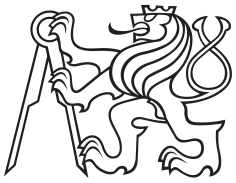
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Hráč hry Logik učený z příkladů

Matěj Blažek

Vedoucí: Vojtěch Franc, Ph.D.
Obor: Otevřená informatika
Studijní program: Počítačové hry a grafika
Květen 2023

Poděkování

Děkuji Vojtěchu Francovi, Ph.D., za schůzky, čas a nasměrování věnovanou tomuto projektu. Dále chci poděkovat ČVUT, že jsem mohl mít příležitost a zázemí na to se vzdělávat. Také chci poděkovat své rodině, za poskytnutí velkého množství elektřiny, nutného k trénování modelů.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 10. května 2023

Abstrakt

Tato práce se zabývá použitím algoritmů posilovaného učení pro vytvoření strojového hráče hry Logik. Za tímto účelem je navržen simulátor hry Logik, který pracuje se symbolickým i grafickým vstupem. Simulátor je použit pro natrénování strojového hráče a k porovnání jeho výkonu s existujícími algoritmy, které pro tuto hru navrhl člověk. V práci jsme pro učení strojového hráče použili algoritmus PPO, a navrhli pro něj vhodnou hodnotící funkci, díky které lze naučit strojového hráče na běžném PC. Naučený hráč dosahuje o něco horších výsledků než nejlepší, lidmi vytvořené algoritmy, které jsou ale použitelné jen pro symbolickou variantu hry. Naproti tomu hráč, naučený pomocí PPO, funguje i na grafické variantě hry, kdy je vstupem obrázek hracího pole.

Klíčová slova: Logik, Posilované učení

Vedoucí: Vojtěch Franc, Ph.D.
Karlovo náměstí 13,
Praha 2

Abstract

This thesis explores the use of reinforcement learning algorithms to create a machine player for the game Logik. To this end, a Mastermind game simulator is designed that works with both symbolic and graphical input. The simulator is used to train the machine player and to compare its performance with existing human-designed algorithms for the game. In this thesis, we have used the PPO algorithm for training the machine player and proposed a suitable evaluation function for it, which can be used to teach the machine player on a regular PC. The trained player performs slightly worse than the best human-made algorithms, but these are only applicable to the symbolic version of the game. In contrast, a player trained using PPO also works on the graphical version of the game, where the input is an image of the playing field.

Keywords: Mastermind, Reinforcement learning

Title translation: Player of Mastermind learned from examples

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

V tomto projektu se zaměříme na hráče deskové hry Logik, zvané taktéž Mastermind, učeného z příkladů. Cílem je napsat program, který se naučí hrát hru Logik sám, v simulovaném prostředí, s minimem apriorní znalosti o strategii hry dodané člověkem. Hra Logik byla záměrně vybrána, má jednoduchá pravidla, ale hrát optimální tahy je pro člověka náročné.

Hráč by měl zvládat hru jak se symbolickým vstupem, tak se vstupem zadaným jako obrázek hracího pole. V případě grafického vstupu je problém výrazně těžší, protože umělý hráč musí nejdříve rozpoznat stav hry z obrázku a ještě optimálně zahrát tah.

Jako hlavní nástroj pro řešení problému jsem použil algoritmy posilovaného učení. Tyto algoritmy učí agenta optimálnímu chování v simulovaném prostředí, které v našem případě představuje hra Logik. Optimální chování se definuje volbou takzvané hodnotící funkce (reward function), jejíž volba závisí na aplikaci, pro kterou se algoritmus posilovaného učení použije.

Podstatná část této práce se věnuje návrhu simulátoru, který musí umožňovat symbolický i grafický vstup, ladění parametrů algoritmu učení a vhodný výběr odměnové funkce, jejíž tvar se ukázal jako stěžejní komponenta pro zajištění konvergence učení.

Hra je podrobně popsána v sekci 1.1. Kapitola 2 se zaměřuje jak na rešerši existující literatury věnující se programovému řešení hry Logik, tak na metody posilovaného učení, které v práci využívám. Kapitola 3 se věnuje implementaci

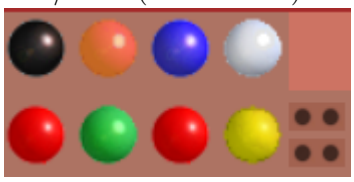
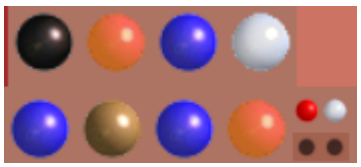
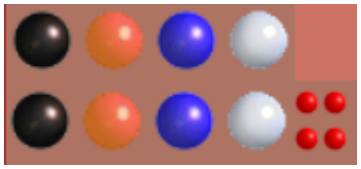
simulátoru se symbolickým vstupem a implementací existujících algoritmů na pozdější porovnání výkonnosti s algoritmy učených z příkladů. Kapitola 4.1 se zaměřuje na použití posilovaného učení na symbolické prostředí a vytvoření grafického prostředí pro toto učení. Kapitola 5 je věnována experimentálnímu porovnání výkonnosti, navržených učících se algoritmů, v závislosti na jejích odměnové funkci, porovnání rychlosti použití algoritmu a porovnání algoritmu s grafickým vstupem při zhoršující se kvalitě vstupu. Kapitola 6 shrne získané poznatky a informace.

1.1 Popis hry Logik

Hra Logik původně vznikla jako stolní hra pro dva hráče [1]. Na začátku hry se hráči domluvili, kdo bude vytvářet kód, neboli uspořádanou sekvenci kolíků různých barev, kdo se tento kód bude snažit prolomit. Kód měl délku 2 až 8 kolíků a 4 až 8 barev podle různých výrobců a kopií hry Logik. Nejčastější byla délka kódu čtyři s šesti různými barvami, proto tuto kopii budeme považovat za standartní velikost hry.

Hádající hráč každé kolo zahraje libovolnou kombinaci čtyř kolíků šesti barev. Za každý kolík správné barvy na správné pozici, dá protihráč jeden červený minikolík. Za každý kolík správné barvy na špatné pozici, dá protihráč jeden bílý minikolík. Cílem hádájícího hráče je co nejdříve uhodnout kód protihráče. Po úspěšném uhodnutí, nebo zahrání deseti neuspěšných tahů si zaznamená hráč počet tahů, za které stihl kód prolomit, a změní si roli s protihráčem. Vyhrává hráč, kterému se podařilo kód prolomit za menší počet tahů.

Obrázek 1.1: Ilustrační obrázek deskové hry

Příklad	Zadání (horní řádek) / Tah (dolní řádek)	Hodnocení tahu
Příklad 1:		0 červených, 0 bílých minikolíků
Příklad 2:		1 červený, 1 bílý minikolík
Příklad 3:		4 červené minikolíky

Tabulka 1.1: Příklad tahů a jejich hodnocení.

Vysvětlení příkladu 1: Zde protihráč ohodnotí protihráčův tah nula červenými minikolíky a nula bílými minikolíky, protože hádající hráč neumístil ani jeden kolík použité barvy v kódu.

Vysvětlení příkladu 2: Zde protihráč ohodnotí tah jedním červeným minikolíkem, protože umístil jeden kolík se stejnou barvu na správnou pozici. Dále hodnotí jedním bílým minikolíkem, jeden za špatně umístěný oranžový kolík.

Vysvětlení příkladu 3: Zde protihráč ohodnotí tah čtyřmi červenými minikolíky. Jelikož hráč prolomil celý protivníkův kód, vyhrál, a započte si počet kol za které kód prolomil.

Velikost hry Dále budu odkazovat na různé velikosti prostředí pomocí zkratek. Pokuď napíši jen Logik, hovořím o standartní variantě s šesti barvami a délkou kódu čtyři. Jestliže změním velikost prostředí budu to označovat Logik x-y, kde x je počet barev a y je velikost kódu. Standartní velikost je tedy Logik 6-4, často zkoumaná prostředí budou například Logik 4-2 a Logik 4-4.

Kapitola 2

Rešerše existující literatury

Vzhledem k popularitě problému posilovaného učení a jeho použití k částečnému vyřešení stolních her, na které dosavaď nemáme výpočetní techniku a paměť pro vyřešení standartním, logickým, algoritmem (například šachy nebo go), je zde velký výběr odborné literatury zaměřené na tyto problémy. Zaměřím se převážně na literaturu zabývající se posilovaným učením a literaturu zabývající se strojovým hraním hry Logik.

2.1 Posilované učení

Posilované učení je metoda strojového učení, která se zaměřuje na učení agenta, v nějakém prostředí pomocí odměn a trestů [14]. Posilované učení se liší od ostatních metod, tím, že se neučí z předem nashromážděných dat, ani logických funkcí napsaných programátorem, ale náhodně experimentuje s různými rozhodnutími a vyhodnocuje, jaké ohodnocení získá za akce, které vykoná. Z úspěšných rozhodnutí se pak snaží extrahovat strategii pro další hry.

Mezi nejúspěšnější algoritmy posilovaného učení patří například AlphaGo [2], AlphaStar [4], AlphaZero [3], zaměřené na stolní a počítačové hry. V reálném prostředí se používá například na Autonomní řízení [5], kategorizace předmětů [6] apod.

Při studiu algoritmů posilovaného učení pro účely této práce jsem čerpal z následujících publikací: [7], [8], [9], [10]. Knihy se zabývají posilovaným učením od základu do podrobností.

2.2 Existující algoritmy navržení člověkem

V roce 1977 Donald Knuth vytvořil algoritmus, který na základě minimax teorie dokázal základní konfiguraci Logik hrát průměrně s efektivitou 4.478, dále dokáže vždy hru vyřešit do pěti tahů. [11].

V roce 1993 Koyama a Lai pomocí vyčerpávajícího hledání do hloubky dokázali vytvořit algoritmus hrající s efektivitou 4.3403. [12]

V roce 2004 Michiel de Bond, prokázal, že hra Logik je NP kompletní. Je tedy velmi obtížné jej vyřešit. Pro velké instance hracího pole je tedy výpočetně náročné nalézt optimální strategii. Nicméně pro malé/střední instance je to možné, jak ukázali Koyama a Lai. [13]

Kapitola 3

Existující algoritmy pro symblolickou variantu

V této kapitole budu popisovat tvorbu simulátoru hry Logik, popis měření výkonnosti a algoritmy vytvořené lidmi pro porovnání s posilovaným učením.

3.1 Symbolický Logik simulátor

Pro simulování her Logik je potřeba vytvořit simulátor, který bude efektivně simulovat opakovaně hry.

Prostředí je vytvořeno pomocí programovacího jazyku Python jako textový simulátor. Simulátor nejprve vygeneruje náhodnou uspořádanou čtveřici celých čísel, kde se každé číslo vyskytuje v rozsahu 0 až 5, reprezentující šest různých barev (možno změnit velikost hry pomocí parametru). Hráč tedy zadává uspořádnou čtveřici čísel reprezentující různé kolíky do funkce, která vrací uživateli uspořádanou dvojici čísel [počet červených minikolíků, počet bílých minikolíků]. Hráč vyhraje, pokuď dostane 4 červené minikolíky. Pokuď hráč nestihne kód prolomit do 10 kol, prohrává, a generuje se nový kód.

3.2 Měření výkonnosti

Metrikou pro měření výkonnosti je očekáváná délka hry, v případě, že zadání je generováno náhodně z uniformního rozdělení. Matematické očekávání jsme aproximovali aritmetickým průměrem počítaným z N her, kdy N bylo v rozmezí 10000 až 100000 podle náročnosti učení algoritmu. Směrodatná odchylka, takto získaných odhadů, byla 0.005-0.003, tedy o tři řády menší, než rozdíly efektivity porovnávaných algoritmů. U každého algoritmu bude specifikován počet odehraných her a směrodatná odchylka. Teoreticky nejhorší výkonnost je 10.00, kdyby algoritmus nedokázal prolomit žádný kód do deseti tahů.

Teoreticky nejlepší výkonnost je 1.00, kdyby algoritmus pokaždé prolomil kód na první pokus. Čím nižší číslo, tím lepší je algoritmus.

3.3 Náhodný algoritmus

Algoritmus, který každý tah zvolí zcela náhodnou akci. Tento algoritmus a jeho varianta popsaná v další sekci, byl vytvořen jako referenční řešení, pro získání představy, jak dobře lze hru hrát pomocí triviální strategie.

Výkonnost	Logik 4-2	Logik 4-4	Logik 6-4
Náhodný algoritmus	7.602	9.827	9.965

Tabulka 3.1: Výkonnost náhodného algoritmu po 100000 hrách se směrodatnou odchylkou 0.003

3.3.1 Náhodný algoritmus bez repetice

Algoritmus zvolí náhodný tah, který ještě nehrál.

Výkonnost	Logik 4-2	Logik 4-4	Logik 6-4
Náhodný algoritmus bez repetice	7.173	9.823	9.965

Tabulka 3.2: Výkonnost náhodného algoritmu s opakováním po 100000 hrách se směrodatnou odchylkou 0.003

Z výkonnosti na zvětšujících se prostředích lze posoudit, že čím větší je prostředí, tím víc se blíží k náhodnému algoritmu, protože počet možných akcí se mnohonásobně zvětšuje.

3.4 Eliminační algoritmus

Algoritmus udržuje množinu všech možných tahů, ze kterých postupně eliminuje tahy podle získaného ohodnocení. Kvůli velikosti množiny tahů, je pro člověka obtížné/nemožné tuto strategii použít.

Popis algoritmu:

1. Algoritmus si vygeneruje množinu všech možných tahů, nazývejme tuto množinu S.
2. Algoritmus vybere tah z množiny S a zahraje ho. Výběr tahu se specifikuje v dalších dvou subsekcích.
3. Za vybraný tah dostane jako odměnu červené a bílé minikolíky. Pokud prolomí kód, algoritmus končí.
4. Porovná všechny tahy z množiny S s posledním zahraným tahem, jako by to bylo řešení. Pokud pro daný tah nezíská stejné ohodnocení, tj. stejný počet malých červených kolíků a stejný počet malých bílých kolíků, vyřadí tah z množiny S.
5. Pokračuje na krok 2 s redukovanou množinou S.

Další podsekce popisují varianty eliminačního algoritmu, které se liší výběrem tahů z množiny S, implementovaným v kroku 2.

3.4.1 Eliminační algoritmus s náhodným výběrem

Algoritmus v kroku 2 vybere náhodný prvek z množiny S.

3.4.2 Eliminační algoritmus s výběrem podle četnosti

Algoritmus v kroku 2 spočte četnost jednotlivých barev z prvků v množině S. Poté z množiny S vybere náhodný prvek, který ovšem obsahuje barvu největší četnosti. Cílem výběru je poskytnout informaci o nejčastější barvě a odstranit co nejvíc prvků z množiny.

Výkonnost	Logik 4-2	Logik 4-4	Logik 6-4
Eliminační algoritmus s náhodným vybíráním	2.86	3.185	4.74
Eliminační algoritmus s vybíráním podle četnosti	3.186	3.698	4.62

Tabulka 3.3: Porovnání výkonností eliminačních algoritmů podle výběru prvku z množiny S po 100000 hrách se směrodatnou odchylkou odhadu 0.003

Z výkonnosti můžeme vidět, že u větších prostředí je druhý přístup lepší. U velmi malých prostředí naopak vybírání prvků s největší četností zvětšuje šanci na vybrání prvku s repeticí jedné barvy, která neposkytne tolik informací a odstraní průměrně méně prvků ze setu S.

Pro další experimenty, bude-li se psát o Eliminačním algoritmu, myslí se s náhodným výběrem.

3.5 Knuthův algoritmus

Algoritmus vymyšlen v roce 1977 Donaldem Knuthem [11]. Tento algoritmus je výjímečný v tom, že standartní velikost hry dohraje vždy do pěti kol, zatímco ostatní podobně výkonné algoritmy mají nejhorší výkonnost 6 kol.

Průběh algoritmu:

1. Algoritmus si vygeneruje množinu všech možných tahů, nazývejme tuto množinu S.
2. První tah si algoritmus vybere libovolnou kombinaci dvojice jedné barvy a druhé dvojice druhé barvy (například 0 0 1 1 nebo 2 1 2 1 atd.).

3. Za vybraný tah dostane jako odměnu červené a bílé minikolíky. Pokud prolomí kód, algoritmus končí.

4. Porovná všechny tahy z množiny S s posledním zahraným tahem, jako by to bylo řešení. Pokud nezíská stejný počet malých červených kolíků a stejný počet malých bílých kolíků, vyřadí tah z množiny S.

5. Vygeneruje si všechny možné kombinace odměn.

6. Za každý tah, který ještě nebyl zahrán (nikoliv jen z množiny S), zjistí počet odebraných tahů z množiny S pro každou vygenerovanou odměnu, kdyby byl zahrán. K tahu pak přiřadí nejmenší počet odstraněných tahů. Tah, který má největší počet těchto minimálně odstraněných tahů, je zvolen jako další tah.

7. Algoritmus pokračuje krokem číslo 3.

Výkonnost	Logik 4-2	Logik 4-4	Logik 6-4
Knuthův algoritmus	2.756	3.631	4.478

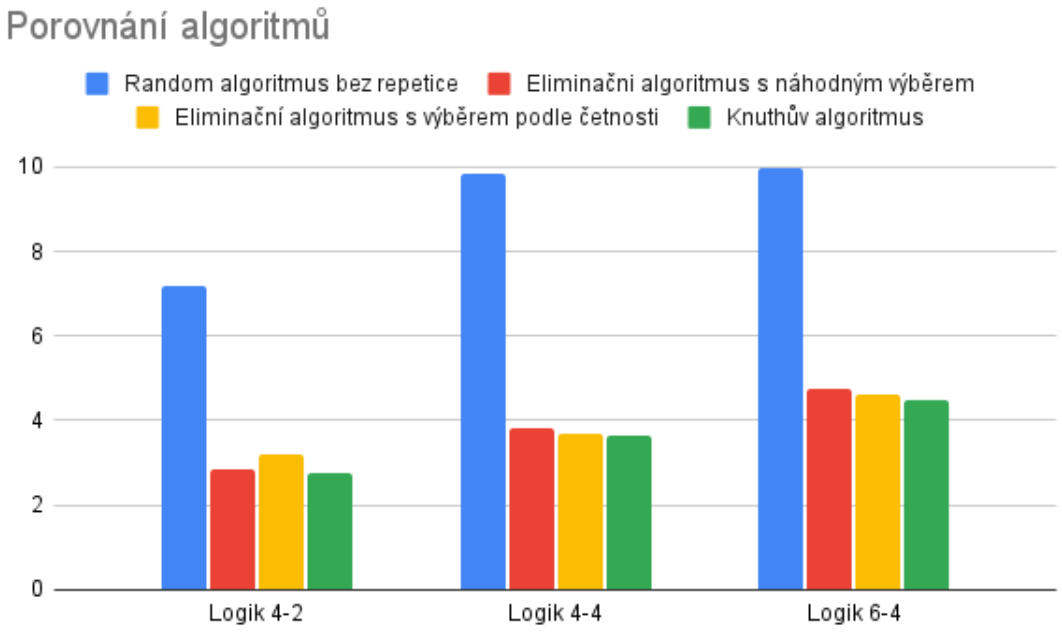
Tabulka 3.4: Výkonnost Knuthovo algoritmu po 10000 hrách s směrodatnou odchylkou 0.005

3.6 Porovnání výkonnosti logických algoritmů

Všechny tyto algoritmy byly odvozeny od pravidel hry a používájí lidmi navrženou strategii k zahrání optimálního tahu. Zde je porovnání výkonnosti těchto algoritmů.

Výkonnost	Logik 4-2	Logik 4-4	Logik 6-4
Náhodný bez repetice	7.173	9.823	9.965
Eliminační s náhodným výběrem	2.860	3.815	4.741
Eliminační s vybíráním podle četnosti	3.186	3.698	4.620
Knuthův	2.756	3.631	4.478

Tabulka 3.5: Porovnání výkonnosti algoritmů navržených člověkem



Obrázek 3.1: Grafické porovnání algoritmů navržených člověkem

Z grafu lze vidět, že Knuthův algoritmus je nejlépe hrající, a budeme ho používat jako referenční řešení v tomto projektu pro pozdější porovnání.

Kapitola 4

Navržené řešení pomocí algoritmů posilovaného učení

Algoritmy pro hraní hry Logik, které jsme popsali v předchozí kapitole, byly vymyšlené člověkem. V této kapitole popíšeme navržené řešení, které spočívá v použití metod strojového učení, jenž dokáže algoritmus pro hraní hry Logik naučit z příkladů, s minimálním použitím apriorní informace o tom, co je správná strategie.

4.1 Posilované učení

Posilované učení je metoda strojového učení, která se zaměřuje na způsob jakým se agent umístěný v nějakém prostředí může učit optimílnímu chování na základě náhodných pokusů , které jsou hodnoceny pomocí odměn a trestů [14]. Posilované učení se liší od ostatních metod tím, že se neučí z předem nashromážděných dat, ani logických funkcí napsaných programátorem, ale experimentuje s různými rozhodnutími a vyhodnocuje, jaké ohodnocení získá za akce, které vykoná.

Ze začátku učení vykonává akce zcela náhodně a čím déle se trénuje, tím více vybírá akce s očekávanou největší odměnou. Cílem algoritmu tedy je maximalizovat celkovou odměnu za všechny své akce.

Hlavní výhodou posilovaného učení je, že nemusíme vymýšlet strategii

chování agenta, ale jen navrhnout prostředí a funkci hodnotící jeho chování. Algoritmus RL strategii chování vymyslí sám, a jeho výkonnost je jedna z nejlepších, pokuď jsou správně implementovány a dostatečně natrenovány (viz AlphaGo [2], AlphaStar [4], OpenAi [15], apod.)

4.2 Výběr algoritmu

Výběr algoritmu je zásadní pro efektivitu v daném prostředí. Mezi standartní algoritmy posilovaného učení můžeme zařadit například:

4.2.1 Q-Learning

V Q-learningu je cílem naučit agenta rozhodovat se na základě ohodnocení, které obdrží za své akce v daném prostředí [7]. Tyto ohodnocení jsou uloženy v Q-tabulce, která přiřazuje každé kombinaci stavu a akce určitou hodnotu, tzv. Q-hodnotu. Tyto hodnoty jsou aktualizovány na základě získaných odměn a výsledných stavů prostředí pomocí tohoto vzorce:

$$Q^{new}(s_t, a_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \Big(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} \Big)$$

Obrázek 4.1: Q-Learning vzorec pro aktualizaci hodnoty

Problém při použití Q-Learning je, že pro každý stav si musí alokovat paměť na Q-hodnoty všech stavů a akcí. Pro standartní hru Logik je zde počet stavů:

$$(PočetBarev^{DélkaKódu} * PočetOhdnocení)^{PočetKol}$$

$$(6^4 * 14)^{10} = 18144^{10} \approx 3.86 * 10^{42}$$

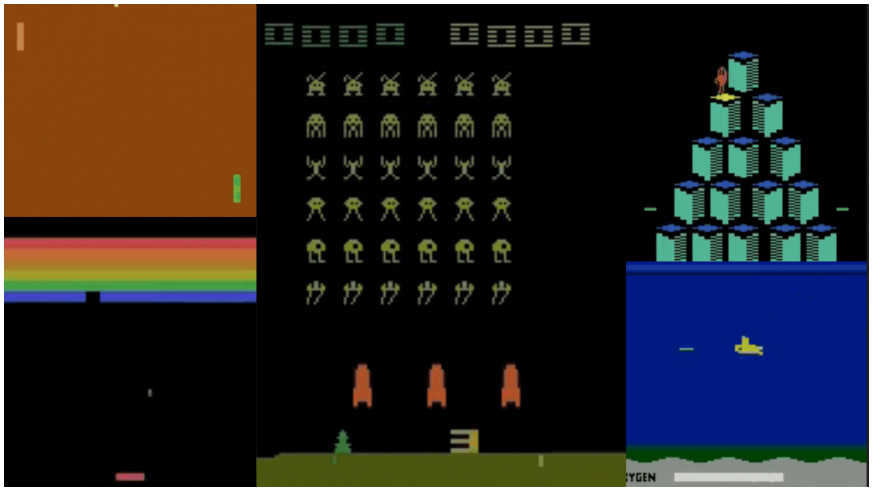
Tolik paměti, na zapamatování Q-hodnot, není v současné době realizovatelné. Samotný Q-learning tedy **není vhodný** pro prostředí Logik.

4.2.2 DQN

Algoritmus DQN (Deep Q Network) kombinuje Q-Learning s hlubokými neuronovými sítěmi [8]. Hlavním cílem je naučit agenta rozhodovat se na základě maximální hodnoty Q, jako algoritmus Q-learning. Hlavní rozdíl je, že DQN používá hlubokou neuronovou síť jako aproximátor Q-funkce. Vstupem do sítě jsou stavy prostředí, a výstupem jsou Q-hodnoty pro každou možnou akci v daném stavu. Tato síť je trénovaná simulací her, právě tímto agentem.

Hlavní výhoda DQN oproti klasickému Q-learningu je v použití hlubokých neuronových sítí. Tyto sítě jsou schopny reprezentovat složitější Q-funkce a naučit se optimalizovat více akcí najednou, což vede k lepšímu výkonu agenta. Navíc tím, že DQN využívá sítě jako approximátor Q-funkce, tak se může učit v prostředích s velkým množstvím stavů a akcí.

Mezi průkopníky této metody patří team DeepMind, který použil DQN k naučení hráčů známých Atari her, jako například Space invaders, Pong, Breakout apod. Vstup algoritmu je stejný, jako u lidského hráče, tj. sekvence obrázků generovaný hrou.



Obrázek 4.2: Ukázka Atari hry, kterou lze hrát pomocí hráče naučeného DQN

Díky approximaci Q-fuknce neuronovou sítí, není problém velikost prostředí Logik.

Algoritmus tedy **je vhodný** k použití, ale podle porovnání výkonnosti na Atari hrách, není nejvýkonnější [17].

4.2.3 PPO

[16]Algoritmus PPO (Proximal Policy Opitmalization) se používá k trénování agentů pro rozhodování v prostředích s velkým počtem akcí a stavů. Hlavním cílem PPO je naučit agenta, jakým způsobem reagovat na prostředí tak, aby maximalizoval zisk (odměnu).

PPO využívá techniky založené na gradientu ztrátové funkce, což znamená, že model se učí na základě gradientů ztrátové funkce, které jsou počítány z trénovacích dat. Tyto gradienty se používají k aktualizaci parametrů modelu, což zlepšuje jeho výkon.

Algoritmus **je vhodný** k použití na prostředí Logik, z výkonů na hře Space Invader, je vidět, že PPO je efektivnější než DQN a také se více hodí na prostředí s velkou množinou akcí, jaká je ve hře Logik[17]. Proto jsem si jako hlavní algoritmus vybral PPO.

4.3 PPO

PPO (Proximal policy optimization) je metoda posilvaného učení popularizovaná společností OpenAI [15].

[16]Algoritmus PPO je reprezentován neuronovou sítí. Jako vstup dostane stav daného prostředí a jako výstup má rozdělení pravděpodobností nad množinou akcí, které může zahrát. Algoritmus využívá cílovou funkci (Action function) a náhradní cíl (value funkci). Cílová funkce, nebo také "Actor", se snaží maximalizovat odměny, zatímco náhradní cíl, nebo také "Critic", hodnotí chování cílové funkce, a podle hodnocení se změní parametry neuronové sítě. Zásadní technokou PPO je ořezávnání (clipping), který zabraňuje příliš velkým skokům v reprezentaci a zlepšuje stabilitu algoritmu.

Hlavní kompnenty metody PPO jsou následující:

Reprezentace: PPO je reprezentováno neuronovou sítí, která jako vstup přijímá stav prostředí a jako výstupy vrací rozdělení pravděpodobností nad množinou akcí.

Cílová funkce: PPO definuje objektivní funkci, která měří výkonnost reprezentace. Cílem objektivní funkce je obvykle maximalizovat očekávanou kumulativní odměnu za všechny tahy.

Náhradní cíl: PPO má náhradní cíl, který aproximuje objektivní funkci a usnadňuje stabilnější aktualizace. Náhradní cíl je založen na poměru mezi pravděpodobnostmi akcí vybraných aktualizovanou reprezentací a pravděpodobnostmi akcí vybraných předchozí reprezentací. Tento poměr se vynásobí odhadem, o kolik byla akce lepší, nebo horší ve srovnání s průměrnou akcí v daném stavu. Cílem je maximalizovat tento náhradní cíl.

Aktualizace reprezentace: PPO provádí několik iterací aktualizace reprezentace s využitím shromážděné dávky zahraných her. V každé iteraci se maximalizuje náhradní cíl pomocí optimalizačních metod, jako je stochastický sestup po gradientu. PPO však zahrnuje omezení aktualizace reprezentace, aby bylo zajištěno, že zůstane blízká předchozí reprezentaci. Toto je hlavní myšlenka tohoto algoritmu a zabraňuje velkým aktualizacím reprezentace, které mohou vést k nestabilnímu učení.

Ořezávání: K vynucení omezení aktualizace politiky používá PPO techniku zvanou ořezávání. Náhradní cíl je oříznut na rozsah určený hyperparametrem, který se nazývá parametr oříznutí. Toto oříznutí zajišťuje, že se aktualizace reprezentace příliš neodchyluje od předchozí reprezentace, a zabraňuje provádění velkých změn reprezentace.

Iterativní aktualizací reprezentace s omezením změn se PPO snaží najít reprezentaci, která v daném prostředí funguje dobře a maximalizuje kumulativní odměny, aniž by příliš rychle měnila směr učení, což může dlouhodobě velmi uškodit algoritmu.

4.3.1 Hyperparametry

Algoritmus PPO má několik hyperparametrů, jejichž nastavení má zasadní vliv na kvalitu naučené strategie a rychlost konvergence učení.

[17]Pro správné fungování algoritmu jsem zvolil původní nastevení hyperparametrů stejné jako natrénovaný model z Atari hry Space Invaders, který jsem poupravoval k zlepšení výkonnosti v prostředí Logik [17].

Mezi hyperparametry patří:

1. **Learning rate**

   Rychlost učení algoritmu, příliš rychlé učení a naučí se hrát neoptimálně. Příliš malý learning rate a rychlost učení bude velmi pomalá.

   Finální hodnota: **0.0001**

2. **Clip range**

   Maximální velikost o kolik se může změnit reprezentace. Příliš malá a nebude se dostatečně rychle učit, příliš velká a jednotlivé modely budou mít mezi sebou příliš velký rozdíl.

   Finální hodnota: **0.15**

3. **Entropy coeficient**

   Entropní koeficient pro výpočet ztrátové funkce

   Finální hodnota: **0.01**

4. Batch size

   Počet odehraných her před předáním dat modelu a aktualizaci modelu.

   Finální hodnota: **256**

Nastavení těchto hyperparametrů bylo kritické pro výkonnost algoritmu. Se špatným nastavením algoritmus nekonverguje k použitelnému řešení.

4.4 Aplikace algoritmu na prostředí Logik

Algoritmus implemtuji pomocí knihovny stable-baselines3 [17], která vyžaduje převedení simulátoru do prostředí knihovny gym [18]. Hra musí mít implmentované dvě nutné metody.

Zaprvé metodu reset, která restartuje hru do nového stavu a vrátí nový resetovaný stav. Zadruhé metodu step, která zahraje tah a vrátí nový stav, informaci, zda je hra dokončena, odměnu za tah a info pro vývojáře na debug.

4.4.1 Odměnová funkce

Velmi důležitá, pro konvergenci algoritmu, je i odměnová funkce. Největší odměna je za dohrání, jinak za každý tah, co není prolomený kód, je základně negativní odměna, čímž algoritmus učení nasměruje k výběru tahu, který vede k rychlému dokončení hry. Jelikož je to velmi velké prostředí s mnoho akcemi, je důležité "nasměrovat"algoritmus částečnými odměnami jinak by se zasekl a opakoval by dokola akce které mu jednou vyhrály hru, ale nejsou celkově úspěšné. Proto je důležité odměňovat částečné kroky. Všechny varianty odměn byly otestovány v sekci 5.1.

4.4.2 Symbolická reprezentace

Pro prostředí gym je potřeba vytvořit symbolickou reprzentaci celého stavu, podle kterého se algoritmus rozhoduje. Každé kolo je reprezentováno šesticí čísel, čtyři čísla reprezentující barvy zahrané a dvě čísla reprezentující počet červených a bílých minikolíků.

Příklad jedné hry vypadá následovně:
Kolo 1:[1, 2, 3, 4, 2, 1,
Kolo 2: 1, 2, 4, 0, 3, 0,
Kolo 3: 1, 2, 4, 5, 3, 0,
Kolo 4: 1, 2, 4, 4, 3, 0,
Kolo 5: 1, 2, 4, 1, 4, 0,
Kolo 6: -1,-1,-1,-1,-1,-1,
Kolo 7: -1,-1,-1,-1,-1,-1,
Kolo 8: -1,-1,-1,-1,-1,-1,
Kolo 9: -1,-1,-1,-1,-1,-1,
Kolo 10: -1,-1,-1,-1,-1,-1]

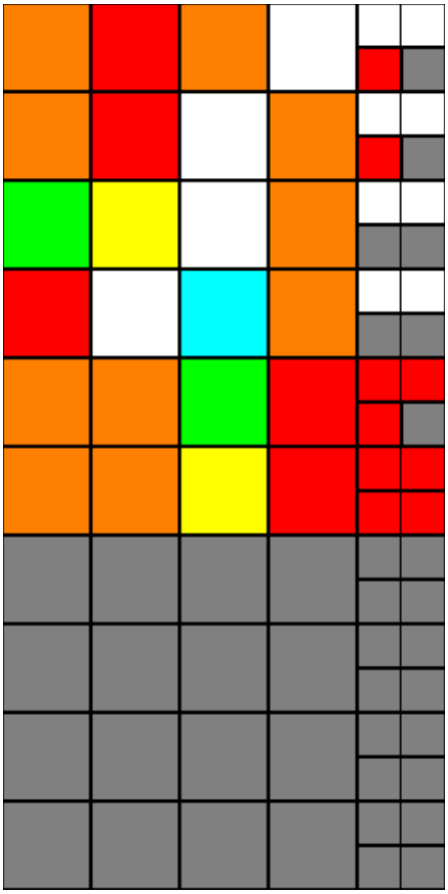
V prvním kole bylo tedy zahráno [1,2,3,4] a odměna byla 2 červené a 1 bílý minikolík. V pátém kole bylo zahráno [1,2,4,1] a odměna byla 4 červené minikolíky, kód tedy byl prolomen a hra je dohrána. Všechny nezahrané tahy jsou nastaveny na -1. Tato reprezentace není efektivní a je doporučeno omezit všechna čísla v symbolizaci na 0 nebo 1. Proto jsem použil metodu enumerate a každé číslo reprezentuji jedničkou na daném indexu. Všechny -1 jsou reprezentovány jenom nulami. Číslo 1 by tedy bylo: 0, 1, 0, 0, 0, 0 <- počet čísel podle počtu barev. Číslo 2 za minikolíky by bylo 0, 0, 1, 0, 0 <- počet čísel podle délky kódu. Celý první tah z minulé hry by byl tedy reprezentován jako:

$$[0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0]$$

Symbolická reprezentace celé hry, uvedené v předchozím příkladu, je následující:
Kolo 1:[0,1,0,0,0,0, 0,0,1,0,0,0, 0,0,0,1,0,0, 0,0,0,0,1,0, 0,0,1,0,0, 0,1,0,0,0
Kolo 2: 0,1,0,0,0,0, 0,0,1,0,0,0, 0,0,0,0,1,0, 0,0,0,0,0,1, 0,0,0,1,0, 1,0,0,0,0,
Kolo 3: 0,1,0,0,0,0, 0,0,1,0,0,0, 0,0,0,0,1,0, 0,0,0,0,0,1, 0,0,0,1,0, 1,0,0,0,0,
Kolo 4: 0,1,0,0,0,0, 0,0,1,0,0,0, 0,0,0,0,1,0, 0,0,0,0,1,0, 0,0,0,1,0, 1,0,0,0,0,
Kolo 5: 0,1,0,0,0,0, 0,0,1,0,0,0, 0,0,0,0,1,0, 0,1,0,0,0,0, 0,0,0,0,1, 1,0,0,0,0,
Kolo 6: 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0,
Kolo 7: 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0,
Kolo 8: 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0,
Kolo 9: 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0,
Kolo 10: 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0]

4.4.3 Grafická reprezentace

Simulátor v případě grafického vstup generuje barevný RGB obrázek, na kterém je zobrazen současný stav hracího pole, včetně získaných odměn za jednotlivé stavy. Grafický vstup je inspirován fyzickou kopií hry Logik 1.1. Příklad grafického vstupu je na obrázku 4.3



Obrázek 4.3: Grafické znázornění jedné dohrané hry

Algoritmus tedy dostává array pixelů, kterou si znormalizuje, aby byly hodnoty pouze mezi 0 a 1. Obrázky se generují převedením symbolického stavu na grafický a tím se potom algoritmus s grafickým vstupem trénuje. Hra je tedy učená z samotných příkladů.

4.5 Výkonnost posilovaného učení

Výkonnost strategie, získané pomocí posilovaného učení, se s délkou učení typicky zlepšuje (může se i zhošit v nějakých částech), proto zde prezentuji data získané postupným tréninkem omezeným na 50 milionů episod, kde jedna episoda je jeden odehraný tah. Hry odehrané v grafu 4.4, jsou odehrané s ohodnocením 2

Základní nastavení

1. Velikost neuronové sítě
   a. Symbolická varianta
   Síť je sekvenční typu Multi-Layer Perceptron (MLP) a má 3 vzájemně propojené vrstvy, každá se skládající z 256 neuronů. Při více použitých neuronů se drasticky zmenší rychlost trénování. Z důvodu omezených výpočetnách prostředí větší než 3 vrstvou MLP síť nepoužíváme.
   b. Grafická varianta
   [17]Síť je konvoluční a skládá se z tří sekvenčních částí, první část je sekvenční neuronová síť, s vstupem 2816 neuronů a výstupem 512 neuronů, pro celý algoritmus, druhá část je pouze pro část ohodnocující value funkce, s vstupem 512 neuronů a výstupem pouze 1 neuron s ohodnocením, jako vstup má výsledky z první části. Třetí část je ohodnocení action funkce s vstupem 512 neuronů a výstupem 24 neuronů,vstup jsou také výsledky z první části.
2. Odměnová funkce
   Pro obě prostředí byla použita funkce 2.

Zde je průběh trénování za 50 milionů episod.

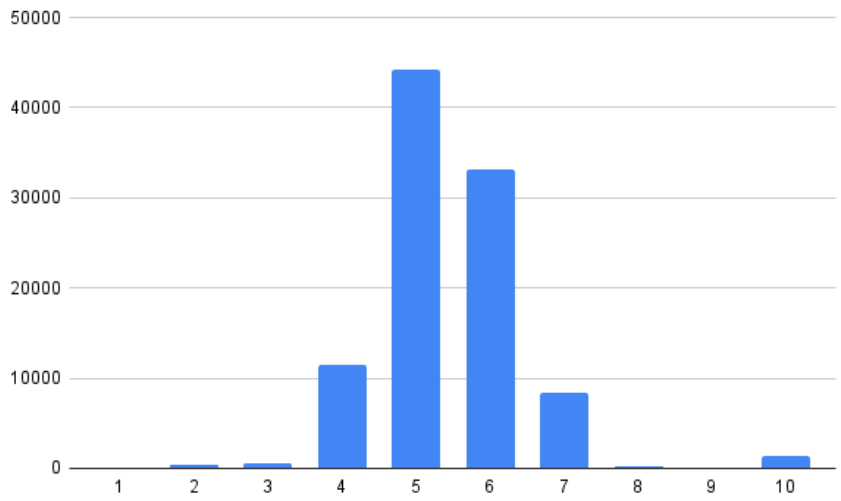
Obrázek 4.4: Učení s základními parametry, osa X: Počet episod, osa Y: Průměrná délka tahu za posledních 100 her

Počet episod	0	10M	20M	30M	40M	50M
Výkonnost modelu	10	5.95	5.43	5.43	5.318	5.33

Tabulka 4.1: Učení modelu PPO se symbolickým vstupem se základními parametry

Ze grafu 4.4 a tabulky 4.4 lze vypozorovat, že se algoritmus rychle dostane k častečnému neoptimálnímu řešení, a poté se velmi pomalu přibližuje optimálnějšímu řešení v prostředí Logik 6-4.

S natrénovaným modelem je výkonnost za 100 000 her: **5.438**. S rozdělením výsledků 4.5:

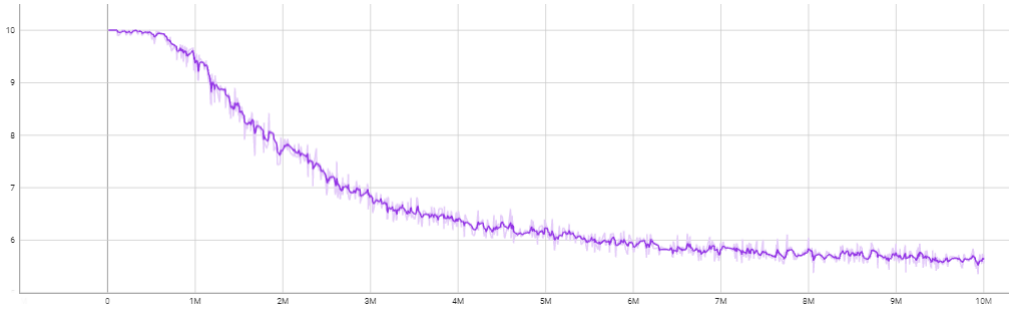


Obrázek 4.5: Rozdělení výsledků natrénovaného modelu za 100000 her.

Z rozdělení 4.5 je vidět, že většina her je vyřešena za 5 až 6 pokusů. Dále je tu významná menšina her, které se algoritmu nepovedlo vyřešit a téměř žádné hry vyřešené v 8 nebo 9 tazích. Příčinou může být učení algoritmu. Učení zastaví s dohranou hrou. Jelikož masivní většinu her dohrává do šesti tahů, nenaučí se model dobře žádný z dalších tahů, protože není na tyto tahy dostatečně natrénován.

4.5.1 S grafickým vstupem

Jelikož zpracování grafického vstupu je cca 15x pomalejší budu ukazovat výsledek učení pouze do 10M kroků.



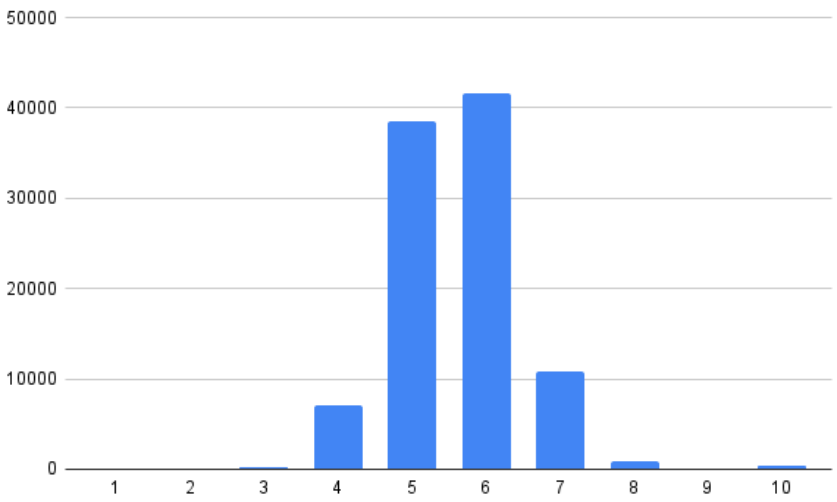
Obrázek 4.6: Rozdělení výsledků grafického vstupu natrénovaného modelu za 100000 her.

Počet episod	0	2M	4M	6M	8M	10M
Výkonnost modelu	10	7.75	6.42	5.95	5.86	5.56

Tabulka 4.2: Učení grafického vstupu s základními parametry

Z rozdělení 4.7 můžeme vidět většinu her trvajících 6 kol a překvapivě méně her skončí neprolomeným kódem než u symbolického vstupu. Důvodem může být právě horší průměrné trénování, takže algoritmus se naučí dobře hádat i po šestém tahu.

S natrénovaným modelem je výkonnost za 100000 her: **5.602**. S normálovým rozdělením:



Obrázek 4.7: Učení grafického vstupu

Kapitola 5

Experimenty

V této kapitole budeme pozorovat efektivitu algoritmů s velkým zaměřením na rychlost učení posilovaného učení za různých předpokladů. V této kapitole experimentálně vyhodnotíme efektivitu algoritmů posilovaného učení PPO a výkon naučených hráčů. Zaměříme se na měření rychlosti učení s různou odměnovou funkcí 5.1, rychlost rozhodování různých algoritmů 5.2 a měření výkonosti grafického vstupu s zhoršující se kvalitou vstupu 5.3.

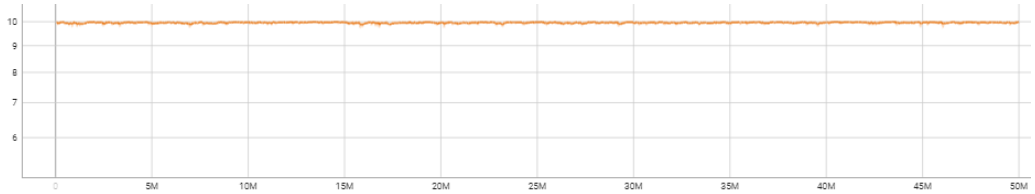
5.1 Porovnání odměnové funkcí v prostředí Logik 6-4

Zde budeme pozorovat výkonnost učení posilovaného učení v závislosti na odměnovou funkci. Všechny tyto pokusy jsou pro větší rychlost učení trénovány na symbolické variantě.

Odměnová funkce

1. Odměna pouze za dohrání hry

   Odměnová funkce hodnotí všechny tahy odměnou -1. Výjimka je pouze u tahu, co prolomí kód, ten hodnotí odměnou 1.



Obrázek 5.1: Učení s odměnou pouze za dohrání hry. Osa X: Počet episod. Osa Y: Průměrná délka tahu za posledních 100 her.

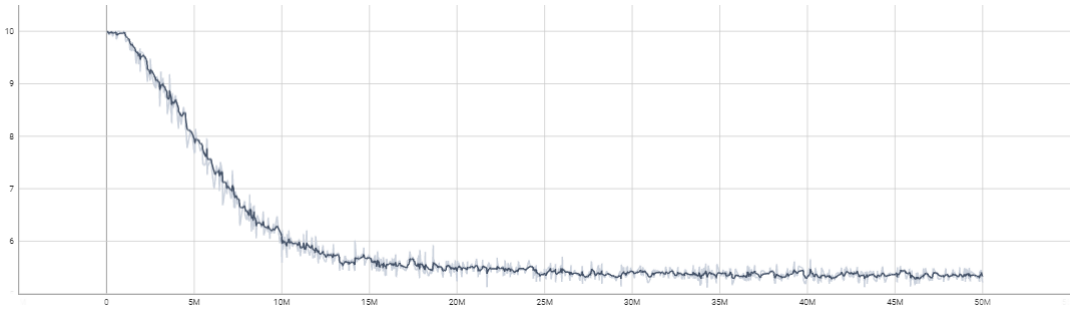
Z grafu lze vidět, že algoritmus se vůbec nezlepšuje a potřebuje alespoň základní nasměrování. Dále bude tato odměna automaticky zakomponována ve všech dále zmíněných odměnových funkcí.

2. Základní ohodnocení tahu

   Algoritmus si k základní odměně přičte 0.15 za každý bílý a 0.24 za každý červený minikolík. Cílem tohoto ohodnocení je, aby stále byla odměna menší než 0, aby se snažil co nejdříve prolomit kód a zároveň prioritizovat červené nad bílými kolíky.

   Další funkce v základním ohodnocení je, že pokud zahraje barvu, která stoprocentně nemůže být v kód je tah penalizován zpátky na -1. Tyto barvy se zjištují pouze z tahů, které jsou ohodnoceny žádným nebo čtyřmi minikolíky. Pokud použije barvu z tahu, co neměla žádne minikolíky, je penalizován, protože tato barva v kódu být nemůže. Pokud naopak nepoužije barvu z tahu, co měl 4 minikolíky jako odměnu (například 2 červené, 2 bílé), je taktéž penalizován, protože kód musí obsahovat pouze barvy z tohoto tahu.

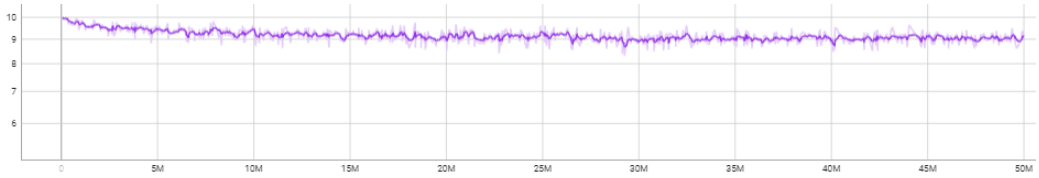
   Také pokud zahraje kód, který už byl hrán, je hodnocen odměnou -1.



Obrázek 5.2: Učení základního ohodnocení tahu, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

3. Odměna za tah, u kterého je možné že je kód

   Algoritmus si na začátku hry vytvoří množinu všech možných kódů. Po zahrání každého tahu je tato množina zmenšena o všechny tahy, které kód být už nemůžou. Pokud je zahraný kód v této množině přičte se k odměně 0.5. Odměna za tah v této množině, který ale neprolomí kód je tedy -0.5.

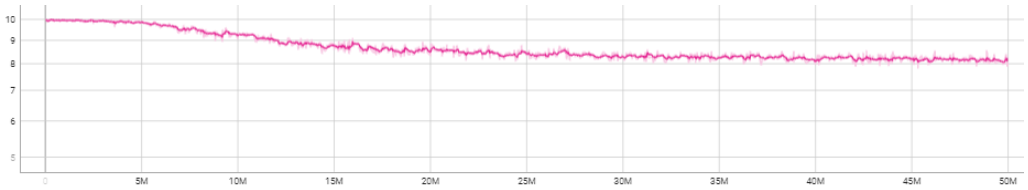


Obrázek 5.3: Učení ohodnocení kontrolující zda je tah v množině možných tahů, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

Z grafu je vidět, že se nenaučí prolomit kód a zasekne se průměrně okolo devíti tahů za hru.

4. Odměna podle počtu odstraněných tahů

   Zlepšení předchozí odměnové funkce. Základní odměna je -1 a přičte se procentuální počet odstraněných tahů z množiny možných tahů. Pokud tedy v množině je možný počet tahů před zahráním 100 a po zahrání 10, odměna za tento tah je -0.1, protože odstraní 90% tahů.

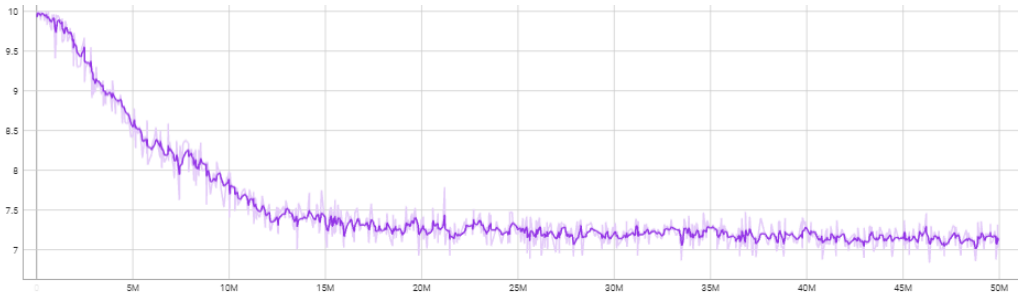


Obrázek 5.4: Učení ohodnocení podle počtu odstraněných tahů, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

Z grafu je vidět učí rychleji než algoritmus předtím, ale stále nedostatečně.

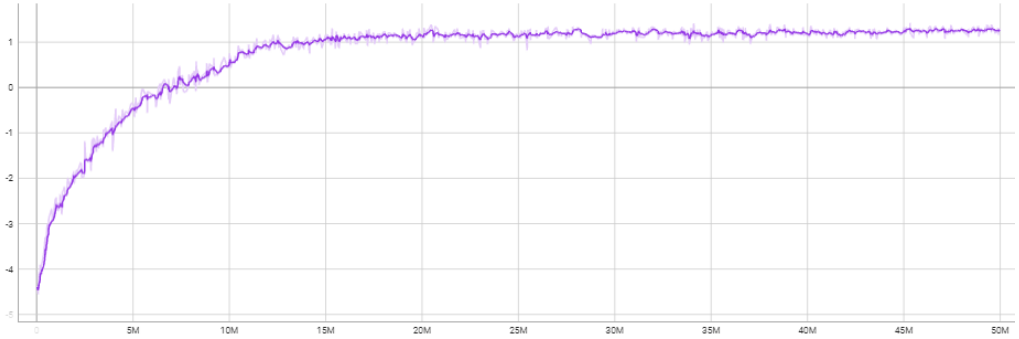
5. Odměna podle počtu odstraněných + základní ohodnocení

   Odměna za počet odstraněných tahů mi logicky přišla jako krok dobrým směrem, který by mohl ještě vylepšit výkonnost základního hodnocení. Zkombinoval jsem tedy obě hodnocení.



Obrázek 5.5: Učení ohodnocení podle počtu odstraněných tahů + základní ohodnocení, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

Algoritmus se dostal přibližně na výkonnost 7, což není nejlepší. Rozhodl jsem se zkontrolovat odměnovou funkci.



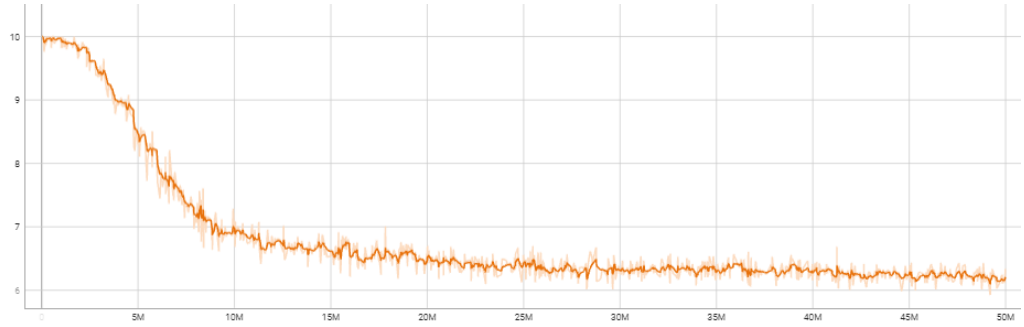
Obrázek 5.6: Zvyšování odměny s učením, Osa X: Počet episod, Osa Y: Odměna episody

Z grafu je vidět, že odměnová funkce roste přes 0, díky konstantním odměnám za kolo je možné, že aby model maximalizoval odměny, hraje tahy, které nejsou optimální k co nejmenší délce hry, ale odměnu udržuje každý tah lehce nad nulou a tím si maximalizuje počet bodů za hru.

Je tedy potřeba upravit odměnovou funkci aby odměna nebyla konstantní.

6. Odměna podle počtu odstraněných + upravené základní ohodnocení

Rozdíl od předchozí odměnové funkce je pouze upravení konstantních hodnot odměn za minikolíky na relativní. Místo 0.24 za červený kolík dostane odměnu (1/4) * (-reward) a místo 0.15 za bílý (1/6) * (-reward) po snížení za ubrání tahů. Pokud tedy bylo ubráno 20% tahů, reward se nastavý na -0.8 a poté se přičte 0.2 (0.8 * 1/4) za každý červený a 0.133 (0.8 * 1/6) za každý bílý minikolík. Odměna tedy nikdy za tento tah nepřesáhne 0, pokud neuhádne kód.



Obrázek 5.7: Učení ohodnocení podle počtu odstraněných tahů + základní ohodnocení, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

5.1.1 Experimenty s Prostředím Logik 4-2

Jen pro porovnání, jestli je chyba konceptuální nebo je způsobena nízkým počtem epizod, jenž jsem musel použít kvůli omezenému výpočetnímu výkonu, jsem zopakoval experiment 2 na menším prostředí pouze s 16 možnými akcemi (místo 1296). Pokud algoritmus zvládne vyřešit prostředí s jinou odměnovou funkcí na jednoduším prostředí, teoreticky by měl zvládnout vyřešit i prostředí s velkým akčním prostorem s silnější výpočetní technikou.

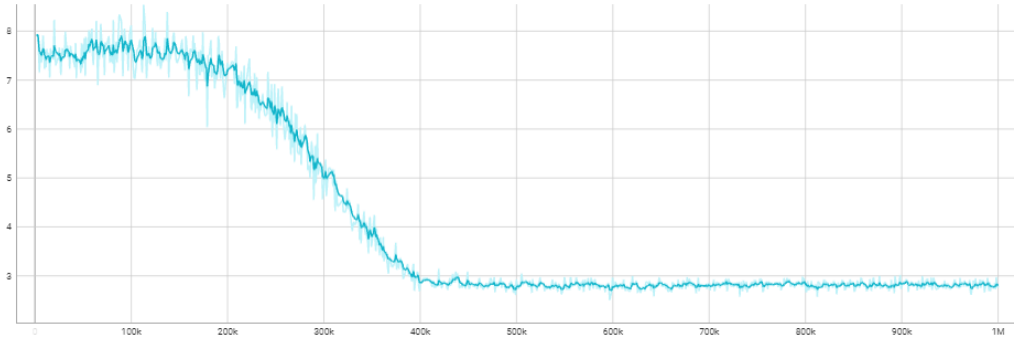
Pro toto malé prostředí nám stačí na naučení 1M kroků.

1. Odměna pouze za dohrání hry

Obrázek 5.8: Učení na prostředí Logik 4-2, odměna pouze za dohrání, Osa X: Počet epizod, Osa Y: Průměrná délka tahu za posledních 100 her

Algoritmus dosáhl optima, odměnová funkce je tedy **validní**.

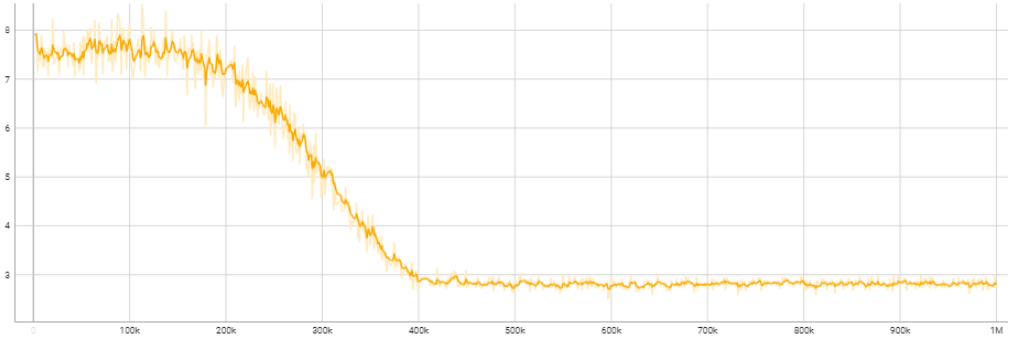
2. Základní ohodnocení tahu



Obrázek 5.9: Učení na prostředí Logik 4-2, základní ohodnocení, Osa X: Počet epizod, Osa Y: Průměrná délka tahu za posledních 100 her

Algoritmus dosáhl optima, odměnová funkce je tedy **validní**.

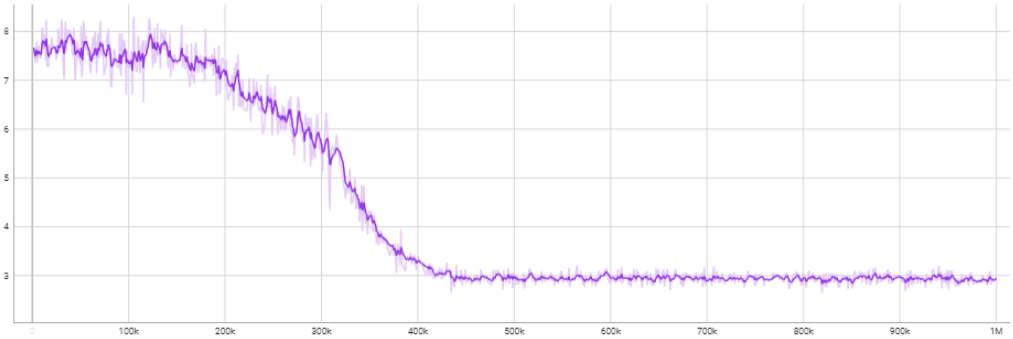
3. Odměna za tah, u kterého je možné že je kód



Obrázek 5.10: Učení na prostředí Logik 4-2,tah je možný kód, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

Algoritmus dosáhl optima, odměnová funkce je tedy **validní**.

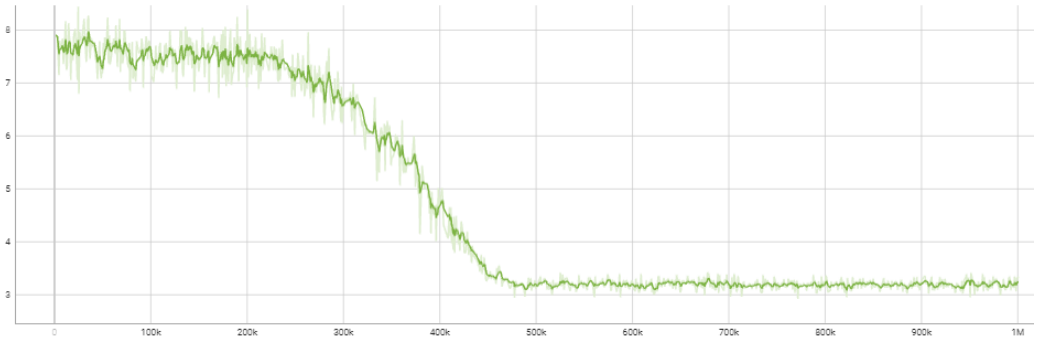
4. Odměna podle počtu odstraněných tahů



Obrázek 5.11: Učení na prostředí Logik 4-2, odměna podle počtu odstraněných tahů, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

Algoritmus dosáhl optima, odměnová funkce je tedy **validní**.

5. Odměna podle počtu odstraněných + základní ohodnocení Zde jsem trochu změnil odměnovou funkci a místo odměn 0.24 a 0.15 za červené a bílé minikolíky jsem zdvojnásobil odměnu, protože je zde poloviční délka kódu, aby dobře odměna simulovala větší prostředí.

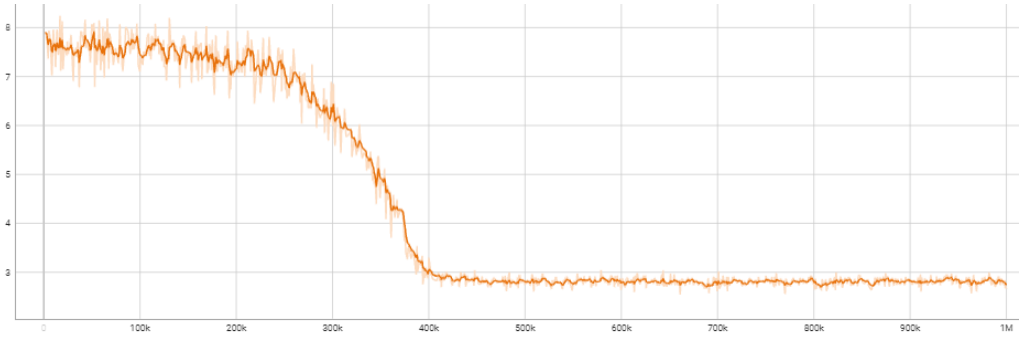


Obrázek 5.12: Učení na prostředí Logik 4-2,odměna podle počtu odstraněných + základní ohodnocení, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

Zde algoritmus nedosáhl optima, výkonnost je přibližně o 0.4 nižší než u optima, algoritmus se tedy snaží hromadit body, prodlužováním hry. Tato odměnová funkce **není validní**.

6. Odměna podle počtu odstraněných + upravené základní ohodnocení

   Zde abych lépe simuloval změny prostředí, změnil jsem relativní odměnu z 1/4 na 1/2 za červený a z 1/6 na 1/3 za bílý minikolík.



Obrázek 5.13: Odměna podle počtu odstraněných + upravené základní ohodnocení, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

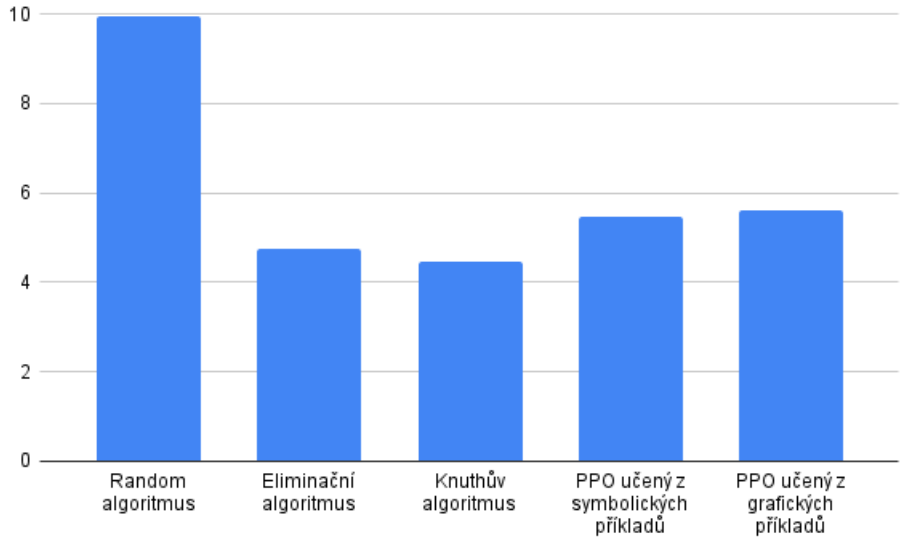
Algoritmus dosáhl výkonnosti přibližně 3.0, optimum je přibližně 2.8, minimalizace počtu tahů, tedy **není optimální** cesta k řešení.

5.2 Porovnání rychlosti rozhodování mezi algoritmy

Velmi důležité v praktickém použití je i rychlost rozhodování algoritmu. Pokud uděláme například algoritmus, který si hrubou sílou vypočte všechny možné

varianty, až do konce hry a zvolí ten nejvíce efektivní kód, bude velmi dlouho trvat, než to vypočítá a v praktickém použití by algoritmus nebyl efektivní.

Zde je pro porovnání výkonnost všech algoritmů.



Obrázek 5.14: Porovnání výkonu algoritmů

Zde vidíme, že Knuthův algoritmus má nejlepší výkonnost, eliminační algoritmus je těsně za ním a PPO učené z příkladů je relativně o 1.1 méně výkonné než Knuthův. Zkusíme si porovnat časovou náročnost na dohrání 100 her.

Algoritmus	Náhodný	Eliminační	Knuthův	PPOs	PPOg
Sekund/100 her	0,01	0,29	1386,87	1,3	4,05

Tabulka 5.1: Porovnání rychlosti rozhodování algoritmů

Z dat jde poznat exhaustivní výpočet všech možných řešení Knuthovým algoritmem. Nejlepší poměr efektivity na čas má eliminační algoritmus, který je jen o trochu méně efektivní než Knuthův algoritmus a potřebuje velmi krátkou dobu na dohrání.

Algoritmy PPO mají lehce podoptimální výkonnost a přijatelnou časovou efektivitu, ale za to mají velkou výhodu samostatného učení. A PPO s grafickým vstupem je jediný z těchto algoritmů, který vůbec dokáže přijmout

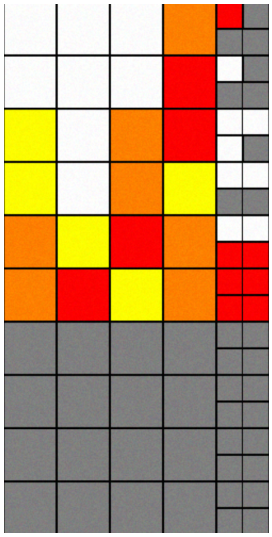
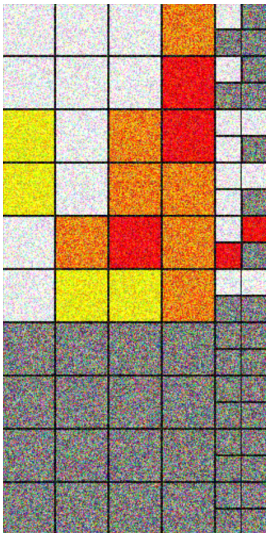
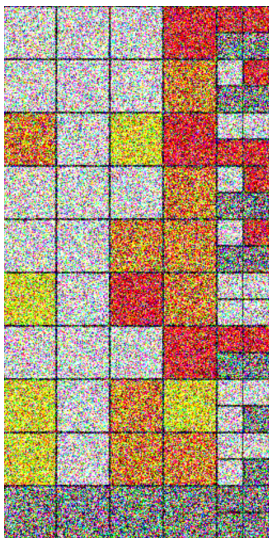
jako vstup obrázek a nemusí se dělat žádný klasifikátor, který by se pokoušel převádět obrázky do symbolické varianty.

5.3 Porovnání výkonnosti PPO algoritmu s grafickým vstupem při zhoršující kvalitě vstupu

Experimenty popsané v předchozí částech používaly jako grafický vstup nezkreslený synteticky generovaný obrázek. V reálním prostředí jsou obrázky zatížené šumem a ovlivněné velkým množstvím zkreslení, která jsou způsobena např. různým nasvícením scény, orientací a posunutím hracího pole, prespektivní zkreslením atd. V této sekci popíši několik typů zkreslení obrázků a empiricky vyhodnotím jejich vliv na rychlost učení PPO algoritmu a výkon naučeného hráče.

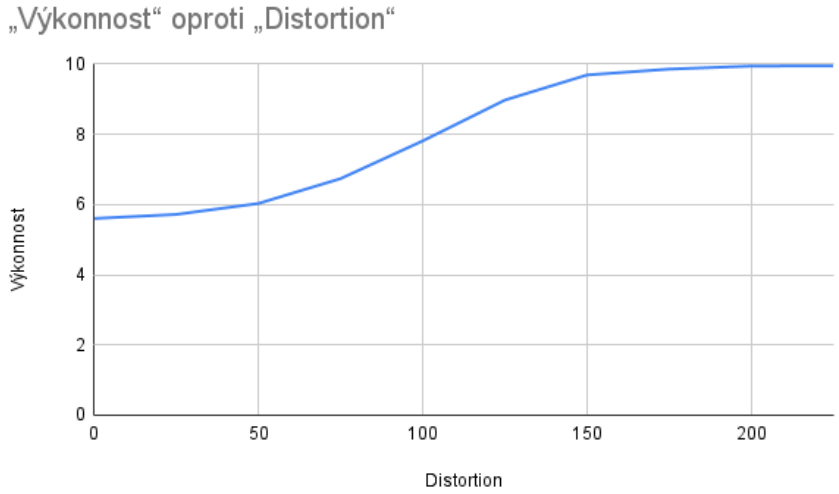
Zkreslení 1, náhodná změna barvy:

Do každého z RGB barevných kanálů se náhodně přidá náhodná barva. Maximální odchylka od původní hodnoty barvy se řídí parametrem "distortion". Pokud je tedy velký distortion, je maximální odchylka od původní barvy větší.

Příklady zkreslení	Grafická reprezentace	hodnota zkreslení
Příklad 1:		Distortion 10
Příklad 2:		Distortion 100
Příklad 3:		Distortion 245

Tabulka 5.2: Příklady zkreslení 1, náhodné změna barvy

Zde je změřená výkonnost s zvětšujícím se distortion.

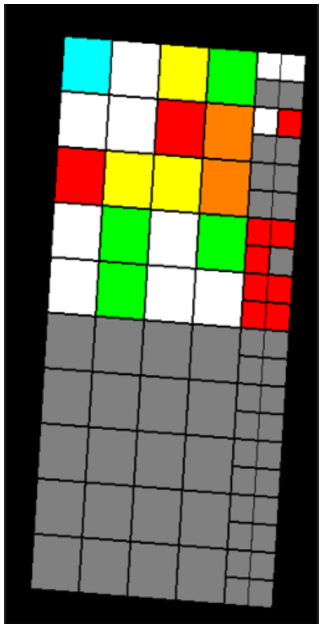
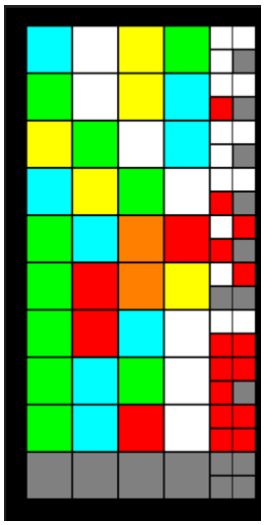
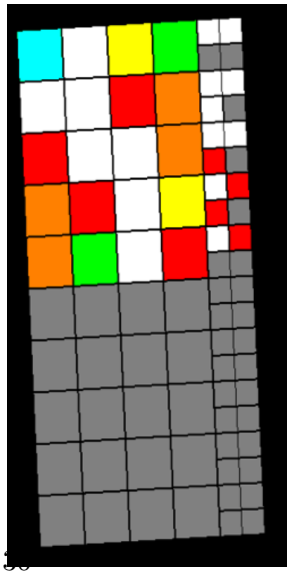


Obrázek 5.15: Poměr výkonnosti grafického vstupu s Zkreslením obrázku

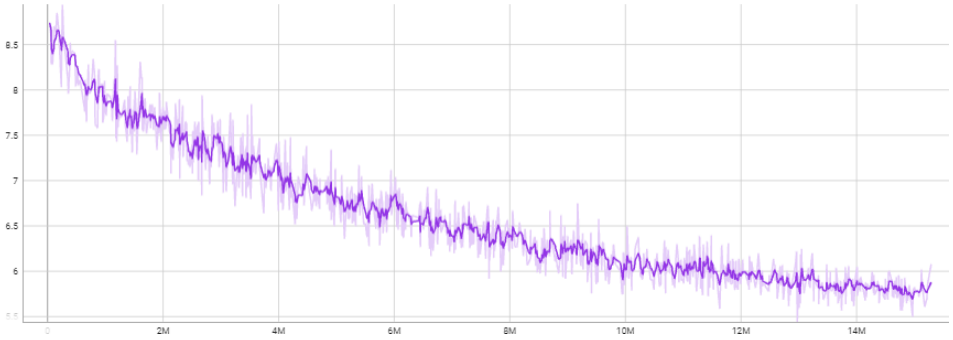
Z uvedených výsledků lze vyvodit, že s tímto zkrelsením se velmi rychle zhoršuje výkonnost. U zkreslení 200 má výkonnost jako Náhodný algoritmus.

Zkreslení 2, náhodná rotace a změna velikosti:

Při držení kamery se v reálném životě klepe ruka a není přesně srovnaná se samotnou hrou. Jako druhý typ zkreslení jsem si zvolil posouvání, zmenšování a otáčení náhodně celé herní plochy.

Příklady zkreslení	Grafická reprezentace
Příklad 1:	
Příklad 2:	
Příklad 3:	

Tabulka 5.3: Příklady zkreslení 2.



Obrázek 5.16: Učení na zkreslených datech, Osa X: Počet episod, Osa Y: Průměrná délka tahu za posledních 100 her

Výkon algoritmu na zkreslených datech, 5.882, je poměrně s vstupem bez zkreslení jen o 0.28 horší, což na takto moc zhoršená data je velmi dobrý výsledek.

Kapitola 6

Závěr

V této práci se podařilo vytvořit simulátor hry Logik se symbolickým i grafickým vstupem. Simulátor jsem použil k natrénování strojového hráče a k porovnání jeho výkonu s existujícími algoritmy, které pro tuto hru vymysleli lidé. Pro porovnání jsem například použil známý Knuthův algoritmus, hrající hru Logik téměř optimálně. Pro naučení strojového hráče jsem použil metody posilovaného učení. Z velké řady existujících algoritmů pro posilované učení se mi podařilo vybrat jeden, PPO, který se ukázal jako nejvhodnější pro prostředí hry Logik. Pro zprovoznění PPO algoritmu bylo třeba naladit jeho hyper-parametery a hlavně navrhnout vhodnou odměnovou funkci (reward function), která umožňuje naučit hráče s dobrým výkonem. Vhodná volba odměnové funkce se ukázala jako zásadní, pro úspěšné použití PPO algoritmu. Zjistil jsem, že triviální odměnové funkce, které vůbec nepoužívají lidskou znalost, nezaručují konvergenci v přijatelném čase. Experimentálně jsem nalezl odměnovou funkci, která na základě jednoduchých pravidel umožní PPO algoritmu rychle nalézt dobrou strategii hraní i přes velký akční prostor, který hra Logik má. Pomocí PPO se mi podařilo natrénovat strojového hráče, který na menších konfiguracích hry Logik dosahuje výkonu blízkou nejlepším metodám a na velkých instancích je přibližně jen o 20% horší. Ukázal jsem, že PPO algoritmus lze použít i pro natrénování hráče pro Logik s grafickým vstupem, tj. kdy hráč dostává stejný vstup jako člověk, tedy pouze obrázek. Hráč naučený pomocí PPO dokáže současně rozpoznávat stav hry z obrázku a hrát tahy s rozmným výkonem. Ukázal jsem, že učící algoritmus funguje i na obrázky zatížené různým typem zkreslení.

V mých experimentech se ukázalo, že algoritmy posilovaného učení jsou velmi robustní, pokud se použijí na prostředí s malým prostorem stavů a akcí. Pro malé instance většinou dostačuje trivilání odměnové funkce, která

odměňuje pouze dosažení cílového stavu, tj. prolomení hádaného kódu. U větších prostředí je zásadní vhodná volba odměnové funkce stejně jako naladění hyperparamterů, bez kterého algoritmy většinou konvergují.

Dále jsem zjistil, že pro symbolickou variantu hry Logik je nejvýkonější (ve smyslu průměrného počtu tahu k dokončení hry) Knutův algoritmus, který je ale současně výpočetně nejnáročnější. Lehce nižší výkonost má Eliminační algoritmus, jehož výpočetní nároky jsou ale o několik řádů nížší. Eliminační algoritmus je tedy vhodný pro velké instance hry, nebo v případě méně výkonného harwdare a vysokých nároků na rychlost hráče. Výkon hráče naučeného pomocí PPO nedosahuje úrovně Knutova algoritmu, ale velmi se jí blíží.

Učení hráče pomocí posilovaného učení, lze použít i pro grafickou variantu hry Logik narozdíl Knutova nebo Eliminačního algoritmu, které vyžadují symbolický vstup. Rozšířit symbolické algoritmy pro hraní grafické verze není triviální. Přístup postavený na převodu (rozpoznání) grafického vstupu na symbolickou reprenezetaci, např. pomocí neuronové sítě, by musel fungovat zcela bezchybně, protože symbolické algoritmy nejsou vůči chybám robustní. Chybný vstup může vést k nepředvídatelnému chování symbolického algoritmu, např. Knuthův nebo Eliminační algoritmus v případě chybně zadaného vstupu nemusí vygenerovat další tah. Naproti tomu strojový hráč naučený pomocí posilovaného učení, vždy vygeneruje další tah, který bude v nejhorším případě neoptimání.

Literatura

[1] Weisstein, Eric W. "Mastermind."From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/Mastermind.html

[2] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

[3] arXiv:1712.01815

[4] Arulkumaran, K.; Cully, A. & Togelius, J. (2019), 'AlphaStar: An Evolutionary Computation Perspective'

[5] Boric, S., Schiebel, E., Schlögl, C., Hildebrandt, M., Hofer, C. and Macht, D.M., 2021. Research in Autonomous Driving – A Historic Bibliometric View of the Research Development in Autonomous Driving. International Journal of Innovation and Economic Development, 7(5), pp.27-44.

[6] arXiv:2208.04511

[7] Reinforcement Learning: An Introduction (2018), ISBN 9780262039246

[8] Mastering Reinforcement Learning with Python (2020) ISBN 9781838644147

[9] Machine learning, The art and science of algorithms that make sense of data (2012), ISBN 9781107422223

[10] Neural Networks from Scratch (2020), ISBN 97824164224345

[11] Knuth, Donald (2011). Selected papers on fun and games. Center for the Study of Language and Information. p. 226. ISBN 9781575865843.

[12] Koyama, Kenji; Lai, Tony (1993). "An Optimal Mastermind Strategy". Journal of Recreational Mathematics (25): 230–256

[13] De Bondt, Michiel C. (November 2004), NP-completeness of Master Mind and Minesweeper, Radboud University Nijmegen

[14] Russell, Stuart J.; Norvig, Peter (2010). Artificial intelligence : a modern approach (Third ed.). Upper Saddle River, New Jersey. pp. 830, 831. ISBN 978-0-13-604259-4.

[15] Introducing OpenAI, December 12, 2015.

[16] arXiv:1707.06347

[17] stable-baselines3, Antonin Raffin and Ashley Hill and Adam Gleave and Anssi Kanervisto and Maximilian Ernestus and Noah Dormann, Stable-Baselines3: Reliable Reinforcement Learning Implementations, 2021, http://jmlr.org/papers/v22/20-1364.html

[18] Openai gym, Brockman, Greg and Cheung, Vicki and Pettersson, Ludwig and Schneider, Jonas and Schulman, John and Tang, Jie and Zaremba, Wojciech (2016) arXiv:1606.01540

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Blažek** Jméno: **Matěj** Osobní číslo: **492080**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Hráč hry Logik učený z příkladů

Název bakalářské práce anglicky:

Player of Mastermind learned from examples

Pokyny pro vypracování:

Seznam doporučené literatury:

- Sutton, R. S. and Barto, A. G. Reinforcement Learning: An Introduction. The MIT Press. 2018.
- Goodfellow, I. and Bengio, Y. and Courville A. Deep Learning. The MIT Press. 2016.
- Koyama, K. and Lai, T. W. 'An Optimal Mastermind Strategy.' J. Recr. Math. 25, 251-256, 1993.
- Knuth, D. E. 'The Computer as a Master Mind.' J. Recr. Math. 9, 1-6, 1976-77.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Vojtěch Franc, Ph.D. Strojové učení FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **13.02.2023** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **22.09.2024**

Ing. Vojtěch Franc, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta