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

NEURONOVÉ SÍTĚ PRO AUTONOMNÍ ŘÍZENÍ AUTA

NEURAL NETWORKS FOR AUTONOMOUS CAR DRIVING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MAREK DOPITA**
AUTHOR

VEDOUCÍ PRÁCE **doc. RNDr. PAVEL SMRŽ, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Dopita Marek**
Program: Informační technologie
Název: **Neuronové sítě pro autonomní řízení auta**
Neural Networks for Autonomous Car Driving
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se se způsoby rozpoznávání objektů a s dostupnými implementacemi pro snadnou konfiguraci a trénování neuronových sítí.
2. Shromážděte datové sady pro průběžné testování systému.
3. Na základě získaných poznatků navrhněte a implementujte systém, který dokáže rozpoznat vybraný druh dopravních situací (např. průjezd křižovatkou) a navrhovat v nich co nejlepší řešení.
4. Vyhodnoťte výsledky systému na reprezentativním vzorku dat.
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky.

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- funkční prototyp řešení

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrž Pavel, doc. RNDr., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstrakt

V této práci jsou představeny principy neuronových sítí se zaměřením na autonomní vozidla. Na těchto informacích je vytvořen návrh implementace systému, který umožňuje řídit automobil bez řidiče. Ten staví na základě nástrojů, které umožňují snadnou tvorbu a testování autonomních vozidel. Jde o CARLA simulátor a Leaderboard.

Návrh rozděluje jízdní trasy vozidel do tří rozdílných situací. Každá situace vyžaduje využití jiných senzorů, proto je vytvořen specifický autonomní agent, který je schopen situaci rozpoznat a přepnout mezi různými návrhy neuronových sítí. Každá taková síť je specifická svými vstupy a je učena na konkrétní situaci.

Jsou vytvořeny programy, které jsou schopny jednoduše za pomoci CARLA Leaderboard posbírat datovou sadu. Poté je představen způsob, jak lze posbíraná data rozdělit do kategorií tak, aby byla každá kategorie možná použít na učení její neuronové sítě.

Abstract

In this work, the principles of neural networks are introduced with a focus on autonomous vehicles. Based on this information, a proposal for the implementation of a system is created, which allows to drive a car without a driver. It builds on tools that allow easy creation and testing of autonomous vehicles. It is CARLA simulator and ranking.

The proposal divides vehicle routes into three different situations. Each situation requires the use of different sensors, so a specific autonomous agent is created that is able to recognize the situation and switch between different neural network designs. Each such network is specific in its inputs and is taught in a specific situation.

Programs are created that are able to easily collect a data set using the CARLA Leaderboard. Then, a way is introduced to how the collected data can be divided into categories so that each category can be used to learn its neural network.

Klíčová slova

umělá inteligence, senzory, neuronové sítě, gradientní sestup, zpětná propagace, konvoluční neuronové sítě, autonomní řízení, simulátory, CARLA, CARLA simulátor, CARLA Leaderboard, sběr datových sad, návrh neuronových sítí, učení neuronových sítí, implementace autonomního agenta

Keywords

artificial intelligence, sensors, neural networks, gradient descent, backpropagation, convolutional neural networks, autonomous driving, simulators, CARLA, CARLA simulator, CARLA Leaderboard, datasets collection, neural networks design, neural networks training, implementation of autonomous agent

Citace

DOPITA, Marek. *Neuronové sítě pro autonomní řízení auta.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Pavel Smrž, Ph.D.

Neuronové sítě pro autonomní řízení auta

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. RNDr. Pavla Smrže, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....

Marek Dopita

11. května 2022

Obsah

Kapitola 1

Úvod

Autonomní řízení v automobilovém průmyslu je právě nastupující trend. I přesto, že koncept neuronových sítí je známý mnoho let, teprve někdejší pokrok v oblasti senzoriky a především v oblasti výpočetního výkonu počítačů dovolil jejich využití v tak komplexních problémech, jako je řízení automobilů. Využití neuronových sítí je ten nejlepší způsob, který je k dispozici. Umožňují naučit počítače osazené v automobilech řešení veškerých dopravních situací, které mohou nastat.

Důvodem rozvoje autonomního řízení není jen přirozená lidská lenost, i když ta hraje jistě nějakou roli. Hlavním důvodem je bezpečnost. Dle policie České republiky v roce 2020 zemřelo na českých silnicích 450 osob [27]. Z toho u 417 osob (tedy 90,7 %) je uváděn jako viník řidič motorového vozidla. Člověk se jako každý živý tvor dopouští chyb, z toho důvodu využíváme moderních technologií tam, kde lidské schopnosti již nestačí. Palubní počítač dokáže předpovědět kolizi a zareagovat mnohem rychleji a efektivněji než kterýkoli člověk.

Důvodů ke studiu tématu je několik. Především se jedná o téma s velkým potenciálem a je zcela jisté, že v následujících letech propukne velký vývoj autonomních systémů. Téma nebude rozebírané pouze odborníky, ale bude diskutováno i laickou veřejností. Aktuální stav má mnoho výhod a v podstatě jsou všechny dveře otevřené. Neexistuje myšlenka, která by byla špatná. Osobně bych se v budoucnu rád věnoval výzkumu autonomních zařízení, které mají široké využití od automobilového průmyslu na Zemi, přes vesmírné satelity, až po rovery na Marsu. Výběr tématu souvisí s mou snahou zjistit si nezbytné informace, na kterých mohu v dalších letech svého studia a profesní kariéry stavět.

Autonomní řízení se aktuálně nachází ve svých počátcích. Pro klasifikaci vozidel z hlediska automatizace je využíván standard J3016 od SAE International (Společnost automobilových inženýrů) [24]. Ten rozděluje automobily do šesti úrovní. Historicky je řídili pouze lidé. Nebyla v nich žádná chytrá elektronika, která by napomáhala řízení. Tento stav vyjadřuje úroveň 0. Teprve pár let jsou využíváni jízdní asistenti. Ti částečně zasahují do řízení, ale hlavní úlohu má stále řidič. Jedná se o systémy první úrovně. Je to například adaptivní tempomat nebo adaptivní udržování v jízdním pruhu. Běžní jsou také parkovací asistenti nebo asistent, který hlídá mrtvé úhly ve zpětných zrcátkách. Rozšiřují se vozidla třetí úrovně, které řídí z větší části počítač. Člověk je v těchto vozidlech ovšem nepostradatelný, jeho úlohou je dohlížet na správnou funkčnost systému a případně převzít řízení. Čtvrtá úroveň vozidel umožňuje samostatné řízení bez lidského faktoru za určitých podmínek. Těmi mohou být místo operování automobilu nebo rychlost. Plně autonomními vozidla úrovně pět existují pouze v rámci testovacího provozu.

Veškerá inteligence a rozhodování je postaveno na neuronových sítích, ty jsou specifikované v kapitole 2. V ní jsou neuronové sítě zasazeny do kontextu umělé inteligence. Součástí kapitoly jsou základní informace o senzorech využívaných pro autonomní vozidla. Jsou brány jako vstup neuronových sítí, který zpracovávají. Dále je popsán princip a rozebrána matematika na pozadí, která souvisí s jejich funkcí. Také jsou k nalezení podrobné informace o systému, jak je možné neuronovou síť učit. Pro zpracování obrázků jsou představeny konvoluční neuronové sítě.

Po vysvětlení funkce neuronových sítí se v kapitole 3 představí způsoby, jak jednoduše navrhovat systémy pro autonomní řízení. Místo zdlouhavé stavby automobilů budou popsány simulátory. Součástí této kapitoly je popis žebříčku od společnosti CARLA, který slouží jako metrika porovnávání kvality mezi implementacemi systémů pro vozidla. Je zde vysvětleno, co se vyhodnocuje, kde se to vyhodnocuje a jak je popsána kvalita výstupu. Představeny jsou veškeré prostředky vhodné pro tvorbu vlastního řešení.

V kapitole 4 je možné se dočíst, jakým způsobem bylo navrhnuto řešení pro autonomní řízení v CARLA simulátoru. Současně s popisem architektury je zde vysvětlena i samotná implementace. Celkové řešení se skládá z pořízení a přípravy datové sady, vytvoření neuronových sítí a programů pro trénink a validaci. Také popisuje sestavení programu, který řídí automobil. V kapitole 5 je vysvětlen výstup z testování automobilu.

Kapitola 2

Neuronové sítě a senzory autonomních vozidel

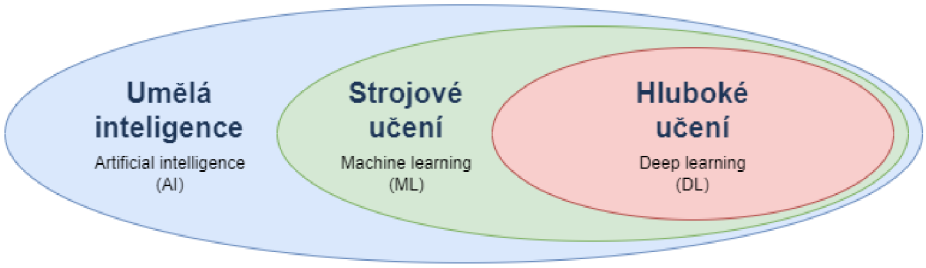
Neuronové sítě (Neural networks) nebo také umělé neuronové sítě (Artificial neural networks) jsou nedílnou součástí umělé inteligence a strojového učení. I přesto, že se může zdát, že se jedná o vynález z posledních let, jejich počátky sahají již do roku 1943, kdy Warren S. McCulloch a Walter Pitts zveřejnili svoji studii [20], která zkoumala funkce lidského mozku a jak vytváří složité vzory pomocí propojených mozkových buněk neboli neuronů. Na této práci v průběhu let stavělo mnoho dalších výzkumů. Neuronové sítě se postupně utvářely a vylepšovaly.

Za posledních deset let se neuronové sítě objevují téměř všude. Jejich využití je prakticky neomezené z důvodu, že si berou inspiraci z chování lidského mozku. Tak jako mozek, jsou schopny rozpoznání a klasifikace objektů svého okolí. Využívají se jako vyhledávací algoritmy, rozpoznávače řeči a obrazů a samozřejmě v autonomních vozidlech pro řízení.

Popis sítí skvěle rozšiřuje elektronická kniha od Michaela Nielsena „Neural Networks and Deep Learning“ (Neuronové sítě a hluboké učení) [21].

2.1 Umělá inteligence, strojové učení a souvislost s neuronovými sítěmi

Pojmy, které se v kontextu neuronových sítí často vyskytují a je nutné jim porozumět. Ve zjednodušené formě se jedná o vzájemné podmnožiny, které vystihuje obrázek 2.1. Umělá inteligence obsahuje jako jednu ze svých podčástí strojové učení, ta využívá hlavně neuronové sítě, pokud se využívá v síti více neuronových vrstev [2.3], jedná se o hluboké učení.



Obrázek 2.1: Souvislosti umělé inteligence, strojového učení a hlubokého učení.

Umělá inteligence (Artificial intelligence - AI)

Pod umělou inteligencí [28] je možné si představit veškeré schopnosti stroje napodobovat lidské chování. Umělou inteligenci jde dále dělit. Základní rozdělení je na slabou a silnou umělou inteligenci.

Mezi slabou umělou inteligenci patří schopnosti stroje napodobit jednu konkrétní úlohu, může se jednat například o hraní šachové hry, interpretace emocí a hlasoví asistenti.

Druhou skupinou jsou silné umělé inteligence, ty se momentálně nevyskytují, jejich schopností je plně napodobit či dokonce předčit člověka v jakékoli možné situaci.

Strojové učení (Machine learning - ML)

Základní vlastností člověka je schopnost se učit novým věcem. Aby byly stroje schopné lidem konkurovat, i ony potřebují způsoby, jak se učit a přizpůsobovat. Této oblasti umělé inteligence se věnuje právě strojové učení [11].

Nejedná se o nic jiného než algoritmy, které se mohou v průběhu modifikovat. Přesnost výsledného algoritmu závisí na jeho učení. Rozeznává se několik způsobů, jak je možné algoritmy učit. Učení probíhá na speciálních datových sadách, které obsahují výtah z dat, se kterými bude algoritmus pracovat. Ty mohou dosahovat mnoho podob dle typu učení.

Učení s učitelem (Supervised learning)

Algoritmy využívající tuto techniku učení jsou učeny na datových sadách, které jsou předem nějakým způsobem označeny nebo rozděleny dle očekávaného výstupu. Algoritmus zná výstup a je schopen se naučit parametry vstupů, které vedou ke správnému řešení. Využitím této metody učení je klasifikace objektů do dvou (binární) nebo více tříd (více-třídní). Další využití je regrese, tedy odhad náhodné veličiny na základě vstupu.

Učení bez učitele (Unsupervised learning)

Největší množství algoritmů využívá právě učení bez učitele. Učení zde probíhá na datových sadách, které nejsou nijak označeny ani klasifikovány. Datové sady jsou tak podmnožina dat reálného systému, ve kterém bude algoritmus pracovat. Algoritmy v této kategorii hledají podobnost a vzory mezi daty. Jsou vhodné pro shlukování, kdy algoritmy rozdělí datové sady na základě jejich podobnosti. Jsou schopny mezi daty detekovat anomálii, nebo informace mezi daty, které se často vyskytují pospolu.

Zpětnovazebné učení (Reinforcement learning)

Využívá dvou modulů. Prvním je agent neboli systém, který začne vydávat akce. Tyto akce se projeví v prostředí, ve kterém se nachází. Jedná se vždy o kladnou nebo zápornou odezvu, ta se projeví pomocí odměn, nebo trestů. Agent se po celou dobu učení snaží získat odměny a vyhnout se trestům. Postupnou iterací se agent stává chytřejším a učí se novým věcem. Bude taky vědět jaké akce má vydávat tak, aby získal co největší odměnu. Mimo autonomní vozidla se tento typ učení využije i v robotice nebo videohrách.

Hluboké učení (Deep learning - DL)

Hluboké učení [33] je podmnožinou strojového učení, které využívá stejných metod. Jediným rozdílem je množství zpracovávaných dat. Algoritmy pro hluboké učení pracují s extrém-

ním množstvím dat, které potřebují co nejefektivněji zpracovat a dosáhnout požadovaného výsledku. K tomu se ukázaly jako užitečné neuronové sítě, které obsahují velké množství neuronových vrstev. Více o vrstvách v kapitole 2.3. Vhodným příkladem využití hlubokého učení jsou právě autonomní vozidla.

2.2 Vstupy neuronových sítí aneb senzory vozidel

Další analogii s mozkem jde sledovat v celkové funkci. Mozek přebírá informace ze svého okolí pomocí smyslů. U řízení to je především zrak a sluch. K tomu potřebuje orgány, které mu dodají informace. Neuronové sítě pracují na stejném principu. Pouze nahradily lidské orgány za elektronické senzory [3]. Těch existuje mnoho typů, přičemž každý má odlišné vlastnosti. Ty ovlivňuje i kvalita a cena zvoleného typu. Nelze očekávat, že automobil dokáže dosáhnout výborných výsledků s nekvalitními čidly.

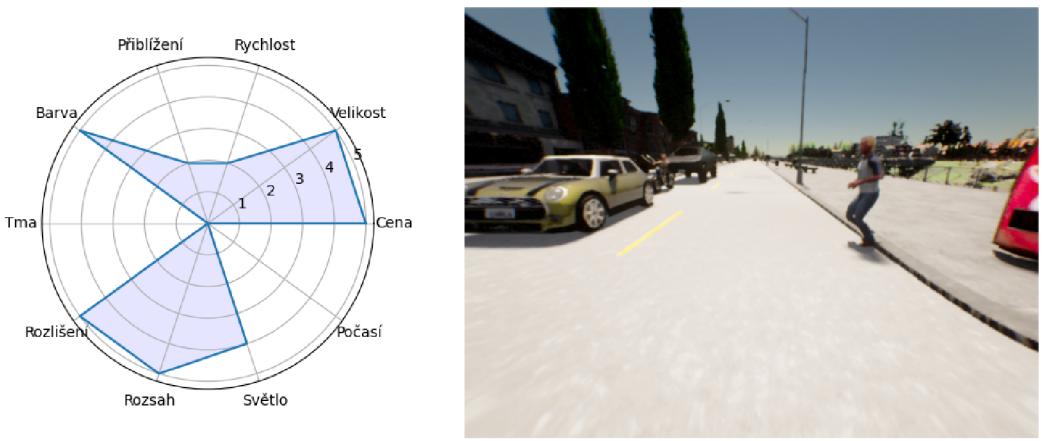
Využití jednoho senzoru na automobilech většinou nestačí. Různé společnosti využívají rozdílné kombinace senzorů, které považují za nejideálnější k dosažení nejlepších výsledků. Mezi ty hlavní patří kamera, RADAR a LiDAR [19]. Jsou to hlavní senzory, pomocí kterých automobil rozpoznává své okolí a konkrétní popis je níže. K těmto třem se přidává i ultrazvuk. Ten je schopen rozeznat překážky na krátkou vzdálenost. Je využíván především jako parkovací senzor.

Kromě těchto hlavních senzorů lze nalézt i další. Ty místo okolí zaznamenávají spíše informace o automobilu. Je to například GNSS senzor, který zjistí GPS polohu. IMU senzor pro záznam akcelerace a náklonu. Může mezi ně patřit i tachometr a jiné [5].

Kamera

Základní senzor, který ve své podstatě funguje na stejném principu jako lidské oko. Dokáže zaznamenat odražené světlo od objektů a vytvoří obrázek svého okolí. Typicky se jedná o obrázky složené ze tří barevných složek RGB. Červená, zelená a modrá.

Kamery jsou naprosto běžné, díky tomu je jejich cena velmi přívětivá a výrobci jsou schopni osadit automobily více kusy. Je to prakticky jediný senzor, díky kterému lze rozeznat barvy na semaforech. Jeho nevýhodou je, že při zhoršených podmínkách velmi rychle ztrácí na účinnosti. Déšť, sníh, mlha, jasná obloha, všechny tyto věci zapříčiní selhání kamer. Vlastnosti kamery s ukázkou výstupu popisuje obrázek 2.2.

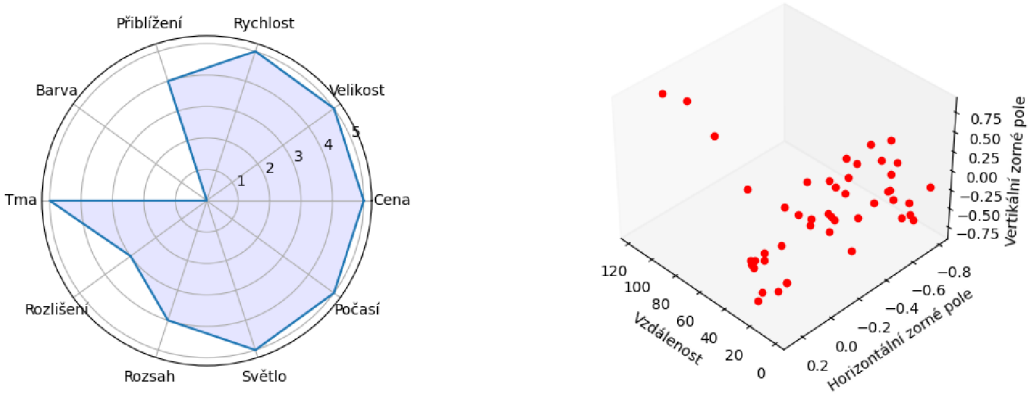


Obrázek 2.2: Vlastnosti a ukázka výstupu kamery.

RADAR (Radio Detection and Raging)

Čidlo vysílá rádiové (elektromagnetické) vlny, které se šíří okolím. Díky jejich odrazům zpět do senzoru jde dopočítat poloha, vzdálenost a rychlost pohybu nalezeného objektu.

Senzory jsou malé a levné. Pracují v jednom směru s určitým pozorovacím úhlem. Dokáže proniknout přes špatné světelné podmínky. Neposkytuje zpětné vizuální informace, výstupem je pouze skupina bodů v okolí. Obrázek 2.3 popisuje vlastnosti RADARu a ukázku jeho výstupu.

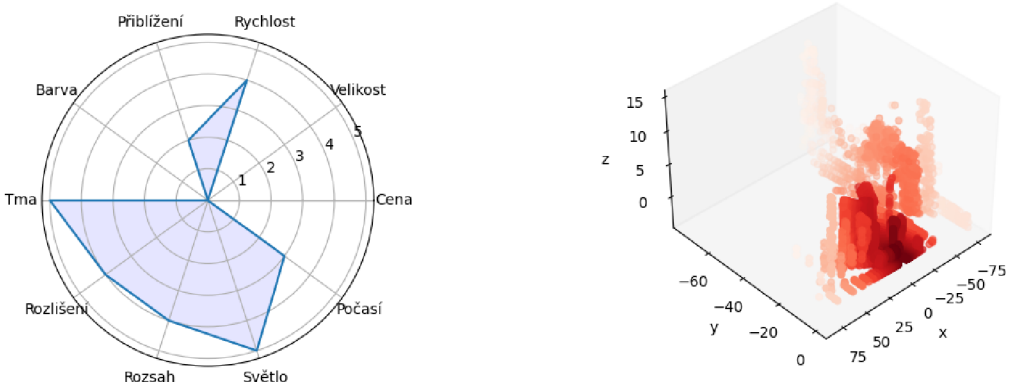


Obrázek 2.3: Vlastnosti a ukázka výstupu RADARu.

LiDAR (Light Detection and Raging)

LiDAR je v autonomních vozidlech stále ještě novinkou. Tento typ senzorů existuje dlouho. Jeho problémem byla ještě donedávna velikost. V posledních letech jsou vynalézány menší typy, kterými lze osadit vozidlo. To se negativně projevuje na ceně.

Funguje na stejném principu jako RADAR. Pouze místo rádiových vln používá světlo. Jeho vlnová délka je o několik řádů menší, to znamená vyšší rozlišovací schopnosti. Jsou ve většině případů otočné a vytváří mapu bodů svého okolí, kde došlo k odrazu paprsku.



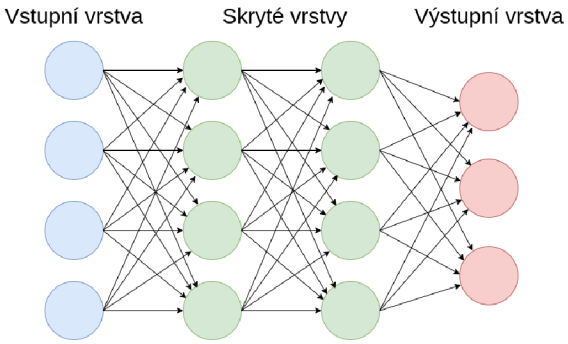
Obrázek 2.4: Vlastnosti a ukázka výstupu LiDARu.

Vzhledem k malé vlnové délce světla, nedokáže účinně proniknout rozptýlenými částečky v atmosféře jako je například déšť, mlha nebo smog. Vlastnosti a výstup LiDARu je na obrázku 2.4.

2.3 Princip neuronových sítí

Principiálně jsou neuronové sítě [15] velmi jednoduché struktury, jak je možné pozorovat na obrázku 2.5. Jedná se o skupinu uzlů, které označujeme jako neurony. Ty jsou organizovány do vrstev. Každá neuronová síť jich má několik, mezi nimi je vstupní vrstva. Na tyto neurony jsou přivedeny vstupní hodnoty, které má síť zpracovat. Po zpracování dat dostaneme výsledek na výstupní vrstvě. Mezi těmito plochami se může nacházet několik skrytých vrstev. Počet vrstev i počet neuronů v jedné řadě závisí na řešeném problému. Typicky není žádné jedno správné řešení, konečný počet vrstev a uzlů je výsledkem experimentování s neuronovou sítí.

Každý neuron jedné vrstvy je propojen s každým neuronem vrstvy následující, takto vzniká komplikovaná síť. Princip výpočtu zahrnuje přivedení vstupních dat na vstupní vrstvu, ta rozešle svá data všem neuronům další vrstvy. Zde proběhnou matematické operace a jejich výsledky jsou opět odeslány vrstvě v pořadí. Takto se data šíří celou sítí, dokud nedorazí k výstupní vrstvě, zde je možné naleznout výsledky řešení.



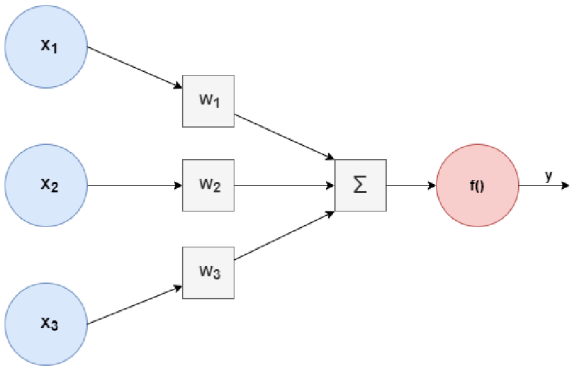
Obrázek 2.5: Část struktury neuronových sítí.

Matematika za neuronovými sítěmi

Protože by pouhé rozesílání dat mezi neurony [15] nebylo k užitku, musí síť obsahovat matematické operace [23], které budou transformovat data. Ty jsou umístěny na každém neuronu a je možné je vyjádřit jednoduchou matematickou rovnicí.

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

Jelikož vysvětlení z rovnice nemusí být patrné, obrázek 2.6 popisuje rovnici za pomoci grafu. Na vstupu uzlu máme několik neuronů z předchozí vrstvy, jejich hodnoty jsou označeny jako x. Každou z těchto hodnot vynásobíme váhou w, která je předem známá pro každou ze vstupních hodnot. Kromě vah je možné se setkat i s biasy, o nich více v jejich vlastní sekci 2.3. Takto transformované hodnoty jsou sečteny. Součet je poté vložen do aktivační funkce neuronu, jejím výsledkem je nová hodnota neuronu y, která je předávaná neuronům v následující vrstvě.



Obrázek 2.6: Graf výpočtu hodnoty neuronu.

Jelikož se pracuje s velkým počtem prvků, je výhodné pro implementaci algoritmů využít matice. Vlastnosti matic značně zjednodušují výpočty, navíc je možné tyto výpočty na počítači lépe akcelerovat, například souběžným výpočtem.

V následující rovnici je příklad využití matic pro vrstvu o třech neuronech, která transformuje data z předešlé vrstvy o dvou neuronech.

$$\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} f(w_{11}x_1 + w_{21}x_2) \\ f(w_{12}x_1 + w_{22}x_2) \\ f(w_{13}x_1 + w_{23}x_2) \end{bmatrix}$$
$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} * \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} = \begin{bmatrix} f(x_1w_{11} + x_2w_{21}) & f(x_1w_{12} + x_2w_{22}) & f(x_1w_{13} + x_2w_{23}) \end{bmatrix} \tag{2.2}$$

Aktivační funkce

Výše v kapitole bylo zmíněno, že každý neuron obsahuje aktivační funkci [29], ta je v sítích potřebná, jelikož dodává nelinearitu. Pokud by síť neobsahovala nelinearitu, veškeré transformace, které by neurony prováděly by bylo sčítání a násobení hodnot a vah. To by vedlo na čistě lineární operace, výsledkem by byla pouze lineární regrese. Veškerá inteligence neuronových sítí by byla ztracena.

Jak již samotný název napovídá, funkce se jmenuje aktivační, jelikož říká, zda má být její neuron aktivován, či nikoli. Pokud je hodnota funkce vysoká znamená to, že daný neuron má v síti velký význam, nízká hodnota naopak naznačuje, že neuron má pro vstupní hodnoty význam malý.

Existuje množství aktivačních funkcí, které je možné použít. Jejich použití závisí na řešeném problému. Při tvoření neuronové sítě se aktivační funkce přidělují jednotlivým vrstvám modelu. I zde neexistuje správné a jednotné řešení jaké aktivační funkce využít a jedná se o výsledek experimentování se sítí, lze ovšem využít několika vodítek [22].

První skupinou aktivačních funkcí jsou funkce binární. Pokud hodnota překročí zadanou hranici, neuron vrací hodnotu 1, jinak 0. Jejich použití není časté, jelikož jsou neuronové sítě zpravidla používané na komplexnější problémy jako například třídní klasifikace, pro kterou tato funkce nelze použít. Na rovnici níže je vidět zápis binární funkce, práh zde tvoří konstanta C.

$$f(x) = \begin{cases} 0 & -\infty < x < C \\ 1 & jinak \end{cases} \tag{2.3}$$

Druhá skupina jsou lineární funkce. I zde je jejich využití omezené. Jdou označit za funkce identity, výstup je proporčně závislý na vstupu. Zásadní chybou této skupiny je, že při jejich využití zanikají veškeré výhody více vrstev. Jelikož u lineárních operací nezáleží na pořadí, je možné veškeré váhy spojit a výsledek odeslat do jedné vrstvy lineárních funkcí. Příklad funkce z této kategorii je vidět v následujícím předpisu. Výstup funkce je přímo závislý na dvou konstantních parametrech A a B.

$$f(x) = A * x + B \tag{2.4}$$

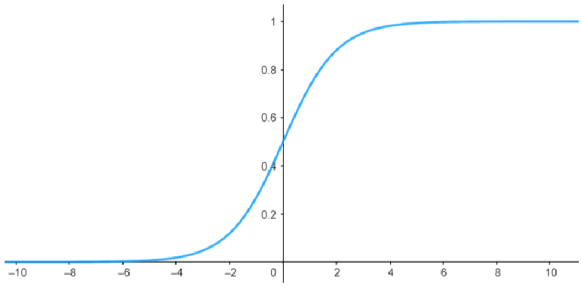
Poslední skupinou jsou nelineární funkce, ty jsou hlavní podstatou sítí. Nelineárních funkcí existuje nekonečně mnoho, pro potřeby neuronových sítí se setkáváme s určitou skupinou funkcí, které mají své pojmenování a různé vlastnosti.

Sigmoid

Aktivační funkce využívaná jak ve skrytých, tak ve výstupních vrstvách. Výstupem funkce, je vždy hodnota mezi 0 a 1. Kladné hodnoty jsou transformovány blíže k 1, záporné naopak blíže k 0. Má následující předpis.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

Vzhledem k jejímu oboru hodnot je tahle funkce k nalezení v sítích pracující s pravděpodobností. Výstupem transformací je hladký průběh. Vzhledem k vlastnostem její derivace je neuronová síť využívající tuto aktivační funkci těžší k učení, více informací o učení neuronových sítí v sekci 2.4.



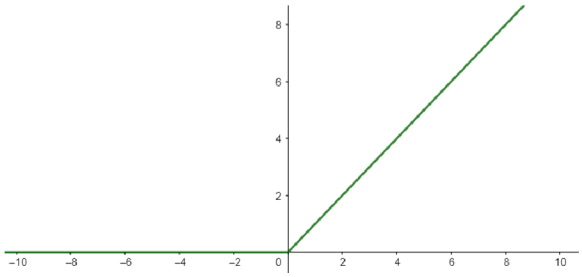
Obrázek 2.7: Sigmoid aktivační funkce.

ReLU

Funkce využívaná především ve skrytých vrstvách. Její výhoda je jednoduchá implementace. Jednoduše vrátí větší číslo mezi 0 a zadanou hodnotou na ose x. Také by se dalo říct, že pro všechna záporná čísla vrací funkce 0, zatímco pro všechna kladná a nulu vrátí identickou hodnotu. Rovnice funkce je v tomto případě velmi jednoduchá.

$$f(x) = max(0, x) \tag{2.6}$$

Vzhledem k tomu, že ReLu všechna záporná čísla mapuje na 0, je tato funkce více efektivní při výpočtech, přesto mohou nastat problémy. Výsledný model nemusí tak přesně odpovídat, jelikož je ztracena informace o záporných hodnotách, se kterými se nepočítá.



Obrázek 2.8: ReLU aktivační funkce.

Softmax

Příklad funkce, která se využívá na výstupní vrstvě. Pomocí ní jsou hodnoty neuronů transformovány do hodnot mezi 0 a 1, přitom si zachovávají původní rozložení, největší hodnoty jsou umístěny blízko 1, nejnižší blíže 0 a vše ostatní mezi nimi. Je vhodná pro třídní klasifikaci, hodnota na výstupním neuronu říká, s jakou pravděpodobností vstup spadá do třídy, kterou uzel představuje.

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{2.7}$$

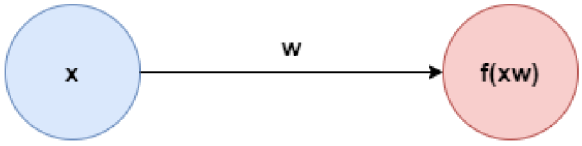
Transformovaná hodnotu neuronu je možné získat tak, že epsilon je umocněn původní hodnotou uzlu. Následně je mezivýsledek podělen součtem přes všechny hodnoty uzlů ve vrstvě, přičemž i zde jsou původní hodnoty mocninami epsilonu.

Váhy a biasy

Základní aktivační funkce bohužel nestačí. Je nutno využít prvků pro jejich transformace. Každý neuron v jedné vrstvě pracuje se stejnou aktivační funkcí, to by mělo za následek, že každý neuron vrstvy by pracoval naprosto stejně. Cílem je funkce na každém neuronu upravit, toho lze dosáhnout právě váhy a biasy, které byly krátce zmíněny na začátku kapitoly.

Váhy

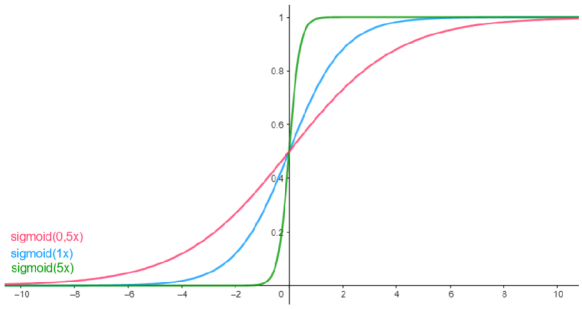
Každé propojení neboli synapse zahrnuje jedno číslo, které vyjadřuje jeho váhu.



Obrázek 2.9: Váhy propojení neuronů.

Váha propojení je vynásobena se vstupem neuronu a poslána do aktivační funkce. Jak je možné pozorovat na obrázku 2.10, různá hodnota vah ovlivňuje roztažení aktivační funkce

sigmoid na ose x. Hodnoty, které jsou větší jak jedna, funkci zkracují. Její průběh roste rychleji. Naopak hodnoty menší funkci roztáhnou a hodnoty rostou pomaleji.



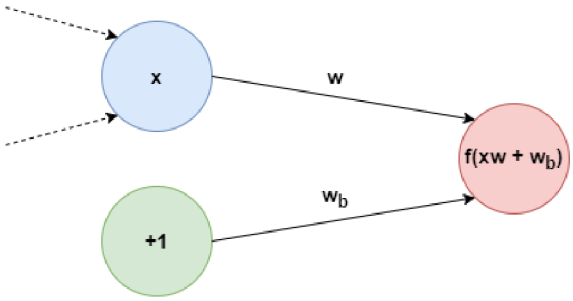
Obrázek 2.10: Chování vah na grafu aktivační funkce sigmoid.

Biasy

Samotné roztažení funkcí může být nedostatečné, proto se velmi často využívá biasů. Jsou to hodnoty, které se přičítají k hodnotě vstupu vynásobeé s váhou. Existují dvě implementace, jak je možné tohoto chování dosáhnout. První z nich zahrnuje speciální matici posunů, která je po vynásobení matic vah a vstupů přičtena tak, jak to vyjadřuje následující rovnice. Velká písmena značí operace s maticemi. Jinými slovy, každý neuron si uchovává v matici hodnotu biasu.

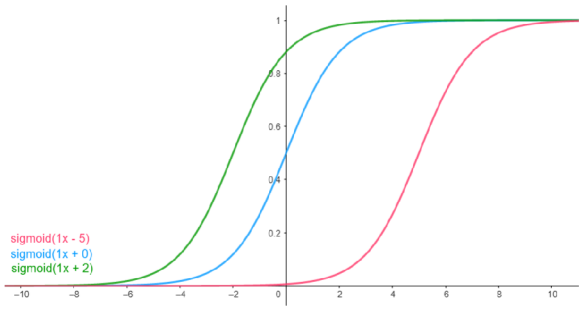
$$y = f(XW + B) \tag{2.8}$$

Druhé řešení je přidat do každé vrstvy jeden neuron, který není napojen na předešlou vrstvu a jeho výstupem je vždy hodnota 1. Výstup je napojen na všechny neurony následující vrstvy a každá synapse má váhu, ta se násobí konstantou 1, váha tedy přímo udává hodnotu posunu. Výhodou tohoto řešení je, že dojde pouze k rozšíření matic, které se násobí a posun je tak implicitní. Není nutné přičítání další matice.



Obrázek 2.11: Biasy neuronů pomocí neuronu předešlé vrstvy.

Chování biasů je možné pozorovat na grafu na obrázku 2.12. Jedná se o konstantu, která celou funkci dokáže posunout po ose x. Záporné hodnoty funkci posouvají směrem doprava, kladné hodnoty naopak doleva.



Obrázek 2.12: Chování biasů na grafu aktivační funkce sigmoid.

Typy neuronových vrstev

Do teď byl popsán jen jeden typ vrstev neuronových sítí. Ta se označuje jako lineární vrstva, jelikož vstupem aktivační funkce je funkce lineární.

$$Y = XW + B \tag{2.9}$$

Neuronové sítě se mohou skládat z více typů vrstev. Mohou to být například vrstvy konvoluční a sdružovací, které jsou zapotřebí v konvolučních sítích. Jejich funkce je popsána v sekci 2.5.

Speciálním typem vrstev jsou vrstvy označované jako dropout [34]. Ty jsou využívány pouze při učení neuronových vrstev. Jejím cílem je s pravděpodobností nastavit některým neuronům vstupní hodnotu nula. Díky tomu, že se náhodně vypne několik neuronů se omezí pravděpodobnost, že nastane jev přeučení (overfitting). Je to stav, kdy se neuronová síť dokáže naučit pravidelnosti v datové sadě a výstupem bude pouze lineární regrese. Pokud dochází k vyhodnocování dat neuronovou sítí, jsou dropout vrstvy vypnuty.

Také existují normalizační vrstvy. Nejvýznamnější z nich je batch normalizace [32]. Jedná se o vrstvu, která provádí normalizaci výstupů aktivačních funkcí jiných vrstev. Normalizovaná data jsou důležitá. Pokud by k normalizaci nedocházelo, neuronová síť by se učila mnohem pomaleji.

Typů je ještě obrovské množství. Jejich přehled je k nalezení například v dokumentaci knihovny Pytorch [26] v Pythonu, která poskytuje prostředky pro práci s neuronovými sítěmi.

Výsledek matematických operací

To je k principu funkce neuronových sítí vše. Neuronová síť vezme vstupní data a dokáže je transformovat na data výstupní. Výstupní data mohou nabývat mnoho podob v závislosti na požadovaném chování. Ve výsledku se jedná vždy o transformaci jedné funkce na druhou. Je už jedno jak tato funkce bude interpretována. Může se jednat například o obrázek, kdy jednotlivé neurony představují hodnoty pixelů obrázků. Stejně tak se může jednat o pravděpodobnosti. Hodnoty výstupních uzlů například říkají pravděpodobnost, že vstupní data spadají do třídy definované na neuronu. Jsou vhodné i pro autonomní řízení vozidel, jelikož jsou schopny interpretovat údaje ze všech možných typu senzorů. Možnosti jsou neomezené.

2.4 Učení neuronových sítí

Jedna z otázek principu neuronových sítí stále nebyla odpovězena. Kde je možné získat hodnoty vah a biasů? Odpověď je jednoduchá, síť se je naučí sama. Výsledný proces učení není nic jiného než hledání vhodných hodnot parametrů jednotlivých neuronů. Z počátku jsou parametry nastaveny náhodně. V průběhu učení dochází k jejich zpřesňování.

Chybová funkce

Dalším problémem, se kterým je třeba se vypořádat je schopnost určit, zda jsou výstupy neuronové sítě správné. Dalo by se říci, že správné řešení není za pomoci neuronových sítí nikdy dosaženo, vždy se mu snažíme pouze co nejvíce přiblížit. Vzdálenost od správného řešení nám říká chyba. K jejímu výpočtu existují chybové funkce [10], ty musí být vždy zvoleny vhodně k řešenému problému.

Střední čtvercová chyba (Mean Squared Error - MSE)

Chybová funkce využívající se mimo jiné i ve statistice. Počítá střední odchylku druhých mocnin rozdílu hodnot získaných z neuronové sítě a referenčních výsledků. Princip je následující: rozdíl hodnoty naměřené neboli hodnoty neuronové sítě a referenční hodnoty určuje odchylku pokusu od správného řešení. Jelikož odchylka nemůže být záporná, je použita druhá mocnina. Výsledky všech pokusů jsou sečteny a poděleny počtem pokusů, tím vzniká střední hodnota všech provedených pokusů.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.10}$$

Využívá se jako hlavní funkce pro všechny regresní problémy. Využitím druhé mocniny rozdílu se získá vyšší chyba v místech, kde se výstup příliš liší, naopak kde se výstup podobá, je k nalezení nižší chyba.

Střední absolutní chyba (Mean Absolute Error - MAE)

Funkce velmi podobná předešlé. Jediným rozdílem je, že místo druhé mocniny docílí nezáporných čísel pomocí absolutní hodnoty. I zde se provádí rozdíl výstupu s tím očekávaným.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{2.11}$$

I tato funkce je použita na regresní problémy. Rozdílem je, že je více vhodná pro data, ve kterých se vyskytují velké odchylky od střední hodnoty dat, kterým se snaží neuronová síť naučit.

Křížová entropie (Cross Entropy)

Chybová funkce využívaná tam, kde se pracuje s pravděpodobnostmi. Jejím hlavním cílem je zjistit chybu mezi dvěma rozděleními pravděpodobnosti. Rozdělení může nabývat diskrétní nebo spojité podoby, výstupem neuronové sítě je vždy funkce diskrétní. Jedná se o informaci, při níž každé třídě nebo hodnotě je přiřazena pravděpodobnost se kterou nastane. Výpočet

probíhá pro každou třídu zvlášť jako očekávaná pravděpodobnost krát přirozený logaritmus zjištěné pravděpodobnosti. Veškeré tyto výsledky jsou sečteny.

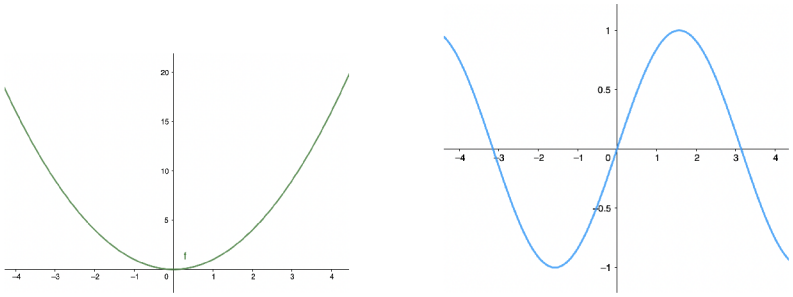
$$CE = -\sum_{i=1}^{N} y_i \log \hat{y}_i \tag{2.12}$$

Křížová entropie je základní chybovou funkcí využívanou při binární či více třídní klasifikaci. Každý výstupní neuron označuje pravděpodobnost, že nastala daná třída, kterou představuje.

Gradientní sestup (Gradient descent)

Jedná se o matematický iterativní algoritmus, který hledá lokální extrém funkce za pomocí jejích derivací. K rozšíření informací o gradientním sestupu je možné využít kapitolu „Optimization Algorithms“ v knize „Dive into Deep Learning“ [37].

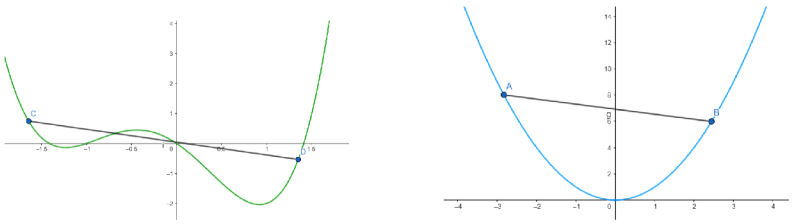
V kontextu neuronových sítí jsou tyto funkce právě funkce chybové, ty mají ovšem nějaké omezení. Aby bylo možné použití algoritmu, funkce musí být diferencovatelné a konvexní. Diferencovatelné funkce jsou takové, které mají ve všech bodech funkce definovanou derivaci. Příklady diferencovatelných funkcí lze pozorovat na obrázku 2.13.



Obrázek 2.13: Diferencovatelné funkce.

Funkce, které nejsou diferencovatelné mají ve svém průběhu jisté rysy. Jedním z nich je konečně nebo nekonečně dlouhý skok. V tomto bodě funkce mění skokově svou hodnotu, tedy není spojitá. Dále jsou to funkce, které obsahují ostré změny hodnot. V místě, kde dochází k překlopení jsou sice definované derivace zleva a zprava, celková derivace ale chybí.

Konvexní funkce se poznají tak, že pokud jsou propojeny jakékoliv dva body funkce a je mezi nimi natažena úsečka, tato úsečka neprotne žádný další bod funkce. Konvexnost lze také zjistit pomocí druhé derivace funkce. Na obrázku 2.14 je možné vidět příklad konvexní a nekonvexní funkce.



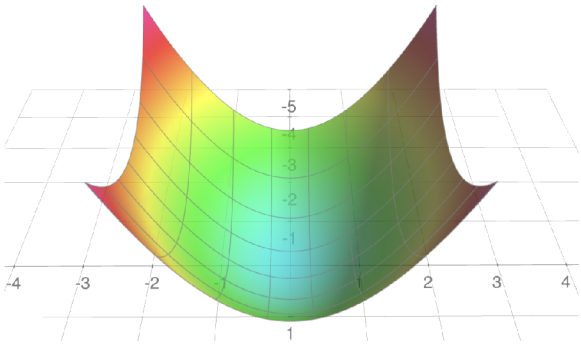
Obrázek 2.14: Nekonvexní funkce (vlevo) a konvexní funkce (vpravo).

Pro všechny chybové funkce musí platit tyto podmínky, poté je možné použít gradientního sestupu.

V prvním kroku sestupu je nutné vypočítat gradient. Ten je závislý na počtu rozměrů funkce, u jednorozměrných funkcí existuje jeden gradient, u vícerozměrných existuje matice gradientů. Každý gradient je derivací funkce v jednom rozměru a říká, jak se v tomto rozměru funkce mění, tedy zda stoupá, klesá nebo je konstantní. U neuronových sítí se derivuje funkce pro každou váhu nebo bias, vždy se jedná o mnoho gradientů.

V následujících příkladech se pro zjednodušení pracuje pouze s dvourozměrnou funkcí, která je popsána rovnicí níže a je vykreslená na obrázku 2.15.

$$f(x, y) = 0,5x^2 + y^2 \tag{2.13}$$



Obrázek 2.15: Vzhled funkce pro ukázku gradientního sestupu.

Jelikož se jedná o dvourozměrnou funkci, je nutné vypočítat dva gradienty. Ty se vytvoří tak, že se předpis funkce derivuje dle osy x pro jeden gradient a dle osy y pro gradient druhý.

$$\frac{\partial f(x, y)}{\partial x} = x \qquad \frac{\partial f(x, y)}{\partial y} = 2y \tag{2.14}$$

Dosazením bodu funkce je možno získat hodnoty derivací. Pokud derivace klesají, lokální minimum je od bodu na pravé straně, naopak pokud stoupají minimum je vlevo. Gradienty jde také zapsat do matice.

$$\nabla f(x, y) = \begin{bmatrix} x \\ 2y \end{bmatrix} \tag{2.15}$$

Gradientní sestup začne v náhodném bodě funkce, vypočítá hodnoty derivací dosazením bodu do gradientu, dle nich se rozhodne, kterým směrem se bude pohybovat. Udělá krok k lokálnímu minimu. Výpočet nového bodu, na který bude proveden krok se vypočítá následující rovnicí.

$$pozice_{n+1} = pozice_n - \eta \nabla f(pozice_n) \tag{2.16}$$

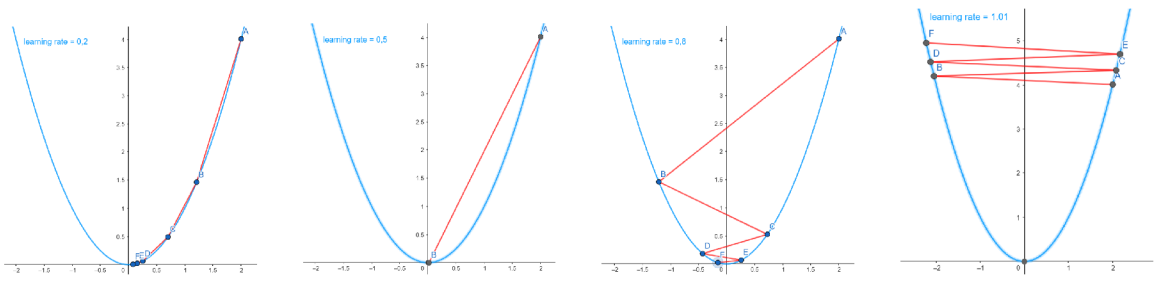
Písmeno η v rovnici znamená learning rate. Udává velikost kroku, kterým se bude sestup řídit. Malý krok může být výpočetně neefektivní, u sedlových funkcí může navíc najít nesprávné minimum. Sedlovou funkci lze vidět na obrázku 2.14. Na definičním oboru má dvě, nebo více minim. Pokud gradientní sestup začne na špatném místě, dopracuje se pouze

k prvnímu minimu, toto minimum nemusí být tím nejmenším. Naopak při velkém kroku se může stát, že funkce nebude konvergovat směrem dolů, dokonce se může stát, že začne být divergentní.

Následující příklad hledání nejmenší hodnoty pomocí gradientního sestupu v jazyce Python je ukázán pro jednoduchou jednorozměrnou funkci.

```
1 def gradient_descent(position, learning_rate, eps, max_iter, grad_fun):
2         for _ in range(max_iter):
3                 difference = learning_rate * grad_fun(position)
4                 if np.abs(difference) < eps:
5                         break
6                 position = position - difference
7         return position
```

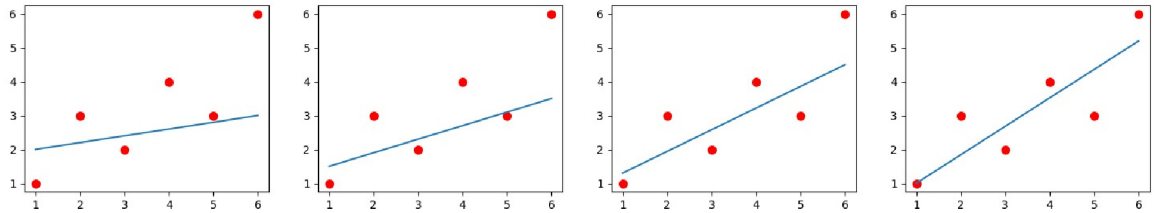
Změnou hodnoty learning rate je možné pozorovat různé chování gradientního sestupu, příklad je k nalezení na obrázku 2.16. Na prvním grafu lze vidět malý learning rate, to způsobí velké množství kroků. Druhý graf je ideální hodnotou, která dokáže najít minimum v jediném kroku. Další dva grafy mají příliš velký learning rate, první z nich extrém přeskočí a musí se k němu vracet, druhý diverguje.



Obrázek 2.16: Vliv learning rate na gradientní sestup.

Po nalezení minima pomocí výše popsaného sestupu, jsou upraveny parametry neuronové sítě, dle nalezené hodnoty za pomocí zpětné propagace, která je popsána v sekci 2.4.

Klasický gradientní sestup má v neuronových sítích zásadní problém. Jeho využití je výpočetně i časově náročné. Je to z toho důvodu, že pro jednu úpravu parametrů, a tedy i změnu výstupu neuronové sítě musí být vyhodnocena celá datová sada. Jako příklad může být neuronová síť, která provádí lineární regresi využívající datovou sadu o šesti bodech. Na následujícím obrázku 2.17 je možné pozorovat čtyři kroky učení. V každém kroku byla vyhodnocena celá datová sada, tedy všech šest bodů. Bylo nalezeno optimální minium pomocí gradientního sestupu a dle minima byly upraveny parametry sítě.

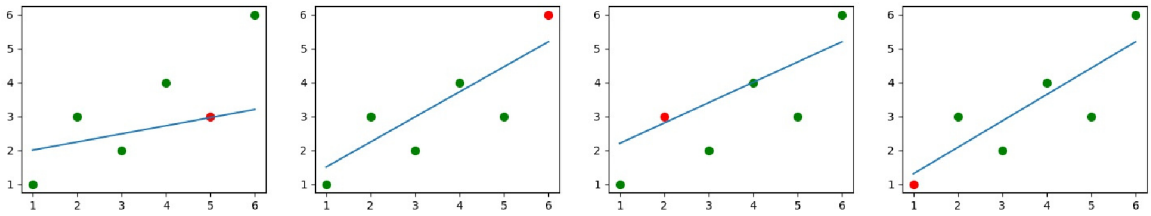


Obrázek 2.17: Ukázka čtyř úprav parametrů sítě za pomocí gradientního sestupu s využitím všech bodů datové sady.

U složitějších a hlubších neuronových sítí to znamená velmi vysoký počet průchodů a výpočtů, které jsou mnohdy redundantní, jelikož jsou vyhodnocovány stále dokola. Z tohoto důvodu vznikly speciální upravené verze [30].

Stochastický gradientní sestup (Stochastic gradient descent)

Tento typ gradientního sestupu nevyužívá k jedné úpravě celou datovou sadu. Vždy náhodně vybere jeden vzorek dat, ke kterému se neuronová síť bude snažit přiblížit.



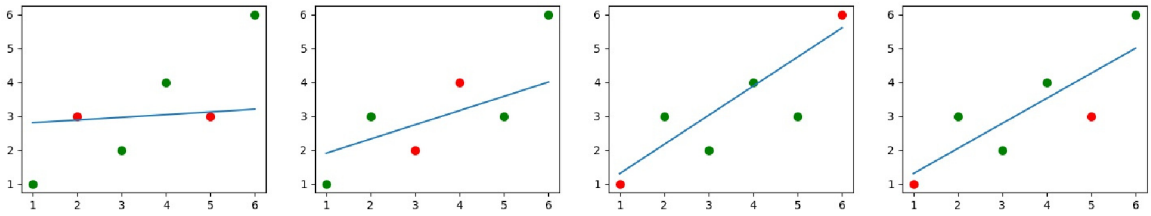
Obrázek 2.18: Ukázka čtyř úprav parametrů sítě za pomocí stochastického gradientního sestupu.

Na obrázku 2.18 výstupu v průběhu učení je možné pozorovat mnohem větší šum. Je to z toho důvodu, že vybraný vzorek datové sady nemusí mít požadovaný efekt. To znamená, že vybraný bod může být od ostatních přílišným extrémem. Učení k takovému bodu může způsobit, že síť se učí špatným hodnotám. V dalším kroku se bude od naučených hodnot opět vracet zpět. To způsobí, že nebude dostatečně schopný najít nejlepší minimum. To se dá vyřešit pomocí automatické úpravy learning rate.

I přes tuto nevýhodu se jedná o dostatečný odhad gradientního sestupu.

Mini-batch gradientní sestup

Mini-batch gradientní sestup by se dal považovat za kombinací klasického a stochastického sestupu. Jde o jistý kompromis mezi výpočetní a statistickou efektivitou. Místo aby algoritmus využíval celé datové sady, jako to je u klasické verze, využívá mini dávek (mini-batch). Nejedná se o nic jiného než o určité množství dat, na které je datová sada rozdělena. Algoritmus poté provede úpravu po vyhodnocení každé takové várky.



Obrázek 2.19: Ukázka čtyř úprav parametrů sítě za pomocí mini-batch gradientního sestupu s velikostí dávky 2.

Velikost dávky je možné si libovolně zvolit. Například na obrázku 2.19 byla dávka zvolena na hodnotu dva. U různých velikostí várky se vlastnosti liší. Větší velikosti chováním odpovídají spíše klasickému gradientnímu sestupu, mají vyšší výpočetní náročnost. Naopak menší dávky se chovají podobně jako stochastická verze. Ideální velikost dávky neexistuje, vždy záleží na získané datové sadě.

Zpětná propagace (Backpropagation)

Jedná se o metodu představenou již v osmdesátých letech minulého století. Jejím cílem je za pomoci chybové funkce získat gradienty, které využije gradientní sestup. Její velký rozvoj ovšem nastal až v roce 1986, kdy David Rumelhart, Geofrey Hinton a Ronald Wiliams představili studii [31], kde popsali, že zpětná propagace pracuje mnohem rychleji než předešlé metody.

Zpětná propagace pracuje s chybovými funkcemi, které byly popsány výše v 2.4. Musí platit, že chybová funkce je funkcí výstupu neuronové sítě. Zároveň tato funkce musí být aritmetickým průměrem přes jednotlivé odchylky mezi získanými a požadovanými výstupy položek datové sady. Například taková funkce MSE [2.10] zcela jistě obě tyto podmínky splňuje.

Tato sekce je sepsána na základě druhé kapitoly „How the backpropagation algorithm works“ jíž výše zmíněné knihy „Neural Networks and Deep Learning“ [21] a na základě článku z výukové webové stránky Brilliant [4].

Jelikož se zde vyskytuje mnoho matematických rovnic, je potřebné vysvětlit konvence, které se zde budou vyskytovat. Budou používané následující proměnné.

Tabulka 2.1: Proměnné používané v sekci.

Proměnná	Popis
δ_i^l	Chyba neuronu i na vrstvě l.
a_i^l	Výsledek součtu vstupů vynásobených váhami a posunu, který je vstupem aktivační funkce na neuronu i vrstvy l, neboli vážený vstup.
b_i^l	Hodnota posunu (biasu) na neuronu i vrstvy l.
w_{ij}^l	Hodnota váhy pro i-tý neuron l vrstvy, který je propojen z neuronu j vrstvy l-1
o_i^l	Výstup neuronu i na vrstvě l.
σ	Aktivační funkce.
r^l	Počet neuronů ve vrstvě l.

Pro lepší pochopení, jaké mají mezi sebou vztahy, takhle se vypočítá výstup jednoho neuronu v síti.

$$\begin{aligned} o_i^l &= \sigma(\sum_{k=1}^{r^{l-1}} w_{ik}^l o_k^{l-1} + b_i^l) \\ a_i^l &= \sum_{k=1}^{r^{l-1}} w_{ik}^l o_k^{l-1} + b_i^l \\ o_i^l &= \sigma(a_i^l) \end{aligned} \tag{2.17}$$

Pro zjednodušení předpokládejme, že hodnoty biasů jsou implementovány způsobem, kdy předchozí vrstva má jeden neuron navíc, který vždy vrací hodnotu 1 a posun udává váha propojení mezi tímto neuronem a následující vrstvou.

$$\begin{aligned} b_i^l &= w_{0i}^l \\ a_i^l &= \sum_{k=1}^{r^{l-1}} w_{ik}^l o_k^{l-1} + b_i^l = \sum_{k=0}^{r^{l-1}} w_{ik}^l o_k^{l-1} \end{aligned} \tag{2.18}$$

Na začátek je nutné definovat chybu na jednom neuronu. Výsledná chyba je derivace chybové funkce dle hodnoty váženého vstupu neuronu.

$$\delta_i^l = \frac{\partial L}{\partial a_i^l} \tag{2.19}$$

Bude se chtít hledat chyba na jednotlivých vahách, a proto se derivuje funkce vzhledem ke hledané váze.

Jak ale derivovat chybovou funkci? Je možné využít vlastností derivací. Chybová funkce je vždy aritmetickým průměrem přes chyby vzorků datové sady. Z vlastností derivací vyplývá, že derivace součtu je stejná jako součet derivací funkcí. Pro příklad je použitá funkce MSE [2.10].

$$\frac{\partial L(X,\theta)}{\partial w_{ij}^l} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial w_{ij}^l}(y_i - \hat{y}_i)^2 = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial L_n}{\partial w_{ij}^l} \tag{2.20}$$

To dává možnost derivovat chybu jednotlivých vzorků. Při použití řetězového pravidla u derivací se získá následující rovnice.

$$\frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial a_j^l}\frac{\partial a_j^l}{\partial w_{ij}^l} \tag{2.21}$$

První část součinu byla již popsána v rovnici 2.19, jde o chybu neuronu. Při výpočtu druhé části se derivuje rovnice 2.18. Derivace vypadá následovně.

$$\frac{\partial a_j^l}{\partial w_{ij}^l} = \frac{\partial}{\partial w_{ij}^l}(\sum_{k=0}^{r^{l-1}} w_{ik}^l o_k^{l-1}) = o_i^{l-1} \tag{2.22}$$

Po dosazení lze pozorovat, že derivace váhy na neuronu je chyba neuronu vynásobená hodnotou výstupu předchozí vrstvy. To je celkem logické vzhledem k funkci neuronové sítě. Data se šíří napříč vrstvami a každá z vrstev modifikuje data s chybou.

$$\frac{\partial L}{\partial w_{ij}^l} = \delta_j^l o_i^{l-1} \tag{2.23}$$

Z rovnice je již patrné proč se mluví o zpětné propagaci. Neuronová síť nejprve provede dopředný průchod tak, aby získala na výstupní vrstvě data. Z těchto dat je vypočítána chyba na neuronech poslední vrstvy. Pomocí této chyby se následně dopočítává iteračním algoritmem každá z předchozích vrstev.

Výpočet chyby výstupní vrstvy

Algoritmus začíná na poslední vrstvě neuronové sítě, která je označena v rovnicích pomocí velkého L. Každý výstupní neuron se dosadí do chybové funkce.

$$L = (\sigma(a_i^L) - \hat{y}_i)^2 \tag{2.24}$$

Z této rovnice je vypočítána chyba neuronu výstupní vrstvy.

$$\begin{aligned} \delta_i^L &= \frac{\partial L}{\partial a_i^l} \\ \delta_i^L &= 2(\sigma(a_i^L) - \hat{y}_i)\sigma'(a_i^L) \end{aligned} \tag{2.25}$$

Poté je dosazeno do rovnice 2.23 a je vypočítána chyba vah neuronů. Výstup předchozí vrstvy sítě je znám z dopředného průchodu.

$$\frac{\partial L}{\partial w_{ij}^{L}} = 2(y_i - \hat{y}_i)\sigma'(a_i^L)o_i^{L-1} \tag{2.26}$$

Výpočet chyby skrytých vrstev

Při výpočtu chyby předešlých vrstev je opět využito derivační řetězové pravidlo. Následuje dosazení prvního členu za chybu následující vrstvy, která již byla vypočtena v předešlém kroku.

$$\begin{aligned}
\delta_i^l &= \frac{\partial L}{\partial a_i^l} \\
\delta_i^l &= \sum_{k=1}^{r^{l+1}} \frac{\partial L}{\partial a_k^{l+1}} \frac{\partial a_k^{l+1}}{\partial a_i^l} \\
\delta_i^l &= \sum_{k=1}^{r^{l+1}} \delta_i^{l+1} \frac{\partial a_k^{l+1}}{\partial a_i^l}
\end{aligned} \tag{2.27}$$

Nyní je potřeba vypočítat druhou část součinu. Pro něj se využije vzorec pro výpočet váženého vstupu do aktivační funkce neuronu 2.18. Jeho derivováním se získá požadovaný součinitel.

$$\begin{aligned}
a_k^{l+1} &= \sum_{i=1}^{r^l} w_{ik}^{l+1} o_k^l \\
a_k^{l+1} &= \sum_{i=1}^{r^l} w_{ik}^{l+1} \sigma(a_i^l) \\
\frac{\partial a_k^{l+1}}{\partial a_i^l} &= w_{ik}^{l+1} \sigma'(a_i^l)
\end{aligned} \tag{2.28}$$

Výsledek je dosazen do rovnice 2.27. Z rovnice je možné vytáhnout derivaci aktivační funkce před sumu, jelikož na ní není závislá. Výsledkem je chyba neuronu na jedné ze skrytých vrstev.

$$\begin{aligned}
\delta_i^l &= \sum_{k=1}^{r^{l+1}} \delta_i^{l+1} w_{ik}^{l+1} \sigma'(a_i^l) \\
\delta_i^l &= \sigma'(a_i^l) \sum_{k=1}^{r^{l+1}} \delta_i^{l+1} w_{ik}^{l+1}
\end{aligned} \tag{2.29}$$

Posledním krokem je výpočet gradientu váhy neuronu skryté vrstvy. Zde je dosazeno do již známe rovnice 2.23.

$$\frac{\partial L}{\partial w_{ij}^l} = \delta_j^l o_i^{l-1}$$
$$\frac{\partial L}{\partial w_{ij}^l} = \sigma'(a_i^l) o_i^{l-1} \sum_{k=1}^{r^{l+1}} \delta_i^{l+1} w_{ik}^{l+1} \tag{2.30}$$

Souhrn učení sítí

Neuronové sítě jsou učené na datových sadách. Ta obsahuje vzorky dat, přičemž každý vzorek je dvojice vstupu sítě a požadovaný výstup. Také je definovaná chybová funkce, která je schopná vypočítat rozdíl mezi výstupem a očekávaným výstupem.

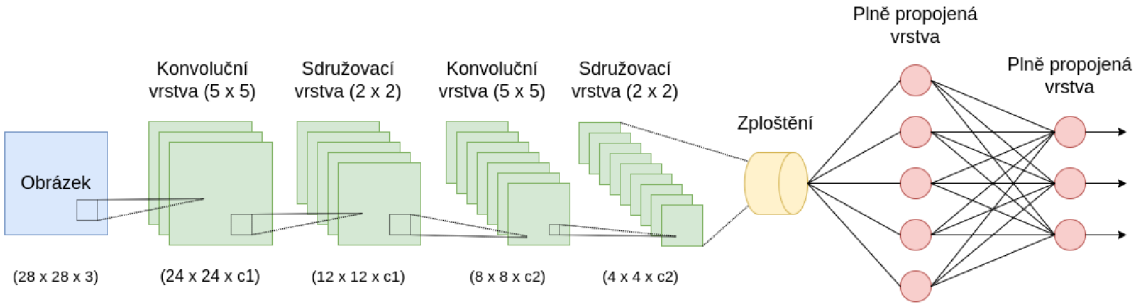
Vstupní data jsou přivedena na první vrstvu sítě a je proveden průchod. Tento průchod se označuje jako dopředný a slouží k získání potřebných informací (výstupů neuronů) pro zpětnou propagaci.

Jakmile se data objeví na výstupní vrstvě, je čas na zpětný průchod. Ten se skládá z několika kroků. Nejdříve je vypočítána chyba na neuronech poslední vrstvy. Jakmile je informace získána, je vypočítána chyba mezi poslední a předposlední vrstvou. Pro výpočet se použije derivace chybové funkce. Následuje výpočet vrstvy předposlední, k ní je nutné znát chybu poslední vrstvy. Poté je i zde vypočítána chyba pro skryté vrstvy. Veškeré vzorce byly představeny výše. Stejným způsobem se chyba propaguje přes všechny vrstvy zpět.

Poté co známe chyby neboli gradienty vah, je použit gradientní sestup, nebo pokročilé algoritmy jako například Adam [18], které z něj vychází. Ty nastavují nové lepší hodnoty vah v síti za pomoci gradientů získaných zpětnou propagací. Pokud se jedná o stochastický sestup, váhy jsou upraveny ihned. U klasického nebo mini-batch gradientního sestupu je vyžadováno více vzorků. V tomto případě se zapamatují hodnoty gradientů a je vyhodnoceno několik dalších průchodů. Jakmile je dosažen nastavený počet, jsou získané gradienty pro každou váhu zprůměrovány a je provedena úprava.

2.5 Konvoluční neuronové sítě (Convolutional Neural Networks - CNN)

Jedná se o speciální typ neuronových sítí využívaných tam, kde jsou zapotřebí zpracovávat obrázky a jiná velká data. I když i běžná neuronová síť dokáže natrénovat malé obrázky, u větších rozlišení vzniká problém. Nejsou tam tak jasné závislosti mezi pixely obrázku. Proto nastupují na řadu konvoluční neuronové sítě [36], které tyto závislosti dokáží zachytit.



Obrázek 2.20: Příklad návrhu konvolučních neuronových sítí.

Architektura [25] tohoto typu neuronové sítě popsaná na obrázku 2.20 se vždy skládá ze dvou základních částí. V první části jsou využívány speciální konvoluční vrstvy s pomocí sdružovacích (pooling) vrstev. Následně jsou výstupy zploštěny a jsou vloženy do druhé části. Ta je složena z plně propojených (fully connected) vrstev.

Tato sekce popisuje problematiku na dvou dimenzionálních konvolučních sítí, které jsou stěžejní pro zpracování výstupu kamerových systémů automobilu. Jedno a více dimenzionální konvoluční sítě pracují na obdobném principu.

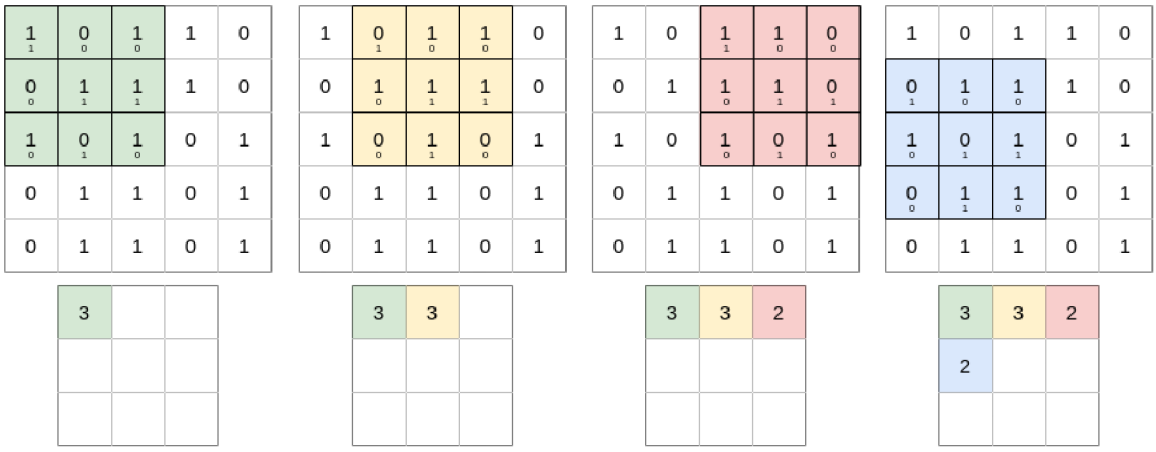
Na vstupu těchto sítí je vždy obrázek. Jedná se o tří dimenzionální tenzor (vícerozměrné pole), typicky popsán za pomoci RGB formátu. Říká, že obrázek má tři kanály, každý z těchto kanálů obsahuje hodnoty pixelů (výška x šířka) pro jednu z hlavních barev (červená, zelená, modrá) v rozsahu 0-255. Ten je vložen do první konvoluční vrstvy.

Konvoluční vrstva (Convolutional layer)

Konvoluční vrstva převezme vstup a pro každý kanál je vytvořeno dvou dimenzionální jádro (kernel) o předem definované velikosti, typicky 5x5, nebo 3x3. U konvolučních vrstev se využívá ReLU aktivační funkce.

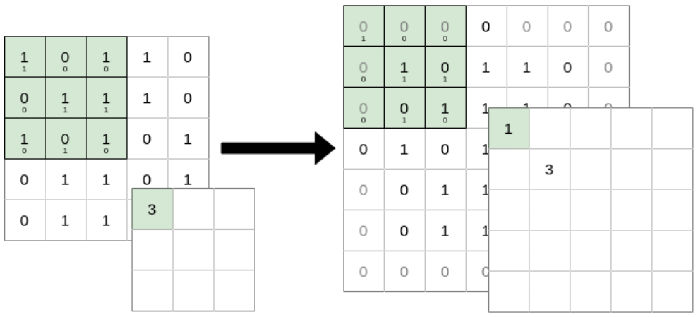
Obsah kernelu je vyplněn hodnotami vah, které se neuronová síť naučila během tréninku. Jádro je přiloženo k obrázku na první pozici. Hodnoty vah a pixelů jsou mezi sebou vynásobeny a tyto násobky sečteny. Výsledkem je jedno číslo, které je vloženo na první pozici výstupu vrstvy. Jádro je s předem daným krokem posunuto na druhou pozici doleva. Kroku se anglicky říká *stride*, ten může nabývat dvou podob. Buďto je krok jedno číslo, které vyjadřuje velikost pohybu po sloupcích i řádcích, nebo může být definován pro každou osu zvlášť. Jakmile je jádro na posledním sloupci, přeskakuje na další řádek tak dlouho, dokud nedojde na konec obrázku.

Obrázek 2.21 popisuje první čtyři kroky posunu konvolučního jádra s velikostí kroku 1x1.



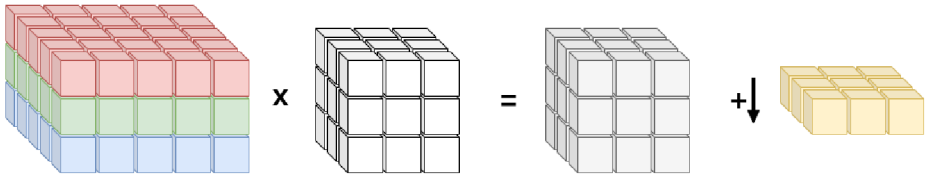
Obrázek 2.21: Ukázka výpočtu výstupu konvoluční vrstvy s krokem o velikosti 1.

Jádro funguje jako filtr, který v obrázku dokáže najít objekty, hrany a další závislosti dle naučených hodnot jádra. Způsobuje, že výstup vrstvy je menší než její vstup. Pokud by byla chtěna velikost výstupu stejně velká jako velikost vstupu, muselo by být využito vycpávky (padding). Jde o přidání sloupců a řádků kolem vstupní matice, ty jsou nastaveny na hodnotu nula. Poté má výstup stejnou velikost jako vstup. Ukázku je možné pozorovat na obrázku 2.22.



Obrázek 2.22: Využití vycpávky vstupu.

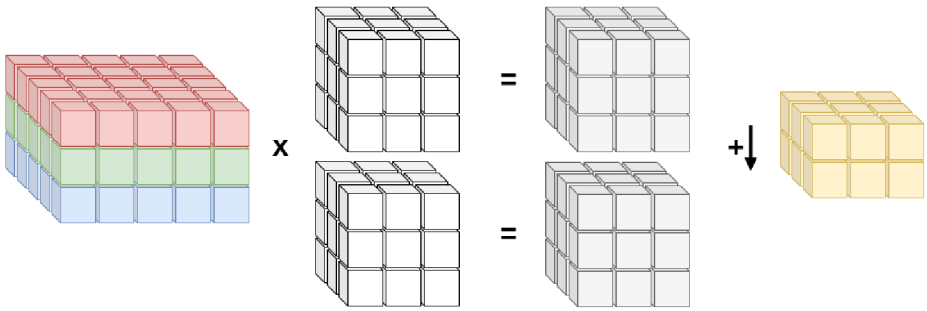
Předchozí příklady braly v potaz pouze jeden vstupní kanál. Je ale jasné, že vstupní obrázek bude mít kanály tři. Následující konvoluční vrstvy mohou pracovat i s vyšším počtem. V tomto případě je filtr vytvořen pro každý ze vstupních kanálů. Místo jedné matice je výstupem filtru matic tolik, kolik je vstupních kanálů, tyto matice jsou mezi sebou sečteny tak, aby na výstupu vznikla pouze jediná matice. Hloubka výstupu bude rovna jedna, viz. obrázek 2.23.



Obrázek 2.23: Funkce konvoluční neuronové sítě s více vstupními kanály.

Díky součtu všech vstupních kanálů, dokáže konvoluční vrstva vyhledávat souvislosti mezi všemi kanály najednou.

Je možné využít konvoluční vrstvu tím způsobem, aby na svém výstupu generovala více než jeden kanál. To je docíleno pomocí více sad filtrů. Například pokud by vrstva očekávala na vstupu tři kanály a na výstupu vracela kanály dva, budou pro každý vstupní kanál vytvořeny dva filtry, dohromady tedy šest jader jako na obrázku 2.24.



Obrázek 2.24: Funkce konvoluční neuronové sítě s více vstupními i výstupními kanály.

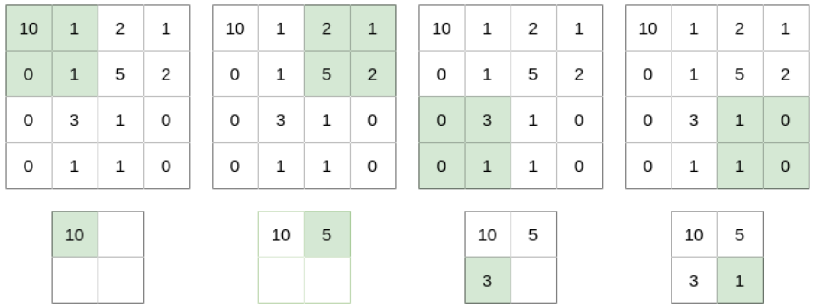
Za zmínku stojí speciální typ konvoluční vrstvy s velikostí jádra 1x1 a krokem jedna. Tato vrstva dokáže redukovat hloubku (počet kanálů) tak, že hledá souvislosti jen mezi nimi a není ovlivněna okolními daty. Neovlivňuje výšku ani šířku dat. K ovlivnění výšky a šířky slouží vrstva sdružovací.

Sdružovací vrstva (Pooling layer)

Za konvoluční vrstvou se často nachází sdružovací vrstva, která provádí sdružení (pooling) za účelem zmenšení výšky a šířky dat. Principiálně je dost podobná konvoluční vrstvě. I zde je k nalezení jádro o definované velikosti, které se posouvá po jednotlivých kanálech vstupu. Tato vrstva ovšem vždy vrátí stejnou hloubku, jakou dostane na svém vstupu. Existují dva typy vrstev.

Max-pooling

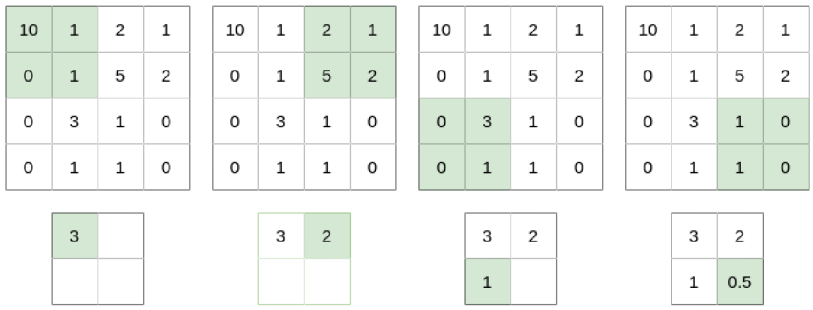
Přiloží jádro k jednomu z kanálů. Z hodnot kanálu daných velikostí přiloženého jádra vybere nejvyšší hodnotu a tu nastaví jako hodnotu výstupu. To udělá pro všechny posuny na všech kanálech. Na následující ukázce 2.25 je velikost jádra 2x2 s krokem velikosti dva.



Obrázek 2.25: Ukázka výběru maximální hodnoty z jednoho kanálu.

Average-pooling

Na rozdíl od hledání nejvyšší hodnoty tento typ provede aritmetický průměr z hodnot definovaných velikostí jádra. Ukázka na obrázku 2.26.



Obrázek 2.26: Ukázka výpočtu průměrné hodnoty z jednoho kanálu.

Plně propojené vrstvy (Fully connected layers)

Jakmile jsou obrázky či jiná vstupní data dostatečně zpracovány konvolučními vrstvami a z nich vyfiltrována důležitá data, mohou být zploštěny z více dimenzí na vektor čísel, které jsou poslány na vstupní vrstvu lineární neuronové sítě, která již byla popsána na začátku kapitoly. Každý neuron jedné vrstvy je propojen na každý neuron vrstvy následující. Na

propojeních jsou definovány váhy, posuny a aktivační funkce. Výstup poslední z vrstev je výstupem celé sítě. Její výstup odpovídá problému, který síť řeší.

2.6 Souhrn neuronových sítí pro autonomní vozidla

Autonomní automobily, jako hlavní řídící mechanismus využívají natrénované modely neuronových sítí. Tvorba takového modelu závisí na vstupech a řešeném problému. Vozidla, jako vstupy používají data ze senzorů. Osazení senzorů se u každého automobilu může lišit v závislosti na finančních prostředcích nebo cílech. Navíc každý senzor má definovaný jiný formát výstupu. To jde pozorovat v následující kapitole 3.3. Vede to i na rozdílné návrhy neuronových sítí.

V dnešní době jde pro implementaci využít mnoho programovacích jazyků. Tím nejtypičtějším je programovací jazyk Python. Ten má k dispozici více knihoven, díky kterým je možné jednoduše stavět modely neuronových sítí. Tvoří se jako definice neuronových vrstev a aktivačních funkcí, které se skládají do jednoho celku. Tvorba jednotlivých neuronů je zcela pod kontrolou knihovny. Knihovna je optimalizována k maximálnímu výkonu, to znamená že jsou psány v nižších jazycích jako C nebo C++. Krom toho poskytují nástroje k jejich trénování. Jsou to například implementace pro optimalizační algoritmy jako Adam nebo stochastický gradientní sestup. Nástroje pro čtení datové sady po várkách a implementace chybových funkcí.

Příklady knihoven pro Python jsou například TensorFlow [1] od společnosti Google. Nadstavbou nad touto knihovnou je Keras [12]. Novou knihovnou nabírající na popularitě je PyTorch [26] od Facebook skupiny zabývající se umělou inteligencí. Poslední ze zmíněných knihoven byla použita k implementaci této práce.

Kapitola 3

CARLA žebříček a simulátor autonomního řízení

Simulátory vývojářům umožňují snadný způsob, jak vyvinout a otestovat model neuronových sítí. Plno situací není možné natrénovat v reálném provozu. Důvodů je hned několik. Buď je pro výzkumné skupiny tvorba fyzického auta a testovací dráhy příliš nákladná, nebo jsou trénované situace nebezpečné (vkročení dítěte před automobil). Simulátory mají mnoho typů pozemních komunikací. Jsou to například města, příměstské oblasti, dálnice, venkov a jiné. K nalezení jsou i dopravní scénáře, těch je velké množství od průjezdu křižovatkou po vkročení chodce do silnice před projíždějící automobil.

Simulátory je možné přirovnat k počítačovým hrám. Stejně jako ony obsahují cesty, domy, křižovatky a další. V tomto prostředí se pohybují nehráčské postavy (anglicky non-player characters - NPC) jako chodci, cyklisté nebo automobily. Hráčem v této simulaci je samotné autonomně řízené auto. Hry jsou tvořeny za jiným účelem než simulátory, mají uzavřený kód, neposkytují možnosti senzorů a mnohdy mají nerealistickou fyziku. Tyto nedostatky vedly ke vzniku simulátorů, jako je například CARLA simulátor [14].

Vývojářské týmy mohou využít CARLA žebříčku (leaderboard) [7]. Je to otevřená platforma, pomocí které je možné vyhodnocovat své implementace autonomních agentů. Autonomním agentem je myšlen program, který řídí automobil v simulátoru CARLA. Týmy mají přímé porovnání, jak je jejich agent dobrý vzhledem k ostatním skupinám. Vyhodnocení probíhá ve dvou fázích. První fází je vyhodnocení v průběhu tvorby agenta na počítačích vývojářského týmu. K tomu CARLA poskytuje potřebné programy. V další fázi se agent odesílá na servery CARLA žebříčku. Tam je autonomní agent otestován na předem neznámých trasách, situacích a povětrnostních podmínkách.

3.1 Cíle CARLA žebříčku

Cíle jsou ve své podstatě velmi jednoduché. Agent musí projet sadu předem definovaných tras, přičemž trasy jsou popsány v XML dokumentu. Každá trasa obsahuje první startovní bod, zde CARLA leaderboard umístí automobil a inicializuje agenta. Agent musí dorazit do cílového (posledního bodu), do tohoto bodu je navigován přes sérii dalších pozic.

Kromě popisu bodů cesty je možné vidět i definice identifikátoru a města, ve kterém se trasa nachází. Navíc je každé cestě možné nastavit specifické počasí. To je složené z vlastností jako oblačnost, množství srážek, intenzita větru, hustota mlhy, vlhkost a postavení slunce.

Popis jedné z tras může vypadat následovně.

```
<route id="0" town="Town01">
<weather
  cloudiness="0" precipitation="0" precipitation_deposits="0"
  wind_intensity="0" sun_azimuth_angle="0" sun_altitude_angle="70"
  fog_density="0" fog_distance="0" wetness="0" />

<waypoint pitch="360.0" roll="0.0" x="338.7" y="22.7" yaw="269.9" z="0.0"/>
...
<waypoint pitch="360.0" roll="0.0" x="1.3" y="47.9" yaw="269.8" z="0.0"/>
</route>
```

Vedle cest je nutné definovat scénáře. Ty jsou k nalezení v JSON dokumentu. Scénáře je možné definovat pro každé město. Součástí definice je místo a orientace na mapě, kde se scénář objeví. Krom toho jsou definovaní i "jiní herci", ty označují jiné chování objektů ve scénáři.

Možnosti scénářů

Součástí žebříčku jsou soubory, které obsahují přesné definice tras, ale i scénářů. Tvorba souboru se scénáři není jednoduchá tak, jako v případě cest. Z toho důvodu existuje soubor, ve kterém jsou popsané veškeré možné scénáře. Ty mohou být rozděleny hned do desíti kategorií.

Scénář 1 - Ztráta kontroly

Agent ztratí kontrolu nad řízením z důvodu špatných podmínek, musí se zotavit a vrátit se do původního směru. Špatnými podmínkami může být myšlena například znečištěná vozovka.

Scénář 2 - Nouzové zastavení

Automobil musí provést nouzové brzdění poté, co předchozí automobil začne prudce zastavovat kvůli neočekávané překážce v cestě.

Scénář 3 - Vyhnutí se překážce bez předchozí akce

Agent musí reagovat na překážku v cestě za pomocí brzdění nebo úhybného manévru. Automobil před touto překážkou neprováděl žádnou předchozí akci jako změnu směru, průjezd křižovatkou a jiné. Překážkou může být myšlen člověk, zvíře nebo jiný objekt.

Scénář 4 - Vyhnutí se překážce s předchozí akcí

Jedná se o stejnou situaci jako vyhnutí se překážce bez předchozí akce. Jediným rozdílem je, že před touto překážkou automobil provede složitější akci. Typickým příkladem může být průjezd křižovatkou, kdy automobil po odbočení musí reagovat na chodce na přechodu na úrovni křižovatky.

Scénář 5 - Změna jízdního pruhu

Automobil změní jízdní pruh na silnici s více pruhy. Změna byla provedena z důvodu, že automobil v čele jede příliš pomalu.

Scénář 6 - Objetí překážky protijedoucím pruhem

Autonomní agent musí objet překážku nebo automobil. Tento scénář se nachází na silnici s jedním jízdním pruhem v každém směru. Aby bylo možné objetí provést, musí přejet do protisměru. V protisměru je normální provoz a musí vyhodnotit, zda nedojde ke srážce s protijedoucím vozidlem.

Scénář 7 - Reakce na špatně jedoucí vozidlo křižovatkou

Automobil v tomto scénáři projíždí rovně křižovatkou na zelenou. Všechny ostatní směry mají červenou a nesmí projet. I přes to některý z ostatních automobilů vjede do křižovatky. Agent musí zareagovat a zabránit srážce uprostřed křížení.

Scénář 8 - Zatočení doleva na křižovatce

Jedná se o standardní průjezd křižovatkou, kdy agent dostane zelenou na semaforu. Snaží se odbočit doleva, musí tedy vyhodnotit pruh z protisměru a dát přednost těmto automobilům, jakmile je pruh volný, může odbočit.

Scénář 9 - Zatočení doprava na křižovatce

Jednoduché zatočení doprava na světelné křižovatce.

Scénář 10 - Projetí křižovatky bez značení

Jedná se o projetí neoznačené křižovatky. Agent musí vyhodnotit, zda je bezpečné křižovatkou projet. Při tomto scénáři platí, že přednost má automobil, který vstoupil do křížení jako první.

Metrika hodnocení

Pro ohodnocení schopnosti agenta plnit úkoly musel CARLA leaderboard zavést metriku. Ta jednoznačně vypočítá, jak moc dobrý daný agent je. Pomocí metriky také dochází k řazení týmu v žebříčku. Metrika je součinem dvou částí.

$$driving\_score = R_i P_i \tag{3.1}$$

R_i v rovnici určuje procento trasy, které agent zvládl zdolat. Pokud některou část trati zdolal agent mimo vytyčenou trasu, je tato část odečtena od výsledného R_i. Například když dorazil úspěšně do cíle a 10% trasy neodpovídá plánu, výsledek je 90 %.

P_i určuje trest za přestupek. Na začátku má přestupek hodnotu jedna. S každým proviněním je tato hodnota snižována dle rovnice.

$$P_i = \prod_{j}^{ped,\ldots,stop} (p_i^j)^{\#infractions_j} \tag{3.2}$$

Žebříček hodnotí těchto pět různých provinění, přičemž každé z nich má rozdílnou váhu dle závažnosti. Horší provinění mají nižší hodnoty blíže 0, snižují více výsledek, ty lehčí mají váhu blíže hodnotě 1.

Tabulka 3.1: Typy provinění CARLA žebříčku.

Název	Váha
Kolize s chodci	0,50
Kolize s jinými vozidly	0,60
Kolize se statickými objekty	0,65
Projetí semaforu na červenou	0,70
Projetí značky stop	0,80

Příklad výpočtu celkového provinění je následující. Předpokládá se, že automobil dorazil do cílového bodu. Přitom bylo 15 % z celkové cesty ujeto mimo vyznačenou trasu. Na trase automobil zaznamenal v následujícím pořadí kolizi s chodcem, kolizi se statickým objektem a na závěr projel na červenou.

Nejprve je vypočítáno celkové procento zvládnuté trasy, výsledkem je 85 %.

$$\begin{aligned} R_i &= 100 - 15 \\ R_i &= 85\% \end{aligned} \tag{3.3}$$

Poté je získána hodnota penalizace za provinění. Hodnoty jsou mezi sebou násobeny, na začátku je jako výchozí hodnota 1.

$$\begin{aligned} P_i &= 1 * 0,5 * 0,65 * 0,70 \\ P_i &= 0,2275 \end{aligned} \tag{3.4}$$

V posledním kroku jsou vynásobené hodnoty penalizací a procenta zdolané cesty. Agent je ohodnocen hodnotou 19,3375.

$$\begin{aligned} driving\_score &= 85 * 0,2275 \\ driving\_score &= 19,3375 \end{aligned} \tag{3.5}$$

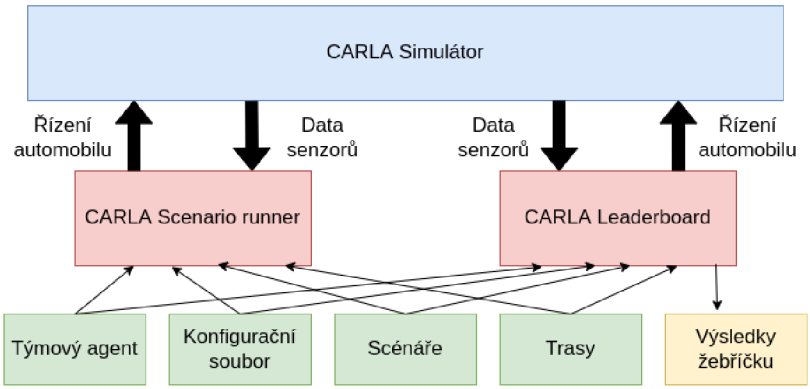
Kromě těchto porušení existují i další, která přeruší celou validaci trasy. Je to odklonění od trasy o více než 30 metrů. Zablokování agenta, kdy agent nevykoná žádnou akci po dobu tří minut. Simulační časový limit je nastaven na dvě minuty. Pokud po tuto dobu není možné navázat spojení mezi serverem a klientem, simulace končí. Posledním je časový limit cesty, ten přeruší hodnocení, pokud zvládnutí trasy trvá příliš dlouho.

Hodnocení agenta probíhá na více simulacích, typech scénářů a cest. Pokud jsou úspěšně dokončeny veškeré cesty, celkový výsledek se vypočítá jako aritmetický průměr všech pokusů.

3.2 Prostředky pro vývojáře CARLA žebříčku

CARLA poskytuje spoustu nástrojů, které vývojářům pomáhají při tvorbě programů. Základním prvkem, se kterým se pracuje je CARLA simulátor ve verzi 0.9.10.1. Nutné jsou

i balíčky Leaderboard a Scenario runner [9]. V těchto balíčcích jsou programy a implementace autonomních agentů. Součástí nich jsou i definice různých tras a scénářů. Kromě toho obsahují programy, které pomocí síťových socketů komunikují se simulátorem. Obrázek 3.1 popisuje komunikaci mezi moduly.



Obrázek 3.1: Komunikace mezi moduly CARLY.

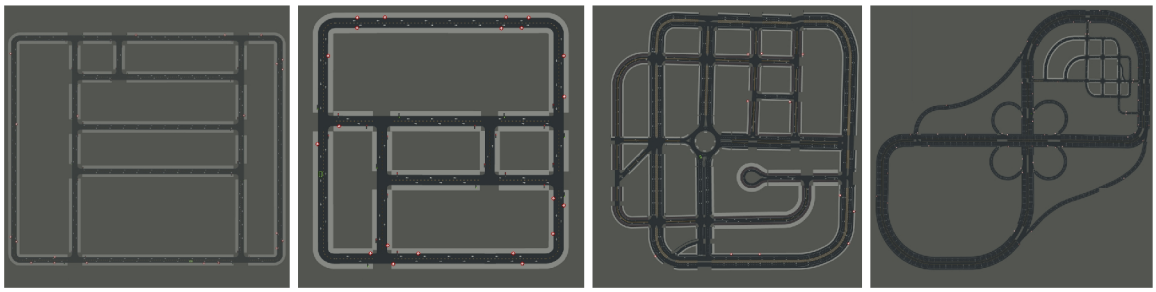
CARLA simulátor

Jedná se o open-source (kdokoli může číst a modifikovat zdrojový kód) klient-server projekt postavený na herním Unreal Engine 4. Server v dané architektuře vykresluje prostředí a simuluje dění. Má aplikační rozhraní, které komunikuje s klientem.

Klient je označován jako agent, ten posílá serveru hlavní čtveřici příkazů. Je to informace o natočení kol (steer), velikost sešlápnutí plynu (throttle), sešlápnutí brzdy (brake) a informace o stavu ruční brzdy (hand brake). Součástí může být i informace, zda automobil používá automatickou převodovku, pokud ne očekává i informaci o zařazeném stupni. Krom toho odesílá další meta příkazy, jako je například resetování simulace. Odpovědí od serveru jsou informace ze senzorů, které jsou definované na automobilu.

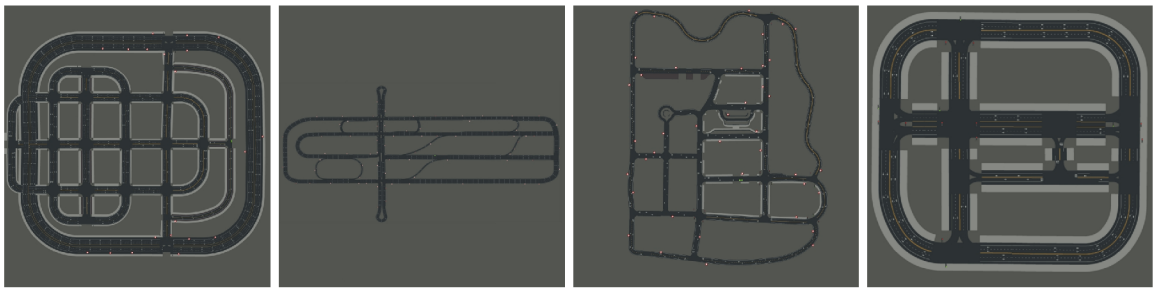
Pro implementaci práce je použita starší verze simulátoru. Ta obsahuje dohromady osm měst. Prvních pět je přímo jeho součástí. Poslední a nejnovější tři je nutné stáhnout zvlášť a naimportovat. Popis jak simulátor i s přídavky nainstalovat je v příloze B.1.

První dvě města označená jako *Town01* a *Town02* si jsou velmi podobná. Obě jsou jednoduchá a obsahují pouze křižovatky typu „T“. Město *Town03* je komplexnější. Má složitější křižovatky. Například takové, kde se kříží pět cest. Krom toho se auto musí vypořádat s kruhovými objezdy a tunely. Pro testování dálnic je připraveno město *Town04*.



Obrázek 3.2: Zleva města Town01, Town02, Town03 a Town04.

Město *Town05* má velké množství křižovatek. Zároveň cesty mají v každém směru více jízdních pruhů. Je vhodné pro učení změny pruhů. Dalším z připravených měst pro dálnice je *Town06*. Obsahuje dlouhé rovné cesty s větším množstvím sjezdů a nájezdů. Ve většině měst jsou rovné silnice. To napravuje *Town07*. Jeho dominantou jsou zakřivené cesty s malým množstvím semaforů. Posledním je město *Town10*. V něm je mnoho rozdílných prostředí, které jsou realističtější.



Obrázek 3.3: Zleva města Town05, Town06, Town07 a Town10.

Všechna města jdou upravovat, nebo tvořit úplně nová. Ukázky měst jsou na obrázcích 3.2 a 3.3. To stejné platí i o modelech dopravních prostředků. Simulátor neobsahuje pouze modely automobilů, ale i dvoukolových prostředků jako jsou motocykly a kola. Samozřejmostí jsou i chodci.

CARLA leaderboard

Balíček leaderboard dostupný z GitHubu obsahuje programy, které zaobalují kód, který tým vytvoří. Umožňuje nastavovat vlastnosti simulace jako je definice cest a scénářů. Obsahuje rodičovskou třídu *AutonomousAgent*, ze které dědí veškeré implementace agentů týmů. Skripty leaderboardu jsou prostředníky mezi agentem a simulátorem. Obsahuje klientský program, který naváže síťové spojení se serverem (simulátorem). Ledearboard také předzpracovává data senzorů ze simulátoru a předává je dále agentovi.

Před spuštěním leaderboardu, tedy klienta, musí dojít k zapnutí CARLA simulátoru. Jak klient, tak server ve výchozím nastavení komunikují na portu 2000. Žebříček je spouštěn skriptem *run_evaluation.sh*. Ten následně spustí program *leaderboard_evaluator.py*, který provede připojení k serveru a vyhodnocuje úspěšnost.

Spouštění vyhodnocování vypadá následovně.

```
1  export SCENARIOS=
2      ${LEADERBOARD_ROOT}/data/all_towns_traffic_scenarios_public.json
3  export ROUTES=${LEADERBOARD_ROOT}/data/routes_devtest.xml
4  export REPETITIONS=1
5  export DEBUG_CHALLENGE=1
6  export TEAM_AGENT=${LEADERBOARD_ROOT}/leaderboard/autoagents/human_agent.py
7  export CHECKPOINT_ENDPOINT=${LEADERBOARD_ROOT}/results.json
8  export CHALLENGE_TRACK_CODENAME=SENSORS
9
10 ./scripts/run_evaluation.sh
```

Před spuštěním se exportuje sedm proměnných, které jsou potřeba k definici hodnot, které leaderboard bude vyhodnocovat.

Tabulka 3.2: Popis proměnných pro spuštění programu žebříčku.

Proměnná	Popis
SCENARIOS	Cesta k JSON souboru s popisem veškerých scénářů, které se v průběhu řízení objeví.
ROUTES	Cesta k XML souboru s popisem cest.
REPETITIONS	Počet opakování, kolikrát se má každá z definovaných cest spustit.
DEBUG_CHALLENGE	Pokud je tato proměnná nastavena na 1, zobrazuje se v simulátoru cesta, kterou má vozidlo projet.
TEAM_AGENT	V Cesta k python souboru s definicí autonomního agenta.
CHECKPOINT_ENDPOINT	Cesta k souboru, do kterého se zapíšou výsledky vyhodnocování.
CHALLENGE_TRACK_CODENAME	Očekává hodnotu SENSORS, nebo MAP, tedy mód vyhodnocování viz. 3.3.

Žebříček vyhodnocuje veškeré věci popsané v sekci 3.1. Výstup výpočtu je vypsán na standardní výstup po dokončení každé z tras. Zároveň je výsledek uložen do definovaného souboru.

CARLA scenario runner

Jedná se o modul svou funkcí podobný samotnému žebříčku. Stejně jako leaderboard umožňuje spouštění vytvořených agentů na definovaných scénářích a trasách. Modul neposkytuje vyhodnocování trasy, a naopak umožňuje definici vlastních scénářů.

3.3 Návrh implementace autonomního agenta

Jak bylo zmíněno již výše, agenti všech týmů musí vycházet ze třídy *AutonomousAgent*. Pro správné fungování je nutné předefinovat některé metody dle potřeby. Kostra takového programu je následující.

```
1  class Agent(AutonomousAgent):
2      def setup(self, path_to_conf_file):
3          self.track = Track.SENSORS # Track.MAP
4
5      def sensors(self):
6          sensors = [ ... ]
7          return sensors
8
9      def run_step(self, input_data, timestamp):
10         return carla.VehicleControl()
11
12     def destroy(self):
13         pass
```

Metoda setup

Slouží k inicializaci všech potřebných věcí, které agent potřebuje k funkci. Týmy mohou tuto metodu využít pro své vlastní definice. Krom toho, je nutné nastavit jeden ze dvou módů, které žebříček dovoluje, ty jsou pojmenovány jako senzory, nebo mapa.

První způsob je využití pouze senzorů. Automobil při tomto použití nemá přístup k mapě trasy a naviguje se pomocí GPS souřadnic zeměpisné délky a šířky. Automobil dostává od CARLA simulátoru následující informace.

```
[
({'lat': -0.0020, 'lon': 0.0030, 'z': 0.0}, RoadOption.LANEFOLLOW),
({'lat': -0.0018, 'lon': 0.0030, 'z': 0.0}, RoadOption.LEFT),
]
```

Mód mapy přidává k možnostem senzorů využití OpenDrive mapy k navigování. Zde se místo GPS souřadnic využívají světové souřadnice X a Y.

```
[
({'x': 338.763153, 'y': 226.451157, 'z': 0.500000}, RoadOption.LANEFOLLOW),
({'x': 338.773590, 'y': 209.161163, 'z': 0.000000}, RoadOption.LEFT),
]
```

U obou módů jsou informace dostupné v proměnné *self._global_plan* pro GPS souřadnice a *self._global_plan_world_coord* pro světové souřadnice. Ty jsou v metodě *setup* nenastaveny, jejich použití je možné až v metodě *run_step*. Plán je vždy dvojicí hodnot, první hodnota je slovník s popisem, kde se cestovní bod nachází. Druhá hodnota je pak informace, jak se má agent v tomto bodě zachovat. Možností chování je hned několik a jsou popsány třídou *RoadOption*.

Tabulka 3.3: Popis chování cestovního bodu.

Název	Chování
RoadOption.LANEFOLLOW	Agent bude do následujícího bodu v pořadí sledovat jízdní pruh, ve kterém se nyní nachází.
RoadOption.LEFT	Agent na křižovatce provede odbočení doleva.
RoadOption.RIGHT	Agent na křižovatce provede odbočení doprava.
RoadOption.STRAIGHT	Agent na křižovatce pokračuje rovno.
RoadOption.CHANGELANELEFT	V tomto bodě se automobil přesune do levého jízdního pruhu.
RoadOption.CHANGELANERIGHT	V tomto bodě se automobil přesune do pravého jízdního pruhu.

V metodě *setup* je zadán jeden parametr *path_to_file*. V tomto parametru je od skriptů žebříčku předána cesta k souboru. Je to stejný soubor, který se nastavuje při spouštění žebříčku. Soubor nemá definovanou podobu, každý tým může tento soubor využít dle svých potřeb. Týmy zodpovídají za správné zpracování souboru. Typicky zde bude předán natrénovaný uložený model neuronové sítě, který agent načte a bude dle něj řídit.

Metoda sensors

Metoda slouží k definici senzorů, které zkoumají okolí na autonomním automobilu. Možnosti senzorů se liší dle módu agenta. Metoda musí vracet list slovníků, kdy každý slovník popi-

suje jeden použitý senzor. Ve slovníku se vždy uvádí typ senzoru, jeho identifikátor, který jednoznačně senzor popisuje, později se díky němu bude přistupovat k získaným datům.

Krom těchto vlastností se využívají i jiné, typicky se jedná o umístění senzoru na automobilu, natočení a další. Specifické vlastnosti jsou uvedeny níže u každého senzoru. Od každého čidla je možné použit pouze určitý omezený počet, zároveň senzory nesmí být dále než tři metry od rodičovského objektu.

Výstupy všech senzorů jsou předávány metodě *run_step* jako slovník, kdy klíčem jsou identifikátory senzorů. Každý senzor vrací n-tici hodnot. První hodnota vždy obsahuje číslo obrazu (frame), kdy byl záznam pořízen, druhá hodnota je specifická pro každý senzor a je popsána níže.

RGB kamera

Senzor získává vizuální obraz ze svého okolí. Simulátor obsahuje několik typů kamer, žebříček využívá pouze RGB kamer, tedy těch, které zaznamenávají okolí ve třech barevných kanálech. Čtvrtým kanálem je průhlednost, hodnoty v tomto kanále jsou typicky na maximální hodnotě.

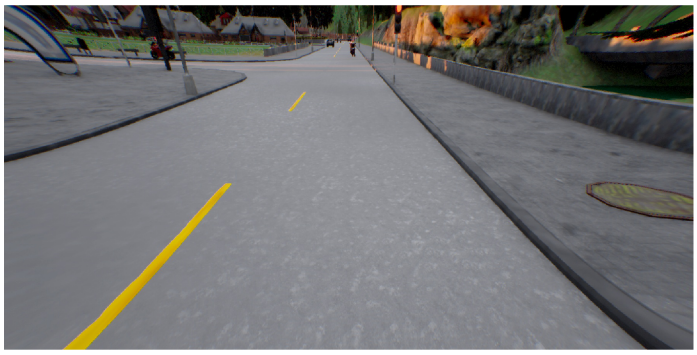
Typ: sensor.camera.rgb
Počet použití: 0-4

Při nastavování kamer v metodě se nastavují následující základní hodnoty.

Tabulka 3.4: Základní vlastnosti kamery.

Vlastnost	Popis
x,y,z	Pozice kamery vzhledem k rodičovskému objektu (automobilu).
roll, pitch, yaw	Natočení kamery vzhledem k automobilu.
width	Šířka obrazu v pixelech zachytávaného kamerou.
height	Výška obrazu v pixelech zachytávaného kamerou.
fov	Horizontální zorné pole ve stupních.

Výstupem je RGBA trojrozměrné pole datového typu *uint8*, hodnoty v poli jsou v rozmezí 0-255. Trojrozměrné pole má velikost (výška, šířka, 4). To znamená že hodnoty jednotlivých pixelů jsou ukládány po řádcích a následně po sloupcích. Pixel obsahuje 4 hodnoty vyjadřující velikost barevných složek červené, zelené a modré barvy, a navíc jednu složku průhlednosti. Příklad výstupu je možné pozorovat na obrázku 3.4.



Obrázek 3.4: Výstup z přední kamery CARLA simulátoru.

Pokud by byl obrázek vypsán jako textový obsah hodnot byl by získán následující výstup.

```
{'FrontCamera': (64915,
    array([
        [ [140, 136, 135, 255], ..., [ 73, 141, 147, 255] ],
        ...,
        [ [141, 135, 136, 255], ..., [106, 102, 106, 255] ]
    ], dtype=uint8))}
```

GNSS

Senzor získávající polohu v GPS souřadnicích. Určuje tedy světovou šířku a délku a nadmořskou výšku.

Typ: sensor.other.gnss
Počet použití: 0-1

Zásadním nastavením GPS senzoru je především jeho pozice v automobilu.

Tabulka 3.5: Základní vlastnosti GNSS senzoru.

Vlastnost	Popis
x,y,z	Pozice kamery vzhledem k rodičovskému objektu (automobilu).

Výstupem je opět dvojice, kdy první číslo udává jako obvykle číslo obrazu pořízení dat. Druhý parametr je pole tří reálných čísel, které udávají v následujícím pořadí zeměpisnou délku, zeměpisnou šířku a nadmořskou výšku.

```
{'GPS': (64915,
    array([-0.0020268 , 0.0030414 , 1.62601002])
)}
```

IMU

Jedná se o kombinaci více čidel. Senzor zprostředkovává údaje akcelerometru. Ten získává zrychlení rodičovského objektu. Součástí je také kompas pro určení natočení ke světovým stranám a gyroskop, který měří natočení podél všech tří os automobilu.

Typ: sensor.other.imu
Počet použití: 0-1

Pro přidání IMU senzoru je nutné definovat jeho umístění a natočení vůči automobilu.

Tabulka 3.6: Základní vlastnosti IMU senzoru.

Vlastnost	Popis
x,y,z	Pozice kamery senzoru k rodičovskému objektu (automobilu).
roll, pitch, yaw	Natočení senzoru vzhledem k automobilu.

Na výstupu je pole sedmi reálných čísel. První tři čísla udávají zrychlení z akcelerometru v ose x, y a z. Tato čísla jsou v m/s^2. Následující tři čísla je výstup gyroskopu opět ve stejných osách v rad/s. Posledním číslem je natočení kompasu v radiánech.

```
{'IMU': (386870,
    array([
        2.30051941e-04, 5.34433988e-04, 9.79699707e+00,
        -8.46036361e-04, 4.51491476e-04, 2.57582380e-03,
        5.49839115e+00
    ])
)}
```

LiDAR

Jedná se o otočný LiDAR, které má stejné vlastnosti jako čidlo společnosti Velodyne. Tento senzor provede v jednom kroku 10 otočení kolem osy, má 64 paprsků a maximální dosah 85 metrů.

Typ: sensor.lidar.ray_cast
Počet použití: 0-1

U tohoto typu senzoru dává smysl nastavení jeho pozice a natočení na automobilu.

Tabulka 3.7: Základní vlastnosti LiDARu.

Vlastnost	Popis
x,y,z	Pozice LiDARu vzhledem k rodičovskému objektu (automobilu).
roll, pitch, yaw	Natočení LiDARu vzhledem k automobilu.

Na výstupu senzoru je pole bodů, kam dopadly paprsky LiDARu. Jelikož je číslo otáčející se, je získán popis celého okolí. Každý zaznamenaný bod má čtyři hodnoty. První tři jsou místo popsané pomocí kartézských souřadnic. Čtvrtou hodnotou je intenzita odraženého paprsku.

```
{'LIDAR': (2146,
    array([
        [ 2.9960039e+01, -3.9062499e-05, 5.2827597e+00, 8.8542378e-01],
        [ 2.9818827e+01, 3.9945310e-01, 5.2583313e+00, 8.8592213e-01],
        ...,
        [-4.2718750e-01, 1.4296874e-02, -2.4679549e-01, 9.9802768e-01],
        [-4.2667967e-01, 1.1406249e-02, -2.4642013e-01, 9.9803048e-01]
        ], dtype=float32)
)}
```

RADAR

Jedná se o RADAR s dosahem až do sta metrů. Identifikuje objekty před senzorem. Získává popis bodů, které detekoval vzhledem k umístění čidla.

Typ: sensor.other.radar
Počet použití: 0-2

Jeho vlastnosti jsou podobné LiDARu, tedy umístění a natočení na automobilu. RADAR ovšem funguje pouze v jednom směru, je nutné nastavit horizontální a vertikální rozsah ve stupních. To znamená, v jakém rozsahu před automobilem budou směrovány rádiové vlny.

Tabulka 3.8: Základní vlastnosti RADARu.

Vlastnost	Popis
x,y,z	Pozice RADARu vzhledem k rodičovskému objektu (automobilu).
roll, pitch, yaw	Natočení RADARu vzhledem k automobilu.
fov	Vertikální a horizontální zorné pole ve stupních.

I zde je výstupem pole detekovaných bodů, přičemž každý bod je popsán čtyřmi reálnými čísly. Tato čísla vyjadřují něco jiného než v případě LiDARu. Místo kartézských souřadnic je využíván polární systém. První tři čísla vyjadřují výšku, azimut (obojí v radiánech) a vzdálenost (metr) detekovaného bodu od senzoru. Čtvrtá hodnota je rychlost pohybujícího se bodu vzhledem k pozorovateli.

```
{'RADAR': (2146,
    array([
        [ 1.52045403e+01, -1.00323528e-01, 1.24539040e-01, 7.67560791e+03],
        ...,
        [ 3.85733528e+01, -2.58420361e-03, -4.08685133e-02, 8.03053857e+03]
        ], dtype=float32)
)}
```

Rychloměr

Jedná se o odvozený senzor, není standardním senzorem CARLA simulátoru. Zjišťuje rychlost rodičovského objektu v m/s.

Typ: sensor.speedometer
Počet použití: 0-1

Tento senzor nemá žádné dodatečné vlastnosti, které jsou potřeba při definici.

Na výstupu je možné najít slovník, který obsahuje jeden klíč s hodnotou *speed*, pod tímto klíčem je k nalezení odhad rychlosti.

```
{'Speed': (386870, {'speed': 9.340688958277544e-06} )}
```

OpenDrive mapa

Další z odvozených senzorů. Jeho využití je možné pouze v případě, že agent má nastaven mód mapy. Umožňuje přístup k mapě okolí.

Typ: sensor.opendrive_map
Počet použití: 0-1

Jedinou vlastností, která je nutná definovat je frekvence čtení.

Tabulka 3.9: Základní vlastnosti OpenDrive mapy.

Vlastnost	Popis
reading_frequency	Frekvence rychlosti obnovy mapy, hodnota 1 označuje čtení v každém okamžiku provolání metody.

Výstupem je XML soubor, v něm je dle standardu popsána mapa okolí. Každá část cesty má vlastní identifikátor. Stejně tak jsou rozlišeny identifikátorem i jízdní pruhy, kladnými čísly pruhy jízdy a zápornými pruhy v protisměru. Celkový popis formátu je možné najít ve standardu [2].

Metoda run_step

Tato metoda je provolávána periodicky. Označuje jeden krok agenta. Vstupem metody jsou dva parametry. Prvním z nich je časové razítko. To označuje čas, kdy se provolal aktuální krok. Druhým parametrem je výstup ze všech senzorů. Jedná se o slovník, jehož klíče jsou identifikátory a hodnoty jsou data získaná z CARLA simulátoru. Vzhled dat byl popsán v předchozí sekci 3.3.

Tým zde implementuje vlastní řízení autonomního agenta. K tomu může kromě výstupů ze senzorů použit i plán cesty, který pro něj žebříček připravil. Ty byly popsány výše 3.3. Jsou dostupné v proměnné *self._global_plan* a *self._global_plan_world_coord* vzhledem k módu, ve kterém agent pracuje.

Ve funkci bude pravděpodobně zapojena neuronová síť, která bude brát tyto vstupy a rozhodovat o výstupech.

Výstupem metody je samotné řízení vozidla. To je popsáno třídou *VehicleControl*, kterou funkce vrací jako výsledek. Třída obsahuje čtyři parametry, které se musí správně nastavit. Prvním parametrem je *VehicleControl.hand_brake*, ten očekává hodnoty *True*, nebo *False*. Jinými slovy ano či ne, dle toho, zda je zabrzděna ruční brzda. Další v pořadí je natočení kol dostupné v parametru *VehicleControl.steer*. To očekává reálné číslo v rozmezí -1 až 1. Hodnota -1 udává, že jsou kola naplno otočená doleva, naopak 1 znamená plné zatočení doprava. Poslední dvojicí jsou parametry *VehicleControl.throttle* a *VehicleControl.brake* pro sešlápnutí plynu a brzdy. Oba parametry očekávají reálné číslo od 0 do 1, přičemž 0 znamená, že pedál nebyl sešlápnut a 1, že byl sešlápnut na plno.

Metoda destroy

Metoda sloužící na úklid a uložení všech rozdělaných věcí, otevřených souborů nebo síťových spojení. Je volána před zánikem funkce agenta.

3.4 Další schopnosti CARLA simulátoru

CARLA simulátor není jediným programem vytvořeným za účelem tvorby autonomních automobilů. Rozhodně se ale jedná o nejkomplexnější nástroj, který je díky otevřenému kódu dostupný pro všechny vývojářské týmy.

V kapitole bylo vysvětleno, jak vytvořit autonomního agenta ze třídy *AutonomouAgent*. Tuto třídu je schopen spustit jak Leaderboard, tak Scenario Runner. Příklad takového spuštění je možné vidět v příloze B.2.

To je ale jen jedna z možností, jak vytvořit program pro autonomní řízení. Samotný simulátor obsahuje programy, které obsahují základní ukázky agentů, které jsou schopny řídit automobil. Ty vůbec nemusí a ani nevyužívají neuronových sítí. Místo toho pracují se znalostí informací ze simulátoru. Jsou schopny zjistit pozici a stav ostatních automobilů, chodců, semaforů a dopravních značek. Souhrnně se těmto informacím říká herci. Agenti jsou schopni z těchto informací zjistit, zda jim nehrozí srážka nebo jestli mohou projet světelnou křižovatkou.

Kromě toho jsou k nalezení i různé ukázky programů, které se připojí na simulátor a vypisují informace na standardní výstup nebo programy které umožní ovládat automobil přes klávesnici. Všechny tyto součásti jsou k nalezení ve složce *PythonAPI* simulátoru. Plno příkladů, jak takové programy napsat je možné dohledat v dokumentaci [8].

Využití tohoto způsobu psaní programů není cílem této práce. I přes to byl autonomní agent simulátoru, který je řízen za pomoci informací simulátoru v práci využit. V třídě *AutonomousAgent* jde inicializovat takového agenta a delegovat řízení na něj. Tento způsob je tak vhodný například ke sběru datové sady.

Kapitola 4

Využití neuronových sítí pro implementaci CARLA leaderboard autonomního agenta

Celková implementace autonomního agenta je rozdělena na několik rozdílných problémů. Prvním je nalezení, nebo vytvoření vhodné datové sady, na které se bude provádět učení a validace modelů. Další problém k vyřešení jsou skripty a programy, které obsahují definici modelů neuronových sítí a nástroje pro trénování. Ty jsou psané pomocí knihovny PyTorch, která umožňuje snadný návrh a učení sítí. Jakmile jsou všechny tyto úkony hotovy a modely sítí jsou natrénované, byl z nich vytvořen autonomní agent, který řídí vozidlo pouze za pomoci senzorů.

Projekt je rozdělen do desíti částí, přičemž každá část je uložena ve vlastní složce.

Tabulka 4.1: Popis součástí projektu.

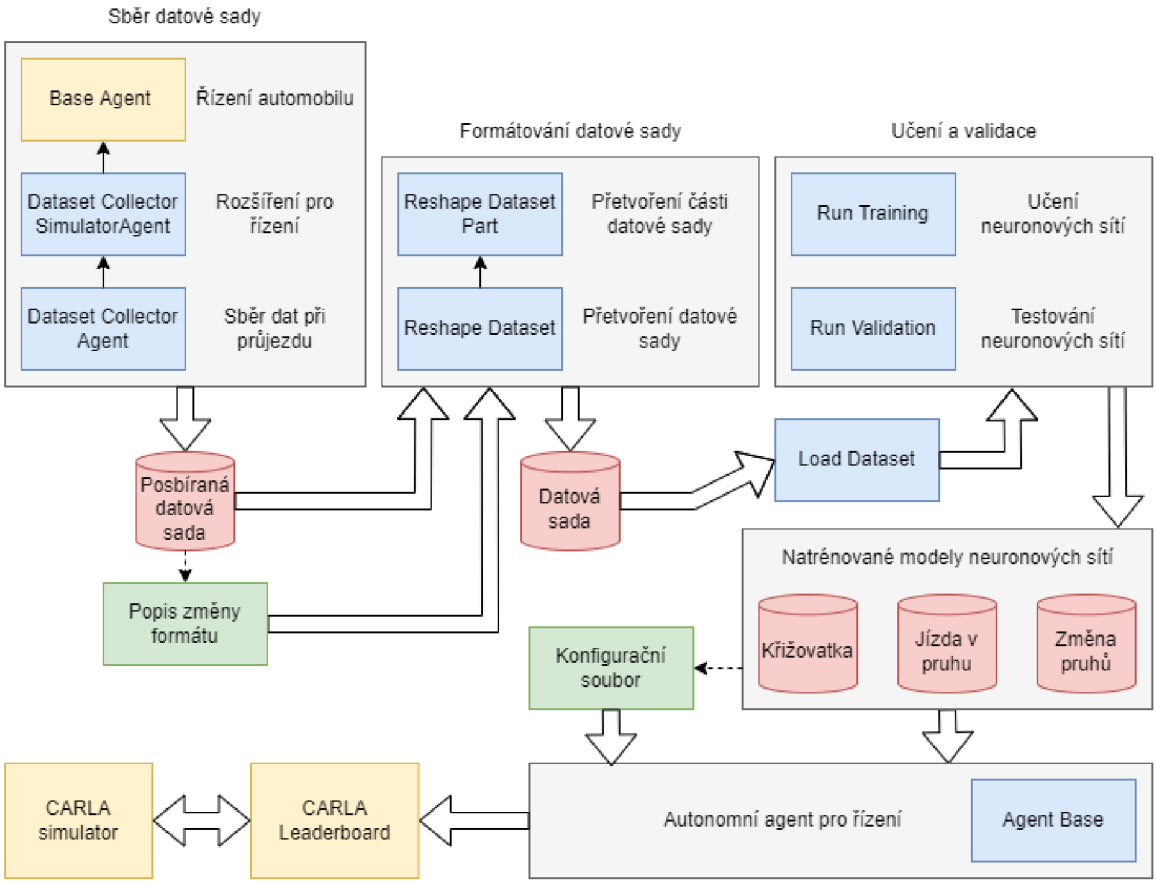
Složka	Popis
autonomous_agents	Obsahuje výsledného agenta pro CARLA žebříček. Také obsahuje konfigurační soubor, který agent bude číst.
datasets	V této složce jsou agenti starající se o pořízení datových sad. Programy pro třídění datové sady a nástroje pro přístup k jednotlivým položkám sad.
drive_results	Do této složky jsou exportovány výsledky vyhodnocování CARLA žebříčku.
drive_routes	Složka obsahuje soubory s popisem tras pro automobily.
drive_scenarios	Soubory s definicí testovaných scénářů a jejich poloha na mapě.
neural_network	Programy se třídami pro tvorbu neuronové sítě pomocí knihovny PyTorch.
scripts	Obsahuje Bash skripty, které spouští veškeré možné části projektu.
trained_models	Ve složce jsou soubory s natrénovanými modely neuronových sítí.
training	Programy pro trénování a validaci modelů sítí.
utilities	Funkce a třídy, které se využívají napříč projektem.

Návrh implementace popisuje obrázek 4.1. Lze zde pozorovat celkovou logiku návrhu a komunikace mezi jednotlivými částmi. Barvami jsou odlišeny části projektu, které mají určité souvislosti.

Žlutá barva označuje programy, které jsou součástí CARLA simulátoru a žebříčku. Ty byly pro tento projekt převzaty a využívány. Modrou barvu mají moduly, které byly vytvořeny.

Jsou rozděleny do několika kategorií. Některé sbírají datovou sadu, další provádí operace nad touto sadou a přetváří ji k použití. K tomu využívají popis, který říká, jakým způsobem má být datová sada přetvořena. Použití datové sady je dosaženo pomocí programu, který načte data z disku. Takto načtená data jsou využity v programech, které učí a validují tři typy neuronových sítí. Sítě jsou za pomoci konfiguračního souboru načteny do autonomního agenta, který se bude starat o samostatné řízení automobilu. Agenta zaobalí programy žebříčku, které ho použijí k simulaci. Během simulace bude CARLA žebříček komunikovat se simulátorem.

Zelenou barvou jsou na obrázku vyznačeny soubory, které byly ručně vytvořeny a obsahují konfigurační popisy. Naopak červenou barvou jsou soubory, nebo více souborů, které byly automaticky vytvořeny a byly permanentně uloženy na disku. Jde o posbíranou, nebo přeformátovanou datovou sadu, a také uložené soubory s hodnotami naučených vah a posunů neuronových sítí.



Obrázek 4.1: Schéma návrhu implementace projektu.

Veškeré části jsou v jednom repositáři. Cesta k tomuto repositáři je exportována do proměnné prostředí *TEAM_CODE_ROOT*, více v příloze instalace B.1.

Existují další definované proměnné prostředí. Všechny jsou cesty k částem projektu, které mohou být umístěné na jiných místech. Je to *DATASETS_ROOT*, v níž jsou vloženy veškeré datové sady, se kterými se pracuje. Další je *LEADERBOARD_ROOT*. Ta obsahuje cestu k repositáři se skripty CARLA žebříčku. *SCENARIO_RUNNER_ROOT* je adresář se skripty pro spouštění a testování scénářů. Ty nejsou v projektu použity, protože se vše ovládá přes žebříček. Poslední proměnnou je *CARLA_ROOT*. V ní je uložen CARLA simulátor. Veškeré proměnné jsou využívány v programech a cesty jsou přidány do *PYTHONPATH*, což umožní viditelnost balíčků odkudkoli.

Před popisem implementace hlavních částí projektu je provedeno seznámení s obecnými nástroji. Jsou připraveny v části projektu *utilities* a využívají se napříč celým projektem. Jedná se o pomocné třídy a funkce. Funkce jsou k nalezení v:

`$TEAM_CODE_ROOT/utilities/main.py`

Obsahuje pět funkcí.

```
def try_parse_int(string):
    ...
def alphanumeric_key(string):
    ...
def print_error(message):
    ...
def log(message):
    ...
def progres_print(count, total, bar_len):
    ...
```

Pomocná funkce *try_parse_int()* přijímá na svém vstupu textový řetězec. Tento text se pokouší přetypovat na celé číslo. Pokud se to daří, vrací číslo (textový řetězec je číselná hodnota). Pokud ne, vrací původní řetězec. Funkce je využívaná pouze další funkcí v pořadí.

Ta se jmenuje *alphanumeric_key()* a také přebírá text. Text, který dostane rozdělí na sekvence. Jedná se o sekvence číselných a ostatních znaků. Části jsou odeslány k pokusu o přetypování. Z číselných textových sekvencí vzniknou čísla, ostatní nebudou změny. Funkce tyto části vrací jako rozdělený list. Příklad může vypadat třeba takto.

```
# vstup
'/path/to/dataset/episode_00000'
# vystup
['path/to/dataset/episode_', 0]
```

Využití funkce spočívá při řazení souborů nebo složek, které jsou vyčteny z adresářů datové sady. Tímto je zaručeno, že složka *episode_00010* bude v pořadí až za složkou *episode_00002*. Řazení číselné části je provedeno numericky, nikoli abecedně.

Funkce *print_error()* pouze vypíše předaný chybový text na standardní chybový výstup programu. Navíc přidává popis, díky kterému se uživatel dozví, kde zjistit více informací. Je volána při tréninku nebo validaci v případě chyby.

Obdobná funkce je i *log()*. Ta je volána také při tréninku nebo validaci kdykoli je potřeba cokoli vypsat na standardní výstup. Obecně vypisuje výsledky tréninku. Přidává ke zprávě časové razítko.

Aby byl uživatel informován o postupu tréninku, existuje funkce *progress_print()*. Vypisuje na standardní výstup aktuální pokrok v provádění validace a tréninku. Uživatel vidí splněná procenta. S každým krokem je provolána a upravena na novou hodnotu.

Kromě funkcí jsou připraveny také třídy dědící od hlavní třídy výjimek. Představují speciální výjimky, které mohou nastat v rámci běhu agenta nebo učení neuronových sítí. Všechny jsou definovány v:

`$TEAM_CODE_ROOT/utilities/exceptions.py`

Soubor obsahuje třídu výjimek *AgentException* vytvořenou pro autonomního agenta. Při inicializaci mu je zadán typ chyby, který nastal během řízení. Na základě typu vypisuje různé chybové hlášky. Typ chyby může být chybný plán cesty, který byl vytvořen žebříčkem a obsahuje body, na kterých agent mění styl řízení (například vjezd a výjezd z křižovatky). To nastane v případě, že plán obsahuje méně než dva body (musí existovat minimálně počáteční a koncový bod). Další chybou je pokus agenta přistoupit k bodu cesty, který neexistuje (přístup mimo meze listu). Nastavována je i výjimka v případě, že agent nerozpozná aktuální stav řízení, který byl vyčten z plánu cesty. Jelikož agent čte několik souborů potřebných při konfiguraci, může vyhodit výjimku, pokud některý z nich neexistuje.

```
class AgentException(Exception):
    def __init__(self, type_name):
        ...
    def __str__(self):
        ...
```

Existuje speciální výjimka *ArgumentParsingException*, která je vyhozena při tréninku a validaci. Vyskytuje se v případě, že byl zadán nepovolená kombinace vstupních parametrů programů. Při inicializaci je nastaven popis, ke kterému porušení argumentů došlo.

```
class ArgumentParsingException(Exception):
    def __init__(self, description):
        ...
    def __str__(self):
        ...
```

V celém programu se přistupuje ke všem typům existujících modelů sítí, typů datových sad a typů chybových funkcí přes připravené funkce, které fungují jako rozcestníky. Je jim vložen typ, který je požadován a oni vrátí jeho inicializovaný objekt. To přispívá jednoduší rozšiřitelnosti a univerzálnosti. Pokud jakákoli funkce ovšem dostane nedefinovaný identifikátor vyhodí výjimku *UndefinedException* s textovým popisem, že nezná požadovaný typ.

```
class UndefinedException(Exception):
    ...
```

Modely neuronových sítí používané v projektu jsou implementovány z několika částí. Například konvoluční sítě mají konvoluční a plně propojenou část. Části jsou předpřipraveny a u výsledných modelů dochází pouze k jejich inicializaci a propojení do jednoho celku. U částí je možné nastavit počet vrstev, neuronů a tak dále. Existují kritéria, která musí být dodržena (například minimální počet vrstev). Pokud nejsou je vyhozena výjimka *NetException*.

```
class NetException(Exception):
    ...
```

Využití představených výjimek a funkcí bude lépe vidět v následujících sekcích, kde budou zasazeny do kontextu s celkovou funkcí programu.

4.1 Pořízení a příprava datové sady

K dispozici je více datových sad pořízených pomocí CARLA simulátoru. Jedním z takových je například datová sada All-In-One Drive [35]. Její nevýhodou je, že neobsahuje informace, které v daný okamžik ovládaly vozidlo. Stejný problém obsahuje i datová sada KITTI-CARLA [13], která je tvořena spíše pro sémantickou segmentaci.

Lepším řešením je posbírat si vlastní datovou sadu pomocí existujícího data kolektoru vytvořeného přímo od firmy CARLA [6]. Ten již poskytuje informace o vozidle. Veškerá data jsou popsána pomocí formátu JSON. I zde je ovšem nevýhoda. Kolektor je vytvořen pro speciální verzi simulátoru. Ten je ve verzi 0.8.4. To vede na další problémy. Simulátor neobsahuje všechna potřebná města a v této verzi je možné nastavit pouze kamery a LiDAR.

Z těchto důvodů byl jako součást projektu vytvořen způsob sbírání dat na potřebné verzi 0.9.10.1. K tomu byl jako základ využit agent pojmenovaný *BasicAgent*, který je součástí programů přiložených k simulátoru. Pokud je cesta k simulátoru exportovaná do proměnné *CARLA_ROOT*, je cesta k agentovi:

`$CARLA_ROOT/PythonAPI/carla/agents/navigation/basic_agent.py`

Rozšíření pro agenta simulátoru

Obyčejný agent ovládá vozidlo za pomocí znalostí okolního světa. Přistupuje do simulátoru a zjišťuje polohu vozidel a ostatních herců. Rozpozná, kde jsou dopravní značky, jestli je na semaforu červená a další. I přesto neobsahuje všechny potřebné schopnosti, které jsou k vytvoření datové sady potřeba. Z toho důvodu je nutné jeho kód mírně upravit. To je uděláno vytvořením třídy *DatasetCollectorSimulatorAgent*, která dědí od *BasicAgent*. Implementace je v souboru:

`$TEAM_CODE_ROOT/datasets/dataset_collector_simulator_agent.py`

```
class DatasetCollectorSimulatorAgent(BasicAgent):
    def run_step(self, debug=False):
        ...
    def _is_vehicle_hazard_pedestrians(self, vehicle_list, \
        ego_vehicle_location, ego_vehicle_waypoint, proximity_th = 10, \
        up_angle_th = 20, low_angle_th = 0):
        ...
    def _is_vehicle_hazard_obstacle(self, vehicle_list, \
        ego_vehicle_location, ego_vehicle_waypoint):
        ...
    def get_incoming_waypoint_and_direction(self, steps=3):
        ...
```

Oproti původnímu agentovi je u nového zcela předefinována metoda *run_step()*. Tato metoda se stará o vyhodnocení jednoho kroku simulace a řídí veškeré řízení automobilu. Nejdříve získává seznamy herců v okolí (semafory, vozidla, chodci a značky stop). Pokud je v dosahu značka stop, zastaví na dalších sto kroků na místě a následně pokračuje dále. Poté provádí kontrolu chodců před vozidlem, pak kontrolu vozidel před řízeným automobilem a na závěr zkontroluje, zdali je v dosahu semafor a svítí zelená barva. Oproti původní metodě upravuje chování rozlišování automobilů, které by mohly být překážkou. Přidává zastavování před chodci a značkami stop. Vrací hodnoty *VehicleControl* pro řízení vozidla, které závisí na nalezení překážky před vozidlem.

Druhá metoda je *_is_vehicle_hazard_pedestrians()*. Ta je využívána pro kontrolu, zda jsou před vozidlem chodci. Ignoruje chodce, které se pohybují na chodníku vedle auta, jelikož ti nejsou podstatní. Ve výchozím nastavení kontroluje pouze rozsah dvaceti stupňů před automobilem. Ten je dostatečný pro zjištění chodce v pruhu, do kterého by mohl automobil narazit.

Nová je i metoda *_is_vehicle_hazard_obstacle()*. Upravuje zjišťování překážejících automobilů. Vyhledávání automobilů, které by mohly překážet nedělá pouze z aktuální pozice, ale porovnává i pozice cesty, kde se řízený automobil bude nacházet za deset kroků simulátoru. Díky tomu vymazává chybu, které se základní agent dopouštěl na křižovatkách. Ten ignoroval automobily na hranicích, kde se měnil identifikátor cest. V těchto místech se auta tvářila, že jsou na jiné cestě, než řízený automobil.

Metoda *get_incoming_waypoint_and_direction()* vrací pozici, kde se bude automobil nacházet za určený počet kroků simulátoru.

Sběr dat pomocí leaderboard agenta

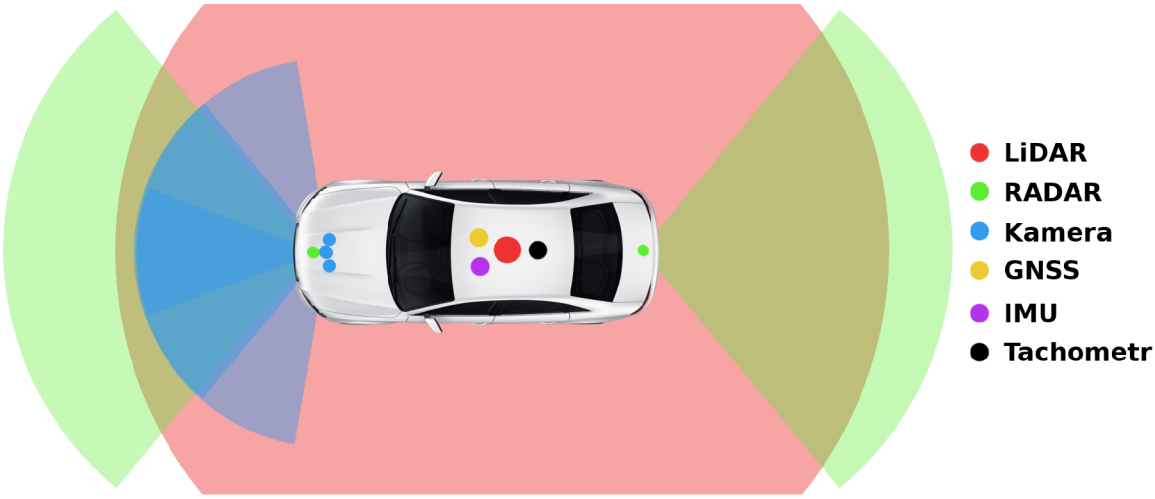
Pro sběr dat existuje třída *DatasetCollectorAgent*. Ta automobil sama neřídí. Pouze inicializuje předchozího agenta a přenechává řízení jemu. Využívá možnosti žebříčku k pořizování dat, definuje senzory, které ukládá a využívá plánu cesty, který žebříček připraví. Je implementovaná v souboru:

`$TEAM_CODE_ROOT/datasets/dataset_collector_leaderboard_agent.py`

```
class DatasetCollectorAgent(AutonomousAgent):
    def setup(self, path_to_conf_file):
        ...
    def sensors(self):
        ...
    def run_step(self, input_data, timestamp):
        ...
    def _collect_data(self, input_data, timestamp, control):
        ...
    def _init_data_collection(self):
        ...
    def _init_agent(self):
        ...
    def _assign_route(self):
        ...
```

Vychází ze třídy *AutonomousAgent* žebříčku, která již byla popsána výše v 3.3. V metodě *setup()* provádí nastavení. Zapamatovává si cestu ke konfiguračnímu souboru, která představuje cestu, kam se bude ukládat datová sada. Nastavuje výchozí hodnoty pro číslo epizody a záznamu.

Senzory definované pro sběr dat jsou k nalezení v *sensors()*. Definice obsahuje tři RGB kamery umístěné v přední části automobilu. První z nich je umístěna na středu, ostatní dvě jsou každá pootočeny o třicet stupňů na každou stranu. Všechny mají šířku 800 a výšku 600 pixelů. Zorné pole činí sto stupňů. Dále je mezi senzory otočný LiDAR na střeše automobilu, jednotka IMU získávající orientaci, GNNS pro zjišťování GPS polohy a tachometr. Krom toho jsou součástí i dva RADARy. Jeden umístěný opět ve předu automobilu a druhý vzadu. Oba jsou ve středové ose a mají zorné pole sto stupňů. Obrázek 4.2 ukazuje jejich umístění.



Obrázek 4.2: Senzory na vozidle.

Funkce *run_step()* je volaná s každým krokem simulace. Při prvním provolání zavolá metody, které inicializují samořídícího agenta, připraví sběr dat a nastaví agentovi cestu, kterou bude následovat. V dalších krocích nechává řízení na definovaném agentovi. Zároveň po každém vyhodnocení kroku volá metodu starající se o ukládání dat získaných ze senzorů.

V *_init_agent()* je nastaven agent, který bude zprostředkovávat řízení. Ten nebude nikdo jiný, než *DatasetCollectorSimulatorAgent* již představený v předchozí sekci 4.1. Celé řízení provádí vzhledem ke své znalosti okolního světa (čte umístění objektů ze simulátoru). Také dodržuje definovanou cestu, která je mu nastavena metodou *_assign_route()* z globálního plánu, který již CARLA žebříček připravil.

Pro sběr dat musí nejprve dojít k přípravě složky s datovou sadou, která se provádí v *_init_data_collection()*. Z definovaného umístění se nejprve vyčtou veškeré epizody, které jsou již vytvořeny. Epizoda je jedna celá trasa vozidla. Obsahuje veškeré záznamy ze senzorů. Pokud složka již obsahuje epizody, vytvoří novou složku pojmenovanou *episode_xxxxx*, přičemž „xxxxx“ udávají její identifikátor. Pokud neexistuje doposud žádná epizoda, vytváří novou s identifikátorem „00000“.

Data jsou v každém kroku sbírána metodou *_collect_data()*. Záznamy jsou ukládány do složky s aktuální epizodou. Jeden krok je zaznamenám pomocí čtyř souborů. Jsou to tři obrázky pojmenované *CameraCentral_yyyyy.png*, *CameraLeft_yyyyy.png* a *CameraRight_yyyyy.png*. Posledním souborem je pole Python knihovny NumPy [16] uložené jako soubor *OtherSensors_yyyyy.npy*. Více o formátu uložení je v následující podkapitole 4.1.

Písmena „yyyyy“ označují identifikátor jednoho kroku sběru dat. Záznamy jsou číslovány od hodnoty „00000“.

Spouštění sběru dat

Pro spuštění sběru dat je připraven bashový skript. Ten očekává již běžící CARLA simulátor. Skript na svém vstupu dostane zadanou složku, která musí být připravena v umístění datové sady, které je definované proměnnou *$DATASETS_ROOT*.

Po kontrole existence umístění se provede spuštění vyhodnocovacího skriptu žebříčku. Tomu je předán agent *DatasetCollectorAgent*, který jak již bylo vysvětleno posbírá záznamy na požadované místo.

Pro sběr datové sady jsou použity veškeré dostupné scénáře definované v souboru:

```
$LEADERBOARD_ROOT/data/all_towns_traffic_scenarios_public.json
```

Scénáře se objevují na 113 kilometrech rozprostřených do 50 tras ve všech možných městech. Tyto trasy jsou definované v žebříčku v souboru:

```
$LEADERBOARD_ROOT/data/routes_training.xml
```

Konkrétní popis, jak spustit sběr datové sady je popsán v příloze B.2.

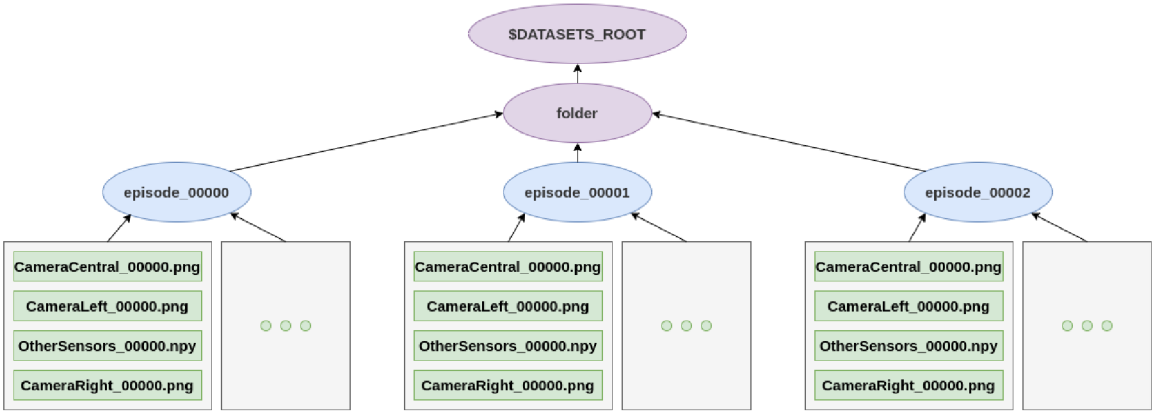
Popis formátu posbíraných dat

Jakmile jsou data posbírány jsou dostupné na na cestě:

```
$DATASETS_ROOT/folder
```

Přičemž složka *folder* byla zadána jako parametr skriptu, který sbíral data. Složka bude obsahovat padesát epizod za předpokladu, že sběr nebyl předčasně ukončen. Každá z epizod bude obsahovat rozdílný počet záznamů v závislosti na délce trasy a situacích, které se v ní vyskytnou. Záznamy jsou vždy čtveřice. Tři ze souborů jsou obrázky. Čtvrtý je uložené asociativní NumPy pole, které má pod klíči hodnoty zbylých senzorů.

Obrázky jsou vždy pojmenované *CameraCentral_xxxxx.png*, *CameraLeft_xxxxx.png* a *CameraRight_xxxxx.png*. Soubor s ostatními senzory je *OtherSensors_xxxxx.npy*, jeho obsahem jsou výstupy senzorů. Posbíraný formát datové sady je naznačen na obrázku 4.3.



Obrázek 4.3: Formát posbírané datové sady.

Třídění dat

Pro použití takto posbírané datové sady se provede její úprava. Roztříděním na tři základní části je docíleno možnosti trénovat různé modely pro různé jízdní situace. První z nich jsou situace, kdy automobil jede v jízdním pruhu, druhou, kdy vozidlo mění jízdní pruhy a poslední z nich jsou situace, kdy je vozidlo v křižovatce. K tomuto účelu byl opět vytvořen bashový skript. Jeho použití je popsané v příloze B.2.

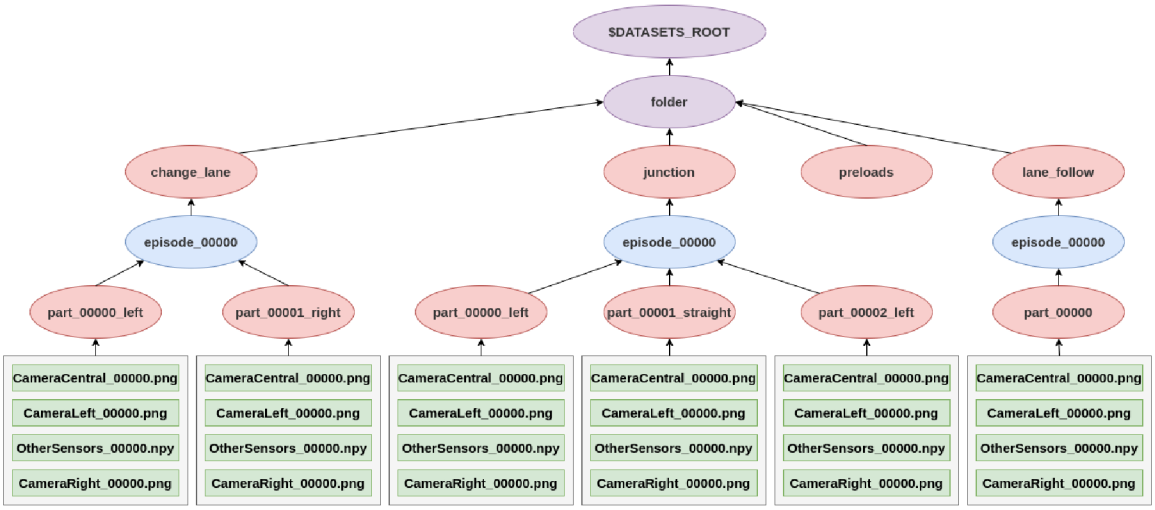
Záznamy se musí ručně projít a zároveň tvořit soubor, který popisuje třídění. V tomto kroku je možné vynechat části datové sady, u kterých došlo k chybě. Například situace, kdy vozidlo narazilo do překážky nebo projelo semaforem na červenou. Skript přečte takto vytvořený soubor na standardním vstupu. Dle obsahu souboru provede skript roztřídění. Na prvním řádku souboru je cesta ke složce s posbíranou datovou sadou. Druhý řádek je cesta k nové složce, kam bude sada roztříděna.

Do nové složky jsou vytvořeny další podsložky. Jsou to *change_lane*, v ní budou uloženy záznamy, kdy automobil přejížděl mezi pruhy. Složka *junction* slouží k uložení dat, kdy automobil projížděl křižovatkou. Jízdy v jízdním pruhu jsou vloženy do složky *lane_follow*. Adresář *preloads* bude sloužit k ukládání přednačtených informací o datové sadě.

Třetí a následující řádky souboru popisují rozdělení datové sady. Na každém řádku je pět parametrů. První popisuje složku s epizodou, druhý jízdní situaci neboli jednu ze tří výše popsaných složek. Třetí argument popisuje identifikátor části. Ten se skládá z části *part_* a pěti číslic pro rozlišení. U změny pruhu se přidává přípona *_left* pro změnu pruhu vlevo a *_right* pro změnu vpravo. Křižovatka má navíc ještě situaci s příponou *_straight*, která popisuje průjezd křižovatkou rovno. Čtvrtý a pátý argument je číslo prvního a posledního záznamu situace. Řádek může vypadat například takto.

```
1 episode_00000 lane_follow part_00000 00000 00007
```

Skript přečte celý řádek a jeho obsah pošle jako argumenty dalšímu skriptu. Ten vytvoří ve všech složkách se situacemi další podsložku pro danou epizodu a do ní složku, která bude mít stejný název jako třetí argument. Bude popisovat část jedné cesty. Do této složky přesune veškeré záznamy, které mají identifikátor mezi čtvrtým a pátým argumentem. Výsledek transformace popisuje obrázek 4.4.



Obrázek 4.4: Formát výsledné datové sady.

Datová sada bude rozdělena na jízdní situace. Každá situace obsahuje veškeré epizody původní datové sady. Původní epizoda je rozdělena na části dle situací. Část je souvislá posloupnost záznamů situace (například průjezd křižovatkou). Každá část je zařazena do složky situace a epizody.

Čtení datové sady

Načítání datových sad je tvořeno tak, aby bylo snadno rozšiřitelné. Je definována funkce *Dataset()* v souboru:

`$TEAM_CODE_ROOT/datasets/main.py`

```
def Dataset(directory, dataset_type_name):
    ...
```

V této funkci je připraven rozcestník pro třídy, které načítají datové sady. Výběr probíhá pomocí jména typu. Jsou definovány tři jména pro načítání datové sady.

Tabulka 4.2: Popis typů tříd pro načítání datové sady.

Jméno typu	Popis
xdopit03_d_cl	Třída, která načítá výše popsaný formát datové sady se situací změny jízdního pruhu.
xdopit03_d_ju	Třída, která načítá výše popsaný formát datové sady se situací průjezdu křižovatkou.
xdopit03_d_lf	Třída, která načítá výše popsaný formát datové sady se situací jízdy v pruhu.

Všechny tři situace vedou na jednu třídu *Xdopit03DatasetLoader* popsanou v souboru:

`$TEAM_CODE_ROOT/datasets/xdopit03_dataset_loader.py`

```
class Xdopit03DatasetLoader:
    def __init__(self, dataset_directory, subtype):
        ...
    def __len__(self):
        ...
    def __getitem__(self, index):
        ...
    def _load_data(self):
        ...
    def _get_start_end_positions(self, part_path, first_id, last_id):
        ...
```

Při inicializaci třídy se předávají dva argumenty. Prvním z nich je umístění datové sady. Druhý je typ neboli situace (*change_lane*, *junction*, *lane_follow*). Součástí inicializace je vyhledávání cest ke všem souborům sady. Každý ze čtyř typů souborů je uložený ve vlastním seznamu. Navíc existuje speciální seznam, který je využit u křižovatek a změny jízdního pruhu. V něm jsou uloženy pozice místa, kde manévr začal a na kterém místě skončil, ty jsou vždy stejné pro celou jednu posloupnost (část).

Seznamy mohou být načteny ze speciálního souboru označeného jako „preload“. Každá ze situací má vlastní přednačtený soubor. Jsou pojmenovány jako *change_lane_preload.npy*, *junction_preload.npy* a *follow_lane_preload.npy*. Jestliže je cesta k datové sadě:

```
$DATASETS_ROOT/dataset/
```

Potom všechny tyto soubory jsou uloženy v:

```
$DATASETS_ROOT/dataset/preloads
```

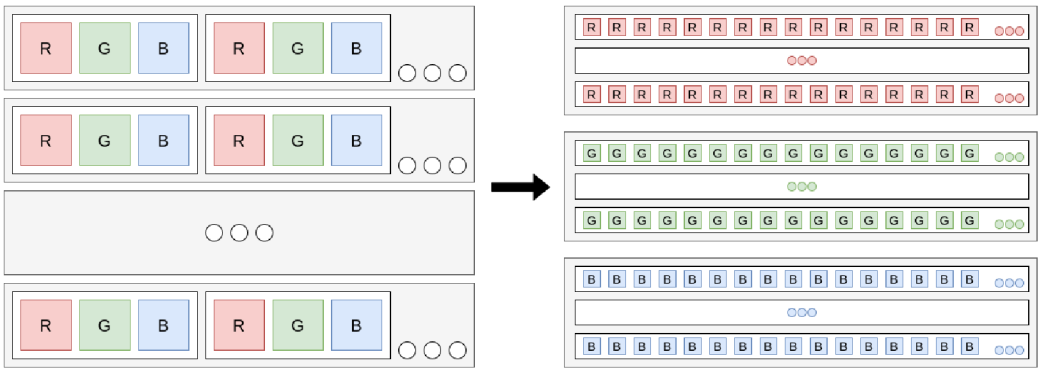
Soubory jsou uložené NumPy pole, které stačí konvertovat na seznamy. Pokud soubor neexistuje musí se seznamy načíst z adresáře. K tomu slouží metoda *_load_data()*. Ta se podívá do adresáře datové sady a složky jízdní situace. Zjistí veškeré epizody, části každé epizody a záznamy každé části. Ověří, že veškeré soubory existují a jejich cesty vloží do seznamů.

V situaci průjezdu křižovatky, nebo změnu pruhu volá *_get_start_end_positions()*. Ta nalezne první a poslední záznam každé části a ze souboru ostatních senzorů vyčte GPS pozici automobilu. Pozice jsou navráceny jako slovník, který je přidán do seznamu. Díky tomu u každého záznamu datové sady bude možné určit na jakém místě automobil začal, kde se nachází a kde má skončit.

Zároveň je vytvořen přednačtený soubor všech takto získaných cest. Vzhledem k velikosti datové sady by neustálé čekání na vyhledání cest trvalo příliš dlouho. Datová sada se v tomto okamžiku již nemění, proto je vhodné takového souboru využít.

Poslední dvě metody musí obsahovat každá třída zpracovávající data. Pomocí nich se přistupuje k záznamům. Funkce *___len___()* vrací velikost datové sady. Ta je stejná jako velikost některého ze seznamů cest k souborům. Ty jsou vždy stejné velké, protože vždy jsou nalezeny čtveřice záznamů a na stejných indexech seznamů jsou uloženy záznamy se stejným identifikátorem. Přístup k jednomu záznamu zprostředkovává metoda *___getitem___()*.

Ta připravuje záznam na vrácení. Na svém vstupu dostane index. Tímto indexem získá cestu k souboru ostatních senzorů a ten načte. Dostane NumPy pole hodnot, které dále rozšiřuje. V případě křižovatky a změny pruhu přidává hodnotu na kterou stranu automobil zatočil a také přidá počáteční a koncovou pozici. Následně jsou data konvertována na tenzor. Jakmile je všechno hotové, jsou načteny všechny obrázky. Je jím upravena velikost na 400x300 pixelů, ta je dostatečná. Hodnoty obrázků jsou transponovány. Původně je uložen jako pole pixelů. Každý pixel se skládá z RGB hodnot. Po transformaci je obrázek definován jako seznam tří seznamů. Kdy každý seznam obsahuje hodnoty pouze R, G, nebo B hodnot.



Obrázek 4.5: Transformace obrázku.

Transformaci popisuje obrázek 4.5. Obsah jsou celočíselné hodnoty v rozsahu 0-255. Konvertují se na reálná čísla a jsou normována podělením do rozsahu 0,0-1,0. To je uděláno z důvodu lepšího zpracování neuronovou sítí. Upravené obrázky jsou přidány do slovníku, který bude funkcí vrácen.

Pomoci těchto dvou metod je docíleno toho, že k záznamům třídy může být přistupováno přes index a je možné i využití příkazu *for* pro průchod celou sadou. Příklad výstupu jednoho záznamu vypadá takto.

```
{'lidar': tensor([
        #[ x, y, z, intenzita],
        [12.260293 , -6.629348 , 2.4576163 , 0.94496125],
        ...,
        [ 4.391162 , -0.05883325, -2.5354607 , 0.9799206 ]
    ]),
'radar_front': tensor([
        #[vyska, azimut, vzdalenost, rychlost pohybu],
        [ 3.13064170e+00, -4.75582600e-01, -6.06540859e-01, -3.7743654e+00],
        ...
        [ 1.21970234e+01, 4.21325527e-02, -5.76828897e-01, -4.34180784e+00]
    ]),
'radar_back': tensor([ ... ]),
'gps': tensor([
        -2.31712787e-05, # zemepisna delka
        1.30543267e-03, # zemepisna sirka
        2.53370946e+00 # nadmorska vyska
    ]),
'imu': tensor([
        2.41409588, -0.81123012, 9.96432877, # zrychleni v~ose x, y,
z~-0.01185672, 0.01109121, 0.01193542, # naklon gyroskopu x, y,
z~0.79959053 # natoceni kompasu
    ]),
'speed': tensor([5.7039]), # rychlost m/s
'control': tensor([ 0.3498 ,0. , -0.02084 ]), # plyn ,brzda, natoceni kol
'direction': tensor([0.]), # 0 = rovne, -1 = vlevo, 1 = vpravo
'start_position': tensor([-4.8627e-04, 1.5162e-03, 2.5336e+00]), # x, y,
z~'end_position': tensor([-4.9599e-04, 1.3042e-03, 2.5338e+00]), # x, y,
z~'c_image': tensor([
        [ # cervena
            #[sloupec 0, sloupec 1 ..., sloupec N]
            [0.0000, 0.0000, 0.0000, ..., 0.5451, 0.5451, 0.5451], # radek 0
            ...,
            [0.6902, 0.6902, 0.6902, ..., 0.8157, 0.8157, 0.8118] # radek M
        ],[ # zelena
            [0.0000, 0.0000, 0.0000, ..., 0.5216, 0.5176, 0.5176],
            ...,
            [0.7333, 0.7373, 0.7333, ..., 0.8471, 0.8471, 0.8431]
        ],[ # modra
            [0.0000, 0.0000, 0.0000, ..., 0.5098, 0.5059, 0.5059],
```

```
            ...,
            [0.7765, 0.7765, 0.7765, ..., 0.8745, 0.8745, 0.8745]
        ]]),
'l_image': tensor([ ... ]),
'r_image': tensor([ ... ])}
```

4.2 Návrh neuronové sítě a její učení

Neuronové sítě jsou navrženy jako tři rozdílné modely. Každý model je trénován na jinou jízdní situaci (křižovatky, změna pruhu, jízda v pruhu). Každý z modelů má rozličnou strukturu a využívá jiné vstupní informace. Jejich implementace je popsána později. Pro jednoduší práci s více modely jsou vytvořeny pomocné funkce v souboru:

`$TEAM_CODE_ROOT/neural_network/main.py`

```
def Model(model_type_name):
    ...
def Loss(loss_function_type_name):
    ...
```

Funkce fungují jako rozcestníky pro výběr, který model sítě nebo která chybová funkce má být použita. V rozcestníku modelů neuronových sítí jsou již zmíněné tři hodnoty.

Tabulka 4.3: Popis typů tříd modelů neuronových sítí.

Jméno typu	Popis
model_xdopit03_cl	Model neuronové sítě pro změnu pruhu.
model_xdopit03_ju	Model neuronové sítě pro křižovatky.
model_xdopit03_lf	Model neuronové sítě pro jízdu v pruhu.

Chybová funkce je možné vybrat pouze ze dvou typů.

Tabulka 4.4: Popis typů chybových funkcí.

Jméno typu	Popis
mean-absolute-error	Chybová funkce MAE popsána v 2.4
mean-squared-error	Chybová funkce MSE popsána výše v 2.4.

Trénování modelů neuronových sítí

I přesto, že se pracuje se třemi rozdílnými modely, programy pro trénink jsou napsány univerzálně pro všechny. Ke spuštění tréninku byl vytvořen skript, jehož spouštění je popsáno v příloze B.2.

Skript ve smyčce dle zvoleného počtu epoch spouští Python program, který provede učení. Epocha znamená jeden průchod tréninku nad celou datovou sadou. Na začátku každé epochy je vypsáno její číslo a čas jejího počátku. Spouštěný program je:

`$TEAM_CODE_ROOT/training/run_training.py`

Veškeré jeho vstupní parametry jsou nastaveny v závislosti na prvním argumentu skriptu, který označuje, jaká neuronová síť bude učena. Nastavují se parametry jako označení modelu k učení. Jména byla popsána v tabulce 4.3. Také jméno pro soubor, ve kterém bude uložen natrénovaný model. Označení datové sady popsané v 4.2. Potřebná je i cesta k datové sadě. Pokud má program pouze pokračovat v učení, může mu být zadán parametr, který popisuje, který soubor má být načtený před začátkem učení. Příklad spuštění jedné epochy učení:

```
$./run_training.py \
    --model=model_xdopit03_ju \
    --save_model=xdopit03_trained_model_ju \
    --dataset=xdopit03_d_ju \
    --dataset_path=$DATASETS_ROOT/dataset \
    --load_model=xdopit03_trained_model_ju.pt
```

Uložené soubory s modelem sítě obsahují hodnoty parametrů (vah a biasů) v sítí. Jsou přímo závislé na jejím návrhu. Tyto soubory jsou vždy uloženy ve složce:

```
$TEAM_CODE_ROOT/trained_models
```

Python program využívá k trénování knihovnu PyTorch [26]. Celé trénování obstarává funkce *execute_training()*.

První je inicializováno načítání datové sady, které bylo popsáno výše za pomoci funkce *Dataset()* [4.1]. Takto připravený objekt je předán PyTorch třídě *DataLoader()*, která ho zaštítí. Přidává k načítání další možnosti. Pomocí něj se nastaví velikost dávky, po které se provede úprava parametrů. V tomto případě je dávka o velikosti jednoho záznamu (výchozí hodnota). Je zapnuta možnost, která způsobí, že při procházení datovou sadou budou záznamy vytahovány v náhodném pořadí.

```
dataset = Dataset(path, dataset)
data_loader = torch.utils.data.DataLoader(dataset, shuffle=True)
```

Jakmile je takhle připravena datová sada dochází k inicializaci jednoho ze tří modelů za pomoci funkce *Model()*, která byla opět popsána dříve v 4.2. Pokud program bude navazovat na trénování, načítá soubor s uloženým postupem a nahrává z něj uložené hodnoty vah a posunů do modelu sítě.

Na řadu přichází definice optimalizačního algoritmu. Zde je zvolen pokročilý algoritmus Adam, který staví na gradientním sestupu [2.4]. Jsou mu předány parametry modelu. Jejich znalost je klíčová. Pomocí ní bude vědět, které parametry budou optimalizovány.

Poslední inicializací před učením je zvolení chybové funkce. Jelikož jde o regresní problém, byla zde zvolena funkce MSE [2.4]. Ta je obecně brána jako výchozí funkce pro regresi. Je opět volena za pomocí funkce *Loss*, která byla popsána na začátku kapitoly 4.2.

```
model = Model(model)
optimizer = optim.Adam(model.parameters())
loss_function = Loss('mean-squared-error')
```

Po všech nastaveních se přistupuje k učení. To probíhá iterací přes objekt, který zastřešuje datovou sadu. Na začátku každé iterace musí dojít k vynulování předchozích vypočítaných gradientů. V opačném případě by došlo k jejich sečtení a negativně by ovlivňovali učení.

```
for data in data_loader:
    model.zero_grad()
```

Nyní je čas provést dopředný krok neuronové sítě. Vstupem kroku jsou veškerá data jednoho záznamu. Každý model pracuje s mírně odlišnými daty na svém vstupu. Aby byl algoritmus pro všechny stejný. Model dostane všechna data a sám si vybere jen ta, která jsou pro něj relativní. Po průchodu je získán výstup sítě. Bude to tenzor tří reálných čísel vyjadřující velikost plynu, brzdění a natočení kol. Z dat záznamu je vyčten stejný referenční tenzor. Obě hodnoty jsou odeslány do chybové funkce k vypočtení odchylky.

```
output = model(data)
target = torch.squeeze(data['control'])
loss = loss_function(output, target)
```

Po vyhodnocení chyby jsou vypočítány nové gradienty pomocí zpětné propagace [2.4]. Z gradientů je optimalizátor schopný sestupem provést úpravu parametrů neuronové sítě. Jakmile jsou parametry upraveny pokračuje se další várkou záznamů.

```
loss.backward()
optimizer.step()
```

Jakmile dojde k ukončení tréninku, pouze se uloží nový stav modelu sítě (hodnoty parametrů) a stav optimalizátoru do permanentního souboru. Ve výsledku se bude jednat o tři soubory, které jsou:

```
$TEAM_CODE_ROOT/trained_models/xdopit03_trained_model_cl.sh
$TEAM_CODE_ROOT/trained_models/xdopit03_trained_model_ju.sh
$TEAM_CODE_ROOT/trained_models/xdopit03_trained_model_lf.sh
```

Každý z nich byl natrénován na jinou z jízdních situací v desítkách či stovkách epoch. Po dokončení tréninku je dobré zkontrolovat schopnosti sítě. K tomuto účelu jsou opět připraveny programy pro validaci.

Validace modelů neuronových sítí

Stejně jako trénink i u validace nezáleží na vybraném modelu sítě. Nástroje jsou univerzální. Dokonce se dá říct, že tyto programy jsou velmi podobné těm, které provádí trénink. I zde byl vytvořen spouštěcí skript. Jeho využití je popsáno v příloze B.2.

Skript zde spouští dle typu modelu, který je zadán prvním argumentem skriptu, Python program:

```
$TEAM_CODE_ROOT/training/run_validation.py
```

Parametry pro spuštění jsou podobné jako v případě tréninku. Jediným rozdílem je chybějící parametr pro uložení souboru s modelem. Validační program pouze čte již natrénovaný soubor, ten musí být v parametrech vždy uveden. Dalšími parametry jsou jméno modelu, typ datové sady a cesta k ní. Příklad spuštění:

```
$./run_validation.py \
    --model=model_xdopit03_ju \
    --dataset=xdopit03_d_ju \
    --dataset_path=$DATASETS_ROOT/dataset \
    --load_model=xdopit03_trained_model_ju.pt
```

Pro validaci je v programu implementována funkce *execute_validation()*.

Jako v případě tréninku je inicializovaní datová sada, třída načítání dat a model sítě, do kterého se načtou uložené váhy.

```
dataset = Dataset(path, dataset)
data_loader = torch.utils.data.DataLoader(dataset, shuffle=False)
model = Model(model)
```

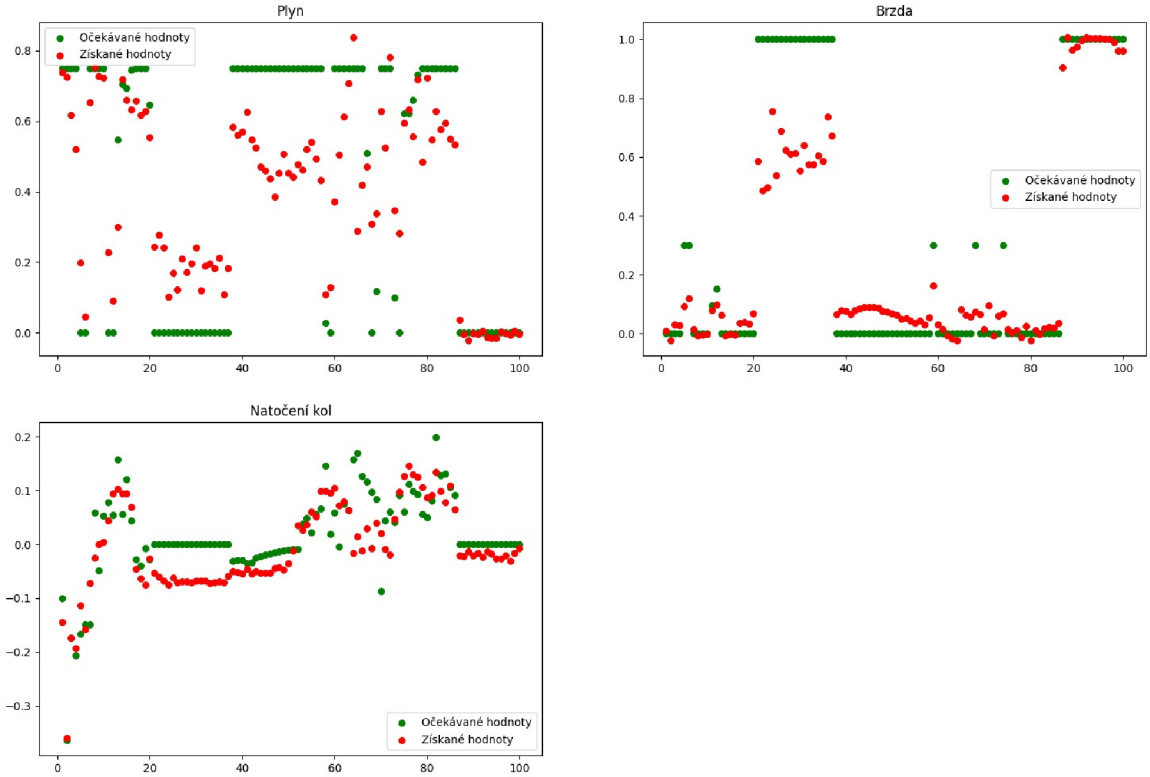
Optimalizátor v případě validace není potřeba. Neuronová síť nebude nikdy upravovat svoje hodnoty. Místo toho se musí modelu říct, že bude provádět vyhodnocování. Díky tomu model automaticky vypne některé vrstvy jako je například „dropout“. Jsou to vrstvy, které mají funkci pouze při tréninku a při vyhodnocování naopak výsledky zhoršují. Chybová funkce je nastavena jako MSE [2.4].

```
model.eval()
loss_function = Loss('mean-squared-error')
```

Algoritmus validace prochází datovou sadu, provádí dopředné kroky a vypočítává chybu. Chyby jednotlivých záznamů přičítá k celkové chybě.

```
for data in data_loader:
    output = model(data)
    target = torch.squeeze(data['control'])
    loss_sum += loss_function(output, target).item()
```

Výstupem validace je součet všech chyb a vypočítaná průměrná chyba na jeden záznam vypsaná na standardní výstup. Kromě toho jsou do listů u každého záznamu posbírány referenční hodnoty datové sady a hodnoty získané z průchodů neuronovou sítí. Na konci validace jsou vykresleny posbíraná čísla do tří grafů za pomoci knihovny Matplotlib [17].



Obrázek 4.6: Výstup validace modelu změny pruhu.

Každý graf popisuje jednu informaci, která řídí automobil. Obrázek 4.6 s grafy je uložen na konci validace do složky:

`$TEAM_CODE_ROOT/trained_models/`

Soubor je pojmenovaný z části jako *validation_result-*. Druhá část názvu je časové razítko, kdy byl soubor vytvořen s koncovkou *.jpg*. Každý graf má vykreslené zelené a červené body. Zelené popisují cílené hodnoty, ke kterým by se měla neuronová síť učit, jsou to data uložená v datové sadě. Ty červené ukazují čísla, která neuronová síť vypočítala. Z grafu jde dobře sledovat, jak moc se výstupy sítě blíží očekávaným hodnotám. Jelikož má validační program vypnutou možnost míchání záznamů v datech, záznamy jsou vždy vyhodnocovány jako sekvence části jízdy. Stejným způsobem bude neuronová síť vyhodnocovat data ze simulátoru v implementaci autonomního agenta.

Nyní je jasné, jak jsou implementované programy pro trénink a validaci. Zbývá vysvětlit návrh sítí, které se používají.

Návrh modelů neuronových sítí

Jak již bylo vysvětleno, jsou implementovány tři rozdílné modely neuronových sítí. Každý s jiným účelem řešení jízdních situací automobilu. Také byla vysvětlena funkce, která za pomoci jména typu modelu vrací jeho inicializovaný objekt [4.2]. Je už i jasné, jak budou sítě trénovány a validovány. Nyní je čas si popsat jejich vnitřní návrh.

I když jsou všechny tři modely odlišné, obsahují některé společné části. Ty byly implementovány zvlášť jako samostatné třídy. Vytvoření cílového modelu spočívá v inicializaci a spojování několika částí do jednoho celku. K tvorbě tříd neuronových sítí je využita knihovna PyTorch. V ní existuje třída *nn.Module*. Od té budou dědit všechny ostatní třídy. Pro správnou funkci musí být minimálně popsána metoda *forward()*. Do ní se zapisuje, jak má vypadat dopředný průchod neuronovou sítí.

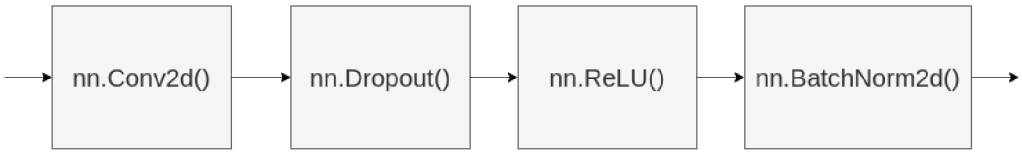
Obecný návrh podčástí modelů

Předpřipravené společné části jsou dvě. První z nich se zabývá konvolucemi ve tří dimenzionálním prostoru na vstupních datech. Její název je *NNConvolutinalPart* a je vytvořena v souboru:

`$TEAM_CODE_ROOT/neural_network/nn_convolutional_part.py`

```
class NNConvolutinalPart(nn.Module):
    def __init__(self, channels, kernel, stride, dropout_probability):
        ...
    def forward(self, data):
        ...
    def _number_of_pixels(self, data):
        ...
    def simulate_output_size(self, size):
        ...
```

V metodě *__init__()* při inicializaci je provedeno sestavení sítě. Umožňuje vytvořit libovolný počet vrstev. Každá vrstva je složena ze sekvence 2D konvoluční vrstvy [2.5] (výška x šířka obrázku), dropout vrstvy [2.3], aktivační funkce ReLU [2.3] a vrstvy normalizační [2.3]. Složení popisuje obrázek 4.7.



Obrázek 4.7: Složení jedné vrstvy konvoluce.

Pro každou takhle postavenou vrstvu musí existovat hodnoty pro její nastavení. Ty jsou předané v parametrech jako čtyři listy kanálů, velikostí jader, posunů jader a pravděpodobností výpadku neuronu vrstvy. Co přesně tyto hodnoty znamenají bylo vysvětleno v sekci 2.5 vysvětlující konvoluční sítě.

```
kanaly = [3, 32, 64, 32, 16],
jadro = [5, 3, 3, 3],
posun = [2, 1, 2, 1],
pravdepodobnost_vypadku = [0.0, 0.2, 0.2, 0.0]
```

Každá hodnota popisuje nastavení právě tolikáté vrstvy, na jakém místě se vyskytuje v listu. Hlavním listem jsou kanály, ten je o jednu hodnotu delší než zbylé tři. Udává počet vstupních kanálů, se kterými pracuje konvoluční vrstva. Pokud je n hodnota počtu vstupních kanálů, potom je n+1 počet výstupních kanálů vrstvy n a zároveň počet výstupních kanálů vrstvy n+1. Číslo navíc v listu udává výstup poslední vrstvy, která není napojena na žádný další vstup. Z příkladu výše to znamená, že existují čtyři vrstvy s takhle definovanými parametry.

```
kan_vstup = 3 , kan_vystup = 32, jadro = 5, posun = 2, prav_vypadku = 0.0
kan_vstup = 32, kan_vystup = 64, jadro = 3, posun = 1, prav_vypadku = 0.2
kan_vstup = 64, kan_vystup = 32, jadro = 3, posun = 2, prav_vypadku = 0.2
kan_vstup = 32, kan_vystup = 16, jadro = 3, posun = 1, prav_vypadku = 0.0
```

To znamená, že list kanálů musí obsahovat minimálně dvě hodnoty (vstup a výstup jedné vrstvy). Ostatní listy musí být přesně o jednu hodnotu kratší než list kanálů. Pokud tomu tak není je vyhozena chybová výjimka sítě.

Je připraven list, do kterého budou ukládány vytvořené vrstvy. Ty jsou tvořeny iterováním v rozsahu zadaného počtu vrstev. Ten je získán jako počet kanálů mínus jedna.

Získá se objekt pro konvoluční vrstvu. Té jsou zadány počty vstupních a výstupních kanálů, velikost jádra a posun. Zároveň je vytvořena dropout vrstva s požadovanou pravděpodobností. Inicializována aktivační funkce ReLU a připravena normalizační vrstva pro počet výstupních kanálů.

Vrstvy jsou propojeny jako sekvenční průchod a přidány do listu vrstev. Jakmile jsou vytvořeny všechny vrstvy, je tento list přetransformován opět jako sekvenční průchod. Díky tomu je vytvořena požadovaná neuronová síť zabývající se konvolucí obrázků.

```
conv = nn.Conv2d(channels[i], channels[i + 1], kernel[i], stride[i])
dropout = nn.Dropout(p = dropout_probability[i])
relu = nn.ReLU(inplace = True)
batch_norm = nn.BatchNorm2d(num_features = channels[i + 1])
self._layers.append(nn.Sequential(*[conv, dropout, relu, batch_norm]))
```

V metodě *forward()* jsou předložena vstupní data sítě. Díky jejímu vytvoření v inicializační části, zde stačí do ní data pouze vložit a získat výstup. Po získání výstupu jsou data zploštěna do jedno dimenzionálního vektoru. Ty jsou výstupem konvoluční části sítě.

```
output = self._layers(data)
flattened_output = output.view(-1, self._number_of_pixels(output))
```

Ke zploštění je potřeba znát celkový počet všech pixelů ve všech kanálech výstupu. K jeho spočítání je připravena metoda *_number_of_pixels()*. Ta je možná využít v případě, že byl již pořízen výstup sítě.

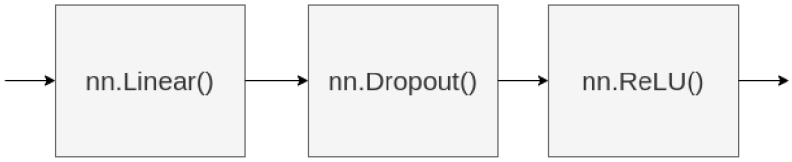
To někdy není možné. V době inicializace se bude chtít zjistit, jak velký je výstupní vektor ještě před tím, než mu budou poslána data. To je z toho důvodu, že za konvoluční částí se nachází část lineární a je třeba jí nastavit velikost vstupu. Kvůli tomu byla vytvořena metoda *simulate_output_size()*. Té stačí vědět velikost vstupního obrázku konvoluční části. Vytvoří si tenzor nul a prožene ho sítí. Na konci spočítá počet výstupních pixelů, který vrací.

Druhou připravenou částí jsou plně propojené lineární vrstvy ve třídě *NNFullyConnPart*. Je k nalezení v souboru:

`$TEAM_CODE_ROOT/neural_network/nn_fully_conn_part.py`

```
class NNFullyConnPart(nn.Module):
    def __init__(self, features, dropout_probability, last_part = False):
        ...
    def forward(self, data):
        ...
```

Zde bylo postupováno obdobným způsobem jako v případě konvoluční části. Místo kanálů se zde mluví o vlastnostech. V podstatě se jedná o počty vstupů vrstvy (neurony) a počet výstupů (neurony další vrstvy). Ty jsou definovány v listu stejným způsobem jako byly kanály. List obsahuje o jednu hodnotu více oproti druhému listu. Tím je list pravděpodobností zahození neuronu. Více listů hodnot není potřeba. Jedna vrstva je složena pouze z lineární vrstvy, dropout vrstvy a aktivační funkce ReLU. Složení je znázorněno na obrázku 4.8.



Obrázek 4.8: Složení jedné vrstvy plně propojené části.

Stejně jako předtím jsou nejprve zkontrolovány vstupní hodnoty. List vlastností musí mít minimálně 2 hodnoty (vstupní a výstupní). Zároveň list pravděpodobností musí být o jednu hodnotu kratší. Pokud tomu tak není je vyhozena výjimka.

Tentokrát je iterováno přes počet velikostí vlastností mínus jedna, což je velikost výstupu poslední vrstvy. V jedné iteraci jsou inicializovány všechny již popsané vrstvy.

```
lin = nn.Linear(features[i], features[i+1])
dropout = nn.Dropout(p = dropout_probability[i])
relu = nn.ReLU(inplace = True)
```

Po inicializaci jsou vloženy do sekvence za sebe. Mohou nastat dva případy. Aktuálně zpracovávána vrstva není poslední. V tom případě je do sekvence přidána i aktivační funkce. Pokud je poslední, výstup je bez aktivace. Poslední vrstvou je myšleno nikoli poslední vrstva

pouze v této části, ale ve všech částech. Jinými slovy aktivační funkce se nepřidá v případě generování poslední vrstvy poslední části celkového modelu, který bude popsán níže. Informace, zda se jedná o poslední část je předána v parametrech.

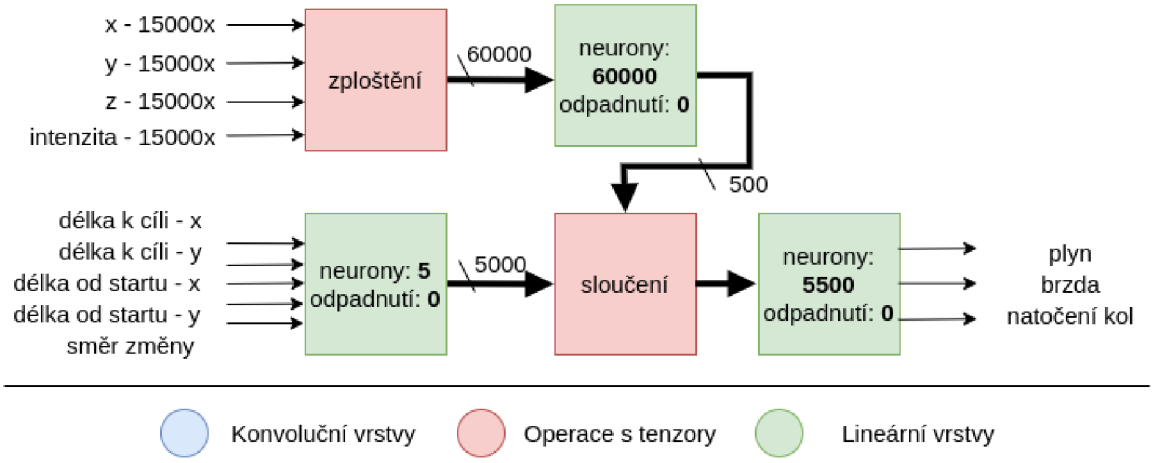
```
if i == len(features) - 2 and last_part:
    layer = nn.Sequential(*[lin, dropout])
else:
    layer = nn.Sequential(*[lin, dropout, relu])
```

Takto vygenerované vrstvy jsou vloženy do listu a po vytvoření všech je list přetvořen jako sekvenční průchod mezi nimi.

Jedinou další metodou je opět *forward()*. Zde je situace velmi lehká. Vstupní data jsou prostě jen vložena na model vytvořený v inicializaci. Výstup je vrácen bez úprav.

Návrh modelu sítě pro změnu pruhu

Pro vyhodnocování změny pruhu je složen model neuronové sítě ze tří lineárních plně propojených vrstev. Na svém vstupu dostává zpracovaná data ze senzorů, nebo z uložených dat datové sady. Ze senzorů si vybere potřebné věci. Pro tento model je získán záznam z LiDARU a záznamy o pozici automobilu. V tomto modelu byl upřednostněn LiDAR, jelikož přední kamery nedokáží rozpoznat automobily, které by mohly být v cílovém pruhu. Neuronová síť je vytvořena v inicializační metodě.



Obrázek 4.9: Návrh neuronové sítě pro změnu pruhu.

Je složena ze tří rozdílných prvků. Její skladbu lze pozorovat na obrázku 4.9. Jeden slouží ke zpracování záznamů z LiDARu. Jeho výstupy jsou ořezány, nebo rozšířeny na 15000 hodnot od každé vlastnosti. Ty jsou zploštěny do jednoho tenzoru a odeslány do první lineární vrstvy o 60000 neuronech. Výstupem je 500 hodnot.

Druhá část se zabývá zpracováním polohy automobilu. Na svém vstupu dostane pětici hodnot. Ty budou popsány níže. Tato část má pouze jednu vrstvu, na jejím výstupu bude k nalezení dalších 5000 hodnot.

Výstupy obou části jsou složeny do jednoho tenzoru o velikosti 5500 různých hodnot. Takto sestavený tenzor je odeslán do poslední části, která se zabývá složením. Také obsahuje jedinou vrstvu. Na výstupu jsou výsledná tři reálná čísla. Tato čísla vyjadřují ovládání automobilu.

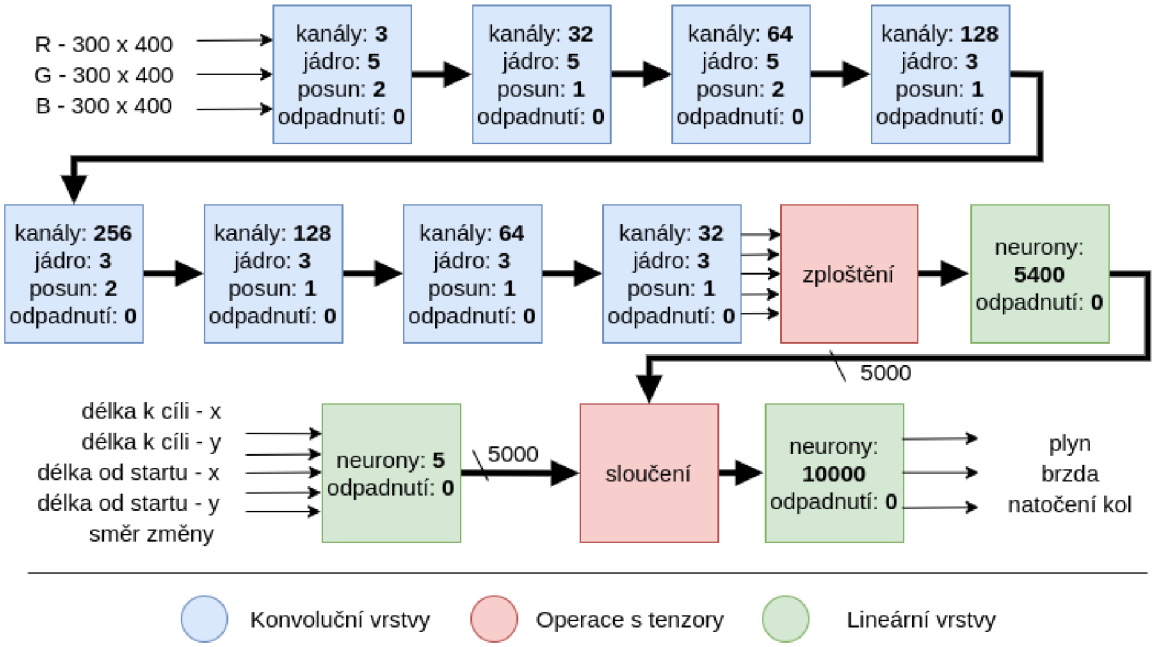
Průchod sítí je popsán v metodě *forward()*. Na začátku si vyčte z dat hodnoty LiDARu a pozici automobilu. Pozice automobilu se skládá z pěti čísel, které jsou sestaveny do tenzoru na začátku funkce. K tomu je využita aktuální pozice automobilu a pozice, kde začala a skončila změna pruhu.

Od cílové pozice je odečtena hodnota aktuální pozice, a naopak od aktuální pozice je odečtena hodnota startovací pozice. Díky tomu jsou získány vzdálenosti k cíli a od startu. Vzdálenost je vyjádřena ve třech číslech. Jsou to souřadnice x, y, z. Pro automobil je výška bezpředmětná, a tak jsou do vstupních dat přidány pouze hodnoty x a y. Jelikož jsou tyto rozdíly minimální, jsou data pro lepší rozlišení vynásobeny 10 000. Pátým a posledním číslem, které je do dat přidáno, je informace o změně směru.

Data LiDARu jsou normovány na 15000 vstupů. Pokud senzor pořídil více hodnot, jsou poslední hodnoty zahozeny, naopak pokud mu chybí, jsou první hodnoty odeslány do sítě dvakrát. Vstupy jsou odeslány do svých sítí. Jejich výstup je propojen do jednoho tenzoru, který je protažen poslední sítí. Metoda vrací výstup této poslední části.

Návrh modelu sítě pro křižovatky

Model pro křižovatku využívá stejně jako ten předchozí informace o poloze automobilu. LiDAR ovšem nahrazuje vstupem středové přední kamery. Tato kamera má dostatečné zorné pole, aby zachytila všechny komplikace, které by uprostřed křižovatky mohly nastat.



Obrázek 4.10: Návrh neuronové sítě pro jízdu křižovatkou.

Model je vytvořen v inicializační metodě a popsán na obrázku 4.10. Obsahuje tři části. První je konvoluční pro zpracování výstupu z kamery. Druhá plně propojená pro zpracování polohy. Jejich výstupy jsou propojeny a odeslány do poslední lineární vrstvy.

V konvoluční části se zpracovávají fotografie v osmi konvolučních vrstvách. Počet kanálů postupně stoupá k hodnotě 256 a poté zpět klesá dolů. První tři vrstvy mají velikost jádra pět na pět pixelů. Jsou díky tomu nalezeny významnější souvislosti v pixelech obrázku.

Zbytek velikostí jader jsou nastaveny na tři. Velikosti výpadků jsou nulové. Posuvy se v různých vrstvách mění. Mají buďto hodnotu jedna, nebo dva.

Po zploštění je vrácen tenzor o velikosti 5400. Dle toho jsou nastaveny vstupní neurony lineární plně propojené části umístěné bezprostředně za tou konvoluční. Části zabývající se zpracováním polohy a následujícím propojením polohy s kamerou jsou naprosto stejné jako v předchozím modelu.

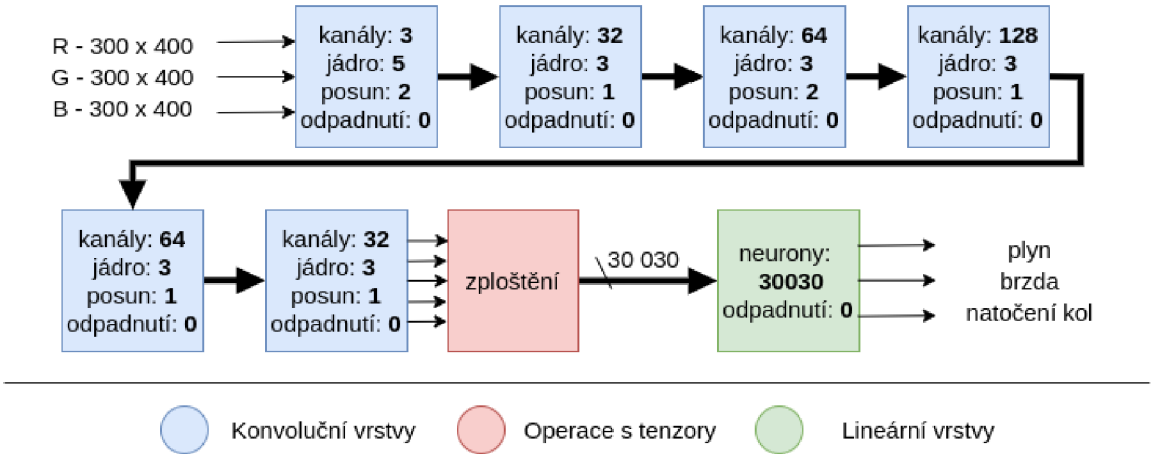
Návrh modelu sítě pro jízdu v pruhu

Oproti předchozím modelům využívá ten, který řeší jízdu v jízdním pruhu pouze přední kameru. Ta je dostatečná pro řešení možných scénářů. Zachytí chodce, semafory a ostatní herce, které jsou před vozidlem.

Výstup kamery je získán z dat senzorů v metodě *forward()*. Tato data jsou odeslána jako vstup neuronové sítě. Z neuronové sítě je získána stejná trojice čísel vyjadřující řízení.

Model je vytvořen v inicializační metodě *___init___()*. Je složen z jedné konvoluční a jedné lineární plně propojené části. Konvoluční část obsahuje pět vrstev. Počet kanálů stoupá postupně od hodnoty tři až ke 128. Poté znovu klesá k velikosti pět. Velikost jádra první vrstvy je 5x5. Všechny ostatní mají standardní velikost 3x3. Je to vytvořeno tak, aby první vrstva hledala souvislosti ve větším rozsahu obrázku. Posun se liší dle vrstev, vždy má hodnotu jedna, nebo dva. Pravděpodobnosti výpadku jsou pro všechny vrstvy vypnuty. Součástí části je i finální zploštění. Po zploštění je ve výstupním tenzoru 30 030 různých čísel.

Tenzor je odeslán lineární plně propojené vrstvě. Ta je velice jednoduchá, obsahuje jednu vrstvu neuronů. Pravděpodobnost výpadku je nastavena na hodnotu nula, tedy žádné výpadky při tréninku. Vrstva zredukuje 30 030 čísel na tři výstupní. Vrstvě je nastaveno, že je poslední. Je bez aktivační funkce. Celkový návrh pro model jízdy v pruhu popisuje obrázek 4.11.



Obrázek 4.11: Návrh neuronové sítě pro jízdu v pruhu.

4.3 Implementace autonomního agenta

Implementovaný autonomní agent je vytvořen v souboru:

`$TEAM_CODE_ROOT/autonomous_agents/agent_base.py`

Pro jeho spuštění musí nejprve běžet CARLA simulátor. Jakmile je tato podmínka splněna, může být spuštěn agent. Pro to je vytvořen skript, jehož aktivace je popsána v příloze B.2.

K jeho spuštění je možné využít připravené scénáře a trasy, které je možné nalézt ve složkách:

```
$TEAM_CODE_ROOT/drive_scenarios
$TEAM_CODE_ROOT/drive_routes
```

Veškeré scénáře jsou uloženy v jednom souboru, který obsahuje veškeré možné dopravní situace na veškerých trasách a městech. Trasy jsou dostupné v několika souborech. Existují soubory s popisem více tras jak testovacích, tak trénovacích. Některé trasy jsou vytažené do vlastních specifických souborů.

Kromě toho se očekává i konfigurační soubor, který nemá definovaný obsah a každý tým do něj může vložit vlastní specifikace. V tomto případě byly jako obsah zvoleny cesty k souborům, kde jsou k nalezení natrénované modely neuronových sítí. Soubor je dostupný na:

```
$TEAM_CODE_ROOT/autonomous_agents/config_file.txt
```

Pro správnou funkcionalitu musí být implementováno v souboru s agentem několik stěžejních částí. První z nich je funkce *get_entry_point()*. Ta vrací řetězec, díky kterému žebříček ví, se kterou třídou autonomního agenta v souboru má pracovat.

Bude to třída *AgentBase*, která dědí od *AutonomousAgent*. Ta byla popsána v předchozí kapitole zabývající se simulací a žebříčkem 3.3. V ní je připraveno několik metod.

Metoda setup

Slouží k hlavní přípravě agenta před zahájením jeho funkce. Nastavuje jeden ze dvou módů. Zde je zvolena možnost senzorů bez využití OpenDrive mapy. Dostane cestu ke konfiguračnímu souboru a provede kontrolu jeho existence.

Otevře konfigurační soubor a vyčte z něj cesty ke třem souborům, ve kterých jsou uloženy natrénované modely neuronových sítí. Jelikož tyto cesty v sobě obsahují proměnné prostředí, jsou zároveň proměnné vyhodnoceny a nahrazeny jejich obsahem. Tím je získána absolutní adresa k souborům.

Provádí se kontrola, jestli existují všechny tři soubory s modely. Pokud neexistují, je vyhozena výjimka agenta jako neplatná cesta k modelu. Když soubory existují provede se za pomoci knihovny PyTorch načtení jejich obsahu.

Po načtení se vytvoří objekty tří neuronových modelů, přičemž každý bude vyhodnocovat výstupy ze senzorů v jiné jízdní situaci (stav řízení). Jsou do nich načteny netrénované a uložené parametry. Modelu bude nastaveno, aby prováděl vyhodnocování. Díky tomu se automaticky vyřadí pomocné vrstvy používané při učení jako je například „dropout“.

```
self._model_cl = Model("model_xdopit03_cl")
self._model_cl.load_state_dict(state_cl['model'])
self._model_cl.eval()
```

Nakonec jsou připraveny proměnné pro další použití. Je to aktuální bod cesty, od kterého se vozidlo vzdaluje. S ním je připravena i poznámka pro index bodu v plánu cesty. Zároveň bude potřeba si pamatovat i bod, ke kterému automobil míří. Nastaví se odchylka od bodu. Označuje maximální vzdálenost mezi automobilem a bodem, která se dá považovat, že vozidlo dorazilo na příslušný bod. Jako poslední jsou připraveny proměnné pro aktuální stav řízení a počet důležitých bodů cesty, po které automobil jede.

Metoda sensors

Zde jsou nastaveny senzory a jejich umístění na automobilu. Rozmístění je stejné jako v případě sběru datové sady 4.1. Nastavení je provedeno jako list senzorů, který je vrácen z metody. Jeden senzor je popsán pomocí slovníku. Podrobný popis, jak nastavit senzory, byl zmíněn v předchozí kapitole 3.3.

Obsahuje tři RGB kamery s rozlišením 800 na 600 pixelů a zorným polem 100 stupňů. Všechny kamery míří ze přední části automobilu. Jedna je ve středové ose, ostatní jsou pootočeny o třicet stupňů na každou stranu. V senzorech je také definován jeden LiDAR, který zabírá celé okolí automobilu do maximální vzdáleností 85 metrů.

```
{'type': 'sensor.camera.rgb', 'id': 'CameraCentral', ... }
{'type': 'sensor.camera.rgb', 'id': 'CameraLeft', ... }
{'type': 'sensor.camera.rgb', 'id': 'CameraRight', ... }
{'type': 'sensor.lidar.ray_cast', 'id': 'Lidar', ... }
```

Další tři senzory jsou senzory zaznamenávající GPS pozici, akceleraci a náklon za pomocí IMU. Také je k využití připraven tachometr.

```
{'type': 'sensor.other.gnss', 'id': 'GPS', ... }
{'type': 'sensor.other.imu', 'id': 'IMU', ... }
{'type': 'sensor.speedometer', 'id': 'Speed'}
```

Poslední jsou nastaveny dva RADARy. Jeden ve předu automobilu a jeden vzadu na středové ose. Oba mají zorné pole nastavené na sto stupňů horizontálně i vertikálně.

```
{'type': 'sensor.other.radar', 'id': 'RADARFront', ... }
{'type': 'sensor.other.radar', 'id': 'RADARBack', ... }
```

Definované senzory jsou předávány simulátoru, ten pořídí záznamy a výsledky vrací metodě *run_step()*

Metoda run_step

Hlavní metoda agenta. Provádí vyhodnocení jednoho kroku simulace. Na vstupu dostane data od simulátoru pořízené ze senzorů.

V prvním kroku zkontroluje, jestli byl nastaven aktuální stav řízení. Ten byl podrobně popsán v kapitole o žebříčku 3.3. Jde například o jízdu v pruhu, projetí křižovatkou doleva, změna pruhu a další. Na základě stavu bude agent měnit modely sítí, které budou řízení vyhodnocovat. Pokud stav nebyl nastaven, znamená to, že se jedná o první volání metody. V takovém případě volá jeho inicializaci. Inicializace nastaví stav řízení, aktuální bod trasy, od kterého se pohybuje, a cílový bod, ke kterému míří.

Poté je ve všech krocích volána metoda pro zpracování výstupu senzorů. Ta vrátí aktuální pozici a data pro neuronovou síť.

Kontroluje se, jestli automobil v aktuálním kroku nedorazil do bodu cesty, ke kterému se navigoval. Ten se nastavil při inicializaci, nebo v předchozích krocích. Pokud do bodu dorazil, je vybrán následující bod trasy a zároveň je změněn stav řízení, který se v bodu cesty mění

Další chování závisí plně na stavu řízení. Dle aktuálně zvoleného stavu se vybere model neuronové sítě, který provede vyhodnocení. Může nastat jeden ze šesti stavů řízení. První z nich je jízda v pruhu. Další tři jsou průjezdy křižovatkou rovno, doleva a doprava. Poslední

dva jsou změny pruhu vlevo a vpravo. Veškeré modely berou na vstupu zpracovaná data ze senzorů z předchozí funkce. Neuronová síť si z nich vybere vlastní potřebné informace. U posledních pěti stavů řízení, tedy u křižovatek a změny pruhu, se před vložením dat přidává informace, v jakém směru probíhá změna.

Výstupem všech modelů neuronových sítí je tenzor tří hodnot. Tyto hodnoty jsou poslány do funkce *_parse_model_output()*. Tato funkce provede poslední úpravu hodnot a vrátí tři reálná čísla vyjadřující natočení kol, sešlápnutí plynu a brzdy.

```
parsed_data, gps_location = self._parse_sensor_data(input_data)
model_output = self._model_lf(parsed_data)
steer, throttle, brake = self._parse_model_output(model_output)
```

Tyto hodnoty se nastaví do objektu třídy *VehicleControl*. Zároveň je i nastaveno vypnutí ruční brzdy. Objekt kontroly vozidla je vrácen jako výstup funkce. Tuto informaci dostane simulátor a dle ní nastaví hodnoty vozidlu před dalším krokem simulace.

Metoda _parse_sensor_data

Funkci jsou předané výstupy ze senzorů. Provede jejich zpracování před tím, než budou poslány do neuronové sítě.

V první části jsou zpracovány výstupy z kamery. Ty jsou zmenšeny na velikost 400x300 pixelů (šířka, výška), pro takové rozlišení je připraven model neuronové sítě. Je provedena transpozice tenzoru. Ta byla podrobně popsána v části zabývající se načítáním dat z datové sady 4.1. Jedná se o stejný postup, kdy je pole pixelů RGBA hodnot přetransformováno do polí hodnot R, G, B a A. Takový formát je zpracovávan neuronovou sítí. Celočíselné hodnoty jsou přetypovány na reálná čísla. NumPy pole obsahující hodnoty je přetypováno na PyTorch tenzor objekt, u kterého je vynechán čtvrtý kanál průsvitnosti. Na závěr je obrázek normován mezi hodnoty nula a jedna.

```
c_image = sensors_data['CameraCentral'][1]
c_image = cv2.resize(c_image, (400,300))
c_image = c_image.transpose(2, 0, 1)
c_image = c_image.astype(np.float)
c_image = torch.Tensor(c_image[0:-1])
c_image /= 255.0
```

Poloha automobilu je zpracována dvakrát. Poprvé zpracování k využití v agentovi jako slovník. Ten se bude používat k porovnávání pozice k jinému bodu cesty, ke kterému auto míří. Druhé zpracování je pro síť. Poloha je přetypována do PyTorch tenzoru. Stejně tak je přetypován i výstup LiDARu, který je zde také využit.

Je získána prvotní a koncová pozice aktuální části cesty. Ta je využívána modely při průjezdech křižovatkou a při změně pruhu. Cílová pozice je pozice následujícího jízdního bodu a startovní pozice je bod, od kterého se automobil nyní vzdaluje.

Veškerá zpracovaná data pro neuronovou síť jsou přidány do slovníku ve formátu, který očekávají modely neuronových sítí.

Návratové hodnoty metody jsou dvě. První z nich je slovník zpracovaných senzorů, který bude poté předán neuronové sítí k vyhodnoceni. Druhá hodnota je slovník s aktuální GPS polohou automobilu.

Metoda _parse_model_output

V metodě je provedena mírná úprava výstupu neuronové sítě. Nejdříve z tenzoru získá všechny tři hodnoty řídící automobil.

Jsou porovnány hodnoty výstupů plynu a brzdy. Pokud má plyn vyšší hodnotu než brzda, je brzdě nastavena hodnota nula. Naopak pokud je více sešlápnuta brzda, nová hodnota plynu je 0. To zapříčiní, že vyšší hodnota bude dominantnější a bude ovládat automobil.

Po úpravě vrací trojici výstupních hodnot. Jde o řízení, hodnotu sešlápnutí plynu a sešlápnutí brzdy.

Metoda _init_agent()

Funkce volána pouze v prvním kroku simulace. Jejím úkolem je příprava agenta k řízení. Agent potřebuje znát aktuální pozici, ke které se naviguje, a stav, ve kterém agent operuje. Funkce provede kontrolu, že cesta obsahuje alespoň dva body (start a cíl).

Pokud kontrolou projde, je volána další metoda, která získá GPS pozici prvního bodu. V tomto bodu je žebříčkem umístěn automobil na začátku simulace. Bod je k dispozici v celkovém plánu na nulté pozici. I tento index je zapamatován pro budoucí použití. Následující cíl získá obdobným způsobem pozici s indexem jedna.

```
self._current_point, self._current_state = self._get_route_point(0)
self._next_point, _ = self._get_route_point(1)
```

Pro vyčtení bodu cesty z globálního plánu za pomoci jeho indexu je použita metoda *_get_route_point()*.

Metoda _get_route_point

Je určena pro vyčtení významného bodu cesty. Je to místo, kde se mění stav řízení. U bodu je označeno jeho umístění na mapě a nový stav řízení, který v tomto bodě začíná.

Metoda nejprve provede kontrolu, že požadovaný index je v rozsahu globálního plánu. Pokud index existuje v globálním plánu, přistoupí se na jeho hodnoty. Metoda vrací dvojici výstupů. První z nich je slovník obsahující GPS zeměpisnou šířku a délku bodu. Druhá návratová hodnota je stav řízení nacházející se od bodu dále.

Metoda _hit_route_point()

Metoda zjišťuje, zdali automobil dorazil do bodu, ke kterému se aktuálně naviguje. Ve třídě je nastavena hodnota odchylky. Pokud se automobil přiblíží k bodu blíž, než je nastavena odchylka, a to jak v rámci zeměpisné délky a šířky, tak je to bráno, že automobil se dostal k cíli.

Jakmile automobil dorazil na cílový bod, je pomocí další funkce nastaven jako výchozí a také je získán nový cílový bod.

Metoda _set_next_point

K nastavení nového navigačního bodu je volána metoda *_set_next_point()*. Ta nastaví předchozí navigační bod jako nový výchozí. Zvedne index aktuálního cílového bodu a přepíše ho dalším bodem z plánu cesty. Také získá nový stav řízení platný od bodu, do kterého automobil právě dorazil.

Kapitola 5

Výstup neuronových sítí, výstup agenta a jeho testování

Tato kapitola se zabývá popisem výstupů veškerých modelů neuronových sítí použitých v řešení projektu. Také jsou zde popsány dosažené výsledky autonomního agenta na množství testovacích tras.

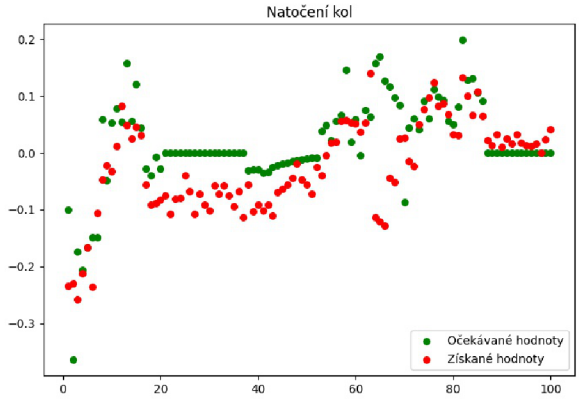
5.1 Hodnocení výstupu neuronových sítí

Pro vyhodnocení výstupu neuronových sítí je možné využít vytvořený validační program, který na svém výstupu produkuje obrázky. U nich je možné lehce porovnat rozdíly mezi referenčními a získanými hodnotami. Pro získání přesvědčivých výsledků validace by použitá datová sada měla být jiná od té na které se provádělo učení. Je to z toho důvodu, že sítě mají přirozený sklon lépe vyhodnocovat data, která znají. Reálných výsledků se dosahuje na datech, která neuronová síť nezná.

Testování sítí je součástí testování samořídícího agenta. Ten provádí stejnou funkci jako validační program pouze s tím rozdílem, že nevyužívá data z testovací datové sady, ale sám je získává ze svých senzorů.

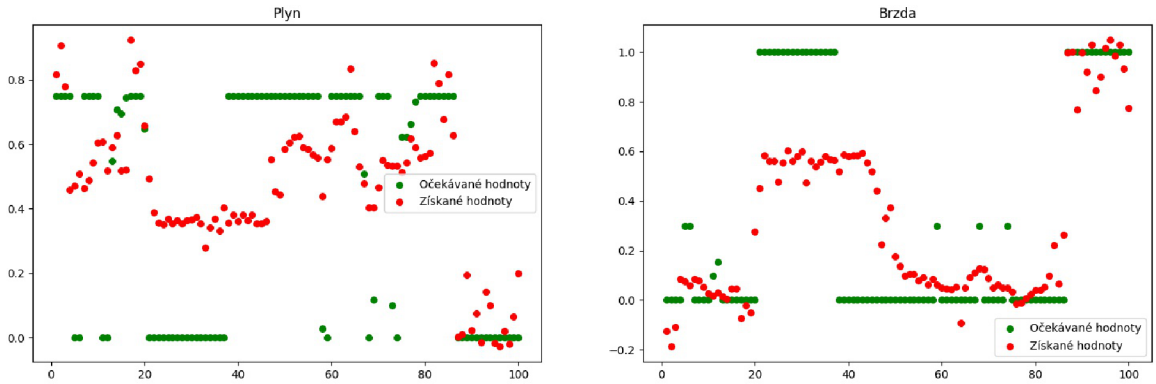
Model pro změnu jízdního pruhu

Výstupy validace tohoto modelu jsou dostupné na obrázcích 5.1 a na 5.2.



Obrázek 5.1: Výstup řízení neuronové sítě pro změnu pruhu.

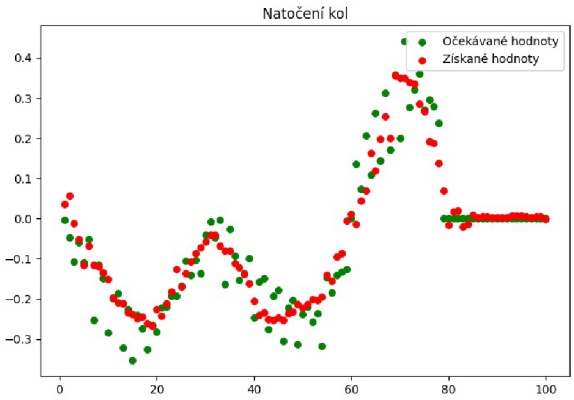
Na obrázku řízení jde pozorovat, že se vyhodnocené natočení kol blíží tomu referenčnímu. To se nedá říct o výstupech plynu a brzdy. Zde provádí autonomní agent úpravu, která porovná hodnoty plynu a brzdy. To docílí toho, že u výstupů v okolí záznamů 30 bude velikost plynu shozena na hodnotu 0. To stejné platí o hodnotách brzdy v opačných případech.



Obrázek 5.2: Výstup plynu a brzdy neuronové sítě pro změnu pruhu.

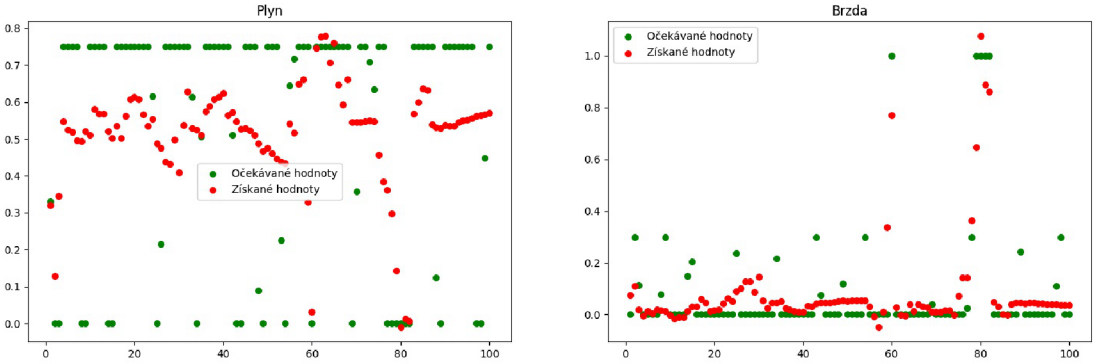
Model pro křižovatky

U tohoto model jde na obrázku 5.3 pěkně pozorovat natočení kol při průjezdu křižovatkou. Na grafu jsou vidět nejprve dvě křižovatky, kdy automobil zatočil vlevo. Na další se vydal doprava. Poslední křižovatka, která je zde vidět je průjezd rovno. To naprosto odpovídá posbírané datové sadě, kde tato sekvence opravdu existuje.



Obrázek 5.3: Výstup řízení neuronové sítě pro křižovatky.

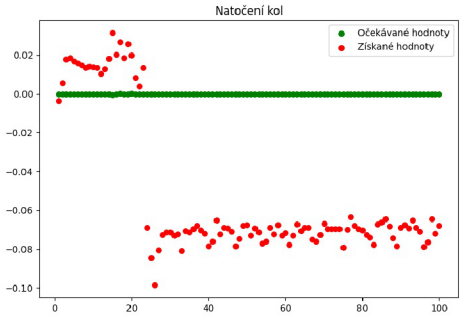
U následujícího obrázku 5.4 je již situace horší. Z nějakého důvodu agent, který sbíral datovou sadu při průjezdu křižovatkou v pravidelných krátkých intervalech nedržel sešlápnutý plynový pedál. To bylo vyřešené učením, kdy se neuronová síť učila každým třetím záznamem datové sady. Myšlenka za tímto byla neposkytnou neuronové sadě k učení dva vstupy, které následují za sebou (velice podobná data ze senzorů) a mají naprosto odlišné referenční hodnoty. I přesto tyto náhodné zastavení táhnou celkový výstup plynu dolů. Hodnoty brzdy jsou dostatečné, jelikož je zde autonomní agent zaokrouhlí na hodnotu nula.



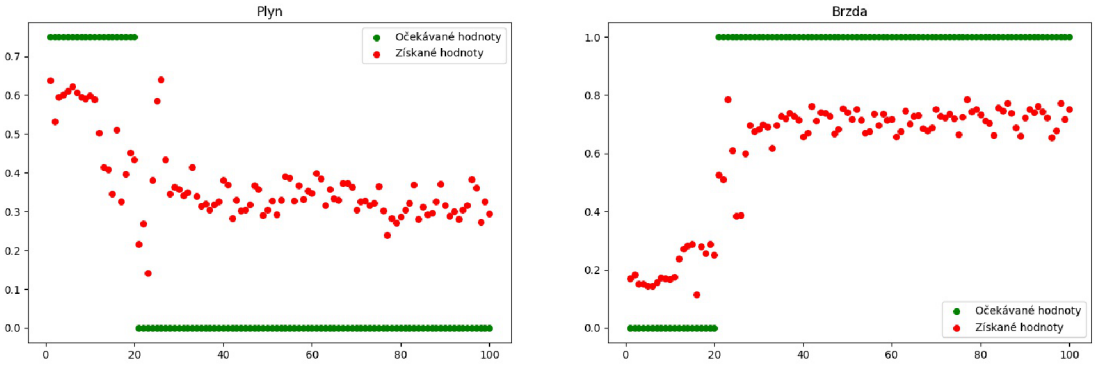
Obrázek 5.4: Výstup plynu a brzdy neuronové sítě pro křižovatky.

Model pro jízdu v pruhu

Na první pohled může natočení kol na obrázku 5.5 působit jako naprosto mimo. Je důležité si zde všimnou hodnot na svislé ose. Výstupy jsou o jeden řád níž a jsou tak velice blízko nule. Hodnoty se blíží nule, jelikož automobil jel v pruhu po přímé linii.



Obrázek 5.5: Výstup řízení neuronové sítě pro jízdu v pruhu.



Obrázek 5.6: Výstup plynu a brzdy neuronové sítě pro jízdu v pruhu.

Obrázek 5.6 popisuje plyn a brzdu modelu, který řeší průjezd jízdními pruhy. Agent zde při sběru datové sady buďto viděl překážku a brzdil, nebo žádnou nezaznamenal a naplno jel. U výstupu sítě zde pozorovat znatelná odchylka od očekávaných hodnot. Zde zase přichází porovnání, které provádí autonomní agent a zaokrouhlí hodnoty do očekávaných hodnot.

5.2 Testování schopností autonomního agenta

K samotnému testování agenta je využit CARLA žebříček. Ten sám o sobě funguje jako způsob, který dokáže ohodnotit úspěšnost implementace. Potřebuje plány cest, na kterých má být agent testován. Ty jsou získány z příkladů, které žebříček obsahuje. Každá cesta je vložena do samostatného souboru ve složce projektu *drive_routes* a popisuje jeden z testů, jehož výsledky jsou popsány níže.

Test 1

Automobil při tomto testu dorazil ke křižovatce, kde měl červenou. Na ní úspěšně zastavil. V okamžiku, kdy padla zelená, se rozjel a projel vlevo křižovatkou. Několik metrů za křižovatkou automobil zastavil a nepokračoval.

Soubor s testem: routes_1_test.xml
Město: Town01
Splněno trasy: 5,94%
Kolize: 0
Projetí semaforu na červenou: 0
Projetí stopky: 0

Test 2

V tomto testu začal automobil v koloně. Přiblížil se k předchozímu autu a zastavil. Jakmile se na semaforu objevila zelená barva, rozjel se a přijel k semaforu. Zde se automobil zasekl a dále nepokračoval. Z pozorování to bylo z důvodu stojících automobilů v protisměru.

Soubor s testem: routes_2_test.xml
Město: Town01
Splněno trasy: 2,51%
Kolize: 0
Projetí semaforu na červenou: 0
Projetí stopky: 0

Test 3

Při tomto testu automobil zvládne úspěšně jet v jízdním pruhu. Dojde k zaváhání v případě, kdy se v protisměru objeví jiný automobil. Po projetí se zotaví a pokračuje. Na určitém bodě se zasekne (pravděpodobně nečistoty na vozovce).

Soubor s testem: routes_3_test.xml
Město: Town01
Splněno trasy: 0,91%
Kolize: 0
Projetí semaforu na červenou: 0
Projetí stopky: 0

Test 4

Test 4 na začátku provedl úspěšné rozjetí automobilu, projetí semaforu na zelenou, odbočení vpravo. Po projetí křižovatky dokázal srovnat jízdu v pruhu. Poté se z neznámých důvodů zastavil.

Soubor s testem: routes_4_test.xml
Město: Town01
Splněno trasy: 9,02%
Kolize: 0
Projetí semaforu na červenou: 0
Projetí stopky: 0

Test 5

Při tomto testu se automobil nedokázal rozjet.

Soubor s testem: routes_5_test.xml
Město: Town01
Splněno trasy: 0,0%
Kolize: 0
Projetí semaforu na červenou: 0
Projetí stopky: 0

Test 6

Automobil se rozjede a jede v jízdním pruhu. Před křižovatkou zastaví před automobilem, který stojí před ním. Nepodaří se mu zotavit po odjetí automobilu v čele kolony.

Soubor s testem: routes_6_test.xml
Město: Town01
Splněno trasy: 27,94%
Kolize: 0
Projetí semaforu na červenou: 0
Projetí stopky: 0

Test 7

V tomto městě se automobilu nepodařilo rozjet. Test vedl k naprostému selhání.

Soubor s testem: routes_7_test.xml
Město: Town03
Splněno trasy: 0,0%
Kolize: 0
Projetí semaforu na červenou: 0
Projetí stopky: 0

Výsledek

Z testování vyplývá, že neuronová síť, která vyhodnocuje jízdu v pruhu je špatná. Automobil se velice často zasekává. Přehnaně reaguje na automobily, které jedou v protějším směru. Také nezvládá jiná města než město *Town01*. Velkým plusem vyplývajícím z testování je, že automobil velice dobře reaguje na překážky. Je schopný zastavit před automobilem, cyklistou i chodcem. To stejné platí o červené barvě na semaforech. Naopak v některých případech má problém se po zastavení zotavit a navázat na předchozí řízení.

Kapitola 6

Závěr

V této práci bylo provedeno seznámení s neuronovými sítěmi. Nejprve se čtenář dozvěděl základní informace a matematickou podstatu, na které jsou sítě postaveny. Bylo provedeno vysvětlení senzorů, které automobily využívají jako vstupy neuronových sítí. Poté čtenář získal složitější informace jako jsou algoritmy gradientního sestupu a zpětné propagace pro použití při jejich učení.

V další kapitole se čtenáři dozvěděli informace o nástrojích CARLA a prostředcích pro jednoduchou tvorbu a testování autonomních automobilů. K testování byly shrnuty cíle CARLA žebříčku a metriky, které používá k hodnocení spolehlivosti. Také zde byla popsána konkrétní metoda, jak je možné vytvořit program pro řízení, která zahrnovala vysvětlení všech dostupných senzorů, cestovních plánů a metod potřebných k implementaci.

Na základě takto představených informací bylo cílem práce navrhnout a vytvořit konkrétní programy a další součástí, které by dokázaly samostatně řídit automobil. Problém byl rozložen na několik částí.

Prvním úkolem bylo získání datové sady. K tomuto účelu bylo vytvořeno několik programů. Některé řídí vozidlo, jiné sbírají informace ze senzorů, které ukládají do permanentních souborů. Pro použití datové sady v návrhu byl vytvořen způsob, který dokáže posbíraná data roztřídit na základě jízdních situací. Cíle této části byly jednoznačně úspěšně naplněny.

Další částí bylo vytvoření neuronové sítě. Zde byl použit návrh, který použil tři rozdílné modely určené na řízení jiného aspektu trasy, jako například průjezd křižovatky. Byly vytvořeny programy, které umožňují trénink a testování těchto návrhů. I v tomto případě bylo dosaženo vytyčených cílů.

Modely neuronových sítí byly učeny k použití v programu, který je označován jako autonomní agent. Ten je schopný komunikovat se simulátorem a řídit vozidlo. Použije naučené sítě a spojí je s dalšími informacemi. Na základě plánu trasy přenechává řízení jiným návrhům neuronových sítí. Agent splnil očekávání, i přesto automobil selhal v různých částech trati. Především se jedná o záseky v částech jízdy v pruhu. Pozitivum je, že vozidlo nezaznamenává kolize, nebo průjezdy na červenou.

Další vývoj této práce by se měl zaměřit na úpravu neuronové sítě pro jízdu v pruhu, ta předvádí nejslabší výsledky. K tomu by bylo vhodné získat datovou sadu, která obsahuje speciální útržky jízd s ojedinělými situacemi. Při aktuálním způsobu sběru dat nedochází například k nasnímání dostatku nečistot na silnici.

Literatura

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Dostupné z: https://www.tensorflow.org/.

[2] ASAM e. V.. *OpenDRIVE 1.6.1*. March 2021. Dostupné z: https://www.asam.net/index.php?eID=dumpFile&t=f&f=4089&token=deea5d707e2d0edeeb4fccd544a973de4bc46a09.

[3] Barnard, M. *Tesla & Google Disagree About LIDAR — Which Is Right?* [online]. Červenec 2016 [cit. 2022-01-24]. Dostupné z: https://cleantechnica.com/2016/07/29/tesla-google-disagree-lidar-right/.

[4] Brilliant.org. *Backpropagation* [online]. Srpen 2016 [cit. 2022-05-06]. Dostupné z: https://brilliant.org/wiki/backpropagation/.

[5] Campbell, S., O'Mahony, N., Krpalcova, L., Riordan, D., Walsh, J. et al. Sensor Technology in Autonomous Vehicles : A review. In: *2018 29th Irish Signals and Systems Conference (ISSC)*. 2018, s. 1–4. DOI: 10.1109/ISSC.2018.8585340.

[6] CARLA team. *CARLA 0.8.4 Data Collector*. 2018. Dostupné z: https://github.com/carla-simulator/data-collector.

[7] CARLA team. *CARLA Autonomous Driving Leaderboard* [online]. únor 2020 [cit. 2022-05-03]. Dostupné z: https://leaderboard.carla.org/.

[8] CARLA team. *CARLA Simulator*. Září 2020. Dostupné z: https://carla.readthedocs.io/en/0.9.10/.

[9] CARLA team. *CARLA Scenario Runner*. 2022. Dostupné z: https://github.com/carla-simulator/scenario_runner.

[10] Chauhan, N. S. *Loss Functions in Neural Networks* [online]. Srpen 2021 [cit. 2022-05-06]. Dostupné z: https://www.theaidream.com/post/loss-functions-in-neural-networks.

[11] Choi, R. Y., Coyner, A. S., Kalpathy Cramer, J., Chiang, M. F. a Campbell, J. P. Introduction to Machine Learning, Neural Networks, and Deep Learning. *Translational Vision Science & Technology*. 9. vyd. Únor 2020, č. 2, s. 14. DOI: 10.1167/tvst.9.2.14. ISSN 2164-2591. Dostupné z: https://tvst.arvojournals.org/article.aspx?articleid=2762344.

[12] Chollet, F. et al. *Keras*. 2015. Dostupné z: https://keras.io.

[13] Deschaud, J.-E. KITTI-CARLA: a KITTI-like dataset generated by CARLA Simulator. *ArXiv e-prints*. 2021.

[14] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. a Koltun, V. CARLA: An Open Urban Driving Simulator. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, s. 1–16.

[15] Hagan, M. a Demuth, H. Neural networks for control. In: *Proceedings of the 1999 American Control Conference (Cat. No. 99CH36251)*. 1999, sv. 3, s. 1642–1656. DOI: 10.1109/ACC.1999.786109. ISSN 0743-1619.

[16] Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P. et al. Array programming with NumPy. *Nature*. Springer Science and Business Media LLC. září 2020, sv. 585, č. 7825, s. 357–362. DOI: 10.1038/s41586-020-2649-2. Dostupné z: https://doi.org/10.1038/s41586-020-2649-2.

[17] Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*. IEEE COMPUTER SOC. 2007, sv. 9, č. 3, s. 90–95. DOI: 10.1109/MCSE.2007.55.

[18] Kingma, D. P. a Ba, J. *Adam: A Method for Stochastic Optimization*. arXiv, 2014. DOI: 10.48550/ARXIV.1412.6980. Dostupné z: https://arxiv.org/abs/1412.6980.

[19] Kocić, J., Jovičić, N. a Drndarević, V. Sensors and Sensor Fusion in Autonomous Vehicles. In: *2018 26th Telecommunications Forum (TELFOR)*. 2018, s. 420–425. DOI: 10.1109/TELFOR.2018.8612054.

[20] McCulloch, W. S. a Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*. 5. vyd. 1943, č. 4, s. 115–133.

[21] Nielsen, M. *Neural Networks and Deep Learning*. Determination Press, 2015. Dostupné z: http://neuralnetworksanddeeplearning.com/.

[22] Nwankpa, C., Ijomah, W., Gachagan, A. a Marshall, S. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. arXiv, 2018. DOI: 10.48550/ARXIV.1811.03378. Dostupné z: https://arxiv.org/abs/1811.03378.

[23] Ognjanovski, G. *Everything you need to know about Neural Networks and Backpropagation — Machine Learning Easy and Fun* [online]. Leden 2019 [cit. 2022-01-24]. Dostupné z: https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

[24] On-Road Automated Driving (ORAD) committee. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Standard J3016_202104. SAE International, 2016. Dostupné z: https://saemobilus.sae.org/content/J3016_202104/.

[25] O'Shea, K. a Nash, R. *An Introduction to Convolutional Neural Networks*. arXiv, 2015. DOI: 10.48550/ARXIV.1511.08458. Dostupné z: https://arxiv.org/abs/1511.08458.

[26] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J. et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach, H., Larochelle, H., Beygelzimer, A., Alché Buc, F. d', Fox, E. et al., ed. *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, s. 8024–8035. Dostupné z: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[27] Policie ČR. *Statistika nehodovosti* [online]. Červen 2008 [cit. 2022-01-24]. Dostupné z: http://www.policie.cz/clanek/statistika-nehodovosti-900835.aspx.

[28] Raghuvanshi, D. Artificial Intelligence: Basics and Terminology. *International Journal of Trend in Scientific Research and Development (ijtsrd)*. 2. vyd. Říjen 2018, č. 6, s. 1539–1541. Dostupné z: https://www.ijtsrd.com/papers/ijtsrd18909.pdf.

[29] Rasamoelina, A. D., Adjailia, F. a Sinčák, P. A Review of Activation Function for Artificial Neural Network. In: *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. 2020, s. 281–286. DOI: 10.1109/SAMI48414.2020.9108717. ISBN 978-1-7281-3149-8.

[30] Ruder, S. *An overview of gradient descent optimization algorithms*. arXiv, 2016. DOI: 10.48550/ARXIV.1609.04747. Dostupné z: https://arxiv.org/abs/1609.04747.

[31] Rumelhart, D. E., Hinton, G. E. a Williams, R. J. Learning representations by back-propagating errors. *Nature*. 323. vyd. 1986, č. 6088, s. 533–536. DOI: 10.1038/323533a0. ISSN 1476-4687. Dostupné z: https://doi.org/10.1038/323533a0.

[32] Santurkar, S., Tsipras, D., Ilyas, A. a Madry, A. How Does Batch Normalization Help Optimization? In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa Bianchi, N. et al., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2018, sv. 31. Dostupné z: https://proceedings.neurips.cc/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf.

[33] Sarker, I. H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. *SN Computer Science*. 2. vyd. Srpen 2021, č. 6, s. 420. DOI: 10.1007/s42979-021-00815-1. ISSN 2661-8907. Dostupné z: https://doi.org/10.1007/s42979-021-00815-1.

[34] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. a Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 15. vyd. 2014, č. 56, s. 1929–1958. Dostupné z: http://jmlr.org/papers/v15/srivastava14a.html.

[35] Weng, X., Man, Y., Cheng, D., Park, J., O'Toole, M. et al. All-In-One Drive: A Large-Scale Comprehensive Perception Dataset with High-Density Long-Range Point Clouds. *ArXiv*. 2020.

[36] Yamashita, R., Nishio, M., Do, R. K. G. a Togashi, K. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*. 9. vyd. 2018, č. 4, s. 611–629. DOI: 10.1007/s13244-018-0639-9. ISSN 1869-4101. Dostupné z: https://doi.org/10.1007/s13244-018-0639-9.

[37] Zhang, A., Lipton, Z. C., Li, M. a Smola, A. J. *Dive into Deep Learning*. arXiv, 2021. DOI: 10.48550/ARXIV.2106.11342. Dostupné z: https://arxiv.org/abs/2106.11342.

Příloha A

Obsah přiloženého paměťového média

K bakalářské práci je přiložena paměťová SD karta o velikosti 32 GB. Její obsah je rozdělen do tří složek a jednoho souboru.

Tabulka A.1: Popis paměťového média.

Složka	Popis
bp_latex_source	Složka obsahující LaTeX zdrojové kódy pro technickou zprávu bakalářské práce.
dataset_example	Kousek z ukázky vlastní posbírané datové sady pro učení neuronových sítí z CARLA simulátoru.
team_code_implementation	Zdrojové kódy a skripty implementace projektu.
bp_xdopit03.pdf	Technická zpráva.
poster.pdf	Plakát k projektu.

Vnitřní struktura složky s LaTeX soubory technické zprávy má následující vnitřní strukturu.

- **./bib-styles/Pysny/*.bst**
  Styly pro dokument.
- **./figures/implementation/*.png**
  Obrázky pro kapitolu implementace autonomního agenta.
- **./figures/neral_networks/*.png**
  Obrázky pro kapitolu o neuronových sítích.
- **./figures/simulators/*.png**
  Obrázky pro kapitolu o CARLA simulátoru a žebříčku.
- **./figures/test/*.png**
  Obrázky pro kapitolu o testování.
- **./template-fig/***
  Obrázky pro šablonu dokumentu.
- **./appendix-01-media-content.tex**
  Obsah první přílohy pro obsah paměťového média.

- **./appendix-02-manual.tex**
  Obsah druhé přílohy s manuálem instalace a spouštění.
- **./bibliography.bib**
  Bibliografické citace BibTeX.
- **./bp-xdopit03.tex**
  Hlavní soubor pro práci.
- **./chapter-01-introduction.tex**
  Obsah první kapitoly s úvodem.
- **./chapter-02-naural_networks.tex**
  Obsah druhé kapitoly s neuronovými sítěmi.
- **./chapter-03-simulators.tex**
  Obsah třetí kapitoly se simulátory.
- **./chapter-04-autonomous_agent.tex**
  Obsah čtvrté kapitoly s implementací autonomního agenta.
- **./chapter-05-tests.tex**
  Obsah páté kapitoly s testováním.
- **./chapter-06-conclusion.tex**
  Obsah šesté kapitoly se závěrem.
- **./fitthesis.cl**
  Definice vzhledu souboru pro bakalářskou práci.
- **./zadani.pdf**
  Zadání bakalářské práce.
- **./Makefile**
  Informace, jak generovat výstup.

Vnitřní struktura složky s ukázkou vlastní posbírané datové sady má následující strukturu.

- **./change_lane/episode_0010/**
  - **part_00000_left/***
    Záznamy ze senzorů z první změny jízdního pruhu vlevo.
  - **part_00001_left/***
    Záznamy ze senzorů z druhé změny jízdního pruhu vlevo.
- **./junction/episode_0010/**
  - **part_00000_left/***
    Záznamy ze senzorů z jízdy první křižovatkou vlevo.
  - **part_00001_straight/***
    Záznamy ze senzorů z jízdy druhou křižovatkou rovně.
  - **part_00002_right/***
    Záznamy ze senzorů z jízdy třetí křižovatkou vpravo.

  - **part_00003_right/***
    Záznamy ze senzorů z jízdy čtvrtou křižovatkou vpravo.
  - **part_00004_left/***
    Záznamy ze senzorů z jízdy pátou křižovatkou vlevo.
  - **part_00005_left/***
    Záznamy ze senzorů z jízdy šestou křižovatkou vlevo.

- **./lane_folow/episode_0010/**
  - **part_0000***
    Záznamy ze senzorů ve více částích, ve kterých jel automobil v jízdím pruhu.

- **./preloads**
  Prázdná složka, kam budou vygenerovány přednačtené soubory.

Vnitřní struktura složky s implementací programů pro bakalářskou práci. Obsahuje sběr datové sady, návrh neuronových sítí, učící a testovací programy, natrénované modely sítí a program pro autonomní řízení vozidla.

- **./autonomous_agents/**
  - **agent_base.py**
    Soubor obsahující implementaci autonomního agenta.
  - **config_file.txt**
    Konfigurační soubor pro autonomního agenta.

- **./datasets/**
  - **dataset_collector_leaderboard_agent.py**
    Soubor obsahující implementaci agenta pro sběr datové sady.
  - **dataset_collector_simulator_agent.py**
    Soubor obsahující implementaci agenta pro řízení automobilu při sběru datové sady.
  - **dataset_reshape_program_example.txt**
    Ukázka vstupu pro skript na třídění datové sady.
  - **dataset_collector_simulator_agent.py**
    Šablona vstupu pro skript na třídění datové sady.
  - **main.py**
    Rozcestník pro datové sady.
  - **xdopit03_dataset_loader.py**
    Třída pro přístup k datové sadě.

- **./drive_results/**
  Složka, do které budou vkládány výsledky řízení agentů.

- **./drive_routes/routes***
  Soubory obsahující definice cest.

- **./drive_scenarios/all.json**
  Soubor se scénáři dopravních situací.

- **./neural_network/**
  - **main.py**
    Rozcestník pro přístup k modelům neuronových sítí a chybových funkcí.
  - **nn_convolutional_part.py**
    Třída obsahující definice konvoluční části neuronové sítě.
  - **nn_fully_conn_part.py**
    Třída obsahující definice lineárních plně propojených částí neuronové sítě.
  - **nn_model_xdopit03_cl.py**
    Třída obsahující definici neuronové sítě pro změnu pruhu.
  - **nn_model_xdopit03_ju.py**
    Třída obsahující definici neuronové sítě pro křižovatky.
  - **nn_model_xdopit03_lf.py**
    Třída obsahující definici neuronové sítě pro jízdu v pruhu.
- **./README.md**
  Soubor s informacemi o projektu.
- **./requirements.txt**
  Popis knihoven k instalaci.
- **./scripts/**
  - **collect_dataset.sh**
    Skript pro spuštění sběru dat.
  - **reshape_dataset_part.sh**
    Skript pro úpravu části datové sady.
  - **reshape_dataset.sh**
    Skript pro úpravu datové sady.
  - **run_agent.sh**
    Skript pro spuštění CARLA leaderboard agenta.
  - **run_simulator.sh**
    Skript pro spuštění CARLA simulátoru.
  - **run_training.sh**
    Skript pro spuštění tréninku neuronové sítě.
  - **run_validation.sh**
    Skript pro spuštění testování neuronové sítě.
- **./trained_models/**
  - **xdopit03_trained_model_cl.pt**
    Naučené hodnoty modelu neuronové sítě pro změnu pruhu.
  - **xdopit03_trained_model_ju.pt**
    Naučené hodnoty modelu neuronové sítě pro křižovatky.
  - **xdopit03_trained_model_lf.pt**
    Naučené hodnoty modelu neuronové sítě pro jízdu v pruhu.

- **./training/**
  - **run_training.py**
    Program pro trénink neuronové sítě.
  - **run_validation.py**
    Program pro testování neuronové sítě.
- **./utilities/**
  - **exceptions.py**
    Speciální třídy výjimek.
  - **main.py**
    Speciální funkce používané v projektu.

Příloha B

Manuál

Tato příloha slouží jako manuál k instalaci a ovládání veškerých komponent, které byly vytvořeny pro bakalářskou práci. Instalace je popsána pro operační systém Linux. Doporučená verze je Ubuntu 18.04 LTS s Python verzí 3.6.9.

B.1 Instalace CARLA programů a kódu týmu (projektu)

Nejdříve je potřeba nainstalovat veškeré programy dostupné od společnosti CARLA. Jedná se o simulátor, skripty pro žebříček a pro spouštění scénářů. Jakmile jsou CARLA komponenty funkční, přijde čas na zprovoznění programů, které byly vytvořeny pro trénink a řízení. Poté je připraveno místo na datovou sadu. Jakmile je vše připravené, dojde k namapování všech komponent přes proměnné prostředí, přes které se k nim bude přistupovat.

Instalace CARLA simulátoru

1. V projektu je využíván CARLA simulátor ve verzi 0.9.10.1. Ten je možné stáhnout na odkaze v zápatí. [1]

2. Po kliknutí na odkaz je stažen *tar.gz* soubor, ten stačí rozbalit a umístit na Vámi zvolené místo v počítači. Toto místo bude v instalaci i projektu odkazováno jako *$CARLA_ROOT*.

3. Součástí simulátoru je i Python API, které obsahuje pomocné programy a agenty pro ovládání simulátoru. Využívají Python knihoven, které je nutné nainstalovat.

```
cd $CARLA_ROOT
pip3 install -r PythonAPI/carla/requirements.txt
```

4. Jakmile je simulátor nainstalován, je potřeba k němu přidat rozšiřující mapy, které nejsou v základní verzi. Mapy je možné stáhnout zde. [2]

5. V momentě, kdy jsou mapy stažené, stačí přesunout *tar.gz* soubor do složky simulátoru určené pro import.

```
mv ~/Downloads/AdditionalMaps_0.9.10.1.tar.gz $CARLA_ROOT/Import
```

[1] https://carla-releases.s3.eu-west-3.amazonaws.com/Linux/CARLA_0.9.10.1.tar.gz
[2] https://carla-releases.s3.eu-west-3.amazonaws.com/Linux/AdditionalMaps_0.9.10.1.tar.gz

6. Následné stačí spustit skript, který mapy rozbalí a vloží je do simulátoru.

```
$CARLA_ROOT/ImportAssets.sh
```

Instalace skriptů CARLA Leaderboard

1. Do Vámi zvoleného umístění na počítači se naklonuje CARLA leaderboard z GITu.

```
git clone -b stable --single-branch \
https://github.com/carla-simulator/leaderboard.git
```

   Složka se skripty žebříčku bude odkazována v instalaci i projektu jako *$LEADERBOARD_ROOT*.

2. Pro skripty je nutné nainstalovat závislosti. Ty jsou definované v připraveném programu.

```
cd $LEADERBOARD_ROOT
pip3 install -r requirements.txt
```

Instalace skriptů Scenario Runner

1. Do zvoleného umístění na počítači se naklonuje repositář Scenario Runner.

```
git clone -b leaderboard --single-branch \
https://github.com/carla-simulator/scenario_runner.git
```

   Složka se skripty je odkazována jako *$SCENARIO_RUNNER_ROOT*.

2. I zde jsou v připraveném souboru definované závislosti.

```
cd $SCENARIO_RUNNER_ROOT
pip3 install -r requirements.txt
```

Instalace týmového kódu

1. Do zvoleného adresáře na počítači se z přiloženého média nakopíruje obsah složky *team_code_implementation*. Také je možné projekt naklonovat z GITu.

```
git clone -b main --single-branch \
https://gitlab.com/xdopit03/carla-autonomous-agent.git
```

   Adresář pro projekt je odkazován jako *$TEAM_CODE_ROOT*.

2. Pro instalaci závislých knihoven je připraven soubor.

```
cd $TEAM_CODE_ROOT
pip3 install -r requirements.txt
```

Příprava na datovou sadu

1. Datová sada je posbíraná pomocí programů připravených jako součást kódu týmu. Je nutné pro ni vybrat umístění v počítači, toto umístění bude odkazované jako *$DATASETS_ROOT*.

Nastavení proměnných prostředí

1. Již zmíněné proměnné všech částí se nyní nastaví jako proměnné prostředí operačního systému. Stačí upravit soubor *.bashrc*.

```
gedit ~/.bashrc
```

2. Na konec souboru stačí přidat několik řádků, které nastavují proměnné do systému. Nejdříve se nastaví proměnné, které budou obsahovat cestu k částem projektu. Stačí změnit cesty k Vašim umístěním.

```
export CARLA_ROOT=CESTA_K_SIMULATORU
export SCENARIO_RUNNER_ROOT=CESTA_KE_SCENARIO_RUNNER
export LEADERBOARD_ROOT=CESTA_KE_CARLA_LEADERBOARD
export TEAM_CODE_ROOT=CESTA_K_IMPLEMENTACI_TYMU
export DATASETS_ROOT=CESTA_K_DATOVE_SADE
```

3. Nakonec stačí upravit *PYTHONPATH*. Je upravena tak, aby Python vždy hledal balíčky i v cestách CARLA programů. Ty budou nyní přístupné odkudkoli v systému.

```
export PYTHONPATH="${CARLA_ROOT}/PythonAPI/carla/":\
"${SCENARIO_RUNNER_ROOT}":"${LEADERBOARD_ROOT}":\
"${CARLA_ROOT}/PythonAPI/carla/dist/"\
"carla-0.9.10-py3.7-linux-x86_64.egg":\
"${TEAM_CODE_ROOT}":"${PYTHONPATH}"
```

4. Následně stačí aktualizovat změny v souboru do terminálu.

```
source ~/.bashrc
```

Nyní jsou všechny komponenty plně funkční a programy jsou připraveny ke spuštění.

B.2 Spouštění komponent projektu

Jakmile je vše nainstalováno může se přistoupit k práci. Součásti projektu je několik spolu nesouvisejících programů. Každý se stará o jinou část potřebnou k vytvoření plně autonomního agenta. Pro všechny tyto operace jsou vytvořeny bashové skripty umožňující jednoduší spouštění. Ty jsou uloženy ve složce:

`$TEAM_CODE_ROOT/scripts/`

Je zde k nalezení skript umožňující spouštění CARLA simulátoru. Také jsou zde programy pro spouštění sběru a úpravy datové sady. Zapnutí tréninku a validace neuronových sítí a samotný start autonomního agenta.

Spouštění CARLA simulátoru

Pro spuštění je připraven bashový skript:

`$TEAM_CODE_ROOT/scripts/run_simulator.sh`

Ten očekává jeden parametr, který určuje úroveň kvality simulace. Pro horší kvalitu je možné použít „Low“, naopak pro kvalitu vysokou bude mít parametr hodnotu „Epic“. Pokud se počítá se získáváním kamerových záznamů, vždy se musí skript spouštět s hodnotou „Epic“.

```
$TEAM_CODE_ROOT/scripts/run_simulator.sh Epic
$TEAM_CODE_ROOT/scripts/run_simulator.sh Low
```

Spouštění sběru datové sady

Před spuštěním sběru dat musí dojít k zapnutí CARLA simulátoru. Způsob jeho zapnutí byl popsán výše B.2.

Jakmile simulátor běží je možné spustit agenta vytvořeného pro sběr dat. K tomu je dostupný skript:

`$TEAM_CODE_ROOT/scripts/collect_dataset.sh`

Očekává jeden parametr. Parametr vyjadřuje složku v úložišti datových sad. Umístění úložiště musí být proměnná prostředí exportovaná jako *$DATASETS_ROOT*. Parametr poté bude složka „folder“ s umístěním:

`$DATASETS_ROOT/folder`

```
$TEAM_CODE_ROOT/scripts/collect_dataset.sh folder
```

Popis úpravy datové sady a její spuštění

Pro transformaci datové sady jsou připraveny dva skripty. Přičemž jeden je spouštěn z toho druhého. Ten je k nalezení na:

`$TEAM_CODE_ROOT/scripts/reshape_dataset.sh`

Aby skript věděl, jak přetransformovat datovou sadu, očekává na standardním vstupu textový soubor, který popisuje jeho chování. Ten je nutné vytvořit za pomoci fotografií datové sady. Na prvním řádku souboru musí být definováno umístění datové sady v *$DATASETS_ROOT*. Druhý řádek je cesta do složky, která je také v umístění datových sad, kam se mají data převést.

Na další řádky se zadává popis částí, které specifikují jízdní situace. Jako první se popíše, o jakou epizodu datové sady jde pomocí *episode_* a jejího identifikátoru. Druhý parametr definuje jednu ze tří jízdních situací (*lane_follow*, *change_lane*, *junction*). Ta je zjištěna z kamerových výstupů. Třetím argumentem je část. Ta popisuje o kolikátou jízdní sekvenci dané situace v epizodě se jedná. Musí mít příponu (*_left*, *_straight*, *_right*) v případě změny pruhu (pouze vlevo a vpravo) a křižovatky. Poslední dva parametry je číslo záznamu, kdy sekvence začala a skončila.

```
/folder/
/new_folder
episode_00000 lane_follow  part_00000   00000 00657
episode_00000 junction  part_00000_left  00658 00746
episode_00000 lane_follow  part_00001   00747 03153
episode_00000 junction  part_00001_left  03154 03244
episode_00000 lane_follow  part_00002   03245 04013
```

Jakmile je textový soubor připraven, může se provést úprava datové sady.

```
$TEAM_CODE_ROOT/scripts/reshape_dataset.sh < config_file.txt
```

Skript v cyklu spouští pomocný program:

`$TEAM_CODE_ROOT/scripts/reshape_dataset_part.sh`

Spouštění tréninku neuronové sítě

Pro učení neuronových sítí je připraven skript, pomocí kterého je zvolen patřičný model k učení:

`$TEAM_CODE_ROOT/scripts/run_training.sh`

Ke správné funkcionalitě skript potřebuje tři argumenty. První argument je označení modelu, který bude trénován. Očekává zadání jednoho z *xdopit03_cl* pro změnu pruhu, *xdopit03_ju* pro křižovatky a *xdopit03_lf* pro jízdu v pruhu. Druhý argument je mnohdy označován jako počet epoch. Epocha znamená jeden průchod tréninku nad celou datovou sadou. Posledním z argumentů je cesta k datové sadě. Skript automaticky přidává příponu umístění *$DATASETS_ROOT*.

```
$TEAM_CODE_ROOT/scripts/run_training.sh xdopit03_cl 50 dataset_folder
$TEAM_CODE_ROOT/scripts/run_training.sh xdopit03_lf 30 dataset_folder
$TEAM_CODE_ROOT/scripts/run_training.sh xdopit03_ju 15 dataset_folder
```

Spouštění validace neuronové sítě

I pro validaci existuje předpřipravený skript k jejímu spuštění:

`$TEAM_CODE_ROOT/scripts/run_validation.sh`

Zde jsou očekávaný argumenty dva. Opět to je jako v případě tréninku jméno trénovaného modelu (*xdopit03_cl*, *xdopit03_ju*, *xdopit03_lf*). Druhým argumentem je cesta k datové sadě. Oproti tréninku zde není počet epoch. Validace probíhá vždy pouze na jednom průchodu datovou sadou.

```
$TEAM_CODE_ROOT/scripts/run_validation.sh xdopit03_cl dataset_folder
$TEAM_CODE_ROOT/scripts/run_validation.sh xdopit03_lf dataset_folder
$TEAM_CODE_ROOT/scripts/run_validation.sh xdopit03_ju dataset_folder
```

Spouštění autonomního agenta

Před spuštěním agenta musí běžet CARLA simulátor. Jeho spuštění bylo popsáno výše B.2. Jakmile je simulátor připraven je možné využít skript ke spuštění agenta:

`$TEAM_CODE_ROOT/scripts/run_agent.sh`

Agent se spouští se čtyřmi argumenty. První z nich je soubor ze složky:

`$TEAM_CODE_ROOT/drive_scenarios`

V něm jsou definované scénáře, které bude vytvořený agent řešit. Druhý je také soubor. Tentokrát ze složky:

`$TEAM_CODE_ROOT/drive_routes`

Obsahem jsou trasy měst, na kterých bude vyhodnocován automobil. Třetím argumentem je konfigurační soubor pro autonomního agenta. Obsah souboru je pro týmy volitelný. Soubor je připravený a je dostupný ve:

`$TEAM_CODE_ROOT/autonomous_agents/config_file.txt`

Soubor obsahuje tři řádky. Na každém řádku je definice cesty k jednomu ze souborů, v němž jsou natrénované hodnoty modelu neuronové sítě. Obsah předpřipraveného souboru je popsán níže.

```
$TEAM_CODE_ROOT/trained_models/xdopit03_trained_model_cl.pt
$TEAM_CODE_ROOT/trained_models/xdopit03_trained_model_ju.pt
$TEAM_CODE_ROOT/trained_models/xdopit03_trained_model_lf.pt
```

Posledním z argumentů udává, který autonomní agent se má použít k vyhodnocování. Je možné použít variantu *human*. Ta znamená, že se spustí agent se speciálním oknem a automobil bude možné řídit pomocí klávesnice. Lidský řidič je implementovaný ve skriptech žebříčku. Kromě lidského řidiče lze použít třídu autonomního agenta, který je uložen v souboru ve složce:

`$TEAM_CODE_ROOT/autonomous_agents/`

Je tam připraven jeden autonomní agent v souboru *agent_base.py*.

```
$TEAM_CODE_ROOT/scripts/run_agent.sh all_sc.json routes.xml \
config_file.txt human

$TEAM_CODE_ROOT/scripts/run_agent.sh all_sc.json routes.xml \
config_file.txt agent_base.py
```