UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET

PRIMENA SARSA ALGORITMA ZA OBUČAVANJE VEŠTAČKOG AGENTA ZA IGRANJE RETRO IGARA

MASTER RAD

Studijski program: Računarstvo i informatika

Kandidat:

Filip Stamenković, br. ind. 342

Mentor:

doc. dr Aleksandar Milosavljević

Niš, decembar 2017. god.

Master rad
Primena Sarsa algoritma za obučavanje veštačkog agenta za igranje retro igara

Zadatak: *Proučiti obučavanje pojačavanjem (Reinforcement Learning) i njegove metode, sa akcentom na Sarsa algoritmu kao glavnog predstavnika on-policy algoritma. Uraditi pregled postojećih okruženja za razvoj i testiranje veštačkih agenata u okruženju retro video igara. Implementirati veštačkog agenta koji će u nepoznatom okruženju jednostavne video igre pokazati sposobnost da nauči polisu, tako da maksimizuje nagradu i ostvari bolji rezultat od prosečnog čoveka.*

Student:

Filip Stamenković, br. ind. 342

Datum prijave rada:
_____

Datum predaje rada:
_____

Datum odbrane rada:
_____

Komisija za odbranu:

_____
1. doc. dr Aleksandar Milosavljević

_____
2. prof. dr Dragan Stojanović

_____
3. doc. dr Bratislav Predić

PRIMENA SARSA ALGORITMA ZA OBUČAVANJE VEŠTAČKOG AGENTA ZA IGRANJE RETRO IGRICA

SAŽETAK

Sticanje znanja, tj. učenje, na osnovu interakcije sa okolinom je koncept koji je sveprisutan u svetu. Dok obučavanje veštačkih agenata, obično (kod nadgledanog učenja) zahteva veliki broj primera sa obeleženim podacima ili (kod algoritma planiranja) uvođenje domenskog znanja. Ovakvi agenti uglavnom postižu loše performanse u okruženjima koja su za njih nepoznata i ne mogu se adekvatno prilagoditi novonastalim nepoznatim situacijama. *Reinforcement Learning* je vrsta mašinskog učenja, koja najčešće, znanje stiče upravo iz nepoznatih okruženja, samo na osnovu interakcije sa okolinom. Ovaj rad se bavi primenom *reinforcement learning* pristupa na obučavanje agenta za igranje retro video igara. Retro video igre su izabrane zbog njihove jednostavnosti. U radu su izložene glavne familije RL algoritma za rešavanje problema, sa akcentom na algoritam Sarsa. Prikazane su pogodnosti i mane korišćenja aproksimacione funkcije u odnosu na tabelarni pristup.

Za implementaciju agenta koji igra retro igrice, razvijena su razna okruženja. Implementirani veštački agenti za video igre *Breakout* i *Pong*, u ovom radu, se izvršavaju na okruženju *Arcade Learning Environment*. Ovo okruženje predstavlja osnovu za mnoga druga okruženja, jedno od tih je i *Retro Learning Environment* na kome se izvršava treći agent, koji igra *Flappy Bird* igru. Glavni cilj ovog rada je pokazati kako veštački agent, implemetiran pomoću Sarsa algoritma, može postići bolje rezultate u odnosu na prosečnog čoveka.

Na kraju ovog rada predstavljeni su rezultati koje agenti ostvaruju. Na osnovu tih rezultata se može videti da agenti relativno brzo ostvaruju maksimalne rezultate u svakoj igri, tj. veći rezultat od prosečnog igrača.

Ključne reči: Reinforcement learning, Sarsa, aproksimaciona funkcija, mašinsko učenje, retro igre, Arcade Learning Environment, Breakout, Flappy Bird.

APPLYING SARSA ALGORITHM FOR TRAINING ARTIFICAL AGENT TO PLAY RETRO GAMES

ABSTRACT

Gaining knowledge, or learning, based on interaction with environment is a concept that is present in the world. While training artificial agents, usually (supervised learning) demands huge amount of training data, or introducing domain knowledge (planning algorithms). This kind of agent usually performs poorly in unknown environments and it can't adapt in new unknown situations. Reinforcement Learning is type of machine learning, which mostly, gains knowledge from unknown environments, only by interacting with them. This paper is based on applications of reinforcement learning approaches for training agents to play retro video games. Retro video games are chosen because of their simplicity. In this paper main RL algorithms are presented, with focus on Sarsa algorithm. Pros and cons of using approximation function instead tabular approach are displayed.

For implementing agent to play retro video games, a lot of learning environments are developed. Agents that are playing Breakout and Pong game, in this paper, are implemented on Arace Learning Environment. This environment is base for many other environments, one of them is Retro Learning Environment, third agent that plays Flappy Bird is developed on this one. Main goal of this paper is to show how artificial agent, implemented with Sarsa algorithm, can achive better result than average human.

At the end of this paper results that agents achive are displayed. Based on these results it is clear that agents relatively quickly achive maximal result in each game, which is better result than average human can achive.

Keywords: Reinforcement learning, Sarsa, linear approximator, machine learning, video games, Arcade Learning Environment, Breakout.

SPISAK KORIŠĆENIH SKRAĆENICA I OZNAKA

SKRAĆENICE

RL	***R**einforcement **L**earning*
MDP	***M**arkov Decision **P**roccess*
MC	**M**onte **C**arlo
TD	**T**emporal **D**ifference
ALE	**A**rcade **L**earning **E**nvironment
NES	**N**intendo **E**ntertainment **S**ystem
SNES	**S**uper **N**intendo **E**ntertainment **S**ystem

OZNAKE

S_t	Stanje u kome se agent nalazi u trenutku **t**
A_t	Akcija koju agent izvršava u trenutku **t**
R_t	Nagrada u trenutku **t**
Gt	Očekivana nagrada koju agent treba da dobije od trenutka **t**
γ	*Discount* faktor, definiše koliko agent vrednuje buduće nagrade u odnosu na trenutnu
$V(S_t)$	Vrednost funkcije vrednosti stanja za stanje S_t
$Q(S_t, A_t)$	Vrednost funkcije vrednosti akcije za stanje S_t i akciju A_t
$v_\pi(s)$	Funkcija vrednosti stanja koja prati polisu π
$q_\pi(s, a)$	Funkcija vrednosti akcije koja prati polisu p
$v_*(s)$	Optimalna funkcija vrednosti stanja
$q_*(s, a)$	Optimalna funkcija vrednosti akcije
α	Faktor koji definiše u kojoj meri nove modifikacije utiču na vrednost
w	Vektor karakteristika.
$\hat{v}(S, w)$	Aproksimaciona funkcija vrednosti stanja za stanje S i vektor karakteristika w

Sadržaj

1 UVOD

Poslednjih nekoliko decenija interesovanje za veštačku inteligenciju je naglo poraslo. Kako su računari postajali moćniji i brži (brži CPU, GPU, veći RAM) tako su developeri konačno imali alat da naprave uspešnog veštačkog agenta, čije se „učenje" moglo obaviti u relativno kratkom vremenskom roku. Pomoću „jačeg" hardware-a, neuronske mreže su brže konvergirale ka optimalnom rešenju, što je dovelo do velikog napretka kod nadgledanog učenja. Ista logika važi i za ***R**einforcement **L**earning* (obučavanje pojačavanjem), samo što je RL pravi „bum" doživeo dosta kasnije u odnosu na nadgledano učenje. Iako su RL algoritmi, koji se danas koriste (*Q-learning*, Sarsa, itd.), osmišljeni krajem prošlog veka, tek od 2013. su stekli popularnost uspehom firme DeepMind. Oni su napravili veštačkog agenta koji je igrao Atari 2600 igre bolje nego prosečan čovek. Nakon njihovog napretka grana RL je postala veoma popularna.

Prvo pitanje koje se postavlja, zašto RL? Način kako agent uči, na osnovu nagrada i kazna, je prema mom mišljenju, približan načinu kako bebe uče. RL agent, kao i beba, odjednom biva bačen u nepoznato okruženje za koje ne zna pravila, samo zna svoje akcije koje može da odradi. Svako stanje u kome se u početku nađe mu je totalno nepoznato, nikad nije iskusio $_{\text{tako}}$ nešto, ali izvršavajući akcije dobija nagradu ili kaznu. Cilj agenta je da nauči iz tog iskustva, iz tih nagrada i kazna, da kada se sledeći put nađe u istoj (ili sličnoj) situaciji pokupi što više nagrada može, a da izbegne kazne. Ova vrsta učenja je skroz drugačija od npr. nadgledanog učenja. Kod nadgledanog učenja postoji neki supervizor koji daje podatke agentu i „govori" na osnovu ulaznih podataka šta se treba naći na ulazu, dok kod RL-a supervizor ne postoji, postoje samo 3 osnovne stvari: **stanje** (u kom se agent nalazi), **akcija** (koju agent izvršava) i **nagrada** (koju agent dobija kada izvrši akciju).

Većina objavljenih radova koji su se bavili RL-om se odnosila na veštačkog agenta koji igra igre. Zašto igre?

Igre, naročito stare retro igre, sa starih konzola Atari 2600, NES, SNES itd., su odličan *benchmark* za testiranje algoritma veštačkog agenta. Stare retro igrice su prilično jednostavne u odnosu na današnje, tako da ako je za neke igre „lako" napraviti veštačkog agenta, to je za stare retro igrice. Verujem da je delom i nostalgija odigrala ulogu u oživljavanju starih igrica kao što su Pong, Breakout, Super Mario itd.

Ovaj rad se bavi teorijskom podlogom za RL algoritme u cilju savladavanja potrebnih veština za implementaciju algoritma na osnovu koga veštački agent može uspešno igrati retro igre.

U drugom poglavlju ovog rada dat je pregled literatura, koje opisuju dostignuća u ovoj oblasti.

Treće poglavlje daje osnovnu teorijsku podlogu RL-a, pa su ukratko opisane osnovne komponente koje RL može da sadrži, kao što su: funkcija vrednosti, model, polisa. Takođe su opisani neki od osnovnih problema sa kojima se suočava svaki agent: istraživanje vs iskorišćavanje, predstavljanje **nagrada** i **kazna**.

U četvrtom poglavlju su predstavljeni algoritmi na osnovu kojih veštački agent pokušava da nauči optimalnu polisu, kako bi sakupio što veću nagradu. Prva 3 potpoglavlja se odnose na algoritme koji se koriste za probleme čiji je broj stanja dovoljno mali da se vrednosti tih stanja mogu čuvati u memoriji. Poglavlje 4.4 prikazuje algoritme za probleme čiji je broj stanja prevelik, pa se uvodi priča o aproksimacionim funkcijama. Detaljnije je obrađena linearna aproksimaciona funkcija.

Peto poglavlje opisuje *open-source* okruženja pomoću kojih se mogu lagano testirati različiti RL algoritmi. Sama okruženja imaju podršku za veliki broj igrica, ali i pored toga moguće je implementirati podršku za novu igru, što će biti i prikazano.

Šesto poglavlje se bavi konkretnom implementacijom veštačkog agenta za igranje retro igara. Pored implementacije prikazani su i rezultati 3 različita agenta koji koriste Sarsa algoritam. Glavni fokus je na agentu koji koristi algoritam Sarsa „u n koraka“ sa linearnom aproksimacionom funkcijom za igranje Breakout igre. Isti agent (sa par sitnih razlika) je iskorišćen za igranje Pong igre. Ova dva agenta rade pomoću ALE okruženja. Treći agent čiji će razultati biti prikazani je agent koji je implementiran na predmetu Soft Computing, a to je tabelarni Sarsa algoritam „u n koraka“ koji igra igru Flappy Bird. Kako ova igra nije „retro“, portovana je za konzolu *Nintendo Enterntainment System*, a korišćeno okruženje je modifikovani *Retro Learning Environment* (koji se bazira na ALE-u).

1.1 Pregled literature

Kao što je već rečeno u uvodu, pravi „bum“ u ovoj oblasti je napravila firma DeepMind. Zbog njihove uspešnosti, 2014. godine ih je kupio Google.

DeepMind ima svoj YouTube kanal, na kome se mogu naći predavanja Davida Silvera, člana te firme i profesora na Stanford univerzitetu. Predavanja [1, 4, 5, 6, 7] se bave osnovama RL-a, i prilično su bazirana na osnovu knjige koje je napisao Sutton, „Uvod u *Reinforcement Learning*“ [2]. Ta knjiga se može smatrati biblijom RL-a, bez znanja iz te knjige veoma je teško napraviti agenta koji rešava RL probleme.

DeepMind je koristio *Q-learning* za rešavanje RL problema, a za *funkcion approximator* je korišćena konvoluciona neuronska mreža, taj algoritam su nazvali „*Deep Q-Learning“* [8].

Sa Stanford univerziteta su pokušali da, iz niza piksela pomoću OpenCV bibilioteke, izvuku razne karakteristike iz slike i te karakteristike kasnije da iskoriste pomoću linearnog aproksimatora funkcije zajedno sa *Q-learning* i Sarsa algoritmom kako bi agent uspešno igrao Atari 2600 igre. U zaključku su naveli da njihov algoritam nije uspeo dovoljno dobro da generalizuje, a i imali su problem da brzo u realnom vremenu izvuku karakteristike sa slike [11].

Druga grupa sa Stanford univerziteta se isto fokusirala na retro igre sa linearnim aproksimator, gde je glavni fokus bila Breakout igra. Testirani su *Q-learning* i Sarsa algoritam. Rezultati njihovog istraživanja su prikazani u radu [12]. U njihovom radu je rečeno da RL algoritmi sa linearnim aproksimatorom mnogo brže konvergiraju ka optimalnoj polisi u odnosu na algoritme koji koriste neuronsku mrežu. Objašnjeno je i kako odabir odgovarajućih karakteristika (*feature*-a) može uticati na učenje agenta, ovaj konkretan problem sam i ja uočio tokom implementacija agenta.

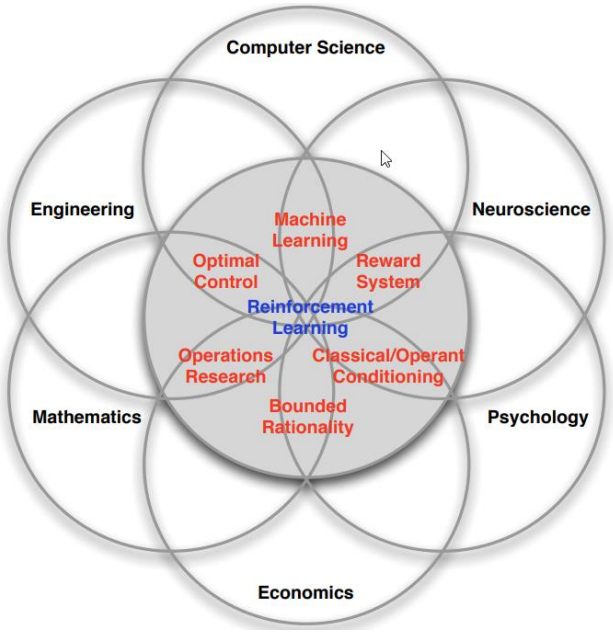
Kako se sve više istraživalo u oblasti RL-a, tako su se pojavila i okruženja za testiranje raznih RL algoritama. Najveći doprinos je donelo okruženje **A**rcade **L**earning **E**nvironment [9]. ALE je bio toliko uspešan da velike firme poput DeepMind i OpenAI nisu imali potrebe da razvijaju svoja rešenja nego su ALE proširili.

Tako je DeepMind napravio Xitari:
https://github.com/deepmind/xitari
A OpenAI je napravio Gym, oba se baziraju na ALE-u:
https://github.com/openai/gym

Izraelski Institut Tehnologije takođe je nadogradio ALE i napravio je **R**etro **L**earning **E**nvironment. Nadogradnja RLE-a je bila ta što je dodata podrška za SNES igre. Kad su dodali podršku za SNES igre testirali su efikasnost *Deep Q-Learning* algoritma nad tim igrama. Rezultate svog rada su objavili u radu [10].

2 OBUČAVANJE POJAČAVANJEM

Da bi objasnili šta je *Reinforcement Learning* (obučavanje pojačavanjem), daćemo pregled raznih naučnih oblasti i delove tih oblasti koji su zaduženi za donošenje odluka.



Slika 2.1 Donošenje odluka u raznim naučnim oblastima. Izvor: [1]

Na osnovu Venovog dijagrama koji je prikazan na slici 2.1, može se videti da svaka naučna oblast ima granu koja se bavi istim problemom, a to je problem donošenja odluka. U računarstvu ta grana je *Machine Learning*, u matematici Operaciona Istraživanja itd.

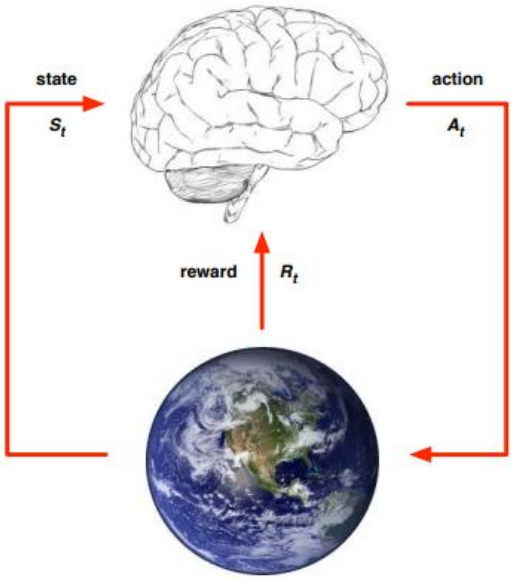
Veliki doprinos RL algoritmima je dala i neuronauka. U nekoliko poslednjih decenija mnoga istraživanja u neuronauci su se bavila nagradnim sistemom, koji je osnova za RL. Ivan Pavlov je koristio nagradni sistem psa tako što je psa nagrađivao hranom svaki put kada pas čuje zvono. James Olds i Peter Milner su 1954. nagradni sistem miša stimulisali elektrošokovima kako bi naučili miša da nađe izlaz iz lavirinta. Iste koncepte (nagrade i kazne) koriste i RL algoritmi kako bi se agent naučio da donosi bolje odluke.

Svaki RL algoritam ima 3 zajednička elementa:

- Stanje S_t
- Akcija A_t
- Nagrada R_t

U RL „svetu“, postoje agent i okruženje. Cilj agenta je da tokom vremena maksimizuje nagradu koju dobija od okruženja. Pored nagrade koju okruženje pruža agentu, okruženje takođe pruža agentu i informaciju o stanju u kom se agent nalazi. Na osnovu tog stanja agent bira svoju narednu akciju. U trenutku t, agent izvršava akciju A_t, prelazi u stanje S_t i dobija nagradu R_t[1]. Ilustracija je data na slici 2.2.

[1] Zapravo nagrada R_t se dobija nakon stanja t, a pre stanja t+1. Neke literature za nagradu koja sledi nakon akcije A_t koriste notaciju R_t, a neke literature koriste R_{t+1}. Kakvu god notaciju koristili, logika je ista, tako da u ovom radu će biti notacija kao što je prikazana. Tj. nakon akcija A_t sledi nagrada R_t.



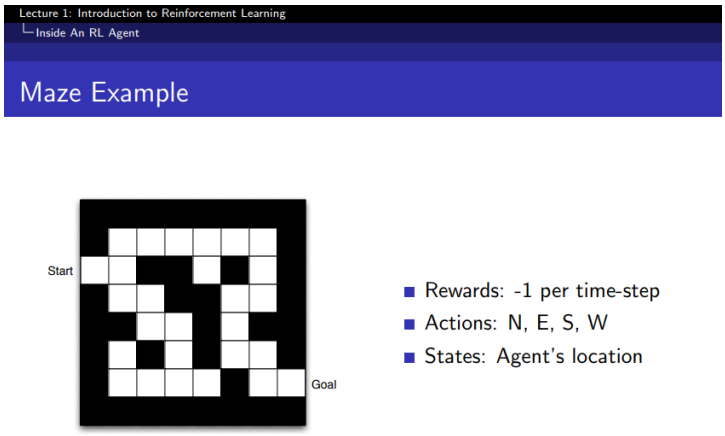
Slika 2.2 RL "svet"

2.1 RL komponente

Jedan od najvećih RL problema je kako u svakom stanju doneti najbolju odluku. Odnosno, problem agenta je kako za stanje S_t izabrati akciju A_t koja će doneti najveću nagradu. Zato svaki RL agent ima jednu (možda i više) od sledeće tri komponente:

Polisa: funkcija ponašanja agenta
Funkcija vrednosti stanja: funkcija koja za svako stanje vraća vrednost koja opisuje koliko je to stanje „dobro“
Model: agentova reprezentacija okruženja

U naredna 3 potpoglavlja se nalazi opis komponenti i primer na osnovu lavirinta. Primer je predstavio David Silver u svom predavanju [1], i prikazan je na slici 2.3.

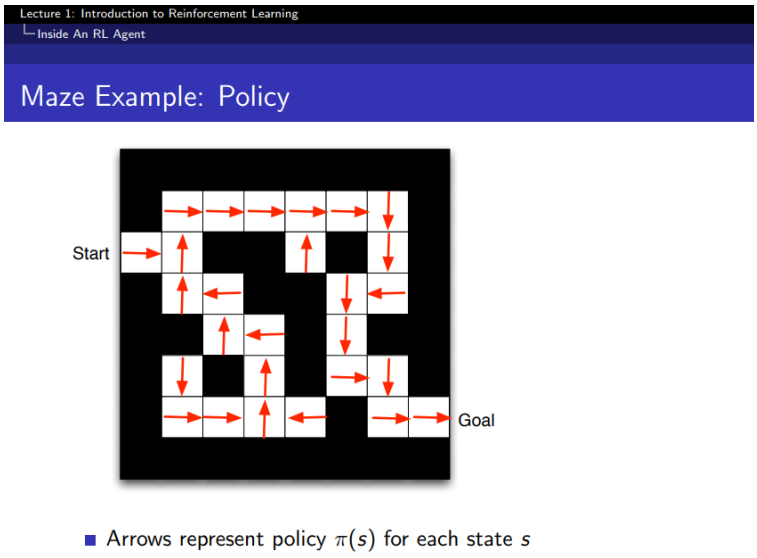


Slika 2.3 Lavirint primer

Cilj agenta, u ovom primeru, je naći što bolji način (u ovom slučaju brži) da pređe lavirint od početka (*Start*) do cilja (*Goal*). Primer je veoma jednostavan, kao što se može videti na slici. Stanje agenta je opisano njegovom lokacijom u lavirintu, Akcije agenta su gore (N), desno (E), dole (S) i levo (W). Prilikom svakog poteza agent dobija nagradu -1, što u principu znači da agent što više vremena provodi rešavajući lavirint, veću kaznu (manju nagradu) će dobiti.

2.1.1 Polisa

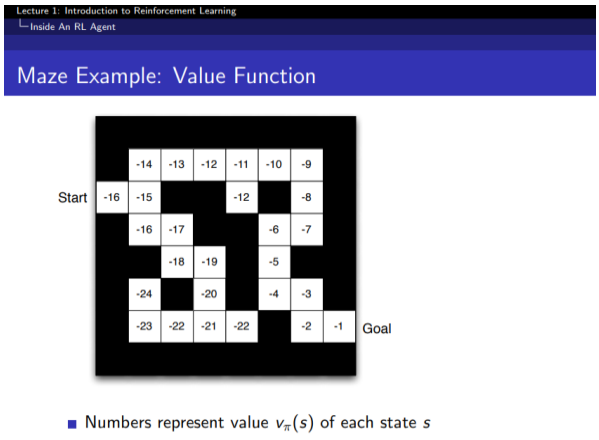
Polisa je funkcija koja opisuje ponašanje agenta. Polisa za svako stanje pamti najbolju akciju koju agent treba da izvrši, tj. pamti akciju koja će doneti najveću nagradu. Jedan od ciljeva RL agenta je i određivanje optimalne polise, tj. agent treba odrediti za svako stanje koja će mu akcija najviše nagrade doneti. Primer optimalne polise za lavirint je dat na slici 2.4.



Slika 2.4 Optimalna polisa

2.1.2 Funkcija vrednosti stanja

Funkcija vrednosti stanja govori za trenutno stanje i koliku nagradu agent očekuje da će dobiti. Jedan od ciljeva RL agenta je izračunavanje optimalne funkcije vrednosti stanja. Primer optimalne funkcije vrednosti je dat na slici 2.5.



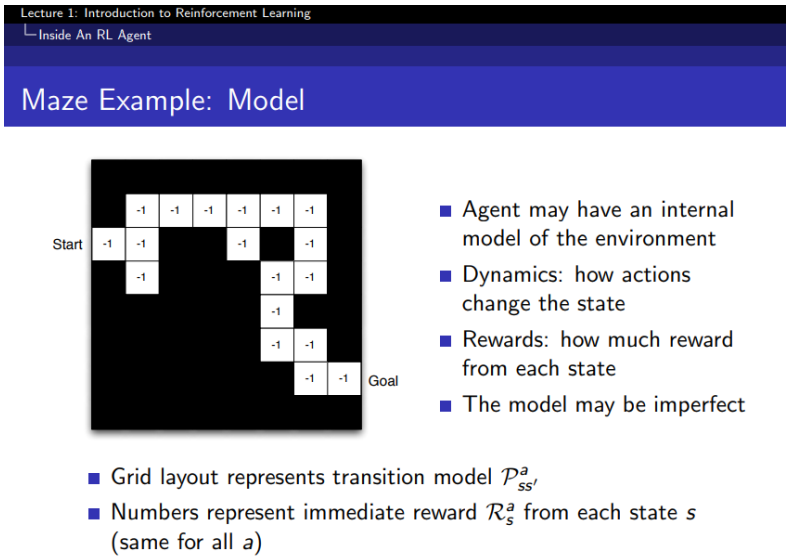
Slika 2.5 Optimalna funkcija vrednosti

Pomoću optimalne funkcije vrednosti moguće je izračunati i optimalnu polisu. Optimalna polisa za svako stanje bi se računala tako što se za to stanje bira akcija koja vodi u naredno stanje koje ima najveću moguću vrednost (vrednost se računa pomoću funkcije vrednosti stanja).

2.1.3 Model

Prvo treba napomenuti da se RL algoritmi mogu podeliti na one koji su bazirani na poznatom modelu (*Model-Based*) i na algoritme koji ne poznaju model (*Model-Free*). Za probleme gde je model poznat pomoću dinamičkog programiranja može se na efikasan način odrediti optimalna polisa (kasnije u radu će biti objašnjeno šta znači optimalna polisa i kako doći do nje). Ovaj rad se bavi problemima gde model nije poznat.

Primer modela za lavirint je dat na slici 2.6.



Slika 2.6 Model lavirinta

Prednost kada agent poznaje model je ta što, pored dinamičkog programiranja, brojni *planning* algoritmi se mogu iskoristiti za rešavanje konkretnog problema. Ali glavni problem je što je u većini slučajeva veoma teško ili nemoguće odrediti model okruženja. To je slučaj kada je broj stanja prevelik ili kada stanja nisu opisana diskretnim nego kontinualnim vrednostima. Primer sa kontinualnim vrednostima za stanja može biti agent čija se pozicija čita iz realnog sveta, tj. nije iz skupa diskretnih vrednosti. Drugi slučaj kada je teško napraviti model okruženja je kada okruženje ima u sebi određeni slučajni faktor. Primer za to bi bila kartaška igra, tu je veoma teško napraviti model zato što agent ne može znati koju će sledeću kartu dobiti od delioca (okruženja).

Ovaj rad se bavi Sarsa algoritmom i koristi funkciju vrednosti stanja i polisu, a okruženje i njegov model su nepoznati. Postoje algoritmi pomoću kojih agent kreira model okruženja i onda taj model kasnije obrađuje pomoću *planning* algoritama, ali oni nisu tema ovog rada. Više informacija o modeliranju okruženja može se naći u 8. poglavlju knjige „Reinforcement Learning: An introduction“ [2].

2.1.4 Nagrade i kazne

Za sad je samo spomenuto da je cilj agenta da maksimizuje nagradu koju dobija od okruženja, ali nije bilo dodatno objašnjenje šta to tačno znači. Bilo je reči da agent u trenutku **t**, bira akciju A_t i dobija nagradu R_t. Iz zadnje rečenice sledi zaključak da je zadatak agenta da modifikuje svoje ponašanje kako bi u trenutku **t**, dobio što je veće moguće R_t. Ovo je sasvim validan zaključak i moguće je napraviti RL agenta koji se ovako ponaša, ali maksimalna moguća nagrada u trenutku ne znači isto što i maksimalna moguća nagrada u dužem vremenskom periodu. Veoma često se dešava da je bolje da se žrtvuje trenutna nagrada, kako bi se u dužem vremenskom periodu ostvarila još veća nagrada. Možda se najbolji primer koji ilustruje ovu tvrdnju može naći kod investicija. Ponekad je bolje ne uzeti odmah zarađeni novac (nagradu), nego je bolje taj novac i dalje investirati kako bi ukupna zarada (nagrada) bila što veća.

Sada se već dolazi do drugog zaključka šta bi bio zadatak RL agenta. Ako posmatramo od trenutka **t** na dalje, nagrade koje okruženje pruža agentu bi bile R_t, R_{t+1}, R_{t+2} itd. Pa bi onda zadatak agenta bio da maksimizuje ukupnu kumulativnu nagradu. Predstavićemo G_t kao ukupnu nagradu od trenutka **t**.

$$G_t = R_t + R_{t+1} + R_{t+2} + \cdots$$

$$G_t = \sum_{i=0} R_{t+i} \tag{1}$$

Zadatak agenta bi bio da u trenutku **t** nauči da maksimizuje G_t. Ali da li je ispravan način? Odgovor bi bio **ne**. Pravi način je negde između maksimizovanja samo R_t i maksimizovanja G_t. Problem zadnje formule je taj što u trenutku **t** nagrade iz daleke budućnosti se isto vrednuju kao i trenutna nagrada, tako da za kontinualni sistem gde kraj ne postoji u svakom trenutku bi G_t bio ∞. Intuitivnije je da se trenutne nagrade više vrednuju nego buduće nagrade, isto, za okuženja koja u sebi imaju slučajan (*random*) faktor, nagrade koje u budućnosti nisu izvesne. Iz tog razloga se uvodi *discount* **γ** faktor na sledeći način:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} \ldots$$

$$G_t = \sum_{i=0} \gamma^i R_{t+i} \tag{2}$$

Vrednost za **γ** mora da bude između 0 i 1 kako bi vrednost za G_t težila realnom broju, a ne ∞. Za γ = 0 imamo slučaj gde agent vrednuje samo trenutnu nagradu, ne i one u budućnosti. Za γ = 1, imamo slučaj da se nagrade u budućnosti vrednuju kao i trenutna nagrada.

U poglavlju 2.1.2 rečeno je da funkcija vrednosti stanja za svako stanje vraća vrednost koja predstavlja očekivanu nagradu koju će agent dobiti.

$$\begin{aligned} V(S_t) &= \mathrm{E}\,[G_t] \\ &= \mathrm{E}\,[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \cdots] \\ &= \mathrm{E}\,[R_t + \gamma\,(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots)] \\ &= \mathrm{E}\,[R_t + \gamma\, G_{t+1}] \\ &= \mathrm{E}\,[R_t + \gamma\, V(S_{t+1})] \end{aligned} \tag{3}$$

Pomoću Bellman[2]-ove jednačine, funkcija vrednosti stanja može se rekurzivno rastaviti. Rastavljanjem jednačine vidi se da je funkcija vrednosti stanja za S_t jednaka sumi očekivane nagrade u trenutku **t** i funkcije vrednosti stanja za S_{t+1} sa uračunatim popustom (*discount* faktorom).

2.1.5 Istraživanje vs iskorišćavanje

Jedan od problema RL-a je „istraživanje vs iskorišćavanje" (*exploration vs exploitation*). Istraživanje podrazumeva da agent češće posećuje manje istražena stanja kako bi saznao više informacije o njima. Iskorišćavanje podrazumeva da agent ide sigurnom putanjom za koju misli da će dobiti više nagrade. Ako agent više istražuje onda će posetiti mnoga stanja i imaće više informacija o okruženju u kome se nalazi, ali isto tako će pokupiti manje nagrade. U suprotnom, ako agent više iskorišćava onda može pokupiti veću nagradu, ali veći deo okruženja može ostati nepoznat agentu. Tako da je potrebno naći balans između ove dve stvari, tj. napraviti agenta koji će dovoljno istraživati nepoznata stanja, ali ne previše kako ne bi propustio da pokupi nagrade tokom svog rada.

Primer zašto je potrebno istraživanje neposećenih stanja:

Pretpostavićemo da agent uvek želi da iskorišćava trenutno stanje, tj. da uvek želi da pokupi najveću nagradu, a da ne želi da istražuje neposećena stanja. Sistem za ovaj primer je takav da su zatvorena dvoja vrata, iza jednih vrata se nalazi ćup sa zlatom. Cilj agenta je da izabere vrata koja će otvoriti kako bi našao ćup. Kada izabere vrata gde se nalazi ćup, agent dobija nagradu +1, a ako nema ćupa nema ni nagrade, tj. nagrada je 0. Sada zamislimo situaciju gde agent otvori prva vrata, vidi ćup sa zlatom i dobije nagradu +1, a ćup se onda prebaci iza drugih vrata. Agent zna da je iza prvih vrata video ćup, modifikuje svoju polisu na osnovu toga, i u narednim koracima uvek će birati prva vrata, zato što će funkcija vrednosti za prva vrata uvek biti pozitivna, zbog nagrade koju je dobio na početku, a funkcija vrednosti za druga vrata će uvek biti 0, zato što nikad nije istražio šta se nalazi iza drugih vrata.

Drugi primer zašto je potrebno balansirati između istraživanja i iskorišćavanja može se naći u stvarnom životu.

Odabir odgovarajućeg restorana - Istraživanje bi bilo probati novi restoran, a iskorišćavanje bi bilo izabrati omiljeni restoran. Ako agent ne istražuje onda će uvek izabrati isti restoran, iako možda postoje drugi restorani koji bi mu doneli veću nagradu (bolja hrana, usluga, niže cene...). A ako agent previše istražuje onda će najverovatnije probati mnoge restorane koji bi mu doneli manju nagradu (gora hrana, usluge...).

Jedna metoda koja omogućava balansiranje između istraživanja i iskorišćavanja je *ε-greedy*. *ε-greedy* mehanizam kaže da agent ima verovatnoću ε da će se ponašati pohlepno (*greedy*), što znači da će iskoristiti trenutno stanje da pokupi najveću nagradu. To takođe znači da je verovatnoća 1-ε da agent će istražiti nova stanja. *ε -greedy* tako zvuči u teoriji, a to u praksi znači da svaki put kada agent treba da izabere akciju, generiše se novi slučajan broj između 0 i 1. Ako je vrednost tog broja manja od *ε* onda će agent izbrati akciju za koju misli da je najbolja, a ako je vrednost slučajnog broja veća, onda agent na slučajan način bira akciju koju će izvršiti.

Postoji dosta predloga i načina kako se može *e* računati ili kako iskoristiti neki drugi mehanizam umesto *e-greedy*, neke od tih načina je opisao Danny Britz u svom blogu [3]. U ovom radu agent će imati dva različita moda, jedan za treniranje, drugi za igranje. U prvom ε će biti konstanta, koja se neće menjati, a u drugom ε će imati vrednost 1, tj. agent u modu za igranje će uvek biti pohlepan i neće uopste istraživati. Agent u modu za igranje neće istraživati

[2] Richard E. Bellman, Američki matematičar, kreator dinamičkog programiranja.

zato što će u modu za treniranje naučiti optimalnu polisu (to je cilj), tako da mu neće biti potrebno da istražuje nova stanja, jer ih je u modu za treniranje već posetio.

2.2 Markov proces odluke

Markov proces odluke (***M**arkov **D**ecision **P**roccess*, skraćeno MDP) formalno opisuje okruženje u kome se nalazi RL agent. „Skoro svi Reinforcement learning problemi mogu da se formalizuju kao MDP“ (Silver, 2015). Da bi se objasnio MDP, krenućemo prvo od **Markovog svojstva**.

Stanje S ima Markovo svojstvo ako i samo ako buduća stanja koja slede nakon stanja S nisu zavisna od stanja koja su prethodila stanju S. Ili formalno definisano:

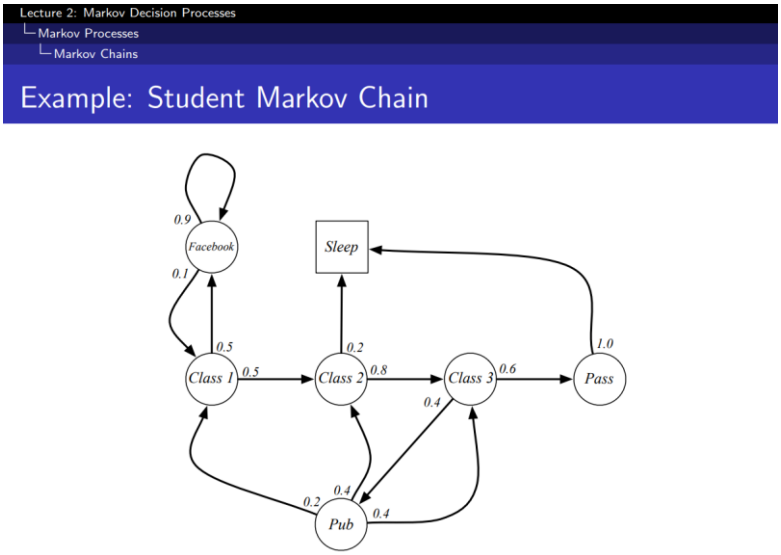
$$\mathrm{P}[S_{t+1} \mid S_t] \;=\; \mathrm{P}[S_{t+1} \mid S_1, S_2, \ldots, S_t] \tag{4}$$

Veoma je značajno da stanja u RL imaju Markovo svojstvo, jer bi to značilo da čim je poznato stanje u kome se agent nalazi, istorija stanja kako je agent došao do trenutnog stanja može se odbaciti. Ako kojim slučajem stanja nemaju Markovo svojstvo, tj. ako prelazak iz stanja S_t u stanje S_{t+1} zavisi od stanja koja su se dogodila pre S_t, onda se definicija stanja treba promeniti kako bi ona imala Markovo svojstvo. Na primer, imamo letilicu kojom upravlja RL agent i stanja su predstavljena pozicijom letelice. U trenutku **t**, letelica se nalazi u stanju S_t i za njen prelazak u stanje S_{t+1} nije dovoljno samo znanje iz stanja S_t. Kako bismo odredili novo stanje (poziciju) letelice moramo znati i njenu brzinu, a ako su stanja predstavljena samo pomoću pozicije onda su nam potrebna i stanja pre S_t kako bi se odredila brzina te letelice. Odnosno, ovako predstavljena stanja nemaju Markovo svojstvo. Kao što je već napomenuto, ovako definisana stanja treba preformulisati na takav način da ona ispunjavaju Markovo svojstvo. Ako stanje definišemo pomoću pozicije letelice i njene brzine, onda imamo stanja koja ispunjavaju Markovo svojstvo. To znači da za prelazak iz stanja S_t u stanje S_{t+1} nije nam potrebna istorija, dovoljno nam je samo znanje iz stanja S_t.

Markov lanac je niz povezanih stanja gde svako stanje poseduje Markovo svojstvo. Markov lanac se sastoji od niza stanja, u kome agent može da se nađe, i od matrice verovatnoće tranzicije. Ta matrica definiše verovatnoću tranzicije iz jednog stanja u drugo.

Primeri koji se koriste u ovom poglavlju su preuzeti sa Silver-ovog predavanja [3].

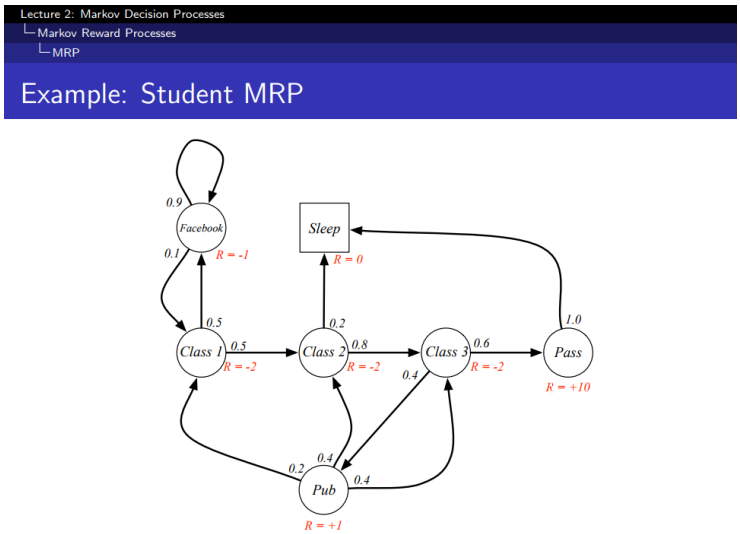
Primer Markovog lanca je ilustrovan na slici 2.7.



Slika 2.7 Primer Markovog lanca

Pomoću Markovog lanca možemo definisati stanja i verovatnoću prelaska iz jednog stanja u drugo. Za RL potrebne su nam i nagrade, tako da ćemo Markov lanac proširiti nagradama, i to bi onda bio Markov nagradni proces (*Markov Reward Process*).

Markov nagradni proces je prošireni Markov lanac, gde je definisana nagrada za svaki prelaz iz jednog stanja u drugo. Takođe se definiše i *discount* faktor (o *discount* faktoru i zašto je on potreban je bilo reči u poglavlju 2.1.4). Na slici 3.7 se nalazi primer Markovog lanca, taj primer ćemo proširiti nagradama kako bi dobili Markov nagradni proces i njegova ilustracija se može videti na slici 2.8.

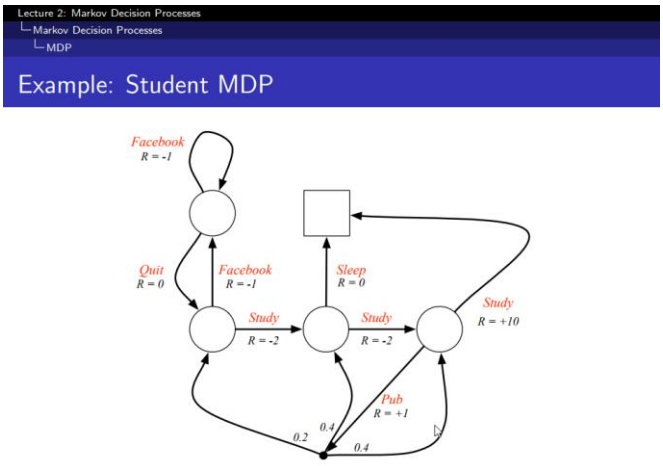


Slika 2.8 Primer Markovog nagradnog procesa

Kod Markovog nagradnog procesa se može naslutiti što bi bio cilj agenta. Cilj agenta bi bio da pokupi što veću nagradu krećući se između stanja. Sa primera koji se nalazi na slici 3.8 mogu se videti nagrade koje agent dobija prelaskom iz jednog stanja u drugo. Npr. agent dobija nagradu +10 kada položi ispit (stanje **Pass** sa slike), nagradu -1 (kaznu) za svaki put kada agent pređe u stanje **Facebook**.

Kada su nagrade dostupne, može se predstaviti i funkcija vrednosti stanja. Funkcija vrednosti stanja za stanje S_t, ima vrednost nagrade koju agent dobija prelaskom u stanje S_t (ova rečenica je tačna samo za slučaj kada je *discount* faktor jednak nuli, detaljnije o ovome će biti reči u sledećem poglavlju).

Ako bismo Markovom nagradnom procesu dodali i akcije koje agent može da odabere, tada bismo dobili Markov proces odluke. Na slici 3.8 se nalazi Markov nagradni proces, taj primer ćemo proširiti akcijama koje agent može da izvrši. Ilustracija proširenja je data na slici 2.9.



Slika 2.9 Primer Markovog procesa odluke

Umesto matrice tranzicije koja definiše verovatnoću prelaska agenta iz jednog stanja u drugo, agent sada može izabrati akcije koje će ga odvesti u sledeće stanje. U MDP-u i dalje mogu da postoje stanja u kojima agent ne može da izabere akciju koju bi izvršio. Ta stanja su obično sakrivena od agenta, a prelazak iz takvog stanja određuje okruženje. Okruženje to određuje *random* faktorom, sa slike 3.9 takvo stanje je *Pub*, okruženje određuje u kom stanju će se agent naći nakon stanja Pub.

„*Funkcija vrednosti stanja* $v_\pi(s)$ kod MDP-a predstavlja očekivanu nagradu koju će agent pokupiti od stanja *s* ako prati polisu π“ [4].

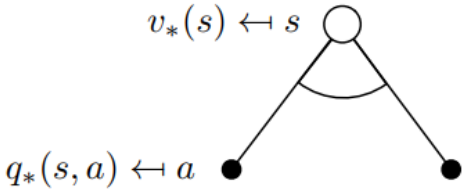
„Funkcija vrednosti akcije $q_\pi(s, a)$ kod MDP-a predstavlja očekivanu nagradu koju će agent pokupiti od stanja *s*, ako izvrši akciju *a* i prati polisu π“[4].

2.2.1 Optimalnost kod Markovog procesa odluke

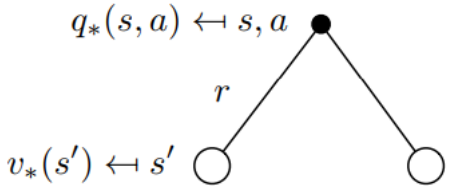
„*Optimalna funkcija vrednosti stanja* $v_*(s)$ je funkcija vrednosti stanja koja ima maksimalnu vrednost za sve moguće polise“ [4].

„*Optimalna funkcija vrednosti akcije* $q_*(s, a)$ je funkcija vrednosti akcije koja ima maksimalnu vrednost za sve moguće polise“ [4].

Veza između funkcije vrednosti stanja i funkcije vrednosti akcije, i kako transformisati jednu u drugu, je ilustrovana u sledeće dve slike (2.10 i 2.11).



Slika 2.10 Funkcija stanja -> akcije



Slika 2.11 Funkcija akcije -> stanja

Priča o optimalnosti kod Markovog procesa odluke je važna, zato što agent koji „zna“ optimalnu funkciju može konstruisati optimalnu polisu, a optimalna polisa ima najbolje performanse. Takođe, čim je poznata optimalna polisa MDP se može smatrati rešenim. Takva optimalna polisa se na dalje može koristi za dobijanje maksimalne moguće nagrade iz MDP-a.

Za probleme gde je poznat model okruženja, MDP se može rešiti pomoću dinamičkog programiranja, ali problemi koji se susreću kod RL-a su toliko veliki da je nemoguće znati sva stanja, nagrade, prelaze iz jednog stanja u drugo itd. Tako da se za nalaženje optimalne polise mogu iskoristiti iterativne metode (npr. Sarsa, *Q-learning*, ...) umesto dinamičkog programiranja.

3 RL ALGORITMI

Pre definicije bilo kog RL algoritma, definicija epizode mora biti objašnjena. Epizoda je niz stanja, akcija i nagrada koje agent prođe, izvrši i pokupi od početnog do završnog stanja. Npr. ako posmatramo neku igru, jedna epizoda bi bila jedna partija koju agent odigra. Nagrada koju agent pokuplja se obično pojavljuje tek na kraju epizode, npr. +1 za pobedu agenta, -1 za poraz i 0 za nerešenu partiju, mada nije neuobičajno da se nagrada pojavi i tokom epizode.

RL algoritmi koji nisu bazirani na modelu (*Model-free*) se mogu podeliti u dve grupe:

- *On-line* algoritmi: algoritmi koji poboljšavaju svoju polisu dok istu koriste da donose odluke. Sarsa i Monte Carlo algoritmi su *on-line*.
- *Off-line* algoritmi: algoritmi koji koriste jednu polisu da donose odluke, a drugu polisu poboljšavaju, kasnije (obično na kraj epizode) polisa koja je korišćena za donošenje odluka se menja polisom koja je poboljšana. *Q-learning* algoritam je primer *off-line* učenja.

Za Model-free algoritme, RL problem se može razložiti na dva problema:

- **Predikcija**: za datu polisu odrediti funkciju vrednosti (stanja i/ili akcije)
- **Kontrola**: optimizovanje polise, tj. nalaženje optimalne polise

U sledećem potpoglavlju biće ukratko objašnjen Monte Carlo algoritam. Iako Monte Carlo algoritam nije tema ovog rada, korisno ga je opisati zbog poređenja sa ostalim RL algoritmima.

3.1 Monte Carlo algoritam

Monte **C**arlo (MC) algoritam ima sposobnost učenja direktno iz interakcije sa okruženjem, nije potrebno znanje modela. MC algoritam rešava RL problem tako što računa prosek svih nagrada dobijenih iz epizode. Kako bi se obezbedilo da nagrada postoji, MC algoritam mora da čeka na kraj epizode. To je glavna mana MC-a. Glavna prednost MC-a je ta što se MC bazira na statistici, tako da ako se svako stanje poseti dovoljan broj puta, MC će naći rešenje RL problema, tj. na osnovu teorije velikih brojeva, MC sigurno konvergira ka optimalnoj polisi.

Na kraju epizode, za svako stanje koje se nalazi u epizodi modifikuje se funkcija vrednosti, tako što se računa prosečna vrednost svih nagrada koje je agent dobio kada je obišao stanje za koje se računa vrednost. Tako da je za svako stanje potrebno i pamtiti koliko puta je obiđeno. Na slici 3.1 se nalazi „lanac“ stanja i akcija za koje se računa funkcija vrednosti.

Slika 3.1 Monte Carlo "lanac"

Formula koja se koristi za modifikaciju funkcije vrednosti stanja kod MC algoritma:

$$\mathrm{V}(S_t) = \mathrm{V}(S_t) + \frac{1}{\mathrm{N}(S_t)} \left(G_t - \mathrm{V}(S_t)\right) \tag{5}$$

$N(S_t)$ – broj koji označava koliko puta je agent posetio stanje S_t

Moguće je odbaciti $N(S_t)$ kako agent pored stanja ne bi pamtio dodatno još jedan niz parametara. Umesto niza $N(S_t)$ uvodi se α parametar (ovaj parametar biće kasnije dodatno objašnjen). Onda bi formula za modifikaciju funkcije vrednosti stanja kod MC algoritma izgledala:

$$\mathrm{V}(S_t) = \mathrm{V}(S_t) + \alpha \left(G_t - \mathrm{V}(S_t)\right) \tag{6}$$

3.2 Temporal-Difference učenje

„Ako se jedna ideja može izdvojiti kao centralna i revolucionarna za RL, to je definitivno ***T**emporal-**D**ifference* (TD) učenje“[3]. TD ima sposobnost učenja direktno iz iskustva, tj. pomoću nagrada koje dobija od okruženja. Glavna prednost TD algoritma je ta što nije potrebno doći do kraja epizode kako bi agent „učio“, zato što agent „uči“ u svakom koraku, tj. nakon svake dobijene nagrade (makar ona bila i 0). Ta prednost se naročito primećuje kod epizoda koju traju dugo ili čak nemaju kraja.

Analizirajući formulu za modifikaciju funkcije stanja, Monte Carlo algoritma (6) i formule (3), dolazi se do formule koju koristi TD:

$$\begin{aligned} \mathrm{V}(S_t) &= \mathrm{V}(S_t) + \alpha \left(\boldsymbol{G_t} - \mathrm{V}(S_t)\right) \\ &= \mathrm{V}(S_t) + \alpha \left(\boldsymbol{R_t} + \boldsymbol{\gamma \mathrm{V}(S_{t+1})} - \mathrm{V}(S_t)\right) \end{aligned} \tag{7}$$

$R_t + \gamma V(S_{t+1})$ – predstavlja **TD cilj**.

$\delta_t = R_t + \gamma V(S_{t+1}) - V(S)$ – predstavlja TD grešku.

Prikazana formula (7), predstavlja TD(0) algoritam koji se koristi za rešavanja problema predikcije.

[3] Reinforcement learning: An introduction [2], 6. poglavlje.

$V(S_t)$ – predstavlja funkciju vrednosti stanja u trenutku **t**.

TD cilj – predstavlja stvarnu nagradu u trenutku **t** koju agent dobija od okruženja (R_t), plus nagrade koje očekuje da će dobiti u budućnosti, sa uračunatim *discount* faktorom ($\gamma V(S_{t+1})$).

TD greška – predstavlja razliku između nove (TD cilj) i stare funkcije ($V(S_t)$) vrednosti za stanje S_t.

γ – *discount* faktor koji govori koliko agent vrednuje buduće nagrade u odnosu na trenutnu nagradu. *Discount* faktor je detaljnije opisan u poglavlju 2.1.4.

α – faktor koji definiše koliki „uticaj" ima TD cilj na vrednost funkcije stanja (*learning rate*). Odnosno, može se smatrati da ovaj faktor definiše u kojoj meri novi podaci brišu stare podatke. Za $\alpha = 1$, imamo slučaj gde se samo nove vrednosti pamte, a za $\alpha = 0$, nove vrednosti se nikad ne pamte, već će funkcija vrednosti uvek imati inicijalnu vrednost.

„Na osnovu stohastičke aproksimacione teorije, ako su sledeća dva uslova ispunjena onda formula (7) konvergira sa verovatnoćom 1" [2].

1) $\sum_{n=1}^{\infty} \alpha_n(a) = \infty$
2) $\sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$

„Prvi uslov garantuje da ako je vrednost za α dovoljno velika, inicijalne vrednosti će biti prevaziđene, dok drugi uslov garantuje konvergenciju ako je vrednost za α dovoljno mala" [2].

Većina dokaza za konvergenciju TD algoritma se odnosi na probleme čiji je broj stanja dovoljno mali da se mogu čuvati u memoriji, kod aproksimatora funkcije (*function approximator*) problem konvergencije je mnogo veći.

Pitanje koje se postavlja je, koji algoritam brže konvergira ka optimalnoj polisi, MC ili TD? Iako matematički još nije dokazano da TD brže konvergira, u praksi se pokazalo da TD brže konvergira ka optimalnoj polisi. Brža konvergencija znači da agent brže „uči", tj. potrebno mu je manje vremena (epizoda) kako bi naučio optimalnu polisu.

3.2.1 Sarsa

Sarsa je algoritam za iterativno računanje optimalne polise kod MDP-a koji se koristi kod RL problema. Algoritam je predstavljen 1994. godine, i puno ime algoritma je ***S**tate-**A**ction-**R**eward-**S**tate-**A**ction*, skraćeno ime „Sarsa" je bilo samo dato u fusnoti u tom radu, ta skraćenica se na dalje zadržala.

Do sada je opisano kako se problem predikcije može rešiti pomoću Monte Carlo ili TD algoritma. Kao što je već napomenuto, to rešava problem određivanja funkcije vrednosti stanja i/ili akcije ako je polisa π poznata. U ovom potpoglavlju biće opisano kako rešiti problem kontrole, odnosno, kako na osnovu iskustva optimizovati polisu, ili konkretnije, na osnovu izabranih akcija i dobijenih nagrada.

Algoritmi za kontrolu mogu se podeliti na *off-line* i *on-line* algoritme. U ovom potpoglavlju biće obrađen *on-line* algoritam Sarsa, *off-line* algoritmi neće biti obrađeni jer nisu tema ovog rada. Prednost on-line algoritma je ta što agent u trenutku kada dobije nagradu može iskoristiti tu nagradu kako bi modifikovao svoju polisu, i od tog trenutka na dalje može koristiti novu modifikovanu polisu. Ukratko, agent čim stekne neko znanje o okruženju, odmah može iskoristiti to znanje za dalji rad.

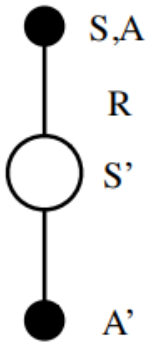
Do sada je u radu bilo govora samo o funkcijama vrednosti stanja, u potpoglavlju 2.2.1 je napomenuto da postoji i funkcija vrednosti akcije. Pa se nameće pitanje, šta je to funkcija vrednosti akcije i zašto bismo koristili tu funkciju kad su formule za funkcije vrednosti stanja relativno jednostavne. Odgovor je jednostavan, **zato što agent ne zna model okruženja**, a i

sve formule koje koriste funkciju vrednosti stanja (5, 6, 7) se mogu primeniti i na funkciju vrednosti akcije, tako da matematički gledano svejedno je koja se funkcija koristi. Ako agent koristi samo funkciju vrednosti stanja, onda će agent moći da oceni koliko je „dobro“ biti u bilo kom stanju, ali pošto je model nepoznat, agent ne može da zna koju akciju treba da izvrši, zato što nema mehanizam da odredi/proceni koliko nagrada će mu doneti izvršena akcija. Zbog tog razloga se koristi funkcija vrednosti akcije.

Formula (7) je funkcija vrednosti stanja, njoj analogna funkcija vrednosti akcije bi bila:

$$\begin{aligned} \mathrm{Q}(S_t, A_t) &= \mathrm{Q}(S_t, A_t) + \alpha \left(G_t - \mathrm{Q}(S_t, A_t)\right) \\ &= \mathrm{Q}(S_t, A_t) + \alpha \left(R_t + \gamma \mathrm{Q}(S_{t+1}, A_{t+1}) - \mathrm{Q}(S_t, A_t)\right) \end{aligned} \qquad (8)$$

Kao što je već rečeno, ime algoritma, Sarsa je dobijeno kao skraćenica od ***S**tate-**A**ction-**R**eward-**S**tate-**A**ction*, što predstavlja pojednostavljen način rada algoritma. Agent se nalazi u stanju S_t, bira akciju A_t, dobija nagradu R_t, prelazi u stanje S_{t+1} i bira akciju A_{t+1}. Tek nakon što se završi ceo ovaj proces, i agent izabere akciju A_{t+1},tek onda agent modifikuje svoju polisu. Ilustracija Sarse je prikazana na slici 3.2.



Slika 3.2 Sarsa

Pomoću formule (8) imamo mehanizam za računanje funkcije vrednosti akcije, ali kako na osnovu funkcije vrednosti akcije da dobijemo polisu? Veoma jednostavno, okruženje u trenutku **t** dostavlja informaciju agentu o stanju u kom se nalazi, agent zna koje akcije bi mogao da izvrši u tom stanju, dovoljno je za te akcije i to stanje da pročita Q vrednosti iz formule (8). U polisu π, za stanje S_t, najbolja akcija koju agent može da izvrši je akcija koja imaju najveću Q vrednosti za to stanje. To znači da se polisa π dobija na osnovu pohlepnog ponašanja u odnosu na funkciju vrednosti akcije q.

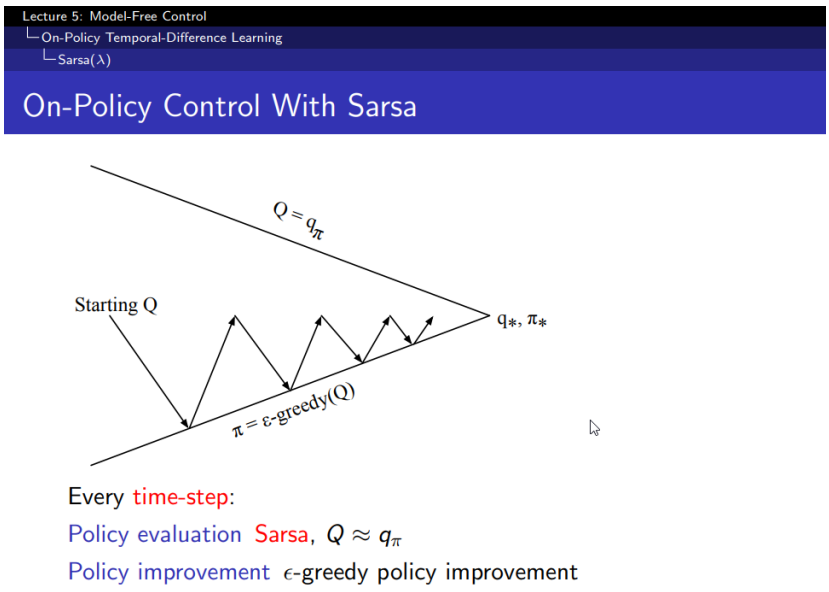
Sarsa je *on-line* algoritam, to znači da u svakom trenutku pomoću formule (8), računamo q_π za polisu π, i u isto vreme računamo polisu π na osnovu pohlepnog ponašanja u odnosu na novo dobijenu funkciju vrednosti akcije q_π. Na ovaj način, Sarsa modifikuje svoju polisu sve dok ne dođe do optimalne polise.

Akcija koja se bira ne bi trebala uvek da bude akcija iz polise π, zbog pomenutog problema u poglavlju 2.1.5. Tako da izabrana akcija neće uvek biti pohlepno izabrana, nego po mehanizmu *ε-greedy*.

Sarsa algoritam je dat u nastavku, opis algoritma je preveden iz knjige [2]:

Inicijalizovati Q(s,a), $\forall s \in S$, $a \in A(s)$, proizvoljno i Q(završno_stanje, ·) = 0
Ponoviti za svaku epizodu:
  Inicijalizovati početno stanje S
  Izabrati A za S pomoću polise izračunate pomoću Q (primer e-greedy)
  Ponoviti za svaki korak u epizodi:
    Izvršiti akciju A, zapaziti R, S'
    Izabrati A' za S' pomoću polise izračunate pomoću Q (primer e-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    S <- S'
    A <- A'
Dok S nije završno stanje

Grafička ilustracija ovog algoritma je data na slici 3.3.

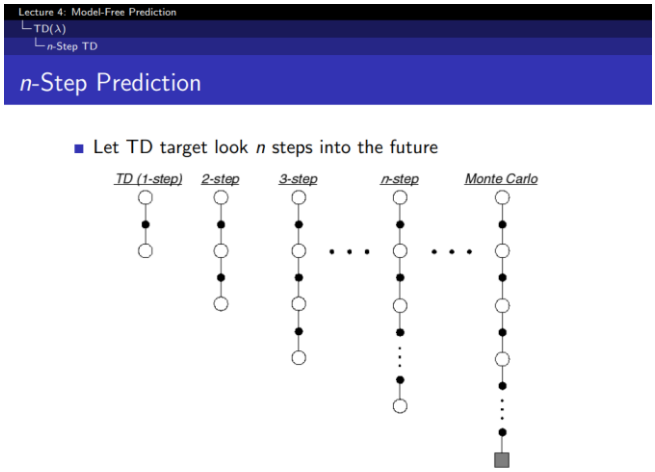


Slika 3.3 Evaluacija i poboljšanje polise u svakom koraku, izvor [6]

3.3 Predikcija u *n* koraka

Prikazani algoritam TD(0) (7) je efikasan, ali i taj algoritam ima mane, jedna od njih je ta što je taj algoritam veoma „pristrastan". Zato što, u svakom trenutku kada se modifikuje funkcija vrednosti stanja, uzima se trenutna nagrada $\mathbf{R_t}$ i nagrada koju algoritam **očekuje** da će dobiti u budućnosti $\boldsymbol{\gamma}$ $\mathbf{V(S_{t+1})}$. „Pristrastnost" dodaje taj faktor što agent ne zna u budućnosti koju će nagradu dobiti, nego koristi očekivanu nagradu. Dok sa druge strane MC algoritam uopste nije „pristrastran", ali ima već spomenuti problem, a to je da agent mora čekati kraj epizode kako bi funkciju vrednosti stanja modifikovao.

Predikcija u *n* koraka obezbeđuje familiju algoritama između TD(0) i MC na taj način što modifikuje svoju funkciju vrednosti stanja nakon ***n*** koraka. Parametar ***n*** definiše koliko koraka TD cilj „gleda u budućnosti". Za $n = 1$, dobija se TD(0) algoritam, za $n = \infty$ (odnosno broj koraka od početka do kraja epizode) dobija se MC algoritam. Na slici 3.4 se nalazi izgled „lanca" stanja i akcija za različite *n* vrednosti.



Slika 3.4 Predikcija za n koraka, izvor [5]

U zavisnosti od *n* parametra TD cilj se drugačije računa. Sledi formula za TD cilj.

$$\begin{aligned} \mathrm{n}=1 \quad & \Rightarrow G_t^{(1)} = R_t + \gamma\, V(S_{t+1}) \\ \mathrm{n}=2 \quad & \Rightarrow G_t^{(2)} = R_t + \gamma R_{t+1} + \gamma^2\, V(S_{t+2}) \\ & \vdots \qquad\qquad \vdots \\ & G_t^{(n)} = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^n V(S_{t+n}) \end{aligned} \tag{8}$$

3.3.1 Sarsa u *n* koraka

Pored predikcije u *n* koraka, moguća je i kontrola u *n* koraka. Za kontrolu u *n* koraka treba modifikovati osnovni Sarsa algoritam. Prvo, formulu (8) treba preformulisati kako bi se definisala nagrada u zavisnosti od funkcije vrednosti akcije, a ne u zavisnosti od funkcije vrednosti stanja. To bi izgledalo:

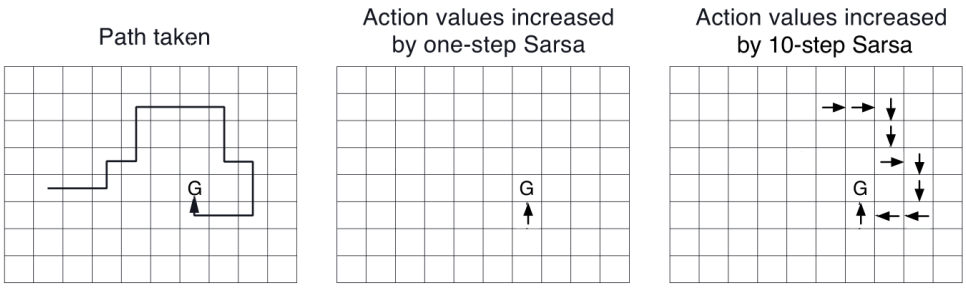
$$\begin{aligned} \mathrm{n}=1 \quad & \Rightarrow q_t^{(1)} = R_t + \gamma\, Q(S_{t+1}, A_{t+1}) \\ \mathrm{n}=2 \quad & \Rightarrow q_t^{(2)} = R_t + \gamma R_{t+1} + \gamma^2\, Q(S_{t+2}, A_{t+2}) \\ & \vdots \qquad\qquad \vdots \\ & q_t^{(n)} = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^n Q(S_{t+n}, A_{t+n}) \end{aligned} \tag{9}$$

Modifikacija funkcije vrednosti akcije za Sarsu u *n* koraka bi bila:

$$\mathrm{Q}(S_t, A_t) = \mathrm{Q}(S_t, A_t) + \alpha\left(q_t^{(n)} - \mathrm{Q}(S_t, A_t)\right) \tag{10}$$

Mehanizam za kontrolu kod Sarse u *n* koraka se ne razlikuje toliko od obične Sarse, jedina razlika je ta što se q_π ne modifikuje posle svakog koraka, nego posle *n* koraka, i to po formuli (10).

Već je napomenuto da mehanizam u „n koraka“ rešava problem „pristrastnosti“ agenta. Ilustracija kako se polisa modifikuje kod obične Sarse i kod Sarsa u *n* koraka je data na slici 3.5.



Slika 3.5 Razlika između obične i n koraka Sarse

Sa slike se može primetiti da kod obične Sarse, u prvoj epizodi se modifikuje Q vrednost samo zadnjeg stanja, tj. stanja koje je prethodilo nagradi. A kod Sarse u *n* koraka, u prvoj epizodi već **n** stanja ima modifikovanu Q vrednost.

3.4 Aproksimizacione metode

Svi algoritmi obrađeni u 3. poglavlju su efikasni tabelarni algoritmi. Ti algoritmi veoma brzo konvergiraju, tj. agent relativno brzo može doći do optimalne polise. Testiranje Sarse sam probao na 2 različita RL problema, oba sa manje od 2000 stanja i agent je uspeo u nekoliko minuta za oba problema da „nauči“ optimalnu polisu.

Za rešavanje nekih realnih, kompleksnijih problema, broj stanja je prevelik. Npr. za igru Backgammon broj stanja je 10^{20}, a za igru Go broj stanja je 10^{180}. Jasno je da je nemoguće držati funkciju vrednosti stanja (i akcije) u memoriji. Ovaj se problem može rešiti pomoću aproksimacione funkcije.

Drugi problem za prevelik broj stanja, pored toga što se vrednosti ne mogu čuvati u memoriji, je taj što će se veoma često agent naći u stanju koje pre toga nije posetio. Tako da je potrebno da agent zna da generalizuje iz sličnih situacija. Vrsta generalizacije koja je potrebna se zove *aproksimaciona funkcija*. Aproksimaciona funkcija uzima realne primere i pokušava da generalizuje sve te primere. Na sreću, veliki broj naučnih oblasti se bavio ovom temom, tako da ništa novo nije neophodno izmisliti za potrebe RL-a. Aproksimacija funkcije je izučavana kod nadgledanog učenja, veštačkih neuronskih mreža, prepoznavanja uzorka itd. Teorijski, sve metode aproksimacione funkcije koje su obrađene u tim oblastima mogu se iskoristiti i za potrebe RL. Formule, ideje i slike koje su predstavljene u ovom potpoglavlju su preuzete sa [7].

3.4.1 Aproksimacije funkcije stanja

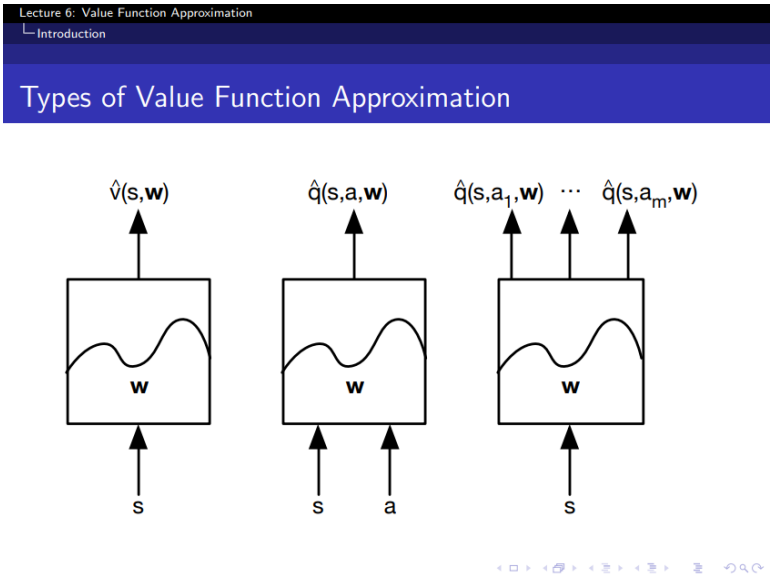
Cilj korišćenja aproksimacione funkcije je da se funkcija vrednosti ne prikaže pomoću tabele, nego pomoću parametarizovane funkcije koja koristi težinski vektor $\boldsymbol{w}$ tako da:

$$\hat{v}(s, \boldsymbol{w}) \approx v_{\pi}(s)$$

$\hat{v}$ može biti linearna aproksimaciona funkcija, gde je $\boldsymbol{w}$ vektor *feature*-a, $\hat{v}$ može biti i neuronska mreža, a $\boldsymbol{w}$ je onda vektor težina između slojeva u mreži. $\hat{v}$ može biti i stablo odlučivanja, ali u ovom poglavlju će biti obrađena i linearna aproksimaciona funkcija i biće dat kratak opis korišćenja neuronskih mreža kao aproksimacione funkcije.

Koja god aproksimaciona funkcija da se koristi, veličina $\boldsymbol{w}$ vektora je mnogo manja nego broj stanja. Promenom jedne težine u vektoru menja se vrednost za više stanja koju aproksimaciona funkcija vraća. Ova vrsta generalizacije daje veliku moć i mogućnost za učenje kod RL algoritma, ali i dosta komplikuje razumevanje samog procesa učenja.

Aproksimaciona funkcija može da se gleda kao i crna kutija, kojoj se za ulaz prosleđuje stanje, a ta kutija vraća vrednost funkcije vrednosti, ilustracija na slici 3.6.



Slika 3.6 Aproksimaciona funkcija

3.4.2 Linearna aproksimaciona funkcija

Ako se J(**w**) definiše kao diferencijalna funkcija za parametarski vektor **w**, gradient od J(**w**) može da se definiše na sledeći način:

$$\nabla_{w} \mathrm{J}(\mathrm{w}) = \left(\frac{\delta J(w)}{w_1}, \frac{\delta J(w)}{w_2}, \ldots, \frac{\delta J(w)}{w_n} \right)^{\mathrm{T}} \tag{11}$$

Pomoću gradient spusta (*gradient discent*) može se naći minimum J(w) funkcije. Formula za nalaženje minimuma bi bila:

$$\Delta w = -\frac{1}{2}\alpha \nabla_w \mathrm{J}(\mathrm{w}) \tag{12}$$

α – parametar koji određuje veličinu koraka kod modifikacije.

Cilj kod gradient spusta je nalaženje minimalne kvadratne greške između aproksimacione funkcije i stvarne vrednosti koju agent dobija od okruženja.

Kod linearne aproksimacije može se definisati vektor karakteristika (*feature vector*) pomoću koga se može opisati svako stanje u kome se može naći agent. Npr. ako se gleda igra šah, u toj igri postoji više od 10^{50} stanja u kome se agent može naći, pa bi čuvanje vrednosti funkcije stanja u tabeli bilo neizvodljivo. Predstavljanje svakog stanja preko vektora karakteristika se može odraditi veoma lako. Npr. vektor možemo gledati kao niz od 64 elemenata, gde svaki element odgovara tačno jednoj ćeliji na šahovskoj tabli, a vrednost koju element ima određuje koja se figura nalazi u toj ćeliji.

Vektor karakteristika se može predstaviti formulom:

$$\mathrm{x}(S) = (x_1(S), x_x(S), \dots, x_n(S))^{\mathrm{T}} \tag{13}$$

Aproksimaciona funkcija koja koristi vektor karakteristika bi izgledala:

$$\hat{v}(S, w) = x(S)^{\mathrm{T}}; w = \sum_{i=0}^{n} x_i(S) w_i \tag{14}$$

Iz zadnje formule sledi da srednja kvadratna greška koju pokušavamo da minimizujemo je:

$$J(w) = E_\pi[(v_\pi(S) - x(S)^{\mathrm{T}} w)^2] \tag{15}$$

Pravilo za modifikaciju je prilično jednostavno:

$$\begin{gathered} \nabla_w \hat{v}(\mathrm{S}, \mathrm{w}) = x(S) \\ \Delta w = \alpha\big(v_\pi(S) - \hat{v}(\mathrm{S}, \mathrm{w})\big) x(S) \end{gathered} \tag{16}$$

Zadnja formula (16), govori da je modifikacija jednaka koraku α puta greška prilikom predikcije $v_\pi(S) - \hat{v}(\mathrm{S}, \mathrm{w})$ puta vektor karakteristika *x(S)*.

Tabelarni metod koji je opisan u prethodnim poglavljima može da se posmatra kao specijalni slučaj vektora karakteristika. Ako bi svako stanje bilo predstavljeno kao jedan element vektora, onda bi u svakom stanju odgovarajući element bio jednak jedinici, a svi ostali elementi bili bi jednaki nuli.

Kod nagledanog učenja imamo test i trening primere na osnovu kojih može da se računa srednja kvadratna greška. Drugim rečima, imamo ulazne podatke i primer koja vrednost treba

da bude izlazna, a kod RL-a, umesto toga imamo nagradu koja se dobija nakon izvršene akcije i očekivanu nagradu u budućnosti.

Do sada je bilo reči o aproksimacionim funkcijama, ali ne i kako se one mogu kombinovati sa RL algoritmima. U nastavku se nalaze formule za predikciju pomoću linearne aproksimacione funkcije:

Za Monte Carlo:

$$\Delta w = \alpha\big(\boldsymbol{G_t} - \hat{v}(S_t, \mathrm{w})\big)\nabla_w \hat{v}(S_t, \mathrm{w}) \tag{17}$$

G_t – za Monte Carlo, nagrada u trenutku **t**, isto kao i u formuli (6)
Za TD:

$$\Delta w = \alpha\big(\boldsymbol{R_t} + \boldsymbol{\gamma \hat{v}(S_{t+1}, \mathbf{w})} - \hat{v}(S_t, \mathrm{w})\big)\nabla_w \hat{v}(S_t, \mathrm{w}) \tag{18}$$

$R_t + \gamma \hat{v}(S_{t+1}, \mathrm{w})$ - za TD, nagrada u trenutku **t**, plus očekivana nagrada u budućnosti, analogno formuli (7).

Već je bilo reči kako jednostavno od formula za predikciju doći do formule za kontrolu, to je bilo objašnjeno kada je Sarsa algoritam bio predstavljen u poglavlju 3.2.1. Analogija je ista i kod aproksimacionih funkcija, tako da bi Sarsa formula za kontrolu pomoću aproksimacione funkcije bila:

$$\Delta w = \alpha\big(\boldsymbol{R_t} + \boldsymbol{\gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})} - \hat{q}(S_{t+1}, A_{t+1}, \mathrm{w})\big)\nabla_w \hat{q}(S_{t+1}, A_{t+1}, \mathrm{w}) \tag{19}$$

Sarsa u *n* koraka koja koristi aproksimacionu funkciju:

$$\Delta w = \alpha\left(\boldsymbol{q_t^{(n)}} - \hat{q}(S_{t+1}, A_{t+1}, \mathrm{w})\right)\nabla_w \hat{q}(S_{t+1}, A_{t+1}, \mathrm{w}) \tag{20}$$

Analogno formuli (9), $q_t^{(n)}$ bi bilo:

$$q_t^{(n)} = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^n Q(S_{t+n}, A_{t+n}, w)$$

Linearne aproksimacione funkcije su zgodne za korišćenje zbog njihove sigurne konvergencije ka optimalnom rešenju. Modifikacija i učenje kod ovih funkcija može da se izvršava relativno brzo, s obzirom da se radi o množenju vektora i matrica, paralelizacija koja može da se iskoristi kod množenja dosta može da ubrza računanje vrednosti.

Pored linearne aproksimacione funkcije postoji i nelinearna aproksimaciona funkcija. Najčešće korišćena nelinearna aproksimaciona funkcija je neuronska mreža. Nakon uspeha kompanije DeepMind, mnogo radova je objavljeno na temu RL gde se kao aproksimator funkcije koristi neuronska mreža. Ova klasa algoritama je dobila i svoj naziv, RL algoritmi koji koriste neuronsku mrežu kao aproksimator funkcije zovu se *deep reinforcement learning* algoritmi. Ovi algoritmi nisu tema ovog rada, tako da neće biti detaljnije obrađeni, DeepMind je objavio odličan rad na ovu temu, kako iskoristiti neuronske mreže zajedno sa *Q-learning* algoritmom za rešavanje RL problema kod Atari 2600 igrica.

3.4.3 Odabir karakteristika (*feature selection*)

Kod mašinskog učenja odabir karaktiristika predstavlja biranje odgovarajućih karakteristika iz skupa svih mogućih. Glavni razlog zašto je *feature selection* važan mehanizam je taj što određeni domen može imati mnogo različitih karakteristika, a samo određene su bitne za rešavanje konkretnog problema. Nije samo poželjno izbaciti „nepotrebne“ karakteristike, već i one karakteristike koje su u međusobnoj korelaciji. Npr. ako jedna karakteristika opisuje X, druga opisuje Y, a treća karakteristika opisuje X + Y onda se treća može izbaciti jer je ona u korelaciji sa prve dve, tj. dovoljne su prve dve za predstavljanje modela. Poenta kod *feature selection*-a je da izbacivanjem karakteristika, gubitak informacija o modelu je minimalan ili u neki slučajevima gubitak i ne postoji.

Pored biranja odgovarajućih karakteristika važno je i u kom opsegu su vrednosti karakteristika predstavljene. U trenutku kada agent dobije nagradu, *feature*-i koji imaju najveću vrednost biće najviše nagrađeni (ili kažnjeni). Iz ovog razloga veoma je važno koju maksimalnu vrednost karakteristike mogu imati i kolika je ona u trenutku dobijanja nagrade.

Primer kako se *feature*-i mogu predstaviti kod retro igrica i kako utiče opseg vrednosti biće detaljnije objašnjeni u 5.2 poglavlju, na igri *Breakout*.

4 OKRUŽENJE ZA TESTIRANJE

Zadnjih nekoliko godina objavljen je veliki broj radova u kojima neka vrsta veštačkog agenta „uči“ da igra neku jednostavnu igricu. Zbog tih potreba, Atari 2600 je ponovo oživeo, delom zbog nostalgije za klasičnim retro igricama sedamdesetih i osamdesedih godina prošlog veka, a delom i zbog jednostavnosti igrica. Jednostavne igre su omogućile i kreiranje agenta sa generalnom inteligencijom, što znači da na osnovu istog algoritma agent može da nauči da igra različite igre. Kako bi *developer*-ima omogućili lakše razvijanje algoritma i mogućnost da se fokusiraju samo na algoritme veštačke inteligencije, a da ne brinu o tome kako pokrenuti stare Atari igre na običnom PC-u, sa Univerziteta u Alberti su napravili okruženje za arkadne igrice (***A**rcade **L**earning **E**nvironment*)[9].

4.1 ALE

ALE predstavlja jednostavan objektno-orijentisan *framework* za razvijanje veštačkog agenta koji igra Atari 2600 igre. ALE se razvija od 2013. godine, *open-source* je projekat napisan u C++ jeziku. ALE koristi Stella emulator kako bi emulirao Atari 2600 igre na PC-u. Trenutno postoji podrška za preko 50 različitih igrica.

Kompletan kod je dostupan na adresi:

https://github.com/mgbellemare/Arcade-Learning-Environment

Karakteristike ALE-a[4]:

- Objektno-orijentisan framework za testiranje agenta
- Lako dodavanje podrške za nove igrice
- Jezgro framework-a je nezavisno od renderovanja slike i zvuka
- Automatsko izvlačenje trenutnog rezultata i generisanja signala za kraj epizode
- Podrška za više platforme
- Komunikacija između agenta i jezgra se obavlja preko *pipe*-ova, tako da je podržano razvijanje agenta u više različitih programskih jezika
- Podrška za Python
- Agenti koji su napisani u C++ jeziku imaju pristup svim karakteristikama ALE-a
- Alati za vizuelizaciju

ALE ima metode koje se izvršavaju svakog *frame*-a, tako da je *developer*-ima lako da se priključe u te metode kako bi se dodala logika agentu koji se implementira. ALE pruža mogućnost da se u svakom *frame*-u pokupi slika, koja se trenutno renderuje, kao niz piksela i/ili da se pročita bilo koja vrednost iz ROM-a igrice.

U ROM-u igrice nalaze se sve informacije o igrici, npr. za igru Pong, u ROM-u se mogu naći informacije o poziciji i brzini loptice, o poziciji protivničkog igrača, poziciji kontrolisanog igrača, trenutni rezultat epizode itd. Tako da se pomoću ROM-a može pročitati trenutni rezultat koji se koristi kao nagrada kod RL algoritma. Ako RL agent uči pomoću piksela, onda agent može iskoristiti, spomenutu, ALE-ovu funkciju za dobijanje niza piksela u svakom frame-u, a ako se stanja definišu drugačije, potrebno je samo pročitati vrednost iz ROM-a na odgovarajućoj adresi.

[4] Karakteristike su preuzete sa zvaničnog repozitorijuma ALE-a

4.2 Dodavanje nove igre u ALE-u

Dodavanje nove igre u ALE-u je vrlo jednostavno. Kako je ALE objektno-orijentisan, svaka igra se nalazi u posebnoj klasi. Tako da prilikom dodavanja nove igre treba kreirati novu klasu, tj. novi .hpp i .cpp fajl. Nova klasa mora da implementira *RomSettings* apstraktnu klasu. Kada se ALE pokrene, na osnovu apstraktne metode *rom()* i učitanog ROM-a, ALE zna objekat koje klase treba da instancira. Apstraktna klasa definiše skup metoda koju svaka klasa za igricu mora da ispuni. U nastavku su date najvažnije apstraktne metode i opis šta njihova implementacija treba da izvršava:

`virtual void reset() = 0;` - metoda koja se izvršava na kraju svake epizode
`virtual bool isTerminal() const = 0;` - određuje da li je stanje u kome se nalazi agent terminalno
`virtual reward_t getReward() const = 0;` - vraća nagradu za novo stanje u kome se agent nalazi
`virtual const char *rom() const = 0;` - ime ROM fajla
`virtual bool isMinimal(const Action &a) const = 0;` - da li akcija a pripada skupu minimalnih akcija
`virtual void step(const System &system) = 0;` - procesira se najnovija informacija iz ALE-a, obično se ovde zapaža trenutno stanje u kome se nalazi agent
`virtual ActionVect getStartingActions();` - definiše se niz akcija potrebnih za startovanje igre, po *default*-noj implementaciji `ActionVect` je prazan vektor

4.3 Druga okruženja/platforme za testiranje algoritma bazirana na ALE-u

ALE je napravio revoluciju sa njihovom platformom, *developer*-ima je toliko bio olakšan razvoj algoritma tako da čak i velike kompanije kao što su DeepMind i OpenAI nisu razvijale svoje platforme, nego su koristili ALE sa dodatnim ekstenzijama/*wrapper*-ima. DeepMind je svoju verziju nazvao xitari. Xitari je *open-source* projekat dostupan na adresi:

https://github.com/deepmind/xitari

OpenAI je svoju platformu nazvao Gym. Gym je open-source projekat dostupan na adresi:

https://github.com/openai/gym

Izraelski Institut Tehnologije razvio je ***R**etro **L**earning **E**nvironment*, koji je isto baziran na ALE-u. U principu to i jeste ALE, ali je dodata podrška za ***S**uper **N**intendo **E**ntertainment **S**ystem*, Sega Genesis i druge konzole [10]. RLE je *open-source* projekat, kod je dostupan na adresi:

https://github.com/nadavbh12/Retro-Learning-Environment

Kolega sa Elektronskog fakulteta, Aleksa Trajković, je napravio sličan korak kao i ljudi sa Izraelskog Instituta Tehnologije, tako što je proširio RLE dodavši mu mogućnost da pokreće igre koje su napravljene za **N**intendo **E**ntertainment **S**ystem. Modifikovani RLE smo kolega i ja zajedno iskoristili za implementiranje agenta koji igra igru Flappy Bird. To je bilo potrebno zato što je napravljen NES port za tu igru, a RLE nije imao mogućnost pokretanja NES igrica.

Implementacija Flappy Bird agenta je bio naš zadatak iz predmenta Soft Computing. Rezultati rada biće predstavljeni u 5.1 poglavlju.

Kod dostupan na adresi:

https://github.com/aleksatr/Retro-Learning-Environment

U ovom potpoglavlju se može videti da je ALE veoma popularan, upravo iz tog razloga ovaj projekat se i dalje razvija, što se može videti na oficijalnom GitHub repozitorijumu.

5 IMPLEMENTACIJA

Kao što je ranije napomenuto, biće prikazana implementacija algoritma Sarsa „u n koraka“ za tri različite igre. Prvi predstavljeni algoritam biće tabelarna Sarsa (bez aproksimacione funkcije) za igranje igre *Flappy Bird*. Algoritam sam implementirao zajedno sa kolegom Aleksom Trajkovićem u sklopu predmeta Soft Computing. Drugi algoritam, na kome je najveći akcenat ovog rada je takođe Sarsa „u n koraka“ sa linearnim aproksimatorom funkcije, a korišćena igrica je *Breakout*. Treća igra za koju će biti prikazani rezultati je igra Pong, a korišćeni algoritam će biti isti kao i za igru *Breakout*, tj. parametri algoritma će biti isti, jedina razlika će biti u karakteristikama koje su potrebne za tu igru.

Tabelarni pristup za igru *Flappy Bird* je iskorišćen zbog jednostavnosti igre, tj. konkretno za tu igru aproksimator funkcije nije potreban jer je broj stanja relativno mali.

Kod igre *Breakout* sam koristio linearni aproksimator umesto nelinearnog iz sledećih razloga:

- Algoritam je specijalizovan za igru
- Hardverska ograničenja
- Jednostavnija implementacija

Nelinearni aproksimator funkcije uglavnom podrazumeva neuronsku mrežu. Prevelik broj radova na ovu temu koristi neuronsku mreže, gde je cilj napraviti generalnog agenta na osnovu piksela. Pod generalnim agentom se podrazumeva da se isti algoritam razvija i testira nad velikim brojem retro igrica, to se može postići zato što je stanje u kome se agent nalazi definisano nizom piksela u tom trenutku.

Korišćen linearni aproksimator je specijalizovan za *Breakout* igru, to znači da vektor karakteristika sadrži samo značajne karakteristike za tu igru. Npr. broj i pozicija preostalih cigli. Specijalizovan agent može brže doći do optimalne polise od generalnog, jer koristi samo neophodne informacije, ali opet algoritam ima manu da funkcioniše samo za igru za koju je specijalizovan.

Drugi razlog, hardverska ograničenja, je naveden iz razloga što se treniranje agenta pomoću neuronske mreže uglavnom obavlja na GPU-u zbog paralelizacije. Radovi koji se bave treniranjem agenta pomoću neuronske mreže (*Deep Q Learning* [7] i njemu slični) svoje agente u proseku treniraju veći broj dana kako bi uspeli da dostignu (ili prestignu) rezultate koje prosečan čovek može ostvariti na određenoj retro igrici. Treniranje je vršeno nad modernim GPU-ovima, a kada bi se koristio CPU potrebno vreme za treniranje agenta bi se dodatno povećalo. Računar na kome sam ja razvijao agenta ima integrisan GPU, tako da bi vreme potrebno za treniranje agenta bilo preveliko. Drugo hardversko ograničenje je RAM memorija, nisam uspeo da pokrenem *Deep Q Learning* algoritam nad RLE okruženju zbog nedostatka RAM-a memorije, a računar na kome sam vršio treniranje ima 8 GB RAM memorije.

Moj cilj je bio da razvijem agenta koji bi uspeo da nauči optimalnu polisu za relativno kratko vreme (par sati), u čemu sam i uspeo.

Računar koji je korišćen za implementaciju i treniranje agenta je Dell Latitude E6440.

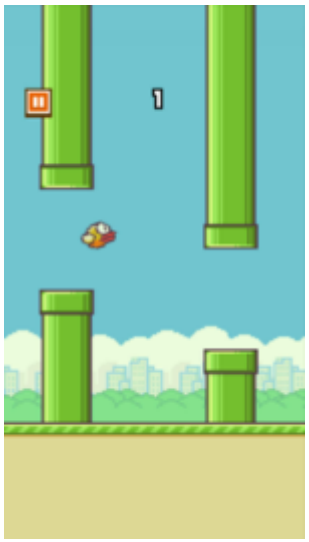
Specifikacija:

- CPU : i7-4610M 3.0 GHz
- RAM : 8 GB DDR3
- Operativni sistem : Ubuntu 16.04

5.1 Flappy Bird

Flappy Bird igra je napravljena za mobilne telefone, konkretnije za Android i iOS platformu. Igra je nastala 2013. godine, pravu popularnost je stekla 2014. godine, a razvio ju je Dong Nguyen u kompaniji dotGEARS. U jednom trenutku 2014. godine, *Flappy Bird* je bila najpopularnija besplatna aplikacija na iOS *App Store*-u. Igra je toliko veliku popularnost stekla, da, iako je bila besplatna, dnevno je donosila profit od 50.000$ samo preko reklama[5].

Velika popularnost je stečena zbog jednostavnosti igre, protagonista je ptica koja konstantno leti s leva na desno i na koju utiče gravitacija. Pritiskom na ekran, ptica zamaše krilima i tako se sprečava da ne padne na zemlju. U ovoj igrici postoje i prepreke kroz koje ptica treba da „prođe“, prolaskom kroz prepreku rezultat igre se povećava za 1. Igra se završava u trenutku kada ptica „pogine“, tj. kada padne na zemlju ili udari u prepreku.



Slika 5.1 Flappy Bird

Flappy Bird nije retro igra, a ovaj rad se bavi retro igricama, međutim zbog svoje velike popularnosti igrica je portovana za NES, tako da je moguće pokrenuti igru pomoću RLE okruženja. RLE u startu nije imao podršku za ovu igru, tako da je podrška morala biti implementirana. Dodavanje nove igre u RLE okruženje je ista kao i za ALE, a to je opisano u poglavlju 4.2.

Klasa koja opisuje igru se može naći na adresi:

- FlappyBird.cpp
- FlappyBird.hpp

Akcije za igru Flappy Bird su:

- Skoči (*Jump*)
- Bez akcije (NOOP)

[5] https://www.theverge.com/2014/2/5/5383708/flappy-bird-revenue-50-k-per-day-dong-nguyen-interview

Stanja u kojima se agent može naći i vrednosti koje ta stanja mogu imati su:

- Visina ptice: 1 – 180
- Smer ptice: 1 – 5
- Visina prepreke: 1 – 180

Na osnovu ovih vrednosti može se videti da je broj različitih stanja u kojima se agent može naći: 180 * 5 * 180 = 162.000.

Mora se ponovo napomenuti da se kod tabelarne Sarse pamte Q vrednosti, što predstavlja „stanje – akcija“ par, tako da bi Flappy Bird predstavljen pomoću gore opisanih stanja zahtevao 324.000 različitih vrednosti, zato što za ovu igru postoje samo dve, već pomenute, akcije.

Problem kod toliko velikog broja Q vrednosti je što agent da bi naučio optimalnu polisu mora da svaki par stanje – akcija poseti „više“ puta. Veštački agent i posle više od 8 sati rada nije uspeo dovoljno znanja da stekne kako bi prolazio prepreke.

Broj „stanje – akcija“ para je prevelik da bi ih agent sve posetio i za svako stanje izračunao odgovarajuću Q vrednost. Tako da je potrebno smanjiti broj različitih stanja. Analiziranjem igrice može se videti da postoje 4 različite vrste prepreka, pa se broj stanja može smanjiti tako što se umesto visine prepreke koristi vrsta prepreke.

Na ovaj način se broj „stanje – akcija“ para smanjuje na: 180 * 5 * 4 *2 = 7.200.

Daljom analizom igrice dolazi se do zaključka da se broj stanja još više može smanjiti, tako što se umesto visine ptice može koristiti relativna visina između ptice i prepreke. Time se broj „stanje – akcije“ para smanjuje na 180 * 5 * 2 = 1.800.

1.800 je relativno mali broj stanja, i ovaj broj stanja agent može da poseti. Sada kada su stanja i akcije definisani potrebno je definisati i nagrade i kazne koje agent dobija igrajući igricu.

Intuitivno bi bilo da ptica dobije kaznu -1 kada pogine i nagradu +1 kada prođe prepreku. Ali zbog bržeg nalaženja optimalne polise mi smo se odlučili za drugi pristup. Kako se prepreke pojavlju uvek u istom intervalu odlučili smo se da agent dobija nagradu +1 za svaki *frame* dok je ptica „živa“, a kaznu od -100 kada ptica pogine. Agenta smo odlučili da nagrađujemo sve dok je „živ“, zato što agent uči da maksimizuje nagradu koju dobija, a konkretno igra *Flappy Bird* je tako koncepirana da što je agent duže živ kroz više će prepreka proći. Agent dobija kaznu kada pogine, a vrednost -100 umesto -1 je izabrana zato što je otprilike 100 *frame*-a razmak između dve prepreke. Zbog ovih 100 *frame*-a, agent bi prilikom prolaska kroz prepreku dobio +100 nagradu umesto +1, zbog toga je za kaznu uzeta vrednost -100.

Već je napomenuto da se ova igra pokreće na modifikovanom RLE okruženju, a pored definisanja stanja, nagrade i kazne postoje i ostali parametri koji se moraju definisati kako bi se igra *Flappy Bird* uspešno pokrenula i kako bi se uopšte dobile potrebne informacije za Sarsa algoritam.

Parametri koje treba definisati su:

- Startne akcije
- Terminalno stanje
- Informacije o rezultatu
- Visina ptice
- Visina prepreke
- Smer ptice

Startne akcije su akcije koje se trebaju definisati za svaku igricu kako bi se ona pokrenula. Neke retro igrice imaju startni meni u kome treba kliknuti na „*Start game*“ ili neku sličnu opciju za pokretanje. Konkretnije, za *Flappy Bird* treba sačekati da se početna slika učita i onda treba pritisnuti „**Start**“ dugme. Koliko treba sačekati se utvrđuje eksperimentalno, za igru Flappy

Bird i ROM fajl koji smo našli eksperimentalno smo utvrdili da se niz startnih akcija sastoji iz 500 **NOOP** akcija i jedne **Start** akcije.

Terminalno stanje predstavlja stanje kada ptica udara u prepreku ili pada na zemlju. Potrebno je tačno utvrditi kada se ovo stanje pojavljuje kako bi znali u kom stanju bi trebalo „kazniti“ agenta, a i kako bi RLE znao kada igru treba prekinuti i pokrenuti ponovo. Terminalno stanje se određuje pomoću dve memorijske lokacije u ROM fajlu, prva memorijska lokacija opisuje da li je ptica živa i nalazi se na adresi 0x003C, druga memorijska lokacija opisuje da li je igra pokrenuta i nalazi se na adresi 0x0049. Terminalno stanje je stanje kada je na prvoj lokaciji vrednost 0 (ptica nija živa) **i** na drugoj lokaciji vrednost 1 (igra je pokrenuta).

Informacija o rezultatu nije bitna za rad algoritma (već je objašnjeno da agent dobija nagradu svakog frame-a kada je živ, a ne prilikom prelaska kroz prepreku), ali se ta informacija koristi kako bi se statistički obradio napredak agenta. Četiri različite memorijske lokacije se koriste za računanje rezultata. Na lokaciji sa adresom 0x0041 pamti se rezultat do vrednosti 10, na lokaciji 0x0042 pamti se deseti deo rezultata, na lokaciji 0x0043 pamti se stoti deo rezultata i na lokaciji 0x0044 pamti se hiljaditi deo rezultata.

Visina ptice je se pamti na lokaciji sa adresom 0x003F. Kao što je već napomenuto moguće su vrednosti od 1 – 180.

Visina prepreke se pamti na lokaciji sa adresom 0x003B. Moguće vrednosti su od 1 – 180. Visina prepreke i visina ptice su bitne informacije, jer se na osnovu njih računa relativna visina ptice koja se koristi za rad Sarsa algoritma.

Informacija o smeru kretanja ptice se nalazi na lokaciji sa adresom 0x0037. Moguće vrednosti su od 1 – 5.

Do sada je bilo reči o parametrima igrice, na kojim adresnim lokacijama se nalaze koje informacije od značaja, a sada će biti reči o parametrima Sarsa „u n koraka“ algoritma koji koristi agent kako bi došao do optimalne polise. Detaljniji opis parametra neće biti dat, jer su svi ti parametri već obrađeni u poglavljima 2. i 3.

U zavisnosti od toga kako su parametri izabrani algoritam brže ili sporije konvergira ka optimalnoj polisi. U nastavku će biti prikazani parametri na osnovu kojih je agent naučio optimalnu polisu i biće prikazani rezultati agenta, tj. biće prikazan napredak agenta kroz epizode.

Parametri su:

- $\alpha = 0.01$ – *learning rate*
- $n = 2$ – broj koraka Sarsa algoritma
- $\varepsilon = 1 / (20 + i)$ – faktor istraživanja, **i** predstavlja broj epizode, tako se postiže da faktor istraživanja vremenom opada
- $\gamma = 0.8$ – *discount* faktor

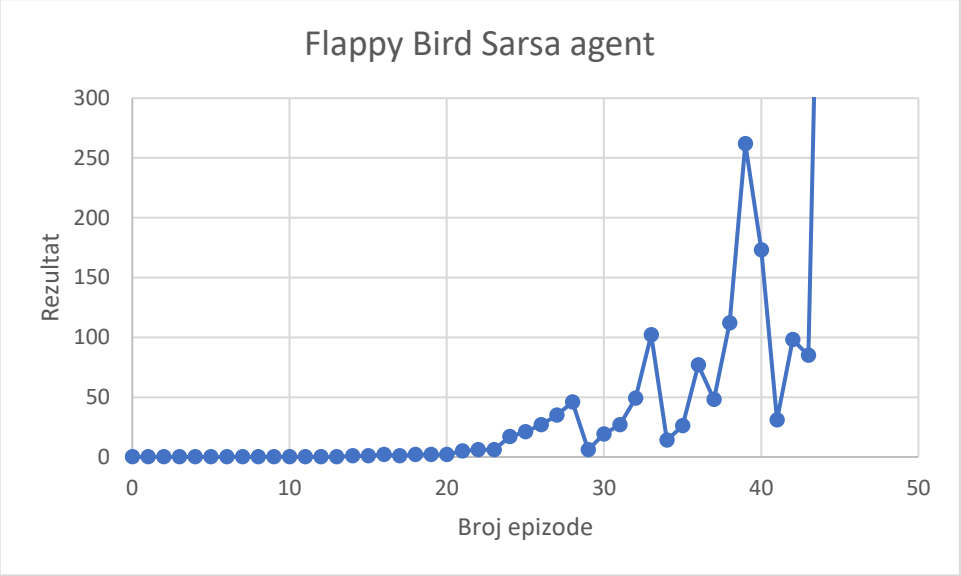
U tabeli 5.1 prikazani su rezultati Sarsa agenta za igru Flappy Bird. U 42. epizodi agent je naučio optimalnu polisu, što znači da može da igra igru a da ptica ne udari u prepreku ili padne na zemlju. U tabeli 5.1 za optimalnu polisu stoji rezultat 682, a kako je rečeno da je ptica naučila da ne „gine“ očekuje se da piše neki rezultat koji teži beskonačnosti. Programer koji je portovao igru na NES sistem nije očekivao da će neko imati strpljenja da igra Flappy Bird epizodu duže od 20 minuta tako da u trenutku kada ptica stigne do 683. prepreke dogodi se *integer overflow* i zbog toga ptica pogine.

Epizoda	Rezultat
10	**0**
15	**1**
21	**5**
24	**21**
25	**48**
30	**20**
33	**102**
42	**682 (optimalna polisa)**

Tabela 5.1 Rezultati Sarsa agenta

Ja sam igrao *Flappy Bird* igru na svom telefonu, i najveći rezultat koji sam ostvario, posle mesec dana igranja, bio je 190. Tako da je agent nakon 33. epizode dostigao rezultat koji prosečni igrač može ostvariti, a od 42. epizode je naučio optimalnu polisu, tj. od 42. epizode agent je naučio da prolazi kroz prepreke na taj način da nikad ne pogine.

Na grafikonu 5.1 prikazan je rezultat u zavisnosti od epizode.



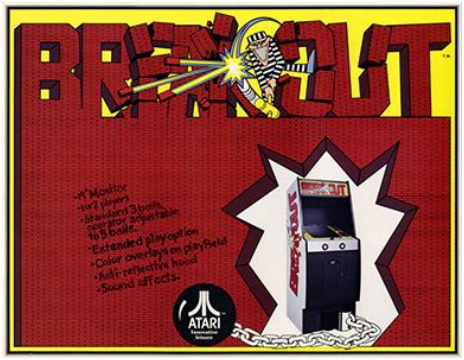
Grafikon 5.1 Rezultat agenta u zavisnosti od epizode

Kod agenta napisan je pomoću C++ jezika i prikazan je u dodatku, poglavlje 8.1.

5.2 Breakout

Breakout je arkadna igrica razvijena od strane Atari kompanije 1976. godine. Koncept igre su osmislili Nolan Bushnell i Steve Bristow, a inspiraciju za igricu su dobili na osnovu prve arkadne igre: Pong. Igru je implementirao Steve Wozniak uz pomoć Steve Jobs-a.

Na slici 5.2 je prikazan prvi flajer za igru *Breakout*.

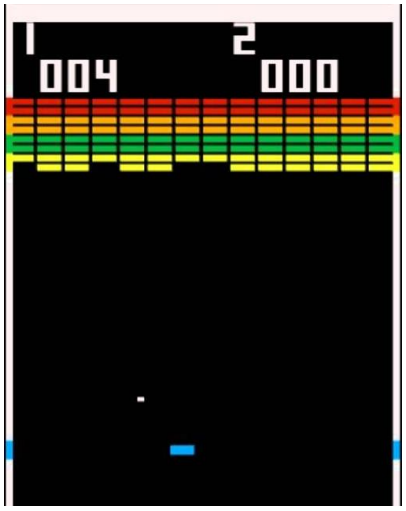


Slika 5.2 Prvi Breakout flajer

Igra se sastoji od 8 slojeva cigli i lopte koja se odbija od njih i od zidova ekrana. Kada lopta pogodi ciglu ona se odbija, a cigla biva uništena. Kada loptica dođe do donjeg dela ekrana igrač „gubi život“. Kako bi igrač sprečio „gubljenje života“ on kontroliše lopaticu koju može pokretati ulevo ili udesno kako bi sprečio lopticu da dodirne donju ivicu ekrana. Loptica se i od lopatice odbija.

Svaka dva sloja cigli imaju drugačiju boju, red boja od najniža dva sloja do najviša dva je: žuta, zelena, narandžasta i crvena. Svaka boja označava broj poena koji agent dobija razbijanjem cigle, tako žute cigle „vrede“ jedan poen, zelene cigle tri poena, narandžaste pet poena i cigle iz najviših slojeva, crvene „vrede“ sedam poena. Igrač ima 3 pokušaja da „očisti dva ekrana cigli“. „Očistiti ekran cigli“ znači srušiti sve cigle koje se nalaze na ekranu, a 3 pokušaja označava da kada loptica treći put dotakne donju ivicu ekrana igra se završava. U svakom sloju postoji 14 cigli, tako da je maksimalni rezultat koji igrač može ostvariti sa jednog ekrana 448, a kako je cilj igre „očistiti“ dva ekrana cigli, to znači da je maksimalni rezultat koji igrač može ostvariti 896. Kako bi dodatno zakomplikovali *Breakout* igru, ljudi iz Atari-a su dodali još par stvari: kada se loptica odbije od gornju ivicu ekrana veličina lopatice se prepolovljuje, brzina loptice se povećava u sledećim trenucima: kada se razbije 4. cigla, kada se razbije 12. cigla, kada se razbiju cigle iz sloja narandžasti ili crvenih.

Na slici 5.3 je prikazan izgled ekrana **originalne** igre.



Slika 5.3 Breakout igra

U zadnjoj rečenici je naglašena reč „originalne“ iz razloga što igra korišćena u ovom radu nije originalna, već portovana originalna za Atari 2600.

Razlika između Atari 2600 *Breakout* igre i prvobitne *Breakout* igre je ta što postoje samo 6 slojeva cigli i ta što igrač ima pet života (umesto tri) da „očisti“ dva ekrana cigli.

Na slici 5.4 prikazan je Atari 2600 *Breakout*.

Slika 5.4 Atari 2600 Breakout

U ovoj igri postoje 3 različite akcije koje agent može da izvrši:
- Levo (LEFT)
- Desno (RIGHT)
- Bez akcije (NOOP)

Broj cigli u jednom redu koji se nalazi u Atari 2600 *Breakout* igri je 18. Kada bi se stanja prikazivala tabelarno, da bi se stanje definisalo potrebno je:
- za svaku ciglu definisati promenljivu da li je cigla „slomljena“ ili ne
- pozicija lopatice
- x pozicija loptice
- y pozicija loptice
- x brzina loptice
- y brzina loptice
- širina lopatice

Cigle se nalaze u šest redova i osamnaest kolona, tako da je broj cigli 18 * 6 = 108. Kako je stanje definisano time da li je cigla „slomljena“ ili ne, za 108 cigli postoji 2^{108} različitih stanja, ~ $3.2 * 10^{32}$.

X i Y koordinata u Atari 2600 može imati najmanju vrednost 1, a najveću vrednost 250.

Loptica se može kretati u tri različite brzine u različitim pravcima, tako da je vrednost brzine u opsegu [-3, +3], odnosno postoje 7 različite vrednosti za brzinu loptice.

Širina lopatice može imati vrednost 8 ili 6, tj. postoje 2 različita stanja.

Broj različitih stanja u kome se agent može naći je:

$$N = 2^{108} * 250 * 250 * 250 * 7 * 7 * 2 = 5 * 10^{41}$$

Već je rečeno da se „stanje – akcija“ par pamti, pošto se koriste Q vrednosti. To znači da se za svako stanje moraju pamtiti 3 različite vrednosti za 3 različite akcije, tako da je broj Q vrednosti koje se treba pamte tabelarno:

$$N_Q = N * 3 = 1.5 * 10^{42}$$

Kako su Q vrednosti iz opsega realnih brojeva, ako bismo ih pamtili pomoću *float* promenljive (4B), da bi smo uspeli sve parove „stanje – akcija“ da čuvamo u memoriji bilo bi nam potrebno:

$$4B * N_Q = 6 * 10^{42} B = 5.6 * 10^{33} GB$$

Jasno je da je nemoguće naći toliku RAM memoriju i koristiti je za treniranje *Breakout* agenta. Zamislimo da je ipak nekako moguće naći toliku RAM memoriju, drugi problem je što se svi parovi „stanje – akcija“ moraju posetiti dovoljan broj puta kako bi agent naučio da, na osnovu stanja u kom se nalazi, izabere akciju koja bi mu donela najviše nagrade u budućnosti. Pomoću ALE okruženja, ako se igra pokreće u vizuelnom modu onda okruženje generiše 60 frejma po sekundi[6], odnosno frekvencijom 60 Hz. Ako je vizuelni mod isključen, tj. ALE se pokreće samo kao konzolna aplikacija, onda radi 100 puta brže, odnosno frekvencijom od 6000 Hz. Zamislimo da agent u svakom frejmu posećuje novo stanje, koje do tada nije posetio (za šta je verovatnoća jednaka nuli), minimalno vreme potrebno da obiđe sva stanja i u svakom stanju da izvrši različitu akciju je:

$$1.5 * 10^{42} / (6000 Hz) = 8.2 * 10^{39} s = 2.6 * 10^{30} godina$$

Iz ovog računa vidi se da je tabelarni metod nemoguće iskoristiti za *Breakout* igru. Najveći udeo kod stanja ima predstavljanje srušenih i nesrušenih cigli, ali taj način predstavljanja cigli je neophodan kako bi agent imao mogućnost da nauči kako da „gađa“ nesrušene cigle.

Zbog ovih ograničenja ne bi bila loša ideja izbaciti faktor srušenih i nesrušenih cigli. Na taj način minimalno vreme potrebno za obilazak svih stanja i potrebna memorija se smanjuje za faktor $3.2 * 10^{32}$. Odnosno, izbacivanjem ovog faktor potrebna RAM memorija bi bila 17.5 GB, a minimalno vreme potrebno za obilazak svih stanja i akcija bi bilo 3 dana. Ova ograničenja su i dalje velika, ali nisu i neizvodljiva. Ali čak i izacivanjem faktora srušenih i nesrušenih cigli, vreme treniranja bi bilo ogromno. Da podsetim „minimalno vreme potrebno za obilazak svih stanja i akcija“ pretpostavlja da se u svakom frejmu agent nalazi u novom nikad viđenom stanju. Teorijski gledano, agent mora svaki par „stanje – akcija“ da poseti **dovoljan** broj puta kako bi naučio optimalnu polisu. To znači da bi treniranje agenta moglo da traje nedeljama, a i čak kada bi agent izračunao optimalnu polisu ona bi opet bila bez „cigli“ faktora. To znači da bi agent naučio samo kako da ne pogine, ali ne i kako da uspešno „gađa“ cigle. To znači da bi se rušenje cigli dešavalo „slučajno“ i zbog toga postoji opasnost da agent upadne u petlju iz koje ne može izaći. Npr. to se može desiti kada ostane samo jedna cigla koju treba srušiti kako bi se „očistio ekran“, a kako je agent naučio samo da „ne gine“ postoji opasnost da agent igra stalno iste poteze na osnovu kojih nikad ne bi poginuo, ali isto tako nikad ne bi ni tu preostalu ciglu srušio.

Zadnja nekoliko pasusa deluju prilično obeshrabrujuće što se tiče implementacija agenta pomoću tabelarne Sarse, ali to nije baš sasvim tačno. Moguće je napraviti agenta koji zna da igra igru Breakout da ne gine, ali ne i da sruši svaku ciglu. To se može postići tako što bi se stanje definisalo samo pomoću relativne razdaljine po Y osi između loptice i lopatice. Ovako definisano stanje je dovoljno agentu da nauči da igra igru da ne pogine, dovoljno je **samo** da lopaticu uvek pozicionira tačno ispod loptice. Na ovaj način, broj različitih stanja je 500 [-250, 250], a kako je broj akcija 3 to znači da postoji 1.500 različitih Q vrednosti koje treba predstaviti tabelarno. U prethodnom poglavlju, za igru *Flappy Bird* agent je imao različitih 1.800 Q vrednosti, i uspeo je za desetak epizoda da izračuna optimalnu polisu. U ovom radu neće biti prikazana implementacija agenta koji pomoću **tabelarne** Sarse igra *Breakout* igru.

[6] Frekvencija kojom ALE generiše frejmove je prikazana u njihovom radu [9], na 255. strani.

Biće prikazani razultati sličnog agenta, tj. biće prikazani rezultati agenta koji ne koristi tabelarni pristup već linearni aproksimator, a jedina karakteristika (*feature*) u vektoru karakteristika, je relativna razdaljina između lopatice i loptice. Na osnovu rezultata, biće jasno da je agent uspeo baš ono što je opisano u ovom pasusu, tj. uspeo je da nauči da „ne gine“, ali ima problem da „očisti ekran“, odnosno ima problema kada na ekranu ostane mali broj (~1) cigli.

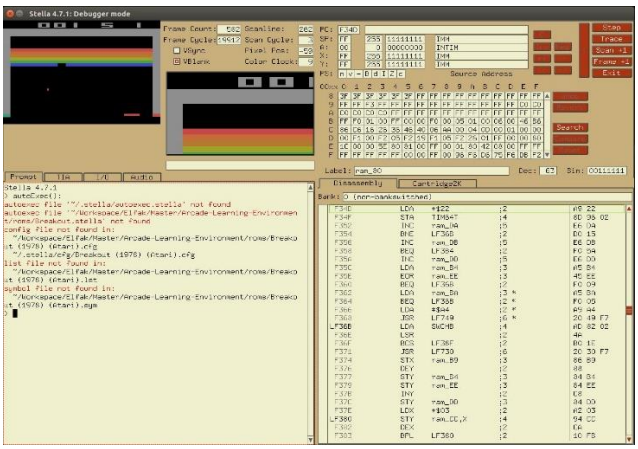
U originalnom ALE kodu postoji podrška za *Breakout* igru, autor ALE-a je definisao samo pozitivnu nagradu za ovu igru, tj. agent dobija pozitivnu nagradu u trenutku kada se cigla sruši, a u svim ostalim trenucima dobija nagradu 0. Druga stvar u originalnom kodu koju sam primetio je problem kod detekcije terminalnog stanja. Terminalno stanje je definisano kada je broj života agenta jednak nuli. Autor verovatno nije pretpostavio da će agent biti u stanju da sruši sve cigle. Taj problem sam rešio u svom kodu, a onda sam prilikom modifikacije drugačije definisao nagradu koju agent dobija. Glavna motivacija za modifikaciju nagrade je ta što sam želeo da „kaznim“ agenta kada izgubi život. Pored kazne kada agent izgubi život, tokom testiranja primetio sam da agent u nekim trenucima zna da se zaglavi u petlji. Kako bi sprečio pojavljivanje petlji, dodatna kazna je uvedena u trenutku kada agent udari lopticu, a da između tog udarca i prethodnog nije srušio nijednu ciglu.

Ukratko, nagrade i kazne za agenta koji igra Breakout su:

- pozitivna nagrada svaki put kada se sruši cigla (vrednost nagrade je ista kao i vrednost cigle)
- negativna nagrada od -10[7] kada agent „pogine“
- -1 negativna nagrada ako agent između dva udarca ne sruši nijednu ciglu

5.2.1 Odabir karakteristika

Za igru *Breakout* opisano je kako stanja treba definisati, kao i opseg mogućih vrednosti za svaku karakteristiku, ali i nije objašnjeno kako dobiti informacije o trenutnom stanju igre. Već je napomenuto da ALE pruža mogućnost čitanja stanja ROM memorije na određenoj lokaciji, tako da je potrebno samo naći na kojim adresama se nalaze koje informacije. Ove adrese nisam uspeo da pronađem, tako da sam morao sam da ih odredim samostalno empirijski. To se može uraditi pomoću Stella emulatora i Stella *debugger*-a. Na slici 5.5 je prikazan izgled Stella *debugger*-a.



Slika 5.5 Izgled Stella debugger-a

[7] Vrednost od -10 je izabrana kako bi se dao veći značaj kazni kada agent „pogine“ umesto kada ne sruši nijednu ciglu između dva udarca. Tačna vrednost od -10 je nasumično izabrana.

Nakon *debug*-iranja i igranja igre nekoliko sati pomoću Stella-e, odredio sam potrebne memorijske lokacije. U tabeli 5.2 su prikazani pojmovi, adrese na kojim se nalaze informacije i moguće vrednosti koje se mogu naći na tim memorijskim lokacijama:

Pojam	Adrese	Moguće vrednosti
Rezultat	**{CD, CE}**	**[0, 10]**
Informacije o ciglama	**[80, A3]**	**[0, 255]**
Brzina loptice	**{E7, E9}**	**[-3, 3]**
Širina lopatice	**EC**	**{6, 8}**
Pozicija lopatice	**C8**	**[1, 250]**
Y pozicija loptice	**E3**	**[1, 250]**
X pozicija loptice	**E5**	**[1, 250]**

Tabela 5.2 Pojmovi, adrese i moguće vrednosti za Breakout

Ukupan broj karakteristika je 45[8].

Kada sam pokušao naivno da uključim ove karakteristike pomoću linearnog aproksimatora nisam uspeo da napravim agenta koji može da generaliziju, tj. agent nije uspeo da nauči nikakvu taktiku, nakon desetine hiljade odigranih epizoda agent se i dalje ponašao kao na početku i u kom god stanju se nađe uvek je birao akciju Levo (LEFT). Zaključak do koga sam došao u tom trenutku je da agent dobija previše informacija na osnovu kojih ne uspeva sam da zaključi koje karakteristike su mu najvažnije. Pokušavao sam sve parametre da promenim kako bi došao do nekog napretka, ali koliko god menjao α, γ, ε i n (broj koraka u Sarsa algoritmu) agent je uvek birao akciju Levo. Do sličnog rezultata i zaključka su došli sa Stanford univerziteta u njihovom radu [12].

Dodavanje samo još jedne karakteristike rešilo mi je ove probleme, a ta dodatna karakteristika je „relativna udaljenost između lopatice i loptice“. Jednostavno dodavanje te karakteristike i dalje nije doprinelo bržem učenju. Tako da sam morao da „naglasim“ linearnom aproksimatoru da je ova dodata karakteristika „značajnija“ za učenje od ostalih karakteristika. To se postiže tako što je opseg mogućih vrednosti za ovu karakteristiku veći nego opseg za ostale karakteristike, odnosno vrednost ostalih karakteristika treba samo podeliti konstantom. Odabir te konstante značajno može doprineti brzini učenja agenta, a rezultati će biti predstavljeni kasnije u radu.

Prethodno je u radu rečeno da postoji 108 cigli i ostalih karakteristika, a sad je navedeno da se koriste 46 karakteristike (dodatna karakteristika „relativna udaljenost“). To je zato što se za te 108 cigle koriste samo 36 memorijske lokacije i umesto 108 različitih karakteristika koje bi imale vrednost 0 ili 1, koristi se 36 različitih karakteristika, a za vrednost se prosleđuje vrednost koja je upisana na odgovarajućoj adresi. Na ovaj način se broj karakteristika značajno smanjuje, a i lakša je implementacija s obzirom da se samo pročita vrednost sa određene lokacije i prosledi se kao karakteristika (*feature*).

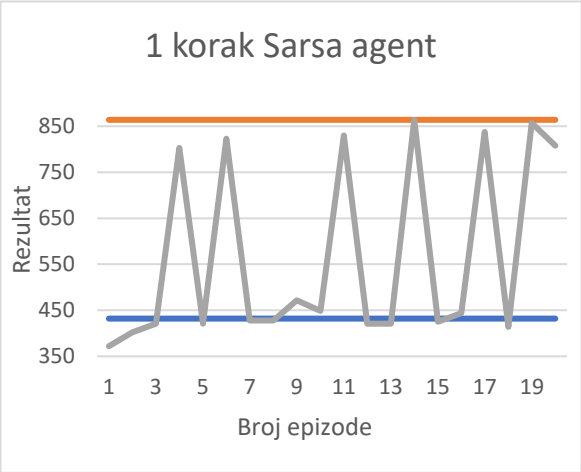
[8] U tabeli 6.2 se nalaze 44 različite karakteristike. Kod linearne aproksimacije dodaje se još jedna karakteristika koja uvek ima konstantu vrednost 1, ta karakteristika je poznata kao i „bias term“ i dodaje se zato što funkcija koja se aproksimira ne mora da prolazi kroz koordinatni početak.

5.2.2 Rezultati

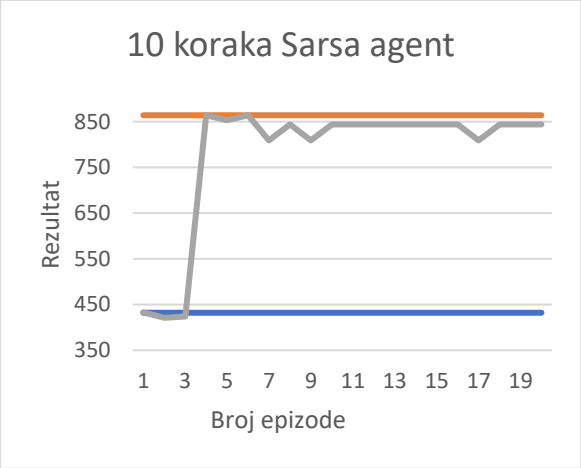
U ovom odeljku se nalaze rezultati koje Sarsa agent postiže igrajući Breakout igru. Na grafikonima biće predstavljeni rezultati u zavisnosti od raznih parametara:

- broj koraka Sarsa algoritma
- α parametar
- γ *discount* faktor
- opseg vrednosti karakteristika (*feature*-a)

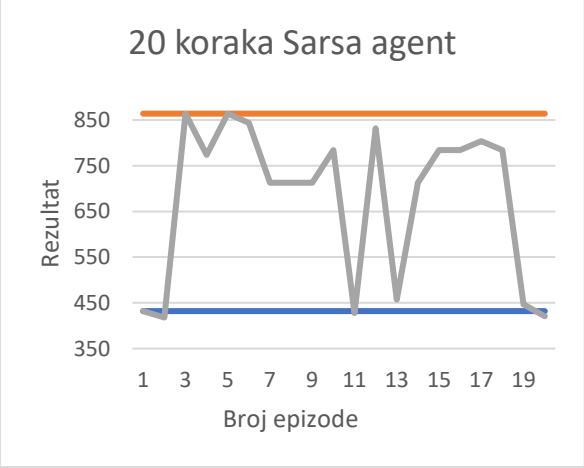
Na grafikonima 5.2, 5.3, 5.4 i 5.5 se nalaze rezultati koje agent ostvaruje u zavisnosti od broja koraka Sarsa algoritma. Na grafikonima se nalaze brojevi koraka: 1, 10, 20 i 100 respektabilno. Grafikoni sadrže granice, donja granica predstavlja rezultat koji agent ostvaruje kada očisti jedan ekran (432), a gornja granica predstavlja maksimalan rezultat koji agent može ostvariti igrajući *Breakout* igru (864).



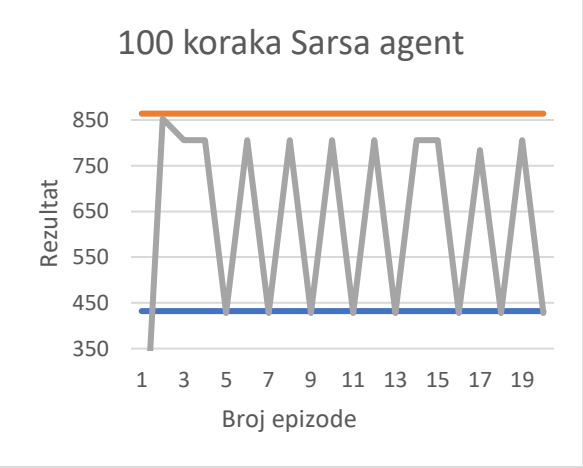
Grafikon 5.2 1 korak



Grafikon 5.3 10 koraka



Grafikon 5.4 20 koraka

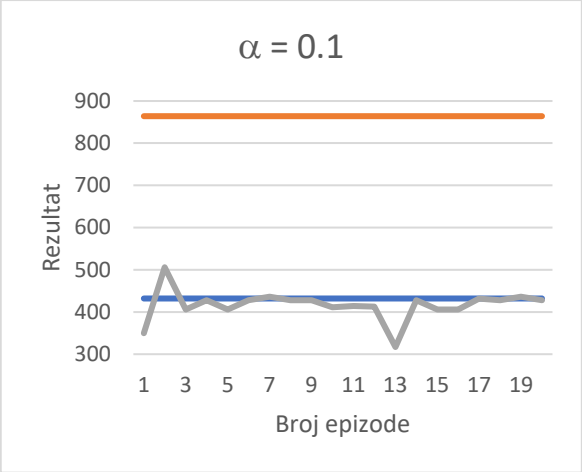


Grafikon 5.5 100 koraka

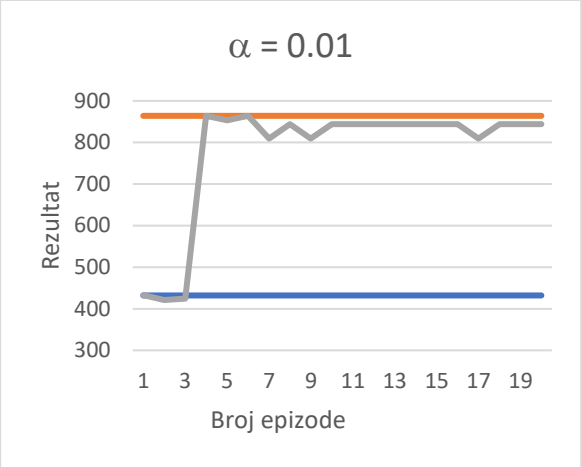
Na osnovu grafikona se može zaključiti da agent najbolji rezultat ostvaruje za algoritam „u 10 koraka“ (grafikon 5.3). To znači da agent odigra 10 akcija pre nego što modifikuje svoju polisu. Rezultati se razlikuju u zavisnosti od broja koraka, dok su drugi parametri:

α = 0.01
γ = 0.95

Grafikon 5.6, 5.7 i 5.8 prikazuju rezultat koji ostvaruje agent u zavisnosti od α (*learning rate*) faktora, dok su broj koraka (N = 10) i *discount* faktor (γ = 0.95) konstantni. Na grafikonima se nalaze 3 različita α faktora: 0.1, 0.01 i 0.001 respektabilno.

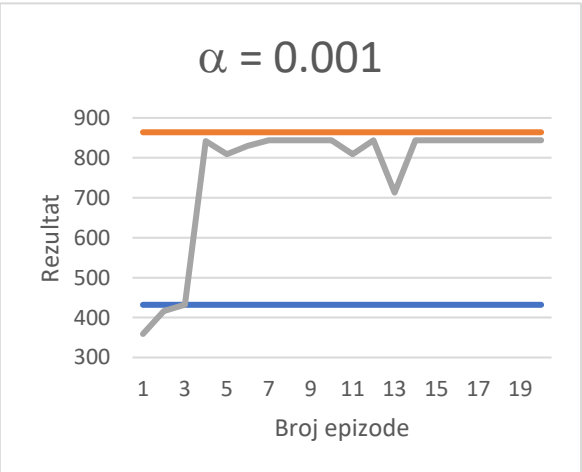


Grafikon 5.2 Korak učenja 0.1



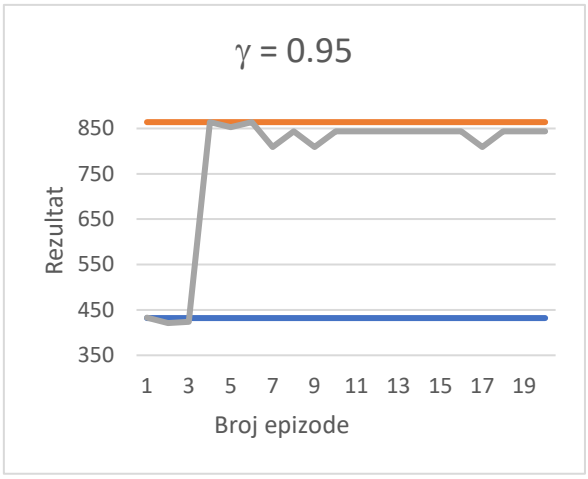
Grafikon 5.3 Korak učenja 0.01

Na osnovu prikazanih grafikona može se zaključiti da kada je *learning rate* 0.1 (grafikon 5.6), onda agent prebrzo „briše" staro znanje novim i zato ne uspeva da ostvari neki respektabilan rezultat, npr. rezultat veći od 800. Dok za *learning rate* 0.001 (grafikon 5.8), agent predugo „zadržava" staro znanje. Iako se ovaj agent bolje ponaša nego kada je vrednost 0.1, svejedno ne uspeva nijednom da očisti oba ekrana, tj. ne uspeva da ostvari maksimalan rezultat od 864.

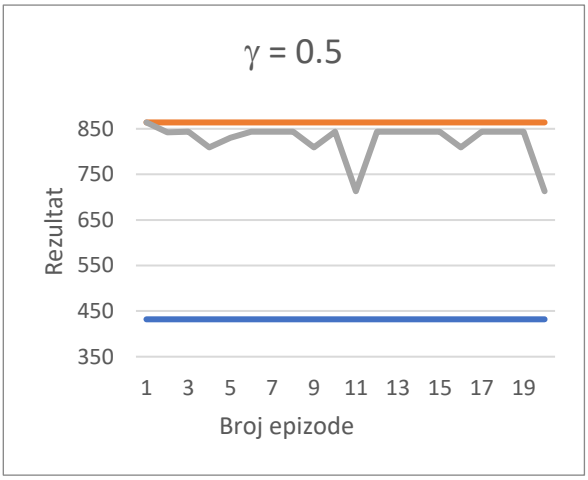


Grafikon 5.4 Korak učenja 0.001

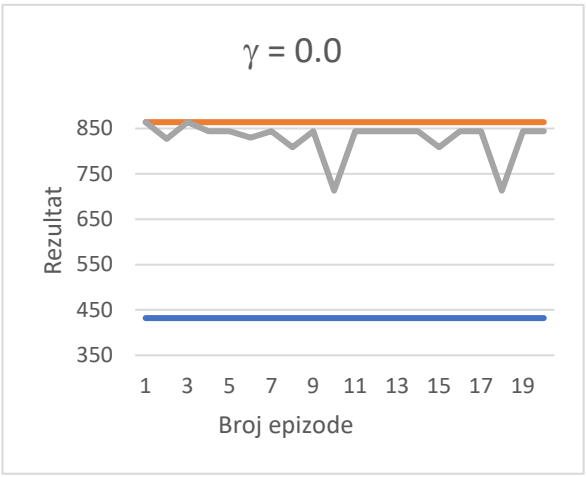
Sledeća četiri grafikona 5.9, 5.10, 5.11 i 5.12, ilustruju rezultat koji ostvaruje agent u zavisnosti od *discount* faktora. Odnosno, u zavisnosti od toga koliko agent vrednuje nagradu koju će dobiti u budućnosti u odnosu na trenutnu nagradu. Na ova 4 grafikona γ je promeljiva, dok su *learning rate* (α = 0.01) i broj koraka (N = 10) konstantni.



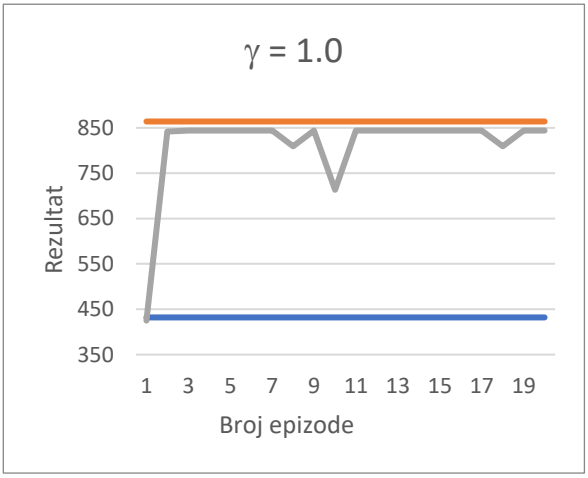
Grafikon 5.6 Discount 0.95



Grafikon 5.5 Discount 0.5

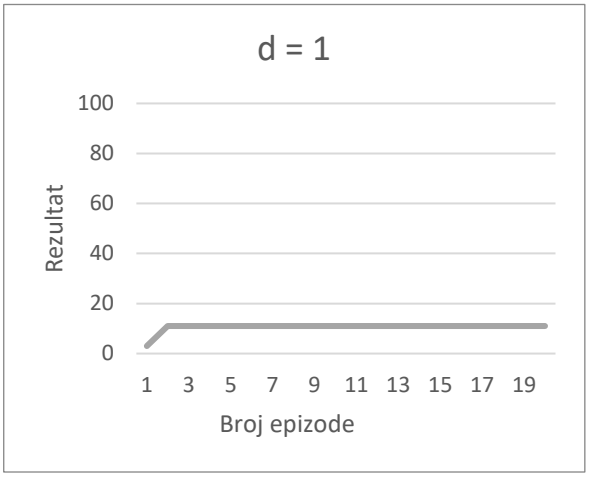


Grafikon 5.8 Discount 0.0

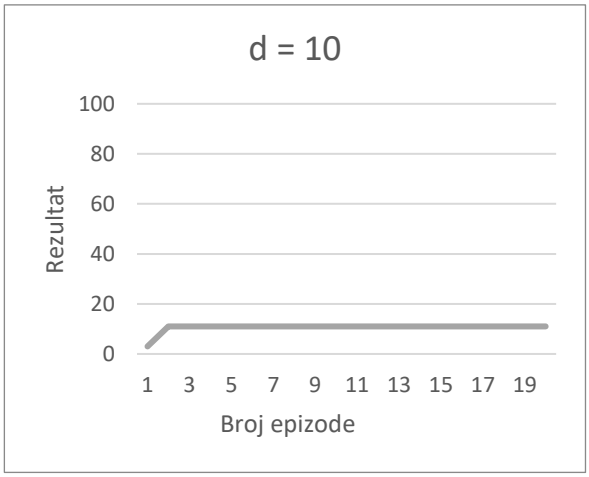


Grafikon 5.7 Discount 1.0

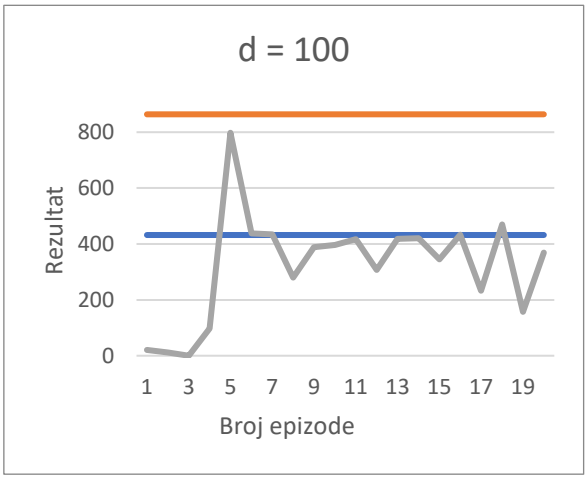
Na kraju su ostali grafikoni koji ilustruju rezultat koji agent ostvaruje u zavisnosti od opsega vrednosti karakteristika. U poglavlju 5.2.1 je naglašeno da „relativna udaljenost između lopatice i loptice“ karakteristika ima veći opseg nego ostale karakteristike. Rečeno je da se vrednost ostalih karakteristika deli nekom konstantom većom od 1 kako bi se smanjio opseg tih ostalih karakteristika, brzina učenja agenta je u velikoj zavisnosti u odnosu na tu konstantu. Neka ime te konstante bude **d**, sledeća pet grafikona (5.13, 5.14, 5.15, 5.16 i 5.17) ilustruju rezultat kada je **d**: 1, 10, 100, 1.000 i 10.000. *Learning rate* ($\alpha = 0.01$), *discount* faktor ($\gamma = 0.5$) i broj koraka ($N = 10$) su konstantni na sledećim grafikonima.



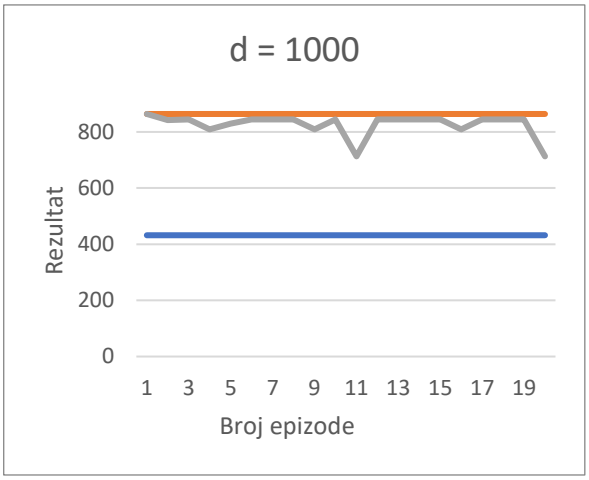
Grafikon 5.13 Razlika opsega vrednosti 1



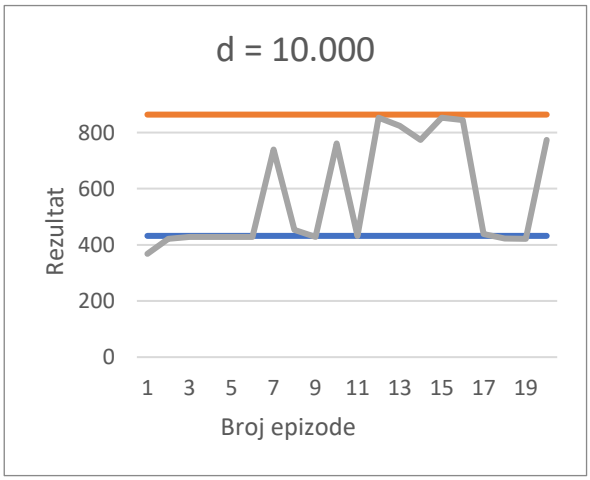
Grafikon 5.14 Razlika opsega vrednosti 10



Grafikon 5.15 Razlika opsega vrednosti 100



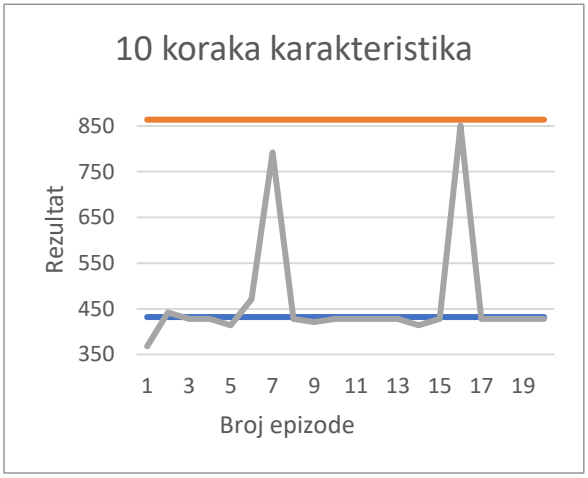
Grafikon 5.16 Razlika opsega vrednosti 1000



Grafikon 5.17 Razlika opsega vrednosti 10.000

Na osnovu grafikona 5.13 – 5.17 se vidi da u zavisnosti kako se izabere parametar **d**, rezultati najviše variraju. Kada **d** ima nisku vrednost, (1 - 10), onda linearni aproksimator ne zna da zaključi koje su karakteristike najvažnije i agent ne uspeva da dođe do optimalne polise. A kada je **d** vrednost prevelika (grafikon 5.17) karakteristika „relativna udaljenost između lopatice i loptice“, je previše dominantna u odnosu na ostale karakteristike, pa agent ne uspeva konstantno da beleži velike rezultate (~800). Isti problem se dešava kada vektor karakteristika sadrži samo spomenutu karakteristiku bez ostalih, tj. kada se u vektor karakteristika ne uključuju karakteristike kao: srušene i nesrušene cigle, veličina lopatice, brzina loptice, pozicija loptice... Agent u ovom slučaju odlučuje „samo“ na osnovu „relativne udaljenosti između loptice i lopatice“, pa su rezultati slični kao i kad druge karakteristike i ne postoje. Primećuje se da agent „ima problem“ da očisti jedan ekran cigli. To je zato što kada ostane mali broj cigli na ekranu, agent ne zna kako da pozicionira lopaticu da bi ih srušio, već se njihovo rušenje dešava slučajno. Dakle, kao što je već rečeno, agent se isto ponaša kao da druge karakteristike ne postoje. Sledeća tri grafikona prikazuju rezultate koje agent ostvaruje kada se u vektor karakteristika nalazi samo karakteristika „relativna udaljenost između lopatice i loptice“. Za primer sam uzeo da je samo broj koraka promenljiva, dok su *learning rate* (α = 0.01) i *discount* faktor (γ = 0.5) konstantni.

Grafikon 5.18 Sarsa u 1 korak



Grafikon 5.99 Sarsa u 10 koraka

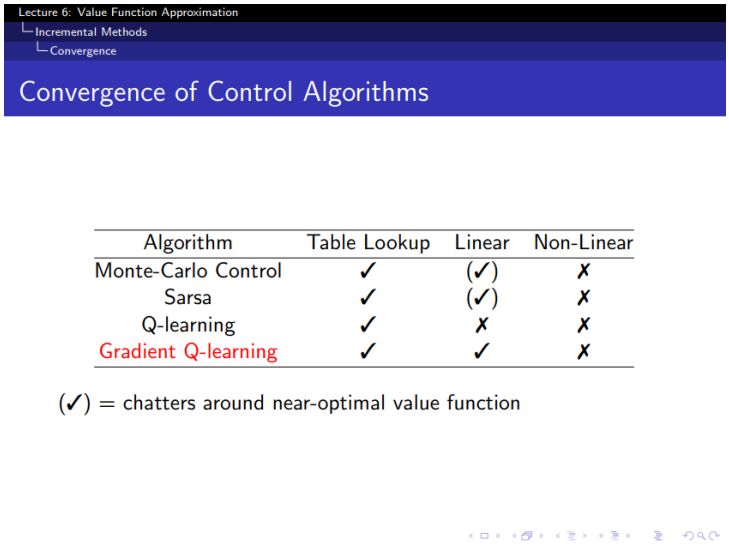
Ako se uporede grafikoni 5.19 i 5.16 vidi se da u prvom agent veoma retko uspeva (4/20) da sruši prvi ekran cigli, dok u drugom agent beleži odlične rezultate. Jedina razlika između ta dva algoritma je što se u prvom slučaju koristi samo jedna, već pomenuta, karakteristika dok u drugom slučaju agent koristi sve karakteristike potrebne za igranje *Breakout* igre.

Grafikon 5.10 Sarsa u 20 koraka

Na osnovu prikazanih rezultata i grafikona, agent je uspeo u onome šta mu je bila namena, da igra *Breakout* igru bolje nego prosečan čovek. U radu koji su objavili iz firme *DeepMind* [8], prikazani su rezultati koje ostvaruju razni algoritmi i rezultati koje ostvaruje prosečan čovek. U njihovom radu prosečan čovek (nakon dva sata igranja) beleži prosečan rezultat 31, *Deep Q-Learning* algoritam, nakon 2 sata igranja, je zabeležio prosečan rezultat od 168. Četiri grafikona koji pokazuju rezultate agenta u zavisnosti od *discount* faktora prikazuju da je, na 3 od 4 grafikona, prosečan rezultat veći od 800. Rezultati su dobijeni na osnovu prvih 20 odigranih epizoda.

Međutim, algoritam je specijalno napravljen za *Breakout* igru, nije u pitanju generalna inteligencija koja može naučiti da igra sve Atari 2600 igre (dok DQN algoritam jeste). Na osnovu svih prikazanih grafikona, vidi se da agent ne konvergira ka optimalnoj polisi, već agent osciluje oko optimalne polise (na nekim grafikonima su te oscilacije minimalne, dok na nekim se vidi da oscilacije mogu biti prilično velike).

RL algoritmi za kontrolu (*Q-Learning* i Sarsa) imaju problem konvergencije kada se koristi aproksimator funkcije. Sarsa algoritam sa linearnim aproksimatorom osciluje oko optimalne polise, dok *Q-Learning* sa linearnim aproksimatorom i oba algoritma sa nelinearnim aproksimatorom divergiraju. Taj problem se dešava zato što podaci koji dolaze do agenta (i aproksimatora) su u velikoj međusobnoj korelaciji. O ovom problemu je pričao i David Silver u svom predavanju [17]. Na slici 5.6 se nalazi prikaz konvergencije raznih algoritama sa aproksimacionom funkcijom.



Slika 5.6 Konvergencija algoritma. Izvor [17].

Kada se kaže da algoritam ne konvergira (divergira ili osciluje) to ne znači da algoritam nikad neće konvergirati. Samo znači da može doći do divergencije ili oscilacije. Postoje različiti mehanizmi koji se mogu iskoristiti kako bi algoritam konvergirao. *Deep Q-Learning* algoritam koristi sledeće mehanizme: dugo vreme koristi jednu polisu za treniranje a drugu za igranje, svi trening podaci se čuvaju kako bi se više puta iskoristili za treniranje, trening podaci koji se čuvaju se promešaju pre nego da se ponovo propuste kroz aproksimator funkcije. Koristeći ove mehanizme, programeri iz kompanije DeepMind-a su uspeli da izbegnu divergenciju *Q-Learning* algoritma, tj. napravili su agenta koji konvergira. U ovom radu nisam koristio nijedan pomenut mehanizam, i zbog toga se dešava oscilacija oko optimalne polise.

Pored pomenutih mehanizama, postepeno smanjivanje α faktora može doprineti konvergenciji algoritma. *Learning rate* faktor kreće od relativno visoke vrednosti i postepeno

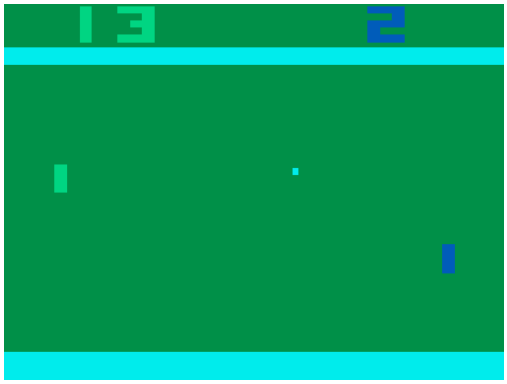
se smanjuje do a ~ 0. Na ovaj način se algoritmu naglašava da su prva iskustva veoma važna, dok su kasnija iskustva (kada bi agent već bio blizu optimalne polise) sve manje i manje važna. Ovaj mehanizam ne sprečava oscilaciranje agenta, samo ga postepeno ublažava, pa se čini da agent konvergira.

Kod agenta napisan je pomoću C++ jezika i prikazan je u dodatku, poglavlje 8.2.

5.2.3 Pong

Jedna od prvih video igara ikada napravljenih je igra Pong, razvijena od strane Atari kompanije 1972. godine. Pong predstavlja simulaciju stoni tenis igre. Na ekranu igrice postoje dva igrača, tj. dve lopatice, jednu kontroliše agent iz Pong igre, a drugu lopaticu kontroliše igrač. Dva igrača igraju igru, koja se završava kad neko od igrača ostvari 21. poen. Poen se ostvaruje tako što protivniku loptica prođe iza lopatice. Izgled ekrana Pong igre se nalazi na slici 5.7.

Akcije koje agent može izvršiti su slične kao i kod *Breakout* igre, samo umesto akcija Levo i Desno, u Pong igri postoje akcije Gore (UP) i Dole (DOWN).



Slika 5.7 Atari 2600 Pong

Implementacija agenta koji igra Pong igru je slična implementaciji agenta koji igra *Breakout* (jedan od razloga zašto Pong nije izdvojen u posebnom poglavlju). Pored implementacije, i korišćeni parametri su isti kao i kod *Breakout* agenta:

- Broj koraka = 10
- γ discount faktor = 0.95
- α learning rate faktor = 0.01

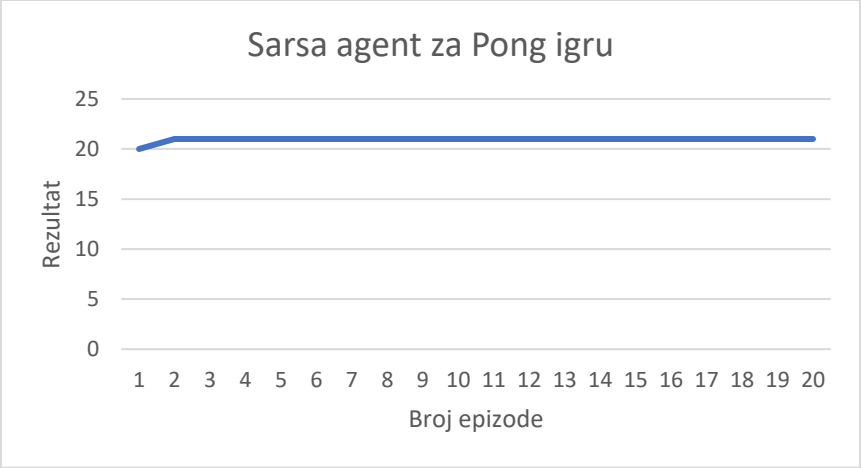
Jedina razlika između ova dva agenta je kod vektora karakteristika, već je napomenuto da je *Brekaout* agent imao 46 karakteristika, a Pong agent koristi samo dve karakteristike. Analizom Pong igre uočene su sledeće karakteristike koje se mogu iskoristiti:

- X pozicija loptice
- Y pozicija loptice
- Pozicija lopatice
- Pozicija protivnikove lopatice
- X brzina loptice
- Y brzina loptice

Kao što je već rečeno u poglavlju 3.4.3 odabir karakteristika, veoma je važan za brzinu učenja agenta, a značajne su i odabrane karakteristike. Daljom analizom igrice može se primetiti da pozicija protivnikove lopatice uvek prati Y komponentu pozicije loptice, tako da

je ta karakteristika redudantna. A što se tiče ostalih 3 karakteristika, one se mogu spojiti u jednu: „relativna udaljenost lopatice od loptice“ (ista karakteristika iskorišćena je i kod *Breakout* igre). Tako da umesto 4 karakteristike može se koristiti samo jedna. Upravo takav vektor karakteristika je i konstruisan za veštačkog agenta, plus naravno karakteristika +1 kao *bias term*.

Ovakva implementacija agenta je postigla neverovatno dobre rezultate. U prvoj rundi prve epizode, agent gubi zato što nema nikakvo iskustvo i uvek bira akciju NOOP, u drugoj rundi agent bira akciju Dole i tu ostvaruje prvi poen. Nakon toga agent više nije izgubio nijednu rundu. Ukratko rečeno, agent u prvoj epizodi pobeđuje 21:1 a u svakoj sledećoj pobeđuje 21:0. Na sledećem grafikonu je ilustracija koja pokazuje performanse ovog agenta.



Grafikon 5.11 Performanse Sarsa agent za Pong igru

6 ZAKLJUČAK

U ovom samostalnom radu imali smo prilike da vidimo osnovne stvari *Reinforcement Learning*-a. Objašnjeno je kako se RL razlikuje od, do sada veoma popularnog, nadgledanog učenja. U radu je prikazano kako su osnovni koncepti RL-a bazirani na stvarima iz realnog života, odnosno način kako agent uči na osnovu iskustva (nagrade i kazne), kako se razrešava dilema da li je bolje pokupiti nagradu odmah ili je bolje odložiti trenutnu nagradu zbog veće potencijalne nagrade u budućnosti.

TD algoritam za predikciju zajedno sa Sarsa algoritmom za kontrolu, daju veoma efikasan mehanizam za računanje optimalne polise, a korišćenjem aproksimacione funkcije ovaj mehanizam se može iskoristiti za rešavanje ozbiljnijih RL problema. Efikasnost Sarsa algoritma je prikazana u 5. poglavlju na 3 različite igre. Za dve od tri igre (*Flappy Bird* i Pong) agent je uspeo da dođe do optimalne polise za manji broj epizoda nego što je potrebno prosečnom čoveka. Kod *Breakout* igre, agent osciluje oko optimalne polise (nije dobijena konvergencija), ali i pored tog oscilovanja, agent je uspeo da postigne bolje rezultate nego prosečan čovek.

Tabelarna Sarsa se pokazala kao efikasan algoritam sa jednostavnom implementacijom. Za relativno mali broj stanja (par hiljada) agent veoma brzo dolazi do optimalne polise. Ali, ovaj pristup ima već naveden problem, u nekim primerima sa velikim brojem stanja (kao npr. za *Breakout* igru) memorijska i vremenska ograničenja kod tabelarnog pristupa su toliko velika da je nemoguće implementirati agenta.

Sarsa algoritam sa linearnim aproksimatorom se pokazao još efikasnije, agent je već u prvim epizodama (i kod igre Pong i *Breakout*) postizao gotovo maksimalan rezultat. Generalizacija koju omogućava linearni aproksimator predstavlja veoma efikasan mehanizam koji omogućava agentu brzo učenje, mnogo brže nego kod tabelarne metode. Ali to brže učenje nije bez mana.

Teorijski je objašnjeno kako odabir karakteristika može uticati na linearni aproksimator, a i prikazano je kod igre *Breakout* kako na osnovu naivno odabranih karakteristika agent ne može da „zaključi“ koje su ključne karakteristike za igranje igre. Drugi problem kod linearnog aproksimatora je konvergencija polise. Agent implementiran za igru *Breakout* osciluje oko optimalne polise, ali i pored tih oscilacija, agent uspeva da postigne mnogo bolje rezultate nego prosečan čovek.

Za sada se RL uglavnom još bazira na istraživanju, upravo iz tog razloga se najveći broj radova i dalje bavi igranjem video igrica. Tu se RL pokazao veoma dobro, tako da se očekuje u budućnosti da RL ozbiljnije krene u rešavanje nekih konkretnijih problema.

7 LITERATURA

[1] Silver D., Reinforcement learning course, Lecture 1: Introduction to Reinforcement learning. Dostupno na: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/intro_RL.pdf (pristupljeno 26. avgusta 2017.).

[2] Sutton Richard S. and Andrew G. Barto. Reinforcement learning: An introduction. Vol. 1. No. 1. Cambridge: MIT press, 1998.

[3] Web-site https://medium.com/@dennybritz/exploration-vs-exploitation-f46af4cf62fe (pristupljeno 28. avgusta 2017.).

[4] Silver D., Reinforcement learning course, Lecture 2: Markov Decision Processes. Dostupno na: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf (pristupljeno 29. avgusta 2017.).

[5] Silver D., Reinforcement learning course, Lecture 4: Model-Free Prediction. Dostupno na: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MC-TD.pdf (pristupljeno 31. avgusta 2017.).

[6] Silver D., Reinforcement learning course, Lecture 5: Model-Free Control. Dostupno na: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/control.pdf (pristupljeno 31. avgusta 2017.).

[7] Silver D., Reinforcement learning course, Lecture 6: Value function approximation. Dostupno na: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/FA.pdf (pristupljeno 03. septembra 2017.).

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[9] M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents, Journal of Artificial Intelligence Research, Volume 47, pages 253-279, 2013.

[10] Bhonker N., Rozenberg S. and Hubara I. Playing SNES in the Retro Learning Environment. ArXiv preprint arXiv:1611.02205, 2016.

[11] Hershey, D., & Wulfe, B. Learning to Play Atari Games.

[12] Berges, V. P., Rao, P., & Pryzant, R. Reinforcement Learning for Atari Breakout.

[13] Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems (Vol. 37). University of Cambridge, Department of Engineering.

[14] https://en.wikipedia.org/wiki/Flappy_Bird (pristupljeno 02. novembra 2017.).

[15] https://en.wikipedia.org/wiki/Breakout_(video_game) (pristupljeno 02. novembra 2017.).

[16] https://stella-emu.github.io/docs/index.html (pristupljeno 06.11.2017.)

[17] Silver D., Reinforcement learning course, Lecture 6: Value function approximation. Dostupno na: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/FA.pdf (pristupljeno 09.11.2017.)

8 DODATAK: IZVORNI KOD AGENTA

8.1 Flappy Bird kod

```
namespace object_model
{
class State
{
  public:
    int height_difference;
    int bird_direction;

    State(int height_difference, int pipe_type, int bird_direction)
        : height_difference(height_difference), bird_direction(bird_direction)
      {}

    State(int index)
        : height_difference(index % 300), bird_direction(index / 300)
      {}

    State(int relative_height, int bird_direction)
        : height_difference(relative_height), bird_direction(bird_direction)
      {}

    int ToIndex()
    {
        return height_difference + bird_direction * 300;
    }

    void FromIndex(int index)
    {
        height_difference = index % 300;
        bird_direction = index / 300;
    }
};
...
#define NUMBER_OF_STATES 1800
#define EXPLORATION 20


namespace object_model
{
class Policy
{
protected:
  Action _actions[NUMBER_OF_STATES];
  double q[NUMBER_OF_STATES * NUMBER_OF_ACTIONS];
```

```
  int times_visited[NUMBER_OF_STATES * NUMBER_OF_ACTIONS];
  std::vector<std::pair<State, Action>> _history;
  std::vector<double> reward_history;
  double epsilon = 1 / EXPLORATION;
  double alpha = 0.01;
  const double discount = 0.8;
...
#define TD_N 2

namespace object_model
{
class Sarsa : public Policy
{
...
bool Sarsa::EvaluateAndImprovePolicy(double reward, bool isFinal)
{
    int numOfPrevSteps = _history.size();
    pair<State, Action> currStateAction = _history[numOfPrevSteps - 1];
    Action currAction = (Action)currStateAction.second;
    State currState = (State)currStateAction.first;
    int currStateIndex = currState.ToIndex();
    int currQIndex = currAction * NUMBER_OF_STATES + currStateIndex;
    Action previousAction;

    if (numOfPrevSteps > TD_N && !isFinal)
    {
        double gama = 1;
        int currentIndex = currQIndex;
        for (int i = 0; i < TD_N; i++)
        {
            gama *= discount;
            int index = numOfPrevSteps - 2 - i;

            currentIndex = EvalueTD(reward, index, currentIndex, gama);
        }
    }
    else if (isFinal)
    {
        //eval
        q[currQIndex] = q[currQIndex] + alpha * (reward - q[currQIndex]);
        //improve
        previousAction = _actions[currStateIndex];
    }

    return _updated;
}
```

```
int Sarsa::EvalueTD(double reward, int index, int currentIndex, double gama)
{
    pair<State, Action> prevStateAction = _history[index];
    Action prevAction = (Action)prevStateAction.second;
    State prevState = (State)prevStateAction.first;
    int stateIndex = prevState.ToIndex();
    int prevQIndex = prevAction * NUMBER_OF_STATES + stateIndex;

    //eval
    q[prevQIndex] = q[prevQIndex] + alpha *
                    ((reward + gama * q[currentIndex]) - q[prevQIndex]);

    //improve
    Action previousAction = _actions[stateIndex];
    return prevQIndex;
}
Action Policy::GetAction(State s)
{
    int index = s.ToIndex(); //calculate index from state
    Action a = _actions[index];

    double r = ((double)rand() / (RAND_MAX));

    if (r < epsilon)
    {
        double randAction = ((double)rand() / (RAND_MAX));
        a = randAction > 0.5 ? JUMP : NOOP;

        cout << "Random action taken. Action = " << a << endl;
    }

    times_visited[NUMBER_OF_STATES * a + index]++;
    _history.push_back(pair<State, Action>(s, a));
    return a;
}
...
for (int episode = 0;; episode++)
    {
    ...
        while (!rle.game_over())
        {
            State currentState(relative_height, direction);
            rle::Action a = policy->GetAction(currentState);
            double reward = rle.act(a);
            policy->EvaluateAndImprovePolicy(reward, rle.game_over());
            policy->AddRewardToHistory(reward);
            totalReward += reward;
        }
```

```
        cout << "Episode " << episode << " ended with score: " << totalReward;
        rle.reset_game();
    }
```

8.2 Breakout kod

```
#define NUMBER_OF_ACTIONS 3
#define NUMBER_OF_FEATURES 46
#define TD_N 10

//score: CD, CE
//brick addresses from: 80-A3
//paddle position: C6, C8
//ball info: E3, E5, E7, E9

using namespace std;

namespace object_model
{
static const unsigned char addresses[] =
{
0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8A, 0x8B, 0x8C,
0x8D, 0x8E, 0x8F, 0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99,
0x9A, 0x9B, 0x9C, 0x9D, 0x9E, 0x9F, 0xA0, 0xA1, 0xA2, 0xA3, 0xC6, 0xC8, 0xE1,
0xE3, 0xE5
};
static const unsigned char ballSpeed[] = { 0xE7, 0xE9 };
static const unsigned char paddleWidth = 0xEC;
class Sarsa
{
private:
  double weights[NUMBER_OF_ACTIONS][NUMBER_OF_FEATURES];
  double features[NUMBER_OF_FEATURES];
  double alpha;
  double discount;
  double history[TD_N][NUMBER_OF_FEATURES];
  double rewards[TD_N];
  int featureHistory[NUMBER_OF_FEATURES][256];
  int actions[TD_N];
  int historyIndex;
  double maxReward;

public:
  Sarsa();
  Action GetAction();
  void UpdateWeights(double reward, Action chosenAction,
                     bool isFinal, bool onlyPlay);
```

```
  void StoreFeatures(int action, double reward)
  {
    for (int i = 0; i < NUMBER_OF_FEATURES; i++)
    {
      history[historyIndex][i] = features[i];
    }
    actions[historyIndex] = action;
    rewards[historyIndex] = reward;
    historyIndex++;
  };

  void ReadFeatures(ALERAM aleRam)
  {
    unsigned char romVal = 0;
    int i = 0;
    for (; i < NUMBER_OF_FEATURES - 5; i++)
    {
      romVal = aleRam.get(addresses[i]);
      featureHistory[i][romVal]++;
      features[i] = romVal / 255000.0f;
    }

    char paddleWidthVal = (char) aleRam.get(paddleWidth);
    char speedX = (char)aleRam.get(ballSpeed[0]);
    char speedY = (char)aleRam.get(ballSpeed[1]);

    features[i++] = speedX / 1000.0f;
    features[i++] = speedY / 1000.0f;

    int player_y_pos = aleRam.get(0xC8);
    int ball_y_pos = aleRam.get(0xE3) - paddleWidthVal;
    int hdiff = player_y_pos - ball_y_pos;

    features[i++] = paddleWidthVal / 2000.0f;
    features[i] = hdiff / 255.0f;
  };

  void PrintWeights();
  void FlushToDisk(char *filename, double totalReward);
  void LoadFromDisk(char *filename);
};
}
```

```
Sarsa::Sarsa()
{
    for (int i = 0; i < NUMBER_OF_FEATURES; i++)
    {
        features[i] = 0.001;
        for (int j = 0; j < NUMBER_OF_ACTIONS; j++)
        {
            weights[j][i] = 0.0;
        }
        for (int j = 0; j < 256; j++)
            featureHistory[i][j] = 0;
    }
    for (int i = 0; i < TD_N; i++)
    {
        actions[i] = 0;
        rewards[i] = 0.0;
        for (int j = 0; j < NUMBER_OF_FEATURES; j++)
        {
            history[i][j] = 0.0;
        }
    }

    historyIndex = 0;
    alpha = 0.01;
    discount = 0.95;
    maxReward = 0.0;
}

Action Sarsa::GetAction()
{
    double maxQ = -std::numeric_limits<double>::max();
    int maxIndex = 0;

    for (int i = 0; i < NUMBER_OF_ACTIONS; ++i)
    {
        double q = 0.0f;
        for (int of = 0; of < NUMBER_OF_FEATURES; ++of)
            q += weights[i][of] * features[of];

        if (q > maxQ)
        {
            maxQ = q;
            maxIndex = i;
        }
    }

    return (Action)(maxIndex * 2 + maxIndex % 2);
}
```

```
void Sarsa::UpdateWeights(double reward, Action chosenAction,
                          bool isFinal, bool onlyPlay)
{
    if (historyIndex < TD_N && !isFinal)
        return;

    historyIndex = 0;
    if (onlyPlay)
        return;

    double q = 0.0f;

    int n;
    double diff = 0.0f;
    double q_ = 0.0;

    if (isFinal)
    {
        diff = alpha * (reward - q);

        for (int i = 0; i < NUMBER_OF_FEATURES; ++i)
        {
            weights[chosenAction][i] += diff * features[i];
        }
        n = historyIndex;
    }
    else
    {
        for (int of = 0; of < NUMBER_OF_FEATURES; ++of)
        {
            q_ += weights[chosenAction][of] * features[of];
        }
        n = TD_N - 1;
    }
    double gama = 1;
    for (; n >= 0; n--)
    {
        gama = gama * discount;
        int action = actions[n];

        q = 0.0;

        for (int of = 0; of < NUMBER_OF_FEATURES; ++of)
        {
            q += weights[action][of] * history[n][of];
        }
```

```
            diff = alpha * (reward + gama * q_ - q);

            reward += rewards[n];

            for (int i = 0; i < NUMBER_OF_FEATURES; ++i)
            {
                weights[action][i] += diff * history[n][i];
            }
        }
    }
```