# Piškvorkové AI PyŠkvor

Jako téma zápočtového programu jsem si vybral piškvorky. Můj program bude umožňovat uživateli hrát proti piškvorkovému algoritmu PyŠkvorovi. Samotná partie bude probíhat v jednoduchém user-interface vytvořeném pomocí knihovny Pygame. K propočtu variant při hře nebude AI používat klasický minimax, ale upravený Monte Carlo Search. K upřesnění propočtu bude sloužit neuronová síť, která bude určovat na které tahy by se měl propočet zaměřit. Neuronová síť bude “trénována” pomocí supervised learning na odehraných partiích výkonných piškvorkových programů.

## Zadání

Vytvořit program, který uživateli umožní hrát piškvorky na desce velikosti 15\*15. Program by měl hrát alespoň na úrovni hospodského hráče. Budou použita standartní pravidla piškvorek, kde vyhrává pouze řada přesně pěti kamenů.

## Výběr algoritmu

I když jsou piškvorky relativně jednoduchá hra, klasický minimax, který se používá u řady jiných her, si na nich vyláme zuby. Problém je v tom, že v piškvorkách má hráč vždy na výběr z obrovského množství tahů. Nemůžeme projít všechny existující možnosti, protože po pár tazích bude možností příliš. Například pokud hrajeme na desce 15\*15 a chtěli bychom počítat pět tahů dopředu, museli bychom projít 225\*224\*223\*222\*221 = 551417630400 tahů.

Z tohoto důvodu jsem se rozhodl použít variantu Monte Carlo Tree Search algoritmu (dále MCTS). Klasický MCTS je založený na náhodných rolloutech. Z dané pozice se vybírají náhodné tahy dokud partie nějak neskončí. Hodnota dané cesty je pak určena tím, kolikrát naše strana v náhodných rolloutech vyhrála. Tento přístup má ale své mouchy. Například v hrách, kde jeden tah v pozici vyhrává, zatímco veškeré ostatní prohrávají by nám náhodné rollouty mohly vracet velmi nepřesné odhady. Algoritmus by také nevěděl na jaké tahy se zaměřit jako první, takže by stejně musel do určité hloubky projít všechny, čímž se vracíme k problému s minimaxem.

Zde přichází na řadu machine learning a neuronové sítě. MCTS vylepšíme neuronovou sítí, která jako input vezme pozici a vrátí tahy, na které se má propočet zaměřit. Díky tomu stačí algoritmu projít jen zlomek variant a dojde ke stejnému výsledku. Této funkci budeme říkat policy network.

Abychom se zbavili nepřesných rolloutů, vytvoříme další funkci, která nám odhadne hodnotu pozice okamžitě bez rolloutů. Původně jsem měl v plánu použít další neuronovou síť, ale nepodařilo se mi ji vytrénovat, tak aby se chovala slušně a vracela rozumné hodnoty. Proto mi nezbylo nic jiného, než value funkci naprogramovat “ručně” na základě atributů pozice. (Například kolik řad 4 kamenů mají jednotlivé strany).

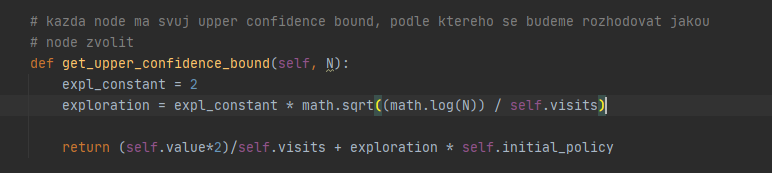
Tato verze MCTS je inspirovaná projektem AlphaZero, což je program, který dosáhl vysoké úrovně v GO, šachách a dalších deskových hrách. [1]

Způsob, jakým MCTS prochází strom variant v tomto programu je výborně vysvětlený v tomto videu: [Monte Carlo Tree Search](https://www.youtube.com/watch?v=UXW2yZndl7U&t=748s)



Upper-confidence bound rovnici uvedenou ve videu jsem ještě rozšířil o output policy funkce.

Uryvek z kodu:



### Machine learning

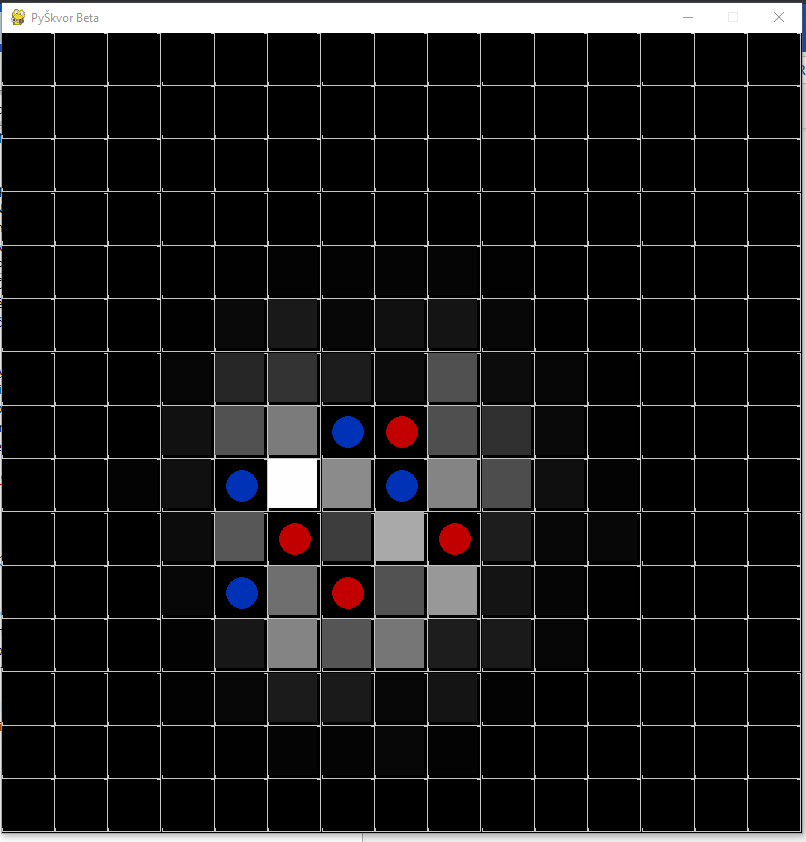
Jak tedy vytrénujeme policy network? V AlphaZero bylo použito deep reinforcement learning, což je velmi zjednodušeně jeden obří pokus-omyl algoritmus. Neuronová síť byla inicializována s náhodnými weights (váha spojů mezi neurony). Další generace sítě byla vytvořena změněním některých weights. Pokud algoritmus používající tuto novou sít porazil algoritmus používající tu starou síť, tak víme, že jsme síť vylepšili a necháme si ji. Pokud vyhraje ten starý algoritmus, novou síť zahodíme a pokusíme se weights upravit jinak. Tímto způsobem může algoritmus dosáhnout takřka perfektní hry, ale k jeho vytrénování je potřeba obrovské množství výpočetní síly. Trénování je tak náročné, že vytrénovat AlphaZero trvalo na superpočítači týdny. Já nemám doma superpočítač, takže tento způsob není vhodný.

Funkci v tomto programu jsem vytrénoval pomocí supervised learning. Modelu jsem ukázal již odehrané piškvorkové partie, ať se z nich naučí hrát. Na internetu jsem našel databázi her z turnaje počítačů obsahující několik tisíc partií. Z partií jsem vytvořil dataset, který se skládá z jednotlivých pozic. Ke každé pozici je přiložený “label”, což je tah, který byl v dané pozici zahrán. Cílem je tedy, aby se naše neuronová sít naučila hrát jako tyto počítače, ale bez jakéhokoliv propočtu.

Podobným způsobem jsem měl v plánu vytrénovat I value funkci, s tím rozdílem, že label pozice není jaký tah byl zahrán, ale jak partie později skončila.

Vizualizace outputu policy funkce:

*Čím světlejší pole, tím více se tah policy funkci zamlouvá*



### Reprezentace pozice

Pozici kamenů na desce jsem se rozhodl reprezentovat pomocí matice. Můj program hraje na 15\*15 desce, takže k reprezentaci je potřeba 15\*15 matice. Kameny začínající hráče jsou reprezentovány jedničkami, kameny druhého hráče minus jedničkami a prázdná pole nulami. K reprezezentování matic v Pythonu jsem použil knihovnu Numpy. Výhodou této reprezentace je, že ji můžeme použít rovnou I jako vstup pro neuronovou síť. (Knihovna Keras vyžaduje vstup v Numpy array). Díky násobení matic pak můžeme pozici jakkoliv otáčet, zrcadlově obracet nebo prohazovat barvy kamenů prostým vynásobením reprezentace minus jedničkou.

Reprezentace tahu

Provezení tahu v piškvorkách je prosté položení kamenu na hrací desku. Žádné již položené kameny se nikdy neodstraňují, což nám ulehčí práci. K provezení tahu je tedy potřeba trojice {řada, sloupec, barva kamene}. V Pythonu provezení tahu vypadá takto:

hraci\_deska[rada][sloupec] = barva\_kamene

### Struktura neuronové sítě

K piškvorkové pozici se můžeme chovat stejně jako k jakémukoliv jinému obrázku. Představme si, že kolečka jsou modré pixely, krížky červené pixely a prázdná pole jsou černá. Tím pádem můžeme policy function trénovat stejným způsobem jako se trénují modely například na rozpoznávání čísel nebo obličejů.

Hledání té nejlepší architektury neuronové sítě je vždy poněkud alchymie. V tomto projektu jsem po mnoha testech různých architektur použil síť, která se skládá ze dvou convolutional vrstev, jedné “klasické” vrstvy o 500 neuronech a outputem do 225 neuronů, kde každý neuron reprezentuje jeden možný tah. (Ve všech pozicích až na prázdnou desku bude míň než 225 možných tahů, takže policy function bude vracet nějaký output I k neproveditelným tahům, takže output musíme později ještě upravit.)

### Static evaluation function

Posláním této funkce je ohodnotit, jak je pozice výhodná pro danou stranu. Jako input bere pozici a jaká strana je na tahu, jako output vrátí číslo mezi 0 a 1, kde 0 je naprosto prohraná pozice, 1 je vyhraná pozice a 0.5 vyrovnaná pozice.

Abychom ohodnotili pozici, rozdělíme ji nejdříve na jednotlivé posloupnosti kamenů – všechny řádky, sloupce a diagonály. V každé posloupnosti potom vyhodnotíme, kolik obsahuje jednotlivých kombinací kamenů. K ohodnocení pozice nás budou zajímat hlavně kombinace 3 a 4 kamenů. Ty dále rozdělíme na kombinace uzavřené cizími kameny z obou stran, uzavřené z jedné strany a otevřené. Dále musíme vzít v potaz I jestli kombinace 4 kamenů nesousedí ob jedno prázdné pole s dalším kamenem té samé barvy, protože vyhrává pouze přesně 5 kamenů v řadě.

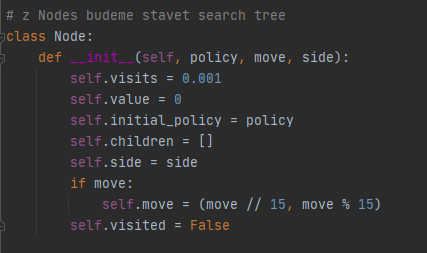
Až najdeme všechny atributy pozice, najdeme rozdíl v počtu kombinací jednotlivých stran. Například pokud mají kolečka o 2 polootevřené kombinace 3 kamenů více než křížky atd. Každému typu kombinace přiřadíme váhu – kombinace 4 neohraničených kamenů bude mít větší váhu než kombinace 3 kamenů. Nyní stačí najít součet rozdílů všech atributů vynásobených jejich váhou, výsledek nacpat do scalovací funkce (já si vybral hyperbolický tan) a dostaneme číslo mezi 0 a 1.

Úryvek z kódu:



## Architektura stromu variant

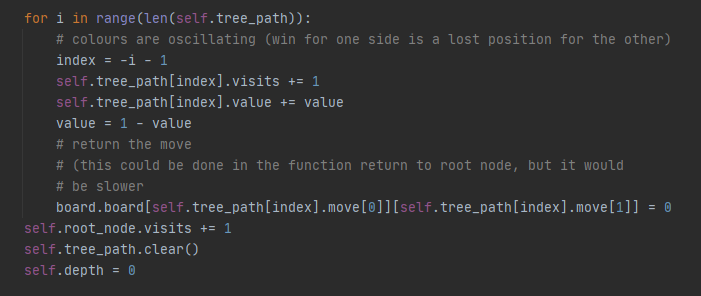
Search tree je sestavován z jednotlivých uzlů (node).



Každý uzel ukládá odkazy na všechny svoje děti, kolikrát jsme ho navštívili během searche a všechny ostatní proměnné potřebně k MCTS. Každý uzel reprezentuje jednu pozici na hrací desce a ukládá tah, kterým se do této pozice dostat z předchozího uzlu.

Uzly neukládají odkaz na svého rodiče, takže kvůli backpropagaci je vždy cesta ve stromě, kterou se search právě vydal uložená v seznamu search\_path.

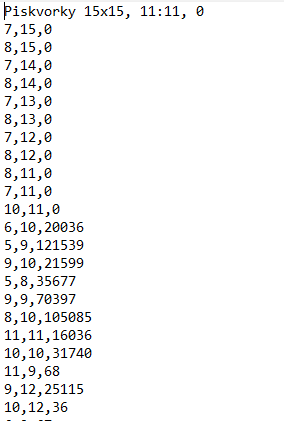
Backpropagace:



### Vytvoření datasetu

K vytvoření datasetu jsem použil volně dostupné partie z piškvorkového turnaje počítačů. <https://gomocup.org/results/> Partie jsou uložené v .psq souborech. Abych z těchto partií dostal dvojice pozic a tahu, který byl v dané pozici zahrán, napsal jsem script create\_dataset.py. Script najde veškeré .psq soubory v daném souboru, partie přehraje a uloží dvě numpy arrays do souborů. První array obsahuje všechny pozice, druhá array všechny příslušné tahy.

*Úryvek z .psq souboru:*



Program

Program se skládá z několika Pythonových souborů. Soubor game\_rules.py obsahuje veškerou piškvorkovou logiku a implementaci Monte Carlo Search. Soubor pygame\_interface.py vytvoří jednoduché prostředí pro hraní piškvorek a umožní zadávání tahů pomocí kliku myší.

K rozběhnutí programu potřebujeme ještě neuronovou sít, která je uložená v samostatné složce, ze které ji při spuštění načteme pomocí knihovny Keras.

K vytvoření neuronové sítě jsou potřeba další dva podprogramy, které ale stačí rozběhnout jednou – nepotřebujeme ke spuštění konečného programu. Create\_dataset.py je na vytvoření vhodného datasetu ze stažených partií a policy\_network.py na vytrénování neuronové sítě.

Konečné řešení se spouští ze souboru pygame\_interface.py. Je možné hrát partii I pouze v konzoli rovnou ze souboru game\_rules.py, ale jednotlivé tahy musíme zadávat textově.

## Alternativní řešení

Namísto MCTS by bylo možné použít upravený Minimax, který by bral v potaz pouze prvních X tahů navrhovaných policy funkcí. Tím by algoritmus nemusel procházet všech 225 tahů, ale pouze jejich zlomek. Při testování jsem ale narazil na problém, kdy policy funkce občas určuje příliš nízkou pravděpodbnost vynuceným tahům. Například pokud má soupeř někde řadu 4 kamenů, musíme ji zablokovat, protože jinak další tah prohrajeme. Tím pádem algoritmus ihned prohraje, protože hrozbu prohry vůbec nezařadí do propočtu. Tento problém nastává hlavně u nestandartních situací – například při vytvoření řady v rohu desky. Je to způsobené tím, že policy funkce byla trénována na počítačových partiích, kde nikdy žádná řada v rohu nevznikla, takže algoritmus neví, jak na danou situaci reagovat. Abychom dostali “chytřejší” policy funkci, museli bychom rozšířit dataset o více nestandartních pozic.

## Testovací partie

Testovací partie s příklady správného fungování jsem nahrál do následujícího youtube playlistu:

https://www.youtube.com/playlist?list=PLJkTG4tYuij2xFWG\_G9Fvp9BIZ\_GOZ2rZ

Informace k jednotlivým testům jsou v názvu a popisu jednotlivých videí.

## Závěr

Při plánování projektu jsem tušil do čeho jdu, ale i tak mi projekt zabral déle, než jsem plánoval. Nejvíc času mi zabralo vytváření evaluační funkce, kterou se mi nepodařilo zprovoznit pomocí machine learning, takže jsem musel spotřebovat další čas a napsat ji ručně. Další oříšek byl debugování Monte Carlo Tree Searche, který sám o sobě není až tak složitý, ale implementovat ho takovým způsobem, aby program hrál na slušné úrovni už rozhodně složité je. Dalším překážka pro silný piškvorkový program napsaný tímto způsobem je rychlost použitého softwaru. Python je pomalejší než jiné jazyky, ale program asi nejvíce zpomaluje knihovna Keras. Její funkce k získání outputu neuronové sítě model.predict() je pro potřeby tree traversalu bohužel velmi pomalá.

I přes to, že můj program prochází jen malý zlomek variant oproti ostatním piškvorkovým programům, jsem spokojený s jeho závěrečnou herní výkonností. Byl jsem překvapen, jak výborně funguje samotná policy funkce bez jakéhokoliv searche. Díky zmiňované vizualizaci jejího outputu jde krásně vidět, že skoro vždy nabízí rozumné tahy. Co je potřeba na programu vylepšit, je value funkce. Napsal jsem ji ručně a nejsem mistr v piškvorkách, takže bere v potaz jen zlomek důležitých atributů pozice. Dále by bylo potřeba najít ideální poměr explorace a exploitace v tree traversalu. To znamená, jestli více prohlédávat do šířky nebo do hloubky a jestli se více řídit policy funkcí nebo value funkcí. Rád bych na těchto věcech zapracoval a do budoucnosti program ještě vylepšil.

Dále tu jsou jednoduché quality of life vylepšení v grafickém interfacu, které bych rád někdy dodělal – aby okno nezamrzalo při propočtu, zadávání délky propočtu přímo v okně, možnost spuštění další partie přímo v okně atd.

I přes to, že PyŠkvor zatím nebude mistr světa v piškvorkách jsem s výsledkem spokojený a troufám si říci, že jsem se naučil hodně nového.

## Reference

[1] Silver, D., Hubert, T. and Julian Schrittwieser, J. (2017). *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. In: DeepMind, London