VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

NAVIGACE POMOCÍ HLUBOKÝCH KONVOLUČNÍCH SÍTÍ

NAVIGATION USING DEEP CONVOLUTIONAL NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. DALIBOR SKÁCEL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAL HRADIŠ, Ph.D.**
SUPERVISOR

BRNO 2018

Abstrakt

Tato práce se věnuje problematice navigace a autonomního řízení za použití konvolučních neuronových sítí. Jsou zde představeny hlavní přístupy využívající zpracování senzorických vstupů uváděné v odborné literatuře a popsána teorie neuronových sítí, imitačního a zpětnovazebního učení. Dále jsou zde popsány nástroje a metody vhodné pro zpracování systému řízení. Jsou vytvořeny dva typy modelů pro řízení vozidel v simulačním prostředí. Modely využívají učících algoritmů DAGGER a DDPG. Vytvořené modely jsou otestovány v prostředí simulátoru TORCS.

Abstract

This thesis studies navigation and autonomous driving using convolutional neural networks. It presents main approaches to this problem used in literature. It describes theory of neural networks and imitation and reinforcement learning. It also describes tools and methods suitable for a driving system. There are two simulation driving models created using learning algorithms DAGGER and DDPG. The models are then tested in car racing simulator TORCS.

Klíčová slova

Autonomní navigace, neuronové sítě, konvoluční neuronová síť, navigace v 3D prostoru, hluboké učení, zpětnovazební učení, simulované řízení vozidel.

Keywords

Autonomous navigation, neural networks, convolutional neural network, 3D navigation, deep learning, reinforcement learning, simulated driving.

Citace

SKÁCEL, Dalibor. *Navigace pomocí hlubokých konvolučních sítí*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Hradiš Michal.

Navigace pomocí hlubokých konvolučních sítí

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením Ing. Michala Hradiše, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Dalibor Skácel
23. května 2018

Poděkování

Děkuji svému vedoucímu Ing. Michalu Hradišovi, PhD. za rady a návrhy, které mi k této práci poskytl. Také děkuji svým rodičům za obrovskou podporu po celou dobu studia.

Obsah

Kapitola 1

Úvod

Problematice autonomního řízení a navigace vozidel je věnováno mnoho odborných prací a v minulém desetiletí dosáhly jejich výsledky znatelného pokroku. Společenská poptávka poslední doby po nejrůznějších autonomních systémech, které lze zahrnout například do konceptů Smart Cities, bezpečnostních systémů, atd. je opravdu vysoká. Autonomní navigace se stává nezbytnou u různých typů robotů pohybujících se v budovách, na cestách i v terénu. U silničních vozidel jsou již běžné parkovací automaty a testují se autonomně řízená vozidla v běžném silničním provozu. Stále se také ve velké míře vyvíjejí počítačové modely pohybující se ve virtuálním prostředí, a to většinou pro účely výzkumu a inovací nových přístupů a technik, a ve hrách.

Mnoho dnešních systémů je založeno na využití vícenásobných senzorů, jako jsou lasery a radary. Vedle toho jsou vyvíjeny čistě pasivní systémy spoléhající pouze na obraz z kamer. Ty jsou dostupnější, menší a energeticky méně náročné než aktivní senzory, ovšem pro získání užitečné informace je třeba složitého zpracování obrazu. V této práci se dále zabývám systémy využívajícími obrazové vstupy i senzory.

V poslední době zaznamenaly obrovský nárůst popularity také neuronové sítě. Je tomu tak především díky dobrým výsledkům, které podávají v odvětvích jako jsou klasifikace obrázků, zpracování přirozeného jazyka a v neposlední řadě hraní her. Právě hraní her má velmi blízko k řízení vozidel v simulačním prostředí. Existuje mnoho her, které se snaží řízení co nejvěrněji zreplikovat a modely neuronových sítí, naučené v takovémto prostředí, mohou být později využity k řízení v reálném světě.

Další technikou, která může být pro autonomní řízení použita je zpětnovazební učení. Jedná se o typ strojového učení, při kterém umělý agent svými akcemi ovlivňuje prostředí a učí se pomocí odezvy, kterou tak vyvolá. Výhodou takového učení je, že agent jen nekopíruje svého učitele, ale sám prozkoumává prostředí a vytváří vlastní strategii chování. Taková strategie může být lepší, než jakou by ho dokázal naučit učitel. Zpětnovazební učení je také často kombinováno s neuronovými sítěmi, touto technikou jsou například učeny programy AlphaGO a AlphaZero, které jsou v hraní šachu a Go zatím neporazitelné.

Cílem této práce je uvést do problematiky autonomního řízení, neuronových sítí a zpětnovazebního učení a poté tyto techniky aplikovat na návrh a implementaci autonomního systému řízení v simulačním prostředí.

Členění práce

Tato práce je rozdělena do osmi hlavních kapitol.

Ve druhé kapitole je popsán úvod do problematiky využití strojového učení pro řízení vozidel a jsou zde představeny a analyzovány hlavní přístupy, které se v ní používají.

Kapitola 3 popisuje vědecké články a práce, které se zabývají podobnými problémy a které byly inspirací k tvorbě této práce. Jedná se především o práce z poslední doby, věnující se autonomnímu řízení a zpětnovazebnímu učení.

V kapitole 4 je popsán úvod do neuronových a konvolučních neuronových sítí, je vysvětleno jakým způsobem se tyto sítě učí a jak mohou být použity k řízení vozidel.

V kapitole 5 je popsán úvod do zpětnovazebního, hlubokého zpětnovazebního učení a imitačního. Dále jsou zde představeny metody, které se v tomto odvětví v praxi používají a které jsou použity pro návrh systému řízení v dalších kapitolách.

Kapitoly 6 a 7 se věnují výběru vhodných nástrojů a návrhu systému strojového učení pro řízení vozidel v simulátoru.

V kapitole 8 je popsáno testování, ladění parametrů a srovnání navrženého systému.

V Závěru je zhodnocen úspěch a přínos této práce a jsou diskutovány možnosti dalšího rozvoje.

Kapitola 2

Strojové učení pro autonomní řízení

V této kapitole jsou shrnuty jednotlivé přístupy řešení problému autonomního řízení a navigace vozidel použité v literatuře. Tyto přístupy se liší nejen tím, jaké vstupní informace jejich systémy využívají, jakými prostředky jsou modelovány a učeny, ale také tím, v jakém prostředí se pohybují a jak jsou testovány. Zmiňuji se také o dalších oblastech využití autonomně řízených systémů využívajících konvoluční neuronové sítě.

2.1 Přístupy k autonomnímu řízení se zpracováním obrazu

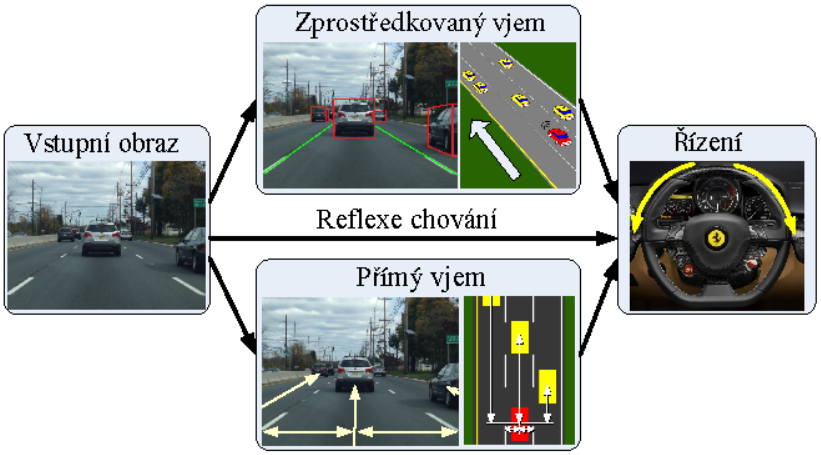
Podle práce Ch. Chena[4] se pro problematiku autonomního řízení v zásadě používají tři hlavní druhy přístupů: Přístupy zprostředkovaných vjemů (mediated perception approaches), které zpracovávají celou scénu k vytvoření rozhodnutí pro řízení vozidla, přístupy odvozené od chování (behavior reflex approaches), které vytvářejí řídící povely přímo ze vstupního obrazu pomocí regrese, a přístup přímého vjemu (direct perception approach), který vytváří řídící rozhodnutí podle omezené skupiny indikátorů. Graficky jsou tyto přístupy zobrazené na obrázku 2.1.

Přístup zprostředkovaných vjemů

Přístupy zprostředkovaných vjemů zahrnují mnoho dílčích částí k rozpoznávání objektů relevantních pro řízení vozidla, jako jsou ostatní účastníci silničního provozu, dopravní značení atd. Tyto rozpoznané objekty a jejich umístění představují obraz skutečného okolí v daném okamžiku, na jehož základě se systém pro řízení vozidla rozhoduje. Nevýhodou tohoto přístupu je, že analyzuje i objekty, které nejsou pro řízení relevantní a systém je pak velmi složitý.

Přístup odvozený od chování

Přístupy odvozené od chování mapují vstupy získané ze senzorů na řídící povely. Pro takovéto mapování z obrazu na úhel natočení volantu byla již v osmdesátých letech použita neuronová síť [16]. Pro naučení modelu bylo vozidlo řízeno člověkem a trénovací data byla pořízena snímáním vozovky kamerou a snímáním natočení volantu. Příkladem přímého přístupu je i článek z roku 2016 *End to end learning for Self-Driving Cars*[3]. Přístup z tohoto článku představuje jedinou palubní kameru, jejíž výstupy se pomocí konvoluční neuronové



Obrázek 2.1: Znázornění přístupů k autonomnímu řízení. Převzato z [4]

sítě přímo transformují na příkazy k řízení. Jediným výstupem učícího procesu je úhel natočení volantu při řízení, bez explicitní detekce jakýchkoliv objektů. Tím je celkový systém jednoduší a při učení se snaží optimalizovat celkovou schopnost systému řídit namísto optimalizace určených mezikroků, jako je detekce silničních pruhů apod. Tato kritéria usnadňují interpretaci systému pro člověka, ale nemusí zajišťovat maximální výkonnost systému. Takovýto systém vidí celý problém mapování vstupních obrazů na úhel natočení volantu jako jediný nedělitelný úkol, který se učí od konce do konce.

Přístup přímého vjemu

Přístup přímého vjemu byl prezentován v roce 2015 v práci Ch. Chena[4] a je kombinací předchozích dvou přístupů. Obraz se namapuje na několik indikátorů dopravní situace, jako jsou vzdálenost vozidla od silničních pruhů, ostatních vozidel a úhel natočení vozidla směrem k vozovce.

2.2 Snímání vstupních dat

Dalším rozdílným prvkem je počet použitých kamer pro snímání obrazů, které se používají jako vstupy neuronových sítí. Je možné použít obraz z jediné kamery [8, 3], dvou nebo více kamer zabírající jiné úhly pohledu [6, 12] nebo více kamer směřujících dopředu, kde kamery posunuté od středu vozidla simulují posun vozidla v jízdním pruhu[4].

Testovací prostředí Systémy se také mohou lišit podle prostředí pro které jsou navrhovány. Od simulovaného prostředí představovaného například 3D dopravním simulátorem přes různé typy dopravní infrastruktury až po volný terén bez vozovky či dopravního značení.

Modely a jejich učení Starší systémy jako ALVINN [16] používaly plně propojené neuronové sítě. Nyní většina moderních systémů[3, 4, 12] používá ke zpracování obrazu konvo-

luční neuronové sítě. Ty jsou používány již přes dvacet let, ale v posledních letech dosáhly velkého rozvoje díky implementaci učících algoritmů konvolučních sítí na paralelních grafických procesorech a dostupnosti vhodných datasetů.

Pro učení systémů řízení se také využívá zpětnovazební učení [9, 2]. Při tomto přístupu není dostupný dataset, ale systém při učení prozkoumává různé stavy a na učí se na základě odměn z hodnotící funkce. Takto je možné naučit systém reagovat na nové prostředí (např. nové typy cest) bez nutnosti sbírání dat. Nevýhodou tohoto přístupu je nutnost využít pro učení simulovaného prostředí. J. Koutník et al. prezentuje ve své práci [9], že využitím zpětnovazebního učení lze dosáhnout výsledků srovnatelných se systémy používající učení s učitelem (*supervised learning*).

Kapitola 3

Související práce

V této kapitole je proveden rozbor prací souvisejících s tématy strojového učení a autonomního řízení. Téma strojového učení se v poslední době stále rozvíjí, především se zvyšováním výkonu výpočetní techniky. Značnou popularitu strojového učení přinesly nedávné velké úspěchy týmu DeepMind při aplikaci učících systémů na hry šach a go [19, 18].

První pokusy použít strojové učení pro řízení vozidel byly popsány již v roce 1989 v práci D. A. Pomerleaua [16], ve které využívá jednoduchou neuronovu síť k přímému mapování obrazů na řídící povely. Podobný přístup používá Yann LeCun [12] při učení terénního vozidla překonávat překážky. Učící systém je postaven na velké šestivrstvé konvoluční síti, na jejímiž vstupu jsou dvojice obrazových snímků s nízkým rozlišením.

Další posun přinesl tým společnosti Nvidia [3], který sestrojil systém využívající třech kamer umístěných na přední straně vozidla. Upravené obrazové snímky z těchto kamer jsou přiváděny na vstup konvoluční sítě, která na výstupu přímo generuje řídící povely. Tento systém ovládá pouze zatáčení vozidla s nutností manuálního ovládání plynu a brzdy, ale je schopen se naučit řídit reálné vozidlo po silnicích a venkovských cestách po zpracování 100 hodin záznamu řízení z kamer. Všechny výše zmíněné práce využívaly lidmi nasbíraný dataset obrazových snímků.

V roce 2013 představil tým Google DeepMind svůj algoritmus DQN [15], který je schopný hrát různé hry z konzole Atari 2600 bez nutnosti předem nachystaného datasetu. Tento a jemu podobné algoritmy se začaly aplikovat i na řízení vozidel. Jelikož se jedná o algoritmy s učením bez učitele (unsupervized learning), které potřebují velký počet iterací algoritmu, využívá se pro učení simulační prostředí. Timothy P. Lillicrap [2] využívá řízení vozidla v simulaci jako jedno z prostředí pro testování algoritmů zpětnovazebního učení se spojitou množinou stavů a akcí. Podobné metody s větším zaměřením na řízení vozidel testuje ve své práci Adythia Ganesh [1].

V následujících podkapitolách jsou podrobněji popsány ty práce z poslední doby, jejichž zaměření je této práci nejblíže.

3.1 DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving

Chen a kolektiv z Princetonské university ve své práci DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving[4] kombinují základní přístupy uvedené v kapitole 2.1, tedy přístup odvozený od chování a přístup přímého vjemu. K učení a testování systému také využívají závodní simulátor TORCS. Obraz pohledu z vozidla namapují na

13 indikátorů dopravní situace, jako jsou např. vzdálenost vozidla od silničních pruhů, ostatních vozidel a úhel natočení vozidla směrem k vozovce.

Pro toto namapování používají konvoluční neuronovou síť typu AlexNet [10]. Tato síť má pět konvoučních vrstev následovaných čtyřmi plně propojenými vrstvami a využívá euklidovskou loss funkci.

Dataset použitý pro naučení konvoluční neuronové sítě je nasbírán ručním řízením auta v simulátoru po dobu 12 hodin se zaznamenáváním obrazu a výstupů senzorů auta obsažených v simulátoru (jako jsou rychlost auta, vzálenosti od okrajů vozovky a natočení auta vzhledem k ose vozovky). Obrazové snímky slouží jako vstupy pro neuronovou síť a senzorická data slouží jako požadovaná výstupní hodnota (ground truth).

Samotné řídící povely (plyn, brzdu a zatáčení) vypočítává heuristický algoritmus, který zpracovává indikátory získané na výstupu neuronové sítě. Natočení volantu přitom závisí na úhlu natočení vozidla k ose vozovky a vzdálenosti od jejího středu. Zatímco hodnoty pedálů brzdy a plynu závisí především na aktuální rychlosti a jsou ovlivněny překážkami (jinými vozidly).

Takto získaný systém dokáže udržet auto v jízdním pruhu a reagovat na ostatní auta na vozovce do vzdálenosti až 60 metrů. Tato práce ukazuje, jak je možné využít obrazová data k řízení vozidla bez nutnosti detailní analýzy každého snímku. Systém byl vedle simulátoru testován i na datech z palubní kamery reálného auta. Výsledky těchto testů ukazují dobrou schopnost systému rozpoznávat silniční pruhy, ale informace o dalších vozidlech bývá nepřesná.

3.2 Využití hlubokého zpětnovazebního učení pro hraní her

Společnost DeepMind spadající pod Google publikovala v roce 2013 hojně citovanou práci Playing Atari with Deep Reinforcement Learning[15]. Je v ní představena metoda Deep Q Network (DQN), která se dokáže učit řešit problémy s vysoce dimenzionální reprezentací stavů. Touto metodou je naučen model, který je schopen hrát většinu her z konzole Atari 2600 na úrovni lidského hráče. Vysoká dimenzionalita problému je jedním ze společných rysů automatického hraní počítačových her a řízení vozidla v simulačním prostředí. Ovšem u her je většinou snazší jednotlivé stavy diskretizovat.

Jejich model se skládá z konvoluční neuronové sítě, která snižuje dimenze vstupu a redukuje vstup na vektor příznaků, a plně propojené neuronové sítě, která tento vektor zpracovává. Model se učí algoritmem Stochastic Gradient Descent (SGD). Je zde představena takzvaná *replay memory*, což je paměť, do které jsou během učení ukládány záznamy zkušeností z her. Z této paměti je v každé iteraci algoritmu vybrán náhodný vzorek záznamů, který je použit pro učení sítě. Další novinkou představenou v této práci je využití tzv. *cílových* (*target*) sítí. To jsou sítě, které jsou kopií právě učené sítě, s tím rozdílem, že změny parametrů se v těchto sítích projevují mnohem pomaleji. *Cílové* sítě se využívají k předpovídání budoucích odměn. Provedené pokusy ukazují, že při jejich použití se systém chová stabilněji.

Velmi dobré výsledky této práce a představení systému hlubokého učení, který je schopen se učit velmi odlišné vysoko dimenzionální problémy, přinesly značný ohlas a pozvedly zájem o zpětnovazební učení.

3.3 koutník

zde bude možná popsán Koutníkův článek o end-to-end řízení

3.4 Hluboké zpětnovazební učení pro spojité řízení

článek o ddpg, možná

3.5 Deep Reinforcement Learning for Simulated Autonomous Driving

Tato práce od týmu vedeným Adithyou Ganeshem ze Stanfordské university z roku 2017 [1] se zabývá využitím hlubokého Q-učení pro autonomní řízení v závodním simulátoru TORCS. Používá knihovny TensorFlow a Keras k učení plně propojených hlubokých neuronových sítí.

Předvádějí na pokusech, že klasické metody Q-učení nejsou pro problém autonomního řízení příliš vhodné. Proto autoři testují metodu Deep Deterministic Policy Gradient (DDPG), poprvé představenou ve článku Timothyho P. Lillicrapa [2], a architekturu Long Short Term Memory (LSTM) sítí.

Agenti se sítěmi naučenými metodou DDPG jsou schopni provést auto několika odlišnými tratěmi v simulátoru. LSTM sítě tak dobré výsledky nepodávají a autoři uvádějí, že je třeba doladit hyperparametry, kterých je v této architektuře vysoký počet. V závěru dodávají, že by bylo dále vhodné u metody DDPG otestovat různé funkce odměn, více strategií explorace a různé nastavení hyperparametrů pro možné získání lepších výsledků.

Kapitola 4

Neuronové a konvoluční sítě

Umělá neuronová síť je výpočetní model inspirovaný lidským mozkem a stejnojmenným biologickým systémem. Skládá se z množství vzájemně propojených výpočetních jednotek zvaných neuronů. V této kapitole je popsán úvod do problematiky neuronových sítí a modelů, které tato práce využívá.

4.1 Neuronové sítě

Neuron Umělý neuron se svojí stavbou inspiruje teorií biologického neuronu. Každý neuron má po vzoru biologického neuronu několik vstupů a jediný výstup. Základní funkce neuronu spočívá ve sběru několika elementárních informací na vstupech, jejich vyhodnocení a poslání odpovídajícího signálu na výstup. Neuron má obecně n vstupů, které tvoří vstupní vektor $\vec{x} = (x_1...x_n)$. Každý z těchto vstupů je ohodnocen vahou, kterou se tento vstup násobí. Všechny váhy pak vytváří vektor $\vec{w} = (w_1...w_n)$. Model neuronu nejčastěji obsahuje i práh (*bias*), který funguje jako další vstup, ale je zpravidla napojen na konstantní hodnotu. Model umělého neuronu je zobrazen na obrázku 4.1. Formálně je neuron definovaný následující rovnicí:

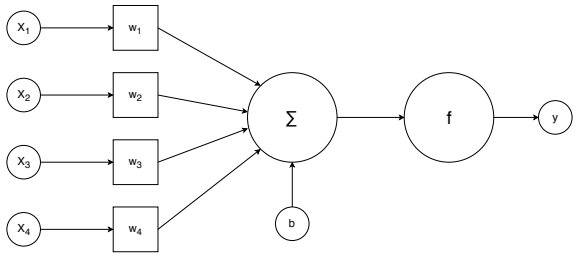
$$y = f(\sum_{i=1}^{n} x_i w_i + b). \tag{4.1}$$

Kde:

- y je výstupní signál neuronu
- n je velikost vstupního vektoru
- x_i je i-tý prvek vstupního vektoru
- w_i je i-tá váha neuronu
- f je přenosová (nebo také nazývaná aktivační) funkce neuronu
- b je práh, neboli bias

Mezi nejčastěji používané přenosové funkce patří Rectified Linear Unit ($ReLU$):

$$f(x) = \begin{cases} x & x > 0 \\ 0 & jinak \end{cases}. \tag{4.2}$$



Obrázek 4.1: Umělý neuron.

V této práci kromě *ReLU* využívám z důvodu omezeného oboru hodnot jako přenosové funkce i sigmoidu:

$$f(x) = \frac{1}{1 + e^{-x}}, \tag{4.3}$$

a hyperbolický tangens:

$$f(x) = tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{4.4}$$

Výstup neuronu je zpravidla napojen na vstup jiného neuronu. Takovýmto propojením několika neuronů vzniká neuronová síť. Počet neuronů a jejich vzájemné propojení udává topologii sítě. Běžná architektura neuronové sítě rozděluje neurony do vrstev, kde jedna je vstupní, jedna výstupní a n tzv. skrytých vrstev mezi nimi. Váhy neuronů v síti bývají reprezentovány váhovou maticí.

Princip učení neuronových sítí

Učením neuronové sítě je myšlen proces úpravy matice vah takovým způsobem, aby síť podávala požadované výstupy. Existují dva hlavní přístupy k učení: učení s učitelem (*supervised learning*) a bez učitele (*unsupervised*).

Učení s učitelem vyžaduje pro jednotlivé hodnoty na vstupu sítě i odpovídající výstupy (v anglické literatuře *ground truth*), které dohromady tvoří trénovací množinu. Matice vah sítě je na začátku učení inicializována náhodnými hodnotami. Vektor vstupních hodnot je přiveden na vstupní vrstvu sítě, následně je zpracován sítí a výsledek na výstupní vrstvě je porovnán s vektorem požadovaných výstupních hodnot. Pomocí dosažených a očekávaných výstupů je spočítána chybová funkce (také nazývána jako *objective function*), podle které je následně upravena matice vah sítě. Chybová funkce může mít různé definice, jedna z nejpopulárnějších je Mean Squared Error (MSE, rozdíl čtverců), která je definovaná jako:

$$E = \frac{1}{2} \sum_{i}^{N} \sum_{j}^{n} (y_{ij} - d_{ij})^2, \tag{4.5}$$

kde n je dimenze prostoru, N je počet vzorů, y je výstup získaný ze sítě a d je požadovaný výstup. Při učení se pak hledá minimální hodnota chybové funkce.

Při učení bez učitele systém nemá k dispozici k jednotlivým vstupním vektorům požadované výstupy. Síť se musí pouze ze vstupních vzorů rozhodnout, které odezvy jsou pro dané vstupy nejvhodnější a podle toho upravit svoji matici vah. Tento přístup se používá například při problémech klastrování.

Metoda zpětného šíření chyby Základní a nejpoužívanější metodou k učení dopředných neuronových sítí je metoda zpětného šíření chyby (*backpropagation*). Vstupem algoritmu je trénovací množina $\{(x_1 d_1), ..., (x_N d_N)\}$

4.2 Optimalizační algoritmy

Optimalizační algoritmy pomáhají minimalizovat (nebo maximalizovat) chybovou funkci E, která se používá pro učení sítě. Základním takto používaným algoritmem je metoda Gradient Descent (taktéž gradientní sestup), ke které existuje řada úprav a vylepšení. V základní verzi tento algoritmus funguje následovně: nechť $\theta_i(t)$ je i-tá váha sítě v čase t, pak aktualizace vah sítě probíhá podle gradientu chybové funkce E podle jednotlivých vah:

$$\theta_i(t+1) = \theta_i(t) - \alpha \frac{\delta E}{\delta \theta_i}, \tag{4.6}$$

kde $\alpha < 1$ je parametr učení (*learning rate*).

Stochastická gradientní metoda Gradient Descent v každé iteraci algoritmu počítá gradienty přes celou trénovací množinu, což je velmi časově náročné. Pro zrychlení je možné gradient aproximovat pouze pomocí vybrané dávky (*batch*) vzorů, která je náhodně vybrána z trénovací množiny. Takováto úprava se nazývá stochastická gradientní metoda (*Stochastic Gradient Descent*, SGD).

Do optimalizačního algoritmu je také možné zapojit tzv. moment, který má za úkol zamezit oscilacím v prostoru chybové funkce. To znamená, že při aktualizaci vah se ke gradientu připočítává rozdíl současných a minulých hodnot vah:

$$\theta_i(t+1) = \theta_i(t) - \alpha \frac{\partial E}{\partial \theta_i} + \alpha_m(\theta_i(t) - \theta_i(t-1)), \tag{4.7}$$

kde α_m je parametr důležitosti momentu.

Další možností úpravy optimalizačního algoritmu je varianta *AdaGrad*, kde je namísto globálního parametru učení použit parametr pro každou váhu zvlášť. Tento parametr také nemusí být pouze statický, ale může exponenciálně klesat, což je využito ve verzi zvané *RMSprop*. Bližší popsání si zaslouží algoritmus *Adam*, který jsem využil v této práci.

Optimalizační algoritmus Adam Algoritmus *Adam* (Adaptive Moment Estimation) je metodou, která počítá adaptivní parametr učení pro každou váhu. *Adam* uchovává exponenciálně klesající průměr minulých gradientů m_t, podobný momentu, a exponenciálně klesající průměr druhých mocnin minulých gradientů v_t. Parametr m_t je nazýván *první moment* a parametr v_t *druhý moment*, definovány jsou následovně:

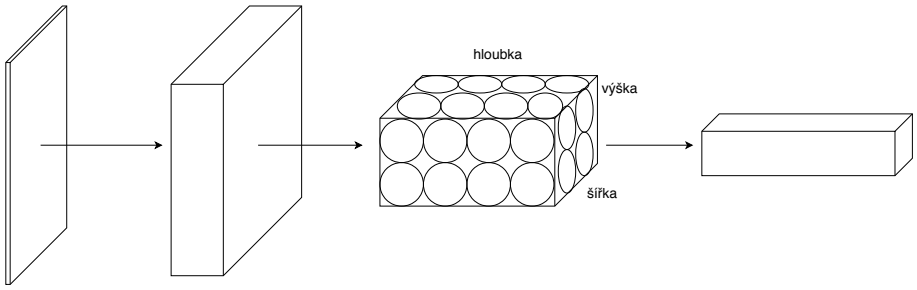
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{4.8}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{4.9}$$

kde β_1 a β_2 jsou parametry klesání (*decay rates*), které autoři algoritmu nastavili na hodnoty $\beta_1 = 0,9$ a $\beta_2 = 0,999$, a g_t je vektor gradientů pro jednotlivé váhy. Momenty m_t a v_t mají v počátečních iteracích algoritmu tendenci blížit se k nule, proto se počítají korekce těchto momentů $\hat{m}_t$ a $\hat{v}_t$:

Algoritmus 1 Adam

$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$t \leftarrow 0$
while θ_t nezkonvergovala **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \Delta_\theta f_t(\theta_{t-1})$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
  $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
  $\theta_{t+1} \leftarrow \theta - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$
end while
return θ_t



Obrázek 4.2: Schematické znázornění konvoluční sítě

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \tag{4.10}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \tag{4.11}$$

Ty jsou použity k aktualizaci vah pomocí pravidla:

$$\theta_{t+1} = \theta - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{4.12}$$

kde parametr ϵ autoři nastavují na hodnotu 10^{-8}. Celý pseudokód této metody je popsán v algoritmu 1.

4.3 Konvoluční síť

Konvoluční neuronové sítě(CNN/ConvNets) jsou velmi podobné běžným neuronovým sítím. Celková síť také vyhodnocuje jednotlivou diferencovatelnou hodnotící funkci: z původních pixelů obrázku na vstupu do hodnot výstupních klasifikačních tříd. Běžné plně propojené neuronové sítě se příliš nehodí na zpracování celých obrazů, protože obraz větší velikosti vede k obrovskému počtu vah. Tato plná propojenost je nadbytečná a velký počet parametrů vede k přeučení (*overfitting*). To znamená, že se síť naučí reagovat na konkrétní dataset, ale

nedokáže výpočet zobecnit na jiná vstupní data. Proto se pro zpracování obrazu zpravidla používají konvoluční sítě, které redukují počet propojení neuronů a tím i vah.

Architektura konvolučních sítí však vychází z explicitního předpokladu, že na vstupu jsou obrazy, které umožňují zakódovat určité vlastnosti do architektury. Tím se stane dopředná funkce mnohem efektivnější pro implementaci a významně redukuje množství parametrů v síti.

Konvoluční neuronové sítě využívají faktu, že vstup sestává z obrazových snímků, a vytvářejí účelnější architekturu. Na rozdíl od běžných neuronových sítí jsou vrstvy konvolučních sítí uspořádány ve třech rozměrech: šířka, výška, hloubka. Rozměr hloubka zde neznamená hloubku celé neuronové sítě, ale představuje třetí dimenzi aktivačního svazku. Každá vrstva konvoluční sítě převádí 3D vstupní svazek na 3D výstupní svazek aktivací neuronu. Šířka a výška jsou dány rozměrem obrazu a hloubka může být například 3 pro RGB kanály. Vizuální reprezentace konvoluční sítě je zobrazena na 4.2.

Vrstvy konvolučních sítí

Jednoduchá konvoluční neuronová síť je tvořena posloupností vrstev, kde každá vrstva transformuje jeden svazek aktivací na jiný prostřednictvím diferencovatelné funkce. V architekturách CNN se používají se tři hlavní typy vrstev: konvoluční (*convolutional layer*), subsamplingová vrstva (*pooling layer*) a plně propojená vrstva (*fully connected layer*).

Konvoluční sítě transformují původní obraz z hodnot pixelů do výsledných tříd. Některé vrstvy obsahují parametry a některé je obsahovat nemusí. Konkrétně konvoluční a plně propojené vrstvy provádějí transformace, které jsou funkcí nejen aktivací vstupního svazku, ale také parametrů (vah a bias neuronů). Jejich parametry se učí metodou gradient descent. ReLU vrstvy a subsamplingové vrstvy implementují pevné funkce.

4.4 Konvoluční vrstva

Konvoluční vrstvy jsou základními stavebními prvky konvolučních sítí a provádějí hlavní díl výpočetní zátěže. Parametry konvolučních vrstev sestávají ze sady učících se filtrů. Každý z filtrů je prostorově malý podle šířky a výšky, ale roztažený na celou hloubku vstupního svazku. Například typický filtr první vrstvy konvoluční sítě může mít rozměry $5 \times 5 \times 3$ (5 pixelů $\times$ 5 pixelů vstupního obrazu jako šířka a výška $\times$ 3 barevné kanály jako hloubka). Během dopředného průchodu se každý filtr posouvá přes šířku a výšku vstupního svazku a počítají se bodové produkty mezi vstupy filtru a vstupem v každé pozici. Při posuvu filtru přes šířku a výšku vstupního svazku se vytvoří 2D příznaková mapa, která dává odezvy tohoto filtru v každé prostorové pozici. Potom síť učí filtry, aby se aktivovaly když rozpoznají nějaký druh vizuálního příznaku, jako například orientovanou hranu nebo barevnou skvrnu, na celé vzory ve vyšších vrstvách sítě. Takto se získá celá sada filtrů na každé konvoluční vrstvě a každý z nich vytvoří samostatnou 2D příznakovou mapu. Tyto mapy se naskládají za sebe podle hloubky a vytvoří výstupní svazek.

Příznaková mapa Cílem použití příznakových map je získání lokální charakteristiky obrazu. Lokální znamená, že se jedná o podobraz, máme-li obraz například 32×32, může být velikost podobrazu například 5×5. Každý pixel tohoto podobrazu bude vstupem jednoho neuronu. Každý neuron tedy bude mít $5 \times 5 = 25$ vstupů a *bias*. Vstupní obraz se pokryje výřezem dané velikosti tak, že podobrazy se přitom překrývají. Protože každý z těchto

podobrazů tvoří vstup jednoho neuronu, je počet neuronů stejný jako počet podobrazů. Sada všech těchto neuronů se nazývá příznaková mapa.

Prostorové uspořádání Velikost výstupního svazku neuronů v konvoluční vrstvě je obecně určena třemi hyperparametry[7]: hloubkou, krokem posunu (*stride*) a nulováním okrajů (*zero padding*).

- Hloubka výstupního svazku je první hyperparametr, odpovídá počtu použitých filtrů. Každý z nich se učí vyhledávat různé příznaky ze vstupu. Pokud první konvoluční vrstva má na vstupu původní obraz, potom neurony v různé hloubce mohou být aktivovány při přítomnosti určitého příznaku, například orientovaných hran či barevných skvrn. Sada neuronů, která je napojená na stejný podobraz se nazývá hloubkový sloupec nebo vlákno(depth column, fibre).
- Při posunování výřezu se pracuje s určitým krokem posunu. Pokud je krok jedna, posunují se filtry po jednom pixelu. Pokud je dva nebo více (což je neobvyklé), filtry se posunou o daný počet pixelů v jednom kroku. Tím se získají prostorově menší výstupní svazky.
- Dalším hyperparametrem je velikost nulovaných okrajů. Vynulování vstupního svazku kolem jeho okrajů je výhodné protože umožňuje řídit prostorovou velikost výstupních svazků. Obvykle se nulování okrajů používá pro zachování stejné velikosti vstupního a výstupního svazku.

Prostorovou velikost výstupního svazku lze vypočítat jako funkci velikosti vstupního svazku W, velikosti vnímaného pole neuronů konvoluční vrstvy (výřezu) F, použitého kroku posunu výřezu S a velikost nulovaného okraje P. Vzorec pro výpočet potřebného počtu neuronů je $(W - F + 2P)/S + 1$. Použití hyperparametrů pro prostorové uspořádání má vzájemná omezení, je třeba zvolit takové parametry, aby výsledný počet potřebných neuronů byl celé číslo.

Sdílení vah

Pro řízení počtu vah v konvoluční vrstě se používá techniky sdílení parametrů. Při jejím použití sdílejí všechny neurony ve stejné hloubce svazku(v řezu svazkem) váhy včetně bias. Díky tomu se při posunutí lokálních příznaků v rámci obrazu se výstup neuronů také jen posune. V reálném případě[11] při velikosti výstupního svazku konvoluční vrstvy $55 \times 55 \times 96$ tvoří vrstvu 290400 neuronů a každý má $11 \times 11 \times 3$ vah a jeden bias. Celkem by tedy bylo potřeba 105705600 parametrů jen v první vrstvě konvoluční sítě, což je obrovský počet. Při použití techniky sdílení parametrů při velikosti svazku $55 \times 55 \times 96$ je pouze 96 řezů, jejichž neurony mají různé parametry. Takže celkový počet vah je $96 \times 11 \times 11 \times 3 = 33848$, navíc je zde $96 \times bias$, což dohromady činí 34944 parametrů. V praxi počítá každý neuron ve svazku během zpětného šíření gradient pro své váhy, ale tyto gradienty se v každém řezu sečtou a update se provede pouze pro jedinou sadu vah pro každý řez.

4.5 Subsamplingová vrstva

Do architektury konvolučních neuronových sítí se mezi dvě po sobě následující konvoluční vrstvy se obvykle vkládá subsamplingová vrstva, která obsahuje stejné množství příznakových map, jako předchozí konvoluční vrstva. Každá příznaková mapa ze subsamplingové

vrstvy je spojena s jednou příznakovou mapou z vrstvy předchozí. Neurony příznakových map subsamplingové vrstvy jsou napojeny na nepřekrývající se podoblasti příznakových map konvoluční vrstvy. Protože se podoblasti nepřekrývají, je každá taková příznaková mapa menší než příslušná mapa předchozí vrstvy.

Účelem subsamplingové vrstvy je významně redukovat prostorovou velikost, aby se snížil počet parametrů a výpočtů v síti a tím i zabránilo přeučení. Subsamplingová vrstva pracuje nezávisle na každém hloubkovém řezu vstupu a mění jeho velikost. Nejběžnější velikost filtrů v subsamplingové vrstvě je 2×2 s krokem 2, ta redukuje počet aktivací o 75%. V praxi se běžně používají pouze dvě varianty subsamplingové vrstvy MaxPooling: s parametry *F=3, S=2* a častější *F=2, S=2*.

Kapitola 5

Zpětnovazební a imitační učení

Tato kapitola popisuje způsoby strojového učení, které lze použít pro řešení problémů, mezi které patří i navigace vozidel.

Jednu skupinu tvoří imitační učení, které napodobuje chování učitele (tzv. *experta*). Druhou skupinou jsou algoritmy zpětnovazebního učení, které se učí na základě zpětné vazby z prostředí. V obou případech je aktivním prvkem agent, často implementovaný prostřednictvím neuronové sítě.

5.1 Zpětnovazební učení

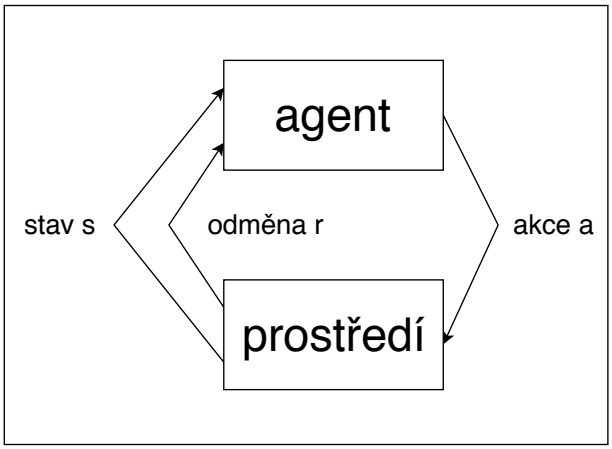
Zpětnovazební učení (také nazýváno posilované učení z anglického *Reinforcement Learning*) je typ strojového učení, které je založeno na zjištění kroků, které vedou k maximalizaci užitku. Teorie zpětnovazebního učení vychází z behaviorální psychologie, kde se jedinec učí na základě pozitivní či negativní zpětné vazby.

Základním mechanismem zpětnovazebního učení je interakce agenta a prostředí. Na základě vykonávaných akcí získává agent z prostředí zpětnou vazbu, zpravidla číselnou odměnu, podle níž se učí. Agent tedy nemá k dispozici ohodnocený dataset z daného prostředí, ale musí průzkumem zjistit, které akce ho dovedou k vyšším odměnám. Zpětnovazební učení nelze přímo zařadit pod učení s učitelem nebo bez učitele. Tuto charakterizaci je nutné aplikovat až na jednotlivé konkrétní metody. Základními prvky posilovaného učení jsou:

- Agent - získává informace z prostředí a vykonává akce
- Prostředí - představuje informace o stavu problému
- Akce - úkony vykonávané agentem, vyvolávající změnu v prostředí
- Odměna - představuje číselné vyhodnocení úspěšnosti akce vykonané agentem
- Strategie agenta - mapování stavu prostředí na pravděpodobnost výběru konkrétní akce

Komunikace agenta s přostředím je také znázorněna na obrázku 5.1. Cílem je maximalizovat odměny v dlouhodobém hledisku. Akce často nemá vliv pouze na odměnu v témže stavu, ale i na odměny v následujících stavech. Více informací o zpětnovazebním učení je možné nalézt v knize R. S. Suttona [20].

Systém je popsán množinou stavů S a množinou akcí A. Strategie (*policy*) $\pi = P(a|s)$ představuje pravděpodobnost zvolení akce a ve stavu s. Celková očekávaná odměna při



Obrázek 5.1: Vztah agenta a prostředí ve zpětnovazebním učení.

rozhodování podle strategie π ze stavu s se nazývá hodnotící funkce (*value function*) a značí $V^{\pi}(s)$. Většina algoritmů zpětnovazebního učení je založena na odhadnutí hodnotící funkce. Podle této funkce se určí, do jaké míry je pro agenta výhodné jít do daného stavu nebo z daného stavu zvolit konkrétní akci. Pro Markovovský rozhodovací proces se $V^{\pi}(s)$ dá vyjádřit formálně:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\{\sum_{k=0}^{\inf} \gamma^k r_{t+k+1} | s_t = s\}, \tag{5.1}$$

kde E_{π} je očekávaná hodnota, které agent dosáhne, pokud následuje strategii π v každém časovém kroku t.

Stejným způsobem lze definovat i hodnotu volby akce a ve stavu s podle strategie π, kterou určuje tzv. *Q-funkce*:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\inf} \gamma^k r_{t+k+1} | s_t = s, a_t = a\}, \tag{5.2}$$

kde r_t je odměna v kroku t a γ je tzv. *slevový* parametr.

Explorace

Explorace je metoda pro výběr akcí, které agent během učení z daného stavu provede. Při náhodném výběru akcí je účinnost agenta velmi nízká.

Jednou z nejčastějších metod je $\epsilon - greedy$. Při této metodě je zvolen parametr ϵ, který značí pravděpodobnost, že agent zvolí náhodnou akci. V opačném případě (s pravděpodobností $1 - \epsilon$) je zvolena akce s nejvyšší hodnotící funkcí. Jako ϵ se volí hodnota v rozsahu 0...1, nejčastěji 0,1 nebo 0,2. Tuto hodnotu je také možné během učení snižovat, díky čemu se ze začátku volí více náhodné akce a jak se zvyšuje naučenost systému, volí se častěji akce s nejvyšší hodnotou.

Další běžně používanou metodou je *SoftMax* selekce. Ta se snaží pokrýt nevýhodu metody $\epsilon - greedy$,která při výběru náhodné akce nebere v potaz hodnotu jednotlivých stavů a méně ohodnocené stavy mají stejnou šanci být vybrány, jako stavy ohodnocené vysoko. Metoda *SoftMax* přiřadí stavům pravděpodobnostní hodnotu, která je funkcí jejich hodnoty Q. Tímto přístupem jsou stavy častěji vybírány, pokud je jejich ohodnocení vyšší,

Algoritmus 2 Q-učení

1: $Q(s,a) \leftarrow 0(s,a)$
2: $s \leftarrow$ Získání počátečního stavu
3: **repeat**
4: Proveď akci $argmax_{\alpha \in A(s)} Q(s,a)$
5: $s', r \leftarrow$ Nový stav, Odměna
6: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a)]$
7: $s \leftarrow s'$
8: **until** není splněno vhodné kritérium konce

ale hůře ohodnocené stavy mají šanci být prozkoumány také. Akce a v čase t je zvolena s pravděpodobností:

$$P = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}, \tag{5.3}$$

kde hodnota τ je kladné číslo zvané teplota. Při nizkých hodnotách τ je větší rozdíl pravděpodobnosti volby mezi hůře a lépe hodnocenými akcemi a při větší teplotě se tento rozdíl naopak vyrovnává.

Q-učení

Algoritmus Q-učení (z anglického Q-learning) je jednou z nejpopulárnějších metod zpětnovazebního učení. Tento algoritmus uvádí funkci $Q^{\pi}(s,a)$, která definuje kvalitu volby akce a ve stavu s při použití strategie:

$$\pi(s) = \arg \max_{a \in A(s)} Q(s,a). \tag{5.4}$$

Před začátkem učení je hodnota Q zvolena arbitrárně. Nechť je agent ve stavu s, získá odměnu $R(s)$ a provede akci a, čímž se dostane do nového stavu s'. Pak je Q funkce aktualizována podle pravidla:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a)], \tag{5.5}$$

kde α je parametr učení a γ je *slevový* parametr, který definuje jak velký význam mají pro agenta odměny získané v budoucnu oproti odměně ze současného stavu s. Celý proces Q-učení je popsán v algoritmu 2.

5.2 Hluboké zpětnovazební učení

Algoritmy jako je Q-učení nejsou ve své základní variantě příliš vhodné pro řešení problémů s vysoce dimenzionálním prostorem stavů či akcí (tedy problémy blízké těm z reálného světa). Je tomu tak z důvodu, že tyto algoritmy si uchovávají tabulku všech navštívených stavů a vykonaných akcí. Tato tabulka by v těchto případech měla velmi velké rozměry, znamenající také velkou prostorovou a časovou náročnost algoritmu. Tento problém se snaží řešit hluboké zpětnovazební učení.

Při hlubokém zpětnovazebním učení (*deep reinforcement learning*) je k aproximaci Q funkce využita vícevrstvá neuronová síť. Tyto metody nabyly značné popularitě poté, co

V. Mnih ve své práci [15] prezentoval algoritmus DQN (Deep Q Network), který rozšiřuje metodu Q-učení o neuronovou síť. Pomocí něj byl schopen naučit model, který dokázal hrát 7 her z konzole Atari 2600. Tento algoritmus poté aktualizoval ve svém dalším článku [14], kde počet naučených her zvýšil na 49, z nichž většinu zvládá hrát na úrovni lepší nebo srovnatelné s lidskými hráči.

5.3 Metoda Deep Q Network

Cílem algoritmu *Deep Q Network* (DQN) je aproximovat užitkovou funkci (v anglické literatuře zvanou *action-value function*), definovanou jako:

$$Q(s,a) = \max_{\pi} \mathbb{E}\big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi\big], \tag{5.6}$$

což je maximální suma odměn r_t snížených o γ v každém časovém kroku t, dosažitelná podle strategie π po provedení akce a ve stavu s. Tato aproximace se značí $Q(s,a|\theta)$, kde θ jsou váhy neuronové sítě, a cílem je, aby platilo $Q(s,a|\theta) \approx Q(s,a)$.

Uvažujme, že agent ve stavu s vykonal akci a, získal odměnu $R(s)$ a přešel do nového stavu s', ve kterém má možnost zvolit jakoukoliv akci $a' \in A(s')$. Pak je pro výpočet chybové funkce použito pravidlo:

$$L = \Big(R(s) + \gamma \max_{a' \in A(s')} Q(s', a'|\theta) - Q(s,a|\theta)\Big)^2. \tag{5.7}$$

Autoři ovšem zjistili, že použití nelineárních metod, jako jsou neuronové sítě, pro aproximaci Q funkce často přináší nestabilitu systému a v některých případech i divergenci učení. DQN překonává problém nestabilního učení pomocí technik *Experience Replay* a *cílových* sítí.

Experience Replay Jelikož prostředí, ve kterém se agent pohybuje, probíhá většinou kontinuálně, jsou po sobě jdoucí stavy velmi podobné a korelované (např. pokud se agent učí řídit vozidlo, při projetí jedné zatáčky se mohou zaznamenat až desítky stavů a všechny budou velmi podobné). Pokud by na vstup neuronové sítě stavy přicházely ve stejném pořadí, v jakém je agent vykonává, docházelo by k přeučení sítě. Aby se tomuto jevu zabránilo, ukládají se stavy ve formě tzv. zkušeností do velkého bufferu nazývaného *Experience Replay* nebo také *Replay buffer*, který je detailně popsán v práci S. Zhanga[22]. Každá zkušenost obsahuje veškeré vstupní informace potřebné pro jednu iteraci učení použitých neuronových sítí. Zkušenost je tvořena čtveřicí:

$$E = (s, a, r, s'), \tag{5.8}$$

kde $s \in S$ je jeden stavů simulačního prostředí, $a \in A$ je akce zvolená agentem ve stavu s, $r = R(s)$ je odměna, kterou agent ve stavu s získá a $s' \in S$ je následující stav, do kterého se agent dostal ze stavu s při zvolení akce a. Při učení se z bufferu náhodně vybere takový počet zkušeností, který odpovídá velikosti jedné dávky (*batch*). Ty se poté použijí pro aktualizaci neuronových sítí. Tímto způsobem dostávají sítě na vstupu stavy a akce z různých částí průběhu simulace a nedochází tak k divergenci.

Cílová síť Další problém při hlubokém učení je výpočet požadovaných výstupů, který je závislý na funkci $Q(s', a'|\theta)$. Při aktualizaci vah Q funkce θ se tedy mění i požadované výstupy, což způsobuje nestabilitu. Pro řešení tohoto problému je nutné, aby se váhy, použité pro výpočet požadovaných výstupů během učení neměnily příliš často.

V algoritmu DQN[15] se proto vytváří kopie Q-sítě $Q'(s, a|\theta')$, jejíž váhy θ' se po fixním počtu kroků přepíší váhami θ aktuální Q-funkce. Síť Q' je poté využívána pro výpočet Q-funkce a autoři ji nazývají *cílová síť* (*target net*).

5.4 Metoda Deep Deterministic Policy Gradient

U doposud zmíněných metod zpětnovazebního učení se předpokládá, že prostor akcí je diskrétní. To však nemusí být vhodné, pokud cílem agenta je naučit se nastavovat spojitou hodnotu, jako je například úhel natočení volantu vozidla nebo úhel paže robota. Pro takové úkoly bylo nutné spojitou hodnotu rozdělit na několik diskrétních hodnot, což má za následek ztrátu přesnosti. Tento problém řeší algoritmus Deep Deterministic Policy Gradient (DDPG), který byl popsán v roce 2016 v práci T. P. Lillicrapa [2].

Algoritmus DDPG patří do rodiny metod typu aktér-kritik (actor-critic), které využívají dvě funkce. První je tzv. *aktér* $\mu(s|\theta^\mu)$, ten má za cíl aproximovat strategii π. Na vstup dostane stav $s \in S$, ve kterém se agent nachází, a jeho výstupem je konkrétní akce $a \in A$ agenta pro daný stav s. Druhou využívanou funkcí je tzv. *kritik* $Q(s, a)$, ten má na vstupu dvojici stav $s \in S$ a akce $a \in A$ a jeho cílem je odhadovat kumulativní očekávané užitky, které agent získá pokud ve stavu s zvolí akci a. V algoritmu DDPG jsou obě tyto funkce aproximovány neuronovými sítěmi.

Aktér je učen podle řetězového pravidla (chain rule) aplikovaného na očekávanou odměnu z počáteční distribuce J vzhledem ke svým vahám θ^μ:

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}\right]. \quad (5.9)\end{aligned}$$

Kritik je učen podle Bellmanovy rovnice stejným způsobem, jako při Q-učení a DQN.

Přímá implementace Q-učení s neuronovými sítěmi je v mnoha prostředích nestabilní. To je způsobeno tím, že učená Q síť je zároveň využita pro výpočet cílové hodnoty. Autoři algoritmu DDPG řeší tento problém podobným způsobem jako V. Mnih [15], využívají *cílové sítě* upravené pro DDPG. Tyto sítě jsou vytvořeny jako kopie sítí kritika $Q'(s, a|\theta^{Q'})$ a aktéra $\mu'(s|\theta^{\mu'})$ a jsou využity k počítání předpokládaných hodnot (tedy hodnot, které jsou předávány Q funkci). Oproti algoritmu DQN jsou zde však vytvořeny sítě pro aktéra i kritika a aktualizují se jiným způsobem. Aktualizace vah těchto sítí probíhá zpožděným kopírováním naučených sítí podle parametru $\tau \ll 1$ následujícími pravidly:

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}, \quad (5.10)$$

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}. \quad (5.11)$$

Aplikace *cílových sítí* má za následek pomalejší učení, ovšem bez nich je problematické učit síť kritika bez divergencí.

Při práci ve spojitém prostoru akcí je nutný nový přístup k problému explorace. Ten je u algoritmu DDPG vyřešen vytvořením upravené strategie μ', která je získána přičtením šumové funkce N ke strategii aktéra:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + N. \quad (5.12)$$

Algoritmus 3 DDPG

Náhodná inicializace sítě aktéra $\mu(s|\theta^\mu)$ a sítě kritika $Q(s,a|\theta^Q)$
Inicializace sítí Q' a μ' pro výpočet výstupů, $\theta^{\mu'} \leftarrow \theta^\mu, \theta^{Q'} \leftarrow \theta^Q$
$R \leftarrow \theta$ inicializace bufferu
for episoda $= 1, ..., M$ **do**
    Inicializace náhodného procesu N pro exploraci prostoru akcí
    $s_1 \leftarrow$ počáteční stav
    for $t = 1, ..., T$ **do**
        Akce $a_t = \mu(s_t|\theta^\mu) + N_t$ vybraná podle sítě aktéra (se zašuměním)
        $s_{t+1}, r_t \leftarrow$ nový stav a odměna po vykonání akce a_t
        Uložení čtveřice (s_t, a_t, s_{t+1}, r_t) do bufferu R
        Vyber náhodnou dávku o velikosti N z bufferu R
        Spočítej požadovaný výstup pro $i = 1, ..., N$:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

        Aktualizuj síť kritika minimalizací chybové funkce:

$$L = \frac{1}{N}\sum_i^N \Big(y_i - Q(s_i, a_i|\theta^Q)\Big)^2$$

        Aktualizuj síť aktéra podle:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{i=1}^N \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

        Aktualizuj parametry pro výpočet výstupů:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

    end for
end for

Jako šumovou funkci autoři používají Uhlenbeckův-Ornsteinův proces [21]. Pomocí něj je možné šum usměrnit vhodným směrem pro konkrétní použité prostředí, což umožní rychlejší konvergenci sítě. Celý pseudokód DDPG je uveden v algoritmu 3.

5.5 Imitační učení

Metody imitačního učení mají za cíl napodobit lidské chování v daných úkolech. Agent využívá k učení demonstrační data z chování vzoru v různých situacích. Mapuje odpozorované chování na řídící akce.

Algoritmus 4 Dataset Aggregation

1: Inicializace datasetu $D \leftarrow \emptyset$
2: Inicializace $\hat{\pi}_i$ jakoukoliv strategií z množiny strategií Π
3: **for** $i = 1, ..., N$ **do**
4: Nechť $\pi_1 = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$
5: Získej dataset $D_i = \{(s, \pi^*(s))\}$ z navštívených stavů podle π_i a akcí získaných expertem
6: Agreguj datasety: $D \leftarrow D \cup D_i$
7: Natrénuj klasifikátor $\hat{\pi}_{i+1}$ na D
8: **end for**
9: **return** nejlepší $\hat{\pi}_i$ při validaci

Dataset Aggregation

Základní myšlenku aplikace konvolučních sítí a učení s učitelem na problém autonomního řízení lze popsat následovně: necháme zkušeného řidiče (lidského nebo AI) tzv. "experta"řídit vozidlo na různých tratích, přičemž bude zaznamenáván výstup senzorů vozidla (např palubní kamery) a akce provedené expertem v čase t. Tím je získán dataset $D(\pi)$, kde π je strategie experta. Tento dataset je poté použit k natrénování neuronové sítě, která ze senzorických vstupů odhaduje řídící povely. Jedná se tedy o *end-to-end* metodu a tzv. imitační učení.

Pokud je ovšem systém implementován tímto způsobem, nedosáhne dobrých výsledků. Projeví se chyba způsobená nepřesností sítě, díky které se agent dostane do stavů, které v datasetu nasbíraném expertem nejsou. Tím chyba začne rapidně narůstat a agent nemá možnost vrátit se k trajektorii experta.

Tomuto problému se v roce 2010 věnoval S. Ross [17]. Ve své práci představil algoritmus Dataset Aggregation (zkráceně DAGGER), upravující dataset přidáním agentem nově navštívených stavů a doučení modelu na tomto rozšířeném datasetu. Tento proces se iterativně provádí dokud agent nedosahuje požadovaných výsledků. Pseudokód této metody je uveden v algoritmu 4.

Kapitola 6

Prostředí a nástroje pro návrh navigačního systému

V této kapitole jsou popsány prostředky pro návrhu systému autonomního řízení. Jsou zde rozebrány techniky a různé možnosti implementace, které lze v tomto systému využít. Také je představen simulátor TORCS, který jsem zvolil pro sběr vstupních dat a testování sytému. Dále jsou představeny knihovny pro hluboké učení a implementaci neuronových sítí, které moje práce využívá.

6.1 Simulační prostředí a vstupní data

Klasické neuronové sítě se učí metodou s učitelem, pro kterou je nutné předem připravit soubor vstupních dat, podle kterých se systém učí. Běžnou metodou získání souboru vstupních dat (datasetu) pro autonomní řízení je zaznamenávání informací během jízdy s lidským řidičem. Těmito informacemi jsou především obraz z jedné nebo více kamer umístěných na vozidle, ale mohou být zaznamenávána i další data z vozidla i okolí. Například to může být aktuální rychlost vozidla, stupeň řazení, úhel natočení vozidla ku vozovce atd.

Sběr dat pro učení rozsáhlých systémů a zachycení různých situací, které mohou nastat, je velmi časově náročný. Pro získání většího množství dat je možné surová data předupravit. Například zrcadlením obrazu z palubní kamery se mohou získat validní data až dvojnásobného objemu.

Jelikož ve své práci využívám metod zpětnovazebního učení, které potřebují velké množství iterací k naučení modelu, je využití simulátoru nezbytné. V reálném prostředí by tyto metody zabraly velké množství času a během učení by snadno mohlo dojít k poškození hardwaru (např. modelu auta).

Simulátor TORCS

Jako prostředí pro implementovaný systém jsem zvolil závodní simulátor TORCS (The Open Racing Car Simulator), který je dostupný pod licencí open source. TORCS slouží také jako závodní počítačová hra, ale především se využívá jako vývojová platforma pro systémy umělé inteligence a autonomního řízení. Je spustitelný na systémech Linuxu, FreeBSD, OpenSolaris, MacOSX a Windows. Pohled řidiče v TORCS je zobrazen na obrázku 6.1.

Hlavním důvodem zvolení simulátoru TORCS je dostupnost aplikačního rozhraní, díky kterému lze programově komunikovat se simulátorem a získávat výstupy potřebné pro učení neuronové sítě. Kromě obrazu simulátor poskytuje další parametry, které je možné využít

Obrázek 6.1: Obrazovka simulátoru TORCS z pohledu řidiče

pro systém autonomního řízení. Pro ovládání vozidla mohou být využity efektory, ovládající akceleraci, brzdění a zatáčení. Kompletní seznam senzorických informací o vozidle, dostupných v simulátoru TORCS, je uveden v tabulce 6.1. Seznam efektorů je popsaný v tabulce 6.2. Díky velkému počtu dostupných tratí pro tento simulátor je možné učit systém pouze za pomoci některých z nich a zjišťovat schopnost systému se přizpůsobit na jiné tratě, které v učícím datasetu nebyly použity.

Tento simulátor již byl použit v předchozích pracích věnovaných autonomnímu řízení za pomoci strojového učení [4, 9]. Již několik let se TORCS používá také pro soutěže strojově ovládaných aut, jako je TORCS Endurance World Championship[13].

6.2 Nástroje pro implementaci

Pro implementaci neuronových sítí existuje řada rámců (tzv. *frameworků*), které usnadňují jejich sestavování a úpravy. V nich jsou základní typy neuronových sítí již naimplementovány. Autor si tak může dotvořit vlastní síť propojením předchystaných vrstev. *Frameworky* jsou naprogramovány v různých jazycích a často umožňují provádět výpočet paralelně na grafickém procesoru, což jej značně urychluje. Mezi nejznámější *frameworky* patří např. Neuroph, Caffe, TensorFlow a Keras. Posledně jmenované jsem zvolil pro implementaci neuronových sítí ve své práci.

TensorFlow TensorFlow je výpočetní knihovna pro *dataflow* programování a práci s modely strojového učení založenými na neuronových sítích. Byla vyvinutá skupinou Google Brain Team pro interní použití společností Google a v roce 2015 uvolněna pro veřejnost pod licencí *open source*.

TensorFlow umožňuje paralelní výpočty s využitím grafické karty, což přináší podstatné zrychlení výpočtů modelů neuronových sítí. Tuto knihovnu jsem použil pro výpočty spojené s neuronovými sítěmi a zpětnovazebním učením.

Keras Pro implementaci jsem využil aplikační prostředí Keras[5]. Jedná se o knihovnu pro práci s neuronovými sítěmi distribuovanou pod licencí *open source*, která byla vyvinuta na

Název	Popis
angle	Úhel mezi směrem vozidla a osou trati
curLapTIme	Čas, který uběhl v současném kole
damage	Stupeň poškození auta
distFromStartLine	Vzdálenost od startovní čáry
distRaced	Ujetá vzdálenost od počátku závodu
fuel	Palivo v nádrži
gear	Aktuální převod
lastLapTime	Čas ujetí posledního kola
opponents	Vektor 36 senzorů, které detekují soupeře do vzdálenosti 100m, každý senzor monitoruje 10 stupňů kolem vozidla
racePos	Pozice v závodu
rpm	Počet otáček motoru za minutu
speedX	Rychlost vozidla ve směru jeho podélné osy
speedY	Rychlost vozidla ve směru jeho příčné osy
track	Vektor 19 senzorů, které detekují vzdálenost k okrajům tratě, každý senzor monitoruje 10 stupňů kolem vozidla
trackPos	Vzdálenost vozidla od středu vozovky
wheelSpinVel	Vektor 4 senzorů reprezentující rychlost rotace kol

Tabulka 6.1: Dostupné senzory v TORCS

Název	Popis
accel	Plynový pedál
brake	Brzdový pedál
gear	Převodový stupeň
steering	Úhel natočení volantu
meta	Požadavek serveru restartovat závod

Tabulka 6.2: Dostupné efektory pro řízení v TORCS

universitě MIT. Keras je napsán v jazyce python a funguje jako nadstavba nad knihovnami pro hluboké učení TensorFlow, CNTK a Theano. Díky tomu je možné programovat modely neuronových sítí bez nutnosti podrobné znalosti knihovny, nad kterou pracuje. Využitou knihovnu nižší úrovně (tzv. *backend*) lze také v případě potřeby vyměnit pouze s minimálním zásahem do vlastního kódu.

Keras umožňuje psát tzv. sekvenční model. Při něm programátor může využít předdefinované typy vrstev neuronových sítí, u nichž jen určí parametry a aktivační funkci. Takto definované vrstvy se skládají za sebe v požadovaném pořadí. Posledním nutným krokem před kompilací modelu je určení chybové funkce a optimalizačního algoritmu, které lze opět zvolit z předdefinovaných možností. Tímto způsobem je vytvořen model neuronové sítě, který je snadno čitelný a přenosný mezi různými knihovnami pro hluboké učení.

V této práci Keras využívám jako nadstavbu nad knihovnou TensorFlow. Výhodou je, že mi umožnil definovat různé typy modelů jednotným způsobem.

AI gym Knihovna AI gym je vytvořena společností Open AI a je určena k učení, testování a porovnávání algoritmů umělé inteligence, jako je např. hluboké zpětnovazební učení. K vytvořeným modelům poskytuje různá prostředí pro učení agenta, jako jsou hry z počítače Atari nebo 3D fyzikální modely. Je díky ní možné vyvinout učící model, který komunikuje pouze s knihovními metodami a není závislý na konkrétním prostředí agenta. Je tedy snadné takovýto model bez nutnosti změn učit a testovat v různých prostředích.

Kihovna AI gym sice neobsahuje prostředí k řízení vozidla, ale využívá ji nadstavba Gym TORCS, kterou využívám ke komunikaci systému se simulátorem TORCS. Komunikace s Gym TORCS pak může být stejná jako při komunikaci s AI gym.

6.3 Propojení simulátoru s AI modelem

Závodní simulátor TORCS je využíván k závodům AI agentů, pro které byl upraven, aby fungoval jako klient-server aplikace. Díky tomu bylo možné, aby jednotliví agenti byli spouštěni mimo hlavní aplikaci simulátoru a nemohli jej nijak nepoctivě ovlivňovat. Více o těchto závodech a k nim vyvinuté architektuře je možné zjistit ve článku o TORCS Racing Championship [13].

Na takto upravený TORCS je napojena knihovna *Gym TORCS*, která umožňuje využít TORCS jako prostředí pro AI model vytvořený v jazyce python (např. pomocí knihovny TensorFlow) stejným způsobem, jako framework *AI gym*. V každé iteraci algoritmu tato knihovna získá tzv. *observaci* obsahující data ze všech dostupných senzorů. Nasnímaná data jsou zpracována a na jejich základě neuronová síť spočítá vhodné hodnoty efektorů, které jsou knihovní funkcí předány zpět do prostředí TORCS. Zpracováním dat před vstupem do neuronové sítě je myšlena především filtrace užitečných senzorů a jejich normalizace do stejného oboru hodnot (v tomto případě $\langle -1, 1 \rangle$).

Filtrace senzorů je nutná, jelikož simulátor TORCS dokáže předávat spoustu senzorických informací (viz tabulka 6.1), z kterých je vhodné pro učení použít jen určitou podmnožinu. Například není vhodné, aby výpočetní model měl informaci o vzdálenosti auta od startovní čáry, jelikož tak by se mohl naučit specifika konkrétní trati (např. na pátém kilometru zatoč vlevo) a pro ostatní tratě by se stal nepoužitelný.

Kapitola 7

Návrh implementovaných modelů

V této kapitole jsou popsány implementované modely pro řízení závodních aut v simulátoru TORCS. Cílem bylo naučit model projet s autem co nejdelší úsek závodní trati, zvládnout jízdu na rozdílných tratích a dosáhnout co nejlepšího času.

Pro implementaci jsem zvolil jeden model založený na imitačním učení, podle algoritmu DAGGER, který se učí podle datasetu vytvořeného expertem.

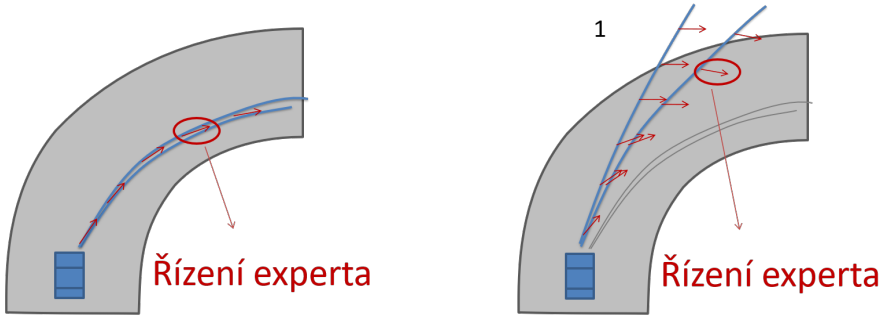
Druhý zvolený model je založený na algoritmu DDPG, který se řadí k nejpokročilejším metodám hlubokého zpětnovazebního učení. Hlavním důvodem pro tuto volbu je jeho schopnost generovat akce ve spojitém oboru hodnot, vhodné pro ovládání volantu a pedálů auta.

7.1 Model imitačního učení

Mým původním záměrem bylo použít konvoluční neuronovou síť k přímému namapování obrazových snímků z pohledu řidiče na řídící povely a podle nich auto navigovat. Podle dostupných zdrojů [**?**] však tento přístup není vhodný pro sekvenční problémy, kde budoucí stavy závisí na předchozích akcích, jako je řízení vozidel. Proto jsem využil algoritmus DAGGER, popsaný v článku S. Rosse [17], který je klasickému *end-to-end* přístupu nejblíže.

V algoritmu jsem jako klasifikátor použil konvoluční síť, schéma sítě je uvedeno na obrázku7.2. Na vstupu síť dostává obrazové snímky ze simulátoru v rozlišení 64×64. Na jejím výstupu je jediná spojitá hodnota v oboru $\langle -1, 1 \rangle$, která určuje natočení volantu. Pedály plynu a brzdy jsou v tomto případě nastavovány automaticky pro udržení zvolené konstantní rychlosti. Model používá jako chybovou funkci Mean Squared Error a pro optimalizaci využívá algoritmus Adam.

Pro rychlejší začátek učení nechávám před spuštěním samotného algoritmu DAGGER síť učit na předem nasbíraném datasetu. Tento dataset může být nasbírán ručním řízením auta v simulátoru. Jedná se však o velice zdlouhavý proces, který navíc pro kvalitní výsledky vyžaduje vhodný ovladač (volant) a zručnost. Proto jsem jako experta využil řídící program (tzv. bot), který je dostupný v rámci simulátoru TORCS. Pomocí něj také ohodnocuji stavy, ve kterých se ocitne agent řízený neuronovou sítí. Vizualizace těchto stavů je uvedena na obrázku 7.1. Tímto způsobem získám stav (snímek) i jeho ohodnocení, potřebné k vytvoření nového datasetu. Tento nový dataset se konkatenuje s původním datasetem a síť je pomocí něj znovu učena. Nové učení sítě začne ve chvíli, kdy agent řízený sítí naboural. Po skončení tohoto učení agent začne nový závod. V okamžiku, kdy je agent schopen jet dostatečně dlouhou dobu bez kolizí je algoritmus ukončen.



Obrázek 7.1: Nové stavy rozšiřující dataset v algoritmu DAGGER.

7.2 Model zpětnovazebního učení

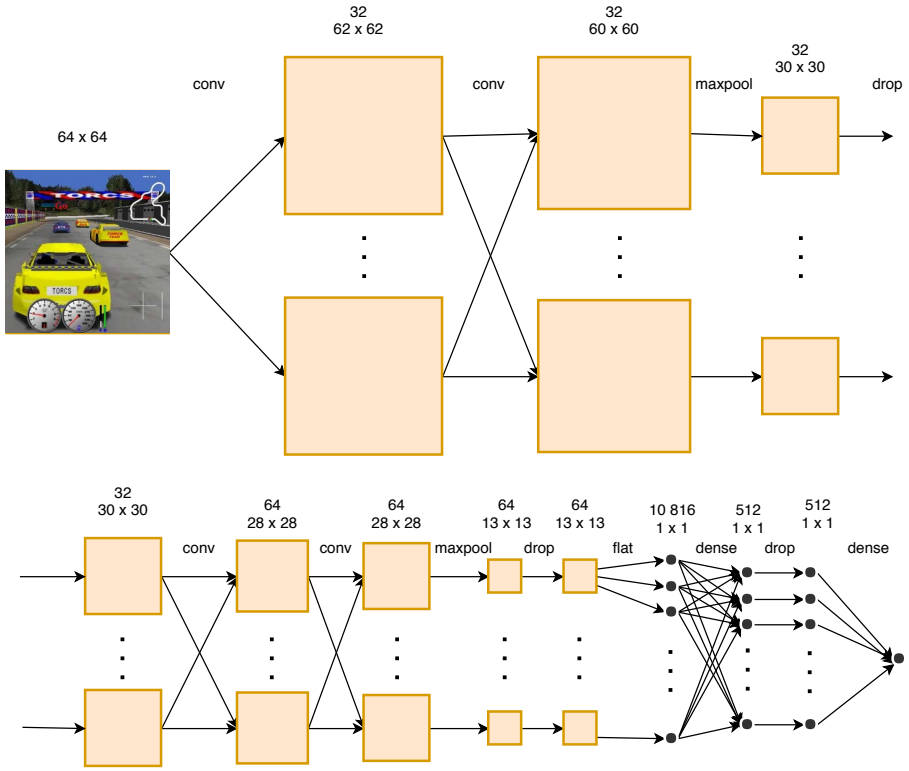
Dalším krokem mé práce bylo rozšíření implementovaného systému o prvky zpětnovazebního učení za účelem dosažení lepších výsledků a možnosti učit se vlastními chybami bez nutnosti použít nasbíraný dataset. Algoritmus DDPG jsem zvolil pro jeho schopnost generovat akce na spojité množině hodnot. Tím je schopný dosáhnout vyšší přesnosti oproti algoritmům s diskrétními akcemi, jako je např. DQN. Akce agenta při řízení vozidla představují ovládání volantu a pedálů a spojité hodnoty pro přesnost rychlé jízdy jsou zde žádané.

Jako vstup do systému jsem využil výstupy ze senzorů auta poskytované simulátorem. Z celkového seznamu senzorů (viz tabulka 6.1) je vhodné vybrat pouze některé, jelikož ne každá informace je pro učení řízení podstatná a zapojení některých informací do procesu učení může být i škodlivé. Jako relevantní jsem určil následující hodnoty: úhel mezi směrem vozidla a osou trati, rychlosti vozidla ve směrech podélné a příčné osy auta x a y, vzdálenost od středu vozovky, vzdálenosti od ostatních aut a vzdálenosti od okrajů vozovky ve všech dostupných úhlech (viz obrázek 7.3). Souhrn těchto hodnot tvoří jeden stav prostředí, ve kterém se agent pohybuje.

Architektura sítě Pro aproximaci funkcí aktéra $\mu(s)$ a kritika $Q(s, a)$ v algoritmu DDPG jsem použil plně propojené neuronové sítě.

Síť aktéra dostává na vstup vektor vybraných senzorických dat, tvořící celkem 29 vstupních hodnot. Na výstupu aktér generuje hodnoty akcelerace a brzdy, které jsou v oboru $\langle 0, 1\rangle$ a hodnotu pro zatáčení v oboru $\langle -1, 1\rangle$. Výstupní neurony pro brzdu a plyn využívají jako aktivační funkci *sigmoidu* a výstupní neuron pro zatáčení používá jako aktivační funkci *hyperbolický tangens*. Důvod použití rozdílných aktivačních funkcí pro výstupní neurony je ten, aby obor hodnot funkce odpovídal rozsahu odpovídající výstupní hodnoty, tedy $\langle 0, 1\rangle$ pro pedály a $\langle -1, 1\rangle$ pro volant. Síť obsahuje 2 skryté vrstvy, z nichž první tvoří 300 a druhou 600 neuronů. Neurony ve skrytých vrstvách používají aktivační funkci $ReLU$. Vizualizace architektury této sítě je zobrazena na obrázku 7.4.

Síť kritika má na vstupní vrstvu přiveden vektor senzorických dat, představující stav prostředí s. Funkce $Q(s, a)$ však potřebuje na vstupu i akce. Ty jsou přivedeny až na druhou skrytou vrstvu sítě. První skrytá vrstva tedy zpracovává jen stav s a celý vstup sítě je zpracováván až od druhé skryté vrstvy. Tato architektura byla doporučená v článku o algoritmu DDPG [2]. Celkem síť obsahuje 3 skryté vrstvy, z nichž první a třetí používá aktivační funkci $ReLU$. Druhá aktivační vrstva, která slouží jako spojení vstupů akcí a stavu, počítá sumu lineárních aktivací z první skryté vrstvy a vstupu akcí. Architektura sítě je zobrazena na obrázku 7.5.



Obrázek 7.2: Konvoluční síť použitá v implementaci algoritmu DAGGER.

Funkce odměn T. P. Lillicrap [2] v experimentu s řízením vozidla počítá funkci odměn jako rychlost auta ve směru osy trati v každém časovém kroku t. Celková odměna R je tedy dána rovnicí:

$$R = \sum_t V_{x,t} \cos \alpha_t, \tag{7.1}$$

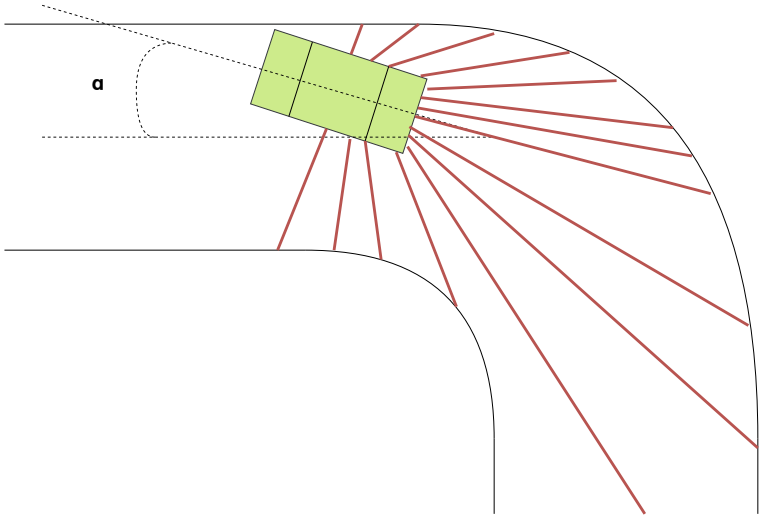
kde α_t je úhel mezi osou trati a podélnou osou auta a $V_{x,t}$ značí rychlost auta ve směru jeho osy v každém okamžiku t. A. Ganesh[1] tuto funkci upravil o dvě záporné složky. První je určena rychlostí auta ve směru kolmém k ose trati, což má za cíl penalizovat agenta za smyky a "křivou"jízdu. Druhá záporná složka je definována vzdáleností auta od středu trati, což má za cíl minimalizovat možnost vyjetí auta z trati tím, že se bude snažit držet uprostřed. Tato upravená funkce odměn lze zapsat jako:

$$R = \sum_t |V_{x,t} \cos \alpha_t| - |V_{x,t} \sin \alpha_t| - |V_{x,t}||x_t - \mu_t|. \tag{7.2}$$

Ganesh také uvádí, že by bylo vhodné prozkoumat i jiné funkce odměn. V této práci testuji obě zmíněné funkce odměn a přináším jednu vlastní. Při projíždění zatáček je ideální stopa často velmi blízko okraje trati, z toho důvodu se domnívám, že funkce odměn by se neměla pokoušet držet auto co nejblíže středu. To může vést k pomalejšímu projetí okruhu, ale také zvýšit šanci vyjetí z trati v zatáčkách. Proto testuji funkci definovanou následovně:

$$R = \sum_t |V_{x,t} \cos \alpha_t| - |V_{x,t} \sin \alpha_t|. \tag{7.3}$$

Nutno dodat, že v případě kolize auta je tato funkce nahrazena pevnou zápornou hodnotou (v základní verzi algoritmu používám $R = -1$), aby se agent pokoušel kolizím vyvarovat.



Obrázek 7.3: Využívané informace o stavu auta z TORCS. Červené čáry označují vektor senzorů detekující okraje trati. Úhel α mezi autem a vozovkou je použit k výpočtu funkce odměn.

Prostředky pro zvýšení stability učení Po vzoru článku o DDPG[2] používám cílové sítě a *replay buffer*, jako prostředky ke zvýšení stability učení. Pro každou síť vytvářím její kopii nazvanou cílová (*target*) síť, které je použita k výpočtu Q-funkce a aktualizuje se podle pravidla:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'. \tag{7.4}$$

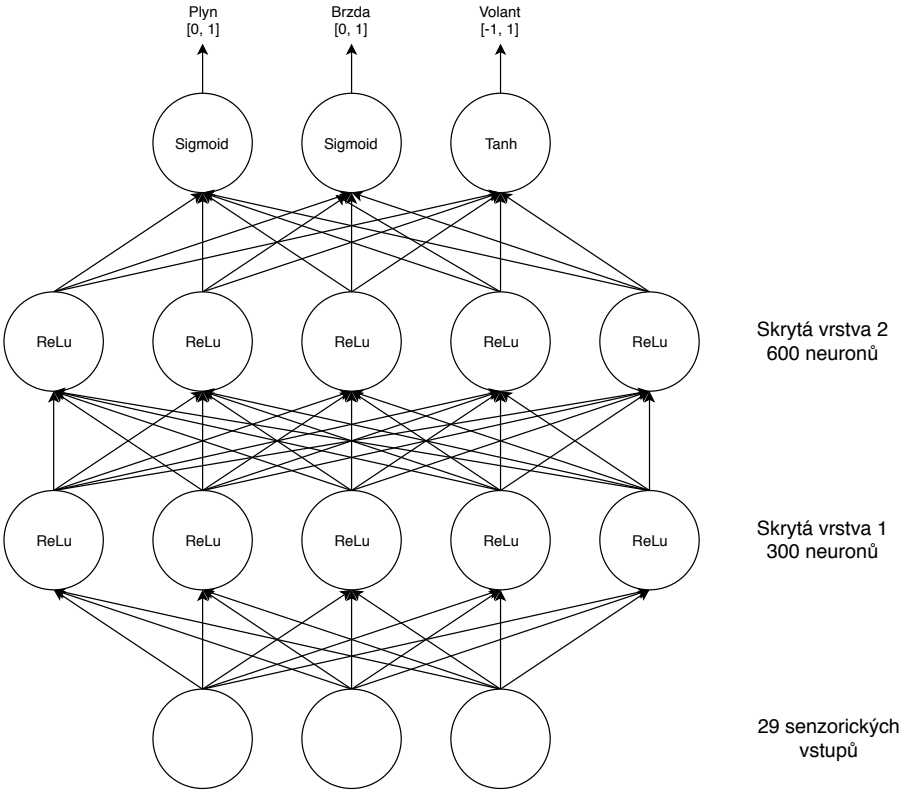
Dalším implementovaným prostředkem je *replay buffer* (popsán v sekci 5.3), ten je nastaven na velikost 100000 záznamů typu $E = (s, a, r, s')$ a podle uniformní náhodné distribuce z něj vybírám dávku (*batch*) o velikosti 32 záznamů, která je použita k jedné iteraci učení sítí.

Způsob explorace Pro spojitý prostor akcí nelze použít explorační metody, jako je ϵ-*greedy*. V té se s určitou pravděpodobností provede náhodná akce. V tomto případě však není možné snadno vybrat jednu akci a z množiny akcí A. Pokud by se taková akce vygenerovala určením hodnot plynu, brzdy a zatočení volantu podle náhodného rozložení, vznikaly by nesmyslné kombinace jako je sešlápnutí pedálů brzdy a plynu zároveň. Dalším problémem je, že zvolení takovéto akce v průběhu jízdy může silně negativně ovlivnit nadcházející stavy, a to i v případě, že akce v těchto stavech už jsou rozumné (např. strhnutí řízení i na krátký moment vede k havárii).

Jako způsob explorace tedy k jednotlivým hodnotám plynu, brzdy a zatáčení připočítávám šum. Ten je podle článku Lillicrapa[2] počítán jako Ornsteinův-Uhlenbeckův proces, jehož parametry se pro jednotlivé položky akce liší. Především šum přidaný k akci brzdy se snaží celkovou hodnotu směřovat k nule, aby se auto vůbec pohybovalo a ostatní akce měly vliv. Rovnice výpočtu je zapsána jako:

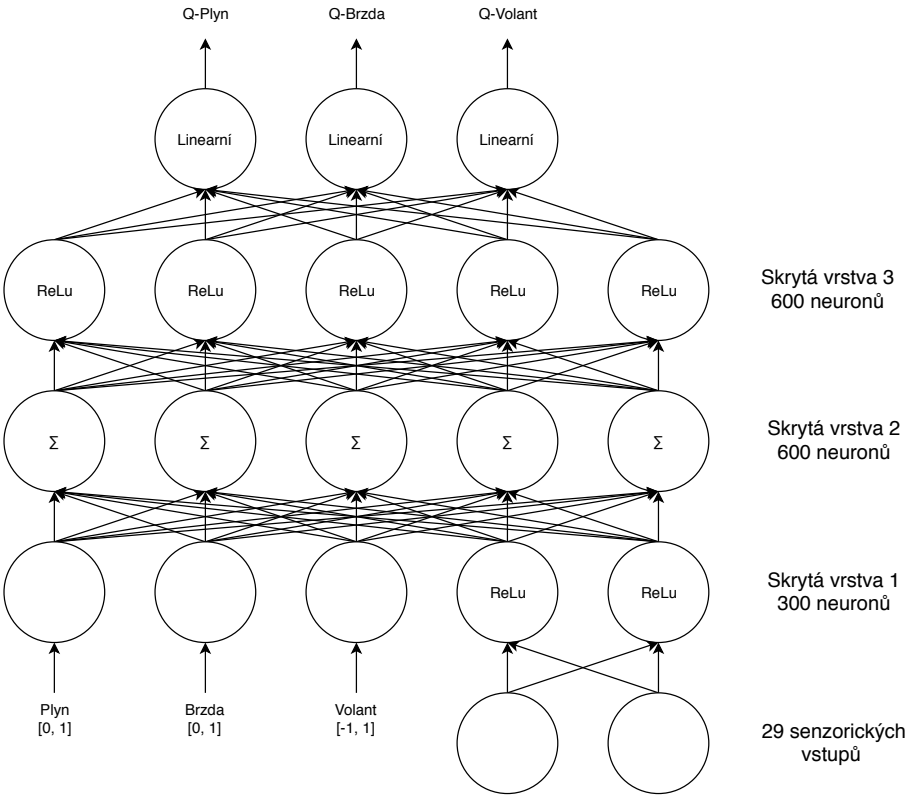
$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t, \tag{7.5}$$

kde $\theta > 0$, μ a $\sigma > 0$ jsou nastavitelné parametry a dW je Wienerův stochastický proces, který je zde nahrazen náhodnou hodnotou vybranou uniformním rozložením z množiny $\langle 0, 1\rangle$.



Obrázek 7.4: Znázornění architektury sítě kritika, navrženou pro metodu DDPG.

Jak model postupem učení vybírá stále vhodnější akce, je vliv šumu dobré v čase tlumit. Proto definuji parametr *explore* a v každém kroku učení vliv šumu snižuji o $\frac{1}{explore}$. Po počtu kroků rovném parametru *explore* tedy šum zcela vymizí a agent se rozhoduje pouze na základě akcí daných naučeným modelem.



Obrázek 7.5: Architektura sítě kritika pro metodu DDPG.

Kapitola 8

Experimenty

V této kapitole jsou popsány experimenty, provedené na navržených modelech. Výsledky experimentů jsou vyhodnoceny a jsou diskutovány možnosti budoucího vývoje.

Pro realizaci experimentů jsem použil počítač s procesorem i5-750 @ 2,67 GHz×4. Pro urychlení výpočtů byl počítač doplněn o grafickou kartu NVidia GTX 770. Experimenty proběhly pod operačním systémem Ubuntu 16.04. Pro implementaci neuronových sítí jsem využil knihovny TensorFlow 1.5 a Keras 2.1. Jako simulační prostředí jsem využil upravenou verzi simulátoru TORCS [**?**]. Vlastní programy jsou napsány v jazyce Python.

V experimentech používám termíny krok a epizoda algoritmu, jejich vysvětlení je následující:

- Krok algoritmu DAGGER - výpočet akce pomocí dopředného průchodu neuronovou sítí, jeden krok v simulátoru TORCS následovaný výpočtem experta pro daný stav a přidání záznamu do nového datasetu
- Epizoda algoritmu DAGGER - algoritmus iteruje po krocích dokud se agent v simulátoru TORCS nevyboural nebo nedokončí 2 kola závodu, poté se nově nasbíraný dataset konkatenuje s původním a použije se k učení neuronové sítě s konstantním počtem iterací
- Krok algoritmu DDPG - výpočet akce pomocí dopředného průchodu neuronovou sítí, jeden krok v simulátoru TORCS následovaný uložením záznamu do *replay* bufferu, získání dávky z bufferu a provedení jedné iterace učení neuronových sítí.
- Epizoda algoritmu DDPG - algoritmus iteruje po krocích dokud se agent v simulátoru TORCS nevyboural nebo nedokončí 2 kola závodu, poté je závod restartován

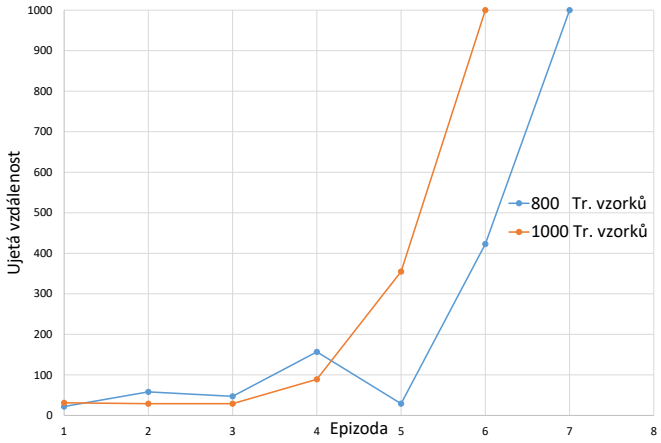
Experimenty probíhaly na šesti vybraných tratí z TORCS, jejich seznam je uveden v tabulce 8.1.

8.1 Testování modelu DAGGER

Model učený pomocí algoritmu DAGGER pro učení s učitelem byl testován v jízdě podle obrazových snímků s rozlišením 64×64. Učení jsem považoval za dokončené, pokud byl agent schopen projet celou tratí bez havárie. Rychlost agenta jsem nastavil na pevnou hodnotu 60km/h. Projetí jednoho kola trati odpovídá cca. 1000 simulačním krokům. Po každé epizodě algoritmu bylo provedeno učení sítě po 100 epoch (1 epocha značí jednu iteraci na celém trénovacím datasetu).

Označení tratě	název v TORCS
Trať 1	Forza
Trať 2	CG Track 2
Trať 3	CG Track 3
Trať 4	Olethros Road 1
Trať 5	Aalborg
Trať 6	Alpine 2

Tabulka 8.1: Označení použitých tratí.



Obrázek 8.1: Průběh učení modelu DAGGER. Ujetá vzdálenost značí počet ujetých simulačních kroků, hodnota 1000 odpovídá ujetí celého kola tratě.

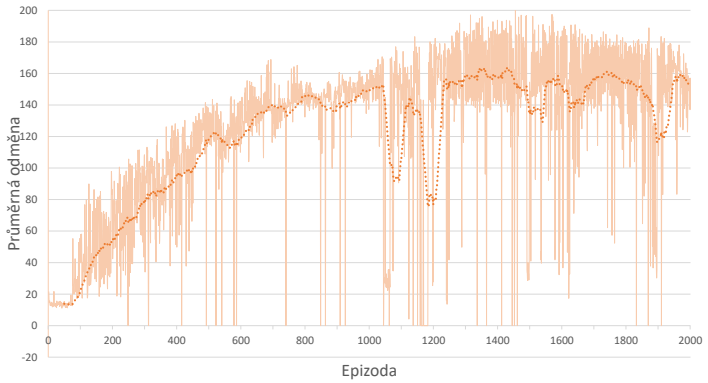
Testoval jsem vliv velikosti počáteční trénovací množiny na rychlost učení. Velikosti množin jsem nastavil na 800 a 1000 vzorků. Pro každou z těchto hodnot jsem provedl 5 pokusů. Při menší velikosti trénovací množiny 800 vzorků ujede model požadovaný počet kroků (1000) v průměru za 7 epizod učení. Pokud se tato množina navýší na 1000 vzorků, požadovanou vzdálenost model dosáhne již po šesti epizodách učení. Ujeté počty kroků pro trénovací množiny o velikosti 800 a 1000 vzorků jsou uvedeny v grafu 8.1.

8.2 Testování modelu DDPG

Experimenty s funkcí odměn Model jsem učil se třemi různými funkcemi odměn, popsanými v sekci 7.2. Cílem bylo zjistit, se kterou funkcí odměn dokáže agent jet stabilněji nebo rychleji. Také může být zajímavé zjistit, jestli se s jednodušší funkcí odměn dokáže agent dříve dostat do stavu, kdy je schopen projet celou trať. Každé učení sítě probíhalo 2000 epizod, přičemž epizodou je myšlen jeden běh simulátoru TORCS dokud agent úspěšně nezajel 2 kola závodu nebo nehavaroval. Z důvodu rozdílné délky epizod je v grafech uvedená průměrná odměna za epizodu, která je spočítána jako suma odměn v každém kroku simulace, dělená počtem kroků v dané epizodě. Varianty funkcí odměn jsou uvedeny v tabulce 8.2.

Označení varianty	funkce odměn
A	$R = \sum_t V_{x,t} \cos \alpha_t$
B	$R = \sum_t \lvert V_{x,t} \cos \alpha_t \rvert - \lvert V_{x,t} \sin \alpha_t \rvert$
C	$R = \sum_t \lvert V_{x,t} \cos \alpha_t \rvert - \lvert V_{x,t} \sin \alpha_t \rvert - \lvert V_{x,t} \rvert \lvert x_t - \mu_t \rvert$

Tabulka 8.2: Označení funkcí odměn pro experimenty.



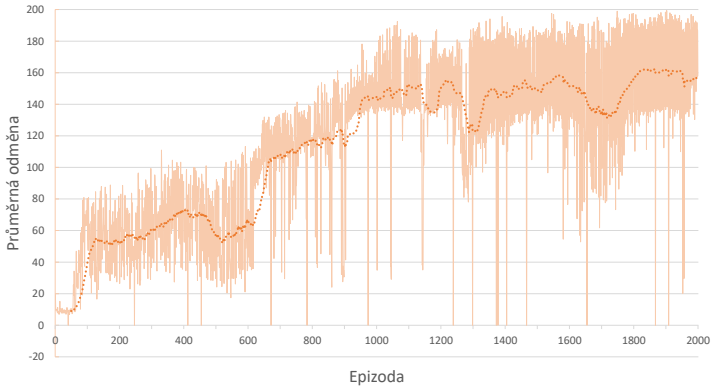
Obrázek 8.2: Průběh učení s funkcí odměn A.

Porovnání ujeté vzdálenosti a rychlosti J. Koutník ve svém článku [9] z roku 2013 učí model neuronové sítě pomocí kombinace evolučních a zpětnovazebních algoritmů. Tento model testuje také v prostředí TORCS. Porovnává jej s pevně naprogramovanými (*hard-coded*) *boty* dostupnými v simulátoru TORCS. Porovnává vzdálenost, jakou je agent schopen ujet bez nárazu a maximální rychlost během jízdy. V tabulce 8.3 do tohoto porovnání zařazuji i svůj model učený algoritmem DDPG. Jízda probíhala na trati 5, použitá síť nebyla na této trati trénována. Výsledky jsou popsány v tabulce 8.3.

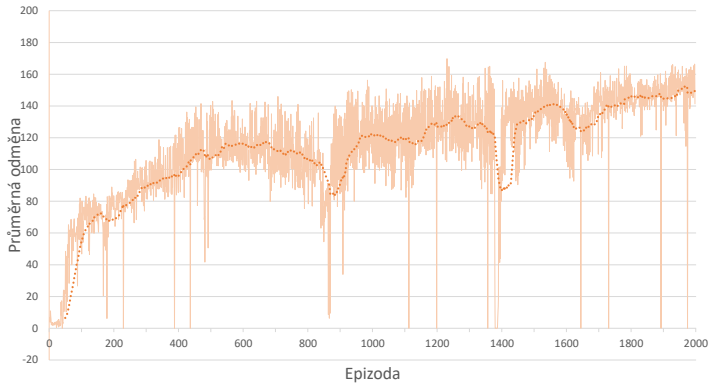
Řidič	$d[m]$	$V_{max}[km/h]$
olethros A	570	147
bt	613	141
berniw	624	149
tita	657	150
inferno	682	150
RNN	625	144
DDPG B	1443	203

Tabulka 8.3: Ujetá vzdálenost na složitější trati 5, která nebyla použita pro trénování. RNN značí model J. Koutníka, DDPG B je vlastní model s funkcí odměn B. Ostatní řidiči jsou boti dostupní v TORCS.

Vliv rychlosti simulace Závody v TORCS v běžném režimu běží v reálném čase, aby bylo auto možné člověkem řídit. Rychlost lze až 4× zrychlit, což je velmi vhodné pro učení zpětnovazebního algoritmu, které může trvat desítky hodin. Při běžné rychlosti mi experiment o 2000 epizodách trval cca. 40 hodin.



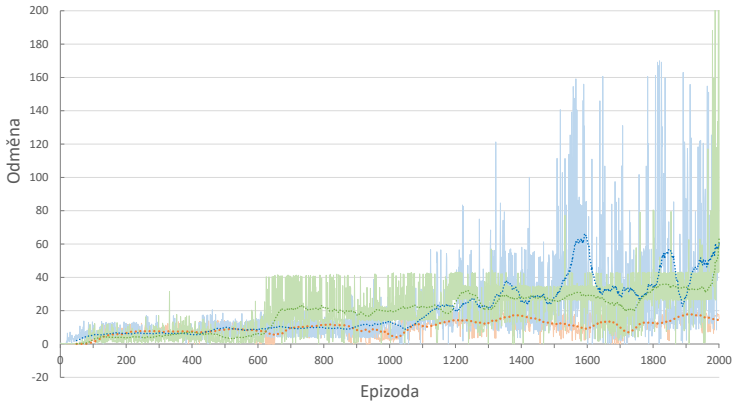
Obrázek 8.3: Průběh učení s funkcí odměn B.



Obrázek 8.4: Průběh učení s funkcí odměn C.

Proto jsem testoval běh při 4× zrychlené simulaci, ovšem výsledky se i při zachování stejných parametrů zhoršily. Komunikace se serverem TORCS při běžné rychlosti funguje na frekvenci 5Hz, při čtyřnásobné rychlosti tedy 20Hz. Podávat akce o této frekvenci by neměl být pro učící algoritmus s plně propojenou sítí na použitém počítači problém a serverová komunikace žádné zpoždění nezaznamenala. Při pozorování jízdy však auto nestíhalo včas reagovat na ostřejší zatáčky a ve vysokých rychlostech často docházelo k vyjetí z trati a tedy i nízkým odměnám. Graf z experimentu s různou rychlostí simulace je uveden na obrázku 8.5.

Z grafu je patrné, že při zrychlení 4× jsou odměny získané z učení opravdu nižší. Rozdíl je vidět obzvlášť v pozdějších epizodách, kde model učený s pomalejší rychlostí simulátoru už zvládá obstojně jezdit. U dvojnásobné rychlosti simulátoru oproti standardní jsem velký rozdíl nezaznamenal a pro učení tedy doporučuji rychlost simulace 2×. Jako důvod tohoto problému bych označil synchronní komunikaci TORCS s propojovacími knihovnami *gym torcs* a algoritmem učení.



Obrázek 8.5: Vliv zrychlení simulace na získané odměny. Zeleně je znázorněn průběh učení bez zrychlení, modře se zrychlením 2× a oranžově se zrychlením 4×.

8.3 Vyhodnocení výsledků experimentů

Z testování vyplynulo, že model naučený pomocí metodou DAGGER je schopen autonomně jezdit po trati po cca. šesti epizodách algoritmu. K rychlejšímu učení může pomoci navýšení počtu vzorků počátečního datasetu. U této metody byla použita konstantní rychlost jízdy, jelikož algoritmus se učí pouze podle jednotlivých obrazových snímků a nelze tedy detekovat aktuální rychlost. Výhodou je, že model je schopen po relativně nízkém počtu iterací učení zvládnout jízdu bez havárií.

Větší důraz byl kladen na algoritmus DDPG, který využívá modernější metody přístupu ke strojovému učení. Model naučený tímto algoritmem byl schopen projet jakoukoli trať, na které probíhalo učení a některé snadnější tratě z těch ostatních. Dosahoval při tom vysokých rychlostí a předčil většinu srovnávaných soupeřů, tvořených pevně naprogramovanými kontroléry dostupnými v simulátoru TORCS. Agent zvládal i úkony jako je "zotavit"se s mírné kolize se soupeřem, vyhnutí soupeři a brzdění před zatáčkami. Vysoká rychlost působila problém při jízdě po složitějších tratích, na kterých se model neučil, jelikož v ní často nezvládl ostré zatáčky a vyjel z trati.

Vyzkoušel jsem 3 funkce odměn, které ovlivňují agentovo chování. V souladu s cílem dosažení co nejlepších časů byla hlavní složkou odměn rychlost ve směru osy trati. Při penalizaci odchylky auta od středu trati je jízda stabilnější, jelikož se sníží tendence auta vyjet z trati. Při závodech ve vysokých rychlostech to ovšem může způsobit horší čas, jelikož auto nejede ideální stopou. Jako kompromis je možné penalizovat pouze rychlost v ose kolmé k vozovce.

Experimenty byl také objeven technický problém, kdy se při zvýšení rychlosti simulace zhoršily výsledky.

Možnosti budoucího vývoje V budoucím vývoji by bylo vhodné u metody DDPG zakomponovat do vstupů sítě i obrazové vstupy. To by usnadnilo použití podobného systému v reálném světě, kde je obtížné získat tak přesné výstupy senzorů, jako podává simulátor. Nicméně by bylo zajímavé aplikovat tento naučený model na model auta vybavený lidarem, či jinými senzory, které by replikovaly výstupy senzorů ze simulátoru. Takovým pokusem by se dalo otestovat, zda tento přístup může mít větší uplatnění v praxi.

Kapitola 9

Závěr

V rámci diplomové práce byl vytvořen přehled o metodách strojového učení použitelných k autonomnímu řízení vozidel.

Hlavním cílem bylo vytvořit kontrolér, schopný jezdit a závodit v simulačním prostředí. Jako zdroj vstupních dat a testovací prostředí jsem zvolil simulátor TORCS, dostupný pod otevřenou licencí a použitý v několika souvisejících pracích. To mi umožnilo snadnější přístup k datům a široké možnosti srovnání výsledků. Pro implementaci neuronových sítí jsem zvolil knihovny TensorFlow a Keras.

Pro tento účel jsem vytvořil dva typy modelů. První z nich je založen na imitačním učení, využívá algoritmus Dataset Aggregation a je implementován pomocí konvoluční neuronové sítě. Tento model byl otestován na šesti tratích, které je schopen konstantní rychlostí projet.

Druhý model je založen na zpětnovazebním učení a jako vstupy využívá data ze senzorů vozidla. Jako učící algoritmus je použit Deep Deterministic Policy Gradient. Tento model je schopný dosahovat velmi dobrých výsledků v porovnání s jinými dostupnými kontroléry. Je schopen se přizpůsobit na většinu jednodušších tratí dostupných v simulátoru TORCS. Problematická pro něj je reakce na náhlé změny trati, obzvláště ve vyšších rychlostech.

Literatura

[1] April Yu; Raphael Palefsky-Smith; Rishi Bedi: Deep Reinforcement Learning for Simulated Autonomous Vehicle Control. 2016: s. 1–6.

[2] Bengio, Y.: Continuous control with deep reinforcement learning. *Foundations and Trends® in Machine Learning*, ročník 2, č. 1, 2009: s. 1–127, ISSN 1935-8237, doi:10.1561/2200000006, 1509.02971v5.

[3] Bojarski, M.; Del Testa, D.; Dworakowski, D.; aj.: End to End Learning for Self-Driving Cars. *arXiv:1604*, 2016: s. 1–9, 1604.07316.
URL http://arxiv.org/abs/1604.07316

[4] Chen, C.; Seff, A.; Kornhauser, A.; aj.: DeepDriving: Learning affordance for direct perception in autonomous driving. *Proceedings of the IEEE International Conference on Computer Vision*, ročník 11-18-December-2015, č. Figure 1, 2016: s. 2722–2730, ISSN 15505499, doi:10.1109/ICCV.2015.312, 1505.00256.

[5] Chollet, F.; aj.: Keras. https://github.com/fchollet/keras, 2015.

[6] Giusti, A.; Guzzi, J.; Cire, D. C.; aj.: A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. *IEEE Robotics and Automation Letters*, ročník PP, č. 99, 2015: s. 1–1, ISSN 2377-3766, doi:10.1109/LRA.2015.2509024.
URL
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7358076

[7] Karpathy, A.; Li, F.-F.; Johnson, J.: CS231n: Convolutional Neural Networks for Visual Recognition. [Online; navštíveno 10.11.2016].
URL http://cs231n.github.io/

[8] Kim, D. K.; Chen, T.: Deep Neural Network for Real-Time Autonomous Indoor Navigation. 2015, 1511.04668.
URL http://arxiv.org/abs/1511.04668

[9] Koutník, J.: Evolving Large-Scale Neural Networks for Vision-Based Reinforcement Learning. 2013.

[10] Krizhevsky, A.; Sutskever, I.; Geoffrey E., H.: ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25 (NIPS2012)*, 2012: s. 1–9, ISSN 10495258, doi:10.1109/5.726791, 1102.0183.
URL https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[11] Krizhevsky, A.; Sutskever, I.; Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, 2012: s. 1–9, ISSN 10495258, doi:http://dx.doi.org/10.1016/j.protcy.2014.09.007, 1102.0183.

[12] LeCun, Y.; Muller, U.; Ben, J.; aj.: Off-road obstacle avoidance through end-to-end learning. *Advances in neural information processing systems*, ročník 18, 2006: str. 739, ISSN 1049-5258.
URL http://books.nips.cc/papers/files/nips18/NIPS2005{_}0742.pdf?q=07751

[13] Loiacono, D.; Lanzi, P. L.; Togelius, J.; aj.: The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in Games*, ročník 2, č. 2, 2010: s. 131–147, ISSN 1943068X, doi:10.1109/TCIAIG.2010.2050590.

[14] Mnih, V.; Kavukcuoglu, K.; Silver, D.; aj.: Human-level control through deep reinforcement learning. *Nature*, ročník 518, č. 7540, 2015: s. 529–533, ISSN 0028-0836, doi:10.1038/nature14236, 1312.5602.
URL http://dx.doi.org/10.1038/nature14236

[15] Mnih, V.; Silver, D.; Riedmiller, M.: Dqn. *Nips*, 2013: s. 1–9, ISSN 0028-0836, doi:10.1038/nature14236, 1312.5602.

[16] Pomerleau, D.: Neural network vision for robot driving. *Intelligent Unmanned Ground Vehicles*, 1997: s. 1–22, ISSN 08933405.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.4105{%}5Cnhttp://www.springerlink.com/index/NL260M610735P703.pdf

[17] Ross, S.; Gordon, G. J.; Bagnell, J. A.: A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. ročník 15, 2010, ISSN <null>, 1011.0686.
URL http://arxiv.org/abs/1011.0686

[18] Silver, D.; Hubert, T.; Schrittwieser, J.; aj.: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. 2017: s. 1–19, ISSN 23289503, doi:10.1002/acn3.501, 1712.01815.
URL http://arxiv.org/abs/1712.01815

[19] Silver, D.; Schrittwieser, J.; Simonyan, K.; aj.: Mastering the game of Go without human knowledge. doi:doi:10.1038/nature24270.
URL https://www.nature.com/articles/nature24270

[20] Sutton, R. S.; Barto, A. G.: [Draft-2] Reinforcement learning : an introduction. *Neural Networks IEEE Transactions on*, ročník 9, č. 5, 2013: str. 1054, ISSN 1045-9227, doi:10.1109/TNN.1998.712192, 1603.02199.

[21] Uhlenbeck, G. E.; Ornstein, L. S.: On the Theory of the Brownian Motion. *Phys. Rev.*, ročník 36, Sep 1930: s. 823–841, doi:10.1103/PhysRev.36.823.
URL https://link.aps.org/doi/10.1103/PhysRev.36.823

[22] Zhang, S.; Sutton, R. S.: A Deeper Look at Experience Replay. 2017, 1712.01275.