

UNIWERSYTET ŚLĄSKI W KATOWICACH
WYDZIAŁ NAUK ŚCISŁYCH I TECHNICZNYCH
INFORMATYKA

Grzegorz Galios
315850

Zastosowanie konwolucyjnej sieci neuronowej w procesie sterowania
samochodem w symulowanym środowisku

PRACA DYPLOMOWA MAGISTERSKA

Promotor: dr Rafał Skinderowicz

Katowice, 2022

Słowa kluczowe: *konwolucyjne sieci neuronowe, uczenie ze wzmacnieniem*

Oświadczenie autora pracy

Ja, niżej podpisany:

imię (imiona) i nazwisko: Grzegorz Galios

autor pracy dyplomowej pt. „*Zastosowanie konwolucyjnej sieci neuronowej w procesie sterowania samochodem w symulowanym środowisku*”

Numer albumu: 315850

Student Wydziału Nauk Ścisłych i Technicznych Uniwersytetu Śląskiego w Katowicach

kierunku studiów: Informatyka - niestacjonarne II stopnia

specjalności: Inteligentne Systemy Informatyczne

Oświadczam, że w/w. praca dyplomowa:

- została przygotowana przeze mnie samodzielnie¹,
- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity Dz. U. z 2006 r. Nr 90, poz. 631, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- nie zawiera danych i informacji, które uzyskałem w sposób niedozwolony,
- nie była podstawą nadania dyplomu uczelni wyższej lub tytułu zawodowego ani mnie, ani innej osobie.

Oświadczam również, że treść pracy dyplomowej zamieszczonej przeze mnie w Archiwum Prac Dyplomowych jest identyczna z treścią zawartą w wydrukowanej wersji pracy.

Jestem świadomy odpowiedzialności karnej za złożenie fałszywego oświadczenia.

.....

Data

.....

Podpis autora pracy

¹ uwzględniając merytoryczny wkład promotora (w ramach prowadzonego seminarium dyplomowego)

Spis treści

Wstęp	1
1 Samochody autonomiczne	3
1.1 Typy czujników	4
1.2 Kamera kontra LiDAR	5
1.3 Rozwiązania wiodące na rynek	8
2 Konwolucyjne sieci neuronowe	18
2.1 Architektura	18
2.2 Trening	22
2.3 Przykłady z literatury	24
2.4 Obszary zastosowań	25
2.5 Problemy i wyzwania na przyszłość	26
2.6 Podsumowanie	29
3 Projekt systemu i opis narzędzi	30
3.1 Projekt systemu	30
3.2 Opis narzędzi	31
4 Opis implementacji	39
4.1 Środowisko Uczenia	39
4.2 Narzędzie magents_learn	43
5 Eksperymenty obliczeniowe	44
5.1 Metodyka eksperymentów	44
5.2 Opis implementacji	44
5.3 Przebieg eksperymentów i analiza wyników	50
5.4 Analiza wytrenowanego modelu	53
5.5 Wnioski	55
Podsumowanie	57
Spis rysunków	58
Bibliografia	60

Wstęp

Przedmiotem pracy jest zastosowanie konwolucyjnej sieci neuronowej oraz technik głębskiego uczenia maszynowego do rozwiązania problemu sterowania samochodem autonomicznym. Ponieważ jest to bardzo złożone zagadnienie, dlatego wiele uwagi zostało poświęconej na potrzeby dobrego zrozumienia jego podstawowych elementów.

Cel i zakres pracy

Jako cel pracy przyjęto opracowanie prostego systemu uczącego konwolucyjne sieci neuronowe w oparciu o symulacje uruchamiane w przygotowanym środowisku wirtualnym. Sieć neuronowa rozwiązuje problem jazdy samochodem po torze wyścigowym w oparciu o obraz z kamer zamontowanych w samochodzie.

Zakres pracy obejmuje poniższe zagadnienia:

1. Przegląd literatury na temat samochodów autonomicznych, wykorzystywanych typów czujników oraz wiodących rozwiązań na rynku.
2. Przegląd literatury na temat konwolucyjnych sieci neuronowych.
3. Zaprojektowanie systemu uczącego konwolucyjne sieci neuronowe oraz wybór odpowiednich narzędzi do jego implementacji.
4. Implementacja systemu uczącego konwolucyjne sieci neuronowe w oparciu o obraz z kamer zamontowanych w wirtualnym samochodzie oraz inne sygnały płynące ze środowiska symulacji.
5. Przeprowadzenie eksperymentów obliczeniowych na utworzonym systemie oraz analiza uzyskanych wyników.

Struktura pracy

Praca jest złożona z pięciu numerowanych rozdziałów, a każdy z nich dotyczy określonego elementu omawianego tematu. Kolejność rozdziałów jest realizacją zasady „*od ogólnego do szczegółu*” - praca rozpoczyna się od omówienia teoretycznych pojęć, niezbędnych do zrozumienia jej dalszej treści. Natomiast dwa ostatnie rozdziały dotyczą bardzo konkretnych oraz praktycznych zagadnień. Lista rozdziałów przedstawia się następująco:

1. Samochody autonomiczne

Rozdział opisuje problematykę zagadnienia samochodów autonomicznych, kładąc szczególny nacisk na zaprezentowanie oraz porównanie ze sobą urządzeń wykorzystywanych w modelu percepcji samochodów. Pokażna część rozdziału została poświęcona opisaniu wiodących rozwiązań oferowanych obecnie na rynku.

2. Konwolucyjne sieci neuronowe

Rozdział zawiera najistotniejsze zagadnienia teoretyczne z zakresu konwolucyjnych sieci neuronowych, ich treningu oraz obszaru zastosowań. Przedstawione zagadnienia to niezbędne minimum, potrzebne do prawidłowego zrozumienia dalszej części pracy.

3. Projekt systemu i opis narzędzi

Rozdział został podzielony na dwie części. W pierwszej z nich zostały opisane główne założenia projektowe dla implementowanego systemu. Z kolei w części drugiej zamieszczono opis narzędzi wykorzystanych podczas implementacji systemu.

4. Opis implementacji

Rozdział przedstawia opis implementacji systemu wykonanego na potrzeby tej pracy. Szczególny nacisk został położony na opisanie kluczowych komponentów wchodzących w skład systemu.

5. Eksperymenty obliczeniowe

Rozdział poświęcony na przedstawienie metodyki przeprowadzanych eksperymentów obliczeniowych oraz analizę uzyskanych wyników.

Rozdział 1

Samochody autonomiczne

Samochody autonomiczne są pojazdami, które mają zdolność do interpretacji swojego otoczenia i bezpiecznego poruszania się po nim przy jak najmniejszej ingerencji człowieka siedzącego na fotelu kierowcy. Poziom automatyzacji sterowania może się różnić dla poszczególnych modeli. Organizacja SAE (*Society of Automotive Engineers*) zdefiniowała sześć poziomów automatyzacji sterowania samochodem [60]. W pełni autonomiczne samochody mają być całkowicie pozbawione kierownicy, pedałów oraz innych przyrządów tradycyjnie używanych do sterowania ręcznego.

W chwili obecnej nie istnieje ani jeden model w pełni autonomicznego samochodu, dopuszczonego do produkcji seryjnej. Co więcej, niewiele wskazuje na to, żeby taki stan rzeczy miał się zmienić w przeciągu kilku kolejnych lat [2, 36]. Zanim to się wydarzy, inżynierowie oraz badacze zajmujący się tym zagadnieniem będą zmuszeni do rozwiązań jeszcze wiele trudnych problemów technicznych. Oprócz tego dochodzi również kwestia dostosowania przepisów prawnych [9].

Aby zrozumieć działanie samochodów autonomicznych, należy zapoznać się z czterema kluczowymi elementami zagadnienia:

1. **Percepcja** - pozwala widzieć świat wokół samochodu, jak również rozpoznawać i klasyfikować obiekty pojawiające się w polu widzenia. Aby podejmować dobre decyzje, obiekty powinny być rozpoznawane tak szybko jak to tylko możliwe. Samochód powinien również znać odległości, jakie dzielą go od tych obiektów. Dzięki percepcji samochód wie, czy w danej chwili może przyspieszyć, czy raczej powinien zwalniać. Do obserwacji otoczenia wykorzystuje się 4 typy czujników: czujnik ultradźwiękowy, radar, LiDAR oraz kamerę. Nie zawsze są wykorzystywane wszystkie typy czujników, choć popularnym podejściem jest ich połączone użycie.
2. **Lokalizacja** - niezbędna do określenia pozycji oraz orientacji samochodu względem innych obiektów obecnych w otaczającej go przestrzeni. Pomaga także w klasyfikacji oraz wizualnej segmentacji obiektów widzianych przez kamerę. Więcej na temat lokalizacji można przeczytać w artykule [4].
3. **Predykcja** - jej zadaniem jest przewidywanie zachowań innych uczestników ruchu

drogowego. Model predykcji w głównej mierze opiera się na obliczaniu trajektorii, po jakich poruszają się obiekty w polu widzenia samochodu [58]. Trajektorie te pomagają w podejmowaniu odpowiednich decyzji w celu uniknięcia kolizji na drodze.

4. **Podejmowanie decyzji** - model podejmowania decyzji musi dawać błyskawiczne i precyzyjne wyniki, działając przy tym w środowisku odznaczającym się wysokim poziomem niepewności. Pod uwagę należy brać wiele czynników, takich jak nieracjonalne zachowanie uczestników ruchu lub też zakłócenia pomiarów na skutek usterki jednego z czujników w samochodzie. Celem modelu jest wybór takiego zestawu zachowań, który będzie najbardziej odpowiedni dla danej sytuacji.

1.1 Typy czujników

Aby widzieć swoje otoczenie, samochody autonomiczne potrzebują urządzeń dostarczających im odpowiednich danych wejściowych. Urządzenia te można ogólnie nazwać czujnikami. Standardowo istnieją cztery typy czujników, jakie można zastosować w samochodzie autonomicznym [51]: **czujnik ultradźwiękowy**, **radar**, **LiDAR** oraz **kamera**. Za wyjątkiem kamery, wszystkie pozostałe czujniki wykorzystują tzw. „regułę czasu przelotu” (z ang. *time-of-flight principle*). Reguła polega na pomiarze odległości od przeszkód oraz prędkości przemieszczania się obiektów na podstawie czasu od emisji sygnału do jego powrotu. Rodzaj sygnału zależy od zastosowanego czujnika - może to być fala ultradźwiękowa, fala radiowa lub wiązka światła. Przykładem zastosowania reguły czasu przelotu jest echolokacja, wykorzystywana m.in. przez nietoperze.

1.1.1 Czujnik ultradźwiękowy

Korzysta z fali ultradźwiękowej o zadanej częstotliwości. Z powodu specyfiki rozchodzenia się fali ultradźwiękowej w powietrzu, jego zasięg wynosi do kilku metrów, dlatego jest głównie wykorzystywany w systemach **wspomagania parkowania** oraz **kontroli marowego pola**.

1.1.2 Radar

Używa fal radiowych, dzięki czemu zasięg jest znacznie większy. Radary są bardzo popularne do pomiarów odległości, natomiast nie nadają się do rozpoznawania obiektów. Jest to spowodowane niską rozdzielczością sygnału. Obecnie w samochodach autonomicznych wykorzystuje się dwa typy radarów:

1. Krótkozasięgowe - do pomiarów na krótkim dystansie, tj. do 30 metrów. Fala radiowa ma częstotliwość ok. 24 GHz. Zalety rozwiązania to niska podatność na zakłócenia

oraz niskie koszty produkcji.

2. Długozaścigowe - do wykonywania pomiarów na dystansie do 250 metrów. Częstotliwość fali radiowej waha się pomiędzy 76 a 77 GHz.

1.1.3 LiDAR

Nazwa to akronim od angielskiego wyrażenia *Light Detection and Ranging*. Pomiarów można dokonywać na krótkich i długich dystansach. Główną zaletą tej technologii jest generowanie danych 3D o wysokiej rozdzielczości, albowiem LiDAR emisuje setki tysięcy impulsów laserowych na sekundę. Największe wady LiDAR-u to wysoka cena oraz wysoka podatność na warunki pogodowe - działanie czujników ulega znaczącemu pogorszeniu pod wpływem deszczu, śniegu lub mgły.

1.1.4 Kamera

Kamera dla maszyny jest tym, czym oko dla człowieka - pozwala widzieć otaczający świat. We współczesnych samochodach autonomicznych jest to niezbędny element wyposażenia. Kamera jako jedyna dostarcza informacji o kolorach, jest również nieocenioną pomocą przy identyfikacji oraz klasyfikacji obiektów należących do otoczenia. Odczytywanie znaków drogowych lub ostrzeganie przed niezamierzonym zjazdem ze swojego pasa byłoby niemożliwe bez wykorzystania kamery.

Niemniej jednak, nie jest to rozwiązanie pozbawione wad - na jej działanie mają wpływ warunki pogodowe, jak również pojedyncza kamera nie dostarcza informacji o głębi obrazu. Aby uzyskać obraz 3D, wymagane jest użycie co najmniej dwóch kamer.

1.2 Kamera kontra LiDAR

Od kilku lat w środowisku osób zajmujących się rozwojem technologii samochodów autonomicznych trwa zagorzała dyskusja. Co jest lepsze - kamera czy LiDAR? Prezentowane są różne stanowiska, a przytaczane argumenty za i przeciw dla każdego z rozwiązań utrwalają w przekonaniu, iż temat nie jest trywialny i wart jest dokładniejszego przyjrzenia się. Dlatego też w niniejszym podrozdziale postanowiłem pokrótkę omówić te zagadnienie [52], przedstawiając zalety i wady każdego z rozwiązań.

1.2.1 LiDAR - zalety

Zwolennicy LiDAR-u wskazują na jego wysoką dokładność oraz precyzję. System LiDAR opracowany przez firmę Waymo jest tak zaawansowany, że potrafi rozpoznawać oraz przewidywać kierunek, w jakim mogą podążać piesi.

Inną zaletą LiDAR-u jest generowanie obrazu 3D bezpośrednio z danych otrzymanych z czujnika. LiDAR jest pod tym względem bardziej dokładny w porównaniu do kamery, ponieważ jego działanie nie będzie zakłócone zmiennym oświetleniem otoczenia.

Kolejnym argumentem jest znacznie mniejsze zapotrzebowanie na moc obliczeniową niż ma to miejsce w przypadku użycia kamery. Dzieje się tak, ponieważ bezpośrednio z danych wejściowych dostarczanych przez LiDAR da się wyciągnąć wiele informacji o przestrzeni wokół samochodu. Te same informacje w przypadku użycia kamery muszą zostać wygenerowane przez komputer.

1.2.2 LiDAR - wady

Wskazując wady LiDAR-u, należy wspomnieć o kilku kwestiach. Pierwszą z nich jest cena - systemy LiDAR były kiedyś bardzo drogie. Oryginalnie używany przez firmę Google zestaw czujników LiDAR dla pojedynczego samochodu kosztował powyżej 75 000 dolarów. Dziś ceny czujników LiDAR są znacznie niższe, niemniej jednak wciąż są wysokie.

Kolejna wada to zakłaczanie sygnału, do jakiego dochodzi gdy w pobliżu siebie znajdą się co najmniej dwa samochody wyposażone w system LiDAR. Im więcej samochodów jednocześnie generujących impulsy świetlne, tym większe ryzyko „oślepienia” innych pojazdów. Producenci będą musieli jakoś temu zaradzić.

Nieznane są również długofalowe skutki pernamentnej ekspozycji na wiązki laserowe generowane przez system LiDAR. Obawy dotyczą zwłaszcza narządu wzroku człowieka [3, 6, 35].

Kolejnym ograniczeniem systemu LiDAR jest pogorszone działanie przy niekorzystnych warunkach pogodowych, takich jak deszcz, śnieg czy mgła. Krople deszczu mogą być przez system interpretowane jako ściana na środku drogi. Niektóre firmy (np. Ford) radzą sobie z tym problemem poprzez opracowanie własnych algorytmów wspomagających interpretację sygnału pobieranego z czujników LiDAR. Niemniej jednak, są to rozwiązania niedojrzałe i wymagające dalszego dopracowania.

Ostatnim palącym problemem jest niemożliwość dostarczenia informacji, które można bez problemu uzyskać z obrazu kamery. Do tych informacji należy m.in. oznakowanie drogi (pionowe i poziome) oraz kolory światel na skrzyżowaniach.

1.2.3 Kamera - zalety

Po pierwsze, kamery są znacznie tańsze od systemów LiDAR, co przekłada się na większą dostępność dla konsumentów. Są również łatwiejsze do wdrożenia, ponieważ kamery

jako urządzenia są bardzo szeroko dostępne na rynku.

Po drugie, samochód wyposażony w kamery znacznie lepiej radzi sobie w trudnych warunkach pogodowych (deszcz, śnieg, mgła) niż ma to miejsce w przypadku samochodu z systemem LiDAR.

Po trzecie, kamery potrafią widzieć świat tak samo jak widzą go ludzie, zatem są w stanie rozpoznawać kolory, oznakowania dróg oraz wiele innych elementów otoczenia.

Ostatnia zaleta jest taka, że kamery da się dużo łatwiej zabudować w nadwozie w taki sposób, aby nie psuć ogólnej estetyki samochodu. Dzięki temu znacznie więcej potencjalnych klientów może być zainteresowana zakupem takiego pojazdu.

1.2.4 Kamera - wady

Kamery są narażone na te same problemy, co ludzkie oko - jazda po słabo oświetlonym terenie lub ostre, oślepiające światło świecące prosto w kamerę mogą powodować zakłócenia w rozpoznawaniu otoczenia. W takich przypadkach samochód nie zawsze jest w stanie podejmować właściwe decyzje na drodze.

Kolejną wadą jest fakt, że surowe dane z kamery są mało użyteczne, ponieważ każda zarejestrowana klatka obrazu jest tylko zlepkiem pikseli o określonym zagęszczeniu. Aby dane pochodzące z kamery mogły być wykorzystane, muszą zostać przetworzone przez odpowiednie oprogramowanie. Do stworzenia takiego oprogramowania jest wymagana zaawansowana wiedza z zakresu sieci neuronowych oraz głębokiego uczenia maszynowego, a także wielka ilość danych potrzebnych do treningu sieci.

Kolejny problem to moc obliczeniowa, niezbędna do przetwarzania w czasie rzeczywistym ogromnej ilości danych generowanych z kamer samochodu podczas jazdy. Przetwarzanie danych nie może przekraczać zakładanych limitów czasowych, ponieważ w przeciwnym razie mogłoby to wpływać na bezpieczeństwo jazdy. Ilość generowanych danych także nie może być znacząco zredukowana, ponieważ trzeba zachować niezbędnego minimum w zakresie liczby klatek na sekundę oraz rozdzielczości obrazów wejściowych. Aby stawić czoła tym wyzwaniom, projektowane są specjalne platformy sprzętowe do wykorzystania w samochodach autonomicznych. Przykładem takiej platformy jest *Full Self-Driving Computer*, zaprezentowany przez firmę Tesla w 2019 roku [53].

1.2.5 Podsumowanie

Po zapoznaniu z argumentami obydwu stron wydaje się, że bardziej przyszłościowym rozwiązaniem dla samochodów autonomicznych może być rezygnacja z dalszego użycia

technologii LiDAR i skupienie się na rozwoju oprogramowania przetwarzającego obraz z kamer zamontowanych w pojazdzie. Uważa tak również wiele wybitnych osób zajmujących się tym tematem, a jednym z ich najgłośniejszych przedstawicieli jest Elon Musk. Wielokrotnie krytykował stosowanie systemu LiDAR w samochodach autonomicznych, uważając to za ślepy zaułek [14]. Z drugiej strony, choć docelowo Elon Musk może mieć rację, to przy obecnym stanie rozwoju oprogramowania LiDAR wciąż wydaje się być atrakcyjną opcją, co zostało bardzo dobrze opisane w artykule [71]. Istnieje bowiem wiele sytuacji, w których LiDAR wciąż daje lepsze efekty niż systemy oparte wyłącznie na obrazie z kamer.

1.3 Rozwiązania wiodące na rynku

W chwili obecnej wiele korporacji o międzynarodowych zasięgach inwestuje ogromne środki w rozwój technologii samochodów autonomicznych. Niniejsze zestawienie jest subiektywną listą najciekawszych rozwiązań obecnych na rynku. Zaprezentowane rozwiązania zostały przeze mnie wybrane, ponieważ charakteryzują się najwyższym stopniem zaawansowania i dają najlepsze perspektywy na rozwój technologii w przyszłości.

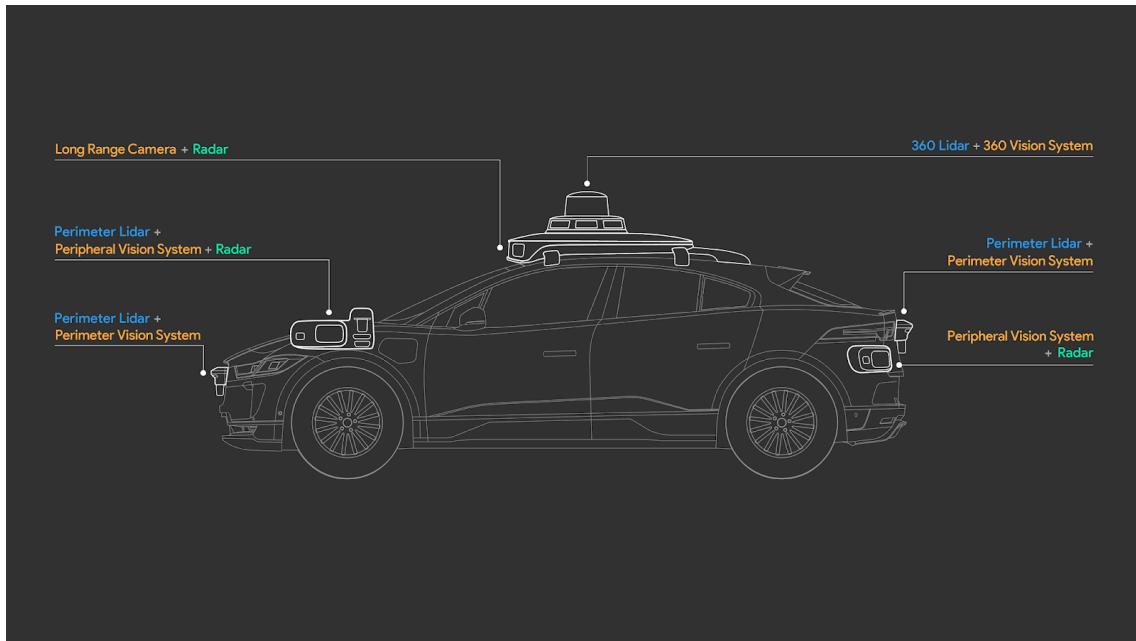
1.3.1 Waymo Driver

Waymo zostało założone w 2009 r. jako projekt firmy Google, później stało się przedsiębiorstwem technologicznym wchodząącym w skład holdingu Alphabet Inc. Waymo zajmuje się rozwijaniem technologii samochodów autonomicznych. Ich obecny system autonomicznej jazdy samochodem nosi nazwę **Waymo Driver** i jest stawiany w gronie najbardziej dojrzałych i przetestowanych rozwiązań, ponieważ samochody Waymo mają na swoim koncie ponad 20 milionów mil przejechanych w prawdziwym świecie i ponad 20 miliardów mil przejechanych w symulacji komputerowej. Trening oraz testy odbywały się w możliwie największym spektrum warunków pogodowych oraz rozpatrywanych scenariuszy drogowych.

Samochód do jazdy po danym regionie potrzebuje precyzyjnych oraz szczegółowych map 3D, posiadających szereg informacji, takich jak profile dróg oraz wszystkie elementy pionowego i poziomego oznakowania.

Obecna, piąta generacja systemu **Waymo Driver** wykorzystuje trzy typy czujników: kamery, radar oraz LiDAR [39]. Układ czujników został zaprezentowany na rysunku 1.1.

Czujniki LiDAR mają wysoką rozdzielcość i zasięg do 300 metrów. Kamery dają bardzo ostry obraz o dobrej jakości. System widzenia obwodowego działa we współpracy z systemem LiDAR-ów obwodowych (z ang. *perimeter lidars*), dzięki czemu samochód lepiej widzi co się dzieje w jego najbliższym otoczeniu - to pomaga w wielu sytuacjach, na



Rysunek 1.1: Układ czujników systemu Waymo Driver

Źródło:

<https://blog.waymo.com/2020/03/introducing-5th-generation-waymo-driver.html>

przykład podczas manewrów parkowania. Radary wspomagają pracę czujników LiDAR w trudnych warunkach pogodowych, odznaczają się także wysoką rozdzielczością i zasięgiem rzędu kilkuset metrów.

1.3.2 NVIDIA DRIVE

Firma NVIDIA od dawna inwestuje w rozwój technologii samochodów autonomicznych. Oferowane przez nich rozwiązania noszą nazwę NVIDIA DRIVE [22]. Są to kompleksowe rozwiązania sprzętowo-programowe, stanowiące bazę do tworzenia autonomicznych pojazdów. NVIDIA prowadzi współpracę z wieloma partnerami działającymi w branży automotive [18], a wśród nich są: Audi, Toyota, Volkswagen, Volvo.

Dlaczego NVIDIA DRIVE jest tak wyjątkowe? Ponieważ otwiera drogę do zupełnie nowych możliwości dla firm zainteresowanych tematyką jazdy autonomicznej. Firmy te nie potrzebują prowadzić działań badawczo-rozwojowych w celu opracowania własnej technologii, tylko mogą wykupić istniejące rozwiązanie i zintegrować go ze swoim systemem. Jednym ze spektakularnych przypadków użycia NVIDIA DRIVE było szybkie wdrożenie usługi przewozów miejskich, wykonane przez firmę Optimus Ride [15].

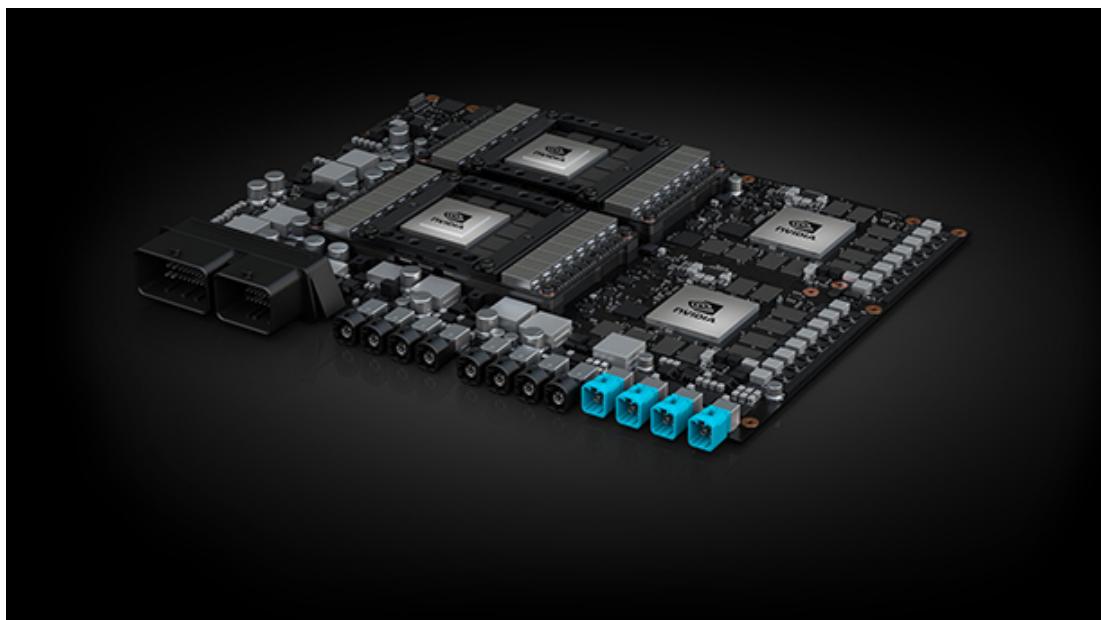
Rozwiązania oferowane w ramach NVIDIA DRIVE można podzielić na dwie części: sprzętową (hardware) oraz programową (software).

Rozwiązania sprzętowe

Sprzęt oferowany w ramach NVIDIA DRIVE tworzy rodzinę platform obliczeniowych o nazwie NVIDIA DRIVE AGX [17]. W skład rodziny wchodzą obecnie dwa modele:

1. NVIDIA DRIVE AGX Pegasus

Posiada moc obliczeniową 320 TOPS (Tera Operations Per Second), co wciąż czyni go jednym z najszybszych rozwiązań dostępnych na rynku. Architekturę oparto na dwóch procesorach NVIDIA Xavier i dwóch układach graficznych TensorCore. Bardzo efektywny energetycznie, pobiera niewielką ilość prądu w stosunku do wydajności jaką posiada. Może zostać wykorzystany przy tworzeniu w pełni autonomicznych systemów jazdy, co oznacza że pedały i kierownica nie są wymagane. Wygląd platformy NVIDIA DRIVE AGX Pegasus został przedstawiony na rysunku 1.2.



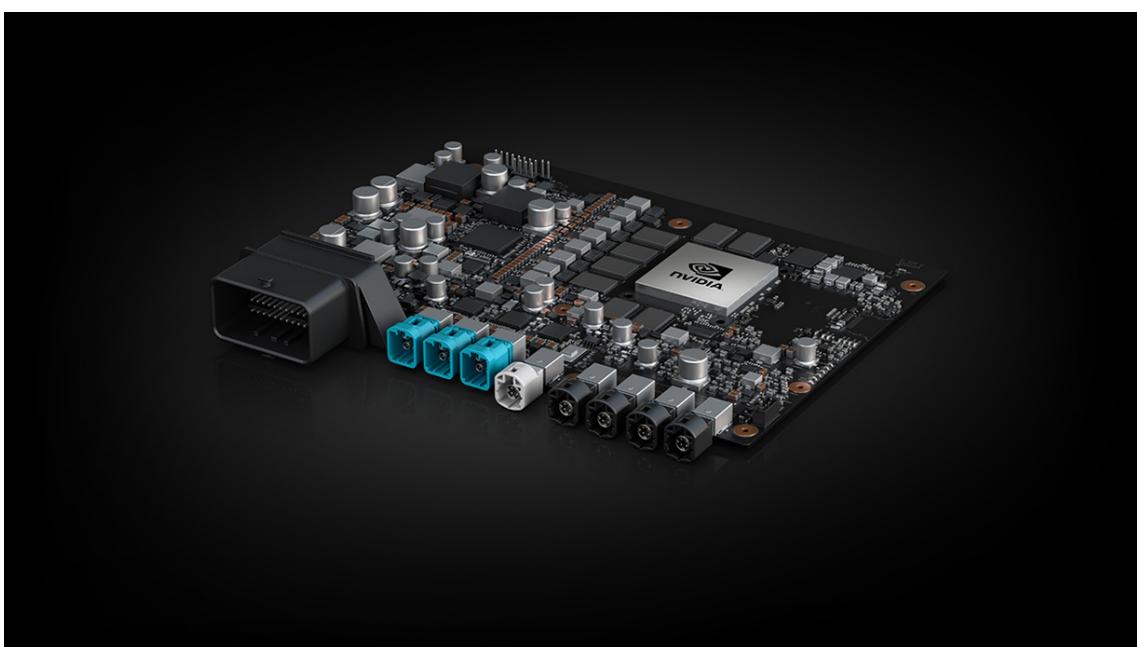
Rysunek 1.2: NVIDIA DRIVE AGX Pegasus

Źródło: <https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware>

2. NVIDIA DRIVE AGX Xavier

Mniejszy od Pegasusa, charakteryzuje się też mniejszą mocą obliczeniową oscylującą w granicach 30 TOPS. Zużycie energii jest jednak bardzo niskie, ponieważ wynosi tylko 30 watów. Ta platforma obliczeniowa jest za słaba żeby obsłużyć w pełni autonomiczne systemy jazdy, ale wystarcza do częściowej automatyzacji sterowania.

Rysunek 1.3 przedstawia wygląd platformy Xavier.



Rysunek 1.3: NVIDIA DRIVE AGX Xavier

Źródło: <https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware>

Oprócz tego, w skład rozwiązań sprzętowych wchodzi także NVIDIA DRIVE Hyperion, czyli najbardziej kompleksowe rozwiązanie oferowane przez firmę NVIDIA [19]. W skład tego rozwiązania wchodzi platforma obliczeniowa (Xavier lub Pegasus), zestaw czujników oraz oprogramowanie sterujące.

Oprogramowanie NVIDIA DRIVE

Oprogramowanie NVIDIA DRIVE składa się z następujących modułów [20]:

1. DRIVE OS

Podstawowy stos oprogramowania. Składa się z systemu operacyjnego czasu rzeczywistego (RTOS), modułu NVIDIA Hypervisor, bibliotek NVIDIA CUDA oraz NVIDIA TensorRT [21], a także innych modułów umożliwiających dostęp do zasobów

sprzętowych. DRIVE OS zapewnia bezpieczne i stabilne środowisko wykonawcze dla uruchamianych aplikacji. Najważniejsze cechy modułu DRIVE OS:

- Wieloużytkownikowy, 64-bitowy system operacyjny;
- Platforma CUDA dla obliczeń równoległych;
- API NvMedia dla akcelerowanych sprzętowo multimedii oraz przetwarzania danych z kamery;
- API graficzne: OpenGL, OpenGL ES, EGL z rozszerzeniami EGLStream;
- Biblioteki głębokiego uczenia maszynowego: TensorRT, cuDNN.

2. DriveWorks

Zestaw do tworzenia oprogramowania (SDK) pozwalający programistom na implementację oprogramowania dla samochodów autonomicznych poprzez dostarczanie zbioru przydatnych bibliotek i narzędzi deweloperskich. Zestaw ten został zaprojektowany z myślą o maksymalnym wykorzystaniu możliwości sprzętowych komputera, ze szczególnym uwzględnieniem jego przepustowości.

Najważniejsze cechy modułu DriveWorks:

- Efektywne wykorzystanie wszystkich dostępnych procesorów;
- Optymalizacja formatu danych komunikacji pomiędzy silnikami sprzętowymi;
- Ograniczenie kopiowania danych do niezbędnego minimum;
- Wykorzystanie bardzo wydajnych algorytmów.

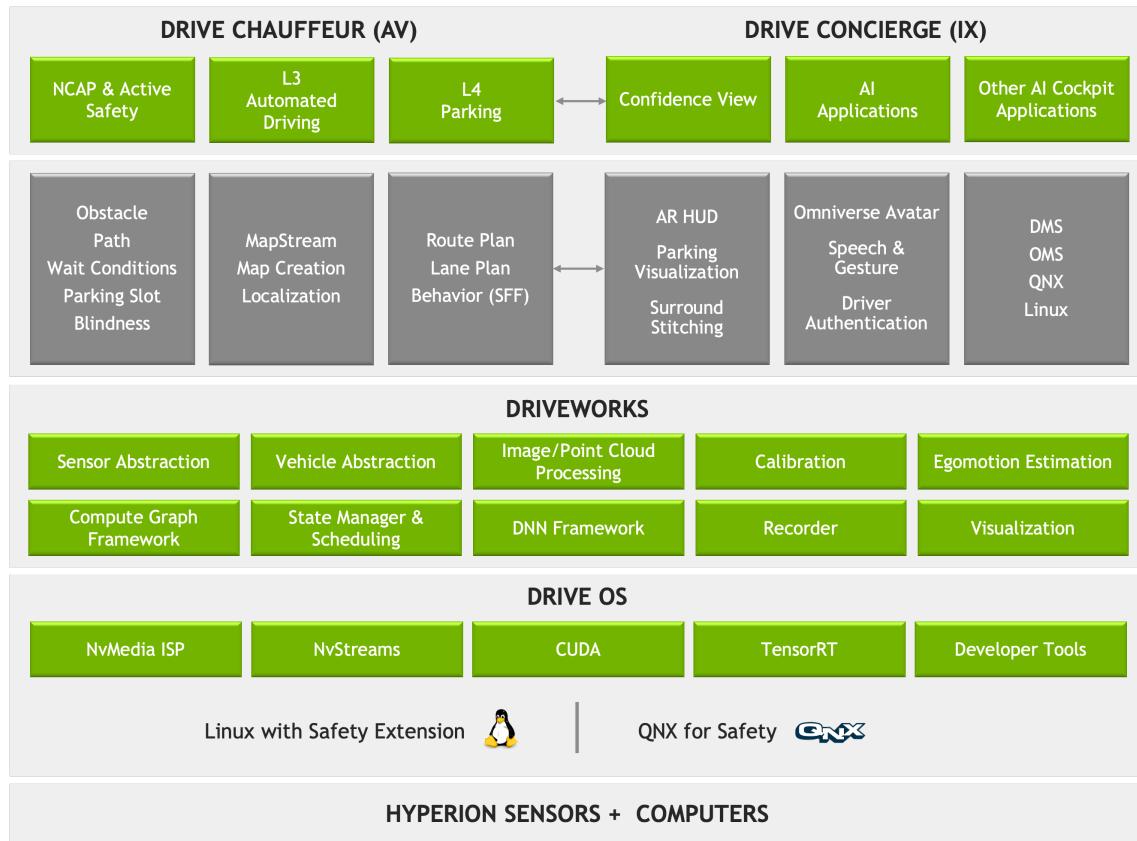
3. DRIVE AV

Dostarcza modułów percepcji, mapowania oraz planowania. Percepcja pozwala na wykrywanie i śledzenie obiektów rejestrowanych przez czujniki oraz obliczanie odległości pomiędzy nimi. Mapowanie zbiera i przypisuje zanonimizowane dane od użytkowników platformy NVIDIA DRIVE Hyperion w celu uzyskania bezpiecznego, niezawodnego i aktualnego pokrycia map w jakości HD. Planowanie służy m.in. do wyznaczania ścieżek przejazdu dla samochodu.

4. DRIVE IX

Otwartoźródłowy moduł oprogramowania do obsługi czujników wnętrza kabiny. Jest niezbędny do wprowadzenia innowacyjnych rozwiązań kokpitu AI, takich jak monitorowanie zachowań kierowcy oraz interakcja głosowa człowieka z maszyną.

Moduły oprogramowania NVIDIA DRIVE zostały zaprezentowane na rysunku 1.4.



Rysunek 1.4: Moduły oprogramowania NVIDIA DRIVE

Źródło: <https://developer.nvidia.com/drive/drive-sdk>

1.3.3 Tesla Autopilot

Firma Tesla zasłynęła produkcją swoich samochodów elektrycznych. Opracowała również zaawansowany system wspomagania jazdy, noszący nazwę **Tesla Autopilot** [72]. System został stworzony w celu zwiększenia bezpieczeństwa oraz komfortu podróżowania kierowcy, pasażerów oraz innych uczestników ruchu drogowego. Każdy nowy samochód Tesli wyposażony jest w osiem kamer, a samochody sprzedawane poza Ameryką Północną są również wyposażone w zestaw czujników radarowych. Autopilot nie zapewnia pełnej autonomii samochodu, dlatego kierowca musi zachować stałą czujność podczas jazdy i być gotów przejąć kontrolę nad sterowaniem, gdy zajdzie taka potrzeba.

W skład Autopilota wchodzą następujące funkcje:

1. **Traffic-Aware Cruise Control** - dopasowuje prędkość samochodu do otaczającego ruchu ulicznego;
2. **Autosteer** - wspomaga kierowanie na drodze posiadającej wyraźnie oznaczone pasy

ruchu oraz wykorzystuje aktywny tempomat, który uwzględnia ruch drogowy wokół samochodu.

Tesla ma także w swojej ofercie pakiet rozszerzeń **Full Self-Driving Capability**, który wzbogaca Autopilota o takie funkcje jak:

1. **Navigate on Autopilot** - umożliwia półautonomiczną jazdę po autostradzie.
2. **Auto Lane Change** - pomaga w zmianie pasa ruchu na autostradzie, gdy jest włączona funkcja **Autosteer**.
3. **Autopark** - pomaga zaparkować samochód (równolegle lub prostopadle).
4. **Summon** - wprowadza oraz wyprowadza samochód z ciasnej przestrzeni za pomocą aplikacji mobilnej.
5. **Smart Summon** - samochód nawiguje po skomplikowanych obiektach typu place parkingowe, aby odnaleźć kierowcę i podjechać do niego.
6. **Stop Sign and Traffic Light Control** - pierwsza funkcja przeznaczona do użytku w ruchu miejskim. Pojazdy Tesli wyposażone w tę funkcję potrafią reagować na znaki stop oraz sygnalizację świetlną. Przy dojeżdżaniu do skrzyżowania ze światłami samochód zwalnia (nawet mając zielone), a kierowca musi nacisnąć pedał przyspieszenia. Wtedy samochód kontynuuje jazdę w trybie automatycznym.

Samochody Tesli są również wyposażone w zestaw aktywnych funkcji bezpieczeństwa:

1. **Automatic Emergency Braking** - wykrywa przeszkody, w które samochód mógłby uderzyć i odpowiednio do sytuacji uruchamia hamulce;
2. **Forward Collision Warning** - ostrzega przed nadciągającą kolizją z wolniej poruszającymi się samochodami;
3. **Side Collision Warning** - ostrzega przed potencjalną kolizją z obiekta rozmieszczonymi wokół samochodu;
4. **Obstacle Aware Acceleration** - automatycznie zmniejsza przyspieszenie samochodu po wykryciu przeszkody podczas jazdy z małą prędkością;
5. **Blind Spot Monitoring** - ostrzega przed obiekta w martwym polu, gdy samochód wykonuje manewr zmiany pasa;
6. **Lane Departure Avoidance** - stosuje manewry korygujące w celu utrzymania się na swoim pasie ruchu;
7. **Emergency Lane Departure Avoidance** - przywraca pojazd z powrotem na swój pas ruchu, gdy wykrywa że samochód zjeżdża ze swojego pasa i może dojść do kolizji na drodze.

Tesla Vision

Tesla jest obecnie liderem na rynku pod względem rozwoju technologii wykorzystania kamer w samochodach autonomicznych. Ich system o nazwie **Tesla Vision** to nowa wersja systemu **Tesla Autopilot**, w której całą percepcję oparto tylko na jednym typie czujnika: kamerze. Począwszy od maja 2021, system ten jest montowany w samochodach Tesli sprzedawanych w Ameryce Północnej [73].

Pomysł całkowitej rezygnacji z radarów oraz LiDAR-ów wzbudził wiele kontrowersji w środowisku osób zainteresowanych rozwojem technologii samochodów autonomicznych. W dość obszernym artykule [74] z maja 2021 roku, autor wysuwa szereg trafnych argumentów przeciwko decyzji Tesli o rezygnacji z montowania radarów w niektórych ich modelach. Radary doskonale uzupełniają dane z kamer, zwłaszcza w sytuacji kiepskiej widoczności na drodze. Z radarami łatwiej jest generować trójwymiarową reprezentację otoczenia samochodu, ponieważ uzyskujemy ją bezpośrednio z odczytów radaru (w przeciwieństwie do obrazu z kamery, gdzie informacja o głębi musi zostać obliczona przez oprogramowanie pojazdu). Sygnał z radarów jest w stanie prześlimać się i wrócić pod samochodami jadącymi bezpośrednio przed samochodem autonomicznym, dzięki czemu samochód może zareagować z większym wyprzedzeniem i uniknąć kolizji, do której doszłoby w innym wypadku.

Autora nie przekonują argumenty typu „*człowiek też nie ma radarów i jakoś sobie radzi z prowadzeniem samochodu*”, ponieważ ignorują one fundamentalny problem, jaki przeszkaździ w osiągnięciu pełnej autonomii pojazdu: nasze oprogramowanie wciąż jest bardzo dalekie od tego, co potrafi ludzki mózg. Dotyczy to zarówno umiejętności sprawnego interpretowania obrazów, jak również logicznego rozumowania oraz zdolności adaptacji do nowego środowiska i radzenia sobie w niestandardowych scenariuszach drogowych. Na każdym z tych pól oprogramowanie wykazuje poważne braki i niedociągnięcia w stosunku do możliwości człowieka. Skoro Tesla nie wprowadziła żadnej rewolucji w tym zakresie, to pozbycie się radarów dających cenne źródło informacji należy uznać za regresję w możliwościach systemu. Zwłaszcza, że nie wprowadzono żadnej redundancji w postaci dodatkowych kamer, dlatego też przy awarii chociaż jednej z nich lub nawet zasłonięciu soczewki (np. z powodu zalegającego śniegu lub ptasich odchodów), percepja samochodu ulega znaczącemu pogorszeniu.

Pomimo powyższych zastrzeżeń wygląda na to, że Tesla nieustannie doskonali swój produkt i wraz z kolejnymi aktualizacjami oprogramowania ich samochody radzą sobie coraz lepiej - również w trudnych warunkach pogodowych, co zostało opisane w artykule [40]. Autor przedstawił odczucia jednego z użytkowników samochodu Tesli, który zauważał po-

stępującą poprawę zachowania samochodu podczas jazdy w intensywnym deszczu. Poprawa ta następowała (w mniejszym lub większym stopniu) po każdej aktualizacji oprogramowania i objawiała się mniejszym i mniej gwałtownym wytracaniem prędkości pojazdu.

Podczas konferencji CVPR z 2021 roku, Andrej Karpathy (szef działu AI w Tesli) wyjaśnił dlaczego Tesla zrezygnowała z wykorzystania czujników LiDAR w swoich samochodach [26]. Karpathy wskazał między innymi na problem przygotowania i utrzymania dokładnych i szczegółowych map 3D terenu, po którym ma się poruszać samochód wyposażony w czujniki LiDAR. „*Musisz wstępnie zmapować środowisko za pomocą LiDAR-u, a następnie stworzyć mapę w wysokiej rozdzielczości i wstawić tam wszystkie pasy i sygnalizacje świetlne*” - powiedział Karpathy. Dodał również: „*Zbieranie, budowanie i utrzymywanie map LiDAR-owych w wysokiej rozdzielczości jest nieskalowalne. Bardzo trudno utrzymać tę infrastrukturę w stanie aktualnym*”. Samochody Tesli nie potrzebują mieć ręcznie przygotowanych map 3D, ponieważ same potrafią wygenerować sobie takie mapy. „*Wszystko co się dzieje, dzieje się po raz pierwszy w samochodzie, na podstawie nagrania z ośmiu kamer otaczających samochód*” - stwierdził Andrej Karpathy. Samochód musi samodzielnie rozpoznać otoczenie, wykryć i zinterpretować wszystkie elementy infrastruktury drogowej oraz ocenić, które z nich są dla niego istotne. Wszystko to bez jakiegokolwiek wcześniejszej informacji na temat drogi, po której samochód będzie się przemieszczać.

Platforma sprzętowa

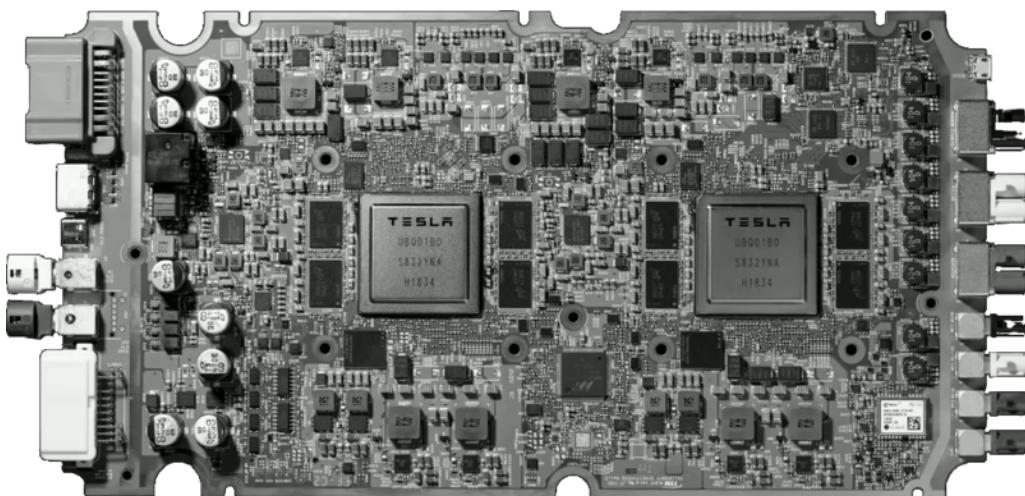
Dodatkowym warunkiem, jaki musi spełniać samochód autonomiczny, jest maksymalny dopuszczalny czas obliczeń podczas jazdy samochodem. Wszystko musi odbywać się w czasie rzeczywistym, a zbyt wielkie opóźnienia w przetwarzaniu danych mogłyby doprowadzić do wystąpienia niebezpiecznej sytuacji na drodze. Na szczęście Tesla dysponuje bardzo dobrym sprzętem, opracowanym we współpracy z firmą Samsung [53, 75]. Nosi nazwę **Full Self-Driving Computer** (FSD) i w momencie swojej premiery w 2019 roku był określany jako „*najbardziej zaawansowany komputer przeznaczony do jazdy autonomicznej*”. Wygląd komputera został zaprezentowany na rysunku 1.5.

Chipset Samsunga osiąga wydajność 144 TOPS i składa się z dwóch redundantnych, niezależnych pakietów - każdy z nich dysponuje własną pamięcią DRAM, układami pamięci flash oraz zasilaniem. „*Każdy z pakietów uruchamia i utrzymuje własną instancję systemu operacyjnego*” - mówi Pete Bannon, projektant układów scalonych pracujący dla Tesli. Dwa pakiety niezależnie od siebie dokonują obliczeń akcji do wykonania, a następnie wymieniają się ze sobą wynikami obliczeń. Żadna akcja nie zostanie zlecona do wykonania kontrolerom samochodu do czasu jej uzgodnienia i zatwierdzenia przez obydwa pakiety. Układ zachowuje

swoją zdolność działania także wtedy, gdy jeden z pakietów (lub któraś z jego części) ulegnie awarii. Elon Musk stwierdził, że: „*nawet jeśli część układu ulegnie awarii, samochód wciąż może kontynuować jazdę*”.

FSD składa się z 6 miliardów tranzystorów i 250 milionów bramek logicznych. Jego moduły pamięci LPDDR4 RAM zapewniają przepustowość 68 GB/s, a zintegrowane procesory przetwarzania obrazów pracują z prędkością do 1 gigapiksela na sekundę. Dodatkowo w zestawie są dwa akceleratory sieci neuronowych, taktowane z częstotliwością 2 GHz i wyposażone w 32 MB pamięci SRAM. Akceleratory mogą dodawać macierze oraz przetwarzać do 1 TB danych na sekundę, a ich wydajność wynosi 36 TOPS dla każdego z nich (czyli łącznie 72 TOPS). Innym ważnym komponentem jest wbudowany układ zabezpieczający, którego zadaniem jest blokowanie kodu nieposiadającego podpisu kryptograficznego Tesli.

Podczas testów zamkniętych, przeprowadzonych przez firmę Tesla, okazało się że komputer FSD jest w stanie przetwarzać 2300 klatek na sekundę - podczas gdy poprzednia generacja komputera osiągała zaledwie 110 klatek na sekundę.



Rysunek 1.5: Komputer Tesla FSD

Źródło: <https://www.autopilotreview.com/tesla-custom-ai-chips-hardware-3>

Podsumowanie

Tesla Autopilot to niezwykle ciekawy i wyróżniający się na tle konkurencji projekt. Przedstawiciele Tesli wnieśli ważny głos do dyskusji nad rozwojem technologii samochodów autonomicznych, zwłaszcza w kontekście wykorzystania obrazu z kamery jako głównego źródła danych dla modelu percepcji pojazdu. Wkład firmy w rozwój technologii jest bezcenny, nawet jeśli zastosowane podejście okaże się ślepym zaułkiem w drodze do osiągnięcia w pełni autonomicznego systemu sterowania samochodem.

Rozdział 2

Konwolucyjne sieci neuronowe

Konwolucyjna sieć neuronowa (CNN) jest specjalnym typem sieci przystosowanej do analizy danych z obrazu [8, 77]. Została zaprojektowana do automatycznego i adaptacyjnego uczenia się o przestrzennych wzorcach obecnych na obrazie oraz o hierarchii, jaką te wzorce tworzą. Rozpoznawanie wzorców przez sieć rozpoczyna się od najprostszych kształtów, a z każdą kolejną warstwą konstruowane są wzorce wyższego poziomu.

Tradycyjne sieci neuronowe były fatalnie przystosowane do analizy obrazów. Aby to zrozumieć, należy zobaczyć w jaki sposób obrazy są przechowywane w formie cyfrowej. Każdy obraz to macierz liczb rzeczywistych o wymiarach **szerokość × wysokość × liczba kanałów**. Z kolei standardowe sieci neuronowe oczekują na wejście danych w formacie tablicy jednowymiarowej. Problemem nie jest niezgodność reprezentacji danych, tylko fatalna skalowość rozwiązania - w samej tylko warstwie wejściowej, sieć musi posiadać liczbę neuronów równą liczbie elementów w macierzy. Wraz ze wzrostem rozdzielczości analizowanych obrazów, lawinowo rośnie liczba parametrów sieci. Dla macierzy o wymiarach $200 \times 200 \times 3$ warstwa wejściowa musiałaby składać się z **120000 neuronów**, a w rzeczywistości wykorzystuje się obrazy o znacznie większych rozdzielczościach. Działanie na tak ogromnej liczbie parametrów jest kłopotliwe i znacznie utrudnia wytrenowanie sieci.

2.1 Architektura

W konwolucyjnych sieciach neuronowych obraz może być przetwarzany przy użyciu znacznie mniejszej liczby parametrów, ponieważ w danej chwili analizowany jest tylko jego fragment. Sieć neuronowa przechodzi po całym obrazie, skanując go fragment po fragmencie i wyszukując wzorce zakodowane w parametrach sieci. Parametry te podlegają zmianie w wyniku treningu sieci. Sieć konwolucyjna jest skonstruowana z kilku typów warstw:

1. **Warstw Konwolucyjnych** (z ang. *Convolutional Layers*),
2. **Warstw ReLU**,
3. **Warstw Łączniowych** (z ang. *Pooling Layers*),
4. **Warstw Normalizacyjnych** (z ang. *Normalization Layers*),
5. **Warstw w Pełni Połączonych** (z ang. *Fully-Connected Layers*).

2.1.1 Warstwy Konwolucyjne

Każda warstwa konwolucyjna składa się z tzw. **filtrów** (zwanych również **kernelami** lub **jądrami**). Każdy filtr jest macierzą o niewielkich wymiarach pod względem szerokości i wysokości, ale w pełni obejmującej trzeci wymiar, czyli liczbę kanałów na piksel. Filtr jest przykładany po kolejno do każdej możliwej pozycji na macierzy wejściowej i wykonywane są obliczenia, analogiczne do przykładu przedstawionego na rysunku 2.1.

$$\begin{array}{|c|c|c|c|c|} \hline
 7 & 2 & 3 & 3 & 8 \\ \hline
 4 & 5 & 3 & 8 & 4 \\ \hline
 3 & 3 & 2 & 8 & 4 \\ \hline
 2 & 8 & 7 & 2 & 7 \\ \hline
 5 & 4 & 4 & 5 & 4 \\ \hline
 \end{array}
 \begin{array}{c}
 *
 \end{array}
 \begin{array}{|c|c|c|} \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 \end{array}
 =
 \begin{array}{|c|c|c|} \hline
 6 & & \\ \hline
 & & \\ \hline
 & & \\ \hline
 \end{array}$$

$$\begin{aligned}
 & 7 \times 1 + 4 \times 1 + 3 \times 1 + \\
 & 2 \times 0 + 5 \times 0 + 3 \times 0 + \\
 & 3 \times -1 + 3 \times -1 + 2 \times -1 \\
 & = 6
 \end{aligned}$$

Rysunek 2.1: Przykład działania filtra w warstwie konwolucyjnej

Źródło: <https://www.v7labs.com/blog/convolutional-neural-networks-guide>

Parametry filtru (czyli elementy macierzy) definiują wzorzec przestrzenny, a ponieważ filtr jest przesuwany po całym obrazie, to zdefiniowany wzorzec może być wykryty w każdym miejscu obrazu. Wyniki zastosowania filtra zapisywane są do macierzy nazywanej **mapą cech** (z ang. *feature map*). W każdej warstwie konwolucyjnej może być zdefiniowanych wiele filtrów, co oznacza, że sieć uczy się wyszukiwać wiele wzorców na raz, niezależnie od siebie.

Warstwy konwolucyjne charakteryzują się kilkoma cechami:

1. **Łącznością lokalną** (z ang. *local connectivity*) - połączenia między neuronami występują tylko w obszarze definiowanym przez nakładane filtry.
2. **Uporządkowaniem przestrzennym** (z ang. *spatial arrangement*) - ilość neuronów na wyjściu z warstwy, jak również sposób ich uporządkowania. Rozmiar wyjścia jest zdefiniowany poprzez trzy hiperparametry:
 - (a) **Głębokość** (z ang. *depth*) - jest równa liczbie filtrów używanych w danej warstwie.

- (b) **Krok** (z ang. *stride*) - oznacza liczbę pikseli, o jaką przeskakuje filtr po każdym dopasowaniu. Jeśli krok ma wartość 1, to filtr każdorazowo jest przesuwany tylko o jeden piksel. Im krok jest większy, tym mniejsze przestrzennie wolumeny będą produkowane.
 - (c) **Wypełnienie zerami** (z ang. *zero-padding*) - pozwala na kontrolę rozmiaru wolumenu wyjściowego poprzez dopełnienie zerami granic danych wejściowych.
3. **Udostępnianiem parametrów** (z ang. *parameter sharing*) - ta sama macierz wag działa na wszystkie neurony w określonej mapie cech. To daje sieci możliwość rozpoznawania tych samych wzorców na różnych obrazach, nawet jeśli wzorce te są względem siebie przesunięte, obrócone lub przeskalowane o jakąś wartość.

2.1.2 Warstwy ReLU

W tych warstwach wykorzystywana jest funkcja aktywacji ReLU [12], której zadaniem jest podmiana wartości ujemnych na zero. To celowy zabieg, aby uniknąć ryzyka wyzerowania się sumowanych wartości.

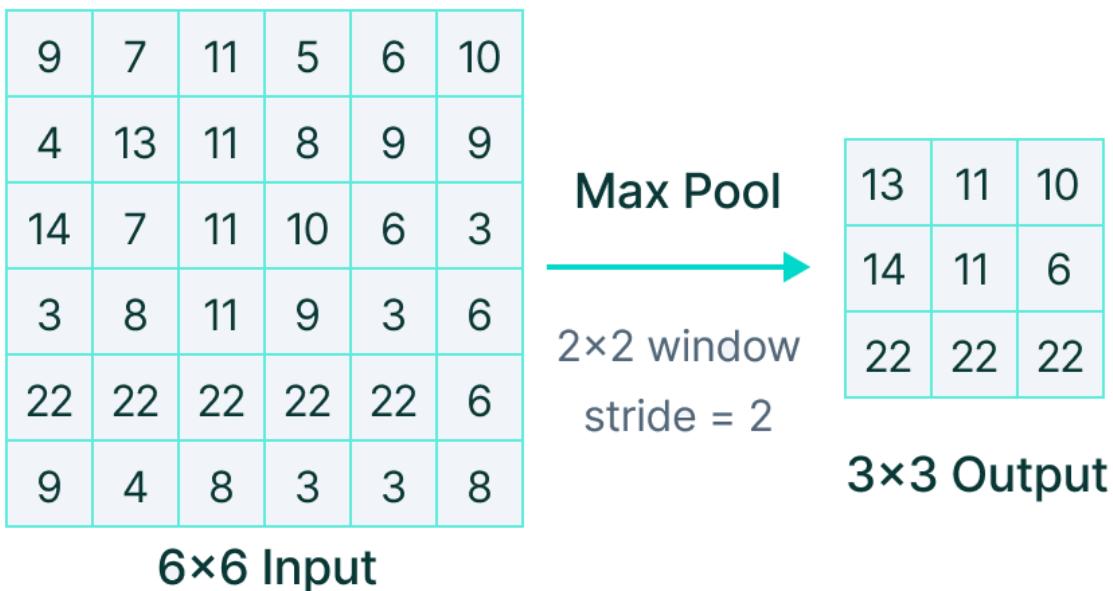
2.1.3 Warstwy Łączniowe

Są zwykle umieszczane pomiędzy dwoma warstwami konwolucyjnymi, aby zmniejszyć rozmiar przestrzenny mapy cech generowanej na wyjście z warstwy konwolucyjnej. Operują na każdym przekroju wolumenu wejściowego z osobna, zmniejszając jego rozmiar. Na sam koniec, pomniejszone przekroje są z powrotem układane w wolumen. Każda warstwa łączniowa posiada dwa hiperparametry: **rozmiar okna** oraz **krok**. Wartości wyjściowe są obliczane według jednego z dwóch schematów - albo jest brane maksimum z wartości pokrytych oknem, albo też obliczana jest ich średnia. Odpowiednio do wybranego schematu, warstwa łączniowa jest typu *Max Pooling* lub *Average Pooling*.

Warstwa *Max Pooling* wybiera największy element z każdego okna mapy cech, co skutkuje powstaniem mapy cech posiadającej najbardziej dominujące cechy poprzedniej mapy.

Warstwa *Average Pooling* oblicza wartość średnią elementów należących do okna mapy cech, więc wynikiem działania jest mapa cech posiadająca uśrednione cechy oryginalnej mapy.

Na rysunku 2.2 zaprezentowano przykład działania warstwy łączniowej typu *Max Pooling*, gdzie rozmiar okna wynosi 2×2 i krok jest równy 2. Warstwy *Max Pooling* są w praktyce częściej wykorzystywane, ponieważ dają większą skuteczność w uczeniu sieci.



Rysunek 2.2: Przykład działania warstwy łączniowej

Źródło: <https://www.v7labs.com/blog/convolutional-neural-networks-guide>

2.1.4 Warstwy Normalizacyjne

Służą normalizacji wyjścia z warstw poprzednich. Warstwa normalizacyjna jest dodawana pomiędzy warstwą konwolucyjną a łączniową, dzięki czemu każda z warstw może się uczyć niezależnie od siebie i w ten sposób uniknąć nadmiernego dopasowania modelu. Pomimo tych zalet, warstwy normalizacyjne są rzadko wykorzystywane w zaawansowanych architekturach, ponieważ mają niewielki wpływ na efektywność uczenia takich sieci.

2.1.5 Warstwy w Pełni Połączone

Stanowią ostatni etap w przepływie danych przez sieć konwolucyjną. Wygenerowane wcześniej mapy cech zostają przekształcone do postaci jednowymiarowej i wprowadzone na wejście do zwykłej, w pełni połączonej sieci neuronowej. Zadaniem tej sieci jest wygenerowanie ostatecznego wyniku zwracanego przez sieć.

2.1.6 Funkcja aktywacji w ostatniej warstwie

Ostatnia warstwa sieci konwolucyjnej ma zazwyczaj inną funkcję aktywacji, a jej wybór jest podykowany typem zadania wykonywanego przez sieć neuronową. Z tego powodu najczęściej stosuje się funkcje:

1. **Softmax** [76] - dla klasyfikacji wieloklasowej;
2. **Sigmoidalna** [55] - dla klasyfikacji binarnej;
3. **Tożsamościowa** - dla regresji do wartości ciągłych.

2.1.7 Łączenie w całość

Znając elementy składowe konwolucyjnej sieci neuronowej, przyjrzyjmy się jak one są ze sobą połączone. Z artykułu [48] dowiadujemy się, że najpopularniejszym schematem tworzenia sieci konwolucyjnej jest schemat następujący:

INPUT \rightarrow ((CONV \rightarrow RELU) * N \rightarrow POOL?) * M \rightarrow (FC \rightarrow RELU) * K \rightarrow FC \rightarrow LAF
gdzie:

- INPUT - warstwa wejściowa;
- CONV - warstwa konwolucyjna;
- RELU - warstwa ReLU;
- POOL? - warstwa łączeniowa. Znak zapytania oznacza, że jest ona opcjonalna;
- FC - warstwa w pełni połączona;
- LAF - funkcja aktywacji warstwy ostatniej;
- $N \geq 0, M \geq 0, K \geq 0$.

2.2 Trening

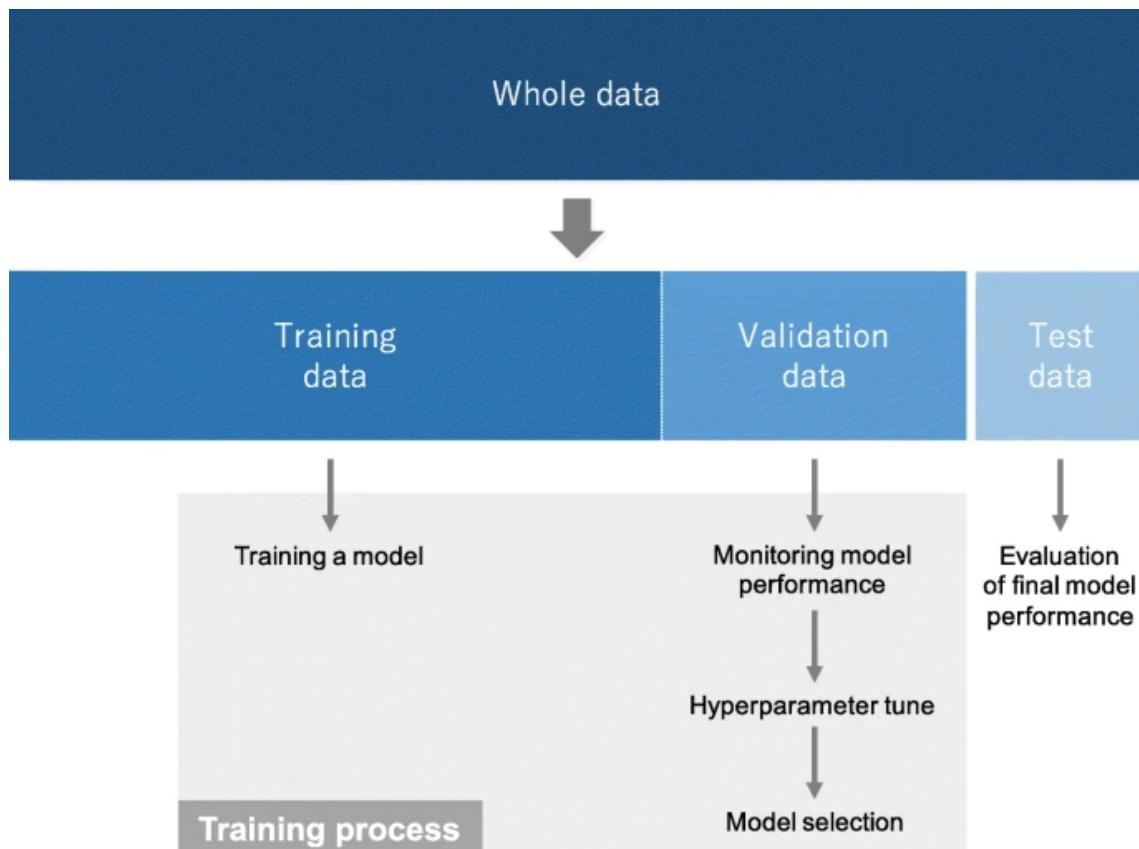
Trening sieci jest procesem poszukiwania odpowiednich filtrów dla warstw konwolucyjnych oraz odpowiednich wag dla warstw w pełni połączonych. Algorytm propagacji wstecznej [41] jest powszechnie wykorzystywany metodą nauki sieci neuronowej, gdzie funkcja straty [13] oraz algorytm spadku gradientu [44] odgrywają kluczową rolę. Skuteczność sieci jest obliczana z funkcji straty przy propagacji wprzód dla danych ze zbioru treningowego, a parametry do wyuczenia (czyli filtry oraz wagi) są aktualizowane zgodnie z wynikami uzyskanymi przez funkcję straty, wspomagając się przy tym algorytmami propagacji wstecznej oraz spadku gradientu.

Przy nauce sieci neuronowych powszechnie korzysta się z dwóch funkcji straty: **entropii krzyżowej** (dla klasyfikacji wieloklasowej) [11] oraz **błędu średniokwadratowego** (dla regresji do wartości ciągłych). Funkcja straty jest jednym z hiperparametrów sieci, który musi zostać wybrany przed rozpoczęciem treningu.

Aby trening sieci mógł się udać, należy zgromadzić odpowiednio duży oraz zróżnicowany zbiór danych. Następnie trzeba ten zbiór podzielić na 3 podzbiory:

1. **Treningowy** - największy z podzbiorów. Jest używany przy treningu sieci, gdzie funkcja straty jest obliczana podczas propagacji wprzód i parametry sieci są aktualizowane podczas propagacji wstecznej.

2. **Walidacyjny** - wykorzystywany do nadzoru skuteczności sieci podczas trwania jej treningu oraz do dostrajania hiperparametrów sieci.
3. **Testowy** - powinien zostać użyty tylko jeden raz: po zakończeniu treningu sieci. Zbiór testowy służy ocenie skuteczności wytrenowanej sieci.



Rysunek 2.3: Podział zbioru danych użytych do wyuczenia sieci

Źródło:

<https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9>

Podział zbioru danych został zaprezentowany na rysunku 2.3. Oddzielne zestawy do walidacji i testów są konieczne, ponieważ zawsze istnieje potrzeba dostrojenia hiperparametrów sieci. Proces ten dokonuje się w oparciu o skuteczność sieci na zbiorze walidacyjnym, co sprawia że część danych z tego zbioru „wycieka” do samego modelu. W rezultacie, sieć może osiągnąć efekt nadmiernego dopasowania się [32] do zbioru walidacyjnego, mimo iż nie jest na nim uczona. Dlatego też do końcowej oceny skuteczności sieci stosuje się całkowicie osobny zbiór danych, który nie był wcześniej „widziany” przez sieć neuronową. W ten sposób można zweryfikować poziom osiągniętej skuteczności na wytrenowanym modelu oraz sprawdzić, czy model ten nie został przeuczony na danych treningowych lub walidacyjnych.

2.3 Przykłady z literatury

Badając zagadnienie sieci konwolucyjnych, nie sposób nie wspomnieć o kilku wypracowanych modelach, które miały niebagatelny wkład w rozwój dziedziny:

1. LeNet

Francuski naukowiec-informatyk Yann LeCun jest znany przede wszystkim ze swoich dokonań w zakresie optycznego rozpoznawania znaków oraz wizji komputerowej. W latach 90. XX wieku wraz ze swoimi współpracownikami zaprojektował wiele architektur, lecz największą sławę przyniosła mu sieć o nazwie LeNet. W obszernym artykule naukowym z 1998 roku [47] została ona dokładnie przedstawiona i opisana. Sieć LeNet mogła być używana m.in. do rozpoznawania odręcznie napisanych cyfr oraz kodów pocztowych.

2. AlexNet

Była to pierwsza praca, która tak bardzo spopularyzowała wykorzystanie konwolucyjnych sieci neuronowych w wizji komputerowej. Stworzona przez trójkę naukowców z Uniwersytetu w Toronto i opisana w artykule [42], AlexNet została wystawiona do konkursu *ImageNet ILSVRC challenge* w 2012 roku i zdobyła swoich rywali, osiągając znacznie lepsze wyniki. Sieć miała architekturę podobną do LeNet, lecz była większa, głębsza i posiadała kilka warstw konwolucyjnych następujących bezpośrednio po sobie (w tamtych czasach normą było, aby po każdej warstwie konwolucyjnej następowała warstwa łączniowa). AlexNet była tak ważnym kamieniem milowym, że jej nazwa pojawia się w każdej poważnej publikacji naukowej traktującej o historii rozwoju sieci konwolucyjnych.

3. GoogLeNet

Zwycięsca konkursu *ILSVRC 2014*, stworzony przez zespół badawczy zdominowany przez pracowników firmy Google [61]. Jego najważniejszym wkładem w rozwój dziedziny było opracowanie tzw. **Modułu Incepcji** (z ang. *Inception Module*), który pozwolił na radykalne zmniejszenie liczby parametrów sieci. Dla porównania, sieć GoogLeNet wystawiona do konkursu posiadała „zaledwie” 4 miliony parametrów, podczas gdy AlexNet miała ich aż 60 milionów. Co więcej, GoogLeNet zastąpił większość warstw w pełni połączonych warstwami łączniowymi typu *Average Pooling*, co pozwoliło na pozbycie się parametrów mających niewielki wpływ na zachowanie sieci. Istnieje kilka wersji sieci GoogLeNet, najpopularniejszą z nich jest **Inception-v4**.

4. VGGNet

Stworzona przez Karen Simonyan i Andrew Zissermana, wykazała że odpowiednia

głębokość jest kluczowym elementem dobrego zachowania sieci [57]. Najlepsza sieć VGGNet posiada 16 warstw i charakteryzuje się jednorodną architekturą, w której od początku do końca wykorzystywane są tylko konwolucje o wymiarach 3×3 oraz łączenia o wymiarach 2×2 . Największą wadą sieci jest jej ogromna liczba parametrów - dla wersji z 16 warstwami było to 140 milionów. Większość parametrów jest w warstwach w pełni połączonych, lecz badacze po pewnym czasie odkryli, że warstwy te mogą zostać usunięte bez znacznej szkody dla skuteczności sieci.

5. ResNet

Zwycięsca konkursu *ILSVRC* w 2015 roku, opracowany przez grupę badawczą z firmy Microsoft [34]. Wyjątkowość tej architektury polega na wprowadzonych innowacjach, dzięki którym stał się możliwy trening sieci o znacznie większej liczbie warstw ukrytych. Było to ważne wydarzenie, ponieważ otworzyło zupełnie nowe możliwości przed badaczami. Przed opublikowaniem sieci ResNet, konwolucyjne sieci neuronowe zdawały się być nie do wytrenowania przy więcej niż kilkunastu warstwach ukrytych. Powodem takiego stanu rzeczy było występowanie zjawiska **zaniku i eksplozji gradien-tu** [10], z którym naukowcy próbowali walczyć, lecz nieskutecznie. Dopiero twórcy architektury ResNet znaleźli odpowiedź na rozwiązań tego problemu.

2.4 Obszary zastosowań

Konwolucyjne sieci neuronowe są obecnie wykorzystywane z sukcesem w wielu zada niach z zakresu wizji komputerowej [43]. Wśród nich można wymienić:

1. **Rozpoznawanie twarzy** - sieci konwolucyjne są powszechnie używane do wykrywania twarzy na obrazach. Sieć może wykrywać i rozpoznawać z dużą dokładnością różne cechy twarzy, takie jak oczy, nos i usta, redukując przy tym zniekształcenia powstałe na skutek złego oświetlenia obrazu.
2. **Rozpoznawanie emocji** - sieci konwolucyjne okazały się bardzo dobrze sprawdzać przy klasyfikacji różnych wyrazów twarzy, odzwierciedlających emocje człowieka. Sieciom w klasyfikacji nie przeszkadza zmienna pozycja twarzy wobec kamery oraz zmienne warunki oświetleniowe.
3. **Wykrywanie obiektów** - odbywa się poprzez klasyfikację obiektów na podstawie ich kształtu, kolorów oraz innych cech charakterystycznych dla danej grupy obiektów. Odpowiednio wytrenowana sieć konwolucyjna potrafi wykrywać bardzo szeroką gamę obiektów. Do wykrywania obiektów wykorzystuje się techniki segmentacji semantycznej oraz instancyjnej.

4. **Samochody autonomiczne** - sieci konwolucyjne są stosowane jako podstawowy komponent oprogramowania modelu percepcji opartego na obrazie z kamer. Trening sieci odbywa się przy wykorzystaniu techniki zwanej *uczeniem ze wzmacnieniem*, która koncentruje się na pozytywnych i negatywnych informacjach zwrotnych ze środowiska, aby poprawić zachowanie sieci w danych sytuacjach.
5. **Tłumaczenie automatyczne** - sieci konwolucyjne po odpowiednim treningu znakomicie sobie radzą z automatycznym tłumaczeniem, a przetłumaczone teksty charakteryzują się wysokim poziomem zgodności z oryginałem.
6. **Analiza danych medycznych** - modele uczenia maszynowego znalazły szerokie pole zastosowań w branży medycznej. Konwolucyjne sieci neuronowe mogą być użyte np. do identyfikacji guzów oraz innych nieprawidłowości na obrazach rentgenowskich. Są również stosowane przy wykrywaniu nowotworów na obrazach z badania tomografii komputerowej. Wytrenowana sieć jest w stanie wykrywać raka z dużą skutecznością, dochodzącą do 95%. Dla porównania, lekarze-patolodzy biorący udział w badaniach osiągali skuteczność poniżej 90%.
7. **Uwierzytelnianie biometryczne** - biometryczne uwierzytelnianie tożsamości użytkownika odbywa się na podstawie identyfikacji zestawu cech fizycznych, związanych z daną osobą. Sieci konwolucyjne świetnie się do tego sprawdzają.
8. **Klasyfikacja dokumentów** - sieci konwolucyjne mogą klasyfikować dokumenty do różnych kategorii. Kategoria może być identyfikowana na podstawie treści dokumentu, rozkładu tekstu lub wielu innych cech. Konwolucyjne sieci neuronowe dla lepszego zrozumienia dokumentu mogą analizować zarówno tekst, jak i obraz. Modele CNN są również z powodzeniem stosowane do generowania streszczeń i podsumowań dokumentów na podstawie ich zawartości.
9. **Segmentacja trójwymiarowych danych medycznych** - konwolucyjne sieci neuronowe znalazły swoje zastosowanie także w segmentacji skanów obrazowania medycznego, takich jak obrazy z badania tomografii komputerowej lub rezonansu magnetycznego. Wytrenowana sieć konwolucyjna potrafi pobrać wycinek obrazu ze skanu 3D i określić, gdzie w obrębie tego obrazu znajdują się poszczególne rodzaje tkanek, a także konkretne narządy człowieka.

2.5 Problemy i wyzwania na przyszłość

Przed naukowcami pracującymi nad rozwojem sieci konwolucyjnych stoi szereg trudnych wyzwań, z którymi będą zmuszeni się zmierzyć.

2.5.1 Wykorzystanie pamięci

Jednym z najważszych gardeł do rozważenia przy projektowaniu sieci konwolucyjnych jest ilość oraz rodzaj dostępnej pamięci, oferowanej przez współczesne karty graficzne. Większość kart jest ograniczona rozmiarem pamięci nieprzekraczającym kilku gigabajtów, co może być niewystarczające dla bardzo głębokich sieci konwolucyjnych z ogromną liczbą parametrów. Ponadto na sieci konwolucyjne mogą zostać nałożone restrykcyjne wymogi maksymalnego czasu odpowiedzi, co oznacza że wydajność sprzętu na którym sieci są uruchomione nagle zyskuje na znaczeniu.Więcej rozważeń na temat efektywnego wykorzystania pamięci przez konwolucyjne sieci neuronowe można znaleźć w artykule [59].

2.5.2 Zagrożenia bezpieczeństwa

Sieci konwolucyjne są obecnie szeroko wykorzystywany rozwiążaniem, począwszy od edytorów zdjęć i wideo, a skończywszy na oprogramowaniu medycznym i samochodach autonomicznych. Chociaż te sieci niesamowicie zbliżyły się w ostatnich latach do postrzegania świata w sposób zarezerwowany dotychczas tylko zwierzętom i ludziom, to wciąż istnieje wiele sytuacji w których sieć neuronowa całkowicie traci swoją skuteczność i generuje błędne wyniki. Głównym źródłem takich problemów są cyberataki dokonywane przez hakerów, a lwią część ataków stanowią tzw. **ataki kontradyktoryjne** (z ang. *adversarial attacks*) [25]. Ataki te polegają w skrócie na dokładaniu do obrazu szumów niewidocznych dla człowieka, które wprowadzają sieć neuronową w błąd. Badania prowadzone w tej dziedzinie przyniosły już wiele zaskakujących wyników. Przykładowo, naukowcy z grupy badawczej LabSix w 2017 roku opublikowali artykuł [7] w którym pokazali, że odpowiednio zmodyfikowana zabawka w kształcie żółwia potrafi całkowicie zmylić popularną sieć neuronową klasyfikującą obrazy, czyli *Google Inception-v3*. Modyfikacje są niezauważalne dla człowieka, a polegały na wprowadzeniu szumów do malowania pokrywającego żółwia. W efekcie, niezależnie od kąta ustawienia żółwia, sieć neuronowa będzie uparcie klasyfikować go jako karabin. Rysunek 2.4 przedstawia stopklatkę z filmu zawartego w opublikowanym artykule.

Bardziej niebezpiecznym przykładem niecnego wykorzystania ataków kontradyktoryjnych są manipulacje przy znakach drogowych, które mają wprowadzać samochody autonomiczne w błąd. Jak wykazały wyniki badań opublikowane w artykule [27], wystarczy nakleić kilka niepozornych nalepek na znak STOP, aby system rozpoznawania znaków drogowych zaczął go klasyfikować inaczej, np. jako znak ograniczenia prędkości. Testy zostały przeprowadzone na systemie wytrenowanym przez autorów badania, a nie na jakimś komercyjnie dostępnym modelu samochodu autonomicznego, dlatego można założyć że te konkretne ata-

ki nie zadziałyby na systemach zamontowanych w samochodach Tesli lub Waymo. Niemniej jednak, ponieważ algorytmy na których opierają się systemy komercyjne nie różnią się zbytnio w swej zasadzie działania (mogą być co najwyżej inaczej wytrenowane), dlatego należy przyjąć, że tego typu ataki są wciąż jak najbardziej możliwe.



Rysunek 2.4: Przykład ataku kontradydktoryjnego - żółw klasyfikowany jako karabin

Źródło: <https://www.labsix.org/physical-objects-that-fool-neural-nets>

Pewną iskierką nadziei na rozwiązywanie tego palącego problemu są badania naukowców z MIT oraz laboratorium MIT-IBM Watson [24], gdzie badacze przedstawiają nową architekturę o nazwie VOneNet, opartą o połączenie aktualnych technik głębokiego uczenia maszynowego z aktualnym stanem wiedzy na temat neurobiologii. Uzyskana w ten sposób architektura wykazuje większą odporność na ataki kontradydktoryjne oraz charakteryzuje się bardziej przewidywalnym zachowaniem.

2.5.3 Klasyfikacja obiektów z przekształceniami

Jednym z wielu wyzwań w dziedzinie wizji komputerowej jest radzenie sobie ze zmiennością danych występujących w świecie rzeczywistym. Ludzki narząd wzroku potrafi prawidłowo rozpoznać obiekt na obrazie, nawet jeśli:

- obiekt jest fotografowany pod różnymi ujęciami;
- obiekt jest przedstawiony na różnych tłaach;
- zdjęcie zostało zrobione w różnych warunkach oświetleniowych.



Rysunek 2.5: Ta sama rzeźba na pięciu różnych ujęciach.

Źródło: <https://iq.opengenus.org/disadvantages-of-cnn>

Gdy obiekty są częściowo zakryte lub nietypowo oświetlone, ludzki wzrok wciąż jest w stanie znaleźć informacje potrzebne do prawidłowej klasyfikacji obiektu na obrazie. Jednak stworzenie konwolucyjnej sieci neuronowej o takich właściwościach jest zadaniem bardzo trudnym, o czym przekonało się wielu badaczy. Sama pozycja obiektu na fotografii nie stanowi większego problemu, ale jeśli obiekt jest fotografowany pod różnymi ujęciami (jak to ma miejsce na rysunku 2.5), wtedy wiele popularnych modeli sieci konwolucyjnej będzie miało ogromny problem z prawidłowym rozpoznaniem obiektu. Rozwiązaniem tego problemu może być rozszerzenie zbioru danych treningowych o fotografie z odpowiednimi przekształceniami obiektu, o ile zostanie to zrobione w sposób umiejętny. Nie zawsze jednak jest to możliwe, dlatego też potrzeba dalszych badań nad sposobami obsłużenia tego problemu.

2.6 Podsumowanie

Sieci konwolucyjne to niezwykle obszerny i zarazem fascynujący temat, którego dokładne omówienie wykracza poza zakres tej pracy. W niniejszym rozdziale zostały poruszone tylko te wątki, które według autora były kluczowe do zrozumienia podstawowych elementów zagadnienia. W celu dalszego pogłębiania wiedzy zaleca się samodzielna eksplorację tematu, a dobrym punktem wyjścia może być literatura cytowana w tym rozdziale.

Rozdział 3

Projekt systemu i opis narzędzi

W poniższym rozdziale zostaną zaprezentowane kluczowe decyzje projektowe oraz założenia, jakie zostały przyjęte w toku prac nad implementacją systemu. Oprócz tego zostanie również przedstawiony opis narzędzi, które zostały wykorzystane do stworzenia aplikacji.

3.1 Projekt systemu

W fazie projektowania należy zdefiniować główne założenia projektowe oraz odpowiednio zaprojektować architekturę systemu. Podsumowując wyniki uzyskane w tej fazie, autor pracy zdecydował się rozpatrzyć trzy najważniejsze zagadnienia:

1. Definicja **rozwiązywanego problemu**,
2. Lista **celów i założeń systemu**,
3. **Architektura systemu**, rozpatrywana na najwyższym poziomie abstrakcji.

3.1.1 Definicja rozwiązywanego problemu

Zaprojektowany system powinien rozwiązywać problem treningu agentów sterujących samochodem w środowisku symulacji. Samochód powinien pokonywać wyznaczony tor wyścigowy w możliwie najkrótszym czasie. Agenci odbierają informacje o swoim aktualnym położeniu dzięki obrazowi z jednej lub kilku kamer doczepionych do samochodu. W odpowiedzi na te informacje, do środowiska symulacji wysyłane są żądania podjęcia określonych zachowań, np. zmiany poziomu wcisnięcia pedału hamulca. Ponadto, środowisko symulacji na bieżąco przyznaje agentom nagrody i kary w zależności od tego, jak dobrze (lub źle) agent radzi sobie z postawionym zadaniem. Odpowiedni system kar i nagród jest kluczowy dla osiągnięcia zadowalających wyników treningu.

Agentami są konwolucyjne sieci neuronowe, a ich parametry są dostrajane przy użyciu metody uczenia maszynowego zwanej **uczeniem ze wzmacnieniem** (z ang. *reinforcement learning*) [49].

3.1.2 Cele i założenia

1. Stworzony system powinien być możliwie prosty w obsłudze.
2. Wytrenowane modele powinny być zapisywane do pliku o ustalonym formacie. Format pliku musi być rozpoznawany i obsługiwany przez aplikację.
3. Dane konfiguracyjne powinny być dostarczane z zewnętrznego pliku o ustalonym formacie.
4. System powinien posiadać możliwość podglądu „na żywo” postępów treningu sieci podczas jego trwania.

3.1.3 Architektura systemu

System składa się z dwóch zasadniczych modułów: Środowiska Uczenia oraz narzędzia `mlagents_learn`. Moduły komunikują się ze sobą za pośrednictwem mechanizmu komunikacji dostarczanego przez zestaw Unity ML-Agents. Wizualizacja architektury systemu została przedstawiona na rysunku 3.1, a więcej informacji na temat każdego z modułów można uzyskać w rozdziale 4-tym, poświęconym opisowi implementacji systemu.



Rysunek 3.1: Wizualizacja architektury systemu.

3.2 Opis narzędzi

W tej sekcji zamieszczam opis narzędzi wykorzystanych do implementacji systemu.

3.2.1 Unity

Multiplatformowy silnik gier 2D i 3D [46], napisany w językach C/C++. Skrypty silnika należy pisać w języku C#. Gry tworzone na silniku Unity mogą być uruchamiane na bardzo wielu platformach sprzętowych i systemowych [67]:

1. Komputery osobiste (PC):

- Windows,
 - Linux,
 - Mac OS X;
2. Urządzenia mobilne (smartfony):
- Android,
 - iOS;
3. Konsole do gier:
- PlayStation 4,
 - PlayStation 5,
 - Xbox One,
 - Nintendo Switch;
4. i wiele innych.

Unity posiada bardzo atrakcyjne warunki licencyjne - niemal wszystkie funkcje silnika są dostępne za darmo dla twórców nieprzekraczających 100 tysięcy dolarów rocznego dochodu.

W Unity powstało wiele popularnych gier, takich jak:

1. Pokémon Go,
2. Hearthstone: Heroes of Warcraft,
3. Firewatch,
4. The Forest,
5. Car Mechanic Simulator,
6. Gwint: Wiedźmińska Gra Karciana,
7. Syberia 3,
8. i wiele innych.

Warto również wspomnieć o module Unity Asset Store, który zezwala na wykorzystanie płatnych i darmowych komponentów, takich jak tekstury, modele czy skrypty.

3.2.2 C#

Język programowania stworzony przez firmę Microsoft jako konkurencja dla Javy [45]. Jest wysokopoziomowym, obiektowym językiem programowania, ściśle zintegrowanym z platformą .NET (pełniącą rolę frameworka oraz środowiska uruchomieniowego). Charakteryzuje się silnym, statycznym typowaniem. C# pozwala na tworzenie aplikacji desktopowych

(Windows, Linux, MacOS), webowych (poprzez użycie frameworka ASP.NET) oraz multiplatformowych aplikacji mobilnych (poprzez wykorzystanie narzędzia Xamarin). Obecnie jest to jeden z najpopularniejszych języków programowania.

3.2.3 Unity ML-Agents

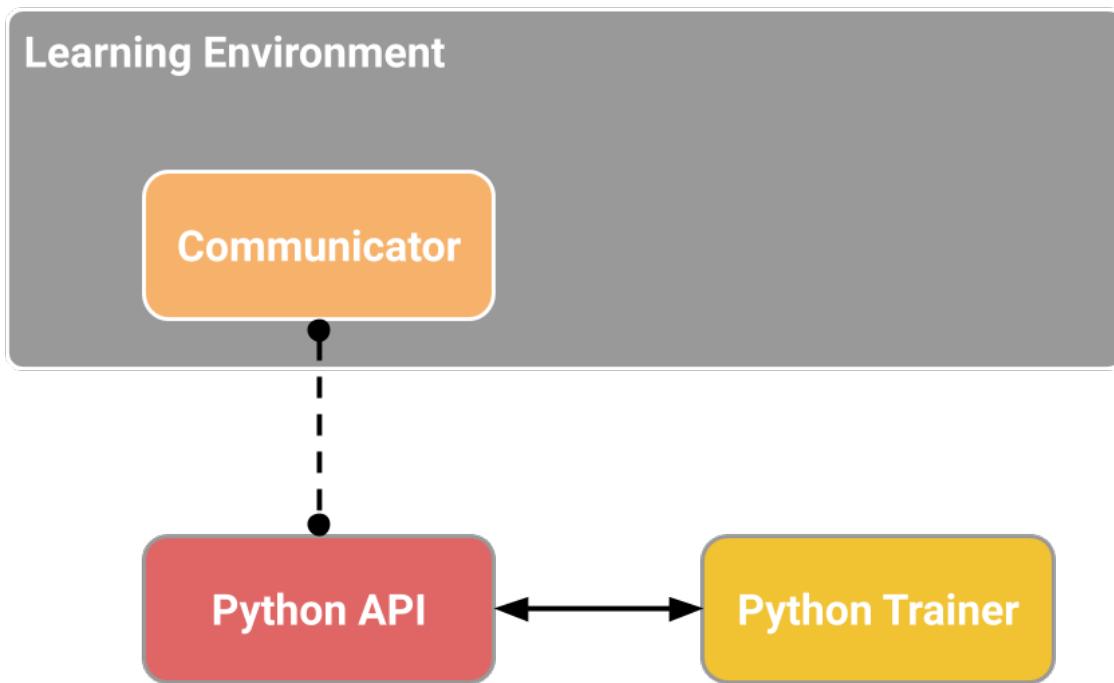
Otwartoźródłowy zestaw narzędzi przeznaczony dla silnika Unity 3D [63]. Został stworzony w celu ułatwienia treningu sieci neuronowych w środowiskach symulowanych komputerowo. Sieci neuronowe mogą być trenowane przy użyciu jednej z kilku możliwych metod uczenia maszynowego:

1. Uczenie ze wzmacnieniem (z ang. *reinforcement learning*) [49]:
 - wykorzystując algorytm PPO [56];
 - wykorzystując algorytm SAC [33];
2. Uczenie przez imitację (z ang. *imitation learning*) [37],
3. Neuroewolucja [50],
4. Inna metoda uczenia maszynowego - zdefiniowana przez użytkownika.

Zestaw Unity ML-Agents składa się z pięciu podstawowych komponentów:

1. Środowisko Uczenia (z ang. *Learning Environment*) - obejmuje scenę Unity oraz wszystkie obiekty znajdujące się na niej. Scena Unity dostarcza środowisko do generowania obserwacji, wykonywania działań i uczenia się.
2. Niskopoziomowe API Pythona (z ang. *Python Low-Level API*) [69] - zawiera niskopoziomowy interfejs programistyczny służący do komunikacji ze Środowiskiem Uczenia. API Pythona nie jest częścią silnika Unity, ale istnieje jako osobny bitt komunikujący się z Unity dzięki Zewnętrznemu Komunikatorowi. API jest zaimplementowane w języku Python i zawarte w pakiecie o nazwie `mlagents_envs`.
3. Zewnętrzny Komunikator (z ang. *External Communicator*) - łączy Środowisko Uczenia z Niskopoziomowym API Pythona.
4. Trenerzy Pythona (z ang. *Python Trainers*) - zawiera wszystkie algorytmy uczenia maszynowego dostarczane przez Unity ML-Agents. Algorytmy zaimplementowane są w języku Python i należą do pakietu o nazwie `mlagents`. Pakiet udostępnia narzędzie wiersza poleceń o nazwie `mlagents-learn`, które obsługuje wszystkie metody treningu i opcje opisane powyżej.
5. Gym Wrapper - opisany w dokumentacji zestawu [68]. Mało istotny dla dalszych rozważań.

Rysunek 3.2 przedstawia diagram opisanych powyżej komponentów zestawu Unity ML-Agents. Gym Wrapper został na nim pominięty.



Rysunek 3.2: Diagram komponentów Unity ML-Agents

Źródło: https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/ML-Agents-Overview.md

Poprawnie skonfigurowane Środowisko Uczenia musi zawierać następujące elementy:

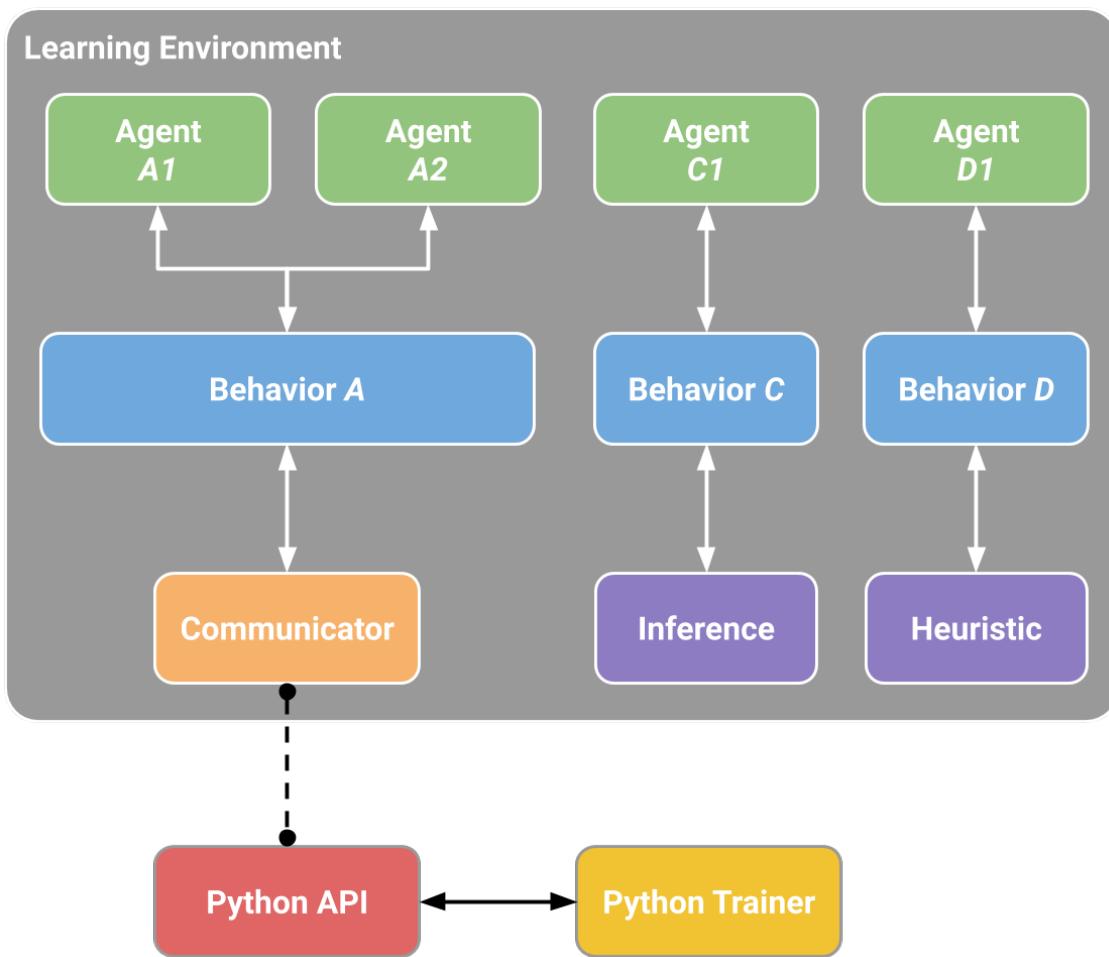
1. Co najmniej jednego Agenta (z ang. *Agent*). Agent to komponent dołączany do obiektu klasy `GameObject`, umiejscowionego na scenie Unity. Każdy Agent jest instancją klasy dziedziczącej po klasie `Agent`. Agent zajmuje się generowaniem Obserwacji, wykonywaniem Akcji oraz przydzielaniem Nagród.
2. Co najmniej jednego Zachowania (z ang. *Behavior*). Zachowanie definiuje określone atrybuty Agenta, takie jak liczba i rodzaj wykonywanych Akcji. Każde Zachowanie jest jednoznacznie identyfikowane poprzez pole `Behavior Name`.

Istnieją trzy rodzaje Zachowań:

- Zachowanie Uczęce (z ang. *Learning Behavior*) - Zachowanie, którego Agent musi się wyuczyć. Tego Zachowania używamy podczas treningu sieci neuronowych;
- Zachowanie Heurystyczne (z ang. *Heuristic Behavior*) - Zachowanie zdefiniowane przez reguły zapisane bezpośrednio w kodzie źródłowym. Przydatne m.in.

przy testowaniu nowego Środowiska Uczenia przed rozpoczęciem treningu sieci.

- Zachowanie Wnioskujące (z ang. *Inference Behavior*) - Zachowanie wykorzystujące wytrenowany model sieci neuronowej. W gruncie rzeczy, Zachowanie Uczące po skończonym treningu sieci neuronowej zmienia się w Zachowanie Wnioskujące.



Rysunek 3.3: Przykład relacji Agentów z Zachowaniami w Środowisku Uczenia

Źródło: https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/ML-Agents-Overview.md

Każdy Agent musi być powiązany z dokładnie jednym Zachowaniem, a Zachowanie może być połączone z więcej niż jednym Agentem. Agenci działający wedle tej samej logiki powinni być powiązani z tym samym Zachowaniem. To wcale nie oznacza, że wszyscy Agenci powiązani z tym samym Zachowaniem muszą współdzielić ze sobą te same Obserwacje, Akcje i Nagrody.

Rysunek 3.3 przedstawia przykładowy diagram zależności Agentów z Zachowaniami w Środowisku Uczenia. Agenci A1 i A2 są podłączeni do Zachowania Uczęcego. Agenci C1 i D1 są podłączeni odpowiednio do Zachowania Wnioskującego i Heurystycznego.

Opisując narzędzie Unity ML-Agents, należałoby również wspomnieć kilka słów o samym Agencie. Każdy Agent wykonuje cyklicznie trzy czynności: generuje Obserwacje, wykonuje zlecone Akcje i oblicza wartość Nagrody. Omówmy każdy z tych elementów:

1. Obserwacje (z ang. *Observations*) - zestaw danych wejściowych, niezbędnych do wykonania zleconego Agentowi zadania. Obserwacje mogą być generowane na kilka sposobów:
 - Nadpisanie metody `Agent.CollectObservations()` i przesłanie obserwacji do dostarczonego (przez parametr metody) obiektu klasy `VectorSensor`. Ten sposób najlepiej wykorzystać do aspektów środowiska, które można opisać numerycznie i niewizualnie.
 - Dodanie atrybutu `[Observable]` do pól i właściwości Agenta. Atrybut `[Observable]` wspiera obecnie podstawowe typy danych (takie jak `int`, `float` czy `bool`), jak również klasy `Vector2`, `Vector3`, `Vector4`, `Quaternion` oraz typy enumeracyjne.
 - Implementacja interfejsu `ISensor`, korzystając z `SensorComponent` dołączanego do Agenta. Obecnie istnieje kilka implementacji komponentu `SensorComponent` dostarczanych przez API zestawu Unity ML-Agents. Najważniejsze z nich to:
 - `CameraSensorComponent` - używa obrazów z kamery jako obserwacji;
 - `RenderTextureSensorComponent` - używa zawartości `RenderTexture` [66] jako obserwacji;
 - `RayPerceptionSensorComponent` - używa informacji z zestawu promieni (z ang. *ray casts*) jako obserwacji.
2. Akcje (z ang. *Actions*) - instrukcje zachowań, które Agent powinien wykonać. Istnieją dwa typy Akcji - Ciągłe (z ang. *Continuous*) oraz Nieciągłe (z ang. *Discrete*). Wszystkie Akcje muszą być zdefiniowane w metodzie `OnActionReceived()`. Akcje Ciągłe to wartości ze zbioru liczb rzeczywistych, podczas gdy Akcje Nieciągłe są wartościami całkowitymi.
3. Nagrody (z ang. *Rewards*) - informacja zwrotna, pozwalająca ocenić jak dobre (lub złe) są Akcje wykonywane obecnie przez Agenta. System Nagród jest najważniejszym elementem w uczeniu ze wzmacnieniem, ponieważ sposób jego skonstruowania ma decydujący wpływ na to, czy trening sieci neuronowej się powiedzie.

3.2.4 Vehicle Physics Pro

Zaawansowany zestaw do symulacji pojazdów dla silnika Unity 3D. VPP zapewnia wydajny, w pełni realistyczny i dokładny model dynamiki dla niemal każdego typu pojazdu i konfiguracji. W skład zestawu wchodzą m.in. [30]:

1. Wierne odwzorowanie pracy podzespołów pojazdu, obejmujące m.in.:
 - pracę silnika (spalinowego lub elektrycznego),
 - pracę skrzyni biegów (manualnej lub automatycznej),
 - pracę układu kierowniczego;
2. Symulacja systemów wspomagania jazdy, w tym:
 - ABS (Anti-lock Braking System),
 - TCS (Traction Control System),
 - ESC (Electronic Stability Control);
3. Liczne rozszerzenia i dodatki, takie jak:
 - efekty wizualne (np. animacja deski rozdzielczej),
 - efekty dźwiękowe (np. dźwięk silnika),
 - zaawansowana diagnostyka pojazdu, obejmująca m.in. pomiary telemetryczne,
 - system nagrywania, odtwarzania i zapisu powtórek wideo;
4. System zniszczeń wizualnych i mechanicznych;
5. Obsługa szerokiej gamy urządzeń wejściowych, takich jak gamepady czy kierownice do gier wyścigowych;
6. Szczegółowa dokumentacja.

Większość aspektów symulacji podlega możliwościom dostosowania do potrzeb użytkownika.

3.2.5 Conda

System zarządzania pakietami i środowiskami, posiadający wsparcie dla wielu języków programowania, m.in.: Python, Scala, Java, Fortran [5]. Conda pozwala łatwo tworzyć, zapisywać i ładować środowiska uruchomieniowe oraz sprawnie przełączać się pomiędzy nimi. Wykorzystując zaledwie kilka prostych poleceń, użytkownik jest w stanie skonfigurować całkowicie odrębne środowisko do uruchomienia wybranej wersji Pythona z określonymi pakietami dodatkowymi. Każde środowisko może posiadać całkowicie odmienną konfigurację.

3.2.6 Netron

Przeglądarka sieci neuronowych [54]. Netron posiada wsparcie dla wielu popularnych technologii wykorzystywanych w uczeniu maszynowym, takich jak ONNX, Keras, czy Baracuda. Ten program został wykorzystany do wizualizacji modelu wytrenowanej sieci neuronowej, otrzymanej w wyniku przeprowadzenia serii eksperymentów obliczeniowych.

3.2.7 Wykorzystane wersje oprogramowania

1. Unity - 2020.3.28f
2. C# - Mono 6.12.0.122
3. Unity ML-Agents - 0.28.0
4. Vehicle Physics Pro - 9.2
5. Conda - 4.9.2
6. Netron - 5.7.6

3.2.8 Specyfikacja techniczna stacji roboczej

Oto specyfikacja techniczna stacji roboczej (komputera), na którym została wykonana implementacja systemu oraz zostały przeprowadzone wszystkie eksperymenty obliczeniowe:

1. Typ urządzenia - Laptop
2. Marka i model urządzenia - Dell Inspiron 7559 [38]
3. Procesor - Intel® Core™ i7-6700HQ (2.6 GHz) [16]
4. Pamięć RAM - SODIMM DDR3 Synchronous 1600 MHz (32 GB)
5. Karty graficzne:
 - nVidia GeForce GTX 960M [23]
 - Intel HD Graphics 530
6. Dysk - PLEXTOR PX-512M7 (SSD 512 GB)
7. System operacyjny - Linux Mint 19.1 Tessa

Rozdział 4

Opis implementacji

System stworzony na potrzebę realizacji niniejszej pracy magisterskiej można podzielić na dwa moduły, zgodnie z opisem architektury zamieszczonym w sekcji 3.1.3:

1. Środowisko Uczenia - stworzone w silniku Unity i odpowiedzialne za symulowanie wirtualnego środowiska, po którym przemieszcza się inteligentny agent.
2. Narzędzie `mlagents_learn` - wchodzi w skład zestawu *Unity ML-Agents* (por. 3.2.3) i odpowiada za trening sieci konwolucyjnych. Jest procesem niezależnym od silnika Unity, a komunikacja ze Środowiskiem Uczenia odbywa się poprzez specjalny mechanizm, którego implementacja pochodzi z tego samego zestawu narzędziowego.

W dalszej części rozdziału znajduje się szczegółowy opis każdego z modułów.

4.1 Środowisko Uczenia

Jest typowym projektem stworzonym na silniku Unity, a w jego skład wchodzi wiele katalogów i plików. W tym rozdziale autor pracy skupił uwagę na opisaniu jedynie najważniejszych elementów potrzebnych do zrozumienia aplikacji, zakładając że resztę informacji czytelnik pracy może pozyskać z dokumentacji silnika, stanowiącej bardzo dobre źródło wiedzy o tym narzędziu.

4.1.1 Wykorzystane assety

Assety to komponenty przeznaczone do wykorzystania w projektach uruchamianych na silniku Unity. Assety mogą być zasobami różnych typów, począwszy od modeli 3D i plików audio, a skończywszy na skryptach języka C#. W ramach silnika Unity udostępniana jest specjalna usługa o nazwie *Unity Asset Store* [62], która zapewnia dostęp do darmowych i płatnych assetów.

W realizowanym projekcie jest wykorzystywanych kilka paczek assetów, lecz najważniejsze z nich są dwa: **Environmental Race Track Pack** oraz **Vehicle Physics Pro**, którego dokładny opis zamieściłem w sekcji 3.2.4.

Environmental Race Track Pack

Darmowa paczka, w skład której wchodzą cztery odmienne tory wyścigowe złożone z relatywnie małej liczby trójkątów [29]:

- „Coastal Race Track” - tor nadbrzeżny,
- „F1 Race Track” - tor Formuły 1,
- „Racing Oval” - tor w kształcie owalnym,
- „Figure 8 Track” - tor w kształcie ósemki.

Na potrzeby implementacji systemu wykorzystano dwa tory z tej paczki, co dokładniej jest opisane w sekcji 4.1.2.

Vehicle Physics Pro

Dokładny opis tego zestawu został zamieszczony w sekcji 3.2.4, tutaj natomiast należy wspomnieć o najważniejszym z wykorzystanych assetów, czyli samochodzie **Sport Coupe**. Jest to dokładnie odwzorowany model 3D samochodu sportowego, posiadający skonfigurowaną fizykę jazdy. **Sport Coupe** jest jednym z dwóch skonfigurowanych modeli samochodów, dołączanych do zestawu Vehicle Physics Pro. Drugim z nich jest **JPickup**, czyli model samochodu typu pickup. Obydwa modele zostały zaprezentowane na rysunku 4.1.

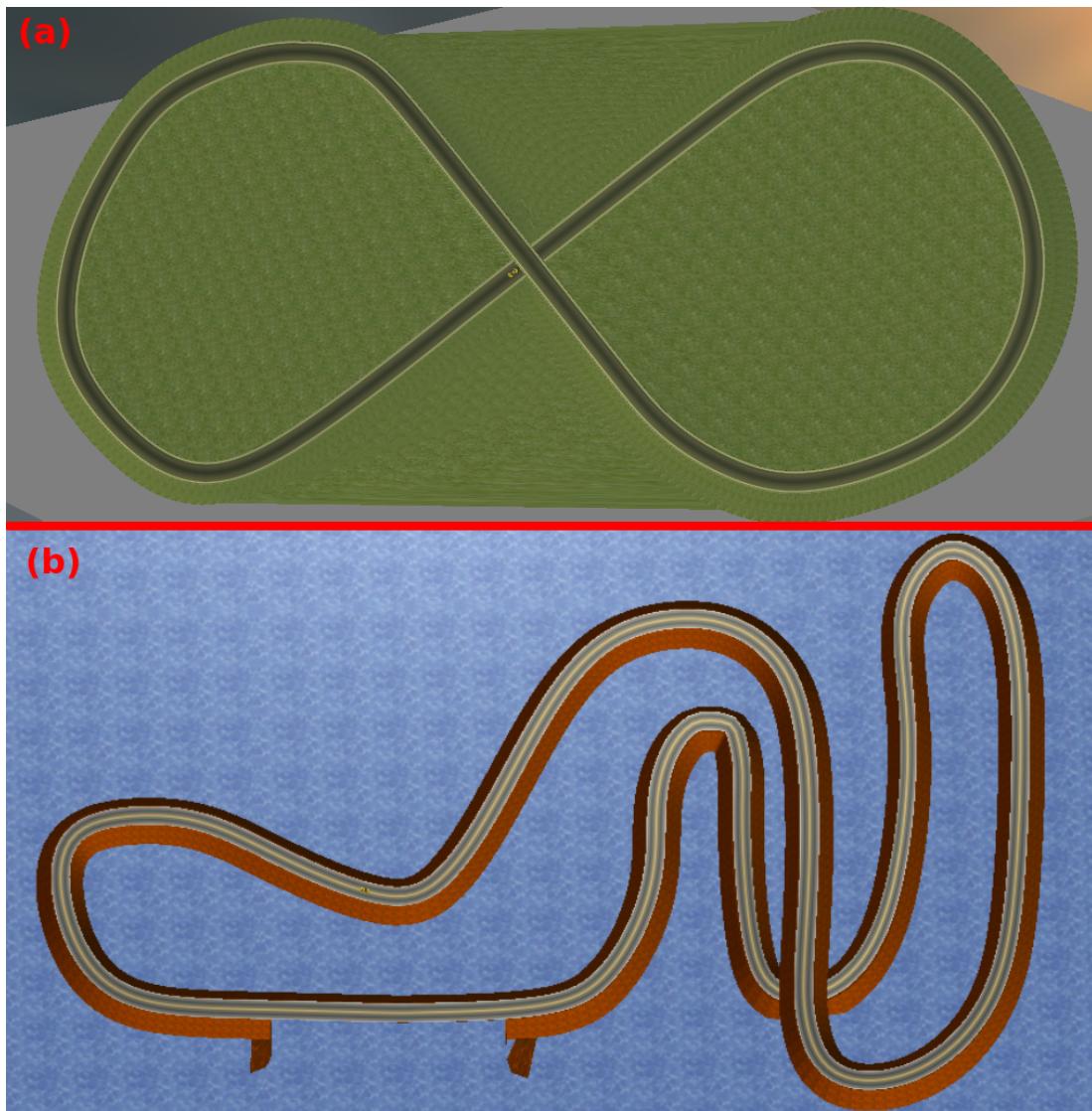


Rysunek 4.1: Modele samochodów dostarczane w pakiecie Vehicle Physics Pro

Źródło: <https://assetstore.unity.com/packages/tools/physics/vehicle-physics-pro-community-edition-153556>

4.1.2 Tory wyścigowe

Na potrzeby implementacji systemu wykorzystano dwa tory z pakietu **Environmental Race Track Pack**: „Figure 8 Track” oraz „Coastal Race Track”. Tory uległy delikatnym modyfikacjom, polegającym przede wszystkim na usunięciu niepotrzebnych elementów sceny oraz zmianie jej oświetlenia. Widok torów z lotu ptaka został uwieczniony na rysunku 4.2.



Rysunek 4.2: Tory wyścigowe wykorzystane w projekcie.

- (a) Tor nr 1 - zmodyfikowany „Figure 8 Track”
- (b) Tor nr 2 - zmodyfikowany „Coastal Race Track”

Tory różnią się od siebie przede wszystkim poziomem trudności - tor nr 2 wymaga od kierowcy znacznie większych umiejętności, ponieważ zakręty są bardziej zróżnicowane.

4.1.3 Konfiguracja sceny Unity

Na potrzeby eksperymentów obliczeniowych przygotowano kilka scen Unity. Zasadniczo różnią się one detalami, które dokładniej zostaną przedstawione w rozdziale 5-tym, natomiast ich rdzeń jest wszędzie taki sam. Najważniejszym obiektem na scenie jest CarAgent, reprezentujący model samochodu sterowany przez sieć neuronową. Obiekt składa się z kiku komponentów, wśród których najważniejsze to:

1. **Vehicle Controller** - komponent importowany z zestawu Vehicle Physics Pro (por. 3.2.4). Odpowiada za ustawienia fizyki kluczowych elementów samochodu, takich jak: układ kierowniczy, hamulce, opony i skrzynia biegów [31]. Na potrzeby implementacji systemu wprowadzono następujące zmiany w konfiguracji komponentu:
 - Zmiana skrzyni biegów na automatyczną oraz wyzerowanie wartości parametrów „*Gear transition time*” oraz „*Shift interval*”;
 - Wyłączenie wspomagania kierownicy i systemów wspomagania jazdy: kontroli trakcji (TCS), kontroli stabilności (ESC) oraz zapobiegania nadmiernemu uśiłkowi kół (ASR).
2. **Behavior Parameters** - komponent importowany z zestawu Unity ML-Agents (por. 3.2.3). Jego celem jest określenie wartości dla kluczowych parametrów trenowanego zachowania, takich jak rozmiar wektora obserwacji oraz rozmiar wektora akcji. Wartości te ulegały zmianie dla poszczególnych eksperymentów obliczeniowych, dlatego ten aspekt implementacji zostanie dokładniej opisany w kolejnym rozdziale.
3. **Klasa dziedzicząca po klasie Agent** - skrypt C#, w którym zawarto implementację klasy dziedziczącej po klasie Agent, wywodzącej się z zestawu Unity ML-Agents. Na potrzeby eksperymentów obliczeniowych przygotowano kilka wersji implementacji tego skryptu, jednak rdzeń odpowiedzialności pozostaje taki sam. Skrypt jest odpowiedzialny za: prawidłową inicjalizację każdego epizodu treningu, przekazywanie obserwacji ze Środowiska Uczzenia do sieci neuronowej, przekazywanie wyjścia z sieci neuronowej do Środowiska Uczzenia oraz obliczanie wartości sygnałów nagrody dla danego kroku symulacji. Oprócz tego, każda wersja implementacji posiada własne, dodatkowe odpowiedzialności.
4. **Decision Requester** - komponent importowany z zestawu Unity ML-Agents. Jego głównym zadaniem jest określenie częstotliwości podejmowania decyzji przez Agenta.
5. **Camera Sensor** - czujnik kamery, importowany z zestawu Unity ML-Agents. Jest niezbędny, jeśli chcemy dostarczać sieci neuronowej obserwacji wizualnych ze środowiska. Parametrami komponentu są m.in. wymiary obrazu oraz typ kompresji.

4.2 Narzędzie mlagents_learn

Główne narzędzie do treningu sieci neuronowych, oferowane przez zestaw Unity ML-Agents [65]. Parametry są przekazywane do narzędzia poprzez wiersz poleceń oraz plik konfiguracyjny utrzymywany w formacie YAML [64]. Zestaw ustawień oraz hiperparametrów zdefiniowanych w pliku konfiguracyjnym zależy w ogromnym stopniu od zastosowanej metody treningowej oraz sposobu konfiguracji Środowiska Uczenia. Konkretna zawartość wykorzystanych plików konfiguracyjnych zostanie zaprezentowana w kolejnym rozdziale.

Wywołanie narzędzia skutkuje uruchomieniem procesu treningowego oraz rozpoczęciem uczenia sieci. W wyniku swojej pracy, proces treningowy generuje trzy artefakty:

1. **Podsumowania** - metryki treningowe, aktualizowane podczas trwania treningu. Są bardzo pomocne przy monitorowaniu wyników uczenia sieci oraz ewentualnych aktualizacjach wartości hiperparametrów. Podsumowania można wyświetlić w eleganckiej formie graficznej, wykorzystując w tym celu narzędzie *TensorBoard* [70].
2. **Modele** - katalog z punktami kontrolnymi modelu aktualizowanego podczas szkolenia, a także ostateczny plik modelu. Plik ostateczny jest generowany jednorazowo, czyli po zakończeniu lub przerwaniu uczenia. Wszystkie wygenerowane modele są plikami w formacie ONNX [28].
3. **Pliki timerów** - zawierają zagregowane metryki procesu uczenia, w tym czas spędziły na określonych blokach kodu. Są bardzo pomocne podczas prac nad poprawą wydajności obliczeń procesu uczenia.

Trening sieci

Podczas wszystkich eksperymentów obliczeniowych wykorzystywałem do treningu sieci metodę uczenia maszynowego o nazwie **uczenie ze wzmacnieniem** (z ang. *reinforcement learning*). Zestaw narzędziowy Unity ML-Agents dostarcza implementacji dla dwóch algorytmów wspierających tę metodę - PPO [56] oraz SAC [33]. Po kilku próbach okazało się, że do realizacji tematu pracy znacznie lepiej sprawdzi się algorytm PPO, dlatego właśnie on został wykorzystany we wszystkich eksperymentach obliczeniowych opisanych w rozdziale 5-tym. PPO jest wyjątkowe pośród innych algorytmów uczenia ze wzmacnieniem, ponieważ zachowuje balans pomiędzy łatwością implementacji, złożonością próbki oraz łatwością dostrajania. Z tego też powodu jest oferowany jako domyślny algorytm uczenia ze wzmacnieniem dla zestawu Unity ML-Agents.

Rozdział 5

Eksperymenty obliczeniowe

W tym rozdziale zostanie zaprezentowana część praktyczna tworzonej pracy, która polega na przeprowadzeniu kilku eksperymentów obliczeniowych w oparciu o przygotowane przez autora oprogramowanie. Rozdział rozpoczyna się od omówienia przyjętej metodyki eksperymentów, następnie zostaną przedstawione najistotniejsze szczegóły implementacyjne poszczególnych eksperymentów obliczeniowych. Rozdział zakończy się analizą uzyskanych wyników oraz opisem wypływających z tego wniosków.

5.1 Metodyka eksperymentów

Metodyka eksperymentów obliczeniowych polega na sformułowaniu listy założeń, na których ma się oprzeć przeprowadzenie wszystkich eksperymentów. Oto najważniejsze z przyjętych założeń:

1. Każdy eksperiment obliczeniowy polega na wytrenowaniu konwolucyjnej sieci neuronowej w przygotowanym Środowisku Uczenia oraz ewaluacji wyuczonego modelu.
2. Parametry dla każdego eksperymentu powinny zostać dostarczone w postaci zewnętrznego pliku konfiguracyjnego o zdefiniowanym formacie. Ścieżkę do pliku konfiguracyjnego należy podać jako argument linii poleceń.
3. Wynikiem przeprowadzonego treningu sieci konwolucyjnej powinien być plik z wytrenowanym modelem.
4. Do samodzielnego przeprowadzenia eksperymentów obliczeniowych należy mieć zainstalowane odpowiednie narzędzia (zgodnie z opisem z rozdziału 3-ciego), jak również posiadać kod źródłowy aplikacji wymaganej do wykonania eksperymentów.
5. Wytrenowany model sieci neuronowej powinien posiadać format danych pozwalający na jego wizualizację przy użyciu zewnętrznego oprogramowania.

5.2 Opis implementacji

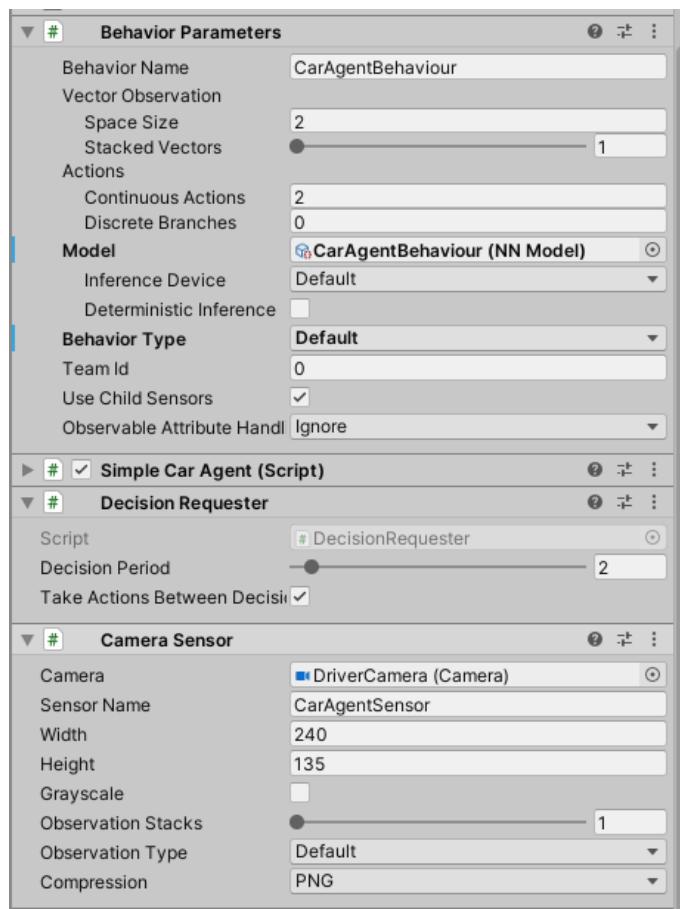
Implementacja systemu została częściowo opisana w rozdziale 4-tym, natomiast każdy z przeprowadzonych eksperymentów wymagał dostosowania pewnych detali w implementacji. Poniżej zamieszczam opis tych elementów implementacji, które nie zostały jeszcze opisane, a są zbyt ważne aby je pominąć.

5.2.1 Środowiska Uczenia

W ramach prac nad częścią eksperymentalną zostały przygotowane 3 Środowiska Uczenia, będące niczym innym jak scenami silnika Unity. Każde Środowisko Uczenia posiada własną charakterystykę oraz pewne cechy szczególne, wyróżniające je na tle pozostałych Środowisk.

RaceTrack_1

Te Środowisko Uczenia zostało oparte na torze wyścigowym „Figure 8 Track” z pakietu **Environmental Race Track Pack** (por. 4.1.2). Najważniejszą cechą Środowiska jest brak oświetlenia, co zostało dołożone w kolejnych Środowiskach. Wykorzystaną tutaj implementacją klasy Agent jest *SimpleCarAgent* (por. 5.2.2). Rysunek 5.1 przedstawia najważniejsze komponenty obiektu sceny *CarAgent*.



Rysunek 5.1: Najważniejsze komponenty obiektu *CarAgent* w Środowisku Uczenia *RaceTrack_1*

Na podstawie rysunku można wysnuć kilka wniosków:

- Do agenta są dostarczane dwie obserwacje wektorowe;
- Agent jest zobligowany do wykonywania dwóch akcji dla każdego kroku symulacji;
- Wykorzystaną implementacją klasy *Agent* jest klasa *SimpleCarAgent*;
- Decyzje są podejmowane dla co drugiego kroku symulacji. Pomiędzy decyzjami należy wykonywać ostatnio obliczone Akcje;
- Wykorzystywane są obserwacje wizualne w postaci pojedynczego czujnika, zbierającego obraz z kamery umieszczonej na miejscu fotela kierowcy. Obserwacje wizualne mają rozdzielcość 240×135 pikseli. Rysunek 5.2 przedstawia widok z kamery stanowiącej źródło danych wejściowych dla obserwacji wizualnych.



Rysunek 5.2: Widok z fotela kierowcy w Środowisku Uczenia *RaceTrack_1*

RaceTrack_2

Jest to Środowisko *RaceTrack_1* z dodanym oświetleniem kierunkowym, którego parametry są losowo ustawiane przed rozpoczęciem każdego epizodu symulacji. Parametry oświetlenia ulegające zmianie to **pozycja**, **rotacja** (orientacja), **kolor** oraz **intensywność**.

RaceTrack_3

Środowisko Uczenia oparte na torze wyścigowym „*Coastal Race Track*” z pakietu **Environmental Race Track Pack** (por. 4.1.2). Ponieważ tor wyścigowy jest bardziej skomplikowany, przygotowana została nowa implementacja klasy *Agent* - klasa *AdvancedCarAgent*.

Reszta komponentów obiektu sceny *CarAgent* pozostała bez zmian. Rysunek 5.3 przedstawia widok z kamery dla tego Środowiska Uczenia.



Rysunek 5.3: Widok z fotela kierowcy w Środowisku Uczenia *RaceTrack_3*

5.2.2 Implementacje klasy Agent

W toku prac nad aplikacją zostało napisanych i przetestowanych wiele wersji implementacji klasy *Agent* z zestawu Unity ML-Agents, jednak ostatecznie do wykorzystania zostały wyznaczone dwie klasy: *SimpleCarAgent* oraz *AdvancedCarAgent*. Choć są do siebie podobne, to jednak istnieje kilka różnic na które koniecznie trzeba zwrócić uwagę.

SimpleCarAgent

Klasa użyta w Środowiskach Uczenia *RaceTrack_1* oraz *RaceTrack_2* (por. 5.2.1). Jest odpowiedzialna za:

1. *Prawidłową inicjalizację każdego epizodu treningu* - przywrócenie samochodu do pozycji startowej oraz zmianę właściwości oświetlenia (jego pozycji, orientacji, koloru oraz intensywności).
2. *Przekazywanie obserwacji ze Środowiska Uczenia do sieci neuronowej* - pobierane są 3 obserwacje ze środowiska - jedna wizualna (obraz z kamery) oraz dwie wektorowe (znormalizowana prędkość samochodu oraz informacja o tym, czy w danej chwili samochód dotyka jakiejś przeszkody).

3. *Przekazywanie wyjścia z sieci neuronowej do Środowiska Uczenia* - sieć neuronowa generuje dwie wartości (zwane Akcjami), które są liczbami rzeczywistymi z domkniętego przedziału od -1 do 1 . Pierwsza wartość to stopień wciśnięcia pedału hamulca lub przepustnicy (gdzie -1 to maksymalnie wciśnięty hamulec, a 1 to maksymalnie wciśnięta przepustnica), a druga to kąt skrętu kierownicy (gdzie 0 oznacza jazdę na wprost, -1 maksymalny skręt w lewo a 1 maksymalny skręt w prawo).
4. *Obliczanie wartości sygnałów nagrody dla danego kroku symulacji* - sygnały nagród są obliczane na podstawie dwóch przesłanek: prędkości samochodu oraz kolizji z przeszkodeami. Nagroda za prędkość jest proporcjonalna do prędkości samochodu, co oznacza że większa prędkość oznacza większą nagrodę. Za kolizje z przeszkodeami przyznawane są kary, proporcjonalne do prędkości z jaką doszło do kolizji z przeszkodeą. Dodatkowo przyznawana jest kara o stałej wartości za każdy krok symulacji, w którym samochód styka się z jakąś przeszkodeą.

Zmienne publiczne, jakie można przypisać do tej klasy z poziomu edytora Unity, to:

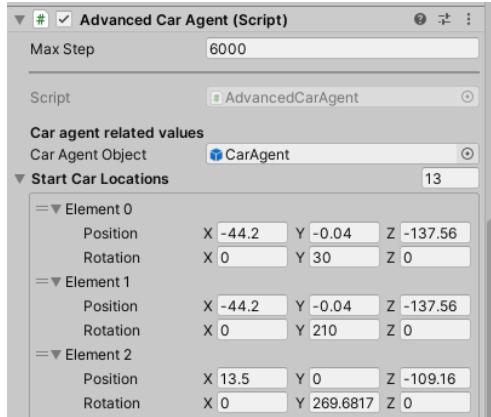
- `MaxStep` - maksymalna liczba kroków symulacji dla danego epizodu;
- `CarAgentObject` - Referencja na obiekt sceny `CarAgent`;
- `StartCarPosition` - Pozycja początkowa samochodu;
- `StartCarRotation` - Orientacja początkowa samochodu;
- `DirectionalLight` - Referencja na obiekt oświetlenia (może być pusta, jeśli oświetlenie nie jest używane).

AdvancedCarAgent

Klasa użyta w Środowisku Uczenia `RaceTrack_3` (por. 5.2.1). Posiada dokładnie te same odpowiedzialności co klasa `SimpleCarAgent`, jak również większość kodu jest taka sama. Główna różnica polega na tym, że klasa `AdvancedCarAgent` wspiera możliwość resetowania samochodu przy starcie nowego epizodu do wielu różnych pozycji oraz orientacji. Listę pozycji i orientacji można uzupełniać z poziomu edytora Unity, co widać na rysunku 5.4, gdzie został zaprezentowany fragment listy `StartCarLocations`.

5.2.3 Konfiguracja treningu

Konfiguracja treningu odbywa się poprzez przygotowanie pliku konfiguracyjnego w formacie YAML [64]. Dla każdego Środowiska Uczenia został przygotowany osobny plik konfiguracyjny, jednak różnice pomiędzy nimi są kosmetyczne i dotyczą detali. Oto zawartość pliku konfiguracyjnego dla Środowiska `RaceTrack_3`:



Rysunek 5.4: *AdvancedCarAgent* - fragment listy StartCarLocations.

```

engine_settings:
  width: 240
  height: 135
  quality_level: 5
  time_scale: 20
  target_frame_rate: -1
  capture_frame_rate: 60
  no_graphics: false
torch_settings:
  device: cuda
behaviors:
  CarAgentBehaviour:
    trainer_type: ppo
    summary_freq: 50000
    time_horizon: 256
    max_steps: 30000000
    keep_checkpoints: 10
    checkpoint_interval: 500000
    threaded: true
    network_settings:
      hidden_units: 128
      num_layers: 2
      normalize: true
      vis_encode_type: simple
      conditioning_type: none
    reward_signals:
      extrinsic:
        strength: 1.0
        gamma: 0.99

```

Najważniejsze ustawienia dotyczą sekcji behaviors, ponieważ one mają największy wpływ na kształt treningu. W tym miejscu warto omówić kilka z nich:

1. **trainer_type** - wybór algorytmu uczącego. Dla wszystkich eksperymentów obliczeniowych wybrano algorytm PPO [56], ponieważ w toku prac nad aplikacją okazał się

- być lepszym wyborem niż SAC [33];
2. `time_horizon` - jak wiele kroków symulacji należy zebrać przed dodaniem ich do „bufora doświadczenia”. Wartość parametru powinna znajdować się w kompromisie pomiędzy mniej stronniczym, ale wyższym oszacowaniem wariancji (długi horyzont czasowy) i bardziej stronniczym, ale mniej zróżnicowanym oszacowaniem (krótki horyzont czasowy). Gdy nagrody są często przyznawane lub epizody trwają dość długo, wtedy mniejsza liczba może być lepszym wyborem. Niemniej jednak, wartość ta powinna być na tyle duża, aby można było uchwycić wszystkie ważne zachowania w sekwencji działań agenta;
 3. `max_steps` - maksymalna liczba kroków symulacji przed zakończeniem treningu;
 4. `keep_checkpoints` - maksymalna liczba modeli sieci neuronowych, pozostawionych do zapisu. Modele te są zapisywane w odstępach określonych wartością parametru `checkpoint_interval`, która oznacza interwał czasowy wyrażony w krokach symulacji treningowej;
 5. `network_settings` - ustawienia sieci neuronowej. Najważniejsze z nich to:
 - `hidden_units` - liczba neuronów w pełni połączonej warstwie ukrytej;
 - `num_layers` - liczba warstw ukrytych;
 - `vis_encode_type` - typ enkodera dla obserwacji wizualnych.

Rozpoczęcie treningu sieci odbywa się poprzez wywołanie komendy `mlagents_learn`, której dokładne wykorzystanie zostało opisane w dokumentacji zestawu Unity ML-Agents [65].

5.3 Przebieg eksperymentów i analiza wyników

Dla każdego z przygotowanych Środowisk Uczenia przeprowadzono dokładnie jeden eksperyment obliczeniowy, polegający na wytrenowaniu konwolucyjnej sieci neuronowej do jazdy samochodem po wybranym torze wyścigowym, a następnie ewaluacji wytrenowanego modelu pod kątem przystosowania do rzeczywistej funkcji celu, jaką jest pokonanie toru wyścigowego w możliwie najkrótszym czasie.

5.3.1 RaceTrack_1

Środowisko najłatwiejsze do wytrenowania konwolucyjnej sieci neuronowej, ponieważ brak jest zmennego oświetlenia sceny, a zakręty są łagodne i niewymagające trudnych manewrów do wykonania. Wytrenowanie modelu zajęło 1 647 798 kroków symulacji.

Obserwacje sieci obejmowały:

1. Obserwacje wizualne z kamery umieszczonej na fotelu kierowcy, czyli obrazki RGB o wymiarach 240×135 pikseli;
2. Obserwacje wektorowe:
 - Bieżąca, znormalizowana prędkość samochodu;
 - Wartość logiczna informująca, czy samochód w danej chwili styka się z jakąś przeszkodą.

Model był trenowany na pojedynczej instancji obiektu sceny *CarAgent*, którego pozycja i orientacja były resetowane na początku każdego epizodu symulacji. Epizod kończył się w momencie przejechania pełnego okrążenia lub po wyczerpaniu zakładanego limitu czasowego dla danego epizodu. Z testów wytrenowanego modelu wynika, że:

1. Model charakteryzuje się niestabilnym tempem jazdy, a wykonane pomiary czasowe wskazują na dość znaczne odchyły w czasach okrążień.
2. Wytrenowany model jest w stanie bezpiecznie przejechać kilka okrążień pod rząd, choć ma tendencje do bardzo agresywnego wchodzenia w pierwszy zakręt, co czasem kończy się poważną kraksą - samochód obraca się o 90 stopni w stosunku do zakładanego kierunku jazdy.
3. Po wystąpieniu poważnej kraksy, wytrenowany model nie potrafi zazwyczaj wrócić do dalszej jazdy, lecz stara się zminimalizować straty poprzez oddalenie się od barierki, tak aby nie było naliczanej kary za kontakt z przeszkodami.
4. Model radzi sobie z jazdą tylko wtedy, gdy kolor drogi wyraźnie kontrastuje z poboczem - przy załączonym oświetleniu kierunkowym sieć się gubi, dojeżdżając do miejsca gdzie droga zbyt się rozjaśnia z powodu oświetlenia.

5.3.2 RaceTrack_2

Środowisko Uczenia bardzo podobnie skonfigurowane do *RaceTrack_1*, jednak z jedną zasadniczą różnicą - trening odbywał się przy załączonym oświetleniu kierunkowym, którego właściwości były losowo ustawiane dla każdego epizodu treningowego. Model został wytrenowany po 4 654 498 krokach symulacji. Obserwacje modelu wykazały, że:

1. Jego tempo jazdy jest znacznie stabilniejsze od poprzedniego modelu, ponieważ różnice między najszybszym i naj wolniejszym okrążeniem były średnio 4-krotnie mniejsze niż dla poprzedniego modelu.
2. Model wciąż wykazuje tendencje do agresywnego wchodzenia w pierwszy zakręt, co często kończy się poważną kraksą.

3. Po wystąpieniu poważnej kraksy (obrótanie o 90 stopni lub więcej), samochód staje w miejscu lub wręcz zatrzymuje się i zaczyna jechać „pod prąd”. Pod tym względem zachowanie tego modelu niezbyt odbiega od zachowania modelu poprzedniego.
4. Model dobrze sobie radzi ze zmiennym oświetleniem, chociaż wciąż się gubi dla niektórych przypadków brzegowych.

5.3.3 RaceTrack_3

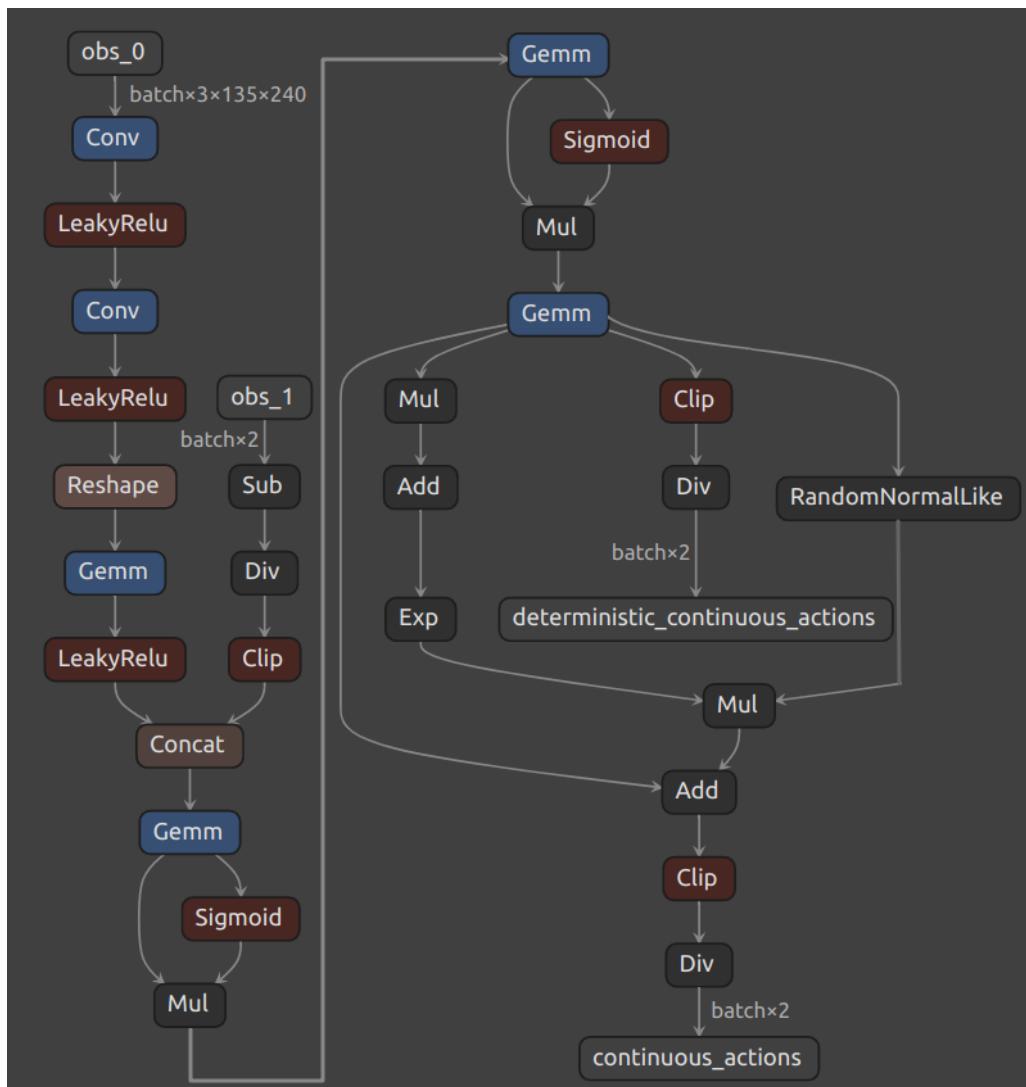
W tym Środowisku Uczenia trening potrwało znacznie dłużej niż na obydwu poprzednich Środowiskach razem wziętych i wyniosło 12 999 786 milionów kroków symulacji. Wynika to z faktu, że te Środowisko jest dużo bardziej trudne do wytrenowania, ponieważ tor wyścigowy jest bardziej zróżnicowany i składa się z zakrętów o bardziej odmiennej charakterystyce.

Z obserwacji wytrenowanego modelu wynika, że:

1. Jego zachowanie jest dosyć nierówne. Z jednej strony, zdarza się że przez dłuższy odcinek czasu samochód całkiem znośnie pokonuje kolejne zakręty, zwalniając przed ostrzejszymi nawrotami. Z drugiej strony, chwilę później na łagodniejszym zakręcie samochód potrafi uderzyć pełnym impetem w barierkę, obrócić się bokiem do drogi i dalej nie ruszyć.
2. Model wypracował sobie strategię utrzymania toru jazdy samochodu przy mocnym zderzeniu z barierką na ostrym zakręcie. W takich sytuacjach następuje skręt kierowcy do skrajnej pozycji w kierunku zakrętu - czyli przy zakręcie w lewo następuje maksymalny skręt kierownicą na lewo.
3. Jeśli samochód po zderzeniu z barierką nie został obrócony prostopadle do kierunku jazdy, lecz pod pewnym mniejszym kątem, to wtedy jest szansa że samochód powróci do dalszej jazdy. Jeśli jednak samochód obróci się o więcej niż 90 stopni, to nie zatrzyma się, ponieważ model nie posiada żadnych informacji na temat tego, w którą stronę samochód powinien się poruszać. W najlepszym wypadku samochód ruszy w drugą stronę.
4. Podobnie jak poprzedni model, całkiem dobrze radzi sobie ze zmiennym oświetleniem.
5. Przejeżdżając przez ciasne nawroty, samochód zwalnia aby uniknąć kolizji. W skrajnych przypadkach potrafi całkowicie się zatrzymać i dalej nie ruszyć, pomimo tego że sytuacja pozwalałyby na dalszą jazdę.

5.4 Analiza wytrenowanego modelu

Model wytrenowany w Środowisku *RaceTrack_3* został załadowany do programu o nazwie Netron (por. 3.2.6). Rysunek 5.5 przedstawia wizualizację załadowanego modelu.



Rysunek 5.5: Wizualizacja modelu wytrenowanego w Środowisku Uczenia *RaceTrack_3*

Więcej informacji na temat każdego z operatorów (czyli typów warstw sieci) można znaleźć na stronie dokumentacji standardu ONNX [1]. Wykorzystane operatory posiadają następujące znaczenie:

1. **obs_0, obs_1** - obserwacje wejściowe (odpowiednio: wizualne i wektorowe);
2. **Conv** - warstwa konwolucyjna;
3. **LeakyRelu** - funkcja aktywacji LeakyRelu;



Rysunek 5.6: Podgląd wytrenowanego modelu w inspektorze edytora Unity

4. **Reshape** - warstwa zmieniająca kształt (wymiary) danych wejściowych;
5. **Add, Sub, Mul, Div** - operacje dodawania, odejmowania, mnożenia i dzielenia;
6. **Clip** - operacja obcinania wartości do zadanej dziedziny;
7. **Concat** - warstwa łącząca;
8. **Gemm** - warstwa ogólnego mnożenia macierzy;
9. **Sigmoid** - funkcja aktywacji Sigmoid;
10. **Exp** - operacja potęgowania;
11. **RandomNormalLike** - generator tensorów losowych.

Ten sam model można również podejrzeć w inspektorze edytora Unity, co zostało zaprezentowane na rysunku 5.6. Wytrenowany model składa się z 30 warstw oraz 1 765 298 wag,

których wartości musiały zostać dostrojone podczas treningu. Cały plik modelu ma rozmiar 7 061 240 bajtów, co w przybliżeniu daje nam 6,73 MB.

W tym miejscu należy wyjaśnić, dlaczego zastosowano powyżej opisaną architekturę modelu sieci. Powody są następujące:

1. Punktem wyjścia była chęć przeprowadzenia treningu w oparciu o dane wizualne, a do takich zadań doskonałym wyborem są konwolucyjne sieci neuronowe (por. rozdział 2-gi). Dlatego architektura sieci posiada warstwy konwolucyjne.
2. Dwie warstwy konwolucyjne są domyślnym wyborem, oferowanym przez zestaw Unity ML-Agents. Po kilku testach, autor postanowił pozostać przy tej wartości, ponieważ okazała się być wystarczająco dobra dla potrzeb treningu.
3. Autor eksperymentował z różnymi rozdzielczościami obrazu wejściowego, rozpoczętając od możliwie najmniejszych wartości. Taki kierunek poszukiwań wydawał się właściwy, ponieważ mniejsze dane wejściowe oznaczają zwykle krótszy trening, a w konsekwencji szybszą ewaluację zaprojektowanego modelu. Jednakże zbyt niska rozdzielczość to brak detali istotnych do nauczenia się modelu, dlatego wybór właściwej rozdzielczości jest zawsze kwestią kompromisu. Po wielu testach, autor ostatecznie przyjął rozdzielczość 240×135 , ponieważ przy niższej rozdzielczości ciasne nawroty byłyauważane zbyt późno i model nie miał możliwości dohamować przed zakrętem.
4. Pozostałe elementy architektury sieci pochodzą z domyślnej konfiguracji zestawu Unity ML-Agent. Konfiguracja ta okazała się sprawdzać bardzo dobrze, dlatego autor pracy postanowił przy niej pozostać.

5.5 Wnioski

W wyniku przeprowadzonych eksperymentów obliczeniowych, autor niniejszej pracy sformułował kilka wniosków:

1. Złożoność Środowiska Uczenia w bezpośredni sposób przekłada się na długość treningu sieci neuronowej. Pomimo braku zmian w topologii sieci, jej hiperparametrami czy też algorytmie uczącym, każde kolejne z opisanych Środowisk Uczenia wymagało zauważalnie więcej obliczeń podczas treningu modelu. Wynikało to z faktu, że każde kolejne Środowisko było bardziej różnorodne, co utrudniało algorytmowi uczącemu odnalezienie powiązań między wzorcami wizualnymi i oczekiwanyymi akcjami.
2. Zaprojektowana funkcja nagród (zależna od prędkości samochodu oraz występujących kolizji) okazała się być wystarczająca jedynie dla wyuczenia się podstawowych ma-

newrów, takich jak jazda po prostej oraz bezkolizyjne pokonywanie łagodnych zakrętów. Przejeżdżając przez ciaśniejsze zakręty, model wykazuje zbyt agresywne zachowanie, co zazwyczaj kończy się poważną stłuczką i brakiem możliwości dalszej jazdy. Aby to zmienić, funkcję nagród należałoby przeprojektować, co jest zadaniem wysoce nietrywialnym i wymagającym wiele czasu poświęconego na testowanie różnych rozwiązań.

3. Pomimo treningu przy zmiennym oświetleniu, rodzaj i natężenie oświetlenia wciąż mają istotny wpływ na jakość zachowania modelu.
4. Model wytrenowany na jednym Środowisku Uczenia nie potrafi się poprawnie zachować na pozostałych Środowiskach. Powódów takiego stanu rzeczy należałoby upatrywać w zauważalnych różnicach wizualnych pomiędzy Środowiskami.
5. Uzyskane wyniki eksperymentów pozwalają na zaobserwowanie korelacji pomiędzy Środowiskiem Uczenia a długością treningu, natomiast nie wnoszą żadnej wiedzy na temat wpływu konfiguracji na skuteczność wyuczonego modelu. Dalsze badania w tym zakresie są jak najbardziej wskazane.

Podsumowanie

Celem pracy było stworzenie prostego systemu uczącego konwolucyjne sieci neuronowe na bazie symulacji uruchamianych w zaprojektowanym środowisku. Zadaniem wytrenowanych modeli była jazda wirtualnym modelem samochodu po przygotowanym torze wyścigowym. Trening sieci neuronowych odbywał się przy użyciu algorytmu PPO, wykorzystującego technikę uczenia maszynowego o nazwie **Uczenie ze Wzmocnieniem** (z ang. *Reinforcement Learning*).

Cel pracy został osiągnięty w całości. Stworzona aplikacja pozwala na wytrenowanie sieci neuronowej na wybranym Środowisku Uczenia. Sieć konwolucyjna została z sukcesem wykorzystana do przetwarzania obserwacji wizualnych, pochodzących z kamery zamontowanej na fotelu kierowcy. Wytrenowane sieci neuronowe potrafią pokonywać wirtualne tory wyścigowe - nawet te o dużym stopniu skomplikowania. Więcej informacji o wytrenowanych modelach należy szukać w rozdziale 5-tym.

Perspektywy dalszych badań w dziedzinie

Czas i wysiłek włożony w napisanie niniejszej pracy pozwolił na zaledwie pobiczne omówienie podjętego tematu. Uczenie maszynowe oraz projektowanie samochodów autonomicznych to dwa bardzo obszerne zagadnienia, na które wielu wybitnych naukowców poświęciło długie lata badań.

Jednym z oczywistych kierunków dalszych badań byłoby zastosowanie zdobytej wiedzy do wytrenowania modelu sterującego samochodem poruszającym się w prawdziwym świecie. Dla ułatwienia badań, eksperymenty należałyby rozpocząć od pojazdu o mniejszych gabarytach, czyli wykonanego w pewnej skali. Kolejny pomysł to dodanie większej liczby kamer, zamontowanych dookoła samochodu. Takie rozwiązanie doprowadziłoby do zwiększenia pola widzenia i pewniejszego zachowania podczas precyzyjnych manewrów, np. w ciasnych nawrotach lub podczas parkowania.

Spis rysunków

1.1	Układ czujników systemu Waymo Driver	9
1.2	NVIDIA DRIVE AGX Pegasus	10
1.3	NVIDIA DRIVE AGX Xavier	11
1.4	Moduły oprogramowania NVIDIA DRIVE	13
1.5	Komputer Tesla FSD	17
2.1	Przykład działania filtra w warstwie konwolucyjnej	19
2.2	Przykład działania warstwy łączniowej	21
2.3	Podział zbioru danych użytych do wyuczenia sieci	23
2.4	Przykład ataku kontradiktoryjnego - żółw klasyfikowany jako karabin	28
2.5	Ta sama rzeźba na pięciu różnych ujęciach.	29
3.1	Wizualizacja architektury systemu.	31
3.2	Diagram komponentów Unity ML-Agents	34
3.3	Przykład relacji Agentów z Zachowaniami w Środowisku Uczenia	35
4.1	Modele samochodów dostarczane w pakiecie Vehicle Physics Pro	40
4.2	Tory wyścigowe wykorzystane w projekcie.	41
5.1	Najważniejsze komponenty obiektu <i>CarAgent</i> w Środowisku Uczenia <i>RaceTrack_1</i>	45
5.2	Widok z fotela kierowcy w Środowisku Uczenia <i>RaceTrack_1</i>	46
5.3	Widok z fotela kierowcy w Środowisku Uczenia <i>RaceTrack_3</i>	47

5.4	<i>AdvancedCarAgent</i> - fragment listy StartCarLocations.	49
5.5	Wizualizacja modelu wytrenowanego w Środowisku Uczenia <i>RaceTrack_3</i>	53
5.6	Podgląd wytrenowanego modelu w inspektorze edytora Unity	54

Bibliografia

- [1] ONNX documentation - Operator Schemas. <https://github.com/onnx/onnx/blob/main/docs/operators.md>. Data dostępu: czerwiec 2022.
- [2] Eric Adams. Why we're still years away from having self-driving cars. <https://www.vox.com/recode/2020/9/25/21456421/why-self-driving-cars-autonomous-still-years-away>, 2020. Data dostępu: maj 2022.
- [3] Thomas Ako. The Best Tips and Guide on Lidar Eye Safety. <https://photronicsreport.com/blog/is-lidar-dangerous-for-our-eyes/>, 2021. Data dostępu: maj 2022.
- [4] Nefi Alarcon. DRIVE Labs: How Localization Helps Vehicles Find Their Way. <https://developer.nvidia.com/blog/drive-labs-how-localization-helps-vehicles-find-their-way/>, 2020. Data dostępu: maj 2022.
- [5] Anaconda. Dokumentacja systemu Conda. <https://docs.conda.io/en/latest>, 2017. Data dostępu: maj 2022.
- [6] Antonio Armenta. Safety Considerations for LiDAR Sensors. <https://control.com/technical-articles/safety-considerations-for-lidar-sensors/>, 2021. Data dostępu: maj 2022.
- [7] Anish Athalye, Logan Engstrom, Andrew Ilyas, Kevin Kwok. Fooling Neural Networks in the Physical World with 3D Adversarial Objects. <https://www.labsix.org/physical-objects-that-fool-neural-nets/>, 2017. Data dostępu: maj 2022.
- [8] Pragati Baheti. A Comprehensive Guide to Convolutional Neural Networks. <https://www.v7labs.com/blog/convolutional-neural-networks-guide>, 2022. Data dostępu: maj 2022.

- [9] Norbert Biedrzycki. Autonomiczne samochody wyprzedzają nasze systemy prawne. <https://businessinsider.com.pl/technologie/autonomiczne-pojazdy-a-przepisy-ruchu-i-prawa/e4jd3n4>, 2019. Data dostępu: maj 2022.
- [10] Yash Bohra. The Challenge of Vanishing/Exploding Gradients in Deep Neural Networks. <https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/>, 2021. Data dostępu: maj 2022.
- [11] Jason Brownlee. A Gentle Introduction to Cross-Entropy for Machine Learning. <https://machinelearningmastery.com/cross-entropy-for-machine-learning/>, 2019. Data dostępu: maj 2022.
- [12] Jason Brownlee. A Gentle Introduction to the Rectified Linear Unit (ReLU). <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>, 2019. Data dostępu: maj 2022.
- [13] Jason Brownlee. Loss and Loss Functions for Training Deep Learning Neural Networks. <https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/>, 2019. Data dostępu: maj 2022.
- [14] Matt Burns. ‘Anyone relying on lidar is doomed,’ Elon Musk says. <https://techcrunch.com/2019/04/22/anyone-relying-on-lidar-is-doomed-elon-musk-says>, 2019. Data dostępu: maj 2022.
- [15] Harsh Chauhan. Here’s Why NVIDIA Can Sideswipe Alphabet in Autonomous Cars. <https://www.fool.com/investing/2019/09/28/heres-why-nvidia-can-sideswipe-alphabet-in-autonom.aspx>, 2019. Data dostępu: maj 2022.
- [16] Intel Corporation. Specifications for Intel® Core™ i7-6700HQ Processor. <https://ark.intel.com/content/www/us/en/ark/products/88967/intel-core-i76700hq-processor-6m-cache-up-to-3-50-ghz.html>, 2015. Data dostępu: maj 2022.
- [17] NVIDIA Corporation. Hardware for Self-Driving Cars. <https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/>. Data dostępu: maj 2022.

- [18] NVIDIA Corporation. Innovations by Automotive Industry Partners. <https://www.nvidia.com/en-us/self-driving-cars/partners/>. Data dostępu: maj 2022.
- [19] NVIDIA Corporation. NVIDIA DRIVE Hyperion Autonomous Vehicle Development Platform. <https://developer.nvidia.com/drive/drive-hyperion>. Data dostępu: maj 2022.
- [20] NVIDIA Corporation. NVIDIA DRIVE SDK. <https://developer.nvidia.com/drive/drive-sdk>. Data dostępu: maj 2022.
- [21] NVIDIA Corporation. NVIDIA TensorRT. <https://developer.nvidia.com/tensorrt>. Data dostępu: maj 2022.
- [22] NVIDIA Corporation. Solutions for Self-Driving Cars and Autonomous Vehicles. <https://www.nvidia.com/en-us/self-driving-cars/>. Data dostępu: maj 2022.
- [23] NVIDIA Corporation. GeForce GTX 960M - Specifications. <https://www.nvidia.com/en-gb/geforce/gaming-laptops/geforce-gtx-960m/specifications/>, 2015. Data dostępu: maj 2022.
- [24] Joel Dapello, Tiago Marques, Martin Schrimpf, Franziska Geiger, David D. Cox, James J. DiCarlo. Simulating a Primary Visual Cortex at the Front of CNNs Improves Robustness to Image Perturbations. *bioRxiv*, 2020.
- [25] Ben Dickson. What is adversarial machine learning? <https://bdtechtalks.com/2020/07/15/machine-learning-adversarial-examples/>, 2020. Data dostępu: maj 2022.
- [26] Ben Dickson. Tesla AI chief explains why self-driving cars don't need lidar. <https://venturebeat.com/2021/07/03/tesla-ai-chief-explains-why-self-driving-cars-dont-need-lidar/>, 2021. Data dostępu: maj 2022.
- [27] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, Dawn Song. Robust Physical-World Attacks on Deep Learning Visual Classification. <https://arxiv.org/pdf/1707.08945.pdf>, 2017. Data dostępu: maj 2022.
- [28] The Linux Foundation®. Open Neural Network Exchange. <https://onnx.ai/>. Data dostępu: czerwiec 2022.

- [29] SpyBlood Games. Environmental Race Track Pack. <https://assetstore.unity.com/packages/3d/environments/roadways/environmental-race-track-pack-63493>, 2016. Data dostępu: czerwiec 2022.
- [30] Angel García. Vehicle Physics Pro - Features. <https://vehiclephysics.com/about/features/>. Data dostępu: maj 2022.
- [31] Angel García. Vehicle Physics Pro - VPVehicleController. <https://vehiclephysics.com/components/vehicle-controller/>. Data dostępu: czerwiec 2022.
- [32] Chirag Goyal. Complete Guide to Prevent Overfitting in Neural Networks. <https://www.analyticsvidhya.com/blog/2021/06/complete-guide-to-prevent-overfitting-in-neural-networks-part-1/>, 2021. Data dostępu: maj 2022.
- [33] Tuomas Haarnoja, Vitchyr Pong, Kristian Hartikainen, Aurick Zhou, Murtaza Dalal, Sergey Levine. Soft Actor Critic—Deep Reinforcement Learning with Real-World Robots. <https://bair.berkeley.edu/blog/2018/12/14/sac/>, 2018. Data dostępu: maj 2022.
- [34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. <https://arxiv.org/abs/1512.03385>, 2015. Data dostępu: maj 2022.
- [35] Jeff Hecht. Safety questions raised about 1550 nm lidar. <https://www.laserfocusworld.com/blogs/article/14040682/safety-questions-raised-about-1550-nm-lidar>, 2019. Data dostępu: maj 2022.
- [36] Elles Houweling. No, you won't get self-driving cars anytime soon. <https://www.verdict.co.uk/the-long-winding-road-to-self-driving-cars/>, 2021. Data dostępu: maj 2022.
- [37] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, Chrisina Jayne. Imitation Learning: A Survey of Learning Methods. *ACM Comput. Surv.*, 50(2), 2017. Data dostępu: maj 2022.
- [38] Dell Inc. Dell Inspiron 15 7000 series. https://dl.dell.com/manuals/all-products/esuprt_laptop/esuprt_inspiron_laptop/inspiron-15-7559-laptop_reference%20guide_en-us.pdf, 2015. Data dostępu: maj 2022.

- [39] Satish Jeyachandran. Introducing the 5th-generation Waymo Driver: Informed by experience, designed for scale, engineered to tackle more environments. <https://blog.waymo.com/2020/03/introducing-5th-generation-waymo-driver.html>, 2020. Data dostępu: maj 2022.
- [40] Joey Klender. Tesla's radar-less Autopilot performance improves in heavy rain. <https://www.teslarati.com/tesla-vision-autopilot-performance-heavy-rain-model-y-video/>, 2021. Data dostępu: maj 2022.
- [41] Simeon Kostadinov. Understanding Backpropagation Algorithm. <https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd>, 2019. Data dostępu: maj 2022.
- [42] Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. F. Pereira, C.J. Burges, L. Bottou, K.Q. Weinberger, redaktorzy, *Advances in Neural Information Processing Systems*, wolumen 25. Curran Associates, Inc., 2012.
- [43] Ajitesh Kumar. Real-World Applications of Convolutional Neural Networks. <https://vitalflux.com/real-world-applications-of-convolutional-neural-networks/>, 2021. Data dostępu: maj 2022.
- [44] Robert Kwiatkowski. Gradient Descent Algorithm — a deep dive. <https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21>, 2021. Data dostępu: maj 2022.
- [45] Marcin Lasota. Jaki język programowania - C#. <https://jaki-jezyk-programowania.pl/technologie/csharp/>, 2022. Data dostępu: maj 2022.
- [46] Marcin Lasota. Jaki język programowania - Unity. <https://jaki-jezyk-programowania.pl/technologie/unity>, 2022. Data dostępu: maj 2022.
- [47] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86:2278–2324, 1998.
- [48] Fei-Fei Li, Jiajun Wu, Ruohan Gao. CS231n: Convolutional Neural Networks for Visual Recognition. <https://cs231n.github.io/convolutional-networks>, 2022. Data dostępu: maj 2022.

- [49] Chris Nicholson. A Beginner’s Guide to Deep Reinforcement Learning. <https://wiki.pathmind.com/deep-reinforcement-learning>, 2021. Data dostępu: maj 2022.
- [50] Paul Pauls. A Primer on the Fundamental Concepts of Neuroevolution. <https://towardsdatascience.com/a-primer-on-the-fundamental-concepts-of-neuroevolution-9068f532f7f7>, 2020. Data dostępu: maj 2022.
- [51] Florian Petit. Sensor fusion – key components for autonomous driving. <https://www.blickfeld.com/blog/sensor-fusion-for-autonomous-driving/>, 2020. Data dostępu: maj 2022.
- [52] AutoPilot Review. LiDAR vs. Cameras for Self Driving Cars – What’s Best? <https://www.autopilotreview.com/lidar-vs-cameras-self-driving-cars/>. Data dostępu: maj 2022.
- [53] AutoPilot Review. Tesla Hardware 3 (Full Self-Driving Computer) Detailed. <https://www.autopilotreview.com/tesla-custom-ai-chips-hardware-3/>. Data dostępu: maj 2022.
- [54] Lutz Roeder. Netron, Visualizer for neural network, deep learning, and machine learning models. <https://github.com/lutzroeder/netron>, 2017. Data dostępu: maj 2022.
- [55] Mehreen Saeed. A Gentle Introduction To Sigmoid Function. <https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/>, 2021. Data dostępu: maj 2022.
- [56] John Shulman, Oleg Klimov, Filip Wolski, Prafulla Dhariwal, Alec Radford. Proximal Policy Optimization. <https://openai.com/blog/openai-baselines-ppo/>, 2017. Data dostępu: maj 2022.
- [57] Karen Simonyan, Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. <https://arxiv.org/abs/1409.1556>, 2014. Data dostępu: maj 2022.
- [58] Atul Singh. PREDICTION in Autonomous Vehicle - All You Need To Know. <https://towardsdatascience.com/prediction-in-autonomous-vehicle-all-you-need-to-know-d8811795fcde>, 2018. Data dostępu: maj 2022.

- [59] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, Andreas Moshovos. Memory Requirements for Convolutional Neural Network Hardware Accelerators. *2018 IEEE International Symposium on Workload Characterization (IISWC)*, strony 111–121, 2018.
- [60] synopsys. What is an Autonomous Car? <https://www.synopsys.com/automotive/what-is-autonomous-car.html>. Data dostępu: maj 2022.
- [61] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper with Convolutions. <https://arxiv.org/abs/1409.4842>, 2014. Data dostępu: maj 2022.
- [62] Unity Technologies. Quick guide to the Unity Asset Store. <https://unity3d.com/quick-guide-to-unity-asset-store>. Data dostępu: czerwiec 2022.
- [63] Unity Technologies. ML-Agents Toolkit Overview. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/ML-Agents-Overview.md, 2021. Data dostępu: maj 2022.
- [64] Unity Technologies. Training Configuration File. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/Training-Configuration-File.md, 2021. Data dostępu: czerwiec 2022.
- [65] Unity Technologies. Training ML-Agents. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/Training-ML-Agents.md, 2021. Data dostępu: czerwiec 2022.
- [66] Unity Technologies. Unity Documentation - RenderTexture. <https://docs.unity3d.com/2020.3/Documentation/ScriptReference/RenderTexture.html>, 2021. Data dostępu: maj 2022.
- [67] Unity Technologies. Unity Documentation Scripting API - BuildTarget. <https://docs.unity3d.com/2020.3/Documentation/ScriptReference/BuildTarget.html>, 2021. Data dostępu: maj 2022.
- [68] Unity Technologies. Unity ML-Agents Gym Wrapper. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/gym-unity/README.md, 2021. Data dostępu: maj 2022.

- [69] Unity Technologies. Unity ML-Agents Python Low Level API. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/Python-API.md, 2021. Data dostępu: maj 2022.
- [70] Unity Technologies. Using TensorBoard to Observe Training. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/Using-Tensorboard.md, 2021. Data dostępu: czerwiec 2022.
- [71] Brad Templeton. Elon Musk's War On LIDAR: Who Is Right And Why Do They Think That? <https://www.forbes.com/sites/bradtempleton/2019/05/06/elon-musks-war-on-lidar-who-is-right-and-why-do-they-think-that/?sh=7176c65b2a3b>, 2019. Data dostępu: maj 2022.
- [72] Tesla. Tesla Support - Autopilot and Full Self-Driving Capability. <https://www.tesla.com/support/autopilot>. Data dostępu: maj 2022.
- [73] Tesla. Tesla Support - Transitioning to Tesla Vision. <https://www.tesla.com/support/transitioning-tesla-vision>. Data dostępu: maj 2022.
- [74] Jason Torchinsky. Tesla's Removing Radar For Semi-Automated Driving On Models 3 And Y And I Don't Understand Why. <https://jalopnik.com/teslas-removing-radar-for-semi-automated-driving-on-mod-1846976679>, 2021. Data dostępu: maj 2022.
- [75] Kyle Wiggers. Tesla claims its latest self-driving chip is 7 times more powerful than its rivals'. <https://venturebeat.com/2019/04/22/tesla-claims-its-latest-self-driving-chip-is-six-times-more-powerful-than-its-rivals/>, 2019. Data dostępu: maj 2022.
- [76] Thomas Wood. What is the Softmax Function? <https://deeppai.org/machine-learning-glossary-and-terms/softmax-layer>. Data dostępu: maj 2022.
- [77] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9:611–629, 2018.