

Rozpoznawanie i Przetwarzanie Obrazów

”Kamera samochodowa z funkcją wykrywania obiektów i panelem konfiguracyjnym”

Michał Bernacki-Janson 264021, Adam Czekalski 264488

11 Czerwca 2024

Spis treści

1 Opis zrealizowanych prac w projekcie	2
1.1 Przebieg prac	2
1.2 Informacje o napotkanych problemach w trakcie prac projektowych	2
1.3 Informacje o niespodziewanych efektach	2
2 Architektura niskopoziomowa wraz z fragmentami kodu programu	3
3 Opis wykorzystanych modeli i metod przetwarzania obrazów	6
4 Wyniki testów / eksperymentów	10
4.1 Podjechanie blisko, centralnie do słupka	10
4.2 Parkowanie przodem przed ścianą	11
4.3 Pieszy przechodzący przed samochodem	12
4.4 Zbliżanie się przodem do pojazdów	13
4.5 Testy aplikacji konfiguracyjnej pod względem wprowadzania parametrów niezgodnych z założeniami projektowymi	14
4.6 Testy na żywo	14
5 Wnioski z projektu	15
5.1 Dotyczące implementacji architektury niskopoziomowej	15
5.2 Dotyczące tworzenia dataset'u	15
5.3 Dotyczące trenowania modelu i samego modelu	16
5.4 Dotyczące implementacji kodu	16
5.5 Wnioski ogólne	16
6 Dokumentacja użytkownika (instrukcja obsługi programu)	16
6.1 Metody uruchomienia programu	16
6.2 Sposób obsługi interfejsu użytkownika	17
6.3 Interpretacja działania programu	18
7 Literatura	19

1 Opis zrealizowanych prac w projekcie

1.1 Przebieg prac

Prace nad projektem rozpoczęły się od sporządzenia dokumentu zawierającego wstępne założenia, jak finalny produkt będzie się prezentować. Następnie nagrano materiały służące do testów oprogramowania. Kamera internetowa IMILAB CMSXJ22A została przymocowana do zderzaka samochodu, a kabel USB przedłużony i przeciągnięty przez otwarte okno do wnętrza samochodu, aż do portu USB w laptopie, który pozwolił nagrać materiały wideo.



Rysunek 1: Implementacja architektury niskopoziomowej w celach testowych

Następnie, rozpoczęto prace nad oprogramowaniem. Na początek skupiono się na tym, aby program mógł wykrywać następujące rodzaje obiektów: osoba, samochód osobowy, słupek oraz ścianę. Każda klasa obiektów może być zaznaczana za pomocą prostokąta na ekranie wraz z wartością pewności modelu o jego przynależności do klasy. Zapewniono również wsparcie dla akceleracji obliczeń na karcie graficznej za pomocą sterowników CUDA. Kolejnym krokiem, było stworzenie konfiguratora pozwalającego na modyfikację parametrów wejściowych programu, t.j.: kadrowanie obrazu, wybór klas obiektów, które mają być wykrywane, ustawienie minimalnej pewności, od której mają być wykrywane obiekty, wybór źródła obrazu, tzn.: wskazanie konkretnej ścieżki do wideo bądź wykrywanie w czasie rzeczywistym za pomocą kamerki internetowej. Oprócz tego, narysowano na ekranie linie parkowania, które także można modyfikować za pomocą konfiguratora: liczba przedziałek, ich wysokość (zasięg), szerokość i kąt nachylenia względem ekranu. Ostatnim elementem było zaimplementowanie sygnału dźwiękowego, wraz ze zwiększającą się jego częstotliwością w miarę zbliżania się do najbliższego obiektu danej klasy. Stworzono także wizualizację linii parkowania w konfiguratorze przed rozpoczęciem wykrywania, na prośbę prowadzącego.

1.2 Informacje o napotkanych problemach w trakcie prac projektowych

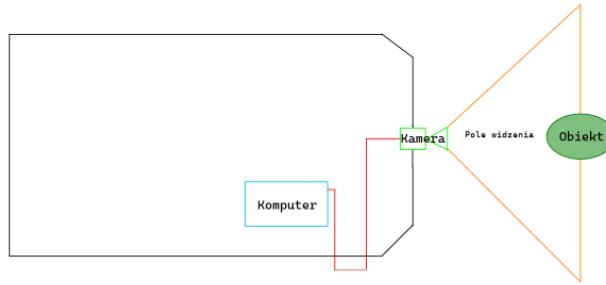
W trakcie realizacji projektu, pojawiły się również problemy. Jednym z nich okazała się sama instalacja biblioteki Ultralytics YOLOv8 oraz jej zależności. Niektóre zależności trzeba było zainstalować w wersji specjalnie dobranej do reszty. Dodatkowo, problem stał się większy kiedy zdecydowano się na użycie akceleracji za pomocą GPU. Wymagało to doinstalowania dodatkowych sterowników, CUDA toolkit i CUDNN oraz ich konfiguracji.

1.3 Informacje o niespodziewanych efektach

Problem napotkano także w trakcie trenowania modelu - u jednego z autorów projektu, wytrenowanie tego samego "datasetu", z tymi samymi parametrami, dawał inne efekty niż u drugiego. Zweryfikowano zainstalowane biblioteki i narzędzia - zauważono pełną zgodność. Stwierdzono zatem, że wynika to z własności

sieci neuronowych. Okazało się także, że przy przygotowywaniu datasetu do modelu na witrynę RoboFlow, użycie dwóch różnych form zaznaczenia obiektów: metodą "polygon" oraz poprzez prostokąt powoduje pominięcie jednego z nich przez model.

2 Architektura niskopoziomowa wraz z fragmentami kodu programu



Rysunek 2: Architektura niskopoziomowa implementowanego systemu

Architekturę oprogramowania realizowano zgodnie z założeniami. Dodatkowo do stworzenia interfacu graficznego wykorzystano bibliotekę CustomTkinter.

Poniżej przedstawiono analizę najważniejszych fragmentów kodu.

```
# Sprawdzenie czy program wykrył dedykowana kartę graficzną z CUDA
print(torch.cuda.is_available())
print(torch.cuda.device_count())
print(torch.cuda.get_device_name(0))
```

Powyższy fragment kodu pozwala na sprawdzenie, czy program uruchomiony został na dedykowanej karcie graficznej z CUDA, w efekcie upewnienie się, że została uruchomiona akceleracja graficzna.

```
# ustawienie parametrow prostokata zaznaczajacego obiekt
box_annotator = sv.BoxAnnotator(
    thickness=2,
    text_thickness=2,
    text_scale=1
)

# przefiltrowanie wykrytych obiektow na interesujace nas klasy, ustalone w konfiguratorze
detections = detections[np.isin(detections.class_id, selected_classes)]

# przefiltrowanie wykrytych obiektow, ktore posiadaja pewnosc wykrycia wieksza badz
# rowna, ustalona w konfiguratorze
detections = detections[detections.confidence > confidence]

# Wykrycie obiektow w otrzymanej klatce video (otrzymanie tablic ze wspolrzednymi 2 punktow
# wykrycia obiektu - lewy gorny rog prostokata oraz prawny dolny rog prostokata)
result = model(frame, agnostic_nms=True)[0]
detections = sv.Detections.from_yolov8(result)

# Stworzenie opisow na ramce kwadratu (nazwa klasy, pewnosc detekcji (wartosc [0-1]))
labels = []
for element in detections:
    label = f'{model.model.names[element[2]}], {element[1]:0.2f}'
```

```

    labels.append(label)

# Nanieśienie oznaczeń na klatkę wideo
frame = box_annotator.annotate(scene=frame, detections=detections, labels=labels)

```

Powyższy fragment kodu pozwala na zaznaczenie wykrytego obiektu na ekranie, to znaczy:

- Ustawienie parametrów prostokąta zaznaczającego obiekt
- Przefiltrowanie wykrytych obiektów na interesujące nas klasy, ustalone w konfiguratorze
- Przefiltrowanie wykrytych obiektów, zgodnie z parametrami ustalonymi w konfiguratorze
- Wykrycie obiektów w otrzymanej klatce wideo (otrzymanie tablic z 2 punktami wykrycia obiektu - lewy górny róg prostokąta oraz prawny dolny róg prostokąta)
- Stworzenie opisów na ramce kwadratu (nazwa klasy, pewność detekcji (wartość [0-1]))
- Nanieśenie oznaczeń na klatkę wideo

```

# rysowanie linii rozpoczynane jest od samego dołu obrazu
start_draw_lines_y = crop_y2
# określenie, w którym miejscu obrazu linie mają przestąpić się rysować, na podstawie ilości
# linii oraz odstępów między nimi, ustalonych w konfiguratorze
end_draw_lines_y = start_draw_lines_y - nr_of_lines * space_between_lines

# skąd zaczynane jest rysowanie linii (w osi y, lewy i prawy skraj osi x)
height_of_line = crop_y2

# wyznaczenie środka obrazu
middle = int(abs(crop_x2 - crop_x1) / 2)
middle_line_left_x = middle - int(offset/2)
middle_line_right_x = middle + int(offset/2)
nr_of_line = 0

start_drawing_bottom_left = middle_line_left_x
start_drawing_bottom_right = middle_line_right_x

# grubość linii: 9px
thickness = 9

middle_line_array = []

# rysowanie linii na klatce obrazu
for i in range(start_draw_lines_y, end_draw_lines_y, -space_between_lines):
    # rysowanie lewej krawędzi linii
    draw_line(start_drawing_bottom_left, height_of_line, middle_line_left_x, i, nr_of_line,
              frame, thickness)

    # rysowanie prawej krawędzi linii
    draw_line(start_drawing_bottom_right, height_of_line, middle_line_right_x, i,
              nr_of_line, frame, thickness)

    # "przesunięcie" parametrow, żeby móc narysować przedziałki
    middle_line_array.append((i, nr_of_line, middle_line_left_x, middle_line_right_x))
    height_of_line = i
    start_drawing_bottom_left = middle_line_left_x
    start_drawing_bottom_right = middle_line_right_x

```

```

# rysowanie linii przedzialki od lewej
draw_line(middle_line_left_x, i, middle_line_left_x + 100, i, nr_of_line, frame,
thickness)

# rysowanie linii przedzialki od prawej
draw_line(middle_line_right_x, i, middle_line_right_x - 100, i, nr_of_line, frame,
thickness)

nr_of_line += 1

# dodanie ustalonego katu pod jakimi linie maja byc rysowane, ustalony w konfiguratorze
middle_line_left_x += angle_of_lines
middle_line_right_x -= angle_of_lines

```

Powyższy fragment kodu pozwala na narysowanie linii parkowania. Algorytm rysowania linii wygląda w następujący sposób:

- Rysowanie linii rozpoczynane jest od samego dołu obrazu.
- Określenie, w którym miejscu obrazu linie mają przestać się rysować, na podstawie ilości linii oraz odstępów między nimi, ustalonych w konfiguratorze.
- Określenie skąd zaczynane jest rysowanie linii (w osi y, lewy i prawy skraj osi x).
- Wyznaczenie środka obrazu w osi x.
- Określenie grubości linii: 9px.
- Wejście do pętli i rysowanie kolejnych poziomów linii. Rozpoczęcie od samego dołu, aż do wyliczonego wcześniej limitu w osi y:
 - Rysowanie lewej krawędzi linii.
 - Rysowanie prawej krawędzi linii.
 - "Przesunięcie" parametrów, żeby móc narysować przedziałkę.
 - Rysowanie linii przedziałki od lewej.
 - Rysowanie linii przedziałki od prawej.
 - Dodanie ustalonego przesunięcia, o jakie przedziałki mają być względem siebie przesunięte, ustalone w konfiguratorze.
- Powrót na początek pętli i ewentualne rysowanie kolejnego poziomu linii.

```

# znalezienie najblitszego obiektu na klatce obrazu i na jego podstawie ustalenie jaki rodzaj
alertu ma zostac uruchomiony
for i in range(len(detections.xyxy)):
    if detections.xyxy[i, 3] > bottom_y_box_closest and detections.xyxy[
        i, 3] >= end_draw_lines_y + space_between_lines:
        bottom_y_box_closest = detections.xyxy[i, 3]
        x_left_box = detections.xyxy[i, 0]
        x_right_box = detections.xyxy[i, 2]

```

Powyższy fragment kodu pozwala na znalezienie najbliższego obiektu na klatce obrazu i na jego podstawie ustalić czy alert, i jaki rodzaj alertu ma zostać uruchomiony.

```

# znalezienie najblitszej przedzialki do tego obiektu i zapisanie danych tej przedzialki
min_distance = 2000
saved_line = ()
for element in middle_line_array:

```

```

if abs(element[0] - bottom_y_box_closest) < min_distance:
    min_distance = abs(element[0] - bottom_y_box_closest)
    saved_line = element

```

Powyższy fragment kodu pozwala na znalezienie najbliższej przedzialki do wykrytego najbliższego obiektu i zapisanie danych tej przedzialki.

```

# jesli wykryty obiekt znajduje sie w zakresie linii parkowania w osi X, alert zostanie
# uruchomiony
if saved_line[2] <= x_left_box <= saved_line[3] or saved_line[2] <=
    saved_line[3]:
    have_to_play = True

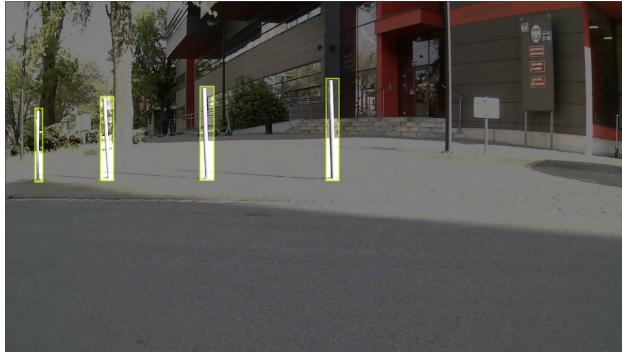
# na podstawie tego, ktory jest to numer przedzialki od dolu, zostanie uruchomiony jeden z
# 3 typow alertow - z największa częstotliwościa sygnału dźwiękowego gdy jest to jedna z
# linii z samego dolu, ze średnia, gdy jest to jedna linia z środka, z najmniejsza, gdy
# jest to jedna z górnich linii
if have_to_play:
    if not mixer.music.get_busy():
        if saved_line[1] == 0 or saved_line[1] == 1:
            mixer.music.load('sounds/beepRed.mp3')
        elif saved_line[1] == 2 or saved_line[1] == 3:
            mixer.music.load('sounds/beepOrange.mp3')
        else:
            mixer.music.load('sounds/beepGreen.mp3')
    mixer.music.play()

```

Powyższy fragment kodu pozwala na dźwiękowy alert informujący kierowcę o wykryciu obiektu w polu zaznaczonym przez linie parkowania. W zależności od odległości obiektu od pojazdu (obiekt znajduje się w danej strefie określonej kolorem linii) uruchamiany jest jeden z 3 różnych alertów - z największą częstotliwością sygnału dźwiękowego gdy obiekt jest najbliżej pojazdu (czerwona strefa), ze średnią gdy jest w środkowej - pomarańczowej strefie oraz z najmniejszą, gdy jest to w najdalszej, zielonej strefie.

3 Opis wykorzystanych modeli i metod przetwarzania obrazów

Początkowo używano biblioteki MobileNet-SSD. Nie obciążała mocno komputera - program działał płynnie nawet na CPU. Jednak nie wykrywała ona ścian ani słupków. Nie znaleziono także intuicyjnych metod na przetrenowanie tego modelu. Wtedy zdecydowano się na użycie modelu Ultralytics YOLOv8. Jest on częścią środowiska RoboFlow, które dostarcza stronę internetową ułatwiającą konfigurację i tworzenie własnych datasetów. Pozwala załadować zestaw zdjęć, na których potem można zaznaczać obiekty i ich klasy, które mają być wykrywane. Model jest pojedynczą konwolucyjną siecią neuronową złożoną z 24 warstw konwolucyjnych i dwóch o pełnych połączeniach. Obraz wejściowy przechodzi przez sieć tylko raz. Pozwala to przyspieszyć działanie modelu, lecz może skutkować niższą dokładnością detekcji i klasyfikacji. Do wytrenowania modelu wykorzystano własny dataset złożony z 69 zdjęć, które były zrzutami ekranu z nagranych materiałów testowych. Obiekty na zdjęciach zostały ręcznie sklasyfikowane zgodnie z założonymi klasami.



Rysunek 3: Przykładowe zaznaczenie klasy typu słupek na zbiorze trenującym

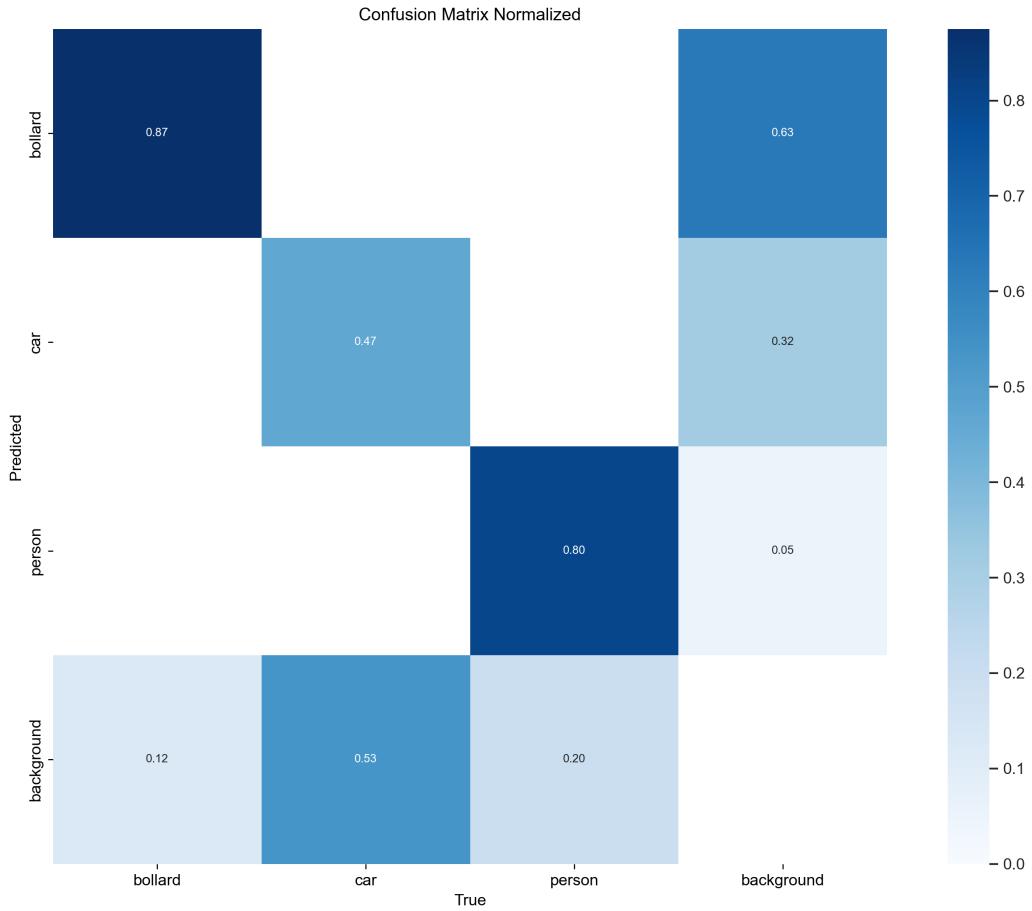
Podzielono dataset w taki sposób, że 78% zdjęć służyło do trenowania modelu, 14% do jego walidacji, a 7% do testowania (Stosunek 7:2:1). Dodatkowo, zdecydowano się na odbicie lustrzane niektórych zdjęć w celu poprawy działania modelu oraz automatyczną orientację i rozszerzenie zdjęć do rozdzielczości FullHD.

The screenshot shows the Roboflow dataset interface with the following details:

- Timestamp:** 2024-05-13 10:47pm
Generated on May 13, 2024
- Model Status:** This version doesn't have a model.
- Actions:** Export Dataset, Edit, ...
- Dataset Summary:** 69 Total Images, View All Images →
- Dataset Split:**
 - TRAIN SET: 54 Images (78%)
 - VALID SET: 10 Images (14%)
 - TEST SET: 5 Images (7%)
- Preprocessing:** Auto-Orient: Applied, Resize: Stretch to 1920x1080
- Augmentations:** Outputs per training example: 3, Flip: Horizontal

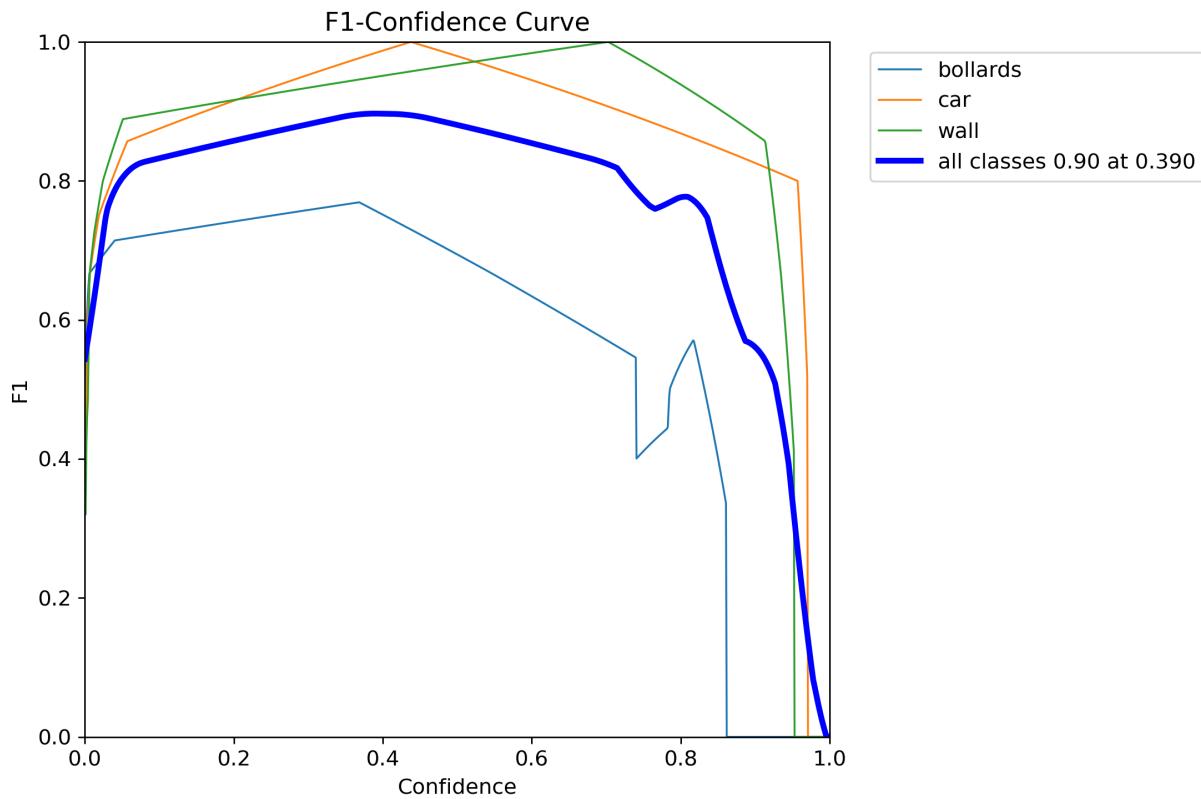
Rysunek 4: Parametry datasetu

Ze strony pobrano stworzony dataset, który następnie został użyty do wytrenowania modelu. W trakcie trenowania modelu wykonano 100 iteracji (epok).



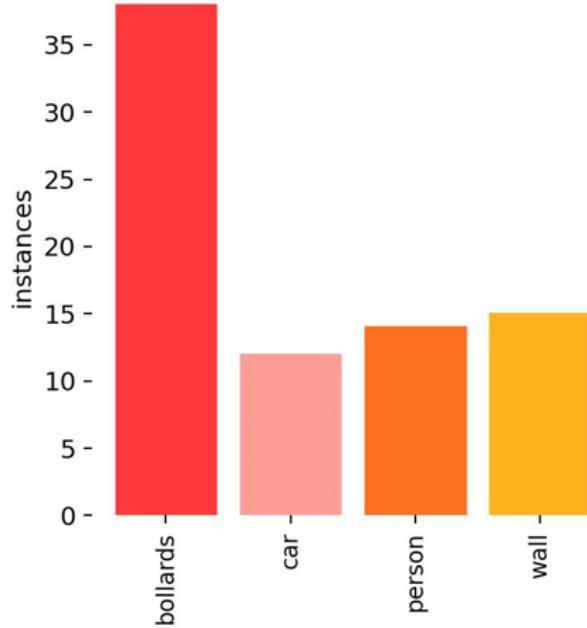
Rysunek 5: Wydajność klasyfikacji obiektów w postaci znormalizowanej macierzy typu "confusion"

Powyższy wykres, podsumowuje wydajność klasyfikacji obiektów. Jak widać, słupki i osoby były dosyć skutecznie wykrywane (w 80%), natomiast auta trochę mniej (w 47%), co jest widoczne w faktycznym działaniu programu.



Rysunek 6: Średnia harmoniczna precyzji i odwołania w różnych programach pewności wykrywania obiektów

Im wartość jest bliższa $y=1$ na powyższym wykresie, tym lepsza wydajność modelu. Można stwierdzić, że użyty model w projekcie nie jest perfekcyjny (choć takie nie istnieją), ale też nie najgorszy, ponieważ wartości y oscylują średnio w zakresach $[0.6;1]$ w przypadku aut i ścian. W przypadku słupków, jest nieco gorzej, oscylują w granicach $[0.4, 0.7]$. Co w efekcie daje wartości y w zakresach $[0.6, 0.9]$ dla całego modelu.



Rysunek 7: Ilość wykrytych obiektów danej klasy podczas trenowania

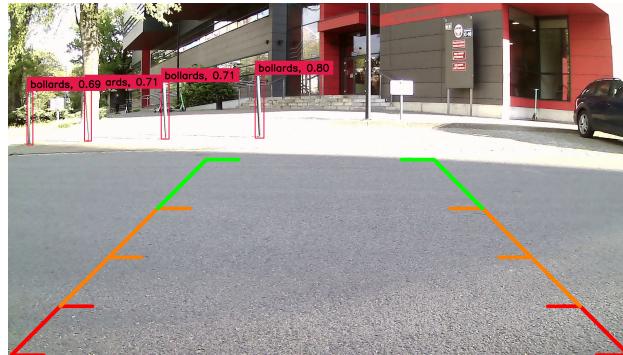
Najczęściej wykrywane były słupki, co nie jest zaskoczeniem, ponieważ na obrazkach zamieszczonych w datasetcie, w miejscu, gdzie nagrywane były materiały wideo, było ich najwięcej. W miejscu nagrywania materiałów wideo także była spora ilość samochodów osobowych, jednak skupiono się na wykrywaniu samochodów znajdujących się najbliżej kamery, dlatego nie wykryto ich zbyt wiele. Jeśli chodzi o pieszego, w materiałach testowych rozważano jedynie przypadek wykrycia jednego pieszego w dwóch różnych odległościach. Nawiązując do ścian, rozważono wykrywanie 3 rodzajów ścian, z czego prawdopodobnie wynika różnica.

4 Wyniki testów / eksperymentów

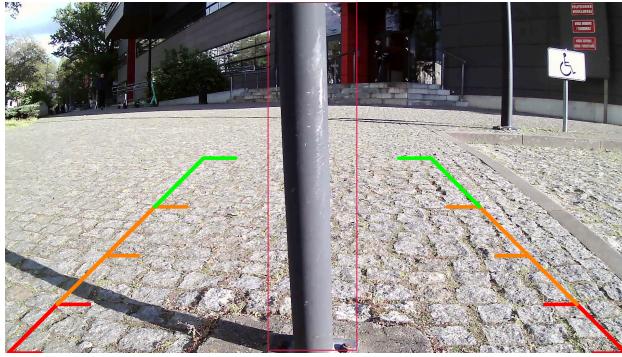
Testy wykonano na wymienionych w etapie nr 1 materiałach wideo oraz na żywo.

4.1 Podjechanie blisko, centralnie do słupka

Cele ustalone na początku zostały spełnione. Na materiale wideo [1], ciemnoszare słupki o wysokości 89.9 cm są wykrywane bez problemów.

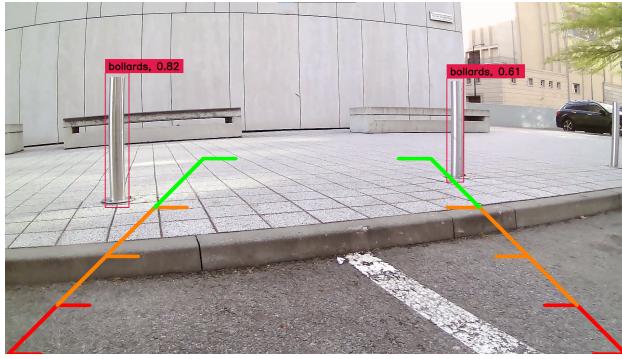


Rysunek 8: Wykrycie słupków ciemnoszarych z dalszej odległości

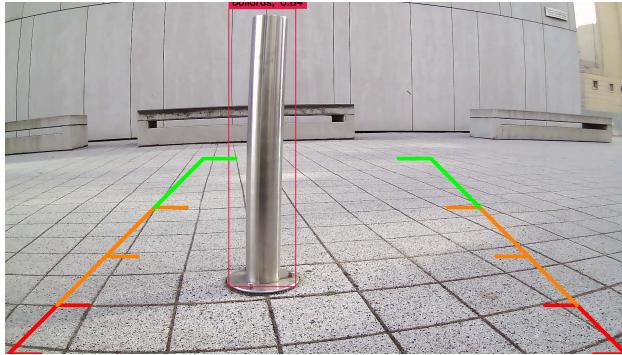


Rysunek 9: Wykrycie słupka ciemnoszarego z najbliższej odległości

Jeśli chodzi o srebrne słupki o wysokości 54.25 cm, również są wykrywane na materiale wideo [2], z drobnym, momentowym podwojonym wykryciem centralnego słupka, gdy pojazd znajduje się bliżej niego



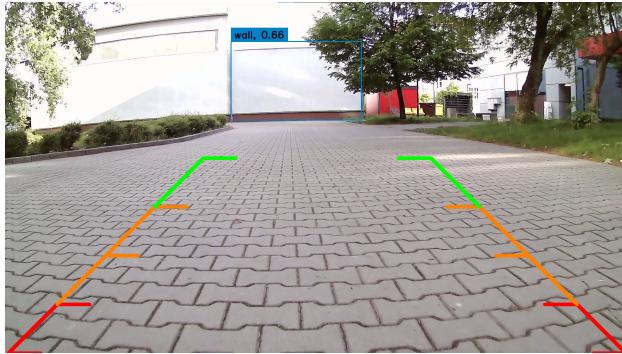
Rysunek 10: Wykrycie słupka srebrnego ze średniej odległości



Rysunek 11: Wykrycie słupka srebrnego z bliskiej odległości

4.2 Parkowanie przodem przed ścianą

Na materiale wideo [3] ściana jest wykrywana już z dalszej odległości.



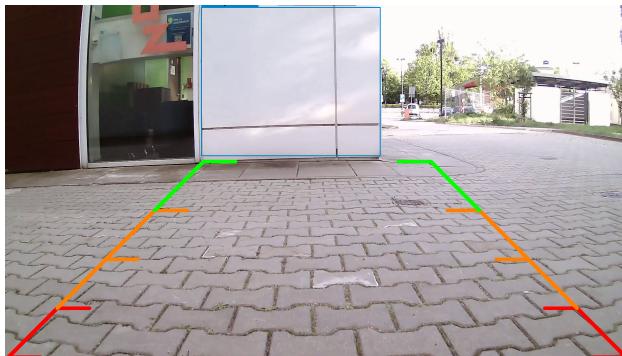
Rysunek 12: Wykrycie czystej, białej, jednolitej ściany z czerwonymi cegłami na dole

Na materiale [4], oba fragmenty ścian są dobrze wykrywane, jedynie przez ułamek sekundy, prawy fragment ściany jest wykrywany podwójnie.



Rysunek 13: Wykrycie czystej, białej, odrobinę chropowatej ściany z szarym chropowatym fragmentem na dole

W materiale [5], ściana w kształcie krzyża jest wykrywana na początku trzykrotnie, w czasie zbliżania się do niej, jest wykrywana podwójnie, po czym poprawnie pojedynczo. Choć zdarzało się do wielokrotnych wykryć ściany to za każdym razem była ona wykrywana, a przy bliższej odległości była wykrywana poprawnie.



Rysunek 14: Wykrycie białej ściany z charakterystycznym wcięciem w kształcie krzyża

4.3 Pieszy przechodzący przed samochodem

W materiałach [6] oraz [7] pieszy wykrywany jest zgodnie z założeniami.

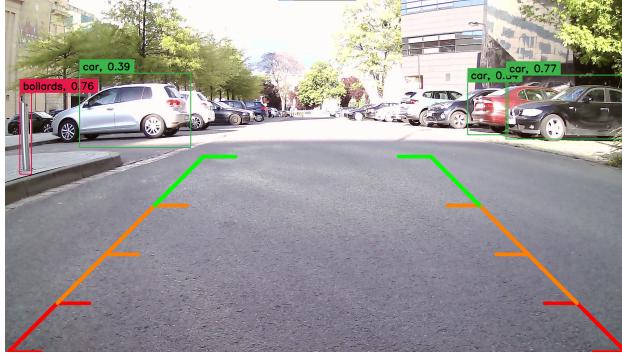


Rysunek 15: Wykrycie przechodnia przed samochodem

4.4 Zbliżanie się przodem do pojazdów

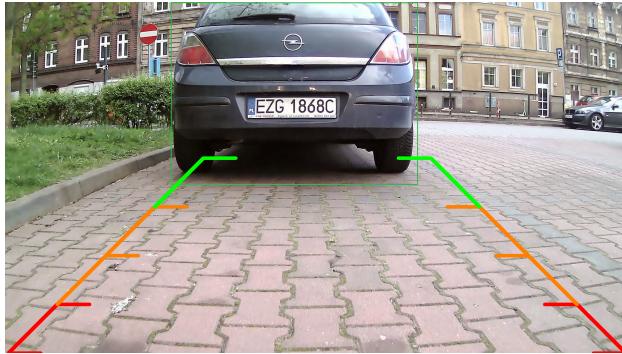
Cele ustalone w początkowej fazie projektu zostały zrealizowane. Na materiale wideo [7], są wykrywane następujące pojazdy:

- Volkswagen Golf VI koloru srebrnego
- Seat, koloru czerwonego
- BMW serii 1, koloru czarnego



Rysunek 16: Zbliżanie się przodem do pojazdów marki Volkswagen, Seat, BMW

W materiale [8], Opel Astra koloru czarnego jest stale wykrywany jako niebezpieczny obiekt, niestety jest czasami mylona ze ścianą, ale także jak w przypadku ścian, użytkownik jest świadomy o niebezpiecznym obiekcie znajdującym się przed nim.



Rysunek 17: Zbliżanie się przodem do pojazdu Opel Astra

W materiale [11], obiekt jest wykrywany dopiero z mniejszej odległości (Toyoty Yaris koloru białego). Jeśli chodzi o wykrywaną klasę obiektu, sytuacja jest podobna jak w przypadku z filmem [8]



Rysunek 18: Zbliżanie się przodem do pojazdu Toyota Yaris

4.5 Testy aplikacji konfiguracyjnej pod względem wprowadzania parametrów niezgodnych z założeniami projektowymi

Program nie uruchomi się jeśli:

- W pole/pola tekstowe zostaną wprowadzone jakiekolwiek znaki niebędące cyframi
- Jeśli nie będzie żadnej podłączonej kamery do komputera
- Wartość znajdująca się w 1. polu ”Crop in x” lub ”Crop in y” będzie większa od tej znajdującej się w 2. polu, bądź wartość w jakimkolwiek z tych pól będzie ujemna

4.6 Testy na żywo

Wykonano również testy na żywo z użyciem kamerki. System działał w pełni poprawnie, zgodnie z założeniami. Screen Capture Nagranie z telefonu



Rysunek 19: Testy na żywo

5 Wnioski z projektu

5.1 Dotyczące implementacji architektury niskopoziomowej

Aby zaimplementować architekturę niskopoziomową, wystarczy następujący przykładowy zestaw narzędzi:

- laptop - do uruchomienia oprogramowania detekcyjnego
- kamera internetowa - do przechwytywania obrazu rzeczywistego

Projekt nie wymaga wielkich nakładów finansowych, żeby osiągnąć działający produkt. Możliwe jest jednak rozbudowanie projektu. Pierwotnym pomysłem było użycie systemu kamer stereoskopowych które pozwoliłyby uzyskać trójwymiarowy obraz otoczenia i zapewnić o wiele dokładniejszy pomiar odległości od obiektów. Wiążałoby się to jednak ze znacznymi kosztami. Warto wspomnieć o dużej wadze akceleracji GPU dla płynnego działania systemu. Warto również zastanowić się nad stosownością użycia sieci neuronowych do określania odległości od obiektów. Projekt realizuje prototyp systemu bezpieczeństwa. Istnieją rozwiązania działające w bardziej zróżnicowanych warunkach zewnętrzny oraz dostarczające dokładniejsze dane o środowisku np. radar, LiDAR, czy zwykły czujnik ultradźwiękowy.

5.2 Dotyczące tworzenia dataset'u

Strona RoboFlow pozwala w intuicyjny sposób przygotować dataset - zaznaczyć prostokątem na obrazie poszczególne obiekty, które mają zostać wykryte. Następnie, jest możliwość pobrania tak przygotowanego

zestawu obrazów, na podstawie którego można wytrenować model. Wywnioskować można, że w dzisiejszych czasach mamy do dyspozycji mnogość gotowych narzędzi, które są bardzo intuicyjne, które ułatwiają nam osiąganie pewnych celów.

Istnieją również inne sposoby zaznaczania obiektów - za pomocą wielokątów oraz automatyczne systemy wspomagania kategoryzacji.

W tym projekcie użyto stosunkowo małego datasetu (69 zdjęć), ponieważ został on stworzony specjalnie na potrzeby tego projektu. Dla lepszego efektu wykrywania obiektów można by stworzyć większy zbiór obrazów.

5.3 Dotyczące trenowania modelu i samego modelu

W pracach nad projektem spodziewano się, że trenowanie modelu na każdym komputerze da takie same efekty, jeśli zostanie zastosowany ten sam model oraz parametry trenowania. Natomiast otrzymano w efekcie dwa różnie wykrywające obiekty modele. Pozwoliło to na lepsze zrozumienie sposobu działania sieci neuronowych i ich stochastycznej natury.

Jeśli chodzi o sam model, najbardziej złożonym problemem, okazało się wykrycie ściany. Spodziewano się, że nie będzie to aż takie trudne, jednak wykrywanie ściany okazało się najmniej skuteczne, prawdopodobnie z tego powodu, że posiada mało charakterystycznych punktów w porównaniu do człowieka, czy słupka. Warto by rozważyć inne sposoby detekcji ścian, np. poprzez wykrywanie obszarów o jednolitym kolorze.

5.4 Dotyczące implementacji kodu

Szacowanie odległości oraz sygnalizowanie użytkownika o zbliżaniu się do obiektu opierało się na geometrii analitycznej, działaniach w dwuwymiarowym układzie współrzędnych i przetwarzaniu zbioru pikseli przechwyconej klatki wideo. Największym wyzwaniem okazało się wykrywanie odległości od obiektu - ustalenie najbliższego obiektu, czy obiekt znajduje się w obrębie linii parkowania i jaki sygnał dźwiękowy uruchomić. Trzeba było zrozumieć w jaki sposób są przechowywane dane o wykryciu obiektów w bibliotece i jak je wyłuskać. Ważnym zadaniem było również postawienie odpowiednich ograniczeń. Samo wykrywanie klas obiektów i zaznaczanie ich na ekranie było o wiele prostsze niż przewidywano - używana biblioteka Ultralytics posiada gotowe metody do wykonania tych zadań. Rysowanie linii wymagało chwili zastanowienia, by stworzyć odpowiedni efekt, ale nie stanowiło wielkiego wyzwania.

Podczas projektu zauważone, że konfiguracja środowiska oraz instalacja narzędzi może stanowić znaczący problem przy przenoszeniu oprogramowania na inne urządzenie. Rozwiązaniem tego problemu byłoby zastosowanie konteneryzacji za pomocą np. Dockera. Nie zastosowano tego rozwiązania ze względu na duże problemy z instalacją odpowiednich sterowników akceleratora CUDA w systemie Linux.

5.5 Wnioski ogólne

Projekt pozwolił zauważać jak wiele zastosowań ma widzenie komputerowe oraz neuronowe. Za pomocą gotowych narzędzi oraz bibliotek można stworzyć w akceptowalnym czasie działające narzędzia. Projekt pozwolił również znacząco poszerzyć wiedzę o sposobie działania oraz użytkowaniu neuronowych sieci konwolucyjnych oraz projektowaniu systemów zgodnie z założeniami klienta.

6 Dokumentacja użytkownika (instrukcja obsługi programu)

6.1 Metody uruchomienia programu

Aby program zadziałał, wymagana jest dedykowana karta graficzna serii NVidia oraz Python w wersji 3.11. Oprócz tego należy kolejno zainstalować narzędziem "pip":

1. ultralytics (pip install ultralytics)
2. supervision w wersji 0.3.0 (pip install supervision==0.3.0)
3. opencv-python (pip install opencv-python)

Następnie należy zainstalować narzędzie CUDA w wersji 12.1.0 oraz CUDNN w wersji 8.9.7 dla CUDA 12.x. Ostatnim krokiem jest instalacja biblioteki pytorch komendą:

```
pip3 install --pre torch torchvision torchaudio --index-url  
https://download.pytorch.org/whl/nightly/cu121
```

Po wykonaniu wymienionych wyżej czynności, można uruchomić program w dowolnym środowisku programistycznym, w którym można programować w języku Python (zalecany PyCharm). Inną opcją jest wpisanie komendy w terminalu, który wskazuje ścieżkę na folder projektu:

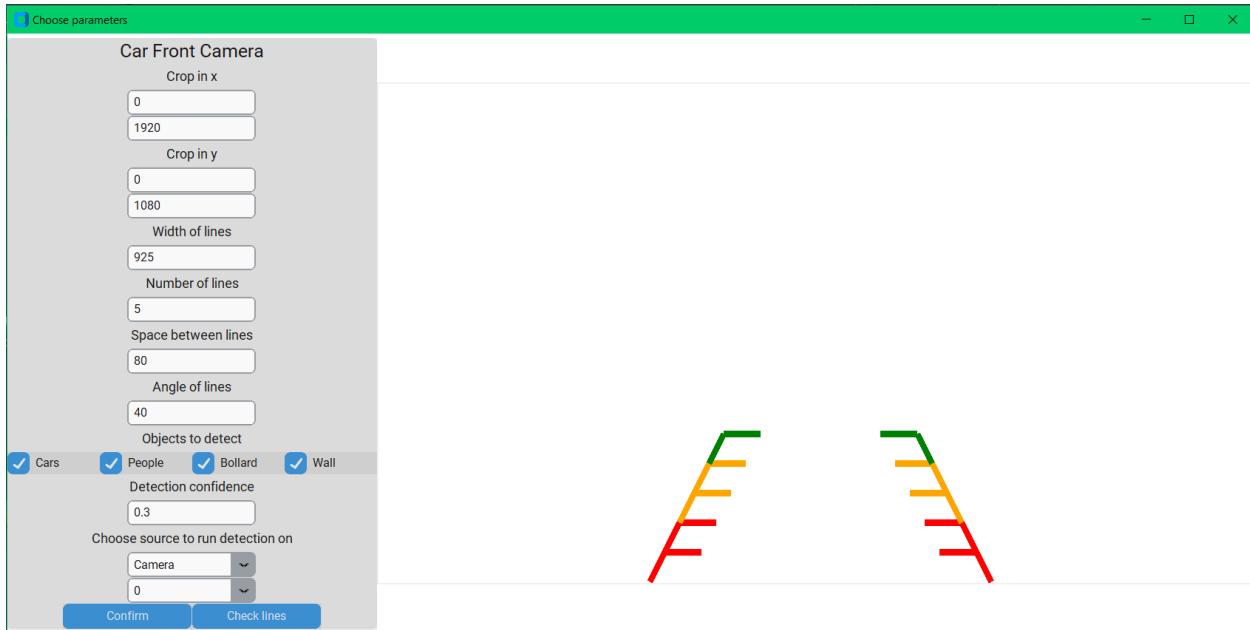
```
py main_with_interface.py
```

Ewentualnie:

```
python3 main_with_interface.py
```

6.2 Sposób obsługi interfejsu użytkownika

Na początku na ekranie ukazuje się konfigurator:



Rysunek 20: Ekran konfiguratora

- Kadrowanie obrazu - ustalenie, jaki fragment obrazu ma być widoczny. Ustawienie zakresu w osi x ("Crop in x") oraz y ("Crop in y"): wartości x i y muszą się zawierać w przedziale rozdzielczości kamery (w pikselach) oraz wartość znajdująca się w pierwszym polu musi być mniejsza bądź równa od tej znajdującej się w drugim polu zarówno dla x, jak i y (domyślne wartości: w przypadku filmu/kamery o rozdzielczości FHD, obejmuje całą szerokość i wysokość obrazu)
- Szerokość linii ("Width of lines") - określa na jakiej szerokości obrazu mają być rysowane linie. Jest to szerokość pomiędzy lewym skrajnym punktem miejsca rozpoczęcia rysowania linii (od dołu) od prawego skrajnego punktu miejsca rozpoczęcia rysowania linii (od dołu) (w pikselach), wartość nie może przekroczyć szerokości klatki obrazu (domyślna, zalecana wartość: 925)

- Liczba linii ("Number of lines") - określa ilość przedziałek liniowych na całej długości linii parkowania (domyślana, zalecana wartość: 5)
- Odstęp między liniami ("Space between lines") - określa odstęp, jaki ma być pomiędzy poszczególnymi przedziałkami (w pikselach) (domyślana, zalecana wartość - 150)
- Kąt nachylenia linii ("Angle of lines") - określa jak bardzo mają być pochylone linie parkowania na ekranie, to znaczy jaki ma być odstęp między kolejnymi przedziałkami w osi x (w pikselach) (domyślana, zalecana wartość: 110)
- Obiekty do wykrycia ("Objects to detect") - zaznaczenie, które obiekty mają być wykrywane
- Pewność wykrycia ("Detection confidence") - ustalenie minimalnej pewności, od której obiekty mają zostać wykryte - wartość [0,1]
- Wybór wejścia do programu ("Choose source to run detection on") - pierwsze pole pozwala wybrać kamerę albo wideo. Drugie natomiast w przypadku kamery - wybrać numer kamery; a w przypadku wideo, pole to nie ma znaczenia.

Naciskając przycisk "Check lines" można sprawdzić, jak linie będą się prezentowały przed rozpoczęciem detekcji. Można w ten sposób dostosować parametry pod własne wymagania.

Po kliknięciu "Confirm", w przypadku wyboru detekcji na materiale wideo - pozwoli wybrać materiał wideo z komputera i po jego załadowaniu, detekcja rozpocznie się. Natomiast w przypadku wyboru detekcji na kamerze internetowej - po kliknięciu "Confirm", detekcja rozpocznie się od razu.

6.3 Interpretacja działania programu

Program zaznacza klasy obiektów na ekranie obwodem w postaci prostokąta. W lewym górnym rogu prostokąta wyświetlna jest klasa wykrytego obiektu oraz pewność, jaką ma program, że jest to obiekt danej klasy. Czym liczba będzie bliższa 1, tym bardziej jest prawdopodobne, że obiekt należy danej klasie. Kiedy prostokąt (wykryty obiekt) będzie znajdował się w obrębie zielonych przedziałek linii parkowania, zostanie uruchomiony sygnał dźwiękowy o małej częstotliwości, co będzie oznaczać, że użytkownik zbliża się do niebezpiecznego obiektu. Gdy znajdzie się w obrębie pomarańczowych linii - sygnał o średniej częstotliwości, co oznacza, że użytkownik jest już bardzo blisko danego obiektu. A gdy w obrębie czerwonych przedziałek - sygnał dźwiękowy stały, użytkownik nie powinien podjeżdżać bliżej do obiektu.

7 Literatura

- Poradnik trenowania modelu RoboFlow - Poradnik RoboFlow
- Artykuł naukowy twórców modelu YOLO - You Only Look Once: Unified, Real-Time Object Detection
- Link do repozytorium w serwisie GitHub - Repozytorium GitHub
- Testowe materiały wideo:

- [1] 2024-04-17 17-43-43.mp4
- [2] 2024-04-17 17-42-13.mp4
- [3] 2024-04-17 17-46-28.mp4
- [4] 2024-04-17 17-47-09.mp4
- [5] 2024-04-17 17-48-47.mp4
- [6] 2024-04-17 17-59-08.mp4, 2024-04-17 17-58-19.mp4
- [7] 2024-04-17 17-42-13.mp4
- [8] 2024-04-17 17-54-36.mp4
- [9] 2024-04-17 17-42-13.mp4
- [10] 2024-04-17 17-42-13.mp4
- [11] 2024-04-17 17-55-37.mp4

- Screencasty