

**WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ**

Damian Bielecki

Śledzenie obiektów przez robota mobilnego z wykorzystaniem
głębokiej sieci neuronowej

Projekt inżynierski

Opiekun pracy:
dr.hab.inż Krzysztof Wiktorowicz prof. PRz

Rzeszów, 2023

Spis treści

1. Wprowadzenie	5
2. Głębokie sieci neuronowe	6
2.1. Wprowadzenie do sieci neuronowych	6
2.2. Głębokie sieci neuronowe	7
2.2.1. Konwolucyjne sieci neuronowe	8
2.2.2. Sieci rekurencyjne	8
2.2.3. Uczenie transferowe	8
2.3. Algorytmy detekcji obiektów na zdjęciach	9
2.3.1. Algorytmy R-CNN	9
2.3.2. Szybka detekcja przy pomocy YOLO	11
3. Przegląd literatury i istniejących rozwiązań	13
3.1. Przegląd istniejących rozwiązań	13
3.2. Opis przyjętego rozwiązania	14
4. Budowa robota	16
5. Uczenie sieci neuronowej	21
5.1. Architektura algorytmu wykrywającego obiekty	21
5.2. Zbieranie danych	22
5.2.1. Automatyczne generowanie danych	22
5.2.2. Ręczne oznaczanie danych	25
5.3. Uczenie sieci	27
5.4. Testy	30
6. Połączenie systemów	33
6.1. Wykonana aplikacja kliencka	33
6.2. Generowanie komend i sterowanie robotem	36
7. Przeprowadzone testy	39
7.1. Testowanie trybu normalnego	39
7.2. Testy trybu śledzenia	41
8. Podsumowanie	43
Literatura	45

1. Wprowadzenie

Obserwując rynek możemy zauważyc ciągły wzrost liczby urządzeń będących coraz bardziej inteligentnych i odpornych na ciągle zmieniające się otoczenie.

Celem projektu inżynierskiego jest zbudowanie robota śledzącego wskazany obiekt, który jest wykryty przez zamontowaną na nim kamerę. Robot będzie sterowany przez komputer przetwarzający obraz przy pomocy odpowiednio wytrenowanej głębokiej sieci neuronowej. Zadaniem sieci będzie rozpoznanie i zlokalizowanie wyznaczonego obiektu, a następnie przekazanie jego pozycji algorytmu sterującego. Dalsza część programu wyśle odpowiednio spersonowane komendy do robota, aby ten ustawił karetkę z kamerą na obiektem.

Główym powodem realizacji takiego tematu jest chęć poznania działania i trenowania głębokich sieci neuronowych rozpoznających obiekty na obrazach.

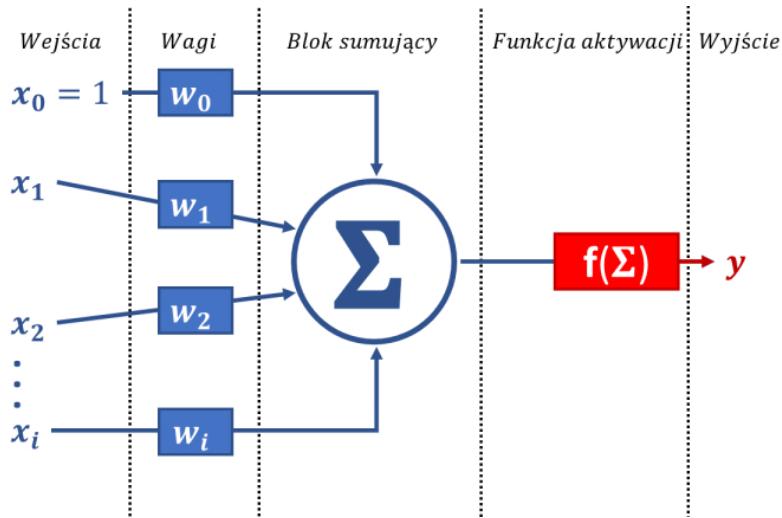
Omówienie rozdziałów

Mając na uwadze zakres pracy i cel projektu, jej treść podzielono na szereg rozdziałów. Rozdział pierwszy ogólnie omawia typy i zasadę działania sieci neuronowych. Rozdział drugi skupia się na przeglądzie literatury, a w szczególności na opisie istniejących rozwiązań wykorzystujących rozpoznanie obiektów przy pomocy głębokiej sieci neuronowej. Rozdział trzeci szczegółowo omawia proces budowy wykorzystywanego dalej robota. Rozdział czwarty opisuje proces pozyskania danych i trenowania sieci rozpoznającej obiekty. Rozdział piąty omawia utworzony program wykorzystujący wytrenowaną sieć neuronową, cały proces sterowania i sposób połączenia wszystkich systemów w działający projekt. Ostatni rozdział przedstawia testy finalnej wersji programu oraz ich krótki opis.

2. Głębokie sieci neuronowe

2.1. Wprowadzenie do sieci neuronowych

Podstawowymi elementami strukturalnymi, z których buduje się sztuczne sieci są neurony. Każdy neuron posiada wejścia, na które podawane są sygnały mnożone przez odpowiednie wagi, sumowane, a następnie po przejściu przez funkcję aktywacji kierowane na wyjście neuronu.



Rysunek 2.1: Schemat pojedynczego neuronu [1]

Wyjście powyższego neuronu możemy opisać wzorem:

$$y = f\left(\sum_{i=0}^N w_i x_i\right) = f(W^T x) \quad (2.1)$$

Gdzie: $x = [1, x_1, x_2, \dots, x_N]$ - to wektor sygnałów wejściowych (1 na początku wektora odpowiada za przesunięcie), $W = [w_0, w_1, w_2, \dots, w_N]$ - jest wektorem wag (w_0 to wartość progowa aktywacji), $f(x)$ to wybrana funkcja aktywacji

Najczęściej stosowane funkcje aktywacji:

$$f(x) = \frac{1}{1 + e^{-x\beta}} \quad (2.2)$$

$$f(x) = \frac{e^x - e^- x}{e^x + e^- x} \quad (2.3)$$

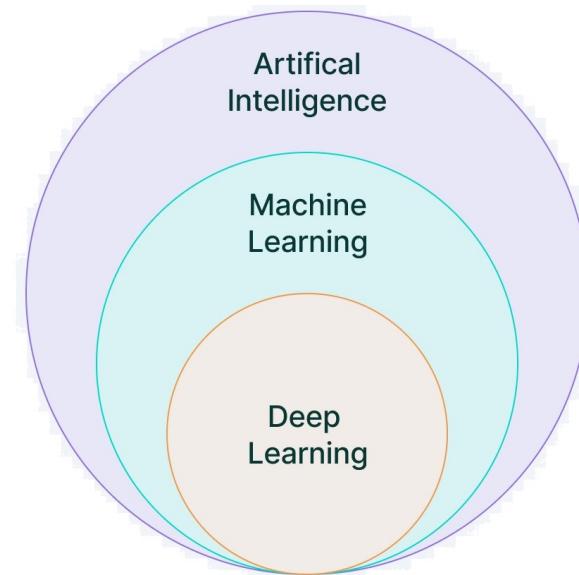
$$f(x) = \begin{cases} 1 & \text{gdy } x > 0 \\ 0 & \text{gdy } x \leq 0 \end{cases} \quad (2.4)$$

Wielowarstwowe sieci neuronowe

W praktycznym zastosowaniu sieć zbudowana z jednej warstwy neuronów nie pozwoli nam na osiągnięcie satysfakcjonujących wyników. Możemy określić dwa typy sieci pod względem liczby warstw:

- uczenie maszynowe - zazwyczaj są zbudowane z trzech warstw (ukryta-ukryta-wyjściowa),
- głębokie uczenie - takie sieci posiadają dużo więcej warstw, a w skomplikowanych sieciach nawet kilkaset.

Relacje pomiędzy nimi przedstawia rysunek 2.2.



Rysunek 2.2: Relacja pomiędzy typami sieci neuronowych [2]

2.2. Głębokie sieci neuronowe

Ze względu na budowę głębokiej sieci neuronowej, ta bardzo dobrze radzi sobie z danymi bez struktury np. obrazy, video, dźwięk czy tekst. W związku z tymi cechami i wraz ze wzrostem wydajności komputerów ta jest wykorzystywana w większej ilości branż. Sieci te stosowane są m.in. w tłumaczeniu i analizie tekstu oraz co w tym przypadku jest ważniejsze klasyfikacji i detekcji obiektów na obrazach.

Jako że uczenie głębokich sieci neuronowych wymaga bardzo dużo czasu i wydajnych macierzy obliczeniowych powstały metody uproszczające cały ten proces.

2.2.1. Konwolucyjne sieci neuronowe

Konwolucja w sieciach neuronowych bazuje na splocie i w większości przypadków oznacza nałożenie filtra na przetwarzany obraz. Operacja splotu polega na całkowaniu sygnału i wygenerowaniu wynikowego obrazu podobnego do oryginalnego. Na podstawie tej operacji działają m.in. filtry takie jak rozmycie obrazu (filtr Gaussa) czy algorytmy wspomagające rozpoznawanie wzorców (falki Gabora). Operacje spłotowe przyśpieszają obliczenia dzięki zmniejszeniu liczby parametrów do wyznaczenia podczas procesu uczenia. Ich dużą zaletą jest:

- uwypuklenie pewnych cech obrazu co ułatwia klasyfikacje,
- łączenie filtrów pozwala na wyróżnieniu wielu cech jednocześnie,
- zmniejsza szum w analizowanych obrazach,
- filtry "poolingowe", które redukują wymaganą ilość połączeń i neuronów co znacząco zwiększa szybkość obliczeń,
- konwolucja pozwala na uniezależnienie się od pozycji wykrywanego obiektu na ekranie.

2.2.2. Sieci rekurencyjne

W typowej sieci neuronowej zakładamy ze wszystkie wyjścia i wejścia są niezależne od siebie co w wielu przypadkach jest złym założeniem (np. analiza szeregu czasowego). Sieci rekurencyjne w porównaniu do zwykłych sieci wyposażone są w pętle sprzężenia zwrotnego dla co najmniej jednej warstwy neuronów. Dzięki tej pętli sieć posiada 'pamięć' i znacznie lepiej przewiduje wyjście dla przebiegów sekwencyjnych. Niestety przez zastosowanie sprzężenia zwrotnego, a więc zwiększenie się liczby występowania różnych przypadków taka sieć musi być znacznie dłużej trenowana.

2.2.3. Uczenie transferowe

Niezależnie od stosowanego algorytmu, uczenie głębokiej sieci neuronowej jest skomplikowane i czasochłonne. W związku z tym została opracowana metoda uczenia, polegająca na użyciu już wytrenowanych sieci i wyuczeniu ich na nowych danych. Zaletą tego rozwiązania jest znaczne przyśpieszenie procesu uczenia, ponieważ sieć

posiada już wstępnie ustalone wartości wag. Dodatkową zaletą przetrenowanych wag jest zmniejszenie ilości wymaganych danych do osiągnięcia zadowalających efektów. Zazwyczaj wytrenowane sieci są bardzo duże i posiadają miliony parametrów, nawet kilkaset kategorii i kilkadziesiąt ukrytych warstw.

2.3. Algorytmy detekcji obiektów na zdjęciach

W praktyce nauczenie sieci neuronowej kategoryzującej obrazy i znajdujące się na nich obiekty nie jest najtrudniejszym zadaniem stawianym przed takimi sieciami. Zdecydowanie większym wyzwaniem jest wskazanie dokładnej pozycji obiektu znajdującego się we wskazanym obrazie. Szukając materiały na temat istniejących rozwiązań można natknąć się na dwa określenia: segmentacja i detekcja. Segmentacja polega na przydzieleniu każdego piksela obrazu do klas wyuczonych w sieci. Detekcja obrazu polega na znalezieniu wszystkich obiektów znanych trenowanej sieci, określaniu ich kategorii oraz wyznaczeniu ich pozycji.

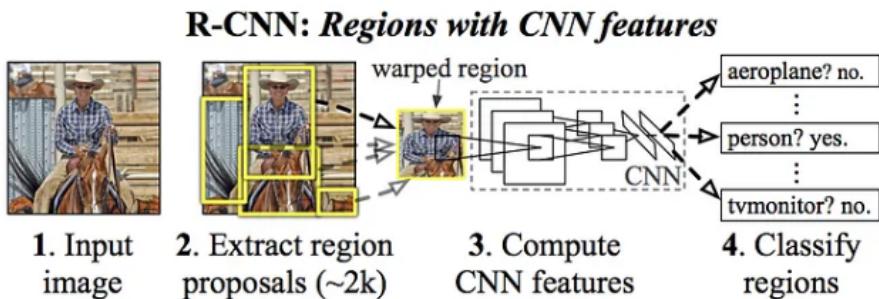


Rysunek 2.3: Pierwsze zdjęcie obrazuje detekcję obiektów, drugie to segmentacja obrazu[3]

2.3.1. Algorytmy R-CNN

Istnieje cała rodzina algorytmów R-CNN (Regions with Convolutional Neural Network), bazujących jak sama nazwa wskazuję na splotowych sieciach neuronowych. Rodzina ta jest grupą pierwszych algorytmów z wysokim wskaźnikiem skuteczności.

W celu rozpoznania obiektów, algorytm nakłada na oryginalny obraz 2000 regionów służących do dalszej klasyfikacji.



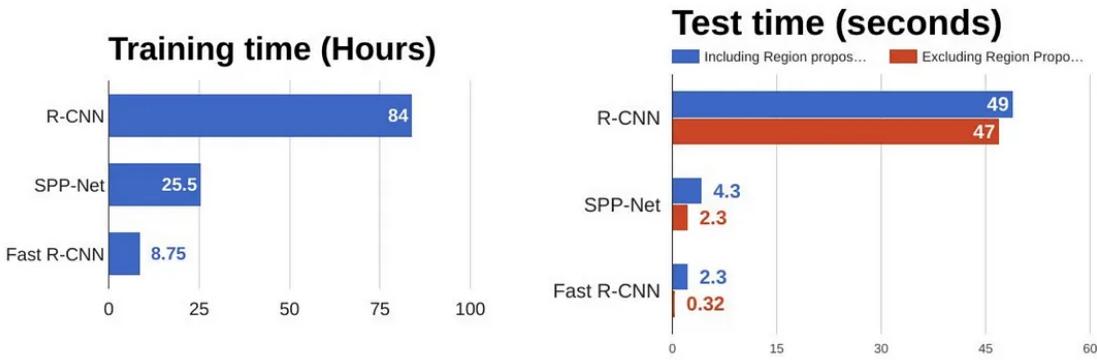
Rysunek 2.4: Kroki działania modelu RCNN [4]

Każdy taki region trafia do sieci neuronowej zajmującej się wyodrębnieniem cech szczególnych dla danej klasy. Następnie wszystkie te dane trafiają do SVM (Support Vector Machine) w celu sklasyfikowania obecności obiektu w danym regionie. Poza klasą, sieć ta przewiduje również cztery przesunięcia, które pomagają w zwiększeniu precyzji obwiedni, szczególnie gdy obiekt znajduje się w kilku regionach.

Taka sieć ma sporo wad takich jak:

- ze względu na klasyfikacje 2000 regionów na zdjęcie, uczenie trwa bardzo długo i nie może być wykonywane w czasie rzeczywistym,
- bardzo długi czas przetwarzania, nawet 47s na zdjęcie,
- algorytm wyszukiwania regionów jest stały i nie uczy się wraz z pozostałym algorytmem co oznacza że może też proponować złe regiony do klasyfikacji.

Kolejna wersja tego algorytmu nazwana Fast R-CNN, pozbywa się generowania za każdym razem 2000 regionów, które potem były przetwarzane. W tej wersji wykorzystano kolejną sieć neuronową do generowania splotowej mapy obiektów. Dalej na podstawie tej mapy identyfikowane są proponowane regiony, które są skalowane do jednego, stałego rozmiaru i sklasyfikowane przez kolejną sieć neuronową. Dzięki temu algorytm nie przetwarza za każdym razem 2000 regionów a operacja splotu wykonywana jest tylko raz.



Rysunek 2.5: Porównanie wydajności rodziny algorytmów R-CNN [4]

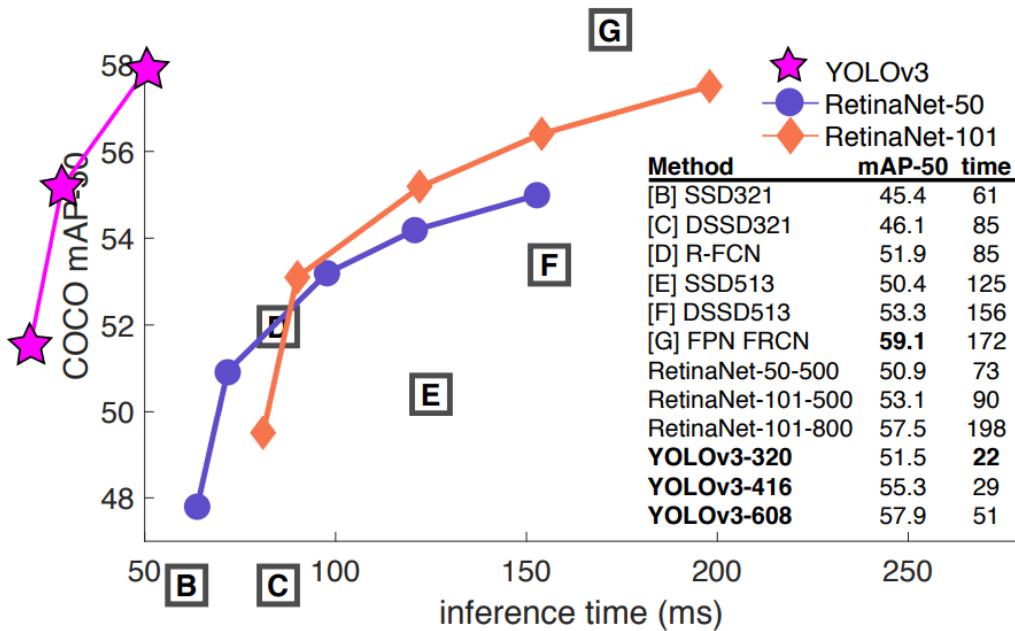
Rysunek 2.5 przedstawia porównanie wydajności w kolejnych wersjach modeli do detekcji obiektów. Widoczna jest duża różnica w czasie przetwarzania obrazu pomiędzy pierwszą i kolejnymi dwoma wersjami algorytmów.

2.3.2. Szybka detekcja przy pomocy YOLO

Algorytm YOLO (You only look once) w przeciwieństwie do pozostałych algorytmów może działać w czasie rzeczywistym. Wszystkie tradycyjne metody detekcji obiektów cały proces dzielą na kilka odrębnych faz, a YOLO wykonuje analizę i predykcje obiektów jednocześnie. Działanie algorytmu można rozpisać w kilku krokach:

- wejściowy obraz dzielony jest na siatkę komórek o rozmiarze $S \times S$, każda taka komórka odpowiedzialna jest za przewidywanie obiektów znajdujących się w jej obszarze,
- dalej dla każdej wygenerowanej ramki wykonywana jest predykcja, poprzez wyznaczenie B obiektów i współrzędnych prostokąta otaczającego obiekt. Wektor wyjściowy ma rozmiar $B \times 5$ wartości, ponieważ cztery pola zarezerwowane są dla rozmiaru i pozycji prostokąta, a piąte, logiczne oznacza czy w danej komórce znajduje się obiekt,
- po przewidzeniu ramek dla wszystkich komórek, przy pomocy techniki NMS (Non-maximum suppression) lokalizowane i usuwane są duplikaty tych samych obiektów,
- ostatecznie na podstawie poprzednich kroków generowane są wyniki w postaci otaczających dany obiekt ramek, wykrytych etykiet i ich prawdopodobieństwa.

Dzięki takiemu podejściu wykrywanie obiektów przy pomocy YOLO jest dużo szybsze, a równocześnie równie skuteczne w porównaniu do tradycyjnych metod.



Rysunek 2.6: Zdjęcie porównujące szybkość i skuteczność różnych algorytmów detekcji obiektów. [5]

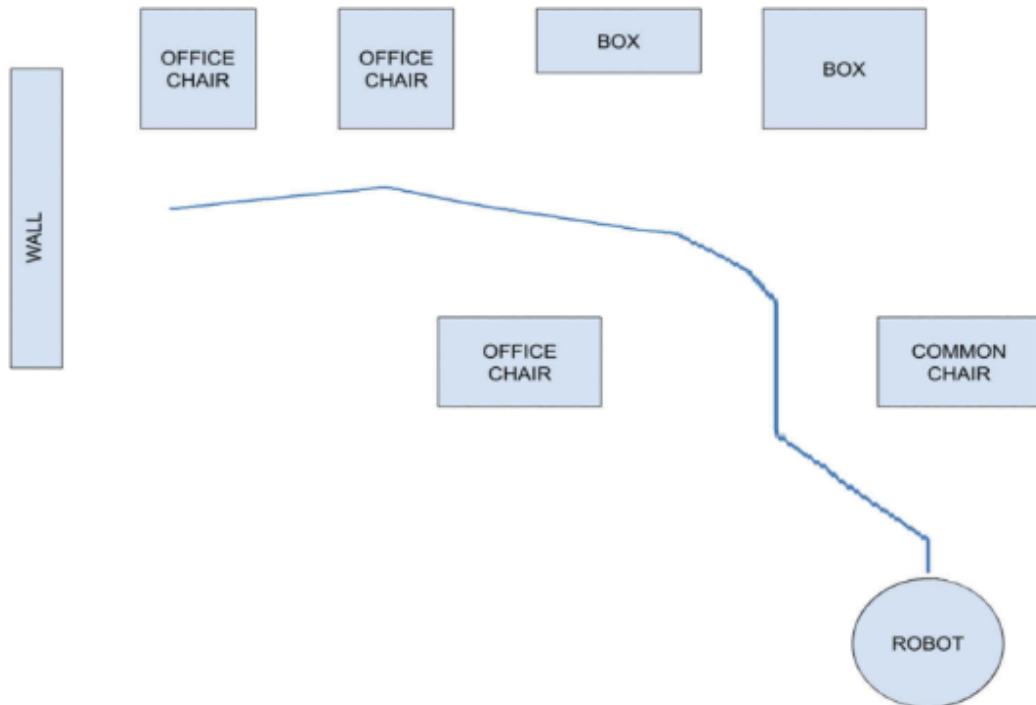
Na rysunku 2.6 widać, że algorytm YOLO w wersji trzeciej jest zdecydowanie szybszy od pozostałych prezentowanych przykładów równocześnie zachowując podobną skuteczność wykrywania. Dla przykładu algorytm R-FCN przetwarza jeden obraz w czasie około 85ms ze skutecznością 51,9mAP, a najgorszy przypadek dla algorytmu YOLO potrzebuje już tylko 55ms i osiąga skuteczność 57,9mAP. Na podstawie przedstawionego przykładu możemy zauważyć, że omawiany algorytm jest zdecydowanie lepszy w rozwiązaniach mających pracować w czasie rzeczywistym.

Parametr mAP (mean Average Precision) jest miarą używaną do porównywania różnych modeli detekcji obiektów na obrazie. Wskaźnik ten wyznaczony jest dla wszystkich klas danych obiektów co oznacza, że uwzględnia ogólną wydajność modelu, a nie skuteczność tylko jednej kategorii. Im wyższa wartość tym lepsza wydajność detekcji.

3. Przegląd literatury i istniejących rozwiązań

3.1. Przegląd istniejących rozwiązań

Przeglądając dostępną literaturę łączącą rozpoznawanie obiektów i roboty, można zauważyc, że takie sieci neuronowe, w dużym stopniu wspomagają analityczne algorytmy sterujące robotami. Większość artykułów opisuje mobilne roboty, których zadaniem jest poruszanie się w nieznanym środowisku. Pierwszy artykuł pt. "Mobile Robot Navigation Using an Object Recognition Software with RGBD Images and the YOLO Algorithm"[\[6\]](#), opisuje wykorzystanie algorytmu YOLO do określenia przeszkód, w nieznanym otoczeniu. Sieć neuronowa została wytrenowana na podstawie około 1100 zdjęć, zawierających: dwa typy krzeseł biurowych, stół i pudełko. Cały proces uczenia, został uruchomiony w chmurze Google i trwał około trzy dni. Aby polepszyć proces sterowania i planowania ścieżki, autorzy wykorzystali sensor Microsoft Kinect, dzięki któremu można określić rzeczywistą odległość obiektów od kamery. Sam algorytm został uruchomiony na komputerze jedno-płytkowym Nvidia Jetson TX2 GPU, który w najgorszych warunkach przetwarzał obraz z prędkością 3,76 klatek na sekundę.

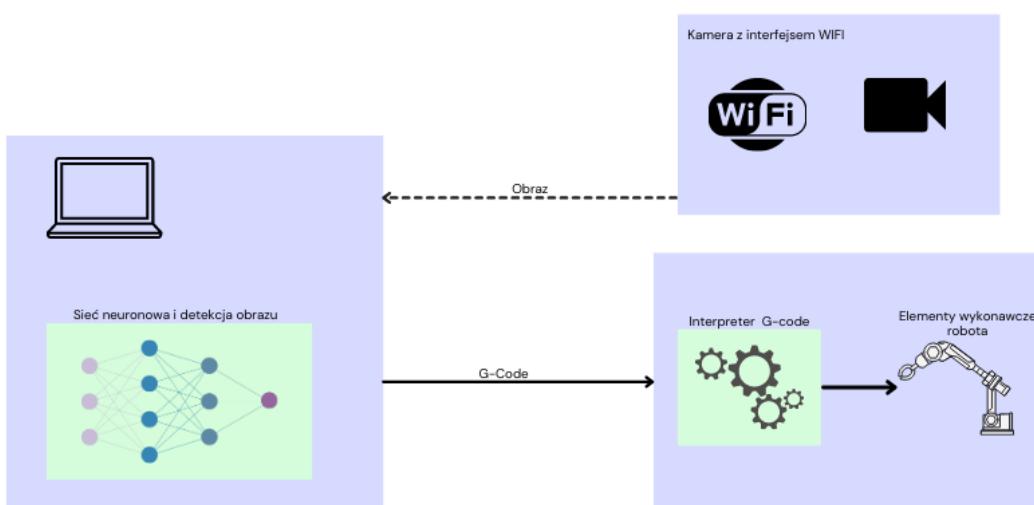


Rysunek 3.7: Ścieżka i otoczenie, w którym poruszał się robot[\[6\]](#)

Na rysunku 3.7 widoczne są oznaczone przeszkody, robot oraz przebyta ścieżka. Jak widać w tym i kolejnych prezentowanych przez autorów testach, robot osiągnął docelową pozycję, mimo że nie wiedział o otaczających go przeszkodach. W zaprezentowanym rozwiąaniu, sieć neuronowa musi rozpoznać każdy otaczający robota obiekt, ponieważ w przeciwnym przypadku, ten nie uzna go za przeszkoda i go nie ominie. W produkcyjnym rozwiążaniu przewidzenie każdego otaczającego robota obiektu jest bardzo trudne.

3.2. Opis przyjętego rozwiązania

Jako iż proces samego uczenia sieci neuronowej jak i jej wykorzystanie jest zbyt wymagające dla dowolnego mikrokontrolera przetwarzającego dane to algorytm wykrywający obiekty, zostanie uruchomiony na laptopie wyposażonym w dedykowaną kartę graficzną RTX3050Ti oraz 6 rdzeniowy procesor AMD Ryzen 5 5600H.



Rysunek 3.8: Schemat ogólny przyjętego rozwiązania

Jak widać na schemacie z rysunku 3.8, możemy wydzielić trzy niezależne współpracujące ze sobą elementy.

Sterownik

Zadaniem sterownika jest pobranie, odpowiednie przygotowanie obrazu (np. przeskakowanie go) i przekazanie go do algorytmu wykorzystującego głęboką sieć neuronową. Algorytm zwraca pozycję i etykiety wszystkich rozpoznanych na obrazie obiektów, a pozostała część programu wybiera śledzoną etykietę, wyznacza jego pozycję w prze-

strzeni roboczej robota i na końcu wysyła odpowiednią komendę.

Robot

Robot zbudowany został w oparciu o laser CNC małej mocy. Maszyna posiada dwie ruchome osie XY z karetką i przyczepionym modułem lasera. Laser i silniki krokowe sterowane są przy pomocy sterownika opartego na płytce Arduino i odbierającym dane przy pomocy portu USB. Zamiast lasera przy pomocy wydrukowanego na drukarce 3D uchwytu została zamocowana kamera nagrywająca przestrzeń roboczą, na której stoi urządzenie. Budowa robota została szerzej opisana w kolejnym rozdziale.

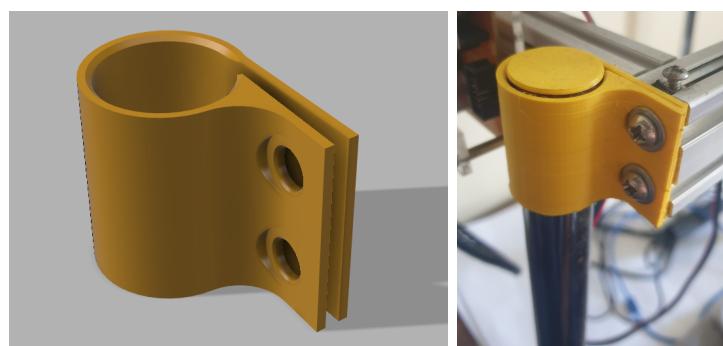
Kamera

Jako kamera została wykorzystany telefon komórkowy z systemem Android nagrywający video w rozdzielczości 1920x1080 i 30 klatek na sekundę. W systemie została zainstalowana aplikacja pozwalająca na strumieniowanie video przy pomocy sieci wifi co ze względu na ciągły ruch kamery znacznie uprościło jej połączenie. W systemie Windows, na którym uruchomiona jest sieć neuronowa, telefon widoczny jest jako zwykła kamera systemowa.

4. Budowa robota

Główna konstrukcja

Robot został zbudowany w oparciu o konstrukcję lasera CNC małej mocy. Główna rama, po której porusza się mocowanie z kamerą, zbudowana jest z profili aluminiowych skręconych śrubami. Urządzenie posiada dwie sterowalne osie napędzane przy pomocy pasków maszynowych i silników krokowych. Konstrukcja stoi na aluminiowych nogach przymocowanych do ramy poprzez wydrukowane na drukarce 3D tuleje zaciskowe. Rozwiążanie to pozwala na regulację każdej nogi osobno i ustawienie poziomu.



Rysunek 4.9: Mocowanie nóg robota, model i zdjęcie z robota

Wszystkie opisywane dalej modele zostały wydrukowane z materiału 'PLA' w temperaturze ok. 210°C i wysokością warstwy 0,15mm.

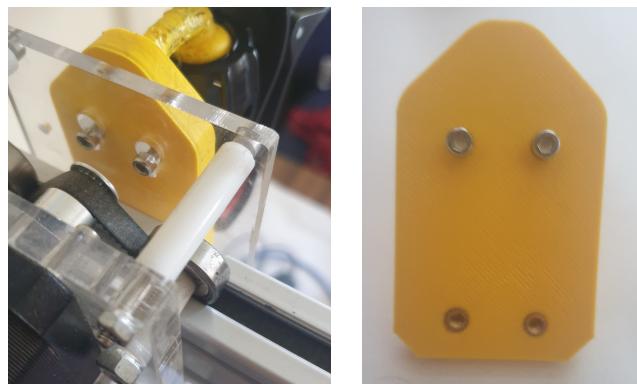
Mocowanie kamery

Moduł z laserem został zdemontowany, a do oryginalnego uchwytu zaprojektowane zostało mocowanie telefonu, będącego w tym przypadku kamerą.



Rysunek 4.10: Mocowanie kamery do karetki

Do zamocowania telefonu wykorzystany został zacisk uchwytu samochodowego. Uchwyt mocowany jest poprzez nakrętkę zaciskającą się na wydrukowanej na wysięgniku kuli. Takie połączenie pozwala na ustawienie kamery pod dowolnym kątem względem filmowanej powierzchni zachowując wysoką sztywność.



Rysunek 4.11: Połączone mocowanie i karetka robota

Do przygotowanych otworów zostały na gorąco wciśnięte mosiężne, gwintowane tulejki, które pozwalają na wkręcenie śrub z gwintem M3. Dzięki nim uchwyt może być sztywnie zamocowany. Finalnie okazało się, że do sztywnego połączenia tych dwóch powierzchni wystarczą tylko dwie śruby.

Napędy

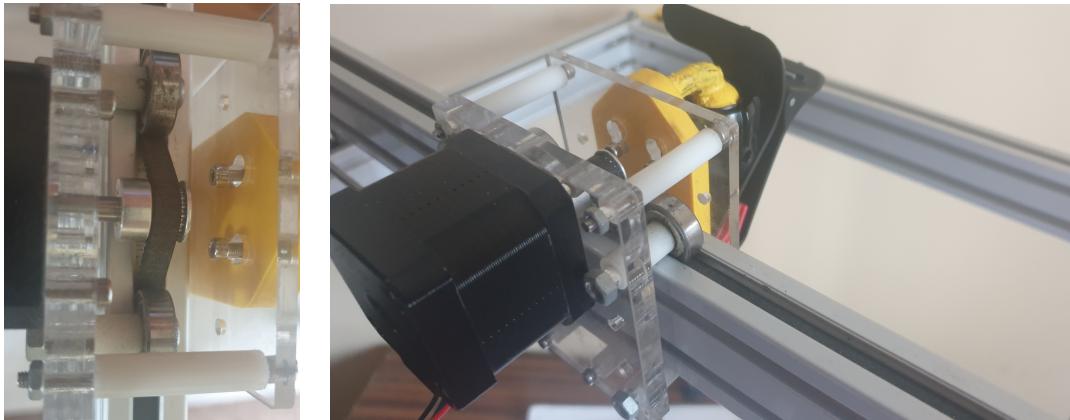
Do napędzania obu osi robota wykorzystane zostały dwa silniki krokowe 4240-15A, o maksymalnym prądzie 1A, kroku osi 1.8° i maksymalnym momencie 53Ncm. Silniki przykręcane są do grubej płyty typu 'plexa'.



Rysunek 4.12: Przekazanie napędu dla pierwszej osi

Pierwsze zdjęcie z lewej pokazuje napęd osi, po której porusza się karetka z

zamocowanym kolejnym napędem. Widać że na osi silnika zamocowane jest sprzęgło podatne (kompensujące drgania), które połączone jest z wałkiem napędowym. Na obu końcach wałka zamocowane są koła zębate napędzające pasek GT-2. Pasek przyczepowany jest na sztywno do ramy a naciąg kontrolowany jest przez dwa dociskające go łożyska. Przeniesienie napędu na drugą stronę, konieczne jest ze względu na dosyć dużą ramę i możliwe skrzywianie się podczas ruchu.

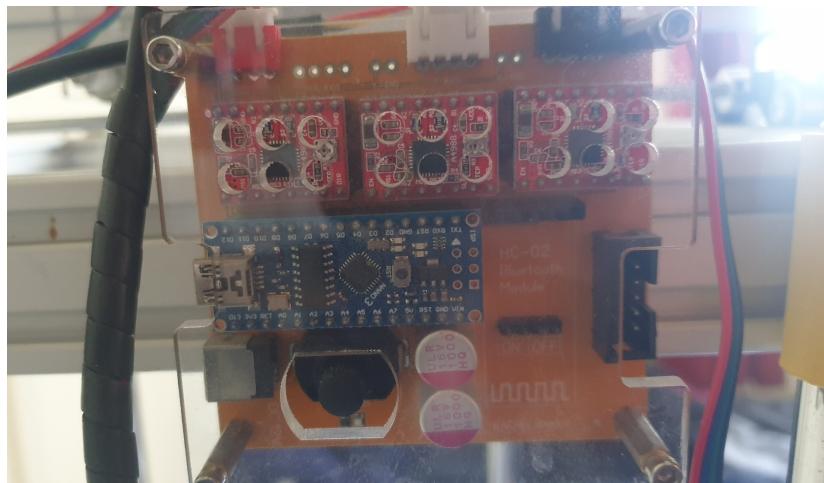


Rysunek 4.13: Napęd osi przesuwającej karetką

Napęd drugiej osi został zrealizowany identycznie, poza tym że w tym przypadku koło zębate jest zamocowane bezpośrednio na osi silnika.

Sterowanie i komunikacja

Sterowanie zrealizowane jest przy pomocy mikrokontrolera Arduino Nano i płytki z odpowiednimi sterownikami silników krokowych. Sterownik ma wgrany program pozwalający, na komunikacje z komputerem przy pomocy portu szeregowego i analizę wysyłanych komend G-Code. Komendy pozwalają na poruszanie robotem z określoną prędkością lub modyfikacje parametrów pracy np. przejście z współrzędnych absolutnych na przyrostowe.



Rysunek 4.14: Zdjęcie sterownika

Wykorzystano sterowniki silników krokowych opartych na układzie A4988. Modułowe sterowniki, pozwalają na szybką wymianę w przypadku awarii.

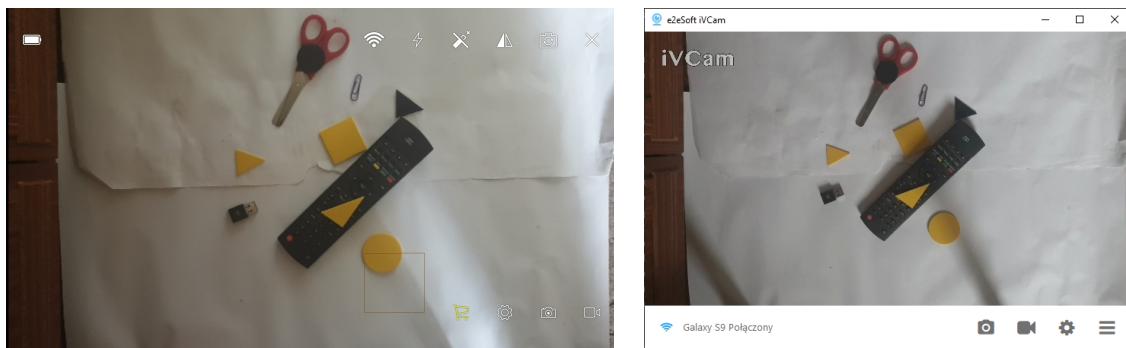
Oprogramowanie sterownika

Do sterowania silnikami i interpretacji wysyłanych poleceń w formacie G-Code użyto otwartego projektu grbl^[7] w wersji 1.1f. Repozytorium z całym projektem można pobrać ze strony github. Dzięki zastosowaniu otwartego projektu, sterownik od razu zna wszystkie potrzebne komendy takie jak G0 (szybki ruch liniowy), G1 (ruch liniowy z narzędziem) czy inne jak G53 (zmiana systemu współrzędnych na absolutne).

Do uruchomienia projektu, potrzebujemy program Arduino IDE, który zawiera w sobie kompilator i potrzebny program programujący mikrokontroler sterownika. Przed komplikacją należy wskazać konkretne piny odpowiadające za sterowanie silnikami. Po wgraniu programu należy odpowiednio skonfigurować podstawowe parametry np. ilość kroków silnika potrzebną do przesunięcia się o 1mm. Ustawienia i zapisania parametrów dokonuje się poprzez wysłanie odpowiedniej kombinacji poleceń przy pomocy dowolnego terminala łączącego się poprzez port szeregowy np. PuTTy.

Kamera

W tym przypadku jako kamerę wykorzystano telefon z zainstalowaną aplikacją iVCam. Aby połączyć się z komputera należy zainstalować program kliencki iVCam, który dodaje wirtualną kamerę do systemu.



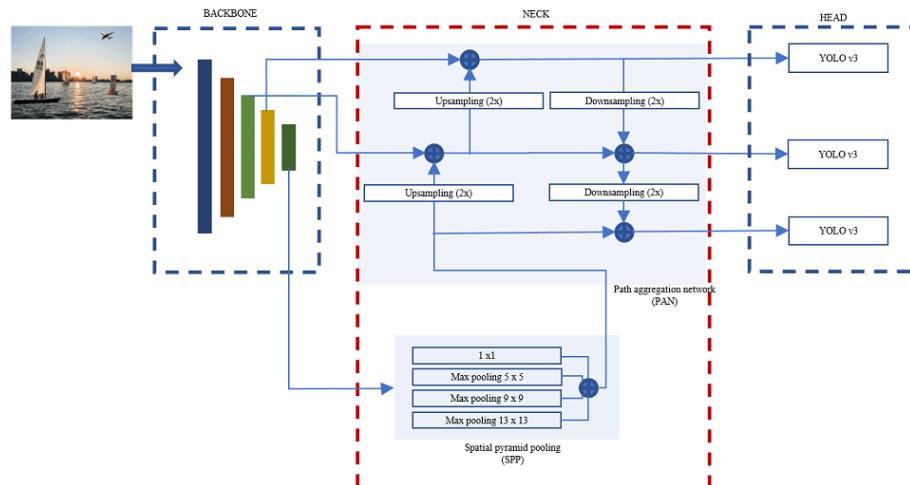
Rysunek 4.15: Połączenie telefonu i komputera przy pomocy aplikacji iVCam

Jak widać na powyższym zdjęciu, telefon połączył się z aplikacją zainstalowaną na komputerze. Aplikacja imituje zwykłą kamerę widoczną przez system operacyjny, dzięki czemu nie była potrzebna żadna dodatkowa biblioteka do obsługi połączenia bezprzewodowego. W porównaniu do wcześniej testowanym protokołem RTSP, to rozwiązanie cechuje się bardzo niskim opóźnieniem i dobrą jakością przesyłanego obrazu. Wadą aplikacji jest znak wodny z logiem, jednak nie przeszkadzał on w pracy sieci neuronowej.

5. Uczenie sieci neuronowej

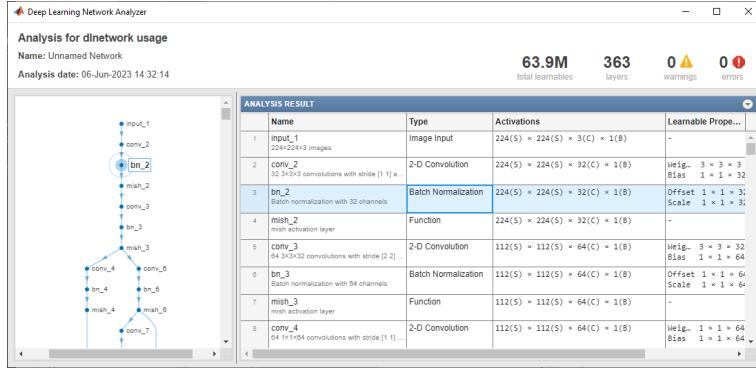
5.1. Architektura algorytmu wykrywającego obiekty

Do zbudowania modelu wykrywającego wskazane klasy na zdjęciach użyto oprogramowania Matlab, dodatku Deep Learning Toolbox oraz innych dodatków pozwalających na przetwarzanie obrazu. Wykorzystany model bazuje na sieci neuronowej nazwanej CSPDarkNet53 i wstępnie wytrenowanym na zbiorze [COCO](#) zawierającym ponad 200 tysięcy oznaczonych zdjęć i 80 różnych kategorii.



Rysunek 5.16: Sposób implementacji modelu YOLOv4 w Matlab’ie [8]

Implementacja algorytmu YOLO w Matlabie została podzielona na trzy różne sekcje. Pierwsza, oznaczona na rysunku 5.16 jako 'backbone', jest szkieletem sieci odpowiadającym za obliczenie mapy cech z obrazów wejściowych. Druga warstwa, łączy mapy cech z danymi z warstw sieci szkieletowej oraz wysyła je do kolejnego modułu. Ostatni segment odpowiada za przetworzenie wcześniej wyodrębnionych cech, przewidzenie obwiedni i klasy obiektów.



Rysunek 5.17: Parametry użytej sieci uzyskane funkcją analyzeNetwork

Użyta sieć składa się z 53 warstw konwolucyjnych, wejścia 224x224x3 oraz trzech wyjść określających prawdopodobieństwo występowania danej klasy. Cała sieć zbudowana jest z 363 warstw a do wyuczenia jest ponad 63mln parametrów.

Istnieje mniejsza sieć nazwana 'Tiny-YOLOv4', która posiada zaledwie około 30 warstw konwolucyjnych, dzięki czemu została znacznie zwięksona szybkość działania, kosztem niewielkiego pomniejszenia skuteczności. Według wielu różnych źródeł wersja ta jest zdecydowanie lepsza do przetwarzania szybko zmieniającego się obrazu.

5.2. Zbieranie danych

5.2.1. Automatyczne generowanie danych

W pierwszej wersji dane miały być generowane przy pomocy automatycznego skryptu. Utworzony program w Matlabie, otwierał wcześniej nagrany film z samym tłem, a następnie na pobranych klatkach obrazu wstawiał szablony z przygotowanymi obrazami docelowych obiektów. Dzięki dynamicznemu generowaniu, można było zachować zróżnicowanie danych pomimo ich dużej ilości.



Rysunek 5.18: Lewe zdjęcie przedstawia maskę z usuniętym tłem a prawe klatkę nagranego filmu z naniesionym obiektem

Jak widać na rysunku 5.18, generowane dane wyglądają dosyć sztucznie, głównie ze względu na tło oraz niekonsekwentne oświetlenie obiektu i tła. Ostatecznie wynikiem działania algorytmu był utworzony zbiór zdjęć oraz jeden plik typu gTruth z zapisanymi obrazami, lokalizacją i klasą obiektów. Poniżej dołączono skróconą wersję programu generującego zdjęcia.

```

1 % automatyczne generowanie danych do uczenia sieci
2 clc; clear; close all;
3
4 videoFiles = ["data/vid/w1.3gp", "data/vid/w2.3gp"];
5 pathToSaveImages = "data/test2";
6 outputImageSize = [224, 224];
7 filterToOutputImage = fspecial("motion", 2, 2);
8
9 dataKolo = {}; dataKwadrat = {}; dataTrojkat = {};
10 dataFileNames = "";
11 for vidName = videoFiles
12     vid = VideoReader(vidName); %wczytanie filmu
13     rrStep = int8(vid.NumFrames);
14     for fIndX = 1 : 5 : vid.NumFrames % iteracja po klatkach
15         frame = read(vid, fIndX);
16
17         % dostosowanie klatki do rozmiaru sieci
18         frame = imresize(frame, outputImageSize);
19
20         % losowanie odpowiedniej maski/obiektu
21         rr = randi([1, 6]);
22         if(rr == 1)
23             [x, map, alpha] = imread("data\mask\kw_mask.png");
24         % wczytywanie kolejnych masek
25         end
26         x = imresize(x, 0.5);
27         alpha = imresize(alpha, 0.5);
28         maskSize = size(x);
29         %wyznaczenie pozycji
30         posX = randi([0, outputImageSize(1) - maskSize(1)]);
31         posY = randi([0, outputImageSize(2) - maskSize(2)]);
32
33         if(rr == 1 || rr == 2) % kwadrat
34             dataKolo(end + 1) = {[[]]};
35             dataKwadrat(end + 1) = {[posX, posY, maskSize(1), maskSize(2)]};
36             dataTrojkat(end + 1) = {[[]]};
37         elseif (rr == 5 || rr == 6 || rr > 5) % kolo
38             dataKolo(end + 1) = {[posX, posY, maskSize(1), maskSize(2)]};
39             dataKwadrat(end + 1) = {[[]]};
40             dataTrojkat(end + 1) = {[[]]};
41         elseif (rr == 3 || rr == 4) % tr
42             dataKolo(end + 1) = {[[]]};
43             dataKwadrat(end + 1) = {[[]]};
44             dataTrojkat(end + 1) = {[posX, posY, maskSize(1), maskSize(2)]};
45         end

```

```

46
47     if (fIndX > rr * rrStep)
48         rr = rr + 1;
49    end
50
51    % wzstawienie szablonu do obrazu
52    frame = insertImageInPos(frame, x, alpha, posX, posY);
53    % dodanie opcjonalnego filtru do zdjecia
54    if exist("filterToOutputImage" , "var") == true
55        frame = imfilter(frame, filterToOutputImage);
56    end
57
58    path = sprintf("%s/%i-%i.jpg", pathToSaveImages, randi(300)
59 , fIndX);
60    dataFileNames = dataFileNames + path + ";";
61
62    %zapisanie obrazu
63    imwrite(frame, path)
64 end
65
66 % zapisywanie danych jako obiektu datastore
67 labels = labelDefinitionCreator();
68 addLabel(labels, "kolo", "Rectangle");
69 addLabel(labels, "kwadrat", "Rectangle");
70 addLabel(labels, "trojkat", "Rectangle");
71 labelData = table(dataKolo', dataKwadrat', dataTrojkat', '
72 VariableNames',{ 'kolo' , 'kwadrat' , 'trojkat' });
73 t = split(dataFileNames, ";"); t(end) = [];
74 f = matlab.io.datastore.FileSet(t);
75 imds = imageDatastore(f);
76 labelDataStore = boxLabelDatastore(labelData);
77 ds = combine(imds, labelDataStore);
78 save("ds", "ds");

```

Listing 1: Generowanie danych

Skrypt otwierał wskazany film i pobierał co 5 klatkę obrazu. Dalej generowana była liczba określająca klasę wstawionego obiektu. Szablon był wczytywany do pliku, a typ i losowo wygenerowana pozycja została zapisana w bazie danych. Na końcu mógł zostać dodany filtr, mający poprawić efekt przejścia pomiędzy połączonymi obrazami. Tak przygotowane zdjęcie zostało zapisane na dysku. Po przetworzeniu filmu dane o obiektach, ich pozycjach oraz odpowiednich zdjęciach były zapisywane do odpowiedniego pliku.

Epoch	Iteration	TimeElapsed	LearnRate	TrainingLoss
1	30	00:01:26	1e-06	742.46
1	60	00:02:16	1e-06	358.22

```
Error using reshape
Number of elements must not change. Use [] as one of the size
inputs to automatically calculate the appropriate size for that
dimension.

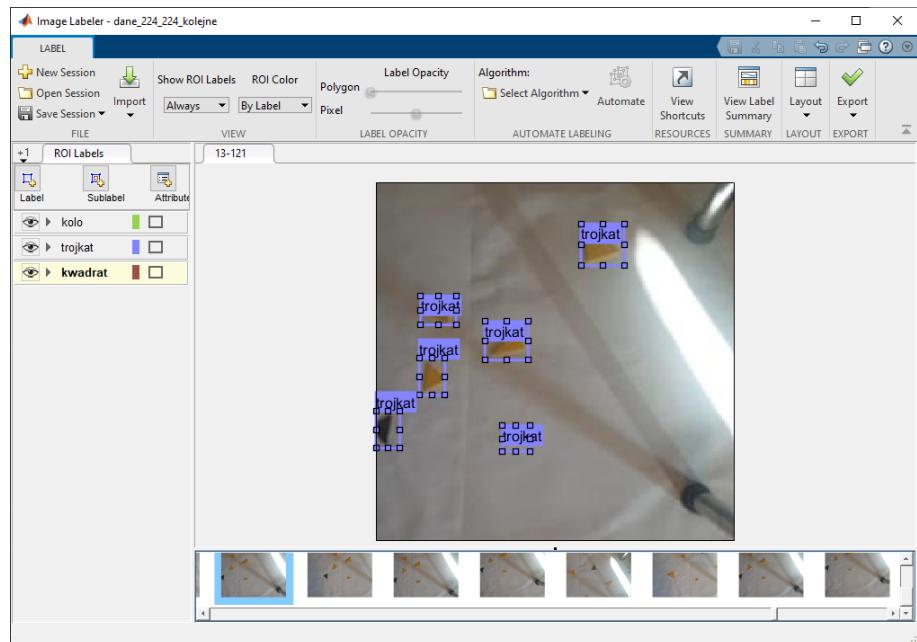
Error in trainYOLOv4ObjectDetector>iGetMaxIOUPredictedWithGroundTruth (line 430)
iou(:,:,batchSize) = reshape(maxOverlap,h,w,c);
```

Rysunek 5.19: Błąd podczas trenowania obrazu

Ostatecznie największym problemem okazało się wytrenowanie sieci na tak wygenerowanych danych. Prawdopodobnie algorytm źle zapisywał dane, ponieważ w trakcie trenowania pierwszej epoki i losowej iteracji trenowanie przerywało się, wyświetlając komunikat z błędem funkcji 'reshape' wywoływanej wewnątrz funkcji trenującej model.

5.2.2. Ręczne oznaczanie danych

Docelowa i działająca sieć została wytrenowana na ręcznie oznaczonych zdjęciach. Dane zostały zebrane na wykonanym już robocie i nagrane poprzez symulowanie ruchów robota i przesuwanie docelowych obiektów. Nagrane filmy zostały przetworzone poprzez skrypt, który pobrał z filmu co którąś klatkę a następnie przeskalała ją do docelowego rozmiaru.



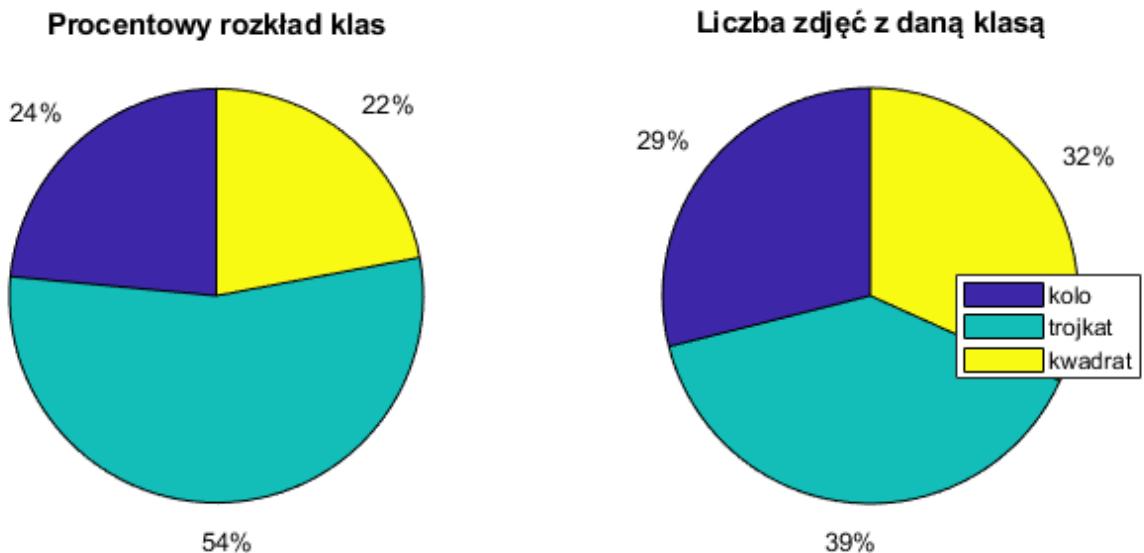
Rysunek 5.20: Program imageLabeler, pozwalający na etykietowanie zdjęć

Rysunek 5.20 przedstawia zrzut ekranu programu, w którym zdefiniowano odpowiednie klasy, wczytano przygotowane dane oraz oznaczono odpowiednie obiekty ich klasami. Baza zawiera ponad 1200 zdjęć z obiektami (zmienione orientacje, pozycje, kształt) i różnym tłem (zmiana oświetlenia, dodatkowe cienie).



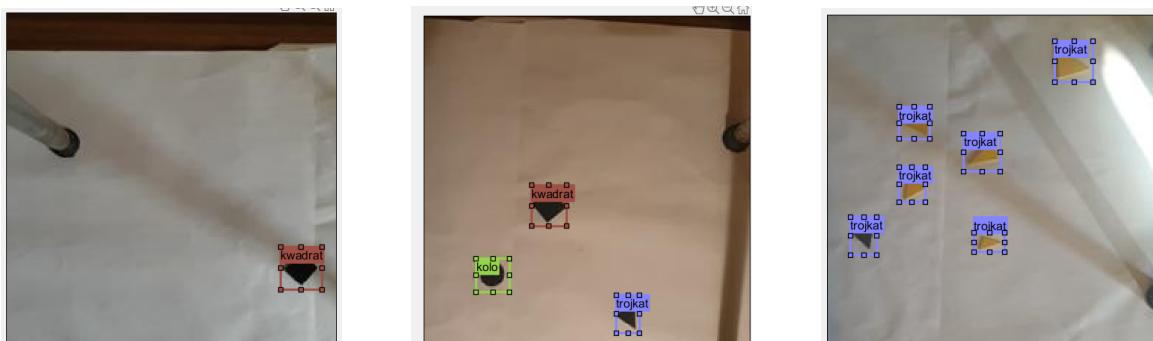
Rysunek 5.21: Rozkład klas w poszczególnych zdjęciach

Baza zdjęć była systematycznie zwiększana wraz z analizą skuteczności działania wytrenowanej sieci. Jak widać najwięcej jest zdjęć zawierających kwadraty, a najmniej, ze względu na uniwersalny kształt, kół. Ostatecznie rozkład klas prezentuje się następująco:



Rysunek 5.22: Ogólny rozkład klas

Jak widać zgodnie z oczekiwaniemi w bazie ze względu na największą ilość konfiguracji jest najwięcej zdjęć trójkątów. Liczba kół i kwadratów jest bardzo podobna. Poniżej zostało zaprezentowane przykładowe zdjęcia.



Rysunek 5.23: Przykładowe otagowane zdjęcia

Pierwsze fotografie zawierają pojedyncze obiekty, jednak wraz z rozwojem, na zdjęciach pojawiało się coraz więcej różnych obiektów.

5.3. Uczenie sieci

Sieć neuronowa była trenowana przy pomocy toolbox'a Deep Learning Toolbox w Matlabie, przy pomocy poniższego skryptu.

```

1 clc; clear;
2 deepLearnignNetworkName = "csp-darknet53-coco";
3
4 classes = ["kol", "kwadrat", "trojkat"];
5 inputSize = [224 224 3];
6 % model do wczytania i douszczania
7 modelToLoad = "detector.mat";
8 %wczytanie obrazow do trenowania
9 imds = load("export0105.mat");
10 imds = imds.gTruth;
11
12 imdsTrain = imageDatastore(imds.DataSource.Source);
13 bldsTrain = boxLabelDatastore(imds.LabelData);
14 ds = combine(imdsTrain, bldsTrain);
15
16 % wyznaczenie funkcji do przetwarzania obrazow
17 trainingDataForEstimation = transform(ds,@(data) preprocessData(
    data, inputSize));
18 numAnchors = 90;
19 [anchors, meanIoU] = estimateAnchorBoxes(
    trainingDataForEstimation, numAnchors);
20 area = anchors(:,1).*anchors(:,2);
21 [~, idx] = sort(area, "descend");
22 anchors = anchors(idx, :);
23 anchorBoxes = {anchors(1:3, :); anchors(4:6, :); anchors(7:9, :)};
24
25 disp("Wczytano dane...");
26 % wczytanie istniejacego detektora lub stworzenie nowego
27 if modelToLoad == ""
28     detector = yolov4ObjectDetector(deepLearnignNetworkName,
29         imds.LabelDefinitions.Name, anchorBoxes, InputSize=inputSize);
30     %analyzeNetwork(detector.Network) % analiza sieci, widok
31     warstw
32 else
33     m = load(modelToLoad);
34     detector = m.detector;
35 end
36
37 disp("Wczytano siec...");
38
39 options = trainingOptions("sgdm", ...
40     InitialLearnRate=0.001, ...
41     MiniBatchSize=16, ...
42     MaxEpochs=20, ...
43     BatchNormalizationStatistics="moving", ...
44     ResetInputNormalization=false, ...
45     ExecutionEnvironment="cpu", ...
46     VerboseFrequency=30, ...
47     Plots="training-progress");
48
49 %uczenie sieci
50 disp("Uczenie");
51 [detector, info] = trainYOLov4ObjectDetector(ds, detector, options);

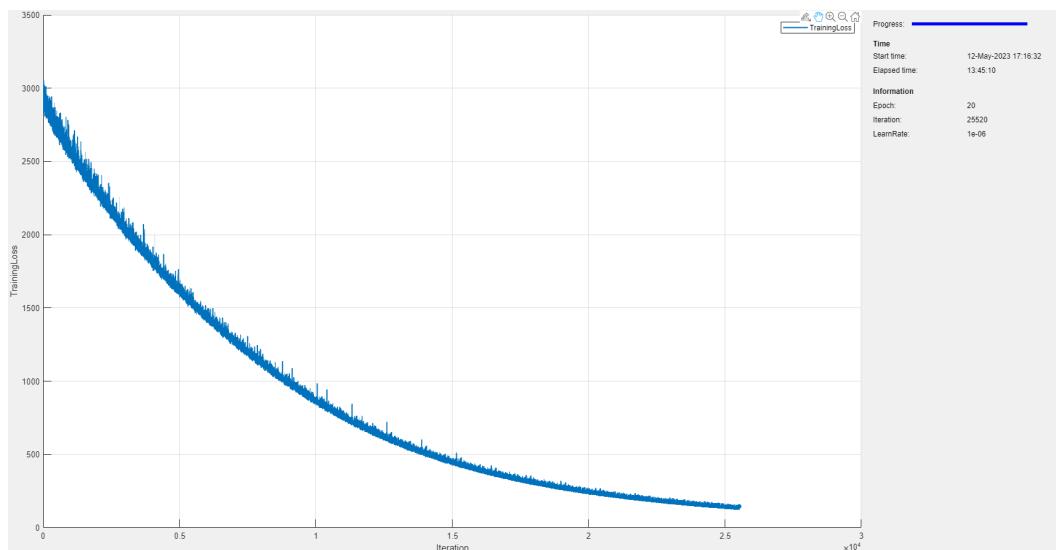
```

Listing 2: Uczenie sieci

Ze względu na małą ilość pamięci graficznej, uczenie odbywa się na procesorze. Podczas jednej iteracji, algorytm przetwarza jednocześnie 16 zdjęć (parametr Mini-BatchSize). Przy takiej konfiguracji, uczenie na GPU nie było możliwe, a jednocześnie po zmniejszeniu tego parametru do 1 uczenie nie było tak skuteczne jak w przypadku jednoczesnego przetwarzania większej ilości obrazów. Parametry uczenia sieci:

- metoda uczenia: sgdm (Stochastic Gradient Descent with momentum),
- współczynnik uczenia: 0.001,
- liczba epok: 20.

Ostatecznie finalny model był douczany przy pomocy powyższego skryptu trzy razy co oznacza, że przeszedł przez 60 epok. Zbyt duża ilość epok uczących sieć może spowodować negatywny wynik z powodu przetrenowania sieci i ścisłego dopasowania się do danych treningowych.



Rysunek 5.24: Przebieg błędu w zależności od liczby epok

Wykres z rysunku 5.24, przedstawia przebieg trenowania i spadku błędu. Widac, że na początku procesu błąd bardzo szybko spada, jednak od połowy uczenia tendencja spada. Zdjęcie pochodzi z pierwszego etapu trenowania, gdzie trenowanie było efektywniejsze. Kolejne dwa etapy trenowania cechowały się spadkiem z około 50 do 0,2 wartości błędu.

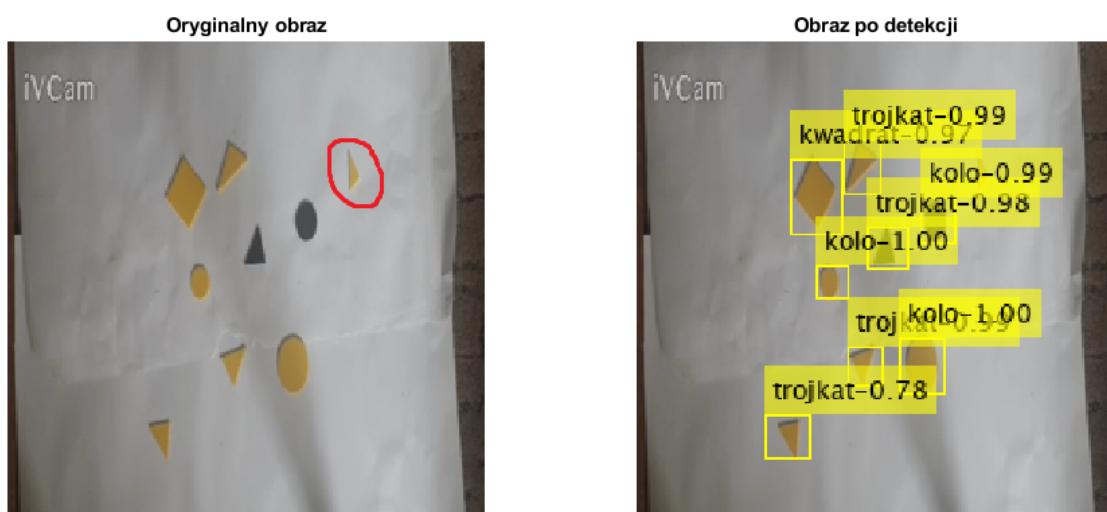
5.4. Testy

Aby sprawdzić skuteczność sieci, przeprowadzono testy polegające na podaniu wcześniej niewidzianego obrazu z wieloma obiektami, ręczne przeanalizowanie wyników sieci i porównanie ich z rzeczywistą liczbą obiektów.



Rysunek 5.25: Pierwszy test z dobrym oświetleniem

Na podstawie wyniku pokazanemu na rysunku 5.25 widać, że sieć w sprzyjających warunkach radzi sobie z rozpoznawaniem różnych obiektów, leżących w różnych orientacjach i pozycjach. Równocześnie należy zwrócić uwagę na bardzo duże wskaźniki pewności sieci wykrytych obiektów.



Rysunek 5.26: Test z dużą ilością różnych obiektów

Analizując przypadek przy dobrym oświetleniu z rysunku 5.26, widać, że sieć ogólnie dobrze poradziła sobie z tak dużą liczbą różnych obiektów. Analizując dokładniej obraz, można zauważyć, że nie wykryty został jeden mniejszy trójkąt (zaznaczony na czerwono).



Rysunek 5.27: Test przy słabym oświetleniu

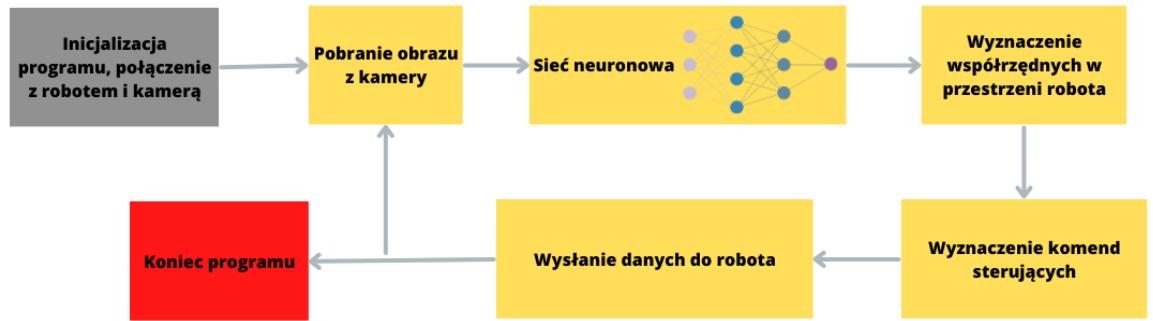
Po ograniczeniu światła sieć rozpoznała tylko cztery z sześciu obiektów. Równocześnie w porównaniu do poprzednich eksperymentów spadły wskaźniki pewności. Należy zauważyć, że pominięte obiekty są bardzo słabo widoczne i w pierwszej chwili nawet ludzkie oko może mieć problem z szybką lokalizacją.



Rysunek 5.28: Testy z nieznanymi obiektami

Ostatni test został przeprowadzony po dodaniu nieznanych sieci obiektów. Wiadać, że sieć nie rozpoznała leżącego na nożyczkach koła i źle sklasyfikowała dwa obiekty (oznaczone pomarańczową otoczką). Żółty kwadrat i leżący niżej moduł do myszki bezprzewodowej oznaczyła jako trójkąt. Podobnie jak w poprzednim przypadku, przy tak niskiej rozdzielczości trudno jest ludzkiem okiem rozpoznać, że nie jest to żaden z wytrenowanych obiektów, choć ten bardziej przypomina kwadrat, a nie trójkąt.

6. Połączenie systemów

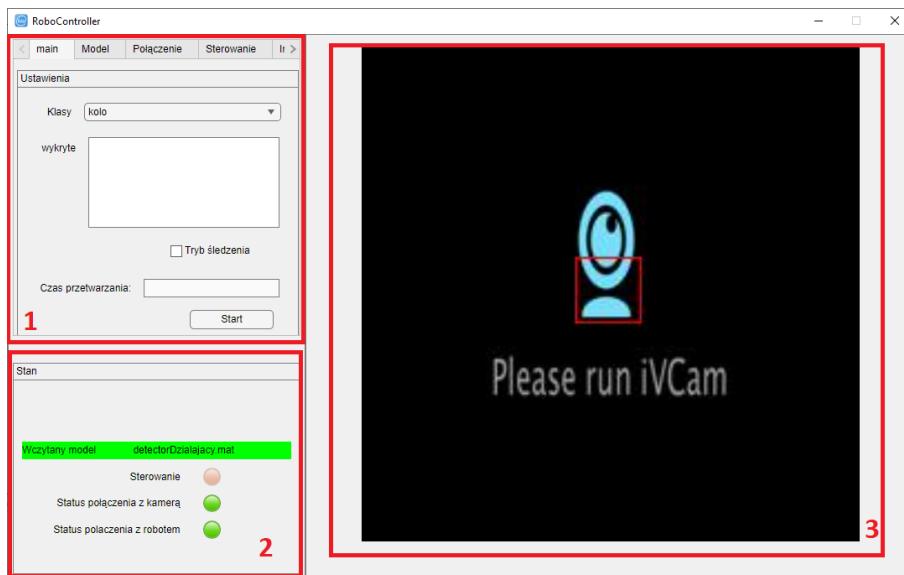


Rysunek 6.29: Ogólny schemat działania algorytmu

Jak widać na algorytmie sterowania z rysunku 6.29, program w pierwszej kolejności inicjalizuje potrzebne komponenty takie jak: kamera, połączenie z robotem oraz wczytanie wybranego modelu sieci neuronowej. Po uruchomieniu głównej pętli algorytmu, pobierany jest obraz z kamery, który przekazywany jest do sieci neuronowej realizującej detekcję obiektów. Po wykryciu obiektów, wybierany jest docelowy, a następnie na podstawie jego pozycji wyznaczana jest komenda odpowiadająca za przesunięcie karetki robota. Cały proces jest zapętlony aż do momentu zatrzymania przez użytkownika.

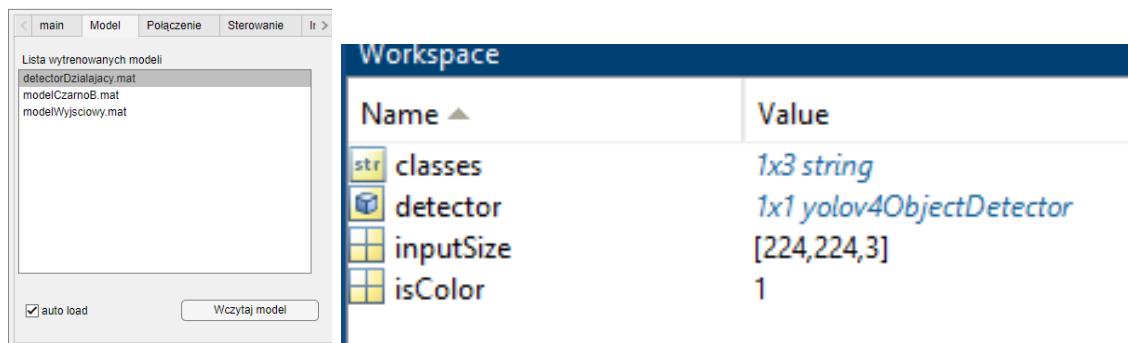
6.1. Wykonana aplikacja kliencka

Program obsługujący komunikacje z robotem i kamerą oraz uruchomienie sieci neuronowej wykonano w Matlab'ie i dodatku AppDesigner, który pozwala na zbudowanie graficznego interfejsu użytkownika i uruchamianie wytrenowanych modułów z siecią neuronową.



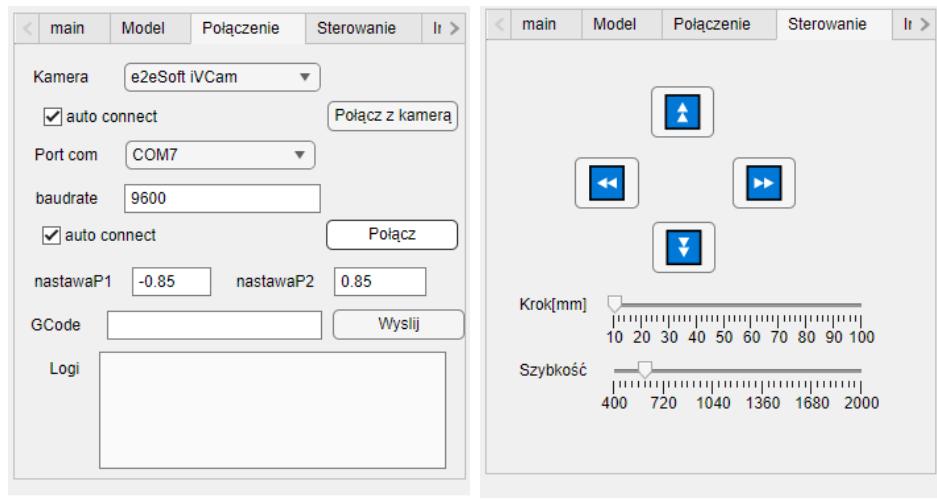
Rysunek 6.30: Widok wykonanego programu

Na rys. 6.30 widoczne jest główne okno aplikacji. Możemy wyróżnić trzy oddzielne sekcje, odpowiadające kolejno za ustawienia sterowania, podgląd stanu poszczególnych modułów oraz podgląd obrazu i nałożonych na niego wyników sieci.



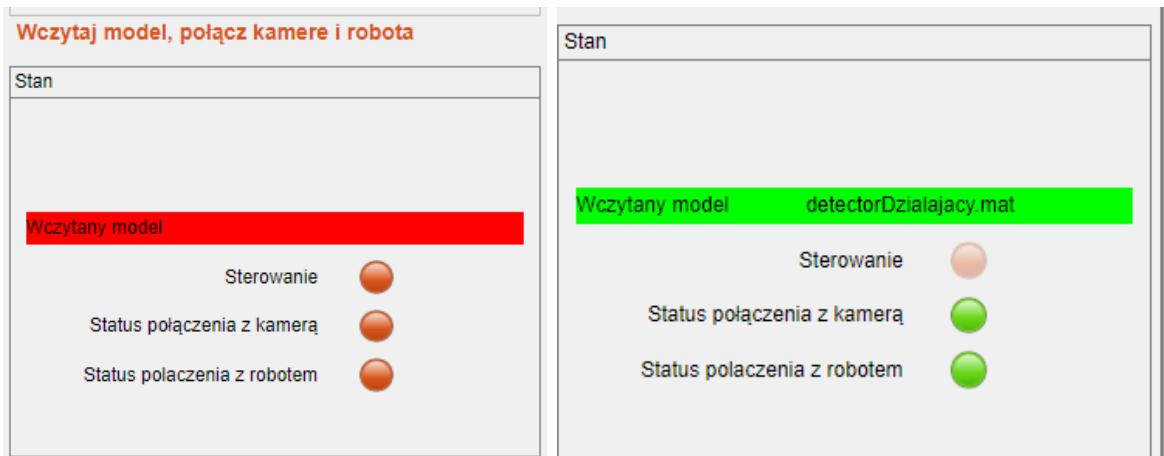
Rysunek 6.31: Wczytanie danych modelu i jego struktura

Program pozwala na wczytanie wybranego pliku z modelem i jego ustawieniami. Opcja 'auto load' oznacza, że model zostanie wczytany automatycznie zaraz po uruchomieniu aplikacji co przyśpieszyło proces testowania programu. Plik przechowuje wytrenowaną sieć neuronową i waży około 233Mb co oznacza, że ładowanie powoduje znaczne spowolnienie programu przy starcie.



Rysunek 6.32: Ustawienia połączeń oraz ręczne sterowanie robotem

Lewy zrzut ekranu przedstawia zakładkę pozwalającą na ustawienia połączenia pomiędzy aplikacją, a robotem i kamerą. Po połączeniu użytkownik ma możliwość wysłania własnej komendy G-Code wprowadzonej w odpowiednim polu. Dane odebrane po wysłaniu polecenia wyświetlane są w polu logów. Ekran z prawej strony po wybraniu prędkości i przesunięcia pozwala na ręczne sterowanie karetką robota.



Rysunek 6.33: Stan aplikacji

Stan aplikacji jest cały czas prezentowany przy pomocy prostych kontrolek zmieniających swój kolor. Wczytanie modelu z siecią sygnalizowane jest poprzez zielony kolor i nazwę wczytanego pliku. Podobnie poprzez kolor sygnalizowany jest stan połączenia z robotem i zamocowaną na nim kamerą. Kontrolka 'Sterowanie' informuje

o tym, czy uruchomiony jest algorytm śledzący wybrany obiekt. Ostatecznie aplikacja została skompilowana do wykonywalnego pliku exe, który można uruchomić na dowolnym komputerze.

6.2. Generowanie komend i sterowanie robotem

Program posiada dwa różne tryby sterowania. Pierwszy tryb, po uruchomieniu algorytmu szuka wybranego obiektu, a po jego znalezieniu wyznacza różnicę pomiędzy środkiem obrazu, a zaznaczoną ramką i jednorazowo wysyła komendę G-Code. Tryb "śledzenia", w przeciwieństwie do poprzedniego cały czas uruchamia sieć neuronową rozpoznającą obiekty i wyznacza komendy odpowiadające za niewielkie przesunięcia. Tryb ten nazwany jest śledzącym, ponieważ dynamicznie reaguje na zmianę pozycji danego obiektu i stale za nim podąża.

```

1 function mainLoop(timer, ev, app)
2     imgSize = app.data.inputSize(1:2); timeStart = tic;
3     frame = (snapshot(app.cam)); frame = imresize(frame, imgSize
4 );
5     if(app.data.isColor == 0)
6         frame = rgb2gray(frame);
7     end
8     if app.TrybledzeniaCheckBox.Value == 1
9         frame = sledzenie(timer, ev, app, frame);
10    else
11        frame = proste(timer, ev, app, frame);
12    end
13
14    frame = insertShape(frame, "rectangle", [imgSize(1)/2 - 15,
15 imgSize(2)/2 - 15, 30, 30], "Color","red");
16    if(app.data.isColor == 0)
17        frame = cat(3, frame, frame, frame);
18    end
19    app.Image.ImageSource = frame;
20
21 function [app, frame, bboxes, scores, labels] = getBoxPos(app,
22 frame)
23     timeStart = tic; app.wykryteTextArea.Value = ["Brak"];
24     [bboxes, scores, labels] = detect(app.data.detector, frame);
25     % detekcja
26     if(app.data.isColor == 0)
27         frame = cat(3, frame, frame, frame);
28     end
29     if isempty(labels) == 0 % wykryto jakieś obiekty
30         headers = [];
31         for ii=1:length(labels)
32             headers = [headers sprintf("%s: %0.2f", labels(ii),
33 scores(ii))];
34         end
35         frame = insertObjectAnnotation(frame,"rectangle",bboxes,
36 headers, 'FontSize', 8);

```

```

31         app.wykryteTextArea.Value = string(labels);
32     end
33     timeEcl = toc(timeStart);
34     str = sprintf("%0.5fs", timeEcl);
35     app.CzasprzetwarzaniaEditField.Value = str;
36 end
37 function frame = sledzenie(timer, ev, app, frame) %algorytm
38     sledzacy
39     [app, frame, bboxes, scores, labels] = getBoxPos(app, frame)
40 ;
41     imgSize = app.data.inputSize(1:2);
42     if(~isempty(bboxes))
43         ind = find(strcmp(string(labels), app.KlasyDropDown.
Value));
44         if app.robot.isConnected && ~isempty(ind)
45             % x, y, width, height
46             box = bboxes(ind, :);
47             score = scores(ind);
48             centerBox = [box(1) + 0.5*box(3), box(2) + 0.5*box
(4)];
49             error = imgSize./2 - centerBox;
50             p = error .* [-0.4, 0.4];
51             for i=1:length(p)
52                 if(p(i) > 0)
53                     p(i) = min([6, round(p(i))]);
54                 else
55                     p(i) = max([-6, round(p(i))]);
56                 end
57             end
58             app.robot.sendGCodeToRobot("G80");
59             cmdToSend = sprintf("G1 X%2.2f Y%2.2f F%i", p(1), p
(2), 2000);
60             res = app.robot.sendGCodeToRobot(cmdToSend);
61         end
62     end
63 function frame = proste(timer, ev, app, frame)
64     imgSize = app.data.inputSize(1:2);
65     if isempty(app.selectedPosFromYOLO)
66         [app, frame, bboxes, scores, labels] = getBoxPos(app,
frame);
67         if isempty(labels) == 0
68             ind = find(strcmp(string(labels), app.KlasyDropDown.
Value));
69             % wyznaczenie komendy wyslanej do robota
70             if ~isempty(ind)
71                 app.selectedPosFromYOLO = ind;
72                 if app.robot.isConnected
73                     % x, y, width, height
74                     box = bboxes(ind, :);
75                     score = scores(ind);
76                     centerBox = [box(1) + 0.5*box(3), box(2) +
0.5*box(4)];
77                     error = imgSize./2 - centerBox;
78                     pp = [str2double(app.nastawaP1EditField.

```

```

79     Value), str2double(app.nastawaP2EditField.Value)];
80             p = error .* pp;
81             cmdToSend = sprintf("G1 X%2.2f Y%2.2f F%i",
82             p(1), p(2), 1500);
83             res = app.robot.sendGCodeToRobot(cmdToSend);
84         end
85     end
86 end
87 end

```

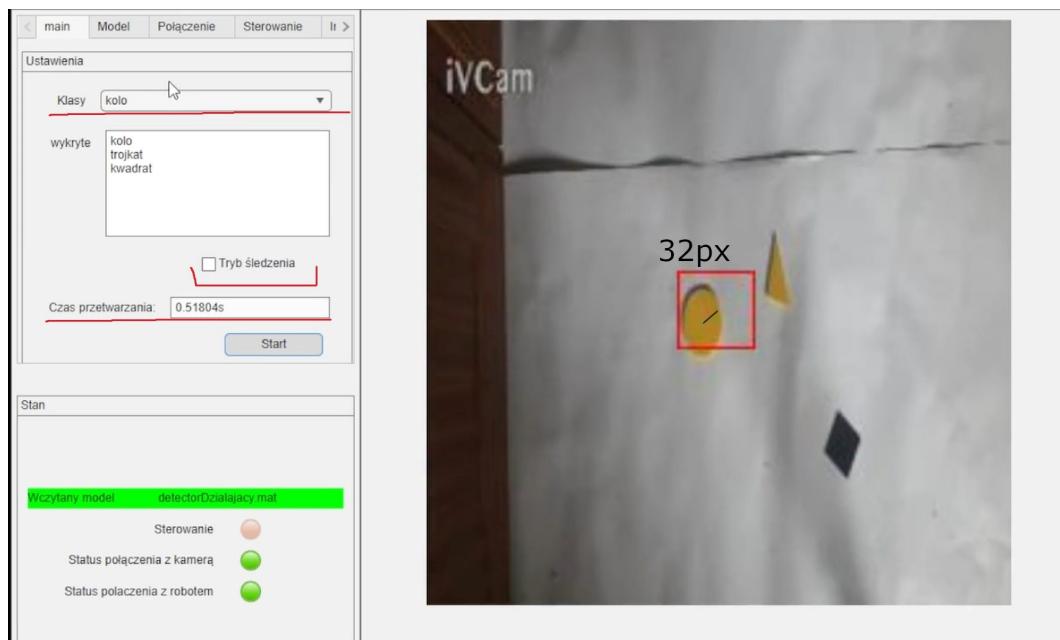
Listing 3: Uczenie sieci

Program z listingu 3, odpowiada za algorytm śledzący obiekty znalezione przez sieć neuronową. Niezależnie od trybu śledzenia wywoływana jest funkcja mainLoop, która odpowiada za pobranie obrazu z kamery, uruchomienie odpowiedniej wersji algorytmu oraz dodanie do finalnego obrazu 'celownika', dodającego punkt odniesienia względem środka. W przypadku prostszej wersji, uruchamiana jest detekcja obiektów przy pomocy sieci neuronowej aż do momentu wykrycia pożąданej klasy. Po wykryciu obiektu algorytm wyznacza potrzebne do osiągnięcia celu przesunięcie, a następnie odpowiednio przygotowane polecenie wysyła do robota. Przesunięcie wyznaczane jest na podstawie różnicy środka obrazu i wykrytego obiektu, ten błąd dalej wzmacniany jest poprzez odpowiednie nastawy P1 i P2. Jak widać jest to regulator typu P, a współczynniki wzmacniania proporcjonalnych odzwierciedlają skalę pomiędzy obrazem, a rzeczywistą odległością. Tryb śledzenia różni się tym, że sieć neuronowa uruchamiana jest ciągle i dzięki temu robot może nieustannie korygować błąd nadążania. Poza tym wyznaczone maksymalne przesunięcie ograniczone jest do stałej wartości tak, aby robot nie wykonywał zbyt dużych niepotrzebnych ruchów.

7. Przeprowadzone testy

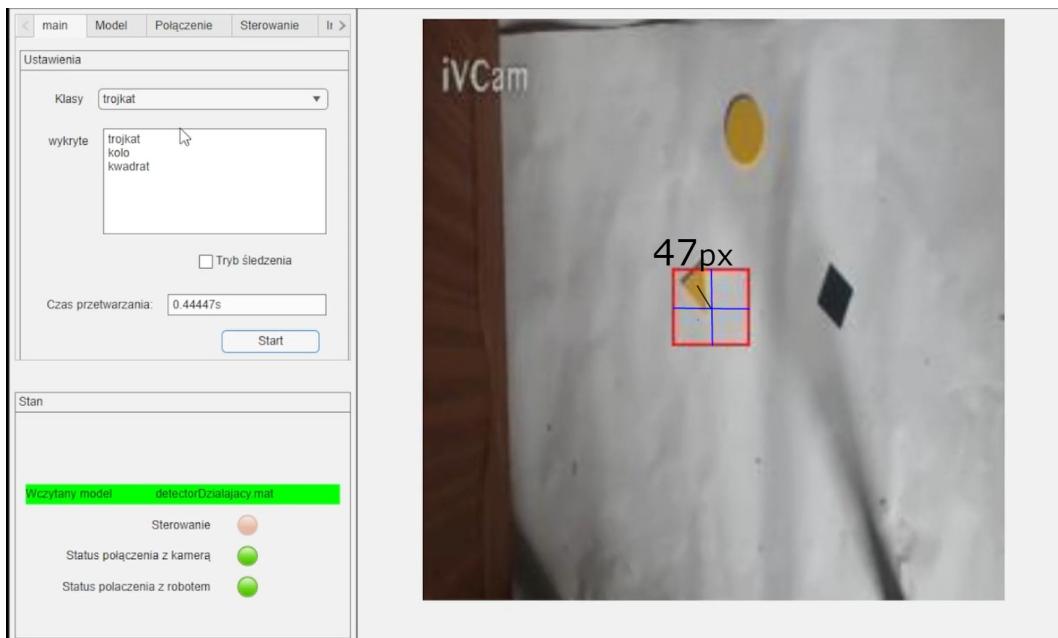
7.1. Testowanie trybu normalnego

Jako pierwsze wykonano testy trybu jednorazowo ustawiającego się nad wybranym obiektem. Aby uruchomić ten tryb, należy wpierw połączyć się z robotem i kamerą, wczytać wybrany model, a następnie w głównej zakładce wybrać docelowy obiekt. Po tym trzeba nacisnąć przycisk start i algorytm będzie zapętlony do momentu wykrycia pierwszego obiektu i wygenerowania komendy przesuwającej uchwyt z kamerą.



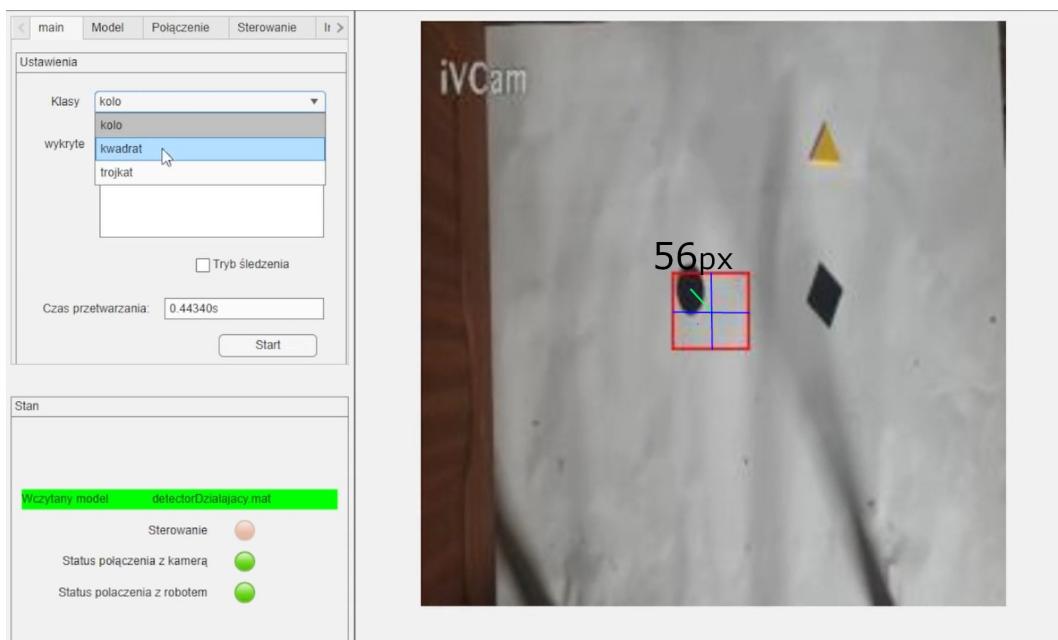
Rysunek 7.34: Test trybu normalnego - wyśledzenie koła

Na rysunku 7.34 widać, że sieć wykryła koło a następnie wygenerowała komendę, która spowodowała przesunięcie. Oznaczony został element interfejsu użytkownika, pozwalający na wybranie docelowego obiektu oraz wyświetlający wszystkie znalezione na zdjęciu klasy. W celu określenia błędu pomiędzy środkiem obrazu a osiągniętą pozycją, dodana została czarna linia. W powyższym przykładzie widać, że linia ta ma 32 piksele, a koło znajduje się w 'celowniku', jednak ich środki nie pokrywają się.



Rysunek 7.35: Test trybu normalnego - znalezienie trójkąta

Przed uruchomieniem kolejnego testu, zmieniono pozycje obiektów i wybraną w programie kategorie. Sieć neuronowa znalazła wszystkie obiekty na obrazie, ale również w tym przypadku ostateczna pozycja cechowała się dużym błędem względem środków. Widać, że nawet w porównaniu do poprzedniego przypadku błąd ten wzrósł.

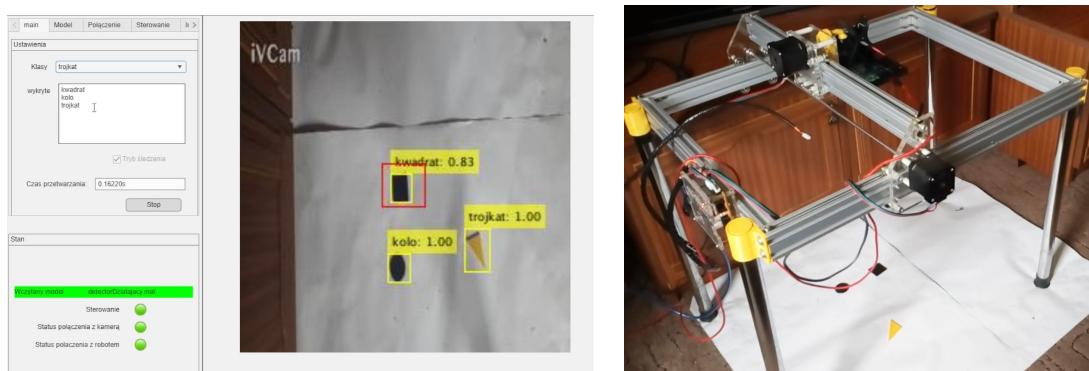


Rysunek 7.36: Test trybu normalnego - znalezienie koła

Ostatni przeprowadzony test polegał na znalezieniu koła. Ponownie, jak w poprzednich przypadkach, pojawił się względnie spory błąd w wyśledzonym obiekcie. Analizując wszystkie te przypadki, można zauważyc, że błąd ten rośnie wraz ze zwiększeniem się odległości pomiędzy pozycją startową a końcową. Prawdopodobnie w głównej mierze wynika to z nie dokładne dobranego współczynnika p w regulatorze wyznaczającym przesunięcie. Warto zauważyc, że w tym trybie algorytm potrzebuje średnio około 400-500ms na analizę obrazu.

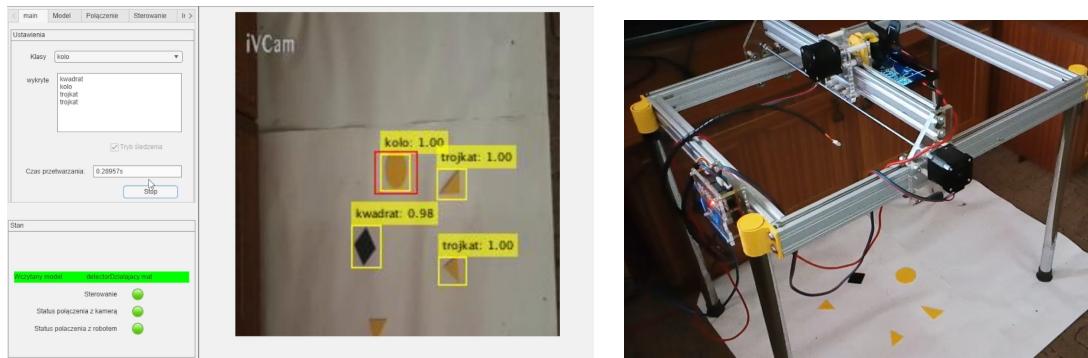
7.2. Testy trybu śledzenia

W związku z błędami widocznymi w poprzednim teście, opracowano drugą wersję algorytmu, który uruchomiony był w pętli i ciągle kompensował błędy pozycjonowania. Nazwany został śledzącym, przez to, że potrafi aktywnie śledzić dynamicznie poruszający się po scenie obiekt. Aby uruchomić ten tryb, należy wykonać te same kroki co w poprzedniej wersji, jednak przed uruchomieniem należy zaznaczyć opcje tryb śledzenia.



Rysunek 7.37: Tryb śledzący, widok z programu i boku robota

Jak widać w tym przypadku, sieć neuronowa analizuje każdą klatkę obrazu (widoczne są wykryte obiekty). Dzięki ciągłej korekcji błędów, docelowy kształt znajduje się praktycznie idealnie na środku, a po ręcznym przesunięciu pozycja jest automatycznie korygowana. Warto zwrócić uwagę na czas przetwarzania, który wynosi około 110ms, co jest znacznie lepszym wynikiem w porównaniu do poprzedniego trybu.



Rysunek 7.38: Śledzenie koła

Podobnie jak poprzednio, śledzenie koła działa równie dobrze, a osiągane błędy są bardzo małe. W porównaniu do poprzedniego trybu, w tym widoczne są duże różnice w sposobie sterowania. W pierwszej wersji algorytmu, cały ruch wykonywany był za jednym razem, a w trybie śledzenia pozycja zmieniana jest w małych krokach.

8. Podsumowanie

Budowa robota

Analizując pracę i konstrukcję zbudowanego robota, można zauważyc, że ogólnie konstrukcja spełniła podstawowe założenia pracy inżynierskiej, jednak można poprawić konstrukcje w wielu miejscach. Podczas finalnej pracy i testowania roboczej wersji algorytmu, gdy ten posiadał sporo błędów, przydatne okazałyby się czujniki krańcowe, które nie pozwoliłyby na próbę wyjazdu karetki robota poza poprawny obszar pracy. Takie wjazdy bardzo zmniejszają żywotność silników, ich sterowników (duży płynący przez uzwojenie prąd) oraz mechanicznych elementów, takich jak paski czy łożyska.

Kolejne konstrukcyjne ulepszenie powinno skupić się na zwiększeniu sztywności nóg utrzymujących ramę. Uzyskana sztywność była wystarczająca, jednak podczas szybkich i gwałtownych ruchów robota było widoczne chybotanie się całej konstrukcji.

Największym problemem okazało się sterowanie w otwartej pętli, które powodowało spore błędy w sterowaniu 'jednorazowym'. Błędy te były spowodowane w znacznej części przez niedokładnie dobrany współczynnik skali oraz przez brak 'świadomości' robota o własnej pozycji i korygowanie jej względem oczekiwanej.

Aby poprawić ten błąd należały dodać do silników enkodery oraz system pozycjonowania (np. dojazd do rogu robota i określenie tej pozycji jako 0,0), dzięki czemu byłoby możliwe uruchomienie sterowania ze sprzężeniem zwrotnym. Dodatkowo dzięki dodaniu systemu pozycjonowania, algorytm śledzący mógłby dokładniej wyznaczać docelowe koordynaty oraz monitorować je w trakcie pracy.

Powyższe błędy udało się zminimalizować trybem "śledzenia", które aktywnie w trakcie pracy kompensowało błędy wynikające w wyżej wymienionych niedokładności.

Sieć neuronowa i algorytm sterowania

Podobnie jak w przypadku fizycznej części robota, udało się osiągnąć założony efekt, jednak w trakcie rozwoju pojawiło się sporo miejsca na usprawnienia. Uważam, że wytrenowana sieć neuronowa bardzo dobrze radzi sobie z wykrywaniem danych obiektów i to niezależnie od ich orientacji, położenia, koloru czy konfiguracji (np. różne typy trójkątów). Analizując model sieci neuronowej, można dojść do wniosku, że jest ona zbyt dużą i powolną jak na tak proste zadanie i rozpoznawanie figur geometrycznych.

Kolejny projekt powinien bazować na zdecydowanie mniejszej sieci neuronowej (np. opisana w rozdziale 5.1 architektura sieci tiny-yoloV4), co pozwoliłoby zwiększenie

szybkości przetwarzania, zwłaszcza na słabszym komputerze, np. Rassbery Pi.

Kolejnym usprawnieniem, jakie można wprowadzić to dodanie danych z obiektami będącymi na zróżnicowanym tle, co pozwoliłoby na działanie w zdecydowanie bardziej zróżnicowanych warunkach.

Posiadając już doświadczenie z tworzeniem aplikacji okienkowych w Matlabie, uważam że komercyjne rozwiązanie powinno być napisane przy pomocy zdecydowanie wydajniejszego językam takiego jak C++ lub podobnego. Ze względu na bardzo szerokie zastosowanie Matlaba, wygenerowana aplikacja zajmowała bardzo dużo miejsca na dysku, a zapisywany model mógłby być ograniczony jedynie do wag sieci, co zmniejszyło by jego uniwersalność w modyfikacji, ale poprawiło zużycie dysku. Dodatkowo dzięki zastosowaniu technologii CUDA, można byłoby lepiej wykorzystać dostępne sprzętowe zasoby.

Literatura

- [1] <https://batmaja.com/sztuczny-neuron/> Dostęp: 07.01.2023
- [2] <https://www.v7labs.com/blog/machine-learning-guide> Dostęp: 07.01.2023
- [3] <https://www.mathworks.com/help/vision/ug/object-detection-using-yolov4-deep-learning.html> Dostęp: 26.01.2023
- [4] <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e> Dostęp: 06.06.2023
- [5] Joseph Redmon, Ali Farhadi "YOLOv3: An Incremental Improvement", University of Washington 2018
- [6] Douglas Henke Dos Reis, Mobile Robot Navigation Using an Object Recognition Software with RGBD Images and the YOLO Algorithm DOI: 10.1080/08839514.2019.1684778
- [7] Projekt grbl, <https://github.com/gnea/grbl/> Dostęp 08.06.2023r
- [8] <https://www.mathworks.com/help/vision/ug/getting-started-with-yolo-v4.html> Dostęp 04.06.2023