

UNIWERSYTET ŚLĄSKI W KATOWICACH
WYDZIAŁ NAUK ŚCISŁYCH I TECHNICZNYCH
INFORMATYKA INŻYNIERSKA

Grzegorz Galios
315850

Zastosowanie algorytmów ewolucyjnych w procesie nauki
sztucznych sieci neuronowych

PRACA DYPLOMOWA INŻYNIERSKA

Promotor: dr Rafał Skinderowicz

Katowice, 2020

Słowa kluczowe: *sieci neuronowe, algorytmy ewolucyjne*

Oświadczenie autora pracy

Ja, niżej podpisany:

imię (imiona) i nazwisko: Grzegorz Galios

autor pracy dyplomowej pt. „*Zastosowanie algorytmów ewolucyjnych w procesie nauki sztucznych sieci neuronowych*”

Numer albumu: 315850

Student Wydziału Nauk Ścisłych i Technicznych Uniwersytetu Śląskiego w Katowicach

kierunku studiów: Informatyka Inżynierska – studia stacjonarne I stopnia

specjalności: Inżynieria Systemów Informatycznych

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 Poziomy automatyzacji sterowania	3
1.2 Zasady działania	5
1.3 Wyzwania stawiane producentom	5
1.4 Korzyści z istnienia	6
1.5 Podejścia stosowane do rozwiązymania problemu	7
1.6 Typy istniejących rozwiązań	8
1.7 Wielkie projekty badawczo-rozwojowe	10
1.8 Środowiska symulacyjne	15
1.9 Projekty modelarskie	18
1.10 Małe prywatne projekty	21
2 Wstęp do uczenia maszynowego	24
2.1 Sieci neuronowe	24
2.2 Typy uczenia maszynowego	30
2.3 Algorytmy ewolucyjne	30
3 Projekt systemu i opis narzędzi	35
3.1 Projekt systemu	35
3.2 Zastosowane technologie	37
4 Opis implementacji	46
4.1 Środowisko Ucznia	46
4.2 Zewnętrzny proces Pythona	56
5 Eksperymenty obliczeniowe	63
5.1 Metodyka eksperymentów	63
5.2 Opis implementacji	64
5.3 Analiza uzyskanych wyników	65
5.4 Analiza wytrenowanego modelu	71
5.5 Wnioski z analiz	72

Podsumowanie	73
Spis rysunków	74
Bibliografia	85

Wstęp

Przedmiotem pracy jest wykorzystanie technik uczenia maszynowego do rozwiązywania problemu sterowania samochodem autonomicznym. Jest to bardzo skomplikowane zagadnienie, dlatego większość uwagi została poświęcona dobremu zrozumieniu podstawowych jego aspektów.

Cel i zakres pracy

Celem niniejszej pracy jest opracowanie prostego systemu uczącego sieci neuronowe w oparciu o symulacje przeprowadzane w wymodelowanym środowisku. Problemem rozwiązywanym przez sieć neuronową jest nawigowanie samochodem po torze wyścigowym. Zakres pracy obejmuje następujące zagadnienia:

1. Przegląd literatury na temat samochodów autonomicznych, wyzwań stojących przed ich twórcami oraz przykładów istniejących rozwiązań z tego zakresu.
2. Przegląd literatury na temat sieci neuronowych i algorytmów ewolucyjnych.
3. Wybór algorytmów ewolucyjnych wykorzystywanych podczas uczenia sieci.
4. Zaprojektowanie aplikacji oraz wybór odpowiednich narzędzi, które ułatwiają jej implementację.
5. Implementacja aplikacji uczącej sieci neuronowej w oparciu o sygnały dostarczane ze środowiska symulacji.
6. Przeprowadzenie eksperymentów obliczeniowych poprzez wykorzystanie utworzonej aplikacji oraz analiza wyników uzyskanych z tych eksperymentów.

Struktura pracy

Praca składa się z pięciu numerowanych rozdziałów. Każdy rozdział dotyczy konkretnego aspektu omawianego tematu, a kolejność rozdziałów realizuje zasadę „*od ogólnego do szczegółu*”. Praca rozpoczyna się od rozdziałów omawiających ogólne zagadnienia teoretyczne, niezbędne do zrozumienia dalszych rozdziałów pracy. Kończy się natomiast rozdziałem będącym opisem bardzo konkretnych i praktycznych aspektów tematu.

Oto lista rozdziałów zawartych w tej pracy:

1. Samochody autonomiczne

Rozdział opisujący problem samochodów autonomicznych. Celem tego rozdziału jest wyrobienie u czytelnika intuicji na temat tego, czym są samochody autonomiczne i jakie wyzwania stoją przed twórcami takich pojazdów. Podczas opisywania tego problemu zostało poruszonych wiele kwestii, których świadomość jest niezwykle ważna do odpowiedniego zrozumienia całej sprawy.

2. Wstęp do uczenia maszynowego

Rozdział zawiera najważniejsze zagadnienia teoretyczne z zakresu sztucznych sieci neuronowych oraz algorytmów ewolucyjnych. Opisane zagadnienia stanowią podstawę teoretyczną, której znajomość jest niezbędna do zrozumienia dalszych rozdziałów pracy.

3. Projekt systemu i opis narzędzi

Rozdział składa się z dwóch części. W pierwszej części zawarte są główne założenia projektowe, jakie zostały przyjęte przed rozpoczęciem prac nad aplikacją. Natomiast druga część rozdziału to opis technologii, jakie zostały wykorzystane podczas implementowania aplikacji.

4. Opis implementacji

Rozdział opisuje implementację aplikacji wykonanej na potrzeby niniejszej pracy. W rozdziale zawarte są stosunkowo szczegółowe informacje na temat poszczególnych komponentów wchodzących w skład aplikacji.

5. Eksperymenty obliczeniowe

W tym rozdziale przedstawiona jest metodyka przeprowadzania eksperymentów oraz analiza uzyskanych tą drogą wyników. Wyniki eksperymentów obliczeniowych pozwalają na wyciągnięcie pewnych wartościowych wniosków, które również zostały opisane w tym rozdziale.

Rozdział 1

Samochody autonomiczne

Autonomiczny samochód to pojazd, który potrafi interpretować swoje otoczenie oraz bezpiecznie poruszać się po nim bez potrzeby ingerencji człowieka [78]. Taki pojazd jest zdolny do wykonywania wszelkich manewrów, jakie mógłby wykonać doświadczony kierowca.

1.1 Poziomy automatyzacji sterowania

Organizacja SAE (Society of Automotive Engineers) definiuje skalę, zakładającą sześć poziomów automatyzacji sterowania pojazdów [77]:

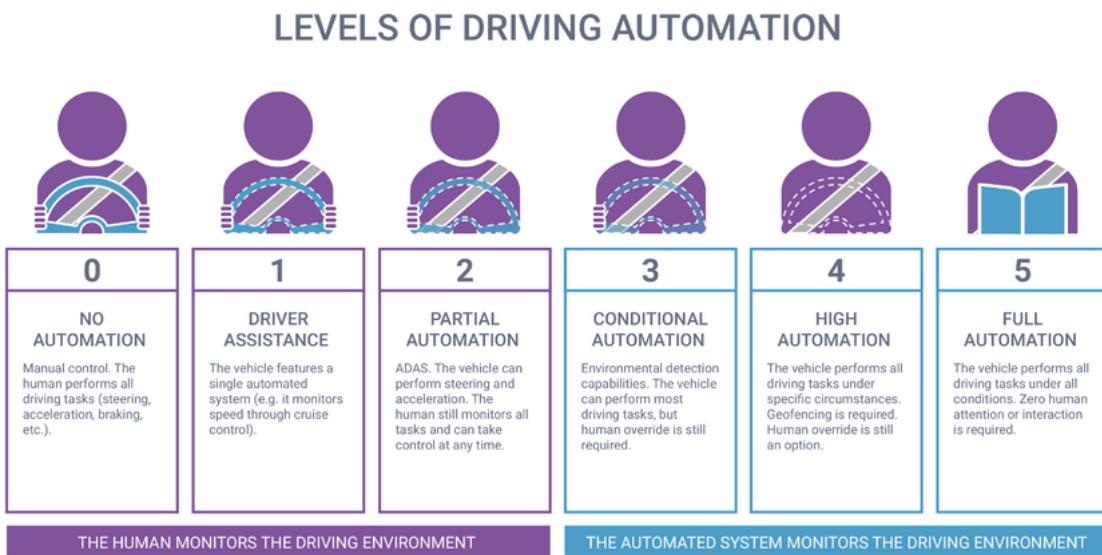
1. Poziom 0 - sterowanie w pełni ręczne (**No Driving Automation**). Dotyczy większości samochodów poruszających się obecnie po drogach.
2. Poziom 1 - wsparcie kierowcy (**Driver Assistance**). Najniższy poziom automatyzacji. Samochód posiada prosty system wspomagający pojedyncze aspekty sterowania, np. kontrola kierownicy lub kontrola przyspieszenia samochodu (tzw. dynamiczny tempomat). Dynamiczny tempomat kwalifikuje się na poziom 1, ponieważ kierowca musi samodzielnie kontrolować pozostałe aspekty sterowania samochodem.
3. Poziom 2 - częściowa automatyzacja sterowania (**Partial Driving Automation**). Oznacza stosowanie zaawansowanych systemów wspomagania kierowcy (ADAS - Advanced Driver Assistant Systems) [38]. Samochód potrafi kontrolować zarówno kierownicę, jak również prędkość pojazdu. Samochód nie jest jednak w pełni autonomiczny, więc osoba siedząca za kierownicą może w każdej chwili przejąć kontrolę nad samochodem. Przykładem systemu będącego na Poziomie 2 jest Tesla Autopilot.
4. Poziom 3 - warunkowa automatyzacja sterowania (**Conditional Driving Automation**). Samochody posiadają system detekcji otoczenia, więc mogą podejmować decyzje, takie jak przyspieszanie obok powoli poruszającego się pojazdu. Jednakże wciąż wymagają nadzoru człowieka. Kierowca musi być przygotowany, że w każdej chwili będzie musiał przejąć kontrolę nad samochodem, jeśli system nie poradzi sobie w danej sytuacji.

Obecna generacja samochodu Audi A8 oferuje system Traffic Jam Pilot [59], który zapewnia autonomicznosć sterowania na Poziomie 3. Jest to pierwszy przypadek, w którym samochód produkowany seryjnie oferuje taki system [57].

5. Poziom 4 - wysoka automatyzacja sterowania (**High Driving Automation**). Samochód potrafi interweniować jeśli „coś pójdzie nie tak”, np. jeśli dojdzie do błędu systemu. W tej sytuacji, samochód nie potrzebuje ludzkiej ingerencji *w większości przypadków*. Jednakże, kierowca wciąż może przejąć kontrolę nad samochodem. Przykłady systemów reprezentujących Poziom 4:

- Samochody elektryczne marki Navya - Autonom Shuttle oraz Autonom Cab [85],
- Firma Waymo udostępniła usługę taksówkarską w Arizonie, gdzie samochody z autonomicznością na Poziomie 4 przewożą klientów [72].

6. Poziom 5 - pełna automatyzacja sterowania (Full Driving Automation). Zachowanie pojazdu nigdy nie wymaga uwagi człowieka. Samochody Poziomu 5 mogą nawet nie posiadać kierownicy ani pedałów. Będą wolne od ograniczeń obszarowych (geofencing), na których mogą się poruszać. W pełni autonomiczne samochody są obecnie testowane w kilku miejscach na świecie, jednakże żaden z tych systemów nie jest dostępny publicznie.



Rysunek 1.1: Poziomy automatyzacji sterowania samochodem (według SAE)

Skala ta została zaakceptowana przez Amerykański Departament Transportu. Jej wizualizacja została przedstawiona na rysunku 1.1.

1.2 Zasady działania

Autonomiczne samochody korzystają z wszelkiego rodzaju urządzeń rejestrujących stan otoczenia, takich jak czujniki, radary i kamery. Radary monitorują pozycję obiektów znajdujących się wokół pojazdu, kamery video rozpoznają znaki drogowe oraz tor ruchu innych obiektów. Czujniki LIDAR (Light Detection and Ranging) odbijają impulsy świetlne od otoczenia samochodu. Dzięki temu są w stanie odmierzać dystans, rozpoznawać pobocza dróg oraz identyfikować oznaczenia pasa ruchu. Czujniki ultradźwiękowe, montowane w kołach, rozpoznają krawężniki oraz inne pojazdy podczas parkowania.

Oprogramowanie przetwarza dane z urządzeń wejściowych, oblicza parametry dalszej jazdy (takie jak kierunek lub prędkość) i wysyła je do silników samochodu, które kontrolują przyspieszenie, hamowanie i kąt skrętu kierownicy.

1.3 Wyzwania stawiane producentom

W pełni autonomiczne samochody są poddawane wyczerpującym testom w kilku miejscowościach na świecie, jednakże żaden z nich nie jest obecnie dostępny publicznie. Od wprowadzenia takich systemów do produkcji seryjnej dzieli nas jeszcze wiele lat [60]. Ponadto, przed projektantami i producentami stoi szereg wyzwań na polu technologicznym, prawnym, środowiskowym, a nawet filozoficznym.

Część z tych wyzwań to:

1. LIDAR - problem wzajemnego zakłócania sygnałów w sytuacji, gdy wiele autonomicznych samochodów będzie jechało blisko siebie. W sytuacji stosowania wielu różnych częstotliwości, czy zakres częstotliwości będzie wystarczająco szeroki aby obsłużyć masową produkcję takich pojazdów?
2. Warunki pogodowe - co się dzieje, gdy samochody autonomiczne podróżują w bardzo ciężkich warunkach? Jeśli śnieg zalega na drodze, oznaczenia pasa ruchu są niewidoczne. Jak kamery i czujniki będą rozpoznawać pasy ruchu, jeśli oznaczenia są zamazywane przez wodę, lód, olej lub błoto?
3. Regulacje prawne - wiele kwestii wymaga uregulowania, co nie będzie łatwe, bo dotyczy bardzo skomplikowanych zagadnień. Najistotniejszym z nich będzie kwestia **ustalenia odpowiedzialności prawnej za ewentualne wypadki powodowane przez samochody autonomiczne**.

Cytując Norberta Biedrzyckiego z portalu Business Insider [23]:

„*Kto w sytuacji kolizji, zagrożenia czy nawet utraty życia, staje się przedmiotem sporu, możliwego pozwu, czy odszkodowania? Kto ma ponosić zasadniczą odpowiedzialność w sytuacji, w której giną osoby w wypadku z udziałem autonomicznych pojazdów? Czy kłopoty będzie miał producent algorytmu, w oparciu o który porusza się autonomiczne auto na drodze, firma produkująca takie auto, czy kierowca – właściciel? A jeśli ten ostatni spowoduje wypadek, czy może on liczyć na wystarczające rozwiązania związane z ubezpieczeniem, które zapewniają mu komfort użytkowania?*

Na tym przykładzie widać wyraźnie, że możliwości technologiczne wyprzedzają nasze czasy. Tutaj właściwie wszystko jest możliwe – bo całe floty autonomicznych ciężarówek czekają już, by wyruszyć w drogę, a myśl o tym, by budować usługi taksówkarskie w oparciu o autonomiczne pojazdy, zaprzata głowę niejednemu biznesmenowi.

Okazuje się jednak, że podstawową barierą, która jeszcze długo będzie blokować zmiany na światowych drogach, będą kwestie niewystarczających uregulowań prawnych.”

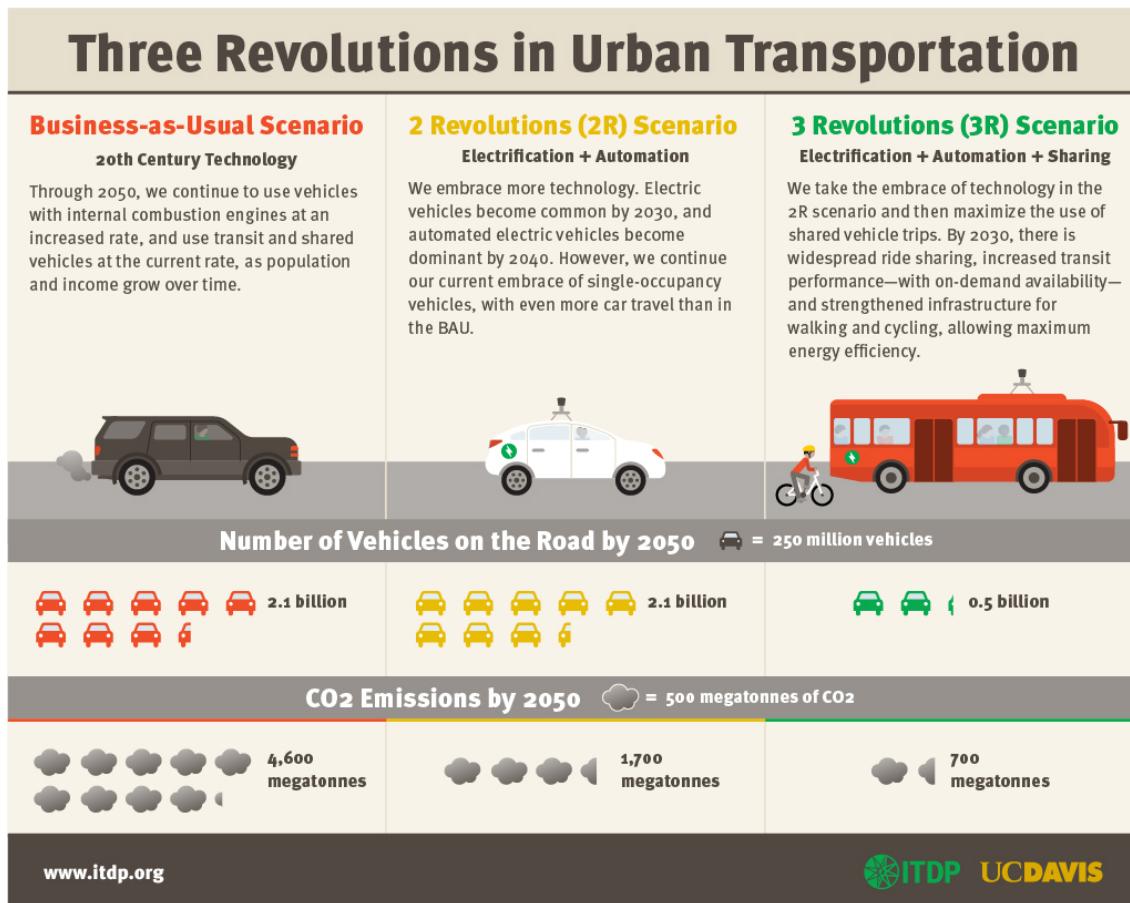
1.4 Korzyści z istnienia

Scenariusze, w których fakt wykorzystania pojazdów autonomicznych podnosi poziom wygody i jakości życia, wydają się nieograniczone. Osoby starsze i niepełnosprawne mogą zyskać niezależność. Można wysłać psa do weterynarza. Albo przywieźć dzieciom na obóz brakujące rzeczy, których zapomniały zapakować.

Oprócz tego, z samochodami autonomicznymi wiąże się ogromną nadzieję, iż przyczynią się do drastycznego obniżenia emisji dwutlenku węgla do atmosfery. W artykule z 2017 roku, opublikowanego przez organizację ITDP, zamieszczono analizę na temat przyszłości transportu drogowego w kontekście wykorzystania samochodów autonomicznych [46]. Autorzy analizy zwrócili uwagę na trzy tendencje we współczesnym spojrzeniu na transport, które jeśli będą występować jednocześnie, uwolnią pełny potencjał drzemiący w samochodach autonomicznych: **automatyzacja, elektryfikacja oraz współdzielenie środków transportu (ridesharing).**

Autorzy szacują, że dzięki tym trzem „rewolucjom”, do roku 2050 będzie można osiągnąć:

1. Zmniejszenie natężenia ruchu miejskiego o 30%;
2. Oszczędności w kosztach transportu na poziomie 40%;
3. Redukcję dwutlenku węgla emitowanego przez miasta o 80% w skali światowej.



1.5 Podejścia stosowane do rozwiązymania problemu

Stworzenie samochodu autonomicznego, który będzie w stanie samodzielnie poruszać się po otoczeniu i służyć jako środek transportu dla milionów ludzi na świecie, nie jest zadaniem trywialnym. Wymaga rozwiązyania szeregu problemów związanych z tym zagadnieniem. Takimi problemami są m.in. rozpoznawanie znaków drogowych oraz predykcja zachowania innych uczestników ruchu drogowego. Każdy z tych problemów może zostać rozpatrzony na wiele sposobów, istnieje zatem wiele alternatywnych rozwiązań dla danego problemu. Często nie można stwierdzić, czy dane rozwiązanie jest lepsze od reszty. Po prostu każde takie rozwiązanie ma swoją specyfikę, a co za tym idzie swoje wady i zalety.

Dlatego jest oczywiste, że firmy podejmujące się zbudowania samochodu autonomicznego mogą przyjmować różne, czasem skrajne podejścia w kwestii rozwiązań stosowanych w procesie tworzenia takiej maszyny.

Jednym z najtrudniejszych problemów do rozwiązyania jest kwestia orientacji w terenie. Zagadnienie te zostało pokrótko przedstawione w artykule [29]. Artykuł prezentuje również

trzy propozycje rozwiązania tego problemu. Warto zauważyć, że każda z tych propozycji znaczco różni się od pozostałych.

1.6 Typy istniejących rozwiązań

Tematyka tworzenia samochodów autonomicznych, bądź też samochodów o dużym stopniu automatyzacji sterowania, stała się w ostatnich latach niezwykle popularna. Z tego powodu istnieje obecnie ogromna liczba projektów nawiązujących do tej tematyki.

Projekty te tworzą bardzo interesujące spektrum przypadków. Różnią się między sobą w wielu aspektach, takich jak:

- stopień skomplikowania projektu,
- podejście do rozwiązywania poszczególnych aspektów projektu,
- zakres tematyczny projektu,
- środowisko testowania,
- i wiele innych.

Na podstawie własnych obserwacji, postanowiłem podzielić te projekty na kilka zasadniczych grup:

1. Wielkie projekty badawczo-rozwojowe

Są to projekty o największym stopniu skomplikowania. Rozwijane przez zespoły składające się z setek lub tysięcy ludzi. Czasem są to zespoły pracujące w ramach międzynarodowych korporacji posiadających gigantyczne budżety i zaplecze kadrowo-techniczne. Innym razem są to projekty open source, w które zaangażowani są ludzie z różnych zakątków świata. Często zdarza się, iż dany projekt jest rozwijany dzięki współpracy kilku partnerów, działających w różnych sektorach gospodarczych lub badawczych.

Projekty należące do powyższej grupy podchodzą do problemu tworzenia samochodu autonomicznego w sposób najbardziej kompleksowy. Ich celem jest najczęściej stworzenie pojazdu poruszającego się w rzeczywistym środowisku, który będzie w stanie poradzić sobie w każdych warunkach bez potrzeby ingerencji człowieka. Część z tych projektów jest rozwijana z myślą o wdrożeniu do produkcji seryjnej i zastosowaniu w transporcie drogowym, m.in. do przewozu osób na masową skalę. Inne projekty mają charakter czysto badawczy. Ich celem jest eksploracja nowych strategii rozwiązywania tego problemu. Jeśli dana strategia okaże się dostatecznie dobra, to może zostać

wdrożona do systemów produkowanych seryjnie.

2. Środowiska symulacyjne

Nie są to projekty ściśle związane z tworzeniem samochodu autonomicznego. Są to raczej projekty, którego celem jest ułatwienie prac nad rozwojem takich maszyn innym twórcom. Osiągane jest to poprzez tworzenie wirtualnych środowisk symulacyjnych, dostosowanych do tego typu potrzeb.

Praca w symulatorze posiada wiele zalet względem operowania na rzeczywistych maszynach. Najważniejszą z nich jest ogromna oszczędność czasu i pieniędzy. Symulatory pozwalają na szybkie i bezpieczne testowanie wymyślanych rozwiązań. Faktem jest, że wszystkie firmy pracujące nad wielkimi projektami badawczo - rozwojowymi, w mniejszym lub większym stopniu korzystają z takich środowisk symulacyjnych.

3. Projekty modelarskie

To projekty, które nie są realizowane ani na rzeczywistych samochodach, ani w środowiskach symulacyjnych. Są to projekty, w których eksperymenty przeprowadzane są na miniaturowych modelach samochodów. Czasami te modele nie są odwzorowaniem żadnego rzeczywistego samochodu. Mogą to być projekty wysokobudżetowe, rozwijane w celach komercyjnych.

4. Małe prywatne projekty

Są to niewielkie projekty, realizowane przez pojedyncze osoby lub małe zespoły złożone z kilku osób. Często podchodzą do problemu w sposób mniej kompleksowy, skupiając się na eksploracji tylko wybranych aspektów problemu. Są to projekty niskobudżetowe, realizowane hobbystycznie, najczęściej w celach edukacyjnych. Rzadko się zdarza, że takie projekty są komercjalizowane. Często są jednak udostępniane publicznie.

Częściej takie projekty są realizowane w środowiskach symulacyjnych (czasem własnoręcznie tworzonych), rzadziej są to projekty modelarskie. Zdecydowanie najrzadszą, praktycznie niewystępującą grupą, są projekty realizowane na rzeczywistych samochodach. Wiąże się to oczywiście z kosztami, jakie należałyby ponieść w takim wypadku.

Chciałbym w tym miejscu zaznaczyć, że powyższy podział ma charakter obiektywny. Użyte przeze mnie sformułowania mogą nie być precyzyjne, co może prowadzić do nieporozumień i mylnych interpretacji. Ponadto, klasyfikacja którą przedstawiłem może nie być kompletna. Być może istnieją projekty, których nie można zaklasyfikować do żadnej z powyższych grup. Temat wymaga dalszych badań, co niestety wykracza poza zakres tej pracy. W dalszej części rozdziału zostaną zaprezentowane przykłady dla każdej z opisanych grup.

1.7 Wielkie projekty badawczo-rozwojowe

W obecnych czasach, wiele korporacji o zasięgu międzynarodowym angażuje się w prace nad rozwojem samochodów autonomicznych. Niniejsze zestawienie zawiera tylko bardzo mały wycinek z listy istniejących rozwiązań.

1.7.1 Waymo

Przedsiębiorstwo technologiczne, które od grudnia 2016 roku należy do holdingu Alphabet Inc. Wcześniej istniało jako jeden z projektów firmy Google. Waymo zajmuje się rozwojem samochodów autonomicznych. Początki sięgają roku 2009, kiedy to firma Google rozpoczęła prace nad pierwszym prototypem w sekretnym laboratorium. Więcej na temat historii projektu można dowiedzieć się w artykule [43].

System na obecnym etapie rozwoju jest uznawany za jeden z najbardziej dojrzałych i przetestowanych systemów istniejących na świecie. Ma za sobą przejechanych kilka milionów mil w rzeczywistym świecie i wiele miliardów mil odbytych w środowiskach symulacji. System był testowany w wielu lokalizacjach znajdujących się na terenie USA. Poprzez jazdy odbywane w bardzo zróżnicowanych warunkach pogodowych, auta były przystosowywane do radzenia sobie w bardzo szerokim spektrum możliwych scenariuszy drogowych.

Dla każdego obszaru, po którym ma się poruszać samochód Waymo, zespół musi wcześniej przygotować szczegółowe mapy 3D z uwzględnieniem takich informacji, jak profile dróg [86], krawężniki i chodniki oraz wszelkiego rodzaju oznaczenia należące do infrastruktury drogowej (jak np. pasy ruchu, znaki drogowe czy sygnalizacja świetlna).

Czujniki - zainstalowane w samochodzie i zarządzane przez skomplikowane oprogramowanie - stale skanują obszar wokół samochodu. Zbierają dane o wszystkich obiektach obecnych na dystansie do trzech długości boiska piłkarskiego w każdą stronę. Skanowane obiekty to m.in. uczestnicy ruchu drogowego (np. piesi, rowerzyści, inne samochody), sygnalizacja świetlna czy prowizoryczne konstrukcje drogowe stawiane na czas remontów (pachołki, słupki itd.).

Oprócz poprawnej identyfikacji zarejestrowanych obiektów, oprogramowanie samochodu potrafi również przewidywać ich zachowanie w najbliższej przyszłości. Predykcja jest dokonywana na podstawie pomiaru prędkości oraz kierunku ruchu. Samochód wie, że różne rodzaje obiektów (np. pieszy, motocyklista, ciężarówka) poruszają się w odmienny sposób. Dla każdego z tych rodzajów samochód jest w stanie trafnie określić wiele możliwych trajektorii ruchu obiektów.

Na podstawie zgromadzonych danych, oprogramowanie może określić dokładną trajektorię ruchu, prędkość, linię przejazdu oraz inne parametry jazdy, niezbędne do dalszego przemieszczania się samochodu w bezpieczny sposób.

Więcej informacji na temat firmy Waymo i rozwijanych przez nią projektów można znaleźć m.in. na ich oficjalnej stronie internetowej: <https://waymo.com>.

1.7.2 NVIDIA

Firma NVIDIA od wielu lat angażuje się w prace badawczo-rozwojowe nad szeroko rozumianą tematyką uczenia maszynowego. Jednym ze szczególnych obszarów zainteresowań firmy jest branża automotive. NVIDIA nie zajmuje się produkcją samochodów autonomicznych, lecz wytwarzaniem kompleksowych rozwiązań sprzętowo-programowych, które mogą stanowić bazę do tworzenia takich pojazdów. NVIDIA współpracuje z wieloma partnerami, którzy kupują od niej te rozwiązania i wykorzystują do rozwoju własnych systemów. Obecnie jest to ponad 370 partnerów [61]. Wśród nich są między innymi: Toyota, Volkswagen, Audi, Volvo.

Strategia biznesowa, oparta na otwarciu się na współpracę z partnerami działającymi w branży transportowej (i nie tylko), przynosi firmie wymierne korzyści finansowe [28]. Duże zainteresowanie współpracą z firmą NVIDIA jest spowodowane faktem, iż oferowany przez nich system otwiera drogę do zupełnie nowych możliwości. Pozwala między innymi na szybkie wdrożenie usługi przewozów miejskich, o czym przekonała się firma Optimus Ride [27].

Rozwiązania w zakresie samochodów autonomicznych, oferowane przez firmę NVIDIA, występują pod wspólną nazwą NVIDIA DRIVE [67]. Rozwiązania NVIDIA DRIVE można podzielić na część sprzętową (hardware) i programową (software).

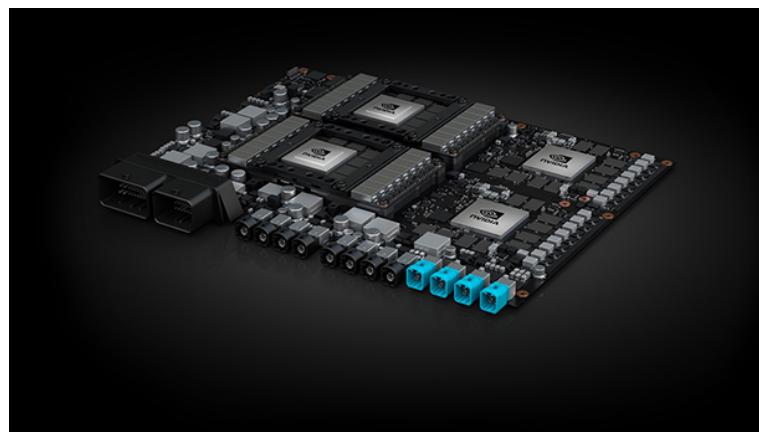
Rozwiązania sprzętowe - NVIDIA DRIVE AGX

NVIDIA DRIVE AGX to rodzina platform obliczeniowych, zbudowanych na bazie procesora NVIDIA Xavier. Obecnie w skład tej rodziny wchodzą dwa modele:

1. NVIDIA DRIVE AGX Pegasus

Osiąga moc obliczeniową 320 TOPS (Tera Operations Per Second), co czyni go naj szybszym rozwiązaniem dostępnym obecnie na rynku. Jego architektura oparta jest na dwóch procesorach NVIDIA Xavier i dwóch kartach graficznych TensorCore.

Cechuje się wysoką efektywnością energetyczną. Potrafi utrzymywać w wykonaniu wiele sieci neuronowych jednocześnie. Został zaprojektowany do bezpiecznej obsługi wysoce zautomatyzowanych oraz w pełni autonomicznych systemów jazdy. Kierowcza i pedały nie są wymagane.



Rysunek 1.2: NVIDIA DRIVE AGX Pegasus

2. NVIDIA DRIVE AGX Xavier

Jest mniejszy od Pegasusa. Dostarcza jedynie 30 TOPS mocy obliczeniowej. Charakteryzuje się jednak bardzo niskim zużyciem energii, pobiera tylko 30 watów. Przeznaczony dla systemów wspomagania kierowcy. Zbyt mało wydajny dla wsparcia w pełni autonomicznych systemów jazdy.



Rysunek 1.3: NVIDIA DRIVE AGX Xavier

3. NVIDIA DRIVE Hyperion

Najbardziej kompleksowe rozwiązanie oferowane przez firmę NVIDIA. W jego skład wchodzi platforma obliczeniowa (Xavier/Pegasus), zestaw kamer/czujników oraz oprogramowanie sterujące.

Więcej informacji o oprogramowaniu znajdzie się w dalszej części rozdziału.

Oprogramowanie NVIDIA DRIVE

W skład pakietu oprogramowania NVIDIA DRIVE wchodzą m.in. następujące komponenty [66]:

1. DRIVE OS Linux

Podstawowy komponent oprogramowania, składający się m.in z wbudowanego systemu czasu rzeczywistego (RTOS), bibliotek NVIDIA CUDA, platformy NVIDIA TensorRT [65] oraz wielu innych modułów. DRIVE OS zapewnia bezpieczne środowisko uruchomieniowe dla aplikacji. Pełni funkcję systemu operacyjnego dla platform obliczeniowych z rodziny NVIDIA DRIVE AGX.

2. DRIVE AV

Dostarcza modułów dla percepji otoczenia (NVIDIA DRIVE Perception [63]), mapowania środowiska (NVIDIA DRIVE Mapping [62]) i planowania trasy (NVIDIA DRIVE Planning [64])

3. DRIVE IX

Dostarcza algorytmów do wizualizacji otoczenia pojazdu, monitoruje zachowanie kierowcy i służy jako asystent kabiny.

4. DriveWorks

Jest frameworkiem dostarczającym zestaw bibliotek i narzędzi dla użytkowników pakietu NVIDIA DRIVE.

Wizualizacja pakietu NVIDIA DRIVE została zaprezentowana na rysunku 1.4.

1.7.3 Udacity

Udacity jest edukacyjną organizacją oferującą tzw. „masywne otwarte kursy online” (MOOCs - Massive Open Online Courses) [24]. Posiadają własną platformę kursową, na której publikowane są kursy dotyczące szeroko pojętej branży IT.

Jednym z kursów, oferowanych przez Udacity, jest *Self-Driving Car Engineer Nanodegree program*. Został stworzony w partnerstwie z gigantami branż automotive oraz IT, m.in. Mercedes-Benz, NVIDIA, Uber, BMW.



Rysunek 1.4: Oprogramowanie wchodzące w skład pakietu NVIDIA DRIVE

Wraz z uruchomieniem kursu, firma Udacity zdecydowała się na dosyć odważny krok. Postanowiła zrealizować, we współpracy z programistami z całego świata, własny projekt samochodu autonomicznego. Projekt ten miał być w całości **otwartoźródłowy**.

Jak opisuje Oliver Cameron, ówczesny lider programu Udacity Self-Driving Car [26]:
 „Gdy decydowaliśmy się na tworzenie programu *Self-Driving Car Engineer*, od razu wiedzeliśmy, że musimy zbudować własny samochód autonomiczny. (...) utworzyliśmy więc zespół *Self-Driving Car Team*. Jedna z pierwszych decyzji jaką podjęliśmy? **Kod open source, pisany przez setki studentów z całego świata!**”

Aby wcielić ten projekt w życie, firma Udacity dokonała niezbędnych przygotowań:

1. Zakup samochodu marki Lincoln MKZ (rocznik 2016)
2. Montaż dodatkowych komponentów:
 - Dwa radarów LIDAR marki Velodyne VLP-16

- Jeden radar marki Delphi
- Trzy kamery marki Point Grey Blackfly
- Xsens IMU [15]
- ECU [14]
- i wiele innych

3. Konfiguracja systemu ROS (Robot Operating System) [1]
4. Stworzenie wstępnej bazy kodu

Po przygotowaniu platformy pod rozwój projektu, można było rozpocząć prace nad tworzeniem samochodu autonomicznego. Ponieważ naczelnym priorytetem Udacity jest bezpieczeństwo, postanowiono że problem będzie rozbitły na kilka podproblemów, a dla każdego z nich zostanie zorganizowane tzw. „wyzwanie Udacity” (*Udacity Challenge*).

Tak też się stało. Oferowane były atrakcyjne nagrody pieniężne i rzeczowe, a organizatorzy wyzwań mieli okazję na dokładne przetestowanie proponowanych rozwiązań na długo przed zainstalowaniem ich w rzeczywistym samochodzie. Do tej pory zorganizowano kilka takich wyzwań, o szczegółach jednego z nich można przeczytać w [25].

1.8 Środowiska symulacyjne

Rozwój samochodów autonomicznych jest bardzo trudnym zadaniem. Jednym z wyzwań stojących przed konstruktorami jest konieczność testowania prototypowanych rozwiązań w samochodzie. Przeprowadzanie takich testów w prawdziwym świecie, na drogach publicznych, jest dla projektów będących na wczesnym etapie rozwoju pomysłem nie tylko bardzo ryzykownym, ale również dość kosztownym. Nawet zakładając, że testy odbywają się w zamkniętym, kontrolowanym środowisku, to i tak jest to proces długi, kosztowny i obarczony wadami.

Aby go ułatwić, a także uczynić tańszym, szybszym i bezpieczniejszym, bardzo popularną techniką jest przeprowadzanie testów w środowisku symulacyjnym. Obecnie istnieje na rynku wiele symulatorów, które są wykorzystywane do powyższych celów. W niniejszej sekcji przedstawię kilka przykładów.

1.8.1 Microsoft AirSim

Jest symulatorem dla dronów, samochodów i innych pojazdów. Zbudowany na bazie silnika Unreal. Posiada również eksperymentalne wsparcie dla silnika Unity. Jest projektem open source, wspierającym wiele kontrolerów lotu [39], takich jak na przykład PX4. Projekt jest rozwijany jako wtyczka dla silnika Unreal, którą można łatwo podpiąć do innych projektów zrealizowanych w tym silniku. Jak twierdzą twórcy simulatora [69]:

„Naszym celem jest aby AirSim służył jako platforma do badań z zakresu AI, uczenia głębo-kiego, wizji komputerowej oraz algorytmów uczenia ze wzmacnieniem dla pojazdów autonomicznych. AirSim udostępnia multiplatformowe API do pozyskiwania danych i kontrolowania pojazdów.”

Symulator posiada rozszerzalną architekturę. Można w łatwy sposób dodawać wsparcie dla nowych typów pojazdów czy platform sprzętowych. Pozwala również konfigurować istniejące modele i uzupełniać je o dodatkowe komponenty, takie jak czujniki lub kamery.

Więcej informacji na temat symulatora AirSim można znaleźć w następujących źródłach:

- Oficjalna strona projektu [70]
- Artykuł naukowy pracowników zespołu Microsoft Research [73]
- Repozytorium projektu na platformie GitHub [2]



Rysunek 1.5: Stopklatka z pracy symulatora AirSim

1.8.2 Voyage Deepdrive

Darmowy, otwartoźródłowy symulator przeznaczony do rozwoju samochodów autonomicznych. Zapoczątkowany przez Craiga Quitera, programistę od lat zaangażowanego w tworzenie oprogramowania open source.

Jak opisuje Drew Gray, szef działu technologicznego [35] w firmie Voyage [40]:
„Historia Deepdrive ma swój początek kilka lat temu, gdy Craig - zainspirowany postępem w dziedzinie AI oraz korzyściami jakie społeczeństwo zyska dzięki samochodom autonomicznym - rozpoczął projekt otwartoźródłowej platformy, która daje możliwość rozwoju technologii self-driving każdej osobie na świecie.”

Gray znał Quitera od dawna. Wiedział o rozwijanym przez niego projekcie i namówił go do współpracy z firmą Voyage. Quiter dołączył do firmy, która to objęła patronat nad dalszym rozwojem symulatora. Niemniej jednak projekt zachował swój pierwotny, otwartoźródłowy charakter.

Deepdrive posiada wiele funkcjonalności, które czynią go atrakcyjnym produktem dla potencjalnych użytkowników. Są to między innymi:

- Łatwy dostęp do danych z czujników
- Trzy zróżnicowane środowiska symulacyjne (mapy)
- Zaawansowane wbudowane AI - agenci mogą m.in. wyprzedzać inne pojazdy, wyznaczać trasy lub inteligentnie radzić sobie na skrzyżowaniach.
- Skrypty Python Unreal - całościowe API silnika Unreal [37] dla języka Python

Symulator Deepdrive kładzie główny nacisk na uczenie kompleksowe (tzw. *end-to-end learning*) oraz głębokie uczenie ze wzmacnieniem (*deep reinforcement learning*). Naukowcy korzystający z symulatora Deepdrive mogą się skupić na rozwiązywanym problemie, a nie na sposobie jego implementacji w symulatorze. Twórcy symulatora mają nadzieję, że dzięki takiemu podejściu praca badawcza nad rozwojem samochodów autonomicznych znacznie przyspieszy.

Praca nad projektem Deepdrive jest ukierunkowana uwagami przekazywanymi przez organizacje, które na co dzień zajmują się rozwijaniem technologii samochodów autonomicznych. Twórcom symulatora zależy na tym, aby Deepdrive odpowiadał realnym potrzebom twórców takich pojazdów.

Wśród nowości w projekcie, zaproponowanych przez firmę Voyage, znajdują się m. in.:

- Tablica liderów (*Leaderboard*) - usługa oferowana w ramach symulatora Deepdrive. Zawiera zestawienia rankingowe dla wyzwań ogłoszonych przez zespół Voyage Deepdrive. Pozwala na rywalizację zespołów badawczych, pracujących nad rozwiązyaniem danego problemu. Każdy kto zapisze się do usługi, może zgłosić własne rozwiązanie, które następnie zostanie ocenione według kryteriów zdefiniowanych przez organizatorów wyzwania. Więcej informacji znajduje się na oficjalnej stronie usługi [31].
- Szczegółowe mapy świata 3D, zbudowane przy pomocy narzędzia *Parallel Domain* [44]. Bogate środowiska symulacyjne zapewniają bardziej realistyczne testy tworzących modeli.
- Możliwość generowania nowych scenariuszy testowych przez użytkowników symulatora
- Zbiór danych treningowych - dane zebrane z ponad 8 godzin jazdy. Mają rozmiar około 100 GB. Są to m. in. obrazy z kamer, kąt skrętu kierownicy czy wartości pedałów gazu i hamulca. Dane zostały wstępnie przetworzone, w celu zwiększenia efektywności treningu.

Rysunek 1.6 przedstawia schemat reprezentujący architekturę systemu. Na jego podstawie można wysnuć ciekawe wnioski na temat sposobu funkcjonowania symulatora.Więcej informacji na temat instalacji systemu oraz jego użytkowania można odnaleźć na stronie repozytorium projektu [3].

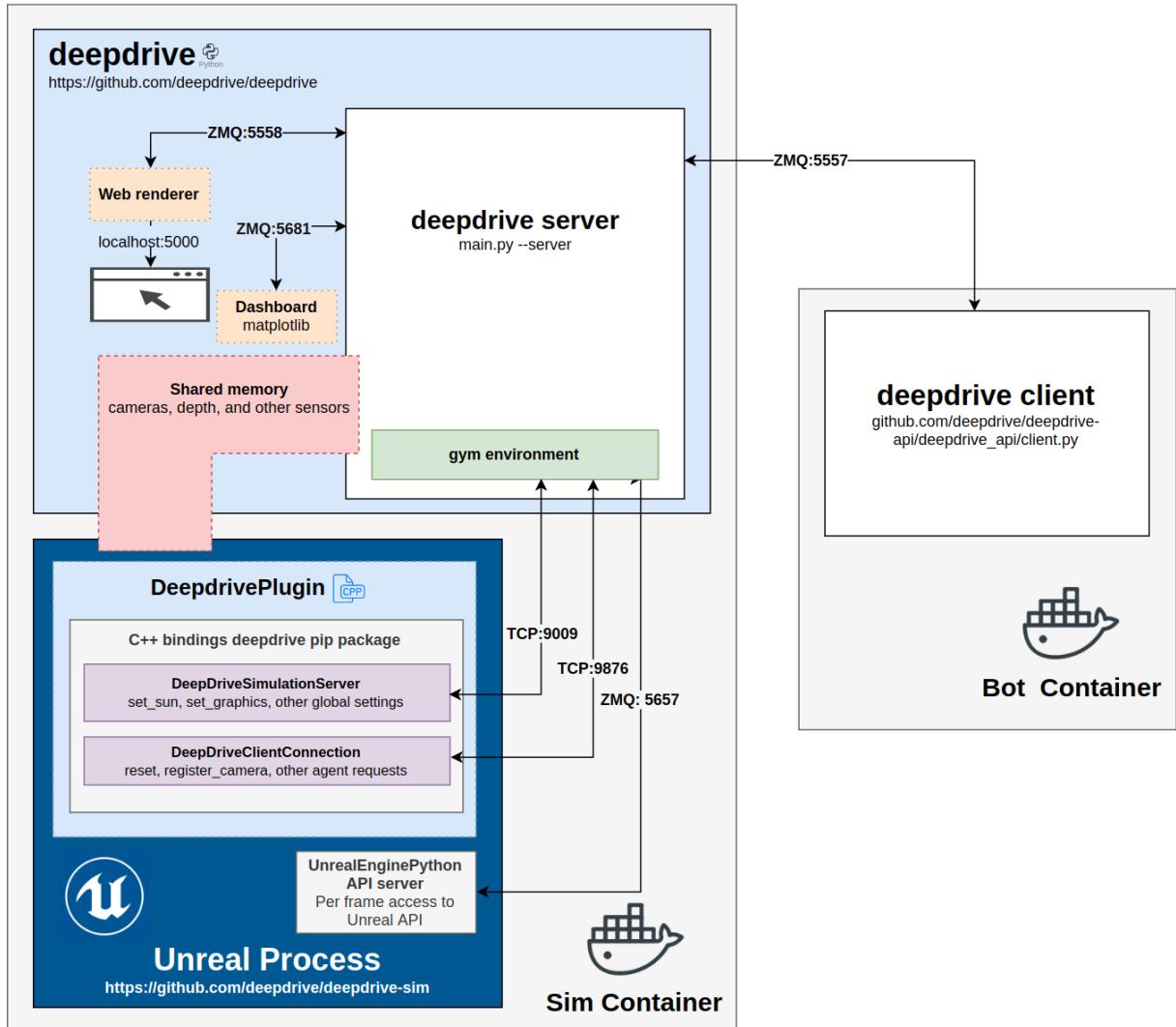
1.9 Projekty modelarskie

Projekty modelarskie charakteryzują się tym, iż tworzone są z myślą o modelach pojazdów wykonanych w pewnej skali. Takie projekty cieszą się dość dużą popularnością, a wynika to z kilku faktów:

1. Modele są o wiele tańsze niż prawdziwe samochody, a oferują zalety niedostępne dla symulatorów. Symulatory, pomimo tego że posiadają potężne możliwości, nie są w stanie dokładnie odwzorować wszystkich zjawisk występujących w rzeczywistym świecie.
2. Modele samochodów posiadają zazwyczaj prostą i modularną konstrukcję, co pozwala na stosunkowo szybkie prototypowanie wymyślnych rozwiązań.
3. Niektórom osobom łatwiej pracować z modelem niż symulatorem. Dzieje się tak w przypadku, gdy takie osoby nie posiadają komputera o niezbędnej mocy obliczeniowej (symulatory są dość wymagające pod tym kątem), a dysponują podstawową wie-

dzą z zakresu elektroniki i budżetem wystarczającym do zakupu niedrogich układów scalonych (typu *Raspberry Pi* lub *Arduino Uno*).

Deepdrive System Architecture



Rysunek 1.6: Architektura systemu Deepdrive

1.9.1 donkeycar

Donkey Car jest otwartoźródłową platformą typu DIY (*Do It Yourself*), przeznaczoną do tworzenia systemów autonomicznych dla samochodów wykonanych w małej skali. Jest bardzo popularna, wielu ludzi korzysta z niej podczas przygotowań do zawodów *DIY Robocars* [4]. Można zakupić gotowy zestaw (za około 200 dolarów) lub zbudować go samodzielnie.



Rysunek 1.7: XiaoR GEEK XR-F2 - przykład gotowego do zakupu zestawu dla platformy Donkey Car

W skład zestawu musi wchodzić:

1. Autko RC (pochodzące z listy wspieranych autek [7] lub zbudowane według wytycznych [6])
2. Raspberry Pi [11]
3. Kamerka
4. Oprogramowanie wykorzystujące bibliotekę donkeycar.

Jest to minimalny zestaw, wymagany do poprawnego funkcjonowania platformy. Oczywiście można go rozszerzać o dodatkowe komponenty, takie jak sensory lub inne urządzenia pomiarowe.

Standardowym samochodem, od którego wiele osób rozpoczyna przygodę z platformą Donkey Car, jest *Donkey*. W oficjalnej dokumentacji projektu [8] znajdują się wszystkie informacje, niezbędne do samodzielnego zbudowania samochodu.

Sercem platformy jest biblioteka donkeycar, napisana w języku Python. Przeznaczona jest do rozwoju samochodów autonomicznych, jej cechami są modularność i minimalizm.

Wykorzystując tę bibliotekę, programowanie samochodu jest bardzo łatwe. Oto przykład z oficjalnej dokumentacji, w którym samochód po rozpoczęciu pętli jazdy wykonuje zdjęcia i zapisuje je do wbudowanej pamięci:

```
import donkey as dk

#initialize the vehicle
V = dk.Vehicle()

#add a camera part
cam = dk.parts.PiCamera()
V.add(cam, outputs=['image'], threaded=True)

#add tub part to record images
tub = dk.parts.Tub(path='~/mycar/data',
                    inputs=['image'],
                    types=['image_array'])
V.add(tub, inputs=inputs)

#start the vehicle's drive loop
V.start(max_loop_count=100)
```

Kod źródłowy biblioteki dostępny jest na stronie repozytorium projektu [9]. Ciekawy przykład zastosowania platformy Donkey Car został opisany w artykule [88].

1.10 Małe prywatne projekty

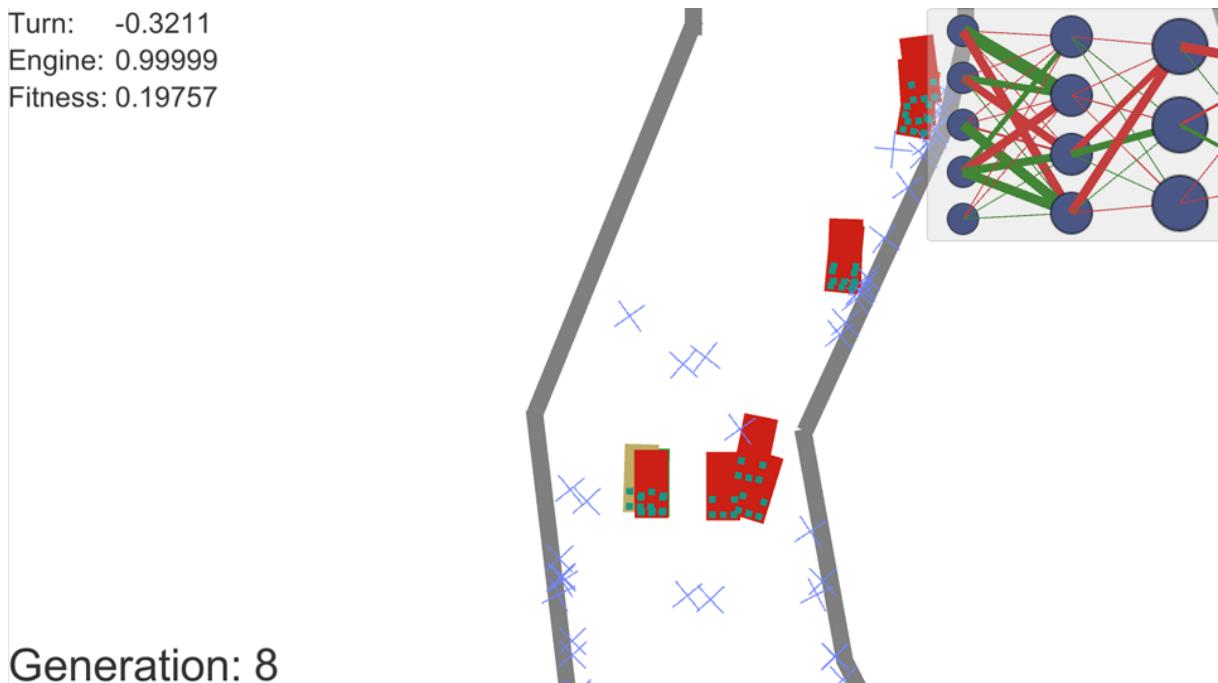
Należą do ostatniej z czterech sklasyfikowanych przeze mnie grup. Cechami charakterystycznymi takich projektów są: niski budżet oraz małe zespoły tworzące (często są to pojedyncze osoby). Zazwyczaj tworzone w celach edukacyjnych - twórca chciał się podszkolić z danego zagadnienia.

1.10.1 Deep Learning Cars

Autorem projektu jest austriacki programista Samuel Arzt [21] [20].

Opis implementacji

Na początku każdej generacji jest tworzonych 20 samochodów (osobników). Każdy samochód posiada własną sieć neuronową oraz zestaw 5 czujników, odmierzających odległość



Rysunek 1.8: Stopklatka z pracy systemu

z przodu samochodu. Wykorzystaną architekturą sieci jest prosta sieć jednokierunkowa (z ang. *feed-forward*) o wymiarach 5-4-3-2, gdzie każdy wymiar to liczba neuronów w danej warstwie sieci. Zatem sieci miały po 5 neuronów w warstwie wejściowej, 4 i 3 neurony w warstwach ukrytych oraz 2 neurony w warstwie wyjściowej.

Sygnały z czujników wysyłane są do neuronów warstwy wejściowej. Czujniki te służą do wykrywania przeszkód w otoczeniu samochodu. Każdy czujnik jest skierowany w inną stronę. Czujniki tworzą „pole widzenia” samochodu o łącznym kącie 90 stopni.

Celem każdego samochodu jest bezkolizyjny przejazd po wyznaczonym torze. Jedyne operacje jakie może wykonać samochód, to skręt w lewo lub prawo. Prędkość wszystkich samochodów jest stała i ustalana przed rozpoczęciem generacji. W chwili kolizji z przeszkodą, samochód jest usuwany. Należy dodać, że nie występują kolizje pomiędzy samochodami.

Kiedy wszystkie samochody zostaną usunięte, bieżąca generacja się kończy. Osobniki do kolejnej generacji są tworzone przy pomocy algorytmu ewolucyjnego. Do reprodukcji wybierane są 2 osobniki z najwyższym przystosowaniem. Przystosowanie jest obliczane na podstawie odległości przebytej przez samochód. Im większa przejechana odległość, tym wyższa wartość przystosowania samochodu. Geny dwóch najlepiej przystosowanych osobników są krzyżowane i mutowane, tworząc 20 nowych osobników (potomków). Genami samochodu

są wagi w jego sieci neuronowej. Uczenie sieci (dostrajanie wartości wag) odbywa się za pomocą algorytmu genetycznego [56].

Reprezentacja graficzna

Dwuwymiarowa symulacja środowiska (toru) została wykonana przy użyciu silnika Unity. Na rysunku 1.8 widać interfejs aplikacji. Można zauważyć, że krzyżyki wyświetlane na ekranie są reprezentacją czujników samochodów. Kamera podąża za samochodem z najlepszym przystosowaniem. Dwa samochody z najlepszym przystosowaniem są wyróżniane kolorem.

Dostępność kodu źródłowego

Cały kod źródłowy projektu został udostępniony przez autora [20].

Rozdział 2

Wstęp do uczenia maszynowego

Uczenie maszynowe jest przężnie rozwijającą się dziedziną informatyki [87]. Organizacje z całego świata angażują się w pracę nad projektami z tego obszaru.

Temat uczenia maszynowego jest zbyt szeroki, żeby móc go streścić w krótkim rozdziale. Zatem w poniższych sekcjach skupię się na opisaniu tylko tych zagadnień, które są niezbędne do zrozumienia dalszych rozdziałów.

2.1 Sieci neuronowe

Sieci neuronowe to jeden z najważniejszych tematów dotyczących uczenia maszynowego. Poznanie podstawowych zagadnień teoretycznych stojących za tym pojęciem jest koniecznym warunkiem zrozumienia uczenia maszynowego w ogóle. Poniżej postaram się po krótce przedstawić najważniejsze aspekty tego tematu.

Sztuczna sieć neuronowa jest matematycznym modelem przetwarzania informacji. Jest inspirowana funkcjonowaniem biologicznego układu nerwowego. Sieci neuronowe są wykorzystywane do równoległego przetwarzania nieliniowych relacji, zachodzących pomiędzy danymi wejściowymi a oczekiwany wyjściem z sieci [30].

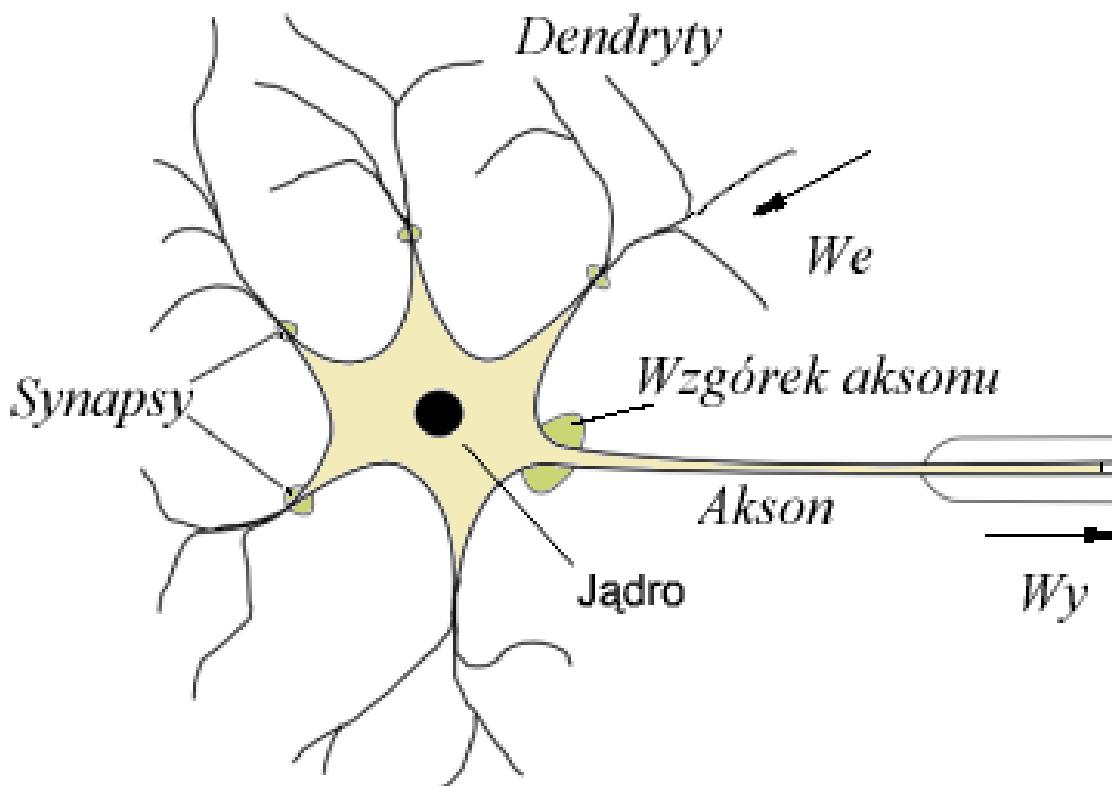
Sieci neuronowe są wykorzystywane do rozwiązywania problemów, z którymi nie radzą sobie tradycyjne modele obliczeniowe. Najważniejszą cechą sieci neuronowych jest ich zdolność uczenia się w oparciu o dostarczany im zbiór danych treningowych.

2.1.1 Neuron biologiczny

Aby lepiej zrozumieć strukturę sztucznego neuronu, należy zapoznać się z modelem neuronu biologicznego. Został on przedstawiony na rysunku 2.1. Najważniejsze elementy, na które należy zwrócić uwagę na rysunku, to:

- Jądro – centrum obliczeniowe neuronu.
- Dendryty – wejścia neuronu. Przesyłają do jądra sygnały poddawane późniejszej obróbce.

- Synapsa – łączy dendryt z jądrem. W synapsie sygnał wejściowy może ulegać wstępnej modyfikacji, to znaczy być wzmacniany lub osłabiany.
- Wzgórek aksonu – łączy jądro z aksonem.
- Akson – wyjście neuronu. Zazwyczaj rozgałęzia się, przesyłając sygnał do wielu kolejnych neuronów.

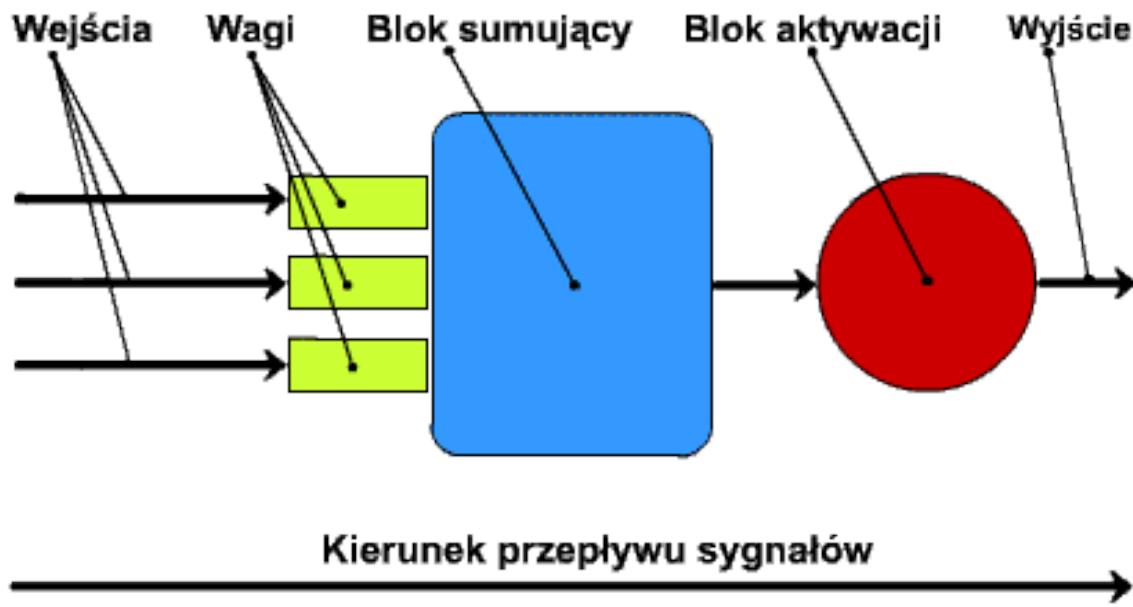


Rysunek 2.1: Model biologicznego neuronu

2.1.2 Sztuczny neuron

Sztuczny neuron jest matematycznym modelem biologicznego neuronu. To podstawowy budulec sieci neuronowej. Każdy neuron może posiadać wiele wejść i tylko jedno wyjście [30]. Dane przetwarzane przez neuron to najczęściej liczby rzeczywiste, miesiące się w zadanym zakresie. Wszystkie wejścia neuronów mają przypisane wagę, czyli wartości liczbowe określające jak ważne jest dane wejście dla neuronu.

Porównując budowę sztucznego neuronu z neuronem biologicznym, można odnaleźć wiele analogii. Model sztucznego neuronu został przedstawiony na rysunku 2.2 [51]:



Rysunek 2.2: Model sztucznego neuronu

Na wejścia neuronu podawane są sygnały wejściowe. Każdy sygnał wejściowy jest przemnażany przez odpowiadającą mu wagę. Przemnożone sygnały wejściowe są następnie sumowane. Sumowanie następuje w bloku sumującym. Uzyskaną wartość nazywamy potencjałem membranowym. Potencjał membranowy jest przekazywany do funkcji aktywacji, która na jego podstawie oblicza wartość podawaną na wyjście neuronu. Zachowanie neuronu jest silnie uzależnione od rodzaju wykorzystywanej funkcji aktywacji.

2.1.3 Funkcje aktywacji

Funkcja aktywacji jest jednym z najważniejszych elementów wpływających na zachowanie sieci neuronowej, a także na efektywność jej uczenia. Istnieje bardzo wiele rodzajów funkcji aktywacji. Każda z nich posiada własną charakterystykę, która określa do jakich problemów powinna być stosowana. Jeśli znamy charakterystykę problemu rozwiązywanego przez sieć neuronową, to możemy dobrać takie funkcje aktywacji, które przyspieszą proces nauki tej sieci. Co warto podkreślić, dana sieć neuronowa może korzystać z więcej niż jednej funkcji aktywacji. Zazwyczaj w takich wypadkach funkcje aktywacji pogrupowane są warstwami sieci.

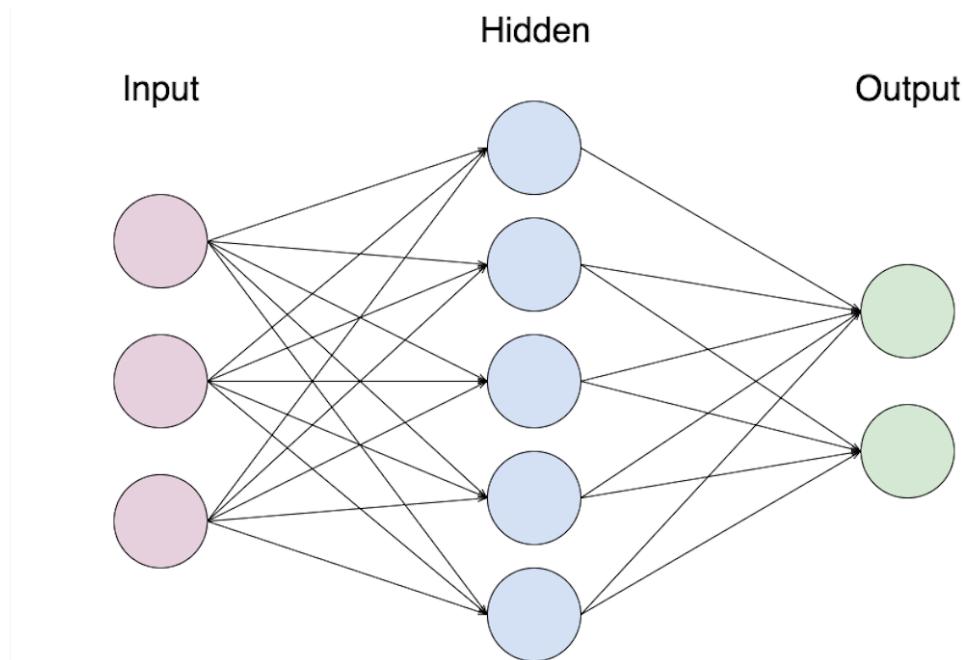
Szczegółowe informacje na temat najczęściej wykorzystywanych funkcji aktywacji, wraz z opisem ich charakterystyki, zostały podane w artykułach [47] [42] [41].

2.1.4 Warstwy sieci neuronowych

Sieci neuronowe zazwyczaj składają się z wielu warstw. Neurony należące do jednej warstwy nie są zwykle ze sobą połączone. Warstwy sieci neuronowej dzieli się na trzy rodzaje [30]:

- warstwa wejściowa – składa się z neuronów pobierających zestaw danych wejściowych,
- warstwa wyjściowa – składa się z neuronów generujących wynik obliczeń sieci neuronowej,
- warstwy ukryte – znajdują się pomiędzy warstwą wejściową a warstwą wyjściową.

Na rysunku 2.3 jest zaprezentowany przykład prostej sieci neuronowej, składającej się z tylko trzech warstw:



Rysunek 2.3: Przykład prostej sieci neuronowej

2.1.5 Klasyfikacja sieci neuronowych

Sieci neuronowe są wykorzystywane do rozwiązywania nietrywialnych problemów dotyczących życia codziennego. Problemy te są względem siebie bardzo zróżnicowane, więc żeby sieć neuronowa mogła radzić sobie z zadanym problemem w sposób optymalny, również musi być jak najbardziej do tego problemu przystosowana.

W efekcie, istnieje bardzo wiele architektur sieci neuronowych. Każda architektura posiada swoją specyfikę i została zaprojektowana do rozwiązywania konkretnej klasy problemów. Więcej na ten temat można przeczytać w doskonałym artykule zatytułowanym „*The Neural Network Zoo*” [83].

2.1.6 Cechy sieci neuronowych

Jak wykazano w pracy [32], najważniejszymi cechami sieci neuronowych są:

1. Własność uniwersalnego aproksymatora (sieci potrafią aproksymować dowolną funkcję, z dowolnie małym błędem).
2. Pozyskiwanie wiedzy z danych.
3. Duża odporność na zakłócenia danych.
4. Równoległa architektura (równoległe przetwarzanie danych).
5. Adaptacyjność względem otrzymywanych danych.
6. Zdolność do samoorganizacji, czyli samodzielnego dostrajania swoich parametrów, w celu lepszego dostosowania się do wykonywanych zadań.

2.1.7 Zalety i wady sieci neuronowych

Według artykułu [58] można wyszczególnić następujące wady i zalety sieci neuronowych:

1. Zalety

- (a) Zdolność samodzielnego uczenia się – sieć uczy się na podstawie dostarczanych jej przykładów.
- (b) Tolerancja błędu – sieć może poprawnie funkcjonować, nawet jeśli część połączeń zostanie uszkodzona. Sieć będzie generować prawidłowe wyniki do czasu, aż liczba uszkodzonych połączeń nie przekroczy „*masy krytycznej*”.
- (c) Zdolność pracy z niekompletną wiedzą – po wytrenowaniu sieci, sieć jest w stanie generować wyniki nawet dla danych wejściowych które są niekompletne. Działalność sieci jest tutaj zależna od stopnia ważności brakujących danych.
- (d) Zdolność równoległego przetwarzania danych.

2. Wady

- (a) Zależność sprzętowa – sztuczne sieci neuronowe wymagają procesorów pozwalających na wykonywanie równoległych obliczeń. Największa wydajność jest

osiągana wtedy, gdy architektura sprzętowa odpowiada architekturze sieci. Z tego wynika, że wydajność sieci neuronowej zależy od architektury sprzętu.

- (b) Brak zrozumienia wnętrza sieci – większość sieci neuronowych musimy traktować jako czarną skrzynkę. Potrafimy je wytrenować, ale nie wiemy jakie zależności (zachodzące wewnętrz sieci) umożliwiają jej prawidłowe funkcjonowanie. Brak tego zrozumienia ma wiele negatywnych konsekwencji, m.in. utrudnia to testowanie tworzonych sieci.
- (c) Trudność projektowania sieci – nie istnieją żadne formalne zasady określające sposób projektowania sieci neuronowej pod wykonywanie danego zadania. Jeśli istnieją już jakieś rozwiązania, to można się nimi inspirować. W przeciwnym wypadku, jedynym sposobem projektowania sieci o optymalnej architekturze jest metoda prób i błędów. Ponieważ poprawne funkcjonowanie sieci jest zależne od wielu jej parametrów, to debugowanie takiej sieci jest bardzo trudnym zadaniem. Dlatego etap projektowania sieci zwykle zajmuje sporo czasu.
- (d) Trudność w reprezentacji problemu – sieci neuronowe operują na danych numerycznych. Niestety, wiele problemów rozwiązywanych przez sieci neuronowe mają zupełnie inną postać. Mogą to być obrazy, słowa lub dźwięki. W takim wypadku, należy w odpowiedni sposób zakodować informacje do postaci numerycznej. Nie jest to łatwe zadanie. Od jakości takiego kodowania może zależeć skuteczność uczenia sieci. Wartości zwracane przez sieć należy z powrotem zdekodować do właściwej postaci, tak aby wyniki mogły być zinterpretowane przez człowieka.

2.1.8 Porównanie z tradycyjnymi metodami programowania

Tradycyjne metody programowania opierają się na ręcznej implementacji całego programu. Każdy fragment algorytmu musi zostać określony i zapisany przez programistę. Dla danego zestawu danych wejściowych, program zawsze będzie wykonywał tę samą sekwencję instrukcji. Powinien również zwracać ten sam wynik.

Przy takim podejściu, użytkownik na podstawie zisanego programu oraz posiadanego zbioru danych wejściowych otrzymuje wyniki generowane przez program. Rolą użytkownika jest określenie, czy otrzymane dane są prawidłowe.

Natomiast Uczenie Maszynowe jest zupełnie inne. Polega ono na *wytwarzaniu przez maszynę algorytmu rozwiązania na podstawie dostarczanego jej zbioru danych treningowych*. Zbiór danych treningowych to najczęściej zestawy danych wejściowych wraz z odpowiadającymi im wynikami.

jącymi im zestawami oczekiwanych danych wyjściowych.

Sieć uczy się poprzez wykonanie pełnego przejścia sieci i porównania otrzymanych wyników z wynikami oczekiwanyimi. Na podstawie tej różnicy obliczana jest wielkość błędu. Znając wielkość błędu, parametry sieci mogą być dostrojone przy użyciu odpowiednich algorytmów uczących. Uczenie sieci trwa aż do wystąpienia jednego z warunków stopu. Warunkiem stopu może być np. osiągnięcie przez sieć wystarczającej dokładności wyników. Więcej informacji na temat różnic pomiędzy dwoma powyżej opisanymi podejściami znajduje się w artykule [50].

2.2 Typy uczenia maszynowego

Biorąc pod uwagę różne strategie obierane podczas treningu sieci neuronowych, każdy trening można przyporządkować do jednego z czterech typów uczenia maszynowego.

Te typy to:

1. Uczenie nadzorowane (*Supervised Learning*)
2. Uczenie pół-nadzorowane (*Semi-supervised Learning*)
3. Uczenie nienadzorowane (*Unsupervised Learning*)
4. Uczenie ze wzmacnieniem (*Reinforcement Learning*)

Każdy typ uczenia maszynowego posiada własne cechy charakterystyczne. Więcej informacji na ten temat znajduje się w artykule [36]. W przypadku mojej aplikacji, wykorzystywany typem uczenia jest uczenie ze wzmacnieniem.

2.3 Algorytmy ewolucyjne

Algorytmy ewolucyjne wywodzą się z teorii ewolucji, która zakłada że najlepiej przystosowane osobniki populacji tworzą najwięcej potomstwa. Potomstwo z kolei przejmuje najlepsze cechy swoich rodziców, co sprawia że z generacji na generację cała populacja staje się coraz lepiej przystosowana do określonych warunków. Takie zjawisko jest możliwe dzięki istnieniu w naturze różnych mechanizmów, takich jak **mutacja**, **krzyżowanie** i **selekcja** [71].

Algorytmy ewolucyjne to algorytmy optymalizacji czarnej skrzynki, czyli optymalizacji bez obliczania pochodnych. Optymalizacja czarnej skrzynki polega na znajdowaniu globalnego optimum funkcji $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ w sytuacji, gdy nie znamy jej postaci analitycznej i pochodne nie mogą być obliczone [71]. Dzieje się tak, gdy funkcja jest bardzo skomplikowana lub wymaga użycia zewnętrznego oprogramowania, jak np. środowiska symulacyjnego

(patrz sekcja 1.8). W świecie rzeczywistym są to bardzo częste przypadki, dlatego też algorytmy ewolucyjne cieszą się dość dużą popularnością.

Poniżej zamieszczam opis dwóch algorytmów ewolucyjnych, z których korzystałem podczas implementacji aplikacji. Algorytmy te posłużyły mi przy dostrajaniu parametrów sieci (wag i biasów). Więcej informacji na ten temat znajduje się w sekcji 4.2.4.

2.3.1 Ewolucja Różnicowa

Jego angielska nazwa to *Differential Evolution* (DE). Po raz pierwszy został zaproponowany w artykule z 1997 roku, którego autorami byli Rainer Storn oraz Kenneth Price [75]. Jest bardzo popularnym algorytmem ewolucyjnym. Na przykład Europejska Agencja Kosmiczna (ESA) wykorzystuje Ewolucję Różnicową do projektowania optymalnych trajektorii lotu potrzebnych do wejścia na orbitę planety przy użyciu możliwie najmniejszej ilości paliwa [52].

W Ewolucji Różnicowej, rozwiązania są reprezentowane jako osobniki należące do populacji. Każdy osobnik jest wektorem liczb rzeczywistych. Te liczby rzeczywiste to parametry funkcji, dla której szukamy globalnego optimum. Funkcja zaś określa, jak dobrze przystosowany jest dany osobnik z populacji.

Algorytm można opisać za pomocą poniższej listy kroków:

1. Inicjalizacja

Tworzenie populacji osobników. Najczęściej osobniki tworzone są w sposób losowy.

2. Sprawdzenie warunku stopu

Algorytm powinien mieć jasno zdefiniowane warunki, dla których jego wykonanie zostanie przerwane. Przykładem warunku stopu może być odnalezienie globalnego optimum lub osiągnięcie maksymalnej liczby dozwolonych iteracji.

3. Dla każdego osobnika x w populacji:

(a) Ocena przystosowania

Osobnik jest oceniany za pomocą funkcji przystosowania. Wartości przystosowań są zapisywane w odpowiedniej strukturze danych. Ta struktura jest potrzebna w dalszych krokach algorytmu.

(b) Mutacja i krzyżowanie

Należy wybrać z populacji trzech osobników a, b, c którzy są różni od siebie oraz różni od osobnika x . Następnie z osobników a, b, c tworzy się tzw. „*mutant vector*”. Jest to osiągane poprzez obliczanie różnicy pomiędzy wektorami b, c , przemnożeniu jej przez stałą zwaną „współczynnikiem mutacji” (*mutation*

factor) i dodaniu przemnożonej różnicy do wektora a . W efekcie tych obliczeń otrzymujemy tzw. „rozwiązywanie kandydackie”.

(c) Wymiana wektorów

Rozwiązywanie kandydackie, otrzymane w poprzednim kroku, jest oceniane za pomocą funkcji przystosowania. Jeśli te rozwiązanie jest lepiej przystosowane od osobnika x , to wektory są podmieniane.

4. Powrót do punktu 2.

Opisany powyżej algorytm to podstawowa wersja Ewolucji Różnicowej. Z takiej wersji korzystam w swojej implementacji. Sprawdza się ona znakomicie. Istnieje jednak wiele innych wersji tego algorytmu. Część z nich została opisana w artykule [71].

2.3.2 Optymalizacja Roju Cząstek

Jego angielska nazwa to *Particle Swarm Optimization* (PSO). Został zaproponowany przez Jamesa Kennedy'ego i Russella Eberharta w artykule z 1995 roku [49].

Jak wynika z artykułu [55], jest to algorytm populacyjny, inspirowany naturalnym zachowaniem zwierząt stadnych. Należy do grupy algorytmów inteligencji rojowej (rozproszonej), do której klasyfikowany jest także m.in. algorytm mrowiskowy (**Ant Colony Algorithm**) oraz algorytm żerowania bakterii (**Bacterial Foraging Optimization Algorithm**).

Podstawowymi pojęciami algorytmu PSO są Populacja (Rój) oraz Osobnik (Cząstka). Cząstka reprezentuje pojedyncze rozwiązanie w n-wymiarowej przestrzeni rozwiązań. Każda cząstka posiada pozycję oraz prędkość. Są to n-wymiarowe wektory liczb rzeczywistych. Pozycja (*position*) określa bieżącą pozycję cząstki w przestrzeni rozwiązań. Prędkość (*velocity*) określa kierunek cząstki oraz odległość, jaką pokona w kolejnej iteracji algorytmu. Nowa pozycja cząstki jest obliczana poprzez dodanie bieżącej prędkości cząstki do starej pozycji.

Populacja to zbiór cząstek. Populacja przemieszcza się po wirtualnej przestrzeni rozwiązań, wykorzystując kooperację cząstek w celu odnalezienia globalnego optimum. Populacja wraz z upływem kolejnych iteracji będzie zbiegać do najlepszego rozwiązania globalnego.

Warto wspomnieć, że algorytm PSO nie wykorzystuje Spadku Gradientu, zatem może on być wykorzystywany do optymalizacji funkcji, które nie mogą być różniczkowane.

Pseudokod algorytmu PSO przedstawia się w sposób następujący:

1. Inicjalizacja populacji

Utwórz n cząstek. Dla każdej cząstki wyznacz wektory pozycji i prędkości, które są inicjalizowane liczbami losowymi z zadanego przedziału.

2. Dla każdej cząstki i :

- (a) Ocena wartości przystosowania cząstki
- (b) Jeśli przystosowanie jest lepsze od najlepszego przystosowania i -tej cząstki, to przypisz bieżącą pozycję cząstki do zmiennej p_i . Jest to zmienna przechowująca najlepszą historycznie pozycję i -tej cząstki.
- (c) Jeśli przystosowanie jest lepsze od najlepszego globalnie przystosowania, to przypisz bieżącą pozycję cząstki do zmiennej p_g . Jest to zmienna przechowująca najlepszą pozycję uzyskaną historycznie przez jakąkolwiek cząstkę z populacji.

3. Sprawdzenie warunku stopu

Przerwij algorytm, jeśli wystąpi warunek stopu. Warunkiem stopu może być np. znalezienie globalnego optimum lub osiągnięcie maksymalnej liczby dozwolonych iteracji.

4. Dla każdej cząstki i :

- (a) Oblicz nową prędkość i -tej cząstki ze wzoru:

$$v_i(k+1) = w v_i(k) + c_1 rand_1(p_i - x_i) + c_2 rand_2(p_g - x_i)$$

Objaśnienia symboli:

x_i – bieżąca pozycja i -tej cząstki

$v_i(k+1)$ – nowa prędkość cząstki (prędkość cząstki w $k+1$ -tej iteracji)

w – parametr bezwładnościowy (*inertial parameter*). Wskazuje na ważność bieżącej prędkości przy obliczeniu nowej prędkości.

c_1, c_2 – współczynniki przyspieszenia. c_1 wskazuje na ważność najlepszej wartości cząstki (p_i), natomiast c_2 na ważność najlepszej globalnej wartości dla całej populacji (p_g).

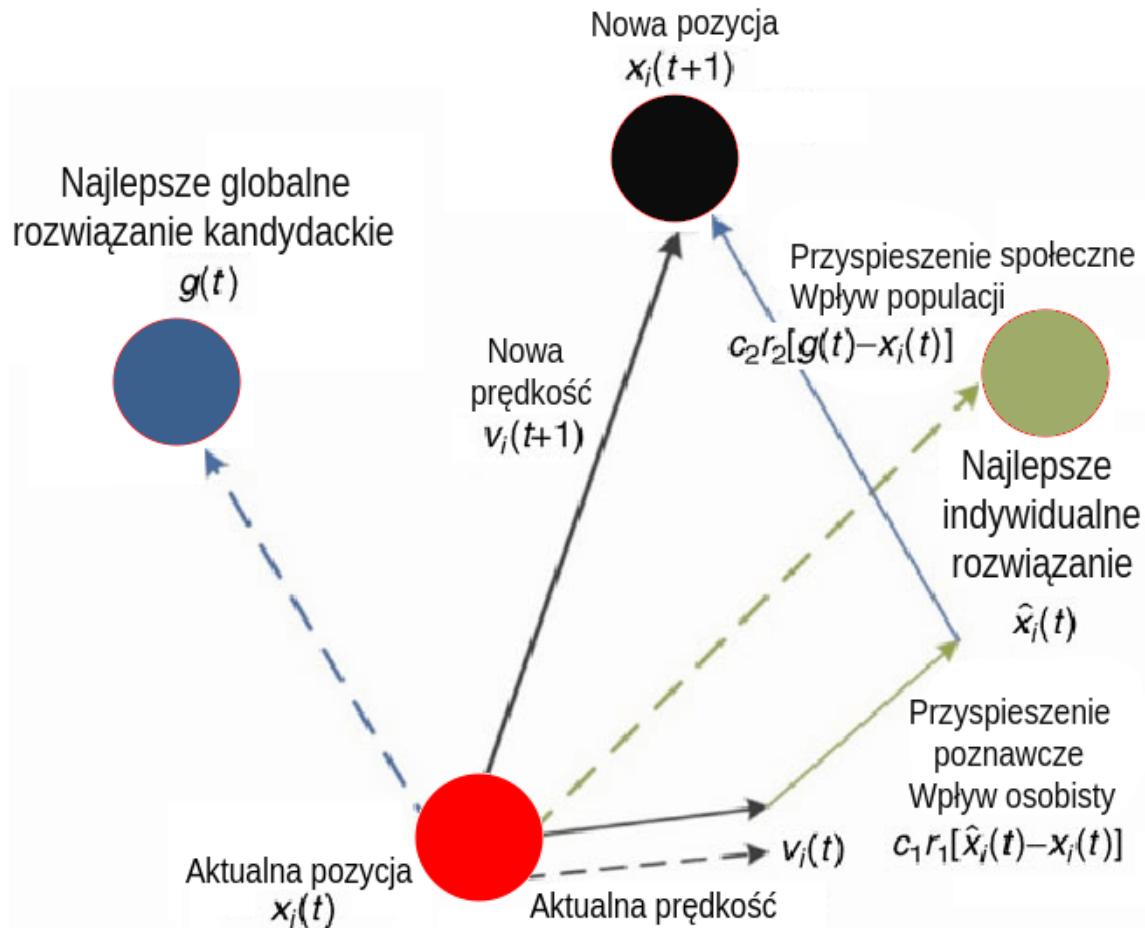
$rand_1, rand_2$ – liczby losowe z przedziału $[0 ; 1]$.

- (b) Wyznacz nową pozycję i -tej cząstki:

$$x_i(k+1) = x_i(k) + v_i(k+1)$$

5. Powrót do punktu 2.

Rysunek 2.4 przedstawia wizualizację procesu obliczania nowej pozycji cząstki:



Rysunek 2.4: Obliczanie nowej pozycji cząstki

Rozdział 3

Projekt systemu i opis narzędzi

W niniejszym rozdziale zostaną przedstawione najważniejsze założenia i decyzje projektowe, jakie zostały przyjęte przed rozpoczęciem prac nad aplikacją. Ponadto zamieszczam opis narzędzi, wykorzystanych podczas procesu implementacji systemu.

3.1 Projekt systemu

Wytwarzanie oprogramowania nie należy do zadań trywialnych. Dotyczy to zwłaszcza prac nad skomplikowanymi systemami informatycznymi. Podstawą sukcesu jest jasne zdefiniowanie założeń projektowych oraz odpowiednie zaplanowanie architektury systemu. Ten etap pracy najlepiej wykonać w początkowych fazach projektu, gdy podejmowanie kluczowych decyzji projektowych nie jest jeszcze obarczone wysokimi kosztami.

Podczas fazy projektowania aplikacji postanowiłem rozpatrzyć trzy zagadnienia, które moim zdaniem są najważniejsze na tym etapie prac. Te zagadnienia to:

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

Każde z tych zagadnień zostało omówione poniżej.

3.1.1 Definicja rozwiązywanego problemu

Problemem rozwiązywanym przez aplikację jest trening agentów. Agenci sterują samochodami wymodelowanymi w środowisku symulacji. Samochody mają pokonywać wyznaczone tory wyścigowe bez powodowania kolizji ze ścianami. Agenci odbierają informacje o swoim bieżącym położeniu dzięki czujnikom odległości, umocowanym na każdym samochodzie. W odpowiedzi na te informacje, agenci wysyłają do środowiska symulacji żądania wykonania określonych akcji, np. zmiany kąta kierownicy w samochodzie kontrolowanym przez agenta. Oprócz zjawisk opisanych powyżej, środowisko symulacji zlicza wartości przystosowania agentów i zwraca listę tych wartości (po jednej dla każdego agenta) po każ-

dym zakończonym epizodzie symulacji. Wartości przystosowania są bardzo ważne, gdyż bez nich nie dałoby się przeprowadzić procesu uczenia agentów.

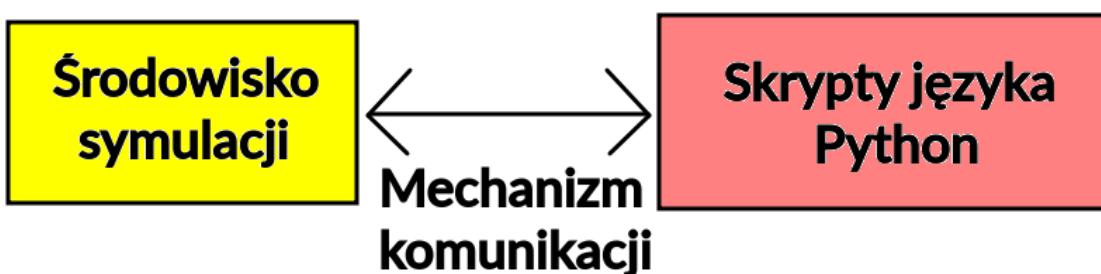
Agentami są jednokierunkowe sieci neuronowe [76] o prostej topologii (domyślnie składają się z trzech warstw o wymiarach 3-5-2). Parametry sieci neuronowych (wagi i biasy) są dostrajane przy użyciu dwóch algorytmów ewolucyjnych: **Ewolucji Różnicowej** (patrz sekcja 2.3.1) oraz **PSO** (patrz sekcja 2.3.2).

3.1.2 Cele i założenia

Każda aplikacja powinna realizować jakieś cele i założenia. Oto lista celów i założeń, jakie przyjąłem dla swojej aplikacji:

1. Aplikacja powinna być prosta w obsłudze;
2. Aplikacja powinna działać na systemach Linux oraz Windows;
3. Aplikacja powinna zapisywać wytrenowane modele do pliku o ustalonym formacie.
Plik ten powinien być później możliwy do odczytu przez aplikację;
4. Dane konfiguracyjne nie powinny być na sztywno zaszyte w kodzie, lecz dostarczane z zewnętrznego pliku o ustalonym formacie;
5. Aplikacja powinna dokumentować swój przebieg w plikach dziennika. Pliki dziennika powinny zawierać wszelkie istotne informacje, które mogą być pomocne przy późniejszym debugowaniu;
6. Eksperymenty obliczeniowe powinny się wykonywać przy minimalnym wysiłku użytkownika. Efektem działania tych eksperymentów powinny być wykresy oraz inne dane, które będzie można zaprezentować przy omawianiu wyników eksperymentów obliczeniowych.

3.1.3 Architektura systemu



Rysunek 3.1: Wizualizacja architektury systemu

Architektura systemu jest stosunkowo prosta. Składa się z dwóch zasadniczych komponentów: **środowiska symulacji** oraz **zestawu skryptów języka Python**. Komponenty komunikują się ze sobą poprzez odpowiedni mechanizm komunikacji. Rysunek 3.1 przedstawia wizualizację omawianej architektury. Więcej informacji na temat poszczególnych komponentów aplikacji można odnaleźć w rozdziale 4-tym. Rozdział ten jest poświęcony opisowi implementacji systemu.

3.2 Zastosowane technologie

W niniejszej sekcji opisuję technologie, z których korzystałem podczas implementacji mojej aplikacji.

3.2.1 Unity

Wieloplatformowy silnik gier 2D lub 3D [53]. Pozwala również na tworzenie innych materiałów interaktywnych - wizualizacje, animacje itp. Silnik został napisany w językach C/C++ (platforma uruchomieniowa) oraz C# (Unity API). Skrypty dla silnika pisze się w języku C#. Gry tworzone na silniku Unity mogą obsługiwać wiele platform sprzętowych i systemów operacyjnych. Platformy wspierane przez wersję 2019.1 [82]:

1. Komputery osobiste (PC):
 - (a) Wspierane systemy operacyjne:
 - Windows,
 - Linux,
 - Mac OS X;
 - (b) Wspierane architektury sprzętowe:
 - x86 - procesor 32-bitowy,
 - x64 - procesor 64-bitowy,
 - Universal - wszystkie procesory,
 - x86 + x86_64 (Universal) - wszystkie procesory (Linux);
2. iOS;
3. Android;
4. WebGL;
5. Samsung TV;
6. Xiaomi;
7. i wiele innych.

Z silnikiem Unity są również kompatybilne hełmy rzeczywistości wirtualnej, takie jak Oculus Rift oraz Gear VR.

Do wersji 4.6 silnik był udostępniany na licencji płatnej lub darmowej zawierającej ograniczoną funkcjonalność, ale wraz z premierą Unity 5 prawie wszystkie funkcje silnika udostępniono w wersji darmowej dla twórców nieprzekraczających 100 tysięcy dolarów dochodów rocznie.

Unity oferuje również tzw. Asset Store, który umożliwia skorzystanie z płatnych lub darmowych komponentów takich jak tekstury lub skrypty. Silnik Unity został przeze mnie wykorzystany do stworzenia środowiska symulacji, niezbędnego w procesie treningu intelligentnych agentów.

3.2.2 Języki programowania

Język programowania jest jednym z najważniejszych narzędzi w rękach każdego programisty. Pozwala on na tworzenie oprogramowania w formie zrozumiałej dla człowieka i wykonywalnej dla maszyny. Wykonanie kodu źródłowego jest możliwe po przetłumaczeniu go do postaci kodu maszynowego, czyli sekwencji rozkazów zrozumiałych dla procesora. Tłumaczeniem kodu źródłowego na język maszyny zajmują się specjalne programy, zwane *kompilatorami* lub *interpreterami*.

Poniżej znajduje się krótki opis języków, które wykorzystałem podczas tworzenia aplikacji.

C#

Język programowania stworzony przez firmę Microsoft [13]. Najważniejsze cechy:

- silne, statyczne typowanie;
- obiektywość z hierarchią o jednym typie nadrzędnym - klasa System.Object;
- mechanizm automatycznego odśmiecania pamięci - tzw. garbage collector;
- kod źródłowy kompilowany do kodu pośredniego (Common Intermediate Language);
- wiele platform uruchomieniowych - .NET Framework, .NET Core, Mono i in.

Język C# jest przeze mnie wykorzystywany do pisania skryptów dla silnika Unity.

Python

Język programowania ogólnego przeznaczenia, o rozbudowanym pakiecie bibliotek standardowych [17]. Najważniejsze cechy:

- projekt Open Source;

- język interpretowany;
- dynamiczne typowanie - weryfikacja typu następuje w czasie wykonania programu. Twórcy języka kierowali się zasadą „duck typing” (kacze typowanie) - „jeśli obiekt zachowuje się jak kaczka, to jest kaczką”;
- silne typowanie - dla każdego typu dozwolone są tylko te operacje, które zostały dla niego zdefiniowane;
- garbage collector zarządza pamięcią;
- wsparcie dla wielu paradygmatów. W Pythonie można programować obiektywko, strukturalnie lub funkcyjnie;
- brak wsparcia dla mechanizmu enkapsulacji;
- prosta, czytelna składnia - bloki tworzone poprzez wcięcia;
- łatwy do nauczenia się;
- istnieje wiele implementacji - CPython, Jython, IronPython i in. Standardową implementacją jest CPython (implementacja w języku C).

W projekcie stosuję język Python w wersji 3.6.8. Do zarządzania zależnościami pakietów stosuję narzędzie Conda.

3.2.3 Frameworki

Framework jest spójnym zestawem modułów i bibliotek programistycznych, tworzących szkielet do budowy aplikacji danego typu. Programowanie przy użyciu frameworka polega na rozbudowywaniu tego szkieletu o dodatkowe komponenty, które są wymagane dla danego projektu. Korzystanie z frameworków ułatwia życie programistom, gdyż autorzy kodu nie muszą tracić czasu na implementację nudnych i powtarzalnych elementów aplikacji, które są identyczne dla wszystkich aplikacji tego typu. Zamiast tego mogą się skupić na implementacji funkcjonalności realizowanych przez ich oprogramowanie.

Unity ML-Agents

Otwartoźródłowa wtyczka do Unity, ułatwiająca tworzenie środowisk treningowych dla inteligentnych agentów [16]. Agenci mogą być uczeni rozmaitymi technikami, m. in.:

- uczenie ze wzmacnieniem (Reinforcement Learning);
- uczenie przez naśladowanie (Imitation Learning);
- neuroewolucja.

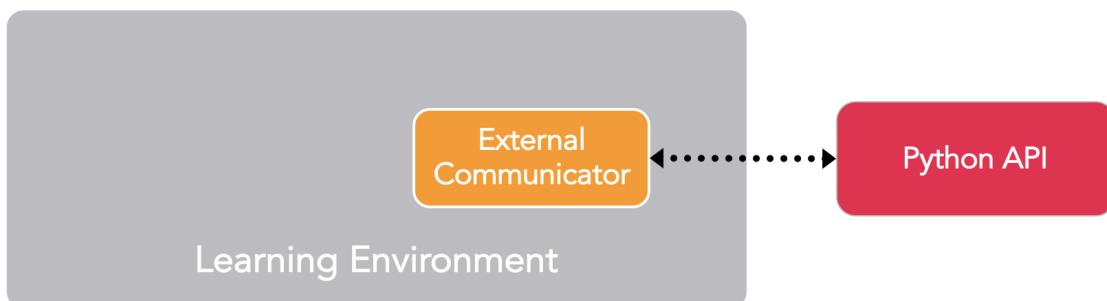
Framework udostępnia Pythonowe API, pozwalające na sterowanie środowiskiem ze skryptu

napisanego w Pythonie. Komunikacja pomiędzy zewnętrznym procesem Pythona, a środowiskiem Unity odbywa się poprzez mechanizm gniazd (z ang. *sockets*).

Unity ML-Agents składa się z trzech zasadniczych komponentów:

1. Środowisko Uczenia (Learning Environment) – symulacja środowiska treningowego.
2. Python API – zawiera zbiór klas i metod języka Python, pozwalających na tworzenie skryptów kontrolujących Środowisko Uczenia. Wśród kodu wchodzącego w skład API są przykładowe algorytmy uczenia maszynowego, dostarczone przez twórców frameworka. Skrypty wykorzystujące Python API nie są częścią Unity. Istnieją jako samodzielne, zewnętrzne procesy.
3. External Communicator – zapewnia komunikację pomiędzy Środowiskiem Uczenia i zewnętrznym procesem Pythona.

Na rysunku 3.2 został zobrazowany diagram opisanych komponentów. Z diagramu wynika, że External Communicator wchodzi w skład Środowiska Uczenia.



Rysunek 3.2: Uproszczony model frameworka Unity ML-Agents

Trening zachowań agentów w symulowanych środowiskach jest możliwy dzięki zdefiniowaniu trzech podstawowych pojęć:

1. Obserwacje (Observations) – zbiór informacji rejestrowanych przez agenta w każdym kroku symulacji. Istnieją dwa typy obserwacji:
 - Obserwacja Wektorowa (Vector Observation) – wektor liczb zmiennoprzecinkowych.
 - Obserwacje Wizualne (Visual Observations) – obrazy z kamer i/lub dane z renderowanych tekstur

2. Akcje (Actions) – instrukcje wydawane przez kod sterujący agentem. Akcje są podejmowane na podstawie obserwacji otrzymanych od agenta. Akcje mają najczęściej postać wektora liczb zmiennoprzecinkowych.
3. Sygnały nagród (Reward Signals) – wartości liczbowe, uzyskiwane co pewien czas ze Środowiska Uczenia. Sygnały nagród są miarą, wyznaczającą stopień poprawności wykonania zadań przez agenta.

Uproszczony opis pętli symulacji:

1. Następuje krok symulacji.
2. Agent zbiera obserwacje i wysyła „na zewnątrz”.
3. Kod sterujący agentem wyznacza akcję na podstawie bieżących obserwacji.
4. Agent wykonuje akcję.
5. Opcjonalnie: Środowisko emituje sygnał nagrody.
6. Powrót do punktu 1.

Celem treningu jest zazwyczaj zmaksymalizowanie sumy zdobytych nagród.

Środowisko uczenia zawiera trzy dodatkowe komponenty, ułatwiające organizację procesu uczenia:

1. Klasa Agent (agenci) – instancje klas dziedziczących po klasie Agent są przypinane do instancji klasy GameObject. Dlatego każdy obiekt sceny Unity może być agentem. Agenci są odpowiedzialni za generowanie obserwacji, wykonywanie akcji oraz emisję sygnałów nagród. Każdy agent jest połączony z dokładnie jednym mózgiem, czyli obiektem klasy dziedziczącej po klasie Brain
2. Klasa Brain (mózg) – klasa interfejsowa, odpowiedzialna za logikę podejmowania decyzji przez agentów. Każdy agent musi być przypisany do dokładnie jednego mózgu, natomiast jeden mózg może zarządzać wieloma agentami. Mózg podejmuje decyzje, jakie akcje dla poszczególnych obserwacji powinien wykonać agent. Ścisłej mówiąc, mózg jest komponentem, który otrzymuje obserwacje od agenta i wysyła mu akcje do wykonania.

Framework Unity ML-Agents udostępnia 3 klasy dziedziczące po klasie Brain:

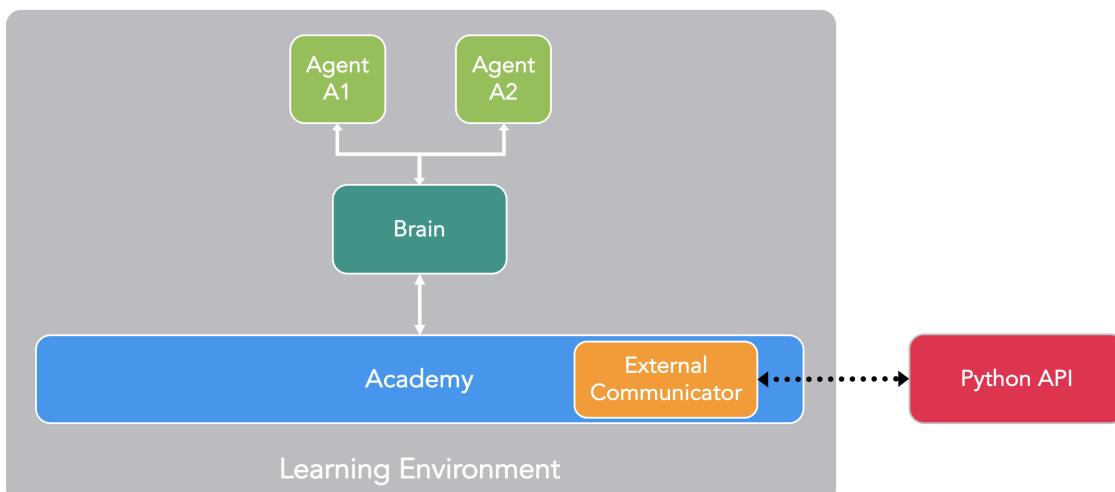
- LearningBrain – przeznaczony do uruchamiania wytrenowanych modeli oraz trenowania nowych.
- HeuristicBrain – przeznaczony do bezpośredniego zapisu w kodzie strategii zachowania agentów.

- PlayerBrain – mapuje przyciski klawiatury na konkretne akcje. Może być wykorzystywany do ręcznego testowania stworzonego Środowiska Ucznia.
3. Klasa Academy (akademia) – klasa interfejsowa, odpowiedzialna za organizację procesu zbierania obserwacji i podejmowania decyzji. Dla każdej sceny Unity może istnieć jeden i tylko jeden obiekt klasy dziedziczącej po klasie Academy. Akademia ma kilka istotnych parametrów, które mogą być ustawiane w oknie inspektora. Komponent zewnętrznego komunikatora (External Communicator), który jest odpowiedzialny za komunikację pomiędzy Środowiskiem Ucznia a zewnętrznym procesem Pythona, znajduje się wewnątrz obiektu akademii. Jest więc częścią klasy Academy.

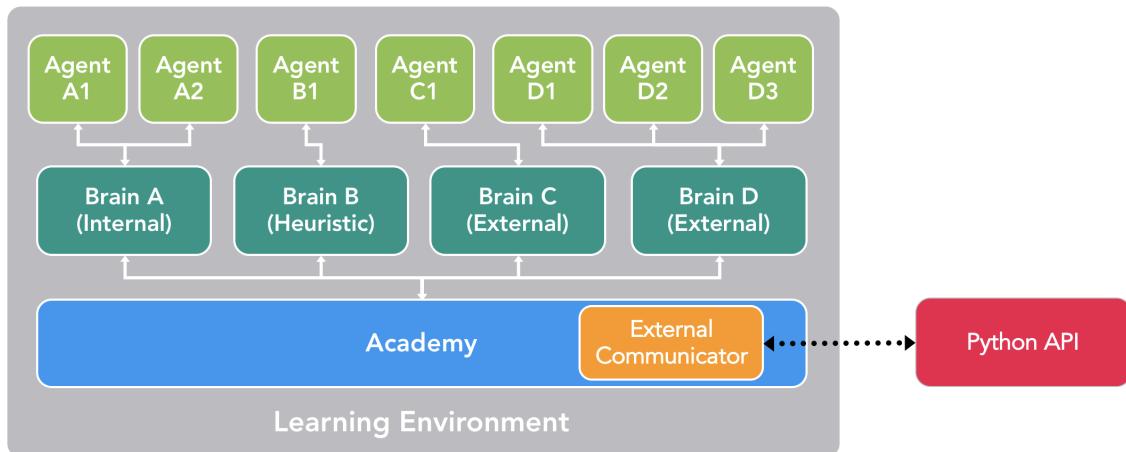
Omówione do tej pory komponenty (Learning Environment, External Communicator, Agent, Brain, Academy oraz Python API) tworzą drzewo zależności. Liczba mózgów i agentów może być w takim drzewie dowolna, o ile będą zachowane zasady opisane powyżej (każdy mózg ma co najmniej jednego agenta, każdy agent jest przypisany do dokładnie jednego mózgu).

Przykłady takich drzew:

- Drzewo z jednym mózgiem i dwoma agentami:



- Drzewo z wieloma mózgami i wieloma agentami:



Dodatkowe funkcjonalności oferowane przez framework Unity ML-Agents:

1. Wsparcie dla Dockera [19]
2. Wsparcie dla treningu w chmurze obliczeniowej:
 - AWS,
 - Microsoft Azure.

Więcej informacji na temat omawianej wersji frameworka można znaleźć w oficjalnej dokumentacji, dostępnej pod adresem:

<https://github.com/Unity-Technologies/ml-agents/tree/0.9.1/docs>

PyTorch

Otwartoźródłowy framework uczenia maszynowego. Jest przeznaczony dla języków Python i C++. Najważniejsze cechy i funkcjonalności frameworka [18] [74]:

1. Prostota oraz zgodność z filozofią Zen Python [68]
2. Łatwy dostęp do zawartości zmiennych
3. Integracja z biblioteką NumPy - dwie proste funkcje `tensor.numpy` oraz `tensor.from_numpy` pozwalają na proste i co ważne szybkie przekształcanie pomiędzy tablicą numpy a tensorem (tensor jest podstawowym typem tablicowym, używanym we frameworku PyTorch). Oprócz prostego przekształcania, PyTorch przejmuje po bibliotece NumPy całą filozofię pracy - większość operacji możliwych do wykonania na tablicach numpy, można również wykonać na tensorach.

4. Dynamiczne budowanie grafu obliczeniowego [48]
5. Ułatwione uruchamianie na GPU
6. Łatwe operowanie na zbiorach danych dzięki interfejsom Dataset i DataLoader
7. TorchScript - wsparcie dla tworzenia skompilowanych wersji tworzonego modelu.
8. Wsparcie dla standardu ONNX (Open Neural Network Exchange)
9. Oferuje 2 frontendy:
 - Dla Pythona
 - Dla C++
10. Wsparcie dla wielu platform chmurowych, w tym między innymi:
 - Amazon Web Services
 - Google Cloud Platform
 - Microsoft Azure

Framework PyTorch składa się z wielu modułów. Najistotniejszym dla mnie modułem był moduł torch.nn, który wykorzystałem do implementacji sieci neuronowej. Więcej na ten temat można znaleźć w rozdziale poświęconym implementacji systemu.

3.2.4 Biblioteki

Biblioteka programistyczna to zbiór powiązanych tematycznie jednostek kodu (takich jak klasy czy funkcje). Użycie biblioteki to sposób na ponowne wykorzystanie tego samego kodu. Dobór odpowiednich bibliotek pozwala na znaczne przyspieszenie procesu tworzenia oprogramowania. Poniżej znajduje się krótkie zestawienie bibliotek wykorzystywanych w mojej aplikacji.

ddt

Pakiet dla języka Python [22]. Pozwala na łatwe tworzenie parametryzowanych testów. Jest stosowany jako rozszerzenie funkcjonalności frameworków testowych, takich jak unittest (który wchodzi w skład biblioteki standardowej Pythona). Nazwa pakietu pochodzi od metodyki DDT - Data Driven Testing, która zakłada oddzielenie logiki testu od danych na których test operuje. Takie podejście pozwala w znacznym stopniu ograniczyć duplikację kodu w klasach testowych.

docopt

Pakiet pomagający utworzyć interfejs wiersza poleceń [5]. Pakiet docopt jest oparty na konwencji, wykorzystywanej przez lata dla komunikatów pomocy i stron podręcznika man. Opis interfejsu jest sformalizowanym komunikatem pomocy.

matplotlib

Biblioteka do tworzenia wykresów dla języka Python. Uchodzi za wszechstronne, lecz trudne w opanowaniu narzędzie.

3.2.5 Conda

System open source do zarządzania pakietami i środowiskami uruchomieniowymi dla systemów Windows, macOS oraz Linux. Posiada wsparcie dla wielu języków, m. in. Python, R, Ruby, Lua, Scala [12]. Conda potrafi szybko instalować, uruchamiać i aktualizować pakiety wraz z ich zależnościami. Bardzo łatwo można tworzyć nowe wirtualne środowiska dla Pythona.

3.2.6 Wykorzystywane wersje oprogramowania

1. Silnik Unity - 2019.1.12f
2. C# - Mono 6.4.0.198
3. Python - 3.6.8
4. Unity ML-Agents - 0.9.1
5. PyTorch - 1.1.0
6. ddt - 1.2.1
7. docopt - 0.6.2
8. matplotlib - 3.1.1
9. Conda - 4.7.5

3.2.7 Stacja robocza - specyfikacja techniczna

Poniżej zamieszczam specyfikację techniczną stacji roboczej (komputera), którą wykorzystałem do implementacji systemu oraz wszystkich eksperymentów obliczeniowych:

1. Typ urządzenia - Laptop
2. Marka i model urządzenia - Dell Inspiron 7559
3. Procesor - Intel(R) Core(TM) i7-6700HQ (2.6 GHz)
4. Pamięć RAM - SODIMM DDR3 Synchronous 1600 MHz (8 GB)
5. Karty graficzne:
 - nVidia GeForce GTX 960M
 - Intel HD Graphics 530
6. Dysk - PLEXTOR PX-512M7 (SSD 512 GB)
7. System operacyjny - Linux Mint 19.1 Tessa

Rozdział 4

Opis implementacji

Zgodnie z opisem architektury systemu, zawartym w sekcji 3.1.3, zrealizowany przezem projekt można podzielić na dwa moduły:

1. Środowisko Uczenia - zrealizowane w silniku Unity. Odpowiada za symulację środowiska, po którym porusza się inteligentny agent.
2. Zewnętrzny proces Pythona - moduł odpowiedzialny za wykonywanie skryptów języka Python, kontrolujących zachowanie agentów w Środowisku Uczenia.

Zewnętrzny proces Pythona jest niezależnym od silnika Unity procesem. Komunikacja ze Środowiskiem Uczenia odbywa się poprzez mechanizm socketów. Implementacja tego mechanizmu jest wbudowana we framework, choć istnieje również możliwość rozszerzenia tej implementacji. Szczegóły na ten temat można znaleźć w dokumentacji frameworka.

Więcej informacji na temat każdego z modułów znajduje się w dalszej części rozdziału.

4.1 Środowisko Uczenia

Środowisko Uczenia - pomijając detale - jest typowym projektem zrealizowanym w silniku Unity. W skład projektu wchodzi wiele katalogów i plików.

W niniejszej sekcji postanowiłem skupić się na opisaniu tylko tych elementów, które są najbardziej istotne dla zrozumienia aplikacji. Reszta informacji znajduje się w dokumentacji silnika, gdyż dotyczy typowych elementów, obecnych w każdym projekcie Unity.

4.1.1 Wykorzystane assety

Assety są komponentami, których można użyć do tworzenia gier lub innych projektów napędzanych silnikiem Unity. Asset może być jakimkolwiek zasobem do wykorzystania, np. modelem 3D, plikiem audio lub skryptem C#. Twórcy silnika Unity udostępniają usługę Unity Asset Store, dzięki której można nabywać bezpłatne a także płatne assety. Usługa ta jest wbudowana w edytor Unity [79].

W swojej aplikacji również wykorzystuję zewnętrzne assety. Dokładniej mówiąc, jeden pakiet assetów. Pakiet ten nosi nazwę „*Low Poly Destructible 2 Cars no. 8*”, a dokładny opis znajduje się na oficjalnej stronie pakietu [10]. Powyższy pakiet posłużył jako baza dla modelu agenta.



Rysunek 4.1: Zdjęcie reklamujące pakiet „*Low Poly Destructible 2 Cars no. 8*”

4.1.2 Sceny Unity

Są to wymodelowane w edytorze Unity tory wyścigowe, po których poruszają się agenci podczas trwania symulacji. Każdy kolejny tor jest trudniejszy od poprzedniego, co przekłada się na większą liczbę epizodów symulacji, potrzebnych do wytrenowania modelu.

Obecnie Środowisko Uczenia składa się z trzech torów:

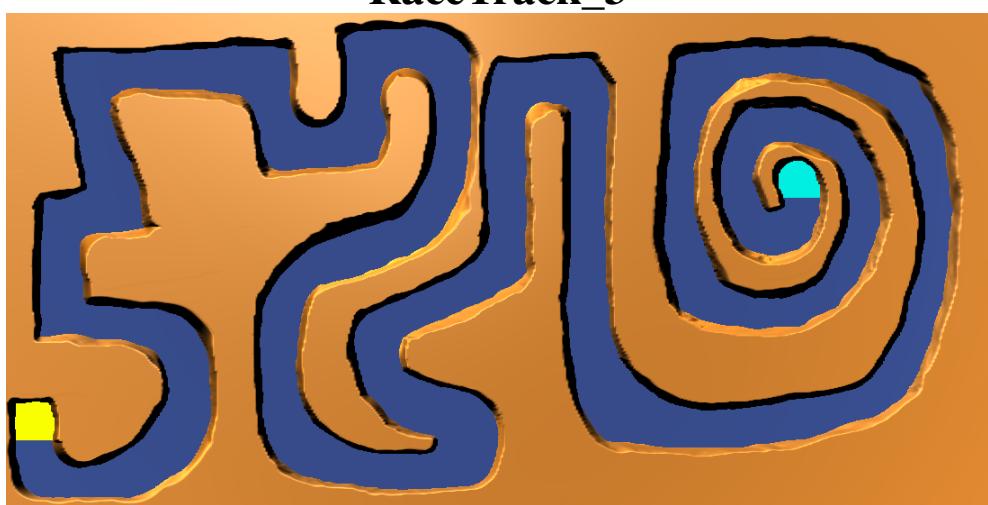
RaceTrack_1



RaceTrack_2



RaceTrack_3

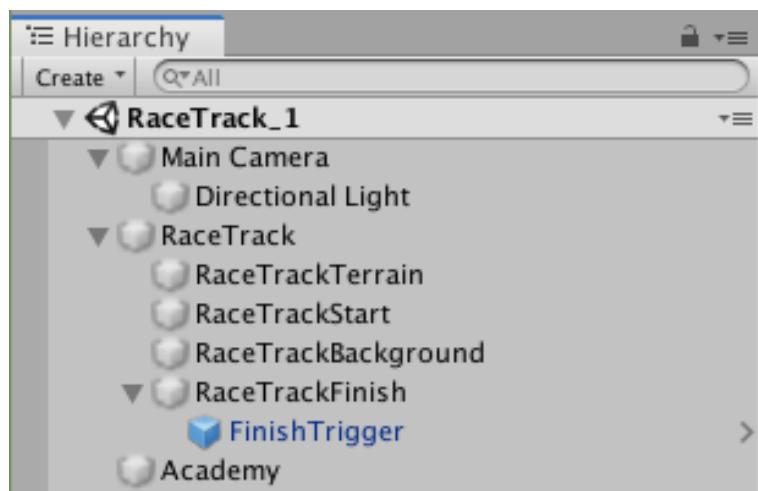


Wszystkie tory wyścigowe mają wspólne cechy. Jedną z nich jest kolorystyka. Każdy kolor ma swoje znaczenie:

- Żółty – miejsce startowe;
- Turkusowy – meta;
- Granatowy – nitka toru, po której porusza się agent;
- Brązowo-pomarańczowy – ściany toru. Służą do wyznaczania granic, w obrębie których agent może się przemieszczać. W przypadku kolizji ze ścianą, agent kończy swoją symulację dla bieżącego epizodu.

Struktura logiczna wszystkich scen jest zasadniczo taka sama. Każda scena składa się z obiektu kamery, obiektu akademii oraz wymodelowanego toru wyścigowego.

Poniżej przykład dla sceny RaceTrack_1:



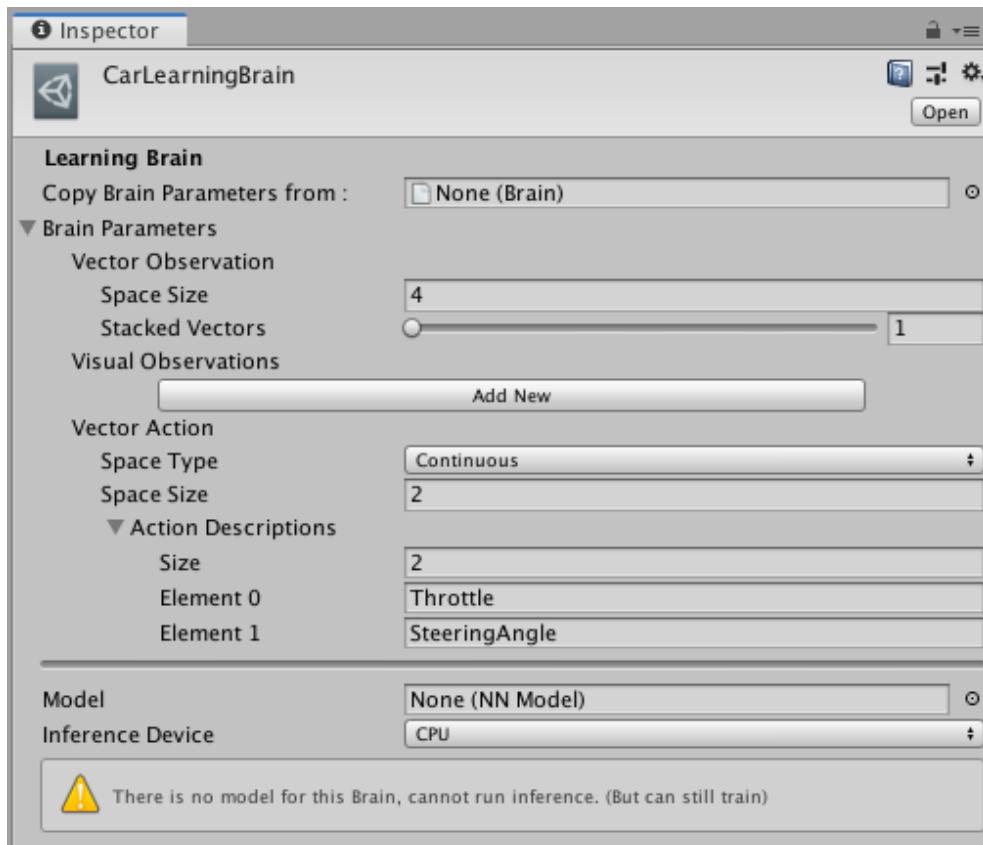
4.1.3 Katalog MyBrains

Katalog MyBrains zawiera dwa obiekty, które dziedziczą po klasie Brain. Są to:

1. *CarLearningBrain*

Obiekt klasy LearningBrain. Używany do uruchamiania wytrenowanych modeli oraz trenowania nowych. Modele są definiowane i wykonywane po stronie zewnętrznego procesu Pythona. Obiekt CarLearningBrain musi zatem komunikować się (za pośrednictwem akademii) z procesem Pythona, którego odpytuje o akcje zwarcane przez model oraz wysyła mu obserwacje zgromadzone przez agenta.

CarLearningBrain posiada kilka parametrów, których wartości wpływają na jego zachowanie. Parametry te są dostępne z poziomu edytora Unity.



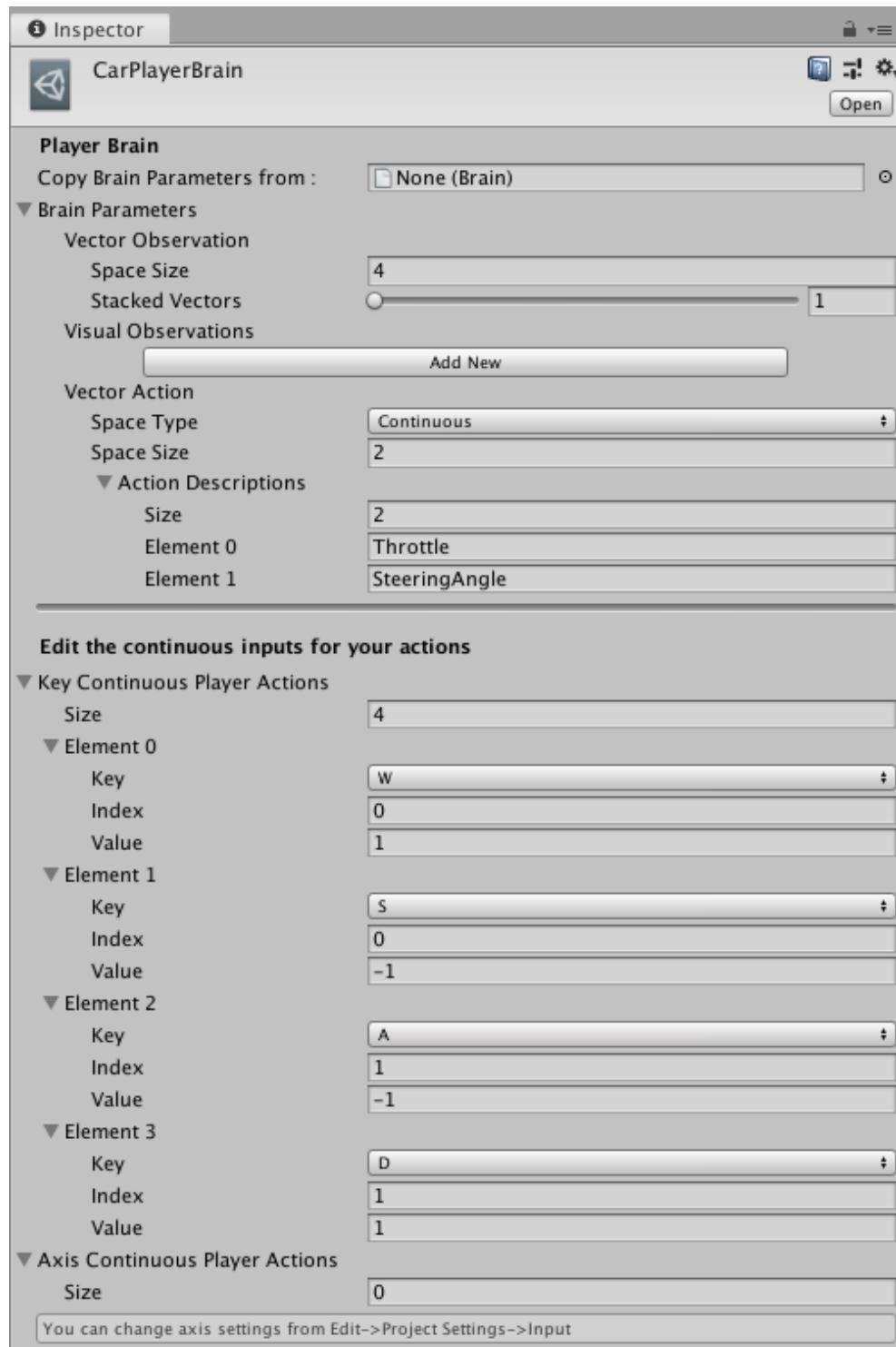
Rysunek 4.2: Widok obiektu *CarLearningBrain* w inspektorze edytora Unity

Najbardziej istotne są parametry występujące w sekcji *Brain Parameters*. Są one definiowane w klasie *Brain*, zatem wszystkie klasy dziedziczące po klasie *Brain* posiadają te parametry. Opis poszczególnych parametrów znajduje się w dokumentacji frameworka Unity ML-Agents [80].

2. *CarPlayerBrain*

Obiekt klasy *PlayerBrain*. Pozwala na kontrolowanie agenta, używając komend klawiatury. Doskonale nadaje się do ręcznych testów Środowiska Uczenia.

Jak widać na obrazku 4.3, *CarPlayerBrain* również posiada sekcję *Brain Parameters*. Bardzo interesująca jest natomiast sekcja poniżej, w której zdefiniowane jest mapowanie poszczególnych przycisków klawiatury na konkretne akcje generowane przez obiekt. Więcej szczegółów na ten temat można znaleźć w dokumentacji [81].

Rysunek 4.3: Widok obiektu *CarPlayerBrain* w inspektorze edytora Unity

4.1.4 Skrypty C#

Aby Środowisko Uczenia działało w sposób poprawny, musiałem zaimplementować skrypty w języku C#. Poniżej zamieszczam krótki opis poszczególnych skryptów.

RaceTrackAcademy.cs

Najważniejszy skrypt w module. Jest odpowiedzialny za zarządzanie Środowiskiem Uczenia i udostępnianie go zewnętrznemu procesowi Pythona.

Klasa RaceTrackAcademy dziedziczy po interfejsie Academy, który definiuje API dla obiektu akademii. Metody abstrakcyjne, które zostały nadpisane przez klasę implementującą to:

1. InitializeAcademy

Jest wykonywana tylko jeden raz - podczas inicjalizacji obiektu akademii.

Służy do przygotowania stanu początkowego Środowiska Uczenia.

W przypadku mojej implementacji, polega to na utworzeniu populacji agentów.

2. AcademyReset

Jest wykonywana po każdym zakończonym epizodzie symulacji. Epizod symulacji kończy się, gdy zostaje wywołana metoda *Done* z klasy Academy.

Metoda *AcademyReset* powinna przygotować Środowisko Uczenia do kolejnego epi-zodu symulacji. W przypadku mojej implementacji, polega to na przemieszczeniu wszystkich agentów do pozycji początkowej i wyzerowaniu ich nagród.

3. AcademyStep

Jest wykonywana dla każdego kroku symulacji. W przypadku mojej implementacji, metoda ta jest odpowiedzialna za sprawdzanie, czy wszyscy agenci zakończyli jazdę w bieżącym epizodzie. Jeśli tak, wywoływana jest metoda *Done*.

Oprócz powyżej opisanych, klasa RaceTrackAcademy zawiera również kilka metod pomocniczych. Wspomagają one tworzenie populacji agentów oraz liczenie agentów którzy zakończyli jazdę w bieżącym epizodzie.

Bardzo ważnym aspektem klasy RaceTrackAcademy, o którym również należy wspomnieć, są publiczne pola instancyjne klasy. To parametry mające duży wpływ na zachowanie Środowiska Uczenia. Wartości tych parametrów mogą być ustawiane z poziomu edytora Unity. Oto ich lista:

- CarAgentBrain - obiekt klasy dziedziczącej po klasie Brain. Więcej szczegółów na ten temat znajduje się w sekcji 3.2.3.

- CarAgentPrefab - zmienna przechowuje referencję do obiektu reprezentującego wzorcowy model agenta. Podczas tworzenia populacji, model ten jest klonowany dla każdego z agentów.
- StartAgentPosition - pozycja startowa agenta. Jest to wektor określający pozycję agenta, jaką będzie on miał na początku każdego epizodu symulacji.
- StartAgentRotation - liczba typu float. Określa, w którą stronę jest skierowany samochód na początku każdego epizodu.
- CarAgentScale - liczba typu float. Określa rozmiar klonowanych modeli w stosunku do rozmiaru modelu oryginalnego (wskazywanego przez zmienną CarAgentPrefab).
- MaxSensorLength - liczba typu float. Określa maksymalny dystans, jaki może być rejestrowany przez czujniki odległości montowane na każdym z samochodów.
- FieldOfView - liczba typu float. Określa „kąt widzenia” samochodu. Wartości podawane są w stopniach. Maksymalna dozwolona wartość (będąca zarazem wartością domyślną) to 180 stopni.
- RaysCount - liczba typu int. Określa liczbę czujników dla każdego samochodu. Musi to być liczba dodatnia, większa lub równa 2.
- MaxSteeringAngle - liczba typu float. Określa maksymalny dozwolony kąt skrętu kół przednich w samochodzie. Wartości podawane są w stopniach. Domyślnie jest to 30 stopni.
- MotorForce - liczba typu float. Określa „moc silnika” w samochodzie.
- PopulationSize - rozmiar populacji agentów.

CarAgent.cs

Drugi najważniejszy skrypt w module. Klasa CarAgent dziedziczy po klasie Agent, która definiuje publiczne API do zarządzania agentem w Środowisku Uczenia. Każdemu agentowi przypada dokładnie jedna instancja klasy CarAgent. Klasa CarAgent nadpisuje kilka metod wirtualnych, zdefiniowanych w klasie Agent. Są to:

1. AgentReset

Określa zachowanie agenta, gdy jest on resetowany. Resetowanie agenta odbywa się po zakończeniu epizodu symulacji, a celem resetowania jest przygotowanie agenta do rozpoczęcia kolejnego epizodu. W przypadku mojej implementacji, resetowanie agenta polega na wyzerowaniu jego nagrody.

2. CollectObservations

Wykonywana dla każdego kroku symulacji. Odpowiada za zbieranie obserwacji agenta i wysyłania tych danych do przypisanego agentowi mózgu.

3. *AgentAction*

Wykonywana dla każdego kroku symulacji. Odpowiada za zdefiniowanie zachowania agenta na podstawie danych (akcji) otrzymywanych z mózgu.

4. *AgentOnDone*

Metoda definiuje zachowanie agenta, gdy ten kończy symulację i zmienna AgentParameters.resetOnDone jest ustawiona na wartość *false*.

Oprócz powyższych metod, klasa CarAgent zawiera również kilka metod pomocniczych. Najważniejsze z nich to:

1. *Constructor* - odpowiedzialna za leniwą inicjalizację obiektu.
2. *SetInputProperties* - odpowiedzialna za stworzenie sensorów samochodu. Robi to na podstawie wartości przekazanych w parametrach metody (maksymalny dystans, kąt widzenia oraz liczba sensorów).
3. *SetOutputProperties* - odpowiada za przygotowanie samochodu do poprawnego reagowania na akcje przesyłane agentowi przez mózg.
4. *SaveEpisodeReward* - zapisuje nagrodę przypadającą agentowi za bieżący epizod. Wartość nagrody jest obliczana na podstawie przejechanego dystansu oraz sygnałów nagród wyemitowanych przez Środowisko Uczzenia.
5. *GetEpisodeReward* - zwraca wartość nagrody, przypadającej agentowi za bieżący epizod.

CarAgentInput.cs

Skrypt zawiera szczegóły implementacyjne na temat zestawu sensorów wejściowych agenta. Najważniejszą metodą tej klasy jest *RenderSensorsAndGetNormalizedDistanceList*. Metoda ta renderuje zestaw sensorów i zwraca listę znormalizowanych wartości zarejestrowanych dystansów. Znormalizowane wartości to takie, gdzie każda z nich jest liczbą zmiennoprzecinkową z przedziału od 0 do 1.

CarAgentOutput.cs

Skrypt zawiera szczegóły implementacyjne na temat „układu napędowego” samochodu. Najważniejszą metodą w tej klasie jest *Update*, która aktualizuje kąt skrętu kierownicy oraz wielkość mocy przekazywanej na koła.

CarSensor.cs

Skrypt zawiera szczegóły implementacyjne na temat pojedynczych sensorów. Definiuje klasę CarSensor, której metody są pomocne przy operowaniu na poszczególnych sensorach.

Najważniejsze metody tej klasy to *Render* oraz *GetNormalizedDistance*.

SensorPropertiesComputer.cs

Skrypt zawiera definicję klasy SensorPropertiesComputer, która dokonuje obliczeń parametrów sensora, czyli pozycji punktu początkowego oraz kierunku wiązki.

RaceTrackFinishTrigger.cs

Skrypt zawiera definicję klasy RaceTrackFinishTrigger, która odpowiada za rozpoznawanie, czy dany agent dojechał do końca toru. Jeśli tak, to klasa przydziela agentowi nagrodę za ukończenie toru i wywołuje na nim metodę *Done*. Dzięki temu agent wie, że dla niego ten epizod symulacji dobiegł końca.

RaceTrackTerrainCollision.cs

Skrypt zawiera definicję klasy RaceTrackTerrainCollision, która odpowiada za wykrywanie kolizji pomiędzy agentem i ścianami toru wyścigowego. Gdy zachodzi kolizja, klasa wywołuje na agencie metodę *Done*.

EnvBuilder.cs

Skrypt odpowiada za budowanie Środowiska Uczenia do postaci samowystarczalnych aplikacji, które nie potrzebują mieć dostępu do edytora Unity aby funkcjonować poprawnie. Tych aplikacji, zwanych inaczej buildami, są trzy sztuki - po jednej na każdy tor wyścigowy. Skrypt pobiera z linii poleceń dwa parametry. Są to:

- Ścieżka do katalogu, w którym będą utworzone wszystkie buildy.
- Platforma systemowa. Obecnie wspierane platformy to Linux i Windows.

4.2 Zewnętrzny proces Pythona

Jest drugim z głównych modułów mojej aplikacji. Zawiera zestaw skryptów języka Python oraz inne pliki, które pozwalają na komunikację i kontrolę Środowiska Uczenia z osobnego procesu, niezwiązanego z procesami silnika Unity.

Na obrazku 4.4 została przedstawiona uproszczona struktura katalogowa tego modułu. Najważniejsze elementy z tej struktury zostały opisane poniżej.

```
.-- config.json
.-- env_builds
.-- experiment.py
.-- experiment_results
.-- .gitignore
.-- make_builds.py
.-- requirements.txt
.-- run.py
src
|-- AgentNeuralNetwork.py
|-- experiment
|   |-- ChartsGenerator.py
|   |   `-- __init__.py
|   |-- __init__.py
|   |-- Logger.py
|   `-- test
|       |-- experiment
|       |   |-- __init__.py
|       |   |   `-- test_ChartsGenerator.py
|       |   |   `-- test_ExperimentDataCollector.py
|       |   |-- __init__.py
|       |   |-- test_AgentNeuralNetwork.py
|       |   |-- test_Logger.py
|       |   `-- training
|       |       |-- __init__.py
|       |       |   `-- test_TrainingResultsRepository.py
|       |       |   `-- test_training_utilities.py
|       |       `-- __init__.py
|       |           `-- TrainingResultsRepository.py
|       |           `-- training_utilities.py
|       `-- train_de.py
`-- training_results
`-- train_pso.py
```

Rysunek 4.4: Struktura katalogowa modułu

4.2.1 Lista pakietów zależności

Zdefiniowana w pliku **requirements.txt**. Zawiera nazwy i numery wersji pakietów, które muszą być zainstalowane aby proces Pythona działał w poprawny sposób. Preferowanym

sposobem instalowania zależności jest utworzenie nowego środowiska wirtualnego (najlepiej przy użyciu programu *Conda*) i zainstalowanie wszystkich zależności w tym środowisku.

4.2.2 Konfiguracja modułu

Wykorzystywana przez niektóre skrypty języka Python. Dane konfiguracyjne zawarte są w pliku **config.json**, który definiuje parametry niezbędne do poprawnego funkcjonowania modułu. Formatem pliku jest JSON [45], a jego aktualna zawartość prezentuje się następująco:

```
{
  "MakeBuilds" : {
    "Unity" : "/home/galgreg/Programy/Unity3D/2019.1.12f1/Editor/Unity",
    "Target" : "Linux"
  },
  "BuildPaths" : {
    "Linux" : {
      "RaceTrack_1" : "env_builds/RaceTrack_1/RaceTrack_1.x86_64",
      "RaceTrack_2" : "env_builds/RaceTrack_2/RaceTrack_2.x86_64",
      "RaceTrack_3" : "env_builds/RaceTrack_3/RaceTrack_3.x86_64"
    },
    "Windows" : {
      "RaceTrack_1" : "env_builds/RaceTrack_1/RaceTrack_1.exe",
      "RaceTrack_2" : "env_builds/RaceTrack_2/RaceTrack_2.exe",
      "RaceTrack_3" : "env_builds/RaceTrack_3/RaceTrack_3.exe"
    }
  },
  "LearningAlgorithms" : {
    "pso" : {
      "numberOfAgents" : 100,
      "W" : 0.729,
      "c1" : 2.05,
      "c2" : 2.05
    },
    "diff_evo" : {
      "numberOfAgents" : 50,
      "mutationFactor" : 0.8,
      "crossProbability" : 0.7
    }
  },
  "TrainingParameters" : {
    "randomSeed" : null,
    "maxNumberOfEpisodes" : 100,
    "maxNumberOfRepeatsIfTrainingFails" : 3,
    "minimalAcceptableFitness": {
      "RaceTrack_1" : 14.00,
      "RaceTrack_2" : 20.00,
      "RaceTrack_3" : 40.00
    },
    "networkHiddenDimensions" : [5]
  }
}
```

Znaczenie poszczególnych sekcji pliku **config.json**:

- MakeBuilds - parametry dla skryptów tworzących buildy Środowiska Uczenia.
- BuildPaths - ścieżki do buildów dla wszystkich wspieranych platform systemowych.
- LearningAlgorithms - parametry dla Ewolucji Różnicowej oraz PSO.
- TrainingParameters - inne parametry, niezbędne do poprawnego działania aplikacji.

4.2.3 Argumenty wiersza poleceń

Niektóre skrypty modułu można wywołać z poziomu wiersza poleceń. Są to:

- train_de.py
- train_pso.py
- run.py
- make_builds.py
- experiment.py

Do przetwarzania argumentów wiersza poleceń jest wykorzystywana biblioteka docopt (patrz sekcja 3.2.4). Oto przykład definicji API wiersza poleceń dla skryptu train_pso.py:

```
def getProgramOptions():
    APP_USAGE_DESCRIPTION = """
Train neural networks to drive a car on a racetrack. Racetrack must be valid
Unity ML-Agents environment. Algorithm used to train is Particle Swarm Optimization.
NOTE: As a config file should be used 'config.json' file or other with appropriate fields.

Usage:
    train_pso.py <config-file-path> (--track-1 | --track-2 | --track-3) [options]
    train_pso.py -h | --help

Options:
    --track-1                         Run training on RaceTrack_1
    --track-2                         Run training on RaceTrack_2
    --track-3                         Run training on RaceTrack_3
    -v --verbose                       Run in verbose mode
    --save-population                  Save population after training
    --population=<pretrained-population> Specify path to pretrained population
    --env-path=<unity-build>          Specify path to Unity environment build
"""
    options = docopt(APP_USAGE_DESCRIPTION)
    return options
```

Parsowanie argumentów linii poleceń odbywa się w funkcji *getProgramOptions*. Łańcuch APP_USAGE_DESCRIPTION zawiera sformalizowany opis, wykonany według schematu określonego w dokumentacji [5]. Funkcja *docopt* przetwarza ten łańcuch znaków oraz

listę argumentów dostarczoną przy wywołaniu z linii poleceń. *docopt* weryfikuje, czy podane argumenty zgadzają się z definicją API. Jeśli tak, to funkcja zwraca słownik zawierający odczytane argumenty. Na przykład dla wywołania:

```
python train_de.py config.json --track-3 --verbose
```

słownik zwrócony przez funkcję *docopt* będzie miał następującą postać:

```
{  
    "<config-file-path>" : "config.json",  
    "--track-1" : False,  
    "--track-2" : False,  
    "--track-3" : True,  
    "--verbose" : True,  
    "--save-population" : False,  
    "--population" : None,  
    "--env-path" : None  
}
```

4.2.4 Skrypty treningowe

Skrypty treningowe odpowiadają za trenowanie nowych modeli. Trening odbywa się przy użyciu jednego z dwóch algorytmów:

- Ewolucji Różnicowej - dla skryptu *train_de.py*
- Optymalizacji Roju Cząstek - dla skryptu *train_pso.py*

Obydwa skrypty można wywołać na jeden z dwóch sposobów - z poziomu linii poleceń lub z poziomu kodu źródłowego innego skryptu.

Wykonanie skryptów ma podobny przebieg. Można go opisać za pomocą poniższej listy kroków:

1. Parsowanie argumentów wywołania;
2. Wczytanie danych z pliku konfiguracyjnego;
3. Nawiązanie połączenia ze Środowiskiem Uczenia;
4. Inicjalizacja zmiennych;
5. Pętla treningowa;
6. Zapis danych uzyskanych poprzez trening.

Wykonanie pętli treningowej odbywa się aż do momentu wystąpienia warunku stopu. Warunkiem stopu jest jedna z następujących sytuacji:

1. Wytrenowany model osiągnął założony pułap wartości przystosowania;
2. Wyczerpała się liczba dozwolonych prób wytrenowania modelu.

Informacje zapisywane po treningu to:

- Plik dziennika, zawierającego opis przebiegu treningu;
- Najlepiej wytrenowany model;
- Cała populacja, której stan wytrenowania odpowiada stanowi po ostatnim przeprowadzonym epizodzie treningowym. Zapis populacji jest opcjonalny i występuje po wywołaniu skryptu z odpowiednim argumentem.

4.2.5 Pozostałe skrypty

Oprócz skryptów treningowych, w module znajdują się także inne skrypty. Są one równie ważne dla poprawnego funkcjonowania aplikacji.

run.py

Skrypt odpowiada za wczytanie wytrenowanego modelu i jego ewaluację w zadanym Środowisku Uczenia. Skrypt może być wywołany na jeden z dwóch sposobów - z poziomu linii poleceń lub z poziomu kodu źródłowego innego skryptu.

make_builds.py

Skrypt odpowiada za tworzenie buildów Środowiska Uczenia. Buildy te są potrzebne dla poprawnego działania skryptu *experiment.py*.

experiment.py

Jeden z najważniejszych skryptów w module. Przeprowadza serię eksperymentów obliczeniowych, których wyniki zapisuje do katalogu *experiment_results*, w podkatalogu o nazwie będącej znacznikiem czasu z momentu zapisu.

Wyniki przeprowadzonych eksperymentów obliczeniowych (jak również metodyka ich przeprowadzania) zostaną omówione w rozdziale 5.

Katalog test

Zawiera zbiór testów jednostkowych modułu. Testy zaimplementowano, wykorzystując framework unittest [34] oraz bibliotekę ddt (patrz sekcja 3.2.4).

AgentNeuralNetwork.py

Plik zawiera definicję klasy *AgentNeuralNetwork*, która implementuje model sieci neuronowej agentów. Klasa jest stosunkowo prosta, składa się tylko z dwóch metod.

Pierwszą z nich jest konstruktor. W konstruktorze definiowana jest topologia sieci neuronowej. Obecnie wykorzystywana topologią sieci jest MLFFNN [76], czyli prosta sieć jednokierunkowa. Konstruktor przyjmuje jeden obowiązkowy parametr - jest nim lista wymiarów sieci.

Drugą metodą klasy jest *forward*. Definiuje ona sposób obliczania wyników generowanych przez sieć neuronową. Wykorzystywaną funkcją aktywacji jest ELU [42].

Zawartość pliku *AgentNeuralNetwork.py* prezentuje się następująco:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class AgentNeuralNetwork(nn.Module):
    def __init__(self, dimensions, requires_grad = False):
        super(AgentNeuralNetwork, self).__init__()
        self._layers = []
        for i in range(len(dimensions) - 1):
            self._layers.append( nn.Linear(dimensions[i], dimensions[i+1]) )
            self._layers[i].weight = \
                nn.Parameter(
                    data = torch.FloatTensor(dimensions[i+1], dimensions[i]) \
                        .uniform_(-2.0, 2.0),
                    requires_grad = requires_grad)
            self._layers[i].bias = \
                nn.Parameter(
                    data = torch.FloatTensor(dimensions[i+1]) \
                        .uniform_(-2.0, 2.0),
                    requires_grad = requires_grad)

    def forward(self, dataToProcess):
        dataToProcess = torch.tensor(dataToProcess)
        for networkLayer in self._layers:
            dataToProcess = F.elu(networkLayer(dataToProcess))
        dataToProcess[0] = min((dataToProcess[0] + 1), 1)
        dataToProcess[1] = min(dataToProcess[1], 1)
        return dataToProcess.tolist()
```

Logger.py

Plik zawiera definicję klasy *Logger*. Klasa ta odpowiada za zarządzanie logami generowanymi przez inne skrypty w module. Najważniejsze metody klasy:

1. *Append* - dodaje podany string do loga jako kolejny rekord. Każdy rekord jest opatrzony tzw. „znacznikiem czasu”, czyli datą i godziną dodania rekordu.
2. *Save* - zapisuje loga do pliku o zadanej lokalizacji.

TrainingResultsRepository.py

Plik zawiera definicję klasy *TrainingResultsRepository*. Odpowiada ona za zapisywanie i odczytywanie danych wygenerowanych podczas treningu. Dane te są zapisywane do katalogu *training_results*, w podkatalogu o nazwie będącej znacznikiem czasu z momentu zapisu. Dane zapisywane po treningu to: plik loga, najlepszy wytrenowany agent oraz (opcjonalnie) cała populacja.

training_utilities.py

Plik zawiera definicje funkcji pomocniczych, które są wykorzystywane w innych skryptach modułu. Jest tutaj zdefiniowana między innymi funkcja przystosowania (wykorzystywana w skryptach treningowych), która implementuje jeden pełny przebieg symulacji w Środowisku Uczenia. Funkcja ta wykorzystuje Pythonowe API, zdefiniowane przez twórców frameworka Unity ML-Agents (patrz 3.2.3).

ChartsGenerator.py

Plik zawiera definicję klasy *ChartsGenerator*, która odpowiada za tworzenie wykresów na podstawie danych wygenerowanych podczas eksperymentów obliczeniowych. Do tworzenia wykresów wykorzystywana jest biblioteka matplotlib (patrz 3.2.4).

ExperimentDataCollector.py

Plik zawiera definicję klasy *ExperimentDataCollector*, która odpowiada za składowanie informacji wygenerowanych podczas eksperymentów obliczeniowych. Dane zgromadzone przez tę klasę są następnie wykorzystywane w klasie *ChartsGenerator* do tworzenia wykresów.

Rozdział 5

Eksperymenty obliczeniowe

W niniejszym rozdziale przedstawiam praktyczne aspekty tworzonej pracy, czyli eksperymenty obliczeniowe wykonane przy użyciu stworzonej przeze mnie aplikacji. Analizę wyników poprzedzam omówieniem metodyki, jaką stosowałem podczas przeprowadzania eksperymentów.

5.1 Metoda eksperymentów

Metoda eksperymentów obliczeniowych to lista przyjętych założeń dla przeprowadzanych eksperymentów. Oto najważniejsze założenia:

1. Wszystkie eksperymenty muszą wykonać się automatycznie, bez potrzeby aktywnego udziału człowieka. Pierwszy eksperiment powinien rozpocząć się po wywołaniu jednej prostej komendy w linii poleceń. Komendą jest nazwa skryptu implementującego główną logikę przebiegu eksperymentów.
2. Parametry dla eksperymentów obliczeniowych muszą być dostarczane z zewnętrznego pliku konfiguracyjnego o ustalonym formacie. Ścieżka do pliku konfiguracyjnego musi być podawana jako argument linii poleceń.
3. Liczba prób musi być podana jako argument linii poleceń. Jedna próba to wykonanie pełnego zestawu eksperymentów. Im więcej prób, tym bardziej wiarygodne wyniki końcowe. Z drugiej strony, zwiększanie liczby prób znacząco wydłuża czas wykonania kompletnej serii eksperymentów.
4. Podczas wykonywania eksperymentów, jak najwięcej istotnych informacji powinno być zapisywanych do pliku dziennika (zwanego również *logiem*). W przypadku problemów z aplikacją, posiadanie logów potrafi znacząco usprawnić proces debugowania.
5. Brak zainstalowanego silnika Unity nie powinien uniemożliwić przeprowadzenia eksperymentów obliczeniowych.
6. Wynikiem końcowym powinien być zestaw wykresów, utworzonych na bazie informacji zgromadzonych podczas przebiegu eksperymentów. Wykresy powinny dotyczyć

istotnych danych, pozwalających na wyciągnięcie konstruktywnych wniosków.

7. Podczas każdej próby przeprowadzanych jest sześć eksperymentów. Każdy eksperiment to trening populacji na jednym z trzech torów wyścigowych, przy użyciu jednego z dwóch dostępnych algorytmów uczenia - **Ewolucji Różnicowej** (patrz sekcja 2.3.1) lub **PSO** (patrz sekcja 2.3.2). Po każdym treningu, najlepiej wyuczony model jest ewaluowany na wszystkich torach wyścigowych. Ewaluacja ma na celu sprawdzenie, czy model wytrenowany na danym torze poradzi sobie z pozostałymi torami wyścigowymi.

5.2 Opis implementacji

Główna logika przebiegu eksperymentów została zaimplementowana w skrypcie *experiment.py*. W celu rozpoczęcia eksperymentów, ten skrypt należy wywołać z linii poleceń. Oto definicja API linii poleceń:

```
def getProgramOptions():
    APP_USAGE_DESCRIPTION = """
Run series of experiments which result in generating charts for IT Engineering Thesis.
NOTE: As a config file should be used 'config.json' file or other with appropriate
fields.

Usage:
    experiment.py <config-file-path> [options]
    experiment.py -h | --help

Options:
    --num-of-trials=<n>      Specify number of trials used to generate data.
                                [default: 10]
    -v --verbose               Run in verbose mode.
"""
    options = docopt(APP_USAGE_DESCRIPTION)
    return options
```

Wywołanie skryptu *experiment.py* wymaga podania co najmniej jednego argumentu. Jest nim ścieżka do pliku konfiguracyjnego (patrz sekcja 4.2.2). Pozostałe dwa argumenty są opcjonalne. Pierwszym z nich jest liczba prób (domyslną wartością jest 10), natomiast drugi parametr jest flagą, której ustawienie decyduje o tym, czy dane zapisywane do dziennika powinny być również wyświetlane w oknie terminala.

Po przetworzeniu argumentów linii poleceń, skrypt *experiment.py* wykonuje kilka czynności mających przygotować grunt pod rozpoczęcie pętli eksperymentów. Wśród tych czynności jest między innymi wczytanie pliku konfiguracyjnego oraz inicjalizacja wymaganych zmiennych.

Po tym etapie następuje wykonanie pętli eksperymentów. Implementacja pętli wygląda następująco:

```
# --- Experiment sequence loop --- #
for trialCounter in range(numberOfTrials):
    for trackNumber in range(1, 4):
        experimentLog.Append("Generating data from 'train_de.py', track: " \
            "{0}, trial: {1}".format(trackNumber, trialCounter + 1))
        generateDataFromTraining(
            train_de,
            "DE",
            trackNumber,
            pathToConfigFile,
            buildPaths,
            experimentLog,
            dataCollector,
            minFitnessDict,
            isVerbose = options["--verbose"])

        experimentLog.Append("Generating data from 'train_pso.py', track: " \
            "{0}, trial: {1}".format(trackNumber, trialCounter + 1))
        generateDataFromTraining(
            train_pso,
            "PSO",
            trackNumber,
            pathToConfigFile,
            buildPaths,
            experimentLog,
            dataCollector,
            minFitnessDict,
            isVerbose = options["--verbose"])
```

Zewnętrzna pętla wykonuje się tyle razy, ile wynosi liczba prób. Dla każdej próby trening odbywa się na wszystkich trzech torach oferowanych przez obecne Środowisko Uczenia. Treningi odbywają się przy użyciu dwóch algorytmów - **Ewolucji Różnicowej** oraz **PSO**. Najważniejszym elementem powyższego kodu jest funkcja *generateDataFromTraining*, która przeprowadza treningi oraz dokonuje walidacji wytrenowanych modeli.

5.3 Analiza uzyskanych wyników

Eksperymenty zostały wykonane na stacji roboczej o specyfikacji opisanej w sekcji 3.2.7. Liczba prób wyniosła 30. Łączny czas obliczeń wyniósł 16 godzin, 23 minuty i 36 sekund, co daje średni czas obliczeń dla jednej próby na poziomie 32 minut i 47 sekund. Parametry algorytmów uczących miały następujące wartości:

1. Parametry dla Ewolucji Różnicowej

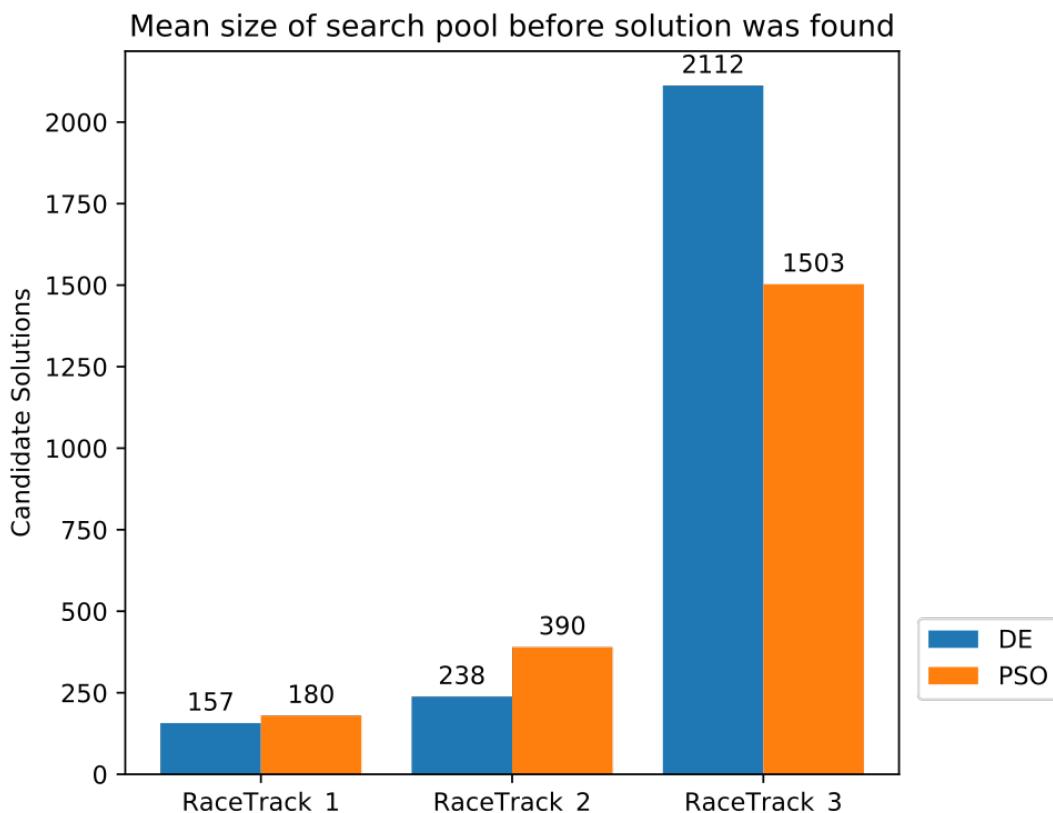
- Rozmiar populacji: 50 osobników
- Współczynnik mutacji: 0.8
- Prawdopodobieństwo krzyżowania: 0.7

2. Parametry dla algorytmu PSO

- Rozmiar populacji: 100 osobników
- $W = 0.729, c_1 = 2.05, c_2 = 2.05$

5.3.1 Liczba wygenerowanych rozwiązań

Rysunek 5.1 przedstawia średnie liczby wygenerowanych rozwiązań kandydackich przed odnalezieniem rozwiązania właściwego. Są to średnie liczone ze wszystkich przeprowadzonych prób. Każda średnia dotyczy danego toru i danego algorytmu uczącego. Im średnia jest mniejsza, tym szybciej model został wytrenowany. Wykres na rysunku 5.1 dokonuje porównania algorytmów uczących (czyli Ewolucji Różnicowej oraz PSO) dla poszczególnych torów.



Rysunek 5.1: Średnia liczba wygenerowanych rozwiązań kandydackich

Z wykresu można wysnuć następujące obserwacje:

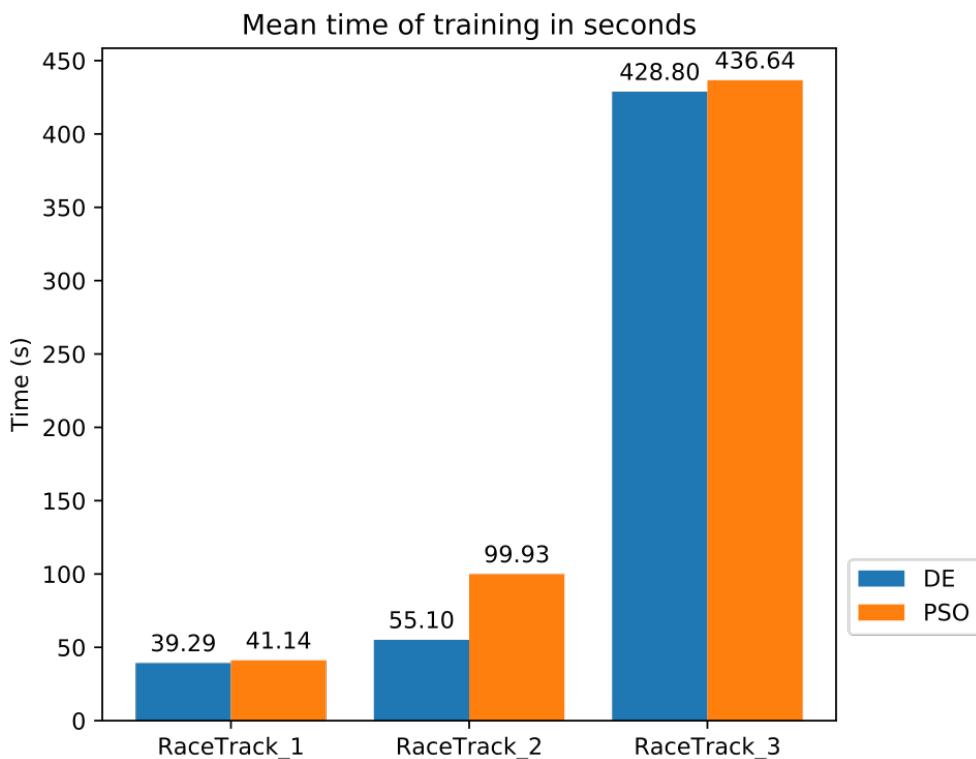
1. Liczba wygenerowanych rozwiązań rośnie wraz ze wzrostem poziomu trudności każdego toru. Tor *RaceTrack_3* jest znacznie bardziej trudny od pozostałych, dlatego wymaga znacznie więcej rozwiązań kandydackich zanim właściwe rozwiązanie zostanie odnalezione.
2. Na torze *RaceTrack_1* obydwa algorytmy radzą sobie porównywalnie. Niewielką przewagę posiada Ewolucja Różnicowa, która okazała się lepsza o zaledwie 7% od algorytmu PSO.
3. Na torze *RaceTrack_2* Ewolucja Różnicowa również jest lepsza, ale przewaga nad PSO jest znacznie większa. Wynosi aż 39%.
4. Wyniki uzyskane dla toru *RaceTrack_3* mogą trochę zaskakiwać, zwłaszcza biorąc pod uwagę wyniki z pozostałych torów. W tym przypadku, **Ewolucja Różnicowa jest znacznie gorsza od PSO**, a przewaga algorytmu PSO wynosi 29%.

Z powyższych obserwacji wynika, że pewne cechy torów wyścigowych „faworyzują” dany algorytm. Natomiast bardzo trudno jest w tej chwili wskazać, jakie to są cechy. Użycanie odpowiedzi na to pytanie wymagałoby znacznie głębszej analizy tematu, co wykracza poza zakres pracy.

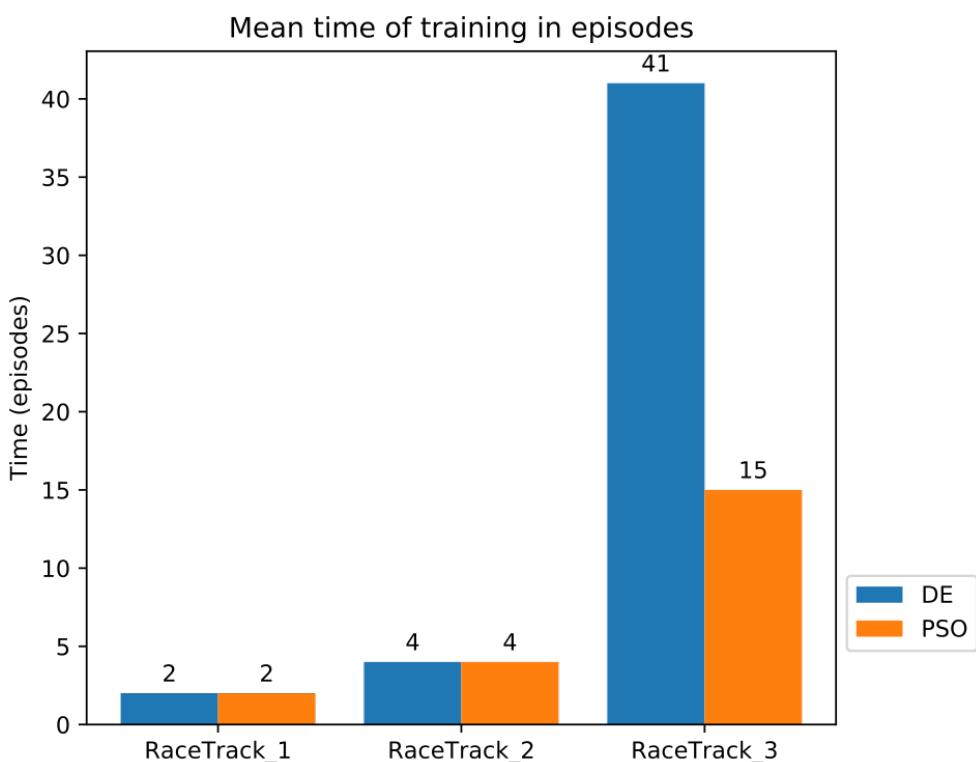
5.3.2 Czas treningu

Rysunek 5.2 przedstawia średnie czasy treningu liczone w sekundach. Czasy te są średnimi ze wszystkich prób. Każdy czas dotyczy treningu na konkretnym torze i przy użyciu konkretnego algorytmu uczącego. Obserwacje jakie można wysnuć z tego wykresu są następujące:

1. Czas treningu wzrasta wraz ze wzrostem poziomu trudności każdego toru. Tor *RaceTrack_3* jest znacznie bardziej trudny od pozostałych, dlatego wymaga znacznie więcej czasu na trening.
2. Ewolucja Różnicowa ma lepszy czas treningu na wszystkich torach wyścigowych, choć dla pierwszego i trzeciego toru różnice te nie są wielkie. Natomiast w przypadku toru *RaceTrack_2* różnica w czasie treningu jest znacznie większa i wynosi aż 45%.



Rysunek 5.2: Średni czas treningu w sekundach

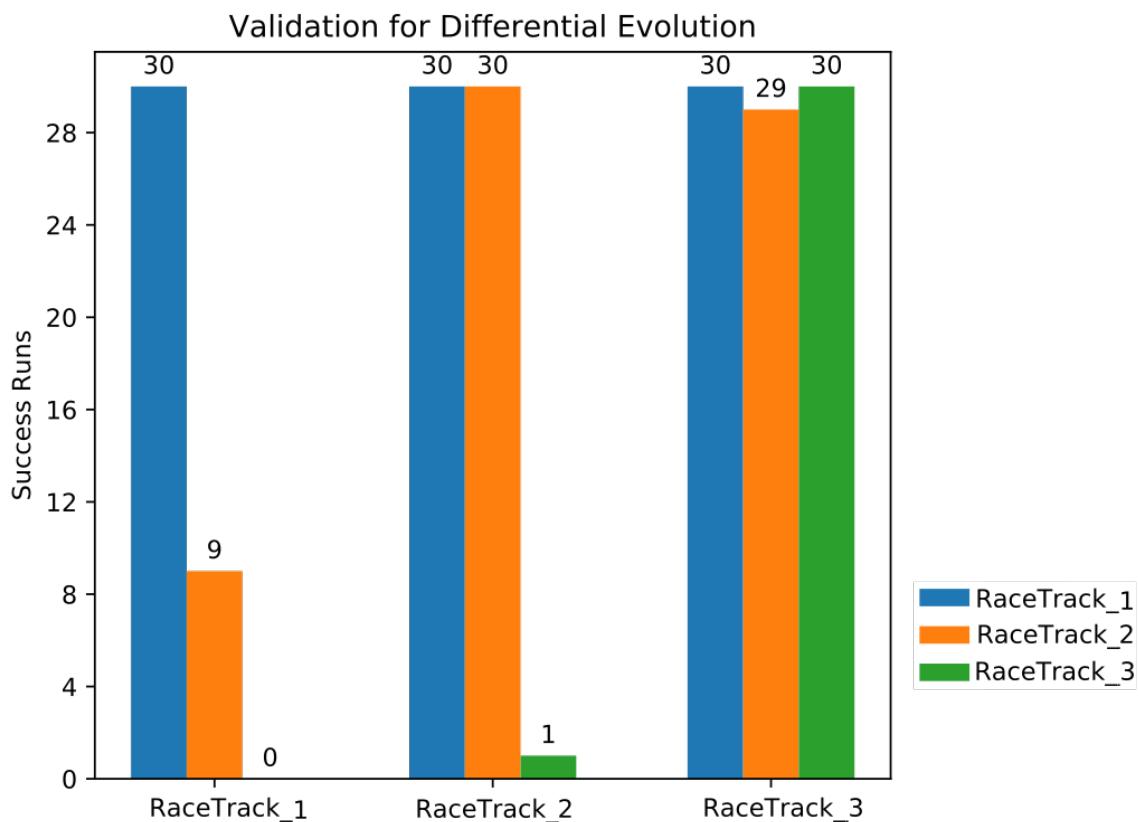


Rysunek 5.3: Średni czas treningu w epizodach

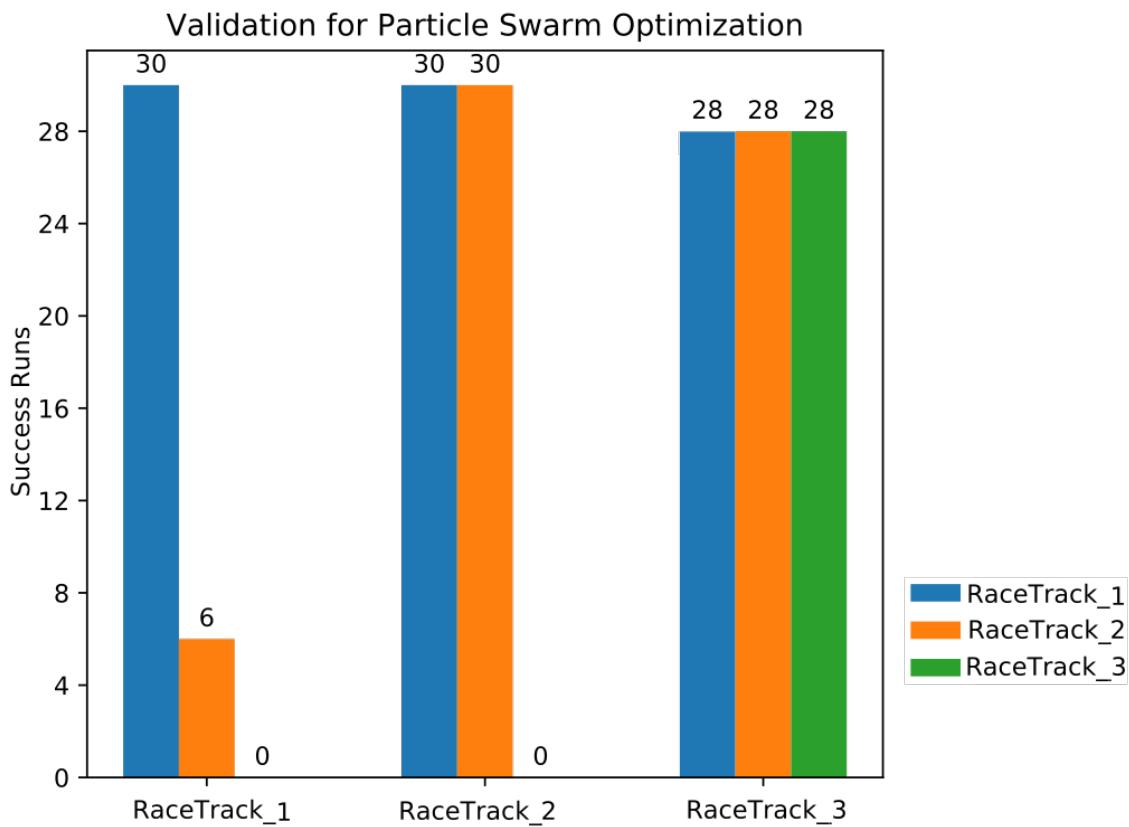
Z kolei na rysunku 5.3 zamieszczono średnie czasy treningu liczone w epizodach. Czasy te dla torów *RaceTrack_1* oraz *RaceTrack_2* są takie same, natomiast przy torze *RaceTrack_3* można zauważać spory rozstrzał pomiędzy wynikami dla poszczególnych algorytmów. Liczba epizodów potrzebna do wytrenowania modelu przy użyciu algorytmu PSO jest o **ponad połowę mniejsza** od liczby epizodów wymaganych przy Ewolucji Różnicowej. Biorąc jednak pod uwagę rysunek 5.2, każdy epizod treningu przy użyciu Ewolucji Różnicowej liczył się średnio o wiele szybciej od analogicznego epizodu dla algorytmu PSO.

5.3.3 Walidacja wyuczonych modeli

Rysunki 5.4 oraz 5.5 przedstawiają wyniki walidacji wytrenowanych modeli. Walidacja polega na sprawdzeniu wytrenowanego modelu na wszystkich torach wyścigowych. Walidacja występuje po zakończeniu każdego treningu. Liczby ponad słupkami oznaczają liczbę walidacji zakończonych sukcesem. Maksymalna liczba pozytywnych walidacji dla danego przypadku jest równa liczbie prób, zatem w tej sytuacji jest to 30.



Rysunek 5.4: Walidacja modeli wytrenowanych Ewolucją Różnicową



Rysunek 5.5: Walidacja modeli wytrenowanych algorytmem PSO

Etykiety na dole wykresów (*RaceTrack_1*, *RaceTrack_2*, *RaceTrack_3*) wyznaczają, na jakim torze modele były trenowane. Kolory słupków wyznaczają, na jakim torze modele były walidowane.

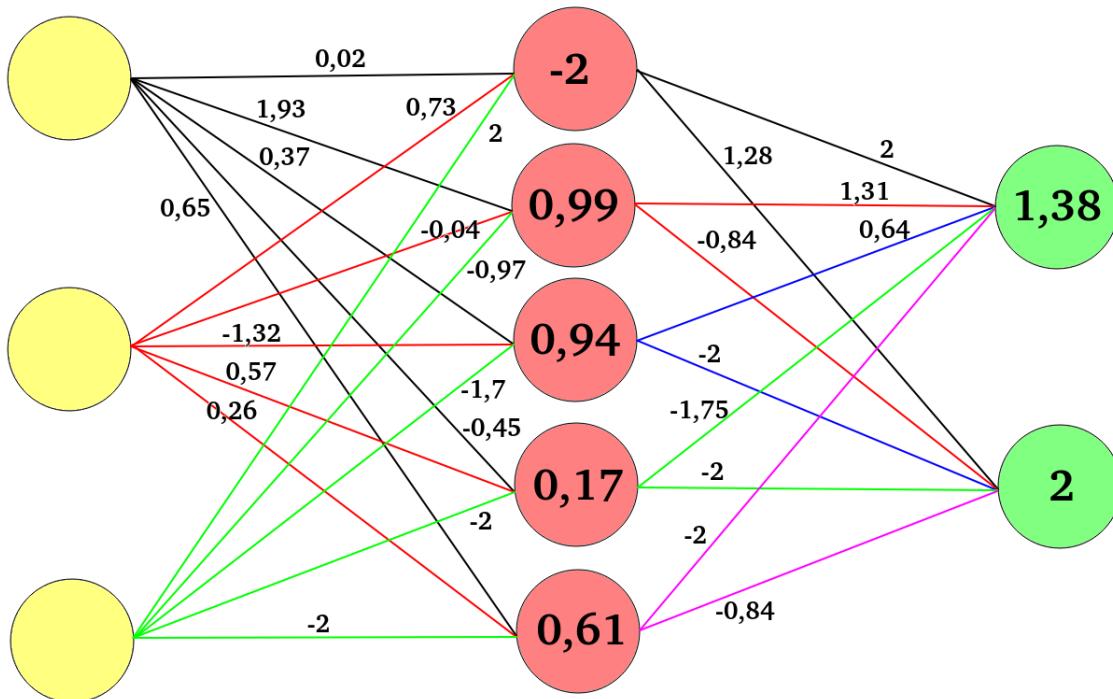
Obserwacje, jakie można wysnuć z rysunków, są następujące:

1. Modele wytrenowane na łatwiejszych torach rzadko przechodzą walidację na torach trudniejszych;
2. Modele wytrenowane na trudniejszych torach zazwyczaj dobrze radzą sobie z torami łatwiejszymi;
3. Ponieważ tor *RaceTrack_3* jest o wiele trudniejszy do wyuczenia się niż pozostałe tory, dlatego też tylko modele wytrenowane na tym torze potrafiły być tam pozytywnie walidowane. Wyjątkiem jest jeden model, wytrenowany na torze *RaceTrack_2* przy użyciu Ewolucji Różnicowej.
4. W przypadku algorytmu PSO, dwukrotnie doszło do sytuacji, w której model szkolony na torze *RaceTrack_3* nie zdążył się wytrenować na tyle dobrze, żeby móc pokonać choćby najprostszy z torów wyścigowych. Ten fakt pokazuje, że trening na trudniej-

szym torze utrudnia również wytrenowanie modelu radzącego sobie na torach łatwiejszych.

- Ponieważ tor *RaceTrack_2* nie jest dużo trudniejszy od toru *RaceTrack_1*, dlatego część modeli wytrenowanych na torze *RaceTrack_1* radziła sobie także na torze *RaceTrack_2*. W przypadku Ewolucji Różnicowej współczynnik ten wynosił 30%, natomiast w przypadku algorytmu PSO było to 20%.

5.4 Analiza wytrenowanego modelu



Rysunek 5.6: Przykład wytrenowanej sieci neuronowej

Rysunek 5.6 przedstawia wizualizację sieci neuronowej jednego z wyuczonych modeli. Jest to model wyuczony na torze *RaceTrack_3*. Na rysunku zostały zobrazowane parametry sieci. Wartości liczbowe zawarte wewnętrz neutronów to ich biasy, natomiast liczby znajdujące się przy krawędziach to wagi poszczególnych połączeń.

Na podstawie bezpośrednich obserwacji rysunku, trudno jest wyciągnąć wartościowe wnioski. Pewien wgląd na „strategię działania”, zakodowaną w parametrach sieci, daje nam rozważenie kilku scenariuszy testowych, czyli sytuacji które mogą się wydarzyć podczas nawigowania samochodem po Środowisku Uczenia. Tabela 5.1 przedstawia wyniki obliczone przez sieć neuronową dla wybranych danych wejściowych. Wartości danych wejściowych zostały dobrane pod kątem rozpatrzenia kilku podstawowych scenariuszy testowych.

Dane wejściowe	Wyniki obliczeń sieci	Scenariusz testowy	Zachowanie samochodu
[0.5, 1.0, 0.5]	[1.0, 1.0]	Prosta szeroka droga, samochód na środku drogi	Wyrównuje do prawej krawędzi drogi
[0.2, 1.0, 0.2]	[0.81, 0.24]	Prosta wąska droga, samochód na środku drogi	Wyrównuje do prawej krawędzi drogi
[1.0, 1.0, 0.3256]	[1.0, 0.0]	Prosta szeroka droga, samochód blisko prawej krawędzi	Jedzie prosto
[1.0, 0.2, 0.3]	[1.0, -0.82]	Zakręt w lewo	Skręca w lewo
[0.3, 0.2, 1.0]	[1.0, 1.0]	Zakręt w prawo	Skręca w prawo

Tabela 5.1: Zachowanie sieci dla wybranych scenariuszy drogowych

Na podstawie zawartości tabeli 5.1 można wysnuć wniosek, że wytrenowana sieć generuje poprawne wyniki. Potrafi skręcać we właściwą stronę podczas zakrętu. Potrafi też jechać prosto gdy tego wymaga sytuacja. Jedyną ciekawostką jest fakt, że omawiana sieć preferuje „trzymania się” blisko prawej krawędzi drogi. Taka właściwość sieci została nabyta podczas procesu uczenia się.

5.5 Wnioski z analiz

Analiza uzyskanych wyników pozwala na empiryczne potwierdzenie faktu zgodnego z intuicją. Im bardziej trudny tor, tym więcej czasu oraz obliczeń jest potrzebnych do wyuczenia na nim modelu. Statystyki dla torów *RaceTrack_1* oraz *RaceTrack_2* są zbliżone, ponieważ poziom trudności tych dwóch torów jest do siebie zbliżony. Natomiast statystyki dla toru *RaceTrack_3* znacznie odbiegają od reszty, ponieważ jest on dużo bardziej trudny od pozostałych torów.

Uzyskane wyniki pozwalają dowiedzieć się czegoś o wpływie Środowiska Uczenia na proces treningu. Niewiele natomiast mówią o samych algorytmach uczących. Wyciągnięcie bardziej wartościowych wniosków w tym zakresie wymagałoby dalszych, dogłębnych badań.

Podsumowanie

Celem pracy było opracowanie prostego systemu uczącego sieci neuronowe, bazującego na symulacjach przeprowadzanych w wymodelowanym środowisku. Zadaniem sieci neuronowych było sterowanie samochodem, poruszającym się po wirtualnym środowisku symulacyjnym. Sieci neuronowe były uczone za pomocą algorytmów PSO i Ewolucji Różnicowej.

Cel pracy został zrealizowany w całości. Stworzona aplikacja umożliwia przeprowadzanie treningu sieci neuronowych. Ewolucja Różnicowa (patrz sekcja 2.3.1) oraz algorytm PSO (patrz sekcja 2.3.2) były z sukcesami zastosowane do uczenia sieci. Wytrenowane sieci neuronowe potrafią pokonywać wymodelowane tory wyścigowe (patrz sekcja 4.1.2), nawet te o dużym stopniu skomplikowania. Więcej informacji o wynikach uczenia sieci można odnaleźć w rozdziale 5.

Perspektywy dalszych badań w dziedzinie

Działania przeprowadzone w ramach tworzenia niniejszej pracy stanowią zaledwie wstęp do badania omawianego zagadnienia. Uczenie maszynowe oraz tworzenie autonomicznych samochodów to dwa bardzo szerokie tematy, którym można poświęcić wiele lat badań.

Jednym z możliwych kierunków dalszego rozwoju byłoby przeprowadzenie eksperymentów na pojeździe poruszającym się po rzeczywistym środowisku. Takim pojazdem mógłby być model samochodu, wykonany w pewnej skali. Innym pomysłem wartym rozważenia jest poszerzenie percepcji samochodu poprzez dołączenie mechanizmu wizji komputerowej, o której opowiada artykuł [84]. Wykorzystanie tej technologii mogłoby w znaczący sposób zwiększyć możliwości tworzonego pojazdu.

Opinie i przemyślenia

Tematyka poruszana w tej pracy jest bardzo ważna i istotna, nie tylko dla osób zainteresowanych nowinkami technicznymi. Uczenie maszynowe oraz szerzej pojęta sztuczna inteligencja zmienią sposób, w jaki będzie funkcjonować społeczeństwo przyszłości. Będą miały istotny wpływ na życie każdego człowieka.

W ciągu najbliższych kilkunastu lat czeka nas wielka rewolucja przemysłowa, w wyniku której zniknie wiele wykonywanych obecnie zawodów [33]. W wyniku tego, sporo ludzi będzie zmuszonych do zmiany pracy.

Istotnym tematem jest także kwestia odpowiedzialnego wykorzystania zdobytych technologicznych. Wiele osób, uznawanych za wybitne w świecie nauki, ostrzegają przed skutkami nieetycznego wykorzystania sztucznej inteligencji. Do takich osób należy m.in. Elon Musk, który wielokrotnie wypowiadał się na ten temat [54]. Bardzo ważne, aby wszystkie osoby zajmujące się tą tematyką miały świadomość odpowiedzialności, jaka na nich spoczywa. Dotyczy to zwłaszcza polityków, uczestniczących w procesie tworzenia dokumentów legislacyjnych. Odpowiednie regulacje prawne z zakresu sztucznej inteligencji są niezbędne. Tylko dzięki nim jest szansa, że zwykły obywatel nie będzie czuł zagrożenia w sytuacji, w której nowoczesne technologie coraz bardziej wkraczają w jego życie codzienne.

Musimy zapewnić, aby sztuczna inteligencja pozostała narzędziem, które jest w naszej kontroli i które nie jest wykorzystywane do celów nieetycznych. Nie możemy dać się zasłonić możliwościami, jakie oferuje nam ta technologia. Nie możemy bagatelizować potencjalnych zagrożeń. Tylko dzięki wypracowaniu odpowiednich zabezpieczeń jest szansa, że sztuczna inteligencja nigdy nie odwróci się przeciwko ludzkości.

Spis rysunków

1.1	Poziomy automatyzacji sterowania samochodem (według SAE)	4
1.2	NVIDIA DRIVE AGX Pegasus	12
1.3	NVIDIA DRIVE AGX Xavier	12
1.4	Oprogramowanie wchodzące w skład pakietu NVIDIA DRIVE	14
1.5	Stopklatka z pracy symulatora AirSim	16
1.6	Architektura systemu Deepdrive	19
1.7	XiaoR GEEK XR-F2 - przykład gotowego do zakupu zestawu dla platformy Donkey Car	20
1.8	Stopklatka z pracy systemu	22
2.1	Model biologicznego neuronu	25
2.2	Model sztucznego neuronu	26
2.3	Przykład prostej sieci neuronowej	27
2.4	Obliczanie nowej pozycji cząstki	34
3.1	Wizualizacja architektury systemu	36
3.2	Uproszczony model frameworka Unity ML-Agents	40
4.1	Zdjęcie reklamujące pakiet „ <i>Low Poly Destructible 2 Cars no. 8</i> ”	47
4.2	Widok obiektu <i>CarLearningBrain</i> w inspektorze edytora Unity	50
4.3	Widok obiektu <i>CarPlayerBrain</i> w inspektorze edytora Unity	51
4.4	Struktura katalogowa modułu	56

5.1	Średnia liczba wygenerowanych rozwiązań kandydackich	66
5.2	Średni czas treningu w sekundach	68
5.3	Średni czas treningu w epizodach	68
5.4	Walidacja modeli wytrenowanych Ewolucją Różnicową	69
5.5	Walidacja modeli wytrenowanych algorytmem PSO	70
5.6	Przykład wytrenowanej sieci neuronowej	71

Bibliografia

- [1] About ROS. <https://www.ros.org/about-ros/>. Data dostępu: grudzień 2019.
- [2] AirSim GitHub repository. <https://github.com/Microsoft/AirSim>. Data dostępu: listopad 2019.
- [3] Deepdrive GitHub repository. <https://github.com/deepdrive/deepdrive>. Data dostępu: grudzień 2019.
- [4] DIY Robocars. <https://diyrobocars.com>. Data dostępu: listopad 2019.
- [5] docopt. <https://docopt.org>. Dokumentacja pakietu docopt.
- [6] Donkey Car - Roll Your Own Car. http://docs.donkeycar.com/roll_your_own. Data dostępu: listopad 2019.
- [7] Donkey Car - Supported cars. http://docs.donkeycar.com/supported_cars. Data dostępu: listopad 2019.
- [8] Donkey Car project documentation. <http://docs.donkeycar.com>. Data dostępu: listopad 2019.
- [9] donkeycar Github repository. <https://github.com/autorope/donkeycar>. Data dostępu: listopad 2019.
- [10] Low Poly Destructible 2 Cars no. 8. <https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-destructible-2-cars-no-8-45368>. Data dostępu: wrzesień 2019.
- [11] What is a Raspberry Pi? <https://opensource.com/resources/raspberry-pi>. Data dostępu: listopad 2019.

- [12] Conda. <https://docs.conda.io/en/latest>, 2017. Dokumentacja dla systemu conda. Data dostępu: październik 2019.
- [13] C Sharp (programming language). [https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)), 2019. Data dostępu: październik 2019.
- [14] Engine control unit. https://en.wikipedia.org/wiki/Engine_control_unit, 2019. Data dostępu: grudzień 2019.
- [15] Inertial measurement unit. https://en.wikipedia.org/wiki/Inertial_measurement_unit, 2019. Data dostępu: grudzień 2019.
- [16] ML-Agents Toolkit Overview. <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md>, 2019. Data dostępu: październik 2019.
- [17] Python (programming language). [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language)), 2019. Data dostępu: październik 2019.
- [18] PyTorch features overview. <https://pytorch.org/features>, 2019. Data dostępu: październik 2019.
- [19] Using Docker For ML-Agents. <https://github.com/Unity-Technologies/ml-agents/blob/0.9.1/docs/Using-Docker.md>, 2019. Data dostępu: październik 2019.
- [20] Samuel Artz. Applying Evolutionary Artificial Neural Networks. https://github.com/ArztSamuel/Applying_EANNs, 2016. Data dostępu: wrzesień 2019.
- [21] Samuel Artz. Deep Learning Cars. <https://arztsamuel.github.io/en/projects/unity/deepCars/deepCars.html>, 2016. Data dostępu: wrzesień 2019.
- [22] Carles Barrobés. DDT's documentation. <https://ddt.readthedocs.io/en/latest>, 2012. Dokumentacja dla modułu ddt.
- [23] Norbert Biedrzycki. Autonomiczne samochody wyprzedzają nasze systemy prawne. <https://businessinsider.com.pl/technologie/autonomiczne-pojazdy-a-przepisy-ruchu-i-prawa/e4jd3n4>, 2019. Data dostępu: październik 2019.
- [24] Pat Bowden. Beginners Guide to Massive Open Online Courses (MOOCs). <https://www.classcentral.com/help/moocs>, 2019. Data dostępu: grudzień 2019.

- [25] Oliver Cameron. Using Deep Learning to Predict Steering Angles. <https://medium.com/udacity/challenge-2-using-deep-learning-to-predict-steering-angles-f42004a36ff3>, 2016. Data dostępu: październik 2019.
- [26] Oliver Cameron. We're Building an Open Source Self-Driving Car. <https://medium.com/udacity/were-building-an-open-source-self-driving-car-ac3e973cd163>, 2016. Data dostępu: grudzień 2019.
- [27] 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: październik 2019.
- [28] Harsh Chauhan. NVIDIA Is Finally Seeing Financial Success in This Market. <https://www.fool.com/investing/2019/08/29/nvidia-is-finally-seeing-financial-success-in-this.aspx>, 2019. Data dostępu: październik 2019.
- [29] Connected. Three Approaches to Solving the Autonomous Vehicle Orientation Problem. <https://medium.com/connected/three-approaches-to-solving-the-autonomous-vehicle-orientation-problem-64862cce4491>, 2019. Data dostępu: listopad 2019.
- [30] James Dacombe. An introduction to Artificial Neural Networks (with example). <https://medium.com/@jamesdacombe/an-introduction-to-artificial-neural-networks-with-example-ad459bb6941b>, 2017. Data dostępu: listopad 2019.
- [31] Voyage Deepdrive. Leaderboard. <https://deepdrive.voyage.auto/leaderboard>, 2019. Data dostępu: grudzień 2019.
- [32] Grzegorz Dudek. Wykład na temat sztucznych sieci neuronowych jako aproksymatorów funkcji. <https://www.youtube.com/watch?v=sWPlLQgwxG8>, 2017. Data dostępu: listopad 2018.
- [33] Aleksandra Dzierżek. Nadchodzi czwarta rewolucja przemysłowa. Roboty będą walczyć z ludźmi? <https://forsal.pl/artykuly/903212,czwarta-rewolucja-przemyslowa-rewolucja-przemyslowa-walka-robotow-z-ludzmi.html>, 2015. Data dostępu: grudzień 2019.

- [34] Python Software Foundation. unittest — Unit testing framework. <https://docs.python.org/3.6/library/unittest.html>. Data dostępu: październik 2019.
- [35] Jake Frankenfield. Chief Technology Officer (CTO). <https://www.investopedia.com/terms/c/chief-technology-officer.asp>, 2019. Data dostępu: listopad 2019.
- [36] David Fumo. Types of Machine Learning Algorithms You Should Know. <https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861>, 2017. Data dostępu: listopad 2019.
- [37] Epic Games. Unreal Engine 4 - features. <https://www.unrealengine.com/en-US/features>. Data dostępu: grudzień 2019.
- [38] Jarosław Garbacz. ADAS - Zaawansowane systemy wspomagania kierowcy. <https://www.markoweszyby.pl/aktualnoci-i-produkty/159-adas-zaawansowane-systemy-wspomagania-kierowcy.html>. Data dostępu: listopad 2019.
- [39] GetFPV. All About Multirotor Drone FPV Flight Controllers. <https://www.getfpv.com/learn/new-to-fpv/all-about-multirotor-fpv-drone-flight-controller>, 2018. Data dostępu: październik 2019.
- [40] Drew Gray. Introducing Voyage Deepdrive. <https://news.voyage.auto/introducing-voyage-deepdrive-69b3cf0f0be6>, 2019. Data dostępu: grudzień 2019.
- [41] Dishashree Gupta. Fundamentals of Deep Learning – Activation Functions and When to Use Them? <https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them>, 2017. Data dostępu: listopad 2019.
- [42] Casper Hansen. Activation Functions Explained - GELU, SELU, ELU, ReLU and more. <https://mlfromscratch.com/activation-functions-explained>, 2019. Data dostępu: listopad 2019.
- [43] Avery Hartmans. How Google's self-driving car project rose from a crazy idea to a top contender in the race toward a driverless future. <https://businessinsider.com.pl/international/how-googles-self-driving-car-project-rose-from-a->

- crazy-idea-to-a-top-contender-in-the/yj2g2ng, 2016. Data dostępu: październik 2019.
- [44] Parallel Domain Inc. Parallel Domain website. <https://www.paralleldomain.com>, 2018. Data dostępu: grudzień 2019.
- [45] Ecma International. The JSON Data Interchange Syntax. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>, 2017. Data dostępu: wrzesień 2019.
- [46] ITDP. Three Revolutions in Urban Transportation. <https://www.itdp.org/2017/05/03/3rs-in-urban-transport/>, 2017. Data dostępu: listopad 2019.
- [47] Pawan Jain. Complete Guide of Activation Functions. <https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044>, 2019. Data dostępu: listopad 2019.
- [48] Ayoosh Kathuria. PyTorch 101, Part 1: Understanding Graphs, Automatic Differentiation and Autograd. <https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/>, 2019. Data dostępu: październik 2019.
- [49] J. Kennedy, R. Eberhart. Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, wolumen 4, 1995.
- [50] Oleksii Kharkovyna. Machine Learning vs Traditional Programming. <https://towardsdatascience.com/machine-learning-vs-traditional-programming-c066e39b5b17>, 2019. Data dostępu: listopad 2019.
- [51] Rafał Klaus. <http://www.cs.put.poznan.pl/rklaus/assn/neuron.htm>. Data dostępu: październik 2019.
- [52] Jérémie Labroquère, Aurélie Héritier, Annalisa Riccardi, Dario Izzo. Evolutionary Constrained Optimization for a Jupiter Capture. *Parallel Problem Solving from Nature – PPSN XIII*. Springer International Publishing, 2014.
- [53] Marcin Lasota. Jaki język programowania - Unity. <https://jaki-jezyk-programowania.pl/technologie/unity>, 2019. Data dostępu: październik 2019.

- [54] Bryan Logan. Elon Musk: Sztuczna inteligencja stanowi większe zagrożenie niż Korea Północna. <https://businessinsider.com.pl/technologie/elon-musk-ostrzega-przed-sztuczna-inteligencja/v63zvd0>, 2017. Data dostępu: grudzień 2019.
- [55] Iran Macedo. Implementing the Particle Swarm Optimization (PSO) Algorithm in Python. <https://medium.com/analytics-vidhya/implementing-particle-swarm-optimization-pso-algorithm-in-python-9efc2eb179a6>, 2018. Data dostępu: listopad 2019.
- [56] Vijini Mallawaarachchi. Introduction to Genetic Algorithms — Including Example Code. <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>, 2017. Data dostępu: październik 2019.
- [57] Audi MediaCenter. The new Audi A8 – conditional automated at level 3. <https://www.audi-mediacenter.com/en/on-autopilot-into-the-future-the-audi-vision-of-autonomous-driving-9305/the-new-audi-a8-conditional-automated-at-level-3-9307>, 2017. Data dostępu: listopad 2019.
- [58] Maad M. Mijwil. Artificial Neural Networks Advantages and Disadvantages. <https://www.linkedin.com/pulse/artificial-neural-networks-advantages-disadvantages-maad-m-mijwel>, 2018. Data dostępu: styczeń 2019.
- [59] MoDo. Tomorrow autonomous driving, today the Traffic Jam Pilot. <https://modo.volkswagengroup.it/en/mobotics/tomorrow-autonomous-driving-today-the-traffic-jam-pilot>, 2019. Data dostępu: grudzień 2019.
- [60] Charles Murray. Automakers Are Rethinking the Timetable for Fully Autonomous Cars. <https://www.plasticstoday.com/electronics-test/automakers-are-rethinking-timetable-fully-autonomous-cars/93993798360804>, 2019. Data dostępu: listopad 2019.
- [61] NVIDIA. Innovations by Automotive Industry Partners. <https://www.nvidia.com/en-us/self-driving-cars/partners/>. Data dostępu: październik 2019.
- [62] NVIDIA. NVIDIA DRIVE Mapping. <https://developer.nvidia.com/drive/drive-mapping>. Data dostępu: listopad 2019.

- [63] NVIDIA. NVIDIA DRIVE Perception. <https://developer.nvidia.com/drive/drive-perception>. Data dostępu: listopad 2019.
- [64] NVIDIA. NVIDIA DRIVE Planning. <https://developer.nvidia.com/drive/drive-planning>. Data dostępu: listopad 2019.
- [65] NVIDIA. NVIDIA TensorRT - Programmable Inference Accelerator. <https://developer.nvidia.com/tensorrt>. Data dostępu: listopad 2019.
- [66] NVIDIA. NVIDIA DRIVE - Software. <https://developer.nvidia.com/drive/drive-software>, 2019. Data dostępu: październik 2019.
- [67] NVIDIA. NVIDIA DRIVE - The Journey to zero accidents. <https://www.nvidia.com/en-us/self-driving-cars/drive-platform/>, 2019. Data dostępu: październik 2019.
- [68] Tim Peters. The Zen of Python. <https://www.python.org/dev/peps/pep-0020>, 2004. Data dostępu: październik 2019.
- [69] Bhagyashree R. Introducing Microsoft's AirSim, an open-source simulator for autonomous vehicles built on Unreal Engine. <https://hub.packtpub.com/introducing-microsofts-airsim-an-open-source-simulator-for-autonomous-vehicles-built-on-unreal-engine>, 2019. Data dostępu: listopad 2019.
- [70] Microsoft Research. Welcome to AirSim. <https://microsoft.github.io/AirSim>, 2018. Data dostępu: listopad 2019.
- [71] Pablo Rodriguez-Mier. A tutorial on Differential Evolution with Python. <https://pablormier.github.io/2017/09/05/a-tutorial-on-differential-evolution-with-python>, 2017. Data dostępu: listopad 2019.
- [72] Alexandria Sage. Waymo unveils self-driving taxi service in Arizona for paying customers. <https://www.reuters.com/article/us-waymo-selfdriving-focus-waymo-unveils-self-driving-taxi-service-in-arizona-for-paying-customers-idUSKBN1041M2>, 2018. Data dostępu: listopad 2019.
- [73] Shital Shah, Debadeepta Dey, Chris Lovett, Ashish Kapoor. Aerial Informatics and Robotics platform. Raport instytutowy MSR-TR-2017-9, Microsoft Research, 2017.

- [74] Krzysztof Sopyła. Dlaczego porzuciłem Tensorflow na rzecz Pytorch. <https://ksopyla.com/machine-learning/pytorch-vs-tensorflow>, 2019. Data dostępu: październik 2019.
- [75] Rainer Storn, Kenneth Price. Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.
- [76] Daniel Svozil, Vladimir Kvasnicka, Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 1997. Data dostępu: październik 2019.
- [77] synopsys. Dude, Where's My Autonomous Car? The 6 Levels of Vehicle Autonomy. <https://www.synopsys.com/automotive/autonomous-driving-levels.html>. Data dostępu: listopad 2019.
- [78] synopsys. What is an Autonomous Car? <https://www.synopsys.com/automotive/what-is-autonomous-car.html>. Data dostępu: listopad 2019.
- [79] Unity Technologies. Quick guide to the Unity Asset Store. <https://unity3d.com/quick-guide-to-unity-asset-store>. Data dostępu: wrzesień 2019.
- [80] Unity Technologies. Brain Properties. <https://github.com/Unity-Technologies/ml-agents/blob/0.9.1/docs/Learning-Environment-Design-Brains.md#brain-properties>, 2019. Data dostępu: wrzesień 2019.
- [81] Unity Technologies. Player Brain properties. <https://github.com/Unity-Technologies/ml-agents/blob/0.9.1/docs/Learning-Environment-Design-Player-Brains.md#player-brain-properties>, 2019. Data dostępu: wrzesień 2019.
- [82] Unity Technologies. Unity User Manual - Build Settings. <https://docs.unity3d.com/2019.1/Documentation/Manual/BuildSettings.html>, 2019. Data dostępu: październik 2019.
- [83] Fjodor van Veen. The Neural Network Zoo. <https://www.asimovinstitute.org/neural-network-zoo>, 2019. Data dostępu: wrzesień 2019.

- [84] Michelle Venables. An Overview of Computer Vision. <https://towardsdatascience.com/an-overview-of-computer-vision-1f75c2ab1b66>, 2019. Data dostępu: grudzień 2019.
- [85] Viknesh Vijayenthiran. Navya already sells fully self-driving cars, including in US. https://www.motorauthority.com/news/1118809_navya-already-sells-fully-self-driving-cars-including-in-us, 2018. Data dostępu: listopad 2019.
- [86] Wikipedia. Geometric design of roads. https://en.wikipedia.org/wiki/Geometric_design_of_roads. Data dostępu: październik 2019.
- [87] Alex Woodie. Machine Learning Market Demonstrates Solid Growth. <https://www.datanami.com/2019/11/19/machine-learning-market-demonstrates-solid-growth>, 2019. Data dostępu: grudzień 2019.
- [88] Devon Yarbrough. How to build a self-driving robot race car. <https://read.acloud.guru/how-i-built-an-autonomous-car-to-drive-awareness-for-women-who-code-bec91f94da91>, 2017. Data dostępu: listopad 2019.