



Politechnika  
Śląska

## PROJEKT INŻYNIERSKI

System wizyjny dla robota mobilnego

Łukasz GRABARSKI

Nr albumu: 300434

**Kierunek:** Automatyka i Robotyka

**Specjalność:** Technologie Informacyjne

**PROWADZĄCY PRACĘ**

dr inż. Krzysztof Jaskot

**KATEDRA** Automatyki i Robotyki

**Wydział** Automatyki, Elektroniki i Informatyki

Gliwice 2025



**Tytuł pracy**

System wizyjny dla robota mobilnego

**Streszczenie**

(Streszczenie pracy – odpowiednie pole w systemie APD powinno zawierać kopię tego streszczenia.)

**Słowa kluczowe**

RaspberryPi, Python, OpenCV, YOLO, Binarzacja

**Thesis title**

Vision system for a mobile robot

**Abstract**

(Thesis abstract – to be copied into an appropriate field during an electronic submission – in English.)

**Key words**

RaspberryPi, Python, OpenCV, YOLO, Binarization



# Spis treści

<b>1 Wstęp</b>	<b>1</b>
1.1 Wprowadzenie w problem . . . . .	2
1.2 Osadzenie problemu w dziedzinie . . . . .	3
1.3 Cel pracy . . . . .	3
1.4 Zakres pracy . . . . .	4
1.5 Struktura pracy . . . . .	4
1.6 Wkład własny autora . . . . .	5
<b>2 Analiza tematu i przegląd literatury</b>	<b>7</b>
2.1 Sformułowanie problemu . . . . .	7
2.2 Stan wiedzy i osadzenie w kontekście aktualnych badań . . . . .	8
2.3 Przetwarzanie obrazu i algorytmy detekcji obiektów . . . . .	9
2.3.1 Algorytmy głębokiego uczenia . . . . .	9
2.3.2 Klasyczne metody przetwarzania obrazu . . . . .	9
2.3.3 Porównanie metod . . . . .	9
2.4 Sieci neuronowe i ich uczenie . . . . .	10
2.4.1 Budowa sieci neuronowej . . . . .	10
2.4.2 Architektura sieci . . . . .	11
2.4.3 Proces uczenia sieci neuronowej . . . . .	11
2.4.4 Zasada działania . . . . .	11
2.4.5 Architektura sieci . . . . .	12
2.4.6 Zastosowania w robotyce mobilnej . . . . .	12
2.5 Binaryzacja . . . . .	12
2.5.1 Zasada działania . . . . .	13
2.5.2 Zastosowania . . . . .	13
2.6 Studia literaturowe i znane rozwiązania . . . . .	13
2.6.1 System programowania i sterowania robota mobilnego . . . . .	13
2.6.2 Aktywny system wizyjny dla robota kroczącego . . . . .	14
2.6.3 System wspomagania nawigacji osób niewidomych . . . . .	15

<b>3 Przygotowanie i trening modelu detekcji obiektów</b>	<b>17</b>
3.1 Przygotowanie danych wzorcowych . . . . .	17
3.2 Platforma Roboflow . . . . .	18
3.2.1 Proces użycia witryny Roboflow . . . . .	19
3.2.2 Trening zestawu danych . . . . .	21
3.3 Trening modelu YOLOv7 . . . . .	23
3.3.1 Przygotowanie środowiska . . . . .	23
3.3.2 Instalacja wymaganych bibliotek . . . . .	23
3.3.3 Pobranie i przygotowanie danych treningowych . . . . .	23
3.3.4 Przygotowanie modelu . . . . .	24
3.3.5 Trening modelu . . . . .	24
<b>4 Proces tworzenia robota mobilnego</b>	<b>25</b>
4.1 Założenia logiczne . . . . .	25
4.2 Konstrukcja robota mobilnego . . . . .	27
4.2.1 Wykorzystane moduły . . . . .	27
4.2.2 Schemat elektryczny . . . . .	29
4.2.3 Projekt robota mobilnego . . . . .	31
4.3 Dostosowanie RaspberryPi . . . . .	34
4.3.1 Spis wymaganych bibliotek . . . . .	35
<b>5 Analiza działania systemu wizyjnego</b>	<b>37</b>
5.1 Specyfikacja sprzętowa . . . . .	38
5.2 Detekcja obiektu za pomocą narzędzi głębokiego uczenia . . . . .	38
5.2.1 Opis działania programu . . . . .	39
5.2.2 Uzyskane rezultaty . . . . .	44
5.3 Detekcja obiektu za pomocą binaryzacji i klasycznych technik przetwarzania obrazów . . . . .	48
5.3.1 Opis działania programu . . . . .	48
5.3.2 Uzyskane rezultaty . . . . .	50
5.4 Porównanie skuteczności algorytmów . . . . .	54
5.4.1 Charakterystyka modelu YOLOv7 . . . . .	55
5.4.2 Charakterystyka modelu YOLOv8 . . . . .	57
5.4.3 Charakterystyka algorytmu binaryzacji . . . . .	59
5.4.4 Wnioski . . . . .	61
<b>6 Problematyka i rozwiązania</b>	<b>63</b>
6.1 Wybór algorytmów detekcji obiektów . . . . .	63
6.2 Implementacja algorytmów śledzenia obiektu . . . . .	64
6.3 Problemy związane z elektroniką i zasilaniem . . . . .	65

6.4 Problemy związane z mechaniką i sterowaniem . . . . .	66
<b>7 Podsumowanie i wnioski</b>	<b>69</b>
7.1 Wnioski . . . . .	69
7.2 Perspektywy dalszych badań . . . . .	70
<b>Bibliografia</b>	<b>71</b>
<b>Spis skrótów i symboli</b>	<b>75</b>
<b>Źródła</b>	<b>77</b>
<b>Lista dodatkowych plików, uzupełniających tekst pracy</b>	<b>79</b>
<b>Spis rysunków</b>	<b>82</b>
<b>Spis tabel</b>	<b>83</b>



# Rozdział 1

## Wstęp

Rozwój technologii komputerowej oraz metod sztucznej inteligencji w ostatnich dekadach znacząco zmienił podejście do projektowania i implementacji systemów wizyjnych. Szczególnie istotne w tym kontekście stały się algorytmy głębokiego uczenia, które umożliwiły precyzyjne rozpoznawanie i analizę obrazów w czasie rzeczywistym. W połączeniu z dostępnością wydajnych i ekonomicznych platform obliczeniowych, takich jak Raspberry Pi, rozwiązania te znajdują szerokie zastosowanie w robotyce mobilnej.

Niniejsza praca koncentruje się na opracowaniu systemu wizyjnego dla robota mobilnego z wykorzystaniem algorytmów YOLOv7, biblioteki PyTorch oraz OpenCV. System ma umożliwić detekcję i śledzenie obiektów w czasie rzeczywistym, co stanowi kluczowy element autonomicznego działania robota w dynamicznym środowisku. Dodatkowo, praca poddaje analizie porównawczej skuteczność algorytmów głębokiego uczenia (YOLOv7, YOLOv8) z podstawowymi metodami przetwarzania obrazów, takimi jak binaryzacja.

Przedstawione zostanie wprowadzenie do przygotowania modelu YOLO, wytrenowania sieci oraz przygotowania środowiska wykonawczego na platformie Raspberry Pi. Ponadto, wykazane zostaną testy działania algorytmów w różnych warunkach środowiskowych i na różnych platformach sprzętowych, co umożliwi ocenę kompromisów między wydajnością a dokładnością w kontekście specyficznych ograniczeń sprzętowych.

## 1.1 Wprowadzenie w problem

Robotyka mobilna jest jedną z najszybciej rozwijających się dziedzin technologii, znajdującej zastosowanie zarówno w przemyśle, jak i w gospodarstwach domowych. Dzięki zastosowaniu systemów wizyjnych, roboty mobilne zyskują zdolność interakcji z otoczeniem, co znacząco zwiększa ich autonomię i zakres funkcjonalności. Systemy te umożliwiają identyfikację obiektów, analizę ich ruchu, a także podejmowanie decyzji w czasie rzeczywistym, co otwiera drogę do zastosowań w dynamicznych środowiskach, takich jak logistyka, rolnictwo czy ratownictwo.

W ostatnich latach dynamiczny rozwój algorytmów sztucznej inteligencji, takich jak głębokie uczenie (ang. *Deep Learning*), oraz dostępność platform sprzętowych, takich jak ekonomiczne Raspberry Pi czy wydajne NVIDIA Jetson, umożliwiły implementację złożonych systemów wizyjnych. Algorytmy te pozwalają na wykrywanie, klasyfikację i śledzenie obiektów w czasie rzeczywistym, co czyni je kluczowymi elementami nowoczesnych systemów autonomicznych.

Jednym z najbardziej efektywnych algorytmów detekcji obiektów jest YOLO (ang. *You Only Look Once*), który wyróżnia się wysoką szybkością działania i zdolnością do pracy na urządzeniach o ograniczonej mocy obliczeniowej. W niniejszej pracy szczególną uwagę poświęcono implementacji algorytmów YOLOv7 i YOLOv8 oraz ich porównaniu z klasycznymi metodami przetwarzania obrazu, takimi jak binaryzacja. Takie podejście pozwala na ocenę kompromisów między skutecznością, wydajnością i wymaganiami sprzętowymi.

Realizacja systemu wizyjnego obejmuje nie tylko wdrożenie algorytmów, ale również ich integrację z modułem sterowania robota mobilnego. W pracy przedstawiono także przygotowanie środowiska wykonawczego na platformie Raspberry Pi, obejmujące konfigurację sprzętową oraz testowanie algorytmów w różnych warunkach środowiskowych i na różnych platformach sprzętowych. Wyniki tych badań pozwalają na identyfikację najbardziej efektywnych rozwiązań w kontekście ograniczeń wynikających z zastosowanego sprzętu.

W niniejszej pracy zaprezentowano kompleksowe podejście do projektowania systemu wizyjnego dla robota mobilnego, obejmujące przygotowanie modelu YOLO, trening sieci neuronowej, konfigurację środowiska na platformie Raspberry Pi oraz analizę porównawczą wyników działania algorytmów w różnych warunkach.

## 1.2 Osadzenie problemu w dziedzinie

Identyfikacja i śledzenie obiektów w czasie rzeczywistym stanowią jedno z największych wyzwań w robotyce mobilnej. Dzięki zaawansowanym metodom przetwarzania obrazu roboty nie tylko „widzą” otoczenie, ale również potrafią je analizować i podejmować decyzje w oparciu o dane wizualne. Tego rodzaju zdolności są niezbędne w systemach autonomicznych, które muszą funkcjonować w zróżnicowanych, dynamicznych i często nieprzewidywalnych środowiskach.

Kluczowym aspektem w rozwoju takich systemów jest opracowanie algorytmów wizyjnych, które zapewniają wysoką precyzyję detekcji i klasyfikacji obiektów, jednocześnie spełniając wymagania czasowe niezbędne do działania w czasie rzeczywistym. Wyzwaniem jest dostosowanie tych algorytmów do pracy na platformach o ograniczonych zasobach obliczeniowych, takich jak Raspberry Pi.

Zastosowania systemów wizyjnych wykraczają jednak poza ziemskie środowiska. Szczerogólnym przykładem są misje kosmiczne, w których autonomiczne roboty, takie jak łaziki marsjańskie, wykorzystują zaawansowane systemy wizyjne do eksploracji nieznanych terenów. Łaziki takie jak Perseverance czy Curiosity stosują algorytmy analizy obrazu do identyfikacji przeszkód, planowania trasy oraz analizy powierzchni Marsa. Wyzwania związane z pracą w ekstremalnych warunkach oraz przy ograniczonej dostępności zasobów obliczeniowych stanowią inspirację dla dalszego rozwoju technologii, które znajdują zastosowanie również na Ziemi.

Ponadto integracja systemu wizyjnego z modułami sterowania i nawigacji robota mobilnego wymaga opracowania odpowiednich rozwiązań programistycznych, które umożliwiają precyzyjne przekładanie informacji wizualnych na decyzje ruchowe robota. Takie podejście jest szczególnie istotne w zastosowaniach przemysłowych, gdzie systemy wizyjne mogą wspierać procesy takie jak nawigacja w magazynach, inspekcja obiektów czy wspomaganie operacji ratunkowych.

## 1.3 Cel pracy

Celem pracy jest opracowanie systemu wizyjnego dla robota mobilnego, który umożliwi identyfikację różnego rodzaju obiektów, śledzenie ich ruchu oraz podążanie za nimi. W szczególności praca koncentruje się na:

- Implementacji algorytmów YOLOv7, YOLOv8 oraz binaryzacji na platformie Raspberry Pi 4B.
- Integracji systemu wizyjnego z układem sterowania robota mobilnego.
- Analizie wydajności systemu oraz ocenie jego funkcjonalności w warunkach rzeczywistych.

## 1.4 Zakres pracy

Zakres pracy obejmuje:

- Analizę literatury i przegląd aktualnego stanu wiedzy w dziedzinie systemów wizyjnych dla robotów mobilnych.
- Projektowanie i implementację systemu wizyjnego, wykorzystując platformy takie jak Roboflow i Google Colab do przygotowania danych oraz treningu modeli YOLO.
- Integrację algorytmów detekcji z systemem sterowania robota.
- Testowanie systemu w różnych warunkach środowiskowych oraz analizę uzyskanych wyników.

## 1.5 Struktura pracy

Praca została podzielona na następujące rozdziały:

- Rozdział pierwszy – Wstęp: Wprowadzenie w problematykę, cel pracy, zakres oraz strukturę dokumentu.
- Rozdział drugi – Analiza tematu: Przegląd literatury i istniejących rozwiązań w dziedzinie systemów wizyjnych.
- Rozdział trzeci – Przygotowanie danych i trening modeli: Opis przygotowania danych oraz procesu treningu modeli YOLO.
- Rozdział czwarty – Proces tworzenia robota mobilnego: Szczegóły konstrukcji robota oraz integracji systemów.
- Rozdział piąty – Analiza działania systemu wizyjnego: Wyniki testów i ich analiza.
- Rozdział szósty – Problematyka i rozwiązania: Omówienie napotkanych wyzwań oraz zastosowanych rozwiązań.
- Rozdział siódmy – Podsumowanie i wnioski: Podsumowanie wyników pracy oraz propozycje dalszych badań.

## 1.6 Wkład własny autora

W ramach niniejszej pracy autor:

- Przygotował zestaw danych treningowych zawierający obrazy obiektu w różnych warunkach oświetleniowych.
- Przeprowadził trening modeli YOLOv7 i YOLOv8 na platformie Google Colab.
- Samodzielnie zaimplementował system wizyjny na bazie modelów YOLO oraz algorytmu binaryzacji.
- Zaprojektowała i skonstruował robota mobilnego, integrując system wizyjny z jego modułem sterowania.
- Przeprowadził testy i analizę działania systemu w rzeczywistych warunkach operacyjnych.



# Rozdział 2

## Analiza tematu i przegląd literatury

### 2.1 Sformułowanie problemu

Współczesna robotyka mobilna stawia liczne wyzwania związane z autonomicznym działaniem robotów w dynamicznych i złożonych środowiskach. Jednym z kluczowych aspektów umożliwiających taką autonomię jest detekcja i śledzenie obiektów w czasie rzeczywistym, co pozwala robotowi na skutecną interakcję z otoczeniem oraz podejmowanie decyzji na podstawie danych wizualnych. Wyzwania te stają się szczególnie istotne w warunkach zmennego oświetlenia, ograniczonej mocy obliczeniowej oraz konieczności integracji zaawansowanych algorytmów wizji komputerowej z systemami sterowania.

Systemy wizyjne odgrywają kluczową rolę w procesie autonomizacji robotów, zapewniając możliwość:

- wykrywania przeszkód oraz nawigacji w środowisku,
- identyfikacji i śledzenia wybranych obiektów,
- bieżącej analizy otoczenia w czasie rzeczywistym.

Te funkcjonalności umożliwiają robotom podejmowanie działań adaptacyjnych w zmieniających się i nieprzewidywalnych warunkach środowiskowych. Wymaga to jednak zastosowania zaawansowanych algorytmów detekcji i śledzenia, które są w stanie działać efektywnie, mimo ograniczonych zasobów sprzętowych dostępnych na platformach takich jak Raspberry Pi.

Niniejsza praca koncentruje się na analizie i implementacji systemów wizyjnych opartych na algorytmach głębokiego uczenia, takich jak YOLO (*You Only Look Once*), oraz na podstawowych technikach przetwarzania obrazów, takich jak binaryzacja. Systemy te zostały zaimplementowane na platformie Raspberry Pi, co pozwala na praktyczne przetestowanie ich efektywności i dostosowanie do rzeczywistych ograniczeń sprzętowych.

## 2.2 Stan wiedzy i osadzenie w kontekście aktualnych badań

Współczesne systemy wizyjne dla robotów mobilnych stawiają sobie za cel umożliwienie robotom autonomicznego działania w dynamicznych środowiskach poprzez analizę obrazu w czasie rzeczywistym. Znaczenie takich systemów podkreślają badania [1], w których przedstawiono system wizyjny umożliwiający robotom nie tylko detekcję i śledzenie ludzi, ale również rozpoznawanie gestów i mimiki twarzy w czasie rzeczywistym. Tego rodzaju podejście znajduje zastosowanie w interakcji człowiek-robot, gdzie wizualne dane są kluczowym medium komunikacji niewerbalnej.

W pracy Aoki i in. [1] zaproponowano trójfazowy system detekcji: detekcję człowieka na podstawie ruchu, śledzenie obiektów na podstawie koloru skóry i odzieży oraz rozpoznawanie mimiki i gestów twarzy. Dzięki zastosowaniu kamer wielokierunkowych oraz technologii przetwarzania w czasie rzeczywistym, robot BUGNOID był w stanie nie tylko zlokalizować ludzi w swoim otoczeniu, ale także rozpoznać gesty, takie jak kiwanie głową (oznaczające „tak”) czy kręcenie głową („nie”). Rozwiążanie to otwiera drogę do bardziej zaawansowanej współpracy między człowiekiem a robotem w środowiskach rzeczywistych.

Podobne wyzwania zostały zidentyfikowane w misjach marsjańskich, gdzie autonomiczne łaziki, takie jak Spirit i Opportunity, korzystały z zaawansowanych systemów wizyjnych do eksploracji powierzchni Marsa. W badaniach Matthies i in. [6] omówiono zastosowanie algorytmów stereo vision, odometrii wizualnej oraz detekcji przeszkód w misji Mars Exploration Rover (MER). Algorytmy te umożliwiły łazikom nawigację w trudnym terenie, analizę rzeźby powierzchni oraz precyzyjne lądowanie przy użyciu systemu Descent Image Motion Estimation System (DIMES). W szczególności, stereo vision pozwoliło na tworzenie trójwymiarowych map terenu, co było kluczowe dla identyfikacji przeszkód i planowania trasy.

Implementacja takich rozwiązań w systemach kosmicznych wymagała optymalizacji algorytmów pod kątem ograniczonych zasobów obliczeniowych. Przykładem jest użycie procesora RAD6000 w łazikach Spirit i Opportunity, o częstotliwości zaledwie 20 MHz, co wymusiło zastosowanie uproszczonych, ale wydajnych algorytmów przetwarzania obrazu [6]. Te doświadczenia pokazały, że technologie opracowane dla eksploracji kosmosu mogą być z powodzeniem adaptowane do systemów wizyjnych dla robotów mobilnych na Ziemi.

Dla robotów mobilnych działających w dynamicznych środowiskach ziemskich kluczowe jest opracowanie algorytmów, które mogą funkcjonować w czasie rzeczywistym przy jednoczesnym dostosowaniu do ograniczeń sprzętowych. Wspomniane badania [1] [6] wskazują na interdyscyplinarny charakter rozwoju systemów wizyjnych, łączący zaawansowaną analizę obrazów z wymaganiem aplikacyjnymi w różnych środowiskach. Systemy te znajdują zastosowanie nie tylko w interakcji człowiek-robot, ale również w misjach eksploracyjnych, logistyce, rolnictwie oraz operacjach ratunkowych.

## 2.3 Przetwarzanie obrazu i algorytmy detekcji obiektów

Współczesne systemy wizyjne opierają się na zaawansowanych technikach przetwarzania obrazu i algorytmach detekcji obiektów, które umożliwiają analizę wizualną w czasie rzeczywistym. Metody te można podzielić na dwie główne kategorie: algorytmy głębokiego uczenia oraz klasyczne metody przetwarzania obrazu.

### 2.3.1 Algorytmy głębokiego uczenia

Algorytmy głębokiego uczenia rewolucjonizują detekcję obiektów dzięki zdolności do automatycznego wydobywania istotnych cech z danych wizualnych. Kluczowe techniki obejmują:

- **YOLO (You Only Look Once)** – jednoopisowy model detekcji obiektów, który przekształca problem detekcji w zadanie regresji. YOLO osiąga wysoką wydajność i dokładność dzięki swojej zwartej architekturze.
- **TensorFlow Object Detection API** – uniwersalne środowisko umożliwiające implementację różnych modeli detekcji, takich jak Faster R-CNN, SSD czy RetinaNet.

### 2.3.2 Klasyczne metody przetwarzania obrazu

Klasyczne metody są prostsze pod względem implementacji i mniej wymagające obliczeniowo. Obejmują techniki takie jak:

- **Binaryzacja** – metoda przekształcająca obraz na dwuwartościowy, co ułatwia segmentację obiektów.
- **Wykrywanie krawędzi** – techniki takie jak filtry Sobela czy Canny'ego, pozwalające na identyfikację konturów obiektów.
- **Klasteryzacja** – grupowanie pikseli na podstawie ich podobieństwa (np. algorytm k-średnich) w celu segmentacji obrazu.

### 2.3.3 Porównanie metod

Podczas gdy algorytmy głębokiego uczenia oferują wysoką precyzję i wszechstronność, klasyczne metody są bardziej ekonomiczne i lepiej sprawdzają się na urządzeniach o ograniczonej mocy obliczeniowej, takich jak Raspberry Pi. Wybór metody zależy od specyficznych wymagań aplikacji.

## 2.4 Sieci neuronowe i ich uczenie

Sieci neuronowe to matematyczne modele inspirowane funkcjonowaniem biologicznych układów neuronalnych. Zostały zaprojektowane w celu rozwiązywania problemów takich jak klasyfikacja, predykcja czy modelowanie złożonych zależności. Podstawową jednostką w sieci neuronowej jest **neuron**, który przetwarza dane wejściowe i generuje odpowiedź w oparciu o zdefiniowane wagi i funkcję aktywacji [3].

### 2.4.1 Budowa sieci neuronowej

Neuron jest podstawową jednostką obliczeniową sieci. Posiada  $n$  wejście  $x_1, x_2, \dots, x_n$ , które są przekształcane zgodnie z wzorem:

$$s = \sum_{i=1}^n w_i x_i + b, \quad (2.1)$$

gdzie  $w_i$  to wagi,  $b$  to przesunięcie (ang. *bias*), a  $s$  to suma ważona wejścia, zwana całkowitym wzbudzeniem neuronu. Pobudzenie  $s$  jest przekształcane za pomocą funkcji aktywacji  $f(s)$ , co daje wyjście neuronu  $y$ :

$$y = f(s). \quad (2.2)$$

**Funkcje aktywacji** Funkcja aktywacji  $f(s)$  determinuje, jak neuron reaguje na wzbudzenie. Przykłady funkcji aktywacji to:

- **Sigmoidalna:**  $f(s) = \frac{1}{1+e^{-s}}$ , umożliwiająca modelowanie nieliniowych zależności w sposób płynny.
- **Tangens hiperboliczny:**  $f(s) = \tanh(s)$ , zapewniający wyjścia w zakresie od  $-1$  do  $1$ .
- **Signum:** Funkcja aktywacji *signum* jest wykorzystywana w sytuacjach, gdzie konieczne jest jednoznaczne rozróżnienie między dodatnim a ujemnym wzbudzeniem:

$$f(s) = \begin{cases} -1, & \text{gdy } v < 0, \\ 1, & \text{gdy } v \geq 0. \end{cases} \quad (2.3)$$

## 2.4.2 Architektura sieci

Sieci neuronowe składają się z trzech typów warstw:

- **Warstwa wejściowa:** Przyjmuje dane wejściowe, np. obrazy, sygnały
- **Warstwy ukryte:** Przetwarzają dane, ucząc się istotnych cech.
- **Warstwa wyjściowa:** Generuje wynik, np. klasyfikację obiektu.

## 2.4.3 Proces uczenia sieci neuronowej

Uczenie sieci neuronowej polega na dostosowywaniu wag  $w_i$  w celu minimalizacji błędu między wynikami przewidywanymi przez sieć a oczekiwanyimi. Wykorzystuje się w tym celu algorytmy gradientowe, takie jak propagacja wsteczna błędu (ang. *backpropagation*).

**Propagacja wsteczna** Algorytm propagacji wstecznej to metoda stosowana w procesie uczenia wielowarstwowych sieci neuronowych. Jej celem jest dostosowanie wag w sieci w taki sposób, aby minimalizować różnicę między wynikami przewidywanymi przez sieć a oczekiwanyimi. Algorytm działa w kilku etapach:

1. **Przepływ w przód:** Dane wejściowe są przetwarzane przez kolejne warstwy sieci. Każdy neuron przekształca dane wejściowe na podstawie swoich wag i funkcji aktywacji, generując sygnały wyjściowe dla następnej warstwy. Na końcu sieci obliczane są wyniki, które są porównywane z wartościami oczekiwanyimi.
2. **Obliczenie błędu:** Na podstawie różnicy między rzeczywistym wynikiem sieci a oczekiwany resultatem obliczany jest błąd, który jest wykorzystywany do dalszych obliczeń.
3. **Propagacja błędu wstecz:** Obliczony błąd jest propagowany wstecz przez sieć, od warstwy wyjściowej do warstw ukrytych. Każda warstwa analizuje, w jakim stopniu jej wagi przyczyniły się do powstania błędu.
4. **Aktualizacja wag:** Wagi neuronów są dostosowywane na podstawie obliczonych błędów, co zmniejsza różnicę między przewidywaniami a oczekiwanyimi wynikami w kolejnych iteracjach.

## 2.4.4 Zasada działania

YOLO (*You Only Look Once*) to algorytm detekcji obiektów, który przekształca cały obraz na jedną siatkę o wymiarach  $S \times S$ . Każda komórka siatki analizuje fragment obrazu i przewiduje:

- ramki ograniczające (ang. *bounding boxes*), które określają potencjalne pozycje obiektów,
- prawdopodobieństwo, że w danej ramce znajduje się obiekt,
- klasę obiektu,

Algorytm działa w czasie rzeczywistym, co oznacza, że przetwarza obraz w jednym przebiegu sieci neuronowej. YOLO jest zoptymalizowane do jednoczesnego wykrywania lokalizacji i klasyfikacji obiektów, co czyni go szybszym w porównaniu do innych algorytmów, które wykonują te zadania oddzielnie.

Przewidywania są przedstawiane w formie wielowymiarowej macierzy (ang. *tensor*) o wymiarach  $S \times S \times (B \cdot 5 + C)$ , gdzie  $S$  to liczba komórek w siatce,  $B$  to liczba ramek ograniczających przewidywanych przez każdą komórkę, a  $C$  to liczba klas obiektów. Dzięki temu YOLO jest w stanie wykrywać wiele obiektów jednocześnie, niezależnie od ich rozmiaru i położenia na obrazie.

Kluczową cechą YOLO jest jego zdolność do analizy całego obrazu naraz, co pozwala algorytmowi rozpoznawać kontekst sceny. Takie podejście zmniejsza liczbę fałszywych pozytywów i zwiększa dokładność w porównaniu do metod opierających się na przeszukiwaniu obrazu kawałek po kawałku.

#### 2.4.5 Architektura sieci

YOLO wykorzystuje sieć konwolucyjną z warstwami  $1 \times 1$  i  $3 \times 3$ , co umożliwia redukcję wymiarów oraz efektywną ekstrakcję cech. Wersja YOLOv7 wprowadza dodatkowe optymalizacje, takie jak moduły E-ELAN i dynamiczne przypisywanie etykiet [9].

#### 2.4.6 Zastosowania w robotyce mobilnej

Systemy wizyjne oparte na YOLO są wykorzystywane w projektach przemysłowych i naukowych. Przykłady zastosowań obejmują:

- nawigację autonomiczną w robotach AGV (ang. *Automated Guided Vehicles*),
- wykrywanie przeszkód i analiza ścieżki w pojazdach autonomicznych,
- systemy inspekcji w przemyśle.

### 2.5 Binaryzacja

Binaryzacja to podstawowa technika przetwarzania obrazu, która przekształca obraz w wersję dwuwartościową (0 i 1), co upraszcza jego analizę. Metoda ta opiera się na

zastosowaniu progu (ang. *thresholding*), który określa granicę oddzielającą piksele jasne od ciemnych.

### 2.5.1 Zasada działania

W procesie binaryzacji każdy piksel  $I(x, y)$  obrazu wejściowego jest porównywany z ustalonym progiem  $T$ :

$$I'(x, y) = \begin{cases} 1, & \text{jeśli } I(x, y) > T, \\ 0, & \text{w przeciwnym razie.} \end{cases} \quad (2.4)$$

Istnieją różne techniki wyznaczania progu, np. metoda Otsu, która automatycznie dobiera  $T$ , minimalizując wariancję wewnętrzklasową [4].

### 2.5.2 Zastosowania

Binaryzacja znajduje szerokie zastosowanie w różnych zadaniach przetwarzania obrazu, takich jak:

- segmentacja obiektów na obrazie, np. identyfikacja obszarów zainteresowania w analizie medycznej,
- detekcja krawędzi, co umożliwia wydobywanie konturów obiektów na obrazie,
- redukcja złożoności obrazu w celu optymalizacji przetwarzania w systemach o ograniczonych zasobach obliczeniowych, takich jak Raspberry Pi,
- wspomaganie systemów wizyjnych w robotyce, np. do śledzenia linii na torze jazdy w autonomicznych pojazdach.

## 2.6 Studia literaturowe i znane rozwiązania

W literaturze naukowej można znaleźć liczne prace dotyczące implementacji systemów wizyjnych w robotyce mobilnej. W niniejszym rozdziale przytoczono przykłady wybranych badań.

### 2.6.1 System programowania i sterowania robota mobilnego

W pracy [7] prof. dr hab. inż. Krzysztof Kozłowski i współautorzy przedstawili konцепcję oraz realizację modułowego robota mobilnego, zaprojektowanego do zastosowań transportowych w zamkniętych środowiskach. Celem projektu było opracowanie systemu

programowania i sterowania, uwzględniającego wymogi bezpieczeństwa oraz przyjaznego interfejsu użytkownika.

**Cel systemu:** Robot został zaprojektowany jako urządzenie transportowe o nośności do 120 kg, dedykowane do przewozu ładunków w pomieszczeniach zamkniętych. Wymagania projektowe obejmowały:

- Modułową budowę, umożliwiającą integrację różnych podsystemów.
- Zapewnienie bezpieczeństwa użytkowania dzięki systemowi zderzaków i czujników.
- Możliwość programowania trasy przejazdu za pomocą przyjaznego interfejsu.

**Metodologia:** Autorzy zastosowali klasyczny model pojazdu dwukołowego z napędem różnicowym oraz pasywnym kołem podporowym. System sterowania oparto na komputerze pokładowym klasy PC oraz dedykowanym oprogramowaniu zbudowanym w architekturze klient-serwer. Wprowadzono język programowania LeoOS Programming Language (LPL), który umożliwia definiowanie trajektorii ruchu robota za pomocą punktów referencyjnych.

**Wyniki:** Robot został wyposażony w zaawansowane systemy sensoryczne, w tym skaner laserowy oraz sieć sensorów ultradźwiękowych i podczerwieni. Wyniki eksperymentalne wskazują na wysoką precyzję sterowania oraz elastyczność w realizacji zadań, takich jak:

- Dokowanie do źródła zasilania.
- Transport ładunków przy pomocy zautomatyzowanego zaczepu.
- Nawigacja w otoczeniu z przeszkodami.

**Wnioski:** Praca autorów pokazuje potencjał zastosowania modularnych systemów sterowania w robotyce mobilnej, szczególnie w środowiskach o dużych wymaganiach bezpieczeństwa. Implementacja języka LPL oraz architektury modułowej zwiększa elastyczność i efektywność systemu, jednocześnie otwierając drogę do dalszego rozwoju.

## 2.6.2 Aktywny system wizyjny dla robota kroczącego

W pracy [5] mgr inż. Przemysław Łabęcki i Dr hab. inż. Andrzej Kasiński przedstawili aktywny system wizyjny, zaprojektowany z myślą o implementacji w robotach kroczących. System ten opierał się na połączeniu kamery oraz oświetlacza laserowego, którego zadaniem było rzutowanie światła na otoczenie w celu pozyskania informacji o geometrii przestrzeni.

**Cel systemu:** Głównym celem opisanego systemu było zapewnienie robotowi informacji 3D o przestrzeni znajdującej się bezpośrednio przed nim, co umożliwiało zarówno nawigację, jak i detekcję przeszkód.

**Metodologia:** - Zastosowano arkusz świetlny emitowany przez laser, a jego ślad rejestrowany był przez kamerę. - Opracowano matematyczny model układu, definiujący relacje geometryczne między kamerą, oświetlaczem i otoczeniem. - Przeprowadzono kalibrację systemu, aby dokładnie określić parametry układu, takie jak kąty widzenia kamery i położenie głównych punktów odniesienia. - Do detekcji śladu wiązki lasera na obrazie wykorzystano techniki przetwarzania obrazu w przestrzeni barw HSV.

**Wyniki:** - System pozwalał na odwzorowanie geometrii otoczenia z błędem względnym nieprzekraczającym 5%. - Przeprowadzono eksperymenty z obiektyami znajdującymi się w różnych odległościach od kamery, wykazując wysoką dokładność w pomiarach wymiarów geometrycznych. - System sprawdził się jako dalmierz i narzędzie do tworzenia mapy przestrzeni 3D, co było szczególnie przydatne w planowaniu ruchu robota kroczącego.

**Wnioski:** Wyniki badań potwierdzają przydatność aktywnych systemów wizyjnych w robotyce mobilnej, szczególnie w warunkach wymagających precyzyjnych pomiarów odległości i analizy przestrzeni. Rozwiążanie to cechuje niski koszt i możliwość integracji z systemami sterowania robotów.

### 2.6.3 System wspomagania nawigacji osób niewidomych

W artykule [8] poświęconym konferencji ICAAIC dotyczącej sztucznej inteligencji i rozumowania maszynowego (ang. *Applied Artificial Intelligence and Computing (ICAAIC)*) autorzy przedstawili system wspomagający nawigację osób niewidomych, wykorzystujący platformę Raspberry Pi oraz algorytm YOLO do detekcji obiektów. Głównym celem systemu jest ułatwienie osobom z dysfunkcją wzroku poruszania się w codziennym otoczeniu poprzez identyfikację i lokalizację przeszkód w czasie rzeczywistym.

#### Komponenty systemu:

- **Raspberry Pi:** Platforma obliczeniowa, na której uruchomiono algorytm YOLO i przetwarzanie danych z kamery.
- **Algorytm YOLO (ang. *You Only Look Once*):** Szybkie i dokładne wykrywanie obiektów w obrazie w czasie rzeczywistym.
- **Moduł kamery:** Rejestruje obraz otoczenia, który jest analizowany przez system.
- **Interfejs użytkownika:** Przekazuje informacje zwrotne w formie dźwiękowej lub wibracyjnej, informując użytkownika o wykrytych przeszkodach.

**Działanie systemu:** System analizuje obraz z kamery w czasie rzeczywistym, identyfikuje obiekty znajdujące się na drodze użytkownika i dostarcza odpowiednie informacje zwrotne. Użytkownik jest w ten sposób ostrzegany o przeszkodach, co umożliwia unikanie kolizji i bezpieczne poruszanie się.

#### Zalety rozwiązania:

- **Niski koszt:** Wykorzystanie Raspberry Pi oraz otwartoźródłowego oprogramowania czyni system przystepnym cenowo.
- **Mobilność:** Kompaktowe komponenty pozwalają na łatwe przenoszenie oraz integrację z akcesoriami, takimi jak laska dla niewidomych.
- **Skalowalność:** Możliwość rozbudowy systemu o dodatkowe funkcje, np. rozpoznanie znaków drogowych lub sygnalizacji świetlnej.

**Wnioski:** Przedstawiony system stanowi obiecujące narzędzie wspierające osoby niewidome w samodzielny poruszaniu się, zwiększając ich bezpieczeństwo i komfort życia. Integracja technologii, takich jak Raspberry Pi oraz algorytmy głębokiego uczenia, umożliwia tworzenie efektywnych i dostępnych rozwiązań wspomagających nawigację.

$$y = \frac{\partial x}{\partial t} \quad (2.5)$$

jak i pojedyncze symbole  $x$  i  $y$  składa się w trybie matematycznym.

# Rozdział 3

## Przygotowanie i trening modelu detekcji obiektów

### 3.1 Przygotowanie danych wzorcowych

Do poprawnego wytrenowania sieci neuronowej, konieczne jest przygotowanie zestawu danych wzorcowych. W przypadku algorytmów głębokiego uczenia kluczowe znaczenie ma jakość i zróżnicowanie danych treningowych, ponieważ sieci neuronowe uczą się rozpoznawania wzorców na podstawie dostarczonych przykładów.

Dane wzorcowe muszą spełniać pewne istotne wymagania, aby zapewnić odpowiednią skuteczność modelu:

- **Różnorodność scenariuszy:** Obrazy powinny przedstawać obiekt w różnych warunkach oświetleniowych (np. światło dzienne, sztuczne oświetlenie, cienie), co pozwala modelowi uogólniać na nowe środowiska.
- **Zmienne pozycje i orientacje:** Dane powinny zawierać obiekt w różnych orientacjach, kątach oraz pozycjach względem kamery, aby uniknąć problemu nadmiernego dopasowania do specyficznego układu.
- **Różne odległości od kamery:** Obiekty powinny być widoczne w kadrze zarówno z bliska, jak i z daleka, co ułatwia modelowi detekcję niezależnie od dystansu.

W praktyce przygotowanie takiego zbioru danych wymaga użycia narzędzi do annotacji, które pozwalają na oznaczenie obiektów w obrazach za pomocą ramek ograniczających (ang. *bounding boxes*). Do tego celu często wykorzystuje się narzędzia do annotacji zdjęć, takie jak **Roboflow** czy **LabelImg**, które wspomagają proces podziału na zbiory treningowe, walidacyjne i testowe.

## 3.2 Platforma Roboflow

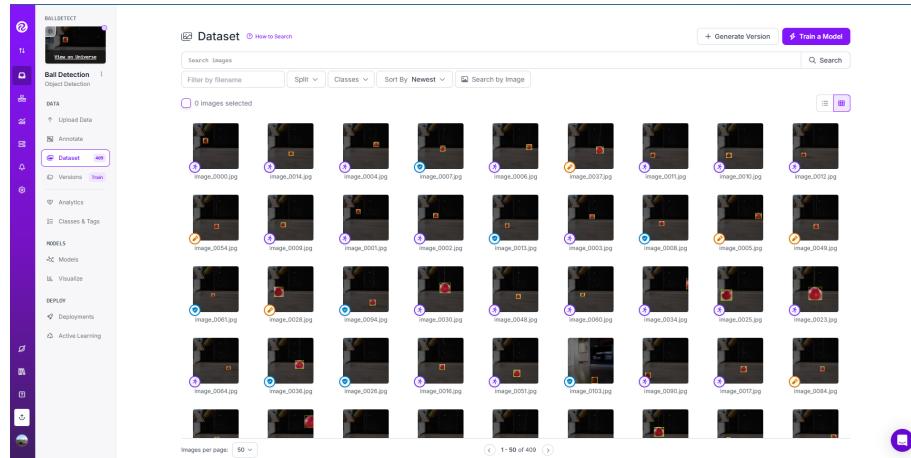
Aby zrealizować założenia pracy, do przygotowania zbioru uczącego wykorzystano platformę Roboflow. Jest to narzędzie dostępne online, które umożliwia zarządzanie danymi, ich etykietowanie oraz konwersję między różnymi formatami. W kontekście tej pracy wykorzystano obrazy przedstawiające czerwoną piłkę w realistycznym środowisku.

Proces przygotowania danych w Roboflow składał się z następujących etapów:

1. **Zbieranie danych** - zgromadzono zestaw około 400 zdjęć przedstawiających czerwoną piłkę w różnych warunkach oświetleniowych i na różnych tłaach.
2. **Annotacja** - każde zdjęcie zostało oznaczone poprzez narysowanie ramki ograniczającej wokół piłki. Roboflow udostępnia intuicyjny interfejs do tego procesu.
3. **Augmentacja danych** - aby zwiększyć różnorodność zbioru treningowego, zastosowano następujące techniki augmentacji:
  - Zmiana jasności
  - Rozmycie
  - Zróżnicowanie otoczenia
4. **Podział danych** - zbiór został podzielony w proporcjach:
  - 70% - zbiór treningowy
  - 20% - zbiór walidacyjny
  - 10% - zbiór testowy
5. **Eksport** - przygotowane dane zostały wyeksportowane w formacie YOLO v7 Pytorch.

### 3.2.1 Proces użycia witryny Roboflow

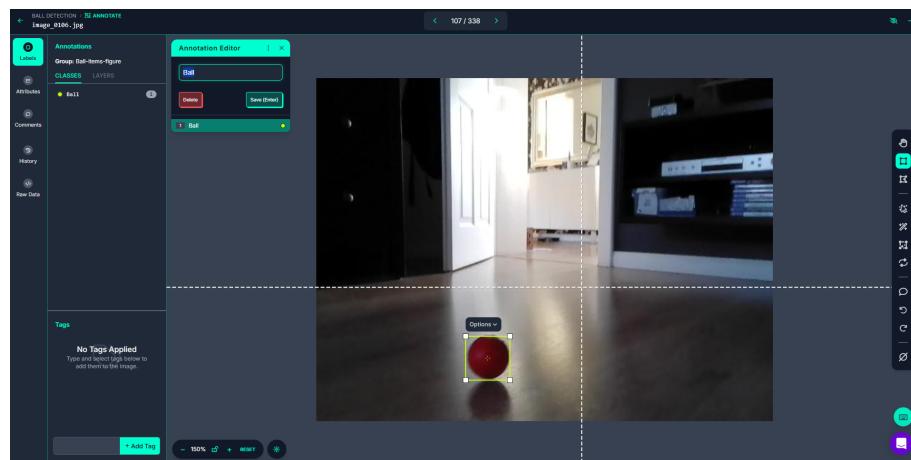
W celu przygotowania danych treningowych, zarejestrowano konto na platformie Roboflow i przystąpiono do etykietowania obiektów. Na rysunku 3.1 przedstawiono widok strony głównej witryny Roboflow.



Rysunek 3.1: Widok strony głównej witryny Roboflow.

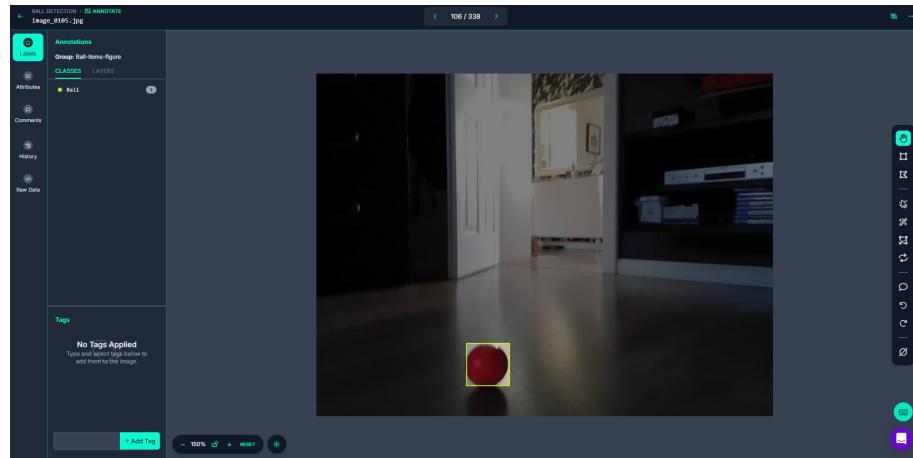
Proces etykietowania można przeprowadzić automatycznie, korzystając z algorytmów detekcji obiektów, lub ręcznie, poprzez zaznaczenie obiektów na obrazie. Korzystając z etykietowania ręcznego istnieje pewność, że obiekty zostaną poprawnie oznaczone w takich przypadkach jak rozmyty obraz bądź słabe oświetlenie.

**Zdjęcie 3.2** przedstawia interfejs użytkownika podczas etykietowania obiektów. Na obrazie widoczna jest czerwona piłka, która została zaznaczona ramką ograniczającą.



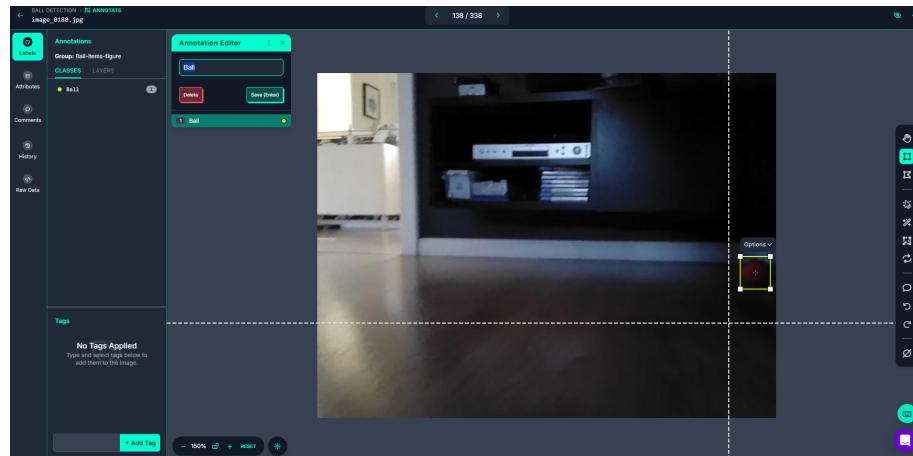
Rysunek 3.2: Etykietowanie obiektów za pomocą witryny Roboflow.

**Zdjęcie 3.3** przedstawia obiekt po zetykietowaniu. Na obrazie widoczna jest czerwona piłka, która została poprawnie oznaczona ramką ograniczającą.



Rysunek 3.3: Obiekt po zetykietowaniu.

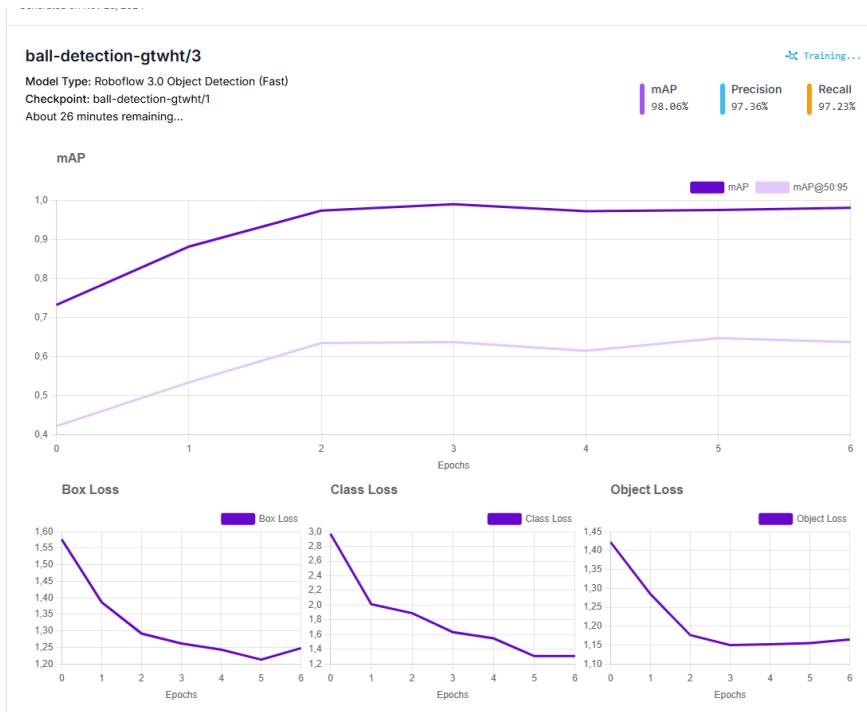
**Zdjęcie 3.4** pokazuje proces etykietowania w warunkach słabego oświetlenia. Pomimo trudnych warunków, obiekt został poprawnie oznaczony, co świadczy o zalecie ręcznego etykietowania w takich przypadkach.



Rysunek 3.4: Etykietowanie w słabym oświetleniu.

### 3.2.2 Trening zestawu danych

Po zakończeniu etykietowania danych, przystąpiono do segmentacji zbioru na zbiór treningowy, walidacyjny i testowy. Następnie przeprowadzono trening danych aby można było wyeksportować je w formacie YOLO v7 Pytorch. Na wykresie 3.5 przedstawiono przebieg treningu w pierwszych epokach. Widać, że wartość funkcji straty maleje wraz z postępem uczenia, co świadczy o skuteczności procesu.



Rysunek 3.5: Wykres pierwszych epok treningu.

Natomiast wykres 3.6 przedstawia przebieg treningu w kolejnych epokach. Widać, że w momencie 20 epoki wartość wskaźnika mAP maleje. **mAP (mean Average Precision)** jest miarą jakości modeli detekcji obiektów, uwzględniającą zarówno *precision*, jak i *recall*. Oblicza się go jako średnią precyzji (AP) dla wszystkich klas obiektów.

Podczas treningu mAP może zmaleć, a następnie wzrosnąć z kilku powodów:

- **Augmentacja danych:** Trudniejsze przypadki augmentacji danych mogą chwilowo obniżyć mAP, ale poprawiają generalizację modelu.
- **Dane walidacyjne:** Specyficzne, trudne obrazy w danych walidacyjnych mogą wpływać na chwilowe obniżenie wyników.

Spadek mAP jest naturalnym zjawiskiem, oznaczającym reakcję modelu na zmiany w procesie treningu. Wzrost mAP po spadku wskazuje na poprawę generalizacji oraz stabilizację procesu optymalizacji.



Rysunek 3.6: Wykres przebiegu treningu z zaistniałym spadkiem mAP.

### 3.3 Trening modelu YOLOv7

Po przygotowaniu danych treningowych przystąpiono do wytrenowania modelu detekcji obiektów YOLOv7, korzystając z platformy Google Colab. Umożliwia ona wykorzystanie zasobów obliczeniowych w chmurze, w tym akceleracji przez GPU (Graphics Processing Unit), które znaczco przyspiesza przetwarzanie dużych ilości danych dzięki równoległemu wykonywaniu obliczeń. Jest to szczególnie istotne w przypadku modeli uczenia maszynowego, takich jak głębokie sieci neuronowe, które wymagają intensywnych operacji macierzowych. Proces treningu przebiegał zgodnie z następującymi krokami:

#### 3.3.1 Przygotowanie środowiska

Utworzenie nowego projektu w Google Colab i skonfigurowano środowiska do korzystania z GPU:

1. Z paska narzędzi **Środowisko wykonawcze** wybrano opcję **Zmień typ środowiska wykonawczego**.
2. Jako **Akcelerator sprzętowy** ustawiono GPU.

#### 3.3.2 Instalacja wymaganych bibliotek

Pobranie repozytorium YOLOv7 z platformy GitHub oraz zainstalowano niezbędne zależności:

```
!git clone https://github.com/WongKinYiu/yolov7  
%cd yolov7  
!pip install -r requirements.txt
```

#### 3.3.3 Pobranie i przygotowanie danych treningowych

Dane treningowe zostały przygotowane na platformie Roboflow, gdzie wygenerowano klucz API, który umożliwił automatyczne pobranie danych do środowiska Google Colab. W notebooku dodano poniższy fragment kodu do zainportowania danych:

```
!pip install roboflow  
from roboflow import Roboflow  
rf = Roboflow(api_key="TWÓJ_KLUCZ_API")  
project = rf.workspace().project("nazwa-projektu")  
dataset = project.version(1).download("yolov7")
```

### 3.3.4 Przygotowanie modelu

Pobranie wstępnie wytrenowanych wag modelu YOLOv7:

```
!wget https://github.com/WongKinYiu/yolov7/releases/download/v0.1/yolov7.pt
```

### 3.3.5 Trening modelu

Model wytrenowano, uruchamiając poniższą komendę:

```
%env WANDB_MODE=disabled  
!python yolov7/train.py --data Ball-Detection-3/data.yaml --cfg yolov7/cfg/training/yolov7.pt
```

Parametry treningu były następujące:

- **-data**: Ścieżka do pliku konfiguracyjnego zestawu danych otrzymanych z Roboflow.
- **-cfg**: Plik konfiguracyjny architektury modelu.
- **-weights**: Wstępnie wytrenowane wagi modelu YOLOv7.
- **-epochs**: Liczba epok (ustawiono na 50).
- **-batch-size**: Rozmiar grupy treningowej (ustawiono na 16).
- **-device**: GPU (0 oznacza pierwsze dostępne GPU).

Po zakończeniu treningu najlepsze wagi modelu są zapisywane w pliku `best.pt` w folderze `runs`. Plik ten będzie wykorzystywany w programie do detekcji obiektu w czasie rzeczywistym. Dodatkowo należy pobrać foldery `models` oraz `utils` z repozytorium YOLOv7 i umieścić je w folderze z projektem.

# Rozdział 4

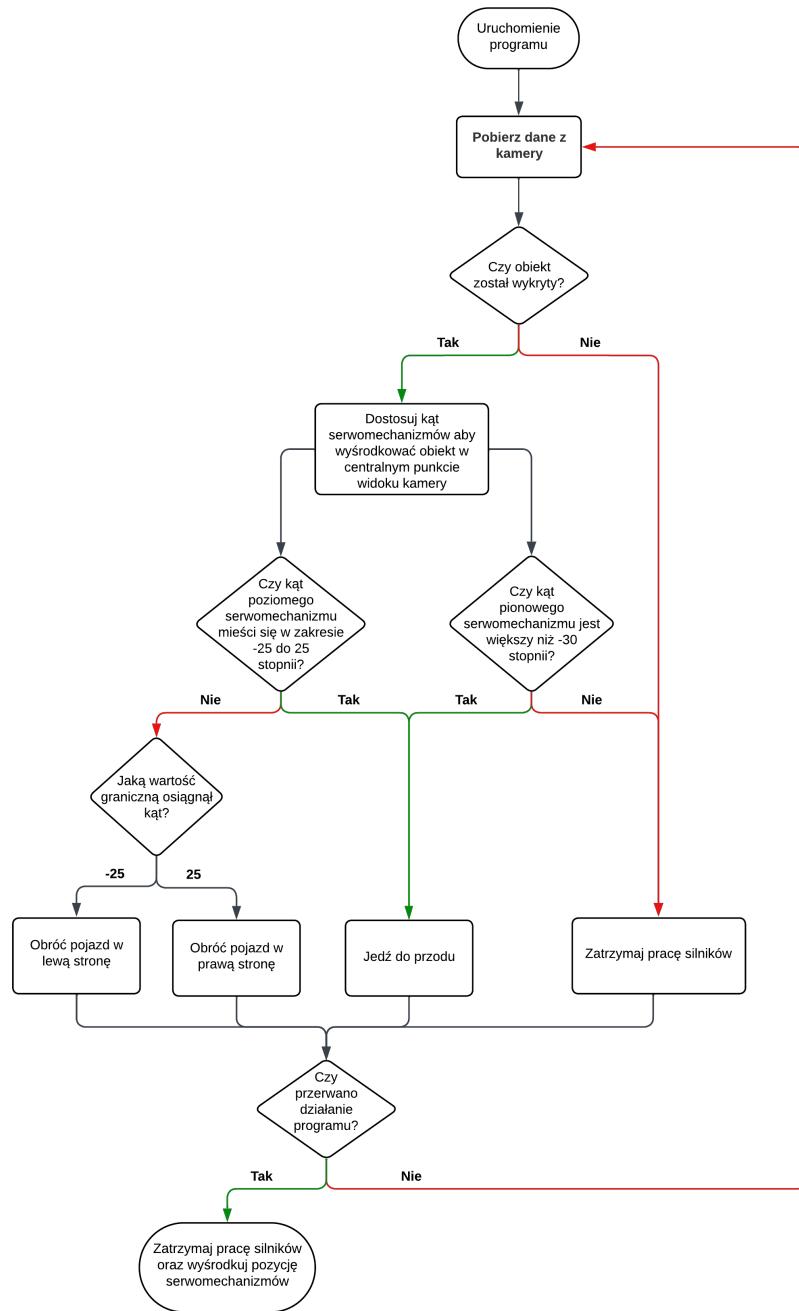
## Proces tworzenia robota mobilnego

W obecnym rozdziale przedstawiony zostanie proces tworzenia robota mobilnego, uwzględniający zarówno założenia logiczne, jak i aspekty techniczne związane z budową oraz implementacją systemów sterowania. W pierwszej części opisane zostaną kluczowe założenia projektowe oraz schemat blokowy algorytmu robota mobilnego opartego na systemie wizyjnym. Następnie zostanie zaprezentowany proces konstrukcji robota, opierając się na modelu 3D stworzonym w programie Autodesk Fusion oraz przedstawienie wykorzystanych modułów i ich sposób podłączenia przedstawionego za pośrednictwem schematu blokowego. Ostatnia część rozdziału dotyczy dostosowania jednostki sterującej w postaci Raspberry Pi, w tym konfiguracji niezbędnych bibliotek umożliwiających integrację podzespołów i realizację funkcjonalności systemu wizyjnego dla robota mobilnego.

### 4.1 Założenia logiczne

Poniżej przedstawiono diagram blokowy ilustrujący działanie algorytmu detekcji obiektów w czasie rzeczywistym. Proces rozpoczyna się od akwizycji obrazu z kamery, który następnie trafia do modelu detekcji obiektów YOLOv7. Na podstawie wyników detekcji robot dostosowuje kąty serwomechanizmów pionowego i poziomego, aby utrzymać obiekt w centrum kadru. Jeśli obiekt przesunie się poza wyznaczone centrum, program oblicza różnicę w pikselach i przelicza ją na odpowiednie wartości kątowe, umożliwiając sterowanie serwomechanizmami w celu zapewnienia ciągłości śledzenia. Celem algorytmu jest nieustanne utrzymanie obiektu w centrum widoku kamery. Program na bieżąco monituruje wartości kątowe serwomechanizmów, które są wykorzystywane do sterowania pracą silników DC odpowiedzialnych za ruch robota. Ruch w przód wykonywany jest jedynie w przypadku spełnienia następujących warunków: obiekt został wykryty, kąt serwomechanizmu poziomego mieści się w zakresie od  $-25^\circ$  do  $25^\circ$ , a kąt serwomechanizmu pionowego przekracza  $-30^\circ$ . W przeciwnym razie robot zatrzymuje się lub wykonuje skręt. Jeśli kąt poziomy osiągnie wartość graniczną ( $25^\circ$  lub  $-25^\circ$ ), robot skręca odpowiednio w prawo lub w lewo.

Program można zatrzymać wyłącznie przez interakcję użytkownika, co powoduje zatrzymanie wszystkich systemów i powrót robota do stanu początkowego. Stan początkowy definiuje sytuację, w której serwomechanizmy ustawione są w pozycji centralnej ( $0^\circ$  dla obu osi), a silniki DC są wyłączone.



Rysunek 4.1: Schemat blokowy działania algorytmu pracy robota z systemem wizyjnym

## 4.2 Konstrukcja robota mobilnego

Konstrukcja robota mobilnego oparta jest na platformie jezdnej z dwoma silnikami DC o napędzie różnicowym, serwomechanizmem pionowym i poziomym oraz kamerą. Robot wyposażony jest w moduł Raspberry Pi 4B, który pełni rolę jednostki decyzyjnej oraz komunikacyjnej. Poniżej przedstawiono schemat blokowy robota mobilnego z zaznaczonymi podstawowymi elementami składowymi.

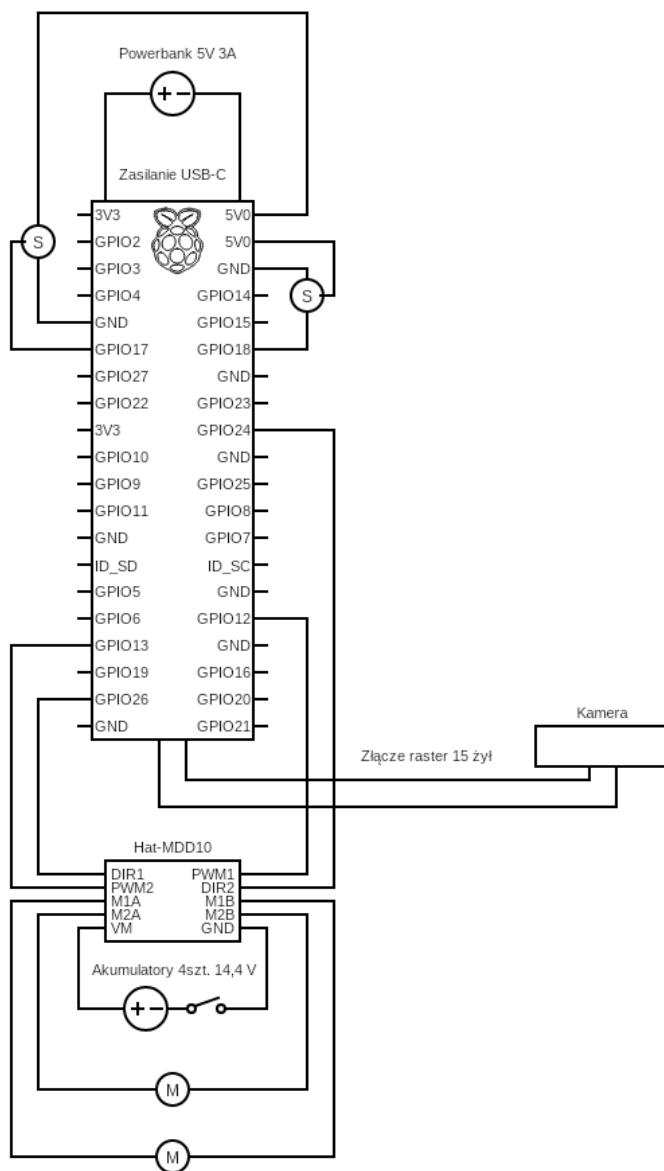
### 4.2.1 Wykorzystane moduły

Aby zrealizować założenia projektowe, wykorzystano następujące moduły i komponenty:

- **Raspberry Pi:** Raspberry Pi 4B WiFi 8GB RAM - jednostka centralna robota, odpowiedzialna za przetwarzanie obrazu, sterowanie silnikami oraz serwomechanizmami.
- **Kamera:** Raspberry Pi Camera HD v3 12MPx, służąca do akwizycji obrazu.
- **Moduł sterownika silników:** Cytron HAT-MDD10 - dwukanałowy sterownik silników DC 24V/10A z funkcją PWM służący jako nakładka na Raspberry Pi.
- **Silniki DC:** Dwa silniki z przekładnią 37Dx68L 30:1 12V 330RPM + enkoder CPR 64 - Pololu 4752 - napędzające koła robota.
- **Zasilanie:** Cztery ogniska 18650 Li-Ion Samsung INR18650-35E 3500mAh 3,6V - zasilanie silników DC.
- **Serwomechanizm pionowy:** Serwo TowerPro SG-90 micro 180 stopni - odpowiada za regulację kąta nachylenia kamery w pionie.
- **Serwomechanizm poziomy:** Serwo TowerPro SG-92R micro - odpowiada za regulację kąta obrotu kamery w poziomie.
- **Uchwyt kamery:** Wykonany z połączenia gotowego uchwytu do serw micro Pan/Tilt oraz dedykowanego uchwytu do kamery Raspberry Pi v2 oraz v3.
- **Obudowa:** Wykonana przy pomocy drukarki 3D z wytrzymałego filamentu Spectrum PETG 1,75mm.
- **Koła:** Dwa Koła Pololu 90x10mm - Zapewniające ruch robota po powierzchni.
- **Zasilanie Raspberry Pi:** Powerbank o parametrach 30000mAh 5V 3A.
- **Mocowanie silników:** Dwie sztuki mocowań aluminiowych do silników 37D Pololu 1084.

- **Huby mocujące:** Dwa aluminiowe huby mocujące 6mm M3 do kół Pololu 1999.
- **Koło wspornikowe pasywne:** Ball Caster 3/4" metalowe Pololu 955 - zapewniające stabilność robota.
- **Taśma do kamery:** Taśma Raspberry Pi kamera 60cm 15 żyłowa raster 1mm - łącząca kamerę z Raspberry Pi.
- **Przewody:** Zestaw przewodów połączeniowych justPi męsko-męskich, żeńsko-żeńskich, żeńsko-męskich.
- **Przełącznik:** Przełącznik dźwigniowy ON-OFF KN3(C)-101 250V/6A - służący do ręcznego wyłączania zasilania silników DC.

#### 4.2.2 Schemat elektryczny



Rysunek 4.2: Schemat elektryczny podłączeń komponentów.

**Symbol M:** Symbolami M oznaczono silniki DC. Każdy z silników jest podłączany ze sterownikiem Hat-MDD10 dwoma przewodami: czerwonym (dodatnim) do pinu M1A lub M2A oraz czarnym (ujemnym) do pinu M1B lub M2B.

**Symbol S:** Symbolami S oznaczono serwomechanizmy. Serwomechanizmy podpięte są bezpośrednio do Raspberry Pi trzema przewodami - czerwonym (dodatnim) do pinu 5V, brązowym (ujemnym) do pinu GND oraz pomarańczowym (sygnałowym) do pinu GPIO.

**Kamera:** Kamera podłączona jest do Raspberry Pi za pomocą piętnastożyłowej taśmy, która łączy się z portem CSI (ang. Camera Serial Interface).

**Hat-MDD10:** Wedle instrukcji producenta [2], sterownik silników DC podłączony jest do Raspberry Pi za pomocą czterech przewodów do pinów GPIO1, GPIO2, PWM1 oraz PWM2. Z uwagi iż to zastosowanie powodowało asynchroniczne działanie silników DC, zdecydowano się na podłączenie sterownika z modułem Raspberry Pi w sposób nakładkowy.

**Powerbank 5V 3A:** Powerbank podłączony jest do Raspberry Pi za pomocą przewodu USB typu A - USB typu C.

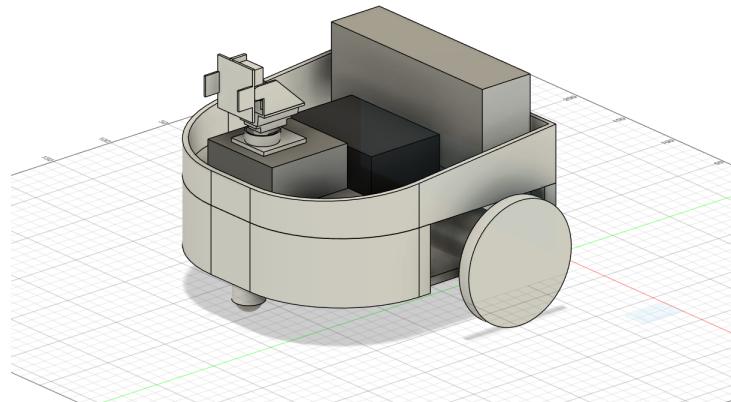
**Akumulatory 18650:** Cztery ogniwa litowo-jonowe podłączone są szeregowo bezpośrednio do sterownika Hat-MDD10 dają łączne maksymalne napięcie 14,4V.

**Przełącznik:** Przełącznik dźwigniowy ON-OFF przerywający ujemny przewód łączący akumulatory z Hat-MDD10.

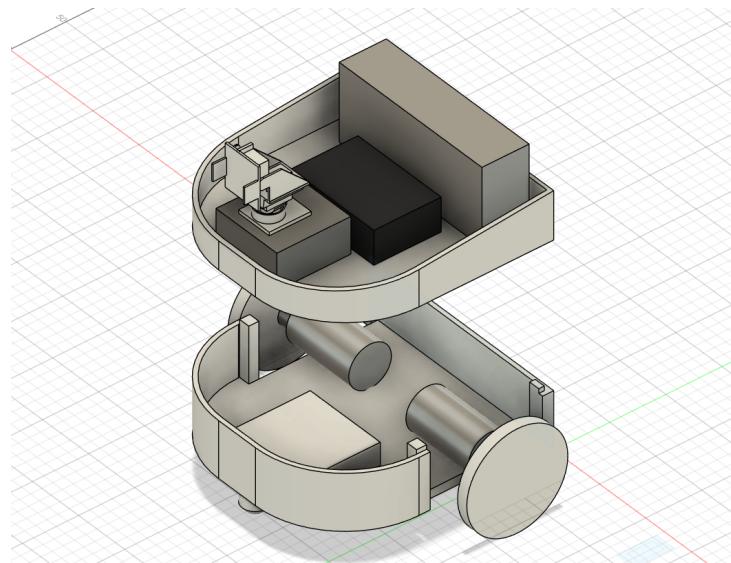
**RaspberryPi:** Moduł Raspberry Pi 4B zasilany z powerbanku 5V 3A oraz podłączony ze sterownikiem Hat-MDD10 w sposób nakładkowy.

### 4.2.3 Projekt robota mobilnego

Jednym z założeń projektu było stworzenie kompaktowego robota mobilnego o możliwie prostym i modułowym układzie konstrukcyjnym. Robot składa się z dwóch pięter: dolnego pełniącego funkcję napędową oraz górnego będącego platformą dla modułów sterujących. Dzięki temu podziałowi możliwe jest zachowanie walorów estetycznych oraz funkcjonalnych.

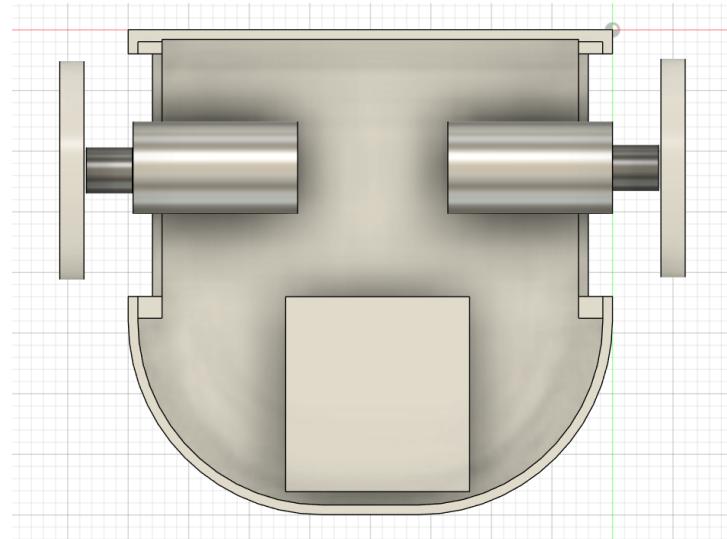


Rysunek 4.3: Projekt konstrukcji robota mobilnego wykonany w programie Autodesk Fusion.



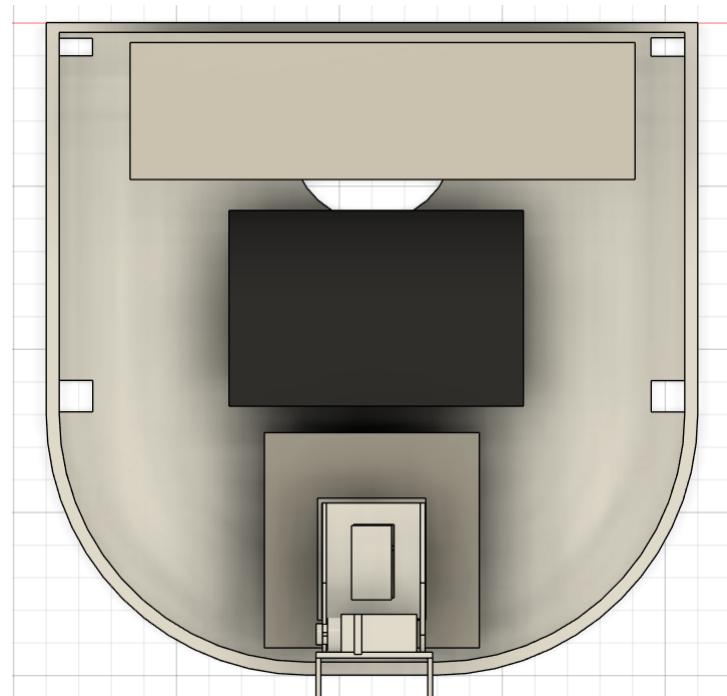
Rysunek 4.4: Podział konstrukcji na piętra.

Obraz 4.5 przedstawia dolne piętro robota, na którym zamontowane są silniki DC wraz z kołami oraz zasilaniem w postaci koszyka z czterema ogniwami 18650 Litowo-jonowymi. Dodatkowo umieszczono przełącznik dźwigniowy ON-OFF, który umożliwia ręczne wyłączenie zasilania silników DC.



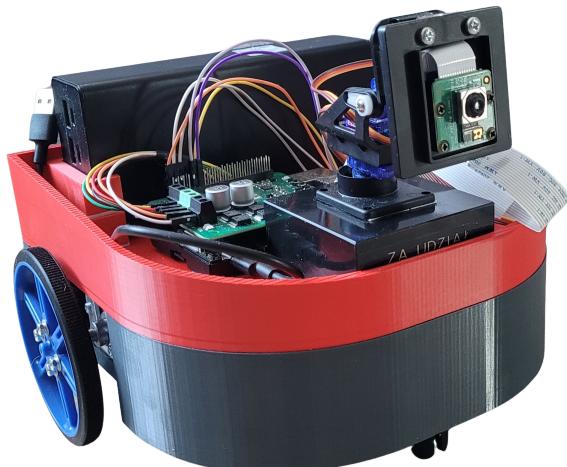
Rysunek 4.5: Widok rzutu górnego na dolne piętro robota.

Obraz 4.6 przedstawia górne piętro robota, na którym zamontowane są takie moduły jak Raspberry Pi, sterownik silników DC, uchwyt na kamerę wraz z serwomechanizmami oraz zasilaniem. Z powodu dużej wagi powerbanka, konieczne było zastosowanie przeciwagi w postaci granitowego bloku na przodzie modułu górnego piętra. Zastosowanie to zapewnia stabilność robota oraz równomierny rozkład masy.



Rysunek 4.6: Widok rzutu górnego na górne piętro robota.

Zdjęcie 4.7 przedstawia gotową konstrukcję robota mobilnego wykonanego z tworzywa sztucznego PETG.



Rysunek 4.7: Gotowa konstrukcja robota mobilnego.

## 4.3 Dostosowanie RaspberryPi

Na Raspberry Pi zainstalowano system operacyjny Debian GNU/Linux 12 (Bookworm), który jest 64-bitową wersją opartą na jądrze Linux 6.6.51, zoptymalizowaną pod architekturę ARM (aarch64). Wybrana wersja systemu jest kompatybilna z bibliotekami istotnymi do poprawnego działania systemu wizyjnego. Aby zoptymalizować pracę programu i zapewnić pełną kontrolę nad jego zależnościami, utworzono dedykowane wirtualne środowisko, w którym przechowywane są wszystkie wymagane biblioteki. Środowisko to zostało stworzone przy użyciu narzędzia `venv` za pomocą poniższego polecenia w terminalu:

```
python -m venv nazwa-env
```

Po utworzeniu środowiska wirtualnego należy je aktywować, używając polecenia:

```
source nazwa-env/bin/activate
```

Program jest uruchamiany wewnątrz tego środowiska, co pozwala na uniknięcie konfliktów między wersjami bibliotek oraz zapewnia powtarzalność działania systemu na różnych urządzeniach. Przykładowe polecenie uruchamiające program w aktywnym środowisku to:

```
python program.py
```

Takie podejście gwarantuje, że wszystkie zależności programu są odpowiednio załadowane i kompatybilne z systemem operacyjnym oraz sprzętem Raspberry Pi.

### 4.3.1 Spis wymaganych bibliotek

Poniżej przedstawiono listę bibliotek wymaganych do uruchomienia systemu wizyjnego oraz sterowania robotem mobilnym, z podziałem na ich funkcje:

- **Platformy uczenia maszynowego:**
  - **torch** – Podstawowa platforma do trenowania i uruchamiania modeli uczenia maszynowego.
  - **models.experimental** – Obsługa modeli YOLO do detekcji obiektów.
  - **utils.general** – Funkcje wspomagające dla YOLO, takie jak filtrowanie wyników i skalowanie współrzędnych.
- **Przetwarzanie obrazu:**
  - **opencv-python (cv2)** – Obsługa obrazów i przetwarzanie wizyjne, np. binaryzacja i filtrowanie.
  - **numpy** – Operacje matematyczne i macierzowe, kluczowe dla analizy danych wizualnych.
  - **Picamera2** – Obsługa kamery Raspberry Pi do akwizycji obrazu.
- **Sterowanie sprzętem:**
  - **RPi.GPIO** – Sterowanie pinami GPIO na Raspberry Pi.
  - **pigpio** – Niskopoziomowe sterowanie GPIO, np. dla serwomechanizmów.
- **Inne ważne biblioteki:**
  - **tqdm** – Pasek postępu przy iteracjach.
  - **PyYAML** – Obsługa plików konfiguracji w formacie YAML.

Szczegółowy spis wszystkich bibliotek został przedstawiony w pliku Requirements.txt dostarczonym w załączniku do pracy.



## Rozdział 5

# Analiza działania systemu wizyjnego

W niniejszym rozdziale przedstawiono analizę działania systemu wizyjnego, który został zaimplementowany w ramach pracy. System ten umożliwia detekcję obiektu na obrazie z kamery Raspberry Pi, a następnie sterowanie serwomechanizmami w celu utrzymania obiektu w centrum kadru. W ramach analizy porównano skuteczność dwóch podejść do detekcji obiektów:

- model YOLOv7 oraz YOLOv8,
- algorytm binaryzacji obrazu z wykorzystaniem przestrzeni barw HSV,

Porównanie zostało przeprowadzone pod kątem dwóch głównych aspektów:

1. **Skuteczności wykrywania obiektów:** Analizowano zdolność algorytmów do wykrywania obiektu w różnych warunkach oświetleniowych oraz w otoczeniu obiektów zakłócających.
2. **Wydajności działania:** Mierzono czas przetwarzania obrazu w jednostkach FPS (klatki na sekundę), co pozwala określić, jak szybko algorytm jest w stanie działać na urządzeniu.

Każde z przedstawionych podejść ma swoje zalety i wady. YOLO to zaawansowany model wykorzystujący sieci neuronowe, które zapewniają wysoką precyzję wykrywania, ale wymagają znacznych zasobów obliczeniowych, co może być wyzwaniem dla urządzeń takich jak Raspberry Pi. Z kolei algorytm binaryzacji obrazu wyróżnia się prostotą i szybkością działania, co czyni go atrakcyjnym w kontekście ograniczonych zasobów sprzętowych.

## 5.1 Specyfikacja sprzętowa

Poniżej przedstawiono specyfikację urządzeń użytych do analizy skuteczności algorytmów:

- **Komputer osobisty:** Urządzenie wykorzystane do testowania algorytmu binaryzacji oraz modelów YOLO.
  - **Procesor:** Intel Core i5-13500H, 12 rdzeni, taktowanie do 2.6 GHz.
  - **Pamięć RAM:** 16 GB DDR5, taktowanie 6400 MHz.
  - **Karta graficzna:**
    - \* Intel Arc A350M Graphics, 1 GB VRAM.
    - \* Intel Iris Xe Graphics, 2 GB VRAM.
  - **System operacyjny:** Microsoft Windows 11 Home, wersja 10.0.22631, architektura 64-bit.
- **Raspberry Pi:** Platforma testowa dla algorytmu binaryzacji oraz modelów YOLO.
  - **Procesor:** Broadcom BCM2711 (Cortex-A72), 4 rdzenie, architektura ARMv8-A, taktowanie 1.5 GHz.
  - **Pamięć RAM:** 8 GB LPDDR4.
  - **System operacyjny:** Debian GNU/Linux 12 (Bookworm).
  - **Kamera:** Raspberry Pi Camera HD v3 12MPx.

W kolejnych sekcjach szczegółowo omówiono działanie algorytmów, przedstawiono wyniki testów oraz przeprowadzono ich porównanie pod względem skuteczności i wydajności.

## 5.2 Detekcja obiektu za pomocą narzędzi głębokiego uczenia

W podrozdziale tym opisano podejście do detekcji obiektu (piłki czerwonej) z wykorzystaniem zaawansowanych narzędzi głębokiego uczenia, takich jak modele z rodziny YOLO (ang. *You Only Look Once*).

W porównaniu do klasycznych technik przetwarzania obrazów, modele głębokiego uczenia wymagają większych zasobów obliczeniowych, ale oferują większą elastyczność i skuteczność w złożonych warunkach otoczenia. W szczególności wykorzystano modele YOLOv7 i YOLOv8, które różnią się między sobą szybkością i dokładnością działania.

### 5.2.1 Opis działania programu

System opiera się na detekcji obiektu (piłki) w czasie rzeczywistym z wykorzystaniem modelu YOLO oraz sterowaniu ruchem robota przy pomocy serwomechanizmów i silników DC. Program realizuje funkcjonalność autonomicznego śledzenia piłki oraz reagowania na jej pozycję w kadrze kamery.

**Wczytanie modelu** Model YOLO, zapisany w pliku `best.pt`, jest wczytywany z użyciem funkcji `attempt_load`. Model jest przełączany w tryb ewaluacyjny (ang. *evaluation mode*), aby umożliwić jego wykorzystanie wyłącznie do predykcji. Kod odpowiedzialny za ładowanie modelu przedstawiono w Kodzie 5.1.

---

```

1 model_path = "/home/lukasgrab/GoodDetection/runs/train/
   ball_detection6/weights/best.pt"
2 device = torch.device('cpu')
3 model = attempt_load(model_path, map_location=device)
4 model.eval()

```

---

Kod 5.1: Ładowanie modelu YOLO do pamięci.

**Detekcja piłki** Podstawowa logika detekcji polega na przetwarzaniu obrazu z kamery, który jest skalowany i przetwarzany przez model YOLO. Wyniki predykcji są filtrowane za pomocą mechanizmu *non-max suppression* w celu eliminacji zbędnych ramek detekcji. Po wykryciu piłki jej pozycja jest obliczana względem środka obrazu. Fragment kodu odpowiedzialny za tę funkcję przedstawiono w Kodzie 5.2.

---

```

1 predictions = model(img)[0]
2 predictions = non_max_suppression(predictions, conf_thres=0.10,
   iou_thres=0.45)
3
4 for det in predictions:
5     if det is not None and len(det):
6         for *xyxy, conf, cls in det:
7             x1, x2 = int(xyxy[0]), int(xyxy[2])
8             y1, y2 = int(xyxy[1]), int(xyxy[3])
9             ball_center_x = (x1 + x2) / 2
10            ball_center_y = (y1 + y2) / 2

```

---

Kod 5.2: Logika detekcji piłki przy użyciu modelu YOLO.

**Logika kwadratu w centrum obrazu** W centrum obrazu rysowany jest kwadrat, który definiuje obszar uznawany za "środek obrazu". Rozmiar kwadratu został dobrany w taki sposób, aby stworzyć większy obszar stabilności dla kamery. Gdyby centrum zostało zdefiniowane jako pojedynczy piksel, nawet niewielkie ruchy obiektu wynikające z drgań kamery, zmieniających się warunków oświetleniowych lub naturalnego ruchu piłki powodowałyby ciągłą próbę korekcji pozycji przez serwomechanizmy. Taka sytuacja skutkowałaby niestabilnością układu.

Kwadrat pozwala na bardziej stabilne śledzenie, ponieważ robot uznaje piłkę za "wycentrowaną", gdy znajduje się w tym obszarze, nawet jeśli nie jest dokładnie w punkcie środkowym.

**Definiowanie ograniczeń kątowych** Sterowanie serwomechanizmami w programie wymagało zdefiniowania zakresów ruchu, aby uniknąć uszkodzenia mechanicznego urządzeń. Dla serwomechanizmu odpowiedzialnego za ruch poziomy (ang. *pan*) przyjęto ograniczenia w zakresie od  $-45^\circ$  do  $45^\circ$ . Z kolei serwomechanizm odpowiedzialny za ruch pionowy (ang. *tilt*) może zmieniać kąt w zakresie od  $-30^\circ$  do  $30^\circ$ . Te wartości są podkutowane fizycznymi możliwościami serwomechanizmów oraz potrzebą zapewnienia odpowiedniej elastyczności ruchu robota.

**Przeliczanie kąta na szerokość impulsu** Sterowanie serwomechanizmami wymaga przekształcenia kąta (w stopniach) na szerokość impulsu (ang. *pulse width*) w mikrosekundach, co odpowiada wymaganiom sterownika serwomechanizmu. Proces ten realizuje funkcja przedstawiona w Kodzie 5.3.

---

```
1 def set_servo_angle(pin, angle):
2     pulse_width = 500 + (angle + 90) * 2000 / 180
3     pi.set_servo_pulsewidth(pin, pulse_width)
```

---

Kod 5.3: Przeliczanie kąta na szerokość impulsu.

Funkcja `set_servo_angle` realizuje następujące kroki:

- `angle` — kąt w stopniach, przekazywany do funkcji. Może przyjmować wartości od  $-90^\circ$  do  $90^\circ$ .
- `pulse_width` — szerokość impulsu w mikrosekundach, obliczana na podstawie wzoru:

$$\text{pulse\_width} = 500 + \frac{(\text{angle} + 90) \cdot 2000}{180} \quad (5.1)$$

gdzie:

- 500 — minimalna szerokość impulsu w mikrosekundach, odpowiadająca kątowi  $-90^\circ$ ,
  - 2000 — maksymalna różnica szerokości impulsu, odpowiadająca zakresowi  $180^\circ$ ,
  - 180 — zakres pełnych kątów ruchu serwomechanizmu ( $-90^\circ$  do  $90^\circ$ ),
  - `angle + 90` — przesunięcie kąta, aby obliczenia obejmowały pełny zakres od  $0^\circ$  do  $180^\circ$ .
- Funkcja wywołuje `pi.set_servo_pulsewidth`, aby ustawić odpowiednią szerokość impulsu na wybranym pinie GPIO (`pin`).

**Sterowanie serwomechanizmami** Sterowanie serwomechanizmami realizuje dynamiczne dostosowanie kątów nachylenia w celu śledzenia pozycji piłki. Pozycja piłki w kadrze obrazu jest przekształcana na współrzędne w pikselach, a następnie na kąty dla serwomechanizmów. Kluczowy fragment kodu odpowiedzialny za sterowanie serwomechanizmami przedstawiono w Kodzie 5.4.

---

```
1 new_pan_angle = current_pan_angle + (normalized_dx * 5)
2 new_tilt_angle = current_tilt_angle - (normalized_dy * 5)
3
4 new_pan_angle = max(MIN_ANGLE_PAN, min(MAX_ANGLE_PAN,
5     new_pan_angle))
6 new_tilt_angle = max(MIN_ANGLE_TILT, min(MAX_ANGLE_TILT,
7     new_tilt_angle))
8
9 set_servo_angle(servopin_pan, new_pan_angle)
10 set_servo_angle(servopin_tilt, new_tilt_angle)
```

---

Kod 5.4: Algorytm sterowania serwomechanizmami.

**Logika sterowania ruchem** Robot realizuje ruch do przodu, gdy piłka jest wykryta i znajduje się w granicach zadanych kątów serwomechanizmów. W przypadku, gdy kąt poziomy (pan) przekroczy wartość graniczną, robot wykonuje skręt w odpowiednią stronę. Logikę sterowania ruchem przedstawiono w Kodzie 5.5.

---

```
1 if abs(current_tilt_angle) < TILT_STOP_THRESHOLD:
2     move_forward()
3 else:
4     stop_motors()
5
6 if abs(current_pan_angle) >= PAN_TURN_THRESHOLD:
7     stop_motors()
8     if current_pan_angle > 0:
9         turn_left()
10    else:
11        turn_right()
```

---

Kod 5.5: Logika sterowania ruchem robota.

**Zgubienie piłki** Jeśli piłka nie zostanie wykryta przez określony czas, robot przechodzi w tryb szukania. W tym trybie serwomechanizmy wykonują w przestrzeni ruchy eliptyczne w celu ponownego odnalezienia obiektu. Ruch ten opiera się na funkcjach trygonometrycznych `sin` i `cos`, które generują odpowiednie kąty dla serwomechanizmów. Kluczowy fragment kodu realizujący ten mechanizm przedstawiono w Kodzie 5.6.

---

```
1 if time.time() - last_detected_time > 1:
2     search_time += 0.1
3     new_pan_angle = PAN_AMPLITUDE * math.cos(search_time)
4     new_tilt_angle = TILT_AMPLITUDE * math.sin(search_time) +
5                         TILT_OFFSET
6     new_pan_angle = max(MIN_ANGLE_PAN, min(MAX_ANGLE_PAN,
7                                     new_pan_angle))
8     new_tilt_angle = max(MIN_ANGLE_TILT, min(MAX_ANGLE_TILT,
9                           new_tilt_angle))
10
11    set_servo_angle(servo_pin_pan, new_pan_angle)
12    set_servo_angle(servo_pin_tilt, new_tilt_angle)
13    current_pan_angle = new_pan_angle
14    current_tilt_angle = new_tilt_angle
```

---

Kod 5.6: Logika zgubienia piłki i ruchu eliptycznego.

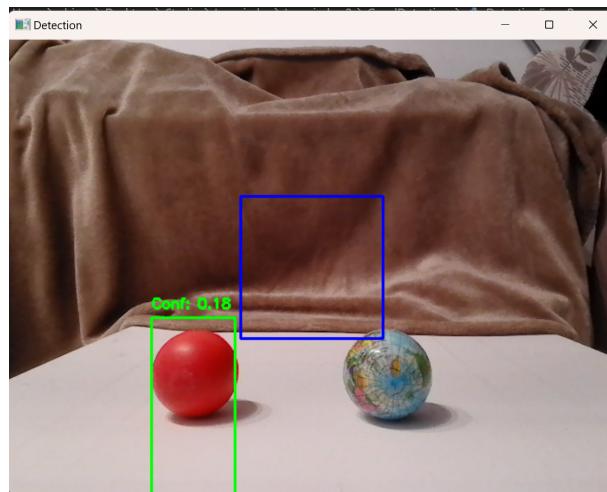
W powyższym kodzie:

- `search_time` — zmienna czasu, która jest inkrementowana w każdej iteracji, aby generować płynne zmiany kąta.
- `math.cos(search_time)` — generuje wartości dla poziomego ruchu serwomechanizmu (`pan`), symulując eliptyczny ruch.
- `math.sin(search_time)` — generuje wartości dla pionowego ruchu serwomechanizmu (`tilt`).
- `PAN_AMPLITUDE` i `TILT_AMPLITUDE` — amplitudy ruchu poziomego i pionowego, które kontrolują zakres ruchu serwomechanizmów.
- `TILT_OFFSET` — przesunięcie pionowe, aby `tilt` oscylował wokół ustalonej wartości.

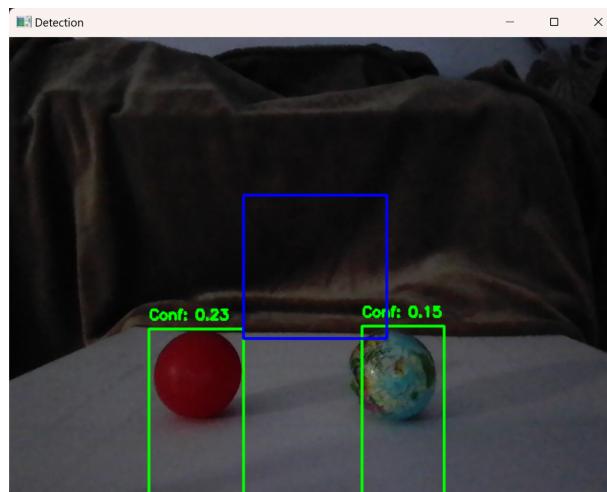
### 5.2.2 Uzyskane rezultaty

W ramach testów systemu wizyjnego przeprowadzono analizę skuteczności detekcji piłki w różnych warunkach oświetleniowych oraz w obecności obiektów potencjalnie zakłócających prawidłowe działanie algorytmu. Celem testów było zweryfikowanie odporności systemu na zmienne warunki środowiskowe oraz jego zdolności do ignorowania obiektów o zbliżonym kształcie lub barwie do śledzonej piłki.

Na Rysunku 5.1 przedstawiono sytuację przy dobrym oświetleniu, gdzie system poprawnie wykrył piłkę czerwoną, jednocześnie ignorując inne obiekty znajdujące się w polu widzenia kamery. Natomiast na Rysunku 5.2 zaprezentowano scenariusz przy gorszych warunkach oświetleniowych, w którym system nadal poprawnie wykrył piłkę czerwoną, jednak błędnie sklasyfikował obiekt o innej barwie jako piłkę.



Rysunek 5.1: Sytuacja przy dobrym oświetleniu — poprawne wykrycie piłki czerwonej oraz zignorowanie piłki o innych barwach.



Rysunek 5.2: Sytuacja przy złym oświetleniu — poprawne wykrycie piłki czerwonej oraz błędne wskazanie piłki o innych barwach.

Analiza wyników wykazała, że system wizyjny działa poprawnie w warunkach dobrego oświetlenia, precyzyjnie identyfikując piłkę czerwoną oraz ignorując inne obiekty. W takich warunkach algorytm wykazuje wysoką precyzję i skuteczność w wykrywaniu obiektu zgodnego z jego charakterystyką.

W przypadku słabego oświetlenia zauważono spadek skuteczności systemu. Algorytm błędnie klasyfikuje obiekt o podobnym kształcie jako piłkę, co może wynikać z ograniczeń modelu detekcji lub niewystarczającej jakości danych treningowych dla scen o niskim poziomie światła. Mimo to system nadal rozpoznaje piłkę czerwoną, co świadczy o jego częściowej odporności na zmienne warunki oświetleniowe.

DETECTED	FPS: 2.83	Pan: 16.26	Tilt: 30.00
DETECTED	FPS: 2.78	Pan: 13.01	Tilt: 30.00
DETECTED	FPS: 2.83	Pan: 11.05	Tilt: 30.00
DETECTED	FPS: 2.92	Pan: 10.23	Tilt: 30.00
DETECTED	FPS: 2.88	Pan: 9.95	Tilt: 30.00
DETECTED	FPS: 2.86	Pan: 9.83	Tilt: 30.00
DETECTED	FPS: 2.87	Pan: 9.69	Tilt: 30.00
DETECTED	FPS: 2.82	Pan: 9.76	Tilt: 30.00
DETECTED	FPS: 2.91	Pan: 9.82	Tilt: 30.00
DETECTED	FPS: 2.78	Pan: 9.88	Tilt: 30.00
DETECTED	FPS: 2.83	Pan: 10.08	Tilt: 30.00
DETECTED	FPS: 2.92	Pan: 10.28	Tilt: 30.00
DETECTED	FPS: 2.99	Pan: 10.47	Tilt: 30.00
DETECTED	FPS: 2.98	Pan: 10.66	Tilt: 30.00
DETECTED	FPS: 2.91	Pan: 10.88	Tilt: 30.00

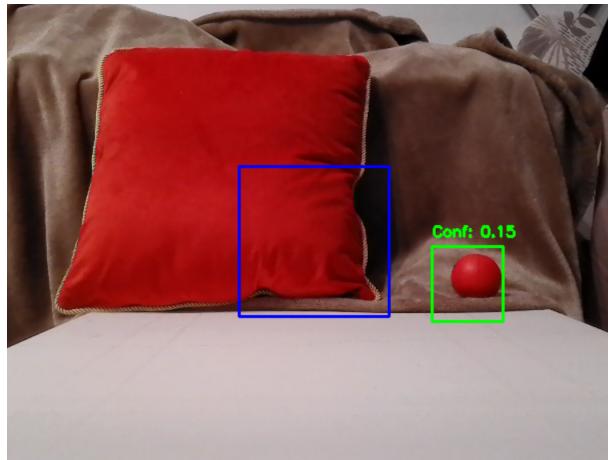
Rysunek 5.3: Zwrot informacji w terminalu o wykryciu piłki czerwonej

Rysunek 5.3 przedstawia widok terminala programu uruchomionego na Raspberry Pi. Jest to widok sytuacji, w której otoczenie jest dobrze oświetlone a program poprawnie wskazał czerwoną piłkę. W sytuacji gorszego oświetlenia i błędnej detekcji dwóch obiektów, program utrzymuje współrzędne serwomechanizmów dla pierwszego zidentyfikowanego obiektu. Ze względu na ograniczone zasoby mocy obliczeniowej Raspberry Pi, zdecydowano się na rezygnację z wyświetlania obrazu bezpośrednio na urządzeniu. Informacje o detekcji, takie jak status wykrycia obiektu "DETECTED"(oznaczające "Wykryto") oraz "SEARCHING"(oznaczające "Szukanie"), wartości kątów serwomechanizmów "Pan"(oznaczający "Obrót" wokół osi pionowej) i "Tilt"(oznaczający "Przechył" wokół osi pionowej) oraz liczba klatek na sekundę (FPS), są przesyłane do terminala komputera połączonego z Raspberry Pi przez SSH.

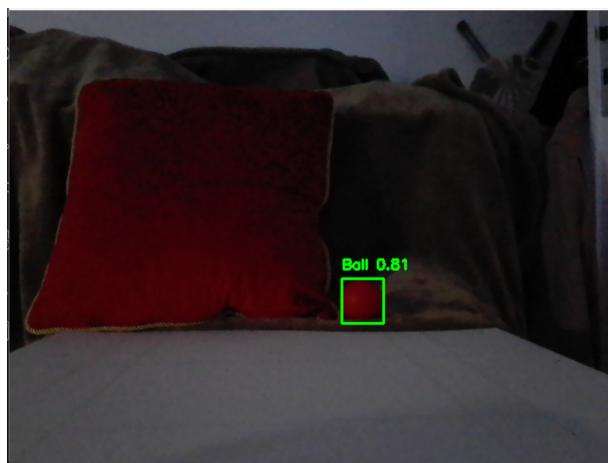
Warto zauważyć, że Rysunki 5.1 i 5.2 ilustrują działanie tego samego programu, lecz uruchomionego na komputerze w celu wizualizacji. Na obrazach widoczny jest niebieski kwadrat w centrum kadru, który wyznacza obszar uznawany za środek obrazu. Dzięki jego zastosowaniu program jest w stanie bardziej stabilnie śledzić pozycję obiektu.

W celu oceny skuteczności zaprojektowanego systemu wizyjnego przeprowadzono testy w różnych warunkach oświetleniowych oraz w obecności obiektów potencjalnie zakłócających detekcję piłki. Analiza wyników opierała się na obserwacji działania programu w scenariuszach symulujących rzeczywiste sytuacje. Na Rysunku 5.4 przedstawiono sytuację, w której system działał w warunkach dobrego oświetlenia. Detekcja piłki czerwonej została przeprowadzona prawidłowo, a inne obiekty, takie jak czerwona poduszka znajdująca się w kadrze, zostały skutecznie zignorowane. Wynik ten potwierdza, że mechanizm filtrowania wyników detekcji, oparty na *non-max suppression*, skutecznie eliminuje zbędne ramki detekcji.

Rysunek 5.5 prezentuje wyniki systemu w trudniejszych warunkach oświetleniowych, w których detekcja piłki czerwonej została przeprowadzona poprawnie, mimo ograniczonej widoczności. Co więcej, system zignorował obiekt o zbliżonym kolorze (czerwoną poduszkę), co świadczy o zdolności modelu do rozróżniania kluczowych cech obiektów.



Rysunek 5.4: Sytuacja przy dobrym oświetleniu - poprawne wykrycie piłki czerwonej oraz zignorowanie czerwonego obiektu.



Rysunek 5.5: Sytuacja przy złym oświetleniu - poprawne wykrycie piłki czerwonej oraz poprawne zignorowanie czerwonego obiektu.

DETECTED	FPS: 2.83	Pan: -8.13	Tilt: 24.07
DETECTED	FPS: 2.73	Pan: -8.20	Tilt: 24.09
DETECTED	FPS: 2.80	Pan: -8.27	Tilt: 24.14
SEARCHING	FPS: 2.83		
DETECTED	FPS: 2.82	Pan: -8.20	Tilt: 24.09
SEARCHING	FPS: 2.85		
DETECTED	FPS: 2.80	Pan: -8.14	Tilt: 23.90
DETECTED	FPS: 2.78	Pan: -8.06	Tilt: 23.85
DETECTED	FPS: 2.90	Pan: -8.10	Tilt: 23.89
DETECTED	FPS: 2.96	Pan: -8.06	Tilt: 23.94
DETECTED	FPS: 2.81	Pan: -7.99	Tilt: 23.91
DETECTED	FPS: 2.81	Pan: -7.94	Tilt: 23.88
DETECTED	FPS: 2.82	Pan: -7.89	Tilt: 23.83
DETECTED	FPS: 2.87	Pan: -7.86	Tilt: 23.83

Rysunek 5.6: Zwrot informacji w terminalu o wykryciu piłki czerwonej i zignorowaniu czerwonego obiektu

Rysunek 5.6 przedstawia widok terminala programu uruchomionego na Raspberry Pi. Jest to widok sytuacji, w której otoczenie jest dobrze oświetlone a program poprawnie wskazał czerwoną piłkę. Jednakże można zauważyć, iż program w pewnych momentach działania nie wykrywał modelu wcale i rozpoczął sekwencję poszukiwania.

## 5.3 Detekcja obiektu za pomocą binaryzacji i klasycznych technik przetwarzania obrazów

W podrozdziale tym przedstawiono podejście do detekcji obiektu (piłki czerwonej) z wykorzystaniem klasycznych technik przetwarzania obrazów, takich jak binaryzacja i analiza konturów. W przeciwnieństwie do metod opartych na zaawansowanych modelach uczenia maszynowego, zastosowane rozwiązanie opiera się na transformacjach kolorystycznych i filtracji, co pozwala na efektywną detekcję przy ograniczonych zasobach obliczeniowych dostępnych na platformie Raspberry Pi.

### 5.3.1 Opis działania programu

Program rozpoczyna swoje działanie od pobrania obrazu z kamery w czasie rzeczywistym, który następnie jest przetwarzany w celu wykrycia piłki czerwonej. Proces ten opiera się na kilku kluczowych krokach:

**Konwersja przestrzeni barw** Pierwszym krokiem jest konwersja obrazu z przestrzeni barw RGB na przestrzeń HSV (ang. *Hue, Saturation, Value*). Przestrzeń HSV umożliwia łatwiejszą separację kolorów, co jest szczególnie przydatne przy detekcji obiektów o specyficznej barwie, takich jak piłka czerwona. Konwersję realizuje funkcja `cv2.cvtColor`, której działanie przedstawiono w Kodzie 5.7.

---

<sup>1</sup> `hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)`

Kod 5.7: Konwersja obrazu z przestrzeni RGB do HSV.

**Binaryzacja obrazu** W celu wyodrębnienia obszarów odpowiadających kolorowi czerwonej piłki zastosowano binaryzację z użyciem dwóch zakresów wartości HSV. Zakresy te uwzględniają różne odcienie czerwieni występujące w przestrzeni barw HSV, co pozwala na dokładniejsze uchwycenie piłki. Fragment kodu odpowiedzialny za binaryzację przedstawiono w Kodzie 5.8.

---

<sup>1</sup> `lower_red1 = np.array([0, 120, 70])`  
<sup>2</sup> `upper_red1 = np.array([10, 255, 255])`  
<sup>3</sup> `lower_red2 = np.array([170, 120, 70])`  
<sup>4</sup> `upper_red2 = np.array([180, 255, 255])`  
<sup>5</sup>  
<sup>6</sup> `mask1 = cv2.inRange(hsv, lower_red1, upper_red1)`  
<sup>7</sup> `mask2 = cv2.inRange(hsv, lower_red2, upper_red2)`  
<sup>8</sup> `mask = cv2.add(mask1, mask2)`

Kod 5.8: Binaryzacja obrazu w oparciu o zakresy HSV.

Binaryzacja generuje maskę, w której piksele należące do zakresów koloru czerwonego mają wartość 1, a pozostałe - 0. Dzięki połączeniu dwóch zakresów (dolnego i górnego), algorytm uwzględnia różnice w odcieniach czerwieni, które mogą występować w różnych warunkach oświetleniowych.

**Analiza konturów** Na kolejnym etapie program identyfikuje kontury obiektów na binaryzowanej masce. Dla każdego z konturów obliczana jest powierzchnia oraz okrągłość, która pozwala odrzucić obiekty o kształtach znacznie różniących się od idealnego okręgu. Kryterium okrągłości obiektu opiera się na stosunku powierzchni konturu do pola powierzchni okręgu opisanego na konturze. Kod odpowiedzialny za analizę konturów przedstawiono w Kodzie 5.9.

---

```
1 contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.  
    CHAIN_APPROX_SIMPLE)  
  
2  
3 for contour in contours:  
4     area = cv2.contourArea(contour)  
5     if area > 300:  
6         ((x, y), radius) = cv2.minEnclosingCircle(contour)  
7         circle_area = np.pi * (radius ** 2)  
8         if 0.6 < area / circle_area < 1.4:  
9             ball_center_x = int(x)  
10            ball_center_y = int(y)  
11            detected = True
```

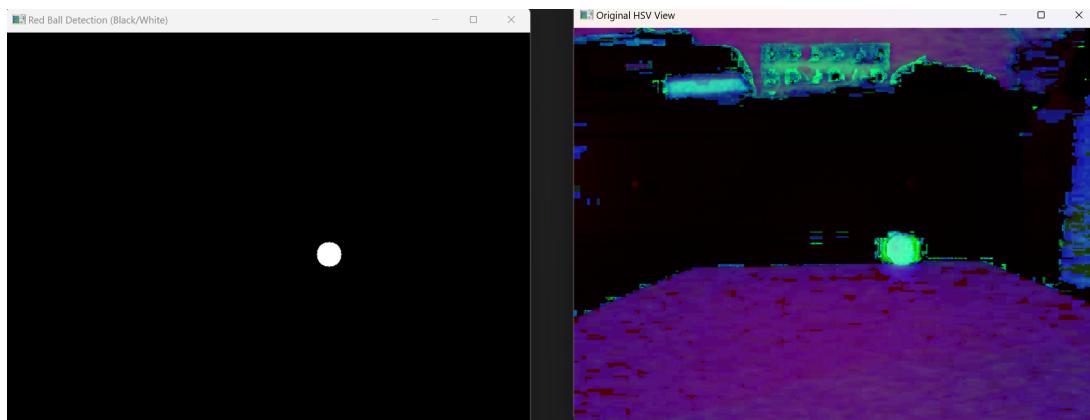
---

Kod 5.9: Analiza konturów w celu identyfikacji piłki.

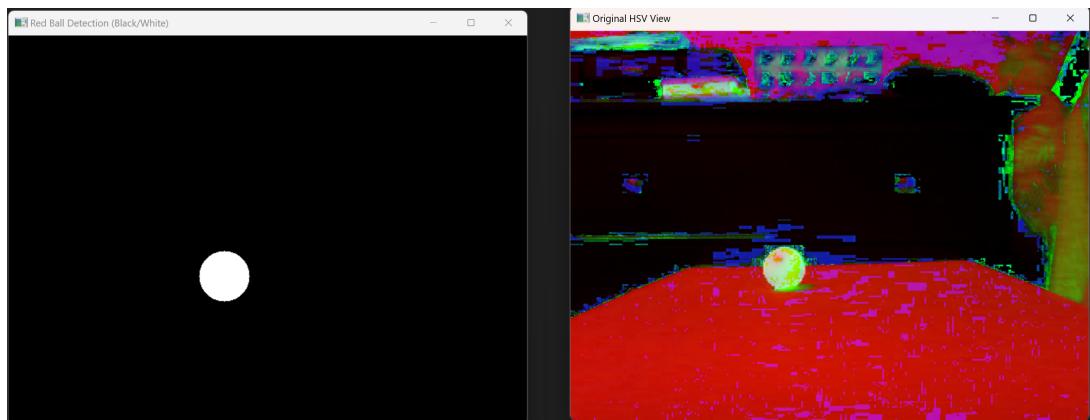
### 5.3.2 Uzyskane rezultaty

Wyniki działania algorytmu detekcji obiektu za pomocą binaryzacji zostały przedstawione w formie dwóch widoków dla każdego przypadku testowego. Pierwszy widok przedstawia obraz po zastosowaniu binaryzacji, gdzie piksele odpowiadające wykrytej barwie zostały zaznaczone na biało, a pozostałe wykluczone. Drugi widok prezentuje obraz w przestrzeni barw HSV, co umożliwia wizualną ocenę zakresów kolorów zastosowanych w procesie binaryzacji. Należy zaznaczyć, że wizualizacja ta została uruchomiona na komputerze, podczas gdy na Raspberry Pi wyświetlna jest jedynie informacja w terminalu o wykryciu bądź braku wykrycia obiektu.

Na Rysunku 5.7 zaprezentowano przykład wykrycia czerwonej piłki w dalekim kadrze przy słabym oświetleniu. Algorytm prawidłowo zidentyfikował piłkę, co dowodzi jego zdolności do poprawnej detekcji w trudnych warunkach oświetleniowych. Podobny przypadek przedstawiono na Rysunku 5.8, gdzie piłka została prawidłowo wykryta w dobrych warunkach oświetleniowych.



Rysunek 5.7: Sytuacja przy złym oświetleniu - poprawne wykrycie czerwonej piłki w dalekim kadrze



Rysunek 5.8: Sytuacja przy dobrym oświetleniu - poprawne wykrycie czerwonej piłki w dalekim kadrze

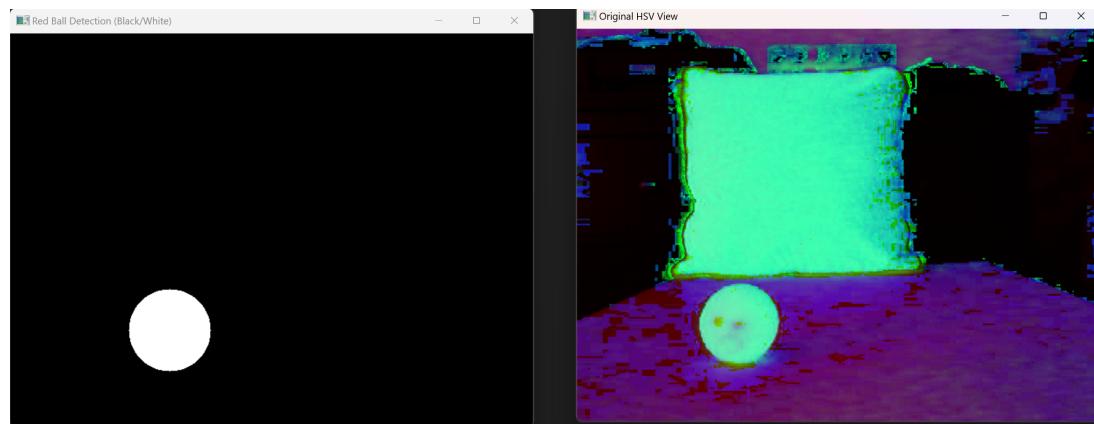
Należy również zauważyc, że podczas działania algorytmu na Raspberry Pi skuteczność detekcji była bardziej wrażliwa na warunki oświetleniowe. W momencie, gdy natężenie światła uległo znacznemu zmniejszeniu, program przestał wykrywać czerwoną piłkę. Na obrazie przedstawionym na Rysunku 5.9 widoczny jest moment, w którym światło zostało zgaszone, co spowodowało przejście informacji w terminalu z "DETECTED"(oznaczające "Wykryto") na "SEARCHING"(oznaczające "Szukanie").

Zjawisko to wskazuje na ograniczenia algorytmu wynikające z braku wystarczającej ilości informacji w warunkach słabego oświetlenia. Raspberry Pi, mimo zastosowania prostych i wydajnych metod takich jak binaryzacja, wykazuje ograniczenia w detekcji obiektów w trudnych warunkach oświetleniowych. Utrata detekcji w takich sytuacjach wynika głównie z ograniczeń samej metody binaryzacji, która opiera się wyłącznie na analizie kolorów, a nie z braku wystarczających zasobów obliczeniowych. Może to stanowić istotne wyzwanie w implementacjach wymagających niezawodnej pracy w zmiennych warunkach oświetleniowych.

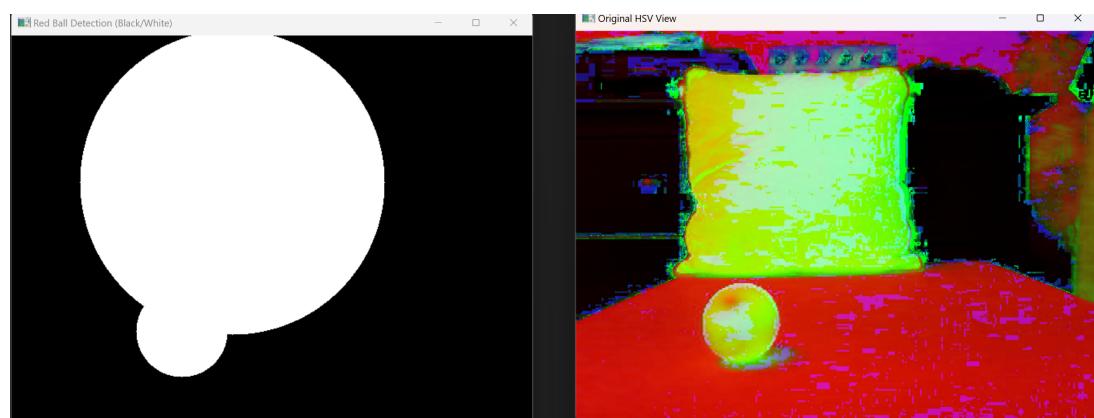
DETECTED		FPS: 26.13		Pan: 10.13		Tilt: 30.00
DETECTED		FPS: 26.63		Pan: 9.88		Tilt: 30.00
DETECTED		FPS: 25.95		Pan: 9.71		Tilt: 30.00
DETECTED		FPS: 25.11		Pan: 9.76		Tilt: 30.00
DETECTED		FPS: 27.12		Pan: 9.82		Tilt: 30.00
DETECTED		FPS: 26.26		Pan: 9.95		Tilt: 30.00
SEARCHING		FPS: 27.19				
SEARCHING		FPS: 25.65				
SEARCHING		FPS: 26.77				
SEARCHING		FPS: 23.32				
SEARCHING		FPS: 16.49				
SEARCHING		FPS: 16.24				
SEARCHING		FPS: 16.35				
SEARCHING		FPS: 14.20				
SEARCHING		FPS: 12.49				
SEARCHING		FPS: 12.12				
SEARCHING		FPS: 11.97				
SEARCHING		FPS: 11.96				
SEARCHING		FPS: 12.01				
SEARCHING		FPS: 11.91				

Rysunek 5.9: Informacja z terminalu RaspberryPi - zgubienie piłki w momencie zgaszenia światła

Rysunki 5.10 i 5.11 przedstawiają wyniki działania algorytmu w bliskim kadrze. W przypadku słabego oświetlenia (Rysunek 5.10), algorytm poprawnie zidentyfikował piłkę oraz zignorował inny obiekt o podobnym kolorze. Natomiast w lepszych warunkach oświetleniowych (Rysunek 5.11) zarejestrowano błędne wskazanie czerwonego obiektu znajdującego się w tle. Wynik ten wskazuje, że intensywne oświetlenie może zwiększyć prawdopodobieństwo fałszywych detekcji.



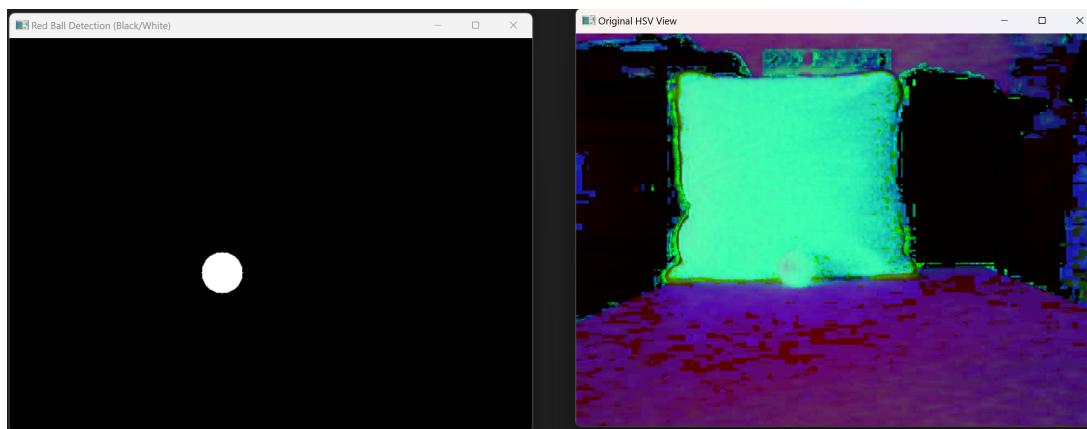
Rysunek 5.10: Sytuacja przy złym oświetleniu - poprawne wykrycie czerwonej piłki w bliskim kadrze oraz zignorowanie czerwonego obiektu



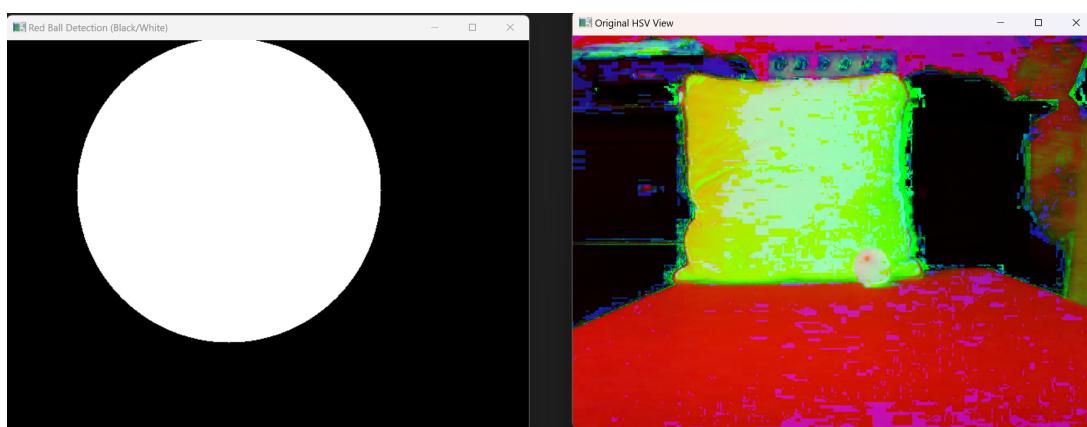
Rysunek 5.11: Sytuacja przy dobrym oświetleniu - poprawne wykrycie czerwonej piłki oraz błędne wskazanie czerwonego obiektu

Kolejny test, zilustrowany na Rysunku 5.12, pokazuje odporność algorytmu na zakłócenia w gorszych warunkach oświetleniowych. W warunkach ograniczonego oświetlenia algorytm poprawnie ignorował czerwone obiekty inne niż piłka (Rysunek 5.12), co dowodzi jego skuteczności w rozróżnianiu obiektów o podobnej barwie.

Jednakże, w sytuacji przedstawionej na Rysunku 5.13, przy dobrym oświetleniu, algorytm błędnie zidentyfikował poduszkę jako piłkę. W efekcie, rzeczywista piłka, znajdująca się na tle poduszki, została "zgubiona" w procesie detekcji. Ten przypadek ujawnia istotne ograniczenie algorytmu, wynikające z faktu, że analiza barwy nie jest w pełni wystarczającym kryterium identyfikacji obiektów, szczególnie w scenariuszach, gdzie różne obiekty o zbliżonym kolorze występują w tej samej scenie.



Rysunek 5.12: Sytuacja przy złym oświetleniu - poprawne wykrycie czerwonej piłki oraz zignorowanie czerwonego obiektu



Rysunek 5.13: Sytuacja przy dobrym oświetleniu - błędne wskazanie czerwonego obiektu zasłoniło piłkę

## 5.4 Porównanie skuteczności algorytmów

W niniejszym podrozdziale dokonano porównania trzech algorytmów detekcji obiektów: YOLOv7, YOLOv8 oraz algorytmu binaryzacji. Analiza została przeprowadzona w oparciu o różne warunki oświetleniowe i otoczenie, a także na dwóch platformach sprzętowych: komputerze osobistym oraz Raspberry Pi. Celem było określenie kompromisu pomiędzy szybkością działania, wyrażoną w klatkach na sekundę (FPS), a skutecznością w wykrywaniu obiektów oraz eliminacji błędnych detekcji.

Algorytmy z rodziny YOLO (ang. *You Only Look Once*) reprezentują podejście oparte na głębokich sieciach neuronowych, które umożliwia jednoczesne lokalizowanie i klasyfikowanie obiektów w obrazie. YOLOv7, jako starsza wersja, cechuje się dobrym kompromisem pomiędzy wydajnością obliczeniową a dokładnością detekcji. YOLOv8, będący nowszym rozwinięciem, oferuje jeszcze większą precyzję i lepsze dostosowanie do bardziej złożonych scen, ale kosztem większych wymagań sprzętowych.

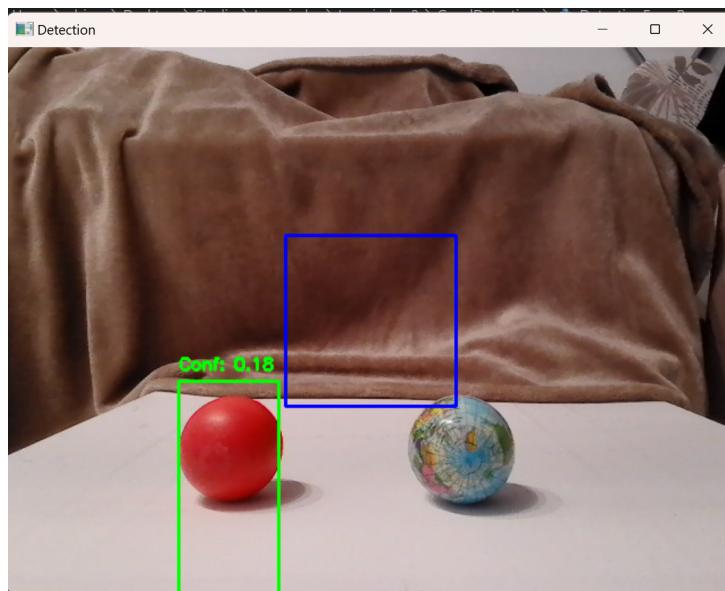
Z kolei algorytm binaryzacji, oparty na segmentacji obrazu w przestrzeni barw HSV, charakteryzuje się bardzo wysoką szybkością działania i niskimi wymaganiemami sprzętowymi. Jego prostota sprawia jednak, że jest bardziej podatny na błędy w trudnych warunkach, takich jak niejednorodne oświetlenie lub obecność obiektów o podobnym kolorze.

W kontekście badania, platformy sprzętowe różniły się znacząco pod względem dostępnych zasobów obliczeniowych. Komputer osobisty, wyposażony w nowoczesny procesor i dedykowaną kartę graficzną, umożliwiał działanie algorytmów YOLO w czasie rzeczywistym z wysoką dokładnością. Raspberry Pi, jako platforma o ograniczonej mocy obliczeniowej, stanowiło wyzwanie dla zaawansowanych algorytmów detekcji, ale dzięki swojej kompaktowej konstrukcji i energooszczędności jest często stosowane w systemach wbudowanych.

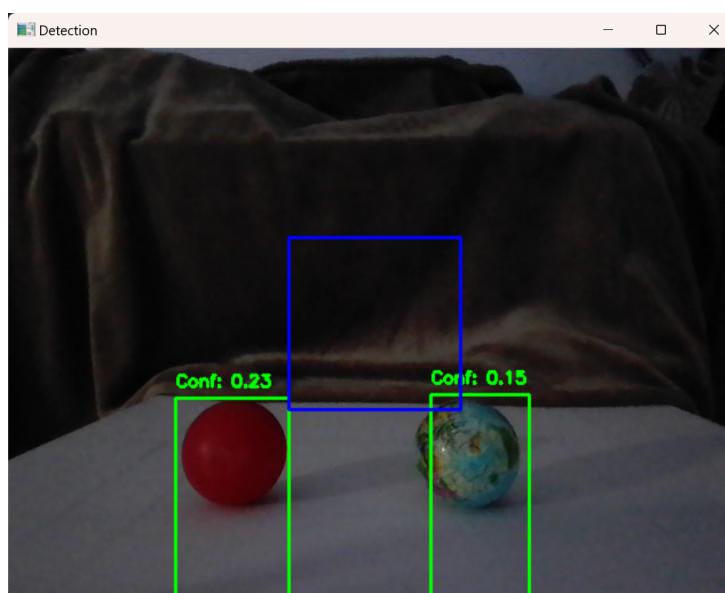
Wyniki analizy uwzględniają różne scenariusze testowe, takie jak działanie algorytmów w warunkach dobrego i słabego oświetlenia, obecność przeszkód oraz różne konfiguracje sprzętowe. Pozwoliło to na kompleksowe porównanie skuteczności i efektywności rozważanych podejść, a także ocenę ich przydatności w systemach czasu rzeczywistego.

### 5.4.1 Charakterystyka modelu YOLOv7

W warunkach dobrego oświetlenia model YOLOv7 wykazuje wysoką skuteczność w detekcji obiektów, co potwierdzają wyniki przedstawione na Rysunku 5.14. Algorytm prawidłowo zidentyfikował piłkę czerwoną, jednocześnie ignorując inne obiekty znajdujące się w kadrze. Kluczowym punktem tego testu było sprawdzenie czy model jest w stanie zignorować obiekty o podobnym kształcie lub barwie do śledzonej piłki, co zostało potwierdzone tylko dla przypadku dobrego oświetlenia. Niestety w warunkach słabego oświetlenia, model YOLOv7 błędnie sklasyfikował obiekt o innej barwie jako piłkę, co przedstawiono na Rysunku 5.15.

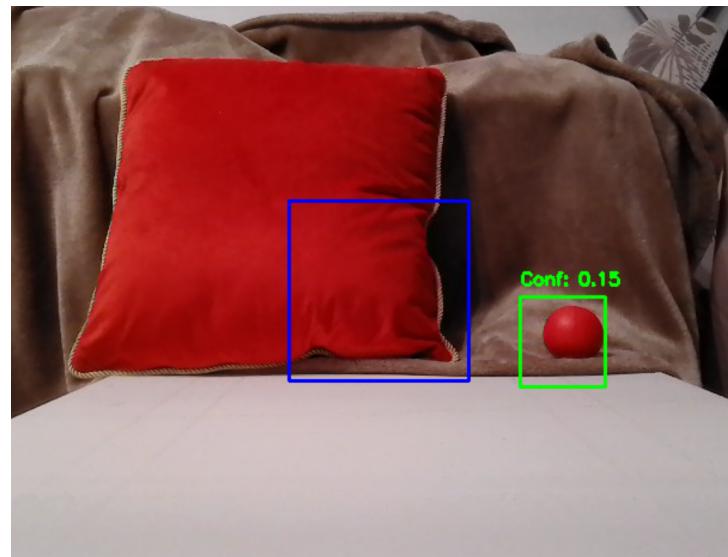


Rysunek 5.14: Działanie modelu YOLOv7 przy dobrym oświetleniu.

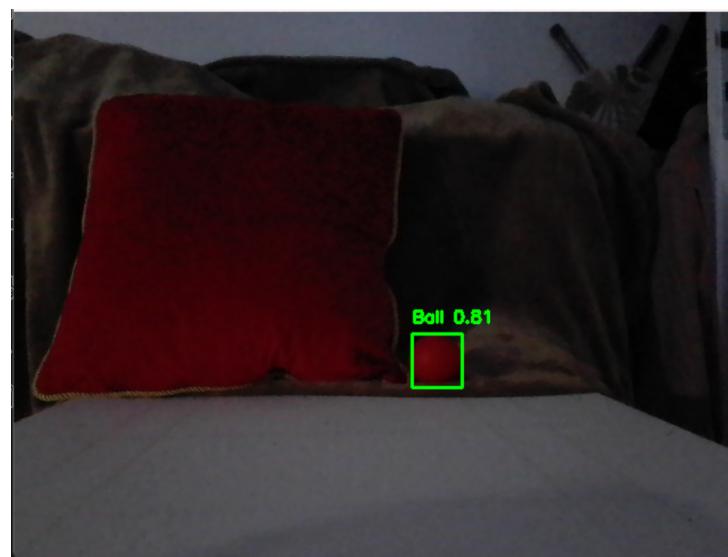


Rysunek 5.15: Działanie modelu YOLOv7 przy złym oświetleniu.

Model YOLOv7 wykazuje zdolność do prawidłowego pomijania obiektów, które nie spełniają kryteriów detekcji. Na Rysunku 5.16 zaprezentowano przykład działania algorytmu przy dobrym oświetleniu, gdzie obiekt w postaci czerwonej poduszki został prawidłowo pominiety. Analogicznie, w warunkach słabego oświetlenia, YOLOv7 również jest w stanie skutecznie ignorować niepożądane obiekty, co ilustruje Rysunek 5.17.



Rysunek 5.16: Działanie modelu YOLOv7 przy dobrym oświetleniu oraz prawidłowe pominiecie obiektu.

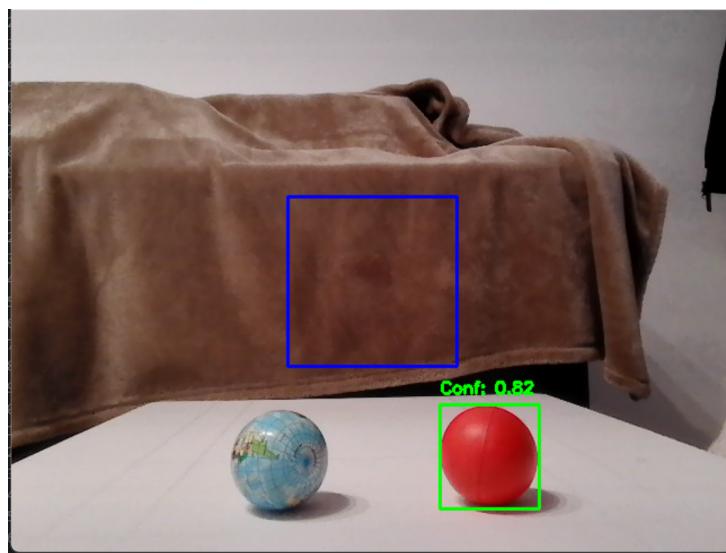


Rysunek 5.17: Działanie modelu YOLOv7 przy złym oświetleniu oraz prawidłowe pominiecie obiektu.

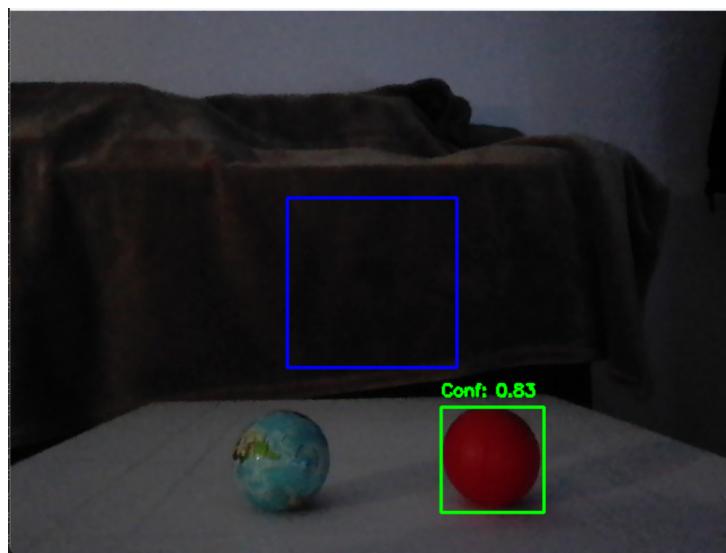
### 5.4.2 Charakterystyka modelu YOLOv8

YOLOv8, będący najnowszą wersją rodziny YOLO, charakteryzuje się najwyższą precyją detekcji spośród analizowanych algorytmów. W warunkach dobrego oświetlenia algorytm osiąga doskonałe wyniki, co zostało przedstawione na Rysunku 5.18, gdzie obiekt został prawidłowo zidentyfikowany.

W trudniejszych warunkach, takich jak gorsze oświetlenie, YOLOv8 wciąż jest w stanie poprawnie wykrywać obiekt o ściśle określonych cechach. Przykład działania algorytmu w takich warunkach zaprezentowano na Rysunku 5.19.



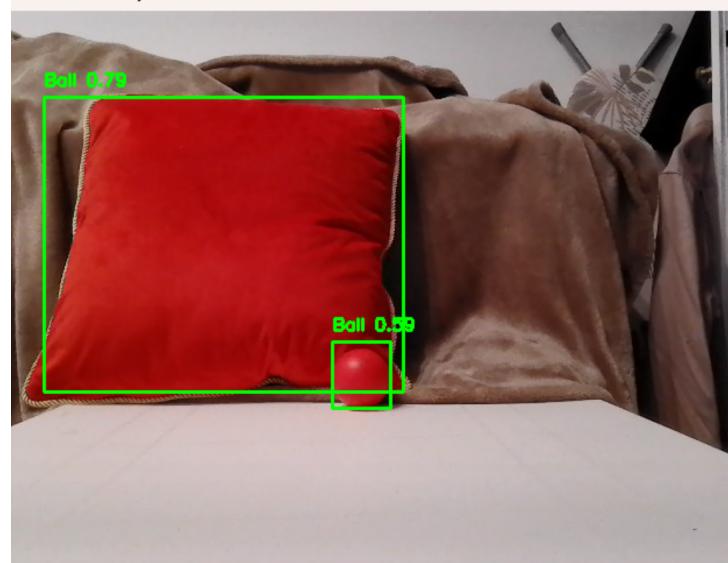
Rysunek 5.18: Działanie modelu YOLOv8 przy dobrym oświetleniu.



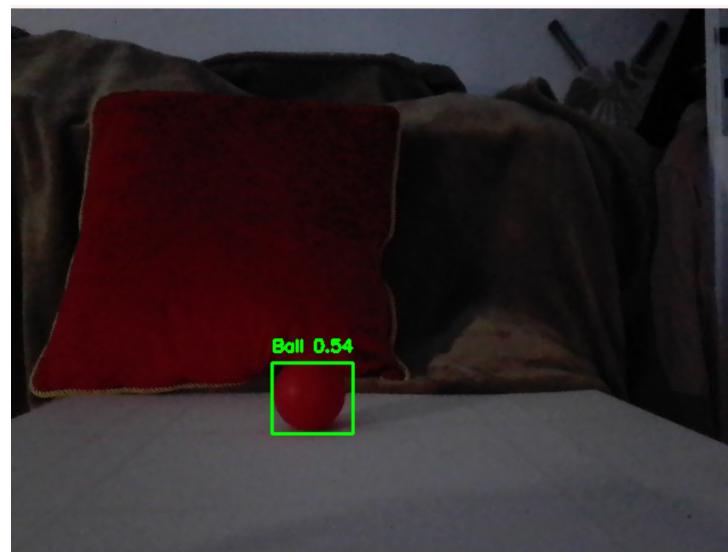
Rysunek 5.19: Działanie modelu YOLOv8 przy gorszym oświetleniu.

Pomimo ogólnie wysokiej dokładności, YOLOv8 czasami wykazuje błędne detekcje. Rysunek 5.20 przedstawia przykład, w którym w warunkach dobrego oświetlenia algorytm błędnie zidentyfikował czerwoną poduszkę jako obiekt piłkopodobny.

Z kolei Rysunek 5.21 pokazuje przypadek, w którym algorytm prawidłowo pominął obiekt w warunkach słabego oświetlenia. Może to być skutkiem zaniku barwy czerwonej w wyniku pogorszenia światła otoczenia.



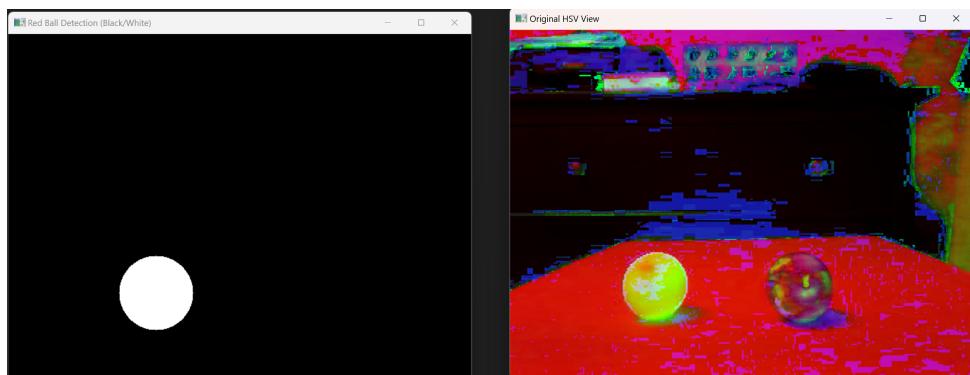
Rysunek 5.20: Działanie modelu YOLOv8 przy dobrym oświetleniu oraz błędne wykrycie obiektu.



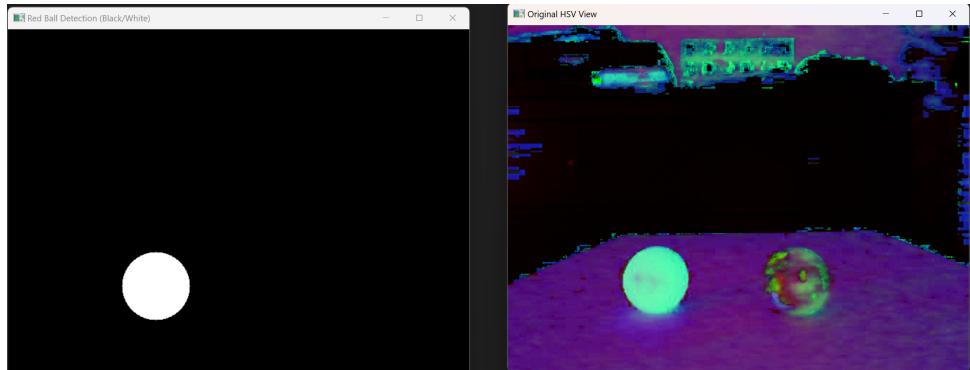
Rysunek 5.21: Działanie modelu YOLOv8 przy gorszym oświetleniu oraz prawidłowe po-minięcie obiektu.

### 5.4.3 Charakterystyka algorytmu binaryzacji

Algorytm binaryzacji opiera się na prostej segmentacji obrazu w przestrzeni barw HSV, co zapewnia mu wyjątkową szybkość działania – na Raspberry Pi osiąga nawet 25-30 FPS. Dzięki tej prostocie algorytm jest szczególnie efektywny w warunkach dobrego oświetlenia, jak pokazano na Rysunku 5.22. W takich przypadkach binaryzacja umożliwia poprawne wykrycie obiektu, co potwierdza jej skuteczność w sprzyjających warunkach. W warunkach słabego oświetlenia, przedstawionych na Rysunku 5.23, algorytm nadal jest w stanie poprawnie zidentyfikować obiekt.

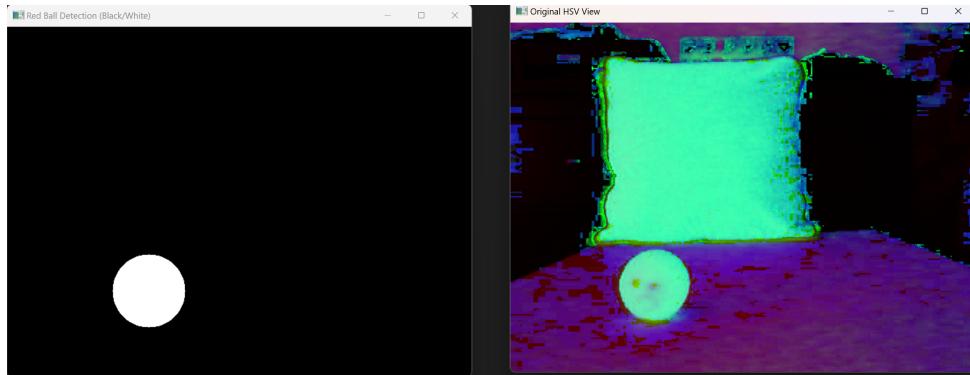


Rysunek 5.22: Działanie binaryzacji w warunkach dobrego oświetlenia.

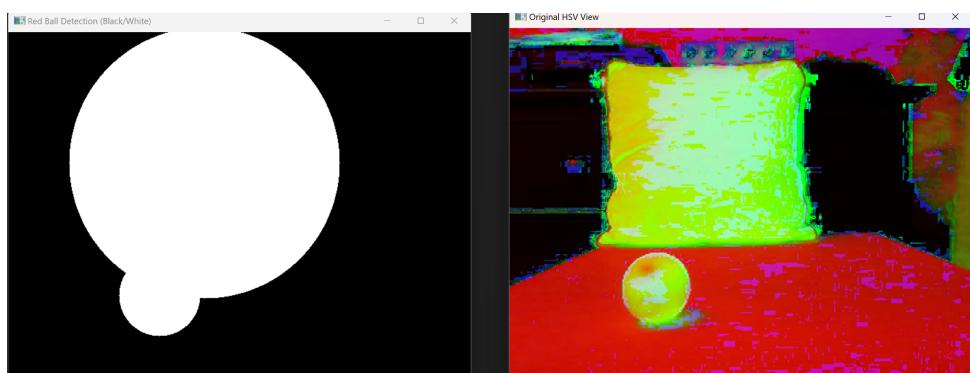


Rysunek 5.23: Działanie binaryzacji w warunkach słabego oświetlenia.

Jednakże, w bardziej złożonych sytuacjach, binaryzacja ujawnia swoje ograniczenia, co zilustrowano na Rysunkach 5.24 i 5.25. W szczególności, w sytuacjach, gdy kolor obiektu zbliża się do kolorystyki tła, algorytm może generować fałszywe detekcje lub błędnie klasyfikować obiekty.



Rysunek 5.24: Działanie binaryzacji w warunkach słabego oświetlenia oraz prawidłowe zignorowanie obiektu.



Rysunek 5.25: Działanie binaryzacji w warunkach słabego oświetlenia oraz błędna detekcja obiektu.

Pomimo swoich ograniczeń, algorytm binaryzacji wyróżnia się wysoką szybkością działania, co czyni go atrakcyjnym rozwiązaniem w systemach o ograniczonych zasobach obliczeniowych, takich jak Raspberry Pi. Niemniej jednak, jego skuteczność zależy w dużej mierze od precyzyjnego doboru zakresów wartości HSV oraz warunków otoczenia, co stanowi istotny element w jego implementacji.

### 5.4.4 Wnioski

**Działanie na komputerze osobistym** Jak przedstawiono w tabelach 5.1 i 5.2, algorytmy detekcji różniły się skutecznością w zależności od warunków oświetleniowych i obecności przeszkód.

Algorytm **YOLOv7** osiągnął dobrą równowagę między szybkością a dokładnością, niezależnie od warunków oświetleniowych i przeszkód, stabilnie działając z prędkością 16.09 FPS. **YOLOv8** wyróżniał się najwyższą dokładnością, jednak kosztem niższej szybkości, osiągając 10.71 FPS. W przypadku scen z przeszkodami miał trudności z ich ignorowaniem w dobrym oświetleniu, co wskazuje na większą czułość detektora. **Binaryzacja**, choć najszybsza z algorytmów (29.61 FPS), miała trudności z prawidłową klasyfikacją obiektów w złożonych scenach, co może prowadzić do fałszywych detekcji.

Tabela 5.1: Wyniki algorytmów detekcji obiektu bez przeszkody w różnych warunkach oświetleniowych na komputerze osobistym.

Algorytm	Dobre oświetlenie		Złe oświetlenie	
	Czy wykryto	FPS	Czy wykryto	FPS
YOLOv7	TAK	16.09	TAK	16.09
YOLOv8	TAK	10.71	TAK	10.71
Binaryzacja	TAK	29.61	TAK	29.61

Tabela 5.2: Wyniki algorytmów detekcji obiektu z przeszkodą w różnych warunkach oświetleniowych na komputerze osobistym.

Algorytm	Dobre oświetlenie		Złe oświetlenie	
	Czy pominął przeszkodę	FPS	Czy pominął przeszkodę	FPS
YOLOv7	TAK	16.09	TAK	16.09
YOLOv8	NIE	10.71	TAK	10.71
Binaryzacja	NIE	29.61	TAK	29.61

**Działanie na Raspberry Pi** Jak przedstawiono w tabelach 5.3 i 5.4, działanie algorytmów na Raspberry Pi było ograniczone przez mniejsze zasoby obliczeniowe.

**YOLOv7** działał stabilnie z prędkością około 2.8 FPS, zachowując swoją skuteczność w ignorowaniu przeszkód oraz wykrywaniu obiektów w różnych warunkach oświetleniowych. **YOLOv8**, choć nadal dokładny, wykazał najniższą wydajność na Raspberry Pi, osiągając zaledwie 1.24 FPS. Algorytm **binaryzacji**, pomimo najwyższej szybkości działania (25.95 FPS w dobrym oświetleniu), miał trudności w detekcji obiektów w słabym oświetleniu, co ujawnia jego ograniczenia w bardziej wymagających warunkach.

Tabela 5.3: Wyniki algorytmów detekcji obiektu bez przeszkody w różnych warunkach oświetleniowych na Raspberry Pi.

Algorytm	Dobre oświetlenie		Złe oświetlenie	
	Czy wykryto	FPS	Czy wykryto	FPS
YOLOv7	TAK	2.80	TAK	2.70
YOLOv8	TAK	1.24	TAK	1.24
Binaryzacja	TAK	25.95	NIE	12.49

Tabela 5.4: Wyniki algorytmów detekcji obiektu z przeszkodą w różnych warunkach oświetleniowych na Raspberry Pi.

Algorytm	Dobre oświetlenie		Złe oświetlenie	
	Czy pominął przeszkodę	FPS	Czy pominął przeszkodę	FPS
YOLOv7	TAK	2.80	TAK	2.70
YOLOv8	NIE	1.24	NIE	1.24
Binaryzacja	NIE	25.95	TAK	12.49

Podsumowując, wybór algorytmu powinien być uzależniony od specyficznych wymagań aplikacji oraz dostępnych zasobów sprzętowych:

- **YOLOv7** jest najlepszym wyborem dla aplikacji wymagających równowagi między dokładnością a szybkością, szczególnie na urządzeniach o ograniczonej mocy obliczeniowej, takich jak Raspberry Pi.
- **YOLOv8** nadaje się do aplikacji, gdzie priorytetem jest maksymalna dokładność, a ograniczenia sprzętowe są mniej istotne, np. na komputerach osobistych.
- **Binaryzacja** jest idealnym rozwiązaniem dla systemów wymagających bardzo szybkiego przetwarzania w czasie rzeczywistym, pod warunkiem, że scena jest dobrze oświetlona i nie zawiera przeszkód o podobnym kolorze.

# Rozdział 6

## Problematyka i rozwiązania

W niniejszym rozdziale przedstawiono i opisano problemy związane z realizacją projektu oraz zaproponowane rozwiązania, które pozwoliły na skuteczne zaimplementowanie systemu wizyjnego dla robota mobilnego. Omówiono wyzwania związane z wyborem algorytmów detekcji obiektów, implementacją algorytmów śledzenia obiektu oraz elektroniką i mechaniką robota.

### 6.1 Wybór algorytmów detekcji obiektów

Pierwszym wyzwaniem w realizacji projektu było podjęcie decyzji dotyczącej wyboru modelu głębokiego uczenia oraz przygotowania odpowiedniego zbioru danych do treningu. Proces rozpoczęto od instalacji aplikacji do oznaczania i etykietowania obrazów (*labelImg*), która miała na celu ułatwienie przygotowania danych wejściowych dla modelu. Problemem okazało się jednak prawidłowe przygotowanie plików, w szczególności plików tekstowych (**.txt**) zawierających etykiety pozycji piłki na obrazie.

Pierwotnym założeniem było wytrenowanie modelu YOLO na komputerze osobistym. Podczas przygotowań napotkano trudności związane z konfiguracją bibliotek i ich integracją. Model YOLO nie inicjował procesu treningowego. W efekcie podjęto próbę zastosowania alternatywnego rozwiązania w postaci TensorFlow. Niestety, wymagało to stworzenia własnego programu zdolnego do rozpoczęcia treningu sieci neuronowej, co okazało się zbyt skomplikowane i nie przyniosło oczekiwanych rezultatów.

Przełom nastąpił dzięki odkryciu platformy Roboflow, która umożliwia intuicyjne etykietowanie obrazów i przygotowanie zestawu danych. Obrazy zostały wygenerowane za pomocą prostego programu w Pythonie, a następnie załadowane na platformę, gdzie zostały oznaczone. Proces przygotowania zbioru danych, w zależności od liczby zdjęć, trwał od kilkunastu do kilkudziesięciu minut. Po zakończeniu etykietowania wybrano format pliku YOLOv7 z PyTorch do pobrania.

Aby przyspieszyć proces treningu sieci, zdecydowano się na wykorzystanie Google Colab, który umożliwia dostęp do jednostek GPU. Po wytrenowaniu modelu pobrano plik

`best.pt`. Próba jego uruchomienia początkowo nie powiodła się z powodu brakujących folderów `utils` oraz `models`, w tym pliku `experimental.py`. Dopiero po odpowiedniej konfiguracji środowiska i odniesieniu się do właściwych ścieżek udało się uruchomić model, ale wyłącznie na komputerze osobistym.

Kolejnym wyzwaniem było dostosowanie modelu do pracy w środowisku Raspberry Pi, gdzie zasoby sprzętowe są ograniczone. YOLOv7 działał na tej platformie z prędkością zaledwie 3 FPS, co okazało się niewystarczające. Podjęto próbę wytrenowania nowego modelu YOLOv8, który następnie przekonwertowano przy użyciu ONNX na TensorFlow Lite. Konwersja spowodowała jednak znaczne straty w jakości, a robot nie był w stanie skutecznie wykrywać piłki, mimo że program działał z prędkością 10 FPS. Dla porównania, uruchomienie YOLOv8 bez konwersji dawało jedynie 0.9–1.2 FPS, co było najgorszym wynikiem spośród testowanych rozwiązań.

Ostatecznie zdecydowano się pozostać przy modelu YOLOv7, pomimo ograniczonej prędkości 3 FPS. Rozważano również zastosowanie starszych wersji YOLO, które mogłyby lepiej sprawdzić się w środowisku Raspberry Pi, ale nie podjęto dalszych działań w tym kierunku. Wynik ten podkreśla znaczenie kompromisu między dokładnością, szybkością działania a ograniczeniami sprzętowymi.

## 6.2 Implementacja algorytmów śledzenia obiektu

Jednym z największych wyzwań podczas realizacji projektu było zaimplementowanie wytrenowanego modelu, przygotowanego na komputerze osobistym z systemem Windows, na platformie Raspberry Pi.

Pierwszym napotkanym problemem był błąd przy instalacji biblioteki PyTorch. Kluczowe okazało się przejście na 64-bitową wersję systemu Raspberry Pi OS, ponieważ wersja 32-bitowa nie była kompatybilna z wymaganą wersją PyTorch. Po zainstalowaniu odpowiedniej wersji systemu i biblioteki, przystąpiono do przesłania wszystkich niezbędnych plików na Raspberry Pi za pomocą protokołu SSH.

Kolejnym wyzwaniem było uruchomienie programu. Po próbie uruchomienia detekcji na Raspberry Pi pojawił się błąd związany z wczytywaniem modelu. Okazało się, że problem wynikał z funkcji `attempt_download(w)` w pliku `experimental.py`. Funkcja ta jest odpowiedzialna za automatyczne pobieranie brakujących wag modelu z sieci. Jednak na Raspberry Pi, z uwagi na brak potrzeby automatycznego pobierania wag, usunięto tę linię kodu.

Zmodyfikowany fragment pliku wygląda następująco:

---

<sup>1</sup> `attempt_download(w)`

Kod 6.1: Usunięcie funkcji `attempt_download` z pliku `experimental.py`.

Zmiana ta pozwoliła na poprawne wczytanie lokalnie przechowywanych wag modelu bez próby ich ponownego pobrania z sieci.

Po tej modyfikacji program udało się uruchomić, jednak pojawił się kolejny problem związany z wyświetlaniem obrazu. Początkowo zakładano podgląd wideo z kamery na komputerze za pomocą VNC. Okazało się jednak, że jednoczesne użycie bibliotek `libcamera` i `OpenCV` powodowało konflikty, uniemożliwiając wyświetlenie okna podglądu wideo w czasie rzeczywistym. Zdecydowano się więc zrezygnować z podglądu obrazu, pozostając jedynie przy wyświetlaniu w konsoli informacji o wykryciu obiektu, liczbie klatek na sekundę (FPS) oraz aktualnej pozycji serwomechanizmów.

Dzięki tym rozwiązaniom możliwe było efektywne uruchomienie algorytmu detekcji na Raspberry Pi, mimo ograniczeń związanych z wydajnością i kompatybilnością środowiska.

### 6.3 Problemy związane z elektroniką i zasilaniem

Jednym z kluczowych wyzwań projektu było zaprojektowanie odpowiedniego systemu zasilania dla Raspberry Pi. Początkowym założeniem było wykorzystanie dwóch ogniw o łącznym napięciu 7.2V jako źródła zasilania. Z uwagi na fakt, że Raspberry Pi może przyjąć maksymalne napięcie wynoszące 5V, konieczne było zastosowanie przetwornicy typu ang. *step-down*, której zadaniem jest obniżenie napięcia wejściowego do bezpiecznego poziomu wyjściowego.

Po podłączeniu dwóch ogniw do wejścia przetwornicy oraz kabla USB-C do jej wyjścia, uzyskano stabilne napięcie 5V, co pozwoliło na uruchomienie Raspberry Pi. Problem pojawił się jednak przy próbie uruchomienia kamery. Okazało się, że natężenie prądowe dostarczane przez przetwornicę było zbyt niskie, aby zasilić zarówno system, jak i moduł kamery. Kamera nie była w stanie poprawnie się zainicjalizować.

Podjęto próbę zastosowania przetwornicy o wyższej wydajności prądowej, zdolnej teoretycznie dostarczyć do 3A. Niestety, rezultat był identyczny – natężenie nadal okazywało się niewystarczające, aby w pełni zasilić wszystkie komponenty systemu.

Ostatecznie, jako rozwiązanie problemu, zdecydowano się na zastosowanie powerbanku jako źródła zasilania dla Raspberry Pi. Powerbank oferujący napięcie 5V i natężenie 3A idealnie sprawdził się w tej roli, zapewniając stabilne zasilanie dla Raspberry Pi, modułu kamery oraz dwóch serwomechanizmów. Takie rozwiązanie pozwoliło na bezproblemową pracę całego systemu i wyeliminowało problemy związane z niedoborem natężenia prądu.

## 6.4 Problemy związane z mechaniką i sterowaniem

Aby robot mobilny mógł skutecznie spełniać swoje zadanie, istotne było zaimplementowanie modułu śledzącego obiekt. Początkowo zakładano, że kamera będzie statycznie umieszczona w jednym miejscu, a śledzenie czerwonej piłki będzie realizowane poprzez obrót całej platformy jeżdżącej. Pomyśl ten okazał się nieefektywny i niepraktyczny, dlatego zdecydowano się na alternatywne rozwiązanie, polegające na poruszaniu samą kamerą.

Do realizacji tego zadania wykorzystano uchwyt na kamerę zawierający dwa serwo-mechanizmy: jeden odpowiedzialny za obrót kamery wokół osi pionowej (*pan*), a drugi za obrót wokół osi poziomej (*tilt*). Dzięki temu rozwiązaniu kamera mogła dynamicznie śledzić ruch obiektu, zwiększając swoje pole widzenia.

**Obrót obrazu i sterowanie serwomechanizmami** Z uwagi na mechaniczne zamocowanie kamery, obraz z kamery był obrócony o  $180^\circ$ . Wymagało to dodatkowego przekształcenia obrazu w programie poprzez funkcję `cv2.rotate(frame, cv2.ROTATE_180)`. Kolejnym wyzwaniem było dostosowanie ruchu serwomechanizmów do nowej orientacji obrazu.

Aby program mógł skutecznie śledzić obiekt, w centrum obrazu umieszczono kwadrat, który służył jako rozszerzone centrum obrazu. Jeżeli obiekt znajdował się w granicach tego kwadratu, kamera nie wykonywała zbędnych korekt, co zapobiegało niestabilności wynikającej z drgań lub zmiennego oświetlenia. Jeżeli jednak piłka przesuwała się poza centrum obrazu, obliczano przesunięcie względem środka według wzoru:

$$\Delta x = x_{\text{środek}} - x_{\text{piłka}}, \quad \Delta y = y_{\text{środek}} - y_{\text{piłka}}, \quad (6.1)$$

gdzie  $x_{\text{środek}}$ ,  $y_{\text{środek}}$  to współrzędne środka obrazu, a  $x_{\text{piłka}}$ ,  $y_{\text{piłka}}$  to współrzędne środka wykrytego obiektu.

Na podstawie tych wartości obliczano kąty wychylenia serwomechanizmów w sposób proporcjonalny:

$$\theta_{\text{pan}} = \theta_{\text{pan}} + k_x \cdot \Delta x, \quad \theta_{\text{tilt}} = \theta_{\text{tilt}} + k_y \cdot \Delta y, \quad (6.2)$$

gdzie  $k_x$  i  $k_y$  to współczynniki skalujące, które określają szybkość korekcji.

**Zależność ruchu kamery i silników DC** Zastosowanie serwomechanizmów umożliwiło uzależnienie ruchów silników DC od kątów wychylenia kamery:

- Robot pozostawał nieruchomy, gdy kąt nachylenia serwomechanizmu pochyłu (*tilt*) osiągał granicę  $-30^\circ$ , co oznaczało, że piłka znajdowała się zbyt blisko.
- Robot jechał do przodu, gdy serwomechanizm *tilt* wskazywał na wprost ( $0^\circ$ ).

- Robot wykonywał skręt w prawo lub w lewo, gdy serwomechanizm *pan* osiągał granicę  $\pm 25^\circ$ .

**Mechanizm poszukiwania obiektu** W przypadku, gdy obiekt znajdował się poza polem widzenia kamery, zaimplementowano funkcję poszukiwawczą, w której serwomechanizmy wykonywały ruch eliptyczny w przestrzeni. Kąty obrotu serwomechanizmów były generowane na podstawie funkcji trygonometrycznych:

$$\theta_{\text{pan}}(t) = A_{\text{pan}} \cdot \cos(\omega t), \quad (6.3)$$

$$\theta_{\text{tilt}}(t) = A_{\text{tilt}} \cdot \sin(\omega t) + \theta_{\text{offset}}, \quad (6.4)$$

gdzie:

- $A_{\text{pan}}$ ,  $A_{\text{tilt}}$  to amplitudy ruchu w płaszczyznach *pan* i *tilt*,
- $\omega$  to prędkość kątowa,
- $\theta_{\text{offset}}$  to stałe przesunięcie kąta dla serwomechanizmu *tilt*.

Dzięki tym mechanizmom robot mógł autonomicznie poszukiwać obiektu, minimalizując czas potrzebny na odzyskanie obiektu w polu widzenia.



# Rozdział 7

## Podsumowanie i wnioski

W ramach niniejszej pracy zrealizowano projekt systemu wizyjnego dla robota mobilnego, którego zadaniem było autonomiczne wykrywanie i śledzenie obiektów. W projekcie wykorzystano zarówno zaawansowane algorytmy głębokiego uczenia (YOLOv7 i YOLOv8), jak i klasyczne techniki przetwarzania obrazów, takie jak binaryzacja i analiza konturów. System został przeanalizowany pod kątem skuteczności i wydajności w różnych warunkach oświetleniowych oraz na dwóch platformach sprzętowych: komputerze osobistym i Raspberry Pi.

Przeprowadzone testy wykazały, że każde z zastosowanych podejść posiada swoje specyficzne mocne i słabe strony. Modele YOLO zapewniły wysoką precyzję detekcji w złożonych warunkach, jednak ich działanie było ograniczone przez zasoby obliczeniowe modułu Raspberry Pi. Z kolei algorytm binaryzacji charakteryzował się znacznie wyższą szybkością, ale jego skuteczność w trudniejszych warunkach była ograniczona.

### 7.1 Wnioski

Na podstawie przeprowadzonych badań można sformułować następujące wnioski:

- Modele YOLOv7 i YOLOv8 są odpowiednie do zastosowań wymagających wysokiej precyzji detekcji, szczególnie w złożonych środowiskach. Ich głównym ograniczeniem jest wysokie zapotrzebowanie na zasoby sprzętowe.
- Algorytm binaryzacji jest wydajnym rozwiązaniem w prostych scenariuszach, gdzie kluczowa jest szybkość działania, a zasoby obliczeniowe są ograniczone.
- Raspberry Pi, pomimo swoich ograniczeń sprzętowych, może być skutecznie wykorzystywane jako platforma do detekcji obiektów w czasie rzeczywistym, szczególnie przy zastosowaniu zoptymalizowanych algorytmów.

## 7.2 Perspektywy dalszych badań

W przyszłości można rozważyć następujące kierunki rozwoju systemu:

- Optymalizację modeli YOLO pod kątem działania na platformach o ograniczonych zasobach obliczeniowych, na przykład poprzez kompresję modelu lub przeniesienie go na TensorFlow Lite.
- Rozbudowanie zbioru danych treningowych o bardziej różnorodne scenariusze oświetleniowe i środowiskowe, co mogłoby zwiększyć uniwersalność algorytmów.
- Integrację segmentacji semantycznej jako uzupełnienia obecnych metod detekcji w celu poprawy skuteczności systemu w bardziej złożonych środowiskach.
- Wdrożenie hybrydowego podejścia, które łączy algorytmy głębokiego uczenia z klasycznymi technikami przetwarzania obrazu, aby wykorzystać ich wzajemne zalety.

Podsumowując, przeprowadzona analiza i testy potwierdziły, że odpowiedni dobór algorytmu detekcji obiektów zależy od dostępnych zasobów sprzętowych oraz wymagań konkretnej aplikacji.

# Bibliografia

- [1] Yoshimitsu Aoki, Manabu Nakakita, Masataka Doi i Shuji Hashimoto. „Real-time vision system for autonomous mobile robot”. W: *Proceedings of the 2001 IEEE International Workshop on Robot and Human Interactive Communication (RO-MAN)*. IEEE. 2001, s. 527–532.
- [2] Cytron Technologies Sdn. Bhd. *Hat-MDD10 User’s Manual*. Wer. 1.0. Accessed January 2025. 2017. URL: <https://www.cytron.io/p-hat-mdd10>.
- [3] Andrzej Bielecki. „Matematyczne podstawy sztucznych sieci neuronowych”. W: *Matematyka Stosowana* 4 (2003), s. 25–55.
- [4] Mengxing Huang, Wenjiao Yu i Donghai Zhu. „An improved image segmentation algorithm based on the Otsu method”. W: *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE. 2012, s. 135–139.
- [5] Przemysław Łabecki i Andrzej Kasiński. „Aktywny pokładowy system wizyjny dla robota autonomicznego”. W: *Pomiary Automatyka Kontrola* 55.9 (2009), s. 731–736.
- [6] Larry Matthies, Mark Maimone, Andrew Johnson i in. „Computer Vision on Mars”. W: *International Journal of Computer Vision* 75.1 (2007), s. 67–92.
- [7] Kamil Miękus, Tomasz Marek, Paweł Proszkowiec i Jerzy Zając. „System wizyjny 3D kołowego robota mobilnego”. W: *Logistyka* 6 (2014), s. 7442–7442.
- [8] R Parvadhavardhni, Pankhuri Santoshi i A Mary Posonia. „Blind navigation support system using Raspberry Pi & YOLO”. W: *2023 2nd International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*. IEEE. 2023, s. 1323–1329.
- [9] Chien-Yao Wang, Alexey Bochkovskiy i Hong-Yuan Mark Liao. „YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors”. W: *arXiv preprint arXiv:2207.02696* (2022). Available online: <https://arxiv.org/abs/2207.02696>. URL: <https://arxiv.org/abs/2207.02696>.



## **Dodatki**



# Spis skrótów i symboli

DNA kwas deoksyrybonukleinowy (ang. *deoxyribonucleic acid*)

MVC model – widok – kontroler (ang. *model–view–controller*)

$N$  liczebność zbioru danych

$\mu$  stopnień przyleżności do zbioru

$\mathbb{E}$  zbiór krawędzi grafu

$\mathcal{L}$  transformata Laplace'a



# Źródła

Jeżeli w pracy konieczne jest umieszczenie długich fragmentów kodu źródłowego, należy je przenieść w to miejsce.

---

```
1 if (_nClusters < 1)
2     throw std::string ("unknown number of clusters");
3 if (_nIterations < 1 and _epsilon < 0)
4     throw std::string ("You should set a maximal number of
5         iteration or minimal difference --- epsilon .");
6 if (_nIterations > 0 and _epsilon > 0)
7     throw std::string ("Both number of iterations and minimal
8         epsilon set --- you should set either number of iterations
9         or minimal epsilon .");
```

---



# **Lista dodatkowych plików, uzupełniających tekst pracy**

W systemie do pracy dołączono dodatkowe pliki zawierające:

- źródła programu,
- dane testowe,
- film pokazujący działanie opracowanego oprogramowania lub zaprojektowanego i wykonanego urządzenia,
- itp.



# Spis rysunków

3.1	Widok strony głównej witryny Roboflow. . . . .	19
3.2	Etykietowanie obiektów za pomocą witryny Roboflow. . . . .	19
3.3	Obiekt po zetykietowaniu. . . . .	20
3.4	Etykietowanie w słabym oświetleniu. . . . .	20
3.5	Wykres pierwszych epok treningu. . . . .	21
3.6	Wykres przebiegu treningu z zaistniałym spadkiem mAP. . . . .	22
4.1	Schemat blokowy działania algorytmu pracy robota z systemem wizyjnym .	26
4.2	Schemat elektryczny podłączeń komponentów. . . . .	29
4.3	Projekt konstrukcji robota mobilnego wykonany w programie Autodesk Fusion. . . . .	31
4.4	Podział konstrukcji na piętra. . . . .	31
4.5	Widok rzutu górnego na dolne piętro robota. . . . .	32
4.6	Widok rzutu górnego na górne piętro robota. . . . .	32
4.7	Gotowa konstrukcja robota mobilnego. . . . .	33
5.1	Sytuacja przy dobrym oświetleniu — poprawne wykrycie piłki czerwonej oraz zignorowanie piłki o innych barwach. . . . .	44
5.2	Sytuacja przy złym oświetleniu — poprawne wykrycie piłki czerwonej oraz błędne wskazanie piłki o innych barwach. . . . .	44
5.3	Zwrot informacji w terminalu o wykryciu piłki czerwonej . . . . .	45
5.4	Sytuacja przy dobrym oświetleniu - poprawne wykrycie piłki czerwonej oraz zignorowanie czerwonego obiektu. . . . .	46
5.5	Sytuacja przy złym oświetleniu - poprawne wykrycie piłki czerwonej oraz poprawne zignorowanie czerwonego obiektu. . . . .	46
5.6	Zwrot informacji w terminalu o wykryciu piłki czerwonej i zignorowaniu czerwonego obiektu . . . . .	47
5.7	Sytuacja przy złym oświetleniu - poprawne wykrycie czerwonej piłki w dalekim kadrze . . . . .	50
5.8	Sytuacja przy dobrym oświetleniu - poprawne wykrycie czerwonej piłki w dalekim kadrze . . . . .	50

5.9	Informacja z terminalu RaspberryPi - zgubienie piłki w momencie zgaszenia światła . . . . .	51
5.10	Sytuacja przy złym oświetleniu - poprawne wykrycie czerwonej piłki w bliskim kadrze oraz zignorowanie czerwonego obiektu . . . . .	52
5.11	Sytuacja przy dobrym oświetleniu - poprawne wykrycie czerwonej piłki oraz błędne wskazanie czerwonego obiektu . . . . .	52
5.12	Sytuacja przy złym oświetleniu - poprawne wykrycie czerwonej piłki oraz zignorowanie czerwonego obiektu . . . . .	53
5.13	Sytuacja przy dobrym oświetleniu - błędne wskazanie czerwonego obiektu zasłoniło piłkę . . . . .	53
5.14	Działanie modelu YOLOv7 przy dobrym oświetleniu. . . . .	55
5.15	Działanie modelu YOLOv7 przy złym oświetleniu. . . . .	55
5.16	Działanie modelu YOLOv7 przy dobrym oświetleniu oraz prawidłowe pominięcie obiektu. . . . .	56
5.17	Działanie modelu YOLOv7 przy złym oświetleniu oraz prawidłowe pominięcie obiektu. . . . .	56
5.18	Działanie modelu YOLOv8 przy dobrym oświetleniu. . . . .	57
5.19	Działanie modelu YOLOv8 przy gorszym oświetleniu. . . . .	57
5.20	Działanie modelu YOLOv8 przy dobrym oświetleniu oraz błędne wykrycie obiektu. . . . .	58
5.21	Działanie modelu YOLOv8 przy gorszym oświetleniu oraz prawidłowe pominięcie obiektu. . . . .	58
5.22	Działanie binaryzacji w warunkach dobrego oświetlenia. . . . .	59
5.23	Działanie binaryzacji w warunkach słabego oświetlenia. . . . .	59
5.24	Działanie binaryzacji w warunkach słabego oświetlenia oraz prawidłowe zignorowanie obiektu. . . . .	60
5.25	Działanie binaryzacji w warunkach słabego oświetlenia oraz błędna detekcja obiektu. . . . .	60

# Spis tabel

5.1	Wyniki algorytmów detekcji obiektu bez przeszkody w różnych warunkach oświetleniowych na komputerze osobistym. . . . .	61
5.2	Wyniki algorytmów detekcji obiektu z przeszkodą w różnych warunkach oświetleniowych na komputerze osobistym. . . . .	61
5.3	Wyniki algorytmów detekcji obiektu bez przeszkody w różnych warunkach oświetleniowych na Raspberry Pi. . . . .	62
5.4	Wyniki algorytmów detekcji obiektu z przeszkodą w różnych warunkach oświetleniowych na Raspberry Pi. . . . .	62