AlphaZero: strojno učenje podrškom bez domenskog znanja

Lončar, Jelena

AlphaZero: strojno učenje podrškom bez domenskog znanja

Lončar, Jelena

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Jelena Lončar

ALPHAZERO: STROJNO UČENJE PODRŠKOM BEZ DOMENSKOG ZNANJA

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, studeni 2020.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

„Šah je više od stoljeća imao ulogu Kamena iz Rosette za ljudsku i strojnu spoznaju. AlphaZero obnavlja posebnu vezu drevne igre na ploči i vrhunske znanosti postizanjem nečeg izvanrednog.", izjava je[1] nekadašnjeg svjetskog prvaka u šahu Garryja Kasparova. Naime, krajem 2017. godine tvrtka specijalizirana za umjetnu inteligenciju DeepMind objavila je *preprint* [21] u kojemu je predstavila algoritam pod nazivom AlphaZero, razvijen kako bi savladao šah, shogi (japanski šah) i igru Go. Krajem 2018. znanstveni rad tvrtke DeepMind o algoritmu AlphaZero [22] objavljen je u prestižnom časopisu Science. Pošto je objavljen, AlphaZero ispunio je medijske naslove i izazvao brojne reakcije — naime, Garry Kasparov nije bio jedini koji se osvrnuo na izuzetnost algoritma AlphaZero; danski šahovski velemajstor Peter Heine Nielsen usporedio je[2] igru algoritma AlphaZero s igrom superiorne vanzemaljske vrste. Također, norveški velemajstor Jon Ludvig Hammer okarakterizirao je[3] igru AlphaZero algoritma kao „ludi napadački šah" s dubokim pozicijskim razumijevanjem. No, što je AlphaZero zapravo i zašto se smatra tako izvanrednim? Tom Simonite, poznat po radu za MIT Technology Review, New Scientist i Wired, u članku [25] opisao je AlphaZero kao „prvog višestruko vještog umjetno inteligentnog šampiona igara na ploči". Naime, AlphaZero je algoritam koji *tabula rasa* može postići nadljudski učinak u raznovrsnim izazovnim domenama. Počevši od nasumičnog igranja i bez ikakvog znanja o domeni osim pravila igre, AlphaZero u samo je 24 sata postigao nadljudsku razinu igranja šaha, shogija i igre Go te je uvjerljivo pobijedio kompjuterske svjetske prvake u svakoj od navedenih disciplina. Računala su i prije algoritma AlphaZero uspijevala pobijediti ljude u kompleksnim igrama, uključujući šah i Go. Međutim, ti su programi tipično bili konstruirani za točno određene igre, pritom iskorištavajući njihova svojstva, poput simetrija ploča na kojima se igraju. S druge strane, AlphaZero ima sposobnost prilagođavanja raznolikim pravilima igre, što je značajan korak naprijed prema ostvarivanju općeg sustava za igranje igara. Kad bi AlphaZero algoritam bio značajno kompleksniji od algoritama računalnih programa uspješnih u kompleksnim igrama koji su mu prethodili, i dalje bi predstavljao vrhunsko postignuće. Međutim, ono što ga čini zbilja izvanrednim njegova je relativna

[1] Izjava u izvornom obliku (na engleskom jeziku) može se pronaći u publikaciji [23].
[2] Izjava u izvornom obliku (na engleskom jeziku) može se pronaći u publikaciji [1].
[3] Izjava u izvornom obliku (na engleskom jeziku) može se pronaći u publikaciji [12].

jednostavnost u odnosu na njegove prethodnike. Detaljni odgovori na pitanja što AlphaZero jest i koje su njegove mogućnosti bit će dani u ovom diplomskom radu — najprije će iscrpno biti opisane pozadinska teorija i struktura AlphaZero metode, a potom će njezine mogućnosti biti demonstrirane i implementacijom algoritma tipa AlphaZero za igru Četiri u nizu (engl. *Connect Four*, *Four in a Row*).

Poglavlje 1

Problemi koje rješava AlphaZero — pozadinska teorija

Prije nego što izložimo strukturu AlphaZero metode, odnosno na koji način AlphaZero rješava odgovarajuće probleme, podrobno opišimo o kojim se problemima ustvari radi i upoznajmo se s temeljnim konceptima na kojima počiva njihovo rješavanje, a samim time i AlphaZero algoritam.

Kao što je navedeno u uvodu, AlphaZero sposoban je savladati izazovne igre na ploči poput šaha i igre Go, no, koje su točno zajedničke karakteristike problema koje AlphaZero rješava? Kako modelirati igre koje je AlphaZero uspješno naučio? Također, kako modelirati problem igranja igara; što nam o njemu može reći umjetna inteligencija?

Intuitivno, možemo odmah reći da AlphaZero nudi elegantan pristup rješavanju problema igranja igara na ploči, gdje su pravila igre fiksirana, aktualni je položaj u potpunosti poznat te postoji protivnik, kojemu je primarni cilj spriječiti igrača da pobijedi u igri. Postojanje protivnika upravo je ono što takve igre čini kompliciranima — pravila igre obično su dosta jednostavna i kompaktna, ali je otežavajući faktor prisutnost protivnika s nepoznatom strategijom, koji nastoji pobijediti u igri.

Kako bismo dali formalan odgovor na postavljena pitanja, u nastavku navodimo relevantne pojmove pozadinske teorije: teorije igara i umjetne inteligencije, osobito njene metode strojnog učenja podrškom.

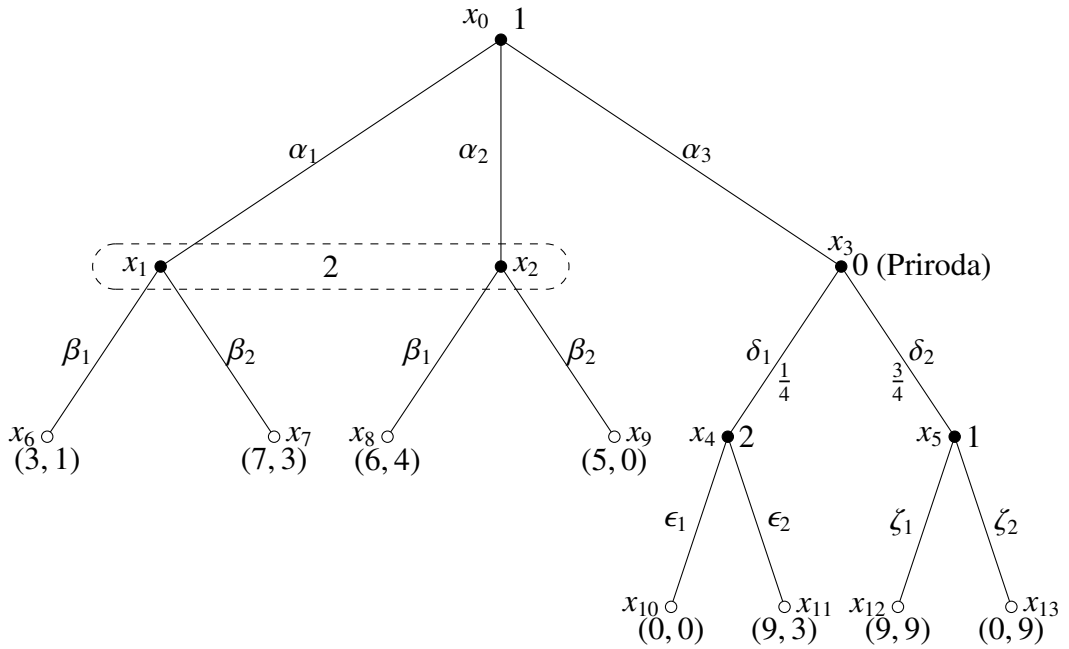
1.1 Teorija igara

U ovom ćemo odjeljku primarno predstaviti takozvane *ekstenzivne igre*, odnosno model sekvencijalnih interakcija između više agenata, čija je reprezentacija bazirana na stablima.

Promotrimo, za početak, igru prikazanu pomoću stabla na slici 1.1, napravljenoj po uzoru na sliku iz [19]. Vrhovi (čvorovi) u stablu na navedenoj slici označavaju različita

stanja tijekom igre. Stanje u potpunosti opisuje neki trenutak u igri: primjerice, položaj svih figura na ploči u šahu, zajedno s informacijom o tome koji je igrač na redu. Djeca nekog vrha predstavljaju sva ona stanja do kojih se može doći jednim potezom u igri. Uočimo da su listovi stabla vrhovi x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12} i x_{13} — njih nazivamo *završnim vrhovima* budući da označavaju moguće završetke igre. Uz svaki od završnih vrhova označeno je ono što bismo intuitivno mogli nazvati nagradama pojedinim igračima za dolazak u odgovarajuće vrhove — ustvari se radi o vrijednostima funkcija korisnosti, koje će formalno biti definirane u definiciji 1.1.2 u nastavku odjeljka. Preostale vrhove dijelimo na *vrhove odluke* i *slučajne vrhove*. Vrhovi su odluke vrhovi x_0, x_1, x_2, x_4 i x_5 — u svakome od njih jedan od igrača odabire jedan potez između nekoliko njih, koji vode do drugih vrhova stabla, odnosno drugih stanja igre. Na slici 1.1 uz svaki je vrh odluke brojem 1 ili 2 naznačeno koji igrač odlučuje o potezu. Slučajan je vrh x_3. U slučajnim vrhovima grananje nije određeno odlukom nekog od igrača, već se odvija slučajno. Kažemo da u takvim vrhovima odluku donosi Priroda, koju često smatramo jednim dodatnim igračem kojeg označavamo pomoću 0.

Slika 1.1: Ekstenzivna igra.



Promotrimo tijek igre prikazane stablom na slici 1.1. Igru započinje prvi igrač (igrač označen pomoću 1) te na raspolaganju ima tri poteza: potez označen pomoću α_1, onaj označen pomoć α_2 i onaj koji predstavlja oznaka α_3. Potezi α_1 i α_2 vode redom u vrhove x_1, odnosno x_2. Možemo primijetiti da su ta dva vrha na dijagramu objedinjena u jedan

skup, za koji je označeno da pripada igraču 2. Grupiranje tih dvaju vrhova označava da u njima drugi igrač odlučuje o potezu, no da nema informaciju o tome koji točno od njih dvojice predstavlja trenutno stanje igre. Odnosno, drugom je igraču poznato da se prvi igrač odlučio za potez α_1 ili potez α_2, no ne zna o kojemu se točno radilo. U skladu s navedenim, u vrhovima x_1 i x_2 drugi igrač može birati između dva poteza: β_1 i β_2. Primjerice, u slučaju da se drugi igrač odluči za potez β_2, a prethodno je prvi igrač odigrao potez α_1, igra završava u završnom vrhu x_7, kojemu odgovaraju korisnost iznosa 7 za prvog igrača i korisnost iznosa 3 za drugog igrača. Da se prvi igrač inicijalno odlučio za potez α_3, o sljedećem bi grananju odlučivala Priroda. U vrhu x_3 Priroda na raspologanju ima dva poteza: δ_1 i δ_2. Dakle, nakon što prvi igrač odigra potez α_3, igra se s vjerojatnošću $\frac{1}{4}$ nastavlja u vrhu x_4 te s vjerojatnošću $\frac{3}{4}$ u vrhu x_5.

Definirajmo sada formalno pojmove intuitivno opisane u prethodna dva odlomka.

Definicija 1.1.1. Usmjereno stablo s korijenom *uređen je par* $T = (X, E)$ *takav da vrijedi sljedeće:*

- X *je konačan skup s barem dva elementa. Njegove elemente nazivamo* vrhovima*;*
- E *je podskup skupa* $X \times X$*. Njegove elemente nazivamo* (usmjerenim) bridovima*. Za brid* (x, y) *reći ćemo da izlazi iz* x *i ulazi u* y*;*
- *Postoji* $x_0 \in X$ *sa svojstvom da za svaki* $x \in X \setminus \{x_0\}$ *postoji jedinstven* put *od* x_0 *do* x*, odnosno niz bridova* $(x_0, x_1), (x_1, x_2), \ldots, (x_{k-1}, x_k), (x_k, x)$*. Njega nazivamo* korijenom*;*
- U x_0 *ne ulazi niti jedan brid.*

Vrhove iz kojih ne izlazi niti jedan brid nazivamo *završnim vrhovima* (ili listovima).

Definicija 1.1.2. Ekstenzivna igra *uređena je sedmorka* $\Gamma = (T, N, P, \mathcal{H}, C, \tau, u)$ *takva da vrijedi sljedeće:*

- $T = (X, E)$ *usmjereno je stablo s korijenom* x_0 *i skupom završnih vrhova* Z*;*
- $N = \{1, \ldots, n\}$ *skup je od* n *igrača;*
- $P : X \setminus Z \to N \cup \{0\}$ *funkcija je koja svakom nezavršnom vrhu pridružuje ili igrača ili Prirodu (koju shvaćamo kao igrača s oznakom* 0*). Ako funkcija* P *nekom vrhu pridružuje igrača, onda taj vrh nazivamo* vrhom odluke*, a ako mu pridružuje Prirodu, nazivamo ga* slučajnim vrhom*;*
- $\mathcal{H} = (H_i)_{i \in N}$ *svakom igraču* i *pridružuje particiju skupa* $P^{-1}(i)$*, odnosno skupa vrhova odluke u kojima je taj igrač na potezu. Skupove* $h \in H_i$ *nazivamo* skupovima

informacija *igrača i. Za svaki takav skup h pretpostavljamo da sa svakim putom u T dijeli najviše jedan vrh te da svi vrhovi u h imaju jednak broj izlaznih bridova;*

- $C = (C_h)_{h\in H}$*, pri čemu je* $H = \cup_{i\in N} H_i$*, svakom* $h \in H$ *pridružuje particiju skupa bridova koji izlaze iz vrhova u h. Particija* C_h *takva je da za svaki vrh* $x \in h$ *i svaki* $c \in C_h$*, c sadrži točno jedan brid koji izlazi iz x. Skupove* $x \in C_h$ *nazivamo* potezima *u h. Pretpostavljamo da je* $|C_h| \geq 2$ *za sve h;*

- $\tau = (\tau_x)_{x\in P^{-1}(0)}$ *svakom slučajnom vrhu pridružuje vjerojatnosnu distribuciju na skupu bridova koji iz njega izlaze. Pretpostavljamo da su svim bridovima pridružene pozitivne vjerojatnosti;*

- $u = (u_i)_{i\in N}$ *svakom igraču i pridružuje funkciju korisnosti* $u_i : Z \to \mathbb{R}$.

Analizirajmo sada primjer sa slike 1.1 u kontekstu definicije 1.1.2. Za početak, uočimo da u primjeru možemo prepoznati usmjereno stablo T sa skupom vrhova $X = \{x_0, \ldots, x_{13}\}$, korijenom x_0 i skupom E od 13 bridova. Skup završnih vrhova je $Z = \{x_6, \ldots, x_{13}\}$, a skup igrača $N = \{1, 2\}$. Nadalje, za funkciju $P : \{x_0, \ldots, x_5\} \to N \cup \{0\}$ vrijedi

$$P(x_0) = P(x_5) = 1, P(x_1) = P(x_2) = P(x_4) = 2, P(x_3) = 0.$$

Skupovi H_1 i H_2 skupova informacija prvog, odnosno drugog igrača jednaki su

$$H_1 = \{\{x_0\}, \{x_5\}\}, H_2 = \{\{x_1, x_2\}, \{x_4\}\}.$$

Većina skupova informacija $h \in H_1 \cup H_2$ jednočlana je pa su particije C_h pridružene njima trivijalne. Primjerice,

$$C_{\{x_4\}} = \{\{(x_4, x_{10})\}, \{(x_4, x_{11})\}\}.$$

Jedini je višečlani skup informacija $\{x_1, x_2\}$ pa je zanimljivo promotriti njemu pridruženu particiju. Naime, vrijedi

$$C_{\{x_1,x_2\}} = \{\{(x_1, x_6), (x_2, x_8)\}, \{(x_1, x_7), (x_2, x_9)\}\}.$$

Primijetimo da je potez $\{(x_1, x_6), (x_2, x_8)\}$ na slici 1.1 označen simbolom β_1, dok je potez $\{(x_1, x_7), (x_2, x_9)\}$ označen pomoću β_2.

Vrh x_3 slučajan je vrh, a njemu pridružena vjerojatnosna distribucija τ_{x_3} bridu (x_3, x_4) pridružuje vjerojatnost $\frac{1}{4}$, a bridu (x_3, x_5) vjerojatnost $\frac{3}{4}$.

Konačno, funkcije korisnosti u_1 i u_2 svakom završnom vrhu pridružuju korisnost za odgovarajućeg igrača. Tako je, primjerice, $u_1(x_7) = 7$ i $u_2(x_9) = 0$.

Od posebnog su interesa u teoriji igara igre kod kojih svaki igrač u svakome trenutku zna točno stanje igre.

Definicija 1.1.3. *Za ekstenzivnu igru kažemo da je s* potpunim informacijama *ako za svakog igrača i te za sve $h \in H_i$ vrijedi $|h| = 1$. U suprotnom, kažemo da se radi o igri s* nepotpunim informacijama.

Šah, kao jedna od igara u kojoj AlphaZero postiže nadljudsku razinu igranja, jedan je od najpoznatijih primjera igre s potpunim informacijama. Osim šaha, primjeri su i jednostavna igra križić-kružić te popularna društvena igra *Čovječe, ne ljuti se*. S druge strane, većina kartaških igara, poput pokera, igre su s nepotpunim informacijama, budući da igračima nije poznato kojim kartama raspolažu njihovi protivnici.

Kada u svakodnevnom govoru govorimo o igranju igara, često spominjemo *strategije* pojedinih igrača. U kontekstu algoritma AlphaZero, bitan nam je pojam *strategije* kao nečega što određuje vjerojatnosti odabira pojedinih poteza u nekom stanju igre. Teorija igara takav pojam formulira kao *bihevioralnu strategiju*, koja je formalno definirana definicijom 1.1.4.

Definicija 1.1.4. Bihevioralna strategija *igrača i preslikavanje je b_i koje svakom skupu informacija $h \in H_i$ igrača i pridružuje vjerojatnosnu distribuciju na skupu C_h. Skup bihevioralnih strategija igrača i označavamo pomoću B_i.*

Neke igre s kojima se svakodnevno susrećemo običavamo karakterizirati kao *kompetitivne*, neke kao *kooperativne*, dok neke ne svrstavamo niti u jednu od tih kategorija. No, kako formalizirati navedeno svojstvo *kompetitivnosti*? Kompetitivnost mnogih poznatih igara koje se mogu modelirati kao ekstenzivne igre proizlazi iz njihovog svojstva opisanog definicijom 1.1.5.

Definicija 1.1.5. *Za ekstenzivnu igru kažemo da je* sa sumom nula *ako za svaki $z \in Z$ vrijedi*

$$u_1(z) + \cdots + u_n(z) = 0.$$

Primjerice, ako se radi o ekstenzivnoj igri sa sumom nula za koju vrijedi $N = \{1, 2\}$, odnosno koju igraju samo dva igrača, ta su dva igrača u striktnoj opoziciji jedan prema drugome. Ako je korisnost za jednog igrača u nekom završnom stanju pozitivna (on pobjeđuje), onda će korisnost za drugog biti negativna (on gubi).

Kombinatorne igre

Različite igre običavaju se klasificirati na temelju više značajki. Neke su od značajki na osnovu kojih se vrši klasifikacija diskretnost, odnosno jesu li potezi diskretni ili se primjenjuju u realnom vremenu, te sekvencijalnost, to jest izvršavaju li se potezi sekvencijalno ili istovremeno. Kao što smo već naveli na početku odjeljka 1.1, ekstenzivne su igre model sekvencijalnih interakcija između agenata — dakle, one su sekvencijalne, a i diskretne.

Nadalje, ekstenzivne igre možemo još klasificirati na temelju sljedećih triju značajki:

- *Suma nula*: Radi li se o igri sa sumom nula;
- *Informacije*: Radi li se o igri s potpunim informacijama;
- *Determinizam*: Je li za ekstenzivnu igru $P^{-1}(0)$ jednak praznom skupu.

Igre s dva igrača koje su sa sumom nula, s potpunim informacijama, koje su determinističke, diskretne i sekvencijalne nazivaju se *kombinatornim igrama*. Šah, shogi i igra Go, savladane od strane AlphaZero algoritma, upravo su kombinatorne igre. Kombinatorne igre općenito su izvrsne za eksperimente umjetne inteligencije, budući da se radi o kontroliranim okolinama definiranim jednostavnim pravilama, koje, međutim, tipično iznjedruju kompleksan tijek igre, a koji predstavlja značajan istraživački izazov. Potonje dobro demonstrira igra Go, o čemu će detaljnije biti riječ u odjeljku 2.1.

1.2 Igranje igara — problem umjetne inteligencije

Kao što smo napomenuli na samom početku ovog poglavlja, AlphaZero rješava probleme igranja igara na ploči u kojima je aktualno stanje u potpunosti poznato i pravila su igre fiksirana. Problem igranja igara jedan je od problema kojima se bavi umjetna inteligencija i ustvari je varijanta problema pretraživanja prostora stanja, s prisutnim protivnikom. Naime, prilikom igranja igara u svakom je stanju potrebno donijeti optimalnu odluku o sljedećem potezu. Drukčije rečeno, potrebno je pronaći optimalnu strategiju. Uzmimo u obzir samo kombinatorne igre, opisane u prethodnom odjeljku, te formalno formulirajmo problem na sljedeći način.

Problem igranja igara možemo opisati, po uzoru na formulaciju iz [27], kao problem pretraživanja sa sljedećim komponentama:

- *Početno stanje* igre $s_0 \in S$, gdje je S skup (prostor) stanja;
- *Funkcija sljedbenika* succ : $S \to \wp(S)$, koja definira valjane (legalne) poteze igre (prijelaze među stanjima);
- *Test na završno stanje* terminal : $S \to \{\top, \bot\}$;
- *Isplatna funkcija* utility : terminal$^{-1}(\top) \times P \to \mathbb{R}$, gdje je P skup igrača igre. Isplatna funkcija uređenom paru završnog stanja s i igrača p pridružuje iznos nagrade koju igrač p dobiva u stanju s. Primjerice, u šahu je utility$(s, p) \in \{1, 0, -1\}$ za svako završno stanje s i za svakog igrača p. Vrijednosti isplatne funkcije u mnogim su igrama upravo 1, 0 ili -1, pri čemu vrijednost 1 odgovara pobjedi, 0 neriješenoj igri, a -1 gubitku.

Početno stanje igre i funkcija sljedbenika definiraju *stablo igre*.

Pretraživanje prostora stanja svodi se na pretraživanje usmjerenog grafa, čiji vrhovi predstavljaju stanja, a lukovi prijelaze između stanja. Krenuvši od početnog stanja problema, pokušavamo pronaći ciljno stanje, a slijed akcija koje nas vode do ciljnog stanja predstavljaju rješenje problema. Pretraživanjem usmjerenog grafa postepeno gradimo stablo pretraživanja. Stablo gradimo proširivanjem pojedinih čvorova, odnosno generiranjem svih sljedbenika nekog čvora pomoću funkcije sljedbenika. Pritom *čvorom n* nazivamo podatkovnu strukturu koja sačinjava stablo pretraživanja, a pohranjuje stanje, ali i još neke dodatne podatke. Točnije, za podatkovnu strukturu čvora n vrijedi $n = (s, d)$, gdje je s stanje, a d dubina čvora u stablu. Karakteristike pojedinog problema pretraživanja sljedeće su:

- $|S|$: broj stanja;
- b: *faktor grananja* stabla pretraživanja;
- d: *dubina optimalnog rješenja* u stablu pretraživanja;
- m: *maksimalna dubina* stabla pretraživanja (pri čemu je moguće da je jednaka ∞).

Postojanje informacije o aktualnom stanju igre te eksplicitnih pravila omogućuje analiziranje stanja, pri čemu analiza podrazumijeva evaluiranje svih mogućih poteza koje u nekom stanju možemo povući te odabir najboljeg poteza. Najjednostavniji pristup evaluaciji iteracija je kroz sve moguće poteze te rekurzivna evaluacija stanja nakon što je pojedini potez povučen. Propagiranjem rezultata igre unatrag, možemo procijeniti očekivanu vrijednost nekog poteza u bilo kojem stanju. Jedna moguća varijanta ove metode naziva se *minimaks* (engl. *minimax*).

Nazovimo igrače MAX (računalo) i MIN (protivnik). U sklopu metode minimaks nastojimo povući najisplativiji potez za igrača MAX, odnosno onaj koji maksimira njegov dobitak, pri čemu nam je u opoziciji protivnik, igrač MIN, kojemu je cilj povući potez koji je najmanje isplativ za igrača MAX, odnosno onaj koji minimizira njegov dobitak. Igrači igraju naizmjenično, tako da čvorovi stabla igre na parnoj udaljenosti od čvora koji predstavlja stanje koje analiziramo odgovaraju igraču MAX, a oni na neparnoj udaljenosti odgovaraju igraču MIN. U skladu s time, iterativno maksimiramo i minimiziramo konačni dobitak spuštanjem niz stablo igre. Pritom treba imati na umu da je u praksi protivnikova strategija nepoznata (i vjerojatno različita od strategije igrača MAX), zbog čega nije moguće savršeno predvidjeti protivnikove poteze.

Metoda minimaks pretražuje u dubinu, odnosno uvijek prvo proširuje najdublji čvor u stablu pretraživanja. S obzirom na to, njezina je prostorna složenost $O(m)$, gdje je m dubina stabla pretraživanja, a vremenska joj je složenost $O(b^m)$, gdje je b faktor grananja igre.

Ako je broj različitih stanja dovoljno malen da se sva u potpunosti analiziraju, što vrijedi za, primjerice, igru križić-kružić, koja ima samo 138 završnih stanja, nije problem spustiti se niz stablo igre iz bilo kojeg stanja te na opisani način odrediti najisplativiji potez.

Međutim, taj pristup nije adekvatan ni za igre srednje kompleksnosti. U slučaju takvih ili još kompleksnijih igara nije moguće za bilo koje stanje analizirati sva stanja dostupna iz njega pa je potrebno donositi vremenski ograničene i nesavršene odluke. Uz pomoć kombinacije pažljivog pretraživanja, kriterija zaustavljanja (*podrezivanja* — često se koristi takozvano alfa-beta podrezivanje) te pametnih unaprijed definiranih evaluacija pojedinih stanja, moguće je proizvesti računalne programe koji igraju kompleksne igre na solidnoj razini. Međutim, za postizanje nadljudske razine, koju doseže AlphaZero, potreban je drukčiji pristup.

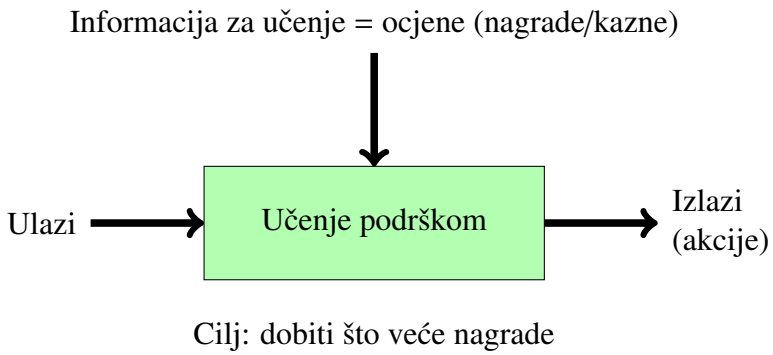
1.3 Strojno učenje podrškom

AlphaZero algoritam je strojnog učenja podrškom (engl. *reinforcement learning*). Cilj je ovog odjeljka dati objašnjenje što strojno učenje podrškom jest općenito te time stvoriti podlogu za razumijevanje algoritma AlphaZero, odnosno njegovih temelja.

Pristupi strojnog učenja tipično se dijele u tri kategorije: nadzirano učenje (engl. *supervised learning*), nenadzirano učenje (engl. *unsupervised learning*) te učenje podrškom. Dok nadzirano učenje podrazumijeva učenje funkcije klasifikacije ili regresije uz prisutnost „učitelja" koji daje primjere sastavljene od ulaznih i ciljnih izlaznih vrijednosti, a nenadzirano se učenje sastoji od samoorganizacije sustava i učenja pravilnosti u podacima bez učitelja, kod učenja podrškom učenje se odvija kroz interakciju s okolišem pokušajem i pogreškom. Ilustracije nadziranog, odnosno učenja podrškom (napravljene po uzoru na ilustracije iz [6]) mogu se vidjeti na slikama 1.2 i 1.3, redom.

Slika 1.2: Ilustracija nadziranog učenja.

Slika 1.3: Ilustracija učenja podrškom.



Kao što je prikazano na slikama, intuitivno možemo reći da, dok se nadzirano učenje odvija uz ciljne izlaze kao informacije za učenje, učenje podrškom podrazumijeva ocjene (nagrade, odnosno kazne) kao informacije za učenje, pri čemu je upravo maksimiranje nagrade cilj učenja.

Strojno učenje podrškom može se smatrati klasom algoritama za rješavanje problema koji se mogu formulirati kao Markovljevi procesi odlučivanja. U narednom ćemo pododjeljku definirati Markovljeve procese odlučivanja i objasniti njihove središnje koncepte (koji su i središnji koncepti strojnog učenja podrškom), a u pododjeljku iza njega predstavit ćemo alternirajuće Markovljeve igre (čiji su, kao što ćemo u navedenom odjeljku i navesti, Markovljevi procesi odlučivanja poseban slučaj), a čiji formalizam AlphaZero slijedi te koje su također model problema koje strojno učenje podrškom rješava.

Markovljevi procesi odlučivanja

Markovljev proces odlučivanja (engl. *Markov decision process*, MDP) matematički je model problema sekvencijalnog donošenja odluka, odnosno učenja njegovog rješavanja kroz interakcije koje donose nagrade. Donositelja odluka nazivamo *agentom*, a ono s čim stupa u interakciju *okolišem*. Agent i okoliš komuniciraju u diskretnim trenucima $t = 0, 1, 2, 3, \ldots$, pri čemu u svakom trenutku t agent prima informaciju o aktualnom *stanju* S_t okoliša, na temelju koje odabire *akciju* A_t, na koju potom okoliš reagira predstavljanjem nove situacije agentu (novog stanja S_{t+1}), kao i davanjem odgovarajuće *nagrade* R_{t+1}.

S obzirom na to da su za AlphaZero algoritam relevantni konačni problemi sekvencijalnog donošenja odluka, u nastavku, kada govorimo o Markovljevim procesima odlučivanja, pretpostavljamo njihovu konačnost. Formalno, (konačan) Markovljev proces odlučivanja definiran je sljedećom definicijom.

Definicija 1.3.1. Konačan Markovljev proces odlučivanja *uređena je petorka* $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$ *takva da vrijedi sljedeće:*

- $\mathcal{S}$ *je konačan skup čije elemente nazivamo* stanjima*;*
- $\mathcal{A}$ *je konačan skup čije elemente nazivamo* akcijama*. Ustvari za svaki* $s \in \mathcal{S}$ *postoji skup akcija* $\mathcal{A}(s)$[1] $\subseteq \mathcal{A}$*, ali radi jednostavnosti ponekad pretpostavljamo poseban slučaj kada je skup akcija jednak za sva stanja te ga tada označavamo pomoću* $\mathcal{A}$*;*
- $\mathcal{R} \subseteq \mathbb{R}$ *je konačan skup čije elemente nazivamo* nagradama*;*
- $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ *je* funkcija dinamike *takva da za sve* $s', s \in \mathcal{S}$*,* $r \in \mathcal{R}$ *i* $a \in \mathcal{A}(s)$ *vrijedi*

$$p(s', r|s, a) = \mathbb{P}(S_t = s', R_t = r \,|\, S_{t-1} = s, A_{t-1} = a). \tag{1.1}$$

Pritom notacija pomoću „|" na lijevoj strani jednadžbe (1.1) *dolazi od notacije uvjetne vjerojatnosti (s desne strane jednadžbe) i koristi se kao podsjetnik na to da* p *za neke* $s \in \mathcal{S}$ *i* $a \in \mathcal{A}(s)$ *definira vjerojatnosnu distribuciju, odnosno da vrijedi*

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \qquad \text{za sve } s \in \mathcal{S} \text{ i } a \in \mathcal{A}(s).$$

- $\gamma \in \langle 0, 1]$ *je* diskontirajući faktor.

Za dane vrijednosti $s \in \mathcal{S}$ i $a \in \mathcal{A}(s)$ te $s' \in \mathcal{S}$ i $r \in \mathcal{R}$, funkcija dinamike uređenoj četvorci (s', r, s, a) pridružuje uvjetnu vjerojatnost da je stanje u trenutku t s' te da je nagrada u istom trenutku r, uz uvjet da su stanje, odnosno akcija, u prethodnom trenutku $t - 1$ bili s, odnosno a. U Markovljevom procesu odlučivanja, vjerojatnosti koje su vrijednosti funkcije p u potpunosti opisuju dinamiku okoliša. Odnosno, prijelazi između stanja zadovoljavaju Markovljevo svojstvo

$$\mathbb{P}(S_{t+1} = s_{t+1}, R_{t+1} = r_{t+1} \,|\, S_0 = s_0, A_0 = a_0, \ldots, S_t = s_t, A_t = a_t) = \\ \mathbb{P}(S_{t+1} = s_{t+1}, R_{t+1} = r_{t+1} \,|\, S_t = s_t, A_t = a_t),$$

što opravdava naziv modela. Drugim riječima, vjerojatnost da slučajne varijable S_{t+1} i R_{t+1} poprime bilo koju od mogućih vrijednosti ovisi samo o neposredno prethodnim stanju i akciji te stoga ne ovisi o stanjima i akcijama koje su prethodile njima.

[1] Akcije koje su elementi skupa $\mathcal{A}(s)$ za neko stanje s ubuduće ćemo nazivati *legalnim akcijama* u stanju s.

Pomoću vrijednosti funkcije dinamike p moguće je izračunati mjere koje daju raznolike informacije o okolišu. Primjerice, možemo definirati *vjerojatnosnu funkciju prijelaza* $\mathcal{P} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ kao

$$\mathcal{P}(s'|s, a) := \mathbb{P}(S_t = s' \,|\, S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a), \quad s', s \in \mathcal{S},\ a \in \mathcal{A}(S).$$

Također, možemo definirati *funkciju nagrade* $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, koja pojedinom uređenom paru stanja i akcije pridružuje očekivanu nagradu za poduzimanje dane akcije u odgovarajućem stanju. Naime, r definiramo kao u jednadžbi

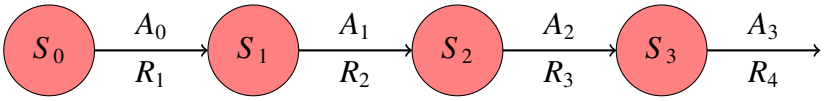
$$r(s, a) := \mathbb{E}(R_t \,|\, S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a), \quad s \in \mathcal{S},\ a \in \mathcal{A}(s). \tag{1.2}$$

Funkciju nagrade možemo definirati i kao funkciju triju varijabli $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, koja uređenoj trojci stanja, akcije i sljedećeg stanja pridružuje očekivanu nagradu za poduzimanje dane akcije u odgovarajućem stanju, rezultat čega je prelazak u navedeno sljedeće stanje, za koju vrijedi

$$r(s, a, s') := \mathbb{E}(R_t \,|\, S_{t-1} = s, A_{t-1} = a, S_t = s') = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{\mathcal{P}(s'|s, a)}, \quad s, s' \in \mathcal{S},\ a \in \mathcal{A}(s).$$

Kao što smo već napomenuli na početku odjeljka, komponente koje čine Markovljev proces odlučivanja su agent (kontrolor) i okoliš (kontrolirani sustav). Naveli smo da, intuitivno, u svakom trenutku t donositelj odluka (agent) opaža trenutno stanje okoliša S_t i odabire akciju A_t te da odabrana akcija obično utječe na sljedeće stanje okoliša S_{t+1} i nagradu R_{t+1}. Taj se proces potom ponavlja, što rezultira nizom stanja, akcija i nagrada $(S_0, A_0, R_1, \ldots, S_t, A_t, R_{t+1}, \ldots)$. Navedeno je prikazano slikom 1.4.

Slika 1.4: Prelasci stanje-akcija i pripadne nagrade.



Bitno je naglasiti da agent nema direktnu kontrolu nad okolišem, već samo indirektnu stohastičku kontrolu putem odabranih akcija.

Dakle, agentova je zadaća rješavanje problema sekvencijalnog donošenja odluka u stohastičkom okolišu te se može reći da on utoliko predstavlja „rješenje". S druge strane, za okoliš možemo reći da predstavlja „problem". Primjerice, igru križić-kružić možemo modelirati na način da agentom smatramo jednog od igrača, a da okoliš uključuje samu igru i drugog igrača.

Kontrolna strategija $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ funkcija je koja akciji a i stanju s pridružuje vjerojatnost da će agent odabrati akciju a u stanju s. Također je moguće definirati i determinističku kontrolnu strategiju $\pi : \mathcal{S} \to \mathcal{A}$ koja stanja preslikava u akcije.

Agent uči primanjem nagrada za interakcije s okolišem. Za razliku od nadziranog učenja, gdje su, kao što smo već naveli, informacije za učenje eksplicitne ciljne izlazne vrijednosti za dane ulaze, specifičnost problema učenja podrškom ta je što su jedine informacije za učenje trenutne nagrade — nemamo primjere za učenje oblika $(s, \hat{\pi}(s))$, gdje je $\hat{\pi}(s)$ idealna akcija u stanju s. Pritom agent sam utječe na distribuciju primjera za učenje nizom akcija koje odabire. Prilikom pretraživanja stanja koja čine okoliš, javlja se poznati problem kompromisa između istraživanja i iskorištavanja (engl. *exploration-exploitation tradeoff*) — radi se o problemu koji se često javlja u sustavima koji uče sekvencijalnim donošenjem odluka nesigurne isplativosti. Dilema se svodi na odabir između ponavljanja odluka koje su u prošlosti bile isplative, odnosno iskorištavanja starih informacija o dobivanju nagrade za prethodno istražena stanja i akcije, te pretraživanja nepoznatih stanja, odnosno donošenja novih odluka u nadi da će dovesti do još većih nagrada.

Uz dano početno stanje S_t, cilj je maksimirati očekivanje diskontirane kumulativne nagrade (također nazivane *povratom* (engl. *return*)), dane jednadžbom

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{1.3}$$

Zadatak koji agent rješava može, ali ne mora imati prirodan završetak. Zadatke koji imaju prirodan završetak, poput igara, nazivamo epizodnim zadacima (engl. *episodic tasks*), dok zadatke koji ga nemaju i mogu se nastavljati dovijeka, poput učenja kretanja prema naprijed, nazivamo kontinuiranim zadacima (engl. *continuing tasks*). Niz vremenskih koraka od početka do kraja jednog epizodnog zadatka naziva se *epizodom*. Ukoliko se radi o epizodnom zadatku, odnosno ako postoji završno stanje S_T, u jednadžbi (1.3) podrazumijevamo da vrijedi $R_{t+k+1} = 0$ kada je $t + k + 1 > T$. Možemo primijetiti da doprinos pojedinih nagrada ukupnoj sumi može ovisiti o njihovom vremenskom odmaku u odnosu na trenutak koji odgovara početnom stanju — razlikujemo takozvane *odgođene* (buduće) nagrade i trenutne. Značaj budućih nagrada običava se (eksponencijalno) umanjiti jer, intuitivno govoreći, preferiramo trenutne nagrade u odnosu na dugoročne. Ako postoji završno stanje, diskontirajući faktor γ može poprimiti bilo koju vrijednost iz intervala $\langle 0, 1]$. Inače, obično je $\gamma < 1$ kako bi se osigurala konvergencija reda u jednadžbi (1.3), a time i dobra definicija diskontirane kumulativne nagrade.

Dakle, agentov je cilj pronaći kontrolnu strategiju π koja maksimira očekivanje povrata, odnosno onu koja je element skupa

$$\arg\max_{\pi} \mathbb{E}(G_t).$$

Jedan su od centralnih koncepata Markovljevih procesa odlučivanja *funkcije vrijednosti*, koje se koriste za pronalaženje poželjnih kontrolnih strategija. Za danu kontrolnu strategiju π, *funkcija vrijednosti stanja* (engl. *state value function*) v_π definirana je jednadžbom (1.4):

$$\begin{aligned} v_\pi(s) &:= \mathbb{E}_\pi(G_t \,|\, S_t = s) && (1.4)\\ &= \mathbb{E}_\pi(R_{t+1} + \gamma G_{t+1} \,|\, S_t = s) && (1.5)\\ &= \mathbb{E}_\pi(R_{t+1} + \gamma v_\pi(S_{t+1}) \,|\, S_t = s) && (1.6)\\ &= \sum_a \pi(a, s) \sum_{s',r} p(s', r|s, a) \cdot \left(r + \gamma v_\pi(s')\right). && (1.7) \end{aligned}$$

Funkcija vrijednosti stanja stanje $s \in \mathcal{S}$ preslikava u očekivani povrat uz s kao početno stanje i kontrolnu strategiju π. Valja napomenuti da je zapis (1.7) dobiven iz (1.6) korištenjem jednadžbe (1.2). U jednadžbama (1.4)–(1.6) simboli očekivanja indeksirani su simbolom π radi naznačivanja da su vrijednosti uvjetovane praćenjem kontrolne strategije π. Također, primijetimo da zapis (1.7) ne sadrži „vremensku odrednicu", kao i da je domena funkcije vrijednosti stanja skup $\mathcal{S}$ — skup stanja, bez podataka o trenucima u kojima su opažena. Naime, bez obzira na to u kojemu trenutku agent dobije informaciju da je stanje okoliša $s \in \mathcal{S}$, očekivani je povrat je jednak. To je posljedica toga što ustvari promatramo stacionarne Markovljeve procese odlučivanja, čija funkcija dinamike ne ovisi o vremenu, kao i stacionarne kontrolne strategije, odnosno kontrolne strategije koje se ne mijenjaju kroz vrijeme.

Slično, definira se *funkcija vrijednosti akcije* (engl. *action value function*) — za danu kontrolu strategiju π, dana je jednadžbom (1.8):

$$\begin{aligned} q_\pi(s, a) &:= \mathbb{E}_\pi(G_t \,|\, S_t = s, A_t = a) && (1.8)\\ &= \mathbb{E}_\pi(R_{t+1} + \gamma G_{t+1} \,|\, S_t = s, A_t = a) && (1.9)\\ &= \mathbb{E}_\pi(R_{t+1} + \gamma v_\pi(S_{t+1}) \,|\, S_t = s, A_t = a) && (1.10)\\ &= \sum_{s',r} p(s', r|s, a) \cdot \left(r + \gamma v_\pi(s')\right). && (1.11) \end{aligned}$$

Slično kao funkcija vrijednosti stanja, funkcija vrijednosti akcije uređeni par stanja i akcije (s, a) preslikava u očekivani povrat uz $s \in \mathcal{S}$ kao početno stanje u kojem se odabire akcija $a \in \mathcal{A}(s)$ i kontrolnu strategiju π. Ponovno, zapis (1.11) dobiven je iz (1.10) korištenjem jednadžbe (1.2). Jednadžbe (1.7) i (1.11) nazivaju se *Bellmanovim jednadžbama*. One su osnova za takozvanu *procjenu kontrolne strategije* (engl. *policy evaluation*), koja računa funkcije vrijednosti za danu kontrolnu strategiju π. Procjena kontrolne strategije također se naziva *problemom predviđanja* (engl. *prediction problem*).

Također se definira *funkcija prednosti* (engl. *advantage function*) — za danu kontrolnu strategiju π, definirana je jednadžbom

$$a_\pi(s,a) := q_\pi(s,a) - v_\pi(s), \quad s \in \mathcal{S},\ a \in \mathcal{A}(s). \tag{1.12}$$

Vrijednosti funkcije prednosti indikatori su toga koliko je isplativo odlučiti se za neku akciju u određenom stanju u odnosu na poduzimanje drugih akcija u istom stanju.

Sada možemo definirati parcijalni uređaj na skupu svih mogućih kontrolnih strategija.

Definicija 1.3.2. *Za kontrolnu strategiju π kažemo da je* bolja ili jednaka *kontrolnoj strategiji π' ako za svako stanje $s \in \mathcal{S}$ vrijedi*

$$v_\pi(s) \geq v_{\pi'}(s). \tag{1.13}$$

Uvijek postoji[2] barem jedna kontrola strategija koja je bolja ili jednaka svim drugim kontrolnim strategijama. Tu kontrolnu strategiju nazivamo *optimalnom kontrolnom strategijom*. Iako je moguće da postoji više od jedne optimalne kontrolne strategije, običaj je sve ih označavati pomoću π_*. Svima njima odgovara ista funkcija vrijednosti stanja. Nju nazivamo *optimalnom funkcijom vrijednosti stanja* i označavamo je pomoću v_*. Ona je definirana jednadžbom

$$v_*(s) := \max_\pi v_\pi(s), \quad s \in \mathcal{S}. \tag{1.14}$$

Optimalne kontrolne strategije također dijele i istu *optimalnu funkciju vrijednosti akcije*, koju označavamo pomoću q_*. Ona je definirana jednadžbom

$$q_*(s,a) := \max_\pi q_\pi(s,a), \quad s \in \mathcal{S},\ a \in \mathcal{A}(s). \tag{1.15}$$

Danom uređenom paru stanja i akcije (s,a) optimalna funkcija vrijednosti akcije pridružuje očekivani povrat u slučaju poduzimanja akcije a u stanju s te potom slijeđenja optimalne kontrolne strategije.

Optimalna funkcija vrijednosti stanja i optimalna funkcija vrijednosti akcije zadovoljavaju takozvane Bellmanove jednadžbe optimalnosti, dane jednadžbama

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}\left(R_{t+1} + \gamma v_*(S_{t+1}) \,|\, S_t = s, A_t = a\right) \\ &= \max_a \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma v_*(s')) \end{aligned} \quad , \quad s \in \mathcal{S}; \tag{1.16}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left(R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,|\, S_t = s, A_t = a\right) \\ &= \sum_{s',r} p(s', r|s, a) \cdot \left(r + \gamma \max_{a'} q_*(s', a')\right) \end{aligned} \quad , \quad s \in \mathcal{S},\ a \in \mathcal{A}(s). \tag{1.17}$$

[2]Dokaz egzistencije optimalne kontrolne strategije može se pronaći u članku [18].

Slično kao u slučajevima jednadžbi koje određuju funkcije vrijednosti stanja, odnosno akcije, u jednadžbama (1.16) i (1.17) implicitno je korištena jednadžba (1.2).

Računanje funkcije vrijednosti za određenu kontrolnu strategiju omogućava nam da pronađemo bolje kontrolne strategije. Naime, pretpostavimo da smo odredili funkciju vrijednosti stanja v_π za proizvoljnu deterministička kontrolnu strategiju π. Neka je $s \in \mathcal{S}$ proizvoljno stanje. Možemo se zapitati bismo li trebali promijeniti kontrolnu strategiju tako da stanju s pridružuje akciju $a \neq \pi(s)$. Znamo koliko je dobro slijediti aktualnu kontrolnu strategiju iz stanja s (tu nam informaciju daje vrijednost $v_\pi(s)$), no, bi li bilo bolje kad bismo kontrolnu strategiju promijenili na opisani način? Na to pitanje možemo odgovoriti tako da razmotrimo odabiranje akcije a u stanju s te potom slijeđenje aktualne kontrolne strategije π. Vrijednost takvog ponašanja daje $q_\pi(s, a)$. Glavni je kriterij je li ta vrijednost veća od $v_\pi(s)$. Ako je veća (ako je bolje odabrati a u stanju s te potom slijediti π, nego što bi bilo cijelo vrijeme slijediti π), onda je za očekivati da je bolje da agent odabere akciju a svaki put kad se nađe u stanju s i da je nova kontrolna strategija općenito bolja. Istinitost toga poseban je slučaj općenitog rezultata iskazanog teoremom 1.3.3, čiji se dokaz može pronaći u knjizi [26, Poglavlje 4].

Teorem 1.3.3 (Teorem poboljšanja kontrolne strategije)**.** *Neka su π i π' deterministčke kontrolne strategije takve da za svaki $s \in \mathcal{S}$ vrijedi*

$$q_\pi\left(s, \pi'(s)\right) \geq v_\pi(s). \tag{1.18}$$

Tada je kontrolna strategija π' bolja ili jednaka kontrolnoj strategiji π. To jest, za svaki $s \in \mathcal{S}$ vrijedi

$$v_{\pi'}(s) \geq v_\pi(s). \tag{1.19}$$

Nadalje, ako je za bilo koje stanje nejednakost u (1.18) *stroga, tada za to stanje mora vrijediti stroga nejednakost u* (1.19)*.*

Za kontrolne strategije π i π' opisane u prethodnom odlomku vrijedi $\pi'(s') = \pi(s')$, za svako stanje $s' \in \mathcal{S} \setminus \{s\}$ i $\pi'(s) = a \neq \pi(s)$. Stoga nejednakost (1.18) očito vrijedi za sva stanja različita od s. Dakle, ako je $q_\pi(s, a) > v_\pi(s)$, tada je promijenjena kontrolna strategija uistinu bolja od π.

Do sada smo proučili kako za danu kontrolnu strategiju i pripadnu funkciju vrijednosti ocijeniti promjenu vrijednosti te kontrolne strategije na jednom elementu domene na određenu akciju. Prirodni je nastavak toga proučiti kako promjena vrijednosti kontrolne strategije na bilo koju od legalnih akcija u nekom stanju utječe na vrijednost funkcije vrijednosti te odabrati akciju koja se čini najboljom na temelju vrijednosti $q_\pi(s, a)$. Drugim

riječima, nastavak je promotriti novu *pohlepnu* kontrolnu strategiju π', danu jednadžbom

$$\begin{aligned}\pi'(s) &:= \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}(R_{t+1} + \gamma v_\pi(S_{t+1}) \,|\, S_t = s, A_t = a) \\ &= \arg\max_a \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma v_\pi(s'))\,.\end{aligned} \tag{1.20}$$

Pritom, iako je $\arg\max_a q_\pi(s,a)$ možda skup kardinalnosti veće od jedan, pomoću $\pi'(s)$ označavamo i bilo koju akciju koja je njegov element — slično kao što smo sve optimalne kontrolne strategije označili pomoću π_*. Pohlepna kontrolna strategija u pojedinom stanju odabire akciju koja kratkoročno (nakon jednog koraka gledanja unaprijed) djeluje najbolje prema v_π. Takva pohlepna kontrolna strategija po konstrukciji zadovoljava uvjete teorema 1.3.3 pa znamo da je bolja ili jednaka originalnoj kontrolnoj strategiji. Proces stvaranja nove kontrolne strategije koji poboljšava originalnu kontrolnu strategiju činjenjem je pohlepnom u odnosu na funkciju vrijednosti stanja originalne kontrolne strategije naziva se *poboljšanjem kontrolne strategije* (engl. *policy improvement*). Učenje se obično sastoji od prepletenih procjene kontrolne strategije i poboljšanja kontrolne strategije.

Pretpostavimo da je nova kontrolna strategija π' jednako dobra kao originalna π. Tada je $v_\pi = v_{\pi'}$ i iz jednadžbe (1.20) slijedi da za sve $s \in \mathcal{S}$ vrijedi

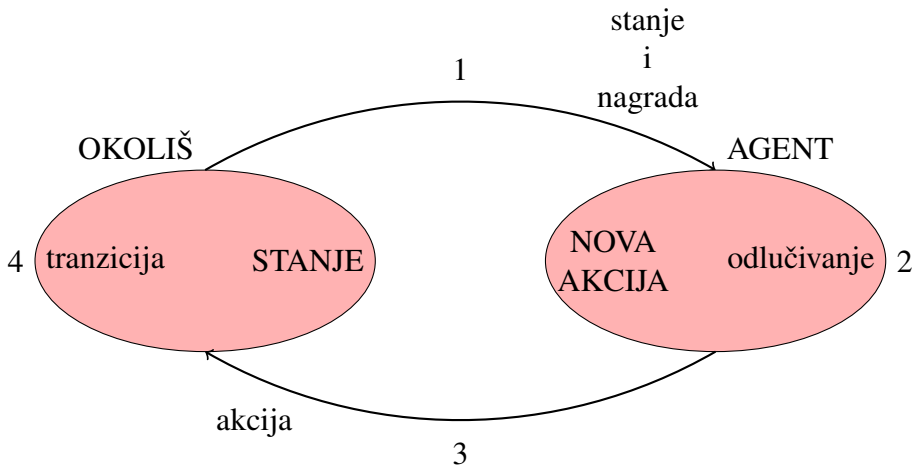
$$\begin{aligned}v_{\pi'}(s) &= \max_a \mathbb{E}(R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \,|\, S_t = s, A_t = a) \\ &= \max_a \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma v_{\pi'}(s'))\,.\end{aligned}$$

Možemo vidjeti da smo dobili izraz koji odgovara jednadžbi (1.16), odnosno Bellmanovoj jednadžbi optimalnosti, iz čega slijedi da je $v_{\pi'}$ jednaka v_* te da su i π i π' optimalne kontrolne strategije. Dakle, poboljšanje kontrolne strategije mora dati strogo bolju kontrolnu strategiju, osim kad je originalna kontrolna strategija već optimalna.

Koncepte do sada predstavljene u ovome odjeljku možemo sumirane prikazati ilustracijom takozvanog ciklusa učenja podrškom, vidljivog na slici 1.5.

Ciklus započinje agentovim proučavanjem okoliša, koje rezultira dobivanjem informacije o stanju i nagradi, što je na slici 1.5 označeno brojem 1. Agent iskorištava informaciju o stanju za donošenje odluke o sljedećoj akciji koju će izvršiti, što je na slici označeno brojem 2. Agent tada izvršava korak 3 sa slike, odnosno prosljeđuje akciju okolišu kako bi ga kontrolirao na sebi povoljan način. Konačno, u koraku 4, stanje okoliša obično se mijenja kao posljedica izvršavanja agentove akcije u prethodnom stanju. Opisani se ciklus dalje ponavlja.

Slika 1.5: Ciklus strojnog učenja podrškom.



Alternirajuće Markovljeve igre

Igre s dva igrača i sumom nula u kojima igrači naizmjence povlače poteze mogu se formulirati kao alternirajuće Markovljeve igre (engl. *alternating Markov games*, AMG). Alternirajuće Markovljeve igre razlikuju se od Markovljevih procesa odlučivanja po svojoj dvo-agentskoj prirodi.

Definicija 1.3.4. Alternirajuća Markovljeva igra *uređena je sedmorka* $(\mathcal{S}_1, \mathcal{S}_2, \mathcal{A}_1, \mathcal{A}_2, \mathcal{R}, p, \gamma)$ *takva da vrijedi sljedeće:*

- $\mathcal{S}_1$ *i* $\mathcal{S}_2$ *su skupovi* stanja *prvog, odnosno drugog agenta, redom;*
- $\mathcal{A}_1$ *i* $\mathcal{A}_2$ *su skupovi* akcija *prvog, odnosno drugog agenta, redom. Ustvari za svaki* $s_1 \in \mathcal{S}_1$ *postoji skup akcija* $\mathcal{A}_1(s_1) \subseteq \mathcal{A}_1$ *te za svaki* $s_2 \in \mathcal{S}_2$ *postoji skup akcija* $\mathcal{A}_2(s_2) \subseteq \mathcal{A}_2$*, ali radi jednostavnosti ponekad pretpostavljamo poseban slučaj kada je skup akcija jednak za sva stanja pojedinog agenta te ga tada označavamo pomoću* $\mathcal{A}_1$ *(za prvog agenta), odnosno* $\mathcal{A}_2$ *(za drugog agenta);*
- $\mathcal{R} \subseteq \mathbb{R}$ *je skup čije elemente nazivamo* nagradama*;*
- $p : \mathcal{S}_2 \times \mathcal{R} \times \mathcal{S}_1 \times \mathcal{A}_1 \cup \mathcal{S}_1 \times \mathcal{R} \times \mathcal{S}_2 \times \mathcal{A}_2 \to [0, 1]$ *je* funkcija dinamike *takva da za sve* $s' \in \mathcal{S}_2, s \in \mathcal{S}_1, r \in \mathcal{R}$ *i* $a \in \mathcal{A}_1(s)$ *vrijedi jednakost* (1.1) *(te da navedena jednakost vrijedi za sve* $s' \in \mathcal{S}_1, s \in \mathcal{S}_2, r \in \mathcal{R}$ *i* $a \in \mathcal{A}_2(s)$*);*
- γ *je kao u definiciji 1.3.1.*

Slično kao u pododjeljku *Markovljevi procesi odlučivanja* odjeljka 1.3, možemo definirati vjerojatnosnu funkciju prijelaza i funkciju nagrade. Neka je $i \in \{1, 2\}$. Najprije definirajmo *vjerojatnosnu funkciju prijelaza* $\mathcal{P} : \mathcal{S}_2 \times \mathcal{S}_1 \times \mathcal{A}_1 \cup \mathcal{S}_1 \times \mathcal{S}_2 \times \mathcal{A}_2 \to [0, 1]$ kao

$$\mathcal{P}(s'|s, a) := \mathbb{P}(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a), \quad s' \in \mathcal{S}_{3-i},\ s \in \mathcal{S}_i,\ a \in \mathcal{A}_i(s).$$

Funkciju nagrade možemo definirati kao funkciju dviju varijabli $r : \mathcal{S}_1 \times \mathcal{A}_1 \cup \mathcal{S}_2 \times \mathcal{A}_2 \to \mathbb{R}$ danu jednadžbom

$$r(s, a) := \mathbb{E}(R_t \mid S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}_{3-i}} p(s', r|s, a), \quad s \in \mathcal{S}_i,\ a \in \mathcal{A}_i(s). \tag{1.21}$$

Također, funkciju nagrade možemo definirati i kao funkciju triju varijabli $r : \mathcal{S}_1 \times \mathcal{A}_1 \times \mathcal{S}_2 \cup \mathcal{S}_2 \times \mathcal{A}_2 \times \mathcal{S}_1 \to \mathbb{R}$ pomoću jednadžbe

$$\begin{aligned} r(s, a, s') &:= \mathbb{E}(R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s') \\ &= \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{\mathcal{P}(s'|s, a)}, \quad s \in \mathcal{S}_i,\ a \in \mathcal{A}_i(s),\ s' \in \mathcal{S}_{3-i}. \end{aligned} \tag{1.22}$$

Funkcija nagrade ponekad se modelira i kao $r : \mathcal{S}_1 \cup \mathcal{S}_2 \to \mathbb{R}$.

Markovljevi procesi odlučivanja mogu se smatrati posebnim slučajem alternirajućih Markovljevih igara kada je $|\mathcal{S}_2| = 0$ (pa onda i $|\mathcal{A}_2| = 0$).

Budući da u alternirajućim Markovljevim igrama postoje dva agenta, procjena kontrolne strategije obuhvaća dvije kontrolne strategije $\pi_1 : \mathcal{A}_1 \times \mathcal{S}_1 \to [0, 1]$ i $\pi_2 : \mathcal{A}_2 \times \mathcal{S}_2 \to [0, 1]$. Bellmanove jednadžbe za procjenu kontrolne strategije dane su jednadžbama

$$v_{\pi_1}(s) = \sum_{a \in \mathcal{A}_1(s)} \pi_1(a, s) \sum_{s' \in \mathcal{S}_2, r} p(s', r|s, a) \cdot \left(r + \gamma v_{\pi_2}(s')\right), \quad s \in \mathcal{S}_1; \tag{1.23}$$

$$v_{\pi_2}(s) = \sum_{a \in \mathcal{A}_2(s)} \pi_2(a, s) \sum_{s' \in \mathcal{S}_1, r} p(s', r|s, a) \cdot \left(r + \gamma v_{\pi_1}(s')\right), \quad s \in \mathcal{S}_2. \tag{1.24}$$

Također, možemo definirati i Bellmanove jednadžbe za funkcije vrijednosti akcije. One su dane jednadžbama

$$q_{\pi_1}(s, a) = \sum_{s' \in \mathcal{S}_2, r} p(s', r|s, a) \cdot \left(r + \gamma \sum_{a' \in \mathcal{A}_2(s')} \pi_2(a', s') q_{\pi_2}(s', a')\right), \quad s \in \mathcal{S}_1,\ a \in \mathcal{A}_1(s); \tag{1.25}$$

$$q_{\pi_2}(s, a) = \sum_{s' \in \mathcal{S}_1, r} p(s', r|s, a) \cdot \left(r + \gamma \sum_{a' \in \mathcal{A}_1(s')} \pi_1(a', s') q_{\pi_1}(s', a')\right), \quad s \in \mathcal{S}_2,\ a \in \mathcal{A}_2(s). \tag{1.26}$$

Nazovimo igrača kojemu odgovara π_1 igračem MAX i igrača kojemu odgovara π_2 igračem MIN te pretpostavimo da je π_2 optimalna „protivna kontrolna strategija" naprama π_1. Tada jednadžbu (1.25) možemo zapisati na način izražen u jednadžbi (1.27). π_2 mijenjamo min operatorom jer se problem, budući da je kontrolna strategija π_1 fiksirana, reducira na Markovljev proces odlučivanja s jednim agentom u kojemu agent nastoji minimizirati primljene nagrade.

$$q_{\pi_1}(s,a) = \sum_{s' \in \mathcal{S}_2, r} p(s', r|s, a) \cdot \Bigg(r + \gamma \min_{a' \in \mathcal{A}_2(s')} \sum_{s'' \in \mathcal{S}_1, r'} p(s'', r'|s', a') \cdot \Bigg(r' + \sum_{a'' \in \mathcal{A}_1(s'')} \pi_1(a'', s'') q_{\pi_1}(s'', a'') \Bigg) \Bigg), \quad s \in \mathcal{S}_1,\ a \in \mathcal{A}_1(s) \tag{1.27}$$

Uz pretpostavku da stanja iz skupa $\mathcal{S}_1$ odgovaraju MAX igraču, a stanja iz skupa $\mathcal{S}_2$ igraču MIN, Bellmanove jednadžbe za optimalne funkcije vrijednosti stanja, koje također impliciraju Nashovu ravnotežu[3], dane su jednadžbama

$$v_1^*(s) = \max_{a \in \mathcal{A}_1(s)} \sum_{s' \in \mathcal{S}_2, r} p(s', r|s, a) \cdot \left(r + \gamma v_2^*(s')\right), \quad s \in \mathcal{S}_1; \tag{1.28}$$

$$v_2^*(s) = \min_{a \in \mathcal{A}_2(s)} \sum_{s' \in \mathcal{S}_1, r} p(s', r|s, a) \cdot \left(r + \gamma v_1^*(s')\right), \quad s \in \mathcal{S}_2. \tag{1.29}$$

Umjesto da alternirajuće Markovljeve igre sa sumom nula promatramo kao uređene sedmorke, možemo ih modelirati i kao uređene petorke $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$ takve da svako stanje iz skupa stanja $\mathcal{S}$ sadrži indikator o tome koji igrač u njemu povlači potez. U tom je slučaju $\mathcal{A}(s)$ skup legalnih akcija u stanju $s \in \mathcal{S}$, ali radi jednostavnosti ponekad pretpostavljamo poseban slučaj kad je skup akcija jednak u svakom stanju i označavamo ga pomoću $\mathcal{A}$. Analogno kao i ranije u radu, definiramo vjerojatnosnu funkciju prijelaza $\mathcal{P} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ i funkciju nagrade $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

Funkciju nagrade alternativno možemo modelirati kao $r : \mathcal{S} \to \mathbb{R}$. U nastavku prihvaćamo takvu formulaciju i usredotočujemo se na igre s dva igrača, sumom nula, determinističkim prijelazima između stanja te pretpostavljamo da za sva nezavršna stanja vrijedi $r(s) = 0$.

Naime, epizodne zadatke strojnog učenja podrškom često karakterizira jednakost svih nagrada osim posljednje nuli. Primjerice, u igri križić-kružić, funkcija nagrade vrlo se često modelira na način da su sve nagrade jednake 0 osim posljednje nagrade, koja je jednaka 1 u slučaju igračeve pobjede, -1 u slučaju gubitka i 0 u slučaju neriješene igre. Posljedično,

[3] Definicija Nashove ravnoteže može se pronaći u definiciji 2.6., na 30. stranici bilješki [19].

određivanje relativne vrijednosti pojedinih akcija koje agent može poduzeti iz nekog nezavršnog stanja dodatno je otežano, što se također u literaturi često naziva problemom pridruživanja zasluga (engl. *temporal credit assignment problem*). Problem pridruživanja zasluga je, uz već u prethodnom odjeljku spomenuti kompromis između istraživanja i iskorištavanja, vjerojatno centralni problem u teoriji i praksi strojnog učenja podrškom.

S obzirom na to da govorimo o igrama s deterministčkim prijelazima, umjesto vjerojatnosne funkcije prijelaza u nastavku koristimo *funkciju prijelaza* $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Funkcija prijelaza uređenom paru stanja $s \in \mathcal{S}$ i akcije $a \in \mathcal{A}$ pridružuje sljedeće stanje u koje je prelazak posljedica odabira akcije a u stanju s.

Neka je t neki trenutak. Definiramo *ishod* igre pridružen trenutku t kao nagradu u završnom trenutku igre T iz perspektive igrača koji odlučuje o akciji u trenutku t. Odnosno, ishod igre pridružen trenutku t je $z_t = \pm r(s_T)$, gdje je S_T stanje koje odgovara završnom trenutku T. Dakle, ako je igrač koji odlučuje o akciji u trenutku t igrač MAX, z_t jednak je $r(s_T)$. Inače vrijedi $z_t = -r(s_T)$.

Neka je $s \in \mathcal{S}$. Pomoću p_s označimo funkciju $p_s : \mathcal{A}(s) \to [0, 1]$ koja definira vjerojatnosnu distribuciju na skupu legalnih akcija $\mathcal{A}(s)$, odnosno označimo pomoću $p_s(a)$ vjerojatnost odabira akcije a u stanju s. Definirajmo sada *politiku* $p : \mathcal{S} \to \cup_{s\in\mathcal{S}} p_s$ tako da vrijedi

$$p(s) := p_s, \quad s \in \mathcal{S}.$$

Za neke $s \in \mathcal{S}$ i $a \in \mathcal{A}$, $p(s)(a)$ ćemo označavati pomoću $p(a|s)$. Ovako definiranu funkciju smatramo pojmom ekvivalentnim kontrolnoj strategiji kakvu smo spominjali ranije u radu. Iako je ideja iza obje funkcije određivanje vjerojatnosne distribucije na skupu akcija legalnih u nekome stanju, domena je kontrolne strategije o kakvoj smo govorili ranije u radu $\mathcal{A} \times \mathcal{S}$ (odnosno, u slučaju alternirajućih Markovljevih igara, $\mathcal{A}_1 \times \mathcal{S}_1$ ili $\mathcal{A}_2 \times \mathcal{S}_2$), dok je domena upravo definirane politike skup $\mathcal{S}$. Razlog zašto sada ističemo i politiku kao pojam zaseban od kontrolne strategije taj je da bismo se notacijom više približili onoj korištenoj u kasnijim odjeljcima rada, u kojima će biti riječ o strukturi AlphaZero metode, a posebno i neuronskoj mreži AlphaZero algoritma te njezinim izlazima za dane ulaze.

Za politiku p definiramo odgovarajuću *funkciju vrijednosti* kao funkciju koja stanju $s = S_t$ pridružuje očekivani ishod pridružen trenutku t ako i prvi i drugi igrač akcije biraju u skladu s politikom p, odnosno kao

$$v_p(s) = \mathbb{E}_p(z_t \,|\, S_t = s), \quad s \in \mathcal{S}.$$

Za igre sa sumom nula, optimalna funkcija vrijednosti v^* dana je jednadžbom

$$v^*(s) = \begin{cases} z_T, & \text{ako } s = S_T \\ \max\limits_a -v^*(\mathcal{P}(s, a)), & \text{inače} \end{cases}, \quad s \in \mathcal{S}. \tag{1.30}$$

Ona stanju $s = S_t$ pridružuje ishod pridružen trenutku t u slučaju da oba igrača slijede optimalnu politiku (onu koja ishod maksimira).

Metode strojnog učenja podrškom

Iteracija kontrolne strategije (engl. *policy iteration*) klasični je algoritam koji generira niz sve boljih i boljih kontrolnih strategija alterniranjem procjene kontrolne strategije (procjene funkcije vrijednosti stanja za trenutnu kontrolnu strategiju) i poboljšanja kontrolne strategije (korištenja trenutne funkcije vrijednosti za dobivanje bolje kontrolne strategije).

Naime, pretpostavimo da je poboljšanjem kontrolne strategije iz originalne determinističke kontrolne strategije π, pomoću funkcije vrijednosti stanja v_π, dobivena bolja kontrolna strategija π'. Tada možemo izračunati $v_{\pi'}$ i ponovno poboljšanjem kontrolne strategije doći do još bolje kontrolne strategije π''. Na taj način možemo dobiti niz monotono sve boljih i boljih kontrolnih strategija i funkcija vrijednosti stanja. Odnosno, možemo dobiti niz

$$\pi_0 \overset{\text{E}}{\rightarrow} v_{\pi_0} \overset{\text{I}}{\rightarrow} \pi_1 \overset{\text{E}}{\rightarrow} v_{\pi_1} \overset{\text{I}}{\rightarrow} \pi_2 \overset{\text{E}}{\rightarrow} \ldots \overset{\text{I}}{\rightarrow} \pi_* \overset{\text{E}}{\rightarrow} v_*,$$

gdje $\overset{\text{E}}{\rightarrow}$ označava procjenu kontrolne strategije (oznaka E dobivena je na temelju engleske riječi *evaluation* iz *policy evaluation*), a $\overset{\text{I}}{\rightarrow}$ označava poboljšanje kontrolne strategije (oznaka I dobivena je na temelju engleske riječi *improvement* iz *policy improvement*). Kao što smo pokazali ranije u radu, svaka tako dobivena kontrolna strategija strogo je bolja od prethodne (osim ako je prethodna već optimalna). Budući da konačni Markovljev proces odlučivanja ima samo konačan broj mogućih deterministčkih kontrolnih strategija, ovaj proces konvergira k optimalnoj kontrolnoj strategiji i optimalnoj funkciji vrijednosti stanja nakon konačnog broja iteracija.

Dok je jednostavan pristup poboljšanju kontrolne strategije pohlepan odabir akcija na temelju funkcije vrijednosti stanja, jednostavan je pristup procjeni kontrolne strategije procjena funkcije vrijednosti stanja na temelju uzoraka epizoda. U slučaju skupova stanja velikog kardinaliteta, aproksimacije su nužne za procjenu pojedine kontrolne strategije i prikaz njezinog poboljšanja.

Strojno učenje podrškom kombinira se s dubokim učenjem, klasom algoritama strojnog učenja koji koriste duboke neuronske mreže, koje imaju sposobnost progresivnog ekstrahiranja apstraktnijih značajki iz neobrađenih ulaza, radi stvaranja efikasnih algoritama koji su sposobni riješiti prethodno nerješive probleme. Neuronske se mreže, zbog svojih svojstava univerzalnih aproksimatora, koriste kao funkcijski aproksimatori kontrolnih strategija i funkcija vrijednosti. Ta se kombinacija naziva dubokim učenjem podrškom (engl. *deep reinforcement learning*) i upravo je ono na čemu počiva AlphaZero algoritam.

Poglavlje 2

Struktura AlphaZero algoritma

2.1 AlphaZero: ispunjenje dugogodišnjeg cilja umjetne inteligencije

Igra šaha najdulje je istraživana domena u povijesti umjetne inteligencije. Šah se dugo nazivao „drozofilom[1] umjetne inteligencije" budući da je bio standardna referentna točka za testiranje i uspoređivanje novih algoritama. Istraživanje računalnog šaha staro je kao i samo računarstvo. Naime, već su Charles Babbage, Alan Turing, Claude Shannon i John von Neumann osmišljali *hardware*, algoritme i teoriju za analizu te igranje šaha. Šah je potom postao veliki izazov za generaciju istraživača na području umjetne inteligencije, što je kulminiralo računalnim šahovskim programima koji su sposobni igrati šah na nadljudskoj razini. Međutim, ti su sustavi bili krajnje prilagođeni svojoj domeni i nisu se mogli generalizirati za druge igre bez značajnog ljudskog truda, dok su općeniti sustavi za igranje igara bili relativno slabi. Upravo je stvaranje programa koji su sposobni samostalno učiti na temelju osnovnih principa dugogodišnja ambicija umjetne inteligencije.

Značajan iskorak za umjetnu inteligenciju postignut je 1997. godine, kad je IBM-ov računalni program Deep Blue porazio ljudskog svjetskog prvaka u šahu Garryja Kasparova. Nakon toga, u naredna dva desetljeća, računalni šahovski programi nastavili su se postojano poboljšavati te postizati rezultate na razini iznad ljudske. Ti su programi procjenjivali stanja na temelju ručno izrađenih značajki, konstruiranih od strane jakih ljudskih igrača i programera. To su činili u kombinaciji s alfa-beta pretraživanjem dobrih performansi, koje je proširivalo prostrano stablo pretraživanja korištenjem velikog broja dovitljivih heurističkih funkcija i prilagodbi specifičnih za domenu.

Iako su se računalni programi za igranje šaha razvijali i nakon uspjeha programa Deep

[1] Drozofila, odnosno *drosophila melanogaster*, koristi se kao model organizma u genetici. Naime, radi se o jednom od najistraživanijih organizama u biologiji, osobito u genetici i razvojnoj biologiji.

Blue, on je ipak pridonio preusmjeravanju pažnje s kompjuterskog šaha ka kompjuterskoj igri Go. Igra Go kroz dugi je niz godina bila smatrana najizazovnijom klasičnom igrom za umjetnu inteligenciju te standardom postignuća umjetne inteligencije u području igara. Naime, radi se o tradicionalnoj igri na ploči za dva igrača koja se igra na presjecima pravokutne mreže, obično dimenzija 19×19. Igrači naizmjence postavljaju svoje figure (crne za jednog i bijele za drugog igrača) na ploču; susjedne figure formiraju takozvane *grupe*, koje se smatraju zarobljenima ako nemaju *sloboda* (susjednih praznih mjesta). Igra završava kada nijedan od igrača više ne može povući potez, a pobjednik je onaj igrač koji kontrolira više teritorija na ploči.

Zbog velikog faktora grananja i poprilične dubine stabla pretraživanja te nedostatka heurističkih funkcija za pouzdanu procjenu vrijednosti nezavršnih stanja, Go je bila smatrana teškom igrom za računala — naime, ljudski su igrači godinama bili značajno bolji u igri Go od računalnih programa. Pojam *složenosti* u kontekstu igara koristi se kako bi označio dvije različite mjere, koje nazivamo složenošću prostora stanja (engl. *state-space complexity*) i složenošću stabla igre (engl. *game-tree complexity*). *Složenost prostora stanja* igre definira se kao broj stanja dostižnih iz početnog stanja igre. Egzaktno računanje složenosti prostora stanja igara poput šaha nije lako ostvarivo pa se uglavnom koriste metode za računanje aproksimacija.

U svrhu definiranja složenosti stabla igre, prisjetimo se odjeljka 1.2 — naime, u njemu smo spomenuli stablo pretraživanja, odnosno naveli smo da pretraživanjem usmjerenog grafa u sklopu pretraživanja prostora stanja postepeno gradimo stablo pretraživanja. Stablo pretraživanja nastaje proširivanjem pojedinih čvorova, odnosno generiranjem njihovih sljedbenika pomoću funkcije sljedbenika. Pritom čvorove koji su generirani, ali još nisu proširenи nazivamo *otvorenim čvorovima*, dok čvorove koji su već proširenи nazivamo *zatvorenim čvorovima*.

Definicija 2.1.1. Stablo pretraživanja pune širine *stablo je pretraživanja dubine d takvo da su za svaki* $i = 0, 1, \ldots, d-1$ *svi čvorovi na razini i zatvoreni čvorovi.*

Definicija 2.1.2. Dubina rješenja *čvora n jednaka je minimalnoj dubini stabla pretraživanja pune širine s korijenom n dovoljnoj za određivanje teoretske vrijednosti igre, pri čemu* teoretskom vrijednošću igre *nazivamo ishod igre u slučaju optimalnog (u smislu maksimiranja vlastite nagrade) igranja svih igrača. Takvo stablo pretraživanja minimalne dubine nazivamo* stablom pretraživanja rješenja *čvora n.*

Primjerice, promotrimo šahovsku poziciju u kojoj bijeli vuče potez i njoj pridružen čvor n. Bijeli može povući bilo koji od 30 poteza. Radi jednostavnosti, pretpostavimo da nakon svakog od mogućih 30 poteza bijelog, crni na raspolaganju ima 20 mogućih poteza, od kojih barem jedan matira bijelog. Tada se stablo pretraživanja rješenja čvora n sastoji od korijena n, tridesetero njegove djece te 600 njegove unučadi.

Definicija 2.1.3. Složenost stabla igre *neke igre broj je listova u stablu pretraživanja rješenja čvora koji odgovara početnom stanju dane igre.*

Dakle, da čvor n iz prethodnog primjera odgovara početnom stanju igre, odgovarajuća bi složenost stabla igre bila jednaka 600.

Mogućih je konfiguracija igraće ploče za Go $3^{361} \approx 10^{172}$, dok je, primjerice, složenost prostora stanja šaha 10^{50} — Go je, što se tiče složenosti prostora stanja, otprilike gugol (10^{100}) puta složenija igra od šaha. Victor Allis u publikaciji [4] naveo je da se tipična Go igra igrana od strane profesionalaca sastoji od otprilike 150 povučenih poteza i da joj je prosječni faktor grananja jednak 250, što daje procjenu složenosti stabla igre jednaku 10^{360}.

Kompjuterski programi za Go koji su koristili alfa-beta pretraživanje dostigli su razinu boljih početnika oko 1997. godine, ali su nakon toga stagnirali do 2006. godine. Metoda koja je inicijalno imala značajan utjecaj na smanjivanje jaza između ljudskih i računalnih igrača igre Go bila je pretraživanje stabla Monte Carlo metodom (engl. *Monte Carlo Tree Search*, MCTS), koja je upravo jedna od glavnih komponenata AlphaZero algoritma (i koja će biti detaljno opisana u odjeljku 2.4). Međutim, dok je MCTS metoda omogućila da računala budu konkurentna najboljim ljudskim igračima na malenim igraćim pločama, računalni programi za Go bazirani na MCTS-u nisu uspjeli postići njihovu razinu igre na standarnoj 19×19 ploči.

Početkom 2016. godine tvrtka DeepMind objavila je rad [20], u kojem je predstavila novi pristup računalnoj igri Go; kreirali su računalni program AlphaGo, koji se temeljio na strojnom učenju podrškom (engl. *reinforcement learning*) te je kombinirao napredno pretraživanje stabla s dubokim neuronskim mrežama. Neuronske mreže koje je upotrebljavao AlphaGo kao ulaz su primale opis igraće ploče za Go te su ga procesuirale pomoću više različitih slojeva s milijunima sinapsama sličnim veza. Jedna je neuronska mreža, takozvana mreža politike (engl. *policy network*) odabirala sljedeći potez u igri, dok je druga neuronska mreža, nazvana mrežom vrijednosti (engl. *value network*), predviđala pobjednika igre. Te su neuronske mreže bile trenirane novom metodom koja se može opisati kao kombinacija nadziranog učenja s podacima iz igara ljudskih stručnjaka kao podacima za treniranje te strojnog učenja podrškom na temelju igranja igara algoritma protiv samoga sebe (engl. *self-play*). Autori su AlphaGo upoznali s velikim brojem ljudskih igara kako bi mu omogućili da stekne razumijevanje razumne ljudske igre. Potom je AlphaGo više tisuća puta igrao protiv različitih verzija samoga sebe, u svakoj pojedinoj igri učeći iz svojih pogrešaka po principu strojnog učenja podrškom. Bez ikakvog pretraživanja unaprijed, neuronske su mreže igrale Go na razini *state-of-the-art* MCTS programa koji su simulirali tisuće nasumičnih *self-play* igara. Algoritam pretrage koji je AlphaGo koristio bio je nova kombinacija Monte Carlo simulacija i *value* te *policy* neuronskih mreža. Tijekom vremena, AlphaGo se poboljšavao i postepeno postajao sve bolji u učenju i donošenju odluka. Uz pomoć opisanih metoda, AlphaGo uspio je pobijediti u 99.8% igara protiv drugih pos-

tojećih Go programa te je pobijedio europskog prvaka u igri Go s rezultatom 5 naprama 0. To je bio prvi put da je računalni program pobijedio ljudskog profesionalnog igrača u igri Go s igraćom pločom standardne veličine, što je bio pothvat za koji se prethodno smatralo da će biti neostvariv još barem jedno desetljeće. AlphaGo nedvojbeno je postao najbolji igrač igre Go koji je ikada postojao.

AlphaGo bio je trijumf svojih tvoraca, no i dalje nije bio sasvim zadovoljavajuć — naime, uvelike je ovisio o ljudskom znanju o igri Go, odnosno učio je koje poteze treba povući djelomično pokušavajući oponašati ljudske prvake u igri te je koristio skup ručno izrađenih heurističkih funkcija kako bi izbjegao najgore pogreške tijekom pretraživanja unaprijed. Istraživači zaslužni za AlphaGo, svjesni njegovih upravo navedenih nedostataka, te s dugogodišnjim ciljem umjetne inteligencije na umu — kreacijom algoritma koji *tabula rasa* uči nadljudsku vještinu u zahtjevnim domenama, upustili su se u pothvat kreiranja nove verzije umjetne inteligencije koja bi imala sposobnost učiti samostalno, *tabula rasa*.

Rezultat tog pothvata bio je algoritam AlphaGo Zero, detaljno opisan u radu [24], objavljenom u listopadu 2017. Algoritam je naziv dobio na temelju činjenice da nije imao pristup nikakvom znanju o igri Go osim pravila igre. Naime, za razliku od AlphaGo algoritma, koji je podrazumijevao treniranje neuronskih mreža nadziranim učenjem na temelju poteza ljudskih stručnjaka i strojnim učenjem podrškom pomoću igranja protiv samoga sebe, AlphaGo Zero bio je zasnovan isključivo na strojnom učenju podrškom. Počevši *tabula rasa* i učeći uz samoga sebe kao učitelja; uz treniranje neuronske mreže da predviđa vlastite odabire poteza te pobjednike vlastitih igara, program AlphaGo Zero postigao je nadljudski učinak, pobjedivši prethodnog šampiona AlphaGo s rezultatom 100 naprama 0.

No, iako je postigao vrhunsku razinu igranja igre Go, izvanrednost postignuća programa AlphaGo Zero nije više bila osobito vezana uz sam Go. Dva mjeseca nakon njegovog objavljivanja, DeepMind objavio je *preprint* [21], kojim je pokazao da se AlphaGo Zero algoritam može generalizirati za bilo koju igru s dva igrača, sa sumom nula i s potpunim informacijama. DeepMind je izbacio „Go" iz naziva i nadjenuo novom sustavu naziv AlphaZero. On je, kao treća iteracija sustava, također bio i nesumnjivo najjednostavnija.

AlphaZero ustvari je općenitija verzija AlphaGo Zero algoritma — verzija koja je prilagođena široj klasi pravila igre. Od originalnog AlphaGo Zero algoritma razlikuje se u nekoliko aspekata. Oni su sljedeći:

- AlphaGo Zero procjenjivao je i optimirao vjerojatnost pobjeđivanja, iskorištavajući pritom činjenicu da Go igre imaju binarni ishod pobjede ili gubitka. Međutim, i šah i shogi mogu završiti neriješeno. Stoga AlphaZero procjenjuje i optimira očekivani ishod;

- Pravila igre Go invarijantna su na rotaciju i osnu simetriju, što je bilo iskorišteno u AlphaGo i AlphaGo Zero algoritmima na dva načina. Prvo, podaci za treniranje

bili su prošireni generiranjem osam simetrija za svako stanje igre. Drugo, tijekom pretraživanja stabla Monte Carlo metodom, konfiguracije igraće ploče bile su transformirane nasumičnim rotacijama i osnim simetrijama prije evaluiranja pomoću neuronske mreže. Kako bi bio prilagođen široj klasi igara, AlphaZero ne pretpostavlja nikakve simetrije. Naime, pravila šaha i shogija asimetrična su;

- Igranje igara algoritma protiv samoga sebe u AlphaGo Zero obavlja najuspješnija verzija od svih generiranih u dotadašnjim iteracijama. Nakon svake iteracije treniranja, učinak novog agenta mjeri se igranjem protiv najboljeg; ako novi agent pobijedi u barem 55% igara, proglašava se najboljim. Nasuprot tome, AlphaZero jednostavno održava jedinstvenu neuronsku mrežu koju ažurira kontinuirano, umjesto na kraju pojedine iteracije. Igranje algoritma samog protiv sebe obavlja se uvijek pomoću najnovijih parametara jedinstvene neuronske mreže.

AlphaZero bio je primijenjen na šah, shogi i igru Go na način da su u sva tri slučaja bili upotrebljavani isti algoritam i arhitektura neuronske mreže. AlphaZero zamijenio je „ručno" prikupljeno znanje i poboljšanja specifična za pojedinu domenu, koja su bila korištena u tradicionalnim računalnim programima za igranje igara, dubokim neuronskim mrežama, općenitim algoritmom strojnog učenja podrškom te općenitim algoritmom pretraživanja stabla. Naime, umjesto ručno definirane funkcije za evaluaciju i heurističkih funkcija za ocjenjivanje poteza, AlphaZero koristi duboku neuronsku mrežu koja na temelju stanja igre kao ulaza vraća vektor vjerojatnosti pojedinih poteza te skalarnu vrijednost koja procjenjuje očekivani ishod igre za danu poziciju. AlphaZero navedene vjerojatnosti i procjenu vrijednosti uči isključivo na temelju igranja protiv samoga sebe. Umjesto alfa-beta pretraživanja s poboljšanjima specifičnima za danu domenu, AlphaZero koristi općeniti algoritam pretraživanja stabla Monte Carlo metodom. Rezultati primjene takvog algoritma, predstavljeni u radu [22], pokazuju da općeniti algoritam strojnog učenja podrškom može *tabula rasa*, bez ljudskog znanja ili podataka specifičnih za domenu, postići nadljudski učinak u mnogim izazovnim igrama — to jest, pokazuju da je upravo AlphaZero ostvarenje dugogodišnjeg cilja umjetne inteligencije.

2.2 AlphaZero na intuitivnoj razini

U ovom ćemo poglavlju podrobno objasniti strukturu AlphaZero algoritma, odnosno metode koja je omogućila postizanje izvanrednih rezultata i ispunjenje ciljeva diskutiranih u odjeljku 2.1. Cijeli se AlphaZero sustav može razložiti na nekoliko dijelova, od kojih ćemo svaki detaljno obrazložiti u zasebnom odjeljku. Za početak, predstavimo AlphaZero algoritam na intuitivnoj razini; navedimo jednostavnu formulu za učenje koja je sadržana u njegovoj srži. Naime, formula se sastoji od sljedećih koraka:

- Mentalno prođi kroz moguće scenarije uzimajući u obzir kako bi protivnik mogao reagirati na tvoje poteze, pritom dajući prioritet putevima koji su više obećavajući, no ne zanemarujući i istraživanje nepoznatog;

- Kada naiđeš na nepoznato stanje, procijeni koliko bi moglo biti isplativo i taj iznos proslijedi unatrag prethodnim stanjima na mentalnom putu koji je vodio do trenutnog stanja;

- Po završetku analize budućih mogućnosti, povuci najistraženiji potez;

- Na kraju igre, vrati se unatrag i ocijeni vrijednosti kojih su stanja bile krivo procijenjene te u skladu s tim ažuriraj svoje znanje.

Navedena formula vrlo nam vjerojatno zvuči poznato jer nas podsjeća na način na koji i sami učimo igrati igre. Kada se prilikom igranja odlučimo za potez za koji se ispostavi da je loš, to učinimo jer smo krivo procijenili vrijednosti rezultirajućih stanja ili jer smo krivo procijenili vjerojatnost da će protivnik odigrati određeni potez pa nismo dovoljno istražili tu mogućnost. Točno su to dva aspekta igranja igara koje AlphaZero uči. Izložimo detaljno u nastavku metodologiju koja mu omogućava da ih uspješno nauči.

2.3 Osnove: kako funkcionira cjelina i koje su njene komponente

AlphaZero algoritam možemo kategorizirati kao algoritam strojnog učenja podrškom, temeljen na igranju protiv samog sebe. Možemo ga shvatiti kao shemu aproksimativne iteracije kontrolne strategije u kojoj se pretraživanje stabla Monte Carlo metodom koristi za poboljšanje kontrolne strategije. Taj se pristup najdirektnije primjenjuje na igre s dva igrača, sa sumom nula i s potpunim informacijama, koje formaliziramo kao alternirajuće Markovljeve igre, opisane u pododjeljku *Alternirajuće Markovljeve igre* odjeljka 1.3.

Postupak poboljšanja kontrolne strategije kreće od kontrolne strategije čije vrijednosti za pojedinu akciju i stanje daje neuronska mreža, sastoji se od pretraživanja stabla Monte Carlo metodom temeljenom na preporukama takve kontrolne strategije te potom projiciranja nove kontrolne strategije proizašle iz pretraživanja u prostor funkcija čiji je aproksimator neuronska mreža. Postupak procjene kontrolne strategije vrši se nad kontrolnom strategijom proizašlom iz pretraživanja na način da se funkcija koja pojedinim stanjima pridružuje ishode igara također projicira u prostor funkcija čiji je aproksimator neuronska mreža. Te se projekcije ostvaruju treniranjem neuronske mreže kako bi njezini izlazi odgovarali vjerojatnosnim distribucijama dobivenim pretraživanjem i ishodima igara koje je algoritam odigrao protiv samoga sebe.

Domensko znanje

Kao što je već bilo navedeno u odjeljku 2.1, AlphaZero nadljudski učinak u mnogim izazovnim domenama postiže *tabula rasa*, bez ljudskog domenskog znanja, samo uz poznavanje pravila igre. Objasnimo detaljnije koja to domenska znanja AlphaZero ipak koristi, bilo eksplicitno bilo implicitno, bilo tijekom treniranja bilo tijekom pretraživanja stabla Monte Carlo metodom. Naime, to su ona znanja koja se mijenjaju ovisno o konkretnoj (alternirajućoj Markovljevoj) igri koju algoritam uči igrati — to jest, ta znanja variraju ovisno o pojedinom algoritmu iz klase AlphaZero algoritama. Kada govorimo o AlphaZero algoritmu, ustvari govorimo o zajedničkim karakteristikama svih algoritama iz njegove klase (neovisno o konkretnoj igri koju pojedini uči igrati).

Primarno, AlphaZero ima savršeno znanje o pravilima igre koju uči igrati. Poznato je početno stanje igre, a pomoću pravila igre određuju se i stanja igre u koja su prelasci posljedica pojedinih akcija povučenih tijekom pretraživanja stabla Monte Carlo metodom i igranja algoritma protiv samoga sebe. Agent za svako stanje zna koje su akcije u njemu legalne. Pravila igre potrebna su i za detekciju završnih stanja te njihovo vrednovanje.

AlphaZero upoznat je sa strukturom ploče na kojoj se igra pojedina igra na ploči koju uči igrati. Na temelju te strukture definiraju se dimenzije ulaznih podataka neuronske mreže. Također, AlphaZero raspolaže znanjem o skupu svih mogućih akcija tijekom igre, na temelju kojeg se definira reprezentacija izlaza neuronske mreže — točnije, vjerojatnosti povlačenja pojedinih poteza u nekome stanju, koje definiraju kontrolnu strategiju. Drugim riječima, arhitektura neuronske mreže usklađena je sa strukturom igraće ploče. Reprezentacija ulaza i izlaza neuronske mreže detaljnije će biti razložena u odjeljku 2.6.

Navedeni oblici domenskog znanja jedini su koje AlphaZero koristi. Dok su mnogi pristupi pretraživanju stabla Monte Carlo metodom koji su prethodili AlphaZero algoritmu podrazumijevali raznovrsne modifikacije i poboljšanja ovisna o domeni, pretraživanje stabla Monte Carlo metodom u algoritmu AlphaZero ne koristi nikakve heuristike niti pravila specifična za domenu. Također, niti jedna se legalna akcija ne isključuje iz razmatranja, što je u mnogim programima prije AlphaZero algoritma bio standardan postupak.

Pregled komponenti

U AlphaZero metodi možemo prepoznati tri glavne komponente. Navedimo ih i ukratko opišimo njihovu ulogu.

- Prva je komponenta pretraživanje stabla Monte Carlo metodom. Kao što smo već naveli ranije u odjeljku 2.3, na pretraživanju stabla Monte Carlo metodom počiva poboljšanje kontrolne strategije. Osnovna je ideja pretraživanja stabla Monte Carlo metodom djelomično nasumičan prolazak kroz stanja igre i proširivanje čvorova u stablu pretraživanja koji ih predstavljaju uz prikupljanje podataka o frekvenciji

povlačenja pojedinih akcija i ishodima igre. Budući da je stablo igre često goleme širine i dubine, ne nastojimo pretraživanjem stabla Monte Carlo metodom izgraditi cijelo stablo, već samo nasumično uzorkujemo puteve koji nam djeluju najviše obećavajuće (po čemu je metoda ustvari i dobila ime). Pretraživanje stabla Monte Carlo metodom detaljno će biti objašnjeno u odjeljku 2.4.

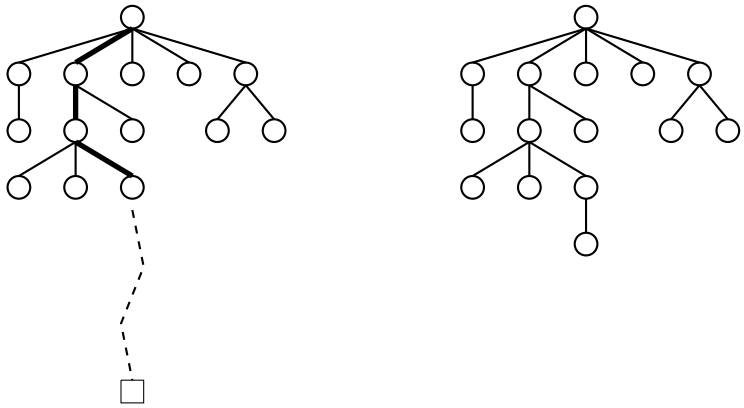
- Za drugu komponentu možemo reći da je upravo igranje igara algoritma protiv samog sebe. AlphaZero održava jedinstvenu neuronsku mrežu, koja se koristi za generiranje igara koje algoritam igra sam protiv sebe. Taj model inicijalizira se nasumičnim parametrima pa program na početku učenja poteze povlači nasumično. Međutim, s vremenom se parametri modela ažuriraju pa njegove poboljšane verzije generiraju sve smislenije i sofisticiranije scenarije igranja. No, što uopće znači igranje algoritma samog protiv sebe? Ono se odnosi na korištenje istog modela za oba igrača jedne igre. Možemo se pitati zašto nam to odgovora, s obzirom na to da je vjerojatnost da model pobijedi u igri protiv samoga sebe otprilike $\frac{1}{2}$. Međutim, to je upravo ono što nam je potrebno — naime, u takvim igrama model ima priliku pokazati svoje najbolje vještine. Na isti princip nailazimo i u svakodnevnom životu: zanimljiviji su nam mečevi između igrača otprilike jednakih vještina, nego oni između početnika i stručnjaka. Ova će komponenta biti detaljnije predstavljena u odjeljku 2.5.
- Treća je komponenta proces treniranja neuronske mreže na temelju podataka prikupljenih tijekom igranja igara posljednje verzije modela samog protiv sebe. Točnije, radi se o treniranju modela na podacima koje čine kontrolna strategija proizašla iz pretraživanja te ishodi igara za pojedina stanja. Taj će proces biti podrobno izložen u odjeljku 2.6.

2.4 Pretraživanje stabla Monte Carlo metodom

Pretraživanje stabla Monte Carlo metodom (engl. *Monte Carlo Tree Search*, MCTS) pristup je pretraživanju stabala i usmjerenih grafova u svrhu pronalaženja optimalnih odluka koji se temelji na uzimanju nasumičnih uzoraka u prostoru mogućih odluka. Može se koristiti uz malo ili nimalo domenskog znanja. MCTS je i kao samostalna metoda, u svom osnovnom ili rafiniranom i proširenom obliku, imao značajan utjecaj na pristupe umjetne inteligencije rješavanju problema iz domena koje se mogu predstaviti kao stabla sekvencijalnih odluka, među kojima su najistaknutije igre te problemi planiranja. Naime, primjena MCTS-a na problem računalne igre Go polučila je uspjehe i prije nego što je njegova kombinacija s mrežama politike i vrijednosti u algoritmu AlphaGo omogućila pobjedu nad svim prethodnim Go programima, kao i ljudskim profesionalnim igračima. Još od 2006. godine, svi najbolji algoritmi za igranje igre Go koriste pretraživanje stabla Monte Carlo metodom. MCTS je takozvani *statistički bilo kad* algoritam (engl. *anytime algorithm*), što znači da

vraća valjano rješenje problema bez obzira na trenutak prekida izvršavanja te da sve bolja rješenja pronalazi što se dulje izvršava. Također, u slučaju MCTS-a, veća računalna snaga implicira bolji učinak.

Slika 2.1: Osnovni proces u pretraživanju stabla Monte Carlo metodom.



Osnovni proces u pretraživanju stabla Monte Carlo metodom ilustriran je slikom 2.1, koja je napravljena po uzoru na sliku iz članka [7]. On obuhvaća inkrementalnu i asimetričnu izgradnju stabla (tipično stabla igre), koju usmjerava takozvana *politika stabla* (engl. *tree policy*). Politika stabla u svakoj iteraciji algoritma pronalazi „najhitniji" čvor trenutnog stabla, pritom nastojeći pronaći kompromis između istraživanja i iskorištavanja, što je problem koji smo predstavili u odjeljku 1.3. Nakon što politika stabla odabere čvor, iz njega se obavlja *simulacija* (sasvim nasumičan ili nekom drukčijom politikom usmjeren niz akcija koje se povlače počevši iz danog stanja dok se ne dosegne završno stanje) te se stablo pretraživanja ažurira na osnovu rezultata. To uključuje dodavanje čvora djeteta koji odgovara stanju u koje vodi akcija povučena u stanju koje predstavlja čvor odabran od strane politike stabla i ažuriranje njegovih statističkih podataka te onih njegovih predaka. Potezi tijekom simulacije povlače se na temelju *zadane politike* (engl. *default policy*), koja u najjednostavnijem slučaju akcije odabire nasumično. Velika je prednost pretraživanja stabla Monte Carlo metodom u odnosu na, primjerice, minimaks pretraživanje ograničene dubine, to što se vrijednosti nezavršnih stanja ne moraju procjenjivati unaprijed definiranim funkcijama, što uvelike umanjuje količinu potrebnog domenskog znanja (kao što smo već naveli ranije u odjeljku, MCTS se može koristiti uz malo ili nimalo domenskog znanja). Jedino je nužno znati vrijednost završnog stanja na kraju pojedine simulacije.

Prije no što detaljnije razradimo algoritam pretraživanja stabla Monte Carlo metodom, predstavimo prvo pozadinsku teoriju koja je dovela do razvoja MCTS metoda, odnosno Monte Carlo metode i bandit-bazirane metode.

Monte Carlo metode

Monte Carlo algoritmi čine jednu od dvije kategorije algoritama sa slučajnošću kao dijelom logike (uz Las Vegas algoritme). Dok Las Vegas algoritmi uvijek vraćaju točan odgovor (ili izvještavaju o grešci) i zauzimaju nasumičan dio resursa, Monte Carlo algoritmi vraćaju odgovore s greškom nasumične veličine, a koja se može smanjiti upotrebom više resursa, obično memorije, i duljim vremenom izvršavanja. Uz bilo koje fiksirano ograničenje na resurse, Monte Carlo algoritam može vratiti približno rješenje (u skladu s time, i pretraživanje stabla Monte Carlo metodom je bilo kad algoritam, kao što smo pojasnili ranije u odjeljku 2.4).

Monte Carlo metode korijene imaju u statističkoj fizici, gdje su se koristile za dobivanje aproksimacija nerješivih integrala, a danas se koriste u širokom spektru domena, uključujući i istraživanje igara. Dvije su klase statističkih problema na koje se najčešće primjenjuju Monte Carlo metode integriranje i optimizacija. Monte Carlo metode općenito su korisne u situacijama kada analitička ili numerička rješenja ne postoje ili ih je preteško izračunati. Mnoge se važne tehnologije koje ostvaruju ciljeve strojnog učenja temelje na uzimanju uzoraka iz neke vjerojatnosne distribucije i korištenju tih uzoraka za formiranje Monte Carlo procjene neke kvantitete.

Monte Carlo metode klasa su tehnika za nasumično uzorkovanje vjerojatnosne distribucije. Tri su glavna razloga zašto bismo htjeli uzimati uzorke iz vjerojatnosne distribucije sljedeća:

- *Procjenjivanje gustoće*: prikupljamo uzorke radi aproksimiranja distribucije dane varijable;
- *Aproksimiranje mjera*: želimo aproksimirati očekivanje ili varijancu neke distribucije;
- *Optimizacija funkcije*: nastojimo na temelju uzorkovanja odrediti koji argument (odnosno, koji argumenti) maksimira ili minimizira ciljnu funkciju.

Upoznajmo se sada s osnovama Monte Carlo uzorkovanja. Kada, primjerice, sumu ili integral ne možemo izračunati egzaktno, obično je moguće napraviti procjenu pomoću Monte Carlo uzorkovanja. Ideja je promatrati sumu ili integral kao očekivanje neke funkcije neke slučajne varijable i procijeniti to očekivanje odgovarajućim prosjekom. Neka je

$$s = \sum_x p(x) f(x) = \mathbb{E}_p \left(f(X) \right),$$

odnosno

$$s = \int p(x) f(x) dx = \mathbb{E}_p \left(f(X) \right),$$

suma, odnosno integral koji treba procijeniti, zapisan kao očekivanje, pri čemu je p vjerojatnosna distribucija (u slučaju sume), odnosno funkcija gustoće (u slučaju integrala) slučajne varijable X.

Bilo sumu bilo integral s možemo procijeniti pomoću statistike

$$\hat{s}_n = \frac{1}{n}\sum_{i=1}^{n} f(X_i),$$

gdje je $X_1, \ldots, X_n$ slučajan uzorak duljine n za X. Takvu procjenu opravdava nekoliko činjenica. Prvo, $\hat{s}_n$ je nepristran procjenitelj za s, što možemo vidjeti iz

$$\mathbb{E}(\hat{s}_n) = \mathbb{E}\left(\frac{1}{n}\sum_{i=1}^{n} f(X_i)\right) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}\left(f(X_i)\right) = \frac{1}{n}\sum_{i=1}^{n} s = s.$$

Također, uz pretpostavku da je $\mathrm{Var}\left(f(X)\right) < \infty$, $\hat{s}_n$ je (slabo) konzistentan procjenitelj za s, odnosno vrijedi

$$\lim_{n\to\infty} \hat{s}_n = s.$$

To slijedi iz teorema 2.4.1, preuzetog iz [10], čiji se dokaz može pronaći u [9].

Teorem 2.4.1. *Neka je $X_1, X_2, \ldots, X_n$ slučajni uzorak iz populacije s konačnom varijancom i neka je μ parametar očekivanja. Tada je aritmetička sredina $\overline{X}_n$:*

1. *nepristran procjenitelj za μ;*
2. *konzistentan procjenitelj za μ.*

Promotrimo varijancu statistike $\hat{s}_n$ u ovisnosti o n, uz danu pretpostavku o konačnosti $\mathrm{Var}\left(f(X)\right)$. Naime, za $\mathrm{Var}\left(\hat{s}_n\right)$ vrijedi

$$\begin{aligned}\mathrm{Var}\left(\hat{s}_n\right) &= \frac{1}{n^2}\sum_{i=1}^{n} \mathrm{Var}\left(f(X_i)\right) \\ &= \frac{\mathrm{Var}\left(f(X)\right)}{n}.\end{aligned} \tag{2.1}$$

Iz jednadžbe (2.1) možemo vidjeti da vrijedi $\lim_{n\to\infty} \mathrm{Var}\left(\hat{s}_n\right) = 0$. Taj nam rezultat govori kako procijeniti nesigurnost Monte Carlo prosjeka, odnosno veličinu greške Monte Carlo aproksimacije. Kako bismo dobili procjenitelj za $\mathrm{Var}(\hat{s}_n)$, računamo aritmetičku sredinu uzorka $f(X_1), \ldots, f(X_n)$, potom pripadnu uzoračku varijancu i na kraju statistiku dijelom brojem n. Centralni granični teorem (teorem 2.4.2, preuzet iz [11]), uz pretpostavku o konačnosti s i $\mathrm{Var}\left(f(X)\right)$, kaže nam da niz $\hat{s}_n$ konvergira po distribuciji normalnoj razdiobi s očekivanjem s i varijancom $\frac{\mathrm{Var}(f(x))}{n}$. Ta nam činjenica omogućava da pronađemo pouzdane intervale za s korištenjem funkcije distribucije normalne razdiobe.

Teorem 2.4.2 (Centralni granični teorem)**.** *Neka je $X_1, X_2, \ldots$ niz nezavisnih jednako distribuiranih slučajnih varijabli s konačnim matematičkim očekivanjem μ i konačnom varijancom $\sigma^2 > 0$. Nadalje, neka je $\overline{X}_n := \frac{X_1+X_2+\ldots+X_n}{n}$ za sve prirodne brojeve n. Tada za sve $a < b$ vrijedi*

$$\lim_{n\to\infty} \mathbb{P}\left(a \leq \frac{\overline{X}_n - \mu}{\sigma}\sqrt{n} \leq b\right) = \Phi(b) - \Phi(a),$$

gdje je $\Phi(x)$ funkcija distribucije jedinične normalne razdiobe.

Prisjetimo se funkcije vrijednosti akcije i procjene kontrolne strategije iz odjeljka 1.3. Za danu kontrolnu strategiju π, stanje s i akciju a, kako bismo mogli aproksimirati vrijednost funkcije vrijednosti akcije na uređenom paru (s, a)? Ta se aproksimacija može napraviti pomoću Monte Carlo metoda, uz pretpostavku da se radi o epizodnom zadatku. Traženu vrijednost nazivamo *Q-vrijednošću* akcije a i označavamo je pomoću $Q(s, a)$. Uz takvu notaciju, tražena se vrijednost može izraziti na način predstavljen u jednadžbi

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} \mathbb{1}_i(s, a) z_i. \tag{2.2}$$

Pritom je $N(s, a)$ u jednadžbi 2.2 broj odabira akcije a u stanju s tijekom Monte Carlo uzorkovanja, $N(s)$ predstavlja broj posjeta stanju s, z_i označava povrat za i-tu simulaciju iz stanja s, a $\mathbb{1}_i(s, a)$ jednako je 1 ako je akcija a bila odabrana u stanju s u i-toj simulaciji iz stanja s, a 0 inače. Drugim riječima, $Q(s, a)$ je prosječni povrat svih simulacija u kojima je u stanju s bila odabrana akcija a.

Bandit-bazirane metode

Klasa *bandit problema* poznata je klasa problema sekvencijalnog donošenja odluka, u kojima se odlučivanje svodi na odabir jedne od K akcija (primjerice, K ručica višeručnog bandit automata za kockanje), od kojih je uz svaku vezana nagrada. Odabir akcije otežan je nepoznavanjem distribucije nagrada — potencijalne je nagrade potrebno procijeniti na temelju prethodnih opažanja. Cilj je maksimirati kumulativnu nagradu otkrivanjem optimalne akcije i njenim što češćim povlačenjem u rastućem broju poteza. To vodi do već predstavljenog problema kompromisa između istraživanja i iskorištavanja — potrebno je balansirati iskorištavanje akcije za koju se trenutno vjeruje da je optimalna i istraživanje drugih akcija koje se trenutno čine suboptimalnima, no za koje postoji mogućnost da se dugoročno pokažu superiornima.

K-ruki bandit definira se pomoću slučajnih varijabli $X_{i,n}$, $1 \leq i \leq K$ i $n \geq 1$, gdje i označava ruku bandita. Odnosno, $X_{i,n}$ predstavlja nagradu dobivenu igranjem i-te ruke bandita po n-ti put. Dakle, uzastopno igranje i-te ruke daje niz $X_{i,1}, X_{i,2}, \ldots$ nezavisnih jednako distribuiranih slučajnih varijabli nepoznate distribucije, s nepoznatim očekivanjem

μ_i. Problemu K-rukog bandita može se pristupiti korištenjem kontrolne strategije koja na temelju prethodnih nagrada određuje koju ruku treba igrati.

Kontrolna bi strategija trebala maksimirati očekivane nagrade prilikom uzastopnog igranja bandita. Ekvivalentno, trebala bi minimizirati igračevo *žaljenje* (engl. *regret*), koje se nakon n igranja definira kao u jednadžbi

$$R_n := \mu^* n - \sum_{j=1}^{K} \mu_j \mathbb{E}\left(T_j(n)\right). \tag{2.3}$$

Pritom μ^* u jednadžbi (2.3) označava najbolju moguću nagradu ($\mu^* = \max_j \mu_j$), a $\mathbb{E}\left(T_j(n)\right)$ predstavlja očekivani broj igranja ruke j u prvih n pokušaja. Drugim riječima, žaljenje je očekivani gubitak zbog propuštanja igranja najbolje ruke. Bitno je naglasiti nužnost dodjeljivanja pozitivnih vjerojatnosti svih rukama u svakom trenutku kako bi se osiguralo da se optimalna ruka ne propusti zbog trenutno obećavajućih nagrada za suboptimalnu ruku.

Zapišimo žaljenje na malo drukčiji način. Naime, ako definiramo $\Delta_j := \mu^* - \mu_j$, korištenjem jednakosti $n = \sum_{j=1}^{K} T_j(n)$ dobivamo

$$\begin{aligned} R_n &= \mu^* n - \sum_{j=1}^{K} \mu_j \mathbb{E}\left(T_j(n)\right) = \mathbb{E}\left(\mu^* n - \sum_{j=1}^{K} \mu_j T_j(n)\right) \\ &= \mathbb{E}\left(\sum_{j=1}^{K} (\mu^* - \mu_j) T_j(n)\right) = \mathbb{E}\left(\sum_{j=1}^{K} \Delta_j T_j(n)\right) \\ &= \sum_{j:\mu_j<\mu^*} \Delta_j \mathbb{E}\left(T_j(n)\right). \end{aligned}$$

U radu [15] Lai i Robbins opisali su kontrolne strategije koje su, uz pretpostavku da su distribucije nagrada u odgovarajućoj klasi, za svaku suboptimalnu ruku j garantirale da vrijedi

$$\mathbb{E}\left(T_j(n)\right) \leq c_j(n) \ln n, \quad \text{gdje } c_j(n) \to c_j \in \mathbb{R} \text{ kad } n \to \infty.$$

Također, pokazali su da uz neke blage pretpostavke proizvoljna kontrolna strategija zadovoljava

$$\mathbb{E}\left(T_j(n)\right) \geq c_j \ln n$$

za velike n, što čini prethodno opisane strategije asimptotski optimalnima. Pokazali su da, za veliku klasu distribucija nagrada, ne postoji kontrolna strategija čije žaljenje raste sporije od $\ln n$. U skladu s tim, smatramo da kontrolna strategija rješava problem kompromisa

između istraživanja i iskorištavanja, odnosno da optimalno rješava problem višeručnog bandita, ako njezino žaljenje ne raste brže od toga.

U članku [5] predstavljena je jednostavna kontrolna strategija, nazvana UCB1, koja uz vrlo malo pretpostavki na distribucije nagrada za sve n postiže

$$\mathbb{E}\left(T_j(n)\right) \leq c \ln n + c', \quad \text{gdje je } 0 \leq c, c' \in \mathbb{R}.$$

Definirajmo

$$\overline{X}_{j,n} = \frac{1}{n} \sum_{t=1}^{n} X_{j,t}, \quad 1 \leq j \leq K, n \geq 1,$$

odnosno prosječnu nagradu za ruku j u n igranja. Uz takvu notaciju, definirajmo determinističku kontrolnu strategiju UCB1 na sljedeći način:

1. Za $n = 1, \ldots, K$, odigraj ruku n (obavi inicijalizaciju igranjem svake ruke jednom);

2. Nakon $n \geq K$ poteza, odaberi ruku

$$i \in \arg\max_j \left(\overline{X}_{j,T_j(n)} + \sqrt{\frac{2 \ln n}{T_j(n)}} \right). \tag{2.4}$$

   Ako je više takvih ruku, jednu odaberi nasumično.

Pritom je UCB u nazivu kontrolne strategije skraćenica za *upper confidence bound*, odnosno gornju pouzdanu ogradu. Član $\overline{X}_{j,T_j(n)}$ u jednadžbi (2.4) potiče iskorištavanje opcija koje su u prošlosti donijele veću prosječnu nagradu, dok član $\sqrt{\frac{2 \ln n}{T_j(n)}}$ potiče istraživanje u prošlosti manje odabiranih opcija.

Teorem 2.4.3. *Neka je $K > 1$ i neka su $X_{i,n}$, $1 \leq i \leq K$ i $n \geq 1$, slučajne nagrade takve da za svaki i te za svaki n vrijedi $\mathcal{R}(X_{i,n}) = [0, 1]$. Pretpostavimo da uzastopno odabiremo ruke slijedeći kontrolnu strategiju UCB1. Tada za svaki $n \geq 1$ vrijedi*

$$R_n \leq \left(8 \sum_{j:\mu_j < \mu^*} \left(\frac{\ln n}{\Delta_j} \right) \right) + \left(1 + \frac{\pi^2}{3} \right) \left(\sum_{j=1}^{K} \Delta_j \right).$$

Dokaz teorema 2.4.3 može se pronaći u [16]. Možemo vidjeti da je žaljenje R_n odozgo ograničeno višekratnikom $\ln n$ uniformno za sve n te da je stoga UCB1 kontrolna strategija koja optimalno rješava problem višeručnog bandita.

Osnovni algoritam pretraživanja stabla Monte Carlo metodom

Pretraživanje stabla Monte Carlo metodom počiva na sljedeća dva fundamentalna koncepta:

- Procjeni stvarne isplativosti poduzimanja pojedine akcije na temelju stohastičkih simulacija;
- Efikasnom iskorištavanju vrijednosti isplativosti akcija za prilagođavanje politike u smjeru najbolji-prvi politike (politike koja prvo proširuje čvor koji najviše obećava po nekom kriteriju).

Algoritam progresivno gradi parcijalno stablo igre, vođen rezultatima prethodno provedenih pretraživanja tog stabla. Stablo se koristi radi procjene vrijednosti pojedinih poteza, pri čemu te procjene (osobito one za poteze koji najviše obećavaju) postaju sve točnije kako se stablo izgrađuje.

Osnovni algoritam pretraživanja stabla Monte Carlo metodom podrazumijeva iterativnu izgradnju stabla pretraživanja dok se ne dosegne neki unaprijed definirani *računski budžet*. Računski budžet obično podrazumijeva neko vremensko, memorijsko ili ograničenje na broj iteracija. Kada se računski budžet dosegne, pretraga se zaustavlja i algoritam vraća akciju koja se pokazala najboljom od svih akcija legalnih u stanju koje predstavlja korijen. Svaki čvor u stablu pretraživanja predstavlja stanje domene, a usmjereni bridovi koji iz pojedinog čvora izlaze i ulaze u čvorove djecu predstavljaju akcije koje iz stanja koje predstavlja taj čvor vode u naredna stanja.

Svaka se iteracija osnovnog algoritma pretraživanja stabla Monte Carlo metodom sastoji od četiri koraka. Koraci su sljedeći:

1. *Selekcija*: Počevši od korijena, rekurzivno se primjenjuje politika odabira djeteta radi spuštanja niz stablo dok se ne dosegne najhitniji proširivi čvor ili čvor koji predstavlja završno stanje. Za čvor kažemo da je *proširiv* ako predstavlja nezavršno stanje i u skladu s domenom ima djece koja još nisu dio stabla. Ako je dosegnut proširiv čvor, algoritam se nastavlja u koraku 2. U slučaju da dosegnemo čvor koji predstavlja završno stanje, preskačemo korake 2–3 i obavljamo propagiranje unatrag vrijednosti dosegnutog završnog stanja, počevši od čvora koji mu odgovara pa unatrag;
2. *Proširivanje*: Odabire se jedna (ili više) legalna akcija u stanju koje predstavlja proširivi čvor te se stablo proširuje dodavanjem čvora djeteta (ili više njih) koje predstavlja sljedeće stanje nakon poduzimanja odgovarajuće legalne akcije u danom stanju;
3. *Simulacija*: Izvodi se simulacija iz novog čvora (čvorova) na temelju zadane politike kako bi se proizveo ishod;

4. *Propagiranje unatrag*: Počevši od tek dodanog čvora, rezultat simulacije propagira se unatrag kroz čvorove odabrane tijekom koraka selekcije radi ažuriranja njihovih statističkih podataka.

Prva tri navedena koraka mogu se grupirati na temelju politika koje koriste. Te su politike sljedeće, a već smo ih spominjali ranije u odjeljku 2.4:

1. *Politika stabla*: Odabire čvor već sadržan u stablu pretraživanja ili kreira list; koriste je koraci selekcije i proširivanja;

2. *Zadana politika*: Nakon što politika stabla odabere prošireni čvor i kreira list, zadana politika generira niz akcija radi dobivanja procjene vrijednosti pojedinih čvorova odabranih od strane politike stabla, odnosno akcija koje predstavljaju bridovi koji ulaze u njih. Koristi je korak simulacije.

Četvrti korak, odnosno korak propagiranja unatrag, ne koristi nijednu politiku, no ažurira statističke podatke čvorova, koji utječu na buduće odluke diktirane od strane politike stabla.

Koraci 1–4 sumirani su algoritmom 1.

Algoritam 1 Osnovni MCTS algoritam.

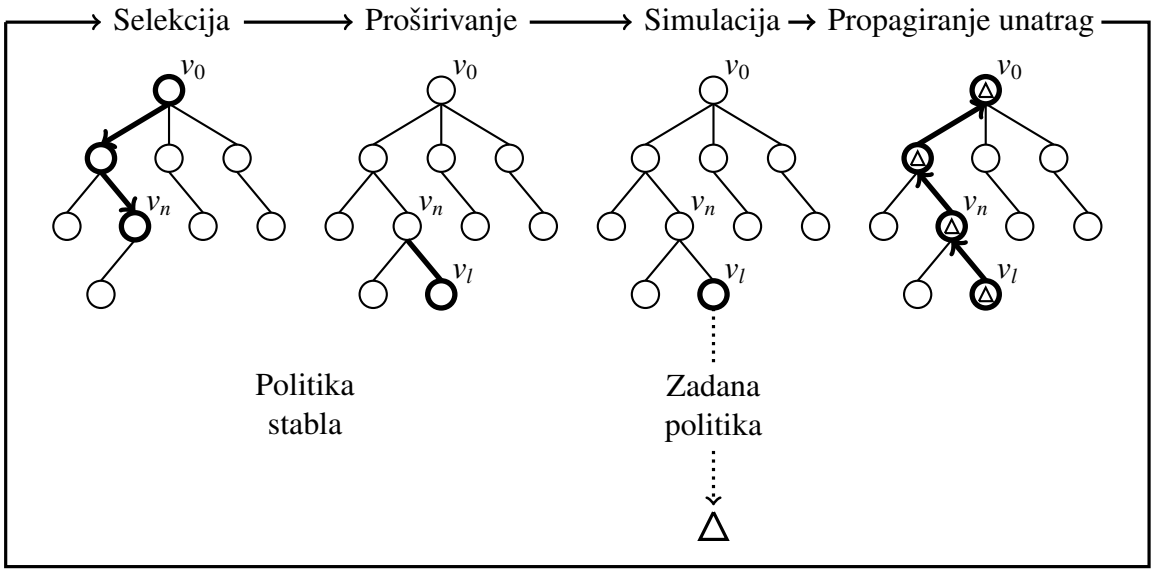
OsnovniMCTS(s_0)
Ulaz: s_0: stanje kojemu želimo da odgovara korijen
Izlaz: a (NajboljeDijete(v_0)): akcija koja vodi k najboljem djetetu korijena

```
kreiraj korijen v_0 koji odgovara stanju s_0
while unutar računskog budžeta do
    v_l ← PolitikaStabla(v_0)
    Δ ← ZadanaPolitika(s(v_l))
    PropagirajUnatrag(v_l, Δ)
return a (NajboljeDijete(v_0))
```

U algoritmu 1 v_0 je korijen stabla pretraživanja, koji odgovara stanju s_0 (koje je ulaz algoritma), v_l je posljednji čvor dosegnut u stadiju politike stabla, a Δ je nagrada za završno stanje dosegnuto pomoću zadane politike, odnosno tijekom simulacije koja počinje iz stanja $s(v_l)$ (stanja kojemu odgovara čvor v_l). Algoritam vraća a (NajboljeDijete(v_0)), odnosno akciju koju predstavlja brid koji vodi k najboljem djetetu korijena v_0, pri čemu točna definicija „najboljeg" djeteta ovisi o konkretnoj implementaciji.

Slika 2.2, napravljena po uzoru na sliku iz [7], prikazuje jednu iteraciju osnovnog algoritma pretraživanja stabla Monte Carlo metodom. Svaki čvor stabla pretraživanja sadrži statističke podatke o vrijednostima nagrada i broju posjeta. Počevši s korijenom v_0, djeca se rekurzivno odabiru u skladu s nekom funkcijom korisnosti sve dok se ne dosegne čvor v_n

Slika 2.2: Jedna iteracija osnovnog MCTS algoritma.



koji predstavlja završno stanje ili je proširiv (primijetimo da v_n ne mora biti list stabla pretraživanja). Potom se u odgovarajućem stanju s, ako je ono nezavršno, odabire prethodno nepoduzeta akcija a i u stablo se dodaje novi list v_l, koji predstavlja stanje s', koje slijedi nakon odabira akcije a u stanju s. Time završava stadij politike stabla aktualne iteracije.

Potom se izvodi simulacija iz tek dodanog lista v_l (odnosno, odgovarajućeg stanja $s(v_l)$), s ciljem dobivanja vrijednosti nagrade Δ u završnom stanju, koja se zatim propagira unatrag kroz v_l i čvorove odabrane u koraku selekcije aktualne iteracije radi ažuriranja statističkih podataka čvorova. Ažuriranje podrazumijeva inkrementiranje brojača posjeta svakog pojedinog čvora i mijenjanje vrijednosti prosječne ili kumulativne nagrade na temelju vrijednosti Δ. Vrijednost nagrade Δ može biti diskretna i u slučaju igre označavati pobjedu, odnosno neriješen ishod ili gubitak, kontinuirana ili, u slučaju kompleksnijih višeagentskih domena, vektor, čije komponente odgovaraju nagradama pojedinih agenata.

Kada se pretraživanje prekine ili je dosegnut računski budžet, algoritam vraća akciju a koja je legalna u stanju kojemu odgovara korijen v_0 na temelju nekog kriterija. Četiri uobičajena kriterija za odabir povratne akcije sljedeća su:

- *Maksimalno dijete*: Odaberi dijete s najvećom prosječnom (ili kumulativnom) nagradom;
- *Robusno dijete*: Odaberi najposjećenije dijete;
- *Maksimalno-robusno dijete*: Odaberi dijete koje je najposjećenije i ima najveću prosječnu (ili kumulativnu) nagradu. Ako takvo ne postoji, onda nastavi pretraživanje

dok se ne postigne zadovoljavajući broj posjeta;

- *Sigurno dijete*: Odaberi dijete koje maksimira donju pouzdanu ogradu, odnosno dijete

$$v_{lcb} \in \arg\max_{v \in C(v_0)} \left(\frac{Q(v)}{N(v)} - c \sqrt{\frac{2 \ln N(v_0)}{N(v)}} \right),$$

gdje je c konstanta i gdje pomoću $C(v)$ označavamo skup djece čvora v, pomoću $N(v)$ broj posjeta čvoru v, a pomoću $Q(v)$ kumulativnu nagradu čvora v. Ako je takve djece više, jedno odaberi nasumično.

Od navedena četiri kriterija, najčešće je korišten kriterij robusno dijete.

Gornje pouzdane ograde za stabla

U ovom ćemo odjeljku predstaviti najpopularniji algoritam iz klase algoritama pretraživanja stabla Monte Carlo metodom, algoritam *gornje pouzdane ograde za stabla* (engl. *Upper Confidence Bound for Trees*, UCT).

Kao što smo već objasnili u ranijim odjeljcima, pretraživanje stabla Monte Carlo metodom iterativno gradi parcijalno stablo pretraživanja u svrhu aproksimiranja (stvarne) vrijednosti akcija legalnih u danom stanju. Točan način na koji se stablo izgrađuje ovisi o tome kako se čvorovi u stablu odabiru, a uspjeh MCTS algoritma ovisi upravo o politici stabla, koja određuje način selekcije čvorova. Politika stabla koja se pokazala uspješnom utemeljena je na kontrolnoj strategiji UCB1 za problem K-rukog bandita, predstavljenoj u pododjeljku *Bandit-bazirane metode* unutar odjeljka 2.4. Ideja je iza takve politike stabla shvatiti problem odabira djeteta u stablu kao višeručni bandit problem, u kojemu nagrade procijenjene Monte Carlo simulacijama, koje odgovaraju vrijednostima pojedinih čvorova djece, odnosno akcija koje vode u njima pridružena stanja, predstavljaju slučajne varijable s nepoznatim distribucijama.

Možemo se pitati: „zašto baš UCB1?" Naime, UCB1 ima neka obećavajuća svojstva, koja smo već diskutirali u pododjeljku *Bandit-bazirane metode*: jednostavna je, efikasna te zbog toga što joj žaljenje ne raste brže od $\ln n$, optimalno rješava problem višeručnog bandita. Stoga je obećavajući kandidat za rješavanje problema kompromisa istraživanja i iskorištavanja u pretraživanju stabla Monte Carlo metodom. Naime, svaki se problem selekcije čvora (akcije) iz postojećeg stabla može modelirati kao nezavisan problem višeručnog bandita. U tom slučaju, uz v_0 kao čvor roditelj i $C(v_0)$ kao skup njegove djece, ponovno možemo definirati slučajne varijable $X_{i,n}$, za $1 \leq i \leq |C(v_0)|$ i $n \geq 1$, te

$$\overline{X}_{j,n} = \frac{1}{n} \sum_{t=1}^{n} X_{j,t}, \quad 1 \leq j \leq |C(v_0)|,\ n \geq 1,$$

odnosno prosječnu vrijednost djeteta j nakon n posjeta. Kao u teoremu 2.4.3, smatramo da vrijedi $\mathcal{R}(X_{i,n}) = [0, 1]$, za svaki $1 \leq i \leq |C(v_0)|$ i $n \geq 1$. Iz toga slijedi da za svaki $1 \leq j \leq |C(v_0)|$ te svaki $n \geq 1$ vrijedi i da je vrijednost $\overline{X}_{j,n}$ unutar intervala $[0, 1]$.

Izložena ideja korištenja strategije UCB1 nalaže odabir djeteta

$$i \in \arg\max_j \left(\overline{X}_{j,T_j(n)} + 2C_p \sqrt{\frac{2 \ln n}{T_j(n)}} \right), \tag{2.5}$$

gdje je n broj posjeta čvoru roditelju, $T_j(n)$ broj posjeta djetetu j, a $C_p > 0$ konstanta. Pritom vrijednost izraza

$$\overline{X}_{j,T_j(n)} + 2C_p \sqrt{\frac{2 \ln n}{T_j(n)}} \tag{2.6}$$

nazivamo UCT vrijednošću djeteta j. U slučaju da je $\arg\max_j \left(\overline{X}_{j,T_j(n)} + 2C_p \sqrt{\frac{2 \ln n}{T_j(n)}} \right)$ skup kardinaliteta većeg od 1, nasumično odabiremo jedan njegov element. Možemo se zapitati: „što ako je $T_j(n) = 0$ za neki j i neki n?" Naime, budući da se u sklopu iteracije pretraživanja stabla Monte Carlo metodom u kojoj se u koraku proširivanja u stablo dodaje čvor j tijekom koraka propagiranja unatrag njegov brojač posjeta inkrementira (odnosno, postavlja na 1), smatramo da vrijedi $T_j(n) > 0$ za svaki $1 \leq j \leq |C(v_0)|$ i za svaki $n \geq 1$.

Promotrimo značenje prvog, odnosno drugog pribrojnika u izrazu (2.6) za UCT vrijednost djeteta j te njihov međusoban odnos. Dok prvi pribrojnik potiče iskorištavanje, drugi vodi k istraživanju pa prvi obično nazivamo i članom iskorištavanja, a drugi članom istraživanja. S povećanjem broja posjeta određenom čvoru, nazivnik se u odgovarajućem drugom pribrojniku povećava pa se doprinos drugog pribrojnika cijelom izrazu smanjuje. S druge strane, ako se povećava broj posjeta drugom djetetu roditelja danog čvora, brojnik odgovarajućeg drugog pribrojnika se povećava te se stoga i vrijednost koja potiče istraživanje povećava. Član istraživanja osigurava da se, uz dovoljno vremena, i djeca s malim prosječnim vrijednostima kad tad odaberu, zbog čega možemo reći da on algoritmu također osigurava i svojstvo *restarta*. Količina istraživanja može se regulirati podešavanjem vrijednosti konstante C_p.

Algoritam se dalje nastavlja kao što je opisano u pododjeljku *Osnovni algoritam pretraživanja stabla Monte Carlo metodom* unutar odjeljka 2.4. Naime, ako je čvor odabran u koraku selekcije proširiv, to jest ako u skladu s domenom ima djece koja još nisu dio stabla, jedno se od te djece nasumično odabire i dodaje u stablo. Potom se primjenjuje zadana politika, koja je u najjednostavnijem slučaju uniformno slučajna, dok se ne dosegne završno stanje. Vrijednost Δ završnog stanja s_T potom se propagira unatrag kroz sve čvorove posjećene tijekom aktualne iteracije, počevši od tek dodanog čvora pa do korijena.

Neka je v čvor u stablu pretraživanja. Tada v sadrži vrijednosti $N(v)$, odnosno broj puta kad je bio posjećen, i $Q(v)$, odnosno kumulativnu nagradu za sva odigravanja koja su

prošla kroz stanje kojemu on odgovara. Na temelju tih dviju vrijednosti možemo izračunati i prosječnu nagradu igraču koji poduzima akciju kojoj odgovarajući brid ulazi u dani čvor; ona je jednaka $\frac{Q(v)}{N(v)}$. Vrijednosti čvora ažuriraju se svaki put kad je on dio odigravanja iz korijena. Jednom kad se dosegne računski budžet, algoritam završava i vraća najbolji potez u korijenu, odnosno onaj kojemu odgovara brid koji ulazi u dijete korijena s najvećom prosječnom nagradom ili s najviše posjeta.

Pseudokod UCT algoritma prikazan je algoritmom 2, a pseudokodovi pomoćnih funkcija prikazani su u algoritmu 3.

Algoritam 2 UCT algoritam.

UCTPretraživanje(s_0)
Ulaz: s_0: stanje kojemu želimo da odgovara korijen
Izlaz: a (NajboljeDijete($v_0, 0$)): akcija koja vodi k najboljem djetetu korijena

```
kreiraj korijen v_0 koji odgovara stanju s_0
while unutar računskog budžeta do
    v_l ← PolitikaStabla(v_0)
    Δ ← ZadanaPolitika(s(v_l))
    PropagirajUnatrag(v_l, Δ)
return a (NajboljeDijete(v_0, 0))
```

U skladu s algoritmom 2, svaki čvor v struktura je podataka koja sadrži sljedeće podatke:

- $s(v)$: stanje koje predstavlja;
- $a(v)$: akcija koju predstavlja brid čiji je ulazni čvor v;
- $Q(v)$: suma nagrada svih simulacija iz iteracija u kojima je v bio posjećen;
- $N(v)$: broj posjeta čvoru v.

Nadalje, $\Delta(v, p)$ u algoritmu 3 predstavlja komponentu vektora nagrade Δ koja odgovara igraču p čija akcija vodi u stanje koje predstavlja čvor v (ako igru igraju dva igrača, onda ona odgovara igraču koji ne povlači potez u stanju koje predstavlja v). Povratna je vrijednost algoritma a (NajboljeDijete($v_0, 0$)), odnosno akcija a koju predstavlja brid koji vodi k djetetu s najvećom prosječnom nagradnom (povratna se akcija odabire po kriteriju maksimalnog djeteta, opisanoj u pododjeljku *Osnovni algoritam pretraživanja stabla Monte Carlo metodom* unutar odjeljka 2.4)). Algoritam bi također mogao povratnu akciju odabrati po kriteriju robusnog djeteta, odnosno mogao bi odabrati akciju koja vodi k najposjećenijem djetetu. U praksi, ta će dva kriterija često (ali ne uvijek) dati istu akciju.

Algoritam 3 Pomoćne funkcije za algoritam 2.

PolitikaStabla(v)
Ulaz: v: korijen stabla
Izlaz: v: posljednji čvor dosegnut u stadiju politike stabla

```
while v odgovara nezavršnom stanju do
  if v je proširiv then
    return Proširi(v)
  else
    v ← NajboljeDijete(v, C_p)
return v
```

Proširi(v)
Ulaz: v: proširivi čvor
Izlaz: v': novododani čvor

```
odaberi a ∈ neiskušane akcije iz skupa 𝒜(s(v))
dodaj novo dijete v' čvora v
    uz s(v') = 𝒫(s(v), a)[a]
    i a(v') = a
return v'
```

NajboljeDijete(v, c)
Ulaz: v: čvor čije dijete odabiremo, c: vrijednost konstante C_p u izrazu (2.6)
Izlaz: v': odabrano dijete

return $\arg\max_{v' \in C(v)} \left(\frac{Q(v')}{N(v')} + c \sqrt{\frac{2 \ln N(v)}{N(v')}} \right)$

ZadanaPolitika(s)
Ulaz: s: početno stanje
Izlaz: Δ: nagrada za završno stanje dosegnuto tijekom simulacije

```
while s nije završno stanje do
  odaberi a ∈ 𝒜(s) uniformno nasumično
  s ← 𝒫(s, a)
return nagrada za stanje s
```

PropagirajUnatrag(v, Δ)
Ulaz: v: novododani čvor, Δ: nagrada za završno stanje dosegnuto tijekom simulacije

```
while v nije null do
  N(v) ← N(v) + 1
  Q(v) ← Q(v) + Δ(v, p)
  v ← roditelj čvora v
```

[a] Funkcija $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ deterministička je funkcija prijelaza definirana u odjeljku 1.3.

Algoritam 4 UCT propagiranje unatrag za igre sa sumom nula i dva igrača koji poteze povlače naizmjenično.

PROPAGIRANJEUNATRAGNEGAMAX(v, Δ)

Ulaz: v: novododani čvor, Δ: nagrada za završno stanje dosegnuto tijekom simulacije agentu koji povlači potez kojemu odgovara brid koji vodi k novododanom čvoru

while v nije *null* **do**
  $N(v) \leftarrow N(v) + 1$
  $Q(v) \leftarrow Q(v) + \Delta$
  $\Delta \leftarrow -\Delta$
  $v \leftarrow$ roditelj čvora v

Algoritam 4 prikazuje alternativni i efikasniji način propagiranja unatrag od onog opisanog u algoritmu 3, koji se obično pronalazi u literaturi za igre sa sumom nula i dva igrača koji poteze povlače naizmjenično, na kakve se najdirektnije i primjenjuje AlphaZero metoda. Takav je pristup analogan *negamaks* varijanti minimaks metode, u kojoj skalarna vrijednost nagrade mijenja predznak na svakoj razini stabla. Dok je Δ u algoritmu 3 vektor čije komponente odgovaraju nagradama pojedinih agenata, u algoritmu 4 Δ je skalarna vrijednost koja predstavlja nagradu agentu koji povlači potez kojemu odgovara brid koji vodi k novododanom čvoru.

UCT algoritam u radovima [13] i [14] predstavili su Kocsis i Szepesvári. Empirijski su demonstrirali funkcioniranje UCT varijante pretraživanja stabla Monte Carlo metodom primijenjene na više domena i pokazali da vjerojatnost neuspjeha (vjerojatnost odabira suboptimalne akcije u korijenu stabla) konvergira k nuli polinomijalnom brzinom kad broj simuliranih igara raste u beskonačnost. Taj dokaz implicira da, uz dovoljno vremena i memorije, UCT stablo konvergira k minimaks stablu te da je iz te perspektive UCT optimalan.

Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu

AlphaZero koristi varijantu algoritma polinomijalne gornje pouzdane ograde za stabla (engl. *Polynomial Upper Confidence Bound for Trees*, PUCT), verziju UCT algoritma opisanog u pododjeljku *Gornje pouzdane ograde za stabla* unutar odjeljka 2.4. U nastavku opisujemo sve korake PUCT algoritma, kakve nalazimo u AlphaZero algoritmu.

Neka je v čvor stabla pretraživanja koji predstavlja stanje s. Za svaku legalnu akciju $a \in \mathcal{A}(s)$ čvor v ima izlazni brid koji odgovara toj akciji. Svaki je brid struktura podataka koja sadrži sljedeće podatke:

- $N(s,a)$: broj posjeta bridu, koji predstavlja akciju a povučenu u stanju s;
- $W(s,a)$: kumulativna vrijednost akcije a u stanju s koju brid predstavlja;

- $Q(s, a)$: prosječna vrijednost akcije a u stanju s koju brid predstavlja;
- $P(s, a)$: prethodna vjerojatnost (engl. *prior probability*) odabira akcije a u stanju s koju brid predstavlja.

Dok je za korak selekcije u pododjeljku *Osnovni algoritam pretraživanja stabla Monte Carlo metodom* unutar odjeljka 2.4 bilo navedeno da se sastoji od primjenjivanja politike odabira djeteta dok se ne dosegne proširivi čvor koji nije nužno list (ili čvor koji predstavlja završno stanje), u pretraživanju stabla Monte Carlo metodom unutar AlphaZero algoritma korak selekcije počinje u korijenu, a završava kada se u nekom trenutku L dosegne list. Neka je stanje koje list predstavlja s_L. U svakom trenutku $t < L$ odabire se jedan od bridova na temelju statističkih podataka pohranjenim u njima. PUCT algoritam nalaže da se u trenutku t odabere brid iz čvora koji predstavlja stanje s_t koji odgovara nekoj (ako ih je više, jednu odabiremo nasumično) akciji

$$a_t \in \arg\max_a \left(Q(s_t, a) + U(s_t, a)\right),$$

pri čemu za neko stanje s i akciju $a \in \mathcal{A}(s)$ vrijedi

$$U(s, a) = C(s)P(s, a)\frac{\sqrt{\sum_{b \in \mathcal{A}(s)} N(s, b)}}{1 + N(s, a)},$$

gdje je

$$C(s) = \ln\left(\frac{1 + \sum_{b \in \mathcal{A}(s)} N(s, b) + c_{\text{base}}}{c_{\text{base}}}\right) + c_{\text{init}},$$

odnosno gdje je $C(s)$ stopa istraživanja, koja sporo raste s vremenom pretraživanja stabla, ali je gotovo konstantna u slučaju kratkog izvršavanja algoritma. Pritom su c_{base} i c_{init} konstante. Za neko stanje s i akciju $a \in \mathcal{A}(s)$ vrijednost izraza

$$\text{PUCT}(s, a) = Q(s, a) + U(s, a) \tag{2.7}$$

nazivamo PUCT vrijednošću akcije a u stanju s.

Prisjetimo se diskusije o značenju pojedinih pribrojnika u izrazu (2.6) iz pododjeljka *Gornje pouzdane ograde za stabla* unutar odjeljka 2.4. Slično kao što je u izrazu za UCT vrijednost drugi pribrojnik poticao istraživanje, tako i drugi pribrojnik u jednadžbi (2.7) također potiče istraživanje. Opisani algoritam pretraživanja na početku preferira akcije s velikom prethodnom vjerojatnošću i malim brojem posjeta, ali asimptotski ipak prednost daje akcijama s velikim prosječnim vrijednostima.

Kada selekcija odabere brid koji ulazi u list, ako list predstavlja nezavršno stanje s_L, proširujemo[2] ga i evaluiramo. Umjesto izvođenja koraka simulacije opisanog u pododjeljku *Osnovni algoritam pretraživanja stabla Monte Carlo metodom* unutar odjeljka 2.4, vrijednost stanja s_L iz perspektive igrača koji u njemu povlači potez dobivamo pomoću neuronske mreže. I proširivanje i evaluacija lista temelje se na pronalaženju vrijednosti neuronske mreže f_θ za ulaz s_L. Naime, izlaz neuronske mreže za ulaz s_L jednak je $f_\theta(s_L) = (\mathbf{p}, v)$, gdje je $\mathbf{p}$ vektor čije komponente predstavljaju vjerojatnosti odabira pojedinih akcija u stanju s_L i gdje je v skalar koji procjenjuje ishod pridružen trenutku L, odnosno vrijednost stanja s_L za igrača koji u njemu povlači potez. Neka je $\mathcal{A}$ skup akcija takav da je $|\mathcal{A}| = n$, $n \in \mathbb{N}$. Tada za vektor $\mathbf{p}$ vrijedi $\mathbf{p} \in [0,1]^n$ i $\sum_{j=1}^{n} \mathbf{p}_j = 1$. Iz $|\mathcal{A}| = n$ slijedi da postoji bijekcija između skupova $\{1, 2, \ldots, n\}$ i $\mathcal{A}$. Odaberimo jednu takvu bijekciju $i : \mathcal{A} \to \{1, 2, \ldots, n\}$ (odabir se u praksi svodi na odabir načina modeliranja domene, odnosno skupa akcija). Uz takav odabir i notaciju, smatramo da vrijedi $\mathbf{p}_{i(a)} = \mathbb{P}(A_L = a \,|\, S_L = s_L)$, za $a \in \mathcal{A}$.

List proširujemo na način da za svaku akciju $a \in \mathcal{A}(s_L)$ u stablo dodamo po jedan brid i odgovarajući čvor dijete. Neka je a akcija legalna u stanju s_L. Tada brid koji odgovara akciji a u stanju s_L inicijaliziramo na sljedeći način: $N(s_L, a)$, $W(s_L, a)$ i $Q(s_L, a)$ postavljamo na 0, a $P(s_L, a)$ postavljamo na $\mathbf{p}_{i(a)}$.

Nakon proširivanja i evaluacije, slijedi korak propagiranja unatrag vrijednosti v. Propagiranje unatrag vrši se i u slučaju da je list predstavljao završno stanje; u tom se slučaju unatrag propagira nagrada za pripadajuće završno stanje. U nastavku simbolom v označavamo vrijednost koja se propagira, neovisno o tome radi li se o vrijednosti koja je izlaz neuronske mreže ili nagradi za završno stanje.

Propagiranje unatrag sastoji se od ažuriranja podataka svih bridova odabranih u koraku selekcije, u trenucima $0 \leq t < L$. Označimo pomoću a_t akciju odabranu u trenutku t, kojemu odgovara stanje s_t (s_0 je stanje koje predstavlja korijen). Za svaki $0 \leq t < L$, inkrementiramo vrijednost $N(s_t, a_t)$, $W(s_t, a_t)$ uvećavamo za v ili $-v$ i na kraju postavljamo $Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}$.

Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu ustvari je algoritam koji na temelju parametara neuronske mreže θ i stanja s_0, koje treba predstavljati korijen, vraća vektor $\boldsymbol{\pi} = \alpha_\theta(s_0)$ vjerojatnosti proizašlih iz pretraživanja, na temelju kojih se odabire jedna od akcija legalnih u stanju s_0. Neka je $a \in \mathcal{A}(s_0)$. Odgovarajuća je komponenta vektora $\boldsymbol{\pi}$ jednaka

$$\boldsymbol{\pi}_{i(a)} = \frac{N(s_0, a)^{\frac{1}{\tau}}}{\sum_{b \in \mathcal{A}(s_0)} N(s_0, b)^{\frac{1}{\tau}}}, \tag{2.8}$$

[2]Dok je korak proširivanja kakav je bio opisan u pododjeljku *Osnovni algoritam pretraživanja stabla Monte Carlo metodom* unutar odjeljka 2.4 podrazumijevao proširivanje stabla dodavanjem jednog čvora djeteta čvora koji proširujemo, u sklopu MCTS-a unutar AlphaZero algoritma čvor se proširuje tako da se u stablo odmah dodaju sva njegova djeca.

gdje je τ takozvani *parametar temperature*, čija je uloga kontrola količine istraživanja. Pretpostavljamo da je $\boldsymbol{\pi}_{i(a)} = 0$ za $a \in \mathcal{A} \setminus \mathcal{A}(s_0)$. Na uobičajene vrijednosti parametra temperature i njihov utjecaj na vrijednost izraza (2.8) dodatno ćemo se osvrnuti u odjeljku 2.5. Na temelju vjerojatnosti proizašlih iz pretraživanja obično se odabiru mnogo bolji potezi od onih čiji odabir nalažu vjerojatnosti odabira pojedinih akcija koje su izlaz neuronske mreže. Stoga se pretraživanje Monte Carlo metodom može smatrati moćnom metodom poboljšanja kontrolne strategije.

2.5 Iteracija kontrolne strategije kroz igranje algoritma samog protiv sebe

Učenje igranja igre AlphaZero postiže igranjem igara algoritma samog protiv sebe, kojim se ostvaruje algoritam iteracije kontrolne strategije. Tijekom igranja igara protiv samoga sebe, algoritam nalaže odabir akcija na osnovu kontrolne strategije proizašle iz pretraživanja stabla Monte Carlo metodom, a ishod z pojedine igre koristi kao uzorak vrijednosti stanja kroz koje je igra prošla — u skladu s tim, na igranje igara algoritma samog protiv sebe na temelju pretraživanja stabla Monte Carlo metodom možemo gledati kao na uspješnu metodu procjene kontrolne strategije. Glavna je ideja algoritama iz klase AlphaZero koristiti takvu metodu procjene kontrolne strategije i pretraživanje stabla Monte Carlo metodom kao metodu poboljšanja kontrolne strategije više puta te na taj način ostvariti proceduru iteracije kontrolne strategije.

Takvim algoritmom strojnog učenja podrškom na temelju igranja igara algoritma samog protiv sebe, uz odabir akcija na temelju vjerojatnosti proizašlih iz pretraživanja stabla Monte Carlo metodom, trenira se neuronska mreža f_θ. Težine neuronske mreže na početku se inicijaliziraju na nasumične vrijednosti θ_0. U svakoj iteraciji $i \geq 1$ generiraju se igre algoritma protiv samoga sebe. Odnosno, u svakoj iteraciji $i \geq 1$ program igra nekoliko igara protiv samoga sebe, od kojih se pojedina igra sastoji od niza stanja $s_1, \ldots, s_T$. Za svako stanje s_t, poziva se algoritam pretraživanja stabla Monte Carlo metodom sa stanjem s_t kao ulazom, koji vraća $\boldsymbol{\pi}_t = \alpha_{\theta_{i-1}}(s_t)$. Kao što smo naveli u pododjeljku *Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu* unutar odjeljka 2.4, $\boldsymbol{\pi}_t$ je vektor vjerojatnosti proizašlih iz pretraživanja, na temelju kojih se u stanju s_t odabire jedna od legalnih akcija. Naime, prisjetimo se da je za akciju $a \in \mathcal{A}(s_t)$ odgovarajuća komponenta vektora $\boldsymbol{\pi}$ jednaka

$$\boldsymbol{\pi}_{i(a)} = \frac{N(s_t, a)^{\frac{1}{\tau}}}{\sum\limits_{b \in \mathcal{A}(s_t)} N(s_t, b)^{\frac{1}{\tau}}}. \tag{2.9}$$

Odabir akcije vršimo na sljedeći način. Možemo definirati diskretan vjerojatnosni prostor $(\Omega, \mathcal{P}(\Omega), P)$, gdje je prostor elementarnih događaja $\Omega = \mathcal{A}$, i diskretnu slučajnu varijablu

$X : \Omega \to \mathbb{R}$ na tom vjerojatnosnom prostoru takvu da je $X(a) = i(a)$, za $a \in \Omega = \mathcal{A}$. Pritom je $i : \mathcal{A} \to \{1, \ldots, |\mathcal{A}|\}$ bijekcija iz pododjeljka *Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu* unutar odjeljka 2.4. Vrijednostima koje varijabla X može poprimiti, odnosno elementima skupa $\mathcal{R}(X) = \{1, \ldots, |\mathcal{A}|\}$, pridružujemo vjerojatnosti na sljedeći način. Neka je $j \in \{1, \ldots, |\mathcal{A}|\}$. Tada je

$$p_j = \mathbb{P}(X = j) = \boldsymbol{\pi}_j.$$

Odnosno, ako je $a \in \mathcal{A}$ neka akcija, smatramo da je njoj pridružena vjerojatnost

$$p_{i(a)} = \boldsymbol{\pi}_{i(a)}.$$

Promatramo kategoričku distribuciju (generalizaciju Bernoullijeve distribucije) s takvim vjerojatnostima i $k = |\mathcal{A}|$ kao parametrima. Ona opisuje moguće vrijednosti slučajne varijable X. Akciju odabiremo uzimanjem uzorka iz takve kategoričke distribucije. Odnosno, odabiremo akciju $a = i^{-1}(X_j)$, gdje je X_j uzorak.

Primijetimo da vjerojatnosti p_i, $i \in \{1, \ldots, |\mathcal{A}|\}$ ovise o vrijednosti parametra temperature τ. U radu [22], τ je postavljen na 1 za prvih 30 poteza jedne igre, a potom se smatra da vrijedi $\tau \to 0$, odnosno da je τ infinitezimalne vrijednosti. Promotrimo kako svaka od te dvije vrijednosti utječe na vjerojatnosti. Za početak, promotrimo slučaj kada je $\tau = 1$. Tada za svaki $a \in \mathcal{A}(s_t)$ vrijedi

$$\boldsymbol{\pi}_{i(a)} = \frac{N(s_t, a)}{\sum\limits_{b \in \mathcal{A}(s_t)} N(s_t, b)},$$

odnosno vjerojatnost pridružena svakoj legalnoj akciji jednostavno je normalizirani broj posjeta bridu koji odgovara toj akciji povučenoj u stanju s_t. Promotrimo sada slučaj kada vrijedi $\tau \to 0$. U tu svrhu, definirajmo skup $\mathcal{A}(s_t)_{\max}$ kao

$$\mathcal{A}(s_t)_{\max} = \arg\max_{a \in \mathcal{A}(s_t)} N(s_t, a).$$

Neka je $a_{\max}$ neki element tog skupa. Pretpostavljamo da je $N(s_t, a_{\max}) \neq 0$, odnosno da je barem jedan brid iz čvora koji predstavlja stanje s_t barem jednom posjećen. Raspišimo

limes izraza (2.9) kada τ teži u 0 na sljedeći način.

$$\lim_{\tau\to 0}\boldsymbol{\pi}_{i(a)} = \lim_{\tau\to 0}\frac{N(s_t,a)^{\frac{1}{\tau}}}{\sum\limits_{b\in\mathcal{A}(s_t)}N(s_t,b)^{\frac{1}{\tau}}} \tag{2.10}$$

$$= \lim_{\tau\to 0}\frac{\frac{N(s_t,a)^{\frac{1}{\tau}}}{N(s_t,a_{\max})^{\frac{1}{\tau}}}}{\frac{\sum\limits_{b\in\mathcal{A}(s_t)}N(s_t,b)^{\frac{1}{\tau}}}{N(s_t,a_{\max})^{\frac{1}{\tau}}}} \tag{2.11}$$

$$= \lim_{\tau\to 0}\frac{\left(\frac{N(s_t,a)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}}{\sum\limits_{b\in\mathcal{A}(s_t)}\left(\frac{N(s_t,b)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}} \tag{2.12}$$

Ako je $N(s_t,a) = 0$ za neki $a \in \mathcal{A}(s_t)$, tada je $\lim\limits_{\tau\to 0}\boldsymbol{\pi}_{i(a)} = \lim\limits_{\tau\to 0} 0 = 0$. Također, kao što smo već naveli u pododjeljku *Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu* unutar odjeljka 2.4, smatramo da je $\boldsymbol{\pi}_{i(a)} = 0$ za $a \in \mathcal{A} \setminus \mathcal{A}(s_t)$. Za svaku akciju $a \in \mathcal{A}(s_t)_{\max}$, izraz (2.12) jednak je

$$\lim_{\tau\to 0}\frac{1}{\sum\limits_{b\in\mathcal{A}(s_t)}\left(\frac{N(s_t,b)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}} = \lim_{\tau\to 0}\frac{1}{\sum\limits_{b\in\mathcal{A}(s_t)_{\max}}1 + \sum\limits_{b\in\mathcal{A}(s_t)\setminus\mathcal{A}(s_t)_{\max}}\left(\frac{N(s_t,b)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}} = \frac{1}{|\mathcal{A}(s_t)_{\max}|}$$

jer je $N(s_t,a) = N(s_t,a_{\max})$ i jer je $\frac{N(s_t,b)}{N(s_t,a_{\max})} < 1$ za svaki $b \in \mathcal{A}(s_t) \setminus \mathcal{A}(s_t)_{\max}$. Neka je sada $a \in \mathcal{A}(s_t) \setminus \mathcal{A}(s_t)_{\max}$ takav da je $N(s_t,a) \neq 0$. Za takvu akciju, izraz (2.12) jednak je

$$\lim_{\tau\to 0}\frac{\left(\frac{N(s_t,a)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}}{\sum\limits_{b\in\mathcal{A}(s_t)_{\max}}1 + \sum\limits_{b\in\mathcal{A}(s_t)\setminus\mathcal{A}(s_t)_{\max}}\left(\frac{N(s_t,b)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}} = \lim_{\tau\to 0}\frac{\left(\frac{N(s_t,a)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}}{|\mathcal{A}(s_t)_{\max}| + \sum\limits_{b\in\mathcal{A}(s_t)\setminus\mathcal{A}(s_t)_{\max}}\left(\frac{N(s_t,b)}{N(s_t,a_{\max})}\right)^{\frac{1}{\tau}}} = 0$$

jer je $\frac{N(s_t,a)}{N(s_t,a_{\max})} < 1$. Dakle, $p_{i(a)} = \frac{1}{|\mathcal{A}(s_t)_{\max}|}$ za svaki $a \in \mathcal{A}(s_t)_{\max}$, a za svaki $a \in \mathcal{A}(s_t) \setminus \mathcal{A}(s_t)_{\max}$ vrijedi $p_{i(a)} = 0$. Odnosno, uniformno slučajno odabiremo jednu od akcija kojoj odgovara brid s maksimalnim brojem posjeta. Ako je $|\mathcal{A}(s_t)_{\max}| = 1$, odabir akcije je deterministički.

Dodatno se istraživanje postiže dodavanjem Dirichletovog šuma prethodnim vjerojatnostima pojedinih bridova koji izlaze iz korijena. Odnosno, neka je s_t stanje koje predstavlja korijen i neka je a neka akcija legalna u stanju s_t. Tada postavljamo

$$P(s_t,a) = (1-\varepsilon)\mathbf{p}_{i(a)} + \varepsilon\boldsymbol{\eta}_{i(a)},$$

gdje je $\mathbf{p}$ vektor vjerojatnosti odabira pojedinih akcija u stanju s_t koji je jedan od izlaza neuronske mreže $f_{\theta_{i-1}}$ za ulaz s_t, $0 \leq \varepsilon \leq 1$ parametar, a $\boldsymbol{\eta} \sim \text{Dir}(\boldsymbol{\alpha})$, pri čemu je $\boldsymbol{\alpha} \in \mathbb{R}_+^{|\mathcal{A}|}$ parametar Dirichletove distribucije.

Stablo pretraživanja dobiveno pretraživanjem stabla Monte Carlo metodom u koraku t koristi se u narednim koracima na način da čvor u koji ulazi brid koji predstavlja odabranu akciju postane novi korijen i da se odgovarajuće podstablo zadrži, a da se ostatak stabla odbaci. Igra završava u koraku T, kada se dosegne završno stanje s_T, kojemu se pridružuje nagrada r_T. Iz svakog pojedinog koraka $1 \leq t < T$ pohranjuje se uređena trojka $(s_t, \boldsymbol{\pi}_t, z_t)$, gdje je z_t ishod igre pridružen trenutku t, definiran u pododjeljku *Alternirajuće Markovljeve igre* unutar odjeljka 1.3.

Po završetku iteracije i, novi parametri neuronske mreže θ_i dobivaju se treniranjem neuronske mreže na temelju primjera za učenje oblika $(s, (\boldsymbol{\pi}, z))$, dobivenih uzimanjem uzorka iz diskretne uniformne distribucije na skupu uređenih trojki $(s_t, \boldsymbol{\pi}_t, z_t)$ iz svih koraka t svih odigranih igara u iteraciji i (ili u nekoliko posljednjih iteracija). Točan način treniranja neuronske mreže, odnosno mijenjanja njenih parametara bit će opisan u odjeljku 2.6. Ažurirani parametri θ_i potom se koriste u igrama algoritma samog protiv sebe u narednoj iteraciji.

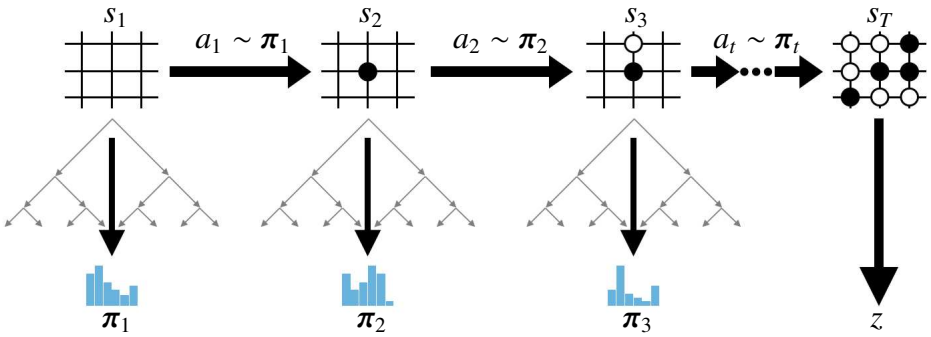
Opisani proces učenja na temelju igranja igara algoritma samog protiv sebe ilustriran je slikom 2.3. Na 2.3a prikazan je proces igranja igara algoritma samog protiv sebe, odnosno prolazak kroz stanja $s_1, \ldots, s_T$, u svakom od kojih se poziva MCTS algoritam α_θ, koji koristi aktualnu neuronsku mrežu f_θ. Ilustrirani su odabiri akcija na temelju vjerojatnosti proizašlih iz pretraživanja stabla Monte Carlo metodom (odabir akcije $a_t \sim \boldsymbol{\pi}_t$ u koraku t) te pridjeljivanje nagrade završnom stanju s_T, na temelju koje se računaju ishodi za pojedine trenutke. Na 2.3b ilustrirano je treniranje neuronske mreže na temelju primjera za učenje, od kojih se svaki sastoji od reprezentacije stanja kao ulazne varijable te vjerojatnosti proizašlih iz pretraživanja i ishoda igre za trenutak koji odgovara ulaznom stanju kao ciljnim varijablama. Neuronska mreža ulazno stanje transformira pomoću brojnih konvolucijskih slojeva s parametrima θ te vraća vektor $\boldsymbol{p}_t$ i skalar v_t.

2.6 Nadzirano učenje

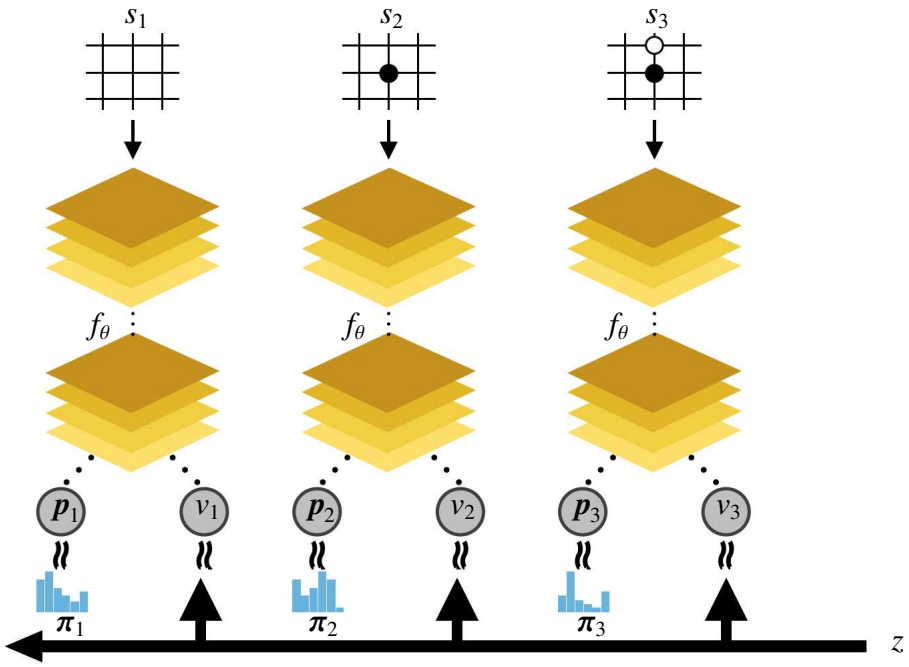
U ovom ćemo odjeljku detaljnije opisati proces nadziranog učenja u AlphaZero algoritmu. Nadzirano učenje u kontekstu AlphaZero algoritma podrazumijeva učenje funkcije $f : \mathcal{S} \to \Delta\mathcal{A} \times \mathbb{R}$, pri čemu je $\Delta\mathcal{A} = \{x \in [0,1]^{\mathcal{A}} : \sum_{a \in \mathcal{A}} x(a) = 1\}$, odnosno funkcije koja aproksimira ciljnu funkciju koja pojedinom stanju $s \in \mathcal{S}$ pridružuje vjerojatnosnu distribuciju na skupu akcija $\mathcal{A}$ i očekivani ishod igre za trenutak kojemu odgovara stanje s. AlphaZero podrazumijeva konvolucijske neuronske mreže kao tip reprezentacije, odnosno podrazumijeva da se prostor hipoteza (engl. *hypothesis space*) algoritma nadziranog učenja

Slika 2.3: Strojno učenje podrškom na temelju igranja algoritma samog protiv sebe koje čini AlphaZero algoritam.

(a) Igranje igara algoritma samog protiv sebe.



(b) Treniranje neuronske mreže.



sastoji od konvolucijskih neuronskih mreža. Kao što smo već naveli, u svakoj se pojedinoj iteraciji neuronska mreža trenira na temelju primjera za učenje oblika $(s, (\boldsymbol{\pi}, z))$, odnosno primjera koji se sastoje od reprezentacije pojedinog stanja s kao ulazne varijable te vjerojatnosti proizašlih iz pretraživanja stabla Monte Carlo metodom i odgovarajućeg ishoda igre kao ciljnim varijablama. No, kako evaluiramo model, ondosno na koji se točno način

ažuriraju parametri neuronske mreže?

Svaki algoritam nadziranog učenja kao svoju komponentu sadrži evaluaciju — način na koji ocjenjujemo ili preferiramo jedan model u odnosu na drugi. Kako bi se kvantificirala uspješnost pojedinog modela, nužna je mjera greške. Osim evaluacije, svaki algoritam nadziranog učenja sastoji se i od optimizacije, odnosno pretraživanja prostora reprezentiranih modela s ciljem postizanja što povoljnijih vrijednosti mjere greške. AlphaZero u sklopu optimizacije ažurira parametre, odnosno traži konvolucijsku neuronsku mrežu koja minimizira mjeru greške izvedenu iz funkcije dane jednadžbom:

$$L((\boldsymbol{\pi}, z), (\mathbf{p}, v)) = (z - v)^2 - \boldsymbol{\pi}^T \log_2 \mathbf{p} + c\|\theta\|^2, \quad (\mathbf{p}, v) = f_\theta(s), s \in \mathcal{S}. \tag{2.13}$$

Dakle, parametri neuronske mreže θ ažuriraju se tako da maksimiraju sličnosti između pojedinih izlaznih varijabli $\mathbf{p}$ i pripadnih vjerojatnosti proizašlih iz pretraživanja stabla Monte Carlo metodom $\boldsymbol{\pi}$ te da minimiziraju razlike između pojedinih predviđenih ishoda v i odgovarajućih ishoda z. Odnosno, funkcija gubitka (engl. *loss function*) (2.13) sumira srednju kvadratnu pogrešku (engl. *mean squared error*, MSE) za ishod z i predviđeni ishod v te vrijednost *cross-entropy* funkcije gubitka za vjerojatnosti proizašle iz pretraživanja stabla $\boldsymbol{\pi}$ i predviđene vjerojatnosti $\mathbf{p}$. Srednja kvadratna pogreška i *cross-entropy* često su korištene funkcije gubitka u strojnom učenju — dok se vrijednost srednje kvadratne pogreške računa kvadriranjem razlike između predikcije i stvarne vrijednosti, *cross-entropy* koristi se u višeklasnoj klasifikaciji na sljedeći način. Ako pomoću p_i označimo *softmax* vjerojatnost pripadnosti ulaznog primjera klasi i (pri čemu ukupno postoji n klasa), a pomoću t_i pripadnu stvarnu vrijednost, vrijednost *cross-entropy* za dani je primjer jednaka

$$L_{CE} = -\sum_{i=1}^{n} t_i \log_2(p_i).$$

Pritom *softmax* vjerojatnostima nazivamo vjerojatnosti dobivene primjenom *softmax* funkcije na izlaz iz neuronske mreže. Naime, *softmax* funkcija $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ definirana je jednadžbom :

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \ldots, K \text{ i } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K. \tag{2.14}$$

Dakle, ako je izlaz neuronske mreže vektor $\mathbf{y} = (y_1, \ldots, y_K)$, za neki $K \in \mathbb{N}$, *softmax* vjerojatnosti dobivene iz tog vektora jednake su

$$p_i = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}, \quad i = 1, \ldots, K.$$

Kao što smo pojasnili, prvi je član s desne strane jednadžbe (2.13) srednja kvadratna greška za z i v, a drugi je član vrijednost *cross-entropy* funkcije gubitka za $\boldsymbol{\pi}$ i $\mathbf{p}$. Treći je

član L_2 regularizacijski član, pri čemu je $c \geq 0$ regularizacijski koeficijent, koji kontrolira razinu L_2 regularizacije. Naime, član $c\|\theta\|^2$ penalizira težine modela različite od nule, čime se nastoji poboljšati generalizacija modela, odnosno njegova prediktivna moć.

U radu [22] kao optimizacijski algoritam koristi se gradijentni spust (engl. *gradient descent*), odnosno gradijentnim se spustom nastoje pronaći parametri konvolucijske neuronske mreže koji minimiziraju navedenu mjeru greške.

Do sada smo nekoliko puta spomenuli da neuronska mreža u AlphaZero algoritmima kao ulaz prima pojedino stanje igre, odnosno njegovu reprezentaciju, i da mu pridružuje vektor vjerojatnosti odabira pojedinih akcija te predviđeni ishod. No, kako točno reprezentirati stanja i kako modelirati izlaze neuronske mreže? U radu [22] korištena je sljedeća reprezentacija stanja. Intuitivno, pojedino se stanje igre s_t s igraćom pločom dimenzija $N \times N$ predaje neuronskoj mreži kao stog slika formata $N \times N \times (MT + L)$, odnosno kao konkatenacija T skupova M ravnina dimenzija $N \times N$ i jednog skupa L ravnina ponovno dimenzija $N \times N$. Neka je $1 \leq t' \leq T$. Tada t'-ti skup M ravnina predstavlja stanje igre $s_{t-t'+1}$, odnosno stanje igre u trenutku $t - t' + 1$, pri čemu se postavlja na nule u slučaju negativnosti trenutka. Igraća ploča u stanju s u reprezentaciji se promatra iz perspektive igrača koji vuče potez u s; skup M ravnina značajki može se particionirati na podskup binarnih matrica koje kodiraju prisutnosti igračevih figura na pojedinim dijelovima ploče, pri čemu se dodaje po jedna ravnina za svaki tip figure, i podskup binarnih matrica koje kodiraju prisutnosti protivnikovih figura (pri čemu i za taj podskup vrijedi pravilo o dodavanju po jedne ravnine za svaki tip figure). Dodatnih L ravnina konstatne su i one označavaju boje igrača, broj poteza i stanje posebnih pravila, specifičnih za konkretnu igru.

Reprezentacija kontrolne strategije — konkretno, dimenzije vektora predviđenih vjerojatnosti pojedinih akcija i vjerojatnosti proizašlih iz pretraživanja stabla Monte Carlo metodom, ovisi o danoj igri. Primjerice, u radu [22] kontrolna strategija za igru Go modelirana je distribucijom na skupu koji se sastoji od $19 \cdot 19 + 1$ elemenata, od kojih jedan predstavlja akciju propuštanja reda, a od ostalih svaki predstavlja akciju postavljanja figure na jednu od mogućih pozicija na igraćoj ploči. Pritom su u istom radu u svim instancama algoritma (algoritmima za sve konkretne igre) akcije koje nisu legalne u nekom stanju „zamaskirane" postavljenjem njihovih vjerojatnosti na nulu i potom normaliziranjem vrijednosti legalnih akcija.

Poglavlje 3

Programsko ostvarenje

U prethodnim smo poglavljima dali odgovor na pitanje što AlphaZero jest — podrobno smo opisali strukturu AlphaZero metode, kao i izložili osnovne koncepte relevantne pozadinske teorije, te smo govorili o uspješnosti AlphaZero algoritma u raznim zahtjevnim domenama — njegovoj, u radu [22] demonstriranoj, superiornosti u šahu, shogiju i igri Go naspram i ljudskih i računalnih igrača. U ovom ćemo poglavlju pokazati njegove mogućnosti implementacijom AlphaZero metode za igru Četiri u nizu pomoću programskog jezika Python i dodatnih biblioteka za njega. Navest ćemo neke implementacijske detalje te u odjeljku 3.6 predstaviti sposobnosti agenta ne samo po završetku učenja, odnosno izvođenja napisanog programa, već i u različitim stadijima učenja — prikazat ćemo napredak u ovisnosti o vremenu, kao i konačno postignut uspjeh u odabranoj domeni.

3.1 Igra Četiri u nizu

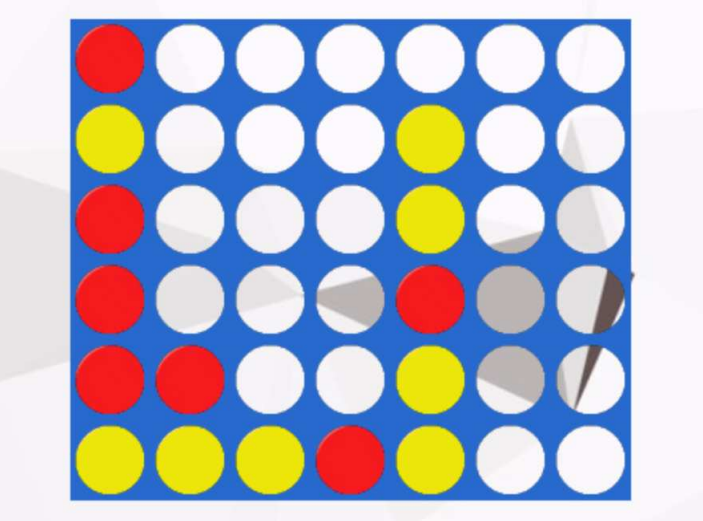
Kao što smo već naveli ranije u radu, mogućnosti AlphaZero metode demonstriramo njenom implementacijom za igru Četiri u nizu. Četiri u nizu društvena je igra za dva igrača, u kojoj svakome od njih biva dodijeljena jedna boja te u kojoj naizmjence ubacuju diskove pripadajuće boje u rešetku, obično sa šest redaka i sedam stupaca. Kada dođe red na pojedinog igrača, on mora ubaciti disk svoje boje u neki od stupaca rešetke s barem jednim praznim mjestom. Pošto jedan od igrača ubaci disk u rešetku, disk upada u nju, na najniže slobodno mjesto — diskovi ubačeni u isti stupac slažu se vertikalno jedan na drugog. Cilj je pojedinog igrača biti prvi koji će formirati vodoravni, okomiti ili dijagonalni niz od četiri diska svoje boje.

Četiri u nizu igra je sa sumom nula. Naime, igrači su u striktnoj opoziciji — ako jedan uspije formirati niz od četiri diska svoje boje, on pobjeđuje, dok drugi gubi. U slučaju da se rešetka u potpunosti ispuni, bez da se formira ijedan niz od četiri diska iste boje, igra se smatra neriješenom. Također, Četiri u nizu je i igra s potpunim informacijama — u svakom

je trenutku dostupna informacija o točnom stanju u kojemu se igra nalazi.

U skladu s navedenim, Četiri u nizu može se modelirati kao alternirajuća Markovljeva igra, a na alternirajuće se Markovljeve igre najizravnije primjenjuje AlphaZero metoda. Primjer realizacije igre Četiri u nizu, u kojoj su boje koje se dodjeljuju igračima crvena i žuta, vidljiv je na slici 3.1.

Slika 3.1: Primjer realizacije igre Četiri u nizu.



Igru Četiri u nizu intuitivno bismo mogli okarakterizirati kao jednostavnu — pogotovo ako je promatramo relativno u odnosu na kompleksne igre poput igre Go — no, ukupan broj različitih stanja igre, prema [3], iznosi $4531985219092 \approx 4.5 \cdot 10^{12}$, što predstavlja izazov računalima u smislu rješavanja igre *brute-force* metodama, odnosno sistematičnim pretraživanjem svih mogućnosti. Upravo zbog neposrednosti svojih pravila, jednostavnosti, koja se ogleda u manjim (u odnosu na kompleksnije igre poput igre Go) zahtjevima na resurse prilikom implementacije AlphaZero algoritma za njeno savladavanje, popularnosti, ali ujedno i netrivijalnosti, igra Četiri u nizu dobar je kandidat za domenu na kojoj bismo u sklopu ovog rada demonstrirali mogućnosti AlphaZero metode.

3.2 Modeliranje igre

Kao što smo naveli u pododjeljku *Domensko znanje* odjeljka 2.3, jedino domensko znanje dostupno AlphaZero algoritmima ono je o pravilima igre, strukturi igraće ploče, skupu stanja i akcija. Naveli smo kako je to ono znanje koje ovisi o konkretnoj igri koju pojedini algoritam iz klase AlphaZero algoritama uči igrati. U skladu s tim, opišimo u ovom odjeljku ta znanja u našoj implementaciji, odnosno kako modeliramo okoliš — stanja,

akcije igre, njena pravila i dinamiku. Naime, s obzirom na to da algoritam koji implementiramo, kao što smo objasnili ranije u radu, počiva na igranju igara samog protiv sebe i pretraživanju stabla Monte Carlo metodom, ključno je omogućiti određivanje sljedećeg stanja nakon poduzimanja bilo koje legalne akcije u bilo kojem stanju.

Igra Četiri u nizu u našoj je implementaciji modelirana po uzoru na model iz 23. poglavlja knjige [17]. Motivacija za odabir upravo takvog modela jednaka je motivaciji autora navedene knjige: nastojanje da reprezentacija pojedinog stanja igre zauzima što je manje memorije moguće, ali da istovremeno omogućava efikasan rad s njima. Naime, tijekom pretraživanja stabla Monte Carlo metodom potrebno je u memoriji čuvati podatke koji predstavljaju čvorove stabla; svaki čvor sadrži podatak o stanju koje predstavlja, a samo stanje potencijalno je predstavljeno „velikim" (u smislu količine memorije koja je potrebna za njegovo pohranjivanje) brojem. S obzirom na navedeno te činjenicu da kvaliteta izlaza algoritma pretraživanja stabla Monte Carlo metodom, odnosno vjerojatnosti proizašlih iz pretraživanja, pa stoga i kvaliteta igre, ovise o broju čvorova posjećenih (pa i pohranjenih) tijekom pretraživanja stabla Monte Carlo metodom, kompaktnost reprezentacije stanja igre potencijalno ima značajan utjecaj na zahtjeve na količinu memorije i performanse tijekom procesa učenja. Osim što od reprezentacije stanja zahtijevamo da bude kompaktna, želimo osigurati i praktičnost rada s njom; htjeli bismo na praktičan i efikasan način moći provjeriti je li neko stanje završno, utvrditi koje su akcije legalne u danome stanju te odrediti sljedeće stanje nakon poduzimanja neke od takvih akcija.

Kako bismo zadovoljili oba navedena kriterija, implementiramo dvije različite reprezentacije jedne te iste konfiguracije rešetke igre te definiranjem odgovarajućih funkcija omogućavamo pretvorbe jedne u drugu. Naime, radi osiguranja kompaktnosti reprezentacije, stanje igre modeliramo pomoću klase `GameState`, koja, među ostalim, sadrži sljedeća dva atributa:

- `state_int`: brojčana reprezentacija rešetke u odgovarajućem stanju;
- `to_play`: indikator toga koji igrač povlači potez u danome stanju (s obzirom na to da je Četiri u nizu igra za dva igrača, jednog označavamo pomoću 1, a drugog pomoću 0).

Pritom je `state_int` takozvani *kodirani* oblik reprezentacije rešetke, koji cijelu rešetku kodira pomoću samo 63 bita, zbog čega u slučaju izvođenja programa na računalu 64-bitne arhitekture osigurava brzinu i efikasnost. Alternativna reprezentacija rešetke naziva se *dekodiranim* oblikom te je ustvari lista čiji su elementi 7 lista, od kojih svaka predstavlja jedan stupac rešetke, odnosno diskove koji se nalaze u njemu. Dok takva reprezentacija zauzima više memorije, praktičnija je za rad pa mogućnost pretvorbe kodiranog oblika u nju omogućava zadovoljenost i drugog ranije navedenog kriterija za odabir modela.

Objasnimo sada detaljnije dva spomenuta oblika reprezentacije rešetke igre. Naveli smo već da je dekodirani oblik ustvari lista 7 lista. Pokažimo na primjeru kako on točno

izgleda. Promotrimo ponovno konfiguraciju rešetke sa slike 3.1. Označimo crvene diskove (kao i igrača kojem pripadaju) pomoću 1, a žute diskove (i pripadnog igrača) pomoću 0. Uz takve oznake, prikazanu konfiguraciju možemo zapisati na sljedeći način:

1						
0				0		
1				0		
1				1		
1	1			0		
0	0	0	1	0		

.

Dekodirani oblik reprezentacije dane konfiguracije jednak je

```
[[0, 1, 1, 1, 0, 1],
 [0, 1],
 [0],
 [1],
 [0, 0, 1, 0, 0],
 [],
 []],
```
(3.1)

odnosno i-ta lista (gdje je $i = 1, \ldots, 7$) unutar liste koja je jednaka dekodiranom obliku lista je čiji su elementi brojčane reprezentacije diskova u i-tom stupcu rešetke, pri čemu prvi element odgovara disku na najnižem mjestu stupca rešetke, dok zadnji element odgovara onome na najvišem popunjenom mjestu odgovarajućeg stupca rešetke.

Kodirani oblik zapravo je cijeli broj čiji se binarni zapis sastoji od $7 \cdot 6 + 7 \cdot 3$ bitova, od kojih prvih $7 \cdot 6$ kodiraju boje diskova u pojedinim stupcima rešetke, dok drugih $7 \cdot 3$ predstavljaju brojeve slobodnih mjesta na vrhovima pojedinih stupaca unutar rešetke. Primjerice, za konfiguraciju prikazanu slikom 3.1, čiji je dekodirani oblik reprezentacije dan u (3.1), binarni se prikaz kodiranog oblika reprezentacije sastoji od sljedećih bitova:

```
[011101, 010000, 000000, 100000, 001000, 000000, 000000
 000, 100, 101, 101, 001, 110, 110],
```

odnosno jednak je

```
01110101000000000001000000010000000000000000000010010110100111011O.
```

Dakle, i-tih (pri čemu je $i = 1, \ldots, 7$) 6 bitova jednako je elementima i-te liste unutar dekodiranog oblika redom od prvog pa do zadnjeg, nadopunjenima nulama na kraju u

slučaju da je odgovarajuća lista duljine manje od 6. Nakon prva 42 bita, j-ta (gdje je $j = 1, \ldots, 7$) 3 bita jednaka su binarnom zapisu broja praznih mjesta na vrhu j-tog stupca rešetke.

Kao što smo već naveli, definiramo funkcije koje omogućavaju pretvorbu jednog oblika reprezentacije u drugi, odnosno funkciju `decode_binary` koja prima kodirani oblik te vraća dekodirani, kao i funkciju `encode_lists`, koja prima dekodirani oblik, a kao izlaz vraća kodirani. Te su funkcije analogne istoimenim funkcijama iz [17, Poglavlje 23], gdje se mogu pronaći i njihova dodatna objašnjenja.

Pravila i dinamiku igre implementiraju, odnosno provjeru je li neko stanje završno, dohvaćanje legalnih akcija u danome stanju i određivanje sljedećeg stanja nakon poduzimanja neke od takvih akcija omogućavaju metode klase `GameState`. Naime, klasa `GameState`, među ostalima, ima sljedeće metode:

- `legal_actions(self)`: funkcija koja na temelju atributa klase `state_int`, pomoću funkcije `decode_binary`, dohvaća dekodirani oblik reprezentacije odgovarajuće konfiguracije rešetke igre te provjerava duljina kojih je lista (njenih elemenata) manja od 6. Kao izlaz vraća listu njihovih indeksa, odnosno indeksa stupaca s praznim mjestima (stupaca u koje je moguće ubaciti novi disk);

- `terminal(field, col)`: statička metoda koja za dani indeks stupca određuje vodi li akcija ubacivanja diska u zadani stupac u stanju u kojemu njeno poduzimanje vodi u stanje s rešetkom čija je konfiguracija u dekodiranom obliku reprezentirana poljem danim varijablom `field` u završno stanje. Točnije, funkcija na temelju dekodiranog oblika reprezentacije konfiguracije rešetke nakon ubacivanja diska u odgovarajući stupac utvrđuje postoji li vodoravni, okomiti ili dijagonalni niz od 4 diska jednake boje, odnosno, ako ne, je li rešetka puna i igra neriješena. Ako funkcija utvrdi da u rešetci postoji niz od 4 diska iste boje, povratna joj je vrijednost 1 (ona označava pobjedu igrača koji je ubacio disk u dani stupac), a inače vraća 0 ili -1: 0 ako je rešetka puna i igra neriješena, a -1 ako ubacivanje diska nije dovelo do završnog stanja;

- `take_action(self, col)`: funkcija koja kao ulaz prima indeks stupca, a vraća stanje u koje je prelazak posljedica ubacivanja diska igrača danog atributom `to_play` u odgovarajući stupac rešetke, zajedno s indikatorom je li sljedeće stanje završno te, u slučaju da jest, vrijednošću poduzimanja te akcije. Sljedeće stanje funkcija definira kao novu instancu klase `GameState`, čija je vrijednost atributa `to_play` jednaka „suprotnoj" vrijednosti istoimenog atributa instance nad kojom je metoda pozvana (ako je ona jednaka 1, vrijednost atributa nove instance postavlja se na 0 i obratno) te čija je vrijednost atributa `state_int` jednaka kodiranom obliku reprezentacije rešetke čiji se dekodirani oblik razlikuje od dekodiranog oblika rešetke koja odgovara

stanju koje predstavlja instanca nad kojom je metoda pozvana u tome što lista koja odgovara stupcu koji određuje ulazni indeks ima jedan element više. Pritom se radi o zadnjem elementu liste, koji je jednak 1 ako je vrijednost `to_play` atributa instance nad kojom se metoda poziva jednaka 1, a 0 inače.

Igra Četiri u nizu modelirana je pomoću klase `Game`, čiji su atributi, među ostalima, sljedeći:

- `grid_shape`: uređeni par čiji je prvi element broj redaka rešetke, a drugi broj stupaca rešetke. Konkretno, u našoj je implementaciji uvijek jednak $(6, 7)$;
- `input_shape`: uređena trojka čiji je prvi element 2, drugi element broj redaka rešetke, a treći broj stupaca rešetke. U našoj je implementaciji uvijek jednak $(2, 6, 7)$. Atribut `input_shape` određuje dimenzije ulaznih podataka neuronske mreže, odnosno dimenzije reprezentacija stanja koje su njen ulaz;
- `action_size`: broj elemenata skupa akcija $\mathcal{A}$, odnosno skupa legalnih akcija u početnom stanju igre. U našoj implementaciji on je jednak broju stupaca rešetke — skup akcija modeliramo kao skup $\{0, 1, \ldots, 6\}$, pri čemu svaki element skupa predstavlja indeks jednog od stupaca rešetke. Drugim riječima, u našem modelu skup legalnih akcija u početnom stanju čine akcije ubacivanja diska u bilo koji od sedam stupaca rešetke.

Navedeni atributi definiraju specifikacije konkretne varijante igre Četiri u nizu koju imamo na umu tijekom implementacije; međutim, omogućavaju da se uz sitne modifikacije koda algoritam implementira za drukčiju varijantu igre, s drugim dimenzijama rešetke.

Osim već spomenutih atributa, klasa `Game` ima i atribute `to_play` te `game_state`. Prilikom stvaranja nove instance klase `Game`, konstruktor inicijalizira atribut `to_play` postavljajući mu vrijednost na 1 (igrač označen pomoću 1 igra prvi) te inicijalizira atribut `game_state` postavljajući njegovu vrijednost na novu instancu klase `GameState`, s 1 kao vrijednošću atributa `to_play` i `encode_lists([[]] * 7)` kao vrijednošću njenog atributa `state_int`. Dakle, instanciranjem klase `Game` inicijaliziramo okoliš koji predstavlja igru Četiri u nizu, i to postavljanjem njegovog početnog stanja na stanje kojemu odgovara prazna rešetka i u kojemu potez povlači igrač označen pomoću 1. Tijekom igranja igara algoritma samog protiv sebe, prelasci u sljedeća stanja ostvaruju se pozivanjem metode `apply` klase `Game` na danoj instanci; za dani indeks stupca `col`, metoda `apply` postavlja vrijednost atributa `game_state` na sljedeće stanje koje je izlaz poziva funkcije `take_action` s argumentom `col` nad instancom klase `GameState` danom vrijednošću atributa `game_state`, a vrijednost atributa `to_play` postavlja na 0 ako je prethodno bila jednaka 1 i obratno. Definicija klase `Game` omogućava i „resetiranje" okoliša; klasa `Game` posjeduje i metodu `reset`, koja vrijednosti atributa `to_play` i `game_state` postavlja na

početne vrijednosti, odnosno one koje odgovaraju početnom stanju (i kakve definira konstruktor klase).

3.3 Implementacija pretraživanja stabla Monte Carlo metodom

Pretraživanje stabla Monte Carlo metodom implementirano je pomoću klasa `TreeNode` i `MCTS`. Klasa `TreeNode` predstavlja čvor stabla pretraživanja te sadrži, među ostalim, sljedeće atribute:

- `state`: stanje, odnosno instanca klase `GameState`, koje čvor predstavlja;
- `prior`: prethodna vrijednost odabira brida čiji je ulazni čvor onaj čvor koji predstavlja dana instanca klase. Ako čvor koji je roditelj prethodno spomenutog čvora predstavlja stanje s, a opisani brid predstavlja akciju a, vrijednost atributa `prior` odgovara vrijednosti $P(s,a)$, opisanoj u pododjeljku *Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu* unutar odjeljka 2.4. Ako čvor koji predstavlja dana instanca odgovara korijenu stabla, vrijednost atributa `prior` jednaka je 0;
- `visit_count`: broj posjeta čvoru koji predstavlja dana instanca klase tijekom pretraživanja stabla Monte Carlo metodom. Ako pomoću s označimo stanje koje predstavlja roditelj prethodno opisanog čvora i pomoću a akciju koju predstavlja brid čiji je ulazni čvor onaj čvor koji predstavlja dana instanca klase, tada vrijednost atributa `visit_count` odgovara vrijednosti $N(s,a)$, predstavljenoj u pododjeljku *Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu* unutar odjeljka 2.4. Slično kao i za atribut `prior`, vrijednost ovog atributa jednaka je 0 za instancu klase koja predstavlja korijen stabla;
- `value_sum`: kumulativna vrijednost akcije koju predstavlja brid koji ulazi u čvor predstavljen danom instancom klase. Ako pomoću s označimo stanje koje predstavlja čvor koji je njegov roditelj, a pomoću a akciju koju predstavlja opisani brid, vrijednost ovog atributa odgovara vrijednosti $W(s,a)$, s kojom smo se također upoznali u pododjeljku *Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu* unutar odjeljka 2.4. Slično kao i za prethodna dva navedena atributa, vrijednost ovog atributa jednaka je 0 za instance koje predstavljaju korijen stabla;
- `children`: rječnik čiji su ključevi akcije (za koje smo ranije naveli da ih označavamo brojevima od 0 do 6) i čije su vrijednosti čvorovi djeca čvora koji predstavlja dana instanca klase. Odnosno, rječnik čije su vrijednosti instance klase `TreeNode` koje

predstavljaju čvorove djecu danog čvora. Ako je dani čvor list ili još nije proširen, rječnik je prazan.

Kao što možemo vidjeti iz navedenih atributa klase `TreeNode`, umjesto definiranja zasebnih struktura podataka koje bi predstavljale bridove i pohranjivanja statističkih podataka u sklopu takvih struktura, što je implementacija koju bi možda najizravnije sugerirao pododjeljak *Pretraživanje stabla Monte Carlo metodom u AlphaZero algoritmu* unutar odjeljka 2.4, po uzoru na rješenje iz pseudokoda dostupnog u sklopu suplementarnih materijala članka [22], uloge čvorova i bridova komprimiramo u samo jednu strukturu podataka te statističke podatke vezane uz bridove pohranjujemo u čvorovima u koje ulaze. Također, primijetimo da među navedenim atributima klase `TreeNode` nema onog koji bi odgovarao prosječnoj vrijednosti akcije iz pododjeljka o MCTS-u u sklopu AlphaZero algoritma unutar odjeljka 2.4 — tu vrijednost ne pohranjujemo izravno zbog jednostavnosti njenog računanja na temelju vrijednosti atributa `value_sum` i `visit_count`. Naime, jedna je od metoda klase `TreeNode` funkcija `value`, koja, ako je vrijednost atributa `visit_count` veća od nule, vraća vrijednost atributa `value_sum` podijeljenog vrijednošću atributa `visit_count`, što, ako pomoću s označimo stanje koje predstavlja roditelj čvora kojemu odgovara instanca klase nad kojom je metoda pozvana, a pomoću a akciju koju predstavlja brid koji ulazi u dani čvor, upravo odgovara vrijednosti $Q(s,a)$ iz naše diskusije o MCTS-u kao dijelu AlphaZero algoritma unutar odjeljka 2.4. Ako je vrijednost atributa `visit_count` jednaka nuli, metoda vraća nulu. Osim metode `value`, klasa `TreeNode` ima i metodu `expanded`, koja daje odgovor na pitanje je li čvor koji predstavlja instanca nad kojom je metoda pozvana proširen ili ne. Ona vraća `True` ako je rječnik `children` dane instance klase neprazan, a `False` inače.

Pojedina se iteracija pretraživanja stabla Monte Carlo metodom ostvaruje pozivom metode `run_simulation` klase `MCTS` — navedena metoda provodi korak selekcije, počevši u korijenu stabla pretraživanja, danim atributom `root` klase `MCTS`, pa sve dok ne dosegne list, odnosno čvor nad čijom odgovarajućom instancom klase `TreeNode` metoda `expanded` vraća `False`. U skladu s prethodno objašnjenom implementacijom čvorova stabla pretraživanja na način da pohranjuju statističke podatke bridova koji u njih ulaze, u pojedinom trenutku selekcije u kontekstu implementacije govorimo o odabiru čvorova djece, umjesto odabira izlaznih bridova. Naime, metoda `run_simulation` u određenom trenutku koraka selekcije jedan čvor dijete odabire pozivom funkcije `select_child`, još jedne metode klase `MCTS`. Metoda `select_child` vraća neku akciju (koja je element skupa $\{0, 1, \ldots, 6\}$) i čvor dijete u koje pripadajući brid ulazi takve da je PUCT vrijednost, dana jednadžbom (2.7), akcije u stanju koje predstavlja roditelj čvora maksimalna. PUCT vrijednost pojedine akcije u danome stanju, odnosno, u skladu s našom implementacijom čvorova stabla pretraživanja, pojedinog čvora djeteta, računa metoda `ucb_score` klase `MCTS`. Metoda `run_simulation` u varijabli `search_path` čuva sve čvorove odabrane tijekom koraka selekcije — tu varijablu vraća kao izlaznu vrijednost, a na temelju nje se pomoću metode

`backpropagate` klase `MCTS` obavlja korak propagiranja unatrag.

Usko grlo (engl. *bottleneck*) pretraživanja stabla Monte Carlo metodom evaluacija je pomoću neuronske mreže nezavršnog stanja koje predstavlja list dosegnut u koraku selekcije, zajedno s proširivanjem danog lista, koje zahtijeva dohvaćanje vjerojatnosti odabira pojedinih akcija u odgovarajućem stanju koje kao izlaz također vraća neuronska mreža. Kako bismo povećali efikasnost pretraživanja stabla Monte Carlo metodom, po uzoru na implementaciju iz knjige [17, Poglavlje 23], obavljamo ga pomoću *mini-batch*eva; naime, prvo obavljamo nekoliko iteracija koraka selekcije — dosežemo nekoliko listova u stablu, a potom obavljamo evaluacije i proširivanja pomoću jednog izvrednjavanja neuronske mreže. Međutim, valja imati na umu da u slučaju takvog pristupa, odnosno, izvođenja nekoliko iteracija koraka selekcije u jednoj seriji, ne dobivamo jednak ishod kao kad iteracije pretraživanja stabla Monte Carlo metodom izvodimo serijski, što može predstavljati manu. Primjerice, ako izvodimo više iteracija pretraživanja stabla Monte Carlo metodom serijski, prva će proširiti korijen, druga neko dijete korijena, treća možda i neko dijete prethodno proširenog djeteta korijena i tako dalje. Međutim, ako na početku pretraživanja stabla Monte Carlo metodom u jednoj seriji izvedemo više iteracija koraka selekcije, svi će oni vratiti proširivi korijen, koji će se u koraku proširivanja proširiti. Kasnije serije (*batch*evi) moći će se sastojati od iteracija raznolikih po pitanju odabranih čvorova, no, *mini-batch* pristup u početku će biti manje efikasan po pitanju istraživanja od sekvencijalnog pretraživanja stabla Monte Carlo metodom. Kako bismo ublažili navedenu manu, u sklopu jednog pretraživanja stabla Monte Carlo metodom izvodimo više *mini-batch* iteracija — konkretno, izvodimo 20 *mini-batch* iteracija, od kojih se svaka sastoji od 8 iteracija koraka selekcije.

Kao što smo već naveli, u sklopu *mini-batch* pristupa pretraživanju stabla Monte Carlo metodom nakon izvođenja jedne serije koraka selekcije, pri čemu jedan korak selekcije provodimo pozivom metode `run_simulation`, sve dosegnute listove koji predstavljaju nezavršna stanja evaluiramo i proširujemo pomoću jednog izvrednjavanja neuronske mreže. Korak propagiranja unatrag potom provodimo za svaki pojedini dosegnuti list, i to pozivom prethodno spomenute metode `backpropagate`.

3.4 Arhitektura neuronske mreže

Nadzirano učenje u sklopu AlphaZero algoritma ostvarujemo pomoću biblioteke Keras s TensorFlow *back end*om. Keras je programsko sučelje (API) za duboko strojno učenje napisan u programskom jeziku Python, koji sadrži implementacije brojnih „gradivnih blokova” neuronskih mreža, poput slojeva i aktivacijskih funkcija, te raznovrsnih često upotrebljavanih optimizacijskih algoritama.

Stvaranje instanci neuronskih mreža i rad s njima omogućeni su definiranjem klase `GeneralModel` i njene podklase `ResCNN`. Klasa `GeneralModel` omogućava rad s općenitim

modelom neuronske mreže: pomoću svojih metoda ostvaruje dohvaćanje izlaza modela za dani ulaz, treniranje neuronske mreže, učitavanje pohranjenih modela ili pohranjivanje onog koji predstavlja instanca klase nad kojom je odgovarajuća metoda za spremanje pozvana, kao i prikaz slojeva, odnosno parametara neuronske mreže. Prilikom stvaranja instance klase `GeneralModel`, konstruktor, među ostalim, prima vrijednosti hiperparametara poput regularizacijskog koeficijenta i stope učenja (engl. *learning rate*) optimizacijskog algoritma te ih pridružuje atributima `regularization_const`, odnosno `learning_rate`, redom. Klasa `GeneralModel`, osim upravo navedenih atributa, ima i atribute `input_dim` te `output_dim`, čije vrijednosti određuju dimenzije ulaznih podataka neuronske mreže, odnosno tenzora izlaznih vrijednosti na temelju kojih dobivamo vjerojatnosti odabira pojedinih akcija, redom. U našem slučaju, vrijednost atributa `input_dim` uvijek je $(2, 6, 7)$, a atributa `output_dim` 7. Naime, o modeliranju akcija već smo govorili u odjeljku 3.2, a o reprezentaciji stanja koju kao ulaz prima neuronska mreža govorit ćemo više u nastavku ovog odjeljka. Klasa `ResCNN`, koja je podklasa upravo objašnjene klase `GeneralModel`, u konstruktoru također prima i podatak koji određuje skrivene slojeve rezidualne konvolucijske neuronske mreže za koju želimo da je predstavlja stvorena instanca klase — taj podatak objekt pohranjuje kao vrijednost atributa `hidden_layers` te na temelju njega definira arhitekturu neuronske mreže, instancira je i pohranjuje kao svoj atribut. Jednostavno eksperimentiranje s hiperparametrima i različitim arhitekturama neuronske mreže omogućeno je ovakvim općenitim definicijama klasa — naime, svi se hiperparametri navode u datoteci `config.py` te se iz nje u pojedinom izvršavanju programskog koda učitavaju i prenose konstruktorima u ovom odjeljku opisanih klasa.

U našoj implementaciji koristimo jednostavniju verziju arhitekture neuronske mreže iz rada [22], koju čine ulazni sloj, koji prima ulazne podatke i prosljeđuje ih narednim slojevima radi daljnjeg procesuiranja, te 5 skrivenih slojeva, koji stvaraju značajke koje se potom prosljeđuju takozvanim glavama politike i vrijednosti (engl. *policy head*, odnosno *value head*, redom), koje su kombinacija konvolucijskog sloja i potpuno povezanog sloja. Glava politike vraća vrijednosti na temelju kojih dobivamo vjerojatnosti odabira pojedine akcije u stanju čija je reprezentacija ulazni podatak neuronske mreže, dok glava vrijednosti vraća podatak tipa `float` koji interpretiramo kao vrijednost ulaznog stanja. Prvi skriveni sloj neuronske mreže konvolucijski je sloj, dok je svaki sljedeći rezidualni blok jednake građe.

Kao što smo istaknuli više puta ranije u radu, neuronska mreža kao ulaz prima reprezentaciju pojedinog stanja igre. Također, ranije smo u ovom odjeljku naveli da je vrijednost atributa `input_dim` objekata tipa `GeneralModel` uvijek $(2, 6, 7)$. Pojasnimo sada detaljnije kako izgledaju ulazni podaci naše neuronske mreže. Naime, pretvorba stanja igre predstavljenih objektima klase `GameState`, opisane u odjeljku 3.2, u ulazne podatke neuronske mreže obavlja se pomoću metode `binary` klase `GameState`. Metoda `binary`

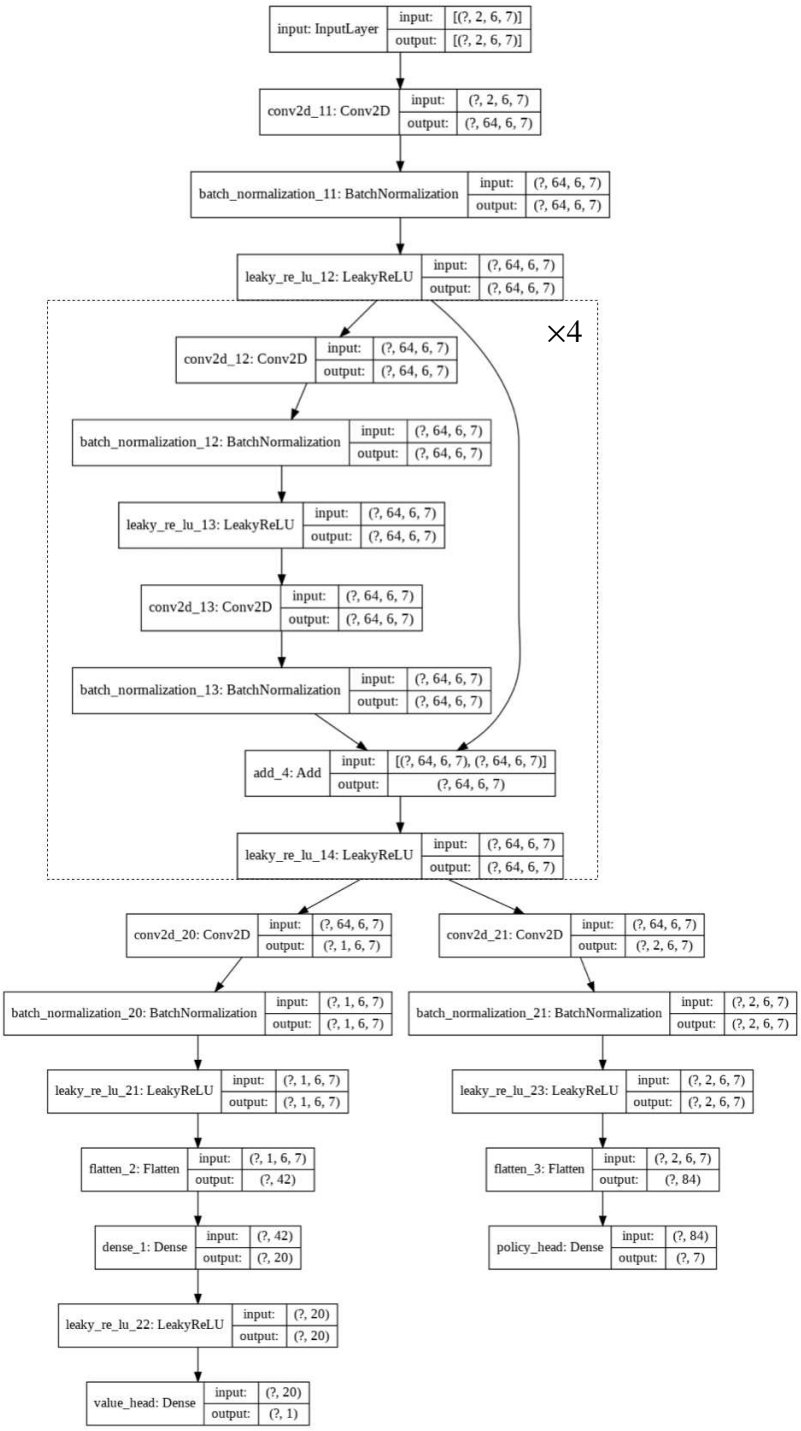
instancira NumPy[1] tenzor ispunjen nulama dimenzija $(2, 6, 7)$, a potom vrijednost atributa `state_int` pomoću metode `decode_binary` pretvara u dekodirani oblik reprezentacije rešetke te na temelju njega na pozicije $(0, i, j)$ $(i = 0, \ldots, 5,\ j = 0, \ldots, 6)$ definiranog NumPy tenzora upisuje jedinice ako je $(5 - i)$-ti element j-te liste dekodiranog oblika reprezentacije rešetke jednak vrijednosti atributa `to_play`. Analogno, na pozicije $(1, i, j)$ $(i = 0, \ldots, 5,\ j = 0, \ldots, 6)$ upisuje jedinice ako je $(5 - i)$-ti element j-te liste dekodiranog oblika reprezentacije rešetke jednak $1 -$ `to_play`. Dakle, ulazni podatak neuronske mreže tenzor je dimenzija $(2, 6, 7)$, čiji elementi na pozicijama $(0, i, j)$ $(i = 0, \ldots, 5,\ j = 0, \ldots, 6)$ predstavljaju diskove igrača koji u danom stanju povlači potez (jednaki su 1 ako njegov disk zauzima odgovarajuće mjesto u rešetci, a 0 inače), dok mu elementi na pozicijama $(1, i, j)$ $(i = 0, \ldots, 5,\ j = 0, \ldots, 6)$ na isti način predstavljaju diskove protivnika igrača koji u stanju koje tenzor predstavlja povlači potez. Takva reprezentacija stanja omogućuje invarijantnost na igrača koji u stanju povlači potez, odnosno omogućava da neuronska mreža stanje analizira uvijek iz perspektive igrača koji u njemu odlučuje o akciji.

Upotrebljena arhitektura rezidualne konvolucijske neuronske mreže prikazana je na slici 3.2. Pritom je rezidualni blok na slici uokviren pravokutnikom isprekidanog ruba jedan od skrivenih rezidualnih slojeva, kojih je ukupno četiri. Dakle, slika je ustvari komprimirani prikaz arhitekture — na mjestu uokvirenog bloka ustvari se nalaze četiri takva bloka povezana serijski. Također, napomenimo da se u našoj implementaciji kao argument `data_format` konstruktoru Keras klase `Conv2D`, koja predstavlja 2D konvolucijski sloj neuronske mreže, uvijek prosljeđuje `channels_first`, što znači da su ulazni podaci dimenzija `(batch_size, channels, height, width)`, gdje `channels` označava broj kanala.

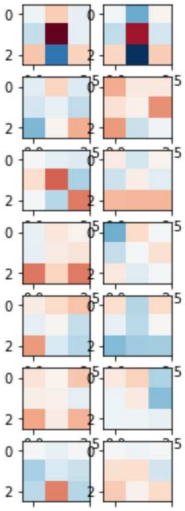
Po uzoru na funkciju `viewLayers` iz članka [8], definirana je metoda `view_layers` klase `GeneralModel`, koja omogućava vizualizaciju filtera konvolucijskih slojeva, ali i parametara ostalih slojeva neuronske mreže koju dana instanca klase predstavlja. Primjer dijela ispisa dobivenog njenim pozivom za prvi konvolucijski sloj jedne verzije naše neuronske mreže vidljiv je na slici 3.3. Kao što je prikazano na slici 3.2, jedan ulazni podatak prvog konvolucijskog sloja dimenzija je $(2, 6, 7)$, odnosno, radi se o mapi značajki (engl. *feature map*) s dva kanala. Broj 3D filtera za dani je konvolucijski sloj jednak 64, odnosno, s obzirom na to da se ulazna mapa značajki sastoji od dva kanala, dani konvolucijski sloj čini $2 \cdot 64 = 128$ 2D filtera dimenzija 3×3. Dakle, na slici 3.3 prikazano je prvih 14 2D filtera prvog konvolucijskog sloja dane neuronske mreže. Odnosno, vizualizirane su težine koje čine odgovarajuće filtere, pri čemu intenzivnija crvena boja predstavlja negativnu težinu veće apsolutne vrijednosti, a intenzivnija plava boja predstavlja veću težinu.

[1] NumPy je biblioteka za programski jezik Python koja daje podršku za velika, višedimenzionalna polja i matrice, koja uključuje veliku kolekciju matematičkih funkcija za rad s takvim poljima na visokoj razini.

Slika 3.2: Korištena rezidualna konvolucijska neuronska mreža.



Slika 3.3: Prikaz filtera jednog konvolucijskog sloja neuronske mreže.



3.5 Proces učenja

Proces učenja započinjemo i ostvarujemo pokretanjem skripte `main.py`. Skripta `main.py` najprije instancira dva objekta klase `ResCNN`: `current_NN` i `best_NN`, čiji su parametri inicijalizirani nasumično, osim ako nije navedeno drukčije, odnosno, osim ako učenje ustvari ne pokrećemo ispočetka, već nastavljamo učiti na temelju, u nekom prethodnom pokretanju, već naučenog modela. Potom skripta instancira pripadajuće objekte klase `Agent` — naime, klasa `Agent` predstavlja agenta strojnog učenja podrškom te pomoću metode `act` implementira njegovo donošenje odluke o akciji koju će poduzeti u danome stanju. Instancirani objekti klase `Agent` pohranjuju se u varijable `current_player` i `best_player`, pri čemu se njihovim konstruktorima prosljeđuju `ResCNN` objekti `current_NN`, odnosno `best_NN`, redom — proslijeđene neuronske mreže svojim izlazima za pojedino stanje kao ulaz određuju kontrolne strategije pojedinih agenata.

Učenje se odvija unutar petlje skripte `main.py` — jedan prolazak kroz petlju odgovara jednoj iteraciji AlphaZero algoritma. Svaka iteracija započinje igranjem određenog broja igara agenta predstavljenog objektom `best_player` samog protiv sebe, a što se ostvaruje pozivom funkcije `play_matches`. Funkcija `play_matches` implementira izvršavanje zadanog broja epizoda strojnog učenja podrškom — u sklopu jedne epizode, počevši od početnog stanja, sve dok se ne dosegne završno stanje, nad proslijeđenim objektom klase `Agent` poziva funkciju `act`, koja, kao što smo već ranije naveli, implementira agentovo donošenje odluke o akciji koju će u danome stanju poduzeti.

Metoda `act` klase `Agent` kao povratnu vrijednost vraća, među ostalim, NumPy polje vjerojatnosti odabira pojedinih akcija proizašlih iz pretraživanja stabla Monte Carlo metodom, na temelju kojih vraća i odabranu akciju, odnosno njen indeks. Nakon pojedinog poziva metode `act` u aktualnoj epizodi, stanje u kojemu se u danom koraku poduzima akcija, zajedno s poljem vjerojatnosti proizašlih iz pretraživanja stabla Monte Carlo metodom, pohranjuju se kako bi na kraju iteracije potencijalno poslužili kao dijelovi primjera za nadzirano učenje, to jest treniranje neuronske mreže. Pritom se, osim danog stanja i njemu pripadnih vjerojatnosti, zajedno s istim vjerojatnostima pohranjuje i stanje kojemu odgovara rešetka osnosimetrična s obzirom na vertikalnu os simetrije. Drugim riječima, iako smo u odjeljku 2.1 naveli da AlphaZero ne pretpostavlja nikakve simetrije, s obzirom na to da implementiramo algoritam iz klase AlphaZero algoritama za igru koja je osnosimetrična s obzirom na vertikalnu os simetrije, ovdje tu simetriju iskorištavamo. Pohranjivanje primjera za učenje realiziramo pomoću klase `Memory`, čiji su atributi sljedeći:

- `st_buffer`: *deque* za kratkoročnu pohranu primjera za učenje; u njemu čuvamo primjere za učenje (ili njihove dijelove) tijekom jedne epizode jedne iteracije, a po njenom završetku, pohranjujemo ih u `lt_buffer` te `st_buffer` praznimo. Elementi *deque*a rječnici su, koji sadrže barem dva, a najviše tri ključa. Naime, mogući su ključevi `state`, `actionProbs` i `value`, a njima su pripadajuće vrijednosti stanje (objekt klase `GameState`), polje vjerojatnosti odabira pojedine akcije u danom stanju, odnosno ishod igre pridružen trenutku kojemu odgovara dano stanje, redom. Prije kraja epizode, svaki rječnik sadrži samo ključeve `state` i `actionProbs` te pripadne vrijednosti, a po završetku jedne epizode, dobiva i ključ `value` te njemu pripadnu vrijednost;

- `lt_buffer`: *deque* za dugoročnu pohranu primjera za učenje; kao što smo već naveli, po završetku pojedine epizode, svi primjeri pohranjeni u `st_buffer` pohranjuju se u `lt_buffer`, nakon čega se *deque* `st_buffer` prazni. Primjeri za učenje tijekom treniranja uzimaju se iz *deque*a `lt_buffer`;

- `memory_size`: kapacitet memorije, odnosno maksimalna duljina *deque*ova pohranjenih u atributima `lt_buffer` i `st_buffer`.

Dakle, jednom kada u sklopu jedne epizode dosegnemo završno stanje, svim se elementima *deque*a `st_buffer` (rječnicima) dodaje ključ `value` s ishodom igre pridruženom trenutku kojemu odgovara dano stanje kao pripadnom vrijednosti. Tako dobiveni rječnici predstavljaju primjere za učenje koji se koriste tijekom treniranja neuronske mreže.

Kapacitet memorije definira se u ranije spomenutoj datoteci `config.py`. Novi elementi dodaju se u `st_buffer` ili `lt_buffer` pomoću metode *append*, koja element koji želimo ubaciti dodaje na desni kraj *deque*a. Ako je maksimalna duljina pojedinog *deque*a već dosegnuta, prilikom ubacivanja novog elementa izbacuju se elementi sa suprotnog kraja.

Dakle, postižemo da se u pojedinom spremniku uvijek nalazi „najnovijih" `memory_size` primjera za učenje (ili njihovih dijelova), odnosno `memory_size` onih najkasnije proizvedenih. Treniranje neuronske mreže ne pokrećemo dok duljina spremnika `lt_buffer` ne dosegne svoju maksimalnu duljinu — tek kad se to jednom dogodi, započinjemo treniranje na temelju primjera za učenje stvorenih u tekućoj iteraciji, a možda i u nekim prethodnima; iteracija iz kojih koristimo primjere za učenje onoliko je koliko ih je bilo potrebno da `lt_buffer` dosegne maksimalnu duljinu.

Ako je po završetku igranja igara agenta `best_player` samog protiv sebe duljina spremnika `lt_buffer` jednaka maksimalnoj duljini, obavljamo treniranje neuronske mreže. Optimizacijski algoritam koji koristimo je *mini-batch* stohastički gradijentni spust s *momentum*om, koji je implementiran Keras klasom `SGD`. Stohastički gradijentni spust može se smatrati stohastičkom aproksimacijom optimizacijskog algoritma gradijentnog spusta budući da, umjesto da računa gradijent na temelju cijelog skupa podataka, gradijent procjenjuje njegovim računanjem na temelju odabranog podskupa skupa podataka. *Momentum* je metoda koja se koristi radi ubrzanja konvergencije, a stohastički gradijentni spust s momentumom jedan je od najpopularnijih optimizacijskih algoritama, koji se koristi za treniranje mnogih *state-of-the-art* modela.

U odjeljku 2.1 naveli smo kako je jedna od razlika između algoritama AlphaZero i AlphaGo Zero ta što u pojedinoj iteraciji algoritma AlphaGo Zero igranje igara samog protiv sebe obavlja agent koji se pokazao najuspješnijim od svih generiranih u dotadašnjim iteracijama (odnosno, čija se verzija neuronske mreže pokazala najuspješnijom) te da se novim najuspješnijim agentom (novom najuspješnijom neuronskom mrežom) proglašava novi (dobiven nakon treniranja neuronske mreže u sklopu dane iteracije) ako pobijedi dotad najboljeg u barem 55% igara, dok AlphaZero održava jedinstvenu neuronsku mrežu koju ažurira kontinuirano. U našoj implementaciji, radi boljeg praćenja napretka tijekom učenja, u tom se kontekstu priklanjamo AlphaGo Zero pristupu. Naime, po završetku treniranja neuronske mreže u sklopu pojedine iteracije, uspjeh novodobivene neuronske mreže (odnosno, pripadnog agenta) provjeravamo igranjem određenog broja igara u kojima su igrači do tad najuspješniji agent (onaj predstavljen objektom `best_player`) te onaj kojemu je pridružena neuronska mreža dobivena posljednjim treniranjem (koji je predstavljen objektom `current_player`) — ako je omjer broja pobjeda agenta `current_player` i agenta `best_player` veći od 1.25, najsupješnijim agentom proglašavamo agenta predstavljenog objektom `current_player`, odnosno, kao novu neuronsku mrežu agenta `best_player` postavljamo onu dobivenu posljednjim treniranjem (težinama neuronske mreže predstavljene objektom `best_NN` proglašavamo težine one dane objektom `current_NN`).

Proces učenja ubrzan je izvršavanjem operacija obavljanih u sklopu nadziranog učenja na grafičkom procesoru (engl. *graphics processing unit*, GPU). Naime, napisane su skripte uglavnom izvršavane na računalnom klasteru Isabella. Isabella se sastoji od 135 računalnih čvorova, koji sadrže 3100 procesorskih jezgri i 12 grafičkih procesora, te kao zajednički

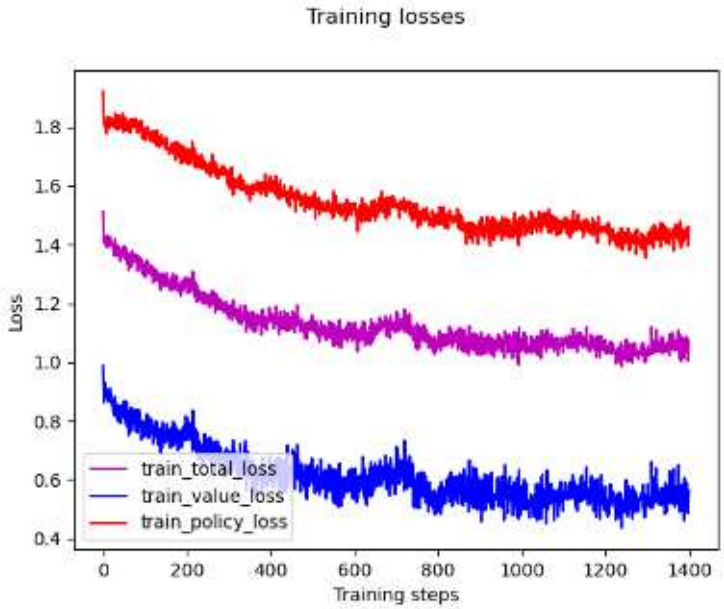
resurs svih znanstvenika u Hrvatskoj omogućava korištenje značajnih računalnih resursa pri zahtjevnim obradama podataka u sklopu znanstveno-istraživačkih projekata. Na Isabelli su dostupna tri poslužitelja Dell EMC PowerEdge C4140 s po četiri grafička procesora NVIDIA Tesla V100-SXM2-16GB, dva procesora Intel Xeon Silver 4114 s ukupno 20 procesorskih jezgri, 384 GB radne memorije i lokalnim spremištem od 3.2 TB NVMe SSD diska. Svaki je proces bio dodijeljen jednom grafičkom procesoru jednog Isabellinog čvora. Osim na Isabelli, skripte su izvršavane i na serveru pascal.math.hr Matematičkog odsjeka Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu.

3.6 Testiranje i rezultati

Tijekom učenja, odnosno izvršavanja skripte `main.py`, opisane u odjeljku 3.5, težine modela predstavljenog objektom `best_NN` (također objašnjenog u odjeljku 3.5) pohranjuju se svaki put kad težine neuronske mreže dobivene treniranjem unutar pojedine iteracije zamijene težine do tad najuspješnije neuronske mreže. U skladu s navedenim, kao rezultat procesa učenja dobivamo niz modela (i pridruženih agenata) stvorenih u različitim trenucima učenja, koji su kao takvi različite uspješnosti u igranju igre Četiri u nizu. Očekujemo da su agenti čije su neuronske mreže nastale u kasnijim trenucima učenja uspješniji od onih kojima odgovaraju ranije neuronske mreže — vrijedi li to u praksi, provjeravamo na više različitih načina, koje opisujemo u nastavku. Proces učenja započet je i obavljen više puta, a u ovom odjeljku predstavljamo rezultate dobivene u jednom od mnogo pokretanja napisanih programa. U svakom je pokretanju korištena različita kombinacija hiperparametara algoritma te je stoga svako polučilo jedinstvene rezultate. Međutim, dok se detalji razlikuju, trend učenja vidljiv je gotovo u svima, a rezultati predstavljeni u ovom odjeljku odabrani su kao jedni od onih koji ga dobro dočaravaju.

Program je izvršavan nekoliko dana (oko 100 sati), a tijekom izvršavanja obavljeno je 150 iteracija algoritma. Na slici 3.4 možemo vidjeti takozvane krivulje učenja (engl. *learning curve*), odnosno grafove koji prikazuju vrijednosti mjera greški izvedenih iz pojedinih funkcija gubitaka, izračunatih za podatke iz skupa podataka za učenje, u ovisnosti o epohama treniranja. Krivulja učenja dobivena na temelju skupa podataka za učenje daje nam informaciju o napretku u učenju.

Pojasnimo pojedine krivulje vidljive na slici 3.4. Legenda u donjem lijevom kutu slike govori nam da je crvenom bojom prikazan *train policy loss*, plavom *train value loss*, a ljubičastom *train total loss*. Mjera greške koju prikazuje krivulja crvene boje ona je izvedena iz funkcije gubitka *cross-entropy*, opisane u odjeljku 2.6. S druge strane, krivulja plave boje prikazuje mjeru greške izvedenu iz srednje kvadratne pogreške, također objašnjene u odjeljku 2.6. Radi objašnjenja krivulje ljubičaste boje, prisjetimo se funkcije gubitka dane jednadžbom (2.13). Ljubičasta krivulja je krivulja učenja vezana uz mjeru greške izvedenu iz funkcije gubitka koja je, uz jednaka objašnjenja pojedinih simbola kao



Slika 3.4: Krivulje učenja dobivene na temelju skupa podataka za učenje.

i u slučaju jednadžbe (2.13), jednaka

$$L((\boldsymbol{\pi}, z), (\mathbf{p}, v)) = \frac{1}{2}(z - v)^2 - \frac{1}{2}\boldsymbol{\pi}^T \log_2 \mathbf{p} + c\|\theta\|^2,$$

odnosno funkcije gubitka koja se od one dane jednadžbom (2.13) razlikuje u tome što su pojedini članovi (osim regularizacijskog člana) pomnoženi faktorom $\frac{1}{2}$. Kao što možemo vidjeti, nijedna komponenta takve mjere greške nema jasnu tendenciju pada (ni rasta) u ovisnosti o epohama treniranja. Uzrok je toga to što se u svakoj iteraciji nakon čijeg je igranja igara najboljeg agenta samog protiv sebe duljina spremnika `lt_buffer` dovoljne duljine (što smatramo „dovoljnom" duljinom, objasnili smo u odjeljku 3.5) mijenja neuronska mreža koju treniramo, što znači da stalno iznova treniramo već istrenirani model.

Napredak tijekom učenja praćen je mjerenjem uspješnosti pojedinih agenata u igrama protiv vrlo jednostavnih algoritama za igranje igre Četiri u nizu. S obzirom na to da se radi o trivijalnim algoritmima, nadmoć našeg agenta naspram onog čiji odabir akcija diktira neki od danih algoritama sama po sebi nije dovoljna da bismo naš algoritam proglasili uspješnim. Međutim, uspješnost naših agenata te njezino povećanje kroz iteracije ipak nam može biti indikator ispravnosti programskog koda i tendencije učenja.

Jednostavni algoritmi koji su nam poslužili za usporedbu sljedeći su:

- Nasumični (engl. *random*) algoritam: algoritam koji definira agenta koji u svakom stanju o akciji odlučuje potpuno slučajno. Za dano stanje određuje koje su akcije u

njemu legalne te nasumično odabire jednu od njih;

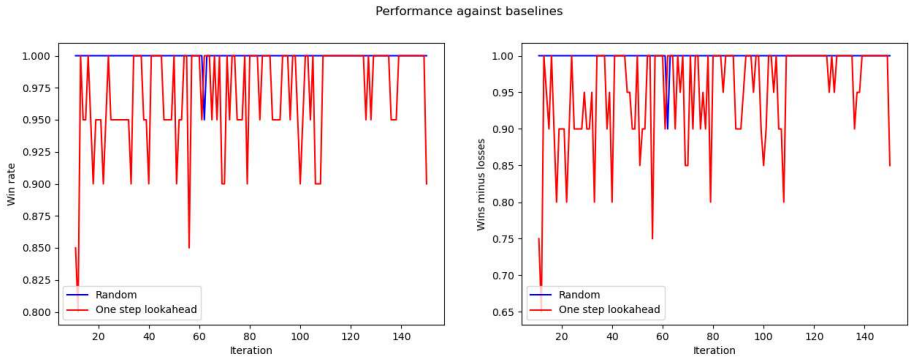
- Algoritam jednog koraka gledanja unaprijed (engl. *one step lookahead*): algoritam koji definira agenta koji o akciji u danome stanju odlučuje tako da uvijek odabere onu koja mu u sljedećem stanju donosi pobjedu ako takva postoji; ako takva ne postoji, a postoji neka koja protivniku onemogućuje da sljedećim potezom ostvari niz od četiri diska svoje boje, odabire nju, dok u protivnom akciju odabire nasumično. Dakle, takav agent uvijek nastoji odabrati akciju koja u sljedećem stanju donosi pobjedu ili, ako takve nema, onu koja onemogućuje prelazak u stanje u kojemu bi protivnik mogao poduzeti akciju koja njemu osigurava pobjedu (a igraču čiji je protivnik pobijedio gubitak).

Nakon svakog treniranja neuronske mreže i eventualnog ažuriranja težina neuronske mreže predstavljene objektom `best_NN` tijekom izvršavanja programa, održavamo „turnir" koji se sastoji od 20 igara između aktualnog najboljeg agenta, odnosno agenta danog objektom `best_player`, i svakog od agenata vezanih uz opisane jednostavne algoritme. Definiramo klase `RandomAgent`, odnosno `OneStepLookaheadAgent`, koje implementiraju nasumični algoritam, odnosno algoritam jednog koraka gledanja unaprijed, redom, odnosno predstavljaju agente čije odabire akcija oni određuju. Uspješnost pojedinih agenata mjerimo pomoću sljedeće dvije metrike:

- omjer njegovih pobjeda i ukupnog broja odigranih igara;
- normalizirana razlika broja njegovih pobjeda i protivnikovih pobjeda (razlika navedena dva broja podijeljena ukupnim brojem odigranih igara).

Razlog zašto osim vrijednosti prve metrike pratimo i one druge navedene sljedeći je. Naime, prva navedena mjera daje nam samo informaciju o broju pobjeda agenta čiju uspješnost mjerimo — igra Četiri u nizu igra je koja može završiti neriješeno, a iz vrijednosti prve mjere ne možemo zaključiti koliki je broj preostalih igara završio njegovim gubitkom (protivnikovom pobjedom), a koliko ih je završilo neriješeno. Kako bismo dobili informaciju i o tim brojevima, uvodimo drugu metriku — ona nam omogućava da, uz poznatu vrijednost prve, za pojedini turnir odredimo i u koliko je igara protivnik pobijedio, a time i koliko je onih neriješenih. Vrijednosti pojedinih mjera u ovisnosti o iteracijama vidljive su na grafovima sa slike 3.5. Pritom na lijevoj strani možemo vidjeti vrijednosti prve metrike, a na desnoj druge metrike. Plavom su bojom označeni grafovi za igre protiv agenta koji igra nasumično, a crvenom oni za igre protiv agenta koji igra uz jedan korak gledanja unaprijed.

Na slici 3.5 možemo vidjeti da nakon svih iteracija algoritma osim jedne naši agenti pobjeđuju agenta koji igra nasumično u svim odigranim igrama. Također, dok na početku procesa učenja u prosjeku postižu nešto lošije rezultate u igrama protiv agenta s jednim



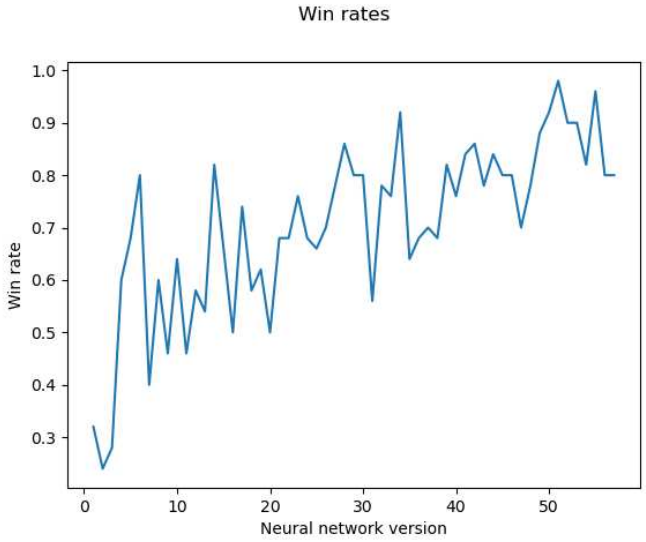
Slika 3.5: Usporedba uspješnosti agenata u pojedinim iteracijama treniranja u igrama protiv jednostavnih algoritama.

korakom gledanja unaprijed, na kraju ga gotovo nakon svake iteracije pobjeđuju u svim igrama.

Ostvaruje li se zbilja izvršavanjem implementacije AlphaZero algoritma napredak u igranju igre Četiri u nizu provjeravamo i mjerenjem uspješnosti pojedinih verzija agenata (odnosno, pripadnih neuronskih mreža) u igrama protiv nešto sofisticiranijeg algoritma za igranje igre Četiri u nizu — minimaks algoritma, čiju implementaciju preuzimamo s web stranice [2]. Za minimaks algoritam kojemu odgovara implementacija na navedenoj web stranici možemo reći da pripada istoj klasi algoritama kao i algoritam jednog koraka gledanja unaprijed opisan ranije u odjeljku. Naime, korištena je implementacija ustvari implementacija općenitog algoritma gledanja unaprijed N koraka (engl. *N-step lookahead*), koja varijabli N pridružuje vrijednost 3. Dakle, radi se o algoritmu koji obavlja pretraživanje stabla igre do dubine 3. On omogućava agentovo razmatranje svih mogućih stanja u koje je prelazak posljedica neke njegove akcije, stanja u koje vodi bilo koja akcija njegovog protivnika poduzeta u sljedećem stanju, kao i stanja u koje je prelazak posljedica odabira bilo koje akcije agenta u nekom od takvih sljedećih stanja. Osnovna je ideja minimaks algoritma odabir akcije koja maksimira nagradu agentu koji o akciji odlučuje, uz pretpostavku da će protivnik odabirati akcije koje će tu nagradu nastojati minimizirati. Dakle, agent i njegov protivnik imaju suprotne ciljeve te se pretpostavlja da protivnik igra optimalno, u smislu minimizacije agentove nagrade.

Kako bismo provjerili napredak u uspješnosti naših agenata naspram agenta koji implementira opisani minimaks algoritam, održavamo „turnir" u kojemu svaka pojedina verzija agenta (odnosno, svaka pojedina verzija neuronske mreže) igra 50 igara protiv minimaks agenta. Uspješnost ponovno mjerimo omjerom broja pobjeda pojedinog našeg agenta i ukupnog broja odigranih igara. Vrijednosti takve mjere uspješnosti u ovisnosti o verziji neuronske mreže mogu se vidjeti na grafu sa slike 3.6.

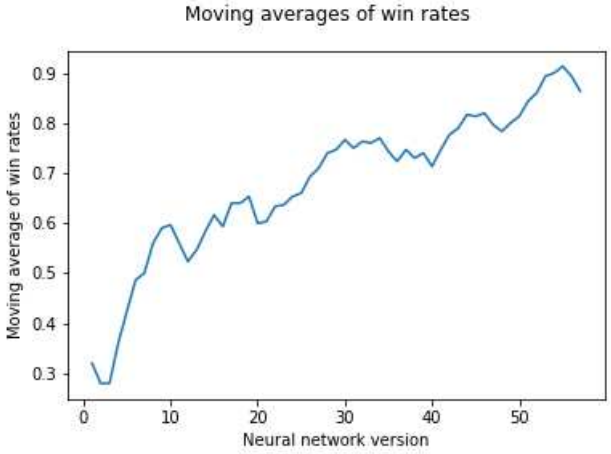
Primijetimo da se na osi apscisa nalaze verzije neuronskih mreža umjesto iteracija algo-



Slika 3.6: Uspješnost pojedinih agenata dobivenih izvršavanjem implementacije AlphaZero algoritma u igrama protiv minimaks agenta.

ritma, koje su bile prisutne na x-osima na slici 3.5. Uzrok je toj razlici to što je uspješnost agenata u igrama protiv jednostavnog nasumičnog algoritma te algoritma jednog koraka gledanja unaprijed mjerena tijekom procesa učenja (nakon pojedine iteracije algoritma, kao što je opisano ranije u aktualnom odjeljku), dok je usporedba sposobnosti naših agenata i onog koji implementira opisani minimaks algoritam provedena nakon što je proces učenja, koji je polučio različite verzije neuronskih mreža čiju uspješnost ispitujemo, završen. Kako bismo bolje vizualizirali i naglasili dugoročne trendove u promjenama vrijednosti omjera pobjeda te „izgladili" kratkoročne fluktuacije, na slici 3.7 prikazujemo takozvane pokretne prosjeke (engl. *moving average*) udjela pobjeda. Drugim riječima, za svaku verziju neuronske mrežu (osim prvih pet) računamo srednju vrijednost mjera uspješnosti pridruženih njoj te pet prethodnih verzija te joj, umjesto omjera broja njenih pobjeda i ukupnog broja odigranih igara, pridružujemo izračunatu srednju vrijednost. S obzirom na to da je za prvih pet verzija nemoguće izračunati takvu srednju vrijednost, njima pridružujemo srednje vrijednosti njihovog udjela pobjeda te svih neuronskih mreža koje im prethode.

Na slici 3.7 možemo vidjeti da uistinu postoji trend povećanja udjela pobjeda pojedinih agenata, odnosno da agenti u prosjeku zaista postaju sve uspješniji u igrama protiv danog minimaks agenta s povećanjem odmaka njihovog nastanka od početka učenja. Na slici 3.6 možemo vidjeti da najuspješniji agenti, kojima odgovaraju neke od najkasnijih verzija neuronskih mreža, pobjeđuju gotovo u svakoj odigranoj igri. Također, možemo uočiti veliku razliku između uspješnosti agenata kojima odgovaraju prve verzije neuronskih mreža te onih kojima odgovaraju posljednje: dok su omjeri pobjeda prva tri agenta (onih kojima od-



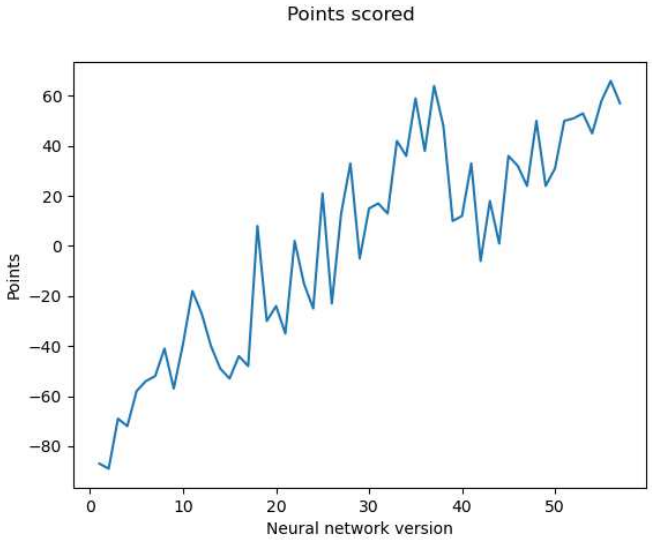
Slika 3.7: Pokretni prosjeci uspješnosti pojedinih agenata dobivenih izvršavanjem implementacije AlphaZero algoritma u igrama protiv minimaks agenta.

govaraju prve tri verzije neuronske mreže) i ukupnog broja igara manji ili približno jednaki 0.3, ti su omjeri za agente kojima odgovaraju neke od najkasnijih verzija neuronskih mreža ponekad veći i od 0.9.

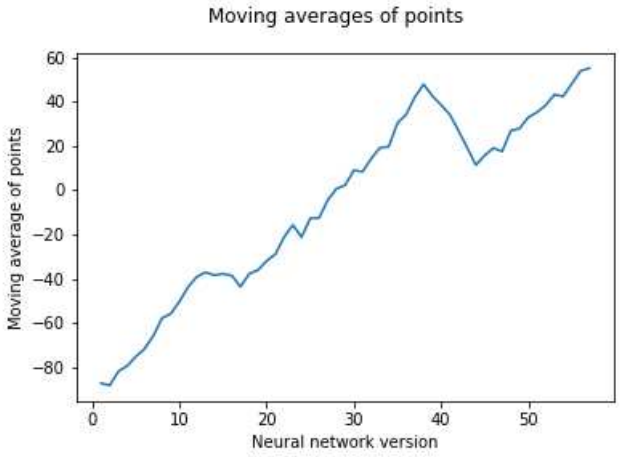
Osim na već navedene načine, napredak tijekom učenja utvrđujemo i promatranjem rezultata igara pojedinih verzija agenata (odnosno, pripadnih neuronskih mreža) protiv ostalih dobivenih verzija. Ponovno održavamo „turnir" u kojemu svaki agent dobiven tijekom izvršavanja programa igra 2 igre protiv svakog drugog od dobivenih agenata. Rezultate pratimo pomoću tablice čiji su broj redaka i broj stupaca jednaki broju različitih verzija neuronskih mreža. Primjerice, rezultate igara između agenta i i agenta j bilježimo tako da u ćeliju na presjeku i-tog retka i j-tog stupca upišemo razliku broja pobjeda i-tog agenta i broja pobjeda j-tog agenta. Za pojedinog agenta, sumiramo vrijednosti u svim ćelijama pripadajućeg retka te dobivenu sumu nazivamo brojem bodova danog agenta. Brojevi bodova za pojedine agente, odnosno njihovo kretanje u ovisnosti o verziji agenta, vidljivi su na slici 3.8.

Ponovno, radi boljeg uočavanja dugoročnih trendova te „izglađivanja" lokalnih fluktuacija, na slici 3.9 prikazujemo pokretne prosjeke brojeva bodova pojedinih agenata (odnosno, njima pripadnih neuronskih mreža). Ponovno za pojedinu verziju neuronske mreže (osim prvih pet) računamo srednju vrijednost njenog broja bodova te brojeva bodova prethodnih pet neuronskih mreža (odnosno, pripadnih agenata). Za prvih pet, slično kao i na slici 3.7, računamo srednje vrijednosti bodova pojedine neuronske mreže te svih koje joj prethode.

Na temelju slika 3.8 i 3.9 možemo zaključiti da postoji trend rasta broja ostvarenih bodova s povećanjem rednog broja verzije neuronske mreže. S obzirom na to da svaka



Slika 3.8: Brojevi bodova pojedinih agenata u igrama protiv svih drugih.



Slika 3.9: Pokretni prosjeci brojeva bodova pojedinih agenata u igrama protiv svih drugih.

verzija od ukupno 57 igra protiv svake druge te da joj za pojedinu igru mogu biti dodijeljena najviše 2 boda, a najmanje -2, maksimalan je broj bodova koji je moguće ostvariti 112, a minimalan -112. Na navedenim slikama možemo uočiti da prve dvije verzije ostvaruju manje od -80 bodova, dok posljednje dvije ostvaruju oko 60. Dakle, razlika u uspješnosti u kontekstu ove metrike može se kvantificirati vrijednošću od oko 140 bodova.

Nadalje, uspješnost pojedinih agenata (onih dobivenih pokretanjem AlphaZero implementacije te onih koji implementiraju druge algoritme) uspoređujemo ispunjavanjem 7×7

tablice na sljedeći način. Broju 1 pridružujemo nasumičan algoritam (odnosno, agenta kojemu on odgovara), brojevima 2–4 varijante minimaks agenata s jednim, odnosno tri ili četiri koraka gledanja unaprijed, redom, broju 5 AlphaZero agenta kojemu odgovara prva verzija neuronske mreže, broju 6 onog kojemu odgovara dvadeseta verzija neuronske mreže, a broju 7 agenta s pridruženom posljednjom verzijom neuronske mreže. Obavljamo igranje 50 igara za svaki par agenata (i, j) (gdje su $i, j \in \{1, \ldots, 7\}$ i $i \neq j$) te u ćeliju na presjeku i-tog retka te j-tog stupca upisujemo broj pobjeda agenta s pridruženim brojem i. Podaci dobiveni na taj način vidljivi su u tablici 3.1.

Tablica 3.1: Tablica ostvarenih brojeva pobjeda pojedinih agenata u igrama protiv svih ostalih.

	Nasumični	1 korak	3 koraka	4 koraka	1. NN	20. NN	57. NN
Nasumični		0	1	0	1	0	0
1 korak	50		8	1	18	3	1
3 koraka	49	41		7	36	10	3
4 koraka	50	47	35		34	12	5
1. NN	49	30	9	8		8	1
20. NN	50	47	35	32	38		21
57. NN	50	49	44	42	48	29	

Podaci iz tablice 3.1 pokazuju nam da, dok agenta s pridruženom prvom verzijom neuronske mreže nadmoćno pobjeđuju i minimaks agent s tri koraka gledanja unaprijed i onaj s četiri koraka gledanja unaprijed, agent kojemu pripada posljednja verzija neuronske mreže minimaks agenta s četiri koraka gledanja unaprijed (koji onog s tri koraka pobjeđuje u 35 igara, a onog s jednim korakom gledanja unaprijed pobjeđuje u čak 47 igara) pobjeđuje u 42 igre (dok njega dani agent pobjeđuje samo 5 puta). Također, ponovno možemo uočiti sličan trend kakav su pokazale slike 3.8 i 3.9, odnosno da se agenti kojima su pridružene kasnije verzije neuronskih mreža pokazuju uspješnijima u igrama protiv agenata s pripadnim ranijim verzijama.

Konačno, utvrđujemo uspješnost pojedinih agenata (to jest, verzija neuronskih mreža) te radimo usporedbu njihovih vještina igranja osobnim igranjem s njima kao protivnicima. Igranje čovjeka protiv dobivenih agenata omogućeno je na dva načina: jedna implementacija od čovjeka traži da putem tipkovnice upisuje indekse stupaca u koje želi ubaciti disk svoje boje, a druga obuhvaća grafičko sučelje te će biti detaljnije opisana u odjeljku 3.7. U ovom odjeljku izlažemo ispise tijekom izvršavanja implementacije igranja igara protiv čovjeka bez grafičkog sučelja.

Promotrimo najprije tijek igre čovjeka protiv agenta kojemu pripada prva verzija neuronske mreže. Pritom je čovjek onaj igrač koji igra prvi, odnosno čiji su diskovi označeni simbolom `x`.

```
Enter the column you wish to drop the disk into:  3
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
----------------------------------------
Action: 1
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'o', '-', 'x', '-', '-', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  3
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', 'o', '-', 'x', '-', '-', '-']
----------------------------------------
Action: 0
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['o', 'o', '-', 'x', '-', '-', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  1
Action: 1
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'x', '-', 'x', '-', '-', '-']
['o', 'o', '-', 'x', '-', '-', '-']
----------------------------------------
Action: 0
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['o', 'x', '-', 'x', '-', '-', '-']
['o', 'o', '-', 'x', '-', '-', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  4
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['o', 'x', '-', 'x', '-', '-', '-']
['o', 'o', '-', 'x', 'x', '-', '-']
----------------------------------------
```

```
Action: 1
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'o', '-', '-', '-', '-', '-']
['o', 'x', '-', 'x', '-', '-', '-']
['o', 'o', '-', 'x', 'x', '-', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  4
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'o', '-', '-', '-', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['o', 'o', '-', 'x', 'x', '-', '-']
----------------------------------------
Action: 5
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'o', '-', '-', '-', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  3
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'o', '-', 'x', '-', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', '-', '-', '-']
['-', 'o', '-', 'x', '-', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  4
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', '-', '-', '-']
['-', 'o', '-', 'x', 'x', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  5
```

```
Action: 5
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', '-', '-']
['o', 'x', '-', 'x', 'x', 'x', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Action: 1
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'o', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', '-', '-']
['o', 'x', '-', 'x', 'x', 'x', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  5
Action: 5
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', 'o', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', 'x', '-']
['o', 'x', '-', 'x', 'x', 'x', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', 'o', '-', '-']
['-', 'o', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', 'x', '-']
['o', 'x', '-', 'x', 'x', 'x', '-']
['o', 'o', '-', 'x', 'x', 'o', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  2
Action: 2
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', 'o', '-', '-']
['-', 'o', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', 'x', '-']
['o', 'x', '-', 'x', 'x', 'x', '-']
['o', 'o', 'x', 'x', 'x', 'o', '-']
----------------------------------------
Action: 4
['-', '-', '-', '-', 'o', '-', '-']
['-', '-', '-', '-', 'o', '-', '-']
['-', 'o', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', 'x', '-']
['o', 'x', '-', 'x', 'x', 'x', '-']
['o', 'o', 'x', 'x', 'x', 'o', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  2
Action: 2
['-', '-', '-', '-', 'o', '-', '-']
['-', '-', '-', '-', 'o', '-', '-']
['-', 'o', '-', 'o', 'o', '-', '-']
['-', 'o', '-', 'x', 'x', 'x', '-']
['o', 'x', 'x', 'x', 'x', 'x', '-']
['o', 'o', 'x', 'x', 'x', 'o', '-']
----------------------------------------
```

Kao što možemo vidjeti iz navedenog ispisa, nije teško pobijediti danog agenta — naime, jednostavno je dovesti igru u stanje u kojemu bilo koja agentova akcija vodi u stanje u kojemu je čovjeku na raspolaganju akcija koja mu osigurava pobjedu. Također, dok bismo od čovjeka možda očekivali da mu se potez u stanju kojemu odgovara treća rešetka od kraja sastoji od ubacivanja diska u treći stupac rešetke (i time onemogućavanja ostvarivanja niza od četiri protivnikova diska u drugom retku odozdo), agent disk ubacuje u peti stupac rešetke te time omogućava čovjeku da svojim sljedećim potezom pobijedi u igri.

S druge strane, promotrimo primjer igre čovjeka protiv agenta kojemu je pridružena posljednja verzija neuronske mreže dobivena tijekom izvršavanja napisanih programa. Ovaj je put računalo (agent dobiven izvršavanjem implementacije AlphaZero algoritma) igrač koji igra prvi, odnosno njegovi su diskovi označeni simbolom `x`.

```
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  4
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
----------------------------------------
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  3
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
----------------------------------------
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'o', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
----------------------------------------
Enter the column you wish to drop the disk into:  4
```

```
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'o', '-', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
---------------------------------------
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
---------------------------------------
Enter the column you wish to drop the disk into:  3
Action: 3
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', '-', '-', '-']
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
---------------------------------------
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', '-', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
---------------------------------------
Enter the column you wish to drop the disk into:  4
Action: 4
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
---------------------------------------
Action: 5
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', '-', '-']
['-', '-', '-', 'x', 'o', 'x', '-']
---------------------------------------
Enter the column you wish to drop the disk into:  5
Action: 5
['-', '-', '-', '-', '-', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', 'o', '-']
['-', '-', '-', 'x', 'o', 'x', '-']
---------------------------------------
Action: 3
```

```
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', 'o', '-']
['-', '-', '-', 'x', 'o', 'x', '-']
---------------------------------------
Enter the column you wish to drop the disk into:  1
Action: 1
['-', '-', '-', 'x', '-', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', 'o', '-']
['-', 'o', '-', 'x', 'o', 'x', '-']
---------------------------------------
Action: 4
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', 'o', '-']
['-', 'o', '-', 'x', 'o', 'x', '-']
---------------------------------------
Enter the column you wish to drop the disk into:  0
Action: 0
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', '-', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
---------------------------------------
Action: 1
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'x', '-', '-']
['-', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
---------------------------------------
Enter the column you wish to drop the disk into:  1
Action: 1
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', '-', '-', 'x', 'x', '-', '-']
['-', 'o', '-', 'o', 'x', '-', '-']
['-', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
---------------------------------------
Action: 1
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', 'x', '-', 'x', 'x', '-', '-']
['-', 'o', '-', 'o', 'x', '-', '-']
['-', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
---------------------------------------
Enter the column you wish to drop the disk into:  0
Action: 0
```

```
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', 'x', '-', 'x', 'x', '-', '-']
['-', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
--------------------------------------
Action: 0
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['-', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
--------------------------------------
Enter the column you wish to drop the disk into:  0
Action: 0
['-', '-', '-', 'x', 'x', '-', '-']
['-', '-', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
--------------------------------------
Action: 0
['-', '-', '-', 'x', 'x', '-', '-']
['x', '-', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
--------------------------------------
Enter the column you wish to drop the disk into:  1
Action: 1
['-', '-', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
--------------------------------------
Action: 1
['-', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
--------------------------------------
Enter the column you wish to drop the disk into:  0
Action: 0
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', '-', 'x', 'o', 'x', '-']
--------------------------------------
Action: 2
['o', 'x', '-', 'x', 'x', '-', '-']
```

```
['x', 'o', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', '-', 'x', 'o', 'o', '-']
['o', 'o', 'x', 'x', 'o', 'x', '-']
--------------------------------------
Enter the column you wish to drop the disk into:  2
Action: 2
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'x', '-', '-']
['o', 'x', 'o', 'x', 'o', 'o', '-']
['o', 'o', 'x', 'x', 'o', 'x', '-']
--------------------------------------
Action: 2
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', '-', 'o', 'o', '-', '-']
['o', 'x', '-', 'x', 'x', '-', '-']
['x', 'o', 'x', 'o', 'x', '-', '-']
['o', 'x', 'o', 'x', 'o', 'o', '-']
['o', 'o', 'x', 'x', 'o', 'x', '-']
--------------------------------------
```

Iz upravo navedenog ispisa možemo vidjeti da je agent s pridruženom posljednjom verzijom neuronske mreže značajno vještiji od onog kojemu odgovara prva verzija — naime, čovjeka koji nije osobito pažljiv i promišljen tijekom igre može jednostavno pobijediti. Možemo uočiti kako kasnija verzija razvija strategiju ranog zauzimanja srednjeg stupca, što se obično preporuča igračima igre Četiri u nizu, dok ranija takve akcije ne poduzima. Također, primijetimo da, dok je čovjek igru protiv prve verzije jednostavno mogao dovesti u stanje u kojemu bilo koja akcija protivnika vodi u stanje u kojemu jednim potezom može pobijediti, ovaj je put računalo bilo to koje je, na manje trivijalan način, igru dovelo do stanja u kojemu čovjek nije mogao izbjeći poraz — bilo koja akcija rezultirala bi stanjem u kojemu agent na raspolaganju ima akciju koja vodi u završno stanje koje je za njega pobjedničko (a, kao što vidimo u priloženom ispisu, takvu je akciju i poduzeo).

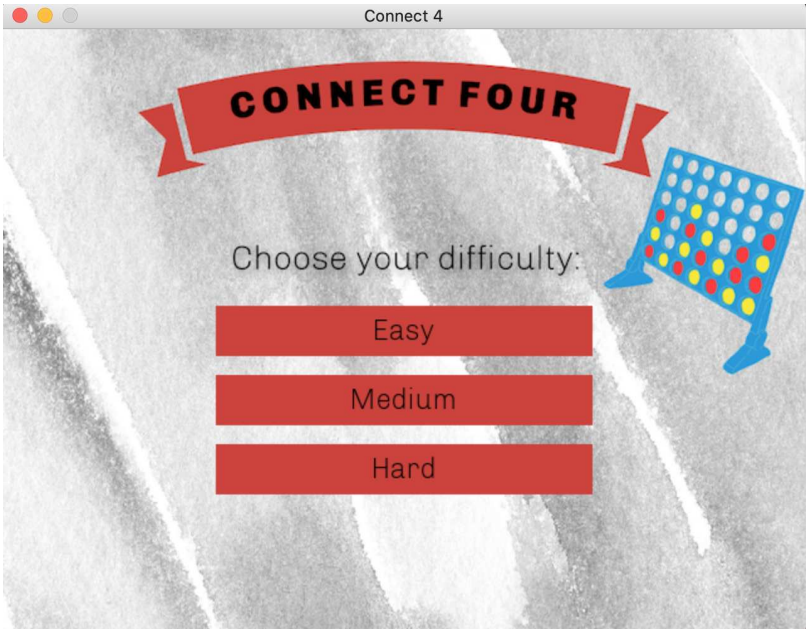
3.7 Grafičko sučelje

Radi jednostavnijeg i zabavnijeg igranja čovjeka protiv agenata dobivenih izvršavanjem naše implementacije AlphaZero algoritma, ali i svojevrsne „materijalizacije" dobivenih rezultata te stvaranja „opipljivog" konačnog proizvoda, implementiramo i grafičko sučelje za igranje igre Četiri u nizu s dobivenim agentima kao protivnicima.

Grafičko sučelje za igru implementirano je pomoću biblioteke pygame. Biblioteka pygame sastoji se od modula za pisanje video igara namijenjenih za upotrebu uz programski jezik Python te se može koristiti na raznim platformama. Igranje protiv AlphaZero agenata ostvaruje se pokretanjem skripte `connect4.py`. Skripta sve grafičke elemente implementira korištenjem pygame klasa i funkcija, a implementaciju logike računalnog

igranja igre ostvaruje korištenjem klasa opisanih u prethodnim odjeljcima. S obzirom na to da skripta `connect4.py` koristi ranije objašnjene klase koje implementiraju igru Četiri u nizu te logiku AlphaZero algoritma, uključujući pretraživanje stabla Monte Carlo metodom koje koristi izlazne vrijednosti neuronske mreže, za pokretanje skripte `connect4.py` potrebno je, uz pygame, imati instalirane biblioteke TensorFlow i Numpy. Također, programi koji uključuju rad s TensorFlow klasama i funkcijama zahtijevaju dostupnost odgovarajućeg grafičkog procesora. Dakle, skriptu `connect4.py` moguće je pokrenuti samo na računalima s grafičkim procesorima koje podržava GPU verzija TensorFlowa te s instaliranim odgovarajućim bibliotekama.

Prilikom pokretanja `connect4.py` skripte, na ekranu se prikazuje glavni izbornik prikazan na slici 3.10.

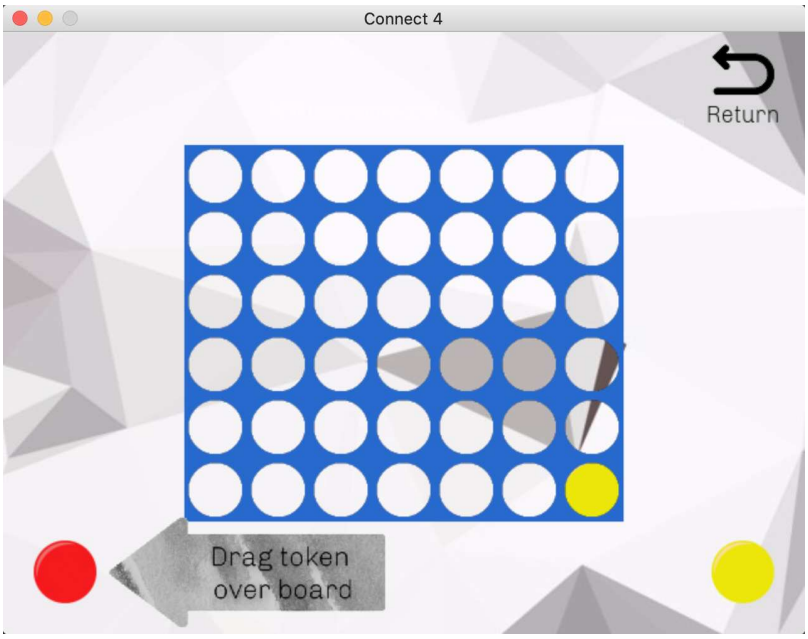


Slika 3.10: Početni izbornik koji omogućuje odabir težine igre te započinjanje igre odabrane težine.

Kao što možemo vidjeti na slici 3.10, igraču je omogućen odabir težine igre, pri čemu su dostupne težine *easy* (lagano), *medium* (srednje) i *hard* (teško). No, što određuje pojedinu razinu, odnosno, po čemu se jedna razlikuje od druge? Prvo, istaknimo da je moguće implementirati igranje čovjeka protiv agenata dobivenih u bilo kojem od pokretanja skripte koja implementira proces učenja. Međutim, u ovom odjeljku usredotočujemo se na neuronske mreže i agente dobivene u istom pokretanju čiji su rezultati predstavljeni u odjeljku 3.6. Općenito, ideja je da odabir lagane igre rezultira ostvarenjem igranja protiv agenta s nekom od prvih verzija neuronske mreže, da odabir teške igre znači igranje protiv agenta čija je

neuronska mreža ona koja je posljednja dobivena u procesu učenja, a da odabir srednje igre vodi k igri s agentom čija je pripadajuća verzija neuronske mreže neka otprilike središnja. Konkretno, za izvršavanje programa s rezultatima navedenim u odjeljku 3.6, lagana je igra ona protiv agenta s prvom verzijom neuronske mreže, srednja je ona protiv agenta s dvadesetom neuronskom mrežom, a teška je protiv posljednje verzije agenta, odnosno neuronske mreže.

Klik na neki od pravokutnika s ponuđenim težinama rezultira prikazom rešetke igre. Prva igra nakon odabira težine odvija se tako da prvo igra računalo te da se, kada dođe red na prvi potez čovjeka, prikaže strelica s uputstvom kako izvršiti potez. Takva je situacija prikazana slikom 3.11. Ljudskom je igraču uvijek dodijeljena crvena boja te on poduzima akciju povlačenjem diska crvene boje s „hrpe" takvih diskova u lijevom donjem kutu ekrana iznad stupca u koji disk želi ubaciti — potom se prikazuje upadanje diska u željeni stupac (ako je ubacivanje diska u njega legalna akcija). Namjera iza takve implementacije bila je postići iskustvo igranja što sličnije onom kakvo je prilikom igranja Četiri u nizu kao fizičke društvene igre. Također, u gornjem desnom kutu ekrana postavljena je strelica, na koju klik omogućava povratak na glavni izbornik igre te odabir potencijalno drukčije težine igre.



Slika 3.11: Grafičko sučelje za igranje igre Četiri u nizu protiv AlphaZero agenta.

Konačno, program pomoću ranije opisanih metoda klase `Game` utvrđuje je li neko dosegnuto stanje završno te, ako jest, kako je igra završila: pobjedom čovjeka, računala ili je neriješena. U svakom slučaju, igrača se o dosezanju završnog stanja i ishodu obavještava

odgovarajućim grafičkim prikazom. Primjerice, u slučaju pobjede čovjeka, grafički je prikaz kao na slici 3.12.

Slika 3.12: Obavijest igraču da je pobijedio računalo u igri Četiri u nizu.

Nakon što jedna igra završi, igrač klikom miša započinje novu igru. Međutim, ima i mogućnost završavanja izvršavanja programa klikom na gumb za zatvaranje prozora u kojemu se program prikazuje. Također, kao što smo ranije već naveli, klikom na strelicu u gornjem desnom kutu uvijek se može vratiti na glavni izbornik programa. Pomoću tako opisanog jednostavnog, ali funkcionalnog i intuitivnog sučelja, moguće je na slikovit i zabavan način osobno provjeriti vještine agenata dobivenih našom primjenom AlphaZero metode.

Bibliografija

[1] *Google's 'superhuman' DeepMind AI claims chess crown*, https://www.bbc.com/news/technology-42251535, posjećena u studenom 2020.

[2] *N-Step Lookahead*, https://www.kaggle.com/alexisbcook/n-step-lookahead, posjećena u studenom 2020.

[3] *Number of legal 7 X 6 Connect-Four positions after n plies*, https://oeis.org/A212693, posjećena u studenom 2020.

[4] L. V. Allis, *Searching for Solutions in Games and Artificial Intelligence*, Ponsen & Looijen, Wageningen, 1994.

[5] P. Auer, N. Cesa-Bianchi i P. Fischer, *Finite-time Analysis of the Multiarmed Bandit Problem*, Machine Learning **47** (2002), 235–256.

[6] M. Bošnjak, *Učenje podrškom*, https://web.math.pmf.unizg.hr/nastava/su/index.php/download_file/-/view/165/, posjećena u studenom 2020.

[7] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis i S. Colton, *A Survey of Monte Carlo Tree Search Methods*, IEEE Transactions on Computational Intelligence and AI in Games **4** (2012), 1–43.

[8] D. Foster, *How to build your own AlphaZero AI using Python and Keras*, https://medium.com/applied-data-science/how-to-build-your-own-alphazero-ai-using-python-and-keras-7f664945c188, posjećena u studenom 2020.

[9] M. Huzak, *Predavanja*, https://web.math.pmf.unizg.hr/nastava/stat/files/StatPred_statprocjena.pdf, posjećena u studenom 2020.

[10] ———, *Statistička procjena*, https://web.math.pmf.unizg.hr/nastava/stat/files/StatProcjena.pdf, posjećena u studenom 2020.

[11] ———, *Vjerojatnost i matematička statistika*, http://aktuari.math.pmf.unizg.hr/docs/vms.pdf, posjećena u studenom 2020.

[12] S. Knapton i L. Watson, *Entire human chess knowledge learned and surpassed by DeepMind's AlphaZero in four hours*, https://www.telegraph.co.uk/science/2017/12/06/entire-human-chess-knowledge-learned-surpassed-deepminds-alphazero, posjećena u studenom 2020.

[13] L. Kocsis i C. Szepesvári, *Bandit based Monte-Carlo Planning*, Machine Learning: ECML 2006 (J. Fürnkranz, T. Scheffer i M. Spiliopoulou, ur.), Springer, Berlin, Heidelberg, 2006, str. 282–293.

[14] L. Kocsis, C. Szepesvári i J. Willemson, *Improved Monte-Carlo Search*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.374.1202&rep=rep1&type=pdf, posjećena u studenom 2020.

[15] T. Lai i H. Robbins, *Asymptotically efficient adaptive allocation rules*, Advances in Applied Mathematics **6** (1985), 4–22.

[16] J. H. Lange, *Bandit-Based Methods*, https://www.ke.tu-darmstadt.de/lehre/archiv/ws-14-15/ml-se/Lange_Jan-Hendrik.pdf, posjećena u studenom 2020.

[17] M. Lapan, *Deep Reinforcement Learning Hands-On - Second Edition*, Packt Publishing, Birmingham, UK, 2020.

[18] A. Modirshanechi, *Why does the optimal policy exist?*, https://towardsdatascience.com/why-does-the-optimal-policy-exist-29f30fd51f8c, posjećena u studenom 2020.

[19] R. Mrazović, *Teorija igara*, https://web.math.pmf.unizg.hr/nastava/tigara/files/tigara-predavanja.pdf, posjećena u studenom 2020.

[20] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel i D. Hassabis, *Mastering the game of Go with deep neural networks and tree search*, Nature **529** (2016), 484–489.

[21] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan i D. Hassabis, *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, CoRR **abs/1712.01815** (2017), `http://arxiv.org/abs/1712.01815`, posjećena u studenom 2020.

[22] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan i D. Hassabis, *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play*, Science **362** (2018), 1140–1144.

[23] D. Silver, T. Hubert, J. Schrittwieser i D. Hassabis, *AlphaZero: Shedding new light on chess, shogi, and Go*, `https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go`, posjećena u studenom 2020.

[24] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel i D. Hassabis, *Mastering the game of Go without human knowledge*, Nature **550** (2017), 354–359.

[25] T. Simonite, *Alphabet's Latest AI Show Pony Has More Than One Trick*, `https://www.wired.com/story/alphabets-latest-ai-show-pony-has-more-than-one-trick`, posjećena u studenom 2020.

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

[27] M. Čupić, *Umjetna inteligencija*, `http://java.zemris.fer.hr/nastava/ui/games/games-20200311.pdf`, posjećena u studenom 2020.

Sažetak

U ovom je diplomskom radu predstavljen AlphaZero, algoritam tvrtke DeepMind koji *tabula rasa* može postići nadljudski učinak u raznovrsnim izazovnim domenama, poput šaha, shogija (japanskog šaha) i igre Go. Naime, dotadašnje prvake u navedenim trima igrama uvjerljivo je pobijedio, a njegovu su izuzetnost šahovski velemajstori istaknuli usporedbom njegovog igranja šaha s, primjerice, igranjem superiorne vanzemaljske vrste. Stvaranje algoritma koji *tabula rasa* stječe nadljudsku vještinu u zahtjevnim domenama bio je dugogodišnji cilj umjetne inteligencije te upravo AlphaZero, sa svojom sposobnošću prilagođavanja raznolikim pravilima igre, predstavlja njegovo ispunjenje i značajan korak naprijed prema ostvarenju općeg sustava za igranje igara. U radu su objašnjeni osnovni koncepti teorije koja leži u pozadini AlphaZero algoritma, podrobno je opisana struktura AlphaZero metode, a njezine su mogućnosti demonstrirane njenom implementacijom za igru Četiri u nizu pomoću programskog jezika Python i njegovih dodatnih biblioteka. U poglavlju 1 navedeni su relevantni pojmovi teorije igara i umjetne inteligencije (s naglaskom na klasu algoritama strojnog učenja podrškom, u koju možemo svrstati i AlphaZero algoritme), diskutirane su zajedničke karakteristike problema koje AlphaZero rješava, predstavljeni su formalni modeli igara koje uspješno savladava te je kroz objašnjenje strojnog učenja podrškom općenito stvorena podloga za razumijevanje temelja AlphaZero metode. Igre koje AlphaZero uspijeva naučiti uspješno igrati (primjerice, šah, shogi i Go) u kontekstu teorije igara modelirane su kao kombinatorne igre, odnosno deterministične ekstenzivne igre s dva igrača, sa sumom nula i potpunim informacijama. Također, opisane su alternirajuće Markovljeve igre, čiji formalizam AlphaZero slijedi te na koji se njegova metoda najizravnije primjenjuje, i njihov poseban slučaj, Markovljevi procesi odlučivanja (koji su formulacija problema koje rješava klasa algoritama strojnog učenja podrškom). U poglavlju 2 detaljno je opisana struktura AlphaZero metode; razložena je na tri komponente: pretraživanje stabla Monte Carlo metodom, igranje igara algoritma samog protiv sebe i nadzirano učenje, od kojih je svaka opširno opisana zasebno te za koje je objašnjeno na koji način tvore funkcionalnu cjelinu AlphaZero algoritma. U poglavlju 3 navedeni su određeni implementacijski detalji programskog ostvarenja AlphaZero metode za igru Četiri u nizu te su predstavljeni rezultati: napredak u sposobnostima igranja tijekom vremena te ostvaren uspjeh po završetku procesa učenja. Prikazan je uspjeh postignut u igrama pro-

tiv drugih algoritama za igranje igre Četiri u nizu, poput algoritma minimaks, ali i protiv čovjeka. Konačno, u sklopu ovog diplomskog rada implementirano je te opisano grafičko sučelje koje korisniku omogućava igranje igre Četiri u nizu protiv agenata dobivenih pokretanjem implementacije AlphaZero metode te koje predstavlja svojevrsnu „materijalizaciju" dobivenih rezultata i „opipljivi" konačan proizvod ovog diplomskog rada.

Summary

In this master thesis we have acquainted the reader with AlphaZero, DeepMind's algorithm capable of achieving, *tabula rasa*, superhuman performance in many challenging domains, such as chess, shogi (Japanese chess) and Go. Not only has AlphaZero managed to convincingly defeat the previous world-champions in all the aforementioned games, but its exceptional abilities have been described by chess grandmasters as those to be expected from a superhuman extraterrestrial species. The creation of an algorithm which would be able to achieve, *tabula rasa*, superhuman skills in various challenging domains has been a longstanding objective of artificial intelligence. AlphaZero, with its ability to adapt to diverse game rules, can be considered to be its realization and a major step towards the attainment of a general game playing system. In this thesis we have explained the basic theoretical concepts underpininng AlphaZero, thoroughly described the structure of the AlphaZero method and demonstrated its possibilities by implementing it for a game called Connect Four (or Four in a Row) using Python as the programming language and its additional libriaries. In chapter 1 we have introduced the pertinent concepts of game theory and artifical intelligence (with emphasis on reinforcement learning — a class of algorithms AlphaZero itself belongs to), studied both which charateristics are common to the problems successfully solved by AlphaZero and how to formally model such problems, as well as facilitated the understanding of the foundations of the AlphaZero method through making sense of reinforcement learning in general. The games AlphaZero has managed to master (such as chess, shogi and Go) have been formulated in the context of game theory as combinatorial games, which are deterministic, zero-sum, perfect information extensive games with two players. Moreover, we have desribed alternating Markov games, whose formalism AlphaZero follows and to which the AlphaZero method can be applied most directly, along with their special case, Markov decision processes (a formulation of the problems solved by reinforcement learning algorithms). In chapter 2 we have comprehensively described the structure of the AlphaZero method; with Monte Carlo tree search, self-play and supervised learning being its three main components, we have expounded on each one of them and elucidated on how they all come together to form a functional whole that constitues the AlphaZero algorithm. In chapter 3 we have discussed certain details of our AlphaZero implementation for the game of Connect Four, as well as presented the

results: the progress in game-playing abilities over the course of training and the achieved level of play at the end of training. We have demonstrated our agents' strengths through plays against other Connect Four algorithms, such as a minimax-based algorithm, as well as in matches against humans. Finally, as part of this master thesis, we have implemented and described a graphical user interface which allows human players to compete against agents obtained during the training process. By creating such a game implementation, we have „materialized" the obtained results and produced a „tangible" final product of this master thesis.

Životopis

Rođena sam 25. veljače 1997. u Zagrebu, gdje sam odrasla i gdje trenutno živim. Nakon završene Osnovne škole Savski gaj, upisala sam B program („matematički" program) V. gimnazije u Zagrebu, u kojoj sam svaki razred završila s prosjekom 5.00. Na državnoj sam maturi ostvarila rekordan broj potpuno riješenih testova — testove iz fizike te matematike i engleskog jezika na višoj razini riješila sam potpuno točno, zbog čega sam primila čestitke Hrvatske akademije znanosti i umjetnosti. Tijekom osnovnoškolskog i srednjoškolskog obrazovanja sudjelovala sam na državnim natjecanjima iz fizike i biologije. 2015. godine upisala sam preddiplomski sveučilišni studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. 2018. godine, kao studentica završne godine navedenog preddiplomskog studija, nagrađena sam od strane Matematičkog odsjeka Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu za izniman uspjeh na studiju. Iste sam godine stekla akademski naziv sveučilište prvostupnice matematike (*univ. bacc. math.*). Potom sam na Matematičkom odsjeku PMF-a Sveučilišta u Zagrebu upisala diplomski sveučilišni studij Računarstvo i matematika. Nagradu Matematičkog odjseka PMF-a Sveučilišta u Zagrebu za izniman uspjeh ponovno sam primila 2020. godine, ovaj put kao studentica završne godine navedenog diplomskog studija. Materinji mi je jezik hrvatski, tečno govorim engleski, a pet sam godina u sklopu osnovnoškolskog obrazovanja učila i talijanski jezik.