT_BARRACKS = 'selectbarracks'
ACTION_BUILD_MARINE = 'buildmarine'
ACTION_SELECT_ARMY = 'selectarmy'
ACTION_ATTACK = 'attack'

possible_actions = [
    ACTION_DO_NOTHING,
    ACTION_SELECT_SCV,
    ACTION_BUILD_SUPPLY_DEPOT,
    ACTION_BUILD_BARRACKS,
    ACTION_SELECT_BARRACKS,
    ACTION_BUILD_MARINE,
```

```
    ACTION_SELECT_ARMY,
    ACTION_ATTACK,
]
```

Prošli isječak koda definira sve akcije koje su dane agentu na odabir. Razdvojeno je od prije definiranih stvarnih akcija uvezenih iz PySC2 dodatka zato što jedna od ovih akcija može sadržavati više koraka od samog izvršavanja akcije (npr. odabir lokacije izvršavanja akcije).

```
Player_Id = features.SCREEN_FEATURES.player_id.index
Player_Relative = features.SCREEN_FEATURES.player_relative.index
Player_Self = 1
Unit_Type = features.SCREEN_FEATURES.unit_type.index
Not_Queued = [0]
Queued = [1]

Reward_Unit_Destroyed = 0.2
Reward_Building_Destroyed = 0.5

RANDOM_IDENTIFIER = 0
NON_RANDOM_IDENTIFIER = 1
END_STEP_IDENTIFIER = 2
```

Još jedan isječak u kojem se definiraju varijable koje će se koristiti kasnije. Najbitnije za samo učenje su dvije varijable koje definiraju nagradu koju agent dobije za svaku uništenu neprijateljsku jedinicu i građevinu.

```
class Network(object):
    def __init__(self, name):

        tf.compat.v1.disable_eager_execution()

        self.learningRate = 0.001
        self.name = name

        with tf.compat.v1.variable_scope(self.name):
            self.episode = tf.Variable(
                initial_value = 0,
                trainable = False,
                name = "episode"
                )

            self.step = tf.Variable(
                initial_value = 0,
                trainable = False,
                name = "step"
                )
```

Na početku implementacije klase *Network* definiraju se *learningRate* koji se prosljeđuje algoritmu za optimizaciju i *name* koji se koristi za razlikovanje glavne mreže od one korištene

za trening.

Slijedi definicija brojača trenutne epizode i koraka, varijable koje nije moguće trenirati. Definirane su ovdje da bi ih se spremilo zajedno sa ostalim varijablama mreže, te iste učitalo prilikom pokretanja.

```
self.inputs = tf.compat.v1.placeholder(
    dtype = tf.float32,
    shape = [None, 6],
    name = "inputs"
    )


self.targets = tf.compat.v1.placeholder(
    dtype = tf.float32,
    name = "targets"
    )

self.incrementEpisode = tf.compat.v1.assign(
    ref = self.episode,
    value = self.episode + 1,
    name = "incrementEpisode"
    )


self.layer1 = tf.compat.v1.layers.dense(
    inputs = self.inputs,
    units = 64,
    activation = tf.nn.relu,
    name = "layer1"
    )

self.layer2 = tf.compat.v1.layers.dense(
    inputs = self.layer1,
    units = 32,
    activation = tf.nn.relu,
    name = "layer2"
    )

self.output = tf.compat.v1.layers.dense(
    inputs = self.layer2,
    units = 8,
    name = "output"
    )

self.flatten = tf.compat.v1.layers.flatten(self.output, name="flatten")

self.actions = tf.compat.v1.placeholder(
    dtype = tf.float32,
    name = "actions"
    )
```

```
        self.prediction = tf.reduce_sum(
            input_tensor = tf.multiply(self.flatten, self.actions),
            name = "prediction"
            )

        self.loss = tf.reduce_mean(
            input_tensor = tf.square(self.targets - self.prediction),
            name = "loss"
            )

        self.optimizer = tf.compat.v1.train.AdamOptimizer(self.learningRate).
            minimize(self.loss, global_step = self.step)

        self.saver = tf.compat.v1.train.Saver()
```

I ovom isječku koda nastavlja se i završava definicija klase *Network*. Započinje se definiranjem *self.inputs* koji očekuje 6 ulaza bez broja gomile.

Za razliku od prošlog agenta, ovdje je izabrana implementacija bez gomile zato što korišteni slojevi (*dense*) ne podržavaju dimenziju gomile. Stoga će se kasnije definirati petlja u kojoj će se više puta prosljeđivati podaci u mrežu, po jednom za svaki element gomile.

Definirana su tri gusta (eng. *dense*) sloja koja služe za izračunavanje Q vrijednosti. *Dense* slojevi su izabrani zato što su i broj varijabli stanja (6) i broj akcija (8) koje agent može izvršavati relativno mali brojevi u usporedbi sa ostalim problemima za koje se koristi duboko učenje (npr. prošli agent opisan u ovom radu, koji je imao 7056 varijabli stanja).

Ostatak implementacije prati prošli primjer zato što su to sve potrebni konstrukti za duboko Q-učenje.

```
class Agent(base_agent.BaseAgent):
    def __init__(self):
        super(Agent, self).__init__()

        self.epsilonMin = 0.01
        self.epsilonMax = 1.0
        self.epsilonDecaySteps = 10000000

        self.trainFrequency = 10000 # 250, 500, 1000
        self.batchSize = 20
        self.discountFactor = 0.95

        self.killed_unit_score_past = 0
        self.killed_building_score_past = 0
        self.previous_state = None
        self.previous_action = None

        tf.compat.v1.reset_default_graph()

        self.network = Network(MAIN_NETWORK_NAME)
```

```
        self.trainingNetwork = Network(TRAINING_NETWORK_NAME)

        self.memory = deque(maxlen = 1000000)

        self.lastState = None
        self.lastAction = None
        self.actions = list(range(len(possible_actions)))

        self.sess = tf.compat.v1.Session()
        if os.path.isfile(INDEX_PATH):
            self.network.saver.restore(self.sess, SAVE_PATH)
            self.updateNetworks()

        else:
            self.sess.run(fetches = tf.compat.v1.global_variables_initializer())
```

U ovom dijelu koda započinje implementacija klase Agent. *self.epsilonMin* i *self.epsilonMax* definirane su kao i u prošloj implementaciji, ali postoji znatna razlika u *self.epsilonDecaySteps* koja je postavljena na 10000000. Razlog tome je zato što se u glavnoj verziji igre odvija znatno veći broj koraka po jednoj rundi. Broj koraka također znatno varira između 1000 i 10000, za razliku od prošlog primjera gdje je bio gotovo uvijek oko 100.

Veličina agentove memorije je također povećana na 1000000 da bi se mreža mogla nositi sa povećanim brojem koraka.

Zadnja razlika između prošle implementacije je način na koji se agentove moguće akcije definiraju, kao lista koja se stvori od cijelih brojeva dužine jednake dužini prethodno definirane liste *possible_actions*.

U nastavku se kao i prije definira čitanje spremljenih podataka o mreži ako postoje, a u slučaju da ne, inicijalizacija nove mreže.

```
    def step(self, obs):
        self.steps += 1

        if obs.step_type == END_STEP_IDENTIFIER:
            self.sess.run(fetches = self.network.incrementEpisode)
            self.network.saver.save(self.sess, SAVE_PATH)

        player_y, player_x = (obs.observation['feature_minimap'][Player_Relative] ==
             Player_Self).nonzero()
        if player_y.any() and player_y.mean() <= 31:
            self.base_top_left = 1
        else:
            self.base_top_left = 0

        unit_type = obs.observation['feature_screen'][Unit_Type]

        depot_y, depot_x = (unit_type == Unit_Terran_Supply_Depot).nonzero()
        if depot_y.any():
            supply_depot_count = 1
```

```
else:
    supply_depot_count = 0
barracks_y, barracks_x = (unit_type == Unit_Terran_Barracks).nonzero()
if barracks_y.any():
    barracks_count = 1
else:
    barracks_count = 0
```

U ovom dijelu koda započinje implementacija funkcije *step*. Step se poziva prilikom svakog koraka interakcije agenta i okoline.

Prepoznaje se lokacija agentove baze, zato što odabrana mapa može započeti u dvije konfiguracije, sa agentovom bazom u gornjem lijevom kutu map, ili donjem desnom. Koordinate baze su potrebne da bi se u odnosu na njih mogle izgraditi sve ostale građevine.

Slijedi prepoznavanje postojećih agentovih zgrada kao što su *depot* i *barracks*, od kojih agent može izgraditi samo jednu od svake.

```
supply_limit = obs.observation['player'][4]
army_supply = obs.observation['player'][5]

killed_unit_score = obs.observation['score_cumulative'][5]
killed_building_score = obs.observation['score_cumulative'][6]

current_state = [
    supply_depot_count,
    barracks_count,
    supply_limit,
    army_supply,
    killed_unit_score,
    killed_building_score
]
```

U ovom isječku koda dohvaćaju se posljednje varijable potrebne za generiranje stanja koje se prosljeđuje agentu, te se stvara lista svih varijabla stanja *current_state*.

Dohvaćaju se vrijednosti koje okružje daje kao indikator uništenih neprijateljskih jedinica (*killed_unit_score*) i građevina (*killed_building_score*). Te se vrijednosti koriste u sljedećem isječku koda gdje se provjerava ako su vrijednosti u ovom koraku veće od onih u prethodnom, i ako jesu nagrada koja se dodjeljuje agentu povećava se za prethodno definirane vrijednosti.

```
if killed_unit_score > self.killed_unit_score_past:
    self.reward += Reward_Unit_Destroyed

if killed_building_score > self.killed_building_score_past:
    self.reward += Reward_Building_Destroyed
```

```
if (len(self.memory) > self.batchSize and self.previous_action is not None):
            states, batchActions, targets = self.getBatch()

            for i in range (0, self.batchSize):
                self.sess.run(
                    fetches = [self.network.loss, self.network.optimizer],
                    feed_dict = {
                        self.network.inputs: np.expand_dims(states[i], 0),
                        self.network.actions: batchActions[i],
                        self.network.targets: targets[i]
                        }
                    )

        if self.steps % self.trainFrequency == 0:
            self.updateNetworks()

        action_index = self.chooseAction(current_state)
        chosen_action = possible_actions[action_index]

        self.killed_unit_score_past  = killed_unit_score
        self.killed_building_score_past = killed_building_score
        self.previous_state = current_state
        self.previous_action = action_index


        if self.lastState is not None:
            self.memory.append((self.lastState, self.lastAction, self.reward,
                current_state))

        self.lastState = current_state
        self.lastAction = action_index
```

Ovaj dio funkcije *step* zadužen je za učenje. Za razliku od prošlog agenta gdje je mreža sama bila zadužena za procesiranje gomila, ovdje je sam agent odnosno ovaj dio funkcije *step* zadužen za to. Dohvaća se gomila te se stvara petlja koja se ponavlja za svaki element gomile te se ponavlja korak prosljeđivanja mreži.

Slijedi poziv funkcije za odabir akcije (*chooseAction*) koja zapravo odabire indeks akcije, te se potom taj indeks koristi da bi se dohvatila stvarna akcija koju će agent izvršavati.

Na kraju se zapisuju trenutne varijabli kao prošle i zapisivanje nove gomile u agentovo pamćenje.

```
if chosen_action == ACTION_DO_NOTHING:
            return actions.FunctionCall(Action_No_Action, [])

        elif chosen_action == ACTION_SELECT_SCV:
            unit_type = obs.observation['feature_screen'][Unit_Type]
            unit_y, unit_x = (unit_type == Unit_Terran_Scv).nonzero()

            if unit_y.any():
```

```
            i = random.randint(0, len(unit_y) - 1)
            target = [unit_x[i], unit_y[i]]

            return actions.FunctionCall(Action_Select_Point, [Not_Queued, target
                ])

    elif chosen_action == ACTION_BUILD_SUPPLY_DEPOT:
        if Action_Build_Supply_Depot in obs.observation['available_actions']:
            unit_type = obs.observation['feature_screen'][Unit_Type]
            unit_y, unit_x = (unit_type == Unit_Terran_Commandcenter).nonzero()

            if unit_y.any():
                target = self.transformLocation(int(unit_x.mean()), 0, int(
                    unit_y.mean()), 20)

                return actions.FunctionCall(Action_Build_Supply_Depot, [
                    Not_Queued, target])

    elif chosen_action == ACTION_BUILD_BARRACKS:
        if Action_Build_Barracks in obs.observation['available_actions']:
            unit_type = obs.observation['feature_screen'][Unit_Type]
            unit_y, unit_x = (unit_type == Unit_Terran_Commandcenter).nonzero()

            if unit_y.any():
                target = self.transformLocation(int(unit_x.mean()), 20, int(
                    unit_y.mean()), 0)

                return actions.FunctionCall(Action_Build_Barracks, [Not_Queued,
                    target])

    elif chosen_action == ACTION_SELECT_BARRACKS:
        unit_type = obs.observation['feature_screen'][Unit_Type]
        unit_y, unit_x = (unit_type == Unit_Terran_Barracks).nonzero()

        if unit_y.any():
            target = [int(unit_x.mean()), int(unit_y.mean())]

            return actions.FunctionCall(Action_Select_Point, [Not_Queued, target
                ])

    elif chosen_action == ACTION_BUILD_MARINE:
        if Action_Train_Marine in obs.observation['available_actions']:
            return actions.FunctionCall(Action_Train_Marine, [Queued])

    elif chosen_action == ACTION_SELECT_ARMY:
        if Action_Select_Army in obs.observation['available_actions']:
            return actions.FunctionCall(Action_Select_Army, [Not_Queued])

    elif chosen_action == ACTION_ATTACK:
        if Action_Attack_Minimap in obs.observation["available_actions"]:
            if self.base_top_left:
                return actions.FunctionCall(Action_Attack_Minimap, [Not_Queued,
                    [39, 45]])
```

```
                return actions.FunctionCall(Action_Attack_Minimap, [Not_Queued, [21,
                    24]])

        return actions.FunctionCall(Action_No_Action, [])
```

Ovaj kod zadužen je za izvršavanje akcije koja je prethodno odabrana. Svaki od *if* i *elif* elemenata služi za obradu jedne od odabranih akcija. U slučaju da nije odabrana niti jedna od prethodno definiranih akcija (što je zapravo samo osiguranje jer se to ne bi smjelo dogoditi) agent izvršava akciju koja ne radi ništa odnosno propušta šansu za izvršavanje radnje.

Akcije označavanja vojske i treniranja marinaca su jednostavne, samo pozivanje prethodno definirane akcije, dok su druge kao izgradnja zgrada i napad kompleksne zato što treba odlučiti gdje postaviti nove građevine.

U ovom primjeru sve je pojednostavljeno da bi se primjer uopće mogao izvoditi samo sa jednom mrežom prihvatljive (u ovom slučaju izvedive) veličine. Kreiranje više mreža omogućilo bi arhitekturu gdje je svaka mreža zadužena za jedan dio odlučivanja, na primjer lokacije gdje izgraditi građevine ili promatranje gdje se nalaze neprijateljske jedinice. Ovakva veća arhitektura bila znatno intenzivnija i zahtijevala bi veću procesorsku moć. Iako je ovo godinama bio način na koji su se razvijali najbolji agenti koji igraju StarCraft 2, AlphaStar u svojem radu opisuju kako su uspjeli sve ove različite funkcionalnosti implementirati kao jedinstvenu neuralnu mrežu [30].

```
    def transformLocation(self, x, x_distance, y, y_distance):
        if not self.base_top_left:
            return [x - x_distance, y - y_distance]
        return [x + x_distance, y + y_distance]
```

Ova funkcija koristi se za osiguravanja da se nove građevine ne izgrade izvan mape. Kao referentne biraju se koordinate neke već postojeće građevine te se u odnosu na nju daju koordinate nove građevine uzimajući u obzir lokaciju agentove baze.

```
def getBatch(self):
        indices = np.random.choice(
            np.arange(len(self.memory)),
            size = self.batchSize,
            replace = False
            )

        batch = [self.memory[i] for i in indices]

        states = np.array(
            [each[0] for each in batch]
            )
        actions = np.array(
            [each[1] for each in batch]
            )
```

```
        rewards = np.array(
            [each[2] for each in batch]
            )
        nextStates = np.array(
            [each[3] for each in batch]
            )

        predictedOutputs = self.sess.run(
            fetches = self.trainingNetwork.output,
            feed_dict = {self.trainingNetwork.inputs: nextStates}
            )

        targets = [rewards[i] + self.discountFactor * np.max(predictedOutputs[i])
            for i in range(self.batchSize)]

        return states, actions, targets
```

Kao i u prethodnom agentu, ova funkcija je zadužena za realizaciju gomile. Osim dohvaćanja podataka iz agentovog sjećanja, zadužena je za izračunavanje ciljeva koji će se proslijediti mreži prilikom učenja.

```
    def chooseAction(self, state):
        step = self.network.step.eval(session = self.sess)
        epsilonDelta = ((self.epsilonMax - self.epsilonMin) / self.epsilonDecaySteps
            ) * step

        epsilon = max(
            self.epsilonMin,
            self.epsilonMax - epsilonDelta
            )

        if epsilon > np.random.rand():
            return np.random.choice(self.actions)

        else:
            inputs = np.expand_dims(state, 0)

            qValues = self.sess.run(
                fetches = self.network.flatten,
                feed_dict = {self.network.inputs: inputs}
                )

            return np.argmax(qValues)
```

Jedina razlika u implementacije funkcije *chooseAction* naspram prijašnjeg agenta je što funkcija daje kao povratna vrijednost (eng. *return value*). Umjesto izračunavanja koordinata napada i vraćanja istih, ova funkcija vraća indeks akcije koju agent treba izvršiti.

```
    def reset(self):
```

```
        self.episodes += 1
        self.lastState = None
        self.lastAction = None
        self.reward = 0

        episode = self.network.episode.eval(session = self.sess)
        step = self.network.step.eval(session = self.sess)
        print("Episode␣", episode)
        print("Step␣", step)
```

Funkcija *reset* zadužena povećanja broja epizode, resetiranje vrijednosti varijabli na početno stanje i ispisivanje broja epizode i koraka u konzolu.

```
    def updateNetworks(self):
        mainNetworkVariables = tf.compat.v1.get_collection(
            key = tf.compat.v1.GraphKeys.TRAINABLE_VARIABLES,
            scope = MAIN_NETWORK_NAME
            )
        trainingNetworkVariables = tf.compat.v1.get_collection(
            key = tf.compat.v1.GraphKeys.TRAINABLE_VARIABLES,
            scope = TRAINING_NETWORK_NAME
            )

        variablesToUpdate = []
        for mainVariable, trainingVariable in zip(mainNetworkVariables,
            trainingNetworkVariables):
            variablesToUpdate.append(trainingVariable.assign(mainVariable))

        self.sess.run(fetches = variablesToUpdate)
```

Ova funkcija izvršava se samo svakih n prethodno definiranih koraka u svrhu boljeg zaključivanja, odnosno izračunavanja Q vrijednosti. Prepisuje sve vrijednosti iz glavne mreže u mrežu za trening.

7. Ponašanje agenta

Svaki od agenata naučio je rješavati problem na svoj način, u nastavku će biti objašnjeno ponašanje koje rezultira kodom koji je bio prikazan u prošlom poglavlju.

7.1. Agent koji igra mini igru

Agent koji igra mini igru uspješno je savladao i riješio obje mini igri na kojima je testiran. U većini testiranja uspije naučiti kako se igra mini igra u manje od 70 epizoda. U prosjeku pokazuje znakove razumijevanja gdje treba poslati jedinice negdje između 10 i 20 epizoda, ali u tom trenutku je još uvijek u fazi istraživanja. Konfiguriran je tako da istražuje sve do otprilike epizode 100 tako što je broj koraka za smanjenje epsilona postavljeno na 1000, a svaka epizoda ima otprilike 100 koraka (+-10 ovisno o brzini izvršavanja).

7.2. Agent koji igra glavnu igru

Iako ovaj agent može pobijediti protivnika, treba mu veliki broj epizoda da to nauči. To je razumljivo s obzirom da glavna igra poprima jako veliki broj mogućih stanja i zahtjeva od agenta da nauči dugotrajnu strategiju, znatno kompleksniju od one koju agent koji igra mini igru mora naučiti. Buduća poboljšanja bila bi uvođenje različitih dubokih neuronskih mreža za procesiranje različitih zadataka, uz istovremeno povećanje akcija koje agent ima na raspolaganju, iako bi to zahtjevalo znatno veću procesorsku moć.

Već je u prijašnjem pokušaju implementacije ovog agenta implementirana mreža koju nije mogao podržati isti stroj koji je kasnije korišten za implementaciju opisanu u ovom radu.

Osim toga, najveća prepreka agentu u učenju je situacija u kojoj agent prilikom istraživanja pošalje sve SCV jedinice (koje su zadužene za skupljanje resursa potrebnih za izgradnju i treniranje daljnjih jedinica), što ga onemogućuje u daljnjem igranju igre sve dok ga neprijatelj ne pobjedi.

8. Zaključak

Najveća prednost dubokog poticanog učenja je mogućnost procesiranja znatno veće količine podataka i samostalnog učenje direktno iz sirovih podataka kakve bi i ljudski igrači imali, uz mogućnosti koju donosi poticano učenje, a to je da agent ne mora imati nikakvo saznanje o okruženju s kojim je u interakciji. Agenti uče direktno i isključivo iz interakcije sa svojom okolinom te stvaraju svoju predodžbu o svijetu korak po korak kroz interakciju s istim.

Oba agenta uspjela su izvršiti svoj zadatak, doći do pobjede u svojim verzijama igre. Agent koji je imao za zadatak naučiti iz kompleksnije reprezentacije okruženja uz jednostavniju verziju igre pokazao se znatno efikasniji u svom učenju od agenta koji je igrao pravu verziju igre, sa pojednostavljenim prikazom. Agenti su sposobni nositi se sa promjenama u okruženju, u oba slučaja igra mijenja privremene ciljeve koje agent mora postići da bi došao do konačne pobjede.

Nedostatak implementacije ove metode kao jedne mreže je taj da agent ima problema sa otkrivanjem dugotrajne strategije. Agent od kojeg se zahtjeva dugotrajna strategija preferira kratkotrajne nagrade i teško dolazi do situacije gdje uspijeva spoznati dugotrajnu strategiju. Do ovoga dolazi usprkos raznim metodama korištenim u svrhu smanjenja učenja politike koja donosi kratkotrajnu nagradu a nije optimalna, bez njih pitanje je bi li ovaj agent ikad uspio pobijediti.

Ovaj proces bio bi još duži da su agentu na raspolaganje dane sve akcije koje ljudski igrač ima. Ovo bi se moglo usavršiti uvođenjem više različitih specijaliziranih dubokih neuronskih mreža od kojih je svaka zadužena za samo jedan pod dio učenja, no za to su potrebni znatni resursi.

Duboko poticano učenje, kao kombinacija dubokog i poticanog učenja, pokazalo se kao vrlo efikasna metoda dubokog strojnog učenja. Oba agenta uspjela su doći do pobjede kroz različite implementacije dubokih neuronskih mreža, potpuno samostalno stječući saznanja o okruženju s kojim su u interakciji, što je i srž dubokog poticanog učenja.

Popis literature

[1] *BeginnersGuide/Overview - Python Wiki*. adresa: https://wiki.python.org/moin/BeginnersGuide/Overview (pogledano 4. 9. 2020.).

[2] *Introduction to TensorFlow*. adresa: https://www.tensorflow.org/learn (pogledano 6. 1. 2024.).

[3] M. Abadi, P. Barham, J. Chen i dr., „TensorFlow: A system for large-scale machine learning,"

[4] O. Vinyals, T. Ewalds, S. Bartunov i dr., „Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[5] S. Russell i P. Norvig, „Artificial intelligence: a modern approach," 2002.

[6] D. Lynton Poole, *Computational Intelligence: A Logical Approach*, ISBN: 978-0-19-510270-3.

[7] *BBC The Next Big Thing - Artificial Intelligence*, rujan 2009. adresa: https://web.archive.org/web/20090925043908/http://www.open2.net/nextbigthing/ai/ai_in_depth/in_depth.htm (pogledano 17. 9. 2020.).

[8] H. Hodson, „DeepMind and Google: the battle to control artificial intelligence," *The Economist*, ISSN: 0013-0613. adresa: https://www.economist.com/1843/2019/03/01/deepmind-and-google-the-battle-to-control-artificial-intelligence (pogledano 28. 8. 2020.).

[9] R. E. Neapolitan i X. Jiang, *Artificial intelligence: With an introduction to machine learning*. CRC Press, 2018.

[10] „AI vs machine learning - difference between artificial intelligence and ML - AWS," Amazon Web Services, Inc. (), adresa: https://aws.amazon.com/compare/the-difference-between-artificial-intelligence-and-machine-learning/ (pogledano 29. 1. 2024.).

[11] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[12] Y. LeCun, Y. Bengio i G. Hinton, „Deep learning," *nature*, sv. 521, br. 7553, str. 436–444, 2015.

[13] R. S. Sutton i A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[14] C. Shyalika, *A Beginners Guide to Q-Learning*, en, studeni 2019. adresa: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c (pogledano 7. 9. 2020.).

[15] Q. Hu i W. Yue, *Markov decision processes with their applications*. Springer Science & Business Media, 2007., sv. 14.

[16] C. J. Watkins i P. Dayan, „Q-learning," *Machine learning*, sv. 8, br. 3-4, str. 279–292, 1992.

[17] Morvan, *Q-learning (Reinforcement Learning)*, en. adresa: https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/tabular-q1/ (pogledano 8. 9. 2020.).

[18] L. Deng, D. Yu i dr., „Deep learning: methods and applications," *Foundations and trends® in signal processing*, sv. 7, br. 3–4, str. 197–387, 2014.

[19] J. Schmidhuber, „Deep learning in neural networks: An overview," *Neural networks*, sv. 61, str. 85–117, 2015.

[20] M. Vasilescu i D. Terzopoulos, „Multilinear (tensor) image synthesis, analysis, and recognition [exploratory DSP]," *IEEE Signal Processing Magazine*, sv. 24, br. 6, str. 118–123, studeni 2007., ISSN: 1053-5888. DOI: 10.1109/MSP.2007.906024. adresa: http://ieeexplore.ieee.org/document/4387945/ (pogledano 8. 1. 2024.).

[21] „Introduction to Scalars Vectors Matrices and Tensors using Python/Numpy examples and drawings." (), adresa: https://hadrienj.github.io/posts/Deep-Learning-Book-Series-2.1-Scalars-Vectors-Matrices-and-Tensors/ (pogledano 9. 1. 2024.).

[22] V. Mnih, K. Kavukcuoglu, D. Silver i dr., „Human-level control through deep reinforcement learning," *Nature*, sv. 518, str. 529–533, 2015. adresa: https://api.semanticscholar.org/CorpusID:205242740.

[23] V. Mnih, A. P. Badia, M. Mirza i dr., „Asynchronous Methods for Deep Reinforcement Learning," *CoRR*, sv. abs/1602.01783, 2016. arXiv: 1602.01783. adresa: http://arxiv.org/abs/1602.01783.

[24] *StarCraft II: Wings of Liberty*, en, Page Version ID: 976499352, rujan 2020. adresa: https://en.wikipedia.org/w/index.php?title=StarCraft_II:_Wings_of_Liberty&oldid=976499352 (pogledano 6. 9. 2020.).

[25] L. H. Hinic, „Q-UČENJE KAO ALGORITAM METODE POJAČANOG UČENJA I NJEGOVA PRIMJENA,"

[26] J. Brownlee. „Why one-hot encode data in machine learning?" MachineLearningMastery.com. (27. srpnja 2017.), adresa: https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/ (pogledano 14. 1. 2024.).

[27] I. AI. „Answer to "What does 1x1 convolution mean in a neural network?"" Cross Validated. (7. veljače 2016.), adresa: https://stats.stackexchange.com/a/194450 (pogledano 14. 1. 2024.).

[28] V. Mnih, K. Kavukcuoglu, D. Silver i dr., „Playing Atari with Deep Reinforcement Learning," *CoRR*, sv. abs/1312.5602, 2013. arXiv: 1312.5602. adresa: http://arxiv.org/abs/1312.5602.

[29] C. M. B. Candidate Ph D. „Strategies for decaying epsilon in epsilon-greedy,” Medium. (20. rujna 2022.), adresa: https://medium.com/@CalebMBowyer/strategies-for-decaying-epsilon-in-epsilon-greedy-9b500ad9171d (pogledano 14. 1. 2024.).

[30] „AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning,” Google DeepMind. (30. listopada 2019.), adresa: https://deepmind.google/discover/blog/alphastar-grandmaster-level-in-starcraft-ii-using-multi-agent-reinforcement-learning/ (pogledano 14. 1. 2024.).

Popis slika