FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Neuronové sítě a evoluční algoritmy pro videohry

Studijní program: N0613A140028 Informační technologie
Studijní obor: Inteligentní systémy

Autor práce: **Bc. Matyáš Horký**
Vedoucí práce: Ing. Karel Paleček, Ph.D.
Ústav informačních technologií a elektroniky

Liberec 2023

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání diplomové práce

Neuronové sítě a evoluční algoritmy pro videohry

Jméno a příjmení:	**Bc. Matyáš Horký**
Osobní číslo:	M21000167
Studijní program:	N0613A140028 Informační technologie
Specializace:	Inteligentní systémy
Zadávající katedra:	Ústav informačních technologií a elektroniky
Akademický rok:	2022/2023

Zásady pro vypracování:

1. Seznamte se s problematikou neuronových sítí a jejich využití v evolučních algoritmech.
2. Vyberte a aplikujte vhodný evoluční algoritmus pro umělou inteligenci v závodní hře TORCS.
3. Natrénujte a vyhodnoťte model umělé inteligence pro řízení na vybrané podmnožině závodních okruhů zahrnutých ve hře.
4. Zhodnoťte úspěšnost modelu na neviděných závodních okruzích.
5. Porovnejte dosažené výsledky s řešeními dostupnými online.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	40-50
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] Goodfellow, I., Bengio, Y., Courville, A. Deep learning. MIT Press, 2016.

[2] Evolutionary Algorithms and Neural Networks: Theory and Applications. 1. Anglie: Springer Cham, 2019. ISBN 978-3-319-93025-1.

[3] Hands-On Neuroevolution with Python: Build high-performing artificial neural network architectures using neuroevolution-based algorithms. 1. Anglie: Packt Publishing, 2019. ISBN B082J28SZ7.

Vedoucí práce:	Ing. Karel Paleček, Ph.D. Ústav informačních technologií a elektroniky
Datum zadání práce:	24. října 2022
Předpokládaný termín odevzdání:	22. května 2023

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 24. října 2022

Prohlášení

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

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

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

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

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

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

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

22. května 2023 Bc. Matyáš Horký

PODĚKOVÁNÍ

Rád bych poděkoval panu Ing. Karlu Palečkovi, Ph.D. za vedení práce a také za cenné rady a připomínky ohledně směřování diplomové práce. Poděkovat bych také chtěl i své rodině za neochvějnou podporu po dobu celého studia.

NEURONOVÉ SÍTĚ A EVOLUČNÍ ALGORITMY PRO VIDEOHRY

ABSTRAKT

V této diplomové práci se zabývám využitím evolučních algoritmů pro trénování umělé inteligence v simulovaném závodním prostředí. K natrénování modelu jsem použil algoritmus NEAT, který má několik zásadních výhod před standardními moderními řešeními. Jako simulované prostředí využívám TORCS, což je open-source závodní simulátor s podporou pro trénování neuronových sítí. Vytvořil jsem skript, který s prostředím komunikuje pomocí protokolu TCP a aplikuje algoritmus NEAT pro trénink modelu. Vyhotovil jsem experimentální analýzu, kde zjišťuji optimální konfiguraci pro trénování na omezené množině závodních tratí s důrazem na schopnost se adaptovat na neznámé tratě. Natrénoval jsem dva modely na základě nejlepší konfigurace podle analýzy a porovnal výsledky s ostatními pracemi. První model je specializován na spolehlivost a druhý na rychlost. Natrénované modely na většině tratí konkurují modelům dostupných přímo v simulátoru TORCS a dosahují nadprůměrných výsledků mezi studentskými pracemi zabývajícími se stejným tématem.

Klíčová slova: NEAT, TORCS, umělá inteligence, strojové učení, neuronové sítě, evoluční algoritmy, videohry

NEURAL NETWORKS AND EVOLUTIONARY ALGORITHMS FOR VIDEO GAME

ABSTRACT

In this thesis, I am dealing with creating artificial intelligence in a simulated environment using genetic algorithms. I chose the NEAT algorithm, which has several significant advantages over regular solutions when dealing with training an artificial intelligence model. I am using TORCS for simulating the environment, which is an open-source racing simulator supporting the training of neural networks. I created a Python script for communication with TORCS and to manage the process of training using the NEAT algorithm. I tested and analyzed multiple configurations, seeking the optimal configuration for training on a limited number of learning tracks while striving to achieve the best results on tracks that the model didn't train on. I trained two the final models according to the best--found configuration and compared them with other models used in TORCS and in other theses. The results obtained are on par with TORCS models and perform above average when compared with other theses.

Keywords: NEAT, TORCS, artificial intelligence, machine learning, neural networks, genetic algorithms, video games

OBSAH

SEZNAM TABULEK

SEZNAM ZKRATEK

TORCS	The Open Racing Car Simulator
NEAT	Neaural Evolution of Augmented Topologies
NN	Neaural Network
AI	Artificial Inteligence
ML	Machine Learning
TWEANN	Topology and Weight Evolving Artificial Neural Networks
CPPN	Compositional Patern Producing Network
RTNEAT	Real-Time NEAT
NPC	Non Playing Character
RPG	Role Playing Games
MOBA	Multiplayer Online Battle Arena

1 Úvod

Svět technologie je nyní ponořen do éry umělé inteligence (AI) a strojového učení. Ze všech stran se nás snaží pohltit fascinující přísliby téměř neomezených možností, jež tyto technologie nabízejí. Ačkoli se těmto oblastem věnuje obrovské množství výzkumu, stále existují oblasti, které jsou teprve na prahu svého objevu. Jednou z těchto oblastí je použití AI v kontextu simulovaných automobilových závodů. Ve své diplomové práci zkoumám tuto oblast, konkrétně v prostředí hry The Open Racing Car Simulator (TORCS).

TORCS je unikátní v tom, že je otevřený, flexibilní a poskytuje realistické prostředí pro simulaci automobilových závodů. Díky těmto vlastnostem představuje ideální platformu pro vývoj a testování AI řidičských strategií. Tato platforma tedy přímo vyzývá k tvorbě AI, které dokáže navigovat po složitých tratích, reagovat na nečekané situace a soupeřit s ostatními řidiči.

V rámci této studie se soustředím na aplikaci algoritmu Neuro-Evolution of Augmenting Topologies (NEAT), metody, která je sice málo využívaná, ale nabízí obrovský potenciál pro vývoj a učení neuronových sítí v závodním prostředí. Klíčovou výhodou NEAT je, že nepotřebuje předem stanovenou strukturu sítě, ale naopak poskytuje síti prostor k vlastní evoluci a adaptaci struktury pro řešení konkrétního problému.

Cílem této práce je zjistit, jak efektivně může být NEAT aplikován v kontextu AI pro hru TORCS, navrhnout a implementovat systém, který využívá NEAT k tréninku AI, a vyhodnotit jeho výkonnost a efektivitu. Tato práce obsahuje zajímavé poznatky a informace získané během trénování a může se stát stavebním blokem pro další pokrok v této fascinující oblasti.

2 REŠERŠE

Umělá inteligence (AI) představuje klíčovou součást široké škály aplikací, včetně videoher, rozpoznávání obrazu a řeči, vyhledávačů a mnoha dalších. Tento termín se vztahuje na systémy, které napodobují lidskou inteligenci a mohou být chápány jako ”myšlenková síla vytvořená člověkem”. Na rozdíl od předem naprogramovaných systémů se umělá inteligence opírá o algoritmy strojového učení, jako jsou například NEAT, posilované učení (reinforcement learning) nebo hluboké učení (deep learning).

Tyto algoritmy umožňují AI systémům provádět odhady a předpovědi založené na historických datech. Aby tyto systémy fungovaly správně a dosahovaly přesných výsledků, je nezbytné poskytnout jim velké množství trénovacích dat, ze kterých mohou vytvořit efektivní rozhodovací modely.

Algoritmy strojového učení jsou schopné učit se z historických dat a adaptovat se na základě těchto informací. Jejich účinnost závisí na dostupnosti trénovacích dat a jsou obvykle nejúčinnější v řešení specializovaných úloh, jako je například rozpoznávání obrazů z omezeného počtu možností.[1]

2.1 AI VE VIDEOHRÁCH

V dnešní době se umělá inteligence stala běžnou součástí většiny videoher. Její prezentace se může lišit - od interakcí s nehratelnými postavami (NPC) po počítačem řízené hratelné postavy. Přístup k designu a implementaci AI se liší v závislosti na jeho účelu a složitosti. V případě dialogů s NPC se často používá jednoduchý model rozhodovacího stromu. Avšak tento přístup se ukazuje jako nedostatečný pro vývoj složitějších forem AI a je obvykle pouze jednou z mnoha součástí komplexního řešení AI.

AI dosahuje optimálních výsledků při rozhodování v uzavřeném systému situací, kde všechny možné scénáře jsou známy předem. Typickým příkladem je šach, kde má každá figura omezený počet možných tahů, pravidla jsou pevně stanovena a herní strategie je dobře definována. Pro ideální využití AI ve hře šachů je postačující robustní stavový automat, který může zahrnout všechny možné scénáře. Nicméně existují situace, kdy takový strojový automat nemůže poskytnout optimální řešení. Stavové automaty se nehodí pro případy s proměnlivými faktory, které nejsou předem známé. Například, pokud bychom chtěli navrhnout AI pro řízení vozidla na neznámé trati, stavový automat by vyžadoval pokrytí všech možných scénářů v editoru tratí, což

je většinou neproveditelné. Alternativou by bylo vytvořit aproximaci a doufat, že nevznikne situace, kterou jsme nepředvídali.

Pro řešení takovýchto úloh s proměnlivými faktory se často využívají neuronové sítě. Tyto sítě nabízejí aproximaci řešení, která se však nezakládá na analytickém přístupu, ale na metodě pokus-omyl řízené specifickou metodologií. Na rozdíl od rigidních stavových automatů, neuronové sítě nemohou vždy poskytnout nejlepší řešení. Výhoda neuronových sítí spočívá v jejich schopnosti učit se a adaptovat se na základě zkušeností. Tento adaptivní proces učení je možný díky velkému množství trénovacích dat, která jsou použita pro „trénink“ sítě. Čím větší je množství a kvalita těchto dat, tím lépe je schopna síť přizpůsobit své rozhodovací procesy a dosáhnout výsledků blízkých ideálnímu řešení. [2]

Historie

Jedním z prvních využití AI pro hraní videoher bylo v roce 1951 do matematické strategické hry zvané Nim. Ve hře se střídají dva hráči a odebírají objekty z balíčků. Obvykle hra skončila vítězstvím AI.[19]

Kolem roku 1990 se častěji začaly objevovat modernější metody pro řízení umělé inteligence jako jsou například koncové stavové automaty.[19]

První závodní hra s počítačem řízenými protivníky byla v roce 1982 hra Pole Position od japonské společnosti Namco. Hráč zde řídil formuli na okruhu s protihráči v pseudo 3D grafice. Další závodní hry, které využívaly prvky AI byly například Super Mario Kart z roku 1992 nebo Driver z roku 1998.[2]

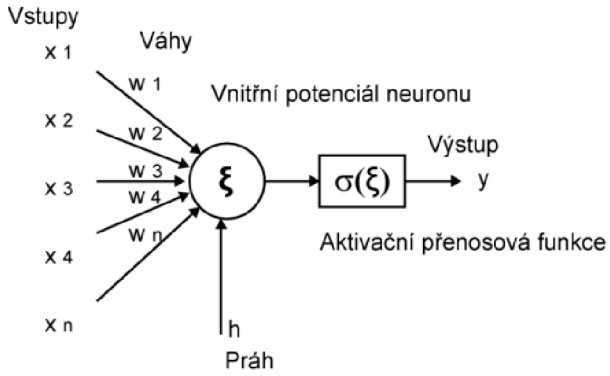
AI na počátku řešila dva hlavní problémy. Ovládání vozidla a hledání nejlepší trasy. V klasických závodních hrách se problém optimální trasy řešil pomocí závodních stop (Racing lines). Každá trasa měla několik předdefinovaných stop, které AI napodobovalo. Hlavní výhodou těchto stop byla jednoduchost a rychlost implementace, avšak pohyb po stopách nevypadal příliš realisticky.[2]

V závodních hrách s otevřeným světem nebylo možné využít závodních stop a tedy bylo nutné najít alternativu. Tady se umělá inteligence závodních her inspirovala strategickými hrami a využívala podobné metody pro hledání optimální cesty.[2]

2.2 NEURONOVÉ SÍTĚ

Neuronová síť je výpočetní model, jehož koncept je inspirován strukturou a funkcí neuronů a nervových sítí živých organismů. Neuronová síť se skládá z jednotlivých neuronů propojených navzájem synaptickými váhami. Každá neuronová síť obsahuje vstupní vrstvu, která zpracovává vstupní data, několik skrytých vrstev, které jsou zodpovědné za rozhodovací proces, a výstupní vrstvu, která generuje výsledky modelu.

V případě, že neuronová síť obsahuje pouze jednu skrytou vrstvu, je tato vrstva nazývána vrstva perceptronů. Perceptron je základní prvek umělých neuronových sítí, který generuje výstup na základě dvou logických úrovní (0 nebo 1). Hlavním omezením perceptronu je jeho schopnost učit se pouze lineárně separabilní problémy, což znamená, že není schopen řešit složitější a nelineární problémy.[3]



Obrázek 2.1: Model neuronu

Pro řešení komplexnějších úloh pomocí neuronových sítí je třeba přidat více skrytých vrstev neuronů za sebou tak, že výstup z předchozí vrstvy je připojen jako vstup do následující vrstvy. Těmto propojením mezi dvěma neurony říkáme vázaná synaptická spojení. Každé spojení mezi neurony má svou váhu, dynamický koeficient, který je závislý na důležitosti informace ve spojení pro řešení problému.

Na obrázku 2.1 lze vidět model neuronu. Vstupy x_1 až x_n násobené jejich váhovými koeficienty w_1 až w_n se slučují pomocí agregační funkce ξ. Pokud hodnota přesáhne minimální hodnotu prahu h, neuron se stane aktivním a výstup agregační funkce vloží do bloku aktivační funkce, která následně určuje výstup neuronu.

Agregační funkce neuronu udává způsob, jakým se zpracovávají vstupní hodnoty. Nejčastěji se jedná o některou z matematických funkcí suma, produkt, minimum, maximum a průměr. Pro případ zvolení agregační funkce suma lze výstup agregační funkce zapsat matematicky jako:

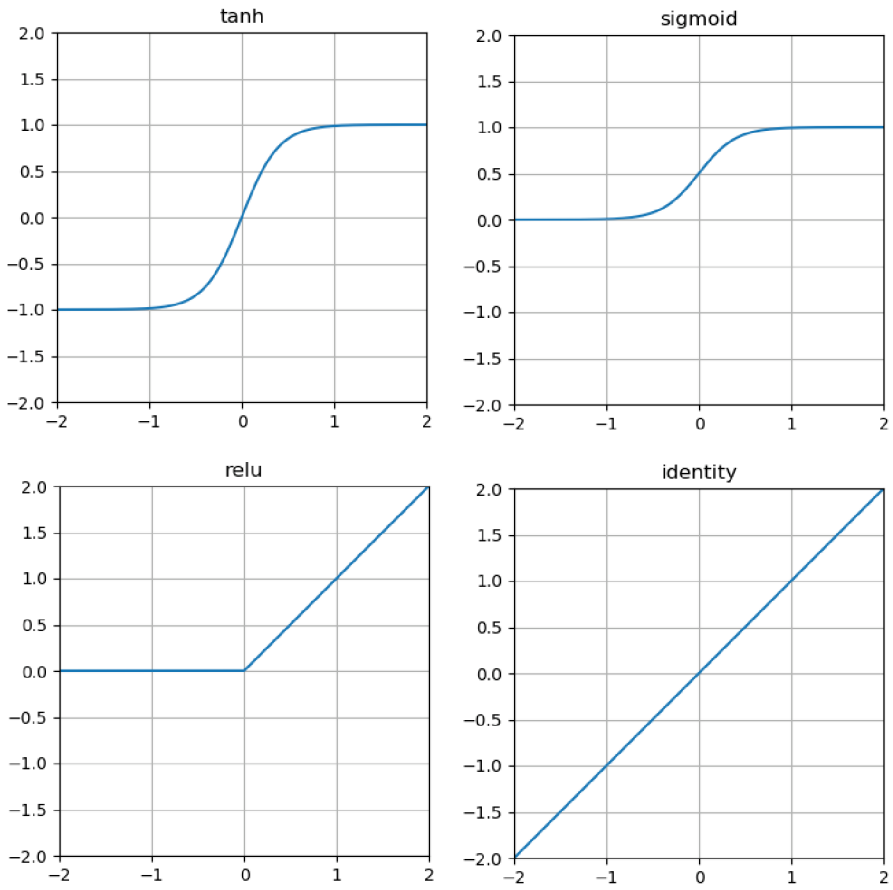
$$\xi = \sum_{i=1}^{n} x_i w_i - h$$

kde x_i představuje vstupní hodnotu, w_i představuje váhový koeficient a n je počet vstupních hodnot.

Bias (posunutí) neuronu h označuje minimální hodnotu pro aktivaci neuronu. Často se bias modeluje jako vstup s pevnou hodnotou 1 a proměnnou hodnotou váhového koeficientu. Tento způsob zajišťuje, že aktivace neuronu se trénuje spolu s neuronovou sítí.[4]

Aktivační funkce určuje finální úpravu výstupu neuronu a ovlivňuje jeho aktivaci. Jejím úkolem je zohlednit nejčastější správné výstupy a přizpůsobit je tak, aby měly nejvyšší přítomnost na výstupu. Například funkce hyperbolického tangensu (tanh) je téměř lineární pro hodnoty v intervalu 0 až 0,5, zatímco hodnoty větší než 0,5 jsou značně zmenšené. Tímto způsobem funkce redukuje četnost výstupů s vysokou hodnotou. Aktivační funkce zároveň udržuje hodnotu výstupu v povoleném intervalu, nejčastěji mezi hodnotami 0 a 1 nebo -1 a 1.

Příklady běžně používaných aktivačních funkcí zahrnují sigmoidní funkci, ReLU (Rectified Linear Unit) a softmax. Výběr aktivační funkce závisí na konkrétních potřebách a cílech daného neuronového modelu.



Obrázek 2.2: grafy nejčastějších aktivačních funkcí

Existuje mnoho typů neuronových sítí, které se liší svou architekturou, principy učení a účelem použití. Některé z nejběžnějších typů neuronových sítí zahrnují:

Jednovrstvé a vícevrstvé perceptronové sítě

Lineární model jednovrstvého perceptronu představuje jeden z nejjednodušších modelů neuronové sítě. Skládá se z vstupní vrstvy, vrstvy perceptronů a výstupní vrstvy. Perceptron umožňuje separovat data do dvou tříd klasifikace pomocí lineárního rozhodovacího prahu.

Vícevrstvá perceptronová síť (MLP) je značně složitější než jednovrstvý perceptron. Obsahuje více vrstev perceptronů, kde veškeré neurony jsou propojeny vazbou, což znamená, že se jedná o plně propojenou síť. MLP se využívají pro složitější úlohy, jako je rozpoznávání řeči nebo složitější klasifikaci dat. Jelikož je model poměrně komplexní, je také náročný na údržbu a výpočetní zdroje při trénování. Pro efektivní trénování MLP se často používá algoritmus zpětného šíření chyby (backpropagation), který minimalizuje chybovou funkci a optimalizuje váhy mezi vrstvami. [5]

Dopředné neuronové sítě

V dopředných neuronových sítích jsou veškeré vazby mezi neurony orientovány ve směru od vstupu k výstupu. Nepřítomnost zpětné vazby znamená, že výstup závisí pouze na současných vstupních hodnotách. Ačkoli dopředné neuronové sítě mohou obsahovat více skrytých vrstev, patří mezi jednodušší typy neuronových sítí.

Dopředné neuronové sítě vykazují lepší výsledky než některé základní algoritmy strojového učení. Nejčastěji se používají modely s jednou nebo dvěma skrytými vrstvami, ty avšak nedosahují úrovně výkonnosti hlubokých neuronových sítí. Hlubokou neuronovou sítí se označuje model s více než dvěma skrytými vrstvami. Pro tyto modely se používají sofistikovanější techniky pro zpracování dat, například konvoluční nebo rekurentní vrstvy. Hluboké neuronové sítě se ukázaly být účinnější při řešení složitějších úloh, jako je rozpoznávání obrazu, zpracování přirozeného jazyka nebo hraní her, avšak jejich trénink vyžaduje mnohem větší množství dat.[5]

Rekurentní neuronové sítě

Rekurentní neuronové sítě (RNN) představují velmi silný nástroj pro zpracování sekvencí dat. V jejich modelu se objevují zpětné vazby, které spojují výstupy neuronů s jejich vstupy, což vytváří smyčky. V rekurentních neuronových sítích se k vstupním hodnotám přidávají také hodnoty minulých výstupů, což umožňuje modelu kombinovat data a získat mnohem větší množství unikátních situací než u dopředných neuronových sítí.

Zpětná vazba přidává modelu schopnost paměti, což z něj činí vhodný nástroj pro úlohy, kde je kontext důležitý, jako například rozpoznávání textu. Rekurentní neuronová síť si tak může pamatovat kontext textu a případně upravit výstup podle něj. RNN se úspěšně používají také v oblastech jako je rozpoznávání řeči, strojový překlad nebo generování textu, kde hraje roli sledování časových závislostí mezi jednotlivými prvky v sekvenci.[6]

Přestože neuronové sítě přinášejí mnoho výhod, mají také své nevýhody. Jsou často považovány za ”černé skříně”, což znesnadňuje pochopení a přizpůsobování jejich chování. Vyžadují také velké množství trénovacích dat, což může být náročné na získání a zpracování. Navíc, trénování hlubokých neuronových sítí pomocí označených dat (učení s učitelem) může být problematické v případech, kdy jsou některé typy dat opomenuty, což vede k trvalým chybám v modelu.

Na druhou stranu, učení bez učitele umožňuje snadnější získávání dat, protože lze použít jakákoliv data správného typu. Nicméně, tento způsob učení má nižší přesnost informace, protože model si musí sám setřídit data do skupin, které se mohou lišit od skupin navržených člověkem. Interpretace výsledků modelu může být také náročná.

Přes tyto nevýhody se neuronové sítě stále dále rozvíjejí a přinášejí nové možnosti v oblasti umělé inteligence. Výzkum v této oblasti neustále hledá způsoby, jak zlepšit jejich efektivitu, přesnost a snadno použitelnost, což umožňuje jejich širší uplatnění v různých průmyslových odvětvích a vědeckých oblastech.[5]

2.3 TRÉNOVÁNÍ NEURONOVÝCH SÍTÍ

Neuronové sítě mohou být trénovány různými způsoby, které se liší ve způsobu, jakým jsou použita trénovací data a jak je hodnocena efektivita učení. Mezi tyto metody patří supervised learning, unsupervised learning, reinforced learning a evoluční algoritmy.

Supervised learning

Při využití supervised learning jsou trénovací data označena správnými výsledky, což implikuje, že pro každý vstupní vzor existuje odpovídající výstup. Hlavní úlohou algoritmu je objevit funkci, která nejvíce odpovídá zobrazení vstupních dat na odpovídající výstupy. Tato metoda je často využívána v úlohách klasifikace a regrese.

Při trénování AI ve videohrách se supervised learning obvykle nevyužívá nebo využívá pouze jako součást hlavního trénovacího algoritmu. Supervised learning umožňuje snadné učení, ale shromažďování dostatečného množství označených dat může být náročné. [7]

Unsupervised learning

Na rozdíl od supervised learning, unsupervised learning pracuje s neoznačenými daty, což znamená, že pro vstupní vzory nejsou k dispozici žádné výstupní hodnoty. Úkolem algoritmu je identifikovat strukturu nebo vzory v těchto datech bez předchozí znalosti očekávaných výsledků. Tento přístup se využívá primárně v úlohách shlukování (clustering) a redukce dimenzionality.

Unsupervised learning se obvykle nepoužívá samostatně pro trénování AI ve videohrách. Pro trénování neuronových sítí v prostředí s definovanými pravidly a možností přístupu k libovolným datům je tento přístup neefektivní a výhodnější je využití reinforced learning.

Ačkoliv je pro unsupervised learning jednodušší shromáždit trénovací data, protože nemusí být označována, trénování je obecně méně efektivní.[7]

Reinforced learning

Reinforced learning modeluje chování agenta v daném prostředí a hodnotí akce, které vykoná. Oproti hodnocení správnosti jednotlivých vstupů je tento přístup zaměřen na efektivitu průchodu prostředím. Reinforced learning je vhodný a nejčastěji využívaný druh algoritmu pro trénování neuronových sítí v aplikacích umělé inteligence ve videohrách.

Existuje řada algoritmů, které spadají do kategorie reinforced learning. Jako příklad lze uvést algoritmus Deep Q-Learning, který prostředí rozdělí na několik oblastí, mezi kterými přechází dostupnými akcemi. Každá akce je při svém provedení hodnocena a toto hodnocení je přičteno k celkovému skóre dané instance modelu. Hodnocení může být jak kladné, tak záporné.

Po procházení prostředím je hodnocení modelu zaznamenáno a následuje úprava jak samotného modelu neuronové sítě, tak prostředí, které je modelováno. Tímto způsobem se postupně zlepšuje výkonnost modelu v daném

prostředí.[1]

Evoluční algoritmy

Evoluční algoritmy představují specifický přístup k trénování neuronových sítí, inspirovaný evolučními procesy. Tyto algoritmy generují populaci jedinců, která reprezentuje různé konfigurace neuronové sítě. Vhodní jedinci jsou vybráni na základě hodnocení jejich výkonnosti a pomocí mutací a křížení těchto jedinců se vytvářejí nové možnosti nastavení neuronové sítě.

Evoluční algoritmy se v mnoha ohledech podobají reinforced learning, i když hodnocení a úpravy modelu probíhají odlišně. Oproti reinforced learning, evoluční algoritmy jsou založené na statistickém přístupu. Stejně jako v reinforced learning, model prochází prostředím a shromažďuje skóre, avšak nikoliv na základě individuálních akcí modelu, ale na základě konečného stavu. Nahromaděné skóre je zaznamenáno a po ukončení generace se na základě skóre provádí křížení a generuje se nová populace.

Pro trénování AI do hry TORCS je nejvhodnější reinforced learning a evoluční algoritmy. Evoluční algoritmy jsou v oblasti videoher méně prozkoumané, ale mají výhodu v tom, že nepotřebují mapovat prostředí, což je pro hru TORCS komplikace. Z tohoto důvodu jsem se rozhodl pro řešení této úlohy pomocí evolučních algoritmů. [9]

Příklady evolučních algoritmů:

- NEAT
- diferenciální evoluce
- světluškový algoritmus
- algoritmus houfu částic
- algoritmus umělých včel
- optimalizace pomocí mravenčí kolonie

2.4 NEAT

NEAT neboli Neural Evolution of Augmenting Topologies je genetický algoritmus pro trénování neuronových sítí. Alternuje strukturu a váhy neuronové sítě tak, aby dávala požadované výsledky. Jeho největší předností je schopnost minimalizovat rozměry vyhledávacího prostoru minimalizací a postupným rozvojem topologie neuronové sítě. Nejčastěji se používá pro trénování bez učitele, ale jeho vlastnosti jej umožňují použít i pro trénování s učitelem. NEAT se řadí do skupiny algoritmu TWEANN, Topology and Weight Evolving Artificial Neural Networks. Tato skupina algoritmů upravuje strukturu neuronové sítě.

Historie a vývoj

NEAT (NeuroEvolution of Augmenting Topologies) byl vytvořen Kenem Stanleym a Risto Miikulainenem v roce 2002 na Texaské univerzitě v Austinu. Před vynálezem NEATu pracovala tradiční evoluce neuronových sítí s neměnnou topologií a pevnými parametry. To výrazně omezovalo možnosti použití neuronových sítí a komplexnost úloh, které byly schopny řešit. NEAT tyto omezení překonal evolucí topologie neuronové sítě. Průlomovou inovací byl způsob, jakým se NEAT stará o křížení neuronových sítí. Ve starších algoritmech se křížení řešilo prohazováním sad parametrů mezi dvěma jedinci. NEAT míchá geny dvou rodičů a vytváří tak potomka.[14]

I po roce 2002 NEAT nadále procházel vývojem. V roce 2007 představil Stanley rozšíření algoritmu nazvané HyperNEAT. Toto rozšíření používá Compositional Pattern Producing Networks (CPPN) pro generování topologií neuronových sítí. Na rozdíl od klasického NEATu je HyperNEAT zaměřen na evoluci velkých neuronových sítí. Používá nepřímé hyperbolické kódování, což umožňuje generování topologií, které jsou oproti klasickému NEATu lépe organizované, například symetrické. [8,10]

V roce 2008 bylo představeno další rozšíření nazvané RTNEAT. Toto rozšíření bylo navrženo tak, aby umožňovalo evoluci neuronových sítí v reálném čase. To přináší hlavně výhody v uplatnění v úlohách z reálného prostředí a pro ovládání robotů. RTNEAT používá modulární přístup, kde jednotlivé moduly neuronové sítě jsou vyvíjeny nezávisle.[11]

Další rozšíření, ES-HyperNEAT, bylo představeno v roce 2012 a rozšiřuje HyperNEAT. V klasickém HyperNEAT algoritmu musel programátor zvolit umístění a počet skrytých neuronů. ES-HyperNEAT toto dokáže určit sám a zároveň splňuje požadavky HyperNEATu.[12]

CoDeepNEAT je další rozšíření představené v roce 2018. Jedná se o algoritmus, kde evoluce probíhá ve vrstvách, který se snaží využít repetitivní struktury v doméně problému. Toho dosahuje rozdělením genomů na dva elementy, ze kterých je možné genom opět zpětně zformovat, ale každý element má vlastní populaci podléhající neuro-evoluci. Těmto elementům se říká moduly. Modul je malá hluboká neuronová síť s předdefinovanou složi-

tostí a konfigurací, která má za úkol reprezentovat jednu instanci repetetivní struktury v doméně problému. Tyto moduly jsou spojeny v souhrnný graf, kterému se říká výkres.

CoDeepNEAT je efektivní pro řešení problémů v repetetivní doméně a má dobré výsledky v oblastech rozpoznávání obrazu, jazykových modelů a dokonce pro generování digitálního umění.

Vývoj NEAT a jeho rozšíření ukazuje, jakým způsobem se evoluční algoritmy mohou vyvíjet a adaptovat na nové problémy a domény. Od svého vzniku v roce 2002 se NEAT a jeho odvozené algoritmy staly základními stavebními kameny pro mnoho aplikací v oblasti umělé inteligence a strojového učení. Díky jejich schopnosti evoluce topologie a parametrů neuronových sítí se stávají flexibilními a účinnými nástroji pro řešení široké škály problémů.[13]

Výhody

- Dynamická topologie
- Zachovává inovace
- Rozdělení populace na druhy
- Kontrola složitosti NN
- Inkrementální evoluce

Nevýhody

- Složitost
- Časová a výpočetní náročnost
- Předpoklad přesné fitness funkce
- Velké množství parametrů, které se musí správně nastavit
- Nevhodný pro příliš jednoduché nebo příliš složité problémy

Pro evoluci topologie neuronové sítě NEAT využívá tři principy. Historické značení, rozdělení do druhů a postupné rozvíjení. [14]

Historické značení

Tento princip umožňuje ukládat informaci o předchozích generacích a vycházet z jimi dosažených výsledků. Jednou z nejdůležitějších informací každého genomu je jeho genová struktura. Může se stát že některé dva chromozomy obsahují stejné geny na různých částech, což může způsobit nekompatibilitu v křížení, jelikož by mohlo dojít ke ztrátě genové informace. NEAT popisuje jednotlivé geny pomocí historického značení unikátním číslem, čímž zamezuje ztrátě informace. Dva geny z odlišných struktur mají stejný původ pokud se schoduje jejich historické značení. Tento princip umožňuje lépe křížit chromozomy v neuronové síti a sledovat postup vývoje. [14]

Rozdělení do druhů

Obvykle po přidání struktury do neuronové sítě, dočasně klesne její hodnocení. Jelikož toto hodnocení je kritické pro výběr přeživších jedinců, nastává riziko okamžitého zahození inovativních struktur. NEAT tento problém řeší rozdělením jedinců do druhů rozlišených podle struktur a váhových rozdílů. Každý jedinec je nejdříve porovnáván ve svém druhu, což dává větší šanci na přežití novým strukturám a mohou být lépe optimalizovány před porovnáváním s veškerou populací.[14]

Postupné rozvíjení

NEAT využívá postupné rozvíjení pro minimalizaci neuronové sítě. Začíná v minimální struktuře, vstupní a výstupní vrstva plně propojené. Během procesu evoluce NEAT postupně prohledává prostory s menším počtem vazeb a následně prostory s větším počtem vazeb. Toto zvyšuje efektivitu a minimalizaci výsledné struktury neuronové sítě.[14]

Kroky a logika algoritmu

NEAT můžeme rozložit do několika následujících kroků:

- Inicializace
- Hodnocení
- Rozdělení do druhů
- Reprodukce
- Selekce
- Evoluce

Prvním krokem je inicializace neuronové sítě. Jak bylo zmíněno v principu postupného rozvíjení, NEAT začíná s neuronovou sítí s minimální strukturou, ovšem populace zpravidla nebývá omezena na jeden model neuronové sítě, tudíž je nutné vygenerovat několik takových sítí s pouze malými náhodnými změnami. Každá síť je reprezentována orientovaným acyklickým grafem, kde uzly reprezentují neurony a hrany reprezentují vazby mezi nimi. Důvodem, proč se používá struktura orientovaného acyklického grafu je, zamezení tvorbě smyček. Tato vlastnost je výhodná pro klasické dopředné neuronové sítě, avšak zamezuje použití rekurentních neuronových sítí, pro jejich použití je nutné algoritmus upravit. Pro použití rekurentních neuronových sítí se místo orientovaného acyklického grafu používá struktura HyperNEAT substrát. Jedná se o vrstvenou mřížku uzlů, která umožňuje reprezentovat dočasnou strukturu rekurentní neuronové sítě. Substrát funguje jako šablona, do které se časem přidávají vazby mezi neurony. [15][16]

Následujícím krokem je hodnocení jedinců o což se stará fitness funkce. Jejím úkolem je ohodnotit, jak dobrý výstup dává daný jedinec. Ovšem správnost výstupu není jediné hodnotící kritérium pro jedince. Druhým kritériem hodnocení je komplexnost neuronové sítě. Čím je neuronová síť složitější, tím víc bodů jí strhne fitness funkce. Toto má docílit zachování co nejjednodušších neuronových sítí, které není potřeba trénovat tak dlouho a nejsou předimenzované vůči trénovacím datům. Následně jsou jedinci seřazení podle jejich hodnocení a zachovává se pouze zvolený počet nejlepších jedinců. [16]

Reprodukci lze rozdělit do tří menších kroků a to selekci, křížení a mutaci. Seřazení jedinců podle jejich hodnocení a následné vybrání nejlepších se říká selekce. Počet těchto přeživších definuje programátor. V selekci je důležité zachovat dostatečný počet rozdílných jedinců a zároveň zachovat geny jedinců s nejlepšími výsledky. Kromě klasického řazení se také používají alternativní metody selekce. Další je inspirovaný ruletou, kdy každý jedinec

dostane počet lístků do losování podle jeho hodnocení a výběr lístku je náhodný. Po vylosování jedince pro zachování se z losování odeberou všechny jeho lístky a pokračuje se dál, dokud není vybrán požadováný počet jedinců.[16]

Vybraní jedinci se poté kříží. Náhodně se vyberou páry jedinců, na jejich základu vznikne nový potomek. Cílem křížení je výměna genetické informace v potomku tak, aby byly zachovány dobré kvality rodičů a odstraněny jejich slabiny. Je několik způsobů křížení. V uniformním křížení se střídavě iteruje přes všechny geny rodičů (1. z jednoho, 1, z druhého, 2. z prvního, ..., poslední z prvního, poslední z druhého). Každý gen má 50% šanci že bude v potomku zachován. Dalším způsobem křížení je jednobodové křížení. Náhodně zvolení rodiče vytvoří dva potomky rozdělené na dvě části. První potomek obsahuje první část z prvního rodiče a druhou část z druhého rodiče. Druhy potomek má části obráceně. Podobným způsobem funguje i dvoubodové křížení, akorát jsou potomci i rodiče rozděleni na tři části. Toto křížení je možné, protože geny jsou označeny unikátním číslem (princip historického mapování) a je tedy možné určit jejich pořadí. To umožňuje potomku podědit geny rodičů a zasochovat jejich kompatibilitu.[16]

Poslední části reprodukce je mutace. Cílem mutace je přivést změny a přidat nové možnosti vývoje a prozkoumat další části vyhledávacího prostoru. Programátor nastavuje šanci mutace pro jednotlivé geny, vazby a váhy. Mezi typické mutace patří změna aktivační funkce, změna váhy a přetržení nebo tvorba vazby. NEAT omezuje počet mutací, tak aby vzniklá neuronová síť nebyla příliš vzdálená od originálu a zachovala se kompatibilita. Samotné neuronové sítě mají tendenci zůstávat v určité části vyhledávacího prostoru, proto NEAT přidává mutace. [16]

Tyto kroky se opakují do dosažení požadovaných výsledků u jedince, většiny jedinců populace nebo po dosažení určitého počtu generací. Toto opakování se nazývá evoluce.[16]

Neuronové sítě ve videohrách

Pro programování AI do videoher jsou neuronové sítě velmi netradiční a spíše experimentální nástroj. První takovéto použití neuronových sítí v závodní videohře přišlo roku 1988 ve hře Neural Racer, ve které soupeře hráče řídila neuronová síť. Oproti ostatním dobovým závodním hrám bylo AI bližší člověku, dokázalo chybovat nebo riskovat. Neuronové sítě této hry byly trénovány metodou zpětné propagace, která se stále běžně používá. Pro trénování byla využita data hráčů z minulých závodů.[17]

Typicky se v dnešní době používají pro tvorbu AI stavové automaty nebo behaviorální stromy. Ty jsou oproti neuronovým sítím mnohem jednodušší na úpravu. Neuronové sítě v AI se používají pro určité úlohy, které by pro stavové automaty nebo behaviorální stromy byly příliš obsáhlé nebo jinak nevhodné. Hlavní nevýhodou těchto technologií je skutečnost, že pro každý případ musí být definovaná odpověď systému. Pokud je problém dosti komplikovaný. Napsat stavový automat, který kryje veškeré možné situace, je téměř nemožné. Pro tyto problémy jsou neuronové sítě vhodné, protože o řešení problému se nestará programátor, ale samotná neuronová síť. [17]

Velkým průlomem v používání neuronových sítí pro AI do videoher je projekt výzkumného střediska OpenAI pod názvem OpenAI five. Projekt trénoval umělou inteligenci s cílem porazit hráče v populární MOBA hře Dota 2. Ve hře Dota 2 hrajete za jednoho z mnoha dostupných hrdinů a snažíte se zničit budovy soupeřů za konstantního přívalu armád NPC z obou stran. Hra podporuje mechaniky RPG, takže za souboje získáváte zkušenosti a peníze, za které je možné vašeho hrdinu vylepšit. Projekt se uskutečnil mezi roky 2016 a 2019 a skončil velkým úspěchem. Zpočátku AI bylo navrženo pouze pro situaci jeden na jednoho, kde naprosto excelovalo. To nebylo až takové překvapení, jelikož tato situace značně zužuje možnosti které můsí AI pokrýt. Profesionálním hráčům hry Dota 2 trvalo přes tisíc her, než se jim podařilo AI porazit. [18]

Opravdovým průlomem byl však standardní herní formát. V normální hře Doty 2 soupeří pět hráčů proti pěti. Klíčovými prvky hry jsou strategie, komunikace, znalost hry a spolupráce mezi hráči. OpenAI five zvládlo většinu z těchto aspektů, jedinou slabinou byla dlouhodobá strategie. Z počátku AI mělo problém myslet dostatečně daleko do hry a to byla jediná šance profíků jak získat vítězství. Tuto slabinu však AI odstranilo dodatečným tréninkem a ke konci projektu už neprohrávalo žádné hry. OpenAI five na konci dosáhlo úspěšnosti na vítězství 99,4% z poměru 7215 výher a 42 proher. Je jasné, že AI získalo značné výhody řízením počítačem jako jsou například zanedbatelný reakční čas, mizivá rozhodovací prodleva, kompletní znalost hry a možnost centrálního řízení, kdy vlastně každý z hrdinů je řízen jako jedna z jednotek nějaké bojové skupiny, ale i po odečtení těchto výhod hra OpenAI five vypadala velmi solidně a mnoho z hráčů by mělo problém držet krok, i kdyby byly nastaveny lidské limitace. [18]

OpenAI five bylo trénováno metodou reinforcement learning. Pro trénová-

ní bylo vytvořeno speciální simulační prostředí reprezentující hru Dota 2. Pro optimalizaci sítě byla použita metoda zvaná policy gradient optimalization. Zpočátku bylo AI trénováno na lidských datech a následně trénovalo v simulovaném prostředí ve hrách, kdy proti sobě soupeřily instance AI. Trénink OpenAI five byl velice náročný a není uvedeno přesně, jak dlouho trénink trval. Bylo uvedeno, že trénink trval několik měsíců v simulovaném prostředí což odpovídá 180 let normálních her. [19]

V moderních závodních hrách mají neuronové sítě unikátní využití. Obvykle se neuronová síť trénuje než dodává dostatečně úspěšný výstup, to ovšem není úplně pravda pro tyto specifické závodní hry. V sérii závodních her Formule 1 je možnost hrát kariéru. Tento herní režim se soustředí na vašeho jednoho řidiče, za kterého se snažíte probojovat až na vrchol. Hra je značně inspirovaná realitou, tudíž se na rozdíl od arkádových her obtížnost nestupňuje s průběhem hry, pouze se zvyšují nároky na umístění. Co dodává režimu kariéry v sérii F1 dodatečný pocit reality je způsob, kterým jsou řešení počítačový oponenti. Hra počítá s faktem, že hráč se během hraní učí ovládat formuli a bude se pravděpodobně zlepšovat. Každý počítačový závodník má vlastní umělou inteligenci, která konstantně snímá data hráče a upravuje neuronovou síť, která řídí počítačového protivníka. To je možné, jelikož neuronová síť je konstantně v režimu trénování a nikdy nedosáhne natrénovaného stavu. U tohoto použití však je nutné zajistit, aby vliv dat hráče nemohl upravit neuronovou síť natolik, aby nebyla schopná formuli dostatečně dobře řídit, je nutné najít kompromis mezi variabilitou a stabilitou. [20]

Další moderní sérií závodních her využívajících neuronových sítí pro extra požitek z hraní je série Forza. Forza je známá série pro svůj poměrně velký důraz na realitu, přesto že by se dala zařadit mezi arkádové hry. Vývojáři Forza přišli s unikátním vylepšením pro multiplayerovou část novějších her ze série. Říkají tomu Drivatar AI, vycházející z kombinace slov driver a avatar. Hráč hraním hry trénuje vlastního avatara řízeného neuronovou sítí. Ten podobně jako u série F1 sbírá data hráče a konstantně trénuje neuronovou síť. V nepřítomnosti hráče pak v multiplayerových závodech jezdí drivatar za něj. To přináší hráči bonusy za závodění a ostatním hráčům zvýšený požitek ze závodění proti soupeři s charakteristickými vlastnostmi člověka. Tento drivatar zároveň sdílí rank hráče, tudíž se lépe tvoří závody se závodníky, jejichž závodní schopnosti na podobné úrovni.[21]

2.5 TORCS A ŘEŠENÍ JEHO AI

TORCS je simulátor závodů, který se v rámci své kategorie těší značné popularitě. Tento open-source simulátor byl poprvé uveden na trh v roce 2001 a nabízí možnost nahradit řízení vozidla uživatelem programem. Původně byl dostupný pouze na operačním systému Linux, nicméně v současnosti je kompatibilní i s operačním systémem Windows. TORCS je vybaven více než 100 různými modely umělé inteligence určenými k řízení vozidla.

V linuxové verzi lze program pro ovládání vozidla psát pomocí jazyků C a C++, stejně jako samotný kód hry. Ovšem v případě verze pro Windows se objevila chyba, která tento přístup znemožňuje. Komunita vytvořila pro Windows improvizované řešení v podobě serverového patche, který přidává do hry speciální volbu jezdce, jenž přijímá příkazy pro řízení vozidla pomocí protokolu TCP. Tento patch umožňuje přidání až deseti vlastních závodníků.

Data mezi klientem a serverem se vyměňují 20krát za sekundu. Při přidání více externě ovládaných jezdců se tato rychlost úměrně snižuje, takže přidáním více jezdců se celková rychlost trénování nezvyšuje. Toto řešení je poněkud nepraktické, neboť hra vyžaduje ovládání prostřednictvím grafického uživatelského rozhraní (GUI) a pouze jezdce lze ovládat programově pomocí zpráv. Tento přístup však nabízí výhodu v podobě možnosti vytvářet ovládací program v libovolném programovacím jazyku. Komunita se pokusila o nápravu tohoto problému tím, že se pokusila umožnit ovládání serveru pomocí parametrů windows, avšak tato metoda se bohužel ukázala být nefunkční.

V základním balíčku hry je dostupných 36 tratí s možností vytváření vlastních tratí. Některé z těchto tratí jsou replikou skutečných závodních okruhů, zatímco jiné jsou čistě fiktivní. TORCS také obsahuje možnost vytvářet vlastní tratě v před připraveném editoru. V základním nastavení hry TORCS může závodit až 20 účastníků, avšak tento počet je možné upravit. I přes určité nevýhody, je TORCS oceňován pro svou flexibilitu a široké možnosti, které poskytuje svým uživatelům, včetně schopnosti přizpůsobit si hru dle vlastních preferencí a potřeb. Tento aspekt, spolu s možností integrace umělé inteligence pro řízení vozidla, činí TORCS unikátním nástrojem nejen pro zábavu, ale i pro výzkum v oblasti umělé inteligence a autonomního řízení.

2.6 POSTUP ŘEŠENÍ AI DO TORCS

Pro vytvoření AI pro řízení vozidla ve hře TORCS jsem postupoval dle následujících kroků:

1. Příprava prostředí
    - Komunikace s TORCS
    - Implementace algoritmu NEAT
    - Automatizace trénování
2. Analýza optimální konfigurace
    - Analýza aktivační funkce
    - Analýza hodnotící funkce
    - Analýza struktury neuronové sítě
    - Analýza agregační funkce
    - Analýza mutačních pravděpodobností
3. Trénování výsledného modelu
    - Univerzální model
    - Specializovaný model

Tento postup jsem realizoval ve skriptu v jazyce python. Pro komunikaci s TORCS používám parser zpráv, který komunikuje pomocí TCP. Pro algoritmus NEAT používám knihovnu NEAT-python, která jej umožňuje snadno implementovat.

Problém vznikl s automatizací trénovacího procesu, jelikož se mi nepodařilo zprovoznit ovládání TORCS pomocí příkazové řádky a byl jsem nucen simulovat vstup do grafického rozhraní. Což významně zpomalilo rychlost trénování modelů.

Při analýze jsem vždy vycházel ze základní úvahy a experimentálně si hypotézu potvrdil nebo vyvrátil. Parametry pro experimenty jsem volil systematicky, aby bylo možné odhalit chování a závislosti na změnách v nastavení modelu. Každá analýza trvala 1000 generací.

Výsledné modely jsem natrénoval dva. První univerzální, který měl trénovací korpus sestaven ze všech druhů tratí dostupných v TORCS. Tento způsob přinesl několik problémů, tak jako řešení byl natrénován specializovaný model se zaměřením na rychlodráhy, který se těmto problémům vyhnul.

3 PŘÍPRAVA PROSTŘEDÍ

Aby bylo možné trénovat neuronovou síť pro řízení závodního auta je třeba získávat data z okolí vozidla. Tyto data dokáže hra TORCS odesílat sama, tudíž program pro optimalizaci a trénink sítě je musí akorát přijmout, zpracovat a odeslat zpět. TORCS poskytuje následující data o stavu vozu:

Tabulka dat o stavu vozidla TORCS			
Název	jednotka	rozsah	počet vstupů
Úhel natočení auta	rad	$[-\pi, +\pi]$	1
Čas kola	s	$[0,+\infty]$	1
Poškození vozidla	1	$[0,+\infty]$	1
Vzdálenost od startu	m	$[0,+\infty]$	1
Celková ujetá vzdálenost	m	$[0,+\infty]$	1
Senzory vidění	m	[0,200]	5 + 19
Palivo	1	$[0,+\infty]$	1
Rychlostní převod	1	[-1,0,1, ... ,6]	1
Čas minulého kola	s	$[0,+\infty]$	1
Vzdálenost od soupeřů	m	$[0,+\infty]$	36
Pozice v závodě	1	[1, ... ,N]	1
Otáčky motoru	rpm	$[0,+\infty]$	1
Rychlost v ose X	km/h	$[-\infty, +\infty]$	1
Rychlost v ose Y	km/h	$[-\infty, +\infty]$	1
Rychlost v ose Z	km/h	$[-\infty, +\infty]$	1
Pozice na trati	1	$[-\infty, +\infty]$	1
Rychlost otáčení kol	rad/s	$[0,+\infty]$	4
Vzdálenost v ose Z	m	$[-\infty, +\infty]$	1

Tabulka 3.1: Tabulka dat o okolí vozidla

Vstup s úhlem natočení auta slouží k jako primární indikátor pro zatáčení a je vztažen ke směru trati. Vzdálenost od startu se každé kolo resetuje a celková ujetá vzdálenost se nepočítá vůči startu, tudíž v jednom kole lze zajet vzdálenost větší než je délka trati. Auto má 5 přesnějších a 19 méně přesných senzorů pro sledování okolí. Ty sledují vzdálenost od okráje trati v jejich daném směru do vzdálenosti 200 metrů. Přesnější měří s odchylkou 1% a méně přesné s odchylkou 10%. Senzory pro vzdálenost od soupeřů fungují podobně, pouze vzdálenost určují od jiného závodníka, nikoliv od okraje

trati. Osa X je v závodě rovnoběžná se směrem trati a osa Y je napříč tratí. Osa Z prochází tratí a má kladné hodnoty nad a záporné pod tratí.[22]

Pro ovládání vozidla hra používá takzvané efektory.

Tabulka efektorů pro komunikaci s TORCS	
Název	rozsah
plynový pedál	[0,1]
brzdový pedál	[0,1]
pedál spojky	[0,1]
řadící stupeň	[-1,0,1, ... ,6]
otáčení volantu	[-1,1]
rozmístění senzorů	[-90,90]
požádat o restart	[0,1]

Tabulka 3.2: Tabulka efektorů

Většina efektorů je pro snadné ovládání v rozsahu od 0 do 1, kde 1 označuje 100%. Spojkový pedál pro ovládání vozidla není nutný, jelikož lze měnit rychlostní převod i bez jeho sešlápnutí. Ve hře TORCS má každé auto 6 rychlostních převodů, zpátečku a neutrál. Zpátečka je označena hodnotou -1 a neutrál hodnotou 0. Pro otáčení hodnota -1 znamená maximum vlevo a hodnota 1 maximum vpravo a to odpovídá úhlu 20°. Rozmístění ů udává, úhel počátečního a posledního senzoru, typicky bývají určené symetricky. Žádost o restart se aktivuje hodnotou 1 a pokud žádají všichni závodníci, závod se restartuje.

Ve své diplomové práci komunikuji s prostředím TORCS pomocí protokolu TCP. Pro parsování zpráv používám modul `argparse`. Objekt parseru je naplněn požadovanými argumenty jako je například IP serveru, port nebo ID bota. Dále vytvářím vlasntí objekt `driver`, který se stará o řízení vozidla. Pro jeho konstrukci používám argument o nastavení závodu. `Driver` má tři vlastní objekty a to vlastní parser pro klíčování zpráv, objekt `carState` vypovídající o stavu vozidla a objekt `carControl`, který slouží jako obálka dat pro odeslání do TORCS. Dále po navázání komunikaci pomocí TCP spouštím trénování pomocí knihovny `neat-python`. `Main` zavolá metodu `eval_genomes()` podle nastavení a vytvoří počáteční neuronovou síť podle konfiguračního souboru. Následně se v objektu `driver` vkládají data do neuronové sítě a výstup hodnotí fitness funkce. O vyhodnocení nejlepších sítí, přežití populace a mutace se knihovna stará automaticky.

Pro inicializaci třidy Driver používám 2 pomocné třídy `carState` a `carControl`, které slouží jako obálka pro uchovávání dat v přehledném formátu. `CarState` uchovává data o stavu vozidla a `carControl` data pro ovládání.

3.1 KOMUNIKACE SE SERVEREM

Komunikace s TORCS serverem probíhá pomocí protokolu TCP. Skript pro trénování tedy musí tuto komunikaci zahrnovat. Mé řešení obsahuje následující implementaci komunikace TCP:

```
# Configure the argument parser
parser = argparse.ArgumentParser(description=
'Python client to connect to the TORCS SCRC server.')

parser.add_argument('--host', action='store',
                    dest='host_ip', default='localhost',
                    help='Host IP address
                    (default: localhost)')
parser.add_argument('--port', action='store', type=int,
                    dest='host_port', default=3001,
                    help='Host port number
                    (default: 3001)')
parser.add_argument('--id', action='store',
                    dest='id', default='SCR',
                    help='Bot ID
                    (default: SCR)')
parser.add_argument('--maxEpisodes', action='store',
                    dest='max_episodes', type=int, default=1,
                    help='Maximum number of learning episodes
                    (default: 1)')
parser.add_argument('--maxSteps', action='store',
                    dest='max_steps', type=int, default=0,
                    help='Maximum number of steps
                    (default: 0)')
parser.add_argument('--track', action='store',
                    dest='track', default=None,
                    help='Name of the track')
parser.add_argument('--stage', action='store',
                    dest='stage', type=int, default=3,
                    help='Stage (0 - Warm-Up,
                    1 - Qualifying, 2 - Race, 3 - Unknown)')

arguments = parser.parse_args()
```

Parser načte požadované parametry pro připojení k serveru. Mezi důležité parametry patří IP serveru, 'localhost', pokud server běží na lokálním stroji. Další nuté parametry jsou port přednastaven v TORCS na hodnotu 3001 a bot id, které je v server patch uvedeno jako 'SCR'. Ostatní parametry pro mne nejsou důležité jelikož nepoužívám zabudovanou trénovací mechaniku TORCS. Nastavuji tedy počet episod na hodnotu jedna, tudíž pouze jeden start závodu a maximum kroků na hodnotu 0. Trať je nastavená na připojeném

serveru a argument stage také není důležitý, proto volím hodnotu 'unknown'.

Po spuštění se skript pokusí odeslat data a čeká na odpověď ze serveru. Vytvoří socket a odešle zprávu pro inicializaci spojení. Pokud se nepodaří zprávu odeslat, skript se předčasně ukončí. Proměnná go udává směr kanálu, tudíž jestli skript právě data odesílá nebo poslouchá server:

```
while not shutdownClient:
    while True:
        go = True
        print('Sending id to server: ', arguments.id)
        buf = arguments.id + d.init()
        print('Sending init string to server:', buf)

        try:
            sock.sendto(str.encode(buf),
            (arguments.host_ip, arguments.host_port))
        except socket.error as msg:
            print("Failed to send data...Exiting...")

            sys.exit(-1)

        try:
            buf, addr = sock.recvfrom(1000)
        except socket.error as msg:
            # TODO Await and try again
            print("didn't get response from server...")
            go = False;

        if go:
            if buf.find(str.encode('***identified***')) >= 0:
                print('Received: ', buf)
                break
```

Po navázání spojení se vytvořený socket naplní daty z parseru a konstantně odesílá na server:

```
currentStep = 0
while True:
    # wait for an answer from server
    buf = None
    try:
        buf, addr = sock.recvfrom(1000)
    except socket.error as msg:
        print("didn't get response from server...")

    if verbose:
        print('Received: ', buf)

    if buf is not None and buf.find(str.encode('***shutdown***')) >= 0:
        d.onShutDown()
        shutdownClient = True
        print('Client Shutdown')
        break

    if buf is not None and buf.find(str.encode('***restart***')) >= 0:
        d.onRestart()
        print('Client Restart')
        break

    currentStep += 1
    if currentStep != arguments.max_steps:
        if buf is not None:
            buf = d.drive(buf, ge, nets, i, config)
    else:
        buf = '(meta 1)'

    if verbose:
        print('Sending: ', buf)

    if buf is not None:
        try:
            sock.sendto(str.encode(buf),
            (arguments.host_ip, arguments.host_port))
        except socket.error as msg:
            print("Failed to send data...Exiting...")

            sys.exit(-1)
curEpisode += 1
if curEpisode == arguments.max_episodes:
    shutdownClient = True
sock.close()
```

Skript odesílá data pro řízení vozidla voláním metody `drive()`, která je vrací do proměnné `buf`. Podmínka u této metdy si udržuje stav o počtu kroků. Tohoto systému nevyužívám, tudíž nastavuji hodnotu počtu kroků na 0 a tato podmínka bude vždy pravdivá, tudíž nebude docházet k restartu. Po dokončení závodu se socket zavře a čeká se na nové spuštění.

3.2 TRÉNOVÁNÍ NEURONOVÝCH SÍTÍ

Trénování probíhá algoritmem NEAT a využívá knihovnu `NEAT-python`. Kostra trénování je uložena v souboru `main`, kde se spravuje inicializace, nastavení, trénování jednotlivých generací a uložení natrénovaného modelu.

Na počátku probíhá nastavení evoluce:

```
def run_neat(config):
    #p = neat.Checkpointer.restore_checkpoint('neat-checkpoint-580')
    p = neat.Population(config)
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.add_reporter(neat.Checkpointer(5))

    best = p.run(eval_genomes, 11)
    with open("best.pickle", "wb") as f:
        pickle.dump(best, f)


if __name__ == '__main__':
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config.txt')

    config = neat.config.Config(
        neat.DefaultGenome,
        neat.DefaultReproduction,
        neat.DefaultSpeciesSet,
        neat.DefaultStagnation,
        config_path)
    value = input("Train/Test?\n")
    if value == "Train":
        run_neat(config)
    else:
        testai()
```

Proběhne vytvoření konfigurační proměnné pro NEAT z externího textového souboru. Dále se skript zeptá na volbu mezi tréninkem a testováním a podle toho určí start učení. Pro testování není nutné učení zapínat. Pokud je vybrán trénink, zavolá se metoda `run_neat(config)`, která buď vytvoří novou nebo načte uloženou populaci. Pro ukládání populace slouží objekt `reporter`. Zároveň jedince s nejlepším skóre je možné uložit jako návratovou hodnotu: `nejlepšíJedinec = p.run(EvolutionMethod,numberOfEpisodes)`. Ten ukládám ve formátu `.pickle`.

Struktura konfiguračního souboru je kompletně uvedená v dokumentaci. Já uvedu pouze atributy se kterými jsem pracoval a experimentoval v této práci.

Tabulka vybraných konfiguračních parametrů		
Název	hodnoty	popis
fitness criterion	min,max,mean	jedinec s zadaným score musí dosáhnout tresholdu pro ukončení
fitness treshold	číslo	hodnota tresholdu
pop size	číslo	počet jedinců v populaci
activation default	relu, identity, sigmoid ...	počáteční aktivační funkce
activation mutation rate	desetinné číslo	šance na mutaci aktivační funkce
activation options	relu, identity, sigmoid ...	aktivační funkce po mutaci
feed forward	True, False	True - zakazuje tvorbu zpětných vazeb
aggregation default	mean, min, max	počáteční slučovací funkce
agregation mutation rate	číslo	šance na mutaci agregační funkce
aggregation option	mean, min, max	slučovací funkce po mutaci
num hidden	číslo	počet skrytých vrstev
num inputs	číslo	počet vstupních neuronů
num outputs	číslo	počet výstupních neuronů

Tabulka 3.3: Tabulka konfiguračních parametrů

Krom těchto základních parametrů je pro správnou funkcionalitu potřeba nakonfigurovat mnoho dalších a to zejména parametry biasu a vah. U těch je třeba nastavit počáteční hodnoty, rozsah a šanci zmutování. Dalším důle-

žitým parametrem je šance na propojení a odebrání vazby mezi sousedními neurony. `NEAT-python` pracuje s principem elitismu, který funguje jako zámek proti vymření. Dá se uplatnit jak na jedince, tak na druhy. Nastavením elitismu říká, že počet jedinců nebo druhů nemůže klesnout pod danou hodnotu a jsou tedy chráněni.

Při experimentování s cílem dosažení nejlepšího výsledku se mi většina parametrů mění, ale TORCS poskytuje data pro 69 vstupních neuronů a k ovládání vozidla stačí 3 až 4 výstupy (záleží jestli má NN sama řadit). Jelikož 69 vstupů není tak velké množství, není nutné přidávat hodně skrytých vrstev, ideálně 2 nebo 3.

Třída `driver` se stará o ovládání auta. Po zavolání dojde k inicializaci, kde se data ze senzorů viděné přepočítají podle jejich rozmístění a natočení na vozidle. Další informace o těchto senzorech jsem uvedl v tabulce 3.1. Inicializace vypadá následovně:

```
def init(self):
    #Return init string with rangefinder angles
    self.angles = [0 for x in range(19)]

    for i in range(5):
        self.angles[i] = -90 + i * 15
        self.angles[18 - i] = 90 - i * 15

    for i in range(5, 9):
        self.angles[i] = -20 + (i - 5) * 5
        self.angles[18 - i] = 20 - (i - 5) * 5

    return self.parser.stringify({'init': self.angles})
```

Nejdůležitější metoda této třídy, `drive`, se volá konstantně během závodu, kdykoliv kdy se očekává vstup hráče. Na začátku se vypočte přírůstek skóre jedince pomocí hodnotící funkce a zkontroluje se jestli tento jedinec má potenciál na dobrý výsledek, pokud ne, je zažádán restart závodu s novým jedincem. Hodnotící funkce kontroluje ujetou vzdálenost a čas strávený na trati. Zároveň od skóre odečítá poškození vozidla během závodění. Tato funkce je základní varianta a během experimentování jsem ji měnil. Pokud má jedinec potenciál, načtená data se vloží do neuronové sítě. Výstupní data ze sítě jsou následně použita pro řízení vozidla. Výstup 0 reprezentuje zatáčení, vstup 1 řazení, 2 plyn, 3 brzdu a 4 spojku. Každý výstup je limitován aby nepřekračoval dovolené meze a odeslán zpět do prostředí TORCS.

```
def drive(self, msg, ge, nets, i, config):
        self.state.setFromMsg(msg)

        ge[i].fitness = self.state.distRaced - 30 * self.state.curLapTime
        if self.state.damage > 1000:
            ge[i].fitness = -100000
            self.control.setMeta(1)

        data = [...]
        output = nets[i].activate(data)
        if output[0] > math.pi / self.steer_lock:
            output[0] = math.pi / self.steer_lock
        if output[0] < -math.pi / self.steer_lock:
            output[0] = -math.pi / self.steer_lock

        gear = round(output[1] * 6)
        self.control.setSteer(output[0] * math.pi)
        print("gear: " + str(output[1]))
        print("steer: " + str(output[0]))
        print("accel: " + str(output[2]))
        print("break: " + str(output[3]))
        print("clutch: " + str(output[4]))

        if gear > 6: gear = 6
        if gear < -1: gear = -1

        self.control.setGear(gear)
        if output[2] < 0: output[2] = 0
        if output[2] > 1: output[2] = 1
        self.control.setAccel(output[2])

        if output[2] < 0: output[3] = 0
        if output[2] > 1: output[3] = 1
        self.control.setBrake(output[3])

        if output[2] < 0: output[4] = 0
        if output[2] > 1: output[4] = 1
        self.control.setClutch(output[4])

        return self.control.toMsg()
```

Tato verze metody `drive` slouží pro ovládání trénovaného jedince. Pro testování slouží velmi podobná metoda, která však vstupní data čte z uložené neuronové sítě. Liší se v následujících řádcích:

```
self.state.setFromMsg(msg)
data = [...]
output = best.activate(data)
```

Obecně lepších výsledků dosahuje jednodušší verze, kde se model neuronové sítě neučí řadit, ale dochází k automatickému řazení. Toho lze jednoduše dosáhnout pomocí dvou podmínek.

```
if self.state.rpm > 7500:
    gear = gear + 1
elif self.state.rpm < 1500:
    gear = gear - 1
if gear > 6: gear = 6
if gear < 1: gear = 1

self.control.setGear(gear)
```

Toto je velice jednoduchá implementace automatické převodovky. Skript kontroluje otáčky motoru vozidla a pokud překročí nebo spadnou pod danou mez, tak přepne na vyšší nebo nižší převod. Jako meze se mi pro trénink osvědčilo 7500 otáček za minutu pro horní hranici a 1500 otáček za minutu pro dolní. Tyto hodnoty my dávají nejlepší průměrné výsledky.

3.3 OVLÁDÁNÍ GRAFICKÉHO ROZHRANÍ

TCP klient nedokáže ovládat nastavení závodu ze dvou důvodů a to:

- Nemá na to oprávnění. Závodník nesmí měnit pravidla závodu.
- Nastavení závodu probíhá před zapnutím serveru.

Proto je nutné nastavování závodu automatizovat simulací grafického vstupu, který je velice neefektivní a značně limituje rychlost automatizace celého procesu učení. TORCS umožňuje menu ovládat klávesnicí i myší.

Pro ovládání myší lze využít knihovnu pynput. Zde uvádím ukázku jednoduchého ovládání myši pomocí této knihovny, které jsem z počátku využíval:

```
from pynput.mouse import Button, Controller as MouseController

mouse = MouseController()
time.sleep(1)

mouse.position = (320, 460)
mouse.press(Button.left)
mouse.release(Button.left)

mouse.position = (320, 120)
mouse.press(Button.left)
mouse.release(Button.left)
```

Tato část kódu restartovala závod po jeho skončení kliknutím na dvě tlačítka. Extrémní nevýhodou je nutnost držet okno TORCS stále neminimalizované a nesmí se sním pohnout.

Pro ovládání klávesnice lze také použít knihovnu pynput. Testoval jsem knihovnu pywinauto, která dokáže simulovat vstup klávesnice pouze do cílového okna, ale grafické rozhraní TORCS serveru když čeká na připojení neodpovídá a vytvoří chybu.

```
from pynput.mouse import Button, Controller as MouseController
from pynput.keyboard import Key, Controller as KeyController

    win32gui.SetActiveWindow(desktop_window_handle)
    pyautogui.press('enter')
    time.sleep(1)
    pyautogui.press('enter')
```

Toto ovládání je o trochu výhodnější než ovládání myší, jelikož se okno může přesouvat. Podobnou nevýhodou jako u ovládání myší je, že okno musí zůstat aktivní.

Ovládání grafického rozhraní lze obejít použitím Docker obrazu `tkpgamification/torcs_server`. Výhodou tohoto obrazu je automatické startování nových závodů. Je potřeba spustit obraz v kontejneru a připojit se na jeho IP adresu skrze skript. Pro spuštění obrazu z python skriptu lze použít například knihovnu `subprocess`. Ovšem tento způsob je nestabilní a náhodně přestává fungovat. Po aktualizaci operačního systému mi již tento způsob ovládání nefungoval.

```
import subprocess

def runDocker():

    torcs_data_dir =r"C:\Program Files (x86)\torcs"
    race_config_file =
    r"C:\Program Files (x86)\torcs\config\raceman\quickrace.xml"
    docker_command =    ["docker", "run", "-p", "3001:3001", "-v",
                         f"{torcs_data_dir}:/root/.torcs", "-v",
                         f"{race_config_file}:
                         /torcs_server_config/race_config.xml",
                         "tkpgamification/torcs_server", "-l",
                         "1", "-r", "1", "-c",
                         "/torcs_server_config/race_config.xml"]
    runas_command = ["runas", "/user:Administrator"] + docker_command
    try:
        output = subprocess.check_output(
        runas_command, stderr=subprocess.STDOUT)
    except subprocess.CalledProcessError as e:
        output = e.output
        print(f"Error running Docker command: {e}")
    print(output.decode())
```

Tento kód spouští obraz na portu 3001 a s konfiguračním souborem `race_config.xml`. Spouští jeden závod a čeká na jedno připojení. Tento příkaz je spuštěn jako administrátor jelikož jej často blokuje zabezpečení sítě a windows. Pro případné ladění kód zachytává chybu z příkazového řádku a vypíše ji do konzole.

4 VOLBA PARAMETRŮ A JEJICH VLIV

V této kapitole se zabývám nalezení vhodné konfigurace pro dlouhodobé trénování. Hlavní sledovaná kritéria jsou rychlost na trénovacích tratích a rychlost na jiných tratích. V pozdějších experimentech sleduji také míru kolize s ostatními závodníky a celkové poškození vozidla. Parametry které nejvíce ovlivňují tréning neuronové sítě a které se snažím optimalizovat jsou:

- Aktivační funkce
- Hodnotící funkce
- Typ neuronové sítě
- Počet skrytých vrstev
- Pravděpodobnost mutace

4.1 VÝBĚR AKTIVAČNÍ FUNKCE A TYPU NEURONOVÉ SÍTĚ

Hodnotící funkce				
skore = vzdálenost - 30 * čas - poškození				
aktivační funkce	0-250 gen	250-500 gen	500-750 gen	750-1000 gen
identity	59:80	1:03:66	1:03:40	1:03:40
identity+(inv)	45:60	45:60	47:59	47:62
sigmoid	-	-	-	-
sigmoid+(inv)	1:35:23	50:54	46:35	50:32
tanh	45:51	48:47	43:88	43:88
tanh + (inv)	46:60	44:03	44:07s	43:54
tanh + (gauss,sigmoid,inv)	47:40	50:08	50:08	50:08
tanh + (sigmoid,inv,relu)	1:19:20	47:75	47:75	53:35

Tabulka 4.1: Tabulka časů v závislosti na aktivační funkci

Tabulka výše zobrazuje naměřené časy v závislostech na počtu generací a aktivační funkci na stejné závodní trati. Tyto měření mají společné ostatní parametry konfigurace neuronové sítě, jako je počet vstupů a výstupů, hodnotící funkce, pravděpodobnost mutace, agregační funkce, typ neuronové sítě a další.

Nejhůře z testovaných funkcí dopadla funkce sigmoid, která sama nedokázala auto ovládat při daných parametrech a pokud byla přidána jako možnost mutace, pak výsledek byl vždy horší nežli pokus bez funkce sigmoid.

Funkce identita take není příliš vhodná a potřebuje doplnit funkcí inverse, aby se blížila časům funkce tangents hyperbolicus, která dosáhla nejlepších výsledků a blíží se času hráče začátečníka.

V další analýze zkoumám pouze aktivační funkce založené na funkci hyperbolický tangens a přidávám do nich další možnosti pro mutaci. V experimentu je použitá rozdílná síť oproti prvnímu měření a to s rekurentní strukturou. Tato struktura by měla umožňovat detailnější učení s reakcemi na minulé stavy, ovšem snižuje rychlost učení.

Hodnotící funkce				
skore = vzdálenost - 30 * čas - poškození				
aktivační funkce	0-250 gen	250-500 gen	500-750 gen	750-1000 gen
tanh	44:56	45:75	44:43	44:46
tanh+(inv)	44:48	44:09	45:72	46:82
tanh+(relu)	1:10:29	1:08:22	1:08:44	1:08:51
tanh+(sin)	55:24	1:05:86	56:15	58:22
tanh+(cube)	-	-	-	-
tanh+(clamped)	1:13:47	1:06:54	1:00:17	54:00
tanh+(elu)	1:13:98	1:12:41	1:08:01	1:09:64
tanh+(log)	57:10	52:82	1:09:25	55:12
tanh+(log + inv)	-	1:20:73	1:26:18	1:17:14
tanh+(identity)	1:02:23	1:11:72	1:15:57	1:11:65
tanh + (gauss,sigmoid,inv)	47:40	50:08	50:08	50:08
tanh + (sigmoid,inv,relu)	1:19:20	47:75	47:75	53:35

Tabulka 4.2: Tabulka časů v závislosti na aktivační funkci

Na této analýze zakládám volbu aktivační funkce v dalších analýzách a při konečném trénování modelu. Aktivační funkce hyperbolický tangens se potvrdila jako nejlepší dostupná možnost a to bez další přidaných funkcí s výjimkou funkce inverse. Použití rekurentních sítí v krátkodobé analýze nepřineslo žádné zlepšení a tudíž od této struktury v dalších modelech ustupuji.

4.2 VÝBĚR HODNOTÍCÍ FUNKCE

Pro tuto úlohu asi nejdůležitějším parametrem je hodnotící funkce. Pro výpočet skóre modelu řídící automobil jsem uvažuji do hodnotící funkce relevantní následující veličiny:

- uražená vzdálenost
- čas
- poškození auta
- průměrnou rychlost

Uražená vzdálenost slouží jako hlavní způsob modelu jak získat kladné skóre. Veličinou času podporuji rychlou jízdu negativním koeficientem okolo hodnoty 30, což odpovídá průměrné závodní rychlosti vozidla. Odečítáním od skóre poškození auta se snažím předejít kolizím. Některé hodnotící funkce ztrácí skóre za počet uzlů v modelu. Tímto se snažím udržet model jednoduchý a tudíž snadněji naučitelný.

Další veličiny jako jsou otáčky jednotlivých kol nebo vzdálenost od okraje trati se ukázaly jako přílišně komplikující trénovací proces a nejlepší výsledky dává kombinace výše zmíněných.

Seznam hodnotících funkcí:

1. $score = distance - 28 \cdot time - damage - 0,1 \cdot nodes_c$
2. $score = distance - 29 \cdot time - damage - 0,05 \cdot nodes_c + 200 - speedX$
3. $score = 1,5 \cdot distance - 32 \cdot time - damage - 0,2 \cdot nodes_c$
4. $score = distance - 28 \cdot time - damage + \frac{distance}{time}$
5. $score = distance - 28 \cdot time - damage - 0,05 \cdot nodes_c$
6. $score = distance - 28 \cdot time - damage - 0,1 + speedX - 200$
7. $score = 15 \cdot \frac{distance}{time} - damage$
8. $score = 1,1 \cdot distance - 40 \cdot time - 0,25 \cdot damage$
9. $score = distance - 30 \cdot time - damage - 1 \cdot nodes_c$
10. $score = distance - 30 \cdot time - damage - 3 \cdot nodes_c$
11. $score = distance - 30 \cdot time - damage - 10 \cdot nodes_c$
12. $score = distance - 30 \cdot time - 0,5 \cdot damage - 10 \cdot nodes_c$

hodnotící funkce	0-250 gen	250-500 gen	500-750 gen	750-1000 gen
1	1:02:68	57:40	1:02:55	1:02:98
2	1:24:01	1:23:98	1:31:43	1:26:44
3	1:17:12	1:11:98	1:20:79	1:23:41
4	43:39	1:10:29	1:16:12	1:20:93
5	56:54	56:34	53:92	52:11
6	1:29:29	1:28:64	1:23:73	1:24:58
7	1:10:15	56:03	1:01:23	59:60
8	-	-	-	-
9	48:22	54:60	51:46	52:24
10	49:58	52:35	50:49	49:74
11	50:02	51:68	51:12	51:12
12	47:67	47:61	47:85	47:44

Tabulka 4.3: Tabulka časů v závislosti na hodnotící funkci

Modifikace hodnotící funkce se odvíjejí od základního funkce, kterou jsem vypočítal z poměru jednotlivých veličin, které TORCS poskytuje. Tyto modifikace nedosahují stejně dobrých výsledků jako původní funkce.

Nejlepší čas byl dosažen pomocí funkce číslo 4 a to 43:39, což překonává dosavadní nejlepší natrénovaný čas, ovšem pravděpodobně se jedná o náhodný výsledek, jelikož časy v následujících generacích jsou velmi špatné. Funkce číslo 8 založená na průměrné rychlosti nedokázala dokončit jediné kolo. Po zpětné analýze to bylo způsobeno výpočtem průměrné rychlosti. TORCS počítá průměrnou rychlost na konci kola a tudíž pokud model nedojede kolo, má automaticky hodnotu 0. Tudíž počáteční iterace modelu se vůbec nedokázaly zlepšit a dojet kolo.

Přínosem této analýzy bylo zjištění, že koeficient u poškození vozidla je v základní funkci příliš vysoký a tudíž se model spíše snaží jezdit bezpečně a ne rychle, což pro závodní počítačovou hru není preferované. Dalším poznatkem je fakt, že trestání modelu za jeho složitost značně snižuje výkyvy v zajetých časech a model se chová jako konzervativnější.

4.3 VOLBA POČTU SKRYTÝCH VRSTEV A MUTAČNÍ PRAVDĚPODOBNOSTI

Další analyzovanou vlastností modelu je vliv skrytých vrstev. Počet skrytých vrstev obvykle roste s se složitostí problému, který model řeší. Můj konkrétní model zpracovává 33 vstupů pro řízení v jízdě sólo a 64 vstupů a 3, případně 4, výstupy. To je velmi málo oproti standardním konfiguracím pro hluboké neuronové sítě, tudíž v analýze experimentuji pouze s několika jednotkami skrytých vrstev.

počet skrytých vrstev	0-250 gen	250-500 gen	500-750 gen	750-1000 gen
0	-	-	2:39.48	2:45.51
1	0:58.41	0:54.73	0:55.02	0:54.08
2	0:45.51	0:48.47	0:43.88	0:43.88
3	0:49.20	0:52.15	0:48.12	0:47.45
4	0:53.34	0:56.28	0:51.17	0:50.37
5	0:57.48	1:00.41	0:54.22	0:53.29
6	1:07.62	1:04.54	0:57.27	0:59.21
7	1:15.76	1:18.67	1:11.32	1:08.13
8	-	1:22.80	1:23.37	1:17.05
9	-	1:58.93	1:36.42	1:34.97

Tabulka 4.4: Tabulka časů v závislosti počtu skrytých vrstev

Z této analýzy jsem odhadl optimální počet skrytých vrstev a to 2 skryté vrstvy. Pro 0 skrytých model nebyl schopný obsáhnout komplexnost řízení a po 500 generaci si našel alternativní způsob, kterým byl schopen dorazit do cíle. Model s 1 skrytou vrstvou už byl relativně schopen ovládat vozidlo, ovšem model se 2 skrytými vrstvami dál značně lepší výsledky. Pro modely se 3 a více skrytými vrstvami se začaly výsledky zhoršovat. Toto je pravděpodobně způsobenou zvýšenou složitostí modelu a tudíž proces trénování je pomalejší. Ačkoliv by více skrytých vrstev mohlo dát lepší výsledky, rozhodl jsem se vyjít z této analýzy a zvolit 2 skryté vrstvy pro dlouhodobě trénovaný model

Volba mutační pravděpodobnosti

Touto analýzou jsem nestrávil tolik času jako u ostatních. Vyzkoušel jsem několik možností mutací a to jak aktivační, tak agregační funkci. Mnohem větší vliv se ukázala mutace pro aktivační funkci. Obecně větší pravděpodobnost mutace podporuje hledání nových řešení, tudíž pro krátkodobé učení je lepší volit větší pravděpodobnost a později ji snižovat. Tudíž tento parametr u dlouhodobého trénování upravuji v průběhu učení.

5 TRÉNOVÁNÍ FINÁLNÍHO MODELU

Vzhledem k problémům s paralelizací trénování nezbylo tolik času na finální model, kolik jsem původně plánoval. Ukázalo se, že největším faktorem pro trénování je výběr trénovacích dat. Model samozřejmě dosahuje nejlepších výsledků, pokud byl trénován i testován na jedné a té same trati, ovšem u dalších tratí dochází k problému s overfittingem. Místo jednoho finálního modelu jsem tedy natrénoval 2 modely.

První, univerzální, model je soustředěný jak bylo prvotně zamýšleno na univerzalitu a je natrénován na širokém spektru tratí. Při trénování tento model vykazoval zhoršující se výsledky s přidáváním dalších tratí, ovšem byl schopen všechny tratě po finálním tréninku dokončit. Ukázalo se, že některé tratě obsahují prvky, na jejichž sledování senzory vozidla nejsou úplně uzpůsobeny. Mezi tyto prvky patří například naklonění dráhy a změna povrchu vozovky. Naklonění vozovky lze částečně odvodit ze změny polohy vozidla v ose Z, ovšem to se vztahuje k centru vozidla a tudíž není možné určit směr naklonění. Zároveň většina ostatních tratí tyto prvky nepoužívá a tedy NEAT má tendence tomuto senzoru přikládat malou váhu.

Z tohoto důvodu jsem natrénoval druhý, specializovaný, model, který je natrénován pouze na tratích složených ze standardních prvků. Tento model byl trénován na tratích označených jako ”speedway” neboli rychlodráha. Tyto tratě mají typicky jednoduchý tvar. Tento model si vedl velmi dobře i na dalších tratích typu ”speedway” (na kterých se netrénoval), ovšem nedokázal dojet komplikované tratě.

Ukázalo se, jak významný vliv na výsledek modelu má množina trénovacích dat. Jako další krok je určitě vhodné lépe analyzovat jednotlivé dostupné tratě a vybrat trénovací data tak, aby obsahovala všechny prvky, které by se v závodě mohly objevit. Zároveň v této práci nevěnuji pozornost ostatním jezdcům. Hra sice nepodporuje mechaniky jako je draftování (jízda ve vzduchovém polštáři za jiným závodníkem), ale je vhodné vyhnout se kolizím s ostatními.

trať	univerzální model	specializovaný model	Berniw model
Speedway 1	57:99 (Trénován)	43:94 (Trénován)	46:72
A Speedway	42:59 (Trénován)	37:17 (Trénován)	38:13
E-Track 5	42:12	37:89 (Trénován)	39:62
Dirt 1	42:33 (Trénován)	36:18	39:00
CG Track 2	1:10:49	1:28:41	1:03:49
CG Track 3	-	1:25:14	1:13:40
Forza	3:41:48	-	1:40:14
Olethros road	2:38:75	2:53:15	2:05:73
Ruudskogen	-	-	1:14:12
E Speedway	42:12	37:89	54:09
Michigan Speedway	48:92	54:93	44:43
Dirt 2	-	-	1:30:04
Dirt 3	1:19:20	1:07:51	1:11:73
Dirt 4	1:50:80	1:32:64	1:29:78

Tabulka 5.1: Tabulka výsledků finálních modelů

6 POROVNÁNÍ VÝSLEDKŮ A DISKUZE

Porovnání mých modelů s existujícími přístupy ukazuje, že navzdory omezenému tréninku, jsou tyto modely schopné dosáhnout srovnatelných výsledků. Specializovaný model se ukázal jako konkurenceschopný modelu Berniw (model jezdce přímo z TORCS) v oblasti rychlosti na jednoduchých tratích, zatímco univerzální model prioritizoval spolehlivost.

V práci „Autonomní řidič závodního vozu v TORCS“ [23] autor využívá genetické algoritmy podobně jako já v této práci. Avšak jeho metoda zahrnuje prvotní mapování trati a analýzu optimální stopy, což je rozdílný přístup od mého, který se zaměřuje na přímou adaptabilitu modelu. Autorův přistup se ukázal jako méně vhodný, pro tuto konkrétní úlohu, jelikož dosáhl horších výsledků. Například na trati Speedway 1 jeho model zajel kolo za 2:03:10, kdežto můj specializovaný model za 43:94.

Vědecký článek „Driving in TORCS using modular fuzzy controllers“ [24] představuje alternativní přístup, který se vyznačuje použitím fuzzy logiky místo neuronových sítí. Model založený na stavovém automatu se ukázal jako velmi efektivní, zejména v reálných závodních podmínkách, kde autor závodí s ostatními jezdci. Na trati Speedway 1 dosáhl času 46:16 a na trati E-track 5 času 29:80. Tyto výsledky naznačují, že rozdílné metody mohou být účinné pro různé situace a kontexty.

Ačkoli mé modely vykazovaly pozoruhodné výsledky, naráží na několik omezení. Specializovaný model má potíže s komplexnějšími tratěmi a není schopen se přizpůsobit naklonění dráhy. Na druhou stranu, univerzální model má tendenci jezdit blízko vnitřního okraje trati, což může vést k problémům na tratích s určitými vlastnostmi, jako je přítomnost štěrku nebo pevných překážek.

Tyto výsledky naznačují, že budoucí práce by se měla zaměřit na zlepšení adaptability modelů na různé typy tratí a jízdních podmínek. Může být také prospěšné prozkoumat integraci různých metod, jako je kombinace genetických algoritmů s fuzzy logikou nebo stavovými automaty, aby se zlepšila univerzálnost a výkonnost modelů.

7 ZÁVĚR

Vytvořil jsem skript v jazyce Python, který dokáže komunikovat se simulátorem TORCS pomocí protokolu TCP. Ve skriptu je použit algoritmus NEAT pro trénování neuronové sítě jako autonomního jezdce.

Provedl jsem analýzu parametrů a experimentálně určil optimální konfiguraci.

Natrénoval jsem dva modely: univerzální a specializovaný. Univerzální model byl trénován tak, aby byl schopen dokončit co nejvíce tratí. Specializovaný model byl trénován na jednodušších asfaltových tratích s primárním zaměřením na rychlost.

Výsledky modelů na tratích, které nebyly zahrnuty do trénování, byly signifikantně horší než na tratích kde byl model trénován. Přesto specializovaný model na některých z těchto tratí předčil model Berniw (model jezdce přímo v TORCS), avšak nebyl schopen několik tratí dokončit. Tyto tratě, které model nedokončil, se vyznačují složitější dráhou, na kterou se model nedokázal dostatečně adaptovat a předpovědět potenciální situace. Hlavním problémem specializovaného modelu byla neschopnost přizpůsobit se naklonění dráhy, což vedlo k téměř jisté kolizi.

Univerzální model rovněž nedokončil některé tratě, ale z odlišných důvodů. Tento model se specializoval na jízdu u vnitřního okraje trati, což v kontextu simulátoru TORCS představuje významný problém. Na krajích vozovky často bývá štěrk, který mění jízdní vlastnosti auta, a protože simulátor TORCS neposkytuje informace o těchto povrchových změnách, model to není schopen zaznamenat. Zároveň na některých tratích se při okraji vozovky vyskytují pevné překážky, kterým je třeba se vyhnout. Tento model se obzvláště potýká s problémy na tratích s užší vozovkou.

NEAT se ukázal jako algoritmus schopný natrénovat model pro jízdu vozidla ve hře TORCS, ovšem není ideálním řešením. Pro lepší výsledky je třeba NEAT kombinovat s jinými způsoby pro trénování neuronových sítí. Dalším způsobem pro značné zlepšení výsledků je důkladná analýza jednotlivých tratí, výběr vhodné množiny pro trénování a delší trénovací doba.

ZDROJE

[1] Reinforcement learning. Geeksforgeeks [online]. A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305: GeeksForGeeks [cit. 2023-05-21]. Dostupné z: https://www.geeksforgeeks.org/what-is-reinforcement-learning

[2] BIGDATA. History of AI Use in Video Game Design. Big Data Analytics News [online]. BDAN, 2021 [cit. 2023-05-21]. Dostupné z: https://bigdataanalyticsnews.com/history-of-artificial-intelligence-in-video-games/

[3] Neuronové sítě. ČVUT [online]. Praha: ČVUT, 2017 [cit. 2023-05-21]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/a6m33dvz/dvz2017-05-nnet.pdf

[4] KOSTADINOV, Siemon. Understanding Backpropagation Algorithm. In: Towards Data Science [online]. Towards Data Science, 2019 [cit. 2023-05-21]. Dostupné z: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

[5] PILAT, Martin. Neuronové sítě - úvod. Martin Pilat [online]. [cit. 2023-05-21]. Dostupné z: https://martinpilat.com/cs/prirodou-inspirovane-algoritmy/neuronove-site-uvod

[6] PILAT, Martin. Neuronové sítě - RBF sítě a rekurentní sítě. Martin Pilat [online]. [cit. 2023-05-21]. Dostupné z: https://martinpilat.com/cs/prirodou-inspirovane-algoritmy/neuronove-site-rbf-site-rekurentni-site

[7] A review of supervised machine learning algorithms. In: IEEE Xplore [online]. New Delhi, India: IEEE Xplore, 2016 [cit. 2023-05-21]. Dostupné z: https://ieeexplore.ieee.org/abstract/document/7724478

[8] CUSSAT-BLANC, Sylvain, Kyle HARRINGTON a Jordan POLLACK. Gene Regulatory Network Evolution Through Augmenting Topologies [online]. IEEE Xplore, 2015 [cit. 2023-05-21]. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7018989

[9] Evoluční algoritmy. ČVUT [online]. Praha: ČVUT [cit. 2023-05-21]. Dostupné z: http://mech.fsv.cvut.cz/ leps/teaching/mmo/prednasky/prednaska08_EAs.pdf

[10] CHEN, Lin. NeuroEvolution of Augmenting Topologies with Learning for Data Classification. In: IEEE Xplore [online]. USA: IEEE Xplore, 2020 [cit. 2023-05-21]. Dostupné z: https://ieeexplore.ieee.org/abstract/document/4250190

[11] DRCHAL Jan, Jan KOUTNIK, Miroslav SNOREK a Ondřej KAPRAL. NeuroEvolution of Augmenting Topologies with Learning for Data Classification. In: IEEE Xplore [online]. ResearchGate, 2009 [cit. 2023-05-21]. Dostupné z: https://ieeexplore.ieee.org/abstract/document/4250190

[12] HEIDEMREICH, Hunter. HyperNEAT: Powerful, Indirect Neural Network Evolution. Towardsdatascience [online]. towardsdatascience, 2019 [cit. 2023-05-21]. Dostupné z: https://towardsdatascience.com/hyperneat-powerful-indirect-neural-network-evolution-fba5c7c43b7b

[13] The ES-HyperNEAT Users Page. Eplex [online]. Eplex [cit. 2023-05-21]. Dostupné z: http://eplex.cs.ucf.edu/ESHyperNEAT/

[14] MOHAMMAD. Neuro-Evolution of Augmenting Topologies Algorithm. Brainy Loop [online]. Brainy Loop, 2022 [cit. 2023-05-21]. Dostupné z: https://brainyloop.com/neuro-evolution-of-augmenting-topologies/

[15] STANLEY, Kenneth O., Bobby D. BRYANT, Risto MIIKKULAINEN a Student Member, IEEE. Real-Time Neuroevolution in the NERO Video Game. IEEE Xplore [online]. IEEE Xplore, 2005 [cit. 2023-05-21]. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1545941

[16] Tensor Flow Neuro Evolution Framework: CoDeepNEAT Overview [online]. [cit. 2023-05-21]. Dostupné z: https://tfne.readthedocs.io/en/latest/codeepneat/codeepneat-overview.html

[17] MOHAMED, Abdal. Artificial Inteligence in Racing Games. In: University of Birmingham [online]. University of Birmingham [cit. 2023-05-21]. Dostupné z: https://www.cs.bham.ac.uk/ ddp/AIP/RacingGames.pdf

[18] VINCENT, James. OpenAI's Dota 2 defeat is still a win for artificial intelligence. In: The Verge [online]. The Verge, 2018 [cit. 2023-05-21]. Dostupné z: https://www.theverge.com/2018/8/28/17787610/openai-dota-2-bots-ai-lost-international-reinforcement-learning

[19] OpenAI Five defeats Dota 2 world champions. In: OpenAI [online]. OpenAI, 2019 [cit. 2023-05-21]. Dostupné z: https://openai.com/blog/openai-five-defeats-dota-2-world-champions/

[20] GORDON, Rob. F1 22 Review: A Speedy But Greedy Gas-Guzzler. In: Screen Rant [online]. Screen Rant, 2022 [cit. 2023-05-21]. Dostupné z: https://screenrant.com/f1-22-game-review/

[21] THOMPSON, Tommy. How Forza's Drivatar Actually Works. In: Game Developer [online]. Game Developer, 2022 [cit. 2023-05-21]. Dostupné z: https://www.gamedeveloper.com/design/how-forza-s-drivatar--actually-works

[22] LOIACONO, Daniele, Luigi CARDAMONE a Pier Luca LANZI. Simulated Car Racing Championship Competition Software Manual. In: Arxiv [online]. Arxiv, 2013 [cit. 2023-05-21]. Dostupné z: https://arxiv.org/pdf/1304.1672.pdf

[23] BĚHAL, Lukáš. Autonomní řidič závodního vozu TORCS. Brno, 2012. Diplomová práce. VUT. Vedoucí práce Ing. Jiří Jaroš.

[24] MOHAMMED, Salem, Antonio MORA, Juan Julián Merelo GUERVÓS a Pablo García SÁNCHEZ. Driving in TORCS Using Modular Fuzzy Controllers. Reseatchgate [online]. Reseatchgate, 2017 [cit. 2023-05-21]. Dostupné z: https://www.researchgate.net/publication/315639430_Driving_in_TORCS_Using_Modular_Fuzzy_Controllers