Weryfikacja możliwości sterowania łazikiem za pomocą sieci neuronowych

(TODO)

Paweł Dybiec

Praca licencjacka

Promotor: dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

TODO

Streszczenie

Sieci neuronowe są w stanie kierować samochodem na podstawie obrazu z kamery[2]. Tematem tej pracy implementacja i przetestowanie autonomicznej jazdy łazika Aleph 1 korzystającej z konwolucyjnych sieci neuronowych.

TODO ENG abstract

Spis treści

Rozdział 1.

Preliminaria

Ta praca została zrealizowana w ramach przedmiotu "Projekt: autonomiczna jazda łazikiem". Z jego powodu(trzeba zmienic to wyrazenie), 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 w tym rozdziale poruszone będą:

- Łazik Aleph 1
- Podstawy sieci neuronowych.
- Architektura ROS

1.1. Łazik

Łazik Aleph 1 powstał z inicjatywy koła naukowego 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 ocenianie w czterech kategoriach[2]:

- Science Cache Task – pobieranie i badanie próbek w terenie
- Extreme Retrieval and Delivery Task – transport pakunku w różnych warunkach terenowych

[1] Strona koła naukowego Continuum: http://continuum.uni.wroc.pl/
[2] Regulamin konkursu URC: http://urc.marssociety.org/home/requirements-guidelines

- 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, przykładowo dla regresji 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 wystarczająco dobre.

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 1 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 piksli 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 soie z widzeniem maszynowym, są w stanie klasyfikować bezproblemowo obrazki[3]. 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. ROS

ROS 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 dostarczane 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 ROSie jest wierzchołek(node), jego głównym zdaniem jest wykonywanie obliczeń. Wierzchołki razem tworzą graf, a komunikują się za pomocą tematów(topic).

[3] https://www.cs.toronto.edu/~kriz/imagenet_classification_with_deep_convolutional.pdf

Taka architektura (inspirowana budową mikrojądra) zapewnia lepsza ochronę na błędy w porównaniu do architektury monolitycznej. Dodatkowo pojedyńczy element można bezproblemowo przepisać, i to w innym języku.

1.3.2. Topic

Tematy(topic) pozwalają bezproblemowo zapewnić komunikację międzyprocesową w ROSie. Każdy node może zadelkarować chęć nadawania bądź nasłuchiwania na danym temacie. Przykładowo moduł jazdy autonomicznej może zasubskrybować obraz z kamery Kinect, a 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 jedno (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 z pomocą innych usług służących do wizualizacji różnych typów danych takich jak obraz, wartości zmieniające się w czasie, chmury punktów lub mapa terenu.

Rozdział 2.

Trenowanie sieci i zbieranie danych

W celu autonomicznej jazdy wytrenowałem konwolucyjną sieć neuronową (CNN) 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, żeby odrzucić modele, które nie radzą sobie w tak prostych warunkach. Dodatkowo zbieranie danych oraz testowanie modelu jest łatwiejsze, ponieważ nie wymaga przygotowywania sprzętu, oraz opuszczenie toru przez model jest nieszkodliwe w porównaniu do opuszczenia drogi przez fizycznego łazika.

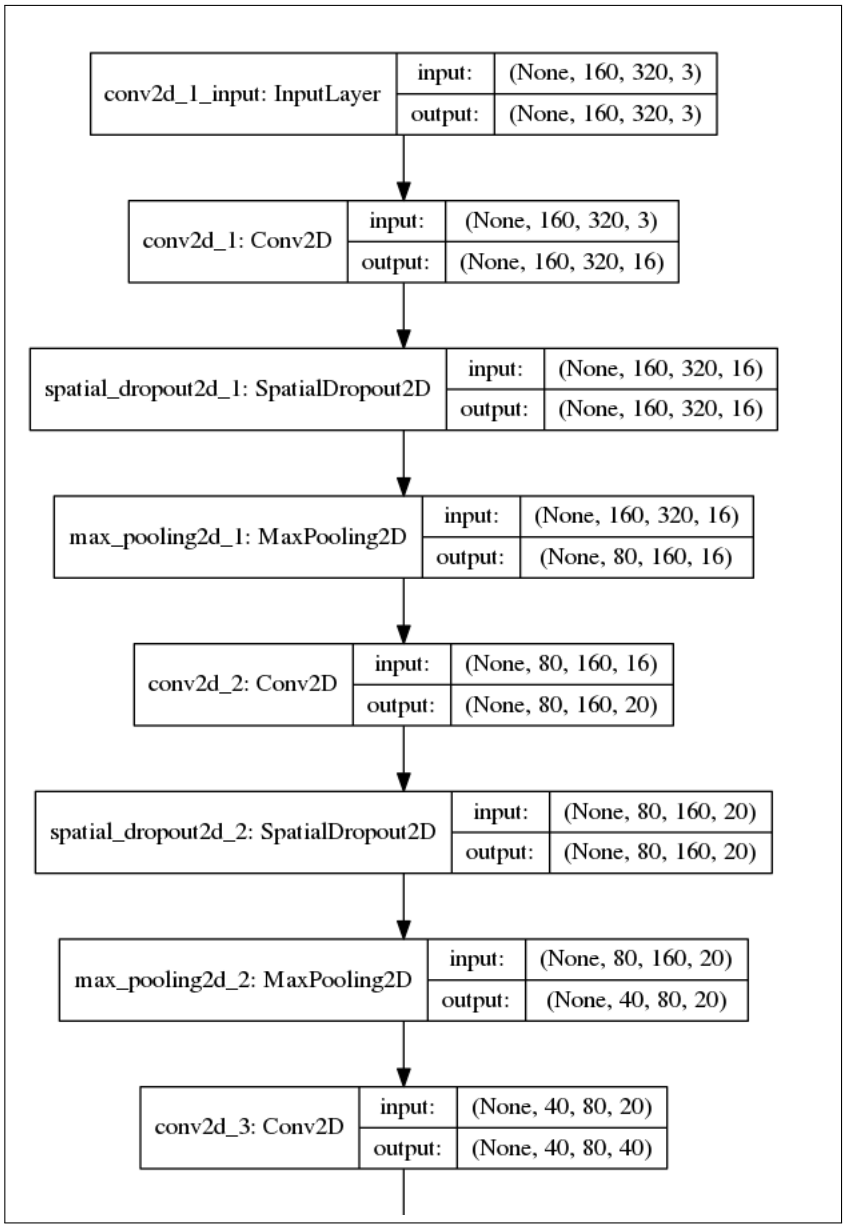
Architektura sieci pochodzi z rozwiązania chauffeur w konkursie udacity self driving car[1], ale została zaadaptowana do interfejsu symulatora oraz łazika (oprócz obrotu potrafi też zadać prędkość). Konwersja z rosbaga (format nagrań ROSa) do naszego formatu oraz sam symulator zostały wykonane przez 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 3 kamer, jedna skierowana na wprost, a pozostałe były obrócone o 20 stopni względem środkowej. Podczas uczenia wykorzystywany był obraz ze wszystkich trzech kamer. Dla kamery środkowej porządanym wynikiem

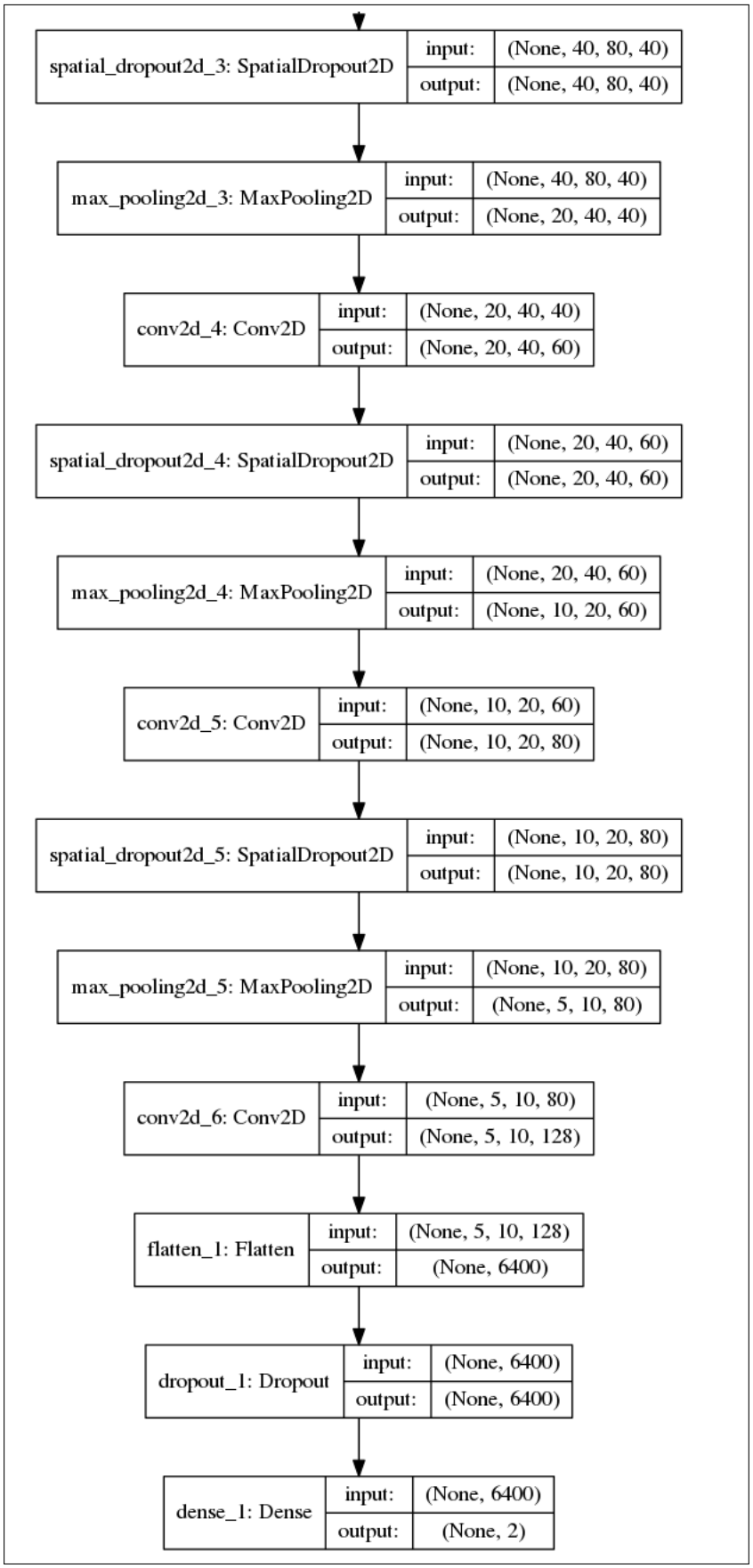
[1] Repozytorium dostępne podhttps://github.com/udacity/self-driving-car/tree/master/steering-models/community-models



były nagrane dane z sterowania, natomiast dla kamery lewej wynik był nieznacznie zaburzony w prawo, i analogicznie dla trzeciej kamery. 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, tylko jeździć przeciwnie do ruchu wskazówek zegara, a po ustawieniu modelu w przeciwnym kierunku potrafi przejechać cały tor bezproblemowo.

3.1. Na co zwraca uwagę

Aktywność sieci dla obrazków została wygenerowana za pomocą metody Integrated Gradients[1].

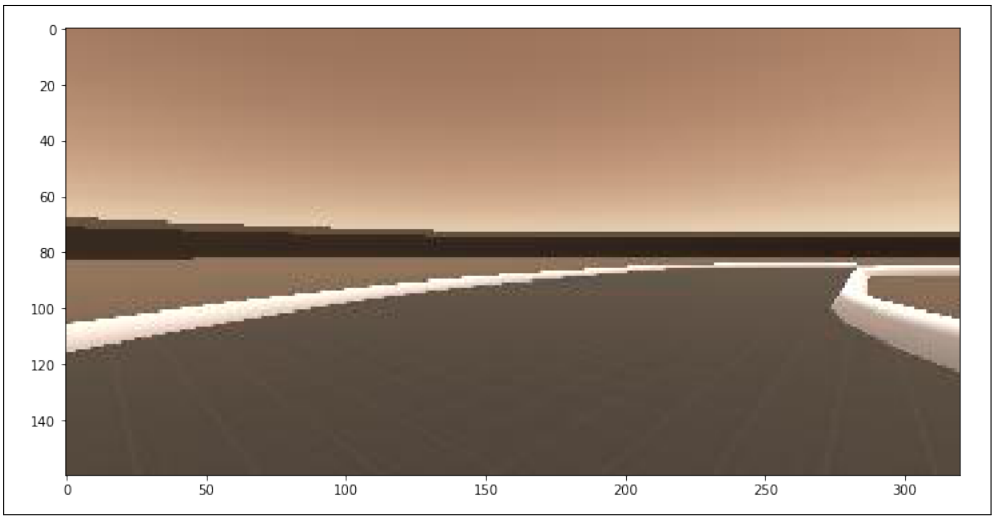
Co było oczywiste w przypadku symulatora,sieć zwraca głównie uwagę na miejsca, gdzie pojawiają się granice drogi3.1.. Co ciekawe reaguje też na ścianę tworzącą horyzont, ponieważ zmienia wygląd w zależności od odległości i może pomóc w orientacji (na tej trasie).

Z kolei dla łazika intensywność w najbardziej aktywnym miejscu jest dużo mniejsza, co oznacza że nie sugeruje się tylko jednym miejscem. Ale mimo tego najbardziej zwraca uwagę na kratkę na podłodze, która mogła by wystarczyć do nawigacji.

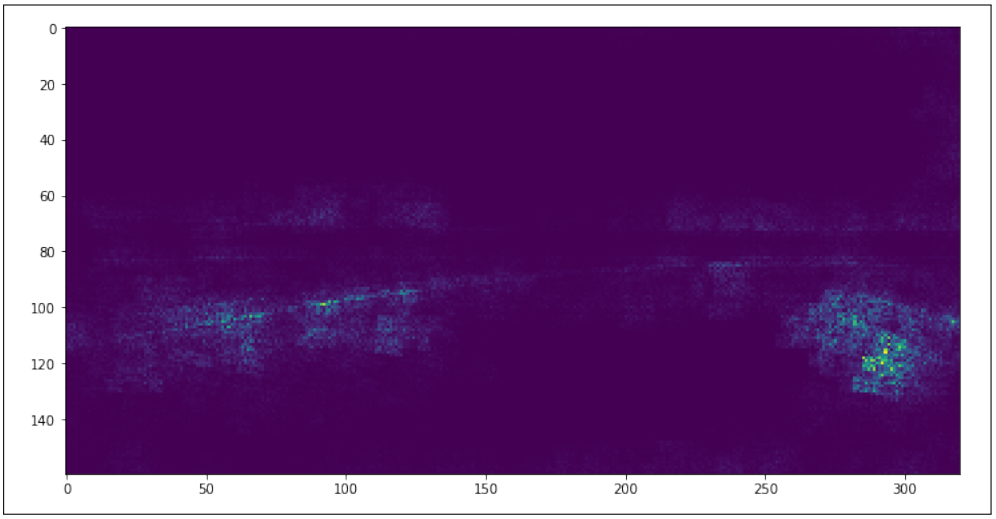
3.2. W porównaniu do nagrania

Na wykresie 3.2. widać że sieć (pomarańczowy kolor), mniej gwałtownie zmienia szybkość obrotu niż kierowca (kolor niebieski). Ale w podobnych momentach zauważa, że należy skręcić.

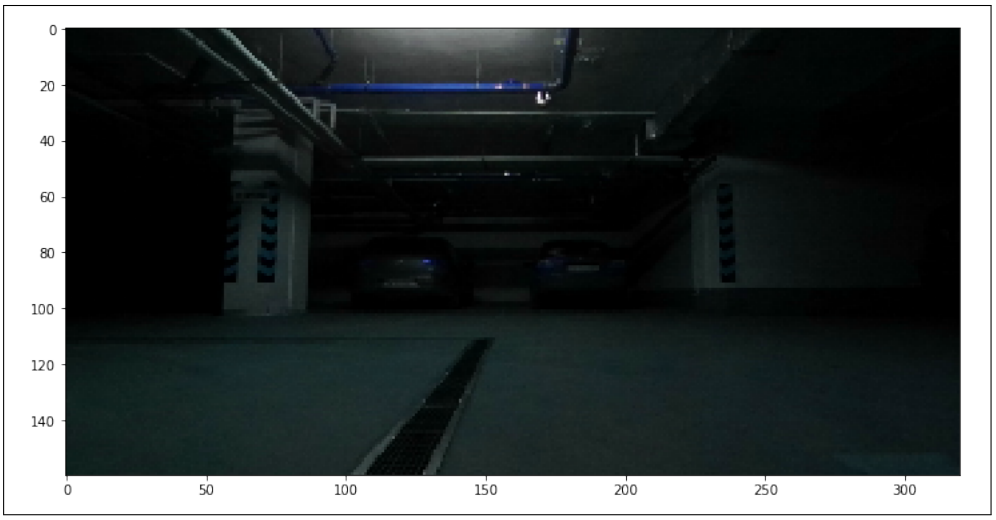
[1] https://arxiv.org/abs/1703.01365



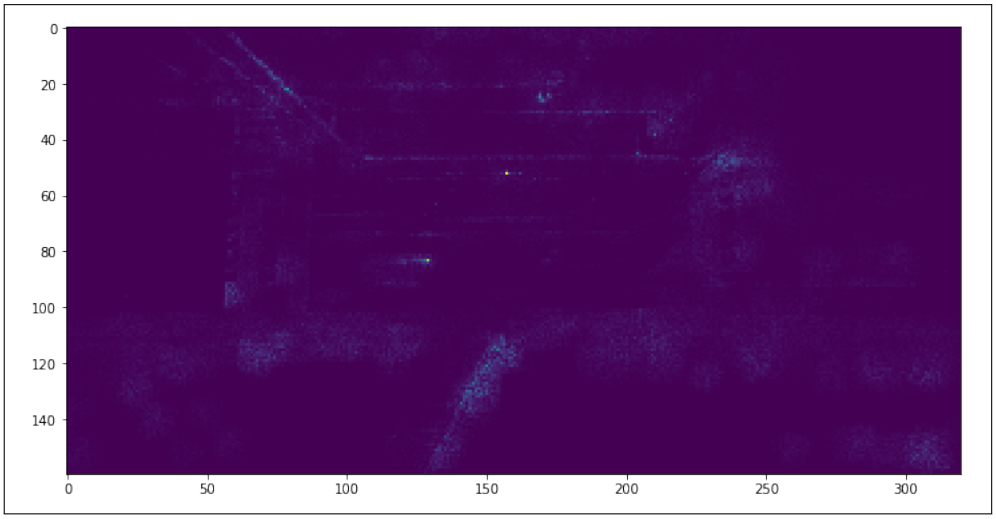
Rysunek 3.1: Obraz z symulatora



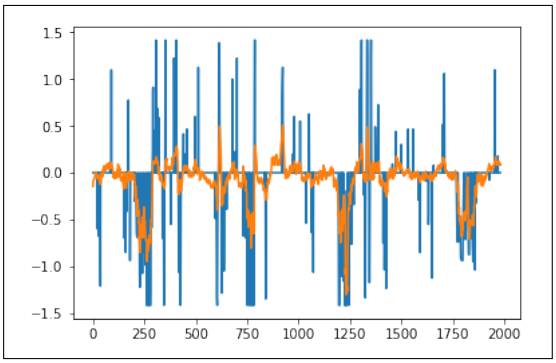
Rysunek 3.2: Na co sieć patrzy, symulator



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 utrzymać się na torze, a taka sama zredukowana architektura nie radziła sobie dobrze w przypadku nagrań z prawdziwego łazika. Natomiast usunięcie nieliniowości z warstw konwolucyjnych tak okaleczyła zdolności sieci, że nie potrafiła się utrzymać na wirtualnym torze.

Z kolei usunięcie dropoutu, bardzo szybko powodowało overfitting i 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. Widocznie większość interesujących cech już jest znaleziona w ramach konwolucji, i dla tak prostych danych nie pomaga zwiększenie modelu.

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 urzymać się na torze.

Na dodatek sieć uczona na obrazie kolorowym bezproblemowo działa, gdy zredukuje się obraz do skali szarości a następnie powtórzy kanał trzykrotnie.

Rozdział 4.

Co dalej

Najprostszym następnym krokiem jest zwiększenie danych o dodatkowy wymiar, i nauczenie takiej sieci decyzji na podstawie k (niekoniecznie) ostatnich zdjęć. Innym prostym rozwiązaniem, które można z tym połączyć jest zmiana perspektywy kamery na zdjęcie z góry.

Bardziej ambitnym pomysłem jest wytrenowanie rekurencyjnej sieci neuronowej (RNN), gdyby ją dobrze nauczyć sama wyciągnie kontekst. Ale problemem przy jej trenowaniu będzie fakt, że prostą strategią dla takiej sieci jest powtarzanie ostatniego wypisanego wyniku, a to dlatego że prędkość jest ciągła.

Kolejnym rozwiązaniem jest reinforced learning, sieć karało by się za każdą interwencję lub wyjechanie poza trasę. Niestety problemem tutaj jest fakt, że jak błąd prawdziwego pojazdu może być kosztowny lub niebezpieczny.

Oczywiście pozostają też rozwiązania nie używające sieci neuronowych, można przykładowo stworzyć program pilnujący aby łazik nie wjechał w przeszkodę.

Bibliografia

[1] autor Tytuł, 2018

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