FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Efektivní implementace neuroevoluce pro úlohy posilovaného učení
Student:	Vladimír Votava
Vedoucí:	doc. Ing. Pavel Kordík, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Umělá inteligence 2021
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Prostudujte techniky neuroevoluce v režimu posilovaného učení a efektivní implementace umožňující běh v prostředí webového prohlížeče. Vyberte vhodné algoritmy a benchmarkové úlohy, které naimplementujete ve vlastním prostředí. Proveďte sérii experimentů, ve kterých doložíte funkčnost vaší implementace a vyhodnoťte implementované algoritmy na úlohách jak z pohledu kvality dosaženého řešení, tak z pohledu rychlosti konvergence. Dbejte na rozšiřitelnost a dokumentaci prostředí, a zveřejněte svou práci jako open source.

Elektronicky schválil/a Ing. Magda Friedjungová, Ph.D. dne 8. listopadu 2023 v Praze.

Bakalářská práce

EFEKTIVNÍ IMPLEMENTACE NEUROEVOLUCE PRO ÚLOHY POSILOVANÉHO UČENÍ

Vladimír Votava

Fakulta informačních technologií
Katedra aplikované matematiky
Vedoucí: doc. Ing. Pavel Kordík, Ph.D.
16. května 2024

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2024 Vladimír Votava. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Votava Vladimír. *Efektivní implementace neuroevoluce pro úlohy posilovaného učení.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Poděkování vi

Prohlášení vii

Abstrakt viii

Seznam zkratek ix

Úvod 1

1 Teoretický základ 2

1.1 Učení s učitelem . . . 2
1.2 Posilované učení . . . 2
1.2.1 Základní prvky . . . 2
1.2.2 Množina problémů . . . 3
1.2.3 Využívání vs prozkoumávní . . . 3
1.3 Hluboké učení . . . 3
1.3.1 Struktura neuronových sítí . . . 3
1.3.2 Algoritmy hlubokého učení . . . 4
1.4 Evoluční algoritmy . . . 4
1.4.1 Proces učení v evolučních algoritmech . . . 4
1.5 Neuroevoluce . . . 5
1.5.1 Základní prvky neuroevoluce . . . 6
1.5.2 Aplikace neuroevoluce na paradigmata strojového učení . . . 6
1.5.3 Relevantnost neuroevoluce . . . 7
1.5.4 Kódování genotypu . . . 8
1.5.5 Konvenční neuroevoluce . . . 8
1.5.6 NEAT . . . 8
1.6 API . . . 9
1.7 Javascript . . . 9
1.7.1 Typescript . . . 10
1.7.2 Výkon . . . 10
1.8 C++ . . . 10
1.9 WebAssembly/WASM . . . 11
1.9.1 Emscripten . . . 11
1.10 Web Workers . . . 11
1.11 Herní engine . . . 12
1.12 Pomocné nástroje . . . 12
1.12.1 Vite . . . 12
1.12.2 Vue . . . 13
1.12.3 Docker . . . 13

2 Návrh řešení a komparativní analýza 14
2.1 Motivace . . . 14
2.2 Návrh řešení . . . 15
2.2.1 Využité technologie . . . 15
2.2.2 Rozhraní . . . 15
2.2.3 Proces učení . . . 16
2.3 Existující řešení . . . 16
2.3.1 Brain.js . . . 17
2.3.2 Neataptic.js . . . 17
2.3.3 TensorFlow.js . . . 17

3 Implementace knihovny 18
3.1 ELGINE . . . 18
3.1.1 Sdílený stav . . . 18
3.1.2 Controls.ts . . . 18
3.1.3 Entity.ts . . . 19
3.1.4 Elgine.ts . . . 19
3.2 NE-WASM . . . 22
3.2.1 Neuronová síť . . . 22
3.2.2 Šablona pro algoritmus neuroevoluce . . . 23
3.2.3 Třída pro export do WebAssembly . . . 25
3.2.4 Implementace WebAssembly do Typescriptu . . . 26
3.3 Jádro . . . 29
3.3.1 Příprava pro implementaci prostředí . . . 29
3.3.2 Zpracovávání paralelního požadavku na výpočet . . . 31
3.3.3 Třída AsmNeGym - běh neuroevoluce . . . 31
3.3.4 Třída AsmNeGym - API . . . 34
3.3.5 Třída AsmNeGym - běh genotypu . . . 35
3.4 Shrutí architektury . . . 35

4 Implementace benchmarkových úloh a neuroevolučních algoritmů 38
4.1 Flappy bird . . . 38
4.2 Závodní auto . . . 39
4.3 Konvenční neuroevoluce (CNE) . . . 40
4.3.1 Kódování . . . 41
4.3.2 Generování nové generace . . . 41
4.4 NEAT . . . 42
4.4.1 Kódování . . . 42
4.4.2 Generování nové generace . . . 43
4.5 Testovací stránka . . . 43

5 Výsledky a evaluace 45
5.1 Ověření funkčnosti řešení . . . 46
5.2 Otestování akcelerace paralelizací výpočtu . . . 46
5.3 Experimentování s velikosti sítě algoritmu CNE . . . 47
5.4 Experimentování s hyperparametry mutace algoritmu NEAT . . . 49
5.5 Porovnání implementovaných algoritmů NEAT a CNE . . . 49

6 Závěr 52

Obsah příloh 56

Seznam obrázků

1.1 Klasifikace metod optimalizace. Převzato z [5] 5
1.2 Proces evolučního algoritmu. Převzato z [5] 6
1.3 Křížení na odlišných neuronových sítích odhadující stejnou funkci. Převzato z [8] 7
1.4 Křížení dvou genotypů pomocí historických značek. Převzato z [8] 9

2.1 Diagram návrhu knihovny 16

3.1 Propojení jednotlivých částí knihovny 35

4.1 Benchmarková úloha flappy bird 39
4.2 Benchmarková úloha závodní auto 40
4.3 Ukázka fenotypu z CNE 41
4.4 Ukázka fenotypu z NEAT 42
4.5 Ukázka demo stránky 44

5.1 Experiment 1 46
5.2 Experiment 3 48
5.3 Experiment 4 50
5.4 Experiment 5 51

Seznam tabulek

5.1 Srovnání výkonu podle počtu jader 47

Seznam výpisů kódu

3.1 Controls.ts 19
3.2 Entity.ts 20
3.3 Elgine.ts 21
3.4 ENeuronType.hpp 22
3.5 Utils.hpp 22

3.6 CNetwork.hpp 23
3.7 CNeuroevolutionBase.hpp 24
3.8 AsmCore.cpp 27
3.9 Makefile 28
3.10 asmcore.d.ts 28
3.11 AsmNeGym.ts:AsmCore 28
3.12 AsmNeGym.ts:Vector 29
3.13 Environment module 30
3.14 AsmNeUtils.ts 30
3.15 AsmNeWorker.ts 32
3.16 AsmNeGym.ts:train 33
3.17 AsmNeGym.ts:API 34
3.18 AsmNeGym.ts:run 36

Chtěl bych poděkovat především vedoucímu bakalářské práce, doc. Ing. Pavel Kordík, Ph.D. za to že vedl mojí práci i přesto, že jsem si téma zvolil sám. Zároveň chci poděkovat svojí partnerce, která mi dělala oporu při psaní bakalářské práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

Abstrakt

Bakalářská práce má za cíl vytvořit knihovnu, která umožňuje běh neuroevolučních algoritmů v prostředí webového prohlížeče. Ani neuroevoluce ani strojové učení nejsou v prostředí webového prohlížeče moc populární. Problém spočívá ve výkonu, monotónnosti jazyků a dalších faktorech. Rešerše přinese návrh řešení umožňující efektivní vývoj neuroevolučních algoritmů v prostředí webového prohlížeče. K tomu především využijeme technologie jako WebAssembly a Web Workers.

Výsledná knihovna umožňuje vývojáři efektivně vyvíjet nové algoritmy neuroevoluce. Zároveň má knihovna k dispozici jednoduchý engine, ve kterém si mohou vývojáři implementovat vlastní modely prostředí, aniž by se museli starat o běh v prostředí JavaScriptu / webového prohlížeče. V rámci knihovny jsou také implementovány dva algoritmy (NEAT a HyperNEAT) a dvě prostředí v podobě jednoduchých her. Implementace knihovny byla na těchto prostředích otestována, čímž byla doložena její funkčnost a data o rychlosti konvergence. Knihovna je koncipována jako open-source projekt s kvalitní strukturou a dokumentací. Díky tomu může být dále rozšiřována a vylepšována.

Klíčová slova posilované učení, neuroevoluce, webové technologie, WebAssembly, C++, JavaScript, vývojová knihovna, profilovací nástroje, benchmark

Abstract

The Bachelor's thesis aims to create a library that allows the running of neuroevolutionary algorithms in a web browser environment. Neither neuroevolution nor machine learning are very popular in the web browser environment. The problem lies in performance, the monotony of languages, and other factors. The literature review will bring a solution design that enables efficient development of neuroevolutionary algorithms in a web browser environment. To this end, technologies such as WebAssembly and Web Workers will be primarily utilized.

The resulting library enables developers to efficiently develop new neuroevolution algorithms. The library also has a simple engine available, in which developers can implement their own environmental models without having to worry about running in the JavaScript/web browser environment. Within the library, two algorithms (NEAT and HyperNEAT) and two environments in the form of simple games are also implemented. The implementation of the library was tested on these environments, thereby demonstrating its functionality and data on the speed of convergence. The library is conceived as an open-source project with a quality structure and documentation, allowing it to be further expanded and improved.

Keywords reinforcement learning, neuroevolution, web technologies, WebAssembly, C++, Javascript, development library, profiling tools, benchmark

Seznam zkratek

AI	Umělá Inteligence (Artificial intelligence)
WASM	WebAssembly

Úvod

V dnešní době je AI všudypřítomné a konkrétně umělé neuronové sítě jsou velmi nadějné, co se budoucnosti AI týče. Díky hlubokému učení máme různé algoritmy, jak tyto sítě nakonfigurovat, aby splnily úkol, který jim zadáme.

Neuroevoluce je jedna z těchto metod a díky evolučním algoritmům v kombinaci s neuronovými sítěmi je to silný nástroj, který může řešit celou řadu úkolů. Nejedná se o revoluční technologii, ale díky svojí specifické formě a stylem, ve kterém funguje, se v konkrétních případech stále jedná o relevantní odvětví hlubokého učení neuronových sítí.

Pro jejich vývoj, implementaci a testování existují sofistikované a zajímavé nástroje, které poskytují celou řadu funkcionalit a předpřipravených dílčích částí, které usnadňují práci. Problém je, že malé procento z nich se věnuje implementaci v prostředí webového prohlížeče a ty, které existují, mají jisté nedostatky. Proto se ve své práci budu věnovat analýze neuroevoluce a moderním webovým přístupům a technologiím, které jsou v dnešní době k dispozici. Pomocí toho vytvořím nástroj/knihovnu, která bude spravovat běh neuroevoluce a zároveň bude efektivně řešit problémy, co sebou neuroevoluce přináší.

Moje bakalářská práce si tedy klade za cíl rozšířit dostupnost neuroevoluce i do webových technologií a díky tomu umožnit vývojářům/výzkumníkům více vyvíjet, implementovat a testovat nové metody neuroevoluce nebo zrychlit a umožnit jednoduché trénování sítě pro vlastní úlohy.

Cíl

Cílem teoretické části tedy bude vymezení pojmů týkající se posilovaného učení a neuroevoluce. Zároveň budu analyzovat různé webové technologie, které mohou dát výhodu při implementaci a výpočtu. Zanalyzuji všechny informace a v rámci toho budu hodnotit kritické a problematické části neuroevolučních algoritmů a prozkoumávat možnosti, které webové technologie nabízejí k řešení těchto problémů. Pro účely výsledného testování prozkoumám možné úlohy, které jsou vhodné pro doložení funkčnosti mého řešení. Zadefinuji konkrétní neuroevoluční algoritmy, které budou dostatečně rozličné pro ukázku obecnosti knihovny.

Cílem praktické části bude samotná implementace knihovny. Ta bude řešit spojení webového prostředí, algoritmu a simulace, na které se algoritmus bude učit. V rámci toho udělám návrh celé aplikace a implementace dílčích částí. Konkrétně tedy rozhraní pro implementaci vlastního prostředí, rozhraní pro implementaci algoritmů a jádra, které se bude starat o propojení těchto dvou rozhraní a bude za uživatele řešit většinu práce.

Pro testování výsledné knihovny navrhnu dvě testovací prostředí a vyberu dva neuroevoluční algoritmy, které do knihovny implementuji. Zároveň kromě základních požadavků do knihovny navíc zakomponuji nástroje pro lepší ladění. Z výsledků udělám porovnání a zároveň vyhodnotím celkovou úspěšnost a funkčnost mého řešení.

Kapitola 1
Teoretický základ

1.1 Učení s učitelem

Ve strojovém učení existují tři základní paradigmata. Učení s učitelem („supervised learning"), učení bez učitele („unsupervised learning") a posilované učení („reinforcement learning"). Učení s učitelem je nejrozšířenější a nejpoužívanější paradigma. Funguje na principu toho, že jsou vstupní data spojené s očekávanou výstupní hodnotou. Cílem algoritmů tohoto odvětví je pak najít takovou funkci, která bude mít největší úspěšnost ve správném určování výstupu odhadované funkce vůči očekávanému výsledku.

Jako názorný příklad může být rozpoznávání psaných čísel z obrázků. Vstup bude 1024 pixel hodnot z šedotonního obrázku (rozlišení 32×32). Jako výstup budou použity číslice 0-9, podle toho, jaká číslice se na obrázku vyskytuje. Hlavní rys učení s učitelem je tedy fakt, že je zde očekávaný výsledek vstupu. Díky tomu je algoritmus schopný říct, jestli se rozhodl správně nebo špatně. [1]

1.2 Posilované učení

1.2.1 Základní prvky

Tzv. prostředí, je nějaký model s definovanou sadou pravidel a stavem, kde se následně stav mění v čase podle pravidel. Do takového modelu není problém přidat míru pravděpodobnosti, nebo proměnlivost prostředí v čase.

Do tohoto modelu se následně vloží tzv. agent, který má možnost dělat nějaké akce a díky tomu ovlivňovat prostředí. Z tohoto prostředí má možnost „vnímat" jeho stav. Tento popis může připomínat život člověka. Žijeme v nějakém prostředí a za agenta lze považovat člověka. Jako prostředí bude naše planeta. Člověk může pozorovat stav prostředí přes smysly jako je zrak, čich a hmat. Člověk kromě pozorování může dělat nějaká akce, jako je například je kopnutí do míče. Jakmile se tak stane, tak může pozorvat jak míč letí. Touto akcí jsme tak ovlivnili naše prostředí.

Konfigurace agenta se nazývá strategie (policy). Strategie pak říká, jakou akci má agent zvolit podle vjemů z prostředí v daný okamžik. Je dobré zmínit, že agent nemusí nutně znát celé prostředí. Naopak hodně problémů, které řešíme v posilovaném učením ani tuto možnost neposkytují. Tohle si jde představit na lidech. Můžeme se vrátít k příkladu s míčem. Pokud míč přeletí za plot, tak člověk ztratí informaci o tom, co se s míčem stalo. Přitom nezmizel, pouze je pozorování prostředí omezeno na určitou oblast.

Další důležitý koncept je tzv. fitness funkce. Tato funkce přiřazuje ke stavu agenta nějaké číslo, které obecně vyjadřuje hodnocení agenta. Tudíž čím vyšší toto číslo je, tím víc se agentovi daří.

Tohle je stěžejší a důležitý koncept. Je stejně tak důležitý jako je prostředí nebo agent. Později pak tento koncept využijeme k tomu, abychom určili, která rozhodnutí byli dobrá a která špatná. [2]

1.2.2 Množina problémů

S těmito prvky už jde definovat, o co v posilovaném učení jde. Je zde prostředí a agent, který nemá správně nastavenou strategii. To je logické, jelikož pokud bychom věděli, jak jí správně nastavit, tak bychom neměli žádný problém. Cílem posilovaného učení je tedy nastavit strategii agenta tak, aby se maximalizovala hodnota z hodnotíci funkce, Lajcky řečeno se snažíme naučit agenta rozhodovat v prostředí tak, aby měl co nejlepší výsledky.

Algoritmy posilovaného učení využívají hlavně zpětné vazby na akci, kterou agent zvolil, a to v podobě hodnoty z fitness funkce. Není tedy potřeba přesně znát prostředí a ani způsob podle kterého se počítá fitness funkce. Naopak cílem je navrhnout řešení/algoritmus, který zjistí optimální konfiguraci strategie agenta místo manuálního nastavení. Pokud jsou tedy k dispozici nějaké vjemy z prostředí a lze zhodnotit akci agenta, tak v tu chvíli je možné aplikovat posilované učení.

K vyřešení problému tedy stačí mít prostředí, agenta a fitness funkci. O vše ostatní se postará algoritmus, který aplikujeme. Z tohoto popisu mohou být jasné jisté limitace tohoto přístupu. Pokud nějaký z těchto základních prvků chybí, tak nejsme schopni tímto způsobem problém řešit. Pokud například nebudeme schopní vytvořit fitness funkci, tak algoritmus nemá z čeho brát zpětnou vazbu a tudíž zjistit, jestli je akce dobrá, nebo ne. Bez prostředí nebo agenta je jasné, že problém nelze řešit vůbec. Ale například nedostatečná informace z prostředí, může být také důvodem, proč se nemusí dojít k uspokojivému výsledku. Pokud bychom například chtěli agenta naučit ovládat auto, ale dali mu k dispozici pouze předpověď počasí, tak se pravděpodobně nikdy nenaučí toto auto řídit. [2]

1.2.3 Využívání vs prozkoumávní

Důležitým aspektem je tzv. „využívání vs prozkoumávání“ (exploitation vs exploration). Toto dilema spočívá v tom, že u agent je potřeba maximalizovat výstup z fitness funkce a je tedy potřeba aby zkoušel nové přístupy v dané problematice. Zároveň je chtěné, aby využil už nabytou znalost k maximalizaci fitness funkce. Otázka tedy zní, jak správně nastavit poměr využívání nabytých znalostí a jak moc prozkoumávat nové přístupy.

Tímto problémem se dále budu zabývat v sekci 1.5, kde se ukáže, jak konkrétně tento problém řeší neuroevoluční algoritmy. Obecně ale platí, že je potřeba spíše využívat již nabyté znalosti s menší mírou prozkoumávaní. Jelikož při velké míře prozkoumávání se může zkazit již kvalitní konfigurace. [2]

1.3 Hluboké učení

Hluboké učení („deep learning“) je odvětví strojového učení, které k aproximování funkce používá neuronové sítě. V dnešní době se jedná o nejpopulárnější a nejúspěšnější způsob, ke kterému strojové učení směřuje. Její základy jsou inspirované lidským mozkem. [3]

1.3.1 Struktura neuronových sítí

Neuronová síť se skládá z neuronů, které mají vstupy a výstupy. Neuron má aktivační funkci, pomocí které je daný neuron buď aktivovaný, nebo deaktivovaný. Běžné aktivační funkce jsou například sigmoid funkce (nebo logistická funkce), Tanh (hyperbolický tangens) nebo ReLU

(Rectified Linear Unit). Spojení mezi neurony se nazývá synapse. Ta má na obou koncích neuron. Jeden vstupní a jeden výstupní. Zároveň má přiřazené nějaké číslo, kterému se říká váha. Tato váha se pronásobí hodnotou, kterou nabývá vstupní neuron a pošle se jako vstup do vstupního neuronu.

Neuronová síť je potom skupina neuronů, co jsou propojeny pomocí synapsí. V neuronové sítí rozdělujeme neurony podle jejich účelu. Vstupní neurony jsou ty, které jako vstup nemají synapse, ale vstupní parametry neuronové sítě.

Zatímco výstupní jsou naopak ty, které svůj vstup nepropagují dál do synapsí, ale jsou jako výstup neuronové sítě. Všechny ostatní neurony (tedy ty, které přijímají vstup ze synapsí a výstup propagují přes synapse do dalších neuronů) nazýváme skryté neurony. Díky nim jsme schopní aproximovat složitější funkce.

Proces, při kterém se vypočítá výstup sítě ze vstupních parametrů, se nazývá dopředné šíření. Při tomto procesu se do vstupních neuronů vloží vstupní parametry. Následně každý neuron spočítá pomocí aktivační funkce, jaký výstup bude mít. Tento výsledek se následně propaguje dál pomocí všech synapsí, co jsou s tímto neuronem spojené. Synapse tedy pronásobí výsledek již spočítaného neuronu svojí váhou a následně ho pošle jako vstupní parametr do výstupního neuronu této synapse. Jakmile jsou všechny vstupy neuronu spočítané, tak se sečtou a vloží se do aktivační funkce. Výsledek z ní se stejně jako předtím propaguje dál. Tento proces se zastaví ve chvíli, kdy se dojde až k výstupním neuronům, které už jsou výsledkem dopředného šíření. Lze si to představit jako funkci, kde vstupní parametry funkce jsou čísla, které jsou předány vstupním neuronům a výstup této funkce jsou hodnoty, co jsou ve výstupních neuronech. [3]

1.3.2 Algoritmy hlubokého učení

Jak je vidět z popisu neuronové sítě, tak je všechno dané, kromě vah mezi neurony a struktury sítě (tudíž kolik neuronů se použije a jakou budou mít strukturu). Přesně o tom je hluboké učení. Podle teorému o univerzální aproximaci [4] se pomocí neuronových síti, která má jednu a víc skrytých vrstev, může aproximovat jakákoliv funkce. Díky tomu je hluboké učení využívané ve všech paradigmatech strojového učení. V učení s učitelem je to například algoritmus zpětného vyhledávání (backtracking), který se pomocí diferenciálního výpočtu snaží zjisti správné nastavení vah. V posilovaném učení je to například algoritmus deep q-learning, který jako vstup bere vjemy agenta a jako výstupní neurony jsou akce, které si agent může zvolit. [3]

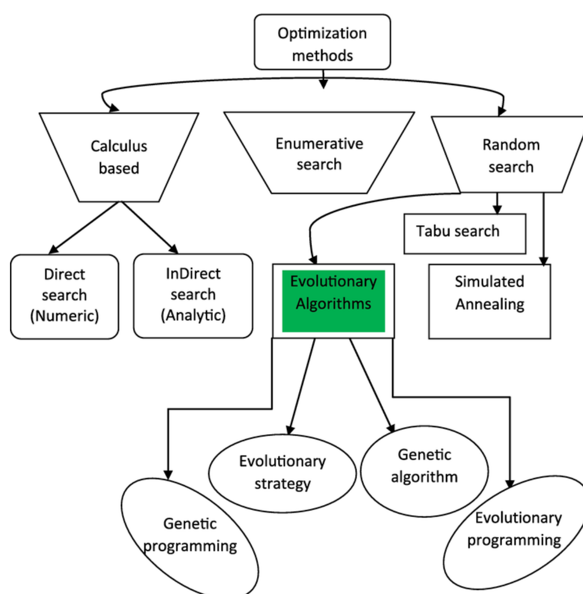
1.4 Evoluční algoritmy

Evoluční algoritmy jsou optimalizační metody z rodiny metod náhodného prohledávání, viz 1.1. Tento způsob je silně inspirovaný evolucí v přírodě. Základní princip spočívá v tom, že je nějaká optimalizační úloha.

Evoluční algoritmus na počátku dostane náhodné řešení úlohy (například v podobě konkrétních parametrů). Každý z těchto řešení je reprezentování jako genotyp. Následně se spočítá fitness funkce a každý genotyp se ohodnotí. Skupina genotypu se nazývá populace a proto se na jeden genotyp občas referuje jako jedinec. [5]

1.4.1 Proces učení v evolučních algoritmech

Při procesu selekce se následně vyberou ty jedinci, co měli nejvyšší hodnotu z fitness funkce. Selekce zajistí, že se zachovají pouze ty genotypy, které dávají nejlepší výsledky, a tudíž zvyšují pravděpodobnost, že další generace bude ještě lépe adaptována na požadovanou úlohu. Tento proces se liší podle použitého algoritmu. Některé berou několik nejlepší jedinců. Jiné zase přiřadí podle hodnoty fitness funkce každému genotypu šanci, že bude v procesu selekce vybrán. Vybraní jedinci jsou poté použiti k tvorbě nové generace. Tyto genotypy se nazývají rodiči.



Obrázek 1.1 Klasifikace metod optimalizace. Převzato z [5]

Nejdříve se provede křížení, kde se vyberou určité prvky z rodičů. Díky tomu dochází ke kombinaci genetických informací, což umožňuje vznik nových jedinců, kteří mohou nést různorodé a potenciálně výhodné kombinace vlastností obou rodičů. Tento proces přispívá ke genetické diverzitě populace a je klíčovým mechanismem pro inovace a adaptaci v evolučních algoritmech.

Po křížení a generování nových jedinců se provede mutace. To funguje tak, že se s určitou mírou náhody upraví jednotlivé geny v novém genotypu u náhodně vybraných potomků. To znamená, že se upraví dílčí části genotypu náhodnou hodnotou. Pokud by se mutace vynechala a použil by se pouze proces křížení, tak by nebylo možné mít jiné geny, než jsou v populaci přítomné. Mutace tedy přidává další možnost inovace. Mutace ale nemusí mít vždy pozitivní výsledek, jelikož nově příchozí inovace při mutaci může jedinci uškodit a snížit mu hodnotu fitness funkce. Nastáva otázka, jaká by měla být míra mutace.

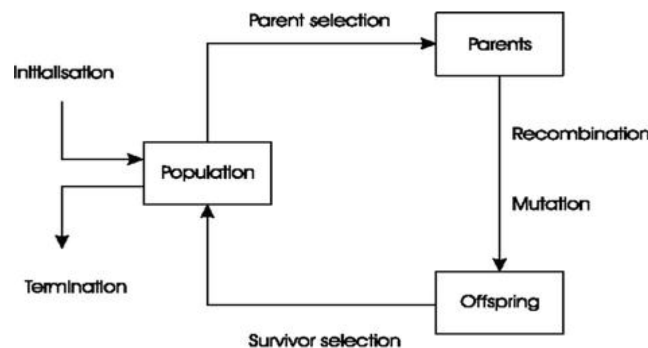
Při generaci nové populace (tzv. generace) se provede nejdřív selekce, křížení a mutace. Každý jedinec je následně ohodnocen fitness funkcí a proces se opakuje. Zastavit se může podle času, počtu iterací anebo podle dostatečné fitness funkce. Celý proces je navržený tak, aby zachovával nejlepší geny a genotypy, ale zároveň do řešení přidal i jistou míru inovace. Míra mutace je tzv. hyperparametr, který se může nastavit, stejně jako velikost populace. To může silně ovlivnit, jak se bude proces učení vyvíjet. [5]

Celý proces je znázorněný v 1.2

Tato metoda je tzv. black-box přístup, což znamená, že evoluční proces neví o jaký problém se jedná. Ale přesto dokáže optimalizovat řešení. Stačí k tomu fitness funkce a umět nad řešením provádět operaci mutaci. Takže stačí genotyp, ze kterého se vytvoří síť a následně se vrací jedno číslo, což je hodnota fitness funkce. Z matematického pohledu se dá říct to, že optimalizujeme funkci $f(w)$, kde w je vektor hodnot, pro sestavení sítě a hodnota z této funkce vyjadřuje skóre. Nic jiného se o funkci neví a ani neočekává. Proto black-box přístup (přístup černé skřínky). [6]

1.5 Neuroevoluce

Neuroevoluce je metoda, kdy se kombinuje metoda neuroevolučního algoritmu a neuronových sítí. Jako genotyp se použije neuronovou síť, která se bude následně optimalizovat pomocí neuroevoluční algoritmu. K tomu aby se mohla na nějaký problém aplikovat neuroevolce stačí hodnotící



Obrázek 1.2 Proces evolučního algoritmu. Převzato z [5]

funkce, která bude jedincům přiřazovat jejich skóre. To se udělá tak, že se všichni agenty spustí na instanci problému, který problém řeší a fitness funkce následně vrátí jejich skóre. [7]

1.5.1 Základní prvky neuroevoluce

Teď popíšu jednotlivé prvky evolučního algortimu a jak se s nimi pracuje v neuroevoluci. Tohle pomůže s následným návrhem architektury knihovny, jelikož je potřeba znát obecné rysy, které se musí implementovat.

Jak již bylo řečeno, tak genotyp je v neuroevoluci reprezentován jako neuronová síť. V sekci 1.3 bylo zmíněno, že v neuronových sítích je úkolem optimalizovat váhy synapsí a architekturu sítě, pokud to konkrétní algoritmus umožňuje. Genotyp jsou tedy neurony a jejich spojení. Zároveň pokud je potřeba, tak se mohou zvolit i další parametry do genotypu, jako je aktivační funkce.

Samotný genotyp je pouze informace o tom, jak má výsledná síť vypadat. Fenotyp je pak sestavná síť, pomocí které už jde dělat algoritmus dopředného šíření a vyhodnotit fitness funkci a tím pádem i ohodnotit danou strukturu. Mezi genotypem a fenotypem i většiny algoritmů není značný rozdíl, ale v sekci 1.5.4 je ukázka toho, že se mohou značně lišit, pokud budeme používat nepřímé kódování.

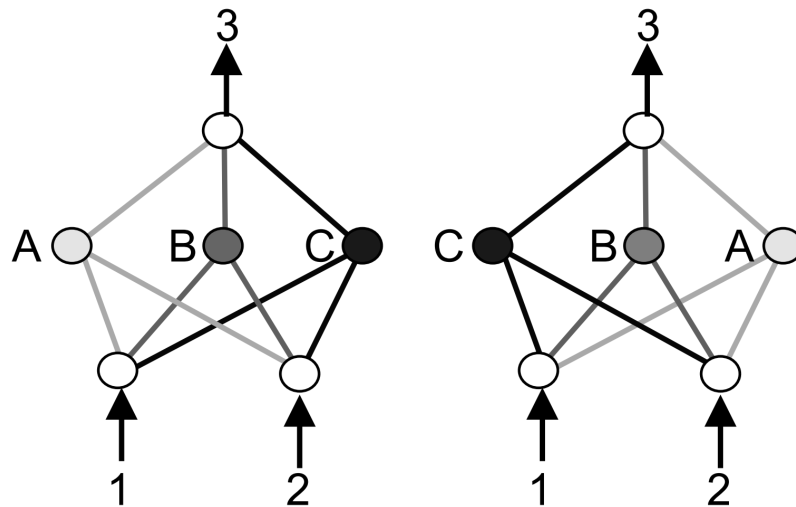
Selekce je přímočará a stejně jako v popisu evolučních algoritmů stačí vybrat jedince, co mají nejlepší hodnotu fitness funkce.

Křížení lze u neuronových sítí dokázat tak, že se budou střídavě brát váhy z rodičů. Problém ale nastává tehdy, když neuroevoluce optimalizuje architekturu sítě. V tu chvíli už nelze vzít váhu z rodiče na synapsi, pokud se zároveň nevzala i architektura z toho samého rodiče. Vznikaly by totiž případy, kdy se převezme synapse a její váha z jednoho rodiče, ale ve výsledné architektuře budou chybět neurony, které tato synapse spojuje. To kvůli tomu, že druhý rodič je mít nemusí. Další problém je, že různé neuronové sítě, mohou simulovat stejnou funkci. Na obrázku 1.3 jsou vidět dvě sítě, které jsou zrcadlově stejné a odhadují stejnou funkci. Pokud by se první neuron vzal z první neuronové sítě a třetí neuron z druhé sítě, tak vznikne úplně jiná síť, a tudíž tento typ křížení úplně zkazí požadovaný výsledek a jenom dvě z šesti možných kombinací je chtěná. [8]

Mutace v neuroevoluci je dost specifická pro každý algoritmus. Ale obecně platí, že mění váhy v synapsích mezi neurony. Některé algoritmy, které zároveň optimalizují i architekturu sítě, mohou pomocí mutace provádět operace jako přidávání nových neuronů a přidávání/odstraňování synapsí mezi neurony. [7]

1.5.2 Aplikace neuroevoluce na paradigmata strojového učení

Otázkou může být, zda neuroevoluce patří do paradigmatu učení s učitelem nebo do posilovaného učení. Odpovědí je že tato metoda může být použita na oba přístupy. Pokud se aplikuje přístup



Obrázek 1.3 Křížení na odlišných neuronových sítích odhadující stejnou funkci. Převzato z [8]

na problém učení s učitelem, tak lze pomocí neuronové sítě modelovat funkci pro predikci. Díky tomu, je známay správný výstup, tak lze ohodnotit jednotlivé jedince v generaci. Dále se může pokračovat podle běžného postupu neuroevoluce.

Pokud bude neuroevoluce v podobě posilovaného učení, tak strategie agenta bude reprezentovaná jako neuronová síť, které bude přijímat vjemy jako vstup a akce agenta jako výstup. Sice se nebude upravovat strategie agenta při běhu v prostředí, jako to některé algoritmy dělají, ale zaznamenávat se bude až výsledné skóre. Díky tomu lze přiřadit agentovi skóre, které získá podle toho, jak dobře si vedl v řešení úkolu a pak pomocí neuroevolučního algoritmu vytvořit novou generaci. [7]

1.5.3 Relevantnost neuroevoluce

V článku [6] se výzkumníci z OpenAI věnují relevantnosti evolučních strategiím v komparaci s tradičními metodami posilovaného učení. Evoluční strategie jsou podmnožina evolučních algoritmů jak to je vidět v obrázku 1.1 Neuroevoluce a evoluční strategie si jsou podobné a fakta, která článek zmiňuje jsou schodná i pro neuroevoluci.

Není potřeba algoritmus zpětné propagace. Ten se používá v neuronových sítí, pro konfiguraci parametrů. Funguje na bází zpětného procházení a zjišťování částečného gradientu a díky tomu upravuje váhy na synapsích. K tomu je potřeba vypočítat hodnotu derivace aktivační funkce pro všechny neurony. Jelikož tento proces v neuroevoluci není, tak je kód menší a 2-3x rychlejší. Zároveň je méně náročný na pamět, jelikož si není potřeba pamatovat tolik informací z prostředí (například historie akcí).

Neuroevoluce je oproti klasickým metodám posilovaného učení vysoce paralelizovatelná. Jelikož není potřeba vyhodnocovat jedince z populace postupně, tak jde distribuovat výpočet do několika na sobě nazávislých procesů. Zároveň se při paralelizaci musí překopírovávat data z jednoho procesu do druhého. To vytvoří režii navíc. Data které jsou potřebná překopírovat u neurovoluce je pouze genotyp sítě a následně hodnota fitness funkce při dokončení simulace prostředí. U tradičních metod, kde se drží historie stavů, nebo procesů a musí se synchronizovat data mezi sebou, to může představovat obtížný úkol.

Menší senzitivita vůči hyperparametrům patří mezi další výhody. U metod jako je DQN [9] je potřeba precizně nastavit hyperparametry, jako je „frame-skip“. Výsledek může být zcela jiný v zásvislosti na takovýchto parametrech. Neuroevoluce je vůči tomuto odolnější a při malé změně se sice může snížit konvergence k optimálnímu řešení, ale výsledek se v podstatě nezmění.

Článek taky kromě již zmíněného ukazuje srovnání řešení stejných úloh pomocí evolučních strategií a posilovaného učení. Výsledky jsou srovnatelné a mělo by vypovídat o tom, že evoluční strategie jsou pořád relevantní. [6]

1.5.4 Kódování genotypu

S neuronovou sítí, která je uložená někde v paměti jako kód, nelze jen tak jednoduše manipulovat. Proto existuje genotyp, podle kterého se síť generuje. Ten je většinou reprezentován, jako řetězec. Ten jde jednoduše exportovat, přenášet a ukládat.

Přímočaré řešení je navrhnout strukturu řetězce, ze které budou dostupné všechny informace o neuronové sítí. Což znamená mít informaci o všech neuronech, jejich typy (vstupní, výstupní, skryté) a aktivační funkce. Dále se musí držet informace o synapsích a jejich vah. Tomuto přístupu se říká přímé kódování a používá ho většina neuroevolučních algoritmů. Jeho výhoda je přímočarost a jednoduchost.

Druhý přístup je nepřímé kódování. Při takovém přístupu genotyp o výsledné neuronové sítí nic neříká. Genotyp totiž reprezentuje jinou strukturu a až ta následně generuje finální neuronovou síť. [10]

1.5.5 Konvenční neuroevoluce

Konvenční neuroevoluce je základní algoritmus, jak provádět neuroevoluci. Tento nesnaží se nesnaží chytře aplikovat neuroevoluci, ale pouze jí stroze implementovat.

Struktura neuronové sítě je předem určená a nijak se v průběhu nemění. Podle hyperparametrů se vytvoří příslušný počet skrytých vrstev a v nich příslušný počet neuronů. Propojení je napříč všemi neurony v sousedících vrstvách. Váhy jsou inicializovány náhodně a mají přesně dané pořadí. To může zjednodušit kódování a zároveň usnadnit křížení. Jelikož jsou váhy v přesném pořadí, tak při křížení je dané, které váhy vybírat. Při selekci se vezme pouze určité procento nejlepších jedinců. Z nich se následně náhodně vybírá a generuje nová generace.

Tento algoritmus je spíše pro demonstrací funkčnosti a dokáže vyřešit jednoduché problémy. [11]

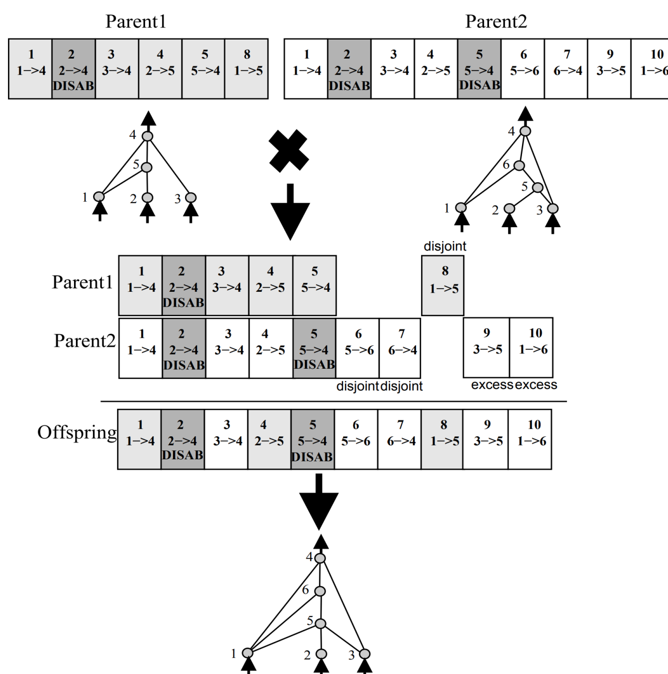
1.5.6 NEAT

„Neuroevoluce rozšiřujících topologií“ (Neuroevoluce rozšiřujících topologií) neboli NEAT, je velmi úspěšný neuroevoluční algoritmus z roku 2002. Tento algoritmus stejně jako ostatní upravuje váhy díky křížení genotypů a mutacemi. Zároveň popisuje několik principům a díky nim budeme moct efektivně trénovat neuronovou síť.

Kromě úprav vah na synapsích tento algoritmus upravuje strukturu sítě. To znamená, že se struktura sítě stejně jako váhy sapsí vyvijí společně. Prí náhodné mutaci je totiž šance, že se přidá nový neuron, nebo nová synapse. Díky tomu se nemusí předem definovat, jak má vypadat struktura sítě a ta se v průběhu sama upravuje.

To ale přináší problém, jelikož dvě různé nemusí být kompatibilní pro křížení. Tento problém byl již avizován v sekci 1.5.1. NEAT to řeší přes zavedení tzv. historických značek. Díky nim má každý úprava na struktuře sítě (přidání synapse, nebo neuronu) unikátní číslo. Toto číslo se generuje s globálního počítadla, které nelze zmenšit. Zároveň již přidaná změna struktury nejde smazat. Proto NEAT zavadí invalidační parametr, který vypne genom s příslušnou historickou značkou.

Tohle umožňuje jednoduché a přímočaré křížení. Stačí pouze vzít dva genotypy a namapovat geny se stejnou historickou značkou. Poté se jen náhodně vybere, jestli se má vzít gen z prvního nebo druhého genotypu. Může však nastat situace, kdy se na jednom genotypu vyvine víc genů, než na druhém. Disjunktní neurony jsou ty, které se nachází mezi společnými geny a přebytečné



Obrázek 1.4 Křížení dvou genotypů pomocí historických značek. Převzato z [8]

jsou ty, které jsou navíc na konci genotypu. Tyto speciální neurony se budou vždy brát z rodiče, který má lepší hodnotu fitness funkce. Celý tento proces je ukázaný na obrázku 1.4.

Algoritmy které kromě vah synapsí upravují zároveň strukturu sítě, nastavovali počáteční architekturu sítě náhodně. To přináší problém velké dimenzionality. To znamená, že se konfigurovalo velké množství vah. NEAT používá přístup tvz. minimální strukturu. Počáteční struktura neobsahuje žádné skryté neurony (tudíž zbydou pouze vstupní a výstupní neurony). Tento přístup poté postupně staví komplexnější architekturu sítě pomocí přidávání dalších neuronů. Díky tomu se sníží dimenzionalita, velikost genotypu. Natrénovaná síť je menší, než pokud by se začalo s náhodnout strukturou. [8]

1.6 API

API je zkratka pro aplikační programové rozhraní (application programming interface). Takové rozhraní pak slouží na propojení programů. Zároveň zaručuje, že všechno ostatní co se v programu děje, bude fungovat tak jak má. Díky tomu vývojář, který využívá nějaký program nemusí znát jak interně funguje.

Vystavené API autorem programu je většinou oproti interním částem programu zdokumentovano. To ještě více zjednoduší práci vývojářům, kteří API využívají.

API se nemusí vztahovat pouze na komunikaci mezi programy, ale často se používá i při komunikaci v dílčích částech kódu, knihoven a nebo celých aplikacích. [12]

1.7 Javascript

Javascript je programovací jazyk z roku 1995 a je používaný na 99 % webových stránek. Jedná se o skriptovací jednovláknový jazyk s dynamickým typováním. Javascript se převážně používá

pro skriptování na webových stránkách a používá ho 99 % webových stránek[1].

Jeho účel byl zaměřen původně na manipolaci s html prvky na webu. Postupem času s optimalizací interpretů a lepším výkonem počítačů se začal používat pro kompletní ovládání obsahu a interaktivitu stránky. K tomu přispěl velmi výkonný interpretor od společnosti google V8[2].

Jelikož je Javascript dynamický jazyk, tak mohou nastat chyby s nekompatibilnímy typy. Tohle ještě více zhoršuje fakt, že Javascript je skriptovací jazyk a chyba se objeví, až při spuštění programu.

Javascript má k dispozici množství webových API[3], které slouží k různým účelům a díky nim je možné manipulovat s webovými dokumenty, komunikovat přes síť, zpracovávat multimediální obsah, sledovat polohu uživatele, nebo i pracovat s grafikou přímo v prohlížeči. Tyto API poskytují silnou platformu pro tvorbu interaktivních a dynamických webových aplikací, které mohou být přizpůsobeny pro různé zařízení a potřeby uživatelů. [13]

1.7.1 Typescript

Typescript je staticky typovaý programovací jazyk. Jedná se o nadstavbu jazyku Javascriptu. Typescript díky statickému typovaní vyřeší problémy, které jsme avizoval v předchozí sekci.

Při kompilaci proběhne kontrola typů, pokud je chyba v kódu a některé typy spolu nejsou kompatibilní, tak je vývojář na tyto chyby upozorněn. Pokud kompilace proběhne bez chyb, tak se vygenerujou Javascriptové soubory, které lze v prohlížeči pustit. [14]

1.7.2 Výkon

Javascript je dostatečně rychlý pro manipulaci s webovou stránkou, ale nepoužívá se pro náročné výpočty. Podle článku[15] je Javascript jeden z nejpomalejších. Proto je na náročné výpočty používat jiný jazyk.

Pokud je přesto chtěné v prostředí webu dělat náročné výpočty (jako je neuroevoluce), tak se musí použít moderní technologie, které web nabízí. Tyto technologie si víc popíšeme v sekci 1.9 a 1.10.

1.8 C++

C++ je jazyk vyvinutý v roce 1985 a je kombinací nízkoúrovňového a vysokoúrovňového jazyka. Je to nadstavba nad jazykem C a je objektový. Podle článku[15] je C++ jeden z nejrychlejších programovacích jazyků.

Jazyk C++ je zároveň hodně univerzální. Díky tomu, že to je nížkoúrovňový jazyk, tak program vytvořený v C++ může být snadno přenesen na různé platformy a operační systémy. To je umožněno standardizací jazyka, kterou udržuje International Organization for Standardization[4] (ISO). Standardní C++ kód, který nevyužívá specifické vlastnosti systému nebo nestandardní rozšíření, lze kompilovat a spouštět na jakékoliv platformě s kompatibilním kompilátorem. Toto přenositelnost z něj činí ideální volbu pro vývoj křížově platformního softwaru, od desktopových aplikací až po složité operační systémy a vestavěné systémy.

Pro jednodušší programování v C++ se používá standardní knihovna[5]. Ta má v sobě zabudované základní datové struktury (například dynamické pole) a další pomocné funkcionality. [16]

[1] https://w3techs.com/technologies/details/cp-javascript
[2] https://v8.dev/
[3] https://developer.mozilla.org/en-US/docs/Web/API
[4] https://www.iso.org/home.html
[5] https://en.cppreference.com/w/cpp/standard_library

1.9 WebAssembly/WASM

WebAssembly (zkráceně WASM) je kód, který je spustitelný v webovém prohlížeči. Jedná se o jazyk podobný assembleru[6]. WASM soubor je binární, takže je kompaktní a zároveň disponuje jednoduchými nízkoúrovňovímy instrukcemy. Díky tomu má WASM téměr stejnou rychlost, jako jiné nízkoúrovňové jazyky.

Zároveň díky nízkoúrovňovému rozhraní může kompilovat jiné nízkoúrovňové jazyky do WASM. Při procesu kompilace se pouze zamění strojové instrukce za WASM instrukce. Vygenerované soubory jsou `.wasm` a takovýto typ souborů už lze spouštět na webu.

Pokud se na webu spouští WASM kód, tak je zamýšlené, aby běžel koordinovaně s Javascriptem. Proto jsou v WASM přímo zabudované mechanismy, jak komunikovat s Javascript kódem a zároveň jak přes Javascript pouštět WASM kód.

WASM má aktuálně dva případy, kdy je dobré ho využít. Pokud je program napsaný v nízkoúrovňovém jazyku a cílem je ho spustit na webu nebo pokud je program výpočetně náročný a potřebujeme výpočet urychlit. Naopak používat WASM jako absolutní náhradu za Javascript může být náročné a proto je dobré dělat kombinaci těchto dvou jazyků. [17]

1.9.1 Emscripten

Emscripten je open-source nástroj pro kompilaci jazyků jako C, C++ a nebo jiné jazyky co podporuje LLVM[7] do WASM. Každý kód, který není závislý na nějaké specifické funkcionalitě konkrétního systému (je tzv. přenositelný) se dá zkompilovat přes Emscripten. Emscripten je plnohodnotný kompilátor a dá se nahradit, místo běžného kompilátoru.

Výstupem je pak `.wasm` soubor a zároveň pomocný Javascript soubor, který umožňuje jednoduché napojení a zároveň řeší nížkoúrovňové problémy. Jako je například alokace paměti v Javascriptu, jelikož v C++ je paměť spravována explicitně, na rozdíl od Javascriptu, kde se o paměť stará garbage collector. Tento Javascript soubor také zajišťuje komunikaci mezi WebAssembly a JavaScriptem, včetně převodu datových typů a volání funkcí mezi oběma jazyky.

Zároveň Emscripten poskytuje C++ hlavičky[8]. Ty umožní vybrat, jaký kód se má zpřístupnit do Javascriptu a definovat, jaké funkce a datové struktury budou vystaveny. Díky tomu se může v C++ kódu specifikovat, které části se mají kompilovat do WebAssembly a jakým způsobem budou dostupné v Javascriptu. Tento proces umožňuje efektivní integraci komplexních C++ aplikací do webového prostředí s minimálními úpravami původního kódu. [18]

1.10 Web Workers

Web Workers je webový standart z roku 2009[9]. Ten umožňuje, aby Javascript kód běžel na pozadí vedle hlavního Javascript vlákna. K tomu využívá vícejádrové procesory a kód běží na samostatném vlákně.

Web Workers mají dvě hlavní využití a to je abstrakce výpočtu do samostnaného vlákna a díky tomu neblokuje hlavní vlákno, které ovládá interaktivní prvky webu. Druhý důvod je paralelizace na náročné výpočty. Díky tomu aplikace může naplno využívat výkon procesoru.

Pro ovládání Web Workers je sada tříd a funkcí z webového API Web Workers [19]. Pro zjištění počtu jader na zařízení slouží `navigator.hardwareConcurrency` z webového API Navigator[10]. Využití informace o maximálním počtu jader na procesoru je důležitá, pokud je potřeba maximalizovat výkon. Bez této informace nezbývá než náhodné zvolit počet Web Workerů. Pokud

[6] https://www.ibm.com/docs/en/zos/2.1.0?topic=introduction-assembler-language
[7] https://llvm.org/
[8] https://learn.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=msvc-170
[9] https://www.w3.org/TR/2009/WD-workers-20091029/
[10] https://developer.mozilla.org/en-US/docs/Web/API/Navigator

počet Web Workerů bude menší než počet jader, tak se nevyužije maximální výkon procesoru. Pokud se zvolí větší počet, tak se výkon oproti maximálnímu počtu jader nezmění, ale režie spojená s přepínáním jader se zvýší.

Při vytvoření Web Workeru je potřeba specifikovat, jaký kód se má spustit. To jde udělat buď tak, že se v řetězci předá funkce do konstruktoru. Takový přístup má nevýhodu, jelikož ve funkci se nesmí používat žádné závislosti na jiný kód a všechny pomocné třídy nebo funkce musí být obsažené v předávané funkci. Druhá možnost je do konstruktoru předat dedikovaný soubor co se následně spustí na novém vlákně. Ten může využívat celý kód v souboru a zároveň lze importovat další kód z jiných souborů.

Jelikož kód běží úplně v jiném vlákně, tak k němu nelze přistupovat přímo z hlavního vlákna. Proto Web Workerse API poskytuje komunikace v podobě tvz. odchytávání zpráv. V hlavním vlákně se pak vytvoří funkce, která se spustí pokud Web Worker vyšle zprávu. Jako argument mohou být i specifická data, co bude Web Worker posílat (například výsledky výpočtu). Pokud potom Web Worker pošle zprávu, tak se přes funkci zpracuje. Tento princip lze využít i naopak, pokud je potřeba posílat data do Web Workeru. [19]

1.11 Herní engine

Herní engine je sada nástrojů, která za vývojáře řeší základní problémy při vývoji her. Není přesně specifikované co všechno musí obsahovat, ale jako příklad může být:

- grafické vykreslování hry
- aktualizace stavu hry v čase
- synchronizace podle výkonu stroje (tak aby hra běžela plynule)
- zpracování vstupu od uživatele
- fyzikální simulace (kolize mezi objekty, gravitace)

Účelem je vývojářům ušetřit složitou a redundantní práci, při vývoji nové hry. Zároveň se nemusí jednat pouze o hry, ale v herním engine se může udělat simulace nějakého prostředí. Nebo v kontextu neuroevoluce se zamění vstup od uživatele za neuronovou síť. [20]

1.12 Pomocné nástroje

V této sekci budu představovat všechny zbylé technologie, které budou na vytvoření knihovny potřeba. Jelikož nejsou stěžejní, tak není potřeba prozkoumávat detaily těchto technologií, ale je potřeba je popsat.

1.12.1 Vite

Vite[11] je nástroj pro vývoj a sestavování Javascript kódu do produkční verze. Skládá se ze dvou části, první je vývojový mód. Vite dělá vývojářům kompilaci a vystavení jejich aplikaci v reálném čase. To usnadňuje vývoj, jelikož po každé změně v Javascript kódu se nemusí provádět kompilaci aplikace. Druhá část je kompilace do produkční verze a zároveň dělá několik optimalizací:

- kompilace Typescript souborů na Javascript soubory (pokud aplikace Typescript obsahuje)
- minifikace proměnných, kvůli zmenšení finálních Javascript souborů

[11] https://vitejs.dev/guide/

- rozdělení Javascript kódu do tzv. kusů pro lepší načítání z webového prohlížeče
- podpora Web Workers a WASM pro jednodušší implementaci
- algoritmus na odstranění nepoužitých částí kódu (tree shaking[12])

Výstup kompilace jsou `.html`, `.js` a `.wasm` soubory. Ty už reprezentují spustitelnou aplikaci přes webový prohlížeč.

1.12.2 Vue

VueJS[13] je Javascript framework na tvorbu moderních uživatelských rozhraní. Soustředí se hlavně na interaktivní rozhraní a je postaveno na jazyku Javascript, HTML a CSS.

Pro tvorbu webových aplikací nabízi řadu funkcionalit. Reaktivní stavy slouží k automatickému propisování do HTML kódu stránky. Abstrakce dílčích webových částí do tzv. komponent, které se následně dají přepoužívat.

Zdrojový kód aplikace postavené v VueJS je uložený v `.vue` souborech, které se následně pomocí rozšíření do nástroje Vite zkompilují do Javascript souborů.

1.12.3 Docker

Docker[14] je open-source platforma, které umožňuje nasazování aplikací pomocí kontejnerů. Kontejnery umožní vývojáři zabalit svojí aplikaci do samostatného prostředí, které má k dispozici potřebné nástroje pro běh aplikace.

Díky tomu není potřeba řešit architekturu a dostupné nástroje na systému, kde aplikace běží. Stačí pouze nainstalovaný docker na systému, kde spouštíme aplikaci. Všechno ostatní už je obsaženo v připraveném kontajneru.

Do docker kontajneru se mohou připravit postupné kroky pro spuštění aplikace. To znamená, že lze například připravit i kompilační proces a následné vystavení aplikace při spuštění docker kontejneru.

[12] https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking
[13] https://vuejs.org/guide/introduction
[14] https://docs.docker.com/get-started/overview/

Kapitola 2
Návrh řešení a komparativní analýza

2.1 Motivace

Než popíšu návrh řešení, tak je dobré zmínit proč je knihovna relevantní. Na první pohled se může zdát, že v rámci bakalářské práce se bude implementovat zastaralý algoritmus, co už přežil svoji dobu. Jazyk Javascript je pomalý oproti ostatním jazykům, které se používají pro strojové učení. A zároveň v prostředí, které není optimalizované pro strojové učení. Přesto si myslím, že to smysl má a v této kapitole budu rozebírat různé důvody, proč se tomuto tématu vyplatí věnovat.

První věc, která by někoho mohla překvapit, je fakt, že tato bakalářská práce implementuje knihovnu pro Javascript v kombinaci s C++. Python by byla o dost lepší volba, jelikož se jedná o populární jazyk jak z řad výzkumníků, tak i vývojářů. Zároveň v pythonu už existují zajímavé a sofistikované knihovny, zaměřené na tuto problematiku. Díky tomu že může dělat složité výpočty v C++ podprogramu, tak je dokáže být výkonný. Javascript je přesto relevantní. Podle dotazníku ze StackOverflow [21] z roku 2023 je Javascript nejpopulárnější jazyk mezi vývojáři. Proto by po takovém vývojovém prostředí mohla být poptávka.

Pokud bych se zaměřili pouze na skupinu výzkumníků/vývojářů, co se věnují vývoji umělé inteligence, tak bych dostal jiné výsledky (nejpopulárnější by byl nejspíš python). To ale znamená, že pokud někoho zaujme neuroevoluce, tak nemá jinou možnost, než se naučit python. To stejné platí pokud vývojář umí jiný jazyk co vychází z jazyka C (tzv. „C like languages“). Přechod z takového jazyka do Javascriptu by mohl být jednodušší, jelikož Javascript vychází z jazyka C.

V úvodu jsem zmiňoval, že existuje malé množství knihoven, co se věnují neuroevoluci v prostředí webu. Existujících řešení většinou implementují pouze jeden algoritmus na míru a žádná snaha o rozšířitelnost či efektivitu není. Najdou se však sofistikované knihovny, které jsou výkonnostně optimalizované. Jedná se však o knihovny, co jsou zaměřené primárně na učení s učitelem. Zároveň se tyto knihovny nesnaží optimalizovat specifické procesy, které neuroevoluce obnáší, jako například vyhodnocení populace. O různých a již existujících řešení se budu věnovat v sekci 2.3.

Způsob, jak je Javascript koncipovaný a implementovaný přináší několik problémů, jedním z nich je výkon. V porovnání s ostatnímy jazyky vyšel jako jeden z nejhorších [15]. Tím že Javascript je omezený na jedno jádro, tak ztrácíme možnost paralelizace. Dalším problémem je přenositelnost, jelikož Javascript beží pouze v runtime prostředí jako je webový prohlížeč nebo samostatných runtime prostředí jako je NodeJS[1].

[1]https://nodejs.org/en/about

2.2 Návrh řešení

Shrnul jsme proč by mělo smysl dělat knihovnu, kterou v rámci bakalářské práce mám implementovat. Taky jsem avizovali některé problémy, které s takovým přístupem přichází. V této sekci využijeme znalosti z teoretické části. Díky nim uděláme navrh knihovny, kde využijeme představené technologie. Návrh bude efektivně řešit problematiku neuroevoluce v prostředí webového prohlížeče.

2.2.1 Využité technologie

Hlavní problém, který byl nastíněn již u evolučních algoritmů v sekci 1.4 je evaluace (výpočet fitness funkce) pro jednotlivé genotypy. Tento problém se ještě více implikuje, pokud bude velká populace. V režimu posilovaného učení to znamená udělat víc simulací běhu prostředí. Ta může být velmi náročné a znamená to spustit celou simulaci a algoritmus dopředného šíření neuronové sítě.

Využiji faktu, že nezáleží, v jakém pořadí proběhne evaluace jednotlivých genotypů. Díky tomu je zde prostor pro paralelizaci. To umožňuje technologie Web Workers, která dovoluje spouštět procesy ve vlastním vlákně a může se zpracovávat více simulací prostředí naráz. Limit je v tomto ohledu pouze počet jader na daném stroji, kde neuroevoluci spouštíme.

Další věc, kterou je potřeba zvážit je univerzálnost a rychlost samotné implementace algoritmu. Zde využiji WebAssembly, které umožňuje kompilovat a spouštět C++ kód ve webovém prohlížeči. Díky tomu může být v C++ samotná implementace neuronových sítí a algoritmu pro neuroevoluci. To dává rychlost a zároveň přepoužitelnost, jelikož C++ kód je na rozdíl od Javascriptu univerzálnější.

Celé řešení bude navíc zabaleno v moderních technologiích jako je Typescript pro typovost v Javascriptu:

- Vue - implementace demostránky
- Vite - kompilace a optimalizace výsledných Typescript/Javascript souborů
- Docker - jednoduché spuštění projektu na vlastním zařízení

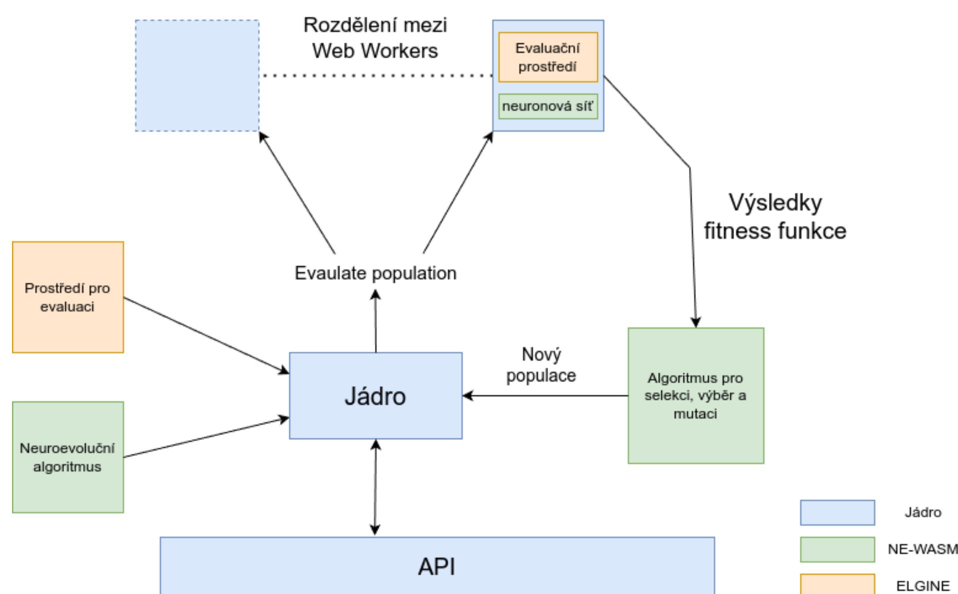
2.2.2 Rozhraní

Knihovna bude rozdělené na tři částí. Rozhraní pro implementaci neuroevoluce (dále ho budeme referovat jako NE-WASM), rozhraní pro implementaci prostředí (to dále budeme referovat jako ELGINE) a jádro, které bude tyto dvě části spojovat a bude poskytovat API pro uživatele.

NE-WASM část bude dávat vývojářům rozhraní, díky kterému stačí nadefinovat základní prvky neuroevoluce, jako je generování nové generace, nebo převod genotypu na fenotyp. K tomu budou mít vývojáři připravené API, kde si pomocí funkcí sestaví vlastní síť a o vše ostatní už se postará knihovna.

ELGINE bude jednoduchý herní engine. Ten dá vývojářům možnost implementovat vlastní prostředí, bez hlubší znalosti webových prohlížečů. Zároveň ELGINE obsahuje speciální režim, kdy je prostředí ovládané neuronovou sítí. V tomto prostředí se nic nevykresluje a běh není omezen na určitý počet snímků za sekundu. To urychlí běh při procesu učení, kdy nejde o vizuální stránku běhu. Využití ELGINE není nutné. Pokud má vývojář vlastní prostředí, tak si ho může přes API napojit na jádro stejně, jako by to udělal s ELGINE. Běh však nemusí být adekvátně optimalizován a proces učení nemusí být stejně efektivní.

Jádro aplikace bude hlavní část knihovny, která se bude starat o spojení dvou předešlých částí. Zároveň bude mít API, díky kterému budeme moct ovládat běh neuroevoluce a další pomocné nástroje pro export/import algoritmu nebo získávání statistických dat. Jádro si bude



Obrázek 2.1 Diagram návrhu knihovny
V návrhu je vidět proces, jak spolu jednotlivé části komunikují. Zároveň je barevně znázorněno, jestli se jedná o jádro, NE-WASM nebo ELGINE část.

samo spouštět instance prostředí, provádět evaluaci a distribuci výpočtu na víc vláken. Zároveň bude provádět všechny procesy v neuroevoluci pomocí přímého propojení s algoritmem přes WebAssembly.

Pro ukázku a otestování bude mít knihovna připravenou demo stránku, kde budou implementovány veškeré funkcionality, které knihovna nabízí. A to včetně dvou algoritmů a dvou prostředí, na kterých jde knihovna spustit. To může sloužit jako inspirace, jak implementovat a napojit vlastní neuroevoluní algoritmus.

2.2.3 Proces učení

Tato podsekce by měla ještě víc přiblížit, jak bude knihovna používat jednotlivé části při samotném běhu neuroevolučního procesu.

Jádro dostane jako vstupní parametry prostředí a algoritmus. Následně se pak inicializuje prvotní populace. Jednotlivci z populace se rozdělí do web workerů, a poté se udělá evaluace každého z nich.

Evaluace bude probíhat tak, že se nejdřív sestaví neuronovou síť z genotypu. Následně se spustí prostředí, které bude mít k dispozici neuronovou síť. Kód si před spuštěním simulace zaintegruje síť jako ovládání. Poté proběhne samotná simulace ovládá přes síť.

Výsledky s jedinci se následně vrátí a použijí se v NE-WASM algoritmu, pro generaci další generace. Tento proces se opakuje, dokud nejsou splněny podmínky pro ukončení (například dostatečná hodnota z fitness funkce).

Všechno je znázorněno v obrázku 2.1.

2.3 Existující řešení

Neuroevoluce není nová metoda, tudíž existuje mnoho knihoven a řešení, které již někdo implementoval. V předchozí sekci jsem přiblížil, jak by měla knihovna vypadat a jaké všechny

výhody by díky takovému návrhu měla získat. Tato sekce slouží pro srovnání již existujících řešení a komparace s naším návrhem.

2.3.1 Brain.js

Brain.js[2] je velmi populární knihovnu pro implementaci klasických úloh strojového učení, jako je regrese nebo klasifikace. Disponuje velmi jednoduchým API, takže práce s touto knihovnou je velmi příjemná. Zároveň je populární kvůli interní implementaci knihovny gpu.js, která umí efektivně využít grafickou kartu k urychlení výpočtu.

Nevýhoda spočívá v jeho zaměření, jelikož je dělaný na učení přímo na datech. Proto nemá žádnou podporu pro efektivní běh úlohy v režimu posilovaného učení. Pro využití knihovny brain.js k neuroevoluci, by bylo potřeba ještě implementovat dílčí části navíc. A zároveň bychom pořád měli problém s náročností simulace prostředí, které zabere nejvíc času.

2.3.2 Neataptic.js

Knihovna Neataptic.js[3] je založena na knihovně Synaptic[4]. Obě tyto knihovny jsou v době psání této bakalářské práce 5 let nespravované. Jedná se o knihovnu co se specializuje na neuroevoluci, ale implementovaná je pouze metoda NEAT a knihovna je architekturou připravená pouze na genetické algoritmy. Pro implementaci nového algoritmu je potřeba dobře znát strukturu knihovny a není zde jednoduché API pro implementaci nového algoritmu.

Zároveň knihovna není nijak optimalizována na výkon. Obsahuje sice modul pro využití více vláken přes Web Workers, ale není dostupná pro neuroevoluci. Pouze pro úlohy na učení s učitelem.

2.3.3 TensorFlow.js

TensorFlow.js[5] (TFJS) je ze všech již zmíněných knihoven/nástrojů nejzajímavější. Jedná se o odnož velmi populární python knihovny *TensorFlow*. Tato knihovna je nejlepší volba pro implementaci běžných úloh ve strojovém učení v prostředí webového prohlížeče. TFJS má high-level API pro využití konkrétních algoritmů, až po low-level API pro práci s neuronovou sítí. Jeho bezkonkurenční výhoda je kromě výše uvedeného také výkon. TFJS disponuje několika podpůrnými knihovnami (tzv. backend) pro výpočet.

Základní je CPU backend, kde se pro výpočet využívá procesor, což není tak zajímavé. Zajímavější možnost je WebGL backend, která stejně jako brain.js pro lepší výkon využívá hardwarovou akceleraci přes GPU. Poslední možnost je WASM backend. Jedná se o kombinaci WebAssembly, distribuce mezi vlákny a google knihovny XNNPACK [22] (optimalizovaný C/C++ kód pro WASM). Díky kombinaci těchto technologií je TFJS ideální volba tam, kde je potřeba výkon (například rozpoznávání obličejů na stránce). Tento přístup se velmi podobá mému návrhu, ale postrádá podporu neuroevoluce. Tudíž by bylo učení omezeno rychlostí simulace prostředí. Pro vlastní implementaci některého z neuroevolučních algoritmů, by bylo možné využít jen low-level API, pro sestavování a běh neuronových sítí. Tato knihovna se však ze všech možností nejvíc přiblížil k tomu, co bychom chtěli.

[2] https://github.com/BrainJS/brain.js#brainjs
[3] https://github.com/wagenaartje/neataptic
[4] https://github.com/cazala/synaptic
[5] https://github.com/tensorflow/tfjs/tree/master#tensorflowjs

Kapitola 3
Implementace knihovny

V této části budu dělat implementaci samotné knihovny. Postupně budu implementovat dílčí části (ELGINE, NE-WASM a jádro). Veškerý kód je k dispozici v github repozitáři[1], který je vedený jako open-source projekt.

3.1 ELGINE

První část se bude věnovat tvorbě webového herního enginu (tato část se v projektu referuje jako ELGINE). Ten slouží pro simulaci prostředí. Vývojáři tak budou moct jednoduše implementovat vlastní prostředí, na kterém chtějí aplikovat neuroevoluci.

Kromě toho je prostředí optimalizované pro běh neuroevoluce. ELGINE je implementovaný v jazyce typescript a je rozdělen do několika abstraktních tříd. Engine se vykresluje na `HTMLCanvasElement`[2].

3.1.1 Sdílený stav

Napříč všemi objekty se předává generický typ `TSharedState`, který reprezentuje sdílený objekt, který je přístupný z různých částí kódu. Díky tomu si může vývojář definovat vlastní strukturu sdíleného stavu. Ten ELGINE propaguje do různých funkcí a vývojář ho může upravovat, nebo podle něj implementovat logiku.

Tento objekt nemá žádnou předem danou strukturu a je čistě na vývojáři, jaké složení objektu si vytvoří. Sdílený stav slouží na ukládání globálních dat o simulaci. Například skóre, rozměry prostředí, ovládání agenta nebo vjemy z prostředí.

3.1.2 Controls.ts

Třída `CControl` slouží k implementaci ovládání agenta, tento abstraktní objekt se skládá ze dvou veřejných metod, jak je vidět v kódu 3.1.

Metoda `onmount` se spustí pouze jednou, při inicializaci prostředí. Tato metoda slouží například k registraci vstupu z klávesnice, nebo pro inicializaci ovládání přes neuronovou síť.

Metoda `updateState` je hlavní část této třídy. Parametr této metody je sdílený stav. Metoda se spouští v každé iteraci aktualizace prostředí a díky ní se může aktualizovat sdílený stav. V sdíleném stavu je uložená informace o tom, jestli agent provede nějakou akci. Metoda následně

[1] https://github.com/prosteNoBody/asm-ne-gym
[2] https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement

Výpis kódu 3.1 Controls.ts

```
abstract class CControl<TSharedState> {
    public abstract updateState(state: TSharedState): TSharedState;
    public onmount() {};
}
```

může upravit tento stav, podle interní logiky. To může udělat například tím, že vjemy ze sdíleného stavu vloží do neuronové sítě, následně udělá algoritmus dopředného šíření a podle výstupu z neuronové sítě upraví sdílený stav.

Díky tomu, že se jedná o samostatnou třídu, tak se může vytvořit několik tříd podle toho, jakou strategii ovládání agenta chceme zvolit. Takže ve výsledku pro jedno prostředí může být ovládání klávesnicí a druhé, které se může ovládat přes neuronovou síť.

3.1.3 Entity.ts

Další část ELGINE je třída `CEntity`, která reprezentuje entitu v simulaci. Z kódu této třídy 3.2 popíšu pouze důležité prvky.

Základní informace o entitě, které si chceme zaznamenat jsou:

- id - identifikátor, který slouží pro referenci na objekt
- position - pozice entity v 2D prostoru (skládá se z x a y pozice)
- size - reprezentuje velikost objektu v 2D prostoru

Informace o pozici a velikosti jsou pak důležité při vykreslování a je potřeba, aby je obsahovala každá entita. Třída při deklaraci přijímá generický typ sdíleného stavu, díky tomu se může synchronizovat sdílený stav napříč třídami, i když nejsou třídy přímo propojené.

Metoda `update` slouží k aktualizaci interního stavu objektu. V parametrech metody se předává již zmíněný sdílený stav. Druhý parameter je tzv. tik, který určije kolikátá aktualizace proběhla. To jde využít pokud je potřeba dělat nějakou akci jenom jednou za určitou dobu. Metoda `render` umožňuje vykreslování entity. Jako parametr se předává nativní webový objekt `CanvasRenderingContext2D`, který má API[3] pro vykreslování na `HTMLCanvasElement`. Poslední důležitá metoda je `collide`, která se spustí pokud dojde ke kolizi s jinou entitou. Jako parametr je kromě sdíleného stavu také entita, se kterou kolize proběhla. Pomocí toho se může rozlišit, jaká akce se má provést.

3.1.4 Elgine.ts

Poslední a hlavní třídá je CElgine 3.3. Poděděná třídá z této abstraktní už reprezentuje hotové prostředí. Třídá přijímá dva generické typy. Sdílený stav a typ entity, která má jako generický typ stejný sdílený stav jako má entita. Díky tomu je zajištěno, že všechny části mají stejný sdílený stav.

Základní data, které si třída elgine uchovává jsou:

- ctx - objekt `CanvasRenderingContext2D`, který umožňuje vykreslování
- entities - pole, ve kterém jsou uložené všechny registrované entity
- shared state - sdílený stav, tento objekt se následně distribuuje do metod, které ho využívají

[3] https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D

Výpis kódu 3.2 Entity.ts

```
abstract class CEntity<TSharedState> {
    protected readonly _id: UUID;
    protected readonly _pos: TPosition;
    protected readonly _size: TSize;

    constructor(size: TSize, position: TPosition, collisions: boolean);

    public abstract update(state: TSharedState, ticks: number): void;
    public abstract render(ctx: CanvasRenderingContext2D): void;
    public abstract collide(
        state: TSharedState, entity: CEntity<TSharedState>
    ): void;
}
```

- control - instance objektu `CControl` pro zvolení ovládací strategie agenta

Tato třída se stará hlavně o vykreslování a správného provolání metod na objektech `CEntity` nebo `CControl`. Jedna z takových metod je `update`, která nejdříve zavolá `updateState` z třídy `CControl` a pak proiteruje zaregistrované entity a na nich spustí metodu `CEntity.update`. Nakonec proběhne iterace mezi všemi entitami mezi sebou, jestli podle jejich pozice a velikosti nedošlo ke kolizi. Pokud ano, tak se na entitách, které kolidují mezi sebou, spustí metoda `CEntity.colide`. Pro vykreslování na `HTMLCanvasElement` slouží metoda `render`, která postupně proiteruje všechny entity a zavolá metody `CEntity.render`. Pro zaregistrování entity je připravená metoda `registerEntity`, která přidá entitu do interního pole entit.

Po inicializaci třídy ještě není vytvořená vazba na cílový `HTMLCanvasElement`, kde probíhá vykreslování. K tomu slouží metoda `mount`. Jako parametr je `HTMLCanvasElement`. Tato metoda je důležitá při běhu prostředí.

Pro běh prostředí je připravená metoda `run`. Tato metoda se nejdřív podívá, jestli je v třídě uložený `HTMLCanvasElement`. Pokud je, tak proběhne běžný průběh s vykreslováním. To dělá metoda `renderableLoop`, která kromě aktualizace prostředí taky vykresluje. Zároveň si interně řeší obnovovací frekvenci (udržovat určitý počet iterací za sekundu), aby prostředí probíhala plynule. To zařizuje funkce `requestAnimationFrame`[4].

Pokud třídá nemá vazbu na `HTMLCanvasElement` tak proběhne běh bez vykreslování. To zpracovává metoda `loop`. Ta pouze provolává aktualizaci prostředí, dokud simulace neskončí. Jelikož není potřeba řešit obnovovácí frekvenci, tak simulace proběhne výrazně rychleji. To je důležité pokud je potřeba rychlá evaluace prostředí s výsledkem simulace a nejde o vizuální stránku. Toto je využito při trénování neuronové sítě, kde se na pozadí spouští několik simulací naráz, kde jde o rychlost.

[4]https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

Výpis kódu 3.3 Elgine.ts

```
abstract class CElgine<
    Entity extends CEntity<TSharedState>, TSharedState
> {
    protected _ctx: undefined | CanvasRenderingContext2D;
    protected _entities: Array<Entity> = [];
    protected _sharedState: TSharedState;
    protected _control: CControl<TSharedState>;

    constructor (control: CControl<TSharedState>);

    public mount(ctx: CanvasRenderingContext2D): void {
        this._ctx = ctx;
    }

    protected registerEntity(entity: Entity): void;

    private render(): void;
    private update(): void {
        this._control.updateState(this._sharedState);
        this.updateEntities();
        this.updateColissions();
    }
    private loop(): void {
        while (this._isActive)
            this.update();
    }
    private renderableLoop(timestamp: DOMHighResTimeStamp): void {
        let delta = timestamp - this._lastFrameTimeMs;
        let wasUpdatedFlag = false;
        while (delta >= TIMESTEP) {
            wasUpdatedFlag = true;
            // one game process
            this.update();
            this.render();
            this._lastFrameTimeMs += TIMESTEP;
            delta -= TIMESTEP;
        }
        requestAnimationFrame(t => this.renderableLoop(t));
    }
    public run(): void {
        this._isActive = true;

        if (this._ctx)
            renderableLoop();
        else
            this.loop();
    }
}
```

Výpis kódu 3.4 ENeuronType.hpp

```
enum class ENeuronType {
    INPUT,
    HIDDEN,
    OUTPUT,
};
```

Výpis kódu 3.5 Utils.hpp

```
class ActivationFunctions {
public:
    static double SIGM (double x);
    static double TANH (double x);
    static double RELU (double x);
    static double LRELU (double x);
    static double GELU (double x);
    static double SIN (double x);
    static double COS (double x);
    static double ABS (double x);
    static double GAUS (double x);
    static double IDENTITY (double x);
    static double QUAD (double x);
};
```

3.2 NE-WASM

Druhá část knihovny se stará o implementací rozhraní, které vývojáři zjednoduší implementaci neuroevolučního algoritmu. Kód je implementován v jazyku C++. Díky tomu je možné přepoužít algoritmus v jiné aplikaci, jelikož C++ je univerzální jazyk. Rozhraní je v podobě abstraktní třídy. V rámci toho je připravená i třída pro export veškerého kódu do WASM.

3.2.1 Neuronová síť

Základním prvkem v této části je neuronová síť. Důležitý důraz je kladen hlavně na univerzálnost knihovny. Proto je potřeba mít možnost postavit jakoukoliv neuronovou síť. Naivní způsob by byl, kdyby se síť stavěla po skrytých vrstvách. Tento přístup však není vhodný pro algoritmy s jinou architekturou, který strukturu tvoří jinak a není přesně rozdělená na vrstvy. U konvenční neuroevoluce by tento přístup stačil. U algoritmu NEAT, který svojí architekturu staví postupně a náhodně, by už tento přístup nevyhovoval. Proto se zaměřím pouze na aspekty, které každá neuronová síť musí implementovat a pouze ty připravit vývojáři a dát mu tak maximální možnosti.

Základním prvkem sítě je neuron. O něm vždy potřebujeme vědět informaci, jestli je vstupní, výstupní nebo skrytý. Proto je připravený enum 3.4, který tuto informaci reprezentuje. Zároveň neuron vždy obsahuje aktivační funkci, tudíž je potřeba mít možnost si jí zvolit. Pro neuronové sítě se používá především aktivační funkce sigmoid, ale některé algoritmy [10] používají i jiné aktivační funkce. Pro tento účel je vyvořená pomocná třída 3.5, která dává širokou škálu aktivačních funkcí, které lze využít a může se do budoucna rozšířit o další.

Dalším obecným prvkem je synapse, ta je důležitá k propojení neuronů mezi sebou. Ta obsa-

Výpis kódu 3.6 CNetwork.hpp

```
class CNetwork {
private:
    std::vector<CNeuron> m_neurons;
    std::vector<size_t> m_in_neurons;
    std::vector<size_t> m_out_neurons;
public:
    void registerNode(std::function<double(double)>, ENeuronType);
    void registerConnection(size_t, size_t, double);
    std::vector<double> forward(const std::vector<double>&);
};
```

huje informaci o tom, z jakého neuronu vychází a do jakého vstupuje. Zároveň je potřeba držet informaci o tom, jakou má váhu.

Z těchto prvků už můžu vytvořit třídu `CNetwork` 3.6. Ta reprezentuje připravené rozhraní pro vývojáře, ze kterého lze postavit jakákoliv neuronová síť.

Pro tvorbu sítě se používají dvě metody. `registerNode`, která jako parametry přijímá aktivační funkci a zároveň typ neuronu. Tento neuron ještě není propojen s ostatními a tudíž je ještě potřeba využít metodu `registerConnection`. Ta přijímá tři parametry. První dva jsou indexy vstupního a výstupního neuronu. Tento index závisí na tom, jak byly neurony zaregistorvané. Poslední parametr je kontinuální hodnota typu `double`, která reprezentuje váhu, kterou bude výstup z neuronu přenásobený, než se pošle do výstupního neuronu.

Poslední metoda třídy `forward` slouží k spuštění algoritmu dopředné šíření. Ta se může opakovaně spouštět na již sestavené síti. Jako vstupní parametr přijímá pole kontinuálních hodnot, které se následně předají do vstupních neuronů. Následně proběhne algoritmus dopředného šíření a z výstupních neuronů se hodnoty převedou na pole kontinuálních hodnot. Velikost tohoto pole závisí na počtu výstupních neuronů.

Zaregistrované neurony se zaznamenávají v soukromé proměnné `m_neurons`. V proměnné `m_in_neurons` a `m_out_neurons` se udržují indexy z pole `m_neurons` a určují, které neurony jsou vstupní a výstupní. Díky tomu nemusíme registrovat neurony v pořadí podle toho, jestli je vstupní, skrytý nebo výstupní.

K tomu aby se mohl provést algoritmus dopředného šíření jsou implementovány ještě dvě pomocné třídy `CNeuron` a `CConnection`. `CNeuron` interně obsahuje třídu `CConnection`. Tyto dvě třídy se starají o výpočet, který se postupně propojuje všechny neurony. Nejsou však důležité pro vývojáře, které implementuje vlastní neuroevoluční algoritmus, ale jejich implementaci lze nalézt v repozitáři[5].

Díky tomu máme třídu, ze které jde udělat jakoukoliv síť s jakoukoliv architekturou. To přidává na univerzálnosti a nejsou zde žádné limitace pro to, jaký algoritmus lze pomocí toho implementovat.

3.2.2 Šablona pro algoritmus neuroevoluce

Hlavní třída v NE-WASM části knihovny je `CNeuroevolutionBase` 3.7. Tato třída už jde použít, pro implementaci vlastního neuroevolučního algoritmu. Jedná se o abstraktní třídu, která má předdefinované tři metody, které musí vývojář doplnit, aby byl algoritmus funkční ve zbytku knihovny.

Opět se zaměřuji hlavně na univerzálnost. Proto můj návrh požaduje od vývojáře jen základní operace neuroevolučních algoritmů, které následně budou volány z jádra knihovny. Zakódovaný

[5]https://github.com/prosteNoBody/asm-ne-gym

Výpis kódu 3.7 CNeuroevolutionBase.hpp

```
class CNeuroevolutionBase {
protected:
    CNetwork m_network;
public:
    virtual ~CNeuroevolutionBase() {};

    // name
    virtual std::string getName() const = 0;

    // managing population
    virtual std::string initialPopulation(
        const std::vector<double>&
    ) const = 0;
    virtual std::string generateGeneration(
        const std::vector<double>&,
        const std::vector<double>&,
        const std::string&
    ) const = 0;

    // network managment
    virtual int buildGenome(const std::string& genome) = 0;
    std::vector<double> forward(const std::vector<double>& inputs) {
        return m_network.forward(inputs);
    };
};
```

genotyp nebo celá populace se předává pomocí řetězce. Není to sice kompaktní způsob, ale je dost univerzální na to, aby se s ním dalo dále pracovat. Pokud by vývojář algoritmu přesto chtěl kompaktnější způsob, jak kódovat genotyp, tak může například využít schéma Base64[6]. Ten umožňuje reprezentovat binární data v podobě ascii znaků a tudíž jde aplikovat v návrhu třídy. Konkrétní způsob, jak by mělo kódování vypadat není určeno, proto si vývojář může zvolit vlastní, které pak přepoužije napříč metodami, která v rámci rozhraní musí implementovat. To je z toho důvodu, že některé algoritmy používají přímé kódování a nepřímé kódování nebo si přidávají dodatečné informace k genotypu. Například algoritmus NEAT si přidává historické značky.

Třída je rozdělená na dvě části. Jedna je pro vytváření nebo upravování genotypů a druhá je pro vytvoření sítě a spuštění algoritmu dopředného šíření. Metoda `initialPopulation` slouží pro vytvoření prvotní populace. Jako parametr přijímá pole kontinuálních hodnot, které reprezentují hyperparametry. Díky tomu lze nastavit například velikost prvotní generace. Návratová hodnota této metody je řetězec, který obsahuje všechny genotypy. Ty jsou zakódované v interní struktuře, kterou určuje vývojář/algoritmus. Nastává tu ale problém, jelikož jádro pak potřebuje tento řetězec rozdělit na pole, kde budou jednotlivé genotypy. Tento problém se vyřeší tak, že mezi jednotlivé genotypy se přidá speciální znak, který reprezentuje oddělovač. Kvůli univerzálnosti knihony je ale potřeba dát možnost si nastavit, co má být oddělovací znak. Proto na první pozici v řetězci je tento znak. Díky tomu si jádro načte pouze první znak a podle toho rozdělí řetězec, zde však musí vývojář algoritmu myslet na to, že tento znak nesmí být použit pro reprezentaci genotypu.

[6] https://developer.mozilla.org/en-US/docs/Glossary/Base64

Tento přístup jsem zvolil kvůli tomu, že už existuje mnoho implementací neuroeovlučních algoritmů. Pokud bych předem určil, jaký znak má být oddělovač, tak by to mohlo být nekompatibilní s již existující implementací. Zároveň by byla možnost nevracet řetězec, ale pole řetězců. Díky tomu by se sice vyřešil problém s jasným rozdělením populace na jednotlivé genotypy. Problém je v univerzálnosti. Řetězec má jednoduchou reprezentaci v paměti, za to C++ vektor[7] ne.

Další metoda pro vytváření nové generace je `generateGeneration`. Ta přijímá tři parametrya- Pole hyperparametrů, pole výsledky z fitness funkce a aktuální generaci. Pole výsledků z fitness funkce je seřazené od nejlepšího po nejhorší a stejně tak je seřazená i aktuální generace, která je reprezentována jedním řetězcem (stejně jako u `initialPopulation`).

Metoda slouží pro tvorbu nové generace. Tu vrací v návratové hodnotě jako řetězec ve stejné reprezentaci, jako u `initialPopulation`. Z popisu neuroevolučních algoritmů je k tvorbě nové generace důležitá hodnota z fitness funkce. Jelikož jí zde máme k dispozici, tak implementovaný algoritmus může provést selekci rodičů. Následně podle interní reprezentace provede operace mutace a křížení (popřípadě i jiné operace). Tady návrh knihovny opět neříká, jakým způsobem by se tento proces měl provádět, pouze k němu dá všechny možné informace. Pro selekci má metoda k dispozici hodnoty z fitness funkce za každého jedince. Na operaci mutace se přes hyperparametry může využít hodnota rychlosti mutace (mutation rate). Veškeré interní operace a generování nového řetězce reprezentující novou generaci si musí vývojář implementovat sám.

Jak už jsem zmínil, tak obě metody přijímají pole hyperparametrů. To ale nemá předem danou strukturu, proto si zde vývojář může vytvořit vlastní interní strukturu. Tyto hodnoty se budou nastavovat přes jádro knihovny a budou se dále propagovat až do této třídy.

Druhá část rozhraní se věnuje sestavení sítě a spouštění algoritmu dopředného šíření. Metoda `buildGenome` slouží k sestavení neuronové sítě. K tomu využije objekt `CNetwork`, který byl implementován v předchozí části. Jako parametr metody je jeden genotyp z populace. Podle implementace a za pomocí již zmíněných metod `m_network.registerNode` a `m_network.registerConnection` se sestaví síť. Druhá metoda v této části je `forward`, která pouze volá již implementovanou metodu `m_network.forward` na třídě `CNetwork`. Jako parametr přijímá hodnoty do vstupních neuronů a jako výstup jsou hodnoty z výstupních neuronů.

Dalo by se namítnout, že kvůli rozhraní se budou muset několikrát v každé metodě implementovat stejné procesy. Díky tomu, že je celé rozhraní v jedné třídě, tak se počítá s tím, že si ho vývojář rozšíří o pomocné struktury a třídy podle potřeb konkrétního algoritmu. To je myšleno tak, že například funkce na převod z řetězce do nějaké interní struktury genotypu, se naprogramuje pouze jednou. Následně se taková pomocná funkce použije v metodě `generateGeneration` a zároveň `buildGenome` nebo i `initialPopulation`.

Na začátku této podsekce jsem zmiňoval, že třída obsahuje tři metody. V třídě je ale ještě jedna metoda `getName`, která vrací unikátní identifikátor algoritmu v podobě řetězce. Jedná se o pomocnou metodu, která umožní uživateli výběr algoritmus. Využijeme jí hned v následující sekci.

Ukázku implementace z této třídy si ukážeme v sekci 4.

3.2.3 Třída pro export do WebAssembly

Poslední část NE-WASM třída `AsmCore` 3.8, která spojuje WebAssembly a implementované neuroevoluční algoritmy z třídy `CNeuroevolutionBase`.

Kód, který generuje Emscripten při kompilaci C++ kódu do jeho WASM podoby se skládá ze zdrojového kódu v souboru `.wasm`. Zároveň přidává pomocný `.js` soubor, který slouží k ovládání WASM souboru.

Pro exportování funkce, třídy nebo enumu použiju knihovnu embind[8]. Pak může pomocí jednoduché syntaxe říct, co se má exportovat do Javascript kódu a jak se exportované funkce/třídy

[7] https://en.cppreference.com/w/cpp/container/vector
[8] https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html

mají referencovat v Javascript kódu.

Knihovna si klade za cíl jednoduchou integraci svého vlastního neuroevolučního algoritmu, ale zároveň je potřeba zdrojový kód C++ kompilovat. Proto jsem zvolil cestu, kde využijeme polymorfismu třídy `CNeuroevolutionBase`. Je připravená proměnná, který reprezentovat ukazatel na objekt. Tento objekt si vytvoříme pomocí třídy ze standardní knihovny C++[9] `unique_ptr`. To je vidět v konstruktoru třídy. Ten přijímá řetězec, který reprezentuje unikátní identifikátor konkrétního algoritmu. Podle toho následně vytvoří instanci třídy, pokud nastala shoda mezi parametrem konstruktoru a metodou `CNeuroevolutionBase.getName`.

Ostatní metody třídy jsou pouze obalem pro jejich stejnojmenný ekvivalent v `CNeuroevolutionBase`. Třída tedy pokrývá všechny implementované algoritmy v knihovně. Třída i deklarace její metod se pak využije v funkci pro exporotvání WASM kódu. To je vidět pod deklarací třídy, kde předáme název třídy. Následně určíme parametry konstruktoru a pak zaregistrujeme jednotlivé metody pro export. Jelikož se Emscripten snaží o vytvoření co nejmenšího kódu, tak exportuje pouze to, co je určené pro export. Proto je ještě potřeba deklarovat export C++ datové struktury s generickým typem double (`std::vector<double>`). Typ C++ řetězce se exportuje automaticky do Javascript řetězce.

Jak už jsem zmiňoval, tak tento přístup umožní nejjednodušší způsob, jak integrovat nový neuroevoluční algoritmus do WASM. V ukázce kódu 3.8 stačí přidat `#import` deklaraci s příslušným názvem souboru, kde je algoritmus. Potom je potřeba rozšířit konstruktor objektu o další podmínkovou větev a vytvořit instanci daného algoritmu. V ukázce je vidět integrace `CNE` a `NEAT`, které budu popisovat v sekci 4.

Místo toho přístupu by se dal zvolit i jiný přístup a to export přímo v souboru, kde je implementován neuroevoluční algoritmus. Vzniká tam však řada úkonů, které musí programátor dodatečně udělat. První problém je s deklarací exportu pro Emscripten. Ta by se musela dělat na každém souboru, zároveň by se musel přidat do kompilačního příkazu a ještě by se musela udělat integrace do Typescriptu, kterému se budu věnovat v další části. Proto je tento způsob nejjednodušší a dostatečně univerzální.

S rostoucím počtem implementovaných algoritmů, se bude zvětšovat výsledný soubor, proto je potřeba zahrnout pouze ty algoritmy, které jsou zrovna potřeba.

3.2.4 Implementace WebAssembly do Typescriptu

Ještě je potřeba implementovat proces, díky kterému se bude moct C++ kód spouštět z Typescriptu. První krok je kompilace 3.9. V něm použiju nástroj Emscripten.

Nejdříve připravím správně přepínače příkazu `emcc`. `-lembind` je pro Emscripten C++ bind knihovnu. Ta byla využita v `AsmCore.cpp`, kde jsem zadefinoval, jaká třída má být exportována. Další je přepínač `-O2`, který udělá optimalizace C++ kódu. Tento druh přepínačů jsou běžné pro klasické kompilátory, následující jsou specifické pro WASM. `-s EXPORT_ES6=1` slouží pro exportování pomocného Javascript kódu jako ES6 modul[10]. To je moderní přístup, který se bude v jádru knihovny používat. Díky tomu bude konzistentní importování kódu. `-s MODULARIZE=1` slouží pro minifikaci (zmenšení) vygenerovaného Javascript kódu. `-s ENVIRONMENT="web"` je specifikace cílového prostředí. Díky tomu se zredukují výsledné soubory. Emscripten totiž alternativně počítá s během v NodeJS. Poslední přepínač je `--no-entry`. Díky tomu Emscripten nepočítá s hlavní funkcí `main`, jak to je u C++ běžné. Vstupní bod nepotřebujeme, jelikož si exportujeme celou třídu a její metody. Zbytek kódu pro kompilaci je určení vstupních souborů a název výstupního. Výstup bude Javascript soubor `asm_core.js` a `asm_core.wasm`. Tyto soubory reprezentují spustitelný kód v prohlížeči.

Knihovna napsaná v jazyku Typescript, který je typovaný. Výstupní soubor Emscripten je Javascript bez typů. Tudíž bychom ho nemohli použít v Typescriptu. Proto přidáme deklarativní

[9] https://en.cppreference.com/w/cpp/standard_library
[10] https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export

Výpis kódu 3.8 AsmCore.cpp

```
#include <emscripten/bind.h>

#include "algorithms/Neat.hpp"
#include "algorithms/CNE.hpp"

class AsmCore {
private:
    std::unique_ptr<CNeuroevolutionBase> algorithm;
public:
    // Algorithm register
    AsmCore(const std::string& algorithmType) {
        if (Neat().getName() == algorithmType)
            algorithm = std::make_unique<Neat>();
        else if (CNE().getName() == algorithmType)
            algorithm = std::make_unique<CNE>();
    };

    // wrapper functions
    std::string initialPopulation(
        const std::vector<double>& hyperparameters
    ) {
        return algorithm->initialPopulation(hyperparameters);
    }
    std::string generateGeneration(
        const std::vector<double>& hyperparameters,
        const std::vector<double>& fitness,
        const std::string& population
    ) {
        return algorithm->generateGeneration(
            hyperparameters, fitness, population
        );
    }
    int buildGenome(const std::string& genome) {
        return algorithm->buildGenome(genome);
    }
    std::vector<double> forward(const std::vector<double> inputs) {
        return algorithm->forward(inputs);
    }
};

EMSCRIPTEN_BINDINGS(asm_core) {
    emscripten::class_<AsmCore>("AsmCore")
        .constructor<const std::string&>()
        .function("initialPopulation", &AsmCore::initialPopulation)
        .function("generateGeneration", &AsmCore::generateGeneration)
        .function("buildGenome", &AsmCore::buildGenome)
        .function("forward", &AsmCore::forward)
        ;

    emscripten::register_vector<double>("Vector");
}
```

Výpis kódu 3.9 Makefile

```
WASM_FLAGS := -lembind -O2 -s MODULARIZE=1
    -s EXPORT_ES6=1 -s ENVIRONMENT="web" --no-entry

emcc $(WASM_FLAGS) -o build_wasm/asm_core.js wasm/AsmCore.cpp
    wasm/CNetwork.cpp wasm/CNeuron.cpp wasm/CConnection.cpp
```

Výpis kódu 3.10 asmcore.d.ts

```
declare module "@build_wasm/*.js" {
    export default function (): Promise<{
        AsmCore: typeof import("@core/types/AsmNeGym").AsmCore,
        Vector: typeof import("@core/types/AsmNeGym").Vector,
    }>;
};
```

Typescript soubor[11]. Ten definuje rozhraní souboru, které pak následně při importování souboru bude aplikováno.

Pokud v kódu bude importován Javascript soubor ve složce `@build_wasm/` (zde je potřeba zmínit že používám Vite alias cesty[12]), tak Typescript aplikuje definovaný rozhraní 3.10.

Toto rozhraní obsahuje třídu `AsmCore`, kterou jsem popsal v předchozí kapitole a `Vector`, který reprezentuje dynamicky alokované pole ze standardní knihovny C++. Tudíž rozhraní obou tříd je předem určené, ale jejich typy musíme ručně zadefinovat 3.11 3.12.

Z rozhraní je vidět, že pouze mimikuje již představené třídy. Je tu však jedna metoda navíc, `delete`. Ta je přidána Emscripten nástrojem a slouží k dealokaci paměti. Tudíž se musí manuálně spustit, než přestaneme používat objekt a ztratíme na něj referenci. Je to způsobené tím, že Javascript nemá destruktor. Pokud by se tato metoda nepoužila, tak by sice nedošlo k tzv. úniku paměti, ale rychle by došla dostupná paměť.

[11] https://www.typescriptlang.org/docs/handbook/declaration-files/templates/module-d-ts.html
[12] https://vitejs.dev/config/shared-options#resolve-alias

Výpis kódu 3.11 AsmNeGym.ts:AsmCore

```
export declare class AsmCore implements AsmClass {
    delete(): void;
    constructor(algorithm: string);
    initialPopulation(hyperparameters: Vector): string;
    generateGeneration(
        hyperparameters: Vector,
        fitness: Vector,
        population: string
    ): string;
    buildGenome(genome: string): number;
    forward(inputs: Vector): Vector;
}
```

Výpis kódu 3.12 AsmNeGym.ts:Vector

```
export declare class Vector implements AsmClass {
    delete(): void;
    constructor();
    push_back(value: number): void;
    get(index: number): number;
    size(): number;
}
```

3.3 Jádro

Jádro se hlavní část knihovny, stará se o propojení všech částí (ELGINE a NE-WASM) a zároveň dát vývojáři API na konfiguraci a ovládání knihovny.

Hlavní úkol jádra je nastavení konfigurace od vývojáře a následné spuštění procesu učení. V procesu učení bude mít za úkol získat z NE-WASM novou generaci přes funkci `AsmCore.generateGeneration` (alternativně při počátečním spuštění použije `AsmCore.initialPopulation`).

Následně rozdělí populaci na jednotlivé genotypy a spustí Web Workers procesy (počet těchto procesů závisí na počtu jader na daném stroji). Každý Web Worker dostane sadu genotypů, které bude mít za úkol evaluovat. Kromě genotypu dostane tzv. modul, který reprezentuje prostředí, na kterém se bude genotyp evaluovat.

Z Web Workeru se následně po evaluaci všech genotypů vrátí hodnoty fitness funkcí jednotlivých genotypů. Po dokončení výpočtu ze všech Web Workerů se znovu složí řetězec reprezentující populaci. Na něm se zavolá `AsmCore.generateGeneration` a proces se opakuje.

Tento zjednodušený proces nyní vysvětlím na kódu jádra a také popíšu rozhraní, které jádro nabízí.

3.3.1 Příprava pro implementaci prostředí

Knihovna obsahuje ELGINE, který umožňuje vývojáři jednoduše implementovat vlastní prostředí. ELGINE ale není přímo zakomponovaný v jádru, jelikož by potom nešlo implementovat prostředí, které by nebylo vytvoření v ELGINE.

Proto jsem implementoval rozhraní tzv. modulu 3.13, kde si vývojář napojí jakýkoliv protředí (včetně prostředí vytvoření v ELGINE). Modul musí bý typu `AsmNeModule`, který reprezentuje funkci. Tato funkce dostane dva parametry. První je funkce, která jako argument přijímá pole čísel. Tato funkce bude spouštět algoritmus dopředného šíření a jako návratovou hodnotu vrátí výsledek dopředného šíření. Druhý parametr je `HTMLCanvasElement`, který je nepovinný. Pokud je tento parametr přítomný, tak to znamená, že se má simulace prostředí vykreslovat. Pokud není, tak je očekávané, že simulace proběhne bez vykreslovaní.

Tento běh je automaticky připravený v ELGINE, podle toho, jestli se zvolá metoda `Elgine.mount`. Ale pokud implementované prostředí není vytvoření přes ELGINE, tak stačí napojit ovládání na funkci `calculateAction`. Zároveň je potřeba rozlišovat mezi během s vykreslováním nebo bez, podle parametru `HTMLCanvasElement`.

Algoritmus dopředného šíření se děje v exportovaném objektu z `AsmCore.cpp`. Pro jednodušší práci jsem vytvořil pomocnou funkci, která tento proces abstrahuje do pomocné funkce `calculateOutputs` 3.14.

Funkce přijímá fenotyp, tudíž objekt z `AsmCore.cpp`, na kterém již proběhla metoda `AsmCore.buildGenome`. Potom přijímá třídu `Vector`, která je exportována přes Emscripten a nakonec pole hodnot pro vstup do algoritmu dopředného šíření.

Výpis kódu 3.13 Environment module

```
export type AsmNeModule = (
    calculateAction: (inputs: Array<number>) => Array<number>,
    canvas?: HTMLCanvasElement
) => Promise<number>;

const module: AsmNeModule =
    (calculateActions, canvas): Promise<number> =>
{

    return new Promise(resolve => {
        // run environment
        return fitness;
    })
};
export default module;
```

Výpis kódu 3.14 AsmNeUtils.ts

```
export const calculateOutputs = (
        phenotype: AsmCore,
        vector: typeof Vector,
        inputs: Array<number>
    ): Array<number> => {
    // fill vector
    const vectorInputs = new vector();
    inputs.forEach(val => vectorInputs.push_back(val));

    // do calculations
    const vectorOutputs = phenotype.forward(vectorInputs);

    // convert vector to array
    const outputs: Array<number> = [];
    for (let i = 0; i < vectorOutputs.size(); i++) {
        outputs.push(vectorOutputs.get(i));
    }

    vectorInputs.delete();
    vectorOutputs.delete();

    return outputs;
}
```

Funkce nejdřív vytvoří instanci C++ objektu `Vector` a převede hodnoty z Javascript pole, do C++ pole. Následně zavolá metodu dopředného šíření z fenotypu a výsledek se uloží do nové instance `Vector`. Výsledek se převede zpět na podobu Javascript pole a na obou objektech `Vector` se zavolá destruktor a z funkce se vrátí výsledek.

Je důležité zmínit, že je potřeba pracovat s exporotvanou třídou nafukovacího pole z C++, jelikož s Javascript polem neumí WASM pracovat. Tato metoda tedy abstrahuje redundantní práci, který budeme využívat při trénování a při vizuální instanci běhu.

3.3.2 Zpracovávání paralelního požadavku na výpočet

Jádro si pro výpočet vytvoří příslušný počet Web Workerů podle počtů jader k dispozici na daném systému. Pro spuštění nového Web Worker je potřeba mít separátní soubor s kódem. Proto jsem vytvořil `AsmNeWorker.ts` 3.15. Ten si po načtení přidá odposlouchávání na zprávy z hlavního vlákna.

Pokud příjde zpráva z hlavního vlákna, tak se z obsahu zprávy extrahují parametry. `module` určuje jaké prostředí se má načíst. Toto prostředí musí být uložené v složce `demo/modules/`. Parametr `algorithm` určuje, který algoritmus se má aplikovat. S tímto parametrem se zavolá konstruktor třídy `AsmCore.cpp`. Poslední parametr `genomes` je pole genotypů, které je potřeba ohodnotit.

Funkce nejdřív načte prostředí podle parametru `module`. Následně pro každý genom vytvoří fenotyp pomocí funkce `AsmCore.buildGenome` a na něm spustí simulaci. Pomocí funkce `calculateOutputs` vytvoříme anonymní funkci, kterou předáme běhu prostředí. Díky tomu může prostředí provolávat algoritmus dopředného šíření.

Po dokončení běhu prostředí se hodnota fitness funkce uloží do Pole. Jakmile se provede simulace všech genotypů, tak se výsledký vrátí do hlavního vlákna přes nativní metodu `postMessage`.

3.3.3 Třída AsmNeGym - běh neuroevoluce

Hlavní třída celé knihovny je `AsmNeGym`. Tato třídá kombinuje všechny doposud zahrnuté části a stará se o vystavení API. Celá třída je velmi obsáhlá a proto její obsah zjednoduším a rozdělím na popis běhu neuroevoluce a popis API. Pro přesný kód doporučuji nahlédnout přímo do repozitáře.

Běh neuroevoluce zařizují tři metody 3.16. Metoda `workerFitnessCalculation` se stará o správné předání spuštění kódu do Web Worker procesu. Také se stará o následné zpracování výsledků z Web Worker výpočtu.

`evaluateGeneration` má za úkol vyhodnotit celou generaci. Populaci rozdělí na jednotlivé genotypy. Tyto genotypy pak rovnoměrně rozdělí (podle počtu dostupných Web Workerů) a pro každý balíček genotypů spustí metodu `workerFitnessCalculation`. Jako návratovou hodnotu vrací hodnotu fitness funkce každého jedince z populace.

Poslední je metoda `train`, která zařizuje inicializaci procesu neuroevoluce. Při inicializaci se vytváří Web Workers, nastavují se ukončovací podmínky a vytvoří instanci `AsmCore`, kvůli funkcím `AsmCore.initialPopulation` a `AsmCore.generateGeneration` s příslušným algoritmem.

Po inicializaci probíhá iterativně evoluční algoritmus, který přes `AsmCore.generateGeneration` vygeneruje novou generaci. Ta se následně evaluuje přes metodu `evaluateGeneration` a tento proces se opakuje dokola.

Pro ukončení procesu trénování sítě se používají tři kritéria, které se dají určit jako parametr metody `train`. Počet iterací, doba běhu a specifická hodnota fitness funkce. Tyto podmínky dovolují specifikovat, jak dlouho bude trénování spuštěno. Metoda `checkCriterion` kontroluje, jestli některé z podmínek není splněná a pokud ano, tak se proces zastaví.

`train` je na rozdíl od předchozích dvou metod veřejná, tudíž se dá volat mimo interní kód třídy. Tato metoda je tedy API, které už přímo spouští vývojář co knihovnu používá.

■ **Výpis kódu 3.15** AsmNeWorker.ts

```
addEventListener(
    "message", async (e: MessageEvent<WorkerInputData>
) => {
    const { module, algorithm, genomes } = e.data;

    // load passed module
    const { default: environmentRun }: { default: AsmNeModule } =
        await import(`../demo/modules/${module}.ts`);

    const results: WorkerOutputData = [];
    for (let i = 0; i < genomes.length; i++) {
        const phenotype = new AsmCore(algorithm);
        phenotype.buildGenome(genomes[i]);

        fitness = await environmentRun(
            (inputs: Array<number>) => calculateOutputs(
                                            phenotype,
                                            Vector,
                                            inputs
                                        ),
            undefined
        );

        phenotype.delete();
        results.push(fitness);
    }

    postMessage(results);
});
```

Výpis kódu 3.16 AsmNeGym.ts:train

```
export class AsmNeGym {
    // ...

    async train (criterion: AsmNeGymTrainCriterion) {
        // ... pre-training setup
        const workerPool = this.createWorkerPool();

        while (true) {
            if (this.checkCriterion()) break;
            generation = AsmCore.generateGeneration();
            this.evaluateGeneration(workerPool, generation);
        }

        // ... post-training cleanup
    }

    private async evaluateGeneration(
        workerPool: Array<Worker>,
        generation: string
    ): Promise<Array<number>> {
        const genomes = this.splitGeneration(generation);
        const batchSize = Math.floor(genomes.length / workerPool.length);

        for (let i = 0; i < workerPool.length; i++) {
            const workerGenomes = genomes.splice(0, batchSize);

            this.workerFitnessCalculation(workerPool[i], workerGenomes)
                .then(fitness => result.push(fitness));
        }
        // ... wait for all workers to complete
        return result.sort();
    }

    private workerFitnessCalculation(
        worker: Worker, genomes: Array<string>
    ): Promise<WorkerOutputData> {
        return new Promise(resolve => {
            worker.addEventListener(
                "message",
                (e: MessageEvent<WorkerOutputData>) => resolve(e.data),
                { once: true }
            );

            const workerInput: WorkerInputData = {
                module: this.m_module,
                algorithm: this.m_algorithm,
                genomes: genomes,
            };
            worker.postMessage(workerInput);
        });
    };
}
```

Výpis kódu 3.17 AsmNeGym.ts:API

```
export class AsmNeGym {
    constructor(
        module: string,
        algorithm: string,
        hyperparameters: Array<number>
    );

    setModule(module: string): void;
    setAlgorithm(algorithm: string): void;
    setHyperparameters(hyperparameters: Array<number>): void;
    setThreads(threads: number): void;
    getPopulation(): string;
    setPopulation(population: string);
    forceStop(): void;

    getFitnessHistory(): Array<number>;
    clearFitnessHistory(): void;
    getBestGenome(): Genome;
    getLastGenome(): Genome;

    // ...
}
```

3.3.4 Třída AsmNeGym - API

Třída dovoluje vývojáři komunikovat a konfigurovat přes API metody 3.17.

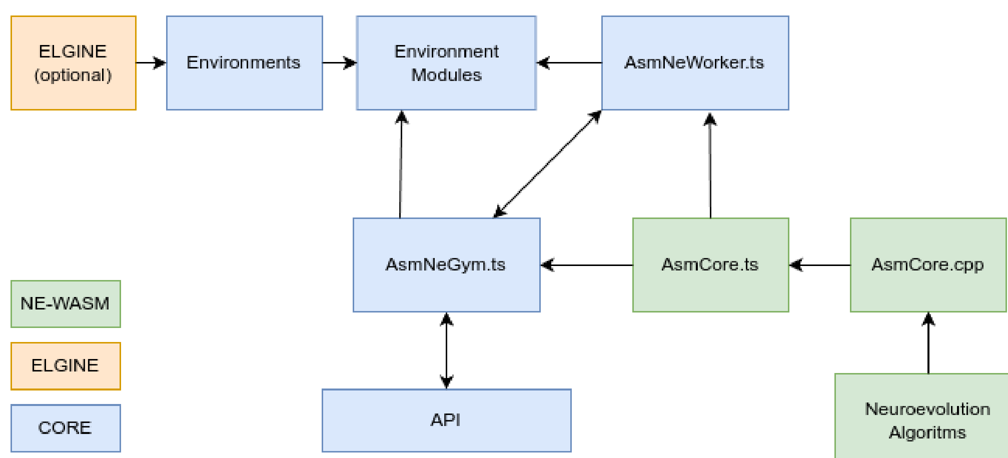
Pro základní konfiguraci slouží konstruktor třídy, kde se nastvuje modul pro určení prostředí, ve kterém bude probíhat neuroeovluce. Jako další parametr je algoritmus, který určuje jaký neuroevoluční algoritmus se použije pro trénování. Tento paramtr se dál propaguje do třídy `AsmCore`. Poslední povinný parametr je pole čísel, které reprezentují hyperparametry. Ty se pak dále propagují do metod `AsmCore`. Všechny tyto parametry jde upravit i po inicializaci třídy přes večejné metody třídy. Slouží k tomu `setModule`, `setAlgorithm`, `setHyperparameters`.

Z globální proměnné `navigator.hardwareConcurrency` se určí počáteční hodnota počtu vláken, na kterém trénování bude spuštěno. Pro nastavení paralelizace slouží metoda `setThreads`, která počet procesů při trénování nastaví explicitně.

Při běhu neuroevoluce je povinný aspoň jeden z ukončujcích parametrů. Pokud je však potřeba ukončit trénování dřív, tak k tomu slouží metoda `forceStop`. Ta proces trénování zastaví po doběhnutí aktivní generace a proběhne uložení výsledků.

Zbytek metod z API se soustředí na informace ohledně trénování. Při trénování si knihovna zaznamenává aktuální populaci uloženou jako řetězec. Nejlepší genotyp za celý průběh trénování a nejlepší genotyp z poslední generace. Tyto všechny inforamce lze následně získat přes metody `getPopulation`, `getBestGenome` a `getLastGenome`. Pro nastavení populace slouží metoda `setPopulation`. To umožňuje proces učení zastavit a budoucnu jí obnovit s poslední populací při které byl zastaven. Zároveň to umožňuje vytrénování nějaké populace na konkrétní úloze, kterou následně můžeme upravit a na stejné populaci pokračovat v lehce upraveném prostředí dané úlohy.

Pro statistické údaje o vývoji populace slouží metoda `getFitnessHistory`, která vrací pole hodnot fintess funkce nejlepšího jedince pro každou generaci. Toho využiji na demo stránce, pro vykreslení grafu. Historii je možné smazat pomocí metody `clearFitnessHistory`.



Obrázek 3.1 Propojení jednotlivých částí knihovny

3.3.5 Třída AsmNeGym - běh genotypu

Po procesu trénování sítě je potřeba nějakým způsobem vizualizovat výsledný genotyp z populace. Aby si vývojář spustil genotyp, tak si musí sám implementovat třídu `AsmCore`, kde bude muset sestavit genotyp a následně spustit prostředí s voláním metody `AsmCore.forward`.

Pro usnadnění je připravená metoda `runGenome` 3.18. Ta řeší proces, který jsem popsal nahoře. Využívá k tomu již nastavené parametry jako je prostředí, na kterém se má genotyp spustit, nebo algoritmus, který se má použít. Kód je skoro identický, jako kód pro běh v Web Workeru. Rozdíl je v parametru `HTMLCanvasElement`, který určuje, že se má simulované prostředí vizualizovat.

Další dvě metody `runBestGenome` a `runLastGenome` pouze přepoužívají již zmíněnou metodu `runGenome`. Stejná jako metoda `runGenome` přijímá parametr `HTMLCanvasElement`, ale genotyp je předem určený. Buď se jedná o dosud nejlepší genotyp, nebo o nejlepší genotyp z poslední generace.

3.4 Shrutí architektury

Celá knihovna má hodně částí, které jsou navzájem propojené a po popisu jednotlivých částí nemusí být úplně jasné, jak jsou navzíjem propojené. Proto ještě shrnu propojení jednotlivých částí knihovny, které je znázorněné v obrázku 3.1.

Hlavní třída je `AsmNeGym`, která vystavuje API vývojáři, který knihovnu implementuje. `AsmNeGym` používá z `AsmCore` metody `AsmCore.initialPopulation` a `AsmCore.generateGeneration` pro tvorbu prvotní populace a následně pro generování nové generace.

`AsmNeWorker` je spouštěn ve vlastním procesu hlavním vláknem, přes technologii Web Worker a slouží pro evaluaci genotypů. Nejdřív si načte prostředí přes jeho modul a následně si načte `AsmCore`, kde spustí `AsmCore.buildGenome` a pak provádí `AsmCore.forward`, při běhu prostředí. Tento proces se opakuje pro každý genotyp a do hlavního vlákna vrací výsledky.

`AsmCore` je C++ kód exportovaný přes Emscripten do WebAssembly podoby, která je spustitelná v webovém prostředí. Do `AsmCore` jde implementovat vlastní neuroeovluční algoritmus, pomocí abstraktní třídy `CNeuroevolutionBase`.

Modul pro prostředí jako parametr přijímá funkci, která v pozadí dělá algoritmus dopředného šíření. Jako druhý parametr přijímá `HTMLCanvasElement`, pokud má být běh prostředí vizualizován. Modul se spouští v `AsmNeWorker` jako běh bez vizualizace, nebo v třídě `AsmNeGym` s vizualizací.

Výpis kódu 3.18 AsmNeGym.ts:run

```
export class AsmNeGym {
    // ...

    private m_bestGenome;
    private m_lastGenome;

    async runBestGenome(canvas: HTMLCanvasElement): Promise<number> {
        return await this.runGenome(this.m_bestGenome.genome, canvas);
    }

    async runLastGenome(canvas: HTMLCanvasElement): Promise<number> {
        return await this.runGenome(this.m_lastGenome.genome, canvas);
    }

    async runGenome(
        genome: string,
        canvas: HTMLCanvasElement
    ): Promise<number> {
        const { default: environmentRun }: { default: AsmNeModule } =
            await import(`../demo/modules/${this.m_module}.ts`);
        const { Vector, AsmCore } = await AsmCoreWasm();

        const phenotype = new AsmCore(this.m_algorithm);
        phenotype.buildGenome(genome);
        return await environmentRun(
            (inputs: Array<number>) => calculateOutputs(
                phenotype,
                Vector,
                inputs
            ),
            canvas
        );
    }
}
```

Pro implementaci vlastního prostředí je dostupný ELGINE, což je herní engine, který umožňuje jednoduchou implementaci vlastního prostředí. ELGINE má v sobě zabudovaný mechanismy pro běh bez vykreslování, což zrychluje běh prostředí a díky tomu i proces trénování.

Kapitola 4

Implementace benchmarkových úloh a neuroevolučních algoritmů

Pro ukázku funkčnosti knihovny budu implementovat dvě hry, které budou reprezentovat prostředí pro posilované učení a dva algoritmy neuroevoluce. Představím pouze základní principy, bez podrobného rozebírání kódu, protože se nejedná o samotnou knihovnu. Kód benchmarkových úloh a neuroevolučních algoritmů je veřejně dostupný v open-source repozitáři.

4.1 Flappy bird

První benchmarková úloha je populární hra flappy bird. Hráč ovládá malé ptáče, které se snaží proletět co nejdále mezi řadami trubek, které se liší výškou a jsou rozmístěny ve vertikálních rozestupech. Ptáče bude reprezentovat agenta. Hra je ovládána jediným tlačítkem nebo dotykem na obrazovku, který způsobí, že agent mírně vzlétne vzhůru. Když hráč tlačítko nepoužívá, agent klesá dolů kvůli gravitaci. Cílem hry je dostat se co nejdále bez toho, aby agent narazil do trubek nebo spadl na zem. Hra končí, když agent narazí do trubky nebo spadne na zem, a hráčovo skóre je výsledkem počtu uletěných pixelů.

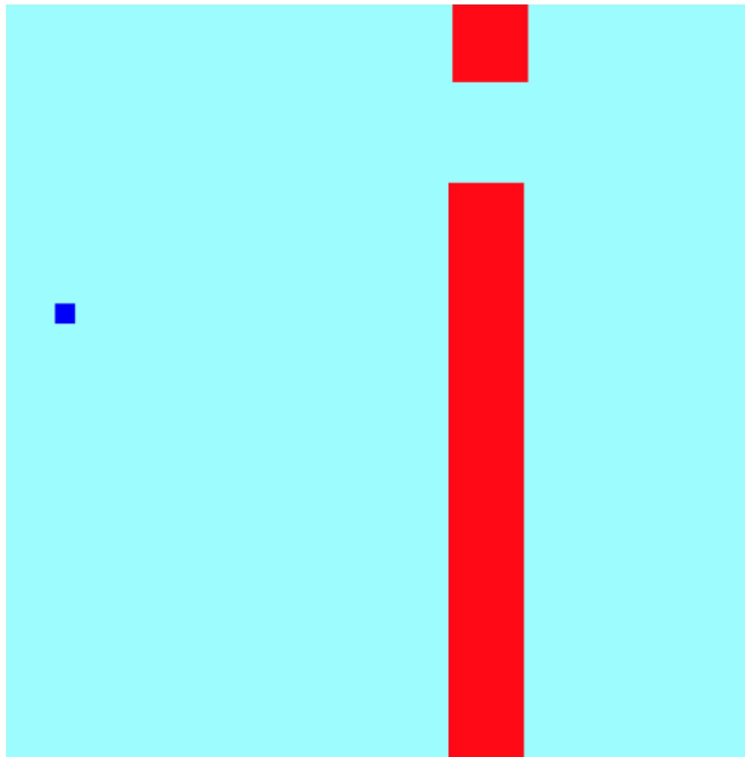
Pro vytvoření stejné hry pomocí ELGINE si vytvoříme dve základních entit. První bude `CBird`, který reprezentuje agenta. Entita postupně v update metodě simuluje pád. Zároveň pokud je aktivována akce skok, tak agent vyskočí. Při nárazu do trubky se hra ukončí a výsledek je zpracován dál.

Druhý entita je `CPipe`. Jedná se o jednoduchou entitu, která se pouze posouvá směrem k agentovi. Pokud se pozice trubky dostane mimo viditelné pole, tak se odstraní a vygeneruje se nová trubka. Ta bude mít oproti předchozí náhodnou pozici díry, kterou má za úkol agent proletět. Jelikož hra může být nekonečná, maximální povolená doba běhu jsou dvě minuty (v kódu to je vyjádřeno jako 120 000 tiků). Graficky je agent reprezentován jako modrý čtverec. Trubky jsou velké červené obdelníky.

Pro použití neuroevoluce je potřeba určit, co bude reprezentovat vstup do neuronové sítě. Není potřeba dávat každý pixel z prostředí, ale stačí pouze nezbytné informace pro překonání trubky.

Budu tedy zaznamenávat:

- pozice ptáčka na mapě (jedná se pouze o vertikální pozici)
- rychlost ptáčka (kladná pokud letí nahoru, nebo záporná pokud padá k zemi)
- vzdálenost trubky od ptáčka



Obrázek 4.1 Benchmarková úloha flappy bird

- vertikální pozice mezery v trubce

Tyto informace by měli být dostatečné, aby se neuronová síť naučila. Výstup z neuronové sítě bude aktivovat akci skok.

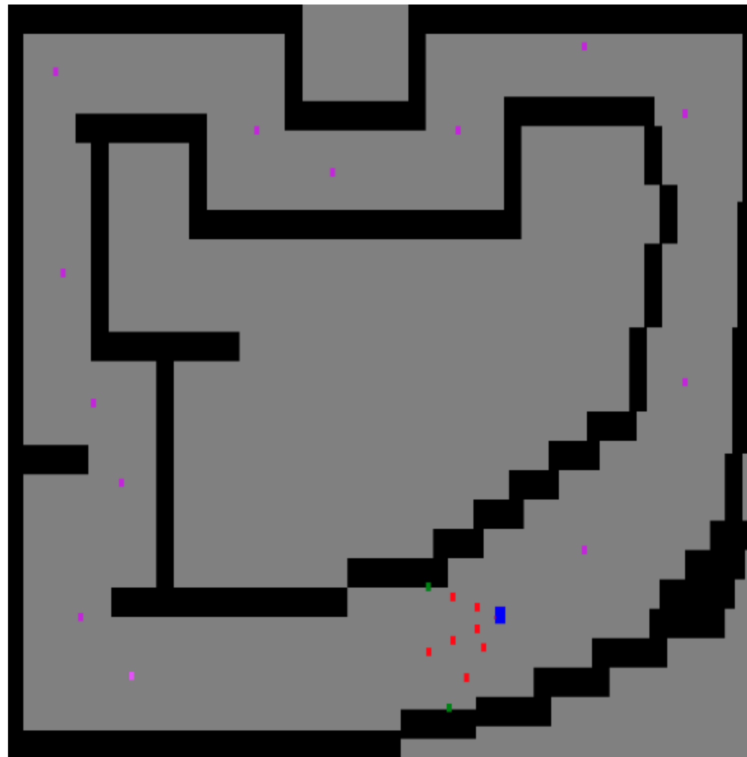
Hrá je naschvál jednoduchá. Ze vstupu má neuronová síť veškeré informace z prostředí a zároveň má pouze jednu akci. Proto je tato úloha koncipovaná jako testovací, na které otestuju funkčnost neuroevolučních algoritmů.

4.2 Závodní auto

Druhý benchmarková úloha je závodní auto. Auto bude reprezentovat agenta. Auto má dva parametry. Rychlost, která mu dává možnost zpomalovat nebo zrychlovat (tato hodnota má dolní a horní omezení). Druhým parametrem je úhel, kterým se určuje směr auto v 2D prostoru. Změna úhlu je pomalejší, čím rychleji auto jede. To by mělo simulovat reálný svět, kde při vysoké rychlosti nejde změnit směr jízdy tak jednoduše jako při pomalé jízdě.

Hra se skládá ze dvou základních entit a jedné pomocné. Entita `CCar` reprezentuje agenta. V aktualizační metodě se provádí úprava úhlu a rychlosti, podle aktivních akcí agenta. Zároveň se aktualizuje pozice. Druhá entita `CBarrier` slouží pro sestavené dráhy, kde bude auto jezdit. Mapa je předem určená podle bariér. Cílem auto je se naučit dráhu projet. Pokud nararzí do bariéry tak hra končí.

Pomocná entita `CCarSensor` slouží jako senzor auta, který se aktivuje pokud bude v kolizi s bariérou. Skupina seonzorů budou postaveny do kuželovité formace směrem, kterým jede auto 4.2. Jako vjemy z prostředí budou stavy senzorů (děvet vstupů) a informace o aktuálním stavu auta. Tudíž bude mít agent omezenou informaci z prostředí. Jelikož mají senzory pevně danou vzdálenost od auta, tak agent nepozná jak daleko je od bariéry, která je za nejvzdálenějším



Obrázek 4.2 Benchmarková úloha závodní auto

senzorem. Jako výstup má agent čtyři akce. Zvýšení nebo snížení akcelerace a změna úhlu (tudíž zatáčení doleva/doprava).

Úloha již není tak jednoduchá, jelikož je zde více vstupů (celkem 11) a agent má čtyři možné akce. Zároveň nemá úplnou informaci o prostředí a „vidí“ pouze přes senzory. Aby byl agent schopný najít řešení, musí se přidat určité mechanismy do simulace, které pomůžou s optimalizací.

Algoritmus nemá kontext simulace, tudíž se musí chtěné chování promítnout do skóre agenta. Proto bude dostávat body za počet odcestovaných pixelů. Aby se však agent nezačal pouze točit dokola, tak přidám mechanismus, který bude detekovat opakované pohyby na jednom místě. Daší problém nastává při směru jízdy, nebo při volbě cesty. Agent, který zvolí delší cestu zakončenou nárazem je lépe hodnocená, než pokus projet kratší a jistější cestou. Proto na trať přidám body, které budou agenta postupně odměňovat za jejich dosažení. Zároveň při nárazu agenta, bude dodatečně odměněn podle toho, jak blízko byl dalšímu bodu.

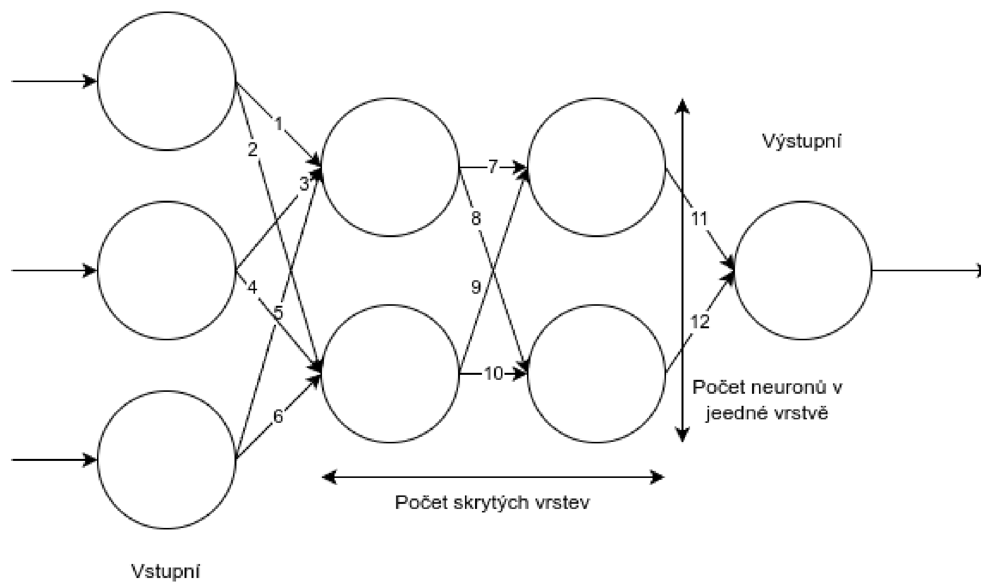
Tyto mechanismy se budou snažit usnadnit učení a vyhnout se lokálním extrémům nebo nechtěnému chování.

4.3 Konvenční neuroevoluce (CNE)

Jako první budu implementovat konvenční neuroevoluci. Není to žádný konkrétní algoritmus, pouze základní principy neuroevoluce.

Tento algoritmus bude optimalizovat pouze váhy, tudíž bude mít pevně danou architekturu sítě. Struktura sítě se bude skládat z vrstev neuronů a každá vrstva bude plně propojená s okolními vrstvami

Tu se bude určovat pomocí dvou hyperparametrů. Jeden bude určovat počet skrytých vrstev a druhý bude určovat počet neuronů v vrstvě. Další hyperparametry budou určovat vstupní



Obrázek 4.3 Ukázka fenotypu z CNE

neurony, výstupní neurony, velikost populace a šanci na mutaci.

4.3.1 Kódování

Pro zakódování struktury sítě stačí pouze infromace o počtu vstupních/výstupních neuronech a k tomu počet skrytých vrstev a počet neuronů v jedné skryté verstvě. Díky tomu se může odvodit přesnýé neurony v fenotypu. Zakódujeme si všechny tyto informace v číselné podobě do genotypu. Druhá část kódování bude obsahovat synapse mezi neurony. K tomu je potřeba vědět z jakého neuronu synapse vychází, do jakého neuronu vstupuje a její váhu. Jelikož je architektura daná, tak víme kolik synapsí bude a jaké budou propojovat neurony. Proto stačí pouze váha synapse a zbytek si lze odvodit.

Na obrázku 4.3 je vidět neuronová síť, kde jsou tři vstupní neurony a jeden výstupní. Síť má dvě skryté vrstvy po dvou neuronech. Synapse jsou indexované zvrchu dolů a pak zleva doprava.

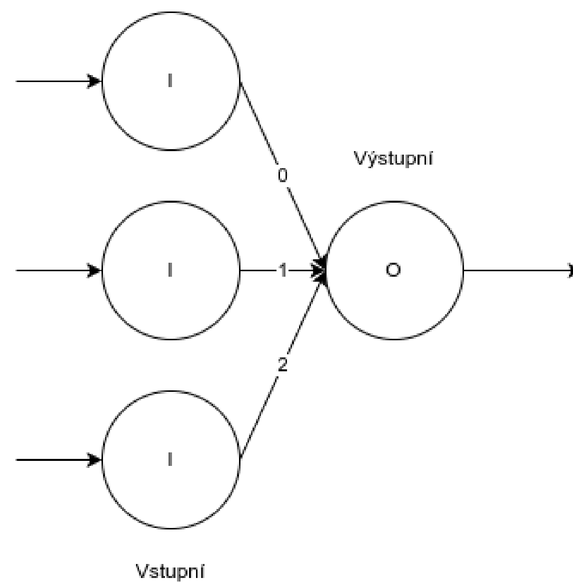
Příklad zákódovaného genotypu z příkladu v 4.3 je

```
"3;1;2;2|6.803754;-2.112341;5.661984;5.968801;8.232947;-6.048973;-3.295545;
5.364592;-4.444506;1.079399;-0.452059;2.577418"
```

4.3.2 Generování nové generace

Při generování nové generace je vybráno nejlepších 15 % jedinců z populace. Ze zbylých jedinců se stanou potencionální rodiči, kteří mají šanci být vybrání k generaci nového potomka. Při generování zvyšuje šanci na výběr větší hodnota fitness funkce jedince.

Na začátku tvorby potomka z nové generace se nejdřív vyberou dva rodiče. Jelikož je architektura pevně daná, tak víme přesně které synapse mezi sebou můžeme křížit. Náhodně se vybere jestli konkrétní synapse bude převzata z prvního nebo druhého rodiče. S určitou šancí se může vygenerovat nová váha. Tato šance se určuje podle hyperparametru.



Obrázek 4.4 Ukázka fenotypu z NEAT

4.4 NEAT

Algoritmus NEAT jsem již popisoval podrobněji v sekci 1.5.6. Pro jednoduchost implementace nebudud dělat implementaci zachování druhů, která populaci rozdělí na druhy, které se pak porovnávají mezi sebou.

Základní princip NEATu je proměnlivá architektura neuronové sítě a zároveň optimalizace vah. Počáteční genotyp budou pouze vstupní a výstpní neurony navzájem propojeny.

4.4.1 Kódování

Kódování je zde složitější, jelikož je architektura proměnlivá. Protože se může měnit počet neuronů, tak budou zakódovány explicitně. Použijeme na to jeden znak a kódování bude následující:

- „I“ - vstupní neuron
- „H“ - skrytý neuron
- „O“ - výstupní neuron

Jejich pořadí bude určovat jejich indexy, které využiji u kódování synapsí.

Druhá část bude zakódování synapsí. Jeliokož se jedná o promněnlivá strukturu, tak se musí držet všechny informace ohledně sypase. Proto pro každou synapsi zakóduju informaci o tom, jaké neurony spojuje, váhu synapse, informaci o tom jestli je synapse aktivní a historickou značku, podle které pak budeme provádět křížení.

Na obrázku 4.3 je vidět neuronová síť, kde jsou tři vstupní neurony a jeden výstupní. Na fenotypu je vidět označení jednotlivých neuronů a u synapsí je vidět historická značka, která je nutná pro křížení.

Příklad zakódovaného genotypu z příkladu v 4.4 je

```
IIIO|0,3,1.000000,0,1;1,3,1.000000,1,1;2,3,1.000000,2,1
```

4.4.2 Generování nové generace

Při generování nové generace se stejně jako u konvenční neuroevoluce vybere nejlepší 15 % jedinců, ze kterých se stanou rodiče pro další generaci. Poté se náhodně vyberou dva jedinci jako rodiči nového potomka.

Při tvorbě genotypu se náhodně vyberou dva jedinci, ze kterých se následně berou jednotlivé genomy. Přebytečné genomy se berou pouze z lepšího ze dvou rodičů. Přebytečné genomy horšího rodiče se zahazují. Stejné genotypy se náhodně vyberou z jednoho, nebo z druhého rodiče.

Po vytvoření nového potomka se s nastavenou mírou mutace může provést ještě operace mutace. Pokud se tak stane, tak se náhodně vybere náhodná mutace, ze tří možných. Největší šanci má mutace váhy. Ta vybere náhodnou váhu a té změní váhů na náhodnou hodnotu. Druhá možná mutace je přidání synapse. Ta se přidá mezi neurony, které ještě mezi sebou nemají žádné spojení. U této operace je důležité dávat pozor na vytvoření cyklů. Proto se při přidání nové synapse se musí otestovat, jestli nevznikl smyčka. Poslední mutace s nejmenší šancí je přidání nového neuronu, která vybere náhodnou synapsi, kterou zneaktivní. Poté přidá neuron a spojí ho dvěma synapsemi tak, aby tyto synapse nahradili původní.

4.5 Testovací stránka

Pro spojení všech dosud vytvořených částí (knihovna, benchmarkové úlohy a algoritmy) jsem vytvořil demo stránku 4.5. Ta bude využívat třídu `AsmNeGym` a bude obsahvat uživatelské rozhraní pro ovládání různých částí knihovny.

Hlavní část demo stránky obsahuje běh prostředí v vykreslovacím módu. Pod ním se nachází kontrolní panel. V něm lze sputit proces trénování pomocí tlačítka „Train“. V tu chvíli se spustí neuroevoluce. Zároveň se spustí simulovaný běh nejlepšího genotypu z poslední generace. Při vykreslované simulaci prostředí na pozadí probíhá trénování. Po skončení vykreslované simulace se vezme nejlepší genotyp z poslední generace a proces se opakuje. Pro zastavení procesu učení slouží tlačítko „Stop next iteration“, který po skončení ukončí trénování. Pro běh nejlepšího genotypu slouží tlačítko „Run best genome“. Ta je napojená na metodu `AsmNeGym.runBestGenome`.

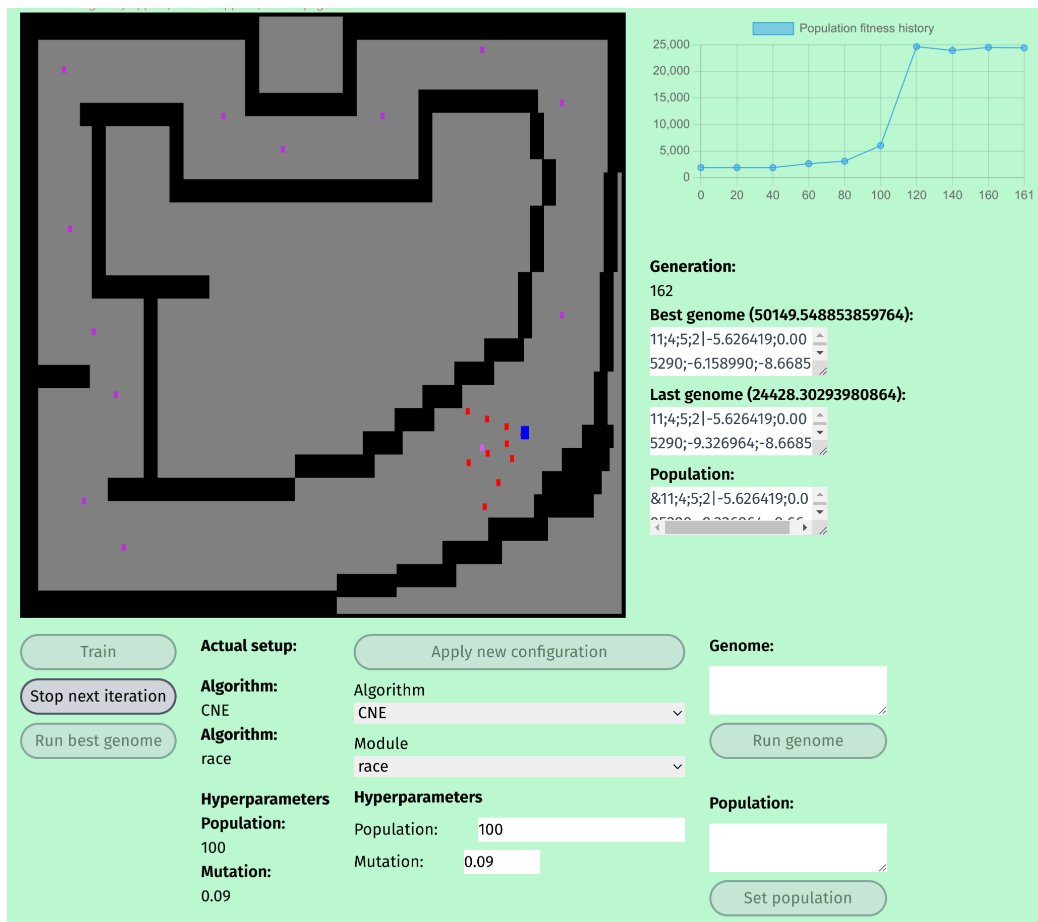
Další část ovládací panelu je informativní a vypisuje aktuální konfiguraci třídy `AsmNeGym`. Zde jsou informace o aktivním algoritmu, prostředí a hyperparametry. Další část panelu slouží pro nastavení této konfigurace. Pro aplikování nastavených parametrů je potřeba potvrdit výběr přes tlačítko „Apply new configuration“.

Poslední část ovládacího panelu je pro nastavení trénované populace, nebo běh konkrétního genotypu. Pokud budu trénovat a chci si trénovanou populaci uložit a následně znova sputit, tak jí přes „Set population“ mohu znova načíst. Pokud budu chtít sdílet nejlepší genotyp, tak mohu poslat pouze genotyp. Následně tento genotyp přes „Run genome“ mohu spustit na svém stroji.

Poslední část vedle vykreslovaného běhu prostředí jsou informace ohledně trénování. Nachází se zde graf hodnot fitness funkce jednotlivých generací. Pomocí funkce `AsmNeGym.getFitnessHistory` se získají všechny data. Z nich se udělá průžeř pouze na deset hodnot a ty se vykreslí na graf. Tento graf se postupně v průběhu neuroevoluce aktualizuje.

Další jsou informace ohledně aktuální generace, nejlepší genotyp poslední generace, nebo aktuální populace. Ta může být sdílena a opět nastavená přes kontrolní panel.

Tyto funkce jsou pouze ukázkové a s API, které `AsmNeGym` vystavuje, by šlo udělat mnohem více funkcionalit. Demo je v rámci open-source repozitáře dostupná na stránce https://prostenobody.github.io/asm-ne-gym/.



Obrázek 4.5 Ukázka demo stránky

Kapitola 5

Výsledky a evaluace

V této kapitole vyhodnotím hypotézy, které budou vypovídat o funkčnosti knihovny a implementovaných řešení. Pro úpravu konfigurace budu používat API knihovny `AsmNeGym`. Pro získávání dat budu používat funkci `getFitnessHistory`. Funkce `getFitnessHistory` zaznamenává pouze nejlepšího jedince z populace, tudíž zde budu zanedbávat celkový stav populace. Díky tomu se ztratí informace, která by mohla mít dobrou vypovídací hodnotu. Do budoucna by se API mohlo rozšířit, aby nabízel i komplexnější informace jako je tato. Pro experimenty, které budu v této sekci dělat to však význam značně nezmění, ale je potřeba tento fakt brát v potaz.

Exporotvané výsledky experimentů budu exportovat do python skriptu, kde následně provedu vizualizaci přes nástroj `seaborn`[1]. Veškeré data z experimentů jsou dostupné v příloze ve složce `/graphs`, včetně skriptu na zpracování dat do grafů.

Pro experimenty jsou ze sekce 4 přichystané dva neuroeovluční algoritmy (NEAT a CNE) a dvě benchmarkové úlohy (Flappy bird a závodní auto). CNE je jednoduchý algoritmus, který jednoduše používá základní prvky neuroeovluce. Má pevně danou architekturu, tudíž se může předpokládat, že se hodí na jednoduché úlohy, kde dokážeme odhadnout potřebnou velikost sítě. Druhý je NEAT, který začíná s minimální architekturou sítě, ale postupem se může vyvinout do komplexní architektury. V obou algoritmech je velký aspekt náhody, jelikož výběr genomů z rodiče se děje náhodně. Zároveň mutace se mohou náhodně upravit a stejně tak se mohou objevit v náhodných místech. Lze říct, že v algoritmu NEAT je větší míra náhody. To je kvůli většímu množství operací, které se mohou při mutaci provést. Stejně tak se na náhodné místo může přidat neuron nebo synapse.

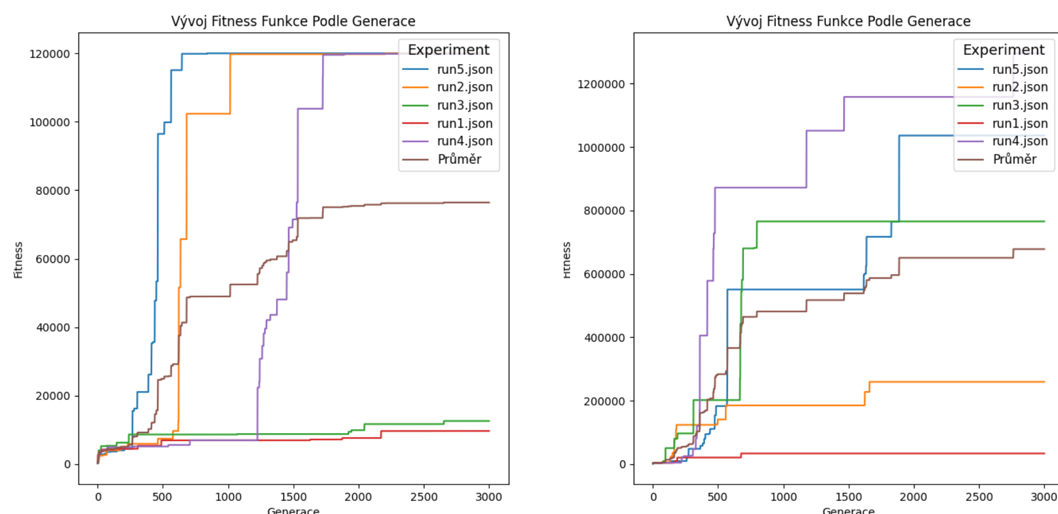
Benchmarková úlohy flappy bird reprezentuje jednoduchou úlohu. Agent má čtyři vstupy a jeden výstup. V úlože je míra náhodu v pozici díry pro průlet agenta v trubce. Tudíž se nejdná o konzistentní prostředí. Úloha závodního auta představuje komplexnější úlohu. Agent má jedenáct vstupů a čtyři výstupy. Žádná míra náhody v prostředí není přítomná, tudíž se jedná o konzistentní prostředí. Tato úloha slouží pro komplexnější experimenty.

Pro experimenty používám osobní notebooku ROG Zephyrus G14, jeho parametry jsou:

- Procesor - AMD Ryzen™ 9 6900HS Mobile Processor
- Jádra - 8 jader procesoru
- Vlákna - 16 vláken na 8 procesorech pomocí technologie hyper-threading
- RAM - 16GB DDR5 on board + 16GB DDR5-4800 SO-DIMM

Důležitý je počet jader procesoru, který využiji v sekci 5.2.

[1] https://seaborn.pydata.org/



(a) Experiment 1 - flappy bird

(b) Experiment 1 - závodní auto

Obrázek 5.1 Experiment 1

5.1 Ověření funkčnosti řešení

V prvním experimentu otestuji funkčnost celé knihovny. Funkčnost budu testovat jak na úloze flappy bird, tak na úloze závodního auta.

Jako úspěch ve hře flappy bird bude skóre větší než 50 000 a v úloze závodního auta to bude skóre větší než 60 000. Tyto hodnoty fitness funkce budu používat i v ostatních experimentech. Hyperparametr populace bude 100 a míra mutace bude 0.09. Tyto hodnoty jsou zvolené náhodně. Jako algoritmus jsem zvolil jednodušší CNE, jelikož je to primitivní implementace neuroeovluce. Tudíž je menší šance, že při neúspěchu by se mohlo jednat o chybu v implementaci algoritmu. Speciální hyperparametry tohoto algoritmu slouží k nastavení architektury sítě. Použiji tedy dvě skryté vrstvy a pět neuronů v jedné skryté vrstvě.

Pro testování využiju připravené API třídy `AsmNeGym`, které umožňuje nastavit ukončující podmínku. Jako podmínku jsem zvolil 3000 iterací. Zároveň budu běh opakovat a následně udělám průměr výpočtu Tento princip budu využívat i u ostatních experimentů.

Z grafů 5.1 je vidět, že obě úlohy byly vyřešené úspěšně. Výsledky běhu flappy bird 5.1a se tři z pěti běhů podařilo úlohu úspěšně vyřešit. Ve dvou případech se populace dostala do lokálního maxima.

V grafu výsledků úlohy závodního auta 5.1b se podařilo získat mnohonásobné skóre, než jsem požadoval (některé populace přesáhli fitness skóre přes milión). Je však vidět, že některé populace se taktéž zasekli v lokálním maximu.

Výsledky prmůěmrů tohoto pokusu jsou úspěšné. Díky tomu můžu potvrdit, že knihovna i implementace je funkční a dokáže řešit úlohu neuroeovluce.

5.2 Otestování akcelerace paralelizací výpočtu

V předchozím experimentu jsem ověřil funkčnost knihovny. Další funkce, kterou knihovna umožňuje, je paralelizace výpočtu. Pomocí API `AsmNeGym.setThreads` lze nastavit míra paralelizace. Ta je při inicializaci nastavená na maximalní počet, který používaný stroj umožňuje.

Tabulka 5.1 Srovnání výkonu podle počtu jader

1 jádro	4 jádra	16 jader
290.3759s	85.9592s	25.7598s
157.1004s	96.8112s	25.448s
183.8577s	165.0749s	36.1415s

Proto vyzkouším nastavení paralelizace na jedno, čtyři a šestnáct jader. Očekávaný výsledek by mělo být zrychlení výpočtu.

Pro získání času, jak dlouho výpočt trval použiju nativní webovou funkci `performance.now`[2]. Jako úlohu pro tuto úlohu jsem zvolil závodní auto a CNE. Velikost populace bude 100 a míra mutace bude 0.09. Architektura sítě budou dvě skryté vrstvy a pět neuronů v jedné vrstvě.

Výsledek bude čas, za který se zvládne vypočítat tisíc generací s touto konfigurací úlohy neuroeovluce. K tomu využiju API, které umožňuje ukončení po určitém počtu iterací.

Každou míru paralelizace budu opakovat třikrát, aby se vyhnulo ojedinělím připadům, kde se agent zasekl v lokálním maximu, nebo náhodou našel optimální řešení nadprůměrně rychle.

Výsledky je možné vidět v tabulce 5.1. Z výsledků je vidět, že je zde výrazné zrychlení. Mezi využitím jednoho jádra a šestnácti je asi šestinásobné zrychlení. Z časů je tedy vidět, že paralelizace výpočtu opravdu zrychlila výpočet a je funkční.

Z některých hodnot tabulky je vidět jistá nekonzistence a markantní nárůst času. To je způsobeno řešenou úlohou. Pokud bude optiomální řešení nalezeno v počátečních generací a agent je tudíž úspěšný v řešení úlohy, tak simulace prostředí trvá dýl. To znamená, že je potřeba větší výpočetní výkon na simulaci populace a tudíž i zvýšení času pro výpočet. Dá se však pozorovat, že při jednom jádru se při tomto případu čas zvětší dvojnásobně. Pokud využijeme šestnáct jader, tak se čas změní pouze o zlomek běžné hodnoty.

Další zajímavé pozorování je míra zrychlení. Ta není přímo úměrná počtu jader. To je způsobeno tím, že je zde režie navíc, která je způsobena přepínáním jader, které zabere nějaký čas navíc. Další problém je pak synchronizace vláken. Po rozdělení genotypů do vláken se může stát, že výpočet jednoho jádra bude výrazně pomalejší při výpočtu. Tudíž se při čekání na dokončení všech vláken může stát, že jedno vlákno bude zbytečně blokovat ostatní a zvýší se čas výpočtu.

5.3 Experimentování s velikosti sítě algoritmu CNE

V dalších experimentech se budu věnovat optimalizaci hyperparametrů. Tento problém budu řešit na úloze závodního auta, jelikož se jedná o komplexnější úlohu, bez míry náhody. V posledním experimentu použiju nalezené hodnoty pro porovnání algoritmů z pohledu rychlosti konvergence.

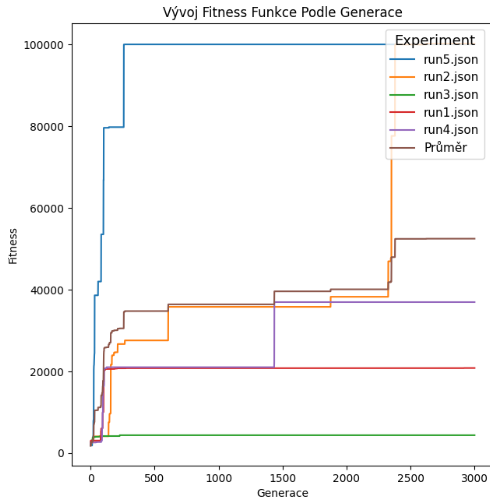
Jako první budu hledat optimální nastavení architektury sítě pro CNE. Velikost populace a míru mutace pevně zafixuji na hodnoty 100 a 0.09. Testovat budu tři konfigurace. Jedna konfigurace bude mít dvě skryté vrstvy a dva neurony v skryté vrstvě. Jako další budou dvě skryté vrstvy a čtyři neurony v skryté vrstvě a jako poslední budou čtyři srykté vrstvy a dva neurony v skryté vrstvě. Poslední vyzkouším architekturu, kde budou čtyři vrstvy a čtyři neurony za vrstvu.

Pro rychlejší výpočet také přidám limit ukončení hry při dosažení skóre 100 000. Jelikož jako úspěšně optimalizovanou síť beru v rámci experimentu 60 000, tak to nevadí.

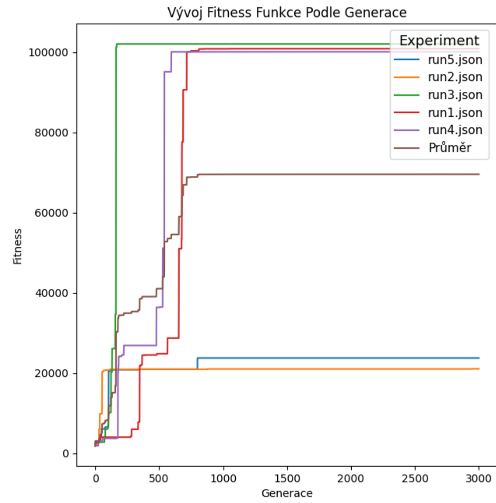
Výsledky jsou vidět v grafu 5.2. Experiment jsem zopakoval třikrát z důvodu snížení vlivu náhodných faktorů. Architektura 2x2 5.2a v průměru nedosála požadovaného výsledku. Ze všech běhů uspěli pouze dvě a druhá uspěla až okolo generaci 2 500. Výsledek by mohl být způsobený tím, že je velikost sítě moc malá pro tuto úlohu.

Nejlépe si vedla architektura 4x2 5.2b, která v průměru dosáhla požadovaného skóre. Tuto architekturu zvolím v posledním experimentu. Tři běhy dosáhli požadovaného skóre a další dva

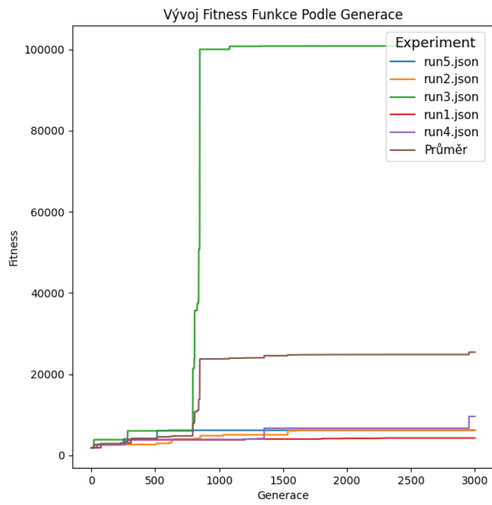
[2]https://developer.mozilla.org/en-US/docs/Web/API/Performance/now



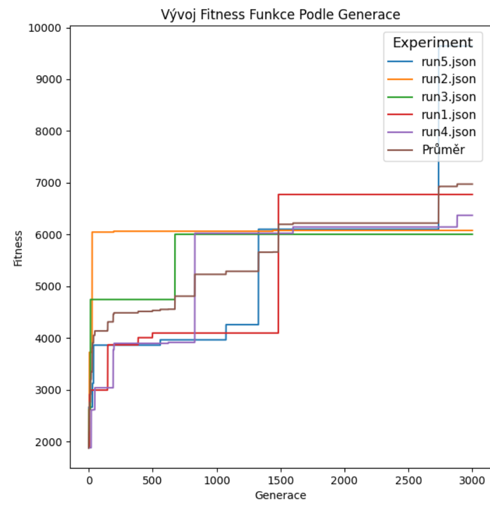
(a) Experiment 3 - achitektura 2x2



(b) Experiment 3 - architektura 4x2



(c) Experiment 3 - architektura 2x4



(d) Experiment 3 - architektura 4x4

Obrázek 5.2 Experiment 3

dosáhli pouze lokálního maxima 20 000. Úspěšné běhy dosáhly úspěchu před generací 1 000.

Architektura 2x4 5.2a byla druhá nejhorší. Pouze jeden běh dosáhl požadovaného výsledku. Ostatní běhy se zasekli v lokálním maximu, které dosahovalo okolo skóre 10 000. Problém může být „úzká“ architektura.

Poslední architektura byla 4x4 5.2d, která byla ze všech běhů nejhorší. Žádný z běhů nedosáhl požadovaného skóre. Větší architektura si tedy vedla hůře, jak menší architektura 2x2. Toto může být způsobené tím, že je velikost sítě až moc velká. Počet synapsí v této síti je 72, oproti nejlepší architektuře, která má pouze 40 synapsí k optimalizaci.

5.4 Experimentování s hyperparametry mutace algoritmu NEAT

Další experiment bude optimalizace míry mutace algoritmu NEAT. NEAT nemá pevně danou architekturu. Ta se vyvijí pomocí mutace. Takže oproti CNE, kde se pomocí mutace upravujou váhy synapsí, algoritmus NEAT v rámci mutace upravuje i architekturu sítě. Proto je důležité najít optimalní hodnotu tohoto hyperparametru.

Jako benchmarkovou úlohu opět zvolím závodní auto a velikost populace fixně nastavím na 100. Testovací hodnoty budou `0.10`, `0.20`, `0.30` a `0.40`.

Výsledky které jsou vidět v grafu 5.3 ukazují výsledky celkem dvaceti běhů s různými parametry. Při mutaci s hodnotou `0.10` jsem dostal nejhorší výsledky. Ani jeden běh nedosáhl požadovaného skóre. Problém může bude v tom, že je míra mutace moc malá, tudíž se ani po 3 000 iterací nezvládne poskládat dobrá architektura.

Další tří hodnoty mají podobnou průměrnou hodnotu z výsledků a to okolo 30 000. Při mutaci `0.30` už jeden běh dosáhl maximálního skóre, oproti mutaci `0.10`. Jeho průměrná hodnota je pod 20 000.

Další dvě hodnoty `0.20` a `0.40` skóre nad 20 000. Ale hodnota mutace `0.20` má pouze jedno úspěšné řešení, tudíž hodnota `0.40` bude lepší volba, jelikož má dvě řešení, které dosahují nad hodnotu 20 000.

Efektivita algoritmu NEAT je z těchto výsledků diskutabilní. Tento problém popíšu a zkusím objasnit důvody pro tyto výsledky v další sekci.

5.5 Porovnání implementovaných algoritmů NEAT a CNE

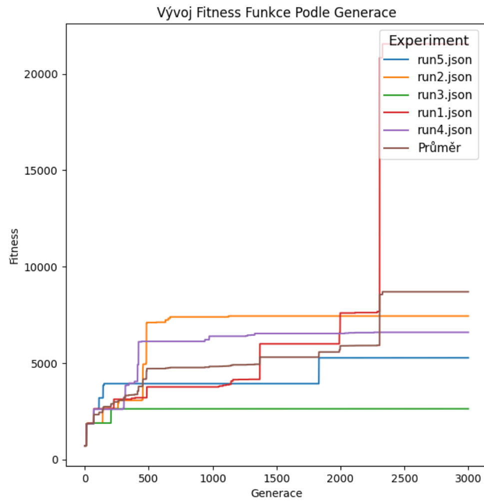
V této sekci porovnám algoritmus CNE a NEAT podle jejich výsledků z přechožích běhů. Pro porovnání vyberu data z nejlepších hyperparametrů pro jednotlivé algoritmy, které jsem získal v předchozích experimentech.

Pro CNE zvolím architekturu 4x2 a pro NEAT zvolím data s mírou mutace `0.40`.

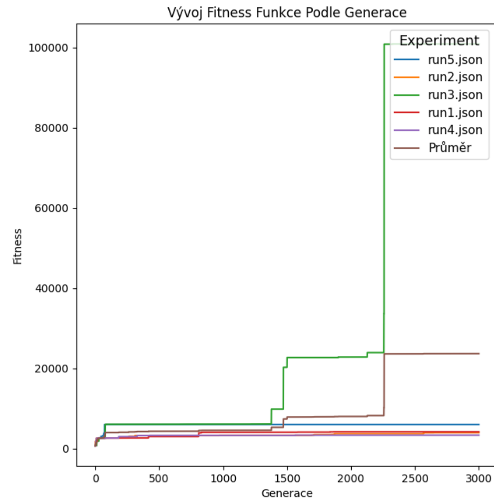
Z dat je jasně vidět, že CNE si vede lépe, než algoritmus NEAT. Průměrná hodnota CNE je okolo 70 000, oproti algoritmu NEAT, který má pouze okolo 30 000.

Z pohledu konvergence je taky úspěšnější CNE, který dosáhl požadované hodnoty před iterací 750 (pokud zanedbáme neúspěšné běhy). Ze všech běhů se algoritmu NEAT povedlo dokonvergovat pouze jednou, u CNE to je ve čtyřech případech.

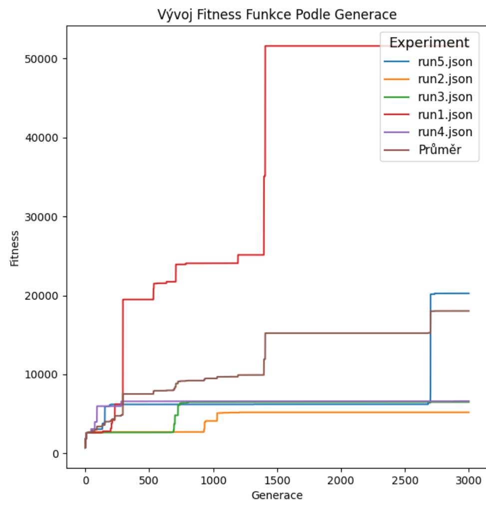
Problém algoritmu NEAT může být nedokonalá implementace. Jeden problém je absence druhů, která napomáhají zachování inovace v architektuře. Bez tohoto principu se hůře vyvijejí složitější architektury. Druhý problém je balanc mutačních operací. Aktuálně se nejčastěji provádí operace mutace váhy synapse (84 ku 10). S menší pravděpodobností se provádí přidání nové synapse (14 ku 100) a s nejmenší pravděpodobností se provádí oprace přidání nového neuronu (2 ku 100). Proto by lepší implementace algoritmu NEAT mohla zvýšit jeho efektivitu a rychlost konvergence k optimálnímu řešení.



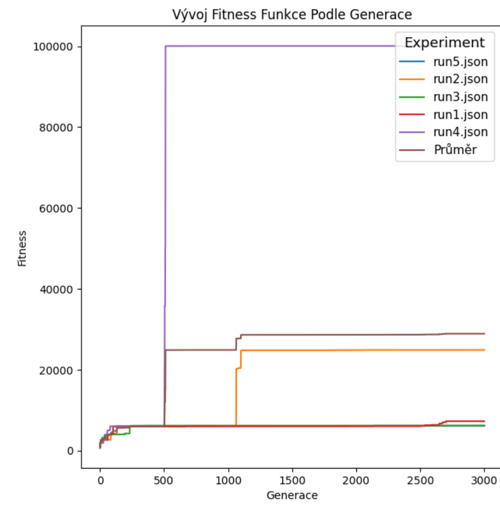
(a) Experiment 4 - míra mutace 0.10



(b) Experiment 4 - míra mutace 0.20

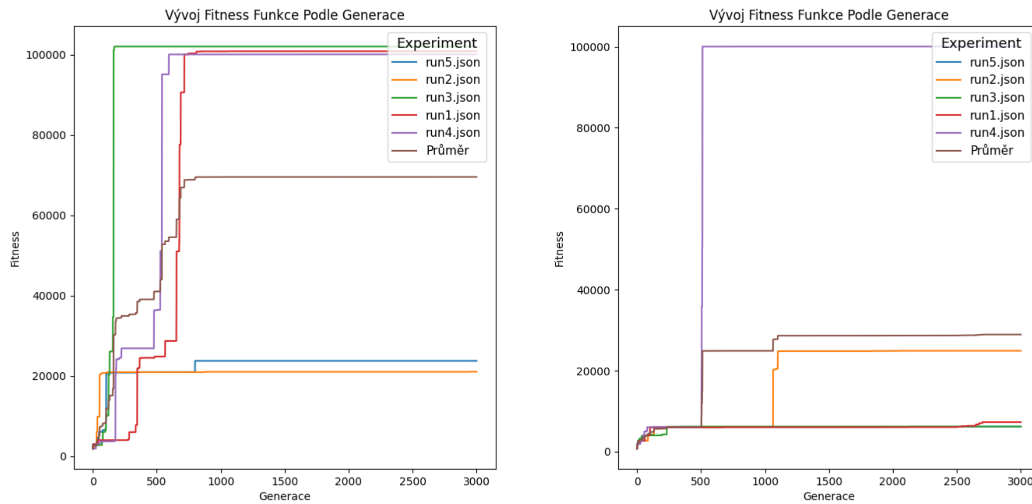


(c) Experiment 4 - míra mutace 0.30



(d) Experiment 4 - míra mutace 0.40

Obrázek 5.3 Experiment 4



(a) Experiment 5 - CNE

(b) Experiment 5 - NEAT

Obrázek 5.4 Experiment 5

Druhá možnost, proč si algoritmus NEAT nevede tak dobře jako CNE je, protože benchmarkové úlohy jsou až moc primitivní. A k optimálnímů řešení stačí menší počet neuronů, která má CNE k dispozici už od začátku. Algoritmus NEAT se k větší architektuře sítě musí postupně propracovat a tudíž straví víc času optimalizací sítě a optimalizace vah, oproti CNE, kterému stačí pouze správně optimalizovat váhy.

Z těchto dvou poznatků by do budoucna bylo dobré provést benchmarkové testy na složitější úloze a zároveň se pokusit vylepšit implementaci algoritmu NEAT.

Kapitola 6
Závěr

Moje bakalářská práce se věnovala vývoji knihovny, která pomocí moderních webových technologií umožňuje rychlý a efektivní vývoj a běh neuroevolučních algoritmů v prostředí webového prohlížeče.

Samotná knihovna se skládá ze tří částí. Nejdříve jsem implementoval jádro, které dělá veškerou koordinaci. Jádro jsem implementoval v jazyce Typescript a pro běh neuroevoluce jsem využil paralelizaci výpočtu pomocí technologie Web Workers. Jádro také poskytuje několik nástrojů, jako je například zaznamenávání výsledků, export/import běhu algoritmu. Část, která se stará o implementační rozhraní neuroevolučních algoritmů je napsaná v C++, což dává jistou flexibilitu a lepší přepoužitelnost. Zároveň je pro uživatelé knihovny připravené rozhraní, které by jim mělo usnadnit práci a vývoj svých vlastních algoritmů. Díky technologii WebAssembly jsem byl schopný C++ kód spustit v prostředí webového prohlížeče. Poslední část, kterou jsem implementoval v rámci knihovny, je jednoduchý engine, pro implementaci vlastního prostředí, na kterém se následně algoritmus bude učit. Díky tomu se vývojář nějakého prostředí nebo algoritmu může soustředit pouze na modelování a nezaobírat se specifičností webového prostředí. Zároveň veškeré části jsou modulární a vývojář, který bude knihovnu používat, si může vybrat, jaké části chce používat. Celý projekt je připravený v prostředí docker, které umožňuje jednoduché spuštění projektu a vývoje bez nutnosti instalace nebo složité použití veškerých nástrojů.

Pro testování knihovny jsem implementoval dvě prostředí pomocí připraveného engine. První je jednoduchá úloha pro otestování funkčnosti konceptu a druhá je složitější a bude tesetovat efektivnosti implementace a algoritmů. Zároveň jsem implementoval dva neuroevoluční algoritmy. Konvenční neuroevoluci a NEAT. Ty jsem implementoval pomocí připraveného rozhraní, abych ukázal univerzálnost knihovny. Poté jsem udělal evaluaci a porovnání všech výsledků, kde jsem vyhodnotil jestli knihovna funguje a zároveň porovnal implementaci konvenční neuroevoluce a NEAT. Na výsledků jsem usoudil, že knihovna je funkční a efektivně využívá všech technologií, které knihovna obsahuje.

Primárním cílem knihovny je zvýšit dostupnost a rozšíření neuroevoluce. Běh v prohlížeči zajišťuje, že nástroj je univerzální a dostupný. Díky jednoduchosti Javascriptu a připravenému rozhraní v C++ je možné snadno experimentovat s vlastními implementacemi. Akcelerace moderními technologiemi zlepšuje dostupnost a eliminuje potřebu silného stroje, čímž ještě více zvyšuje dostupnost. V budoucnosti se například nabízí možnost spolupráce mezi uživateli pomocí sdílených výpočtů nebo kompetitivní soutěže o tvorbu nejlepšího algoritmu. Využití všech aspektů knihovny v praxi a implementace aplikace založené na mé knihovně nejsou součástí mé bakalářské práce, ale možnosti jsou obrovské.

Knihovna je tedy pouze funkční prototyp, který by se dál mohla zlepšovat, aby se jako celek nebo jako dílčí části mohla aplikačně používat. V knihovně by se mohl zlepšit ekosystém díky rozdělení na více samostatných balíčků. Zároveň by se dalo zlepšit API, nebo více modularizovat

dílčí části tak, aby uživatel mohl využít nebo vyměnit pouze určité části. Po technologické stránce by se knihovna mohla rozšiřovat díky implementaci dalších zajímavých konceptů, které se do mojí bakalářské práce nevešli, jako je využití hardwarové akcelerace pomocí webGL, sofistikovanější implementace neuronových sití pomocí balíčků jako je google/XNNPACK, nebo lepší nástroje pro lepší ladění a odchytávání chyb.

Bibliografie

1. HASTIE, Trevor; TIBSHIRANI, Robert; FRIEDMAN, Jerome. Overview of Supervised Learning. In: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York, NY: Springer New York, 2009, s. 9–41. ISBN 978-0-387-84858-7. Dostupné z DOI: 10.1007/978-0-387-84858-7_2.
2. SUTTON, Richard S.; BARTO, Andrew G. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN 0262039249.
3. DONGARE, AD; KHARDE, RR; KACHARE, Amit D et al. Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*. 2012, roč. 2, č. 1, s. 189–194.
4. HORNIK, Kurt; STINCHCOMBE, Maxwell; WHITE, Halbert. Multilayer feedforward networks are universal approximators. *Neural Networks*. 1989, roč. 2, č. 5, s. 359–366. ISSN 0893-6080. Dostupné z DOI: https://doi.org/10.1016/0893-6080(89)90020-8.
5. TAMILSELVI, S. Introduction to Evolutionary Algorithms. In: VENTURA, Sebastián; LUNA, José María; MOYANO, José María (ed.). *Genetic Algorithms*. Rijeka: IntechOpen, 2022, kap. 1. Dostupné z DOI: 10.5772/intechopen.104198.
6. SALIMANS, Tim; HO, Jonathan; CHEN, Xi; SIDOR, Szymon; SUTSKEVER, Ilya. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. 2017. Dostupné z arXiv: 1703.03864 [stat.ML].
7. MIIKKULAINEN, Risto. Neuroevolution. In: *Encyclopedia of Machine Learning*. Ed. SAMMUT, Claude; WEBB, Geoffrey I. Boston, MA: Springer US, 2010, s. 716–720. ISBN 978-0-387-30164-8. Dostupné z DOI: 10.1007/978-0-387-30164-8_589.
8. STANLEY, Kenneth O.; MIIKKULAINEN, Risto. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*. 2002, roč. 10, č. 2, s. 99–127. Dostupné také z: http://nn.cs.utexas.edu/?stanley:ec02.
9. MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; GRAVES, Alex; ANTONOGLOU, Ioannis; WIERSTRA, Daan; RIEDMILLER, Martin. *Playing Atari with Deep Reinforcement Learning*. 2013. Dostupné z arXiv: 1312.5602 [cs.LG].
10. STANLEY, Kenneth O.; D'AMBROSIO, David B.; GAUCI, Jason. A hypercube-based encoding for evolving large-scale neural networks. *Artif. Life*. 2009, roč. 15, č. 2, s. 185–212. ISSN 1064-5462. Dostupné z DOI: 10.1162/artl.2009.15.2.15202.
11. BÄCK, Thomas. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. USA: Oxford University Press, Inc., 1996. ISBN 0195099710.

12. REDDY, Martin. *API Design for C++*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 9780123850034.
13. MOZDEVNET. *JavaScript: MDN*. [B.r.]. Dostupné také z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. [Accessed 01-05-2024].
14. MOZDEVNET. *TypeScript - MDN web docs glossary: Definitions of web-related terms: MDN*. [B.r.]. Dostupné také z: `https://developer.mozilla.org/en-US/docs/Glossary/TypeScript`. [Accessed 01-05-2024].
15. PEREIRA, Rui; COUTO, Marco; RIBEIRO, Francisco; RUA, Rui; CUNHA, Jácome; FERNANDES, João Paulo; SARAIVA, João. Ranking programming languages by energy efficiency. *Science of Computer Programming*. 2021, roč. 205, s. 102609. ISSN 0167-6423. Dostupné z DOI: `https://doi.org/10.1016/j.scico.2021.102609`.
16. STROUSTRUP, Bjarne. *The C++ programming language*. 3rd. Addison-Wesley, 1997. ISBN 978-0-201-88954-3.
17. MOZDEVNET. *TypeScript - MDN web docs glossary: Definitions of web-related terms: MDN*. [B.r.]. Dostupné také z: `https://developer.mozilla.org/en-US/docs/Glossary/TypeScript`. [Accessed 01-05-2024].
18. DEVELOPERS, Emscripten. *About Emscripten*. [B.r.]. Dostupné také z: `https://emscripten.org/docs/introducing_emscripten/about_emscripten.html`. [Accessed: 2023-05-01].
19. GREEN, Ido. *Web Workers: Multithreaded Programs in JavaScript*. O'Reilly Media, Inc., 2012. ISBN 9781449322090.
20. GREGORY, J. *Game Engine Architecture, Third Edition*. CRC Press, 2018. ISBN 9781351974288. Dostupné také z: `https://books.google.cz/books?id=1g1mDwAAQBAJ`.
21. STACK OVERFLOW. *Stack Overflow Developer Survey 2020* [online]. 2020 [cit. 2021-05-02]. Dostupné z: `https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages`.
22. GOOGLE. *Google/XNNPACK: High-efficiency floating-point neural network inference operators for Mobile, server, and web*. [B.r.]. Dostupné také z: `https://github.com/google/XNNPACK#xnnpack`.

Obsah příloh

```
readme.txt ........................ stručný popis obsahu média
exe ........................ adresář se spustitelnou formou implementace
src
  impl ........................ zdrojové kódy implementace
  thesis ........................ zdrojová forma práce ve formátu LaTeX
text ........................ text práce
  thesis.pdf ........................ text práce ve formátu PDF
```