FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Generování šachových pozic pomocí neuronových sítí
Student:	Vladislav Bobko
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Generování a klasifikace šachových pozic a taktik pro trénink je časově a výpočetně náročnou úlohou vyžadující většinou záznam všech dílčích tahů pro reálné celé hry. Cílem práce je navrhnout a vytvořit framework pro trénování generativních neuronových sítí za účelem generování šachových pozic, bez přesné definice hry a pouze na základě existujících dat.

Kroky:
1. Proveďte rešerši generování šachových pozic, generativních neuronových sítí a jejich využití ve hře šachu.
2. Navrhněte vhodnou reprezentaci dat a připravte vstupní dataset.
3. Navrhněte několik vhodných variant neuronových sítí pro trénovaní na vstupním datasetu.
4. Navrhněte a implementujte framework pro trénování neuronových sítí architektury GAN pro navržené neuronové sítě.
5. Navrhněte, spočítejte a porovnejte metriky jak vstupních, tak i generovaných dat natrénovaných modelů.
6. Proveďte experimenty a vyhodnoťte navržené přístupy.
7. Diskutujte vhodnost a omezení použití GAN architektury pro generování šachových pozic.

Elektronicky schválil/a Ing. Karel Klouda, Ph.D. dne 7. února 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Generování šachových pozic pomocí neuronových sítí

Vladislav Bobko

Katedra aplikované matematiky
Vedoucí práce: Ing. Jaroslav Kuchař, Ph.D.

10. května 2022

Poděkování

Chtěl bych poděkovat Ing. Jaroslavu Kuchaři, Ph. D za odborné vedení a za ochotu a vstřícnost během konzultací poskytnutých k zpracování této práce. Také bych rád poděkoval své rodině za podporu a motivaci k dokončení studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2022 . . . . . .

Odkaz na tuto práci

Bobko, Vladislav. *Generování šachových pozic pomocí neuronových sítí.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Práce se zaměřuje na návrh a implementaci frameworku pro trénování generativních neuronových sítí architektury GAN za účelem generování šachových pozic. Nejdříve jsou navrhnuty zakódování šachových pozic přo převod do maticové podoby. Poté jsou zvoleny základní typy neuronových sítí pro generátor i diskriminátor. Dále jsou navrženy metriky pro porovnání vstupních šachových pozic s vygenerovanými daty a samotný trénovací framework. Navržené modely byly natrénovány na předzpracovaném datasetu ze serveru Lichess.org v jazyce Python v interaktivním prostředí Jupyter Notebooku. V závěru byly provedeny experimenty a vyhodnocena kvalita navrženého přístupu. V posledním experimentu natrénovaný model zapouzdřující navržené základní typy neuronových sítí generuje až 29,7 % validních šachových pozic.

Klíčová slova: generativní kontradiktorní sítě, šachové úlohy, generování pozic, Python

Abstract

The bachelor thesis focuses on design and implementation of GAN architecture training framework for generating chess positions using generative neural networks. First, encodings for input chess positions are designed in order to transform them into a matrix form. Then basic types of neural networks are chosen for both generator and discriminator. Lastly, metrics are designed to compare the input chess positions with generated ones as well as the framework itself. Proposed models were trained on preprocessed dataset from server Lichess.org in Python language in interactive environment of Jupyter Notebook. In the end, experiments were performed and the quality of the proposed approach was evaluated. In the last experiment the trained model encapsulating proposed types of neural networks generates up to 29,7 % valid chess positions.

Keywords: generative adversarial networks, chess puzzles, position generation, Python

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V době koronavirové pandemie vzrostl zájem o šachy díky možnosti hrát libovolný počet a formát partií on-line na webových stránkách, který přetrvává dodnes. Některé servery také nabízí možnost zlepšování dovednosti šachu pomocí řešení šachových úloh. Šachová úloha zobrazuje reálnou šachovou pozici, jehož řešením je určitá sekvence tahů, která vede buď k jasné výhodě, nebo k vyrovnání pozice. Pomocí řešení těchto úloh je tedy možné trénovat rozeznávání podobných situací v reálných hrách.

Generování úloh je ale časově a výpočetně náročné. Každá úloha je generována z reálné hry (ať už zahrané na internetu nebo na desce) a obsahuje přesně jednu variaci, která vede k výhodě. Je také třeba přesně definovat pravidla, která klasifikují a detekují typ úloh.

Téma jsem si zvolil jelikož jsem také začal hrát šachy během pandemie a chyběl mi nástroj, pomocí kterého bych si mohl vygenerovat umělé šachové úlohy pouze na základě podmnožiny příkladů bez specifického popisu.

Hlavním cílem práce je navrhnout a vytvořit framework pro trénování generativních kontradiktorních sítí za účelem generování šachových pozic bez přesné definice hry šachu pouze pomocí existujících dat.

V první kapitole je popsána šachová úloha a náročnost její generace. V následující kapitole jsou představeny generativní neuronové sítě, pozornost je věnována hlavně generativním kontradiktorním sítím a jejich využití ve hře šachu. Třetí kapitola popisuje přípravu vstupních dat, výběr variant neuronových sítí a návrh frameworku a metrik. Čtvrtá kapitola popisuje implementaci a využité technologie. V další kapitole jsou uvedeny metriky natrénovaných modelů a příklady vygenerovaných pozic. V diskuzní kapitole debatuji vhodnost a omezení použití generativních kontradiktorních sítí pro generování šachových pozic. Závěrečná kapitola shrnuje poznatky a popisuje možnosti dalšího vývoje.

Kapitola 1

Zavedení problematiky

1.1 Šachové úlohy

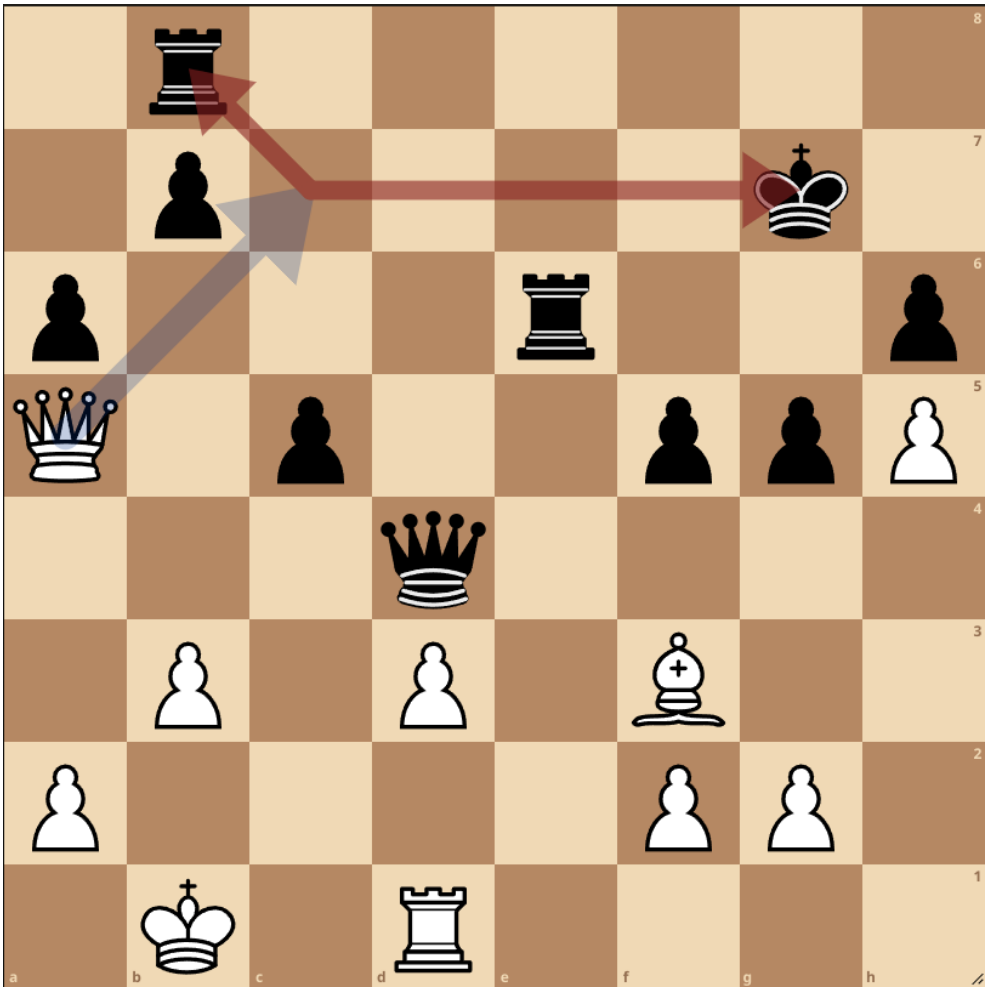
Zadání šachové úlohy zobrazuje validní situaci, ze které je možné zahrát více různých sekvencí tahů. Variace představuje seznam tahů jedné sekvence.

Pozice je strojem ohodnocená číslem ve prospěch jednoho ze dvou hráčů. Ohodnocení pozice je heuristická funkce pro určení relativní hodnoty pozice (tedy šance na výhru). Pokud bychom mohli znát všechny možné ukončené variace této pozice, výsledné ohodnocení by mělo pouze hodnoty -1 (prohra), 0 (remíza) a 1 (výhra). Heuristická funkce tedy určuje relativní hodnotu pozice na základě např. součtu relativních hodnot figurek dané barvy, rozmístění pěšáků, aktivita a koordinace figurek. V současné době existuje mnoho programů pro volby nejlepšího tahu z dané pozice (tzv. „enginy“), kde si každý program definuje vlastní podobu heuristické funkce.

Řešením úlohy se myslí právě jedna taková variace. Tato variace je jednoznačně určená, tedy zahráním jakékoli jiné sekvence tahů vede k jasnému zhoršení ohodnocení pozice z pohledu hráče na tahu. Vyřešením úlohy tedy myslíme nalezení oné sekvence tahů.

1.1.1 Příklad „vidlice“

Jako příklad takové úlohy je typ „vidlice“: situace, v níž jeden hráč útočí na více než jednu figurku soupeře, soupeř ale není schopen ubránit všechny. Obrázek 1.1 znázorňuje příklad úlohy „vidlice“: na tahu je bílý hráč a řešením je tah královnou na políčko **d7**, kde útočí na soupeřova krále a věž, ale soupeř musí reagovat na šach a nedokáže ubránit věž.



Obrázek 1.1: Příklad úlohy, typ „vidlice“ [1]

1.2 Notace

Ve hře šachu se používá v soušasné době několik notací pro zápis pozic, jednotlivých tahů či celých her. Zadání šachové úlohy je zapsáno pomocí kombinací dvou notací, představených v následujících dvou podkapitolách.

Dané notace umožňují snadné zobrazení šachovnic dat - téměř všechny grafické implementace hry šachu umí pracovat s zápisem pozic v těchto notacích.

1.2.1 Forsyth–Edwards Notace

Forsyth-Edwards Notation, zkráceně FEN, popisuje jednu konkrétní pozici šachové hry bez uvedení seznamu tahů, pomocí kterých je možné se do této pozice dostat.

FEN je založená na standardu z 19. století pro záznam pozice navrženého novinářem Davidem Forsythem. Tento standard byl dále mírně rozšířen Stevenem Edwardsem pro použití s počítačovým softwarem.

Záznam FEN obsahuje jeden textový řádek složený ze šesti částí:

1. Rozpoložení figurek,
2. barva hráče na tahu,
3. dostupnost rošády,
4. v případě možného tahu mimochodem jeho cílové políčko, v opačném případě pomlčka,
5. „poloviční tahy“: počet jednotlivých tahů jakéhokoliv hráče od posledního zajetí nebo tahu pěšce,
6. počet dvojtahů (začíná číslem 1 a zvyšuje se po každém tahu černého hráče).

Příkladná startovní pozice v FEN notaci:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ ^ ^^^^ ^ ^ ^
                    1                     2  3   4 5 6
```

1.2.2 Algebraická notace

Algebraická notace zapisuje jednotlivé tahy. V současné době se využívají dvě varianty, plná a zkrácená verze.

Pro názvy figurek jsou použity velká první písmena anglických názvů figurek mimo koně (jelikož začíná na stejné písmeno jako král): K pro krále („king“), Q pro královnu („queen“), R pro věž („rook“), B pro střelce („bishop“), N pro koně („knight“), ale jelikož písmeno K označuje krále, používá se tedy písmeno N), P pro pěšáka („pawn“).

1.2.2.1 Plná verze

Plná verze zápisu obsahuje označní figurky, její počáteční a cílové políčko. Formát tahu tedy vypadá následovně:

```
{figurka}{výchozí políčko}-{cílové políčko}
```

1.2.2.2 Zkrácená verze

Nejpoužívanějším formátem zápisu v současné době je zkrácená algebraická notace („shortened algebraic notation“), která oproti plné verzi neobsahuje výchozí políčko.

Formát zkrácené formy: `{figurka}{cílové políčko}`.

Tento zápis ale může vést k nejednoznačným tahům, například když se dvě figurky stejného typu mohou přesunout na stejné políčko. V tomto případě se za označení figurky do zápisu doplní jednou z následujících možností:

1. Písmeno sloupce výchozího políčka,
2. číslo řádku výchozího políčka,
3. kombinace písmena sloupce a čísla řádku výchozího políčka (pokud ani jedno zvlášť nestačí k jednoznačnému určení figurky).

1.2.3 Zápis úlohy

Zápis úlohy obsahuje výchozí pozici ve Forsyth-Edwards notaci a následnou sekvenci tahů (řešení) v algebraické notaci - ať už zkrácené nebo plné.

Příkladem je zápis úlohy „vidlice“ zobrazené na obrázku 1.1, jejíž řešení je uvedené zkrácené algebraické notaci a spočívá v tahu bílou dámou na políčko C7 a (za předpokladu nejlepšího tahu soupeře, kdy černý hráč táhne králem na políčko F6) následně vzetí černé věže na B8 královnou:

```
1r6/1p4k1/p3r2p/Q1p2ppP/3q4/1P1P1B2/P4PP1/1K1R4 w - - 0 1,
Qc7+ Kf6 Qxb8
```

1.3 Náročnost generace úloh

V současné době se k vytváření úloh přistupuje především manuálně. Pro vygenerování takové úlohy je potřeba strojem ohodnotit všechny pozice z reálné hry a pro každou pozici nalézt všechny možné variace zvolené délky.

Úloha se vygeneruje tehdy, pokud právě jedna variace určité pozice splňuje kritéria řešení.

V praxi se negenerují úlohy s řešením delším než 10 tahů (ačkoliv to je možné) a také je každá úloha zařazena do určité kategorie, která popisuje typ úlohy.

Na webovém serveru Lichess je volně dostupný dataset s více než 2 miliony úloh. Autoři projektu tvrdí, že pro vygenerování takového počtu úloh bylo potřeba více než 35 let procesorového času[4].

1.4 Cíle práce

Hlavním cílem práce je navrhnout a vytvořit framework pro trénování generativních kontradiktorní sítí za účelem generování šachových pozic, trénování základních typů neuronových sítí a porovnání výsledků se vstupním datasetem.

Teoretická část se zaměřuje na popis šachové úlohy/problému, prozkoumání dostupných úloh a analýzu současné metody generování. Dále se věnuje analýze předchozí práci Iana Goodfella[5] a modelu GAN. Dalším cílem teoretické části je analyzovat současné využití GAN a získat přehled o využití v šachu.
Na základě těchto poznatků je možné navrhnout a implementovat framework pro trénování generativních kontradiktorních sítí na základě existující množiny šachových pozic.

Cílem praktické části je navrhnout vhodný framework pro trénování GAN. Dalším cílem je zvolit vhodný formát reprezentace existujících šachových pozic, získání vstupního datasetu a převod datasetu do zvoleného formátu. Dále se zaměřuje na navržení základních typů neuronových sítí a jejich trénování na implementovaném frameworku. Dalším cílem je návrh a implementace metrik pro porovnání reálných dat a dat generovaných pomocí natrénovaných sítí.

Framework má umožnit trénování uživatelsky definovaných sítí na vybrané vstupní množině šachových pozic, vytvoření umělých šachových pozic pomocí natrénované sítě a porovnání se vstupní množinou.

Kapitola 2

Generativní neuronové sítě

Tony Jebara uvádí tři různá paradigmata strojového učení[6]:

1. Generativní - snaží se odhadnout rozdělení pravděpodobnosti přes všechny proměnné,
2. diskriminativní - důležité je pouze konečné mapování ze vstupu x na výstup y,
3. imitativní - pasivní vnímání chování v reálném světě a učení se z něj.

Modely generativního učení nachází v současné době mnoho využití, počínaje augmentací původního datasetu, přes nalezení vhodného zakódování dat za účelem redukce dimenzionality, až po resyntézu obrazů.

Dva takové generativní modely přímo využívají neuronové sítě: tzv. autoenkodéry („autoencoders“) a generativní kontradiktorní sítě („generative adversarial nets“).

V této kapitole podrobněji představím tyto dva modely a uvedu jejich současné využití, pozornost budu věnovat především generativním kontradiktorním síťím.

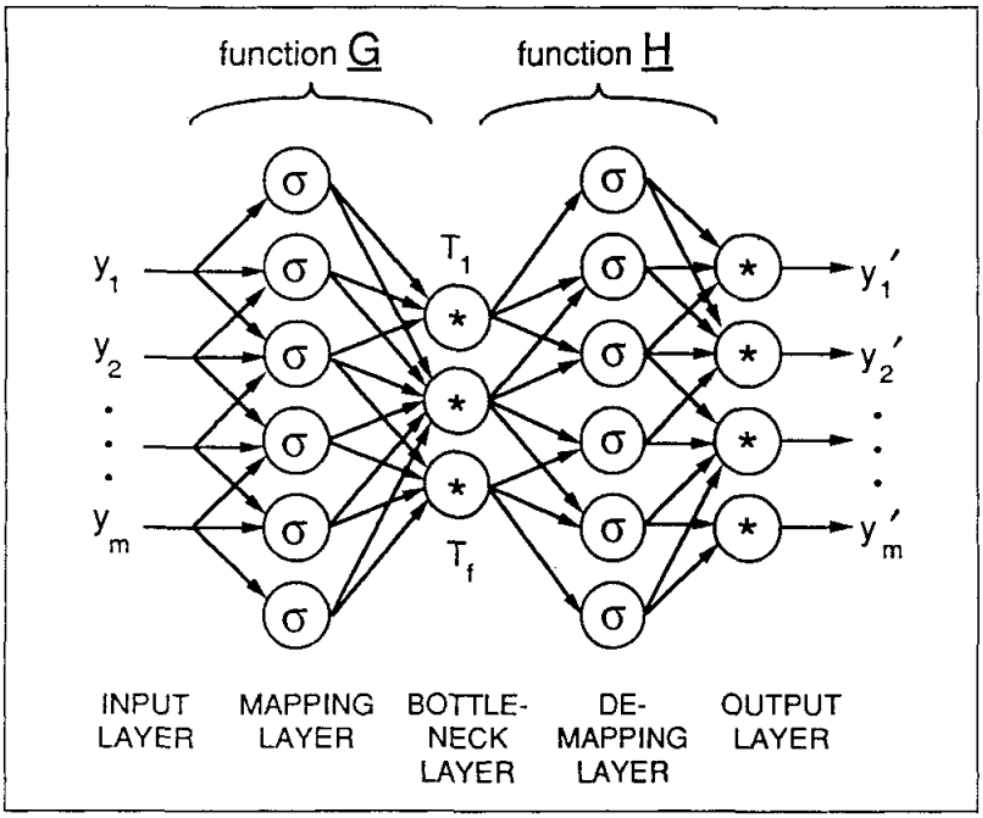
2.1 Autoenkodéry

Autoenkodéry představují typ architektury neuronové sítě, která se snaží najít vlastní reprezentaci vstupních dat. V této kapitole představuji původní návrh autoenkodéru, jeho variační rozšíření a na konec jeho aplikace z pohlednu generativního učení.

2.1.1 Autoenkodér

Dle historického průzkumu Jurgena Schmidhubera[7], autoenkodéry byly navrženy jako metoda pro před-trénování bez učitele v práci „Modular Learning in Neural Networks“[8] Danyho H. Ballarda.

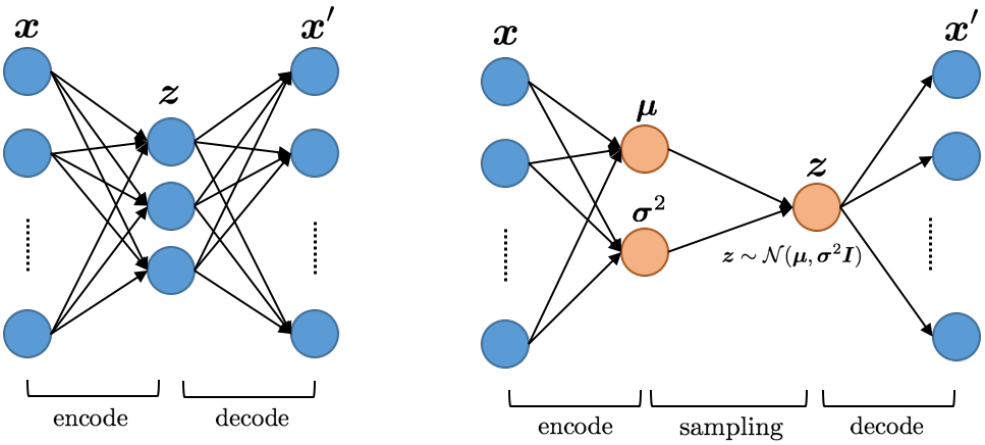
V současné době jako autoenkodér považujeme model nelineární analýzy hlavních komponent („NLCPA“) využívající asociativní neuronové sítě navržené Markem A. Kramerem[2]. Model představuje trénování neuronové sítě pro nalezení identického zobrazení, kde vstupní data jsou reprodukované ve výstupní vrstvě. Síť obsahuje „hrdlo“ („bottleneck“), které nutí síť nalézt kompaktní reprezentaci vstupních dat, což vede k redukci dimenzionality a vytvoření mapy příznaků připomínající skutečné rozložení původních parametrů[2].



Obrázek 2.1: Architektura pro určení f nelineárních příznaků pomocí autoasociativních sítí [2]

2.1.2 Variační enkodéry

Mezi generativní modely řadíme spíše tzv. variační autoenkodéry, poprvé zmíněné Diederikem Kingma a Maxem Wellingem[9], které mohou generovat nová data s podobnými vlastnostmi jako data původní. Oproti původním enkodérům, které se snaží nelézt identické zobrazení, se variační enkodéry učí pravděpodobnostní rozdělení vstupních dat.



Obrázek 2.2: Schéma autoenkodéru (vlevo) oproti schématu variačního autoenkodéru (vpravo)[1]

2.1.3 Využití v současné době

. Ve svém dalším článku[10] Max Welling a Diederik P. Kingma zmiňují další práce obsahující praktické využití variačních autoenkodérů z pohledu generativního paradigmatu strojového učení:

- **Generace přirozeného jazyka**: VAE model je schopný generovat koherentní syntakticky správné nové věty, které je možné vkládat v originální text [11].
- **Astronomie**: generace kalibračních dat jako alternativa k získávání drahých vysoce kvalitních pozorování[12].
- **Resyntéza obrazu**: jednoduchá úprava zakódovaných dat před samotným dekódováním umožňuje sémanticky smysluplné úpravy původního obrazu ke generaci nových obrazů[13].

2.2 Generative adversarial nets

Ian Goodfell spolu s kolegy poprvé popsal Generative Adversarial Nets (v překladu generativní kontradiktorní sítě, dále jen „GAN“) v červnu roku 2014[5]. Jedná se o třídu frameworků strojového učení, která obsahuje dvě neuronové sítě: diskriminační (dále jen „diskriminátor“) a generační síť (dále jen „generátor“).

Základní myšlenka GAN je založená na „nepřímém“ trénování pomocí diskriminátoru: obě sítě v iteracích spolu hrají hru s nulovým součtem. Cíl generátora je vytvořit taková nová data, která diskriminátor klasifikuje jako reálná a naopak cíl diskriminátora je odlišit vygenerovaná data od reálných. Po každé iteraci se obě sítě přizpůsobí na základě výsledků z dané iterace.

V původním článku je hra s nulovým součtem popsána hodnotovou funkcí:

$$V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}}[\log(D(x))] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z))]$$

V další sekci představím binární křížovou entropii a s její pomocí odvodím ztrátové funkce pro generátor i diskriminátor za účelem odvození původní hodnotové funkce.

2.2.1 Křížová entropie

V teorii informace je křížová entropie mezi dvěma rozděleními pravděpodobnosti p a q průměrný počet bitů potřebných pro identifikaci události vybrané z množiny[14]. Běžně se používá ke kvantifikaci rozdílu mezi dvěma rozděleními pravděpodobnosti - v kontextu strojového učení je to míra chyby pro kategorické problémy klasifikace více tříd[15].

Křížová entropie rozdělení q vůči rozdělení p na dané množině je definována následovně[14]:

$$H(p, q) = -\mathbb{E}_p[\log q]$$

Pro diskrétní pravděpodobnostní distribuce p a q[14]:

$$H(p, q) = -\sum_{x \in \mathbf{X}} p(x) \log q(x)$$

2.2.2 Generátor

Cílem generační sítě vygenerovat taková data, která diskriminační síť bude klasifikovat jako reálná.

Cíl generátoru můžeme navrhnout jako maximalizaci chyby špatně klasifikovaných umělých dat:

$$\max_G = \{\text{CHYBA}(1 - D(G(z))\}$$

Při použití křížové entropie jako chyby:

$$\max_G = \{-\mathbb{E}_z[\log(1 - D(G(z)))]\}$$

Původní článek popisuje cíl generátoru jako minimalizaci, kterou můžeme dostat po změně znaménka přechodzí rovnice při použití křížové entropie jako chyby:

$$\min_G = \{\mathbb{E}_z[\log(1 - D(G(z)))]\}$$

2.2.3 Diskriminátor

Cílem diskriminační sítě je rozlišit, zda klasfikovaná data jsou reálná či ne. Intuitivně lze navrhnout problém jako minimalizaci následujících dvou chyb:

- Správně klasifikovaných reálných dat,
- špatně klasifikovaných umělých dat.

Tedy navrhnutá ztrátová funkce má podobu:

$$\mathbb{L}_D = \text{CHYBA}(D(x)) + \text{CHYBA}(1 - D(G(z)))$$

Cíl diskriminátoru je tedy minimalizace navržené ztrátové funkce s použitím křížové entropie jako chyby:

$$\min_D\{-\mathbb{E}_x[\log(D(x))] - \mathbb{E}_z[\log(1 - D(G(z)))]\}$$

Původní článek ovšem popisuje cíl diskriminátoru jako maximalizaci správné klasifikace jak reálných, tak umělých chyb. Pro získání původní rovnice je třeba pouze změnit znaménko:

$$\max_D\{\mathbb{E}_x[\log(D(x))] + \mathbb{E}_z[\log(1 - D(G(z)))]\}$$

2.2.4 Algoritmus

Jinými slovy spolu D a G hrají hru s nulovým součtem se stejnou hodnotovou funkcí, kterou se D snaží maximalizovat a G naopak minimalizovat:

$$\max_D \min_G V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}}[\log(D(x))] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z))]$$

Ian Goodfell se svými kolegy popisuje následující algoritmus trénování modelu[5]:

Algoritmus 1 Trénink modelu architektury GAN

for `počet iterací` **do**
  for k `kroků` **do**
    $\{z^{(1)}, \ldots, z^{(m)}\} \leftarrow m$ syntektických vzorků.
    $\{x^{(1)}, \ldots, x^{(m)}\} \leftarrow m$ vzorků vstupních dat.
    Aktualizuj diskriminátor vzestupem jeho stochastického gradientu:

$$\bigtriangledown_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^i) + \log(1 - D(G(z^i))]$$

  end for
  $\{z^{(1)}, \ldots, z^{(m)}\} \leftarrow$ Vygeneruj m syntektických vzorků.
  Aktualizuj generátor sestupem jeho stochastického gradientu:

$$\bigtriangledown_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z^i))]$$

end for

2.2.5 Využití v současné době

V současné době nachází různé podtřídy architektury GAN využití především v práci s obrazovými daty, například:

- Odstranění deště z fotografií[3],
- automatické detekce glaukomu[16],
- tvorba prostorových dat[17].

Obrázek 2.3: Původní fotografie a výsledné po odstranění deště[3]

Mimo základní variantu GAN existuje řada dalších, kde za zmínku stojí např. Deep Convolution GAN[18] („DCGAN“), Conditional GAN[19](„cGAN“) nebo i Wasserstein GAN[20]. Své využití modely nalezly také i v jiných oblastech, například:

- **Zpracování přirozeného jazyka**: doplňování do textu[21],
- **Časové řady**: generace výstupů senzorů autonomních vozidel[22].

2.2.6 Využití ve hře šachu

Muthuraman Chidambaram a Yanjun Qi ve své práci[23] představují model Style Transfer Generative Adversarial Networks („STGAN“) jako rozšíření architektury GAN: diskriminátor se učí předpovídat, zda byl daný úkol proveden určitým stylem a generátor se učí úkol plnit. V této práci autoři zkoumají aplikaci STGANů na úkolu naučit se hrát šachy ve stylu určitého hráče.

Victor Sim se podobným způsobem pokusil o vytvoření GAN sítě, která by byla schopná hrát šachy stylem jako současný mistr světa Magnus Carlsen[24].

Kapitola 3

Návrh

V této kapitole se věnuji přípravě vstupních datasetů, volby zakódování vstupních dat, navrhuji jednotlivé modely generátoru i diskriminátoru, volím metriky pro porovnání dat a uvádím požadavky na samotný framework.

3.1 Příprava datasetu

Jako zdroj dat jsem si vybral volně dostupný dataset úloh ze serveru Lichess.org obsahující více než 2 miliony úloh. Pro trénování modelů jsem se rozhodl vytvořit dva datasety, které budou obsahovat pouze zápis pozice a řešení vhodné k okamžitému využití (tedy zápis pozice představuje přímo zadání a seznam tahů řešení začíná tahem řešícího hráče):

- Dataset obsahující všechny úlohy z původního datasetu (celkem 2 440 300 úloh),
- dataset obsahující pouze úlohy typu „vidlice“ (celkem 414 700 úloh).

3.1.1 Relativní hodnota kamenů

V současné době se pro ohodnocení šachové pozice využívá různých charakteristik, jako je například aktivita hráče, počet kamenů, jejich umístění nebo také jejich relativní hodnota. Existuje mnoho různých ohodnocení samotných kamenů, nejpoužívanější znázorňuje tabulka 3.1:

Tabulka 3.1: Relativní hodnoty kamenů

Kámen	Relativní hodnota
pěšec	1
střelec	3
jezdec	3
věž	5
dáma	9
král	99

3.1.2 Zakódování kamenů

Jednou z největších výzev této práce bylo nalezení vhodného zakódování samotných šachových kamenů. Hodnoty před samotným trénováním budou přeškálovány na interval $[0, 1]$ a je potřeba zakódovat celkem 13 hodnot (6 bílých, 6 černých a prázdné místo). Intuitivně, zakódované hodnoty by také měly zachovat relativní hodnotu samotných kamenů v rámci barvy, tedy například: absolutní rozdíl hodnoty zakódované královny a pěšáka by měl být větší než absolutní rozdíl střelce a pěšáka stejné barvy.

Na základě požadavků v předchozím odstavci jsem zvolil tři kódování kamenů:

- **Jednoduché** v intervalu $[0, 12]$: kameny bílého hráče od 0 do 5 začínaje králem, kameny černého hráče od 7 do 12 počínaje pěšcem, 13 označuje prázdné políčko.
- **Škálované**: oproti jednoduchému zakódování jsou vzdálenosti mezi jednotlivými typy kamenů zvětšené a přeškálované na interval $[1, 255]$.
- **Inspirované „One-hot“** zakódováním: pro experimenty, které pracují pouze s jedním typem kamene na šachovnici, se kámen zakóduje na interval $[0, 1]$, kde krajní hodnoty označují různou barvu daného kamene a poloviční hodnota 0.5 označuje prázdné políčko.

Tabulka 3.2: Jednoduché a škálované kódování kamenů

Barva	Kámen	Jednoduché	Škálované
bílá	král	1	1
	dáma	2	115
	věž	3	120
	střelec	4	123
	jezdec	5	124
	pěšák	6	127
žádná	volné políčko	7	128
černá	pěšák	8	129
	jezdec	9	131
	střelec	10	132
	věž	11	135
	dáma	12	141
	král	13	255

3.2 Výběr variant neuronových sítí

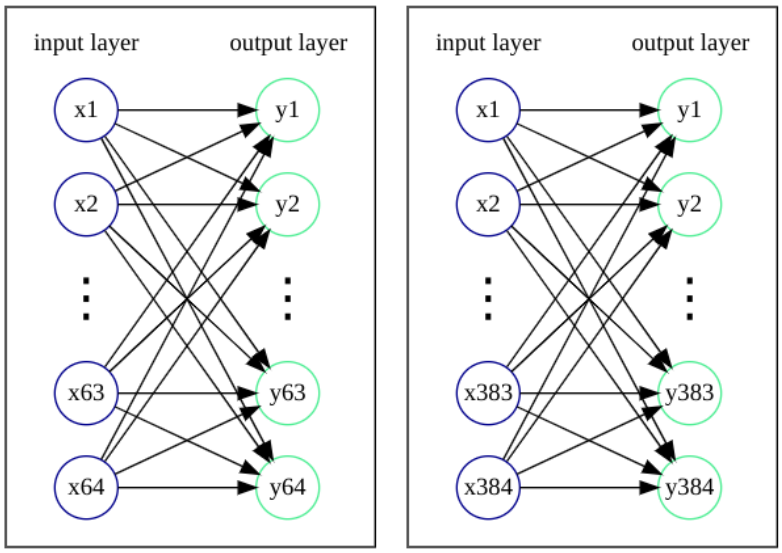
Při výběru variant neuronových sítí jak pro generační, tak i pro diskriminační síť, jsem vycházel především z formátu vstupních dat. Ve své podstatě mám k dispozici vstupní data pouze dvou velikostí:

1. Vektor velikosti 64: pro každé políčko šachovnice jedna hodnota,
2. vektor velikosti 384: pro každý typ kamene samostatná šachovnice (inspirace převzata z modelu RGB - každý „kanál“ popisuje jeden typ kamene, jednotlivé vektory šachovnic jsou následně „spojeny“ za sebe v libovolném, ale pevně zvoleném pořadí).

Navržené modely generátoru a diskriminátoru popisuji v samostatných podsekcích.

3.2.1 Generátor

Struktura modelu generační sítě je z hlediska návrhu jednodušší než struktura sítě diskriminační. Model generátoru bere jako vstup náhodný vektor pevné délky a následně vygeneruje vzorek. V této práci si vystačuji s použitím dvou vrstev, vstupní a výstupní, kde obě mají stejný počet neuronů na základě velikosti vektoru vstupních dat. Využívám tedy dva modely generační sítě lišící se pouze velikostí vstupních a výstupních vektorů:



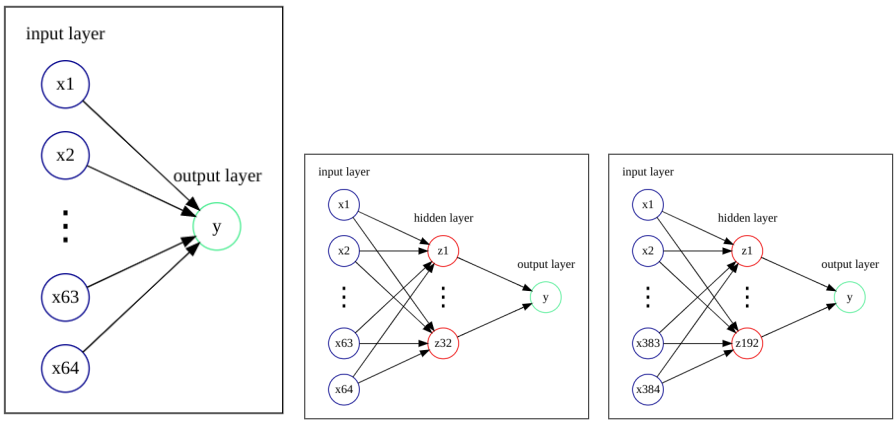
(a) v1 (b) v2

Obrázek 3.1: Navržené modely generátoru

3.2.2 Diskriminátor

Každá navržená diskriminační sít obsahuje minimálně dvě vrstvy: **vstupní** a **výstupní**. Počet neuronů v první vrstě odpovídá velikosti vstupních dat a vzhledem k účelu neuronové sítě výstupem má být vždy skalár, tedy ve výstupní vrstvě bude pouze jeden neuron. Ve své práci využívám pouze jednoho zástupce takové sítě, kde velikost vstupní vrstvy se bude rovnat počtu políček na šachovnici (označen „v1“).

Dále jsem se rozhodl využít neuronové sítě s jednou skrytou vrstvou. Na základě formátu vstupních jsem zvolil dva modely, kde počet neuronů v první vrstvě je 64, resp. 384 a počet neuronů v skryté vrstvě odpovídá polovině velikosti vstupní vrstvy (označeno „v2“, „v3“). Struktury vrstev jsou zobrazeny na obrázku 3.2:



(a) v1 (b) v2 (c) v3

Obrázek 3.2: Navržené modely diskriminátoru

Tento typ neuronové sítě je pouze jeden z mnoha možných variant. Jako možný směr dalšího vývoje se nabízí vyzkoušení jiných, odlišných typů neuronových sítí.

3.3 Metriky

Ve své práci potřebuji metriky, pomocí kterých budu schopen ohodnotit kvalitu vygenerovaných dat a objektivně porovnat se vstupními daty. Hlavním kritériem je determinace, zda je vygenerovaná pozice validní, tedy jestli pozice:

- Obsahuje přesně jednoho černého a jednoho bílého krále,
- ani jeden z králů není v matu,
- nenastala remíza (tato podmínka zajišťuje že pozice obsahuje alespoň jeden další kámen jakékoliv barvy),
- neobsahuje více než 15 kamenů od každé barvy nepočítaje krále,
- počet kamenů jednoho typu nepřesahuje maximum pro daný typ kamene (tedy i při možné proměně pěšce).

Navrhnul jsem následující metriky pro porovnání generovaných dat se vstupními:

- Počet samostatných kamenů zvlášť (počet králů, královen, pěšáků ...),
- počet všech kamenů,
- počet prázdných políček.

Pro každou metriku vypočítávám průměr, medián, horní a dolní kvartil jak pro vstupní, tak i vygenerované šachovnice v celé šarži. Jako dodatečný ukazatel kvality modelu také dopočítávám podíl validních pozic z vygenerované šarže.

3.4 Trénovací framework

Framework by měl být schopen pracovat s různými modely, různými vstupními daty a zároveň poskytnout informace ohledně kvality modelu. Jinými slovy, framework vykonává následující funkce:

- Vytvoří instanci obou modelů,
- převede vstupní data do potřebné podoby,
- natrénuje instance modelů na upravné podobně vstupních dat,
- napočítá metriky generátoru,
- vrátí natrénované instance modelů.

Kapitola 4

Implementace

V této kapituje se věnuji popisu implementační části práce. Představím použité nástroje a blíže popíši získání datasetu a jeho transformace, třídy modelů a metrik, pomocných funkcí a samotný framework.

4.1 Použité nástroje

Jako platformu vývoje jsem si zvolil jazyk Python v prostředí interaktivního vývoje Jupyter Notebooku, jelikož pro tento jazyk existuje velké množství knihoven/algoritmů pro strojové učení. Zároveň také existuje mnoho podpůrných knihoven pro práci se hrou šachu. Ve své práci používám konkrétně tyto knihovny:

- **python-chess**[25] pro zobrazení šachovnice a práce s ní,
- **prettytable**[26] pro formátovaný výstup ve formě tabulky,
- **pandas**[27] pro práci s datasetem,
- **numpy**[28] pro pomocné numerické výpočty,
- **multiprocessing** pro paralelní výpočty,
- **PyTorch**[29] pro existujicí implementace modelů.

4.2 Pomocné funkce a třídy

Pro zjednodušení práce s daty jsem si vytvořil dvě pomocné funkce pro rozhodnutí, zda je šachová pozice a zápis validní či ne a třídu poskytující převod šachovnice ve formátu FEN a opačně.

4.2.1 Detekce platnosti

Funkce jsem pojmenoval **valid_fen** a **valid_board**, kde první jako parametr příjmá zápis FEN a druhá pouze první část zápisu popisující samotnou pozici. První funkce kontroluje samotný zápis a využívá druhou funkci, která implementuje podmínky zmíněné ve sekci 3.4, kde poslední maximální počty pro každý kámen popisuje tabulka 4.1(jedná se o počet kamenů jednoho druhu a počet možných nahrazení všech pěšáků při postupu na poslední pole desky):

Tabulka 4.1: Maximální počet kamenů

Kámen	Maximální počet
král	2
dáma	9
věž	10
střelec	10
jezdec	10
pěšák	8

4.2.2 Detekce „vidlic“

Pro detekci detekci jsem vytvořil několik pomocných funkcí:

- „**kámen bráněn**“: jestli existuje nějaký další kámen, který útočí na políčko „bráněného“ kamene,
- **kámen může být zabrán soupeřovým kamenem s menší hodnotou**: jestli existuje alespoň jeden soupeřův kámen s menší relativní hodnotou útočící na políčko kamene hráče na tahu,
- **napadnuté soupeřovy kameny**: vrací seznam dvojic všech napadnutých soupeřových kamenů a jejich políček,
- **kámen je špatném políčku**: funkce detekuje zda je kámen na daném políčku nebráněn a zároveň nějaký soupeřův kámen s menší relativní hodnotou útočí na toto políčko.

Proces detekce vidlice znázorňuje algoritmus 2

Algoritmus 2 Detekce vidlice v zadané pozici

Vstup: šachová pozice S, barva hráče na tahu s, zobrazení typu kamenů na relativní hodnoty h
Výstup: True/False jestli pozice obsahuje vidlici z pohledu hráče na tahu
Název funkce: „contains_fork“

```
T ← všechny legální tahy hráče s z pozice S
for t in T do A ← tažený kámen
    if t proveden na špatné políčko then
        continue                                   ▷ přeskoč tento tah
    end if

    počet cílů ← 0
    C ← napadnuté soupeřovy kameny
    for c in C do                                  ▷ pro každý napadnutý kámen
        if c je pěšec then
            continue                               ▷ ignorujeme pěšce
        end if
        if h(A) < h(c) or                          ▷ hod. útočícího < hod. napadnutého
            (
              c není bráněn and
              c nemůže napadnout A
            ),  then
            počet cílů ← počet cílů +1
        endif
    end for

    if počet cílů > 1 then
        return true                                ▷ šachovnice obsahuje vidlici
    end if
end for
return false                                       ▷ šachovnice neobsahuje vidlici
```

4.2.3 Transformační třída

Pro ulehčení práce s formátem dat jsem pro převod zvolil implementaci formou třídy. Samotná třída poskytuje metody převodu zápisu šachovnice ve formátu FEN a zpět, kde převod z matice do FEN zápisu poskytuje pouze první část (tedy pouze kameny a jejich pozice). V základní variantě třída obsahuje pouze jeden argument - zakódování - a dvě metody pro převod (s pomocnými metodami pro převod), **from_matrix** a **to_matrix**. Samotné dekódování nejdřív přeškáluje hodnoty zpět na původní interval (jelikož zakódovaná data jsou přeškálovaná na interval $[0, 1]$) a poté vybere kámen, jehož hodnota je k přeškálované nejblíže.

Pokud by například při použití škálovaného kódování byla přeškálovaná hodnota rovna 117, tak hodnota leží mezi kódovanými hodnotami bílé dámy a věže (viz. tabulka 3.2). Jelikož ale zakódovaná hodnota dámy (115) blíže, než hodnota věže (120), je dekódovaná hodnota označena jako bílá dáma.

V případě, kdy je velikost vektoru rovna 384 (samostatná pozice pro každý typ kamene), se nejprve inicializuje prázdná pozice a poté se prochází jednotlivé části šachovnice a na pozici výsledné šachovnice se vypíše kámen s větší relativní hodnotou. Rozšířenou variantu třídy popisuji v sekci 5.3.

4.3 Příprava datasetu

Dataset vstupních úloh jsem získal z webového serveru Lichess.org, který umožňuje stažení jak celé databáze her, tak i celé databáze šachových problémů zdarma. V době vývoje obsahovala databáze okolo 2 miliónů úloh. Každá pozice v datasetu je uvedená ve formátu Forsyth-Edwards notace.

4.3.1 Předzpracování datasetu

Dataset je možné stáhnout ve formátu zakomprimované csv tabulky, která obsahuje následující soupce:

1. Identifikátor úlohy,
2. zápis **pozice** ve Forsyth-Edwards notaci,
3. řešení úlohy jako **seznam tahů** v plné algebraické notaci,
4. odchylka hodnocení,
5. popularita,
6. počet pokusů o vyřešení dané úlohy,
7. typ úlohy,
8. odkaz na reálnou hru, ze které úloha pochází.

Pro potřeby této práce jsou důležité pouze sloupce se zápisem pozice a řešením úlohy. Zápis pozice ovšem není vhodný k okamžitému využití: na stránce datasetu je přitomné upozornění, že zápis představuje pozici **před tahem soupeře** (tedy před předpokládanou chybou) a seznam tahů začíná tahem soupeře.

Dataset je tedy potřeba předzpracovat: pro každou úlohu je třeba ze zadané pozice zahrát první (soupeřův tah), vygenerovat z zápis ve formátu FEN a ze seznamu řešení smazat první tah.

Samotné předzpracování probíhá paralelně dle zvoleného počtu paralelních procesů (v samotném notebooku detekuji počet jader procesoru a využívám o jedno méně procesů, než je jader). Při prvním postupu jsem rozděloval dataset na stejné velikosti pro zvolený počet jader a spouštěl předzpracování na každou část zvlášť v jednotlivém procesu. Tento postup ovšem není ideální, jelikož každý proces si musí držet v paměti přidělenou část původního datasetu a nově vytvořené záznamy, což vede ke značnémů zpomalení zpracování.

Značného zrychlení jsem dosáhnul iterativním opakováním předchozího postupu:

Algoritmus 3 Iterativní paralelní předzpracování datasetu

Vstup: dataset X, počet paralelních procesů p, počet iterací t
Výstup: Předzpracovaný dataset X'
Název funkce: „preprocess_dataset“

$X' \leftarrow \{\}$ ▷ výsledný předzpracovaný dataset
$D \leftarrow [X_i]_{i=1}^{t}$ ▷ rozděl původní dataset na t kusů
for i **in range** (t) **do** ▷ pro každou iteraci
  $Z_i \leftarrow \{\}$ ▷ předzpracovaná i-tá část datasetu
  $K \leftarrow [D_{i_j}]_{j=1}^{p}$ ▷ rozděl i-tý kus datasetu na p částí
  for j **in range** (p) **do**
    $R_j \leftarrow$předzpracuj(K_j) ▷ spusť proces předzpracování
    $Z_i = Z_i \cup \{R_j\}$ ▷ ulož výsledek k výsledkům iterace
  end for
  $X' = X' \cup \{Z_i\}$ ▷ ulož výsledky iterace
end for
return X'

S pomocí iterativního postupu jsem na procesoru Intel i5-4570 získal přibližně pětinásobné zrychlení ($\approx$ 1 hodina oproti $\approx$ 5 hodin).

4.3.2 Třídy reprezentace

Knihovna PyTorch nabízí (a pro určité vnitřní implementace vyžaduje) dvě rozhraní reprezentace a práce s datasetem: **torch.utils.data.Dataset** a **torch.utils.data.DataLoader**. Třída Dataset představuje třídu pro uložení vzorků a Dataloader obaluje třídu Dataset pro umožnění snadného a iterabilního přístupu k datům. V této práci je tento přístup velmi výhodný, jelikož je žádoucí nechat už zpracovaný dataset netknutý a zároveň každý model pracuje s jinou velikostí vstupních dat. Pro tyto potřeby jsem implementoval funkci **instantiate-dataset-loader** 4:

Algoritmus 4 Vytvoření instance DataLoaderu

Vstup: dataset X, počet paralelních procesů k, velikost šarže n, instance transformátoru T
Výstup: instance třídy **torch.utils.data.DataLoader**
Název funkce: „instantiate_dataset_loader“

$D \leftarrow [X_i]_{i=1}^{k}$ ▷ rozděl původní dataset na k kusů
for i **in range** (k) **do**
  $p_i \leftarrow$ paralelní proces s D_i a T ▷ generace maticových reprezentací
  $M_i \leftarrow$ výsledek procesu p_i
end for
$M = \bigcup_{i=1}^{k} M_i$ ▷ spoj všechny výsledky
$S \leftarrow$ vytvoř instanci třídy **torch.utils.data.Dataset** nad M s n
$L \leftarrow$ vytvoř instanci třídy **torch.utils.data.DataLoader** nad S
return L

4.4 Modely

Knihovna **PyTorch** obsahuje rozhraní pro vytvoření vlastních modelů - tyto modely musí být vytvořené jako samostatné třídy dědící od třídy **nn.Module**. Tato třída využívá náhradní optimalizační algoritmus pro stochastický gradientní sestup s hyperparametry **míry učení** a **beta1** a jako aktivační funkci využívá sigmoid. Samotná vnitřní struktura modelů je definována pomocí existujících implementací v knihovně a je znázorněná v tabulce 4.2:

Tabulka 4.2: Příkazy použité k definici vnitřních struktur sítí

Název sítě	Struktura
D_v1	nn.Linear(64, 1)
D_v2	nn.Sequential (nn.Linear(64, 32), nn.ReLU(), nn.Linear(32, 1))
D_v3	nn.Sequential (nn.Linear (384, 192), nn.ReLU(), nn.Linear(192, 1))
G_v1	nn.Linear(64, 64)
G_v2	nn.Linear(384, 384)

Pro usnadnění práce s vytvářením modelů jsem vytvořil funkci „CreateModels“ pro inicializaci modelů generátoru a diskriminátoru, která jako argumenty příjmá názvy tříd a vrací dvojici instancí těchto tříd.

4.5 Třída metrik

Výpočet jednotlivých metrik jsem realizoval jako samostatnou třídu, která ve své členské proměnné typu pole obsahuje odkazy na samostatné metriky implementované jako vnitří metody této třídy. Konstruktor třídy příjmá čtyři parametry: instance generátoru, transformátoru, dataloaderu a počet iterací. Výpočet všech metrik je prováděn ve vnitřní mětodě, která provádí výpočty jednotlivých metrik obsažených v poli. Výpočet je znázorněn algoritmem 5:

Algoritmus 5 Výpočet metrik

Vstup: dataset D, generátoru G a transformátoru T, počet iterací t, velikost šarže n
Výstup: tabulka vypočtených metrik pro původní a vygenerovaná data
Název metody: „metricize“

$R = \{\}$ ▷ výsledky všech metrik
for i **in range** (t) **do**
  $R_i = \{\}$ ▷ výsledky metrik této iterace
  $X = [x_i]_{i=1}^{n} \leftarrow n$ vzorků původních dat z D
  $Z = [z_i]_{i=1}^{n} \leftarrow n$ vygenerovaných dat pomocí G,
  Převeď Z do FEN notace pomocí T

  for každou metriku **do**
    $T_x \leftarrow$ výpočet dané metriky pro X
    $T_z \leftarrow$ výpočet dané metriky pro Z

    $R_m \leftarrow$ průměr, medián, horní a dolní kvartil pro T_x a T_z
    $R_i = R_i \cup \{R_m\}$ ▷ ulož výsledek k výsledkům iterace
  end for
  $R = R \cup \{R_i\}$ ▷ ulož výsledky iterace
end for
return R

4.6 Trénovací framework

Samotná kostra trénování jedné sítě byla inspirována tutoriálem ze serveru PyTorch[30], kterou jsem následně zakomponoval do funkce testování více sítí přizpůsobené pro vstupní dataset šachových pozic.

Vypočtené výsledky následně kompiluji do tabulky pomocí knihovny PrettyTable[26] a vypisuji do složky **metrics** s názvem verze diskriminátoru, generátoru.

Trénování jedné dvojice sítí je implementováno funkcí **train**, která byla inspirována návodem ze serveru PyTorch, ale upravená pro potřeby této práce. Funkce příjmá počet epoch, velikost vstupního vektoru a instance tříd generátoru, diskriminátoru a DataLoaderu. Následně provádí samotné strojové učení a výpočet chyby diskriminátoru a generátoru a vrací dvě dvojice:

- Naučený model generátoru G a jeho chyby L_g,
- naučený model diskriminátoru D a jeho chyby L_d.

Výsledný trénovací framework zobrazuje algoritmus 6:

Algoritmus 6 Framework pro trénování neuronových sítí

Vstup: vstupní dataset D, název tříd generátoru g a diskriminátoru d, instance transformátoru T, počet epoch e a velikost šarže n.
Výstup: Dvojice nautrénovaných modelů generátoru a diskriminátoru
Název funkce: „testNetwork“

▷ inicializuj třídy generátoru a diskriminátoru (viz. sekce4.4)
$G, D \leftarrow$ CreateModels (g, d)

▷ vytvoř instanci třídy DataLoaderu (viz. alg. 4)
$L \leftarrow$ instantiate_dataset_loader $(D, \ldots, n, T)$

▷ trénuj modely na vybraném datasetu po dané epochy
$G, L_g, D, L_d \leftarrow$ **train**(G, D, L, e)

▷ napočítej metriky naučeného generátoru a původních dat,
▷ viz. algoritmus 5
$M \leftarrow$ metricize(L, D, T, n)

Zanes do grafu vývoj chyb L_g, L_d a ulož do souboru
Ulož M do souboru
return G, D

Kapitola 5

Experimenty

V této kapitole provádím trénování navržených modelů nad předzpracovanými datasety a provádím experimenty. Nejprve implementuji pomocné funkce, poté trénuji zvolené modely na původním datasetu. Následně trénuji navržené modely na datasetu šachovnic obsahující pouze jeden typ kamene a poté skládám výsledky do jedné finální šachovnice. V poslední sekci představuji nové zapouzdřené modely diskriminátoru a generátoru, trénuji je na obou navržených datasetech a uvádím příkladné výstupy.

5.1 Pomocné funkce

Pro potřeby experimentů jsem dodatečně implementoval tři pomocné funkce:

- **show_board**: grafické zobrazení pozice specifikovanou argumentem v zápisu první části FEN pomocí knihovny python-chess[25].
- **generate_board**: funkce jako argumenty příjmá model generátoru a instanci transformátoru a vrací vygenerovanou pozici ve formátu první části FEN zápisu.
- **generate_until**: snaží se vygenerovat jednu validní pozici, než dosáhne limitu (který je argumentem funkce společně s identickými argumenty funkce předchozí). Jakmile narazí na validní pozici, ihned ji vrací s vypsáním počtu proběhlých iterací před první validní pozicí, v opačném případě pouze informuje o neúspěchu.

5.2 Učení navržených sítí na původním datasetu

Cílem prvního experimentu je zjistit nejlepší vhodnou kombinaci dvou sítí generátoru a diskriminátoru a kódování dat. Pro tento experiment byl použit pouze původní dataset. Výsledky experimentu nejsou nijak ohromující: z tabulky 5.1, na nímž jsou uvedeny příkladné vygenerované pozice pro každou kombinaci, je vidět, že u modelů nedochází k výraznému učení.

Tabulka 5.1: Příklady vygenerovaných pozic z prvního experimentu

model		kódování	
G.	D.	normální	škálované
v1	v1		
v1	v2		
v2	v3		

Lze si ale všimnout dvou skutečností: kombinace generátoru „v1“ a diskriminátoru verze „v1“ a „v2“ v normálním zakódování obsahuje značně více prázdných políček, avšak neobsahují žádné krále. Zato kombinace generátoru „v1“ a diskrimátoru „v2“ ve škálovaném zakódování obsahuje **přesně** dva

krále, bílého a černého. Pro přesnější pohled jsem z napočítaných metrik zkompiloval tabulku 5.2, která pro model kombinace generátoru „v1“ a diskriminátoru „v2“ uvádí počet všech králů ze 128 vzorků vygenerovaných pozic v pěti iteracích (v každé reálné pozici je právě jeden černý a jeden bílý král, tedy ideálně by mělo být 128 králů od každé barvy):

Tabulka 5.2: Porovnání vybraných metrik na modelu G_v1 a D_v2

Iterace	Počet králů	Kódování	
		normální	škálované
1	bílých	11	36
	černých	0	104
2	bílých	18	50
	černých	0	84
3	bílých	20	51
	černých	0	85
4	bílých	10	48
	černých	0	95
5	bílých	15	53
	černých	0	84

Počet králů považuji za mnohem významnější ukazatel kvality modelu než rozložení a počet ostatních figurek. Metrika svědčí o schopnosti modelu vytěžit znalost, že každá reálná šachovnice obsahuje **vždy** dva krále různé barvy. Z tohoto předpokladu usuzuji, že „normální“ zakódování není velmi vhodné - modely používající toto kódování nejsou schopny si s učením poradit, v následujících experimentech tento typ vynechávám a využívám pouze škálované kódování.

Příkladná šachovnice a napočítané metriky vedou k závěru, že si tento model může dokázat poradit pouze s jedním typem kamene, což slouží jako předpoklad a zadání k následujícímu experimentu.

5.3 Samostatná síť pro jeden typ kamene

V tomto experimentu se pokouším natrénovat kombinaci „G_v1“ a „D_v2“ na původním datasetu pro každý samostatný typ kamene, tentokrát ale s využitím inspirovaného „one-hot“ zakódování. Pro tento experiment je za potřebí buď upravit dataset tak, aby obsahoval pouze jeden typ kamene (a tedy i duplikovat dataset) nebo upravit samotný proces kódování dat. Rozhodl jsem se pro úpravu třídy transformátoru: do konstruktoru třídy přidávám seznam vyloučených typů kamenů v jedné barvě. Při vytváření instance třídy se kódovací i dekódovací tabulky upraví tak, aby vyloučené kameny pro obě barvy měly stejné hodnoty jako prázdné políčko, čímž dojde k samotnému vyloučení těchto kamenů z pozic.

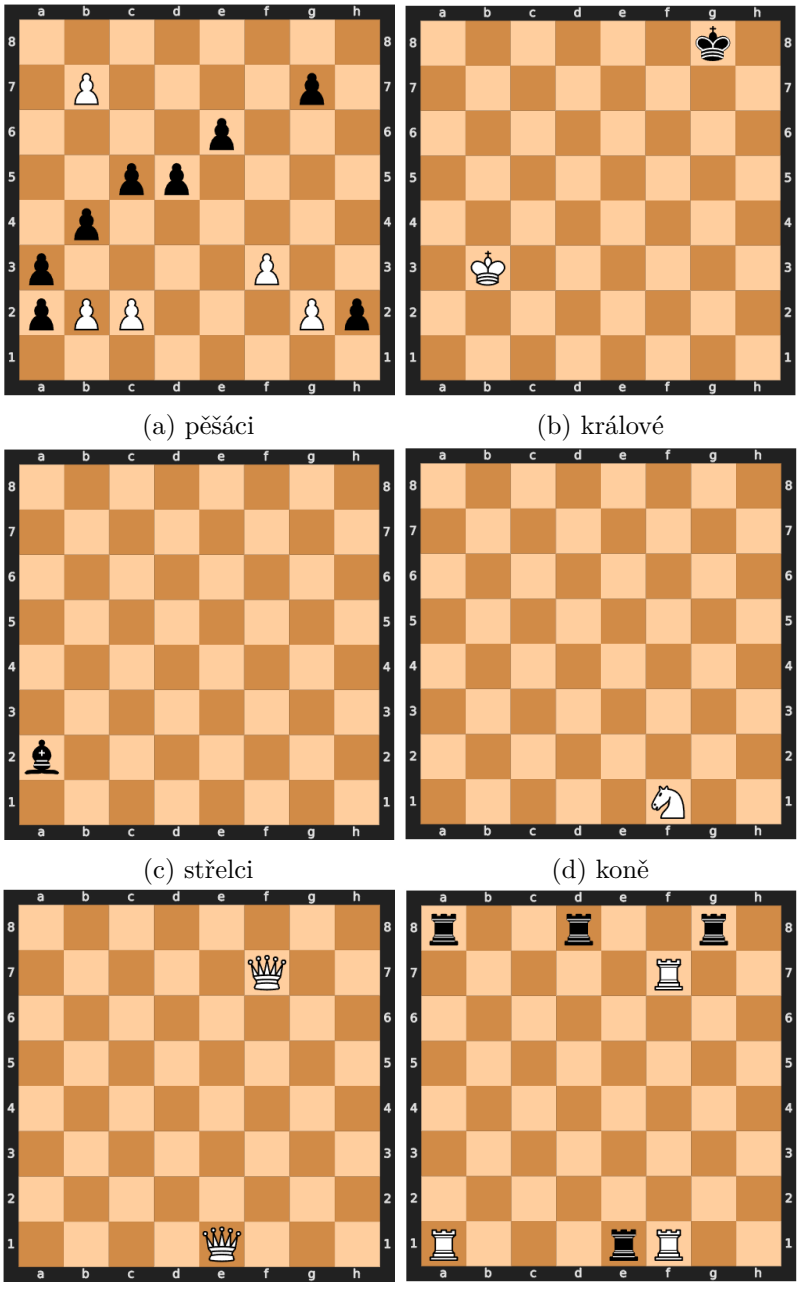
V tomto experimentu nepovažuji počet validních pozic jako podstatnou metriku, soustřeďuji se především na počet jednotlivých kamenů obou barev.

Z příkladů uvedených na obrázku 5.1 je vidět, že se relativní počty kamenů dodržují včetně barev a pozic - nejlépe je to vidět na počtu pěšců, který je významně vyšší než ostatních kamenů a zároveň jsou kameny černé barvy ve vyšších řadách, než bílé. Je opodstatnělé předpokládat, že u modelu dochází k učení, čemuž také svědčí napočtené metriky. Pro přehlednější pohled na metriky jsem připravil tabulku 5.3 zobrazující porovnání průměrných počtů kamenů pro jednotlivé sítě (v sloupci „Data“ označuje písmeno **R** reálná data a písmeno **F** generovaná data).

Tabulka 5.3: Porovnání průměru počtu kamenů natrénovaných modelů se vstupními daty

Iterace	Data	Průměr počtu kamenů					
		králů	královen	věží	střelců	koní	pěšáků
1	R	2	1.4	2.9	1.8	1.5	9.9
	F	2.7	0.9	5.7	3.4	0.9	15.1
2	R	2	1.5	2.7	1.6	1.2	10.1
	F	2.6	1.1	5.4	3.3	1.1	14.7
3	R	2	1.3	2.8	1.5	1.1	10
	F	2.9	1.1	5.4	3.2	0.9	15.1
4	R	2	1.3	2.9	1.6	1.1	10
	F	2.7	0.9	5.6	3.2	1	15.2
5	R	2	1.4	2.8	1.5	1.2	10
	F	2.6	1.1	6	3.5	1	15.1

Tento experiment mne přivedl na myšlenku složení výsledků jednotlivých sítí do jedné finální pozice, kteoru implementuji v následujícím experimentu.



(a) pěšáci (b) králové

(c) střelci (d) koně

(e) královny (f) věže

Obrázek 5.1: Příklady vygenerovaných pozic druhého experimentu obsahující pouze jeden typ kamene

5.4 Skládání samostatných sítí

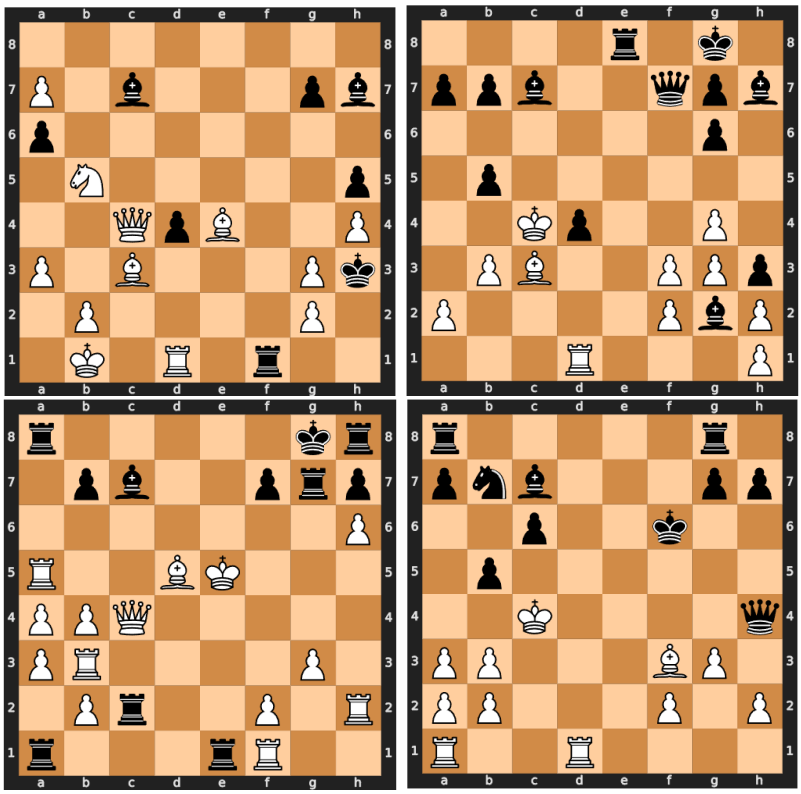
Metodika takového složení už v této práci byla použita (konkrétně v podsekci 4.2.3). Při skládání výsledků v situaci, kdy se na stejné pozici nachází více kamenů se na výslednou pozici zapíše kámen s nejvyšší relativní hodnotou.

V tomto experimentu nedochází k žádnému strojovému učení modelů, pouze skládání výsledků. Pro potřeby toho experimentu jsem doplnil implementaci o dvě funkce:

- **generate_combined_board**: pro každou síť vygeruje ukázkovou matici, následně je spojí do výsledné matice a převede do zápisu pozice ve formátu FEN (obdobně jako **funkce generate_board**)
- **combine_until**: (obdobně jako funkce **generate_until**).

V tomto experimentu jsou modely schopné vygenerovat validní pozice (avšak ne vždy), ukázkové validní pozice uvádím v obrázku 5.2. Ikdyž jsou králové ve všech příkladech v šachu, pozice je stále validní, jelikož nejsou v matu (existuje validní tah těchto kamenů).

Obrázek 5.2: Ukázkové spojené výsledky sítí z druhého experimentu.



Přestože jsou výsledky pozitivní, nedochází mezi sítěmi k žádnému sdílení znalostí (sítě jsou trénovány zvlášť), což mě přivedlo k poslednímu experimentu.

5.5 Zapouzdřené sítě

Pro dosažení sdílení znalostí mezí sítěmi je zapotřebí je zahrnout do jednotného procesu učení, ideálně do jednoho modelu. Knihovna PyTorch nabízí takové rozhraní jak pro diskriminační, tak pro generační síť. Pro tento experiment jsem implementoval dva nové modely („G_e“, „D_e“), které zapouzdřují modely z přechozího experimentu.

Modely příjmají a generují vektory velikosti 384 a využívají inspirovaného ”one-hot”zakódovaní, kde každá síť využívá pouze tři hodnoty na intervalu $[0, 1]$. Každá sít představuje jiný typ kamene, tudíž musím dodržet pořadí: stejně jako v postupu popsaném v sekci 4.2.3, sítě představují kameny vzestupně dle jejich relativní hodnoty (počínaje pěšcem a konče králem).

Následující dvě kombinace trénuji jak na originálním, tak i na datasetu obsahující pouze hlavolamy typu „vidlice“:

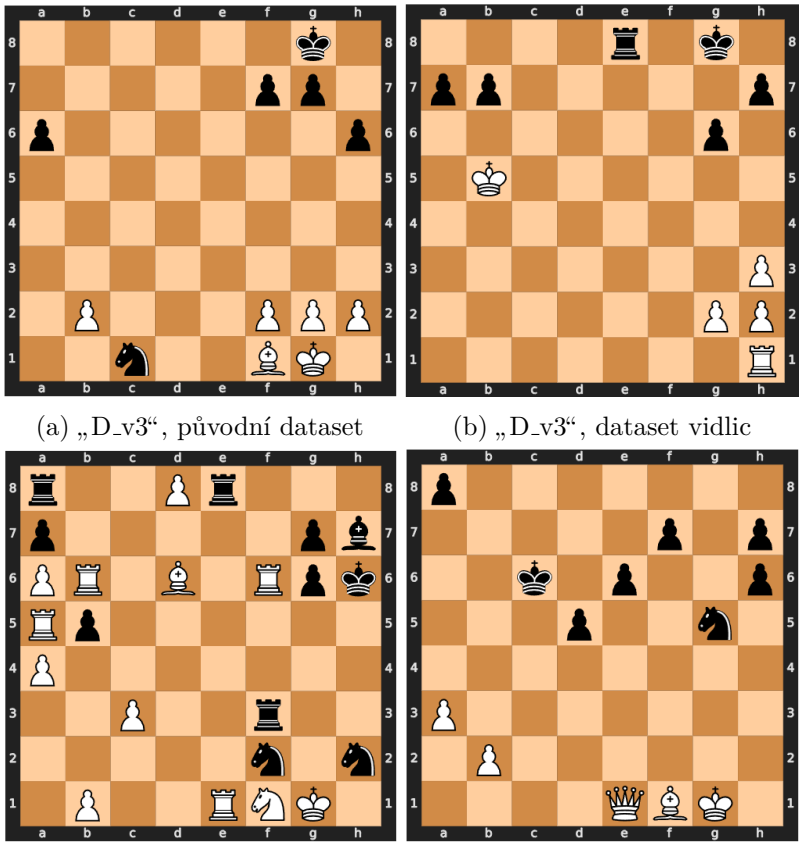
- Zapouzdřeného modelu generátoru a diskriminátoru,
- Zapozdřeného generátoru a třetí verze diskriminátoru („v3“ ilustrovaného na obrázku 3.2).

Ukázkové validní příklady uvádím v obrázku 5.3, kde popis každé šachovnice obsahuje typ diskriminátoru a dataset, na kterém byla trénována kombinace modelů (vždy se využívá zapouzdřený generátor, ale jiná verze diskriminátoru).

Vypočítané metriky ukazují, že natrénované modely generují až čtvrtinu validních pozic z vygenerované množiny vzorků velikosti 128. Pro přehlednost znázorňuje výsledky tabulka 5.4:

Tabulka 5.4: Počet validních pozic z šarže velikosti 384

Iterace	Původní dataset		Dataset vidlic	
	D_v3	D_e	D_v3	D_e
1	26	10	19	5
2	38	6	14	4
3	37	11	8	5
4	34	4	20	3
5	27	6	14	4



(a) „D_v3“, původní dataset (b) „D_v3“, dataset vidlic

(c) „D_e“, původní dataset (d) „D_e“, dataset vidlic

Obrázek 5.3: Příkladné výstupy ze čtvrtého experimentu

5.6 Shrnutí výsledků

Výsledky modelů natrénovaných na původním datasetu shrnuje tabulka 5.5. Tabulka pro každou danou kombinaci verzí modelů a kódování uvádí podíl validních pozic z množiny vygenerovaných vzorků velikosti 128 pro pět iterací (kde model „e“ diskriminátoru a generátoru označuje zapouzdřené verze z experimentu 5.5).

Tabulka 5.5: Podíl validních pozic z množiny vygenerovaných vzorků velikosti 128, zaokrouhleno

Verze modelů		Kódování	Iterace				
Gen.	Dis.		1	2	3	4	5
v1	v1	normální	0%	0%	0%	0%	0%
		škálované	0%	0%	0%	0%	0%
v1	v2	normální	0%	0%	0%	0%	0%
		škálované	0%	0%	0%	0%	0%
v2	v3	normální	0%	0%	0%	0%	0%
		škálované	0%	0%	0%	0%	0%
Složené výsledky samostatných sítí z experimentu ze sekce 5.4		škálované	12,5%	7%	11,7%	2,3%	13,3%
e	v3	one-hot	20,3%	29,7%	28,9%	26,6%	21,1%
e	e	one-hot	7,8%	4,7%	8,6%	3,1%	4,7%

Kapitola 6

Diskuze

Navržený framework splňuje vymezené požadavky, dokáže trénovat různé typy sítí s různým formátem dat.

„Normální“ zakódovaní vstupních dat se neukázalo být velmi efektivní: napočítané metriky svědčí o neschopnosti natrénovaných modelů rozeznat mezi typy jednotlivých kamenů a vyvodit z nich patřičné závěry. Škálované zakódování neposkytuje mnohem lepší výsledky, ale dokáže vygenerovat správný počet králů, což poukazuje na schopnost rozeznávání králů od ostatních kamenů či volných políček. Nejlepší výsledky přineslo inspirované „one-hot“ zakódování, které pro jeden typ kamene jasně odděluje volné políčko od dvou barev daného kamene.

Natrénované navrhnuté základní typy neuronových sítí nejsou schopné generovat validní pozice, avšak ukazují dobré výsledky při trénovaní na vstupních datech obsahující pouze jeden kámen. Při skládání výsledků takových sítí je sice možné dosáhnout vytvoření validních pozic, ovšem modely nevyvozují žádné vztahy mezi typy kamenů, jelikož jsou natrénovany pomocí vstupních dat obsahující pouze jeden daný typ kamene.

Nejlepších výsledků dosahují modely, které v sobě zapouzdřují více modelů pro jednotlivé kameny a tím jsou tím zahrnuté do jednoho procesu učení. Takové modely jsou schopné přímo generovat validní šachové pozice.

Architektura GAN je tedy schopná generování validních šachových pozic bez jakýchkoli apriori znalostí o samotné hře šachu či jejich pravidel. Omezení této architektury tkví především ve vhodné volbě formátu vstupních dat (především zvolneého kódování) a zvolených typů neuronových sítí.

Závěr

Cílem této práce bylo navrhnout a vytvořit framework pro trénování generativních kontradiktorních sítí pro generování šachových pozic, trénování základních typů neuronových sítí a ohodnocení výsledků.

Nejprve bylo zvoleno vhodné zakódování vstupních dat, na základě cehož byl poté předzpracován zvolený dataset. Poté byly navrženy základní typy neuronových sítí, zvoleny metriky pro porovnání vstupních dat s generovanými a navrhnut samotný framework.

Testovací framework byl napsán v jazyce Python v platformě Jupyter Notebook. Framework umožňuje definici vlastních neuronových sítí, výběr množiny trénovacích dat a nastavení různých parametrů trénování (například velikost vstupního datasetu nebo počet iterací). Natrénované sítě jsou následně ohodnocené na základě porovnání vypočítaných metrik generovaných a vstupních datech.

Navržené sítě vykazují dobré výsledky pouze pro zadání obsahující jeden typ kamene a při skládání výsledků jednotlivých sítí jsou vytvořené validní pozice. Ovšem až zakomponování jednotlivých sítí do jednoho zapozdřeného modelu je framework schopen generovat validní pozice.

Pro budoucí vývoj se nabízí rozšířit implementaci o další, složitější typy neuronových sítí a využití generovaných pozic natrénovaných modelů jako šachových hlavolamů.

Literatura

[1] ReNom: AE&VAE. 2018. Dostupné z: https://www.renom.jp/notebooks/tutorial/generative-model/VAE/fig4.png

[2] Kramer, M. A.: Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, ročník 37, č. 2, 1991: s. 233–243.

[3] Zhang, H.; Sindagi, V.; Patel, V. M.: Image De-raining Using a Conditional Generative Adversarial Network. *CoRR*, ročník abs/1701.05957, 2017, 1701.05957. Dostupné z: http://arxiv.org/abs/1701.05957

[4] Duplessis, T.: lichess.org open database. https://database.lichess.org/, accessed: 2021-12-22.

[5] Goodfellow, I. J.; Pouget-Abadie, J.; Mirza, M.; aj.: Generative Adversarial Networks. 2014, doi:10.48550/ARXIV.1406.2661. Dostupné z: https://arxiv.org/abs/1406.2661

[6] Jebara, T.: *Discriminative, Generative and Imitative Learning*. Dizertační práce, MIT, 2002. Dostupné z: /ref/jebara/jebara4.pdf

[7] Schmidhuber, J.: Deep Learning in Neural Networks: An Overview. *CoRR*, ročník abs/1404.7828, 2014, 1404.7828. Dostupné z: http://arxiv.org/abs/1404.7828

[8] Ballard, D. H.: Modular Learning in Neural Networks. In *Proc. AAAI*, 1987, s. 279–284.

[9] Kingma, D. P.; Welling, M.: Auto-Encoding Variational Bayes. 2013, doi:10.48550/ARXIV.1312.6114. Dostupné z: https://arxiv.org/abs/1312.6114

[10] Kingma, D. P.; Welling, M.: An Introduction to Variational Autoencoders. *CoRR*, ročník abs/1906.02691, 2019, 1906.02691. Dostupné z: http://arxiv.org/abs/1906.02691

[11] Bowman, S. R.; Vilnis, L.; Vinyals, O.; aj.: Generating Sentences from a Continuous Space. *CoRR*, ročník abs/1511.06349, 2015, 1511.06349. Dostupné z: http://arxiv.org/abs/1511.06349

[12] Ravanbakhsh, S.; Lanusse, F.; Mandelbaum, R.; aj.: Enabling Dark Energy Science with Deep Generative Models of Galaxy Images. 2016, doi:10.48550/ARXIV.1609.05796. Dostupné z: https://arxiv.org/abs/1609.05796

[13] White, T.: Sampling Generative Networks: Notes on a Few Effective Techniques. *CoRR*, ročník abs/1609.04468, 2016, 1609.04468. Dostupné z: http://arxiv.org/abs/1609.04468

[14] Wikipedie: Křížová entropie — Wikipedie: Otevřená encyklopedie. 2021, [Online; navštíveno 17. 04. 2022]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=K%C5%99%C3%AD%C5%BEov%C3%A1_entropie&oldid=20325842

[15] theateist: Feb 2017. Dostupné z: https://stackoverflow.com/questions/41990250/what-is-cross-entropy

[16] Bisneto, T. R. V.; de Carvalho Filho, A. O.; Magalhães, D. M. V.: Generative adversarial network and texture features applied to automatic glaucoma detection. *Applied Soft Computing*, ročník 90, 2020: str. 106165, ISSN 1568-4946, doi:https://doi.org/10.1016/j.asoc.2020.106165. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1568494620301058

[17] Wu, A. N.; Biljecki, F.: GANmapper: geographical content filling. *CoRR*, ročník abs/2108.04232, 2021, 2108.04232. Dostupné z: https://arxiv.org/abs/2108.04232

[18] Radford, A.; Metz, L.; Chintala, S.: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR*, ročník abs/1511.06434, 2016.

[19] Mirza, M.; Osindero, S.: Conditional Generative Adversarial Nets. *CoRR*, ročník abs/1411.1784, 2014, 1411.1784. Dostupné z: http://arxiv.org/abs/1411.1784

[20] Arjovsky, M.; Chintala, S.; Bottou, L.: Wasserstein GAN. 2017, doi:10.48550/ARXIV.1701.07875. Dostupné z: https://arxiv.org/abs/1701.07875

[21] Fedus, W.; Goodfellow, I.; Dai, A. M.: MaskGAN: Better Text Generation via Filling in the_____. 2018. Dostupné z: https://arxiv.org/abs/1801.07736

[22] Zec, E. L.; Arnelid, H.; Mohammadiha, N.: Recurrent conditional gans for time series sensor modelling. In *Time Series Workshop at International Conference on Machine Learning,(Long Beach, California)*, 2019.

[23] Chidambaram, M.; Qi, Y.: Style Transfer Generative Adversarial Networks: Learning to Play Chess Differently. *CoRR*, ročník abs/1702.06762, 2017, 1702.06762. Dostupné z: http://arxiv.org/abs/1702.06762

[24] Sim, V.: Dec 2020. Dostupné z: https://towardsdatascience.com/magnusgan-using-gans-to-play-like-chess-masters-9dded439bc56

[25] Niklas Fiekas: python-chess: a chess library for Python. Dostupné z: https://github.com/niklasf/python-chess

[26] Jazzband: PrettyTable. Dostupné z: https://github.com/jazzband/prettytable

[27] Wes McKinney: Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, editace Stéfan van der Walt; Jarrod Millman, 2010, s. 56 – 61, doi:10.25080/Majora-92bf1922-00a.

[28] Harris, C. R.; Millman, K. J.; van der Walt, S. J.; aj.: Array programming with NumPy. *Nature*, ročník 585, č. 7825, Září 2020: s. 357–362, doi:10.1038/s41586-020-2649-2. Dostupné z: https://doi.org/10.1038/s41586-020-2649-2

[29] Paszke, A.; Gross, S.; Massa, F.; aj.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, editace H. Wallach; H. Larochelle; A. Beygelzimer; F. d'Alché-Buc; E. Fox; R. Garnett, Curran Associates, Inc., 2019, s. 8024–8035. Dostupné z: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[30] Inkawhich, N.: DCGAN Tutorial. Dostupné z: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Příloha **A**

Seznam použitých zkratek

GAN Generative Adversarial Networks

FEN Forsyth-Edwards Notation

SAN Shortened Algebraic Notation

NLCPA Nonlinear Principal Component Analysis

VAE Variational Autoencoder

STGAN Style Transfer Generative Adversarial Networks

DCGAN Deep Convolution Generative Adversarial Nets

cGAN Conditional Generative Adversarial Nets

RGB Red Green Blue

Příloha **B**

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu CD
- `thesis.pdf` ..... text práce ve formátu PDF
- `src`
  - `chess-gan.ipynb` ..... implementace ve formátu Jupyter Notebooku
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `data` ..... adresář s datasety
    - `lichess_db_puzzle.csv.bz2` .... originální komprimovaný dataset
    - `processed_puzzles.csv` ..... předzpracovaný dataset
- `experiments` ..... adresář s výstupy jednotlivých experimentů
  - `metrics` ..... adresář s napočítanými metrikami
  - `examples` ..... adřesář s ukázkovými pozicemi