



WYDZIAŁ FIZYKI TECHNICZNEJ
I MATEMATYKI STOSOWANEJ

Imię i nazwisko studenta: Filip Zygumuntowicz

Nr albumu: 191213

Poziom kształcenia: Studia drugiego stopnia

Forma studiów: stacjonarne

Kierunek studiów: Matematyka

Specjalność: Geometria i grafika komputerowa

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Identyfikacja i badanie właściwości produktów spożywczych z wykorzystaniem machine learning i widzenia komputerowego

Tytuł pracy w języku angielskim: Identification and testing of food product properties using machine learning and computer vision

Opiekun pracy: dr inż. Bartosz Reichel

Student's name and surname: Filip Zygmuntowicz

ID: 191213

Cycle of studies: postgraduate

Mode of study: Full-time studies

Field of study: Mathematics

Specialization: Geometry and Computer Graphics

MASTER'S THESIS

Title of thesis: Identification and testing of food product properties using machine learning and computer vision

Title of thesis (in Polish): Identyfikacja i badanie właściwości produktów spożywczych z wykorzystaniem machine learning i widzenia komputerowego

Supervisor: dr inż. Bartosz Reichel

Politechnika Gdańsk
Wydział Fizyki Technicznej i Matematyki Stosowanej

FILIP ZYGMUNTOWICZ

Kierunek: Matematyka

Specjalność: Geometria Obliczeniowa i Grafika Komputerowa

Numer albumu: 191213

IDENTYFIKACJA I BADANIE WŁAŚCIWOŚCI
PRODUKTÓW SPOŻYWCZYCH Z WYKORZYSTANIEM
MACHINE LEARNING I WIDZENIA
KOMPUTEROWEGO

Praca magisterska napisana
pod kierunkiem dr inż. Bartosza Reichela

Gdańsk 2023

SPIS TREŚCI

1. Wstęp	2
2. Wprowadzenie do sieci neuronowych	4
2.1. Czym są sieci neuronowe?	4
2.2. Zastosowania sieci neuronowych	5
2.3. Definicje najważniejszych pojęć	7
2.4. Zasada działania	10
2.5. Konwolucyjne sieci neuronowe	15
2.6. Problemy rozwiązywane przez sieci neuronowe	18
2.7. Konstruowanie sieci neuronowych w praktyce	20
3. Projekt dyplomowy	23
3.1. Badanie błędu pomiaru	23
3.2. Trenowanie sieci neuronowej do detekcji obiektów	30
3.3. Opis ważniejszych fragmentów kodu aplikacji	37
3.4. Bilans i metoda połączenia go z aplikacją	44
3.5. Praktyczny test działania aplikacji	45
4. Podsumowanie	53
Źródła z linkami na dzień 29.11.2023	57

1. WSTĘP

W przeciągu ostatnich kilku lat pojęcie sztucznej inteligencji mocno zyskało na popularności. To, co kiedyś było jedynie teoretycznym konceptem i elementem filmów sci-fi, dzisiaj wydaje się być częścią rzeczywistości. Warto jednak zaznaczyć, że istniejące rozwiązania nie spełniają wszystkich kryteriów prawdziwej sztucznej inteligencji. W rzeczywistości są to jedynie algorytmy, które całkiem dobrze taką sztuczną inteligencję naśladują. Według oficjalnej definicji ze strony Parlamentu Europejskiego ([2]) sztuczna inteligencja to zdolność maszyn do wykazywania ludzkich umiejętności, takich jak rozumowanie, uczenie się, planowanie i kreatywność. Nasze rozwiązania wykazują co najwyżej zdolność do uczenia się. Definiujemy również Artificial General Intelligence (AGI [3]), czyli taką sztuczną inteligencję, która jest w stanie rozwiązać każdy problem logiczny jaki jest w stanie rozwiązać człowiek lub zwierzę. Do stworzenia prawdziwego AGI jest jeszcze przed nami bardzo daleka droga. Pierwszym krokiem, który musimy wykonać jest pełne zrozumienie procesów zachodzących w ludzkim mózgu. Obecnie nasza wiedza na ten temat jest dosyć powierzchowna. Przy użyciu tej powierzchownej wiedzy zostały stworzone sieci neuronowe ([4]), które są głównym tematem tej pracy. Sieci neuronowe to wiodąca metoda używana do naśladowania sztucznej inteligencji. Ich możliwości są ograniczone, opierają się na aproksymacji. Zwykły algorytm działa z praktycznie stu procentową skutecznością o ile jest wykorzystywany w przewidziany sposób, sieć neuronowa nigdy nie będzie w 100 procentach nieomylna. Mimo to, sieci są w stanie wykonywać zadania, które bez nich byłyby niesamowicie skomplikowane, a nawet praktycznie niemożliwe. Brak stu procentowej skuteczności nie jest również na tyle dużym problemem, na ile może się wydawać w przypadku zadań, w których sieć ma zastąpić człowieka. Przytoczymy przykład autonomicznych samochodów, które wykorzystują sieci do podejmowania decyzji na drodze ([5]). Często można spotkać się z opinią że nie powinny nigdy zostać dopuszczone do ruchu właśnie przez możliwość błędu, który może kosztować ludzkie życie. Zapomina się przy tym, że człowiek również popełnia błędy. Z pomocą statystyk, możemy wyliczyć częstotliwość ludzkich błędów i prawdopodobnie wynik ten nie będzie wiele niższy od marginesu błędu wytrenowanej sieci neuronowej, o ile nie wyższy. Wierzę, że sieci neuronowe będą grały główną rolę w rozwoju technologicznym w następnych latach, dlatego wybrałem je jako temat mojej pracy magisterskiej. Motywacją do wyboru tematu części projektowej było moje doświadczenie z aplikacjami służącymi do śledzenia bilansu kalorycznego.

Większość z nich działa na podobnej zasadzie. Za przykład weźmy aplikację Fitatu ([6]). Po otwarciu aplikacji wita nas tabela z produktami spożyтыmi danego dnia i opcja dodania kolejnego produktu. Jej wybór przeniesie nas do okna wyszukiwarki, w której możemy wyszukać interesujący nas produkt. Po wyszukaniu wybieramy produkt i aplikacja przenosi nas do następnego okna, w którym określamy jego wagę lub ilość, co dodaje go do bilansu. Z takiego sposobu działania aplikacji wynikają następujące problemy:

- Wyszukiwanie produktu nie zawsze jest łatwe, bo nie wiadomo pod jaką dokładną nazwą został dodany do bazy danych.
- Ten sam produkt jest często dodany wielokrotnie z różniącymi się wartościami odżywczymi, ponieważ baza danych jest aktualizowana przez użytkowników.
- Niektóre produkty mają wiele odmian, na przykład jabłka i użytkownik nie wie jaką aktualnie spożywa.
- Największy problem, czyli ilość czasu jaki zajmuje cały proces. W sumie w celu dodania jednego produktu użytkownik musi kliknąć w co najmniej 5 miejsc na ekranie.

Wymienione problemy mogą się wydawać mało istotne, ale cele dla których używa się takich aplikacji, jak schudnięcie, wymagają bardzo dużo samodyscypliny i motywacji. Sporo użytkowników może zrezygnować ze śledzenia kalorii przez to że proces korzystania z aplikacji jest zbyt nużący, dlatego należy go usprawnić na tyle na ile to możliwe.

Moja aplikacja umożliwia użytkownikowi zrobienie zdjęcia produktu i dodanie go do swojego jadłospisu jednym kliknięciem. Wytrenowana sieć neuronowa w czasie rzeczywistym przetwarza klatki podglądu kamery smartfona i zaznacza na ekranie pozycje wykrytych obiektów oraz ich przewidziane nazwy. Po naciśnięciu guzika wysyła ona potrzebne dane do bilansu, z których aproksymowana jest również waga obiektu i wartości odżywcze, które można dodać do śledzonego dziennego jadłospisu. Aplikacja została stworzona w języku programowania Python, sieć neuronowa została wytrenowana za pomocą biblioteki TensorFlow ([7]), a w celu przeniesienia jej na urządzenie mobilne skorzystałem z bibliotek Kivy ([8]) i pyjnius ([9]). Część z bilansem została napisana przy pomocy web frameworka Flask ([10]) oraz języków JavaScript, HTML i CSS z bazą danych SQLite. Pierwszą część pracy poświęcam na wstęp do teorii sieci neuronowych, omówione zostaną podstawowe zasady działania oraz kilka bardziej zaawansowanych aspektów, z których korzystałem przy projekcie. Druga część stanowi opis procesu twórczego tworzenia aplikacji oraz wyjaśnienie działanie poszczególnych jej komponentów. Ta część ma na celu pokazanie czytelnikowi procesu tworzenia aplikacji z wykorzystaniem sieci neuronowych od strony praktycznej. W celu **pełnego** zrozumienia rozdziałów z części teoretycznej oczekuję od czytelnika znajomości:

- (1) rachunku różniczkowego na poziomie Analizy Matematycznej II
- (2) języka angielskiego w stopniu podstawowym

Dla części projektowej:

- (1) języka Python i podstaw działania biblioteki Kivy
- (2) podstaw z zakresu web developmentu

2. WPROWADZENIE DO SIECI NEURONOWYCH

Ten rozdział ma na celu wprowadzenie czytelnika w świat sieci neuronowych. Przedstawiam ich zastosowania oraz zasadę działania zarówno od strony teoretycznej jak i praktycznej.

2.1. Czym są sieci neuronowe?

Sieci neuronowe należą do grupy algorytmów uczenia maszynowego, czyli algorytmów mających na celu generalizowanie i rozwiązywanie skomplikowanych problemów bez korzystania z dokładnych instrukcji, a poprzez "uczenie się". Za pomocą tych algorytmów oraz **danych treningowych** tworzy się modele, które na podstawie podanych danych, w iteratywnym procesie dostosowują składowe algorytmu. Algorytmy uczenia maszynowego dzielimy na 3 kategorie:

- Algorytmy uczenia nadzorowanego - takie, w których w procesie uczenia algorytm korzysta z ręcznie spreparowanych przez człowieka danych treningowych, specjalnie dostosowanych do potrzeb algorytmu. Przykładem takiego algorytmu są właśnie sieci neuronowe.
- Algorytmy uczenia nienadzorowanego - takie w których algorytm otrzymuje niespreparowane, surowe dane z których sam następnie wyciąga wnioski. Algorytmem z tej kategorii jest na przykład algorytm k-means, który grupuje przedstawione dane na k części sugerując się odległością między elementami w danej przestrzeni metrycznej. Często używa się takich algorytmów w celu wstępnego pogrupowania danych do innych zadań.
- Algorytmy uczenia przez wzmacnianie - takie, które korzystają ze sprzężenia zwrotnego, czyli podejmuje jakieś działanie i na podstawie jego wyniku stwierdza czy było prawidłowe czy nie. Często korzystają z systemu punktowego, które algorytm zdobywa za prawidłowe działania i traci za te nieprawidłowe. Najbardziej popularnym zastosowaniem jest tworzenie sztucznej inteligencji dla przeciwników w grach.

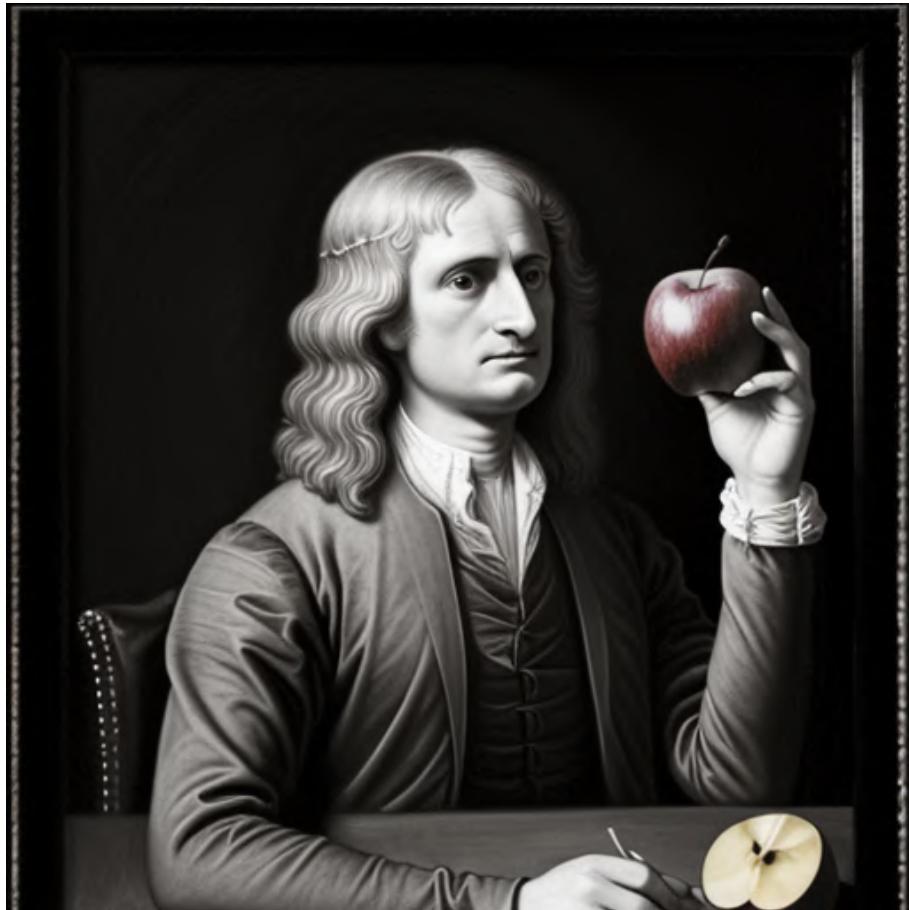
Sieci neuronowe inspirowane są budową ludzkiego mózgu, w którym procesy myślenia opierają się na sygnałach które wysyłają między sobą połączone w sieć neurony. Przy konstrukcji sieci neuronowej na samym początku decydujemy w jakim formacie sieć ma przyjmować dane i w jakim formacie ma je zwracać. Na przykład jeśli konstruujemy sieć klasyfikującą, która ma rozpoznawać cyfry na zdjęciu, to danymi wejściowymi są obrazy (o określonej rozdzielczości), a danymi wyjściowymi rozpoznana cyfra.

2.2. Zastosowania sieci neuronowych.

Definicja 2.1. **Promptem** nazywamy zestaw instrukcji w postaci zwykłego tekstu przekazywanego do wytrenowanego algorytmu w celu uzyskania zamierzonych efektów. Dużą zaletą promptów jest fakt że, są w formie zrozumiałej dla człowieka. Zazwyczaj są one zdaniem, którym opisalibyśmy rezultat, który chcemy otrzymać gdyby wykonywała dla nas zlecenie inna osoba.

Kilka znanych przykładów zastosowań sieci neuronowych, w których przy użyciu promptów można uzyskać ciekawe rezultaty:

Przykład 2.2. Modele DALL-E i Midjourney, lub open-sourcowy odpowiednik Stable Diffusion ([11, 12, 13]), czyli algorytmy do generowania obrazów. Danymi wejściowymi są między innymi prompt - opisujący co ma się znajdować na wygenerowanym obrazie oraz jego techniczne właściwości takie jak na przykład rozmiar soczewki aparatu, rozmycie, poziom kontrastu.



RYSUNEK 1. Przykład obrazu wygenerowanego przez Stable Diffusion z promptem *Isaac Newton eating an apple*

Przykład 2.3. Modele generujące głos, takie jak ElevenLabs lub open-sourcowe TurtleTTS i Tacotron2 + WaveGlow ([14, 15, 16]). Danymi wejściowymi jest w tym przypadku tekst który ma zostać wypowiedziany w wygenerowanym pliku dźwiękowym.

Przykład 2.4. Modele językowe takie jak GPT (wykorzystywany przez ChatGPT) lub PaLM (wykorzystywany przez Google Bard) ([17, 18]). Służą one do przetwarzania podanego prompta i wyciągania z niego istotnych treści, a następnie generowania sensownej odpowiedzi podobnie jak wyszukiwarka internetowa.

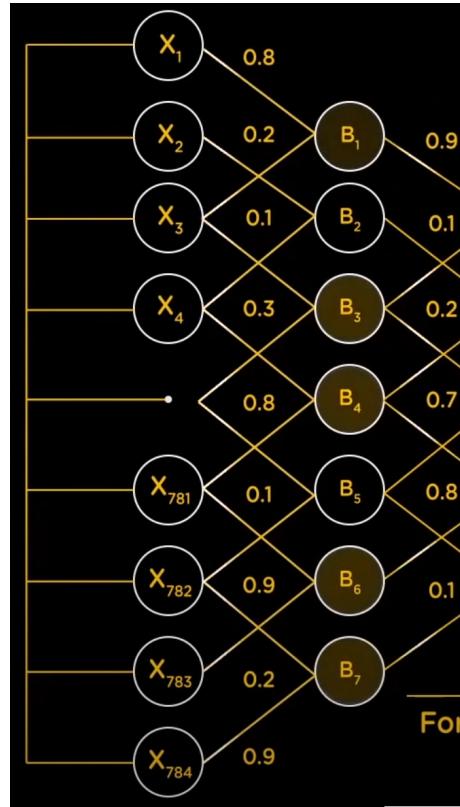
Poza modelami działającymi na zasadzie promptów wprowadzanych przez użytkownika mamy również sieci neuronowe przyjmujące jako dane wejściowe obrazy lub dźwięk.

Przykład 2.5. Amazon Alexa, Windows Cortana i inni głosowi asystenci korzystają z sieci neuronowej wytrenowanych, by z nagranego głosu otrzymać wypowiadany tekst.

Przykład 2.6. Praktycznie każdy produkowany dzisiaj smartfon posiada w fabrycznej aplikacji kamery algorytm do rozpoznawania twarzy. Korzysta on z sieci neuronowej, w której danymi wejściowymi jest obraz, a wyjściowymi współrzędne prostokąta w którym na zdjęciu znajduje się twarz.

2.3. Definicje najważniejszych pojęć.

Sieć neuronowa składa się z warstw neuronów połączonych ze sobą w taki sposób, że począwszy od pierwszej warstwy (wejściowej) każdy neuron połączony jest z każdym neuronem z warstwy następnej aż do ostatniej warstwy w sieci (wyjściowej). Zanim formalnie zdefiniujemy użyte pojęcia warto spojrzeć na ich reprezentację graficzną:



RYSUNEK 2. 784 neurony wejściowe połączone z 7 neuronami w warstwie następnej ([20])

Definicja 2.7. **Neuron** reprezentuje jednostkę danych przekazywanych w sieci neuronowej. Ma swoją wartość, wagi i bias. W warstwie wejściowej są to składowe spłaszczonej do 1-wymiarowego wektora wartości wejściowej, dlatego ich ilość zależy od wymiarów danych wejściowych.

Definicja 2.8. **Warstwa** to zbiór neuronów. Warstwa wejściowa znajduje się na początku sieci, wyjściowa na końcu, a warstwy ukryte znajdują się pomiędzy. Każda sieć posiada warstwę wejściową i wyjściową. Do warstw ukrytych i wyjściowych przyporządkowujemy funkcje aktywacji.

Definicja 2.9. Każdy neuron z warstwy wejściowej i ukrytej ma przyporządkowaną liczbę rzeczywistą zwaną **wagą**.

Definicja 2.10. **Bias** to liczba rzeczywista przyporządkowana do danego neuronu z warstwy ukrytej lub wyjściowej.

Definicja 2.11. Funkcja aktywacji to funkcja, której wynik stanowi wartość dla danego neuronu z warstwy ukrytej lub wyjściowej. Przyjmuje ona jako argument sumę kombinacji liniowych wszystkich neuronów z warstwy poprzedniej i ich wag powiększoną o bias danego neuronu. Mówiąc, że funkcje aktywacji w warstwach ukrytych "aktywują" dane neurony, ponieważ często działają one na zasadzie zwracania wartości 0 jeśli dany neuron nie spełnił wyznaczonych założeń, co eliminuje go z uczestnictwa w procesie wyznaczania kolejnych neuronów.

Przykład 2.12. Do najbardziej powszechnych funkcji tożsamościowych zaliczamy następujące funkcje:

- (1) Funkcja tożsamościowa.

$$id(x) = x$$

- (2) Funkcja progowa dla ustalonego progu p .

$$f(x) = \begin{cases} 0 & \text{dla } x < p \\ 1 & \text{dla } x \geq p \end{cases}$$

- (3) ReLU dla ustalonego progu p . Jest to najczęściej używana funkcja do trenowania sieci neuronowych pracujących na danych w postaci obrazów. Uważa się, że najlepiej odzwierciedla działanie neuronów biologicznych.

$$f(x) = \begin{cases} 0 & \text{dla } x < p \\ x & \text{dla } x \geq p \end{cases}$$

- (4) Sigmoid logistyczny. Charakteryzuje się tym, że każdy element jej przeciwoobrazu zawiera się w przedziale $(0,1)$ dzięki czemu wynik możemy traktować jako prawdopodobieństwo aktywacji danego neuronu.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

- (5) Funkcja softmax. Zwraca wektor z wartościami z przedziału $(0,1)$.

$$s(x_n) = \left(\frac{e^{x_1}}{\sum_{i=1}^n e^{x_i}}, \dots, \frac{e^{x_n}}{\sum_{i=1}^n e^{x_i}} \right)$$

Działanie algorytmu dla neuronów w danej warstwie można przedstawić następująco:

$$x_k = f(b_k + \sum_{i=1}^n w_i \cdot x_i), \text{ gdzie:}$$

f - funkcja aktywacji

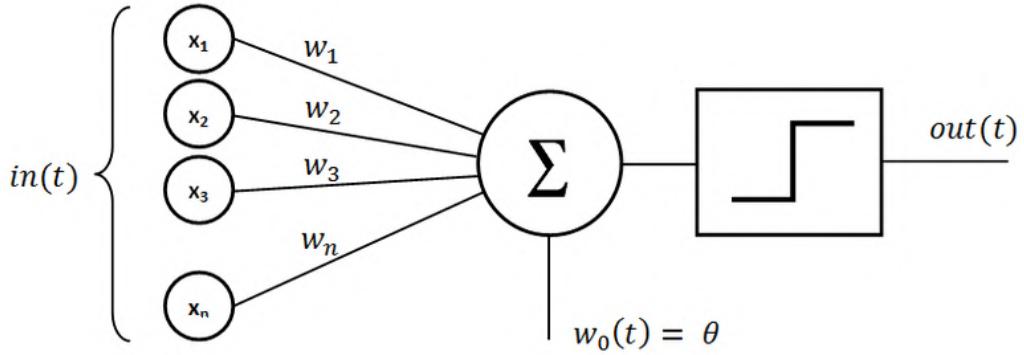
x_k - wartość neuronu k

b_k - bias neuronu k

w_i, x_i - waga i wartość neuronu i z warstwy poprzedniej

n - ilość neuronów w warstwie poprzedniej

Przykład 2.13. Najprostszy model sieci neuronowej - **Perceptron** ([19]), składa się z warstwy wejściowej i wyjściowej składającej się z jednego neuronu z funkcją aktywacji. Służy on do klasyfikacji binarnej, czyli przypisania wprowadzanych danych do jednej z dwóch klas.



RYSUNEK 3. Graficzne przedstawienie Perceptrona ([19])

Funkcją aktywacji w tym przypadku jest funkcja proporcowa. Przez brak jakichkolwiek warstw ukrytych perceptron ma bardzo ograniczone możliwości, często podawanym przykładem jest fakt że niemożliwe jest wytrenowanie go do naśladowania funkcji XOR.

2.4. Zasada działania.

Podsumowując: sieć neuronowa działa na zasadzie podawania danych z jednej warstwy do następnej aż do osiągnięcia danych wyjściowych zwracanych przez ostatnią warstwy. Proces uczenia polega na kolejnych iteracjach tego procesu, korzystając przy tym z odpowiednio dobranych danych treningowych i dostosowywaniu wag i biasu neuronów w warstwach w taki sposób, aby produkowały one oczekiwany wynik.

Definicja 2.14. **Danymi treningowymi** w kontekście sieci neuronowych nazywamy specjalnie dobrany zestaw danych wejściowych z **etykietami**, czyli z jakąś formą oznaczenia oczekiwanych danych wyjściowych. W przykładzie sieci klasyfikującej obrazy taki zestaw to zdjęcia pogrupowane w foldery względem oczekiwanych klas.

Poniższy opis opiera się na źródle ([21]).

Definicja 2.15. Skonstruujmy sieć neuronową z warstwą wejściową i wyjściową. Niech warstwa wejściowa zawiera w sobie n neuronów z losowo dobranymi wagami, losujemy również bias w warstwie wyjściowej. Zdefiniujmy m-wymiarowy ($m = n + 1$) wektor v taki, że:

$$v = (w_1, \dots, w_n, b), \text{ gdzie } w_j, b - \text{wagi neuronów i bias}$$

Założymy że mamy zestaw k danych treningowych, których etykiety tworzą wektor t^k wartości oczekiwanych. Po jednej iteracji sieci otrzymujemy wektor wartości wyjściowych y^k . Następne kroki zależą od dobranej przez nas funkcji aktywacji dla warstwy wyjściowej, rozpatrzmy dwa przypadki:

- **Funkcja aktywacji zwraca wartość liniową (tożsamościowa, ReLU)**

Stosujemy metodę **regresji liniowej**. Obliczamy błąd za pomocą dobranej funkcji błędu, zastosujemy pierwiastek błędu średniokwadratowego:

$$J(y^k, t^k) = \sum_{i=1}^k \frac{(y^i - t^i)^2}{k} \quad (\text{używamy indeksów górnych w celu rozróżnienia od wymiarów danych treningowych})$$

Na początek lekko zmodyfikujmy argumenty naszej funkcji. Obecnie zmiennymi są wartości wejściowe w warstwie, a stałymi jej wagi i bias. Jako że nasz zbiór treningowy jest stały, możemy zamienić je miejscami. Otrzymamy w ten sposób funkcję $J(v)$. Naszym celem jest zminimalizowanie wartości funkcji błędu poprzez modyfikację wektora v , czyli znalezienie jej minimum lokalnego. Takie minimum lokalne jest hiperpłaszczyzną w $m + 1$ -wymiarowej przestrzeni. Chcemy wyznaczyć dla każdego v_i kierunek prowadzący w stronę tego minimum. Możemy to zrobić w następujący sposób:

$$\vec{d}_j = \frac{\partial J}{\partial v_j} = \frac{\partial \sum_{i=1}^k \frac{(y^i - t^i)^2}{k}}{\partial v_j} = \frac{1}{k} \sum_{i=1}^k \frac{\partial (y^i - t^i)^2}{\partial v_j} = \frac{2}{k} \sum_{i=1}^k \frac{\partial y^i}{\partial v_j} (y^i - t^i)$$

(1) Dla funkcji aktywacji tożsamościowej:

$$y^i = v_m + \sum_{j=1}^{m-1} v_j \cdot x_j^i \implies \frac{\partial y^i}{\partial v_j} = x_j^i$$

w celu uproszczenia obliczeń dla biasu, zakładamy że $\forall_{i \leq k} x_m^i = 1$. Otrzymujemy:

$$\vec{d}_j = \frac{2}{k} \sum_{i=1}^k x_j^i (y^i - t^i)$$

(2) Dla ReLU z progiem p , najpierw obliczamy jej pochodną:

$$\frac{\partial f(a)}{\partial v_i} = \begin{cases} 0 & \text{dla } a < p \\ 1 & \text{dla } a \geq p \end{cases}$$

Napotykamy problem, ponieważ pochodna ta nie jest zdefiniowana w punkcie $a=p$. Dla uproszczenia założymy po prostu że jest równa tyle co dla $\geq p$, nie wpłynie to mocno na wynik. Podstawiając do naszego wzoru otrzymujemy:

$$\vec{d}_j = \begin{cases} 0 & \text{dla } b + \sum w x < p \\ \frac{2}{k} \sum_{i=1}^k x_j^i (y^i - t^i) & \text{dla } b + \sum w x \geq p \end{cases}$$

Aby zbliżyć v_j do minimum przesuwamy je w kierunku przeciwnym do d_j pomnożonym dodatkowo przez **learning rate**. Learning rate to współczynnik o wartości od 0 do 1, używany w celu ograniczenia prędkości przesuwania się wag. W późniejszej części pracy opowiem o nim więcej.

Podsumowując, w każdej iteracji sieci aktualizujemy wagę w następujący sposób:

$$w_j \rightarrow w_j - \lambda \vec{d}_j, \text{ gdzie } \lambda \text{ to learning rate.}$$

Wszystkie powyższe wyliczenia działają tak samo niezależnie od wartości biasu, dla tego dla każdego neuronu biasy aktualizujemy w ten sam sposób co wagę:

$$b \rightarrow b - \lambda \vec{d}_m$$

Proces aktualizacji wag nazywa się często **spadkiem gradientu (gradient descent)**.

- **Funkcja aktywacji zwraca prawdopodobieństwo (sigmoid logistyczny, softmax)**

Zazwyczaj używamy tych funkcji aktywacji w warstwie wyjściowej sieci, kiedy chcemy by finalnym wynikiem było prawdopodobieństwo przynależenia danych do jednej z κ predefiniowanych klas zamiast wartości liniowej. Stosujemy **regresję logistyczną**. Rozpatrzmy dwa przypadki:

- (1) Kiedy nasza sieć ma być binarnym klasyfikatorem czyli $\kappa = 2$, korzystamy z sigmoida logistycznego. Przypomnijmy wzór:

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

Potrzebna nam będzie również w następnych krokach jej pochodna:

$$\frac{\partial \sigma(a)}{\partial a} = \frac{e^{-a}}{(1 + e^{-a})^2} = \sigma(a) \frac{e^{-a}}{(1 + e^{-a})} = \sigma(a)(1 - \sigma(a))$$

Niech $a^i = b + \sum_{j=1}^n w_j x_j^i$. Skonstruujmy funkcję:

$$P(t^i|x^i, w) = \begin{cases} \sigma(a^i) & \text{dla } t^i = 1 \\ 1 - \sigma(a^i) & \text{dla } t^i = 0 \end{cases} = \sigma(a^i)^{t^i} (1 - \sigma(a^i))^{1-t^i}$$

Traktujmy ją jako prawdopodobieństwo tego, że z wektorem wag w i wartością wejściową x^i otrzymamy jako wartość wyjściową etykietę t^i . Zdarzenie wystąpienia takiego wyniku dla danej pary wartości wejściowej i etykiety jest niezależne od zdarzenia dla innej pary, stąd:

$$P\left(\bigcap_{i=1}^k t^i | x^i, w\right) = \prod_{i=1}^k P(t^i | x^i, w)$$

Otrzymaliśmy wzór na prawdopodobieństwa tego, że każda z danych wejściowych otrzyma odpowiednią etykietę. Następny krok ma na celu maksymalizację tej funkcji względem wag. W tym celu definiujemy funkcję kosztu:

$$J(v) = -\ln\left(\prod_{i=1}^k P(t^i | x^i, w)\right) = -\sum_{i=1}^k \ln(\sigma(a^i)^{t^i} (1 - \sigma(a^i))^{1-t^i})$$

Dla prostszych obliczeń nakładamy logarytm naturalny. Jest on funkcją rosnącą, więc zmaksymalizowanie go zmaksymalizuje również jego wnętrze. Jako że funkcja kosztu jest o przeciwnym znaku, to zminimalizowanie jej sprawi że zmaksymalizujemy prawdopodobieństwo. Możemy to zrobić za pomocą znanej nam już metody spadku gradientu:

$$\begin{aligned} \vec{d}_j &= \frac{\partial J(v)}{\partial v_j} = -\sum_{i=1}^k \frac{\partial \ln(\sigma(a^i)^{t^i} (1 - \sigma(a^i))^{1-t^i})}{\partial v_j} = \\ &= -\sum_{i=1}^k \frac{\partial \ln(\sigma(a^i)^{t^i})}{\partial v_j} + \frac{\partial \ln(1 - \sigma(a^i))^{1-t^i}}{\partial v_j} \end{aligned}$$

Korzystając z chain rule:

$$\begin{aligned} \frac{\partial \ln(\sigma(a^i)^{t^i})}{\partial v_j} &= t^i \frac{\partial \ln(\sigma(a^i))}{\partial \sigma(a^i)} \frac{\partial \sigma(a^i)}{\partial a^i} \frac{\partial a^i}{\partial v_j} = \\ &= t^i \frac{1}{\sigma(a^i)} \sigma(a^i)(1 - \sigma(a^i)) x_j^i = t^i(1 - \sigma(a^i)) x_j^i \\ \frac{\partial \ln(1 - \sigma(a^i)^{t^i})}{\partial v_j} &= t^i \frac{\partial \ln(1 - \sigma(a^i))}{\partial (1 - \sigma(a^i))} \frac{\partial (1 - \sigma(a^i))}{\partial a^i} \frac{\partial a^i}{\partial v_j} = \\ &= t^i \frac{1}{1 - \sigma(a^i)} (-\sigma(a^i))(1 - \sigma(a^i)) x_j^i = -t^i \sigma(a^i) x_j^i \end{aligned}$$

Tak samo jak przy regresji logistycznej, zakładamy że $\forall_{i \leq k} x_m^i = 1$

$$\vec{d}_j = - \sum_{i=1}^k t^i (1 - \sigma(a^i)) x_j^i - t^i \sigma(a^i) x_j^i = - \sum_{i=1}^k t^i x_j^i (1 - 2\sigma(a^i))$$

Ponownie, w każdej iteracji sieci modyfikujemy wagi i bias, pamiętając o learning rate:

$$v_j \rightarrow v_j - \lambda \vec{d}_j$$

- W przypadku $\kappa > 2$, musimy zmienić architekturę naszej sieci. Zmieniamy funkcję aktywacji na funkcję softmax, a co za tym idzie zmieniamy liczbę neuronów wyjściowych na κ .

$$s(x_n) = \left(\frac{e^{a_1}}{\sum_{\alpha \leq \kappa} e^{a_\alpha}}, \dots, \frac{e^{a_\kappa}}{\sum_{\alpha \leq \kappa} e^{a_\alpha}} \right)$$

x_n - n wartości neuronów z warstwy wejściowej

$$a_\alpha = b_\alpha + \sum_{j=1}^n w_j x_j$$

Obliczmy pochodną jej składowej:

$$\begin{aligned} \frac{\partial s_\alpha(x_i)}{\partial x_i} &= \frac{(e^{x_i} \sum e^{x_j}) - (e^{x_i} \cdot e^{x_i})}{(\sum e^{x_j})^2} = \frac{e^{x_i}(-e^{x_i} + \sum e^{x_j})}{(\sum e^{x_j})^2} = \\ &= s_\alpha(x_i) \frac{-e^{x_i} + \sum e^{x_j}}{\sum e^{x_j}} = s_\alpha(x_i)(1 - s_\alpha(x_i)) \end{aligned}$$

Wartością wyjściową naszej funkcji są wektory, każda składowa takiego wektora α to prawdopodobieństwo przynależności wartości wejściowej x^i do klasy α . Pierwiastek błędu kwadratowego nie nadaje się tutaj do roli funkcji błędu, zamiast tego skorzystamy ze średniej entropii krzyżowej (average cross-entropy). Danej wzorem:

$$J = \frac{1}{k} \sum_{i=1}^k \left(- \sum_{\alpha \leq \kappa} t_\alpha^i \ln(y_\alpha^i) \right)$$

gdzie y_α^i, t_α^i - kolejno rzeczywiste ($y_\alpha^i = s_\alpha(x^i)$) i oczekiwane prawdopodobieństwo przynależenia wartości wejściowej i do klasy α . Pozostało obliczyć spadek gradientu dla funkcji straty:

$$\frac{\partial t_\alpha^i \ln(y_\alpha^i)}{\partial v_j} = \frac{\partial t_\alpha^i \ln(y_\alpha^i)}{\partial y_\alpha^i} \frac{\partial y_\alpha^i}{\partial a_\alpha^i} \frac{\partial a_\alpha^i}{\partial v_j} = \frac{1}{y_\alpha^i} y_\alpha^i (1 - y_\alpha^i) x_j^i = y_\alpha^i (1 - y_\alpha^i) x_j^i$$

$$\vec{d}_j = \frac{\partial J}{\partial v_j} = \frac{1}{k} \sum_{i=1}^k \left(- \sum_{\alpha \leq \kappa} t_\alpha^i y_\alpha^i (1 - y_\alpha^i) x_j^i \right)$$

Co znowu daje nam kierunek.

$$v_j \rightarrow v_j - \lambda \vec{d}_j$$

W obu przypadkach powtarzamy spadek gradientu, aż do momentu uzyskania jak najniższej wartości funkcji błędu. Za akceptowalną przyjmuje się zazwyczaj wartość około 0.05. Dla sieci neuronowej z 1 warstwą te 2 algorytmy nam w zupełności wystarczą do wytrenowania sieci. Jednak w przypadku sieci wielowarstwowej napotykamy pewien problem. W każdym z poprzednich przykładów tworzyliśmy funkcję $J(v)$ i stosowaliśmy na niej spadek gradientu. W każdym przykładzie funkcja ta wymagała od nas znajomości danych wejściowych, wag, biasu, funkcji aktywacji danej warstwy, wartości wyjściowych oraz oczekiwanych wartości wyjściowych. Tak jak w ostatniej warstwie sieci dane o oczekiwanych wartościach mamy z etykiet, tak w warstwach wewnętrznych nie jesteśmy w stanie ich określić. Dlatego musimy zastosować dodatkowo **algorytm propagacji wstecznej**.

Przyjmijmy n-warstwową sieć ze znanyymi nam jej wszelkimi parametrami i zetykietowanymi danymi treningowymi. Wyliczmy \vec{d} dla ostatniej warstwy korzystając z regresji liniowej lub logistycznej, mamy do tego wszystkie potrzebne dane. Otrzymamy końcowo:

$$\vec{d}_n = \frac{\partial J}{\partial v_n} = \frac{\partial J}{\partial y_n} \frac{\partial y_n}{\partial a_n} \frac{\partial a_n}{\partial v_n} = \frac{\partial J}{\partial y_n} \frac{\partial y_n}{\partial a_n} y_{n-1}$$

Zauważmy, że wzór ten jest prawdziwy dla każdej warstwy $1 \leq i \leq n$ oraz jedyna jego część, która wymaga od nas znajomości oczekiwanych danych wyjściowych w warstwie to $\frac{\partial J}{\partial y_i}$, resztę możemy wyliczyć z dostępnych danych. W celu wyliczenia brakującego składnika korzystamy z twierdzenia o pochodnych cząstkowych funkcji złożonej ([22]):

$$\frac{\partial J}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial y_i}$$

Z pomocą powyższego wzoru możemy wyliczyć potrzebną wartość dla każdej warstwy poczynając od warstwy n-1 i poruszając się w tył. Stąd nazwa "propagacji wstecznej".

2.5. Konwolucyjne sieci neuronowe.

Konwolucyjne sieci neuronowe to typ sieci stosowany, kiedy danymi wejściowymi są obrazy. Wyróżniamy w nich 3 typy warstw:

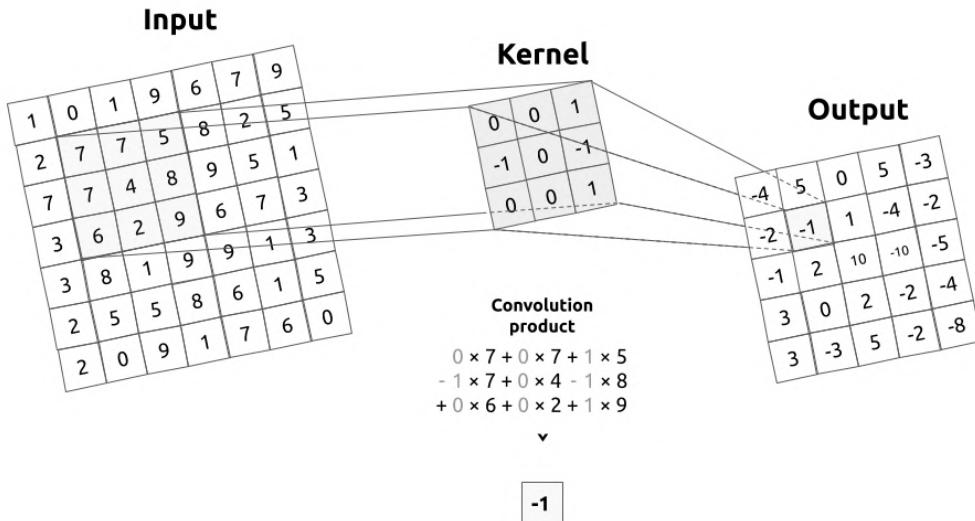
- **Warstwy konwolucyjne ([23])**

Każda taka warstwa ma przyporządkowaną maskę (ozn. ω), czyli tablicę o wymiarach $s_1 \times s_2$ (zazwyczaj $s_1=s_2$) wypełnioną wartościami pikseli oraz bias (ozn. b). Maska ta jest wykorzystywana do operacji splotu wykonywanej na danych wejściowych:

$$G(x, y) = b + w * F(x, y) = b + \sum_{a=0}^{s_1} \sum_{b=0}^{s_2} w(a, b) \cdot F(x + a, y + b)$$

gdzie $G(x,y)$, $F(x,y)$ - wartości pikseli o współrzędnych x,y

Operacja ta jest wykonywana domyślnie na (prawie) każdym pikselu w obrazie i z wartości G tworzony jest nowy obraz o wymiarach zależnych od wymiarów maski i pewnych parametrów warstwy.



RYSUNEK 4. Działanie maski 3×3 na obrazie o rozdzielcości 7×7 , wynikiem jest obraz 5×5 ([23])

Maskę "nakładamy" na piksele obrazu zaczynając od piksela $(0,0)$. Nie "mieści" się ona jednak na każdym z pikseli, nałożenie jej na piksel oddalony o mniej niż 3 piksele (horizontalnie lub wertykalnie, nie na ukos) od prawej lub dolnej krawędzi obrazu poskutkowałoby niewystarczającą liczbą pikseli do operacji splotu. Dlatego piksele znajdujące się tam są ignorowane, co skutkuje innymi wymiarami obrazu wyjściowego.

Dla maski określamy parametry:

- strides - domyślnie poruszamy naszą maską o 1 piksel w prawo i 1 piksel w dół, możemy jednak zmodyfikować te wartości wedle naszego uznania i poruszać np. 2 w prawo i 3 w dół, wtedy ustawiamy strides na (2,3)
- padding - za pomocą paddingu możemy dodać do obrazu po bokach lub z góry i z dołu dodatkowe piksele o wartościach 0, z paddingiem (1,2) dodamy po lewej i prawej stronie po jednej warstwie pikseli, a z góry i z dołu dwie. Użyteczny gdy chcemy zachować wymiary obrazu.
- dilation - odstęp pomiędzy pikselami w masce, domyślnie są ustawione obok siebie w zwartej tablicy, ale mogą być oddzielone. Jeśli ustawimy dilation na (2,2) w masce 3x3 to pierwszy piksel maski pokryje piksel (x,y) obrazu, drugi pokryje piksel (x+2, y), trzeci (x+4, y), czwarty (x, y+2) itd.

Biorąc pod uwagę wszystkie parametry, możemy podać wzór na wysokość obrazu wyjściowego:

$$H_{out} = \frac{H_{in} + 2P_H - D_H(K_H - 1) - 1}{S_H}$$

P, D, K, S - to kolejno padding, dilation, wymiar maski, strides. Szerokość obliczamy analogicznie.

Warstwy konwolucyjne służą do wyodrębnienia pożądanych cech z obrazu, możemy wytrenować te warstwy traktując wartości pikseli w masce i bias tak jak byśmy traktowali wagi i bias w warstwach ukrytych.

• Warstwy grupujące ([24])

Warstwy te działają na tej samej zasadzie co warstwy konwolucyjne, poruszają się po wszystkich pikselach od lewego górnego rogu. Do warstwy przyporządkowuje się maskę o wymiarach f x f (zawsze są sobie równe), operacja którą maska wykonuje zależy od typu warstwy:

- maska max pooling - ze wszystkich pikseli obrazu na które została nałożona wybieramy ten o największej wartości.
- maska average pooling - zamiast największego wyliczamy średnią wartość pikseli w granicach maski.

Dla maski określamy parametr stride (S), który działa identycznie jak w warstwach konwolucyjnych, z tą różnicą, że dla obu wymiarów musi być taki sam. Wzór na wysokość obrazu wyjściowego:

$$H_{out} = \frac{H_{in} - f + 1}{S}$$

Dla szerokości analogicznie. Warstwy te są wykorzystywane w celu ograniczenia liczby pikseli i zaoszczędzenia na mocy obliczeniowej, a także w celu zgeneralizowania wartości pikseli po warstwie konwolucyjnej (z tego powodu te dwie warstwy są zazwyczaj ustawiane w sieci obok siebie), co pomaga zapobiec zjawisku nadmiernego dopasowania, które zostanie lepiej opisane w następnych podrozdziałach.

- **Warstwy w pełni połączone**

Są to warstwy identyczne do warstw jakie definiowaliśmy w poprzednim podrozdziale. Jako wartość wejściową przyjmują spłaszczone do wektora wartości pikseli obrazu.

2.6. Problemy rozwiązywane przez sieci neuronowe.

W zależności od funkcji aktywacji używanej w ostatniej warstwie, sieć neuronowa może służyć do rozwiązywania problemów różnego rodzaju.

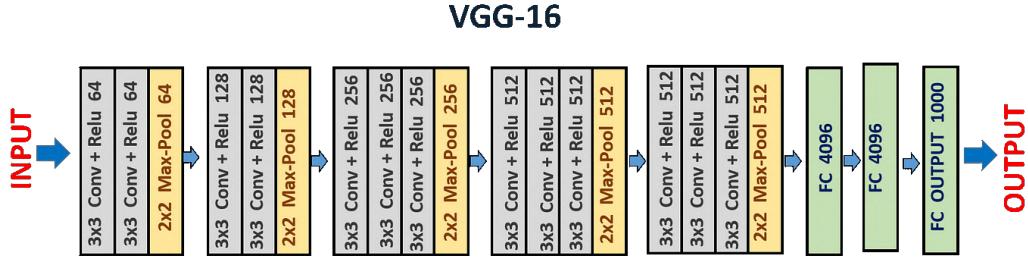
Definicja 2.16. Siecią regresyjną nazywamy sieć której danymi wyjściowymi są liczby rzeczywiste. Może być używana np. do przewidywania cen akcji na giełdzie.

Definicja 2.17. Sieć klasyfikująca jako wynik zwraca wektor prawdopodobieństw, tak zwanych pewności. Informują one na ile sieć jest pewna, że dane wejściowe przynależą do każdej z klas. Wybierając największą z klas dokonujemy klasyfikacji obiektu. Sieć może być wykorzystywane na przykład w celu detekcji spamu na skrzynce mailowej.

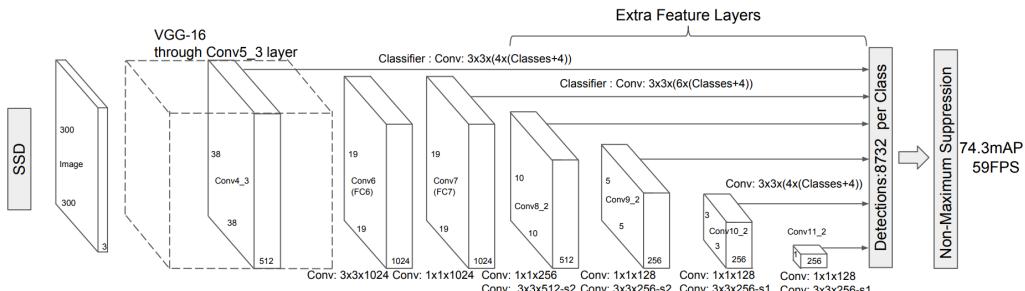
Definicja 2.18. Bounding box to prostokątny kształt, w którym zawiera się obiekt wykryty przez sieć neuronową do detekcji obiektów. Zazwyczaj zapisuje się go jako współrzędne dwóch przeciwnie wierzchołków, bo jest to minimalna wymagana ilością informacji do jednoznacznego wyznaczenia prostokąta na płaszczyźnie. Powinien on obejmować wykryty obiekt tak zwarcie jak to tylko możliwe, czyli żaden wierzchołek nie powinien wykraczać poza jego granice.

Definicja 2.19. Sieć do detekcji obiektów, czyli sieć którą zastosuję w aplikacji. Najprostszy przykład takiej sieci to podzielenie obrazu na segmenty i przepuszczenie każdego z tych segmentów przez sieć klasyfikującą. Następnie segmenty, w których został wykryty obiekt są dzielone na coraz to mniejsze, aż do momentu uzyskania bounding boxów. Taki algorytm jest niesamowicie kosztowny obliczeniowo, dlatego powstały rozwiązania optymalizujące. Większość z nich działa na podstawie wyznaczania kandydatów w których może się znajdować obiekt żeby nie trzeba było sprawdzać każdego regionu klasyfikatorem. Jednym z takich rozwiązań jest architektura SSD: Single Shot MultiBox Detector ([25]), z której będę korzystał w projekcie.

Wyróżnia się ona tym, że w początkowych fazach działania umieszcza na obrazie kandydatów, a później przewiduje wartości ich przesunięcia w celu uzyskania prawidłowych bounding boxów. Architektura zaczyna się od klasyfikatora wysokiej jakości, w założeniu VGG-16 który składa się z kilku par warstw konwolucyjnych i grupujących zakończony trzema warstwami w pełni połączonymi. Dodawane są do niego warstwy przewidujące przesunięcia kandydatów.



RYSUNEK 5. Bazowy klasyfikator VGG-16([26])



RYSUNEK 6. Single Shot MultiBox Detector ([25])

Największą zaletą SSD jest jej wydajność. Architektura jest idealnym wyborem dla aplikacji mobilnych, tym bardziej gdy ma wykonywać detekcję w czasie rzeczywistym.

2.7. Konstruowanie sieci neuronowych w praktyce.

Definicja 2.20. Hiperparametrami sieci neuronowej nazywamy parametry, które wpływają na działanie sieci, ale nie są trenowalne. To znaczy nie zmieniamy ich wartości poprzez spadek gradientu.

Podsumowując cały temat sieci neuronowych, są to algorytmy które przy odpowiednio dobranych danych treningowych i hiperparametrach mogą być wytrenowane do naśladowania działania innego algorytmu. Warto w tym miejscu wspomnieć o Uniwersalnym Twierdzeniu o Aproksymacji ([27]), które mówi że z pomocą sieci neuronowej z co najmniej jedną warstwą ukrytą, jesteśmy w stanie zaproksymować dowolną funkcję ciągłą. W praktyce jednak skonstruowanie takiej sieci nie jest łatwym zadaniem. Przy trenowaniu sieci możemy napotkać szereg różnych problemów, opiszę kilka z nich:

- **Przetrenowanie sieci (overfit).** Zjawisko to zachodzi jeśli sieć nauczy się podanych danych na pamięć. Czyli wyciągnie z nich parametry które nie mają znaczenia przy wyznaczonym zadaniu. Na przykład w przypadku trenowania sieci klasyfikującej obrazy, sieć może zacząć rozpoznawać obiekty na podstawie tła na jakim się znajdują. W celu zapobiegnięcia przetrenowania należy dobierać zróżnicowane dane treningowe (ważne jest aby dane w tej samej klasie przedstawiały to samo, ale nie były identyczne) oraz w miarę możliwości usunąć z nich szum informacyjny, czyli bezużyteczne dla wyznaczonego zadania dane. Nie należy również trenować sieci zbyt długo na tych samych danych.
- **Niedotrenowanie sieci (underfit).** Mamy do czynienia z niedotrenowaniem w przypadku gdy sieć nie wyciągnęła wystarczającej ilości cech z podanych danych. Sieć taka zbyt generalnie podchodzi do wyznaczonego zadania, w przypadku sieci klasyfikującej wykrywa ona obiekty tam, gdzie nie powinna. W celu rozwiązania problemu możemy zwiększyć ilość danych treningowych lub zwiększyć ilość trenowalnych parametrów sieci poprzez dodanie warstwy lub zwiększenie liczby neuronów w warstwach ukrytych.
- **Przeskoczenie minimum lokalnego funkcji błędu.** Przy opisie algorytmu propagacji wstecznej pojawiło się pojęcia parametra learning rate, jest to parametr służący do kontrolowania szybkością z jaką modyfikujemy wagę w algorytmie spadku gradientu. W przypadku gdy parametr learning rate ustawimy zbyt wysoki to pojawi się ryzyko przeskoczenia przez wagę minimum lokalnego funkcji błędu co sprawi, że w kolejnych iteracjach funkcja błędu będzie rosnąć.
- **Utknięcie funkcji błędu poza minimum lokalnym.** Przeciwieństwo poprzedniego problemu w przypadku gdy dobierzemy zbyt niski learning rate. Funkcja może się zatrzymać w nieporządanym prez nas minimum lokalnym i nie być w stanie wyjść poza nie przez zbyt niską wartość gradientu. Dobranie bardzo niskiej wartości skutkuje również zwiększeniem czasu potrzebnego na wytrenowanie sieci, a nie zawsze przynosi benefity.

Sama diagnoza występowania jednego z wyżej wymienionych problemów nie jest kwestią trywialną. Nie istnieje coś takiego jak uniwersalny przepis na wytrenowanie dowolnej sieci. Poza kilkoma wskazówkami jak te wymienione wyżej, wytrenowanie sieci od zera wymaga od nas stosowania metody prób i błędów. Korzystamy z dwóch typów metryk do określenia jak dobrze sprawuje się nasz model:

- Wyznaczane podczas treningu. Biblioteki takie jak tensorflow lub pytorch będą nam zawsze wyświetlać aktualną wartość funkcji błędów. Nie jest to jednak wartość funkcji dla danych treningowych, a dla danych ze zbioru testowego.

Definicja 2.21. Przed rozpoczęciem trenowania sieci dzielimy nasz zbiór poetykietowanych danych na zbiór treningowy i zbiór testowy. **Zbiór treningowy** to dane, które są bezpośrednio przepuszczane przez sieć i na ich podstawie liczymy spadek gradientu. **Zbiór testowy** służy do obliczania funkcji błędu, z pomocą której kontrolujemy postęp uczenia. Dostarcza ona nam informację kiedy zakończyć trening, a także pomaga wykryć ewentualne anomalie, jak na przykład nagły wzrost wartości funkcji błędu (dzieje się tak często przy zbyt dużym learning rate).

- Wyznaczane po zakończeniu trenowania modelu. W celu określenia jakości sieci przepuszczamy przez sieć dane ze zbioru ewaluacyjnego.

Definicja 2.22. Zbiorem ewaluacyjnym nazywamy zbiór danych nieużywanych przy trenowaniu sieci. Jego celem jest zbadanie jakości jej treningu.

I wyznaczamy na podstawie danych wyjściowych **tablicę pomyłek**(confusion matrix).

$$M_{n \times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Gdzie a_{ij} oznacza ilość przypadków przyporządkowania wartości wejściowej z realną etykietą i do klasy j. Z jej pomocą możemy podzielić rezultaty na 3 kategorie, osobno dla każdej klasy:

- (1) True Positives(TP)- przypadki, w których element należy do klasy i oraz sieć przyporządkowała go do klasy i
- (2) False Postives (FP)- nie należy do i, przyporządkowany do i
- (3) False Negatives (FN)- należy do i, nie przyporządkowany do i

Następnie dla każdej z klas wyznaczamy 2 metryki:

- precyzyja (precision) - mierzy z jaką dokładnością model omija False Positives.

$$P_i = \frac{|TP|_i}{|TP|_i + |FP|_i}$$

- czułość (recall) - z jaką dokładnością omija False Negatives.

$$R_X = \frac{|TP|_i}{|TP|_i + |FN|_i}$$

W celu oceny wszystkich klas modelu, jako metrykę możemy przyjąć średnią arytmetyczną danej metryki dla wszystkich klas. Do ogólnej oceny działania całego modelu, bez podziału na klasy używa się również metryki dokładności (accuracy):

$$A = \frac{|TP|}{|TP| + |FP| + |FN|}$$

Zazwyczaj wystarczająca jest metryka dokładności, ale czasami zależy nam bardziej na uniknięciu False Positivów niż False Negativów i vice versa. Na przykład w przypadku sieci rozpoznającej osobę na widoku z kamery w zamkniętym sklepie, w celu wykrycia włamania. Mniejsze konsekwencje spotkają nas z powodu wykrycia osoby tam gdzie jej nie było, niż z powodu nie wykrycia osoby tam gdzie była.

Szeroko stosowaną regułą jest dzielnie naszych danych na treningowe, testowe i ewaluacyjne w proporcjach 70-15-15.

3. PROJEKT DYPLOMOWY

Zacznijmy od definicji kilku prostych pojęć używanych przy opisie działania sieci.

Definicja 3.1. Interferencją nazywamy pojedyncze wykonanie przez sieć wyznaczonego zadania, bez aktualizowania wag. W przypadku sieci do detekcji obiektów jest to wykrycie na podanym zdjęciu klas obiektów i wyznaczenie ich bounding boxów.

Definicja 3.2. Przy definicji procesu uczenia w poprzednim rozdziale założyłem że za każdym razem przepuszczamy przez sieć wszystkie dostępne dane treningowe. W rzeczywistości zazwyczaj nie jesteśmy w stanie tego zrobić, ponieważ pamięć w komputerze jest ograniczona. Aby temu zaradzić definiujemy hiperparametr **batch size**, który oznacza liczbę danych która przechodzi przez sieć w każdej iteracji. W naszym przypadku oznacza ilość zdjęć.

Definicja 3.3. Epoką nazywamy jedno pełne przejście danych treningowych z *batch_size* przez sieć i wykonanie na nich propagacji wstecznej.

Przed rozpoczęciem pisania jakiekolwiek aplikacji należy się zastanowić, czy jej założenia są możliwe do zrealizowania. Tak jak nie przewiduję dużych problemów przy trenowaniu sieci do detekcji obiektów, tak część która ma badać ich wartości odżywcze z pewnością nie będzie działać idealnie. W celu wyliczenia tych wartości musimy znać wagę obiektu, którą możemy otrzymać znając jego objętość i gęstość. Gęstość nie jest problemem, do każdego obiektu można ją z góry przypisać, wyzwaniem jest zmierzenie objętości. Wracając do początkowych założeń pracy, aplikacja ma być wygodna w obsłudze na tyle na ile to możliwe, a na pewno wygodniejsza od klasycznego wybierania produktów z listy. Nie możemy oczekiwać od użytkownika podania większej ilości informacji niż jedno zdjęcie, co daje nam jedynie dwuwymiarowy rzut badanego obiektu. Musimy więc przyjąć, że każdy obiekt posiada regularny kształt, czyli taki którego objętość możemy wyliczyć ze wzoru znając jedynie kilka jego parametrów. Problemów tego typu z pewnością napotkamy dużo więcej i każdy z nich będzie skutkował spadkiem dokładności. Dlatego zaczniemy od sprawdzenia z jakim błędem będziemy się mierzyć.

3.1. Badanie błędu pomiaru.

Aby zbadać jakikolwiek wymiar obiektu ze zdjęcia, potrzebujemy dodatkowych informacji o samym zdjęciu. Musimy znać dane techniczne aparatu, odległość badanego obiektu od zdjęcia oraz kąt nachylenia kamery. Prawie każdą z tych danych jesteśmy w stanie uzyskać na każdym smartfonie bez ingerencji użytkownika. Wyjątkiem jest odległość aparatu od obiektu od obiektu. O ile smartfon nie jest wyposażony w dalmierz laserowy to użytkownik musiałby za każdym razem sam tę odległość mierzyć, co byłoby albo bardzo niewygodne albo bardzo nieprecyzyjne. Dlatego lepszym rozwiązaniem jest wymóg ustalenia na zdjęciu obiektu o znanych rozmiarach, aby służył jako punkt odniesienia, najlepiej jakby był to przedmiot który każdy ma zawsze przy sobie. Idealnym kandydatem jest moneta. Każda moneta jest bita na ten sam wymiar, który z biegiem czasu nie zmienia się w stopniu znacznym oraz

posiada niezmienny, regularny kształt koła. Zrobiłem więc kilka zdjęć jabłka z monetą obok oraz zmierzyłem jego rzeczywiste wymiary przy pomocy suwmiarki oraz wagę za pomocą wagi kuchennej. Na tym etapie musimy przyjąć uproszczenie co do kształtu jabłek. Uznałem że najbliższej jest im do elipsoidy.

Średnica pionowa	5.7 cm
Średnica pozioma 1	7.4 cm
Średnica pozioma 2	7.4 cm
Waga	149 g
Gęstość jabłka	0.75 g/cm ³

Obliczamy objętość elipsoidy ze wzoru:

$$v = \frac{4}{3}\pi \cdot \left(\frac{7.4}{2}\right)^2 \cdot \frac{5.7}{2} \approx 163$$

Następnie korzystając z gęstości możemy wyliczyć wagę:

$$m \approx 163 * 0.75 = 122$$

Uproszczenie co do kształtu poskutkowało relatywnie niewielkim błędem. Prawdopodobnie będzie on podobny dla większości innych owoców i warzyw, problem może się pojawić dla produktów o nieregularnym kształcie, na przykład cukru lub mąki. W tym przypadku nie jesteśmy w stanie zmierzyć wagi z jednego zdjęcia, dlatego aplikacja będzie je jedynie identyfikować bez liczenia kalorii. Kolejnym etapem jest wyznaczenie ile średnicy monety zmieści się w średnicach jabłka. W tym celu napisałem program, którego kod umieszczam poniżej.

```

1 import cv2
2 import os
3 import math
4
5
6 def read_img(name):
7     path = os.path.join(os.getcwd(), f"apple_measure/{name}")
8     return cv2.imread(path)
9
10
11 img = read_img("bboxes/7.jpg")
12
13 # original photos were taken with a phone and they are in 1200/3000
# resolution
14 # which is unreadable on desktop, that's why we use the downsize variable
15 downsize = 1200/3000
16 thickness = 2
17
18 img = cv2.resize(img, (int(img.shape[1]*downsize), int(img.shape[0]*
downsize)))
19
20
21
22
23 def draw_vertical_line(start_point, length=1, size=200):
24     color = (0, 255, 0)
25     start_point = (int(start_point[0]*downsize), int(start_point[1]*
downsize))
26     end_point = (start_point[0] + int(size*downsize), start_point[1])
27     cv2.line(img, start_point, end_point, color, thickness)
28     cv2.circle(img, end_point, 3, (0, 0, 0), -1)
29     count = math.floor(length) - 1
30     for i in range(count):
31         start_point = end_point
32         end_point = (start_point[0] + int(size*downsize), start_point[1])
33         cv2.line(img, start_point, end_point, color, thickness)
34         cv2.circle(img, end_point, 3, (0, 0, 0), -1)
35     last_part = length - (count + 1)
36     start_point = end_point
37     end_point = (start_point[0] + int(last_part*size*downsize),
start_point[1])
38     cv2.line(img, start_point, end_point, color, thickness)
39
40

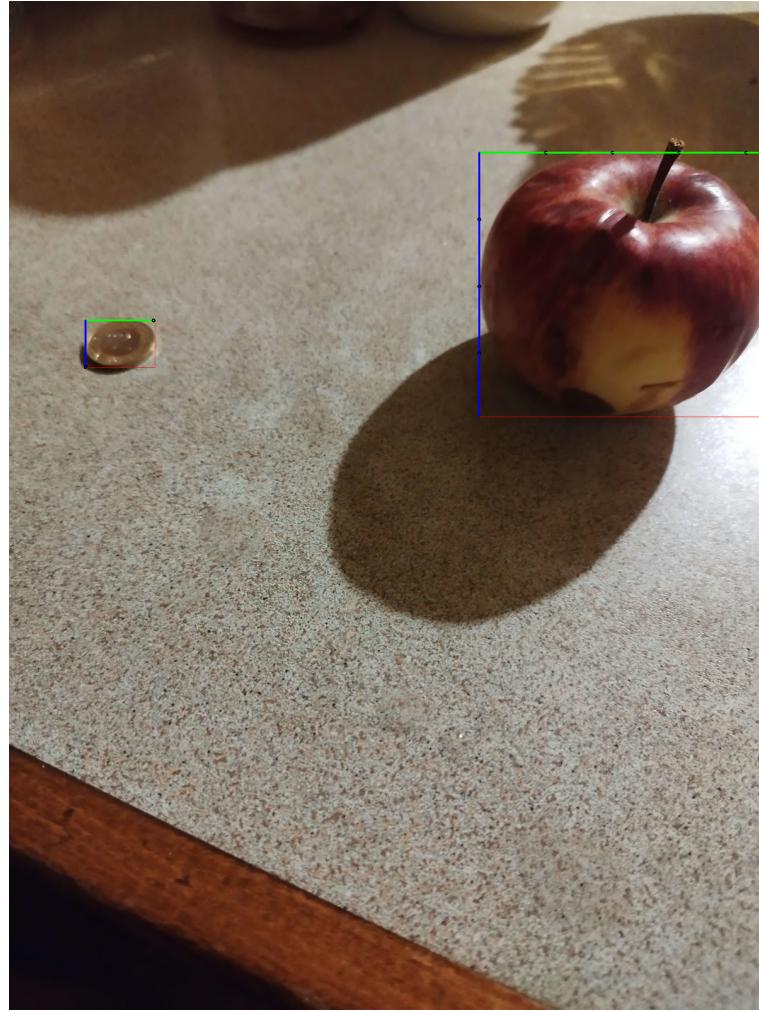
```

```

41 def draw_horizontal_line(start_point, length=1, size=200):
42     color = (255, 0, 0)
43     start_point = (int(start_point[0]*downsize), int(start_point[1]*
44     downsize))
45     end_point = (start_point[0], start_point[1] + int(size*downsize))
46     cv2.line(img, start_point, end_point, color, thickness)
47     cv2.circle(img, end_point, 3, (0, 0, 0), -1)
48     count = math.floor(length) - 1
49     for i in range(count):
50         start_point = end_point
51         end_point = (start_point[0], start_point[1] + int(size*downsize))
52         cv2.line(img, start_point, end_point, color, thickness)
53         cv2.circle(img, end_point, 3, (0, 0, 0), -1)
54     last_part = length - (count + 1)
55     start_point = end_point
56     end_point = (start_point[0], start_point[1] + int(last_part*size*
57     downsize))
58     cv2.line(img, start_point, end_point, color, thickness)
59
60 start_point = (304, 1265)
61 draw_vertical_line(start_point, size=270)
62 draw_horizontal_line(start_point, size=185)
63
64 start_point = (1867, 600)
65 l1 = 4.18
66 l2 = 5.65
67 draw_vertical_line(start_point, size=270, length=l1)
68 draw_horizontal_line(start_point, size=185, length=l2)
69
70 cv2.imshow("Measurement", img)
71 cv2.waitKey(0)
72 cv2.imwrite("measurement.png", img)

```

LISTING 1. Kod do ręcznego wyznaczania bounding boxów



RYSUNEK 7. Efekt działania programu

Proces wyznaczania ilości monet w jabłku polegał na wizualnym dopasowaniu punktów początkowych dla monety i jabłka oraz argumentów size i length w linijkach 59-66 za pomocą metody prób i błędów. W ten sposób wyznaczyliśmy ręcznie dla obiektów **bounding boxes**, tak jak to będzie robić później sieć neuronowa. Dla danego zdjęcia otrzymaliśmy wartości length 4.18 w średnicy poziomej i 5.65 w średnicy pionowej. Korzystając z danych o średnicy monety (21.5 mm) wyznaczamy objętość:

$$4.18 \cdot 2.15 = 8.9869$$

$$v = \frac{4}{3}\pi \cdot \left(\frac{8.9869}{2}\right)^2 \cdot \frac{12.1475}{2} \approx 513.70$$

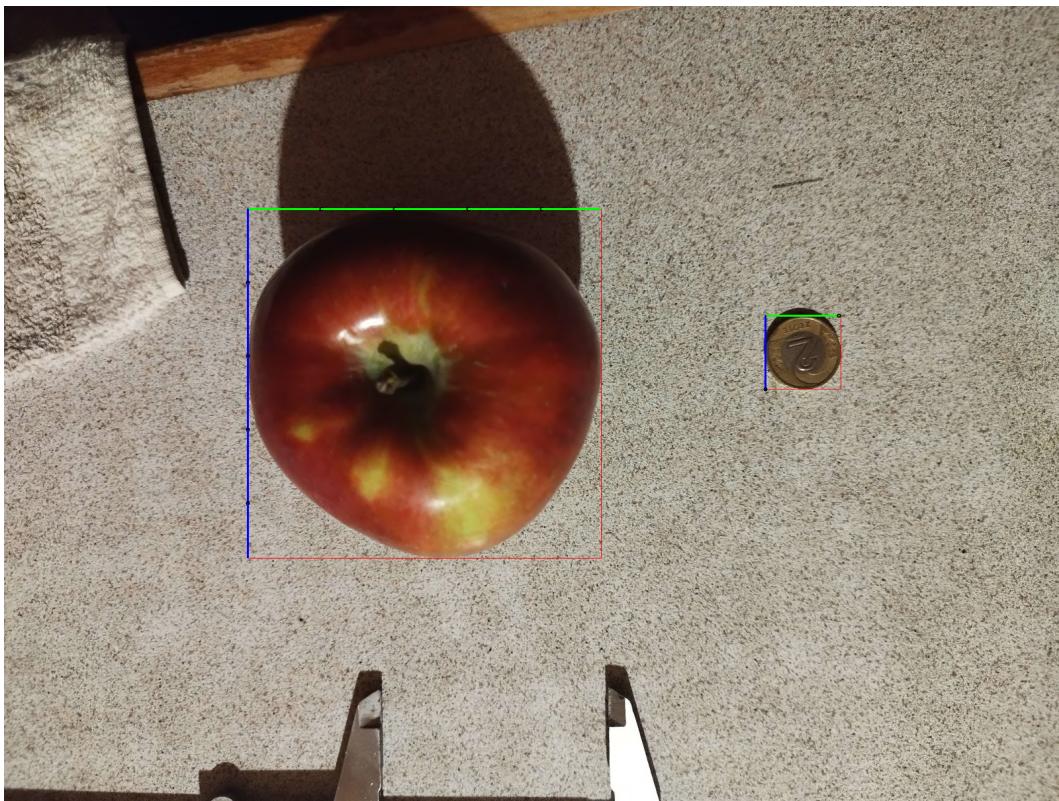
Następnie korzystając z gęstości możemy wyliczyć wagę:

$$m \approx 513.70 * 0.75 \approx 385.27$$

Niestety, wyniki które uzyskaliśmy nie mają żadnego pokrycia z rzeczywistością - masa jabłka wyszła ponad trzykrotnie większa niż powinna. Głównym powodem takiego wyniku

jest perspektywa. Jeśli zdjęcie nie jest zrobione z góry, to moneta nie ma kształtu idealnego koła tylko elipsy, a mierzony obiekt jest ustawiony pod kątem co daje spore zniekształcenie z perspektywy algorytmu. Rozwiązaniem jest dodanie wymogu dla użytkownika w postaci robienia zdjęcia z perspektywy ortograficznej. Rodzi to jednak problem, ponieważ nie zmierzmy wtedy wysokości jabłka. Dlatego ostatnim uproszczeniem które musimy przyjąć jest to, że nasze obiekty mają stałe proporcje, abyśmy mogli wyznaczyć te niewidoczne na zdjęciu korzystających z tych widocznych. W przypadku jabłka możemy je wziąć z poprzedniego przykładu, $\frac{5.7}{7.4}$ średnicy poziomej wobec pionowej. Sprawdzimy czy te proporcje mają odzwierciedlenie w rzeczywistości na przykładzie zdjęcia i pomiarów innego jabłka.

Średnica pionowa	6.2 cm
Średnica pozioma 1	8.2 cm
Średnica pozioma 2	8.4 cm
Waga	216 g



Z programu wyliczyłem że średnice poziome to kolejno 5.04 i 4.94 średnicy monety. Podstawiając do proporcji, konwertując na centymetry i wyliczając masę otrzymujemy:

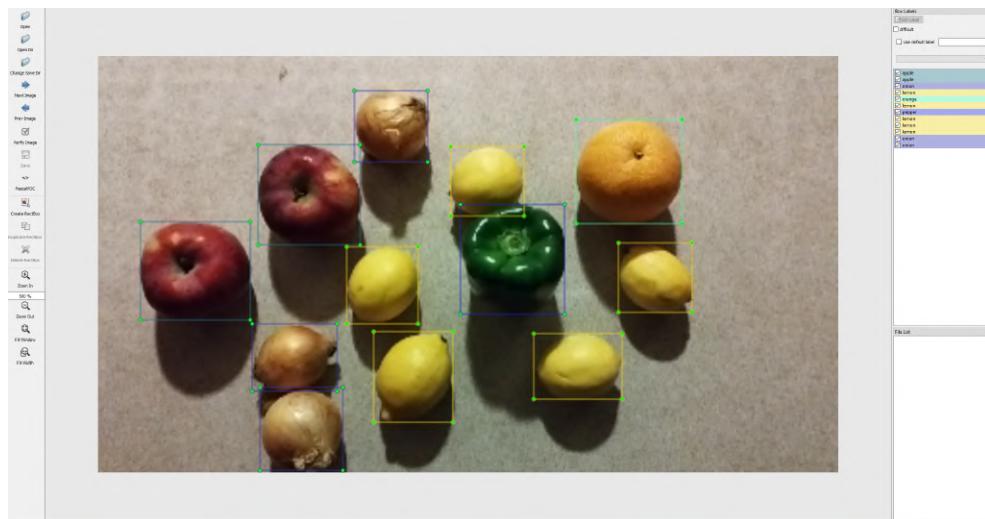
Średnica pionowa	8.181 cm
Średnica pozioma 1	10.621 cm
Średnica pozioma 2	10.836 cm
Waga	369 g

Wynik nie jest idealny, ale jest obarczony dużo mniejszym błędem niż poprzedni. Możliwe że wynika on z dosyć nieregularnego kształtu fotografowanego jabłka względem typowego. Widać również, że przyjęte proporcje średnic w przypadku tego jabłka pasują. Podsumowując, nasza metoda nie jest zbyt precyzyjna. Ale należy zadać pytanie: z jakim błędem użytkownik mierzy się dodając produkty tradycyjną metodą w aplikacjach dietetycznych, czyli z listy? O ile sam je wszystkie waży, to z niewielkim, ale jest to mało praktyczne i zajmuje czas. Sporo osób zamiast z wagi korzysta z predefiniowanych "porcji". Wpisuję w aplikacji, że zjedli na przykład jedno jabłko i aplikacja dodaje do bilansu jabłko o średniej wadze, nieważne jakich rozmiarów było jabłko użytkownika. W dodatku, użytkownik raczej nie wpisuje dokładnej odmiany jabłka a to również wpływa na jego rzeczywistą kaloryczność, sieć neuronowa nie miałaby problemu z rozpoznaniem odmiany. Funkcja mierzenia kalorii nie musi być również narzucona, można dać użytkownikowi możliwość wyboru pomiędzy automatycznym mierzeniem wagi, wybieraniem średniej lub ręcznym wpisywaniem, więc aplikacja dalej ma sens.

3.2. Trenowanie sieci neuronowej do detekcji obiektów.

Danymi wejściowymi będzie obraz, a wyjściowymi etykiety wykrytych obiektów i ich bounding boxy. W poniższych punktach opiszę proces wytrenowania sieci krok po kroku.

- (1) Pierwszym krokiem było zgromadzenie odpowiednich danych treningowych. W tym miejscu należy zaznaczyć, że aplikacja jest bardziej przedstawieniem konceptu, niż skończonym produktem. Im więcej zdefiniujemy klas, tym więcej dana klasa będzie potrzebować danych treningowych, a im więcej danych treningowych tym więcej potrzebnej mocy obliczeniowej do wytrenowania sieci. Dlatego zdefiniowałem jedynie 5 klas: jabłko, paprykę, pomarańcz, cebulę i cytrynę. Zebrałem 250 zdjęć i naniosłem na nie etykiety z pomocą programu LabelImg.([28]).



RYSUNEK 8. Wygląd okna programu

Etykiety zostały zapisane w plikach .xml z nazwą pliku taką samą jak wygenerowane zdjęcie.

```

1 <object>
2   <name>orange</name>
3   <pose>Unspecified</pose>
4   <truncated>0</truncated>
5   <difficult>0</difficult>
6   <bndbox>
7     <xmin>242</xmin>
8     <ymin>487</ymin>
9     <xmax>503</xmax>
10    <ymax>759</ymax>
11  </bndbox>
12</object>
```

LISTING 2. Format zapisu etykiet generowany przez LabelImg

Następnie podzieliłem dane na zbiór treningowy, testowy i ewaluacyjny w proporcjach 75/15/15.

- (2) Kolejnym krokiem jest wybór architektury sieci. Mógłbym stworzyć ją od podstaw korzystając z api Keras dołączonego do TensorFlow ([29]), ale nie byłby on zbyt efektywny w porównaniu z gotowymi rozwiązaniami. Dlatego skorzystałem z gotowego modelu z TensorFlow Model Zoo ([30]) i wytrenowałem go na własnych danych. Jest tam dostępny szeroki wachlarz modeli do wyboru, łącznie z rozpisanymi danymi o prędkości i dokładności ich interferencji. Do mojej aplikacji wybrałem model oparty na architekturze SSD, "SSD MobileNet V2 FPNLite 640x640". Pomijam opis procesu jego instalacji i tworzenia środowiska, ponieważ jest on długi, nieciekawy i opiera się na przepisywaniu komend (zainteresowanego czytelnika odsyłam do źródeł [31]).
- (3) Przed rozpoczęciem treningu możemy zmienić hiperparametry modelu w dołączonym pliku pipeline.config. Jest ich cała masa, ale najbardziej interesujące są dla nas trzy z nich: *batch_size*, *learning_rate_base* i *warmup_learning_rate*.

W przypadku *batch_size*, główną zaletą ustawienia go na wartość większą niż 1 jest przyśpieszenie procesu trenowania sieci kosztem lekkiej utraty dokładności. Przy *learning_rate* działa to podobnie, jednakże przy zbyt wysokiej wartości zajdzie ryzyko przeskoczenia minimum lokalnego, a przy zbyt niskim ryzyko utknięcia funkcji błędu przed jego osiągnięciem. Dlatego należy wytrenować kilka sieci na różnych wartościach learning rate i wybrać najlepszą z nich. W naszym modelu mamy możliwość skorzystania z fazy rozgrzewki, czyli rozpoczęcia treningu z niższą wartością *warmup_learning_rate*, która będzie stopniowo rosła przez określoną liczbę początkowych kroków, aż do osiągnięcia wartości *learning_rate_base*. Proces ten jest bardzo pomocny w przypadku trenowania sieci z wyjątkowo wysokim zróżnicowaniem

danych treningowych, w naszym przypadku raczej tak nie jest, ale nie zaszkodzi z niego skorzystać. Wytrenowałem 3 modele, każdy z lekko zmienionymi parametrami. Każdy z nich miał ustawiony *batch_size* na 1:

- Pierwszemu ustawiłem *warmup_learning_rate* na 0.015 i *learning_rate_base* na 0.08
- Drugi ustawiłem tak samo jak pierwszy, ale z podwójną liczbą kroków (parametr *total_steps*)
- Trzeciemu ustawiłem *warmup_learning_rate* na 0.015 i *learning_rate_base* na 0.02

Wytrenowanie każdego z nich zajęło na karcie graficznej RTX 3080 poniżej godziny.

- (4) Modele wyeksportowałem do plików .tflite. Interferencję takiego modelu będziemy wykonywać za pomocą modułu tensorflow.lite.

```

1 from tensorflow.lite.python.interpreter import Interpreter
2
3 interpreter = Interpreter(model_path=r"model.tflite")
4 interpreter.allocate_tensors()
5 input_details = interpreter.get_input_details()
6 output_details = interpreter.get_output_details()
```

LISTING 3. Kod ładowający model

Wyświetlenie zmiennej *output_details* w konsoli da nam informacje o formacie danych wyjściowych sieci:

```

1 [{"name": 'StatefulPartitionedCall:1', 'index': 339, 'shape': array
2   ([ 1, 10]), 'shape_signature': array([ 1, 10]), 'dtype': <class
3     'numpy.float32'>, 'quantization': (0.0, 0), ,
4     'quantization_parameters': {'scales': array([], dtype=float32), ,
5       'zero_points': array([], dtype=int32), 'quantized_dimension': 0},
6     'sparsity_parameters': {}}, {'name': 'StatefulPartitionedCall:3',
7       'index': 337, 'shape': array([ 1, 10, 4]), 'shape_signature':
8       array([ 1, 10, 4]), 'dtype': <class 'numpy.float32'>, ,
9       'quantization': (0.0, 0), 'quantization_parameters': {'scales':
10      array([], dtype=float32), 'zero_points': array([], dtype=int32),
11      'quantized_dimension': 0}, 'sparsity_parameters': {}}, {'name': ,
12     'StatefulPartitionedCall:0', 'index': 340, 'shape': array([1]), ,
13     'shape_signature': array([1]), 'dtype': <class 'numpy.float32'>, ,
14     'quantization': (0.0, 0), 'quantization_parameters': {'scales':
15       array([], dtype=float32), 'zero_points': array([], dtype=int32),
16       'quantized_dimension': 0}, 'sparsity_parameters': {}}, {'name': ,
17     'StatefulPartitionedCall:2', 'index': 338, 'shape': array([ 1,
18       10]), 'shape_signature': array([ 1, 10]), 'dtype': <class 'numpy.
19       float32'>, 'quantization': (0.0, 0), 'quantization_parameters':
20       {'scales': array([], dtype=float32), 'zero_points': array([], ,
21         dtype=int32), 'quantized_dimension': 0}, 'sparsity_parameters': {}}]

```

LISTING 4. Zwracana wartość output_details

Interesujące są dla nas wartości w kluczach "shape". Są to wymiary zwracanych przez sieć danych o pozycjach bounding boxów, prawdopodobieństwach występowania w boxach obiektów oraz klasach tych obiektów. Potrzebne nam one będą w późniejszym etapie. Interferencję przeprowadzamy w następujący sposób:

```

1 interpreter.set_tensor(input_details[0]['index'], input_data)
2 interpreter.invoke()
3 classes = interpreter.get_tensor(output_details[3]['index'])[0]
4 boxes = interpreter.get_tensor(output_details[1]['index'])[0]
5 scores = interpreter.get_tensor(output_details[0]['index'])[0]

```

Gdzie input_data to wartości pikseli zdjęcia załadowanego na przykład przez bibliotekę OpenCV (najlepiej jeszcze znormalizowane do wartości od 0 do 1) i przeskalowanego do rozdzielczości 640x640, ponieważ takich wymiarów oczekuje model.

- (5) Ostatnim krokiem jest podjęcie decyzji którą z wytrenowanych sieci wybierzemy do aplikacji. W tym celu dla każdej należy stworzyć tablicę pomyłek na podstawie danych ze zbioru ewaluacyjnego. Najpierw musiałem ustalić jak zdefiniuję przypadki TP, FP i FN. Przypominam, że jedno zdjęcie ze zbioru może zawierać w sobie kilka obiektów do wykrycia, czyli każde ma przyporządkowaną jakąś liczbę bounding boxów.

- True Positive - przewidziany bounding box posiada skorelowany rzeczywisty bounding box oraz obydwa mają przypisaną daną klasę
- False Positive - przewidziany bounding box ma przypisaną daną klasę oraz nie posiada skorelowanego rzeczywistego bounding boxa LUB posiada taki, który ma przypisaną inną klasę
- False Negative - przewidziany bounding box nie ma przypisanej danej klasy, ale posiada skolerowaną rzeczywistą bounding box, który ma przypisaną daną klasę LUB jeden z rzeczywistych bounding boxów ma przypisaną tę klasę ale nie posiada skorelowanego przewidywanego bounding boxa

Należy również zdefiniować co rozumiem przez korelację. Uznałem że dwa bounding boxy są ze sobą skorelowane jeśli są podobnego rozmiaru, czyli iloraz ich obwodów jest pomiędzy 0.9 a 1.1 oraz suma odległości ich wierzchołków od siebie (to znaczy dla każdego z wierzchołków odległość od odpowiadającego wierzchołka) jest mniejsza niż 10% obwodu mniejszego bounding boxa. Dla każdego z modeli wyznaczyłem macierz pomyłek i wyliczyłem metryki jakości. W każdej z nich zawarłem dodatkowy wiersz i kolumnę oznaczające przypadki, w których model wykrył obiekt danej klasy na zdjęciu na którym go nie było lub obiekt był na zdjęciu i nie został wykryty.

(a) $warmup_learning_rate = 0.015$, $learning_rate_base = 0.08$

Confusion matrix							
Predicted	apple	pepper	orange	onion	lemon	no_object	
	50 22.32%	1 0.45%	33 14.73%	33 14.73%	1 0.45%	58 25.89%	
apple	51 98.04%	1 1.96%					
pepper	1 0.45%	29 12.95%					
orange		1 0.45%	33 14.73%				
onion				33 14.73%	1 0.45%	2 0.89%	
lemon					58 25.89%	3 1.34%	
no_object	3 1.34%	4 1.79%	1 0.45%	1 0.45%	3 1.34%		
sum_col	54 92.59% 7.41%	35 82.86% 17.14%	34 97.06% 2.94%	34 97.06% 2.94%	62 93.55% 6.45%	5 0.00% 100.00%	224 90.62% 9.38%
Actual	apple	pepper	orange	onion	lemon	no_object	sum_ln

$$P_1 = \frac{1}{5} \cdot \left(\frac{50}{50+1} + \frac{29}{29+1} + \frac{33}{33+1} + \frac{33}{33+3} + \frac{58}{58+3} \right) \approx 0.96$$

$$R_1 = \frac{1}{5} \cdot \left(\frac{50}{50+4} + \frac{29}{29+6} + \frac{33}{33+1} + \frac{33}{33+1} + \frac{58}{58+4} \right) \approx 0.93$$

$$A_1 = \frac{203}{203 + 9 + 12} \approx 0.91$$

(b) a) z dwa razy większa liczba kroków treningowych

Confusion matrix							
Predicted	apple	pepper	orange	onion	lemon	no_object	
Actual	apple	36 16.22%	1 0.45%				37 97.30% 2.70%
apple		11 4.95%				1 0.45%	12 41.67% 8.33%
pepper			15 6.76%		3 1.35%	1 0.45%	19 78.95% 21.05%
orange				7 3.15%			7 100% 0.00%
onion							36 83.33% 16.67%
lemon			4 1.80%	1 0.45%	30 13.51%	1 0.45%	111 0.00% 100.00%
no_object	18 8.11%	23 10.56%	15 6.76%	26 11.71%	29 13.06%		222 44.59% 55.41%
sum_col	54 66.67% 33.33%	35 31.43% 68.57%	34 44.12% 55.88%	34 20.56% 79.41%	62 48.39% 51.61%	3 0.00% 100.00%	222 44.59% 55.41%

$$P_2 = \frac{1}{5} \cdot \left(\frac{36}{36+1} + \frac{11}{11+1} + \frac{15}{15+3+1} + \frac{7}{7} + \frac{30}{30+6} \right) \approx 0.91$$

$$R_2 = \frac{1}{5} \cdot \left(\frac{36}{36+18} + \frac{11}{11+24} + \frac{15}{15+19} + \frac{7}{7+26} + \frac{30}{30+32} \right) \approx 0.42$$

$$A_2 = \frac{99}{99+12+111} \approx 0.45$$

(c) *warmup_learning_rate* = 0.015, *learning_rate_base* = 0.02

Confusion matrix							
Predicted	apple	pepper	orange	onion	lemon	no_object	
Actual	apple	50 21.83%					50 100% 0.00%
apple		1 0.44%	33 14.41%				34 97.06% 2.94%
pepper			33 14.41%				35 94.29% 5.71%
orange				32 13.97%	1 0.44%	2 0.87%	35 92.43% 8.57%
onion					59 25.76%	6 2.62%	65 90.77% 9.23%
lemon							10 0.00% 100.00%
no_object	3 1.31%	2 0.87%	1 0.44%	2 0.87%	2 0.87%		229 90.39% 9.61%
sum_col	54 92.59% 7.41%	35 94.29% 5.71%	34 97.06% 2.94%	34 94.12% 5.88%	62 95.16% 4.84%	10 0.00% 100.00%	229 90.39% 9.61%

$$P_3 = \frac{1}{5} \cdot \left(\frac{50}{50} + \frac{33}{33+1} + \frac{33}{33+2} + \frac{32}{32+3} + \frac{59}{59+6} \right) \approx 0.95$$

$$R_3 = \frac{1}{5} \cdot \left(\frac{50}{50+4} + \frac{33}{33+2} + \frac{33}{33+1} + \frac{32}{32+2} + \frac{59}{59+3} \right) \approx 0.95$$

$$A_3 = \frac{207}{207+12+10} \approx 0.90$$

Z tych wyników można wyciągnąć następujące wnioski:

- Sieć a) jest najbardziej dokładna.
- Sieć b) została przetrenowana. Wykazała bardzo dużą ilość False Negativów ponieważ nauczyła się danych treningowych na pamięć i dane z małymi odstępami od nich uznała za niepasujące.
- Sieć c) wykazuje lekko mniejszą dokładność niż a), ale posiada mniej False Negativów

Ostatecznie wybrałem sieć a). Sieć c) jest gorsza do wyznaczonego zadania, ponieważ nawet biorąc pod uwagę jej większą metrykę czułości, to w mojej aplikacji ważniejsze jest aby owoce były wykrywane tam gdzie są niż aby nie były wykrywane tam gdzie ich nie ma (to znaczy False Positives są mniej istotne niż False Negatives), ponieważ użytkownik będzie miał pełen podgląd na działanie sieci i w razie czego może dokonać korekcji zdjęcia.

3.3. Opis ważniejszych fragmentów kodu aplikacji.

Do stworzenia aplikacji do detekcji produktów spożywczych wybrałem bibliotekę Kivy, która pozwala na napisanie w pythonie UI i podłączenia do niego modelu tflite. Po napisaniu aplikacji skompilowałem ją do pliku instalacyjnego .apk z pomocą narzędzia buildozer. Aplikacja została przetestowana na telefonie Redmi Note 11 Pro 5G z Androidem w wersji 11 i architekturą arm64-v8a. Aplikacja nie jest kompatybilna z systemem IOS. Kod aplikacji dzieli się na 5 plików .py oraz jedną pomocną bibliotekę napisaną w javie.

3.3.1. *android_permissions.py*.

Aplikacja wymaga dostępu do kamery telefonu, internetu oraz odczytu plików zewnętrznych. Moduł odpowiada za zebranie tych permisji.

3.3.2. *applayout.py*.

Służy do stworzenia UI aplikacji. Definiuje guziki na ekranie i ich funkcjonalność i umieszcza widok kamery z *edgedetect.py* w odpowiednie miejsce na ekranie. Guziki są cztery, ich funkcjonalność od lewej do prawej strony umiejscowienia na ekranie:

- zmniejszenie miejsca na monetę na ekranie
- zeskanowanie kodu QR
- przechwycenie obrazu z zaznaczonymi produktami i przekierowanie użytkownika do bilansu
- zwiększenie miejsca na monetę na ekranie

Poza guzikami, użytkownik ma również możliwość przemieszczenia miejsca na monetę w miejsce w którym przytrzymał palec (tzw. long press).

3.3.3. *edgedetect.py*.

Aby stworzyć widok kamery do aplikacji posłużyłem się dodatkową biblioteką camera4kivy. Za kamerę odpowiada zawarta w niej klasa *Preview*. W klasie istnieje możliwość zdefiniowania metody *analyze_pixels_callback*, która zapewnia dostęp do aktualnie wyświetlanej klatki obrazu oraz *canvas_instructions_callback*, która pozwala na modyfikację tej klatki.

```

1 from kivy.graphics import Color, Rectangle, Line
2 import numpy as np
3 from camera4kivy import Preview
4 from model import DetectionModel
5
6 class EdgeDetect(Preview):
7
8     def __init__(self, **kwargs):
9         super().__init__(**kwargs)
10        self.model = DetectionModel()
11        self.detections = []
12        self.auto_analyze_resolution = [640, 640]
13        self.coin = ()
```

```

15     def analyze_pixels_callback(self, pixels, image_size, image_pos, scale
16         , mirror):
17
18         rgba = np.fromstring(pixels, np.uint8).reshape(image_size[1],
19                                         image_size[0], 4)
20
21         detections, coin = self.model.annotate(rgba, scale, mirror,
22         image_pos, image_size)
23
24         self.detections = detections
25
26         self.coin = coin
27
28
29     def canvas_instructions_callback(self, texture, tex_size, tex_pos):
30
31         Color(0.83,0.68,0.21,1)
32
33         Line(circle=self.coin)
34
35         for detection in self.detections:
36
37             Color(1,1,1,1)
38
39             Line(rectangle=(detection['x'], detection['y'], detection['w'],
40                 ], detection['h']), width = 3)
41
42             Rectangle(size = detection['t'].size,
43
44                 pos = [(detection['x'] + detection['w'])//2, (detection['y'] +
45                 detection['h'])],
46
47                 texture = detection['t'])

```

W konstruktorze definiowany jest atrybut model, instancja klasy z modułu *model.py*, który zostanie opisany poniżej. W metodzie *analyze_pixels_callback* tworzona jest tablica pikseli z obecnej klatki, a następnie przekazywana do metody *annotate* atrybutu model. Metoda ta dokonuje detekcji obiektów i zwraca je w postaci listy, która jest wykorzystywana do narysowania na podglądzie bounding boxów z etykietami w metodzie *canvas_instructions_callback*. Oprócz tego zwracane są dane do narysowania miejsca na monetę.

Użytkownik ma zadanie umieścić na zdjęciu monetę 2-złotową i ustawić telefon na takiej wysokości, aby rozmiar monety pokrył się z rozmiarem wyznaczonego miejsca na monetę. Zarówno bounding boxy jak i miejsce potrzebują do bycia wyznaczonym wszystkich argumentów z funkcji *analyze_pixels_callback*. Ponieważ widok kamery w aplikacji, a rzeczywisty przechwytywany obraz różnią się przez implementację obsługi kamery w Kivy. Argumenty oznaczają kolejno:

- image_size - rozmiar rzeczywistego obrazu,
- image_pos - umiejscowienie obrazu w interfejsie Kivy
- scale - współczynnik przeskalowania obrazu
- mirror - boolean mówiący czy obraz został obrócony w pionie (zależy od typu i ustawień kamery, obrót w poziomie zachodzi natomiast zawsze)

3.3.4. *model.py*.

Definiuje klasę DetectionModel, która wykonuje zadanie detekcji obiektów. Jest to najważniejsza klasa w całym projekcie.

```

1 from jnius import autoclass
2 import os
3 import numpy as np
4 import cv2
5 import time
6 from plyer import accelerometer
7 from kivy.core.text import Label as CoreLabel

```

LISTING 5. Używane biblioteki

```

1 class DetectionModel:
2
3     def __init__(self):
4         accelerometer.enable()
5         self.scores = []
6         self.boxes = []
7         self.classes = []
8         self.timer = 0
9         self.last_acc = (0,0,0)
10
11
12     File = autoclass('java.io.File')
13     Interpreter = autoclass('org.tensorflow.lite.Interpreter')
14     InterpreterOptions = autoclass('org.tensorflow.lite.
15         Interpreter$Options')
16
17     model_path = File(os.path.join(os.getcwd(), 'model.tflite'))
18     options = InterpreterOptions()
19     interpreter = Interpreter(model_path, options)
20     interpreter.allocateTensors()
21     input_output_class = autoclass('org.test.InputOutputClass')
22
23     self.input_output_class = input_output_class()
24     self.interpreter = interpreter
25     self.labels = ["apple", "pepper", "orange", "onion", "lemon"]
26     self.last_result = []

```

W konstruktorze definiowane są atrybuty które będą przechowywać dane wyjściowe sieci, zbiór nazw etykiet, timer i dane o ostatnim odczycie przyśpieszeniomierza telefonu, rozpoczynany jest również proces śledzenia jego pomiarów. Definiowany jest również atrybut przechowujący dane z ostatniej wykonanej interferencji, wykorzystywany przy komunikacji

z bilansem. Następnie tworzony jest interpreter modelu, w trochę inny sposób niż w poprzednim podrozdziale. Kivy umożliwia używanie w aplikacji funkcji zawartych w pythonie, ale nie pozwala na korzystanie z zewnętrznych bibliotek jak TensorFlow. Dlatego korzystam z biblioteki pyjnius, która umożliwia wykonywanie kodu w Javie (czyli w języku który w przeciwieństwie do pythona każdy telefon z Androidem domyślnie potrafi kompilować), w tym z zewnętrznych bibliotek (narzędzie buildozer zawiera w pliku .apk wszystkie wymagane Javowe biblioteki). Za pomocą funkcji autoclass pobieram wszystkie potrzebne klasy z Javy i tworzę interpreter. Korzystam przy tym z pomocniczej klasy InputOutputClass, którą sam napisałem.

```

1 package org.test;
2 import java.util.Map;
3 import java.util.TreeMap;
4 import java.nio.ByteBuffer;
5 import java.util.List;
6
7 public class InputOutputClass {
8
9     public Object get_input(ByteBuffer inputData){
10         Object[] inputs = new Object[1];
11         inputs[0] = inputData;
12         return inputs;
13     }
14
15     public Map<Integer, Object> get_output(){
16         Map<Integer, Object> outputMap = new TreeMap<>();
17         float[][][] boxes = new float[1][10][4];
18         float[][] scores = new float[1][10];
19         float[] notImportant = new float[1];
20         float[][] classes = new float[1][10];
21         outputMap.put(0, scores);
22         outputMap.put(1, boxes);
23         outputMap.put(2, notImportant);
24         outputMap.put(3, classes);
25         return outputMap;
26     }
27 }
```

LISTING 6. InputOutputClass.java

Była mi ona potrzebna, ponieważ w/javowej wersji Tensorflowa należy samodzielnie podać wymiary danych wejściowych i wyjściowych (sposób uzyskania informacji o wymiarze danych wyjściowych modelu zawarłem w poprzednim podrozdziale).

```

1  def get_input_object(self, image):
2      input_data = np.expand_dims(image, axis=0)
3      input_mean = 127.5
4      input_std = 127.5
5      input_data = (np.float32(input_data) - input_mean) / input_std
6      input_data = ByteBuffer.wrap(input_data.tobytes())
7      return self.input_output_class.get_input(input_data)
8
9  def get_output_object(self):
10     return self.input_output_class.get_output()
11
12 def interference(self, image):
13     inp = self.get_input_object(image)
14     outp = self.get_output_object()
15     self.interpreter.runForMultipleInputsOutputs(inp, outp)
16     scores = outp.get(0)[0]
17     boxes = outp.get(1)[0]
18     classes = outp.get(3)[0]
19     return scores, boxes, classes
20
21 def update_annotation_parameters(self, image):
22     image = image[:, :, :3]
23     image = cv2.resize(image, (640, 640))
24     self.scores, self.boxes, self.classes = self.interference(image)

```

Metoda *get_input_object* dokonuje normalizacji obrazu i zwraca go w wymaganym formacie, *get_output_object* zwraca jawową mapę, która zostanie wypełniona danymi wyjściowymi. Funkcja *interference* dokonuje interferencji na podanym obrazie i zwraca dane wyjściowe.

Metoda *update_annotation_parameters* rozpoczyna cały proces wykrywania obiektów na obrazie poprzez skalowanie go do oczekiwanej przez sieć rozdzielczości i przekazanie go do interferencji i kończy aktualizując parametry.

```

1  def get_results(self, image, image_scale, mirror, image_pos,
2      image_size):
3      scores, boxes, classes = self.scores, self.boxes, self.classes
4      imH, imW, _ = image.shape
5      results = []
6      coin_x = 3*imW//4 * image_scale[0] + image_pos[0]
7      coin_y = imH//4 * image_scale[1] + image_pos[1]
8      coin_radius = imW//7 * image_scale[0]
9      coin = (coin_x, coin_y, coin_radius)
10     last_result = []
11     for i in range(len(scores)):
12         if ((scores[i] > 0.9) and (scores[i] <= 1.0)):
13             ymin = int(max(1,(boxes[i][0] * imH)))
14             xmin = int(max(1,(boxes[i][1] * imW)))
15             ymax = int(min(imH,(boxes[i][2] * imH)))
16             xmax = int(min(imW,(boxes[i][3] * imW)))
17             x = xmin
18             y = ymin
19             w = xmax - xmin
20             h = ymax - ymin
21             y = max(image_size[1] -y -h, 0)
22             if mirror:
23                 x = max(image_size[0] -x -w, 0)
24                 x = round(x * image_scale[0] + image_pos[0])
25                 y = round(y * image_scale[1] + image_pos[1])
26                 w = round(w * image_scale[0])
27                 h = round(h * image_scale[1])
28                 object_name = self.labels[int(classes[i])]
29                 label = CoreLabel(font_size = w//6)
30                 label.text = f'{object_name}'
31                 label.refresh()
32                 last_result.append({"name": object_name, "width": w/(coin_radius*2), "height": h/(coin_radius*2)})
33                 results.append({"x":x, "y": y, "w": w, "h": h, "t": label.texture})
34             self.last_result = last_result
35             return results, coin
36
37     def annotate(self, image, scale, mirror, image_pos, image_size,
38         optimize=True):
39         if optimize is False:
40             self.update_annotation_parameters(image)
41         elif time.time() - self.timer > 0.05:
42             current_acc = accelerometer.acceleration[:3]

```

```

41         if np.linalg.norm(np.array(self.last_acc) - np.array(
42             current_acc)) < 1:
43             self.update_annotation_parameters(image)
44             if time.time() - self.timer > 1:
45                 self.timer = time.time()
46                 self.last_acc = accelerometer.acceleration[:3]
47             results, coin = self.get_results(image, scale, mirror, image_pos,
48             image_size)
49             return results, coin

```

Z pomocą metody *get_results* wyznaczane są bounding boxy i miejsce na monetę na podglądzie kamery. Oprócz tego, zapisywane są dane o ostatniej interferencji. Funkcja *annotate* łączy wszystkie poprzednie. Interferencja jest dosyć wymagającym obliczeniowo procesem, dlatego zastosowałem 2 sztuczki optymalizacyjne:

- (1) Interferencja wykonywana jest maksymalnie raz na 0.05 sekundy
- (2) Interferencja nie jest wykonywana jeśli użytkownik poruszył mocno telefonem. Dzięki temu aplikacja nie wykonuje detekcji podczas przemieszczania telefonu z obiektu na obiekt, niewielkie ruchy są akceptowalne. Mierzenie ruchu jest wykonywane za pomocą przyśpieszeniomierza.

Z tego właśnie powodu definiowane są parametry scores, boxes i classes, ponieważ w klatkach w których nie zaszła interferencja, na obraz nakładane są po prostu poprzednio wyliczone bounding boxy (fakt że interreferencja zachodzi tylko gdy telefon nie znajduje się w ruchu sprawia że są one zawsze dokładne).

3.3.5. *main.py*.

Łączy wszystkie inne pliki i buduje z nich całą aplikację. Definiowana jest w nim funkcja, która wykonywana jest za każdym razem gdy użytkownik zrobi zdjęcie w aplikacji. Funkcja ta służy do komunikacji aplikacji z bilansem, wysyła ona do api bilansu zdjęcie (w pliku o określonej nazwie) obecnego widoku z zaznaczonymi obiekty, a następnie otwiera przeglądarkę użytkownika i przekierowuje go do strony dodawania produktów do bilansu.

Przy przekierowywaniu, jako parametr query przekazywane są zakodowane w base64 ([32]) dane o ostatniej detekcji. W tym miejscu należy zaznaczyć, że model jest w stanie wykryć maksymalnie 10 obiektów naraz, co daje około 700 znaków tekstu w base64. Liczba tej jest daleko do limitu znaków w query (który, w zależności od przeglądarki wynosi średnio 60 tysięcy).

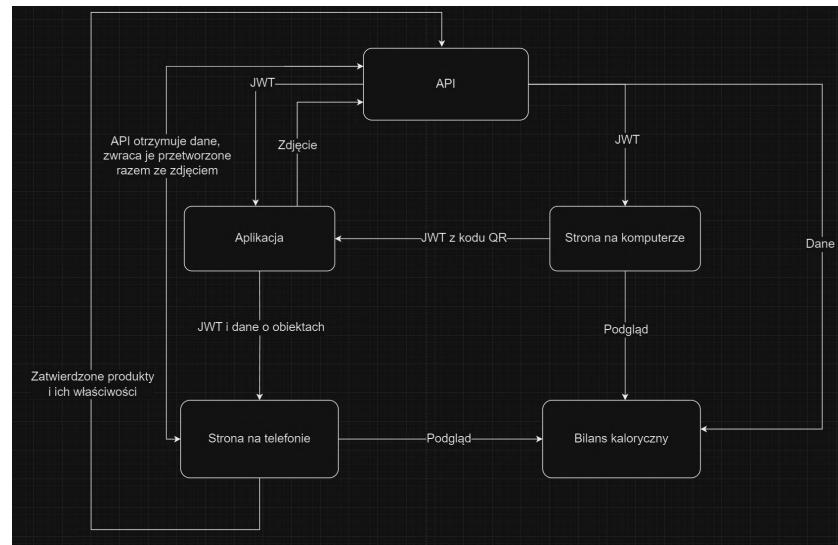
3.4. Bilans i metoda połączenia go z aplikacją.

Część bilansową aplikacji stworzyłem jako osobną stronę internetową dostępną (na dzień 29.11.2023) pod adresem: <http://185.194.141.183:5002>. Zawarłem w niej ukryty system logowania. Dla każdego użytkownika wchodzącego na stronę po raz pierwszy generowany jest *JWT* zawierający id.

Definicja 3.4. *JWT (JSON Web Token)* ([33]) to metoda autoryzacji pozwalająca na zapis dowolnych danych w formacie json i zaszyfrowanie ich jako string. Jest on generowany po stronie serwera przy użyciu sekretnego klucza. Każdy kto posiada token jest w stanie go odszyfrować i uzyskać dostęp do zawartych w nim danych, ale nie jest w stanie ich zmodyfikować bez znajomości sekretnego klucza. Autoryzacja następuje poprzez wymianę tokena pomiędzy klientem a serwerem i weryfikację czy token został utworzony przy użyciu tego samego sekretnego klucza.

Raz wygenerowany klucz zapisuje się w *localStorage* przeglądarki i wszystkie rekordy w bazie danych danego użytkownika są w relacji z zawartym w nim id.

Aby połączyć bilans z aplikacją do wykrywania produktów użytkownik może zeskanować kod QR dostępny na stronie, jeśli tego nie zrobi, to aplikacja wygeneruje swój własny token, który zostanie również przypisany do aplikacji w przeglądarce mobilnej. Zawarte w tokenie id służy również jako nazwa zdjęcia ostatniego wykrycia, które jest wyświetlane na ekranie dodawania produktów.

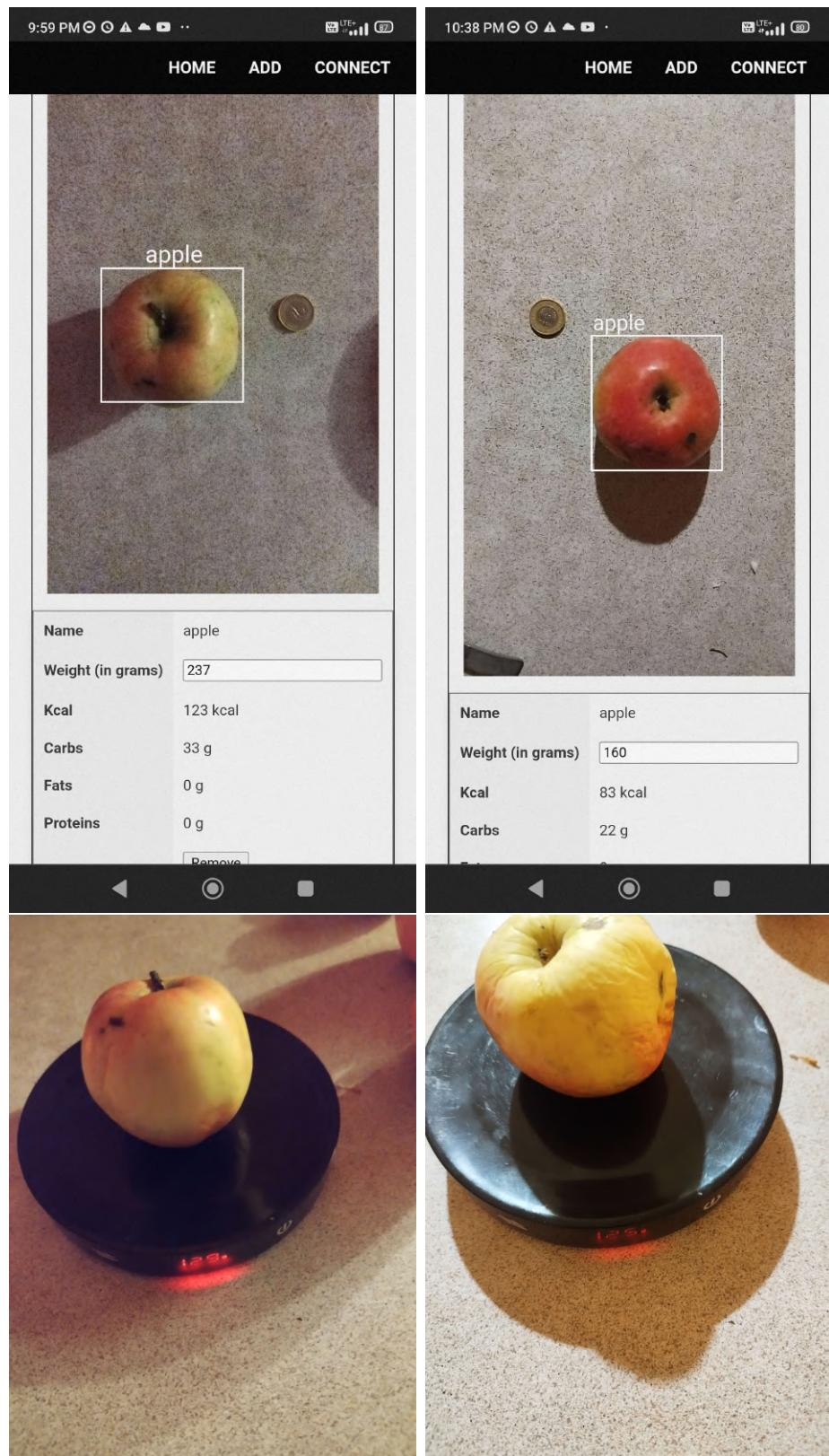


RYSUNEK 9. Diagram wymiany danych w projekcie

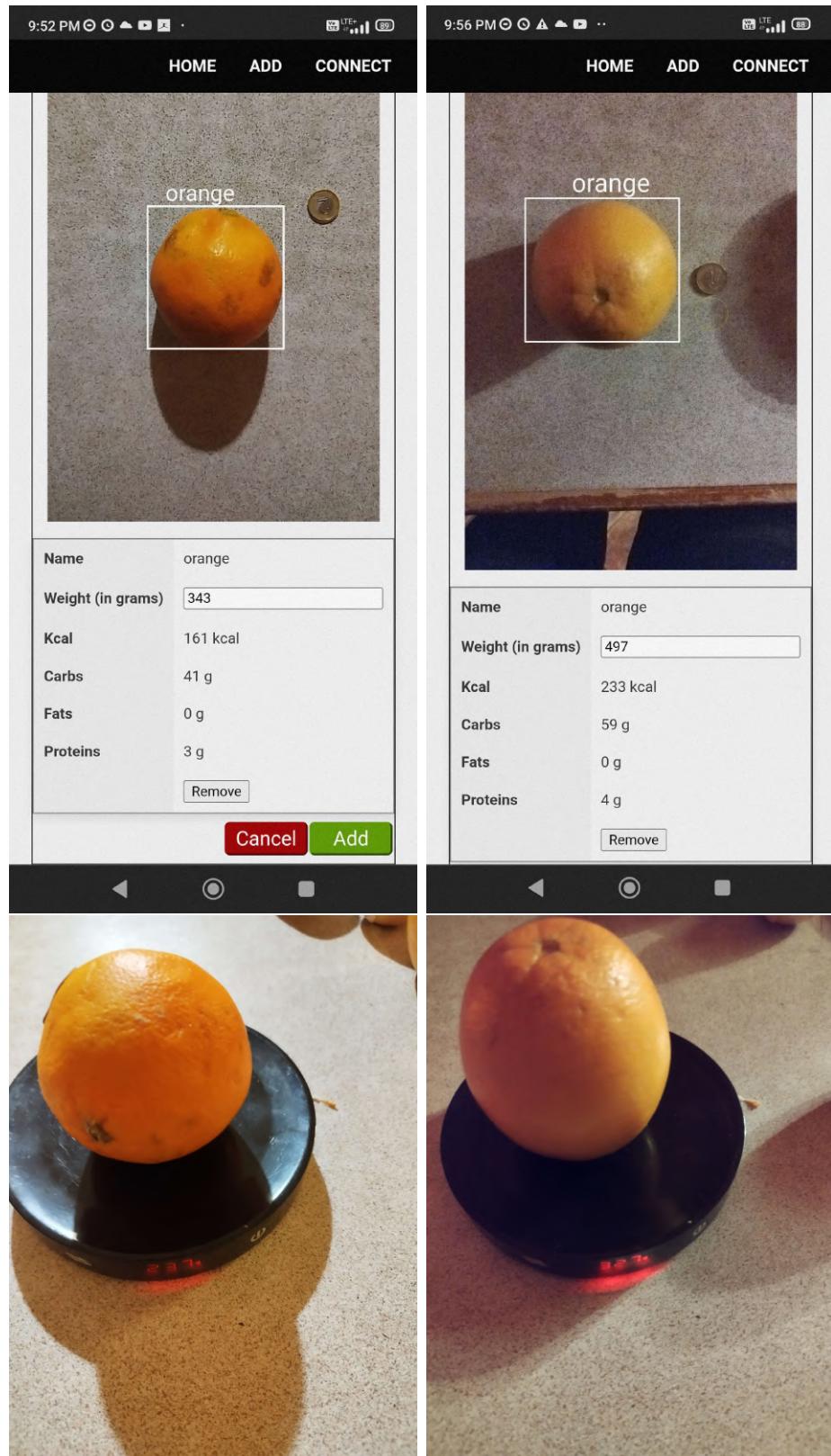
Bilans składa się z pięciu endpointów REST API i trzech stron:

- /api/products_image - odpowiada za przekazywanie zdjęcia pomiaru produktów w aplikacji
- /api/get_token - generowanie JWT
- /api/products - dodawanie i pobieranie bilansu użytkownika
- /api/qr - generowanie i oczytywanie kodu qr z JWT
- /api/nutrition - wymiana danych z aplikacji (tych zakodowanych w base64 w query) zawierających nazwę produktu ze zmierzona monetą wysokością i szerokością bounding boxa, na wyliczone wartości odżywcze. Algorytm polega na sprawdzeniu w której stronie jest obrócony produkt poprzez porównanie wysokości i szerokości bounding boxa, wyznaczenie brakującego wymiaru z przyporządkowanej w bazie danych dla każdego produktu proporcji i wyliczenie objętości, a następnie wagi i wartości odżywczych.
- /home - strona z widokiem tabeli z dodanymi do bilansu produktami z możliwością zmiany daty
- /add - strona wykorzystywana przy dodawaniu do bilansu wyznaczonych przez aplikację produktów, użytkownik ma możliwość usunięcia wykrytego produktu i zmianę proponowanej wagi, co aktualizuje od razu wartości odżywcze
- /connect - strona z kodem QR zawierającym JWT z localStorage, użytkownik może go zeskanować aplikacją w celu połączenia jej z bilansem na komputerze (bilans na telefonie synchronizuje się z aplikacją przy dodawaniu produktów w /add)

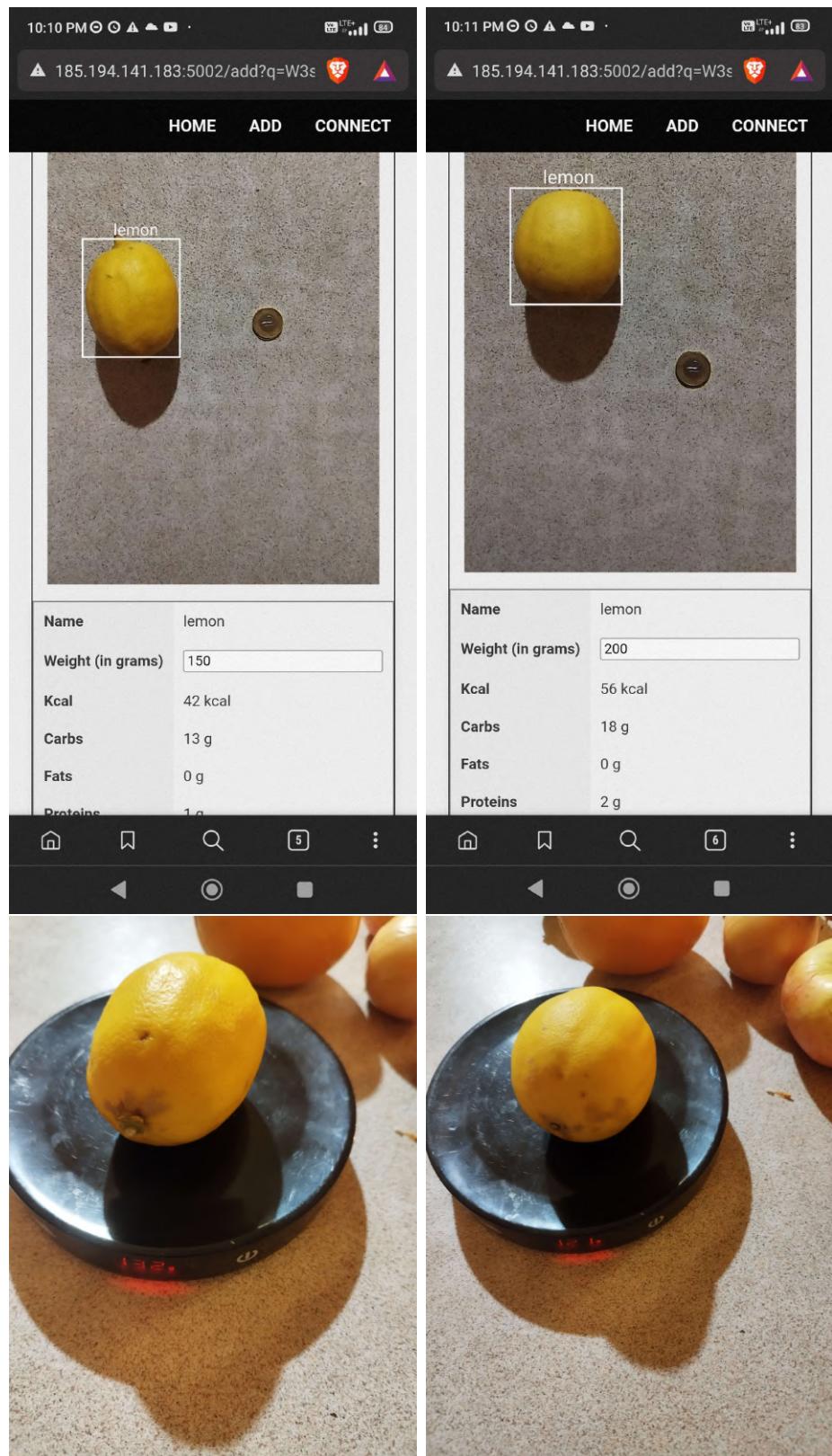
3.5. Praktyczny test działania aplikacji. Do testów praktycznych wybrałem po 2 jabłka, cytryny, cebule i pomarańcze. Zmierzyłem je za pomocą wagi kuchennej i za pomocą aplikacji, wyniki prezentują się następująco:



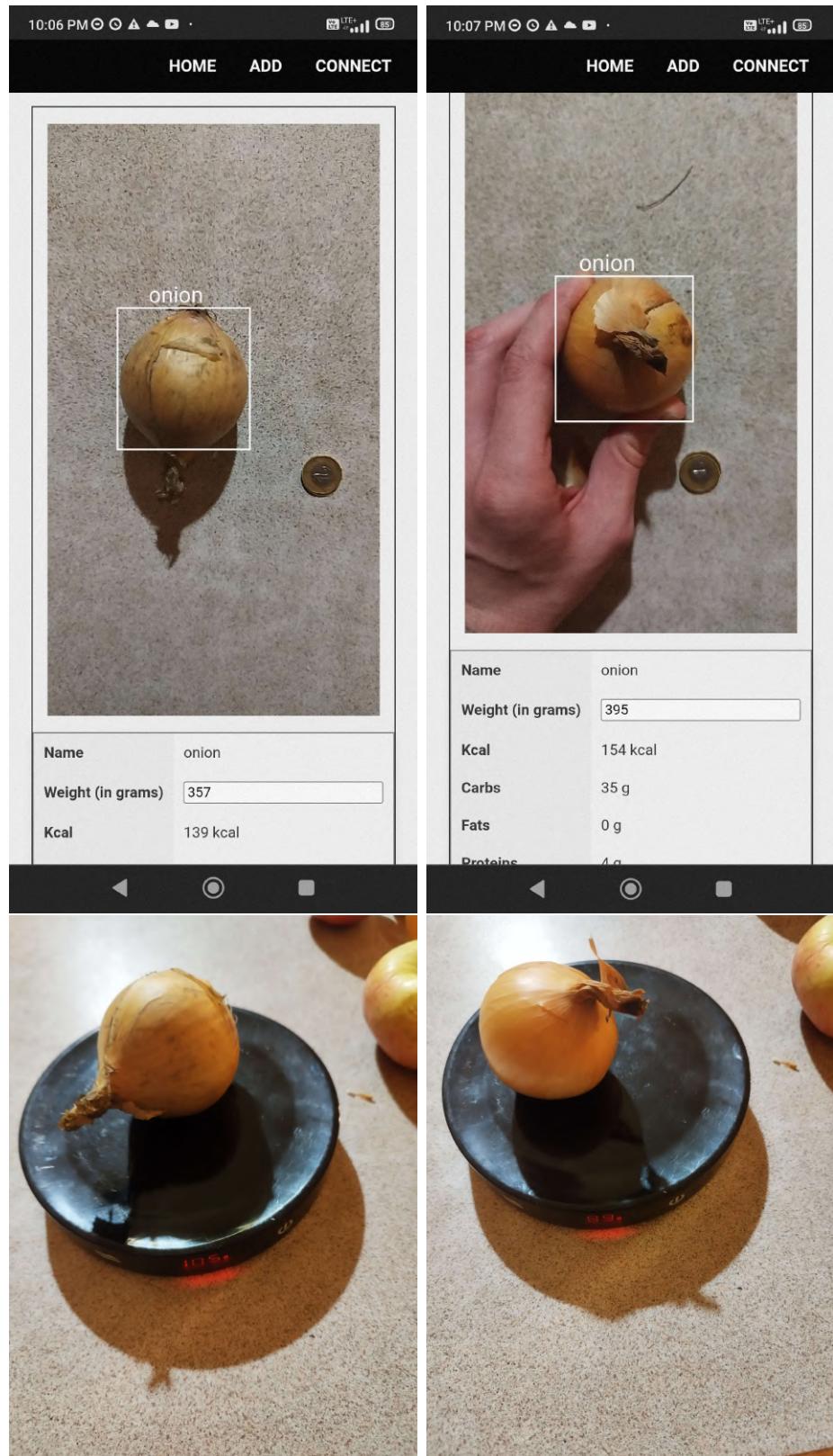
RYSUNEK 10. Jabłka



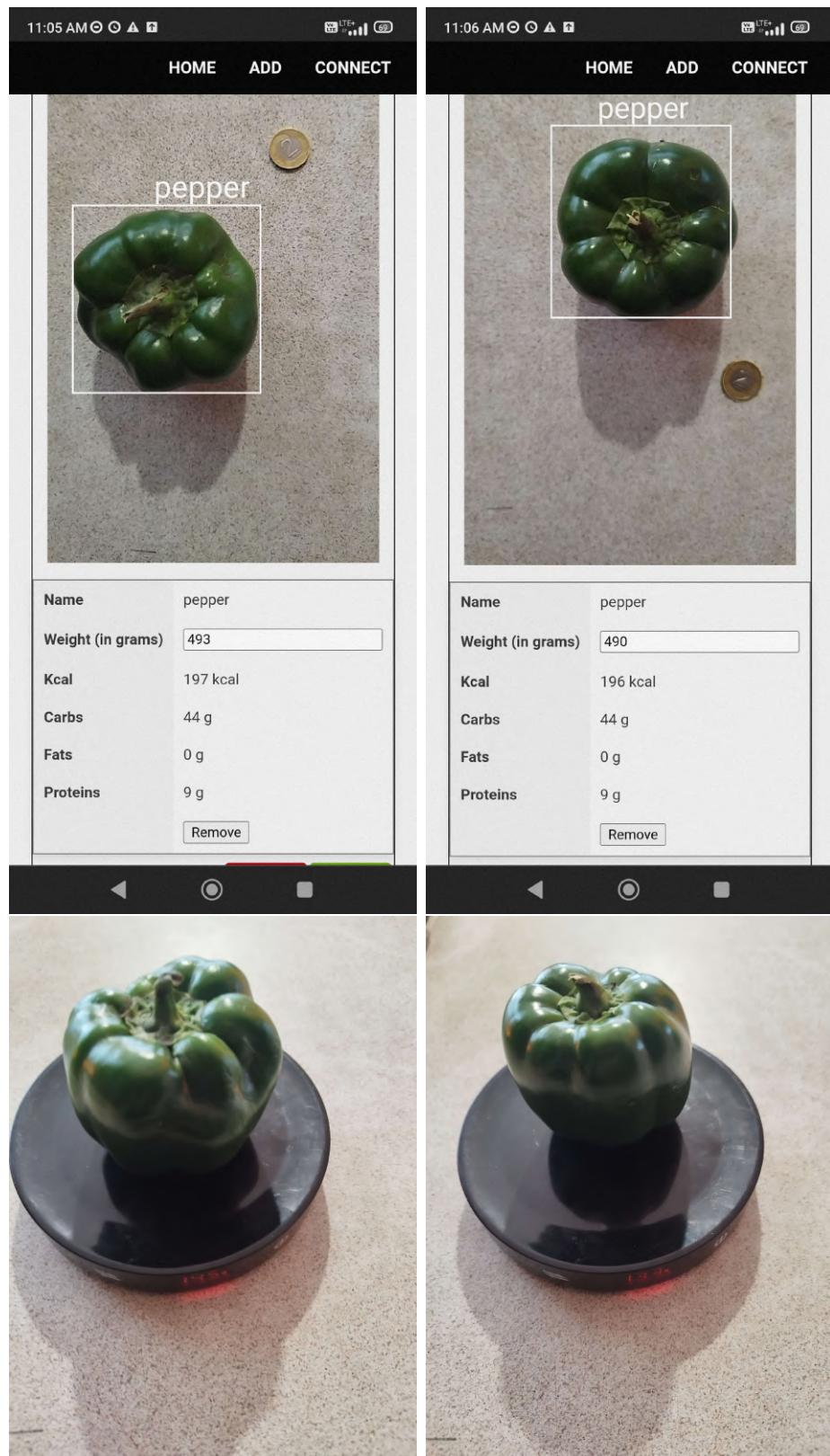
RYSUNEK 11. Pomarańcze



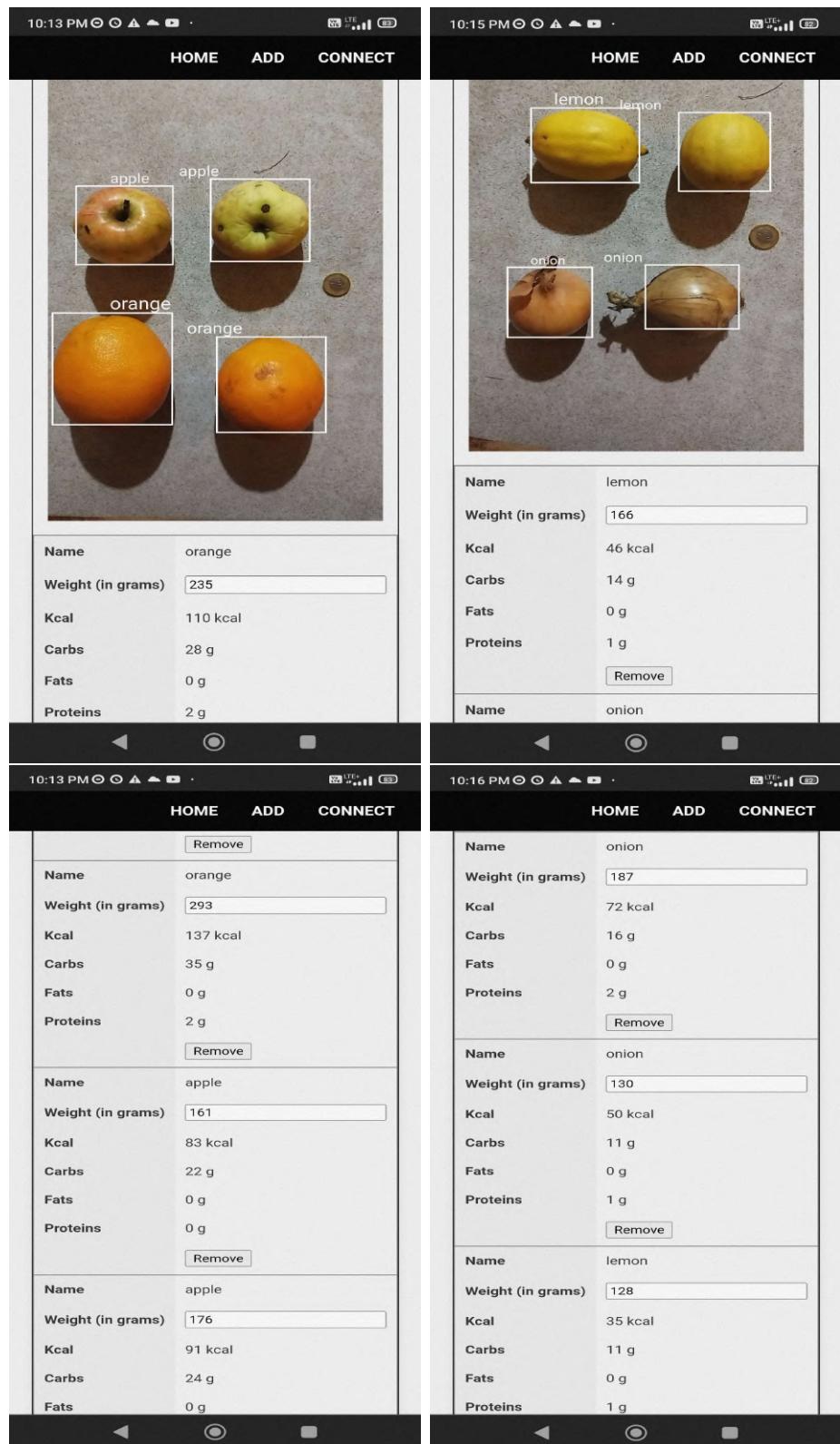
RYSUNEK 12. Cytryny



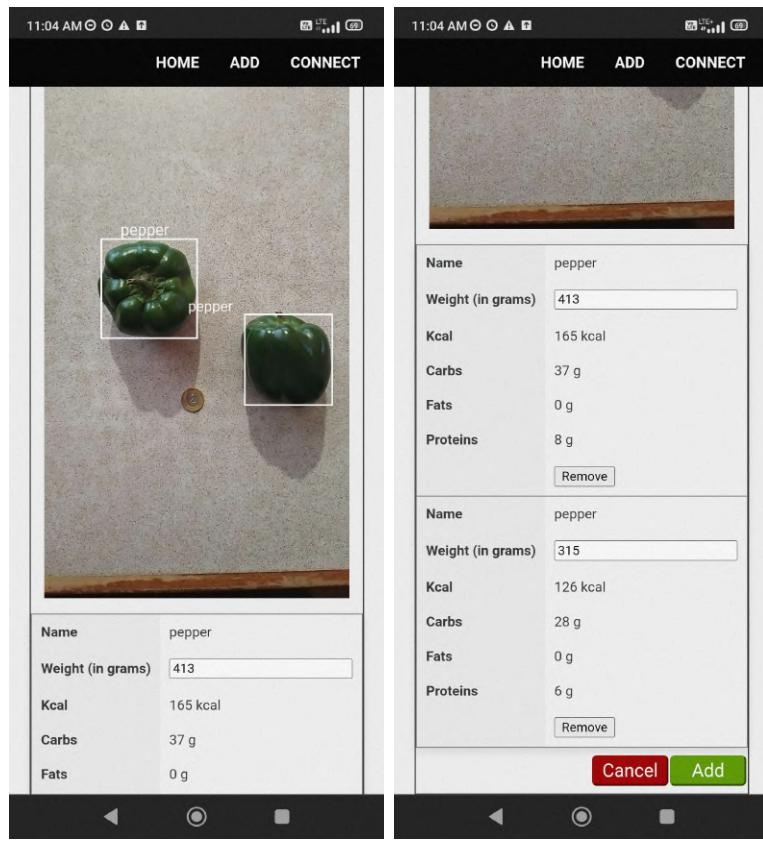
RYSUNEK 13. Cebule



RYSUNEK 14. Papryki



RYSUNEK 15. Pomiar grupowy 1



RYSUNEK 16. Pomiar grupowy 2

Wyciągnąłem z nich następujące wnioski:

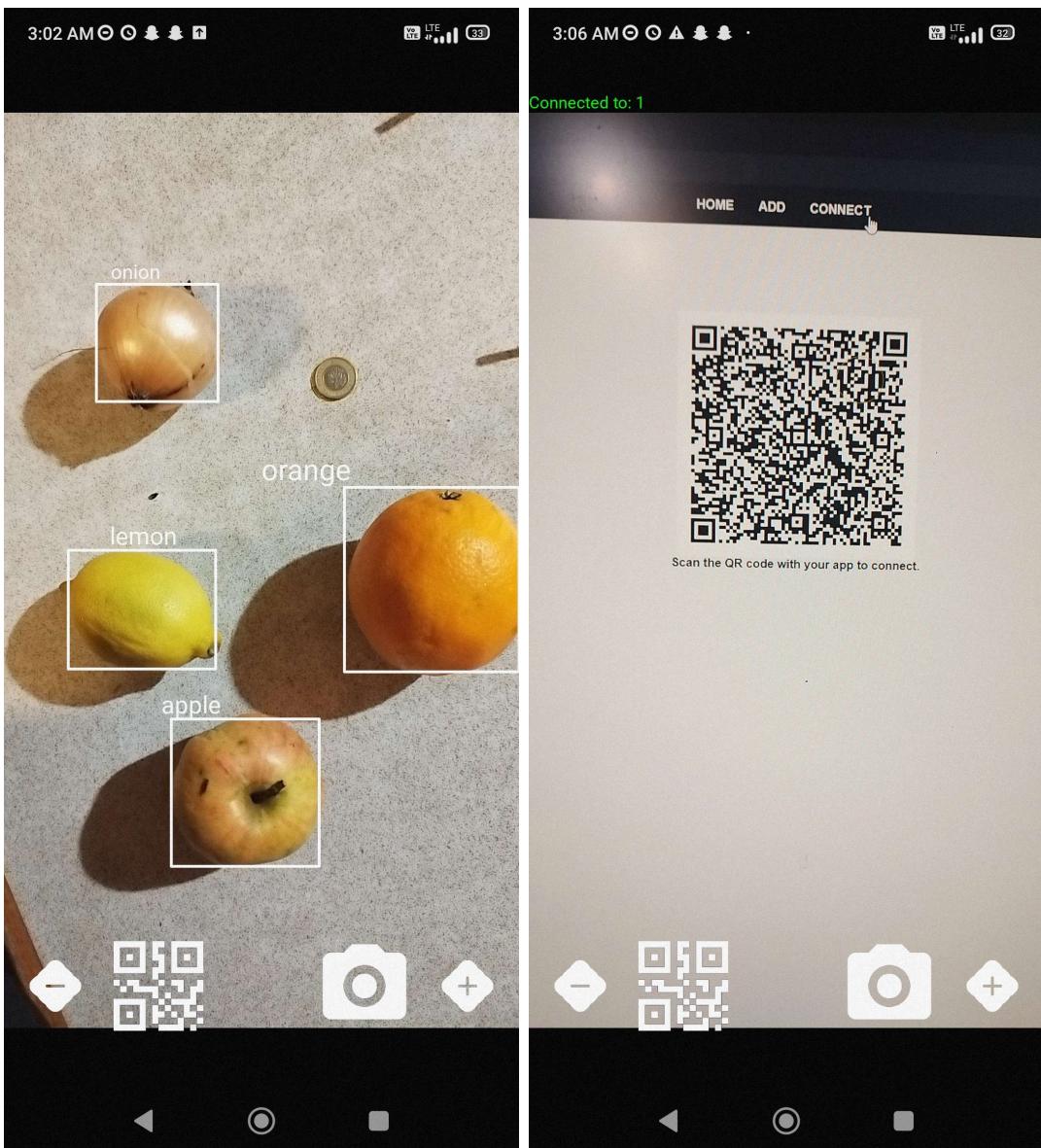
- Pomiar jest bardziej precyzyjny niż się spodziewałem. W niektórych przypadkach wynik wyszedł bardzo bliski rzeczywistemu.
- Aplikacja najlepiej mierzy cytryny i jabłka, a najgorzej papryki i cebule.
- Aplikacja wydaje się lepiej badać produkty o kształcie bardziej podłużnym. Słaby wynik przy papryce może być spowodowany wyjątkowo nieregularnym kształtem warzywa.
- Wynik wyszedł bardziej dokładny przy pomiarze grupowym. Być może było to spowodowane tym, że lepiej ustawiłem kamerę mając do dyspozycji więcej punktów odniesienia.

4. PODSUMOWANIE

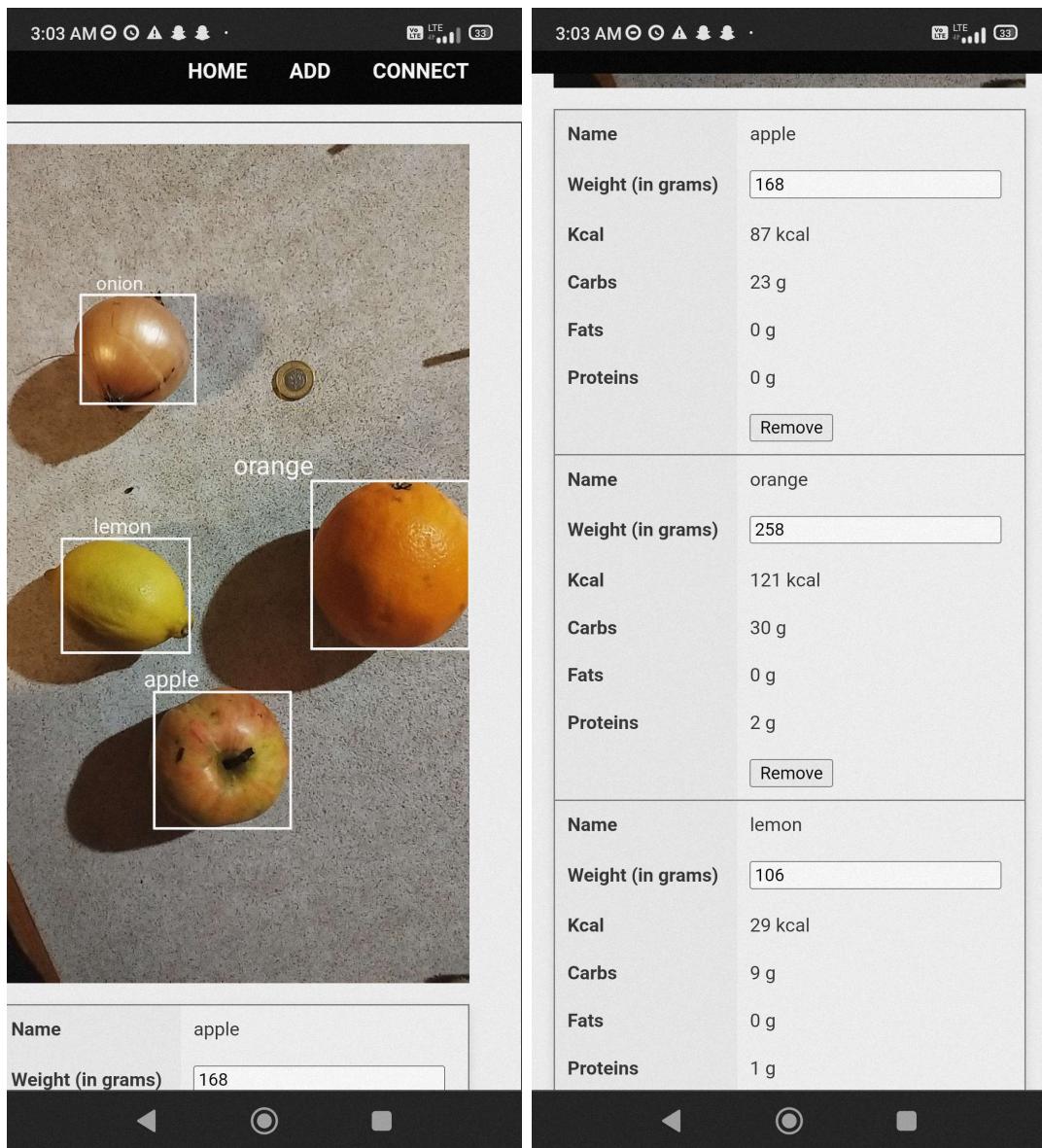
Podsumowując, aplikacja sprawuje się całkiem dobrze. Podczas testów nie spotkałem się z sytuacją, w której nie byłem w stanie dodać jakiegoś produktu przez błędne działanie sieci neuronowej. Zdarza się, że myli ona klasy, najczęściej wykrywa cytryny jako pomarańcze (czego można było się spodziewać), jabłka jako cebulę i vice versa, ale lekka korekta pozycji telefonu lub produktu pozwala na ominięcie błędu. Pomiar wagi to loteria, raz działa bardzo dobrze, raz bardzo źle, ale fakt że zdarza mu się dokładnie zmierzyć wagę produktu z dwuwymiarowego zdjęcia jest warty odnotowania. Część wizualna aplikacji jest prowizoryczna, ale w mojej ocenie intuicyjna dla użytkownika. W celu rozwoju aplikacji można by do niej dodać następujące funkcje:

- Więcej klas obiektów. Aplikacja potrafiąca rozpoznać jedynie 5 produktów nie jest zbyt użyteczna.
- Powiększenie zbioru danych treningowych oraz większe zróżnicowanie poprzez na przykład dodanie zdjęć z innych lokalizacji, o innych porach dnia, wykonanych innym telefonem.
- Opcjonalny jawny system logowania, w celu zapisu danych i przenoszenia ich między przeglądarkami.
- Opcjonalny precyzyjny tryb badania wagi obiektów dla telefonów wyposażonych w dalmierz. Posiadając sposób na badanie rozmiaru obiektu bez punktu odniesienia byłoby również możliwe stworzenie jego modelu w 3D na przykład poprzez nagranie wideo.
- Tryb dodawania produktów w sposób klasyczny, czyli z predefiniowanej listy.
- Podsumowanie spożytych kalorii, miesięczne raporty, porady żywieniowe.
- Stworzenie web view w aplikacji, w którym można by wyświetlać bilans aby nie otwierać zewnętrznej przeglądarki.
- Dodanie większej ilości wspieranych monet do mierzenia niż moneta 2 zł. Wytreningowanie dodatkowej sieci do detekcji monet na podglądzie.
- Dodanie obsługi dla innych języków niż angielski.
- Stworzenie portu na systemy IOS.

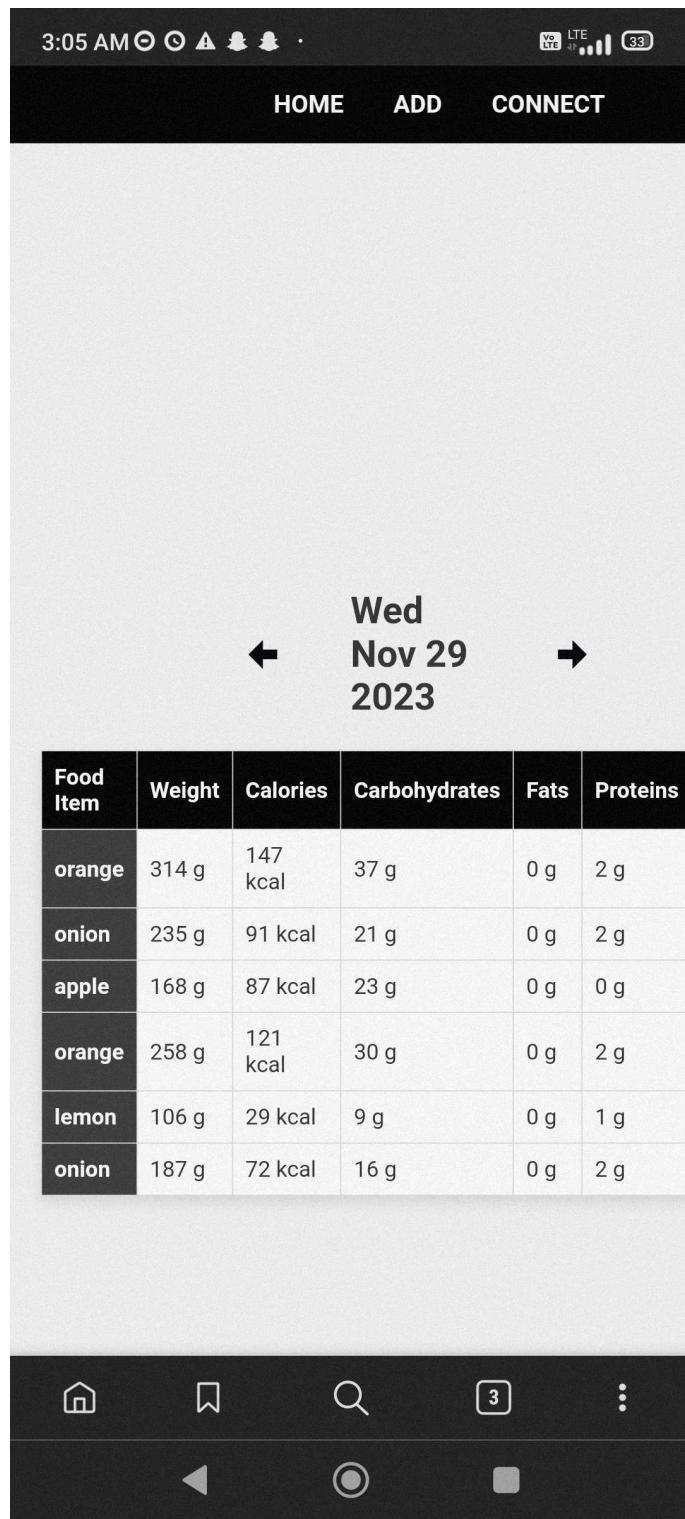
Część z nich planowałem zatrzymać już w obecnej wersji, ale zabrakło mi czasu. Do pracy dołączyłem kod źródłowy aplikacji i strony, został również udostępniony w repozytorium na githubie ([34]). Na sam koniec umieszczam kilka screenów prezentujących działanie aplikacji:



RYSUNEK 17. Detekcja produktów i łączenie aplikacji ze stroną na komputerze



RYSUNEK 18. Wygląd ekranu dodawania produktów



RYSUNEK 19. Tabela bilansu na dany dzień

ŹRÓDŁA Z LINKAMI NA DZIEŃ 29.11.2023

- [1] Weiqing Min et. al, Large Scale Visual Food Recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence
- [2] Oficjalna definicja sztucznej inteligencji ze strony Parlamentu Europejskiego
[https://www.europarl.europa.eu/news/pl/headlines/society/20200827ST085804/sztuczna-inteligencja-co-to-jest-i-jakie-ma-zastosowania#:~:text=Sztuczna%20inteligencja%20to%20\(SI\)%20zdolno%C5%9B%C4%87,uczenie%20si%C4%99%C2%20planowanie%20i%20kreatywno%C5%9B%C4%87.](https://www.europarl.europa.eu/news/pl/headlines/society/20200827ST085804/sztuczna-inteligencja-co-to-jest-i-jakie-ma-zastosowania#:~:text=Sztuczna%20inteligencja%20to%20(SI)%20zdolno%C5%9B%C4%87,uczenie%20si%C4%99%C2%20planowanie%20i%20kreatywno%C5%9B%C4%87.)
- [3] The limits of machine intelligence
<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6776890/>
- [4] Subana Shanmuganathan: "Artificial Neural Network Modelling: An Introduction"
https://link.springer.com/chapter/10.1007/978-3-319-28495-8_1
- [5] DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars
<https://dl.acm.org/doi/pdf/10.1145/3180155.3180220>
- [6] Aplikacja dietetyczna Fitatu
<https://www.fitatu.com/>
- [7] Biblioteka do uczenia maszynowego TensorFlow
<https://www.tensorflow.org/?hl=pl>
- [8] Framework do tworzenia aplikacji mobilnych w pythonie Kivy
<https://kivy.org/>
- [9] Framework do korzystania z klas Javy w pythonie.
<https://pyjnius.readthedocs.io/en/latest/>
- [10] Web framework Flask
<https://flask.palletsprojects.com/en/3.0.x/>
- [11] Model DALL-E
<https://openai.com/research/dall-e>
- [12] Model midjourney
<https://www.midjourney.com/home?callbackUrl=%2Fexplore>
- [13] Model Stable Diffusion
<https://github.com/CompVis/stable-diffusion>
- [14] Strona ElevenLabs do generowania głosów
<https://elevenlabs.io/>
- [15] Model turtle tts
<https://github.com/neonbjb/tortoise-tts>
- [16] Tacotron 2 + WaveGlow do generowania głosów
https://pytorch.org/hub/nvidia_deeplearningexamples_tacotron2/
- [17] ChatGPT
<https://chat.openai.com/>
- [18] Google Bard
<https://bard.google.com/>
- [19] The Perceptron: A Model for Brain Functioning <https://journals.aps.org/rmp/abstract/10.1103/RevModPhys.34.123>
- [20] <https://www.youtube.com/watch?v=bfmFfD2RICg>
- [21] V. Zocca, G. Spacagna, D. Slater, P. Roelants: "Deep Learning - Uczenie głębokie z językiem Python." 2017
- [22] Twierdzenie o pochodnych cząstkowych funkcji złożonej
<https://edu.pjwstk.edu.pl/wyklady/am/scb/main105.html>

- [23] Warstwy konwolucyjne
<https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaaff>
- [24] Warstwy grupujące
<https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/>
- [25] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.- Y. Fu, and A. C. Berg. "Ssd: Single shot multibox detector."
<https://arxiv.org/pdf/1512.02325.pdf>
- [26] The Architecture and Implementation of VGG-16
<https://pub.towardsai.net/the-architecture-and-implementation-of-vgg-16-b050e5a5920b>
- [27] Uniwersalne twierdzenie o aproksymacji
https://cognitivemedium.com/magic_paper/assets/Hornik.pdf
- [28] Program LabelIMG
<https://github.com/qaprosoft/labelImg>
- [29] Model Sequential <https://keras.io/api/models/sequential/>
- [30] Repozytorium z gotowymi modelami TensorFlow
https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md
- [31] Poradnik instalacji modelu z tensorflow model zoo, Part 1-3
<https://www.youtube.com/playlist?list=PL0aoTDj9Nwghdp04hgPPSC8pSzg0kyCXS>
- [32] Algorytm kodowania base64
<https://developer.mozilla.org/en-US/docs/Glossary/Base64>
- [33] RFC 7519 JSON Web Token (JWT) <https://datatracker.ietf.org/doc/html/rfc7519>
- [34] Repozytorium projektu
https://github.com/filipzygmuntowicz/neural_network_based_diet_app