math.e

Hrvatski matematički elektronički časopis

AlphaZero: strojno učenje podrškom bez domenskog znanja

Zvonimir Bujanović, Jelena Lončar
Sveučilište u Zagrebu, Prirodoslovno matematički fakultet, Matematički odjel

Sažetak

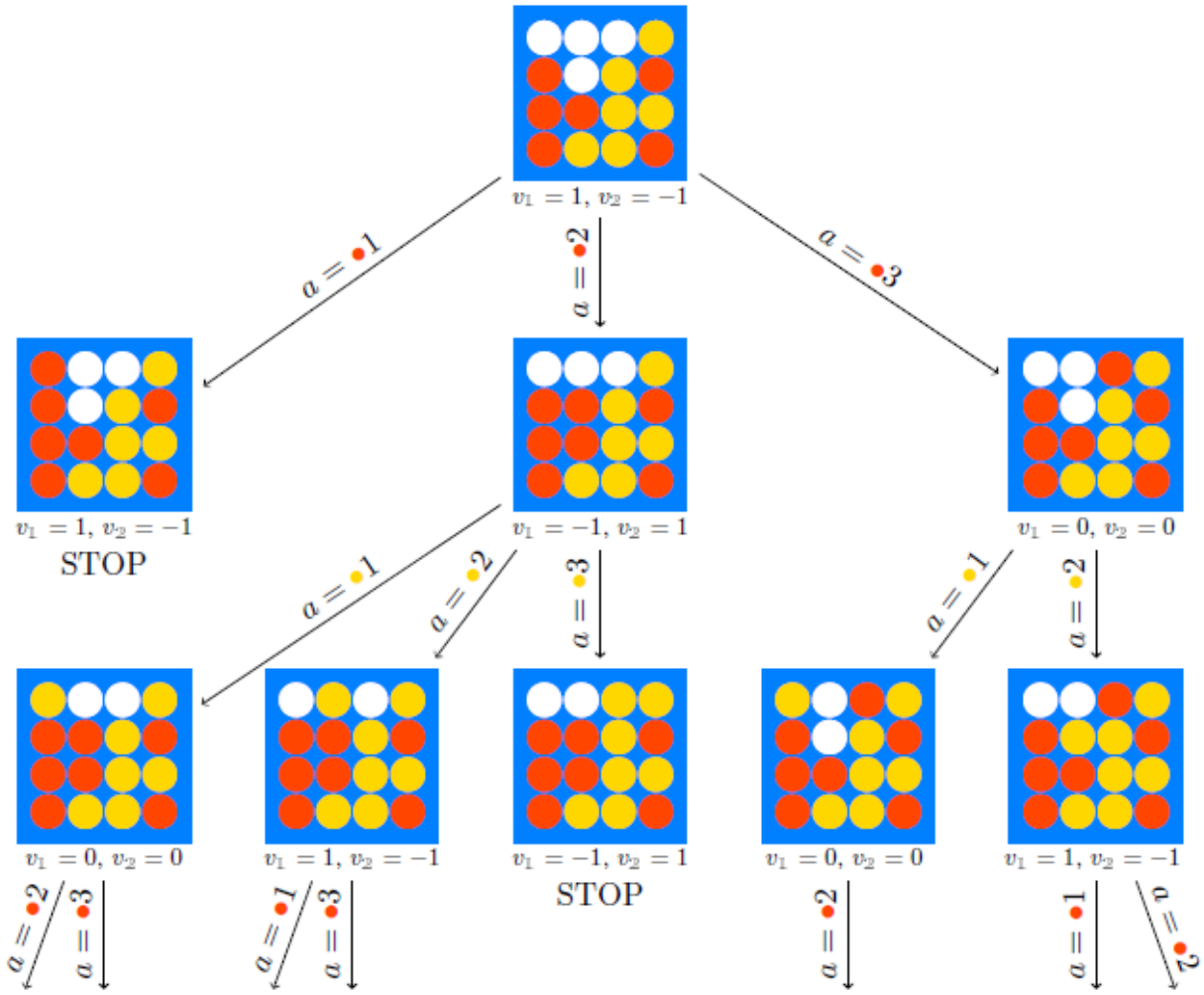
U ovom članku ćemo opisati *AlphaZero*, algoritam tvrtke DeepMind koji *tabula rasa* (to jest, bez unaprijed implementirane ikakve strategije igranja osim samih pravila) može postići nadljudski učinak u raznovrsnim izazovnim domenama, poput šaha, shogija (japanskog šaha) i igre Go. Predstavljen u [14], ovaj algoritam je uvjerljivo pobijedio ponajbolje svjetske igrače u navedenim trima igrama, a njegovu su izuzetnost šahovski velemajstori usporedili s igrom kakvu bi prezentirala superiorna vanzemaljska vrsta. Stvaranje algoritma koji *tabula rasa* stječe nadljudsku vještinu u zahtjevnim domenama bio je dugogodišnji cilj umjetne inteligencije te upravo AlphaZero, svojom sposobnošću prilagođavanja raznolikim pravilima igre, predstavlja njegovo ispunjenje i značajan korak naprijed prema ostvarenju općeg sustava za igranje igara. U članku ćemo izložiti osnovne koncepte algoritma AlphaZero, te demonstrirati rezultate dobivene njegovom implementacijom za igru *Connect Four* (Četiri u nizu) pomoću programskog jezika Python i njegovih dodatnih biblioteka. Za dodatne pojedinosti čitatelja upućujemo na diplomski rad [10].

1 Kombinatorne igre i reprezentacija pomoću stabla

Algoritam AlphaZero nudi elegantan pristup rješavanju problema igranja igara na ploči, gdje su pravila igre fiksirana, aktualni položaj u igri je 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. Preciznije, algoritam AlphaZero pogodan je za tzv. *kombinatorne igre za dva igrača*, a takve igre imaju sljedeća svojstva:

- Sekvencijalnost: igrači, označimo ih sa $\mathbf{1}$ i $\mathbf{2}$, naizmjenice vuku svoje poteze (akcije).
- Potpuna informacija: oba igrača u svakom trenutku imaju dostupne sve informacije o trenutnom stanju igre. Na primjer, u šahu oba igrača vide pozicije svih figura (i svojih i suparničkih) pa je šah igra s potpunom informacijom. S druge strane, u pokeru igrač ne vidi protivničke karte, pa to nije igra s potpunom informacijom.
- Suma *nagrada* koje igrači dobiju na kraju igre je jednaka nuli. Na primjer, možemo reći da je nagrada za pobjedu u šahu jednaka 1, za poraz -1, a za remi 0. Drugim riječima, u takvim igrama pobjeda jednog igrača znači poraz drugog.
- Determinizam: za svako stanje igre s i za svaku akciju a koja je dozvoljena u tom stanju postoji jedinstveno novo stanje $\tilde{s} = \mathsf{next}(s, a)$ u koje igra prelazi nakon što je odigrana akcija a. Drugim riječima, u igri nema faktora slučajnosti.

Kombinatorne igre se tipično modeliraju pomoću stabla, poput onog prikazanog na slici 1. Vrhovi (čvorovi) u stablu 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. Ako analiziramo situaciju u nekom danom trenutku igre, onda se u korijenu stabla nalazi opis stanja u tom trenutku. Djeca nekog vrha predstavljaju sva ona stanja do kojih se može doći jednim potezom u igri, a završna stanja u igri (ona koja predstavljaju kraj igre) su listovi u stablu. Svakom završnom stanju je pridružena nagrada koju pojedini igrač dobije ako igra završi u tom stanju; kao što smo naveli gore, suma tih nagrada za svako stanje je nula.



Slika 1: Primjer stabla igre *Connect Four* na 4×4 ploči (pravila igre opisana su na početku sekcije 4). U korijenu stabla je stanje u kojem je na potezu igrač $\mathbf{1}$, te koji ima na raspolaganju tri akcije: svoj crveni žeton može ubaciti u neki od prva tri stupca. Nakon bilo koje od akcija igrača $\mathbf{1}$ (osim ubacivanja u prvi stupac koje rezultira završetkom igre i pobjedom $\mathbf{1}$), na redu je $\mathbf{2}$ koji ponovno može ubaciti svoj žuti žeton u bilo koji od stupaca koji nije pun. Ispod čvorova stabla navedene su vrijednosti v_1 i v_2 pripadnih stanja za svakog od igrača.

Prirodno pitanje koje si svaki igrač postavlja tijekom igre jest: koja je najveća moguća nagrada koju mogu ostvariti svojim potezima iz stanja s u kojem se igra trenutno nalazi, bez obzira na to kako igra drugi igrač? Tu najveću moguću nagradu

zovemo *vrijednost* stanja s za danog igrača i, te označavamo sa $v_i(s)$. Lako se, indukcijskim argumentom počevši od listova stabla, vidi da za svako stanje s vrijedi $v_{\mathbf{1}}(s) = -v_{\mathbf{2}}(s)$. Označimo sa $\mathcal{A}(s)$ skup dozvoljenih poteza (akcija) koji se smiju napraviti u stanju s, a sa $\mathcal{N}(s) = \{\mathsf{next}(s,a) : a \in \mathcal{A}(s)\}$ skup svih stanja u koja igra može prijeći nakon stanja s. Promotrimo situaciju u kojoj je igrač $\mathbf{1}$ na potezu u igri koja je u stanju s. Njegov optimalni potez vodi ga u stanje u kojem je njegova nagrada najveća moguća, odnosno, vrijedi:

$$v_{\mathbf{1}}(s) = \max_{\tilde{s} \in \mathcal{N}(s)} v_{\mathbf{1}}(\tilde{s}). \tag{1}$$

No, na potezu u stanju $\tilde{s}$ je igrač $\mathbf{2}$. Iz njegove perspektive,

$$v_{\mathbf{1}}(\tilde{s}) = -v_{\mathbf{2}}(\tilde{s}) = -\max_{\hat{s} \in \mathcal{N}(\tilde{s})} v_{\mathbf{2}}(\hat{s}) = -\max_{\hat{s} \in \mathcal{N}(\tilde{s})} (-v_{\mathbf{1}}(\hat{s})) = \min_{\hat{s} \in \mathcal{N}(\tilde{s})} v_{\mathbf{1}}(\hat{s}),$$

pa uvrštavajući u (1) dobivamo

$$v_{\mathbf{1}}(s) = \max_{\tilde{s} \in \mathcal{N}(s)} v_{\mathbf{1}}(\tilde{s}) = \max_{\tilde{s} \in \mathcal{N}(s)} \min_{\hat{s} \in \mathcal{N}(\tilde{s})} v_{\mathbf{1}}(\hat{s}). \tag{2}$$

Iz ove jednakosti slijedi rekurzivni algoritam *minimax* koji, barem u teoriji, može pronaći pobjedničku strategiju ako ona postoji. Algoritam generira stablo igre, pri čemu korijen opisuje trenutno stanje s u igri, te izračunava vrijednost stanja za svaki čvor koristeći vrijednosti stanja njegove djece i formulu (2). Pretpostavimo, bez smanjenja općenitosti, da je u stanju s na potezu igrač $\mathbf{1}$. Optimalna akcija koju on može napraviti je ona koja vodi do djeteta korijena maksimalne vrijednosti. Problem s ovim pristupom jest taj što se stablo svake imalo složenije igre izuzetno brzo grana—ako u svakom stanju možemo odabrati između n akcija, onda analiza prvih ℓ poteza u igri vodi na stablo sa $\mathcal{O}(n^\ell)$ čvorova. To znači da računalo u praksi ne može provesti ovaj algoritam u razumnom vremenu, nego je nužno ograničiti dubinu stabla; u čvorovima na najvećoj dubini treba nekom heuristikom pronaći aproksimaciju vrijednosti pripadnog stanja. Time dobivamo Algoritam 1 koji nazivamo *minimax*$\ell\}$, gdje je ℓ ta unaprijed zadana najveća dubina. Za čvor u smo u algoritmu sa $\mathsf{state}(u)$ označili pripadno stanje, a sa $\mathsf{value}(u) \approx v_{\mathbf{1}}(\mathsf{state}(u))$ aproksimaciju vrijednosti tog stanja za igrača $\mathbf{1}$ koji je na potezu u korijenu. Računalne programe koji vuku poteze slijedeći strategiju opisanu nekim algoritmom još nazivamo i *agentima*.

Kako odrediti aproksimativnu vrijednost nekog stanja? Jedna mogućnost je napraviti jednu ili više simulacija igre od tog stanja nadalje, koristeći pritom neku trivijalnu strategiju (na primjer, u nedostatku bolje ideje, posve slučajno odabirati poteze), te vratiti prosječnu nagradu koju je u svim simulacijama ostvario igrač $\mathbf{1}$. Alternativno, za konkretnu igru moguće je procijeniti situaciju u kojoj se nalazi igrač: na primjer, u šahu aproksimativna vrijednost stanja bi mogla biti 0.9 ako igrač $\mathbf{1}$ svojim potezom može zaprijetiti šahom, a -0.9 ako je u takvoj situaciji protivnik i slično. Ove ideje navodimo samo za ilustraciju; u praksi se, naravno, koriste nešto složenije aproksimacije.

ALGORITAM 1: *minimax-ℓ*

funkcija `minimax`(*čvor u, dubina d, igrač p*)
- **if** state(u) *je završno stanje*
  - **return** nagrada za igrača $\mathbb{1}$ u stanju state(u);
- **if** $d = 0$
  - **return** heuristička aproksimacija vrijednosti stanja state(u) za igrača $\mathbb{1}$;
- **if** $p = \mathbb{1}$
  - value$(u) = -\infty$;
  - **for** $a \in \mathcal{A}(\text{state}(u))$
    - Dodaj novo dijete c čvoru u koje odgovara stanju next(state$(u), a)$;
    - value$(u) = \max\{\text{value}(u), \texttt{minimax}(c,\ d-1,\ \mathbb{2})\}$;
  - **return** value(u);
- **else**
  - value$(u) = +\infty$;
  - **for** $a \in \mathcal{A}(\text{state}(u))$
    - Dodaj novo dijete c čvoru u koje odgovara stanju next(state$(u), a)$;
    - value$(u) = \min\{\text{value}(u), \texttt{minimax}(c,\ d-1,\ \mathbb{1})\}$;
  - **return** value(u);

U glavnom programu:
Napravi stablo koje se sastoji samo od korijena k koji opisuje trenutno stanje s u igri;
Pozovi `minimax`(k, ℓ, *igrač $\mathbb{1}$ koji je na potezu u stanju s*);
Odigraj akciju koja vodi do djeteta čvora k koje ima najveću vrijednost;

2 Pretraživanje stabla Monte Carlo metodom

Pretraživanje stabla Monte Carlo metodom (engl. *Monte Carlo Tree Search*, MCTS) je alternativna metoda rješavanja problema s prevelikim brojem čvorova u stablu igre. Promotrimo ponovno situaciju u kojoj trebamo odabrati akciju za igrača $\mathbf{1}$ na potezu u stanju s. Umjesto da napravi potpuno stablo igre u čijem korijenu je stanje s, MCTS nastoji usmjeriti izgradnju stabla tako da otkriva stanja za koje je izglednije da imaju veću vrijednost za igrača $\mathbf{1}$. To se postiže formiranjem vjerojatnosnog modela igre: MCST želi odrediti broj $\pi(a)$ koji predstavlja vjerojatnost da je $a \in \mathcal{A}(s)$ optimalna akcija za igrača $\mathbf{1}$. Tu vjerojatnost će aproksimirati statistikom dobivenom simulacijama velikog broja partija koje sve započinju stanjem s. Tijekom simulacija se postupno gradi stablo igre, a akcije u simulacijama se biraju na dva načina, *politikom stabla* (eng. *tree policy*) i *zadanom politikom* (eng. *default policy*):

- Politika stabla definira kako u čvoru u do sada izgrađenog stabla odabrati sljedeću akciju u tijeku jedne simulacije. Odabir sljedeće akcije nas pomiče na dijete tog čvora (ako ono već od ranije postoji u stablu) ili dodaje novi čvor u stablo (ako odabranu akciju nismo proveli niti u jednoj ranijoj simulaciji).
- Zadana politika definira provedbu simulacije od stanja u novododanom čvoru (vidi gornju točku) do završetka igre. Zadana politika ima ulogu koja je analogna heurističkoj aproksimaciji vrijednosti stanja kod algoritma *minimax-ℓ*.

U svakom čvoru u MCTS stabla čuvamo sljedeće informacije: $N(u)$ je broj simulacija koje su prošle kroz čvor u (tj. simulacija koje su u nekom trenutku bile u stanju iz tog čvora), a $Q(u)$ je suma svih (aproksimacija) vrijednosti stanja state(u) u svim simulacijama za onog igrača čiji potez vodi na to stanje. U svakoj

simulaciji ove se vrijednosti osvježe za sva stanja kroz koje simulirana igra prođe.
Pseudokod osnovnog MCTS algoritma je dan u Algoritmu 2.

ALGORITAM 2: Osnovni MCTS algoritam – odabir akcije iz stanja s

```
funkcija MCTS_simulacija()
    u = korijen do sada sagrađenog MCTS stabla;
    while true do
        p = igrač na potezu u stanju u;
        u = politika_stabla(u);
        if state(u) je završno stanje igre
            Δ = nagrada u stanju state(u) za igrača p;
            N(u) = N(u) + 1; Q(u) = Q(u) + Δ;
            break;
        else if čvor u još nije posjećen, tj. N(u) = 0 then
            Δ = zadana_politika(u), tj. aproksimacija vrijednosti stanja state(u) za p;
            N(u) = 1; Q(u) = Δ;
            for a ∈ A(state(u))
                Stvori novo dijete c čvora u do kojeg vodi akcija a;
                N(c) = 0; Q(c) = 0;
            break;
    while u nije korijen MCTS stabla do
        u = roditelj čvora u;
        Δ = −Δ;
        N(u) = N(u) + 1; Q(u) = Q(u) + Δ;

funkcija politika_stabla(čvor u)
    if postoji dijete c čvora u koje još nije posjećeno, tj. N(c) = 0
        return c;
    else
        Odaberi dijete c čvora u koristeći, na primjer, formulu (2.1) ili (3.1);
        return c;

U glavnom programu:
Napravi stablo koje se sastoji od korijena k koji opisuje trenutno stanje s u igri;
for a ∈ A(s)
    Stvori novo dijete c čvora k do kojeg vodi akcija a;
    N(c) = 0; Q(c) = 0;
while vrijeme za odigravanje jednog poteza u igri nije isteklo do
    Pozovi MCTS_simulacija();
Odigraj akciju koja vodi do maksimalnog ili do robusnog djeteta čvora k;
```

Preostaje još pitanje kako implementirati politiku stabla. Jasno je da su, iz perspektive igrača koji je na potezu u stanju $\mathsf{state}(u)$, perspektivnija ona djeca c tog čvora za koja je u ranijim simulacijama ostvario veću prosječnu nagradu $\frac{Q(c)}{N(c)}$.
S druge strane, nužno je povremeno odigrati akciju koja ranije nije bila često ili uopće igrana kako ne bismo propustili otkriti neki novi, obećavajući smjer u igri. Taj balans između iskorištavanja postojeće statistike za maksimizaciju nagrade i istraživanja novih akcija postiže se politikom tzv. gornje pouzdane ograde za stabla (eng. *UCT, Upper Confidence Bound for trees*): ako sa $\mathcal{C}(u)$ označimo skup djece čvora u, onda politika stabla UCT odabire onu akciju iz čvora u koja vodi u dijete

$$\arg\max_{c\in\mathcal{C}(u)}\left(\frac{Q(c)}{N(c)}+\alpha\sqrt{\frac{2\ln N(u)}{N(c)}}\right). \tag{3}$$

Akcije koje su se u prethodnim simulacijama pokazale boljima će imati veći prvi pribrojnik, a akcije koje ranije nisu bile često birane će imati veći drugi pribrojnik; tipična vrijednost koeficijenta α je $\sqrt{2}$. Teorijsko opravdanje formule (3) može se pronaći u člancima [??]ctssurvey,Kocsis06banditbased,Kocsis2006ImprovedMS}.

Nakon što je napravljen dovoljan broj simulacija, igrač $\mathbf{1}$ treba odabrati akciju dozvoljenu u stanju zapisanom u korijenu MCTS stabla. To se najčešće obavlja na jedan od sljedećih načina:

- Odaberi akciju koja vodi na dijete c s najvećom prosječnom nagradom $\frac{Q(c)}{N(c)}$, odnosno, dijete koje daje formula (3) uz $\alpha=0$ (tzv. *maksimalno dijete*);
- Odaberi akciju koja vodi na najčešće posjećeno dijete korijena (tzv. *robusno dijete*). Ako sa $\mathsf{child}(u,a)$ označimo dijete čvora u do kojeg vodi akcija a, onda je logika iza ovog odabira u modeliranju vjerojatnosti poduzimanja akcije a sa $\pi(a)\approx\frac{N(\mathsf{child}(u,a))}{\sum_{c\in\mathcal{C}(u)}N(c)}$.

Nakon što i protivnički igrač odigra svoj potez, igrač $\mathbf{1}$ bi mogao graditi posve novo MCTS stablo s korijenom u novonastaloj situaciji. Bolji pristup je zadržati odgovarajuće podstablo postojećeg MCTS stabla jer ono već sadrži dio statistike za simulacije iz prethodnog stanja igre.

3 Algoritam AlphaZero

Najizraženiji problem koji preostaje u algoritmu opisanom u prethodnoj cjelini je definicija smislene zadane politike, odnosno, što bolje aproksimacije vrijednosti stanja za listove do sada izgrađenog MCTS stabla. AlphaZero taj problem rješava korištenjem duboke neuronske mreže f_θ. Implementacija iz [13] sugerira da neuronska mreža na ulazu dobije kodirano stanje s, a da joj izlaz, uz aproksimaciju $\hat{v}$ vrijednosti tog stanja bude i tzv. *vektor politike* (eng. *policy vector*) $\hat{\pi}$:

$$(\hat{v},\hat{\pi})=f_\theta(s).$$

Vektor politike svakoj akciji a pridružuje vjerojatnost $\hat{\pi}(a)$ odigravanja te akcije u stanju s; uočimo analogiju ove ideje s motivacijom za algoritam MCTS. Neuronska mreža izgrađuje svoj vjerojatnosni model igre temeljem vjerojatnosti $\hat{\pi}$, dok istovremeno MCTS algoritam izgrađuje svoj model temeljem vjerojatnosti π. Cilj treniranja neuronske mreže je da, osim dobivanja što realnijih aproksimacija vrijednosti stanja koje odgovaraju stvarnim ishodima igara, izgradi vjerojatnosni model koji se podudara s onim dobivenim MCTS stablom.

Opišimo sada proces treniranja neuronske mreže, čiji pseudokod je prikazan i u Algoritmu 3. Težine neuronske mreže na početku se inicijaliziraju na nasumične vrijednosti θ_0. U svakoj iteraciji treniranja 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, bez obzira na to je li na potezu igrač $\mathbf{1}$ ili $\mathbf{2}$, poziva se algoritam pretraživanja stabla Monte Carlo metodom sa s_t u korijenu stabla. Formula (3) je pri tome malo prilagođena da uzima u obzir i vektor politike neuronske mreže: u svakom čvoru c se dodatno pamti i vrijednost $P(c):=\hat{\pi}(a)$, gdje je a akcija koja nas je dovela u taj čvor iz njegovog roditelja. Ova vrijednost se zapiše u čvor prilikom njegovog stvaranja u Algoritmu 2, tj. nakon poziva funkcije `zadana_politika` koja sada evaluira neuronsku mrežu i odmah se, uz $N(c)=0$, $Q(c)=0$ inicijalizira i $P(c)$ za svako dijete c čvora u. Nova politika stabla PUCT (eng. *Polynomial* UCT) glasi:

$$\arg\max_{c\in\mathcal{C}(u)}\left(\frac{Q(c)}{N(c)}+\alpha(u)P(c)\frac{\sqrt{N(u)}}{1+N(c)}\right),\quad \alpha(u)=\ln\left(\frac{1+N(u)+c_{\text{base}}}{c_{\text{base}}}\right)+c_{\text{init}}. \tag{4}$$

Ovdje su c_{base} i c_{init} konstante; vidimo da je izbor akcije koja ranije nije bila često birana dodatno ponderiran s vjerojatnosti koju joj je pridružila neuronska mreža. Po izgradnji stabla, MCST algoritam vraća procjenu vjerojatnosti odigravanja pojedinih

akcija iz korijena stabla (označimo ga s k) u obliku vektora π_t,

$$\pi_t(a) := \frac{N(\mathsf{child}(k,a))^{1/\tau}}{\sum_{c\in\mathcal{C}(k)} N(c)^{1/\tau}}. \tag{5}$$

Ovdje je τ tzv. *parametar temperature* koji se u prvih nekoliko (na primjer, 10) poteza svake igre postavlja na 1, a zatim se uzima $\tau \to 0$. U stanju s_t program će slučajno odabrati neku akciju a s vjerojatnosti $\pi_t(a)$, te ju odigrati—uočimo da izbor parametra temperature ima za posljedicu to da je u početku igre vjerojatnost odabira akcije proporcionalna posjećenosti djeteta korijena MCTS stabla, dok se kasnije uvijek odabire najposjećenije (robusno) dijete. Pripadno podstablo MCTS stabla izgrađenog za stanje s_t zadržava se i u narednom koraku.

Igra završava u koraku T, kada se dosegne završno stanje s_T, a kojemu pripada nagrada r_T. Iz svakog pojedinog koraka $1 \leq t < T$ pohranjuje se uređena trojka (s_t, π_t, v_t), gdje je $v_t = \pm r_T$ nagrada iz perspektive igrača koji igru dovodi u stanje s_t. Po završetku iteracije i, novi parametri neuronske mreže θ_i dobivaju se treniranjem neuronske mreže na temelju primjera za učenje oblika (s, π, v), dobivenih uzimanjem uzorka iz diskretne uniformne distribucije na skupu uređenih trojki (s_t, π_t, v_t) iz svih koraka t svih odigranih igara u iteraciji i. Ažurirani parametri θ_i potom se koriste u igrama algoritma samog protiv sebe u narednoj iteraciji (odnosno, kao zadana politika u narednoj iteraciji).

Samo treniranje na temelju primjera (s, π, v) provodi se ovako: neuronska mreža se evaluira za ulaz s, te vraća $(\hat{v}, \hat{\pi}) = f_\theta(s)$, a njezini parametri θ se ažuriraju s ciljem minimizacije funkcije greške (eng. *loss function*)

$$L((v,\pi),(\hat{v},\hat{\pi})) := (v-\hat{v})^2 - \sum_{a\in\mathcal{A}(s)} \pi(a)\log_2\hat{\pi}(a) + c\|\theta\|^2. \tag{6}$$

Analizirajmo malo gornji izraz. Primjer (s, π, v) predstavlja jedan ishod odigrane igre koja je prošla kroz stanje s: ona je završila nagradom v, a MCTS stablo je procijenilo vjerojatnosti akcija vektorom π. S druge strane, neuronska mreža će aproksimirati vrijednost tog stanja sa $\hat{v}$, a vjerojatnosti vektorom $\hat{\pi}$. Minimizacijom prvog pribrojnika u (6) nastojimo prilagoditi parametre mreže tako da njezina aproksimacija vrijednosti stanja bude što bliža stvarnoj nagradi v. Drugi pribrojnik mjeri sličnost vjerojatnosnih distribucija π i $\hat{\pi}$ pomoću tzv. *cross-entropy* funkcije

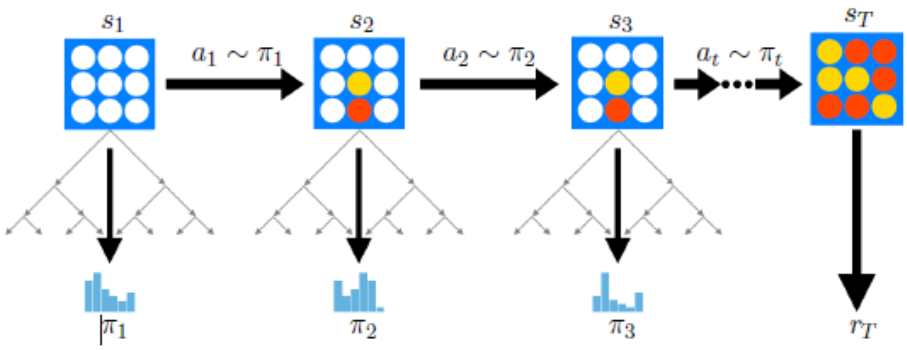
$$H(\pi,\hat{\pi}) := -\sum_{a\in\mathcal{A}(s)} \pi(a)\log_2\hat{\pi}(a)$$

koja postiže minimum (za fiksni π) kada je $\hat{\pi} = \pi$, pa je cilj minimizacije funkcije L postići da neuronska mreža daje što sličniju prognozu vjerojatnosti igranja pojedinih akcija kao što ih daje MCTS stablo. Treći pribrojnik u (6) je tzv. regularizacijski član—minimizacija norme vektora parametara θ je standardna tehnika kojom se smanjuje opasnost od tzv. *overfittinga*, tj. pojave da mreža daje ispravna predviđanja za podatke kojima ju treniramo, a loša za podatke koje ne vidi u procesu treninga. Optimizacijski algoritam za minimizaciju izraza (6) koristi gradijentni spust. Po završetku procesa treniranja, dobivenu neuronsku mrežu možemo pohraniti, a MCTS algoritam koji ju koristi kao zadanu politiku zatim primjenjivati kao AlphaZero agenta.

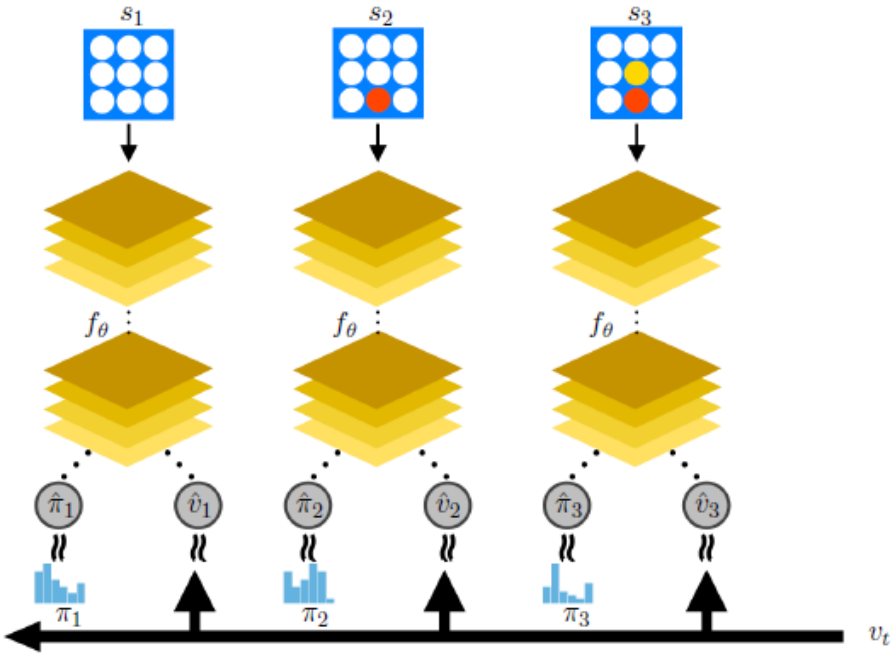
ALGORITAM 3: Procedura treniranja u algoritmu AlphaZero

Slučajno inicijaliziraj težine neuronske mreže f_θ;
for *iteracija* $i = 1, 2, \ldots$
  for *igra* $g = 1, 2, \ldots$
    $s_1 =$ početno stanje igre; $T = 1$;
    while s_T *nije završno stanje igre* **do**
      Sagradi MCTS stablo izvjesne veličine sa s_T u korijenu, za zadanu politiku koristi neuronsku mrežu f_θ;
      Odaberi akciju $a \in \mathcal{A}(s_T)$ slučajno temeljem vjerojatnosti (3.2);
      $s_{T+1} = \mathsf{next}(s_T, a)$; $T = T + 1$;
    Odredi nagradu r_T i pripadne aproksimacije vrijednosti $v_t = \pm r_T$ za $t = 1, \ldots, T$;
    Spremi (s_t, π_t, v_t) u niz ukupne duljine L (pamti samo najnovijih L trojki, $L \gg T$);
  Slučajno odaberi K trojki (s, π, v) iz niza i provedi treniranje neuronske mreže f_θ;

Opisani proces učenja ilustriran je slikom 5. Na slici 5 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, a koji pak koristi aktualnu neuronsku mrežu f_θ kao zadanu politiku. Ilustrirani su odabiri akcija na temelju vjerojatnosti proizašlih iz pretraživanja stabla Monte Carlo metodom (odabir akcije $a_t \sim \pi_t$ u koraku t) te pridjeljivanje nagrade završnom stanju s_T, na temelju koje se računaju ishodi za pojedine trenutke. Na slici 5 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 $\hat{\pi}_t$ i skalar $\hat{v}_t$. Arhitektura same neuronske mreže je također važna za uspjeh opisanog algoritma; ideja je slična za većinu igara na ploči pa ćemo ju u idućoj cjelini opisati za igru *Connect Four*.



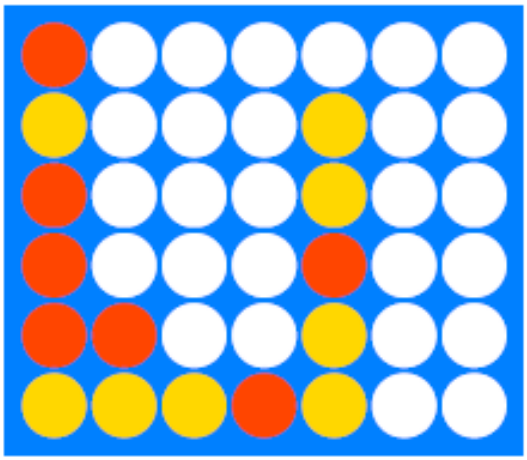
(a) Igranje igara algoritma samog protiv sebe.



(b) Treniranje neuronske mreže.

4 Rezultati za igru *Connect Four*

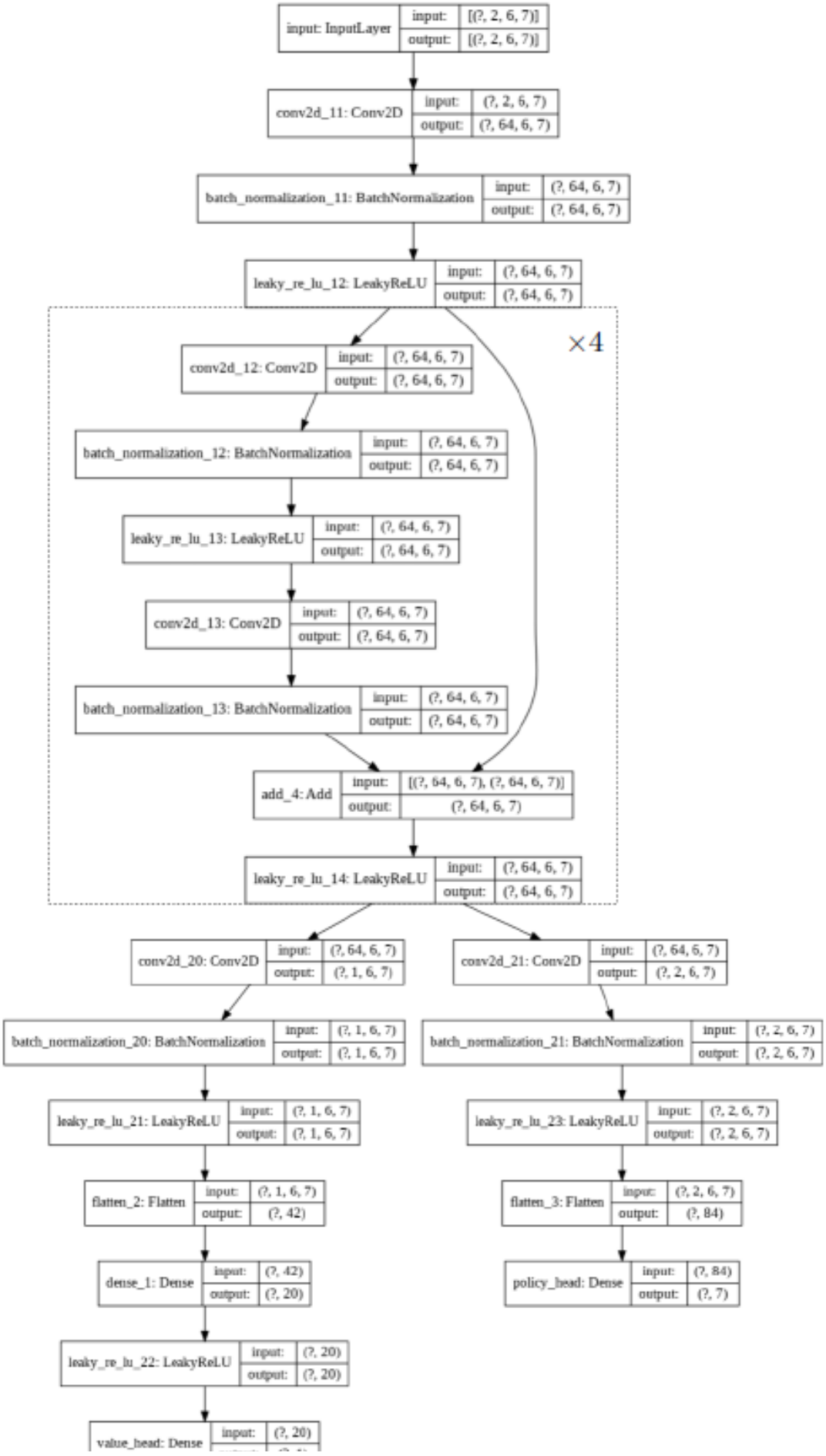
Za ilustraciju algoritma AlphaZero, napravili smo implementaciju za igru *Connect Four* pomoću programskog jezika Python i dodatnih biblioteka za njega. Vjerujemo da su čitatelji upoznati s ovom igrom, no opišimo ju ukratko. *Connect Four* je društvena igra za dva igrača, u kojoj je svakome od njih dodijeljena jedna boja (žuta ili crvena) te u kojoj oni 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. Kada 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 pojedinog igrača je biti prvi koji će formirati vodoravni, okomiti ili dijagonalni niz od četiri diska svoje boje.



Slika 8: Jedno stanje u tijeku igre *Connect Four* na ploči standardnih dimenzija 6×7.

Igru *Connect Four* intuitivno bismo mogli okarakterizirati kao jednostavnu —pogotovo ako ju promatramo u odnosu na kompleksne igre poput igre Go. Međutim, ukupan broj različitih stanja igre, prema [1], iznosi $4531985219092 \approx 4.5 \cdot 10^{12}$, što predstavlja izazov računalima u smislu rješavanja igre *brute-force* metodama poput algoritma *minimax*, odnosno sistematičnim pretraživanjem svih mogućnosti. Upravo zbog neposrednosti svojih pravila, jednostavnosti, koja se ogleda u manjim (u odnosu na npr. Go) zahtjevima na resurse prilikom implementacije AlphaZero algoritma za njeno savladavanje, popularnosti, ali ujedno i netrivijalnosti, igra *Connect Four* dobar je kandidat za demonstraciju mogućnosti AlphaZero metode. Naša implementacija je izvedena po uzoru na [9, Poglavlje 23], a koristili smo biblioteku Keras [4] te Tensorflow [2]. Keras je programsko sučelje (API) za duboko strojno učenje napisan u programskom jeziku Python, koji sadrži implementacije brojnih "građevnih blokova" neuronskih mreža, poput slojeva i aktivacijskih funkcija, te raznovrsnih često upotrebljavanih optimizacijskih algoritama.

Arhitektura neuronske mreže za algoritam AlphaZero je skicirana na slici 9. Ulazni sloj reprezentira stanje u igri kao tenzor dimenzije $(2, 6, 7)$, odnosno, dvije matrice dimenzije ploče za igru (6×7). Svaki element tenzora je 0 ili 1: na mjestu $(0, i, j)$ piše 1 ako i samo ako igrač na potezu ima svoj žeton na mjestu (i, j) ploče za igru, a na mjestu $(1, i, j)$ piše 1 ako i samo ako igrač koji nije na potezu ima svoj žeton na mjestu (i, j) ploče za igru. 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. Složenije igre, poput šaha, imale bi po jedan sloj za svaku vrstu figure (top, pješak, \dots) svake boje, te dodatne slojeve za specijalna stanja (na primjer, sloj pun jedinica ako je igrač na potezu napravio rokadu, a nula ako nije i slično). Nakon ulaznog sloja neuronske mreže slijede skriveni slojevi: prvi je konvolucijski sa 64 filtera, a zatim slijede 4 identična rezidualna bloka. Svaki blok se sastoji od po dva konvolucijska sloja sa 64 filtera, čiji izlaz se zbraja s ulazom u taj blok. Nakon toga se mreža račva u dvije grane: *value head*, čiji izlaz će biti $\hat{v}$, te *policy head*, čiji izlaz će biti $\hat{\pi}$. Ranije verzije algoritma AlphaZero su imale odvojene neuronske mreže za $\hat{v}$ i $\hat{\pi}$, no pokazalo se boljim i efikasnijim imati samo jednu mrežu—ideja je logiku igre pohraniti u slojevima prije grananja i nema potrebe te informacije dva puta odvojeno trenirati. Kao funkciju aktivacije u cijeloj mreži koristimo *LeakyReLU*, a nakon svakog konvolucijskog sloja se radi normalizacija *batch*-a (treniranje se provodi odjednom za više primjera (s, π, v) koje zajednički nazivamo *batch*). Za dodatne informacije oko raznih tipova slojeva, aktivicijskih funkcija, kao i ostalih bitnih odluka koje treba donijeti kod dizajna neuronskih mreža čitatelja upućujemo na izvrsnu knjigu [6].


input: InputLayer
input: [(?, 2, 6, 7)]
output: [(?, 2, 6, 7)]
conv2d_11: Conv2D
input: (?, 2, 6, 7)
output: (?, 64, 6, 7)
batch_normalization_11: BatchNormalization
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
leaky_re_lu_12: LeakyReLU
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
×4
conv2d_12: Conv2D
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
batch_normalization_12: BatchNormalization
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
leaky_re_lu_13: LeakyReLU
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
conv2d_13: Conv2D
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
batch_normalization_13: BatchNormalization
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
add_4: Add
input: [(?, 64, 6, 7), (?, 64, 6, 7)]
output: (?, 64, 6, 7)
leaky_re_lu_14: LeakyReLU
input: (?, 64, 6, 7)
output: (?, 64, 6, 7)
conv2d_20: Conv2D
input: (?, 64, 6, 7)
output: (?, 1, 6, 7)
conv2d_21: Conv2D
input: (?, 64, 6, 7)
output: (?, 2, 6, 7)
batch_normalization_20: BatchNormalization
input: (?, 1, 6, 7)
output: (?, 1, 6, 7)
batch_normalization_21: BatchNormalization
input: (?, 2, 6, 7)
output: (?, 2, 6, 7)
leaky_re_lu_21: LeakyReLU
input: (?, 1, 6, 7)
output: (?, 1, 6, 7)
leaky_re_lu_23: LeakyReLU
input: (?, 2, 6, 7)
output: (?, 2, 6, 7)
flatten_2: Flatten
input: (?, 1, 6, 7)
output: (?, 42)
flatten_3: Flatten
input: (?, 2, 6, 7)
output: (?, 84)
dense_1: Dense
input: (?, 42)
output: (?, 20)
policy_head: Dense
input: (?, 84)
output: (?, 7)
leaky_re_lu_22: LeakyReLU
input: (?, 20)
output: (?, 20)
value_head: Dense
input: (?, 20)

Slika 9: Rezidualna konvolucijska neuronska mreža za igru *Connect4*. Svaki sloj je opisan nazivom koji mu je dodijelila biblioteka Keras (na primjer, conv2d_11), tipom sloja (na primjer, Conv2D za 2D-konvolucijski sloj), te dimenzijama ulaznog i izlaznog tenzora (na primjer, (?, 2, 6, 7) za tenzor reda 4 čija je prva dimenzija proizvoljna (veličina *batcha*), a preostale su redom 2, 6 i 7). Niz slojeva unutar iscrtkanog pravokutnika ponavlja se 4 puta.

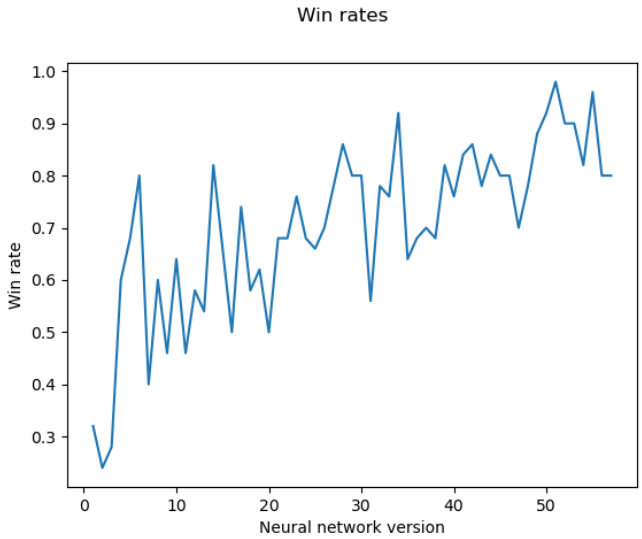
Proces treniranja izvršavan je na računalnom klasteru Isabella smještenom u Sveučilišnom računskom centru Sveučilišta u Zagrebu (SRCE). Isabella se sastoji od 135 računalnih čvorova, koji sadrže 3100 procesorskih jezgri i 12 grafičkih procesora, te je zajednički resurs svih znanstvenika u Hrvatskoj. Svaki proces učenja izvodili smo samo na jednom specijaliziranom čvoru tipa Dell EMC PowerEdge C4140, pomoću jednog od četiri dostupna grafička procesora NVIDIA Tesla V100-SXM2-16GB. Program je izvršavan nekoliko dana (oko 100 sati), a tijekom izvršavanja obavljeno je 150 iteracija algoritma.

Tijekom učenja težine modela pohranjuju se svaki put kad se neuronska mreža dobivena treniranjem unutar pojedine iteracije pokaže boljom od najuspješnije neuronske mreže. Dvije mreže, odnosno, pripadne agente, uspoređujemo tako da odigraju jedna protiv druge određen broj igara, a novu mrežu proglašavamo boljom ako pobijedi dosad najbolju mrežu u 55% tih igara. Na taj način smo nakon 150 iteracija dobili niz od ukupno 57 neuronskih mreža (i pridruženih agenata) stvorenih u različitim trenucima učenja, koji stoga imaju različitu uspješnost u igranju igre. Očekujemo da su agenti čije su neuronske mreže nastale u kasnijim trenucima učenja uspješniji od onih kojima odgovaraju ranije neuronske mreže.

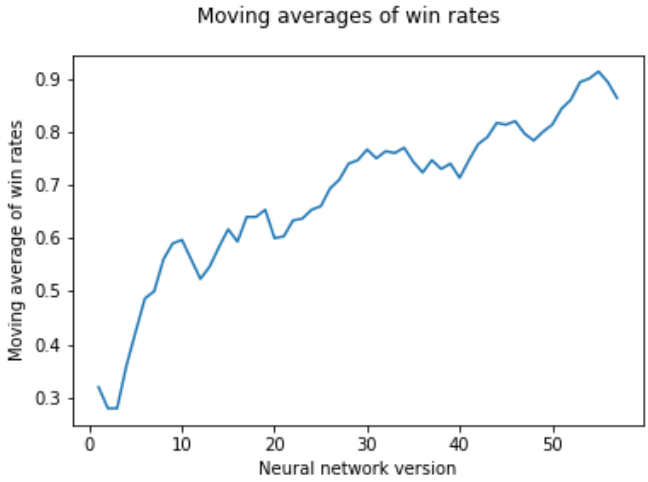
Napredak tijekom učenja praćen je mjerenjem uspješnosti pojedinih agenata u igrama protiv sljedećih algoritama za igranje igre *Connect Four*:

- 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;
- *minimax-*ℓ, za razne vrijednosti ℓ, čiju implementaciju smo preuzeli iz [5];
- agenti nastali u drugim iteracijama treniranja algoritma AlphaZero.

Slika 10 prikazuje napredak u uspješnosti naših agenata naspram algoritma *minimax-3*. Ovdje održavamo "turnir" u kojemu svaka pojedina verzija agenta (odnosno, svaka pojedina verzija neuronske mreže—na osi apscise) igra 50 igara protiv *minimax* agenta. Uspješnost mjerimo omjerom broja pobjeda pojedinog našeg agenta i ukupnog broja odigranih igara—taj omjer je prikazan na osi ordinate slike 10. Kako bismo bolje vizualizirali i naglasili dugoročne trendove te "izgladili" kratkoročne fluktuacije, na slici 10 prikazujemo takozvane pokretne prosjeke (engl. *moving average*) udjela pobjeda. Za svaku verziju neuronske mreže (osim prvih pet) računamo srednju vrijednost mjera uspješnosti pridruženih njoj te pet prethodnih verzija (za prvih pet verzija računamo srednju vrijednost udjela pobjeda te i svih prethodnih neuronskih mreža). Na toj slici možemo vidjeti da zaista postoji trend povećanja udjela pobjeda pojedinih agenata, odnosno da agenti postaju sve uspješniji u igrama protiv *minimax* agenta. 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 odgovaraju 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.



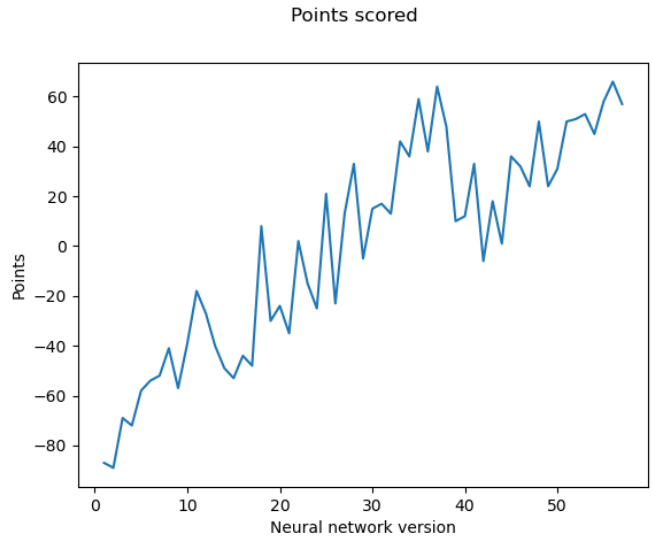
a: Uspješnost pojedinih AlphaZero agenata protiv algoritma *minimax-3*.



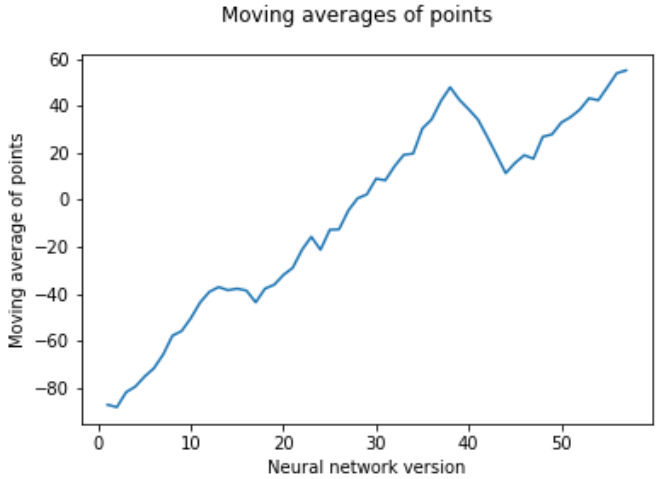
b: Pokretni prosjeci uspješnosti pojedinih AlphaZero agenata protiv algoritma *minimax-3*.

Slika 10: Turniri AlphaZero agenata protiv algoritma *minimax-3*.

Napredak tijekom učenja utvrđujemo i promatranjem rezultata igara u kojima suprotstavljamo sve verzije AlphaZero agenata jedne protiv drugih. Održavamo "turnir" u kojemu svaki AlphaZero agent igra 2 igre protiv svakog drugog. Sada svakom agentu za svaku pobjedu dajemo jedan bod, a za svaki poraz oduzimamo jedan bod. Budući da imamo ukupno 57 AlphaZero agenata, maksimalan broj bodova koji je moguće ostvariti je 112, a minimalan -112. Na slici 13 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. Ponovno, radi boljeg uočavanja dugoročnih trendova na slici 13 prikazujemo pokretne prosjeke brojeva bodova pojedinih agenata. Očito, broj ostvarenih bodova raste kako treniranje odmiče.



a: Brojevi bodova pojedinih AlphaZero agenata u igrama protiv svih drugih.



b: Pokretni prosjeci brojeva bodova pojedinih AlphaZero agenata u igrama protiv svih drugih.

Slika 13: Turniri u kojima sudjeluju sve varijante AlphaZero agenata.

Konačno, uspješnost raznih agenata za igru *Connect Four* uspoređujemo ispunjavanjem tablice 1 na sljedeći način. Svaki od 7 odabranih algoritama odigrao je turnir od 50 igara sa svakim od preostalih 6. U pripadnom retku tablice piše koliko pobjeda je taj agent ostvario protiv agenta iz odgovarajućeg stupca. Dok agenta s pridruženom prvom verzijom neuronske mreže nadmoćno pobjeđuju i *minimax-3* i *minimax-4*, agent kojemu pripada posljednja verzija neuronske mreže pobjeđuje u čak 42 igre protiv *minimax-4*. Zanimljivo je uočiti i trend napretka algoritama *minimax-*ℓ s rastućim ℓ; no već za $\ell = 5$ vrijeme njihovo izvršavanja postaje preveliko. Također, ponovno možemo uočiti sličan trend kakav su pokazale slike 13 i 13, odnosno, da se agenti kojima su pridružene kasnije verzije neuronskih mreža pokazuju uspješnijima u igrama protiv agenata s pripadnim ranijim verzijama.

Tablica 1: Tablica ostvarenih brojeva pobjeda pojedinih agenata u igrama protiv svih ostalih.

	Nasumični	*minimax-1*	*minimax-3*	*minimax-4*	1. NN	20. NN	57. NN
Nasumični		0	1	0	1	0	0
minimax-1	50		8	1	18	3	1
minimax-3	49	41		7	36	10	3
minimax-4	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	

5 Zaključak

U ovom članku smo opisali arhitekturu algoritma *AlphaZero*, koja je svojim spojem Monte Carlo stabla pretraživanja i duboke neuronske mreže donijela revoluciju u sposobnosti umjetne inteligencije za igranje kombinatornih igara. Iako je igra *Connect Four* na kojoj smo ilustrirali ovaj algoritam jednostavna u usporedbi s šahom ili igrom Go, a računalni hardver na kojem je izvršeno treniranje za red veličine slabiji od onog pomoću kojeg je *DeepMind* [14, 12] trenirao svoje algoritme za šah i Go, osnovna ideja algoritma, kao i njegova implementacija je posve ista—i primjenjiva za praktički bilo koju kombinatornu igru. Jedina informacija koju *AlphaZero* algoritmi dobivaju jesu pravila igre, odnosno, popis dozvoljenih akcija u danom trenutku. Idući korak bi bilo uklanjanje i te informacije, te davanje potpune slobode algoritmu da istraživanjem otkrije i njih te ovlada okruženjem čija dinamika mu je posve nepoznata: algoritam *MuZero* [11]. {10}

Bibliografija

[1] The on-line encyclopedia of integer sequences: Number of legal 7 x 6 Connect-Four positions after n plies. https://oeis.org/A212693.

[2] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games, 4:1–43, 2012.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] Alexis Cook. Intro to game AI and reinforcement learning, lesson 3: N-step lookahead. https://www.kaggle.com/alexisbcook/n-step-lookahead .

[6] Aurélien Géron. Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, Inc., 2nd edition, 2019.

[7] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In J. Furnkranz, T. Scheffer, and M. Spiliopoulou, editors, Machine Learning: ECML 2006, pages 282–293. Springer, Berlin, Heidelberg, 2006.

[8] L. Kocsis, C. Szepesvári, and J. Willemson. Improved Monte-Carlo search. Univ. Tartu, Estonia, Tech. Rep. 1, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.374.1202&rep=re... , 2006.

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

[10] Jelena Lončar. AlphaZero: strojno učenje podrškom bez domenskog znanja. Diplomski rad, PMF - Matematički odsjek, 2020.

[11] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. Nature, 588:604–609, 2020.

[12] 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, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[13] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362:1140–1144, 12 2018.

[14] 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, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

ISSN 1334-6083