RĪGAS TEHNISKĀ UNIVERSITĀTE
Datorzinātnes un informācijas tehnoloģijas fakultāte
Lietišķo datorsistēmu institūts
Mākslīgā intelekta un sistēmu inženierijas katedra

Akims PARAHINS
Akadēmiskās bakalaura studiju programmas „Datorsistēmas" students
(stud. apl. nr. 181RDB180)

PĀRMEKLĒŠANĀ UN MĀKSLĪGAJOS NEIRONU TĪKLOS SAKŅOTU RISINĀJUMU INTEGRĀCIJA DIVPERSONU SPĒĻU AR PILNU INFORMĀCIJU REALIZĀCIJAI

Bakalaura darbs

Zinātniskais vadītājs
Dr.sc.ing., asociētais profesors
E. LAVENDELIS

Rīga 2021

DARBA IZPILDES UN NOVĒRTĒJUMA LAPA

Bakalaura darbs izstrādāts ***Mākslīgā intelekta un sistēmu inženierijas katedrā.***

Ar parakstu apliecinu, ka visi izmantotie materiāli ir norādīti literatūras sarakstā un iesniegtais darbs ir oriģināls.

Darba autors:
stud. **A.Parahins** ……..……..
(paraksts, datums)

Bakalaura darbs ieteikts aizstāvēšanai:

Zinātniskais vadītājs:
Dr.sc.ing., asociētais profesors **E.Lavendelis** ..……...
(paraksts, datums)

Bakalaura darbs pielaists aizstāvēšanai:

Bakalaura akadēmiskās studiju programmas "Datorsistēmas" direktors:
Dr.sc.ing., asoc. prof. **E.Lavendelis**.....……...
(paraksts, datums)

Bakalaura darbs aizstāvēts Lietišķo datorsistēmu institūta Gala pārbaudījumu komisijas ……...gada……..…sēdē un novērtēts ar atzīmi ( )...…...…
(gads) (datums, mēnesis)

Lietišķo datorsistēmu institūta Gala pārbaudījumu komisijas

sekretāre…….....……..…
(uzvārds, paraksts)

ANOTĀCIJA

[MAŠĪNMĀCĪŠANĀS, SPĒLES KOKA PĀRMEKLĒŠANA, GOMOKU]

Bakalaura darba tips:
2. tips: Aktuālo jomas problēmu risinājumi

Vēsturiski divpersonu spēļu datorrealizācijās ir dominējušas pārmeklēšanas algoritmos balstītas tehnikas, bet pēdējos gados tos sāka aizvietot neironu tīklos balstīti risinājumi, sakarā ar dziļās mašīnmācīšanās strauju attīstīšanu. Apvienojot šīs divas tehnikas, var panākt labākus rezultātus: uzlabot ātrdarbību un lēmumu pieņemšanas kvalitāti.

Darbā tika izpētīta iespēja savienot spēles koka pārmeklēšanas algoritmus un neironu tīklus vienotā mākslīga intelekta risinājumā. Darba teorētiskajā daļā tika izpētīti dziļās mašīnmācīšanās, neironu tīklu un spēles koka pārmeklēšanas algoritmu tēmas, kā arī iespēja iepriekš mīnētās koncepcijas pielietot Gomoku spēles kontekstā.

Darba praktiskajā daļā tika implementēts viens no spēles koka pārmeklēšanas algoritmiem, un, izmantojot mūsdienīgus dziļās mašīnmācīšanās ietvarus, tika izveidots un apmācīts neironu tīkls, kurš pēc tam tika integrēts pārmeklēšanas algoritmā.

Darba pamattekstā ir 44 lappuses, 30 attēli, 7 tabulas, 28 nosaukumu informācijas avoti un 3 pielikumi.

ABSTRACT

[MACHINE LEARNING, GAME TREE SEARCH, GOMOKU]

Bachelor thesis type:
Type 2: Solutions to current problems in the field

Historically, computer implementations of two-person games were based on tree search algorithms, but in recent years they have been replaced by neural network-based solutions due to the rapid development of deep machine learning. By combining these two techniques, it is possible to achieve better results: improve speed and decision-making quality.

The possibility to combine game tree search algorithms and neural networks in a single artificial intelligence solution was investigated. In the theoretical part of the work, the topics of deep machine learning, neural networks and game tree search algorithms where studied, as well as the possibility to apply the previous mined concepts in the context of Gomoku game.

In the practical part of the work, one of the game tree search algorithms was implemented, and using modern deep machine learning frameworks, a neural network was created and trained, which was then integrated into the search algorithm.

The thesis contains 44 pages, 30 figures, 7 tables, 28 information sources and 3 appendixes.

АННОТАЦИЯ

[МАШИННОЕ ОБУЧЕНИЕ, ПОИСК ПО ДЕРЕВУ, ГОМОКУ]

Тип бакалаврской работы:

Тип 2: Решение актуальных проблем в области

Исторически в компьютерных реализациях игр для двух лиц преобладали методы, основанные на алгоритмах поиска, но в последние годы они были заменены решениями на основе нейронных сетей из-за быстрого развития глубокого машинного обучения. Комбинируя эти два метода, можно достичь лучших результатов: повысить скорость и качество принятия решений.

В работе была исследована возможность объединения алгоритмов поиска по дереву и нейронных сетей в едином решении искусственного интеллекта. В теоретической части работы были изучены темы глубокого машинного обучения, нейронных сетей и алгоритмов поиска по дереву игр, а также возможность применения вышеупомянутых концепций в контексте игры Гомоку.

В практической части работы был реализован один из алгоритмов поиска по дереву, и с использованием современных библиотек глубокого машинного обучения была создана и обучена нейронная сеть, которая затем была интегрирована в алгоритм поиска по дереву.

Основной текст работы содержит 44 страницы, 33 изображения, 7 таблиц, 28 источников информации и 3 приложения.

SATURS

IEVADS

Izstrādāt pārmeklēšanā sakņotu mākslīgo intelektu, kurš prastu sastādīt konkurenci cilvēkiem stratēģiskās spēlēs, ir netriviāls uzdevums, bieži vien ne visas cilvēku zināšanas var pilnībā pārnest un izmantot mākslīga intelekta risinājumā. Līdz pat 2015. gadam pārstāvēja viedoklis, ka mākslīgais intelekts nekad nevarēs salīdzināties ar labākiem Go spēlētajiem, spēles sarežģītības dēļ, tomēr, pielietojot jaunākas atziņas dziļās mašīnmācīšanās jomā, datorprogrammai "AlphaGo" izdevās uzvārēt Eiropas čempionu spēlē Go ar rezultātu 5:0 2015. gadā oktobrī (Silver & Hassabis, 2016). Tāpēc ir svarīgi mēģināt pielietot jaunas pieejas mākslīgo intelektu izstrādē spēļu jomā, lai turpinātu attīstību šajā virzienā.

Darba mērķis ir realizēt neironu tīklā sakņotu pozīcijas novērtējumu spēlei «Gomoku» un izmantot to pārmeklēšanā sakņotā algoritmā.

Mērķa sasniegšanai tika izvirzīti sekojošie uzdevumi:

- Veikt esošo pārmeklēšanā sakņoto algoritmu analīzi un izvēlēties piemērotāko Gomoku spēles realizācijai.
- Aprakstīt dažāda veida neironu tīklus, un to pielietošanas iespējas spēļu stāvokļu novērtēšanai.
- Izstrādāt un apmācīt neironu tīklu, ar kura palīdzību būs iespējams novērtēt spēles «Gomoku» pozīcijas
- Realizēt «Gomoku» spēlēs grafa pārmeklēšanas algoritmu labākā gājienā meklēšanai, novērtējot spēles stāvokļus, ar izstrādāto neironu tīklu.
- Izstrādāt Gomoku spēlēs datorrealizāciju.
- Salīdzināt iegūto neironu tīklā sakņotu risinājumu ar citu risinājumu, kurš nav balstīts uz neironu tīkliem.

Darba pirmajā nodaļā ir aprakstīti Gomoku spēles noteikumi un spēlēs koka pārmeklēšanas algoritmi ar dziļāko ieskatu Minimax un Monte Karlo algoritmos. Darba otrajā nodaļā ir sniegts ieskats dziļā mašīnmācīšanās. Darba trešajā nodaļā ir aprakstīts risinājuma izstrādēs process, kā arī tika pārbaudīts gatavs risinājums.

1. GOMOKU UN SPĒLES KOKA PARMEKLĒŠANA

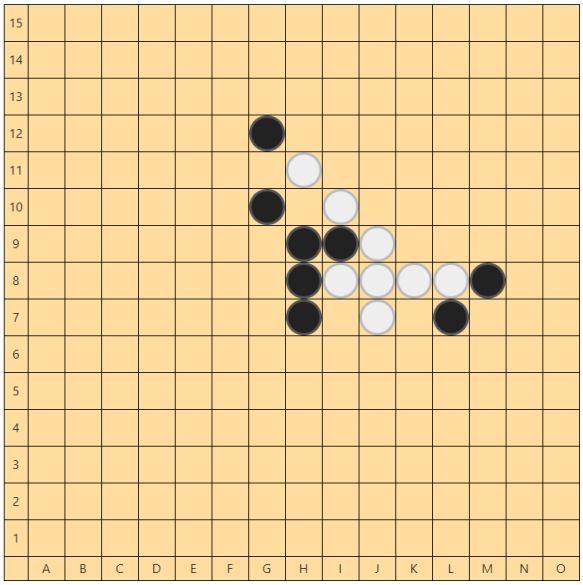
1.1. Gomoku

Gomoku ir divpersonu nulles summas spēle ar pilnu informāciju.

Spēle ar pilnu informāciju ir spēle, kur var paredzēt visus iespējamus iznākumus katram spēles stāvoklim, jo abiem spēlētajiem ir pieejama visa informācija par pašreizējo spēles stāvokli, un spēlē nav nekādu nejaušības faktoru (piemēram, spēļu kauliņu) (Diderich, 2008; Tzeng, 1988; Kaindl 1990).

Nulles summas spēle (Zero-sum game) ir divpersonu spēle, kur viena spēlētāju ieguvums vienāds ar otra spēlētajā zaudējumu. Piemēram, ja spēlē ir punktu sistēma, un viens no spēlētajiem ieguva X punktus, tad no otra spētajā punktu skaita tiek atņemti X punkti. (Tzeng, 1988).

Gomoku parasti spēlē uz laukuma ar izmēriem 15 reiz 15, kuru var redzēt 1.1. attēlā, izmantojot melnus un baltus akmeņus. Spēlētāji secīgi izvieto savus akmeņus uz laukuma, pirmo gājienu veic spēlētājs ar melniem akmeņiem. Spēles mērķis ir izveidot piecu akmeņu garu rindu (vertikāli, horizontāli vai diagonāli), spēlētājs, kurš pirmais sasniegs šo mērķi, uzvar. Uzskata ka Gomoku parādījās Ķīnā agrāk nekā 2000 gadu pirms mūsu ēras. Kopš šī laika parādījās arī vairāki Gomoku spēlēs paveidi, visizplatītākā no tiem ir Renju. Renju galvenā atšķirība ir tādā, ka spēlētājam, kurš pirmais veic gājienu (parasti melnie), ir vairāki iespējamo gājienu ierobežojumi, kuri kompensē pirmā gājiena priekšrocību (Nosovsky & Sokolsky, 1999). Šajā darbā tiks apskatītai tikai pamata Gomoku versija, bez jebkādiem ierobežojumiem un papildu spēles noteikumiem.

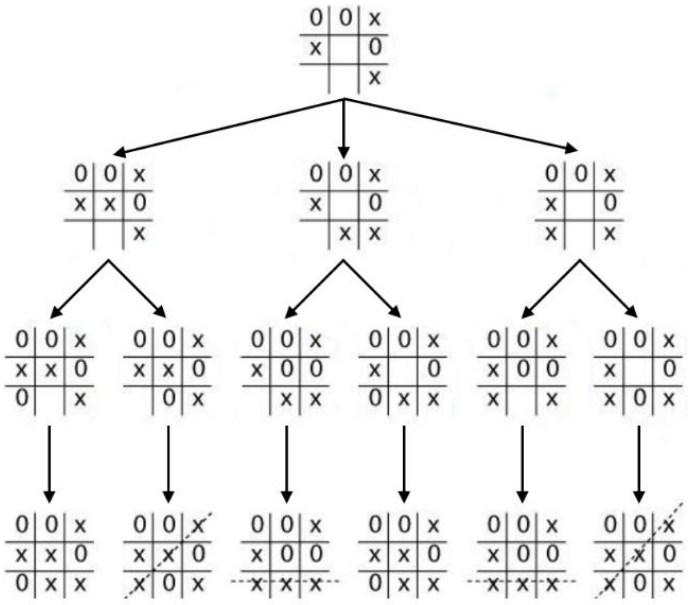


1.1. att. Gomoku spēles piemērs

Kad viens no spēlētajiem veido trīs vai četru akmeņu (***trijnieks*** vai ***četrinieks***) garu rindu (rindā var būt vienā tukša vieta) tiks uzskatīts, ka spēlētājs veido draudu, veicot ***uzbrukuma gājienu***, ja šo rindu iespējams pagarināt līdz 5 akmeņiem pēc viena vai diviem gājieniem. Ja gājienā tiek veidots tikai viens drauds, tad oponents var viegli aizsargāties, tāpēc uzbrukumam un tālākai uzvarai ir nepieciešams veidot vairākus draudus vienā gājiena. Kad uzbrukuma gājienā tiek veidoti vairāki draudi, rezultējošu akmeņu savstarpēju novietojumu sauc par **dakšu** (Nosovsky & Sokolsky, 1999).

1.2. Spēles koks un pārmeklēšanas algoritmi

Spēles koks ir grafs, kura virsotnes atbilst spēles stāvokļiem un loki – iespējamiem gājieniem, spēlēs koka fragmentu var redzēt 1.2. attēlā. Spēles koka konstruēšana ļauj atrast iespējamus spēles stāvokļus un ceļus, kuri noved pie katra atrasta stāvokļa. Bet parasti kad spēles koks ir pārāk liels, ir vērts caurskatīt stāvokļus tikai dažus gājienus uz priekšu. Kad spēles koks ir izveidots, strupceļa virsotnes (virsotnes, kurām nav pēcteču) var būt novērtētas. Novērtējums ir skaitliska vērtība, kura var būt balstīta uz heiristisko novērtējuma funkciju (balstās uz zināšanām par spēli), vai, ja ar šo gājienu spēle beidzas, iepriekš definēta konstanta vērtība (Elnaggar, Aziem u.c., 2014).



1.2. att. Spēles koka fragments spēlei krustiņi-nulles (Modificēts no (Elnaggar, Aziem u.c., 2014))

Vispopulārākie pārmeklēšanas algoritmi balstoties uz (Elnaggar, Aziem u.c., 2014), ir :

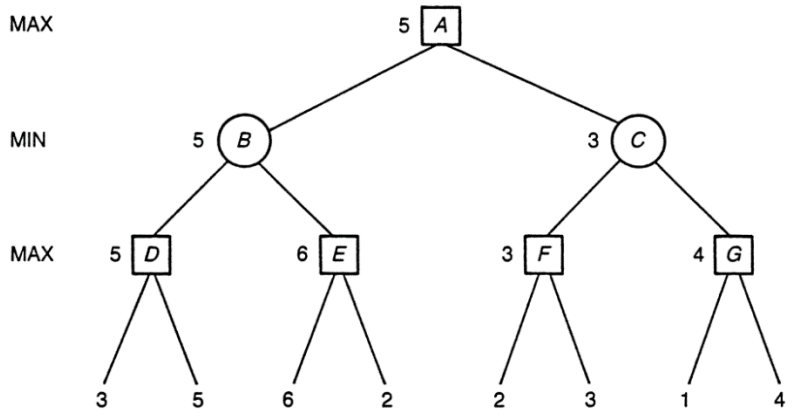
- **MiniMax** – Algoritms rekursīvi apiet visu koku vai koka fragmentu ar noteiktu dziļumu un novērtē virsotnes, balstoties uz pēcteču vērtējumiem, atkarība no tā, kurš spēlētājs veic gājienu apskatāmā stāvoklī, virsotnei tiek piešķirts vai nu vislielākais vai nu vismazākais pēcteča novērtējums.
- **NegaMax** – Algoritms ir līdzīgs Minimax, tikai tas fokusējas uz maksimālajām vērtībām viena spēlētāja gadījumā, bet otra spēlētāja novērtējumu zīmē ir mainīta uz pretēju.
- **Alfa-Beta** – Ir iespējamais MiniMax vai NegaMax algoritmu uzlabojums, kas ļauj atmest spēles koka zarus, kad izpildās noteikti nosacījumi. Tas paaugstina algoritmu efektivitāti, jo nav nepieciešams novērtēt katru pozīciju spēles kokā.
- **NegaScout** – Uzlabots alfa-beta algoritms, kas ļauj nogriezt vairāk virsotņu.
- **N-Best, ProbCut un Multi-ProCut** – Algoritmi, kuri atšķirībā no iepriekšējiem algoritmiem, var mainīt pārmeklēšanas dziļumu.
- **Monte-Carlo Tree Search** – Koka pārmeklēšanas algoritms izvēlās zarus nejauši, un iepriekšēju pārmeklēšanas iterāciju rezultāti tiek saglabāti un izmantoti atkārtoti; ar katru jaunu iterāciju algoritma novērtējumi kļūt precīzāki. Galvenais algoritma trūkums – ir vajadzīgs liels atmiņas apjoms.

1.2.1. Minimax algoritms

Minimax algoritms ir viens no vienkāršākiem algoritmiem, kuru var pielietot spēles koka pārmeklēšanai divpersonu spēlei ar pilnu informāciju un nulles summu (Tzeng, 1988).

Katra spēlētāja mērķis maksimizēt savu ieguvumu, bet tā, kā pēc nulles summas principa viena spēlētajā ieguvums ir vienāds ar otra spēlētajā zudumu, minimax spēles koka virsotnes tiek novērtētas no viena spēlētāja perspektīvas un tiek pieņemts, ka viens spēlētājs maksimizē, bet otrs minimizē virsotņu novērtējumus, attiecīgi minimax algoritms sadala spēles koku uz secīgiem *MAX* un *MIN* līmeņiem, atkarība no tā, kurš spēlētajās (maksimizētājs vai minimizētājs) veic gājienu šajā līmeni (Diderich, 2008; Tzeng, 1988; Kaindl 1990).

Lai novērtētu spēles stāvoklī, tiek konstruēts stāvokļu apakškoks, t.i., spēles koka fragments, ar visiem iespējamajiem spēles stāvokļiem, kurus var sasniegt no izvēlēta stāvokļa. Strupceļa virsotnes tiek novērtētas, izmantojot atbilstošu heiristisku pieeju, vai tai tiek piešķirta noteikta konstanta vērtībā, gadījumā, ja šajā strupceļā virsotnē spēle beidzas. Pēc tām virsotnēm, kuru pēctečiem jau ir dots novērtējums, tiek arī piešķirts novērtējums, balstoties uz to, kādā līmenī atrodas virsotne, attiecīgi ja virsotne ir MAX līmenī, tad virsotnei tiek piešķirts lielākais no tiešo pēcteču novērtējumiem, un pretēji, ja virsotne ir MIN līmenī, tad mazākais no tiešo pēcteču novērtējumiem. Minimax algoritma piemēru var redzēt 1.3. attēlā. Algoritms savā būtībā ir rekursīvs, proti minimax algoritms tiek rekursīvi izsaukts katram pēctecim, ja tā nav strupceļa virsotne. (Diderich, 2008; Tzeng, 1988; Kaindl 1990). Sanāk, ka spēles koka virsotnēm novērtējums var būt piešķirts tikai tad, kad ir zināmi novērtējumi visiem pēctečiem, šis trūkums daļēji tiek atrisināts Alfa-Beta algoritmā.



1.3. att. Ar minimax algoritmu novērtēts spēles koks (aizgūts no (Tzeng, 1988))

1.2.2. Alfa-Beta algoritms

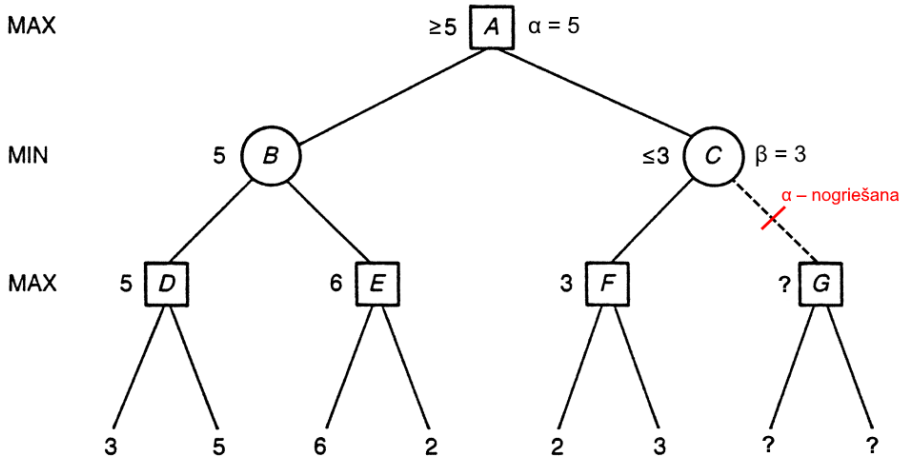
Alfa-Beta algoritms ir minimax algoritma modifikācija, kura ļauj "nogriezt" jeb neaplūkot neperspektīvus spēles koka zarus, kuri nevar ietekmēt pārmeklēšanas rezultātu. Šāda veida uzvedība ir sasniegta piešķirot virsotnēm *"Alfa"* un *"Beta"* vērtības, kuras sauc arī par robežām. Vērtība **α** ir vismazākais un **β** ir vislielākais iespējamais virsotnes novērtējums. Zinot vienu no robežām, var nogriezt zarus un pārtraukt apakškoka pārmeklēšanu, ja (Diderich, 2008; Tzeng, 1988; Kaindl 1990):

- Virsotnes **β** vērtība ir mazāka vai vienāda ar jebkura priekšteča **α** vērtību (tiek sakta par **α – nogriešanu**).
- Virsotnes **α** vērtība ir lielāka vai vienāda ar jebkura priekšteča **β** vērtību (tiek saukta par **β - nogriešanu**).

.

Darbā (Elnaggar, Aziem u.c., 2014) dots intuitīvs piemērs, kas parādā kāpēc nogriešana nemaina pārmeklēšanas rezultātu: izteiksmēs *Max{8, Min{5, X}}* un *Min{3, Max {7, Y}}* nemaina savu rezultātu neatkarīgi no tā, kādās būs X un Y vērtības.

Ja spēlēs koku no attēlā 1.3. pārmeklēt izmantojot Alfa-Beta algoritmu, nevis minimax, tad virsotnes C labo zaru būtu iespējams nogriezt, tā kā parādīts 1.4. attēlā, tāpēc ka pēc virsotnes F novērtēšanas būtu skaidrs, ka virsotnes C novērtējums būs vienāds vai mazāks par virsotnes F = 3 novērtējumu, bet virsotnes A novērtējums lielāks vai vienāds ar virsotnes B = 5 novērtējumu.



1.4. att. Alfa-beta nogriešanas piemērs (modificēts no (Tzeng, 1988))

1.2.3. Montekarlo metode

Minimax un Alfa-Beta algoritmi tiek uzskatīti par pilnās pārlases pieejām, jo tie apskata visus iespējamus stāvokļus (Elnaggar, Aziem u.c., 2014), kas nozīmē, ka šo algoritmu efektivitāte un ātrdarbība ir salīdzinoši maza, jo spēles stāvokļu skaits pieaug eksponenciāli, atkarībā no pārmeklēšanas dziļuma, it īpaši priekš Gomoku, kur spēles sākuma fāzē katram spēlētājam ir pieejami vairāk par 200 iespējamu gājienu, kas minimax gadījumā nozīmē ka spēles kokā ar dziļumu 3 būs vairāk nekā 8 miljonu strupceļa virsotņu, kuras vajadzēs novērtēt.

Montekarlo spēles koka pārmeklēšanas metode (MCTS, Monte-Carlo Tree Search) būtiski samazina pārmeklēšanā iesaistītu virsotņu daudzumu, simulējot nejaušus gājienus stāvokļa novērtēšanai un veidojot asimetrisku spēles koku, kurā virsotnes tiek apskatītas pēc prioritātes.

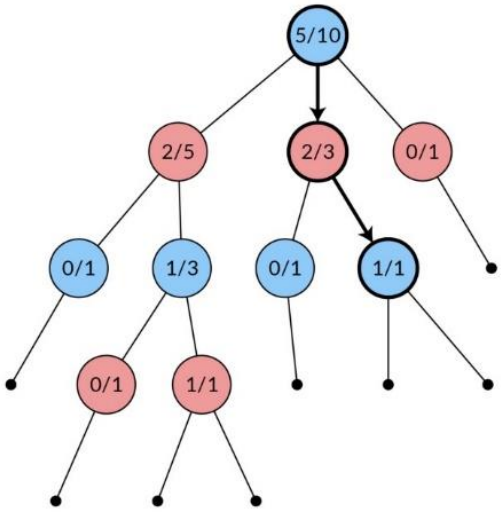
Iepriekš minēto iemeslu dēļ, šajā darba tika implementēts Monte-Karlo spēles koka pārmeklēšanas algoritms, tā kā viņš ir pietiekami efektīvs spēlēs ar lielu zarošanas pakāpi tādās kā Gomoku.

MCTS algoritms sastāv no 4 posmiem, kuri cikliski atkārtojas, uzlabojot spēles stāvokļu novērtējumus un paplašinot spēles koku katrā nākamajā iterācija (Browne, 2012).

Posmu apraksti balstoties uz (Browne, 2012; Winands, 2016; Lui, 2017):

1. Posms: Izvēle (Selection)

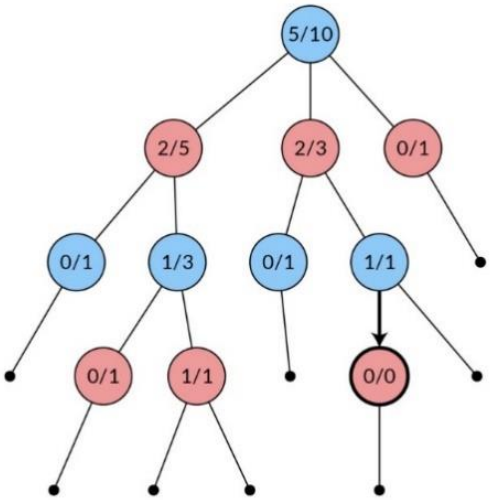
Pirmajā posmā esošajā spēlēs kokā tiek meklēta perspektīva virsotne, kurai vēl nav novērtēti visi pēcteči. Sākot no saknes, rekursīvi tiek izvēlēti perspektīvi gājieni. Eksistē vairākas pieejas nākamā gājiena/virsotnes izvēlei, kuras var būt balstītas uz pašreizējo virsotņu vidējo novērtējumu, apmeklējumu skaitu un citiem faktoriem, vai var būt pilnīgi nejauši izvēlētas. Virsotnes izvēles algoritma piemēru, kurš balstās uz lielāko uzvaru / apmeklējumu proporciju, var redzēt 1.5. attēlā.



1.5. att. Virsotnes izvēle. Treknās bultas atspoguļo izvēlētos gājienus. (aizgūts no (Lui, 2017))

2. Posms: Paplašināšana (Expansion)

Iepriekšējā posmā atrasta virsotne tiek paplašināta - tai tiek pievienots jauns apakšstāvoklis, kurš atbilst līdz šim neapskatītam gājienam. Paplašināto spēles koku var redzēt 1.6. attēlā.



1.6. att. Spēles koka paplašināšana (aizgūts no (Lui, 2017))

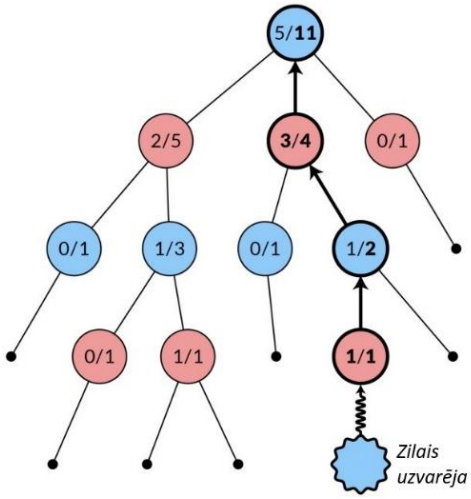
3. Posms: Simulēšana (Simulation)

Tiek imitēta spēle, sakot ar virsotni, kura tika izveidota iepriekšējā posmā, parasti veicot nejaušus gājienus, tiklīdz būs sasniegtas spēles beigas. Parasti ir vērts ierobežot iespējamus gājienus, izmantojot kādu heiristisku pieeju, kura neprasa lielus skaitļošanas resursus, lai nejauši gājieni būtu labāki un lai būtu iespējams ātrāk sasniegt spēlēs beigas.

4. Posms: Atpakaļizplatīšana (Backpropagation)

Kad ir zināms simulācijas rezultāts, kuru parasti reprezentē ar vērtību 1, ja simulēšanas rezultātā spēlētājs uzvārēja un 0 citos gadījumos, tad šis rezultāts tiek piešķirts jaunajai virsotnei un rekursīvi padots jaunas virsotnes priekštečiem, atjaunojot to novērtējumus, līdz brīdim, kad būs sasniegta spēles koka sakne, jeb pašreizējais spēles stāvoklis.

1.7. attēlā var redzēt vienu no atpakļizplatīšanas pieejām, kad katra spēles koka virsotne satur informāciju par pretinieka uzvaru skaitu šajā apakškokā (pirmais skaitlis) un kopējo simulāciju jeb apmeklējumu skaitu apakškokā (otrais skaitlis). Atbilstoši ja simulācijas rezultātā uzvārēja zilais spēlētājs, tad uzvaru skaits tiek palielināts sarkanā spēlētāja līmeņos, un zilā spēlētāja līmeņos, ja uzvārēja sarkanais. Informācijas saglabāšana apgrieztā veidā palīdz vienkāršot izvēles algoritmu, jo spēlētājam, kuram jāveic gājiens, no katras virsotnes uzreiz ir pieejams katra gājiena uzvaru daudzums.



1.7. att. Simulācijas rezultāta atpakaļizplatīšana. (modificēts no (Lui, 2017))

Kad algoritms beidz savu darbību (piemēram pēc noteikta iterāciju skaita, vai laika), jāizvēlas nākamais gājiens, balstoties uz iegūto Monte Karlo pārmeklēšanas koku. Eksistē vairākās stratēģijas labāka gājiena izvēlei (Browne, 2012):

- "Max child": izvēlēties gājienu, kurš noved pie virsotnes ar vislielāko novērtējumu.
- "Robust child": izvēlēties gājienu, kurš noved pie virsotnes ar vislielāko apmeklējumu skaitu
- "Max-Robust child": izvēlēties gājienu, kurš noved pie virsotnes ar vislielāko novērtējumu un apmeklējumu skaitu, ja tāda virsotne neeksistē, tad jāturpina pārmeklēšana.
- "Secure child": izvēlēties gājienu, kurš noved pie virsotnes ar vislielāko apakšējo ticamības robežu (ticamības robežas tiek aprēķinātas, balstoties uz vidējo novērtējumu un apmeklējumu skaitu: jo mazāks ir apmeklējumu skaits salīdzinājuma ar priekšteču, jo lielāka ir robežu amplitūda)

1.2.4. Izvēles posms un izlūkošanas-izmantošanas kompromiss

Vissvarīgākais posms MCTS algoritmā ir *Izvēle*, jo no tā ir atkarīgs, kuri spēles koka zari būs pārmeklēti, un kā izskatīsies galīgais spēles koks. Visvienkāršākā pieeja ir izvēlēties gājienus nejauši, līdzīgi tam, kā tas notiek simulēšanas posmā, bet šī pieeja nav pārāk efektīva, un prasa daudz vairāk laika, lai dotu precīzu pozīciju novērtējumu, tāpēc eksistē vairākas stratēģijas perspektīvu gājienu izvēlēšanai.

MCTS algoritmā ir ļoti svarīgi atrast balansu starp izlūkošanu (exploration) (apskatīt gājienus, kuriem apmeklējumu skaits nav pietiekoši liels, lai uzskatītu, ka novērtējums ir precīzs) un izmantošanu (exploitation) (apskatīt perspektīvus / uzvaru nesošus gājienus). Šo uzdevumu sauc par **izlūkošanas-izmantošanas kompromisu** (Exploration-exploitation trade-off), šis uzdevums attiecas pie **K-roku bandīta problēmas** (K-armed bandit problem) (Browne, 2012).

K-roku bandīta problēmas klasiskais piemērs ir spēļu automāts ar *K* dažādām rokām, pagriežot katru roku, lēmējpersona saņem noteiktu atlīdzību. Lēmējpersonai vajag secīgi izvēlēties rokas pēc tādā principa, kurš maksimizētu kopējo atlīdzību, izmantojot informāciju no iepriekšējiem mēģinājumiem. Katras rokas pagriešanas atlīdzība ir nejauša un atbilst kādām noteiktam varbūtības sadalījumam, kura parametrus lēmējpersona nezina (Mannor, 2017). Sanāk, ka lēmējpersonai vajag noteikt roku ar vislielāko vidējo atlīdzību, balstoties uz saņemtajām atlīdzībām, un izdarīt to, izmantojot pēc iespējas mazāk mēģinājumu.

Eksistē vairākās stratēģijas izlūkošanas-izmantošanas kompromisa atrisināšanai, bet MCTS algoritmā visbiežāk tiek pielietots **UCT** algoritms, kurš ir algoritma **UCB1** paveids

(Winands, 2016). UCB1 algoritms izmanto formulu (1.1) lai aprēķināt **augšējo ticamības robežu** (UCB, upper confidence bound). Izvelēs posmā tiek aprēķināts tekoša stāvokļa tiešo pēcteču UCB vērtības, un pēc tam tiek izvēlēts gājiens, kurš noved pie stāvokļa ar vislielāko UCB. UCB1 algoritms nodrošina, ka stāvokli, kuru apmeklējumu skaits ir mazs, tiks apskatīti vēlreiz, pat ja stāvokļa vidējais novērtējums ir mazs, jo formulas (1.1) daļa $\sqrt{\frac{\ln n_i}{N}}$ palielinās, katru reizi, kad tiek apskatīts cits pēctecis. UCB1 parasti pielieto atlīdzības intervālam [0, 1], gadījumā ja intervāls nav [0, 1], tad algoritmu vajag pielāgot, mainot koeficientu *C*, kurš intervālam [0,1] parasti ir vienāds ar $\sqrt{2}$ (Browne, 2012).

$$\mathrm{UCB} = \bar{x}_i + C\sqrt{\frac{\ln n_i}{N}} \tag{1.1}$$

Kur: $\bar{x}_i$ – i-ta pēcteča novērtējumu vidējais aritmētiskais

n_i – i-ta pēcteča apmeklējumu skaits

N – apskatāma stāvokļa kopējais apmeklējumu skaits

C – izlūkošanas koeficients

1.2.5. Novērtēšanas ierobežojums

Kad ir pieejama stāvokļu novērtēšanas funkcija, ir iespējams saīsināt simulācijas posma dziļumu, izmantojot novērtēšanas ierobežojuma (Evaluation Cutoff) pieeju. Simulācijas dziļums var būt dinamisks, vai fiksēts. Kad simulācijas posmā tiek sasniegts maksimālais dziļums, iegūtā pozīcija tiek novērtēta, izmantojot novērtēšanas funkciju. Šāda veida optimizācija ļauj samazināt simulācijas laiku un uzlabot novērtējumus, gadījumā ja novērtēšanas funkcija ir pietiekoši efektīva un precīza. Ieticamais dziļums šai pieejai ir 5 gājieni vai mazāk (Winands, 2016).

2. DZIĻĀ MĀCĪŠANĀS UN MĀKSLĪGIE NOIRONU TĪKLI

2.1. Mākslīgais neironu tīkls

Mākslīgais neironu tīkls (MNT) ir matemātiskais modelis, kas sastāv no vairākiem savstarpēji saistītiem mezgliem jeb neironiem (Bailer-Jones, Gupta u.c., 2001; Jain, Mao u.c., 1996). MNT arhitektūras iedvesmas avots ir bioloģiskais neironu tīkls, kuri pārstāv cilvēku un dzīvnieku smadzenēs (LeCun, Bengio u.c. 2015; Jain, Mao u.c., 1996).

Mākslīga neironu tīklā pamatā ir neironi. Neironiem ir viena vai vairākas ieejas, aktivizācijas funkcija, nobīde un izeja, savukārt katrai saitei starp diviem neironiem ir savs **svars**. Neirons strādā pēc sekojoša principa: neirons saņem ieejas signālus, kuri tiek reizināti ar atbilstošiem svariem, aprēķinā ieejas signālu summu, atņem nobīdes vērtību, pārveido iegūto starpību atbilstoši aktivizācijas funkcijai un izdod rezultātu reizinātu ar svaru atbilstoši formulai (2.1) (Lippman, 1987; Jain, Mao u.c., 1996).

$$y = f\left(\sum_{i=0}^{N-1} w_i x_i - \theta\right) \tag{2.1}$$

Kur: N – ieeju skaits,

x_i – i-tās ieejas vērtība,

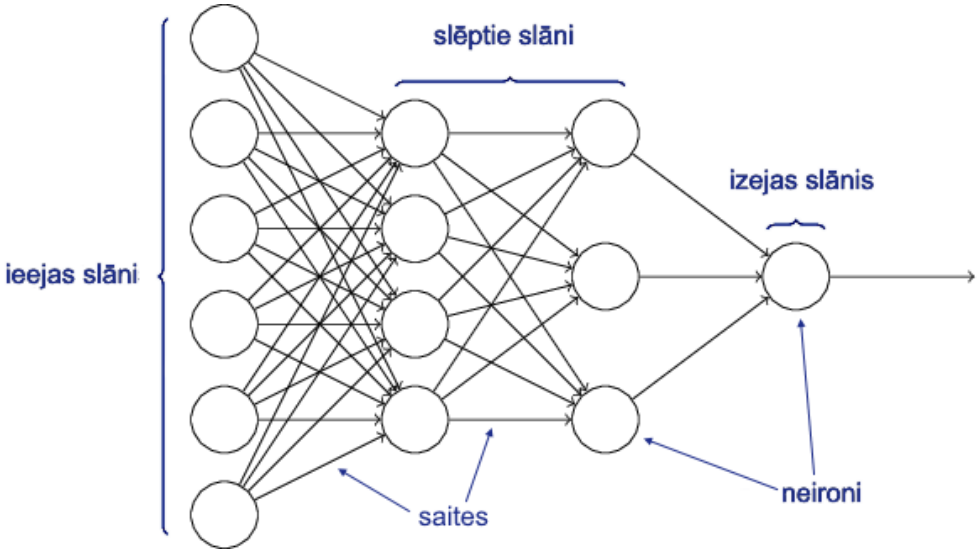
w_i – i-tās ieejas (saites) svars,

θ – nobīde,

f – aktivizācijas funkcija.

2.1.1. Slāņi

Mākslīgajos tiešsaistes neironu tīklos neironi tiek grupēti slāņos. Katra slāņa neironi ir savienoti ar iepriekšējā un nākamā slāņa neironiem ar saitēm. Pirmā slāņa neironu ieejas pieņem ievaddatu vektoru, kurš tiek padots neironu tīklam, un pēdējā slāņa neironi izdod rezultātu. Slāni starp ieejas un izejas slāni tiek saukti par slēptajiem slāņiem (Bailer-Jones, Gupta u.c., 2001). To kā neironi slāņos ir saistīti vienvirziena neironu tīklā var redzēt 2.1. attēlā.



2.1. att. Vienvirziena neironu tīkla arhitektūra (modificēts no (Nielsen, 2015).).

Praksē, rēķinot katra slāņa izejas vērtības, netiek izmantota formula (2.1), jo tas prasa izskaitļot katra neirona vērtību individuāli. Turpretim, izmantojot matricu reizināšanu, var aprēķināt visas slānī esošu neironu izejas vērtības uzreiz, kas ir daudz efektīvāk, jo datorā matemātiskas operācijas notiek daudz ātrāk paralēli, nevis secīgi. It īpaši labi paralēliem aprēķiniem ir piemērots datora grafiskais procesors (GPU, Graphics Processing Unit), kurš ir optimizēts šāda veida darbībām (Николенко, Кадурин u.c., 2018; Elnaggar, Aziem u.c., 2014).

Slāņa izeju aprēķināšana izmantojot svaru un ieejas vērtību matricas parādīta formulā (2.2) (Николенко, Кадурин u.c., 2018).

$$\begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix} = y = f(Wx) = \begin{pmatrix} f(w_1^\top x) \\ \vdots \\ f(w_k^\top x) \end{pmatrix} \tag{2.2}$$

Kur:

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \qquad w_i = \begin{pmatrix} w_{i1} \\ \vdots \\ w_{in} \end{pmatrix}, \qquad W = \begin{pmatrix} w_1 \\ \vdots \\ w_k \end{pmatrix} = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{k1} & \cdots & w_{kn} \end{pmatrix}$$

x – ieejas vektors,

k – neironu skaits slānī,

n – ieeju skaits vai iepriekšēja slāņa neironu skaits,

w_{ij} – i-tā neirona j-tais ieejas svars,

f – aktivizācijas funkcija,

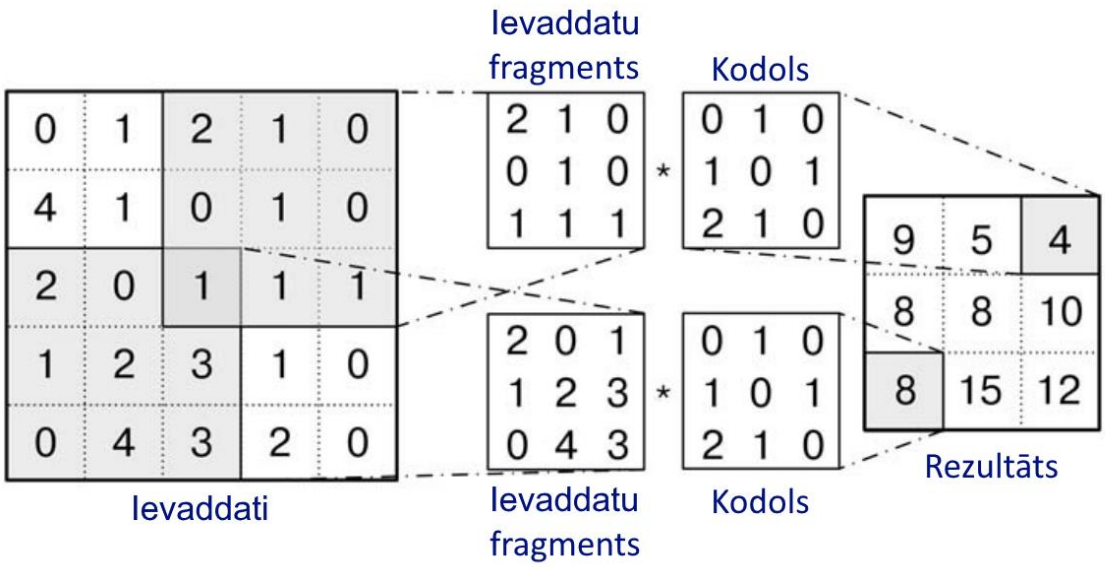
y_i – i-tā neirona izeja.

Lai mākslīgais neironu tīkls realizētu nepieciešamo funkciju, to vajag apmācīt. Apmācības gaitā tiek meklēti svaru un nobīžu vērtības, ar kurām neironu tīkls būtu spējīgs izdod sagaidāmo rezultātu (Bailer-Jones, Gupta u.c., 2001).

2.1.2. Konvolūcijas neironu tīkli

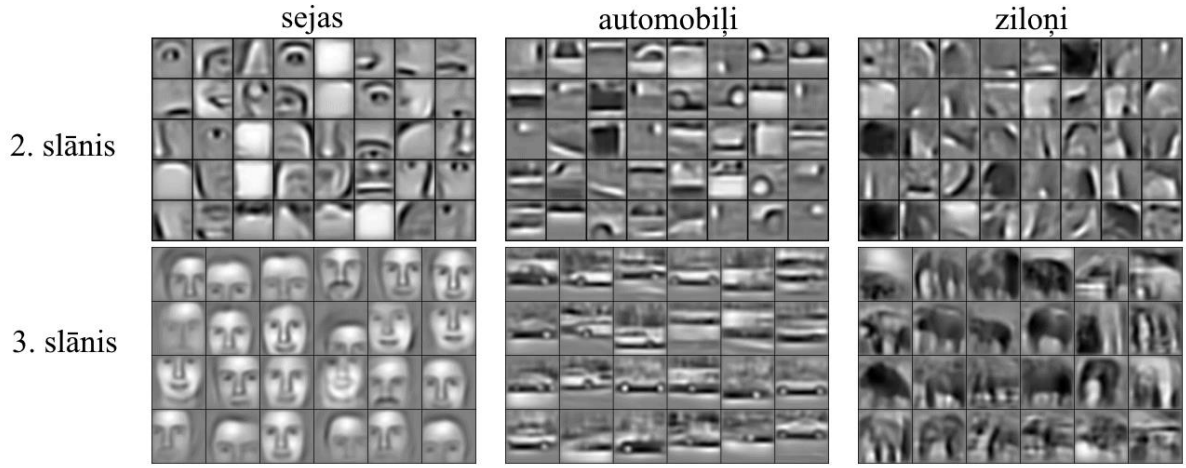
Konvolūcijas neironu tīklu (KNT) arhitektūra vislabāk piemērota divdimensionālu datu apstrādei, kur dati atrodas režģim līdzīgā struktūrā, piemēram, attēlu klasifikācijai. Apmācības procesā KNT spēj iegaumēt blakus esošu datu īpašības un korelācijas neatkarīgi no tā izvietojumā režģī, kā arī, padodot konvolūcijas slāņa rezultātu nākamajam slānim, vairākas mazākas īpašību grupas var būt apvienotas, veidojot jaunas īpašību kopas lielākā mērogā (LeCun, Bottou u.c., 1998; Yamashita, Nishio u.c., 2018). Šī pieeja divdimensionālu datu apstrādē palīdz būtiski samazināt nepieciešamu svaru skaitu, dēļ tā, ka viena slāņa ietvaros svari tiek izmantoti atkārtoti vairākas reizes (Николенко, Кадурин u.c., 2018).

KNT darbības principā pamatā ir kodols, kas ir divdimensionāls masīvs ar svariem, kurš pilda iezīmju noteikšanas funkciju. Vienkāršākā gadījumā ievaddatu kopa tiek sadalīta uz mazākiem gabaliem - logiem, kuru izmērs ir vienāds kodola izmēru, pēc tām katra loga elements tiek reizināts ar attiecīgo kodola elementu un tiek aprēķināta šo reizinājumu summa, kura pēc tam tiek ievietota rezultējoša masīvā (LeCun, Bottou u.c., 1998; Николенко, Кадурин u.c., 2018). Konvolūcijas neironu tīkla darbības piemēru var redzēt 2.2. attēlā.



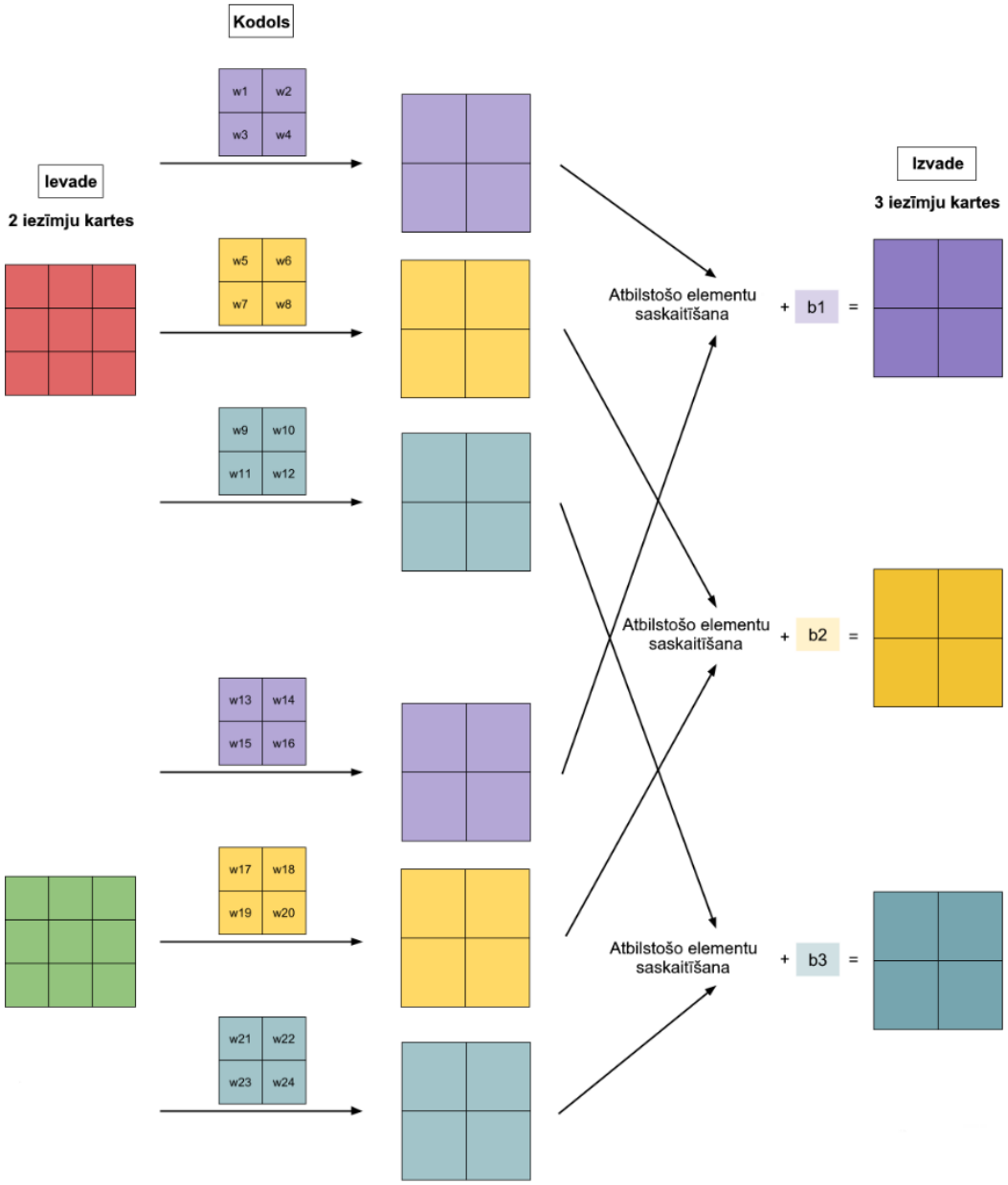
2.2. att. Piemērs, kā tiek rēķināts konvolūcijas rezultāts (modificēts no (Николенко, Кадурин u.c., 2018).)

Katrs kodols ir atbildīgs par noteiktu iezīmju noteikšanu, jo vairāk kodolu, jo vairāk dažādu iezīmju KNT var atrast. Konvolūcijas slāņa darbības rezultātā atrastas iezīmes tiek sadalītas atsevišķos kanālos, kurus arī sauc par *iezīmju kartēm* (feature map), un kuru skaits atbilst kodolu skatam slānī (Николенко, Кадурин u.c., 2018). Dažādu objektu iezīmju vizualizāciju apmācītos neironu tiklos var redzēt 2.3. attēlā.



2.3. att. KNT Iemācītas iezīmēs seju, automobiļu un ziloņu atpazīšanai. (modificēts no (Lee, Grosse, u.c., 2009)

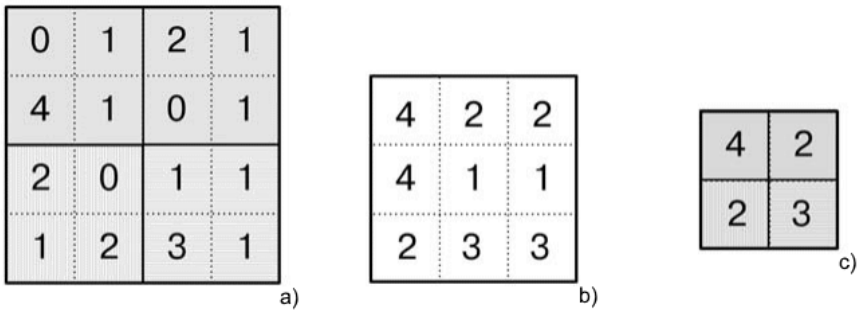
Kad runa iet par krāsainu attēlu apstrādi (piemēram dati .jpeg vai .png formātā), kuriem katru pikseli reprezentē ar trim vērtībām jeb trim krāsu kanāliem – sarkano, zaļo un zilo, tad ir vērts arī konvolūcijas neironu tīkla ieeju sadalīt uz trim iezīmju kartēm, kur katrā karte atbilst krāsas kanālam (Николенко, Кадурин u.c., 2018). 2.4. attēlā var redzēt kā tiek rēķināts konvolūcijas slāņa rezultāts ievaddatiem ar diviem kanāliem (iezīmju kartēm) un trim kodoliem. Kā var redzēt svaru skaits palielinās, jo katram kanālam ir savi kodola svari.



2.4. att. Piemērs, kā tiek rēķināts konvolūcijas slāņa rezultāts ar dīvam ieejas iezīmju kartēm un trim kodoliem (modificēts no (Karim, 2019)).

Lai vēl vairāk samazinātu svaru skaitu konvolūcijas neironu tīklā, tiek izmantoti arī **apvienošanas slāņi** (pooling layers), kurus ievieto pēc konvolūcijas slāņiem. Apvienošanas slāņi samazina iezīmju karšu izmērus, apvienojot vairākus blakus elementus vienā pēc noteikta principa. Viens no biežāk izmantojamiem apvienošanas slāņu veidiem ir "max-pooling" slāni, kuri apvienošanas procesā atstāj tikai vislielāko vērtību. Parasti apvienošana notiek sekojoši: iezīmju kartē tiek sadalīta uz N reiz N logiem, kuri var vai nevar pārklāties atkarība no soļa; katram logam tiek izskaitļota viena vērtība (piemēram "max-pooling" gadījumā - maksimāla vērtība starp visiem loga elementiem); no iegūtam vērtībām tiek veidota jauna iezīmju karte.

Kaut vai apvienošanas procesā informācija tiek daļēji zaudēta, apvienošanas slāņi ļauj neironu tīklam atpazīst iezīmes kuras ir nedaudz nobīdītas (Goodfellow, Benigo u.c., 2016; Nielsen, 2015; Николенко, Кадурин u.c., 2018). 2.5. attēlā var redzēt "max-pooling" slāņa darbības principu.



2.5. att. "Max-pooling" piemērs ar loga izmēru 2 reiz 2: a) sākuma iezīmju karte; b) iezīmju karte pēc apvienošanas, logi ar soli 1 (pārklājas); c) iezīmju karte pēc apvienošanas, logi ar soli 2 (aizgūts no (Николенко, Кадурин u.c., 2018))

2.2. Neironu tīklu aktivizācijas funkcijas

Neironu tīklos parasti izmanto dažādas aktivizācijas funkcijas atkarībā no uzdevuma un neironu tīkla modeļa. Aktivizācijas funkcijai jābūt nelineārai un diferencējamai. Nelinearitāte ļauj neironu tīkla izejai būt nelineāri atkarīgai no ieejas, kas pretējā gadījumā būtu neiespējams. Ja aktivizācijas funkcija nebūs diferencējama, tad nebūs iespējams pielietot gradienta metodi, uz kuras bāzējas gandrīz visas neironu tīklu apmācības metodes (Goodfellow, Benigo u.c., 2016).

Vēsturiski neironu tīklos izmantoja sigmoidas un hiperboliska tangensa (tanh) aktivizācijas funkcijas, bet mūsdienu vienvirziena neironu tīklos tos aizvietoja ReLU tipa funkcijas. Neironu tīkli, kuri izmanto ReLU aktivizācijas funkciju slēptajos slāņos, rada mazāko kļūdu salīdzinot ar tīkliem, kuri izmanto sigmoidas vai tanh funkcijas, kā arī ReLU funkcijas rezultāta aprēķināšana un atvasinājuma noteikšana prasa daudz mazāk skaitļošanas resursu, jo ReLU funkcijas formula (2.4) ir ļoti vienkārša (Goodfellow, Benigo u.c., 2016, Nielsen, 2015).

$$\mathrm{ReLU} = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.3}$$

Funkcijai ReLU eksistē arī vairākas modifikācijas (Николенко, Кадурин u.c., 2018):

- "Leaky ReLU" – negatīvam argumenta vērtībām rezultāts nav vienāds ar 0, bet ar αx, kur α iepriekš definēta konstanta vērtība intervālā $0 < \alpha < 1$. "Leaky ReLU" funkcijas izmantošana palīdz izvairīties no situācijām, kad lielāka daļa no neironiem nedod nekādu signālu, jo parasta ReLU funkcija pārveido visus negatīvus signālus par 0.
- Parametrizēts ReLU – līdzīgi "Leaky ReLU" argumenta negatīvam vērtībām tiek izmantota izteiksme αx, bet parametrizēta ReLU gadījumā, α vērtība nav konstanta un tiek noteikta neironu tīkla apmācības gaitā, līdzīgi tam, kā tiek noteikti tīkla svari.
- ELU (Exponential Linear Unit) – argumenta negatīvam vērtībām ELU rezultāts ir vienāds ar $\alpha(e^x - 1)$.

2.3. Neironu tīklu apmācīšana

Neironu tīklu apmācīšana ir process, kura laikā neironu tīkla svari tiek koriģēti, lai mainītu NT uzvedību. Eksistē vairākas pieejas neironu tīklu apmācīšanai, visbiežāk tiek izmantotas sekojošas pieejas:

1. **Pārraudzītā mācīšanās (Supervised learning)** – ievaddati pirms apmācības sākuma tiek apstrādāti, un katram datu eksemplāram (piemēram, attēls) ir piešķirts rezultāts, kuru sagaida no neironu tīkla, jeb iezīme. Piemērs: ievaddati ir attēlu datu kopa, kur katram attēlam ir dots satura apraksts. Runājot par spēles pozīciju novērtēšanu, pārraudzītās mācīšanās izmantošanai būtu nepieciešams izmantot pozīcijas datu kopu, kur katrai pozīcijai jau ir dots novērtējums, tomēr šajā gadījumā, neironu tīkla novērtējumu kvalitāte būs ierobežotā ar apmācības gaitā izmantoto novērtējumu kvalitāti (Takada, Iizuka u.c., 2017).
2. **Nepārraudzītā mācīšanās (Unsupervised learning)** – ievaddatiem nav definētu iezīmju. Parasti tiek izmantota datu klasterizācijai, dažāda veida informācijas ieguvei no datiem, datu reprezentācijas veidošanai (Goodfellow, Benigo u.c., 2016).
3. **Stimulētā mācīšanās (Reinforcement learning)** – tiek izmantota gadījumos, kad intelektuālais aģents atrodas noteiktajā vidē un var veikt noteiktas darbības. Katrai darbībai var būt piešķirts novērtējums, kuru nosaka pati vide, un pēc

katras darbības mainās stāvoklis, kurā atrodas aģents. No stāvokļa ir atkarīgi darbību novērtējumi, un aģenta iespējamība veikt šīs darbības (Николенко, Кадурин u.c., 2018).

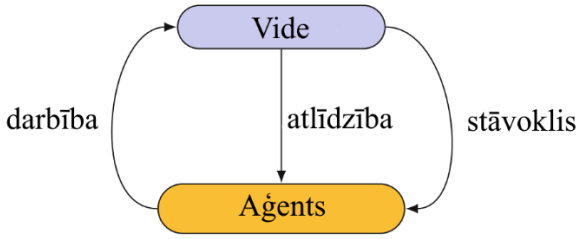
4. **Evolucionārā stratēģija (Evolution Strategies, ES)** ir viena no optimizācijas pieejām, kuru veiksmīgi izmanto neironu tīklu apmācībai (Chellapilla & Fogel, 1999; Salimans, Ho u.c. 2017). ES algoritms, līdzīgi neironu tīkliem, balstās uz atklājumiem bioloģijas nozarē - algoritmā pamatā ir evolūcijas teorija (Fogel, 1994). Tomēr ES algoritms prasa daudz vairāk laika, nekā *nepārraudzītās, vai stimulētās mācīšanās* pieejas, jo ES optimizācija balstās uz stohastiskām mutācijām, nevis uz deterministiskām metodēm, tādam kā *gradienta metode* (Fogel, 1994).

Ir vairāki veiksmīgi neironu tīklu piemērī spēļu pozīciju novērtēšanai, kuri balstās uz pārraudzītu un/vai stimulētu mācīšanu (Agostinelli, Hocquet u.c., 2018). Kaut vai tieši Gomoku gadījumā esoši pētījumi liecina, kā pārraudzīta mācīšanā nedod labus rezultātus (Zhang, 2016). Tas var būt saistīts ar to, ka nav pietiekami daudz iezīmētu datu no reālajām spēlēm, un ar to, ka spēlei Gomoku ir daudz dažādu paveidu ar atšķirīgiem noteikumiem, kas nozīmē, ka atsevišķos gadījumos datu apjoms var būt ļoti mazs. Turpretim stimulētā mācīšanās dod labus rezultātus pozīciju novērtēšanai spēlē "Hex" (Takada, Iizuka u.c., 2017), kura ir pietiekami līdzīga spēlei Gomoku.

2.3.1. Stimulēta mācīšanās

Kā bija iepriekš rakstīts, Stimulēta mācīšanās ir neironu tīklu apmācības pieeja, kura neprasa iezīmētus datus, bet ņem visu nepieciešamu informāciju no vides, kurā neironu tīklos balstīts aģents atrodas. Tieši tāpēc stimulēta mācīšanās ir ļoti efektīva spēļu jomā (Agostinelli, Hocquet u.c., 2018).

Stimulētas mācīšanās ideja – dot iespēju aģentam veikt darbības noteiktā vidē, pēc kuriem vides stāvoklis mainās un aģents saņem atlīdzību, savukārt aģenta uzdevums ir maksimizēt saņemtu atlīdzību (Agostinelli, Hocquet u.c., 2018; Николенко, Кадурин u.c., 2018). Aģenta un vides mijiedarbības principu var redzēt 2.6. attēlā.



2.6. att. Aģenta un vides mijiedarbība (modificēts no (Agostinelli, Hocquet u.c., 2018))

Gadījumā ja laiks ir ierobežots, kopējo kumulatīvo atlīdzību aprēķina pēc formulas (2.4) (Николенко, Кадурин u.c., 2018).

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T = \sum_{k=t+1}^{T} r_k \tag{2.4}$$

Bet, ja laiks nav ierobežots, tad atlīdzību ir vērts samazināt atkarība no saņemšanas laika: jo vēlāk tā ir saņemta, jo mazāk to vērtībā ietekmes kopējo atlīdzību.

Šim nolūkam tiek izmantota konstante $\gamma < 1$ kā parādīts formulā (2.5) (Николенко, Кадурин u.c., 2018).

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.5}$$

Lai atrastu optimālo darbības plānu, daži algoritmi izmanto "vērtības funkciju" V(s), ar kuras palīdzību var iegūt potenciālo atlīdzības vērtību, ja aģents sasniegs stāvokli s pēc formulas (2.6) (Николенко, Кадурин u.c., 2018).:

$$V^{\pi}(\mathrm{s}) = \mathbb{E}_{\pi}[R_t | s_t = s] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right] \tag{2.6}$$

Stimulētā mācīšanās parasti savā pamātā izmanto Markova lēmumu pieņemšanas procesu lai aprakstīt to, kā aģents pāriet no viena stāvokļa uz citu, ja stāvokļu skaits ir ierobežots un nav liels, un katras parejas varbūtība nav atkarīga no iepriekšējiem stāvokļiem (Agostinelli, Hocquet u.c., 2018; Николенко, Кадурин u.c., 2018). Bet gadījumā, ja iespējamo stāvokļu

skaits ir liels (piemēram spēlē Go), tad jāizmanto cita pieeja, kuru sauc par dziļajiem Q-tīkliem (Николенко, Кадурин u.c., 2018).

Tā kā stimulētās mācīšanās pamātā ir ideja par to, ka neironu tīkls pats izvēlas vajadzīgu darbību/gājienu, šo pieeju grūti izmantot šī darbā nolūkos, jo neironu tīklam jābūt integrētām pārmeklēšanās algoritmā, tādā veidā, ka neironu tīkls tikai dod pozīciju nevērtējumus, bet lēmums par to, kādu gājienu veikt, tiek pieņemts ar pārmeklēšanas algoritma palīdzību.

2.3.2. Pārraudzītā mācīšanās

Pārraudzītā mācīšanās ir viena no klasiskajām un visvairāk izmantojamām apmācības pieejām dziļā mašīnmācīšanās jomā, tā rada ļoti lielu efektivitāti, pie nosacījumā, ka ir pieejams pietiekami liels izzīmētu datu apjoms (LeCun, Bengio, u.c. 2015, Николенко, Кадурин u.c., 2018).

Lielāku daļu no uzdevumiem, kurus risina izmantojot neironu tīklus, var sadalīt divas kategorijas (Николенко, Кадурин u.c., 2018):

1. **Regresijas uzdevumos** neironu tīkla rezultāts ir skaitlis noteiktā nepārtrauktā diapazonā.
2. **Klasifikācijas uzdevumos** neironu tīkla rezultāts ir varbūtībās, kuras nosaka piederību pie iepriekš definētām diskrētām klasēm.

Tā kā šajā darbā neironu tīkls tiek izmantots pozīciju novērtēšanai, un sagaidāmais neironu tīkla rezultāts ir skaitliska vērtība, kura nosaka pozīcijas relatīvu izdevīgumu vienam vai otram spēlētajam, var secināt, ka pozīcijas novērtēšana ir regresijas uzdevums.

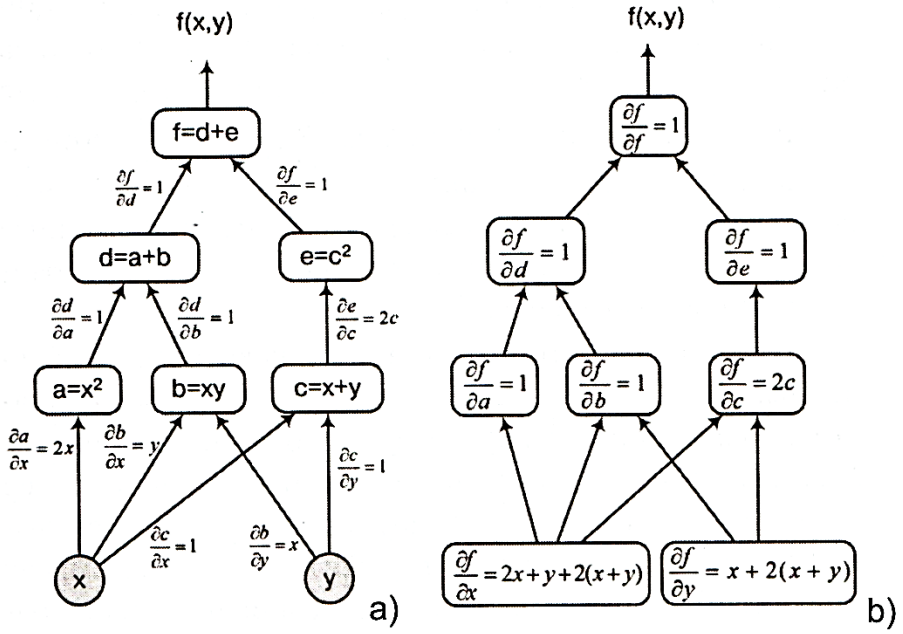
Apmācības procesu bieži vien ir vērts apstādināt noteiktajā laikā, lai nepieļautu neironu tīkla pārmērīgo pielāgošanu (overfitting). Pārmērīga pielāgošana ir nevēlama parādība, kura ir saistīta ar to, ka neironu tīkls var "iegaumēt" trenēšanas datu kopas iezīmēs, nevis mēģināt atrast noteiktas sakarības datu kopā. Lai sekotu tām, vai neironu tīkls nav pārmērīgi pielāgots, izmanto validācijas datu kopu. Validācijas datu kopa nepiedalās apmācības procesā, bet tiek izmantotā, lai pārbaudītu, kā neironu tīkls uzvedas, kad uz ieeju tiek padoti dati, kuri atšķīrās no apmācības datiem. Sanāk ka neironu tīklam nav iespējas "iegaumēt" validācijas datus, un kļūda, kura tiek iegūta uz validācijas datiem, tuvāk reprezentē neironu tīkla precizitāti reālajos pielietošanas apstākļos (Goodfellow, Benigo u.c., 2016; Николенко, Кадурин u.c., 2018).

Var teikt, ka neironu tikla mācīšanās procesā tiek risināts optimizācijas uzdevums, kura mērķis ir atrast tādus svarus, pie kuriem neironu tīkls izdos rezultātus ar vismazāko kļūdu (regresijas uzdevumos parasti tiek rēķināta vidējā kvadrātiskā kļūda), salīdzinot ar sagaidāmo

rezultātu jeb datu iezīmi. Pārsvarā visi optimizācijas algoritmi, kuri tiek izmantoti dziļās mašīnmācīšanās uzdevumos, balstās uz gradienta metodi (Goodfellow, Benigo u.c., 2016; Николенко, Кадурин u.c., 2018).

Gradienta metode izmanto funkcijas atvasinājumus, lai noteiktu kādā mērā parametrus jāpalielina/jāsamazina, lai pēc iespējas ātrāk sasniegtu lokālo maksimumu. Lai atrastu viena parametra vērtības izmaiņu, jāaprēķina funkcijas parciālais atvasinājums. Parciālais atvasinājums pēc parametra nosaka funkcijas vērtības pieaugumu, ja pieaugs tikai tas parametrs, pēc kura tika atvasināta funkcija. Neironu tīkla kļūdu uz apmācības datiem ir iespējams izteikt kā funkciju ar ļoti lielu mainīgu skaitu, tāpēc neironu tīkla svaru noteikšanai ir iespējams pielietot gradienta metodi. (Goodfellow, Benigo u.c., 2016)

Lai efektīvi aprēķināt parciālus atvasinājumus pēc katra neironu tīkla parametra (svara), tiek pielietota atpakaļizplatīšanas metode, kuras pamatā ir skaitļošanas grafa (Computational graph) veidošana un salikto funkciju diferencēšana. Skaitļošanas grafs sadala funkciju uz vairākām operācijām, katra skaitļošanas grafa virsotne ir funkcijas starprezultāts. Savukārt atpakaļizplatīšanas metode virzas pretēji skaitļošanas grafa virzienam, un izmantojot formulu (2.7) secīgi aprēķinā katras virsotnes parciālo atvasinājumu, šādā veidā atkārtoti izmantojot iepriekš iegūtus parciālus atvasinājumus (Goodfellow, Benigo u.c., 2016; Николенко, Кадурин u.c., 2018).



2.7. att. Funkcijas $f(x,y) = x^2 + xy + (x+y)^2$: a) skaitļošanas grafs; b) parciālie atvasinājumi iegūti ar atpakaļizplatīšanas metodi (modificēts no (Николенко, Кадурин u.c., 2018))

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.7}$$

Lai gan pietiekami lielu datu kopu ar novērtētām Gomoku pozīcijām, kuru vārētu izmantot pārraudzītai mācīšanai, nav iespējams atrast, to ir iespējams izveidot, izmantojot pārmeklēšanas algoritmu tādu kā MCTS, jo savā darbībās gaitā algoritms veido novērtējumus katrai apskatītai pozīcijai. Tāpēc šajā darbā tika izvēlēta pārraudzītas mācīšanas pieeja neironu tīkla apmācībai.

3. MAKSLĪGA INTELEKTA RISINĀJUMA IZSTRĀDE

Pārmeklēšanā un mākslīgajos neironu tīklos sakņots risinājums sastāv no dīvām daļām: Monte Karlo pārmeklēšanas algoritma (MCTS) un neironu tīkla pozīciju novērtēšanai. Kaut vai MCTS algoritms ir spējīgs pats novērtēt pozīcijas, neironu tīkls var veiksmīgi papildināt pārmeklēšanas algoritmu: uzlabot novērtējumu precizitāti un ātrdarbību.

Risinājuma implementēšanai tika izmantotas programmēšanas valodas Python un C#.

Valodai Python pieejams plašs dziļās apmācības rīku klāsts, ar kuru palīdzību var viegli veidot neironu tīklu modeļus un apmācīt tos. Kaut vai lielāka daļa no Python bibliotēkām ir izstrādāta izmantojot tādas kompilējamas valodas kā C un C++, kuras nodrošina šo rīku ātrdarbību, programmas, kas uzrakstītas Python valodā izpildās daudz lēnāk nekā kompilējamo valodu ekvivalenti.

C# valoda atšķirībā no Python ir stingri tipizēta, kompilējama valoda, kas ļauj C# programmām izpildīties daudz efektīvāk un ātrāk, kā arī stingra tipizācija ļauj vieglāk atkļūdot kodu un rakstīt sarežģītus algoritmus. Tāpēc Gomoku spēles loģika, pārmeklēšanas algoritmi un neironu tīklu novērtēšana tika rakstītā C# valodā. Savukārt Python tika izmantots neironu tīklu veidošanai un evolūcijas stratēģijas algoritma realizēšanai.

Pilnu risinājuma pirmkodu var apskatīt šeit: https://github.com/akmsprhns-edu/BakD-GomokuTrainer

Neironu tīklu apmācība un spēļu imitēšana risinājuma pārbaudei notika, izmantojot sekojošu skaitļošanas aparatūru un programmatūru:

Procesors – AMD Ryzen 5 2600 3.40 GHz

Grafiskais procesors – NVIDIA GeForce GTX 1070

RAM atmiņa – 16 GB

Operētājsistēma – Microsoft Windows 10 Pro, 20H2 versija

C# - 9.0 versija

.NET – 5.0 versija

Python – 3.8.8 versija, 64-bit

Tensorflow - 2.3.1 versija

Keras - 2.4.3 versija

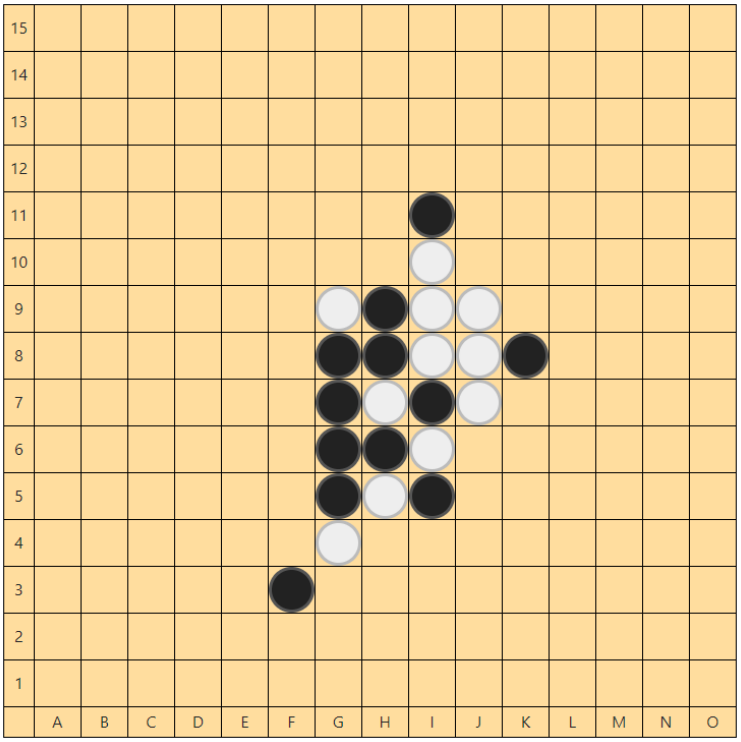
3.1. Spēles koka pārmeklēšana

Tika pieņemts, kā pozīciju novērtējumi būs intervālā [-1, 1]. Jo lielāks ir pozīcijas novērtējums, jo labvēlīgāka tā ir pirmajam spēlētājam, un pretēji, atbilstoši nulles summas spēles principiem. Atbilstoši, ja pozīcijas novērtējums ir 1, tad tajā viennozīmīgi uzvar pirmais spēlētājs, un otrais ja novērtējums ir -1. Ja pozīcijas novērtējums ir tuvu 0, tad pozīciju var uzskatīt par līdzvērtīgu.

Stāvokļa novērtējums ir izskaitļots sekojoši:

- Ja stāvoklī spēle beidzas, un uzvar viens no spēlētajiem tad pozīcijas novērtējums ir vienāds ar 1 ja uzvarēja pirmais spēlētājs, un -1 ja uzvarēja otrais spēlētājs.
- Ja stāvoklī spēle beidzas tādēļ, ka vairs nav iespējamo gājienu, un neviens no spēlētājiem neuzvārēja, tad pozīcijas novērtējums ir vienāds ar 0.
- Parējos gadījumos, kad spēle var turpināties, novērtējums ir vienāds ar vidējo aritmētisko no visiem atpakaļizplatīšanas procesā uzkrātiem novērtējumiem.

3.1. attēlā ilustrētas pozīcijas, ar MCTS algoritmu iegūtos gājienu novērtējumus var apskatīt 3.1. tabulā. Var novērot, ka vislabāko novērtējumu otram spēlētajam (baltajam) saņēma gājieni jeb pozīcijas pēc gājieniem J6 un J10 ar novērtējumiem -0.4417 un -0.2759 attiecīgi, jo tie veido atvērto četrinieku, kuru pirmais spēlējas (melnais) nevarēs aizvērt ar vienu gājienu, un nākamajā gājienā pirmais spēlētājs uzvārēs. Kā arī 3.1. tabulā var redzēt vēl vienu MCTS algoritma īpatnību – visvairāk tika apskatīti tieši uzvaru nesošie gājieni J6 un J10, tas nozīmē ka izvēles posma tie tika izvēlēti visvairāk reizes.



3.1. att. Pozīcijas piemērs.

3.1. tabula. Iespējamo gājienu novērtējumi pozīcijai 3.1 attēlā

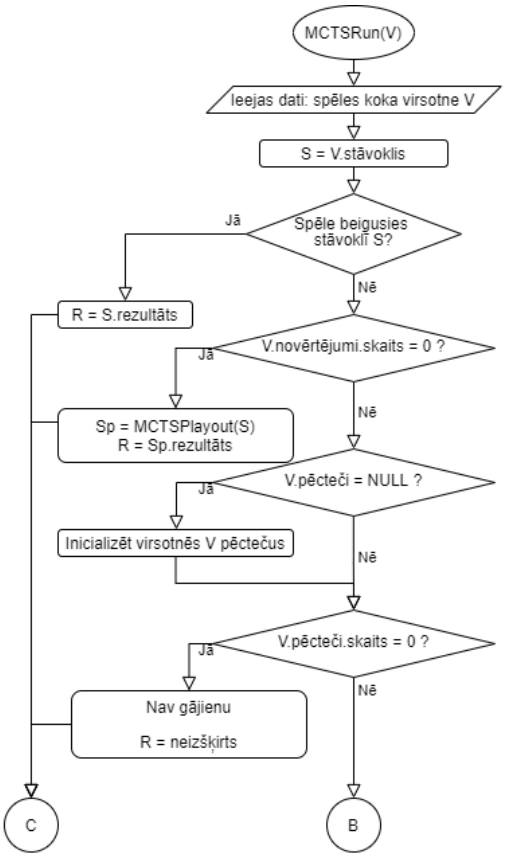
Gājiens	E2	E3	E4	F2	F4	F5	F6
Apmeklējumu skaits	68	57	71	95	132	59	100
Novērtējums	0,1176	0,1579	0,0986	0,0316	-0,0303	0,1525	0,02
Gājiens	F7	F8	F9	F10	G2	G3	G10
Apmeklējumu skaits	35	120	90	31	54	100	66
Novērtējums	0,3143	-0,0167	0,0444	0,3548	0,1852	0,02	0,1212
Gājiens	H3	H4	H10	H11	H12	I4	I12
Apmeklējumu skaits	70	112	130	25	70	135	55
Novērtējums	0,1143	0	-0,0308	0,44	0,1143	-0,037	0,1636
Gājiens	J4	J5	J6	J10	J11	J12	K6
Apmeklējumu skaits	168	158	10772	1185	268	59	35
Novērtējums	-0,0714	-0,0633	-0,4417	-0,2759	-0,1418	0,1525	0,3143
Gājiens	K7	K9	K10	L7	L8	L9	
Apmeklējumu skaits	50	90	421	64	33	137	
Novērtējums	0,2	0,0444	-0,1924	0,125	0,3333	-0,0365	

Izvēles posmā tika pielietots UCB1 algoritms, izmantojot formulu (1.1) ar konstanti $C = 2\sqrt{2}$, tādēļ kā rekomendējama vērtība novērtējumu intervālam [0, 1] ir $C = \sqrt{2}$, un tā kā tiek

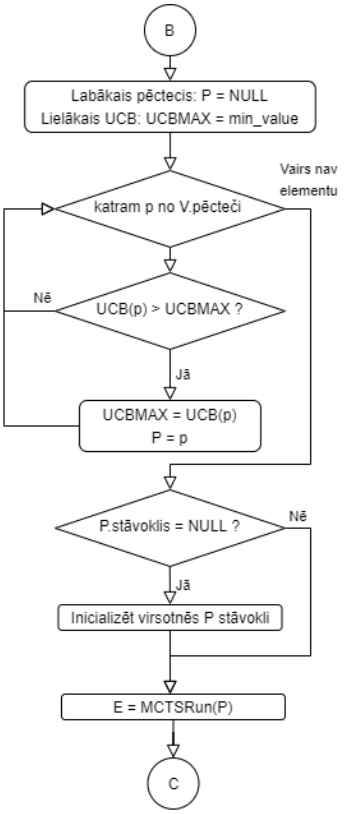
izmantots novērtējumu intervāls [-1, 1], ir vērts palielināt šo konstanti divreiz, lai kompensētu divreiz lielāku novērtējumu amplitūdu.

3.1.1. MCTS implementācija

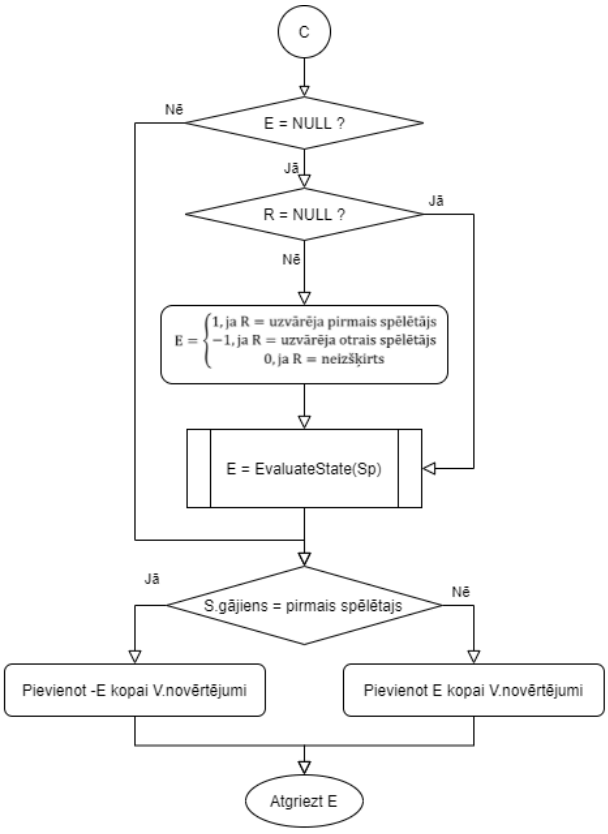
Pamata MCTS algoritma loģika atrodas klasē `MonteCarloTreeSearch` metodē `MCTSRun`, šis metodēs pirmkods ir atrodams 1. pielikumā, savukārt `MCTSRun` algoritmu var redzēt 3.2. - 3.4 attēlos. 3.2. attēlā var redzēt nepieciešamas pārbaudēs, kuras jāveic, pirms turpināt izvēles posmu. 3.3. attēlā ir paradīts, kā tiek izvēlēta nākama spēles koka virsotne, balstoties uz UCB1 algoritmu. 3.4. attēlā var redzēt, kā tiek papildināta apskatāmas virsotnes novērtējumu kopa. `MCTSRun` metode savā darbības laikā izsauc pati sevi, lai turpinātu izvēles posmu un izpildes beigās atgriež iegūtu novērtējumu, šādā veida novērtējums rekursīvi atpakaļizplatās, kad funkcijas atgriež savas vērtības izsaucējam.



3.2. att. Metodes MCTSRun algoritma fragments A

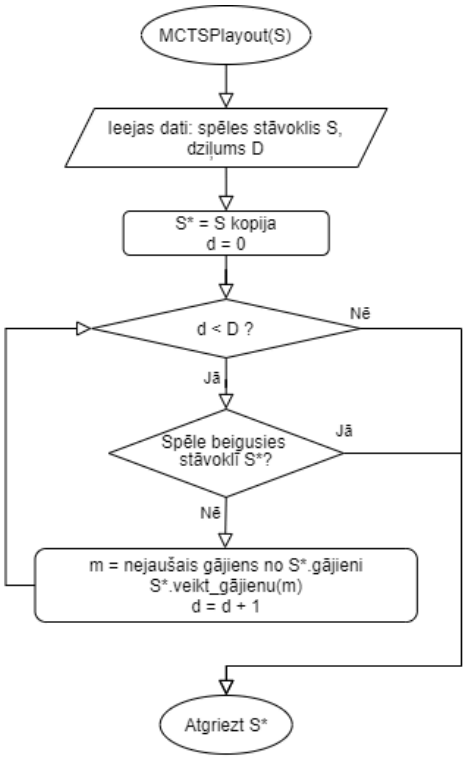


3.3. att. Metodes MCTSRun algoritma fragments B



3.4. att. Metodes MCTSRun algoritma fragments C

`MCTSRun` īsteno MCTS algoritma izvēles, paplašināšanas un atpakaļizplatīšanas posmus, savukārt simulēšanas posmu īsteno metode `MCTSPlayout`, kura tiek izsaukta no metodes `MCTSRun`, ja apskatāmai virsotnei vēl nav neviena novērtējuma. `MCTSPlayout` pirmkods ir atrodams 2. pielikumā, savukārt metodes algoritmu var redzēt 3.5. attēlā.

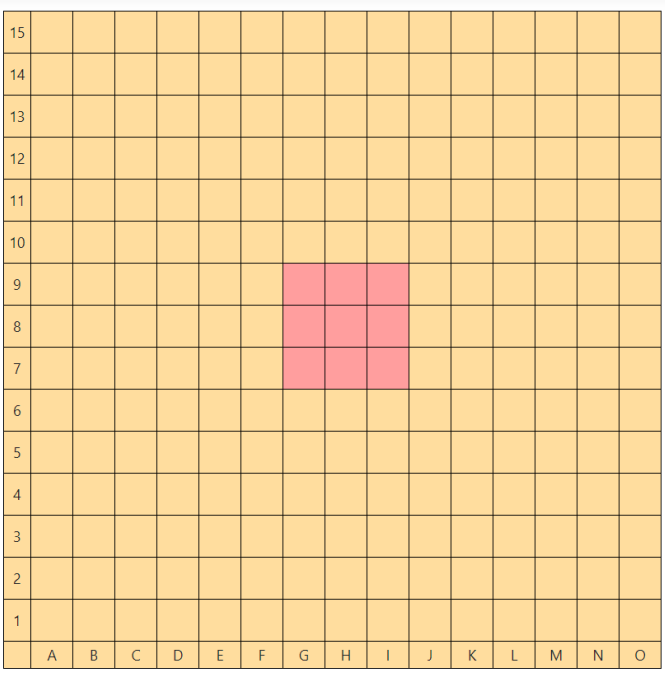


3.5. att. Metodes MCTSPlayout pirmkods.

3.1.2. Problēmsfēras zināšanu izmantošana

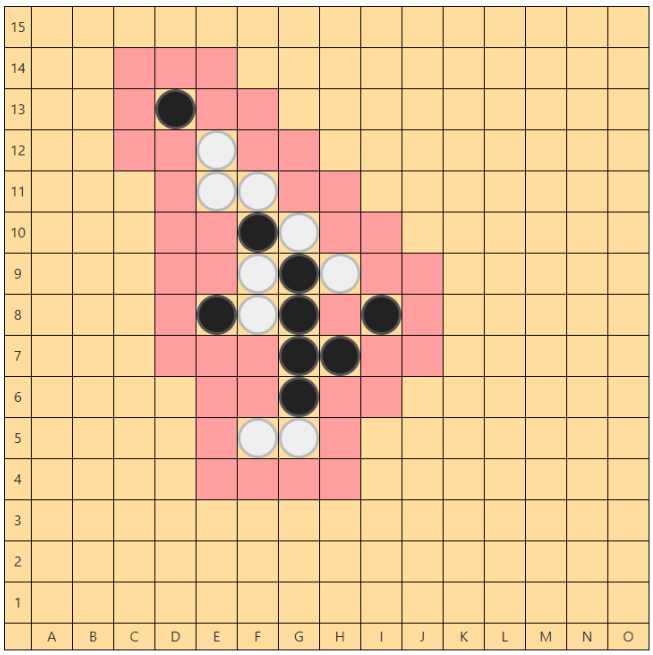
Lai samazinātu zarošanas pakāpi un vienkāršotu pārmeklēšanu, izvēles un simulēšanas posmos tika izmantoti daži heiristiski pieņēmumi, lai izslēgtu sliktus gājienus, kurus nav vērts apskatīt.

Visnozīmīgākie lauki ir spēles laukuma centrā, jo tajā ir vairāk iespējamo gājienu, bet lauki tuvāk spēles laukuma mālam ir sliktāki, jo tur var rasties tāda situācija, kā akmeņu rinda būs ierobežota ar laukuma robežu. Tāpēc spēlēs sākumā, pirmais gājiens ir ierobežots ar centrālu 3x3 kvadrātu (attēlā 3.6. atzīmēti ar sarkanu krāsu), jo nav vērts apskatīt pārējos laukus tuvāk malām. Pateicoties šim ierobežojumam, pēc pirmajā gājienā, iespējamo pozīciju skaits samazinās no 255 līdz 9, kas būtiski samazina spēles koku.



3.6. Att. Tukšs laukums. Lauki, kuros var novietot pirmo akmeni, iekrāsoti sarkanā krasā.

Partijas vidusposmā, kad spēles laukumā jau atrodas vairāki akmeņi, ir iespējams izslēgt gājienus, kuri novieto kauliņus par tālu no centrā vai jau esošiem akmeņiem, jo šāds gājiens nekā neietekmē tekošu spēles pozīciju un dod pretiniekam priekšrocības. Tiek pieņemts, ka gājienu var veikt tikai tad, ja kaut vienā blakus laukā jau atrodas cits akmens, neatkarīgi no krāsas. Piemēru, ar iespējamiem gājieniem, var redzēt 3.7. attēlā.



3.7. Att. Pozīcijas piemērs. Atļauti gājieni iekrāsoti sarkanā krasā.

3.2. Neironu tīkla arhitektūra

Neironu tīkls tika izveidots, izmantojot bibliotēkas Keras rīkus, un balstās uz konvolūciju neironu tīklu piemēriem no darbā (Chollet, 2017). Tā kā pārraudzītās mācīšanās pieeja ar Keras bibliotēku ļauj pietiekami ātri apmācīt neironu tīklu (vienas stundas laikā uz augstāk minētas aparatūras) un viegli mainīt neironu tīkla struktūru, eksperimenti tika veikti ar dīvam neironu tīklu arhitektūrām, kuram galvenā atšķirība bija konvolūciju slāņos:

- 1. variantā tika izmantoti 3 konvolūciju slāņi ar kodola izmēru 2 reiz 2
- 2. variantā tika izmantoti 4 konvolūciju slāņi ar kodola izmēru 3 reiz 3

Neironu tīkla otra varianta definējums valodā Python, izmantojot bibliotēku Keras, ir atrodams 3. pielikumā.

Neironu tīklam ir divas ieejas:

- Ieeja A – 225 vērtību gara ieeja. Uz šo ieeju tiek padota pozīcija, kura tiek iekodēta viendimensionāla datu masīvā, izmantojot 3 vērtības – 0, 1 un -1. Katra vērtība atspoguļo attiecīga spēlēs laukuma lauka stāvokli: ja i-tais lauks ir brīvs tad i-tais elements masīvā ir vienāds ar 0; ja lauks ir aizņemts ar pirmā spēlētāja akmeni tad i-tais elements vienāds ar 1; ja lauks ir aizņemts ar otrā spēlētāja akmeni tad i-tais elements vienāds ar -1. Lauki ir numurēti atbilstoši formulai (3.1); lauku, rindu un kolonu numurēšana sākas ar 0; kolonas ir numurētas no kreisās puses uz labo; rindas ir numurētas no augšas uz lēju.
- Ieeja B – Uz šo ieeju tiek padota tikai viena vērtība: 1, ja pozīcijā nākamais gājiens jāveic pirmajam spēlētājam; -1, ja nākamais gājiens jāveic otrajam spēlētājam.

$$\text{indekss} = \text{r} * 15 + \text{k} \tag{3.1}$$

Kur: r – rindas indekss,
k – kolonas indekss.

Neironu tīklā visi aktivizācijas slāņi, izņemot pēdējo, izmanto aktivizācijas funkciju "Leaky ReLU", savukārt pēdējais aktivizācijas slānis izmanto hiperboliska tangensa aktivizācijas funkciju, kas ierobežo neironu tīkla rezultātu ar intervālu (-1 , 1), lai novērtējumu uzreiz varētu izmantot pārmeklēšanas algoritmā.

3.3. Neironu tīkla apmācībā

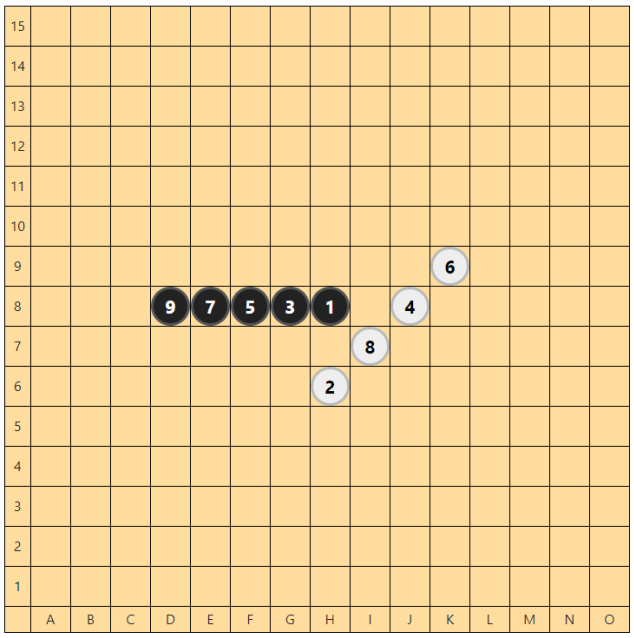
3.3.1. Evolucionāras stratēģijas metode

Tika mēģināts apmācīt neironu tīklu, izmantojot *evolucionāras stratēģijas* (ES) metodi, izmantojot OpenAI piedāvātu algoritmu. Apmācības procesā tika veidoti vairāki neironu tīklu modeli, ar vienādu arhitektūru bet atšķirīgiem svariem. Tika pielietotas divas pieejas:

1. Katrs tīkls tika izmantots kopā ar vienkāršu algoritmu, kurš izmantoja neironu tīklu, lai novērtētu pozīcijas pēc viena gājiena, un veica gājienus bastoties uz labāko novērtējumu. Lai varētu simulēt pēc iespējas vairāk spēļu vienā laika posmā, apmācības laikā netika izmantoti spēles koka pārmeklēšanas algoritmi, jo tie prasa daudz skaitļošanas resursu, kas būtiski samazina apmācības ātrumu. Vienas ES iterācijas vidējais ilgums ir ap 30 sekundēm uz augstāk minētas aparatūras.
2. Katrs tīkls tika izmantots kopā ar MCTS pārmeklēšanas algoritmu ar samazinātu iterāciju skaitu – 500 un ierobežoto simulācijas dziļumu - 5, kas nozīme kā katram gājienam jānovērtē 500 pozīcijas, un tās pozīcijās, kurās spēle nebeidzas, tiek novērtētas ar neironu tīklu. Šī pieeja ir daudz lēnāka nekā pirmā, viena ES iterācija aizņem ap 4 minūtēm.

Apmācība notiek cikliski, katrā ES iterācijā tiek imitētas spēles starp vairākiem neironu tīkliem, pēc tām katram neironu tīklam tika piešķirts noteikts punktu skaits, atkarībā no uzvaru skaita. Šie punkti pēc tam tiek apstrādāti ar ES algoritmu, kurš pēc tam ģenerē jaunus svarus tīkliem nākamajai apmācības iterācijas, balstoties uz tīklu, kurš ieguva labāko rezultātu iterācijā. Pēc 2000 iterācijām pirmās pieejas gadījumā, un 500 iterācijām otras pieejas gadījumā apmācītais neironu tīkla modelis tika integrēts Monte Karlo pārmeklēšanas algoritmā.

Pirmie testi paradīja, ka evolucionāras stratēģijas pieeja, nav piemērota šī uzdevuma risināšanai, tādēļ kā vienas spēles imitācija aizņem pārāk daudz laika, kas neļauj iegūt nepieciešamu spēļu daudzumu evolucionāras stratēģijas realizēšanai. Pat pēc 8 apmācības stundām neironu tīkls dod kļūdainus novērtējumus. Kā rezultāts neironu tīklā sakņots risinājums nevar uzvarēt tīro MCTS algoritmu. Parasti spēle beidzas jau pēc 9 gājieniem, jo neironu tīklā sakņots risinājums nemēģina aizsargāties un ļauj oponentam uztaisīt piecinieku un uzvarēt. Tāpēc tika pieņemts lēmums ES metodes vietā izmantot pārraudzītas mācīšanas pieeju. Tipisko spēlēs piemēru starp MCTS algoritmu un neironu tīklu risinājumu, kas tika apmācīts ar ES metodi var redzēt 3.8. attēlā.



3.8. att. Spēles piemērs. Tīrais MCTS algoritms (melnie), pret neironu tiklā sakņotu risinājumu apmācīto ar ES metodi (baltie). Melnie uzvarēja.

3.3.2. Pārraudzītā mācīšanas

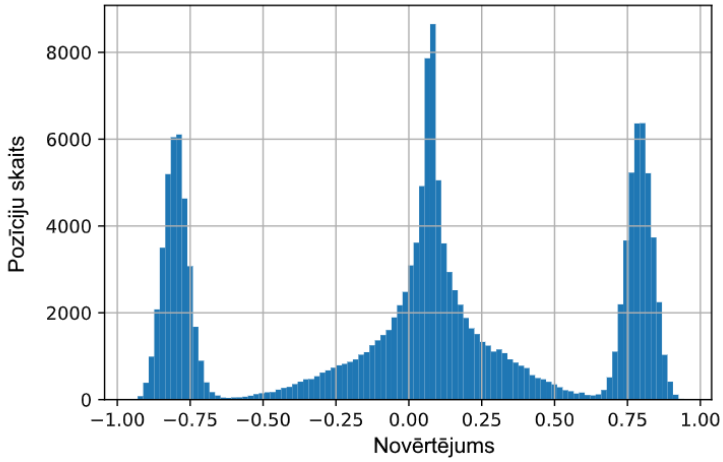
Lai neironu tīklu būtu iespējams apmācīt, izmantojot pārraudzītās mācīšanas pieeju, nepieciešams iegūt pietiekami lielu iezīmētu datu apjomu. Tā kā brīva piekļuvē nebija iespējams atrast piemēroto datu kopu, nepieciešamie dati tika uzģenerēti, izmantojot MCTS algoritmu. MCTS algoritms darbības procesā veido novērtējumus katrai apskatītai pozīcijai spēlēs kokā, bija nepieciešams imitēt vairākās spēlēs, kur algoritms spēlē pats ar sevi. Spēlēs procesā, pirms katrā gājiena, novērtētās pozīcijas tika saglabātas tekstā failā. Katra faila rinda satur informāciju par vienu pozīciju, savukārt katra rindā satur 228 vērtības. Rindas struktūra atbilstoši 3.9. attēlam:

- Spēles laukums - 255 vērtības, katra vērtība atbilst vienam laukam: 0 ja lauks ir brīvs; 1 ja lauks aizņemts ar pirmā spēlētāja akmeni; -1 ja lauks aizņemts ar otra spēlētāja akmeni
- Tekoša spēlētāja gājiens – 1 ja šajā pozīcijā gājiens jāveic pirmajam spēlētājam, -1 ja otrajam.
- MCTS apmeklējumu skaits – cik daudz reizēs algoritms apmeklēja šo stāvokli
- MCTS novērtējums – gājienā novērtējums no -1 līdz 1.

Nr.	1	2	3	4	...	222	223	224	225	226	227	228
Dati	**0**	**0**	**0**	**0**	**....**	**0**	**0**	**0**	**0**	**1**	**1080**	**-0,244444444**
Skaidrojums	Spēles laukums									Spēlētājs	Apmeklējumu skaits	Novērtējums

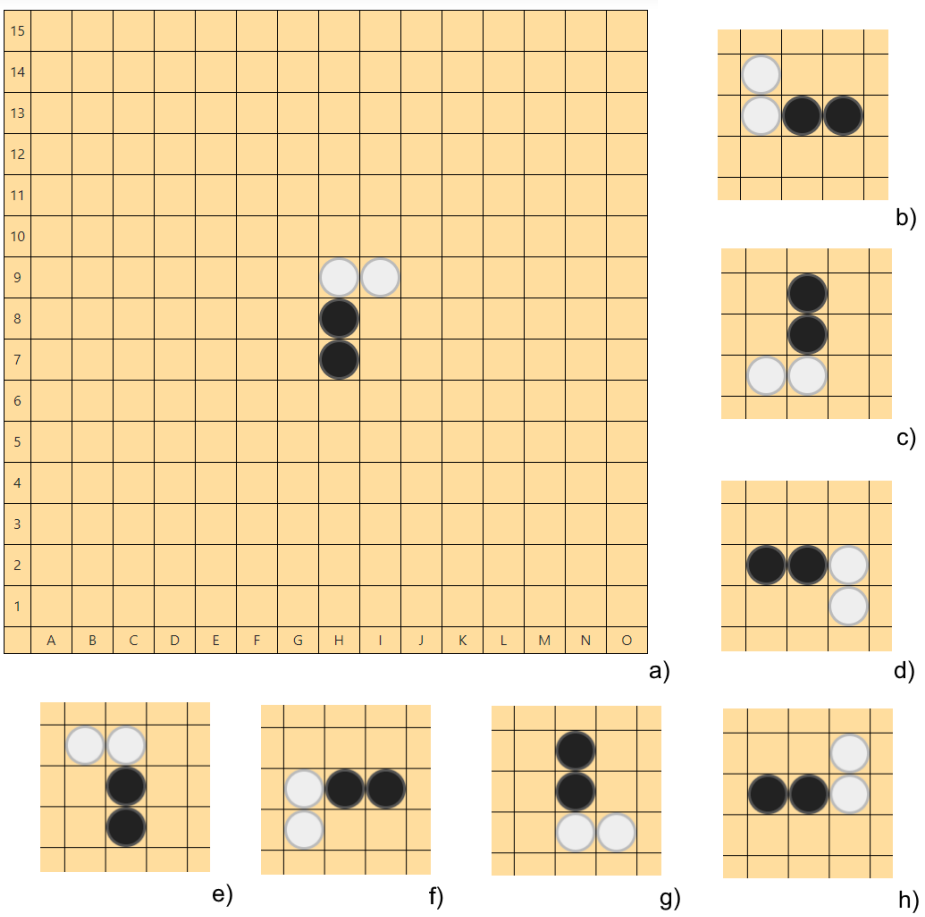
3.9. att. Apmācības datu struktūra

Pēc tam fails ar ~150 000 pozīcijām, tika apstrādāts, lai sagatavotu datus apmācībai. Kā var redzēt 3.10. attēlā iegūtu datu sadalījums nav balansēts, datu kopā ir daudz vairāk pozīciju ar novērtējumiem tuvāk 0, 0.8 un -0.8, kas var negatīvi ietekmēt apmācības procesu, tāpēc no datu kopas tika izslēgtas vairākas pozīcijas, lai izlīdzinātu novērtējumus.



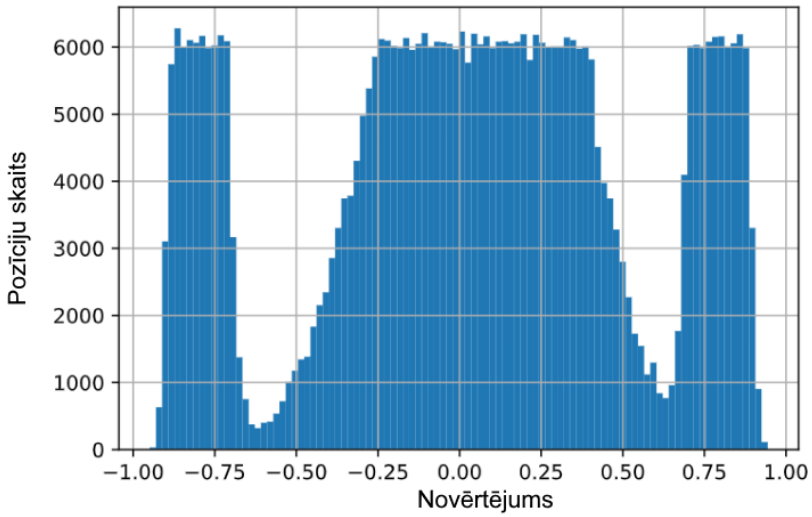
3.10. att. Datu sadalījums pēc novērtējuma.

Kad tiek veikta konvolūciju neironu tīkla apmācība uz attēliem, bieži vien datu kopu palielina, transformējot attēlus. Transformācija var iekļaut sevī tādas darbības kā attēla rotācija, atspoguļošana, tālummaiņa, nobīde un citas. Transformācijas rezultātā attēla saturs paliek viegli atpazīstams cilvēkam, un datu iezīme netiek mainīta (Chollet, 2017). Šo pieeju var arī izmantot Gomoku pozīciju datu kopai, jo katrai pozīcijai eksistē vairākas vienlīdzīgas pozīcijas. Kā paradīts attēlā 3.11. attēlā, vienu un to pašu pozīciju var reprezentēt astoņos dažādos veidos, pagriežot un atspoguļojot to. Šādā veidā pozīciju datu kopu tika palielināta astoņās reizēs.



3.11. att. Simetriskas pozīcijas, kuras tika iegūtas ar transformācijām: a) sākuma pozīcija; b) pēc pagriešanas pa 90°; c) pēc pagriešanas pa 180°; d) pēc pagriešanas pa 270°; e) pēc vertikālās atspoguļošanas; f) pēc vertikālās atspoguļošanas un pagriešanas pa 90°; g) pēc vertikālās atspoguļošanas un pagriešanas pa 180°; h) pēc vertikālās atspoguļošanas un pagriešanas pa 270°

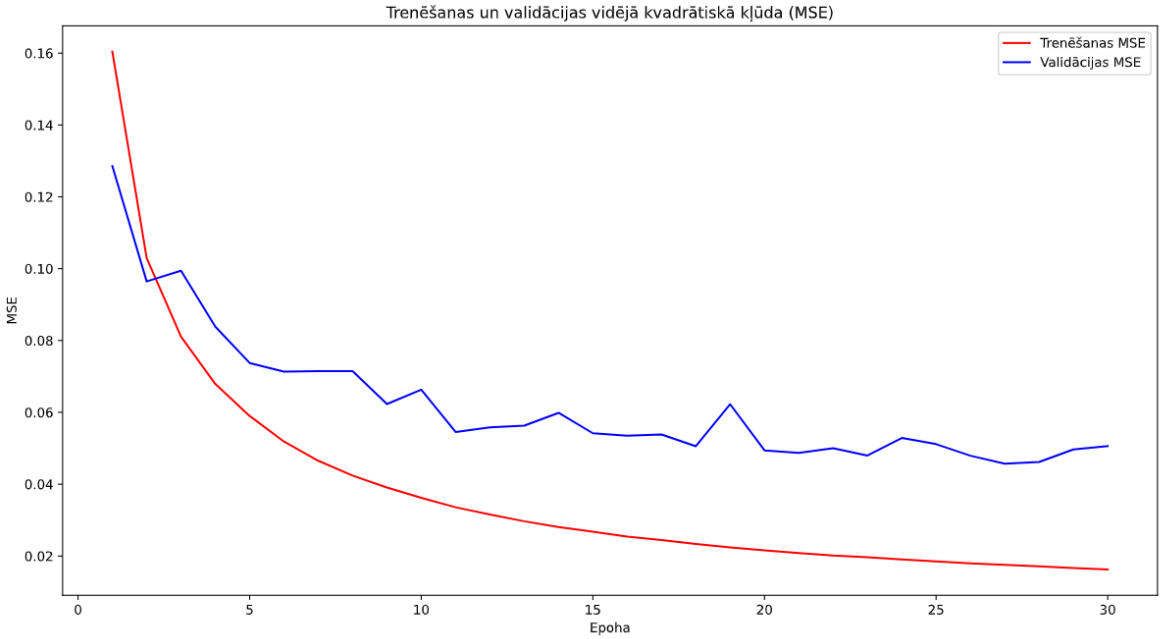
Pēc visām manipulācijām, trenēšanas datu kopas apjoms sasniedza 340 000 pozīcijas, bet validācijas datu kopā bija 84 000 pozīcijas. Rezultējošu apmācības datu sadalījumu var redzēt 3.12. attēlā. Dēļ MCTS algoritma specifikas, noteiktajos novērtējuma intervālos vienalga pārstāv datu trūkums, kas var negatīvi ietekmēt mācīšanas procesu, diemžēl šo problēmu pilnīgi neizdevās novērst ar biežāk sastopamu pozīciju izslēgšanu.



3.12. Apmācības datu kopas sadalījums pēc novērtējuma

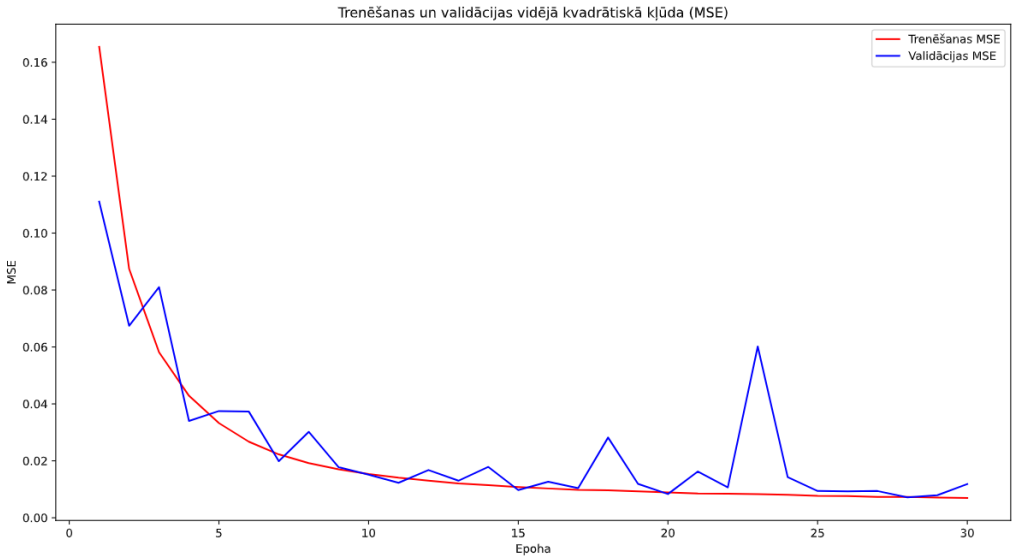
Apstrādātie dati tika sadalīti uz trenēšanas datiem un validācijas datiem proporcija 4:1. Neironu tikla mācīšanās ilgums - 30 epohas.

1. neironu tīkla variantā apmācības beigās vidējā kvadrātiskā kļūda (MSE) uz validēšanas datiem sastādīja 0.0506, bet uz trenēšanas datiem 0.0163. Apmācības grafikā 3.13. attēlā var redzēt, ka pēc 20 epohām jau sākas pārmērīga pielāgošana (overfitting), jo samazinās tikai trenēšanas kļūda, bet validācijas kļūda paliek vienā līmenī, tāpēc nav vērts turpināt apmācību vairāk par 30 epohām.



3.13. att. Vidējā kvadrātiskā kļūda atkarībā no epohas 1. neironu tīkla variantā.

2. neironu tīkla variantā apmācības beigās MSE uz validēšanas datiem sastādīja 0.0118, bet uz trenēšanas datiem 0.007, kas ir daudz mazāk nekā 1. variantā un liecina par labākiem rezultātiem. Apskatot apmācības grafiku 3.14. attēlā var redzēt, ka validācijas un trenēšanas kļūdu grafiki saplūst kopā, tas var būt saistīts ar to, ka pozīciju datu kopā ir daudz līdzīgu pozīciju, jo datu kopas ģenerācijas procesā no katras spēles tika paņemtas vairākas pozīcijas. Viens no iespējamiem risinājumiem – ģenerēt apmācības un trenēšanas datu kopas atsevišķi, lai starp datu kopām būtu lielāka atšķirība.



3.14. att. Vidējā kvadrātiskā kļūda atkarībā no epohas 2. neironu tīkla variantā.

3.4. Neironu tīkla integrācija pārmeklēšanas algoritmā

Tā kā MCTS algoritms ir izveidots, izmantojot C# valodu, bet neironu tīkls tika veidots un apmācīts Python vidē, to bija nepieciešams eksportēt. .NET ietvars, kurš tika izmantots kopā ar C#, piedāvā bibliotēku ML.NET, kura ļauj izmantot neironu tīklus kopa ar C# programmu. Lai importēt neironu tīkla risinājumu, ML.NET piedāvā izmantot universālu formātu ONNX, tāpēc pēc apmācīšanas neironu tikls tika saglabāts failā ONNX formātā, kurš pēc tam tiek izmantots C# programmas darbības gaitā.

Lai integrētu neironu tīklu pārmeklēšanas algoritmā, tika izmantota pieeja, aprakstīta 1.2.5. nodaļā. Esoša klase, kura realizē Monte Karlo pārmeklēšanu, tika mantots, un metode, kura ir atbildīga par pozīciju novērtēšanu, tika pārdefinēta, lai izmantotu ML.NET ietvaru un iepriekš apmācītu un saglabātu neironu tīklu ONNX formātā. Tādā veidā pats MCTS algoritms palīka neaiztikts, un turpināja funkcionēt, kā bija aprakstīts iepriekš, ar vienīgu atšķirību –

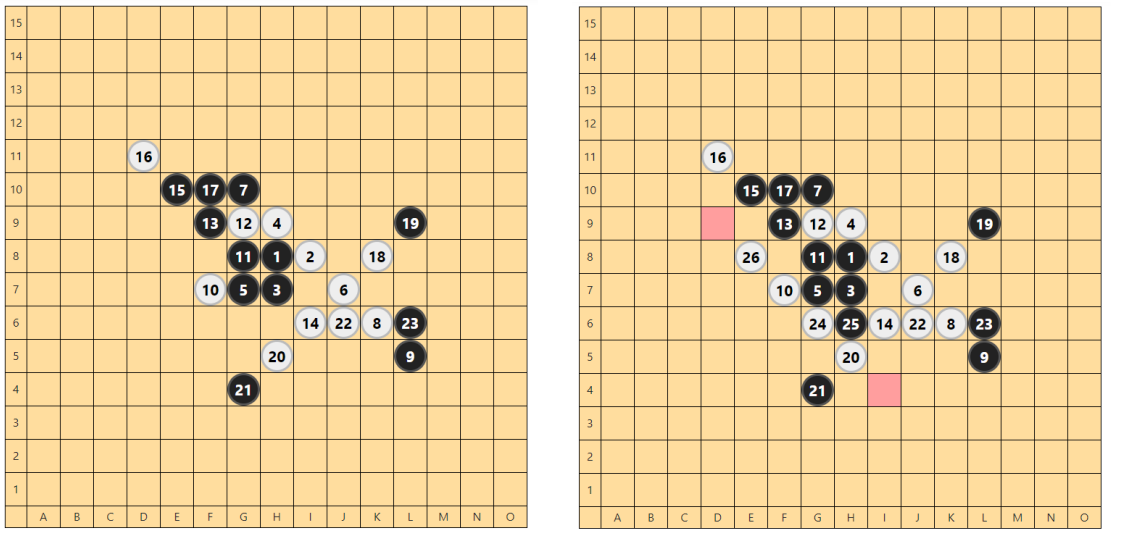
simulēšanas posma dziļums bija ierobežots ar 3 gājieniem, pēc kuriem iegūta pozīcija uzreiz tika novērtēta.

3.5. Risinājuma pārbaude

3.5.1. Monte Karlo pārmeklēšana

Analizējot Monte Karlo pārmeklēšanas algoritma novērtējumus, var secināt, ka algoritms spēj atrast uzvaru nesošus gājienus pietiekami efektīvi. Piemēram, pozīcijā 3.15.a attēlā MCTS algoritmam izdevās identificēt labāko gājienu G6, kurš ļauj baltam spēlētajam uzvārēt pēc 5 gājieniem, kā paradīts attēlā 3.15.b. Kā var redzēt attēlā 3.15.c, algoritms no 15000 iterācijām 7000 reizes apmeklēja tieši gājienu G6, kas sakrīt ar sagaidāmo rezultātu, jo MCTS algoritmam vairāk jāapmeklē perspektīvi gājieni.

Analizējot novērtējumus attēlā 3.15.c var redzēt vienu no Monte Karlo algoritma trūkumiem: kaut vai pēc gājiena G6 eksistē gājienu secība, kura ļauj baltajam spēlētajam uzvārēt ar 100% varbūtību, gājiena G6 novērtējums ir tikai 0,1392 no iespējama intervāla [-1;1], kas ir tuvāk nullei (novērtējums vienlīdzīgai pozīcijai), nekā vieniniekam (novērtējums pozīcijai, kura ļauj baltajiem uzvārēt). Tas ir saistīts ar to, kā MCTS algoritms pēc gājiena G6 pārmeklē visus iespējamus gājienus, un lielāka daļa no tām neļauj uzvarēt, kas samazina gājiena G6 novērtējumu.



C10	C11	C12	D9	D10	D12	E6	E7	E8	E9	E11	E12	F3	F4	F5	F6	F8
111	134	103	214	225	159	85	225	175	159	177	87	116	138	134	223	89
-0,2252	-0,194	-0,2427	-0,1121	-0,1022	-0,1572	-0,2941	-0,1022	-0,1429	-0,1572	-0,1412	-0,2874	-0,2241	-0,1884	-0,194	-0,1031	-0,2809
F11	G3	G5	G6	G11	H3	H4	H6	H10	H11	I4	I5	I7	I9	I10	J5	J8
91	74	80	7042	179	218	210	275	288	122	248	210	130	165	202	118	120
-0,2747	-0,3243	-0,3	0,1392	-0,1397	-0,1101	-0,1143	-0,0764	-0,0694	-0,2131	-0,0887	-0,1143	-0,2	-0,1515	-0,1188	-0,2203	-0,2167
J9	K4	K5	K7	K9	K10	L4	L7	L8	L10	M4	M5	M6	M7	M8	M9	M10
113	171	80	431	122	220	202	220	153	246	109	155	101	91	95	113	68
-0,2212	-0,1462	-0,3	-0,0209	-0,2131	-0,1091	-0,1188	-0,1091	-0,1634	-0,0894	-0,2294	-0,1613	-0,2475	-0,2747	-0,2632	-0,2212	-0,3529

c)

3.15. att. Spēles piemērs: a) Pozīcija pēc 23 gājiena, balta spēlētāja gājiens; b) Pozīcija pēc 26 gājiena, melna spēlētāja gājiens, baltie uztaisīja atvērto četrinieku, un melnie zaudē pēc jebkura gājiena, jo baltiem ir divi lauciņi, kuri ļauj izveidot piecinieku (atzīmēti ar sarkanu); c) Pozīcijas a) gājienu novērtējumi (no augšas uz lēju: gājiens, apmeklējumu skaits, novērtējums), ar zaļo atzīmēts labākais gājiens pēc novērtējuma un apmeklējumu skaita

3.5.2. Salīdzinājums ar integrēto risinājumu

Risinājuma pārbaudēs daļā izstrādātais Monte Karlo pārmeklēšanā un neironu tīklā sakņots risinājums (MCTS un NT risinājums) tika salīdzināts ar tīru Monte Karlo spēles koka pārmeklēšanas algoritmu, imitējot spēli starp tiem. Tika veikti vairāki testi, lai pārbaudītu, kā mainās algoritmu efektivitāte, atkarība no MCTS iterāciju skaita. Pēc teorijas, jo lielāks ir iterāciju skaits, jo lielāks būs apmeklētu pozīciju spēlēs koks un jo lielāka būs novērtējumu precizitāte. Katrā testa tika uzspēlētas 400 spēļu.

3.2 - 3.4 tabulās paradītas spēļu imitācijas rezultāti un statistika, izmantojot 1. neironu tīkla variantu (3 konvolūcijas slāni, kodolu izmērs 2 reiz 2). Kā var redzēt, uzvaru procents ir ļoti atkarīgs no MCTS iterāciju skaita. Palielinot iterāciju skaitu, būtiski palielinās tīra MCTS algoritma uzvaru skaits. Kad iterāciju skaits ir zem 1500, tad MCTS un NT risinājuma uzvaru procents ir lielāks par 65%, bet ja iterāciju skaits ir virs 3000 tad MCTS un NT risinājums jau

zaudē vairāk nekā 65% no visām spēlēm, pie 5000 iterācijām neironu tīklu risinājums uzvarēja tikai 22% spēļu.

Skatoties detalizētāk, arī ir redzams, ka, ja risinājums spēlē kā pirmais spēlētājs (veic pirmo gājienu), tad risinājuma uzvaru procents vienmēr ir lielāks, neatkarīgi no algoritma vai iterāciju skaita, tas var būt saistīts ar to, kā spēlē Gomoku pirmajam spēlētājam vienmēr ir priekšroka.

3.2. tabula. Spēļu apkopojums starp MCTS algoritmu un neironu tiklā sakņotu risinājumu(1. variants). MCTS iterāciju skaits – 1500.

	Kopā			Kā pirmais spēlētājs			Kā otrais spēlētājs		
	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents
MCTS un NT	400	258	64,50%	200	145	72,50%	200	113	56,50%
MCTS	400	142	35,50%	200	87	43,50%	200	55	27,50%

3.3. tabula. Spēļu apkopojums starp MCTS algoritmu un neironu tiklā sakņotu risinājumu (1. variants). MCTS iterāciju skaits – 3000.

	Kopā			Kā pirmais spēlētājs			Kā otrais spēlētājs		
	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents
MCTS un NT	400	178	44,50%	200	112	56,00%	200	66	33,00%
MCTS	400	222	55,50%	200	134	67,00%	200	88	44,00%

3.4. tabula. Spēļu apkopojums starp MCTS algoritmu un neironu tiklā sakņotu risinājumu (1. variants). MCTS iterāciju skaits – 5000.

	Kopā			Kā pirmais spēlētājs			Kā otrais spēlētājs		
	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents
MCTS un NT	400	91	22,75%	200	57	28,50%	200	34	17,00%
MCTS	400	309	77,25%	200	166	83,00%	200	143	71,50%

Savukārt ja izmantotu neironu tīkla 2. variantu rezultāti būtiski mainās. Kā var redzēt 3.5 - 3.7 tabulās MCTS un NT risinājums spēj uzvarēt 80% spēlēs, kad pārmeklēšanas iterāciju skaits ir 3000, un 54% spēlēs, kad iterāciju skaits ir 9000. Šis rezultāts ir daudz labāks, salīdzinot ar 1. neironu tīkla variantu, tomēr grafikā 3.16 attēlā var novērot tendenci: jo lielāks

ir pārmeklēšanas iterāciju skaits, jo mazāks ir uzvaru skaits MCTS un NT risinājumam, kas liecina par to, ka tīrais MCTS risinājums spēlēs labāk pie lielākā iterāciju skaitā.

3.5. tabula. Spēļu apkopojums starp MCTS algoritmu un neironu tiklā sakņotu risinājumu (2. variants). MCTS iterāciju skaits – 3000.

	Kopā			Kā pirmais spēlētājs			Kā otrais spēlētājs		
	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents
MCTS un NT	400	321	80,25%	200	168	84,00%	200	153	76,50%
MCTS	400	79	19,75%	200	47	23,50%	200	32	16,00%

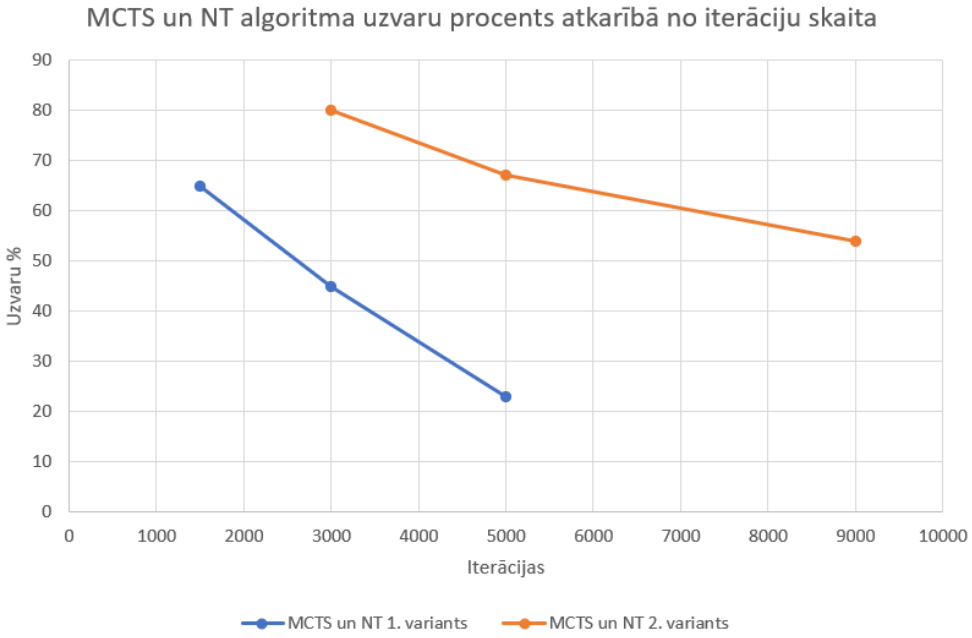
3.6. tabula. Spēļu apkopojums starp MCTS algoritmu un neironu tiklā sakņotu risinājumu (2. variants). MCTS iterāciju skaits – 5000.

	Kopā			Kā pirmais spēlētājs			Kā otrais spēlētājs		
	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents
MCTS un NT	400	267	66,75%	200	149	74,50%	200	118	59,00%
MCTS	400	133	33,25%	200	82	41,00%	200	51	25,50%

3.7. tabula. Spēļu apkopojums starp MCTS algoritmu un neironu tiklā sakņotu risinājumu (2. variants). MCTS iterāciju skaits – 9000.

	Kopā			Kā pirmais spēlētājs			Kā otrais spēlētājs		
	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents	Spēles	Uzvaru skaits	Uzvaru procents
MCTS un NT	400	216	54,00%	200	134	67,00%	200	82	41,00%
MCTS	400	184	46,00%	200	118	59,00%	200	66	33,00%

Ņemot vērā to, ka MCTS un NT risinājums novērtē pozīcijas labāk nekā tīrais MCTS risinājums, kad abu algoritmu iterāciju skaits ir zem 9000, var secināt, ka MCTS un NT risinājums var sasniegt rezultātus līdzīgus tīram MCTS algoritma pie mazāka iterāciju skaitā, kas nozīmē, ka neironu tīkla izmantošana MCTS algoritmā ļauj samazināt nepieciešamo iterāciju skaitu.



3.16. att. MCTS un NT algoritma uzvaru procents atkarībā no iterāciju skaita

SECINĀJUMI

Bakalaurā darbā tika izpētīti tādi spēlēs koku pārmeklēšanas algoritmi kā Minimax, Alfa-beta, Monte Karlo; tika izpētīti neironu tiklu darbības principi, arhitektūras, apmācības pieejas, un citi ar dziļās mašīnmācīšanās jomu saistīti temati. Tika izdarīti secinājumi par piemērotāka pārmeklēšanas algoritma izvēli.

Praktiskajā daļā tika implementēts viens no pārmeklēšanas algoritmiem: Monte Karlo pārmeklēšana (MCTS). Tika izstrādāts un apmācīts neironu tīkls, izmantojot pārraudzītas mācīšanās pieeju. Pārraudzītas mācīšanas īstenošanai tika uzģenerēta nepieciešama pozīciju datu kopa, kur pozīciju iezīmes (novērtējumi) bija iegūti izmantojot implementētu Monte Karlo pārmeklēšanas algoritmu. Apmācītais neironu tīkls tika integrēts MCTS algoritmā, un salīdzināts ar nemodificēto MCTS algoritmu risinājumā pārbaudēs daļā.

Darba autora secinājumi par pētījumu:

- Monte Karlo pārmeklēšana ir viens no efektīvākajiem pārmeklēšanas algoritmiem spēļu kokiem ar lielu zarošanas pakāpi.
- Neironu tīkli daudzdimensionālu datu apstrādei visbiežāk balstās uz konvolūciju slāņiem, jo tie ļauj izteikt lokālas iezīmes, piemēram, noteiktas akmeņu struktūras spēlē Gomoku.
- Neironu tikla efektivitāte ir ļoti atkarīga no tīkla arhitektūras un no izmantotiem datiem pārraudzītās mācīšanās procesā.
- Ģenētisko algoritmu izmantošana neironu tīkla apmācībai ir daudz lēnāka, salīdzinot ar pārraudzītās mācīšanas pieeju.
- Eksistē plašs literatūras un zinātnisko darbu klāsts par saistītiem ar pārmeklēšanas algoritmiem un dziļās mašīnmācīšanās tematiem.

Risinājumā pārbaudēs daļā tika veikts salīdzinājums, imitējot spēlēs starp abiem algoritmiem. Salīdzinājuma rezultāti tika apkopoti, analizēti, un tika izdarīti sekojoši secinājumi:

- Monte Karlo pārmeklēšanas algoritms funkcionē, kā bija paredzēts.
- Neironu tīkla integrācija MCTS algoritmā, izmantojot "novērtēšanas ierobežojuma" pieeju, var uzlabot MCTS algoritma efektivitāti pie noteiktiem apstākļiem.

Kaut vai šajā darbā izstrādātais pārmeklēšanā un mākslīgajos neironu tīklos sakņots risinājums radā labus rezultātus, darba autors piedāvā sekojošus integrēta risinājuma uzlabojumus, kuri varētu uzlabot iegūtos rezultātus:

- Izmantot lielāko pozīciju datu kopu pārraudzītās mācīšanas īstenošanai.
- Izmantot citu algoritmu pozīciju datu kopas izveidei, kurš varētu dot precīzākus novērtējumus.
- Trenēšanas un validācijas datu kopas veidot neatkarīgi viens no otra.
- Veikt izmaiņas neironu tīklā arhitektūrā, kuras varētu samazināt apmācības kļūdu.
- Izmantot citu neironu tīkla integrācijas pieeju Monte Karlo algoritmā.

Šajā darbā visgrūtākais bija izstrādāt pārmeklēšanas algoritmu, kurš varētu strādāt pietiekami ātri - ļoti daudz laika aizņēma programmatūras optimizēšana un atkļūdošana. Neironu tīkla veidošanas posmā galvenais izaicinājums bija iezīmētu datu iegūšana.

LITERATŪRA

Agostinelli F., Hocquet G., Singh S. & Baldi P. “From Reinforcement Learning to Deep Reinforcement Learning: An Overview,” in *Braverman Readings in Machine Learning*, L. Rozonoer, B. Mirkin, I. Muchnik, Eds. Cham, Switzerland: Springer International Publishing, 2018, pp. 298–328. doi: /10.1007/978-3-319-99492-5_13

Bailer-Jones C. A., Gupta R., & Singh H. P. “An introduction to artificial neural networks,” 2001, 18 p. arXiv: astro-ph/0102224

Browne C. B. u.c., "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012, doi: 10.1109/TCIAIG.2012.2186810.

Chellapilla K. & Fogel D. B. “Evolving Neural Networks to Play Checkers Without Relying on Expert Knowledge,” *IEEE Transactions On Neural Networks*, vol. 10, no. 6, pp. 1382–1391, Nov. 1999. doi: 10.1109/72.809083

Chollet F. *Deep Learning with Python*. New York, Unated States of America: Manning Publications, 2017. 396 p.

Diderich C.G. & Gengler M. “Minimax Game Tree Searching,” in *Encyclopedia of Optimization*, C. Floudas, P. Pardalos, Eds. Boston, United States of America: Springer. 2008. doi: /10.1007/978-0-387-74759-0_370

Elnaggar A., Gadallah M., Mostafa M. & Eldeeb H. “A Comparative Study of Game Tree Searching Methods,” *International Journal of Advanced Computer Science and Applications*, vol. 5, pp. 68–77, Jun. 2014. doi: 10.14569/IJACSA.2014.050510

Fogel D. B. "An introduction to simulated evolutionary optimization," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 3–14, Jan. 1994. doi: 10.1109/72.265956.

Goodfellow I., Bengio Y. & Courville A. *Deep Learning.* MIT press. 2016. 773 p.

Jain A. K., Mao J. & Mohiuddin K. M. “Artificial neural networks: a tutorial,” *Computer*, vol. 29, no. 3) pp. 31–44, March 1996, doi: 10.1109/2.485891.

Kaindl, H. “Tree Searching Algorithms,” in *Computers, Chess, and Cognition*, T. A. Marsland, J. Schaeffer, Eds. New York, United States of America: Springer New York, 1990, pp. 133–158. doi:10.1007/978-1-4613-9080-0_8

Karim R. *Counting No. of Parameters in Deep Learning Models by Hand*. 2019 [cited – April 23. 2021]. Available at https://towardsdatascience.com/counting-no-of-parameters-in-deep-learning-models-by-hand-8f1716241889

LeCun Y., Bengio Y., & Hinton G. "Deep learning" in *Nature*, no. 7553 (vol. 521), pp. 436–444. 2015. doi:10.1038/nature14539

LeCun Y., Bottou L., Y. Bengio & P. Haffner. "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324, doi: 10.1109/5.726791.

Lee H., Grosse R., Ranganath R., & Ng A. Y. "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*, 2009, pp. 609–616. doi:10.1145/1553374.1553453

Lippmann R., "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol. 4, no. 2, pp. 4–22, Apr. 1987. doi: 10.1109/MASSP.1987.1165576.

Lui M. *General Game-Playing With Monte Carlo Tree Search*. 2017-2018 [cited – April 16, 2021]. Available at https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238

Mannor S., "k-Armed Bandit," in *Encyclopedia of Machine Learning and Data Mining*, C. Sammut, G.I. Webb, Eds. Boston, United States of America: Springer US, 2017, pp. 687–690. doi: 10.1007/978-1-4899-7687-1_424

Nielsen M. A. *Neural Networks and Deep Learning*. Determination press. 2015.

Nosovsky A. & Sokolsky A. *Renju for beginners*, 1999, 30 p.

Salimans T., Ho J., Chen X., Sidor S. & Sutskever I. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," 2017, 13 p. arXiv:1703.03864

Silver D. & Hassabis D. AlphaGo: Mastering the ancient game of Go with Machine Learning. 2016. [cited – April 23, 2021]. Available at https://ai.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html

Takada K., Iizuka H. & Yamamoto M. "Reinforcement Learning for Creating Evaluation Function using Convolutional Neural Network in Hex," in *Proceedings of the 2017 Conference on Technologies*

and Applications of Artificial Intelligence (TAAI), 2017, pp. 196–201, doi: 10.1109/TAAI.2017.16.

Tzeng CH. “Games and Minimax Values,” in *A Theory of Heuristic Information in Game-Tree Search. Symbolic Computation (Artificial Intelligence)*. Berlin, Germany: Springer Berlin Heidelberg, 1988, pp. 7–17. doi: 10.1007/978-3-642-61368-5_2

Winands, M. H. M. “Monte-Carlo Tree Search in Board Games,” in *Handbook of Digital Games and Entertainment Technologies*. Singapore, Singapore: Springer Singapore, 2016, pp. 47–76. doi:10.1007/978-981-4560-50-4_27

Yamashita R., Nishio M. u.c. “Convolutional neural networks: an overview and application in radiology,” *Insights into Imaging*, vol 9, pp 611–629, Aug. 2018. doi: 10.1007/s13244-018-0639-9

Zhang R. “Convolutional and Recurrent Neural Network for Gomoku,” 2016, 6 p.

Николенко С., Кадурин А. & Архангельская Е. *Глубокое обучение: Погружение в мир нейронных сетей*. Санкт-Петербург, Россия: Питер, 2018, 476 стр.

PIELIKUMI

1. pielikums. Metodes MCTSRun pirmkods

```
private float MCTSRun(GameTreeNode node)
{
  float? evaluation = null;
  GameResult? simulationResult = null;
  GameState playoutGameState = null;
  if (node.GameState is null)
    throw new ArgumentNullException(nameof(node.GameState));

  var isGameOverResult = node.GameState.IsGameOver();
  if (isGameOverResult != null)
  {
    //Spēle beigusies (uzvarēja viens no spēlētajiem)
    simulationResult = isGameOverResult;
  }
  else if (node.Evals.Count == 0)
  {
    //Atrasta iepriekš neizmeklēta virsotne
    //(3. Posms: Simulēšana)
    playoutGameState = MCTSPlayout(node.GameState, _playoutDepth);
    simulationResult = playoutGameState.IsGameOver();
  }
  else
  {
    if (node.Children is null)
    {
      //Virsotnēs pēcteči nav inicializēti, inicializējam tos
      node.Children = ExpandNode(node, false);
    }
    if (!node.Children.Any())
    {
      //Pēc inicializācijas pēcteču skaits ir 0,
      //kas nozīme, ka no šīs virsotnēs vairs nav gājienu (neizšķirts)
      simulationResult = GameResult.Tie;
    }
    else
    {
      //Turpinām pārmeklēšanu, jāizvēlas virsotne ar lielāku UCB
      //( 1. Posms: Izvēle)
      double bestUCB = double.MinValue;
      KeyValuePair<PlayerMove,GameTreeNode>? bestChild = null;
      foreach (var child in node.Children)
      {
        var ucb = UCB(EVAL(child.Value), node.Evals.Count(), child.Value.Evals
.Count());
        if (ucb >= bestUCB)
        {
          bestUCB = ucb;
```

```
            bestChild = child;
          }
          if (bestUCB == double.MaxValue)
            break;
        }
        // Virsotnēs pozīcija nav inicializēta, inicializējam to;
        // (2. Posms: Paplašināšana)
        if (bestChild.Value.Value.GameState is null)
        {
          bestChild.Value.Value.GameState = node.GameState.MakeMove(bestChild.Va
lue.Key);
        }

        // Rekursīvi izsaucam šo pašu metodi.
        // (4. Posms: Atpakaļizplatīšana)
        evaluation = MCTSRun(bestChild.Value.Value);
      }

    }

    if (evaluation == null)
    {
      if (simulationResult != null)
      {
        evaluation = simulationResult.Value switch
        {
          GameResult.FirstPlayerWon => 1,
          GameResult.SecondPlayerWon => -1,
          GameResult.Tie => 0,
          _ => throw new NotImplementedException()
        };
      }
      else if(playoutGameState != null)
      {
        evaluation = EvaluateState(playoutGameState);
      } else
      {
        throw new Exception("Something went wrong, unable to obtain evaluation")
;
      }
    }

    if (node.GameState.PlayerTurn == PlayerColor.Second)
    {
      node.Evals.Add(evaluation.Value);

    }
    else if (node.GameState.PlayerTurn == PlayerColor.First)
```

```
    {
      node.Evals.Add(-evaluation.Value);
    }
    else
    {
      throw new NotImplementedException();
    }
    return evaluation.Value;
  }
```

```
private GameState MCTSPlayout(GameState sourceGameState, int maxDepth)
{
  //Kopējam stāvokli, lai to varētu modificēt un nesabojāt esošu spēles koku.
  var gameState = sourceGameState.Copy();

  // Cikliski simulējam gājienus, kamēr nav sasniegts maksimālais dziļums,
  // vai spēle nav beigusies.
  for (var depth = 0; depth < maxDepth; ++depth)
  {
    var gameOver = gameState.IsGameOver();
    if (gameOver != null)
    {
      //Spēle ir beigusies, beidzam simulēšanas posmu.
      break;
    }

    //Veicam nejaušu gājienu.
    var moves = GetMoves(gameState).ToList();
    var randomMove = moves.ElementAt(Random.Next(moves.Count()));
    gameState.MakeMoveInPlace(randomMove);
  }
  //Atgriežam sasniegto stāvokli
  return gameState;
}
```

3. pielikums. Neironu tīkla definējums (2. variants)

```
input_shape = (15,15,1)
input_len = input_shape[0] * input_shape[1] * input_shape[2]

# Ieejas slānis A
inputA = keras.Input(shape=input_len, name="input_board")
# Ieejas slānis B
inputB = keras.Input(shape=1, name="input_turn")
# Transformācijas slānis 1D -> 2D
a = layers.Reshape(input_shape)(inputA)
# 1.Konvolūciju slānis
a = layers.Conv2D(64, kernel_size=(3, 3), kernel_initializer=keras.initializer
s.HeNormal())(a)
# Aktivizācijas slānis
a = layers.LeakyReLU(alpha=0.2)(a)
# 2. Konvolūciju slānis
a = layers.Conv2D(64, kernel_size=(3, 3), kernel_initializer=keras.initializer
s.HeNormal())(a)
# Aktivizācijas slānis
a = layers.LeakyReLU(alpha=0.2)(a)
# 3. Konvolūciju slānis
a = layers.Conv2D(64, kernel_size=(3, 3), kernel_initializer=keras.initializer
s.HeNormal())(a)
# Aktivizācijas slānis
a = layers.LeakyReLU(alpha=0.2)(a)
# 4. Konvolūciju slānis
a = layers.Conv2D(64, kernel_size=(3, 3), kernel_initializer=keras.initializer
s.HeNormal())(a)
# Aktivizācijas slānis
a = layers.LeakyReLU(alpha=0.2)(a)
# Transformācijas slānis 2D -> 1D
a = layers.Flatten()(a)
# 1. Pilnsaistes slānis
a = layers.Dense(64, kernel_initializer=keras.initializers.HeNormal())(a)
# Aktivizācijas slānis
a = layers.LeakyReLU(alpha=0.2)(a)
# Konkatenācijas slānis
a = layers.Concatenate()([a, inputB])
# 2. Pilnsaistes slānis
a = layers.Dense(32, kernel_initializer=keras.initializers.HeNormal())(a)
# Aktivizācijas slānis
a = layers.LeakyReLU(alpha=0.2)(a)
# Izejas slānis ar aktivizācijas funkciju ‘tanh’
a = layers.Dense(1, activation='tanh', name="output")(a)

model = keras.Model(inputs=[inputA,inputB], outputs=a)
```