Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Navigace v neznámém a pevně daném prostředí pomocí deep reinforcement learning algoritmu

2022

Bc. Gabriela Hrubá

Vedoucí práce:
RNDr. Martin Trnečka, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Gabriela Hrubá
Název práce:	Navigace v neznámém a pevně daném prostředí pomocí deep reinforcement learning algoritmu
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2022
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	RNDr. Martin Trnečka, Ph.D.
Počet stran:	48
Přílohy:	1 DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Gabriela Hrubá
Title:	Deep Reinforcement Learning Navigation in an Unknown and Fixed Environment
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2022
Study field:	Computer Science, full-time form
Supervisor:	RNDr. Martin Trnečka, Ph.D.
Page count:	48
Supplements:	1 DVD
Thesis language:	Czech

Anotace

Deep reinforcement learning je oblast umělé inteligence, kombinující přístup reinforcement learning a hlubokého učení. Pro stavební stroj, poskytnutý firmou Technotrade, je vytvořena simulace pohybu v pevně daném prostředí. Použitím algoritmu deep Q-learning, který spojuje algoritmus Q-learning s umělými neuronovými sítěmi, je vytvořen model, navigující nakladač v daném virtuálním prostředí do cílové souřadnice okolo rozmístěných překážek.

Synopsis

Deep reinforcement learning is an area of machine learning which combines the principles of reinforcement learning and deep learning. Construction vehicle is provided by the company Technotrade. A driving simulation in a stable fixed environment is developed for the given vehicle. Using the deep Q-learning algorithm, which merges Q-learning algorithm with artificial neural networks, a model is implemented. This model navigates the vehicle to the goal coordinates around surrounding obstacles in a virtual environment.

Klíčová slova: reinforcement learning; neuronové sítě; deep reinforcement learning; deep Q-learning; machine learning

Keywords: reinforcement learning; neural networks; deep reinforcement learning; deep Q-learning; machine learning

Děkuji vedoucímu diplomové práce RNDr. Martinu Trnečkovi, PhD. za velmi cenné rady, vstřícný přístup a pohotovou komunikaci. Také děkuji Danielu Batlovi (Technotrade) za konzultace a pomoc při práci s nakladačem. Na závěr bych chtěla poděkovat mé rodině a Tomovi za podporu a dodávání motivace.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracovala samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Přístup strojového učení zvaný *reinforcement learning* se v poslední době stává velmi populární při řešení problémů, které mají vhodně specifikovatelné prostředí a odpovídající proveditelné kroky. Reinforcement learning je zejména efektivní v oblastech robotiky a videoher [2]. Hlavní nevýhodou tohoto přístupu je velmi rychle narůstající výpočetní a paměťová složitost v závislosti na komplexnosti prostředí, ve kterém algoritmus pracuje. Tento problém řeší přístup *deep reinforcement learning*, který kombinuje reinforcement learning s hlubokým učením. Hluboké učení umožňuje aproximaci daného složitého prostředí a nezpůsobuje výrazné zvýšení výpočetní a paměťové složitosti. Možné oblasti použití deep reinforcement learning algoritmů jsou díky této aproximaci větší než u samotného reinforcement learningu a zahrnují problémy týkající se zpracování řeči, zdravotnictví i financí [3].

Téma této diplomové práce bylo vytvořeno ve spolupráci s firmou Technotrade, která měla zájem o automatizaci stavebního stroje pomocí umělé inteligence. Vybraným stavebním strojem byl nejprve nakladač s kloubovým zatáčením, který byl nakonec vyměněn za nakladač s pásovým pohonem. Nakladač se pohybuje v neměnném prostoru vymezeném čtvercovou sítí, ve kterém jsou také umístěny překážky. Algoritmus má za úkol navigovat nakladač z aktuální pozice do předem dané cílové souřadnice kolem rozmístěných překážek. Reprezentace stavu celkového prostředí spolu s nakladačem je pro přístup reinforcement learning natolik komplexní, že jsem v této práci implementovala deep reinforcement algoritmus *deep Q-learning*, který je velmi vhodný a často používaný v oblasti robotiky k řešení složitějších problémů.

Vzhledem k povaze zadání, jsem se rozhodla nejdříve zaměřit na učení ve virtuálním prostředí, protože umožňuje rychlejší učení modelu a také poskytuje bezpečné podmínky při učení oproti reálnému světu. Pro snadné zobrazení naučeného chování nakladače je implementována i jednoduchá grafická vizualizace celkové simulace.

2 Reinforcement learning

Reinforcement learning, neboli zpětnovazební učení, patří do podoblasti umělé inteligence zvané strojové učení. Reinforcement learning přistupuje k učení podobně jako například lidé. Lidé během učení interagují se svým okolním prostředím a registrují následné změny na základě provedených akcí. Tyto změny poté považují za formu zpětné vazby, která může být pozitivní, nebo negativní. Na tomto principu je založena oblast reinforcement learning.

V této kapitole se věnuji základním principům strojového učení reinforcement learning. Čerpám zejména z mé bakalářské práce [4], která se tímto tématem a Q-learning algoritmem zabývala, a z knihy Reinforcement learning: An introduction [5].

2.1 Markovův rozhodovací proces

Markovův rozhodovací proces formalizuje proces učení a interakce agenta s prostředím. Je definovaný jako n-tice $\langle S, A, T, R\rangle$ obsahující množinu stavů S, množinu akcí A, míru okamžité odměny R a přechodovou funkci T.

Množina stavů S obsahuje veškeré dostupné stavy agenta a množina akcí A obsahuje proveditelné akce agenta. Míra okamžité odměny R určuje hodnotu zpětné vazby za provedení akce a ve stavu R. Tuto míru lze zapsat jako $R : S \times A \times S' \to \mathbb{R}$.

Přechodová funkce $T : S \times A \times S \to [0, 1]$ vyčísluje pravděpodobnost přechodu ze stavu s do stavu s' pomocí akce a. Tuto pravděpodobnost zapisujeme pomocí výrazu $T(s, a, s')$.

2.2 Dělení algoritmů

Vztah mezi stavem s a akcí a je definovaný pomocí strategie π (anglicky policy), na jejichž základě agent vybírá akci, která bude v aktuálním stavu provedena. Veškeré rozhodování agenta se tedy řídí podle předem zvolené strategie.

Reinforcement learning algoritmy se na základě jejich přístupu ke strategiím dělí do dvou skupin. Algoritmy používající přístup *on-policy* se snaží o optimalizaci strategie π, která je používána i během učení. Algoritmy ze skupiny *off-policy* také používají v průběhu učení předem určenou strategii π, jejich cílem je ale nalezení optimální strategie π^*, která je odlišná od strategie π. π tedy reprezentuje trénovací strategii a π^* výslednou optimální strategii.

Dále se v reinforcement learningu rozlišují algoritmy na základě dostupnosti modelu prostředí. *Modelové* (anglicky *model-based*) algoritmy používají k výpočtu model Markovova rozhodovacího procesu. Na základě tohoto modelu může agent předpovědět vliv akce na dané prostředí i následující stav a obdrženou odměnu. Výsledná strategie agenta vzniká na základě plánování v daném modelu. *Bezmodelové* (anglicky *model-free*) algoritmy nepotřebují znát model Markovova rozhodovacího procesu, protože hledají výslednou strategii na základě zkou-

šení v daném prostředí namísto plánování. Tyto algoritmy získávají informace o prostředí pomocí vykonávání akcí a zaznamenávání vzniklých změn v prostředí.

2.3 Q-learning algoritmus

Algoritmus Q-learning patří do skupiny bezmodelových off-policy algoritmů. Tento algoritmus je populární díky své jednoduchosti a široké použitelnosti. Hlavním principem algoritmu je výpočet Q hodnot, určující úspěšnost akce v daném stavu a Q funkce, která aktualizuje Q hodnotu akce a_t v daném stavu s_t.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big) \tag{1}$$

Obecná Q funkce se zapisuje ve tvaru $Q : S \times A \to \mathbb{R}$. Lze ji chápat jako funkci, přiřazující míru úspěšnosti akci a v daném stavu s. Tuto Q hodnotu značíme $Q(s, a)$. Q hodnoty pro všechny možné kombinace dvojic stavů a akcí $s \times a$ jsou uloženy v předem určené tabulce.

Po každém provedení akce a_t v čase t a stavu s_t je aplikována rovnice (1) k aktualizaci příslušné Q hodnoty dané dvojice (s_t, a_t). Provedením akce a_t ve stavu s_t agent přejde do stavu s_{t+1} a obdrží odměnu r_t. Algoritmus pomocí formule $\max_a Q(s_{t+1}, a)$ počítá s maximální možnou Q hodnotou v následujícím stavu s_{t+1}, tedy předpokládá situaci, kdy se algoritmus řídí výslednou optimální strategií π^* a vybírá pouze nejoptimálnější akce. Algoritmus ale není nucen tuto akci ve stavu s_{t+1} skutečně provést, pokud se aktuálně řídí trénovací strategií π. Při aktualizaci Q hodnoty se počítá s nejlepším možným výsledkem, ke kterému daná dvojice (s_t, a_t) může vést. K aktuální Q hodnotě $Q(s_t, a_t)$ je přičten součet obdržené odměny r_t s rozdílem optimální Q hodnoty ve stavu s_{t+1} a aktuální Q hodnoty.

Míra učení α (anglicky *learning rate*) určuje důležitost nového výsledku Q hodnoty oproti staré hodnotě. Tato konstanta ovlivňuje rychlost aktualizace Q hodnot.

Diskontní faktor γ (anglicky *discount factor*) ovlivňuje přínos budoucích stavů. Tato konstanta určuje, zda se pracuje s problémem u kterého se následky akce projeví ihned nebo později (například šachy).

Celý proces učení je rozdělen do jednotlivých *učících epizod*. Algoritmus během učící epizody provádí zvolené akce a aktualizuje odpovídající Q hodnoty. Učící epizoda končí, pokud aktuální stav prostředí odpovídá cílovému stavu nebo pokud počet provedených akcí dosáhne stanoveného maximálního počtu.

Q-learning algoritmus nejprve inicializuje Q hodnoty v Q tabulce na 0, protože algoritmus dopředu nezná, jaký vliv mají akce v daných stavech. Na začátku každé učící epizody je nejprve aktuální stav s_t nastaven na zvolený počáteční stav. Poté probíhá cyklus, ve kterém se provede zvolená akce a_t, přechod do nového stavu s_{t+1} a aktualizace dané Q hodnoty $Q(s_t, a_t)$. Nová aktualizovaná Q hodnota je poté uložena v Q tabulce. Pseudokód je uveden v algoritmu 1.

Algoritmus 1 Q-learning [4]

Require: discount factor γ, learning rate α
  inicializace $Q(s, a) = 0, \forall s \in S, \forall a \in A$
  for each epizoda **do**
    inicializace počátečního stavu na stav s_t
    repeat
      výběr akce $a_t \in A(s_t)$ na základě použité strategie a hodnoty Q
      provedení akce a_t
      zaznamenání nového stavu s_{t+1} a získané odměny r_t
      $Q(s_t, a_t) = Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big)$
      $s_t := s_{t+1}$
    until s_{t+1} je cílový stav nebo počet iterací dosáhne určeného počtu iterací

2.3.1 Poměr zkoumání a užívání

Při učení se algoritmus snaží najít pro každý stav nejvhodnější akci zaručující maximální odměnu. Nejdříve je ale potřeba v daném stavu vyzkoušet větší množství akcí, aby bylo možné určit, která z nich je ta nejlepší. Tento princip průzkumu různých akcí se nazývá *zkoumání*. Přílišné zkoumání ale není prospěšné a je nutné někdy začít upřednostňovat akce generující větší odměny. Princip *užívání* zaručuje, že algoritmus v daném stavu vybírá nejvhodnější akci s maximální odměnou.

Každá trénovací strategie se snaží najít nejlepší vyvážení mezi zkoumáním a užíváním. Hledání tohoto nejlepšího poměru se nazývá *dilema zkoumání a užívání* (anglicky *Exploration Exploitation Dilemma*) a doposud nebyla nalezena jednoznačná odpověď garantující ideální poměr pro naučení algoritmu.

U Q-learning algoritmu je nejčastěji používaná trénovací strategie zvaná ϵ-greedy. Tato strategie určuje výběr prováděných akcí během učení a snaží se co nejlépe vyvažovat zkoumání a užívání. Základem této strategie je proměnná $\epsilon \in \langle 0, 1\rangle$. Tato proměnná určuje míru zkoumání oproti užívání a v průběhu učícího algoritmu je její hodnota postupně snižována. Při každém výběru akce je nejprve vygenerována náhodná hodnota v intervalu $\langle 0, 1\rangle$. Pokud je tato náhodná hodnota větší než ϵ, tak se v daném stavu provede nejoptimálnější akce s největší odměnou (největší Q hodnotou pro daný stav a akci). Pokud je náhodná hodnota menší než ϵ, je provedena náhodná akce. Tento přístup zaručuje, že na začátku algoritmu je upřednostňováno zkoumání různých akcí a v průběhu učení se při snižování hodnoty ϵ postupně přejde k užívání nejvhodnějších akcí.

3 Umělé neuronové sítě

Hluboké učení (anglicky *deep learning*) je jeden z populárních přístupů ke strojovému učení používající nelineární vícevrstvé funkce k aproximaci optimálního řešení problému. Nejčastěji je tato funkce realizována vícevrstvou umělou neuronovou sítí.

Umělá neuronová síť je matematický model skládající se z umělých neuronů. Tyto neurony jsou mezi sebou propojeny a pomocí tohoto propojení si navzájem posílají a přijímají signály. Nevýhodou takového propojení je zvýšená komplexnost při větším počtu neuronů a obtížnost interpretace jeho významu. Této nevýhodě se anglicky říká *black box* neboli černá skříňka, protože lze říci, že nevidíme jak neuronová síť uvnitř skutečně funguje a pozorujeme pouze výstupy neuronové sítě na základě daných vstupů. I přes tuto komplikaci popularita umělých neuronových sítí stále roste, zejména kvůli rozsáhlém použití neuronových sítí v oblastech klasifikace, rozpoznávání vzorů, zpracování dat a tak dále.

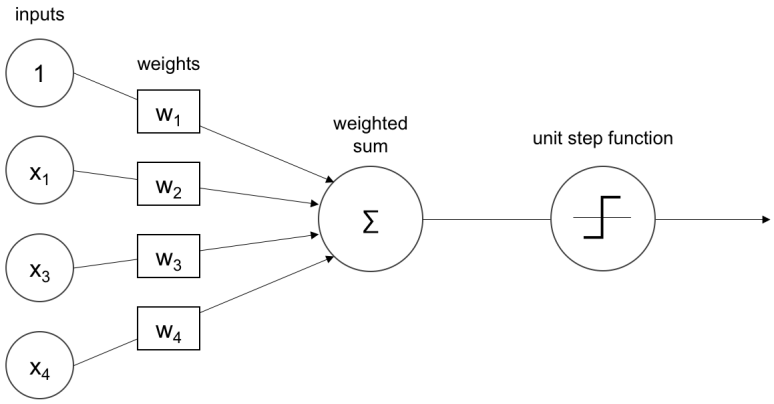
V této kapitole čerpám zejména z knihy Grokking deep learning [6] a ze studijních materiálů předmětu Umělá inteligence [7] od profesora Bělohlávka.

3.1 Perceptron

Perceptron představuje jednoduchý model neuronu. Příklad perceptronu lze vidět na obrázku 1. Perceptron je složen ze vstupů, vah, vážených součtů vstupů, prahů neuronu a výstupu. Vstupní signál x_i perceptron přijímá z daného zdroje, například z okolních neuronů nebo vstupních dat. Tento signál se může skládat z více než jedné hodnoty a každá tato hodnota má přidělenou váhu w_i. Tyto váhy ovlivňují sílu i přínos (kladný, záporný) každé vstupní hodnoty. Vážené vstupy jsou poté sečteny a tento vážený součet je porovnán s hodnotou prahu perceptronu θ. Pokud je tento součet vyšší než práh, pak výstupem perceptronu je hodnota 1, jinak je výstupem hodnota 0. Na obrázku 1 je namísto prahové hodnoty určena schodová aktivační funkce *unit step function*, která vrací 1 pokud je vstup kladný a 0 pokud je záporný. Taková aktivační funkce je ekvivalentní k nastavení prahové hodnoty na 0.

Chování perceptronu je určeno pouze váhami vstupních signálů w_i a hodnotou prahu θ. Cílem učení perceptronu je nalezení vhodných hodnot parametrů vah a prahu. Učící algoritmus je tím pádem algoritmus hledající odpovídající hodnoty parametrů.

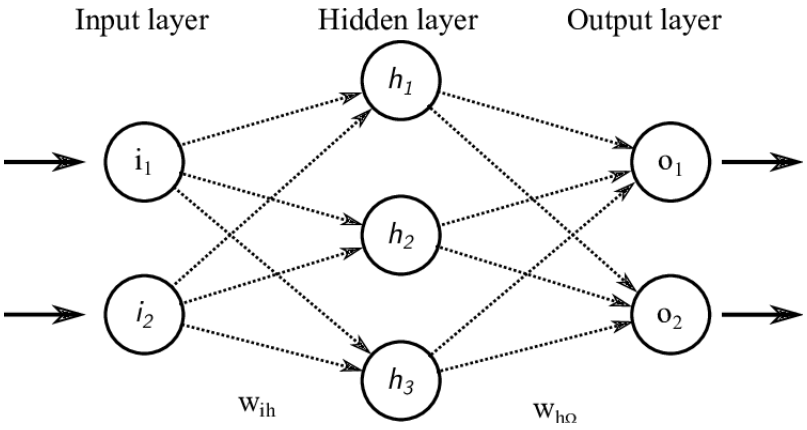
Perceptron má určité omezení a je schopen se naučit pouze lineárně separabilní vstupně-výstupní množiny. Toto omezení poté vedlo k vývoji a použití vícevrstvých neuronových sítí. Pokud je oblast učení vhodná k naučení perceptronem, pak učící algoritmy konvergují k úspěšnému výsledku v konečném čase.



Obrázek 1: Perceptron [8]

3.2 Vícevrstvá síť s dopředným šířením

Vícevrstvá síť s dopředným šířením je matematický model sloužící k aproximaci nelineární funkce. Vícevrstvé sítě mohou mít více jak jednu výstupní hodnotu. Základem této sítě je libovolné množství propojených neuronů, které jsou uspořádány ve vrstvách. Výstup každého neuronu ve vrstvě $l-1$ je napojen na vstupy všech neuronů ve vrstvě l. Každé takové propojení neuronů má nastavenou váhu w_i, která určuje sílu signálu jednoho neuronu na vstupu druhého. Síť má vstupní a výstupní vrstvu neuronů, kde velikost vstupní vrstvy určuje počet hodnot na vstupu a velikost výstupní vrstvy určuje počet výstupních hodnot. Síť má také libovolný počet takzvaných skrytých vrstev. V těchto skrytých vrstvách dochází k hlavnímu výpočtu sítě. Jejich název odkazuje na fakt, že při práci s neuronovou sítí se interaguje zejména se vstupní a výstupní vrstvou a zbylé vrstvy uvnitř sítě lze považovat za skryté.



Obrázek 2: Příklad vícevrstvé neuronové sítě [9]

Neuronová síť funguje jako více mezi sebou propojených perceptronů. Každý vstupní signál neuronu má určenou váhu, která se násobí se vstupní hodnotou. Tyto vážené vstupy jsou poté sečteny a použity na vstupu nelineární aktivační

funkce daného neuronu, která nahradila základní prahovou hodnotu definovanou u perceptronů.

Cílem učení neuronové sítě je, aby neuronová síť uměla vyjádřit podstatu a princip, na kterém je založena daná oblast učení. Je nutné správně určit architekturu neuronové sítě (počet neuronů, počet skrytých vrstev) a poté pomocí učícího algoritmu nalézt odpovídající hodnoty daných vah, prahů a strmostí aktivační funkce. Velikost dané neuronové sítě je ovlivněna složitostí problému. Pokud je problém na danou síť příliš složitý a síť má problém s učením, pak je vhodné zvážit přidání další vrstvy nebo zvýšení počtu neuronů ve vrstvách. Naopak pokud se síť učí problém příliš rychle a snadno, je vhodné zkusit učení na menší síti, protože je možné, že dochází k *přeučení* a síť pouze vrací vypozorované vzory ze vstupních dat a nezná princip daného problému. Cílem učení neuronové sítě by také měla být schopnost vracet správné výsledky i na vstupy, které síť nikdy předtím nedostala. Tomuto principu se říká *generalizace*.

3.3 Aktivační funkce

U perceptronu byly možné výsledné hodnoty pouze 0 nebo 1, protože perceptron používá k výpočtu výsledku pouze hodnotu prahu (skoková aktivační funkce). Aktivační funkce neuronů neobsahují pouze prahovou hodnotu, ale také strmost funkce. Obor hodnot takových funkcí je dynamičtější a neuron není omezen pouze na odpovědi ANO (hodnota 1) a NE (hodnota 0). V této podkapitole čerpám zejména ze zdroje [10], který obsahuje přehled používaných aktivačních funkcí.

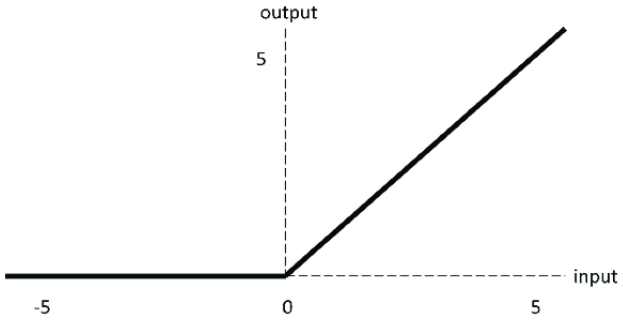
Při použití metody gradientního sestupu (popsáno níže v kapitole 3.5) pracujeme s derivací aktivační funkce neuronu. Zavedení těchto funkcí je důležité, protože u skokové funkce není definovaná derivace v bodu prahové hodnoty a tedy nelze s funkcí pracovat.

Lineární aktivační funkce $f(x) = c \cdot x$ vrací výsledek úměrný vstupní hodnotě (v případě neuronu součet vážených vstupů). Obor hodnot této funkce je $(-\infty, \infty)$. Derivace této funkce je konstantní hodnota c a gradient tedy není nijak ovlivněný vstupní hodnotou funkce. Více vrstev neuronů obsahujících lineární aktivační funkce lze nahradit pouze jednou vrstvou, protože spojení lineárních funkcí lze z jejich podstaty nahradit jednou lineární funkcí. Lineární aktivační funkce je často využívaná ve výstupní vrstvě neuronové sítě, kde tyto hodnoty reprezentují výstup celé neuronové sítě [11]. V tomto případě často bývá hodnota c nastavena na 1.

Sigmoidální přenosová funkce $f(x) = \frac{1}{1+e^{-x}}$ je populární nelineární funkce, která je velmi citlivá na změny vstupních hodnot v intervalu $(-2, 2)$. Obor hodnot této funkce je $(0, 1)$, což vhodně omezuje výsledné hodnoty neuronů. Vzhledem k tomu, že je tato funkce nelineární, tak výsledkem kombinace více těchto funkcí je opět nelineární funkce. Tato funkce je vhodná k použití ve skrytých vrstvách za sebou.

Usměrněná lineární funkce (ReLU, viz obrázek 3) $f(x) = \max(0, x)$ je také používána ve skrytých vrstvách neuronové sítě. Tato nelineární funkce je v sou-

časné době jednou z nejpoužívanějších a je populární zejména díky své úspěšnosti při učení. ReLU, se stanoveným oborem hodnot $[0, \infty)$, neomezuje vysoké hodnoty číslem 1. Je také responzivní na celém svém kladném intervalu, narozdíl od funkce sigmoidální, která je responzivní zejména na menší změny okolo intervalu $(-2, 2)$. ReLu funkce je výpočetně jednodušší oproti sigmoidální funkci. Neurony s negativní vstupní hodnotou vracejí hodnotu 0, což je přínosné, protože je často nežádoucí, aby se tyto neurony aktivovaly (vracely pozitivní hodnotu). Hlavní nevýhodou ReLU funkce je možná narůstající pasivita určitých neuronů, které nebyly aktivovány (hodnota funkce byla nulová) a neodpovídají tedy na změny chyb, protože jejich gradient je nulový. Existují variace ReLU funkce, které se snaží tento problém eliminovat.



Obrázek 3: Graf aktivační funkce ReLU [12]

3.4 Chyba neuronové sítě

Při hodnocení celkové úspěšnosti neuronové sítě i při samotném učení je nutné definovat míru chybovosti. Tato chybovost je reprezentována takzvanou *střední kvadratickou chybou* (anglicky *mean squared error*). Čistá chyba (anglicky *pure error*) je definována jako rozdíl mezi správným očekávaným výsledkem k danému vstupu a skutečným výsledkem neuronové sítě. Ke zdůraznění větších chyb a zároveň ke snížení důrazu méně odlišných výstupů je tento rozdíl umocněn na druhou. Umocnění také zapříčiní, že hodnota chyby je vždy kladná. Taková chyba pouze sděluje, o jak velkou hodnotu se neuronová síť mýlila, ale nesděluje, zda je výsledek neuronové sítě větší či menší v porovnání s očekávaným výsledkem. Omezení na pozitivní hodnoty chyby je důležité, protože při práci s neuronovou sítí obsahující více výstupů se počítá s hodnotou průměrné chyby ze všech výstupních neuronů. Pokud by jeden výstup měl chybu -100 a druhý výstup chybu 100, pak by výsledná průměrná chyba byla 0, což není žádaný výsledek.

Střední kvadratická chyba neuronové sítě s jedním výstupním neuronem vzhledem ke vzoru p je definována jako $E_p = \frac{1}{2}(y(x^p) - o^p)^2$, kde $y(x^p)$ je výstup neuronové sítě pro vstup x^p a o^p je hodnota požadovaného výstupu získaná z trénovací množiny T.

Proces učení je možné sledovat na rozdílech v celkových chybách neuronové sítě. Také je možné ve změně chybovosti sítě sledovat vliv vybraného parametru. Cílem učícího algoritmu je minimalizace chyby neuronové sítě.

3.5 Backpropagation

Algoritmus *backpropagation* je oblíbený učící algoritmus. Proces učení neuronové sítě je možné považovat za minimalizaci chyby sítě pomocí úpravy parametrů vah, prahů a strmostí aktivační funkce. Nejvíce jsou v procesu učení měněny hodnoty vah.

Změna váhy w v čase $t+1$ je definována jako $w(t+1) = w(t) - \alpha \nabla E(w(t))$ kde α reprezentuje míru učení. Hodnota váhy $w(t+1)$ by měla zapříčinit menší hodnotu chybové funkce E. Klíčový je výpočet $\nabla E(w(t))$, reprezentující hodnotu gradientu chybové funkce E. Tento gradient lze vyjádřit jako $\frac{\partial E}{\partial w}$, protože se snažíme nalézt takové hodnoty vah w, aby chybová funkce neuronové sítě byla minimální.

Gradient funkce $f : \mathbb{R}^k \to \mathbb{R}$ v bodě a je vektor $\nabla f(a)$, který reprezentuje směr největšího růstu dané funkce f v a. Opačná hodnota k $\nabla f(a)$ je směr největšího klesání f v a. K nalezení minima dané funkce začneme v libovolném bodě a a postupně přecházíme do bodů, které leží ve směru největšího klesání funkce. Takto pokračujeme, dokud se nebude aktuální bod nacházet v minimu dané funkce f. Tento princip se nazývá *metoda gradientního sestupu* a je využíván k výpočtu minimalizace chyby neuronové sítě.

Gradient funkce f v bodě a lze zapsat jako vektor parciálních derivací v bodě a.

$$\nabla f(a) = \left\langle \frac{\partial f}{\partial x_1}(a), \ldots, \frac{\partial f}{\partial x_n}(a) \right\rangle$$

Metoda backpropagation pomocí gradientního sestupu postupně upravuje hodnoty parametrů neuronové sítě s cílem minimalizace chyby. Backpropagation končí pokud síť dosáhla předem stanovené minimální přesnosti nebo po provedení maximálního počtu povolených iterací. U této metody je možné, že použití gradientního sestupu nalezne pouze lokální minimum funkce chyby E.

4 Deep Q-learning algoritmus

Největším problémem reinforcement learning algoritmů je velmi rychle narůstající výpočetní a paměťová složitost v komplexnějších prostředích. Například Q-learning algoritmus musí ukládat do tabulky Q hodnoty všech možných dvojic *(stav, akce)* a při větším počtu stavů a akcí je práce s touto tabulkou nepraktická. Pokud máme například prostředí s 1000 stavy a 1000 akcemi, pak by bylo potřeba uložit milion Q hodnot. Takovéto prostředí by se ještě dalo považovat za jedno z jednodušších v porovnání se hrou *Go* která má 10^{170} stavů anebo *Star Craft 2* s 10^{270} stavy [13]. K oběma těmto hrám existují varianty deep Q-learning algoritmů, které je hrají na profesionální úrovni.

Řešení problému reinforcement learning přístupu spočívá v hledání aproximátoru takového komplexního prostředí. U Q-learning algoritmu hledáme funkci $Q_\Theta(s, a)$, která je schopna aproximovat Q hodnotu jakékoliv dvojice *(stav, akce)* s pomocí vektoru parametrů θ. Tento přístup se nazývá *Approximate Q-learning* [14].

Největšími průkopníky v této oblasti se stala společnost DeepMind, která v roce 2013 vydala článek [15] přinášející nový typ algoritmu zvaný Deep Q-learning. Bylo to poprvé, kdy se k aproximování Q hodnoty použila umělá neuronová síť (nazývaná *Deep Q-network*). Tento algoritmus propojuje Q-learning algoritmus s konvoluční neuronovou sítí a úspěšně se naučil hrát 49 klasických Atari her a 22 z nich je schopen hrát lépe než profesionální lidský hráč. Jedinými vstupy agenta byly pixelové reprezentace aktuálního obrazu hry a bodové skóre hry.

Aktuálně nejzajímavější objev v deep reinforcement learning odvětví je popsán v článcích [16, 17], opět od společnosti DeepMind, zabývajících se použitím deep reinforcement learning algoritmu k ovládání jaderné fúze v tokamaku[1]. Tento objev má nezanedbatelný vliv na pokrok v oblasti udržitelné energie.

Vzhledem ke složitosti prostředí pro navigaci multifunkčního nakladače jsem se rozhodla v této práci použít deep Q-learning algoritmus. Princip tohoto algoritmu je popsán v následující podkapitole, kde jsem čerpala zejména ze zdroje [14].

4.1 Princip deep Q-learning algoritmu

Deep Q-learning algoritmus používá neuronovou síť, která má na vstupu aktuální stav prostředí a na výstupu jsou Q hodnoty všech akcí agenta. Tento přístup je rozdílný od klasického Q-learning algoritmu, kde je vstupem dvojice *(stav, akce)* a Q-tabulka vrací odpovídající Q hodnotu.

Učící algoritmus neuronových sítí backpropagation (viz 3.5) pracuje s chybovou funkcí E, která se v základu skládá z rozdílu mezi cílovou hodnotou a výsledkem neuronové sítě. Reinforcement learning ale nemá dopředu stanovenou cílovou hodnotu, jako je tomu u učení s učitelem (anglicky supervised learning).

[1] „Zařízení vytvářející toroidální magnetické pole, používané jako magnetická nádoba pro uchovávání vysokoteplotního plazmatu. Dnes se tokamaky považují za jednu z nejnadějnějších cest k realizaci řízené jaderné fúze.“ [18]

Je třeba zadefinovat cílovou hodnotu Q_{target} sloužící k učení neuronové sítě. Funkce výpočtu cílové Q hodnoty je definována velice podobně jako Q funkce (1). Hodnotu odměny r získáme provedením dané akce a ve stavu s. Maximální očekávanou Q hodnotu Q_θ obdržíme při použití neuronové sítě se vstupem následujícího stavu s' a výběrem nejvyšší hodnoty z výsledků sítě. Z tohoto důvodu neuronová síť vrací Q hodnoty pro všechny akce. To velice zjednodušuje hledání maximální očekávané Q hodnoty.

$$Q_{target}(s, a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a') \tag{2}$$

Deep Q-learning, stejně jako klasický Q-learning, může používat ϵ-greedy strategii k vyvážení poměru zkoumání a užívání. Pokud je náhodná proměnná větší než ϵ, tak je vybrána akce s nejvyšší predikovanou Q hodnotou, jinak je vybrána náhodná akce.

Bylo zjištěno, že účinnost neuronové sítě je negativně ovlivněna korelací mezi vstupními daty. U deep Q-learning algoritmu jsou vstupními daty stavy prostředí a tyto stavy jsou mezi sebou silně závislé. K redukci této korelace se ukládají všechny kroky *(stav, akce, následující stav)* společně s hodnotou odměny do experience bufferu a neuronová síť je trénována na náhodných vzorcích stejné velikosti z tohoto bufferu. Před začátkem učení je nejprve nutné provést určitý počet kroků, aby se alespoň zčásti zaplnil buffer k výběru trénovacího vzorku.

Při trénování neuronové sítě v deep Q-learning algoritmu je nejprve vybrán náhodný vzorek z bufferu. Tento vzorek obsahuje čtveřici *(stav, akce, následující stav, odměna)*. Na základě hodnoty následujícího stavu použijeme neuronovou síť pro výpočet maximální nadcházející Q hodnoty. Spolu s obdrženou odměnou lze pomocí rovnice (2) vypočítat cílovou Q hodnotu pro dvojice *(stav, akce)*. Poté se získá Q hodnota $(stav, akce)$ z neuronové sítě. Ze získané a cílové Q hodnoty lze vypočítat chybu neuronové sítě pomocí střední kvadratické chyby a následně aplikovat metodu gradientního sestupu k minimalizaci chyby.

4.2 Dvě neuronové sítě

V základním deep Q-learning algoritmu, který je popsán výše, je použita jedna neuronová síť k predikci Q-hodnot i k výpočtu cílových Q-hodnot. Cílem učení sítě je, aby predikovaná hodnota byla co nejblíže požadované hodnotě. K dosažení tohoto cíle jsou měněny proměnné neuronové sítě. Tyto úpravy proměnných nezmění pouze predikovanou hodnotu, ale i cílovou hodnotu, ke které se ta predikovaná snaží dostat. To poté vede k nestabilitě neuronové sítě, která se může chovat nepředvídatelně.

Řešení se objevilo v publikaci [19] od společnosti DeepMind. Je zavedena druhá neuronová síť zvaná *Target network*. Tato síť je inicializována jako kopie hlavní sítě a je používána pouze při výpočtu cílové Q hodnoty pro získání maximální nadcházející Q hodnoty. Váhy této sítě se mění pouze po určitém počtu kroků, aby se předešlo příliš časté změně cílové Q hodnoty. Při aktualizaci vah se

pouze zkopírují váhy z hlavní sítě. Hlavní neuronová síť slouží k výběru aktuálně vykonávané akce a provádí aktualizaci vah po každém kroku.

5 Hardwarové vybavení

Ve spolupráci s firmou Technotrade byl nejprve vybrán nakladač používající princip *kloubového zatáčení* (anglicky *articulated steering*). V průběhu spolupráce byl kloubový nakladač nahrazen nakladačem používajícím *smykové řízení* (anglicky *skid steering*).

Dále je v této diplomové práci popsána zejména implementace virtuální simulace samotného nakladače a jeho prostředí spolu s deep Q-learning algoritmem. Při plánované budoucí implementaci učícího algoritmu s fyzickým nakladačem bude nakladač připojen k přenosnému počítači nVidia Jetson Nano [20], který bude obstarávat komunikaci mezi deep Q-learning algoritmem a ovládáním samotného nakladače.

5.1 Nakladač s kloubovým zatáčením

Multifunkční nakladač *Dapper 5000* (viz obrázek 4) používá k pohybu kloubové řízení, které při zatáčení otáčí celou polovinou stroje. Díky tomu je stroj schopný zatáčet i na místě.

Nakladač je ovládaný prostřednictvím joysticku. Naklonění joysticku ve směru horizontální osy určuje míru zatočení stroje, na vertikální ose rychlost a směr jízdy (dopředu, dozadu). Míra naklonění joysticku je v simulaci reprezentována intervalem $[-100,00\%,\ 100,00\%]$. Pro zjednodušení učení deep Q-learning algoritmu je míra zatočení nakladače (horizontální osa joysticku) reprezentována hodnotou odpovídajícího úhlu, kde maximální úhel natočení $35°$ reprezentuje hodnotu $100,00\%$ naklonění joysticku. Agentovi také není umožněno nastavit jakoukoliv hodnotu z intervalu $[-35°, 35°]$, ale má k dispozici pět možných hodnot úhlů $\{7°, 14°, 21°, 28°, 35°\}$ společně s odpovídajícími zápornými hodnotami a neutrálním úhlem $0°$. Toto omezení možných hodnot úhlů zatočení výrazně zjednodušuje výpočet matematické reprezentace pohybu (více v sekci 5.1.1) a také samotný proces učení. Maximální rychlost nakladače je z bezpečnostních důvodů omezena na $10\,km/h$ a agent má k dispozici rychlosti $\{-3, 0, 3, 7, 9, 10\}$.

5.1.1 Matematická reprezentace pohybu

Vzdálenosti mezi středem os kol a otáčecím kloubem jsou v případě nakladače Dapper 5000 rozdílné (kloub se nenachází zcela uprostřed nakladače), což znamená, že má při zatáčení dva různé poloměry kružnic otáčení. Jelikož je rozdíl ve vzdálenostech pouhých $10\,cm$, je pro zjednodušení problému ignorován a výpočet je prováděn pouze s jednou kružnicí otáčení.

Vzhledem k tomu, že má agent k dispozici pět hodnot úhlů zatočení na každou stranu, bylo možné poloměry těchto pěti kružnic otáčení vypočítat dopředu, před zahájením učícího procesu, bez nutnosti opakovaného výpočtu poloměrů.

Geometrie zatáčení je naznačena v obrázku 5. Na obrázku je zobrazen nakladač s nesymetrickým umístěním otáčecího kloubu (vzdálenosti l_1, l_2) a naznačené poloměry kružnic otáčení r_1 a r_2. V případě symetrického umístění kloubu

Obrázek 4: Mlutifunkční nakladač Dapper 5000 [21]

platí $r_1 = r_2$. Bod O reprezentuje střed kružnice otáčení, který lze získat jako průsečík polopřímek kolmých k osám kol. Úhel γ reprezentuje *úhel natočení* nakladače. Úhel θ určuje *směrový úhel* (anglicky *heading angle*), který reprezentuje natočení nakladače vzhledem k ose souřadnic (například vzhledem k ose x).

Rovnice k výpočtu pohybu a souřadnic nakladače s kloubovým řízením jsou převzaty z publikace [22]. Výpočet poloměru kružnice otáčení je popsán v rovnici (3), kde l je vzdálenost osy kol k otáčecímu kloubu a γ určuje úhel natočení. Úhlová rychlost ω je vyjádřena jako podíl mezi rychlostí nakladače a poloměrem kružnice otáčení $\omega = \frac{v}{r}$. Rovnice (4) definuje výpočet nových souřadnic x', y' a nového směrového úhlu θ' z původních hodnot x, y a θ, kde časový rozdíl mezi těmito souřadnicemi je obsažen v Δt. Rovnice z publikace [22] obsahuje chybu ve výpočtu souřadnice x', obsahující namísto první operace sčítání operaci odčítání.

$$r = \frac{l + \frac{l}{\cos\gamma}}{\tan\gamma} \tag{3}$$

$$\begin{aligned} x' &= \cos(\omega\Delta t) \cdot r \cdot \sin(\theta) + \sin(\omega\Delta t) \cdot r \cdot \cos(\theta) - r \cdot \sin(\theta) + x \\ y' &= \sin(\omega\Delta t) \cdot r \cdot \sin(\theta) - \cos(\omega\Delta t) \cdot r \cdot \cos(\theta) + r \cdot \cos(\theta) + y \\ \theta' &= \omega \cdot \Delta t + \theta \end{aligned} \tag{4}$$

Pokud je úhel natočení γ roven nule, pak by v rovnici (3) docházelo k dělení nulou. Nakladač v takovém případě směřuje rovně, nikam nezatáčí a nemáme určenou žádnou kružnici otáčení ke které by se počítal poloměr. Při implementaci je tato situace vyřešena nahrazením rovnic (3) a (4) rovnicí pro výpočet pohybu po přímce.

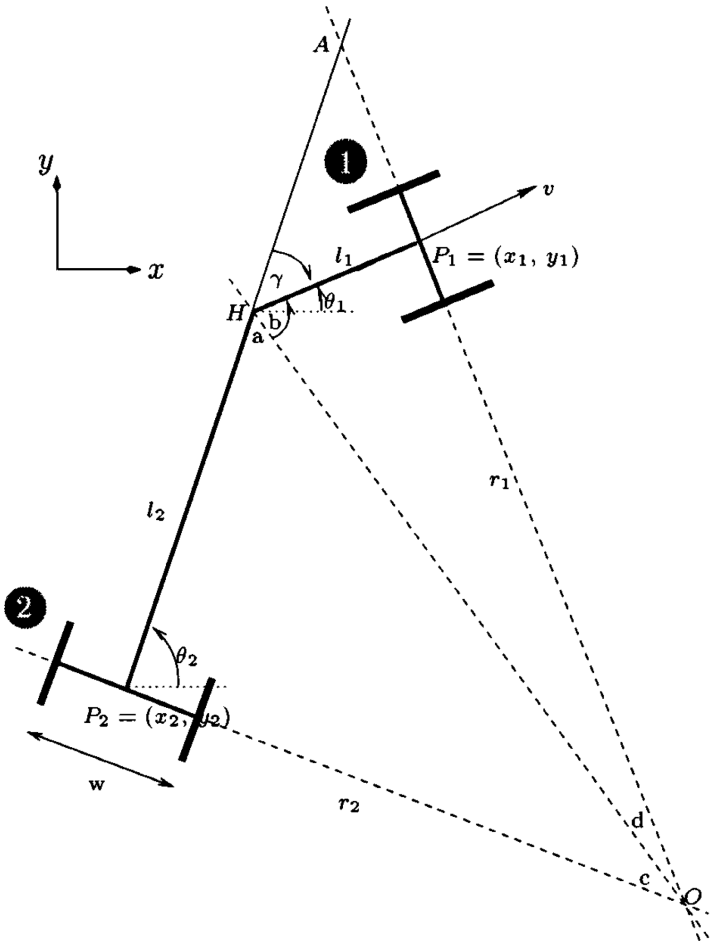
5.1.2 Důvod změny stroje

Z důvodu problému s dostupností stroje Dapper 5000 a níže popsaných komplikací došlo po domluvě s firmou Technotrade ke změně stroje. Navrhovaný stroj používá jiný typ pohybu, který je podrobněji popsaný v následující podkapitole. Veškeré poznatky a implementace uvedené v této kapitole jsou v případě zájmu firmy použitelné.

Při implementaci simulace nakladače s kloubovým řízením je nutné pro správnou funkčnost senzorů vzdálenosti nakladač reprezentovat pomocí dvou obdélníků (obrázek 6). Použité vzorce (4) však vypočítávají pouze středové souřadnice celého nakladače. Pro výpočet dvou souřadnic je potřeba najít odpovídající vztah mezi těmito souřadnicemi a vypočítaným středem.

5.2 Nakladač se smykovým řízením

Smykový (anglicky *skid-steer*) nakladač pro pohyb využívá princip smyku. Pohyb a směr nakladače určuje míra pohonu levé a pravé části nezávisle na sobě. Tyto nakladače mohou být kolové nebo i pásové a vynikají zejména svou schopností pohybu na malém prostoru. V této diplomové práci jsem měla k dispozici



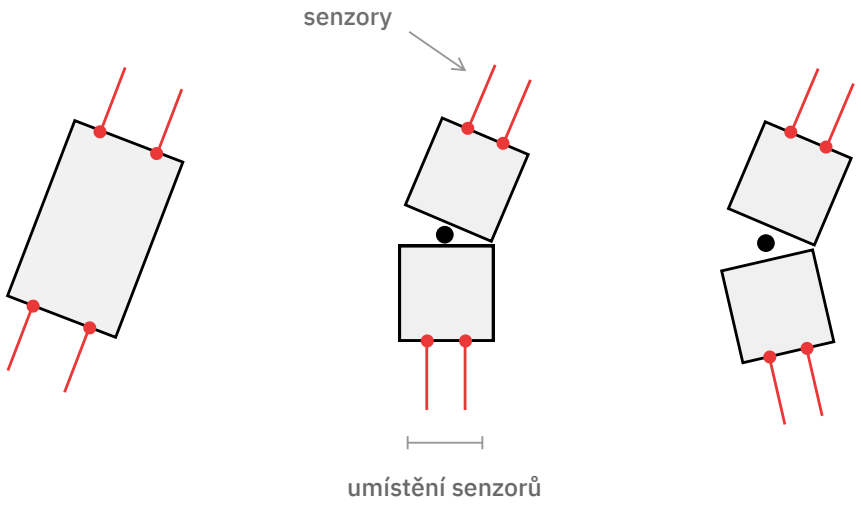
Obrázek 5: Geometrie zatáčení kloubovým řízením [23]

pásový nakladač Kovaco MiniZ (viz obrázek 7), jehož pohyb je podobný pohybu vojenských tanků.

Smykový způsob pohybu umožňuje různé způsoby zatáčení, kde je ostrost daného zatočení určena rozdílem rychlostí mezi jednotlivými stranami. Čím více jsou odlišné rychlosti stran, tím prudčeji stroj zatočí. Pro jemné zatočení je možné pouze zpomalit jednu stranu oproti té druhé. Většího zatočení dosáhneme celkovým zastavením jedné strany a prudšího zatočení docílíme invertováním pohybu jedné strany vůči té druhé. Při pohybu dopředu a dozadu je nastavena stejná rychlost na levé i pravé strany nakladače.

Pohyb nakladače je reprezentován pomocí dvojice hodnot, určujících rychlost a směr levého a pravého pásu. Opět je počet rychlostních hodnot v simulaci omezen z důvodu snížení výpočetní náročnosti. Nakladač má k dispozici rychlostní hodnoty z množiny $\{-0{,}56; -0,28; -0,14; 0; 0,14; 0,28; 0,56\}$.

Dále se v této práci budu věnovat zejména nakladači se smykovým řízením, ke kterému je navržena celá implementace deep reinforcement learning algoritmu.



Obrázek 6: Zkreslené umístění zadních senzorů kloubového nakladače

5.2.1 Matematická reprezentace pohybu

Smykové řízení lze, narozdíl od kloubového řízení, reprezentovat pouze jedním obdélníkovým tvarem a samotný princip pohybu je tak jednodušší. Tento typ řízení je ale velmi ovlivněn mírou prokluzu pásů. Prokluzování nelze dopředu předpovědět, protože záleží na několika faktorech, jako například typ a nerovnost povrchu, po kterém se nakladač pohybuje, ale i těžiště samotného nakladače (zda převáží nakladač náklad, nebo je prázdný). Mnoho matematických výpočtů smykového řízení se spoléhá i na senzory měřící aktuální míru prokluzování pásů stroje (například gyroskopický senzor) [25]. Pokud tyto hodnoty prokluzu nejsou známy, je potřeba počítat s tím, že reálná aplikace výpočtu se může ve větší míře lišit od virtuální simulace.

Při studiu a implementaci matematického modelu smykového řízení jsem nejprve použila znalosti z článku [25]. Rovnice (5) z tohoto článku popisuje výpočet nových souřadnic x', y' a směrového úhlu θ', kde v reprezentuje celkovou rychlost nakladače a její výpočet je popsán rovnicí (6), kde proměnné v_r a v_l reprezentují rychlost pravého a levého pásu nakladače.

$$\begin{aligned} x' &= \sin(\theta) \cdot v \ + x \\ y' &= \cos(\theta) \cdot v \ + y \\ \theta' &= \omega + \theta \end{aligned} \tag{5}$$

$$v \ = \frac{v_r + v_l}{2} \tag{6}$$

Výpočet úhlové rychlosti ω popisuje rovnice definovaná níže. Hodnota

Obrázek 7: Kovaco MiniZ [24]

proměnné *rozchod* je vzdálenost středu levého pásu od středu pravého pásu.

$$\omega = \frac{v_r - v_l}{rozchod} \tag{7}$$

Při testování simulace, využívající tyto rovnice k výpočtu, se model při zatáčení pohyboval po kružnici, avšak míra rotace byla extrémně vysoká. Tento styl pohybu se zdál nepřirozený a nesprávný. Další akademické články popisující matematický model smykového řízení zahrnovaly ve svých výpočtech i znalost míry prokluzu pásů, a taková informace není ve virtuální simulaci k dispozici. Článek [26] také počítá se znalostí míry prokluzování pásů, ale výpočet je rozdělen do více částí, takže pomocí určitých úprav lze tento kinematický model použít bez znalosti hodnot prokluzu. Výsledné rovnice jsou podobné rovnicím (5), hlavním rozdílem je ale použití goniometrické funkce cos u výpočtu souřadnice x' a goniometrické funkce sin při výpočtu y'. Po prohození těchto goniometrických funkcí, simulace odpovídá pohybu smykového řízení pásového nakladače a rovnice (5) tedy obsahuje chybu. Správný vzorec k výpočtu nových souřadnic je obsažen v rovnici definované níže.

$$\begin{aligned} x' &= \cos(\theta) \cdot v \ + x \\ y' &= \sin(\theta) \cdot v \ + y \\ \theta' &= \omega + \theta \end{aligned} \tag{8}$$

5.3 nVidia Jetson Nano

Sada pro vývojáře nVidia Jetson Nano je přenosná kompaktní výpočetní jednotka, navržená pro běh a učení neuronových sítí. Součástí je microSD karta, která obsahuje operační systém Linux spolu s ovladači a knihovnami potřebnými k práci s neuronovými sítěmi. [20]

Tento počítač od firmy nVidia není používaný k hlavní fázi učení deep reinforcement algoritmu, ta probíhá na serveru Katedry informatiky Univerzity Palackého, ale pouze na doučování samotného nakladače v terénu a překladu pohybových příkazů učícího algoritmu do instrukcí ovládajících nakladač. Nakladač je k nVidii připojen pomocí CAN sběrnice. Překlad instrukcí z jazyka Python poskytuje knihovna CANlib SDK [27] od firmy Kvaser, která vyvíjí CAN převodníky používané firmou Technotrade.

Na nVidii byly nainstalovány všechny potřebné knihovny k deep Q-learning algoritmu a samotné simulaci. Pro připojení k zařízení byl využíván příkaz *screen* a Micro-USB konektor [28]. Je tedy možné na zařízení pouštět nejen deep Q-learning algoritmus, ale i vizuální reprezentaci virtuální simulace nakladače.

5.4 Senzorové vybavení nakladače

V této podkapitole je popsáno plánované senzorové vybavení nakladače a očekávaný způsob práce s těmito snímači.

K úspěšnému automatizovanému pohybu ve volném prostoru bez kolizí je nutné, aby nakladač obsahoval senzory vzdálenosti. Měření těchto senzorů je poté posíláno po CAN sběrnici deep Q-learning modelu, který na základě těchto hodnot rozhodne o dalším pohybu. Jednou z možností je použití jednoho LiDAR senzoru umístěného na střeše nakladače, snímajícího okolí stroje v rozsahu 360°. Další možností je využití více ultrazvukových snímačů na všech stranách nakladače. Při práci na simulaci jsem použila čtyři ultrazvukové snímače, umístěné na každé straně nakladače.

Algoritmus také potřebuje získávat informace o pozici nakladače v předem definované pohybové mřížce. Model počítá s jedním GPS lokátorem, který určuje pozici nakladače. Tento lokátor ale není velmi citlivý na menší změny polohy stroje. Při běhu nakladače bude pozice získávána z vypočtu simulace. Jednou za určitý počet kroků nebo po ujetí určité vzdálenosti, bude tato pozice porovnána s hodnotou GPS lokátoru a poté případně upravena.

6 Simulace prostředí

Tato kapitola se zabývá popisem použitých knihoven a principů sloužících k zprovoznění virtuální simulace pohybu nakladače v předem určeném nezmapovaném prostředí. Také je popsána implementace vizuální simulace, která pomocí knihovny Pygame graficky zobrazuje pohyb nakladače, pozice překážek a stavy senzorů.

6.1 Knihovna pygame

Pygame je knihovna pro jazyk Python, sloužící zejména k tvorbě videoher a grafických vizualizací [29]. Knihovna obsahuje obsáhlou dokumentaci zahrnující příklady použití jednotlivých funkcí. Knihovna je používána zejména v souboru `simulation.py`, kde realizuje celkové zobrazení vizuální simulace pohybu nakladače.

6.1.1 Popis souboru *simulation.py*

Soubor `simulation.py` se skládá z více zadefinovaných funkcí, které uvnitř používají funkce knihovny pygame. Funkce `initialize_screen()` vytváří pomocí pygame funkce `pygame.display.set_mode()` okno o zadaných rozměrech, poté se volá funkce `generate_visual_grid()`, která do okna vykreslí mřížku prostoru po které se může nakladač pohybovat. Mřížka je vykreslena uvnitř pygame objektu `Surface`, který vytváří povrch, na kterém lze zobrazit jakýkoliv tvar nebo obrázek. Mřížka odpovídá rozměrům $1m \times 1m$ a vzhledem k přehlednější vizuální reprezentaci je zavedena konstanta `SCALING_COEFICIENT`, jenž násobí všechny souřadnice před jejich vizuálním zobrazením.

Funkce `update_visualization()` nejprve vymaže obsah stávajícího okna a zakreslí prostorovou mřížku. Poté vynásobí všechny body obdélníku nakladače konstantou `SCALING_COEFICIENT` a výsledný seznam bodů předá funkci `pygame.draw.polygon()` k vykreslení. Podobný proces se provede i u senzorů a překážek, pouze je zde přidáno různé barevné označení, rozlišující odlišné stavy těchto objektů (například stav kdy senzor detekuje překážku).

6.1.2 Omezení knihovny Pygame

Během práce s knihovnou pygame jsem narazila na několik nedostatků a omezení knihovny. Pokud byl nakladač reprezentován pomocí objektu `Rectangle`, tak jej nebylo možné otáčet o daný úhel a bylo nutné objekt nejprve vykreslit od objektu `Surface`, který je následně možné otáčet pomocí funkce `pygame.transform.rotate()`.

Dalším omezením knihovny pygame bylo samotné otáčení objektu `Surface`, které je možné pouze kolem levého horního rohu objektu. Pro správné vykreslení nakladače bylo potřeba otočit obdélník kolem středu nakladače na hodnotu směrového úhlu θ. Pro takovou rotaci bylo potřeba implementovat pomocnou funkci

`convert_center_to_topleft()`, která převádí středové souřadnice nakladače na souřadnice levého horního rohu. Celkový proces otočení vizuální reprezentace nakladače je popsán ve zdrojovém kódu 1. Řešení tohoto problému jsem čerpala ze zdroje [30].

```
1 topleft_position = convert_center_to_topleft(position)
2
3 rotated_loader = pygame.transform.rotate(loader_image,heading_angle)
4 new_rect = rotated_loader.get_rect(
5   center=loader_image.get_rect(topleft=topleft_position).center)
6
7 screen.blit(rotated_loader, new_rect.topleft)
```

Zdrojový kód 1: Otočení objektu Surface obsahující obdélník nakladače

Tyto přístupy zobrazení obdélníku a zejména jeho otáčení ale nebyly velmi efektivní. Finálně objekt `Loader`, obsažený v souboru `loader.py`, obsahuje atribut `polygon`, který reprezentuje pozici nakladače pomocí obdélníku. Samotná simulace pouze vykresluje body tohoto polygonu. Stejný přístup je používán i k vykreslování senzorů a překážek.

6.2 Mřížka prostoru

Jak už bylo zmíněno výše, nakladač se pohybuje po předem definované mřížce, která slouží k navigaci nakladače v prostoru. Jelikož je poloha nakladače reprezentována jako souřadnice políčka, ve kterém se nakladač nachází, neposkytuje mřížka přesnou polohu nakladače, ale pouze její hrubý odhad. Míra nepřesnosti takovéto lokalizace závisí na velikosti jednotlivých políček. Čím je větší políčko, tím je obecnější poloha nakladače. V této implementaci má jedno políčko rozměry $1m \times 1m$ a mřížka má velikost 10×10 políček.

Během učení deep Q-learning algoritmu je pozice nakladače reprezentována pomocí souřadnice políčka mřížky, tedy jako dvojice celočíselných hodnot. Souřadnice daného políčka je získána z přesné pozice simulace zaokrouhlením na celá čísla. V simulaci je tedy stále uložena i přesná pozice nakladače, která je poté využita například k vizualizaci.

Cílem deep Q-learning algoritmu tedy není dostat střed nakladače na cílovou pozici, ale kvůli zaokrouhlování pozice nakladače je pouze nutné, aby nakladač dojel na políčko mřížky určené cílovou pozicí. Vzhledem k možným nepřesnostem simulace, kvůli materiálu a nerovnosti terénu, je žádané, aby algoritmus počítal s nějakou odchylkou v cílové souřadnici.

6.3 Implementace překážek

Překážky jsou reprezentovány pomocí kruhu, který je zadán souřadnicemi středu a poloměrem. Všechny tyto překážky jsou uloženy v třídě `Loader` pomocí se-

znamu obsahujícího souřadnice jejich středů. Konstanta `OBSTACLE_RADIUS` s hodnotou $0,5$ reprezentuje hodnotu poloměru kruhů překážek. Jedna překážka má tedy v průměru $1m$ a zabírá jedno políčko mřížky.

Samotné překážky jsou poté vygenerovány funkcí `generate_obstacles()`, která pomocí souřadnic středů a poloměru vytvoří instance třídy `Circle` z knihovny `Collision`.

6.4 Implementace senzorů

Veškerá implementace senzorů vzdálenosti je obsažena v souboru `sensor.py`, který obsahuje definici třídy `Sensor`. Tato třída simuluje funkci ultrazvukových snímačů.

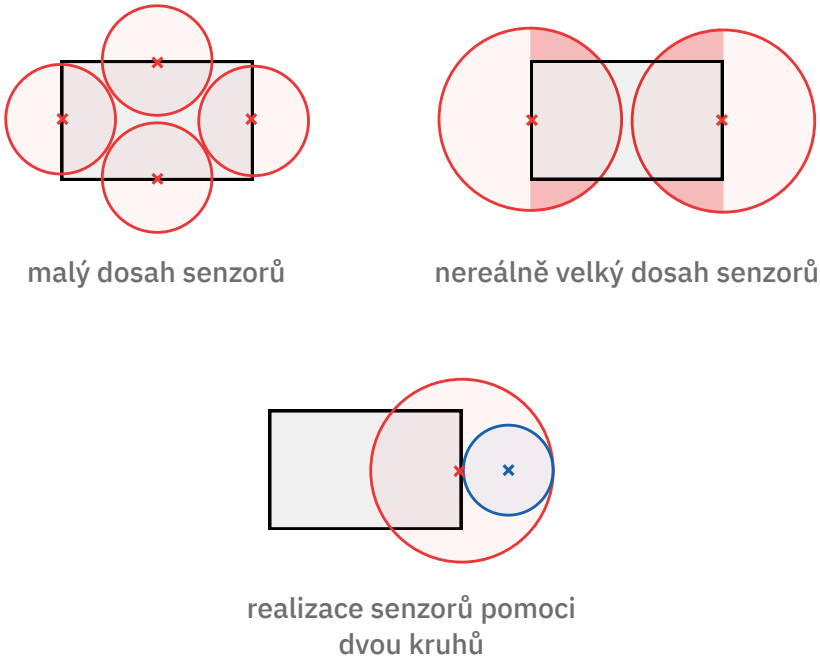
Při vytváření instance této třídy je předáván argument `sensor_position`, který pomocí textového řetězce určuje, na které straně nakladače se daný senzor nachází. Na základě této pozice se provede nastavení atributů `operator` a `additional_right_angle`, které slouží k výpočtu umístění senzoru. Pomocí těchto atributů se provede posunutí souřadnic ze středu nakladače. Například při výpočtu umístění předního senzoru je nastaven atribut `operator` na operaci sčítání `self.operator = operator.add` a atribut `additional_right_angle` obsahuje hodnotu 0. Na základě těchto hodnot se ke středové souřadnici nakladače přičte polovina délky nakladače, a tato souřadnice je posunuta ve směru úhlu θ. Při výpočtu zadního senzoru je nastavena operace odčítání a hodnota úhlu zůstává 0 a boční senzory mají nastaven atribut `additional_right_angle` na hodnotu pravého úhlu. Metoda `update()` je periodicky volána k aktualizaci umístění senzorů na základě nových souřadnic nakladače.

Nejprve byl senzor definován jako kruh, jehož střed je umístěn na hraně obdélníku nakladače. Toto řešení ale nebylo ideální, protože takový senzor měl malý dosah a při zvětšení poloměru kruhu senzoru jeho detekce již zasahovala i do sousedních stran nakladače, což nereflektuje reálnou situaci.

Střed kruhu bylo potřeba zachovat na hraně obdélníku nakladače, protože následná vzdálenost překážky od nakladače je počítána mezi středy těchto kruhů. Nový návrh senzoru se skládá ze dvou kruhů, jednoho menšího a druhého většího. Větší kruh má střed umístěný na hraně nakladače a používá se pouze k vypočítání měřené vzdálenosti a vzhledem ke svému většímu poloměru zasahuje i do sousedících stran nakladače. Menší kruh má střed posunutý více do prostoru a je používán k detekci kolize s překážkou. Jakmile je detekována kolize menším kruhem, tak větší kruh spočítá vzdálenost mezi svým středem a středem detekované překážky. Princip výpočtu této vzdálenosti je obsažen v podkapitole 6.5.

6.5 Knihovna Collision

Collision je knihovna jazyka Python sloužící k výpočtu kolizí mezi zadefinovanými geometrickými objekty. V této podkapitole čerpám zejména z dokumentace knihovny Collision [31].

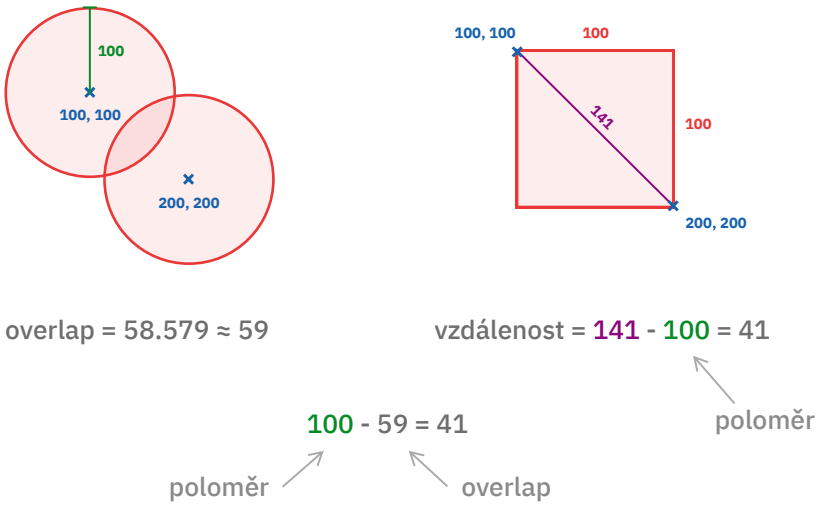


Obrázek 8: Různé návrhy implementace senzorů

Hlavními třídami této knihovny jsou `Vector`, `Circle`, `Poly` a `Response`. Třída `Vector` reprezentuje 2D vektor/bod a je definována souřadnicemi x a y. Kruh je reprezentován třídou `Circle`, která v konstruktoru přijímá středový bod a poloměr kruhu. Středový bod kruhu je instancí třídy `Vector`. Třída `Poly` reprezentuje konvexní polygon. Polygon je určen středovou souřadnicí a seznamem bodů, které jsou relativní ke středu polygonu. Instanci třídy `Poly` lze také získat ze středové souřadnice, šířky a délky polygonu pomocí funkce `Poly.from_box(pos, width, height)`. Třída `Response` slouží k zaznamenání výsledku kolize dvou objektů.

Nakladač je reprezentován třídou `Poly`. Objekt je vytvořen použitím funkce `poly.from_box(pos, width, height)`, protože jsou k dispozici údaje o délce a šířce nakladače. Při aktualizaci pozice je vypočítána nová hodnota směrového úhlu θ a středové souřadnice nakladače. Je tedy nutné aktualizovat tyto hodnoty i v reprezentaci nakladače objektem `Poly`. Aktualizace je provedena nastavením atributů `angle` a `pos` této třídy. Překážka v prostoru je reprezentována pomocí třídy `Circle`, které je předána souřadnice středu překážky a poloměr.

Ke kontrole možné kolize mezi dvěma objekty a a b je použita funkce `collision.collide(a, b, response = None)`, která vrací pravdivostní hodnotu na základě toho, zda nastala kolize mezi dvěma objekty, či nikoliv. Pokud je nastaven parametr `response` na instanci třídy `Response`, pak jsou dodatečné informace o kolizi uloženy do této instance. Funkce je využívána jak při kontrole kolize mezi nakladačem a překážkou, tak i při detekci překážky pomocí sen-



Obrázek 9: Výpočet vzdálenosti středu senzoru od hrany překážky

zorů. Kolize nakladače se kontroluje funkcí `check_collisions()` obsažené v souboru `loader.py`. Třída `sensor.py` také obsahuje funkci `check_collisions()` ke kontrole kolizí, která navíc vrací i nejmenší detekovanou vzdálenost mezi senzorem a překážkou. K výpočtu vzdálenosti mezi senzorem a překážkou slouží funkce `calculate_distance()` v souboru `sensor.py` zobrazená ve zdrojovém kódu 2. Tato funkce je zavolána pouze v případě, že menší kruh třídy `Sensor` detekoval kolizi s danou překážkou `obstacle`. Funkce vytvoří instanci třídy `Response`, která se poté předává funkci `collide()`. Hodnota atributu `collision.Response.overlap` je míra překrytí objektů kolize na nejkratší ose, vyjádřená číselnou hodnotou. Vzdálenost mezi středem senzoru a hranou překážky lze vyjádřit odečtením hodnoty překrytí od poloměru senzoru (viz obrázek 9).

```
1  def calculate_distance(self, obstacle):
2    collision_response = Response()
3
4    collide(self.bigger_circle, obstacle, collision_response)
5
6    new_distance = Sensor.BIGGER_SENSOR_RADIUS - collision_response.
       overlap
7
8    # saving distance if it is the new minimal
9    if new_distance < self.min_distance and new_distance >= 0:
10     self.min_distance = new_distance
```

Zdrojový kód 2: Výpočet kolize a vzdálenosti senzoru a překážky

7 Implementace algoritmu

Tato kapitola se zabývá implementací deep Q-learning algoritmu k navigaci nakladače v nezmapovaném prostředí. Nejdříve se věnuje rozdělení a struktuře zdrojových kódů. Poté je popsána naivní implementace, která slouží zejména k pochopení principů tohoto algoritmu. Na závěr je popsána implementace využívající knihovnu *Keras-rl*. Tato knihovna obsahuje veškeré funkce potřebné k práci s neuronovými sítěmi a reinforcement learning učením. Výsledná implementace dokáže úspěšně navigovat nakladač k cílové souřadnici okolo rozmístěných překážek.

7.1 Rozdělení zdrojového kódu

Zdrojový kód celkové implementace je rozdělen do čtyř hlavních souborů `loader.py`, `simulation.py`, `sensor.py` a `deep_q_learning.py`. Implementace používající knihovnu Keras-rl navíc obsahuje soubor `loader_env.py`, který je více popsán v sekci 7.3.

Soubor `loader.py` se zabývá implementací pohybu nakladače a definuje třídu `Loader`. Popis matematického modelu pohybu nakladače je obsažen v kapitole 5.2. Soubor `sensor.py` obsahuje implementaci třídy `Sensor`. Tato implementace je blíže popsána v kapitole 6.4. V souboru `simulation.py` je použita zejména knihovna *pygame*, a to k zobrazení vizuální simulace pohybu nakladače. Popisu této knihovny se věnuje kapitola 6.1.

Soubor `deep_q_learning.py` obsahuje implementaci algoritmu deep Q-learning. K provedení zvolené akce, výpočtu nových souřadnic nakladače a zjištění měření jednotlivých senzorů je používán soubor `loader.py`, který vnitřně používá funkcionalitu souboru `sensor.py`. Ke kontrole a zobrazení naučeného modelu slouží soubor `simulation.py`, který pomocí souboru `deep_q_learning.py` zobrazuje vybrané akce naučeného deep Q-learning modelu.

7.2 Naivní implementace

Motivací k vyzkoušení vlastní implementace deep Q-learning algoritmu byla zejména snaha o pochopení principů a získání citu pro mechaniku tohoto algoritmu.

Vzhledem k vysoké výpočetní náročnosti učícího procesu algoritmu běžel výpočet na serveru Katedry informatiky Univerzity Palackého v Olomouci. Na výpočetní jednotce nVidia Jetson Nano by učení algoritmu trvalo velmi dlouhou dobu a jednotka je určena spíše ke spouštění již naučeného modelu algoritmu, který by ovládal reálný nakladač.

Tato implementace není výpočetně optimalizovaná a z důvodu jednoduchosti učení obsahuje pouze pohyb nakladače bez senzorů a překážek. Výsledná implementace je natolik výpočetně náročná, že se algoritmus dokázal spolehlivě naučit navigovat do cíle pouze u zjednodušené verze prostředí, kde se nakladač pohybuje po horizontální ose dopředu a dozadu a cílová souřadnice se nachází na stejné vertikální souřadnici jako nakladač.

Při návrhu této implementace jsem čerpala zejména z knihy Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow [14], která obsahuje kapitolu o implementaci deep Q-learning algoritmu.

7.2.1 Popis implementace

Nejprve jsou vybrány hodnoty pro diskontní faktor a míru učení. Jak již bylo zmíněno v kapitole 2.3, diskontní faktor γ určuje důležitost budoucích hodnot a hodnota této konstanty byla nastavena na $\gamma = 0,95$. Míra učení α ovlivňuje, jak moc je nahrazena stará Q hodnota novou. Tato konstanta byla nastavena na $\alpha = 0,0001$, protože jsou pro správný běh důležité i předchozí Q hodnoty. Dále byly nastaveny hodnoty pro velikost učícího vzorku, počet učících epizod a maximální počet kroků v každé epizodě. Experience buffer, ukládající provedené kroky, je implementován pomocí Python struktury `deque`, která reprezentuje frontu pomocí dvousměrného spojového seznamu [14].

Neuronová síť predikující Q hodnoty akcí (*Q-network*) je vytvořena pomocí knihovny *TensorFlow Keras* [1]. Funkce vracející výslednou neuronovou síť je zobrazena ve zdrojovém kódu 3. Daná síť obsahuje čtyři skryté vrstvy obsahující 64, 128 a 64 neuronů. Aktivační funkcí skrytých vrstev je zvolena funkce ReLU, popsaná v kapitole 3.3. Ve výstupní vrstvě je nastavena lineární aktivační funkce. Neuronová síť *target network* je získána pomocí funkce `clone_model()` jako kopie sítě `Q-network`.

```
1  def create_neural_network(action_count, optimizer):
2      model = Sequential()
3      init = HeUniform()
4      model.add(Dense(64, input_dim=3, activation='relu',
           kernel_initializer=init))
5      model.add(Dense(128, activation='relu', kernel_initializer=init))
6      model.add(Dense(64, activation='relu', kernel_initializer=init))
7      model.add(Dense(action_count, activation='linear',
           kernel_initializer=init))
8
9      model.compile(loss='mse', optimizer=optimizer)
10     return model
```

Zdrojový kód 3: Inicializace neuronové sítě pomocí knihovny Keras [1]

Učící fáze algoritmu je implementována ve funkci `train_network()` a `training_step()`, které jsou obsaženy ve zdrojových kódech 4, 5.

Funkce `train_network()` se stará o obecný průběh učící fáze. Obsahuje vnější cyklus iterující přes určitý počet učících epizod a vnitřní cyklus iterující přes daný počet kroků v každé epizodě. Na začátku každé epizody je nakladač vrácen do počátečního stavu. Stav nakladače je reprezentován trojicí *[souřadnice-x, souřadnice-y, zaokrouhlený směrový úhel]*. Každý krok začíná výpočtem odpovídající hodnoty ϵ. Poté se zavolá funkce `perform_action()`, která na základě

ϵ-greedy strategie zvolí akci a tato akce je provedena pomocí výpočtu třídy `Loader`. Výsledkem funkce `perform_action()` je nový stav nakladače, aktualizovaný experience buffer, obdržená odměna za provedenou akci a pravdivostní hodnota, vypovídající, zda nakladač dojel do cíle. Pokud již bylo provedeno více než 50 epizod, je zavolána funkce `training_step()` a po každých 50 epizodách jsou aktualizovány váhy neuronové sítě `target_network`.

Funkce `training_step()` slouží k učení neuronové sítě *Q-network*. Výpočetní postupy této funkce jsou převzaty ze zdroje [14]. Nejprve je vybrán učící vzorek dané velikosti z experience bufferu pomocí funkce `sample_experiences()`. Prvky tohoto vzorku obsahují informace o stavu, akci, odměně, následujícím stavu a indikátor dosažení cílové souřadnice. Poté je použita neuronová síť `target_network` k predikci všech Q-hodnot následujících stavů z učícího vzorku. Algoritmus Q-learning předpokládá, že při výběru následujících akcí bude vybírána ta nejoptimálnější, generující nejvyšší Q-hodnotu. Tyto nejvyšší Q-hodnoty jsou poté použity v klasické Q-funkci, která je popsána v rovnici (1). Výsledek této funkce slouží jako cílová hodnota, ke které se síť `Q-network` snaží přiblížit. Proto je následně vytvořena maska, pomocí které jsou vybrány pouze akce generující již získané nejvyšší Q-hodnoty. Použitím metody gradientního sestupu a hodnoty střední kvadratické chyby jsou poté upraveny váhy sítě `Q-network`.

Po dosažení maximálního počtu epizod je neuronová síť `Q-network` uložena pomocí funkce knihovny Keras `network.save()`.

K vizualizaci naučeného deep Q-learning modelu je používán soubor `simulation.py`. Nejprve jsou načteny zvolené neuronové sítě a poté je opakovaně volána funkce `use_network()`, která po načtení aktuálního stavu nakladače použije neuronovou síť `Q-network` k predikci Q-hodnot všech akcí. Následně je provedena akce, u které neuronová síť vrátila nejvyšší Q-hodnotu. Po provedení této akce je vypočítán nový stav. Poté je ověřeno, zda se nakladač dostal do cílové souřadnice. Nakonec program souboru `simulation.py` vizuálně reprezentuje nový stav nakladače.

7.2.2 Zhodnocení naučených modelů

Při učení agenta, který měl k dispozici 7 různých rychlostí pro oba pásové pohony, nebylo dosaženo uspokojivého řešení. Tento model je pro neoptimalizovaný algoritmus příliš složitý a celková doba učení by byla velmi dlouhá. Výsledný agent nejčastěji jezdil do určité souřadnice a zpět, nebo poháněl pouze jeden pás, což znamenalo konstantní zatáčení nakladače. Dodatečně bylo implementováno i omezení pohybu nakladače v mřížce, kde nakladač nebyl schopen vyjet z mřížky 10×10 a pokud se o to pokusil, tak se souřadnice nezměnila a obdržel negativní odměnu. Toto omezení zaručuje, že se nakladač zbytečně neučí na stavech mimo mřížku pohybu, ale omezení nemělo nějak značný vliv na úspěšnost učení. Větší vliv na chování agenta měl počet učících epizod. Při 2000 epizodách chování agenta naznačovalo, že nerozpoznává změny v souřadnici osy y. Při 5000 epizodách bylo chování agenta nadějné, protože z počáteční souřadnice dojel téměř k cílové souřadnici, od které ho dělilo pouze jedno pole mřížky. Při

10000 epizodách agent opět vybíral akce vedoucí k nekonečné pohybové smyčce.

Dále byl vyzkoušen zjednodušený princip pohybu, který nakladači umožňoval akce dopředu, dozadu, otoč o 90° doleva a otoč o 90° doprava. Tento pohyb tedy prováděl změny směrového úhlu přesně o 90°, což zmenšilo celkový počet možných stavů. Zmíněné zjednodušení pohybu ale nemělo větší vliv na celkovou úspěšnost učení.

Velmi důležitý je také návrh funkce odměny. Odměna je definována jako rozdíl vzdáleností staré a nové souřadnice k cíli. Pokud je nová souřadnice blíže k cíli než stará, pak je odměna pozitivní, a pokud je nová souřadnice dál, tak je odměna negativní. Byly prováděny změny ve výši odměny za dosažení cíle, změny v intervalu možných hodnot odměny (normalizace na interval $[-1, 1]$) i změny v samotném výpočtu odměny, ani jedna z těchto úprav ale významně neovlivnila výsledný model.

Uspokojivé výsledky poskytuje model, používající velmi zjednodušený princip pohybu, který obsahuje pohyb pouze po horizontální ose prostředí. Nakladač se tedy může pohybovat pouze dopředu a dozadu a souřadnice cíle se nachází daný počet políček před, nebo za nakladačem. I přes větší zjednodušení prostředí je tento model přínosný, protože z něj lze odvodit, že algoritmus je schopný učení a největší překážkou je složitost prostředí a učícího procesu.

7.3 Knihovna Keras-rl

Knihovna Keras-rl [32] nabízí veškeré potřebné funkcionality k implementaci deep Q-learning algoritmu. Pomocí této knihovny je vytvořen program, který úspěšně naviguje nakladač do cílové souřadnice kolem rozmístěných překážek. Knihovna byla navržena zejména k použití s prostředím OpenAI Gym [33] a obsahuje implementace různých typů deep reinforcement learning algoritmů. Pro použití samotné knihovny bez prostředí Gym je vyžadováno rozšíření několika abstraktních tříd knihovny.

7.3.1 Popis použití knihovny

K použití knihovny Keras-rl bez prostředí Gym je nutné implementovat třídy reprezentující prostředí a možný prostor akcí nakladače. Tyto třídy dědí z abstraktních tříd knihovny Keras-rl. Soubor `loader_env.py` obsahuje definice tříd `LoaderEnv` a `LoaderSpace`.

Třída `LoaderEnv` specifikuje vlastnosti prostředí, ve kterém se agent nachází. Tato třída dědí z abstraktní třídy `Env`. Ve třídě jsou nejprve definovány rozsahy hodnot možných odměn, akcí a stavů. Je také vytvořena nová instance třídy `Loader`, se kterou třída `LoaderEnv` spolupracuje. Dále je definována hodnota cílové souřadnice a míra diskretizace směrového úhlu. Následně jsou uvedeny metody třídy `LoaderEnv`. Metoda `step()` přijímá index vykonávané akce a uvnitř volá metodu `perform_action()` k vykonání akce a aktualizaci pozice nakladače. Poté generuje odpovídající nový stav prostředí a volá metodu

`generate_reward()` s argumenty nového a starého stavu prostředí k výpočtu obdržené odměny. Nakonec metoda `step()` vrací trojici *(nový stav, odměna, dosažení cíle)*. O generování nového stavu prostředí se stará metoda `generate_state()`, která zjistí souřadnice pozice nakladače, zaokrouhlí hodnotu směrového úhlu a načte měřené hodnoty senzorů vzdálenosti. Výsledný generovaný stav je seznam obsahující hodnoty souřadnic, směrového úhlu a vzdáleností.

K zadefinování hodnot možných akcí uvnitř třídy `LoaderEnv` je použita třída `LoaderSpace`, která dědí z abstraktní třídy `Space`. Konstruktor této třídy přijímá vstupní hodnotu `n`, která reprezentuje počet možných akcí nakladače. Jednotlivé akce jsou reprezentovány číselnou hodnotou indexu, kde index je hodnota z intervalu $[0, n-1]$. Třída také definuje metodu `sample()`, která vybírá náhodný index akce a metodu `contains()`, která vrací pravdivostní hodnotu určující zda je argument metody odpovídající index nějaké akce.

Soubor `deep_q_learning_keras.py` nejprve vytvoří instanci třídy `LoaderEnv`, pomocí které ovládá a získává informace o prostředí nakladače. Princip vytvoření neuronové sítě je velmi podobný již zmíněnému zdrojovému kódu 3 z kapitoly o vlastní implementaci. Tato neuronová síť je použita ve funkci `network_setup()`, která vytváří instanci třídy `DQNAgent` z knihovny Keras-rl [34]. Funkce nejprve zvolí odpovídající strategii podle toho, zda se agent bude učit, nebo bude vykonávat naučený model. Poté je nastaven experience buffer jako instance třídy `SequentialMemory` knihovny Keras-rl. Při vytváření objektu třídy `DQNAgent` se předává nejen vytvořená neuronová síť, experience buffer a strategie, ale i počet možných akcí, minimální počet kroků před začátkem učení a hodnota argumentu `enable_double_dqn`. Tento argument určuje, zda bude algoritmus používat dvě neuronové sítě nebo pouze jednu (target network a q network, více zde). Poté je síť zkompilována s určeným optimalizačním algoritmem a funkcí chyby sítě. Pokud má program simulovat již naučený model, tak jsou navíc načteny váhy odpovídající neuronové sítě. Celá funkce `network_setup()` je zobrazena ve zdrojovém kódu 6. Trénování deep Q-learning agenta je provedeno pomocí metody `fit()` třídy `DQNAgent()`. Této metodě je předávána instance prostředí a celkový počet kroků učení. Po dokončení učení jsou výsledné váhy neuronové sítě uloženy do souboru.

7.3.2 Zhodnocení naučených modelů

Při použití knihovny Keras-rl byl nejprve vyzkoušen agent, který používal zjednodušený způsob pohybu (akce: dopředu, dozadu, otoč o 90° doleva, otoč o 90° doprava) s neomezeným pohybem mimo mřížku. Tento agent se učí velmi rychle (do hodiny), potřebuje na naučení menší počet epizod a konzistentně jezdí do cíle.

Poté byly do prostředí implementovány překážky a senzory. Agent v prostředí s překážkami nebyl úspěšný a výsledný naučený pohyb se skládal pouze z couvání dozadu. To bylo způsobeno chybou v implementaci senzorů, kde měření senzoru obsahovalo hodnotu nekonečno, pokud nebyla v dosahu senzoru žádná překážka. Implementovaná neuronová síť ale nebyla schopná hodnotu nekonečno zpracovat a k žádnému učení nedocházelo. Po přepsání hodnoty nekonečno hodnotou 1000

(náhodná hodnota větší než dosah senzoru) se model začal učit bez obtíží objíždět překážky až do cílové souřadnice. Počet kroků celkového procesu učení je větší, než u učení se zjednodušeným pohybem, i tak je učení tohoto modelu překvapivě rychlé a efektivní. Učení na serveru katedry informatiky trvá přibližně hodinu.

Dále byly do prostředí přidány další překážky, nacházející se přímo v optimální cestě nakladače ke zvýšení obtížnosti. Agent se naučil vyhnout i dodatečným překážkám a k dojetí do cíle začal vybírat jinou cestu bez překážek.

Celkově se podařilo vyvinout model řídící nakladač k cílové souřadnici a také jej zároveň naviguje kolem rozmístěných překážek. Informace o okolí model zpracovává ze senzorů vzdálenosti, které jsou na každé straně nakladače. Vytvořený model představuje úspěšnou implementaci deep Q-learning algoritmu a je v budoucnu potencionálně aplikovatelný na řízení reálného nakladače.

Výsledný model dokáže v simulaci, na základě senzorů vzdálenosti a znalosti aktuální pozice, navigovat nakladač k cílové souřadnici bez toho, aniž by došlo ke kolizi s umístěnými překážkami. Model je možné po dalším vývoji a trénování aplikovat na řízení nakladače v reálném prostředí.

8 Budoucí vývoj

Téma této diplomové práce je pouhou částí velmi komplexního úkolu, jehož cílem je autonomní navigace nakladače pomocí reinforcement learning algoritmu ve fyzickém prostředí. Byla bych ráda, kdyby má práce na tomto projektu ve spolupráci s firmou Technotrade pokračovala i po dokončení studia. V této kapitole stručně zmíním možné budoucí problémy a rozšíření této autonomní navigace.

8.1 Implementace algoritmu s reálným nakladačem

Na výsledky této práce, která implementuje pohyb a učení nakladače ve virtuálním prostředí, může navazovat další projekt, který má za cíl zprovoznit tento algoritmus s reálným nakladačem ve skutečném fyzickém prostředí. Použití jakéhokoliv reinforcement learning algoritmu mimo virtuální prostředí je ale velmi složité a v současné době stále výrazně převažuje implementace reinforcement learningu mimo reálné prostředí.

Reinforcement learning algoritmy spoléhají na přesnost reprezentace stavu prostředí a interakcí agenta s prostředím. Tuto přesnost nelze zcela zajistit v reálném světe, kde nemá na prostředí vliv pouze agent, ale i nepředvídatelné okolní vlivy. V těchto implementacích je nutné k reinforcement learning algoritmu přidat například sadu IF-THEN pravidel, které se snaží o kompenzaci těchto nepřesností.

8.1.1 Přesnost pohybu nakladače

Podstatná informace o stavu prostředí jsou souřadnice nakladače. Fyzický nakladač bude obsahovat GPS lokátor k určení jeho pozice. Tento lokátor má ale omezenou citlivost a také může nastat nějaká odchylka v měření. Proto by měl program zároveň provádět i výpočet virtuální simulace a počítat jak se souřadnicemi GPS, tak i s virtuálními souřadnicemi. Tyto virtuální souřadnice ale také nemusí odpovídat pozici nakladače ve fyzickém prostředí. Pohyb reálného nakladače je totiž ovlivněn i setrvačností pohybu (například pohyb maximální rychlostí a náhlé zastavení) a mírou prokluzování a tření pásů, které je závislé na nerovnostech a materiálu terénu. Pohyb je také ovlivněn váhovým rozložením nakladače, které ovlivňuje polohu těžiště stroje. Možné řešení spočívá v přidání IF-THEN pravidel, které budou nějakým způsobem monitorovat a kompenzovat tyto odchylky v pohybu nakladače.

8.2 Rozšíření o proměnlivé prostředí

Hlavní potencionální rozšíření této aplikace deep Q-learning algoritmu je zahrnutí proměnlivých pozic překážek, kde by se agent neučil informaci o konkrétní poloze překážek, ale více by pracoval s měřenými hodnotami senzorů. Při následné implementaci s reálným nakladačem je schopnost vyvarovat se kolizi s náhodně

umístěnou překážkou velmi důležitá, protože fyzické prostředí se často a nepředvídatelně mění a tato schopnost by mohla být pro správné fungování nakladače kritická.

Dále by bylo možné do učícího procesu zahrnout i proměnlivé cílové a počáteční souřadnice. Agentovi by se měnil nejen směr cílového pohybu, ale i vzdálenost a optimální cesta. Takové rozšíření by ale mohlo zcela narušit funkčnost algoritmu, protože se jedná o významnější změny, které zcela mění princip učení. Není jasné, zda by deep Q-learning algoritmus dokázal pojmout podstatu pohybu mezi dvěma stále se měnícími souřadnicemi, jelikož nelze ovlivnit podle jakých informací se algoritmus učí. V této situaci by bylo nutné, aby algoritmus zhodnotil rozdíly v aktuální a cílové souřadnici a na základě těchto rozdílů spolu s jeho směrovým úhlem by rozhodl o rotaci nakladače a směru jízdy. Pravděpodobně by bylo nutné upravit reprezentaci stavu prostředí a zahrnout do něj i vzdálenost nebo spíše směrový vektor k cílové souřadnici. Jinak by totiž dané q-hodnoty nereprezentovaly proměnlivost prostředí. Někdy by mohla být q-hodnota nadmíru snížena (akce zrovna vedla k oddálení od cíle) a někdy by byla ve stejné situaci nadmíru zvýšena.

Na rozšíření zahrnující proměnlivé pozice překážek jsem již začala pracovat. Myslím si, že algoritmus nebude mít s tímto rozšířením větší problém a dokáže navigovat nakladač do cílové souřadnice.

```
def train_network(loader, q_network, target_network, output_count,
    action_list, experience_buffer):

   total_episode_rewards = []

   for episode in range(EPISODE_COUNT):

   # reset the environment
      loader.set_position([0, 0])
      loader.set_heading_angle(0)


   # state represented as [x_coordinate, y_coordinate, heading_angle
      ]
      state = loader.get_grid_position()
      discretized_heading_angle = discretize_heading_angle(
         loader.get_heading_angle())
      state = np.append(state, discretized_heading_angle)

      total_reward = 0

      for _ in range(STEP_COUNT):
         epsilon = max(1 - episode / (EPISODE_COUNT - 100), 0.01)
         state, experience_buffer, done, reward = perform_action(
            loader, q_network, state, action_list, epsilon,
               experience_buffer)

         total_reward += reward

     # reached the goal
         if done:
            break

      if episode > 50:
         training_step(q_network, target_network,
                   experience_buffer, output_count)
      if episode % 50 == 0:
         target_network.set_weights(q_network.get_weights())

      total_episode_rewards.append(total_reward)

   return q_network, target_network
```

Zdrojový kód 4: Implementace celkového učení Q-network sítě

```
1  def training_step(q_network, target_network, optimizer,
       loss_function, experience_buffer, batch_size, discount_factor,
       output_count):
2
3      # experience sample
4      states, action_indexes, rewards, next_states, dones =
           sample_experiences(
5          experience_buffer, batch_size)
6
7      # predicting all Q values of the next state in each experience
8      # for target predicting we use target neural network
9      next_Q_values = target_network.predict(next_states)
10     # we assume the agent chooses the optimal action in next_state ->
           maximum Q value
11     max_next_Q_values = np.max(next_Q_values, axis=1)
12
13     # target Q values for each (state, action) pair in experiences
14     target_Q_values = (rewards + (1 - dones) *
15                        discount_factor * max_next_Q_values)
16
17     # the neural network returns Q values for all actions
18     # we know we only want the action that was chosen in the
           experience
19     # we mask out all the other actions
20     # actions contain action indexes
21     # e. g. actions = [1, 1, 0], output_count = 2 then mask = [[0,
           1], [0, 1], [1, 0]]
22     mask = one_hot(action_indexes, output_count)
23
24     with GradientTape() as tape:
25         # Q values for all actions in the experience states
26         all_Q_values = q_network(states)
27         # multiplying all Q values with the mask will zero out all the
               actions we don't want
28         # sum over axis=1 will remove all the zeros from the excessive
               actions
29         # Q values is a tensor with one Q value for each (state,
               action) pair in experiences
30         Q_values = reduce_sum(all_Q_values * mask, axis=1, keepdims=
               True)
31         # loss as a mean squared error between target and predicted Q
               value
32         loss = reduce_mean(loss_function(target_Q_values, Q_values))
33
34     # gradient descent step to minimize loss
35     gradients = tape.gradient(
36         loss, q_network.trainable_variables)
37     optimizer.apply_gradients(
38         zip(gradients, q_network.trainable_variables))
```

Zdrojový kód 5: Učící krok Q-network sítě

```
1  def network_setup(q_network, action_count):
2     if LOAD_MODEL:
3        policy = GreedyQPolicy()
4     else:
5        policy = EpsGreedyQPolicy()
6
7     memory = SequentialMemory(limit= 100000, window_length=1)
8     dqn = DQNAgent(model=q_network, enable_double_dqn=True,
         nb_actions=action_count, memory=memory, nb_steps_warmup=100,
9                target_model_update=1e-2, policy=policy)
10
11    dqn.compile(Adam(learning_rate=1e-3), metrics=['mae'])
12
13    if LOAD_MODEL:
14       dqn.load_weights('/path/to/model/weights/weight.hdf5')
15
16    return dqn
```

Zdrojový kód 6: Nastavení instance třídy *DQNAgent*

Závěr

Výsledkem této práce je implementace deep Q-learning algoritmu, který je schopen navigovat nakladač v neměnném virtuálním prostředí z počáteční do cílové souřadnice. Algoritmus zároveň ovládá nakladač tak, aby se vyhýbal okolním překážkám, které jsou umístěny v prostoru. Práce dále obsahuje matematickou reprezentaci pohybu nakladače a implementaci celkového virtuálního prostředí s překážkami. Model nakladače obsahuje čtyři senzory vzdálenosti, které slouží k prevenci kolize. Také je implementována jednoduchá grafická vizualizace prostředí k zobrazení pohybu nakladače. Pomocí grafické vizualizace je i barevně rozlišena detekce překážky pomocí senzorů nebo kolize nakladače s překážkou.

Před výslednou implementací deep Q-learning algoritmu, používající knihovnu Keras-rl, jsem navrhla vlastní naivní implementaci obsahující základní funkcionalitu algoritmu deep Q-learning. Pro vytvoření vlastní implementace jsem se rozhodla z důvodu lepšího pochopení principů algoritmu. Vzhledem ke složitosti řešeného problému a absenci optimalizací výpočtu je tato implementace schopná navigovat nakladač pouze ve velmi zjednodušeném prostředí.

V závěru práce je popsán budoucí vývoj tohoto projektu, který zahrnuje implementaci algoritmu s reálným nakladačem. Jsou probrány možné problémy této aplikace algoritmu v nekonzistentním fyzickém prostředí. Popsané je také rozšíření této implementace o proměnlivé pozice překážek, na kterém jsem již začala pracovat.

Conclusions

The result of this thesis is an implementation of deep Q-learning algorithm, which is able to navigate a track loader to the specified goal position in a stable virtual environment. The surrounding environment contains a number of obstacles. The algorithm navigates the loader around these obstacles preventing any collision. This thesis also contains a mathematical representation of the loader's movement and the implementation of virtual environment with obstacles. The model of the loader includes four distance sensors which are used for preventing collisions. A simple graphical visualization for visual monitoring of the loader's movement is implemented as well. The visualization uses color coding to show if an obstacle is detected by the sensor or that it has collided with the loader.

Before developing the resulting deep Q-learning implementation which uses Keras-rl library I have developed an my own naive implementation of this algorithm. My implementation only contains the basic functionality of the algorithm. I have decided to develop my own implementation so I could understand the principles of this algorithm better. Considering the difficulty of the navigation problem and absence of any optimizations this implementation is only capable of navigating the loader in a very simplified environment.

The plans for future development of this project are described at the end of the thesis. These plans include implementing the algorithm with a real physical loader or extending this implementation with variable obstacle positions which I have already began developing. Some of the possible problems with these plans are also discussed in the last section.

A Obsah přiloženého DVD

doc/
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní zdrojové kódy této diplomové práce. Obsažen je adresář `Own Implementation/` s veškerými naivními implementacemi deep Q-learning algoritmu a také adresář `Keras Implementation/` s výslednou implementací používající knihovnu keras-rl.

src/readme.txt
: Instrukce pro instalaci a spuštění programů implementujících učení deep Q-learning algoritmu a zobrazování vizuální simulace. Soubor dále obsahuje příkaz pro instalaci všech knihoven, potřebných k bezproblémovému spuštění souborů.

videos/
: Ukázková videa zobrazující vizuální simulace naučených modelů.

Literatura

[1] *Module TensorFlow Keras*. [online]. [cit. 2022-3-21]. Dostupný z: ⟨https://www.tensorflow.org/api_docs/python/tf/keras⟩.

[2] CAREW, Joseph M. *Reinforcement learning* [online]. 2021 [cit. 2022-4-8]. Dostupný z: ⟨https://www.techtarget.com/searchenterpriseai/definition/reinforcement-learning⟩.

[3] FRANÇOIS-LAVET Vincent, et al. *An introduction to deep reinforcement learning*. 2018. Dostupný také z: ⟨https://arxiv.org/pdf/1811.12560.pdf⟩.

[4] HRUBÁ, Gabriela. *Reinforcement learning na platformě LEGO MindStorms*. 2020.

[5] SUTTON, Richard S; BARTO, Andrew G. *Reinforcement learning: An introduction*. Second edition. 2018. Dostupný také z: ⟨http://incompleteideas.net/book/RLbook2020.pdf⟩. ISBN 978-0-262-19398-6.

[6] TRASK, Andrew W. *Grokking deep learning*. 2019.

[7] BĚLOHLÁVEK, Radim. *Umělá inteligence*. 2020.

[8] YANG, Danny. *How Perceptrons Work* [online]. 2019 [cit. 2022-2-22]. Dostupný z: ⟨https://yangdanny97.github.io/blog/2019/02/28/Perceptrons⟩.

[9] JAHR, Katrin; SCHLICH, Robert; DRAGOS, Kosmas; SMARSLY, Kay. Decentralized autonomous fault detection in wireless structural health monitoring systems using structural response data. 2015.

[10] V, Avinash Sharma. *Understanding Activation Functions in Neural Networks* [online]. 2017 [cit. 2022-2-27]. Dostupný z: ⟨https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0⟩.

[11] BROWNLEE, Jason. *How to Choose an Activation Function for Deep Learning* [online]. 2021 [cit. 2022-2-27]. Dostupný z: ⟨https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/⟩.

[12] SULTAN, Hossam H; SALEM, Nancy M; AL-ATABANY, Walid. Multiclassification of brain tumor images using deep neural network. *IEEE Access*. 2019, roč. 7, s. 69215–69225.

[13] MORALES, Miguel. *Grokking deep reinforcement learning*. 2020.

[14] GÉRON, Aurélien. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. 2019.

[15] MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David aj. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*. 2013. Dostupný také z: ⟨https://arxiv.org/pdf/1312.5602v1.pdf⟩.

[16] DEGRAVE, Jonas; FELICI, Federico; BUCHLI, Jonas aj. Magnetic control of tokamak plasmas through deep reinforcement learning. *nature*. 2022, roč. 602, s. 414–419.

[17] DEEPMIND. *Accelerating fusion science through learned plasma control* [online]. 2022 [cit. 2022-2-21]. Dostupný z: ⟨https://deepmind.com/blog/article/Accelerating-fusion-science-through-learned-plasma-control⟩.

[18] WIKIPEDIE. *Tokamak — Wikipedie: Otevřená encyklopedie* [online]. 2022 [cit. 2022-4-24]. Dostupný z: ⟨https://cs.wikipedia.org/wiki/Tokamak⟩.

[19] MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David aj. Human-level control through deep reinforcement learning. *nature*. 2015, roč. 518, č. 7540, s. 529–533.

[20] *Jetson Nano Developer Kit*. [online]. [cit. 2022-3-8]. Dostupný z: ⟨https://developer.nvidia.com/embedded/jetson-nano-developer-kit⟩.

[21] *Dapper 5000*. [online]. [cit. 2022-3-2]. Dostupný z: ⟨http://www.dapper.cz⟩.

[22] HELLSTRÖM, Thomas; RINGDAHL, Ola. *Path planning for off-road vehicles with a simulator-in-the-loop*. 2008.

[23] CORKE, Peter I; RIDLEY, Peter. Steering kinematics for a center-articulated mobile robot. *IEEE Transactions on Robotics and Automation*. 2001, roč. 17, č. 2, s. 215–218.

[24] *Kovaco MiniZ*. [online]. [cit. 2022-4-4]. Dostupný z: ⟨https://www.kovaco electric.com/cs/miniz⟩.

[25] FREDRIKSSON, Håkan; ANDERSSON, Ulf; HYYPPÄ, Kalevi. Track loader kinematics. *IFAC Proceedings Volumes*. 2010, roč. 43, č. 23, s. 139–142.

[26] YAMAUCHI, Genki; NAGATANI, Keiji; HASHIMOTO, Takeshi; FUJINO, Kenichi. Slip-compensated odometry for tracked vehicle on loose and weak slope. *Robomech Journal*. 2017, roč. 4, č. 1, s. 1–11.

[27] *Kvaser CANlib SDK*. [online]. [cit. 2022-3-8]. Dostupný z: ⟨https://www.kvaser.com/canlib-webhelp/index.html⟩.

[28] *Getting Started with Jetson Nano Developer Kit*. [online]. [cit. 2022-3-9]. Dostupný z: ⟨https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#setup⟩.

[29] *Getting Started with Jetson Nano Developer Kit*. [online]. [cit. 2022-3-9]. Dostupný z: ⟨https://www.pygame.org/docs/⟩.

[30] *How do I rotate an image around its center using Pygame?* [online]. [cit. 2022-3-10]. Dostupný z: ⟨https://stackoverflow.com/questions/4183208/how-do-i-rotate-an-image-around-its-center-using-pygame⟩.

[31] *Collision*. [online]. [cit. 2022-3-12]. Dostupný z: ⟨https://pypi.org/project/collision/⟩.

[32] PLAPPERT, Matthias. *keras-rl* [online]. 2016 [cit. 2022-3-30]. Dostupný z: ⟨https://github.com/keras-rl/keras-rl⟩.

[33] BROCKMAN, Greg; CHEUNG, Vicki; PETTERSSON, Ludwig aj. *OpenAI Gym* [online]. 2016 [cit. 2022-3-30]. Dostupný z: ⟨https://arxiv.org/pdf/1606.01540.pdf⟩.

[34] *DQNAgent*. [online]. [cit. 2022-4-5]. Dostupný z: ⟨https://keras-rl.readthedocs.io/en/latest/agents/dqn/⟩.

[35] IBARZ, Julian; TAN, Jie; FINN, Chelsea aj. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*. 2021, roč. 40, č. 4-5, s. 698–721.

[36] WIKIPEDIE. *Strojové učení — Wikipedie: Otevřená encyklopedie* [online]. 2021 [cit. 2022-2-18]. Dostupný z: ⟨https://cs.wikipedia.org/w/index.php?title=Strojov%C3%A9_u%C4%8Den%C3%AD&oldid=20640896⟩.

[37] WIKIPEDIE. *Střední kvadratická chyba — Wikipedie: Otevřená encyklopedie* [online]. 2021 [cit. 2022-2-23]. Dostupný z: ⟨https://cs.wikipedia.org/w/index.php?title=St%C5%99edn%C3%AD_kvadratick%C3%A1_chyba&oldid=20367446⟩.