Projekt SUI

Umělá inteligence pro Válku kostek

Dokumentace

Řešitelé

Daniel Konečný (xkonec75)
Filip Jeřábek (xjerab24)
Tomáš Ryšavý (xrysav27)

Datum

30. 12. 2020

Stručný způsob řešení

- Prohledávání stavového prostoru pomocí expectiminimax algoritmu.
- Algoritmus počítání pravděpodobnosti výhry hráčů dle aktuálního rozestavení desky pomocí neuronové sítě.
- Využití pravděpodobnosti výhry jednotlivých hráčů (vyhodnocené neuronovou sítí) a skóre hráčů jako heuristické funkce v expectiminimaxu.

Popis prohledávání stavového prostoru

Implementované třídy

- **`PlayerController`**

  Tato třída se využívá při generování stromu. Její instance je vytvořena při prvním požadavku tahu AI. Činnost spočívá v určování hráčů, kteří budou při simulaci provádět tah. Hráč je určen podle maximálního počtu tahů pro jednoho hráče a jeho pořadí ve hře. Obsahuje jednoduché rozhraní, takže po každém tahu stačí zavolat metodu `i_just_played` (případně `i_skip` - pokud hráč chce skončit kolo předčasně) a následným zavoláním `get_next_player` je možné zjistit, kdo v dalším tahu bude hrát. Bližší popis těchto funkcí:

  `get_next_player(self)`
  Funkce, vrací dalšího hráče v pořadí. Pomocí funkce `get_players_on_board` jsou získáni hráči, kteří ještě mají nějaké pole na desce a seřazeni podle pořadí hráčů. Takže funkce `get_next_player` pracuje pouze s “živými” hráči.

  `i_just_played(self)`
  Po zahrání tahu dá AI vědět kontroleru pomocí této funkce, že právě zahrála jeden tah.

  `i_skip(self)`
  Pokud už AI nemá jak nebo nechce dál hrát, tak voláním této funkce dává na vědomí kontroleru, že již dohrála.

  `should_finish(self)`
  Kontroler vrací true nebo false. Na základě této hodnoty se rekurzivně generují další uzly stromu, nebo se generování ukončí, dané uzly jsou označeny za listy a začne proces výpočtu ohodnocení grafu.

- **`ExpMMNode`**

  Třída představující jeden uzel (či list) ve stromové struktuře algoritmu Expectiminimax

  `simulate_attack(self, from_data, to_data)`
  Při procházení stromem simulujeme jednotlivé tahy všech protihráčů. Funkce tedy přijímá oblast odkud kam má zaútočit a provede simulaci daného tahu jak pro výhru, tak i prohru útočícího hráče. Vrací pak pozměněné hrací desky v obou variantách.

`exp_mm_rec(self, player_controller)`

Rekurzivně volaná funkce, která prochází strom, generuje nové uzly a po rozgenerování všech uzlů a návratu v rekurzi vrací heuristickou hodnotu dané větve. Je zde kladen důraz na pravděpodobnost výhry a prohry, takže jednotlivé cesty v grafu k těmto uzlům jsou váhově ohodnoceny a poté je z nich vypočítán vážený průměr. Pro zmenšení procházeného stromu jsme omezili rozgenerovávání:

a. *Generování uzlu pro možnost výhry a prohry při útoku* - pokud je pravděpodobnost získání daného pole menší, než `lose_rate_treshold`, tak dále danou větev stromu vůbec nerozgenerováváme (je pouze malá pravděpodobnost, že by AI útok vyhrála). Pokud je pravděpodobnost naopak větší než `win_rate_treshold`, tak počítáme s tím, že AI pole získá a nerozgenerováváme variantu prohry. Pokud je pravděpodobnost získání pole mezi těmito hranicemi, pak rozgenerováváme možnost prohry i výhry.
b. *Omezení šířky stromu* - definicí konstanty `max_num_of_turn_first_level` určujeme maximální možný počet následníků prvního uzlu, tedy kořenu stromu. Počet následníků kořenu odpovídá počtu porovnávaných tahů. Definicí druhé konstanty `max_num_of_turn_variants` určujeme maximální počet následníků všech ostatních uzlů ve stromu. Zvyšování hodnoty této konstanty odpovídá polynomiálnímu růstu uzlů ve stromu a tím pádem i polynomiálnímu růstu doby potřebné k provedení predikce.
c. *Omezení hloubky stromu* - další konstantou `max_num_of_turns_per_player` omezujeme počet tahů, které může maximálně AI v jednom kole udělat. Tímto řešením opět snížíme přesnost odhadu ohodnocení tahu, ale také zmenšíme počet generovaných uzlů. Každým přidaným tahem roste složitost výpočtu exponenciálně.

Funkce po dokončení zanoření a návratu rekurze vrací jednotlivým uzlům (z kterých byla tato funkce zavolána) průměr těchto hodnot z jeho následníků. Uzel pak tuto hodnotu předá nadřazenému uzlu a takto je prováděn výpočet a předávání až do doby, kdy je dosažen kořenový uzel. Ten nepočítá průměr hodnot jeho následníků, ale vybere a uloží si následníka, který vrátí nejlepší hodnotu. Tah, kterým byl vygenerován příslušný následovník se provede. Doporučení tahů získáváme pomocí funkce `possible_turns` ve třídě AI, jejichž počet dále omezíme pomocí dříve popsaných možností.

`calculate_heuristic(self, player)`

Pro ohodnocení listů grafu je použita heuristická funkce. Ta využívá skóre hráče normované na interval mezi 0 a 1 a pravděpodobnost výhry hráče na této desce získané z neuronové sítě popsané níže. Nakonec udělá váhovaný součet těchto dvou hodnot a vrátí znovu číslo mezi 0 a 1 značící, jak dobrý daný tah pro hráče je.

- **AI**

Třída vycházející z `dt.ste.py`.

`ai_turn(self, board, nb_moves_this_turn, nb_turns_this_game, time_left)`

Funkce volaná při každém tahu naší AI. Zde se AI chová různě podle toho, kolik má zbývajícího času, prvních pár kol hraje stejně, jako by hrála `dt.ste`. Až dosáhne zbývajícího času 11 sekund (cca po 10ti tazích) začne rozgenerovávat strom. Tuto strategii jsme zvolili z důvodu příliš velkého rozgenerovávaného stromu na začátku hry, kdy nám generování stromu sebralo většinu času a často jsme pak prohrávali na vypršení času.

Maximální počet rozgenerovaných tahů je určen konstantou `max_num_of_turn_first_level`, ve výchozím stavu se rovná 3. Pro každý tah poté rozgenerovává další uzly v grafu podle hranic `lose_rate_treshold` a `win_rate_treshold`, stejně jako ve třídě `ExpMMNode` ve funkci `exp_mm_rec`.

Pokud se dostupný čas zmenší pod 5 sekund omezíme generování stromu na pouhé 2 tahy. Pokud nám na hraní zbývají pouze 3 sekundy opět strom vůbec negenerujeme a AI se chová jako `dt.ste`.

`possible_turns(self, board=None, player_name=None)`

Tato funkce vrací možné tahy aktuálního hráče.
Původní z AI `dt.ste` rozšířili o možnost volání na libovolného hráče nad libovolnou deskou.
Chtěli jsme počítat pravděpodobnost nejen dle pravděpodobnosti útoku a udržení na pole, na které se útočí, ale i pravděpodobností udržení pole, ze kterého se útočí. Ve výsledné verzi však byla AI méně úspěšná.

Vyhodnocení přístupu

Jednotlivé úpravy jsme vždy testovali pomocí hry dvou hráčů. Zde nás zaskočilo, že při volání:
`python3 ./scripts/dicewars-ai-only.py -r -n 1 --ai dt.ste dt.stei`
Záleží na pořadí volaných AI. První má větší pravděpodobnost výhry. Proto jsme naši AI dávali na druhé místo při kontrolování našich výsledků. Tyto “duely” jsme kombinovali s rozsáhlejším testováním v turnajích. Jak již bylo zmíněno, expectiminimax je u této hry v reálném čase možné provádět jen ve velmi omezené míře. Toto tvrzení jsme si ověřili při experimentu, kdy jsme upravili počáteční čas pro AI na několik hodin a při každém provedeném tahu jsme přidali několik minut. Zmírnili jsme omezení pro generování stromu (leč i tak byl strom stále dost omezený) a spustili pomocí předchozího příkazu 40 her o 4 hráčích. Výsledkem bylo, že 40 her trvalo přibližně 8 hodin. Navíc ale naše AI vyhrála pouze 5 ze 40 her. Prohry byly způsobeny byť mírnějším, ale stále nezanedbatelným omezením stromu. Je nutné zde zmínit, že do výsledné (odevzdané) podoby se AI v mnohém změnila.

Slabiny tohoto přístupu

Jelikož počet uzlů v jednotlivých úrovních stromu při simulování všech možností tahů ve hře narůstá polynomiálně (pro různé tahy) a exponenciálně (pro různý počet tahů), je velký problém v reálném čase prohledat potřebné množství stavů pro vytvoření použitelné predikce.
Při testování jsme se snažili nalézt rovnováhu mezi složitostí generování stromu, tedy jeho velikostí, a výslednou úspěšností AI. Bohužel jsme zjistili, že v reálném čase nejsme schopni prohledat dostatečnou oblast grafu aby naše AI byla výrazně lepší než ty referenční. Je však srovnatelná s referenčním řešením STE, které u nás vykazovalo z referenčních řešení nejlepší výsledky. Algoritmus se tedy chová jako expectiminimax se všemi jeho aspekty, jen bylo nutné kvůli složitosti hry ho značně omezit a znehodnotit tím jeho schopnost predikce.

Popis strojového učení

Pro dosažení lepších výsledků jsme se rozhodli modelovat pravděpodobnost výhry jednotlivých hráčů na dané desce. K modelování této pravděpodobnosti jsme zvolili jednoduchou neuronovou síť, kterou jsme natrénovali na vyšších desítkách tisíc různých desek a uložili její architekturu i s vahami pro pozdější jednoduché a rychlé vyhodnocení jakékoliv desky.

Trénovací data

Získání trénovacích dat probíhalo na spuštěném turnaji, kde hrály vždy 4 umělé inteligence proti sobě. Pro trénink jsme využili umělé inteligence, které se používají pro vyhodnocení projektu. Každá hra se uloží do datové struktury `dictionary`, kde jsou uloženy jednotlivé desky v už zpracované a čitelné podobě, a také počet hráčů v dané hře. Každá deska obsahuje informaci o tom, po kolika tazích hra právě je, kdo je na tahu a poté samotné rozložení desky. V rozložení desky je uložena každá provincie, tedy konkrétně její vlastník, počet kostek na ní a její sousední provincie. Takto zpracovaná hra je poté pomocí `pickle` uložena pro následné další zpracování do datové sady.

```
game: {
      players:
      battles: [
            {
                  player: 0/1/2/3
                  turn: 0...x
                  areas: [
                        {
                              owner: 0/1/2/3
                              dices: 1...8
                              adjacent_areas: [x, y,...]
                        },
                        ...
                  ]
            },
            ...
      ]
}
```

Data z jednotlivých her se zpracovávají přímo do datové sady tvořené `NumPy nd-array`. Každý jeden trénovací vzorek tvoří jednorozměrné pole obsahující zpracované informace o desce a stavu hry umístěné po sobě. Jedná se o tyto informace.

- Sousedství provincií - Prvně je vytvořena matice o rozměrech 29x29 (podle celkového počtu provincií na hrací desce) a v té jednička na určitém místě značí sousedství provincií (index ve sloupci a index v řádku jsou jednotlivé provincie). Nula pak značí, že dané provincie spolu nesousedí. Tato matice je diagonálně symetrická, je tedy možné ji omezit pouze na polovinu (v našem případě polovinu nad hlavní diagonálou). Samotnou diagonálu taky nemusíme uvažovat, neboť provincie sama sebe nijak neovlivňuje. Pokud jednotlivé řádky umístíme za sebe (flattening), nepřijdeme o žádnou informaci o desce, ale snížíme dimenzionalitu pro jednodušší zpracování. Výsledkem je celkem 406 hodnot (místo původních 841), které popisují sousedství jednotlivých provincií.

- Vlastnictví provincií - Každou provincii vlastní jeden ze 4 hráčů, tuto informaci můžeme uložit pomocí one-hot encoding do matice 29x4, kterou můžeme také serializovat a tím pádem dosáhnout jednorozměrného pole o 116 hodnotách.
- Počet kostek na provinciích - Jedná se o jednoduché číslo udávající kostek na dané provincii, ve výsledku tedy pole těchto hodnot. Pro lepší reprezentaci těchto dat využíváme převedení na interval mezi 0 a 1 (vydělením maximálním počtem kostek, tedy 8). Dosáhneme tím lepšího trénování vah sítě, kdy jej negativně neovlivňují vysoké hodnoty počtu kostek.
- Počet odehraných tahů - Ve hře je důležité, kolik tahů bylo odehráno, hra například může být teprve na začátku a předpovědět výherce může být složitější, nebo až nereálné. Proto je tato informace také používána k vyhodnocení. Jedná se o číslo mezi 1 a 0 z důvodů uvedených už v předchozích bodě. Normalizovat však takto počet tahů není jednoduché, každá hra trvá různý počet tahů. Rozhodli jsme se tedy použít treshold 50 tahů, po kterých už je automaticky tato hodnota nastavena na 0, jinak se jedná o hodnotu z intervalu 1 až 0.

Výherci jednotlivých her (a tedy cílové hodnoty, tzv. targets) jsou zakódováni pomocí one-hot encoding a přiřazeni vždy ke všem deskám dané hry. V samotném záznamu ze hry není výherce uložen, určí se podle stavu desky v posledním tahu při zpracování do datové sady.

Všechny informace o deskách jsou uloženy do cílového dvourozměrného pole reprezentující celou datovou sadu. Ta je pak zamíchána, aby nezůstaly jednotlivé desky her u sebe. Poté je datová sada rozdělena v poměru 4:1 na trénovací a testovací data. Poté je uložena ve formátu `npy` pro rychlé načtení při tréninku

Neuronová síť určující pravděpodobnost výhry

Architektura neuronové sítě je následující:

Vrstva	**Nastavení**	**Tvar**	**Parametry**
Input		(552)	0
Dense	Uzly: 1024 Aktivační funkce: LeakyReLU	(1024)	566272
Dropout	Míra: 0.4	(1024)	0
Dense	Uzly: 1024 Aktivační funkce: LeakyReLU	(1024)	1049600
Dropout	Míra: 0.4	(1024)	0
Dense	Uzly: 4 Aktivační funkce: SoftMax	(4)	4100

Trénink byl prováděn na datové sadě o 77466 vzorcích a poté bylo provedeno testování na 8608 vzorcích s úspěšností 100 % (loss: 7.9422e-08). K tréninku se používaly batches o velikosti 256 vzorků, jako ztrátová funkce byla použita categorical crossentropy a při tréninku se využívalo optimalizátoru Adam. Data byla trénována na 10 epoch. Po skončení tréninku byl vyexportován a uložen celý model neuronové sítě jako soubor ve formátu h5. To umožní jednoduché a rychlé použití pro pozdější predikci.

Vyhodnocení přístupu

Hlavní slabinou tohoto přístupu je modelování samotné pravděpodobnosti výhry. Dle výsledků, které jsme pozorovali se nejedná o dobrou metriku pro rozhodování o jednotlivých krocích. Často se po několika krocích nezmění pravděpodobnost výhry nijak významně. Model také poměrně často určuje někoho jako jasného vítěze a přitom to dle pohledu na samotnou desku zdaleka jasné není. Po několika tazích mu to však dojde.
Samotné předpovídání výhry však model zvládá poměrně dobře a nakonec se málokdy mýlí, jak je vidět na testovacích datech z datové sady. V celkovém řešení jsme tedy nevyužili pouze informace pouze z této sítě, ale kombinujeme ji s informací o skóre hráčů. V případě nízké předpovídané pravděpodobnosti výhry využíváme pouze skóre, protože by se pravděpodobnost výhry pravděpodobně s jakýmkoliv tahem nezměnila nijak výrazně.

Vyhodnocení řešení

Když teda shrneme jak naše řešení funguje, tak je založené na `dt.ste`. Naše AI bere až 3 doporučené tahy, které by udělalo `dt.ste`, pro každý z nich pak nasimuluje tahy ostatních hráčů, podle výsledného stavu desky pak heuristická funkce spočítá, jak výhodné je pro nás učinit tento tah. Z tahů, ze kterých pak `dt.ste` vybere prostě první vybere nejlépe ohodnocený. Heuristika počítá stav na základě největšího souvislého regionu a výstupu z modelu neuronové sítě, který počítá pravděpodobnosti výhry pro jednotlivé hráče na vstupní desce. Pro efektivnější využití časové dotace pro naši AI prvních pár tahů hrajeme stejně jako by hrála `dt.ste`. Stejně tak pokud již dochází čas.
Naše AI dává velmi srovnatelné výsledky jako `dt.ste`, a to z výpočetních důvodů. Nejsme schopni prohledávat potřebný stavový prostor tak, aby to mělo nějaký lepší výsledek. Kdybychom měli větší výpočetní výkon a více času, tak by byla mnohem přesnější ve výběru tahů.

V průběhu vývoje jsme řešení nespočetněkrát testovali, nejprve na “duelech” pouze s původní `dt.ste` později s dalšími v turnajích, které sice trvaly déle, ale výsledky dávaly objektivnější. Testovali jsme jak celkový poměr vyhraných a prohraných her, tak i samotné chování AI (například zda je někdy naše AI zastavena timeoutem a kolik času zbývá v kterých fázích hry do konce) a podle toho jsme upravovali konstanty, které ovlivňují velikost rozgenerovaného stromu tahů. Testovali jsme i poměr tahů, které AI zahraje stejně jako `dt.ste`, bez toho aby prohledávala strom vůči tahům se simulováním hry. Vychází to zhruba nastejno.

Dodatkové soubory

V odevzdaném archivu jsou ve složce “changed_for_training” soubory, které jsme upravili při implementaci a jsou třeba na reprodukci našeho řešení:

- server/game.py - je doplněný o ukládání desky z každého tahu do souborů pickle, které jsou pak zpracovány do trénovací sady (je třeba vytvořit v kořenovém adresáři projektu složku logs);
- client/ai_driver.py - je doplněný o predikci pravděpodobnosti výhry získané z neuronové sítě po každém tahu nějaké AI.

Pro zpracování datové sady z logů z her stačí spustit soubor dataset.py z odevzdaného balíčku.
Pro natrénování neuronové sítě stačí poté spustit soubor model.py z odevzdaného balíčku.
Samotná trénovací data nejsou součástí

Krátká zpětná vazba

Projekt nás bavil a mrzí nás, že jsme na něj neměli více času, kvůli ostatním předmětům (hlavně kvůli TINu). Původně jsme chtěli naimplemenovat model pomocí reinforcement learningu, ale hra je tak příliš komplexní, že se nám to nepodařilo. Což nás poměrně mrzí, že jsme museli přistoupit k prostému prohledávání stavového prostoru.

Děkujeme za jeden z mála zajímavých projektů na FITu!