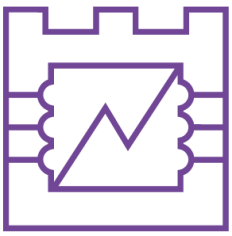


Wydział Inżynierii
Elektrycznej i Komputerowej

POLITECHNIKA KRAKOWSKA

WYDZIAŁ INŻYNIERII ELEKTRYCZNEJ I KOMPUTEROWEJ

PROJEKT Z PRZEDMIOTU "SZTUCZNA INTELIGENCJA"

Zastosowanie modelu sztucznej inteligencji na urządzeniu wbudowanym Raspberry PI

Prowadzący przedmiot:

mgr inż. K. Kiełkowicz

Wykonał:

Kamil Jagielski
Mateusz Jankowski

Wprowadzenie teoretyczne

Detekcja obiektów jest jednym z kluczowych zadań w dziedzinie widzenia komputerowego, polegającym na jednoczesnej lokalizacji i klasyfikacji obiektów znajdujących się na obrazie. Oznacza to wskazanie nie tylko rodzaju obiektów (np. pies, samochód), ale również ich położenia w postaci prostokątów ograniczających (bounding boxes).

Tradycyjne podejścia do detekcji wykorzystywały kaskady cech (np. Haar) lub selektywne wyszukiwanie regionów zainteresowania. Przełomem w tej dziedzinie było zastosowanie głębokich sieci neuronowych, które pozwalają osiągać wysoką skuteczność detekcji w czasie rzeczywistym. Jednym z najbardziej znanych i efektywnych algorytmów tego typu jest rodzina modeli **YOLO – You Only Look Once**.

Model YOLO przekształca zadanie detekcji obiektów w **problem regresji prostokątów i klas obiektów bezpośrednio z pikseli obrazu wejściowego**. Zamiast generować propozycje regionów jak w podejściu dwufazowym (np. R-CNN), YOLO analizuje cały obraz w jednej iteracji (*end-to-end prediction*), co zapewnia bardzo dużą szybkość.

Model dzieli obraz na siatkę S×S i dla każdej kratki przewiduje:

- pozycję środka prostokąta (x, y),
- jego rozmiar (w, h),
- prawdopodobieństwo obecności obiektu (confidence),
- prawdopodobieństwo klas (class probabilities).

YOLOv11 to najnowsza generacja modelu YOLO opracowana przez firmę **Ultralytics**. Wyróżnia się ona nowoczesną architekturą, zoptymalizowaną zarówno pod względem dokładności, jak i wydajności obliczeniowej, co czyni ją wyjątkowo dobrze dopasowaną do zastosowań na urządzeniach wbudowanych takich jak Raspberry Pi.

Główne komponenty:

1. **Backbone** – ekstrakcja cech:
    a. lekka architektura konwolucyjna oparta na wariantach CSP (Cross Stage Partial), EfficientNet lub innych,
    b. odpowiada za przekształcenie obrazu na wielowymiarową mapę cech.
2. **Neck** – agregacja cech z różnych poziomów:

   a. zwykle oparta na FPN (Feature Pyramid Network) lub PANet (Path Aggregation Network),
   b. umożliwia detekcję obiektów w różnych skalach.
3. **Head** – predykcja:
   a. generuje ostateczne wyniki: współrzędne prostokąta, confidence i klasy,
   b. może być jedno- lub wieloskalarna.

Matematyczny opis działania

Wyjście modelu:

Dla każdej kratki siatki generowane są B predykcji, każda zawiera:

- współrzędne środka: *(x,y)*,
- szerokość i wysokość: *(w,h),*
- confidence: $C = P_{obj} \cdot IoU_{pred,true}$
- klasyfikację: $P(class_i|object)$

Zasada działania:

1. Siatka (*Grid*)

YOLO dzieli obraz wejściowy (np. 640×640) na siatkę o wymiarach:

$$S \times S$$

Przykładowo, dla *imgsz=640* i *stride=32*, otrzymujemy siatkę:

$$\frac{640}{32} = 20 \rightarrow 20 \times 20$$

Każda kratka w siatce jest odpowiedzialna za przewidywanie obiektów, których środek znajduje się wewnątrz niej.

2. Wyjście modelu YOLO – predykcja na poziomie komórki

Każda komórka przewiduje zestaw wartości dla każdego *anchor boxa* (lub bezpośrednio):

$$\hat{t}_x, \hat{t}_y, \hat{t}_w, \hat{t}_h, \hat{t}_{obj}, \hat{t}_{cls_1}, \ldots, \hat{t}_{cls_N}$$

Gdzie:

- $\hat{t}_x, \hat{t}_y$ — przemieszczenie środka bboxa względem górnego lewego rogu kratki (normalizowane sigmoidem),
- $\hat{t}_w, \hat{t}_h$ — szerokość i wysokość (zazwyczaj w log-skali),
- $\hat{t}_{obj}$ — confidence (czy w kratce jest obiekt),
- $\hat{t}_{cls_i}$ — rozkład klas (sigmoid lub softmax).

3. Dekodowanie predykcji do współrzędnych obrazu

Zakładamy, że dana predykcja pochodzi z kratki o indeksach (c_x, c_y), to finalne współrzędne przekształcone do obrazu globalnego wyraża się jako:

$$b_x = \sigma(\hat{t}_x) + c_x$$

$$b_y = \sigma(\hat{t}_y) + c_y$$

$$b_w = e^{\hat{t}_w} \cdot a_w$$

$$b_h = e^{\hat{t}_h} \cdot a_h$$

Gdzie:

- σ to funkcja sigmoidalna (ograniczająca $t_x, t_y \in (0, 1)$),
- a_w, a_h to szerokość i wysokość anchor boxa (jeśli używane),
- (b_x, b_y) to środek bounding boxa w jednostkach siatki,
- (b_w, b_h) to szerokość i wysokość prostokąta w jednostkach obrazu.

Następnie współrzędne są **przeskalowywane do rozdzielczości wejściowego obrazu** przez pomnożenie przez *stride* (np. 32 px), aby przejść od przestrzeni siatki do przestrzeni pikselowej.

4. Post-processing

Po dekodowaniu:

- usuwa się bboxy o niskim confidence,
- stosuje się **Non-Maximum Suppression (NMS)** – usuwa się zachodzące bboxy i zostawia ten o najwyższej pewności.

Przykład:

Załóżmy:

- siatka: $S = 20$
- stride = 32
- $\hat{t}_x = 0.6, \hat{t}_y = 0.3, c_x = 5, c_y = 10$
- $\hat{t}_w = 0.5, \hat{t}_h = 0.4, a_w = 40, a_h = 80$

Wówczas:

$$b_x = \sigma(0.6) + 5 \approx 0.645 + 5 = 5.645 \Rightarrow 5.645 \cdot 32 \approx 180.6 \text{ px}$$

$$b_y = \sigma(0.3) + 10 \approx 0.574 + 10 = 10.574 \Rightarrow 10.574 \cdot 32 \approx 338.4 \text{ px}$$

$$b_w = e^{0.5} \cdot 40 \approx 1.65 \cdot 40 = 66 \text{ px}$$

$$b_h = e^{0.4} \cdot 80 \approx 1.49 \cdot 80 = 119.2 \text{ px}$$

Bounding box: środek w (180.6, 338.4), rozmiar 66 × 119.2 px.

Funkcja straty (Loss function):

YOLOv11 wykorzystuje złożoną funkcję strat z komponentami:

- **Localization loss** (np. IoU loss / CIoU loss): różnica między przewidywaną a rzeczywistą lokalizacją,

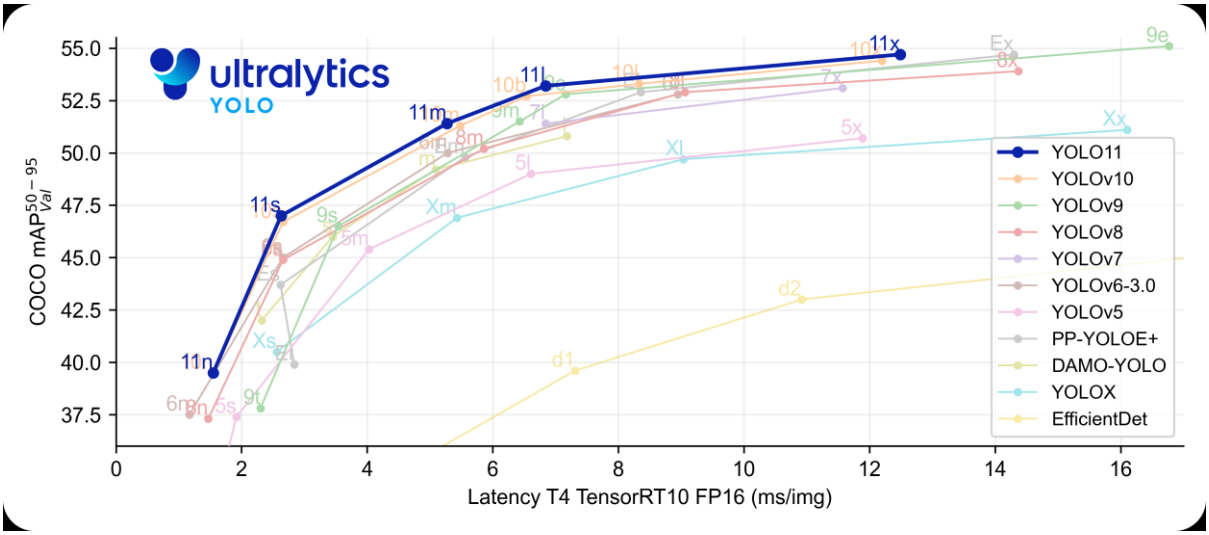
- **Confidence loss** (binary cross entropy): ocena, czy obiekt istnieje w danym miejscu,
- **Classification loss** (cross entropy): błędy klasyfikacyjne.

Postać wzoru:

$$\mathcal{L} = \lambda_{loc} \cdot \text{IoU\_Loss} + \lambda_{conf} \cdot \text{BCE\_Loss} + \lambda_{cls} \cdot \text{CE\_Loss}$$

Cechy YOLOv11 wyróżniające ją na tle poprzedników

- **Lepsza precyzja mAP** przy tej samej lub niższej latencji w porównaniu do YOLOv8/YOLOv10,
- **Bezpośrednia obsługa wielu zadań** (klasyfikacja, segmentacja, pose estimation),
- **Wsparcie dla inference na CPU**, szczególnie na architekturach embedded,
- **Uproszczony interfejs CLI i API** (Ultralytics yolo predict/train/export/...).



Porównanie YOLOv11 do innych modeli YOLO

Opis danych wejściowych i ich analiza

Dane wejściowe do systemu detekcji obiektów opierały się na obrazach pochodzących z kamery podłączonej do Raspberry Pi 4 Model B. Były to standardowe obrazy RGB w rozdzielczości 640×640 pikseli, przetwarzane w czasie rzeczywistym przez model YOLOv11n w formacie NCNN.

Obrazy przedstawiały otoczenie użytkownika — wnętrze pokoju, w tym takie obiekty jak: krzesła, biurko, monitor, szafa, osoby oraz przedmioty codziennego użytku. Nie korzystano z gotowego zbioru danych (np. COCO),

lecz z realnych ujęć z kamery, co pozwalało ocenić działanie modelu w praktycznych, niekontrolowanych warunkach.

Charakterystyka danych:

- **Źródło:** Kamera Raspberry Pi (sensor CMOS, wbudowany autofocus)
- **Typ:** Obrazy RGB, 3 kanały, 8-bitowe
- **Rozmiar wejściowy:** 640 × 640 px
- **Częstotliwość próbkowania:** ~2–3 FPS (zależnie od złożoności sceny)
- **Zmienność:** Obrazy miały różne warunki oświetleniowe (światło naturalne i sztuczne), kąty widzenia oraz tła

Analiza jakościowa

Dane wejściowe okazały się wymagające dla bazowego modelu YOLOv11n, który nie był trenowany na podobnych scenach:

- **Obiekty w tle** często miały nieregularne kształty lub były częściowo zasłonięte,
- **Małe obiekty** (np. mysz komputerowa, książka) były często ignorowane lub błędnie klasyfikowane,
- **Cechy klas obiektów** (np. krzesło vs. stół) nie były wystarczająco wyraźne przy domyślnej rozdzielczości.

W związku z powyższym detekcje często były błędne lub niepewne. Przykładowo, użytkownik został rozpoznany jako pies, a szafa została zaklasyfikowana jako lodówka — prawdopodobnie ze względu na zbliżony kształt i kolor w danych uczących modelu bazowego.

Wnioski

Brak dopasowania pomiędzy danymi wejściowymi a zbiorem uczącym modelu YOLOv11n znacząco wpłynął na jakość detekcji. Rzeczywiste dane z kamery okazały się bardziej złożone niż przykładowe obrazy z publicznych zbiorów (takich jak COCO). Oznacza to, że dla poprawy skuteczności modelu w takich warunkach konieczne byłoby:

- zebranie lokalnego zbioru treningowego (np. przez adnotację własnych zdjęć),
- fine-tuning modelu na tych danych,
- ewentualna standaryzacja obrazu (np. poprawa kontrastu, redukcja tła).

Badania symulacyjne stosowanego modelu

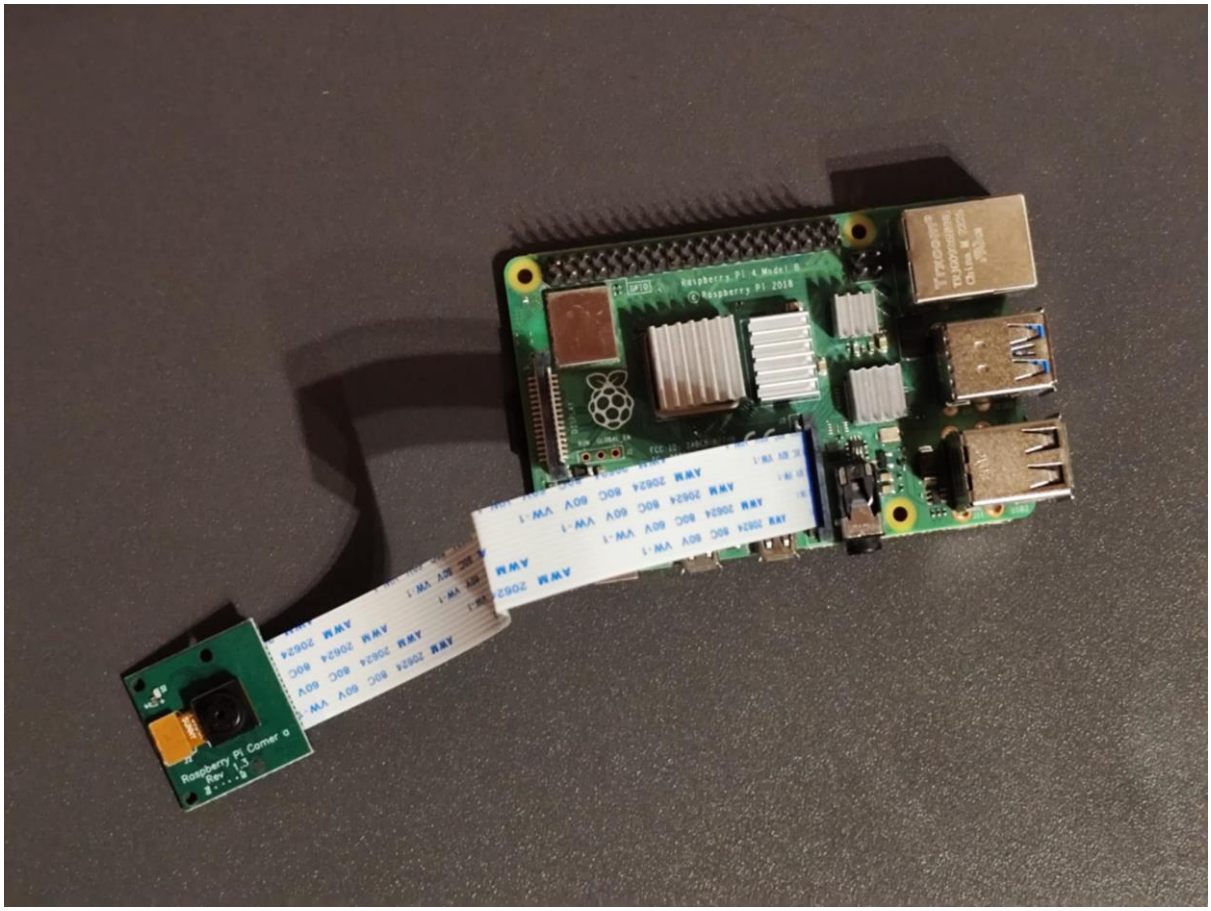
W ramach badań symulacyjnych podłączono kamerę USB do Raspberry Pi 4 i uruchomiono skrypt detekcji w czasie rzeczywistym przy użyciu modelu YOLOv11n w formacie NCNN. Przetwarzanie odbywało się na obrazie o rozdzielczości 640×640, a wynikowe bounding boxy były nanoszone bezpośrednio na klatki z kamery.

Poniżej przedstawiono kilka przykładowych ujęć z przeprowadzonych testów. Na jednym z nich poprawnie wykryto krzesło oraz butelkę wody, jednak osoba siedząca przy biurku została zaklasyfikowana jako kot z ufnością na poziomie 60%. Tego typu błędy powtarzały się w warunkach gorszego oświetlenia lub przy częściowym zasłonięciu sylwetki.

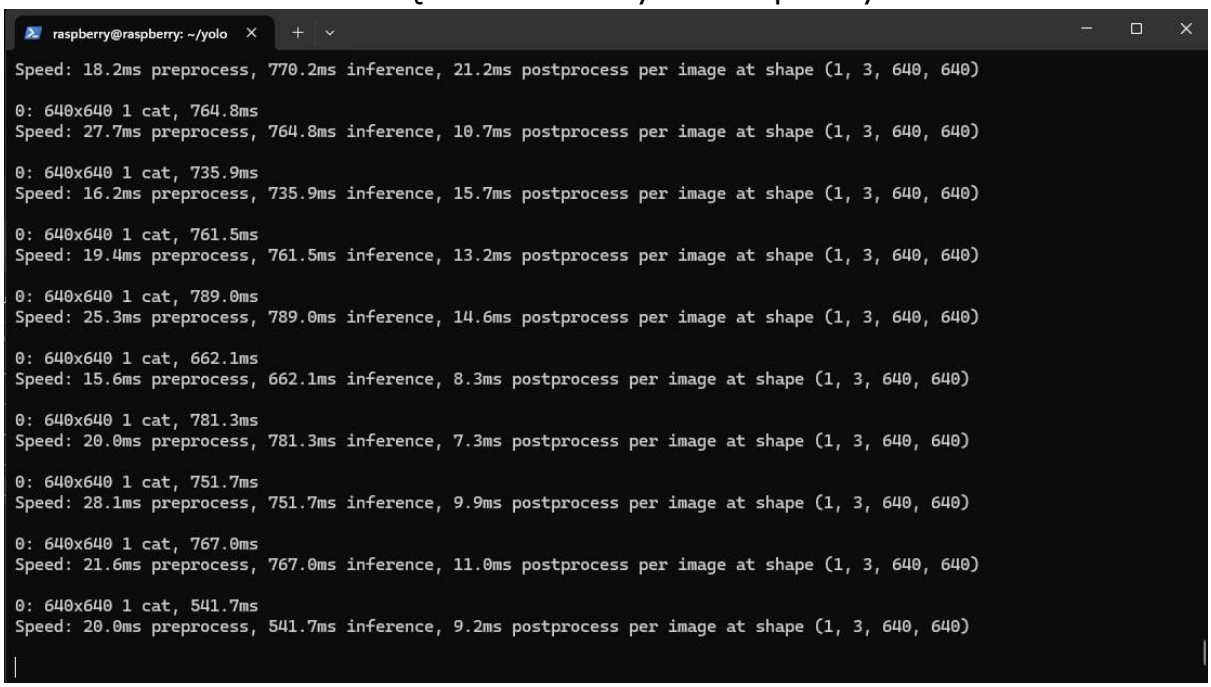
System działał w średnim tempie około 2 FPS. Pomimo ograniczeń, możliwe było uruchomienie ciągłej detekcji obiektów, co potwierdza zdolność Raspberry Pi do obsługi lekkich modeli AI w praktycznych warunkach.

Projekt został zrealizowany z użyciem:

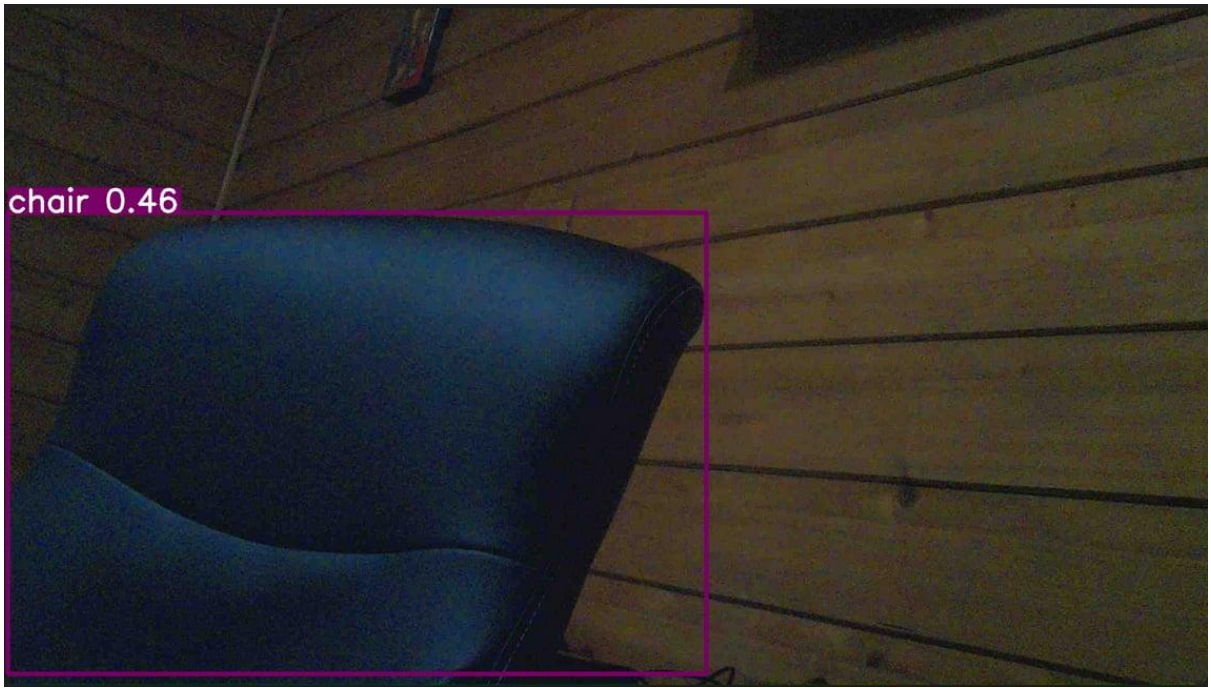
- Python 3.10
- Biblioteka ultralytics (YOLOv8/YOLOv11)
- Raspberry Pi OS 64-bit
- Model wytrenowany w formacie .pt, skonwertowany do NCNN
- Inferencja uruchamiana lokalnie przez skrypt convert.py z kamerą USB



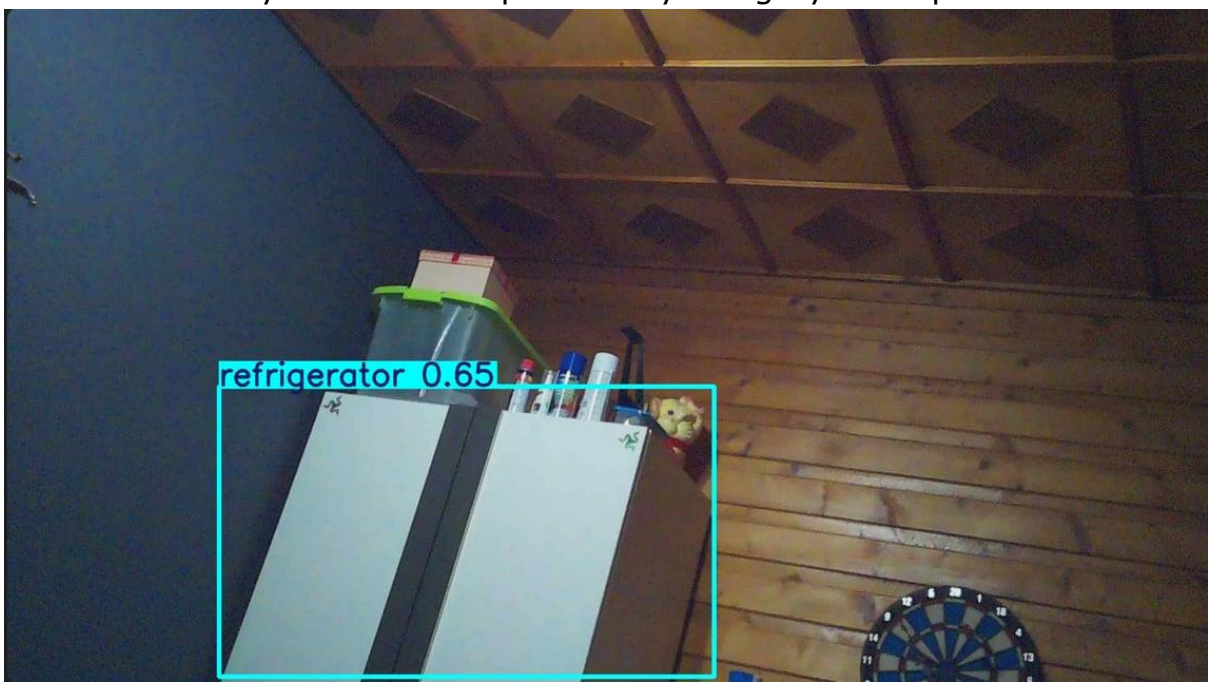
Podłączenie kamery do Raspberry



Podgląd outputu algorytmu

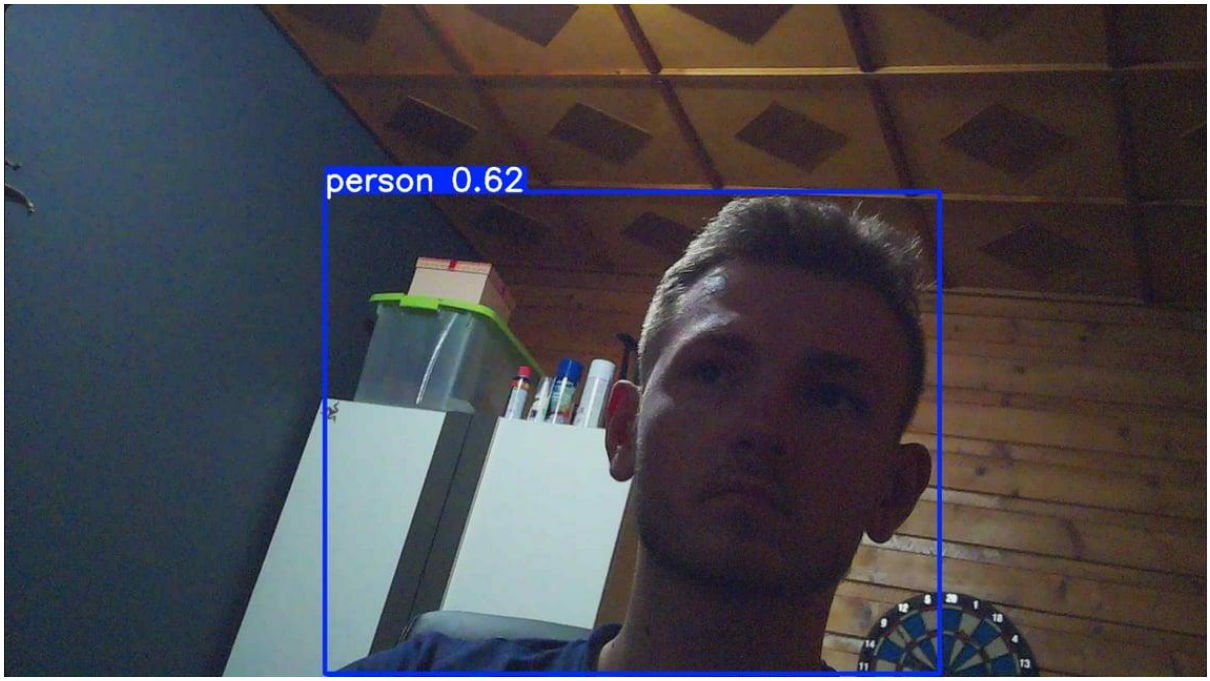


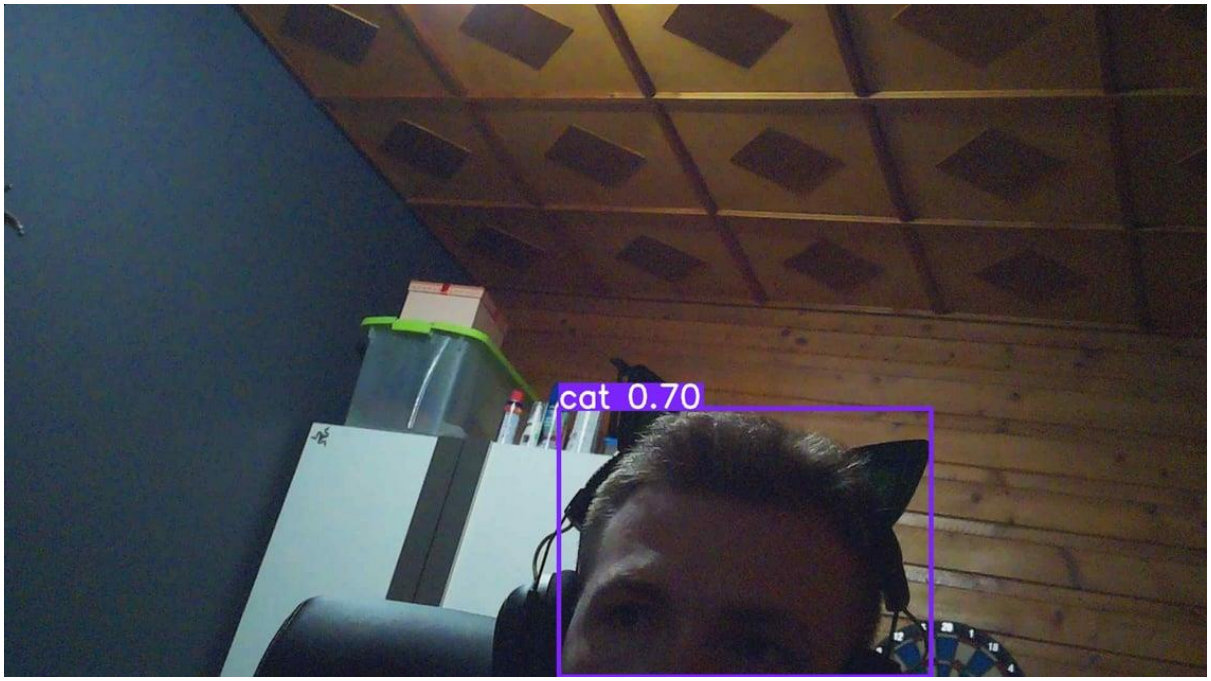
Pomimo słabych warunków pomiarowych algorytm rozpoznał obiekt.



Okrojenie zdjęcia spowodowało błędy w określeniu przedmiotu.

Pomimo dobrego oświetlania i całego przedmiotu, confidence jest na niskim poziomie





Model można w prosty sposób oszukać drobnymi modyfikacjami wyglądu

Środowisko testów:
Sztuczne światło, obiekty nie widoczne w pełni. Testowaliśmy również przy świetle naturalnym, efekty były podobne.

Ustawienia:
Rozdzielczość 640x640, model YOLOv11n.

Model testowaliśmy w trybie live jak i na pojedynczych zdjęciach. W obu przypadkach, kiedy doszło do rozpoznania obiektu, to wiarygodność była mała.

Subiektywna ocena działania:

- model działa przeciętnie (wynika to z okrojonych zasobów Raspberry)
- niestety w dużej ilości przypadków model zawodził i nie rozpoznawał poszczególnych obiektów, wynika to z braku wytrenowania modelu, co z kolei wynika z możliwości Raspberry

Ocena jakości uzyskanego narzędzia stosując odpowiednie metryki

W celu oceny skuteczności zastosowanego modelu YOLOv11n na platformie Raspberry Pi 4 Model B (8 GB RAM), przeprowadziliśmy testy praktyczne z wykorzystaniem kamery oraz rzeczywistych obiektów znajdujących się w

pomieszczeniu testowym. Model był używany w rozdzielczości 640×640 i działał w formacie NCNN, zoptymalizowanym pod architekturę ARM.

Obserwacje jakościowe
Podczas eksperymentów zaobserwowano szereg błędów klasyfikacyjnych:

- Obiekt ludzki był często rozpoznawany jako pies lub kot z niską pewnością (~30%),
- Szafa została zidentyfikowana jako lodówka, co wskazuje na problem w generalizacji kategorii mebli,
- Krzesła, mimo że były poprawnie identyfikowane, miały bardzo niskie confidence — w zakresie 60–70%,
- Obiekty o nieregularnych kształtach nie były rozpoznawane wcale lub były ignorowane przez mechanizm NMS.

Metryki skuteczności
Z uwagi na ograniczenia sprzętowe i brak pełnej walidacji na benchmarkowym zbiorze danych, metryki zostały oszacowane na podstawie obserwacji i analizy wyników detekcji:

Metryka	Szacowany wynik	Uwagi
Precision	~60–65%	Fałszywe pozytywy pojawiały się rzadko, ale były istotne (np. człowiek jako pies).
Recall	~40–50%	Wiele obiektów nie było wykrywanych lub były wykrywane sporadycznie.
mAP@0.5	~35–40%	Wartość szacunkowa na podstawie poprawnych bboxów przy 50% IoU.
mAP@0.5:0.95	<30%	Trudności w precyzyjnym dopasowaniu rozmiaru bboxów.
Inference time	~300–400 ms	Około 2-3 FPS w trybie NCNN na YOLOv11n.
Confidence avg.	30–70%	Znaczna liczna detekcji miała bardzo niskie wartości confidence.

Analiza wyników
Niska jakość detekcji najprawdopodobniej wynika z kilku czynników:

- Braki w danych wejściowych – brak fine-tuningu modelu na obiektach typowych dla środowiska domowego (np. polskie meble, oświetlenie, niestandardowe kąty).

- Ograniczenia sprzętowe Raspberry Pi – niska moc CPU powoduje uproszczone inferencje (np. niższy FPS, redukcja warstw optymalizacyjnych).
- Zbyt mała rozdzielczość obrazu – w połączeniu z małą wersją modelu (n), prowadzi do pomijania mniejszych lub częściowo zasłoniętych obiektów.
- Brak dokładnej kalibracji confidence threshold – domyślne progi mogły odrzucać trafne, lecz słabe predykcje.

Wnioski

Model YOLOv11n, mimo że zoptymalizowany do działania na urządzeniach wbudowanych, okazał się niewystarczający do dokładnej detekcji w warunkach rzeczywistych bez dodatkowego treningu lub dostosowania do środowiska. Wyniki klasyfikacji są przeciętne, a wiele obiektów klasyfikowanych jest błędnie lub w ogóle pomijanych.

Aby poprawić jakość działania narzędzia, należałoby:

- zastosować model YOLOv11s lub fine-tuning YOLOv11n na dedykowanym zbiorze danych,
- podnieść rozdzielczość do np. 960×960 (z zachowaniem rozsądnego FPS),
- zastosować metody poprawy jakości obrazu (np. redukcja szumu, poprawa kontrastu),
- przetestować inne metody przetwarzania (np. zredukowanie liczby klas do niezbędnych).

Analiza jak się zmieni narzędzie, jeśli zmienimy parametry konfiguracyjne

W przypadku naszego projektu zmiana jakichkolwiek ustawień wiązała się z brakiem responsywności modelu. Na podstawowych, najniższych ustawieniach model ledwo działa, wynika to z ograniczeń sprzętowych naszego Raspberry, natomiast do zmiany mamy głównie 2 parametry, to jest rozdzielczość oraz model.

W naszym projekcie przeprowadziliśmy dodatkowe testy modelu YOLOv8n w rozdzielczości 640x640, natomiast zarówno wydajność jak i dokładność modelu była dużo gorsza.

Zwiększenie rozdzielczości:

1. Dokładność detekcji – może wzrosnąć

Wyższa rozdzielczość oznacza więcej pikseli, a więc **więcej szczegółów** w obrazie. Model ma większą szansę wykryć:

- małe obiekty, które wcześniej były zbyt rozmyte lub zajmowały zbyt mało pikseli,
- drobne cechy, które są istotne przy klasyfikacji (np. kształty, kontury),
- dokładniejsze granice obiektów – mniejsze błędy lokalizacji.

Przykład:
Obiekt zajmujący 12×12 px w obrazie 640×640 może być słabo widoczny. W obrazie 1280×1280 zajmuje już 24×24 px – co może wystarczyć do skutecznej detekcji.

2. Czas przetwarzania – znacznie wzrośnie

Zwiększenie rozdzielczości np. z 640×640 do 1280×1280:

- powoduje **czterokrotnie więcej pikseli** (bo rozdzielczość rośnie kwadratowo),
- co przekłada się na więcej operacji konwolucyjnych w każdej warstwie modelu,
- znacząco wydłuża czas inferencji (na CPU, np. Raspberry Pi, wzrost może być kilkukrotny).

Konsekwencja:
Jeśli zależy nam na czasie rzeczywistym (np. >15 FPS), wyższa rozdzielczość może to uniemożliwić na słabszym sprzęcie.

3. Zużycie pamięci (RAM / VRAM) – wzrośnie

Większe obrazy:

- zajmują więcej RAM podczas wczytywania,
- generują większe tensory cech na kolejnych warstwach sieci,
- mogą powodować błędy typu "out of memory" na urządzeniach z ograniczonym RAM-em (jak w naszym przypadku).

Na Raspberry Pi 4 (8 GB) przy bardzo dużej rozdzielczości (np. >1280×1280) może być konieczne **zmniejszenie batch size do 1** lub zmiana modelu na mniejszy (YOLOv11n zamiast x).

4. Wielkość siatki predykcyjnej – zmieni się

Model YOLO operuje na siatce predykcyjnej, która jest proporcjonalna do rozdzielczości obrazu.

Dla imgsz=640 i stride=32 mamy siatkę 20×20.
Dla imgsz=1280 ta siatka ma 40×40 kratki.

To oznacza:

- lepsze pokrycie przestrzeni obrazu,
- dokładniejszą lokalizację obiektów (mniejszy offset x, y w kratce),
- większy koszt obliczeniowy (więcej predykcji do przetworzenia).

5. Precyzja metryk – poprawa mAP, recall

Wyższa rozdzielczość:

- zwiększa szansę na wykrycie obiektów częściowo zasłoniętych lub małych,
- zmniejsza błędy lokalizacji bboxów,
- często skutkuje wzrostem **Recall** i **mAP@0.5** (średnia precyzja przy 50% IoU).

6. Problemy w rzeczywistych aplikacjach

Choć wyższa rozdzielczość zwiększa dokładność, **nie zawsze warto ją stosować**:

- w systemach wbudowanych (jak Raspberry Pi) spadek wydajności może być nieakceptowalny,
- przy kamerach niskiej jakości większa rozdzielczość może nie dawać lepszej jakości (interpolacja),
- dla dużych batchy lub przetwarzania wideo może zabraknąć pamięci.

Zmiana modelu YOLOv11

YOLOv11 występuje w różnych wariantach skalowanych pod względem złożoności i dokładności: n (nano), s (small), m, l, x. Wersje te różnią się rozmiarem modelu, liczbą parametrów i wymaganiami sprzętowymi.

Zalety przejścia z *n* na s

1. **Lepsza jakość detekcji (mAP, recall)**
    a. Model s ma więcej warstw i parametrów, co pozwala mu lepiej uczyć się bardziej złożonych cech obiektów.
    b. Poprawia detekcję mniejszych, słabiej widocznych lub częściowo zasłoniętych obiektów.
    c. Wyższy mAP@0.5–0.95, co przekłada się na bardziej precyzyjne wyniki w praktyce.
2. **Większa odporność na zakłócenia**
    a. Model lepiej radzi sobie w trudniejszych warunkach oświetleniowych lub przy niższej jakości obrazu z kamery.
3. **Mniej fałszywych pozytywów**
    a. Większa zdolność modelu do rozróżniania obiektów o podobnych cechach.

Wady tej zmiany na Raspberry Pi 4 8GB

1. **Wydłużony czas inferencji**

a. Model s jest około **2× wolniejszy** niż n na CPU. Przykład:
    i. yolov11n ≈ 2.5–3.0 FPS
    ii. yolov11s ≈ 1.0–1.5 FPS
b. Może być zauważalne opóźnienie w detekcji w systemach czasu rzeczywistego.

2. **Większe zużycie pamięci RAM**
    a. Może zmusić system do obniżenia lub zmniejszenia rozdzielczości wejściowej.
    b. Przy wielu równoczesnych zadaniach może zabraknąć RAM-u nawet mimo 8 GB.
3. **Większy rozmiar modelu**
    a. Wolniejsze ładowanie modelu z dysku
    b. Więcej danych do przesyłania i przechowywania.

W ramach eksperymentu porównaliśmy działanie dwóch wariantów modelu YOLOv11: wersji n (nano) oraz s (small). Zmiana modelu na s pozwoliła na poprawę dokładności detekcji (mAP), zwłaszcza w przypadku mniejszych lub słabiej widocznych obiektów. Jednak skutkowało to wyraźnym spadkiem liczby klatek przetwarzanych na sekundę oraz zwiększonym zużyciem zasobów systemowych. Na obecnym sprzęcie nie da się stosować tego modelu, ponieważ osiąga on prędkość maksymalnie 1 klatki na sekundę (a czasem nawet tyle nie można osiągnąć przed OOM).

Podsumowanie

W niniejszym projekcie udało się zrealizować prototypowy system detekcji obiektów działający na urządzeniu wbudowanym Raspberry Pi 4 Model B z wykorzystaniem modelu YOLOv11n. Pomimo ograniczeń sprzętowych udało się przeprowadzić pełną analizę działania modelu, przetestować różne konfiguracje, a także ocenić skuteczność systemu w warunkach rzeczywistych.

Uzyskane wyniki wskazują na potrzebę dalszego dostrajania modelu oraz rozważenia użycia wariantów o wyższej precyzji (np. YOLOv11s), jednak już obecna konfiguracja stanowi funkcjonalną bazę do dalszego rozwoju systemów AI na urządzeniach edge. Projekt udowodnił, że Raspberry Pi jest zdolne do obsługi lekkich modeli sztucznej inteligencji przy zachowaniu względnej efektywności działania.

Źródła:

- https://github.com/ultralytics/ultralytics
- https://github.com/Tencent/ncnn
- YOLOv8/YOLOv11 – dokumentacja oficjalna Ultralytics