POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Komputerowe sieci sterowania (ARK)

PRACA DYPLOMOWA INŻYNIERSKA

Moduł wykrywania przeszkód dla systemu sterowania pojazdem autonomicznym

A module for obstacle detection for a steering system in a selfdriving car

AUTOR:
Robert Jan Czwartosz

PROWADZĄCY PRACĘ:
dr Marek Bazan

KONSULTANT:
prof. dr hab. inż. Ewa Skubalska-Rafajłowicz

OCENA PRACY:

WROCŁAW 2018

Spis treści

Rozdział 1

Wstęp

Obecnie automatyzacja obejmuje wiele dziedzin, w tym również motoryzację. Zautomatyzowane samochody potrafią między innymi: jeździć po autostradzie jak i w korku, utrzymywać bezpieczną odległość od pojazdu, zmieniać pas ruchu. Każde z wymienionych zadań pojazdu autonomicznego wymaga umiejętności wykrywania przeszkód.

W pracy podjęto się napisania oprogramowania wykrywającego przeszkody i testowania go na danych z Udacity Challenge (link do repozytorium: `https://github.com/udacity/didi-competition`). Celem Ucacity Challenge jest stworzenie oprogramowania wykrywającego przeszkody na drodze. Organizatorzy udostępnili dane, na których można uczyć i testować sieć neuronową. Kryterium oceny zawodników jest wartość wskaźnika IoU - Intersection over Union(definicja jest wyjaśniona w następnym rozdziale). W repozytorium Udacity Challenge można znaleźć instrukcję (`https://github.com/udacity/didi-competition/blob/master/docs/GettingStarted.md`) wprowadzającą w zagadnienia związane z danymi i ich wizualizacją oraz podane są reguły konkursu.

W rozdziale 2. określono cel pracy i opisano na czym polega wykrywanie przeszkód. Rozdział 3. zawiera teorię dotyczącą sieci neuronowych. W rozdziale 4. opisana została metoda pracy nad problemem wykrywania przeszkód. W rozdziałach 5 i 6 opisane zostały odpowiednio metoda przetwarzania danych z Udacity Challenge oraz konfiguracja rozwiązania. Rozdział 7. zawiera wyniki. W rozdziale 8. opisane zostały możliwości rozwoju oprogramowania wykrywającego przeszkody. W rozdziale 9. znajdują się podsumowanie i wnioski wyciągnięte z pracy nad problemem wykrywania przeszkód. Dodatki A, B i C zawierają instrukcje instalacji oprogramowania, którego użyto w pracy.

Rozdział 2

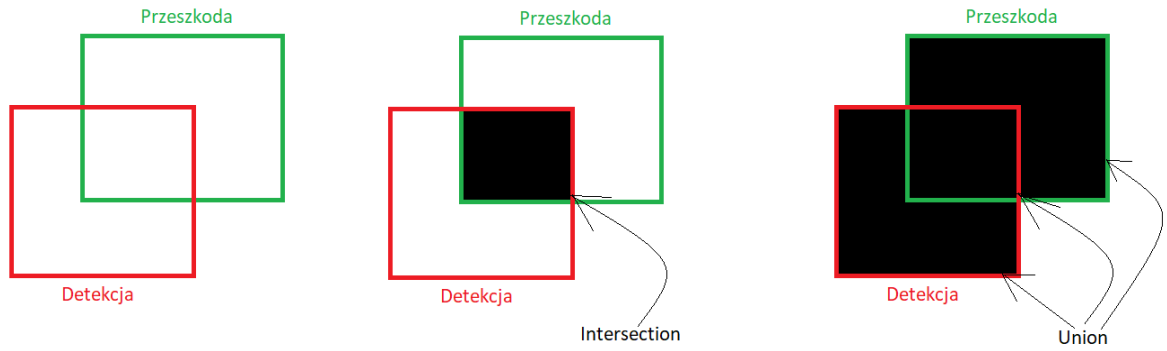
Cel i założenia pracy

Celem pracy było napisanie programu wykrywającego lokalizację przeszkód i ich wymiarów na podstawie skanu 3D z Lidaru. Dla uproszczenia przeszkody były przedstawiane jako prostopadłościany oraz ich orientacja nie była brana pod uwagę. Zatem każda przeszkoda była określona przez 6 liczb: współrzędne położenia środka (3 liczby), wysokość, szerokość i długość. Program spełniający cel pracy powinien przetwarzać pliki z danymi 3D na lokalizację i wymiary przeszkód. Rozwiązanie zostało oparte na konwolucyjnych sieciach neuronowych, których opis znajduje się w następnym rozdziale. Metoda wykrywania przeszkód polegała na tym, że sieć otrzymywała na wejściu skan z lidaru 3D, i następnie zwracała na wyjściu położenie i wymiary prostopadłościanu otaczającego przeszkodę. Miarą dokładności wyznaczania lokalizacji i wymiarów był współczynnik IoU(Intersection over Union) [22].

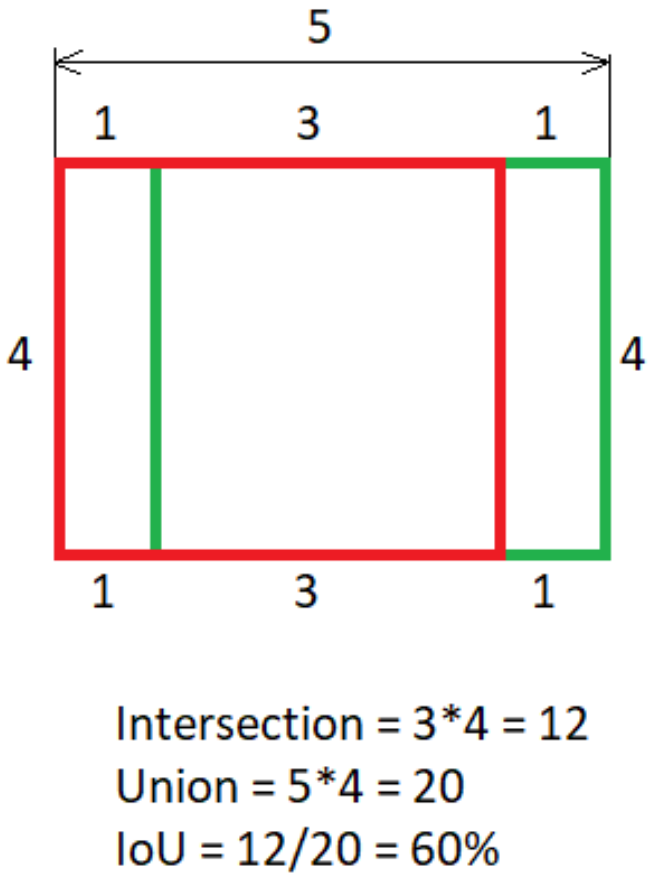
$$IoU = \frac{Intersection}{Union}$$

- *Intersection* — objętość części wspólnej obszaru przeszkody i obszaru wskazywanego przez program
- *Union* — objętość sumy obszaru przeszkody i obszaru wskazywanego przez program

Obszary były definiowane jako prostopadłościany. Natomiast suma obszarów to przestrzeń zajmowana przez co najmniej jeden z prostopadłościanów.



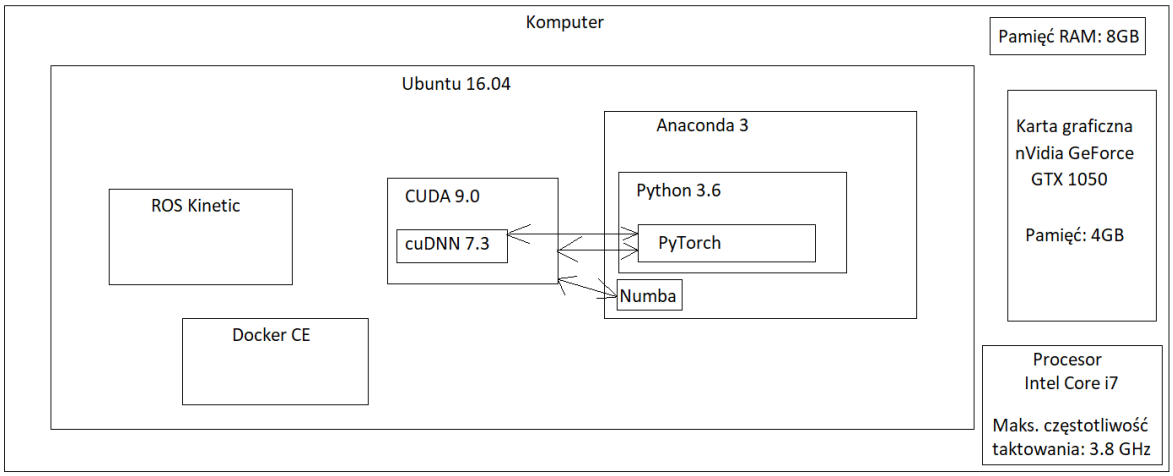
Rysunek 2.1: Zilustrowanie definicji obszarów Intersection i Union na przykładzie prostokątnych obszarów detekcji i przeszkody. W przypadku oceny detekcji obliczany jest współczynnik IoU dla prostopadłościanów. Jednak obliczenia wskaźnika dla prostopadłościanów są przeprowadzane w analogiczny sposób jak dla prostokątów.



Rysunek 2.2: Przykład obliczeń wskaźnika IoU.

Podczas sprawdzania efektywności programu wykrywającego przeszkody, dla każdej detekcji obliczana była wartość IoU. Ostatecznie obliczana była średnia arytmetyczna wyznaczonych wartości IoU. Obliczona średnia stanowiła miarę efektywności programu wykrywającego przeszkody.

W tej pracy został użyty system Ubuntu 16.04 ([19]). Na tym systemie został zainstalowany ROS (*ROS - Robotic Operating System*), a konkretnie ROS Kinetic ([17]). Dodatkowo w pracy zostało użyte oprogramowanie Docker CE [4]. Oprogramowanie ROS Kinetic wraz z oprogramowaniem Docker CE użyto do rozpakowania plików zawierających dane.



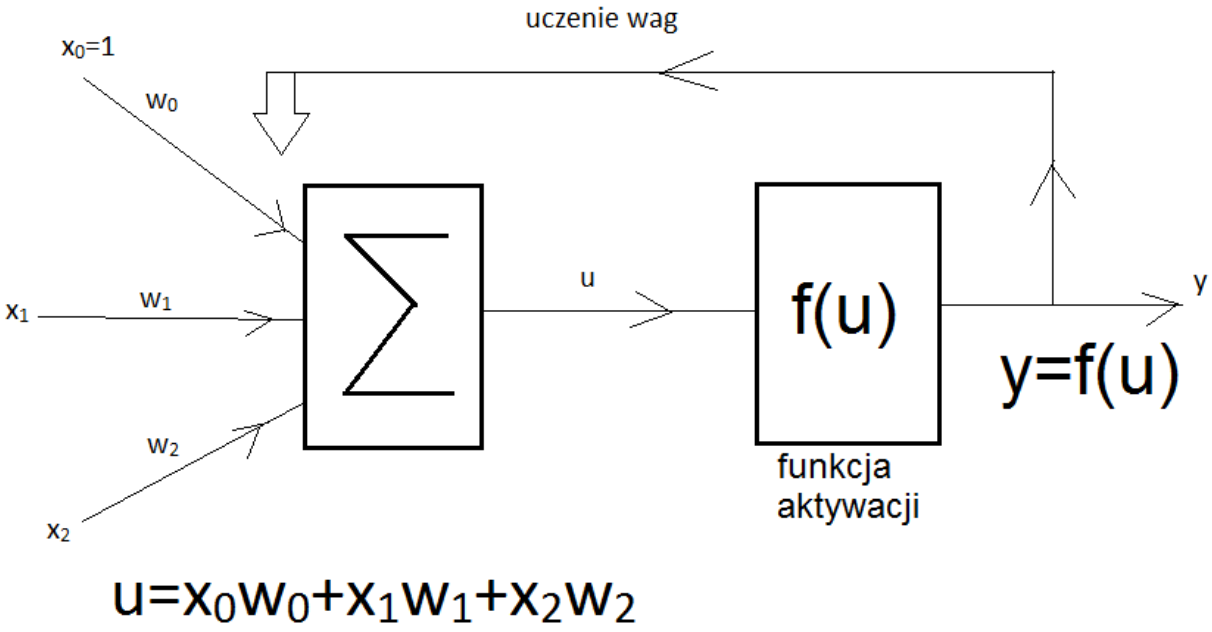
Rysunek 2.3: Narzędzia wykorzystane do tworzenia oprogramowania([2], [3], [4], [19], [17], [1]) wykrywającego przeszkody.

Przy stosowaniu konwolucyjnych sieci neuronowych, wykorzystywana była karta graficzna GTX 1050 razem z oprogramowaniem CUDA Toolkit 9.0 (*CUDA - Compute Unified Device Architecture*) i cuDNN (*cuDNN - CUDA Deep Neural Network*) 7.3.0.

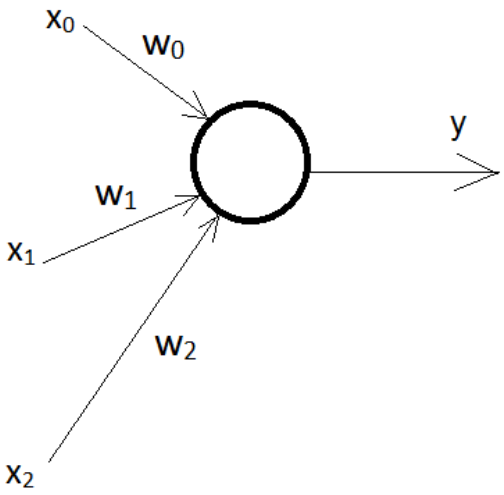
Rozdział 3

Teoria dotycząca sieci neuronowych

Rozwiązywanie problemu za pomocą sieci neuronowych polega na przybliżeniu pewnego procesu złożoną funkcją [10]. Najprostszą siecią neuronową jest perceptron. Perceptron zwraca na wyjściu wartość funkcji aktywacji ze średniej ważonej wejść.

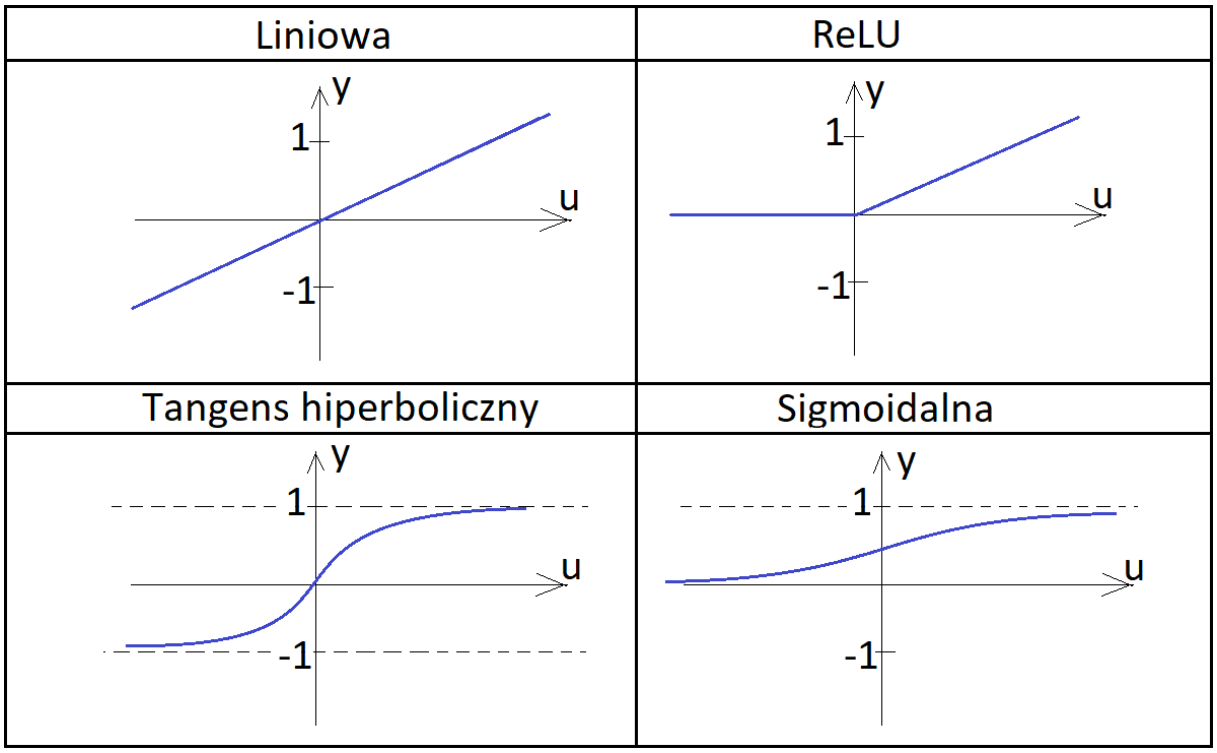


Rysunek 3.1: Działanie perceptronu.



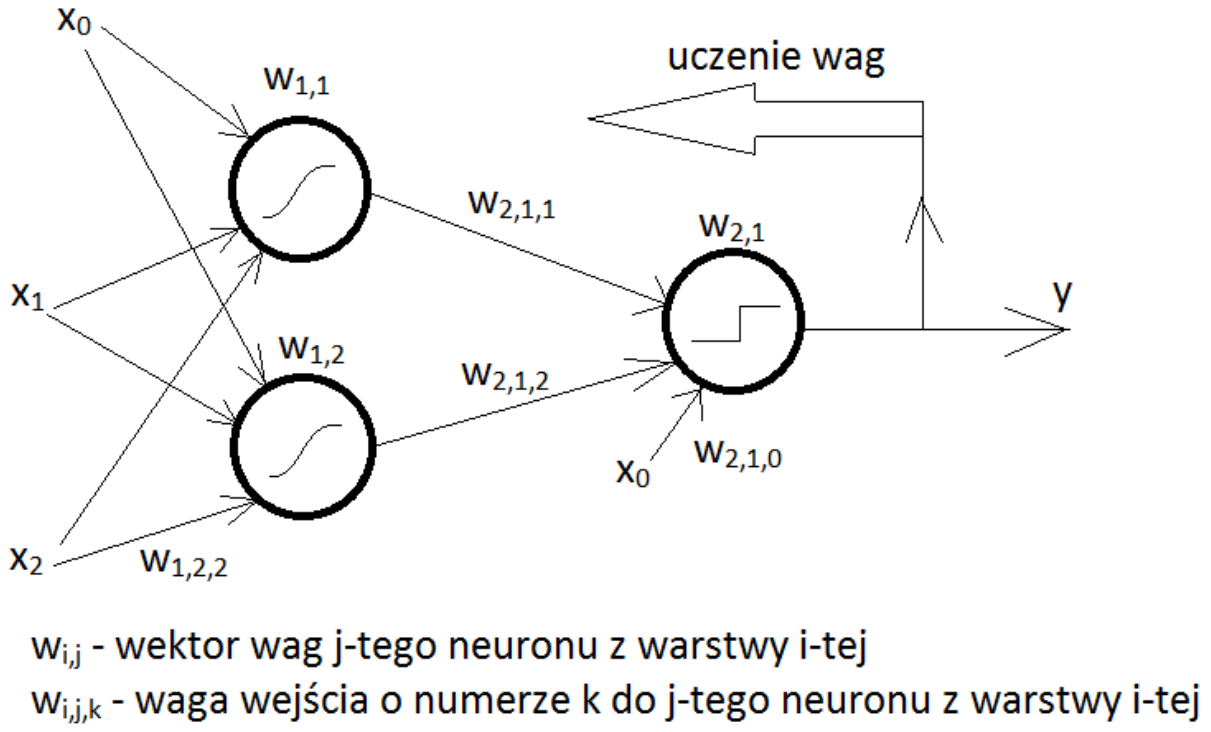
Rysunek 3.2: Uproszczony sposób przedstawienia perceptronu.

Podstawowymi funkcjami aktywacji są: funkcja liniowa, funkcja prostująca (ReLU - Rectified Linear Unit), tangens hiperboliczny oraz sigmoida.

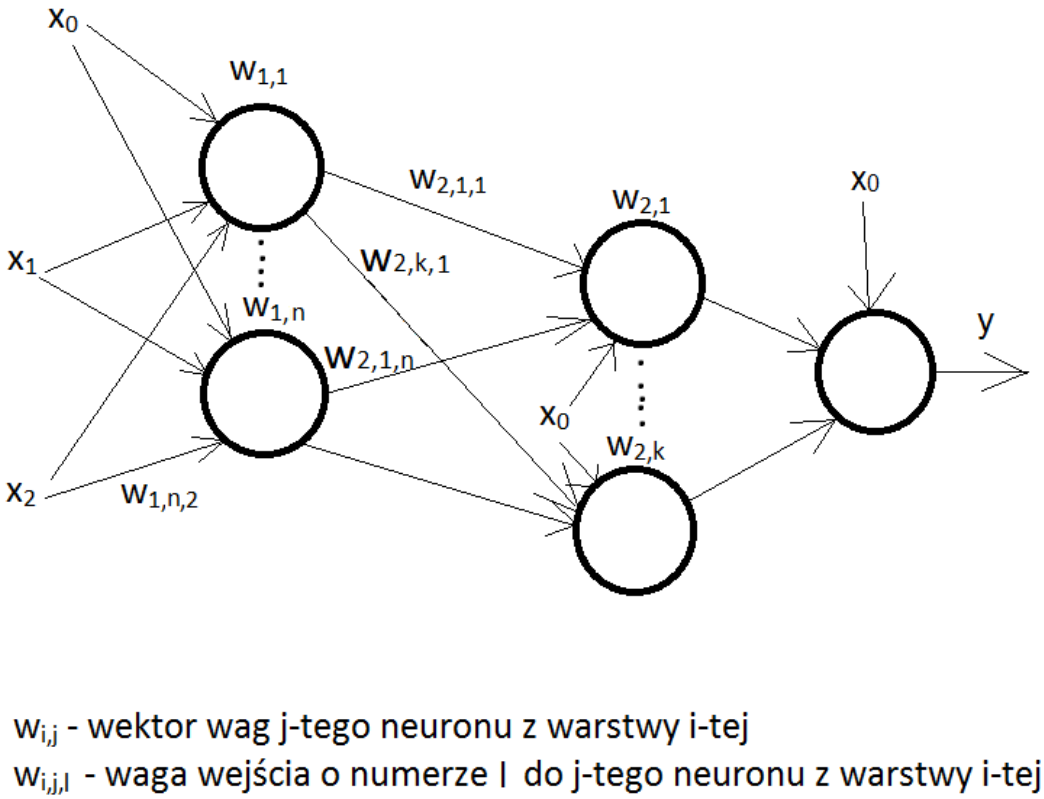


Rysunek 3.3: Wykresy podstawowych funkcji aktywacji.

Z grupy perceptronów o takiej samej funkcji aktywacji można utworzyć warstwę. Natomiast z grupy warstw można utworzyć sieci wielowarstwowe, nazywane perceptronem wielowarstwowym.



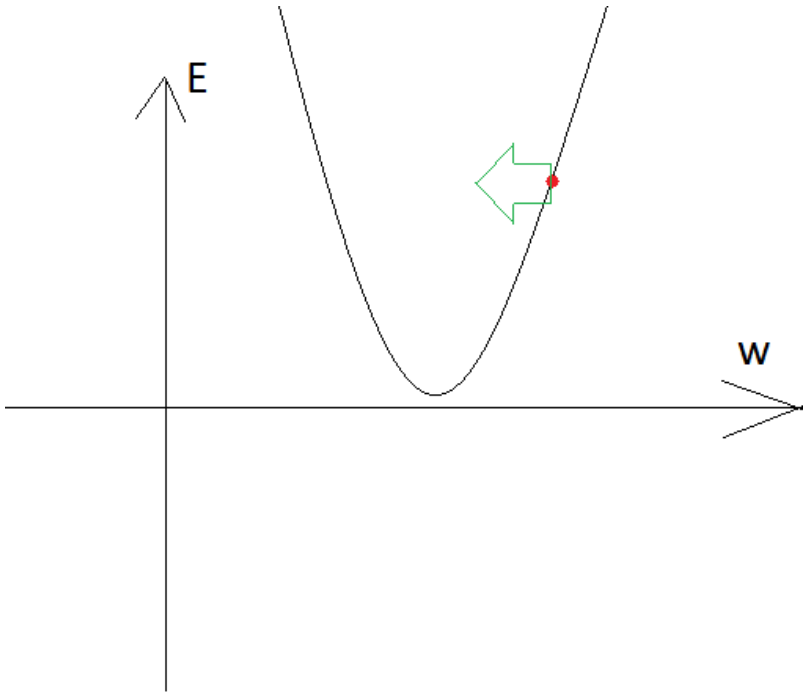
Rysunek 3.4: Perceptron dwuwarstwowy z dwoma neuronami w warstwie wejściowej oraz jednym neuronem w warstwie wyjściowej.



Rysunek 3.5: Perceptron wielowarstwowy z jedną warstwą ukrytą.

Uczenie sieci neuronowej polega na dobraniu wag, tak aby funkcja strat była jak naj-

mniejsza [10]. Metody uczenia sieci dzielą się na gradientowe (np.: metoda najszybszego spadku, metoda najszybszego spadku z momentum, metoda gradientów sprzężonych) i bezgradientowe (np.: symulowane wyżarzanie, algorytmy genetyczne). W uczeniu sieci neuronowych stosuje się metody gradientowe, gdyż istnieje możliwość obliczenia gradientu funkcji strat. Najprostszą z metod gradientowych jest metoda najszybszego spadku. Metoda najszybszego spadku polega na modyfikowaniu wag w kierunku przeciwnym do kierunku gradientu funkcji strat. Wagi mogą być modyfikowane po każdym przykładzie lub po przetworzeniu całego zbioru uczącego i obliczeniu średniej wartości poprawki wag. Jednak najczęściej się stosuje metodę modyfikacji wag, polegającą na modyfikacji wag po przetworzeniu pewnej liczby przykładów (nazywanej rozmiarem kroku) i obliczeniu średniej wartości poprawki wag.

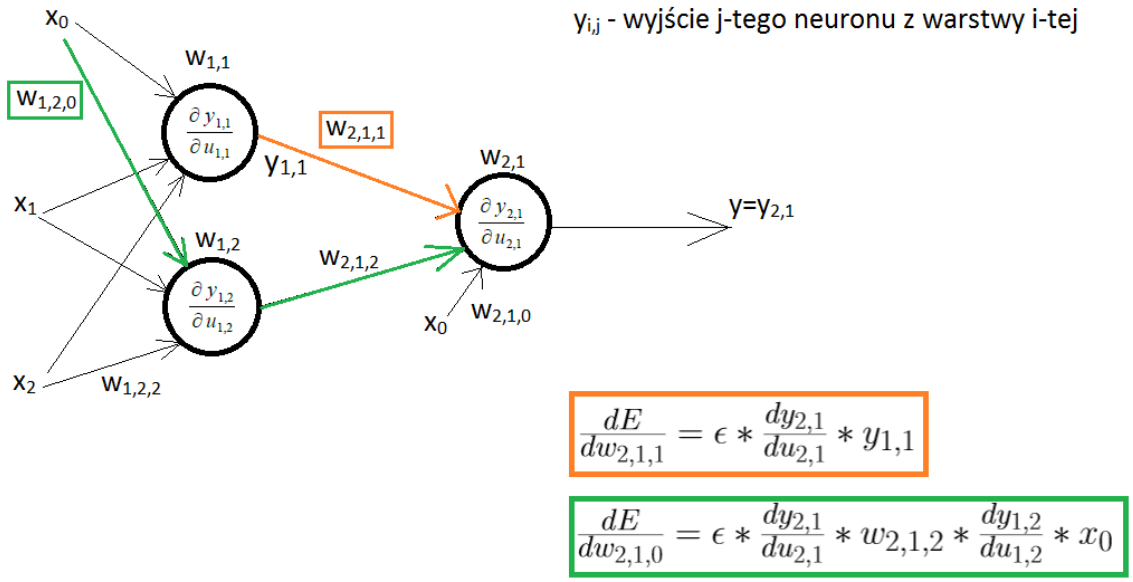


Rysunek 3.6: Zobrazowanie metody największego spadku na przykładzie uczenia perceptronu z jednym wejściem oraz funkcją strat będącą błędem średniokwadratowym.

Jednak w oprogramowaniu wykrywającym przeszkody, do nauki sieci użyto optymalizatora ADAM(Adaptive Moment Estimation - adaptacyjna estymacja momentu) [12]. Parametrami algorytmu ADAM są: α - współczynnik uczenia, β_1, β_2, oraz ϵ. Celem algorytmu jest znalezienie wektora parametrów θ, takiego aby funkcja $f(\theta)$ przyjmowała jak najmniejszą wartość. Algorytm zaczyna się od inicjalizacji:
$m_0 \leftarrow 0$ - inicjalizacja wektora pierwszego momentu
$v_0 \leftarrow 0$ - inicjalizacja wektora drugiego momentu
$t \leftarrow 0$ - inicjalizacja kroku.
Następujące operacje są ciągle powtarzane:
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t\left(\theta_{t-1}\right)$ - obliczenie gradientu
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ - obliczenie obciążonej wartości estymatora pierwszego momentu
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ - obliczenie obciążonej wartości estymatora drugiego momentu
$\widehat{m}_t \leftarrow m_t / \left(1 - \beta_1^t\right)$ - obliczenie poprawionej wartości estymatora pierwszego momentu
$\widehat{v}_t \leftarrow v_t / \left(1 - \beta_2^t\right)$ - obliczenie poprawionej wartości estymatora drugiego momentu

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$ - aktualizacja parametrów. Czynności są powtarzane do momentu osiągnięcia zbieżności wektora parametrów.

Do obliczenia pochodnej funkcji strat po wadze wejścia do warstwy ukrytej, stosuje się metodę propagacji wstecznej błędu [10].



$$\frac{dE}{dw_{2,1,1}} = \epsilon * \frac{dy_{2,1}}{du_{2,1}} * y_{1,1}$$

$$\frac{dE}{dw_{2,1,0}} = \epsilon * \frac{dy_{2,1}}{du_{2,1}} * w_{2,1,2} * \frac{dy_{1,2}}{du_{1,2}} * x_0$$

gdzie ϵ jest wartością błędu oraz $\frac{dy}{du}$ jest wartością pochodnej funkcji aktywacji.

Rysunek 3.7: Obliczanie pochodnej funkcji strat (będącej sumą kwadratów błędów) metodą propagacji wstecznej.

Bardzo duże (np.: rzędu tysięcy) lub bardzo małe (np.: rzędu części tysięcznych) liczby na wejściu sieci spowalniają trening sieci. W celu przyspieszenia treningu sieci neuronowej stosowana jest normalizacja danych wejściowych [9]. Operacja normalizacji polega na odjęciu od liczby na wejściu wartości średniej (z wartości na danym wejściu z przykładów zawartych w danym kroku) i podzieleniu wyniku przez odchylenie standardowe. Jednak znormalizowane wartości na wejściach sieci nie gwarantują znormalizowanych wartości na wejściach warstwy ukrytej. W celu zapewnienia znormalizowanych wartości na wejściach warstwy ukrytej stosowana jest warstwa normalizująca. Warstwa normalizująca najpierw wykonuje operację normalizacji, a następnie wynik operacji normalizacji jest przeskalowywany przez współczynnik γ i przesuwany o współczynnik β. Współczynniki γ i β są wagami, które podlegają uczeniu. Dla przykładu, w trzech kolejnych przykładach otrzymano na wejściu do warstwy ukrytej wartości: 2, 5 i 11. Współczynniki γ i β wynoszą odpowiednio 3 i 1. Dla przykładu pierwszego obliczenie znormalizowanej wartości wejścia do warstwy ukrytej przebiega następująco:

$B = \{x_1 = 2, x_2 = 5, x_3 = 11\}$

$\gamma = 3, \beta = 1$

$\mu_B = \frac{x_1+x_2+x_3}{3} = \frac{2+5+11}{3} = 6$

$\sigma_B^2 = \frac{(2-6)^2+(5-6)^2+(11-6)^2}{3} = \frac{16+1+25}{3} = 14$

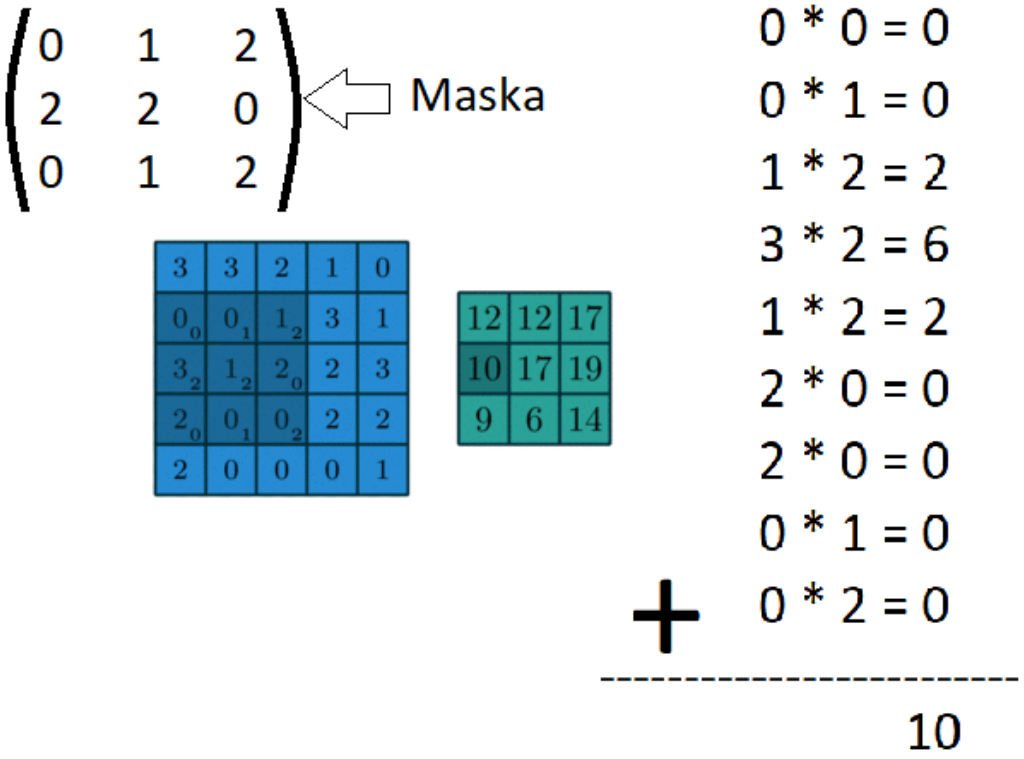
$\sigma_B = \sqrt{\sigma_B^2} = \sqrt{14} \approx 4$

$\hat{X}_1 = \frac{x_1-\mu_B}{\sigma_B} \approx \frac{2-6}{4} \approx -1$

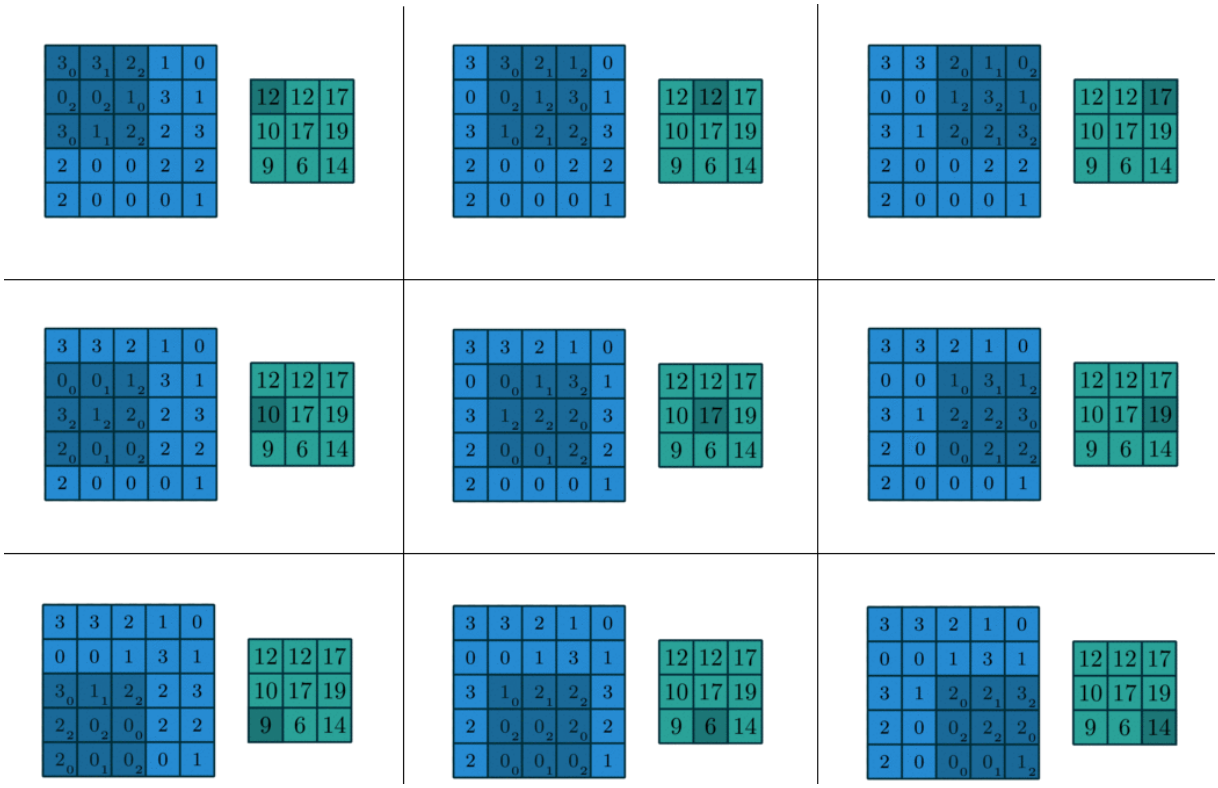
$\hat{Y}_1 = \gamma * \hat{X}_1 + \beta = 3 * (-1) + 1 = -3 + 1 = -2$

Do przetwarzania obrazów stosowane są sieci konwolucyjne [11]. Podstawowym poję-

ciem jest operacja konwolucji. Operacja konwolucji jest wykonywana poprzez nałożenie maski (jądra) na macierz wejściową w lewym górnym rogu, następnie pomnożenie elementów maski z odpowiednimi elementami macierzy wejściowej, dodanie wszystkich wyników mnożenia i zapisanie wyniku dodawania do macierzy wyjściowej. Maska jest przesuwana wierszami od lewej do prawej i obliczane są kolejne elementy macierzy wyjściowej. Parametrami, które podlegają treningowi są wartości elementów maski.

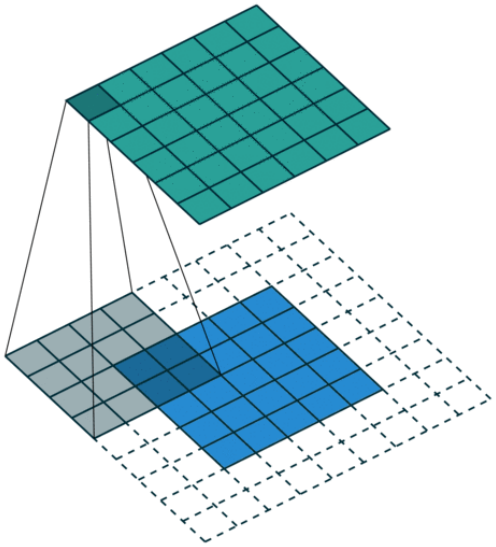


Rysunek 3.8: Sposób obliczania wartości elementu macierzy wyjściowej [5].



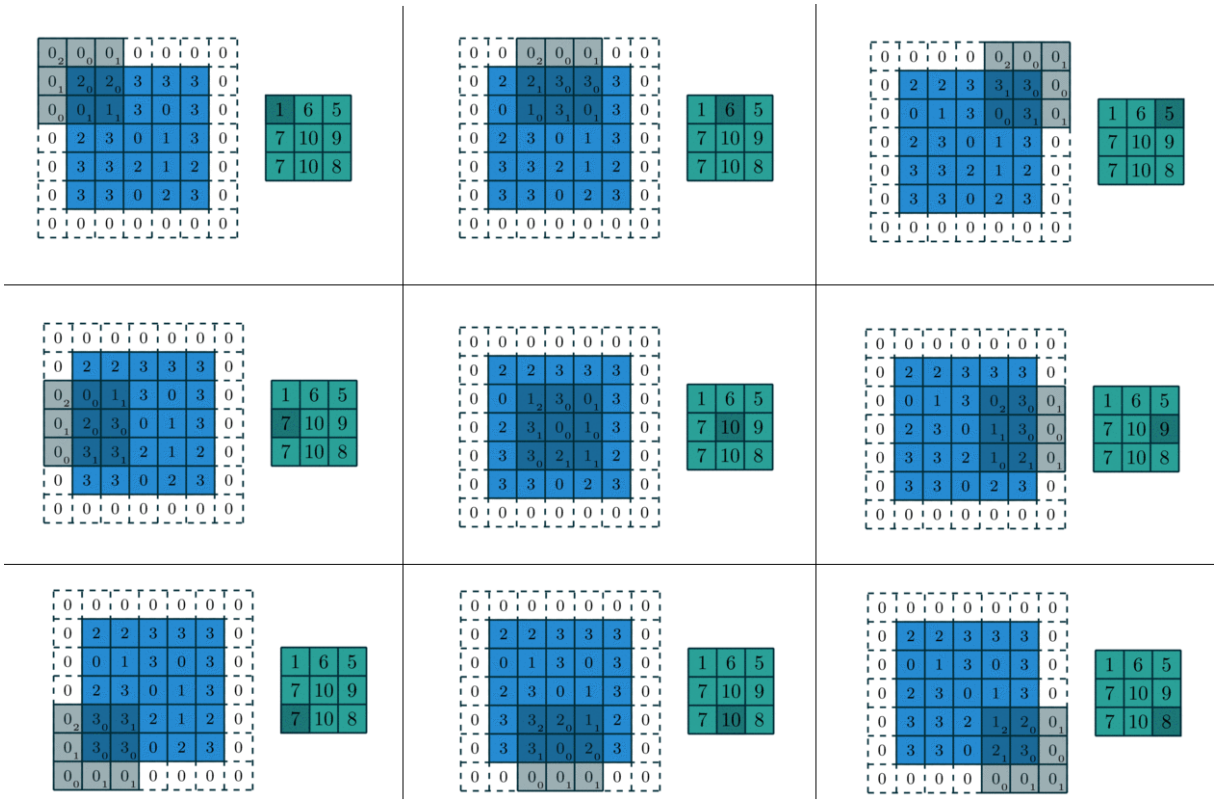
Rysunek 3.9: Operacja konwolucji macierzy 5 x 5 z maską, przedstawioną na poprzednim rysunku [5].

W sieciach konwolucyjnych może być stosowane wypełnienie zerami przestrzeni poza macierzą wejściową.



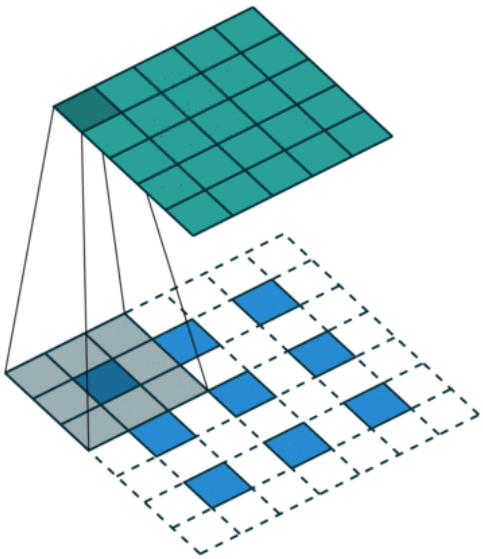
Rysunek 3.10: Wypełnienie zerami przestrzeni poza macierzą wejściową [5].

Również może być stosowane zwiększenie ilości pól, o które przesuwana jest maska.



Rysunek 3.11: Maska porusza się co drugie pole w prawo oraz co drugi wiersz [5].

Aby spowolnić ruch maski, można wypełnić zerami obszar pomiędzy polami macierzy wejściowej.



Rysunek 3.12: Poprzez zastosowanie wypełnienia zerami obszaru pomiędzy polami, ruch maski jest spowolniony dwukrotnie [5].

Rozdział 4

Metoda rozwiązywania problemu

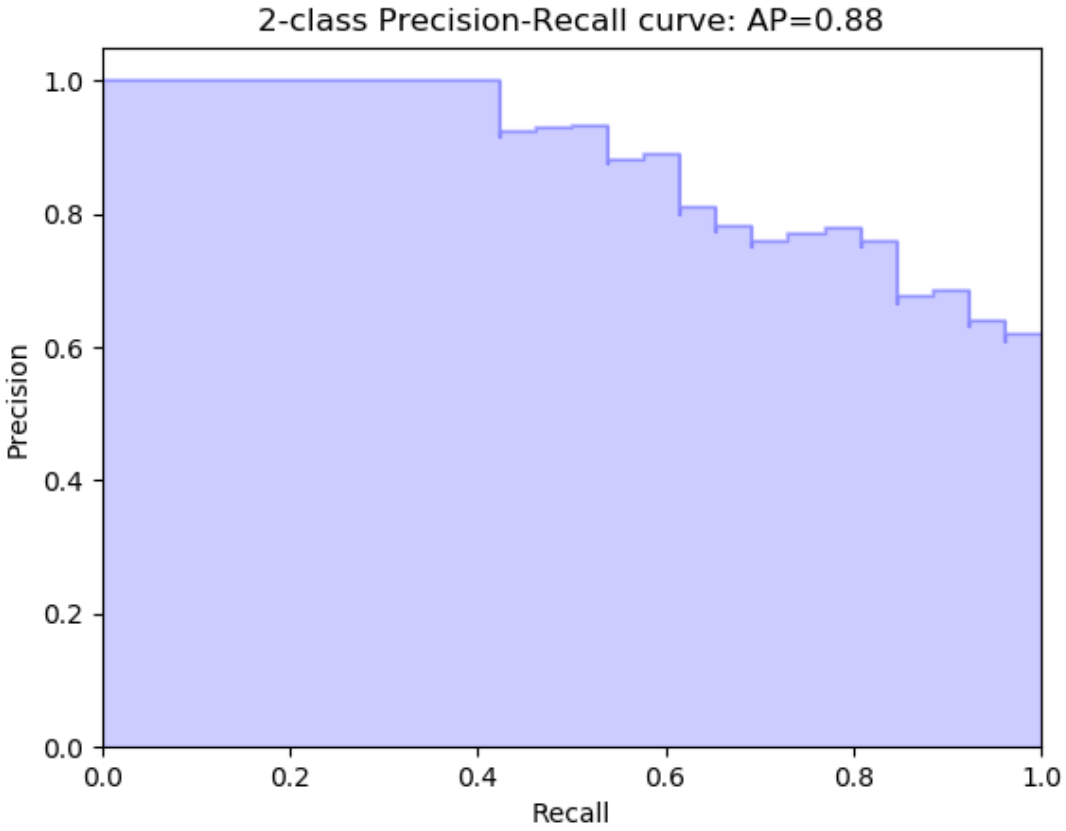
Pierwszym etapem było znalezienie wszystkich rozwiązań opensource, które określały lokalizację i wymiary przeszkód na podstawie skanu 3D z lidaru. Po znalezieniu rozwiązań uporządkowano je od najlepszego do najgorszego według wybranego kryterium. Jeśli najlepszego rozwiązania nie można było uruchomić, było ono odrzucane. Po odrzuceniu rozwiązania, wybierane było kolejne spośród uporządkowanych rozwiązań.

Kolejnym etapem było przetworzenie danych uczących z Udacity Challenge do formatu, który jest wymagany w najlepszym rozwiązaniu. Następnie uruchomiono rozwiązanie na przetworzonych danych, na przetworzonych danych. Jeżeli uruchomienie się nie powiodło to podejmowano próbę uruchomienia kolejnego rozwiązania.

Ostatnim etapem pracy było sprawdzenie dokładności wykrywania przeszkód na danych testowych Udacity. Dokładność wyznaczania przeszkód jest określana przez wskaźnik IoU.

Oprogramowanie, nad którym pracowałem, zostało stworzone na bazie kodu opensource. Jedynym znalezionym źródłem rozwiązań problemu wykrywania przeszkód z danych 3D jest ranking KITTI [6]. Ranking jest prowadzony w trzech kategoriach: wykrywanie samochodów, wykrywanie pieszych, wykrywanie rowerzystów. W każdej kategorii rozwiązania są posortowane według średniej precyzji (*Average Precision* w skrócie AP) uzyskanej na umiarkowanym (*Moderate*) poziomie trudności. Przy opisie sposobu wyznaczania AP korzystałem z artykułu [8].

Precyzja jest definiowana jako iloraz liczby poprawnie wykrytych przeszkód (*true positives*) przez liczbę wszystkich wykrytych przeszkód. Poprawnie wykryta przeszkoda jest przeszkodą, dla której IoU jest większe od pewnego progu detekcji. Próg detekcji dla kategorii wykrywania samochodów wynosi 0.7, a dla pozostałych dwóch kategorii 0.5. Przy ocenie jakości detekcji wraz z precyzją jest obliczana czułość detekcji (*Recall*). Czułość detekcji jest definiowana jako iloraz liczby poprawnie wykrytych przeszkód przez liczbę wszystkich przeszkód.



Rysunek 4.1: Przykładowa krzywa precision-recall. Źródło: `https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html`

Do obliczenia AP potrzebna jest krzywa p(r) nazywana *krzywą precision-recall*. Krzywa jest wyznaczana poprzez obliczanie precyzji i czułości dla kolejnych wykrytych przeszkód. Precyzja dla K-tej przeszkody jest obliczana jako iloraz liczby poprawnie dotychczas wykrytych przeszkód przez K. Natomiast czułość dla K-tej przeszkody jest ilorazem liczby poprawnie dotychczas wykrytych przeszkód przez liczbę wszystkich przeszkód. Działanie wzorów na precyzję i czułość przedstawiono na przykładzie z czterema przeszkodami.

Tabela 4.1: Precyzja i czułość dla kolejnych detekcji.

Nr wykrytej przeszkody	Czy poprawnie wykryto przeszkodę?	Precyzja	Czułość
1	Nie	0.00	0.00
2	Tak	0.50	0.25
3	Tak	0.67	0.50
4	Nie	0.50	0.50
5	Nie	0.40	0.50
6	Tak	0.50	0.75
7	Tak	0.4	1

Przykładowo dla przeszkody nr 6 precyzja p i czułość r są obliczane w następujący sposób

$$p = 3/6 = 0.5$$

$$r = 3/4 = 0.75$$

Po wyznaczeniu krzywej precision-recall, AP oblicza się ze wzoru

$$AP = \frac{1}{11} * \sum_{R \in \{0.0, 0.1, ..., 1.0\}} \max_{r:r \geqslant R} p(r),$$

, gdzie r oznacza czułość oraz p(r) oznacza precyzję odczytaną z krzywej precision-recall.

Wybrano rozwiązania opensource, które brały udział w każdej kategorii. Kryterium uporządkowania rozwiązań jest suma wskaźników AP w każdej kategorii. Najlepszym rozwiązaniem według kryterium sumy wskaźników AP jest F-PointNet. Tabele 5.2, 5.3, 5.4 i 5.5 przedstawiają rankingi rozwiązań osobno dla każdej kategorii oraz ranking według kryterium sumy wskaźników AP.

Tabela 4.2: Ranking rozwiązań opensource w kategorii wykrywania samochodów

lp.	Rozwiązanie	AP[%]
1	SECOND	73.66
2	AVOD-FPN	71.88
3	F-PointNet	70.39
4	AVOD	65.78

Tabela 4.3: Ranking rozwiązań opensource w kategorii wykrywania pieszych

lp.	Rozwiązanie	AP[%]
1	F-PointNet	44.89
2	AVOD-FPN	42.81
3	SECOND	42.56
4	AVOD	31.51

Tabela 4.4: Ranking rozwiązań opensource w kategorii wykrywania rowerzystów

lp.	Rozwiązanie	AP[%]
1	F-PointNet	56.77
2	SECOND	53.85
3	AVOD-FPN	52.18
4	AVOD	44.90

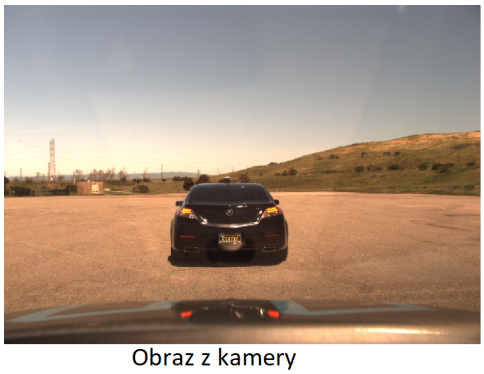
Tabela 4.5: Ranking rozwiązań opensource według kryterium sumy wskaźników z trzech kategorii

lp.	Rozwiązanie	Suma AP[%]	Artykuł
1	F-PointNet	172.05	[16]
2	SECOND	170.07	brak
3	AVOD-FPN	166.87	[14]
4	AVOD	142.19	[14]

Rozdział 5

Konwersja danych z Udacity Challenge

Dane opublikowane przez Udacity ([20]) są w postaci plików z rozszerzeniem *.bag*. W tych plikach są zapisane wiadomości wysyłane przez czujniki do komputera z oprogramowaniem ROS (*Robotic Operating System*). Dane są podzielone na zbiór danych testowych (katalog Didi-Release-2/Data/3/) i treningowych (katalogi Didi-Release-2/Data/1/ oraz Didi-Release-2/Data/2/).



Obraz z kamery

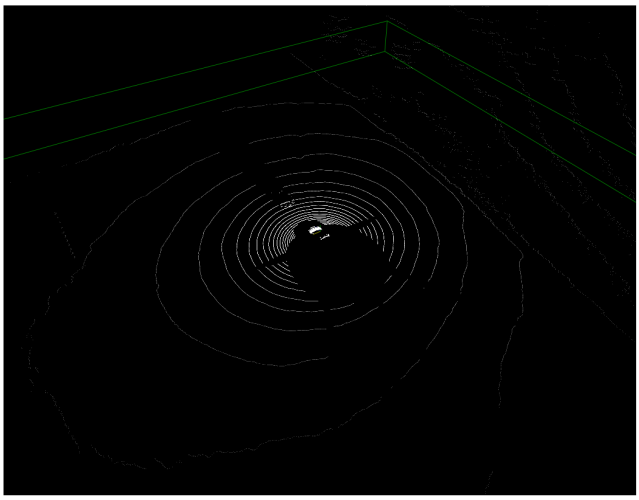
$$\mathbf{P}_{rect}^{(i)} = \begin{pmatrix} f_u^{(i)} & 0 & c_u^{(i)} & -f_u^{(i)} b_x^{(i)} \\ 0 & f_v^{(i)} & c_v^{(i)} & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\mathbf{T}_{velo}^{cam} = \begin{pmatrix} \mathbf{R}_{velo}^{cam} & \mathbf{t}_{velo}^{cam} \\ 0 & 1 \end{pmatrix}$$

$$\mathbf{y} = \mathbf{P}_{rect}^{(i)} \; \mathbf{R}_{rect}^{(0)} \; \mathbf{x}$$

```
P0: 176.5 0.000000 30 646.5 0.000000 83 84.5 0.000000 0.000000 0.000000 1.000000 0.000000
P1: 176.5 0.000000 30 646.5 0.000000 83 84.5 0.000000 0.000000 0.000000 1.000000 0.000000
P2: 176.5 0.000000 30 646.5 0.000000 83 84.5 0.000000 0.000000 0.000000 1.000000 0.000000
P3: 176.5 0.000000 30 646.5 0.000000 83 84.5 0.000000 0.000000 0.000000 1.000000 0.000000
R0_rect: 1 0 0 0 1 0 0 0 1
Tr_velo_to_cam: 1 0 0 0.4 0 1 0 0 0 0 1 0.77
Tr_imu_to_velo: 1 0 0 -0.8 0 1 0 0 0 0 1 0.8
```

Macierze kalibracji



Chmura punktów

```
                W    L    H    x    y    z
Car -1 -1 1 2 1 2 1 1.701800 4.521200 1.397000 9.586657 0.155863 -1.103499 0.01
```

Informacje o przeszkodzie

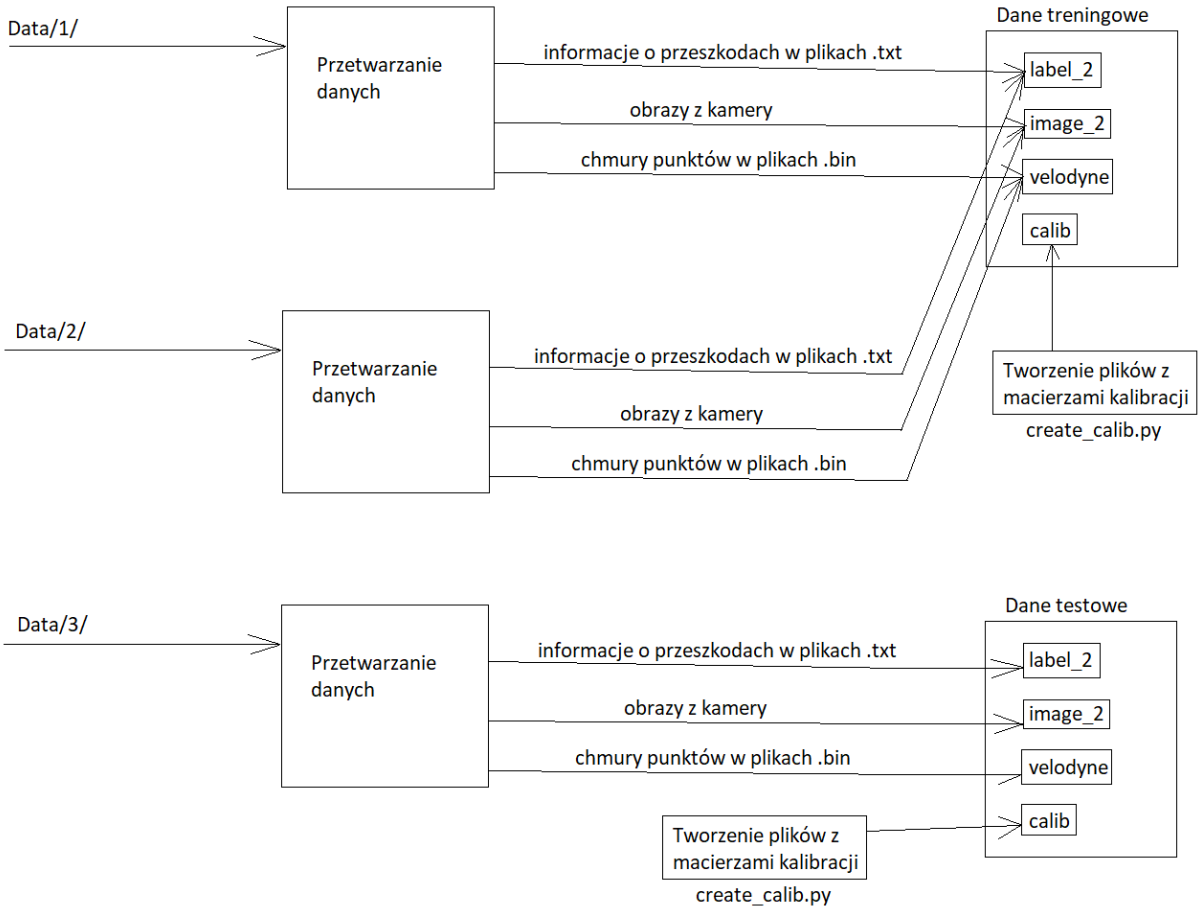
Rysunek 5.1: Dane, zawarte w plikach .bag. Macierze kalibracji kamery zostały znalezione w repozytorium Udacity [21] w folderze calibration/. Natomiast macierz przekształcenia pomiędzy układem kamery, a układem lidaru została dobrana przy pomocy wizualizacji.

Celem konwersji jest przetworzenie tych danych na format danych uczących zbioru KITTI. Dane uczące KITTI są podzielone na 3 foldery:

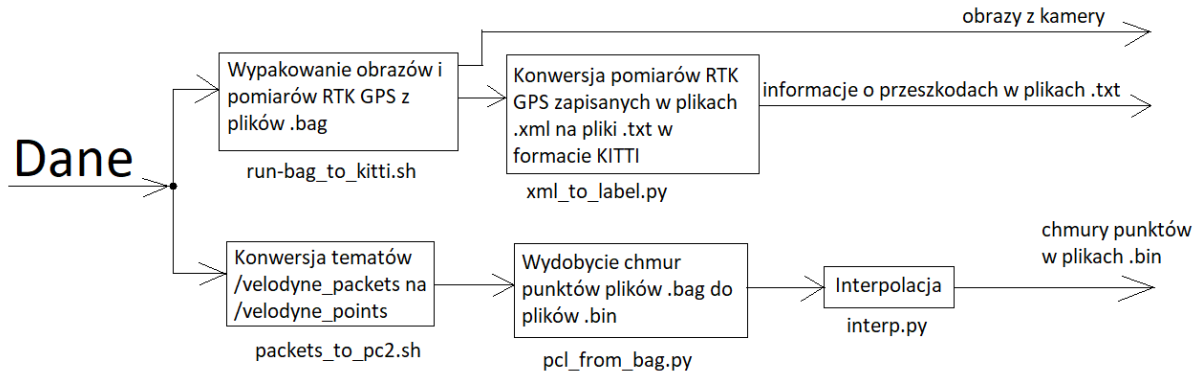
- velodyne - zawiera pliki .bin z chmurą punktów, każdy punkt jest reprezentowany przez 4 liczby (x, y, z, intensywność)

- label - zawiera pliki .txt z informacjami o przeszkodach między innymi: rodzaj przeszkody(np. samochód), współrzędne środka prostopadłościanu oraz wymiary przeszkody. Każdy wiersz pliku zawiera informacje o jednej przeszkodzie.
- calib - zawiera pliki .txt z macierzami potrzebnymi do kalibracji lidaru i kamery z nadajnikiem GPS

W celu umożliwienia wizualizacji danych, dodano folder image_2 z obrazami.



Rysunek 5.2: Schemat ogólny przetwarzania danych na zbiór uczący i testowy.

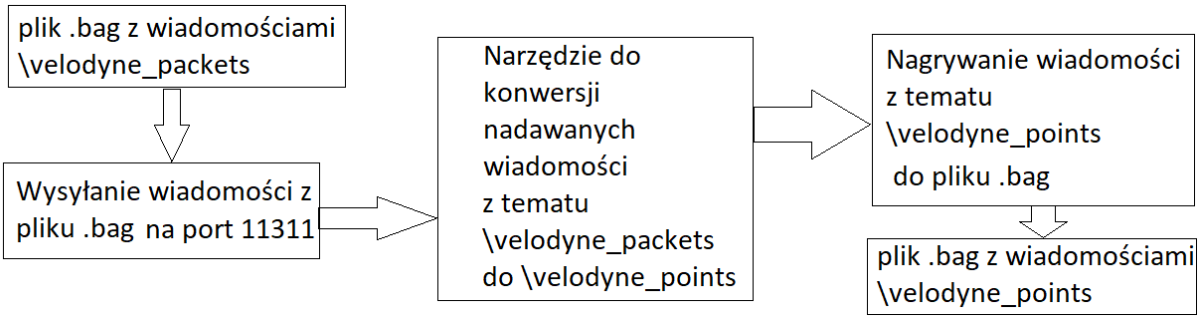


Rysunek 5.3: Schemat przetwarzania plików .bag na chmury punktów, dane o przeszkodach, dane kalibracji i obraz z kamery.

Najpierw wydobyto z plików .bag pliki .txt z informacją o położeniach i wymiarach przeszkód. W tym celu wydobyto położenie i wymiary przeszkód zapisane w plikach .xml za pomocą programu run-bag-to-kitti.sh. Następnie pliki .xml przekonwertowano na pliki .txt, które odpowiadają formatowi danych KITTI. Obrazy również zostały wydobyte programem run-bag_to_kitti.sh.

Drugim etapem było wydobycie chmur punktów z plików .bag. Aby wydobyć chmury punktów z danych Didi-Release-2, przed wykonaniem programu pcl_from_bag.py należało wykonać konwersję wiadomości z tematów /velodyne_packets na /velodyne_pointcloud. Odbywało się to w następujący sposób:

1. Uruchomienie w terminalu polecenia *roscore*, następnie przejść do kolejnego terminalu
2. Zainstalować pakiet Velodyne dla wersji ROS Kinetic używając polecenia *sudo apt-get install ros-kinetic-velodyne*
3. Uruchomić polecenie *rosrun velodyne_pointcloud cloud_node _calibration:=/opt/ros/kinetic/share/velodyne_pointcloud/params/32db.yaml*
4. W następnym terminalu uruchomić narzędzie do nagrywania wiadomości poleceniem *rosrun rosbag record -O "./pc/$file" /velodyne_points &*
5. W następnym terminalu uruchomić odtwarzanie wiadomości z tematu /velodyne_packets poleceniem *rosbag play -q "$file"*
6. Po zakończeniu odtwarzania wiadomości należy zakończyć nagrywanie, a następnie pozostałe terminale.

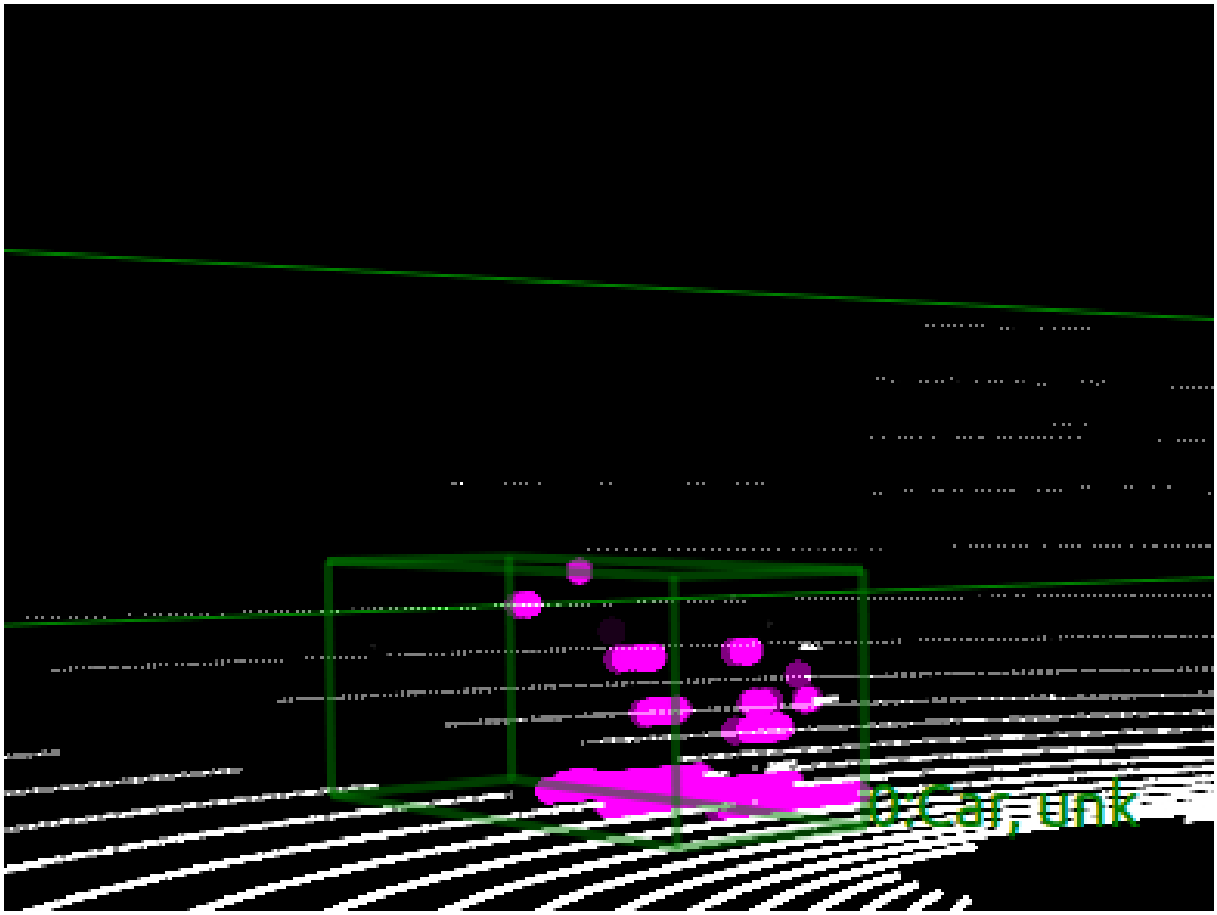


Rysunek 5.4: Schemat działania skryptu przetwarzającego pliki .bag z wiadomościami */velodyne_packets* na pliki .bag z wiadomościami */velodyne_points*.

Procedura konwersji wiadomości /velodyne_packets na velodyne_pointcloud została zautomatyzowana poprzez skrypt packets_to_pc2.sh. Tworząc oprogramowanie napisano program pcl_from_bag.py służący do wydobycia chmur punktów z otrzymanych danych i zapisania każdej z nich w pliku .bin. Program pcl_from_bag.py korzysta z modułów ROS, np. rospy. Chmury punktów były pobierane z lidaru rzadziej niż położenie z RTK GPS. Więc zastosowano interpolację polegającą na przyporządkowaniu każdej chmury punktów do najbliższego (pod względem czasu) pomiaru położenia. Tworząc oprogramowanie, napisano program interp.py służący do interpolacji.

Macierze potrzebne do kalibracji otrzymano, korzystając z plików z repozytorium Udacity w folderze calibration/. Macierze projekcji i korekcji [7] znajdują się w pliku ost.txt.

Pozostałe macierze dobrano w oparciu o wizualizację(opisaną w następnym rozdziale), osobno dla każdego z katalogów (Data/1, Data/2, Data/3). Dobierano macierze, tak aby prostopadłościan(obrazujący przeszkodę) pokrywał punkty należące do przeszkody. Tworząc oprogramowanie, napisano program create_calib.py służący do tworzenia plików z macierzami potrzebnymi do kalibracji.



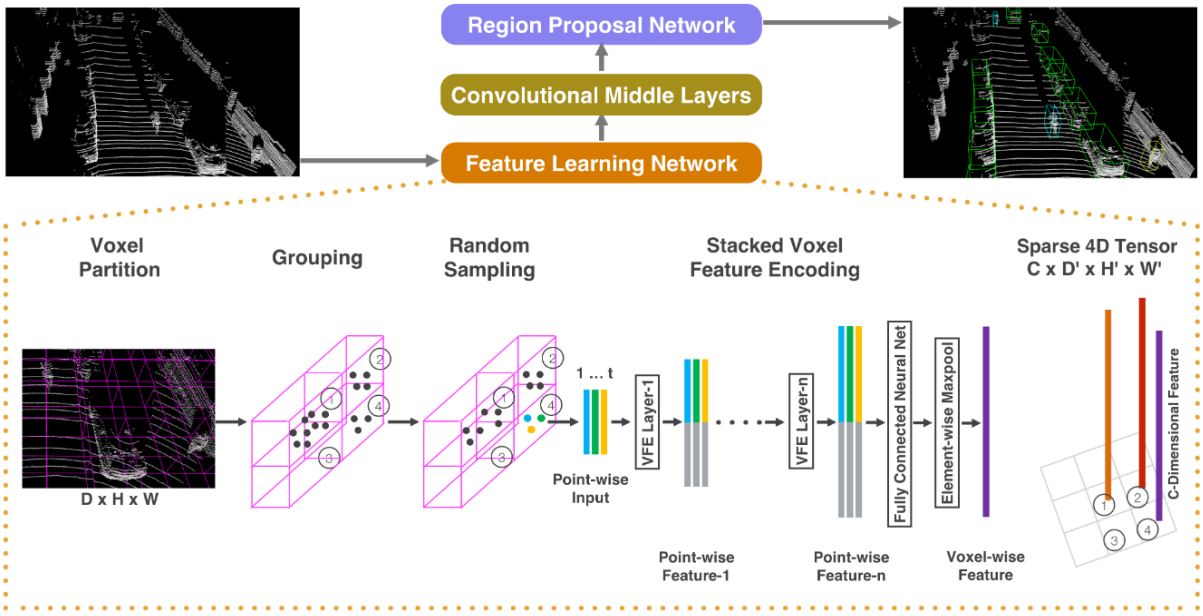
Rysunek 5.5: Parametry macierzy były dobierane tak, aby zielony prostopadłościan obejmował wszystkie punkty zawarte w przeszkodzie.

Rozdział 6

Konfiguracja rozwiązania

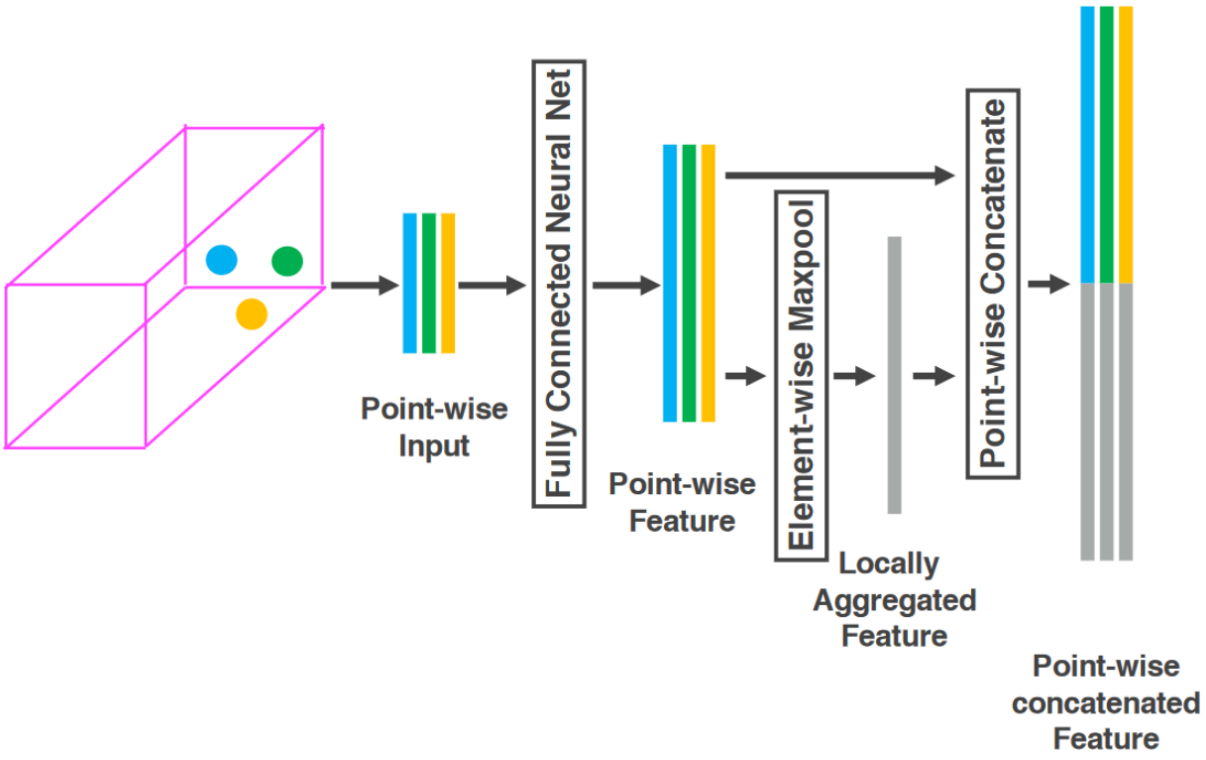
Próba uruchomienia najlepszego rozwiązania na danych z Udacity zakończyło się niepowodzeniem, ponieważ rozwiązanie wymaga danych położenia przeszkody na obrazie. Dane Udacity takiej informacji nie zawierają. Kolejnym rozwiązaniem według stworzonego rankingu jest rozwiązanie SECOND.

Rozwiązanie SECOND opiera się na sieci neuronowej *VoxelNet* [25] składającej się z trzech części Feature Learning Network, Convolutional Middle Layers, Region Proposal Network.



Rysunek 6.1: Struktura sieci VoxelNet.[25]

Feature Learning Network odpowiada za podział przestrzeni na woksele (najmniejszy element w grafice trójwymiarowej), ograniczenie liczby punktów znajdujących się w jednym wokselu oraz przetworzenie wokseli na wielowymiarowe wektory cech. Ograniczenie liczby punktów przypadających na jeden woksel odbyło się poprzez losowe wybranie pewnej liczby punktów. Przetworzenie wokselu na wektor odbywa się za pomocą sieci składającej się z n warstw VFE (Voxel Feature Encoding Layer).



Rysunek 6.2: Warstwa VFE.[25]

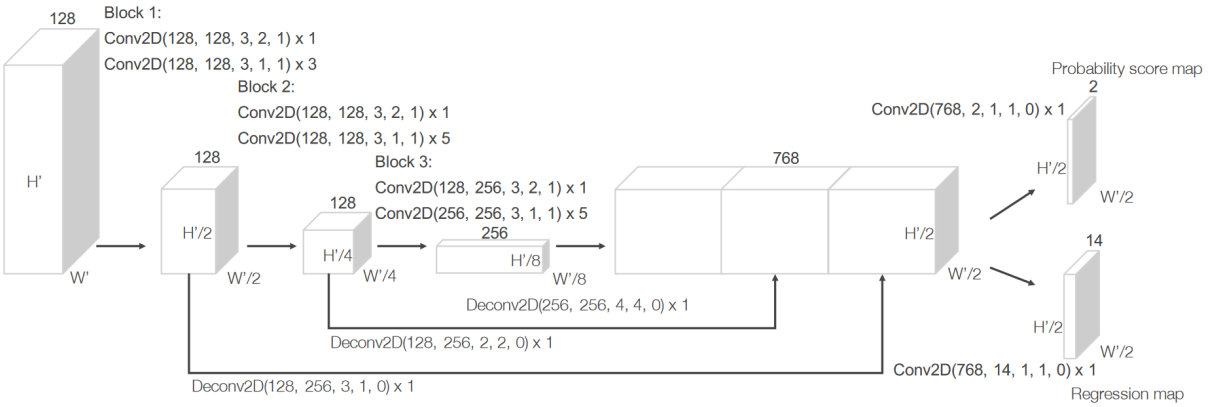
Na wejściu jest zbiór punktów. Najpierw Każdy punkt osobno przechodzi przez sieć FCN (Fully Connected Neural Net). FCN składa się z warstwy o liniowej funkcji aktywacji, warstwy normalizującej oraz warstwy prostującej (Rectifier). Po przejściu wszystkich punktów przez sieć FCN otrzymuje się t wektorów cech punktów f_i (na rysunku są to kolorowe prążki) dla każdego punktu. Każdy z tych wektorów ma C_1 współrzędnych. Następnie tworzony jest wektor $\hat{f}$ (odpowiada mu na rysunku prążek koloru szarego) , którego współrzędne wyrażają się wzorem:

$$\hat{f}^{(k)} = \max_{i \in \{1,\dots,t\}} f_i^{(k)},$$

gdzie $k \in 1, 2, ..., C_1$ jest numerem współrzędnej. Tak utworzony wektor $\hat{f}$ jest łączony poprzez operację konkatenacji z każdym wektorem f_i. Na wyjściu całej warstwy VFE znajduje się zbiór t wektorów. Każdy z wektorów jest konkatenacją wektora f_i oraz wektora $\hat{f}$. Po przejściu przez n warstw VFE każdy z uzyskanych wektorów przechodzi przez warstwę FCN. Każdy z uzyskanych wektorów cech punktów f_i ma C współrzędnych. Następnie tworzony jest wektor cech wokselu F według następującego wzoru:

$$\hat{F}^{(k)} = \max_{i \in \{1,\dots,t\}} F_i^{(k)},$$

gdzie $k \in 1, 2, ..., C$ jest numerem współrzędnej. Po obliczeniu dla każdego niepustego wokselu wektora cech otrzymuje się czterowymiarowy tensor. Tensor przechodzi przez środkowe warstwy konwolucyjne (*Convolutional Middle Layers*), a następnie przez sieć RPN - *Region Proposal Network* .



Rysunek 6.3: Struktura sieci RPN.[25]

Sieć RPN składa się z trzech bloków warstw konwolucyjnych. Każdy z bloków zmniejsza rozmiar swojego wejścia o połowę. Po każdej warstwie konwolucyjnej następuje warstwa normalizująca oraz warstwa prostująca (ReLU). Na wyjściu sieci VoxelNet jest mapa prawdopodobieństw oraz mapa regresji. Mapa prawdopodobieństw określa dla każdego obszaru prawdopodobieństwo że przeszkoda go zajmuje. Mapa regresji określa dla każdego obszaru następujące parametry: znormalizowaną różnicę położenia środka obszaru od środka przeszkody $\Delta x = \frac{x_c^d - x_c^o}{d^o}$, $\Delta y = \frac{y_c^d - y_c^o}{d^o}$, $\Delta x = \frac{z_c^d - z_c^o}{h^o}$; logarytm ilorazu wymiarów przeszkody i obszaru $\Delta l = log(\frac{l^d}{l^o})$, $\Delta w = log(\frac{w^d}{w^o})$, $\Delta h = log(\frac{h^d}{h^o})$ oraz różnicę kątów orientacji $\Delta\theta = \theta^d - \theta^o$, gdzie indeks górny określa, czego dotyczy parametr (d - detekcja, o - obszar). Mapa regresji określa dla każdego obszaru wektor $u = (\Delta x, \Delta y, \Delta z, \Delta l, \Delta w, \Delta h, \Delta\theta)$. Funkcja strat wyraża się wzorem:

$$L = \alpha \frac{1}{N_{pos}} \sum_i L_{cls}(p_i^{pos}, 1) + \beta \frac{1}{N_{neg}} \sum_j L_{cls}(p_j^{neg}, 0) + \frac{1}{N_{pos}} \sum_i L_{reg}(u_i, u_i^*),$$

gdzie i jest indeksem obszaru zajmowanego przez przeszkodę, N_{pos} jest liczbą obszarów zajmowanych przez przeszkodę, j jest indeksem obszaru niezajmowanego przez przeszkodę, N_{neg} jest liczbą obszarów niezajmowanych przeszkodę, p_i^{pos} jest prawdopodobieństwem że obszar o indeksie i jest zajmowany przez przeszkodę, p_j^{neg} jest prawdopodobieństwem że obszar o indeksie j jest niezajmowany przez przeszkodę, u_i^* jest wektorem stworzonym w analogiczny sposób jak wektor u_i (różnica polega na tym, że zamiast parametrów detekcji są brane parametry danej przeszkody). Składowe dla mapy prawdopodobieństw są definiowane następująco:

$$L_{cls}(p^{pos}, 0) = log(p^{pos})$$
$$L_{cls}(p^{neg}, 0) = log(1 - p^{neg})$$

Pierwsza z nich jest definiowana dla obszarów, zajmowanych przez przeszkodę. Druga z nich jest definiowana dla obszarów niezajmowanych przez przeszkodę. Dla mapy regresji funkcja strat wyraża się wzorem:

$$L_{reg}(u, u^*) = \begin{cases} |u - u^*| & \text{jeśli } |u - u^*| > \gamma; \\ \frac{1}{|\gamma|}(u - u^*)^2 & \text{jeśli } |u - u^*| \leqslant \gamma \end{cases},$$

gdzie γ jest dobranym parametrem funkcji.

Pierwszym krokiem w konfiguracji rozwiązania jest zainstalowanie środowiska Anaconda 3 w katalogu domowym. W tym środowisku stworzyłem środowisko z programem Python 3.6.7 i potrzebnymi modułami wykonując następujące komendy:

source /anaconda/bin/activate/

conda create -n secondEnv python=3.6.7

conda activate secondEnv

pip install shapely fire pybind11 pyqtgraph

pip install tensorboardX protobuf numba

sudo apt-get install libboost-all-dev

Moduł PyTorch jest biblioteką służącą do korzystania z sieci neuronowych. Do jego instalacji służy polecenie:

pip install torch

Potem zainstalowałem narzędzie SparseConvNet według instrukcji podanej przez twórcę rozwiązania SECOND. Aby zainstalować SparseConvNet należało uruchomić terminal w katalogu second.pytorchUdacity i wykonać następujące polecenia:

conda install pytorch-nightly -c pytorch

conda install google-sparsehash -c bioconda

conda install -c anaconda pillow

git clone https://github.com/facebookresearch
/SparseConvNet.git

cd SparseConvNet/

bash build.sh

Następnie ustawiłem trzy zmienne wskazujące na położenie wybranych elementów CUDA. Wykonałem to poprzez wstawienie ma początek pliku /bashrc następujących linii:

export NUMBAPRO_CUDA_DRIVER=/usr/lib/
x86_64-linux-gnu/libcuda.so

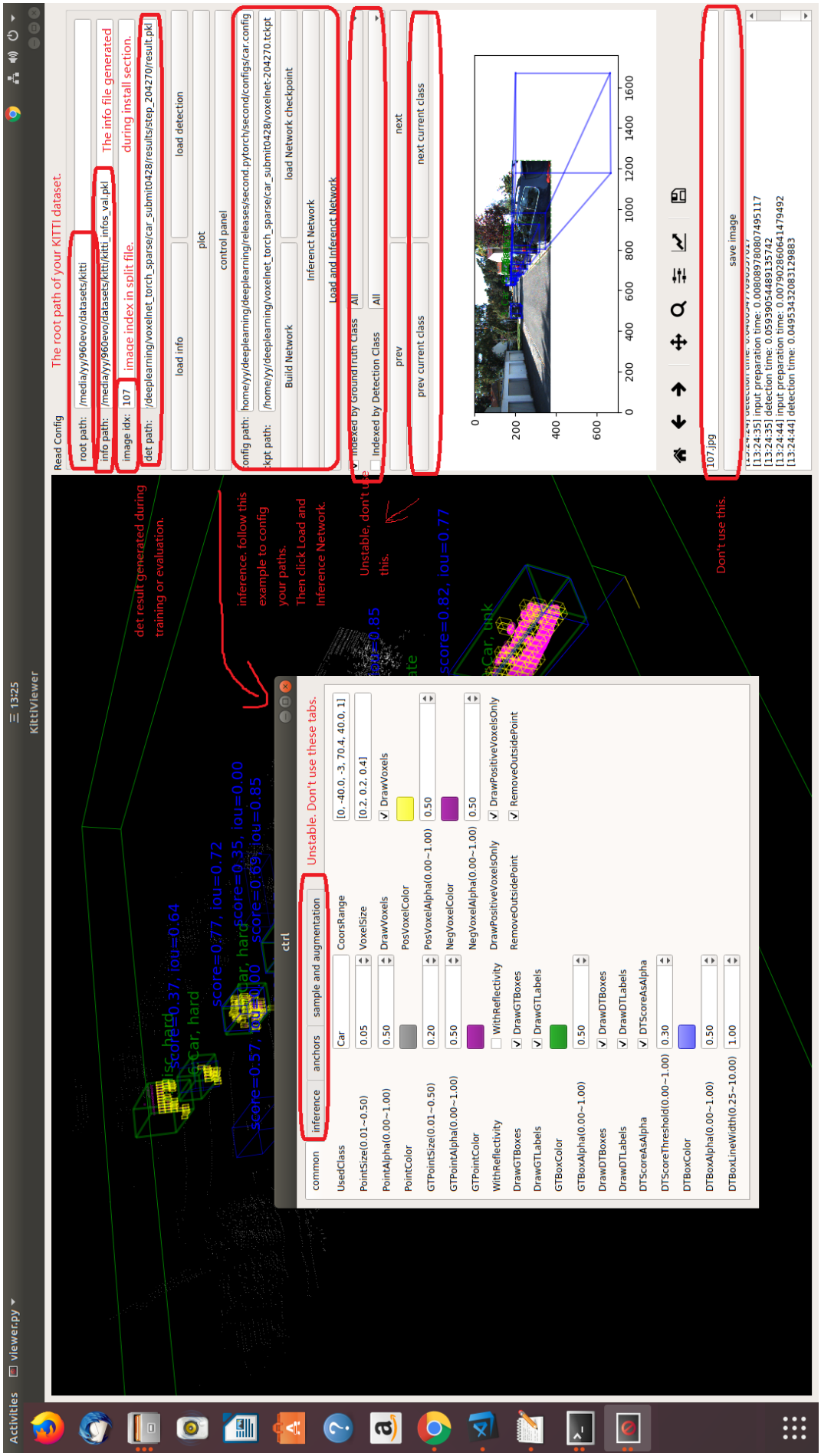
export NUMBAPRO_NVVM=/usr/local/cuda/nvvm/
lib64/libnvvm.so

export NUMBAPRO_LIBDEVICE=/usr/local/cuda/
nvvm/libdevice

Kolejnym etapem było przygotowanie danych treningowych i testowych do podania ich na sieć neuronową. Służą temu polecenia:

python create_data.py create_kitti_info_file
–data_path=./data/KITTI/

python create_data.py create_groundtruth_database
–data_path=./data/KITTI/



Rysunek 6.4: Wygląd okna programu do wizualizacji danych.[24]

Po przygotowaniu danych można wykonać wizualizację danych. W repozytorium tego rozwiązania znajduje się program viewer.py służący do wizualizacji danych.

Podczas wizualizacji danych okazało się że wartości współrzędnych **x** i **z** wektora translacji jest błędna. Rozwiązałem ten problem poprzez dobranie wartości współrzędnych **x** i **z** dla każdego ze zbiorów (Data/1/, Data/2/ i Data/3/) metodą prób i błędów. Również zauważyłem że w niektórych podzbiorach danych pomiary GPS wskazują położenie przeszkody niezgodne z obrazem z kamery oraz ze skanem 3D. Tymi podzbiorami są:

- Dane uczące
  - w folderze Data/1/
    - 4_f, 6_f, 15, 17, 18, 19, 20, 21_f, 23, 26
  - w folderze Data/2/
    - 3_f, 11_f, 13, 14_f
- Dane treningowe
  - w folderze Data/3/
    - 12_f, 13_f, 14, 15_f

Te podzbiory zostały usunięte z danych uczących i testowych.

Przed treningiem sieci ustawiłem w pliku car.tiny.config odpowiednie ścieżki do danych:

- Linia 120: database_info_path: "/ścieżka/do/KITTI/kitti_dbinfos_train.pkl"
- Linia 146: kitti_info_path: "/ścieżka/do/KITTI/kitti_infos_train.pkl"
- Linia 147: kitti_root_path: "/ścieżka/do/KITTI/"
- Linia 190: kitti_info_path: "/ścieżka/do/KITTI/kitti_infos_val.pkl"
- Linia 191: kitti_info_path: "/ścieżka/do/KITTI/kitti_infos_test.pkl"
- Linia 192: kitti_root_path: "/ścieżka/do/KITTI/"

W pliku car.tiny.config linia 191 jest zakomentowana znakiem '#' natomiast w pliku car.tiny.config zakomentowana jest linia 190. Poza tą różnicą tekst z pliku car.tiny.config jest taki sam jak w car.tiny.config.test.

Aby sieć wykrywała przeszkody położone za pojazdem należy zmienić niektóre parametry pliku car.tiny.config w następujący sposób:
Linia 4: point_cloud_range : [-52.8, -32.0, -1, 52.8, 32.0, 3]
Linia 61: post_center_limit_range: [-70.4, -40, -5.0, 70.4, 40, 5.0]
Linia 82: anchor_ranges: [-52.8, -32.0, -1.78, 52.8, 32.0, -1.78]

W celu zmniejszenia ilości zużywanej pamięci zmieniłem rozmiar sieci neuronowej.

Linia 12: —> num_filters: [16, 32]

Linia 24: —> num_filters: [32, 64, 64]

Linia 26: —> num_upsample_filters: [64, 64, 64]

Linia 22: —> layer_nums: [2, 3, 3]

Następnie uruchomiłem trening poleceniem **python train.py train −−config_path=./second/configs/car.tiny.config −−model_dir=‘pwd‘/model/**. Na końcu sprawdziłem sieć na danych testowych, wykonując następujące polecenia:

- wygenerowanie wyjścia sieci na podstawie danych testowych

  python train.py evaluate
  −−config_path=./second/configs/car.tiny.config.test
  −−model_dir=‘pwd‘/model/ −−pickle_result=False

- Konwersja wyjścia sieci na format danych Udacity (pliki .xml)

  cd ./model/eval_results/

  python2 trans_labels_to_Udacity.py ./step_*/

- Uruchomienie programu evaluate_tracklets.py

  cd ../../../

  python ./Didi-Release-2/tracklets/python/evaluate_tracklets.py ‘pwd‘/second.pytorchUdacity/model/eval_results/3/tracklets/ ‘pwd‘/Didi-Release-2/Data/3/TRACKLETS/

W czasie konwersji, pliki .txt z liniami określającymi położenie i wymiary przeszkód są zamieniane na pliki .xml. Do tego celu, tworząc oprogramowanie, napisano program trans_labels_to_Udacity.py.

Rozdział 7

Wyniki numeryczne

Średni wskaźnik IoU na danych testowych Udaity wynosi 40%. 64% detekcji ma wskaźnik IoU większy niż 50%, natomiast 27% detekcji ma wskaźnik IoU większy niż 70%. Trening wymagał przejścia 5 razy przez cały zbiór danych uczących, który zawiera 3500 elementów. Podczas wizualizacji zauważyłem że sieć neuronowa nie wykrywa przeszkód które są oddalone od pojazdu więcej niż ok. 25m. Czas trwania treningu wynosił ok. 1h na karcie graficznej nVidia GeForce GTX 1050 (4GB pamięci) i procesorze Intel Core i7(maksymalna częstotliwość taktowania: 3.8GHz). Czas detekcji przeszkody wynosił średnio ok. 0.025s.

Ze względu na zasięg ok.25m i drogę hamowania, zaprogramowana metoda wykrywania przeszkód ma zastosowanie tylko dla prędkości nie przekraczającej 50km/h. Ta prędkość dotyczy pojazdu sterowanego oraz innych uczestników ruchu drogowego, dlatego zastosowanie oprogramowania ogranicza się do jazdy w terenie zabudowanym.

Rozdział 8

Perspektywy rozwoju

Oprogramowanie można rozwijać pod względem skuteczności oraz zasięgu detekcji. W detekcji (na danych testowych) zauważono błąd systematyczny, polegający na tym że przeszkoda ciągle jest wykrywana o ok. 0.3m za nisko. Źródło problemu nie jest dokładnie zidentyfikowane, więc wyeliminowanie błędu systematycznego detekcji jest sposobem na rozwinięcie oprogramowania. Kolejnym sposobem ulepszenia detekcji jest zmiana parametrów sieci neuronowej, tak aby wskaźnik IoU się poprawił. Innym sposobem udoskonalenia oprogramowania może być rozszerzenie wejścia sieci neuronowej o obraz z kamery. Wtedy sieć neuronowa wykrywałaby przeszkody na podstawie skanu 3D z lidaru oraz obrazu z kamery.

Rozdział 9

Podsumowanie i wnioski

Zrealizowano pracę wykonując najpierw przegląd rozwiązań, wybór najlepszego z nich według wybranego kryterium, przetworzenie danych do odpowiedniej postaci i konfigurację rozwiązania. Do realizacji przetworzenia danych oraz konfiguracji rozwiązania użyto następującego oprogramowania: CUDA 9.0, cuDNN 7.3, ROS Kinetic, Docker CE, Anaconda 3, Python 3.6 wraz z modułem PyTorch służącym do korzystania z sieci neuronowych. Całe oprogramowanie było używane pod systemem Ubuntu 16.04. Praca odbywała się na komputerze z kartą graficzną GTX 1050, która była wykorzystana w celu przyspieszenia obliczanie wartości wyjściowej oraz trening sieci neuronowej.

Dodatek A

Instalacja CUDA 9.0 oraz CUDNN 7.3

Pobrać plik instalatora i cztery pakiety ze strony `https://developer.nvidia.com/cuda-90-download-target_os=Linux&target_arch=x86_64&target_distro=Ubuntu&target_version=1604&target_type=deblocal`. Razem powinno być pięć pobranych plików **.deb**. Następnie przejść do katalogu z pobranymi plikami i wykonać następujące polecenia:

```
sudo dpkg -i cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64.deb
for dir in /var/cuda-repo-*/
do
    sudo apt-key add $dir/7fa2af80.pub
done
sudo apt-get update
sudo apt-get install cuda
```

Następnie trzeba dodać dwie linie na początek pliku /.bashrc

```
export PATH=/usr/local/cuda-9.0/bin$PATH:+:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda-9.0/
lib64$LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH
```

Aby sprawdzić poprawność instalacji należy wykonać:

```
cuda-install-samples-9.0.sh /
cd /NVIDIA_CUDA-9.0_Samples
make
cd bin/x86_64/linux/release/
./deviceQuery
./bandwidthTest
```

Wykonanie programów deviceQuery oraz bandwidthTest powinno wyświetlić na końcu "Result = PASS".

Więcej informacji o CUDA 9.0 znajduje się na stronie
`https://docs.nvidia.com/cuda/archive/9.0/`

Aby zainstalować CUDNN należy wejść na stronę `https://developer.nvidia.com/rdp/cudnn-archive` i pobrać trzy zaznaczone na ilustracji pliki. W razie konieczności trzeba założyć konto i zalogować się na stronie `https://developer.nvidia.com`. Po pobraniu plików należy wejść do katalogu z pobranymi plikami i wykonać:

```
sudo dpkg -i libcudnn7*
```

W celu weryfikacji poprawności instalacji należy wykonać:

```
cp -r /usr/src/cudnn_samples_v7/ $HOME

cd $HOME/cudnn_samples_v7/mnistCUDNN

make clean && make

./mnistCUDNN
```

Po wykonaniu powinien pojawić się napis "Test passed!".

Więcej informacji o CUDNN znajduje się na `https://docs.nvidia.com/deeplearning/sdk/cudnn-install/index.html#installlinux`

Dodatek B

Instalacja ROS Kinetic

W celu instalacji ROS Kinetic należy wykonać:

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

sudo apt-key adv –keyserver hkp://ha.pool.sks-keyservers.net:80 –recv-key 421C365BD9FF1F7178

sudo apt-get update

sudo apt-get install ros-kinetic-desktop-full

sudo rosdep init

rosdep update

echo "source /opt/ros/kinetic/setup.bash" » /.bashrc

source /.bashrc

Następnie należy zainstalować przydatne pakiety

sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential

sudo apt-get install ros-kinetic-velodyne

Więcej informacji o ROS Kinetic na `http://wiki.ros.org/kinetic/Installation/Ubuntu`. Poradnik na temat wizualizacji danych Udacity w oprogramowaniu ROS znajduje się na `https://github.com/udacity/didi-competition/blob/master/docs/GettingStarted.md`

Dodatek C

Instalacja Docker CE

W celu instalacji Docker CE należy wykonać:

sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo apt-key fingerprint 0EBFCD88

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

sudo apt-get update

sudo apt-get install docker-ce

Aby sprawdzenić poprawność instalacji należy uruchomić:

sudo docker run hello-world

Więcej informacji o Docker CE znajduje się na stronie
`https://docs.docker.com/install/linux/docker-ce/ubuntu/`

Bibliografia

[1] anaconda developers. https://www.anaconda.com/download/, data dostępu: 5.11.2018.

[2] cuda developers. https://docs.nvidia.com/cuda/archive/9.0/, data dostępu: 5.11.2018.

[3] cuda developers. https://docs.nvidia.com/deeplearning/sdk/cudnn-install/index.html#installlinux, data dostępu: 5.11.2018.

[4] docker ce developers. https://docs.docker.com/install/linux/docker-ce/ubuntu/, data dostępu: 5.11.2018.

[5] V. Dumoulin, F. Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, mar 2016.

[6] A. Geiger. The kitti vision benchmark suite, http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d, data dostępu: 31.10.2018.

[7] A. Geiger, P. Lenz, R. Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[8] J. Hui. https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-dete data dostępu: 31.10.2018.

[9] S. Ioffe, C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[10] W. J. J. Żurada, M. Barski. Sztuczne sieci neuronowe, 1996.

[11] A. G. Josh Patterson. Deep learning. praktyczne wprowadzenie, 2018.

[12] D. P. Kingma, J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[13] J. Ku, M. Mozifian, J. Lee, A. Harakeh, S. Waslander. Joint 3d proposal generation and object detection from view aggregation. *IROS*, 2018.

[14] J. Ku, M. Mozifian, J. Lee, A. Harakeh, S. L. Waslander. https://arxiv.org/pdf/1712.02294.pdf, data dostępu: 5.12.2018.

[15] M. E. McGrath. Autonomous vehicles: Opportunities, strategies, and disruptions, 2018.

[16] C. R. Qi, W. Liu, C. Wu, H. Su, L. J. Guibas. Frustum pointnets for 3d object detection from rgb-d data. *arXiv preprint arXiv:1711.08488*, 2017.

[17] ros kinetic developers. `http://wiki.ros.org/kinetic/Installation/Ubuntu`, data dostępu: 5.12.2018.

[18] scikit-learn developers. `https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html`, data dostępu: 5.12.2018.

[19] ubuntu 16.04 developers. `http://releases.ubuntu.com/16.04/`, data dostępu: 5.12.2018.

[20] Udacity, Didi. `http://academictorrents.com/details/18d7f6be647eb6d581f5ff61819a11b9c21769c7`, data dostępu: 31.10.2018.

[21] Udacity, Didi. `https://github.com/udacity/didi-competition/`, data dostępu: 31.10.2018.

[22] Udacity, Didi. `https://github.com/udacity/didi-competition/blob/master/docs/GettingStarted.md`, data dostępu: 31.10.2018.

[23] D. S. P. R. Valentino Zocca, Gianmario Spacagna. Python deep learning: Next generation techniques to revolutionize computer vision, ai, speech and data analysis, 2017.

[24] Y. Yan. `https://github.com/traveller59/second.pytorch`, data dostępu: 26.10.2018.

[25] O. T. Yin Zhou. Voxelnet: End-to-end learning for point cloud based 3d object detection, data dostępu: 26.11.2018.