UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Računalništvo in matematika – 2. stopnja

Matej Vitek

FighterZero: Pristop samo-igranja za učenje igranja pretepaške igre z globokim spodbujevalnim učenjem

Magistrsko delo

Mentor: izr. prof. dr. Peter Peer

Ljubljana, 2018

Zahvala

Na tem mestu bi se najprej zahvalil mentorju izr. prof. dr. Petru Peeru za čas in nasvete, ki mi jih je namenil pri izdelavi aplikacije in pisnega dela magistrske naloge ter tudi za vse dosedanje (in nadaljnje) sodelovanje pri različnih projektih.

Poleg tega bi se zahvalil doktorskima študentoma Žigu Emeršiču in Blažu Medenu za pomoč pri uporabi knjižnic Keras in TensorFlow ter uporabi laboratorijske opreme.

Nazadnje pa bi se zahvalil še očetu in mami za spodbudo pri izdelavi magistrskega dela.

Kazalo

Kazalo slik

Kazalo tabel

Kazalo algoritmov

Seznam uporabljenih kratic

kratica	angleško	slovensko
(A)NN	(artificial) neural network	nevronska mreža
CNN	convolutional neural network	konvolucijska nevronska mreža
RL	reinforcement learning	spodbujevalno učenje
MCTS	Monte Carlo tree search	drevesno preiskovanje Monte Carlo
DQN	Deep Q-Network	globoka Q-mreža
HP	hit/health points	točke zdravja
FPS	frames per second	sličice na sekundo
CPU	central processing unit	centralna procesna enota
GPU	graphics processing unit	grafična procesna enota

Program dela

Implementirajte učljivega agenta, ki se bo s samo-igranjem naučil igrati pretepaško igro za dva igralca.

Najprej poiščite primerno ogrodje (igro), ki bo omogočala implementacijo takega agenta. Po potrebi v ogrodje dodajte funkcionalnosti, ki jih boste tekom razvoja agenta potrebovali.

Nato se lotite implementacije agenta. Za osnovo naj vam služijo agenti AlphaGo [52], AlphaGo Zero [54] in AlphaZero [53], ki jih je za namizne igre razvila Googlova ekipa DeepMind.

Odločiti se boste morali za strukturo nevronske mreže, ki jo prilagodite obliki podatkov, ki jih boste iz ogrodja prejemali. Verjetno bo tu šlo za slikovno informacijo – v tem primeru lahko uporabite podobno mrežo, kot jo uporablja AlphaZero. Nato to nevronsko mrežo implementirajte z uporabo primernih knjižnic za globoko učenje. Pazite na časovno zahtevnost, namreč v pretepaških igrah boste imeli na voljo za odločanje precej manj časa kot AlphaZero – temu primerno načrtujte velikost nevronske mreže. Prirediti boste morali tudi algoritem za spodbujevalno učenje, da bo primeren za pretepaške igre. Pazite, da ogrodje omogoča simuliranje prihodnjih stanj igre, saj je simulacija potrebna za delovanje drevesnega preiskovanja Monte Carlo. Če takega simulatorja v ogrodju ni, ga boste morali implementirati sami.

Več primerov uporabe globokega spodbujevalnega učenja (ki pa, razen agentov Alpha, ne vsebujejo samo-igranja – le-to je namreč precejšnja novost v literaturi) lahko najdete v zbirki literature spodaj.

Osnovna literatura

[53] D. Silver in dr., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, ArXiv e-prints (2017)

[54] D. Silver in dr., *Mastering the game of Go without human knowledge*, Nature **550**(7676) (2017) 354–359, doi: 10.1038/nature24270

[52] D. Silver in dr., *Mastering the game of Go with deep neural networks and tree search*, Nature **529** (2016) 484–489

[39] V. Mnih in dr., *Human-level control through deep reinforcement learning*, Nature **518** (2015) 529 EP

[23] X. Guo in dr., *Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning*, v: Advances in Neural Information Pro-

cessing Systems 27 (ur. Z. Ghahramani in dr.), Curran Associates, Inc., 2014, str. 3338–3346

[69] S. Yoon in K.-J. Kim, *Deep Q networks for visual fighting game AI*, v: 2017 IEEE Conference on Computational Intelligence and Games (CIG), 2017, str. 306–308, doi: 10.1109/CIG.2017.8080451

[43] E. Perot in dr., *End-to-End Driving in a Realistic Racing Game with Deep Reinforcement Learning*, v: 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2017, str. 474–475, doi: 10.1109/CVPRW.2017.64

[49] P. B. S. Serafim in dr., *Towards Playing a 3D First-Person Shooter Game Using a Classification Deep Neural Network Architecture*, v: 2017 19th Symposium on Virtual and Augmented Reality (SVR), 2017, str. 120–126, doi: 10.1109/SVR.2017.24

[27] N. Justesen in S. Risi, *Learning macromanagement in starcraft from replays using deep learning*, v: 2017 IEEE Conference on Computational Intelligence and Games (CIG), 2017, str. 162–169, doi: 10.1109/CIG.2017.8080430

[56] M. Stanescu in dr., *Evaluating real-time strategy game states using convolutional neural networks*, v: 2016 IEEE Conference on Computational Intelligence and Games (CIG), 2016, str. 1–7, doi: 10.1109/CIG.2016.7860439

Podpis mentorja:

Izvleček

Naslov: FighterZero: Pristop samo-igranja za učenje igranja pretepaške igre z globokim spodbujevalnim učenjem

Področje globokega učenja je v zadnjem desetletju doživelo precejšen razcvet. Uporablja se za reševanje premnogih problemov, v zadnjih petih letih pa precej tudi za igranje iger. Dva pomembna dosežka sta bila globoke Q-mreže (DQN) in AlphaZero. DQN se je naučila igrati klasične igre za Atari 2600 (Pong, Space Invaders, itd.), AlphaZero pa se je s samo-igranjem naučil igrati šah, šogi in Go. Mi smo na temelju AlphaZero poskusili zgraditi agenta FighterZero, ki bi se prav tako s samo-igranjem naučil igrati pretepaške računalniške igre. Rezultati so bili manj uspešni, kot smo pričakovali, saj se je časovna zahtevnost izkazala za nepremagljivo oviro.

Math. Subj. Class. (2010): 68T05
Ključne besede: umetna inteligenca, inteligentni agent, igre, samo-igranje, globoko učenje, spodbujevalno učenje, drevesno preiskovanje Monte Carlo, nevronske mreže, razvoj iger

Abstract

Title: FighterZero: a Self-Playing Deep Reinforcement Learning Agent for Fighting Game AI

Deep learning has been a field of great academic interest and substantial breakthroughs over the last decade. Its applications are many and over the last five years it has spread also to the field of game playing, owing largely to two chief accomplishments of Google's DeepMind team: Deep Q-Networks (DQN), which learned to play classic Atari 2600 games, and AlphaZero, which learned, strictly through self-play, to play the board games chess, shogi and Go. In this thesis we attempted to build on the success of AlphaZero by adapting its self-playing architecture to fighting games, a popular genre of video games. The results were, however, less successful than we had expected and hoped, as the time constraints proved to be an insurmountable obstacle.

Math. Subj. Class. (2010): 68T05
Keywords: artificial intelligence, intelligent agent, games, self-playing, deep learning, reinforcement learning, Monte Carlo tree search, neural networks, game development

Poglavje 1

Uvod

Umetna inteligenca je področje, ki se je začelo razvijati v petdesetih letih prejšnjega stoletja, uporablja pa se v številne zelo raznolike namene. Tako pristope umetne inteligence najdemo v medicini, v gospodarstvu, v robotiki, v matematiki, v biometriji, ...

V zadnjem desetletju pa se je precej razširilo predvsem področje globokega učenja. K temu so največ pripomogli precejšnji napredki v hitrosti in sposobnosti strojne opreme, ki je na voljo. Algoritmi globokega učenja so sicer, vsaj v osnovni obliki, obstajali že pred desetletji, vendar pa s takratno strojno opremo niso bili zmožni doseči dobrih rezultatov na področjih, kjer se uporabljajo danes. Globoko učenje v zadnjih letih na mnogih problemih dosega boljše dosežke kot katerikoli drugi pristopi. Uporablja se za prepoznavo govora [22, 24, 70], računalniški vid [13], procesiranje naravnega jezika [7, 20, 67], v priporočilnih algoritmih [14], v bioinformatiki [10], ...

Mi se v tem delu osredotočamo na uporabo globokega učenja za igranje iger. V ta namen se umetna inteligenca uporablja že prav od začetka. Tako je že v petdesetih letih prejšnjega stoletja Arthur Samuel razvil program za damo, ki je nekoliko kasneje igro igral na nivoju zmerno dobrega človeškega igralca [47]. Še danes je igranje iger eno najbolj razširjenih področij v umetni inteligenci.

V tem delu razvijemo agenta, ki se sam nauči igrati pretepaško igro. Naš agent temelji na enem od izrednih dosežkov umetne inteligence v zadnjem desetletju: DeepMindovem agentu AlphaZero, ki se je sam naučil igrati šah, šogi in Go ter v vseh treh igrah premagal dosedanje najboljše računalniške agente razvite za vsako od teh treh iger posebej. Tako je AlphaZero pri vseh treh igrah dosegel nadčloveški nivo igranja.

Pretepaške igre so (za razliko od večine preostalih priljubljenih zvrsti računalniških iger) namenjene boju med dvema igralcema, obenem pa so deterministične (angl. *deterministic* – to pomeni, da trenutno stanje okolja in izbira akcije enolično določata naslednje stanje [45]) in popolnoma vidne (angl. *fully observable* – agentu je v vsakem stanju na voljo informacija o celotnem stanju okolja [45]). Vse te lastnosti spominjajo na prej omenjene igre za igralno desko šah, šogi in Go in res so prav zaradi teh lastnosti pretepaške igre primerne za priredbo agenta AlphaZero.

1.1 Struktura naloge

V 2. poglavju pregledamo pristope globokega učenja, ki so se predvsem v zadnjih petih letih uporabljali tako za igranje iger kot tudi za reševanje sorodnih problemov.

V 3. poglavju predstavimo ogrodje, ki ga uporabljamo za implementacijo našega agenta. Natančneje opišemo uporabljeno igro FightingICE ter na kratko povemo, kaj smo igri dodali za lažjo uporabo v raziskavah.

V 4. poglavju predstavimo glavne tri metode, ki jih naš algoritem uporablja: nevronske mreže, spodbujevalno učenje in drevesno preiskovanje Monte Carlo. Pri vsaki od metod na kratko razložimo glavno idejo metode, pri čemer si pomagamo z diagrami, in podamo literaturo za bolj podroben pregled. Nato povemo, kako te tri metode združimo v končni algoritem FighterZero, pokažemo pa tudi psevdokodo celotnega algoritma. Opišemo tudi arhitekturi obeh mrež, ki ju verziji našega algoritma uporabljata. Nadaljujemo s pregledom težav, do katerih je prišlo pri implementaciji in testiranju algoritma ter opišemo, kako smo se lotili reševanja teh težav. Poglavje končamo z opisom postopka testiranja – opišemo, kakšne nastavitve ogrodja smo uporabili in agente, ki smo jih v testiranju uporabili kot nasprotnike.

V 5. poglavju v tabelni obliki prikažemo rezultate, ki jih je naš agent dosegel proti testnim agentom. Rezultate nato analiziramo in pojasnimo nekaj podrobnosti.

V 6. poglavju naredimo pregled celotne naloge. Še enkrat omenimo težave, s katerimi smo se srečali pri razvoju agenta in kako uspešne so bile rešitve, ki smo jih uporabili. Omenimo, katerih rešitev nam ni uspelo implementirati zaradi pomanjkljivosti ogrodja in premislimo, kako bi se agent odrezal ob drugačni izbiri igre in pripadajočega ogrodja ter programskega jezika. Na koncu predlagamo tudi nekaj izboljšav, ki bi morda pripomogle k boljšemu delovanju agenta, če bi uspeli razrešiti težave s časovno zahtevnostjo.

1.2 Licenca

Izvorna koda celotne aplikacije je objavljena na spletnem repozitoriju [62].

Del kode, ki predstavlja ogrodje in igro FightingICE, je last laboratorija Intelligent Computer Entertainment (ICE) Lab., Ritsumeikan University. Vse pravice in lastništvo za ta del kode pripadajo izključno temu laboratoriju. Koda ogrodja je na spletni repozitorij naložena zaradi manjših sprememb, ki so natančneje opisane v razdelku 3.1. Te spremembe so v skladu z dovoljenjem za uporabo v raziskovalne namene, ki ga laboratorij ICE podaja na svoji spletni strani [26].

Izvorna koda agenta FighterZero je originalna lastnina avtorja tega dela. Zaščitena je z licenco GNU General Public Licence v3, ki omogoča prosto distribucijo in predelavo pod določenimi pogoji. Podrobnosti licence so dostopne na uradni spletni strani licence GNU GPL [21].

Natančnejša informacija o tem, kateri del kode na repozitoriju pripada ogrodju in kateri našemu agentu, je na voljo v datoteki `readme.txt`, ki se prav tako nahaja na spletnem repozitoriju [62].

Poglavje 2

Pregled področja

Uporaba umetne inteligence za igranje računalniških iger je v zadnjih dveh desetletjih eno širših področij umetne inteligence tako v industriji kot tudi v akademskih raziskavah. V zadnjih petih letih pa se je (predvsem na akademski strani) začelo v ta namen uporabljati tudi globoko spodbujevalno učenje, ki je v umetni inteligenci relativno mlado področje.

Prva uspešna implementacija globokega spodbujevalnega učenja za igranje iger je prišla leta 2013, ko je Googlova ekipa DeepMind globoko Q-mrežo (angl. *deep Q-network*, DQN) naučila igrati klasične igre za Atari 2600 (med njimi Pong, Seaquest in Space Invaders) [38, 39].

DeepMindov DQN sicer temelji na še enem pomembnem dosežku. Že leta 1995 je G. Tesauro [60] uspešno uporabil kombinacijo spodbujevalnega učenja in nevronske mreže za implementacijo agenta za igranje igre backgammon. Ta dosežek pa je za nas pomemben tudi, ker je uporabljal pristop samo-igranja, ki ga za učenje uporablja (med drugimi) tudi naš algoritem.

Leta 2014 je nato ekipa z michiganske univerze nadgradila DQN s tremi pristopi drevesnega preiskovanja Monte Carlo (angl. *Monte Carlo tree search*, MCTS), ki so pomagali pri učenju mreže [23]. MCTS se v njihovem pristopu uporablja le za učenje, dejansko igranje pa potem izvaja mreža sama. Ta pristop je opazno izboljšal rezultate DQN.

Januarja 2016 je ekipa DeepMind presenetila svet s svojim agentom AlphaGo [52], ki je v igri Go premagal evropskega prvaka Fana Huija z rezultatom 5-0. S tem je postal prvi računalniški program, ki je v igri Go premagal profesionalnega igralca – to je bil dosežek, ki se mu do tedaj ni niti približal katerikoli računalniški agent. Nekoliko kasneje (marca 2016) je rahlo spremenjena verzija z imenom AlphaGo Lee s 4-1 premagala še Leeja Sedola, zmagovalca 18 mednarodnih naslovov. Še kasneje (januarja 2017) pa je novejši AlphaGo Master premagal takrat najmočnejše človeške igralce z rezultatom 60-0 [54]. Podobno kot v [23] je tudi v AlphaGo globoko spodbujevalno učenje združeno z MCTS. Vendar pa se v AlphaGo MCTS uporablja tudi pri samem igranju, ne le pri učenju mreže. AlphaGo uporablja dve mreži, ki se naučita napovedati strategijo in vrednost stanj. Za učenje mreže strategije so najprej uporabili nadzorovano učenje na primerih potez strokovnjakov iz preteklih iger, nato pa so obe mreži naučili še s spodbujevalnim učenjem s samo-igranjem.

Leta 2017 je DeepMind nato AlphaGo nadgradil v AlphaGo Zero [54], ki je vse

prejšnje verzije agenta AlphaGo premagal prepričljivo. Glavna razlika med agentoma je, da se je AlphaGo Zero igro Go naučil igrati povsem brez kakršnegakoli predznanja (razen pravil igre). Učenje je torej v celoti realizirano s spodbujevalnim učenjem s samo-igranjem brez uporabe preteklih iger človeških strokovnjakov. Poleg tega sta mreži strategije in vrednosti združeni v eno, saj imata sedaj enoten postopek učenja. Nadalje je bil vhod v obe mreži v AlphaGo sestavljen iz 84 ročno določenih atributov za vsako pozicijo na igralni deski (teh je v profesionalni verziji igre Go $19 \times 19 = 361$). Skupaj je bil torej vhod v mreži velikosti $19 \times 19 \times 84 = 30324$. AlphaGo Zero pa uporablja le trenutno stanje igralne deske (bolj natančno 8 zadnjih stanj in informacijo o tem, kateri igralec je na potezi), kar zahteva vhod velikosti $19 \times 19 \times 17 = 6137$. Ta pristop za razliko od AlphaGo torej ne uporablja nikakršne človeške interpretacije pozicije. Nekaj implementacijskih razlik je tudi v MCTS, ki je v AlphaGo Zero nekoliko poenostavljen.

Proti koncu leta 2017 pa je DeepMind svoj algoritem posplošil še na igri šah in šogi in predstavil splošnega agenta AlphaZero [51, 53]. Ta je v vsaki od teh treh iger premagal do tedaj najboljše računalniške igralce: v Goju je premagal AlphaGo in AlphaGo Zero, v šogiju je premagal program Elmo (zmagovalec svetovnega prvenstva CSA 2017), v šahu pa je premagal program Stockfish (zmagovalec svetovnega prvenstva TCEC 2016). V AlphaZero je dodana možnost izenačenih izidov (in bolj splošno kakršnihkoli izidov, ki omogočajo točkovanje). Poleg tega se AlphaZero ne zanaša več na invariantnost igre na rotacije in zrcaljenje. V igri Go taka invariantnost velja, v šahu in šogiju (in bolj splošno v igrah) pa ne. Nadalje je razlika tudi v postopku učenja. V AlphaGo Zero je bila shranjena najboljša mreža, ki jo je po vsakem koraku učenja nova mreža nadomestila le, če je proti njej zmagala vsaj 55% medsebojnih iger. V AlphaZero se shranjuje le ena mreža, ki se uči brez kakršnegakoli sprotnega preverjanja uspešnosti proti prejšnjim mrežam. Nazadnje pa je bila razlika tudi v prilagajanju parametrov mreže. V AlphaGo Zero so se parametri tekom učenja prilagajali s postopkom bayesovske optimizacije, v AlphaZero pa parametri ostanejo enaki čez celoten postopek učenja.

Poleg dosežkov DeepMinda pa se je tudi v splošnem v zadnjih treh letih precej razcvetela uporaba globokega učenja za igranje iger. Kar nekaj del na to temo uporablja podobne pristope, kot jih je uporabila ekipa DeepMind, nekateri pa poskušajo tudi z drugačnimi pristopi globokega učenja. Tako najdemo npr. uporabo DQN za igranje igre Pac-Man [44], iger za Atari 2600 [18], igre FlappyBird [3] in Tetrisa [57]. Poskus nadgradnje AlphaGo najdemo v [71] in v [33]. HearthBot [50] uporablja prilagodljive nevronske mreže (angl. *adaptive neural networks*) za igranje spletne igre s kartami Hearthstone. Prav tako v literaturi najdemo uporabo globokega učenja za igranje 3D dirkalnih iger [43]. Kakršnokoli 3D okolje predstavlja nekoliko težji problem, saj dobimo iz okolja več dimenzij podatkov. Za igranje 3D prvoosebne streljaške igre zato zaenkrat obstaja le nekaj delnih agentov, ki se posvetijo posameznim problemom, ki jih agent v takem okolju sreča, ne pa celotni igri [49, 63]. Za še nekoliko težje področje pa se izkažejo realno-časovne strateške igre. Za realno-časovno igro Starcraft najdemo dve deli, ki naredita nekaj korakov proti razvoju celotnega agenta z globokim učenjem [27, 56].

Nekaj literature obstaja tudi na temo nadgrajevanja DQN, da bi dosegli bolj človeško obnašanje agenta [37]. V igrah pogosto želimo, da agent sicer predsta-

vlja izziv, ni pa nepremagljiv. V tem delu tako obnašanje dosežejo z združenjem spodbujevalnega in nadzorovanega učenja, za evaluacijo pa uporabijo tako dosežene točke kot tudi Turingov test, ki se najpogosteje uporablja za merjenje "človeškosti" agentov.

Najbližja našemu delu pa je uporaba DQN za igranje pretepaških iger [69]. Vendar niti to delo niti kakšno od zgoraj naštetih (z izjemo nadgradenj AlphaGo [33, 71]) ne uporablja pristopa samo-igranja, ki ga je ekipa DeepMind uporabila v svojih agentih Alpha [52, 53, 54]. Uporaba le-tega je torej na področju pretepaških iger novost, ki je, kolikor nam je znano, v literaturi ne najdemo.

Poglavje 3

Ogrodje FightingICE

FightingICE [26] je pretepaška igra (angl. *fighting game*), ki so jo razvili v Intelligent Computer Entertainment (ICE) Lab., Ritsumeikan University. Razvita je bila v namen uporabe v raziskavah in na tekmovanju FighTing Game AI Competition (FTGAIC), ki ga prav tako organizira ta laboratorij v okviru IEEEjeve konference CIG [17].

Igralec igra enega od treh likov, ki so v igri na voljo: Zen, Garnet in Lud. Cilj igre je s premikanjem, udarci, branjenjem, protiudarci in posebnimi napadi premagati nasprotnika. Posamezni liki se med sabo razlikujejo po tipih udarcev, ki jih znajo izvesti, za lažjo uporabo v raziskavah pa so vhodne kombinacije (torej zaporedja pritiskov gumbov), ki jih te udarci zahtevajo, pri vseh treh enake. Za potrebe našega agenta smo se osredotočili le na en lik in tako so vse bitke, ki smo jih izvedli v tem delu, potekale v obliki Zen proti Zenu. Delo pa bi se sicer trivialno posplošilo tudi na preostala dva lika, saj so vhodne kombinacije edina informacija, ki jo agentu damo vnaprej, te pa so za vse like enake.

Ena igra je sestavljena iz treh rund, vsaka od rund pa traja 60 sekund, po katerih je zmagovalec runde igralec, ki je izgubil manj točk zdravja (angl. *hit points*, HP). Ogrodje omogoča tudi način omejenih točk zdravja, pri katerem igralca začneta z določenim številom HP (privzeta vrednost je 400) in se runda predčasno konča, če kateri od njiju izgubi vse. Po koncu runde zmagovalec dobi točko, poraženec pa nič, v primeru izenačenega izida pa dobita točko oba. Ob zaključku igre je zmagovalec tisti igralec, ki je zbral več točk – možen je torej tudi neodločen izid celotne igre.

Več informacij o poteku igre, likih in ogrodju samem je na voljo na spletni strani tekmovanja [26]. Na sliki 3.1 pa vidimo, kako igra izgleda. Polna pravokotnika zelene in oranžne barve predstavljata HP igralcev, nad njima pa vidimo še tekstovno informacijo o HP in energiji igralcev (energija je potrebna za posebne napade, pridobimo pa jo tako z napadanjem kot tudi z obrambo). Na vrhu pa je informacija o številki trenutne runde in preostanku časa v rundi. Prazna pravokotnika, ki obdajata lika, pa označujeta površino, ki jo lika zasedata z vidika logike igre. Izrisana sta le v pomoč programerju, medtem ko v slikovno informacijo, ki jo agenta prejmeta, nista vključena.

Ogrodje je implementirano v Javi, omogoča pa tudi uporabo Pythona preko ovojnice Py4J [16]. To možnost smo uporabili tudi mi, saj smo za globoko učenje

Slika 3.1: Igra FightingICE v načinu omejenih točk zdravja.

uporabljali knjižnici Keras [11] in TensorFlow [1], ki sta na voljo le v Pythonu[1].

3.1 Spremembe ogrodja

Za lažjo uporabo je bilo potrebno ogrodje popraviti in mu dodati nekaj novih možnosti. Tako obogateno ogrodje je na voljo na repozitoriju [62]. Taka uporaba in popravljanje ogrodja za raziskovalne namene sta v skladu z navodili, ki so podana na spletni strani ogrodja [26] (več o tem v razdelku 1.2).

V ogrodje smo dodali možnost izbire naključnega lika. Naključna izbira se lahko izvede le ob začetku prve igre (iger namreč lahko poženemo več zapored), lahko pa tudi na začetku vsake. V našem delu smo na koncu sicer uporabljali le Zena, a je za morebitno posplošitev na preostala dva lika taka možnost naključne izbire v veliko pomoč.

Dodali smo možnost, da agente implementirane v Javi dodamo v ogrodje kar v kodi. Pred tem je bilo agente potrebno zapakirati v .jar datoteko in jih prenesti v pravilen direktorij, kar je lahko nerodno, če moramo to početi ob vsakem popravku, ki ga na agentu izvedemo.

Če želimo poganjati agente napisane v Pythonu, moramo najprej ogrodje pognati z zastavico `--py4j`. S tem se ogrodje zažene kot strežnik za ovojnico Py4J. Šele ko strežnik uspešno zaključi inicializacijo, lahko nanj povežemo pythonskega agenta. Postopku inicializacije smo zato ob zaključku dodali izpis sporočila. To omogoča

[1] TensorFlow ima sicer na voljo tudi javanski vmesnik, vendar pa Keras Jave ne podpira.

uporabo skripte za samodejen zagon pythonskega agenta takoj, ko se inicializacija strežnika konča. Tako skripto smo tudi implementirali.

Dodali smo možnost spreminjanja števila rund na igro in trajanja vsake runde z zastavicami `--rounds` in `--time`.

Implementirali smo tudi približek simulatorja slikovne informacije – več o tem v razdelku 4.4.4.

Pythonskim agentom smo dodali podporo za uporabo niti.

Poglavje 4

Metode

Naš agent temelji na agentu AlphaZero, ki je sestavljen iz treh različnih metod umetne inteligence: nevronske mreže, spodbujevalnega učenja in drevesnega preiskovanja Monte Carlo. V tem poglavju naredimo kratek pregled vsake od teh treh posameznih metod, nato pa si pogledamo, kako se združijo v končni algoritem in kako se naš algoritem razlikuje od osnovnega algoritma AlphaZero. Na koncu še opišemo težave, s katerimi smo se srečali pri implementaciji in rešitve, ki smo jih uporabili.

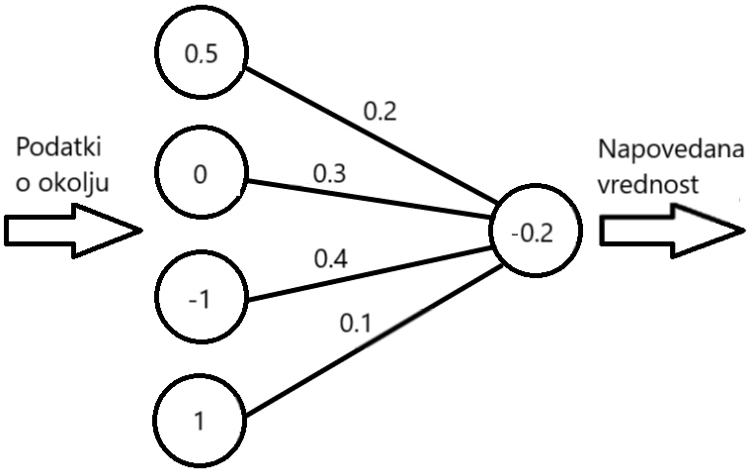
4.1 Nevronske mreže

Nevronske mreže (angl. *(artificial) neural networks*, (A)NN) [35, 66] so eno najbolj razširjenih področij strojnega učenja.

Nevronska mreža je v osnovni obliki utežen polni dvodelni graf. Sestavljena je iz vhodne in izhodne plasti (angl. *input/output layer*), ki sta sestavljeni iz vozlišč – tem v nevronskih mrežah pravimo nevroni. Na vhodno plast vnesemo podatke o trenutnem stanju (to je lahko npr. človekova telesna temperatura, teža, višina, starost, ...) v obliki t. i. oznak (angl. *labels*) – normiranih realnih števil. Na izhodni plasti pa na koncu dobimo vrednost, ki jo poskušamo napovedati (npr. ali je človek zdrav ali ne). Vhodni in izhodni nevroni so med sabo povezani z uteženimi povezavami, posamezne uteži na povezavah pa povejo, kakšen doprinos ima vrednost na vhodnem nevronu pri izračunu vrednosti izhodnega nevrona. Ko rečemo, da učimo nevronsko mrežo, to v resnici pomeni spreminjanje teh uteži tako, da nevronska mreža čim bolje napoveduje izhodno vrednost. Preprost primer nevronske mreže je prikazan na sliki 4.1.

Ker tako preproste mreže niso dovolj za reševanje težjih problemov, vstavimo med vhodno in izhodno plast še eno ali več skritih plasti (angl. *hidden layer*). Tako je vhodna plast polno povezana s prvo skrito plastjo, le-ta polno povezana z naslednjo in tako naprej do izhodne plasti. Na ta način zgradimo splošno nevronsko mrežo, ki je zmožna aproksimirati funkcijo, ki slika iz vhodnega vektorja v izhodni, ne glede na to, kakšna ta funkcija je (glej izrek 4.1.1 v [15]). Če je skritih plasti več, govorimo o globoki nevronski mreži (ta definicija sicer v literaturi ni enotna).

Globoko učenje poleg globokih nevronskih mrež sicer zajema še povratne nevronske mreže (angl. *recurrent neural networks*, RNN) [12, 36, 46], mreže globokega



Slika 4.1: Preprosta nevronska mreža. Na vhodni plasti so označene štiri vhodne vrednosti, povezave pa so označene z utežmi, ki jim pripadajo. Na izhodni plasti na koncu dobimo izhodno vrednost, ki je v tem primeru ena sama.

prepričanja (angl. *deep belief networks*) [6], globoke Boltzmannove stroje [6], ... Za natančnejšo definicijo globokega učenja, več arhitektur globokih mrež in več primerov uporabe le-teh je odličen pregled globokega učenja narejen v [48].

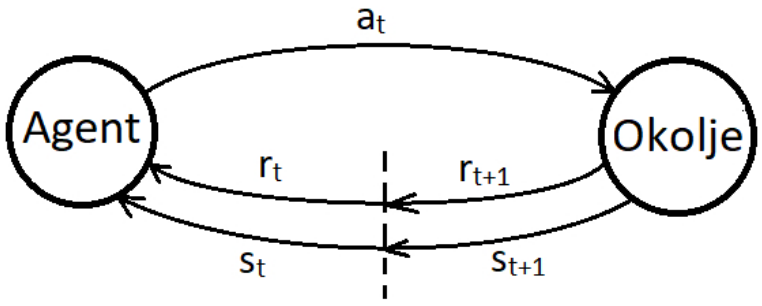
Konvolucijske nevronske mreže (angl. *convolutional neural networks*, CNN) [30, 72, 73] so posebna vrsta globokih nevronskih mrež, ki se predvsem uporablja za učenje na slikovnih podatkih. Prav tako kot zgoraj opisana splošna globoka nevronska mreža je tudi CNN sestavljena iz vhodne plasti, nekaj skritih plasti in izhodne plasti, ima pa precej manjše število povezav med sosednimi plastmi in precej manj uteži, ki se jih mora naučiti. Tako delujejo precej bolj učinkovito, kar je pri učenju iz slik pomembno, saj so ponavadi slikovni vhodi precej večji od podatkovnih.

Nevronske mreže se večinoma uporabljajo za nadzorovano učenje (angl. *supervised learning*), kjer se učijo iz že znanih, označenih primerov. Mi pa nevronske mreže uporabljamo v spodbujevalnem učenju.

4.2 Spodbujevalno učenje

Spodbujevalno učenje (angl. *reinforcement learning*, RL) [28] je še eno področje strojnega učenja. Ukvarja se z učenjem optimalnega obnašanja agenta v nekem danem okolju. Formalno se problem spodbujevalnega učenja opiše s sledečimi elementi:

- **Agent** je računalniški program, ki bi ga s spodbujevalnim učenjem radi naučili čim boljšega vedenja.
- **Okolje** je svet, v katerega je agent postavljen. Lahko gre za resnični svet ali pa virtualno okolje.
- **Množica stanj okolja** (in agenta v njem) $\mathcal{S} = \{s_i\}$. Informacije o teh stanjih agent prejema preko senzorjev. Lahko pa je v nekaterih (ali tudi vseh) stanjih



Slika 4.2: Cikel spodbujevalnega učenja.

zmožen pridobiti le delno informacijo.

- **Množica akcij**, ki so agentu na voljo $\mathcal{A} = \{a_i\}$ in funkcija pravil $\mathcal{R}\colon \mathcal{S} \times \mathcal{A} \to \{True, False\}$, ki nam pove, katere akcije so agentu na voljo v katerem stanju.
- **Nagrada** $r \in \mathbb{R}$ je takojšnja povratna informacija, ki jo agent dobi iz okolja ob izvedbi akcije. Nagrada je lahko pozitivna (izboljšanje stanja) ali negativna (poslabšanje stanja).

Cikel poteka takega problema vidimo na sliki 4.2. Agent ob času t od okolja prejme nagrado r_t (odziv na predhodno spremembo) in informacijo o trenutnem stanju s_t. Nato se odzove na informacijo z akcijo a_t in v naslednjem časovnem koraku spet dobi povratno informacijo o spremembi okolja in nagrado.

V različnih algoritmih spodbujevalnega učenja se uporabljajo še naslednje oznake:

- **Strategija** (angl. *policy*) $\pi\colon \mathcal{S} \to \mathcal{A}$ je agentova funkcija izbiranja akcij. Namesto v $a \in \mathcal{A}$ lahko slika tudi v verjetnostni vektor velikosti $|\mathcal{A}|$, v katerem potem vsak od elementov predstavlja verjetnost, da agent izbere pripadajočo akcijo.
- **Vrednost stanj** $V_\pi\colon \mathcal{S} \to \mathbb{R}$ je funkcija, ki nam ob strategiji π za vsako stanje pove njegovo pričakovano vrednost – torej kako zaželeno je tako stanje. Bolj formalno, $V_\pi(s) = \mathrm{E}\left[R \mid s, \pi\right]$, kjer je $R = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$ in je $\gamma \in [0, 1]$ parameter, ki se imenuje diskontni faktor (angl. *discount factor*), E pa označuje pričakovano vrednost. Diskontni faktor γ določa, kolikšen delež k vrednosti prispevajo prihodnje nagrade. Če je $\gamma = 0$, se upošteva le takojšnja nagrada, če pa je $\gamma = 1$, pa imajo vse prihodnje nagrade enak donos. R v tej definiciji predpostavlja, da akcije v prihodnosti še naprej izbiramo po strategiji π.
- **Q-vrednost** $Q_\pi\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ je funkcija, ki ob strategiji π za vsako stanje pove pričakovano vrednost vsake od možnih izbir akcije. Formalno torej $Q_\pi(s, a) = \mathrm{E}\left[R \mid s, a, \pi\right]$.

Če poznamo funkcijo V_π ali Q_π, lahko vedno izberemo najboljšo možno akcijo (pri V_π moramo sicer poznati še posledice naših akcij). Cilj spodbujevalnega učenja je torej, da čim bolje aproksimiramo eno izmed teh dveh funkcij. Večina najbolj priljubljenih metod se osredotoča na aproksimacijo Q-vrednosti. Ena izmed teh je

npr. Q-učenje (angl. *Q-learning*) [64, 65], ki deluje tako, da inkrementalno izvaja akcije. Ob časovnem koraku t v stanju s_t izbere akcijo a_t in nato izvede posodobitev:

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_{t-1}, a_{t-1}) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a)).$$

Parameter $\alpha \in [0, 1]$ se imenuje hitrost učenja.

V našem algoritmu Q-učenje združimo z drevesnim preiskovanjem Monte Carlo, s čimer dobimo primer metode Monte Carlo za spodbujevalno učenje. Za bolj podroben opis spodbujevalnega učenja v splošnem priporočamo [28] in [58]. Globoko spodbujevalno učenje je podrobno predstavljeno v [48] in [4].

4.3 Drevesno preiskovanje Monte Carlo

Drevesno preiskovanje Monte Carlo (angl. *Monte Carlo tree search*, MCTS) [9] je hevristično preiskovalni algoritem. V osnovni obliki se uporablja za igranje iger za 2 igralca, kjer igralca igrata eden proti drugemu. To so lahko prave igre (npr. šah, igre s kartami, računalniške igre, ...) ali pa bolj splošne (teoretske) igre po definiciji iz teorije iger. Obstajajo pa tudi razširitve algoritma za igre z več igralci [19, 40, 41, 68].

MCTS poteka v štirih korakih, ki jih vidimo tudi na sliki 4.3. Eni ponovitvi teh štirih korakov pravimo simulacija. Koraki so:

1. **Izbira vozlišča**: Od korena drevesa se sprehodimo do lista. Postopek vsake izbire otroka v sprehodu je odvisen od konkretne implementacije, najpogosteje pa se uporablja formula UCT [29]:

$$\frac{w_v}{n_v} + c\sqrt{\frac{\ln N}{n_v}}. \tag{4.1}$$

   V sprehodu vsakič izberemo otroka z maksimalno vrednostjo UCT. V vozlišču v oznaka w_v označuje število zmag, ki so bile do sedaj dosežene iz v, n_v pa število vseh simulacij, ki so bile do sedaj iz v izvedene. $\frac{w_v}{n_v}$ je torej delež zmag, ki smo jih iz v dosegli. Parameter c je konstanta raziskovanja, N pa pomeni število vseh simulacij, ki smo jih do sedaj izvedli.

   Tak postopek izbire doseže dobro ravnovesje med izbiranjem akcij, ki so že znane kot dobre in akcij, ki jih še nismo dobro raziskali. Konstanta raziskovanja c uravnava razmerje med tema dvema izbirama. Če je c visok, se večkrat izberejo še neraziskane akcije, če pa je nizek, pa se večkrat izberejo dobre akcije. V praksi se ponavadi vzame vrednost $\sqrt{2}$, izbiro pa potem izboljšamo empirično.

2. **Razširitev**: Če doseženi list predstavlja končno stanje, je simulacija zaključena – pridobimo informacijo o izidu in skočimo na 4. korak. Sicer list razširimo tako, da ustvarimo njegove naslednike z izvedbo vseh možnih akcij (odvisno od implementacije lahko tukaj ustvarimo tudi le enega naslednika ali pa jih ustvarimo nekaj).

Tukaj je morda vredno omeniti, da so vse izvedbe akcij del simulacije in jih v resnici ne izvajamo. Zato MCTS zahteva, da imamo za dano igro na voljo simulator, ki nam zna iz trenutnega stanja in izbrane akcije povedati, kakšno bo naslednje stanje.

Od ustvarjenih naslednikov nato izberemo enega. Postopek izbire je lahko hevristično, lahko pa povsem naključen.

3. **Evaluacija**: Izbranega naslednika nato ocenimo z metodo, ki je odvisna od izbranega algoritma. V osnovni obliki MCTS se uporablja naključno igranje (angl. *rollout, playout*), lahko pa le-to nadomestimo tudi s kako hevristično oceno stanja. V našem algoritmu se za evaluacijo stanja uporablja nevronska mreža.

4. **Vzvratno širjenje izida**: Dobljeno oceno izida na koncu razširimo nazaj po isti poti, kot smo jo prehodili v 1. koraku.

Bolj podroben pregled različnih izvedb MCTS je na voljo v [8].

4.4 Končni algoritem FighterZero

Naš algoritem FighterZero temelji na algoritmu AlphaZero [51, 52, 53, 54], ki ga je za šah, šogi in Go implementirala Googlova ekipa DeepMind. Ker ogrodje FightingICE omogoča dva načina pridobivanja informacij o stanju – podatkovni in slikovni – smo implementirali dva agenta, ki se med seboj razlikujeta v uporabljeni nevronski mreži.

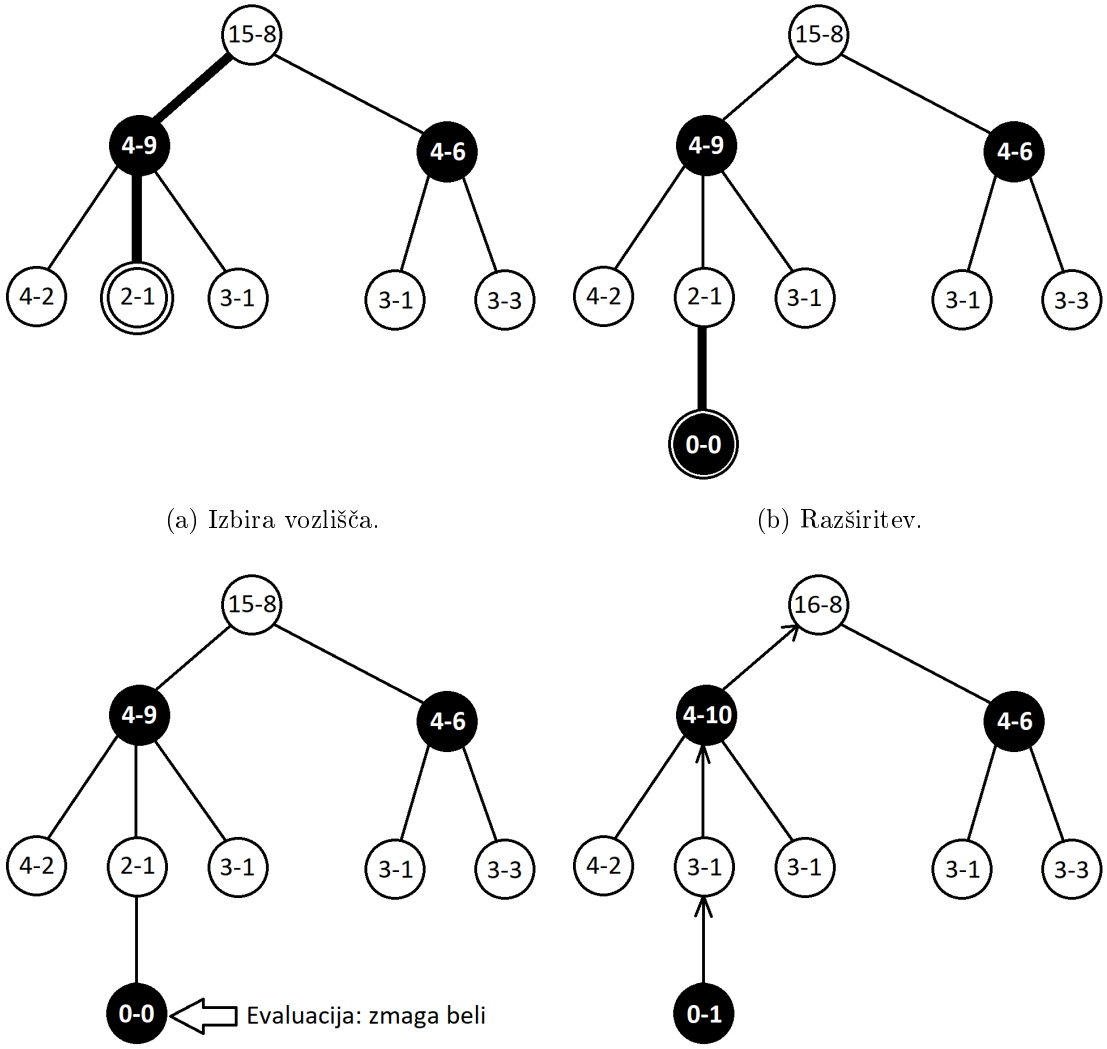
4.4.1 Nevronska mreža

Nevronska mreža, ki je v algoritmu uporabljena, dobi na vhod informacijo o stanju s. Ima dva izhoda: evaluacijo trenutnega stanja $v(s)$ (ali stanje vodi v zmago ali poraz) in strategijo $\vec{\boldsymbol{p}}(s)$ (iz razdelka 4.2 se spomnimo, da je to verjetnostni vektor nad množico možnih akcij).

Z $\mathcal{A}$ označimo množico vseh možnih akcij ($|\mathcal{A}| = 41$). Nevronski mreži, ki smo ju uporabili imata sledeči strukturi:

- **Podatkovna mreža**: Vhodna plast je velikosti ~300 (toliko podatkov se namreč da pridobiti o trenutnem stanju iz ogrodja). Med vhodno in izhodno plastjo so 4 polno povezane skrite plasti, ki po velikosti padajo po principu linearne interpolacije med vhodno velikostjo (~300) in izhodno velikostjo ($|\mathcal{A}|$). Za izhod strategije ima mreža izhodno plast velikosti $|\mathcal{A}|$. Za evaluacijo stanja ima še eno polno povezano skrito plast velikosti $\frac{|\mathcal{A}|}{2}$, nato pa izhodno plast velikosti 1. Arhitekturo te mreže vidimo na sliki 4.4a.

- **Slikovna mreža**: Gre za konvolucijsko mrežo preostankov (angl. *convolutional residual network*). Ta mreža temelji na tisti, ki jo uporablja tudi AlphaZero. Konvolucijske mreže preostankov so se namreč izkazale za izredno dobre pri analizi slik [25].



(a) Izbira vozlišča.

(b) Razširitev.

(c) Evaluacija.

(d) Vzvratno širjenje izida.

Slika 4.3: Cikel ene simulacije drevesnega preiskovanja Monte Carlo (za boljšo preglednost je drevo narisano le do globine 2). V vsakem vozlišču je zapisan trenutni rezultat dosežen iz tega vozlišča v obliki *zmage-porazi*. Opazimo, da se na vsakem nivoju rezultati obrnejo, saj algoritem vedno deluje z vidika igralca, ki je na potezi.

Vhodna plast je velikosti 96×64. Nato sledi začetna konvolucijska plast s konvolucijskim jedrom velikosti 7×7. Nato pridejo 4 plasti preostankov. Na koncu za izhod strategije pride še konvolucijska plast s konvolucijskim jedrom velikosti 1×1 in izhodna plast velikosti $|\mathcal{A}|$, za evaluacijo stanja pa prav tako konvolucijska plast z jedrom 1×1, nato še ena polno povezana plast velikosti $\frac{|\mathcal{A}|}{2}$ in na koncu izhodna plast velikosti 1. Ta mreža je prikazana na sliki 4.4b.

Plast preostankov (angl. *residual layer*) je sestavljena iz 2 konvolucijskih plasti z jedrom 3×3. Rezultat teh dveh konvolucij se na koncu prišteje vhodu, ki je prišel v plast preostankov.

V obeh mrežah se po vsaki od plasti izvede še normalizacija in aktivacija nevronov. Za preprečevanje prekomernega prileganja (angl. *overfitting*) pa smo uporabili odpadne plasti (angl. *dropout layers*) [55].

4.4.2 Izboljšanje strategije z MCTS

Nevronsko mrežo bi radi naučili čim bolje izbrati strategijo. To dosežemo z drevesnim preiskovanjem Monte Carlo. V drevesu, ki ga preiskujemo, vozlišča predstavljajo stanja, povezave med vozlišči pa akcije. Z s torej označimo vozlišče, s parom (s, a) pa povezavo. V koraku izbiranja vozlišča (glej razdelek 4.3) namesto UCT iz enačbe (4.1) uporabljamo zgornjo mejo Q-vrednosti, ki jo označimo z $U(s, a)$ in jo izračunamo po sledeči enačbi:

$$U(s,a) = Q(s,a) + c \cdot \vec{\boldsymbol{p}}(s)[a] \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}. \tag{4.2}$$

$Q(s, a)$ je tu trenutna ocena, ki jo ima algoritem za Q-vrednost para (s, a) (glej razdelek 4.2). Oznaka $\vec{\boldsymbol{p}}(s)[a]$ pomeni vrednost, ki v vektorju strategije $\vec{\boldsymbol{p}}(s)$ pripada akciji a. $N(s)$ in $N(s, a)$ sta števili obiskov vozlišča s in povezave (s, a). Konstanta c je parameter raziskovanja, ki deluje podobno, kot smo opisali v razdelku 4.3.

Na koncu vrnemo izboljšano strategijo $\vec{\boldsymbol{\pi}}(s)$, ki jo dobimo tako, da za vsako akcijo v vektor vnesemo delež obiskov akcije iz začetnega stanja tekom simulacij algoritma MCTS, torej:

$$\vec{\boldsymbol{\pi}}(s) = \left[\frac{N(s,a)}{N(s)}\right]_{a \in \mathcal{A}}.$$

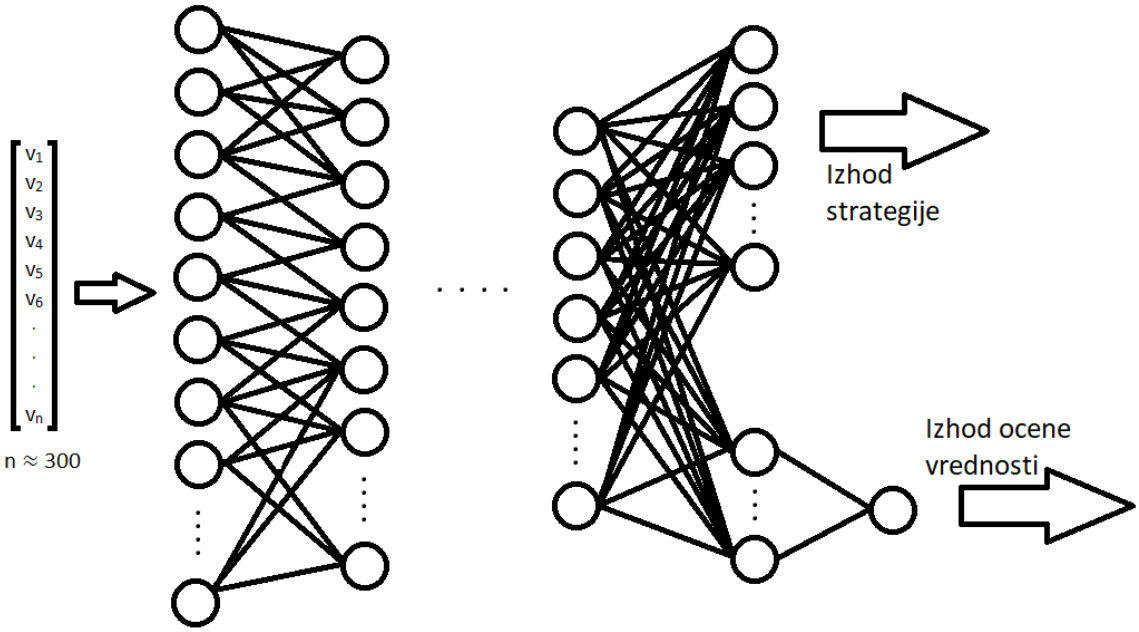
Nekoliko bolj podrobno je postopek opisan v psevdokodi 1.

4.4.3 Samo-igralno spodbujevalno učenje

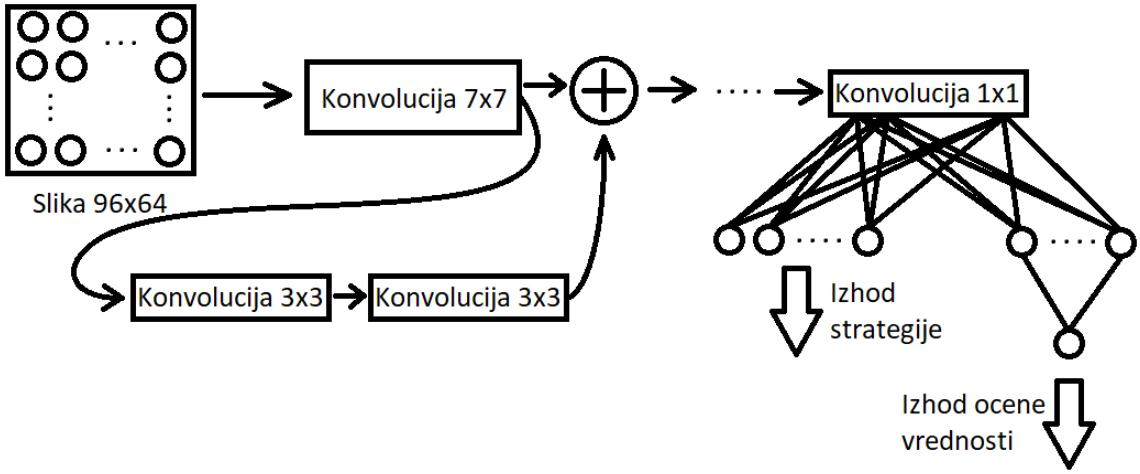
Za učenje nevronske mreže (izboljšanje njenih napovedi) potrebujemo izboljšane vektorje strategije in točne evaluacije stanj. V razdelku 4.4.2 smo opisali, kako dobimo boljše vektorje strategije, točne evaluacije pa dobimo v obliki nagrade že iz igre same, ko se ta zaključi. V AlphaZero je ta nagrada 1, če je igralec na koncu zmagal partijo, -1 pa, če je izgubil (v šahu je možen še remi, ki ima vrednost 0).

Mi pa smo nagrado definirali drugače, saj smo jo prilagodili našemu problemu. V našem algoritmu je nagrada sestavljena iz nagrade runde in nagrade igre:

$$r = \rho \cdot r_r + (1 - \rho) \cdot r_i. \tag{4.3}$$



(a) Podatkovna mreža. Za večjo preglednost so nekatere povezave med vhodno in prvo skrito plastjo opuščene.



(b) Slikovna mreža. Po konvolucijski plasti 7×7 vidimo razgrnjeno plast preostankov, ki se potem ponovi še nekajkrat.

Slika 4.4: Uporabljeni nevronski mreži. V obeh so zaradi preglednosti izpuščene normalizacijske, aktivacijske in odpadne plasti.

Psevdokoda 1 Naša verzija MCTS.

```
    ▷ This is the main function of policy improvement.
 1: function GetBetterPolicy(tree_root, nn, simulator, game_rules)
 2:     while ¬ game_rules.action_choice_time_expired() do
 3:         Search(tree_root, nn, simulator, game_rules)
 4:     π⃗(s) = [N(s,a)/N(s)]_{a∈A}
 5:     return π⃗(s)

 6: function Search(s, nn, simulator, game_rules)
 7:     if game_rules.game_ended(s) then
 8:         return −game_rules.reward(s)        ▷ Always return opposite of evaluation (alternating rewards for alternating players).
 9:     if ¬ s.visited then
10:         s.visited = True
11:         p⃗(s), v(s) = nn.predict(s)
12:         return −v(s)
13:     with game_rules.get_valid_actions(s) do
14:         a = Find best action according to eq. (4.2).
15:     next = simulator.next(s, a)
16:     v(s) = Search(next, nn, simulator, game_rules)
17:     Q(s,a) = (N(s,a)·Q(s,a)+v(s)) / (N(s,a)+1)        ▷ Update Q-value estimate with new v(s).
18:     N(s,a)++
19:     return −v(s)
```

Parameter ρ je tu konstanta, ki pove kolikšen delež prinese vsaka izmed posameznih nagrad. Mi smo uporabili $\rho = 0,8$.

Nagrada igre r_i se dodeli na koncu celotne igre kot v AlphaZero: 1 za zmago, -1 za poraz in 0 za neodločen izid. Nagrada runde pa se dodeli na koncu runde in je za igralca P_1 definirana kot

$$r_r = \frac{\mathrm{HP}_1 - \mathrm{HP}_2}{\max\left(\mathrm{sHP}_1, \mathrm{sHP}_2\right)},$$

za igralca P_2 pa zrcalno. sHP_i označuje HP igralca P_i ob začetku runde, $\mathrm{HP}_i \in [0, \mathrm{sHP}_i]$ pa HP, ki ga je imel na koncu runde. Opazimo, da sta r_r in r_i na intervalu $[-1, 1]$, torej je na tem intervalu tudi njuno uteženo povprečje r.

Taka nagrada precej bolje zajame dejanski vpliv akcije, saj nam rezultat runde pove o tem vplivu precej več kot rezultat celotne igre. Vseeno pa moramo upoštevati tudi rezultat celotne igre, saj se nekatere posledice akcij (npr. pridobitev energije) prenesejo med rundami.

Za celoten postopek učenja uporabljamo samo-igralno spodbujevalno učenje. Agenta ustvarimo z nevronsko mrežo s poljubno izbranimi utežmi (mi smo jih inicializirali naključno). Nato agent igra oba lika (torej igra sam proti sebi) in tako odigra eno igro, po kateri dobi učne primere oblike $(s_t, \vec{\boldsymbol{\pi}}_t, r_t)$. Pri tem je $\vec{\boldsymbol{\pi}}_t$ strategija, ki jo iz MCTS dobimo v stanju s_t, $r_t \in [-1, 1]$ pa je nagrada z vidika igralca, ki je bil ob času t na potezi (akcije izvajamo izmenično, čeprav so v ogrodju možne tudi sočasne izbire akcij). Nagrada se podeli šele na koncu igre.

S temi učnimi primeri na koncu igre potem mrežo naučimo tako, da minimizira sledečo funkcijo napake:

$$\sum_t \left((v(s_t) - r_t)^2 - \vec{\boldsymbol{\pi}}_t \cdot \ln \vec{\boldsymbol{p}}(s_t)\right).$$

Tako poskušamo doseči, da se nevronska mreža nauči čim bolje oceniti trenutno stanje in čim bolje izbrati najboljšo akcijo.

Te korake agent potem ponavlja. V našem algoritmu smo odigrali 500 iger z vsako mrežo, posamezna igra pa je bila sestavljena iz 3 rund po 30 sekund (vse skupaj torej približno 1 dan učenja). Celoten algoritem samo-igranja je opisan v psevdokodi 2 (da je psevdokoda enostavnejša, smo igre skrajšali na 1 rundo).

4.4.4 Reševanje težav in optimizacija

Za implementacijo algoritma smo poleg ogrodja FightingICE [26] (in pripadajoče ovojnice Py4J [16]) uporabili Pythonovi knjižnici Keras [11] in TensorFlow [1].

Ovojnica Py4J je že pri sami implementaciji povzročila nekaj težav, saj ob uporabi večnitnih programov (kot je ogrodje FightingICE) povsem odpove njeno obravnavanje izjem (angl. *exception handling*). Tako pri kakršnikoli napaki v našem (pythonskem) delu kode dobimo nazaj precej neuporabno poročilo o napaki:

> Exception in thread "main"py4j.Py4JException: An exception was raised by the Python Proxy. Return Message: x

Psevdokoda 2 Samo-igranje in učenje.

```
    ▷ This is the main function of our algorithm.
 1: function SelfPlay(game_rules, nn_type ∈ {'Data','Image'})
 2:     nn = build_neural_network(nn_type)
 3:     nn.randomize_weights()
 4:     for _ in game_rules.number_of_games() do
 5:         training_examples = PlayGame(nn, game_rules)
 6:         nn.train(training_examples)
 7:     return nn


 8: function PlayGame(nn, game_rules)
 9:     training_examples = []
10:     s = game_rules.starting_state()
11:     simulator = game_rules.get_simulator()
12:     while ¬ game_rules.game_ended(s) do
13:         π⃗(s) = GetBetterPolicy(s, nn, simulator, game_rules)
14:         training_examples.add([s, π⃗(s), _ ])
15:         a = game_rules.random_action(probability_distribution=π⃗(s))
16:         s = simulator.next(s, a)
17:     Assign rewards to training_examples as per equation (4.3).
18:     return training_examples
```

To težavo smo na koncu razrešili tako, da smo (precej v nasprotju s splošnimi pravili pisanja lepe kode) našo celotno kodo ovili v

```
try:
   our_code()
except Exception as e:
   print(e.args)
```

Večino dodelave ogrodja FightingICE smo opisali že v razdelku 3.1, vendar pa je bila potrebna še ena sprememba. Ogrodje namreč sicer omogoča pridobivanje slikovne informacije o trenutnem stanju, vendar pa enake funkcionalnosti nima implementirane za simulirana stanja iz simulatorja, ki je v ogrodje vgrajen. Simulator pa nujno potrebujemo v drevesnem preiskovanju Monte Carlo (glej psevdokodo 1 in razdelek 4.3). Ta težava torej povsem onemogoči delovanje agenta s slikovno nevronsko mrežo, saj v MCTS ne moremo dobiti slikovne informacije o simuliranih stanjih.

Zato smo implementirali približek slikovnega simulatorja, ki je iz slike trenutnega stanja, podatkov o trenutnem stanju in podatkov o simuliranem stanju (pridobljenih iz vgrajenega simulatorja) naredil približek slike v simuliranem stanju. To je izvedel tako, da je popravil pravokotnike, ki predstavljajo HP in energiji likov, nato pa še lika preslikal na pravi novi poziciji (in ju pobrisal s stare). S tem simulatorjem smo si potem pomagali pri agentu s slikovno nevronsko mrežo.

Delovanje našega približka simulatorja vidimo na sliki 4.5. Simulator deluje na slikovni informaciji, ki jo dobimo iz ogrodja, ta pa je že nekoliko predelana. Natančneje, pomanjšana je na velikost 96×64 in spremenjena v črno-belo. Sivo ozadje in uporaba barvnega negativa za enega od igralcev sta dosežena z nastavitvami ogrodja (glej razdelek 4.5). Kot vidimo na sliki, simulator ni popoln, saj ne zna pravilno popraviti položaja rok in nog. Za implementacijo tega bi potrebovali možnost simulacije dejanske slikovne izvedbe akcij, tega pa ogrodje ne omogoča.

Več težav pa se je pojavilo po končani implementaciji, saj je celoten algoritem deloval precej prepočasi:

1. Igra FightingICE teče s 60 sličicami na sekundo (angl. *frames per second*, FPS), vsaka od teh sličic pa je eno stanje, v katerem naj bi izvedli postopek izbire akcije. To pomeni, da imamo za eno izvedbo funkcije `GetBetterPolicy` iz psevdokode 1 na voljo približno 16 ms. Za primerjavo, AlphaZero je imel v igri proti Stockfishu za eno potezo na voljo celo minuto (torej 3600-krat več časa).

2. Klici javanske kode preko Py4J so časovno precej zamudni. Zato je grajenje podatkovnih vhodnih vektorjev[1] trajalo precej predolgo, saj je tam teh klicev približno 300 (toliko kot nam ogrodje daje na voljo podatkov o stanju). Za gradnjo enega samega vektorja je program tako porabil kar 30–40 ms, zgraditi pa bi jih morali v postopku ene same izbire več – toliko, kolikor napovedi bi radi naredili. Ta težava torej precej oteži delovanje agenta s podatkovno nevronsko mrežo. Pri slikovni nevronski mreži te težave ni, saj je za pridobitev trenutne slike potreben le en javanski klic, v MCTS pa – kot opisano zgoraj – uporabljamo svojo, simulirano sliko.

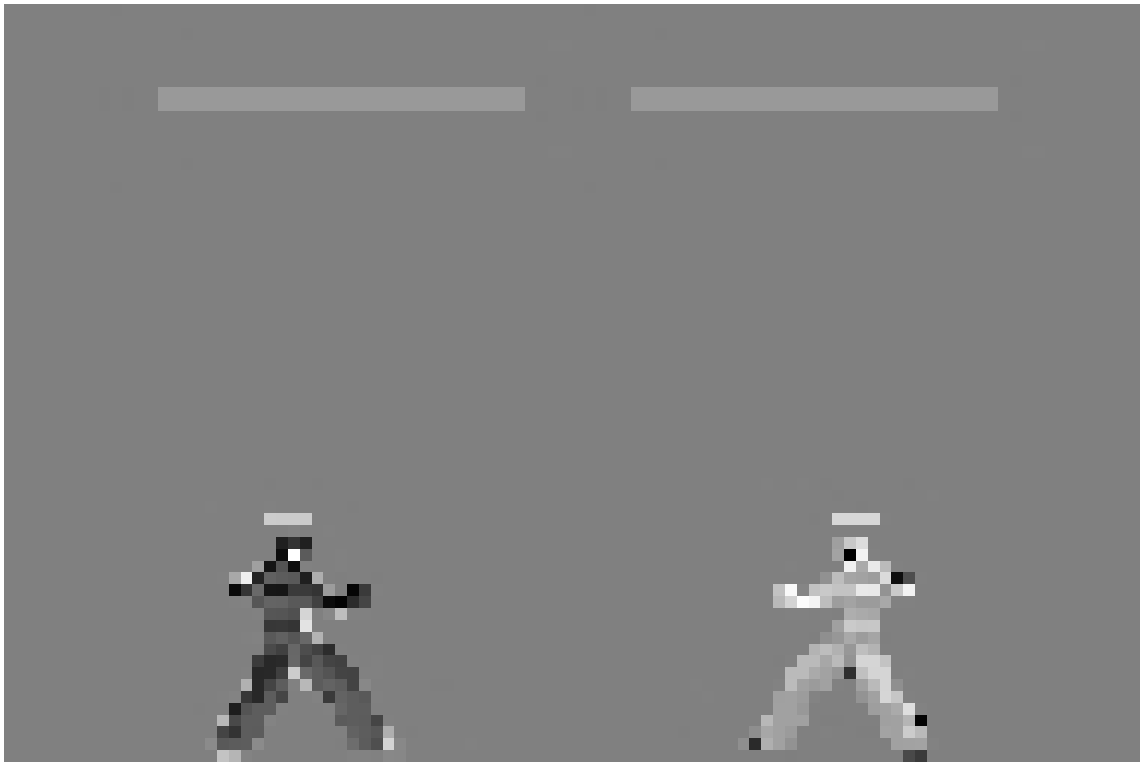
3. Posamezne napovedi nevronske mreže so prav tako trajale nekoliko predolgo: 4–5 ms pri podatkovni mreži in 5–6 ms pri slikovni. Ta čas bi si želeli zmanjšati, saj bi teh napovedi radi naredili več kot le 3–4 na odločitev.

Teh težav smo se lotili na naslednje načine:

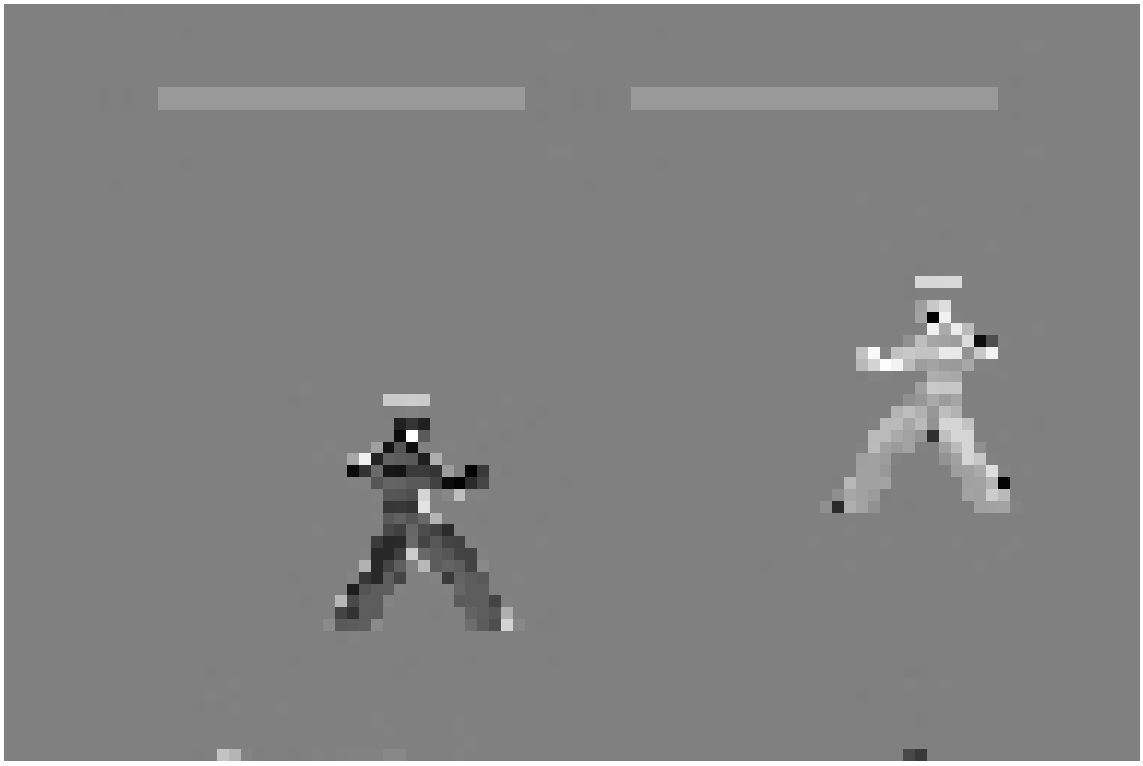
1. Kljub temu da igra res teče s 60 FPS, izbire akcije ni potrebno izvesti prav v vsaki sličici. Akcije namreč večinoma ne zahtevajo vnosa na sličico natančno, poleg tega pa se zaporedne izbire pogosto ignorirajo, saj akcije večinoma ne moremo izvesti, če se izvaja že kakšna druga. Zato smo čas izbire raje podaljšali in se odločali le na vsakih 5 sličic. Poskusili smo podaljšati tudi na 10 sličic, vendar pa se je to izkazalo za pretirano tako pri našem agentu kot tudi, ko smo enako podaljšanje preizkusili na že obstoječih delujočih agentih iz ogrodja. Sprememba na izbiranje vsakih 5 sličic je čas izbire podaljšala na nekoliko bolj sprejemljivih 80 ms.

   Poskusili smo tudi celotno igro upočasniti (da bi tekla npr. z 10 FPS), vendar pa je ta nastavitev precej globoko zapečatena v ogrodje in je nismo mogli spremeniti, ne da bi s tem pokvarili delovanje ogrodja.

[1] Zgraditi vhodni vektor pomeni iz stanja izluščiti podatke in jih preoblikovati v obliko, ki jo potem lahko pošljemo v nevronsko mrežo.



(a) Začetna slika.



(b) Rezultat naše simulacije.

Slika 4.5: Delovanje našega približka simulatorja. Levi lik je izvedel hitri korak naprej (angl. *forward dash*) in posebni napad, ki ga je nekoliko dvignil. Desni lik pa je med tem skočil nazaj.

2. Za dolgo grajenje podatkovnega vhodnega vektorja ni bilo druge rešitve, kot da smo vektor precej zmanjšali in tako zmanjšali število javanskih klicev, ki se izvedejo ob vsaki gradnji. Tako smo na koncu pristali na 8 vhodnih atributih, in sicer: X in Y koordinati obeh likov, HP obeh likov in energiji obeh likov. S tem se je čas gradnje enega podatkovnega vektorja zmanjšal na 1–2 ms.

3. Keras smo med napovedovanjem poskusili poganjati na CPU namesto na GPU. Poganjanje na GPU je boljše, ko lahko izkoristimo visoko število paralelnih jeder, ki jih GPU ponuja, pri eni sami napovedi pa se precej dela izvaja sekvenčno, kar pomeni, da se visoka paralelnost ne izkorišča dovolj dobro. V postopku učenja smo potem preklopili nazaj na GPU.

   Nevronski mreži smo zmanjšali. Pri podatkovni mreži smo število skritih plasti prepolovili na dve, poleg tega pa se je ob zmanjšanju vhodnega vektorja (glej prejšnjo točko) precej zmanjšalo tudi število nevronov v mreži. V slikovni mreži smo prav tako poskusili z dvema plastema ostankov namesto štirih, na koncu pa tudi z le eno.

   Namesto Kerasovega `model.predict` smo uporabili funkcijo direktno iz Kerasove hrbtenice – TensorFlowa. S tem smo se želeli izogniti zamudnim odvečnim delom Kerasovega napovedovanja.

   S temi koraki smo čas napovedi uspeli skrajšati na 3–4 ms za obe mreži.

Po implementaciji vseh teh optimizacij smo algoritem uspeli pripeljati do tega, da je izvajal 20–30 simulacij MCTS v postopku ene odločitve. Za primerjavo, AlphaZero je v bitki proti Stockfishu izvajal približno 80.000 simulacij na sekundo, torej je na vsako potezo izvedel nekaj milijonov simulacij.

4.5 Metodologija testiranja

Za testiranje smo naša agenta pognali proti trem nasprotnikom, ki so na voljo na spletni strani ogrodja FightingICE [26]. V naraščajočem vrstnem redu glede na težavnost so to bili:

1. **KickAI** je najpreprostejši agent, ki stalno izvaja akcijo brcanja ne glede na situacijo.

2. **MctsAi** je agent, ki, prav tako kot naša dva, temelji na drevesnem preiskovanju Monte Carlo. Uporablja pa klasično obliko tega preiskovanja, torej ne izvaja napovedi z nevronsko mrežo, pač pa uporablja naključno igranje. Ta agent je uradni primerjalni agent tekmovanja in se tudi uporablja kot nasprotnik v njihovem hitrostnem tekmovanju (angl. *speedrunning competition*).

3. **GigaThunder** je zmagovalec tekmovanja FTGAIC iz leta 2017. Je precej bolj kompleksen in za vsakega od likov uporablja drugačna pravila obnašanja. Prav tako si pomaga z drevesnim preiskovanjem Monte Carlo, ima pa še mnogo drugih pravil oblike če-potem, po katerih se ravna. Ne uporablja pa nobenih bolj naprednih metod strojnega učenja.

Vsaka tekma je bila sestavljena iz 9 iger po 3 runde. Vsaka runda je pri tem trajala 60 sekund, oziroma dokler eden od likov ni izgubil vseh točk zdravja. Oba lika sta vsako rundo začela s 400 HP. Za izgubljeno igro agent ni prejel nobene točke, za zmago je prejel točko, za morebiten neodločen izid pa 0,5 točke. Ogrodje smo pognali z argumenti, ki se uporabljajo tudi na uradnem tekmovanju:

```
javaw Main --limithp 400 400 --grey-bg --inverted-player 1
```

Poglavje 5

Rezultati

V tabeli 5.1 so izidi, ki sta jih naša dva agenta dosegla pri igranju proti agentom iz ogrodja in s tekmovanja. Izid oblike x-y pomeni, da je naš agent dosegel x točk, nasprotnik pa y. Nasprotniki in metodologija testiranja so natančneje opisani v razdelku 4.5.

Nasprotnik / Mreža	KickAI	MctsAi	GigaThunder
Podatkovna	9-0	0-9	0-9
Slikovna	8,5-0,5	0-9	0-9

Tabela 5.1: Rezultati testnih iger naših dveh agentov proti tekmovalnim.

Rezultati so bili dokaj neuspešni – oba agenta sta zmage dosegla le proti najosnovnejšemu nasprotniku.

Za glavno težavo se je izkazal kratek časovni interval, ki sta ga agenta imela na voljo za posamezno izbiro akcije. AlphaZero je pri igranju šaha proti Stockfishu za vsako potezo dobil 1 minuto časa za razmislek, poleg tega pa so ga poganjali na računalniku s 4 tenzorskimi procesnimi enotami. Naša dva agenta sta za izbiro akcije imela približno 16 ms, po implementaciji daljšega intervala za izbiro (glej razdelek 4.4.4) pa 80 ms. Poleg tega se je precej časa izgubilo tudi pri uporabi ovojnice Py4J. Agenta sta na koncu tako (kljub vsem poskusom optimizacije, ki smo jih uporabili) izvajala približno 20–30 simulacij v algoritmu MCTS. Za primerjavo povejmo, da je AlphaZero na sekundo izvedel približno 80.000 simulacij. Pri šahu je preiskovalno drevo sicer bolj razvejano, poleg tega pa pogosto zahteva precej globlje preiskovanje kot pretepaška igra, vendar pa je taka razlika v številu preiskanih stanj vseeno prevelika za primerljivo delovanje.

Poleg tega se je z uporabljenimi postopki optimizacije izgubilo tudi nekaj natančnosti (kot opisano v razdelku 4.4.4), kar je prav tako negativno vplivalo na delovanje agentov.

Agenta sta vseeno uspela premagati vsaj najosnovnejšega nasprotnika. Prišlo je sicer do nekaj izenačenih rund (v primeru slikovnega agenta celo do ene izenačene celotne igre), saj sta se naša agenta občasno ujela v zanko izbire iste akcije. To se zgodi, če agent v nekem stanju zelo visoko oceni akcijo premika, ki se konča na istem

mestu (to je npr. skok naravnost navzgor). Ker agent potem spet konča na istem mestu, se stanje spremeni le s premikom nasprotnika. Ker pa se KickAI sam po sebi ne premika, pridemo spet v isto stanje in zato agent spet visoko oceni isto akcijo. V tem primeru se potem lika nikoli ne dotakneta in pride do izenačenega rezultata.

To težavo bi lahko reševali z ročno implementiranimi pravili ali pa bi akcije z nevtralnimi premiki izključili iz postopka izbire. Vendar pa bi se s tem oddaljili od namena agenta, ki naj bi se igro naučil igrati s čim manj dodatnimi informacijami, poleg tega pa se težava pojavi samo v bitkah proti agentom, ki ne izvajajo nikakršnih premikov (taki agenti pa so v splošnem redki).

Poglavje 6

Diskusija

Pristop, ki smo ga izbrali je sicer v osnovi deloval, vendar pa smo se srečali s premnogimi težavami, od katerih so bile nekatere nerešljive v začrtanem časovnem okvirju.

Za ključno težavo z vidika časovne zahtevnosti se je izkazala ovojnica Py4J, ki jo za svoj pythonski vmesnik uporablja ogrodje FightingICE. Tako bi v tem ogrodju verjetno dobili boljše rezultate, če bi uporabili programski jezik Java in temu primerno prilagodili izbiro knjižnice za globoko učenje. Vendar pa tudi že samo ogrodje ni popolno, saj med drugim (vsaj zaenkrat) ne omogoča simulacije na slikovnih podatkih. S tem je precej onemogočen pristop s konvolucijsko nevronsko mrežo (kot jo uporablja tudi AlphaZero), pri katerem smo se morali zadovoljiti z implementacijo približka simulatorja. Žal pa so preostala podobna ogrodja še bolj nedokončana, tako da zaenkrat, kolikor nam je znano, boljša izbira ne obstaja.

Potrebi po simulatorju bi se lahko izognili tudi tako, da bi v mrežo poleg trenutnega stanja vnesli še zaporedje izbranih akcij agenta in nasprotnika. Mrežo bi potem naučili ocenjevati vrednost kombinacije trenutnega stanja in izvedbe izbranih akcij. S tem simulatorja ne bi potrebovali, poleg tega pa bi vsakič v postopku izbire akcije vhodni vektor gradili le enkrat, v simulacijah MCTS pa bi mu dodali le zaporedja akcij. Precej pa bi se s tem seveda povečala kompleksnost učenja mreže.

Upočasnitev ogrodja bi nam omogočila vsaj prikaz tega, kako dobro bi naš agent deloval, če časovna zahtevnost ne bi bila usodna. Žal pa nam take upočasnitve ni uspelo doseči.

Slikovno in podatkovno mrežo bi, če bi rešili težave s časovno zahtevnostjo, lahko tudi združili. Na ta način bi hkrati uporabljali tako slikovno informacijo kot tudi podatke o stanju, ki nam jih omogoči ogrodje. Taka mreža bi bila večja in kompleksnejša (kar pomeni, da bi napovedovanje in učenje trajalo dlje), bi se pa verjetno povečala tudi natančnost napovedovanja.

Smiselna bi bila morda tudi implementacija katere izmed metod sočasnega drevesnega preiskovanja Monte Carlo [5, 31, 42, 59]. Za razliko od šaha, šogija in podobnih iger igralci potez v pretepaških igrah ne izbirajo izmenično, MCTS pa je v osnovi namenjen igram z izmeničnimi potezami. Kljub temu v praksi osnovni algoritem deluje dobro tudi v primeru sočasnih potez. V našem primeru zamuda pri izbiri akcije, do katere pride zaradi predpostavke o izmeničnosti ni bistvena. Zamuda namreč večinoma ne škoduje, saj pri večini akcij takojšen odziv ni potreben. Če pa bi zamuda pomagala (da najprej vidimo, kaj bo naredil nasprotnik), pa bi jo

tudi v sočasni verziji algoritma lahko dosegli preprosto tako, da ne naredimo nič.

Pristop preigravanja izkušenj (angl. *experience replay*) [2, 32, 34, 38, 61, 74] v podobnih okoljih pomaga pri preprečevanju prekomernega prileganja (angl. *overfitting*). Zaporedni učni primeri so si pri zvezno-časovnih igrah pogosto zelo podobni, kar lahko povzroči prekomerno prileganje mreže podatkom. Pri preigravanju izkušenj se temu izognemo tako, da mreži občasno namesto trenutnega učnega primera posredujemo učni primer iz preteklosti. Mi smo za preprečevanje prekomernega prileganja v mreži uporabili odpadne plasti, poleg tega pa pomaga tudi, da v mrežo pošiljamo sliko samo na vsakih 5 sličic. Vseeno bi morda preigravanje izkušenj lahko pripomoglo k boljšemu konvergiranju učenja.

Agenta s slikovno mrežo bi brez težav posplošili tudi na igranje drugih pretepaških iger, dokler le-te omogočajo implementacijo simulatorja. Ta je v algoritmu nujen, saj brez njega ne moremo izvajati MCTS.

Poglavje 7

Zaključek

V nalogi smo najprej pregledali pristope, ki so se do sedaj uporabljali za globoko učenje v igrah. Nato smo predstavili ogrodje FightingICE in dodatke, ki smo jih v ogrodje implementirali v namen lažje uporabe v raziskavah. Predstavili smo glavne tri metode, ki smo jih v našem algoritmu uporabili (nevronske mreže, spodbujevalno učenje in drevesno preiskovanje Monte Carlo) in opisali, kako se na koncu združijo v celoto. Opisali smo še težave, s katerimi smo se ob implementaciji algoritma srečali in rešitve, ki smo jih uporabili. Na koncu smo še prikazali in analizirali dobljene rezultate, nato pa se še nekoliko posvetili težavam in rezultatom ter možnim nadgradnjam in izboljšavam.

Implementirali smo samo-igralnega učljivega agenta za pretepaško igro FightingICE. Pri tem smo uporabili pristop, ki je temeljil na algoritmu AlphaZero, ki ga je za igranje iger na igralni deski razvila Googlova ekipa DeepMind. Naš agent je sicer deloval in se je učil, vendar pa se je kratka časovna doba odločanja (16 ms) izkazala za problematično, še posebej ob težavah, ki so nastale pri uporabi ovojnice Py4J. Tako je agent izvajal premalo simulacij MCTS, kar je onemogočilo tako dobro učenje kot tudi dobro igranje igre.

Implementacija torej lahko služi kot osnova za samo-igralnega učljivega agenta za pretepaške igre, vendar pa v trenutnem stanju v tem ogrodju ni primerna za tekmovanje. Nadgradnja ogrodja z uporabo boljše pythonske ovojnice ali pa prevedba agenta v programski jezik Java (in uporaba temu primernih knjižnic za globoko učenje) pa bi ga lahko pripravila do precej boljšega delovanja in morda celo do zmage proti najboljšim agentom tekmovanja.

Literatura

[1] M. Abadi in dr., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, `https://www.tensorflow.org/`, 2018, accessed: 2018-08-25.

[2] S. Adam, L. Busoniu in R. Babuska, *Experience Replay for Real-Time Reinforcement Learning Control*, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) **42**(2) (2012) 201–212, doi: 10.1109/TSMCC.2011.2106494.

[3] N. Appiah in S. Vare, *Playing FlappyBird with Deep Reinforcement Learning*, teh. por., Stanford University, 450 Serra Mall, Stanford, CA 94305, USA, 2016.

[4] K. Arulkumaran in dr., *Deep Reinforcement Learning: A Brief Survey*, IEEE Signal Processing Magazine **34**(6) (2017) 26–38, doi: 10.1109/MSP.2017.2743240.

[5] D. Beard, P. Hingston in M. Masek, *Using Monte Carlo Tree Search for replanning in a multistage simultaneous game*, v: 2012 IEEE Congress on Evolutionary Computation, 2012, str. 1–8, doi: 10.1109/CEC.2012.6256428.

[6] Y. Bengio, *Learning deep architectures for AI*, Foundations and Trends® in Machine Learning **2**(1) (2009) 1–127.

[7] Y. Bengio in dr., *A neural probabilistic language model*, Journal of machine learning research **3**(Feb) (2003) 1137–1155.

[8] C. B. Browne in dr., *Survey of Monte Carlo Tree Search Methods*, IEEE Transactions on Computational Intelligence and AI in Games **4**(1) (2012) 1–43, doi: 10.1109/TCIAIG.2012.2186810.

[9] B. Brügmann, *Monte Carlo Go*, teh. por., Citeseer, 1993.

[10] D. Chicco, P. Sadowski in P. Baldi, *Deep autoencoder neural networks for gene ontology annotation predictions*, v: Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, ACM, 2014, str. 533–540.

[11] F. Chollet in dr., *Keras*, `https://keras.io`, 2018, accessed: 2018-08-25.

[12] T. W. S. Chow in Y. Fang, *A recurrent neural-network-based real-time learning control strategy applying to nonlinear systems with unknown dynamics*, IEEE transactions on industrial electronics **45**(1) (1998) 151–161.

[13] D. CireşAn in dr., *Multi-column deep neural network for traffic sign classification*, Neural networks **32** (2012) 333–338.

[14] P. Covington, J. Adams in E. Sargin, *Deep Neural Networks for YouTube Recommendations*, v: Proceedings of the 10th ACM Conference on Recommender Systems, ACM, 2016, str. 191–198.

[15] B. C. Csáji, *Approximation with artificial neural networks*, Faculty of Sciences, Etvs Lornd University, Hungary **24** (2001) 48.

[16] B. Dagenais in dr., *Py4J - A Bridge between Python and Java*, `https://www.py4j.org/`, 2018, accessed: 2018-08-25.

[17] Department of Data Science & Knowledge Engineering, Maastricht University, IEEE, *Conference on Computational Intelligence and Games (CIG)*, `https://project.dke.maastrichtuniversity.nl/cig2018/`, 2018, accessed: 2018-08-25.

[18] J. Fu in I. Hsu, *Model-Based Reinforcement Learning for Playing Atari Games*, teh. por., Stanford University, 450 Serra Mall, Stanford, CA 94305, USA, 2016.

[19] E. Galván-López in dr., *Heuristic-Based Multi-Agent Monte Carlo Tree Search*, v: IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications, 2014, str. 177–182, doi: 10.1109/IISA.2014.6878747.

[20] F. A. Gers in J. Schmidhuber, *LSTM recurrent networks learn simple context-free and context-sensitive languages*, IEEE Transactions on Neural Networks **12**(6) (2001) 1333–1340.

[21] *GNU General Public License v3*, `http://www.gnu.org/licenses/gpl-3.0.html`, accessed: 2018-08-25.

[22] A. Graves in dr., *Biologically plausible speech recognition with LSTM neural nets*, v: International Workshop on Biologically Inspired Approaches to Advanced Information Technology, Springer, 2004, str. 127–136.

[23] X. Guo in dr., *Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning*, v: Advances in Neural Information Processing Systems 27 (ur. Z. Ghahramani in dr.), Curran Associates, Inc., 2014, str. 3338–3346.

[24] A. Hannun in dr., *Deep Speech: Scaling up end-to-end speech recognition*, arXiv preprint arXiv:1412.5567 (2014).

[25] K. He in dr., *Deep Residual Learning for Image Recognition*, CoRR **abs/1512.03385** (2015).

[26] Intelligent Computer Entertainment Lab. (ICE Lab.), Ritsumeikan University, *FTGAIC: FighTing Game AI Competition*, `http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-2h.html`, accessed: 2018-08-25.

[27] N. Justesen in S. Risi, *Learning macromanagement in starcraft from replays using deep learning*, v: 2017 IEEE Conference on Computational Intelligence and Games (CIG), 2017, str. 162–169, doi: 10.1109/CIG.2017.8080430.

[28] L. P. Kaelbling, M. L. Littman in A. W. Moore, *Reinforcement learning: A survey*, Journal of artificial intelligence research **4** (1996) 237–285.

[29] L. Kocsis in C. Szepesvári, *Bandit based Monte-Carlo Planning*, v: ECML-06. Number 4212 in LNCS, Springer, 2006, str. 282–293.

[30] Y. LeCun in dr., *Gradient-based learning applied to document recognition*, Proceedings of the IEEE **86**(11) (1998) 2278–2324, doi: 10.1109/5.726791.

[31] D. Lenz, T. Kessler in A. Knoll, *Tactical cooperative planning for autonomous highway driving using Monte-Carlo Tree Search*, v: 2016 IEEE Intelligent Vehicles Symposium (IV), 2016, str. 447–453, doi: 10.1109/IVS.2016.7535424.

[32] L.-J. Lin, *Reinforcement Learning for Robots Using Neural Networks*, doktorska disertacija, Carnegie Mellon University, Pittsburgh, PA, USA, 1992, uMI Order No. GAX93-22750.

[33] M. Lu in X. Li, *Deep reinforcement learning policy in Hex game system*, v: 2018 Chinese Control And Decision Conference (CCDC), 2018, str. 6623–6626, doi: 10.1109/CCDC.2018.8408296.

[34] B. Luo, Y. Yang in D. Liu, *Adaptive Q-Learning for Data-Based Optimal Output Regulation With Experience Replay*, IEEE Transactions on Cybernetics (2018) 1–12, doi: 10.1109/TCYB.2018.2821369.

[35] W. S. McCulloch in W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, The Bulletin of Mathematical Biophysics **5**(4) (1943) 115–133, doi: 10.1007/BF02478259.

[36] T. Mikolov in dr., *Recurrent neural network based language model*, v: Eleventh Annual Conference of the International Speech Communication Association, 2010.

[37] S. Miyashita in dr., *Developing game AI agent behaving like human by mixing reinforcement learning and supervised learning*, v: 2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2017, str. 489–494, doi: 10.1109/SNPD.2017.8022767.

[38] V. Mnih in dr., *Playing Atari with Deep Reinforcement Learning*, ArXiv e-prints (2013).

[39] V. Mnih in dr., *Human-level control through deep reinforcement learning*, Nature **518** (2015) 529 EP.

[40] J. A. M. Nijssen, *Monte-Carlo Tree Search for Multi-Player Games*, doktorska disertacija, Maastricht University, 2013.

[41] J. A. M. Nijssen in M. H. M. Winands, *Playout Search for Monte-Carlo Tree Search in Multi-player Games*, v: Advances in Computer Games (ur. H. J. van den Herik in A. Plaat), Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, str. 72–83.

[42] P. Perick in dr., *Comparison of different selection strategies in Monte-Carlo Tree Search for the game of Tron*, v: 2012 IEEE Conference on Computational Intelligence and Games (CIG), 2012, str. 242–249, doi: 10.1109/CIG.2012.6374162.

[43] E. Perot in dr., *End-to-End Driving in a Realistic Racing Game with Deep Reinforcement Learning*, v: 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2017, str. 474–475, doi: 10.1109/CVPRW.2017.64.

[44] K. Ranjan, A. Christensen in B. Ramos, *Recurrent Deep Q-Learning for PAC-MAN*, teh. por., Stanford University, 450 Serra Mall, Stanford, CA 94305, USA, 2016.

[45] S. J. Russel in P. Norvig, *Artificial Intelligence: A Modern Approach*, Upper Saddle River: Prentice-Hall, 3 izd., 2010.

[46] H. Sak, A. Senior in F. Beaufays, *Long short-term memory recurrent neural network architectures for large scale acoustic modeling*, v: Fifteenth Annual Conference of the International Speech Communication Association, 2014.

[47] J. Schaeffer, *One jump ahead: challenging human supremacy in checkers*, Springer Science & Business Media, 2013.

[48] J. Schmidhuber, *Deep learning in neural networks: An overview*, Neural Networks **61** (2015) 85–117, doi: https://doi.org/10.1016/j.neunet.2014.09.003.

[49] P. B. S. Serafim in dr., *Towards Playing a 3D First-Person Shooter Game Using a Classification Deep Neural Network Architecture*, v: 2017 19th Symposium on Virtual and Augmented Reality (SVR), 2017, str. 120–126, doi: 10.1109/SVR.2017.24.

[50] A. R. d. Silva in L. F. Wanderley Goes, *HearthBot: An Autonomous Agent based on Fuzzy ART Adaptive Neural Networks for the Digital Collectible Card Game Hearthstone*, IEEE Transactions on Computational Intelligence and AI in Games **PP**(99) (2017) 1–1, doi: 10.1109/TCIAIG.2017.2743347.

[51] A. Silver, *The future is here – AlphaZero learns chess*, `https://en.chessbase.com/post/the-future-is-here-alphazero-learns-chess`, accessed: 2018-08-09.

[52] D. Silver in dr., *Mastering the game of Go with deep neural networks and tree search*, Nature **529** (2016) 484–489.

[53] D. Silver in dr., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, ArXiv e-prints (2017).

[54] D. Silver in dr., *Mastering the game of Go without human knowledge*, Nature **550**(7676) (2017) 354–359, doi: 10.1038/nature24270.

[55] N. Srivastava in dr., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research **15** (2014) 1929–1958.

[56] M. Stanescu in dr., *Evaluating real-time strategy game states using convolutional neural networks*, v: 2016 IEEE Conference on Computational Intelligence and Games (CIG), 2016, str. 1–7, doi: 10.1109/CIG.2016.7860439.

[57] M. Stevens in S. Pradhan, *Playing Tetris with Deep Reinforcement Learning*, teh. por., Stanford University, 450 Serra Mall, Stanford, CA 94305, USA, 2016.

[58] R. S. Sutton in A. G. Barto, *Introduction to reinforcement learning*, **135**, MIT press Cambridge, 1998.

[59] M. J. W. Tak, M. Lanctot in M. H. M. Winands, *Monte Carlo Tree Search variants for simultaneous move games*, v: 2014 IEEE Conference on Computational Intelligence and Games, 2014, str. 1–8, doi: 10.1109/CIG.2014.6932889.

[60] G. Tesauro, *Temporal Difference Learning and TD-Gammon*, Communications of the ACM **38**(3) (1995) 58–68, doi: 10.1145/203330.203343.

[61] N. Thakoor in B. Bhanu, *Selective experience replay in reinforcement learning for reidentification*, v: 2016 IEEE International Conference on Image Processing (ICIP), 2016, str. 4250–4254, doi: 10.1109/ICIP.2016.7533161.

[62] M. Vitek, *FighterZero: A Deep Learning AI for the ICE Fighting Game AI Competition Framework*, `https://github.com/MatejVitek/FighterZero`, accessed: 2018-08-25.

[63] Y. Wang, H. He in C. Sun, *Learning to Navigate through Complex Dynamic Environment with Modular Deep Reinforcement Learning*, IEEE Transactions on Games (2018) 1, doi: 10.1109/TG.2018.2849942.

[64] C. J. C. H. Watkins, *Learning from delayed rewards*, doktorska disertacija, King's College, Cambridge, 1989.

[65] C. J. C. H. Watkins in P. Dayan, *Q-learning*, Machine learning **8**(3-4) (1992) 279–292.

[66] P. Werbos in P. John, *Beyond regression : new tools for prediction and analysis in the behavioral sciences* (1974).

[67] Y. Wu in dr., *Google's neural machine translation system: Bridging the gap between human and machine translation*, arXiv preprint arXiv:1609.08144 (2016).

[68] R. Xiongli, R. Xiongxing in H. Yinglai, *UCT-RAVE algorithm applied to multiplayer games with imperfect information*, v: 2011 6th IEEE Joint International Information Technology and Artificial Intelligence Conference, **1**, 2011, str. 312–315, doi: 10.1109/ITAIC.2011.6030213.

[69] S. Yoon in K.-J. Kim, *Deep Q networks for visual fighting game AI*, v: 2017 IEEE Conference on Computational Intelligence and Games (CIG), 2017, str. 306–308, doi: 10.1109/CIG.2017.8080451.

[70] D. Yu in L. Deng, *Automatic Speech Recognition: A Deep Learning Approach*, Springer-Verlag London, 2015.

[71] R. Zhang, *Convolutional and Recurrent Neural Network for Gomoku* , teh. por., Stanford University, 450 Serra Mall, Stanford, CA 94305, USA, 2016.

[72] W. Zhang, *Shift-invariant pattern recognition neural network and its optical architecture*, v: Proceedings of annual conference of the Japan Society of Applied Physics, 1988.

[73] W. Zhang in dr., *Parallel distributed processing model with local space-invariant interconnections and its optical architecture*, Appl. Opt. **29**(32) (1990) 4790–4797, doi: 10.1364/AO.29.004790.

[74] D. Zhao in dr., *Deep reinforcement learning with experience replay based on SARSA*, v: 2016 IEEE Symposium Series on Computational Intelligence (SSCI), 2016, str. 1–6, doi: 10.1109/SSCI.2016.7849837.