Weryfikacja możliwości sterowania łazikiem za pomocą sieci neuronowych

(Veryfing the remote control capabilities of rover using neural networks)

Paweł Dybiec

Praca licencjacka

Promotor: dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

12 II 2018

Streszczenie

Sieci neuronowe są w stanie kierować samochodem na podstawie obrazu z kamery[1]. Tematem tej pracy jest implementacja i przetestowanie autonomicznej jazdy łazika Aleph 1 korzystającej z konwolucyjnych sieci neuronowych. Na początku tej pracy zostanie przybliżony sposób działania sieci neuronowych oraz systemu ROS wykorzystanego szeroko w robotyce. Następnie przedstawię sposób proces trenowania sieci neuronowych sterujących łazikiem oraz modelem w symulatorze. Na koniec przedstawię wyniki treningu oraz działanie modelu.

This dissertation investigates control capabilities for convolutional neural network. This essay aims to explain basics of used model and software. In the end I will present results of training and the way this model works in practice.

Spis treści

Rozdział 1.

Preliminaria

Ta praca powstała w ramach przedmiotu "Projekt: autonomiczna jazda łazikiem".

Podczas realizacji przedmiotu powstało wiele rozwiązań dla zadań z "konkursów łazikowych".

Żaden spośród łazików biorących udział w University Rover Challenge nie używa sieci neuronowych bezpośrednio do nawigacji, ale prawie wszystkie używają ROS (Robot Operating System) jako podstawy całego oprogramowania. Z tego powodu rozdziale zostaną poruszone poniższe zagadnienia:

- Łazik Aleph 1
- Podstawy sieci neuronowych
- Architektura ROS

1.1. Łazik Aleph 1

Łazik Aleph 1 powstał z inicjatywy Koło Pasjonatów Mechaniki i Informatyki „Continuum"[1] w roku 2014. Od tego czasu został zaprezentowany na konkursach takich jak European Rover Challenge (ERC) oraz University Rover Challenge (URC). Przez ostatnie dwa lata łazik był doceniany na konkursie URC (w roku 2016 zajął 3 miejsce, a w roku 2017 – 2 miejsce).

Podczas konkursu URC pojazdy były oceniane w czterech kategoriach[2]:

- Science Cache Task – pobieranie i badanie próbek w terenie

[1] Strona Koła Pasjonatów Mechaniki i Informatyki „Continuum": http://continuum.uni.wroc.pl/

[2] Regulamin konkursu URC: http://urc.marssociety.org/home/requirements-guidelines

- Extreme Retrieval and Delivery Task – transport pakunku w różnych warunkach terenowych
- Equipment Servicing Task – zdolności manualne (umiejętność podnoszenia, przenoszenia obiektów, obsługa przycisków, przełączników i innych narzędzi)
- Autonomous Traversal Task – umiejętność poruszania się pomiędzy wyznaczonymi punktami

1.2. Podstawy sieci neuronowych

Głębokie sieci neuronowe (deep neural networks) to popularny model w uczeniu maszynowym. Celem sieci jest przybliżenie funkcji f^*, przyporządkowującej argumentom x wartości y, funkcją $f(x, \theta) = y$ oraz znalezienie parametru θ, który da najlepsze przybliżenie.

1.2.1. Jak działają

Sieci neuronowe są zazwyczaj złożone z wielu różnych funkcji, nazywanych warstwami. Przykładowo, $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$, wtedy $f^{(3)}$ jest wartwą wyjściową, której wyniki powinny odpowiadać funkcji f^*. Wyniki dla pozostałych warstw nie są znane, z tego powodu nazywa się je ukrytymi warstwami.

1.2.2. Jak trenować

W wyniku trenowania chcemy znaleźć takie θ, żeby $f^*(x) \approx f(x, \theta)$. W tym celu należy zdefiniować funkcję kosztu $L(\theta)$ tzn. odległości modelu od celu, zależną od θ, np. dla problemu regresji, L to średni błąd kwadratowy dla danych uczących. Dla takiej funkcji chcielibyśmy teraz znaleźć minimum. Minimum globalne może być trudne do znalezienia, ale minima lokalne zazwyczaj są wystarczające do większości zastosowań.

Gdyby L byłoby funkcją jednej zmiennej, wystarczyłoby zacząć w losowym miejscu i wielokrotnie wykonać następujący krok $x = x - \epsilon L'(x)$, aby dotrzeć do minimum lokalnego. Dla funkcji wielu zmiennych podobny algorytm działa, ale pochodną należy zastąpić gradientem $x = x - \epsilon \nabla_x L(x)$.

1.2.3. Popularne warstwy

Warstwa liniowa (linear lub dense)

jest najbardziej podstawową warstwą. Każdy element wyjściowy (m wartości) jest kombinacją wszystkich wejść (n wartości) danej warstwy (powiększoną o stałą).

Zatem taka warstwa jest parametryzowana macierzą rozmiaru $n \cdot m$ oraz wektorem rozmiaru m.

Dwie takie sąsiednie warstwy liniowe można by zredukować do jednej, ponieważ $W_2 \cdot (W_1 \cdot x + b_1) + b_2 = W \cdot x + b$, gdy $W = W_2 \cdot W_1$ oraz $b = W_2 \cdot b_1 + b_2$. Zatem dowolnie głęboką sieć złożoną z takich warstw możnaby zredukować do jednej takiej warstwy, ale po każdej funkcji liniowej aplikuje się funkcję nieliniową np. $tanh$ lub $relu$ ($\max(0, x)$). Dzięki temu sieci neuronowe są w stanie pokryć znacznie większą przestrzeń funkcji niż tylko liniowe.

Inną warstwą, już specjalizowaną w przetwarzaniu danych położonych na pewnej kracie, jest warstwa konwolucyjna. Przykładowym wejściem dla takiej warstwy może być dwuwymiarowa siatka pikseli. Natomiast wyjściem jest obraz o zbliżonej (lub tej samej) rozdzielczości, którego wartość jest kombinancją liniową spójnego bloku pikseli z wejścia. Ważną cechą takiej warstwy jest fakt, że wszystkie wyjścia korzystają z tych samych parametrów, co powoduje że znajdują te same wzorce położone w innych miejscach.

Kolejnym typem warstw specjalizowanym w przetwarzaniu obrazów jest pooling. Dzieli ona wejście na spójne rozłączne bloki, na każdym z nich osobno aplikuje funkcję np. max lub avg. Taka operacja powoduje niewrażliwość na małe przemieszczenia wejść. Dodatkowo zmniejsza to rozmiar wejścia w kolejnych warstwach, co zmniejsza liczbę parametrów.

1.2.4. Uwagi

Konwolucyjne sieci neuronowe bardzo dobrze radzą sobie z widzeniem maszynowym, są w stanie klasyfikować bezproblemowo obrazki[2]. Ale w przypadku patrzenia na tylko jedną kratkę nie są wstanie wyciągnąć wniosków. Co w przypadku nawigacji oznacza, że proces sterowania jest tylko ciągiem spontanicznych decyzji bez planowania trasy. Dodatkową konsekwencją takiej architektury jest ukryte założenie, że dla każdego obrazu z kamery jest tylko jedna poprawna odpowiedź.

1.3. Architektura ROS

ROS (Robot Operating System) to rozbudowany framework przeznaczony do programowania robotów. Składa się na niego wiele bibliotek oraz narzędzi mających na celu zbudowanie klastra komputerów tworzących spójny system. Dostarcza on abstrakcję nad sprzętem, środki komunikacji między procesami oraz oraz inne funkcjonalności gwarantowane przez typowy system operacyjny. Ze względu na modułową budowę oraz architekturę peer-to-peer procesy mogą bezproblemowo działać na różnych komputerach.

1.3.1. Node

Podstawową jednostką w ROS-ie jest wierzchołek (node), jego głównym zadaniem jest wykonywanie obliczeń. Wierzchołki razem tworzą graf i komunikują się za pomocą tematów (topic).

Taka architektura (inspirowana budową mikrojądra) w porównaniu do architektury monolitycznej, zapewnia lepszą ochronę przed błędami. Dodatkowo pojedyńczy element można bezproblemowo przepisać także w innym języku programowania.

1.3.2. Topic

Tematy (topic) pozwalają bezproblemowo zapewnić komunikację międzyprocesową w ROS-ie. Każdy node może zadeklarować chęć nadawania bądź nasłuchiwania na danym temacie. Przykładowo, moduł jazdy autonomicznej może zasubskrybować obraz z kamery Kinect i publikować na temacie reprezentującym kierunek ruchu. Tematy są otypowane, co gwarantuje że wszystkie wiadomości wysłane na tym samym temacie mają taką samą strukturę.

1.3.3. Gotowe moduły

ROS dostarcza wiele gotowych modułów pozwalających szybko rozpocząć projekt. Jednym z nich jest Odom, który zbiera informacje o położeniu i prędkości z wielu źródeł danych i łączy je w nowy strumień danych (o większej pewności). Przykładowo, dane może zbierać z czujnika GPS, prędkości obrotowej kół oraz akcelerometru.

Inny moduł potrafi tworzyć mapy na podstawie obrazu z kamery oraz mapy głębokości. Wynik tej rekonstrukcji można obejrzeć za pomocą innych usług służących do wizualizacji różnych typów danych takich jak obraz, wartości zmieniających się w czasie, chmury punktów lub mapy terenu.

Rozdział 2.

Trenowanie sieci i zbieranie danych

W celu autonomicznej jazdy wytrenowałem konwolucyjną sieć neuronową (CNN – Convolutional Neural Network) przetwarzającą obraz z kamery bezpośrednio w porządaną prędkość liniową oraz obrotową. Takie podejście pozwala szybko zbierać dane uczące. Wystarczy tylko nagrać obraz z kamery oraz prędkość nadaną przez kierowcę.

Wersja sterująca w symulatorze powstała aby odrzucić modele, które nie radzą sobie w tak prostych warunkach. Dodatkowo zbieranie danych oraz testowanie modelu jest łatwiejsze. Po pierwsze, nie wymaga przygotowywania sprzętu. Poza tym, opuszczenie toru przez model nie stwarza zagrożenia uszkodzenia łazika lub jego otoczenia.

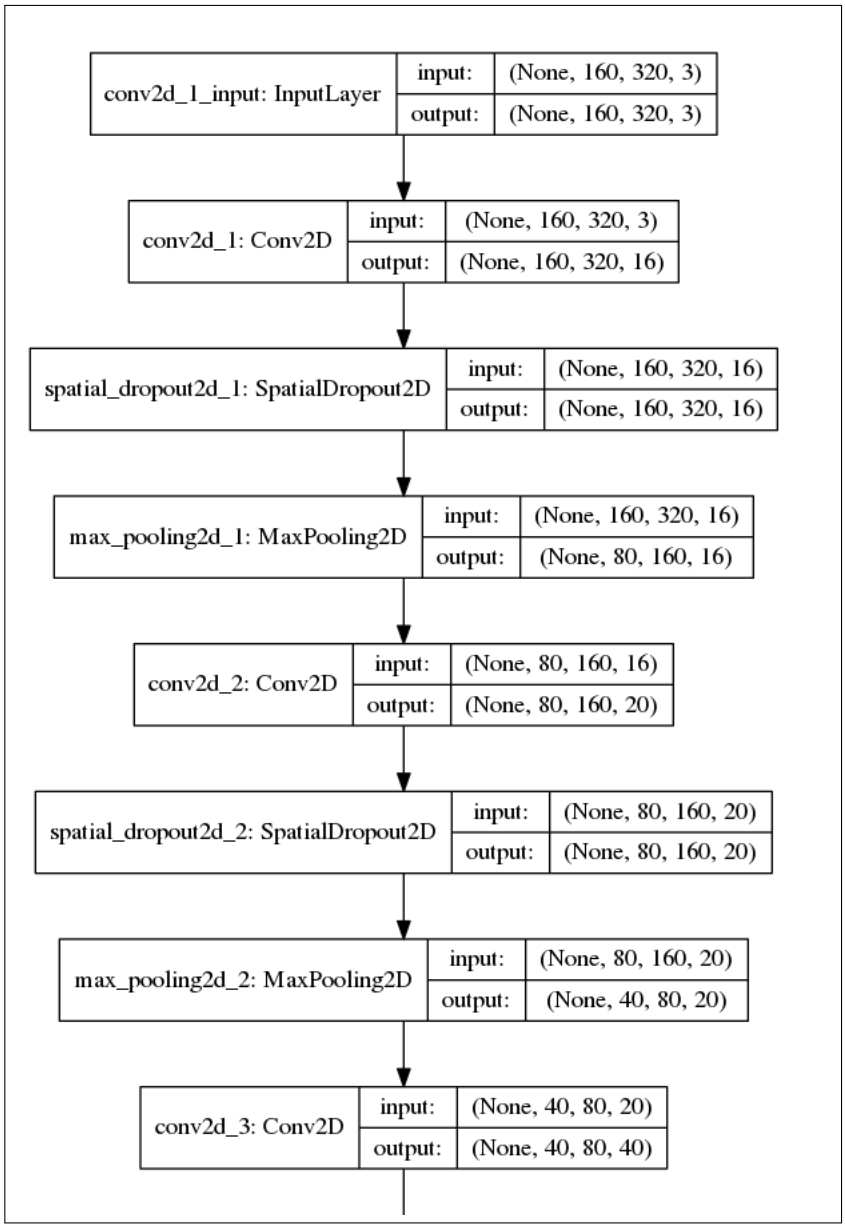
Architektura sieci pochodzi z rozwiązania *chauffeur* w konkursie Udacity – Self Driving Car[1], ale została zaadaptowana do interfejsu symulatora oraz łazika. Sieć wykorzystana w moim projekcie, w odróżnieniu od rozwiązania konkursowego, zadaje też prędkość. Konwersja z rosbaga (format nagrań ROS-a) do naszego formatu oraz sam symulator zostały wykonane przez innych członków projektu.

2.1. Trening na symulatorze

Nagrania do nauki na symulatorze zostały wykonane przeze mnie i dwóch innych uczestników projektu. Łączna długość nagrań wynosi około 50 minut, z czego 5 minut zostało przeznaczone na zbiór walidacyjny, a reszta była zbiorem uczącym.

Obrazy pochodzą z trzech kamer: jednej skierowanej na wprost, dwóch obróconych o 20 stopni względem środkowej. Podczas uczenia wykorzystywany był obraz ze

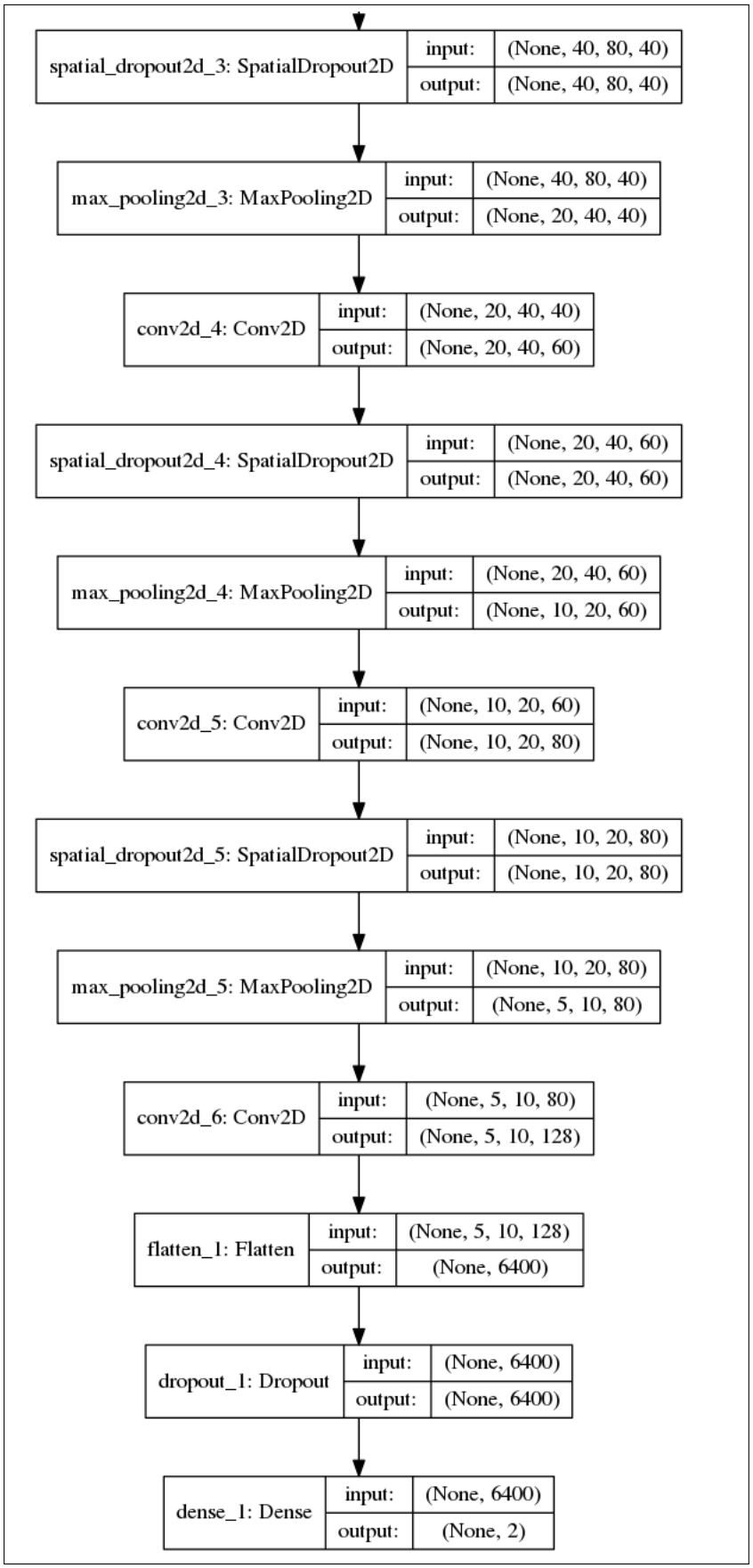
[1] Repozytorium dostępne pod https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models



wszystkich kamer. Dla kamery środkowej pożądanym wynikiem były nagrane dane ze sterowania. Dla kamery lewej wynik był nieznacznie zaburzony w prawo, natomiast analogicznie dla kamery prawej – w lewo. Dodatkowo, obraz z kamery środkowej był dodany w postaci symetrycznego odbicia ze zmienionym kierunkiem skrętu.

2.2. Trening z nagrań łazika

Do treningu zostało wykorzystane około 150GB nagrań z łazika, zawierających obraz z kamery oraz informacje o sterowaniu. Jest to kilkanaście objazdów po podziemnym garażu. Na większości ujęć widać kratkę zazwyczaj zgodną z kierunkiem jazdy, ale zakręcającą pod kątem prostym. Około 6GB danych zostało wykorzysta-



Rysunek 2.1: Architektura sieci

nych jako zbiór walidacyjny.

Nagrania pochodzą z jednej fizycznej kamery. Dla każdego ujęcia zostało dodane jego odbicie lustrzane z przeciwnym kątem skrętu.

Rozdział 3.

Wyniki sieci

Wytrenowana sieć potrafi przejechać zarówno cały tor na symulatorze, jak i podziemny garaż instytutu. Na dodatek sieć trenowana pod symulator uczyła się jeździć tylko przeciwnie do ruchu wskazówek zegara. Po ustawieniu modelu w przeciwnym kierunku, sieć potrafi bezproblemowo przejechać cały tor.

3.1. Na co zwraca uwagę

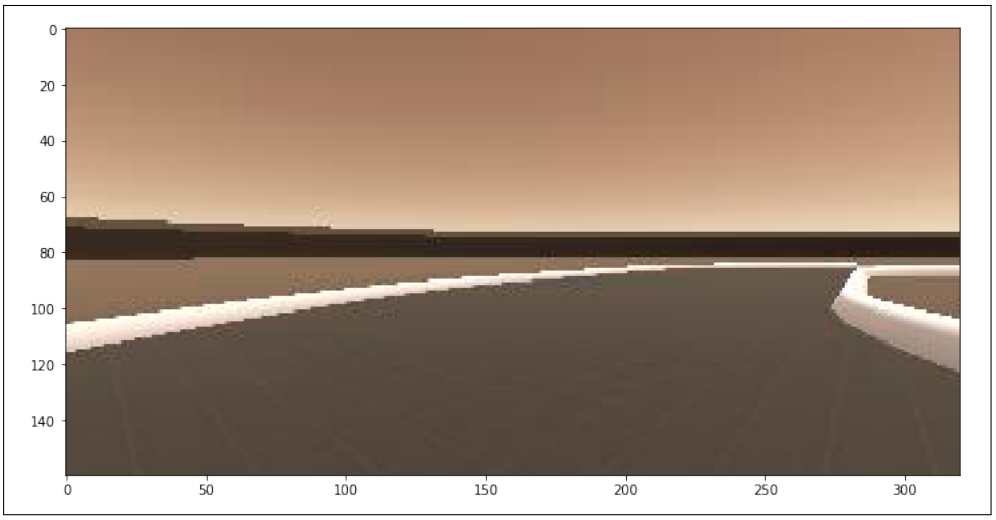
Aktywność sieci dla obrazków została wygenerowana za pomocą metody Integrated Gradients[3].

Co było oczywiste w przypadku symulatora, sieć zwraca głównie uwagę na miejsca, gdzie pojawiają się granice drogi 3.2. Co ciekawe, reaguje też na ścianę tworzącą horyzont, ponieważ zmienia wygląd w zależności od odległości i może pomóc w orientacji (na trasie treningowej).

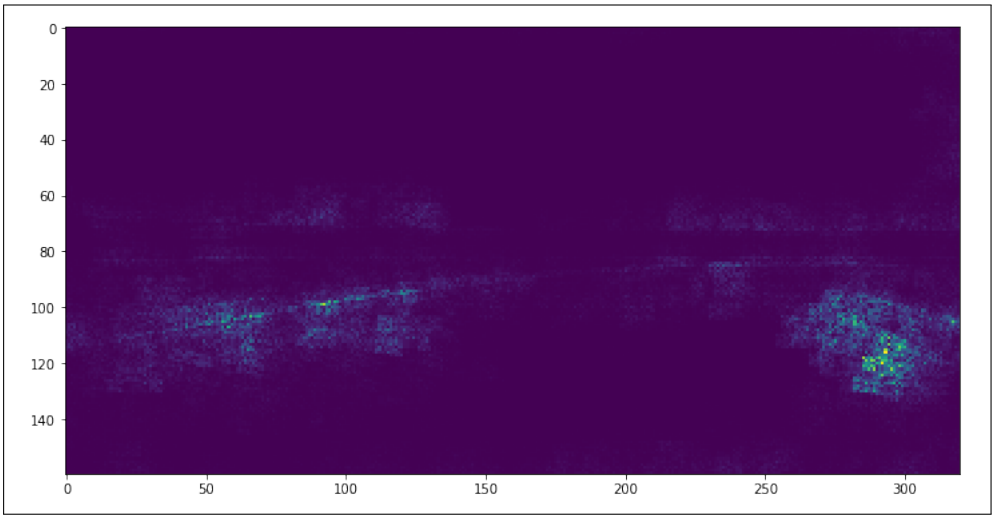
Z kolei dla łazika intensywność w najbardziej aktywnym miejscu jest dużo mniejsza 3.4. Oznacza to, że nie sugeruje się tylko jednym obszarem z kamery. Najbardziej jednak zwraca uwagę na kratkę na podłodze, która mogłaby wystarczyć do nawigacji.

3.2. W porównaniu do nagrania

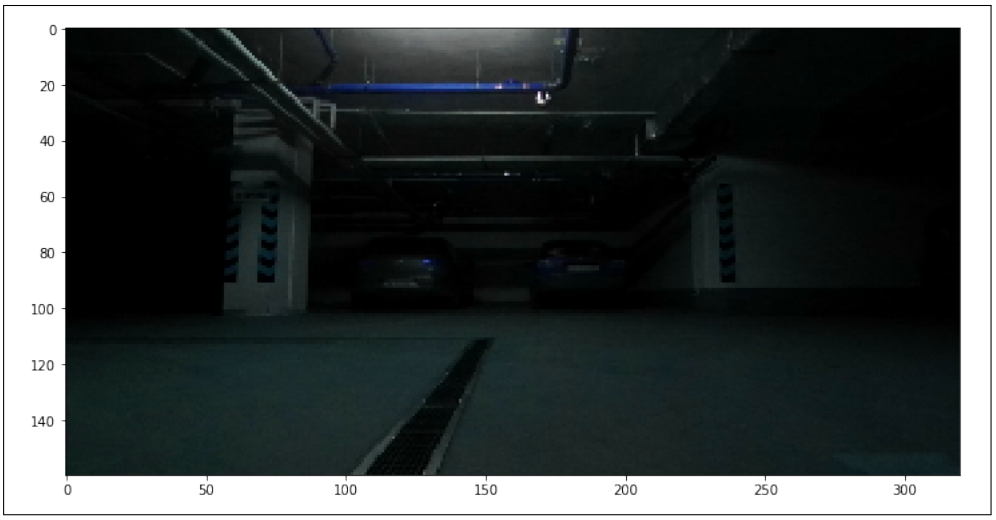
Na wykresie 3.5 widać, że sieć (pomarańczowy kolor) mniej gwałtownie zmienia szybkość obrotu niż kierowca (kolor niebieski). Jednak sieć reaguje w podobnych momentach co kierowca na konieczność wykonania skrętu.



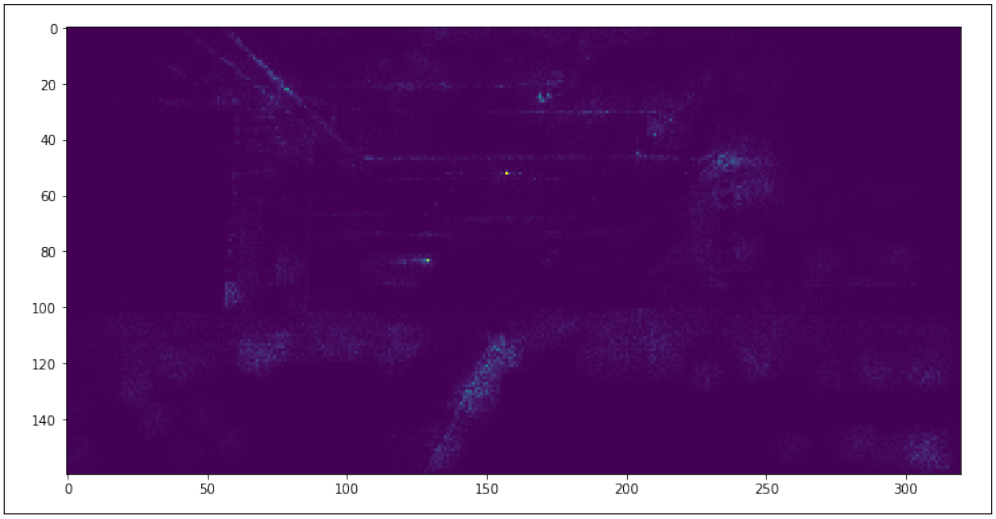
Rysunek 3.1: Obraz z symulatora



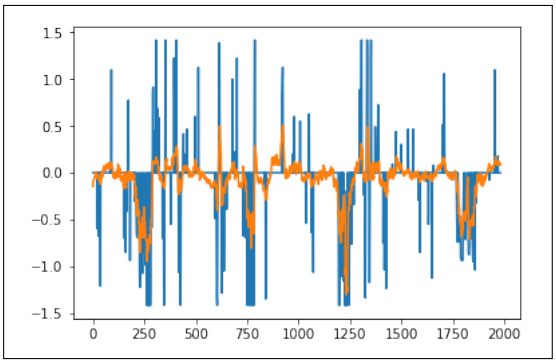
Rysunek 3.2: Na co sieć patrzy, symulator



Rysunek 3.3: Obraz z nagrania



Rysunek 3.4: Na co sieć patrzy, nagranie



Rysunek 3.5: Prędkość obrotowa: sieć vs kierowca

3.3. Wpływ architektury

W przypadku sieci pod symulator, usunięcie niektórych warstw konwolucyjnych pozwalało modelowi nadal utrzymywać się na torze, natomiast taka sama zredukowana architektura nie radziła sobie dobrze w przypadku nagrań z prawdziwego łazika. Z kolei usunięcie nieliniowości z warstw konwolucyjnych ograniczyło zdolności sieci do takiego stopnia, że nie potrafiła się utrzymać na wirtualnym torze.

Usunięcie dropoutu bardzo szybko powodowało overfitting i sieć radziła sobie dobrze tylko na danych uczących. Z kolei dodanie warstw liniowych na końcu nie poprawiało, ani nie pogorszało zbytnio wydajności sieci, przynajmniej dla nagrań z symulatora. Na tej podstawie można wywnioskować, że większość interesujących cech obrazu została już znaleziona w ramach warstw konwolucyjnych, więc dla tak prostych danych zwiększenie modelu jest nieefektywne.

Co ciekawe, w przypadku wytrenowanego już modelu do symulatora zredukowanie rozdzielczości obrazów dziesięciokrotnie w każdym wymiarze (z rozdzielczości 320x160 do 32x16), i zwykłe przeskalowanie w górę przed zewaluowaniem wystarczy, żeby utrzymać się na torze.

Ponadto, sieć uczona na obrazie kolorowym działa bezproblemowo, gdy zredukuje się obraz do skali szarości. Jedyne, co należy wykonać to stworzyć obraz kolorowy, w którym każdy z kanałów RGB będzie powtórzonym obrazem wejściowym.

Rozdział 4.

Podsumowanie

Sieć neuronowa wykonana w ramach projektu działa na ten moment dość intuicyjnie, co znaczy, że nie wykonuje żadnego planowania, decyzje podejmuje na bieżąco. Jest w stanie pokonać prostą trasę, w której tylko jeden tor może zostać uznany za prawidłowy.

4.1. Możliwe usprawnienia

Aktualna sieć może nie być w stanie rozwiązać wszystkich problemów, w szczególności planowania trasy. Dlatego też warto rozważyć opisane poniżej podejścia.

Najprostszym rozszerzeniem obecnej architektury może być wzięcie pod uwagę czasu – np. reagowanie na k ostatnich obrazków. Dzięki temu rozwiązaniu sieć byłaby w stanie wywnioskować aktualną prędkość, ale wciąż jej decyzje pozostałyby spontaniczne.

Większym wyzwaniem byłoby wytrenowanie rekurencyjnej sieci neuronowej (RNN), zdolnej do przekazania samej sobie kontekstu. Trenowanie takiej sieci mogłoby się okazać bardziej problematyczne. Prostą strategią dla takiej sieci jest powtarzanie ostatniego wypisanego wyniku. Mogłoby to doprowadzić do sytuacji, w której łazik wykonuje wciąż te same operacje, nie reagując na zmianę warunków.

Kolejnym rozwiązaniem jest reinforced learning, które polega na wskazywaniu sieci sytuacji, w których sieć popełniła błąd. Przykładowo, za taką sytuację można uznać konieczność interwencji człowieka lub wyjechanie łazika poza trasę. Niestety, modelu wytrenowanego na symulatorze nie da się przenieść bezpośrednio na pojazd (np. ze względu na znaczną różnicę obrazu). Dlatego też omawiane rozwiązanie można trenować jedynie w rzeczywistości. Stwarza to niestety problemy – błąd prawdziwego pojazdu może być kosztowny lub niebezpieczny.

Dodatkowo, każde ze wspomnianych rozwiązań można usprawinić poprzez zmianę perspektywy kamery na ujęcie z góry.

Oczywiście pozostają też rozwiązania nieużywające sieci neuronowych. Można na przykład stworzyć program, który korzystając z mapy otoczenia lub mapy głębokości z kamery nadzorowałby trasę łazika, zmniejszając ryzyko wjechania pojazdu w przeszkodę.

Zaproponowane powyżej rozwiązania mogą zostać zastosowane w przyszłości, w ramach dalszego rozwoju projektu.

Bibliografia

[1] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, Karol Zieba End to End Learning for Self-Driving Cars, 2016

[2] Alex Krizhevsky, Ilya Sutskever,Geoffrey E. Hinton ImageNet Classification with Deep Convolutional Neural Networks, 2012

[3] Mukund Sundararajan, Ankur Taly, Qiqi Yan Axiomatic Attribution for Deep Networks, 2017