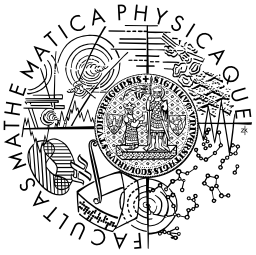


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Bc. Jiří Kučera

Strojové učení pro řízení simulovaných vozidel

Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot, Ph.D.
Studijní program: Informatika
Studijní obor: Umělá inteligence

Praha 2020

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Tímto bych rád poděkoval vedoucímu práce Jakubu Gemrotovi a konzultantovi Juraji Blahovi za veškerý čas a rady, které mi při tvorbě této práce věnovali.

Název práce: Strojové učení pro řízení simulovaných vozidel

Autor: Bc. Jiří Kučera

Katedra: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Auta ve virtuálních světech jsou typicky ovládána ručně vytvořenými pravidly. Vytváření těchto pravidel je často časově náročné a každá úprava prostředí může výsledné chování narušit. Hlavním cílem této práce je prozkoumat vhodné metody strojového učení a vytvořit jejich prostřednictvím dobře vypadající simulaci aut jezdících po městské silniční síti. Výsledným modelem je neuronová síť přímo ovládající plyn, brzdu a volant auta. Síť je schopná sledovat cestu a vyhýbat se srážkám s ostatními agenty na křižovatkách bez semaforů. Pro trénování jsme použili algoritmus Proximal policy optimization a trénování jsme vylepšili technikami curriculum learning, GAIL, curiosity a behavioral cloning. V experimentech jsme ukázali, že ačkoli výsledné chování není zcela perfektní, je dostatečně dobré pro potencionální použití v simulaci.

Klíčová slova: Umělá Inteligence Strojové Učení Navigace Simulace

Title: Machine Learning for Driving of Virtual Vehicles

Author: Bc. Jiří Kučera

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: Cars in virtual worlds are typically controlled by handcrafted rules. Creating such rules is often time-consuming and has to be repeated every time environment is altered. The goal of this thesis is to explore suitable machine learning techniques and create a good looking simulation of cars driving on an urban road network. The resulting model is a feedforward network directly controlling throttle, steering, and brake. The network is capable of following the assigned road and avoid collisions with other agents on crossroads without traffic lights. The model was trained using the Proximal policy optimization algorithm enhanced by GAIL, curiosity, behavioral cloning, and curriculum learning. In this paper, we have also shown that the resulting behavior, while not completely perfect, is good enough for use in a simulation.

Keywords: Artificial Intelligence Machine Learning Navigation Simulation

Obsah

Úvod **3**

1 Umělé neuronové sítě **5**
1.1 Umělá neuronová síť 5
1.2 Učení s učitelem 8
1.3 Zpětnovazební učení 8
1.3.1 Formální definice 9
1.3.2 Algoritmy zpětnovazebního učení 11
1.3.3 Proximal policy optimization 11
1.3.4 Rozšiřující přístupy 13
1.4 Neuroevoluce 15

2 Unity a ml-agents **16**
2.1 Unity 16
2.2 Ml-agents 16

3 Související práce **17**

4 Řešení **18**
4.1 Mapa a cesty 18
4.2 Použité neuronové sítě 19
4.2.1 Vstupy 20
4.2.2 Výstupy 24
4.3 Funkce odměny 24
4.4 Trénování 27
4.5 Výsledná simulace 28

5 Experimenty a výsledky **29**
5.1 Způsoby evaluace 29
5.2 Výsledné sítě 30
5.3 Hyperparametry sítě 32
5.4 Efekt curriculum learning 33
5.5 Efekt umocnění signálu zatáčení 33
5.6 Neuroevoluce 35

Závěr **36**

Seznam použité literatury **37**

Seznam obrázků **40**

A Uživatelská dokumentace **41**
A.1 Instalace potřebných programů 41
A.2 Popis prostředí 41
A.2.1 Parametry prostředí 41
A.2.2 Zobrazení statistik 43

A.3 Simulace 43
A.4 Trénování 43

B Programátorská dokumentace **46**

C Popis elektronických příloh **47**

Úvod

V této kapitole si představíme motivaci této práce, vymezíme problém, který budeme řešit, a stručně shrneme obsah této práce.

Motivace práce

Auta ovládané umělou inteligencí jsou běžnou součástí velké řady simulací a počítačových her. Vytváření algoritmů, která je ovládají, ale může být časově náročné, kvalitní logika musí být schopná zohledňovat velké množství informací z okolí a být dostatečně robustní, aby se chovala rozumně i v nestandardních situacích. Dodatečné úpravy prostředí pak můžou vytvořit nové problémy, které je třeba řešit ručními úpravami algoritmu. Bylo by tedy užitečné vytvořit metodu, která je schopna si nějakým způsobem sama vyvinout vhodnou logiku, např. prostřednictvím prozkoumávání prostředí.

Vymezení problému

Primárním cílem této práce je vytvořit za pomoci metod strojového učení simulaci aut jezdících po silniční síti tvořenou ze silnic a křižovatek bez semaforů. Auta by měla být schopna sledovat externě vygenerovanou cestu a vyhýbat se kolizím s ostatními auty. Výsledná simulace by měla vypadat přirozeně a neměla by rušit pozorovatele častými chybami.

Druhým naším cílem je vylepšit simulaci přidáním parkovacích míst. Agenti by měli začínat na parkovacím místě, bezpečně najet na silnici a na konci cesty opět zaparkovat na přiřazené parkovací místo. Vzhledem k tomu, že se obě chování v jistých ohledech značně liší, budeme parkování realizovat druhým modelem. Cílem modelu určeného ke sledování cesty pak bude dostat se na pozici kousek před parkovacím místem, kde řízení převezme druhý model.

Výsledné řešení se pokusíme podložit experimenty potvrzujícími, proč jsme toto řešení zvolili.

Jako rámec práce jsme si zvolili určitá omezení. Silnice mají v každém směru pouze jeden pruh. Všechny silnice a křižovatky jsou dostatečně uzpůsobeny k tomu, aby jimi bylo možné projet ve všech směrech bez popojíždění tam a zpět. Agent může začínat a končit pouze na parkovacím místě sousedící s jeho pruhem, ne na místech nacházejících se přes silnici. Z časových důvodů jsme se také nakonec věnovali pouze příčnému parkování. Na mapě se kromě aut stojících na některých parkovacích místech nevyskytují žádné pohyblivé či nepohyblivé překážky.

Přehled práce

V první části si představíme teoretické pozadí práce, vysvětlíme si pojem neuronových sítí a představíme si různé přístupy používané k jejich trénování. V dalších částech práce si představíme používané technologie a následně stručně probereme existující přístupy určené k trénování problémů podobných našemu zadání. Ve čtvrté kapitole si důkladně představíme námi použité sítě a způsoby

trénování. Na závěr práce analyzujeme vlastnosti výsledných chování a podpoříme některá učiněná rozhodnutí konkrétními daty z experimentů.

1. Umělé neuronové sítě

V této kapitole si představíme základní teorii o neuronových sítích a s nimi spojenými metodami strojového učení.

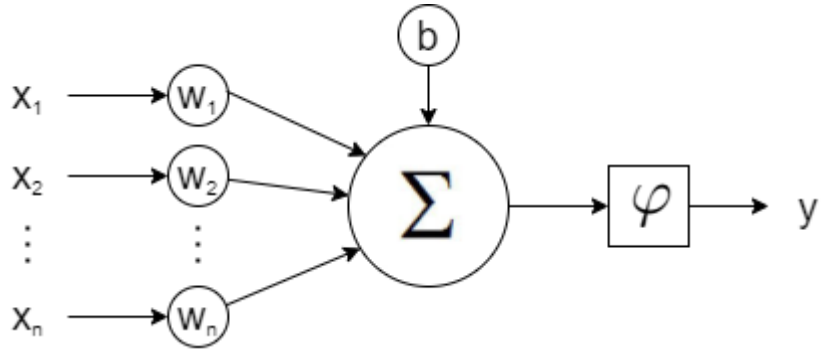
1.1 Umělá neuronová síť

Umělé neuronové sítě jsou dnes pravděpodobně nejpopulárnějším modelem strojového učení. Jsou inspirovány nervovými soustavami reálných živočichů. V poslední letech zaznamenal výzkum v oblasti umělých neuronových sítí značných úspěchů například v oblastech strojového překladu (Wu a kol., 2016), syntézi řeči (Oord a kol., 2016), či strojového vidění (He a kol., 2016).

Základním stavebním prvkem neuronových sítí jsou neurony. Každý neuron reprezentuje matematickou funkci ve tvaru:

$$y = \varphi(\sum_{i=1}^{n} w_i x_i + b),$$

která mapuje vstupní vektor $\vec{x} = (x_1, x_2, ..., x_n)$ na výstupní hodnotu y za použití váhového vektoru $\vec{w} = (w_1, w_2, ..., w_n)$, hodnoty biasu b (reprezentujícího lineární posun) a aktivační funkce φ. Hodnotu $\xi = \sum_{i=1}^{n} w_i x_i + b$ nazýváme *potenciál* neuronu. Nákres celého neuronu si můžeme prohlédnout na obrázku 1.1.



Obrázek 1.1: Nákres umělého neuronu.

Bias nemusíme nutně brát jako separátní hodnotu, pro jednodušší zápis můžeme neuronu přidat jeden imaginární vstup s konstantní hodnotou 1 a bias můžeme brát jako $(n+1)$-ní hodnotu váhového vektoru $\vec{w}$, potenciál pak můžeme jednodušeji zapisovat jako $\xi = \sum_{i=1}^{n} w_i x_i$.

Nejjednodušším typem umělé neuronové sítě je perceptron (Rosenblatt, 1958). Perceptron funguje jako lineární separátor v n-dimenzionálním prostoru, tedy reprezentuje $(n-1)$-dimenzionální nadrovinu. Pro každý vstupní vektor $\vec{x}$ vrací hodnotu 0 nebo 1 podle toho, na které straně $(n-1)$-dimenzionální nadroviny se x nachází. Jeho aktivační funkce zpravidla vypadá takto:

$$\varphi(\xi) = \begin{cases} 1 & \text{pokud } \xi > 0 \\ 0 & \text{jinak} \end{cases}$$

Perceptron umí rozlišit pouze množiny, které jsou lineárně separabilní. Složitější sítí, která umí separovat i obecnější množiny je tzv. vícevrstvý perceptron (*Multilayer perceptron* - MLP). MLP se skládá ze vstupních neuronů, reprezentujících vstupní vektor, jedním či více výstupními neurony sloužícími ke klasifikaci vstupního vektoru a z jedné či více skrytých vrstev. Každý neuron (kromě vstupních neuronů) je spojen synapsemi se všemi neurony v předchozí vrstvě. K-tou vrstvu tedy můžeme reprezentovat *váhovou maticí* W^k vhodných rozměrů a výstupy vrstev pak můžeme počítat pomocí vzorce

$$\vec{y_k} = \varphi(W^k \vec{y}_{k-1}),$$

kde $\vec{y_k}$ značí výstup k-té vrstvy.

Neurony v daných skrytých vrstvách zpravidla používají stejnou aktivační funkci. Jednou z běžných funkcí je např. *sigmoid* funkce:

$$\varphi(\xi) = (1 + e^{-\xi})^{-1}$$

mapující potenciál ξ na interval $(-1, 1)$. Dalšími běžnými funkcemi jsou např. hyperbolický tangens *tanh*, či rectifier linear unit (ReLU)

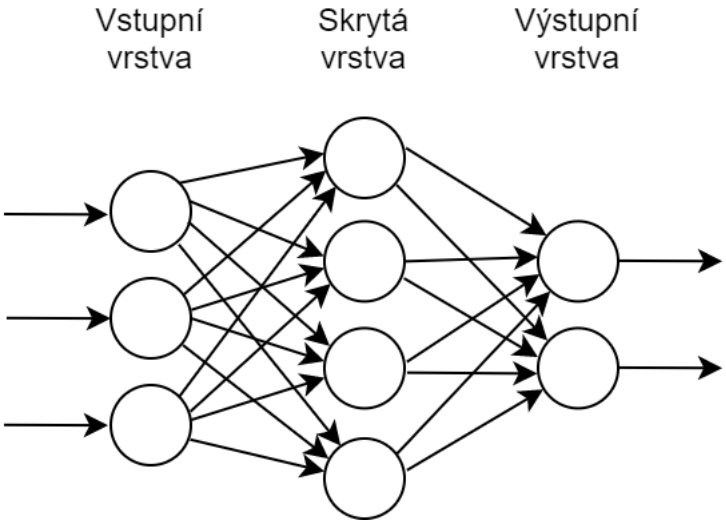
$$\varphi(\xi) = max(0, \xi),$$

které se chovají lépe v hlubších sítích s více skrytými vrstvami. Zajímavá je i tzv. *swish* (Ramachandran a kol., 2017) funkce:

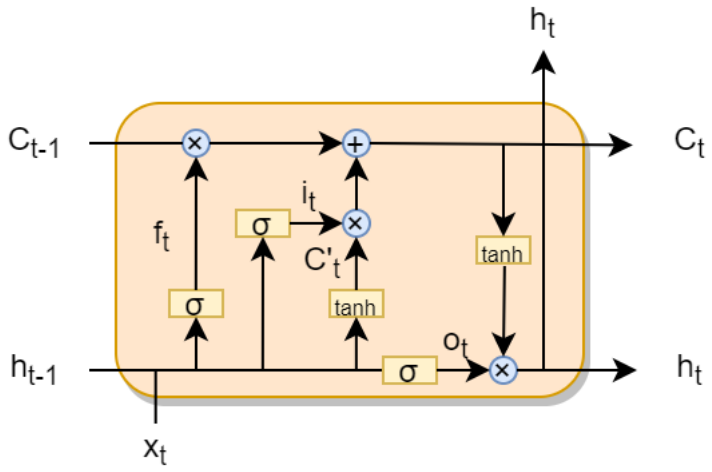
$$\varphi(\xi) = \xi * (1 + e^{-\xi})^{-1},$$

která se chová velmi podobně jako ReLU, ale oproti ní má spojité derivace.

Obecnější verzí vícevrstvého perceptronu je *dopředná neuronová síť* (Feedforwad neural network - FNN), které převádí vstupní vektor $\vec{x}$ na výstupní vektor $\vec{y}$. Nákres této sítě si můžeme prohlédnout na obrázku 1.2.



Obrázek 1.2: Nákres dopředné neuronové sítě.



Obrázek 1.3: Nákres LSTM buňky

Hluboké neuronové sítě

Standardní FNN sítě podporují jen dopředné hrany a tvoří tak acyklický graf. *Rekurentní neuronové sítě* (*Recurrent neural networks - RNN*) toto paradigma porušují povolením zpětných hran. Síť díky nim získává možnost předávat si informace mezi jednotlivými vyhodnoceními sítě a vzniká tak jakási paměť v podobně vnitřního stavu. To je užitečné zejména v oblastech porozumění psanému textu či mluvenému projevu.

Nejběžnější verzí RNN je LSTM buňka (Hochreiter a Schmidhuber, 1997) poprvé popsaná v roce 1997. Nákres buňky si můžeme prohlédnout v obrázku 1.3. LSTM si interně uchovává uchovává 2 stavy. V každém kroku dostane na vstupu předchozí stavy c_{t-1} a h_{t-1} a vstup x_t. Data jsou dále zpracována pomocí sigmoid a tanh přechodových funkcí a třech bran, které regulují proud informací v rámci buňky.

Formálně můžeme LSTM buňku zapsat jako následující sadu operací:

$$f_t = \sigma(x_t U^f + h_{t-1} W^f),$$

$$i_t = \sigma(x_t U^i + h_{t-1} W^i),$$

$$o_t = \sigma(x_t U^o + h_{t-1} W^o),$$

$$C'_t = \tanh(x_t U^g + h_{t-1} W^g),$$

$$C_t = \sigma(f_t * C_{t-1} + i_t * C'_t),$$

$$h_t = \tanh(C_t) * o_t,$$

kde $U^f, U^i, U^o, U^g, W^f, W^i, W^o$ a W^g jsou matice vhodných velikostí.

RNN sítě se běžně trénují prostřednictvím *backpropagation trough time* algoritmu. Aby se síť zbavila zpětných hran, rozbalí se přesně tak, jak v ní proudily data, a stane se víceméně běžnou dopřednou sítí s jediným rozdílem: v nově vytvořené hluboké síti některé hrany sdílejí váhy. Novou síť je ale možné již trénovat standardním backpropagation algoritmem.

Dalším možným typem hlubokých sítí jsou takzvané *konvoluční neuronové sítě* (*convolutional neural networks - CNN*) (Fukushima a Miyake, 1982). Využívají

se hlavně v oblasti strojového vidění a zpracování obrazu a v posledních letech bylo za jejich pomoci vytvořena řada modelů jako například AlexNet (Krizhevsky a kol., 2012) nebo ResNet (He a kol., 2016) sloužící ke strojovému rozpoznávání obrazu.

1.2 Učení s učitelem

Trénování FNN řadíme do kategorie *učení s učitelem*. Trénovací data pro metody z této kategorie jsou množiny dvojic vektorů $\{(\vec{x}^{(1)}, \vec{y}^{(1)}), (\vec{x}^{(2)}, \vec{y}^{(2)}), ..., (\vec{x}^{(N)}, \vec{y}^{(N)})\}$, jejich pomocí chceme vytvořit model, který ze vstupních vektorů $\vec{x}$ co nejpřesněji produkuje odpovídající odhady cílových vektorů $\vec{y}$. Proces učení tedy spočívá v hledání takových parametrů θ daného modelu (v případě FNN jde o váhové vektory $\vec{w}$ a hodnoty biasů jednotlivých neuronů), které minimalizují vhodnou chybovou funkci.

Běžně používanou chybovou funkcí je tzv. *Střední kvadratická chyba* (Mean square error - MSE) definovaná jako:

$$E = \frac{1}{N} \sum_{i=1}^{N} \|\vec{y}^{(i)} - \vec{o}^{(i)}\|^2,$$

kde $\vec{o}^{(i)}$ reprezentuje výstup sítě pro vstupní vektor $\vec{x}^{(i)}$.

Pro trénování FNN se běžně používá tzv. *backpropagation* algoritmus (Rumelhart a kol., 1986). Tento algoritmus nejprve iniciuje síť náhodnými parametry z vhodného náhodného rozdělení a poté opakovaně vezme náhodnou dvojici ($\vec{x}$,$\vec{y}$), spočítá výstup sítě $\vec{o}$ při vstupu $\vec{x}$ a zjistí hodnotu chybové funkce E. Následně spočítá parciální derivace $\Delta_E w_{ij}$ chyby E pro každou váhu w_{ij} v síti. Pro váhu synapse w_{ij} mezi neuronem skryté vrstvy a výstupním neuronem můžeme derivaci spočítat pomocí vzorce

$$\Delta_E w_{ij} = (y_j - o_j)\varphi'(\xi_j)y_i = \delta_j y_i$$

a pro synapsi vedoucí do neuronu skryté vrstvy

$$\Delta_E w_{ij} = (\sum_k \delta_k w_{jk})\varphi'(\xi_j)y_i = \delta_j y_i.$$

Ze vzorců je vidět, že hodnoty parciálních derivací vedoucích do neuronu j jsou přímo závislé na hodnotách parciálních derivací synapsí vedoucích z neuronu j do neuronů následující vrstvy, v praxi se proto parciální derivace počítají odzadu a chyby se propagují zpět do předních vrstev (odtud název *backpropagation* algoritmus).

Pseudokód algoritmu si můžeme prohlédnout v tabulce Algoritmus 1.

1.3 Zpětnovazební učení

Zpětnovazební učení (*Reinforcement learning* - RL) je dalším přístupem pro trénování modelů strojového učení. Model typicky reprezentuje mozek agenta, který se pohybuje v neznámém světě. Svět agent vnímá pomocí množiny senzorů, které tvoří vstupní vektor $\vec{x}$, výstup modelu pak nějakým způsobem kóduje, jakou

Algorithm 1 Backpropagation algoritmus

```
1: iniciuj váhy sítě náhodnými hodnotami
2: for epoch ← 1...max_epoch do
3:     promíchej vstupní data
4:     for all (x⃗, y⃗) ← data do
5:         o⃗ ← výstup sítě pro x⃗
6:         spočítej hodnotu chybové funkce E
7:         for každou váhu w_ij do
8:             spočítej Δ_E w_ij
9:             w_ij ← w_ij + αΔ_E w_ij
```

akci má agent provést. Výhodou zpětnovazebního učení oproti učení s učitelem je, že nemusíme vědět, jak by se pro dané hodnoty vstupních senzorů měl agent zachovat, stačí nám, že dokážeme rozpoznat, jestli se agent zachoval správně nebo špatně.

1.3.1 Formální definice

Model světa pro zpětnovazební učení je v zásadě Markovův rozhodovací proces. Definujeme jej jako pětici $\langle S, A, P, R, \rho_0 \rangle$, kde

- S je množina možných stavů světa,
- A je množina možných akcí agenta,
- $P : S \times A \to \mathcal{P}(S)$ je přechodová funkce, kde $P(s'|a, s)$ reprezentuje pravděpodobnost, že se po provedení akce a ve stavu s dostaneme do stavu s',
- $R : S \times A \times S \to \mathbb{R}$ je funkce odměny (*reward funciton*), kde $r_t = R(s_t, a_t, s_{t+1})$ reprezentuje odměnu, kterou agent dostane po přechodu ze stavu s_t do stavu s_{t+1} použitím akce a_t,
- ρ_0 je pravděpodobnostní distribuce počátečních stavů.

Přechodová funkce může být také deterministická, pak jí definujeme jako $f : S \times A \to S$. Stavy světa mohou být plně pozorovatelné, pak agent dostává na vstupu reprezentaci celého stavu s, nebo můžou být částečně pozorovatelné, pak agent dostává množinu pozorování $o \subset s$. Funkce odměny bývá často zjednodušena tak, že závisí pouze na současném stavu $r_t = R(s_t)$ nebo na současném stavu a následující akci $r_t = R(s_t, a_t)$.

Strategie (*policy*) je funkce určující pro každý stav doporučenou akci. Strategie může být buď deterministická, pak ji značíme μ_θ a platí:

$$a_t = \mu_\theta(s_t),$$

nebo stochastická, pak jí značíme jako π_θ:

$$a_t \sim \pi_\theta(\cdot|s_t),$$

kde θ je množina parametrů modelu (v případě neuronových sítí množina synaptických vah).

Trajektorií nazýváme posloupnost stavů a akcí $\tau = (s_0, a_0, s_1, a_1, ...)$ reprezentující průchod agenta světem dle přechodové funkce P. Rádi bychom byli schopni nějak ohodnotit, jak dobře si agent v daném průchodu počínal, pro trajektorii tedy definujeme *funkci užitku* (*return function*) $R(\tau)$. V případě konečných posloupností jí definujeme jako jednoduchou aditivní funkci:

$$R(\tau) = \sum_{t=0}^{T} r_t.$$

Pro nekonečné posloupnost by se tato suma často rovnala ∞ nebo $-\infty$, zavádíme tedy *faktor slevy* $\gamma \in (0,1)$, který snižuje význam odměn v budoucnosti:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

Funkci odměny $J(\pi)$ pro policy π pak definujeme jako:

$$J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)].$$

Dále můžeme zavést ohodnocení jednotlivých stavů. Funkce odměny sama o sobě udává pouze jednorázovou odměnu, kterou dostaneme za průchod stavem v dané situaci, ale už nehodnotí, jaký potenciál máme k získání odměn v budoucnosti, pokud se v daném stavu aktuálně nacházíme. Zavádíme tedy *funkci užitku* (*On-policy value function*) $V^\pi(s)$ pro stavy:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s],$$

reprezentující očekávaný užitek, pokud začneme ve stavu s a akce vybíráme podle strategie π. Podobně můžeme definovat funkci užitku $Q^\pi(s,a)$ (*On-policy action-value function*) i pro dvojici stavu a akce:

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a],$$

reprezentující očekávaný užitek, pokud začneme ve stavu s, provedeme akci a a dále se chováme podle strategie π. Dále by bylo užitečné mít možnost ohodnotit, jaká je maximální možná hodnota užitku, pokud začínáme v daném stavu, definujeme tedy podobně *optimální funkci užitku* (*Optimal value function*) $V^*(s)$ pro stavy:

$$V^\pi(s) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s]$$

a $Q^*(s,a)$ pro páry stavů a akcí (*Optimal action-value function*):

$$Q^*(s,a) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a],$$

udávající očekávaný užitek, pokud začneme ve stavu s, provedeme akci a a dále se chováme podle optimální strategie π.

Dále definujeme *funkci vylepšení* (*advantage function*) $A^\pi(s,a)$:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s),$$

která hodnotí, jaký užitek bychom získali, kdybychom ve stavu s zvolili akci a místo zvolení náhodné akce podle rozdělení $\pi(\cdot|s)$.

1.3.2 Algoritmy zpětnovazebního učení

Existuje celá řada algoritmů k učení modelů pomocí zpětnovazebního učení. V posledních letech bylo vyvinuto množství systémů, které ve svých disciplínách předčily nejmodernější herní programy, či profesionální lidské hráče. Za zmínku stojí například model AlphaZero (Silver a kol., 2017), který zvládl porazit nejlepší současné herní systémy pro šachy, go a japonskou hru šógi. Soubor pěti sítí OpenAI Five (Berner a kol., 2019) používající populární algoritmus *Proximal policy optimization* (PPO) (Schulman a kol., 2017) dokázal porazit v roce 2019 nejlepší profesionální tým světa v počítačové hře DoTA a *Deep Q-Netowrks* (DQN) (Mnih a kol., 2013) v roce 2013 zpopularizovaly zpětnovazební učení tím, že se naučily hrát množství klasických Atari her na úrovni lidských hráčů, či lépe.

Taxonomie

Jedno z nejdůležitějších dělení algoritmů zpětnovazebního učení je na tzv. *Model-free* a *Model-based* přístupy. Model-based přístupy se snaží naučit model prostředí, tedy přechodovou funkci P a funkci odměn R. Díky vytvoření modelu prostředí dokáže agent lépe plánovat a předvídat, co se stane po provedení určitě akce, slavným zástupcem této skupiny je například AlphaZero (Silver a kol., 2017). Vytvořit dobrý model bývá ale často těžké a většinou to nebývá ani nutné, model-free přístupy, které se pouze snaží naučit policy π_θ tak bývají mnohdy populárnější variantou.

Model-free metody dále můžeme dělit na *policy optimization* a *Q-Learning* metody. Policy optimization metody se snaží přímo optimalizovat θ prostřednictvím stochastic gradient ascent algoritmu buď přímo na funkci odměny $J(\pi_\theta)$ (*Vanilla policy gradient*), nebo vlastní objective funkce, za použití funkce vylepšení $A^{\pi_\theta}(s, a)$, což dělají například algoritmy PPO (Schulman a kol., 2017) a TRPO (Schulman a kol., 2015). Aby byly algoritmy schopné odhadnout hodnotu A^{π_θ}, musí se kromě výsledné strategie navíc ještě učit model $V_\phi(s)$ aproximující funkci odměny $V^\pi(s)$. Tento model pak nazýváme *critic* a pro síť reprezentující strategii používáme označení *actor*.

Oproti tomu Q-learninig metody se snaží aproximovat optimální funkci užitku $Q^*(s, a)$ pomocí $Q^\theta(s, a)$ typicky za použití Bellmanových rovnic. Výslednou policy pak získávají přímo z hodnoty Q funkce:

$$a(s) = \arg\max_a Q_\theta(s, a).$$

Známým zástupcem této skupiny je například DQN.

Policy optimalization a Q-learning je možné i kombinovat, což dělají například přístupy *Soft actor critic* (SAC) (Haarnoja a kol., 2018), či *Deep Deterministic Policy Gradient* (DDPG) (Lillicrap a kol., 2015).

1.3.3 Proximal policy optimization

V této sekci se blíže zaměříme na přístup *Proximal polici optimization*, který v této práci používáme. PPO byl poprvé představen v roce 2017 a překonal tehdejší nejmodernější policy optimalization metody jako podobně fungující *Trust Region Policy Optimization* (TRPO) (Schulman a kol., 2015) či A2C (Mnih a kol., 2016).

Problémem standardního vanilla policy gradient algoritmu je jeho nestabilita: v praxi se i zdánlivě malá změna v parametrech modelu θ může ve výsledku značně projevit na výsledném chování policy π_θ, což může narušovat proces učení. PPO proto zavádí vlastní objective funkci $L^{\text{CLIP}}(s, a, \theta_k, \theta)$, kterou optimalizuje namísto $J(\pi_\theta)$:

$$L^{\text{CLIP}}(s_t, a_t, \theta_k, \theta) = \min\Big(r_t(\theta)A^{\pi_{\theta_k}}(s_t, a_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A^{\pi_{\theta_k}}(s_t, a_t)\Big),$$

kde: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$, a ϵ je vhodně zvolený parametr (typicky 0.2).

Vzorec $r_t(\theta)A^{\pi_{\theta_k}}(s_t, a_t)$ shodně používá i TRPO a reprezentuje základní hodnotu objective funkce. Druhý term, $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A^{\pi_{\theta_k}}(s_t, a_t)$, limituje, jak moc se může objective funkce zlepšit změnou policy. Tím odebírá stimuly k příliš velkým krokům, které by mohly poškodit výsledné chování agenta.

Algoritmus funguje tak, že opakovaně nejprve získá trajektorie simulací agenta ve světě podle aktuální π_{θ_k} a následně upraví parametry θ maximalizací funkce L^{CLIP} stochastickým gradient algoritmem a poté aktualizuje parametry ϕ funkce odměny. Pseudokód algoritmu si můžeme prohlédnout v tabulce 2.

Existuje ještě druhá verze algoritmu PPO, která místo *clip* funkce přidává postih za rozdíly mezi π_{θ_k} a π_θ pomocí KL-divergence:

$$L^{\text{KLPEN}}(s_t, a_t, \theta_k, \theta) = r_t(\theta)A^{\pi_{\theta_k}}(s_t, a_t) - \beta\text{KL}[\pi_{\theta_k}(\cdot|s_t), \pi_\theta(\cdot|s_t)],$$

kde β je parametr, který se mění v průběhu běhu algoritmu. V praxi se ale více používá PPO-Clip verze.

Algorithm 2 PPO-Clip

1: iniciuj parametry ϕ a θ náhodnými hodnotami
2: **for** $k \leftarrow 1, 2, ...$ **do**
3: získej množinu trajektorií $\mathcal{D}_k = \{\tau_i\}$ a odpovídajících odměn R_t podle strategie π_{θ_k}
4: spočítej odhady A_t funkce vylepšení na základě aktuální funkce užitku V_{ϕ_k}
5: aktualizuj parametry policy:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T}\sum_{\tau\in\mathcal{D}_k}\sum_{t=0}^{T}\min\Big(r_t(\theta)A^{\pi_{\theta_k}}(s_t, a_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A^{\pi_{\theta_k}}(s_t, a_t)\Big),$$

stochastickým gradient ascent algoritmem.
6: aktualizuj parametry funkce odměny:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T}\sum_{\tau\in\mathcal{D}_k}\sum_{t=0}^{T}\Big(V_\phi(s_t) - R_t\Big),$$

gradient ascent algoritmem.

1.3.4 Rozšiřující přístupy

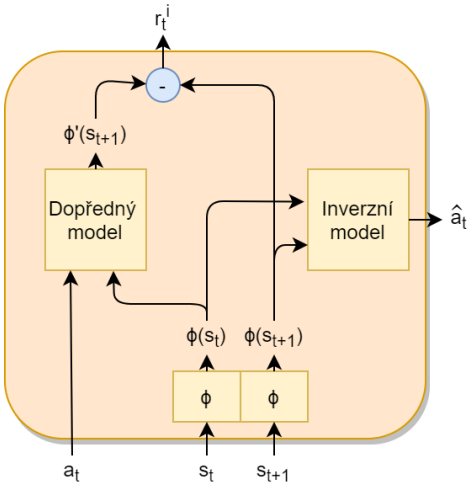
Curriculum learning

Existuje celá řada přístupů, jak zlepšit efektivitu výše zmíněných algoritmů na reálných problémech. Jedním z nich je například *Curriculum learning* (Bengio a kol., 2009). Filosofie Curriculum learning je, že model nejprve trénujeme na lehčích úlohách a až když je agent rozumně zvládá, přecházíme ke složitějším problémům.

V experimentech se ukázalo, že curriculum learning vede k rychlejšímu trénování, protože model ze začátku neztrácí čas zbytečně těžkými problémy. Současně v určitých případech může vést k lepší výsledné policy, protože se v prostoru parametrů θ dostaneme do oblastí s lepšími lokálními maximy.

Curiosity

Dalším možným přístupem je přidání Curiosity modulu (Pathak a kol., 2017) do sítě. Ten se snaží agenta motivovat k objevování nových zkušeností zavedením vnitřních odměn r_t^i. Agent se pak snaží optimalizovat součet $r_t = r_t^i + r_t^e$, kde r_t^e jsou standardní odměny z prostředí.



Obrázek 1.4: Nákres curiosity modulu.

Nákres curiosity modulu si můžeme prohlédnout na obrázku 1.4. Modul dostává na vstupu trojici s_t, a_t, s_{t+1}, stavy si nejprve převádí do vlastní reprezentace $\phi(s)$. K natrénování této reprezentace slouží tzv. *inverzní model*, který na vstupu dostává $\phi(s_t)$ a $\phi(s_{t+1})$ a snaží se předpovědět, jak vypadala akce a_t. Proces učení pak spočívá v minimalizování

$$L_I(\hat{a}_t, a_t),$$

kde $\hat{a}_t$ je odhad vygenerovaný inverzním modelem a L_I je vhodná loss funkce hodnotící chybu odhadu. K samotnému generování odměn r_t^i slouží tzv. *dopředný model*, který na vstupu dostává a_t a s_t a snaží se odhadnout reprezentaci $\phi(s_{t+1})$ následujícího stavu s_{t+1}. Trénování tohoto modelu pak spočívá v minimalizování

$$L_F(\phi(s_{t+1}), \phi'(s_{t+1})) = \|\phi(s_{t+1}) - \phi'(s_{t+1})\|_2^2,$$

kde $\phi'(s_{t+1})$ je odhad dopředného modelu. Výsledné odměny se počítají pomocí:

$$r_t^i = \frac{\eta}{2}\|\phi(s_{t+1}) - \phi'(s_{t+1})\|_2^2,$$

kde η je vhodná konstanta.

Curiosity se hodí zejména v prostředích se vzácnými odměnami, nicméně experimenty prokázaly, že má značný efekt, i když jsou odměny časté.

Apprenticeship learning

Apprenticeship learning je proces, kdy se policy tvoří pozorováním expertní policy π_E. Expertní know-how je typicky předáváno prostřednictvím vzorových trajektorií $T_E = \{\tau_1, \tau_2, ..., \tau_m\}$ a agent se z nich snaží získat znalosti. Expertem může být například i člověk ručně řídící agenta, můžeme tedy zrychlit učení sítě tím, že osobně agentovi předvedeme, jak by se měl v daných situacích chovat.

Jedním možným přístupem je například *Behavioural cloning* (Bain a Sammut, 1995), které se přímo snaží naučit expertní policy π_E ze vzorových trajektorií prostřednictvím učení s učitelem. Problémem tohoto přístupu bývá, že expert typicky prochází jen malou část stavového prostoru, a výsledný agent se pak lehce může dostat do situace, kde neví, co dělat.

Direct policy learning toto řeší tím, že střídá proces učení a simulace agenta. Ve chvíli, když se agent dostane do neznámé situace, požádá expertní policy o přidání nových vzorových trajektorií obsahujících danou situaci. Nevýhodou tohoto přístupu je, že je potřeba expert, který je stále online a dokáže kdykoli vygenerovat nové trajektorie, nefunguje tedy dobře např. se vstupem od uživatele.

Inverse reinforcement learning (Abbeel a Ng, 2004) se liší tím, že nevyžaduje žádné odměny z prostředí. Dostává na vstupu pouze expertní trajektorie a z nich si snaží vytvořit vlastní funkci odměny $r_\psi(s,a)$, pomocí které se pak standardním způsobem snaží naučit cílovou policy π_θ. Iterativně pak střídá učení r_ψ a π_θ, dokud se π_θ nechová dostatečně podobně jako π_E ve vzorových trajektoriích.

Generative adverasarial imitation learning

Generative adversarial imitation learning (Ho a Ermon, 2016) (GAIL) je dalším zástupcem ze skupiny apprenticeship learning přístupů. Přidává síť reprezentující diskriminátor $D_R : S \times A \to (0,1)$. Tato síť se učí odhadovat, zda daná dvojice stavu a akce byla generována expertní policy π_E, či aktuální policy π_{θ_k}. Pro trénování této sítě lze použít standardní učení s učitelem na datech ze vstupních trajektorií.

Podobně jako curiosity, GAIL následně přidává další sérií odměn

$$r_t^d = -\eta \log(D_R(s_t, a_t)),$$

kde η je vhodná konstanta. Tento výraz je maximalizován, když si diskriminátor myslí, že (s_t, a_t) pochází od π_E, tedy když $D_R(s_t, a_t) = 0$. Agent je tedy motivován vykonávat podobné akce jako expertní policy π_E. V ideálním případě se dostane do stavu, kdy diskriminátor D_R nebude schopný rozpoznat, které policy patří daný vstup a bude vydávat pokaždé hodnotu kolem 0.5.

1.4 Neuroevoluce

Další možnou metodou k trénování neuronových sítí je použití *genetických algoritmů* (Mitchell, 1998). Genetický algoritmus pracuje s *populací* - skupinou jedinců z nichž každý je typicky reprezentovaný vektorem čísel - tzv. *genomem*. Každý genom reprezentuje jedno řešení daného problému, což jsou v případě neuroevoluce buď přímo parametry sítě θ, nebo v některých případech (Stanley a Miikkulainen, 2002) genom kóduje i samotnou strukturu sítě.

Genetický algoritmus pracuje po generacích. Nejprve na začátku iniciuje první generaci náhodnými jedinci, a následně opakovaně tvoří novou generaci z předchozí pomocí několika operátorů. Typicky používá operátory *selekce*, která vybírá kvalitní jedince k reprodukci, *křížení*, které kombinuje genomy vybraných jedinců a tvoří z nich nové jedince, a *mutace*, které provádí drobné změny v nově vytvořených jedincích. Operátor selekce potřebuje být schopen nějak ohodnotit kvalitu jedinců v populaci, k tomu slouží tzv. *fitness funkce*, která každému jedinci přiřazuje reálnou hodnotu hodnotící, jak kvalitně jedinec dokáže plnit daný úkol.

Výhodou neuroevoluce oproti zpětnovazebnímu učení je, že nemusíme být schopní hodnotit konkrétní chyby agentova chování, stačí nám umět posoudit, jak dobře se agent chová jako celek. V posledních letech se ukázalo, že neuroevoluce může být konkurenceschopná alternativa zpětnovazebního učení (Salimans a kol., 2017) (Such a kol., 2017).

2. Unity a ml-agents

V této kapitole si představíme konkrétní technologie používané v této práci.

2.1 Unity

Unity (Helgason a kol., 2007) je game engine vyvíjený firmou Unity Technologies a poprvé představený v roce 2005. Od té doby se propracoval na jeden z nejpopulárnějších nástrojů pro tvorbu her, či aplikací. Velká část jeho úspěchu pramení z jeho univerzálnosti: v současné době je v něm možné tvořit 2D, 3D, či VR aplikace pro více než 25 platforem. Ačkoli je populárnější spíše mezi menšími studii, byly v něm vytvořeny i některé velké hry jako Cities:Skylines, Heartstone, či Escape from Tarkov.

Primárním jazykem pro programování v Unity je C#, dříve byla podporována ještě vlastní varianta jazyka JavaScript zvaná UnityScript.

2.2 Ml-agents

ML-Agents (Juliani a kol., 2017) je oficiální projekt Unity Technologies sloužící k trénování AI agentů v prostředí Unity. Vyvíjený je od roku 2017, v době psaní této práce je nejnovější verze 0.17 z července 2020, nicméně v této práci jsme používali starší verzi 0.13.1 z ledna 2020. ML-Agents bylo naprogramováno kombinací dvou jazyků: v C# byla vytvořena část zajišťující simulaci agentů v rámci Unity a v jazyce Python probíhá samotné učení za pomoci frameworku TensorFlow (Abadi a kol., 2016).

Agenty je možné trénovat zpětnovazebním učením prostřednictvím algoritmů Proximal Policy Optimization (Schulman a kol., 2017) či Soft Actor-Critic (Haarnoja a kol., 2018). Proces učení je možné obohatit prostřednictvím Curriculum learnirng (Bengio a kol., 2009), Curiosity (Pathak a kol., 2017), Behavioral Cloning (Bain a Sammut, 1995) nebo Generative Adversarial Imitation Learning (Ho a Ermon, 2016). Agentům je možné přidat paměť v podobě LSTM buňky či je možné jim přidat vizuální vstupy z kamery.

3. Související práce

V této kapitole si představíme několik prací, které se věnovaly nějaké variantě tématu autonomního řízení simulovaných vozidel za použití strojového učení.

(Ganesh a kol., 2016) trénovali agenta jezdícího po trati v TORCS (Wymann a kol., 2000) simulátoru pomocí Q-learning metody. Výsledný agent byl schopný spolehlivě projíždět množství různých tratí.

(Yu a kol., 2016) natrénovali DQN síť pro ovládání jednoduchého závodního auta pouze za pomoci vizuálních vstupů. Síť produkovala diskrétní výstupy simulující vstup od uživatele.

(Zhu a kol., 2018) používali senzorické vstupy z reálného auta a za pomoci DDPG algoritmu natrénovali několik variant jednoduché sítě určené k bezpečnému autonomnímu sledování auta. Výsledné sítě vydávaly pouze cílovou hodnotu akcelerace.

(Chen a kol., 2020) vytvořili síť pro ovládání simulovaného vozidla v městském prostředí. Síť jako vstupy používala data z čelní kamery a 360° lidaru a byla trénována SAC algoritmem. Výsledný model navíc generoval schématický nákres aktuální situace okolo agenta z ptačího pohledu, což může být užitečné např. k demonstraci, jak agent chápe aktuální stav světa.

(Kaushik a kol., 2018) se zaměřili na problém předjíždění aut v TORCS simulaci. Síť dostávala množinu ad-hoc vstupů jako pozice agenta na silnici nebo vzdálenost okolních aut a byla trénována za použití DDPG algoritmu. Práce demonstrovala význam curriculum learning pro daný problém, postupné učení agenta na problémem přiměřených aktuálním schopnostem agenta výrazně zvýšilo úspěšnost výsledného modelu.

(Shalev-Shwartz a kol., 2016) se věnovali problému bezpečného přejíždění mezi pruhy na větší silnici. Rozdělili policy na dvě části, první část ze stavu světa generuje krátkodobý cíl agenta typu jako např. předjetí sousedního auta, druhá část policy pak převádí agentův cíl na konkrétní akce. Agent se učí pouze první část, druhá část je pevně daná a je vytvořena tak, aby dosahovala lokálních agentových cílů bezpečným způsobem.

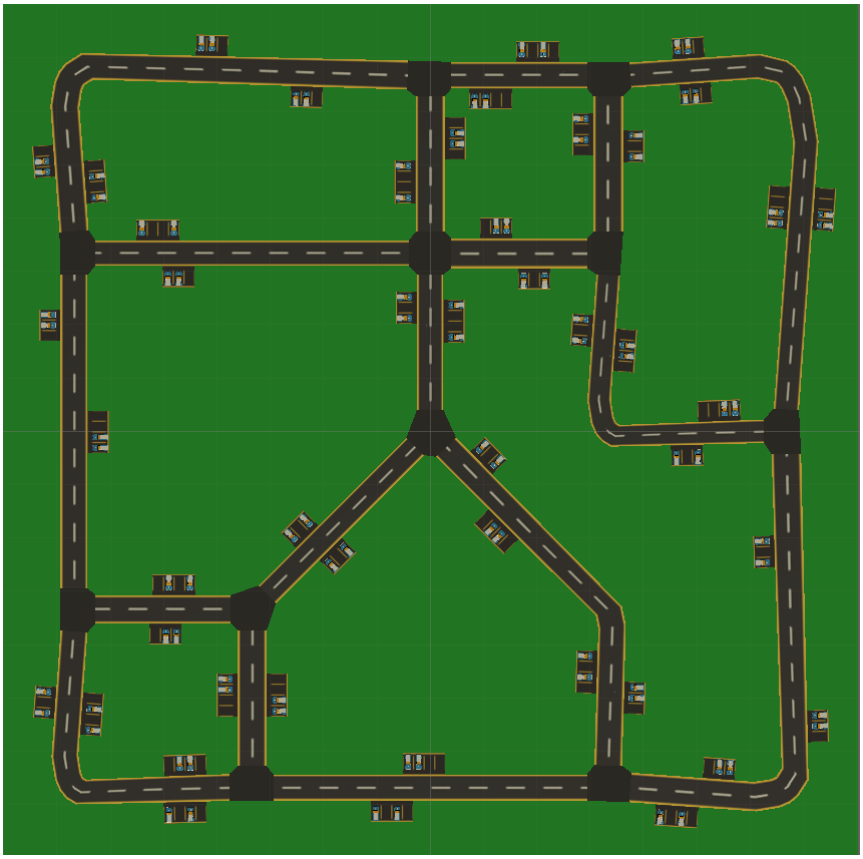
(AbuZekry a kol., 2019) se zaměřili na porovnávání několika variant genetického algoritmu s DDQN algoritmem. Porovnávání bylo provedeno na problému držení se ve středu pruhu za použití vizuálních vstupů. Síť používala konvoluční vrstvy a vydávala pouze signál natočení volantu. V experimentech se ukázalo že varianty genetického algoritmu dokážou na daném problému překonat výsledky učení DDQN algoritmem.

4. Řešení

V této kapitole si představíme neuronové sítě, které jsme použili k řešení cílů představených v úvodu práce. Zároveň si popíšeme situace, v kterých jsme agenty trénovali, a odměny, kterými jsme agenty motivovali.

4.1 Mapa a cesty

Nejprve si představme prostředí, ve kterém jsme agenty trénovali. Trénovací mapu si můžeme prohlédnout na obrázku 4.1. Velikost mapy byla zvolena tak, aby na ní rozumné množství agentů zvládlo vytvořit středně hustý provoz, ve kterém se agenti běžně potkávají, ale hustý zácpy se netvoří příliš často. Prostředí neobsahuje žádné kopce, všechny jeho části mají stejnou výškovou pozici. Přidání kopců je jedním z možných rozšíření této práce.



Obrázek 4.1: Mapa používaná pro trénování

Mapa se skládá ze 3 základních stavebních prvků: silnic, křižovatek a parkovacích míst. Každá silnice má náhodnou šířku z malého rozsahu. Křižovatky mají určitou minimální velikost, aby bylo možné se na nich vytočit. Silnice napojené na křižovatku záměrně mezi sebou nesvírají ostré úhly, kdyby se u nějaké křižovatky vyskytl velmi ostrý úhel, musela by pak daná křižovatka být neesteticky velká, aby se na ní dalo ve všech situacích vytočit.

V mapě používáme pouze příčná parkovací místa. Každá parkovací oblast obsahuje 3-4 parkovací místa a v každé oblasti jsou vygenerována dvě nehybná auta sloužící jako překážky. Díky tomuto nastavení jsou parkovací místa dostatečně různorodá a agenti se musí naučit reagovat na přítomnost i absenci překážek v sousedních parkovacích místech. Cílové pozice na parkovacích místech jsou mírně posunuty dozadu, aby bylo možné z parkovacího místa vyjet bez vjíždění do protisměru.

Křižovatky a silnice jsou dány vstupním souborem v OSM XML (OSM, 2004) formátu. Program by měl být schopen načíst libovolnou mapu v tomto formátu včetně reálných map, nicméně reálné mapy často nebývají použitelné. Běžným problémem například bývá, že se několik silnic či křižovatek nachází příliš blízko u sebe. Před importováním reálných map bývá tedy často potřeba provést ruční úpravy. Do načtené mapy jsou dodatečně dogenerovány náhodná parkovací místa a šířky silnic dle aktuálního nastavení.

Cesta

Základní formou cesty je náhodná procházka po grafu silniční sítě. Cesta začíná na náhodném místě na náhodné silnici a končí silnicí v bodě těsně před křižovatkou. Je možné ji obohatit o začátek či konec na parkovacím místě příslušnému dané silnici. Počáteční místo každé cesty musí být dostatečně daleko od ostatních agentů na mapě, aby nehrozila nevyhnutelná kolize. Současně na cílové parkovací místo může v jednu chvíli mířit pouze jeden agent.

Každá cesta je interně reprezentována jako množina úseček. Na silnicích tyto úsečky vedou vždy středem pruhu, na křižovatce je cesta reprezentována jedinou úsečkou spojující konečné body odpovídajících pruhů. Všechny vygenerované křižovatky jsou konvexní, nemůže tedy nastat situace, kdy by taková úsečka vedla mimo křižovatku. Na rozdíl od silnic, kde se musí držet v rámci svého pruhu, se agent po křižovatkách může pohybovat libovolně a je tedy na něm, jakou si zvolí strategii, jak předcházet kolizím.

Popsaná reprezentace cesty slouží primárně k navigování agenta, agent není striktně vzato nucen přesně sledovat dané úsečky, může se pohybovat libovolně v rámci svého pruhu. Nicméně je za přesnou jízdu více odměňován, jak budeme popisovat níže.

4.2 Použité neuronové sítě

Ovládání agentů je zajištěno pomocí jednoduchých dopředných sítí. Obě použité sítě mají 4 skryté vrstvy a v každé z nich 256 neuronů. Pro trénování přidáváme do sítí GAIL a curiosity moduly, které na základě aktuálních stavů a akcí generují dodatečné odměny pro agenta. Tyto moduly byly podrobněji popsány v kapitole 1.3.4. Sítě vyhodnocujeme 8.33 krát za sekundu.

Jednou z možných variant vylepšení sítě bylo přidání paměti v podobně LSTM buňky. S pamětí jsme experimentovali, ale nakonec používáme síť bez ní. Máme k tomu několik důvodů. Prvním důvodem je, že podle dokumentace ML-Agents používaná LSTM buňka funguje dobře pouze s diskrétními výstupy sítě[1]. S těmi

[1] https://github.com/Unity-Technologies/ml-agents/blob/release-0.13.1/docs/Feature-Memory.md#limitations

my nepracujeme, ale bylo by možné síť předělat tak, že bychom na výstupu místo každého neuronu používali skupinu neuronů používající one-hot kódování, kde by každý neuron reprezentoval určitou výstupní hodnotu původního neuronu. Samotná tato změna by ale mohla způsobit, že by výsledná síť mohla fungovat hůře.

Dalším důvodem je, že přidání rnn by zpomalilo trénování. Jednak kvůli tomu, že by bylo potřeba posílat větší množství dat mezi Unity a TensorFlow, ale také proto, že přidaná komplexita by způsobila, že by bylo potřeba provést více simulací pro podobný výsledek. Vzhledem k tomu, že trénování sítě už tak trvá několik dní, další prodloužení učení by výrazně zhoršilo naší možnost experimentovat.

Posledním důvodem je, že pro náš problém není paměť striktně potřeba. Pokud agentovi dodáme na vstup dostatek informací jako např. předchozí výstupy sítě či rychlost okolních překážek, je možné vytvořit dobré chování i bez paměti.

4.2.1 Vstupy

Normalizace vstupů

Veškeré vstupy jsou normalizované na (-1,1), popř. na (0,1).

Pro normalizaci vzdáleností jsme použili jednotnou konstantu dohledu zvolenou tak, aby agent viděl dostatečně daleko a měl dostatek času zareagovat na okolní překážky, ale zároveň, aby agenta nerušily objekty, které jsou od něj bezpečně daleko. Příklad dohledu agenta si můžeme prohlédnout na obrázku 4.4. Vzdálenosti jsou normalizované na (0,1) tak, že maximální vzdálenost se zobrazí na 0 a minimální 1, aby síť byla stimulovaná právě tehdy, když je blízko agenta agenta překážka.

Normalizaci rychlostí provádíme pomocí maximální hodnoty, které jsou agenti na mapě schopni dosáhnout, časové údaje normalizujeme pomocí odpovídajících časových limitů, poškození auta maximálním poškozením a úhly pomocí konstanty 180. Všechna normalizace je prováděna tak, aby hodnota 1 reprezentovala situaci, které by si agent měl všímat, tedy např. končící časový limit nebo velmi poničené auto.

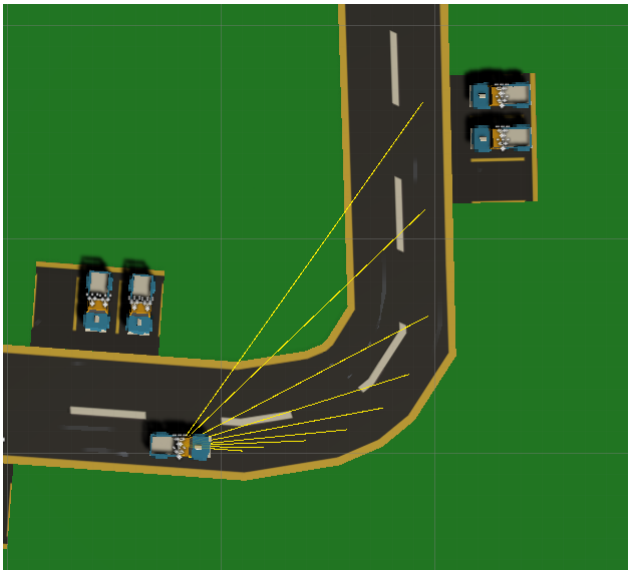
Řídící neuronová síť

Síť sloužící k jízdě po silnici má celkem 353 vstupů. Můžeme je shrnout následovně:

- souřadnice následujících bodů - 10x2 hodnot
- detekce krajů silnice - 24 hodnot
- detekce překážek - 74x4 hodnot
- aktuální rychlost agenta - 3 hodnoty
- předchozí výstupy sítě - 3 hodnoty
- procentuální postup na aktuální trati
- příznak, zda se agent nachází na silnici nebo na křižovatce

- vzdálenost do následující křižovatky
- globální časový timeout
- časový timeout z nedostatku progresu
- poškození auta
- příznak, zda se agent právě někoho dotýká

První kategorií jsou souřadnice následujících bodů v cestě. Agent si počítá souřadnice 10 bodů v pevně definovaných vzdálenostech od aktuální pozice agenta na cestě. Hodnoty jsou voleny tak, aby agent měl dostatečnou představu o tvaru cesty v dohledové vzdálenosti a mohl např. včas přibrzdit před zatáčkou. Informace jsou síti předávány v podobě dvojic hodnot *směrový úhel-vzdálenost*. Příklad si můžeme prohlédnout na obrázku 4.2.



Obrázek 4.2: Detekce pozic následujících bodů na cestě

Další kategorií jsou hodnoty odpovídající detekci vzdáleností okrajů silnice. Detekce je prováděna v pevně daných 24 směrech, ukázku si můžeme prohlédnout na obrázku 4.3.

Následuje skupina vstupů poskytujících informace o okolních překážkách. Tato skupina je zdaleka největší, celkem používáme 74 detekčních paprsků. Překážky v podobě okolních aut bývají často relativně malé, používáme tedy velké množství paprsků, aby se pokud možno nestávalo, že se v dohledové vzdálenosti agenta nachází překážka, o které agent neví. Paprsky si můžeme prohlédnout na obrázku 4.4.

Pro každý paprsek předáváme síti 4 hodnoty. První je normalizovaná vzdálenost překážky. Další 2 hodnoty reprezentují relativní rychlost detekovaného objektu vzhledem k agentovi. Každý paprsek používá svůj vlastní souřadný systém, kde jedna souřadnice odpovídá směru paprsku a druhá jeho horizontální kolmici. Díky této konverzi by agent měl být schopen lépe odhadnout míru nebezpečí kolize s překážkou.

Obrázek 4.3: Detekce vzdáleností okrajů silnice

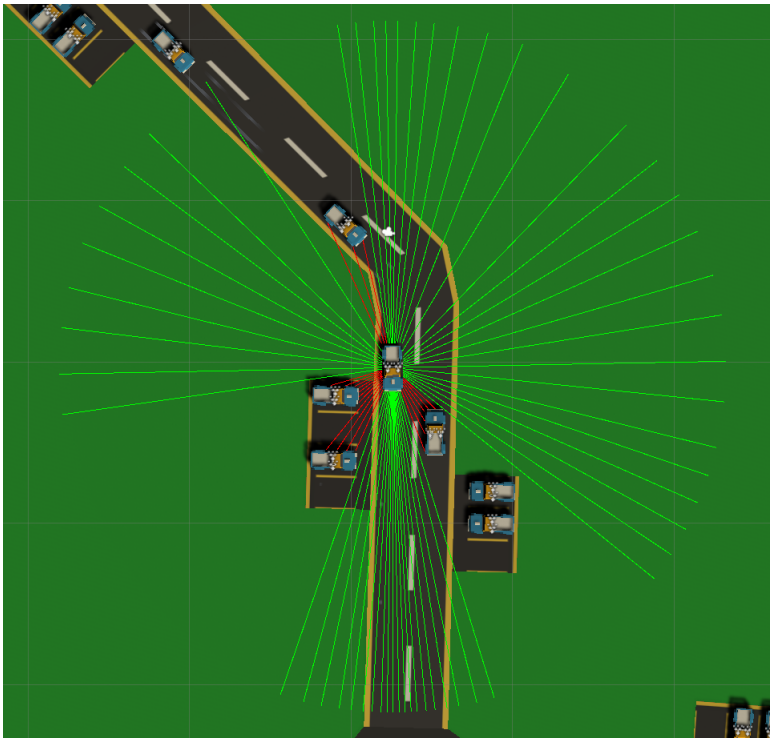
Čtvrtou hodnotou je úhel svírající dopředné vektory agenta a detekované překážky. Tato informace je potřeba hlavně ve chvíli, když se před agentem nachází stojící auto. Bez této informace by agent neměl možnost rozpoznat, zda se auto chystá jet proti němu a zda se mu tedy má snažit vyhnout, nebo jestli pouze nestojí v zácpě.

Pokud paprsek nedetekuje žádnou překážku, všechny 4 odpovídající hodnoty jsou rovny nule.

Zbylé vstupy odpovídají základním informacím o agentovi a aktuálním experimentu. První 3 hodnoty reprezentují aktuální rychlosti agenta ve všech třech směrech převedená do lokálních souřadnic. Další 3 vstupy odpovídají výstupům po posledním vyhodnocení sítě a fungují jako částečná náhrada paměti, aby měl agent informaci, co naposledy dělal.

Následuje hodnota reprezentující, jaké procento trati už agent ujel. Tento vstup je přidán primárně kvůli algoritmu PPO, aby si mohl vytvořit dobré ohodnocení stavů.

Další vstup reprezentuje příznak, zda je agent nachází na křižovatce nebo na silnici. To se agentovi může hodit, aby věděl, kdy si dávat větší pozor a kdy protijedoucí auta nepředstavují téměř žádné nebezpečí. Následující vstup odpovídá vzdálenosti do následující křižovatky, další dvě hodnoty odpovídají časům do konce epizody (tyto časy budou popsány níže), předposlední hodnota reprezentuje aktuální poškození auta a poslední dává informaci, zda se agent aktuálně někoho dotýká.



Obrázek 4.4: Detekce překážek

Parkovací neuronová síť

Vstupy druhé sítě jsou velmi podobné vstupům první sítě. Můžeme je shrnout následovně:

- detekce krajů silnice - 24 hodnot
- detekce překážek - 74x3 hodnot
- parkovací místo - 3 hodnoty
- aktuální rychlost agenta - 3 hodnoty
- předchozí výstupy sítě - 3 hodnoty
- globální časový timeout
- čas do úspěšného konce experimentu
- poškození auta
- příznak, zda se agent právě někoho dotýká

Pro detekci krajů silnice a překážek používáme stejné paprsky jako u první sítě, jediný rozdíl je, že u překážek nepředáváme úhly svírající dopředné vektory agenta a překážky, protože při parkování není tato informace striktně potřeba. Parkující místo reprezentujeme třemi vstupy: první 2 hodnoty kódují pozici parkovacího místa podobně jako v předchozí síti ve formátu *úhel-vzdálenost* a třetí hodnota reprezentuje úhel svírající dopředný vektor agenta a směrová přímka

cílové pozice. Agent má povoleno na cílové místo zajet jak jízdou vpřed tak couváním, výsledný úhel před normalizací tedy bude z rozsahu $\langle -90, 90 \rangle$.

Následují standardní vstupy reprezentující rychlost agenta a předchozí výstupy sítě, dva vstupy reprezentují čas do konce epizody (také tyto časy budou popsány níže) a poslední dvě hodnoty opět udávají aktuální poškození auta a příznak, zda se agent dotýká jiného agenta.

4.2.2 Výstupy

Obě námi používané sítě mají 3 výstupy:

- plyn
- natočení volantu
- brzda

Výstupy odpovídající plynu a volantu můžou nabývat hodnot z rozsahu (-1,1), signál brzdy bere v potaz hodnoty pouze z rozsahu (0,1). Brzda slouží primárně k úplnému zastavení agenta na parkovišti, brzdit v průběhu jízdy lze i pomocí signálu plynu. Výslední agenti většinou k brzdění používají kombinaci obou signálů.

Hodnotu signálu natočení volantu před předáním autu umocňujeme na 2, což pomáhá agentovi dělat drobnější změny směru a méně kličkovat.

4.3 Funkce odměny

Dle doporučení ml-agents jsou velikosti odměn navrhnuty tak, aby valná většina epizod končila s kumulativní odměnou z rozsahu (-1,1). Nicméně vzhledem k tomu, že sčítáme několik typů odměn, nebylo možné se vyhnout občasným případům, kdy odměna opustí tento rozsah.

Při navrhování funkce odměny naším cílem nebylo přesně hodnotit, jak dobře si agent vede, spíše jsme se snažili volit odměny, které dostatečně motivují agenty ke správnému chování. V experimentech se ukázalo, že je obecně lepší používat spíše pozitivní motivaci než negativní. I když používané postihy nejsou relativně příliš velké, hojnější používání postihů může vést k horšímu učení a síť může mít tendenci dostávat se do stavu, při kterém agent pouze stojí na místě a nic nedělá. Současně je dobré volit obtížnost úlohy tak, aby měl agent reálnou šanci dosáhnout cíle.

Pokud bychom také chtěli přidávat postihy za několik různých typů chyb, byl by problém nastavit správně velikost těchto postihů tak, aby zároveň jednotlivé postihy nebyly příliš malé na to, aby dostatečně ovlivnily chování agenta, a zároveň aby nebyly tak velké, že by narušily učení. Naším řešením tedy bylo vnořit některé tyto postihy do základní odměny za postup.

Jízda po silnici

Námi používaná funkce tedy dává odměny za postup v cestě a dosáhnutí cíle a postihy za bourání do ostatních agentů, změnu signálu zatáčení a předčasné

ukončení epizody z důvodu vyjetí ze silnice nebo kvůli časového limitu. Odměna za postup má za cíl dávat agentovi směr, kterým by se měl ubírat, odměna za dosáhnutí cíle slouží jako dodatečná motivace k tomu, aby se snažil nedělat chyby, které by epizodu předčasně ukončili, ale aby se snažil dojet až do cíle.

Velikost odměny za postup je ovlivněna tím, jak dobře agentovo chování v průběhu jízdy vypadá, konkrétně jestli se drží středu silnice a jestli udržuje dostatečný odstup od aut před ním. Postih za změnu signálu zatáčení je nutný, aby se agent snažil zbytečně nekličkovat a výsledné chování vypadalo přirozeně. Podobně bychom mohli zavést postihy i za změnu signálů brzdy a plynu, nicméně nechceme zahlcovat agenta negativními postihy a i bez těchto postihů se agent chová dostatečně dobře. Ze stejných důvodu nepoužíváme ani žádný postih za čas, který se často používá k motivování agenta, aby se snažil splnit úlohu co nejrychleji.

Pro trénování jízdy po silnici tedy používáme následující odměny: za dosažení cíle agent dostává konstantní odměnu 0.3. Dále za postup v cestě agent dostává souhrnnou odměnu 0.5 za celou cestu. V každém frame (50/sek) se kontroluje aktuálně dosažený postup na trati a pokud agent vylepšil svojí nejlepší dosaženou pozici, přičte se odpovídající část odměny vynásobená dvěma koeficienty. První z nich definujeme jako:

$$c_{obs} = clip(d_{obs}/4, 0, 1),$$

kde d_{obs} je vzdálenost nejbližšího auta před agentem[2]. Pro agenta je tedy výhodnější udržovat si určitý odstup od agentů před ním. Druhým koeficientem je koeficient za správnou jízdu. Počítáme směrové úhly dvou bodů na cestě ve fixních vzdálenostech 3 a 5 před agentem a pokud jsou oba úhly menší než 2°, má tento koeficient hodnotu 2, jinak je jeho hodnota 1. Agent je tedy odměňován za stabilní jízdu středem silnice bez kličkování. Jednodušší variantou tohoto koeficientu by byl koeficient založený pouze na vzdálenosti od středu, v experimentech se ale ukázalo, že takovýto koeficient nefunguje dostatečně dobře, agent ve výsledku kličkuje podél středu pruhu, protože není nucen být správně natočen.

Pokud agent vyjede ze své cesty nebo mu dojde čas, dostává odměnu -1 a ukončí se epizoda. Používáme dva časové limity. První z nich je prostý, agent má časový limit na celou epizodu, tento limit je ale nastaven na takovou hodnotu, že se typicky moc neprojeví. Druhý limit udává maximální čas, po který agent může stát na místě bez pohybu: agenti mají 30 sekund, aby ujeli aspoň 5 metrů. Tento čas se ale neinkrementuje, pokud je agent v zácpě, tedy před ním stojí podobně natočený agent, nebo pokud se agent pohybuje vpřed rychlostí alespoň 1. Druhou výjimku používáme hlavně proto, aby nebyla epizoda zbytečně ukončena ve chvíli, když už se agent rozjel. Teoreticky by bylo možné ji částečně zneužít popojížděním tam a zpět, nicméně v praxi se toto chování neobjevilo, pravděpodobně proto, že inkrementování časovače není přímo vázané na žádnou odměnu, pouze na ukončení epizody.

Za bourání do ostatních aut agent dostává dva druhy postihů. První je za samotný akt nabourání, pokaždé když agent do někoho narazí, dostane odměnu -0.1. Dále dostává odměnu -0.001 každý frame (tedy celkem $-0.05/sek$), když se někoho dotýká. Ve chvíli, když součet těchto odměn dosáhne -1, ukončujeme epizodu. Dříve jsme používali přímočařejší řešení, kdy jsme po každé srážce dávali

[2]Maximální hodnota této proměnné je vzdálenost dohledu, tedy 20

odměnu -1 a ukončovali epizodu, nicméně se ukázalo, že to není dobré řešení, agenti nebyli schopni dosáhnout stavu zcela bez kolizí a časté velké postihy za bourání škodily procesu učení. Námi používaná verze způsobí, že srážky občas nastanou, ale nejsou příliš časté a agenti se s nimi umí vypořádat. Výsledná simulace pak sice není zcela bez srážek, nicméně na pohled vypadá dobře.

Parkování

Při trénování parkování agent dostává odměnu za správné umístění a postihy za čas, změnu signálu zatáčení a předčasné ukončení epizody kvůli časovému limitu, bourání nebo vyjetí ze silnice.

Základní funkci odměny pro parkování definujeme následovně:

$$r^p = \max(1 - d/d_{max}, 0) * \max(1 - a/a_{max}, 0),$$

kde d je vzdálenost od parkovacího místa, a je úhel mezi dopředným vektorem agenta a cílovou přímkou a d_{max} a a_{max} jsou vhodné konstanty definované aktuální lekcí. Tyto konstanty na začátku trénování nastavujeme na vysoké hodnoty, aby se měl agent při trénování čeho chytnout, a postupně je snižujeme, aby byl agent nucen být přesnější.

Hodnota této funkce je spočítána v každém framu a agent dostává odměnu rovnou $r^p_t - r^p_{t-1}$, což znamená, že obdrží postih, pokud se od cílové polohy vzdálí.

Experimentovali jsme ještě s možností, že se odměna za parkování přidá pouze jednou na konci epizody, nicméně průběžně aktualizovaná funkce funguje lépe, protože se má agent při trénování lépe čeho chytnout.

Na rozdíl od trénování jízdy, při trénování parkování ukončujeme epizodu po prvním nárazu. Ukázalo se, že pokud povolíme více nárazů, agent se dostatečně nepoučí a ve výsledku stále příliš bourá do okolních aut. Pokud mu dáváme větší postih a po nárazu okamžitě ukončujeme epizodu, agent je nucen být opatrnější a výsledné chování vypadá dobře. Je to pravděpodobně způsobeno tím, že jedna epizoda trvá oproti jízdě po silnici velmi krátkou dobu a menší postih si agent může spíše dovolit ignorovat, protože pro něj může být výhodnější soustředit se spíše na parkování než na eliminaci srážek. U jízdy po silnici si to ale dovolit nemůže, protože je nucen jet dlouhou trať a i občasné srážky můžou způsobit, že nezvládne dojet do cíle.

Dalším možnost skončení experimentu je vyjetí ze silnice nebo časový limit, v obou případech dostává agent odměnu -1. V případě parkování používáme pouze globální časový limit: agent má 40 sekund na celou epizodu.

Pro korektní ukončení epizody se agent musí zastavit se na místě, kde $r^p > 0.4$, a zůstat tam alespoň sekundu bez pohybu. Za korektní ukončení už nedáváme žádnou dodatečnou odměnu. Hodnotu tohoto časovače dostává agent na vstupu.

Agent také dostává každý frame malý časový postih. Ten slouží k tomu, aby se agent snažil danou úlohu vyřešit co nejrychleji. Bez tohoto postihu agent na cílovém místě popojíždí tam a zpět, dokud nenalezne dostatečně dobrou pozici, naším cílem ale bylo vytvořit agenta, který se na místo dostane na první pokus.

Agent také dostává malý postih za změnu signálu zatáčení. Důvod k zavedení tohoto postihu plyne z toho, že ve výsledné simulaci se síť pro parkování aktivuje ve chvíli, když se agent dostane dostatečně blízko parkovacímu místu. V této situaci se typicky agent pohybuje určitou rychlostí a síť učená bez tohoto postihu

většinou začne hodně kličkovat, což kazí dojem z výsledné situace. Proto při trénování parkování iniciujeme agenta s vhodnou dopřednou rychlostí a používáme zmíněný postih.

4.4 Trénování

Trénování provádíme na výše představené mapě a umisťujeme do ní 40 agentů. Celkově se snažíme agenty trénovat v těžším prostředí než jaké pak budeme používat ve výsledné simulaci. Výslední agenti nejspíš nebudou nikdy zcela perfektní, nicméně díky tomuto přístupu budou výsledné chyby menší. Například při generování mapy používáme užší silnice než ve výsledné simulaci (šířka 1.1 vs náhodná šířka z rozmezí (1.1,1.4)) a také dříve ukončujeme epizodu, pokud se agent přiblíží okraji tratě.

Sítě nejprve krátce trénujeme pomocí Behaviour cloning, v průběhu učení pak používáme GAIL a curiosity moduly. Jako expertní vstupy pro BC a GAIL používáme ručně nahrané příklady. Bohužel příklady nejsou dokonalé, auto bylo ve vzorové simulaci ovládáno pomocí šipek, výstupy sítí tedy nabývaly pouze hodnot z množiny -1, 0, 1 a auto tedy značně kličkovalo a občas i vyjelo ze silnice. GAIL tedy používáme primárně pro rychlejší nastartování trénování a časem ho zcela vypínáme.

Pro zlepšení výsledku také dvakrát dělíme learning rate hodnotou 10, typicky v situacích, když učení nějakou dobu stagnuje.

Celkové trénování sítí trvá na našem počítači kolem 3 dnů při paralelní simulaci ve 3 instancích Unity prostředí.

Konkrétní hodnoty používané v curriculum learning je možné si zobrazit v elektronické příloze práce ve složce CONFIG/CURRICULA.

Jízda po silnici

Pro trénování agentů používáme curriculum learning, trénování je tedy rozděleno na několik lekcí. Mezi lekcemi přecházíme ve chvíli, když agent dané zadání dostatečně zvládá, tedy když je průměrná kumulativní odměna za epizodu dostatečně velká.

V první lekci se agenti učí základní jízdě: generujeme jim krátké cesty, každá pouze přes jednu křižovatku. Díky krátkým cestám se agenti málokdy potkávají, a tak agenti nedostávají moc postihů za srážky, které by je v tuto chvíli akorát zbytečně mátly. Také neumocňujeme signál zatáčení a nepostihujeme agenty za jeho změnu. V experimentech se ukázalo, že agenti mají problém se naučit zatáčet, pokud signál umocňujeme už od začátku, rychlejší bývá je trénovat nejprve bez umocňování a přidat ho až dodatečně. Agent natrénovaný na standardní verzi nemá problém se přeučit na verzi s umocňováním.

V druhé lekci pouze zvyšujeme obtížnost prodloužením minimální délky cesty. Ve třetí lekci konečně zavádíme umocňování signálu zatáčení a přidáváme i postih za jeho změnu.

V dalších lekcích prodlužujeme délku cesty, čímž nutíme agenty být přesnějšími. Postupně také zavádíme start na parkovacím místě, aby se agent naučil z něj vyjíždět. Z pohledu agenta je vyjíždění z parkovacího místa značně podobné

sledování cesty, zavádíme ho tedy až později, protože to díky již získaným dovednostem agent relativně snadno zvládá. Na závěr ještě agenty nějakou dobu trénujeme s 80 agenty na mapě.

Parkování

Na začátku každé epizody je agent postaven na náhodné místo na silnici několik metrů před parkovacím místem a je mu vygenerována náhodná rychlost vpřed. Snažíme se napodobovat situace, které budou nastávat ve výsledné simulaci ve chvíli, když je parkovací síti bude předána kontrola.

V lekcích curriculum learning primárně upravujeme konstanty používané ve funkci odměny. Začínáme s vysokými hodnotami, aby se mělo učení čeho chytnout, a postupně je snižujeme, aby byl agent nucen být přesnější. Postihy za čas a změnu signálu zatáčení také nepoužíváme hned od začátku, zavádíme je později až když se agent zvládne na cílové místo nějakým způsobem dostat.

4.5 Výsledná simulace

Ve výsledné simulaci je naším cílem, aby vypadala co nejlépe, snažíme se tedy zvýšit úspěšnost agentů tím, že uvolníme některá omezení světa. Konkrétně zvýšíme časový limit na zlepšení postupu na dvojnásobek, ukončení epizody z důvodu vyjetí ze silnice provádíme až, když je agent 5 sekund středem auta mimo silnici a zvýšíme maximální možné poškození agentů u obou chování na dvojnásobek. U časového limitu dáváme sítím na vstup informace normalizované podle staré maximální hodnoty[3], agenti se totiž někdy naučí konat až ve chvíli, kdy jim opravdu dochází čas, a to už někdy bývá pozdě, samotné zvýšení časového limitu by tedy nemuselo dostatečně pomoci.

Bohužel žádné z těchto omezení nemůžeme zcela zrušit, agenti nejsou zcela bezchybní a v některých situacích situacích je lepším řešením agenta restartovat, než mít na silnici špatně chovající se auto. Například se například stává, že se proti sobě postaví 2 agenti na křižovatce a neví, jak se objet, a tak pouze čekají. Takové případy je vhodné po určitém čase restartovat.

[3]normalizovanou hodnotu ale ořezáváme, aby byla max 1

5. Experimenty a výsledky

V této kapitole si představíme vlastnosti výsledných sítí a chování a podpoříme některá provedená rozhodnutí ohledně způsobu trénování pomocí názorných experimentů.

Experimenty byly prováděny na stolním počítači s 16GB RAM a procesorem Intel Core i5-8500 CPU @ 3.00GHz. Veškeré trénování probíhalo na procesoru, žádný model jsme netrénovali přímo na grafické kartě.

5.1 Způsoby evaluace

Hlavním cílem této práce bylo vytvořit simulaci, která vypadá dobře na pohled. Něco takového je těžké samo o sobě ohodnotit numericky, budeme tedy používat několik objektivních metrik.

Hlavní naší metrikou bude úspěšnost výsledného chování, definujeme jí jako podíl epizod, ve kterých se agent dostal do cíle bez nutnosti restartování z důvodu hrubé chyby. Další používanou metrikou je průměrné poškození, což můžeme přeložit jako průměrný počet drobných srážek za epizodu. Dále měříme průměrný čas na úspěšnou epizodu, což nám může ukázat, jestli síť není přehnaně opatrná a neztrácí zbytečně čas v porovnání s ostatními sítěmi.

Důležitou metrikou je i podíl cesty, který agent urazil korektně umístěný, toto umístění jsme popisovali v kapitole 4.3 o odměnách. Hlavní smysl této metriky je ohodnotit, jak moc agent kličkuje a tato formulace této metriky nám přijde názornější než metrika hodnotící míru zatáčení nebo změny zatáčení. Tyto metriky totiž můžou být zavádějící, pokud se například agent bude dostávat do složitých situací na křižovatkách.

Přesnost parkování budeme měřit výslednou odměnou, jak jsme popsali v kapitole 4.3. Pro potřeby funkce odměny používáme konstanty $d_{max} = 5$ a $a_{max} = 90$.

Cumulative reward zobrazovaný v Tensorboard v průběhu trénování je pro nás spíše orientační, v průběhu trénování měníme v rámci curriculum learning několikrát parametry prostředí, reward tedy nelze vždy použít jako absolutní metriku, pokud neporovnáváme sítě natrénované za stejných podmínek.

Na grafech odměn je nicméně vidět, že náš způsob trénování vede k určitému overfittingu, po každém restartování učení a vygenerování nových náhodných pozic parkovacích míst následuje mírný propad průměrné odměny. Pro lepší výsledky by bylo pravděpodobně lepší častěji měnit mapu, ale to by s sebou neslo další problémy v podobě předčasného ukončování epizod. Výsledná ale chování vypadají dobře i bez změn mapy, tak to nebylo striktně potřeba.

Stejně tak jelikož trénování trvá dlouho a neměli jsme možnost používat počítač, na kterém by celou dobu běželo pouze trénování, rychlost simulace při trénování se v průběhu dne mohla značně lišit, a tak výsledný čas potřebný k trénování je také spíše orientační.

Kvalitu výsledných sítí testujeme při následujícím nastavení: 40 agentů, délka tratě alespoň 1000, šířka silnic $1.1 - 1.4$ a používáme mírnější nastavení restartování popsané v kapitole 4.5. Také pro každý test používáme stejný seed při generování generování parkovacích míst a šířek silnic.

	úspěšnost	čas	umístění	srážky	parkování
cesta	98.29%	186.01s	65.27%	1.02	-
cesta z parkoviště	97.12%	184.80s	64.30%	1.37	-
parkování	98.39%	10.25s	-	0.27	0.84
kompletní simulace	93.66%	198.21s	64.23%	2.37	0.82

Tabulka 5.1: Parametry výsledných chování.

	úspěch	srážka	timeout	vyjetí ze silnice
cesta	98.29%	1.19%	0.40%	0.13%
cesta z parkoviště	97.12%	2.61%	0.26%	0.00%
parkování	98.39%	0.17%	0.94%	0.50%
kompletní simulace	93.66%	2.54%	2.11%	1.69%

Tabulka 5.2: Důvody ukončení epizody výsledných chování.

5.2 Výsledné sítě

V této sekci si představíme vlastnosti výsledných neuronových sítí a odpovídajících chování. Ukázky simulace je možné si prohlédnout na krátkých videích v příloze práce, delší ukázky si můžeme prohlédnout na adresách `https://youtu.be/zQ003KpHcv4` pro samotnu jízdu a `https://youtu.be/Ex5pPEGExJE` pro jízdu s parkováním.

Statistiky výsledných chování si můžeme prohlédnout v tabulkách 5.1 a 5.2. Druhý řádek obou tabulek reprezentuje nastavení, kdy agent začíná na parkovacím místě, musí z něho bezpečně vyjet a následně sleduje standardní trať jako v prvním případě. Ve čtvrtém případě se navíc ještě na konci tratě přepne na parkovací síť a musí zaparkovat na přiřazené parkovací místo.

Z tabulek je vidět, že základní chování sice nejsou perfektní, nicméně mají poměrně dobré úspěšnosti. Pravděpodobnost selhání 1.71% v průběhu sledování cesty trvající 186 sekund v simulaci se 40 agenty znamená, že v průměru bude potřeba nějakého agenta restartovat jednou za cca 4.5 minuty. Tedy se to stává tak vzácně, že je velká šance, že si samotného restartování externí pozorovatel ani nevšimne.

Dalším faktorem je, jak chování vypadá na pohled. Z videa se můžeme přesvědčit, že chování pozorovatele neruší častými významnějšími chybami, agent zvládá jet po rovných silnicích středem pruhu bez většího kličkování, jenom občas po zatáčkách mu chvíli trvá, než se srovná. V kolonách zvládá udržovat přirozený odstup.

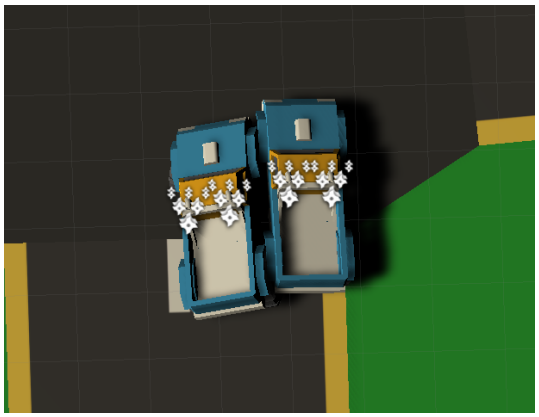
Na křižovatkách je ale patrné, že chování bylo strojově natrénováno. Projíždění rušnými křižovatkami probíhá značně živelně, přednost mívá zpravidla ten, kdo se na křižovatku dostane první. Občas agenty taková strategie dostane do stavu, kdy jsou příliš blízko autu, které před nimi projíždí, v takových případech se agenti naučili lehce couvat. V ideálním případě bychom se mohli obejít zcela bez couvání, nicméně nebereme ho jako větší chybu. Agenti jinak nejsou přehnaně opatrní, jednou křižovatkou zvládne projíždět několik agentů najednou, pokud jim to jejich dráhy dovolí.

Jediné větší problémy jízdy, které by mohly rušit výslednou simulaci jsou občasné zbytečné brzdění v zatáčkách ve chvílích, když agent nezvládne správně

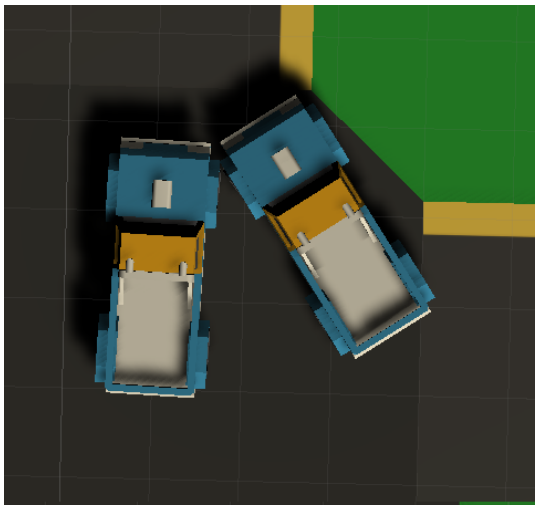
rozpoznat, že protijedoucí auto není hrozba, a občasné srážky. Ty nastávají primárně na křižovatkách a občas při vyjíždění z parkovacího místa, což je důvod menší úspěšnosti v experimentech založených na cestě z parkoviště oproti experimentům na standardní cestě. Agenti špatně ohodnotí danou situaci a nastane srážka, ze které se většinou dokážou rychle dostat couváním. Ve vzácných případech se ale rozhodnou necouvat, nebo se může stát, že se do sebe zaklesnou a jedou dále bok po boku, dokud jeden z agentů není restartován. Příklady výsledných situací si můžeme prohlédnout na obrázcích 5.1 a 5.2.

Na parkovacím chování je znát, že už nám nezbyl dostatek času na experimentování. Samotné chování vypadá v izolaci rozumně, ve valné většině případů se agent dostane na přiřazené parkovací místo, i když výsledná parkovací pozice bývá občas nepřesná, což může vizuálně kazit simulaci.

Horší situace nastává ve chvíli, kdy obě chování zkombinujeme. Řídící chování je na přítomnost ostatních agentů dostatečně zvyklé a parkující agenti mu nedělají větší problémy. Oproti tomu parkovací chování je problém natrénovat na přítomnost ostatních agentů, parkovací agenti při se trénování příliš často nepotkávají. Výsledné chování pak není dostatečně robustní a častěji se pak dostává do situace, kdy si neví rady a skončí epizodu timeoutem nebo srážkou.



Obrázek 5.1: Příklad 1 problému řídícího chování.



Obrázek 5.2: Příklad 2 problému řídícího chování.

5.3 Hyperparametry sítě

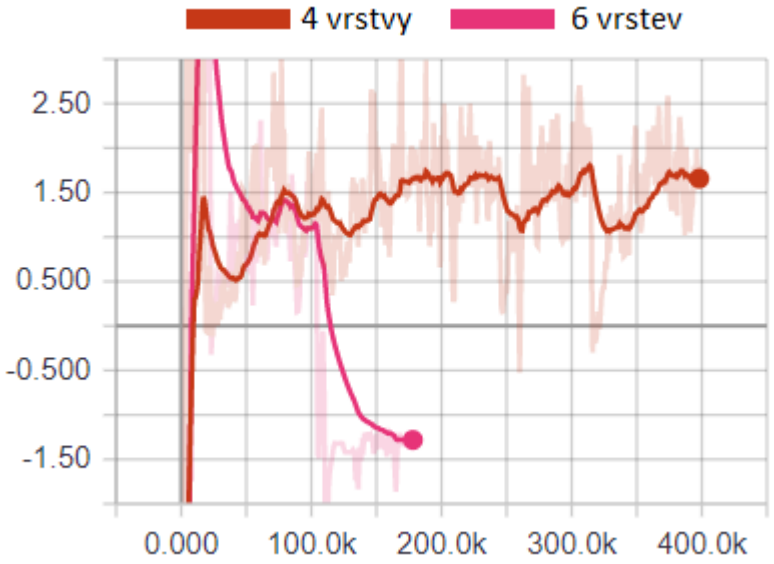
V této sekci se zaměříme na výběr hyperparametrů sítě. Porovnávali jsme 3 verze sítě s 3, 4 a 5 skrytými vrstvami, v každé vrstvě 256 neuronů, každou síť jsme nechali trénovat cca 3 až 4 dny. Výsledky si můžeme prohlédnout v tabulce 5.3.

	úspěšnost	čas	umístění	srážky
3 vrstvy	96.42%	251.48s	54.56%	1.24
4 vrstvy	98.82%	234.07s	54.98%	0.81
5 vrstev	100.00%	247.96s	55.02%	0.54

Tabulka 5.3: Porovnání úspěšností sítí s různými hyperparametry.

Z dat je znatelné, že všechny výsledné sítě mají rozumnou úspěšnost a výsledná chování by byla potencionálně použitelná pro simulaci. Z výsledků je také znatelný trend, kdy přidání vrstev zvyšuje úspěšnost výsledného modelu, bývá to ale vykoupeno pomalejším a méně stabilnějším trénováním, kdy kumulativní odměna sítě má větší výkyvy a síť má větší tendenci se dostat do stavu, kdy se agent přestane hýbat a síť se přestane učit. Příklad takového učení si můžeme zobrazit na obrázku 5.3 ze staršího experimentu, kdy se při identickém nastavení síť se 4 vrstvami v pořádku učila, zatímco síť se 6 vrstvami se učit zcela přestala. Z tohoto důvodu většinou radši používáme síť se 4 vrstvami, kde tyto problémy nenastávají, ačkoli v tomto experimentu vyšla o něco lépe síť s 5 vrstvami.

Podobné problémy mohou nastat také přidáním většího počtu neuronů do vrstev nebo při přílišném používání negativních odměn, jak jsme popisovali v kapitole 4.3.



Obrázek 5.3: Příklad problému při trénování sítí s více vrstvami.

Vyšší úspěšnost sítí se 4 a 5 vrstvami je také vykoupena tím, že všechny sítě mají znatelně horší faktor umístění a cca o 30% delší čas na epizodu než naše výsledná síť. Výslední agenti jsou tedy opatrnější na úkor ostatních metrik.

5.4 Efekt curriculum learning

V této sekci se zaměříme na efekt zvyšování obtížnosti prostředí na výsledné chování, konkrétně budeme budeme měnit délku tratě a počet agentů ve scéně.

V tomto experimentu jsme z časových důvodů netrénovali sítě zcela od začátku, pro inicializaci parametrů jsme použili již částečně natrénovanou síť se 4 vrstvami po cca 1.5 dne trénování a trénovali jsme ji dalších zhruba 11 hodin s odpovídajícím nastavením. Výsledky jednotlivých experimentů je možné si prohlédnout v tabulce 5.4.

	úspěšnost	čas	umístění	srážky
40 agentů, délka 300	91.22%	175.56s	59.90%	4.64
40 agentů, délka 1000	96.99%	193.80s	60.72%	2.07
80 agentů, délka 1000	96.29%	194.21s	60.72%	1.50
40 agentů, délka 2000	97.47%	188.84s	59.23%	1.78

Tabulka 5.4: Porovnání úspěšnosti sítí trénovaných v různých prostředích.

Z dat je vidět jasný trend, kdy zvýšení délky trénovací trasy nutí agenta chovat se opatrněji, agenti trénovaní na trase délky alespoň 1000 mají znatelně větší úspěšnost a menší průměrné poškození než agenti trénovaní na cestě délky 300, je to ale opět vykoupeno delším průměrným časem na epizodu. Trénování na cestě délky 2000 už nepřináší tak výrazné zlepšení.

Zvýšení množství agentů na mapě v tomto experimentu neukázalo významnější vliv na výsledný model, oproti experimentu se základním počtem agentů se pouze lehce liší průměrné poškození aut.

Z experimentů je vidět, že vyšší obtížnost úkolu má jistý vliv na výsledný model. Agenty ale není možné na dlouhé trati trénovat už od začátku, protože nepřiměřeně náročný úkol by brzdil učení, proto používáme curriculum learning a obtížnost zvyšujeme postupně.

5.5 Efekt umocnění signálu zatáčení

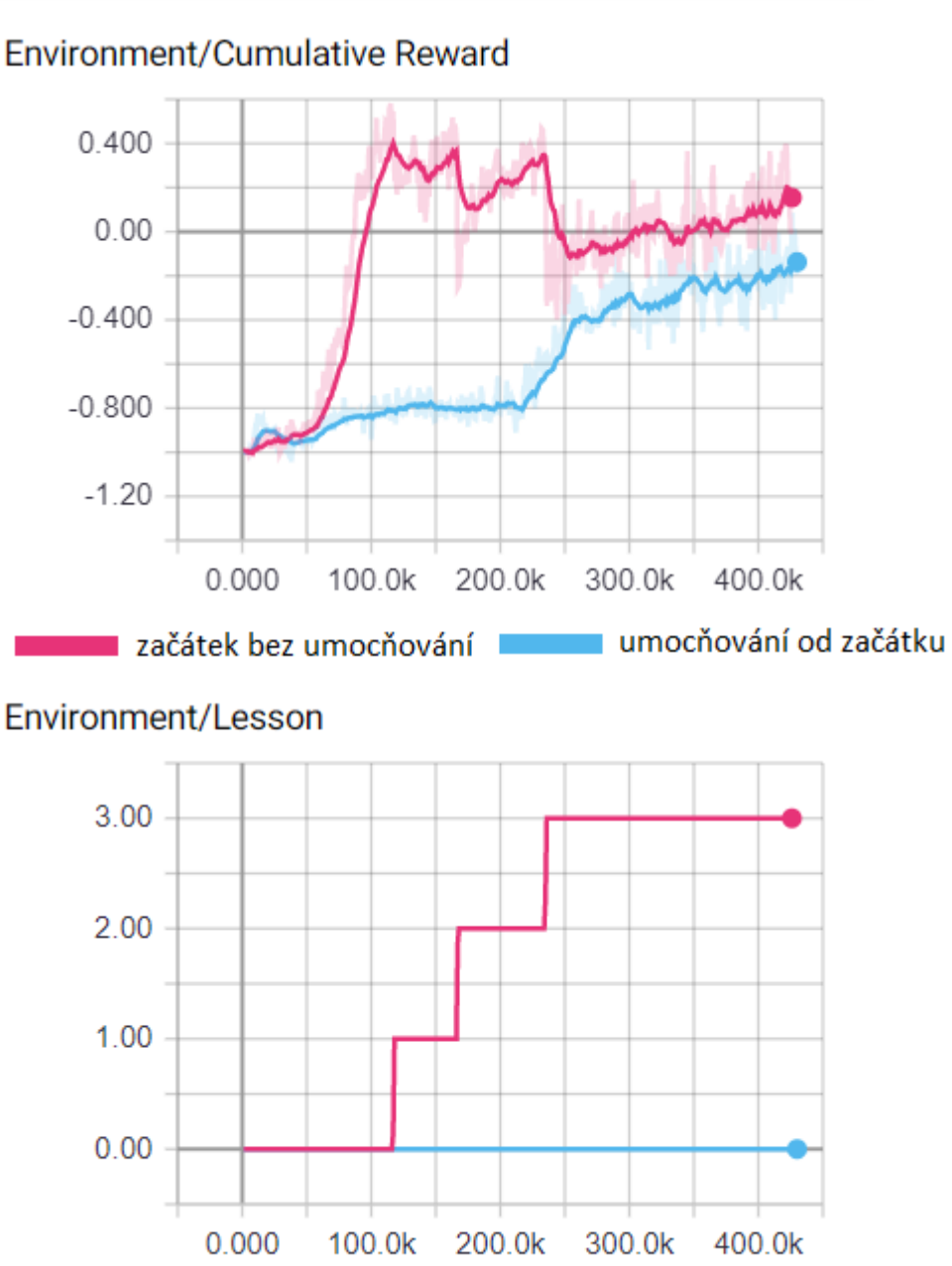
Umocnění steering signálu používáme za účelem omezení kličkování agenta. Význam této úpravy si můžeme prohlédnout v tabulce 5.5. Podobně jako v experimentu 5.4 jsme obě sítě netrénovali zcela od začátku, ale použili jsme již částečně natrénovanou síť, kterou jsme nechali trénovat cca 10 hodin, což poskytlo dostatek času na přeučení se na danou verzi výstupu.

exponent	úspěšnost	čas	umístění	srážky
1	97.17%	191.43s	41.3%	1.51
2	96.99%	193.80s	60.72%	2.07

Tabulka 5.5: Porovnání efektu umocňování signálu zatáčení.

Z údaje o umístění agentů je zřejmé, že pokud neumocňujeme signál zatáčení, má výsledná síť výrazně větší problém se stabilně držet uprostřed silnice.

Umocnění tohoto signálu je ale potřeba zavést až v síti, která už umí obstojně jezdit, jinak může tato úprava značně zpomalit učení. Názorné porovnání si můžeme prohlédnout na obrázku 5.4. Oba experimenty měly téměř stejné nastavení, lišily se pouze v exponentu signálu a délkách trénovacích cest. V standardním experimentu snižujeme obtížnost v podobě délky tratě ve chvíli, když přecházíme na jiné nastavení exponentu. V druhém experimentu toto dělat nemusíme. V rámci základního experimentu začneme signál umocňovat od 2. lekce, od 3. lekce dále mají oba experimenty zcela identické nastavení.



Obrázek 5.4: Porovnání způsobů trénování umocněného signálu zatáčení.

Z výsledných grafů je zřetelně vidět, že pokud umocňujeme signál hned od začátku, výrazně to zhorší proces učení. Zatímco první síť se naučila slušně jezdit

i s upraveným exponentem, druhá síť ve stejném časovém úseku nezvládla konzistentně zdolávat ani nejkratší trať. Umocnění signálu totiž působí, že je výrazně těžší naučit se správně zatáčet, což se nejvíce projeví na netrénované síti.

5.6 Neuroevoluce

V rámci práce jsme prováděli i řadu experimentů s neuroevolucí. Ta ve výsledku neposkytla dostatečně dobré výsledky, agenti se sice naučili poměrně rychle sledovat cestu, ale nepodařilo se nám vytvořit síť, která by dokázala předcházet srážkám. V tomto textu jsme se jí tedy moc nevěnovali, nicméně v této sekci si krátce popíšeme použitý postup a na jaké jsme narazili problémy.

Pro nastavení parametrů evoluce jsme vycházeli z metody popsané v (Such a kol., 2017), která poskytuje dobré výsledky v porovnání s přístupy zpětnovazebního učení. Pro rychlejší trénování jsme používali výrazně menší sítě, většinou jen s 2 skrytými vrstvami a $32-128$ neurony a také s menším množstvím vstupů.

Používali jsme populaci se 40 jedinci. Genomy jedinců přímo kódovaly váhy sítě a byly inicializovány tak, že geny odpovídající biasům se rovnaly 0 a genům odpovídajícím vahám mezi neurony byla přiřazená náhodná hodnota z rozdělení $N(0, \sqrt{2/n_{input}})$, kde n_{input} reprezentuje počet vstupů cílového neuronu.

Při tvorbě následujících generací jsme vždy zkopírovali nejlepšího jedince, zbylou populacie jsme tvořili tak, že jsme vždy vybrali náhodného top 5 jedince (tzv. *truncation selekce*) a zmutovali jsme je tím, že jsme ke každému genu přičetli náhodné číslo z rozdělení $N(0, 0.02)$.

Největším problémem neuroevoluce bylo vytvořit vhodnou fitness funkci. Pro trénování sledování cesty jsme vygenerovali jednu klikatou silnici a nechali 40 agentů, z nichž každý reprezentoval jiného jedince, ať se ze stejné pozice dostanou co nejdál. Výsledná hodnota fitness pak byl dosažený postup na silnici. Jednou za 20 generací jsme vygenerovali novou silnici s cílem minimalizovat přeučení, ale přitom zajistit, aby byl stále prostor pro trénování aktuálního problému. Jakmile agent nejlepšího jedince dokázal stabilně projíždět celou silnici bez chyby, přešli jsme na trénování na trénovací mapě, kde jsme všem agentům vygenerovali stejnou náhodnou cestu. Ohodnocení jedné generace tímto způsobem trvalo cca 30 sekund a trénování takové sítě trvalo v řádu několika hodin. Výslední agenti neměli problém projet libovolnou cestu na mapě.

Tento princip tedy fungoval dobře pro samotnou jízdu, už se nám ale nepodařilo vytvořit vhodnou funkci pro trénování vyhýbání se kolizím. Problém bylo rozpoznat, kdy se agent skutečně zachoval dobře a vlastním přičiněním se vyhnul srážce a kdy měl pouze štěstí. Snažili jsme se to objektivně ohodnotit a při oceňování sítě jsme používali 40 agentů ovládaných jedním jedincem na různých kombinacích nastavení mapy, tratě a výsledné fitness hodnoty, nicméně v průběhu evoluce dominoval vždy jedinec, který měl při daném nastavení štěstí, ale nevyvinulo se chování, které by se vědomě dokázalo vyhýbat srážkám.

Pomoct by mohlo například výrazně delší ohodnocování fitness funkce na velkém množstvím různých situací, což by mělo do určité míry omezit význam štěstí. Vyžadovalo by to ale bohužel přílišné množství času na každé ohodnocení fitness a nebylo v našich možnostech takové experimentování zrealizovat.

Závěr

V této práci jsme se věnovali problému vytvoření simulace aut jezdících po silniční síti a schopných vyhýbat se kolizím s ostatními auty. Model ovládající auta měl být vytvořen použitím metod strojového učení. Sekundárním cílem této práce bylo naučit výsledné agenty parkovat.

V první kapitole této práce jsme si definovali pojem neuronových sítí a představili jsme si některé přístupy, které se používají pro jejich učení. V následující kapitole jsme si krátce představili používané technologie a dále jsme probrali několik prací, které řešili podobný problém. V další kapitole jsme důkladně představili výsledný model a postupy, které jsme použili k trénování. Nakonec jsme experimentálně ohodnotili vlastnosti výsledného chování a podpořili jsme některá učiněná rozhodnutí konkrétními daty.

Výsledné modely jsou dvě neuronové sítě se čtyřmi skrytými vrstvami. Na vstupu dostávají značné množství dat ze senzorů auta, jde převážně o detekci okolních překážek a okrajů silnice a pozice následujících bodů v cestě. Tři výstupy obou sítí určují hodnoty plynu, brzdy a natočení volantu auta. Sítě jsme trénovali algoritmem Proximal Policy Optimization a pro vylepšení učení jsme používali přístupy GAIL, curiosity, curriculum learning a Behavioral cloning.

První výsledná síť je schopná sledovat externě vygenerovanou cestu a s velkou úspěšností se vyhýbat srážkám s ostatními agenty na městské síti bez světelné signalizace. Ačkoli výsledné chování není zcela perfektní, velké chyby nastávají dostatečně vzácně, takže vytvořená síť je potencionálně použitelná pro simulaci. Parkovací síť pracuje s o něco horší úspěšností, v izolaci sice dokáže dobře parkovat, v reálném prostředí mívá někdy problém správně reagovat na okolní agenty.

V budoucnu bychom se mohli více zaměřit na přesnost parkování, na které nám už nezbylo dostatek času. Dalšími rozšířeními práce můžou být různé úpravy parametrů prostředí: přidání kopců, nebo různých statických či pohyblivých překážek. Možným upravením problému je také přidání vizuálních vstupů a použití konvolučních vrstev.

V přílohách práce si ještě představíme GUI vytvořeného prostředí, předvedeme si, jak sítě používat a trénovat a stručně si představíme strukturu vytvořeného programu.

Seznam použité literatury

(2004). Openstreetmap xml format. https://wiki.openstreetmap.org/wiki/OSM_XML.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M. a kol. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Abbeel, P. a Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1.

AbuZekry, A., Sobh, I., Hadhoud, M. a Fayek, M. (2019). Comparative study of neuroevolution algorithms in reinforcement learning for self-driving cars. *European Journal of Engineering Science and Technology*, **2**(4), 60–71.

Bain, M. a Sammut, C. (1995). A framework for behavioural cloning. In *Machine Intelligence 15*.

Bengio, Y., Louradour, J., Collobert, R. a Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553380. URL https://doi.org/10.1145/1553374.1553380.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J. W., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F. a Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *ArXiv*, **abs/1912.06680**.

Chen, J., Li, S. E. a Tomizuka, M. (2020). Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning. *arXiv preprint arXiv:2001.08726*.

Fukushima, K. a Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer.

Ganesh, A., Charalel, J., Sarma, M. D. a Xu, N. (2016). Deep reinforcement learning for simulated autonomous driving.

Haarnoja, T., Zhou, A., Abbeel, P. a Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*.

He, K., Zhang, X., Ren, S. a Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Helgason, D., Francis, N. a Ante, J. (2007). Unity. `https://unity.com/`.

Ho, J. a Ermon, S. (2016). Generative adversarial imitation learning. In *Advances in neural information processing systems*, pages 4565–4573.

Hochreiter, S. a Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, **9**(8), 1735–1780.

Juliani, A., Berges, V.-P., Teng, E. a et al., A. C. (2017). Ml-agents. `https://github.com/Unity-Technologies/ml-agents`.

Kaushik, M., Prasad, V., Krishna, K. M. a Ravindran, B. (2018). Overtaking maneuvers in simulated highway driving using deep reinforcement learning. In *2018 ieee intelligent vehicles symposium (iv)*, pages 1885–1890. IEEE.

Krizhevsky, A., Sutskever, I. a Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Lague, S. (2019). Path creator. `https://github.com/SebLague/Path-Creator`.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. a Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Milsonneau, J. (2018). Arcade car physics. `https://github.com/Saarg/Arcade_Car_Physics`.

Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. a Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. a Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. a Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Pathak, D., Agrawal, P., Efros, A. A. a Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17.

Ramachandran, P., Zoph, B. a Le, Q. V. (2017). Swish: a self-gated activation function. *arXiv: Neural and Evolutionary Computing*.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, **65**(6), 386.

Rumelhart, D. E., Hinton, G. E. a Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, **323**(6088), 533–536.

Salimans, T., Ho, J., Chen, X., Sidor, S. a Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. a Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. a Klimov, O. (2017). Proximal policy optimization algorithms. *ArXiv*, **abs/1707.06347**.

Shalev-Shwartz, S., Shammah, S. a Shashua, A. (2016). Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K. a Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, **abs/1712.01815**.

Stanley, K. O. a Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, **10**(2), 99–127.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O. a Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K. a kol. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Wymann, B., Espié, P., Guionneau, C., Dimitrakakis, C., Coulom, R. a Sumner, A. (2000). Torcs, the open racing car simulator". `https://sourceforge.net/projects/torcs/`.

Yu, A., Palefsky-Smith, R. a Bedi, R. (2016). Deep reinforcement learning for simulated autonomous vehicle control. *Course Project Reports: Winter*, pages 1–7.

Zhu, M., Wang, X. a Wang, Y. (2018). Human-like autonomous car-following model with deep reinforcement learning. *Transportation research part C: emerging technologies*, **97**, 348–368.

Seznam obrázků

1.1 Nákres umělého neuronu. 5
1.2 Nákres dopředné neuronové sítě. 6
1.3 Nákres LSTM buňky 7
1.4 Nákres curiosity modulu. 13

4.1 Mapa používaná pro trénování 18
4.2 Detekce pozic následujících bodů na cestě 21
4.3 Detekce vzdáleností okrajů silnice 22
4.4 Detekce překážek 23

5.1 Příklad 1 problému řídícího chování. 31
5.2 Příklad 2 problému řídícího chování. 31
5.3 Příklad problému při trénování sítí s více vrstvami. 32
5.4 Porovnání způsobů trénování umocněného signálu zatáčení. 34

A.1 První část nastavení prostředí simulace 42
A.2 Druhá část nastavení prostředí simulace 42
A.3 Statistiky o provedených epizodách 44

A. Uživatelská dokumentace

A.1 Instalace potřebných programů

Základním programem pro spuštění výsledné simulace je program Unity, který je možné stáhnout na stránce `https://unity.com/`.

Pro trénování neuronových sítí na systému Windows používáme příkazovou řádku Anaconda Prompt, kterou je možné stáhnout pomocí odkazu:
`https://www.anaconda.com/products/individual#windows`

Potřebné prostředí je možné vytvořit prostřednictvím souboru `environment.yml` nacházejícího se v elektronické příloze práce příkazem:

```
conda env create -f environment.yml
```

prostředí je následně potřeba aktivovat pomocí:

```
conda activate mlag13
```

A.2 Popis prostředí

A.2.1 Parametry prostředí

Veškeré parametry prostředí je možné nastavit v objektu `Settings`. Možné nastavení si můžeme prohlédnout na obrázcích A.1 a A.2. Většina položek má vysvětlující názvy a po najetí kurzorem na názvy položek se zobrazí vysvětlující popisky, v této sekci si tedy vysvětlíme pouze některé z nich.

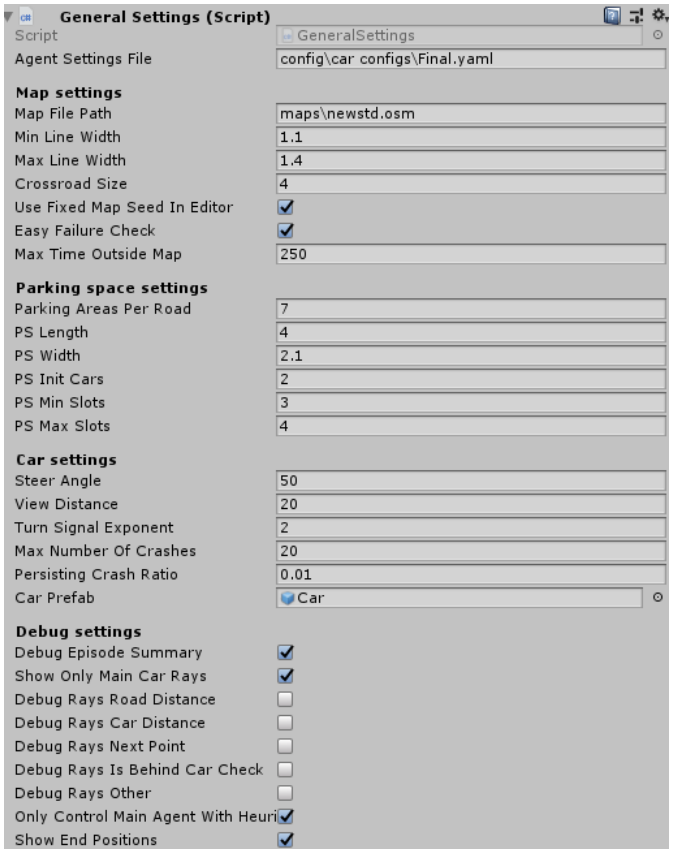
První položka `Agent Settings File` určuje cestu k souboru s popisem vstupů agentovy sítě.

V sekci `Map settings` je možné nastavit parametry mapy. První položka určuje umístění vstupního OSM souboru, `Use Fixed Map Seed In Editor` nastavuje, zda se v prostředí unity bude používat pevně daný seed pro generování parkovacích míst a šířek silnic a `Easy Failure Check` zapíná lehčí nastavení výsledné simulace popsané v kapitole 4.5.

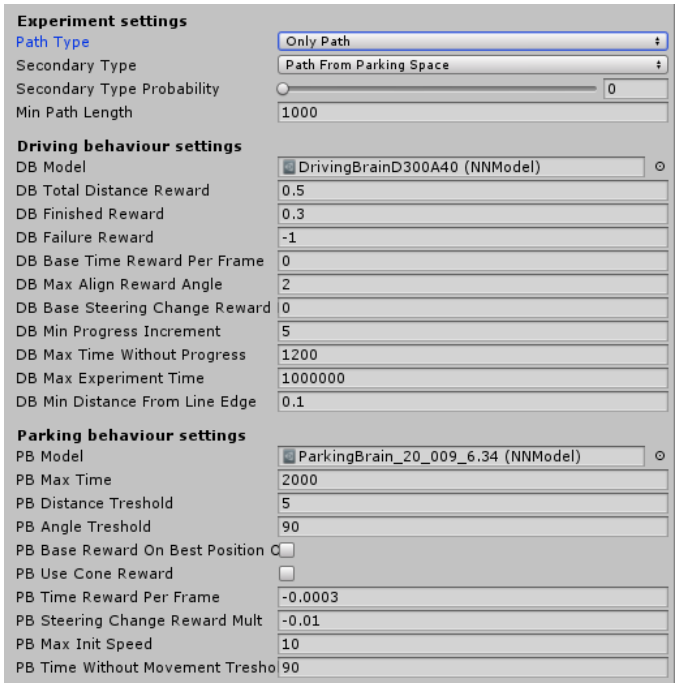
V sekci `Debug settings` je možné nastavit vypisování shrnutí jednotlivých epizod do konzole pomocí `Debug Episode Summary`. Další nastavení slouží k zobrazování různých informací scéně pomocí paprsků, většinou jde o paprsky reprezentující různé vstupy jednotlivých sítí.

V sekci `Experiment settings` je možné nastavit typ a délku generované cesty. Sekundární typ je možné použít, pokud chceme s určitou pravděpodobností používat alternativní variantu cesty. Základními typy jsou jednoduchá cesta a samotné parkování, jednoduchou cestu je déle možné obohatit o začátek či konec na parkovacím místě. Varianta `Full simulation` navíc nastavuje, že místo teleportování mezi epizodami se budou agenti převtělovat do zaparkovaných aut. Tato simulace ale momentálně nevypadá dostatečně dobře, protože výsledné parkovací chování není dostatečně přesné.

Poslední dvě sekce slouží k nastavení parametrů jednotlivých chování. `DB Max Align Reward Angle` nastavuje maximální úhel pro získání bonusu za správné



Obrázek A.1: První část nastavení prostředí simulace



Obrázek A.2: Druhá část nastavení prostředí simulace

umístění, `DB Base Steering Change Reward` určuje postih za změnu signálu zatáčení. Aby měla tato hodnota správný efekt a její vliv byl stejný při různých délkách tratě, je před aplikováním vždy vydělena délkou aktuální trasy. Výsledný postih je tedy počítán pomocí $|s_{t-1} - s_t| * c/d$, kde c je zmíněná konstanta, s_{t-1} a s_t jsou hodnoty steering signálu a d je délka tratě. Podobně upravovaný je i postih za čas `DB Base Time Reward Per Frame`.

`DB Min Distance From Line Edge` nastavuje minimální vzdálenost, na kterou se agent může přiblížit středem auta k okraji silnice. Toto nastavení nefunguje, pokud používáme `Easy Failure Check` zmíněný výše.

V sekci o parkování `PB Distance/Angle Treshold` nastavují konstanty používané ve funkci odměny, `PB Base Reward On Best Position` zapíná variantu, kdy agent nedostává postihy za zhoršení pozice a při nastavení `PB Use Cone Reward` bude agent odměňován pouze, pokud se bude nacházet v pozici, ze které se lze přímo dostat na parkovací místo bez popojíždění. Poslední položka `PB Time Without Movement Treshold` nastavuje, jak dlouho musí agent nechybně setrvat na korektním místě, než bude epizoda úspěšně ukončena.

Na objektech jednotlivých agentů je možné si zobrazit různé informace o aktuálním stavu agenta včetně aktuálních vstupů a výstupů sítě. Pro přehlednost jsou vstupy rozděleny do 4 kategorií, v rámci kategorií jsou seřazeny v pořadí, v jakém byly popisovány v kapitole 4.2.1.

A.2.2 Zobrazení statistik

Statistiky o skončených epizodách jsou shromažďovány v objektu `Statistics`, výsledný výpis je možné si prohlédnout na obrázku A.3.

Dvojice nastavení `Auto Pause` v horní části obrázku slouží k pozastavení simulace po určitém časovém úseku pro férové ohodnocení experimentů. Kousek pod nimi je možné upravit grafické znázorňování konečných pozic na mapě.

Dále se v sekci `Ending statistics` ukládají informace frekvencích jednotlivých typů ukončení a v sekci `Episode statistics` jsou zobrazeny ostatní statistiky prezentované v kapitole 5. Poslední položka `Stats` přehledně zobrazuje procentuální pravděpodobnosti jednotlivých zakončení.

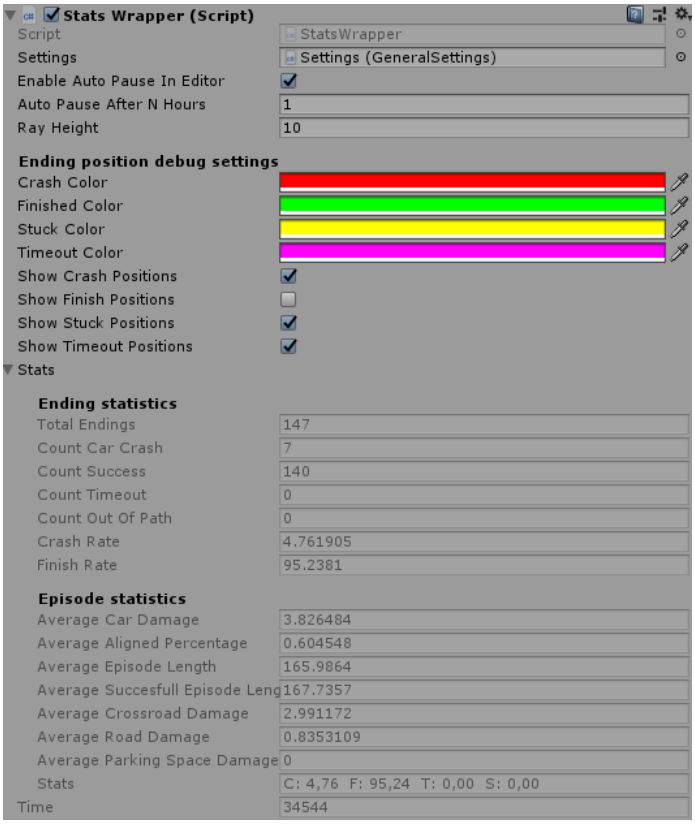
A.3 Simulace

Spuštění simulace je možné pomocí scény `SimulationScene`, která je umístěna ve složce Assets/Global/Scenes. Před spuštěním simulace doporučujeme zkontrolovat, že v nastavení používáme správné sítě a generujeme správný typ cest.

Simulaci je možné zrychlit prostřednictvím `Edit/Project Settings.../Time/TimeScale`.

A.4 Trénování

Pro trénování jsou připraveny scény `DBTrainingScene` pro trénování jízdy a `PBTrainingScene` pro trénování parkování. Obě scény jsou umístěny ve složce Assets/Global/Scenes.



Obrázek A.3: Statistiky o provedených epizodách

Trénování je možné spustit buď přímo v instanci Unity nebo v separátním procesu. Vygenerovat z aktuální scény spustitelný soubor je možné v menu `File/BuildSettings...`, kde je nejprve potřeba označit aktuální scénu pomocí tlačítka `AddOpenScenes` a následně spustit generování pomocí tlačítka `Build`.

Pro trénování je potřeba mít připravený *.yaml* soubor s parametry trénování, případně další *.yaml* soubor s nastavením curriculum learning. Námi používané soubory je možné nalézt ve složkách CONFIG a CONFIG/CURRICULA.

Samotné trénování jízdy pak můžeme po aktivování správného conda prostředí spustit z kořenové složky elektronických příloh například příkazem:

```
mlagents-learn config/DB_Final.yaml
        --curriculum=config/curricula/DB_Final
        --run-id=DB_Test --train
```

a následným spuštěním simulace scény `DBTrainingScene` v Unity. Pokud bychom tuto scénu sestavili do složky BUILDS/DB, můžeme trénovat mimo prostředí Unity příkazem:

```
mlagents-learn config/DB_Final.yaml
        --curriculum=config/curricula/DB_Final
        --run-id=DB_Test --train
        --env="builds/DB/Unity Environment"
        --num-envs=3
```

který spustí trénování ve 3 instancích simulace

Průběh trénování můžeme sledovat v prostředí Tensorboard, které můžeme v kořenové složce spustit příkazem:

```
tensorboard --logdir=summaries --host=127.0.0.1
```

Tensorboard pak můžeme otevřít v prohlížeči na lokální adrese `http://127.0.0.1:6006/`.

B. Programátorská dokumentace

V této příloze se zaměříme na základní přehled námi naprogramovaného kódu. Většinu kódu nalezneme ve složce Assets/Global/Scripts, jedinou výjimkou jsou třídy ve složceAssets/Editor ovládající některé prvky editoru, u nich je umístění vynucené prostředím Unity.

Pro vytvoření výsledné simulace jsme kromě ML-Agents používali navíc ještě assety Arcade Car Physics (Milsonneau, 2018) pro simulování aut a Bézier Path Creator (Lague, 2019) pro generování silnic.

Námi vytvořený kód je strukturovaný do několika skupin tříd. První skupina se nachází ve složce `Maps`. Třída `RealMap` slouží k uchovávání základních informací o mapě, třídy `MapNode` a `MapWay` slouží pro reprezentaci bodů a cest v průběhu načítání mapy z OSM XML formátu, `Crossroad`, `Road` a `ParkingSpace` pak reprezentují jednotlivé stavební bloky načtené mapy. `PathGenerator` slouží k generování cesty dle zadaných parametrů, výsledná cesta je pak reprezentovaná instancí třídy `Path`. Postup agenta na dané cestě následně kontroluje třída `PathChecker`. Třídy ve vnořené složce MapGenerator slouží k procedurálnímu generování jednoduchých map pro trénování pomocí neuroevoluce a třídy ve složce `Obstacles` zvyšují komplexitu problému přidáním překážek.

Další skupinou jsou třídy ve složce Agents. Základní funkčnost agenta zajišťuje třída `GenericAgent`, konkrétní agentova chování pak implementují třídy v podsložce AgentBehaviours, konkrétně se jedná o třídy `DrivingBehaviour` a `ParkingBehaviour`. Kód používaný oběma chováními implementuje jejich abstraktní předek `GenericAgentBehaviour`. `RLAgent` slouží jako wappper třídy `GenericAgent` pro komunikaci s `RLAcademy` zajišťující běh experimentů.

V třídách `Utility` a `Extensions` jsou umístěny různé statické a extension funkce, které používáme v různých částech programu. `Statistics` a `StatsWrapper` slouží u uchovávání informací o výsledcích epizod v aktuálním běhu. Jednoduchý skript `CollisionHandler` je určen k přeposílání informací o srážkách z objektu obsahující `Rigidbody` do scriptu agenta. `GeneralSettings` na jednom místě uchovává většinu nastavení aktuálního experimentu. Třída `RLAcademy` je potomek abstraktní třídy `Academy` zajišťující běh agentů a instance třídy `LastingDebugRay` slouží k uchovávání informací o aktuálně zobrazených debug paprscích.

C. Popis elektronických příloh

V této sekci si krátce popíšeme důležitý obsah elektronických příloh práce.

Ve složce Assets nalezneme veškeré zdrojové soubory pro spuštěni simulace, ve složce Config a jejích podsložkách nalezneme konfigurační soubory používané v experimentech.

V adresáři Summaries jsou uloženy Tensorboard záznamy popsaných experimentů, výsledné modely najdeme ve složce Assets/Global/NeuralNets.

V příloze také najdeme dvě krátká videa s ukázkami výsledných simulací a soubor `environment.yml` sloužící k vytvoření conda prostředí vhodného k trénování sítí.