AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Praca dyplomowa inżynierska

Opracowanie algorytmu sterowania pojazdem autonomicznym z wykorzystaniem algorytmów sztucznej inteligencji w oparciu o dane odczytane z czujników

Development of an algorithm for controlling an autonomic vehicle using artificial intelligence algorithms based on data read from sensors analysing the robot's environment.

Autor: *Karol Kocur*
Kierunek studiów: Informatyka Stosowana
Opiekun pracy: *Dr inż. Piotr Kustra*

Kraków, 2021

Spis treści

1 Wstęp

Integracja między światem cyfrowym i fizycznym postępuje w coraz szybszym tempie. Pojawiają się inteligentne pralki, lodówki a nawet kuchenki. Z roku na rok przybywa przedmiotów posiadających przydomek „inteligentny", lecz fala tych zmian nie dotyka jedynie życia domowego. Proces transformacji technologicznej jest widoczny również w przemyśle, co realizuje się poprzez powolne wprowadzanie koncepcji, zakładającej wykorzystanie nowej technologii, w celu powstania symbiozy między pracą wykonywaną przez ludzi i maszyny. Jedną z faz, jakie zostały wyznaczone do realizacji tego przedsięwzięcia jest etap autonomii (*autonomy wave*), któremu warto przyjrzeć się nieco bliżej [1].

Autonomia, według definicji słownika języka polskiego, to samodzielność oraz możliwość do podejmowania decyzji [2]. Jednakże, w odniesieniu do wyżej wymienionego programu, pojęcie to oznacza zautomatyzowanie czynności manualnych, które obecnie wykonywane są przez pracowników, jak również rozwiązanie problemu transportu, przez pojazdy nie posiadające kierowców. Warto zwrócić szczególną uwagę na drugą część powyższej definicji, ponieważ określa on działanie, jakie chciałbym w tej pracy przeanalizować.

Temat pojazdów, zdolnych przemieścić się z punktu A do B bez udziału człowieka, od kilku lat stopniowo przybiera na sile. Obecnie obowiązujące na świecie trendy, bardzo ułatwiają rozwój technologii dotyczących takich przedsięwzięć. Coraz więcej koncernów samochodowych wprowadza do swojej oferty urządzenia pozwalające przejąć część lub nawet całość obowiązków kierowcy.

Prekursorem tych działań jest amerykański producent elektrycznych samochodów osobowych – firma Tesla, która kilka miesięcy temu ogłosiła opracowanie nowego typu autopilota, sprowadzającego rolę człowieka jedynie do wciśnięcia przycisku „start" oraz wprowadzenia miejsca docelowego [3].

Pojazdy autonomiczne znalazły również zastosowanie w przemyśle. Coraz częściej można spotkać roboty, które nie posiadając osoby nimi sterującej, potrafią odebrać przedmiot w punkcie A oraz dostarczyć go do punktu B zachowując przy tym wszelkie środki ostrożności.

Elementem wspólnym w obu tych pojazdach jest algorytm, mający za zadanie wyznaczyć ruch pojazdu w czasie t+1 bazując na odczytach z czujników w czasie t. Ilość danych, ich różnorodność, a nawet dokładność, między poszczególnymi egzemplarzami, mogą się różnić,

lecz schemat zawsze pozostaje taki sam i właśnie ten algorytm działania chciałbym w tej pracy przybliżyć.

2 Przegląd istniejących autonomicznych pojazdów

W dzisiejszych czasach coraz częściej można dostrzec duże zainteresowanie przedmiotami wykorzystującymi algorytmy sztucznej inteligencji. To zjawisko nie jest przypadkowe, ponieważ wiele z tych urządzeń dostosowuje się do środowiska docelowego, co pozwala uniknąć kosztów modernizacji. Takie podejście zostało również poruszone w założeniach czwartej rewolucji przemysłowej zakładającej integrację przemysłu z najnowszymi osiągnięciami nauki, dlatego też, wykorzystanie tego typu maszyn będzie stopniowo wzrastać.

Pierwszym przykładem pojazdu autonomicznego, bazującym na podobnej dziedzinie nauki, jaka została wykorzystana w tej pracy, jest produkt „Freight1500" firmy Fetch Robotics. Robot ten pozwala na transport ładunków, których waga nie przekracza 1500kg, a dzięki zastosowaniu ośmiu kamer oraz dwóch czujników LIDAR, umożliwia precyzyjne oraz bezkolizyjne dostarczenie ich do miejsca docelowego. Dodatkowym atutem kamer, jest możliwość analizy obrazu 3D, co pozwala na wykrycia przeszkód znajdujących się na różnej wysokości [4].

Innym przykładem, są pojazdy, które swoją popularność zdobyły podczas obecnie trwającej epidemii. Ich głównym zadaniem jest świadczenie usług z zakresu doręczania wszelkiego rodzaju obiektów, od paczek po żywność. Firma STARSHIP jest jednym z twórców tego typu pojazdów. Trasa, po jakiej porusza się robot, jest wyznaczana za pomocą kamer, czujników ultradźwiękowych oraz GPS, dzięki tak obszernym informacjom, istnieje możliwość zdefiniowania zachowania robota np. przy przejściu dla pieszych. W czasie światowej pandemii, system zdobył dość dużą popularność, wykonując pięć tysięcy dostaw wyłącznie przy użyciu pojazdu autonomicznego, co zostało potwierdzone przez firmę [5].

Takie rozwiązania posiadają bardzo wiele zalet i w przyszłości będą coraz częściej wykorzystywane, ponieważ w wielu przypadkach, dzięki ich niezawodności, możliwe jest ograniczenie liczby wypadków, a co za tym idzie zwiększenie poziomu ogólnego bezpieczeństwa. Jednakże ta technologia posiada również wady a jedną z nich jest cena, jaką należy uiścić, nabywając urządzenie posiadające autonomiczny system. Wynika to głównie z faktu, że cała procedura wchodząca w skład Przemysłu 4.0 zaczyna być dopiero powoli testowana przez większe firmy, lecz z czasem, zainteresowanie tą tematyką będzie stopniowo wzrastać, co prawdopodobnie spowoduje spadek cen.

3 Algorytmy stosowane do nawigacji pojazdem autonomicznym

Sterowanie pojazdem autonomicznym nie jest prostym zadaniem, ponieważ wymaga uwzględnienia wielu czynników, które podlegają ciągłym modyfikacjom. Ilość parametrów jakie należy wziąć pod uwagę jest ogromna, co znacznie utrudnia opracowanie algorytmu potrzebnego do wyznaczenia kolejnego ruchu pojazdu. Mimo tego, zostały opracowane systemy, które spełniają postawione im zadania, doprowadzając do celu pojazd bez pomocy osoby sterującej.

3.1 Logika rozmyta – model Mamdani

Modele logiki rozmytej są w stanie przewidywać zachowanie badanego obiektu na podstawie reguł IF-THEN. Dzięki temu, posiadają szerokie zastosowanie w robotach, służących w rolnictwie do uprawy ziemi [6]. Dużym plusem w ich wykorzystywaniu, jest łatwe dobieranie zaplanowanych akcji do wcześniej przewidzianych zmiennych wejściowych oraz bardzo prosty sposób implementacji. Niestety ze względu na mnogość parametrów, może się okazać, że nie wszystkie przypadki zostały dokładnie sprawdzone i pojazd nie będzie wiedział jaki ruch ma w danej chwili wykonać [7].

3.2 Bug Algorithm

Kolejnym sposobem służącym do nawigowania robotami jest *bug algorithm.* Został on wynaleziony w latach osiemdziesiątych ubiegłego wieku, poprzez udoskonalenie dwóch rozwiązań służących wyznaczaniu najkrótszej drogi tj. algorytm A* oraz Dijkstra [8]. Obecnie jego nowsza wersja *Bug2*, polega na podążaniu robota za linią, która jest wyznaczona podczas uruchomienia algorytmu i prowadzi z punktu początkowego do celu [9]. Niestety, ze względu na specyfikę działania, roboty bazujące na tym algorytmie są one bardzo czułe na odczyty danych z czujników i wszelkie zakłócenia drastycznie obniżają ich skuteczność [10].

3.3 Deep Q-learning

Algorytm powstał poprzez połączenie dwóch rozwiązań. Pierwszym z niech jest *Q-learning,* wykorzystujący do działania środowisko, w jakim dany obiekt ma egzystować, w celu oceny akcji przez niego podejmowanych. Natomiast drugim rozwiązaniem jest *Deep learning,* dostarczający wielowarstwowy model, który korzystając z algorytmu Q-learning, jest w stanie określić najbardziej odpowiednią akcję dla danego stanu środowiska [11]. Zachowanie prowadzonego obiektu, w przeciwieństwie do algorytmów wykorzystujących model Mamdani, nie jest odgórnie zdefiniowane bazując na wiedzy eksperckiej konstruktora. Algorytm sam,

poprzez ocenę każdego ruchu, definiuje najlepsze schematy postępowań [12]. Dzięki dużej liczbie danych jaką można pozyskać ze środowiska oraz nieograniczonej liczbie prób istnieje możliwość nauczenia algorytmu sekwencji ruchów idealnie nadających się do kierowania autonomicznym pojazdem.

3.4 Podsumowanie

Analizując wady oraz zalety powyższych algorytmów, stwierdzam, że najbardziej odpowiednim do wykonania postanowionego celu będzie *Deep Q-learning*. Dzięki wykorzystaniu sieci neuronowych oferuje on dostosowanie się do nieznanego środowiska, w jakim pojazd będzie się poruszał, jak również pozwala rozbudować bazę zachowań o elementy nie przewidziane przez autora algorytmu. Dodatkowym atutem wynikającym z zastosowania sieci neuronowych jest szybkość ich działania oraz brak dużej bazy danych, co pozwala na używanie go na urządzeniach mobilnych, gdzie wymagania dotyczące pamięci są dużo bardziej restrykcyjne.

4 Cel Pracy

Celem pracy jest zaprojektowanie, wykonanie oraz przetestowanie algorytmu wykorzystującego sztuczną inteligencje, który posłuży do sterowania autonomicznym pojazdem. Robot posiadający gotowe rozwiązanie, powinien móc dojechać do celu wyznaczonego przez współrzędne GPS, przy równoczesnym ominięciu przeszkód stojących na jego drodze. Planowanymi źródłami informacji o środowisku, w którym dany robot się znajduje będą:

- Radar wykorzystujący wiązkę lasera (LIDAR), w celu wykrycia oraz oceny odległości od przedmiotów znajdujących się w pobliżu pojazdu,
- Magnetometr pozwalający na wyznaczenie kierunku w jakim robot ma się poruszać,
- GPS, dzięki któremu można uzyskać obecne położenie jak również określić współrzędne punktu docelowego.

W celu osiągnięcia głównego zadania zamierzone jest wykonanie poniższych zadań szczegółowych:

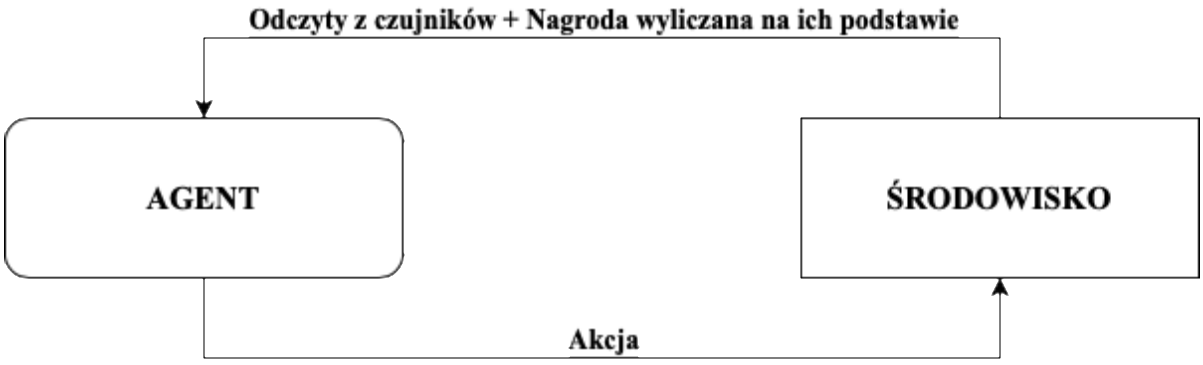
- Dobór odpowiedniego algorytmu,
- Przygotowanie wirtualnego środowiska wykorzystywanego w treningu pojazdu,
- Przygotowanie modelu odwzorowującego obiekt rzeczywisty,
- Dodanie odpowiednich czujników wchodzących w interakcję z ww. środowiskiem oraz zbierających dane,
- Obsługa symulacji w taki sposób, aby zapewnić poprawność danych wykorzystywanych podczas uczenia,
- Implementacja oraz testy algorytmu.

5 Omówienie wybranego algorytmu

Po analizie metod prowadzących do rozwiązania głównego problemu przedstawionych w rozdziale 3, najlepszym wyborem okazał się *Deep Q-Learning*.

5.1 Reinforcment learning

Reinforcment learning należy do grupy rozwiązań, które do nauki wykorzystują uczenie nadzorowane, a podstawą ich działania jest psychologia tresury zwierząt. Środowisko, które zostanie wykorzystane podczas procesu uczenia, zawiera warunki z jakimi gotowy algorytm będzie obcował. W tak przygotowanej strukturze, umieszczany jest obiekt (np. robot) zwany agentem. Jego zadaniem jest znalezienie efektywnej taktyki, poprzez wykonywanie ruchów w oparciu o dane jakie zostaną zebrane ze środowiska przed rozpoczęciem akcji. Następnie, dane posunięcie poddawane jest ocenie, po której agent zostaje nagrodzony za dobrze wykonany ruch, lub ukarany, gdy ten ruch jest niepoprawny (Rys. 5.1).



Rys. 5.1 Reinforcment learning (źródło: Opracowanie własne)

Obiekt dąży do zgromadzenia jak największej wartości przypisanej do nagrody, co jest uzyskiwane poprzez maksymalizację bonusów lub minimalizację kar. Dzięki swojej uniwersalności algorytmy te są wykorzystywane w różnych dziedzinach nauki począwszy od robotyki po psychologię [13].

5.2 Q-learning

Q-learning [14] to jeden z najczęściej używanych algorytmów z grupy *Reinforcment learning.* Algorytm ten zachowuje koncept zawiązany z agentem, środowiskiem, nagrodą oraz karą. Schemat jego działania dzieli się na dwie części. Pierwszą z nich jest eksploracja

polegająca na losowym doborze akcji do danych zebranych ze środowiska oraz oceną tego zachowania. Jej celem jest poznanie otoczenia, jak również uzupełnienie podstawowej bazy stan-akcja, zwanej *Q-table* (Rys. 5.2), która w dalszej części posłuży do wyznaczania najbardziej odpowiedniego zachowania. Odczyty z czujników umieszczone w tabeli są obserwacją pobraną ze środowiska, więc mogą występować w różnej postaci. Przykładowo, podczas analizy gry „Kółko i krzyżyk", może być tam umieszczona informacja o różnych kombinacjach stanu planszy. Natomiast akcje w takim wypadku, mogą oznaczać postawienie kółka w danej komórce.

Q-table			
	Akcja 1	**Akcja 2**	**Akcja 3**
Odczyty z czujników 1	Wartość Q-value dla akcji 1 wykonanej przy odczytach z czujników 1	Wartość Q-value dla akcji 2 wykonanej przy odczytach z czujników 1	Wartość Q-value dla akcji 3 wykonanej przy odczytach z czujników 1
Odczyty z czujników 2	Wartość Q-value dla akcji 1 wykonanej przy odczytach z czujników 2	Wartość Q-value dla akcji 2 wykonanej przy odczytach z czujników 2	Wartość Q-value dla akcji 3 wykonanej przy odczytach z czujników 2

Rys. 5.2 *Q-table* (Źródło: Opracowanie własne)

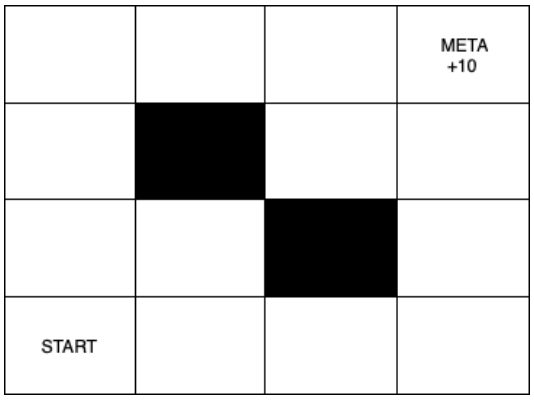
Drugim etapem jest eksploatacja, w której na podstawie wcześniej stworzonej *Q-table*, dobierany jest taki ruch, który posiada największą wartość *Q-value*. W celu wyznaczenia najbardziej odpowiedniej akcji, podejmowanej dla danego stanu, zastosowano poniższy wzór [15]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[R + \gamma\, maxQ(s',a') - Q(s,a)] \tag{5.1}$$

gdzie:

- Q – funkcja definiująca wartość nagrody za akcje a w stanie s,
- s – obecny stan,
- a – wykonana akcja w obecnym stanie,
- R – nagroda uzyskana za wykonanie akcji a powodującej przejście ze stanu s do s',
- γ - współczynnik definiujący ważność nagrody z kolejnego stanu ($0 < \gamma < 1$),
- s' – kolejny stan,
- $maxQ(s', a')$ - maksymalna wartość nagrody ze wszystkich akcji dla stanu s',
- α - współczynnik definiujący stopień ważności przyszłych wartości *Q-values*.

W celu przedstawienia działania algorytmu stworzono planszę (Rys. 5.3), która posłuży jako środowisko dla robota, umieszczonego w bloku z napisem „START" i potrafiącego z założenia wykonać tylko trzy akcje: krok w przód, obrót w lewo o 90°, obrót w prawo o 90 °.



Rys. 5.3 Przykładowe środowisko (Źródło: Opracowanie własne)

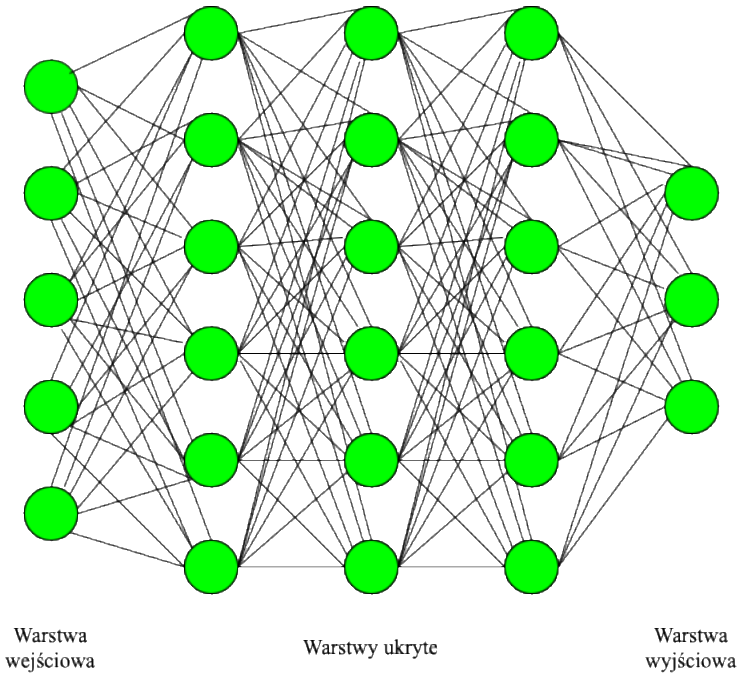
`

Pierwszym etapem jest opisana wcześniej eksploracja, podczas której robot poznaje środowisko wykonując losowe akcje niezależne od obserwacji, uzupełniając przy tym *Q-table,* która początkowo jest zainicjowana zerami. Do każdorazowej aktualizacji *Q*-values, wykorzystywany jest wzór 5.1. To dzięki niemu robot, z biegiem czasu, stopniowo przechodząc w tryb eskalacji, zaczyna zdobywać cel. Dzieje się tak, ponieważ wartość zapisana w *Q-table* dla danego stanu *s* oraz akcji *a* jest zmieniana po przejściu w stan *s'* biorąc pod uwagę największą wartość Q w tym stanie.

Q-learning niestety posiada wadę i jest nią wielkość bazy, w której przetrzymywana jest para stan-akcja oraz wartość nagrody jaką jej przydzielono. Dla przykładu, jeżeli w danym środowisku ilość wszystkich stanów wynosi *m*, a liczba akcji jakie może podjąć agent wynosi *n*, to wielkość bazy będzie wynosić *m* x *n*. Ponadto, podczas dobierania odpowiedniego zachowania dla danych pobranych ze środowiska należy przeprowadzić (*n*-1) porównań, natomiast podczas aktualizowania tablicy ilość porównań wynosi *m* (*n*-1). Niestety tak duża ich ilość nie jest dopuszczalna podczas działania w prawdziwym świecie. Złożoność środowiska, definiująca ilość parametrów jest ogromna, co sprawia, że czas poświęcony aktualizacji bazy rośnie wykładniczo [16]. Rozwiązaniem problemu jest zastosowanie sieci neuronowej, która przyjmie rolę bazy definiującej określony ruch robota w oparciu o dane pobrane ze środowiska.

5.3 Głębokie sieci neuronowe

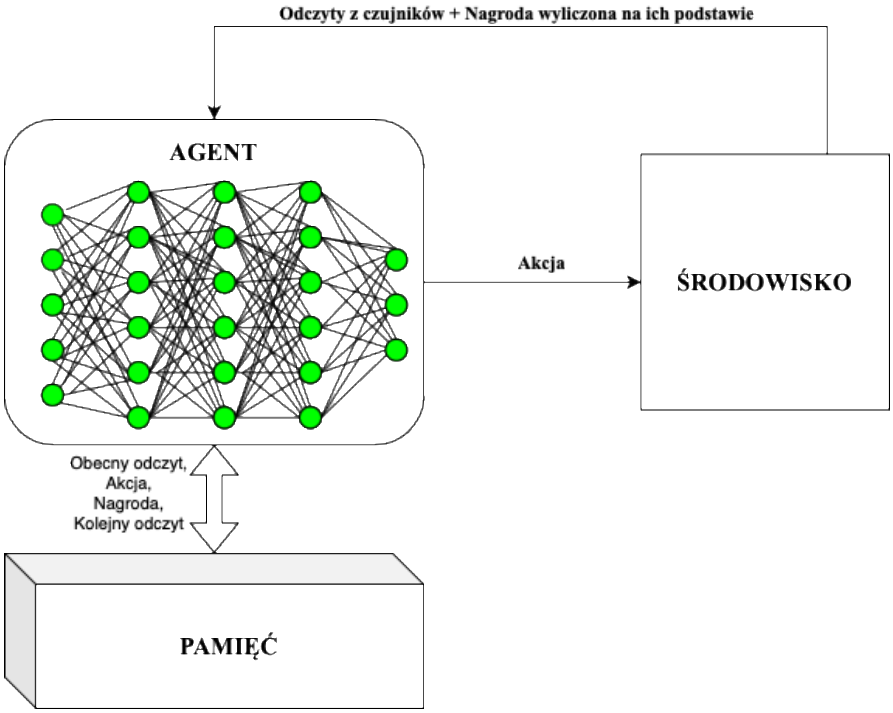
Głębokie sieci neuronowe (*Deep neural network)* to specyficzny rodzaj klasycznych sieci neuronowych [17]. Posiadają one „głębokość", która jest definiowana poprzez ilość ukrytych warstw pomiędzy wejściem a wyjściem (Rys. 5.4). Próg przejścia między klasyczną, a głęboką wersją sieci nie jest sprecyzowany, lecz uważa się, że przy ilości warstw większej niż dwie, sieć postrzegana jest jako głęboka. Ponadto istnieją jeszcze sieci bardzo głębokie posiadające ilość warstw ukrytych powyżej dziesięciu. Dzięki tak dużej ilości parametrów, stworzona struktura jest w stanie identyfikować różne informacje wejściowe oraz na ich podstawie obliczać odpowiednią wartość wyjściową [18]. Przykładem takich sieci, mogą być te, wykorzystywane w medycynie [19]. Dane, do tego typu rozwiązań często skaładają się z zdjęć, wykresów oraz zwykłych analiz. Dzięki zastosowaniu głębokich sieci neuronowych istnieje możliwość połączenia ich w jedną całość oraz zaproponowania diagnozy będącej wynikiem ich analizy.



Rys. 5.4 Głęboka sieć neuronowa (Źródło: Opracowanie własne)

5.4 Deep Q-learning

Deep Q-learning [20] jest to połącznie algorytmu *Q-learning* z możliwościami sztucznej inteligencji. Wykorzystywana sieć neuronowa przyjmuje obserwacje jakich dokonał agent w środowisku, natomiast jej wyjściem jest wektor wartości nagrody dla poszczególnych akcji [21]. Aby przełamać zależności między poszczególnymi obserwacjami występującymi podczas stosowania podstawowego algorytmu *Q-learning* zastosowano pamięć wykonywanych ruchów [22]. Jej wprowadzenie pozwala na uczenie sieci neuronowej poprzez wykorzystanie pamięci wykonanych ruchów, z której losowo będzie pobierana pewna próbka zawierająca stan, akcje, wartość nagrody oraz kolejną obserwację po wykonaniu zapisanego ruchu (Rys. 5.5). Takie połączenie dwóch metodologii działania pozwala na przystosowanie sieci do radzenia sobie w warunkach rzeczywistych, co przekłada się coraz większe zastosowanie w pojazdach autonomicznych.



Rys. 5.5 Schemat działania *Deep Q-learning* (Źródło: Opracowanie własne)

6 Dobór czujników do analizy środowiska

Przy wykorzystaniu algorytmu opisanego w poprzedni rozdziale należy zapewnić dane, które zobrazują środowisko w jakim znajduje się robot. Do tego celu zostaną wykorzystane trzy urządzenia.

6.1 LIDAR – RPLIDAR A1

Lidar (Rys. 6.1) to urządzenie służące do pomiaru odległości między obiektem znajdującym się w środowisku, a pojazdem, na którym jest zamontowany. Jego działanie polega na emitowaniu sygnałów laserowych, które odbijają się od otoczenia. Następnie to obicie jest przechwytywane i na jego podstawie obliczana jest odległość oraz kąt powstały między obiektem a czujnikiem [23].

Rys. 6.1 RPLIDAR A1 (Źródło: https://www.slamtec.com/en/Lidar/A1https://www.slamtec.com/en/Lidar/A1)

6.2 Moduł GPS

GPS (Rys. 6.2) to urządzenie pozwalające zlokalizować obiekt poprzez otrzymanie jego współrzędnych geograficznych, jak również dzięki niemu można określić położenie wyznaczonego celu, do którego pojazd ma dążyć.

Rys. 6.2 Moduł GPS (Źródło: https://botland.com.pl/moduly-gps/14643-modul-l76x-multi-gnss-gpsbdsqzss-waveshare-16332.html)

6.3 Magnetometr

Magnetometr (Rys. 6.3) to urządzenie, które pozwala na pomiar pola magnetycznego, dzięki któremu jest w stanie określić kierunek poruszania się obiektu, posiadającego taki moduł.

Rys. 6.3 Magnetometr (Źródło: https://botland.com.pl/czujniki-9dof-imu/13155-dfrobot-gravity-10dof-ahrs-3-osiowy-akcelerometr-zyroskop-i-magnetometr.html)

7 Oprogramowanie wykorzystane w tworzeniu symulacji

Aplikacja poprzez zastosowanie algorytmu korzystającego z uczenia maszynowego wymaga stworzenia środowiska wirtualnego. Jest to spowodowane obecną sytuacją na świecie, która wymaga ograniczenia kontaktów, co uniemożliwia współpracę z osobami tworzącymi pojazd autonomiczny. Wykreowana sztucznie przestrzeń, w której pojazd będzie mógł doskonalić nawigowanie do celu, powinna odzwierciedlać warunki, jakie pojazd napotka podczas egzystowania w świecie rzeczywistym.

7.1 Gazebo

Gazebo jest to symulator umożliwiający testowania rozwiązań, projektowanie robotów, wykonywanie testów oraz trenowanie algorytmów opartych o sztuczną inteligencję. Dzięki temu programowi istnieje możliwość odtworzenia realistycznych warunków w świecie wirtualnym. Jednym z jego zalet jest zaimplementowany silnik, który pozwala symulować fizykę odpowiadającą tej w świecie rzeczywistym. Kolejnym autem jest interfejs graficzny, dzięki któremu istnieje możliwość tworzenia prostych figur geometrycznych, pomagających w kreowaniu środowiska [24]. Gazebo dostarcza również prosty edytor graficzny służący do tworzenia ścian, schodów, okien i drzwi, co pozwala na wykonania zamkniętego środowiska testowego. Wszystkie elementy, które są tworzone w tym programie wykorzystują specjalnie do tego przygotowany język SDFormat.

7.2 SDFormat

Jest to język oparty na formacie XML, używany w Gazebo do szczegółowego definiowania obiektów biorących udział w symulacji. Podczas tworzenia środowiska daje on możliwość zdefiniowania terenu, który może zostać zaprojektowany w zewnętrznym programie, poprzez utworzenie jego siatki oraz dobrania koloru, lub udostępnia sposobność zbudowania go z wykorzystaniem już zaimplementowanych prostych struktur geometrycznych. Ponadto kolejnym autem tego języka jest możliwość definiowania zakresu ruchów między wcześniej stworzonymi elementami, co pozwala odwzorować poruszanie się poszczególnych elementów zgodnie z zachowaniem w świecie rzeczywistym. Na domiar tego istnieje wariant tworzenia sensorów, właściwości powierzchni oraz sił działających między poszczególnymi obiektami, a to wszystko może zostać poddane kontroli poprzez dołączeni zewnętrznych programów tworzonych w języku C++ oraz wykorzystujących API poszczególnych komponentów. Dodatkowo, język udostępnia możliwość definiowania źródeł światła, które mogą okazać się kluczowe podczas pracy z wirtualną kamerą [25].

7.3 Robot Operating System (ROS)

ROS to platforma programistyczna pozwalające tworzyć oprogramowanie wykorzystywane w działaniu robotów. Zawiera ono zestaw zaimplementowanych rozwiązań w postaci bibliotek oraz narzędzi, pozwalających zredukować trudności związane z tworzeniem oprogramowania [26]. Struktura, opiera się na połączeniach procesów *peer-to-peer* przy użycia infrastruktury ROS. Jej podstawą są węzły, które komunikują się między sobą z wykorzystaniem wiadomości (*Messages*) używając do tego kanałów oraz wzorca *publish-subscribe*. Ponadto platforma umożliwia wykorzystanie serwisów, wysyłających komunikaty w sposób synchroniczny [27].

7.4 OpenAI_ROS

Jest to zestaw narzędzi, wywodzący się z OpenAI_Gym[1], który został przystosowany do działania w środowisku Gazebo + ROS. Zarówno jego podstawowa wersja jak i opisywana w tym rozdziale zawiera struktury, które pomagają tworzyć oraz uczyć algorytmy sztucznej inteligencji z dziedzin *Reinforcment learning.* Dostarcza klasy, które zapewniają sterowanie środowiskiem symulacji poprzez użycie odpowiednich serwisów jak również zapewnia strukturę umożliwiającą przejrzysty podział funkcjonalności między różnymi plikami [28].

[1] https://gym.openai.com

8 Oprogramowanie wykorzystane w tworzeniu algorytmu

Tworzenie środowiska nie miałoby sensu bez algorytmu mającego na celu sterowanie robotem. Aby go wykonać, wykorzystano różne języki programowania jak również biblioteki wspomagające implementację rozwiązań wykorzystujących uczenie maszynowe.

- Python – jest to język wysokiego poziomu wspierający paradygmat programowania obiektowego. Dzięki rozbudowanej bazie bibliotek oraz dużej liczbie osób aktywnie w nim programujących zyskał popularność w dziedzinach nauki zajmujących się analizą danych w szczególności uczenia maszynowego,

- Tensorflow – to platforma stworzona do tworzeni rozwiązań wykorzystujących zalety uczenia maszynowego. Zawiera zestaw narzędzi oraz bibliotek wspomagających pracę z sieciami neuronowymi poprzez dostarczanie gotowego API do ich budowy, trenowania oraz walidacji,

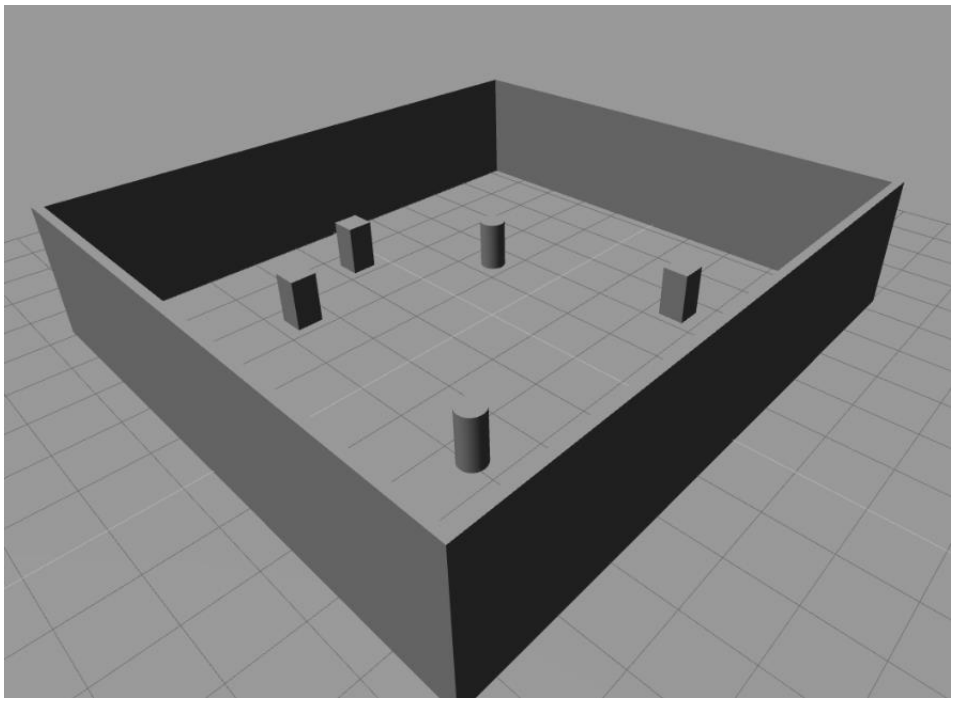
- Keras – API ułatwiające korzystanie z wyżej opisanej biblioteki. Minimalizuje ilość akcji wymaganych do zdefiniowania przez użytkownika oraz dostarcza przejrzyste informacje odnośnie błędów występujących podczas pracy nad rozwiązaniem wykorzystującym sieci neuronowe.

9 Wykonanie symulacji

Pierwszą czynnością jaką należy wykonać jest zbudowanie wirtualnego świata w jakim robot będzie egzystował. Przegotowane środowisko musi jak najbardziej przypominać to naturalne. Ponadto sam pojazd musi zostać przygotowany w taki sposób, aby jak najbardziej odwzorowywał zachowanie rzeczywistego obiektu, dzięki temu łatwiej będzie przenieść model uczony w świecie wirtualnym do świata rzeczywistego.

9.1 Utworzenie środowiska

Pierwszym etapem, wymaganym do zrealizowania celu jest przygotowanie odpowiedniego środowiska treningowego (Rys. 9.1), w którym agent może się poruszać ucząc się odpowiednich zachowań. Zostało to zrealizowane z wykorzystaniem symulatora Gazebo, który zapewnia dostęp do edytora modeli. Dzięki niemu można wyznaczyć granice środowiska, tak aby model nie poruszał się bez końca w jedną ze stron. Dodatkowo, wewnątrz zdefiniowanego obszaru umieszczono trzy prostopadłościany oraz dwa walce symulujące przeszkody. Głównym zadaniem pojazdu w tym środowisku jest osiągnięcie celu, który jest wyznaczany na początku symulacji oraz każdorazowo zmieniany po jego zdobyciu.



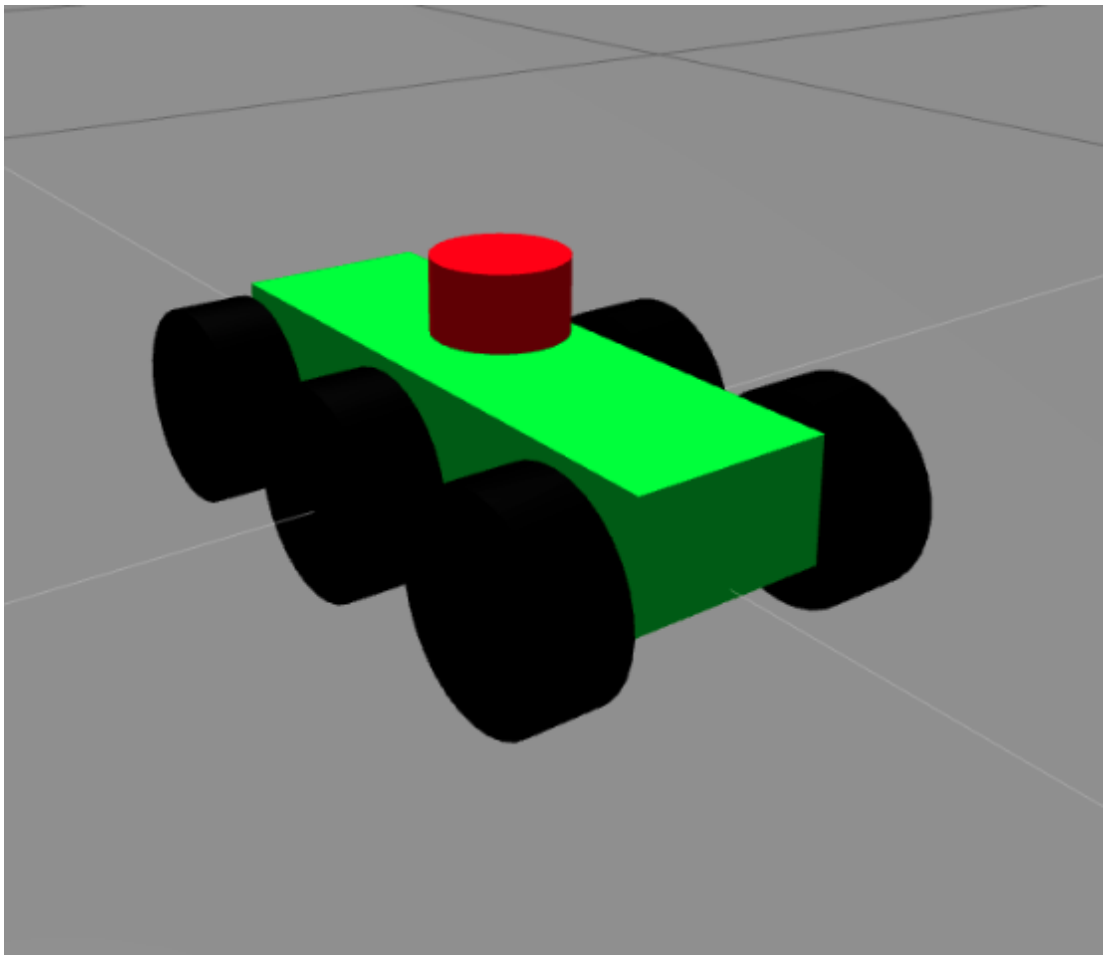
Rys. 9.1 Wirtualne środowisko treningowe (Źródło: Opracowanie własne)

9.2 Wykonanie modelu pojazdu

Na wykonanie modelu składało się dużo więcej czynności głównie wynikających z możliwości jego poruszania, a także specyficznego ruchu, który wykonywał skręcanie obiektu bez zastosowania skrętnej osi kół. Ponadto model musiał dostarczać informacje odpowiadające tym, które dostarcza pojazd w świecie rzeczywistym, co również wymagało odpowiednich komponentów.

9.2.1 Struktura

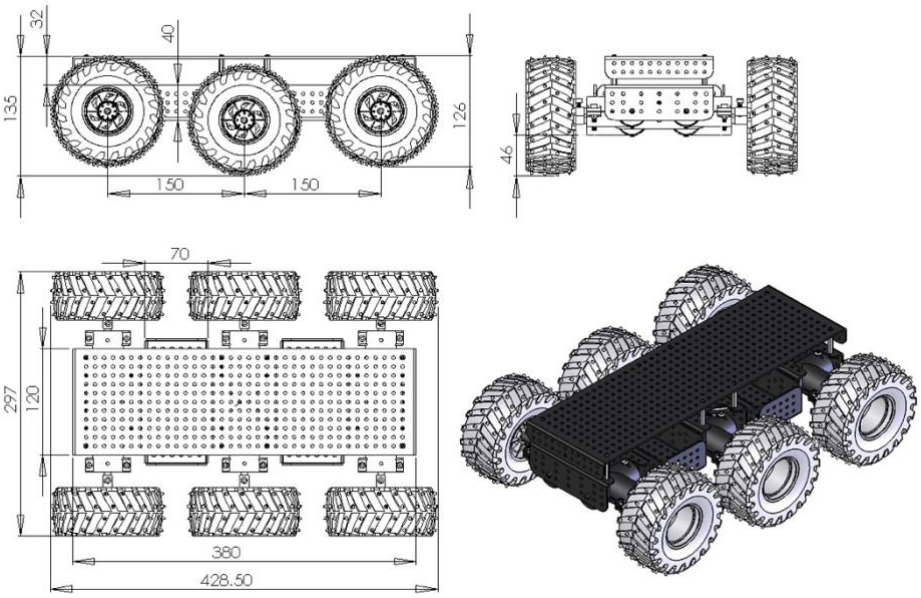
Rys. 9.2 przedstawia model pojazdu jako uproszczone odwzorowanie jego rzeczywistego odpowiednika. Obiekt posiada zielony blok główny odwzorowujący korpus, w którym umieszczone są wszystkie silniki oraz układy elektryczne. Na jego górnej ścianie znajduje się element spełniający funkcję czujnika laserowego (LIDAR), natomiast jego kołami są walce.



Rys. 9.2 Model pojazdu (Źródło: Opracowanie własne)

9.2.2 Wymiary

Wymiary elementów wirtualnych zostały odwzorowane z zachowaniem rzeczywistego rozmiaru elementów. Osiągnięto to poprzez wykorzystanie rysunków technicznych podwozia (Rys. 9.3).



Rys. 9.3 Schemat podwozia pojazdu (Źródło: http://electropark.pl/img/cms/Napedy/podwozia-ramy/7437_6.jpghttp://electropark.pl/img/cms/Napedy/podwozia-ramy/7437_6.jpg)

9.2.3 Połączenia między częściami pojazdu

Aby wszystkie użyte elementy mogły się poruszać oraz tworzyły jedną zwartą całość konieczne jest zdefiniowanie między nimi połączeń. W opisywanym pojeździe wykorzystano ich dwa rodzaje. Pierwszym z nich, jest użyte podczas spinania kół z głównym elementem, łączenie typu *„continous”.* Dzięki niemu jest możliwość obracania elementu podanego jako *child.* Dodatkowo w celu ustalenia położenia elementu potomnego względem bazowego, podanego w tagu *parent* wykorzystano tag *origin*. Definiuje on przesunięcie elementu *child* względem środka elementu zdefiniowanego w tagu *parent.* Ponadto, deklaracja *axis* definiuje osie obrotu danego elementu, korzystając przy tym z parametru xyz, który poprzez wstawienie wartości jeden wyznacza obrót wokół danej osi. (Rys. 9.4).

```
<joint type="continuous" name="rear_left_wheel">
  <origin xyz="-0.15 0.10 -0.02"/>
  <child link="rear_left_wheel"/>
  <parent link="base_link"/>
  <axis xyz="0 1 0"/>
</joint>
```

Rys. 9.4 Definicja połączenia "continous" (Źródło: Opracowanie własne)

Drugim z połączeń zastosowanych w pojeździe jest to, występujące między korpusem a liderem. Ze względu na implementację zachowania tego czujnika w Gazebo, który w odróżnieniu od rzeczywistego nie musi wykonywać obrotów, aby zbierać dane, połącznie zostało zdefiniowane jako *„fixed”*. Zapewnia ono stabilność między dwoma elementami, ponieważ typ tego połączenia polega na zablokowaniu wszystkich stopni swobody (Rys. 9.5).

```
<joint name="rplidar_joint" type="fixed">
  <axis xyz="0 1 0" />
  <origin xyz="0 0 0.07"/>
  <parent link="base_link"/>
  <child link="laser"/>
</joint>
```

Rys. 9.5 Definicja połączenia "fixed" (Źródło: Opracowanie własne)

9.2.4 Sterowanie

Aby pojazd poruszał się zgodnie z poleceniami algorytmu, należało zaimplementować mechanizm, który pozwoli przełożyć odpowiednie komendy na określony ruch kół. Do tego celu został wykorzystany dodatek, zapewniony przez ROS. Należy w nim zdefiniować jakie koła mają uczestniczyć w ruchu, poprzez podanie nazw wcześniej stworzonych złączeń, korzystających z czterech parametrów o adekwatnych nazwach. Ich rozstaw, uzupełniając wartość w tagu *wheelSeparation*. Średnicę, korzystając z *wheelDiameter* oraz nazwę kanału, z którego będą przychodzić komendy ruchu, w tagu *topicName*. Ponadto jest możliwość zdefiniowania częstotliwości, wykorzystując parametr *updateRate,* z jaką będzie odbywało się sprawdzenie czy nie został opublikowany nowy ruch. Dzięki parametrowi *torque* istnieje wariant ustalenia maksymalnego momentu obrotowego kół, którego wartość została ustalona

tak aby najlepiej odwzorować rzeczywiste zachowanie pojazdu. Dodatkową opcją, którą posiada to rozwiązanie, jest publikacja obecnych danych odometrycznych[2] pojazdu, które są tworzone na podstawie głównego elementu podanego w tagu *robotBaseFrame* a następnie wysyłanych z kanału, którego nazwa jest podawana w tagu *odometryTopic*. (Rys. 9.6).

```
<plugin name="skid_steer_drive_controller" filename="libgazebo_ros_skid_steer_drive.so">
  <updateRate>10.0</updateRate>
  <robotBaseFrame>base_link</robotBaseFrame>
  <wheelSeparation>0.20</wheelSeparation>
  <wheelDiameter>0.12</wheelDiameter>
  <torque>60</torque>
  <leftFrontJoint>front_left_wheel</leftFrontJoint>
  <rightFrontJoint>front_right_wheel</rightFrontJoint>
  <leftRearJoint>rear_left_wheel</leftRearJoint>
  <rightRearJoint>rear_right_wheel</rightRearJoint>
  <topicName>cmd_vel</topicName>
  <odometryTopic>odom</odometryTopic>
</plugin>
```

Rys. 9.6 Definicja sterowania kołami zewnętrznymi (Źródło: Opracowanie własne)

Niestety przedstawiony wyżej dodatek daje możliwość zdefiniowania ruchu tylko dwóm parom kół, co dla opisywanego pojazdu nie jest wystarczające. Problemem pojazdu staje się wykonanie obrotu, który korzystając tylko z dwóch par kół nie w pełni oddaje zachowanie rzeczywistego obiektu. Rozwiązaniem, jest dołożenie kolejnego modułu, zapewnionego przez ROS, który pozwala na podanie tożsamych informacji jak poprzednik, lecz uwzględnia tylko jedną oś (Rys. 9.7). Sterowanie nie ulegnie zmianie, dzięki zastosowaniu kanałów, które zapewniają odczytywanie znajdujących się na nich informacji przez wiele odbiorców, wykorzystując wzorzec *publish–subscribe.* Natomiast, aby nie duplikować wiadomości wysyłanych na kanał związany z obecnym położeniem pojazdu, moduł udostępnia parametr *publishOdom,* który umożliwia zabezpieczenie się przed takim zachowaniem. Dodatkową opcją tego modułu jest możliwość zdefiniowania przyśpieszenia kół, wykorzystując tag *wheelAcceleration*, którego wartośc wynosi 1 rad/s^2. Ostatnim z parametrów jest *legacyMode,*

[2] Odometria - dział miernictwa, który zajmuje się pomiarem odległości. W tym celu wykorzystuje czujniki, które określają zmianę pozycji obiektu względem pozycji startowej w czasie. Wykorzystywana często w robotyce (Źródło: https://www.wikiwand.com/pl/Odometria)

który pozwala na odwrócenie kierunku poruszania się kół, co umożliwia dopasowanie go do poprzednio zdefiniowanego modułu.

```
<plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
  <updateRate>10</updateRate>
  <leftJoint>middle_left_wheel</leftJoint>
  <rightJoint>middle_right_wheel</rightJoint>
  <wheelSeparation>0.20</wheelSeparation>
  <wheelDiameter>0.12</wheelDiameter>
  <wheelAcceleration>1</wheelAcceleration>
  <wheelTorque>60</wheelTorque>
  <commandTopic>cmd_vel</commandTopic>
  <odometryTopic>odom</odometryTopic>
  <robotBaseFrame>base_link</robotBaseFrame>
  <publishOdom>false</publishOdom>
  <legacyMode>true</legacyMode>
</plugin>
```

Rys. 9.7 Definicja sterowania kołami wewnętrznymi (Źródło: Opracowanie własne)

9.2.5 LIDAR

Ostatnim elementem, który należy umieścić w pojeździe jest LIDAR. Jak zostało to już wspomniane, znajduje się on na korpusie i jest sztywno do niego przymocowany. Aby mógł on zbierać dane, zastosowano rozwiązania należące do Gazebo, natomiast za publikowanie zgromadzonych informacji odpowiedzialny jest dodatek zapewniony przez ROS. Czujnik w parametrze *samples* pozwala na zdefiniowanie ilości wiązek lasera, służących do pomiaru odległości. Ich zakresu działania poprzez uzupełnienie parametru *min_angle* oraz *max_angle,* których jednostką są radiany, a wartości obecnie wynoszą odpowiednio zero oraz dwa pi, co pozwala na zbieranie informacji skanując pełne otoczenie pojazdu. Ponadto istnieje możliwość zdefiniowania długości wiązki wykorzystując parametry zdefiniowane w tagu *range,* poprzez uzupełnienie wartość *min* oraz *max,* która obecnie jest zgodna ze specyfikacją czujnika odległości, jaki został użyty w rzeczywistym obiekcie. Ponadto w tagu *topicName,* zdefiniowano kanał, na którym są publikowane zebrane informacje (Rys. 9.8).

```
<sensor type="gpu_ray" name="rp_lidar_a1">
    <pose>0 0 0.15 0 0 0</pose>
    <visualize>false</visualize>
    <update_rate>10</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>36</samples>
          <min_angle>0</min_angle>
          <max_angle>6.28318530718</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.15</min>
        <max>12.0</max>
      </range>
    </ray>
    <plugin name="lidar_scan" filename="libgazebo_ros_gpu_laser.so">
      <topicName>scan</topicName>
    </plugin>
  </sensor>
```

Rys. 9.8 Definicja czujnika LIDAR (Źródło: Opracowanie własne)

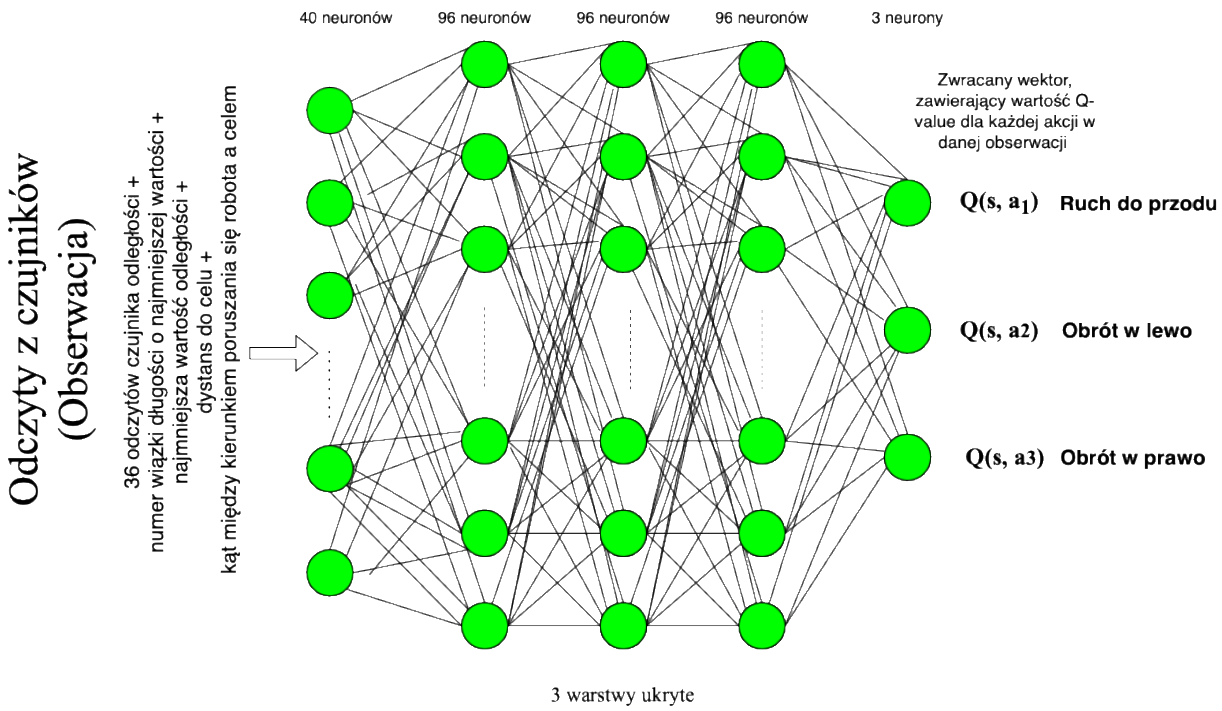
10 Implementacja algorytmu trenującego sieć neuronową

Aby zrealizować postawiony cel, należało połączyć wszystkie dotychczas stworzone systemy w jedną spójną całość. Wykorzystano do tego język Python oraz API dostarczone przez ROS, dzięki któremu istnieje możliwość pobierania danych, które model zebrał przy użyciu czujników, jak również wysyłania poleceń, określonych z wykorzystaniem sztucznej inteligencji, które agent powinien wykonać.

10.1 Implementacja modelu sieci

Proces uczenia został zrealizowany przy użyciu dwóch identycznych modeli sieci neuronowych. Składają się one z warstwy wejściowej zawierającej czterdzieści neuronów, które odpowiadają danym pobranym z obserwacji, trzech warstw ukrytych, wykorzystujących funkcję aktywacji „ReLu”, każda po 96 neuronów oraz warstwy wyjściowej składającej się z ilości neuronów odpowiadającej liczbie akcji wynoszącej trzy, z liniową funkcją aktywacji (Rys. 10.1). Struktura ta powstała bazując na sieciach neuronowych, stosowanych w podobnej tematyce [29]. Jednakże parametry modelu były dopasowywane indywidualnie do problemu poprzez empiryczne testy.

Jego podstawowa wersja, uczestniczy w określaniu ruchów pojazdu oraz wykorzystywana jest w procesie uczenia, jako sieć neuronowa służąca do przewidywania ruchów w oparciu o stan obecny. Natomiast dodatkowy model, zwany docelowym, przewiduje akcję na podstawie drugiego odczytu z czujników, który został zapisany w analizowanym rekordzie, umieszczonym w pamięci ruchów. Aktualizacja tej sieci odbywa się każdorazowo, poprzez przepisanie wartości wag z modelu podstawowego do docelowego. Występuje ona po zakończeniu epizodu (epizod – pojedyncza próba zdobycia celu przez robota, przerywana w przypadku uderzenia w przeszkodę lub przekroczenia liczby wykonanych kroków). Zastosowanie dwóch sieci neuronowych w implementacji algorytmu *Deep Q-learning* powoduje zwiększenie stabilności i efektywności podczas procesu uczenia.



Rys. 10.1 Zaimplementowany model sieci neuronowej (Źródło: Opracowanie własne)

10.2 Rozdzielenie stanu eksploracji i eksploatacji

Schemat działania algorytmu *Q-learning* dzieli się na dwa stany, które zostały omówione w rozdziale 5.2. Aby odróżnić eksplorację, gdzie większość ruchów jest przypadkowa, a eksploatację, w której akcje przesyłane do pojazdu są wynikiem działania sieci neuronowej, zastosowano zmienną o nazwie „Epsilon". Jej wartość przed rozpoczęciem symulacji wynosi jeden i jest sukcesywnie zmniejszana o jeden procent wartości co każdy wykonany epizod. To rozwiązanie pozwala na płynne przejście z fazy eksploracji do eksploatacji i jest realizowane przez Fragment kodu 10.1, w którym wylosowana liczba z przedziału od zera do jeden jest porównywana z wartością zmiennej Epsilon. Jeżeli jest ona większa, to akcja jest definiowana przez numer argumentu, który posiada największą wartość, z wektora *Q-values* będącego wynikiem działania sieci. Natomiast w przeciwnym wypadku, numer ten, jest losową liczbą z przedziału tych, które definiują akcję.

```
if np.random.random() > self.epsilon:
    q_values = self.model.predict(current_lidar_data)
    action = np.argmax(q_values)
else:
    action = np.random.randint(0, self.n_actions)
```

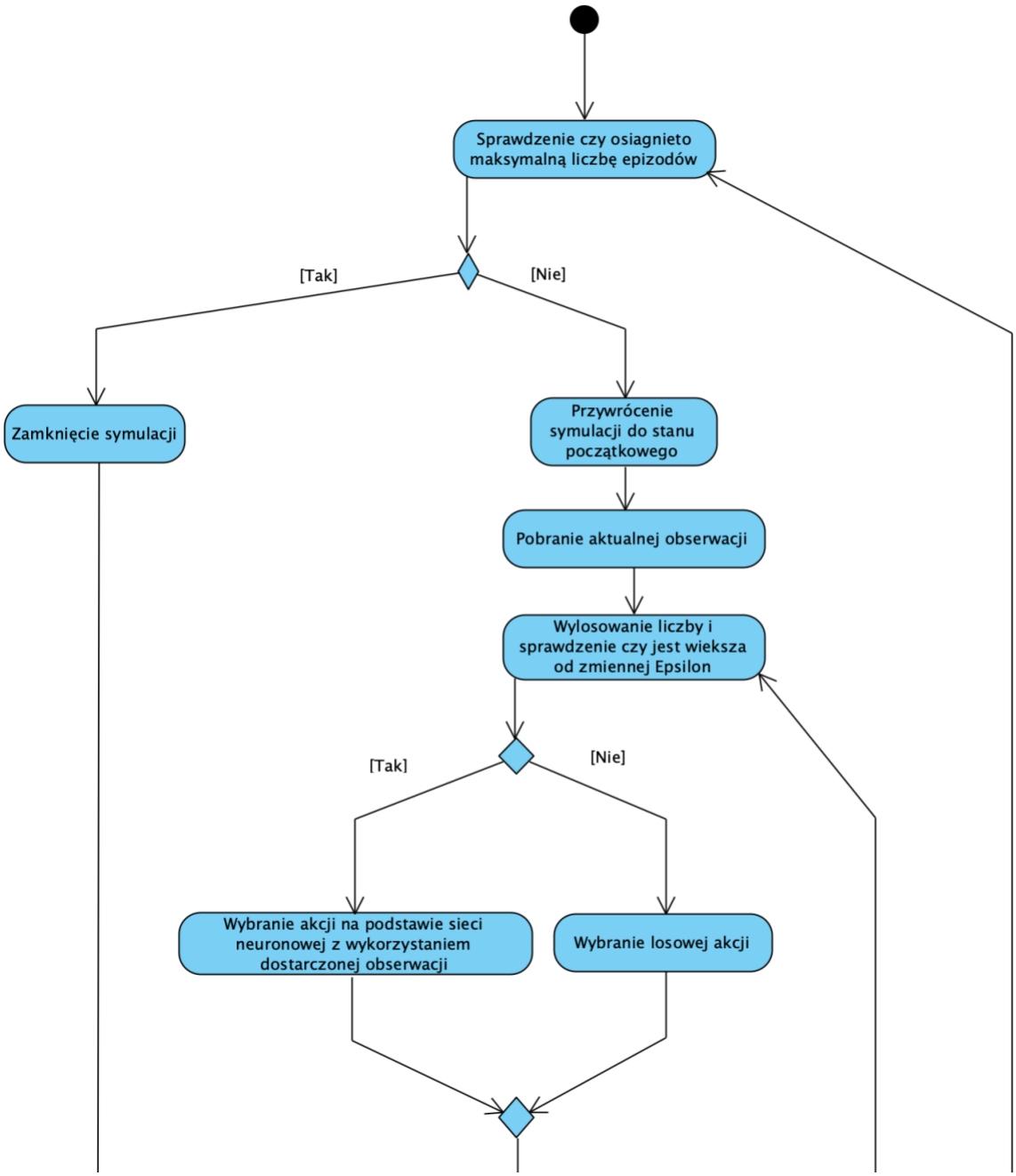
Fragment kodu 10.1 Rozdzielenie eksploracji i eksploatacji (Źródło: Opracowanie własne)

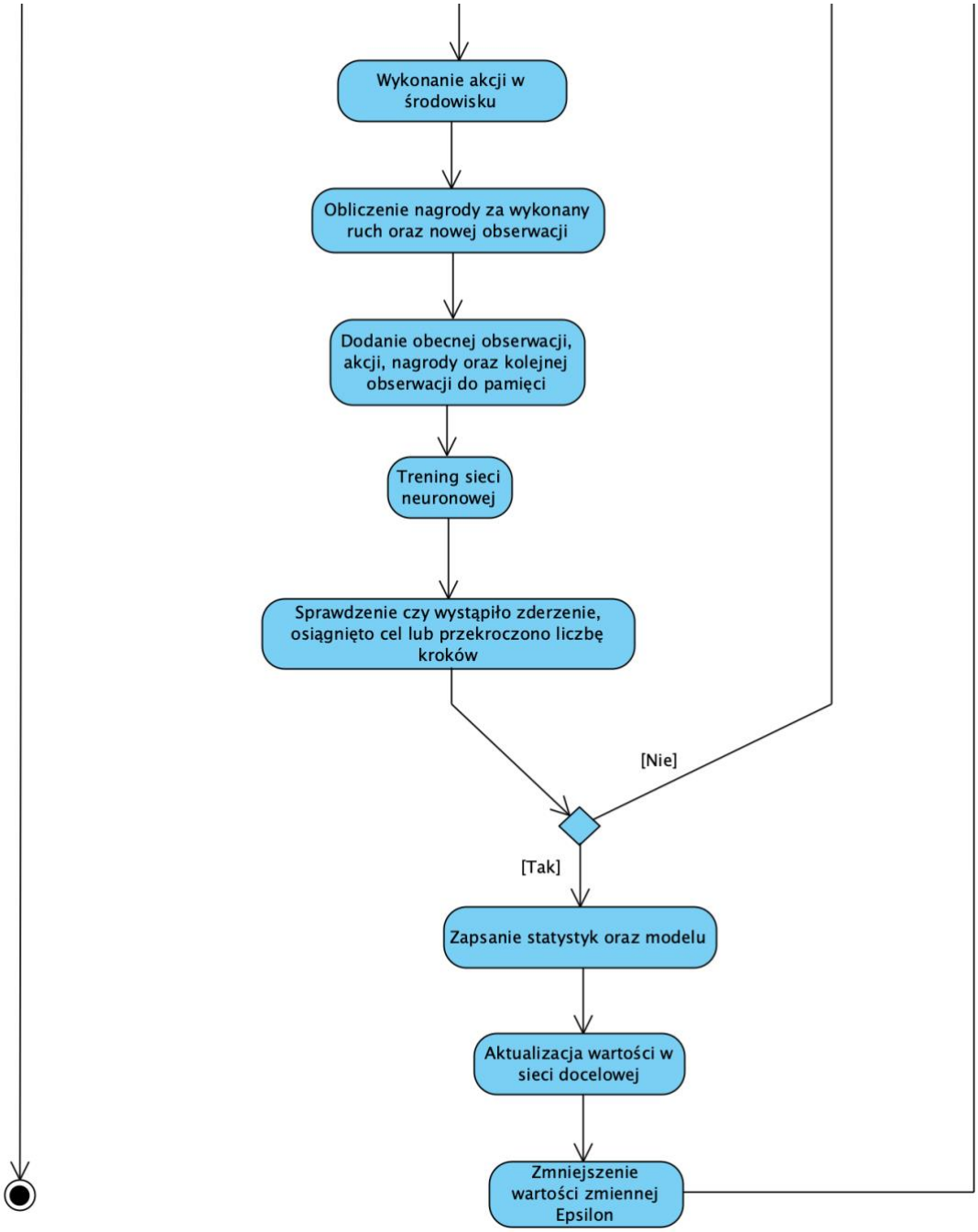
10.3 Ogranicznenie symulacji

Aby zapobiec działaniu symulacji w nieskończoność, wprowadzono dwa ograniczenia. Pierwsze z nich dotyczy ilości wykonanych epizodów i powodem jego powstania był bardzo mały wzrost skuteczności modelu, a nawet jej spadek, który występował po przekroczeniu granicy około tysiąca epizodów. Ograniczenie, z wartości dwóch tysięcy, umożliwiało poprawne nauczenie modelu odpowiednich zachowań, jak również ograniczało duże zużycie zasobów, nieodzwierciedlone progresem skuteczności. Drugim z obostrzeń, był limit kroków jakie mógł wykonać agent w każdym z epizodów. Genezą jego powstania było przerwanie epizodu, opierającego się głównie na skręcaniu, co było powodowane błędnie określoną nagrodą, przyznawaną za każdy ruch. Ograniczenie to wynosiło około sześciu tysięcy akcji.

10.4 Przedstawienie poszczególnych etapów symulacji

W celu zaprezentowania sekwencji kroków wykonywanych przez opisywane rozwiązanie, zaprojektowano diagram czynności przedstawiony na Rys. 10.2

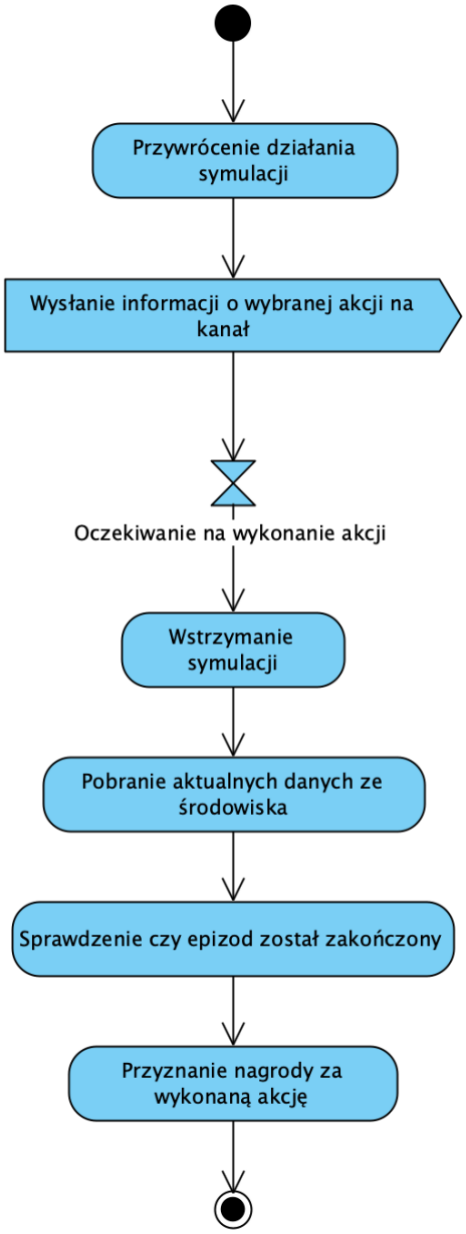

Sprawdzenie czy osiagnieto maksymalną liczbę epizodów
[Tak]
[Nie]
Zamknięcie symulacji
Przywrócenie symulacji do stanu początkowego
Pobranie aktualnej obserwacji
Wylosowanie liczby i sprawdzenie czy jest wieksza od zmiennej Epsilon
[Tak]
[Nie]
Wybranie akcji na podstawie sieci neuronowej z wykorzystaniem dostarczonej obserwacji
Wybranie losowej akcji



Rys. 10.2 Diagram czynności przedstawiający proces trenowania modelu (Źródło: Opracowanie własne, przy użyciu Visual Paradigm na licencji z Akademii Górniczo-Hutniczej)

10.5 Pobranie danych z czujników i wykonanie akcji

Kluczowym etapem działania całego rozwiązania jest ten, który udostępnia możliwość wysłania akcji do agenta, jak również pobrania od niego danych. Jego schemat działania został przedstawiony na Rys. 10.3.



Rys. 10.3 Implementacja metody pozwalającej na wykonanie akcji i pobranie danych (Źródło: Opracowanie własne z wykorzystaniem Visual Paradigm na licencji Akademii Górniczo-Hutniczej)

Etap ten został zaimplementowany jako metoda, wykorzystująca funkcjonalności udostępniane przez różne systemy. Pozwala użytkownikowi jednym jej wywołaniem na zarządzanie wykonywaniem symulacji, poprzez jej pauzowanie i uruchamianie, wysłanie wybranej akcję do pojazdu, pobranie danych z czujników, sprawdzenie zakończenia epizodu oraz obliczenie nagrody. Schemat jej działania przypomina wzorzec projektowy „Fasada", który ułatwia korzystanie z systemu, udostępniając jeden prosty interfejs, który maskuje bardziej wymagające użycie metod. Poszczególne części przedstawionego etapu zostaną omówione w poniższych rozdziałach.

10.5.1 Zatrzymanie/Wznowienie symulacji

Manipulowanie symulacją jest możliwe dzięki zastosowaniu serwisów dostarczonych przez ROS. Ich wykorzystanie daje pewność, że dane pochodzące z różnych czujników zawsze będą dokładnie z tego samego odcinka czasu, jak również eliminują one ewentualne własnowolne poruszanie się pojazdu, gdy algorytm definiujący ruch, zapisuje dane do pliku lub trenuje model, co może zająć dłuższy okres czasu.

10.5.2 Wykonanie akcji

Między zatrzymaniem, a wznowieniem symulacji, następuje wykonanie akcji, które jest realizowane z wykorzystaniem Fragment kodu 10.2. Jego działanie opiera się na utworzeniu obiektu oraz nadaniu mu odpowiednich wartości definiujących ruch, w zależności od wyznaczonej akcji. Kolejnym etapem, jest przekazanie gotowego obiektu do metody move_base, która publikuje ruch na kanale obsługiwanym przez dodatek opisany w rozdziale 9.2.4, a następnie przez trzy dziesiąte sekundy czeka na jego wywołanie. Po tym czasie następuje odczytanie wskazań czujników oraz wykonanie kolejnej akcji. Taka implementacja ruchu pozwala na dowolne dobranie kąta obrotu, ponieważ przy bardzo małej korekcji ruchy wystarczy podać jedno lub dwa polecenia skrętu, natomiast w sytuacji wymagającej półpełnego obrotu, ciąg tych poleceń może być dużo dłuższy, co będzie skutkowało obrotem o dość duży kąt.

```
def _set_action(self, action):
    cmd_vel_value = Twist()

    if action == 0:
        cmd_vel_value.linear.x = self.linear_forward_speed
        cmd_vel_value.angular.z = 0

    elif action == 1:
        cmd_vel_value.linear.x = 0
        cmd_vel_value.angular.z = self.angular_speed

    elif action == 2:
        cmd_vel_value.linear.x = 0
        cmd_vel_value.angular.z = -1 * self.angular_speed

    self.move_base(cmd_vel_value)
```

Fragment kodu 10.2 Wykonanie akcji (Źródło: Opracowanie własne)

10.5.3 Pobieranie aktualnych danych ze środowiska

Kolejnym elementem wyszczególnionym w diagramie czynności jest pobieranie danych ze środowiska. Pierwszym etapem uzyskiwania danych jest filtracja odczytów uzyskanych przy użyciu czujnika laserowego. Odbywa się to z wykorzystaniem kodu, przedstawionego poniżej (Fragment kodu 10.3), w którym dane, znajdujące się poza zakresem minimalnym lub maksymalnym, są odpowiednio zastępowane, tak aby sieć neuronowa nie dostawała sprzecznych informacji. Ponadto odczyty zaokrąglane są od części setnych metra, ponieważ nie ma potrzeby używania większej precyzji. Na dodatek, dane zawsze przychodzą posortowane względem kąta z jakiego zostały pobrane, co również jest brane pod uwagę przy formowaniu ostatecznej obserwacji.

```
def filter_lidar_data(self,data):
    filtred_data = []

    max_laser_value = data.range_max
    min_laser_value = data.range_min

    for i, item in enumerate(data.ranges):
        if item == float ('Inf') or numpy.isinf(item):
            filtred_data.append(max_laser_value)
        elif numpy.isnan(item):
            filtred_data.append(min_laser_value)
        else:
            item = np.round(item, self.dec_obs)
            filtred_data.append(item)

    return filtred_data
```

Fragment kodu 10.3 Filtracja danych pochodzących z czujnika LIDAR (Źródło: Opracowanie własne)

Dodatkową możliwością wykorzystywaną do pozyskania danych jest obliczenie odległości dzielącej w linii prostej pojazd oraz cel. Jest to możliwe dzięki wysyłanym przez układ sterowania informacjom, a docelowo poprzez bazowanie na odczytach z systemu GPS. Ponadto, w danych otrzymywanych z symulacji znajduje się informacja o orientacji pojazdu

względem osi X, co w świecie rzeczywistym zostanie zastąpione poprzez odczyty z magnetometru. Zestawiając te informacje z poprzednio opisywanymi, istnieje możliwość obliczenia kąta znajdującego się między kierunkiem poruszania pojazdu a celem.

Tak przygotowany zestaw danych tworzy pakiet obserwacji, który jest używany w sieci neuronowej, celem rozstrzygnięcia najbardziej odpowiedniego posunięcia (Fragment kodu 10.4).

```
def _get_obs(self):
    laser_scan = self.laser_scan
    discretized_observations = self.discretize_observation(laser_scan)
    discretized_observations.append(np.argmin(discretized_observations)) # Numer odczytu posiadającego najmniejsza wartosc
    discretized_observations.append(np.min(discretized_observations)) # Odleglosc do najblizszego obiektu
    discretized_observations.append(self.__get_distance_to_goal()) # Dystans do celu
    discretized_observations.append(self.get_angle_to_goal()) # Kat miedzy pojazdem a celem
    return discretized_observations
```

Fragment kodu 10.4 Zestaw obserwacji (Źródło: Opracowanie własne)

10.5.4 Sprawdzenie czy epizod został zakończony

Jedną z końcowych operacji przedstawionej wcześniej metody wykonywania ruchu, jest sprawdzenie czy epizod dobiegł końca. Na tym etapie może się to dokonać tylko na dwa sposoby. Pierwszym z nich jest znalezienie odczytu odległości, który zawiera mniejszą wartość niż wcześniej ustalona jako minimalna. Oznacza to, że pojazd znalazł się zbyt blisko przeszkody i istnieje ryzyko kontaktu. Drugim sposobem, jest osiągnięcie zamierzonego punktu. Aby dokonać jego weryfikacji korzysta się z odczytów położenia. Jeżeli współrzędne pojazdu pokrywają się z tymi definiującymi cel, to epizod zostaje zakończony ze względu na zdobycie celu i jest losowany nowy punkt, do którego pojazd ma podążać (Fragment kodu 10.5).

```
def _is_done(self, observations):
    self.goal_achived = observations[-2] < self.min_goal_distance
    self.episode_done = self.min_range > observations[-3] > 0

    if self.episode_done:
        return True
    elif self.goal_achived:
        self.goal_pos_x = random.randint(X_FROM_GOAL, X_TO_GOAL)
        self.goal_pos_y = random.randint(Y_FROM_GOAL, Y_TO_GOAL)
        return True
    else:
        return False
```

Fragment kodu 10.5 Weryfikacja zakończenia epizodu (Źródło: Opracowanie własne)

10.5.5 Przyznawanie nagrody za wykonaną akcję

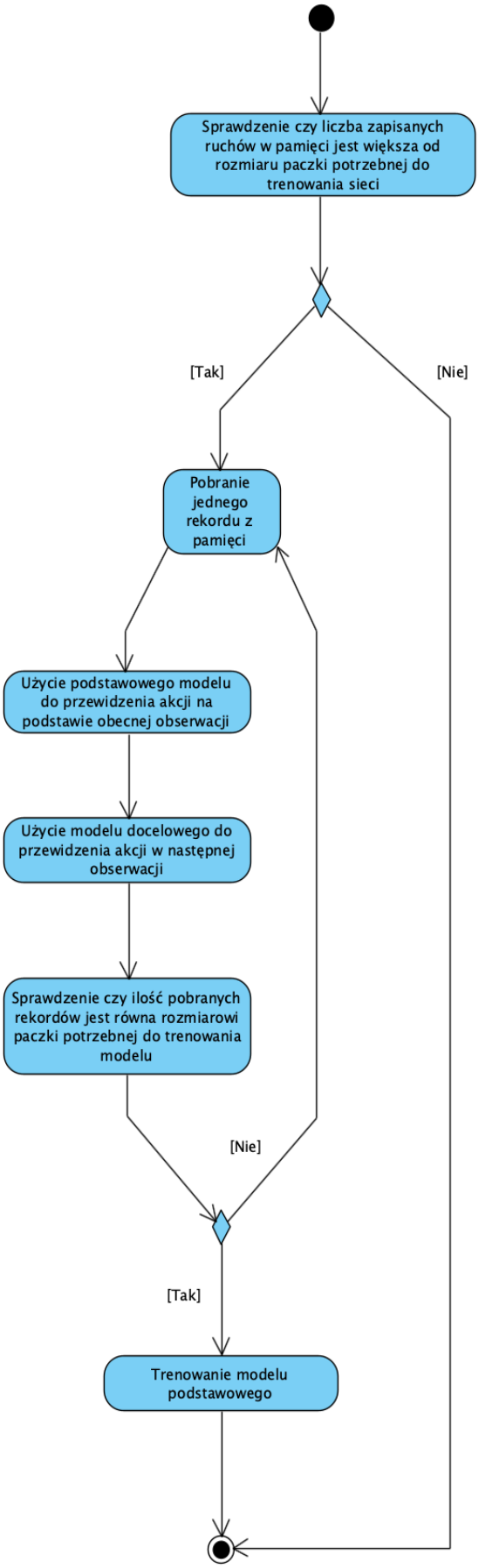
Etapem finalnym jest ocena wykonanej przez model akcji. Wartość zwracana z tej metody definiuje zarówno szybkość jak i skuteczność uczenia. Źle dobrana funkcja nagrody może powodować błędne dążenie do celu lub jego całkowite pominięcie. Niemniej jednak, kluczowym elementem tej metody, jest przyznawanie odpowiedniej wartości liczbowej na zakończenie epizodu. Wartość ta, różni się w zależności od powodu jego zakończenia. Gdy pojazd znajdzie się zbyt blisko przeszkody, otrzymuje dużą wartość ujemną, potocznie nazwaną karą, którą nie jest w stanie otrzymać podczas normalnego poruszania się po środowisku. Natomiast, gdy cel zostanie osiągnięty, przyznawana jest duża wartość dodatnia, zwana nagrodą, również nieosiągalna podczas zwykłego poruszania. Nie mniej ważnym elementem jest nagroda, otrzymywana po każdorazowym wykonaniu ruchu i to dzięki niej, a dokładnie dzięki jej wartości pojazd wie, że wykonuje poprawną akcję lub wręcz przeciwnie (Fragment kodu 10.6). Istnieje również podejście przyznające nagrodę tylko na końcu epizodu, lecz wymaga ono dużej ilości czasu poświęconego na uczenie sieci neuronowej oraz nie zawsze może przynieść pożądane efekty.

```
def _compute_reward(self, observations, done, action):
    angle_to_goal = observations[-1]

    if self.goal_achived:
        return self.goal_achived_reward
    elif self.episode_done:
        return self.end_episode_points

    reward = 0

    if action == 0:
        reward = math.cos(angle_to_goal)

    return np.round(reward, 2)
```

Fragment kodu 10.6 Przyznawanie nagrody (Źródło: Opracowanie własne)

10.6 Trenowanie modelu

Metoda pozwalająca na doskonalenie modelu stanowi niejako serce całego systemu. To dzięki niej sieć neuronowa z czasem „nabywa umiejętności" pozwalających, na podstawie danych wejściowych, określać adekwatne zachowania przybliżające pojazd do osiągniecia celu. Schemat jej działania został przedstawiony z wykorzystaniem diagramu czynności na Rys. 10.4.



Rys. 10.4 Schemat metody wykorzystywanej podczas uczenia modelu (Źródło: Opracowanie własne)

To właśnie ta metoda implementuje algorytm *Q-learning* w celu osiągnięcia zamierzonych efektów. Fragment kodu 10.7 przedstawia implementację części wzoru omówionego w rozdziale 5.2, który jest wykorzystywany podczas przygotowywania zestawu danych służących do uczenia modelu. Parametry zastosowane do działania wyżej przytoczonego wzoru wynoszą odpowiednio: α–0.001, γ–0.99. Pierwszy z nich wykorzystywany jest jako parametr w algorytmie optymalizacyjnym sieci. Natomiast drugi, stosowany jest w poniższym fragmencie kodu i zgodnie z fragmentem wzoru$[R + \gamma\, maxQ(s', a')]$ służy do wyznaczania ważność nagrody z kolejnego stanu. Ich wartości zostały dobrane w sposób empiryczny.

```
for i in range(self.batch_size):
    state = mini_batch[i][0]
    action = mini_batch[i][1]
    reward = mini_batch[i][2]
    next_state = mini_batch[i][3]
    done = mini_batch[i][4]

    q_value = self.model.predict(state)

    next_target = self.target_model.predict(next_state)

    if not done:
        next_q_value = reward + self.gamma * np.amax(next_target)
    else:
        next_q_value = reward
```

Fragment kodu 10.7 Implementacja wzoru wykorzystywanego w *Q-learning* (Źródło: Opracowanie własne)

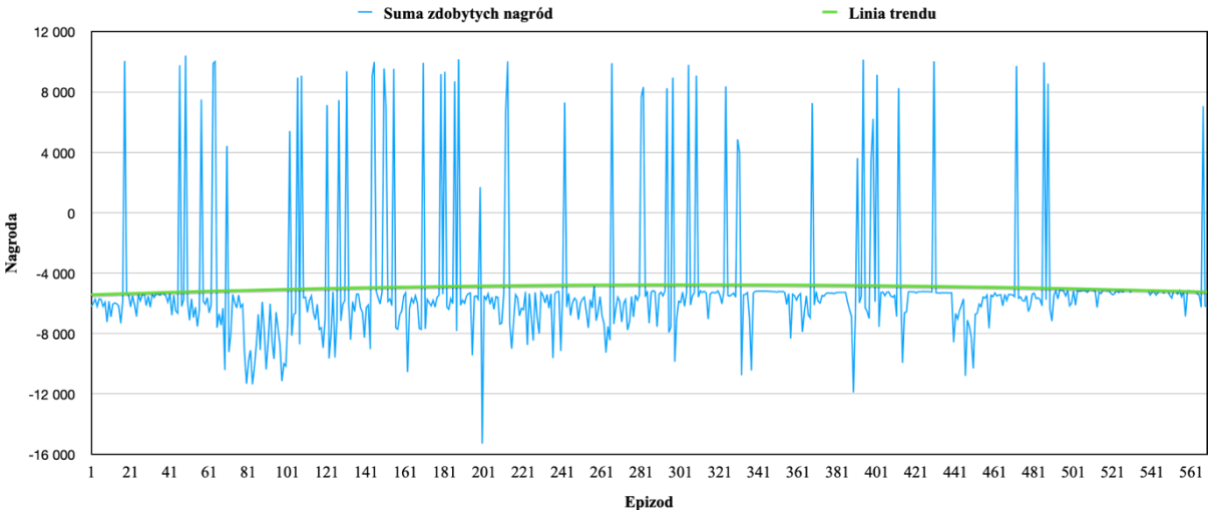
11 Modyfikacje funkcji nagradzającej agenta

Kluczowym elementem algorytmu *Q-learning* jest nagroda jaką agent dostaje za wykonanie ustalonej czynności. To dzięki niej model potrafi ocenić pożądane zachowanie, jak również opracować schemat ruchów, który pozwoli osiągnąć zamierzony cel.

Błędem, poczynionym przy pierwszych próbach osiągnięcia celu, była równomierna ocena każdego ruchu robota. Zazwyczaj prowadziło to do gratyfikacji skręcania odbywającego się w miejscu, przez co robot nie był narażony na zderzenie z przeszkodą, co wiązałoby się z przyznaniem dużej kary. Takie zachowanie powodowało długi czas trwania epizodu z brakiem konkretnych efektów. Rozwiązaniem było rozróżnienie dwóch typów poruszania się, stosując adekwatne nagrody dla każdego z nich.

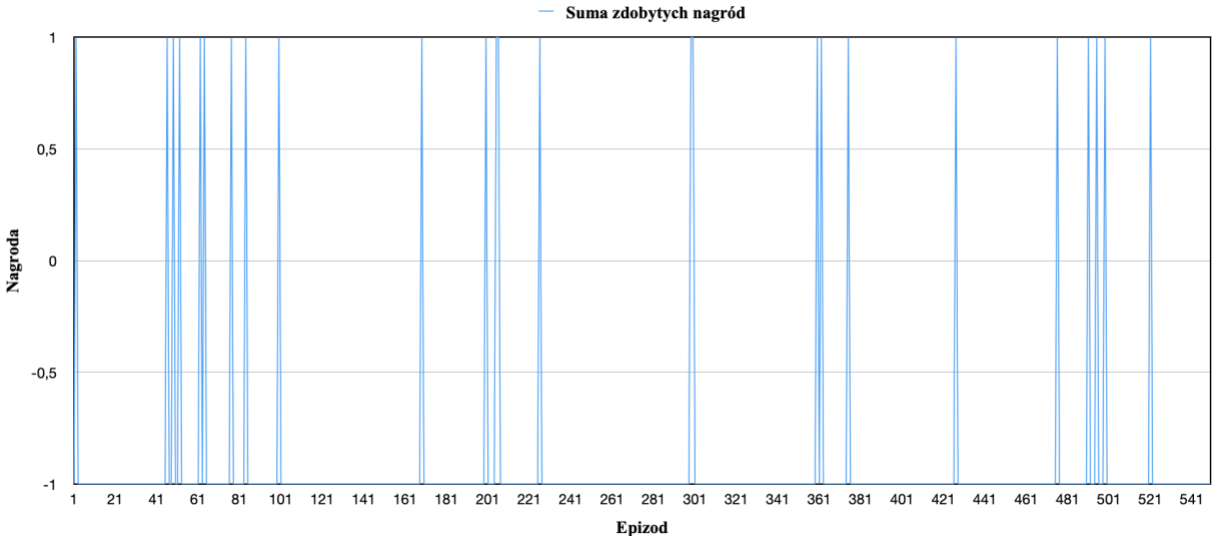
Kolejnym błędem, występującym podczas tworzenia takiej funkcji, było zbyt wymagające zadanie, za które była przyznawana nagroda. Przykładem tego, jest gratyfikowanie agenta tylko jeżeli zmniejszył odległość do celu, przy czym wartość tej odległości jest zapisywana i nagroda przyznawana jest tylko gdy ulegnie ona zmniejszeniu. Takie podejście, dobrze obrazuje przykład, w którym początkowa odległość do celu wynosi dziesięć, robot porusza się w zamierzonym kierunku zmniejszając tę odległość do dziewięciu, za co dostaje nagrodę. Następnie, robi obrót i wraca do miejsca startowego zwiększając odległość do pierwotnej jej wartości. Raz jeszcze wykonuje ruch do celu, zmniejszając odległość o jeden, lecz tym razem nie dostaje nagrody, ponieważ najmniejszą zapisaną wartością odległości przez cały czas było dziewięć, więc tym ruchem pojazd nie wykonał żadnego progresu.

W początkowych fazach, większość ruchów pojazdu jest przypadkowa i nie zawsze podąża on w dobrym kierunku. Powoduje to nagromadzenie neutralnych ruchów w pamięci algorytmu, przez co robot podczas uczenia nie ma możliwości maksymalizacji poprawnych akcji. Wyniki takiego podejścia zostały przedstawione na Wykres 11.1. Można tam zobaczyć proces odwrotny do zamierzonego, gdzie linia trendu, zamiast rosnąć, spada, powodując dość rzadkie osiąganie celu przez pojazd, a co za tym idzie, bardzo powolną naukę prawidłowych zachowań.



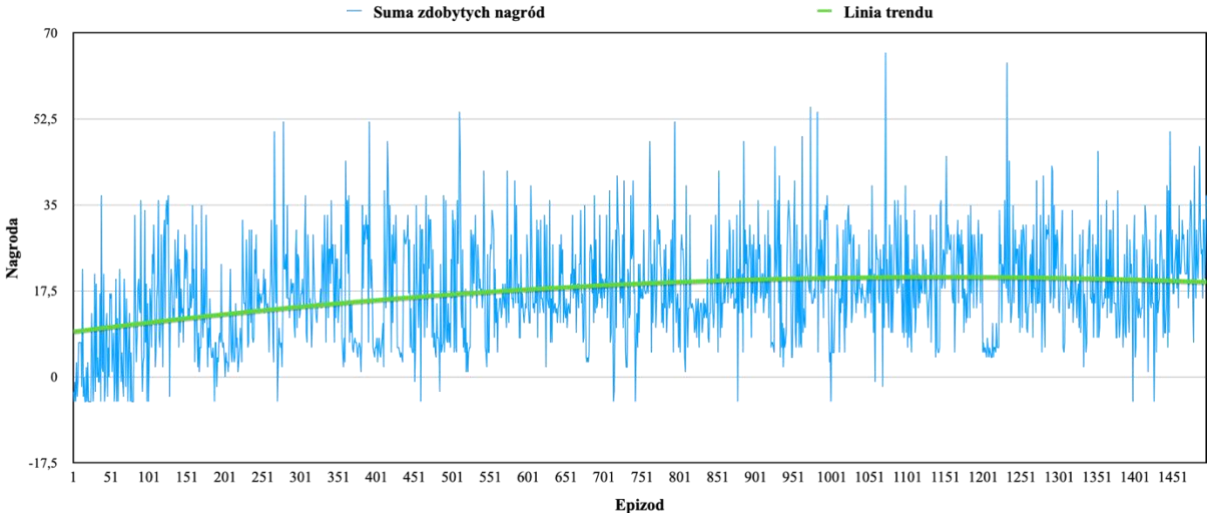
Wykres 11.1 Ilość sumarycznie zdobytych nagród w każdym z epizodów dla ściśle określonej funkcji (Źródło: Opracowanie własne)

Szczególnym przypadkiem, również wpisującym się w założenia zbyt wymagającego zadania, jest zdefiniowanie wartości przyznawanej tylko przy zakończeniu epizodu. Agent dostaje nagrodę, jeśli osiągnął cel, lub karę, gdy źle wykonał zadanie. Tak zdefiniowana funkcja, powoduje dużą dezorganizację w początkowych ruchach pojazdu, ponieważ otrzymując taką samą wartość za każdy ruch, nie jest on w stanie ocenić, czy był on dobry czy zły. Konsekwencją takich działań jest bardzo długi czas uczenia, jak również trwania epizodu, ponieważ pojazd będąc z dala od przeszkód wykonuje losową sekwencję ruchów i zwykle kończy się to przekroczeniem dopuszczalnej ilości kroków. Wykres przedstawiający rozkład nagród w takim podejściu został przedstawiony na Wykres 11.2.



Wykres 11.2 Ilość sumarycznie zdobytych nagród w każdym z epizodów dla bonusu przyznawanego pod koniec podejścia (Źródło: Opracowanie własne)

Najlepszym podejściem do rozwiązania tego problemu okazała się funkcja przyznająca nagrodę tylko gdy pojazd wykonywał ruch do przodu. Do obliczenia jej wartości został wykorzystany kąt między pojazdem a celem, który odpowiednio wynosił zero, gdy pojazd podążał w linii prostej do zadanego punktu oraz π bądź -π, gdy poruszał się w przeciwnym kierunku. Taki zakres argumentów idealnie pasuje do funkcji cosinus, która dla zera przyjmuje wartość jeden stopniowo przechodząc w wartości ujemne, a kończąc na minus jeden. Dzięki temu, agent może łatwo ocenić pożądane zachowanie co w konsekwencji sprawia, że proces uczenia sieci neuronowej przebiega dużo szybciej. Zastosowanie takiej wersji nagrody pozwoliło na to, że model podczas swoich najlepszych podejść w liczbie sto, osiemdziesiąt osiem razy osiągnął cel. Wykres prezentujący otrzymaną nagrodę w każdym z epizodów został przedstawiony na Wykres 11.3.



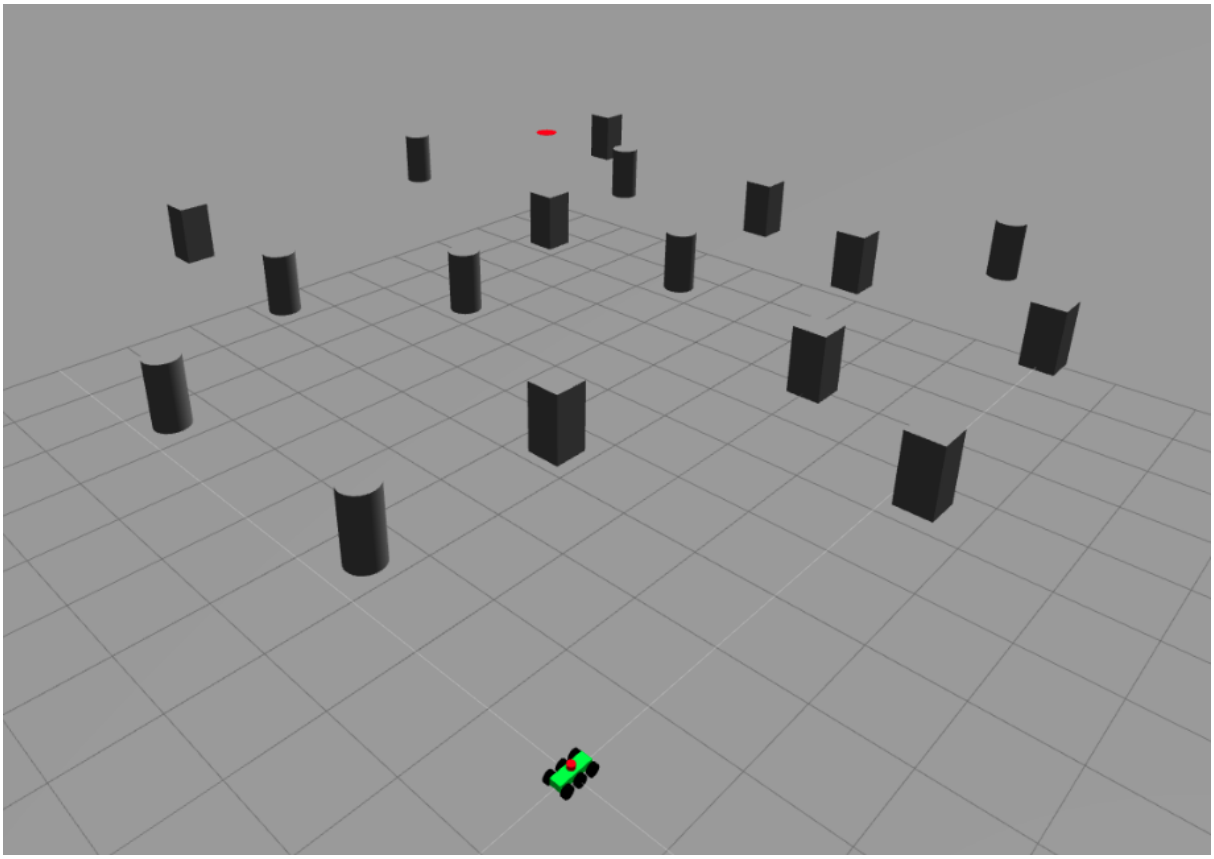
Wykres 11.3 Ilość sumarycznie zdobytych nagród w każdym z epizodów dla funkcji cosinus (Źródło: Opracowanie własne)

12 Testy rozwiązania

W celu uzyskania jak najbardziej optymalnego algorytmu, poddawano go weryfikacji zarówno podczas implementacji jak i po finalizacji prac nad całym projektem. Testy opierały się głownie na umieszczeniu robota w wirtualnie stworzonym środowisku odmiennym od tego, w którym doskonalił swoje umiejętności.

12.1 Środowisko testowe

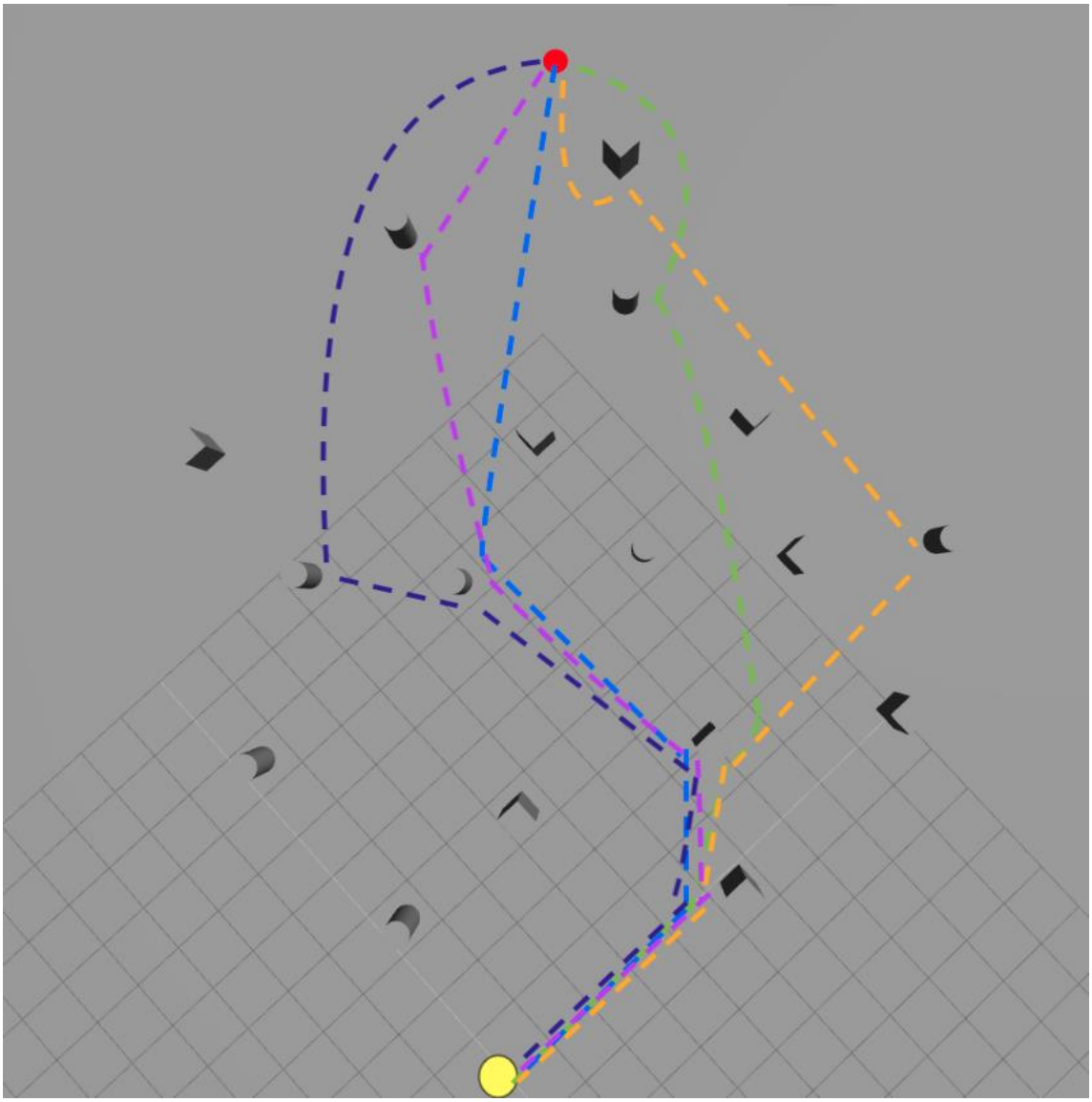
Do przeprowadzenia testów opracowano specjalne środowisko, zawierające losowo ustawione przeszkody, mające na celu utrudnienie poruszania się agenta w drodze do celu. Aby wyróżnić punkt docelowy, w jego miejscu ustawiono czerwone walec, który posiada wysokość mniejszą niż poziom działania czujnika laserowego, dzięki czemu nie zostanie on odebrany jako przeszkoda. Opisywane środowisko zostało przedstawione na Rys. 12.1 i w przeciwieństwie do planszy treningowej nie posiada ono granicy w postaci ścian wyznaczających jego rozmiar, co jest założeniem intencjonalnym, aby zweryfikować działanie w środowisku otwartym, które nie posiada ograniczeń.



Rys. 12.1 Środowisko testowe pojazdu (Źródło: Opracowanie własne)

12.2 Finalne testy rozwiązania

Po opracowaniu ostatecznego rozwiązania wykonano testy, aby sprawdzić poprawność działania algorytmu. Wykazały one zadawalającą skuteczność wynoszącą osiemdziesiąt procent, liczoną w postaci przejazdów, które dotarły do celu bez kontaktu z przeszkodami. Pojazd nie poruszał się jednym torem prowadzącym do celu, lecz zdefiniował kilka różnych dróg zaznaczonych na Rys. 12.2. Jest to spowodowane różnymi kątami natarcia na poszczególne przeszkody, jak również zaimplementowanym do czujnika odległości, zakłóceniom.



Rys. 12.2 Drogi wyznaczone przez pojazd podczas testów (Źródło: Opracowanie własne)

13 Podsumowanie

Współcześnie, niezależnie od warunków panujących na świecie, zapewnienie bezpieczeństwa jest jednym z najwyższych priorytetów niezależnie od warunków panujących na świecie. Wykorzystanie pojazdów autonomicznych pozwala zwiększyć jego poziom, dlatego z roku na rok, ta tendencja przybiera sile. Powoduje to rozwój algorytmów umożliwiających ich poprawne sterowanie, co jest elementem kluczowym, pozwalającym spełnić postawione im zadania zarówno w przemyśle jak i codziennym użytkowaniu.

Celem, jaki został postawiony w niniejszej pracy, było opracowanie algorytmu sterowania pojazdem autonomicznym z wykorzystaniem mechanizmów sztucznej inteligencji w oparciu o dane odczytane z czujników. Do jego zrealizowania, po analizie możliwych rozwiązań, wykorzystano *Deep Q-learning*. Metoda ta wykorzystuje sieci neuronowe, aby przewidzieć najbardziej odpowiednie posunięcie, bazując na danych, które zostały pobrane z otoczenia. W celu przeprowadzenia procesu uczenia, została stworzona wirtualna przestrzeń, jak również zbudowano model pojazdu, który jest odwzorowaniem projektu realizowanego w innej pracy inżynierskiej. Poprzez wykonanie wyżej wymienionych czynności, zrealizowano cele szczegółowe, postawione na początku tej pracy, co równocześnie pozwoliło spełnić jej główne założenie.

Opracowane rozwiązanie zostało poddane testom, które przeprowadzono na specjalnie do tego przygotowanym środowisku. Pozwoliło to uzyskać obraz działania algorytmu w nieznanych dotąd warunkach, potwierdzając jednocześnie jego umiejętności adaptacyjne. Pierwotnie, testy miały odbyć się z wykorzystaniem pojazdu, który posłużył jako wzór do stworzenia modelu wirtualnego, lecz obecnie panująca pandemia i nakaz ograniczenia kontaktów uniemożliwiły wykonanie robota na czas.

Zaprezentowane rozwiązanie pozwala na dalszy rozwój algorytmu, dzięki czemu będzie on mógł bardziej precyzyjnie dobierać odpowiedni ruch lub zwiększyć zakres wykonywanych akcji. Elementem, który mógłby znacznie ubogacić obecne rozwiązanie jest kamera. Dzięki niej możliwe byłoby kontrolowanie otaczających obiektów, co znalazłoby szerokie zastosowanie zarówno w pojazdach uczestniczących w ruchu drogowym, podczas rozpoznawania znaków drogowych, jak również maszynach przemysłowych, przewożących towary w miejsce zdefiniowane na ich etykietach.

Bibliografia

[1] https://www.parp.gov.pl/storage/publications/pdf/Czwarta-rewolucja-przemysowa_200730.pdf, Data dostępu: 26.12.2020

[2] https://sjp.pwn.pl/sjp/autonomia;2551312.html, Data dostępu:26.12.2020

[3] Danielle Muoio, Krzysztof Majdan, https://businessinsider.com.pl/technologie/nowe-technologie/elon-musk-autopilot-nowe-samoprowadzace-sie-samochody-tesli/4f2h434, Data dostępu: 26.12.2020

[4] https://fetchrobotics.com/products-technology/virtualconveyor/freight-robots/, Data dostępu: 26.12.2020

[5] https://www.starship.xyz/company/, Data dostępu: 26.12.2020

[6] Wenyu Zhang, Jingyao Gai, Zhigang Zhang, Lie Tang, Qingxi Liao, Youchun Ding: Double-DQN based path smoothing and tracking control method for robotic vehicle navigation, Computers and Electronics in Agriculture, s. November 2019

[7] Hong, tang & nakhaeinia, danial & karasfi, babak. (2012). Application of fuzzy logic in mobile robot navigation. 10.5772/36358.

[8] Yufka, alpaslan & parlaktuna, osman. (2009). Performance comparison of the bug's algorithms for mobile robots. 10.13140/rg.2.1.3616.6564.

[9] Mcguire, kimberly & croon, guido & tuyls, karl. (2018). A comparative study of bug algorithms for robot navigation.

[10] K.n. mcguire, g.c.h.e. de croon, k. Tuyls, a comparative study of bug algorithms for robot navigation, robotics and autonomous systems, volume 121,2019,

[11] Mousavi, sajad & schukat, michael & howley, enda. (2018). Deep reinforcement learning: an overview. Lecture notes in networks and systems. 426-440. 10.1007/978-3-319-56991-8_32.

[12] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, Patrick Pérez (2020). Deep Reinforcement Learning for Autonomous Driving: A Survey

[13] Neural Networks and Statistical Learning, KE-LIN DU and M. N. S. SWAMY Enjoyor Labs, Enjoyor Inc., China Concordia University, Canada April 28, 2013

[14] Watkins C. J. C. H. and Dayan P., 1992. Technical Note: Q-learning. *Machine Learning*, 8, 3-4, 279–292

[15] Jang, Beakcheol & Kim, Myeonghwi & Harerimana, Gaspard & Kim, Jong. (2019). Q-Learning Algorithms: A Comprehensive Classification and Applications. IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2941229.

[16] Ee Soong Low, Pauline Ong, Kah Chun Cheah, Solving the optimal path planning of a mobile robot using improved Q-learning, Robotics and Autonomous Systems, Volume 115, 2019, Pages 143-161

[17] Kwasigroch A. , Grochowski M. Rozpoznawanie obiektów przez głębokie sieci neuronowe XXVIII cykl seminarów zorganizowanych przez ptetis Oddział w Gdańsku ZASTOSOWANIE KOMPUTERÓW W NAUCE I TECHNICE 2018 (XXVIII; 2018; Gdańsk, Polska)

[18] Taofeek D. Akinosho, Lukumon O. Oyedele, Muhammad Bilal, Anuoluwapo O. Ajayi, Manuel Davila Delgado, Olugbenga O. Akinade, Ashraf A. Ahmed, Deep learning in the construction industry: A review of present status and future innovations, Journal of Building Engineering, Volume 32, 2020

[19] Esteva, Andre & Robicquet, Alexandre & Ramsundar, Bharath & Kuleshov, Volodymyr & DePristo, Mark & Chou, Katherine & Cui, Claire & Corrado, Greg & Thrun, Sebastian & Dean, Jeff. (2019). A guide to deep learning in healthcare. Nature Medicine. 25. 10.1038/s41591-018-0316-z.

[20] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning.

[21] Hado van Hasselt and Arthur Guez and David Silver. Deep Reinforcement Learning with Double Q-learning. Google Deep Mind, Dec 2015

[22] Jianqing Fan, Zhaoran Wang, Yuchen Xie, Zhuoran Yang. A Theoretical Analysis of Deep Q-Learning. 2nd Annual Conference on Learning for Dynamics and Control Proceedings of Machine Learning Research vol 120:1–4, 2020

[23] https://www.generationrobots.com/media/rplidar-a1m8-360-degree-laser-scanner-development-kit-datasheet-1.pdf/, Data dostępu: 26.12.2020

[24] http://gazebosim.org/, Data dostępu: 26.12.2020

[25] http://sdformat.org/, Data dostępu: 26.12.2020

[26] https://www.ros.org/about-ros/, Data dostępu: 26.12.2020

[27] http://wiki.ros.org/ROS/Introduction/, Data dostępu: 26.12.2020

[28] http://wiki.ros.org/openai_ros/, Data dostępu: 26.12.2020

[29] Junli Gao, Weijie Ye, Jing Guo and Zhongjuan Li, Deep Reinforcement Learning for Indoor Mobile Robot Path Planning, 25 September 2020, Data dostępu: 18.01.2020