

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Jernej Habjan

**Učenje realno-časovne strateške igre z
uporabo globokega spodbujevalnega
učenja**

DIPLOMSKO DELO

UNIVERZITETNI ŠTDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Matej Guid
SOMENTOR: prof. dr. Branko Šter

Ljubljana, 2018

COPYRIGHT. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil L^AT_EX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogu:

Tematika naloge:

Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode uporabiti, morda bo zapisal tudi ključno literaturo.

Zahvaljujem se mentorju doc. dr. Mateju Guidu in somentorju prof. dr. Branku Šteru, prijateljem in družini, ki so mi pomagali pri pisanju diplomske naloge.

Kazalo

Povzetek

Abstract

1	Uvod	1
2	Realno-časovne strateške igre	5
2.1	Strategija	7
2.2	Taktika	7
2.3	Abstrakcija prostora	8
3	Predstavitev algoritma Alpha Zero	9
3.1	Zgodovina	9
3.2	Potek učenja	10
4	Definiranje pravil igre	13
4.1	Stanje igre	14
4.2	Akcije	16
4.3	Kodiranja	18
4.4	Konec igre	20
5	Učenje modela	25
5.1	Zgradba nevronske mreže	26
5.2	Izbira parametrov	27

6 Vizualizacije	29
6.1 Pygame	29
6.2 Unreal Engine 4	31
7 Rezultati	35
7.1 Učenje z ustavljeno funkcijo s časovno omejitvijo 100	35
7.2 Povečevanje časovne omejitve na 200	39
7.3 Sprememba konfiguracij zlata	39
7.4 Zmanjševanje velikosti šahovnice	41
7.5 Vizualizacija rezultatov v Unreal Engine 4	42
8 Diskusija	45
9 Zaključek	47
Literatura	50

Seznam uporabljenih kratic

kratica	angleško	slovensko
MCTS	Monte Carlo tree search	Monte-Carlo drevesno preiskovanje
UE4	game engine Unreal Engine 4	celostni pogon Unreal Engine 4
RTS	real-time strategy	realno-časovna strateška
One hot	one hot	kodiranje z eno enico v zapisu vsakega stanja
JSON	JSON JavaScript Object Notation	notacija za označevanje JavaScript objektov

Povzetek

Naslov: Učenje realno-časovne strateške igre z uporabo globokega spodbujevalnega učenja

Avtor: Jernej Habjan

Z obstoječim Alpha Zero algoritmom smo implementirali učenje in priporočanje akcij v realno-časovni strateški igri. Pregledali smo krajšo zgodovino globokega spodbujevalnega učenja na igrah in povzeli zakaj je pristop samostojnega učenja najprimernejši. Za strateško igro smo definirali figure in njihove akcije in zakodirali kompleksno stanje igre s kodirnikom. Prav tako smo definirali ustavitevne pogoje pri igri, ki nima končnega števila potez na podlagi zmanjšanja življenjskih točk figur. Rezultate smo prikazali s Python modulom Pygame in v celostnem pogonu Unreal Engine 4. V obeh vizualizacijah lahko igramo proti naučenem modelu, ali pa opazujemo, kako se dva računalniška nasprotnika bojujeta med sabo. Na koncu smo še pregledali rezultate in povzeli učinek učenja algoritma.

Ključne besede: Alpha Zero, realno-časovna strateška igra, Unreal Engine.

Abstract

Title: Teaching of real-time strategy game using deep reinforcement learning

Author: Jernej Habjan

With the existing Alpha Zero algorithm, we implemented learning and recommending actions in a real-time strategy game. We examined the shorter history of deep stimulating learning in games and summarized why the self-learning approach is most appropriate. For a strategic game, we defined the figures and their actions and encoded the complex state of the game with the encoder. We also defined the stopping conditions of the game, which has no final number of moves based on damage to the figures. The results were displayed with the Python Pygame module and the Unreal Engine 4 integrated drive. In both visualizations we can play against the learned model, or we can observe how two computer opponents are fighting each other. In the end, we have also reviewed the results and summarized the learning effect of the algorithm.

Keywords: Alpha Zero, real-time strategy game, Unreal Engine.

Poglavlje 1

Uvod

Razvijanje inteligenčnega agenta v realno-časovnih oziroma RTS igrah je problem, s katerim se mora soočiti večina razvijalcev teh iger. Agentove akcije so pa pogosto predvidljive, saj se človeški igralec nauči njihovih načinov delovanja in jih tako lažje premaga. Če pustimo agentu, da sam opravlja akcije nekontrolirano, bo te akcije izvajal naključno, ki so pa večino časa slabše kot vnaprej definirana taktika. Lahko pa agentu podamo hevristiko, po kateri se mora ravnati in ta bo poskušal izvesti čim boljšo akcijo, vendar bo za njen izračun porabil predolgo časa, saj bo moral pregledati cel preiskovalni prostor, ki pa pri realno-časovnih strateških igrah zna biti prevelik. Na primer 10 figur v igri, kjer ima vsaka 5 možnih potez, se razveji na možen faktor $5^{10} \approx 10$ milijonov možnih akcij. Za igro StarCraft je ocenjenih možnih vsaj 10^{1685} možnih akcij, kjer je za šah 10^{47} in 10^{171} za igro Go [3].

Preiskovanje prostora z grobo silo torej odpade. Ostanejo nam potem hevristični algoritmi, kot na primer Alpha-Beta rezanje ali Monte-Carlo drevesno preiskovanje oziroma MCTS. Ampak algoritom Alpha-Beta rezanje deluje dobro samo pod pogoji, da obstaja zanesljiva evaluacijska funkcija in da ima igra majhen vejitveni prostor, kar je pa lastnost veliko klasičnih namiznih iger kot Go in video iger. Zato se je v takšnih primerih bolje odločiti za algoritmom MCTS [1]. Ta pa ima pomankljivost, da si stanj igre ne zapomni skozi iteracij več iger, kjer bi lahko to vrednost stanja uporabil za bolj

natančen izračun naslednjih stanj.

Za pomnjenje stanj so primerne globoke nevronske mreže, ki pa z učenjem ugotovijo zakonitosti v učni množici in skozi mnogo iteracij izboljšajo svojo napoved določenega izhoda ob določenem vhodu. To je pa točno to, kar potrebuje MCTS kot začetno stanje, iz katerega lažje izračuna najboljšo akcijo.

Da pa nevronska mreža dobi dovolj vhodnih podatkov za učenje, pa moramo realizirati algoritmom, ki bo igral proti drugem računalniškem nasprotniku, in s tem izgradil dovolj veliko učno množico z rezultati zmag oziroma porazov teh iger. Ob tem izhodu nevronska mreža posodobi svojo napoved akcij za določeno stanje igre.

To je glavna ideja o implementaciji algoritma, ki jo pa vsebuje algoritmom Alpha Zero, ki smo ga uporabili v tej diplomski nalogi. Algoritmom se nauči igranja raznih namiznih iger, kot tudi naše realno-časovne strateške igre z igranjem mnogo iger sam proti sebi, kjer boljša različica algoritma napreduje v naslednjo iteracijo. Ko je model nevronske mreže naučen, ga lahko uporabimo, da nam priporoči akcijo v določenem stanju. S tem lahko implementiramo računalniškega igralca, ki pridobiva akcije od naučenega modela in jih izvršuje, kot tudi priporočilni sistem za akcije, ki jih prikazujemo človeškemu igralcu. Algoritmom nam priporoči akcijo in ne tipa strategije, katerega naj izberemo, kar bi potrebovalo še bolj abstrakten pogled na igro.

O strateških igrah, njihovih abstrakcijah in zakaj so tako zanimive za raziskovanje umetne inteligenčne bomo več spoznali v poglavju 2. Za tem si bomo podrobnejše pregledali sestavo Alpha Zero algoritma in zakaj je primeren za našo realno-časovno strateško igro v poglavju 3. Ko bomo imeli sestavljen algoritmom učenja, bomo zanj sestavili RTS igro v poglavju 4 in izpostavili, kaj so glavne težave pri teh igrah v zvezi z njihovim učenjem. Sestavljen algoritmom bomo naučili na igri v poglavju 5, kjer bomo pregledali razne parametre pri učenju in naučen model potem preizkusili z vizualizacijo v Python modulu Pygame in celostnem pogonu Unreal Engine 4 v poglavju 6. Rezultate učenja bomo potem še ocenili in ugotovili, katera vrsta učnih parametrov nam je podala najboljši rezultat v poglavju 7. V poglavju 8

pa bomo rezultate pregledali s širše perspektive, omenili bomo tudi možne nadaljnje pristope in izboljšave, ki bi pomagale pri učenju algoritma. Pregledali bomo tudi koristnost aplikacije naučenega modela v pogon kot je Unreal Engine in kaj so njegove omejitve. Na koncu bomo še na hitro pregledali dosežke in prispevke diplomske naloge v poglavju 9.

Poglavlje 2

Realno-časovne strateške igre

Realno-časovne strateške oziroma RTS igre so žanr strateških iger, kjer igralec nadzoruje množico figur, in poskuša premagati nasprotnika z izgradnjo ekonomije, izboljšavo raznih tehnolog in urjenjem primernih vojaških figur, ki dodajo dodano vrednost končnem cilju poraza nasprotnega igralca in s tem zmagi igre. Primer RTS igre je na primer Age of Empires II ali igra StarCraft.

Izzivi realno-časovnih iger so naslednji:

- Upravljanje z viri,
- izbira akcij ob nevednosti,
- prostorsko in časovno razmišljanje,
- sodelovanje med več agenti,
- modeliranje nasprotnika in učenje,
- nesporno načrtovanje v realnem času.

Zdajšnji izzivi:

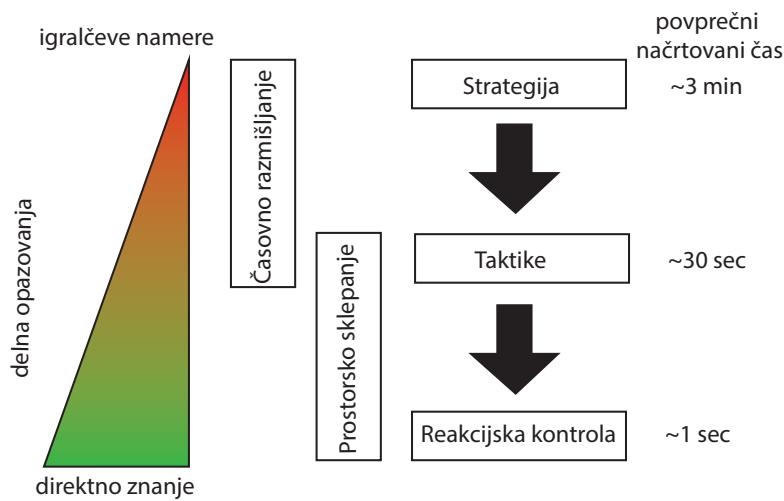
- planiranje: V realno-časovni igri je vidno kot več nivojev abstrahiranega stanja igre. Višji kot je nivo, bolj dolgoročni so cilji, kot na

primer gradnja ekonomije, na nižjem nivoju je pa premik posamezne figure ipd.,

- Učenje: Predhodno učenje, ki uporablja posnetke že odigranih iger, učenje v igri, ki uporablja po večini spodbujevalno učenje in modeliranje nasprotnika, učenje med igrami,
- negotovost: Negotovost nastane zaradi nevidnosti nasprotnika in njegovih potez v vsakem trenutku. Prav tako pa ne vemo akcij, ki jih bo nasprotnik izvedel, zato zgradimo drevo, ki nam pove kaj je najverjetnejše da bo nasprotnik naredil,
- prostorsko in časovno razumevanje: Prostorsko razumevanje je usmerjeno k postavljanju stavb in pozicijo vojske za obrambo in napad, Časovno razumevanje je pa usmerjeno k ugotavljanju, kdaj je primerna izdelava hiš za ekonomijo in kdaj pa za napad,
- izkoriščanje znanja domen: Izkoriščanje znanje botov. StarCraft je kompleksen, in to ostaja še odprt problem,
- razdelitev nalog 2.1: Strategija, ki je najvišja abstrakcija (3 min planiranje), Taktika, ki je implementacija trenutne strategije (pozicija vojske, hiš - 30 sekundno planiranje), Reakcijska kontrola, ki je implementacija taktike, ki je osredotočena na posamezno figuro, Analiza terena, ki se osredotoča na strnjena območja in na višinsko prednost, Pridobivanje znanja, s katerim pridobivamo informacije o taktiki nasprotnika [4].

Pogosto razdelimo odločanje na dva dela:

- micro, kjer kontroliramo figure posamezno,
- macro, kjer se osredotočimo na ekonomijo in izdelavo figur.



Slika 2.1: Razdelitev nalog glede na čas reakcije in abstrakcije nalog.

2.1 Strategija

V strateških igrah je velikokrat uporabljen pristop direktnega kodiranja strategije, ki uporabljajo avtome končnih stanj, kjer lahko razbijemo delovanje na več stanj kot so napadanje, nabiranje surovin, popravilo itd. in hitro menjavanje med njimi. Direktno kodiranje prinese dobre pričakovane rezultate, vendar se lahko igralec nauči strategije in ga tako agenta hitro porazi. Planirani pristopi ponujajo večjo prilagodljivost kot direktno kodirani.

2.2 Taktika

Taktika spada pod neposrednejši nadzor figur kakor strategija in je bolj osredotočena na kontrolo določenih točk na mapi, zmagi posameznih bitk in iskanje ožin, kjer je nasprotnik šibkejši. Taktika temelji na analizi terena, ki ga lahko razbijemo na kompozicijo ožin.

2.3 Abstrakcija prostora

Razbiranje strategije in taktike je za algoritme umetne inteligence težji, saj potrebuje višji nivo abstrakcije prostora, figur in akcij, kot za izbiro posameznih nizkonivojskih akcij.

Prav tako problem nastane zaradi negotovosti, kjer ne vidimo nasprotnikovih figur in potez v vsakem trenutku. Predikcija nasprotnikovih potez je tako veliko težja, tako da vsi algoritmi s tako negotovostjo ne delujejo. Mi smo se za diplomsko nalogo odločili, da imata oba računalniška agenta popoln vpogled na stanje igre in nasprotnikove akcije.

Poglavlje 3

Predstavitev algoritma AlphaZero

3.1 Zgodovina

Igranje iger je popularno področje znotraj vede o umetni inteligenci. Eden izmed prvih programov je bil programski pogon Checkers (Samuel 2000), ki se je naučil igranja z metodami samo-igranja in strojnega učenja in ne z metodami, ki temeljijo na pravilih. Leta 2002 je Deep Blue premagal človeškega profesionalnega igralca šaha z nadčloveško sposobnostjo igranja. Pri teh igrah je faktor vejanja akcij še relativno majhen in je lažje oceniti končno pozicijo iz danega stanja. Rečeno je bilo, da igre kot npr. Go, ki imajo toliko večji faktor vejanja 10^{171} v primerjavi s šahom, ki pa ima 10^{47} , ne bo možno ugotoviti vrednost končnega stanja še nekaj desetletij.

Ampak algoritem AlphaGo [5] je naredil preboj, s tem da uporablja metodo globokega spodbujevalnega učenja in algoritom Monte-Carlo drevesno preiskovanje. Oktobra 2016 je premagal profesionalnega Go igralca na podlagi učenja na domenskem znanju iger, ki so bile odigrane od ekspertov. Te sistemi so temeljili na predznanju ekspertov za učenje in evaluacijo modela.

Leto za tem, je bil razvit algoritam AlphaGo Zero [6], ki opisuje pristop k učenju brez domenskega znanja ekspertov, ampak uporablja metodo samo-

igranja. Novi model je prav tako premagal AlphaGo algoritom, kar predstavlja odlične rezultate z vidika, da AlphaGo Zero ne potrebuje človeško usmerjanje pri učenju.

Računalniki se lahko tako naučijo reševanje problema brez človeških ekspertov, ki delajo napake in nimajo takojšnjega vpogleda na celotno učno množico, kot to imajo računalniki.

Za tem je bil razvit algoritmom Alpha Zero, ki vzame ideje AlphaGo Zero kot temelj, ampak je model generaliziran za poljubne igre, kot na primer šah, Shogi, Go, kjer algoritmom potrebuje samo pravila igre, ta pa se uči z globokimi nevronskimi mrežami in tabula rasa algoritmom za spodbujevalno učenje. Zaradi te generalizacije algoritma, lahko algoritmom apliciramo na našo RTS igro, kjer moramo definirati pravila igre. AlphaZero je drugačen od AlphaGo Zero tako, da AlphaZero vrača rezultate, ki so lahko drugačni od zguba, poraz, kot tudi neodločeno. Prav tako se razlika pojavi v tem, da so se igre pri algoritmu AlphaGo Zero zgenerirale iz vsej prejšnjih iteracij, in se je potem moč modela izračunala proti najboljšim igralcem, medtem ko AlphaZero samo hrani eno nevronsko mrežo, ki se stalno posodablja, namesto da čaka iteracijo da se konča.

3.2 Potek učenja

Alpha Zero se uči verjetnosti in ocenitve končnega stanja izključno z igranjem proti samemu sebi. Te potem uporabi pri preiskovanju z glavno namensko metodo Monte-Carlo drevesnim preiskovanjem, da razišče drevo stanj za akcijo. Drevo preišče prostor in vrne verjetnost zmage pri izbiri določene akcije iz trenutnega stanja imenovano Π in oceno končnega stanja iz trenutnega stanja v , ki zavzema vrednosti -1 ali 1 (Poraz, zmaga). Alpha Zero izvede več serij igranja iger proti svojim nasprotnikom, ki predstavlja zdajšnji najboljši model igranja. Rezultat igranja igre je lahko -1 za poraz, +1 za zmago in 0 za neodločeno. Po vsaki seriji učenja, se izvede proces igranja Arena, kjer oba naučena modela igrata drug proti drugemu nekaj iger, in se na to določi

zmagovalen model, ki sedaj postane najboljši model, če je razlika v številu zmag večja za nek faktor. V našem primeru je bil ta faktor 60%. Parametri nevronske mreže so za tem popravljeni, da minimizirajo napako med napoved stanja nevronske mreže in dejanskim rezultatom igre in da maksimizirajo podobnost predikcijo potez nevronske mreže z dejanskimi vrednostnimi akcijami, ki jih je vrnil MCTS. Oziroma parametri se nastavijo z gradientnim spustom na funkcijo izgube, ki sešteje napako srednjega korena (mean-squared error) in prečne entropije (cross entropy). Nevronska mreža sprejme učne množice stanja iger in vrne ravni vektor napovedi akcij v trenutnem stanju in napoved zmage.

Uporaba algoritma MCTS je v tem algoritmu drugačna kakor v splošni uporabi. Število iteracij je namenjeno biti veliko manjši, kakor v njegovi klasični uporabi, kjer je število iteracij več sto tisoč. MCTS pripomore k izboljšavi napovedi stanja, ki ga vrne nevronska mreža z raziskovanjem prostora. Algoritem ne uporablja simulacij za pridobitev končnega stanja igre, napoved stanja, ki ga vrne nevronska mreža samo izboljša.

Vozlišče, ki še ni bilo obiskano, se vzpostavi s napovedjo nevronske mreže in za tem vzvratno propagira napoved stanja. Če je vozlišče končno stanje igre, vzvratno propagira končno stanje. MCTS v tem primeru prejme par sto iteracij (v našem primeru 30 - 50) in ne več tisoč, kot jih izvajajo drugi algoritmi. V sklopu diplomske naloge govorimo o MCTS iskanjih in ne simulacijah.

Izrek 3.1 formula po kateri računa verjetnost zmage pri določeni akciji v algoritmu MCTS

$$R(s, a) = Q(s, a) + c \text{puct} P(s, a) \sqrt{\frac{\sum b * N(s, b)}{N(s, a)}} \quad (3.1)$$

Glavno učenje algoritma poteka z igranjem iger, ki pa se za razliko od MCTS-ja odigrajo do konca in se dodajo v seznam učnih primerov. Konec vsake iteracije igranja epizod iger se nevronska mreža uči na podlagi teh učnih primerov. Za tem preveri moč novo naučenega modela z igranjem

proti starejši različici modela in se shrani novi model samo če je boljši od starejšega za določen odstotek.

Poglavlje 4

Definiranje pravil igre

Igro smo definirali po Surag Nairjevi predlogi za Alpha Zero, ki je na voljo na repozitoriju Github (<https://github.com/suragnair/alpha-zero-general>). Igra je dodana kot modul, ki vsebuje definicijo igre in njena pravila, igralce, vizualizacijo in izgradnjo modela.

Igra je definirana v kvadratni mreži 8×8 , kjer polje lahko vsebuje največ eno figuro. Ostale igre, ki so napisane za to različico Alpha Zero izvedbe, kot na primer štiri v vrsto, gobang, othello, tri v vrsto, vsebujejo črno-bele figure. Zakodirane so lahko z eno številko: -1 za igralca -1, +1 za igralca +1 ali 0, če je polje prazno. Pri teh igrah je dimenzija kodiranja 2-dimenzionalna, kjer dimenzijske predstavljajo višino in širino igralne plošče. Pri RTS igrah pa moramo vedeti poleg igralca, komur ta figura pripada, tudi stanje te figure, na primer trenutno zdravje in tip figure. Zato je prostor kodiranja 3-dimenzionalen, kjer je tretja dimenzija zakodirano stanje figure. Če bi dovolili, da na posamezno polje spada več figur, se dimenzija ponovno poveča za 1.

Ob prvem poizkusu definiranja igre smo se zapledli v ne-numeričen prikaz igre, ki pa je zelo spremenil Surag-Nairjevo implementacijo algoritma Alpha Zero. Implementacija igre se je učila zelo počasi, saj je bilo preverjanje akcij počasnejše od sedajšne implementacije. Algoritem je na voljo na naslednjem repozitoriju:

<https://github.com/JernejHabjan/TD2020-Object-AlphaZero>.

4.1 Stanje igre

V tem razdelku smo opisali zapis posamezne figure, njihove akcije in kaj naredijo in tip kodiranja stanja igre, ki ga potem sprejme nevronska mreža.

Igro smo definirali tako, da je čim bolj skladna s samim algoritmom Alpha Zero, kot tudi da je njena aplikacija dovolj primerljiva z obstoječimi strateškimi igrami kot npr. StarCraft. S tem v mislih, smo definirali nekaj preprostih pravil, ki jih ta igra upošteva:

- figure se ne požirajo: Vojaške figure ne napadejo drugih figur tako, da če je akcija napad možna, se postavijo na polje sovražne figure in s tem prepišejo sovražnikovo figuro in s tem jo uniči. Prav tako imajo vse figure določeno zdravje, ki ga vojaške figure v večih korakih zmanjšajo z napadom,
- ena figura na polje: S tem ni možno blokiranje figur, s tem da se figura postavi na polje zlata in blokira sovražno figuro da jih nabere,
- igralec zgubi, če zgubi vse figure - več o ustavitevih pogojih spodaj v razdelku 4.4,
- nabiranje zlatnikov je enkratna operacija, ki poteka podobno kot pri igri StarCraft, kjer mora figura pristopiti do polja zlata, zlatnike pobrati in jih za tem vrniti v glavno hišo. Nabiranje zlatnikov v tem primeru ne poteka tako, da se figura pomakne do polja zlata in s tem prične avtomatično pridobivati zlatnike, brez da bi jih vračal na odlagališče.

Sprva moramo definirati figure, ki bodo imele določeno vlogo v igri. Nabor figur je majhen, saj nočemo, da preiskovalni prostor postane prehitro prevelik.

- polje zlata - Vir surovin, ki predstavljajo denar v igri, s katerim lahko igralec gradi nove stavbe in uri nove figure. Vir zlata je neomejen in ne mora biti uničen,
- delavec - Figura namenjena gradnji hiš in nabiranju zlata,
- vojašnica - Stavba namenjena urjenju vojaških figur,
- vojak - Figura namenjena napadanju sovražnikovih figur,
- glavna hiša - Stavba namenjena urjenju delavcev in vračanju surovin zlata.

Na posameznem polju je lahko največ ena figura, tako da igralec ne more blokirati surovin zlata nasprotnemu igralcu, če to surovino ne obkoli v celoti.

Realizirali smo atributi figur. Pomembno je, da so te atributi numerični, da lahko podamo stanje igre kot N -dimenzionalen vektor, ki ga nevronska mreža lahko sprejme in se iz teh numeričnih podatkov uči. Prav tako je pomembno, da ima vsako polje na šahovnici enako število atributov, tudi če je to polje prazno. Vsako prazno polje ima vanj vpisan atribut čas igranja, ki je splošen za celo igro, vsa ostala polja pa imajo vrednost 0.

- ime igralca: Določa igralca, h kateremu ta figura pripada. Igralec lahko nadzoruje samo svoje figure, izvaja akcije na svojih figurah in napada sovražnikove figure,
- tip figure: Atribut predstavlja numerično predstavitev tipa figure kot na primer polje zlata, delavec ipd. Stanje igre potrebuje zapise tipov figur na poljih, da program ve, katere akcije tem figuram pripadajo,
- trenutno zdravje: Koliko zdravja ima trenutna figura. Zdravje se lahko povečuje do nekega maksimuma z akcijo zdravi in znižuje z napadom figure,
- nosi zlatnike: Poseben atribut za delavce, ki predstavlja vrednost 1, če figura nosi zlatnike in 0, če ga ne nosi. To se upošteva pri nabiranju

in vračanju zlata, kjer se ti dve akcije ne zgodita v roku ene poteze, ampak se mora stanje prenašati skozi več potez,

- denar: Trenutna količina zbranega denarja za posameznega igralca. To polje se ob spremembi količine denarja spremeni v vseh figurah tega igralca,
- čas igranja: To polje predstavlja koliko potez se je v trenutni igri že izvedlo. Atribut je prisoten v vseh poljih in se spremeni v vseh poljih šahovnice, ko se izvede nova akcija.

Poseben primer je figura polje zlata, ki ne pripada nobenemu igralcu v večini RTS igrah. V tem primeru pa smo podali vsakemu igralcu svoje polje zlata, da je igra simetrična in nevronska mreža ne interpretira prazno polje igralca kot prazno polje. Prav tako se figure polje zlata ne spreminja atribut zdravja, saj jo ne moremo poškodovati. Zlata je neomejeno in ko igralec odloži zlatnike v glavno hišo, se vsem figuram tega igralca nastavijo zlatniki na novo dobljeno vrednost. Prav tako se pri izgradnji nove stavbe ali urjenju figure število zlatnikov zmanjša za vse figure tega igralca.

4.2 Akcije

Prav tako moramo definirati akcije, ki jih te figure lahko izvajajo. Vsaka figura ne more izvajati vseh akcij, kot na primer stavbe se ne morejo premikati, same figure kot delavec in vojak pa ne morejo uriti novih figur 4.1.

- premiki (4 smeri) - Figuri vojak in delavec se lahko premakneta na sosednje polje na šahovnici, če je to mesto prazno,
- naberi zlatnike - Delavec lahko nabere zlatnike če je v neposredni bližini figure polje zlata in jih za trenuten čas drži pri sebi,
- vrni zlatnike - Delavec vrne zlatnike, ki jih drži pri sebi v glavno hišo, na kar se igralcu prišteje denar,

- napadi (4 smeri)- Vojak lahko napade sovražno figuro, če je ta v neposredni bližini in jo rani za določen faktor,
- izuri delavca (4 smeri)- Glavna hiša lahko izuri novo figuro delavec, če ima dovolj denarja, na kar se igralcu odšteje denar,
- izuri vojaka (4 smeri)- Vojašnica lahko izuri novo figuro vojak, če ima dovolj denarja, na kar se igralcu odšteje denar,
- izgradi vojašnico (4 smeri)- Delavec lahko izgradi vojašnico na prazno mesto zraven njega, na kar se igralcu odšteje denar,
- izgradi glavno hišo (4 smeri)- Delavec lahko izgradi glavno hišo na prazno mesto zraven njega, na kar se igralcu odšteje denar,
- zdravi (4 smeri) - Figura lahko zdravi sosednjo prijateljsko figuro, če ta nima polnega življenja, na kar se igralcu odšteje denar.

Akcijo nedejavnosti smo izključili iz algoritma, ker ne prinaša nobene dodatne vrednosti igralcu ter zagotavlja daljše iteracije igre.

Sprva smo definirali nekatere izmed zgornjih akcij, na način izbire prvega mesta med sosednjimi polji, ki ustreza akciji. Naslednje akcije so se izvajale po zaporedju:

- naberi zlatnike,
- vrni zlatnike,
- napadi,
- delavec,
- vojak,
- vojašnica,
- glavna hiša,

- zdravi.

```

coords = [(x - 1, y + 1),
           (x, y + 1),
           (x + 1, y + 1),
           (x - 1, y),
           (x + 1, y),
           (x - 1, y - 1),
           (x, y - 1),
           (x + 1, y - 1)]
for n_x, n_y in coords:
    # check action condition or execute action

```

Ko je doseženo prvo prazno polje v tem zaporedju, se tam izgradi nova hiša ali izuri nova figura. Ko je prva sovražna figura izbrana v tem zaporedju, je napadena. Rezultat tega je gradnja hiš in urjenja figur v spodnji levi kot šahovnice, saj so izbrana prva polja v zaporedju, kot naprimer x-1, y+1, in širjenje proti zgornjim desnim kotom, ko so vsa ostala polja zasedena, oziroma tam ni sovražnih figur.

Algoritem smo popravili tako, da smo spremenili te akcije (razen naberi zlatnike in vrni zlatnike), da so posamezne akcije za vsako izmed štirih sosednjih polj. Za vpeljavo posameznih akcij smo se odločili, ker ne moramo izbrati polja po zgornjem zaporedju tako, da bi bilo za oba igralca enako. Sedaj je igra scela uravnovešena glede na igralca.

Popravek za to bi bilo definiranje zgornjih navedenih akcij za vsako izmed teh polj, kar bi se prevedlo v veliko večji prostor akcij.

4.3 Kodiranja

Definirali smo še začetno stanje vsake igre, kjer sta igralca postavljena v sredino mreže z njihovima glavnima hišama, zraven njiju pa ima vsak igralec svoje polje zlata. Vsakemu igralcu se doda na začetku določena količina

Ime figure	Akcije	Zdravje	Strošek izdelave
Polje zlata	/	10	0
Delavec	smeri premikanja, vojašnica, glavna hiša, naberij in vrni zlatnike, zdravi	10	1
Vojašnica	vojak, zdravi	10	4
Vojak	smeri premikanja, napad, zdravi	20	2
Glavna hiša	delavec, zdravi	30	7

Tabela 4.1: Tu so še opisane figure z njihovimi akcijami in nastavljenimi atributi.

denarja za izgradnjo začetnih delavcev. V našem primeru je bilo to 1, tako da je lahko izgradil samo enega delavca.

Sedaj pa smo potrebovali zakodirati to stanje igre, v numerični prikaz, ki ga bo nevronska mreža lahko interpretirala. To stanje lahko zakodiramo z desetiškim kodiranjem, vendar obstaja možnost, da nevronska mreža sloni proti boljšim obravnavanjem pozitivnih števil za igralca +1, kot za igralca -1. Ravno iz tega razloga obstaja kodiranje One hot, ki spremeni desetiška števila v binarni vektor.

Akciji naberij in vrni zlatnike, ostaneta še vedno po 1 akcijo, ker za delavca ni razlika, iz katerega sosednjega polja zlata vzame zlatnike, kot ni razlike pri vračanju njih.

4.3.1 Desetiško

Pri desetiškim kodiranjem, predstavimo vsak atribut figure z eno desetiško številko. Ker imamo figure s 6 atributi, lahko stanje zakodirane igre predstavimo z dimenzijami širina x višina x 6.

Igralec predstavlja številko -1 za igralca -1, 1 za igralca 1 in 0 za prazno

polje.

4.3.2 One-hot

- ime igralca: 2 bita zaradi treh različnih možnosti: 00 predstavlja prazno polje, 01 predstavlja igralca 1 in 10 igralca -1,
- tip figure: 3 biti, saj imamo 5 različnih figur,
- trenutno zdravje figure: 5 bitov saj hočemo predstaviti večjo številko, zaradi odštevanja zdravja z ranjujočo funkcijo 4.1,
- nosi zlatnike: 1 bit, kjer vrednost lahko zajema vrednost nosi - 1 ali ne nosi - 0,
- denar: 5 bitov, kjer pustimo da igralec gradi ekonomijo in shranjuje denar, da ga potem lahko na hitro zapravi na figurah, ko ga ima dovolj za njihovo izgradnjo,
- čas igranja: $2^{11} = 2048$, kar pusti igralcu dovolj časa da odkriva nove poteze, ampak ga dovolj hitro omeji, da se konča igra in začne nova.

Dimenzija zakodiranega prostora je tako $8 \times 8 \times 22$, kar je 3.6-krat števil, ki kodira posamezno stanje igre. To zna otežiti učenje nevronske mreže, ker ima s tem kodiranjem več števil, pri katerih mora ugotoviti primernost posameznega števila.

4.4 Konec igre

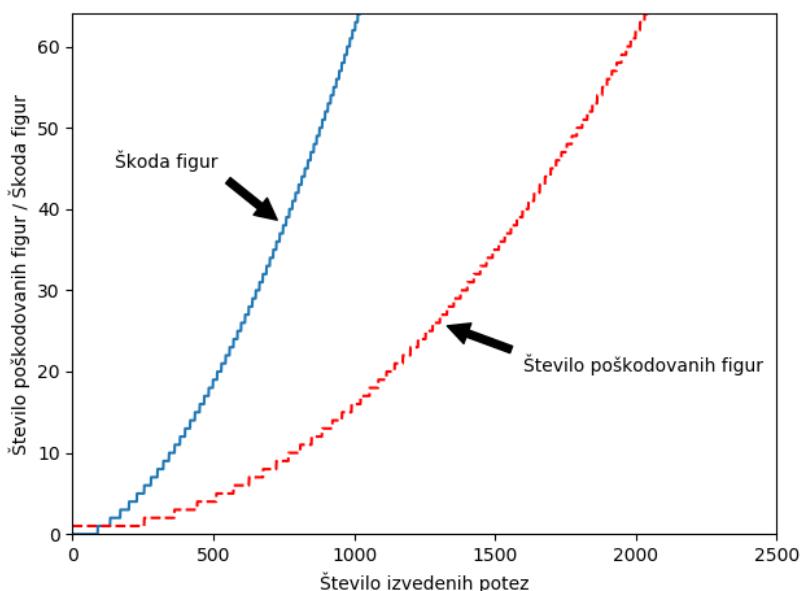
Konec igre se izvede pod določenimi pogoji:

- igralec nima za izvesti več nobene možne akcije,
- vse figure igralca so uničene,
- ko se izteče čas.

Definirati smo morali sintetičen konec igre, saj se lahko igra zacikla tako, da se na primer delavci premikajo v ciklu in se tako igra nikoli ne konča. Hoteli smo prioritizirati aktivne igralce, ki nabirajo zlatnike in imajo več enot kakor nasprotni igralec.

4.4.1 Reševanje problema neskončnega števila potez

Čakanje, da se čas igre izteče, je problematično, saj učenje modela poteka zelo počasi, še posebej ko MCTS raziskuje prostor. Za to smo razvili funkcijo, ki prisili model k izvajjanju akcij v zgodnjem času igre, drugače se figuram preveč začnejo zmanjševati življenske točne in so zato eliminirane s šahovnice.



Slika 4.1: Na grafu sta narisani dve funkciji, ki določata zmanjšanje življenskih točk se obravnava v določeni figuri v danem času igre(modra) in koliko igralcev je bilo poškodovanih v trenutnem časovnem okviru(rdeča).

Vidimo, da je krivulja zmanjšanja življenskih točk veliko bolj stroga in se v igri začenja že zelo zgodaj. To je zato, ker želimo hitro odpraviti nedejavnih igralcev in tiste, ki zbirajo zlatnike in pridobivajo nove figure. Vidimo tudi

y osi od 0 do 64, kar je največje število igralcev za enega igralca, tako da bo pri približno 2000 korakih vsaka figura dobil smrtno poškodbo, zato časovni potek nikoli ni dosežen.

Figure lahko tudi uporabijo akcijo zdravljenja, s čimer povečajo trenutno zdravje določene figure do največ njenega maksimuma.

Z zgoraj definirano funkcijo je bil problem določiti pravšnjo stopnjo količine zmanjšanja življenjskih točk figur in izločevanje neaktivnih igralcev dovolj zgodaj v igri, in sicer problem z balansiranjem količine in stroškom zdravljenja. Če so bili stroški dovolj nizki, so igralci stalno samo zdravili figure in končali igro pri približno tisoč potezah, kar je pregloboko za normalno igro. Za to je bilo potrebno povisati strošek zdravljenja, kar je pa privedlo do hitrejšega nenadnega umiranja figur, saj igralci niso imeli dovolj kovancev za zdravljenje, na kar je bilo potrebno povisati količino vrnjenih kovancev iz figure zlato.

4.4.2 Ustavitevni pogoj

Ustavitevni pogoj deluje tako, da se na številu določenih potez igra preprosto prekini in oceni zmagovalca po eni izmed spodaj navedenih formul. Igra se prekine po na primer 100 ali 200 potezah, če se do takrat igralca med sabo še nista spopadla. V spodnjih treh enačbah sta označena igralec 1 z oznako p1 in igralec 2 z oznako p2.

Izrek 4.1 *Prvi: igralec 1 zmaga, če ima več denarja kot igralec 2*

$$p1.zlatniki > p2.zlatniki \quad (4.1)$$

Prvi izrek je dober ustavitevni pogoj, za testiranje igralcev pri nabiranju zlatnikov, kjer zmaga preprosto tisti, ki jih nabere več.

Izrek 4.2 *Drugi: igralec 1 zmaga, ko je seštevek zdravja vseh figur igralca 1 je večji od seštevka zdravja vseh figur igralca 2*

$$\sum p1.figure.zdravje > \sum p2.figure.zdravje \quad (4.2)$$

Če je pogoj za zmago večje število življenja svojih figur kakor nasprotnikovih hkrati pomeni, da lahko igralec nabira več zlatnikov in z njimi gradi nove hiše in uri nove enote, kar zagotavlja za igralca večjo skupno vsoto življenja figur in hkrati zagotavlja težo k urjenju vojaških enot z namenom, da sovražnim enotam zmanjša število življenjskih točk.

Izrek 4.3 *Tretji: igralec 1 zmaga, ko je seštevek zdravja vseh figur igralca 1 plus njegov denar je večji od seštevka zdravja vseh figur igralca 2 plus njegov denar*

$$\sum p1.figure.zdravje + p1.zlatniki > \sum p2.figure.zdravje + p2.zlatniki \quad (4.3)$$

K drugemu izreku smo še pripeli trenutno število shranjenih zlatnikov igralca, kar dodatno doprinaša motivacijo igralca k nabiranju novih zlatnikov.

Poglavlje 5

Učenje modela

Učenje te igre je zapleteno zaradi pogojev konca igre. Algoritem pričakuje, da se igra konča z uporabo MCTS, vendar se pa lahko igra zacikla če igralec večkrat ponovil isto potezo, na kar Python javi napako zaradi prevelike globine rekurzije. To se lahko reši z uporabo časovnih omejitev, kjer se MCTS ustavi ko se izteče čas, vendar lahko povzroči netočno MCTS drevo, ker vozišča niso pravilno ovrednotena med povratnim propagiranjem. Potrebno je najti ustrezno končno stanje ali spremeniti vir, da izključite časovne omejitve, ker ne vrnejo najboljših rezultatov.

Prav tako se nam je porodila učna ideja o postopnim učenjem modela. Najprej bi začeli učiti model na preprostem končnem pogoju kot na primer številu izdelanih delavcev. Ko bi model uspešno ustvarjal delavce, bi pogoj spremenili o na primer nabiranju zlata, tako da bi model že vedel o gradnji delavcev, kar bi nadgradil še z nabiranjem zlata. Težava se pojavi zaradi kodiranja stanja, ki ni istih dimenzij kot prejšnjo stanje, kjer smo imeli drugi ustavitevni pogoj z drugačnim številom akcij pri figurah. Možna učna ideja Ideja je, da se s postopnim učnim modelom spremeni stanje konca igre. Najprej začnite učiti model na preprost način končne igre, kot so izdelava delavcev in ko model uspešno ustvarja delavce, dodajte še en pogoj poleg tega že izvedenega modela. Možna težava se lahko pojavi zaradi velikosti modela

5.1 Zgradba nevronske mreže

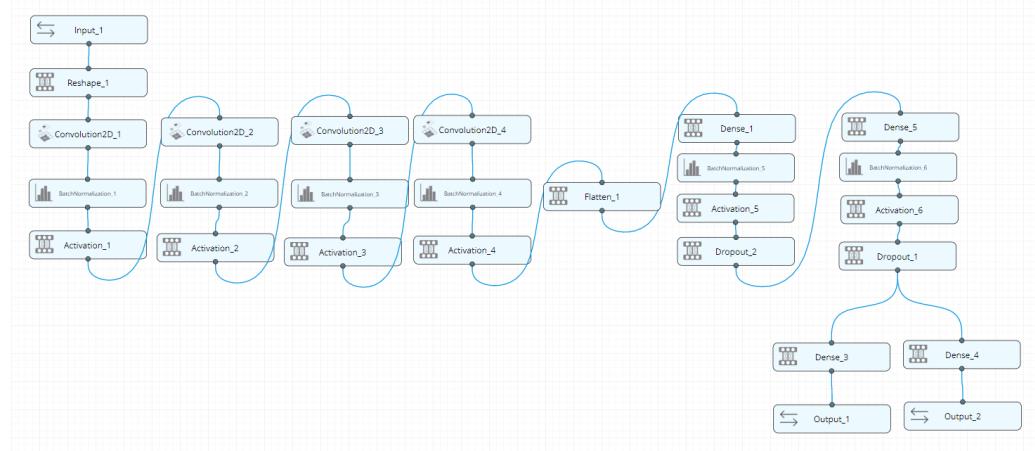
V sklopu te diplomske naloge se nismo podajali v spremnjanje zgradbe nevronske mreže, temveč smo vzeli že izgrajeno nevronsko mrežo, primerno za učenje igre Othello. To ni najprimernejši pristop, kar je mogoče tudi poslabšal zmožnost in hitrost učenja modela, o čemer smo več prediskutirali v poglavju 8.

Uporabili smo modul Keras znotraj TensorFlow knjižnice, za implementacijo modela nevronske mreže. Programska koda za izgradnjo modela je lažje berljiva v modulu Keras kakor v TensorFlow, zato smo se zanj tudi odločili. Ker pa je Keras implementiran znotraj TensorFlow knjižnice od verzije 1.9 izdani leta 2017, ni bilo večjih težav z inštalacijo te knjižnice na odjemalčevem računalniku z vtičnikom tensorflow-ue4.

Model za vhod vzame učne množice stanja iger dimenzij širina x višina x število kodirnikov, kar je v našem primeru $8 \times 8 \times 6$. Potem gre ta učna množica skozi 4 konvolucijske nivoje, kjer je velikost filtra 3. Prva dva konvolucijska nivoja imata oblogo ničel okrog matrike, tako da se velikost konvolucijskega nivoja ne zmanjša, druga dva pa tega oblage nimata, kar zniža velikost nivoja iz 8 na 4. tako da je izhod zadnjega konvolucijskega nivoja dimenzije velikost serije x (širina - 4) x (višina - 4) x število kanalov Vsaka izmed konvolucijskih nivojev se normalizira z "normalizacijo serije", ki normalizira aktivacijo prejšnjega nivoja ob vsaki seriji, ti pa se spustijo skozi relu aktivacijsko funkcijo. Za tem se izhod zadnjega nivoja normalizirane konvolucije izravna v 1-dimenzionalni vektor in se poda dvema polno povezanima nivojem, ki sta ponovno normalizirana z "normalizacijo serije". Tedva nivoja sta potem ponovno spuščena skozi aktivacijsko funkcijo relu podana v Dropout funkcijo, ki prepreči prekomerno prileganje.

Za tem je izgrajen polno povezan nivo P_i , ki ima toliko število izhodov, koliko je možno število akcij v igri za vsako celico, ki ima aktivacijsko funkcijo softmax, prav tako je pa izgrajen polno povezan nivo V , ki ima en izhod, ki predstavlja zmago ali poraz z tanh aktivacijsko funkcijo. Za izhod P_i se nastavi funkcija izgube kategorična prečna entropija, za izhod V pa srednja

napaka korena (mean-squared error).



Slika 5.1: Model nevronske mreže, uporabljen za učenje igre. Model je bil izdelan z uporabo aplikacije na voljo na <https://deeprcognition.ai/>

5.2 Izbira parametrov

Cpuct je parameter drevesnega raziskovanja. V našem primeru je bil nastavljen na vrednost 1.

Število iteracij predstavlja število učenj algoritma na odigranih ighrah in število izbire boljšega modela. Število epizod nam zagotovi pridobitev dovolj velikega nabora odigranih iger, nad katerimi se potem algoritem uči. Število MCTS iskanj predstavlja število raziskanih vozlišč v iteraciji igre. MCTS iskanja se ne izvršijo do konca igre, ampak do neraziskanega vozlišča oziroma če je raziskano vozlišče konec igre. Število primerjanj modelov: Kolikokrat se bosta trenutni model k se uči in njegova prejšna različica pomerila med sabo, da se ohrani boljši. Število ohranjenih učnih primerov nam zagotavlja, da ohranjamo novejše učne primere in starejše zavrzemo. V našem primeru je bil nastavljen na manjšo vrednost (8), saj same igre zasedejo veliko pomnilnika.

Poglavlje 6

Vizualizacije

6.1 Pygame

S Python knjižnico Pygame smo izdelali vizualizacijo, ki je primerna za pregled igre med samim razvijanjem. Šahovnica je označena s črtami, med katerimi so s krogi izrisane figure, kjer njihove barve predstavljajo svoj tip figure in obroba krogca igralca -1 ali +1. V krogcih je tudi napisano zdravje za to figuro in zastavica, ali delavec prenaša zlato. Zgoraj je izpisano, koliko denarja ima posamezen igralec in koliko potez sta igralca že odigrala. Prav tako so izpisane vse možne akcije, ki jih igralec lahko izvrši z določeno figuro.

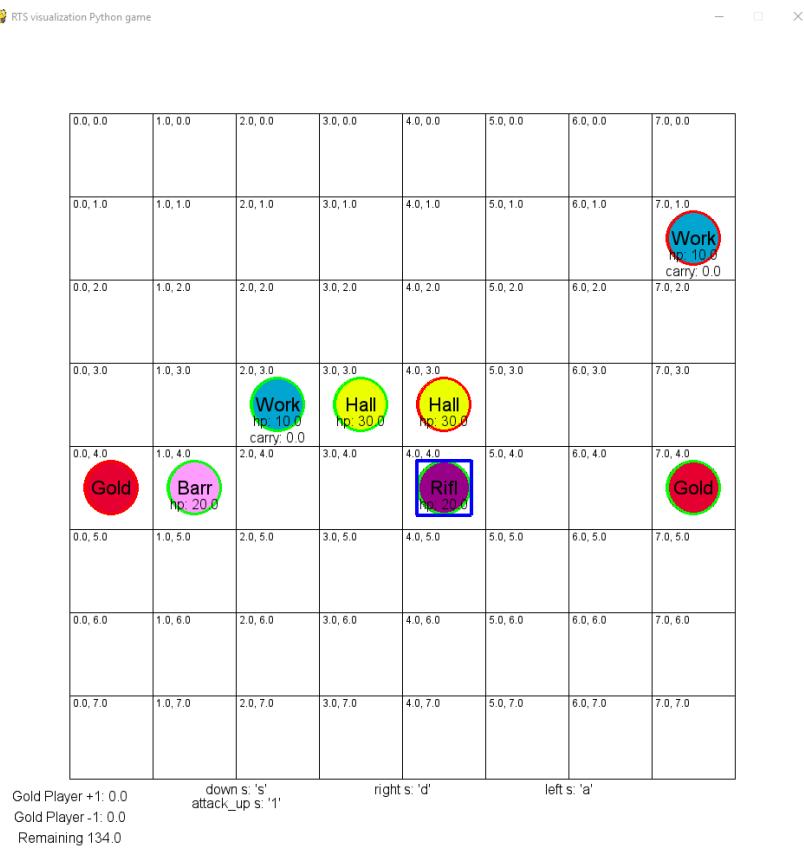
Igralec lahko nadzoruje svoje figure s tipkovnico in miško. Uporabnik mora najprej izbrati figuro z levim miškinim klikom in potem izbrati določeno akcijo, ki je izpisana na zaslonu. Uporabnik lahko spremeni figuro, tako da jo označi s klikom desnega miškinega gumba na prazno mesto.

- premikanje: igralec lahko premakne delavce in vojake za 1 kvadrat v vseh 4 smereh če so prazni s klikom na eno od 4 mest,
- napadanje: z izbrano vojaško figuro lahko uporabnik napade sovražne figure, ki so v dosegu,
- zbiranje in vračanje sredstev: z izbranim delavcem lahko uporabnik porabi sredstva, tako da klikne desno miškino tipko na polje zlata, če

je v dosegu. To velja tudi za vračanje sredstev, vendar mora biti delavec v bližini glavne hiše,

- gradnja: Za gradbene figure in zgradbe mora uporabnik uporabiti eno od bližnjic na tipkovnici.

Prav tako lahko uporabnik igra igro s pisanjem akcij v konzoli.



Slika 6.1: Na zgornji sliki človeški igralec igra izgrajeno strateško igro proti računalniškim nasprotnikom.

6.2 Unreal Engine 4

Unreal Engine 4 je odprtakodni program podjetja Epic Games, ki je namenjen hitri izdelavi računalniških iger. Obstajajo še drugi celostni pogoni kot je na primer Unity.

Unreal Engine 4 omogoča hitro ustvarjanje iger s pomočjo posebnih diagramov (angl. blueprint) in hkrati podpira programski jezik C++, ki ga uporabimo za hitro izvedbo velikega števila matematičnih izrazov [2].



Slika 6.2: Zgornja slika predstavlja igro z uporabniškim vmesnikom.

V temu programu smo oblikovali vizualizacijo igre, ki nam je omogočilo bolj moderen in realističen prikaz realno-časovne strateške igre, saj je vizualiziran seveda v 3-D in ne 2-D kot v Pygame.

V igri lahko izvajamo akcije preko uporabniškega vmesnika, kot tudi z bližnjicami na tipkovnici. Kompleksnejši uporabniški vmesnik nam pa zagotavlja več funkcionalnosti kakor igra izdelana v Pygame. Z njim lahko izbiramo več figur ter jih grupiramo v skupine. Figuram postavljamo več zaporednih akcij, ki jih ta izvršuje v tem vrstnem redu. Človeški igralec ima tudi vpogled na manjši zemljevid, prikazan v levem spodnjem kotu, na katerem vidi sovražne in svoje figure. Nekaj je pa tudi kozmetičnih funkcij kot na primer (fog of war), dnevno-nočni cikel, animacije za vojskovanje, premi-

kanje, nastavitev hitrosti časa ipd. Igra ima tudi implementirana izvajalna drevesa, kjer na primer agentu naročimo nabiranje zlatnikov, ta pa jih sam vrača v najbližje odlagališče zlatnikov. Prav tako so ta drevesa za napadanje, tudi ko je agent nedejaven, da se prosto premika okrog, išče stavbe itd. Igra ima implementirane tudi razne zvočne efekte, učinke delcev za prikaz zmanjšanja življenjskih točk sovražnim enotam. V igri lahko tudi stanje igre shranimo na disk in ga pozneje naložimo, da igro lahko igramo naprej. Ob tem zapisovanju igre smo ugotovili pravi postopek, kako zajeti igro v Unreal Engine in ga zapisati v določen format, ki ga potem lahko lažje prenašamo. To nam je koristilo tudi pri kodiranju igre v JSON format, da smo jo lahko poslali naučenem modelu v Python skripto. Igra podpira tudi preprosto analitiko za primerjavo denarja med igralci, tipi figur ipd.

Igra je zasnovana tudi tako, da se lahko dva igralca med sabo pomerita preko mreže s spletnim podsistemom Steam ali preko lokalne mreže, preko katerega se lahko tudi komunicirata. Oba igralca v tem primeru na svoji lokalnem računalniku poganjata naučena modela in od njega zahtevata priporočila akcij. Ker pa lahko igralca izbereta več različnih map, na katerih bosta igrala, bi bilo potrebno za vsako izmed teh map izgraditi svoj model, saj imajo lahko mape drugačne dimenzije v širini in višini. Prav tako z zdajšnjim algoritmom niso prokriti primeri, da določeno polje ne bi bilo dosegljivo (voda, skalovje), tako da mape morajo biti kvadratne in vsa polja so dosegljiva. Nekatere izmed zgoraj navedenih funkcij je izdelal Nick Pruehs v vtičniku ue4-rts (<https://github.com/npruehs/ue4-rts>), kot recimo nekej izvajalnih dreves, zemljevid, (fog of war).

Potrebno je bilo preslikati akcije in figure v urejevalnik Unreal Engine, da se tam figure primerno premikajo in izvajajo akcije. Potrebno je bilo (mapirati) animacije, efekte, da premikanje in napadanje zgleda dokaj realistično. Ko igralca pričneta z igranjem igre, se naloži TensorFlow model, katerega bosta igralca uporabljala za pridobivanje akcij. Prav tako se inicializira MCTS. Za lažjo komunikacijo z Python modulom in vračanje povratnih klicov v engine smo uporabili vtičnik tensorflow-ue4 (Jan Kanie-

wski <https://github.com/getnamo/tensorflow-ue4>), ki nadgradi vtičnik UnrealEnginePython(20tab <https://github.com/20tab/UnrealEnginePython>) s TensorFlow komponento. Ta komponenta se avtomatično naloži na odjemalčevem računalniku in zagotavlja, da lahko ta uporablja vse funkcionalnosti TensorFlowa.

Ko igralec ali računalniški nasprotnik poda zahtevo za pridobitev akcije, se prvo pridobi vse podatke o igri in se jih mapira v JSON format, da se ga potem pošlje Python skripti. V tem JSON formatu so zapisane figure s kodirniki (x , y , igralec, tip figure, zdravje, nosi zlatnike, zlatniki, preostal čas). Potem se seveda asinhrono pošlje Python skripti ta JSON string, ta pa izgradi novo šahovnico s figurami na podlagi prejetih zakodiranih figur. Skripti se poda tudi ime igralca, ki zahteva akcijo. Skripta za tem pokliče funkcijo za pridobitev verjetnosti akcij, ki izvede določeno število MCTS iteracij in izbere tisto z največjo verjetnostjo. Python skripta pa vrne koordinati x in y ter akcijo, ki se potem ta izvede v celostnem pogonu s preslikanimi svojimi akcijami za gor, dol, napad, naberipd. Potrebno je bilo tudi paziti z orientacijo koordinatnega sistema, saj je bil v Python igri drugače orientiran kot v pogonu Unreal Engine. Potrebno ga je bilo obrniti za -90° v osi Z (pogon Unreal Engine 4: +x gor, +y desno, Python igra: +x desno, +y dol)

Ta predstavitev omogoča tudi igranje dveh človeških igralcev enega proti drugemu preko internetne mreže, kjer vsak igralec pridobiva priporočene ukaze iz modela. Človeški igralec lahko igra tudi proti računalniškim igralcem, ki pa vsaki 2 sekundi zahteva za novo najboljšo akcijo. Lahko si pa ogledamo dva računalniška igralca igrati drugi proti drugemu.

6.2.1 Prenos stanja igre

Za vsakega od igralcev, se vzpostavi svoja komponenta za pridobivanje akcij, ki naloži model da je pripravljen na pridobivanje predikcij. V trenutku lahko samo eden od igralcev pridobi predikcijo, saj pride do konfliktov, če se na primer oba igralca odločita figuro premakniti na isto polje. Ko igralec pošlje prošnjo za predikcijo, zraven še pošlje svoje stanje igre, in kateri igralec je

tisti, ki pošilja prošnjo. Na to algoritem nastavi trenutno igro na poslano in izbere najprimernejšo akcijo. Po izvedbi izbrane akcije, igra počaka še določen čas, da se akcija izvede do konca, za tem pa lahko ta postopek ponovi še drugi igralec.

Človeški igralec lahko akcijo pridobi kadarkoli, a mora počakati da se trenutna prošnja za predikcijo konča. Za njim se postavi v čakalno vrsto tudi računalniški nasprotnik.

Poglavlje 7

Rezultati

V temu poglavju bomo predstavili več konfiguracij učenja in njihovih rezultatov v vizualizaciji Pygame. Naučene modele, ki so med seboj kompatibilni, bomo med seboj primerjali in izpostavili, zakaj je zmagovalna konfiguracija boljša od poražene. Za tem bomo naučen model povezali z pogonom Unreal Engine, kjer bosta dva računalniška nasprotnika igrala igrino drug proti drugemu ter primer, kjer človeški igralec igra proti računalniškem igralcu ki uporablja naučen model za izbiro akcij.

Pri samem učenju smo uporabili TensorFlow 1.9.0, Python 3.6, za vizualizacijo pa Pygame 1.9.4 in Unreal Engine 4.20.

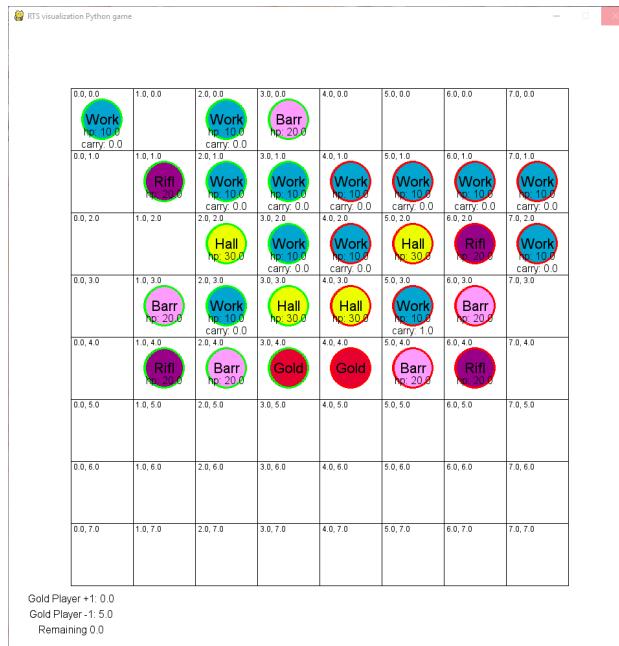
Za učenje smo na voljo podali vse možne akcije.

7.1 Učenje z ustavljeno funkcijo s časovno omejitvijo 100

V prvi fazi smo poskusili učenje modela z manjšo časovno omejitvijo ter dovolj velikim številom iteracij tako da algoritmu potrebuje kar nekaj časa, da dokonča z izvajanjem učenja.

Tip konfiguracije	7.1	7.2	7.3	7.4
časovna omejitev	100	200	200	200 / fun
iteracije	40	20	30 + 30	20
epizod	8	8	8	8
MCTS iskanj	50	50	30	50
primerjave	20	20	20	20
zgodovina učnih primerov	8	8	8	8
epohi	100	100	100	100
začetni zlatniki	20	20	1	1
povečevanje zlatnikov	5	5	1	1
zmanjšanje življenjskih točk	20	20	20	20
količina zdravljenja	20	20	20	20
stroški zdravljenja	5	5	5	5
število polj	8 x 8	8 x 8	8 x 8	6 x 6

Tabela 7.1: V tej tabeli so napisane konfiguracije definicij igre in učenja za spodaj opisane primere učenja modela.



Slika 7.1: Vizualizacija naučenega modela dveh računalniških nasprotnikov po izteku časovne omejitve v Pygame. Uporabljen model je naučen z numeričnim kodirnikom.

7.1.1 Numerično

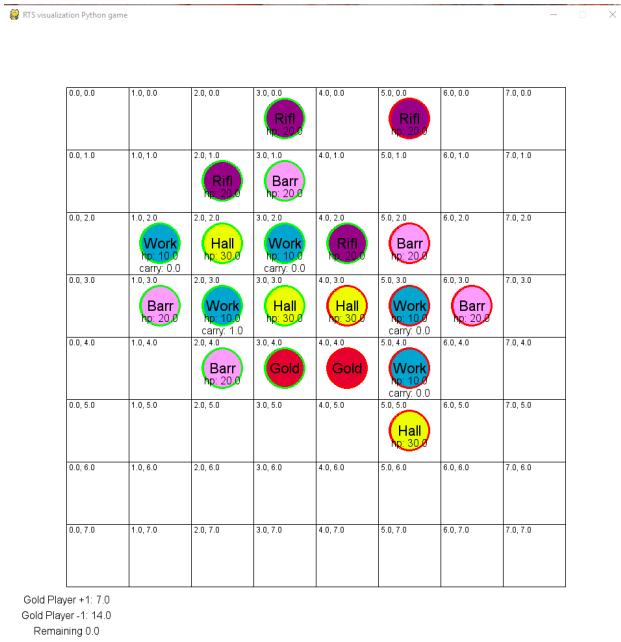
Učenje je potekalo od 2018-11-08 22:20 do 2018-11-10 02:15, kar je 1 dan, 3 ure in 55 minut.

Numerični kodirnik se je bolj osredotočil na izdelavo delavcev, ki so najcenejše figure. Kar pomeni, da takoj ko je igralec imel dovolj denarja za izdelavo figure, jo je izdelal. Igralca sta s svojimi figurami dobro nabirala zlatnike, s tem da sta glavna hiša in polje zlata neposredno drug ob drugem.

7.1.2 One - hot

Učenje je potekalo od 2018-11-10 08:33 do 2018-11-11 16:435, kar je 1 dan, 8 ur in 12 minut.

Z numeričnim kodiranjem se je algoritem osredotočil bolj na izdelavo



Slika 7.2: Vizualizacija naučenega modela dveh računalniških nasprotnikov po izteku časovne omejitve v Pygame. Uporabljen model je naučen z one-hot kodirnikom.

vojaških figur in nabiranju samih zlatnikov. Delavca sta stalno nabirala zlatnike in jih vračala v glavno hišo.

7.1.3 Primerjava

Prav tako smo primerjal numerično kodiranje proti one-hot kodiranju na enakih konfiguracijah modelov ter igre. Izkazalo se je da one-hot kodiranje prinaša boljše rezultate po ocenitvi modelov z igranjem 20 iger drug proti drugemu, kjer je bil rezultat 20:0 za model enkodiran z one-hot načinom.

Algoritem, naučen z OneHot kodiranjem ne more delovati pri Pit z numeričnim kodirnikom in obratno. Vsak model mora imeti svoja določene nastavitev (model učen z OneHot - oneHot, numerični pa numerično nastavitev za kodiranje).

Modele z različnimi konfiguracijami je med sabo težko primerjati, ne

moreta igralca imeti različnih nastavitev za model, razen če jih explicitno prepišemo. Algoritem bi bilo potrebno še predelati, tako da se lahko posameznem igralcu doda konfiguracija pravil igre in učnih konfiguracij, kot tudi tip ustavljene funkcije ipd. S tem bi lahko primerjali dva popolnoma različna modela na isti konfiguraciji igre (npr šahovnica 8 x 8).

7.2 Povečevanje časovne omejitve na 200

V tem koraku smo izbral kodirnih one-hot, ker se je v primerjavi pri prejšnji konfiguraciji obnesel boljše.

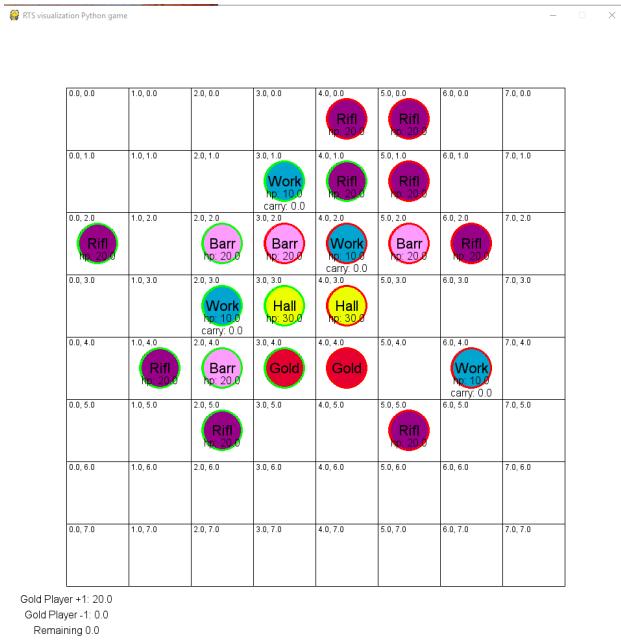
V tem primeru smo povečali časovno omejitev na 200 in znižali število iteracij na 20. Povečali smo časovno omejitev, saj so se pri prejšni nastaviti 100 igre prehitro končvale. Znižali pa smo število iteracij na 20, saj bodo posamezne igre trajale dlje, in smo želeli približno isto časovno dolžino učenja, kot pri prejšnjem primeru, da lahko modela vsaj vizualno primerjamo.

Algoritem se je bolj osredotočil na izdelavo vojaških figur in vojskovanja kot v prvi iteraciji učenja z manjšo časovno omejitvijo

7.3 Sprememba konfiguracij zlata

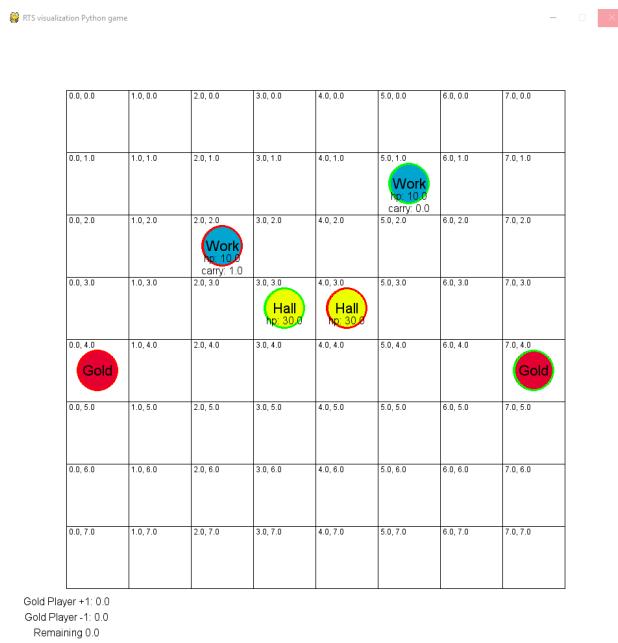
Polji zlata smo pomaknil iz sredine na levi in desni rob, tako da se morajo delavci pomakniti do roba, nabratи zlatnike in jih vrniti nazaj v glavno hišo, ki je pa še vedno na sredini mreže. Prav tako smo spremenili nastavitev, koliko začetnih zlatnikov imata igralca iz 20 na 1, kar dovoli izgradnjo enega delavca na začetku igre. Prav tako smo zmanjšali količino vrnjenega denarja iz 5 na 1, kar bi zagotavljalo, da morata igralca izbrati mnogo pravih sekvenč pomikanja do polja zlata, pridobiti zlatnike in jih vrniti v glavno hišo, preden bi lahko izgradili novo enoto.

Model smo gradili 48 ur v dveh delih po 30 iteracij. Prvi del učenja je potekal od 2018-11-12 21:41 do 2018-11-13 21:42. Prva iteracija ni vračala nobenih koristnih rezultatov, saj sta se izurjena delavca samo sprehajala po



Slika 7.3: Vizualizacija naučenega modela v Pygame z povečano časovno omejitvijo na 200 in znižano število iteracij na 20.

šahovnici, brez da bi nabirala zlatnike. Po nadaljevanju učenja obstoječega modela, ki je potekal od 2018-11-13 22:20 do 2018-11-14 22:26 smo pridobili boljše rezultate, ki pa še vedno niso dobri. Izurjena delavca sta tako kot prej skoraj naključno hodila po šahovnici, s tem da je kdo izmed njiju izvedel akcijo naberi zlatnike. Zlatnikov potem ni vrnil do skoraj konca igre s časovno omejitvijo 200. Po vrnitvi zlatnikov je igralec takoj za tem izdelal dodatno enoto, kar je prineslo dovolj prednosti za zmago. Zlatnike vrne v glavno hišo proti koncu iteracije igre. Mogoče algoritem čaka na konec igre, da preseneti nasprotnika, vendar bolj verjetno je pa da proti koncu igre MCTS začne bolj delovati, saj vrača prave vrednosti stanja igre, ki pa so končna stanja. V tem primeru se je iz delovanja jasno razbral, da MCTS ne vrača primernih rezultatov oziroma ne izboljša uteži modela dovolj dobro. Dober primer je ta, da delavec hodi okrog glavne hiše z nabranimi zlatniki, vendar jih ne vrne v glavno hišo, kar bi povečalo njegov števec točk in se s tem postavil



Slika 7.4: Vizualizacija naučenega modela v Pygame z polji zlata na robovih in glavnimi hišami v sredini.

v prednost pred nasprotnikom.

S tem inkrementalnim učenjem modela v večih korakih smo prikazali, koliko počasi učenje te igre poteka in hkrati dokazali da se model izboljšuje. Počasnost učenja je predvsem zaradi velikega števila akcij, ki se lahko na šahovnici pripetijo na vsakem polju. Vseh možnih akcij je 30, kar privede v $8 \times 8 \times 30 = 1920$ števil, ki jih prejme nevronska mreža kot vhodni nivo. Prav tako je zaradi števila akcij počasno preverjanje katere izmed njih so veljavne, kar upočasni iteriranje stanj iger.

7.4 Zmanjševanje velikosti šahovnice

V tej konfiguraciji učenja pa smo zmanjšali velikost šahovnice iz 8×8 na 6×6 . Prav tako smo zmanjšali število iteracij iz 30 na 20, saj bi teoretično bilo potrebnih manj iteracij za učenje manjše šahovnice.

Zmanjševanje šahovnice je privedlo do pričakovanih rezultatov. Igralca sta občasno nabirala zlatnike, saj je polje zlatnikov v bližini glavne hiše, tako da se delavec postavi med glavno hišo in zlatnike in jih nabira brez da bi se moral premakniti. Zaradi nabranih zlatnikov potem igralca izdelata nove delavce.

Število nabranih zlatnikov in izdelanih delavcev je še vedno majhno.

7.4.1 Ranjujoča ustavitev funkcija

V tem primeru smo pa uporabili ranjujočo ustavitev funkcijo, opisano v sekciji 4.4.1. Po določenem številu potez (v našem primeru 90) funkcija prične izmenično zmanjševati življenske točke igralčevih enot.

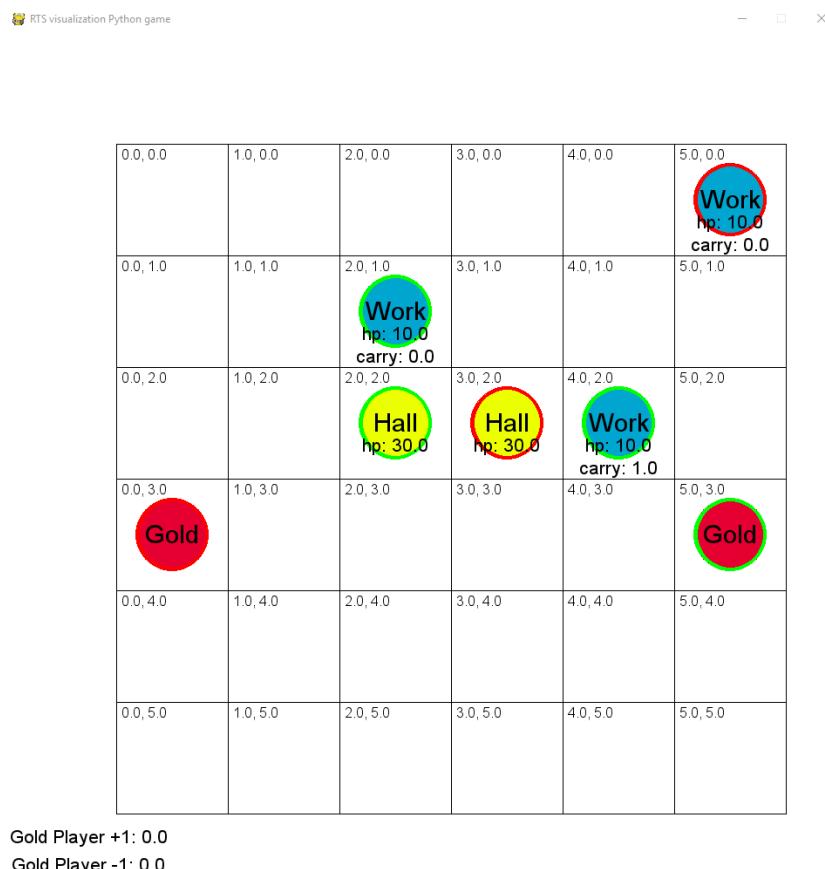
Uporaba funkcije v primerjavi z ustavitevom časom slabše vpliva na učenje, saj je prinesla slabše rezultate. Igralca nabereta manj zlatnikov in posledično izdelata manj figur.

7.5 Vizualizacija rezultatov v Unreal Engine

4

Proti koncu smo rezultate še vizualizirali v pogonu Unreal Engine 4. Izbiranje potez deluje izmenjujoče z zamikom pol sekunde, da se akcije končajo, preden se pošlje nov zahtevek z novim kodiranjem stanja igre.

Igranje proti računalniškem nasprotniku je z uporabo tega algoritma možno, vendar je nasprotnik prelahek, da bi bil primeren za njegovo aplikacijo v modernejše strateske igre.



Slika 7.5: Zgornja slika prikazuje stanje igre pri 70. potezi, ki je naučena z uporabo ranjujoče ustavitevne funkcije.



Slika 7.6: Zgornja slika predstavlja igranje igre dveh računalniških nasprotnikov enega proti drugemu.

Poglavlje 8

Diskusija

Kot smo v poglavju 7 ugotovili, učenje modela poteka zelo počasi, vendar se model uspešno uči. Zasnova definiranja igre in njenega enkodiranja je prava, ker učenje poteka uspešno. Poteka pa počasi zaradi ustavitevnega pogoja in ocenitvenih funkcij, ki ugotovi zmagovalca ob tem ustavitevem pogoju, ki mogoče ni pravi zmagovalec ob koncu igre. Prav tako MCTS naredi zelo majhno število iskanj in ne simulira igre do konca oziroma določene globine, kjer bi ugotovil stanje igre ter to stanje vrnil nazaj, ampak stanje igre pridobi od naučenega modela, ki velikokrat ni pravo.

V sklopu te diplomske naloge se prav tako nismo podajali v spreminjanje zgradbe nevronske mreže, temveč smo vzeli že izgrajeno nevronska mreža, primerno za učenje igre Othello. Mogoče bi ravno spremembra strukture nevronske mreže privredla do boljših rezultatov, saj je matrika stanja igre veliko globja, kakor pri igri Othello.

Pri definiranju igre je veliko problemov povzročalo uravnovešanje parametrov igre, kot na primer število vrnjenih zlatnikov, količina in cena zdravljenja ipd. Dober primer neuravnovešene konfiguracije igre sta bila prav količina in cena zdravljenja, kjer je stalno samo nabiral zlatnike in zdravil svoje figure, tako da je dosegal vedno daljše število narejenih potez. Zaradi tega je učenje potekalo zelo počasi, saj so posamezne igre trajale predolgo da so se končale. Največ problemov pa je pri definiranju igre povzročal ustavitevni pogoj. Oce-

nitev stanja igre ni najboljša, saj je pridobljena po preprosti formuli, ki pa seveda ne vključuje vse dejavnike igre.

Algoritem se da dobro aplicirati v pogon Unreal Engine z nekaj vtičniki, opisanimi v razdelku 6.2. Več igralcev lahko na istem računalniku zahteva priporočila akcij. Če pa igra poteka preko mreže, pa vsak igralec na svojem računalniku poganja algoritom, ki ne ovira delovanje drugih TensorFlow sej na računalnikih drugih igralcev. Sama igra, napisana v Pythonu, na kateri je bil naučen model se pa ne preslika direktno v dejansko igro napisano v Unreal Engine. v tej igri se akcije ne zgodijo instantno, saj vojaške enote in delavci potrebujejo nekaj časa da se premaknejo na drugo lokacijo, izvedejo akcijo kot na primer nabiranje zlatnikov, ki tudi ni instantna. Zaradi trajanja akcij, asinhronosti pridobivanja priporočila od Python modula se lahko zgodi, da stanje igre ni več takšno, kot smo ga poslali v python skripto, in bi bila priporočena akcija z asinhrono skripto drugačna. V nekaterih primerih se ob takih pogojih dve figuri premakneta na isto polje, oziroma izgradi hiša na polju, kjer je trenutno enota. Rešitev za to bi bila vpeljava zahtevanja priporočil akcij ko so akcije zaključene, ter nezmožnost izvajanja akcij v času od zahtevka priporočila do vrnjenega rezultata. Ta rešitev pa ni primerna, saj bi bila primerna za strateške igre, vendar ne za podkategorijo realno-časovnih strateških iger.

Naučen model prav tako vrača samo 1 akcijo, ki pa ne more vključevati večjo skupino figur, kot na primer vseh vojaških enot, da se premaknejo proti sovražniku za napad. Večino teh akcij pa so tudi omejene na sosednja polja, kot na primer pomik gor, dol, napad gor ipd., kar tudi ni primerna aplikacija v dejansko igro, kjer se lahko figure premikajo v poljubnih dolžinah in smereh.

Poglavlje 9

Zaključek

V diplomski nalogi smo povzeli kaj realno-časovna strateška igra je, da smo jo lahko uspešno tudi implementirali. Pregledali smo njihove nivoje nadzorovanja in abstrakcije in s kakšnega zornega kota na njih gledajo nevronske mreže. Za tem smo se podali v raziskovanje algoritmov za učenje te strateške igre in smo naleteli na algoritem Alpha Zero. Na hitro smo pregledali njegovo zgodovino in korake k samostojnemu algoritmu, ki je primeren za reševanje poljubne igre z metodami samoučenja. Ta algoritem smo potem še podrobnejše pregledali, da smo njegov proces učenja in igranja iger razumeli, da smo lahko za tem definirali svojo strateško igro v Pythonu. Definirali smo pravila igre in glavne cilje, ki jih strateška igra mora imeti. Pri tem smo morali paziti da smo se držali okvira Surag-Nairjeve implementacije algoritma Alpha Zero, da smo lahko zanj pripravili svojo strateško igro, ki je kompatibilna z njegovim algoritmom. Za tem smo definirali akcije ki jih figure lahko izvajajo in jih abstrahirali, tako da so za algoritem nedvoumne in hitre. Da pa smo nevronski mreži lahko podali stanje igre, smo ga morali pravilno zakodirati. Izbrali smo desetiški in one-hot način kodiranja, med katerima se je one-hot izkazal uspešnejši, saj ne prioritizira večjih zakodiranih števil kot boljših. Za tem smo ugotovili pravšnji način evalvacije stanja igre in njen ustavitevni pogoj. Definirali smo ustavitevno ranjujočo funkcijo, ki dovojuje bolj aktivnim igralcem daljše igranje, vendar jih kaznuje iz razlogov, ki

sami strateški igri niso naravni. Drugi pristop ustavitvenega pogoja je bil časovni iztek, pri katerem se je po določenem številu potez presodilo, kateri igralec je zmagovalen po določenem kriteriju. Ko smo imeli definirano igro, smo morali še ugotoviti primerne nastavitev uporabiti pri samem učenju igre in izbrati parametre. Za tem smo pripravili vizualizacijo igre v Pythonu z modulom Pygame, ki prikaže igro v preprosti šahovnici in figure s krogci. V pogonu Unreal Engine 4 pa smo pripravili bolj kompleksno strateško igro, ki je boljša predstavitev dejanske realno-časovne strateške igre. V tej igri pošiljamo zahteve za akcije preko vtičnika v python skripto, v katero podamo trenutno stanje igre, nazaj pa dobimo priporočeno akcijo. Zahteve lahko pošilja računalniški nasprotnik ali človeški igralec, ko v najboljšo akcijo ni prepričan. Za tem smo se posvetili predvsem učenju modelov z različnimi konfiguracijami ter jih vizualizirali v Pygame in Unreal Engine. Ugotovili smo, da je učenje počasnejše zaradi kompleksnosti igre, vendar da uspešno poteka. Diplomska naloga je dober prispevek Surag-Nairjevim igram za Alpha Zero, ki razširi preproste igre črno-belih figur v figure večih atributov. V to igro smo pripeljali tudi časovne kompleksnosti in jo razširili z vizualizacijo v Pygame in Unreal Engine 4.

Igra in Alpha Zero General algoritmom sta na voljo na naslednjih repozitorijih:

<https://github.com/JernejHabjan/TrumpDefense2020>

in

<https://github.com/JernejHabjan/alpha-zero-general>

Literatura

- [1] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.
- [2] Marko Kladnik. Primerjava igralnih pogonov unity in unreal engine. Diplomska naloga, Fakulteta za elektrotehniko in računalništvo, Univerza v Ljubljani, 2015.
- [3] Santiago Ontanón. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702, 2017.
- [4] Santiago Ontañon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, and David Churchill. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, IEEE Computational Intelligence Society, 2013.
- [5] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.
- [6] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.