Računarska gimnazija

MATURSKI RAD

Napredne tehnike programiranja

Primena učenja sa podsticajem za rešavanje Atari igara

Učenik:
Ognjen Nešković

Mentor:
dr Filip Marić

Beograd, maj 2022.

Sadržaj

1 Uvod **1**

1.1 Osnove 1

1.2 Motivacija 2

2 Metode **3**

2.1 Uvod 3

2.2 Evaluacija politike (policy evaluation) 5

2.3 Optimizacija vrednosti (value iteration) 9

2.4 Stohastička aproksimacija funkcije vrednosti stanja 13

2.5 Učenje Q vrednosti 15

2.6 Aproksimacija funkcije 17

2.6.1 Potpuno povezana neuronska mreža (multilayer perceptron) 17

2.6.2 Konvoluciona neuronska mreža 18

2.7 Deep Q-learning 21

3 Implementacija i rezultati ekperimenata **23**

3.1 Memorija (replay buffer) 23

3.2 DQN Algoritam 24

3.3 Treniranje 26

3.4 Eksperimenti 26

3.4.1 Cartpole swingup 26

3.4.2 Atari Breakout 29

4 Zaključak **32**

1

Uvod

1.1 Osnove

Još od davnina ljudi su se nadmetali u igranju igara poput šaha, kineske igre Go, Backgammon itd. U mnogim kulturama oni koji su bili najbolji u ovim igrama bili su izuzetno poštovani, pa bi neki ljudi posvećivali čak i čitav svoj život usavršavanju svoje strategije u nekoj od ovih igara. Stoga, dugo se smatralo da je sposobnost igranja ovih igara nešto jedinstveno za ljude i da je stvaranje mašine ili algoritma sposobnog da pobedi najboljeg šahistu ili drugog profesionalnog igrača gotovo nemoguće.
Ovaj izazov mučio je matematičare stotinama godina i ostao je neprevaziđen sve do 1997. godine kada je konačno računar (Deep blue) prvi put pobedio najboljeg šahistu tog vremena Garija Kasparova (Campbell et al., 2002). Deep blue je koristio alfa-beta algoritam pretrage, heuristike, ekspertsko znanje i specijalni hardver napravljen kako bi probao što veći broj poteza po sekundi. Postojao je veći broj šahovskih mašina koje su prethodile Deep blue mašini. Kako bi se razvio Deep blue bilo je potrebno puno šahista, programera, vremena i novca. Dakle, jasno je da iako su velik uspeh, Deep blue i njemu slične mašine nisu veoma generalne ili uopšte primenljive van vrlo specijalizovanog domena za koji su napravljene.
Velika prekretnica došla je sa razvojem neuronskih mreža, učenja sa podsticajem i hardvera. Sve ovo omogućilo je da se razviju generalni algoritmi koji bez velike modifikacije mogu naučiti da veoma dobro igraju veliki broj igara (Mnih et al., 2015). Sledeći veliki korak u mašinskom igranju igara došao je sa programom Alpha Go koji je 2016. godine pobedio svetskog šampiona Li Sedola u igri Go. Alpha Go je koristio učenje sa podsticajem, konvolucione neuronske mreže, Monte Karlo pretragu, takođe delimično je bio treniran na igrama koje su igrali profesionalni Go igrači (Silver et al., 2017). Godinu dana kasnije - 2017. godine objavljena je verzija Alpha Go-a koja ne koristi ikakvo ekspertsko znanje - Alpha Go Zero, koja

postiže čak bolji rezultat od verzije trenirane sa ekspertskim igrama. Poslednji veliki napredak je program AlphaStar koji igra stratešku video igru StarCraft II gde pobeđuje najbolje igrače ove igre (Vinyals et al., 2019).
Glavna razlika između metoda korišćenih za rešavanje šaha (alfa-beta pretraga) i savremenih metoda je način kako se igra modeluje. U savremenim metodama učenja sa podsticajem problemi koje je potrebno rešiti predstavljaju se kao Markovljevi procesi odlučivanja. Ako se problem ovako prestavi u nekim jednostavnim slučajevima se može rešiti dinamičkim programiranjem, a za složenije slučajeve se rešenje može aproksimirati neuronskim mrežama.

1.2 Motivacija

Igre ili opštije, problemi koji se mogu predstaviti kao Markovljevi procesi odlučivanja su od velikog teoretskog značaja za mašinsko učenje. Pored toga što donekle pružaju uvid u mogućnosti računara i razlike između ljudske i veštačke inteligencije, veliki broj procesa u stvarnosti poput problema optimalne kontrole (upravljanje robotima i mašinama), regulisanje sistema za hlađenje, kompresije videa itd. mogu se modelovati kao Markovljevi procesi odlučivanja. Složene igre pružaju dobar način da se granice savremenih metoda preciznije odrede, što primenu onda čini znatno lakšom.

2

Metode

2.1 Uvod

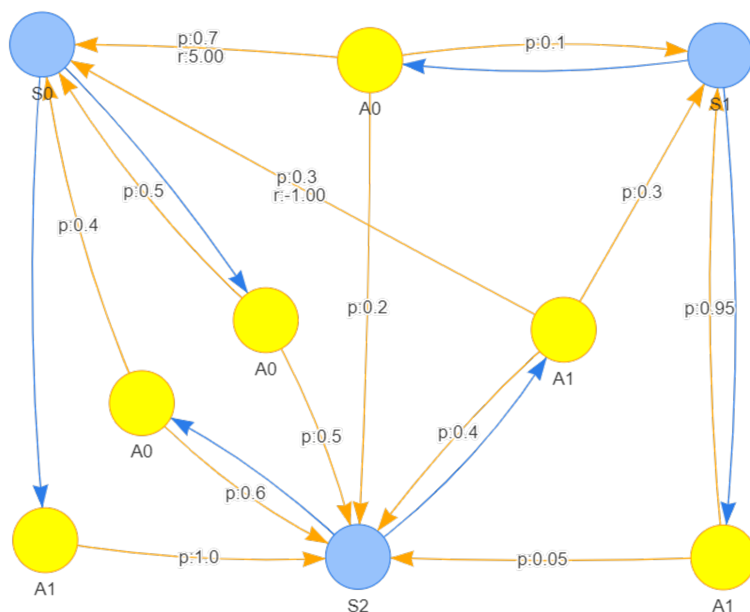
U narednom delu će biti razmatrane varijante Markovljevog procesa odlučivanja i rešenja koja su moguća za te varijante, pa je potrebno definisati Markovljev proces odlučivanja i uvesti potrebnu notaciju. Markovljevi procesi odlučivanja su uopštenje Markovljevih lanaca sa dodatim akcijama, za definiciju Markovljevog procesa potrebno je difinisati sledeće:

- Skup stanja S
- Skup akcija A i A_s - skup mogućih akcija u stanju s
- $P_a(s, s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$ - verovatnoću da se završi u stanju s' ako je u stanju s izvršena akcija a
- $R_a(s, s')$ - nagrada koja se ostvaruje kada se u stanju s izvrši akcija a i završi u stanju s'
- Funkcija (politika) $\pi(s) : S \to A$ koja predstavlja igrača (onog koji donosi odluku), pa za dato stanje s iz skupa stanja S određuje akciju koju treba izvršiti

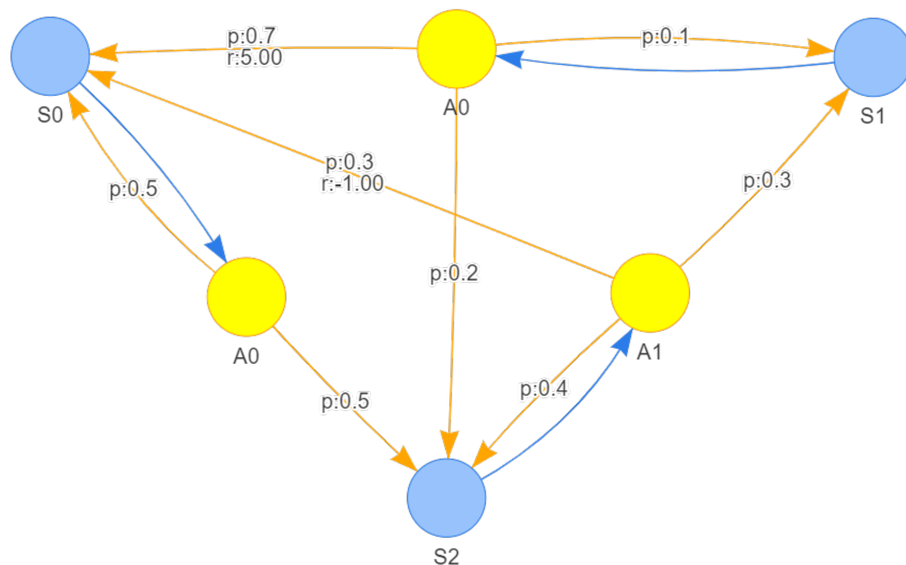
Rešenje Markovljevog procesa odlučivanja je optimalna funkcija π^*:

$$\text{argmax}_{\pi^*} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{\pi^*(s_t)}(s_t, s_{t+1})\right]$$

Posmatrajmo jednostavan Markovljev proces odlučivanja sa tri stanja S_0, S_1, S_2 i dve moguće akcije A_0, A_1 (slika 1).



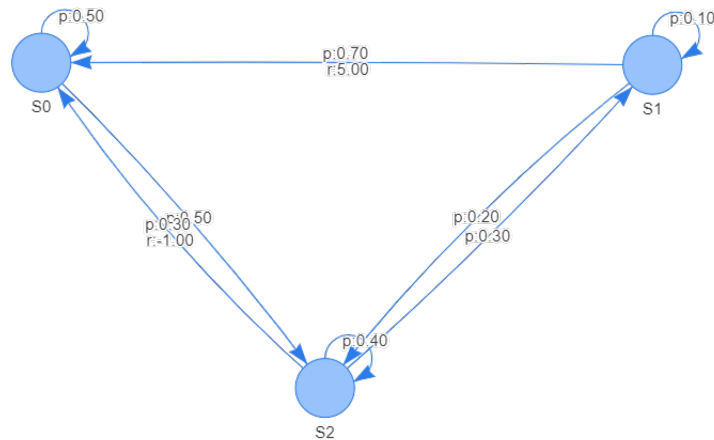
Slika 1: Primer Markovljevog procesa odlučivanja



Slika 2: Markovljev proces pod politikom π

Primetimo da ukoliko se fiksira neka politika π, na primer $\pi(S_0) = A_0, \pi(S_1) = A_0, \pi(S_2) = A_1$ proces sa slike 1 postaje ekvivalentan novom procesu (slika 2).

Dodatno, lako je videti da je tako dobijen proces (slika 2) ekvivalentan sledećem Markovljevom lancu, sa dodatim nagradama (slika 3). Kada je dat ovakav lanac,



Slika 3: Ekvivalentan Markovljev lanac sa nagradama

kao sa slike 3, postavlja se pitanje: kako efikasno odrediti očekivanu nagradu u tom lancu, ako je dato početno stanje ili verovatnoća za svako stanje da bude početno. Ako je ovo moguće uraditi, onda je odmah dat efikasan način da se evaluira očekivana nagrada koja će biti ostvarena ako se koristi neka fiksna politika.

2.2 Evaluacija politike (policy evaluation)

Kada je politika fiksirana, traženje očekivane nagrade koja će biti ostvarena ako se prati data politika je ekvivalentno nalaženju očekivane nagrade u nekom Markovljevom lancu. Prvo, uvedimo sledeću notaciju:

- S_t - Skup svih sekvenci dužine t. Na primer ako se iz stanja N_0 poseti stanje N_1, pa stanje N_2 itd. pa se na kraju poseti stanje N_t sekvenca bi bila $(N_0, N_1, N_2, \cdots, N_t)$
- $S_{t,u}$ - Skup svih sekvenci stanja dužine t koje se završavaju sa stanjem u.
- $T(u, v)$ - Verovatnoća da se iz stanja u pređe u stanje v.
- $p(s), s \in S_t$ - Verovatnoća da se neka sekvenca dogodi. Ako je data sekvenca $s = (N_0, N_1, N_2, \cdots, N_t)$ onda je $p(s) = T(N_0, N_1) \cdot T(N_1, N_2) \cdots T(N_{t-1}, N_t)$
- $R(u, v)$ - Nagrada kada se iz stanja u završi u stanju v.
- $r(s), s \in S_t$ - Nagrada za datu sekvencu s.

$$r(s) = R(N_0, N_1) + \gamma R(N_1, N_2) + \gamma^2 R(N_2, N_3) \cdots + \gamma^{t-1} R(N_{t-2}, N_{t-1})$$
$$\gamma < 1$$

- $L(u,t)$ - Verovatnoća da se posle t koraka završi u stanju u.

$$L(u,t) = \sum_{s \in S_{t,u}} p(s)$$

- $E_r(t)$ - Očekivana nagrada posle t koraka.

$$E_r(t) = \sum_{s \in S_t} p(s)r(s)$$

Lema 2.1. $L(u,t) = \sum_v T(v,u)L(v,t-1)$

$$L(u,t) = \sum_{s \in S_{t,u}} p(s)$$
$$L(u,t) = \sum_{N_0,N_1,\cdots,N_{t-1}} T(N_0,N_1) \cdot T(N_1,N_2) \cdots T(N_{t-2},N_{t-1}) \cdot T(N_{t-1},u)$$
$$L(u,t) = \sum_v \sum_{N_0,N_1,\cdots,N_{t-2}} T(N_0,N_1) \cdot T(N_1,N_2) \cdots T(N_{t-2},v) \cdot T(v,u)$$
$$L(u,t) = \sum_v T(v,u) \sum_{N_0,N_1,\cdots,N_{t-2}} T(N_0,N_1) \cdot T(N_1,N_2) \cdots T(N_{t-2},v)$$
$$L(u,t) = \sum_v T(v,u)L(v,t-1)$$

Ako definišemo da je $L(t)$ vektorska funkcija, a T matrica, možemo prethodnu jednačinu zapisati elegantnije kao:

$$L(t) = L(t-1)T$$

Kako je $L(0) = L_0$ gde je L_0 verovatnoća da se počne iz svakog stanja. Lako se može pokazati da je:

$$L(t) = L_0T^t$$

Lema 2.2. $E_r(t) = E_r(t-1) + \gamma^{t-1} \sum_u \sum_v T(u,v)R(u,v)L(u,t)$

$$E_r(t) = \sum_{s \in S_t} p(s)r(s)$$
$$E_r(t) = \sum_u \sum_{s' \in S_{t-1,u}} p(s') \sum_v T(u,v)(R(s') + \gamma^{t-1} R(u,v))$$
$$E_r(t) = \sum_u \sum_{s' \in S_{t-1,u}} p(s')(\sum_v T(u,v)R(s') + \gamma^{t-1} \sum_v T(u,v)R(u,v))$$
$$E_r(t) = \sum_u \sum_{s' \in S_{t-1,u}} p(s')(R(s') \sum_v T(u,v) + \gamma^{t-1} \sum_v T(u,v)R(u,v))$$
$$E_r(t) = \sum_u \sum_{s' \in S_{t-1,u}} p(s')(R(s') \cdot 1 + \gamma^{t-1} \sum_v T(u,v)R(u,v))$$
$$E_r(t) = \sum_u \sum_{s' \in S_{t-1,u}} p(s')(R(s') + \gamma^{t-1} \sum_v T(u,v)R(u,v))$$
$$E_r(t) = \sum_u \sum_{s' \in S_{t-1,u}} \left(p(s')R(s') + p(s')\gamma^{t-1} \sum_v T(u,v)R(u,v) \right)$$
$$E_r(t) = \sum_u \sum_{s' \in S_{t-1,u}} p(s')R(s') + \gamma^{t-1} \sum_u \sum_{s' \in S_{t-1,u}} p(s') \sum_v T(u,v)R(u,v)$$
$$E_r(t) = \sum_{s' \in S_{t-1}} p(s')R(s') + \gamma^{t-1} \sum_u \sum_{s' \in S_{t-1,u}} p(s') \sum_v T(u,v)R(u,v)$$
$$E_r(t) = E_r(t-1) + \gamma^{t-1} \sum_u \sum_{s' \in S_{t-1,u}} p(s') \sum_v T(u,v)R(u,v)$$
$$E_r(t) = E_r(t-1) + \gamma^{t-1} \sum_u \left(\sum_v T(u,v)R(u,v) \cdot \sum_{s' \in S_{t-1,u}} p(s') \right)$$
$$E_r(t) = E_r(t-1) + \gamma^{t-1} \sum_u \sum_v T(u,v)R(u,v) \cdot L(u,t)$$

Dodatno možemo primetiti sledeće:

$$\sum_u \sum_v T(u,v)R(u,v) \cdot L(u,t) = \sum_u L(u,t) \sum_v T(u,v)R(u,v)$$

Ukoliko definišemo $L(t)$ kao vektorsku funkciju, tako da je $L(t)_u$ verovatnoća da se u trenutku t bude u stanju u i primetimo da je $\sum_v T(u,v)R(u,v)$ konstantan vektor $\vec{c}$. Možemo zapisati izraz kao:

$$E_r(t) = E_r(t-1) + \gamma^{t-1} \vec{L(t)} \cdot \vec{c}$$

Lako je videti da je:

$$E_r(t) = \sum_{i=1}^{t} \gamma^{i-1} \vec{L(i)} \cdot \vec{c}$$

$$E_r(t) = \vec{c} \cdot \sum_{i=1}^{t} \gamma^{i-1} \vec{L(i)}$$

Za aciklične Markovljeve lance, odnosno za aperiodične matrice T izračunavanje $E_r(t)$ za dovoljno veliko t je dovoljno za nalaženje očekivane nagrade. Međutim za periodične matrice T je potrebno pokazati da $\lim_{t\to\infty} E_r(t)$ postoji.

Lema 2.3. $(\exists a) \lim_{t\to\infty} E_r(t) = a$
Ako se u izraz $E_r(t) = \vec{c} \cdot \sum_{i=1}^{t} \gamma^{i-1} L(\vec{i-1})$ zameni definicija za $\vec{L(i)}$ dobija se sledeći izraz:

$$E_r(t) = \vec{c} \cdot \sum_{i=1}^{t} \gamma^{i-1} \vec{L_0} T^{i-1}$$

$$E_r(t) = \vec{c} \cdot \left(\vec{L_0} \cdot \sum_{i=1}^{t} \gamma^{i-1} T^{i-1} \right)$$

Uvedimo matricu $A = \gamma T$, pa je prethodni izraz jednak:

$$E_r(t) = \vec{c} \cdot \left(\vec{L_0} \cdot \sum_{i=1}^{t} A^i \right)$$

Potrebno je naći $\lim_{t\to\infty} E_r(t) = \vec{c} \cdot \left(\vec{L_0} \cdot \sum_{t=0}^{\infty} A^t \right)$
$\sum_{t=0}^{\infty} A^t$ je Nojmanov red primenjen na prostor R^n.
Dovoljan uslov za konvergenciju Nojmanovog reda je da postoji norma tako da važi $||A|| < 1$.
Najlakše je odabrati $||A||_\infty$. Lako je videti da:

$$||A||_\infty = \max_i \sum_j |A_{i,j}|$$

$$||A||_\infty = \gamma < 1$$

Znamo da suma konvergira, dodatno lako je pokazati da za sumu $S = \sum_{i=0}^{t} A^i$ važi $S(I - A) = I - A^{t-1}$.
Ukoliko postoji inverz $(I - A)^{-1}$ (što postoji kada Nojmanov red konvergira) sledi da $S = (I - A^{t-1})(I - A)^{-1}$.

Kada $t \to \infty$ onda je $A^{t-1} = 0$. Pa je $S = I(I-A)^{-1} = (I-A)^{-1}$.
Čime je dokazano da $\lim_{t\to\infty} E_r(t) = \vec{c} \cdot \left(\vec{L_0} \cdot (I-A)^{-1}\right)$.

Korisno je napomenuti da postoji još jedan značajan način da se pronađe očekivana nagrada koju neka politika ostvaruje, a to je pomoću takozvanog *Belmanovog operatora očekivanja*. Može se definisati očekivana vrednost za politiku π iz stanja s kao $V^\pi(s)$. Funkcija V se naziva funkcija vrednosti stanja. Funkcije stanja mogu se posmatrati kao vektori u Banahovom prostoru $\mathbb{R}^{|S|}$ gde je S skup svih mogućih stanja. Na ovom prostoru se definiše i norma, na primer maksimum norma $||v||_\infty$. Može se uvesti Belmanov operator očekivanja za stacionarnu determinističku politiku π:

$$B^\pi(V)_s = \sum_{s'} T(S, \pi(S), S')(R(S, \pi(S), S') + \gamma V(S'))$$

ili slično i za nedeterminističke politike. Može se pokazati (Puterman, 2014) da je Belmanov operator očekivanja kontrakcija na datom prostoru. Stoga postoji stacionarna tačka $B^\pi(x) = x$. Dokazuje se (Puterman, 2014) da je tačka x upravo V^π. Sličan dokaz će detaljnije biti izložen za algoritam optimizacije vrednosti.

2.3 Optimizacija vrednosti (value iteration)

Uvedimo notaciju za očekivanu vrednost koja se može ostvariti iz stanja (funkciju vrednosti stanja) s ako se prati neka stacionarna, deterministička politika π:

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')R(s, \pi(s), s') + \gamma V^\pi(s')$$

Primetimo da sa obzirom da je funkcija V diskretna, a broj stanja konačan, možemo reći da se sve funkcije vrednosti stanja nalaze u Banahovom prostoru $R^{|S|}$ gde je S skup svih mogućih stanja. Dodatno u ovom prostoru uzećemo beskonačno normu $||v||_\infty := \max(|v_1|, |v_2|, ..., |v_n|)$. Algoritam optimizacije funkcije vrednosti stanja nalazi funkciju stanja $V^*(s)$ koja predstavlja maksimalnu moguću očekivanu vrednost koja se može ostvariti iz stanja s ako se i u narednim stanjima izvršavaju optimalne akcije. Kasnije će biti pokazano da V^* odgovara stacionarnoj, determinističkoj, optimalnoj politici π^*. U svrhu nalaženja V^* uvedimo takozvani *Belmanov optimizacioni operator* B^* koji deluje na vektore iz datog Banahovog prostora:

$$B^*(V)_s = \max_a \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V_{s'})$$

Algoritam optimizacije funkcije vrednosti stanja uključuje iterativnu primenu Belmanovog operatora na početnu funkciju vrednosti stanja (tj. na početni vektor). Kako bi pokazali da algoritam konvergira i da konvergira upravo na vrednost V^* biće dokazano da je operator B^* kontrakcija na datom Banahovom prostoru.
Prema Banahovoj teoremi fiksne tačke ukoliko je B^* kontrakcija na prostoru $R^{|S|}$ onda postoji vektor x^* takav da važi $B^*(x^*) = x^*$ i dodatno za sve ostale vektore x u tom prostoru važi da ukoliko se definiše red $x_n = B^*(x_{n-1}), x_0 = x$ onda je $\lim_{n\to\infty} x_n = x^*$. Na kraju potrebno je još pokazati da je $x^* = V^*$, odnosno da algoritam konvergira na optimalnu vrednost.

Teorema 2.1. B^* je kontrakcija, tj: $(\forall U \in R^{|S|}, \forall V \in R^{|S|}, U \neq V)||B^*(V) - B^*(U)||_\infty \leq k||V - U||_\infty$ gde $k \in [0, 1)$
Po definiciji je:

$$B^*(V)_s = \max_a \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V_{s'})$$

Pošto je skup akcija konačan, postoji akcija a_1 koja maksimizuje datu sumu, pa se može zapisati:

$$B^*(V)_s = \sum_{s'} T(s, a_1, s')(R(s, a_1, s') + \gamma V_{s'})$$

Analogno za U_s

$$B^*(U)_s = \sum_{s'} T(s, a_2, s')(R(s, a_2, s') + \gamma U_{s'})$$

Pošto akcija a_2 maksimizuje prethodnu sumu važi sledeća nejednakost:

$$B^*(U)_s \geq \sum_{s'} T(s, a_1, s')(R(s, a_1, s') + \gamma U_{s'})$$

Stoga za razliku $B^*(V)_s - B^*(U)_s$ važi nejednakost:

$$B^*(V)_s - B^*(U)_s \leq \sum_{s'} T(s, a_1, s')(R(s, a_1, s') + \gamma V_{s'}) - \sum_{s'} T(s, a_1, s')(R(s, a_1, s') + \gamma U_{s'})$$

$$B^*(V)_s - B^*(U)_s \leq \sum_{s'} (T(s, a_1, s')(R(s, a_1, s') + \gamma V_{s'} - R(s, a_1, s') - \gamma U_{s'}))$$

$$B^*(V)_s - B^*(U)_s \leq \gamma \sum_{s'} (T(s, a_1, s')(V_{s'} - U_{s'}))$$

Pošto je $|x| \geq x$, važi:

$$B^*(V)_s - B^*(U)_s \leq \gamma \sum_{s'} T(s, a_1, s')|V_{s'} - U_{s'}| \ (a)$$

Primetimo da ukoliko razmenimo vrednosti U i V dobija se sledeća nejednakost:

$$B^*(U)_s - B^*(V)_s \leq \gamma \sum_{s'} T(s, a_1, s')|V_{s'} - U_{s'}| \ (b)$$

Iz a, b sledi:

$$|B^*(V)_s - B^*(U)_s| \leq \gamma \sum_{s'} T(s, a_1, s')|V_{s'} - U_{s'}|$$

Ukoliko se $|V_{s'} - U_{s'}|$ zameni sa $\max_{s''}|V_{s''} - U_{s''}|$ desna strana nejednakosti se može samo povećati pa važi nejednakost:

$$|B^*(V)_s - B^*(U)_s| \leq \gamma \sum_{s'} T(s, a_1, s')\max_{s''}|V_{s''} - U_{s''}|$$
$$|B^*(V)_s - B^*(U)_s| \leq \gamma\max_{s''}|V_{s''} - U_{s''}|$$
$$|B^*(V)_s - B^*(U)_s| \leq \gamma||V - U||_\infty$$

Primetimo da nejednakost $|B^*(V)_s - B^*(U)_s| \leq \gamma||V - U||_\infty$ važi za svako U, V i svako stanje s, što je dovoljan uslov za:

$$||B^*(V) - B^*(U)||_\infty \leq \gamma||V - U||_\infty$$

A kako je $\gamma < 1$ teorema je dokazana.

Lema 2.4. $(\forall s)(\forall V \in \mathbb{R}^{|S|})B^*(V)_s \geq B^\pi(V)_s$
Za stacionarne, determinističke politike je dokaz jednostavan, pa je izložen dokaz za stacionarne nedeterminističke politike koje su nadskup stacionarnih, determinističkih politika. Po definiciji operatora B^* važi:

$$B^*(V)_s = \max_a \gamma \sum_{s'} T(s, a, s')(R(s, a, s') + V_{s'}) \geq \gamma \sum_{s'} T(s, a, s')(R(s, a, s') + V_{s'})(\forall a)$$

Ako se obe strane nejednakosti pomnože sa $\pi(a|s)$ i sumiraju po svakom a dobija se sledeća nejednakost:

$$\sum_a \pi(a|s)B^*(V)_s \geq \gamma \sum_a \pi(a|s) \sum_{s'} T(s, a, s')(R(s, a, s') + V_{s'})(\forall a)$$
$$\sum_a \pi(a|s)B^*(V)_s \geq \gamma \sum_a \sum_{s'} \pi(a|s)T(s, a, s')(R(s, a, s') + V_{s'})(\forall a)$$

Pošto je definicija $B^\pi(V)_s = \gamma \sum_a \sum_{s'} \pi(a|s)T(s, a, s')(R(s, a, s') + V_{s'})$ i kako je $\sum_a \pi(a|s)B^*(V)_s = B^*(V)_s$. Dokazano je da $B^*(V)_s \geq B^\pi(V)_s$.

Lema 2.5. Operator B^* optimizuje funkciju vrednosti stanja, odnosno nakon primene B^* funkcija vrednosti stanja se ne može pogoršati:
$(\forall V)(\forall s)B^*(V)_s \geq V_s$
Uzmimo politiku π tako da je V stacionarna tačka B^π. Treba dokazati:

$$B^*(V)_s \geq V_s$$

A pošto je $(\forall s)B^\pi(V)_s = V_s$ nejednakost je ekvivalentna:

$$B^*(V)_s \geq B^\pi(V)_s$$

Što važi na osnovu leme 2.4 čime je dokaz kompletiran.

Lema 2.6. Stacionarna tačka V^* operatora B^* je optimalna funkcija vrednosti, odnosno: $(\forall V)(\forall s)V_s^* \geq V_s$
Pretpostavimo suprotno: $(\exists v)(\exists s)V_s^* < V_s$
Na osnovu monotonosti operatora B^* može se primeniti operator B^* na desnu stranu nejednakosti proizvoljan broj puta:

$$V_s^* < B^*(V)_s$$
$$V_s^* < \lim_{k\to\infty}(B^*)^k(V)_s$$

Pošto je B^* kontrakcija, a V^* stacionarna tačka za B^* važi:

$$\lim_{k\to\infty}(B^*)^k(V)_s = V_s^*$$

Nejednakost je onda ekvivalentna:

$$V_s^* < V_s^*$$

Što je kontradikcija, čime je dokazana početna lema.

Ovim je pokazano da je moguće pronaći optimalnu funkciju vrednosti stanja počevši od bilo koje funkcije vrednosti stanja i iterativnom primenom Belmanovog optimizacionog operatora na tu funkciju. Još preostaje da se dokaže da postoji stacionarna, deterministička politika koja dostiže očekivanu vrednost optimalne funkcije stanja.

Teorema 2.2. U proizvoljnom Markovljevom procesu odlučivanja politika:

$$\pi^*(s) = \text{argmax}_a \left(\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{s'}^*) \right)$$

je stacionarna, deterministička politika čija je očekivana nagrada $(\forall s)V_s^{\pi^*} = V_s^*$, politika π^* je optimalna.
Kako je V^* stacionarna tačka B^* važi $(\forall s)B^*(V^*)_s = V_s^*$.

$$V_s^* = B^*(V^*)_s$$
$$V_s^* = \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{s'}^*)$$
$$V_s^* = \sum_{s'} T(s,\pi^*(s),s')(R(s,\pi^*(s),s') + \gamma V_{s'}^*)$$

Kako je $V_s^{\pi^*} = \sum_{s'} T(s,\pi^*(s),s')(R(s,\pi^*(s),s') + \gamma V_{s'}^{\pi^*})$, a iz definicje π^* je $V_{s'}^{\pi^*} = V_{s'}^*$ dokazano je da:

$$(\forall s)V_s^* = V_s^{\pi^*}$$

Zajedno sa metodom za pronalaženje optimalne funkcije stanja V^* ova teorema daje efikasan način da se pronađe optimalna politika odlučivanja π^*, što upotpunjuje algoritam optimizacije vrednosti.

2.4 Stohastička aproksimacija funkcije vrednosti stanja

U kontekstu primene Markovljevih procesa odlučivanja potrebno je razmatrati i procese u kojima verovatnoće nisu unapred poznate. Na primer, za primenu Belmanovog optimizacionog operatora B^* potrebno je znati vrednosti $T(s,a,s')$ i $R(s,a,s')$ za svako s, a i s'. U praksi ove vrednosti najčešće nisu poznate, a kasnije će se razmatrati i procesi gde je broj stanja previše velik da bi se te vrednosti uopšte čuvale ili sračunale. Pošto nije moguće znati prave vrednosti, neophodno je aproksimirati verovatnoću. Pošto je Belmanov operator očekivanja:

$$B^\pi(V)_s = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{s'})$$

Gde je $a = \pi(s)$.
Primetimo da se data suma može razdvojiti:

$$B^\pi(V)_s = \sum_{s'} T(s,a,s')R(s,a,s') + \sum_{s'} T(s,a,s')\gamma V_{s'}$$

Dodatno, primetimo da je $\sum_{s'} T(s,a,s')R(s,a,s')$ očekivana vrednost nagrade iz stanja s, kada se izvrši akcija a, odnosno: $\sum_{s'} T(s,a,s')R(s,a,s') = \mathbb{E}(R|s,a)$.

Slično, suma $\sum_{s'} T(s,a,s')V_{s'}$ je očekivana vrednost buduće nagrade, odnosno: $\sum_{s'} T(s,a,s')\gamma V_{s'} = \gamma\mathbb{E}(V|s,a)$. Belmanov operator se onda može ekvivalentno zapisati kao:

$$B^{\pi}(V)_s = \mathbb{E}(R|s,a) + \gamma\mathbb{E}(V|s,a)$$

U cilju aproksimacije potrebnih očekivanih vrednosti biće dokazane sledeće leme:

Lema 2.7. Za red $E_n = \frac{1}{n+1}(nE_{n-1} + x_n), E_0 = x_0$ važi $E_n = \frac{1}{n+1}\sum_{i=0}^{n} x_i$
Lema će biti dokazana indukcijom. Induktivna hipoteza je: $E_n = \frac{1}{n+1}\sum_{i=0}^{n} x_i$. Baza je $E_0 = \frac{1}{1}\sum_{i=0}^{0} x_i = x_0$, a po definiciji reda je $E_0 = x_0$ čime je baza indukcije dokazana.
Uzmimo da važi tvrdnja za $n-1$: $E_{n-1} = \frac{1}{n}\sum_{i=0}^{n-1} x_i$. Po definiciji je $E_n = \frac{1}{n+1}(nE_{n-1} + x_n)$. Ako se zameni vrednost za E_{n-1}, dobija se jednakost:

$$E_n = \frac{1}{n+1}(n\frac{1}{n}\sum_{i=0}^{n-1} x_i + x_n)$$

$$E_n = \frac{1}{n+1}(\sum_{i=0}^{n-1} x_i + x_n)$$

$$E_n = \frac{1}{n+1}\sum_{i=0}^{n} x_i$$

Čime je dokazana lema.

Lema 2.8. Neka je definisan red: $E_n = E_{n-1} + \frac{1}{n+1}(x_n - E_{n-1}), E_0 = x_0$, onda je $\lim_{n\to\infty} E_n = \mathbb{E}(X)$
Prvo je dokazana ekvivalencija sa jednostavnijim redom:

$$E_n = E_{n-1} + \frac{1}{n+1}(x_n - E_{n-1})$$

$$E_n = E_{n-1} + \frac{x_n}{n+1} - \frac{E_{n-1}}{n+1}$$

$$E_n = \frac{n}{n+1}E_{n-1} + \frac{x_n}{n+1}$$

$$E_n = \frac{1}{n+1}(nE_{n-1} + x_n)$$

Zajedno sa prethodnom lemom sledi:

$$E_n = \frac{1}{n+1}\sum_{i=0}^{n} x_i$$

Primetimo da je izraz $\frac{1}{n+1}\sum_{i=0}^{n} x_i$ srednja vrednost uzorka $x_0, x_1, ..., x_n$. Odnosno:

$$E_n = \frac{1}{n+1}\sum_{i=0}^{n} x_i = \overline{X_n}$$

Čime se početno tvrđenje svodi na:

$$\lim_{n\to\infty} \overline{X_n} = \mathbb{E}(X)$$

Što je tvrdnja zakona velikih brojeva.

Algoritam određivanja funkcije vrednosti stanja pod politikom π se može formulisati na sledeći način:

$$V_s \leftarrow \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{s'})$$

Gde je $a = \pi(s)$.
Ako je data epizoda generisana politikom π: $s_0, a_0, r_0, s_1, a_1, r_1, ..., s_n, a_n, r_n$. Na osnovu date epizode, motivisani prethodnim lemama možemo formulisati stohastičku verziju prethodnog algoritma:

$$V_{s_t} \leftarrow V_{s_t} + \alpha_t(R_t + \gamma V_{s_{t+1}} - V_{s_t})$$

Formalnije je dokazano (Sutton and Barto, 2018) da i ova verzija konvergira na pravu vrednost V^π.

2.5 Učenje Q vrednosti

Slično kao funkcija vrednosti stanja, može se definisati funkcija očekivane nagrade $Q^\pi(s,a)$ koja predstavlja očekivanu nagradu ako se u stanju s izvrši akcija a i nadalje deluje po politici π. Veoma slično kao za funkciju vrednosti stanja V^π, za Q funkciju se pokazuje da važi:

$$Q^\pi(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma Q(s', \pi(s)))$$

Postoji optimalna Q funkcija $Q^*(s,a)$ koja predstavlja maksimalnu očekivanu nagradu koja se može dobiti ako se u stanju s izvrši akcija a i nadalje deluje optimalno.

$$Q^*(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q^*(s',a'))$$

Ako je optimalna politika π^* važi:

$$V^* = V^{\pi^*}(s) = \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^{\pi^*}(s'))$$

Sa obzirom da su Q funkcija i V funkcija povezane sledećom jednakosti:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) Q^{\pi}(s,a)$$

Za politiku π^* važi:

$$V^{\pi^*}(s) = \max_a Q^{\pi^*}(s,a)$$

Stoga se jednačina za Q^* može zapisati ekvivalentno na sledeći način:

$$Q^*(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^{\pi^*}(s'))$$

Stoga je jasno da ako su date vrednosti optimalne funkcije V^* odmah su date i vrednosti optimalne Q funkcije Q^*. Što opravdava uvođenje algoritma optimizacije Q vrednosti koji je analogan prethodnom algoritmu učenja optimalne vrednosti stanja V^*.

$$Q(s,a) \leftarrow \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q(s',a'))$$

Zatim sledi i stohastička verzija tog algoritma:
Ako je data epizoda: $E : s_0, a_0, r_0, s_1, a_1, r_1, ..., s_n, a_n, r_n$
Onda je u iteraciji n, za svako s i svako a:

$$Q_n(s,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(s,a) + \alpha_n(r_n + \gamma \max_{a'} Q_{n-1}(s',a')), & \text{ako } s_n = s, a_n = a \\ Q_{n-1}(s,a), & \text{u suprotnom} \end{cases}$$

Kada $n \to \infty$ dokazano je da $Q_n(s,a) \to Q^*(s,a)$ (Watkins and Dayan, 1992) (Sutton and Barto, 2018) pod uslovima:

- Kada $n \to \infty$ svako stanje i svaka akcija će se pojaviti u sekvenci E beskonačno puta, odnosno: $(\forall s)(\forall a) p(s_n = s, a_n = a) > 0$.
- $(\forall s)(\forall a) \sum_i^{\infty} \alpha_{n_i(s,a)} = \infty$, ali $\sum_i^{\infty} \left(\alpha_{n_i(s,a)}\right)^2 = \text{const}$ gde je $n_i(s,a)$ i-ta pozicija u sekvenci E kada je akcija a izvršena u stanju s.
- Nagrade su konačne.

2.6 Aproksimacija funkcije

Ako je dat skup podataka $S : \{(\vec{x_0}, \vec{y_0}), (\vec{x_1}, \vec{y_1}), ..., (\vec{x_n}, \vec{y_n})\}, x_i \in \mathbb{R}^n, y_i \in \mathbb{R}^m$ cilj aproksimacije funkcije je pronaći funkciju $f(x) : \mathbb{R}^n \to \mathbb{R}^m$ koja minimizuje empirijski i / ili strukturalni rizik. Minimizacija empirijskog rizika definisana je kao minimizacija izraza:

$$\frac{1}{n} \sum_i L(f(x_i), y_i)$$

Gde je $L(\hat{y}, y) : (\mathbb{R}^m, \mathbb{R}^m) \to \mathbb{R}$ funkcija gubitka koja određuje koliko je neko predviđeno $\hat{y}$ slično pravom y. L može i ne mora biti metrika, na primer često se koriste funkcije gubitka:

- Apsolutna greška: $L_1(\hat{y}, y) = ||\hat{y} - y||_1$
- Kvadratna greška: $L_2(\hat{y}, y) = (||\hat{y} - y||_2)^2$
- Cross entropy: $L(\hat{y}, y) = -\sum_i y_i \ln(\hat{y}_i)$

Postoji veliki broj metoda za pronalaženje funkcije koja minimizuje funkciju gubitka poput linearne i logističke regresije, metode potpornih vektora, stabla odlučivanja, metode k najbližih komšija, neuronskih mreža itd.
Neuronske mreže ili specifičnije potpuno povezane neuronske mreže (multilayer perceptron) su univerzalni aproksimatori funkcija. Dokazano je (Hanin and Sellke, 2017) da neuronske mreže sa dovoljno velikim brojem slojeva mogu aproksimirati bilo koju kontinualnu funkciju na nekom zatvorenom intervalu. Odnosno za svako ϵ se može odabrati arhitektura neuronske mreže tako da se greška aproksimacije spusti ispod ϵ.
Zahvaljujući ovoj generalnosti su neuronske mreže postale glavni izbor za skoro sve probleme nadgledanog učenja.

2.6.1 Potpuno povezana neuronska mreža (multilayer perceptron)

Potpuno povezane neuronske mreže su funkcije oblika:

$$f(\vec{x}) = a(...a(a(\vec{x} \cdot W_0 + b_0) \cdot W_1 + b_1)...) \cdot W_n + b_n$$

Gde $x \in \mathbb{R}^{n_0}, W_0 \in \mathbb{R}^{n_0 \times n_1}, W_1 \in \mathbb{R}^{n_1 \times n_2}, ..., W_n \in \mathbb{R}^{n_{n-1} \times m}, b_0 \in \mathbb{R}^{n_1}, b_1 \in \mathbb{R}^{n_2}, ..., b_n \in \mathbb{R}^m$. A $a(\vec{x})$ aktivaciona funkcija. m je dimenzija izlaznog sloja, tj. dimenzija vektora y u skupu podataka $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$, a n_0 dimenzija vektora x.

Neuronska mreža minimizuje funkciju gubitka L odabirom optimalnih matrica težina $W_0, W_1, ...W_n$ i vektora (*biasa*) $b_0, b_1, ...b_n$. Neke česte aktivacione funkcije su ReLU, sigmoidna funkcija, tangens hiperbolični itd.
Algoritmi koji minimizuju funkciju greške i pronalaze optimalne vrednosti težina i biasa zasnivaju se na algoritmu gradijentnog spusta. Algoritam gradijentnog spusta zasniva se na nalaženju nule gradijenta funkcije gubitka:

$$\nabla L(Y, f(x, w_0, w_1, ..., w_n, b_0, b_1, ..., b_n)) = 0$$

Odnosno nuli parcijalnih izvoda:

$$\forall (i \in \{0, 1, ..., n\}) \frac{\partial L}{\partial W_i} = 0, \frac{\partial L}{\partial b_i} = \vec{0}$$

Eksplicitno nalaženje nula parcijalnih izvoda je često veoma velike složenost ili nemoguće. Zato se u praksi pribegava korišćenju numeričkih metoda za aproksimaciju nula. Osnovni algoritam gradijentnog spusta čini sledeća jednačina:

$$\forall (i \in \{1, 2, ..., n\})$$
$$W_{i,t} = W_{i,t-1} - \eta \frac{\partial L}{\partial W_i}(W_{i,t-1})$$
$$b_{i,t} = b_{i,t-1} - \eta \frac{\partial L}{\partial b_i}(b_{i,t-1})$$

Gde je η hiperparametar (*stopa učenja* / learning rate). Algoritam gradijentnog spusta je numerička metoda prvog reda koja nalazi lokalni minimum funkcije tako što se spušta niz gradijent (zato je znak $-$ ispred gradijenta). Algoritam gradijentnog spusta konvergira na lokalni minimum pod vrlo ograničavajućim uslovima - na primer dovoljni uslovi za pronalaženje lokalnog minimuma funkcije $g(x)$ su da je funkcija konveksna i njen gradijent Lipšic kontinualan. U cilju prevazilaženja ovih ograničenja smišljene su mnoge varijante algoritma gradijentnog spusta: AdaGrad, AdaDelta, RMSProp, Nestorov, Adam...
U praksi najuspešniji i najčešće korišćen je Adam (adaptive moment estimation) koji je takođe metoda prvog reda. Adam koristi takozvani *prvi i drugi moment* (vrednosti sračunate na osnovu prethodnih gradijenata) kako bi se stabilizovala aproksimacija gradijenta ili izbegli neki zaravnjeni delovi.

2.6.2 Konvoluciona neuronska mreža

Konvoluciona neuronska mreža je specifičan oblik neuronske mreže koji se najčešće koristi kada je potrebno analizirati podatke poput slika ili videa. Slike su najčešće podaci visoke dimenzionalnosti, na primer slika dimenzija 100×100 (sa još 3 kanala

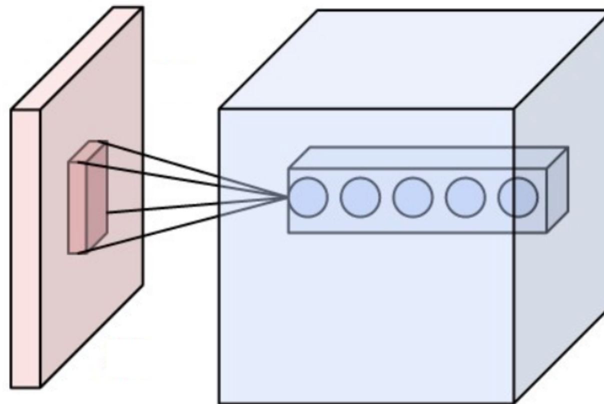
za r,g i b) je matrica 30000 brojeva. Ako bi ovu matricu ispravili u vektor veličine 30000, koristili potpuno povezanu mrežu i uzeli da sloj nakon ulaznog sloja ima samo 100 neurona matrica težina bi bila matrica dimenzije (30000×100) odnosno imala bi 3 miliona parametara koje je potrebno naučiti. Ovo značajno usporava i otežava treniranje pa je poželjno nekako smanjiti broj parametara koje je potrebno optimizovati. Na sreću slike su podaci koji poseduju veliku *prostornu lokalnost* - odnosno moguće je izvući puno informacija iz slike posmatrajući samo neki region slike. Na primer može se zaključiti da je mačka na slici ukoliko se na dva manja regiona detektuju oči i u sličnom regionu detektuje krzno, što je mnogo efikasnije nego posmatranje čitave slike istovremeno.
Glavni element konvolucione neuronske mreže jeste *konvolucioni sloj*.

Input | Kernel | Output

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}$$

Slika 4: Primer operacije konvolucije kernela (2×2) sa slikom (3×3) sa korakom 1

Konvolucioni sloj sačinjen je od nekoliko *filtera* / kernela (matrica realnih brojeva) koji su najčešće malih dimenzija $(2 \times 2, 3 \times 3, 5 \times 5, ...)$. Da bi se dobio izlaz konvolucionog sloja za svaki filter se računa operacija *konvolucije*, odnosno matematički ispravnije kros-korelacija filtera i ulazne slike.



Slika 5: Primer konvolucionog sloja

Najčešće ulazna slika ima dubinu veću od 1. Zbog toga se najčešće za jedan filter

računa konvolucija sa čitavom dubinom slike. Na primer ako je ulazna slika dimenzije $(100 \times 100 \times 3)$ ako se uzme filter (5×5) za konvoluciju bio bi dubine takođe 3 tj. dimenzija $(5 \times 5 \times 3)$. Takođe konvolucioni sloj ima nekoliko različitih filtera pa se za svaki od filtera računaju konvolucije, time se dobija dubina u izlazu tog sloja (na primer na slici iznad postoji 5 filtera pa je izlaz dubine 5). Konvolucioni sloj takođe određuje veličina koraka (tj. *stride*). Stride predstavlja za koliko se filter pomeri prilikom računanja proizvoda sa ulaznom slikom.

Input

0	2	2	7
3	2	5	8
6	7	8	4
6	2	2	1

*

Kernel

0	1
2	3

=

Output

14	21	41
35	43	36
25	18	11

Slika 6: Primer konvolucije slike $(4 \times 4 \times 1)$ sa filterom (2×2) sa korakom 1

Input

0	2	2	7
3	2	5	8
6	7	8	4
6	2	2	1

*

Kernel

0	1
2	3

=

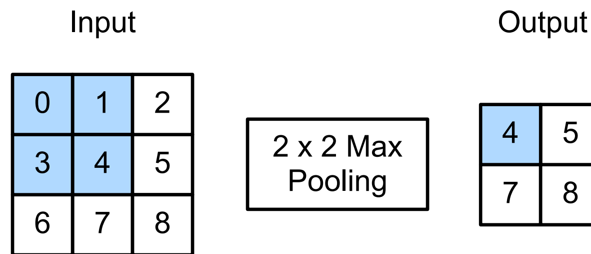
Output

14	41
25	11

Slika 7: Primer konvolucije slike $(4 \times 4 \times 1)$ sa filterom (2×2) sa korakom 2

Veličina filtera, broj filtera i korak definišu jedan konvolucioni sloj. Takođe, nakon računanja konvolucije vrednosti iz rezultujuće matrice se prosleđuju u aktivacionu funkciju, najčešće ReLU. Pored konvolucionih slojeva koriste se i takozvani *pooling* slojevi.

Pooling slojevi računaju neku funkciju (najčešće maksimum ili prosek) na nekom malom delu slike određenom pomoću veličine kernela, slično kao i konvolucija. Cilj pooling slojeva je da smanje dimenzionalnost ulazne slike. Izlazi poslednjeg pooling sloja se izravnjaju u vektor koji se zatim prosleđuje u jedan ili više potpuno povezanih slojeva, kako bi se na kraju dobio izlazni vektor odgovarajuće dimenzije.



Slika 8: Primer max pooling sloja sa dimenzijom (2×2)

Kako konvolucioni slojevi vrše diferencijabilne operacije jasno je da se i na ovakve neuronske mreže mogu primeniti algoritmi zasnovani na određivanju gradijenta.

2.7 Deep Q-learning

Kada je broj mogućih stanja u nekom Markovljevom procesu previše velik nalaženje Q vrednosti za svaki par stanja i akcije je praktično nemoguće. Na primer ako treba rešiti igru koja je predstavljena igraču grafički, poput većine video igara ili jednostavnije - arkadnih Atari igara, stanje je cela slika koja je predstavljena igraču. Na primer za Atari igre slika je dimenzija (160×192) i postoji 128 mogućih boja. Broj stanja je ograničen odozgo sa $128^{160 \cdot 192}$ što je očigledno previše velik broj stanja da bi se uopšte čuvao u memoriji.
Međutim, pretpostavka koja ipak omogućava da agent nauči optimalnu politiku je da iako su stanja visoko dimenziona količina informacija je prilično ograničena. Na primer verovatno je dovoljno znati da se na ekranu nalaze dva neprijatelja na određenoj udaljenosti kako bi se igralo optimalno. Nada je da ukoliko se koristi model sposoban da dovoljno apstrahuje stanja da će takav model moći da prepozna slična stanja i izgradi optimalnu politiku nad samo neophodnim informacijama. Međutim, aproksimacija Q funkcije pomoću nelinearnog modela je težak problem koji je tek skorije u nekoj meri rešen. Poznate su situacije gde nelinearni aproksimatori Q funkcije ne konvergiraju na optimalnu Q vrednost, šta više divergiraju u potpunosti (Lu et al., 2018). Međutim, ukoliko se Q funkcija aproksimira neuronskom mrežom i treniranje se vrši na specifičan način u praksi je često moguće dobiti veoma dobre rezultate.
Kako bi se aproksimirala Q funkcija posmatrajmo jednačinu za stohastičku verziju nalaženja optimalne Q vrednosti:
Ako je data epizoda: $E : s_0, a_0, r_0, s_1, a_1, r_1, ..., s_n, a_n, r_n$
Onda je u iteraciji n, za svako s i svako a:

$$Q_n(s,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(s,a) + \alpha_n(r_n + \gamma \text{max}_{a'} Q_{n-1}(s',a')), & \text{ako } s_n = s, a_n = a \\ Q_{n-1}(s,a), & \text{u suprotnom} \end{cases}$$

Ili u alternativnoj, ekvivalentnoj formulaciji:

$$Q_n(s,a) = \begin{cases} Q_{n-1}(s,a) + \alpha_n(r_n + \gamma \text{max}_{a'} Q_{n-1}(s',a') - Q_{n-1}(s,a)), & \text{ako } s_n = s, a_n = a \\ Q_{n-1}(s,a), & \text{u suprotnom} \end{cases}$$

Vrednost $r_n + \gamma \text{max}_{a'} Q_{n-1}(s',a') - Q_{n-1}(s,a)$ naziva se TD-greška (temporal difference error). Ako se ovo posmatra kao vrednost koju treba minimizovati može se upotrebiti neuronska mreža na sledeći način: Preformulišimo Q funkciju kao neuronsku mrežu, sa parametrima θ odnosno $Q(s,a,\theta)$. Onda je TD greška ekvivalentna:

$$L_n(\theta_n) = (r_n + \gamma \text{max}_{a'} Q(s'_n, a', \theta_n) - Q(s_n, a_n, \theta_n))^2$$

Učenje teče isto kao i za normalne neuronske mreže tako što se za svaki uzorak (s_n, a_n, r_n, s'_n) parametri mreže menjaju na sledeći način:

$$\theta_n = \theta_{n-1} - \eta \nabla L_n(\theta_{n-1})$$

Ovako formulisano učenje međutim nije preterano uspešno. Ako se učenje posmatra kao regresija i vrednost $r_n + \gamma \text{max}_{a'} Q(s'_n, a', \theta_n)$ kao vrednost koju treba predvideti vidi se da se promenljiva koju treba predvideti veoma često menja što destabilizuje učenje, takođe pokazano je da ovakav način učenja često precenjuje vrednost nagrade (Van Hasselt et al., 2016) pa se u praksi fiksiraju dva seta parametara θ i θ'. Funkcija gubitka se definiše kao:

$$L_n(\theta_n) = (r_n + \gamma \text{max}_{a'} Q(s'_n, a', \theta'_n) - Q(s_n, a_n, \theta_n))^2$$

Na neki fiksan broj koraka se parametri mreže θ' postavljaju na vrednost parametara θ. Takođe moguće je i formulisati funkciju gubitka na sledeći način kako bi se izbeglo precenjivanje nagrade:

$$L_n(\theta_n) = (r_n + \gamma Q(s', \text{argmax}_{a'} Q(s'_n, a', \theta_n), \theta'_n) - Q(s_n, a_n, \theta_n))^2$$

Ovo se naziva *Double DQN* (Van Hasselt et al., 2016). Pored ovih tehnika, primećeno je da kada se neuronske mreže koriste za aproksimaciju Q funkcije, često dolazi do divergencije ili lošeg učenja zbog velike korelacije među Q vrednostima ako četvorke (s,a,r,s') predstavljaju uzastopne četvorke neke epizode. Kako bi se izbegla korelacija četvorke iskustva (s,a,r,s') koje mreža generiše prilikom interakcije sa okruženjem dodaju se u velik niz iskustava odakle se prilikom treniranja biraju nasumične četvorke (s,a,r,s') na kojima se mreža trenira. Ovo se naziva *experience replay* (Mnih et al., 2015).

3

Implementacija i rezultati ekperimenata

U okviru rada implementiran je algoritam Deep Q-learning. Algoritam je zatim primenjen za rešavanje problema optimalne kontrole *Cartpole swing-up*, a zatim i Atari igru *Breakout*. Neuronske mreže implementirane su pomoću biblioteke *PyTorch* (Paszke et al., 2019), a okruženja sa kojima agent interaguje su deo standardne implementacije *OpenAI gym* (Brockman et al., 2016).

3.1 Memorija (replay buffer)

Replay bafer je implementiran u klasi `ReplayBuffer`. Cilj replay bafera je čuvanje četvorki iskustva (s, a, r, s'). U kodu vrednosti s, a, r i s' se čuvaju u odvojenim torch tensorima, kako bi se lakše ceo bafer prebacio na memoriju grafičke karte. Suštinu ove klase čine sledeće metode:

```
def add(self, state, action, reward, state_, done):
        mem_index = self.mem_count % self.mem_size

        self.states[mem_index]  = torch.tensor(state, dtype=torch.float32)
        self.actions[mem_index] = action
        self.rewards[mem_index] = reward
        self.states_[mem_index] = torch.tensor(state_, dtype=torch.float32)
        self.dones[mem_index] =  done

        self.mem_count += 1

def sample_batch(self, batch_size):
        mem_max = min(self.mem_count, self.mem_size)
```

```
        batch_indices = torch.randint(0,mem_max,(batch_size,))

        states  = self.states[batch_indices].float()
        actions = self.actions[batch_indices]
        rewards = self.rewards[batch_indices]
        states_ = self.states_[batch_indices].float()
        dones   = self.dones[batch_indices]

        return (states, actions, rewards, states_, dones)
```

Kao što se može videti iz koda takođe se čuvaju i vrednosti bool tipa koje označavaju da li je epizoda završena sa tim uzorkom (s, a, r, s').

3.2 DQN Algoritam

U klasi DQNAgent implementiran je glavni deo Deep Q-learning algoritma. U konstruktor klase prosleđuje se niz PyTorch slojeva, na primer:

```
architecture = [
                torch.nn.Linear(input_shape[0], 64),
                torch.nn.ReLU(),
                torch.nn.Linear(64, 64),
                torch.nn.ReLU(),
                torch.nn.Linear(64, n_actions)
            ]
```

Gde su `input_shape` i `n_actions` parametri koji predstavljaju dimenzije stanja koje proizvodi okruženje i broj mogućih akcija u okruženju. U konstruktoru klase se zatim inicijalizuju sledeće promenljive (neuronska mreža, algoritam za optimizaciju, ciljna mreža i loss funkcija):

```
        self.model = torch.nn.Sequential(*architecture).to(TORCH_DEVICE)
        self.optimizer = torch.optim.Adam(self.model.parameters(),
                lr = self.learning_rate)
        self.target_model = torch.nn.Sequential(*architecture).to(TORCH_DEVICE)
        self.target_model.load_state_dict(self.model.state_dict())
        self.loss_fn = torch.nn.SmoothL1Loss()
```

Definisane su i dve najbitnije metode, `select_action`:

```
def select_action(self, state, greedy=False):
        state = torch.from_numpy(state).float().unsqueeze(0).to(TORCH_DEVICE)
```

```
        # Akcija se bira pohlepno (koristi se pri evaluaciji)
        if greedy:
            with torch.no_grad():
                return self.model(state).argmax()
        # Akcija se bira po epsilon-greedy strategiji
                # (sa verovatnoćom epsilon se bira nasumična akcija)
        else:
            if torch.rand(1) > self.eps:
                with torch.no_grad():
                    return self.model(state).argmax()
            else:
                return torch.randint(0, self.n_actions, (1, 1))
```

I metoda `update_batch`:

```
def update_batch(self, batch_samples):
        batch_states, selected_actions, batch_rewards, batch_next_states, batch
        selected_actions = selected_actions.reshape(-1, 1)

        # Predviđaju se Q vrednosti pomoću trenutnog modela
        pred_q = self.model(batch_states)
        # Izdvajaju se samo vrednosti za akcije koje su zapravo izvršene
        pred_q = pred_q.gather(1, selected_actions).flatten()

        # Predviđaju se Q vrednosti pomoću ciljnog modela
        true_q = self.gamma * self.target_model(batch_next_states).max(1)[0] *

        # Računa se loss funkcija i poziva se backward
        loss = self.loss_fn(pred_q, true_q)
        self.optimizer.zero_grad()
        loss.backward()

        # Vrednosti gradijenta se ograničavaju
        for param in self.model.parameters():
                param.grad.data.clamp_(-1, 1)
        self.optimizer.step()

        # Smanjuje se trenutno epsilon
        if self.eps > self.eps_min:
                self.eps *= self.eps_decay
```

```
        # Ako je potrebno poziva se metoda za kopiranje težina u ciljnu mrežu
        self.target_update_remaing -= 1
        if self.target_update_remaing <= 0:
                self.update_target_model()
                self.target_update_remaing = self.target_update_interval

        return loss.item()
```

3.3 Treniranje

Za potpunu implementaciju potrebna je još glavna petlja za treniranje agenta. Suštinski deo tog koda je izložen u sledećem isečku:

```
env = gym.make("CartPole-v1") # Na primer inicijalizuje se okruženje cartpole
dqn_agent = DQNAgent(env.observation_space.shape, env.action_space.n)
buffer = ReplayBuffer(env.observation_space.shape, learning_starts, max_buffer_len)

train_step_cnt = 0
while train_step_cnt < total_train_steps:
    current_state = env.reset()

    done = False
    while not done:
        action = int(dqn_agent.select_action(current_state))
        next_state, reward, done, _ = env.step(action)
        buffer.add(current_state, action, reward, next_state, done)
        current_state = next_state

        # Agent se trenira na nasumičnom uzorku iz bafera
        if buffer.can_sample():
            sample = buffer.sample_batch(batch_size)
            loss = dqn_agent.update_batch(sample)
            train_step_cnt += 1
```

3.4 Eksperimenti

3.4.1 Cartpole swingup

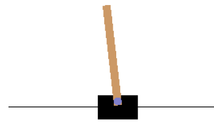
U problemu cartpole swingup agent upravlja kolicima za koje je zglobno fiksirana šipka. Kolica se mogu pomerati levo i desno, a šipka počinje malo pomerena od

ravnotežnog položaja. Cilj agenta je da održi šipku uspravnom (između $\pm 12°$).

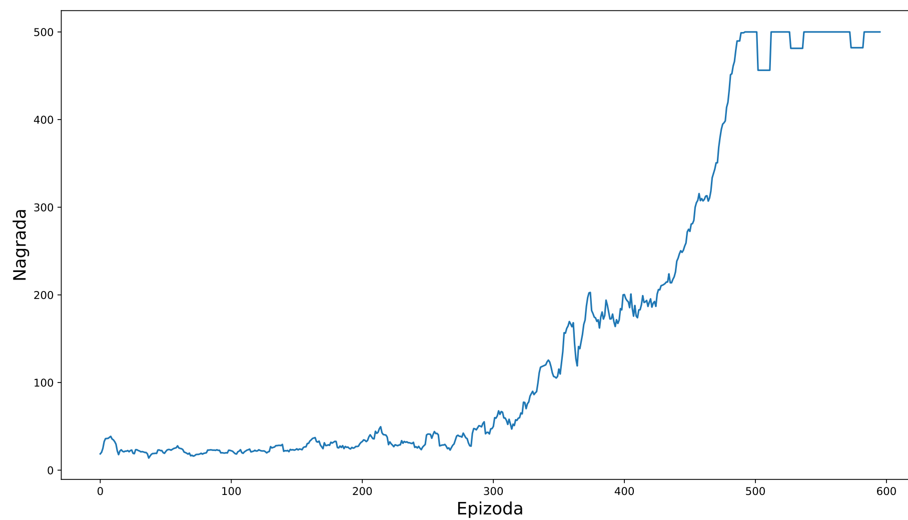
	Vrednost	**Minimum**	**Maksimum**
1	Pozicija kolica	-4.8	4.8
2	Brzina kolica	$-\infty$	∞
3	Ugao šipke	≈ -0.418 rad	≈ 0.418 rad
4	Ugaona brzina šipke	$-\infty$	∞

Tabela 3.1: Stanje koje agent dobija od okruženja Cartpole swingup

Agent dobija $+1$ nagradu za svaki korak koji izvrši dok se epizoda ne završi, epizoda se završava ako ugao šipke nije između $\pm 12°$ ili epizoda traje više od 500 koraka.Korišćena je arhitektura mreže sa dva skrivena sloja od po 64 neurona. Postignuta je maksimalna moguća nagrada 500.



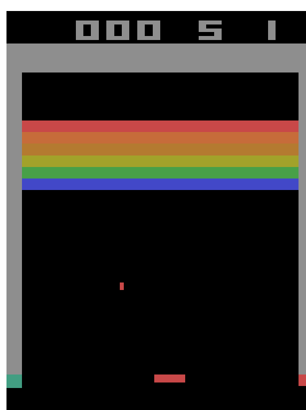
Slika 9: Primer stanja s $[0.06, 0.17, -0.12, -0.49]$ u okruženju Cartpole swingup



Slika 10: Prosek nagrada u zadnjih 10 epizoda tokom treniranja na problemu Cartpole swingup

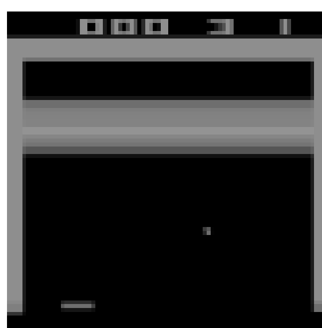
3.4.2 Atari Breakout

U igri Breakout postoji loptica koja kreće iz sredine ekrana i pada na dole. Agent upravlja palicom na dnu ekrana kojom odbija lopticu ka gore. Na vrhu ekrana nalaze se cigle koje se unište kada loptica udari u njih. Cigle su različite boje i vrede različiti broj poena. Loptica se odbija drugačije u zavisnosti u koji deo palice udari. Cilj je uništiti sve cigle. Agent ima 5 života.
Agent preciznije ima 4 moguće akcije: ništa, ispali lopticu (na početku igre), pomeri palicu levo i desno. Stanje koje agent dobija od okruženja je slika dimenzija $210 \times 160 \times 3$. Slika je jednostavno predprocesirana tako što je pretvorena u crno-



Slika 11: Primer stanja u okruženju Atari Breakout

belu (monohromatsku) sliku i rezolucija smanjena na 84×84. Rezultujuća slika je onda dimenzija $84 \times 84 \times 1$.



Slika 12: Primer stanja u okruženju Atari Breakout nakon predprocesiranja

Mreža korišćena za rešavanje ove igre je konvoluciona mreža sa 3 konvoluciona sloja:

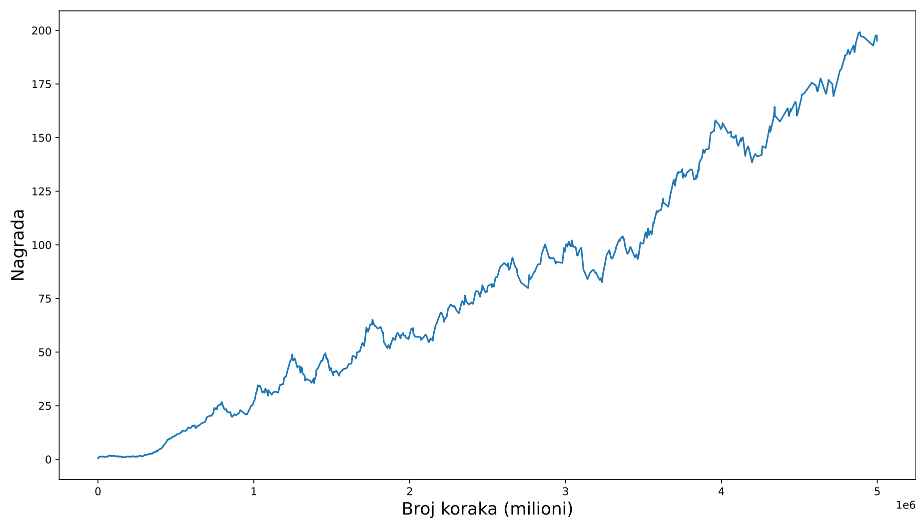
1. 32 filtera dimenzija 8×8, sa korakom 4, ReLU

2. 64 filtera dimenzija 4×4, sa korakom 2, ReLU
3. 64 filtera dimenzija 3×3, sa korakom 1, ReLU

Nakon konvolucionih slojeva dodat je jedan sloj od 512 neurona praćen ReLU aktivacionom funkcijom. Izlazni sloj ima 4 neurona, koliko i akcija. U kodu je ova arhitektura definisana na sledeći način:

```
dqn_architecture = [
    torch.nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8, stride=4),
    torch.nn.ReLU(),
    torch.nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
    torch.nn.ReLU(),
    torch.nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
    torch.nn.ReLU(),
    torch.nn.Flatten(),
    torch.nn.Linear(3136, 512),
    torch.nn.ReLU(),
    torch.nn.Linear(512, env.action_space.n)
]
```

Model uspešno igra igru Atari Breakout i dostiže nagradu od ≈ 200.



Slika 13: Nagrada tokom treniranja na igri Atari Breakout

Hiperparametar	**Cartpole**	**Breakout**	**Opis**
total_train_steps	100 000	5 000 000	Ukupan broj koraka za treniranje
max_buffer_len	10 000	100 000	Najveći broj četvorki (s, a, r, s') koji se mogu naći u baferu
learning_starts	500	5 000	Nakon koliko koraka potpuno nasumičnog igranja počinje učenje
target_update_freq	3000	10 000	Broj koraka nakon kog se kopiraju parametri trenutne mreže u ciljnu mrežu
batch_size	32	32	Broj uzoraka (četvorki (s, a, r, s')) na kojima se trenira mreža
gamma	0.99	0.99	*Discount factor* - vrednost kojom se množi buduća nagrada
eps	1.0	1.0	Početna vrednost ϵ korišćena u ϵ-greedy politici
eps_min	0.1	0.1	Minimalna vrednost ϵ
eps_decay	0.9998	0.999997	Vrednost kojom se množi trenutno ϵ nakon svakog koraka treniranja
learning_rate	0.00025	0.00025	Hiperparametar za Adam

Tabela 3.2: Vrednosti hiperparametara korišćenih za eksperimente Cartpole i Breakout

4

Zaključak

U radu su izložene metode učenja sa podsticajem od jednostavnijih ka složenijim. Dokazane su najznačajnije teoreme vezane za algoritme optimizacije funkcije vrednosti stanja i evaluacije politike. Pokazano je u kakvim Markovljevim procesima se mogu koristiti klasični algoritmi poput optimizacije vrednosti, a kada je potrebno posegnuti za ozbiljnijim metodama.
Za rešavanje većih i nepoznatih Markovljevih procesa uvedene su stohastičke metode za koje su izloženi neki osnovni dokazi i intuicija. U svrhu aproksimacije Q funkcije su sažeto uvedene potpuno povezane i konvolucione neuronske mreže. Pokazana je njihova primena za aproksimaciju Q funkcije, a zatim je to praktično primenjeno za rešavanje problema optimalne kontrole Cartpole swingup, pa i konačno za rešavanje igre Atari Breakout.
Ovim su ukratko izložene savremenije metode učenja sa podsticajem. Pokazano je da se one mogu jednostavno implementirati i trenirati da rešavaju izuzetno složene i visokodimenzione probleme pomoću relativno pristupačnog hardvera. Međutim, treba napomenuti da se polje mašinskog učenja razvija izuzetno brzo i da su algoritmi izloženi u ovom radu, iako impresivni, već značajno nadograđeni i u praksi se više ne koriste samo izvorne verzije algoritma Deep Q-learning kao što je on ovde dat. Svakako, Deep Q-learning kao što je predstavljen u ovom radu čini osnovu većine trenutno najboljih algoritama za učenje sa podsticajem.

Literatura

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3), 279–292.

Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, *134*(1-2), 57–83.

Puterman, M. L. (2014). *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence*, *30*(1).

Hanin, B., & Sellke, M. (2017). Approximating continuous functions by relu nets of minimal width. *arXiv preprint arXiv:1710.11278*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, *550*(7676), 354–359.

Lu, T., Schuurmans, D., & Boutilier, C. (2018). Non-delusional q-learning and value-iteration. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/5fd0245f6c9ddbdf3eff0f505975b6a7-Paper.pdf

Sutton, R., & Barto, A. (2018). *Reinforcement learning, second edition: An introduction*. MIT Press. https://books.google.rs/books?id=uWV0DwAAQBAJ

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F.

d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, *575*(7782), 350–354.