Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky

Vyhodnocení agentů hlubokého Q-učení pomocí testu algoritmického IQ

DIPLOMOVÁ PRÁCE

Studijní program: Znalostní a webové technologie

Autor: Bc. Michal Dvořák
Vedoucí práce: Ing. Ondřej Vadinský, Ph.D.

Praha, červen 2024

Poděkování

Především děkuji svému vedoucímu práce panu Ing. Ondřeji Vadinskému, Ph.D za možnost pracovat na tomto projektu. Bez jeho znalostí a nesčetných rad a výbornému vedení by tato práce nejspíše nevznikla. Velké díky mu patří také za jeho trpělivost. Dále bych chtěl poděkovat kolegům Ing. Petru Zemanovi a Bc. Janu Štiplovi za jejich příspěvky k AIQ testu, jejich rady a možnost konzultovat kód. Děkuji také rodině a přátelům za podporu a trpělivost.

This work was funded by the Internal Grant Agency of Prague University of Economics and Business (F4/41/2023). Computational resources were kindly provided by the project "e-Infrastruktura CZ" (e-INFRA CZ LM2018140) supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Abstrakt

Hluboké Q-učení propojuje dva směry oboru umělé inteligence, a to neuronové dopředné sítě a posilované učení. Posilované učení představuje metodu, kdy se agent, v podobě například umělé inteligence, učí postupným řešením daného problému. Problém představuje prostředí, které s agentem komunikuje a předává mu informaci v jakém se nachází stavu. Agent na tuto informaci reaguje provedením akce, za kterou mu prostředí může dá odměnu. Agentovým cílem je naučit se takové chování, aby nasbíral co největší počet odměn. Hluboké Q-učení je nadstavbou klasické metody řešení problémů posilovaného učení zvané Q-učení. Tato metoda používá k učení svého chování Q-funkci. Hluboké Q-učení aproximuje tuto Q-funkci neuronovou sítí, díky čemuž dokáže pracovat s prostředími s velkými stavovými prostory.

V rámci této práce byl implementován agent hlubokého Q-učení do testu algoritmického IQ (AIQ), který vychází právě z posilovaného učení, a využívá koncept univerzální inteligence. Velmi zjednodušenou myšlenkou AIQ testu je otestovat agenta na co nejvíce prostředích, změřit jeho nasbíranou odměnu a přes složitost prostředí sečíst výsledky do finálního AIQ skóre.

V důsledku výzkumu hlubokého Q-učení byly finálně implementovaní dva agenti. První agent využívá původní znění z práce o hlubokém Q-učení a obsahuje mechanismus sbírající agentovi interakce v testu zvaný replay memory, a dále obsahuje jednu neuronovou síť, která se učí agentovu politiku a tedy chování. Druhý agent vychází z rozšiřující práce o hlubokém Q-učení a do této architektury přidává druhou neuronovou síť, která pomáhá tvořit učící data, čímž stabilizuje proces učení hlavní neuronové sítě pro agentovu politiku.

Test algoritmické inteligence a samotná implementace agenta využívá programovací jazyk Python a populární knihovny jako NumPy či PyTorch. Z důvodu velkého parametrového prostoru byl do testu přidán algoritmus založený na evolučních algoritmech. Pomocí nich byly nalezeny konfigurace, která pro prvního agenta nezískala lepší výsledek než konfigurace z původních článků, ale u konfigurace pro agenta s dvěma neuronovými sítěmi byla nalezena konfigurace, která je se statistickou významností lepší než konfigurace z původního článku. Pro toto ověření byl použit Studentův t-test.

Výsledky agentů hlubokého Q-učení byly porovnány s výsledky agenta prostého Q-učení, protože bylo zajímavé zjistit, jak velký vliv aproximace Q-funkce přes neuronovou síť bude mít. Podle Studentova t-testu ale nebyly výsledky na testovaném nastavení AIQ testu signifikantně rozdílné.

Otestování agentů ani nepotvrdilo statistický rozdíl mezi výsledky obou architektur. Při porovnání všech nejlepších konfigurací agentů testu, oba agenti hlubokého Q-učení získali přední příčky ve výsledném skóre.

Tyto výsledky byly získány pomocí testu algoritmického IQ, který byl nastaven pro všechny agenty stejně. Rozdíly mezi agenty byly testovány pomocí Studentova t-testu.

Klíčová slova

algoritmické IQ, evoluční algoritmy, hluboké Q-učení, měření inteligence, neuronové sítě, posilované učení

Abstract

Deep Q-learning connects two branches of artificial intelligence: feed-forward neural networks and reinforcement learning. Reinforcement learning is a method in which an agent that can be artificial intelligence learns by solving a given problem. The problem is an environment that communicates with the agent and gives him information about the state of the environment. The agent reacts to this information by taking action, for which the environment can reward him. Agent's goal is to learn a behavior that allows him to collect the biggest sum of cumulated rewards.

Deep Q-learning is a superstructure of a classical method for solving reinforcement learning problems called Q-learning. This method uses the Q-function to learn its behavior. Deep Q-learning approximates the Q-function by a neural network. Thanks to that, the agent can work with environments that have large state spaces.

In this thesis, the agent was implemented into an algorithmic IQ (AIQ) test, which builds upon the concept of reinforcement learning and uses the concept of Universal intelligence. The very simplified thought of the AIQ test is to test an agent in the most environments possible, measure its accumulated reward, and, by taking into account the difficulty of that environment, count the final AIQ score.

Two agents were implemented as a result of later research on Deep Q-learning. The first agent comes out of the first research paper about Deep Q-learning and consists of a mechanism for gathering the agent's experiences from interactions with the environment within the test, called Replay memory. He also uses a single neural network that learns the agent's behavior. The second agent comes from a follow-up research paper about DQL, in which the second neural network is added to the agent's architecture. This second neural network helps to create learning data for the main neural network, which helps to stabilize the learning of the main neural network that still handles its behavior.

AIQ test and implemented agents use Python as a programming language with popular libraries such as NumPy and PyTorch. The test was extended by a search algorithm based on evolutionary algorithms because of the large hyperparameter spaces of both agents, and evolutionary algorithms help to search this space heuristically and more efficiently.

With these evolutionary algorithms, agent configurations were found, and the configuration of the first agent didn't get better results than the configuration from the research paper, but for the second agent, a configuration was found that was statistically significantly better than the configuration from the research paper.

The results of the DQL agents were compared with the results of the basic Q-learning agent because it was interesting to test the significance of the approximation of Q-function by the neural network. The results of this experiment don't prove that DQL is any better than Q-learning for the current AIQ setting.

There weren't any significant differences between the two DQL architectures themselves either. Agents got the best ranks by final scores when tested against the best configurations of all other agents in the AIQ test.

All results were gathered using the algorithmic IQ test, which was set up for each agent in the same way. The results of the agents were tested using the Student's t-test.

Keywords

algorithmic IQ, Deep Q-learning, evolutionary algorithms, measuring intelligence, neural networks, reinforcement learning

Obsah

Seznam obrázků

Seznam algoritmů a kódů

Seznam použitých zkratek

ADAM Adaptive Moment Estimation
AIXI Artificial Intelligence with eXponential Intelligence
AIQ Algorithmic Intelligence Quotient
BF Brainf*ck
DQ_L|DQL Deep Q-learning
DDQ_L Deep Q-learning s target network
DQN Deep Q-learning network
PPO Proximal Policy Gradient
RMSProp Root Mean Square Propagation
TPU Tensor Processing Unit
VPG Vanilla Policy Gradient

Úvod

Tématem této diplomové práce je vyhodnocení agentů hlubokého Q-učení pomocí testu algoritmického IQ. Toto téma mi přijde zajímavé, protože se dotýká dvou fascinujících témat a to umělé inteligence, která je teď v plném rozkvětu, a zjišťováním, zda je umělá inteligence opravdu inteligentní a popřípadě jak moc. Tato diplomová práce byla řešena na Katedře informačního inženýrství v rámci výzkumného projektu IGA.

Stručný přehled současného stavu poznání

Otázkou zda mohou být umělé systémy inteligentní, a jak to ověřit, se zabýval již Alan Turing (1950). Od té doby se toto odvětví značně posunulo. Na inteligenci umělých systémů se již nenahlíží pohledem, jak moc je umělá inteligence podobná chováním člověku, ale spíše jak dobře zvládne řešit problémy bez pomoci z vnějšku.

Zatím nejzajímavější metoda takového měření inteligence na základě schopnosti řešit problémy je univerzální inteligence (Legg a Hutter, 2007). Jejím principem je otestovat jedince na všech možných problémech a s ohledem na složitosti testovaných problému vyvodit finální inteligenci. Tento princip ale není v praktickém ohledu proveditelný. Z toho důvodu vznikla z konceptu univerzální inteligence její aproximace v podobě algoritmického IQ (Legg a Veness, 2011).

Test algoritmického IQ převádí problémy s nevyčíslitelností univerzální inteligence na jejich aproximaci, ovšem za cenu nejistoty výsledků. Problémy, které testovaní agenti řeší, představují programy programovacího jazyka referenčního stroje, a jejichž složitost je vyjádřena délkou těchto programů.

Test algoritmické inteligence vychází z konceptu posilovaného učení. Problém zde představuje prostředí, které je v nějakém stavu. Prostředí s agentem neustále komunikuje, a agent tak získává pozorování těchto stavů. Na základě těchto pozorování provádí akce, které mění stav prostředí. S každou změnou prostředí získá agent i odměnu. Agentovým úkolem je naučit se takové chování, které mu přinese ve výsledku největší součet odměn v daném prostředí za celý běh.

Jednou z metod řešení úloh posilovaného učení je Q-učení (Watkins a Dayan, 1992). Ta pro výběr akcí nepracuje pouze s odměnami, ale používá i takzvanou kvalitativní funkci, která určuje jak dobré je provedení akce v daném stavu prostředí, pokud se bude agent chovat nadále podle svého aktuálně naučeného chování. Snaží se tak zohledňovat i agentovu budoucnost. Akce jsou vybírány tak, aby ve finále získal agent největší počet odměn.

Q-učení selhává pokud je počet možných stavů prostředí a počet možných akcí příliš velký. Hluboké Q-učení (Mnih, Kavukcuoglu, Silver, Graves et al., 2013) modeluje Q-funkci neuronovou sítí. Tím odpadá problém s počty stavů a akcí, protože neuronové sítě dokáží zpracovávat

takřka libovolně veliká data. Autoři byli díky tomu schopní nechat agenta hrát videohry herní konzole Atari 2600 (Bellemare, Naddaf et al., 2013), kde dokonce agent překonal lidské hráče ve většině her.

Při učení agenta hlubokého Q-učení se ale často stávalo, že jeho učení divergovalo, a nebyl se tak schopný správně učit. Z tohoto důvodu vzniklo rozšíření agenta, které přidalo do architektury druhou neuronovou síť (Mnih, Kavukcuoglu, Silver, Rusu et al., 2015). Tato síť má za úkol vytvářet učící data ze získaných zkušeností pro hlavní neuronovou síť, která zodpovídá za agentovo chování. Přidání druhé neuronové sítě mělo za následek v průměru o 83 % lepší výsledky než měla původní architektura z důvodu stabilizování učení. Výsledky tak překonaly v té době všechny známé umělé systémy hrající videohry na Atari 2600.

Cíle práce

Za cíle své diplomové práce si kladu:

1. Implementovat agenty hlubokého Q-učení:
   (a) Porozumět architekturám hlubokého Q-učení.
   (b) Implementovat agenta s původní architekturou využívající jednu neuronovou síť.
   (c) Implementovat navazující verzi agenta s dvěma neuronovými sítěmi.
2. Nalézt vhodné konfigurace pro použití v testu algoritmického IQ:
   (a) Otestovat konfigurace agentů ze článku Mnih, Kavukcuoglu, Silver, Rusu et al. (2015).
   (b) Prozkoumat konfigurační prostor agentů a pokusit se najít tu nejlepší konfiguraci.
3. Změřit výslednou algoritmickou inteligenci:
   (a) Porovnat výsledky obou architektur hlubokého Q-učení mezi sebou.
   (b) Porovnat výsledky hlubokého Q-učení a prostého Q-učení.
   (c) Porovnat výsledky všech implementovaných agentů testu proti agentům hlubokého Q-učení.

Struktura práce

V rámci této diplomové práce bude nejdřív představena historie měření inteligence (kapitola 1). Je důležité vymezit co vůbec inteligence znamená, protože i pohled na výklad tohoto pojmu se časem měnil, a i dnes je jeho výklad poněkud volný (sekce 1.1). Následně bude představeno měření inteligence umělých inteligencí od Turingova testu (v sekci 1.2), přes C-Test jakožto první změnu pohledu na celou problematiku testování inteligence (v sekci 1.3), až po koncept univerzální inteligence (sekce 1.4).

Univerzální inteligence není dokonalá, a je důležité toto brát v potaz. Bude proto představená důležitá kritika tohoto testu v podobě kdykoliv přerušitelného testu (sekce 1.5). Následně bude

představená a vysvětlená aproximace univerzální inteligence zvaná algoritmická inteligence (sekce 1.6). Ta umožnila vznik praktické podoby testu, který je využívaný v rámci této diplomové práce.

Dále budou představeny relevantní informace o umělých inteligencí (kapitola 2). Nejdříve budou vysvětleny neuronové sítě, jak fungují, jak se učí, a jaké výhody a nevýhody přináší (sekce 2.1). Dále bude vysvětlen koncept posilovaného učení, na kterém stojí praktický test algoritmického IQ s důrazem na to, jak prostředí fungují, jak probíhá komunikace s agentem, a jak se agent v takovém prostředí učí (sekce 2.2).

Na koncept posilovaného učení navazuje metoda Q-učení, která představuje možný přístup k učení agenta v rámci posilovaného učení (sekce 2.2.2). Pochopení Q-učení je důležité k pochopení hlubokého Q-učení, které propojuje Q-učení a neuronové sítě. Dále bude vysvětlena architektura a fungování hlubokého učení společně s její nadstavbou z Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) (sekce 2.3).

Pohledávání konfiguračního prostoru není jednoduchý úkol. Práce představí možnou metodu řešení tohoto problému v podobě evolučních algoritmů (sekce 2.4). Evoluční algoritmy poskytují možnost jak heuristicky prohledávat konfigurační prostor agentů, a využívají k tomu principy genetiky a evoluce.

Dále bude představena praktická implementace agentů hlubokého Q-učení (kapitola 3). Agenti společně využívají společné částí, jako například neuronovou síť nebo strukturu pro sbírání zkušeností (sekce 3.4). Poté bude rozebrána implementace původního agenta s jednou neuronovou sítí (sekce 3.5), a následně i implementace architektury s druhou neuronovou sítí target network (sekce 3.6).

Následně bude ukázána má implementace prohledávání konfiguračního prostoru agentů za pomocí evolučních algoritmů (kapitola 4). Budou ukázány výsledky tohoto prohledávání, které budou porovnány s výsledky agentů využívajících konfiguraci z původního článku (sekce 4.3).

Nakonec budou provedeny experimenty s nalezenými konfiguracemi. Nejdříve budou porovnány výsledky obou architektur mezi sebou (sekce 5.1). Poté budou testovány rozdíly prostého Q-učení a hlubokého Q-učení (sekce 5.2). Následně budou otestováni agenti s jejich nejlepšími konfiguracemi proti sobě (sekce 5.3).

1. Inteligence umělých inteligencí

Tato kapitola slouží jako vhled do testování obecné inteligence umělých inteligencí, tj. člověkem vytvořených strojů a programů. Obecnou inteligencí je zatím myšleno chápání inteligence v kontextu prosté intuitivní představy o lidské či zvířecí inteligenci. Kapitola proto rozebere pojem inteligence, protože je důležité vymezit, co vlastně očekáváme, že bude dělat umělé inteligence inteligentními. Dále probere myšlenky Alana Turinga, jeho testu a variant tohoto testu, aby bylo možné pochopit, jak se vyvíjelo hledání odpovědi na otázku inteligence umělých inteligencí. Hlavně se ale kapitola bude zabývat konceptem Univerzální inteligence, jako zatím nejrobustnějšímu konceptu testování inteligence umělých inteligencí.

1.1 Inteligence

Pod pojmem inteligence si určitě každý něco představí. Inteligentní školák nemá sebemenší problém s vyjmenovanými slovy a násobilkou, inteligentní středoškolák zakončí školu s maturitou se samými jedničkami, a třeba inteligentní zločinec dokáže úspěšně unikat zákonům. Každý zhruba víme, co inteligence je. Problém ale je vymyslet a vymezit přesnou definici tohoto pojmu. I samotní psychologové s tím měli problém.

Například Spearman (1904), který propojil znalosti psychologie a statistiky objevil korelaci v kognitivních testech a popsal tak inteligenci jako obecnou schopnost řešit kognitivní problémy. Tuto obecnou inteligenci nazval jako „g faktor“. K obecné inteligenci Spearman přijímá i existenci specifická inteligence, která se týká konkrétních schopností při řešení dílčích kognitivních úloh. Obecná inteligence je podle Spearmana nadřazená té specifické, a člověku s vyšší obecnou inteligencí se tak bude lépe dařit v kognitivních aspektech života lépe než člověku s nízkou obecnou inteligencí.

Tento pohled na inteligenci jako na jednu dominantní vlastnost dal za vznik řadě debat a kritik mezi psychology. V návaznosti na tento pohled vznikl i pohled přesně opačný, který říká, že inteligence se dělí na spoustu dílčích částí, které se sice vzájemně ovlivňují, ale ne natolik, aby se dalo mluvit o jedné propojující části. Za nejrobustnějším pohled z toho směru je brána Gardnerova teorie mnohočetné inteligence (Gardner, 1986). Gardner dělí inteligenci na tyto typy: verbální, logicko-matematická, prostorová, hudební, tělesně-kinestetická, interpersonální, intrapersonální, přírodní. Každý člověk má každou z těchto inteligencí rozvinutou různě, a celkový obraz těchto inteligencí tvoří to, co se nazýváme inteligencí.

Gardnerův pohled se zdá i podle selského rozumu správnější než pohled o jednotné inteligenci. Přece jen ze svého okolí známe lidi, kteří jsou zdatnější na sporty než na logické úlohy, a naopak. V odborné komunitě psychologů má ale tato teorie velkou řadu kritiků, například práce (Visser et al., 2006), a to hlavně z důvodu nedostatečného počtu empirických důkazů pro ověření této teorie. Je ale zajímavé, že z nějakého důvodu tato teorie zní správně, a ačkoliv není dokázaná, přispěla minimálně tím, že znovu vyvolala diskuzi na téma, co je inteligence, což poukazuje na to, že stoprocentní definici ještě nemáme.

Z důvodu různých pohledů na pojem inteligence je zajímavá jedna konkrétní práce (Gottfredson, 1997), která získala názory 52 odborníků na inteligenci a s ní související obory, a spojila je dohromady do jednotné definice, která zní:

> „Inteligence je velmi obecná duševní schopnost, která mimo jiné zahrnuje: schopnost uvažovat, plánovat a řešit problémy, abstraktní myšlení, chápání složitých myšlenek, rychlé učení a učení se ze zkušeností. Nejedná se pouze o učení se z knih, akademické dovednosti ani o chytrost při psaní testů. Inteligence spíše odráží širší a hlubší schopnost porozumět svému okolí – pochopit ho, dát věcem smysl nebo zjistit co dělat.“

Z této definice už lze převzít nějaké koncepty, podle kterých bychom mohli zjišťovat inteligenci umělých inteligencí. Rozhodně totiž chceme, aby umělá inteligence dokázala řešit problémy. Také určitě chceme, aby se dokázala učit a používala svoje dávné zkušenosti. Zároveň očekáváme, že umělá inteligence bude schopná porozumět svému okolí, pochopit aktuální situaci, a adekvátně reagovat. Pojem inteligence tedy odteď vnímáme jako spojení těchto vlastností a schopností. Jak je ale možné tyto všechny vlastnosti měřit a zjistit, zda je něco inteligentní? Jak tato měření aplikovat bez omezení na jednu formu inteligence (tedy stejná metodika půjde použít u člověka, zvířete i stroje)?

1.2 Turingův test a jeho nástupci

Když se Alan Turing na počátku padesátých let minulého století zamýšlel, jak zjistit, jestli dokáží stroje myslet, jen těžko měl v hlavě takovou definici myšlení a inteligence jako z předchozí podkapitoly. Jeho přístup k této otázce je ale i tak neméně zajímavý a důležitý. Turing (1950) k ověření inteligence stroje pracoval s poněkud dystopickou otázkou a to: „Dokáže stroj přesvědčit člověka o tom, že aktuálně jedná s jiným člověk?“ Pro zjištění odpovědi na tuto otázku použil Turing Imitační hru.

Imitační hra v původním znění, které se původně netýkalo inteligence strojů, pracuje se třemi účastníky: mužem (A), ženou (B) a nějakým vyšetřovatelem. Vyšetřoval sedí v oddělené místnosti od A i B, muž i žena dostanou náhodně označení X a Y, a vyšetřovatel si dopisuje s oběma účastníky a pokládá jim otázky. Jeho úkolem je na konci hry říct, jaké pohlaví je X a jaké pohlaví je Y.

Tuto hru změníme na test inteligence stroje tak, že muže a ženu nahradíme za libovolného člověka a testovaný stroj. Vyšetřovatel si tak dopisuje buď s člověkem nebo se strojem a jeho úkolem je rozhodnout, kdo je člověk, a kdo je stroj. V tomto provedení nazýváme tuto hru jako *Turingův test* (Turing, 1950).

Turingův test je ve své podstatě velice jednoduchý, když se ale nad testem zamyslíme, objevíme jeho nedostatky. Prvním možným problémem může být to, že výsledek testu je do velké míry ovlivněn vyšetřovatelem. Ten například pokud má velké empatické cítění, tak dokáže do nějaké míry rozeznat stroj podle, jeho „nelidských“ odpovědí na některé osobní a hluboké otázky. Neempatického člověka, který se bude ptát pouze na faktické otázky, lze zmást daleko jednodušeji. Na druhou stranu od inteligentního stroje v tomto pojetí očekáváme, že bude schopen dávat najevo i právě tyto lidské vlastnosti, aby dokázal zmást i právě takto empatického člověka.

Testuje ale vlastně Turingův test nějak inteligenci? Ve své podstatě ani ne, test pouze zjišťuje, zda stroj dokáže odpovídat na otázky tazatele tak, aby ho dokázal zmátnout. V dnešní době už dokonce díky text-generujícím umělým inteligencím dokážeme tento test překonat, což ukazuje například (Nov et al., 2023). Na tento problém, ale upozorňoval už dříve (Searle, 1980), který kritizoval právě Turingův test kvůli tomu, že zjišťuje pouze zda stroj dokáže dostatečně dobře napodobovat člověka, ale nezjišťuje nijak úroveň jeho inteligence.

Searle (1980) toto demonstroval na své hypotetické verzi testu nazvanou jako Čínský pokoj. Tento nový test představuje situaci, kdy je člověk zavřený v pokoji a pod dveřmi jsou mu podávány papírky se zprávami obsahující samé čínské znaky. V pokoji je k dispozici návod, jak pracovat s čínskými symboly na základě jejich tvaru a pořadí. Podle tohoto návodu dokáže člověk v pokoji sestrojit odpověď a poslat ji zpět pod dveřmi. Zvenčí to vypadá, že osoba v pokoji opravdu umí čínsky a konverzace dále pokračuje. Ve skutečnosti ale osoba v pokoji nemá nejmenší tušení, co za zprávu jí přišlo a co poslala zpět. Stroj, který má k dispozici stejná pravidla by byl také schopen úspěšně odpovídat, ale také by neměl nejmenší tušení o

významu zpráv. Turingův test je tak možné ošálit obyčejným programem, který bude mít k dispozici dostatečně robustní seznam pravidel, jejichž významu nebude ale vůbec rozumět.

Úplný Turingův test (Harnad, 1991) je také kritikou původního Turingova testu a zároveň návrhem jeho možného zlepšení. Hlavní myšlenkou je to, že testovaný jedinec musí být mimo odpovídání i schopný něco vykonávat v reálném světě podle předchozí komunikace. K jazykové rovině se tak do testu přidává i rovina vztahová, kdy testovaný stroj musí rozumět vztahům slov k reálným objektům, a musí být schopen pochopit, co se po něm chce. Tím že stroji zadáme slovně nějaký úkol a budeme schopni na vlastní oči vidět, že stroj úkol pochopil a splnil, tak to nám má říct, že máme před sebou něco, co oplývá inteligencí, alespoň právě podle Harnada.

Úplný i normální Turingův test spolu ale sdílí ten samý problém. Oba testy se zakládají na lidském chování, řeči a interakcích člověka se světem jako měřítku inteligence. Tato perspektiva však čelí omezením, protože oba modely zdůrazňují lidský jazyk jako primární prostředek pro vyjádření inteligence. Tím se přehlíží skutečnost, že inteligence existuje i mimo lidský kontext, a že i jiné živočišné druhy projevují známky inteligentního chování, bez schopnosti rozumět lidskému jazyku. Je proto klíčové nahlížet na testování inteligence tak, aby zahrnovala inteligenci v širším pojetí bez vztahu ke světu lidí.

1.3 C-Test

Turingův test má tedy své problémy při praktickém řešení otázky, zda mohou stroje myslet. Existuje ale například i názor, že Turing test nezamýšlel, jako prakticky použitelnou pomůcku, pro ověření inteligence stroje, nýbrž jako stavební kámen pro výzkumníky umělé inteligence, kteří se od něj mohli odrazit a navrhnout vlastní řešení testu inteligence (Fostel, 1993).

Turingův test jistě odhalil čeho se při testování inteligence vyvarovat, a pomohl najít vlastnosti, které by správný test měl mít. Příklady takových vlastností udává Hernandez-Orallo (2000). Podle něj je důležité, aby test:

- nebyl booleovský,
  - Nedá se říct, že něco je inteligentní a něco není. Inteligence je spojitá veličina, která jde popsat nějakým rozsahem hodnot.
- byl rozdělitelný,
  - Inteligence nejspíš není prostá jednodílná věc, ale jedná se o komplexní pojem zahrnující spoustu různých aspektů. Ačkoliv nejspíš existuje propojující g-faktor, test by se neměl omezit jen na tuto společnou schopnost, ale měl by testovat inteligenci z vícero pohledů.
- nebyl antropomorfní,
  - Existují i jiné druhy inteligence než ta lidská. Test inteligence by se tak měl zaobírat všeobecností a upustit od lidského vnímání inteligence.
- byl založený na využití počítačů,
  - Testování inteligence, a hlavně pak inteligence umělých inteligencí by mělo být algoritmizovatelné a vypočitatelné pomocí počítačů.
- byl smysluplný,
  - Inteligence by neměla mít význam toho, co je měřené pomocí testů inteligence. Inteligence by měla vyjadřovat nějakou smysluplnou a pochopitelnou konkrétní schopnost, a tu by měl správný test měřit.

Hernandez-Orallo (2000) v návaznosti na předchozí vyjmenované vlastnosti testu inteligence přichází s vlastním testem nazvaný C-Test (C pochází z anglického slova Comprehension, které znamená *Porozumění*). Hlavní myšlenkou C-Testu je představit testovanému subjektu nějakou úvodní situaci, a chce se po něm, aby situaci porozuměl a dokázal ji vysvětlit, popřípadě doplnit. Výsledná inteligence je pak odvozena od počtu vyřešených situací, ale také podle správnosti řešení a i času, který trvalo subjektu situaci vyřešit.

Nejjednodušším příkladem řešených situací může být test doplňování posloupnosti znaků. Testovaný subjekt je do situace uveden tak, že mu je předán kus posloupnosti na jejímž základě by mělo být možné říct, jak bude sekvence pokračovat, nebo doplnit znak na vynechané místo. Příkladem takové úlohy může být například sekvence *a, c, e, g,* _, kdy jako další symbol je očekáván znak *i*.

Zajímavý je Orallův pohled na obtížnost úloh. Navrhuje totiž, aby byly subjektu nejdříve

předloženy ty nejtěžší úlohy, a každá následující by měla být vždy jednodušší nebo stejně složitá, jak předchozí. Výsledkem je tak zjištění, jak dobře se dokáže testovaný subjekt přizpůsobit a nalézt vysvětlení pro úlohy, kterým se zvyšuje pochopitelnost postupně.

Na problém tohoto testu poukazuje práce (Legg a Hutter, 2007), ve které při analýze možných testů inteligence vadilo autorům, že test je příliš statický. Testovanému subjektu jsou předloženy všechny informace naráz a je jen na něm, jak s těmito informacemi naloží a úlohu vyřeší. Oproti tomu dynamický test by obnášel nutnost nějak aktivně se zapojit a komunikovat se svým okolím nebo kontextem dané úlohy, aby ji byl schopen vyřešit. Test by tak byl více interaktivní a poskytoval zpětnou vazbu v reálném čase. Oproti statickému testu se tak více testuje řešení problémů z pohledu toho, jak se řeší problémy v reálném světě – musí se nejdřív hodně krát selhat, aby se zjistilo, jak lze uspět.

Kromě toho je ale C-test vynikajícím konceptem pro zkoumání schopnosti porozumění situacím. Autor tohoto testu jej opřel o řadu teoretických konceptů jako je Kolmogorova složitost (Kolmogorov, 1968) nebo Solomonoffovo induktivní odvozování (Solomonoff, 1964). Solomonoffovo induktivní odvozování se zaměřuje na hledání programů generující nějakou posloupnost, které představují onu situaci, kterou agent musí pochopit. Na takto nalezené programy navazuje druhý koncept, Kolmogorova složitost. Ta udává složitost řetězce jako délku nejkratšího programu, který může daný řetězec vygenerovat. Nalezené programy jsou tak vždy podrobeny Kolmogorově složitosti s cílem nalézt ten nejkratší možný program, a právě tím popsat danou situaci. Obtížnost absolvování testu tak není nějaká subjektivní hodnota, ale je měřena zcela objektivními mírami, které vychází právě z těchto konceptů.

1.4 Testování univerzální inteligence

Doposud v práci zazněly možné pohledy, jak se dívat na pojem „inteligence“. Zazněla také otázka „jak zjistit, že je něco inteligentní“ a jak tuto otázku aplikovat na člověkem vytvořené stroje a systémy viz Turing (1950). Zazněly i konkrétní návrhy, jak by šla inteligence těchto systémů měřit. Tato kapitola bude věnována v dnešní době jednomu z nejrobustnějších koncept inteligence umělých inteligencí, které vychází, učí se z chyb, a navazuje na témata, která doposud zazněla v této kapitole. Tento koncept se nazývá Univerzální inteligence, jeho význam rozebere obsah této kapitoly.

1.4.1 Definice univerzální inteligence

Pro každý správný test inteligence, by se mělo nejdříve vymezit, co vlastně „inteligence“ znamená, a co tedy vlastně budeme testovat. Hutter (2007) kvůli tomu v rámci své rešerše pro Univerzální inteligenci prošli desítky definic od různých psychologů a neurovědců s cílem vytvořit takovou definici, která by byla jednoduše uchopitelná, odrážela by obecné přesvědčení o tom, co inteligence znamená, a zároveň vedla na nějakou matematickou formuli, díky které by ji šlo vyhodnotit. Výsledná definice, se kterou přišli, zní takto:

> „Inteligence měří agentovu schopnost dosahovat cílů v široké množině prostředí.“

Agent, je onen testovaný subjekt, inteligentní entita, o které zjišťujeme, jestli je inteligentní, a do jaké míry je inteligentní. Agent musí být schopen komunikovat s prostředím a ovlivňovat ho, a naopak prostředí mu musí být schopné odpovídat a dávat zpátky zpětnou vazbu. Pro účely této definice musí existovat pro agenta i nějaký cíl. Cílem je něco, čeho se agent musí snažit dosáhnout tím, že právě ovlivňuje dané prostředí. K tomu, aby byl agent schopen poznat, co je jeho cílem, získává od prostředí odměnu, která mu říká, jak si aktuálně vede. Jeho cílem je tedy ve finále dosáhnout nejvyššího možného součtu získaných odměn.

Pokud si vzpomeneme na definici inteligence z konce kapitoly 1.1, tak ta zněla následujícím způsobem: „Inteligence je velmi obecná duševní schopnost, která mimo jiné zahrnuje: schopnost uvažovat, plánovat a řešit problémy, abstraktní myšlení, chápání složitých myšlenek, rychlé učení a učení se ze zkušeností. Nejedná se pouze o učení se z knih, akademické dovednosti ani o chytrost při psaní testů. Inteligence spíše odráží širší a hlubší schopnost porozumět svému okolí – pochopit ho, dát věcem smysl nebo zjistit co dělat.“ (Gottfredson, 1997)

Definice z Hutter (2007) na první pohled pokrývá hlavně část „řešit problémy“, protože prostředí reprezentují nějaký interaktivní problém. K tomu, aby někdo dokázal řešit nějaký problém, tak by měl být schopný ho pochopit, a měl by být schopný i pochopit jeho kontext v rámci, kterého se problém řeší. Tím, že test inteligence vychází z této myšlenky a využívá k tomu interakce s prostředím, tak vzniklá dynamický test, který umožňuje agentovi prozkoumávat možnosti a učit se. Očekáváme totiž, že aby byl agent schopný řešit problémy v různých

prostředích, tak musí oplývat schopností dané prostředí pochopit (zahrnuje vlastně myšlenku C-Testu (Hernandez-Orallo, 2000) z dřívějška), a zároveň právě schopností učit se, aby byl schopen nalézt nejlepší řešení daného problému, a dokázat ho zopakovat. Do nějaké míry tedy tento test zjišťuje i schopnosti chápat a učit se. Zda agent bude schopen abstraktního myšlení asi není vlastnost, kterou tímto testem dokážeme zjistit, ale mimo to tato jednoduchá definice docela pokrývá obecnou představu o inteligenci vymezenou dříve.

Z první části definice „inteligence měří agentovu schopnost dosahovat cílů“ nemusí být hned tak jasné, co znamená „dosahovat cílů“, nic méně právě toto mají autoři nejlépe podchycené. Tím, že každé prostředí modeluje nějaký problém, tak agentovým cílem je prostředí pochopit a problém vyřešit. Jak je ale toto agentovi komunikováno? V prostředí posilovaného učení se používá pojem odměna. Agent komunikuje s prostředím tak, že provede nějakou akci, a pokaždé, co tak učiní, tak mu prostředí poskytne informaci o tom, co se změnilo a poskytne mu nějakou odměnu. Odměna slouží jako takové vodítko, které říká agentovi, jestli jeho poslední akce přispěla k vyřešení problému nebo nikoliv (odměna může být i záporná a agent tak může být „potrestán“). Od inteligentního agenta očekáváme, že jen z pomocí odměn bude schopen problém daného prostředí pochopit, a bude schopný se naučit jeho řešení. Výsledek agenta v rámci jednoho prostředí je poté součet získaných odměn během jednoho řešení daného problému. Dosahovat cílů tedy znamená, naučit se, jak získat co nejvíce těch nejvyšších možných odměn v daném prostředí.

Poněkud náročnější otázka na uvedenou definici je, co znamená „v široké množině prostředí?“ Prostředí, jak již zaznělo, představuje nějaký problém. Inteligence by měla znamenat, že agent dokáže řešit spoustu různorodých problémů. Ideálně bychom měli měřit inteligenci na všech možných rozdílných problémech, abychom si mohli být jistí agentovými znalostmi.

Agentův výsledek v daném prostředí by měl být různě důležitý pro finální výsledek, pokud prostředí reprezentuje složitý problém oproti prostředí, které představuje jednoduchý problém. Určit složitost prostředí, ale není tak přímočaré, jak by se mohlo zdát. Stejný problém lze totiž reprezentovat různě složitým zadáním. (Hutter, 2007) toto ukazují na následujícím příkladu: Dostaneme jako zadání sekvenci čísel, například 2,4,6,8 a úkolem bude doplnit číslo, které následuje. To je docela jednoduchý úkol, další hodnota bude 10, a pravidelnost je v tom, že každý k-tý prvek je roven hodnotě $2k$. Každopádně řešením by mohlo být klidně i číslo 58, podle polynomu $2k^4 - 20k^3 + 70k^2 - 98k + 48$. Proč tedy první intuitivní řešení je 10, když existují i jiné možnosti. Odpovědí je koncept zvaný Occamova břitva. Occamova břitva říká velice jednoduše, že nejpravděpodobnější řešení problému je to nejjednodušší. Po inteligentním agentovi tak chceme, aby byl schopný s tímto konceptem pracovat a k jednotlivým prostředím takto přistupoval. Zároveň je ale potřeba, aby řešení problému v prostředích také odpovídala konceptu Occamovy břitvy.

Formálně k popisu složitosti prostředí používá Hutter (2007) koncept Kolmogorovy složitosti (Kolmogorov, 1968). Kolmogorova složitost binárního řetězce x je definována jako délka nejkratšího programu, který dokáže vytvořit x:

$$K(x) := \min_p \{l(p) : U(p) = x\} \tag{1.1}$$

kde p je binární řetězec, který nazýváme program, $l(p)$ je poté délka tohoto řetězce v bitech, a U takzvaný *referenční stroj*. Program je v našem kontextu takový program, který simuluje prostředí představující konkrétní problém.

Pro pochopení Kolmogorovy složitosti si představme si, že máme řetězec, který tvoří milion nul. I když tento řetězec obsahuje milion hodnot, vytvoření programu, který tento řetězec vypíše milionkrát, vyžaduje jen několik instrukcí. Naopak, pokud máme řetězec plný náhodných hodnot, jako je například sekvence 11011010001011001, program generující tuto posloupnost bude muset použít mnohem více instrukcí než program vypisující nuly, a tudíž bude mít z pravidla vyšší Kolmogorovu složitost.

Důležitou vlastností Kolmogorovy složitosti je nezávislost na výběru referenčního stroje U. Referenční stroj představuje abstraktně programovací jazyk nebo popřípadě nějaký stroj či systém, který splňuje Turingovskou úplnost, a tedy dokáže spustit program, který dokáže spustit i Univerzální Turingův stroj (Turing, 1937) Pokud vyměníme U, které toto splňuje za nějaký jiný univerzální referenční stroj U', tak na stroji U' z podstaty věci půjde vytvořit program, který bude simulovat chování U, a díky tomu bude možné vypočítat $U(p)$. Pokud bychom zkusili vypočítat Kolmogorovu složit s použitím U', který simuluje U, tak se výsledná složitost zvýší o délku programu, která simuluje U. Toto zvýšení bude ale stejné pro všechny možné vstupy x na daném referenčním stroji, a tudíž závislost referenčního zdroje je v tomto smyslu zanedbatelná a podstata testu zůstává zachována. Tato vlastnost ukazuje univerzálnost této složitosti, která je důležitá, protože z podstaty Turingova stroje ho stejně budeme muset simulovat v nějakým jiném výpočetním prostředí.

Pro použití v testu univerzální inteligence se Kolmogorova složitost používá k vyjádření složitosti programu, který popisuje dané prostředí. Tyto programy dokážeme zakódovat do binárních řetězců, které se označují jako i. Myšlenka za zakódováním každého prostředí řetězcem je taková, že složitá prostředí budou zakódovaná dlouhým a složitým řetězcem, a jednoduchá prostředí budou zakódována krátkým a jednoduchým řetězcem. Kolmogorova složitost pak bude vyjadřovat délku nejkratšího programu generující označení nějakého prostředí, a ta bude tím pádem přímo úměrná složitosti daného prostředí. Toto se dá popsat následujícím vztahem: $K(\mu_i) = K(i)$, kde symbol μ označuje prostředí.

Nicméně programů popisujících prostředí je mnoho, a proto existuje i potenciálně mnoho různých binárních řetězců představující tyto různé programy. Abychom tak zachovali koncept Occamovy břitvy, je potřeba programům přiřadit pravděpodobnost jejich výběru, která se odvíjí od jejich složitosti. Tím vznikne rozdělení nad množinou všech programů, které popisují dané prostředí, a toto rozdělení označujeme za tzv. algoritmické rozdělení pravděpodobnosti. Je důležité, že v tomto rozdělení se snižuje pravděpodobnost výběru programu s každým jeho bitem. Z důvodu toho, že bit může nabývat dvou hodnot, tak s každým bitem klesá pravděpodobnost výběru programu dokonce o hodnotu $\frac{1}{2}$. Algoritmické rozdělení pravděpodobnosti je definováno předpisem $2^{-K(\mu)}$. Toto rozdělení vnáší do práce s prostředími probabilismus, a zároveň poskytuje řadu vlastností, které řeší hlavně induktivní odvozování (Hutter, 2007).

Nyní známe všechny potřebné koncepty, ze kterých Hutter (2007) poskládali definici Univerzální

inteligence. Univerzální inteligence Υ agenta π má formálně následující předpis:

$$\Upsilon(\pi) := \sum_{\mu \in E} 2^{-K(\mu)} V_\mu^\pi \tag{1.2}$$

,kde E představuje množinu všech prostředí μ s počitatelnou distribucí odměn na Turingově stroji U. $2^{-K(\mu)}$ označuje algoritmické rozdělení pravděpodobnosti podle Kolmogorovy složitosti prostředí $K(\mu)$ a V_μ^π je očekáváný součet odměn agenta π v prostředí μ neboli hodnotová funkce.

Univerzální inteligenci agenta zjistíme tedy tak, že ho necháme působit ve všech prostředích z množiny E. Tím že agent v prostředí působí, tak získává odměny, z jejichž součtu získáme výsledek hodnotové funkce V. Tento výsledek vynásobíme složitostí prostředí, která je vyjádřená rozdělením $2^{-K(\mu)}$ všech programů popisující dané prostředí. Pokud sečteme všechny tyto výsledky napříč všemi prostředími z E, tak získáme hodnotu Univerzální inteligence agenta. Tato rovnice tak odpovídá definici z dřívějška, a to: „Inteligence měří agentovu schopnost dosahovat cílů v široké množině prostředí."

Hutter (2007) ale upozorňují, že takto vymezená definice bohužel není jednoduše převeditelná na praktický test, ale jedná se pouze o koncept inteligence umělých inteligencí v nejobecnějším a zároveň v nejrobustnějším pojetí. Problémem je, že Kolmogorova složitost není spočitatelná, ale lze pouze odhadovat její hodnotu. Problematika hledání nejkratšího program odkazuje na tzv. problém zastavení, který spadá v teoretické informatice do nerozhodnutelných problémů (Sipser, 2013). Další problém je, že výsledek hodnotové funkce prostředí může být různě velký v různých prostředích. Některá prostředí mohou vést dokonce na výsledky sahající k nekonečnu. Hutter (2007) z toho důvodu omezují výsledek hodnotové funkce shora hodnotou jedna. Nakonec posledním očividným problémem je množina prostředí E. Ta je totiž nekonečná, praktický test tak dokáže maximálně odhadnout inteligenci na základě nějakého vzorku z této množiny.

1.4.2 Očekávaná inteligence různých agentů

Hutter (2007) ve své práci velice chytře ukazují, proč je koncept Univerzální inteligence užitečný na jednoduchém myšlenkovém experimentu. Jejich úvaha zároveň hezky ukazuje na to, co musí agent splňovat a umět, aby dosáhl vysoké hodnoty Υ.

Nejhorší hodnotu Univerzální inteligence získá agent, který se chová naprosto náhodně, a tedy v každém kroku vybírá svou akci náhodně. A to z toho důvodu, že ačkoliv kvůli povaze některých prostředí sice získá nějaké odměny tak o to, že jeho celkové skóre bude velice nízké, se postará agregace výsledků ze všech prostředí, a ne pouze z té malé množiny, kde se náhodným výběrem trefí do akcí s navázanou odměnou.

Dále uvažujme specializovaného agenta, který sice dokáže v nějakém velmi specifickém prostředí získat extrémně vysoké skóre, ale mimo něj nedokáže vyřešit žádný jiný problém. Takový agent může být například bot hrající šachy. Dá se očekávat, že v prostředí představující šachy

získá agent vysoký součet odměn, nicméně u každého jiného prostředí bude jeho výsledek takřka nulový. Agregace výsledků se opět postará o to, že celková univerzální inteligence specializovaného agenta bude velice nízká.

Nejlepších výsledků dosáhne nějaký obecný agent, který dokáže prozkoumávat prostředí, a tedy bude se záměrně dostávat do nových situací, ve kterých bude zkoušet dělat různé akce. Během toho bude od prostředí získávat různě velké odměny. V závislosti na jeho schopnosti spojit si tyto odměny se svým chování se naučí zákonitosti daného prostředí, které povedou k získávání dalších odměn. Takový agent si nejspíše povede dobře na jednodušších prostředích, ale to, co bude tyto obecné agenty mezi sebou odlišovat, je schopnost chápat vztahy složitých prostředí, a nacházet v nich vhodná řešení.

1.5 Kdykoliv přerušitelný test inteligence

Kdykoliv přerušitelný test inteligence (Hernández-Orallo a Dowe, 2010), je velice důležitá práce, protože rozebírá na nedostatky konceptu Univerzální inteligence z předchozí podkapitoly. Poukazuje hlavně na její problémy týkající se nepraktičnosti testu v ohledu na reálné použití a sama práce představuje nové koncepty a návrhy řešení diskutovaných problémů, a to hlavně nevyčíslitelnosti testu. Zároveň ale práce sama představuje některé nové koncepty testování inteligence, ale jen formou formální specifikace a bohužel ne praktické implementace.

Prvním problém, na který je v rámci práce upozorňováno, je vzorkování a výběr prostředí. Použití rozdělení založené na Kolmogorově složitosti (Kolmogorov, 1968) značně upřednostňuje prostředí jejíž nejkratší programy jsou opravdu krátké, a tedy představují jednoduché problémy. Zajímavější závěry se ale získají, když budou agenti testováni na složitějších prostředích. Je ale zároveň také poukazováno na to, že delší program nutně nemusí odpovídat složitému prostředí. Některá prostředí mohou popisovat složité vztahy i s použitím krátkého programu, zatímco některá prostředí popsaná dlouhým programem mohou popisovat jednoduchý problém, ale uměle ho zdržovat zbytečnými instrukcemi. Je také potřeba zařídit, aby se pracovalo pouze s užitečnými prostředími, některá prostředí mohou být zbytečná, protože nepracují s akcemi agenta, a dávají konstantní odměnu nezávisle na svém stavu, nebo prostředí která jsou velice podobná jiným prostředím z celkové množiny všech prostředí.

Déle je v rámci Kdykoliv přerušitelného testu poukazováno na problémy prostředí citlivých odměny. Taková prostředí se vyznačují tím, že úspěšnost agenta se odvíjí od odměn, ke kterým se dostane, a které získá. Problém je, pokud je prostředí natolik složité, že v závislosti na vnitřním stavu stejné sekvence akcí poskytují jiné výsledky. Je poukazováno, že problém nastane, pokud prostředí obsahuje takzvané Nebe a Peklo. Nebe je označení pro situace, kdy agent po provedení určité sekvence akcí získá extrémně vysoké odměny. V takové situaci agent ztrácí chuť prozkoumávat zbytek prostředí a uvízne v tomto bodě navždy, neboť je spokojen s odměnami, které získává. Peklo je opačné označení pro situaci, kdy se agent dostane do takového bodu, že všechny akce poskytují natolik nízkou odměnu, že agent bude mít problém najít z tohoto bodu cestu, protože žádná akce nedokáže přinést lepší odměnu.

Očividným problémem Univerzální inteligence je použití Kolmogorovy složitosti a její nevyčíslitelnost. V kdykoliv přerušitelném testu je poukazováno na to, že vhodnou náhradou Kolmogorovy složitosti je Levinova K't složitost (Levin, 1973), která je v podstatě aproximací Kolmogora. Je ale poukazováno, že při použití Levinovy složitosti je žádoucí stanovit maximální čas na reakci prostředí na zvolenou akci. Ušetří se tak čas vyhodnocování programů prostředí, které jsou sice krátké, ale kvůli vnitřním procesům uměle prodlužují chování prostředí. V rámci testu, tak budeme mít záruku, že všechna prostředí mají stejnou maximální dobu reakce.

Univerzální inteligence je ještě kritizovaná v ohledu výpočtu celkového skóre, kdy se uvažuje nekonečný interakční čas s prostředím, a celková suma odměn je omezená shora hodnotou jedna. V rámci kdykoliv přerušitelného testu je navrhováno použít omezený počet interakcí s

prostředím a poté získanou sumu odměn zprůměrovat právě počtem interakcí. Předpoklad průměrování je ale použití takzvaných vyvážených prostředí, kde se odměny vyskytují v intervalu $< -1; 1 >$. Náhodné chování agenta by tak vedlo na výsledky, které by v průměru vedly na hodnotu kolem nuly.

Kdykoliv přerušitelný test nezávisle na Univerzální inteligenci poukazuje na problém, kdy obecně přístupy k testování inteligence umělých systémů neberou v potaz, jak rychle se systém naučil řešení daného prostředí, a jak rychle dokáže systém reagovat. Pokud by dva různí agenti (systémy) měli výsledky stejně dobré, ale jeden agent by je získal za desetinu času, co druhý agent, tak by přece měl být rychlejší agent označen za chytřejšího. Kvůli tomu je ale nutné nejdříve zajistit, aby prostředí reagovala okamžitě, jinak by se totiž v testu, který bere v potaz reálný čas, agentovi značně ublížilo, kdyby byla na straně prostředí prodleva. Kvůli praktičnosti je ale ještě lepší možnost tento interakční čas vzít v potaz při výpočtu odměny nebo skóre a odstranit tak závislost na reakčním čase prostředí, čímž se bude prostředí při finálním výpočtu tvářit jako prostředí s okamžitou reakční dobou.

Důležitá vlastnost navrženého testu je ale práce v omezeném čase. Chceme, proto aby test měl následující vlastnosti: pokud poskytneme testu málo času, tak získáme velice hrubý odhad inteligence systému, naopak čím víc času ale přidáme, tím přesnější odhad získáme. V prvotních okamžicích testu chceme získat co nejlepší představu o schopnostech testovaného agenta, zajímá nás, kolik času potřebuje k řešení úloh, a na jaké složitosti tyto úlohy začínají. Pokud toto test stihne zjistit, tak nám dokáže poskytnout prvotní odhad úrovně agentových schopností.

Test proto nejdříve předkládá testovanému subjektu velmi jednoduché problémy a dává na vyřešení velmi málo času. Postupně ale testuje na složitějších a složitějších problémech, ke kterým zároveň poskytuje více a více času. Pokud úlohu testovaný subjekt nestihne vyřešit, tak test sníží obtížnost úkolu. Během testování se čas na vyřešení bude zvyšovat pokaždé o daný kus, a komplexita problémů se bude měnit podle získaného skóre. Pokud systém nestihl nasbírat skóre v daném čase, tak se komplexita sníží. V opačném případě se komplexita zvýší úměrně tomu, jak vysoké skóre systém stihl získat, což pomůže se dostat na adekvátní úroveň schopností agenta. Celková inteligence je získána jako součet všech získaných skóre, na kterých byl agent testován, vydělených právě tímto počtem.

1.6 Algoritmický inteligenční kvocient

Legg a Hutter (2007) ve své práci vymezili univerzální inteligenci, která je popsaná v předešlých podkapitolách, ovšem univerzální inteligence má jeden problém – nelze ji vypočítat. Univerzální inteligence pracuje s koncepty jako je množina všech prostředí E a Kolmogorova složitost. Na tyto problémy reaguje Kdykoliv přerušitelný test (Hernández-Orallo a Dowe, 2010), představený v předchozí podkapitole, a který poskytuje návrhy jejich řešení, nebo alespoň částečně. Pro připomenutí problém s množinou všech prostředí je v zásadě ten, že je nekonečná. Kolmogorova složitost má zase v reálném použití ten problém, že nelze najít pro prostředí ten nejkratší možný program, který ho popisuje. To ale neznamená, že univerzální inteligence, tak jak je vymezená, nemá reálné využití. Legg a Veness (2011) ve své práci navazují na univerzální inteligenci, přejímají některé koncepty z Kdykoliv přerušitelného testu, a popisují možnou aproximaci univerzální inteligence, kterou označují jako algoritmický inteligenční kvocient (algorithmic intelligence quotient), zkráceně AIQ.

Kolmogorova složitost se v Univerzální inteligenci používá k tvorbě algoritmického rozdělení, díky kterým dokážeme zjišťovat složitost každého prostředí s ohledem na nejjednodušší programy, které prostředí popisují. Možnou náhradou algoritmického rozdělení je tzv. univerzální rozdělení vycházející z Levinovy K^t složitosti (Levin, 1973). To v praxi znamená, že pokud máme sekvenci bitů (binární řetězec), tak můžeme použít referenční stroj k odvození pravděpodobnosti, že se tato sekvence objeví. Pravděpodobnost, že sekvence x začíná konečným binárním řetězcem se označuje $M_U(x)$ a má předpis:

$$M_U(x) := \sum_{p:U(p)=x*} 2^{-l(p)} \tag{1.3}$$

, kde iterujeme přes všechny programy p, které když se spustí na referenčním stroji U, tak vytvoří sekvenci bitů $x*$, která začíná posloupností x. $l(p)$ je poté délka takového programu.

Jelikož Kolmogorova složitost $x*$ je délka nejkratšího programu generující řetězec $x*$, tak podle definice největší prvek z rozdělení M je roven $2^{-K(x*)}$. V souladu s Occamovou břitvou bude množina všech posloupností začínající binárním řetězcem s nízkou složitostí mít vysokou pravděpodobnost podle M. Rozdíl je, že nyní se pracuje s délkou všech programů generující řetězec začínající sekvencí $x*$ namísto jen nejkratšího programu. Výhodou tohoto rozdělení je, že se z něj daleko jednodušeji provádí náhodný výběr, jelikož pravděpodobnost výběru programu p rovnoměrným vzorkováním po sobě jdoucích bitů je rovna $2^{-l(p)}$. Pro náhodný výběr sekvence z M provedeme náhodný výběr programů p a spustíme ho na referenčním stroji U. V podstatě se ale provádí náhodný výběr n programů $S = p_1,p_2,\cdots,p_n$ náhodným přidáváním bitů, dokud programy nejsou kompletní. Výsledné S ale není množinou, protože nic nebrání možnosti, že se nějaký program v sekvenci objeví vícekrát.

S takto vygenerovanými programy můžeme odhadnout agentovu univerzální inteligenci podle vztahu:

$$\hat{\Upsilon}(\pi) := \frac{1}{N}\sum_{i=1}^{N}\hat{V}_{p_i}^{\pi} \quad , kde \qquad \hat{V}_{p_i}^{\pi} := \frac{1}{N}\sum_{j=1}^{k} r_j \tag{1.4}$$

, kde je nahrazena očekávaná odměna prostředí V_μ^π za $\hat{V}_{p_i}^\pi$, tedy za celkovou sumu odměn z jednoho průchodu prostředím $U(p_i)$. Celková suma odměn nyní uvažuje konečný počet iterací, jak navrhl Kdykoliv přerušitelný test (Hernández-Orallo a Dowe, 2010), a akumulované odměny v rámci jednoho prostředí jsou tak zprůměrovány počtem iterací. Výsledek sumy všech odměn napříč všemi prostředími z testu je nakonec zprůměrován vydělením počtem všech těchto prostředí N. Jelikož provádíme náhodný výběr programů namísto prostředí, může se tak stát, že vícero různých programů bude definovat stejné prostředí. Ze vzorce také zmizela část přiřazující programům váhu $2^{-l(p)}$, a to kvůli tomu, že je zahrnutá ve způsobu výběru vzorku S, kdy samotná šance výběru programu klesá s každým jeho bitem. Agent tak bude přirozeně testován více na jednodušších prostředích a méně na složitějších.

1.6.1 Implementace algoritmického inteligenčního kvocientu

Dalším krokem pro implementaci praktické verze Univerzální inteligence byla pro Legg a Veness (2011) volba referenčního stroje, jehož programy budou tvořit množinu prostředí. Referenční stroj by se měl ideálně svou strukturou podobat skutečnému světu, aby jeho programy odpovídaly problémům, které se skutečně vyskytují v reálném světě. Z praktického hlediska je ale lepší zvolit takový jazyk, pro který se budou jednoduše tvořit funkční programy. Takové jazyky musí ideálně splňovat, že libovolná kombinace jejich instrukcí bude tvořit syntakticky správný program. Legg a Veness (2011) se z tohoto důvodu rozhodli pro jazyk BF (Easter, 2020), v němž se jednoduše tvoří validní programy díky tomu, že se skládá pouze z 8 prostých symbolů. Tyto symboly a jejich význam lze vidět v Tabulce 1.1. Instrukce BF odráží instrukce Turingova stroje, každopádně k původním 8 byly pro potřeby testu přidány ještě symboly # a %, které nejsou pro BF standardní.

Tabulka 1.1: instrukce jazyka BF

BF	Význam	Alternativa v C
>	Posun ukazatele vpravo	`p++;`
<	Posun ukazatele vlevo	`p−;`
+	Zvýš hodnotu buňky o jedna	`*p++;`
-	Sníž hodnotu buňky o jedna	`*p−;`
.	Zapiš hodnotu na výstup	`putchar(*p);`
,	Načti vstup	`*p = getchar();`
[	Pokud je hodnota buňky nenulová, začni cyklus	`while(*p){`
]	Vrať se na začátek cyklu	`}`
%	Vygeneruj náhodnou hodnotu na pracovní pásce	`*p = rand();`
#	Konec programu	`return 0;`

Jazyk BF pracuje se třemi páskami: jedna uchovává agentovi akce, druhá obsahuje odměnu a nový stav, a třetí se nazývá pracovní a obsahuje vnitřní stavy a proměnné daného prostředí. Program se spustí, zapíše první stav a nulovou odměnu, dále počká, až agent zapíše na pásku svou akci. Hodnotu agentovi akce program načte a zpracuje, a poté zapíše na výstupní

pásku nový stav a odměnu. Odměna je vždy v první buňce výstupu a má hodnoty v rozsahu $< -100{,}100 >$. Z těchto hodnot i vychází maximální hodnota výsledného skóre, která může být až 100.

Následující symbol na výstupní pásce je pro hodnotu pozorování. Všechny symboly (na vstupní, pracovní a výstupní pásce) jsou celočíselné hodnoty, na které je aplikována operace modulo. To zaručí, že hodnoty nepřekročí stanovený rozsah Aby generované programy skončili, tak každý pokus o zapsání do nadbytečných buněk pro pozorování a odměnu slouží jako signál pro ukončení programu. Mezi symboly BF jazyka, byl ale přidán symbol `#`, který reprezentuje standardní konec programu, takže program lze ukončit i validně takto. Zároveň díky tomu lze podle něj při vzorkování programů jednotlivé programy od sebe jednoduše oddělit. Výsledkem je tak v této verzi AIQ přibližně 90 % programů, které nepřekročí výpočetní limit a skončí s výstupem v každé iteraci. Z náhodně vybraných programů byla odstraněna velká část nevhodného kódu jako `+-`, `><` nebo nekonečných cyklů `[]`. Také byly odstraněny všechny programy, které například nikdy nenačítaly vstup nebo nikdy nezapisovali na pásku. Na straně testu je do programů ještě přidáván na začátek jeden bit, který ovlivňuje, zda odměny mají obrácené znaménko. To umožňuje dvojí testování toho samého prostředí, což přináší přesnější výsledky.

V této podobě měl test problém s časovou náročnost na získání přesných výsledků. Standardní chyba, neboli přesnost střední hodnoty, pro vzorkovaná data klesala velice pomalu s rostoucí velikostí vzorku vybraných programů. Z tohoto důvodu byly přidány do testu techniky na snížení odchylek odhadů AIQ skóre.

První použitá metoda je paralelizace, která označuje rozdělení jednotlivých mezi výpočtů, které existují v rámci jednoho velkého výpočtu, do samostatných procesů, které mohou na víceprocesorových počítačích běžet souběžně a ušetřit tak čas (Midkiff, 2012). V rámci AIQ testu se využívá paralelizace pro samotné testování agentů na prostředích. To umožňuje využití plného potenciálu procesorů v počítači, tím že na každém jádru se spustí jeden test jednoho prostředí. Podle Legg a Veness (2011) při psaní původní práce toto dokázalo urychlit testování 10x.

Druhá použitá metoda je stratifikovaný výběr vzorků (Étoré a Jourdain, 2010). Tato metoda spočívá na myšlence, že se nejdříve prostor všech vzorků (programů) Ω rozdělí na vzájemně rozdílné množiny Ω_1,$\Omega_2, \cdots, \Omega_n$ tak, že sjednocením těchto množin získáme opět prostor Ω. Každé jednotlivé Ω_i se nazývá stratum. Při testování se poté hlídá, která strata mají výsledky s největším rozptylem. Taková strata mají největší vliv na výsledný rozptyl a jsou tak testovány více pro zpřesnění výsledků. Strata s nízkým rozptylem výsledků tak mohou být testovány méně a celý test je daleko efektivnější.

Třetí použitá metoda pro snížení rozptylu se nazývá společná náhodná čísla (Glasserman a Yao, 1992), a pomáhá snižovat rozptyl výsledků tak, že studované systémy, jejichž vlastnosti zjišťujeme, tak testujeme pokaždé s tou samou posloupností náhodných čísel. V prostředí AIQ testu funguje tak, že namísto toho, aby se dva agenti π, π' testovali na nezávislých vzorcích prostředí, tak je možné odhadnout rozdíl AIQ, když je použita stejná množina vzorků.

Společná náhodná čísla snižují šanci, že jeden z agentů bude testován na jednodušších vzorcích.

Poslední metoda, která slouží ke snížení rozptylu, se nazývá antitetické varianty. Zde přichází na řadu bit, přidaný k programům, který ovlivňuje, zda agent získává normální odměny nebo jestli dostává odměny s otočeným znaménkem. Metoda totiž spočívá v použití dvou vzorků, jejichž odhadované hodnoty jsou vzájemně opačné negativně korelované a samotné vzorky jsou vzájemně opačné, ke snížení rozptylu výsledků, které vznikly použitím původních vzorků (Hammersley a Morton, 1956). Pokud odhad normálního programu označíme jako $\hat{Y}_1$ a výsledek programu s opačnými odměnami jako $\hat{Y}_2$ celkový rozptyl daného programu se dá vypočítat jako:

$$Var(\hat{X}) = \frac{1}{4}[Var(\hat{Y_1}) + Var(\hat{Y_2}) + 2Cov(\hat{Y_1},\hat{Y_2})] \tag{1.5}$$

Tato metoda také chytře splňuje podmínku uvedenou z Kdykoliv přerušitelném testu (Hernández-Orallo a Dowe, 2010) ohledně vyvážených prostředích. Náhodné chování agenta při zprůměrování prostředí s normálními odměnami a poté s invertovanými odměnami povede na výslednou hodnotu blízkou nule.

1.6.2 Rozšíření testu algoritmické inteligence

Práce (Vadinský, 2018) dále navazuje na univerzální inteligenci (Hutter, 2007) a konkrétně na její praktickou implementaci jako algoritmické IQ (Legg a Veness, 2011) s cílem odhalit nedostatky původní verze testu a jejich odstranění. (Vadinský, 2018) se zaměřuje hlavně na důležitost univerzálního stroje v rámci AIQ, sémantickou a syntaktickou analýzou BF programů a jejich problémů, a v poslední řadě na výpočetní náročnost testu.

Bylo zjištěno, že aktuální programy BF se skládají z 74 % z programů, které obsahují zbytečný kód. To samozřejmě zesložiťuje interpretaci chování prostředí, ale také v návaznosti na to, že složitost prostředí je udávána délkou jeho programu, tak zbytečný kód přidává na složitosti jinak jednoduchým programům, a ovlivňuje tak celkové výsledky testu. Test byl tak rozšířen o metodu testující programy a jejich části pomocí sémantických a syntaktických tříd, včetně regulárních výrazů, pomocí kterých se zbytečný kód identifikuje a následně redukuje.

Vadinský (2018) dále objevil, že generované programy jsou z 17 % bez diskriminační síly. To znamená, že testování agentů na prostředích reprezentovaných těmito programy nijak nepřispívá k finálnímu skóre, protože nesplňují podmínku citlivosti vůči odměně (Hernández-Orallo a Dowe, 2010), což nejčastěji znamená, že odměny prostředí nijak nezávisí na akcích agenta, a jsou buď vždy stejné a nebo úplně náhodné. Tyto problematická prostředí se těžko odstraňují všechna, ale jejich počet lze redukovat opět syntaktickou a sémantickou klasifikací částí programů.

V rámci Vadinský (2018) byl test dále rozšířen o metody vícekolového zpřesňování odhadů AIQ skóre založené na Hernández-Orallo a Dowe (2010), dále o metody zvyšující výpočetní efektivitu testu, a v poslední řadě o metody pro vzorkování programů prostředí implementující

Hibbardův limit minimální délky programů (Hibbard, 2009) (což snižuje závislosti výsledků na zvoleném referenčním stroji).

Praktická verze AIQ testu byla do této doby implementovaná v programovacím jazyce Python ve verzi 2. Při navazujících pracích s AIQ testem se ukázalo jako problematická zpětná kompatibilita kódu a nemožnost použití nových Python knihoven, které nejsou s verzí Pythonu 2 použitelné. V rámci práce (Zeman, 2023) byl proto původní kód přepsán do Python verze 3, ve které funguje AIQ test i v čase psaní této práce.

2. Umělé inteligence a jejich učení

Tato kapitola poskytne úvod do oblasti umělých inteligencí a s nimi souvisejícími obory, které jsou relevantní pro tuto diplomovou práci. Konkrétně to jsou neuronové sítě, posilované učení, Q-učení a hluboké Q-učení. Tématem diplomové práce je totiž implementace agenta posilovaného učení využívající neuronové sítě, a všechny tyto koncepty spadají do oboru umělých inteligencí. Z toho důvodu bude v sekci 2.1 vysvětleno, co jsou to neuronové sítě, v čem přinášejí výhody a nevýhody, a jak se s nimi pracuje. Dále v sekci 2.2 bude vysvětlen koncept posilovaného učení, kdy agent se učí pomocí odměn, které získá během svých interakcí s prostředím. Na posilovaném učení staví AIQ test z předchozí kapitoly. V rámci posilovaného učení bude vysvětlena metoda Q-učení, které místo čistých odměn pracuje s užitkem, který rozšiřuje pojem odměny o očekávané odměny v budoucnu závislé na agentově chování. Z Q-učení později vychází moji implementovaní agenti. Tito agenti využívají metodu zvanou hluboké Q-učení, která spojuje neuronové sítě a zmíněné Q-učení a bude vysvětlena v sekci 2.3. Na závěr bude v rámci sekce 2.4 vysvětleno evoluční programování jako metoda „inteligentního“ prohledávání prostoru parametrů, kterými lze nastavit a ovlivnit učení a chování agenta.

Pojem umělá inteligence zahrnuje širokou oblast technologií, konceptů, matematiky a statistiky. Jak bylo představeno v předchozí kapitole, tak tento pojem nejdříve označoval stroje, které dokázali myslet viz (Turing, 1950). Samotný pojem „umělá inteligence“ vznikl v rámci projektu (McCarthy, 1955), který tak označoval různé aspekty učení a další jiné aspekty inteligence, které lze simulovat strojem. (Harris, 2023) označuje období, které následovalo po vzniku tohoto projektu jako „období mrazu a plodnosti“, kvůli tomu že se opakovala období, kdy došlo ke značným objevům a průlomům následovaných obdobími prázdna a ticha, bez žádných posunů. Důvod, proč existovala období „mrazu“ byl ten, že v oněch dobách se výzkum směřoval hlavně teoretickými rovinami, které narážely na problémy, které v dané době, s jejich omezenými technologickými prostředky, nešly vyřešit. Podle (Stone et al., 2016) je velký pokrok v tomto oboru v poslední dekádě dán hlavně tomu, jak jsou výkonné dnešní počítače, a díky dostupnosti ohromného množství dat.

Právě jedním z plodů „období hojnosti“ je hluboké Q-učení (Mnih, Kavukcuoglu, Silver, Graves et al., 2013), které představilo první úspěšné propojení neuronových sítí a posilovaného učení, a vznikl tak agent umělé inteligence, který dokázal plynule hrát videohry herní konzole Atari 6200. Tento agent v mnohých hrách značně překonal ve výsledném skóre lidské hráče. K pochopení, jak tento agent funguje, povede text následujících sekcí této kapitoly.

2.1 Neuronové sítě a hluboké učení

Hluboké učení jako pojem představuje metodiky a používání hlubokých neuronových sítí. Poprvé s tímto pojmem přišla práce Hinton et al. (2006), která ukázala, že je možné efektivně natrénovat a používat hluboké neuronové sítě k modelování složitých dat. Tato seknce představí neuronové sítě, jak se používají a jak se učí.

Jak zaznělo v úvodu této kapitoly, McCulloch a Pitts (1943) jako první představili a matematicky popsali fungování neuronů v mozku. Agostini (2017) popisuje funkci jednoho neuronu v mozku jako jednotku zpracovávající elektrické nebo chemické signály, které obdrží od jiných neuronů, a výsledný zpracovaný signál předává dál ke zpracování dalším neuronům. Informaticky by se dalo říct, že neuron získá vstupní data, zpracuje je, a předá dál. Důležitou vlastností neuronu podle Agostini (2017) je, že ne vždy výsledek předá dál. Někdy se stane, že vstupní signály nejsou důležité, a neuron tak na ně nereaguje. Každý neuron má totiž nastavený práh, který musí vstupní signály překonat.

2.1.1 Popis umělého neuronu

S těmito znalostmi by už nemělo být složité pochopit, jak umělý neuron funguje. Grossi a Buscema (2007) popisuje matematický předpis umělého neuronu následovně.:

$$y = f(\sum_{i=1}^{k} w_i x_i + b_i) \tag{2.1}$$

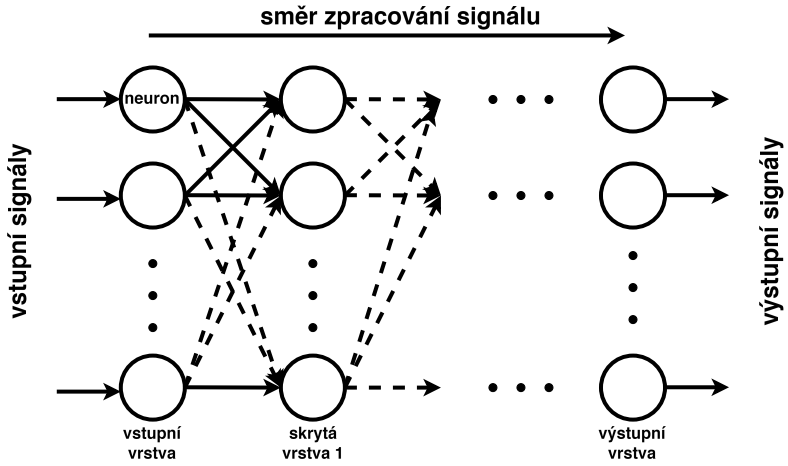
Výsledná hodnota y odpovídá hodnotě zpracovaného signálu, který neuron posílá dál. Tento signál je dán součtem všech vstupních signálů x_i, jejichž počet je k, a které jsou vynásobeny vahou w. Tato váha udává důležitost signálu z daného neuronu právě při výpočtu prahové hodnoty, kterou musí signály z předcházejících neuronů překonat. Některé signály mohou být ale velice jemné a malé, ale přesto důležité. Od toho je součástí vzorce ještě prahová hodnota b, označovaná jako bias, která je přičítána nebo odčítána od aktuálního signálu. Práh, který musí tato suma překročit je dána tzv. aktivační (prahovou) funkcí f.

Všimněte si, že kdybychom měli pouze jeden neuron s jedním vstupem, tak nám uvnitř aktivační funkce zbude předpis odpovídající lineární funkci $wx + b$. Jeden samotný neuron, tak dokáže reprezentovat lineární funkci neboli přímku.

2.1.2 Popis neuronové sítě

Pokud zvýšíme počet vstupů, neuron bude reprezentovat rovinu v prostoru. Síla neuronů ale spočívá v jejich propojení do neuronové sítě, jejíž diagram lze vidět na Obrázku 2.1, která poté dokáže reprezentovat aproximaci libovolné spojité funkce (Elbrächter et al., 2021).

Neuronová síť, která je složená z takovýchto neuronů, se nazývá perceptron (Rosenblatt, 1958), pokud má jednu vrstvu, nebo vícevrstvý perceptron, pokud má vrstev více. Uspořádání



Obrázek 2.1: Diagram struktury neuronů v neuronové síti

neuronů v těchto sítích se označuje jako dopředná neuronová síť (Grossi a Buscema, 2007), a to z toho důvodu, že neurony z jedné vrstvy poskytují signál pouze neuronům z vrstvy následující, viz schéma na Obrázku 2.1. Každý neuron z jedné vrstvy je spojen s každým neuronem z vrstvy následující. Signály tak putují od samotného vstupu na neurony vstupní vrstvy, z nich poté na neurony takzvaných skrytých vrstev až konečně na neurony výstupní vrstvy. Hloubka neuronové sítě udává z kolika skrytých vrstev se neuronová síť skládá.

Neuronová síť tedy funguje tak, že je jí předán vstupní vektor, jehož hodnoty jsou přes vážený součet zpracovány aktivačními funkcemi neuronů od vstupní vrstvy, až po výstupní vrstvu, jejíž výstup je opět vektor hodnot. Tento proces se nazývá dopředný průchod. Pro učení neuronové sítě je klíčové mít ke každému takovému vstupnímu vektoru cílové hodnoty, které určují, jaký by měl být správný výstup z výstupní vrstvy po provedení dopředného průchodu. Díky tomu dokážeme vyjádřit chybovou funkci, kterou Rumelhart et al. (1986) popisují takto:

$$E(x,\theta) = MSE = \frac{1}{n}\sum_{i=1}^{n}(t_i - f_{\theta,i}(x))^2 \tag{2.2}$$

Chybová funkce E pro vstupní vektor x s vahami sítě θ je dána jako průměr kvadratické odchylky mezi výstupním vektorem sítě $f_\theta(x)$ a cílovou hodnotou v podobě vektoru t, kde i označuje složky těchto vektorů. Tato chybová funkce se označuje jako Mean squared error (střední kvadratická chyba), a vyjadřuje o kolik se v průměru liší výstup od očekávaného výstupu. Učení neuronové sítě spočívá v minimalizaci této chyby pomocí změn matice vah neuronových spojení θ. Minimalizace se provádí pomocí gradientu této chybové funkce, nicméně kvůli složitosti neuronových sítí tento gradient nelze vypočítat analyticky, ale využívá se k tomu iterativní přístup zvaný jako gradientní sestup se zpětnou propagací chyby (Rumelhart et al., 1986).

Gradientní sestup pracuje na principu nalezení gradientu chybové funkce pro jednotlivé váhy $w \in \theta$ a poté provedení malé změny hodnot vah v opačném směru k nalezenému gradientu. Tento proces gradientního sestupu je opakován, dokud se funkce nenalezne v lokálním minimu. Kvůli povaze neuronových sítí a jejich složité struktuře vedoucí na spoustu složených funkcí nelze provést gradientní sestup takto jednoduše, ale je zapotřebí použít metodu zpětné

propagace. Zpětná propagace nám poskytuje postup, jak pomocí řetízkového pravidla dostat hodnotu z chybové funkce výstupních neuronů postupně přes všechny skryté vrstvy až na vrstvu vstupní a umožní tak provést gradientní sestup na každou váhu v síti.

2.1.3 Výhody a nevýhody neuronových sítí

Dopředné neuronové sítě tvoří podle Mahanta (2017) mocný nástroj. Jako hlavní výhody použití těchto neuronových sítí uvádí:

- schopnost naučit se a modelovat nelineární a složité vztahy – To je důležité, protože v reálném světě je většina vztahů mezi věcmi nelineární a složitá.
- vysoká schopnost zobecňovat – Po naučení se z prvotních dat a jejich vztahů dokáží odvodit nové neznámé vztahy, které dokáží využít při zpracování nových neznámých dat
- minimální omezení na data – Neuronové sítě si dokáží poradit s libovolnými daty, má-li jich dostatek k učení. Dokáže dokonce modelovat heteroskedasticitu, tedy data s vysokou volatilitou a proměnlivým rozptylem, díky právě schopnosti odhalit nové vztahy, aniž by byly předem známy.

Mijwil (2018) ve svém článku popisuje výhody a nevýhody používání hlubokého učení. Důležité je ale rozebrat i nevýhody hlubokého učení a neuronových sítí. Jako hlavní uvádí následující:

- výpočetní náročnost – K trénování neuronové sítě je potřeba výkonného hardwaru, ať už v podobě procesorů podporující paralelní výpočty, nebo v podobě grafických procesorů či procesorů pro práci s tensory (TPU).
- nevysvětlitelnost chování neuronové sítě – Kvůli složité povaze architektury sítě a kvůli tomu, že nelze odhadnout co má jaký neuron naučeno za vnitřní interpretaci, nelze určit jakým způsobem došla neuronová síť k finálnímu výsledku.
- neexistující pravidla pro návrh neuronové sítě – Počet vrstev a jejich šířky, stejně tak učící parametry nelze dopředu nijak odhadnout a je tak potřeba nalézt tyto hodnoty za pomocí experimentování.

Důležitému problému neuronových sítí se věnoval ve své práci Dr. Marcus (Marcus, 2018), kde se píše:

> „Aby byly neuronové sítě schopné správně zobecňovat, musí se obecně použít velké množství dat. Testovací data musí být podobná trénovacím, aby bylo možném získat nové odpovědi a interpolovat je mezi staré. Například pro neuronovou síť s devíti konvolučními vrstvami o 60 milionech parametrech a 650 tisíci neurony se muselo k trénování použít zhruba 1 milion trénovacích dat.“,

a později:

> „Systémy hlubokého učení fungují hůře pokud je k dispozici limitovaný počet

> trénovacích dat, nebo když se testovací množina významně liší od trénovací množiny, nebo pokud je prostor příkladů moc široký a plný různých nových informací.“,

Je tedy důležité vzít v potaz, že neuronové sítě jsou náročné na data, kterých je potřeba nejméně v řádů vyšších tisíců až milionů, a tato trénovací data by měla obsahovat všechny informace, se kterými by si měla neuronová síť umět poradit (tedy i například šum, který může vzniknout při používání modelu v reálném světě mimo speciálně připravených trénovacích dat). Počet trénovacích dat by zároveň měl být i úměrně velký k rozměru jednotlivých dat, a tedy počtu unikátních zpracovávaných informací.

2.2 Posilované učení

Jak již zaznělo v Kapitole 1.4 o Univerzální inteligenci a později v Kapitole 1.6 o Algoritmickém inteligenčním kvocientu, tak tyto testy využívají koncept posilovaného učení. Ten je ve světě umělých inteligencí specifický svým přístupem, jakým stanovuje problémy a učící proces. Obecně se totiž setkáme spíše s takzvaným učením s učitelem, ze kterého intuitivně vychází proces učení neuronových sítí z předchozí podkapitoly. Bishop (2006) říká, že učení s učitelem je charakteristické tím, že pro všechna učící data známe i očekávaný správný výsledek. Modely učíme tak, že jim tato data předkládáme a měníme jejich vnitřní strukturu a parametry, dokud není rozdíl mezi výsledky a očekávanými výsledky dostatečně malý.

Koncept posilovaného učení se od učení s učitelem velice liší. Sutton a Barto (2018) označují posilované učení jako učení se ze vzájemných interakcí. V posilovaném učení neexistuje nic jako trénovací data, ale je představen problém odpovídající prostředí, které poskytuje informace o jeho aktuální podobě a je na agentovi (člověku, modelu, umělé inteligenci), aby provedl nějakou akci. Provedením akce se změní aktuální stav prostředí, a agent zároveň obdrží odměnu. Vyřešení problému odpovídá nejčastěji získání maximální odměny, nicméně agent může obdržet i zápornou odměnu (trest), která signalizuje, že se nechová správně. Agent se tak působením v prostředí musí z těchto odměn naučit, co má, v jaké situaci dělat, jakou akci má v daném stavu provést, aby získal nebo se přiblížil maximální odměně. Samotný agent se učí postupně metodou pokus-omyl. Jednotlivá prostředí mohou být libovolně složité a podle François-Lavet et al. (2018) přináší posilované učení tu výhodu, že agent vůbec nemusí dané prostředí vůbec znát nebo mu rozumět. Stačí mu pouze s prostředím interagovat a sbírat z této interakce informace a učit se z nich.

Posilované učení s sebou nese, ale i řadu nevýhod. Tyto nevýhody popisuje (Arulkumaran et al., 2017) následovně:

- Optimální chování agenta se hledá metodou pokus-omyl interakcemi agenta s prostředním pouze za pomocí odměn a trestů. Nelze nalézt optimální chování dopředu.
- Jednotlivá pozorování agenta závisí pouze na jeho vlastních akcích a jsou tak silně časově korelované.
- Agent se musí vyrovnávat s časovými závislostmi na dlouhé časové vzdálenosti, protože agentovi akce mohou přinést odměnu až za velký počet časových kroků. Tento problém se označuje jako sparse reward problem.

Na další problém, se kterým se musí potýkat agenti posilovaného učení, upozorňuje (Otterlo a Wiering, 2012), kteří ho nazývají jako Exploration-Exploitation Trade-off. Problém spočívá v tom, že agent zpočátku většinou nezná prostředí, ve kterém se nachází, a musí ho nejdříve prozkoumat, tím že zkouší různé akce a zjišťuje jejich dopady na prostředí. Agent dostává zpětnou vazbu pouze v podobě odměn, ale neví, která akce by mu přinesla největší odměnu, dokud ji sám nezkusí. Za nějaký čas se agent naučí politiku, která přináší nějaké dobré výsledky, ale stejně je potřeba sem tam vyzkoušet nějaké nové chování, aby agent mohl zjistit, co je možné ještě zlepšit.

Toto zkoušení neznámého sice často přinese horší výsledky, protože zvolená akce neodpovídá naučené politice, ale pokud by agent nové akce nezkoušel, nikdy by nezjistil, co může ve své politice zlepšit. Zároveň se prostředí může časem měnit a agent potřebuje prostředí neustále prozkoumávat, aby věděl, jaká politika je pro něj v daný moment nejlepší. Aby se agent učil, tak musí prozkoumávat. Aby ale měl dobré výsledky, tak musí využívat, co se naučil.

Tato sekce dále popíše detailněji koncept posilovaného učení a jeho součásti. Představí také jak se intuitivně dají řešit problémy posilovaného učení. Nadstavbou tohoto intuitivního řešení je poté koncept Q-učení, který už je konkrétním způsobem, jak se dají reálné problémy posilovaného učení řešit, a který bude také vysvětlen. Q-učení lze dále zlepšit pomocí metody Eligibility traces, jejíž vysvětlení zakončí tuto sekci.

2.2.1 Formální popis posilovaného učení

K práci s posilovaným učením je nutné nejen pochopit princip, ale hlavně jak tento koncept funguje prakticky: co je to prostředí, jakým způsobem informuje agenta o svém stavu, jak fungují odměny, a jak podle nich učit agenta. To vše vysvětlí tato podkapitola.

François-Lavet et al. (2018) popisuje obecnou úlohu posilovaného učení jako stochasticky řízený proces v diskrétním čase, kde agent ovlivňuje prostředí v následující posloupnosti.:

1. Agent začíná v nějakém stavu prostředí $s_0 \in S$.
2. V každém časovém kroku t agent musí vykonat akci $a_t \in A$.
3. Akce vyvolá změnu stavu prostředí $s_t \rightarrow s_{t+1}$, načež agent obdrží nové pozorování o stavu s_{t+1} a odměnu $r_t \in r$.
4. Kroky 2. a 3. se opakují, dokud stav s_{t+1} není tzv. terminální stav (agent se dostal do cíle) nebo hodnota t nepřekročí maximální hodnotu, kdy je pokus ukončen.

Arulkumaran et al. (2017) využívá k popisu prostředí a dále posilovaného učení konceptu Markovského rozhodovacího procesu, který popisuje prostředí následujícím způsobem. Prostředí se skládá z:

1. množiny stavů S a distribuce počátečních stavů $p(s_0)$, která udává pro každý stav pravděpodobnost, že agent v něm bude začínat. Jednotlivé stavy mají pro agenta podobu vektoru skládající se z hodnot odpovídají příznakům reprezentující daný stav prostředí,
2. množiny možných akcí A,
3. přechodové funkce $T : S \times A \times S \rightarrow [0,1]$, která přiřazuje pravděpodobnost, se kterou vykonání konkrétní akce $a_t \in A$ v konkrétním stavu $s_t \in S$, dostane prostředí do konkrétního stavu $s_{t+1} \in S$,
4. funkce odměn $r : S \rightarrow \mathbb{R}$, která přiřazuje odměnu za akci a_t ve stavu s_t, která vedla do stavu s_{t+1},
5. a diskontního koeficientu budoucí odměny $\gamma \in [0,1)$. Čím je hodnota γ menší, tím větší důraz je kladen na okamžitou odměnu získanou agentem. Naopak vyšší hodnota vede

agenta více k zohledňování odměn, které lze získat v budoucnu, a tím stanovuje jeho dlouhodobé cíle.

Z definice si lze všimnout, že přechodová funkce nemusí být deterministická, a tedy opakováním akce a ve stavu s, se nemusíme vždy dostat do stejného stavu s_{t+1}. To, protože přechodová funkce udává pravděpodobnost, se kterou následující stav nastane a ta nemusí mít nutně hodnotu 1 pro každý přechod mezi stavy.

Zajímavá je také funkce odměn. Z jejího zápisu si lze všimnout, že funkce odměn vrací reálná čísla. Odměna tudíž nemusí být pro agenta jen něco pozitivního, ale lze za určitou akci v konkrétním stavu uložit agentovi i trest v podobě záporné odměny.

Dále v posilovaném učení definujeme politiku π. Ta podle Sutton a Barto (2018) popisuje, jakým způsobem agent vybírá jednotlivé akce. Formálně se jedná o funkci, která pro každý stav vrací pravděpodobnost výběru jednotlivých akcí: $\pi : S \rightarrow p(A = a|S)$.

Pokud je prostředí epizodické, a tedy stav se vrátí do původního po každé epizodě s délkou t_{max}, tak posloupnost stavů, akcí a odměn se nazývá trajektorie politiky. Každá trajektorie jedné politiky sbírá (kumuluje) odměny z prostředí, čímž se získá kumulativní odměna $R = \sum_{t=0}^{t_{max}} r_{t+1}$. Podle Arulkumaran et al. (2017) je cílem posilovaného učení nalézt optimální politiku π^*, která získá nejvyšší kumulativní odměnu neboli ji maximalizuje pro všechny stavy:

$$\pi^* = \underset{\pi}{argmax} E[r|\pi] \tag{2.3}$$

Sutton a Barto (2018) uvádí, že populárním způsobem, jak naučit agenta maximalizovat kumulativní odměnu je práce s hodnotovou funkcí V, která každému stavu přiřazuje očekávanou odměnu, kterou lze v tomto stavu získat. Hodnotová funkce je daná vzorcem:

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=0}^{t_{max}} \gamma^k r_{t+k} | s_t = s\right\}, \tag{2.4}$$

kde hodnota stavu s při vybírání akcí podle politiky π je rovna součtu odměn ze stavů dosažitelných z původního stavu s až po maximální dobu t_{max}. Každá odměna získatelná v budoucím časovém kroku k je diskontovaná diskontním koeficientu budoucí odměny γ, jehož hodnota klesá vlivem zvyšujícího se exponentu v podobě k. Parametr γ tedy udává, jak moc velký důraz dáváme odměnám v budoucnosti.

Hodnotová funkce má rekurzivní vlastnosti, a proto ji lze přepsat do rekurzivní podoby nazvané jako Bellmanova rovnice (Bellman a Dreyfus, 2010):

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s\} \\ &= E_{\pi}\{r_t + \gamma V^{\pi}(s_{t+1}) | s_t = s\} \\ &= \sum_{s'} T(s,\pi(s),s')(r(s,a,s') + \gamma V^{\pi}(s')) \end{aligned} \tag{2.5}$$

Jedná se tedy o součet odměn r získatelných ve stavech s' podle pravděpodobnosti, že se z původního stavu s dostaneme do stavů s' s použitím politiky π podle přechodové funkce T. Ke

každé získatelné odměně ve stavu s' je přičtena rekurzivně diskontovaná hodnota hodnotové funkce stavu s', která odpovídá opět součtu odměn získatelných ze stavů, na které se lze dostat ze stavu s' atd.

Optimální politika π^* se nemusí definovat pouze vůči kumulativní odměně, ale může se definovat i vůči hodnotové funkci. Pro optimální politiku platí, že $V^{\pi^*}(s) > V^{\pi}(s)$, tedy že optimální politika π^* získá nejvyšší hodnotu pro všechny stavy $s \in S$ ze všech možných politik π. Sutton a Barto (2018) ukazuje, že pro takovou optimální politiku platí:

$$V^{\pi^*}(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s')(r(s,a,s') + \gamma V^*(s')), \tag{2.6}$$

tedy že optimální politika π^* volí takové akce, které přinesou nejvyšší odměnu s ohledem na hodnotu hodnotové funkce v budoucnu, a tedy:

$$V^{\pi^*}(s) = \operatorname*{argmax}_{a \in A} \sum_{s' \in S} T(s,a,s')(r(s,a,s') + \gamma V^*(s')), \tag{2.7}$$

Otterlo a Wiering (2012) uvádí, že politika, která vybírá akce podle tohoto pravidla, se nazývá chamtivá politika (angl. greedy policy), protože vybírá „chamtivě“ tu nejlepší akci, kterou má k dispozici, podle hodnotové funkce. Analogicky k problému Exploration-Exploitation Trade-off, z úvodu k posilovanému učení, zavádí -greedy politiku, která s pravděpodobností ϵ vybírá náhodné akce, a tudíž prozkoumává prostředí. S pravděpodobností $1 - \epsilon$ poté vybírá akce podle naučené politiky.

2.2.2 Q-učení

Objevování metod pro hledání optimální politiky je hlavním úkolem výzkumů v oblasti posilovaného učení. Nejúspěšnější metodou z počátku posilovaného učení je metoda Q-learning (Watkins a Dayan, 1992). Tato poměrně jednoduchá, zato ale robustní metoda, dokáže řešit spoustu úkolů posilovaného učení a je použita jako základ v mnoha pokročilejších přístupech jako například (Mnih, Kavukcuoglu, Silver, Graves et al., 2013).

Hledání politiky podle hodnotové funkce, popsanou Bellmanovou rovnicí v předchozí kapitole, je sice intuitivní, nicméně v praktickém využití má ale jeden problém. Je potřeba dopředu znát okamžitou odměnu všech možných akcí v daném stavu (tedy je potřeba znát funkci odměn, přechodovou funkci, a obecně celé prostředí).Watkins a Dayan (1992) místo hodnotové funkce používají tzv. funkci kvality neboli Q-funkci. Ta místo toho, aby na vstupu měla pouze aktuální stav, jako hodnotová funkce, tak pracuje jak s aktuálním stavem, tak i s jednou danou akcí. Udává nám tak kvalitu nebo očekávání jakou hodnotu přinese zvolená akce a ve stavu s. Q-funkce má nesledující předpis:

$$Q^{\pi^*}(s,a) = r(s,a) + \gamma \sum_{s' \in S} \max_{a' \in A} T(s,\pi(s'),s')\, Q^*(s',a')). \tag{2.8}$$

Jedná se o okamžitou odměnu za vykonání akce a ve stavu s a sumu diskontovaných Q-hodnot, které přinese vybírání akcí podle politiky π nadále. Hodnotová funkce $V^*(s)$ byla nahrazena

za hodnotu stavu v podobě $\underset{a}{max}Q^*(s_t,a)$. Užitek podle optimální politiky pro stav s_t je stejný jako nejvyšší hodnota Q funkce podle stejné politiky pro možné akce. Optimální politika poté vybírá akci, která má nejvyšší Q-hodnotu:

$$\pi^*(s) = \underset{a}{argmax}\, Q^*(s,a) \tag{2.9}$$

Odtud plyne výhoda použití Q-funkce, pokud se jí totiž agent dokáže naučit, tak dokáže velice jednoduše rozhodnout jaká je jeho optimální akce díky porovnání hodnot naučené Q-funkce mezi množnými akcemi v daném stavu, což u hodnotové funkce pracující hlavně s funkcí odměn není možné.

Watkins a Dayan (1992) popisují chování agenta využívající Q-učení v každém kroku t následujícím způsobem:

1. Získá se aktuální stav s_t.
2. Vybere se a provede se akce a_t.
3. Získá se pozorování nového stavu s_{t+1} a obdrží se odměna r_t.
4. Agent upraví předchozí hodnoty Q-funkce Q_{t-1} úměrně k učícímu parametru α, následujícím způsobem:

$$Q_t(s_t,a_t) = (1-\alpha)Q_{t-1}(s_t,a_t) + \alpha(r_t + \gamma \underset{a'}{max}\, Q_{t-1}(s_{t+1},a')) \tag{2.10}$$

5. Pro všechny ostatní kombinace hodnot stavů a akcí $X = \{(s \in S, a \in A)\}/\{(s_t, a_t)\}$ nastav hodnotu na hodnotu z předchozí epizody $Q_t(X) = Q_{t-1}(X)$

Tento způsob použití Q funkce vyžaduje použití vyhledávací tabulky, která bude uchovávat hodnoty Q-funkce pro všechny relevantní dvojice s,a. Tímto iterativním přístupem hodnoty Q-funkce zkonvergují, a bude tak nalezena optimální politika. Důkaz konvergence lze najít v Watkins a Dayan (1992).

2.2.3 Eligibility Traces

Samotné Q-učení je poměrně jednoduchá metoda a některé problémy nemusí stačit. Její možnou nadstavbou jsou takzvané eligibility traces (český překlad by mohl být stopy způsobilosti). Sutton a Barto (2018) popisuje eligiblity traces následujícím způsobem:

> „Jedná se o dočasný záznam o výskytu události jako navštívení stavu prostředí nebo provedení akce. Stopa označuje ty parametry paměti, které jsou spojené s aktuální událostí, jako způsobilé pro probíhající změny učením. Při úpravách hodnoty Q-funkce je připisována chyba nebo správnost předchozího výsledku pouze těmto způsobilým stavům a akcím. Díky tomu eligibility traces pomáhají propojit předchozí události s aktuální učící informací.“

Jednodušeji řečeno, eligibility traces poskytují způsob, jak upravovat Q-hodnotu i pro akce a stavy, které předcházely dané odměně nebo trestu v aktuálním stavu. Q-učení bez eligiblity traces upravuje Q-hodnotu pouze pro aktuální stav a zvolenou akci.

Eligibility traces mají podobu matice el mapující všechny kombinace stavů a akcí. Tato matice je v úvodu nulová. Její hodnoty jsou ale upravovány během agentových interakcí s prostředím po provedení konkrétní akce a ve stavu s. Vzorec pro úpravu hodnot eligiblity traces podle Sutton a Barto (2018) vypadá následovně.:

$$el_t(s,a) = \begin{cases} \gamma\lambda el_{t-1}(s,a) + 1, & s = s_t, a = a_t \\ \gamma\lambda el_{t-1}(s,a), & \text{jinak} \end{cases} \tag{2.11}$$

Kde el_t je hodnota eligibility traces pro aktuální dvojici stav a provedená akce. λ je parametr zvaný trace decay a má danou hodnotu ($0 \leq \lambda \leq 1$). γ je standardní diskontní koeficient odměny, a nakonec s_t,a_t jsou stav a akce v aktuálním okamžiku.

Sutton a Barto (2018) popisuje postup použití eligibility traces s Q-učením v každé interakci s prostředím následujícím způsobem:

1. Zvolí se akce a podle aktuální politiky π pro aktuální stav s.
2. Získá se odměna r a nový stav $s\prime$.
3. Upraví se hodnota Q-funkce pro předchozí stav a provedenou akci podle vzorce:
$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha\delta el_t(s,a). \tag{2.12}$$
4. Přičte se hodnota 1 k eligibility trace pro předchozí stav a provedenou akci.
5. Pro všechny kombinace stavů a akcí mimo předchozí stav a akci:
   - se upraví Q-hodnota podle $Q_t(s,a) = Q_{t-1}(s,a) + \alpha\delta el_t(s,a)$,
   - a upraví se hodnota eligibility traces podle $el_t(s,a) = \gamma\lambda el_{t-1}(s,a)$
6. Postup se vrátí na bod 1 a celý proces se opakuje.

Hodnota δ v krocích 3 a 5a má následující hodnotu:

$$\delta = r + \gamma \max_{a' \in A} Q(s', a') - Q(s,a). \tag{2.13}$$

Výsledná hodnota $\alpha\delta$ tak odpovídá standardnímu pravidlu pro úpravu Q-hodnoty ze podsekce 2.2.2 o Q-učení. Je dobré si uvědomit, že pokud bude hodnota $\lambda = 0$, tak celý proces Q-učení bude fungovat jako bez použití eligibility traces, protože jejich hodnota bude vždy nulová mimo aktuální stav, který bude mít hodnotu 1, tedy ve finálním vzorci pro úpravu Q-hodnoty nebude mít žádný vliv.

Výše zmíněná metoda eligibility traces se nazývá akumulační, protože eligibility hodnoty s každou návštěvou stavu a provedení akce zvyšují svou hodnotu o 1. V některých případech se ale mohou zlepšit výsledky agenta, pokud se použije druhá metoda označovaná podle Sutton a Barto (2018) jako nahrazovací metoda anebo dutch metoda.

Pokud je totiž podruhé navštíven ten samý stav a provedena ta samé akce, tak akumulační metoda zvedne hodnotu eligibility pro daný stav a akci nad hodnotu 1. Nahrazovací metoda

oproti tomu nastaví hodnotu pro daný stav a akci na hodnotu 1 pokaždé namísto přičítání, což zvláště v situacích, kdy agent pracuje s velmi dynamickým prostředím, vede na lepší výsledky.

2.3 Hluboké Q-učení

Tato kapitola propojuje poznatky ze sekce 2.1 o neuronových sítích a sekce 2.2.2 o Q-učení. Hlavním zaměřením této diplomové práce je totiž integrování těchto znalostí do implementace agenta využívajícího hluboké Q-učení.

Metoda hlubokého Q-učení (deep Q-learning, zkráceně DQL), která je popsána v práci Mnih, Kavukcuoglu, Silver, Graves et al. (2013), spojuje principy hlubokého učení s řešením úloh posilovaného učení pomocí Q-učení. Samotné Q-učení má určitá omezení, na která narazíme při jeho použití v praxi. Problém vzniká zejména v případě, kdy je stavový vektor prostředí příliš rozsáhlý, nebo když prostředí umožňuje velké množství možných akcí. V tabulkovém Q-učení každý možný stav a každá možná akce tvoří jedinečnou položku v matici. Velikost této matice roste exponenciálně s rostoucím prostorem akcí a stavů, což představuje potenciální problém. Agent zároveň nemá informace o hodnotách Q-funkce pro stavy a akce, které ještě nenavštívil.

Jak ale zaznělo v sekci refsec:neuronky, tak neuronové sítě poskytují řadu výhod jako je schopnost zobecňovat problémy a zpracovávat složitá data. Podle Mnih, Kavukcuoglu, Silver, Graves et al. (2013) lze vyřešit právě zmíněné problémy Q-učení, pokud aproximujeme tabulkovou Q-funkci neuronovou sítí, a to právě díky schopnosti zobecňovat prostor stavů a akcí.

Mnih, Kavukcuoglu, Silver, Graves et al. (2013) zavádějí dopřednou neuronovou síť, která aproximuje právě Q-funkci. Tato neuronová síť je učená klasickou metodou stochastického gradientního sestupu. Z kapitoly o hlubokém učení ale víme, že k učení touto metodou jsou potřeba vstupní data a cílové výstupní hodnoty proti kterým se síť učí. Autoři ke sbírání těchto učících vzorků používají mechanismus zvaný replay memory. Replay memory je datová struktura, která během života agenta uchovává všechny jeho interakce s daným prostředí. Tyto interakce mají následující tvar: $e_t = (s_t,a_t,r_t,s_{t+1})$, kde s_t představuje pozorování stavu prostředí v čase t; a_t představuje akci, kterou agent provedl v čase t; r_t je poté odměna, kterou získal agent v čase t za provedenou akci a_t ve stavu s_t; a finálně s_{t+1} je nové pozorování stavu, do kterého se agent dostal provedení akce a_t ve stavu s_t. Replay memory zároveň musí umožňovat náhodný výběr vzorku z těchto dat.

Použití replay memory odstraňuje spoustu problémů vznikajících z kombinace posilovaného učení s hlubokým učením. Jako hlavní Mnih, Kavukcuoglu, Silver, Graves et al. (2013) uvádí následující:

1. Každý vzorek může být použit vícekrát, což umožňuje efektivnější využití dat, kterých hluboké učení vyžaduje spoustu.
2. Sekvenčně sbíraná data jsou velmi často mezi sebou hodně korelovaná. Náhodný výběr z nich odstraňuje tuto přímou korelaci a snižuje tak odchylky mezi úpravami vah sítě napříč jednotlivými iteracemi.
3. Učením agenta se mění jeho politika výběru akcí, čímž se zásadně mění i distribuce

dat získaných při sekvenčním učení. To často vede k uvíznutí v lokálním minimu nebo dokonce k divergenci učení. Takto má ale agent k učení stále k dispozici původní data, i pokud se zásadně změní jeho politika. To pomáhá proti uvíznutí a možným oscilacím, čí divergencím učící funkce.

Za pozornost stojí, jak Mnih, Kavukcuoglu, Silver, Graves et al. (2013) namodelovali neuronovou síť, která má za úkol aproximovat Q-funkci. Q-funkce má totiž tvar $Q(s,a)$, a je tudíž závislá na stavovém vektoru a dané akci. Naivním řešením by tedy bylo namodelovat neuronovou síť jako síť, která má počet vstupů podle počtu dimenzí vektoru pozorování + například one-hot encoding akcí. Jako výstup bychom poté učili neuronovou síť získat Q hodnotu, která by odpovídala upravené Bellmanově rovnici 2.8 ze sekce 2.2.2. Autoři ale tento přístup oprávněně kritizují. Při hledání nejvyšší Q-hodnoty by se musel provádět dopředný průchod sítí pro každou možnou akci. To by vedlo k lineárně rostoucí složitosti metody s rostoucím počtem možných akcí. Místo toho byla použita architektura, kdy jako vstup je použit vektor pozorování stavu s_t, a výstupní vrstva obsahuje tolik neuronů, kolik je v prostředí možných akcí. Síť se tak učí pro každé pozorování (reprezentované vektorem s_t) nalézt Q hodnoty všech akcí v jediném dopředném průchodu sítí.

Autoři ve svém článku prezentují postup, jak funguje DQL algoritmus. Problém pro mojí práci je, že jejich algoritmus není obecný a je napsány přímo pro problém, kterému se v práci věnují, a to naučit DQL agenta hrát hry na simulátoru herní konzole Atari (Bellemare, Naddaf et al., 2013). Původní algoritmus, tak zahrnuje i zpracování obrazového vstupu pomocí konvolučních síti, které teprve tvoří vektor pozorování s_t. Můj obecnější postup vychází z Mnih, Kavukcuoglu, Silver, Graves et al. (2013), ale oprošťuje se od zpracování obrazu pro tvorbu s_t a s_{t+1}. Tuto verzi zachycuje Kód 2.1.

Kód 2.1: Algoritmus Deep Q-learning s použitím Experience Replay

1. vytvoř prázdnou replay memory R s maximální kapacitou N
2. vytvoř neuronovou síť $\hat{Q}$ aproximující Q-funkci s náhodnými vahami
3. pro epizodu od 1 do M opakuj
4. pro t od 1 do t_{max} opakuj
5. s pravděpodobností ε vyber náhodnou akci a_t
6. jinak vyber akci $a_t = \max_{a \in A} \hat{Q}(s_t,a,\theta)$
7. proveď akci a_t v prostředí a získej nový stav s_{t+1} a odměnu r_t
8. ulož zkušenost $e_t = (s_t,a_t,r_t,s_{t+1})$ do R
9. získej J náhodných vzorků pozorování (s_t,a_t,r_t,s_{t+1}) z R
10. Vytvoř vektor y o velikosti J s prvky:

$$y_j = \begin{cases} r_j, & \text{pro terminální stav } s_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1},a';\theta), & \text{pro neterminální stav } s_{j+1} \end{cases}$$

11. Proveď krok gradient descent metody na chybovou funkci:

$$\left(y_j - \hat{Q}(s_j,a_j;\theta)\right)^2$$

V prvních krocích je potřeba vytvořit a inicializovat neuronovou síť a replay memory. Dále probíhá cyklus přes počet epizod. Jedna epizoda končí, pokud agent dorazí do terminálního stavu, který ukončí prostředí nebo počet jeho interakcí přesáhne hodnotu t_{max}. Poté následuje výběr akce. Agent k tomu používá ε-greedy politiku, aby bylo zachováno prozkoumávání prostředí. Pokud akce není zrovna zvolena náhodně, tak je vybrána pomocí neuronové sítě. Té je předložen aktuální stav, a vrátí nám naučené Q-hodnoty akcí. Agent vybere tu s nejvyšší hodnotou a předá ji prostředí, které vrátí nový stav a odměnu.

Aktuální stav, provedená akce, odměna a nový stav jsou uloženy do replay memory k učení. Poté nastává učící sekce. Z replay memory se vybere J náhodných záznamů, z jejichž odměn, akcí a následujících stavů jsou vypočítány pomocí Bellmanovy rovnice cílové hodnoty y_j pro neuronovou síť. Všimněte si, že Q-hodnotu zde tvoří opět neuronová síť $\hat{Q}$. S aktuálními stavy se provede druhý dopředný průchod neuronovou sítí a vyberou se Q-hodnoty akcí, které byly získány z replay memory společně s těmito aktuálními stavy. Dále je vypočtena chybová funkce rozdílu dopředného průchodu a cílových hodnot y_j, a na tuto chybu je proveden gradientní sestup, čímž se neuronová síť učí.

Je dobré si všimnout, že se neuronová síť učí vlastně proti svému vlastnímu odhadu. Tento odhad ale navíc obsahuje reálně obdrženou odměnu z prostředí. Není-li pak hodnota discount faktoru $\gamma = 1$, hodnoty $\hat{Q}$ funkce by měly nakonec konvergovat k optimu (Mnih, Kavukcuoglu, Silver, Graves et al., 2013).

Tento původní postup, ale není bez chyby, k tomu se autoři vyjadřují ve své navazující práci Mnih, Kavukcuoglu, Silver, Rusu et al. (2015). Hlavním problémem, který je v této práci zmíněn je stabilita procesu učení. Autoři přicházejí s použitím druhé neuronové sítě $\hat{Q}_{tar}$ s parametry $\hat{\theta^{-}}$ a je nazývaná jako target network. Tato síť je zpočátku vytvořena se stejnými parametry jako původní síť (dále jako policy network). Každých n-kroků jsou váhy neuronů z policy network překopírovány do target network. Target network ale po dobu těchto n-kroků představuje stabilní bod při učení původní policy network.

Bez target network se totiž cílové hodnoty y_j lišily v každém kroku, i když vycházely ze stejných hodnot z replay memory. To kvůli tomu, že se policy network v každém kroku učí, a mění se tak výsledné Q-hodnoty, které ovlivňují výsledek Bellmanovi rovnice. Díky tomu mohou učené Q-hodnoty značně divergovat, protože v jednom kroku je předložena cílová hodnota A a hned v následujícím je pro ten samý stav a akci předložena hodnota B. Tyto hodnoty se sice vzhledem k učícímu kroku od sebe zas tak neliší, nicméně v některých situacích to může způsobovat problémy. Implementaci s dvěma neuronovými sítěmi popisuje algoritmus 2.2:

Kód 2.2: Algoritmus Deep Q-learning s použitím Target network

1. vytvoř prázdnou replay memory R s maximální kapacitou N
2. vytvoř neuronovou síť policy network $\hat{Q}$ s náhodnými vahami
3. vytvoř neuronovou síť target network $\hat{Q}_{tar}$ se stejnými vahami jako $\hat{Q}$
4. pro epizodu od 1 do M opakuj
5. pro t od 1 do t_{max} opakuj
6. s pravděpodobností ε vyber náhodnou akci a_t
7. jinak vyber akci $a_t = \max_{a \in A} \hat{Q}(s_t,a,\theta)$
8. proveď akci a_t v prostředí a získej nový stav s_{t+1} a odměnu r_t
9. ulož zkušenost $e_t = (s_t,a_t,r_t,s_{t+1})$ do R
10. získej J náhodných vzorků pozorování (s_t,a_t,r_t,s_{t+1}) z R
11. Vytvoř vektor y o velikosti J s prvky:

$$y_j = \begin{cases} r_j, & \text{pro terminální stav } s_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}_{tar}(s_{j+1},a';\theta^-), & \text{pro neterminální stav } s_{j+1} \end{cases}$$

12. Proveď krok gradient descent metody na chybovou funkci:

$$\left(y_j - \hat{Q}(s_j,a_j;\theta)\right)^2$$

13. Zvyš hodnotu m o jedna a pokud $m = n$, tak $\theta^- = \theta$ a vynuluj m.

Tím že se původní články Mnih, Kavukcuoglu, Silver, Graves et al. (2013) a Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) zabývají agentem hrajícím Atari hry, tak se museli autoři zabývat zpracováním obrazových dat, které představují jednotlivé stavy prostředí. Pro zpracování obrázků byly použity konvoluční vrstvy, které se při učení chovají jako běžné neurony neuronové sítě. Tyto vrstvy tak přidaly do modelu další komplexitu navíc, kvůli které je o to víc žádoucí se zabývat právě stabilitou učení a cílových hodnot. Autoři Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) změřili maximální počet získaných bodů v Atari hrách bez a s použitím target network, a výsledky s použitím target network jsou v průměru o 36 % vyšší.

Metoda hlubokého Q-učení předvedla state-of-the-art výsledky právě na zmiňovaném simulátoru Atari her (Bellemare, Naddaf et al., 2013). Tato metoda později svojí flexibilitou dala možnost vzniku dalším metodám. Ukázkou takové metody může být například Double Q-learning (Hasselt et al., 2015), která využívá dvou neuronových sítí, které zároveň počítají Q-hodnotu. První slouží k vyhodnocení nejlepší akce v každém kroku, a druhá odhaduje hodnotu Q-funkce této akce. Další metodou vycházející z hlubokého Q-učení je metoda C51 (Bellemare, Dabney et al., 2017), která modeluje Q-hodnotu jako náhodnou proměnnou a síť se učí její rozdělení.

2.4 Evoluční algoritmy pro hledání optimálních konfigurací agentů

V této kapitole se trochu vzdálíme od neuronových sítí a posilovaného učení, a přejdeme do nové fáze testování agentů, a to k hledání jejich ideálních parametrů. Před testováním implementovaných agentů bude nejdříve potřeba najít jejich správnou konfiguraci. Výsledek agenta totiž neurčuje pouze jeho architektura, ale i jeho nastavení. Agent může mít sebe lepší architekturu, ale se špatným nastavením by nemusel vůbec pracovat podle očekávání.

(Dasgupta a Michalewicz, 1997) uvádějí, že téměř každý problém lze abstraktně vnímat jako hledání řešení v prostoru všech možných řešení. Často se jedná o optimalizační úlohy, kde cílem je nalézt nejlepší řešení. Například právě při optimalizaci agentů posilovaného učení hledáme takové nastavení agenta, které zajistí nejlepší výsledky po jeho naučení. Podobným více ilustračním optimalizačním problémem může být Knapsack problém (Martello a Toth, 1990), kde je optimalizačním cílem vybrat nejhodnotnější kombinaci předmětů, kterou lze umístit do batohu s omezenou velikostí a nosností.

Podle Eiben a Smith (2015) evoluční algoritmy představují metodu, která dokáže řešit optimalizaci hledání řešení a to pomocí principů inspirovaných evolucí. Vždy máme nějakou populaci možných řešení, kdy každému individuu z této populace dokážeme přiřadit hodnotu říkající, jak kvalitní řešení představuje. V knapsack problému to může být součet hodnot jednotlivých cenností, a u agenta jeho dosažené skóre. Myšlenkou poté je, že výběrem těch nejlepších jedinců v populaci a jejich křížením dokážeme postupně tvořit nové generace skládající se z lepších řešení než generace předchozí. Dále jako v přírodě občas dochází k mutacím, tedy náhodné změně jednoho z genů některých jedinců v populaci, čímž se zachovává rozmanitost jedinců.

Že obecný evoluční algoritmus vycházející z této myšlenky dokáže najít opravdu nejlepší výsledek je dokázáno v Eiben, Aarts et al. (1991). Podle tohoto článku dokáže evoluční algoritmus najít globální optimum, pokud splňuje následující vlastnosti:

- Evoluční algoritmus musí být monotónní. To znamená, že průměrné skóre v populaci se napříč generacemi zlepšuje.
- Výběr přeživších musí být konzervativní, což znamená, že do další generace musí být vždy vybrán alespoň jeden z nejlepších individuí.
- Evoluční algoritmus musí být štědrý, a tedy každý jedinec musí mít šanci přežít a podílet se na tvorbě nové generace.
- Individua v generaci musí tvořit strukturu quasi-propojených sousedů. To znamená, že dokážeme určit, jak blízko jsou od sebe individua v prostoru řešení, a dokážeme určit posloupnost operací, které by musely nastat, aby se z prvního stal druhý z dvojice.
- Procesy úvodní inicializace, mutací a křížení musí být schopné vytvořit jedince představující globální optimum.

Obecný evoluční algoritmus se skládá podle Eiben, Aarts et al. (1991) z následujících kroků:

1. Vytvoř úvodní generaci.
2. Vyber přeživší z populace.
3. Z přeživších vytvoř nové potomky.
4. Rozšiř populaci o tyto potomky.
5. Opakuj od kroku 2. dokud není hledání ukončeno.

Implementace jednotlivých kroků s ohledem na uvedené vlastnosti závisí na problému, který chceme pomocí evolučních algoritmů řešit. Jednotlivé strategie každého kroku budou vysvětleny dále. Je nutno ale zmínit, že existují tři populární implementace těchto kroků:

- Fogel et al. (1966) představil Evoluční programování, které pracují s genomy složených z reálných čísel, a jedince nových generací tvoří pouze za pomocí mutací.
- Dále Holland (2019) přišel s Genetickými algoritmy, které využívají stromových struktur pro definování genomů a hodí se tak pro řešení grafových problémů.
- Schwefel (1995) nakonec definoval evoluční strategie pracující s vektory reálných čísel, které vytvářejí další generace prostřednictvím křížení a mutací.

Jedinečností evolučních strategií je však využití sebe přizpůsobení (self-adaptation). To znamená, že součástí genomů jsou i parametry pro vytváření potomků, jako je například pravděpodobnost mutací nebo velikost kroku, o který se mutované části genomu mohou změnit atd. Myšlenka je taková, že jak se samotná řešení zlepšují evolucí, zlepšuje se i samotné hledání těchto řešení.

Jednotlivé implementace kroků evolučního algoritmu jsou závislé na povaze řešeného problému, a hlavně na implementaci genomů. Eiben a Smith (2015) uvádí, že evoluční algoritmy lze aplikovat na problémy jejichž řešení mají podobu: binárních řetězců, vektorů reálných čísel, permutace hodnot anebo stromových struktur. Metody pro křížení a mutace jsou tak často specifické pro každou z těchto podob. Dále budou popsány pouze metody pro genomy tvořené jako vektory reálných čísel, protože s touto reprezentací se bude později pracovat při hledání konfigurací agentů AIQ testu.

2.4.1 Výběr přeživších

Výběr přeživších je ta část evolučního algoritmu, ve které ohodnotíme jednotlivá individua v populaci vzhledem k tomu, jak kvalitní řešení představují. K tomu se využívá fitness funkce, která pro každý genom určí jeho hodnotu. Například u knapsack problému z dřívějška se jedná o součet hodnot jednotlivých předmětů v batohu.

Po ohodnocení jednotlivých individuí přichází na řadu výběr těch jedinců, kteří přežijí do další generace a zároveň budou tvořit potomky další generaci. Podle Eiben a Smith (2015) existuje několik různých metod, které můžeme při implementaci zvážit. Každá totiž poskytuje nějaké plusy a mínusy.

Mechanismus volby přeživších je nutno zvážit z důvodu toho, že se celý evoluční algoritmus

může dostat do stavu předběžné konvergence. To značí situaci, kdy jedinci s výjimečně vysokým skóre velmi rychle převezmou kontrolu nad celou populací, a tím se značně zmenší prostor prohledávaných řešení, a hledání uvízne v lokálním optimu. Je proto nutné vybírat přeživší tak, aby byla zachována diverzita populace. Jako nejpoužívanější metody výběru přeživších, které toto umožňují, udává Jebari (2013) ruletový výběr (Roulette wheel selection), výběr podle pořadí (Linear rank selection) a turnajový výběr (Tournament selection).

2.4.2 Křížení přeživších

Eiben a Smith (2015) popisují proces křížení jako proces, ve kterém se spojí genové informace z dvou rodičů do jednoho, který tvoří potomka. Tím, že se křížení děje až po výběru přeživších, tak očekáváme, že oba rodiče obsahují žádoucí vlastnosti, dosahující dobrého skóre, a tak promíchání těchto vlastností nám umožní vytvořit potomka, který díky těmto vlastnostem dosáhne ještě lepšího skóre. To nemusí nastat vždy, ale alespoň zjistíme, jakou hodnotu přinese řešení nacházející se mezi dvěma rodiči v prostoru řešeních.

Při práci s genomy vyjádřenými vektory reálných čísel máme podle Eiben a Smith (2015) pro křížení na výběr ze dvou možností. První se nazývá diskrétní rekombinace, a její myšlenka je jednoduchá. Určí se jeden nebo více bodů v genomu, a hodnoty do tohoto bodu převezme potomek od jednoho z rodičů, a hodnoty od tohoto bodu dále převezme od druhého z rodičů. Druhá metoda se nazývá aritmetická rekombinace, která spočívá v tom, že hodnoty jednotlivých složek genomu potomka vzniknou aritmetickým průměrem hodnot rodičů nad danými složkami. Obě tyto metody lze také kombinovat, takže část může vzniknout průměrováním, a druhá část pouze překopírováním hodnot.

2.4.3 Mutace

Mutace je operace nahrazení některých hodnot genomu za hodnotu novou. Jedná se o hlavní mechanismus prozkoumávání prostoru všech řešení, protože na rozdíl od křížení, které nám poskytuje body v prostoru, které mají nějaké souřadnice již známé, nebo se nachází v podprostoru mezi dvěma rodiči, tak mutace nám umožňují navštívit úplně nová místa v prostoru všech řešení. Evoluční programování (Fogel et al., 1966) jako konkrétní implementace evolučního algoritmu si vystačí dokonce pouze s mutacemi a křížení vůbec neprovádí.

Eiben a Smith (2015) popisují mutace jako stochastické operace, která pracují s parametrem p_c označující pravděpodobnost provedení mutace na dané složce genomu. Nejjednodušší forma mutace na genomech vyjádřených reálnými čísly se nazývá Náhodné resetování. Pro každou složku genomu musí být vymezen obor přístupných hodnot a s pravděpodobností p_c dojde k nahrazení aktuální složky genomu náhodnou hodnotou z tohoto oboru hodnot.

Jako druhou možnost provedení mutace popisují Eiben a Smith (2015) metodu nazývanou Creep mutation. Myšlenka za touto metodou je přičtení nebo odečtení malé hodnoty od

aktuální hodnoty složky genomu, kdy tato hodnota je nejčastěji zvolena náhodně. To, jestli k mutaci na dané složce dojde je opět ovlivněno pravděpodobností p_c. Do této metody lze přidat další pravděpodobností parametr, který bude určovat, jestli když k mutaci dojde, tak jestli změna bude velká (big creep) nebo malá (small creep). Pokud jednotlivé složky musí spadat do předem daného oboru hodnot, tak je potřeba hlídat, aby po změně složky nepřekročili meze tohoto oboru hodnot.

3. Implementace agentů

Náplní této diplomové práce je implementace agentů hlubokého Q-učení (sekce 2.3) do AIQ testu (sekce 1.6). Tato kapitola nejdříve představí výchozí podobu praktické implementace AIQ testu (sekce 3.1) a následně bude pojednávat právě o implementaci agentů hlubokého Q-učení (sekce 3.5 a 3.6), a všech podpůrných nástrojů, které umožňují chod těchto agentů (sekce 3.4).

O agentech mluvím v množném čísle, protože se ukázalo zajímavé zjistit, jak si bude vést v AIQ testu původní agent hlubokého učení podle Mnih, Kavukcuoglu, Silver, Graves et al. (2013) proti verzi rozšířené o druhou neuronovou síť target network z Mnih, Kavukcuoglu, Silver, Rusu et al. (2015). Tato kapitola tak popíše praktické implementace obou těchto verzí hlubokého Q-učení.

Finální podoba AIQ testu s všemi mnou implementovanými částmi je dostupná na platformě GitHub `https://github.com/TheMischko/AIQ-DQN-DP`.

3.1 Implementace AIQ testu

AIQ test v jeho aktuální podobě je naprogramován v jazyce Python verze 3 (Zeman, 2023), já konkrétně využíval verzi 3.8 pro programování jeho rozšíření. Úvodní verze skriptu je dostupná na adrese: `https://github.com/xvadooo/AIQ` (Ondřej Vadinský et al., 2024) Struktura adresáře testu a jeho soubory mají následující podobu:

- Adresář `agents`, který obsahuje implementace testovatelných agentů.
  - `__init__.py` je soubor tvořící z adresáře modul. Obsahuje výčet všech agentů, které lze aktuálně v rámci testu používat.
  - `Agent.py` je základní třída agenta, kterou každý nový agent musí implementovat, aby mohl být přidán do testu.
  - `Manual.py` představuje agenta ovládaného manuálně uživatelem.
  - `Random.py` je agent, který volí své akce čistě náhodně.
  - `Freq.py` implementuje jednoduchého agenta využívající frekvenční tabulku.
  - `Q_l.py` představuje agenta využívající Q-učení společně s technikou Eligiblity Traces.
  - `HQ_l.py`, který implementuje vylepšeného agenta používající Q-učení s automaticky se měnící rychlostí učení (Hutter a Legg, 2008).
  - `MC_AIXI.py` představuje aproximaci teoretického modelu superinteligence AIXI (Veness et al., 2010).
  - `VPG.py` představující agenta implementující Vanilla Policy Gradient metodu (Zeman, 2023).
  - `PPO.py` je následovník agenta `VPG` a implementuje metodu Proximal Policy Gradient (Zeman, 2023).
- Adresář `refmachines` obsahuje referenční stroje a vygenerované vzorky programů prostředí.
  - Adresář `samples` obsahuje poté právě ony navzorkované programy v souborech `*.samples`.
  - `__init__.py` je soubor tvořící z adresáře modul referenčních strojů.
  - `BF.py` představuje referenční stroj používající jazyk BF (Easter, 2020).
  - `BF_sampler.py` obsahuje kód pro generování programů prostředí pro referenční stroj BF.
  - `ReferenceMachine.py` slouží jako základní třída, kterou musí implementovat, každé nové prostředí, aby mohlo být používáno AIQ testem.
- `AIQ.py` je hlavním skriptem testu. Tento soubor je potřeba spouštět a obsahuje veškerou logiku testování s použitím agenta z adresáře `agents` a referenčního stroje z adresáře `refmachines`.
- `ComputeFromLog.py`, jelikož je možné průběžně akumulovanou odměnu prostředí AIQ testu ukládat do souboru, tak tento skript dokáže uložená měření přepočítat zpět na AIQ skóre.
- `Results2CSV.sh` je skript, který dokáže výsledky skriptu `ComputeFromLog.py` převést do souboru typu CSV.

3.2 Hlavní skript testu

Hlavní skript testu je soubor `AIQ.py`, který se spouští jako každý jiný script v jazyce python, a tedy příkazem `python AIQ.py`, pokud se nacházíme v adresáři testu. Pokud ne, tak se cesta ke skriptu bude samozřejmě lišit. Skript dále přebírá argumenty ze standardního vstupu. Výčet nejpoužívanějších z těchto argumentů popisuje následující seznam:

- `-a agent_name,param1,param2,...`
  - Slouží k volbě agenta, který bude testován, a nastavení jeho parametrů oddělených čárkami bez mezer.
- `-r ref_machine,param1,param2,...`
  - Slouží k volbě referenčního stroje s předáním parametrů opět v podobě hodnot oddělených čárkami bez mezer.
- `-l episode_length`
  - Nastavuje počet interakcí, které agent absolvuje v rámci každého testovaného prostředí.
- `-s sample_size`
  - Slouží pro nastavení počtu, kolik programů bude náhodně vybráno, a ovlivňuje tak počet na kolika prostředích bude agent testován. (čím vyšší bude tento počet tím nižší bude výsledná standardní chyba)
- `-t threads`
  - Ovlivňuje kolik procesů bude test používat při paralelizaci.
- `–log`
  - Zapnutí ukládání výsledků do samostatného souboru.
- `–verbose_log_el`
  - Tento příkaz umožňuje ukládat průběžné výsledky AIQ skóre po každé tisíceré iteraci.
- `–log_agent_failures`
  - Slouží k zapnutí logování jednotlivých programů, na kterých byl agent testován. Tento záznam se nachází u hlavních výsledků v logu po zapnutí `–log`. Je tak možné zjistit jaké skóre získal agent na jakém programu.

Příklad použití skriptu testu může vypadat způsobem jako ukazuje kód 3.1.

Kód 3.1: Úkázkový skript spuštění AIQ testu

```
1 python AIQ.py -r BF -a Q_l,0.0,0.5,0.5,0.05,0.9 -l 100000 -s 5000 --log    -
    verbose_log_el
```

3.3 Základní třída agenta

Dále bude představen soubor, který je pro implementaci nových agentů klíčový, a to soubor `Agent.py` v adresáři `agents`. Tento soubor totiž obsahuje kód základní třídy agenta, ze které

každý nový agent musí vycházet (dědit). Tato třída definuje tři důležité funkce, které musí každý agent implementovat a jejich definice je vidět v kódu 3.2. Mimo implementací těchto funkcí se samozřejmě očekává, že každý nový agent bude mít i vlastní konstruktor volající konstruktor této základní třídy.

Kód 3.2: Základní třída agenta

```
1  class Agent:
2      def __init__( self, refm, disc_rate ):
3          self.num_obs     = refm.getNumObs()
4          self.num_actions = refm.getNumActions()
5          self.sel_mode    = 0
6          self.disc_rate   = disc_rate
7
8      def __str__( self ):
9          raise NameError("You need to override Agent.__str__")
10
11     def reset( self ):
12         raise NameError("You need to override Agent.reset!")
13
14     def perceive( self, new_obs, reward ):
15         raise NameError("You need to override Agent.perceive!")
```

Nejdůležitější funkcí definovanou v základní třídě agenta je funkce `perceive`. Skrz tuto funkci totiž prostředí komunikují s agentem. Agentovi je přes parametry funkce předáno nové pozorování a aktuální odměna. Pozorování je konkrétně reprezentováno parametrem `new_obs`, který má podobu pole, jehož složky představují buňky výstupní pásky referenčního stroje.

Pro představu u referenčního stroje `BF(5)` se jedná o pole s jednou složkou jejíž hodnota může nabývat hodnot 0 až 4, u referenčního stroje `BF(5,4)` bude mít pole čtyři složky v tom samém rozsahu.

Druhý parametr této funkce se nazývá `reward`. a představuje získanou odměnu reprezentovanou jedinou číselnou hodnotou v rozsahu od $< -100; 100 >$. Očekává se, že agent na základě těchto dvou hodnot a svých vnitřních stavů vrátí z této funkce akci, kterou chce provést. Akce musí být celočíselná hodnota v rozsahu od $< 0;$ počet možných akcí), a tedy v případě referenčního stroje `BF(5)` se jedná o rozsah $< 0{,}4 >$.

Další dvě funkce, které agent musí implementovat, jsou již přímočařejší. Funkce `reset` je volána před každým testováním na novém prostředí v podobě programu referenčního stroje. Očekává se, že při zavolání této funkce agent vyresetuje své vnitřní stavy a připraví se na testování na novém neznámém prostředí.

Poslední povinná funkce k implementování je funkce `__str__`, která je běžnou součástí jazyka Python. AIQ test nicméně používá tuto funkci ke generování textového identifikátoru, který jednoznačně popisuje agenta a jeho aktuální konfiguraci. Očekává se, že vrácený řetězec

bude začínat názvem třídy agenta (`Freq`, `Q_l`, `MC_AIXI`) a poté bude následovat v kulatých závorkách výčet hodnot parametrů, tak jako by se pouštělo testování agenta s těmito parametry v rámci volání skriptu `python AIQ.py`. Toho test využívá při vypisování názvu agenta do hlavičky výsledků testu, ale i k pojmenovávání souborů logů při použití testu s příznaky `-log` a `-verbose-log-el`.

3.4 Implementace obecných částí hlubokého Q-učení

Má implementace agentů zahrnuje implementaci agenta hlubokého učení s jednou neuronovou sítí a poté implementaci druhého agenta pracující s dvěma neuronovými sítěmi. Ačkoliv oba agenti mají hodně společného, musí v testu existovat jedinečná třída pro oba z nich. I tak lze ale společné chování a společné části vytáhnout z implementací agentů ven do samostatných částí, které agenti mohou využívat, a nemusí je implementovat každý zvlášť.

Samostatně lze totiž vytvořit například replay memory, jejíž implementaci budou oba agenti využívat úplně stejně. Stejně tak lze vyextrahovat z agentů i kód neuronové sítě, která se mezi agenty neliší, pouze ji architektura s dvěma neuronovými využívá dvakrát. Když se zblízka podíváme i na samotné chování agentů, jak implementují rozhodovací politiku, zjistíme, že i obecné chování agentů lze implementovat pro oba společně v rámci samostatné třídy. Tento přístup i odráží správné zásady programování jako třeba zásadu DRY (don't repeat yourself) (Hunt a Thomas, 2000). Tato podkapitola bude právě vysvětlovat implementaci těchto sdílených částí.

3.4.1 Implementace neuronové sítě

První ze sdílených částí obou agentů je samotná kostra agenta, kterou představuje dopředná neuronová síť. Architektura podle (Mnih, Kavukcuoglu, Silver, Graves et al., 2013) odpovídá vícevrstvému perceptronu, který má vstupní vrstvu s počtem neuronů odpovídající vektoru pozorování nebo vektoru sekvence pozorování představující krátkou historii předcházejících stavů. Dále neuronová síť obsahuje skryté vrstvy jejíž počty a rozměry musí být konfigurovatelné. Výstupní vrstva neuronové sítě musí obsahovat tolik neuronů, kolik je v prostředí možné provést akcí.

K implementaci neuronové sítě jsem využil knihovnu PyTorch (Paszke et al., 2019), a konkrétně její modul `nn` pro neuronové sítě. Výhodou je využití veškeré obecné logiky neuronové sítě, jako je například zpětná propagace a učení sítě, bez nutnosti ji implementovat. PyTorch umožňuje vytvořit neuronovou síť jako vlastní třídu, která dědí ze základní třídy `nn.Module`.

Moje třída implementující neuronovou síť se nazývá obecně `NeuralNet`. Konstruktor této třídy přijímá důležité parametry k tvorbě sítě o správných velikostech. Jeho kód lze vidět v kódu 3.3. První parametr `input_size` představuje velikost vstupní vrstvy, a očekáváme, že bude odpovídat vektoru pozorování nebo historie pozorování. Druhý parametr `output_size` naopak představuje velikost výstupní vrstvy, která bude mít rozměry podle počtu možných akcí. Třetí parametr `linear_layer_sizes` bude mít podobu pole, jehož složky budou velikosti jednotlivých skrytých vrstev. Tento přístup umožní experimentovat s velikostmi ale i počty skrytých vrstev bez nutnosti měnit kód neuronové sítě.

Kód 3.3: Třída neuronové sítě

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, output_size, linear_layer_sizes):
        super().__init__()
        self.layers = nn.ModuleList()
        normal_function = nn.init.xavier_normal_

        last_layer_size = 0
        for i in range(len(linear_layer_sizes)):
            if linear_layer_sizes[i] <= 0:
                break
            prev_size = linear_layer_sizes[i - 1] if (i - 1) >= 0 else
input_size
            self.layers.append(
                nn.Linear(int(prev_size), int(linear_layer_sizes[i]))
            )
            last_layer_size = int(linear_layer_sizes[i])
        self.layers.append(
            nn.Linear(last_layer_size, int(output_size))
        )


        for layer in self.layers:
            normal_function(layer.weight)
```

Po představení argumentů konstruktoru třídy následuje inicializace všech vnitřních parametrů sítě. Jednotlivé vrstvy v PyTorch jsou reprezentovány jako moduly, a jelikož jich budeme mít proměnlivý počet, tak jednotlivé vrstvy budeme uchovávat v kolekci `ModuleList`. Aktuální implementace vyžaduje alespoň jednu skrytou vrstvu. Tvorba vrstev neuronové sítě probíhá tak, že iterujeme pole `linear_layer_sizes` a do kolekce vrstev layers se přidá nová lineární vrstva `nn.Linear`, která má počet vstupních neuronů rovnou počtu výstupních neuronů v předcházející vrstvě. Tento počet uchovává proměnná `prev_size`.

Napojení na vstupní vrstvu se udělá tak, že přidáme podmínku, která pro první vrstvu nastaví počet vstupů rovnou hodnotě `input_size`. Po celém cyklu přidáme stejným způsobem ještě vrstvu výstupní, jejíž počet výstupních neuronů odpovídá hodnotě `output_size`. V posledním kroku inicializujeme váhy vrstev pomocí inicializační funkce `normal_function`.

Po inicializaci neuronové sítě je potřeba určit, jak bude probíhat dopředný průchod celou sítí. Toho se docílí tak, že třída `NeuralNet` implementuje funkci `z nn.Module` zvanou `forward`. Má implementace této funkce je ukázána v kódu 3.4. Parametr této funkce je označený jako `x`, a představuje matici vstupních dat neboli pozorování. Nejdříve se zjistí, jestli tyto vstupní hodnoty mají správný datový typ příkazem `x.type`. Poté díky tomu, že naše vrstvy

máme správně sekvenčně uložené v kolekci `layers`, tak přes tyto vrstvy jednoduše iterujeme a přepisujeme hodnotu proměnné `x` tím, že proměnnou `x` předáme jako vstup jednotlivým vrstvám a tento výsledek ještě zabalíme do aktivační funkce v podobě sigmoidy.

Aktivační funkci ale nechceme použít pro výsledek výstupní vrstvy. Iterujeme tak pouze do předposlední vrstvy a průchod poslední vrstvou bez aktivační funkce provádíme samostatně a tento výsledek rovnou vracíme jako výsledek celého dopředného průchodu.

Kód 3.4: Dopředný průchod neuronovou sítí

```
1     def forward(self, x):
2         x = x.type(torch.float32)
3         for layer in self.layers[:-1]:
4             x = torch.sigmoid(layer(x))
5         return self.layers[-1](x)
```

Funkce sigmoidy je hojně používaná pro aktivační funkce neuronových sítí, je ale při jejím použití důležité myslet na to co píše Brownlee (2021) a tedy pracovat pouze s hodnotami v rozsahu $(-1; 1)$ a inicializovat váhy podle Xavierova rozdělení. Proto je v kód 3.4 použita inicializační funkce `nn.init.xavier_normal_()`.

Následně jsem ještě v modulu neuronové sítě definoval dvě statické metody vracející chybovou funkci a optimalizátor gradientního sestupu, které lze vidět v kódu 3.5. Jako chybová funkce je použita metoda `SmoothL1Loss` (Girshick, 2015), která je podobná metodě střední kvadratické chyby, jen se zvládá lépe vypořádávat s odlehlými hodnotami, a obecně zaručuje robustnější učení. Jako optimalizátor gradientního sestupu je možno zvolit mezi metodou RMSProp, která je používaná v originálních pracích o hlubokém Q-učení, nebo metodou ADAM, která je v dnešní době velice populární pro učení neuronových sítí, protože například oproti metodě RMSProp, pracuje jak s prvním, tak s druhým momentem gradientu, což vede na provedení přesnějšího kroku gradientní sestupu. ADAM zároveň vypočítává hybnost gradientu, díky které dokáže lépe vyhlazovat výkyvy hodnot v chybové funkci (Kingma a Ba, 2017).

Kód 3.5: Chybová funkce a optimalizátor

```
def get_optimizer(model, learning_rate=0.00025, use_rmsprop=True):
    if use_rmsprop:
        return torch.optim.RMSprop(
            model.parameters(),
            lr=learning_rate,
            momentum=0.95,
            eps=0.01
        )
    return torch.optim.Adam(
        model.parameters(),
        lr=learning_rate,
        weight_decay=0.01,
        amsgrad=True,
    )
def get_criterion(reduction = 'mean'):
    return nn.SmoothL1Loss(reduction=reduction)
```

3.4.2 Implementace replay memory

V původní práci o hlubokém Q-učení přisuzují Mnih, Kavukcuoglu, Silver, Graves et al. (2013) velkou část efektivity struktuře zvané replay memory. Tato datová struktura umožňuje uchovávat zkušenosti agenta v podobě uspořádané čtveřice: pozorování, provedená akce, získaná odměna a nové pozorování; a dále umožňuje z těchto zkušeností provádět náhodný výběr.

V mém kódu jsem tyto čtveřice označil jako přechody (transitions) a pevně jsem definoval jejich strukturu díky pythonovské funkci `namedtuple`. Replay memory jsem poté vytvořil jako třídu, které v sobě uchovává tyto přechody v rámci kolekce zvané `deque` označené jako `memory`, které lze nastavit maximální velikost. To je další vlastnost, o které mluví Mnih, Kavukcuoglu, Silver, Graves et al. (2013) v rámci replay memory. Jak lze vidět v kódu 3.6, tak velikost této kolekce stanovuje parametr konstruktoru třídy replay memory nazvaný jako `capacity`.

Třída dále obsahuje funkci pro přidání dat do replay memory s názvem push, která z dat vytvoří právě `namedtuple Transition` a přidá ji do interní kolekce `memory`. Náhodný vzorek přechodů poté lze získat použitím funkce sample, které se předá počet požadovaných vzorků parametrem `batch_size` a přes funkci `sample` knihovny `random` je vrácen náhodný vzorek dané velikosti. Tato funkce vrací vždy náhodný vzorek bez opakování jednotlivých položek.

Kód 3.6: Třída ReplayMemory

```
Transition = namedtuple(
    'Transition',
    ('state', 'action', 'next_state', 'reward')
)

class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        transition = Transition(*args)
        self.memory.append(transition)

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

3.4.3 Obecná třída agentů posilovaného učení

Jelikož oba budoucí agenti posilovaného Q-učení spolu nesdílejí jen některé vybrané části, jako neuronovou a síť a replay memory, ale sdílí i nějakou společnou logiku, například politiku výběru akcí, tak mi přišlo vhodné tuto společnou logiku vyextrahovat do samostatné třídy, ze které oba agenti budou později dědit.

Tuto společnou základní třídu jsem pojmenoval `IDeepQLAgent`, kdy již samotný název by měl navozovat dojem, že se nejedná o třídu žádného konkrétního agenta, ale spíše o rozhraní (interface), kterého využívají jiní agenti. Konstruktor této základní třídy přebírá všechny potřebné parametry, které by obecný agent hlubokého Q-učení měl potřebovat a ukládá je do vnitřních proměnných. Mezi tyto společné parametry patří:

- `learning_rate` – tento parametr ovlivňuje, jak velký krok se provádí v rámci gradientního sestupu při učení neuronové sítě,
- `gamma` – vychází z parametru posilovaného učení a jedná se diskontní faktor pro budoucí odměny,
- `batch_size` – říká agentovi, jak velké vzorky má náhodně vybírat z replay memory pro učení,
- `epsilon` – slouží k zadání počáteční hodnoty epsilon pro epsilon-greedy politiku agenta,
- `epsilon_decay_length` – udává počet kroků, během kterých se lineárně snižuje hodnota epsilon z maximální hodnoty (nejčastěji 1) na minimální hodnotu $< 0{,}1$). Pokud agent pracuje se statickou hodnotou epsilon, nastavuje se tato hodnota na nulu.
- `neural_size_l1, neural_size_l2, neural_size_l3` – nastavují šířky skrytých vrstev neuronové sítě. Hodnota `neural_size_l3` může být nula, a v tom případě je třetí vrstva

z architektury vynechána.

- `use_rmsprop` – slouží k volbě optimalizační metody, hodnota 1 použije RMSProp, zatímco hodnota 0 použije metodu ADAM.
- `history_length` – nastavuje délku historie, která tvoří vektor pozorování pro vstup neuronové sítě.
- `Lambda` – nastavuje hodnotu parametru λ pro eligiblity traces.
- `eligiblity_strategy` – Slouží pro výběr metody, kterou využívají eligibility traces. Hodnota 0 označuje verzi nahrazovací metody, která pracuje pouze s jedinou předcházející akcí. Hodnota 1 slouží pro výběr akumulační metody, a hodnota 2 stojí za dutch metodou neboli za standardní nahrazovací metodou.

Tyto parametru jsou v těle konstruktoru uloženy do atributů třídy, aby mohli být později použity v jiných metodách. Zároveň se v těle konstruktoru inicializují další pomocné proměnné například `self.prev_action = None`, která uchovává poslední zvolenou akci, kterou lze v metodě `perceive` použít jako akci, která vedla do aktuální stavu.

Základní třída dále implementuje metodu `reset`, viz kód 3.7. Ta je volána, když AIQ test začne testovat agenta na novém prostředí, a očekává, že v této funkci dojde k vyresetování vnitřních proměnných do původních stavů. Je zde například vytvořena prázdná instance replay memory pro ukládání učících vzorků, ale také i třeba hlavní neuronová síť policy network s úvodními vahami. Na výchozí hodnotu se zde nastavuje i hodnota epsilon a počet učících kroků `steps_done`. Pokud agent pracuje s eligility traces, tak se v rámci funkce `reset` inicializuje vektor jejich hodnot eligibility na nulu.

Kód 3.7: Funkce reset obecné třídy IDeepQLAgent

```
1  def reset(self):
2      self.memory = ReplayMemory(10000)
3      # Network evaluating Q function
4      self.policy_net = NeuralNet(
5    self.neural_input_size,self.num_actions,
6    self.neural_size_l1, self.neural_size_l2,
7    self.neural_size_l3)
8      self.optimizer = get_optimizer(
9    self.policy_net,
10   learning_rate=self.learning_rate)
11     self.steps_done = 0
12     self.epsilon = self.starting_epsilon
13     self.eligibility = torch.zeros((self.batch_size,
14                                     self.num_actions))
15     self.reset_values_for_plots()
```

Důležitá je ale hlavně funkce `perceive`, kterou lze vidět v kódu 3.8. Tato funkce je totiž odpovědná za interakci agenta s prostředím. Prostředí ji v každé iteraci zavolá a předá

agentovi nové pozorování `observations` a odměnu `reward`, a očekává, že agent na základě těchto informací vrátí prostředí akci, kterou chce provést. Oba agenti tento krok dělají stejně: nejdříve se zpracuje pozorování, které se společně s předchozím pozorováním, provedenou akcí, získanou odměnou a zpracovaným novým pozorování uloží do replay memory. Dále se provede učící mechanismus `learn_from_experience`, který si každý agent musí implementovat sám. Poté se vybere akce pomocí metody implementující politiku agenta `get_action`, která bude ukázána později. Akce i nové pozorování se uloží a zvolená akce je vrácena prostředí.

Kód 3.8: Metoda perceive pro komunikaci s prostředím

```
    def perceive(self, observations, reward):
        new_state_vec = self.observations_to_vec(observations)
        new_state_with_history_tensor =
self.add_history_to_observation(new_state_vec)
        new_state_with_history_unsqueezed =
new_state_with_history_tensor.unsqueeze(0)
        if (self.current_state_with_history is not None)
 and (self.prev_action is not None):
            self.memory.push(
                self.current_state_with_history,
                self.prev_action,
                new_state_with_history_unsqueezed,
                torch.tensor(reward / self.REWARD_DIVIDER,
  dtype=torch.float32).unsqueeze(0))
        self.rewards_given.append(reward / self.REWARD_DIVIDER)
        self.learn_from_experience()
        opt_action = self.get_action(new_state_with_history_tensor)
        self.prev_action = torch.tensor(opt_action).unsqueeze(0)
.unsqueeze(0)
        self.cached_state_raw = observations
        self.current_state_with_history =
new_state_with_history_unsqueezed
        self.save_prev_values(opt_action, new_state_vec,
      new_state_with_history_unsqueezed)

        return opt_action
```

Funkce `observations_to_vec` slouží k převedení pole pozorování, které má podobu pole hodnot výstupní pásky referenčního stroje, na one-hot encoding v podobě PyTorch struktury zvané tensor. Aktuální implementace umožňuje rozšířit tento tensor pozorování o další pozorování, která předcházela aktuálnímu stavu v podobě historie. K tomu slouží funkce `add_history_to_observation`. Tato funkce vytvoří vektor, který obsahuje one-hot encoding předchozích pozorování o zadaném počtu, a za tyto hodnoty posadí aktuální pozorování. Simuluje se tak skládání vícero obrázků k tvorbě jednoho pozorování z původní práce Mnih,

Kavukcuoglu, Silver, Graves et al. (2013).

Odměny v aktuální podobě AIQ testu nabývají hodnot v rozmezí $< -100; 100 >$. Z tohoto důvodu je při ukládání odměny do replay memory její hodnota nejdříve vydělena konstantou `REWARD_DIVIDER`, která má hodnotu 100. Odměna, která poté figuruje při učení neuronové sítě, má díky tomu hodnoty v rozmezí $< -1; 1 >$. Hodnoty, které slouží neuronové síti jako učící vzory je vždy dobré normalizovat. To jsem se rozhodl udělat na základě tohoto tvrzení z Sarle (2002):

> „Pokud používáme dvě a více cílových proměnných a chybová funkce je citlivá na stupnici, což střední rozdíl chyby například je, tak variabilita hodnot cílových proměnných mezi sebou navzájem může ovlivňovat, jak dobře se síť naučí danou cílovou proměnnou."

Další společnou metodou je funkce `get_action`, kterou lze vidět použitou v metodě `perceive`. Funkce `get_action` na vstupu získá aktuální pozorování a vrátí akci. Tato funkce implementuje epsilon-greedy politiku. Jak lze vidět na druhém řádku kódu 3.9, tak nejdříve je načteno náhodné číslo z intervalu $< 0; 1 >$, které je porovnané s aktuální hodnotou `epsilon`. Pokud je náhodné číslo menší než epsilon, tak je vrácena náhodná akce, tedy s pravděpodobností ϵ je vrácena náhodná akce. Naopak s pravděpodobností $1 - \epsilon$ je zavolána metoda `compute_action_from_Q_value`, které je předán stav.

Kód 3.9: Implementace ε-greedy politiky

```
1     def get_action(self, state):
2         is_random = random.random() < self.epsilon
3         legal_actions = [
4                 action for action in range(self.num_actions)
5         ]
6         action = None
7         if is_random:
8             action = random.choice(legal_actions)
9         else:
10            action = self.compute_action_from_Q_value(state)
11        return action
```

Funkce `compute_action_from_Q_value` je velice jednoduchá, jak lze vidět v kódu 3.10. V bloku `torch.no_grad` se provede dopředný průchod neuronovou sítí, která vrátí vektor odhadnutých Q-hodnot pro možné akce, a z těchto odhadů je vybrán index té nejvyšší hodnoty pomocí `np.argmax` funkce. Blok `torch.no_grad` je použit z důvodu, aby se nám nepočítaly gradienty na výstupu neuronové sítě mimo proces učení. Knihovna PyTorch totiž nativně ke každému výsledku přibaluje i referenci na zdroj výsledku, aby šla právě provést zpětná propagaci chyby, což aktuálně není žádoucí.

Kód 3.10: Výběr akce podle výsledku neuronové sítě

```
1 def compute_action_from_Q_value(self, state):
2         with torch.no_grad():
3             action_values = self.policy_net.forward(state).tolist()
4             policy = np.argmax(action_values)
5             return policy
```

Nyní následuje výčet zbylých funkcí základní třídy agenta hlubokého Q učení, s krátkým popisem bez ukázky kódu:

- `save_prev_values`
  - Metoda, která uloží do vnitřních proměnných vybranou akci a stav, ve kterém byla tato akce provedena, a připraví správně historii stavů s ohledem na aktuální stav.
- `get_learning_batches`
  - Metoda, která z replay memory náhodně vybere záznamy v počtu podle nastavené velikosti batche.
- `decrement_epsilon`
  - Slouží k automatickému snížení hodnoty epsilon o předem daný krok. Funkce se sama přeskakuje, pokud není epsilon decay nastaven.
- `learn_from_experience`
  - Tuto metodu si musí každý agent implementovat sám a slouží k samotné učící logice agenta.
- `reset_values_from_plots`
  - Při ladění chyb se hodí zobrazovat grafy vývoje učení tedy například hodnot chybové funkce. Tato metoda vymaže všechna pole sbírající hodnoty k vykreslení, aby se na novém programu začínalo od začátku.
- `append_q_values`
  - Tato funkce zpracuje Q-hodnoty akcí a správně je uloží do ladícího pole pro graf vývoje Q-hodnot.
- `reset_trace`
  - Tato metoda implementuje funkci podle Suri (2021) pro resetování hodnot eligiblity traces.
- `update_trace`
  - Pomocná metoda, která vrací přímo novou hodnotu eligibility traces podle nastavené strategie (Suri, 2021).
- `update_eligibility`
  - Metoda, která by se měla volat v rámci `learn_from_experience`. Její implementace podle Suri (2021) přijímá vektor chyby rozdílu Q-hodnot z neuronové sítě a cílových hodnot, a podle hodnot eligibility traces tuto chybu pozmění. Metoda tedy upraví jak aktuální hodnotu eligibility traces tak vektor chyb.

3.5 Implementace základního agenta hlubokého Q-učení

Po vytvoření potřebných komponent hlubokého posilovaného učení jsem již mohl vytvořit plnohodnotného agenta vycházející z Mnih, Kavukcuoglu, Silver, Graves et al. (2013). Tohoto agenta jsem implementoval jako třídu s názvem `DQ_l`, která dědí ze základní třídy `IDeepQAgent` představené v předchozí sekci. Díky tomu je zaručeno jednak, že nový agent bude kompatibilní s AIQ testem díky tomu, že `IDeepQAgent` rozšiřuje obecnou třídu agentů Agent, a zároveň bude mít přístup ke všem komponentám z předchozí sekce. Pro plnohodnotnou funkčnost agenta je potřeba v novém agentovi implementovat pouze učící mechanismus v rámci funkce `learn_from_experience` a poté ještě připravit správnou textovou reprezentaci agenta v rámci funkce `__str__`.

3.5.1 Učící mechanismus

Učící mechanismus je volán při každé interakci s prostředím. Jak lze vidět v kódu 3.11, tak kvůli tomu je důležité nejdříve zkontrolovat, zda máme dostatek učících dat v replay memory, aby se z ní mohl získat náhodný vzorek o velikosti `batch_size`. Pokud dostatek dat nemáme, tak funkce končí a agent dále interaguje s prostředím, dokud nenasbírá dostatek dat.

Pokud data máme, tak jsou získaná z replay memory pomocí funkce `get_learning_batches`, která je všechny převede do správných datových typů, které mají podobu PyTorch tensorů. Takto získáme: původní pozorování `state_batch`, provedené akce v těchto pozorováních `action_batch`, získané odměny za tyto akce `reward_batch`, a nová pozorování `next_state _batch`. Hodnoty těchto proměnných na jednotlivých indexech odpovídají posbíraným zkušenostem v rámci stejných interakcí agenta s prostředím.

Dále je potřeba zjistit aktuální Q hodnotu pro provedené akce v původních stavech. Toho docílíme tak, že provedeme dopředný průchod neuronovou sítí `policy_net` s daty původních stavů `state_batch`, a vybere pouze Q hodnoty provedených akcí z interakcí pomocí funkce `gather(1, action_batch)`. To se dělá proto, že jen k těmto akcím aktuálně známe získanou odměnu.

V dalším kroku vytvoříme cílová data, proti kterým budeme počítat chybu neuronové sítě. Tato cílová data odpovídají Bellmanově rovnici, a tedy se jedná o odměny plus Q-hodnoty nejlepších akcí v následujících stavech. Tyto Q-hodnoty jsou získány opět dopředným průchodem `policy_net`, ale tentokrát s hodnotami následujících stavů v podobě `next_state_batch`. Získané Q-hodnoty se ještě musí vynásobit diskontním koeficientem gamma před sečtením s odměnami. Tento výpočet se provádí v bloku `with torch.no_grad`, z toho důvodu, že PyTorch na všechny hodnoty nabaluje i funkci pro zpětnou propagaci chyby. Pro tyto hodnoty, které slouží jako cílová data, ale není žádoucí, aby nějak pozměňovali váhy neuronové sítě, a to zařídíme právě obalením výpočtu do tohoto bloku.

Kód 3.11: Učící mechanimus agenta DQ_l

```
def learn_from_experience(self):
    if len(self.memory) < self.batch_size:
        return
    state_batch, action_batch, reward_batch, next_state_batch =
self.get_learning_batches()

    q_values = self.policy_net(state_batch).gather(1, action_batch)

    q_next_values = None
    with torch.no_grad():
        q_next_values = reward_batch + self.gamma
 * self.policy_net(next_state_batch).max(1)[0]

    if self.uses_eligibility:
        loss = self.criterion(q_values, q_next_values.unsqueeze(1))
        loss = self.update_eligibility(action_batch, loss)
        loss = loss.mean()
    else:
        loss = self.criterion(q_values, q_next_values.unsqueeze(1))
    self.optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 1)
    self.optimizer.step()

    self.decrement_epsilon()
    self.steps_done += 1
```

Po vytvoření cílových hodnot se musí určit chyba sítě, kterou uchovávám v proměnné `loss`. Výpočet této proměnné ovlivňuje, jestli agent zrovna používá eligibility traces. Chybu neuronové sítě nám vypočte chybová funkce v podobě proměnné criterion, která odpovídá variantě střední kvadratické chyby přes získané Q-hodnoty a cílová data z Bellmanovi rovnice. Chybová funkce většinou vrací jedinou hodnotu odpovídají průměru přes všechna data, pokud se ale používají eligibility traces, tak funkce vrací vektor před průměrováním, ten je poté upraven funkcí `update_eligibility`, která zahrne právě zmiňované eligibility traces a poté je na něm manuálně provedeno zprůměrování.

Na proměnné `loss` už nyní lze provést funkci `backward`, která provede zpětnou propagaci chyby a vypočte gradient na všechny hodnoty, které se podíleli na tvorbě proměnné `loss`. V tomto kontextu se jedná hlavně o všechny váhy neuronových spojení neuronové sítě `policy_net`, která byla použita pro výpočet hodnot `q_values`, protože jediná nebyla obalená v bloku `torch.no_grad`. Nejdříve je ale potřeba odstranit gradienty na vahách sítě z předchozí iterace učící metody, aby nedošlo k jejich akumulaci pomocí příkazu `optimizer.zero_grad`. Dále

se provede zmíněná zpětná propagace příkazem backward. Pro lepší stabilitu učení je lepší oříznout gradienty vah sítě přesahující hodnotu 1 a nastavit je na hodnotu 1 pomocí funkce `clip_grad_value_`. Tato operace je známá jako gradient clipping a pomáhá podle Zhang et al. (2021) zabránit možnému explozivnímu růstu gradientů.

Úpravu samotných vah podle gradientů provede `optimizer`, představující variantu gradientního sestupu, pomocí funkce `step`.

Nyní byl proveden jeden učící krok, váhy neuronové sítě byly upraveny, aby se snížil rozdíl mezi vypočtenými Q-hodnotami oproti hodnotám z Bellmanovi rovnice. Dále už je jen potřeba provést epsilon decay, tedy snížení hodnoty epsilon pomocí funkce `decrement_epsilon`.

3.5.2 Parametry agenta a jeho textová reprezentace

Implementace textové reprezentace agenta v rámci metody `__str__` nebude ukázána, protože se jedná o jednoduché formátování textu. Výsledný řetězec má ale následující podobu: nejdříve je vypsán název agenta, tedy `DQ_l`. Následně v kulatých závorkách jsou vypsány parametry agenta v následujícím pořadí:

1. learning rate neuronové sítě,
2. gamma,
3. velikost učících vzorků,
4. počáteční epsilon,
5. délka epsilon decay,
6. šířka první vrstvy neuron. sítě,
7. šířka druhé vrstvy neuron. sítě,
8. šířka třetí vrstvy neuronové sítě,
9. příznak, zda se používá RMSProp,
10. délka historie pozorování,
11. lambda,
12. strategie eligibility traces

Tyto parametry odpovídají parametrům konstruktoru základní třídy `IDeepQLAgent` ze sekce 3.4.3. Doporučené hodnoty těchto parametrů pro použití tohoto agenta lze nalézt v příloze A.

3.6 Implementace agenta s target network

Druhého agenta hlubokého Q-učení jsem pojmenoval DDQ_l, a představuje architekturu představenou v Mnih, Kavukcuoglu, Silver, Rusu et al. (2015). Tato architektura agenta pracuje se dvěma neuronovými sítěmi. První, která se označuje jako policy network, se tradičně stará o rozhodování agenta při interakcích s prostředím, tak jako tomu bylo u předchozího agenta DQ_l. Policy network se také stejně jako u předchozí agenta učí v každé iteraci v rámci funkce `learn_from_experience`.

Změna ale nastává v tom, jak se získávají cílové hodnoty pro určení chyby této sítě. Předchozí agent k tomu využívá Bellmanovu rovnici, ve které se Q-hodnota určila pomocí policy network. V této architektuře se právě v tomto kroku nahradí policy network za druhou neuronovou síť zvanou target network. Tato síť má úplně stejný počet neuronů a úplně stejné jejich uspořádání do vrstev jako policy network. Rozdíl je v tom, že target network se ale neučí pomocí klasické zpětné propagace, nicméně po uplynutí zadaného počtu kroků se váhy policy network překopírují do target network, a na krátký moment jsou tyto sítě identické, dokud není proveden další krok učícího algoritmu, a váhy policy network jsou znovu upraveny.

Target network, tím že je jakýmsi historickým obrazem policy network, a po zadanou dobu se nemění, tak umožňuje stabilnější tvorbu cílových hodnot, proti kterým se policy network učí. Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) ukázali, že tímto se stabilizuje celý proces učení, a je tím možné na složitých prostředích, ve kterých je učení často nestabilní, dosáhnout vyšších výsledků. Pokud by totiž v původní architektuře došlo ke stavu, kdy z replay memory dostaneme stejné vzorky v rámci dvou po sobě jdoucích učících cyklů, tak se cílové hodnoty z tohoto vzorku budou lišit, protože se mezitím změnili váhy policy network, a výsledek z Bellmanovi rovnice bude rozdílný. Pokud ale po tyto dva po sobě jdoucí kroky bude v Bellmanově rovnice figurovat neuronová síť, jejíž váhy se v těchto krocích nezmění, tak výsledky budou identické.

Pro implementaci agenta, který stojí na této architektuře jsem vytvořil nového AIQ agenta `DDQ_l`, který dědí ze základní třídy agentů hlubokého Q-učení `IDeepQLAgent`. Druhou neuronovou síť `target_net` vytváří agent v rámci `reset` funkce agenta, jak lze vidět v kódu 3.12. Jelikož jde o přetížení funkce `reset`, kterou implementuje i třída `IDeepQLAgent`, tak se nesmí zapomenout implementace z této třídy zavolat, jinak by agent nefungoval správě. V kódu 3.12 je dále vytvořena neuronová síť úplně stejným způsobem jako je vytvořena `policy_net` v `reset` funkci základní třídy `IDeepQLAgent`. Je důležité, aby neuronová síť měla úplně stejný počet vrstev o stejných velikostech jako `policy_net`, aby šly později překopírovat váhy mezi sítěmi. Zároveň je žádoucí, aby sítě měly od prvního okamžiku identické úvodní váhy. Toho docílíme načtením vah do `target_net` pomocí funkce `load_state_dict`.

Kód 3.12: Inicializace target network

```
def reset(self):
    IDeepQLAgent.reset(self)
    self.target_net = NeuralNet(
                        self.neural_input_size,
                        self.num_actions, [
                            self.neural_size_l1,
                            self.neural_size_l2,
                            self.neural_size_l3])
    self.target_net.load_state_dict(self.policy_net.state_dict())
```

Pro mechanismus překopírování vah jsem vytvořil v rámci agenta další funkci nazvanou `copy_network_weights`, která se stará právě o překopírování vah policy network do target network. Implementaci této funkce lze vidět v kódu 3.13. PyTorch umožňuje iterovat přes jednotlivé vrstvy neuronové sítě, protože je ukládá do slovníkové podoby, kterou lze získat příkazem `state_dict`. Překopírování je tak prostý cyklus nad indexy toho slovníku a překopírování hodnot pod tímto indexem z vah policy network do slovníku vrstev target network pod identickým indexem.

Kód 3.13: Synchronizační funkce vah sítí

```
def copy_network_weights(self):
    policy_net_state_dict = self.policy_net.state_dict()
    target_net_state_dict = self.target_net.state_dict()

    policy_net_keys = set(policy_net_state_dict.keys())
    target_net_keys = set(target_net_state_dict.keys())

    if policy_net_keys != target_net_keys:
        raise ValueError("Policy and target network state
     dictionaries have different keys.")

    for key in target_net_state_dict:
        target_net_state_dict[key] =
            policy_net_state_dict[key] * self.tau
            + target_net_state_dict[key] * (1-self.tau)
    self.target_net.load_state_dict(target_net_state_dict)
```

Jak si lze všimnout, tak v mé implementaci nejde jen o prosté překopírování hodnot, ale použil jsem nový parametr agenta nazvaný `tau`, který ovlivňuje podíl v rámci, kterého jsou překopírovány váhy z policy network do target network. Nastavením hodnoty $\texttt{tau} < 1$ lze docílit pomalejšímu přibližování target network k policy network. Naopak hodnotou 1 se docílí překopírování, tak jak bylo myšleno v původní práci Mnih, Kavukcuoglu, Silver, Rusu et al.

(2015). Tento nový parametr `tau` je definován v rámci konstruktoru třídy `DDQ_l`, a následuje v celkovém výčtu po parametrech z `IDeepQAgent` hned za dalším novým parametrem, který nastavuje počet učících kroků, které musí uběhnout pro překopírování vah mezi sítěmi nazvaný `update_interval_length`.

3.6.1 Učící algoritmus

Hlavní rozdíl agenta s touto architekturou je učící algoritmus, tedy implementace funkce `learn_from_experience`. Ten rozdíl oproti implementaci agenta `DQ_l` z kapitoly 3.5.1 je v nahrazení policy network v části počítající cílové hodnoty, proti kterým se policy network učí, za target network. Implementaci této funkce lze najít v kódu 3.14.

Tento kód se konkrétně liší od původní implementace s policy network pouze blokem `torch.no_grad`. V něm nejdříve provedeme dopředný průchod target network s hodnotami budoucích stavů z replay memory, který vrátí Q-hodnoty akcí pro tyto stavy. Z těchto Q-hodnot jsou vybrány ty s nejvyšší hodnotou pomocí funkce `max`. Tyto hodnoty jsou poté použity v Bellmanově rovnici, kdy jsou vynásobeny diskontním koeficientem `gamma` a sečteny s hodnotami odměn `rewards` z replay memory. Z těchto hodnot je dále vypočten `loss` úplně stejně jak v předchozí implementaci.

Na konci funkce `learn_from_experience` je ještě provedeno překopírování vah sítí. Nejdříve se ověří, zda uběhl počet učících kroků odpovídající parametru `update_interval_length`, a pokud ano, tak se zavolá funkce `copy_network_weights` popsaná dříve.

Kód 3.14: Učící mechanismus agenta s target network

```
def learn_from_experience(self):
    ...
    q_next_values = None
    with torch.no_grad():
        target_next_state_results = self.target_net(next_state_batch)
        max_next_state_q = target_next_state_results.max(1)[0]
        q_next_values = reward_batch + self.gamma
                        * max_next_state_q
    ...
    self.decrement_epsilon()
    self.steps_done += 1

    if self.steps_done % self.update_interval_length == 0:
        self.copy_network_weights()
```

3.6.2 Parametry agenta a jeho textová reprezentace

Implementace textové reprezentace agenta v rámci metody `__str__` stejně jako u agenta `DQ_l` nebude ukázána, protože se opět jedná o jednoduché formátování textu. Je ale důležité uvést pro přehlednost, jaké všechny parametry tento nový agent `DDQ_l` má. Doporučené hodnoty těchto parametrů lze najít v příloze Příloha 2: Tabulka doporučených parametrů pro agenta `DDQ_l`. Jejich výčet v pořadí, ve kterém vystupují v konstruktoru třídy `DDQ_l` a i v textové reprezentaci je zde:

1. learning rate neuronové sítě,
2. gamma,
3. velikost učících vzorků,
4. počáteční epsilon,
5. délka epsilon decay,
6. šířka první vrstvy neuron. sítě,
7. šířka druhé vrstvy neuron. sítě,
8. šířka třetí vrstvy neuronové sítě,
9. příznak, zda se používá RMSProp,
10. délka historie pozorování,
11. tau,
12. počet kroků pro překopírování vah,
13. lambda,
14. strategie eligibility traces

Tyto parametry odpovídají z většiny parametrům konstruktoru základní třídy `IDeepQAgent` ze sekce 3.4.3. Doporučené hodnoty těchto parametrů pro použití tohoto agenta lze nalézt v příloze B.

4. Hledání konfigurací agentů

Jedním z cílů této diplomové práce je zjištění, jakých výsledků oba agenti, z předchozí kapitoly 3, dosahují v rámci AIQ testu. Agentův výsledek ale většinou neurčuje pouze jeho vnitřní architektura, která udává přístup, jakým agent řeší jednotlivá prostředí posilovaného učení. Má-li agent nějaké hyperparametry neboli nastavitelné hodnoty, tak finální výsledek určuje i jak je zadaná konfigurace hyperparametrů dobrá. Agent může mít sebe lepší architekturu, ale pokud je špatně nastavená, může agent klidně zůstat i na nulovém výsledku. Pro zjištění nejlepšího možného výsledku agenta, je potřeba nalézt jeho nejlepší konfiguraci.

Výčet hyperparametrů obou agentů hlubokého Q-učení byl představen pro každého agenta v rámci jeho podkapitoly v předchozí kapitole. Pro připomenutí se jedná o 12 parametrů pro `DQ_l` agenta, jejichž výčet lze najít v sekci 3.5.2, a 14 parametrů pro `DDQ_l` agenta, jejichž seznam lze najít v sekci 3.6.2. Obor hodnot těchto parametrů je většinou spojitý, a teoreticky tak tyto parametry mohou mít nekonečně mnoho hodnot.

Jako výchozí bod hyperparametrů agentů mohou posloužit hodnoty z článku Mnih, Kavukcuoglu, Silver, Rusu et al. (2015), ve kterém byli agenti představeni. Pro agenty hlubokého Q-učení tyto parametry představují konfiguraci, která ale řeší problémy jiného testu. Konkrétně zpracování obrazu Atari her a jejich samotné hraní. Tento koncept samozřejmě pořád vychází z posilovaného učení, nicméně se dá očekávat, že konfigurace agentů musela být značně robustnější, než by byla konfigurace řešící problémy AIQ testu, který má jednodušší vyjádření stavů a obecně menší jejich prostor a jednodušší vazby mezi nimi.

Až bude nějaká nová konfigurace nalezena, tak bude jistě zajímavé ověřit, jak moc je konfigurace z článků horší než případná nalezená lepší konfigurace. Zatím nám ale toto nastavení hyperparametrů pomůže se od nějakých hodnot odrazit. Tyto původní konfigurace lze najít v Příloze A a Příloze B.

4.1 Systematické prohledání prostoru parametrů

Nejsnadněji proveditelné hledání možných konfigurací by bylo vytvoření nějakého výčtu hodnot, otestování agentů s těmito konfiguracemi a zvolení té s nejlepším výsledkem AIQ skóre. Například pro hodnotu γ (diskontního faktoru) používají autoři ve svém článku hodnotu 0,99 (Mnih, Kavukcuoglu, Silver, Rusu et al., 2015). Tato hodnota je svrchu omezená hodnotou jedna, tudíž bude zajímavé zjistit výsledky agentů pro hodnoty nižší. Vytvoří se tak například pět hodnot = 0,8; 0,6; 0,4; 0,2; 0,01. Agenti se s těmito hodnotami otestují a nalezne se tak aktuální nejlepší hodnota pro parametr γ. Interval byl poměrně hrubý, tudíž by v druhém kole chtělo otestovat parametr v okolí této aktuálně nejlepší hodnoty. To je tedy $5 + 5$ výpočtů pro dva agenty, tedy 20 otestování pro jeden parametr, a to je ještě velice hrubé.

Bylo by troufalé si myslet, že ale hodnoty parametrů jsou mezi sebou nezávislé, když jde o výsledné skóre agentů. Nelze proto hledat nejlepší hodnoty jednoho parametru za druhým samostatně, ale musí se provést testování všech možných kombinací hodnot všech hyperparametrů.

Jak zaznělo dříve v této kapitole, jednodušší agent hlubokého Q-učení má 12 parametrů, z toho 9 jich má spojitý obor hodnot a zbylé tři mají 2, 5 a 2 možné hodnoty. Pokud by se obory spojitých hyperparametrů navzorkovali nejhrubším schůdným způsobem do deseti hodnot, tak počet všech kombinací parametrů agenta bude $10^9 \cdot 2 \cdot 5 \cdot 2 = 2 \cdot 10^{10}$.

Otestování všech těchto kombinací hodnot a vybrání té nejlepší konfigurace bychom jako metodu pojmenovali „brute force hledání“. Tento přístup je ale výpočetně, a hlavně časově velmi velmi náročný. Dokázal by každopádně poskytnou bez debat tu nejlepší možnou konfiguraci obou agentů.

4.2 Implementace prohledávání prostoru hyperparametrů pomocí evolučních algoritmů

Evoluční algoritmy, které byly představeny již dříve v rámci kapitoly 2.4, poskytují přístup efektivnějšího prohledávání prostoru možných řešení, než jaký nabízí brute force přístup. Jako efektivnější je bráno, že bude procházeno daleko menší množství konfigurací, čímž se značně redukuje výpočetní náročnost celého hledání. Nalezení té opravdu nejlepší konfigurace, sice zabere pořád velké množství času, ale evoluční algoritmus se postupně, v rámci jednotlivých generací, víc a víc přibližuje této nejlepší konfiguraci. Je tak možné hledání ukončit v průběhu, a získat tak „dostatečně“ dobrou konfiguraci, jako aktuálně tu nejlepší nalezenou. Vlastnost evolučních algoritmů je zároveň ta, že výsledek rychle zkonverguje k této dostatečně dobré konfiguraci, a následné inkrementální zlepšení mezi generacemi postupně klesá.

Jelikož tento typ hledání konfigurací lze aplikovat na libovolného agenta AIQ testu, který má nějaké hyperparametry, tak byl samotný test rozšířen o evoluční prohledávání parametrů. Z pohledu struktury aplikace byl do kořenového adresáře testu přidán nový adresář `genetics`, který má následující obsah:

- `genetics/`
  - `agents_ref/` - obsahuje nastavení pro generování hyperparametrů pro jednotlivé agenty AIQ testu
  - `context/` - obsahuje datové struktury v rámci, kterých se předává nastavení jednotlivých částí prohledávání
  - `environment/` - obsahuje kódy jednotlivých sekcí evolučních algoritmů jako mutace, výběr přeživších atd.
  - `Environment.py` - obsahuje hlavní třídu, jejíž úlohou je správa chodu testu, uchovávání informací o generacích a propojování jednotlivý sekcí kódu z adresáře `environment/`
  - `eval_function.py` - obsahuje kód vyhodnocující skóre jednotlivých konfigurací, a tedy spouští pro ně AIQ test a zpracovává jeho výsledek
  - `Individual.py` - třída, jejíž instance zaobalující jednotlivou konfiguraci a její výsledek pro následné znovu použití bez nutnosti konfiguraci opětovaně vyhodnocovat

Mimo celý tento adresář `genetics` byl v rámci testu přidán do jeho kořenového adresáře Python skript, přes který se celé vyhledávání pomocí evolučních algoritmů spouští - `FindParamsGenetics.py`. Tento skript se musí spustit opět z terminálu jako samotný AIQ test, a toto volání má následující podobu:

```
1 python FindParamsGenetic.py <parametry>
```

Možné parametry pro spouštění skriptu popisuje tabulka v rámci Přílohy D.

V průběhu hledání jsou na konci každé generace zapsány do výsledkového souboru všechny konfigurace z aktuální generace a jejich výsledky. Zároveň jsou do standardního výstupu

zapisovány konfigurace a skóre přeživších, kteří budou tvořit generaci novou. Do standardního výstupu lze zapisovat i jednotlivé kroky algoritmu, pokud je zapnut příznak `debug`. Po vyhodnocení poslední generace je vypsána do standardního výstupu nejlepší nalezená konfigurace a její skóre.

4.2.1 Implementace hlavního cyklu

Hlavní část mé implementace evolučního algoritmu je třída `Environment` ze souboru `Environement.py`. Tato třída je vytvořena v rámci skriptu `FindParamsGenetics.py`, jehož úkolem je načíst parametry a nastavení od uživatele a předat je v rámci kontextu instanci `Environmentu`. Na `Environmentu` je dále zavolána funkce `simulate` která na základě předaných nastavení spustí vyhledávání přes evoluční algoritmy, a celé vyhledávání řídí.

Implementaci této řídící logiky v rámci funkce `simulate` lze vidět v Kódu 4.1. Z této ukázky kódu byly pro přehlednost odstraněny všechny výpisy do standardního výstupu v rámci funkcí `print`, `self.debug_print` a `self.log_results`, a jejich přidružené funkce. Funkce začíná tak, že se vytvoří úvodní populace v rámci funkce `initialize_population` modulu `environment`. Populace je vlastně pole instancí třídy `Individual`, která uchovává parametry agenta v podobě konfigurace, a dále výsledek AIQ pro tuto konfigurace, pokud už byl tento jednotlivec vyhodnocen. Dále se připraví proměnná, která bude uchovávat přeživší mezi generacemi `best_individuals`.

Na řádku 5 již začíná hlavní smyčka prohledávání. Počet iterací, které se v rámci prohledávání provedou, odpovídá počtu epoch z parametrů předaných v rámci volání hlavního skriptu. Prvním krokem této smyčky je vyhodnocení aktuální populace v rámci funkce `evaluate_population`. Ta bude vysvětlena dále, ale zatím stačí vědět, že tato funkce otestuje všechny neotestované konfigurace na AIQ testu, a vrátí populaci s uloženými hodnotami skóre. Pro jednoduchost další práce vrací tato funkce i pole skóre pro konfigurace na shodných indexech. Z těchto skóre se vybere n-nejlepších, kteří jsou vypsáni v rámci vynechaného debugu, a zároveň jsou uloženy. Pokud by aktuální epocha byla poslední, tak se v rámci řádků 12 a 13 smyčka ukončí.

Důležitější krok pracující s vyhodnocenou populací se odehrává v rámci řádku 14, kde se volá funkce `select_parents`. Ta na základě nastavené strategie zavolá příslušnou funkci modulu `environment`, která z následující generace vybere přeživší pro tvorbu nové generace. Z toho důvodu je použito označení `parents`. Možné strategie pro tento krok jsou: ruletový výběr, výběr podle pořadí a nebo turnajový výběr. Výsledná proměnná `parents` pořád odpovídá poli referencím na již vyhodnocené konfigurace v rámci instancí třídy `Individual`, tudíž nebude potřeba tyto konfigurace znovu testovat.

Hodnota proměnné `population` je vymazána na řádku 15, a novou populaci budou tvořit pouze přeživší z proměnné `parents`, jak lze vidět na řádku 16. Z této kolekce konfigurací se pak vytvoří noví jedinci, tak aby počet jedinců v populaci odpovídal nastavené hodnotě. To se stane díky funkci `create_new_generation` modulu `environment`, v rámci které se provede

křížení a následné mutace jedinců. Vše je tak nakonec připraveno na další iteraci celé této hlavní smyčky.

Kód 4.1: Hlavní smyčka evolučního algoritmu

```
def simulate(self):
  population = environment.initialize_population(
    self, self.context.population, self.seed_population)
  best_individuals = None
  for i in range(self.context.epochs):
      population, scores = environment
        .evaluate_population(self, population)
      best_individuals_indices = select_top_scores(
        scores, self.context.count_select
      )
      best_individuals = [population[i] for i in best_individuals_indices]
      if i == self.context.epochs - 1:
          break
      parents = self.select_parents(population, scores)
      population.clear()
      population.extend(parents)
      population = environment.create_new_generation(self, population)
  value_list = [
    best_individuals[i].eval() for i in range(len(best_individuals))
  ]
  best_index = np.argmax(np.array(value_list))
  best_individual = best_individuals[best_index]
```

4.2.2 Implementace konfigurací a jejich generování

Každé nové hledání konfigurací začíná zavoláním funkce `initialize_population`. Jejím cílem je vytvoření úvodní generace konfigurací. K pochopení této funkce je nejdříve potřeba představit, jakým způsobem jsou vyjádřeny jednotlivé konfigurace agentů, a jakým způsobem jsou generovány.

Každý agent AIQ testu, pro kterého je potřeba umožnit prohledávání jeho parametrů v rámci evolučních algoritmů, musí mít vytvořenou párovací třídu v rámci adresáře `genetics/agents_ref`. Tato párovací třída propojuje agenta z AIQ testu s prohledávacím procesem. Obsahuje přesnou referenci na konkrétního AIQ agenta, a zároveň obsahuje generátory jeho parametrů. Název této třídy je poté používán v rámci parametru `-a` při spouštění hlavního skriptu vyhledávání. Tento název ale teoreticky nemusí nutně odpovídat názvu agenta v AIQ testu. Může se totiž stát, že pro jednoho agenta bude potřeba vytvořit vícero možností generátorů, a tak může existovat vícero tříd jednoho agenta s různými názvy. Důležitá je vždy explicitní reference na agenta v těle třídy.

Ukázka takového generátoru pro AIQ agenta `DDQ_l` lze vidět v Kódu 4.2. Každý generátor musí dědit ze základní třídy `AgentReference`, která udává základní funkcionalitu. Dále musí konkrétní reference implementovat metody `get_test_agent_name` a `get_generators`. Úlohou `get_test_agent_name` je vracet přesný název třídy AIQ agenta, ke kterému se tato reference vztahuje. `get_generators` poté musí vracet pole funkcí, které pro jednotlivé parametry AIQ agenta generují jejich platné hodnoty. Tyto funkce musí být v rámci pole na stejné pozici jako jsou přidružené parametry v rámci konstruktoru AIQ agenta.

V mé implementaci pro agenta `DDQ_l` jsem se rozhodl použít pro generování hodně spojitých hyperparametrů, jako je learning rate, γ nebo později τ, kaskádu funkcí knihovny NumPy, která diskretizuje interval mezi dvěma hodnotami pomocí funkce `linspace` a poté z tohoto intervalu vybere náhodnou hodnotu přes `random.choice`. Pro learning rate, v podobě hned prvního parametru, jsem například diskretizoval interval $(0.00001, 0.01)$ na dvě stě mezikroků, ze kterých se vybírá jeden náhodný. Jednodušší parametry a jejich řady hodnot jsem poté definoval přes Python funkci `randint` modulu `random`.

Kód 4.2: Reference evolučního algoritmu na agenta DDQ_l

```
class DDQ_l(AgentReference):
  def get_test_agent_name(self):
      return "DDQ_l"

  def get_generators(self):
      use_eligiblity_traces = random.random() > 0.95
      return [
          # Learning rate [0.01, 0.00001]
          lambda: np.random.choice(np.linspace(0.01, 0.00001, 200)),
          # Gamma value [0.01, 0.99]
          lambda: np.random.choice(np.linspace(0.01, 0.99, 100)),
          # Batch size in size of 2^n for n [2, 9] generating numbers
          # 4, 8, 16, 32, 64, 128, 256, 512
          lambda: 2 ** (random.randint(2, 9)),
          # Epsilon
          lambda: 1,
          # Epsilon decay [100, 10000]
          lambda: random.randint(1, 100) * 100,
          # Size of neural net layer 1 [16, 256]
          lambda: random.randint(1, 16) * 16,
          ...
      ]
```

Třídě `Environment` je při jejím vytvoření předána i instance na agentovu referenci, a proto při volání funkce `initialize_population` je jednoduché se dostat k jednotlivým generátorům agentových parametrů. Kód této inicializační funkce lze vidět v Kódu 4.3. Pokud je dostupná seed populace, tak je nejdříve nečtena ta (seed populace označuje výchozí generaci načtenou

ze souboru), a až poté podle toho kolik konfigurací ještě chybí do plné generace se vytvoří nová individua v rámci cyklu na řádcích 4 až 10. Jednoduché vytvoření nové konfigurace na řádku 10 je možné díky metodě základní třídy `AgentReference`, která obsahuje metodu `generate_param`, jejíž implementaci lze vidět v Kódu 4.4. V podstatě se jen vezme definované pole generátorů parametrů, a postupnou iterací se tyto generátory zavolají a výsledky se přidají do pole.

Kód 4.3: Tvorba úvodní generace evolučního algoritmu

```
1  def initialize_population(environment, size: int, seed_population: list):
2    population = get_valid_from_seed(environment, seed_population)
3    num_generate = 0 if len(population) >= size else size-len(population)
4    for i in range(num_generate):
5        genome = environment.agent_ref.generate_params()
6        population.append(Individual(
7            genome,
8            environment.agent_ref.get_test_agent_name(),
9            environment.context
10       ))
11   return population
```

Kód 4.4: Generování parametrů agenta

```
1  def generate_params(self):
2    generator = self.get_generators()
3    params = list()
4    for g in generator:
5        params.append(g())
6    return params
```

4.2.3 Implementace evaluace skóre konfigurací

Hlavní smyčka evolučního prohledávání z Kódu 4.1 začíná vyhodnocením aktuální generace. Vyhodnocení agentů v závislosti na jejich komplexnosti může být časově a výpočetně velmi náročné. Při implementaci vyhodnocování celé generace jsem chtěl tento proces co nejvíce zrychlit. Z toho důvodu již otestované konfigurace v rámci instancí třídy `Individual` si pamatují své výsledky, a tudíž pokud se v generaci nachází nezměněný jedinec z předchozí generace, tak ho není třeba podruhé vyhodnocovat.

Pokud ale přijde na vyhodnocení nových konfigurace, tak jsem se nechtěl spoléhat na časovou závislost, kterou má AIQ test při vyhodnocování agenta s rostoucím počtem výpočetních jader. AIQ totiž pracuje s jistou mírou paralelizace, každopádně má myšlenka byla, že by mohlo být výpočetně a časově efektivnější zapojit paralelizaci i do vyhodnocování generací, a tedy, že bude rychlejší otestovat c-agentů zároveň na AIQ testech využívajících t-jader, než testovat sekvenčně jednoho agenta na AIQ testu s $c \cdot t$ jádry. Oba parametry jsou každopádně

nastavitelné při spouštění vyhledávání, a tudíž kdyby se tato myšlenka ukázala jako nesprávná, tak nastavením parametru `-c` na hodnotu 1, bude testování probíhat sekvenčně.

Implementace takového vyhodnocování lze vidět v Kódu 4.5. Nejdříve je v rámci řádku 3 zjištěno na kolik částí se bude muset paralelní vyhodnocování rozdělit. Dále jsou vytvořeny fronty pro neotestovaná a otestovaná individua. Při souběžné práci více procesů je potřeba ohlídat možnosti jejich uváznutí nebo souběhů, a právě použití front místo polí zajistí synchronizaci toku dat. Fronta `population_queue` je tak naplněna individui z generace, a jednotlivé paralelní procesy si na začátku své existence z ní vytáhnou jednu konfiguraci, která je zároveň touto operací odstraněna z fronty.

Kód 4.5: Vyhnocování aktuální populace konfigurací

```
1  def evaluate_population(environment: Environment, population: List[Type[
       Individual]]):
2    allowed_threads = environment.context.num_agents
3    iterations = math.floor(environment.context.population / allowed_threads)
4    population_queue = mp.Queue()
5    evaluated_queue = mp.Queue()
6    for individual in population:
7        population_queue.put(individual)
8    for i in range(iterations):
9        processes = list()
10       for t in range(allowed_threads):
11           processes.append(EvalProcess(evaluated_queue, population_queue))
12       for process in processes:
13           process.start()
14       for process in processes:
15           process.join()
16   processes = list()
17   num_rest = environment.context.population - (iterations * allowed_threads)
18   # ... vyhodnoceni zbytku podle num_rest ...
19   pop = list()
20   values = list()
21   for i in range(environment.context.population):
22       individual = evaluated_queue.get()
23       pop.append(individual)
24       values.append(individual.eval())
25
26   return pop, values
```

Činnost jednotlivých vláken odpovídá definici z třídy `EvalProcess`, která přijímá jako parametry konstruktoru obě fronty. Z první fronty si získá nezpracované individuum, na kterém zavolá funkci `individual.eval()`. Tato funkce vrátí již uložený výsledek, pokud existuje, a nebo zavolá takzvanou evaluační funkci která konfiguraci otestuje. Každá instance třídy

`Individual` obsahuje referenci na tuto funkci, a dokáže ji tak zavolat s vlastní hodnotou konfigurace a dalším nastavením testu z předávaného kontextu.

Úlohou evaluační funkce je v podstatě spuštění AIQ testu s konfigurací daného individua a uložení jejího skóre. Mou implementaci této funkce lze vidět v Kódu 4.6. Princip evaluační funkce je ten, že si každý vyhodnocovací proces otevře vlastní instanci terminálu aktuálního operačního systému, a v rámci tohoto terminálu spustí AIQ test, jehož výstup čte a zpracovává. Na řádku 2 se tak volá lokální funkce `build_script_string`, která připraví příkaz na spuštění AIQ testu s aktuální konfigurací. Následující řádek 3 odvodí polohu AIQ skriptu relativně od polohy skriptu evaluační funkce. Knihovna `subprocess` umožní spustit AIQ test přes terminál, což je potvrzeno zavoláním funkce `communicate` na vzniklém objektu terminálového procesu `process`.

Po otestování skrz AIQ test, které může klidně trvat hodiny, se do proměnných `stdout` a `stderr` zapíše celý výstup průběhu testu a jeho chybový výstup. Ten je dále převeden na text a zpracován funkcí `parse_output`, která z textového výstupu testu dokáže vyčíst výsledek. Ten je v podobě střední hodnoty AIQ skóre, které ignoruje intervaly spolehlivosti, které jsou součástí výstupu AIQ testu. Ideálně by se totiž hledání parametrů mělo provádět s nastaveným velkým množstvím vzorků programů, čímž by se tato hodnota nejistoty blížila k nule. Pokud náhodou dojde v testu k fatální chybě, tak výsledek takové konfigurace bude největší možné záporné číslo, čímž je konfigurace více méně diskreditována z dalšího zpracování. Tento výsledek je poté předán zpět do `EvalProcess` a společně s konfigurací je zapsán do výstupní fronty paralelního zpracování, ze které se ve finále sestaví nová populace.

Kód 4.6: Evaluační funkce jednoho individua

```
1  def eval_function(genome, agent_name, test_settings: TestSettings):
2    script_string = "python " + build_script_string(genome, agent_name,
         test_settings)
3    script_path = os.getcwd().split("genetics")[0]
4    args = script_string.split(" ")
5    process = subprocess.Popen(args, cwd=script_path, stdout=subprocess.PIPE,
         stderr=subprocess.PIPE)
6    stdout, stderr = process.communicate()
7
8    output = stdout.decode("utf-8")
9    err = stderr.decode("utf-8")
10   if err:
11       print("Evaluation errors:" + err)
12   parsed_value = parse_output(output)
13   return parsed_value
```

4.2.4 Implementace výběru přeživších pro tvorbu další generace

Po vyhodnocení celé populace přichází na řadu výběr přeživších, kteří se budou podílet na tvorbě nové generace. Z kapitoly 2.4 víme, že přístupů k výběru přeživších je hned několik. Aktuální podoba evolučních algoritmů přidaných v rámci AIQ testu implementuje tři přístupy: ruletový výběr, výběr podle pořadí a turnajový výběr. Která strategie pro výběr bude použita určuje hodnota příznaku `-r` hlavního skriptu prohledávání. Jednotlivé implementace těchto strategií se nachází v souboru `select_parents.py` modulu `environment`.

Ruletový výběr je metoda, kdy pravděpodobnost výběru individua závisí na jeho skóre proti sumě všech skóre v populaci. Lze si to představit tak, že všechna individua umístíme na kruh představující ruletu, a každá jedna konfigurace bude zabírat tolik prostoru, jako je její pravděpodobnost výběru. Poté se zvolí náhodná hodnota a postupně se sčítají pravděpodobnosti jednotlivých individuí, dokud se nepřekročí ona náhodná hodnota, a vybere se to individuum, u kterého se toto stalo.

Implementaci takové metody lze vidět v Kódu 4.7. V této implementaci se nejdříve aplikuje tzv. "okénkovací metoda"(Eiben a Smith, 2015) v rámci funkce `apply_windowing`, která v podstatě přičte ke všem skóre hodnotu jedna a odečte hodnotu nejnižšího skóre. Tím se lehce narovná distribuce výběru jednotlivých prvků. Dále se zjistí suma těchto skóre, a připraví se pole `roulette_wheel`, které obsahuje uspořádané trojice, které obsahují počáteční bod v ruletě, koncový bod v ruletě a referenci na individuum. Tato ruleta je poté předána funkci `create_from_roulette_wheel`, která vybere z rulety tolik náhodných hodnot, kolika se rovná hodnota proměnné `parents_count`.

Kód 4.7: Ruletový výběr

```
1  def roulette_wheel_fitness_selection(
2      old_population: List[Type[Individual]],
3      scores: List[float],
4      parents_count: int):
5    windowed_scores = apply_windowing(scores)
6    scores_sum = get_sum_of_scores(windowed_scores)
7    roulette_wheel = list()
8    current_prob = 0
9    for i in range(len(old_population)):
10       prob = windowed_scores[i] / scores_sum
11       roulette_wheel.append((current_prob, current_prob + prob, old_population
             [i]))
12       current_prob += prob
13   return create_from_roulette_wheel(roulette_wheel, parents_count)
```

Implementace `create_from_roulette_wheel` je poměrně jednoduchá, jak lze vidět v Kódu 4.8. Vytvoří se pole přeživších `parents`, které dokud nemá tolik prvků, kolik udává parametr `parents_count`, tak generuje náhodné hodnoty, a hledá k nim takový záznam, který danou

hodnotu obsahuje mezi svými hranicemi `low` a `high` z uspořádané trojice.

Kód 4.8: Výběr přeživších podle rulety

```
def create_from_roulette_wheel(roulette_wheel: List[Tuple[float, float, Type[
    Individual]]], parents_count):
  parents = list()
  rand_generator = random.Random()
  while len(parents) < parents_count:
      rand = rand_generator.random()
      for low, high, individual in roulette_wheel:
          if low <= rand < high:
              parents.append(individual)
              break
  return parents
```

Výběr podle pořadí je metoda, velice podobná ruletovému výběru, protože také pracuje s principem, že se jednotlivá individua seřadí do rulety. Rozdíl ale je ten, že pravděpodobnost výběru není daná hodnotou skóre, ale spíš jeho pořadím. Je definován parametr s, jehož hodnotou lze určovat proporce úseků jednotlivých individuí. Pravděpodobnost výběru je podle Eiben a Smith (2015) dána vzorcem:

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)} \tag{4.1}$$

p_i je pravděpodobnost výběru hodnoty s indexem i, s je onen hyperparametr z rozsahu $(1 < s < 2)$ a μ je počet jednotlivců v ruletě. Implementace takové metody je ukázána v Kódu 4.9.

Kód 4.9: Výběr podle pořadí

```
def roulette_wheel_rank_selection(
    old_population: List[Type[Individual]],
    scores: List[float],
    parents_count: int):
  s_param = 1.5
  sorted_indices = sorted(range(len(scores)), key=lambda k: scores[k])
  sorted_population = [old_population[i] for i in sorted_indices]
  roulette_wheel = list()
  current_prob = 0
  for i in range(len(sorted_population)):
      prob = ((2 - s_param) / len(sorted_population))
            + (2 * i * (s_param - 1))
            / (len(sorted_population) * (len(sorted_population) - 1))
      roulette_wheel.append((current_prob, current_prob + prob,
          sorted_population[i]))
      current_prob += prob
  return create_from_roulette_wheel(roulette_wheel, parents_count)
```

Poslední metoda výběru přeživších s názvem **turnajový výběr** už nestojí na principu rulety. Metodika je taková, že jsou vybírány postupně dvojice individuí, a do další generace je vybráno to s vyšším výsledným skóre. Do takového souboje může být vybrán klidně dvakrát ten samý jedinec, čímž se zachovává možnost, že přeživší může být libovolný jednotlivec (pokud bude vybrán dvakrát ten nejhorší z populace).

Implementaci takové strategie výběru lze vidět na Kódu 4.10. Tato strategie má také hyperparametr určující počet účastníků jednotlivých turnajů v podobě proměnné `count_competitors`, který je aktuálně nastaven na hodnotu 2. Opět je definováno prazdé pole, které se plní dokud jeho velikost neodpovídá hodnotě parametru `parents_count`. V rámci plnící smyčky jsou na řádku 11 vybrány náhodné indexy účastníků. Funkce `choices` provádí výběr s opakováním. Z účastníků je na řádku 14 vybrán ten s nejvyšším skóre, a ten je přidán mezi přeživší.

Kód 4.10: Turnajový výběr

```
1  def tournament_selection(
2      old_population: List[Type[Individual]],
3      scores: List[float],
4      parents_count: int):
5    count_competitors = 2
6    if len(old_population) < (count_competitors * 2):
7        raise Exception(f"Population is too small...")
8    parents = list()
9    rand_generator = random.Random()
10   while len(parents) < parents_count:
11       competitors_indices = rand_generator.choices(
12         range(len(old_population)), k=count_competitors
13       )
14       best_competitor_index = max(
15         competitors_indices, key=lambda index: scores[index]
16       )
17       best_competitor = old_population[best_competitor_index]
18       parents.append(best_competitor)
19   return parents
```

4.2.5 Implementace mutací a křížení

Když bylo vysvětleno jak se dostat až do bodu, kdy jsou pro aktuální generaci vybráni přeživší, ze kterých se bude tvořit nová generace, tak nezbývá než ukázat, jak se z nich nová generace vytvoří. Metoda tvořící novou generaci nejdříve doplní k přeživším jejich zmutované obrazy pomocí funkce `mutate`. Její kód lze vidět v Kódu 4.11.

Myšlenkou mutování je vzít již existující konfiguraci, lehce ji pozměnit, a otestovat zda změna vedla k lepšímu výsledku nebo horšímu. V kontextu prohledávání `context` je metodě předán

počet, kolik individuí je potřeba dovytvořit mutacemi parametrem `num_mutations`. Podle toho se provede cyklus na řádcích 4 až 19, který nejdříve vybere náhodnou konfiguraci z populace a vytvoří její kopii, aby jí mohl upravovat nezávisle na originálu. Dále se prochází přes všechny její hyperparametry a s pravděpodobností rovné hodnotě proměnné `mutation_prob` se hodnota na aktuálním indexu nahradí novou hodnotou z generátoru daného hyperparametru. Po průchodu přes všechny parametry konfigurace, se nová vzniklá konfigurace zabalí do nové instance třídy `Individual` a přidá se do aktuální generace.

Kód 4.11: Mutace konfigurací

```
1  def mutate(population: List[Type[Individual]], agent_ref: AgentReference,
       context: Context):
2    mutated_gens = list()
3    rand_gen = random.Random()
4    for i in range(context.num_mutations):
5        chromosome_to_mutate = copy.deepcopy(
6          random.choice(population).get_genome()
7        )
8
9        indices = [i for i in range(len(agent_ref.get_generators()))]
10       for i in indices:
11           rand_val = rand_gen.random()
12           if rand_val <= context.mutation_prob:
13               chromosome_to_mutate[i] = agent_ref.get_generators()[i]()
14
15       mutated_gens.append(Individual(
16           chromosome_to_mutate,
17           agent_ref.get_test_agent_name(),
18           context)
19       )
20   return mutated_gens
```

Křížení je proces, kdy se vezmou konfigurace dvou a více přeživších a promícháním jejich genetické informace se zkusí přijít na novou lepší konfiguraci. Funkce, která tento mechanismus implementuje, se nazývá `crossover` a její kód lze vidět v Kódu 4.12. V rámci kontextu se opět předává počet kolik potomků se má vytvořit v rámci proměnné `num_crossover`. Dominantní sekcí implementace je opět cyklus, který tvoří nové potomky dokud jejich počet nedosáhne právě hodnoty `num_crossover`.

Na řádku 8 lze vidět, že z přeživších jsou vybráni dva rodičové. Tentokrát se používá funkce `sample`, která vybírá bez opakování, aby byla zajištěna unikátnost rodičů. Na řádku 13 se vyberou dva body v rámci pole hyperparametrů, mezi kterými dojde k prohození hodnot, které lze označit jako A a B. První potomek vznikne tak, že bude obsahovat od indexu 0 až po bod A hodnoty parametrů prvního rodiče, od A do B hodnoty druhého rodiče, a od B do konce opět hodnoty prvního rodiče. Druhý potom vznikne přesně inverzně. Oba potomci

jsou zařazeni do populace, a cyklus se opakuje, dokud je potřeba přidávat další jedince do populace, ale tentokrát s jinými rodiči.

Kód 4.12: Křížení konfigurací

```
def crossover(
    best_individuals: List[Type[Individual]],
    agent_ref: AgentReference,
    context: Context):
  children = list()
  rand_gen = random.Random()
  while len(children) < context.num_crossover:
      parents_idxs = rand_gen.sample(range(len(best_individuals)), 2)
      p1 = best_individuals[parents_idxs[0]]
      p2 = best_individuals[parents_idxs[1]]
      p1_chromosome = p1.get_genome()
      p2_chromosome = p2.get_genome()
      crossover_points = sorted(rand_gen.sample(range(len(p1_chromosome)), 2))
      start, end = crossover_points
      child1 = p1_chromosome[:start] + p2_chromosome[start:end] +
          p1_chromosome[end:]
      child2 = p2_chromosome[:start] + p1_chromosome[start:end] +
          p1_chromosome[end:]

      children.append(Individual(
          child1,
          agent_ref.get_test_agent_name(),
          context
      ))
      children.append(Individual(
          child2,
          agent_ref.get_test_agent_name(),
          context
      ))
  return children[:context.num_crossover]
```

4.3 Nastavení evolučního algoritmu

Nastavení hyperparametrů evolučního algoritmu má vliv na rychlost a spolehlivost prohledávání prostoru parametrů. Nicméně hledání těch nejlepších možných hodnot vede na stejný problém jako u hledání hodnot hyperparametrů agentů AIQ testu, tedy prostor je rozsáhlý a jeho prohledání časově náročné. Nezávisle na nastavení by ale měl každý evoluční algoritmus splňující podmínky z kapitoly 2.4 postupně dokonvergovat k nejlepšímu řešení. Z toho důvodu beru hledání nejlepší možné konfigurace evolučního algoritmu jako časově nevýhodnou úlohu, a samotné nastavení pro potřeby AIQ testu odvodím z obecných pouček, které zmiňuje například Eiben a Smith (2015). Jako takovou hlavní poučku, kterou zmiňuje, beru to, že pokud máme omezené časové možnosti, a chceme nalézt nejlepší možnou konfiguraci v co nejkratším čase, tak je lepší se zaměřit na větší velikost generace než na samotný počet iterací, jelikož samotné zlepšení výsledků mezi generacemi klesá, a největší zlepšení probíhají v prvních generacích.

Výsledné nastavení hyperparametrů prohledávání evolučním algoritmem, která jsem použil já, lze vidět v tabulce 4.1. Velikost generace jsem zvolil 30, s myšlenkou, že v takto velké skupině by mohlo být zastoupeno dostatečně velké množství vzájemně rozdílných konfigurací, a zároveň by vyhodnocení jedné generace nemuselo být zas tak výpočetně náročné. Z těch 30 se po vyhodnocení vybere 15 individuí pro tvorbu další generace. S touto velikostí by se nemělo stávat, že se nejlepší konfigurace nevybere jako přeživší, a zároveň zde není tak velký prostor, aby mezi přeživšími převládali případné duplikáty.

Test byl spuštěn úvodně na 25 iterací. Má implementace umožňuje vytvořit nultou generaci ze souboru, a tudíž lze kdykoliv na vyhledávání navázat v druhém běhu, pokud se jako nultá (seed) generace použije poslední generaci z prvního běhu.

Důležité jsou dále parametry nastavující vnitřní chování testu. V rámci nich jsem nastavil pravděpodobnost mutací na hodnotu 20 %. Jelikož agenti mají jednotlivě 12 a 14 parametrů, tak nastavení, které by zaručovalo průměrně zmutovaní jednoho parametru, by odpovídalo zhruba 8 % a 7 %. Můj důvod pro zvolení vyšší hodnoty, byla nejistota ohledně časové náročnosti hledání, a chtěl jsem tak pokrýt co nejvíc prostoru parametrů agentů v co nejkratším čase. Tato hodnota zaručí, že v průměru budou zmutovány tři parametry agenta, což by ještě nemělo být tak agresivní, aby prohledávání nedokázalo správně konvergovat.

Další parametr vnitřního chování je strategie výběru přeživších. Já se rozhodl použít turnajový výběr, který spočívá v tom, že vybere dva jedince z populace, a jako přeživšího vybere toho s vyšším skóre. Tato metoda mi přišla, že by mohla zaručit co nejunikátnější složení skupiny přeživších, a tedy i opět co nejrychlejší pokrytí prostoru možných hodnot.

Nastavení AIQ testu, na 25000 iterací pro sample programů o velikosti 1000, bych označil jako výkonnostní kompromis. Sample velikosti 1000 dává standardní chybu odhadu AIQ skóre kolem 1.5, což je taková nejhrubší hodnota, se kterou lze mezi sebou výsledky agentů odlišovat. Počet iterací je komplikovanější rozhodnutí. Finálně jsou totiž zajímavé výsledky agenta až na 100000 iterací, nicméně srovnáním výsledků agentů na 50000. kroku lze dostat

takřka stejné pořadí jako by bylo na 100000. kroku. Test na 50000 kroků je ale pořád dost výpočetně náročný, tak jsem se rozhodl použít hodnotu ještě nižší. Výsledky kolem 10000. kroku jsou nevhodné, protože existují agenti a jejich konfigurace, které sice mají nižší skóre na 10000. iterace nicméně ve finále zvládnou dokonvergovat k lepším výsledkům. Mé použití 25000 iterací, tak beru jako kompromis mezi náročností a podceněním pomaleji konvergujících agentů.

Parametr	Hodnota	Význam parametru
`-c`	15	počet přeživších tvořící další generaci
`-e`	25	počet epoch prohledávání
`-i`	25000	počet iterací AIQ testu
`-m`	0,2	pravděpodobnost mutací
`-n`	8	počet souběžně testovaných agentů
`-p`	30	celková velikost jedné populace
`-r`	2	strategie výběru = turnajový výběr
`-s`	1000	počet vzorků programů AIQ testu
`-t`	8	počet jader AIQ testu

Tabulka 4.1: Parametry evolučního algoritmu

4.3.1 Rozsahy prohledávaných hyperparametrů agentů

Správně zvolené přípustné hodnoty hyperparametrů agentů jsou velmi důležité. Pokud se zvolí rozsah hodnot špatně, výsledky budou špatné. Pokud se zvolí příliš řídké rozdělení, tak nebude možné nalézt nejlepší konfiguraci, a pokud naopak bude rozdělení příliš husté, tak vyhledávání bude časově velmi náročné. Obecně je ale lepší zvolit hustější rozdělení než řidší, protože je důležitější obsáhnout možnou nejlepší hodnotu daného parametru než pracovat pouze s "nejbližší možnou"hodnotou (Eiben a Smith, 2015).

Tabulka 4.2 zachytává rozsahy použité pro parametry agentů hlubokého Q-učení. Některé hodnoty jako je gamma nebo tau mají jednoduché obory hodnot, protože se v podstatě jedná o procentuální údaj, a tak může nabývat hodnot od 0 od 1 (potažmo od 0 % po 100 %). Ostatní parametry pak v zásadě vychází z okolí hodnot, které byly použity v rámci článku (Mnih, Kavukcuoglu, Silver, Rusu et al., 2015), který obsahuje popis agentů hlubokého Q-učení. Tyto výchozí hodnoty lze najít v rámci příloh A a B.

4.4 Výsledky hledání konfigurací

Spuštění evolučního algoritmu a hledání nejlepších konfigurací bylo provedeno na výpočetním centru Metacetrum. Zde byly zadány dvě úlohy, pro každého agenta jedna, odpovídající nastavení hledání z tabulky 4.1. Jednotlivé úlohy byly zadány jako výpočty trvající 14 dní s využitím 64 CPU a 128 GB RAM pamětí.

Parametr	Min	Max	DQ_l	DDQ_l	Poznámka
Learning rate	0,00001	0,01	✓	✓	
Gamma	0,01	0,99	✓	✓	
Velikost učícího batche	4	512	✓	✓	Hodnoty byly generovány jako mocniny dvou.
Epsilon	1	1	✓	✓	Byly hledány konfigurace s epsilon decay.
Délka epsilon decay	100	10000	✓	✓	Hodnoty po stovkách.
První vrstva neuronové sítě	16	256	✓	✓	Násobky 16.
Druhá vrstva neuronové sítě	16	256	✓	✓	Násobky 16.
Třetí vrstva neuronové sítě	0	256	✓	✓	S pravděpodobností 35 % byla třetí vrstva vynechána.
Gradientní algoritmus	0	1	✓	✓	0 = RMSProp, 1 = ADAM
Délka historie	0	4	✓	✓	
Tau	0,01	1		✓	
Délka intervalu učení druhé sítě	5	500		✓	
Lambda	0,01	1	✓	✓	
Strategie eligibility traces	0	2	✓	✓	

Tabulka 4.2: Rozsahy parametrů agentů pro jejich prohledávání

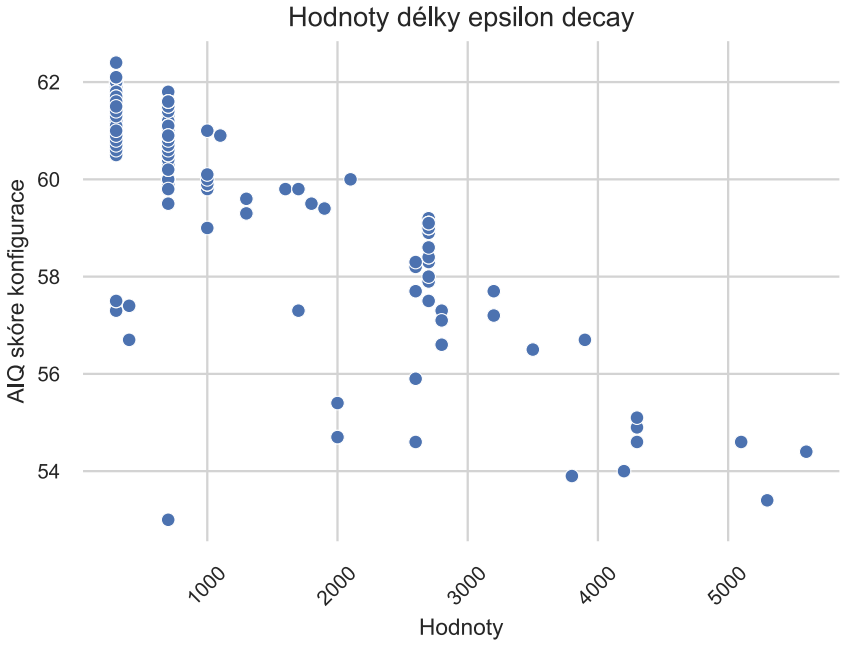
První z testů, konkrétně ten pro `DQ_l` agenta, se dostal na řadu pro výpočet zhruba po třech týdnech čekání ve frontě, a tedy výsledky byly dostupné po 5 týdnech od zadání výpočtu. Druhý výpočet čekal ve frontě necelých 9 týdnů, a výsledek tak byl získán po 11 týdnech od zadání výpočtu do systému. Dá se předpokládat, že důvod proč tomu tak bylo, je potřeba 64 procesorů, aby se za 14 dní stihlo vypočítat zadaných 25 generací, a na přidělení takové výkonu se bohužel čeká v dlouhé frontě. I tak se z výsledků ukázalo, že s tímto výkonem se za 14 dní stihlo projít pouze 21 generací pro `DQ_l` agenta a 22 generací pro `DDQ_l` agenta oproti očekávaným 25.

Z tohoto důvodu už prohledávání nebudu replikovat v rámci své práce, a budu nadále prezentovat výsledky jen tohoto jednoho průchodu prohledávání konfigurací obou agentů hlubokého Q-učení, a následnou hlubší analýzu konfigurací přenechávám svým následovníkům. Ono totiž tím, že výpočet nestihl doběhnout celý, tak neproběhly všechny kroku výpočtu, které archivují finální výsledky prohledávání. Následující analýza tak bude pokrývat pouze výsledky z debug logu průběhu prohledávání, který vypisuje alespoň přeživší aktuální generace,

a průměrný výsledek generace.

4.4.1 Výsledky procházených hodnot parametrů

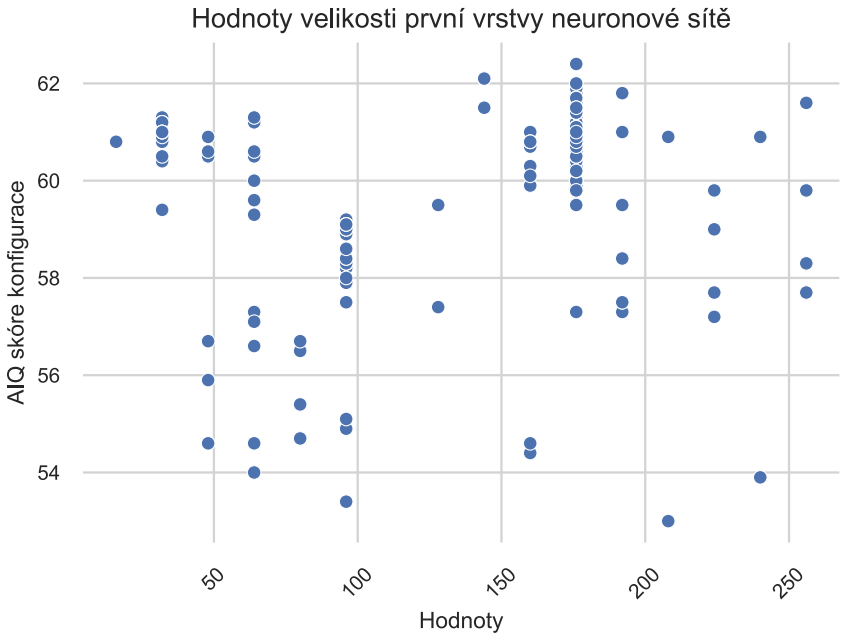
První věcí, než rozeberu samotné výsledky, jsou hodnoty jednotlivých parametrů konfigurací agentů. Z těchto hodnot jsem vytvořil grafy, aby bylo možné jednoduše ukázat, že prohledávání prostoru parametrů probíhalo podle očekávání, a opravdu bylo otestováno široké množství hodnot. Nutno říci, že mít data všech jedinců, kteří byli v jednotlivých generacích, tak by data byla výřečnější. Na následující graf se tedy musí koukat pohledem, že jde hlavně o ty lepší konfigurace z všech procházených, jelikož se jedná o data přeživších. Jednotlivé hodnoty parametrů jsem vynesl do grafů proti hodnotám AIQ skóre, kterého konfigurace, jež je použily, dosáhly. Celkové AIQ skóre je nicméně závislé i na hodnotě všech ostatních parametrů, ale tím že některé hodnoty jasně poukazovaly i na nějakou možnost korelace s jejich přímou hodnotou a výsledným AIQ skóre, tak jsem zvolil tento druh grafu.



Obrázek 4.1: Graf hodnot parametru Epsilon decay přeživších DDQ_l agenta

Příklad takového parametru s možností korelace je Epsilon decay u `DDQ_l` agenta, jehož hodnoty lze vidět na Obrázku 4.1. Lze pozorovat, že čím nižší je tato hodnota, tím lepšího AIQ skóre agent ve výsledku dosáhl. Takové pozorování je velice zajímavé, protože samotný parametr udává jakou část celého testu na daném programu agent prozkoumává prostředí. To že agent je schopný dosáhnout vyšších skóre již s velmi krátkou hodnotou epsilon decay (v rozsahu od 100 do 1000) nám ukazuje, že agent se naučí podle něj nejlepší politiku velice rychle, a zvyšováním času, kdy koná náhodné akce jen ubírá z jeho celkového výsledku. Agent byl testován na 25000 iteracích a tedy fakt, že již od 1000 iterace provádí hlavně svou naučenou politiku, značí, že se danou politiku naučil za 4 % času stráveného nad problémem.

Hodnoty většiny ostatních parametrů vůči AIQ skóre konfigurace moc neukazují existenci nějaké možné korelace a body v grafu tvoří spíše náhodné shluky viz Obrázek 4.2, který zobrazuje počty neuronů v první vrstvě neuronové sítě `DDQ_l` agenta. Podle tohoto grafu by se dalo říct, že složitost neuronové sítě na problémy AIQ testu nemá žádný výrazný vliv ale tím, že výsledné skóre závisí na celé konfiguraci nelze tento závěr nijak potvrdit. Je totiž docela možné, že druhá vrstva neuronové sítě vykompenzovala případný nižší počet neuronů první vrstvy, a proto byla konfigurace schopná získat vysoké AIQ skóre. U parametrů agenta vše souvisí se vším. Zbytek parametrů obou agentů lze najít v Příloze D.



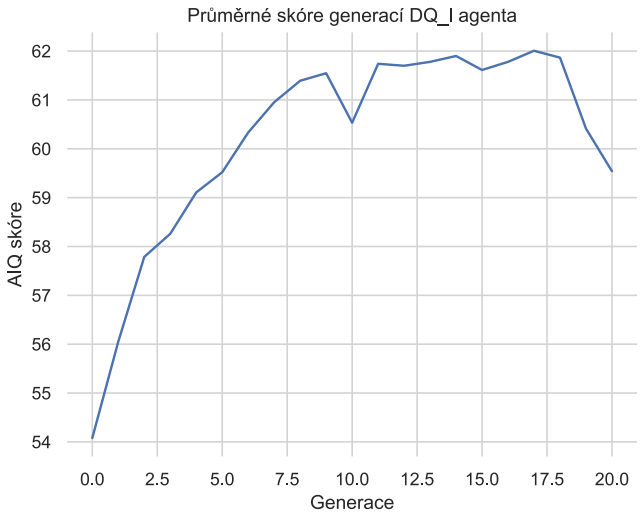
Obrázek 4.2: Graf hodnot velikosti první vrstvy neuronové sítě přeživších DDQ_l agenta

4.4.2 Vyhodnocení chodu testu

K ukázání, že použití evolučního algoritmu opravdu po jednotlivých generacích přicházelo s lepší a lepší konfigurací, použiji průměrné AIQ skóre přeživších jednotlivých generací. Tato hodnota by nejdříve měla být nízká a s každou následující generací, by se měla zvyšovat. Takové chování lze pozorovat na Obrázku 4.3 pro `DQ_l` agenta a na Obrázku 4.4 pro `DDQ_l` agenta. Tyto průměry pro agenty hodně oscilují, každopádně si zachovávají rostoucí tendenci, která je lépe pozorovatelná u `DDQ_l` agenta.

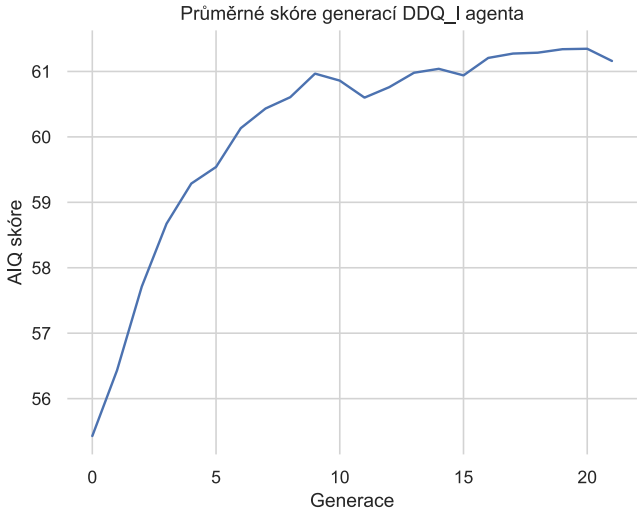
To, že hodnoty takhle hodně oscilují, a že po prvotním nárůstu průměrného AIQ skóre průměr zas tak neroste, neberu jako překvapující, protože evoluční algoritmus byl nastaven spíše k agresivnějšímu a co nejrychlejšímu průzkumu prostorů hodnot parametrů. Pokud bychom chtěli nastavit pomalejší prohledávání, které bude pomalu zlepšovat výsledky přeživších, musela by se snížit pravděpodobnost mutací, snížit se počet vybíraných přeživších, a zvolit konzervativnější strategie výběru.

Evoluční algoritmus zároveň pracuje s faktem, že každý jedinec musí mít nějakou, klidně velice



Obrázek 4.3: Graf hodnot velikosti první vrstvy neuronové sítě přeživších DDQ_l agenta

malou, pravděpodobnost výběru mezi přeživší nehledě na to, jak špatné jeho skóre ve finále je. I z toho důvodu průměrné skóre tak oscilují, protože přeživší nutně neodpovídají těm nejlepším konfiguracím dané generace. Použití turnajového výběru, ze všech implementovaných strategií volby přeživších, dává relativně nejnižší šanci výběru té nejlepší konfigurace z dané generace. To z toho důvodu, že do turnaje jsou vybíráni všichni jedinci se stejnou pravděpodobností. Já jsem tento druh výběru použil právě z důvodu zachovávání diverzity mezi populacemi a rychlejšímu prohledání prostoru s udržením jistého průměru skóre napříč generacemi.



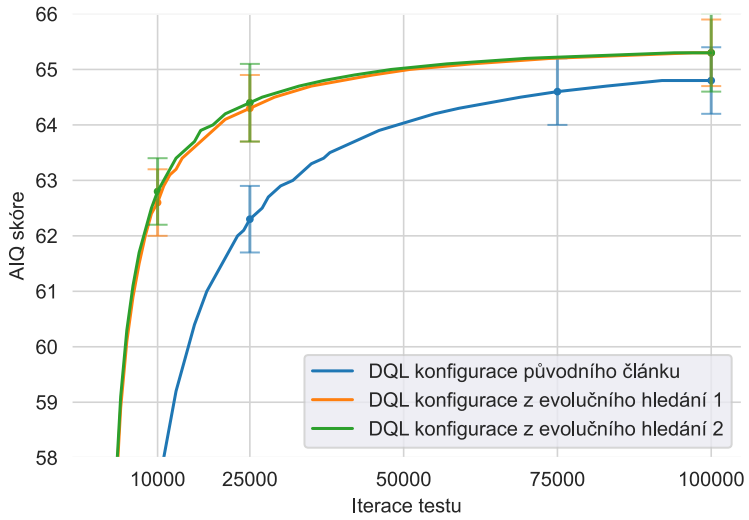
Obrázek 4.4: Graf hodnot velikosti první vrstvy neuronové sítě přeživších DDQ_l agenta

4.4.3 Vyhodnocení nejlepších nalezených konfigurací

I přesto, že prohledávání proběhlo pouze na 21 a 22 generacích tak nalezené konfigurace ukazují na lepší výsledky než jakých dosahují konfigurace z referenčních zdrojů. Evoluční

prohledávání pracovalo s AIQ testem nastaveným na 25000 iterací. Tato hodnota je ale nedostačující k vyvozování závěrů o finálním AIQ skóre agentů. Z toho důvodu jsem spustil 5 nejlepších nalezených konfigurací k dotestování s nastavením testu na 100000 iterací a k snížení standardních chyb jsem použil hodnotu 5000 jako sample size počtu testovaných programů. Výsledná hodnota AIQ skóre se pak s 95% pravděpodobností nachází někde v intervalu ±0,6 okolo výsledku testu.

Obrázek 4.5 zobrazuje dvě nejlepší z původních pěti konfigurací agenta `DQ_l` z genetického prohledávání společně s konfigurací ze článku Mnih, Kavukcuoglu, Silver, Rusu et al. (2015). Jak z něj lze vidět, tak obě nalezené konfigurace porážejí konfiguraci z článku z pohledu konvergence, kdy na 10000. iteraci je rozdíl mezi střední hodnotou skóre až 5 bodů, a na 25000. iteraci je rozdíl kolem 2 bodů AIQ skóre.



Obrázek 4.5: Vývoj AIQ skóre během testu nejlepších konfigurací DQ_l agenta

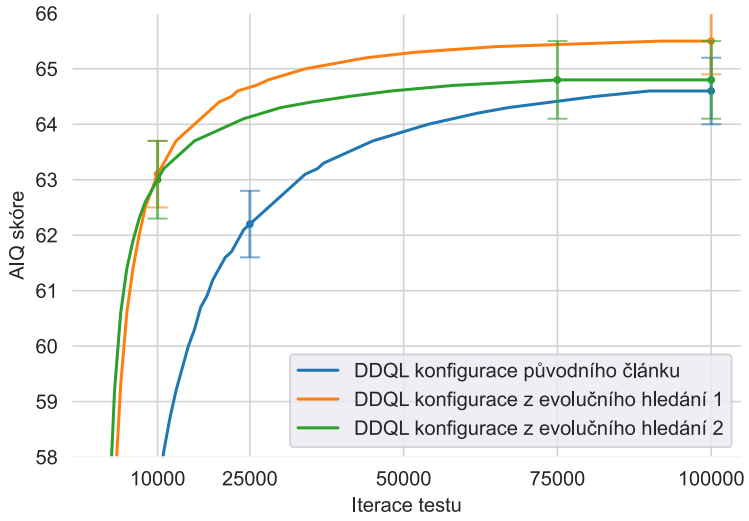
O samotném výsledku na poslední 100000. iteraci asi nelze na první pohled tvrdit, že by nalezené konfigurace byly prokazatelně lepší než konfigurace z článku, protože její výsledné AIQ skóre se nachází někde na intervalu (64,2; 65,4) se střední hodnotou 64,8. Výsledek té nejlepší konfigurace z evolučního algoritmu se oproti tomu nachází někde na intervalu (64,7; 65,9) se střední hodnotou 65,3.

Na tyto výsledky lze aplikovat Studentův t-test, protože součástí výsledků AIQ testu je i směrodatná odchylka měření. Jako nulovou hypotézu zadáme, že střední hodnota AIQ skóre konfigurace ze článku je shodná se střední hodnotou AIQ skóre nejlepší nalezené konfigurace. Jako alternativní hypotézu bereme, že AIQ skóre konfigurace z článku je nižší než AIQ skóre konfigurace z evolučního algoritmu. Pokud se jako hladina významnosti použije hodnota 95 %, a jako počet vzorků 5000, tak se přijde na to, že nulovou hypotézu nelze vyvrátit, a tudíž výsledky nejsou signifikantně rozdílné. Na testu s délkou 10000 iterací proto nelze prohlásit, že by pomocí evolučního algoritmu byla nalezena statisticky signifikantně lepší konfigurace pro agenta `DQ_l` než jakou uvádí autoři DQN ve svém článku.

Z pohledu na na data DDQ_l agenta, jejichž grafy lze vidět na Obrázku 4.6, tak lze zjistit, že situace je obdobná. Obě konfigurace z evolučního algoritmu konvergují rychleji než konfigurace ze článku. Zde jsou ale už vidět rozdíly i napříč oběma nalezenými konfiguracemi, kdy první konverguje lehce pomaleji, ale díky tomu je schopná ve finále získat značně vyšší AIQ skóre.

Z hlediska finálního výsledku na 100000. iteraci je rozdíl daleko viditelnější. Střední hodnota `DDQ_l` konfigurace z článku se v tomto případě nachází na hodnotě 64,6, což tvoří interval pro výslednou hodnotu (64; 65,2). Výsledek té lepší ze dvou testovaných konfigurací evolučního algoritmu má střední hodnotu výsledku na hodnotě 65,2, což tvoří interval (64,9; 66,1).

Opět se provede Studentův t-test, a i v tomto případě se použije hypotéza, že obě střední hodnoty jsou stejné. Jako alternativní hypotéza se použije tvrzení, že AIQ skóre konfigurace z článku je nižší než AIQ skóre konfigurace z evolučního algoritmu. Výsledek v tomto případě vyvrací nulovou hypotézu, a oba výsledky měření jsou statisticky signifikantně rozdílné. Pro nalezenou konfiguraci `DDQ_l` agenta se tak může tvrdit, že byla evolučním algoritmem nalezena lepší konfigurace než jaká byla publikována ve článku Mnih, Kavukcuoglu, Silver, Rusu et al. (2015).



Obrázek 4.6: Vývoj AIQ skóre během testu nejlepších konfigurací DDQ_l agenta

Obě nalezené konfigurace a výčty jejich parametrů lze najít přehledně v tabulkách v rámci Přílohy A pro `DQ_l` agenta a v rámci Přílohy B pro `DDQ_l` agenta. Obě tyto tabulky obsahují i hodnoty parametrů z článku Mnih, Kavukcuoglu, Silver, Rusu et al. (2015).

5. Vyhodnocení agentů a rozbor výsledků

Předchozí kapitoly představily agenty hlubokého Q-učení nejdříve po teoretické stránce, poté byla představena jejich implementace v rámci AIQ testu, a následně byly nalezeny konfigurace parametrů agentů vhodné pro jejich použití v rámci AIQ testu. Tyto nalezené konfigurace byly porovnány s konfiguracemi z původních vědeckých článků, které hluboké Q-učení představují, v rámci kapitoly 4.4.3.

Tato kapitola představí výsledky agentů hlubokého Q-učení využívající tyto konfigurace nejdříve mezi sebou (sekce 5.1), protože bude zajímavé ověřit, jestli `DDQ_l`, jakožto evoluce `DQ_l`, opravdu bude mít lepší výsledky.

Dále budou prozkoumány výsledky obou agentů vůči agentům, kteří již jsou v rámci testu implementováni (sekce 5.3). Konkrétně se jedná o:

- agenta implementující frekvenční tabulku,
- agenta implementující Q-učení (Watkins a Dayan, 1992),
- agenta implementující Q-učení s automatickou úpravou rychlosti učení (Hutter a Legg, 2008),
- agenta implementující metodu Vanilla Policy Gradient z práce Zeman (2023),
- agenta implementující metodu Proximal Policy Gradient z práce Zeman (2023).

Zvláštní důraz bude kladen na vztah agentů hlubokého Q-učení a agenta prostého Q-učení (sekce 5.2). Oba druhy totiž využívají k tvorbě politiky Q-funkci, a bude tak zajímavé prozkoumat, jak implementace Q-funkce pomocí neuronových sítí ovlivnila výsledky v AIQ testu oproti původní implementaci pomocí tabulky Q-hodnot.

5.1 Porovnání implementovaných agentů DQ_l a DDQ_l

Z kapitoly 2.3 o hlubokém Q-učení víme, že `DDQ_l` agent se od agenta `DQ_l` liší tím, že `DDQ_l` obsahuje druhou neuronovou síť, která prakticky tvoří učící hodnoty pro hlavní neuronovou síť. Ta se u obou architektur učí politiku řešící dané prostředí. Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) ve své práci popisující přínosy použití druhé neuronové sítě a ukazují i praktické výsledky obou architektur.

5.1.1 Rozbor výsledků z původních článků o hlubokém Q-učení

Původní články používaly k vyhodnocování agentů simulátor starých arkádových her herní konzole Atari 2600 (Bellemare, Naddaf et al., 2013). Výsledky obou architektur se proto liší v závislosti na jednotlivých hrách, i přesto lze ale z výsledků pozorovat, že architektura se dvěma neuronovými sítěmi vždy dosahuje lepších výsledků než architektura původní. Toto jsou publikované výsledky pro pět her.:

- Breakout, kde byl výsledek dvou neuronových sítí o 31,6 % lepší.
- Enduro, kde byl výsledek o 21 % lepší.
- River Raid, kde byl výsledek o 81,5 % lepší.
- Seaquest, kde byl výsledek o 251,8 % lepší.
- Space Invaders, kde byl výsledek o 31,7 % lepší.

Z publikovaných výsledků tedy vyplývá, že architektura se dvěma neuronovými sítěmi je v průměru o 83 % lepší než architektura původní. Na takto omezených datech má velký vliv výsledku ze hry Seaquest (Cartwright, 1983). V této hře hráč ovládá ponorku, jejímž cílem je zachraňovat potápěče, zatímco se vyhýbá nebo sestřeluje nepřátele v podobě žraloků nebo jiných ponorek. K tomu všemu má ještě hráčova ponorka omezený počet kyslíku a je tak potřeba se pravidelně vynořovat na hladinu. Hráč získává body za každého sestřeleného nepřítele a za každého zachráněno potápěče při vynoření. Pokud je ponorka zcela zaplněná šesti potápěči, hráč získá bonusové skóre za zbývající kyslík při vynoření. Jedná se tedy o poměrně složitou hru.

Oproti tomu například hra Enduro (Miller, 1983), kde je rozdíl výsledku nejnižší, je závodní hra, ve které je hráčovým cílem kontrolovat neustále zrychlující auto a předjíždět soupeře. Pokud v rámci jednoho závodu nepředjede daný počet aut, hra končí. Ve hře se také střídá den a noc, ovlivňující viditelnost ostatních aut, a mění se i počasí, které lehce mění chování auta na vozovce. I tak se ale jedná, v porovnání se Seaquestem, o hru se značně jednoduššími herními mechanikami.

Podobná jednoduchá složitost by se dala popsat i u hry Breakout (Bushnell, 1976), kdy hráč ovládá plošinu, od které se odráží míč bořící cihly při nárazu, za jejichž zničení dostává hráč body. Cihly mají různé barvy podle toho v jaké vrstvě se nachází, a od této barvy se odvíjí i počet získaných bodů za jejich zničení s tím, že čím vyšší vrstva, tím více bodů. Pokud

hráč nestihne odrazit míč svojí plošinou a nechá míč propadnout, tak ztrácí život. Hra se ve specifických intervalech zrychluje, a po zničení nejvyšší vrstvy se dokonce plošina zkrátí na polovinu.

Ve známé hře Space invaders (Nishikado, 1978) hráč ovládá vesmírnou loď ve spodní části obrazovky a musí sestřelovat mimozemské lodě v horní části obrazovky zatímco se musí vyhýbat jejich střelám. Mimozemské lodě začínají v pěti řadách po jedenácti a společně se hýbou zleva doprava, a po každé této otočce se pohnou o jednu řadu níže. Hra končí pokud hráč ztratí tři životy nebo se libovolná loď dostane na spodek herní obrazovky. Tuto hru by také šlo označit za mechanicky jednoduchou.

Ve hře River Raid (Shaw, 1982) hráč ovládá letoun letící nad řekou, ze které nesmí vybočit, zatímco sestřeluje nepřátelské lodě a letouny, za které dostává body. Hráč může s letounem manévrovat za strany na stranu a také může zrychlovat či zpomalovat. Hráč si musí hlídat hladinu své palivové nádrže, kterou musí doplňovat barely s palivem, které taktéž plavou na řece. Hráč tyto barely může i ničit, pokud je nepotřebuje, a za to také dostává body. Řeka se během letu různě kroutí a rozděluje do vícero proudů. Tato hra už je mechanicky složitější, kvůli nutnosti strategizovat ohledně paliva, než Breakout, Enduro nebo Space invaders, a složitostí se spíše přibližuje k obtížnosti Seaquestu.

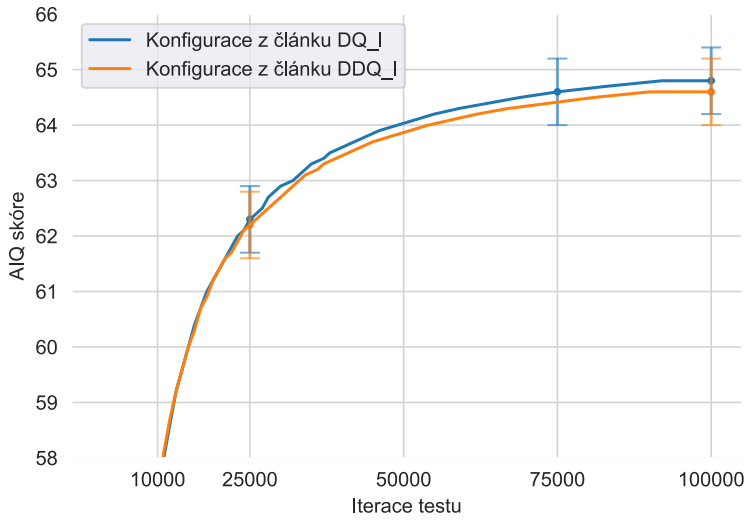
S těmito znalosti by se o výsledcích dalo říct, že rozdíl obou architektur hlubokého Q-učení je tím větší, čím složitější hru hraje. V jazyce posilovaného učení by toto tvrzení šlo přeformulovat jako, že rozdíl mezi výsledky agentů obou architektur se zvyšuje s tím, jak se zvyšuje složitost řešeného prostředí. Pět výsledků samozřejmě nestačí k podložení takového tvrzení, nicméně jedná se o zajímavé pozorování.

5.1.2 Rozbor výsledků z AIQ testu

Pro zjištění rozdílů mezi architekturami `DQ_l` agenta a `DDQ_l` agenta použijeme konfigurace z původních článků. Hodnoty jednotlivých parametrů jsou dostupné v rámci příloh A a B. Tyto parametry jsou totiž shodné pro obě architektury, což je důležité na objektivní posouzení vlivu architektury.

V rámci AIQ testu byl použit referenční stroj BF5 a samotný test byl nastaven na 100000 iterací a velikost vzorku 5000 pro testované programy. Tento sample size poskytuje test výsledky s přesností ±0,6. Graf na Obrázku 5.1 zachycuje vývoj AIQ skóre v průběhu testu pro oba agenty. Lze si všimnout, že výsledek `DQ_l` agenta je o trochu lepší s výsledným skóre 64,8. Architektura s dvěma neuronovými sítěmi - `DDQ_l` zato dosáhla výsledku 64,6.

Pokud na tyto výsledky aplikujeme Studentův t-test, kde jako nulovou hypotézu položíme předpoklad, že oba výsledky jsou si rovny, a jako alternativní hypotézu, že výsledek `DQ_l` agenta je lepší, tak zjistíme, že na hladině významnosti 95 % výsledek nezamítá nulovou hypotézu. Výsledky obou agentů tak nejsou statisticky signifikantně rozdílné, a tudíž mohou být označeny za stejné. Jak je toto ale možné, když autoři původní článku uvádí, že by



Obrázek 5.1: Vývoj AIQ skóre obou DQN architektur s konfigurací podle původního článku

architektura s dvěma neuronovými sítěmi by měla mít stabilnější proces učení a dosahovat v průměru o 83 % lepších výsledků?

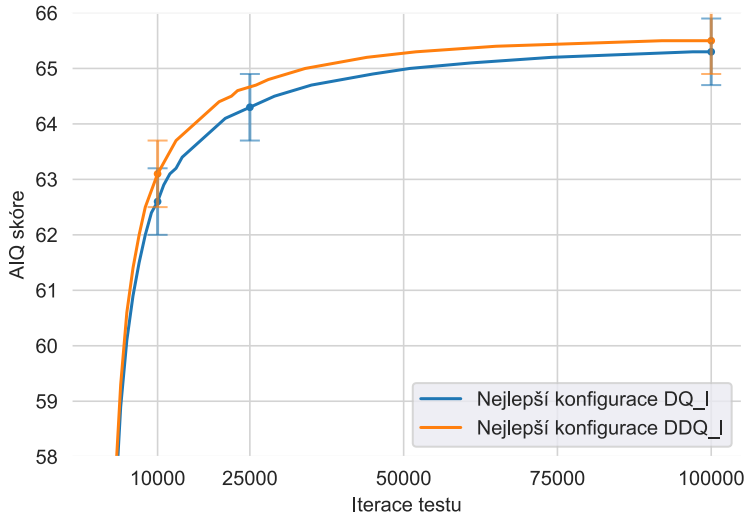
Odpověď by se z části mohla skrývat v pozorování z předchozí podkapitoly, kdy rozdíly byly v původním článku tím markantnější čím složitější hry agenti hráli. Samotná prostředí v AIQ testu na BF5 stroji modelují problémy, kde agentovi je předán stav v podobě celého čísla nabývající hodnot od 0 do 5, na která agent musí zareagovat akcí v podobě celého čísla z rozsahu od 0 do 5. Agent hrající Atari hru, jejíž stav je reprezentován jako snímek hrací plochy, představuje pro agenta nejspíš násobně složitější problém než aktuálně nejsložitější prostředí AIQ testu.

Z toho důvodu, pokud by hypotéza o rozdílech ovlivněných složitostí prostředí byla pravdivá, by dávalo smysl, že na jednoduchých prostředích se rozdíl mezi oběma architekturami neukáže. To je každopádně zajímavé zjištění, které stojí za další prošetření. V rámci AIQ testu bylo provedeno toto měření AIQ skóre konfigurací ze článku ještě jednou na testu využívající referenční stroj BF11. Tím pádem se zvětšil stavový prostor a i prostor akcí, ale výsledky opět nenasvědčovaly tomu, že by mezi agenty byly markantní rozdíly. Nejspíš zvětšení těchto prostorů není z pohledu agenta dostatečné zesložitění prostředí, aby se tyto rozdíly ukázaly.

Pokud se srovnají nejlepší nalezené konfigurace z evolučního prohledávání konfiguračního prostoru mezi sebou, vzniknou tak výsledky, které lze vidět v grafu na Obrázku 5.2. Nejlepší aktuálně nalezená konfigurace `DQ_l` agenta dosahuje AIQ skóre o hodnotě 65,3. Nejlepší nalezená konfigurace `DDQ_l` agenta dosahuje AIQ skóre o hodnotě 65,5. Rozdíl těchto výsledků je tedy opět 0,2 bodu.

Studentův t-test opět odhalil, že tyto výsledky na hladině významnosti 95 % nejsou signifikantně rozdílné. Je ale také potřeba hledět na to, že není zaručeno, že jsou tyto nalezené konfigurace z prostoru všech konfigurací opravdu ty nejlepší, ani že by jejich vzdálenost k této

případné nejlepší konfiguraci byla stejně velká.



Obrázek 5.2: Vývoj AIQ skóre obou DQN architektur s nejlepšími nalezenými konfiguracemi

Z tohoto důvodu bych hypotézu o tom, že je architektura `DDQ_l` lepší než architektura `DQ_l`, bych v rámci AIQ testu považoval za nepotvrzenou. Důvod je pravděpodobně přílišná jednoduchost prostředí AIQ testu, na kterých se nepodařilo zachytit rozdíly mezi architekturami.

5.2 Vztah agentů hlubokého Q-učení k agentovi prostého Q-učení

Tato podkapitola prozkoumá vztahy agentů hlubokého Q-učení s agentem klasického Q-učení. Oba typy totiž využívají koncept Q-hodnoty a Q-funkce představené v sekci 2.2.2 k učení svých politik. Z toho důvodu bude zajímavé provést pozorování jak modelování Q-funkce pomocí neuronových sítí ovlivní výsledek AIQ testu.

Rozdíl by teoreticky nemusel být nijak veliký, a to právě z důvodu, že tvorba politiky odpovídá stejným konceptům, a liší se pouze implementace vyhodnocování hodnoty Q-funkce. Na druhé straně Mnih, Kavukcuoglu, Silver, Graves et al. (2013) uvádí, že DQN poráží všechny dříve testované agenty posilovaného učení, a tudíž i Q-learning. Je tedy stejně tak možné, že `DQ_l` i `DDQ_l` budou mít výrazně lepší výsledky.

Jako hlavní nevýhodu Q-learningu uvádí Mnih, Kavukcuoglu, Silver, Graves et al. (2013) nutnost práce s nízko-dimenzionálním vyjádřením stavů. Klasický Q-learning totiž implementuje Q-funkci pomocí vyhledávací tabulky, kde klíč tvoří kombinace stavu a zvolené akce. Jak tyto dva vektory rostou v rozměrech, tak exponenciálně roste i velikost této vyhledávací tabulky. Aplikace takového algoritmu na obrazová data je tedy takřka neproveditelná. Nahrazení této tabulky neuronovou sítí nám umožní zpracovávat takřka libovolné stavové prostory. AIQ test s použitím referenčního stroje BF5 používá stavový prostor o pěti různých hodnotách. S tím si hravě poradí i tabulkový Q-learning, proto nelze očekávat, že se tato výhoda DQN architektury v AIQ testu nějak projeví.

Neuronové sítě během učení dokáží odhalovat neznámé vztahy mezi vstupními daty. Tato vlastnost by jistě mohla pomoci k naučení se robustnější politiky, než jakou nabízí explicitní vyjádření stavů v klasickém Q-learningu. Otázka poté je, zda se tyto skryté vztahy ukáží v prostředích AIQ testu.

Implementace Q-learningového agenta v AIQ testu má oproti implementacím agentů hlubokého Q-učení statickou hodnotu Epsilon, tedy parametru, který udává pravděpodobnost, že agent provede náhodou akci. Agenti hlubokého Q-učení využívají Epsilon decay, tedy metodu, kde Epsilon začíná na hodnotě 1 a poté agent lineárně v každém učícím kroku tuto hodnotu sníží až na spodní hranici. Přijde mi proto pro objektivní porovnání rozdílů agentů nutné použít agenty hlubokého Q-učení s konfigurací, kde je Epsilon decay vypnuté a hodnota Epsilonu je nastavená na stejnou statickou hodnotu pro všechny tři architektury (`Q_l`, `DQ_l`, `DDQ_l`).

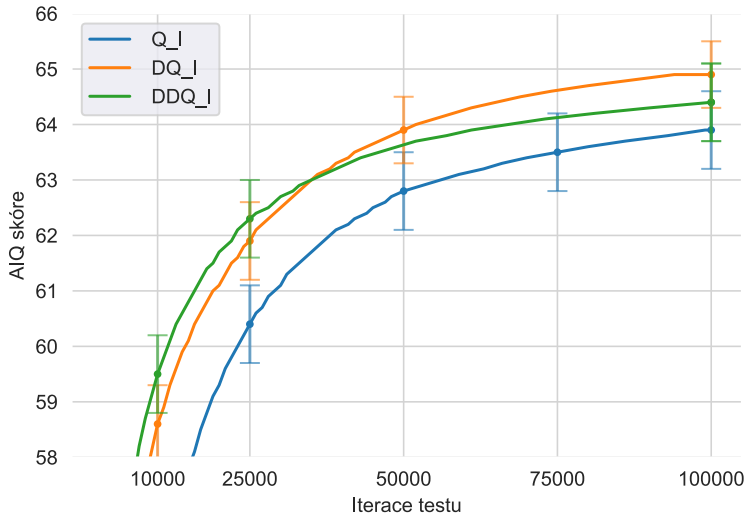
Nastavení AIQ testu pro tento experiment využívá referenční stroj BF5, délku testu nastavenou na 100000 iterací a sample size programů o velikosti 5000. Konfigurace `Q_l` agenta odpovídá nejlepší konfiguraci z předchozích prací o AIQ testu (Zeman, 2023), (Vadinský, 2018). Jedná se o konfiguraci `Q_l,0.0,0.5,0.5,0.005,0.95`. Tato konfigurace používá jako Epsilon hodnotu 0,005.

Konfigurace agentů `DQ_l` a `DDQ_l` odpovídá konfiguraci z evolučního prohledávání. Tyto hodnoty parametrů jsou dostupné v přílohách A a B. Hodnota délky Epsilon decay bude

v tomto případě nastavena na hodnotu 0, čímž se tato funkcionalita vypne a agent bude pracovat se statickou hodnotou Epsilon, která odpovídá hodnotě, kterou používá `Q_l` agent tedy 0,005.

Výsledek tohoto experimentu zachytává graf na Obrázku 5.3. Agent `Q_l` zde dosáhl AIQ skóre o hodnotě $63{,}9 \pm 0{,}6$, agent `DQ_l` dosáhl AIQ skóre o hodnotě $64{,}9 \pm 0{,}6$, a agent `DDQ_l`dosáhl hodnoty AIQ skóre rovnou hodnotě $64{,}4 \pm 0{,}6$. Agenti hlubokého Q-učení konvergují daleko rychleji než původní Q_agent a oba dosahují i vyššího výsledku.

Zajímavá je právě konvergence `DQ_l` a `DDQ_l` agentů, kdy můžeme pozorovat, že DDQ_l agent konverguje rychleji, snad vlivem stabilnějšího učení, ale ve finále dosáhne nižšího výsledku než `DQ_l`.



Obrázek 5.3: Vývoj AIQ skóre obou DQN architektur s nejlepšími nalezenými konfiguracemi

Pokud na tyto výsledky použijeme Studentův t-test, kdy jako nulovou hypotézu bere, že výsledky dvou agentů jsou schodné, a jako alternativní hypotézu bereme, že se od sebe liší, tak na hladině významnosti 95 % zjistíme, že:

- výsledek `DQ_l` agenta se signifikantně liší od výsledku `Q_l` agenta,
- výsledek `DDQ_l` agenta se signifikantně neliší od výsledku `Q_l` agenta,
- výsledek `DDQ_l` agenta se signifikantně neliší od výsledku `DQ_l` agenta.

Staticky významně lepší výsledek tak získala pouze původní architektura s jednou neuronovou sítí. I tak jde ale o rozdíl jednoho bodu AIQ skóre, tudíž lze tvrdit, že hluboké Q-učení dosahuje v rámci AIQ testu lehce lepších výsledků, ale tyto výsledky nejsou nijak drasticky lepší.

5.3 Porovnání agentů DQ_l a DDQ_l vůči všem ostatním dříve implementovaným agentům AIQ testu

Původním cílem této práce byla implementace agentů hlubokého Q-učení, jejich otestování a porovnání s ostatními agenty již obsažené v AIQ testu. Agenti hlubokého Q-učení zde využívali nejlepší konfigurace z evolučního prohledávání a tentokrát mají zapnuté epsilon decay. Tato podkapitole bude rozebírat výsledky právě takového experimentu.

Jako relevantní agenti, k porovnání s agenty hlubokého učení, jsou následující architektury:

- agent implementující frekvenční tabulku (označovaný `Freq`),
- agent implementující klasické Q-učení (označovaný `Q_l`),
- agent implementující hierarchické Q-učení (označovaný `H_l`),
- agent implementující architekturu Vanilla Policy Gradient (označovaný `VPG`),
- a agent implementující architekturu Proximal Policy Gradient (označovaný `PPO`).

Tito agenti budou využívat nejlepší známé konfigurace z předchozích prací o AIQ testu: Zeman (2023) a Vadinský (2018). Tyto konfigurace bez rozboru významu jednotlivých parametrů jsou:

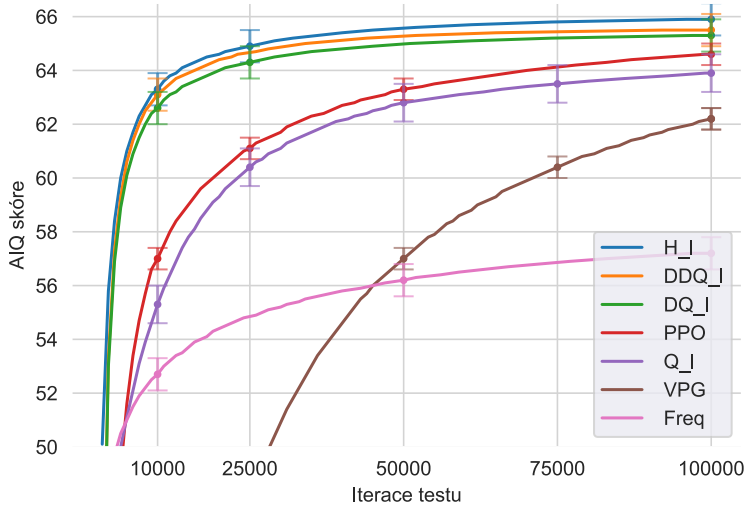
- `Freq(0.07)`,
- `Q_l(0.0,0.5,0.5,0.005,0.95)`,
- `H_l(0.0,0.0,0.995,0.01,0.8)`,
- `VPG(50,80, 0.99,0.0003,0.001,0.97)`,
- `PPO(50,80,80,0.99,0.0003,0.001,0.97,0.2,0.01)`,

Samotný AIQ test využívá referenční stroj BF5, s konfigurací na 100000 iterací a sample size programů o velikost 5000. Toto nastavení nám dává výsledky AIQ skóre na 95% intervalu spolehlivosti o rozsahu ±0,6 kolem střední hodnoty, která je brána jako výsledné skóre AIQ testu. 100000 iterací je zároveň dostatečně dlouhá doba, aby i výsledky pomaleji se učících agentů stihly zkonvergovat.

Výsledky tohoto finální experimentu zachytává graf na Obrázku 5.4 nebo dále Tabulka 5.1 s konkrétními středními hodnotami AIQ skóre pro vybrané iterace.

Z těchto výsledků, ryze podle výsledných středních hodnot měření, vyplývá následující pořadí od nejlepšího agenta po nejhoršího:

1. `H_l` agent s AIQ $65{,}9 \pm 0{,}6$
2. `DDQ_l` agent s AIQ $65{,}5 \pm 0{,}6$
3. `DQ_l` agent s AIQ $65{,}3 \pm 0{,}6$
4. `PPO` agent s AIQ $64{,}6 \pm 0{,}6$
5. `Q_l` agent s AIQ $63{,}9 \pm 0{,}6$
6. `VPG` agent s AIQ $62{,}2 \pm 0{,}6$
7. `Freq` agent s AIQ $57{,}2 \pm 0{,}6$



Obrázek 5.4: Vývoj AIQ skóre všech agentů s jejich nejlepší známou konfigurací

Z těchto výsledků se již dokáže s jistotou říct, že agent `Freq` je z testovaných agentů nejhorší, protože jeho výsledek vyšel nejnižší, a ani když se vezme v potaz interval ±0,6, tak se ve výsledné oblasti nenachází žádný z výsledků ostatních agentů. Druhý nejhorší poté vyšel agent `VPG`, který je svým výsledkem podstatně blíže k agentům na prvním až pátém místě, ale bohužel se s nimi interval jeho AIQ skóre nijak neprotíná.

Tabulka 5.1: AIQ skóre testovaných agentů podle iterací

Iterace	5000	10000	25000	50000	100000
`H_l`	61,0	63,3	64,9	65,5	65,9
`DDQ_l`	60,6	63,1	64,6	65,2	65,5
`DQ_l`	60,1	62,6	64,3	64,9	65,3
`PPO`	51,6	57,0	61,1	63,3	64,6
`Q_l`	51,1	55,3	60,4	62,8	63,9
`VPG`	18,3	30,9	48,2	57,0	62,2
`Freq`	51,0	52,7	54,8	56,2	57,2

Zajímavější je situace ohledně výsledků zbylých agentů. Jejich výsledky si jsou navzájem velice blízko, a jejich výsledné intervaly AIQ skóre mají mezi sebou často jistý průnik. K otestování, zda výsledky těchto agentů se mohou prohlásit za odlišné, použijeme Studentův t-test. Jako nulovou hypotézu se vždy položí předpoklad, že jsou si výsledky dvou agentů rovny. Alternativní hypotéza je předpoklad, že se výsledky statisticky signifikantně liší. V t-testu se pracuje s 95% hladinou spolehlivosti. AIQ test totiž společně se střední hodnotou a intervaly AIQ skóre vrací i údaj o střední hodnotě daného měření, která umožňuje použití t-testu. Po křížovém otestování výsledků zbylých agentů zjistíme následující:

- skóre `Q_l` agenta a `PPO` agenta se nijak signifikantně neliší,
- skóre `Q_l` agenta a `DQ_l` agenta se signifikantně liší,

- skóre `PPO` agenta a `DQ_l` agenta se nijak signifikantně neliší,
- skóre `PPO` agenta a `DDQ_l` agenta se signifikantně liší,
- skóre `H_l` agenta a `DDQ_l` agenta se nijak signifikantně neliší,
- skóre `H_l` agenta a `DQ_l` agenta se nijak signifikantně neliší,
- skóre `DDQ_l` agenta a `DQ_l` agenta se nijak signifikantně neliší.

Z těchto výsledků jako nejlepší výsledky lze označit skóre agentů `H_l` a `DDQ_l`. Ty se sice signifikantně neliší od `DQ_l` agenta, ale jinak jsou signifikantně lepší než ostatní agenti. Výsledek agenta `DQ_l` se signifikantně neliší od výsledku agenta `PPO`, a proto nemůže být zařazen na první příčku, ačkoliv se od agentů `H_l` a `DDQ_l` signifikantně neliší. Přiřazuji mu proto druhou příčku. Agent `PPO` je signifikantně horší oproti agentům na první příčce. Zároveň se signifikantně neliší od `DQ_l`, ale ani od agenta `Q_l`, který už je z pohledu `DQ_l` signifikantně horší. Z toho důvodu se agent `PPO` umístí na třetí příčce. Na čtvrtou příčku umisťuji agenta `Q_l`, který je signifikantně horší než `H_l`, `DDQ_l` i `DQ_l`.

Po provedení statistického testu významnosti rozdílů výsledků vypadá pořadí agentů následovně:

1. `H_l` agent s AIQ $65{,}9 \pm 0{,}6$ a `DDQ_l` agent s AIQ $65{,}5 \pm 0{,}6$
2. `DQ_l` agent s AIQ $65{,}3 \pm 0{,}6$
3. `PPO` agent s AIQ $64{,}6 \pm 0{,}6$
4. `Q_l` agent s AIQ $63{,}9 \pm 0{,}6$
5. `VPG` agent s AIQ $62{,}2 \pm 0{,}6$
6. `Freq` agent s AIQ $57{,}2 \pm 0{,}6$

Zajímavé je se ale podívat na výsledky i z pohledu rychlosti konvergence. Agent `Freq` konverguje ze všech agentů nejpomaleji. Výsledek agenta `VPG` také konverguje velice pomalu a z grafu vývoje jeho skóre to možná vypadá, že při použití vyššího počtu iterací by jeho výsledek mohl být vyšší, a možná by mohl konkurovat i výsledkům v první pětce. Agent `Q_l` konverguje daleko rychleji než předchozí zmínění agenti, ale zdaleka ne tak rychle jako agenti `H_l`, `DDQ_l` a `DQ_l`. Skóre agenta `PPO` konverguje podobně jako skóre `Q_l` a z t-testu vyplívá, že rozdíly nejsou statisticky významné, takže lze prohlásit, že konvergují stejně rychle. U zbylých tří agentů `H_l`, `DQ_l` a `DDQ_l` je zajímavé jak podobná skóre si udržují po celou dobu běhu testu. V žádném bodě nemají naměřené střední hodnoty navzájem odstup více než jednoho bodu AIQ skóre. Zároveň ze všech měřených agentů konvergují nejrychleji. To vypovídá o tom, že nejlepší politiku pro dané prostředí jsou se agenti schopni naučit velice brzy, a později se jen podle ní chovají. Tato velmi rychle naučená politika je zároveň tak dobrá, že agentům přináší nejlepší výsledky po celou dobu testování až do konce.

Závěr tohoto experimentu zní následovně: Oba implementovaní agenti hlubokého Q-učení získali nejlepší naměřený výsledek oproti již dříve implementovaným agentům. O tento nejlepší výsledek se dělí s agentem hierarchického Q-učení, mezi jejichž výsledky nebyl naměřen vzájemně statisticky signifikantní rozdíl. S tímto agentem zároveň sdílí i nejrychlejší konvergenci mezi testovanými agenty.

U výsledku agenta hlubokého Q-učení s jednou neuronovou sítí `DQ_l` nebyl nalezen signifikantně lepší výsledek oproti agentovi `PPO`, který byl ale také vysoký. Z toho důvodu ho označujeme jako možná mírně horšího než u architektury s dvěma neuronovými sítěmi (`DDQ_l`), ačkoliv se jejich výsledky vzájemně signifikantně nelišily.

Závěr

Cílem této diplomové práce bylo navázání na výzkum měření inteligence za pomocí míry zvané Univerzální inteligence (Legg a Hutter, 2007), a konkrétněji její aproximaci v rámci Algoritmického IQ (Legg a Veness, 2011). Práci s tímto druhem měření inteligence se věnuje na Fakultě Informatiky a Statistiky Vysoké školy Ekonomické v Praze Vadinský (2018) a následně Zeman (2023), a tato diplomová práce tak navazuje a dále rozšiřuje poznání o testu Algoritmického IQ.

V rámci této diplomové práce byl tento AIQ test rozšířen o dva nové agenty, u kterých bylo požadováno změření jejich hodnoty Algoritmické inteligence a jejich zasazení do kontextu výsledku ostatních agentů. Tito agenti využívají metodu hlubokého Q-učení (Mnih, Kavukcuoglu, Silver, Graves et al., 2013; Mnih, Kavukcuoglu, Silver, Rusu et al., 2015). Hluboké Q-učení představuje koncept, kdy výpočet Q-funkce z klasického Q-učení provádí neuronová síť, která se tuto Q-funkci musí naučit. Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) tento koncept rozšiřuje o přidání druhé neuronové sítě do procesu učení, jejímž úkolem je stabilizovat proces učení Q-funkce. První z přidaných agentů, `DQ_l`, tak vychází z původního konceptu hlubokého Q-učení, a druhý přidaný agent, `DDQ_l`, čerpá z rozšiřujícího konceptu s druhou neuronovou sítí.

Oba agenti byli implementováni v jazyce Python s využitím populárních knihoven pro práci s neuronovými sítěmi PyTorch (Paszke et al., 2019). Jazyk Python byl pro implementaci vybrán z důvodu toho, že samotná implementace AIQ testu je naprogramována v jazyce Python.

Hledání optimálních konfigurací agentů

Pro správné otestování agentů není ale nutná pouze implementace jejich architektur, ale také nalezení vhodných konfigurací jejich parametrů. Agenti totiž mohou být spuštěni k otestováním s řadou volitelných nastavení, které mohou i radikálně měnit jejich schopnosti učení, interakci s testem a tedy i finální výsledky. Agenti implementovaní v rámci této práce mají možnost nastavení až 14 parametrů, často se spojitým intervalem hodnot. Nalezení správné konfigurace tak nebyl jednoduchý úkol.

Z důvodu takto obrovského konfiguračního prostoru implementovaných agentů byla do testu přidána funkcionalita na prohledávání parametrů agentů využívající evoluční algoritmy. Tato funkcionalita umožňuje vytvoření konfigurace parametrů agentů, která vlastně udává, jaké hodnoty může daným parametrům agenta přiřadit. S těmito nastaveními se vytvoří generace agentů, kteří jsou otestováni na AIQ testu, a z jejich výsledků se pomocí principů evolučních algoritmů vytvoří další generace, která postupně získává lepší a lepší výsledky, čímž se heuristicky prohledává konfigurační prostor agentů, a postupně toto prohledávání směřuje k nejlepší možné konfiguraci bez nutnosti otestování všech možných konfigurací.

Tento proces se ukázal jako velice výpočetně náročný, a i s využitím velkého výpočetního výkonu díky Gridovému výpočetnímu centru Metacentrum se nepodařilo provést úspěšné prohledávání delší než 22 generací. Na výsledcích z tohoto běhu evolučních algoritmů, ale bylo ukázáno, že průměrné skóre mezi generacemi roste a prohledávání na základě evolučních algoritmů funguje. V případě architektury agenta se dvěma neuronovými sítěmi byla dokonce nalezena lepší konfigurace než jakou uvádí Mnih, Kavukcuoglu, Silver, Rusu et al. (2015).

Experimenty s implementovanými agenty a jejich výsledky

S implementovanými agenty a s jejich nalezenými konfiguracemi byly dále prováděny experimenty. Nejdřív byl otestován vztah obou architektur hlubokého Q-učení, protože Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) uvádí, že přidání druhé neuronové sítě zlepšilo výsledky agenta v průměru o 83 %. Tyto rozdíly se každopádně v rámci AIQ testu nepodařilo naměřit, a v našem testu vycházejí výsledky obou agentů bez statisticky významného rozdílu. K tomuto závěru byl použit na výsledky AIQ testu Studentův t-test.

Nutno ale říci, že Mnih, Kavukcuoglu, Silver, Rusu et al. (2015) používali k testování agentů simulátor herní konzole Atari 2600 (Bellemare, Naddaf et al., 2013), a agenti tak byli trénováni a testováni na hraní videoher. AIQ test oproti tomu testuje agenty na řešení řádově jednodušších problémech, a tak důvod toho proč nebyly naměřeny žádné podstatné rozdíly mezi konfiguracemi může být přílišná jednoduchost AIQ testu, na kterém se stabilnější učení nemá šanci projevit.

Dále byl zkoumán rozdíl hlubokého Q-učení a klasického Q-učení. Oba přístupy totiž využívají stejný koncept pro učení politiky na základě Q-funkce. Zde byl naměřen statisticky signifikantní rozdíl pouze u jedné architektury hlubokého Q-učení proti klasickému Q-učení. I tak se ale výsledky nelišily o více než jeden bod AIQ skóre. Maximum AIQ skóre je sto, tudíž v nejlepším případě přineslo hluboké Q-učení při stejných podmínkách zlepšení o 1 %.

V rámci nejlepších konfigurací byl rozdíl podstatně vyšší, ale to je dáno i tím, že hluboké Q-učení obnáší i jiné triky než pouze nahrazení Q-funkce neuronovou sítí, které byly pro hlavní část tohoto experimentu trochu potlačeny. Konkrétně agent klasického Q-učení používá fixní hodnotu epsilon, která ovlivňuje jak často provádí náhodné akce a prozkoumává prostředí. Hluboké Q-učení dynamicky mění hodnotu epsilonu pomocí metody epsilon decay. Při tomto experimentu byla hodnota epsilonu zafixována pro všechny agenty.

Posledním provedeným experimentem bylo srovnání agentů hlubokého Q-učení vůči všem do té doby implementovaným agentům. Zde již byly použity nalezené konfigurace z evolučních algoritmů. Vůči agentovi využívající frekvenční tabulku bylo hluboké Q-učení o 8,3 bodů AIQ skóre lepší, oproti nejlepší konfiguraci agenta Q-učení bylo hluboké Q-učení o 1,6 bodů lepší, a vůči agentovi využívající hierarchické Q-učení nebyl naměřen statisticky významný rozdíl. Moji agenti déle byli testováni proti agentům z Zeman (2023), kdy hluboké Q-učení bylo o 3,3 bodů lepší než agent Vanilla Policy gradient, a 0,9 bodu lepší než agent využívající

Proximal Policy Gradient metodu, kdy byl zjištěn signifikantní rozdíl jen u jedné z architektur hlubokého Q-učení. Agenti hlubokého Q-učení se tak vcelku dělí o první příčku s agentem `H_l`, který využívá verzi Q-učení s automatickou úpravou rychlosti učení (Hutter a Legg, 2008).

Všechny výsledky AIQ testu vychází s jistou nejistotou v podobě intervalů spolehlivosti okolo střední hodnoty. Finální výsledky seřazené podle těchto středních hodnot vypadají následovně:

1. `H_l` agent s AIQ $65{,}9 \pm 0{,}6$
2. `DDQ_l` agent s AIQ $65{,}5 \pm 0{,}6$
3. `DQ_l` agent s AIQ $65{,}3 \pm 0{,}6$
4. `PPO` agent s AIQ $64{,}6 \pm 0{,}6$
5. `Q_l` agent s AIQ $63{,}9 \pm 0{,}6$
6. `VPG` agent s AIQ $62{,}2 \pm 0{,}6$
7. `Freq` agent s AIQ $57{,}2 \pm 0{,}6$

Komplikace v průběhu řešení

Samotný průběh této práce zkomplikovalo prvotní zděšení nad vzájemnými výsledky obou architektur agentů hlubokého Q-učení. Nejdříve se problém skrýval v implementaci, kdy prvně původní architektura s jednou neuronovou sítí získávala značně lepší výsledky než architektura druhá. Tento problém byl odstraněn, ale výsledky pořád naznačovaly tomu, že AIQ skóre obou architektur vychází shodně, snad možná pořád trochu ve prospěch původní architektuře s jednou sítí.

Po důkladných kontrolách jednotlivých implementací, a zkoušení spuštění testu na vyšším stavovém prostoru, jsme s vedoucím práce dospěli k závěru, že tyto výsledky jsou správné, ale že možná problém tkví v samotném AIQ testu. Pro takto robustní agenty, kteří zvládají podle původních článků hrát videohry, jsou problémy řešené v rámci AIQ testu nejspíš natolik jednoduché, že se rozdíl neukáže. Když píši jednoduché tak nemyslím, že jsou triviální k vyřešení, to by jinak agenti dosahovali daleko vyšších hodnot AIQ skóre. Spíš tím myslím, že na takto omezeném stavovém prostoru, který ani tak není zdaleka vytížen do maxima, je situace taková, že se agent buď naučí daný problém vyřešit, a nebo se ho nenaučí, a počet neuronových sítí tento fakt nijak neovlivní.

Při pozorování samotného procesu učení obou architektur jsme ale pozorovali daleko hladší vývoj a konvergenci jednotlivých učených Q-hodnot z Q-funkce, a tudíž jsme pozorovali, že druhá neuronová síť opravdu stabilizuje učení. Zdá se ale, že tento proces v AIQ testu při použití referenčního stroje `BF` nijak zásadně výsledné AIQ skóre neovlivní.

Druhým úskalím této práce bylo samotné hledání nejlepších konfigurací obou agentů. Zavedení evolučního algoritmu byl jistě krok správným směrem, bez něj bych možná tak zvládl prozkoumat nejbližší okolí konfigurace z původních článků. Bohužel jsem ale nepočítal s časovou náročností tohoto prohledávání. Kdybych věděl, že na výsledky budu čekat v jednom případě

až 11 týdnů, tak bych se snažil zajistit funkčnost této funkcionality daleko dříve a daleko rychleji, abych se samotným prohledáváním mohl začít dříve. Nalezené konfigurace jsou dobré, ale dle mého jsou ještě hodně daleko od nějaké potenciálně nejlepší. K nalezení nejlepší by bylo potřeba projít daleko více generací než pouhých 22, které se mi povedlo projít v rámci této práce. Funkcionalita evolučních algoritmů každopádně umožňuje znovuspuštění s výchozí populací. Ta by mohla představovat mnou nalezené konfigurace, a prohledávání by tak mohlo navázat na to, kde já skončil.

Obecně si ale myslím, že zadání práce bylo splněno, a navíc byla objevena nová poznání, která by mohla sloužit jako podklad pro další výzkum v oblasti testování Algoritmického IQ.

Všechen přidaný kód a výsledky experimentů jsou dostupné skrz platformu GitHub `https://github.com/TheMischko/AIQ-DQN-DP`.

Bibliografie

AGOSTINI, Susan D., 2017. Review of "Current Practice of Clinical Electroencephalography, 4th Edition," by John S. Ebersole, Aatif M. Husain, and Douglas R. Nordli. *The Neurodiagnostic Journal* [online]. Vol. 57, no. 1, s. 104–109 [cit. 2023]. ISSN 2164-6821, ISSN 2375-8627. Dostupné z DOI: 10.1080/21646821.2016.1270694.

ARULKUMARAN, Kai, DEISENROTH, Marc Peter, BRUNDAGE, Miles a BHARATH, Anil Anthony, 2017. A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine* [online]. Roč. 34, č. 6, s. 26–38 [cit. 2023]. ISSN 1053-5888. Dostupné z DOI: 10.1109/MSP.2017.2743240.

BELLEMARE, Marc G., DABNEY, Will a MUNOS, Rémi, 2017. *A Distributional Perspective on Reinforcement Learning* [online]. arXiv [cit. 2023]. Č. arXiv:1707.06887. Dostupné z arXiv: 1707.06887[cs,stat].

BELLEMARE, Marc G., NADDAF, Yavar, VENESS, Joel a BOWLING, Michael, 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research* [online]. Roč. 47, s. 253–279 [cit. 2023]. ISSN 1076-9757. Dostupné z DOI: 10.1613/jair.3912.

BELLMAN, Richard a DREYFUS, Stuart, 2010. *Dynamic programming*. 1. Princeton Landmarks in Mathematics ed., with a new introduction. Princeton, NJ: Princeton University Press. Princeton Landmarks in mathematics. ISBN 978-0-691-14668-3.

BISHOP, Christopher M., 2006. *Pattern recognition and machine learning*. New York: Springer. Information science and statistics. ISBN 978-0-387-31073-2.

BROWNLEE, Jason, 2021. *How to Choose an Activation Function for Deep Learning* [MachineLearningMastery.com] [online]. 2021-01-17. [cit. 2023]. Dostupné z: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

BUSHNELL, Noaln, 1976. *Breakout* [AtariOnline.org] [online]. 1976-05-13. [cit. 2024]. Dostupné z: https://atarionline.org/atari-2600/breakout.

CARTWIGHT, Steve, 1983. *Seaquest* [AtariOnline.org] [online]. [cit. 2024]. Dostupné z: https://atarionline.org/atari-2600/seaquest.

DASGUPTA, D. a MICHALEWICZ, Zbigniew, 1997. *Evolutionary algorithms in engineering applications*. Berlin: Springer. ISBN 978-3-662-03423-1. OCLC: 861706268.

EASTER, Brandee, 2020. Fully Human, Fully Machine: Rhetorics of Digital Disembodiment in Programming. *Rhetoric Review* [online]. Vol. 39, no. 2, s. 202–215 [cit. 2023]. ISSN 0735-0198, ISSN 1532-7981. Dostupné z DOI: 10.1080/07350198.2020.1727096.

EIBEN, A. E., AARTS, E. H. L. a VAN HEE, K. M., 1991. Global convergence of genetic algorithms: A markov chain analysis. In: SCHWEFEL, Hans-Paul a MÄNNER, Reinhard (ed.). *Parallel Problem Solving from Nature*. Berlin, Heidelberg: Springer Berlin Heidelberg, s. 3–12. ISBN 978-3-540-70652-6.

EIBEN, A. E. a SMITH, J. E., 2015. *Introduction to Evolutionary Computing*. 2nd ed. 2015. Berlin, Heidelberg: Springer Berlin Heidelberg : Imprint: Springer. Natural Computing Series. ISBN 978-3-662-44874-8. Dostupné z DOI: 10.1007/978-3-662-44874-8.

ELBRÄCHTER, Dennis, PEREKRESTENKO, Dmytro, GROHS, Philipp a BÖLCSKEI, Helmut, 2021. Deep Neural Network Approximation Theory. *IEEE Transactions on Information Theory*. Roč. 67, č. 5, s. 2581–2623. Dostupné z DOI: 10.1109/TIT.2021.3062161.

ÉTORÉ, Pierre a JOURDAIN, Benjamin, 2010. Adaptive Optimal Allocation in Stratified Sampling Methods. *Methodology and Computing in Applied Probability* [online]. Vol. 12, no. 3, s. 335–360 [cit. 2023]. ISSN 1387-5841, ISSN 1573-7713. Dostupné z DOI: 10.1007/s11009-008-9108-0.

FOGEL, Lawrence J., OWENS, Alvin J. a WALSH, Michael J., 1966. *Artificial intelligence through simulated evolution.* Oxford, England: John Wiley & Sons. Artificial intelligence through simulated evolution. Pages: xii, 170.

FOSTEL, Gary, 1993. The Turing test is for the birds. *ACM SIGART Bulletin* [online]. Vol. 4, no. 1, s. 7–8 [cit. 2023]. ISSN 0163-5719. Dostupné z DOI: 10.1145/173993.173996.

FRANÇOIS-LAVET, Vincent, HENDERSON, Peter, ISLAM, Riashat, BELLEMARE, Marc G. a PINEAU, Joelle, 2018. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning* [online]. Vol. 11, no. 3, s. 219–354 [cit. 2023]. ISSN 1935-8237, ISSN 1935-8245. Dostupné z DOI: 10.1561/2200000071.

GARDNER, Howard, 1986. Frames of Mind: The Theory of Multiple Intelligences. In: dostupné také z: https://api.semanticscholar.org/CorpusID:146322489.

GIRSHICK, Ross, 2015. *Fast R-CNN* [online]. arXiv [cit. 2023]. Č. arXiv:1504.08083. Dostupné z arXiv: 1504.08083[cs].

GLASSERMAN, Paul a YAO, David D., 1992. Some Guidelines and Guarantees for Common Random Numbers. *Manage. Sci.* Roč. 38, č. 6, s. 884–908. ISSN 0025-1909. Place: Linthicum, MD, USA Publisher: INFORMS.

GOTTFREDSON, Linda S., 1997. Mainstream science on intelligence: An editorial with 52 signatories, history, and bibliography. *Intelligence* [online]. Vol. 24, no. 1, s. 13–23 [cit. 2023]. ISSN 01602896. Dostupné z DOI: 10.1016/S0160-2896(97)90011-8.

GROSSI, Enzo a BUSCEMA, Massimo, 2007. Introduction to artificial neural networks: *European Journal of Gastroenterology & Hepatology* [online]. Vol. 19, no. 12, s. 1046–1054 [cit. 2023]. ISSN 0954-691X. Dostupné z DOI: 10.1097/MEG.0b013e3282f198a0.

HAMMERSLEY, J. M. a MORTON, K. W., 1956. A new Monte Carlo technique: antithetic variates. *Mathematical Proceedings of the Cambridge Philosophical Society* [online]. Vol. 52, no. 3, s. 449–475 [cit. 2023]. ISSN 0305-0041, ISSN 1469-8064. Dostupné z DOI: 10.1017/S0305004100031455.

HARNAD, Stevan, 1991. Other bodies, other minds: A machine incarnation of an old philosophical problem. *Minds and Machines* [online]. Vol. 1, no. 1, s. 43–54 [cit. 2023]. ISSN 0924-6495, ISSN 1572-8641. Dostupné z DOI: 10.1007/BF00360578.

HASSELT, Hado van, GUEZ, Arthur a SILVER, David, 2015. *Deep Reinforcement Learning with Double Q-learning* [online]. arXiv [cit. 2023]. Č. arXiv:1509.06461. Dostupné z arXiv: 1509.06461[cs].

HERNANDEZ-ORALLO, Jose, 2000. Beyond the Turing Test. *Journal of Logic, Language, and Information* [online]. Roč. 9, č. 4, s. 447–466 [cit. 2023]. ISSN 09258531, ISSN 15729583. Dostupné z: http://www.jstor.org/stable/40180237. Publisher: Springer.

HERNÁNDEZ-ORALLO, José a DOWE, David L., 2010. Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence* [online]. Vol. 174, no. 18, s. 1508–1539 [cit. 2023]. ISSN 00043702. Dostupné z DOI: 10.1016/j.artint.2010.09.006.

HIBBARD, Bill, 2009. Bias and No Free Lunch in Formal Measures of Intelligence. *Journal of Artificial General Intelligence* [online]. Roč. 1, č. 1, s. 54–61 [cit. 2023]. ISSN 1946-0163. Dostupné z DOI: 10.2478/v10229-011-0004-6.

HINTON, Geoffrey E., OSINDERO, Simon a TEH, Yee Whye, 2006. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*. Roč. 18, s. 1527–1554. Dostupné také z: https://api.semanticscholar.org/CorpusID:2309950.

HOLLAND, John H., 2019. *Adaptation in Natural and Artificial Systems - An Introductory Analysis with Applications to Biology, Control, and Artificial I*. Cambridge: The MIT Press. ISBN 978-0-262-27555-2. OCLC: 1235962848.

HUNT, Andrew a THOMAS, David, 2000. *The pragmatic programmer: from journeyman to master*. Reading, Mass: Addison-Wesley. ISBN 978-0-201-61622-4.

HUTTER, Marcus, 2007. On universal prediction and Bayesian confirmation. *Theoretical Computer Science* [online]. Vol. 384, no. 1, s. 33–48 [cit. 2023]. ISSN 03043975. Dostupné z DOI: 10.1016/j.tcs.2007.05.016.

HUTTER, Marcus a LEGG, Shane, 2008. *Temporal Difference Updating without a Learning Rate* [online]. arXiv [cit. 2023]. Č. arXiv:0810.5631. Dostupné z arXiv: 0810.5631[cs].

JEBARI, Khalid, 2013. Selection Methods for Genetic Algorithms. *International Journal of Emerging Sciences*. Roč. 3, s. 333–344.

KINGMA, Diederik P. a BA, Jimmy, 2017. *Adam: A Method for Stochastic Optimization* [online]. arXiv [cit. 2023]. Č. arXiv:1412.6980. Dostupné z arXiv: 1412.6980[cs].

KOLMOGOROV, A. N., 1968. Three approaches to the quantitative definition of information *. *International Journal of Computer Mathematics* [online]. Vol. 2, no. 1, s. 157–168 [cit. 2023]. ISSN 0020-7160, ISSN 1029-0265. Dostupné z DOI: 10.1080/00207166808803030.

LEGG, Shane a HUTTER, Marcus, 2007. *Universal Intelligence: A Definition of Machine Intelligence* [online]. arXiv [cit. 2023]. Č. arXiv:0712.3329. Dostupné z arXiv: 0712.3329[cs].

LEGG, Shane a VENESS, Joel, 2011. *An Approximation of the Universal Intelligence Measure* [online]. arXiv [cit. 2023]. Č. arXiv:1109.5951. Dostupné z arXiv: 1109.5951[cs].

LEVIN, L. A., 1973. On the notion of a random sequence. Roč. 14, č. 5, s. 1413–1416. Dostupné také z: https://www.cs.bu.edu/fac/lnd/dvi/ran72.pdf.

MAHANTA, Jahnavi, 2017. *Introduction to Neural Networks, Advantages and Applications* [Medium] [online]. 2017-07-12. [cit. 2023]. Dostupné z: https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-96851bd1a207

MARTELLO, Silvano a TOTH, Paolo, 1990. *Knapsack problems: algorithms and computer implementations*. Chichester: J. Wiley. Wiley-interscience series in discrete mathematics and optimization. ISBN 978-0-471-92420-3.

MCCULLOCH, Warren S. a PITTS, Walter, 1943. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics* [online]. Vol. 5, no. 4, s. 115–133 [cit. 2023]. ISSN 0007-4985, ISSN 1522-9602. Dostupné z DOI: 10.1007/BF02478259.

MIDKIFF, Samuel P., 2012. Program parallelization. In: MIDKIFF, Samuel P. *Automatic Parallelization* [online]. Cham: Springer International Publishing, s. 51–68 [cit. 2023]. ISBN 978-3-031-00608-1 978-3-031-01736-0. Dostupné z DOI: 10.1007/978-3-031-01736-0_3. Series Title: Synthesis Lectures on Computer Architecture.

MIJWIL, Maad, 2018. Artificial Neural Networks Advantages and Disadvantages.

MILLER, Larry, 1983. *Enduro (Atari 2600) online game* [AtariOnline.org] [online]. 1983-05. [cit. 2024]. Dostupné z: https://atarionline.org/atari-2600/enduro.

MNIH, Volodymyr, KAVUKCUOGLU, Koray, SILVER, David, GRAVES, Alex, ANTONOGLOU, Ioannis, WIERSTRA, Daan a RIEDMILLER, Martin, 2013. *Playing Atari with Deep Reinforcement Learning* [online]. arXiv [cit. 2023]. Č. arXiv:1312.5602. Dostupné z arXiv: 1312.5602[cs].

MNIH, Volodymyr, KAVUKCUOGLU, Koray, SILVER, David, RUSU, Andrei A., VENESS, Joel, BELLEMARE, Marc G., GRAVES, Alex, RIEDMILLER, Martin, FIDJELAND, Andreas K., OSTROVSKI, Georg, PETERSEN, Stig, BEATTIE, Charles, SADIK, Amir, ANTONOGLOU, Ioannis, KING, Helen, KUMARAN, Dharshan, WIERSTRA, Daan, LEGG, Shane a HASSABIS, Demis, 2015. Human-level control through deep reinforcement learning. *Nature* [online]. Vol. 518, no. 7540, s. 529–533 [cit. 2023]. ISSN 0028-0836, ISSN 1476-4687. Dostupné z DOI: 10.1038/nature14236.

NISHIKADO, Tomohiro, 1978. *Space Invaders* [AtariOnline.org] [online]. [cit. 2024]. Dostupné z: https://atarionline.org/atari-2600/space-invaders.

NOV, Oded, SINGH, Nina a MANN, Devin, 2023. *Putting ChatGPT's Medical Advice to the (Turing) Test* [online]. arXiv [cit. 2023]. Č. arXiv:2301.10035. Dostupné z arXiv: 2301.10035[cs].

ONDŘEJ VADINSKÝ, PETR ZEMAN a JAN ŠTIPL, 2024. *AIQ test*. Dostupné také z: https://github.com/xvado00/AIQ.

OTTERLO, Martijn van a WIERING, Marco A., 2012. Markov Decision Processes: Concepts and Algorithms. In.

PASZKE, Adam, GROSS, Sam, MASSA, Francisco, LERER, Adam, BRADBURY, James, CHANAN, Gregory, KILLEEN, Trevor, LIN, Zeming, GIMELSHEIN, Natalia, ANTIGA, Luca, DESMAISON, Alban, KÖPF, Andreas, YANG, Edward, DEVITO, Zach, RAISON, Martin, TEJANI, Alykhan, CHILAMKURTHY, Sasank, STEINER, Benoit, FANG, Lu, BAI, Junjie a CHINTALA, Soumith, 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library* [online]. arXiv [cit. 2023]. Č. arXiv:1912.01703. Dostupné z arXiv: 1912.01703[cs,stat].

ROSENBLATT, F., 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* [online]. Vol. 65, no. 6, s. 386–408 [cit. 2023]. ISSN 1939-1471, ISSN 0033-295X. Dostupné z DOI: 10.1037/h0042519.

RUMELHART, David E., HINTON, Geoffrey E. a WILLIAMS, Ronald J., 1986. Learning representations by back-propagating errors. *Nature* [online]. Vol. 323, no. 6088, s. 533–536 [cit. 2023]. ISSN 0028-0836, ISSN 1476-4687. Dostupné z DOI: 10.1038/323533a0.

SARLE, Warren, 2002. *comp.ai.neural-nets FAQ, Part 2 of 7: Learning* [comp.ai.neural-nets FAQ, Part 2 of 7: Learning] [online]. 2002-10-11. [cit. 2023]. Dostupné z: http://www.faqs.org/faqs/ai-faq/neural-nets/part2/.

SEARLE, John R., 1980. Minds, brains, and programs. *Behavioral and Brain Sciences* [online]. Vol. 3, no. 3, s. 417–424 [cit. 2023]. ISSN 0140-525X, ISSN 1469-1825. Dostupné z DOI: 10.1017/S0140525X00005756.

SHAW, Carol, 1982. *River Raid* [AtariOnline.org] [online]. [cit. 2024]. Dostupné z: https://atarionline.org/atari-2600/river-raid.

SCHWEFEL, Hans-Paul, 1995. *Evolution and Optimum Seeking*. ISBN 978-0-471-57148-3.

SIPSER, Michael, 2013. *Introduction to the theory of computation*. Third edition, international edition. Australia Brazil Japan Korea Mexiko Singapore Spain United Kingdom United States: Cengage Learning. ISBN 978-1-133-18779-0 978-1-133-18781-3 978-0-357-67058-3.

SOLOMONOFF, R.J., 1964. A formal theory of inductive inference. Part I. *Information and Control* [online]. Vol. 7, no. 1, s. 1–22 [cit. 2023]. ISSN 00199958. Dostupné z DOI: 10.1016/S0019-9958(64)90223-2.

SPEARMAN, C., 1904. "General Intelligence," Objectively Determined and Measured. *The American Journal of Psychology* [online]. Roč. 15, č. 2, s. 201–292 [cit. 2023]. ISSN 00029556. Dostupné z: http://www.jstor.org/stable/1412107. Publisher: University of Illinois Press.

SURI, Karush, 2021. *Deep Eligibility Traces*. Dostupné také z: https://github.com/karush17/Deep-Eligibility-Traces.

SUTTON, Richard S. a BARTO, Andrew G., 2018. *Reinforcement learning: an introduction*. Second edition. Cambridge, Massachusetts: The MIT Press. Adaptive computation and machine learning series. ISBN 978-0-262-03924-6.

TURING, A. M., 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* [online]. Vol. s2-42, no. 1, s. 230–265 [cit. 2023]. ISSN 00246115. Dostupné z DOI: 10.1112/plms/s2-42.1.230.

TURING, A. M., 1950. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind* [online]. Vol. LIX, no. 236, s. 433–460 [cit. 2023]. ISSN 1460-2113, ISSN 0026-4423. Dostupné z DOI: 10.1093/mind/LIX.236.433.

VADINSKÝ, Ondřej, 2018. *Analýza předpokladů inteligentního chování počítačových systémů*. Praha. Dostupné také z: https://vskp.vse.cz/73718. Dis. pr. Vysoká škola ekonomická v Praze.

VENESS, Joel, NG, Kee Siong, HUTTER, Marcus a SILVER, David, 2010. *Reinforcement Learning via AIXI Approximation* [online]. arXiv [cit. 2023]. Č. arXiv:1007.2049. Dostupné z arXiv: 1007.2049[cs].

VISSER, Beth A., ASHTON, Michael C. a VERNON, Philip A., 2006. g and the measurement of Multiple Intelligences: A response to Gardner. *Intelligence* [online]. Vol. 34, no. 5, s. 507–510 [cit. 2023]. ISSN 01602896. Dostupné z DOI: 10.1016/j.intell.2006.04.006.

WATKINS, Christopher J. C. H. a DAYAN, Peter, 1992. Q-learning. *Machine Learning* [online]. Vol. 8, no. 3, s. 279–292 [cit. 2023]. ISSN 0885-6125, ISSN 1573-0565. Dostupné z DOI: 10.1007/BF00992698.

ZEMAN, Petr, 2023. *Assessing Policy Optimization agents using Algorithmic IQ test*. Praha. Dis. pr. Vysoká škola ekonomická v Praze.

ZHANG, Qian, LU, Jie a JIN, Yaochu, 2021. Artificial intelligence in recommender systems. *Complex & Intelligent Systems* [online]. Vol. 7, no. 1, s. 439–457 [cit. 2023]. ISSN 2199-4536, ISSN 2198-6053. Dostupné z DOI: 10.1007/s40747-020-00212-w.

Přílohy

A. Tabulka doporučených parametrů pro agenta DQ_l

Název parametru agenta	Hodnota podle (Mnih, Kavukcuoglu, Silver, Rusu et al., 2015)	Hodnota z genetického algoritmu
learning_rate	0,0003	0,0005
gamma	0,99	0,91
batch_size	32	4
epsilon	1	1
epsilon_decay_length	2000	500
neural_size_l1	64	128
neural_size_l2	512	224
neural_size_l3	0	0
use_rmsprop	1	0
history_length	2	3
lambda	0	1
eligibility_strategy	0	2
Dosažené skóre	64,8	65,3

B. Tabulka doporučených parametrů pro agenta DDQ_l

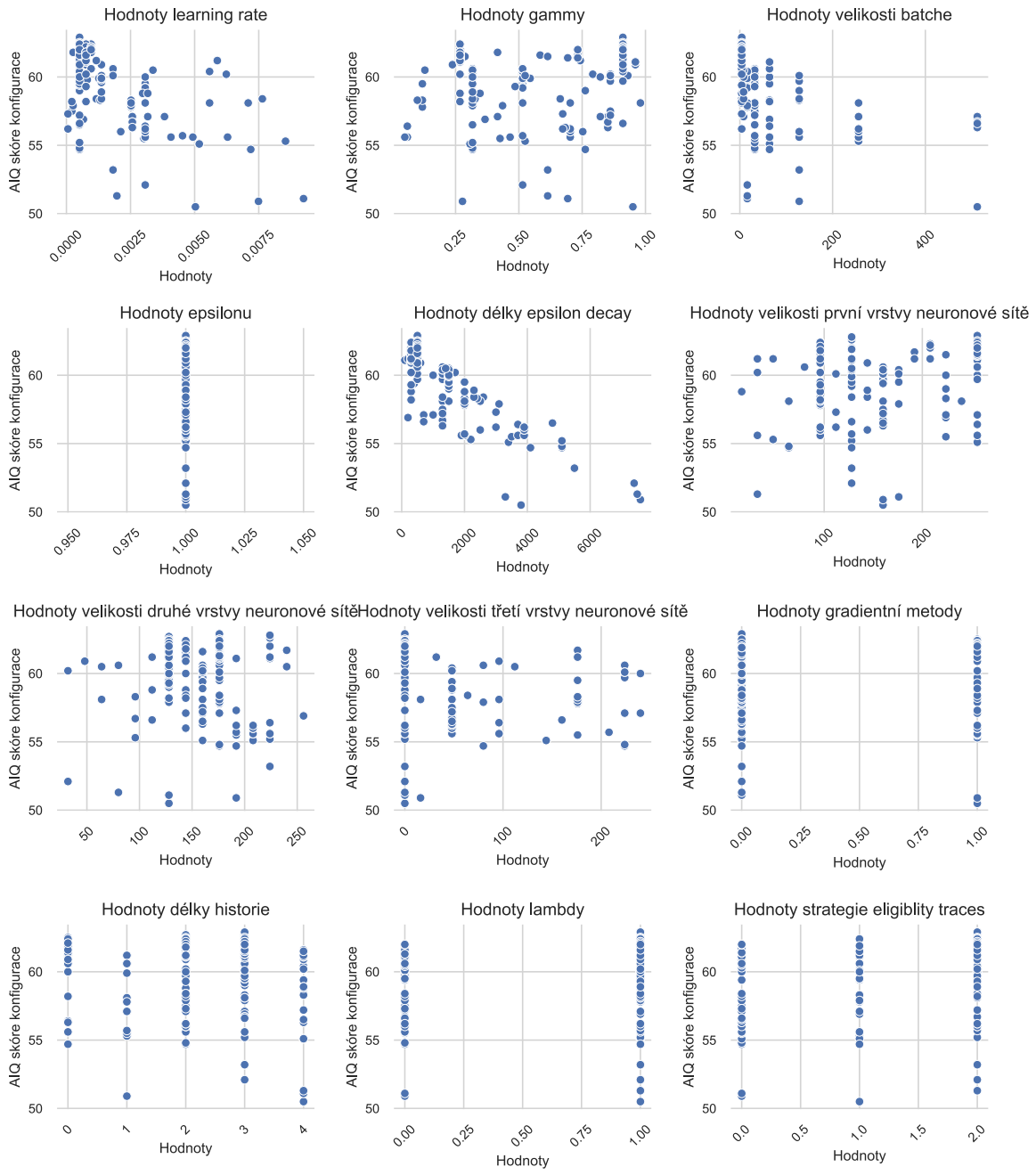
Název parametru agenta	Hodnota podle (Mnih, Kavukcuoglu, Silver, Rusu et al., 2015)	Hodnota z genetického algoritmu
learning_rate	0,0003	0,0005
gamma	0,99	0,91
batch_size	32	4
epsilon	1	1
epsilon_decay_length	2000	500
neural_size_l1	64	256
neural_size_l2	512	176
neural_size_l3	0	0
use_rmsprop	1	0
history_length	2	3
tau	1	0,485
update_interval_length	200	195
lambda	0	1
eligibility_strategy	0	2
Dosažené skóre	64,6	65,2

C. Parametry skriptu evolučního algoritmu

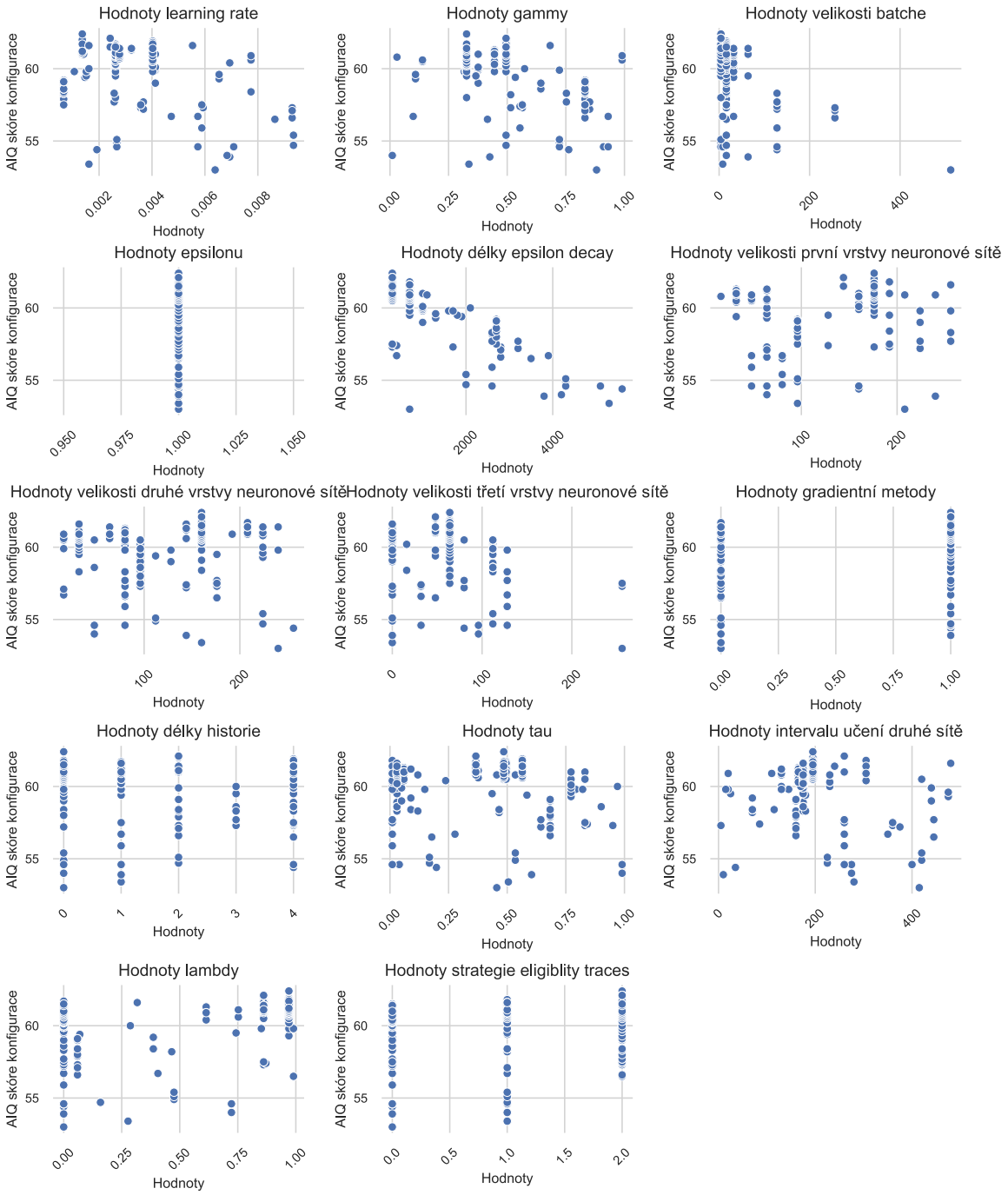
Příznak v terminálu	**Povinný?**	**Název parametru**	**Význam parametru**
`-a`	Ano	agent	Podobně jako v AIQ se tímto určuje, jaký agent má být testován. Název musí odpovídat definici agenta z adresáře genetics/agents_ref.
`-c`	Ano	count select	Udává počet přeživších, které se vyberou pro tvorbu následující generace.
`–debug`	Ne	debuggování	Umožňuje zapnout detailní výpis průběhu hledání do standartního výstupu.
`-e`	Ano	epochs	Nastavuje kolik epoch bude hledání trvat neboli kolik generací bude vytvořeno, než se hledání ukončí.
`-i`	Ano	iterations	Nastavuje počet iterací, které budou předány AIQ testu pro testování každé konfigurace agenta.
`–log`	Ne	logging	Zapne v AIQ testu logování výsledků do adresáře log/.
`–log_el`	Ne	logging intermediate results	Zapne v AIQ testu logování výsledků po tisíci iteracích, které budou uloženy v rámci adresáře log-el/.
`-m`	Ano	mutation probability	Nastavuje pravděpodobnost mutací při tvorbě nových individuí tak, že pro každý parametr bude s touto pravděpodobností zmutován. Nejběžněji se nastavuje jako ; a v průměru tak bude zmutován jeden parametr každého individua.
`-n`	Ano	number of agents	Udává kolik agentů bude v rámci paralelizace testováno najednou. Je potřeba počítat, že celý hledací proces zabere až ; počet vláken.
`-p`	Ano	population size	Udává z kolika individuí se bude skládat každá jedna generace.
`-r`	Ano	strategy	Nastavuje, jakou strategií bude probíhat volba přeživších do další generace. 0 = ruletový výběr, 1 = výběr podle pořadí a 2 = turnajový výběr.

Příznak v terminálu	**Povinný?**	**Název parametru**	**Význam parametru**
`-s`	Ano	sample	Nastavuje pro AIQ test na jak velkém vzorku programů budou jednotlivé konfigurace testovány.
`–seed_file`	Ne	file with generation zero	Pokud přes tento parametr bude připojen soubor obsahující seznam konfigurací, tak první generace bude obsahovat hlavně konfigurace z tohoto souboru, a až poté vygeneruje nové, pokud v tomto souboru nenajde tolik konfigurací, aby byla generace plná. Obsah tohoto souboru by mělo být pole konfigurací zapsané v JSON formátu.
`-t`	Ano	threads	Udává kolik jader využije jedna instance AIQ testu, při testování jedné konfigurace. Je potřeba počítat, že celý hledací proces zabere až ; počet vláken.

D. Rozsahy hodnot přeživších v evolučním algoritmu



Obrázek D.1: Grafy rozsahů hodnot přeživších DQ_l agenta



Obrázek D.2: Grafy rozsahů hodnot přeživších DDQ_l agenta