TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Využití strojového učení pro hraní videoher

Bakalářská práce

Studijní program: B2646 Informační technologie
Studijní obor: Informační technologie

Autor práce: **Martin Tvrdík**
Vedoucí práce: Ing. Karel Paleček, Ph.D.
Ústav informačních technologií a elektroniky

Liberec 2021

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání bakalářské práce

Využití strojového učení pro hraní videoher

Jméno a příjmení: **Martin Tvrdík**
Osobní číslo: M16000061
Studijní program: B2646 Informační technologie
Studijní obor: Informační technologie
Zadávající katedra: Ústav informačních technologií a elektroniky
Akademický rok: **2020/2021**

Zásady pro vypracování:

1. Seznamte se s problematikou neuronových sítí, reinforcement learningu a neuroevolučních algoritmů.
2. Na základě rešerže implemetujte vybrané zástupce metod reinforcement learningu a neuroevoluce.
3. Vybrané metody aplikujte a porovnejte ve virtuálním prostředí zvolených her implementovaných v jazyce Python.
4. Zhodnoťte výhody a nevýhody obou přístupů a diskutujte možné praktické využití v reálných aplikacích.

Rozsah grafických prací:	Dle potřeby dokumentace
Rozsah pracovní zprávy:	30-40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] GOODFELLOW, I., Bengio, Y., Courville, A. Deep learning. MIT Press, 2016
[2] BOSHOP, C. Pattern Recognition and Machine Learning. 2006. ISBN 13:978-0387310732
[3] Online kurz „Reinforcement Learning“,
https://eu.udacity.com/course/reinforcement-learning–ud600

Vedoucí práce:	Ing. Karel Paleček, Ph.D. Ústav informačních technologií a elektroniky
Datum zadání práce:	19. října 2020
Předpokládaný termín odevzdání:	17. května 2021

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 19. října 2020

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

16. května 2021 Martin Tvrdík

Využití strojového učení pro hraní videoher

Abstrakt

Práce se zabývá problematikou neuronových sítí ve spojení se zpětnovazebním učením a genetickými algoritmy (neuroevoluce). Tyto dvě rozdílné metody přístupů budou implementovány na jednoduchém virtuálním prostředí atari her. Hlavním výstupem práce je implementace a porovnání těchto dvou zmíněných metod a vyvození závěrů. Na základě tohoto výstupu bude diskutováno potenciální využití těchto dvou diametrálně odlišných přístupů v reálných aplikacích a jejich hlavní výhody a nevýhody.

Klíčová slova: neuronové sítě, umělá inteligence, zpětnovazební učení, agent, genetické algoritmy, neuroevoluce

Using machine learning to play video games

Abstract

Thesis is about problematic of neural networks in connection with reinforcement learning and genetic algorithms (neuroevolution). These two different methods will be implemented on simple virtual environment of atari games. The main output of this thesis is comparison of those two methods to draw conclusions. We will discuss potential usage of those two methods in the real world applications and their pros and cons based on outcome of our experiment.

Keywords: neural networks, artificial intelligence, reinforcement learning, agent, genetic algorithms, neuroevolution

Poděkování

Tímto bych chtěl poděkovat vedoucímu mé bakalářské práce, kterým je Ing. Karel Paleček Ph.D., za poskytnutí cenných rad ohledně zpracování práce a především za jeho pomoc při finálním porovnávání metod na provedených experimentech. Zároveň bych mu chtěl poděkovat i za vstřícnost a pomoc s odborným textem.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

AI	Artificial intelligence
DDQN	Double Q-learning
DNN	Neural Network
DQN	Deep Q-learning
FF	Feedforward
FFC	Fully Connected Layers
GA	genetický algoritmus
MSE	Mean Square Error
NEAT	NeuroEvolution of Augmenting Topologies
NN	Neural Network
RL	Reinforcement learning
rtNEAT	real-time NEAT
SARSA	State–action–reward–state–action
SGD	Stochastic Gradient Descent

1 Úvod

Již delší dobu se zajímám o problematiku umělé inteligence a neuronových sítí. Osobně si myslím, že jsme na začátku probíhající technologické revoluce a vidím v tom obrovskou budoucnost ve všech možných odvětvích a to nejen těch technických. Z toho důvodu jsem si vybral téma, které zahrnuje tuto problematiku. Chci se pokusit nějaký takový problém sám vyřešit a samozřejmě se také o celé této problematice dozvědět více informací.

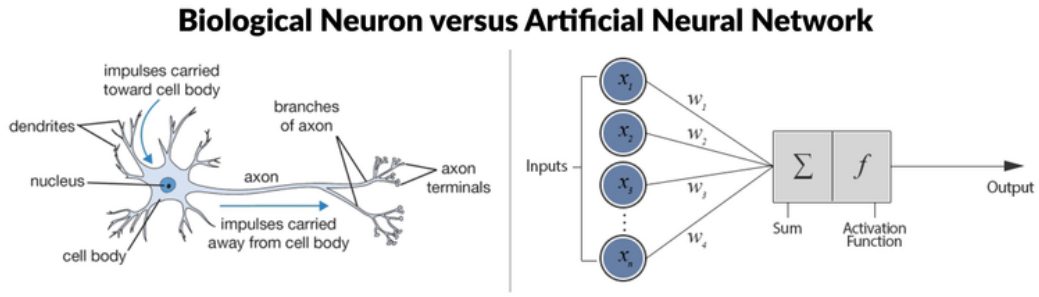
Téma o hraní videoher jsem si vybral nakonec z toho důvodu, že se jedná o plně virtuální prostředí a nemusím řešit žádné speciální vybavení a vše lze řešit plně programově. Dále vídím, že se tento průmysl (video hry) stejně jako všechno ostatní neustále vyvíjí a je zde požadavek i na vyšší umělou inteligenci v prostředí, která byla donedávna téměř výhradně řešena pouze formou předem nascriptovaných událostí, dialogů či reakcí a nad tento rámec nebyla vůbec interaktivní. Zapojením neuronové sítě lze toto v určitém smyslu zlepšit a rozšířit. V této práci bohužel nemám dostatek prostoru k dosažení nějakého zásadního objevu, ale rád bych alespoň dokázal, že lze tyto formy pro tvoření umělé inteligence ve hrách úspěšně použít. Následně bych na základě zjištění z této práce rád stavěl v další navazující práci.

V rámci problému bych proti sobě rád postavil dva zcela odlišné přístupy a porovnal jejich výhody a nevýhody. To bude také hlavním výstupem této práce. Implementuji jednoho zástupce zpětnovazebního učení což je jeden z hlavních zástupců strojového učení vedle učení s učitelem a učení bez učitele. Zde síť bude samotnou hru hrát a budeme ji učit na předchozích zkušenostech formou poskytování zpětné vazby (odměny) a půjde především o to právě tuto odměnu maximalizovat. Druhým alternativním přístupem pak bude využití genetických algoritmů a neuroevoluce. Zde se pokoušíme napodobit evoluci tak jak ji známe z biologie a necháváme náhodné sítě soupeřit mezi sebou. Následně jsme díky optimalizačním genetickým algoritům schopni vybrat nejvhodnější jedince a provést nad nimi evoluce a vytvořit tak novou generaci jedinců, kteří by měli v daných videohrách dosahovat lepších výsledků než generace předchozí.

Součástí práce bude také diskuze výhod a nevýhod těchto dvou přístupů a možná reálná aplikace v jiných odvětvích na základě těchto zjištění. Hlavním zdrojem informací pro tuto práci jsou vědecké články, které se již podobnou problematikou v minulosti zabývaly.

2 Neuronové sítě

Neuronové sítě (anglicky Neural Networks), často také nazývané umělé neuronové sítě (Artificial Neural Networks) jsou pokročilou aplikací strojového učení. Jedná se o kolekci uzlů, kterým říkáme umělé neurony a vazeb mezi nimi. Tyto umělé neurony následně kolektivně činí rozhodnutí na základě vstupních parametrů, vah a biasu. Tento proces je výrazně inspirován způsobem jakým funguje lidský mozek. Zde si biologické neurony podobným způsobem předávají informace o vstupním signálu pomocí takzvaných synapsí a na jejich základě činí svá rozhodnutí. [1]



Obrázek 2.1: Porovnání umělého a biologického neuronu [2]

2.1 Struktura neuronové sítě

Každý umělý neuron má jeden a více vstupů, ale vždy pouze jeden výstup, který může být poslán do libovolného počtu dalších neuronů. Vstupy jsou pronásobeny váhami a sečteny k čemuž je dále přičten bias. Na tento výsledek je následně aplikována tzv. aktivační funkce, která zajišťuje nelinearitu. Výsledek z této funkce je zaslán jako vstup jednomu či více dalších neuronů. Váhy jsou parametrem při trénování a jsou tím hlavním co určuje jak bude síť reagovat na vstupní data. Trénováním neuronové sítě je tedy myšleno nalezení takových parametrů (vah), abychom dosáhli co nejoptimálnějšího řešení daného problému.

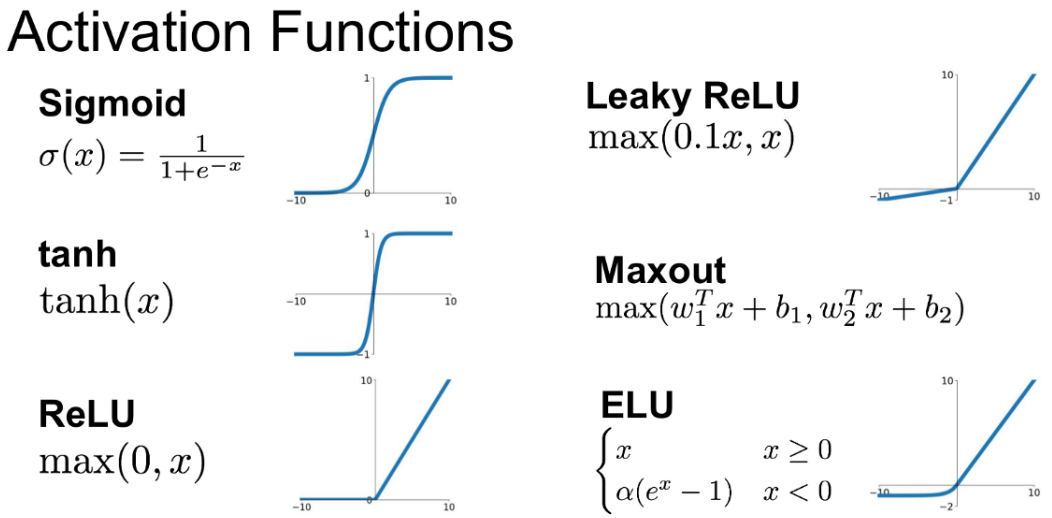
Neurony se umisťují do vrstev. Zpravidla máme vždy jednu vstupní vrstvu, která slouží jako mediátor a přijímá vstupní data, která by v ideálním případě měla být

předzpracovaná a normalizovaná před tím než se zašlou jako vstup do sítě. Následně máme libovolný počet skrytých vrstev. Dle počtu těchto skrytých vrstev se můžeme bavit o tom jestli se jedná o hlubokou neuronovou síť (Deep Neural Network). Poslední vrstvou v síti je výstupní vrstva. Tato vrstva standardně výstup prožene skrz distribuční funkci, která nám výsledné číslo namapuje na jednu či více pravděpodobností. V případě, že bychom například řešili problém klasifikace obrázku do více než dvou tříd, tak bychom ve většině případů použili funkci softmax, která by nám na výstupu vrátila s jakou pravděpodobností obrázek patří do jednotlivých tříd. Z toho již lze jednoduše vybrat maximální pravděpodobnost a zařadit obrázek do třídy s nejvyšší pravděpodobností. Pokud bychom řešili binární problém tak by bylo zase vhodné použít například funkci sigmoid.

2.2 Aktivační funkce

Aktivační funkce jsou nezbytnou součástí modelu neuronové sítě. Mají vliv na rychlost učení a výslednou přesnost modelu po trénování na datasetu [3]. V neuronové síti zaručují nelinearitu, která je důležitá z toho důvodu, aby výsledek sítě nebyl lineární funkcí, ale pokryl celý prostor možností.

Nejčastěji se setkáme s funkcemi ReLu a sigmoid, ale v posledních letech se objevují stále nové aktivační funkce, které za velmi specifických okolností mohou dosahovat výrazně lepších výsledků [3]. Ve většině případů se jedná o ojedinělé experimenty se specifickým datasetem. Z toho důvodu se budu v práci držet především funkcí ReLu a sigmoid, protože se jedná již o obecně prověřené funkce na všech různých typech datasetů.



Obrázek 2.2: Ukázka aktivačních funkcí [4]

2.3 Typy neuronových sítí

Neuronové sítě lze rozdělit mnoha různými způsoby. V této kapitole bych chtěl stručně popsat co jsou to konvoluční sítě, dopředné sítě (FF) a sítě s plně spojenými vrstvami (FCN). Tyto sítě popisuji z toho důvodu, že je budu v práci využívat a budu je tedy v textu práce i poměrně často zmiňovat.

2.3.1 Konvoluční síť

Konvoluční síť se používá převážně u problémů, kde pracujeme s obrázkem jako vstupem do sítě. Používá se zde tedy matematická konvoluce z čehož vychází název tohoto typu sítě. Konvoluce síti umožňuje zkoumat různé menší části obrázku izolovaně a hledat prvky a podobnosti pouze v určitém regionu obrazu.

2.3.2 Dopředná síť

Dopředná síť (feedforward) je taková síť, kde všechny vazby vrstvy v síti směřují do jedné z následujících vrstev v pořadí a síť tedy neobsahuje žádnou rekurentní vazbu. Rekurentní síť je pak tedy logickým opakem dopředné sítě. Dopředná síť se využívá více obecně. Rekurentní sítě umožňují držení paměti a jsou vhodné pro zpracování sekvenčních dat jako je například zpracování řeči.

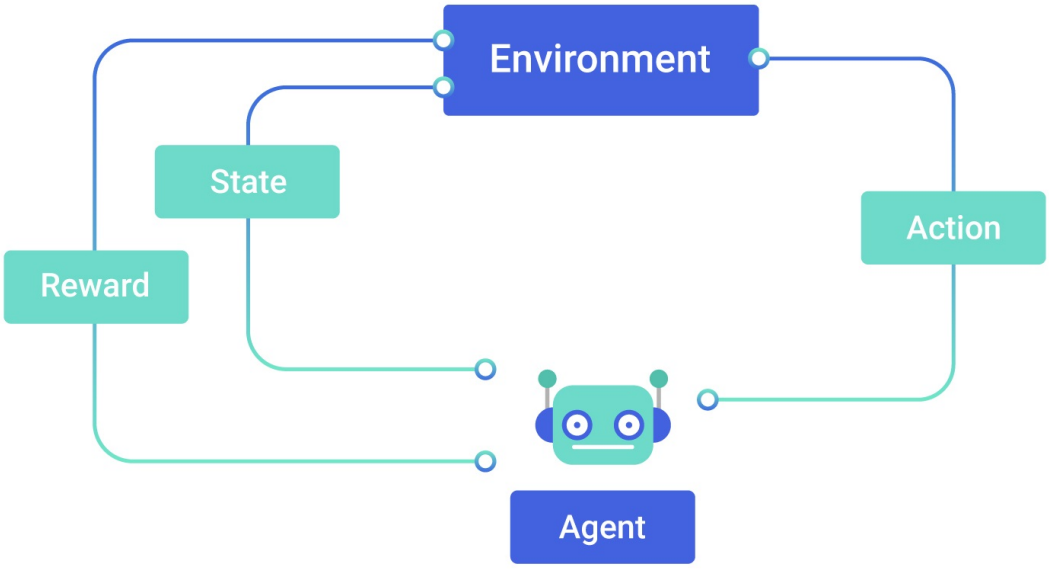
2.3.3 Síť s plně spojenými vrstvami

Tento typ sítě znamená, že všechny neurony ve vrstvě jsou spojené se všemi neurony ve vrstvě nadcházející. Pro splnění podmínky musí toto platit pro všechny vrstvy v síti. Se sítí s plně spojenými vrstvami budu pracovat především u problémů se zpětnovazebním učením. U genetických algoritmů budu naopak pracovat s nespojitými vrstvami, protože filozofií řešení není trénování a síť tedy netrénujeme. Naopak začínáme s nespojitou sítí a postupně přidáváme spojení mezi neurony v jednotlivých iteracích a sledujeme jak si výsledná síť vede.

3 Zpětnovazební učení

Zpětnovazební učení (reinforcement learning) je jedním ze tří základních odvětví strojového učení společně s učením s učitelem (supervised learning) a učením bez učitele (unsupervised learning). Princip spočívá v tom, že agent se orientuje ve virtuálním či reálném prostředí a sbírá zkušenosti. Následně se ze svých minulých zkušeností zpětně učí a to takovým způsobem, aby maximalizoval kumulativní odměnu v prostředí. Toto odvětví strojového učení je opět inspirováno tím jakým způsobem funguje mozek a učení u lidí i zvířat, kde mozek podobným způsobem reaguje na negativní či pozitivní podněty a tím se učí [5].

Základním stavebním kamenem je samotná interakce agenta s prostředím. Prostředí je reprezentováno aktuálním stavem (stavem může být například obrázek), který je prezentován agentovi. Agent na základě tohoto stavu provede akci. Následně prostředí přejde do nového stavu a dojde k vyhodnocení zda bude agent odměněn či penalizován za provedenou akci. Celý proces je názorně ukázán na diagramu (3.1).



Obrázek 3.1: Princip zpětnovazebního učení [6]

Přístup je unikátní v tom jakým způsobem jsou získávána data pro učení. Narozdíl od učení s učitelem i učení bez učitele nedodáváme agentovi žádný vstupní dataset. Při učení s učitelem je nutné dodat páry vstupních a výstupních dat. Hledáme tedy podobnosti na základě již existujících označkovaných dat. U učení bez učitele naopak máme pouze vstupní data a hledáme libovolné podobnosti mezi vstupními daty (třídění dat do skupin). Zde je velmi často nutné, aby výsledky vyhodnotil člověk, který se pohybuje v odvětví řešeného problému.

U zpětnovazebního učení je přístup úplně jiný a žádný dataset nepotřebujeme. Agent si data sbírá sám svou vlastní interakcí s prostředím a učí se ze svých chyb. Dochází k postupnému vyrovnávání poměru náhodných akcí (exploration) a vlastními rozhodnutími na základě naučeného chování z předchozích zkušeností (exploitation).

3.1 Agent

Agentem je ve smyslu zpětnovazebního učení myšlen program, který na základě stavu a předchozích zkušeností činí nová rozhodnutí. Jeho hlavním cílem je maximalizovat budoucí kumulativní odměnu prováděním optimálních akcí. Tím pádem se agent snaží provádět optimální akce, které mu zaručí co nejvyšší odměnu a zároveň se snaží předcházet akcím, které by tuto odměnu naopak snižovaly.

3.2 Metody zpětnovazebního učení

3.2.1 Model-Based

Modelový přístup je specifický v tom, že vyžaduje modelování prostředí a tento model prostředí následně přímo trénujeme. V rámci prostředí lze pak plánovat akce dopředu a model je schopný sekvenci budoucích akcí naplánovat ještě před provedením jakékoli akce. Toto je největší výhodou tohoto přístupu. V ostatních metodách nás vždy zajímá pouze aktuální stav na jehož základě provedeme jednu budoucí akci. Nevýhodou přístupu je, že agent je pouze tak dobrý jako naučený model prostředí, který je složitější k naučení. Model také nikdy nebude univerzální a je tedy nutné navrhnout nový model pro každé prostředí.

Výhody a nevýhody přístupu jsou podrobněji zmiňovány ve studii publikované na researchgate [7]. Zde se mimo jiné píše například o tom, že v reálné aplikaci (např. učení robota) je u této metody mnohem nižší riziko, že dojde k poškození robota nebo prostředí. To může být jedním ze zásadních důvodů proč zvolit právě tuto metodu v reálné aplikaci, kde se operuje s drahým vybavením. V této práci se nicméně budu zabývat čistě virtuálním prostředím a není to pro mě relevantní. Zároveň chci, aby

výsledný algoritmus byl co nejvíce univerzální při použití napříč různými hrami. Proto se touto metodou nadále v práci již nebudu zabývat.

3.2.2 Policy-Based

V těchto metodách se učíme tzv. policy funkci, která mapuje stav na danou akci takovým způsobem, aby se maximalizovala odměna.

$$\pi(a|s,\theta) = P_r\{A_t = a|S_t = s, \theta_t = \theta\}$$

Policy ve vzorci značená jako π je tedy pravděpodobnost, že se provede akce a ve stavu s při parametrech θ. Cílem je nalezení optimálních parametrů θ. Výhodou této metody je, že lépe konverguje a může se dobře naučit i stochastické politiky což by mohlo být významné například při hře, která je určená pro dva hráče a model hraje proti člověku. Nevýhoda je naopak v tom, že metoda má tendenci konvergovat do lokálního optima [8]. Přístup jsem nezvolil z toho důvodu, že modely budu učit pouze na videohrách hraných proti počítači a proto jsou do určitého stupně deterministické. Také mám obavy z toho, že by mi algoritmus konvergoval k lokálnímu optimu a agent by se naučil řešit jeden specifický problém v rámci hry, který by ale nebyl tím nejoptimálnějším řešením pro dosažení co nejlepšího celkového výsledku v dané hře.

3.2.3 Value-Based

Zde se učíme reagovat přímo na stav či pár stav-akce. Následně vybíráme novou akci, od které očekáváme nejvyšší odměnu. Tuto metodu jsem zvolil pro svou práci. Výběr této metody pro RL by neměl výsledky v žádném případě negativně ovlivnit a poskytuje vše co pro své experimenty potřebuji a je pro mě tedy zcela dostačující. Je také výrazně jednodušší na implementaci a samotné pochopení toho jak celý algoritmus funguje. Níže uvádím příklad základní value funkce. Kromě této funkce existuje celá řada dalších funkcí zájmu (tzv. function of interest) jako jsou například Q value nebo Advantage function. Obě tyto funkce vycházejí z níže zmíněné V-Value funkce.

$$V^{\pi}(s) = E\left[\sum_{k=0}^{\inf} \gamma^k r_{t+k} | s_t = s, \pi\right]$$

E je ve vzorci očekávaná hodnota, γ je tzv. discount faktor, který určuje jak vysokou váhu budou mít akce provedené v daleké minulosti na budoucí odměnu. π je zvolená politika, která nicméně nemá žádnou souvislost s policy-based přístupem [9]. Jedná se o to jakým způsobem agent vyvažuje exploration a exploitation, který je v rámci tohoto přístupu nezbytný a je blíže vysvětlen v následující kapitole.

Očekávaná optimální hodnota (nejedná se o odměnu) je pak definována následujícím vzorcem.

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

3.3 Průzkum prostředí

Value-Based metoda zpětnovazebního učení, kterou jsem vybral v předchozí kapitole je postavená na kompromisu průzkumu prostředí (náhodné akce) a vlastním rozhodnutím agenta - jedná se o tzv. exploration vs. exploitation trade-off. Většina modelů je pak postavených na tom, že na začátku učení probíhá průzkum prostředí velmi často. Postupem času během učení agenta dochází k postupnému snižování frekvence náhodných akcí až do bodu, kdy je šance na provedení náhodné akce téměř nulová.

Provádění náhodných akcí je při učení agenta důležité z toho důvodu, aby byl agent vystaven co největšímu počtu různých stavů. Díky tomu se agent může naučit reagovat a provádět optimální akce v prostředí. Tento předpoklad vychází z toho, že agent nedostane k dispozici žádný iniciální dataset a je odkázaný pouze na svou vlastní interakci s prostředím. Přístup by se dal rozdělit na metody tzv. greedy approach a random approach. V prvním případě by agent vždy provedl jím zvolenou akci. Ve druhém případě by se vždy provedla náhodná akce. Obě tyto metody jsou extrémními příklady, které se v RL snažíme vyvažovat [10].

3.3.1 Epsilon-greedy

Jedná se o jednoduchou metodu, která kombinuje greedy a random approach a to takovým způsobem, že zavádí hyperparametr ϵ. Agent provádí průzkum prostředí v náhodných intervalech s pravděpodobností ϵ. V ostatních případech provede vlastní akci s pravděpodobností $1 - \epsilon$. Parametr ϵ standardně začíná na větším čísle, které může na začátku trénování být například 1. Náhodnou akci tedy zpočátku provedeme vždy. Postupem času pak programově snižujeme ϵ na nižší hodnotu, která se ke konci trénování může blížit například hodnotě 0.05. Nevýhodou této metody je, že agent bere v potaz pouze akce, od kterých v dané chvíli očekává největší odměnu a zbytek vždy ignoruje.

3.3.2 Boltzmann Approach

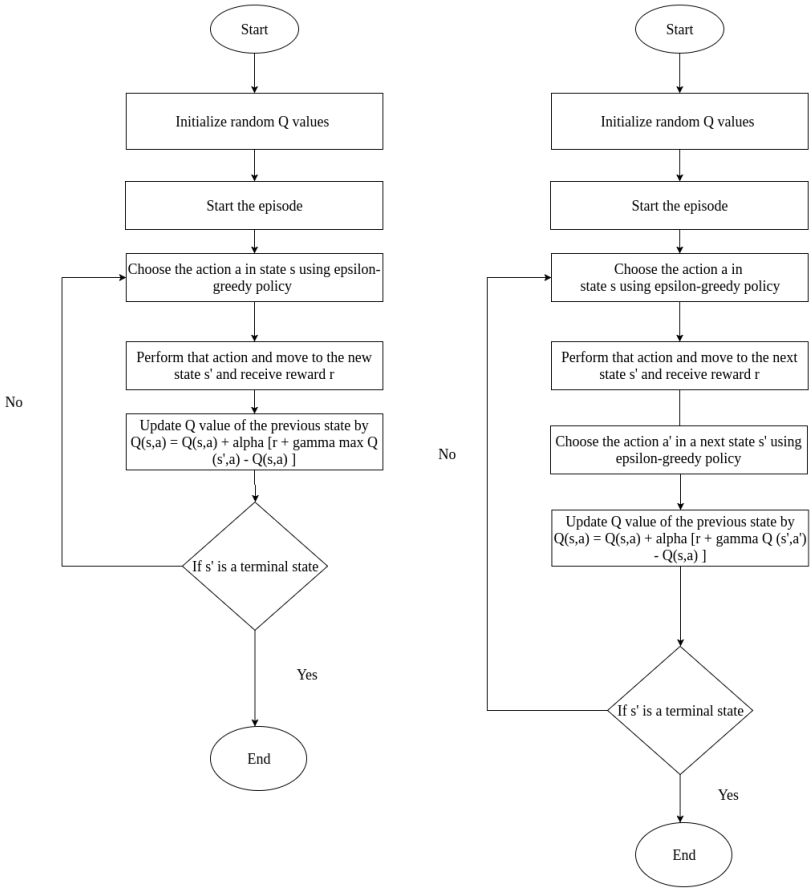
V této metodě neprovádíme akce zcela náhodně a ani nevybíráme nejoptimálnější akci. Místo toho přiřadíme všem akcím pravděpodobnost na základě toho jakou odměnu od nich agent očekává. Takového chování lze docílit například aplikováním funkce softmax přes množinu očekávaných odměn v případě provedení každé jednotlivé akce. Hlavní nevýhodou metody je existující logická nepřesnost. Agent má k dispozici pouze data o tom jak moc optimální daná akce je. To neznamená, že si je s takovou pravděpodobností také jistý, že daná akce je ta nejoptimálnější. Takovou informaci agent k dispozici nemá.

Existuje mnoho dalších metod a studijí, které se touto problematikou blíže zabývají a hledají nejoptimálnější metody průzkumu. Tato práce se touto problematikou nezabývá a proto zde zmiňuji pouze dvě základní a implementačně jednoduché metody. Obě dvě metody jsou srovnatelné. Ve své práci pro RL jsem vybral a následně implementoval epsilon-greedy metodu a to z toho důvodu, že se jedná o častěji používanou a přímočařejší metodou, která je jednodušší na implementaci a následnou manipulaci parametru během učení.

3.4 Value-Based algoritmy

V této kapitole se budu bavit o dvou hlavních (nejčastěji používaných) algoritmech při využití Value-Based metodiky. Prvním z nich je Q-learning, který je zástupce tzv. off-policy učení a druhým z nich je SARSA (on-policy učení). Zde je nutno zmínit, že se nejedná o žádnou souvislost s Policy-Based přístupem. Jedná se o to jakou politiku algoritmus učení využívá pro predikci nejlepších akcí a následně během procesu trénování (např. aktualizace Q-values). Pokud je tato policy funkce (např. epsilon-greedy) v obou případech stejná tak se jedná o on-policy učení, v opačném případě se pak jedná o off-policy učení [11].

Oba algoritmy mají výhody, které mohou být v různých situacích přínosem, ale také přítěží. SARSA je například konzervativnější v tom smyslu, že pokud narazí na případ, kdy je riziko negativní odměny (penalizace) velké, tak se do takového stavu vůbec nedostaneme. Q-learning by naopak takový stav zaznamenal a byl schopný se z něj následně učit. Off-policy učící algoritmy budou mít obecně větší zkoumaný prostor, kdežto SARSA se takovým případům raději vyhne. To může mít za následek ignorování potencionální nejoptimálnější cesty k řešení. SARSA by tedy byl vhodnější algoritmus v případě, že bychom algoritmus aplikovali v reálném světě, kde by mohly vzniknout finanční škody z důvodu poškození drahého vybavení (např. robota). V mém případě nic takového nehrozí a širší zkoumaný prostor beru spíše jako pozitivní přínos. To je hlavní důvod proč jsem si pro svou práci zvolil algoritmus Q-learningu. Popisovaný rozdíl mezi algoritmy je vidět na obrázku 3.2.



Obrázek 3.2: Porovnání algoritmů: Q-learning (nalevo), SARSA (napravo) [12]

3.4.1 Q-learning

Tato off-policy metoda vyhodnocuje tzv. Q value. Jedná se o hodnotu provedené akce v určitém stavu. Agent se na základě algoritmu tuto hodnotu pokouší co nejlépe predikovat tak, aby maximalizoval kumulativní odměnu [13].

$$Q^{\pi}_{t+1}(s_t, a_t) = (1 - \alpha)Q^{\pi}_{t}(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a} Q^{\pi}_{t}(s_{t+1}, a) \right)$$

Ze vzorce je vidět, že se jedná o predikci nové optimální hodnoty na základě stavu a všech možných akcí. Tato hodnota je vynásobena tzv. discount faktorem γ, který určuje jak moc jsou minulé akce relevantní v rámci aktuálního a budoucího stavu. Následně dojde k připočtení odměny r a přenásobení učícím parametrem α. Tento výsledek sečteme s hodnotou z předchozího stavu přenásobenou doplňkem k α a dostaneme soubor Q values v jednotlivých akcích a stavech čemuž říkáme Q table. Ukázkové rozložení Q table (se smyšlenými hodnotami) je k nahlédnutí v tabulce 3.1 níže.

Tabulka 3.1: Ukázkové rozložení Q table

	a_1	a_2	a_3	$a_{..}$	a_n
s_1	5	10	8	..	6
s_2	12	-3	6	..	-1
s_3	12	-3	6	..	8
$s_{..}$	..	..	..	..	..
s_n	1	8	2	..	-2

Q table se používá pouze pokud nechceme používat neuronové sítě. V opračném případě mluvíme o Deep Q Learning (DQN). Tabulka na jejímž základě agent doposud činil všechna rozhodnutí je v takovém případě nahrazena neuronovou sítí. V této práci se chci zabývat neuronovými sítěmi a proto se budu zabývat touto upravenou variantou, která neuronové sítě obsahuje.

DQN

Jak již bylo zmíněno výše, tak DQN je Q learning, kde je Q table nahrazena neuronovou sítí. To samotné ale nestačí, protože ve chvíli kdy odebereme tabulku, tak agent nemá z čeho se učit. Proto zavádíme tzv. experience replay. Buffer kam ukládáme určitý počet předchozích stavů. Agent buffer následně využívá jako dataset, ze kterého se učí. Celý proces je velmi dobře vysvětlen v originálním DQN článku, kde byla celá tato metoda představena [14]. Autoři zde podobně jako já prováděli experimenty na atari hrách v prostředí Gym od OpenAI a představili právě tuto metodu, která je nyní uznávaná jako jeden ze standardních postupů v RL.

DDQN

Jedná se o vylepšenou verzi DQN (Double DQN). Metoda si klade za cíl vyřešit nedostatky DQN, kde v některých případech mohly vzniknout nepřesnosti z důvodu statistických chyb a přecenění či podcenění některých stavů. Z toho důvodu zavádíme druhou neuronovou síť. Jedna síť je pak neustále učena, ale výslednou akci vybíráme ze sítě druhé na základě indexů získaných ze sítě první. Váhy v cílové síti se neaktualizují každou iteraci, ale vždy periodicky po více iteracích což zase zajišťuje lepší stabilitu učení. Více informací lze dohledat v článku, který problémy metody DQN vysvětluje a představuje metodu DDQN. [15].

DDQN je metoda, kterou budu implementovat ve své práci v části pro RL. Metoda vyhovuje všem požadavkům a řeší problémy, na které bych mohl narazit. Také splňuje to, že se v rámci řešení používají neuronové sítě. Po provedení vlastních experimentů na vybraných atari hrách při použití DDQN budu výsledky porovnávat s výsledky z neuroevoluce na metodě, kterou zvolím na základě druhé části rešerše.

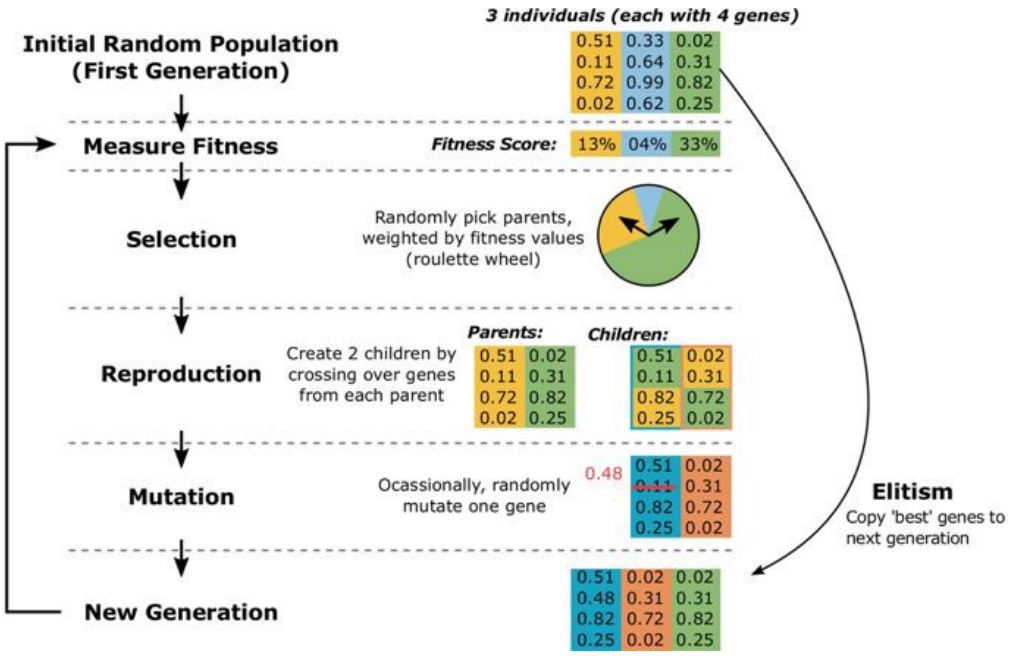
4 Neuroevoluce

Neuroevoluce je forma umělé inteligence, kde generujeme neuronové sítě. Sítě spolu následně soupeří a probíhá výběr nejvhodnějších jedinců. Toho docílíme právě pomocí evolučních algoritmů, které vybírají nejvhodnější kandidáty z populace a vyvíjí je takovým způsobem, aby byli postupně schopni řešit daný problém co nejoptimálnějším způsobem. Nejčastěji pro ně najdeme využití v evoluční robotice, hraní her a nebo evolučních simulacích pro výzkumné účely [16].

Největší výhodou neuroevoluce je praktičnost použití a také mnohonásobně nižší nároky na výpočetní techniku. Můžeme ji aplikovat téměř na libovolný problém a jediné co potřebujeme vedle již existujícího algoritmu (např. NEAT) je definice tzv. fitness funkce. Tato funkce nám vrací číselnou hodnotu, která určuje jak si daný jedinec vede při řešení problému. Algoritmus není pro jednotlivé případy téměř vůbec nutné upravovat a to stejné platí o architektuře sítě. Samotná architektura sítě v tomto případě v podstatě neexistuje, protože síť se generuje na základě algoritmu náhodně a pouze se dodržují pravidla pro generování určená algoritmem. Výsledná síť se bude tedy zpravidla vždy lišit. Jediné co musíme pro jednotlivé případy měnit jsou parametry evoluce což mohou být například pravděpodobnosti, které určují jakým způsobem se síť bude generovat.

4.1 Genetický algoritmus

Genetický či evoluční algoritmus funguje na velmi jednoduchém principu. V prvním kroku vytvoříme počáteční populaci neuronových sítí. Tyto sítě necháme samovolně interagovat s prostředím a sledujeme jak si vedou. Následně určíme několik jedinců, kteří si na základě nějakého kritéria (fitness funkce) vedli nejlépe a vybereme je jako rodiče pro následující generaci. Kritérium může být například dosažené skóre ve hře. Nová generace pak bude mít podobnosti se svými rodiči a můžeme předpokládat, že někteří potomci si v některých případech povedou lépe než jejich rodiče. Takto iterujeme do doby dokud se nedostaneme k určité generaci a nebo do doby dokud nějaký jedinec z populace nedosáhne námi požadovaného výsledku. Celý proces se snaží částečně modelovat skutečný scénář genetické evoluce a přirozeného výběru. Princip vysvětleného procesu si lze prohlédnout na obrázku 4.1.



Obrázek 4.1: Ilustrace genetického algoritmu [17]

4.1.1 Populace

Termínem populace je v genetických algoritmech myšlen počet jedinců, se kterými pracujeme, a nad kterými provádíme algoritmy křížení, mutace, atd. Velikost populace je většinou určena číselnou konstantou v řádu desítek či stovek. Výhodou menší populace pak bude rychlost výpočtu. Větší populace zase pokryje širší prostor výsledků a zpravidla dosáhneme lepších výsledků při řešení problémů.

4.1.2 Fitness funkce

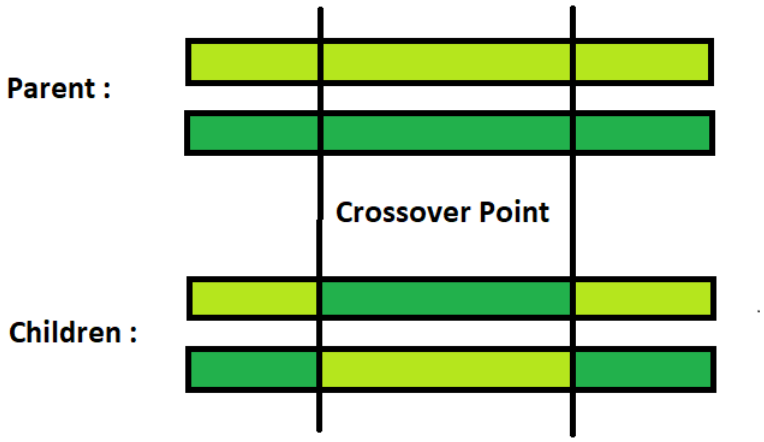
Fitness funkce nám pro každého jedince a pro každou iteraci spočítá vhodnost. Číslo nám určuje jak dobře si jedinec vede při řešení problému. Na základě tohoto čísla můžeme vybrat nejvhodnější kandidáty z populace pro další křížení a pro vznik dalších generací. Funkce bude pro každý problém jiná. Pro problematiku videoher by se mohlo obecně jednat o dosažené skóre ve hře. Případně bychom mohli každou videohru řešit zvlášť a definovat určitou kombinaci skóre, doby hraní, počtu poražených nepřátel, počet sesbíraných bodů,.. Pak bychom mohli každé z těchto charakteristik přiřadit určitou váhu a pronásobit. Viz ilustrační příklad níže. Tímto bychom mohli docílit specifického chování. Bude evoluce preferovat co nejdelší dobu přežití nebo se zaměří na získání maximálního skóre za co nejkratší dobu?

$$fitness = 0.5(score) + 0.2(time) + 0.1(enemies\_killed) + 0.1(points)$$

4.1.3 Základní principy

Křížení

Křížení (crossover) je jeden ze základních principů evolučních algoritmů. Křížení lze implementovat mnoha různě složitými způsoby. Zdaleka nejjednoduším příkladem křížení by bylo vžít dva rodiče s nejvyšší vhodností a náhodně určit bod křížení. Pak by bylo možné posouváním tohoto bodu vytvořit větší množství potomků. Bod je nutné posouvat z toho důvodu, abychom zaručili, že každý z potomků bude jiný. Ukázka procesu je k nahlédnutí na obrázku 4.2.



Obrázek 4.2: Ukázka pro dva body křížení [18]

V tomto ukázkovém příkladě je nutný předpoklad, že oba křížení jedinci mají stejnou strukturu neuronové sítě. U většiny pokročilejších algoritmů narazíme na problém, že tento předpoklad neplatí. Proto je nutné s tímto počítat a implementaci křížení implementovat tak, aby bylo možné vzít v potaz dvě odlišné struktury sítě a ty vzájemně křížit. Jedno z takových řešení (pro algoritmus NEAT) je k náhlednutí například v této práci na ResearchGate [19].

Mutace

Další metodou v genetických algoritmech je provádění náhodných mutací. Mutace může nastat u libovolného jedince v populaci, ale pravděpodobnost, že mutace nastane, je zpravidla nastavena nízko. V nejjednodušší formě může mutace znamenat, že se náhodně změní váha jednoho spojení neuronů. V pokročilejších algoritmech se bude jednat především o to, že síti přidáme nový neuron, vazbu mezi neurony či celou novou vrstvu. Nebo naopak některý z těchto prvků ze sítě odebereme. Cílem je zajistit kontinuální vývoj sítě a zamezit potencionální stagnaci v nějakém bodě.

Elitismus

Elitismus zajišťuje, že určitá malá část nejlépe si vedoucích jedinců v populaci postoupí do další generace beze změny. Tímto zajistíme, že aktuálně nejlepší evoluční cesta se v žádném případě neztratí. Ke ztrátě by mohlo dojít například během náhodných mutací a křížení pokud by všichni potomci měli v další generaci horší výsledky než jejich rodiče.

4.1.4 Shrnutí

Genetické algoritmy budou metody uvedené výše optimálně kombinovat různými způsoby. Vždy budeme chtít používat křížení a mutace s tím, že mutace se neprovede vždy, ale pouze s nějakou pravděpodobností. Elitismus se nepoužívá ve všech případech a jedná se spíše o optimalizaci dle typu problému. Abychom zajistili širší spektrum genetické informace tak v některých případech můžeme část dalších generací generovat náhodně stejným způsobem jakým jsme vytvořili generaci první. Pokud bychom měli populaci o velikosti 100, tak bychom například mohli 85 nových potomků vytvářet formou křížení, 10 náhodně vygenerovat a 5 nejlepších brát formou elitismu. Šance na mutaci pro každého z jedinců v populaci by pak mohla být například 2 %. Nicméně neexistuje žádná poučka o tom jak by tento poměr měl vypadat a se všemi parametry se dá hýbat dle vypozorovaných výsledků během evoluce a dále optimalizovat.

4.2 NEAT

NEAT (NeuroEvolution of Augmenting Topologies) je velmi populárním genetickým algoritmem. Algoritmus pochází z roku 2002, kdy byl poprvé popsán Kennethem O. Stanleym. Algoritmus je neustále vyvíjen a existuje mnoho různých variant. Smyslem algoritmu je nejen nalézt optimální váhy jednotlivých spojení neuronů v síti, ale zároveň se snažíme najít i optimální rozložení sítě (topologii). Toho dosáhneme náhodným přidáváním či odebíráním neuronů, vazeb a vrstev během procesu mutace jedince v populaci.

Algoritmus zavádí tzv. genomy, které si uchovávají informaci o tom jak zpětně zrekonstruovat neuronovou síť každého jedince. Dále zavádí pojem druhů (species), který slouží k rozdělení populace na základě genomické vzdálenosti. Genomická vzdálenost je vypočtená hodnota, která určuje jak podobní si jsou jedinci v populaci a zda se jedná o stejný druh. V rámci algoritmu se zavádí mezní parametr, který určuje tuto mezní hodnotu. Zmíněné údaje se následně použijí k mezidruhovému křížení a lze tedy křížit i jedince s různou topologií sítě. Pro více informací lze náhlednou přímo do originální práce, kde je vše detailně popsáno [20].

4.2.1 Varianty algoritmu

rtNEAT

Jedná se o rožšíření pro NEAT jehož cílem bylo algoritmus upravit tak, aby bylo možné jedince vyvíjet i v reálném čase. Pokud bychom chtěli NEAT použít například pro učení robotů, tak bychom mohli uvažovat o použití této nástavby a sledovat jejich zlepšování v reálném čase. V případě, že se jedná o čistě virtuální prostředí a máme možnost celý proces simulovat v jednotlivých iteracích, tak nám to žádnou výhodu nepřináší.

HyperNEAT

Tato nástavba vznikla za účelem zlepšení celého procesu neuroevoluce na velmi velkých strukturách dat. Bylo dokázáno, že na velkých datech má tento přístup vyšší efektivitu než samotný NEAT, ale na menších datech dosahují algoritmy velmi podobných výsledků a jsou srovnatelné [21]. Z toho důvodu nevidím důvod pro použití v mé práci, kde budu pracovat s relativně malými daty.

odNEAT

Speciálně upravená verze algoritmu, která je modelovaná specificky pro robotiku a učení robotů v reálné aplikaci. Zavádí ostrovní model, kde jsou spolu roboti schopni interagovat a předávat si informace.

4.2.2 Výběr algoritmu

Pro práci jsem zvolil NEAT, protože se mi zdá přiměřené tohoto zástupce neuroevoluce použít pro porovnání s obdobně pokročilým zástupcem zpětnovazebního učení, kde jsem zvolil metodu DDQN. Použiji pouze základní verzi algoritmu, protože rozšíření nejsou pro mou práci relevantní. Nástavby jsou vhodné spíše pro velmi specifické a výrazně složitější případy. V mém případě budu pracovat pouze s virtuálním prostředím a s menším objemem dat a jejich výhod bych tedy ani zdaleka nedokázal využít.

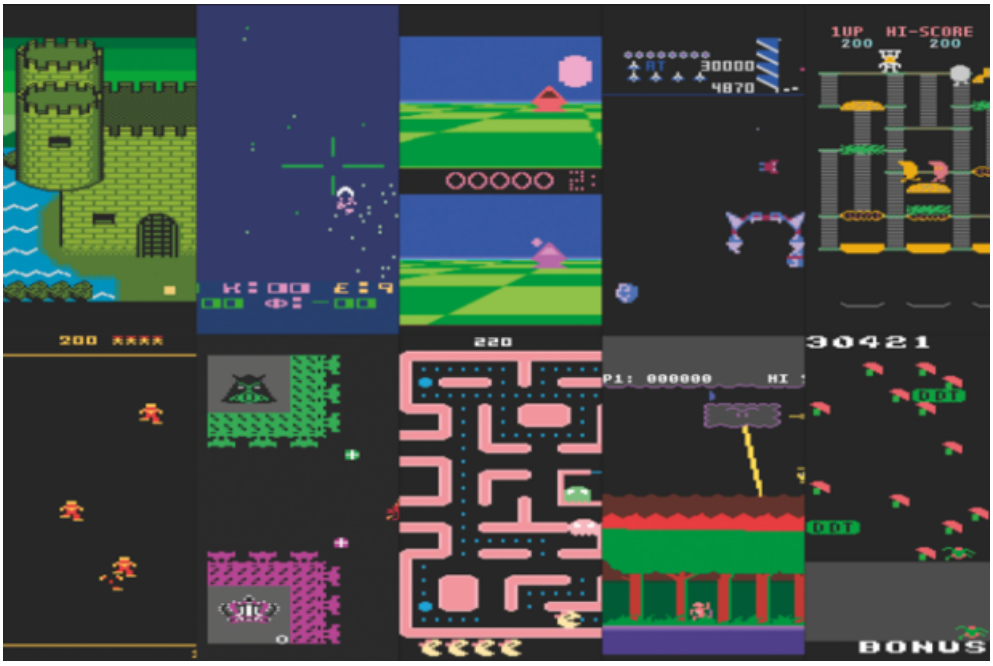
Z důvodu vyšší složitosti a cíle dosažení co nejlepších výsledků pro porovnání nebudu celý algoritmus implementovat zcela sám od základů. Místo toho využiji z větší části knihovnu neat-python, která je aktivně vyvíjená a má rozsáhlou a detailní dokumentaci. Tato knihovna již řeší problémy jako je mezidruhové křížení a budu se moci více soustředit na jiné problémy a ladění evolučních parametrů.

5 Virtuální prostředí gym

Gym je dlouhodobě vyvíjený open source projekt skupiny Open AI [22], který nabízí celou řadu virtuálních prostředí kontroly robotů, učení algoritmů, atari her, atd. Celý projekt je určen převážně pro vývoj algoritmů a metod zpětnovazebního učení používaných při vývoji umělé inteligence a jejich následné porovnávání. Není primárně určen pro metody neuroevoluce, ale pro mé účely je dostačující.

5.1 Atari hry

Atari 2600 je konzole, která se prodávala již v roce 1977. Knihovna gym hry implementuje tak, aby se co nejvíce podobaly originálním hrám z této konzole. Na obrázku 5.1 si lze prohlédnout ukázku 10 náhodně vybraných her, aby bylo možné si udělat rychlou představu o tom jak celé virtuální prostředí vypadá.



Obrázek 5.1: Ukázka virtuálního prostředí atari her [23]

V rámci práce nebylo reálné si virtuální prostředí videoher implemetovat zcela sám a proto jsem pro účely práce zvolil toto předem připravené a volně dostupné prostředí. Variantu od OpenAI jsem si vybral z toho důvodu, že je ve své podstatě ojedinělá a nabízí jednoduchou a intuitivní práci se simulací. Prostředí lze jednoduše procházet pomocí předem definovaného počtu akcí a při každém provedení akce dostaneme nové pozorování (data).

Gym nám poskytuje atari hry ve dvou odlišných verzích. Jednou z nich je RAM, kde nám dodává odpovídající data a simulaci toho co se dělo v paměti RAM v těchto herních konzolích během hraní hry. Zde pracujeme s daty o velikosti 128 bytů což byla právě velikost RAM těchto konzolí v roce 1977. Konkrétně nám prostředí vrací pole o délce 128, kde jednotlivé hodnoty nabývají hodnot 0-255. Druhou variantou, kterou nám atari gym nabízí jsou pozorování ve formě aktuálního snímku obrazovky hry. Jednotlivé pixely dostáváme ve formě RGB obrázků o velikosti 210x160x3 což opět odpovídá tomu jak byla konzole původně konstruována.

Právě varianta, která poskytuje obrázkové výstupy byla použita v originální práci popisující algoritmus DQN jak již zmiňuji v kapitole o DQN algoritmu. V práci autoři zjistili, že nejlepších výsledků dosáhli v případě, že jednotlivé snímky transformovali na velikost 84x84 ve stupních šedi a do sítě neposílali snímky po jednom, ale jako sekvenci 4 snímků v řadě v rámci jednoho vstupu rožšířeného o tuto dimenzi (84x84x4). V práci si vypůjčím jejich architekturu konvoluční sítě a natrénuji ji na vlastní implementaci DDQN algoritmu. Tímto získám relevantní výsledky, které mohu použít pro základní referenci toho jak by výsledky měly přibližně vypadat. Hlavním řešením bude porovnání RAM variant při použití zpětnovazebního algoritmu DDQN, kde vytvořím i vlastní jednoduchou síť a neuroevolučního algoritmu NEAT.

5.2 Další použité knihovny

Celá knihovna gym je implementována v jazyce python což je asi nejrozšířenější jazyk pro práci s neuronovými sítěmi. Celou práci koncipuji právě v tomto jazyce. Konstrukci algoritmu DDQN a neuronových sítí a jejich následné trénování provádím pomocí knihovny pytorch. Pro konstrukci algoritmu NEAT používám knihovnu neat-python. V nějakých případech používám pro práci s poli a zpracování dat knihovnu numpy. Výsledné grafy a vizualizaci učení a evoluce generuji pomocí knihoven matplotlib a visualize.

5.3 Předzpracování dat

Jak již je vysvětleno v předchozích kapitolách tak v tomto případě nemáme žádný konkrétní dataset pro učení neuronové sítě. Nicméně simulací hry získáváme data v reálném čase, která používáme pro trénování a dodáváme je do sítě, která na základě těchto dat činí vlastní rozhodnutí. Tato data nám poskytuje právě výše zmíněná knihovna gym a její prostředí atari her. Knihovna nám dodává 3 základní parametry. Stav, odměnu a informaci o konci hry. Nicméně je za potřebí data pro účely zpětnovazebního učení i neuroevoluce, alespoň částečně transformovat. To je důležité z toho důvodu, aby se nám s daty lépe pracovalo a abychom pracovali v obou porovnávaných případech se stejnými daty

5.3.1 Wrapper

Nad prostředím jsem si napsal wrapper, který mi kromě 3 výše zmíněných parametrů (stav, odměna, konec hry) dále vrací ještě informaci o aktuálním skóre ve hře pro statistické účely a přidává druhý parametr s informací o dalším konci hry. Tato informace je důležitá pro hry, kde existuje koncept více životů. V takovém případě ukončuji hru už ve chvíli, kdy dojde ke ztrátě prvního života. Tato úprava zásadně neovlivní trénování, ale dojde k opravdu výraznému urychlení trénování i evoluce u her, kde hráč disponuje více životy. Jedná se tedy spíše o optimalizaci pro urychlení evoluce a učení sítě.

Ořez odměny (reward clipping)

Další úpravou je něco čemu se říká reward clipping. Každé prostředí odměňuje hráče jiným způsobem. Při nějaké akci může být odměna 1 (sebrání krmiva ve hře Pacman) a nebo také 100 (sežrání nepřítele). Tyto odměny poskytované prostředím upravuji tak, aby odměna mohla nabývat pouze tří hodnot (1, 0, -1). Hlavním důvodem této úpravy je zamezení přeučování neuronové sítě, která by se snažila najít v krátkém časovém horizontu pouze jednu strategii, která by sice v krátkodobém horizontu mohla maximalizovat skóre, ale zároveň by mohlo dojít k přeučení a mohli bychom se dostat do lokálního optima. V takovém případě by se již síť nic nového nenaučila a nikdy bychom nedosáhli optimálního řešení. Tato úprava byla představena již v práci představující DQN algoritmus. Druhým a menším důvodem je to, že se chci soustředit na to, aby síť hrála hru co možná nejobecněji a nesnažila se najít exploity což by na druhou stranu mohlo být zcela jistě zajímavé téma pro jiný experiment.

6 Implementace DDQN

V této kapitole představím svou vlastní implementaci DDQN algoritmu, který jsem vybral na základě rešerše jako zástupce RL. Algoritmus se mi pro výsledné porovnání s neuroevolucí zdál nejvhodnější (důvody jsou popsané v kapitole o RL). Jak již je popsáno v této kapitole tak metoda vychází z Q learningu a DQN. Používáme neuronovou síť a nepoužíváme Q table, která slouží mimo jiné jako paměť předchozích zkušeností. Proto je nutné paměť sítě nějakým způsobem implementovat. Originální DQN práce zavádí termín Replay Memory.

6.1 Replay Memory

Jedná se o soubor předchozích zkušeností, kterými si agent prošel. Ve své práci ji implementuji jako pole objektů, které si uchovávají aktuální stav, provedenou akci, nadcházející stav, odměnu přechodu stavů v rámci provedení dané akce a informaci o tom jestli nastal konec hry. Jedná se o třídu, která implementuje cyklický buffer, který po naplnění své kapacity zahodí nejstarší záznam a přidá záznam nový.

```
def push(self, state, action, next_state, reward, done):
    if len(self.memory) < self.capacity:
        self.memory.append(None)
    self.memory[self.position] = Transition(state, action, next_state,
        reward, done)
    self.position = (self.position + 1) % self.capacity
```

V práci nastavuji tento buffer na velikost 100000 záznamů. Dá se tedy říci, že si uchovávám informaci o posledních 100000 stavech, kterými si agent během simulace prošel. Tato data následně slouží k trénování. K tomuto účelu mám ve třídě implementovanou druhou metodu, která náhodně vybere vzorek přechodů stavů o velikosti batche, se kterým pracuji během trénování.

```
def sample(self, batch_size):
    return random.sample(self.memory, batch_size)
```

6.2 DDQN

Z rešerše je patrné, že DDQN vychází z metody DQN a zavádí druhou neuronovou síť. Tyto sítě musí začínat ze stejného bodu a tedy mít stejné parametry.

```
policy_net = DQNNetwork(INPUT_CHANNELS, N_ACTIONS).to(device)
target_net = DQNNetwork(INPUT_CHANNELS, N_ACTIONS).to(device)
target_net.load_state_dict(policy_net.state_dict())
```

Na předchozím výseku kódu je vidět vytvoření dvou sítí a zkopírování parametrů jedné z nich do sítě druhé. Síť je vytvořená pomocí knihovny pytorch a je zasílána na gpu pro rychlejší trénování. Samotnou architekturu sítě v této kapitole řešit nebudu, protože nesouvisí se samotným algoritmem, který je vůči různým sítím relativně flexibilní a je možné ho použít víceméně s libovolnou sítí. Architektury tedy zmíním až v rámci provádění experimentů a porovnávání metod.

6.2.1 Průzkum prostředí (epsilon-greedy)

K průzkumu prostředí jsem si v rámci rešerše zvolil epsilon-greedy strategii. V kódu níže je vidět jak celý proces provádím. Konstanta `START_EPS` zpočátku začíná na hodnotě 1 a tudíž se v první iteraci vždy provede náhodná akce. Smyslem konstanty `END_EPS` je naopak zaručit, že k průzkumu prostředí bude docházet v určité frekvenci vždy ať už budeme v libovolně vysoké iteraci. Zde je nastaveno na 0.05 čímž dosáhneme toho, že minimálně v 5 % případů budeme vždy provádět náhodnou akci a agent bude mít tendenci stále objevovat nové dříve neprobádané stavy a neustále se učit. Hodnota `epsilon` klesá pozvolna a `END_EPS_LIMIT_STEP` zajišťuje, že po provedení miliónu iterací se už bude vždy brát pouze `END_EPS`. Hodnota `s` vyjadřuje počet provených iterací (kroků).

```
epsilon = np.maximum(END_EPS, START_EPS -
    (EPS_INT * s / END_EPS_LIMIT_STEP))

p = random.random()
if epsilon < p:
    actions = policy_net(image.unsqueeze(0).to(device).float() / 255.)
    action = torch.argmax(actions)
    action = action.detach().cpu().numpy()
else:
    action = env.action_space.sample()
```

6.2.2 Normalizace dat

V předchozím výseku kódu lze vidět, že dochází k tzv. normalizaci dat, kde data dělím číslem 255, protože se jedná o data celých čísel v intervalu 0-255 a v rámci normalizace bych je rád dostal do intervalu reálných čísel mezi 0 a 1. Bylo prokázáno, že neuronová síť dosahuje při normalizaci dat lepších a stabilnějších výsledků. Hlavním důvodem proč data normalizuji až nyní a ne v rámci předzpracování dat je to, že jsem v takovém případě omezen pamětí. V jedné z mých prvních implementací jsem tuto chybu udělal a můj `ReplayMemory` buffer neukládal integery, ale floaty. Nicméně jsem brzy zjistil, že i na grafikách s pamětí 32GB, které jsem používal v rámci google collab prostředí, došlo při použití obrázkových dat velmi rychle k vyčerpání kapacity a program skončil chybou. Proto normalizaci dat provádím až ve chvíli kdy jsou samotná data posílána do sítě jako parametr.

6.2.3 Proces trénování

Celý proces trénování v tuto chvíli funguje tak, že iteruji přes jednotlivé stavy prostředí a dělám dopředný průchod nad iniciální neuronovou sítí s náhodnými váhami. Pokud dojde k logickému konci hry tak pokračuji v iteraci a pouze vyresetuji prostředí do původního stavu. Získaná data ukládám do bufferu. Ve chvíli, kdy buffer dosáhne určitého počtu dat, v mém případě jsem minimum nastavil na hodnotu 5000, tak se přesouvám k procesu trénování, kde již pomocí knihovny pytorch a DDQN algoritmu aktualizuji samotné váhy v síti.

Váhy sítě neaktualizuji v každé iteraci, ale pouze každou n-tou iteraci (v mém případě každou čtvrtou iteraci). Tímto dávám agentovi prostor více obměňovat stará data v bufferu předtím než provedu zpětný průchod a aktualizaci vah, což je výpočetně nejnáročnější operace celého programu. Toto dělám z toho důvodu, abych urychlil proces trénování, protože během provádění experimentů jsem zjistil, že tímto přístupem začnu dosahovat výsledků mnohem rychleji. V rámci úspěšného učení dochází k zisku nových a kvalitnějších dat. Nicméně buffer stále obsahuje data z předchozích iterací, kde se prováděli akce téměř vždy náhodně. Vzhledem k velikosti bufferu a náhodnému výběru dat o velikosti batche 32 si tuto optimalizaci mohu dovolit. V případě, že bych měl malý buffer tak by tento přístup mohl způsobit problémy v procesu učení kvůli ztrátě dat. Tato optimalizace je pro mě důležitá z toho důvodu, že trénování i na velmi výkonných grafikách trvá hodiny a tímto jsem schopný výsledný čas trénování snížit téměř na čtvrtinu.

1. V programu nejprve načítám přechody z `ReplayMemory` a tato data upravuji tak, abych s nimi mohl lépe pracovat v rámci knihovny pytorch.

```
transitions = replay_memory.sample(BATCH_SIZE)
batch = Transition(*zip(*transitions))
```

```
state_batch = torch.stack(batch.state).to(device).float() / 255.
action_batch = np.stack(batch.action)
next_state_batch = torch.stack(batch.next_state)
                    .to(device).float() / 255.
reward_batch = torch.from_numpy(np.asarray(batch.reward))
                .to(device)
done_batch = torch.from_numpy(np.asarray(batch.done)).to(device)
```

2. Následně provedu dopředný průchod `policy_net` a získám očekávané Q hodnoty dle skutečně provedené akce. Tímto získám vektor o jedné dimenzi a velikosti batche. Tato hodnota je nutná pro výpočet hodnoty `loss` popsané níže a trénování sítě. Podobným způsobem zpracuji nadcházející stav. V tomto případě, ale získám maximální hodnoty přes batch v rámci všech možných akcí, které nám síť vrátila. Tyto hodnoty použiji jako indexy pro získání Q values (targets) ze sítě `target_net`.

```
Qs = policy_net(torch.cat([state_batch, next_state_batch]))
state_Q, next_state_Q = torch.split(Qs, BATCH_SIZE, dim=0)
state_Q = state_Q[range(BATCH_SIZE), action_batch]
next_state_Q_max = torch.max(next_state_Q, 1)[1].detach()
target_Q = target_net(next_state_batch)[range(BATCH_SIZE),
```

3. Dalším krokem je již provedení samotného výpočtu, který vychází z Q learningu (viz rešerše). Provádíme výpočet hodnoty `loss`, zpětný průchod sítě a díky pytorch objektu optimizer jsme schopni aktualizovat váhy v síti. Optimizer je definován pouze pro `policy_net`, která je právě zmíněnou pomocnou síti v rámci trénování `target_net`.

```
target = reward_batch + GAMMA * target_Q * done_batch

loss = huber_loss(state_Q.float(), target.float())
loss.backward()
optimizer.step()
```

4. V posledním kroku již pouze v určitém intervalu iterací kopíruji váhy ze sítě `policy_net` do `target_net`. Váhy neaktualizuji v každém kroku, ale v určitém intervalu což zajišťuje lepší stabilitu trénování (vychází z rešerše). Následně ukládám samotný model sítě pro budoucí použití a testování.

```
if s % TARGET_NET_UPDATE == 0:
    target_net.load_state_dict(policy_net.state_dict())

if s % STEPS_SAVE == 0:
    torch.save(target_net, ENV_NAME + str(s) + ".pt")
```

Výše je popsána nejdůležitější část algoritmu. Kromě toho si po celou dobu sbírám statistiky epizod (epizodou je myšlena jedna instance hry), ze kterých následně tvořím grafy. Celý program pro zpětnovazební učení a jeho metodu DDQN má okolo 400 řádků kódu.

6.2.4 Parametry trénování

V programu využívám optimizér Adam s krokem učení `learning_rate` nastaveným na hodnotu 10^{-4}. Pár experimentů jsem zkoušel provádět i s optimizérem SGD, ale výsledky byly srovnatelné a Adam se ukázal být o něco rychlejší variantou. Nepoužívám tedy klasický stochastic gradient descent (klesání podle gradientu). Pro loss používám huber loss. Zde bych mohl použít například také mean squared error (MSE) loss. Nicméně huber loss je pro mou aplikaci vhodnější, protože si umí lépe poradit s krajními hodnotami.

Síť trénuji na 2×10^6 kroků. Každý krok reprezentuje jeden aktuální stav. Váhy cílové sítě aktualizuji každých 5000 kroků. Velikost batche mám nastavenou na 32. Hodnotu `GAMMA`, což je parametr algoritmu Q learningu, který určuje jak moc velký význam mají hodnoty daleko v minulosti na kumulativní odměnu v budoucnosti, mám nastavenou na hodnotu 0.99. Čím blíže je číslo k 1, tím větší váhu minulé stavy dostávají v rámci kumulace celkové odměny. V mém případě dává smysl toto číslo nastavit právě co nejblíže jedné, aby akce provedené i v daleké v minulosti dostávaly nějakou váhu.

7 Aplikace algoritmu NEAT

V této kapitole představím svou aplikaci algoritmu NEAT, který je již více rozepsán v rešeršní části práce. Algoritmus jsem zvolil především proto, že se jedná o uznávaný a pokročilejší algoritmus z oblasti neuroevoluce a proto mi přijde vhodné ho porovnávat s obdobně pokročilým DDQN algoritmem z odvětví RL. Jak již jsem vysvětlil v rešerši tak NEAT nebudu implementovat od základů tak jako jsem to udělal u DDQN, ale z větší části využiji knihovny neat-python. Aplikace knihovny pak probíhá takovým způsobem, že zadefinujeme velikost vstupní a výstupní vrstvy a konfigurační soubor. Síť pak zpočátku nemá žádné vazby ani žádné další neurony a postupnými iteracemi je do těchto sítí v populaci přidává. Následně je populace sítě postavena před problém a vyberou se nejlepší jedinci v populaci, kteří vytvoří novou generaci.

Knihovna pro tyto účely obsahuje třídu `Population`, která přijímá paremetr což je třída `Config`. Tato třída obsahuje cestu ke konfiguračnímu souboru a iniciální vstupní parametry. Konfigurační soubor nastaví jak velké jednotlivé neuronové sítě budou (minimální počet skrytých vrstev), kolik budou mít vstupních a výstupních neuronů což odpovídá počtu vstupů a počtu možných proveditelných akcí. Dále lze nastavit další parametry jako je například velikost populace, které aktivační funkce se budou v generovaných sítích používat a spoustu dalších. Třída dále obsahuje metodu `run`, která přijímá dva parametry. Těmi jsou počet generací (počet iterací algoritmu NEAT) a funkce, která určuje pro jaký účel se snažíme naší populaci vyvinout a prochází jednotlivé genomy. Metoda `run` vrací na konci běhu nejvhodnějšího jedince pro řešení problému.

```
config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
    neat.DefaultSpeciesSet, neat.DefaultStagnation, CFG_FILE)
p = neat.Population(config)
best_genome = p.run(eval_genomes, NUM_GENERATIONS)
```

Funkce `eval_genomes`, kterou předávám metodě `run` musí iterovat přes všechny genomy. Jednotlivé genomy si v tomto případě lze představit jako detailní objekt všech jedinců v populaci obsahující všechny informace k tomu, aby bylo možné v každé iteraci znovu sestrojit neuronovou síť každého jedince. Tato data se také používají k

mezidruhovému křížení sítí různých topologií, které by jinak nebylo možné. Z toho důvodu si genomy drží i informaci o genomické vzdálenosti. Genomická vzdálenost je parametr, který určuje jak jsou si výsledné neuronové sítě podobné a zda se jedná o stejný druh (specie). To se děje z toho důvodu, aby byla co největší diverzifikace v populaci. NEAT má tendenci nové jedince vytvářet tak, aby si druhy byly co nejméně podobné. V případě, že dojde ke stagnaci druhu, žádný jedinec v populaci neudělá určitý počet generací pokrok při řešení problému, tak jsou všichni jedinci tohoto druhu vyřazeni z populace a nahrazeni novými.

Níže uvádím příklad funkce `eval_genomes`, kterou jsem implementoval pro svůj případ atari her. V rámci každého genomu je nutné zkonstruovat síť a pomocí ní provést odehrání jedné instance hry. Následně je nutné definovat fitness funkci a určit vhodnost genomu. Pro můj případ jsem fitness funkci definoval pouze jako dosaženou odměnu po odehrání instance hry. Bohužel nám v tomto směru virtuální prostředí neposkytuje žádné další dodatečné informace, protože jak už bylo zmíněno v kapitole o virtuálním prostředí, tak prostředí je primárně určené pro RL algortimy, které žádná další data nevyžadují. Pokud bychom tato data měli tak by bylo možné fitness funkci sestavit lépe a učit i nějaké cílové chování.

```
def eval_genomes(genomes, config):
    for genome_id, genome in genomes:
        state = env.reset()
        high_score = 0
        net = neat.nn.feed_forward.FeedForwardNetwork
                    .create(genome, config)

        while True:
            state = state.flatten() / 255.
            output = net.activate(state)
            action = output.index(max(output))
            state, reward, done, _, additional_done = env.step(action)
            high_score += reward
            if done or additional_done:
                break

        fitness = high_score
        genome.fitness = fitness
```

Dále má knihovna nástroje pro hlášení statistik během evoluce a vykreslení výsledků do grafů včetně toho jak výsledná síť vypadá a jak postupně vymíraly jednotlivé druhy.

```
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
```

```
p.add_reporter(stats)
p.add_reporter(neat.Checkpointer(10, filename_prefix=ENV_NAME))

visualize.draw_net(config, best_genome, True)
visualize.plot_stats(stats, ylog=False, view=True)
visualize.plot_species(stats, view=True)
```

Kromě toho je tu samozřejmě i možnost výslednou neuronovou síť uložit do souboru a aplikovat ji při testování či v praxi. Je také možné přerušit trénování a v určitém bodu navázat. Toho lze využít pokud bychom chtěli vyvíjet pro vysoký počet generací či velkou populaci. Už i pro mé nastavení trvala evoluce u některých her i 24 hodin. Knihovna python-neat narozdíl od knihovny pytorch použité u DDQN algoritmu neumožňuje výpočty na gpu a program tedy v některých případech běžel déle než DDQN algoritmus, přestože by proces evoluce měl být mnohem méně výpočetně náročný než samotné učení sítě.

7.1 Konfigurace algoritmu

Jak jsem zmínil v předchozí kapitole tak celý algoritmus se chová na základě konfigurace. Konfigurace je textový soubor, který obsahuje souhrn předem definovaných parametrů, které je možné nastavit či některé z nich úplně vynechat. Zde zmíním pár nejdůležitějších parametrů, které jsem upravoval a které se mi po nějaké době testování a ladění zdály optimální. U těchto parametrů je zcela jistě ještě velký prostor pro zlepšení, ale bohužel je hledání nejoptimálnějších parametrů hodně náročné z časových důvodů a vyžadovalo by stovky testů při kombinaci různých variant.

Populace

V rámci svého programu pracuji s velikostí populace `pop_size` o 100 jedincích. Větší populace má výhodu v tom, že obsáhneme větší prostor možností. Nicméně za to pak platíme výrazně delší dobou běhu algoritmu. Populace o 100 jedincích mi přišla jako dobrý kompromis.

Stagnace

Tímto parametrem určujeme po kolika iteracích, kdy žádný jedinec v druhu neudělal pokrok, dojde k úplnému vyhynutí druhu. Stagnaci lze úplně vypnout. Já zde parametr `max_stagnation` nastavuji na hodnotu 15, protože chci zaručit, že v případě stagnace druhu dojde k vygenerování nových jedinců a obsažení širšího prostoru

možností. Nevýhodou je, že pokud nevyužíváme elitismu tak může vyhynout i druh s aktuálně nejvhodnějším jedincem.

Aktivační funkce

Aktivační funkci `activation_default` jsem nastavil na ReLu z toho důvodu, že v jednom experimentu jsem chtěl porovnávat metody ve chvíli kdy jsou si sítě co nejpodobnější a v rámci DQQN používám pouze tuto aktivační funkci. Nicméně se mi osvědčilo ponechat šanci mutace `activation_mutate_rate` na 10 %, kde je tedy v každé iteraci šance, že dojde k mutaci v nějakém neuronu na jinou aktivační funkci a pokud síť dosáhne lepších výsledků tak se mutace zachová do dalších generací.

Vazby mezi neurony

Šance na mutaci a přidání nové vazby mezi neurony `conn_add_prob` v síti jsem nastavil na 95 %. Šance pro odebrání náhodného spojení neuronů `conn_delete_prob` je pak 10 %. Přidání neuronů je vysoké z toho důvodu, že jsem chtěl aby v síti bylo co nejvíce parametrů. Samotné neurony do sítě přidávám s šancí 75 % parametrem `node_add_prob`. Šance pro odebrání náhodného neuronu v síti je 2 % a je to řešeno v rámci nastavení parametru `node_delete_prob`.

Parametry neuronové sítě

Vstupní vrstva sítě `num_inputs` je definována pro 128 vstupů což vychází z virtuálního prostředí, kde používám (RAM). Počet výstupů `num_outputs` se liší pro každou atari hru a odpovídá celkovému počtu všech možných akcí v dané hře. Samotná neuronová síť je pak nastavená jako dopředná a ve výchozím stavu začíná bez jakýchkoliv vazeb mezi neurony. Síť by bylo možné nastavit i jako rekurentní a přidat tím možnost zpětných vazeb. Nicméně jak už jsem psal výše tak bych rád pro lepší porovnání zachoval co nejvyšší podobnost se sítí z DDQN, kde používám také pouze dopřednou síť.

Váhy

Posledním důležitým bodem je nastavení vah a jejich mutace. Váhy v síti mají 50 % šanci zmutovat (`weight_mutate_rate`) což prakticky znamená, že jsou pronásobeny konstantou 0.725 (`weight_mutate_power`). Je zde také 10 % šance, že váha bude plně nahrazena a dojde k vygenerování nového čísla v intervalu (-30, 30).

8 Výsledky a porovnání přístupů

V rámci této kapitoly bych chtěl ukázat proces učení obou přístupů a porovnat je. Budu se soustředit hlavně na data poskytnutá z prostředí atari her ve formě RAM paměti (128 vstupů). Dále udělám porovnání prostředí stejné hry s obrázkovým vstupem. Zde vezmu architekturu sítě z originální DQN práce a nátrenuji ji na svém DDQN algoritmu. Toto porovnání bude sloužit jako určitá validace, protože díky originální práci popisující algoritmus DQN vím, že již nějakých smysluplných výsledků s touto sítí a postupy někdo dosáhnul.

Původně jsem chtěl porovnávat DDQN s algoritmem NEAT na obrázkových datech nicméně jsem zjistil, že algoritmus NEAT pracuje pouze s běžnými dopřednými neuronovými sítěmi a neumí pracovat s konvolučními vrstvami. Nezdálo se mi tedy správné a nestranné porovnávat konvoluční síť s obyčejnou dopřednou sítí nad obrázkovými daty. Dále jsem také zjistil, že NEAT si vede výrazně lépe při menším počtu vstupů a na obrázkových vstupech nefungoval příliš dobře. Bylo by nutné spustit evoluci na obrovský počet generací, aby se zaregistrovaly všechny vstupy na což nemám výpočetní kapacitu.

Druhým měřítkem porovnání toho jak si tyto dva algoritmy vedou a zda se vůbec zlepšují v hraní dané hry bude náhodný agent. Tohoto agenta nechám odehrát sto instancí hry s čistě náhodnými akcemi. Následně provedu průměr odměn na jednu odehranou hru. Díky tomu budu jednoznačně schopný říct jak dobře oba algoritmy fungují a zda vůbec k nějakému učení či evoluci dochází.

8.1 Kritérium pro zhodnocení výsledků

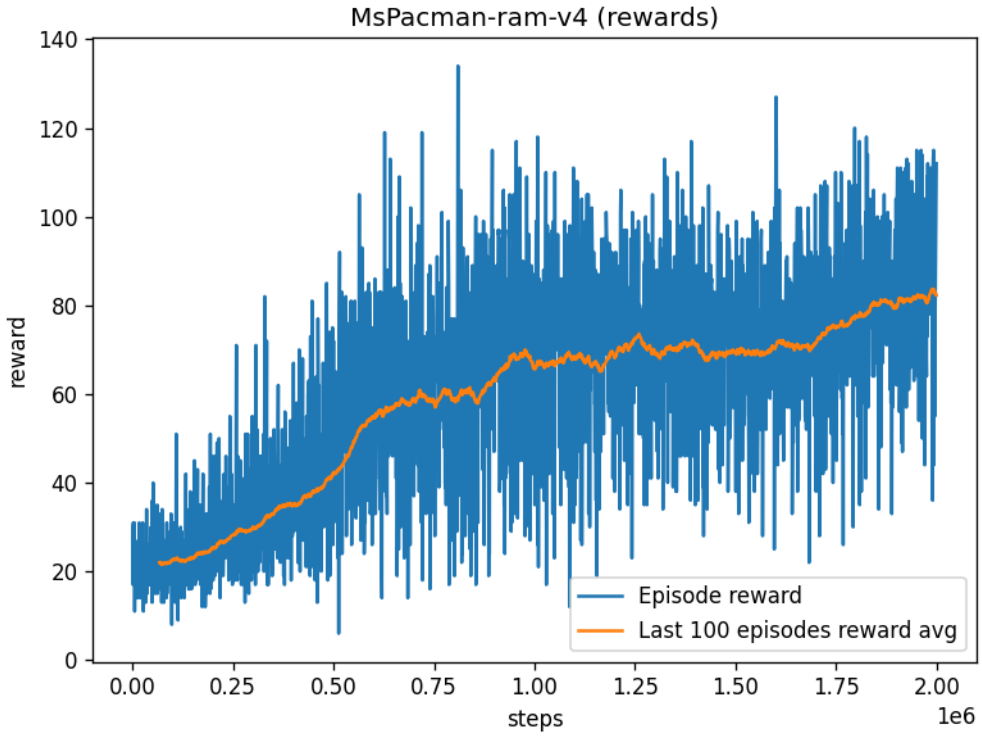
Výsledky budu vyhodnocovat a porovnávat na základě získané odměny (reward). Tento pojem je již blíže vysvětlen v předchozích kapitolách. Odměnu ořezávám do množiny čísel (-1, 0, 1) a to z toho důvodu, abych zabránil přeučení sítě. V případě, že je odměna v prostředí hry pro nějakou akci větší než 0 tak ji oříznu na hodnotu 1. To stejné platí pro záporné hodnoty, kde ořezávám na hodnotu -1. Pokud je odměna 0 tak zůstává neměnná. Tomuto procesu se také říká tzv. reward clipping a zavádí se z toho důvodu, abychom při zpětnovazebním učení neuvízli v lokálním optimu.

Síť by se totiž mohla přeučit a už se nikdy neposunout dále a najít optimálnější řešení. Pro neuroevoluci je stejný proces zaveden z toho důvodu, abychom pracovali se stejnými vstupními daty a měli jsme stejné a pokud možno nezaujaté výsledky, které je možné porovnávat. Nicméně v případě neuroevoluce přeučení nehrozí.

Pokud se tedy v následujícím textu, grafech či tabulkách zmiňuji o pojmu odměna či reward, tak je tím myšlena právě takto oříznutá odměna. Tato odměna je v obou algoritmech velice variabilní napříč iteracemi a epizodami jednotlivých odehraných her v průběhu běhu algoritmů. Proto budu jako výsledek pro NEAT brát průměr nejlépe si vedoucích jedinců z posledních 10 generací a pro algoritmus DDQN budu brát průmer z posledních 100 odehraných epizod.

8.2 Porovnání procesu učení a evoluce

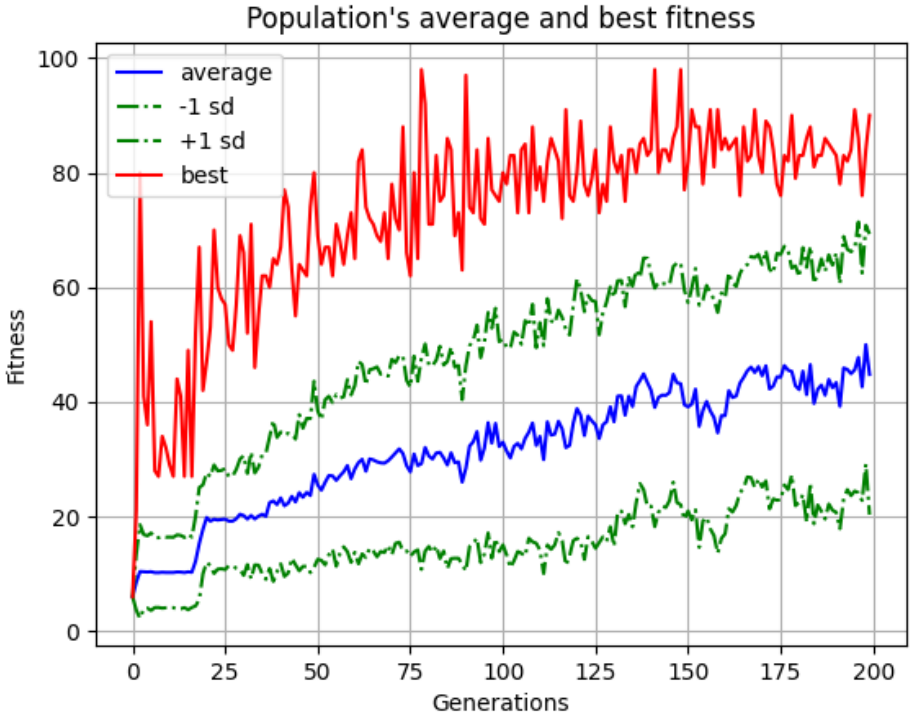
V této kapitole bych pouze chtěl pouze krátce poukázat na praktické rozdíly mezi algoritmy DDQN a NEAT na jednoduchých grafech získaných z obou procesů. Jako příklad použiji hru MsPacman, kde obě metody dosáhly srovnatelné odměny.



Obrázek 8.1: Ukázka učení a vývoje odměny při použití algoritmu DDQN

Na grafu 8.1 je vidět jakým způsobem probíhá zpětnovazební učení při použití algoritmu DDQN v průběhu 2×10^6 kroků (stavů). Ze začátku se síť neučí dokud se alespoň částečně nenaplní Replay Memory buffer a následně lze vidět, že dochází k

poměrně stabilnímu zlepšování díky trénování neuronové sítě. K největšímu zlepšení dochází v první čtvrtině a pak se tempo výrazně zpomalí. Eventuálně by došlo k úplné stagnaci trénování na nějaké hodnotě. Lze si všimnout, že nějaký prostor pro další trénování a zlepšení tu stále je. Z grafu je také vidět, že výsledky jsou v jednotlivých epizodách dost nestabilní a proto je pro nějaké rozumné porovnání nutné prokládat graf průměrnými výsledky z více epizod.



Obrázek 8.2: Ukázka evoluce a vývoje odměny při použití algoritmu NEAT

V případě neuroevoluce je na grafu 8.2 vidět jak odlišným způsobem celý NEAT funguje. V grafu evidujeme odměnu nejlepšího jedince v populaci a průměr odměn všech jedinců v populaci v každé generaci. Celá evoluce pak v tomto případě probíhá po dobu 200 generací. Průměr jedinců v populaci ukazuje pomalé a stabilní zlepšení, ale nejedná se o příliš relevantní údaj z toho důvodu, že NEAT vytváří neustále nové a náhodné jedince, kteří tento průměr logicky táhnou dolů. Nejdůležitějším údajem jsou odměny nejlepšího jedince v populaci. Tyto odměny následně průměruji za dobu posledních 10 generací, abych dosáhnul stabilnějšího a co nejméně variabilního výsledku pro porovnání.

Z grafu si lze také všimnout, že NEAT dosáhnul vyšší odměny mnohem dříve než DDQN (fitness v grafu odpovídá odměně). Už jedna z prvních generací byla schopna dosáhnout odměny blízko číslu 80. To je způsobeno z větší části náhodou. Některý z jedinců se náhodně vygeneroval se sítí, která byla schopna uhádnout optimální sekvenci akcí v dané instanci hry. V dalším běhu hry se jí už ale výsledek nepodařilo zreplikovat, protože prostředí atari her není plně deterministické. Z toho důvodu jsou

výsledky nejlepších jedinců variabilní a pohybují se nahoru a dolů i ve chvíli kdy samotná síť zůstane v přechodu z jedné generace do generace další neměnná. Abychom se nespoléhali právě na tuto náhodu jednoho běhu, tak je nutné iterovat přes více generací. Výsledná neuronová síť je pak flexibilní při řešení problému. V grafu je také vidět, že v posledních přibližně 50 generacích už nedošlo k žádnému zlepšení. Je tedy možné, že už by výsledek s aktuálním nastavením běhu algoritmu nebylo možné dále zlepšit. Na druhou stranu to nelze říci s jistotou, protože neuroevoluce má narozdíl od zpětnovazebního učení vlastnost dosahovat skokových průlomů.

8.3 Výsledky hlavního experimentu

V tabulce 8.1 lze vidět výsledky z několika vybraných atari her. Prvním sloupcem v tabulce je random agent což představuje průměr odměny ze 100 odehraných instancí hry při provádění náhodných akcí. V dalším sloupci lze vidět výsledky algoritmu DDQN aplikovaném na RAM vstupy. Jedná se o dopřednou a plně spojitou síť se 128 vstupy (RAM) a 3 skrytými vrstavami o velikostech 512, 256 a 128. Poslední vrstvou je vrstva výstupní, která se liší dle typu hry a její velikost odpovídá počtu akcí, které lze provést v dané hře. Ve třetím sloupci jsou výsledky algoritmu NEAT. V tomto případě jde opět o RAM vstupy a sítě jsou generovány náhodně.

Všechny předchozí sloupce vychází z RAM dat. K tomu přidávám pro porovnání ještě poslední sloupec. Zde přebírám architekturu neuronové sítě z původní práce DQN, kde autoři pracovali s obrázkovými daty. Tuto konvoluční síť natrénuji svým DDQN algoritmem na obrázkových vstupech (IMG) a používám to jako opěrný bod pro porovnání, protože z práce vím, že již s touto sítí někdo jiný dosáhl nějakých výsledků. Jedná se o síť se třemi konvolučními vrstvami a dvěma lineárními vrstvami. Nepřebírám žádná jejich data ani předem naučené parametry sítě. DDQN trénuji v obou případech po dobu dvou milionů kroků (stavů). U algoritmu NEAT evoluci spouštím u všech her po dobu 200 generací s populací o velikosti 100.

Tabulka 8.1: Porovnání odměny (reward) ve vybraných hrách

Prostředí	Random	DDQN (RAM)	NEAT (RAM)	DDQN (IMG)
Alien	15.84	77.7	60.1	73.8
Boxing	0.63	84.1	9.7	69.8
MsPacman	22.49	82.3	83.1	98.0
Pong	-20.12	-11.4	-19.6	17.5

8.3.1 Porovnání algoritmů DDQN a NEAT

Z tabulky 8.1 lze vyčíst, že oba algoritmy překonaly agenta, který prováděl čistě náhodné akce. Z tohoto pohledu lze tedy říct, že implementace obou algoritmů DDQN a NEAT byla úspěšná. Z tabulky to číselně vypadá tak, že zpětnovazební učení a jeho vybraný zástupce DDQN si až na jeden případ vedl mnohem lépe než NEAT jako zástupce neuroevoluce. Z toho tedy lze na první pohled usoudit, že zpětnovazební učení si u tohoto problému vede lépe. K tomu bych v této kapitole udělal jakési další porovnání na jednotlivých hrách včetně subjektivního pocitu při konečném testování a pozorování toho jak si výsledné modely vedly při hraní hry.

Alien, MsPacman

Tyto dvě hry jsou si velmi podobné a jsou to také hry, kde NEAT dosahoval podobných výsledků jako DDQN. Z toho usuzuji, že NEAT může být v nějakých případech lepší variantou při menším počtu akcí, protože tyto dvě hry mají menší počet výstupních akcí. Alien má pak o dvě akce více než MsPacman a již si lze všimnout výrazně většího rozdílu v odměně. Při pozorování výsledných modelů během hraní hry si lze všimnout, že NEAT se spíše učí sekvenci akcí a neučí se přímo reagovat na prostředí. U DDQN jsou vidět například i náznaky vyhýbání se nepřátelům. To je pravděpodobně dané tím, že zde penalizujeme agenta za prohrání hry. Pro NEAT bychom museli lépe formulovat fitness funkci. K tomu nám virtuální prostředí bohužel neposkytuje vhodná data.

Boxing

Boxing je příkladem hry s největším počtem možných akcí z her co jsem vybral pro tento experiment. Zároveň si zde DDQN vede mnohem lépe než NEAT. Z toho tedy opět usuzuji, že větší počet akcí bude hlavním důvodem rozdílu odměn. Větší počet akcí totiž může způsobit, že výsledná síť vygenerovaná algoritmem NEAT, která začíná bez jakýchkoli vazeb, vynechá vazbu na specifickou akci a vůbec se jí nenaučí řešit. Pak bychom tedy museli evoluci provádět na větším počtu generací, ale ani zde bychom neměli při velkém počtu akcí jistotu, že se neuronová síť naučí správně pracovat se všemi akcemi.

Pong

Z této hry mám smíšené pocity, protože nefunguje příliš dobře ani jedna RAM varianta. Síť převzaná z originální DQN práce po natrénování na obrázkových datech funguje dobře a algoritmus DDQN byl téměř schopný dosáhnout maximální možné

odměny ve hře (21). Při pozorování si myslím, že hlavní problém je v tom, že je zde malý prostor pro zlepšení v rámci provádění počátečních náhodných akcí a průzkumu prostředí. Je zde malá šance, že se při náhodných akcích podaří získat nějakou odměnu a celý proces tedy může trvat mnohem déle. Síť s obrázkovým vstupem toto svým způsobem řeší tím, že nepracuje pouze s jedním obrázkem, ale se sekvencí 4 obrázků v řadě za sebou v rámci jednoho vstupu.

8.4 Vedlejší experiment

Ve vedlejším experimentu se pokusím porovnat tyto dvě metody na stejném počtu parametrů. Vezmu neuronovou síť vygenerovanou algoritmem NEAT a pokusím se podobně velkou síť natrénovat algoritmem DDQN. Vzhledem k tomu, že se nejedná o plně spojitou síť a síť má mnoho nepravidelných vrstev a vazeb, tak ji nelze jednoduše převzít a přetrénovat. Proto se pro zjednodušení procesu pouze podívám na to kolik má výsledná síť parametrů (vazeb) a zkusím DDQN natrénovat na jiné neuronové síti, která bude mít obdobný počet těchto parametrů. Pro 200 generací se mi pro NEAT vygeneruje síť, která má přibližně 140 vazeb. Tomu by tedy v počtu parametrů přibližně odpovídala plně spojitá síť se 128 vstupy, jedné skryté vrstvě s jedním neuronem a výstupní vrstvě o velikosti počtu akcí v prostředí. Výsledky lze vidět v tabulce 8.2.

Tabulka 8.2: Porovnání odměny při podobném počtu parametrů

Prostředí	Random	DDQN (RAM)	NEAT (RAM)
Alien	15.84	23.0	60.1
Boxing	0.63	-11.7	9.7
MsPacman	22.49	48.0	83.1
Pong	-20.12	-20.7	-19.6

Z výsledků si lze všimnou, že v tomto případě NEAT zcela vyhrává a v některých případech jsou výsledky DDQN algoritmu horší než výsledky random agenta k čemuž pravděpodobně dochází z toho důvodu, že se DDQN agent přizpůsobuje hluku v prostředí a vyhodnotí, že se nemůže nic jiného naučit. Důvodem proč si zde NEAT vede lépe je to, že se kromě hledání vah zaměřuje také na strukturu celé neuronové sítě a hledá i optimální rozložení vrstev, vazeb a jednotlivých neuronů, které nemusí být pravidelné. DDQN je oproti tomu závislé na větší dopředné a plně spojité síti. Na druhou stranu lze v předchozí kapitole vidět, že s větší sítí, na kterou je algoritmus dimenzován si již DDQN vede mnohem lépe a ve většině případů NEAT předčí s velkou rezervou. Z tohoto experimentu si lze odnést závěr, že NEAT by měl být schopný lépe optimalizovat problém a jeho řešení na mnohem menší neuronovou síť. To by mohlo být velkou výhodou v případě, že bychom síť chtěli aplikovat někde,

kde bychom toto vyžadovali. Pro představu se bavíme například o stonásobně větší datové náročnosti pro uložení výsledného modelu sítě do úložiště. To by mohlo udělat výrazný rozdíl například při použití v embedded systému, který je ve svém datovém úložišti z nějakého důvodu omezen. Také by měl být teoreticky i výrazně rychlejší samotný dopředný průchod sítě.

8.5 Finální porovnání metod a aplikace v praxi

Z rešerše i z mých vlastních experimentů vyplývá, že obě metody mají nějaké výhody a nevýhody. Největší výhodou zpětnovazebního učení je to, že se jedná o mnohem rozšířenější odvětví. Obsahuje spoustu metod, které jsou vhodné pro použití v reálných aplikacích jako jsou robotika či autonomní řízení, protože nevyžadují mnoho zařízení pro natrénování a je zde mnohem menší šance, že dojde k poškození zařízení nebo okolního prostředí. K trénování nám tedy stačí jeden robot či automobil. Naproti tomu u neuroevoluce bychom potřebovali robotů a automobilů stovky a došlo by pravděpodobně i k jejich poškození.

Neuroevoluce je tedy spíše vhodná do čistě virtuálního prostředí, kde nehrozí žádné škody a máme možnost vše nasimulovat. To nemusí být nutně nevýhoda s tím jakým tempem se toto odvětví rozvíjí a neustále se přiližuje blíže k realitě. Naopak to přináší spoustu výhod a flexibility. Zde se nabízí jedna velké výhoda neuroevoluce a tou je mnohem nižší výpočetní náročnost. Z mých experimentů to nebylo úplně patrné, protože jsem optimalizací DDQN strávil mnohem více času než optimalizací algoritmu NEAT, kde jsem například mohl využít více vláken procesoru a jednotlivé simulace spouštět paralelně.

Číselně vyšlo, že i ve virtuálním prostředí zpětnovazební učení vítězí. Dalo by se zauvažovat nad tím jak moc velký rozdíl může udělat lepší implementace algoritmu NEAT, se kterou nejsem v rámci knihovny python-neat úplně spokojený. Chybí například možnost přidat více než jednu vazbu v jedné generaci. Myslím si, že by se dalo dosáhnout lepších výsledků. Nicméně druhý experiment ukázal i tak jednu výhodu neuroevoluce, kterou je lepší optimalizace počtu parametrů a velikost sítě v poměru k očekávanému výsledku což by jak zmiňuji v předchozí kapitole mohlo být ve velmi specifických případech výhodou při použití výsledného modelu.

Pokud bych měl v tuto chvíli vybrat jednoho vítěze tak bych musel stále vybrat zpětnovazební učení díky mnohem lepší a vhodnější aplikaci v reálném prostředí. Díky většímu rozšíření také existuje mnohem více prací, které se problematikou zabývají a lze dohledat více zdrojů pro práci ve většině programovacích jazycích. Toto by se mohlo časem změnit. Je důležité zmínit, že se nutně jedná o otázku jedno či druhé. Myslím si, že se oba přístupy mohou do budoucna doplňovat a vždy se najde specifický příklad, kdy bude lepší jeden z nich. Možná bude do budoucna možná i jejich přímá kombinace při řešení jednoho problému.

9 Závěr

V bakalářské práci se mi úspěšně podařilo implementovat zástupce zpětnovazebního učení i neuroevoluce. Z obou přístupů jsem na základě detailní rešerše vybral zástupce a ty jsem vůči sobě porovnal na vybraných hrách ve virtuálním prostředí atari her. Pro zpětnovazební učení jsem nakonec zvolil algoritmus DDQN, kde se neuronová síť učí běžným způsobem. U neuroevoluce jsem zvolil algoritmus NEAT, kde dochází ke generování topologie samotné neuronové sítě. Implementaci považuji za úspěšnou z toho důvodu, že oba algoritmy v rámci hlavního experimentu ve všech hrách výrazně předčily agenta, který ve hrách vykonával čistě náhodné akce. Pro další rozšíření práce a širší perspektivu by bylo určitě zajímavé nasbírat i data, kde by hry byly odehrané lidmi a porovnat výsledky i s těmito daty.

Se svou implementací algoritmu DDQN jsem velmi spokojený. S čím nejsem úplně spokojený a co si myslím, že by se dalo výrazně vylepšit je algoritmus NEAT, kde jsem využil knihovnu, která z velké části již tento algoritmus implementuje a měl jsem tedy možnost dělat jen menší úpravy. To mě výrazně omezilo v mých možnostech. Knihovnu jsem využil z toho důvodu, že je tento algoritmus výrazně složitější na správnou implementaci a pravděpodobně by ani nebylo možné vše v rozsahu bakalářské práce imlementovat tak jak potřebuji a s vysokou pravděpodobností bych dosáhnul ještě mnohem horších výsledků. Jedná se o něco na co bych se rád zpětně podíval v budoucí práci, protože na tuto problematiku bych rád navázal v diplomové práci a možná i v nějakých dalších pracích a myslím si, že tato bakalářská práce je k tomu dobře zpracovaným opěrným bodem.

Výsledkem mých experimentů a zjištění je, že zpětnovazební učení v tuto chvíli vítězí především díky vhodnějšímu použití v reálných aplikacích, kde je mnohem menší hrozba poškození drahých přístrojů či okolí. V rámci mnou provedených experimentů na atari hrách také dosáhlo výrazně lepší odměny. Nicméně si myslím, že neuroevoluce se bude dále vyvíjet a je zde určitě potenciál v budoucnu zpětnovazební učení překonat z toho důvodu, že neuroevoluce není závislá na architektuře sítě. Dále si myslím, že se nemusí nutně jednat o hledání toho co je lepší. Obě metody se mohou vzájemně doplňovat a možná bude v budoucnu možné je použít v kombinaci při řešení jednoho problému. V navazující práci bych se chtěl více zaměřit právě na problematiku neuroevoluce.

Použitá literatura

1. GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. MIT Press, 2016. ISBN 9780262035613. Dostupné také z: https://books.google.co.in/books?id=Np9SDQAAQBAJ.
2. WILLEMS, Karlijn. *Keras Tutorial: Deep Learning in Python*. 2019-12. Dostupné také z: https://www.datacamp.com/community/tutorials/deep-learning-python.
3. SZANDAŁA, Tomasz. Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks. In: *Bio-inspired Neurocomputing*. Springer, 2021, s. 203–224.
4. JADON, Shruti. *Different Activation Functions and their Graphs*. Medium, 2018-03. Dostupné také z: https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092.
5. SHTEINGART, Hanan; LOEWENSTEIN, Yonatan. Reinforcement learning and human behavior. *Current Opinion in Neurobiology*. 2014, roč. 25, s. 93–98.
6. DEMUSH, Rostyslav. Reinforcement Learning Examples. *Reinforcement Learning Applications: A Brief Guide on How to Get Business Value from RL*. 2018. Dostupné také z: https://perfectial.com/blog/reinforcement-learning-applications.
7. POLYDOROS, Athanasios S; NALPANTIDIS, Lazaros. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*. 2017, roč. 86, č. 2, s. 153–173.
8. SALLOUM, Ziad. Policy Based Reinforcement Learning, the Easy Way. *Medium*. 2020. Dostupné také z: https://towardsdatascience.com/policy-based-reinforcement-learning-the-easy-way-8de9a3356083.
9. OR, Barak. *Value-based Methods in Deep Reinforcement Learning*. Towards Data Science, 2021-01. Dostupné také z: https://towardsdatascience.com/value-based-methods-in-deep-reinforcement-learning-d40ca1086e1.

10. JULIANI, Arthur. *Simple Reinforcement Learning with Tensorflow Part 7: Action-Selection Strategies for Exploration*. Emergent // Future, 2017. Dostupné také z: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf.

11. SAGAR, Ram. *On-Policy VS Off-Policy Reinforcement Learning: The Differences*. 2020. Dostupné také z: https://analyticsindiamag.com/reinforcement-learning-policy/.

12. RAVICHANDIRAN, Sudharsan. *Hands-On Reinforcement Learning with Python*. Packt Publishing, 2020. Dostupné také z: https://www.oreilly.com/library/view/hands-on-reinforcement-learning/9781788836524/20659243-cadb-46f0-b5c3-3acadd590d67.xhtml.

13. MITCHELL, Tom M et al. Machine learning. 1997.

14. MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; GRAVES, Alex; ANTONOGLOU, Ioannis; WIERSTRA, Daan; RIEDMILLER, Martin. Playing Atari With Deep Reinforcement Learning. In: *NIPS Deep Learning Workshop*. 2013.

15. HASSELT, Hado van; GUEZ, Arthur; SILVER, David. *Deep Reinforcement Learning with Double Q-learning*. 2015. Dostupné z arXiv: 1509.06461 [cs.LG].

16. STANLEY, Kenneth O. *Neuroevolution: A different kind of deep learning*. 2017. Dostupné také z: https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/.

17. BALL, Ananya. 2019. Dostupné také z: https://medium.com/intel-student-ambassadors/demystifying-genetic-algorithms-to-enhance-neural-networks-cde902384b6e.

18. *Crossover in Genetic Algorithm*. 2019. Dostupné také z: https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/.

19. DURÁN, Francisco José; LLORENS, Faraón; PUJOL, Mar; ALDEGUER, Ramón. Driving-Bots with a Neuroevolved Brain: Screaming Racers. *Inteligencia artificial: Revista Iberoamericana de Inteligencia Artificial, ISSN 1137-3601, Nº. 28, 2005 (Ejemplar dedicado a: Nuevas tendencias en Sistemas Multiagente y Soft Computing), pags. 9-16*. 2005, roč. 9. Dostupné z DOI: 10.4114/ia.v9i28.859.

20. STANLEY, Kenneth O.; MIIKKULAINEN, Risto. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*. 2002, roč. 10, č. 2, s. 99–127. Dostupné také z: http://nn.cs.utexas.edu/?stanley:ec02.

21. STANLEY, K. O.; D'AMBROSIO, D. B.; GAUCI, J. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life*. 2009, roč. 15, č. 2, s. 185–212. Dostupné z DOI: 10.1162/artl.2009.15.2.15202.

22. OPENAI. *A toolkit for developing and comparing reinforcement learning algorithms*. [N.d.]. Dostupné také z: https://gym.openai.com/envs/#atari.

23. LYU, Xueguang. *Install OpenAI Gym with Atari on macOS*. Xueguang Lyu, 2021. Dostupné také z: https://xue-guang.com/post/gym-macos/.

Příloha A

- Grafy, natrénované modely neuronových sítí a kompletní zdrojové kódy
  - Zkomprimované a nahrané jako příloha do systému stag
  - Dostupné také z: https://github.com/martin-tvrdik/BP2021