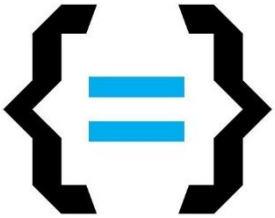


Fakulta Informatiky a Managementu
Univerzita Hradec Králové

Umělá inteligence pro deskové hry

Vypracoval: Jiayuan Hu
Studijní obor: Aplikovaná informatika
Vedoucí práce: Ing. Tomáš Nacházel, Ph.D.

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

vlastnoruční podpis

V Hradci Králové dne 25.4.2024

Poděkování:

Rád bych vyjádřil svou upřímnou vděčnost panu Ing. Tomáši Nacházelovi, Ph.D., za jeho neocenitelnou podporu a odborné vedení v průběhu mého výzkumu a psaní této práce. Jeho odborné znalosti a ochota poskytovat cenné rady byly pro mě neocenitelné a významně přispěly k úspěšnému dokončení této práce.

Abstrakt

Tato práce se zaměřuje na vývoj umělé inteligence v tradičních deskových hrách, zejména na hru Go. Začíná zkoumáním historie a vývoje umělé inteligence v tradičních deskových hrách, jako jsou piškvorky a šachy, s důrazem na Go, které je považováno za jednu z nejsložitějších her. Práce dále analyzuje pravidla Go, neuronovou síť AlphaGo a algoritmus Monte Carlo Tree Search (MCTS), které jsou klíčovými prvky v pokroku umělé inteligence v této hře. Hlavním cílem je vytvořit program s umělou inteligencí, který bude schopen hrát Go. Součástí práce je také studium metod trénování neuronových sítí. Výsledkem této práce je implementace a vyhodnocení programu, který využívá moderní techniky umělé inteligence a je schopný hrát Go proti lidským hráčům.

Abstract

Title: Artificial intelligence for board games

This bachelor thesis focuses on the development of artificial intelligence in traditional board games, in particular the game Go. It begins by examining the history and evolution of artificial intelligence in traditional board games such as ping-pong and chess, with an emphasis on Go, which is considered one of the most complex games. The thesis also analyses the rules of Go, the AlphaGo neural network and the Monte Carlo Tree Search (MCTS) algorithm, which are key elements in the progress of artificial intelligence in this game. The main goal is to create an AI program that will be able to play Go. The work also includes the study of neural network training methods. The result of this work is the implementation and evaluation of a program that uses modern artificial intelligence techniques and is capable of playing Go against human players.

Klíčova slova: desková hra, Go, hluboké učení s posilováním, umělá inteligence, neuronová síť

Key words: board game, Go, deep reinforcement learning, artificial intelligence, neural network

Obsah

Seznám obrázku

1 Úvod

Umělá inteligence není pouze výzkumnou oblastí, která vytváří nové technologie; je to i disciplína, která nám pomáhá lépe porozumět lidské inteligenci a rozhodovacím procesům. V oblasti deskových her, jako je Go, umělá inteligence představuje vynikající prostředek pro zkoumání strategických a rozhodovacích mechanismů, které jsou klíčové pro úspěch v těchto hrách.

Motivace pro tuto práci vychází z triumfu programu AlphaGo nad nejlepšími hráči hry Go v roce 2016. Tento převratný okamžik v oblasti umělé inteligence ukázal, že technologie mají schopnost dosáhnout úrovně nad lidskými schopnostmi.

Tato práce se zaměřuje na představení historie deskových her a umělé inteligence uplatněné na nich, s důrazem na průlomové události a technologické pokroky, které vedly k současnému stavu této oblasti. Dále se bude zabývat analýzou algoritmů a pravidel hry Go, a jejich implementací v softwaru. Tímto způsobem se nejen prozkoumává možnost vytvoření sofistikovaných herních prostředí, ale zároveň se i podporuje další vývoj umělé inteligence a jejího využití v praktických aplikacích.

2 Cíl a metodika

Tato práce se zaměřuje především na nejvýznamnější tradiční deskové hry a jejich historii v oblasti programování. Hlavní částí bude studie hry Go a detailní představení jejích hlavních pravidel.

Cílem bude vytvoření softwaru simulujícího a zjednodušujícího hru Go podobně jako AlphaGo. Tento software umožní hrát jak hru mezi lidmi, tak i mezi člověkem a umělou inteligencí. Dále bude možné hrát zápasy mezi dvěma umělými inteligencemi, což může sloužit k tréninku vlastní umělé inteligence.

Současně budou v tomto projektu využity nástroje umělé inteligence, jako je ChatGPT a DeepL, k provádění kontroly gramatiky a strojového překladu odborných termínů do češtiny.

3 Vývoj a historie umělé inteligence v oblasti tradiční deskové hry

Tradiční deskové hry jako šachy, japonský šogi, čínské šachy a Go atd. jsou populární po celém světě a jsou symbolem inteligence. V dávném východním světě byl Go považován za filozofickou hru, mistři Go byli uctíváni a někteří z nich dokonce mohli být hosty králů a císařů. Podobné situace se vyskytly i v Blízkém východě a Evropě.

V moderní době, s vynálezem počítače, se vědci v oblasti počítačových věd snažili využít nové technologie k řešení mnoha problémů, a deskové hry se staly jedním z jejich cílů. V roce 1997 americká firma IBM představila superpočítač Deep Blue, a v roce 2015 DeepMind vyvinul program AlphaGoZero, které postupně porazily mistra světa v šachu a v Go, což oznamuje úspěch umělé inteligence v další oblasti.

3.1 Piškvorky (Tic-tac-toe)

Hra Tic-tac-toe, známá také jako Piškvorky, je stará a tradiční strategická hra, ve které dva hráči střídavě umisťují své kameny na hrací plochu. Cílem je zabránit soupeři v sestavení určitého počtu svých kamenů v řadě a zároveň sami spojit své kameny. Tato hra má dlouhou historii a je zaznamenána v archeologických dokumentech po celém světě. První nalezeny záznamy o ní byly v deltě Žluté řeky dnešní Číny přibližně 2000 let před naším letopočtem, avšak zmínky o hře byly také nezávisle objeveny v Egyptě i v Římě, a dokonce i ve starověké Americe.[1.]

Díky své jednoduchosti pravidla se tato hra stala cílem a prototypem raných elektronických her. V 50. letech minulého století Josef Kates vynalezl počítač s názvem Bertie the brain, který byl roku 1950 představen na výstavě Canadian National Exhibition a po výstavě byl hned demontován.[2.] Tento počítač měl výšku až 4 metry a jeho účelem bylo právě hrát hru Tic-tac-toe. Je považován za jednu z nejstarších elektronických her na světě. Hráči měli za úkol stisknout tlačítka na ovládacím panelu s devíti políčky a na obrazovce se náhle zobrazovaly jejich tahy a chvíli poté tahy počítače. Hráči také měli možnost volby obtížnosti, přičemž nejvyšší obtížnost byla nastavena na neporazitelnou.

Ačkoliv měl tento počítač výšku až 4 metry, hráči tvrdili, že jeho vítěznostní poměr, i když byl velmi vysoký na nejvyšší obtížnosti, ale nebyl stoprocentní.

Všechno se však změnilo o dva roky později, kdy společnost EDSAC (Electronic Delay Storage Automatic Calculator) vyvinula program s názvem OXO. Tento program dokázal dokonale hrát ve všech 26 830[1]možných větví herního stromu, což znamenalo, že žádný člověk ho nemohl porazit. Tento program je klasickým příkladem algoritmu minimax, který vyhodnocuje strom všech možných tahů. Problémový prostor byl dostatečně malý, takže i 70 let starý počítač, který přijímal čísla od 1 do 9 prostřednictvím otáčecího telefonního ovladače a výsledky zobrazoval na 35x16 pixelovém CRT monitoru, byl schopen tento program spustit.

3.2 Šachy

Šachy, někdy taky jsou označovány jako král všech her, patří mezi jednou z nejpopulárnějších deskových her na světě, která se hraje po celém globu. Jejich původ sahá až do starověké Indické Gutopské říše, kde byly známé pod názvem čaturanga, a postupně se rozšířily do celého světa, kde získaly obrovskou popularitu díky své strategické hloubce a možnostem.

Hra šachy se hraje na hrací desce o velikosti 8x8 polí, která je rozdělena na bílé a černé pole. Každý hráč má na začátku 16 figurek: krále, královnu, dva střelce, dva jezdce, dva věže a osm pěšců. Každá figura má své vlastní pohybové schopnosti. Cílem hry je s pomocí svých figurek srazit krále soupeře do šachu takovým způsobem, že se nedokáže bránit a nebude moct být zachráněn.

Kolem 18. století se objevil první známý automatický šachový stroj na světě nazvaný Turek. Vytvořil jej bratislavský diplomat a vynálezce Wolfgang von Kempelen. Tento šachový automat se skládá z velké skříni se šachovnicí na vrchu, a šachovými figurkami, které reagovaly na tahy lidských hráčů

[1] 9! = 362 880(v prvním tahu 9 políček, v druhém tahu 8 políček atd., ale tento výpočet zahrnuje i případ jako je jeden hráč zahraje v sérii několik kroků za sebou. Po odstranění těchto výjimek výsledek se dostává k číslu 255 168, pokud bude považovat stavy, které jsou totožné po rotaci nebo převrácení, za stejné, pak existuje pouze 26 830 možných herních větví.)[3.]

automaticky pomocí nějakého neznámého mechanismu, jenž byl skrytý v krabici pod šachovnicí. Kempelen předvedl svého Turka císařovně Marii Terezii z Habsburské monarchie, díky tomu tento zdánlivě chytrý stroj rychle zaujal celou Evropu a během několik let se podařil Turek obelstít mnoha významné šachisty, jako jsou Napoleona Bonaparta, Benjamina Franklina a významného matematika a předchůdce moderního počítání Charlese Babbage. Po až téměř 100 let byl Kempelenův podvod konečně odhalen a ukázalo se, že uvnitř stroje ve skutečnosti byl schovaný zkušený šachový mistr.

Roku 1947, jeden z prvních počítačových vědců a legenda Alan Turing napsal program pro šachovou hru. Nicméně v té době byly počítače vzácnou záležitostí, takže tento program nikdy nebyl spuštěn a Turing, který zemřel v mladém věku, neměl už další příležitost jej realizovat. V roce 1950 další průkopník v oblasti počítačů, zakladatel teorie informace Claude Shannon, představil algoritmus minimax pro dvouhráčové hry a v roce 1950 publikoval průkopnický výzkumný článek "Programming a computer for playing chess“, čímž otevřel cestu teoretickému studiu počítačového hraní šachů. [4.]

V porovnání s piškvorkami viz výše je komplexita herního stromu šachů mnohem vyšší, až na úroveň 10^123[5.]bez uvažování o zjednodušení a ořezávání. Pouhé použití algoritmu minimax pomocí výpočtu hrubou silou je téměř nemožné. Proto je naléhavě nezbytně použít některé metody pro ořezávání, a právě proto vzniká alfa-beta ořezávání. Tento algoritmus dosahuje stejného závěru jako algoritmus minimax, ale odstraní větve, které nemají vliv na konečné rozhodnutí. Metoda byla poprvé navržena Johnem McCarthy na konferenci Dartmouth v roce 1956, kde představil podobný prvotní koncept. Arthur Samuel byl prvním člověkem, který skutečně zrealizoval alfa beta ořezávání v praxi, využil ho ve hře dáma. Následně D. Knuth a Ronald W. Moore vylepšili algoritmus v roce 1975 a J.Pearl v roce 1982 prokázal jeho optimální řešení.

Během této doby bylo vytvořeno mnoho inteligentních programů pro hru šachy pomocí těchto algoritmů, ale kvůli nedostatkům algoritmu, a i nedostatku výpočetního výkonu žádný z nich nedokázal dosáhnout skutečné umělé inteligence a porazit profesionální hráče. Až v roce 1989 tým z Carnegie Mellon University vyvinul šachový počítač nazvaný Deep Thought, který se stal prvním počítačovým velmistrem v mezinárodních šachových soutěžích. Tento tým se později připojil k IBM a stal se jádrem týmu, který vyvinul počítač Deep Blue.

V květnu 1997 se v New Yorku konala série šesti partií mezi Deep Blue a Garri Kasparovem, tehdejším mistrem světa a možná nejlepším šachistou v historii. Tento zápas skončil 4:2 vítězstvím počítače Deep Blue, čímž se stal prvním počítačem, který porazil lidského mistra v mezinárodním šachu.

Deep Blue, který se utkal s Kasparovem, měl 2 operační pulty obsahující 30 procesorů (počítačů), z nichž každý obsahoval 480 vlastních šachových čipů. Tento systém byl schopen vyhledávat 2x10^8 šachových pozic za sekundu a zlepšil se i ve vyhledávací hloubce. Díky výkonnosti poskytované specializovanými čipy poskytl tento "hrubou silou" implementovaný algoritmus základ pro Deep Blue. S bohatými znalostmi šachů, databázemi her, a dostatečným testováním proti velmistrům měl konečná verze Deep Blue velmi vysokou úroveň. [6.]

Superpočítač dokázal porazit šachového velmistra především díky dvěma faktorům. Prvním faktorem je bohaté znalosti o šachu. Druhým faktorem je obrovský výpočetní výkon. I když algoritmus pro ořezávání a softwarové vyhledávání her objektivně snižují prostor pro vyhledávání, stále se jedná o řešení hrubou sílu.

Po šachách se výzkumníci zaměřili na Go. Ve výsledku šířka prohledávání u šach je přibližně 30, hloubka prohledávání je přibližně 80 a celkový prostor prohledávání je přibližně 10^50. U Go je šířka prohledávání přibližně 250, hloubka prohledávání přibližně 150 a prostor prohledávání přesahuje 10^170, což je více než počet částic ve vesmíru (10^80).[7.] Kvůli příliš velkému prohledávacímu prostoru nelze v omezeném čase dokončit celý prostor pouze pomocí hodnotící funkce a algoritmu pro ořezávání. Metoda hrubé síly používaná Deep Blue je pro Go naprosto neúčinná. Po dlouhou dobu lidé považovali Go za nepřekonatelnou překážku umělé inteligence. [8.][7.]

3.3 Go

Go, známé také jako Weiqi nebo Baduk, má původ v Číně a je to strategická desková hra pro dva hráče. Jeho historie sahá zpět přibližně do roku 2000 před naším letopočtem, čímž se stává jednou z nejstarších her na světě. Hra se hraje na hracím poli s mřížkou o velikosti 19x19, kde hráči střídavě pokládají

černé a bílé kameny na průsečíky linií s cílem získat co nejvíce území a obklíčit soupeřovy kameny.[9.]

Go se vyvinulo v Číně a neslo v sobě filozofický, vojenský a kulturní význam. S postupem času se hra rozšířila do Japonska, Koreje a dalších zemí, kde vznikly různé školy a styly hraní.

Go zdůrazňuje kombinaci dlouhodobé strategie a taktiky na menší úrovni, což oslovuje hráče i intelektuální nadšence. Má velkou komunitu hráčů v Asii a díky své složitosti a strategické povaze získává pozornost i na mezinárodní úrovni.

Rozdělujeme vývoj umělé inteligence ve hře Go do tří generace:

3.3.1 První generace

Zaměřuje se na rozpoznávání vzorů hry a používání heuristických algoritmů. Úroveň je nižší než amatérští hráči s nižším danem. První program, který dokázal zahrát celou kompletní hru Go, byl vytvořen v roce 1968 americkým vědcem Albertem L. Zobristem jako součást jeho disertační práce.

Avšak v průběhu dalších 20 let měly programy umělé inteligence problém dokonce i s vítězstvím proti začátečníkům.

V roce 1987 pán Ing Chang-ki, obchodník a velký milovník Go, zveřejnil odměnu ve výši milionu dolarů za program umělé inteligence, který by dokázal porazit profesionálního hráče v Go. Tato odměna byla rozdělena do několika úrovní, přičemž nejnižší úroveň měla cenu 100 000 nových tchajwanských dolarů, což bylo tehdy ekvivalentní 4 000 amerických dolarů.[10.] Nejnižší odměna byla udělována prvnímu programu, který by byl schopen porazit amatérského hráče se sedmnácti kameny handicapu. [2]

Až do roku 1990, profesor chemické fakulty na Sunjatsenově univerzitě Chen Zhixing strávil několik měsíců svého volného času na vytvoření programu nazvaného "Shoutan"[3]. Krátce po vytvoření Shoutanu začal jeho výkon rychle stoupat.

V roce 1995 Shoutan dokázal porazit hráče se čtrnácti a dvanácti kameny handicapu, čímž získal odměny za své úspěchy proti amatérským hráčům.

---

[2] Handicap poskytuje slabšímu hráči určitou výhodu na začátku hry

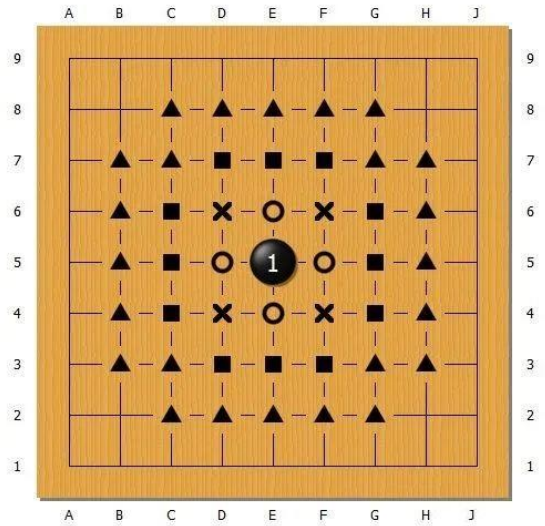
[3] což znamená komunikace mezi rukama, je to archaický pseudonym Go

Dva roky později Shoutan překonal i hranici deseti kamenů handicapu v souboji s amatérskými hráči. Tato tři vítězství přinesla Chen Zhixingovi celkem 600 000 nových tchajwanských dolarů. Zároveň mezi lety 1995 a 1998 Shoutan získal sedm vítězství v po sobě jdoucích ročnících Světového mistrovství počítačového Go, což ho vyneslo na špičku během celé dekády.

Problémy, kterým čelí umělá inteligence při hře, jsou dvojí povahy. Za prvé, je obtížné rozhodnout, kam přesně umístit kámen na hrací desku, protože možností je mnoho. Za druhé, složité je také vyhodnotit situaci na desce, tedy určit, zda se má hrát na výhru nebo na remízu.

Profesor Chen Zhixing vytvořil řešení pro tyto problémy pomocí "fukce síly". Termín "síla" je odborný termín v Go a znamená v podstatě vliv kamene na okolní prostor. Chen Zhixing tvrdil, že tento vliv lze kvantifikovat.

Například, viz obrázek1. na hrací desce se nachází černý kamen. Nejbližší čtyři body, označené kruhy, jsou ovlivněny tímto kamenem hodnotou 4. Body trochu dál, označené křížky, mají vliv 3. Body označené čtverci mají vliv 2. Body označené trojúhelníky mají vliv 1. Vzdálenější oblasti už nejsou ovlivněny.



Obr. 1 Ukázka funkce síly (Zdroj:[22.])

Samozřejmě na hrací desce je mnoho kamenů na různých místech. Vlivy všech kamenů lze sloučit dohromady, aby bylo určeno, zda daný prázdný bod na desce spadá pod vliv černých nebo bílých kamenů.

Nicméně tato "funkce síly" není univerzální. Ve hře Go existuje mnoho složitých vzorů a Shoutan byl omezený v rozpoznání širšího spektra těchto vzorů.

Zároveň, základní hodnocení situace pomocí funkce síly není vždy přesné. To omezovalo další zlepšení Shoutana.

3.3.2 Druhá generace

První generace umělé inteligence ve hře Go má některé obtíže při hodnocení situace, zejména v porovnání se šachy. Ve šachách mají figurky, jako jsou dáma nebo kůň, které jsou jasně unikátní a dosti silné, a proto je možné poměrně snadno hodnotit situaci pomocí výpočtu počtu a umístění figurek. Jednoduše řečeno, mít navíc dámu nebo koně obvykle znamená výhodnou situaci. I když je třeba zohlednit konkrétní umístění figurek, pomocí hodnoty síly figurek lze dosáhnout relativně přesného odhadu situace.

Nicméně ve hře Go mají všechny kameny stejné základní vlastnosti a jediný rozdíl mezi nimi spočívá v jejich umístění na herní desce. To vede k obtížím při hodnocení situace počítačem, protože složitost situace ve hře Go spočívá v zohledňování celkových vztahů mezi všemi kameny na desce, nikoliv pouze v jejich počtu. Použití hodnoty kamenů k jednoduchému hodnocení situace ve hře go není proveditelné, protože vliv každého kamene závisí na jeho roli a umístění v celkové situaci, což je pro počítač velmi složitý úkol.

Za účelem překonání těchto obtíží při hodnocení situace zavedli vědci v téhle generaci algoritmus mini-max, algoritmus pro vyhledávání stromu Monte Carlo (dále jen MCTS). Metoda Monte Carlo je numerická výpočetní metoda založená na pravděpodobnostní statistické teorii, která využívá náhodná čísla k řešení složitých výpočetních problémů. Ve hře Go MCTS hodnotí výhody a nevýhody aktuální situace prostřednictvím simulace velkého množství náhodných partií.

Jinak řečeno, MCTS simuluje pomocí počítače velké množství možných partií, následně statisticky vyhodnocuje výsledky každé partie a nakonec dospívá k hodnocení aktuální situace. Například simulací 10 000 partií, pokud černý hráč vyhraje 5500 partií a bílý hráč vyhraje 4500 partií, pak se aktuální situace považuje za trochu výhodnější pro černého hráče s výhrou 55%.

Výhody tohoto algoritmu zahrnují plné využití výpočetní síly počítače, čímž se simulací více partií dosáhne přesnějších výsledků, a přirozenou podporu paralelních výpočtů. Avšak MCTS má také několik nevýhod, včetně toho, že rozhodování o tazích je založeno pouze na statistických výsledcích, což může vést k nedostatku logické souvislosti mezi tahy.

3.3.3 Třetí generace

V průběhu posledních desetiletí až do roku 2015 čelil vývoj umělé inteligence v oblasti go značným technologickým výzvám. I přesto, že byla vybavena tehdejšími nejmodernějšími algoritmy a technologiemi, výkon umělé inteligence byl dosti omezený. Dokonce i nejlepší umělá inteligence té doby, jako například Zen, dokázala ve fair play porazit jen hráče s amatérským hodnocením 5.dan. [4] Vyzývat profesionální hráče vyžadovalo poskytnutí nejméně pěti kamenů podle pravidel handicapu. V tomto kontextu mnoho odborníků v oboru obecně předpokládalo, že umělá inteligence potřebuje k tomu, aby v go porazila člověka, alespoň několik desetiletí, a možná až století. [8.][16.]

Nicméně vzrušujícím zvratem byl nástup AlphaGo od společnosti DeepMind právě v tomto pesimistickém prostředí. Poprvé se představil v roce 2015 a s výsledkem 5-0 porazil Evropského šampiona, profesionálního hráče 2. dan, Fan Hui. Rychle tak prokázal svou vynikající úroveň ve hře go. Následně v roce 2016 vyzval k souboji jihokorejského několika násobného světového šampiona v go, Lee Sedola, čímž vytvořil důležitý moment v historii go.

AlphaGo pomocí svých unikátních technik hlubokého učení a posilovaného učení úspěšně porazila Lee Sedola, což symbolizovalo průlom v oblasti umělé inteligence v go. V roce 2017 pak AlphaGo dokázala porazit čínského hráče Ke Jie, tehdy světovou jedničku držící neporazitelný rekord po dobu dvou let. Od té doby žádný lidský hráč go nedokázal v fair play zápase porazit špičkovou umělou inteligenci.

Důvodem, proč AlphaGo dokázala dosáhnout takového rychlého pokroku, spočíval v používání hluboké konvoluční neuronové sítě k porozumění složitým vzorům ve hře go a kombinaci metod posilované učení a samohry při tréninku (self-play reinforcement learning). Tato unikátní tréninková metoda umožnila AlphaGo lépe chápat situaci, předpovídat tahy soupeře a projevovat excelentní výkon při dlouhodobém plánování.[11.]

Úspěch AlphaGo nespočíval pouze v technologickém průlomu, ale také v tom, že během svého tréninku plně integrovala zkušenosti lidských profesionálních hráčů go. Skrze hry s hráči na světové úrovni AlphaGo

[4] Dan představuje úroveň dovedností hráče, kde vyšší dan značí vyšší úroveň schopností. U profesionálního hráče je dan inverzní, kde nižší číslo znamená vyšší úroveň dovedností.

absorbovala obrovské množství cenných informací o go, což jí poskytlo hlubší a širší perspektivu při rozhodování. Tato série úspěchů představovala obrovský posun v oblasti umělé inteligence v go, zároveň přinesla nové poznatky pro celý obor umělé inteligence.[12.]

4 Pravidlo Go

4.1 Položení kamenů

- Hra probíhá na hrací desce se devatenácti svislými a devatenácti vodorovnými čarami, což dohromady tvoří 361 průsečíků.
- Hráči mají každý svou barvu kamenů, černý začíná a bílý následuje. Střídají se ve hře po jednom kameni.
- Kámen se umisťuje na průsečík herní desky.
- Po umístění kamene není povoleno jej přesouvat na jiné místo.
- Střídání tahů je právem obou hráčů, ale je dovoleno kterémukoliv hráči odmítnout právo na tah a místo toho použít PASS.

4.2 Kameny a jejich volnosti

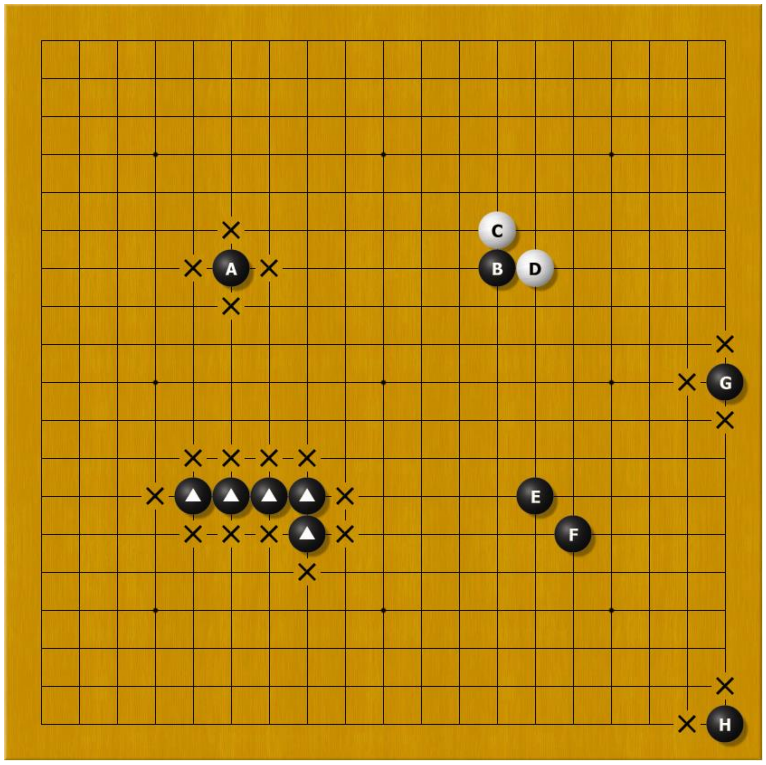
Když je kámen umístěn na herní desce, sousední body na přímce jsou pro tento kámen volnosti. Jak ukazuje obrázek 2 níže, kámen A má celkem 4 volnosti na místech označených X.

Kameny různých barev lze umístit na sousední průsečíky s cílem blokovat nebo obklopit odpovídající kameny. V obrázku má černý kámen B pouze dvě volnosti, neboť dva ze čtyř sousedních průsečíků jsou obsazena bílými kameny C a D. Stejně tak bílé kameny C a D mají každý tři volnosti.

Pokud se kámen nachází na okraji desky, je zřejmé, že má pouze tři sousední průsečíky, a tedy jen tři volnosti. Nachází-li se v rohu, má pouze dvě sousední průsečíky, a tedy jen dvě volnosti. Černý kámen G disponuje pouze třemi volnostmi na třech průsečících označených X, a černý kámen H má pouze dvě volnosti na dvou průsečících označených X.

V případě, že je několik kamenů stejné barvy spojeno dohromady, tvoří tzv. "blok kamene". Blok kamene sdílí svá volnosti a je buď celý živý, nebo celý mrtvý. Pět černých kamenů označených trojúhelníkem jsou spojeny dohromady, tvoříce tak jeden blok kamene. Tento blok kamene disponuje celkem jedenácti

volností na průsečících označených X. Černé kameny E a F nejsou spojeny, a tedy nejsou součástí stejného bloku kamene.



Obr. 2 Ukázka volnosti (Zdroj:[21.])

Volnosti každého kamene jsou zároveň jeho životem. Pokud jsou všechny volnosti bloku kamene (včetně jednotlivého kamene) zablokovány, blok je považován za zabitý a musí být okamžitě odstraněn ze hry.

4.3 Zákazové body

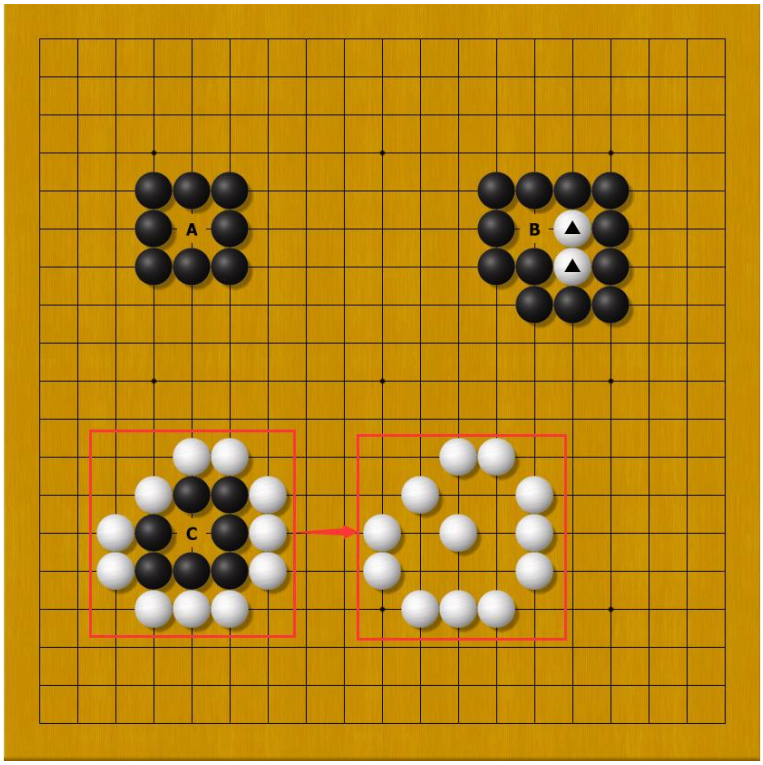
Na hrací desce go nejsou všechny prázdné body vhodné pro umístění kamene (samozřejmě nelze umístit kámen na již obsazené pozice), tyto pozice se nazývají zákazové body. Existují dvě situace:

1. Kámen vložený vlastním hráčem se nachází ve stavu bez volnosti a není schopen zabít žádné okolní kameny soupeře.
2. Vložení kamene vlastním hráčem vytváří pro soupeře stejnou situaci jako v předchozím tahu.

Prázdná místa A a B viz obrázek 3 jsou oba zákazové body pro bílého hráče. V místě A bílý kámen po umístění zůstane ve stavu bezvolnosti. Na místě B bílý kámen po umístění vytvoří blok spolu se dvěma bílými kameny

označenými ▲, který také z přechází do stavu bez volnosti, a navíc nemůže zabít okolní černé kameny.

Místo C není zákazovým bodem pro bílého hráče, protože i když bílý kámen po umístění zůstane bez volnosti, umístění bílého kamene na místě C může vyplnit poslední volnost kolem černých kamenů v okolí, a tím je odstranit.



Obr. 3 Ukázka zákazového bodu (Zdroj:[21.])

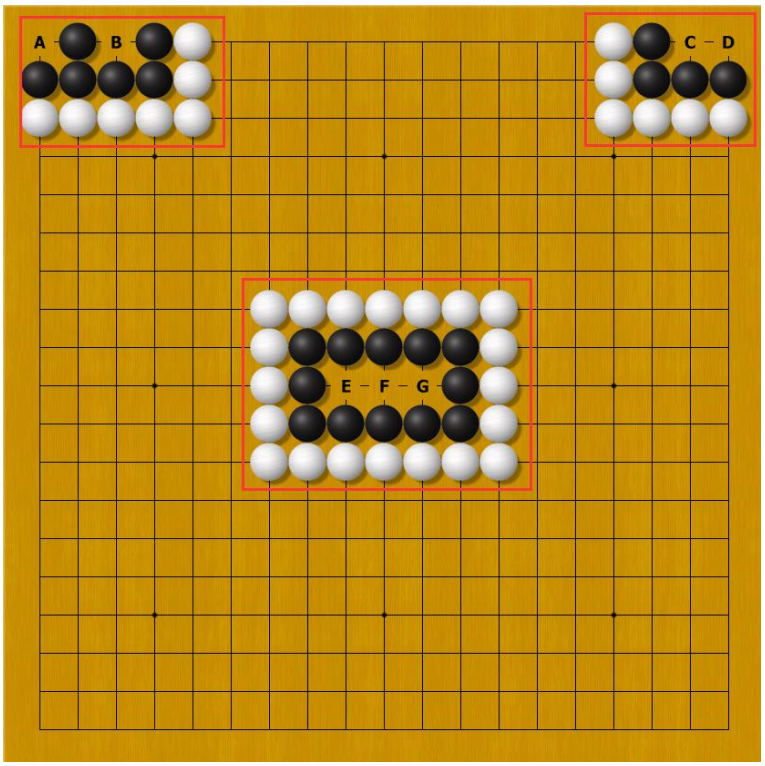
4.4 Živost bloku a oko

Ve zobrazeném obrázku 4 je blok černých kamenů v levém horním rohu obklopen bílými kameny, a jediné dvě volnosti A a B jsou pro bílého hráče zákazové body. Dokud černý hráč položí kámen na pozici A nebo B, bílý hráč nemůže na těchto místech položit kámen. Takže bez ohledu na situaci se tento blok černých kamenů nemůže dostat do situace, kdy by byl zlikvidován. Místa A a B jsou pro černého hráče nazývána "oči". Blok se považuje za živý, pokud má nejméně dvě oči, jinak je považován za mrtvý.

Situace v pravém horním rohu je podobná té v levém, ale i když jsou zde dvě volnosti, existuje pouze jedno oko. Bílý hráč může zabít černý kámen, stačí, když na pozici C nebo D umístí kámen. Pokud bílý hráč umístí kámen na pozici C, černý kámen bude mít pouze jedno oko, bez ohledu na to, zda černý hráč

umístí kámen na pozici D. Pokud černý hráč neumístí kámen na pozici D, bílý hráč může dále umístit kámen na pozici D a sníst tak černý kámen. Pokud černý hráč umístí kámen na pozici D a sní bílý kámen na pozici C, černý kámen stále bude mít pouze jedno oko (protože na pozici D je již černý kámen), a bílý hráč může na pozici C umístit kámen a sníst tak černý kámen.

Ve střední části hrací desky musí černý hráč obsadit pozici F, jinak, jakmile tento bod obsadí bílý kámen, tento blok bude mít pouze jedno oko a bude obtížné uniknout možnosti být úplně odstraněn.



Obr. 4 Ukázka živosti bloku (Zdroj:[21.])

4.5 Dosažení výhry

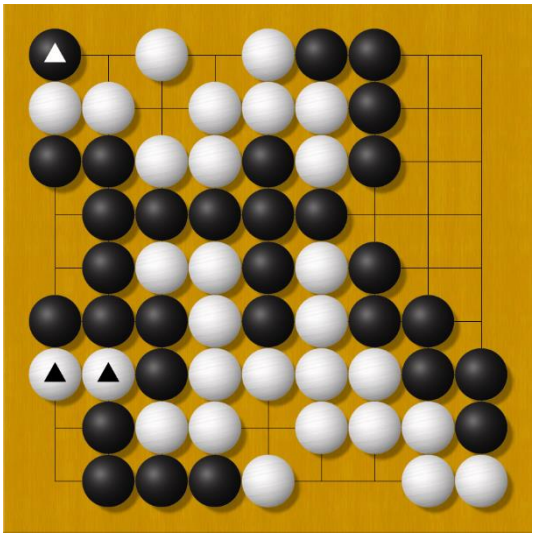
Celkem jsou dva způsoby, jak dosáhnout výhry:

- Výhra nebo prohra ve střední fázi hry (middle game): Pokud hráč rezignuje během hry, tak soupeř vyhrává ve střední fázi hry.
- Když hra dosáhne určité fáze (obvykle když je živost každého bloku definitivní), černý a bílý hráč se dohodnou na ukončení hry a výsledek se rozhodne podle velikosti oblastí na herní desce.

Při dohodě na ukončení hry se výsledek vyhodnocuje následujícím způsobem:

1) Likvidace mrtvých kamenů:

Kameny v obrázku 5 označené symbolem ▲, ať už jsou černé nebo bílé, jsou považovány za mrtvé. Toto je způsobeno tím, že při střídavém umisťování kamenů nemohou již vytvořit dvě oči.



Obr. 5 Likvidace mrtvých kamenů (Zdroj:[21.])

2) Vypočítat velikost území, které černý a bílý hráč obklopili (počet prázdných křížení + počet odpovídajících kamenů):

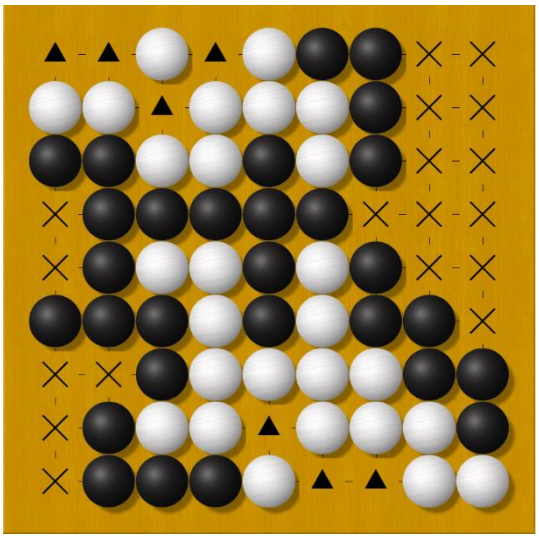
Černý hráč celkově obklopil 18 průsečíků označených X a navíc na hrací desce má 29 černých kamenů. Takže černý hráč má území o velikosti 18 + 29 = 47.

Bílý hráč obklopil 7 průsečíků označených ▲ a na hrací desce má 27 bílých kamenů. Celkově tedy bílý hráč má území o velikosti 7 + 27 = 34.

3) Vypočítat výsledek:

Na hrací desce zobrazené na obrázku 6 je celkem 81 míst (9*9). Celkově tedy polovina území je 40.5. Protože černý hráč hraje první, což je výhoda, při výpočtu výsledku černý hráč potřebuje přidat tři a tři čtvrtiny kamenů (3.75), tedy znamená, že černý hráč musí obklopit území, které je větší než polovina desky a ještě o 3.75 více (celkem 44.25), aby získal vítězství.

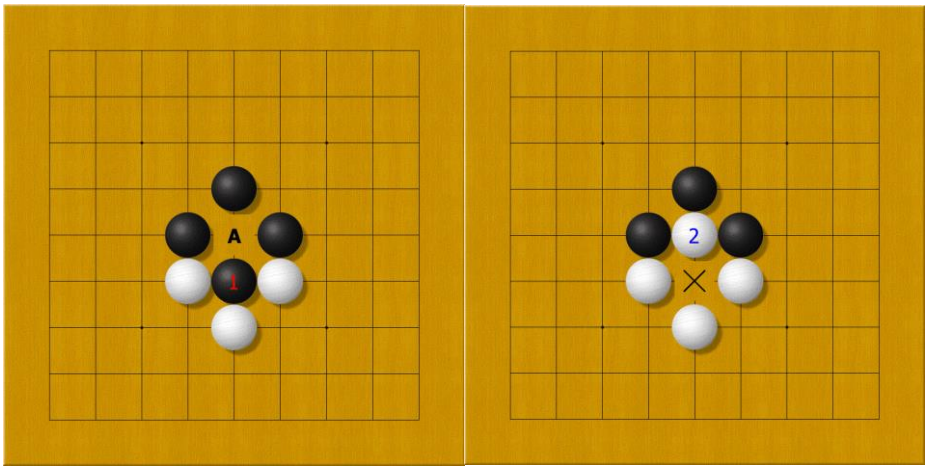
Protože území černého hráče má velikost 47, rozdíl 47-44.25=2.75. Výsledek partie je tedy vítězství černého hráče o 2 a 3/4 kamene.



Obr. 6 Výpočet velikosti území (Zdroj:[21.])

4.6 Ko

Termín "ko", pocházející z japonštiny, znamená nehodu nebo nežádoucí situaci. Popisuje zvláštní situaci během hry, když hráč úspěšně odstraní jeden protivníkův kámen, i když zbývá pouze s jednou volnosti. Protivník nemůže okamžitě na stejné místo položit kámen a získat ho zpět, aby se zabránilo nekonečnému opakování této situace. Proto hráč, který právě odebral kámen, musí na svém tahu zahrát na jiném místě a počkat na další kolo, aby mohl vzít zpět tento kámen.



Obr. 7 a Obr. 8 Ukázka Ko (Zdroj:[21.])

5 Hluboké učení s posilováním

Hluboké učení s posilováním (Deep Reinforcement Learning) je oblast, která stojí za to hlouběji studovat a je velmi zajímavá. Nicméně její matematické principy jsou náročné, překonávají dokonce i hluboké učení, a struktura je složitá, s mnoha koncepty. Učení posilováním je sekvenciální rozhodovací proces, který interaguje s prostředím prostřednictvím inteligentního agenta a snaží se najít sérii pravidel pro rozhodování (tzv. strategie), která umožní systému získat maximální kumulativní odměnu, tedy dosáhnout maximální hodnoty. Prostředí je objekt, se kterým inteligentní agent interaguje, a lze ho abstraktně chápat jako pravidla nebo mechanismy interakce, například v pravidlech go hry.

Matematickým základem a nástrojem pro modelování učení posilováním je Markovův rozhodovací proces (Markov Decision Process, MDP). MDP je obvykle složen ze stavového prostoru, akčního prostoru, funkce přechodu stavů, odměnové funkce a dalších prvků. Tato kapitola se převážně inspirovala zdroji [14.][15.].

5.1 Základní Koncepty

5.1.1 Stav (State):

Reprezentace aktuálního prostředí v daný okamžik, sloužící jako jediný základ pro rozhodování. Ve hře Go je stavem rozložení všech kamenů na herní desce.

5.1.2 Prostor stavů (State Space)

Množina všech možných stavů, značená jako S. Prostor stavů může být diskrétní nebo spojitý, konečný nebo spočetně nekonečný. Ve hře Go je prostor stavů diskrétní konečná množina, zahrnující všechny možné konfigurace desky.

5.1.3 Akce (Action)

Rozhodnutí, které agent učiní na základě aktuálního stavu. Ve hře Go, s 361 pozicemi (19*19) na desce a možností PASS (nezahrát tah), existuje 362 možných akcí. Volba akce může být deterministická nebo následovat pravděpodobnostní rozdělení.

5.1.4 Akční prostor (Action Space)

Označuje množinu všech možných akcí, obvykle značenou jako A. V příkladu hry Go je akční prostor definován jako $A = \{0,1,2 \dots 361\}$, kde i-tá akce značí umístění kamene na i -tou pozici (počínaje od 0), a 361. akce značí PASS.

5.1.5 Odměna (Reward)

Numerická hodnota vrácená agentovi prostřednictvím prostředí po provedení akce. Odměny jsou často definovány uživatelem a hrají klíčovou roli ve výsledcích učení posilováním. Odměny jsou obvykle funkcí stavu a akce.

5.1.6 Přechod stavů (State Transition)

Proces přechodu z aktuálního stavu v čase t na další stav v čase $t + 1$.V příkladu hry Go, na základě aktuálního stavu (konfigurace desky) položí černý nebo bílý kámen, což vytvoří nový stav (novou konfiguraci desky).

5.1.7 Strategie (Policy)

Odkazuje na způsob rozhodování na základě pozorovaného stavu, tj. jak vybrat akci z akčního prostoru. Strategie může být deterministická nebo stochastická. V oblasti reinforcement learning se bezmodelové metody obvykle dělí na učení strategie a učení hodnoty. Cílem učení politiky je získání funkce strategie, která na základě pozorovaného stavu rozhodne.

Stav se označuje jako S, akce jako A a stochastická funkce strategie $\pi: S \times A \mapsto [0,1]$ je funkce hustoty pravděpodobnosti, označovaná jako $\pi(a|s) = \mathbb{P}(A = a|S = s)$. Vstupem do funkce jsou akce a a stavs, výstupem je pravděpodobnostní hodnota mezi 0 a 1. Aktuální stav a všechny akce z prostoru akcí jsou zadány do strategické funkce, což generuje pravděpodobnost každé akce. Na základě pravděpodobností akcí lze provést výběr akce prostřednictvím vzorkování.

Deterministická strategie je zvláštním případem stochastické strategie$\mu: S \mapsto A$. Na základě vstupního stavu s rozhoduje o přímém výstupu akce $a = \mu(s)$, aniž by vydávala pravděpodobnostní hodnotu. Pro daný stav s je rozhodnutí a deterministické, bez prvků náhody.

5.1.8 Funkce přechodu stavu (State Transition Function)

Funkci, kterou prostředí používá k vytváření nového stavu s'. Vzhledem k tomu, že přechody stavů jsou obvykle náhodné, v oblasti posilovaného učení se k popisu těchto přechodů používá pravděpodobnostní funkce přechodu stavu (state transition probability function). Tato funkce je podmíněnou pravděpodobnostní hustotou $p(s'|s,a) = \mathbb{P}(S' = s'|S = s', A = a)$. Vyjadřuje pravděpodobnost pozorování aktuálního stavu s, provedení akce a agentem a změnu stavu prostředí na s'.

Deterministický přechod stavu představuje zvláštní případ náhodného přechodu, kde je pravděpodobnost soustředěna pouze na jednom stavu s'.

5.1.9 Interakce mezi agentem a prostředím (Agent-Environment Interaction)

Proces, kdy inteligentní agent pozoruje stav prostředí s, provede akci a, akce mění stav prostředí, a prostředí poskytuje zpětnou vazbu agentovi ve formě odměny r a nového stavu s'.

5.1.10 Epizody (Episodes)

Pojem pochází z herního průmyslu a označuje proces, kdy agent začíná hru až do úspěšného dokončení nebo ukončení hry.

5.1.11 Trajektorie (Trajectory)

Odkazuje na sled všech pozorovaných stavů, provedených akcí a získaných odměn $s1, a1, r1, s2, a2, r2, s3, a3, r3$ ... během jedné epizody hry.

5.1.12 Markovova vlastnost (Markov Property)

Označuje skutečnost, že následující stav S_{t+1} závisí pouze na aktuálním stavu S_t a akci A_t, a nezávisí na předchozích stavech a akcích. Pokud má přechod stavu Markovovu vlastnost, pak $\mathbb{P}(S_{t+1}|S_t, A_t) = \mathbb{P}(S_{t+1}\,|S_1, A_1, S_2, A_2, \ldots S_t, A_t)$.

5.1.13 Návratnost (Return)

Návratnost je součet všech odměn od aktuálního okamžiku do konce daného kola. Proto se návrat někdy nazývá kumulativní budoucí odměna (cumulative future reward). Jelikož odměny jsou funkcí stavu a akce, má návratnost prvkem náhody. Označme návratnost v čase t jako náhodnou

proměnnou U_t a předpokládejme, že daný tah končí v čase n, pak $U_t = R_t + R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_n$

Návratnost představuje celkovou sumu budoucích odměn a jeden z přístupů k posílenému učení spočívá v hledání strategie, která maximalizuje očekávanou návratnost. Tato strategie je nazývána optimální strategie (optimum policy). Metoda posíleného učení, která se snaží maximalizovat očekávanou návratnost k dosažení optimální strategie, se nazývá metoda učení strategie (policy learning).

5.2 Funkce hodnoty (value function)

5.2.1 Funkce hodnoty akce (Action-Value Function)

Dynamická funkce, která předpovídá hodnotu provedení konkrétní akce v daném stavu. Aby bylo možné rozhodnout, který tah má být proveden, musí AlphaGo odhadnout hodnotu každé možné varianty tahu. Návratnost U_t je vážená suma všech odměn od času t do konce hry. V čase t, pokud je známo hodnotu U_t, lze odhadnout, zda je situace dobrá nebo špatná. U_t je náhodná proměnná, a pokud v čase t pozorujeme stav jako s_t, po provedení rozhodnutí a výběru akce jako a_t, pak náhodnost proměnné U_t pochází z všech stavů a akcí od času $t+1$ jako: $S_{t+1}, A_{t+1}, S_{t+2}, A_{t+2}, \dots, S_n, A_n$.

Chceme-li v čase t odhadnout hodnotu U_t , řešením je vypočítat podmíněnou střední hodnotu od U_t a eliminovat tak náhodnost:

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}, \dots, S_n, A_n}[U_t | S_t = s_t, A_t = a_t]$$

Výsledek $Q_\pi(s_t, a_t)$ je právě funkce hodnoty akce.

Jelikož náhodné proměnné $S_{t+1}, A_{t+1}, S_{t+2}, A_{t+2}, \dots, S_n, A_n$ je eliminováno, takže funkce hodnoty akce $Q_\pi(s_t, a_t)$ závisí pouze na s_t a a_t, a nezávisí na stavech a akcích od času $t+1$ a později. Dále protože pravděpodobnostní funkce pro akcí $A_{t+1}, A_{t+2}, \dots, A_n$ je π, použitím různých hodnot π při výpočtu podmíněné střední hodnoty může vést k různým výsledkům, a proto $Q_\pi(s_t, a_t)$ závisí také na strategické funkci π.

Celkově lze říci, že funkce hodnoty akce $Q_\pi(s_t, a_t)$ v čase t závisí na těchto třech faktorech:

1. Aktuální stav s_t . Čím lepší je aktuální stav, tím vyšší je hodnota $Q_\pi(s_t, a_t)$, tj. očekávání návratnosti je vyšší.

2. Aktuální akce a_t. Čím lepší je tah provedený agentem, tím vyšší je hodnota $Q_\pi(s_t, a_t)$.
3. Strategie π . Kvalita strategie ovlivňuje budoucí kvalitu akcí $A_{t+1}, A_{t+2}, \dots, A_n$, takže čím lepší je strategie, tím vyšší je hodnota $Q_\pi(s_t, a_t)$.

5.2.2 Funkce hodnoty stavu (State-Value Function)

Funkce, která předpovídá hodnotu celého stavu, zohledňující všechny možné akce a následné stavy. Pro AlphaGo by to znamenalo odhadnutí, jaký je celkový odhad úspěchu pro danou herní pozici a strategii.

Vezměme akční hodnotovou funkci $Q_\pi(s_t, a_t)$, kde akce považujeme za náhodnou proměnnou A_t . Poté vypočítáme střední podmíněnou hodnotu vzhledem k A_t a následně eliminujeme A_t, čímž získáme funkci hodnoty stavu:

$$V_\pi(s_t) = \mathbb{E}_{A_t \sim \pi(\cdot|s_t)}[Q_\pi(s_t|, A_t)] = \sum_{a \in A} \pi(a|s_t) \cdot Q_\pi(s_t, a)$$

Funkce hodnoty stavu $V_\pi(s_t)$ závisí pouze na strategie π a aktuálním stavu s_t, nezávisí na akci. Tato funkce $V_\pi(s_t)$ také představuje očekávání návratnosti $V_\pi(s_t) = \mathbb{E}_{A_t, S_{t+1}, A_{t+1}, \dots, S_n, A_n}[U_t | S_t = s_t]$.Střední hodnota eliminuje náhodné proměnné závislé na návratnosti U_t . Čím vyšší je hodnota stavové funkce, tím lepší je očekávaný návrat. Stavová hodnotová funkce umožňuje měřit kvalitu strategie π v daném stavu s_t.

5.3 Neuronové sítě

5.3.1 Strategická síť (Policy Network)

Tato síť hraje klíčovou roli v AlphaGo a je důležitou součástí hlubokého posilovaného učení. Jejím hlavním úkolem je na základě aktuálního stavu herní situace ve hře Go vytvořit pravděpodobnostní rozložení pro každý možný tah. Konkrétně má síť strategie strukturu konvoluční neuronové sítě (CNN). Provádí konvoluční a pooling operace na vstupu, což je stav aktuálního rozložení kamenů na herní desce, a nakonec generuje vektor pravděpodobnosti, který vyjadřuje pravděpodobnost výběru každého možného tahu.

Matematicky řečeno, výstup strategické sítě lze vyjádřit jako podmíněnou pravděpodobnostní distribuci, tj. pravděpodobnost každého jednotlivého akčního kroku vzhledem k aktuální herní situaci. Označme aktuální

situaci jako stav s a akci jako a. Výstup strategické sítě lze potom vyjádřit jako $\pi(a|s)$.

V praxi však strategická síť používá $\pi(a|s;\theta)$ k aproximaci funkce strategie $\pi(a|s)$ pomocí neuronové sítě, kde θ představuje parametry této neuronové sítě. Tato hodnota θ je nejprve inicializována a poté je aktualizována pomocí shromážděných informací o stavech, akcích a odměnách.

Během tréninku strategická síť, pomocí algoritmů posilovaného učení, upravuje pravděpodobnostní distribuci výstupu na základě konečného výsledku hry (výhra, prohra nebo remíza). Tímto způsobem se snaží zvýšit pravděpodobnost úspěšných tahů, což vede k celkovému zlepšení výkonu modelu.

Role strategické sítě není pouze ve volbě následujícího tahu, ale má také klíčový význam při tvorbě stromu prohledávání. V metodě Monte Carlo Tree Search (MCTS) strategická síť řídí prioritu pro prohledávání, což znamená, že tahy, které mají větší pravděpodobnost vede k vítězství, jsou prozkoumány dříve.

5.3.2 Hodnotová síť (Value Network)

Síť, jejíž návrh a trénink jsou zaměřeny na hodnocení aktuální situace v go. Tedy na to, jaká je aktuální výhoda nebo nevýhoda pro hráče. Hlavním úkolem hodnotové sítě je poskytnout celkové hodnocení aktuální situace, což pomáhá umělou inteligenci při rozhodování s dlouhodobější a globálnější perspektivou.

Hodnotová síť využívá $q(s,a;\omega)$ k aproximaci funkce hodnoty $Q_\pi(s,a)$ nebo v$(s;\theta)$ k aproximaci funkce hodnoty stavu $V_\pi(s)$ pomocí neuronové sítě, kde ω představuje parametry této neuronové sítě. Struktura neuronové sítě je předem stanovena a parametry ω jsou na začátku náhodně inicializovány. Tyto parametry se následně upravují prostřednictvím interakce mezi agentem a prostředím během učení.

Společně se strategickou sítí doplňuje hodnotová síť AlphaGo schopnost provádět ořezávání v hledacím stromu a vybírat větve s největším potenciálem. Během tréninku AlphaGo využívá algoritmy posilovaného učení, a prostřednictvím spolupráce s strategickou sítí, se zapojuje hodnotový výstup do optimalizace modelu, čímž zlepšuje přesnost odhadované hodnoty.

Hodnotová síť, hodnotící výhody a nevýhody celkové situace, pomáhá AlphaGo lépe porozumět celkovému průběhu partie a umožňuje lepší vyvážení krátkodobých výhod a dlouhodobých cílů při rozhodování. Tento globální pohled přispívá k tomu, aby AlphaGo mohla činit uvážená rozhodnutí a zvyšuje celkový výkon systému ve složitých strategických hrách, jako je go.

5.4 Monte Carlo

Monte Carlo je město v Monackém knížectví ležící na pobřeží Středozemního moře v Evropě. Monte Carlo tvoří historické centrum Monaka a je známé po celém světě jako prestižní centrum hazardu.

Monte Carlo algoritmus nebo Monte Carlo metoda, je jedním z důležitých numerických výpočetních přístupů, který byl vyvinut v polovině 20. století s rozvojem matematické informatické vědy s vynálezem počítačů. Tato metoda je založena na pravděpodobnostní a statistické teorii a používá náhodná čísla (nebo častěji pseudonáhodná čísla) k řešení různých výpočetních problémů.

Tento přístup spočívá ve využívání náhodných čísel k simulaci různých situací a odhadu výsledků pomocí pravděpodobnostních metod. Termín "Monte Carlo" byl zvolen právě kvůli spojení s hazardními hrami, protože mnoho výpočetních problémů lze přirovnat k matematickým problémům v hazardních hrách.

Algoritmus odhaduje skutečné hodnoty pomocí náhodných vzorků, například odhad očekávané hodnoty nebo aproximace gradientu cílové funkce vzhledem k parametrům neuronové sítě.

Výstup hodnotové sítě představuje očekávaný návrat U_t. V posilovaném učení můžeme hrát hru až do konce, pozorovat všechny odměny $r_1, r_2, \cdots r_n$, a pak vypočítat návrat $u_t = \sum_{i=0}^{n-t} \gamma^i r_{t+i}$. Při tréninku hodnotové sítě slouží u_t jako cíl.

Výhodou Monte Carla je nestrannost (unbiasedness): u_t a je nestranný odhad $Q_\pi(s, a)$. Díky této skutečnosti lze hodnotovou síť trénovat na cíli u_t, což vede k nestranné hodnotové síti.

Nevýhodou Monte Carla je vysoký rozptyl: náhodná veličina U_t závisí na náhodné proměnné $S_{t+1}, A_{t+1}, S_{t+2}, A_{t+2}, \dots, S_n, A_n$, což představuje velkou neurčitost. I když jsou pozorované hodnoty u_t nestranným odhadem U_t, mohou

se ve skutečnosti od $\mathbb{E}[U_t]$ vzdálit. Proto použití u_t jako cíle pro trénování hodnotové sítě může vést k pomalé konvergenci.

Trénink AlphaGo je částečně založen na Monte Carlo stromovém prohledávání, a kvůli nevýhodě velkém rozptylu tohoto přístupu je proces tréninku AlphaGo velmi pomalý.

6 Program Go

Tato aplikace umožňuje simulovat jednoduchou herní desku pro go o velikosti 9x9 a nebo standardní herní desku o velikosti 19x19 a umožňuje hráčům hrát proti sobě a vyhodnocovat pravidla go, včetně pravidel pro ko, zákazový bod, oko atd.(Tato funkce především využívá projekt GymGo, který je open-source od autora aigagror.)

Kromě toho aplikace obsahuje simulátor, který pomocí algoritmu podobného AlphaGo umožňuje vizualizaci tréninku umělé inteligence ve hře go na 9x9 herní desku. Dále obsahuje kompletní sadu nástrojů pro jednoduchý vývoj, trénink a vizualizaci výsledků umělé inteligence ve hře go.

6.1 Logika programu pro hru Go

6.1.1 Způsob reprezentace stavu desky

Hrací plocha Go se skládá z 19 vertikálních a horizontálních segmentů s 361 křižovatkami a každá křižovatka má tři stavy: černý kámen, bílý kámen nebo žádný existující kámen. Teoreticky lze tedy k vyjádření stavu herní desky použít dvourozměrnou matici. Například pokud se k vyjádření stavu desky Go použije matice o rozměrech 19*19, 0 znamená, že na příslušné pozici není žádný kamen, 1 znamená, že na příslušné pozici je černý kámen, a -1 znamená, že na příslušné pozici je bílý kámen.

Přestože výše uvedená metoda reprezentace stavu herní desky může kompletně reprezentovat stav hrací plochy Go, v programátorské praxi je poměrně obtížné pomocí této metody reprezentace vypočítat volnost bloku kamene, efektivní pozici tahu atd. [17.] V jádře GymGo se používá vhodnější reprezentace:

Stav šachovnice Go je reprezentován tenzorem s Shape (NUM_CHNLS, SIZE, SIZE), jehož každý prvek má hodnotu 0 nebo 1, kde hodnota

NUM_CHNLS je 6, což představuje počet kanálů v tenzoru, a SIZE představuje velikost herní desky. Význam dat pro každý kanál je následující:

CHANNEL[0]: kanál 0 je BLACK_CHANNEL, který označuje rozložení černých kamenů. Pozice černého kamene je 1, jinak je 0;

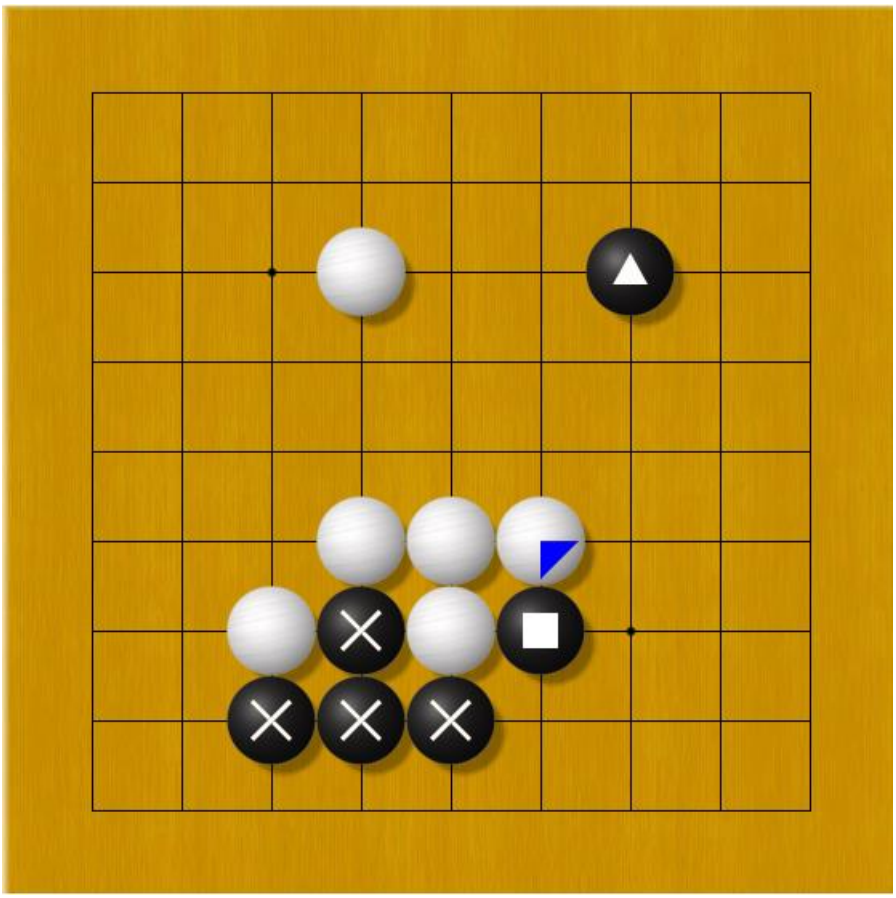
CHANNEL[1]: kanál 1 je WHITE_CHANNEL, který označuje rozložení bílých kamenů. Pozice bílého kamene je 1, jinak je 0;

CHANNEL[2]: kanál 2 je TURN_CHANNEL, udává stranu dalšího tahu, což je matice all-0 nebo all-1. 0: černý, 1: bílý;

CHANNEL[3]: kanál 3 je INVALID_CHANNEL, který označuje neplatnou pozici pro další položení. Neplatná pozice je 1, ostatní jsou 0;

CHANNEL[4]: kanál 4 je PASS_CHANNEL, který udává, zda je předchozí krok PASS nebo ne, a je to matice all-0 nebo all-1. 0: není PASS, 1: je PASS;

CHANNEL[5]: kanál 5 je DONE_CHANNEL, který udává, zda hra po posledním tahu skončila, a je maticí all-0 nebo all-1. 0: neskončila, 1: skončila.



Obr. 9 Ukázkový obrázek pro ilustraci (Zdroj:[21.])

Podle výše uvedeného způsobu reprezentace stavu desky je stav desky zobrazený na obrázku 9 s odpovídající reprezentací následující:

```
[[[0. 0. 0. 0. 0. 0. 0. 0. 0.][[0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 1. 0. 0.] [0. 0. 0. 1. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 1. 1. 1. 0. 0. 0.]
  [0. 0. 0. 1. 0. 1. 0. 0. 0.] [0. 0. 1. 0. 1. 0. 0. 0. 0.]
  [0. 0. 1. 1. 1. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.]][0. 0. 0. 0. 0. 0. 0. 0. 0.]]
         BLACK_CHANNEL                  WHITE_CHANNEL

 [[0. 0. 0. 0. 0. 0. 0. 0. 0.][[0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 1. 0. 0. 1. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 1. 1. 1. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 1. 1. 1. 1. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 1. 1. 1. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.]][0. 0. 0. 0. 0. 0. 0. 0. 0.]]
      TURN_CHANNEL              INVALID_CHANNEL

 [[0. 0. 0. 0. 0. 0. 0. 0. 0.] [[0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0.]][0. 0. 0. 0. 0. 0. 0. 0. 0.]]]
      PASS_CHANNEL              DONE_CHANNEL
```

Obr. 10 Matice obrázku9 (Zdroj: Vlastní tvorba)

6.1.2 Určení konce hry ve hře Go

Podle pravidel hry Go ve výše uvedené podkapitole 5 existují dva druhy situací, které mohou vyvolat konec hry Go:

1) v průběhu hry jeden z černých a bílých hráčů přizná porážku, tj. vyhraje střední hru/prohraje střední hru;

2) když hra dospěje do určité pozice, všechna území na hrací ploše byla rozdělena a oba hráči souhlasí se status quo současného rozdělení území, tj. je dojednány konec.

V praktickém programu může být konec vyvolán průběžným PASS pro černého i bílého, tj. když poslední tah provede černý a černý zvolí PASS, pak bílý také zvolí PASS; když poslední tah provede bílý a bílý zvolí PASS, pak černý také zvolí PASS.

Logika postupu pro realizaci rozhodnutí v koncovce je následující:

- Zkontroluje, zda je PASS_CHANNEL maticí all-1, a použitím proměnné **previously_passed** zjistí, zda předchozí akce byla PASS.
- Zjištění, zda je následující akce PASS.
- Pokud je to PASS, nastaví matici PASS_CHANNEL na matici all-1;
- Ověřit, zda předchozí akce je PASS, tj. zda hodnota proměnné previously_passed je True.
- Pokud je předchozí akce také PASS, pak se matice DONE_CHANNEL nastaví na matici all-1, což znamená, že hra po položení kamene skončila.

Částečná implementace programového kódu je následující:

```
1. # Deep copy the state to modify
2. state = np.copy(state)
3. # Initialize basic variables
4. board_shape = state.shape[1:]  # Rozměry hrací desky: (počet kanálů, výška desky, šířka desky)
5. pass_idx = np.prod(board_shape)  # np.prod() vynásobí všechny prvky v parametru, pass_idx: id pro "pass"
6. passed = action1d == pass_idx  # Pokud id akce odpovídá pass_idx, passed je True
7. action2d = action1d // board_shape[0], action1d % board_shape[1]  # Převedení akce z 1D na 2D formát
8.
9. player = turn(state)  # Určení hráče, který je na tahu
10.previously_passed = prev_player_passed(state)  # Zjištění, zda předchozí tah byl pass
11.ko_protect = None
12.
13.if passed:
14.    # We passed
```

```
    # Pokud byl tento tah pass, nastaví PASS_CHNL ve s
    tate na matici plnou jedniček
    state[govars.PASS_CHNL] = 1
    if previously_passed:
        # Hra skončila
        # Pokud byl předchozí tah také pass, hra skonč
    ila (oba hráči pasovali)
        # Nastaví DONE_CHNL ve state na matici plnou j
    edniček
        state[govars.DONE_CHNL] = 1
else:
    # Tah nebyl pass
    state[govars.PASS_CHNL] = 0
```

6.2 Realizace pravidla Go na úrovni počítače

6.2.1 Způsob vytvoření bloků kamene

Kdykoli je na desce přidán černý nebo bílý kámen, musí být stav desky aktualizován tak, že se zjistí, zda byl zabit soupeřův kámen (např. pokud je následující tah černý, pak je soupeř bílý a naopak).

Aby bylo možné určit, zda byl kámen zabit, je nutné zjistit, zda je množství volností pro daný kámen roven 0. Vzhledem k tomu, že volnost kamene je sdílená, tj. kámen v bloku žije a umírá společně, musí se nejprve rozdělit.

Metoda **scipy.ndimage.measurements.label()** označí prvky matice, které jsou odděleně rozsekány. Aby bylo možné kousky rozdělit na části, může být tato metoda použita tak, že různé části kousků jsou přímo označeny různými štítky.Pro rozdělení černých kamenů na kousky na obrázku 9 se používá tato metoda:

```
from scipy import ndimage
import numpy as np

black_pieces = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 1, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
                [0, 0, 0, 1, 0, 1, 0, 0, 0],
                [0, 0, 1, 1, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0]])
all_black_groups, _ = ndimage.measurements.label(black_pieces)
all_black_groups
```

Výstupní hodnota all_black_groups je následující:

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 2, 0, 3, 0, 0, 0],
       [0, 0, 2, 2, 2, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

6.2.2 Způsob výpočtu volnosti bloku kamene

Volnost bloku neboli dílku kamene se rovná počtu prázdných průsečíků přímo sousedících s tímto blokem. Logika postupu pro výpočet volnosti bloku je následující:

1. vypočítat rozložení všech prázdných průsečíků na desce;
2. vypočítat rozložení průsečíků přímo sousedících s daným blokem;
3. pokud je sousední průsečík prázdným průsečíkem, pak je tento průsečík volnost tohoto bloku. Počet takových prázdných průsečíků je počet volnosti daného bloku kamene.

6.2.3 Matice rozložení všech prázdných průsečíků na desce

Protože BLACK_CHANNEL odpovídá matici černých kamenů a WHITE_CHANNEL odpovídá matici bílých kamenů, získáme distribuční matici všech kamenů přímým sečtením příslušných prvků obou matic. Odečtením distribuční matice všech kamenů maticí all-1 získáme distribuční matici prázdných průsečíků.

```
from scipy import ndimage
import numpy as np

black_pieces = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 1, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 1, 0, 1, 0, 0, 0],
                         [0, 0, 1, 1, 1, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0]])
white_pieces = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 1, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 1, 1, 1, 0, 0, 0],
                         [0, 0, 1, 0, 1, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0]])

all_pieces = np.sum([black_pieces, white_pieces], axis=0)
empties = 1 - all_pieces
empties
```

Distribuční matice výstupních prvků je následující:

```
array([[1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 0, 1, 1, 0, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1],
```

```
[1, 1, 1, 0, 0, 0, 1, 1, 1],
[1, 1, 0, 0, 0, 0, 1, 1, 1],
[1, 1, 0, 0, 0, 1, 1, 1, 1],
[1, 1, 1, 1, 1, 1, 1, 1, 1]])
```

6.2.4 Distribuční matice přímých sousedních průsečíků bloku

Metoda **scipy.ndimage.binary_dilation()** nafoukne matici do určité struktury. Rozbalení černého bloku označeného X na obrázku 9 způsobí, že průsečík přímo sousedící s černými kameny označenými X bude mít hodnotu True.

```
from scipy import ndimage
import numpy as np

black_pieces = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 1, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 1, 0, 1, 0, 0, 0],
                         [0, 0, 1, 1, 1, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0, 0]])
all_black_groups, _ = ndimage.measurements.label(black_pieces)
x_group = all_black_groups == 2
print('x_group before binary dilation:\n', x_group)

x_group_dilation = ndimage.binary_dilation(x_group)
print('x_group after binary dilation:\n', x_group_dilation)
```

Výstupní matice **x_group** a **x_group_dilation** jsou následující:

```
x_group before binary dilation:
 [[False False False False False False False False False]
 [False False False False False False False False False]
 [False False False False False False False False False]
```

[False False False False False False False False False]
[False False False False False False False False False]
[False False False False False False False False False]
[False False False True False False False False False]
[False False True True True False False False False]
[False False False False False False False False False]]

x_group after binary dilation:
[[False False False False False False False False False]
[False False False False False False False False False]
[False False False False False False False False False]
[False False False False False False False False False]
[False False False False False False False False False]
[False False False True False False False False False]
[False False True True True False False False False]
[False True True True True True False False False]
[False False True True True False False False False]]

Je potřeba zmínit, že metoda **scipy.ndimage.binary_dilation()** nejenže uvede hodnoty průsečíků bezprostředně sousedících s blokem jako True, ale také uvede pozici každého kamene v tomto bloku jako True. I přesto, že tyto oblasti obsahují kameny, nemají vliv na výpočet volnosti, protože tyto kameny již byly brány v úvahu.

6.2.5 Distribuční matice pro volnosti jakéhokoliv bloku kamenů

Pokud je sousední průsečík prázdným průsečíkem, pak je tento průsečík volností tohoto bloku kamenů. Můžeme tedy použít matici **empties** k vynásobení s **x_group_dilation** a získat distribuční matici volnosti pro tento blok kamenů. Výpočtem výskytu jedniček v této matici, získáme volnost pro daný kus kamenů.

```
liberties = empties * x_group_dilation
print('liberties:\n', liberties)
num_liberties = np.sum(liberties)
print('num_liberties:', num_liberties)
```

Vstupní matice rozdělení volnosti a množství volnosti pro daný blok jsou následující:

liberties:

[[0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0]

[0 1 0 0 0 1 0 0 0]

[0 0 1 1 1 0 0 0 0]]

num_liberties: 5

6.2.6 Metoda aktualizace matice bloků

1. Aktualizace matice rozložení kamenů pro hráče příštího tahu:

Realizace této matice je poměrně jednoduchá a můžeme přímo převést další akci na souřadnice matice rozložení kamenů, posoudit, zda je souřadnicová pozice platnou pozicí pro položení kamene, pokud je platnou pozicí pro položení, pak nastavit hodnotu prvku pozice matice rozložení kamenů na 1.

2. Aktualizace matice rozložení kamenů pro hráče v předchozím tahu:

Aktualizace matice rozložení kamenů hráče, který provedl poslední tah, tj. poté, co hráč, který provedl další tah, provedl tah, určuje, zda byl některý z kamenů tohoto hráče po tomto tahu zabit.

Po dokončení aktualizace matice rozložení kamenů pro hráče příštího tahu lze metodu výpočtu volnosti popsanou v oddíle 7.2.2 použít k výpočtu volnosti každého kamene v okolí pozice kam hráč předchozího tahu položil kámen a nastavit hodnotu volnosti každého bloku kamenů, jejíž volnosti je v příslušné části matice rozložení kamenů předchozího tahu 0, na 0.

6.2.7 Metoda určení Ko

Podle popisu situace ko v části 5.2 vyplývá to, že k situaci ko dojde, pokud se ve všech čtyřech sousedních průsečících pozice kamene dalšího hráče nacházejí kameny hráče, který provedl předchozí tah. Zároveň, pokud poté, co hráč dalšího tahu zahrál kámen na pozici, kde hráč předchozího tahu má zabitý

blok 1 a počet kamenů v tomto bloku je také 1, vzniká ko. Hráč, který provedl předchozí tah, nemůže okamžitě provést tah na políčko odpovídající zabité kameni.

Proto se při aktualizaci matice rozložení kamenů v předchozím tahu, zaznamenávají společně i souřadnice zabitých kamenů a výše uvedenou metodou se zjišťuje, zda na hrací ploše došlo ke ko. Pokud dojde ke situaci ko, označí se příslušná pozice značkou pro ko.

6.2.8 Výpočet neplatných poloh položení (zákazový bod)

Když hráč následujícího tahu dokončí drop, musí se vypočítat pozice neplatného dropu soupeře dalšího hráče a aktualizovat matice INVALID_CHANNEL.

Podle pravidel hry Go splňuje pozice neplatného tahu soupeře dalšího hráče následující

podmínky:

1. Soupeř hráče, který provádí následující tah, nemůže provést tah v následující pozici:

- na obsazených polích (tj. na polích, kde jsou přítomny kameny neboli na nenulových průsečících);
- pozice označená ko;
- pozice prázdného průsečíku je neplatnou pozicí pro umístění kamene, pokud sousedí s jedním nebo více bloky, které mají pouze jednu volnost, a nesousedí s jedním nebo více bloky, které mají více než jednu volnost, a pokud jsou kolem tohoto prázdného průsečíku kameny;
- prázdný průsečík je neplatnou pozicí bloku, pokud je obklopen bloky hráče, který provádí další tah, a odpovídající kameny hráče, který provádí další tah, mají kolem tohoto prázdného průsečíku nejméně jednu volnost.

2. Soupeř strany, která provádí další tah, může provést tah na následujících polích:

- pokud soupeř strany, která má táhnout jako další, položí na místo, který je schopen zabit libovolný kámen hráče, který má táhnout jako další;

Logika výpočtu zákazových bodů je následující:

1. Vypočítat všechny možné neplatné pozice pro položení dalšího kamene, rozdělené následujícím způsobem:

- pozice všech kamenů každého bloku, které mají více než jednu volnost, hráče, který provádí další tah;
- pozice všech kamenů každého bloku, které mají více než jednu volnost, soupeře hráče, který provádí další tah;

2. Výpočet pozic, které musí být platné ze všech možných neplatných pozic umístění kamene, v následujících dvou případech:

- pozice všech kamenů s pouze jedinou volností hráče, který provede další tah;
- pozice všech kamenů s pouze jedinou volností soupeře hráče, který provede další tah;

3. Najít pozice, ve kterých jsou přítomny kameny na všechny 4 sousední průsečíky;
4. Pak všechny neplatné pozice tahu neboli zákazové body jsou: všechna pole s přítomnými kameny + pole, která jsou po vyloučení všech stoprocentně platných pozic tahu ze všech možných neplatných polí tahu zcela obklíčena + pole označená ko.

   Kód pro realizace:

```
1. def compute_invalid_moves(state, player, ko_protect=No
   ne):
2.
3.     # Všechny kameny a prázdná pole
4.     all_pieces = np.sum(state[[govars.BLACK, govars.WH
   ITE]], axis=0)
5.     empties = 1 - all_pieces  # Prázdná pole jsou 1 -
   všechny kameny
6.     # Nastavení pole neplatných a platných tahů
7.     possible_invalid_array = np.zeros(state.shape[1:])
8.     definite_valids_array = np.zeros(state.shape[1:])
9. # Získání všech skupin
10.#matice všech kamenů a jejich počet, hráče předchozího
    tahu
11.all_own_groups, num_own_groups = measurements.label(st
   ate[player])
12.#matice všech kamenů a jejich počet, hráče následující
   ho tahu
13.    all_opp_groups, num_opp_groups = measurements.labe
   l(state[1 - player])
```

```
    expanded_own_groups = np.zeros((num_own_groups, *state.shape[1:]))
    expanded_opp_groups = np.zeros((num_opp_groups, *state.shape[1:]))
    # Rozšíření skupiny tak, aby každá skupina byla ve vlastním kanálu
    for i in range(num_own_groups):
        expanded_own_groups[i] = all_own_groups == (i + 1)
    for i in range(num_opp_groups):
        expanded_opp_groups[i] = all_opp_groups == (i + 1)
    # Získání všech volností v rozšířené formě
    all_own_liberties = empties[np.newaxis] * ndimage.binary_dilation(expanded_own_groups, surround_struct[np.newaxis])
    all_opp_liberties = empties[np.newaxis] * ndimage.binary_dilation(expanded_opp_groups, surround_struct[np.newaxis])
    # Spočítání volností
    own_liberty_counts = np.sum(all_own_liberties, axis=(1, 2))
    opp_liberty_counts = np.sum(all_opp_liberties, axis=(1, 2))

    # Možné neplatné tahy jsou na jedné volnosti soupeřových skupin a na místech více volností vlastních skupin
    # Určitě platné tahy jsou na jedné volnosti vlastních skupin, více volností soupeřových skupin
    # nebo pokud nejste obklopeni
    possible_invalid_array += np.sum(all_own_liberties[own_liberty_counts > 1], axis=0)
    possible_invalid_array += np.sum(all_opp_liberties[opp_liberty_counts == 1], axis=0)

    definite_valids_array += np.sum(all_own_liberties[own_liberty_counts == 1], axis=0)
    definite_valids_array += np.sum(all_opp_liberties[opp_liberty_counts > 1], axis=0)

    # Všechny neplatné tahy jsou obsazená pole + (možné neplatné tahy mínus stoprocentně platné pozice a jejich obklopena místa)
    surrounded = ndimage.convolve(all_pieces, surround_struct, mode='constant', cval=1) == 4
    invalid_moves = all_pieces + possible_invalid_array * (definite_valids_array == 0) * surrounded
    # Ko-ochrana
    if ko_protect is not None:
        invalid_moves[ko_protect[0], ko_protect[1]] = 1
return invalid_moves > 0
```

6.2.9 Výpočet dalšího stavu herní desky

Na základě aktuálního stavu hrací plochy a další akce hráče, který má provést další tah aktualizuje stav hrací plochy, a to probíhá následovně:

1. určit, zda akce hráče, který provede další tah, je PASS, a aktualizovat matici PASS_CHANNEL a matici DONE_CHANNEL;
2. provádí aktualizaci matic BLACK_CHANNEL a WHITE_CHANNEL podle metody aktualizace matice popsané v části 7.2.6;
3. zaznamenávání pozic ko značek podle metody určování ko popsané v části 7.2.7;
4. výpočet pozice neplatného položení kamene soupeře hráče, který provádí další tah, podle metody výpočtu pozice neplatného tahu popsané v části 7.2.8 a aktualizace matice TURN_CHANNEL;
5. Aktualizace matice TURN_CHANNEL.

Konkrétní postup se provádí takto:

```
1. def next_state(state, action1d, canonical=False):
2.     # Deep copy the state to modify
3.     state = np.copy(state)
4.     # Initialize basic variables
5.     board_shape = state.shape[1:]  # Rozměry hrací des
   ky: (počet kanálů, výška desky, šířka desky)
6.     pass_idx = np.prod(board_shape)  # np.prod() vynás
   obí všechny prvky v parametru, pass_idx: id pro "pass"
7.     passed = action1d == pass_idx # Pokud id akce odpo
   vídá pass_id, passed je True
8.     action2d = action1d // board_shape[0], action1d %
   board_shape[1]  # Převedení akce z 1D na 2D formát
9.     player = turn(state)  # Určení hráče, který je na
                               tahu
10.    previously_passed = prev_player_passed(state)# Zji
   štění, zda předchozí tah byl pass
11.    ko_protect = None
12.    if passed:
13.        # We passed
14.        # Pokud tento tah byl pass, nastaví PASS_CHNL
             na matici plnou jedniček
15.        state[govars.PASS_CHNL] = 1
16.        if previously_passed:
17.            # Game ended
18.            # Pokud byl předchozí tah také pass,
                 hra skončila (oba hráči pasovali)
19.           #Nastaví DONE_CHNL na matici plnou jedniček
20.            state[govars.DONE_CHNL] = 1
21.    else:
```

```
        # Tah nebyl pass
        state[govars.PASS_CHNL] = 0

        # Zkontrolujte, zda je tah platný
        assert state[govars.INVD_CHNL, action2d[0], action2d[1]] == 0, ("Invalid move", action2d)
        # Přidej kámen
        state[player, action2d[0], action2d[1]] = 1
        # Získání sousedních pozic a zkontrolujte, zda bude kámen obklopen kamenem soupeře
        # Získání sousedních pozic k poslednímu tahu a zkontrolování,zda bude kámen obklopen kamenem soupeře
        adj_locs, surrounded = state_utils.adj_data(state, action2d, player)
        # Aktualizuj kameny
        # Aktualizace stavu hrací desky po provedeném tahu, vrátí seznam skupin, které byly zabity
        killed_groups = state_utils.update_pieces(state, adj_locs, player)
        #Pokud byla zabita pouze jedna skupina a ta je dna skupina měla pouze jeden kámen, a tento kámen byl obklopen,
        # aktivuj ochranu ko
        if len(killed_groups) == 1 and surrounded:
            killed_group = killed_groups[0]
            if len(killed_group) == 1:
                ko_protect = killed_group[0]
    # Aktualizuj neplatné tahy
    state[govars.INVD_CHNL] = state_utils.compute_invalid_moves(state, player, ko_protect)
    # Přepnout hráče
    # Nastavení hráče, který je na tahu
    state_utils.set_turn(state)
    # Pokud je nastaveno kanonické zobrazení,změní hráče na černého a prohodí barvy kamenů
    if canonical:
        # Set canonical form
        # Funkce pro nastavení kanonického tvaru hrací desky
        state = canonical_form(state)
return state
```

6.3 Framework hlubokého učení

Tento program využívá rámec Flying Paddle k vytvoření a trénování hlubokých neuronových sítí: strategické sítě a hodnotové sítě - ve strategii AlphaGo.[18.]

6.3.1 Konstrukce neuronové sítě

Postup sestavení neuronové sítě pomocí pádlového rámce je následující:

1. importovat knihovnu Paddle;
2. Definovat třídu zděděnou z paddle **.nn.Layer** a inicializovat podvrstvy (nebo parametry) neuronové sítě v metodě **__init__ ()**;
3. Přepsat metodu **forward()**, ve které je implementován výpočetní tok neuronové sítě.

Podstatou inicializace podvrstvy neuronové sítě v metodě **__init__ ( )** je inicializace parametrů neuronové sítě, přičemž různé podvrstvy v podstatě představují různé části inicializace parametrů a zapouzdření výpočetního toku forward.

Následující dvě metody konstrukce sítě jsou ekvivalentní:

```
1. import paddle
2. # Metoda 1: Použití vestavěného obalu vrstvy v rámci f
   rameworku Paddle
3. class LinearNet1(paddle.nn.Layer):
4.     def __init__(self):
5.         super(LinearNet1, self).__init__()
6.         # Použití vestavěné vrstvy Linear v rámci fram
   eworku Paddle
7.         self.linear = paddle.nn.Linear(in_features=3,
   out_features=2)
8.     def forward(self, x):
9.         return self.linear(x)
10.# Metoda 2: Vlastní definice parametrů neuronové sítě
11.class LinearNet2(paddle.nn.Layer):
12.    def __init__(self):
13.        super(LinearNet2, self).__init__()
14.        # Vlastní definice parametrů neuronové sítě
15.        w = self.create_parameter(shape=[3, 2])
16.        b = self.create_parameter(shape=[2], is_bias=T
   rue)
17.        self.add_parameter('w', w)
18.        self.add_parameter('b', b)
19.
20.    def forward(self, x):
21.        x = paddle.matmul(x, self.w)
22.        x = x + self.b
23.        return x
24.if __name__ == "__main__":
25.    model1 = LinearNet1()
26.    model2 = LinearNet2()
27.    print('Informace o struktuře modelu LinearNet1:')
28.    paddle.summary(model1, input_size=(None, 3))
29.    print('Informace o struktuře modelu LinearNet2:')
30.    paddle.summary(model2, input_size=(None, 3))
```

6.3.2 Trénování neuronové sítě

Postup trénování neuronové sítě pomocí frameworku Flying Paddle je následující:

1. Instanciovat modelový objekt model;
2. Změnit model do režimu eval pomocí model.eval();
3. Definovat optimalizátor opt a zadat optimalizační parametry;
4. Zadání dat do smyčky a provedení procesu dopředného výpočtu modelu pro získání výsledku dopředného výstupu;
5. Výpočet ztrátovosti výsledků dopředného výstupu a značek dat;
6. Provedení zpětného šíření pomocí funkce loss.backward() pro výpočet gradientu ztráty vzhledem k parametrům modelu;
7. Jednorázová aktualizace parametrů modelu pomocí funkce opt.step();
8. Vymazání gradientů parametrů modelu pomocí opt.clear_grad();
9. Zpět na 4 a pokračovat v optimalizaci parametrů modelu.

Ukázka kódu je následující:

```
1. def train(epochs: int = 5):
2.     """
3.     Příklad trénovacího procesu
4.
5.     :param epochs: Počet průchodů datovou sadou
6.     :return:
7.     """
8.     # Inicializace modelu
9.     model = LinearNet1()
10.    # Přepnutí do evaluačního režimu
11.    model.eval()
12.
13.    # Definice optimalizátoru
14.    opt = paddle.optimizer.SGD(learning_rate=1e-
   2, parameters=model.parameters())
15.
16.    for epoch in range(epochs):
17.        # Generování náhodných vstupů a štítků
18.        fake_inputs = paddle.randn(shape=(10, 3), dtyp
   e='float32')
19.        fake_labels = paddle.randn(shape=(10, 2), dtyp
   e='float32')
20.
21.        # Přední průchod
22.        output = model(fake_inputs)
23.        # Výpočet ztráty
24.        loss = paddle.nn.functional.mse_loss(output, f
   ake_labels)
```

```
25.
26.         print(f'Epoch:{epoch}, Loss:{loss.numpy()}')
27.
28.         # Zpětná propagace
29.         loss.backward()
30.         # Aktualizace parametrů
31.         opt.step()
32.        # Vymazání gradientů
33.         opt.clear_grad()
```

Výstup je následující:

Epoch:0, Loss:[1.5520184]

Epoch:1, Loss:[1.6992496]

Epoch:2, Loss:[1.9622276]

Epoch:3, Loss:[2.1343968]

Epoch:4, Loss:[1.221286]

6.3.3 Ukládání a načítání modelových vah

Framework Flying Paddle poskytuje velmi jednoduchou a snadno použitelnou implementaci API pro ukládání nebo načítání parametrů modelu během jeho trénování a aplikace. Podrobnosti jsou následující:

Uložení parametrů modelu:

paddle.save(model.state_dict(, 'save_path/model.pdparams')

Načtení parametrů modelu:

state_dikt = paddle.load('save_path/model.pdparams')

model.set_state_dict(state_dict)

6.4 Architektura sítě umělé inteligence pro program

AlphaGo používá síť strategií a síť hodnot, které pomáhají při prohledávání stromu Monte Carlo a snižují hloubku a šířku prohledávání. Strategie hry v této práci je zcela založena na algoritmu Alpha Go s tím rozdílem, že herní deska 19*19 je změněna na desku 9*9.

Při tréninku AlphaGo se používalo 5000 TPU a v našem programu jsou síť strategie a síť hodnot značně zjednodušeny, aby se snížilo množství výpočtů. U našeho programu se k extrakci rysů ze stavu používají 3 konvoluční vrstvy, a to:

- 32kanálová konvoluce s rozměrem 3 X 3 kroku 1(stride = 1);

- 64kanálová konvoluce s rozměrem 3 X 3 kroku 1(stride = 1);
- 128kanálová konvoluce s rozměrem 3 X 3 kroku 1(stride = 1);

V části strategické sítě se nejprve použije 1 X 1 8kanálová konvoluce k integraci informací s kvartálními kanály, poté se připojí plně propojená vrstva ke kompresi rozměru vektoru příznaků na 256 a nakonec se přistoupí k výstupní vrstvě; v části hodnotové sítě se nejprve použije 1 X 1 4kanálová konvoluce k integraci informací s kvartálními kanály, poté se přistoupí ke dvěma plně propojeným vrstvám, a nakonec se přistoupí k výstupní vrstvě.

Konkrétní kód je následující:

```
import paddle
class PolicyValueNet(paddle.nn.Layer):
    def __init__(self, input_channels: int = 10,
                 board_size: int = 9):
        """
        :param input_channels: Počet vstupních kanálů, výchozí hodnota je 10. Posledních 4 tahů obou hráčů, plus jeden kanál pro aktuálního hráče, plus jeden kanál pro poslední tah.
        :param board_size: Velikost herní desky
        """
        super(PolicyValueNet, self).__init__()

        # Architektura sítě AlphaGo : jeden trup, dvě hlavy
        # Část síťové architektury pro extrakci rysů
        self.conv_layer = paddle.nn.Sequential(
            paddle.nn.Conv2D(in_channels=input_channels, out_channels=32, kernel_size=3, padding=1),
            paddle.nn.ReLU(),
            paddle.nn.Conv2D(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            paddle.nn.ReLU(),
            paddle.nn.Conv2D(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            paddle.nn.ReLU()
        )

        # Část sítě pro strategii
        self.policy_layer = paddle.nn.Sequential(
            paddle.nn.Conv2D(in_channels=128, out_channels=8, kernel_size=1),
            paddle.nn.ReLU(),
            paddle.nn.Flatten(),
```

```
            paddle.nn.Linear(in_features=9*9*8, out_features=256),
            paddle.nn.ReLU(),
            paddle.nn.Linear(in_features=256, out_features=board_size*board_size+1),
            paddle.nn.Softmax()
        )
        # Část sítě pro hodnotu
        self.value_layer = paddle.nn.Sequential(
            paddle.nn.Conv2D(in_channels=128, out_channels=4, kernel_size=1),
            paddle.nn.ReLU(),
            paddle.nn.Flatten(),
            paddle.nn.Linear(in_features=9*9*4, out_features=128),
            paddle.nn.ReLU(),
            paddle.nn.Linear(in_features=128, out_features=64),
            paddle.nn.ReLU(),
            paddle.nn.Linear(in_features=64, out_features=1),
            paddle.nn.Tanh()
        )
    def forward(self, x):
        x = self.conv_layer(x)
        policy = self.policy_layer(x)
        value = self.value_layer(x)
        return policy, value
```

6.5 Implementace MCTS

MCTS je prohledávací algoritmus používaný v umělé inteligenci a hrách, který umožňuje efektivně prohledávat stavový prostor her a rozhodovat o nejlepším tahu. Zde jsou některé obecné poznámky o realizaci tohoto algoritmu v programování:

1. Reprezentace hry: Nejdůležitější součástí implementace MCTS je správná reprezentace hry, ve které se algoritmus používá. Hra musí být reprezentována tak, aby bylo možné provádět tahy a vyhodnocovat stav hry.
2. Strom prohledávání: MCTS prohledává stavový prostor hry pomocí stromu, který reprezentuje možné tahy a jejich dopady na stav hry. Každý uzel stromu odpovídá možnému stavu hry a obsahuje informace o počtu návštěv, hodnotě uzlu a dalších důležitých informacích.

3. Selekce, rozšíření a vyhodnocení: Algoritmus MCTS pracuje v cyklu, který zahrnuje výběr nejlepšího uzlu pro prozkoumání (selekce), rozšíření stromu o nové uzly na základě dostupných tahů (rozšíření) a vyhodnocení stavu hry pomocí simulací nebo heuristik (vyhodnocení).
4. Aktualizace hodnot uzlů: Po vyhodnocení stavu hry jsou hodnoty uzlů stromu aktualizovány na základě výsledku simulace. Tím se zlepšuje schopnost algoritmu předvídat nejlepší tahy v budoucnosti.
5. Opakování a vylepšování: MCTS opakovaně prohledává stavový prostor hry a aktualizuje hodnoty uzlů stromu na základě nových informací získaných během prohledávání. Tím se postupně zlepšuje schopnost algoritmu rozhodovat o nejlepších tazích.[19.] [20.]

   Přesná realizace je následující kód:

```
1. class TreeNode:
2.     """Uzel stromu Monte Carlo"""
3.     def __init__(self, parent, prior_p):
4.         self.parent = parent  # Rodičovský uzel
5.         self.children = {} # Ukládání potomků uzlu
6.         self.n_visits = 0  # Počet návštěv uzlu
7.         self.Q = 0  # Průměrná hodnota uzlu
8.         self.U = 0  # Hodnota U pro výběr uzlu v MCTS
9.         self.P = prior_p  #Pravděpodobnost výběru uzlu
10.    def select(self, c_puct):
11.        """Výběr uzlu v MCTS"""
12.        return max(self.children.items(),
13.                   key=lambda act_node: act_node[1].ge
   t_value(c_puct))
14.    def expand(self, action_priors):
15.        """Rozšíření uzlu"""
16.        for action, prob in action_priors:
17.            if action not in self.children:
18.                self.children[action] = TreeNode(self,
    prob)
19.    def update(self, leaf_value):
20.        """Aktualizace hodnot uzlu"""
21.        self.n_visits += 1
22.        self.Q += 1.0 * (leaf_value - self.Q) / self.n
   _visits
23.    def update_recursive(self, leaf_value):
24.        """Rekurzivní aktualizace uzlů"""
25.        if self.parent:
26.            self.parent.update_recursive(-leaf_value)
27.        self.update(leaf_value)
28.    def get_value(self, c_puct):
29.        """Výpočet hodnoty uzlu"""
30.        self.U = c_puct * self.P * np.sqrt(self.parent
   .n_visits) / (1 + self.n_visits)
```

```
31.         return self.Q + self.U
32.     def is_leaf(self):
33.         """Kontrola, zda je uzel listem"""
34.         return self.children == {}
35.     def is_root(self):
36.         """Kontrola, zda je uzel kořenem stromu"""
37.         return self.parent is None
38. class MCTS:
39.     """Hlavní tělo algoritmu Monte Carlo Tree Search""
    "
40.     def __init__(self, policy_value_fn, c_puct=5, n_pl
    ayout=10000):
41.         self.root = TreeNode(None, 1.0)  # Kořenový uz
    el stromu MCTS
42.         self.policy = policy_value_fn
43.         self.c_puct = c_puct
44.         self.n_playout = n_playout
45.
46.     def playout(self, simulate_game_state):
47.         """Simulace hry v MCTS"""
48.         node = self.root
49.         while True:
50.             if node.is_leaf():
51.                 break
52.             action, node = node.select(self.c_puct)
53.             simulate_game_state.step(action)
54.         action_probs, leaf_value = self.policy(simulat
    e_game_state)
55.         end, winner = simulate_game_state.game_ended()
    , simulate_game_state.winner()
56.         if not end:
57.             node.expand(action_probs)
58.         else:
59.             if winner == -1:
60.                 leaf_value = 0.0
61.             else:
62.                 leaf_value = (
63.                     1.0 if winner == simulate_game_sta
    te.turn() else -1.0
64.                 )
65.         node.update_recursive(-leaf_value)
66.
67.     def get_move_probs(self, game, temp=1e-
    3, player=None):
68.         """Získání pravděpodobností tahů"""
69.         for i in range(self.n_playout):
70.             if not player.valid:
71.                 return -1, -1
72.             if player is not None:
73.                 player.speed = (i + 1, self.n_playout)
74.             simulate_game_state = game.game_state_simu
    lator(player.is_selfplay)
```

```
75.             self.playout(simulate_game_state)
76.         act_visits = [(act, node.n_visits)
77.                       for act, node in self.root.child
   ren.items()]
78.         acts, visits = zip(*act_visits)
79.         act_probs = softmax(1.0 / temp * np.log(np.arr
   ay(visits) + 1e-10))
80.         return acts, act_probs
81.
82.     def get_move(self, game, player=None):
83.         """Získání nejlepšího tahu"""
84.         for i in range(self.n_playout):
85.             if not player.valid:
86.                 return -1
87.             if player is not None:
88.                 player.speed = (i + 1, self.n_playout)
89.             game_state = game.game_state_simulator()
90.             self.playout(game_state)
91.         return max(self.root.children.items(), key=lam
   bda act_node: act_node[1].n_visits)[0]
92.     def update_with_move(self, last_move):
93.         """Aktualizace stromu MCTS po tahu"""
94.         if last_move in self.root.children:
95.             self.root = self.root.children[last_move]
96.             self.root.parent = None
97.         else:
98.             self.root = TreeNode(None, 1.0)
```

7 Analýza programu

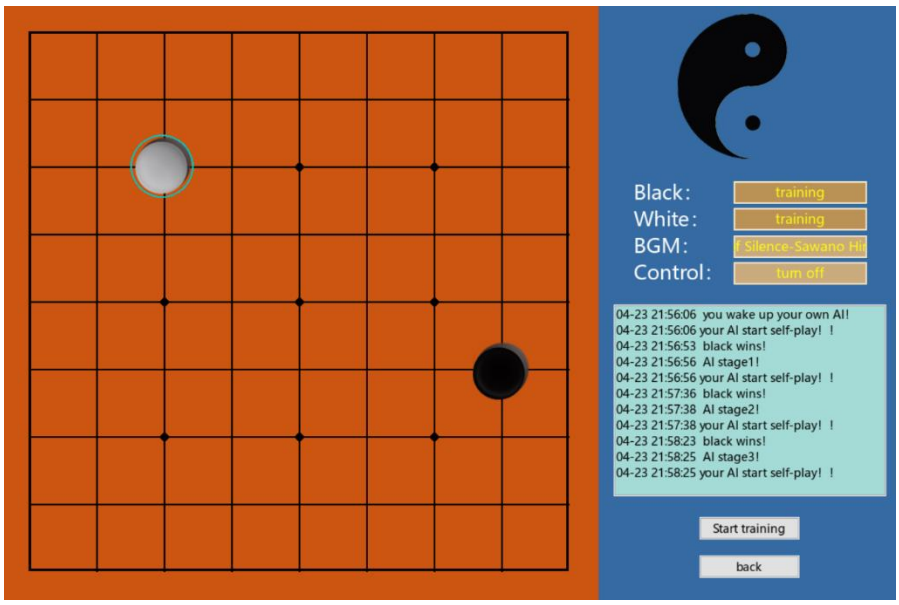
Další částí práce je analýza vytvořeného programu pro hru Go, který nabízí řadu funkcí, včetně simulace pravidel hry Go a možnosti hry dvou lidských hráčů na různě velkých deskách. Program využívá umělou inteligenci MCTS, která napodobuje algoritmus AlphaGo, a udržuje tak stoprocentní úspěšnost vítězství proti robotickému soupeři, který hraje způsobem náhodného položení kamenů, což lze považovat za úspěch. Software navíc umožňuje hráčům trénovat AI na vlastní pěst, což se děje tak, že dva počítačoví hráči vylepšují AI právě pomocí náhodného položení kamenů.

Navzdory významným kladům softwaru má však i některé slabiny. Zaprvé, ani integrované algoritmy, ani ručně trénovaná umělá inteligence nedokáže porazit člověka a proces rozhodování umělé inteligence je poměrně zdlouhavý. Důvodů těchto nedostatků je celá řada, ale hlavní příčina spočívá v samotném zvoleném algoritmu. algoritmus AlphaGo je postaven na MCTS, který sám o sobě vyžaduje extrémně vysoký aritmetický výkon, a to i přesto, že

byl zefektivněn použitím různých metod, jako jsou hluboké neuronové sítě a samoučení atd.. Původní AlphaGo počítal právě pomocí tisíců TPU, a proto jej nebylo možné dokonale reprodukovat na osobním počítači. Ve skutečnosti, i když je nyní mnoho počítačů schopno AlphaGo spustit, skutečné výpočty neprobíhají lokálně, ale prostřednictvím vzdálených serverů v cloudu.

Pokud by bylo třeba tyto problémy v budoucnu řešit, je možné se pokusit optimalizovat stávající algoritmy nebo najít alternativní. Přestože je algoritmus AlphaGo postaven na MCTS, mohou existovat jiné algoritmy, které jsou pro běh na počítačích vhodnější. Očekává se, že úpravou stávajících algoritmů nebo zkoumáním nových algoritmů lze zlepšit efektivitu a výkonnost programu. Zadruhé lze uvažovat o hardwarové akceleraci, která by zvýšila rychlost výpočtu. Osobní počítače se sice nemohou vyrovnat tisícům TPU, ale použití hardwarových akcelerátorů, jako jsou moderní grafické procesory, může být schopno výrazně zvýšit rychlost výpočtů, a tím zkrátit dobu přemýšlení AI.

Je také možné zkusit zadávat výpočetní úlohy ke zpracování na cloudových serverech. Využitím výpočetních zdrojů v cloudu lze výrazně zvýšit výpočetní výkon softwaru, což povede k efektivnějšímu procesu myšlení AI. Tento přístup může softwaru poskytnout výkonnější výpočetní možnosti, ačkoli zvýší určitou latenci sítě.



Obr. 11 Ukázka programu a trénování AI (Zdroj: Vlastní tvorba)

8 Závěr

Tato práce se zabývá vývojem umělé inteligence v různých herních oblastech se zvláštním zaměřením na hry piškvorky (Tic-Tac-Toe), šachy a Go, které jsou předmětem zkoumání této práce. Tyto tři hry nebyly vybrány lehkomyslně, ale velmi promyšleně. Každá z nich představuje nejstarší, nejjednodušší a nejoblíbenější kategorie v historii her, přičemž Go je zároveň nejnáročnějším typem tradiční deskové hry. Proto jsou v této práci detailně popsány různé algoritmy pro hru Go a vytvořen herní software, který tyto algoritmy využívá. Vývoj těchto her v oblasti počítačů představuje nejen pokrok v oblasti her, ale také ztělesňuje neustálý pokrok počítačových algoritmů, hardwaru a softwaru. Každá hra poskytuje jedinečnou cestu úvah pro zkoumání a vývoj algoritmů umělé inteligence a vyžaduje různé přístupy a strategie.

Kromě toho se tato práce snaží podat obsah tak, aby byl srozumitelný i čtenářům, kteří o hře Go nic nevědí, a proto věnuje plnou pozornost výkladu pravidel hry Go. Navzdory skutečnosti, že Go je složitá hra s dlouhou historií a nelineární povahou, článek se snaží začít od základů a strategií, aby čtenář lépe pochopil výzvy, kterým umělá inteligence v této oblasti čelí. Tím, že se práce zabývá historií vývoje hry, výkladem pravidel a zkoumáním algoritmů umělé inteligence, si klade za cíl poskytnout čtenářům komplexní a jasný pohled, který jim pomůže lépe pochopit interakci mezi hrami a umělou inteligencí a také budoucí perspektivy tohoto oboru. Taková komplexní analýza nám pomůže uvědomit si význam umělé inteligence v oblasti her a položí pevné základy pro budoucí výzkum a praxi.

9 Reference

[1.] Františka Orságová (10.10.2017). Kde se vzaly piškvorky a renju? Dostupné na: https://pisqworky.cz/novinky/1851-kde-se-vzaly-piskvorky-a-renju [zobrazeno: 2023-09-15]

[2.] Simmons, Marlene (1975-10-09). "Bertie the Brain programmer heads science council". Ottawa Citizen.

[3.] Schaefer, Steve (2002). "MathRec Solutions (Tic-Tac-Toe)". Mathematical Recreations.

[4.] Shannon C E. Programming a computer for playing chess[J]. Philosophical Magazine, 1950.

[5.] Machines of Loving Grace - John Markoff

[6.] Newborn M. Deep Blue-An artificial intelligence milestone[M].

[7.] John Tromp; Gunnar Farnebäck (2007). "Combinatorics of Go".

[8.] Hsu F H. IBM's Deep Blue chess grandmaster chips[J]. IEEE Computer Society Press, 1999, 19(2):70-81

[9.] Sentego. "history of Go"

[10.] "Go Tournament: Ing Cup". gogameworld.com.

[11.] Silver D, H A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search[J]. Nature, 2016, 529(7587):484-489.

[12.] Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of Go without human knowledge[J]. Nature, 2017, 550(7676):354-359.

[13.] Silver D, Hubert T, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play[J]. Science, 2018, 362(6419):1087-1118.

[14.] Wang, Shusen. "Deep Reinforcement Learning." Stevens Institute of Technology

[15.] Li, Y. "Deep Reinforcement Learning: An Overview." ArXiv preprint arXiv:1701.07274 (2017).

[16.] "An Introduction to Go." The British Go Association.

[17.] huangeddie. "GymGo." GitHub. Dostupné na: https://github.com/huangeddie/GymGo [zobrazeno: 2024-01-23]

[18.] "PaddlePaddle." GitHub. Dostupné na: https://github.com/PaddlePaddle/Paddle [zobrazeno: 2024-01-23]

[19.] Song, Junxiao. "AlphaZero_Gomoku." GitHub. Dostupné na: https://github.com/junxiaosong/AlphaZero_Gomoku [zobrazeno: 2024-01-23]

[20.] C. B. Browne et al., "A Survey of Monte Carlo Tree Search Methods," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, March 2012, doi: 10.1109/TCIAIG.2012.2186810.

[21.] FoxWQ. "Stahování softwaru." Dostupné na: https://www.foxwq.com/soft verze: 1.90.511.8

[22.] TencentWeiqi. "Stahování softwaru." Dostupné na: https://txwq.qq.com/ verze 4.5.04

10 Příloha

Zdrojový kód implementace programu Go je k dispozici v repozitáři na platformě GitHub. Pro spuštění programu je nutné mít nainstalované požadované knihovny a jejich verze, které jsou uvedeny v souboru requirements.txt.

Odkaz na repozitář: https://github.com/HuskySS/BakalarskaPrace

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Jiayuan Hu**
Studium:	I2100209
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Umělá inteligence pro deskové hry**
Název bakalářské práce AJ:	Artificial intelligence for board games

Cíl, metody, literatura, předpoklady:

Cílem této práce je zkoumat aplikace umělé inteligence v oblasti stolních her a představitsouvisející algoritmy. Dále vytvořit program založený na těchto algoritmech.

osnova :

1）Úvod

2）Vývoj umělé inteligence ve stolních hrách

3）Algoritmy pro simulace stolní hry

4) Návrh herního programu

5）Ukázka programu

6） Shrnutí a závěr

Zadávající pracoviště:	Katedra informačních technologií, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Tomáš Nacházel, Ph.D.
Datum zadání závěrečné práce:	15.10.2021