UNIWERSYTET JANA KOCHANOWSKIEGO
W KIELCACH

WYDZIAŁ NAUK ŚCISŁYCH I PRZYRODNICZYCH
Kierunek: Inżynieria danych

Stepan Garciu
Numer albumu 135843

Praca inżynierska

Model pojazdu autonomicznego

Promotor pracy:
dr Janusz Krywult

Praca przyjęta pod względem
merytorycznym i formalnym
w formie papierowej i elektronicznej

…….
/data i podpis promotora/

KIELCE 2022

Spis treści

Spis rysunków

Spis tabel

Wstęp

Celem pracy jest zbadanie struktury konwolucyjnej sieci neuronowej, wykorzystanie jej do rozpoznawania obiektów na obrazie i na podstawie tego stworzenie modelu pojazdu autonomicznego. Na praktycznym przykładzie w symulatorze porównamy dwa modele, już znany model Nvidii dla autonomicznego pojazdu z moim własnym modelem z poprzedniego projektu. Otrzymane wyniki wyświetlimy na wykresie i wyciągniemy wnioski.

Praca składa się z 5 rozdziałów. Pierwszy zawiera analizę dziedziny i opis specyfikacji wymagań stawianych projektowanej symulacji. Drugi rozdział przedstawia realizację projektu w postaci sekwencyjnego wykonywania wszystkich niezbędnych kroków. Część trzecia jest kontynuacją rozdziału drugiego, która szczegółowo opisuje splotowe sieci neuronowe. Czwarty rozdział poświęcony jest symulacji testowej własnego modelu sieci neuronowej. A piąta część porównuje wyniki obu modeli sieci neuronowych.

Symulator Transportu Autonomicznego posłuży jako wirtualna platforma do uruchomienia projektu. Postaram się wykazać, że splotowe sieci neuronowe są w stanie prowadzić pojazd drogą bez ręcznego kontrolowania. Nawet niewielka ilość danych treningowych z wieloma okrążeniami jazdy powinna wystarczyć, aby nauczyć samochód jazdy po zadanej trasie. Pozytywne wyniki określą wydajność, niezawodność i skuteczność uczenia głębokiego.

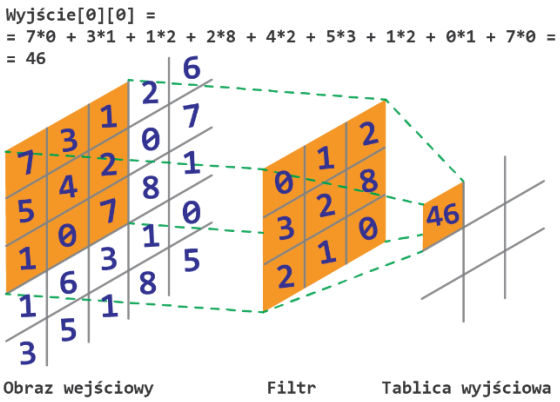
1. Analiza dziedziny i specyfikacja wymagań

Obecnie, pojazdy autonomiczne znajdują się w centrum zainteresowania gigantów technologicznych. Dzięki niezwykle szybkiej ewolucji technologii pomysł ten ostatnio przeszedł z „możliwego" na „dostępny komercyjnie". To, co kilka lat temu wydawało się science-fiction, teraz bardziej przypomina coś, co wkrótce będzie nieodłączną częścią życia. Powodem, dla którego mówię „wkrótce będzie", jest fakt, że chociaż firmy takie jak Tesla [1], Nvidia [2] i Nissan [3] mają oprogramowanie do autonomicznej jazdy, nadal wymagają człowieka, by obserwował drogę i przejmował kontrolę w razie potrzeby. Jednak fascynujące jest obserwowanie, jak daleko zaszliśmy w zakresie innowacji i jak szybko rozwija się technologia. Tak bardzo, że teraz, z pomocą podstawowego głębokiego uczenia, magii sieci neuronowych, możemy zbudować własny potok do autonomicznej jazdy samochodu.

Głębokie uczenie [4] jest jedną z głównych technologii, która umożliwiła samodzielną jazdę. W uczeniu głębokim model komputerowy uczy się wykonywać zadania klasyfikacyjne bezpośrednio na podstawie obrazów, tekstu lub dźwięku. Modele głębokiego uczenia mogą osiągnąć bardzo wysoką dokładność, czasami przewyższającą wydajność przeciętnego człowieka. Modele są szkolone przy użyciu dużego zestawu etykietowanych danych i architektury sieci neuronowych, które zawierają wiele warstw. Głębokie uczenie to technologia, która może pomóc rozwiązać prawie każdy rodzaj problemu naukowego lub inżynierskiego.

Idea stworzenia pełnej automatyzacji procesów bez ingerencji człowieka jest dość nowa, ale bardzo szybko zyskuje na popularności. Zainteresowanie tematem wynika z faktu, że przyspiesza on proces postępu technologicznego. Systemy autonomicznego pojazdu muszą działać zgodnie z zasadami zapisanymi w ludzkim systemie myślenia. Sposób, w jaki mózg przetwarza informacje, zasadniczo różni się od metod stosowanych w konwencjonalnych komputerach cyfrowych. Mózg to niezwykle złożony, nieliniowy, równoległy komputer (system przetwarzania informacji). Posiada zdolność organizowania swoich elementów strukturalnych, zwanych neuronami, tak aby mogły wykonywać określone zadania (rozpoznawanie wzorców, przetwarzanie sygnałów czuciowych, funkcje motoryczne) wielokrotnie szybciej niż najszybsze komputery.

Dopiero splotowe sieci neuronowe (CNN) [5] zrewolucjonizowały rozpoznawanie obrazów. Jest to algorytm, który jest używany do rozpoznawania i klasyfikowania różnych sygnałów wizualnych oraz podejmowania odpowiednich decyzji. Architektura splotowej sieci neuronowej jest analogiczna do wzorca łączności neuronów w ludzkim mózgu i została zainspirowana organizacją kory wzrokowej. Poszczególne neurony reagują na bodźce tylko w ograniczonym obszarze pola widzenia, znanym jako pole odbiorcze. Zestaw takich pól obejmuje cały obszar wizualny.



Rys. 1. Schemat konwolucyjnej sieci neuronowej

Jak pokazano na Rys. 1, każda wartość wyjściowa na mapie obiektów nie musi łączyć się z każdą wartością piksela w obrazie wejściowym. Wystarczy połączyć się z polem odbiorczym, w którym stosowany jest filtr. Ponieważ macierz wyjściowa nie musi mapować bezpośrednio każdej wartości wejściowej, warstwy splotowe (i łączne) są powszechnie nazywane warstwami „częściowo połączonymi”.

1.1. Sformułowanie zadania projektowego

Tematem projektu inżynierskiego jest stworzenie symulacji pojazdu autonomicznego. W pracy skoncentrujemy się na algorytmach głębokiego uczenia wykorzystujących splotowe sieci neuronowe (CNN). Jako wirtualną platformę do uruchamiania ruchu został wykorzystany Symulator Transportu Autonomicznego firmy Udacity [6]. Jest to oprogramowanie symulatora typu open source dla badaczy i deweloperów.

Najpierw zbieramy dane treningowe z symulatora w postaci obrazów tego, co będzie widział samochód podczas jazdy ze sterowaniem ręcznym, a następnie ładujemy obrazy treningowe do modelu głębokiego uczenia, aby mógł się za ich pośrednictwem uczyć. Wdrożony model uczenia głębokiego przyjmuje obrazy jako dane wejściowe, a kąt sterowania jako zmienną docelową. W modelu użyjemy danych z kamery jako danych wejściowych i spodziewamy się, że będzie on przewidywał kąt skrętu podczas samodzielnej jazdy.

Ważnym etapem projektu jest wstępne przetwarzanie zebranych obrazów w celu dokładniejszego szkolenia. Porównamy wydajność dwóch modeli głębokiego uczenia się w równych warunkach. Pierwszym z nich będzie popularny model sieci neuronowej firmy NVIDIA [7], który ma szerokie zastosowanie w komercyjnych projektach. A jego konkurentem będzie model sieci neuronowej, który wykorzystałem we własnym projekcie klasyfikacji odzieży [8].

1.2. Wymagania projektowe

W tym podrozdziale przedstawione są wymagania projektowe – funkcjonalne i niefunkcjonalne. W rozwoju produktu i optymalizacji procesu wymaganiem jest pojedyncza udokumentowana potrzeba fizyczna lub funkcjonalna, którą ma zaspokoić konkretny projekt, produkt lub proces. Ten etap jest powszechnie stosowany w sensie formalnym w projektowaniu inżynierskim, w tym na przykład w inżynierii systemów, inżynierii oprogramowania lub inżynierii przedsiębiorstwa.

1.2.1. Wymagania funkcjonalne

W inżynierii oprogramowania wymaganie funkcjonalne definiuje funkcję systemu lub jego komponentu, gdzie funkcja jest opisana jako specyfikacja zachowania między wejściami i wyjściami. Może ona obejmować obliczenia, przetwarzanie danych oraz szczegóły techniczne, które system musi spełniać.

Wymagania funkcjonalne składają się z 12 etapów:

1. Zbieranie danych
2. Importowanie
3. Wizualizacja i bilansowanie
4. Przygotowanie danych - podział danych do treningu i testowania
5. Augmentacja danych
6. Przetwarzanie wstępne
7. Generator wsadowy
8. Tworzenie modelu Nvidii
9. Tworzenie własnego modelu
10. Zapisywanie i wykreślanie modeli
11. Trening i testowanie
12. Porównanie obu modeli

Musimy najpierw zebrać dane, jadąc drogą kilka razy w jedną i drugą stronę. Następnie importujemy zebrane dane do naszego wirtualnego środowiska wraz z niezbędnymi bibliotekami. Wizualizujemy je, aby sprawdzić, jak dane są zrównoważone.

Podczas przygotowywania danych dzielimy je na dwie grupy. Dane zostaną podzielone w proporcji 80% dla treningu i 20% dla testowania. Podczas procesu augmentacji danych zastosowano techniki zwiększania ilości danych poprzez dodawanie nieznacznie zmodyfikowanych kopii. Wykorzystamy również technikę przetwarzania wstępnego, co pozwoli przekonwertować obrazy do wygodnego formatu. Generator wsadowy pozwoli nam łączyć dane w partiach, co ułatwia wysyłanie ich do naszego modelu.

Następnym krokiem jest stworzenie modeli do uczenia sieci neuronowych. Jako pierwszego użyjemy modelu Nvidii, który jest wykorzystywany w projektach komercyjnych [2]. Drugim będzie mój model, który bazuje na własnym projekcie kategoryzacji odzieży [8]. Wytrenowane modele zostaną zapisane w osobnych plikach. Następnie przychodzi najważniejszy etap: testowanie wydajności w wirtualnej symulacji. Na koniec porównamy oba modele pod kątem jakości ich użyteczności w równych warunkach.

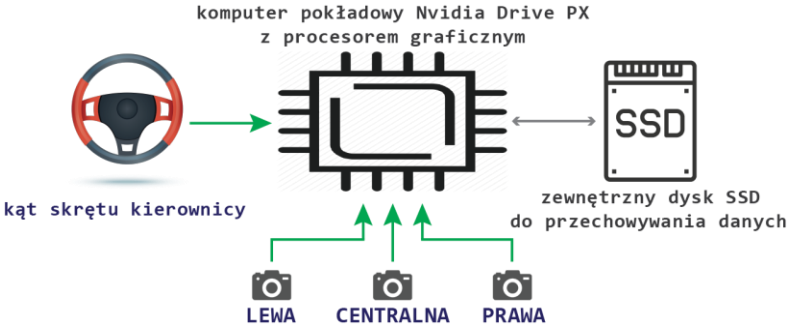
1.2.2. Wymagania niefunkcjonalne

Wymaganie niefunkcjonalne to wymaganie, które określa kryteria, które można wykorzystać do oceny działania systemu, a nie określone zachowania. Są one przeciwstawiane wymaganiom funkcjonalnym, które definiują określone zachowanie lub funkcje. Plan wdrożenia wymagań funkcjonalnych jest szczegółowo opisany w prezentowanym projekcie systemu. Plan implementacji wymagań niefunkcjonalnych jest uszczegółowiony w architekturze systemu, ponieważ są to zazwyczaj wymagania istotne architektonicznie.

W pracy chcemy wykazać, że splotowe sieci neuronowe są w stanie prowadzić pojazd drogą bez ręcznego kontrolowania. Nawet niewielka ilość danych treningowych z wieloma okrążeniami jazdy powinna wystarczyć, aby nauczyć samochód jazdy po zadanej trasie. Pozytywne wyniki określą wydajność, niezawodność i skuteczność uczenia głębokiego.

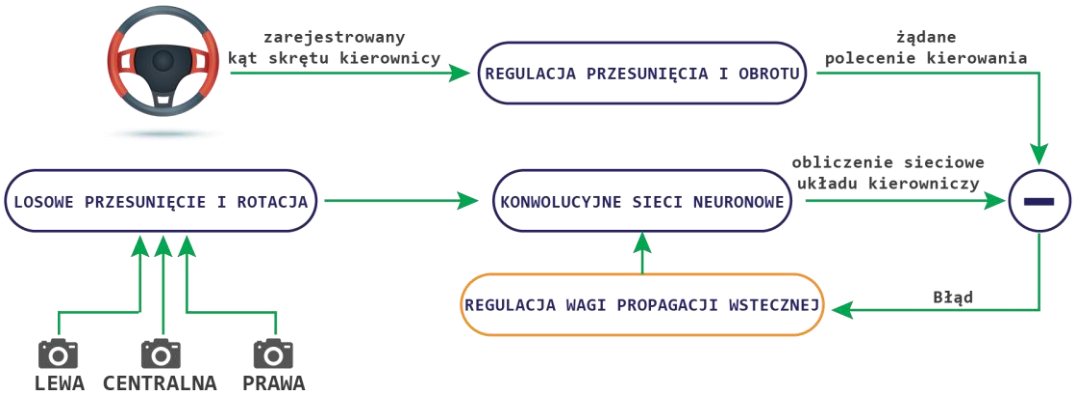
1.3. Przegląd gotowych rozwiązań

Pomysł na ten projekt został zapożyczony od firmy NVIDIA [2]. Wykorzystali oni własny model do trenowania na danych zebranych z prawdziwych samochodów i uzyskali obiecujące wyniki, gdy jeździli nim autonomicznie. Struktura systemu metody NVIDIA jest następująca, 3 kamery zbierają obrazy z kamer i rejestrują kąt pozycyjny tych obrazów. W tym projekcie deweloperzy wykorzystują 9-warstwową sieć splotową. Do trenowania modelu używane są dane z 72 godzin jazdy kierowców w różnych warunkach [7].



Rys. 2. Widok wysokiego poziomu systemu gromadzenia danych

Rysunek 2 pokazuje uproszczony schemat blokowy systemu zbierania danych uczących dla DAVE-2 [9]. Za przednią szybą samochodu do akwizycji danych zamontowane są 3 kamery. Obraz z kamer ze znacznikiem czasu jest rejestrowany jednocześnie z kątem skrętu zastosowanym przez kierowcę.



Rys. 3. Trening sieci neuronowej

Bardziej szczegółowy opis przedstawiono na Rys. 3, na którym widzimy, że następnie obrazy są przesyłane do splotowej sieci neuronowej (CNN), która oblicza proponowane polecenie sterujące. Zasugerowane przez nią polecenie jest porównywane z żądanym poleceniem dla tego obrazu, a wagi sieci neuronowej są dostosowywane, aby zbliżyć wyjście CNN do pożądanego działania. Regulacja wagi odbywa się za pomocą propagacji wstecznej [4].

Cały ten ciągły strumień danych wymaga ogromnej mocy obliczeniowej. Z tego powodu w moim projekcie do generowania poleceń sterujących będę wykorzystywał tylko jedną centralną kamerę, według schematu ogólnego z Rys. 4. Podstawowa architektura kodu została zapożyczona z 3 projektów ze strony Github: [10], [11], [12].

Rys. 4. Ogólny schemat szkolenia pojazdu autonomicznego zastosowany w moim projekcie

Do realizacji projektu nie musimy zajmować się projektowaniem gier i tworzeniem od podstaw symulacji jazdy samochodem. Organizacja Udacity zadbała o to i niedawno udostępniła swój własny symulator samochodu, pierwotnie zbudowany w celach edukacyjnych. Wystarczy go pobrać z sieci [6] i wykorzystać jako platformę do zbierania danych, a następnie sprawdzania skuteczności treningu sieci neuronowej.

Wadą tego rozwiązania jest brak dużej ilości wirtualnej przestrzeni do testowania modelu w różnych warunkach. Jednak dostępny jej rozmiar w zupełności wystarczy, aby uczyć i utrwalać umiejętności projektantów w wirtualnym środowisku Udacity's Self-Driving Car Simulator [6], w którym zapewnione są podstawowe prawa fizyki. W ten sposób są w stanie zaprojektować model pojazdu autonomicznego.

1.4. Analiza wymagań

Jak więc ludzie uczą się jeździć? Czy obliczamy, o ile skręcić na podstawie obserwowanego pasa drogi? Czy muszą to być pasy w określonym kolorze? A jeśli nie ma pasów drogowych? Cóż, nadal będziemy mogli jeździć. Więc pytanie brzmi, jak się nauczyliśmy?

Od wielu lat monitorujemy i zbieramy dane dotyczące zachowań kierowców na drodze. Dzięki temu mamy podstawową wiedzę na temat dróg, pasów, różnych znaków i wielu innych rzeczy. Czyli w tym momencie mamy już w głowie model, który zna podstawy, ale jak już zaczniemy samodzielnie jeździć, trenujemy model jeszcze dalej, tym razem będzie dodatkowa informacja o sterowaniu wraz z przyspieszaniem i hamowaniem. Tak więc w ciągu kilku dni szkolenia uczymy się jeździć. Z biegiem czasu stajemy się coraz lepsi, ponieważ otrzymujemy więcej danych. Kluczem jest tutaj to, że zbieramy dane i na ich podstawie tworzymy model, który uogólnia sposób jazdy.

Koncepcja uczenia głębokiego opiera się na tej samej zasadzie. Sieć neuronowa jest szkolona na podstawie już istniejących zebranych danych, wchłaniając całe przeżyte doświadczenie w procesie ich zbierania. Doświadczenie jest przetwarzane, tworząc model podejmowania właściwych decyzji w każdej nowej sytuacji.

Teraz stajemy przed zadaniem, w oparciu o już istniejące narzędzia, stworzyć sieć neuronową do trenowania autonomicznego transportu. Przede wszystkim system autonomicznej jazdy to taki, który pozwala pojazdowi na bezpieczną jazdę po drogach publicznych bez ingerencji człowieka. Terminy takie jak jazda autonomiczna, samochód bez kierowcy i samochód autonomiczny opisują tę samą technologię: komputer-mózg i czujniki, które mogą prowadzić pojazd zamiast człowieka, przynajmniej w pewnych warunkach.

Autopilot nie jest luksusem, ale koniecznością. Według rocznej statystyki wypadków drogowych na świecie [13]: „Około 1,3 miliona ludzi ginie każdego roku w wyniku wypadków drogowych. Od 20 do 50 milionów ludzi doznaje urazów innych niż śmiertelne, a wiele z nich doznaje niepełnosprawności w wyniku urazu.”

Elon Musk stwierdził, że Tesla z włączonym Autopilotem daje prawie 10 razy mniejszą szansę na wypadek niż w „przeciętnym pojeździe”. Oto dane, na których opiera to twierdzenie. Od 2018 r. Tesla publikuje kwartalny raport porównujący liczbę mil przypadających na wypadek na włączonym i wyłączonym autopilocie. Przedstawiony poniżej raport bezpieczeństwa Tesli [14] za drugi kwartał 2021 r. stwierdza:

„W drugim kwartale odnotowaliśmy jedną awarię na każde 4,41 miliona przejechanych mil, podczas których kierowcy korzystali z technologii Autopilot (autosterowanie i aktywne funkcje bezpieczeństwa).

W przypadku kierowców, którzy nie korzystali z technologii Autopilot (brak funkcji automatycznego sterowania i aktywnego bezpieczeństwa), zarejestrowaliśmy jedną awarię na każde 1,2 miliona przejechanych mil. Dla porównania, najnowsze dane NHTSA pokazują, że w Stanach Zjednoczonych co 484 000 mil dochodzi do wypadku samochodowego."

Sztuczna inteligencja znacznie dokładniej niż ludzki mózg analizuje to, co dzieje się wokół nas. Potrafi podejmować decyzje w sposób zrównoważony, bez uprzedzeń emocjonalnych. Głównym wymaganiem jakie stawiam przed moim projektem jest przestudiowanie i stworzenie modelu uczenia maszynowego do symulacji bezpiecznej jazdy po drogach.

2. Realizacja projektu

Czas rozpocząć realizację projektu. Wszystko, co jest do tego niezbędne, zostanie opisane etapami poniżej.

2.1. Środowisko programistyczne Jupyter Notebook

Jupyter Notebook [15] to środowisko programistyczne, w którym od razu widać wynik wykonania kodu i jego poszczególnych fragmentów. Różnica w stosunku do tradycyjnego środowiska programistycznego polega na tym, że kod można dzielić na kawałki i wykonywać w dowolnej kolejności. W takim środowisku programistycznym można np. napisać funkcję i od razu przetestować jej działanie, bez uruchamiania całego programu.

2.1.1. Tworzenie i konfiguracja wirtualnego środowiska

Zanim zaczniemy, powinniśmy zrozumieć, czym jest wirtualne środowisko i dlaczego jest nam potrzebne? Wirtualne środowisko to izolowana działająca kopia Pythona. Oznacza to, że każde środowisko może mieć własne zależności, a nawet własne wersje Pythona. Jest to przydatne, jeśli potrzebujemy różnych wersji Pythona lub pakietów dla różnych projektów. Utrzymuje to również porządek podczas testowania pakietów i upewniania się, że główna instalacja Pythona pozostaje sprawna. Wirtualne środowisko w Jupiter Notebook zostało utworzone na podstawie porad zawartych w tym artykule [16].

2.1.2. Importowanie wymaganych pakietów i bibliotek

Po wyborze środowiska programistycznego, przy użyciu CMD dodałem wymagane pakiety (Tabela 1) za pomocą następujących poleceń:

Tabela 1. Polecenia instalacji wymaganych pakietów

pip install opencv-python	biblioteka algorytmów przetwarzania obrazów
pip install numpy	praca z tablicami wielowymiarowymi
pip install pandas	manipulacja danymi Pandas jest oparta na bibliotece NumPy
pip install matplotlib	biblioteka do wizualizacji danych w grafice dwuwymiarowej
pip install sklearn	biblioteka uczenia maszynowego
pip install imgaug	zwiększenie liczby obrazów do projektów uczenia maszynowego
pip install tensorflow	biblioteka do budowy i treningu sieci neuronowej
pip install tensorflow-gpu	trening sieciowy na procesorze z GPU
pip install eventlet	pozwala zmieniać sposób uruchamiania kodu
pip install Flask	framework do tworzenia aplikacji internetowych
pip install python-socketio	protokół transportowy, który umożliwia dwukierunkową komunikację między klientami a serwerem

W kolejnym kroku zostały zaimportowane niezbędne biblioteki do realizacji badanego projektu w postaci poniższego kodu:

```
import os
import cv2
import keras
import numpy
import pandas
import random
import matplotlib

# WYŚWIETLANIE OBRAZÓW
from IPython.display import Image

# TWORZENIE MODELU
from imgaug import augmenters
# Augmentery same nie stosują rozszerzeń, ale są potrzebne do użycia meta.
from matplotlib import pyplot
from sklearn.utils import shuffle
from tensorflow.keras.optimizers import Adam
#Optymalizacja Adam to stochastyczna metoda opadania gradientu oparta na adaptacyjnej
estymacji momentów pierwszego i drugiego rzędu.
from tensorflow.keras.models import Sequential
# Model sekwencyjny jest odpowiedni dla zwykłego stosu warstw, w którym każda warstwa
ma dokładnie jeden tensor wejściowy i jeden tensor wyjściowy.
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.layers import Conv2D, Flatten, Dense, MaxPooling2D
# Warstwy do tworzenia sieci neuronowej

# TRENING MODELU
from sklearn.model_selection import train_test_split

# SYMULACJA TESTOWA
import socketio
import eventlet
from flask import Flask
from tensorflow.keras.models import load_model
import base64
from io import BytesIO
from PIL import Image

# OKREŚLENIE CZASU OBLICZEŃ
from time import time

# ZAPISYWANIE DANYCH W FORMACIE .CSV
import csv
```

Następnie zostaje zainicjowany moduł wyświetlania obrazów oraz obsługi plików. Główna część kodu mojego programu bazuje na bibliotece uczenia maszynowego TensorFlow [17]. Jej celem jest stworzenie sieci neuronowej klasyfikacji obrazów, co będzie wykorzystywane podczas treningu transportu autonomicznego. Biblioteki te są potrzebne do symulacji jazdy po wytrenowaniu modelu. Wynik treningu każdego modelu zostaje zapisany w osobnym pliku.

2.2. Dane badawcze

Niezależnie od dziedziny studiów lub preferencji w definiowaniu danych (jakościowych lub ilościowych), staranne zbieranie danych ma zasadnicze znaczenie dla integralności badania. Dobór odpowiednich narzędzi do gromadzenia danych (istniejących, zmodyfikowanych lub specjalnie zaprojektowanych), a także jasno określone instrukcje prawidłowego korzystania z narzędzi ograniczają możliwość wystąpienia błędów.

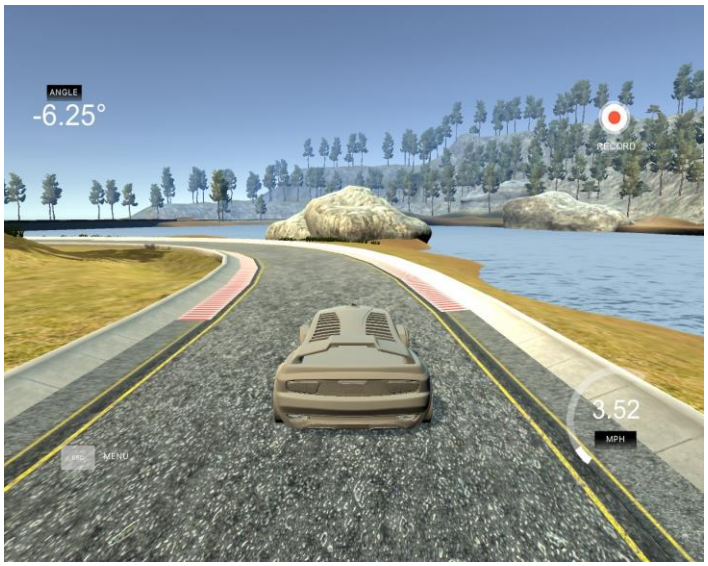
2.2.1. Zbieranie danych

W celu zebrania niezbędnych danych do trenowania sieci neuronowej i późniejszej wirtualnej symulacji wykorzystamy open source'owy symulator samochodu samojezdnego firmy Udacity [18]. Korzystając z tego symulatora najpierw poprowadzimy samochód po zadanej trasie i zbierzemy dane. Następnie wytrenujemy model CNN, aby nauczyć pojazd jeździć na tym torze i przetestujemy go ponownie na symulatorze.

Cała trasa przedstawia zamkniętą pętlą drogi o licznych zakrętach i różnych stopniach łuku. Na drodze są umieszczone pasy o różnych kształtach i typach. Sama droga ma różną nawierzchnię, od piasku po asfalt. W tle widać drzewa i góry. Ta różnorodność środowisk pozwala na wzbogacenie zbioru danych. Cala trasa była okrążana w obu kierunkach, w każdym z nich wykonano 5 przejazdów.

Wyświetlanie zrzutu ekranu w Pythonie realizuje wywołanie funkcji:

```
Image("TrainingMode.jpg", width=450, height=450)
```



Rys. 5. Zrzut ekranu trasy treningowej wirtualnego środowiska z Udacity

2.2.2. Importowanie zbioru danych

filename()

Potrzebujemy funkcji *filename()*, która podzieli ścieżkę danych na podstawie ukośnika i wyciągnie nazwę pliku jako ostatni element. Interesuje nas tylko ostatni element, który możemy bezpiecznie wydobyć poprzez:

```
def filename(fpath): # argumentem będzie ścieżka do pliku
    return fpath.split('\\')[-1]
```

Zebrane w trakcie jazdy dane reprezentują 7 kolumn. Pierwsze trzy kolumny to obrazy z 3 kamer (lewa, prawa i centralna), które robią zdjęcia podczas jazdy. Kolejne kolumny zawierają informacje o różnych właściwościach pojazdu (kąt skrętu, prędkość, itp.)

```
def FullAmountOfInformation(dataPath):
    # Definiujemy nazwy kolumn, które posiadamy
    usecols = ['Srodek', 'Lewa strona', 'Prawa strona', 'Uklad kierowniczy',
               'Przepustnica', 'Hamulec', 'Predkosc']

    # Importujemy informacje z pliku driving_log.csv
    data = pandas.read_csv(os.path.join(dataPath, 'data.csv'), names=usecols)
      # Plik CSV zawiera zwykły tekst, który może być odczytywany przez Pandas

    data['Srodek'] = data['Srodek'].apply(filename)
      # Zmienna data z indeksem ‘Srodek’ przyjmuje wartość nazwy pliku obrazu
    # Wypisujemy całkowitą liczbę zdjęć, które posiadamy
    print('Całkowita liczba zaimportowanych obrazów: ', data.shape[0])
    return data
```

Następnie importujemy zebrane w trakcie jazdy dane:

```
dataPath = 'F:/UJK/PracaDyplomowa/myData'
data = FullAmountOfInformation(dataPath) # Importowanie ścieżki
```

Po wykonaniu tej czynności program zwraca informację, że:

```
Całkowita liczba zaimportowanych obrazów: 9777
```

Do analizy porównawczej treningu i symulacji będziemy potrzebować informacji o rozmiarze importowanych danych:

```
dataLimit = data.shape[0]
dataLimit
```

```
9777
```

W wyniku wykonania kodu otrzymujemy informację, że rozmiar importowanych danych wynosi 9777 rekordów.

2.2.3. Normalizacja danych

balancedDataset()

Potrzebujemy funkcji balancedDataset(), aby uzyskać zrównoważony zbiór danych. Będzie dużo powtarzających się danych o pozycji zerowej (które zostaną usunięte), bo przez większość czasu samochód jedzie prosto. Musimy również upewnić się, że dane po prawej stronie są prawie takie same, jak dane po lewej stronie, ponieważ dane zbierane w obu kierunkach w sumie dają wypadkowy wynik wykorzystywany w procesie uczenia maszynowego pojazdu. Dane będą wysyłane w małych partiach, aby nie przeciążać funkcji. Ważnym aspektem jest wartość (True/False) argumentu *inplace* funkcji *drop()*.

Tabela 2. Dwa stany dla argumentu *inplace*

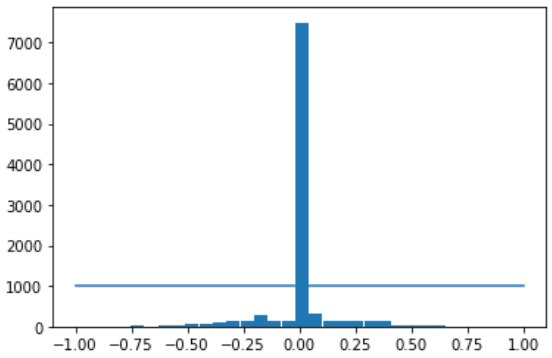
inplace = True	dane są modyfikowane w miejscu, co oznacza, że nic nie zwracają, a ramka danych jest teraz aktualizowana.
inplace = False	co jest wartością domyślną, operacja jest wykonywana i zwraca kopię obiektu.

Funkcja *balancedDataset()* posiada możliwość wizualizacji wstępnego (Rys. 6) i znormalizowanego (Rys. 7) wyniku. Jeśli argumentem *display* jest True, wtedy wykresy, takie jak pokazane na Rysunkach 6 i 7, zostaną wyświetlone.

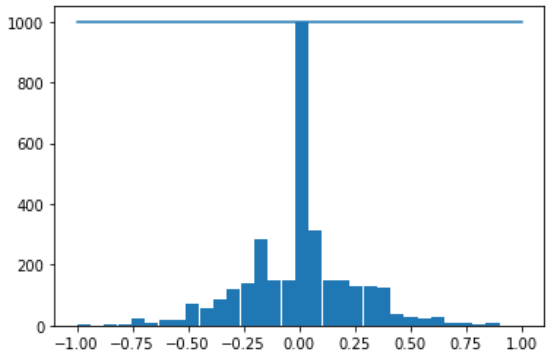
Wizualizacje i transfer danych do zmiennej otrzymujemy, wywołując tę funkcję:

```
data = balancedDataset(data, display=True)
# Jeśli display = True, wtedy zostaną wyświetlone wykresy
```

```
Usunięte obrazy:   6490
Pozostałe obrazy:  3287
```



Rys. 6. Histogram *wszystkich* położeń kierownicy zarejestrowanych w trakcie jazdy treningowej



Rys. 7. Histogram *oczyszczonych* położeń kierownicy zarejestrowanego w trakcie jazdy treningowej

W wyniku normalizacji udało się uzyskać zrównoważony zbiór danych. Powtarzające się dane (w ilości 6490 sztuk) o pozycji zerowej zostały usunięte. Z całkowitej liczby danych (9777 szt.) do bardziej wydajnej analizy pozostało 3287 obrazów.

```
def balancedDataset(data, display = True): # Właściwość wyświetlania wykresów

    num_bins = 31 # Dane będą wysyłane w małych partiach
    sampBin = 1000
    histogr, bins_ = numpy.histogram(data['Uklad kierowniczy'], num_bins)

    if display: #Wstępne wyświetlenie wykresu przed usunięciem niepotrzebnych danych
        centerGraph = (bins_[:-1] + bins_[1:])*0.5
        pyplot.bar(centerGraph, histogr, width = 0.06)
        pyplot.plot((-1, 1),(sampBin, sampBin))
        pyplot.show()

    # Usunięcie zbędnych danych
    deleteListIndex = [] # Lista danych, które chcemy usunąć
    for j in range(num_bins):
        binMeasurementsList  = [] # Lista wszystkich wartości w koszu
        for i in range(len(data['Uklad kierowniczy'])):
            # Sprawdzanie wartości pod kątem zbiegu okoliczności
            if data['Uklad kierowniczy'][i] >= bins_[j] and
data['Uklad kierowniczy'][i] <= bins_[j+1]:
                binMeasurementsList.append(i) # Dołączanie numeru indeksu
        # Tasujemy wartości naszej listy danych z koszyka
        binMeasurementsList = shuffle(binMeasurementsList)

        # Dzielimy nasze wartości z maksymalną liczbą wartości (sampBin = 1000)
        # sampBin => oznacza to maksymalną liczbę jaką możemy uzyskać
        binMeasurementsList = binMeasurementsList[sampBin:]
        # Umieściliśmy go z powrotem na naszej liście usuwania indeksów,
        #    więc możemy go zapisać dla każdego kosza

        # używamy .extend, ponieważ .append nie będzie działać w tym przykładzie
        deleteListIndex.extend((binMeasurementsList))

    # możemy sprawdzić, ile danych usunęliśmy, a ile zostało
    print('Usunięte obrazy: ', len(deleteListIndex))

    # Definiujemy każdy z indeksów, które chcemy usunąć
    data.drop(data.index[deleteListIndex], inplace=True)
    print('Pozostałe obrazy: ', len(data)) # Drukujemy ile zdjęć pozostało

    # Wizualizacja ostatecznych danych, z których będziemy korzystać
    if display:
        # Ten wykres jest podobny do poprzedniego
        histogr, _ = numpy.histogram(data['Uklad kierowniczy'], num_bins)
        pyplot.bar(centerGraph, histogr, width = 0.06)
        pyplot.plot((-1, 1),(sampBin, sampBin))
        pyplot.show()

    return data
```

loadMeasurements()

Po otrzymaniu zrównoważonego zbioru danych przygotujemy je do przetworzenia za pomocą funkcji *loadMeasurements()* i podzielimy je na dane treningu i testowania.

```
def loadMeasurements(dataPath, data):
    imgsPath = [] # Pusta lista ścieżek do obrazów
    steeringProcess = [] # Lista procesów sterowania :)

    for i in  range (len(data)): # Pętla długości danych
        indexedMeasurements = data.iloc[i] # Znajdziemy zindeksowane dane

       # Lista obrazów ze ścieżką
        imgsPath.append(os.path.join(dataPath,'IMG', indexedMeasurements[0]))

        # Lista procesów sterowania (kąt)
        steeringProcess.append(float(indexedMeasurements[3]))

    # Zmieniamy listy na tablice numpy
    imgsPath = numpy.asarray(imgsPath)
    steeringProcess = numpy.asarray(steeringProcess)


    return imgsPath, steeringProcess
```

Funkcja *loadMeasurements()* zwraca dane ścieżki pliku i kąt skrętu. Przekazujemy je do zmiennych *imgsPath* i *steeringsProcesses*.

```
imgsPath, steeringsProcesses = loadMeasurements(dataPath, data)
```

2.3. Przygotowanie danych do przetwarzania

Dane treningowe są w zasadzie tym, czego będziemy używać podczas procesu szkoleniowego. A dane walidacyjne to dane, których będziemy używać do testowania wydajności naszego stworzonego modelu po każdej z epok.

Podczas każdej iteracji kolejnej epoki, przypadki wchodzące w skład ciągu szkoleniowego są jeden po drugim dodawane do sieci neuronowej. W kolejnym kroku wartości wynikowe porównywane są z wartościami dodanymi. W taki sposób określona jest wartość błędu. Ten błąd i informacja o gradiencie funkcji błędu stanowią podstawę do częściowej zmiany wag. Następnie wszystkie te kroki są powtarzane. Po jednokrotnym szkoleniu program zweryfikuje dane walidacyjne, a następnie ponownie optymalizuje z tymi danymi walidacyjnymi.

Należy pamiętać, że dane walidacyjne nie wpływają na wagi ani parametry modeli. Są stosowane tylko do testów. Możemy więc być pewni, że dane walidacyjne są całkowicie niezależne od naszego modelu. Jest to zupełnie nowy zbiór danych. Dlatego potrzebujemy danych walidacyjnych, abyśmy mogli zobaczyć, jak dobrze nasz model poradzi sobie z danymi, których wcześniej nie widział. Aby podzielić zbiory danych, których będziemy tutaj używać, zaimportujemy moduł **train_test_split** [19] biblioteki Scikit-learn [20] Oczywiście możliwe jest wykonanie takich podziałów w inny sposób (np. przy użyciu tylko Numpy)**.** Biblioteka Scikit-learn zawiera przydatne funkcje, dzięki którym jest to trochę łatwiejsze.

W naszym przypadku przyda się parametr **test_size**, który określa rozmiar zestawu testowego. Reszta danych zostanie wykorzystana w zestawie treningowym. Jeśli oba parametry są jawnie ustawione (test_size i train_size), powinny się sumować do 1. Na przykład, w stosunku: 80% na trening, a 20% na walidację.

Teraz zdefiniujmy stan losowy **random_state**, aby uzyskać replikację, w celu ustawienia randomizacji, np. podać wartość 5, która jest wystarczająca:

```
xTraining, xTest, yTraining, yTest = train_test_split(imgsPath,
                    steeringsProcesses, test_size=0.2, random_state=5)
```

W kolejnym kroku wydrukujemy wartości **treningowe** i **walidacyjne**, dzięki czemu możemy zobaczyć, ile mamy obrazów dla każdego zbioru danych:

```
print('Całkowite obrazy treningowe: ', len(xTraining))
print('Całkowite obrazy walidacyjne: ', len(xTest))
```

```
Całkowite obrazy treningowe:    2629
Całkowite obrazy walidacyjne:   658
```

Dzięki funkcji train_test_split() udało nam się podzielić dane na 2 zestawy, obrazy treningowe i walidacyjne, w proporcji 80% i 20%.

2.4. Augmentacja danych

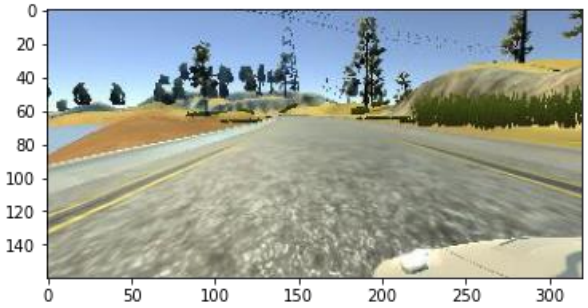
Koncepcja polega na tym, że bez względu na to, ile mamy danych, nigdy nie jest ich za dużo. Więc, co robimy? W danych, które posiadamy urozmaicamy oświetlenie, zmieniamy skalę, przesuwamy się w lewo lub w prawo - co nazywa się panoramowaniem. W ten sposób możemy zastosować różne metody dywersyfikacji naszych danych. Na przykład, gdybyśmy mieli 10 000 obrazów, moglibyśmy uzupełnić nasze dane i uzyskać już 40 000 obrazów. Za pomocą bardzo prostych technik możemy tworzyć nowe dane, zachowując nasze ograniczenia opisywane w poniższych funkcjach.

Rys. 8. Przykład oryginalnego obrazu

flip_function()

Użyjemy funkcji cv2, która się znajduje w bibliotece OpenCV [21]. Wykonuje ona losowe obracanie obrazu w lewo lub w prawo oraz dostosowanie kąta obrotu do zmiennej *steerAngle*. Powodem, dla którego potrzebujemy sterowania, jest to, że użyjemy również odwracania obrazu. Więc jeśli mamy obraz, który miał lewą krzywą i jeśli odwrócimy go w poziomie, krzywa stanie się prawą. Więc będziemy musieli zmienić znak. Realizuje to poniższy fragment kodu. Przykładowy wynik pokazany jest na Rys. 9.

```
def flip_function(picture, steerAngle):
    picture = cv2.flip(picture, 1)
    steerAngle = -steerAngle
    return picture, steerAngle
```

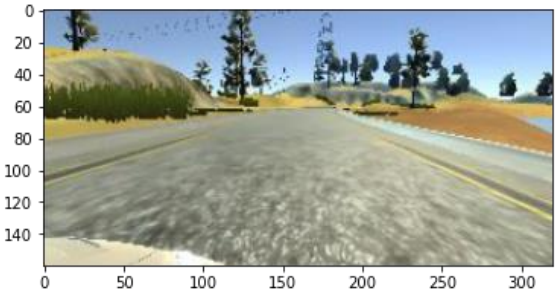


Rys. 9. Oryginalny obraz po zastosowaniu funkcji *flip_function()*

zoom_function()

Ta funkcja skaluje liniowy rozmiar obrazu. W pracy zastosowano zwiększenie obrazu o 20%. Przykład skalowania jest pokazany na Rys. 10.

```
def zoom_function(picture):
    zooming = augmenters.Affine(scale=(1, 1.2))
    picture = zooming.augment_image(picture)
    return picture
```

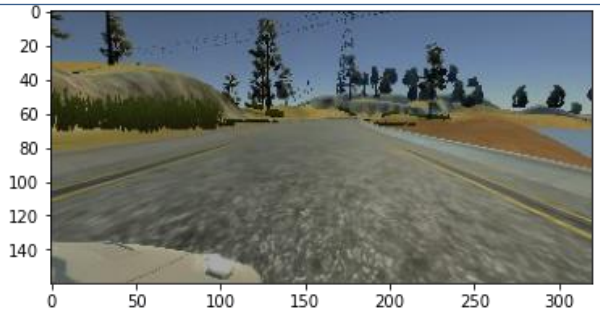


Rys. 10. Wynik działania funkcji *zoom_function()* dla obrazu z Rys. 8

brightness_function()

W procesie augmentacji danych zastosowano również funkcję *brightness_function()*. Zmienia ona losowo jasność zarejestrowanego obrazu w przedziale od 0.4 do 1.2, t.j. w zakresie od najciemniejszego do najjaśniejszego. Przykład działania tej funkcji jest przedstawiony na Rys. 11.

```
def brightness_function(picture):
    brightness1 = augmenters.Multiply((0.4, 1.2))
    picture = brightness1.augment_image(picture)
    return picture
```

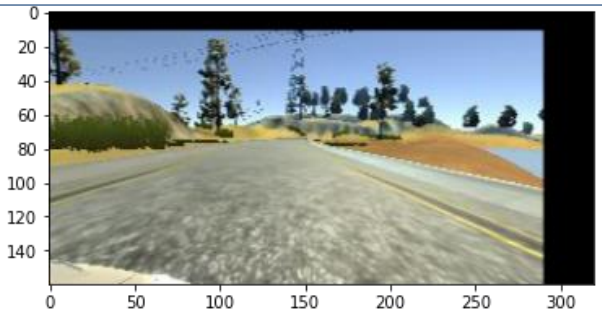


Rys. 11. Oryginalny obraz z Rys. 8 po zastosowaniu funkcji *brightness_function()*

panning_function()

Kolejną zastosowaną procedurę było panoramowanie. Wiadomo, że możemy przesunąć obraz w lewo lub prawo o określoną wartość w ramach zarejestrowanego kadru. W ten sposób możemy przesunąć obraz, na przykład, o plus lub minus 10%. Przykładowy wynik pokazany jest na Rys. 12. Po panoramowaniu obraz zostanie lekko przesunięty w prawo i jest również przesunięty w pionie, ale jak widać przesunięcie jest losowe w celu zwiększenia liczby zmian obrazów.

```
def panning_function(picture):
    panning = augmenters.Affine(
             translate_percent = {'x':(-0.1, 0.1), 'y':(-0.1, 0.1)}
             )
    picture = panning.augment_image(picture)
    return picture
```



Rys. 12. Wynik działania funkcji *panning_function()* dla obrazu z Rys. 8

imageEnlargement()

W procesie uczenia chodzi o to, że nie chcemy zaraz stosować tych wszystkich losowań do każdego obrazu. To nie ma sensu. Możemy więc wygenerować losową wartość, która będzie z zakresu od 0 do 1. Jeżeli liczba losowa jest mniejsza niż 0,5, funkcja zostanie uruchomiona. Oznacza to, że mamy 50% szans na wykonanie którejkolwiek z tych modyfikacji. W ten sposób zwiększamy ilość danych oraz ich różnorodność, co znacznie zwiększa efektywność uczenia maszynowego.

```
def imageEnlargement(imgPath, steeringProcess):
    pict = matplotlib.image.imread(imgPath)

    # ODWRACANIE
    if numpy.random.rand() < 0.5:
        pict, steeringProcess = flip_function(pict, steeringProcess)

    # SKALOWANIE
    if numpy.random.rand() < 0.5:
        pict = zoom_function(pict)

    # JASNOŚĆ
    if numpy.random.rand() < 0.5:
        pict = brightness_function(pict)

    # PRZESUWANIE
    if numpy.random.rand() < 0.5:
        pict = panning_function(pict)

    return pict, steeringProcess
```

Chcę podkreślić, że nie stosujemy tylko jednej transformacji lub jednej modyfikacji na raz, możemy zastosować wiele zmian. Na przykład, możliwe jest przesuwanie, zmiana jasności, odwracanie i także powiększenie obrazu. Teraz każda z tych akcji zostanie zastosowana w losowej kolejności i za każdym razem, gdy ją uruchomimy wynik będzie inny. Każdy nowy obraz jest wynikiem losowych zmian w określonym obrazie z oryginalnej listy.

2.5. Przetwarzanie wstępne

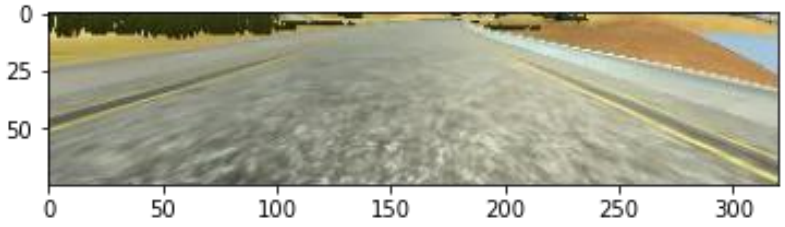
Dlaczego potrzebujemy wstępnego przetwarzania? Po pierwsze, jeśli spojrzymy na nasz obraz, to będziemy chcieli go wykadrować, ponieważ obraz będzie zawierał takie niepotrzebne detale jak np. maska samochodu. Nie chcemy też widzieć na naszym obrazie tła, którym są odległe góry lub drzewa. Możemy wyciąć tylko rejon drogi, dzięki czemu nie uzyskamy żadnych szczegółów, które nie są wymagane w naszym obrazie. W pracy wykorzystane są obrazy kolorowe RGB o szerokości 200 i wysokości 66 pikseli.

```
imgHeight, imgWidth, imgChannels = 66, 200, 3
```

cropping()

Ta funkcja pozwala na kadrowanie obrazu (usunięcie nieba na górze i samochodu z przodu na dole). Przykład działania tej funkcji pokazuje Rys. 13.

```
def cropping(picture):
    return picture[60:135, :, :]
```

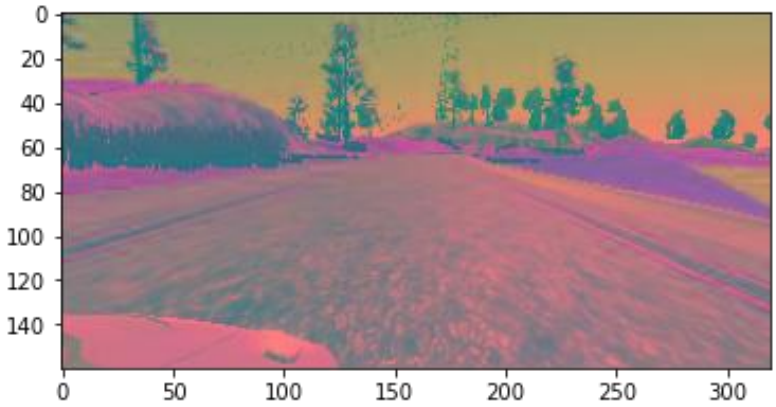


Rys. 13. Zastosowanie funkcji *cropping()* do obrazu z Rys. 8

RGB_to_YUV()

Funkcja ta konwertuje obraz z RGB na YUV. Takie podejście do przetwarzania obrazu jest stosowane podczas uruchamiania modelu NVIDIA. Dzięki tej zmianie obrazu linie staną się wyraźniejsze i bardziej zauważalne. Teraz znacznie łatwiej będzie określić położenie pasów na drodze i kierunek, w którym ona biegnie. Przykład konwersji jest pokazany na Rys. 14.

```
def RGB_to_YUV(picture):
    return cv2.cvtColor(picture, cv2.COLOR_RGB2YUV)
```

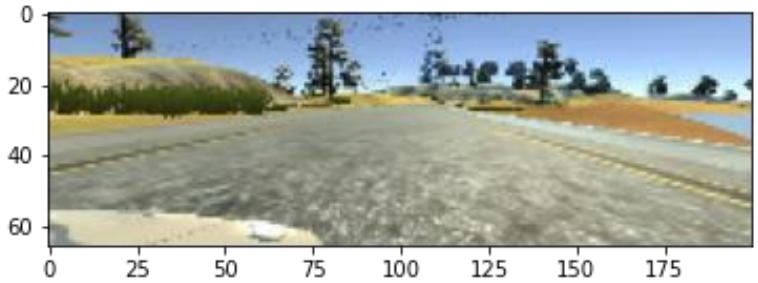


Rys. 14. Wynik działania funkcji *RGB_to_YUV()* dla obrazu z Rys. 8

changeTheSize()

Celem tej funkcji jest zmiana rozmiaru obrazu, który ma być używany w sieci neuronowej. Znormalizujemy rozmiar obrazu na 200x66 pikseli. Zabieg ten jest konieczny w celu zmniejszenia obciążenia procesora, jednak w zupełności wystarczające do uczenia sieci neuronowej. Przykładowy wynik normalizacji pokazany jest na Rys. 15.

```
def changeTheSize(picture):
    return cv2.resize(picture, (imgWidth, imgHeight))
```

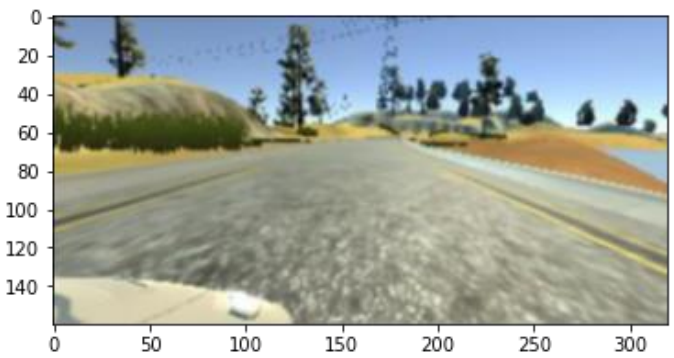


Rys. 15. Oryginalny obraz z Rys. 8 po zastosowaniu funkcji *changeTheSize()*

blurring()

W celu pogorszenia jakości zarejestrowanych obrazów, dodamy trochę Gaussowskiego rozmycia do naszego obrazu, aby model był w stanie rozpoznawać obiekty nawet na niewyraźnych obrazach (np. podczas mgły). W pracy zastosowano rozmiar jądra 3x3. Przykład działania tej funkcji pokazuje Rys. 16.

```
def blurring(picture):
    return cv2.GaussianBlur(picture, (imgChannels, imgChannels), 0)
                                # Rozmiar jądra = 3x3, Sigma X = 0
```



Rys. 16. Zastosowanie funkcji *blurring()* do obrazu z Rys. 8

preliminaryProcessing()
Wstępne przetwarzanie danych może odnosić się do manipulacji danymi przed ich użyciem w celu zwiększenia wydajności i jest ważnym krokiem w procesie eksploracji danych. Proces ten jest jednym z najważniejszych etapów realizacji projektu z wykorzystaniem uczenia maszynowego.

W celu jego realizacji łączymy powyższe funkcje wstępnego przetwarzania obrazów w jedną wspólną:

```
def preliminaryProcessing(picture):

    picture = cropping(picture)
    picture = RGB_to_YUV(picture)
    picture = blurring(picture)
    picture = changeTheSize(picture)
    picture = picture / 255
    return picture
```

Na zakończeniu wszystkich przetwarzań, wartości obrazu są dzielone przez 255, co konwertuje je do zakresu od 0 do 1 i zostanie wykorzystywane przy uczeniu maszynowym.

2.6. Generator wsadowy

Teraz, gdy zakończyliśmy wstępne przetwarzanie obrazów, przejdziemy do generowania wsadowego. Normalizacja wsadowa to technika wykorzystywana do przyspieszania i poprawiania stabilności sztucznych sieci neuronowych poprzez normalizowanie wejść warstw poprzez ponowne wyśrodkowanie i skalowanie. Chodzi o to, że nie wysyłamy wszystkich obrazów jednocześnie do naszego modelu szkoleniowego. To, co robimy, to wysyłanie obrazów w partiach, co również zmniejsza obciążenie procesora. Pomaga to więc w uogólnianiu i daje nam większą swobodę w tworzeniu naszych obrazów i wysyłaniu ich do naszego modelu. Więc to co możemy zrobić, to przed wysłaniem go do modelu, musimy zwiększyć liczbę obrazów, aby uzupełnić niezbędne informacje. Także musimy je wstępnie przetworzyć. W tym celu stworzymy funkcję, która przejmie ścieżkę do obrazów, które zdefiniowaliśmy wcześniej, oraz wartości kontrolowania. Możemy więc na przykład określić, że potrzebujemy partii setek obrazów. Następnie z tej ścieżki dostępu i listy kontrolnej wyodrębnić setki losowych obrazów. Zwiększyć je, wstępnie przetworzyć i wysłać do naszego modelu.

W ostatnim kroku przekonwertujemy te dwie listy do formatu Numpy dla ułatwienia użytkowania w dalszej części projektu. Na przykład, jeśli poprosimy o sto obrazów, konwersja utworzy tę listę stu obrazów wraz z wartościami sterowania, a następnie wyśle je do treningu modelu. Ale wcześniej zastosuje augmentację i wstępne przetwarzanie.

Nie ma potrzeby uzupełniania naszych obrazów testowych, ponieważ zamierzamy użyć tego samego generatora wsadowego do uczenia naszego modelu, a następnie jego testowania. Więc podczas procesu walidacji nadal będziemy prosić o wygenerowanie partii, ale nie chcemy, aby ta partia walidacji była tym, co nazywamy rozszerzoną. Potrzebujemy rozszerzonej partii tylko do trenowania modelu.

Możemy więc stworzyć tutaj flagę i nazwać ją flagą treningową. Jeśli *flagTraining* jest *True*, to nie importujemy obrazu zaraz do wstępnego przetwarzania, ale importujemy go do rozszerzania obrazów. Jednak, jeśli jest to *False*, to nie zwiększy liczby obrazów, po prostu je załaduje i wyśle do wstępnego przetwarzania.

Yield [22] to słowo kluczowe w Pythonie, które służy do powrotu z funkcji bez niszczenia stanów jej zmiennej lokalnej. Instrukcja *yield* działa inaczej niż *return*. Ona zwraca wartość i wstrzymuje działanie funkcji, ale jej nie opuszcza. Każda funkcja, która zawiera słowo kluczowe *yield*, nazywana jest generatorem. Wartości generatora są obliczane przez funkcję generatora dopiero na żądanie [23]**.** Zaletą tej funkcji jest to, że projekt wykorzystujący *yield* umożliwia „zagnieżdżanie" jednego generatora w drugim, czyli tworzenie podgeneratorów.

```
def batchGenerator(imgsPath, steerProcessList, sizeBatch, flagTraining):
    while 1:
        pictBatch = []
        steerProcessBatch = []

        for i in range(sizeBatch):
            indx = random.randint(0, len(imgsPath)-1)
            if flagTraining:
                pict, steeringProcess = imageEnlargement(imgsPath[indx],
                                                         steerProcessList[indx])

      # Spowoduje to rozszerzenie obrazu o więcej opcji,
        a gdy to zrobimy, zastosujemy wstępne przetwarzanie.

            else:
                pict = matplotlib.image.imread(imgsPath[indx])
                steeringProcess = steerProcessList[indx]

            pict = preliminaryProcessing(pict)
            pictBatch.append(pict)
            steerProcessBatch.append(steeringProcess)

        yield (numpy.asarray(pictBatch),numpy.asarray(steerProcessBatch))
```

3. Konwolucyjne sieci neuronowe

Głębokie uczenie staje się bardzo popularnym podzbiorem uczenia maszynowego ze względu na wysoki poziom wydajności na wielu typach danych. Świetnym sposobem na wykorzystanie głębokiego uczenia do klasyfikowania obrazów jest zbudowanie konwolucyjnej sieci neuronowej.

Konwolucyjne sieci neuronowe (CNN) [5] to algorytm głębokiego uczenia, który może przyjąć obraz wejściowy, przypisać znaczenie (nauczane wagi i błędy systematyczne) różnym aspektom/obiektom na obrazie i być w stanie odróżnić jeden od drugiego. Podczas gdy w metodach prymitywnych filtry są tworzone ręcznie, po odpowiednim przeszkoleniu takie sieci są w stanie nauczyć się tych filtrów/charakterystyk samodzielnie.

3.1. Konstruowanie modeli

Architektura CNN (improwizowana wersja wielowarstwowego perceptronu) [5] jest analogiczna do wzorca łączności neuronów w ludzkim mózgu i została zainspirowana organizacją kory wzrokowej. Poszczególne neurony reagują na bodźce tylko w ograniczonym obszarze pola widzenia, znanym jako pole odbiorcze. Zbiór takich pól nakłada się na cały obserwowany obszar. CNN, przechodząc przez zbiór pikseli jeden po drugim i wprowadzając je do sieci neuronowej w celu klasyfikacji obrazu, jest w stanie zidentyfikować krzywe, krawędzie, kształty obiektu na obrazie. Splotowe sieci neuronowe odróżniają się od innych sieci neuronowych dzięki ich doskonałej wydajności w zakresie sygnałów wejściowych. Mają 3 główne typy warstw, którymi są:

Warstwa splotowa jest pierwszą warstwą sieci konwolucyjnej, która służy do przetwarzania danych. Dane wejściowe mają 3 wymiary — wysokość, szerokość i głębokość — która odpowiada kolorom RGB na obrazie. Mamy też detektor cech, zwany także jądrem lub filtrem, który będzie poruszał się po podatnych polach obrazu, sprawdzając, czy dana cecha jest obecna. Pierwszy model będzie wykorzystywał funkcję aktywacji *elu*, a w drugim będzie *selu*. Główną zaletą tych funkcji aktywacji jest to, że mogą przyjmować wartości ujemne [24]. Pracujemy z danymi kąta skrętu, gdzie jazda na wprost wynosi zero, a każdy zakręt dodaje lub odejmuje stosowną wartość.

Warstwa łączenia, znana jest również jako próbkowanie w dół, prowadzi do redukcji wymiarowości, zmniejszając liczbę parametrów na wejściu. Podobnie jak w przypadku warstwy splotowej, operacja łączenia omiata filtr przez całe dane wejściowe, ale różnica polega na tym, że ten filtr nie ma żadnych wag. Zamiast tego jądro stosuje funkcję agregującą do wartości w polu receptywnym, zapełniając tablicę wyjściową. Istnieją dwa główne typy łączenia: *MaxPooling* i *AveragePooling*.

We własnym drugim modelu będziemy używać warstwy *MaxPooling*. Warstwy pooling'u służą do zmniejszania wymiarów map obiektów. A ponieważ w tym modelu mamy ich dużo, zmniejsza to równocześnie liczbę parametrów do badania i liczbę obliczeń wykonywanych w sieci. Ta warstwa wykorzystuje następujące argumenty:

- **Pool size** - jest to liczba całkowita używana do zmniejszania skali.
  Użyjemy (2, 2) co zmniejszy o połowę dane wejściowe w obu wymiarach przestrzennych (wysokość, szerokość).
- **Dropout Layer** - warstwa dropout służy do minimalizacji overfittingu.
- **Flatten Layer** - spłaszcza dane wejściowe do wektora z pojedynczą kolumną, aby przekazać je do sieci neuronowej, która jest nazwana Dense Layer.

Pełna połączona warstwa trafnie opisuje samą siebie. Jak wspomniano wcześniej, wartości pikseli obrazu wejściowego nie są bezpośrednio połączone z warstwą wyjściową w warstwach częściowo połączonych. Jednak w warstwie w pełni połączonej każdy węzeł w warstwie wyjściowej łączy się bezpośrednio z węzłem w poprzedniej warstwie. Ta warstwa wykonuje zadanie klasyfikacji na podstawie cech wyodrębnionych przez poprzednie warstwy i ich różnych filtrów. Podczas gdy warstwy splotowe i zbiorcze zwykle używają funkcji typu *ReLu*, warstwy w pełni połączone zwykle wykorzystują funkcję aktywacji *softmax*, aby odpowiednio klasyfikować dane wejściowe, dając prawdopodobieństwo od 0 do 1. Więc warstwa w pełni połączona służy do przewidywania kąta skrętu.

Biorąc pod uwagę fakt, że pracujemy z obrazami dwuwymiarowymi, główną warstwą w modelach wykorzystywana będzie *Conv2D*. Argumenty używane w warstwie Conv2D są następujące:

- **Filters** - jest to liczba całkowita oznaczająca wymiarowość przestrzeni wyjściowej w każdym splocie.
- **Kernel size** - reprezentuje wysokość i szerokość okna splotu do wykonania operacji konwolucyjnej.
- **Padding** - proces dodawania warstw zer do naszych obrazów wejściowych.
- **Input shape** - podawanie wymiarów wejścia, np. 64x64x3.

3.1.1. Model NVIDIA

Idea modelu została zapożyczona z artykułu firmy NVIDIA [7], w którym znajduje się opis architektury tej sieci. Jej głównym zadaniem jest wytrenowanie wag sieci, aby zminimalizować błąd średniokwadratowy między poleceniem sterowania wyprowadzanym przez sieć a poleceniem ludzkiego kierowcy. Sieć składa się z 9 warstw; w tym 5 warstw splotowych, 3 w pełni połączonych warstw i warstwy normalizacyjnej. Obraz wejściowy jest dzielony na komponenty YUV i przekazywany do sieci.

Pierwsza warstwa sieci wykonuje normalizację obrazu. Następnie wykorzystywane są warstwy splotowe, które zostały zaprojektowane do ekstrakcji cech i wybrane empirycznie w serii eksperymentów z różnymi konfiguracjami warstw. W pierwszych trzech warstwach konwolucyjnych używamy splotów z krokiem 2×2 i jądrem 5×5 oraz w dwóch ostatnich warstwach splotów z wielkością jądra 3×3. Używamy 5 warstw splotowych z 3 w pełni połączonymi warstwami, co prowadzi do wyjściowej wartości kontrolnej, którą jest odwrotny promień skrętu. W pełni połączone warstwy zaprojektowano tak, aby działały jako kontrolery sterowania, ale zauważamy, że trenując system od końca do końca, nie można zrobić wyraźnego podziału między tym, które części sieci działają głównie jako ekstraktor funkcji, a które służą jako kontroler.

W większości przypadków jako funkcję aktywacji używamy *relu*, gdzie wartości ujemne stają się zerowe, jednak w naszym przypadku należy wziąć pod uwagę liczby ujemne. Pracujemy z danymi kąta skrętu, gdzie jazda na wprost wynosi zero, a każdy zakręt inkrementuje lub dekrementuje aktualną wartość. W tym celu użyjemy podobnej funkcji aktywacji *elu*, która również przyjmuje wartości ujemne [24].

```
def NvidiaModel():

    model = Sequential() # Ten model jest w zasadzie sekwencyjny
    model.add(Conv2D(24, (5, 5), (2, 2),
                     input_shape=(66, 200, 3), activation='elu'))

    model.add(Conv2D (36, (5, 5), (2, 2), activation='elu'))
    model.add(Conv2D (48, (5, 5), (2, 2), activation='elu'))
    # Tym razem mamy rozmiar 64, rozmiar jądra 3 na 3, a skok 1 na 1
    model.add(Conv2D (64, (3, 3), activation='elu'))
    model.add(Conv2D (64, (3, 3), activation='elu'))
    # Następnie mamy spłaszczoną warstwę, więc wszystko spłaszczymy
    model.add(Flatten())
    # Pierwsza z nich ma 100 neuronów, a funkcja aktywacji ponownie jest elu
    # Więc mamy jeszcze 3, właściwie 4 gęstsze warstwy
    model.add(Dense(100, activation='elu'))
    model.add(Dense(50, activation='elu'))
    model.add(Dense(10, activation='elu'))
    # Na końcu powinniśmy mieć tylko jedną wartość
    #  i to będzie nasz ostateczny wynik.
    model.add(Dense(1))

    # A ostatnią rzeczą, którą musimy zrobić, to skompilować nasz model
    # Adam - optymalizator z szybkością uczenia 0,0001
    #Użycie najczęstszej funkcji strat, czyli błędu średniokwadratowego (MSE)
    # Jest to problem regresji, mamy ciągłe wyjście, więc używamy MSE
    model.compile(Adam(lr=0.0001), loss='mse')

    return model
```

Tworzenie modelu NVIDIA

Tworzenie tego modelu wiąże się z zapisaniem jego architektury jako zmiennej do późniejszego wykorzystania. Struktura wszystkich warstw modelu NVIDIA wygląda następująco:

- Warstwa splotowa: 5x5, filtr: 24, kroki: 2x2, funkcja aktywacji: ELU
- Warstwa splotowa: 5x5, filtr: 36, kroki: 2x2, funkcja aktywacji: ELU
- Warstwa splotowa: 5x5, filtr: 48, kroki: 2x2, funkcja aktywacji: ELU
- Warstwa splotowa: 3x3, filtr: 64, kroki: 1x1, funkcja aktywacji: ELU
- Warstwa splotowa: 3x3, filtr: 64, kroki: 1x1, funkcja aktywacji: ELU
- W pełni połączona warstwa: liczba neuronów: 100, funkcja aktywacji: ELU
- W pełni połączona warstwa: liczba neuronów: 50, funkcja aktywacji: ELU
- W pełni połączona warstwa: liczba neuronów: 10, funkcja aktywacji: ELU
- Warstwa normalizacyjna: liczba neuronów: 1 (wyjście)

Ogólna reprezentacja struktury wszystkich warstw sieci neuronowej modelu NVIDIA przedstawiona jest w Tabeli 3, którą uruchomiliśmy podczas korzystania z funkcji *summary()* [25].

```
NvidiaModel = NvidiaModel()
NvidiaModel.summary()
```

Tabela 3. Ogólna reprezentacja struktury wszystkich warstw modelu NVIDIA

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 31, 98, 24)        1824

 conv2d_1 (Conv2D)           (None, 14, 47, 36)        21636

 conv2d_2 (Conv2D)           (None, 5, 22, 48)         43248

 conv2d_3 (Conv2D)           (None, 3, 20, 64)         27712

 conv2d_4 (Conv2D)           (None, 1, 18, 64)         36928

 flatten (Flatten)           (None, 1152)              0

 dense (Dense)               (None, 100)               115300

 dense_1 (Dense)             (None, 50)                5050

 dense_2 (Dense)             (None, 10)                510

 dense_3 (Dense)             (None, 1)                 11

=================================================================
Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
_________________________________________________________________
```

3.1.2. Własny model

Budowanie modelu to przede wszystkim unikalna kombinacja różnych konfiguracji warstw. Model Nvidii nie jest jedyną możliwą konfiguracją rozpoznawania obiektów na obrazie. Postanowiłem ulepszyć opisaną wcześniej sieć neuronową, korzystając z modelu mojego poprzedniego projektu klasyfikacji obrazów [8]. Model ten zbudowany jest przy użyciu biblioteki Keras [26], typu open source napisanej w Pythonie, która zapewnia interakcję ze sztucznymi sieciami neuronowymi. Ta biblioteka zawiera liczne implementacje powszechnie używanych elementów składowych sieci neuronowych, takich jak warstwy, funkcje celu i transferu, optymalizatory oraz wiele narzędzi upraszczających pracę z obrazami i tekstem. Poniżej znajduje się funkcja tworzenia architektury własnego modelu:

```
def MyModel():

    # Tworzymy model
    model = keras.models.Sequential()
    model.add(keras.layers.BatchNormalization(
              input_shape=(imgHeight, imgWidth, imgChannels)))

    model.add(keras.layers.Conv2D(24, (5, 5), (2, 2), padding='same',
         input_shape=(imgHeight, imgWidth, imgChannels), activation ='selu'))

    model.add(keras.layers.MaxPooling2D(pool_size=(5, 5), strides=(1,1)))
    model.add(keras.layers.Dropout(0.25))

    model.add(keras.layers.BatchNormalization(
              input_shape = (imgHeight, imgWidth, imgChannels)))

    model.add(keras.layers.Conv2D(36, (5, 5), (2, 2), padding='same',
        input_shape=(imgHeight, imgWidth, imgChannels), activation = 'selu'))

    model.add(keras.layers.MaxPooling2D(pool_size=(5, 5), strides=(1,1)))
    model.add(keras.layers.Dropout(0.25))

    model.add(keras.layers.BatchNormalization(
              input_shape=(imgHeight, imgWidth, imgChannels)))

    model.add(keras.layers.Conv2D(48, (5, 5), (2, 2), padding='same',
          input_shape=(imgHeight, imgWidth, imgChannels), activation='selu'))

    model.add(keras.layers.MaxPooling2D(pool_size=(5, 5), strides=(1,1)))
    model.add(keras.layers.Dropout(0.25))


    model.add(Flatten())
    model.add(Dense(100, activation='selu'))
    model.add(Dense(50, activation='selu'))
    model.add(Dense(10, activation='selu'))
    model.add(Dense(1))

    # Kompilujemy model
    model.compile(loss='mse', optimizer=Adam(learning_rate=.0001))

    return model
```

Tworzenie własnego modelu

Tworzenie modelu wiąże się z zapisaniem jego architektury jako zmiennej w celu późniejszego wykorzystania. Struktura własnego modelu składa się z 17 warstw:

- 3 warstwy normalizacji BatchNormalization
- 3 w pełni połączone warstwy Dense
- 3 warstwy splotowe Conv2D
- 3 warstwy MaxPooling
- 3 warstwy Dropout
- 1 warstwa spłaszczona Flatten
- 1 warstwa wyjściowa

Tabela 4. Struktura wszystkich warstw własnego modelu

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 batch_normalization (Batch   (None, 66, 200, 3)         12
 Normalization)

 conv2d_5 (Conv2D)            (None, 33, 100, 24)       1824

 max_pooling2d (MaxPooling2D) (None, 29, 96, 24)         0

 dropout (Dropout)            (None, 29, 96, 24)         0

 batch_normalization_1 (Batch (None, 29, 96, 24)         96
 Normalization)

 conv2d_6 (Conv2D)            (None, 15, 48, 36)        21636

 max_pooling2d_1 (MaxPooling2D) (None, 11, 44, 36)         0

 dropout_1 (Dropout)          (None, 11, 44, 36)         0

 batch_normalization_2 (Batch (None, 11, 44, 36)        144
 Normalization)

 conv2d_7 (Conv2D)            (None, 6, 22, 48)         43248

 max_pooling2d_2 (MaxPooling2D) (None, 2, 18, 48)          0

 dropout_2 (Dropout)          (None, 2, 18, 48)          0

 flatten_1 (Flatten)          (None, 1728)               0

 dense_4 (Dense)              (None, 100)               172900

 dense_5 (Dense)              (None, 50)                5050

 dense_6 (Dense)              (None, 10)                510

 dense_7 (Dense)              (None, 1)                 11

=================================================================
Total params: 245,431
Trainable params: 245,305
Non-trainable params: 126
```

Na końcu mamy spłaszczoną warstwę, której celem jest po prostu „spłaszczenie" wyjścia, czyli przekształcenie wyjścia 2D na wyjście 1D. Zanim przejdziemy do treningu, wymagana jest kompilacja do sfinalizowania modelu i uczynienia go całkowicie gotowym do użycia. Do kompilacji musimy określić optymalizator i funkcję straty. Możemy skompilować model za pomocą metody *compile*, która posiada następujące atrybuty:

- ***optimizer*** - algorytm mierzący, jak bardzo pożądana wartość różni się od przewidywanej. Istnieją różne optymalizatory, np. SGD, Adam, Adadelta itp.
- ***loss*** - algorytm dopasowywania parametrów wewnętrznych (wagi i odchylenia) modelu w celu zminimalizowania funkcji straty.

Ogólna reprezentacja struktury wszystkich warstw sieci neuronowej własnego modelu przedstawiona jest w Tabeli 4, którą uruchomiliśmy podczas korzystania z funkcji summary() [25].

```
MyModel = MyModel()
MyModel.summary()
```

3.2. Trening

Uczenie modelu to faza cyklu rozwoju nauki o danych, w której programiści starają się dopasować najlepszą kombinację wag i odchyleń do algorytmu uczenia maszynowego, aby zminimalizować funkcję straty w zakresie przewidywania. Ten iteracyjny proces nazywa się *dopasowywaniem modelu*. W naszym przypadku dopasowujemy model, wysyłając partiami zdjęcia i układ kierowniczy.

W celu trenowania modelu dla określonej liczby epok (iteracji w zestawie danych) używana jest funkcja *fit()* [27], która posiada następujące argumenty:

- **batchGenerator** - używamy do produkcji losowych partii danych, na podstawie podanych wielkości partii.
- **steps_per_epoch** - całkowita liczba kroków (partii próbek) do uzyskania z generatora przed zadeklarowaniem zakończenia jednej epoki i rozpoczęciem następnej.
- **epochs** - liczba epok do trenowania modelu.
- **validation_data** - krotka (xTest, yTest), według której oceniane są straty i wszelkie metryki modelu na końcu każdej epoki.
- **validation_steps** - ma znaczenie tylko wtedy, gdy określono *steps_per_epoch*. Jest on całkowitą liczbą kroków (partii próbek) do walidacji przed zatrzymaniem.

Trening modelu NVIDIA

Czas zacząć trening modelu NVIDIA. Proces ten zwykle zajmuje pewną ilość czasu, która różni się w zależności od złożoności sieci neuronowej, ilości wykorzystywanych danych i wydajności procesora. Aby określić, ile czasu zajął modelowi proces uczenia, ustalamy czas rozpoczęcia treningu modelu NVIDIA za pomocą funkcji *time()* [28]:

```
timeBefore1 = time()
```

Zmienna *NvidiaHistory* przyjmuje wynik treningu modelu NVIDIA, którego wydajność została przedstawiona w Tabeli 5 dla 10 epok:

```
NvidiaHistory = NvidiaModel.fit(
                        batchGenerator(xTraining, yTraining, 100, 1),
                        steps_per_epoch=300, epochs=10,
                validation_data=batchGenerator(xTest, yTest, 100, 0),
                validation_steps=200)
```

Tabela 5. Testowanie modelu NVIDIA

```
Epoch 1/10
300/300 [==============================] - 332s 1s/step - loss: 0.0478 -
                                                        val_loss: 0.0279
Epoch 2/10
300/300 [==============================] - 207s 691ms/step - loss: 0.0397 -
                                                        val_loss: 0.0262
Epoch 3/10
300/300 [==============================] - 214s 714ms/step - loss: 0.0375 -
                                                        val_loss: 0.0253
Epoch 4/10
300/300 [==============================] - 197s 659ms/step - loss: 0.0342 -
                                                        val_loss: 0.0228
Epoch 5/10
300/300 [==============================] - 189s 631ms/step - loss: 0.0330 -
                                                        val_loss: 0.0238
Epoch 6/10
300/300 [==============================] - 183s 610ms/step - loss: 0.0306 -
                                                        val_loss: 0.0237
Epoch 7/10
300/300 [==============================] - 186s 622ms/step - loss: 0.0304 -
                                                        val_loss: 0.0235
Epoch 8/10
300/300 [==============================] - 194s 649ms/step - loss: 0.0296 -
                                                        val_loss: 0.0237
Epoch 9/10
300/300 [==============================] - 200s 668ms/step - loss: 0.0284 -
                                                        val_loss: 0.0247
Epoch 10/10
300/300 [==============================] - 199s 665ms/step - loss: 0.0288 -
                                                        val_loss: 0.0235
```

Tabela 5 pokazuje, że po 10 epokach udało się zredukować stratę z 0.0478 do 0,0288 oraz stratę walidacyjną z 0.0279 do 0.0235. Wskaźniki te dowodzą, że model NVIDIA jest skuteczny w rozwiązywaniu naszego problemu. Teraz ustalamy czas zakończenia treningu za pomocą funkcji *time()*:

```
timeAfter1 = time()
```

Kolejnym krokiem jest wyświetlenie otrzymanej informacji o czasie, poświęconym na wszystkie obliczenia podczas treningu.

```
print(
    "Czas obliczeń modelu NVIDIA: ", round(timeAfter1-timeBefore1, 3),
    "sekund = ", round((timeAfter1 - timeBefore1)/60, 3), "minuty")
```

Funkcja *print()* zwraca informację, że:

```
Czas obliczeń modelu NVIDIA: 1909.412 sekund = 31.824 minuty
```

Trening własnego modelu

Przejdźmy teraz do treningu własnego modelu. Ze względu na jego większą złożoność spodziewam się, że ten proces potrwa trochę dłużej, chociaż moc procesora i ilość wykorzystywanych danych są takie same. Wynika to z faktu, że w pierwszym modelu liczba warstw jest o połowę mniejsza niż w moim. Aby określić, ile czasu zajął modelowi proces uczenia, ustalamy czas rozpoczęcia treningu własnego modelu za pomocą funkcji *time()* [28]:

```
timeBefore2 = time()
```

Zmienna *MyHistory* przyjmuje wynik treningu własnego modelu, którego wydajność jest przedstawiona w Tabeli 6 w ciągu 10 epok:

```
MyHistory = MyModel.fit(
                        batchGenerator(xTraining, yTraining, 100, 1),
                        steps_per_epoch=300, epochs=10,
                validation_data=batchGenerator(xTest, yTest, 100, 0),
                validation_steps=200)
```

Tabela 6 pokazuje, że po 10 epokach udało się zredukować stratę z 0.3217 do 0.0365 oraz stratę walidacyjną z 0.0763 do 0.0291. Wskaźniki te dowodzą, że własny model też jest skuteczny w rozwiązywaniu naszego problemu. Teraz ustalamy czas zakończenia treningu za pomocą funkcji *time()*:

```
timeAfter2 = time()
```

Tabela 6. Testowanie własnego modelu

```
Epoch 1/10
300/300 [==============================] - 338s 1s/step - loss: 0.3217
- val_loss: 0.0763
Epoch 2/10
300/300 [==============================] - 328s 1s/step - loss: 0.0869
- val_loss: 0.0553
Epoch 3/10
300/300 [==============================] - 326s 1s/step - loss: 0.0637
- val_loss: 0.0430
Epoch 4/10
300/300 [==============================] - 329s 1s/step - loss: 0.0553
- val_loss: 0.0422
Epoch 5/10
300/300 [==============================] - 335s 1s/step - loss: 0.0506
- val_loss: 0.0383
Epoch 6/10
300/300 [==============================] - 326s 1s/step - loss: 0.0453
- val_loss: 0.0380
Epoch 7/10
300/300 [==============================] - 334s 1s/step - loss: 0.0427
- val_loss: 0.0350
Epoch 8/10
300/300 [==============================] - 340s 1s/step - loss: 0.0402
- val_loss: 0.0323
Epoch 9/10
300/300 [==============================] - 335s 1s/step - loss: 0.0382
- val_loss: 0.0310
Epoch 10/10
300/300 [==============================] - 333s 1s/step - loss: 0.0365
- val_loss: 0.0291
```

Kolejnym krokiem jest wyświetlenie otrzymanej informacji o czasie poświęconym na wszystkie obliczenia podczas treningu mojego modelu.

```
print(
  "Czas obliczeń modelu własnego: ", round(timeAfter2-timeBefore2, 3),
  "sekund = ", round((timeAfter2-timeBefore2)/60, 3), "minuty")
```

Funkcja *print()* zwraca informację, że:

```
Czas obliczeń modelu własnego: 3187.757 sekund = 53.129 minuty
```

Jak więc widzimy, trenowanie własnego modelu trwa o 22 minuty dłużej niż modelu NVIDIA. Wynika to z faktu, że ma on znacznie więcej ukrytych warstw, co zwiększa ilość obliczeń i zajmuje więcej czasu.

3.3. Zapisywanie architektury modeli

W plikach z rozszerzeniem **.h5** [29] zapisuje się wagę i architekturę badanego modelu. Zapisanie każdego z modeli pozwala na uruchomienie wytrenowanej sieci neuronowej w dowolnym momencie. Oszczędza to czas podczas ponownego uruchamiania symulacji bez konieczności uczenia modelu za każdym razem.

Zapisywanie modelu NVIDIA

Zmienna *NvidiaModel*, która otrzymała wynik uczenia modelu NVIDIA tymczasowo przechowuje informacje, które należy zapisać w dedykowanym pliku *NvidiaModel.h5*:

```
NvidiaModel.save('NvidiaModel.h5')
print('NvidiaModel został zapisany')
```

```
NvidiaModel został zapisany
```

Zapisywanie własnego modelu

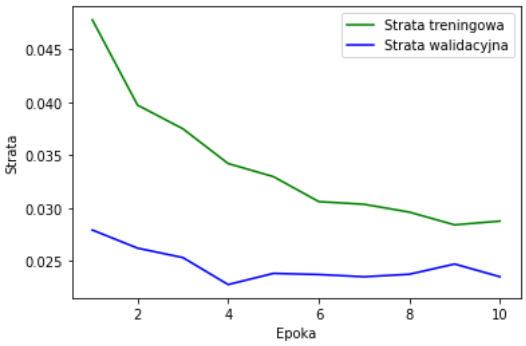
Zmienna *MyModel*, która otrzymała wynik uczenia własnego modelu, tymczasowo przechowuje informacje, które też należy zapisać w osobnym pliku *MyModel.h5*:

```
MyModel.save('MyModel.h5')
print('MyModel został zapisany')
```

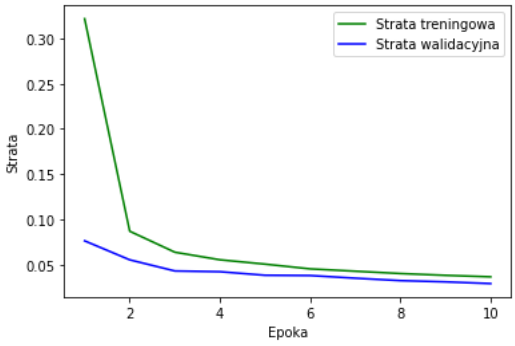
```
MyModel został zapisany
```

3.4. Analiza wydajności modeli

Wizualizacja danych jest ważna, aby zrozumieć, w jaki sposób dane są wykorzystywane w konkretnym modelu uczenia głębokiego, co pomaga w analizie wydajności tych modeli. W naszym przypadku dla obu modeli tworzone są wykresy strat treningu i walidacji. Rysunki 17 i 18 dają dużo informacji zarówno do oddzielnej analizy wydajności każdej sieci neuronowej, jak i do porównania ich między sobą.



Rys. 17. Wykres strat modelu NVIDIA



Rys. 18. Wykres strat własnego modelu

Oba wykresy (Rys. 17 i Rys. 18) przedstawiają straty treningowe i walidacyjne. Pierwszy wykres (Rys. 17) pokazuje, że model NVIDIA zaczyna już z niewielką stratą i stopniowo ją zmniejsza. Na drugim wykresie (Rys. 18), prezentującym mój model, początkowo straty są znacznie wyższe, ale ostatecznie po 10 epokach straty są w przybliżeniu takie same jak w modelu poprzednim. Głównym czynnikiem, odpowiedzialnym za takie zachowanie, jest fakt, że w drugim modelu ukrytych warstw jest znacznie więcej, co komplikuje uczenie na samym początku.

Z porównania obu modeli wynika, że model Nvidii i mój własny model sieci neuronowej są w stanie osiągnąć w przybliżeniu ten sam wynik i być skuteczne w tworzeniu pojazdów autonomicznych. Bazując na tym wniosku mogę stwierdzić, że w symulacji jazdy oba te modele będą miały taki sam wpływ na kierowanie samochodu testowego. W kolejnym rozdziale przetestujemy to, tworząc symulacje testowe autonomicznej jazdy z wykorzystaniem architektury obu modeli.

4. Wirtualna symulacja testowa

Symulacja wirtualna może przetestować kompletny prototyp systemu przy użyciu sztucznego środowiska technologii komputerowej. Wirtualne modelowanie pozwala budować różne tematyczne przestrzenie cech i odtwarzać ich zmiany w czasie, a także umożliwia przesyłanie i przemieszczanie obserwatora w tych wirtualnych przestrzeniach, ukazując zmiany właściwości rzeczywistych obiektów czasoprzestrzennych. Naszym zadaniem jest uruchomienie wirtualnego środowiska do wizualnego testowania efektywności modeli sieci neuronowych w celu obserwacji autonomicznej jazdy pojazdów.

4.1. Inicjalizacja portu i komunikacja

W przypadku, gdy uruchamiamy klienta i serwer Pythona, najlepszą opcją jest użycie protokołu transportowego Socket.IO [30]. Umożliwia to dwukierunkową komunikację opartą na zdarzeniach w czasie rzeczywistym między klientami (zwykle, choć nie zawsze, przeglądarkami internetowymi) a serwerem.

Ten serwer zostanie uruchomiony w aplikacji napisanej przy użyciu frameworka Flask. Flask to lekki framework aplikacji internetowych typu WSGI [31]. Został zaprojektowany tak, aby rozpoczęcie pracy było szybkie i łatwe, z możliwością skalowania do złożonych aplikacji.

Trzecią zmienną jest ograniczenie prędkości z jaką pojazd będzie się poruszał podczas symulacji. Musimy to zrobić, aby wyszkolona sieć neuronowa prawidłowo zarządzała samochodem. Również jest to niezbędne, aby wytrenowana sieć neuronowa miała czas na akceptację przychodzących parametrów i podejmowanie na tej podstawie właściwych decyzji.

```
socket_io = socketio.Server()
simulationApp = Flask(__name__)
speedLimit = 10 # Ograniczenie prędkości
```

Zmienna datasetSimulation() to lista z trzema kolumnami: układ kierowniczy, przepustnica i prędkość. Następnie zapiszemy te dane w osobnych plikach i wyświetlimy je na wykresie w celu porównania wyników działania obu modeli.

```
datasetSimulation = [[], [], []]
```

Telemetria jest niezbędna do odbierania, przetwarzania i rejestrowania informacji pomiarowych, a **@** - handler zwykle oznacza program obsługi czegoś (jakiegoś zdarzenia, połączenia przychodzące, wiadomości itp.)

Przede wszystkim w funkcji *telemetry()* importujemy nasz nowy obraz, który pochodzi bezpośrednio z pojazdu. Jest on następnie konwertowany do formatu Numpy. Musimy wstępnie przetworzyć ten obraz, co znacznie zwiększa wydajność sieci neuronowej. Z tego powodu, że wykonaliśmy wstępne przetwarzanie podczas szkolenia i walidacji, musimy również wykonać wstępne przetwarzanie w testowaniu.

Do wstępnego przetworzenia zaimportowanego obrazu trasy z pojazdu używamy opisanej wcześniej funkcji *preliminaryProcessing()*. Pozwala ona na manipulowanie danymi przed ich użyciem w celu zwiększenia wydajności i jest ważnym krokiem w procesie eksploracji danych. Proces ten jest jednym z najważniejszych etapów realizacji projektu z wykorzystaniem uczenia maszynowego.

Załadowana sieć neuronowa, wraz ze wszystkimi wstępnymi manipulacjami danymi, zapewnia nam niezbędne informacje o układzie sterowania, którą możemy wysłać do naszego symulatora. Zasadniczo ograniczamy prędkość, aby nie przekroczyć maksymalnego progu, który został zdefiniowany powyżej.

```
@socket_io.on('telemetry')
def telemetry(sid, data):
    fixedSpeed = float(data['speed'])
    img = Image.open(BytesIO(base64.b64decode(data['image'])))
    img = numpy.asarray(img)
    img = preliminaryProcessing(img)
    img = numpy.array([img])

    steeringProcess = float(simulationModel.predict(img))
    throttleResult = 1.0 - fixedSpeed / speedLimit
    print(steeringProcess, throttleResult, fixedSpeed)
    processCtrl(steeringProcess, throttleResult)

    datasetSimulation[0].append(steeringProcess)
    datasetSimulation[1].append(throttleResult)
    datasetSimulation[2].append(fixedSpeed)
```

Następnie łączymy się z serwerem i wysyłamy polecenia do naszego symulatora:

```
@socket_io.on('connect')
def connect(sid, environ):
    print('Udało się połączyć!\n')
    print('UKLAD KIEROWNICZY\t PRZEPUSTNICA\t PRĘDKOŚĆ\n')
    processCtrl(0, 0) # Tutaj mamy dane dotyczące sterowania, a następnie prędkość
                      # Na początku wysyłamy po prostu dwa zera
```

W kolejnym kroku następuje wyemitowanie zdarzenia do serwera za pomocą metody *emit()*:

```
def processCtrl(steeringProcess, throttleResult):
    socket_io.emit('steer', data={
        'steering_angle': steeringProcess.__str__(),
        'throttle': throttleResult.__str__()
    })
```

Proces uruchomiania symulacji kończy załadowanie modelu w formacie .h5:

```
simulationModel = load_model('MyModel.h5')
simulationApp = socketio.Middleware(socket_io, simulationApp)
```

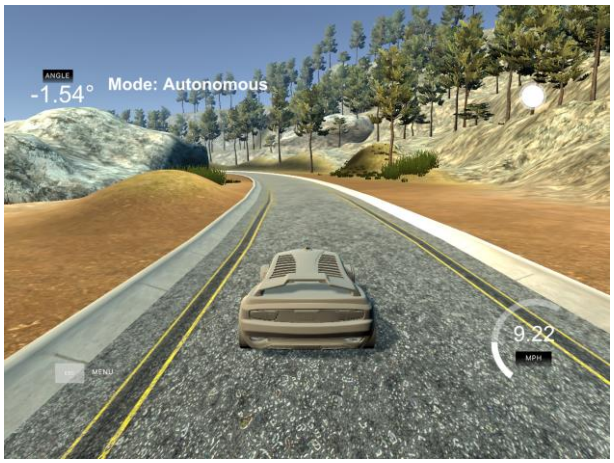
Takie zachowanie jest specyficzne dla tego symulatora, dlatego używamy portu 4567:

```
eventlet.wsgi.server(eventlet.listen(('', 4567)), simulationApp)
```

Następnie serwer symulatora przesyła parametry 3 zmiennych (układ kierowniczy, przepustnica i prędkość) w czasie rzeczywistym. Dane te można wyświetlić w konsoli, jak pokazano w Tabeli 7, a także zapisać jako osobny plik do późniejszej analizy. Każdy z parametrów stale zmienia swoją wartość, ze względu na niejednorodność środowiska, w którym znajduje się jadący pojazd.

Tabela 7. Parametry pojazdu w czasie rzeczywistym

```
(2400) wsgi starting up on http://0.0.0.0:4567
(2400) accepted ('127.0.0.1', 50693)
Udało się połączyć!

UKLAD KIEROWNICZY       PRZEPUSTNICA          PRĘDKOŚĆ

-0.10338                 0.8686                1.314
-0.10237                0.781                 2.19
-0.08551                0.7372                2.628
-0.05425                0.84862               1.5138
-0.04416                0.94622               0.5378

.....

-0.04174                0.07835               9.2165
-0.02751                0.07835               9.2165

127.0.0.1 - - [08/Dec/2021 02:16:05] "GET /socket.io/?EIO=4&transport=websocket HTTP/
1.1" 200 0 282.061040
wsgi exiting
(2400) wsgi exited, is_accepting=True
```



Rys. 19. Zrzut ekranu podczas symulacji autonomicznej jazdy

Po uruchomieniu symulacji pojazdu w trybie jazdy autonomicznej widać, że model jest w stanie przechwytywać dane w czasie rzeczywistym, jadąc prawie środkiem pasa, choć skręca lekko w lewo (podobny styl jazdy był podczas zbierania danych). Na Rys. 19 pojazd porusza się z prędkością 9,22 mil na godzinę (nie przekraczając maksymalnej prędkości 10 mil na godzinę) i skręca w lewo pod kątem -1,54 stopnia bez interwencji człowieka. Mimo że nie uruchamiamy go z dużą prędkością, ponieważ trenowaliśmy tylko dla kilku tysięcy obrazów, co prawie nie liczy się w 72 godzinach danych treningowych, które Nvidia otrzymała dla swojego modelu [7]. Biorąc pod uwagę, że mieliśmy tak małą ilość danych, widać, że opracowany w pracy mój model pojazdu autonomicznego jest wystarczająco efektywny by poruszać się poprawnie po naszym torze.

Wartym podkreślenia w tym modelu jest to, że model się trenował na trasie z bardzo różnymi rodzajami pasów na drodze. Na drodze mamy namalowane paski, żółte pasy, czasami trasa przechodzi przez most. Czasami na moście w ogóle nie mamy pasów i powierzchnia mostu zupełnie inna niż sama droga, ale pojazd nadal jest w stanie przejść tą ścieżką bez żadnych problemów. To samo w sobie jest wielkim sukcesem i widać, że model jest w stanie działać zgodnie z oczekiwaniami i może nauczyć się prowadzić nie tylko na podstawie stwierdzeń *if* i *else*. Na przykład, jeśli widzisz to, zrób to, zobacz to, a potem zrób to. Możemy również dodać więcej danych, możemy dodać większą elastyczność w zakresie powiększania obrazów. W każdym razie jest wiele parametrów, które możemy zmienić, aby usprawnić jazdę pojazdu.

4.2. Zapisywanie otrzymanych danych w pliku .csv

Po przejechaniu pojazdem autonomicznym po zamkniętym torze zmienna *datasetSimulation* chwilowo przyjęła wartości 3 parametrów (układ kierowniczy, przepustnica i prędkość), które teraz należy zapisać w osobnym pliku do późniejszego wyświetlenia na wykresie. Uzyskane dane zapisujemy w pliku *simulation_MyModel.csv* W tym celu używamy metody *csv.writer* [32] z pakietu CSV:

```
with open('simulation_MyModel.csv', 'w') as f:

    write = csv.writer(f)
    write.writerows(datasetSimulation)
```

Tak zwany format CSV (wartości oddzielone przecinkami) jest najpopularniejszym formatem importu i eksportu arkuszy kalkulacyjnych i baz danych. Zapisanie wyników symulacji w pliku daje nam możliwość archiwizacji zachowania różnych modeli i późniejszego porównania wydajności każdego z nich.

Te same kroki symulacyjne opisane w rozdziale 4 wykonujemy z modelem Nvidii, które go dane również zostaną zapisane w osobnym pliku *simulation_NvidiaModel.csv*. Teraz mamy 2 pliki z danymi o trzech parametrach podczas symulacji z wykorzystaniem modelu Nvidii oraz własnego modelu.

5. Porównanie układów sterowania Nvidii i własnego

W celu porównania obu modeli importujemy zapisane dane, które otrzymaliśmy podczas symulacji. W tym celu wykorzystamy bibliotekę oprogramowania *pandas* do przetwarzania i analizy danych. Wstępnie dokonamy transpozycji list i nadamy nazwy każdej kolumnie.

Przeprowadzamy importowanie danych z pliku *simulation_NvidiaModel.csv*:

```
wykres1t = pandas.read_csv('simulation_NvidiaModel.csv', header=None)
wykres1 = wykres1t.T #Wstępne transponowanie listy
wykres1.columns = ['Uklad kierowniczy', 'Przepustnica', 'Predkosc']
wykres1.head()
```

Tabela 8. Wyświetlanie pierwszych 5 wartości 3 kolumn z pliku simulation_NvidiaModel.csv

	Uklad kierowniczy	Przepustnica	Predkosc
0	-0.103382	0.86860	1.3140
1	-0.102366	0.78100	2.1900
2	-0.085509	0.73720	2.6280
3	-0.054250	0.84862	1.5138
4	-0.044160	0.94622	0.5378

Następnie importujemy dane z pliku *simulation_MyModel.csv*:

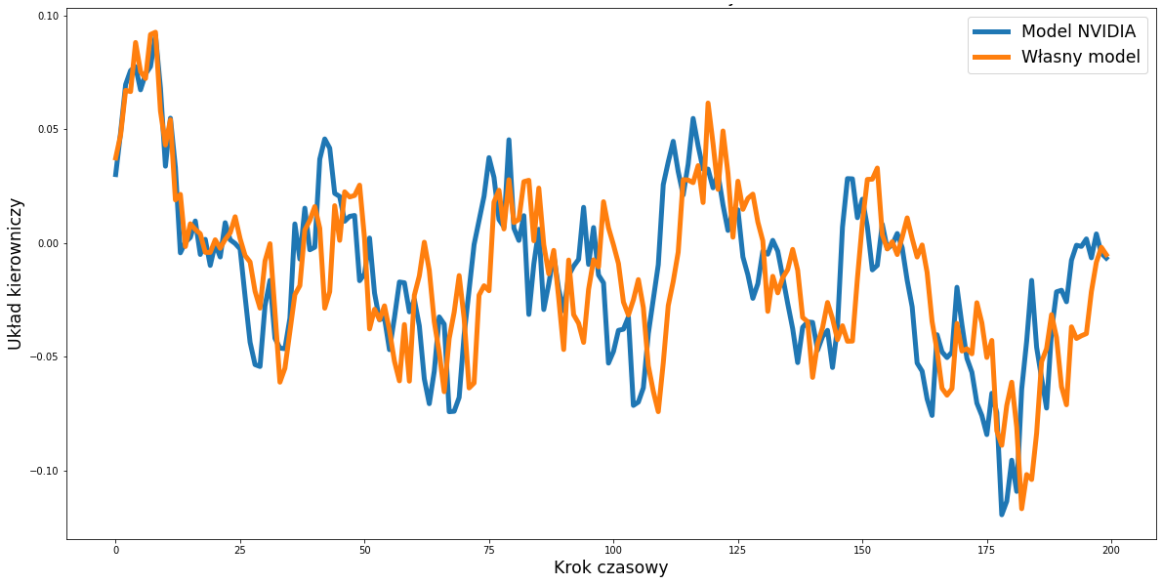
```
wykres2t = pandas.read_csv('simulation_MyModel.csv', header=None)
wykres2 = wykres2t.T # Wstępne transponowanie listy
wykres2.columns = ['Uklad kierowniczy', 'Przepustnica', 'Predkosc']
wykres2.head()
```

Tabela 9. Wyświetlanie pierwszych 5 wartości 3 kolumn z pliku simulation_MyModel.csv

	Uklad kierowniczy	Przepustnica	Predkosc
0	-0.103380	0.86860	1.3140
1	-0.102366	0.78100	2.1900
2	-0.085509	0.73720	2.6280
3	-0.080727	0.83251	1.6749
4	-0.076457	0.82171	1.7829

W celu porównania zachowania pojazdu autonomicznego przy użyciu obu modeli, na jednym wykresie wyświetlimy 300 wartości układu kierowniczy. Wartość 0 na prostej pionowej wykresu oznacza jazdę na wprost, natomiast wartości ujemne oznaczają jazdę w lewo, a wartości dodatnie w prawo. Linia pozioma przedstawia liczbę wszystkich zebranych danych przez pewny czas (w naszym przypadku jest to 300 wartości), co jest równoważne z jednostkowym krokiem czasowym.

```
# Wartości osi X i Y
time_step = range(0, 200)
steering1 = wykres1['Uklad kierowniczy'][100:300]
steering2 = wykres2['Uklad kierowniczy'][100:300]

# Zmiana rozmiaru kształtu na 20x10
pyplot.figure(figsize=(20, 10))

pyplot.plot(time_step, steering1, label="Model NVIDIA", linewidth=5)
pyplot.plot(time_step, steering2, label="Własny model", linewidth=5)
plt.legend(loc="upper right", fontsize="xx-large")

pyplot.xlabel('Krok czasowy', fontsize="xx-large"))
pyplot.ylabel('Uklad kierowniczy', fontsize="xx-large"))
```



Rys. 20. Porównanie układów kierowania modelu NVIDII z własnym

Na podstawie wizualizacji wyników z dwóch Tabel 9 i 10 na Rys. 20 widzimy, że na początku dane wyglądają tak samo, ponieważ zaczynają z tego samego miejsca. Jednak w dalszej części każdy model interpretuje zachowanie na drodze na swój własny sposób. Mimo widocznej różnicy w przebiegu sterowania pojazdem przez model Nvidii i własny, model opracowany przeze mnie przez cały czas jazdy pozostaje w obszarze testowej trasy.

Zakończenie

Celem pracy było zbadanie struktury konwolucyjnej sieci neuronowej, wykorzystanie jej do rozpoznawania obiektów na obrazie i na podstawie tego stworzenie modelu pojazdu autonomicznego. Na praktycznym przykładzie w symulatorze udało się nam porównać 2 modele sieci neuronowych i stwierdzić, że ich działanie jest bardzo podobne. Uważam, że cel pracy został osiągnięty w dość wysokim stopniu i można wyciągnąć następujące wnioski:

1) Technologia głębokiego uczenia to potężne narzędzie do tworzenia pełnej automatyzacji procesów.
2) Konwolucyjne sieci neuronowe są w stanie skutecznie rozpoznawać obiekty na obrazach.
3) Budowa modelu pojazdu autonomicznego to proces twórczy, w którym możliwe jest wniesienie własnych pomysłów. Na przykład, napisanie własnego modelu, który będzie konkurował z popularnymi rozwiązaniami znanych firm, jest całkiem możliwe i wymaga odrobiny wyobraźni i zrozumienia struktury sieci neuronowej.

Oczywiście ten produkt nie jest idealny i bezbłędny. Samochód jest w stanie jeździć po wcześniej zadanej trasie, natomiast przy nowej mapie mogą wystąpić trudności z rozpoznaniem drogi. Zawsze istnieje możliwość zwiększania ilości danych badawczych, dodawania większej elastyczności w zakresie powiększania obrazów i ich modyfikacji. W taki sposób możemy wpływać na wiele innych parametrów, które są w stanie polepszyć skuteczność głębokiego uczenia.

Bibliografia

[1] Tesla Inc., „Autopilot”. URL: https://www.tesla.com/autopilot. [Data uzyskania dostępu: 29 12 2021].

[2] NVIDIA Corporation, „SOLUTIONS FOR SELF-DRIVING CARS & AUTONOMOUS VEHICLES”. URL: https://www.nvidia.com/en-us/self-driving-cars/. [Data uzyskania dostępu: 29 12 2021].

[3] Nissan Motor Co., „What is propilot assist?”. URL: https://www.nissanusa.com/experience-nissan/news-and-events/nissan-propilot-assist.html. [Data uzyskania dostępu: 10 01 2022].

[4] Andrew Trask, „Grokking Deep Learning,” Manning, 2019.

[5] Ron Kneusel, „Practical Deep Learning,” No Starch Press, 2021.

[6] Udacity Inc., „Udacity's Self-Driving Car Simulator”. URL: https://github.com/udacity/self-driving-car-sim. [Data uzyskania dostępu: 29 12 2021].

[7] Mariusz Bojarski, „End to End Learning for Self-Driving Cars,” 2016.

[8] Stepan Garciu, „Project FashionMNIST”. URL: https://github.com/StefanGarcziu/DeepLearning--UJK/blob/main/2_Project__FashionMNIST.ipynb. [Data uzyskania dostępu: 29 12 2021].

[9] Bruckner Calderon, „Autonomous Driving on Nvidia Dave-2”.

[10] Siraj Raval, „How to simulate a self driving car”. URL: https://github.com/llSourcell/How_to_simulate_a_self_driving_car. [Data uzyskania dostępu: 21 01 2021].

[11] Nikhil Arora, „Self-Driving-Car”. URL: https://github.com/Nikhil9786/Self-Driving-Car-Using-Deep-Learning. [Data uzyskania dostępu: 21 01 2022].

[12] Tek Chhetri, „Self-driving Car”. URL: https://github.com/tbchhetri/selfDrivingCar. [Data uzyskania dostępu: 21 01 2022].

[13] Światowa Organizacja Zdrowia , „Road traffic injuries”. URL: https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries. [Data uzyskania dostępu: 29 12 2021].

[14] Tesla Inc., „Tesla Vehicle Safety Report”. URL: https://www.tesla.com/VehicleSafetyReport. [Data uzyskania dostępu: 03 01 2022].

[15] Project Jupyter, [Online]. URL: https://jupyter.org/. [Data uzyskania dostępu: 03 01 2022].

[16] B. Chen, „Create Virtual Environment using “virtualenv” and add it to Jupyter Notebook”. URL: https://towardsdatascience.com/create-virtual-environment-using-virtualenv-and-add-it-to-jupyter-notebook-6e1bf4e03415. [Data uzyskania dostępu: 03 01 2022].

[17] Google Brain, „An end-to-end open source machine learning platform”. URL: https://www.tensorflow.org/. [Data uzyskania dostępu: 21 01 2022].

[18] SudKul, „Udacity's Self-Driving Car Simulator”. URL: https://github.com/udacity/self-driving-car-sim. [Data uzyskania dostępu: 03 01 2022].

[19] Mirko Stojiljković, „Split Your Dataset With scikit-learn's train_test_split()”. URL: https://realpython.com/train-test-split-python-data/. [Data uzyskania dostępu: 15 01 2022].

[20] David Cournapeau, „Scikit-learn. Machine Learning in Python”. URL: https://scikit-learn.org/stable/. [Data uzyskania dostępu: 15 01 2022].

[21] OpenCV team. URL: https://opencv.org/. [Data uzyskania dostępu: 03 01 2022].

[22] „Python | yield Keyword”. URL: https://www.geeksforgeeks.org/python-yield-keyword/. [Data uzyskania dostępu: 15 01 2022].

[23] Tomasz Rodak, „Podstawy programowania w analizie danych”. URL: http://en.math.uni.lodz.pl/~rodakt/dane/analiza_danych/17.18/wyklad_X.html#yield. [Data uzyskania dostępu: 04 01 2022].

[24] Casper Hansen, „Activation Functions Explained - GELU, SELU, ELU, ReLU and more”. URL: https://mlfromscratch.com/activation-functions-explained/. [Data uzyskania dostępu: 03 01 2022].

[25] Shubham__Ranjan, „Python | Pandas Index.summary()”. URL: https://www.geeksforgeeks.org/python-pandas-index-summary/#:~:text=summary()%20function%20return%20a,we%20have%20for%20the%20dataframes.&text=Example%20%231%3A%20Use%20Index.,the%20summary%20of%20the%20Index.. [Data uzyskania dostępu: 16 01 2022].

[26] François Chollet, „Keras: the Python deep learning API”. URL: https://keras.io/. [Data uzyskania dostępu: 16 01 2022].

[27] „keras.Model.fit”. URL: https://www.kite.com/python/docs/keras.Model.fit. [Data uzyskania dostępu: 16 01 2022].

[28] The Python Software Foundation, „time — Time access and conversions”. URL: https://docs.python.org/3/library/time.html. [Data uzyskania dostępu: 16 01 2022].

[29] „Czym są pliki H5?”. URL: https://pliki.wiki/extension/h5. [Data uzyskania dostępu: 05 01 2022].

[30] Guillermo Rauch, „Socket.IO”. URL: https://socket.io/. [Data uzyskania dostępu: 17 01 2022].

[31] „What is WSGI?”. URL: https://wsgi.readthedocs.io/en/latest/what.html. [Data uzyskania dostępu: 17 01 2022].

[32] The Python Software Foundation, „CSV File Reading and Writing”. URL: https://docs.python.org/3/library/csv.html. [Data uzyskania dostępu: 17 01 2022].

Stepan Garciu Kielce, dn. ……
imię i nazwisko studenta
135843
numer albumu
Inżynieria danych, Metody sztucznej inteligencji
kierunek, specjalność
studia stacjonarne I stopnia
rodzaj studiów, forma studiów

Oświadczenie

Przedkładając w roku akademickim 2021/2022 pracę inżynierską pod tytułem:
„MODEL POJAZDU AUTONOMICZNEGO''

oświadczam, że:

- - pracę napisałem samodzielnie,
- - praca nie stanowi istotnego fragmentu lub innych elementów cudzego utworu,
- - praca nie narusza żadnych innych istniejących praw autorskich,
- - wykorzystane w pracy materiały źródłowe zastosowane zostały z zachowaniem zasad prawa cytatu,
- - wersja elektroniczna (na nośniku elektronicznym i/lub w systemie Wirtualna Uczelnia) pracy jest tożsama z wersją drukowaną.

Równocześnie oświadczam, że jestem świadomy, iż na podstawie art. 15a ustawy z 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. Nr 24 poz. 83 ze zm.) Uniwersytetowi Jana Kochanowskiego w Kielcach przysługuje pierwszeństwo w opublikowaniu mojej pracy magisterskiej w terminie 6 miesięcy od daty jej obrony.

W przypadku nieskorzystania przez Uniwersytet Jana Kochanowskiego w Kielcach z prawa pierwszeństwa publikacji wyrażam zgodę na udostępnianie mojej pracy magisterskiej przez Uniwersytet Jana Kochanowskiego w Kielcach dla celów naukowych i dydaktycznych.

Prawdziwość powyższego oświadczenia potwierdzam własnoręcznym podpisem.

……
czytelny podpis studenta