



WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI
I INFORMATYKI

Imię i nazwisko studenta: Dawid Trzciński

Nr albumu: 192236

Poziom kształcenia: Studia drugiego stopnia

Forma studiów: stacjonarne

Kierunek studiów: Automatyka, cybernetyka i robotyka

Specjalność: Robotyka i systemy decyzyjne

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Segmentacja instancyjna i lokalizacja obiektów zapakowanych

Tytuł pracy w języku angielskim: Instance segmentation and localization of packed objects

Opiekun pracy: dr inż. Marek Tatara

OŚWIADCZENIE dotyczące pracy dyplomowej zatytułowanej: Segmentacja instancyjna i lokalizacja obiektów zapakowanych

Imię i nazwisko studenta: Dawid Trzciński
Data i miejsce urodzenia: 12.09.1998, Wejherowo
Nr albumu: 192236

Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki

Kierunek: automatyka, cybernetyka i robotyka

Poziom kształcenia: drugi

Forma studiów: stacjonarne

Typ pracy: praca dyplomowa magisterska

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2019 r. poz. 1231, z późn. zm.) i konsekwencji dyscyplinarnych określonych w ustawie z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (t.j. Dz. U. z 2020 r. poz. 85, z późn. zm.),¹ a także odpowiedzialności cywilnoprawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

06.08.2023, Dawid Trzciński

Data i podpis lub uwierzytelnienie w portalu uczelnianym Moja PG

**) Dokument został sporządzony w systemie teleinformatycznym, na podstawie §15 ust. 3b Rozporządzenia MNiSW z dnia 12 maja 2020 r. zmieniającego rozporządzenie w sprawie studiów (Dz.U. z 2020 r. poz. 853). Nie wymaga podpisu ani stempla.*

¹ Ustawa z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce:

Art. 312. ust. 3. W przypadku podejrzenia popełnienia przez studenta czynu, o którym mowa w art. 287 ust. 2 pkt 1–5, rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.

Art. 312. ust. 4. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 5, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o podejrzeniu popełnienia przestępstwa.

STRESZCZENIE

Praca pod tytułem "Segmentacja instancyjna i lokalizacja obiektów zapakowanych" pokrywa obszerną tematyką, w głównej mierze zajmującą się zagadnieniami związanymi z wykryciu obiektów w obrazie oraz przełożeniu tych informacji na odnalezienie obiektu w przestrzeni. Zadania związane z detekcją, jak i segmentacją są coraz popularniejsze, nie tylko poprzez zwiększenie autonomiczności w środowisku człowieka, ale również dzięki coraz dokładniejszym algorytmom pozwalającym na wspomnianą autonomiczność.

W poniższej pracy przedstawiono potrzebną wiedzę do rozpoczęcia projektowania własnego rozwiązania związanego z problematyką wizyjną. Przedstawione zostały dostępne rozwiązania do detekcji, jak i segmentacji obiektów w obrazie. Przedstawione zostały zbiory danych nadające się do wstępniego przetestowania opracowanego modelu. Na potrzeby pracy stworzony został zbiór danych składający się z obiektów zapakowanych w postaci kartonów. Opisano wyniki dla stworzonego modelu do detekcji obiektów, przetestowane zostały dostępne modele do segmentacji oraz wybrany został jeden z najlepszymi rezultatami. Wykonano dodatkowe testy sprawdzające wydajność w momencie połączenia modelu detekcji z modelem segmentacji. Przeanalizowano dostępne możliwości lokalizacji obiektów w przestrzeni, wybrany został wariant najkorzystniejszy pod względem dostępnych zasobów. Specjalnie do tego celu stworzony został system wizyjny składający się z dwóch kamer, umożliwiający działanie algorytmu do estymacji odległości.

Przeprowadzone zostały testy wykorzystując do tego celu zbiór danych z obiekta zapakowanymi. Po zweryfikowaniu działania modeli na zbiorze danych przystąpiono do testów z wykorzystaniem stereowizji oraz kartonów. Testy przeprowadzonu w celu weryfikacji poprawnego działania modeli oraz algorytmu estymacji odległości, sprzężonych ze sobą.

Słowa kluczowe: głębokie sieci neuronowe, detekcja obiektów, segmentacja instancyjna, lokalizacja w przestrzeni, sieci konwolucyjne, transformery, stereowizja

Dziedzina nauki i techniki zgodna z OECD Nauki inżynieryjne i techniczne, Elektrotechnika, elektronika i inżynieria informatyczna, Robotyka i Automatyka

ABSTRACT

The thesis titled "Instance Segmentation and Localization of Packaged Objects" covers a comprehensive range of topics primarily focused on object detection in images and the translation of this information into locating the object in space. Tasks related to detection and segmentation are becoming increasingly popular, not only due to their potential for enhancing autonomy in human environments but also thanks to increasingly accurate algorithms that enable this autonomy.

In this thesis, the necessary knowledge is provided to initiate the design of one's own solution related to visual issues. Available solutions for object detection and segmentation in images are presented, along with datasets suitable for preliminary testing of the developed model. A dataset consisting of packaged objects in the form of cardboard boxes was created specifically for this thesis. Results for the developed object detection model are described, available segmentation models were tested, and one with the best results was selected. Additional tests were conducted to assess the performance when combining the detection model with the segmentation model. Available options for object localization in space were analyzed, and the most resource-efficient variant was chosen. A vision system, comprising two cameras enabling the operation of a distance estimation algorithm, was created specifically for this purpose.

Tests were conducted using the dataset containing packaged objects. After verifying the models' performance on the dataset, tests were conducted using stereo vision and cardboard objects. These tests were performed to validate the correct operation of the models and the distance estimation algorithm when integrated with each other.

Keywords: deep neural network, object detection, instance segmentation, localization in reality, convolutional neural network, transformers, stereovision

OECD consistent field of science and technology classification: Engineering and technology, Electrical engineering, Electronic engineering, Information engineering, Robotics and automatic control

SPIS TREŚCI

SPIS TREŚCI	6
1 WSTĘP I CEL PRACY	9
1.1 Cel pracy	9
1.2 Układ pracy	9
2 WSTĘP TEORETYCZNY ORAZ PRZEGŁĄD LITERATURY	11
2.1 Głębokie uczenie maszynowe	11
2.1.1 Konwolucyjne sieci neuronowe - CNN	11
2.1.2 Warstwa konwolucyjna	13
2.1.3 Metody optymalizacyjne	16
2.1.4 Funkcje aktywacyjne	19
2.1.5 Transformery	22
2.1.6 <i>Vision in Transformer - ViT</i>	24
2.2 Segmentacja obrazu	26
2.2.1 Rodzaje segmentacji	26
2.2.2 Algorytm segmentacji obrazu	27
2.3 Przetwarzanie obrazu	29
2.3.1 Zniekształcenia w obrazie	30
2.3.2 Parametry kamer	31
3 PRZYGOTOWANIE ŚRODOWISKA	33
3.1 System operacyjny	33
3.2 Język programowania	33
3.3 Biblioteka sieci neuronowych	34
4 IMPLEMENTACJA I EWALUACJA	37
4.1 Zbiór danych	37
4.1.1 Gromadzenie danych	38
4.1.2 Etykietowanie danych	38
4.2 Projektowanie modelu	40
4.2.1 Model do klasyfikacji	40
4.2.2 Model do detekcji	47
4.2.3 Model do segmentacji	53
4.3 Lokalizacja w przestrzeni	55
4.4 Ewaluacja algorytmów	62
5 PODSUMOWANIE	64
SPIS RYSUNKÓW	66
SPIS TABEL	67

DODATEK A INSTRUKCJA DLA UŻYTKOWNIKA	73
A.1 Wymagania systemowe	73
A.2 Język programowania oraz biblioteki	73
A.3 Wykonywanie zdjęć do kalibracji	73
A.4 Uruchomienie głównego skryptu	73

LISTA SYMBOLI

- 2D - układ współrzędnych w przestrzeni dwuwymiarowej
- 3D - układ współrzędnych w przestrzeni trójwymiarowej
- c_x - operator poziomego przesunięcia centra optycznego
- c_y - operator pionowego przesunięcia centra optycznego
- f - długość ogniskowej
- P_w - punkt w rzeczywistym układzie współrzędnych
- X_i - koordynaty układu współrzędnych obrazu
- X_w - koordynaty w rzeczywistym układzie współrzędnych
- Y_i - koordynaty układu współrzędnych obrazu
- Y_w - koordynaty w rzeczywistym układzie współrzędnych
- Z_w - koordynaty w rzeczywistym układzie współrzędnych

LISTA SKRÓTÓW

- BERT - model dla języka naturalnego (ang. *Bidirectional Encoder Representations from Transformers*)
- CNN - konwolucyjne sieci neuronowe (ang. *Convolutional Neural Network*)
- CUDA - zunifikowana architektura do obliczeń równoległych (ang. *Compute Unified Device Architecture*)
- CPU - procesor (ang. *Central Processing Unit*)
- FCN - sieć w pełni konwolucyjna (ang. *Fully Convolutional Networks*)
- FPS - klatki na sekundę (ang. *Frame per seconds*)
- GAN - generatywna sieć przeciwnostawna (ang. *Generative Adversarial Network*)
- GPU - procesor graficzny wykorzystywany w kartach graficznych (ang. *Graphics Processing Unit*)
- ICLR - międzynarodowe konferencje związane ze sztuczną inteligencją (ang. *International Conference on Learning Representations*)
- LSTM - sieć neuronowa do analizy sekwencji danych (ang. *Long Short-Term Memory*)
- MAE - maskowane autoenkodery to skalowalne, samonadzorujące się moduły uczące dla wizji komputerowej (ang. *Masked Autoencoder*)
- Mask R-CNN - jeden z nowocześniejszych modeli do segmentacji instancji i segmentacji obrazu, opracowany na bazie konwolucyjnej sieci neuronowej opartej na regionach (ang. *Mask Region Convolutional Neural Network*)
- MNIST database - zbiór danych odręcznie pisanych liczb (ang. *Modified National Institute of Standards and Technology database*)
- NLP - przetwarzanie języka naturalnego (ang. *Natural language processing*)
- NMS - algorytm eliminujący powtarzające się wyniki (ang. *Non-maximum suppression - NMS*)
- PSPNet - (ang. *Pyramid Scene Parsing Network*)
- px - pixel (ang. *Picture Element*), najmniejszy element obrazu cyfrowego
- RNN - rekurencyjne sieci neuronowe (ang. *Recurrent Neural Network*)
- SSD - model detekcji obiektów w obrazie (ang. *Single Shot Detectro*)
- TPU - akcelerator sztucznej inteligencji opracowany przez Google (ang. *Tensor Processing Unit*)
- VGG - skrót grupy badawczej (ang. *Visual Geometry Group*)
- ViT - architektura przystosowana do problemów wizji komputerowej (ang. *Vision Transformer*)
- VOC2012 - (ang. *Visual Object Classes* - VOC zbiór danych stworzony przez grupę (ang. *Pattern Analysis, Statistical Modelling and Computational Learning - PASCAL*))
- YOLO - architektura detekcji obiektów (ang. *You Only Look Once*)

1. WSTĘP I CEL PRACY

Współczesne technologie wizyjne odgrywają kluczową rolę w różnorodnych dziedzinach życia, począwszy od przemysłu po medycynę i samochody autonomiczne. Jednym z ważnych zastosowań przetwarzania obrazu jest segmentacja instancyjna i lokalizacja obiektów, która umożliwia identyfikację, śledzenie i analizę pojedynczych obiektów w obrazach i sekwencjach wideo. Zagadnienia te stanowią istotne wyzwanie zarówno dla naukowców, jak i inżynierów w dziedzinach związanych z przetwarzaniem obrazu, robotyką, samochodami autonomicznymi, analizą zachowań ludzkich oraz wieloma innymi dziedzinami. Można również odnotować coraz to większy postęp w wydajności podzespołów komputerowych, które to stanowią podstawowe narzędzie związane z projektowaniem głębokich sieci neuronowych. Na rynku pojawia się więcej startupów oferujących rozwiązania związane z wizją komputerową wykorzystując moc obliczeniową w chmurze. Coraz więcej samochodów osobowych posiada system wielu kamer do wspomagania kierowcy oraz opcję autopilota. W ostatnich latach przedstawiono modele pozwalające na bardzo szybką detekcję obiektów, co w połączeniu z modelami do segmentacji na stosunkowo krótki czas reakcji. Udostępniane są coraz to większe zbiory danych, które posiadają *ground truth* (prawidłowe oznaczenia) bardzo istotne w momencie trenowania i testowania własnych modeli. Nowe modele pozwalają na analizę obrazu o coraz to większej rozdzielczości, dzięki czemu można bardziej szczegółowo podzielić obiekty w obrazie na jeszcze mniejsze części.

1.1. Cel pracy

Celem pracy jest opracowanie modelu pozwalającego na segmentację instancyjną obiektów zapakowanych wraz z ich lokalizacją w przestrzeni trójwymiarowej. W ramach pracy należało opracować zbiór danych na podstawie obrazu z wielu kamer, a następnie wyuczyć model i zintegrować dane z różnych kamer. W ramach prowadzonego badania zostaną zrealizowane następujące zadania:

- Przegląd literatury
- Przyjęcie założeń projektowych
- Opracowanie zbioru danych
- Wytrenowanie modelu
- Implementacja lokalizacji obiektów
- Testy i wyniki

1.2. Układ pracy

Praca składa się z pięciu rozdziałów, w których zostały omówione zagadnienia związane z opracowanym tematem oraz rezultaty i wnioski na temat eksperymentów zrealizowanych w ramach wybranego zadania.

Rozdział 2 zawiera wstęp teoretyczny wraz z przeglądem literatury. Przedstawiony został zarys historyczny konwulcyjnych sieci neuronowych, rozbudowane informacje potrzebne do projektowania własnych modeli oraz przedstawione zostały nowe rozwiązania zarówno w detekcji, jak i segmentacji obiektów.

Rozdział 3 zawiera istotne informacje w doborze języka programowania, systemu operacyjnego oraz odpowiednich bibliotek. Zapoznanie się z tym rozdziałem pozwoli na uniknięcie użycia nie aktualnych narzędzi do pracy z sieciami neuronowymi.

Rozdział 4 zawiera opis zbierania potrzebnych danych do utworzenia zbioru treningowego oraz testowego. Znajdują się również informację o tworzeniu modelu do detekcji oraz segmentacji sposób ich testowania oraz usuwania powstałych problemów. Omówione zostało tworzenie systemu wizyjnego potrzebnego do zlokalizowania obiektów w przestrzeni trójwymiarowej. Na końcu połączono wszystkie rozwiązania i przedstawiono wynik pracy.

Rozdział 5 zawiera podsumowanie przeprowadzonego badania, w którym przedstawione zostały najważniejsze punkty całego eksperymentu.

2. WSTĘP TEORETYCZNY ORAZ PRZEGŁĄD LITERATURY

Przed przeprowadzeniem badań związanych z tematem pracy magisterskiej, zapoznano się z literaturą dotyczącą klasyfikacji, segmentacji instancyjnej oraz lokalizacji obiektów w przestrzeni. Przetwarzanie obrazów, w tym przypadku segmentacja obiektów jest dziedziną prężnie rozwijającą się. Jest to związane między innymi z wykorzystaniem tej techniki w branży *automotive*, w której coraz częściej wykorzystuje się kamery do wspomagania kierowców na drogach. Z tego też powodu pojawiła się potrzeba przetwarzania danych z kamer pojazdu w sposób optymalny. Z pozyskanych danych można sklasyfikować przedstawione obiekty, dzięki temu pojazd zaczyna bardziej "rozumieć" otaczający go świat, umożliwiając przy tym podjęcie jak najlepszej decyzji podczas jazdy. Warto również przyjrzeć się segmentacji obiektów na stercie. Jest to szczególnie przydatne dla robotów, które mają za zadanie umieścić np. przesyłki na przenośniku taśmowym, czy też na paletę. W przypadku obiektów zapakowanych, ważnym elementem jest nie tylko ich odpowiednia klasyfikacji wraz z nałożeniem maski segmentacyjnej, ale również poznanie ich wielkości oraz lokalizacji. Stosując odpowiednie równania oraz techniki, można określić potrzebne dane. Poznanie tych danych pozwoli stworzyć algorytm podnoszenia obiektów w odpowiedniej kolejności, dzięki któremu możliwe będzie wysterowanie robota.

2.1. Głębokie uczenie maszynowe

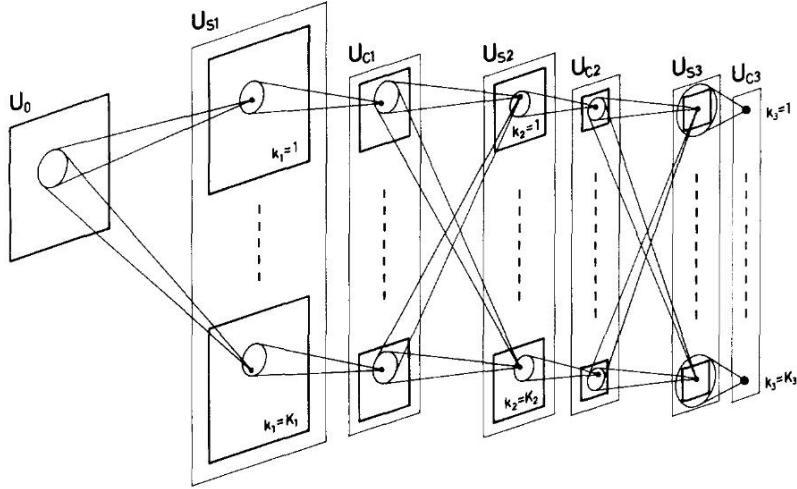
Głębokie uczenie maszynowe jest podzbiorem uczenia maszynowego, które pozwala na trening sztucznej sieci neuronowej wykorzystując do tego duże ilości danych. Wynikiem takiego treningu jest model uczenia głębokiego, który po przeszkoleniu jest w stanie przetwarzać nowe dane. Głębokie uczenie maszynowe może bazować na nieustrukturyzowanych danych w nieprzetworzonej formie (np. obrazach), a następnie określenie ich cech. Może również wykorzystywać zestawy danych z etykietami, w takiej sytuacji mówimy o tzw. uczeniu nadzorowanym. W odróżnieniu od uczenia maszynowego, w większości przebiega ono bez potrzeby bezpośredniego nadzoru człowieka.

2.1.1. Konwolucyjne sieci neuronowe - CNN

Konwolucyjne sieci neuronowe (ang. *Convolutional Neural Networks - CNN*) są klasą sztucznej inteligencji, głównie wykorzystywane do analizy obrazów. Pierwsze wzmianki można znaleźć w artykułach z lat 80-tych ubiegłego wieku. Kunihiko Fukushima w 1980 roku przedstawił "neokognitron" – była to hierarchiczna wielowarstwowa sztuczna sieć neuronowa [1]. Inspiracją do stworzenia takiej sieci neuronowej był artykuł z 1959 roku autorstwa David'a Hubel'a oraz Torsten'a Wiesel'a [2]. Podłączyli oni mikroelektrody do mózgu kota i pokazywali zwierzęciu obrazki, obracającego się podłużnego prostokąta. Podczas eksperymentu odkryli, że na każdy obrazek reagowała wybrana grupa neuronów, w przypadku wyświetlania pionowej kreski aktywna była jedna grupa neuronów, natomiast przy wyświetaniu poziomej kreski aktywna była druga grupa neuronów. Dzięki swoim badaniom odkryli oni dwa typy komórek w pierwotnej korze wzrokowej zwane komórkami prostymi oraz komórkami złożonymi. Zaproponowali tym samym kaskadowy model tych dwóch typów komórek do wykorzystania w rozpoznawaniu wzorców. Warto również wspomnieć, że za swoje odkrycie otrzymali w 1981 r. nagrode Nobla z medycyny [2].

Fukushima wprowadził dwa podstawowe typy warstw w sieciach CNN: warstwy splotowe (nazywana również warstwą konwolucyjną) i warstwy próbkowania. Ogólną ideą jest uchwycenie koncepcji "od prostego do złożonego" i przekształcenie jej w model obliczeniowy do wizualnego rozpoznawa-

nia wzorców. W artykule podzielono je na warstwy proste oraz złożone, które reprezentowane są za pomocą macierzy dwuwymiarowych. Ze względu na taką reprezentację danych autor przedstawił konsepcję diagramu pokazanego na rysunku 2.1 Każdy czworokąt narysowany grubymi liniami reprezentuje płaszczyznę prostą (ang. *S-plane*) lub płaszczyznę złożoną (ang. *C-plane*), zaś czworokąt pionowy narysowany cienkimi liniami, w których zawarte są *S-plane* oraz *C-plane* reprezentują warstwę prostą "US" (ang. *S-layer*) oraz warstwę złożoną "UC" (ang. *C-layer*) [1].



Rysunek 2.1: Diagram prezentujący połączenia między warstwami w necognitron [1]

Dzięki modelowi, który przedstawił Fukushima, naukowcy rozpoczęli badania nad wykorzystaniem konwolucyjnych sieci neuronowych. W 1998 roku Yann LeCun wraz z wspólnikami przedstawili artykuł naukowy, w którym zademonstrowali model CNN przystosowany do rozpoznawania liczb w piśmie odręcznym [3]. Wykorzystali do tego celu zbiór danych (ang. *MNIST database*), który zawiera ponad 60 tysięcy zdjęć treningowych oraz 10 tysięcy zdjęć testowych. Mimo upływu lat ten sam zbiór jest dalej wykorzystywany przez osoby zaczynające swoją naukę z algorytmami do przetwarzania obrazów. Autorzy zwracają uwagę na wielkość zdjęć (rozdzielcość), która wpływa na ilość parametrów spowalniając przez to system, na którym uruchomiony został algorytm. Trzeba również pamiętać, że prace wykonano w 1998 roku, gdzie możliwości komputerów oraz technologii były dosyć mocno ograniczone.

Konwolucyjne sieci neuronowe, jak i cała dziedzina głębokiego uczenia była rozwijana przez kolejne lata, w różnych dziedzinach rozpoznawania wzorców - od przetwarzania obrazów po rozpoznanie głosu. Badacze skupiali się między innymi na metodach zmniejszenia liczby wag parametrów oraz osiąganiu coraz to większej dokładności. Przykładowo, jeżeli badany obraz miał 24 piksele szerokości na 24 piksele wysokości oraz był on w kolorach *RGB*, to liczba wag parametrów dla 10 ukrytych neuronów wynosi 17280. W przypadku przetworzenia całego obrazu należy utworzyć siatkę ukrytych neuronów w pełni połączonych (ang. *fully connected*), przez co liczba wag parametrów wzrasta aż do 995328, liczba ta wynika z konieczności połączenia wejścia z węzłami neuronów. Jednym z rozwiązań przedstawionym przez autorów publikacji jest skupienie się na lokalnych obszarach obrazów niż jego cała interpretacja. Takie rozwiązanie sprawia, że kolejne warstwy ukrytych neuronów otrzymują pomniejszoną ilość danych wejściowych, powiązanych z poprzednią warstwą. Jeden z prostszych modeli wykorzystujący konwolucyjne sieci neuronowe jest model TinyVGG. Przedstawiony w 2017r. przez Xing Fang'a, w celu rozpoznawania emocji u ludzi, z różnych perspektyw [4, 5]. Taki model składa się z warstwy konwolucyjnej, "poolingowej", spłaszczającej oraz funkcji aktywacyjnej. Warto również wspomnieć, w jaki sposób reprezentowane są dane wsadowe. Sieci konwolucyjne w ramach rozwiązywania proble-

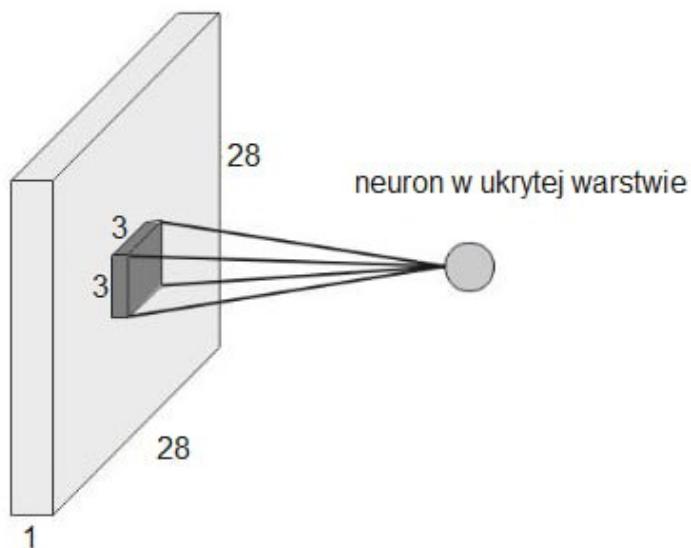
mów wizyjnych wykorzystują tensorzy jako dane wsadowe. Tensor można określić jako wielowymiarową macierz, która posiada takie informacje o obrazie jak szerokość, wysokość oraz głębokość. Szerokość, jak i wysokość definiowana jest poprzez ilość pikseli (rozdzielcość), natomiast głębokość w przypadku danych wsadowych oznacza liczbę kanałów koloru, która może wynosić 3 lub 1 w zależności czy mowa o obrazie RGB czy też obrazie czarno-białym [6, 7].

2.1.2. Warstwa konwolucyjna

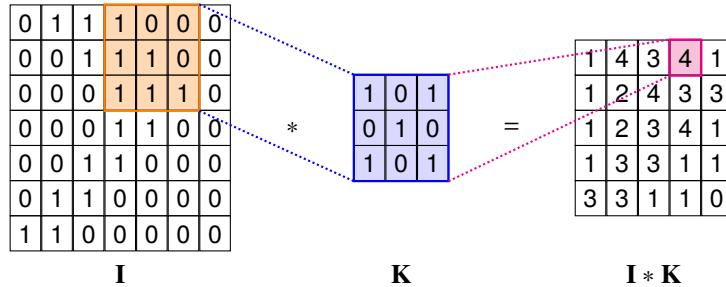
Warstwa ta posiada kluczowy element zwany jądrem bądź filtrem, który pozwala na zredukowanie wielkości danych wsadowych bądź też pozostawienie takiego samego rozmiaru danych wykorzystując wartości hiperparametrów. Hiperparametry służą do określenia, w jaki sposób ma zachować się warstwa. Można wyróżnić trzy najważniejsze hiperparametry:

1. Rozmiar jądra/filtru (ang. *Kernel size*)
2. Wypełnienie (ang. *Padding*)
3. Krok (ang. *Stride*)

Rozmiar jądra odnosi się do wymiaru okna przesuwnego przechodzącego przez dane wejściowe. Wybór wielkości filtra ma bardzo duży wpływ na ilość neuronów w kolejnych warstwach ukrytych, co przekłada się na ilość informacji w lokalnym obszarze. Im większa wielkość filtru, tym mniej informacji przekazujemy dalej, natomiast w przeciwnym wypadku zbieramy większą ilość szczegółów. Dobór wielkości jądra jest kwestią rozpatrywanego problemu, lecz można bardzo często spotkać go o wymiarach 3x3. Na rysunku 2.2 przedstawiono w sposób uproszczony filtr o wymiarach 3x3 przechodzący przez tensor o wymiarach 28x28x1, w dalszej części zostanie zaprezentowany przykład ze zbioru Fashion MNIST o właśnie takich wymiarach [8].



Rysunek 2.2: Zastosowanie filtru na tensorze trzeciego rzędu



Rysunek 2.3: Splot macierzy dwuwymiarowej I stosując filtr K

Rysunek 2.3 przedstawia zasadę działania warstwy konwolucyjnej z filtrem 3×3 oraz krokiem równym 1. W celu uproszczenia wizualnego został przedstawiony przekrój w jednym z wymiarów tensora trzeciego rzędu w postaci macierzy 7×7 . Na pomarańczowym tle I przedstawiony został obszar, dla którego następuje konwolucja z jądrem K . Na różowym tle przedstawiono wynik działania. Operację splotu macierzowego można zapisać według następującego wzoru:

$$(I * K)_{xy} = \sum_{i=1}^h \sum_{j=1}^w K_{ij} \cdot I_{x+i-1, y+j-1} \quad (2.1)$$

gdzie:

I - macierz dwuwymiarowa,

K - filtr dwuwymiarowych,

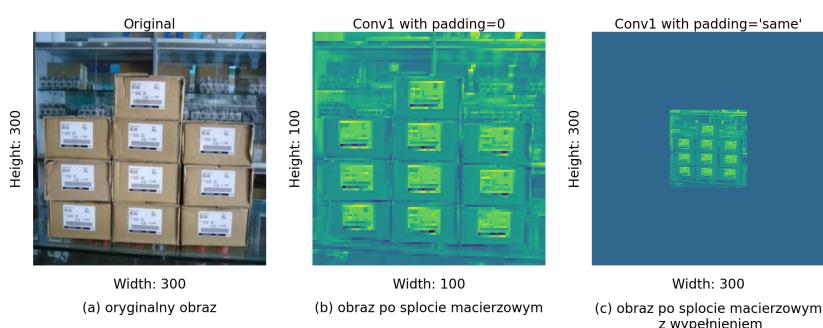
h - wysokość filtra,

w - szerokość filtra,

* - operacja splotu.

W rzeczywistości do wzoru (2.1) należy uwzględnić jeszcze głębię tensora, która zmienia wynik działania. Oznacza to, że filtr przechodzi przez każdą kolejną macierz znajdująca się w tensorze, a następnie poszczególne wyniki są sumowane i zwracane do nowo powstałego tensora.

Wypełnienie jest trzecim wspomnianym wcześniej hiperparametrem, który to należy uwzględnić przy projektowaniu warstwy konwolucyjnej. Służy on do rozszerzenia oryginalnej danej wejściowej z zachowaniem głębi. Dzięki czemu wysokość oraz szerokość tensora po działaniu konwolucji może być taka sam jak wielkość wejściowa lub większa. Najczęściej w miejscu rozszerzonych komórek dopisywane są zera, które nie wpływają na wyciąganie mapy cech, a jedynie utrzymują wskazaną wielkość wyjściową z zerowymi wartościami wokół powstałej mapy.



Rysunek 2.4: Porównanie wielkości obrazu z wypełnieniem oraz bez

Do wykonania rysunku 2.4 wykorzystano warstwę konwolucyjną z wielkością jądra 3x3 oraz wartością kroku 3. Rozszerzone komórki uzupełniono zerami, przez co obszar wokół obrazu jest jednolity. W celu poznania wielkości wyjściowej danych z wykorzystaniem wypełnienia należy posłużyć się (2.2). Ważne jest, aby kontrolować wielkość danych na każdym etapie projektowania modelu. Jest to bardzo ważne w przypadku zastosowania warstwy liniowej.

$$O = \left\lceil \frac{I + 2P - K}{S} \right\rceil + 1 \quad (2.2)$$

gdzie:

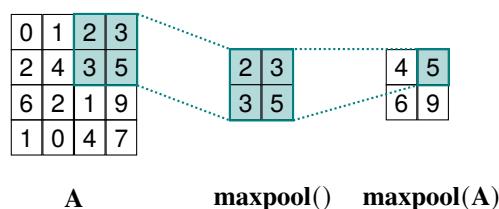
I - wielkość wejścia,

K - wielkość filtra,

P - wielkość wypełnienia,

S - wartość kroku przesuwnego filtra.

Wspomniana wcześniej metoda okna przesuwnego wykorzystuje warstwę gromadzącą (ang. *pooling layer*). Redukując tym samym wielkość badanego obszaru do pewnych rozmiarów, które uzależnione są od wartości parametru kroku, z jakim to okno będzie się przemieszczać po oryginalnym obrazie. Warstwa ta operuje na każdej mapie aktywacji na wejściu i skaluje jej wymiarowość za pomocą funkcji "MAX". Taka warstwa nosi nazwę (ang. *max-pooling layer*) [9].



Rysunek 2.5: Działanie funkcji *max-pooling*

Rysunek 2.5 przedstawia sposób działania wspomnianej funkcji. Jak można zauważycy funkcja ta wy ciąga największą wartość na wybranym obszarze, dlatego tak ważnym elementem jest dobór odpowiednich wymiarów takiego obszaru. Najczęściej w sieciach konwolucyjnych spotyka się filtry o wymiarze 2x2 z krokiem wynoszącym 2 wzdłuż przestrzennych wymiarów wejścia (tak jak na rysunku 2.5). Stosując równanie (2.3) można określić wymiary danych wyjściowych po przeprowadzonej operacji [10, 11].

$$\text{Maxpool}(A) = \left\lceil \frac{A_x - P}{S} \right\rceil + 1 \quad (2.3)$$

gdzie:

A_x - szerokość/wysokość danej wsadowej,

P - wielkość filtra,

S - krok filtra.

W następnych latach rozwijane były zagadnienia związane z konwolucyjnymi sieciami neuronowymi, pojawiły się architektury jak na przykład VGG-Net [12]. Składała się ona ze stosu warstw konwolucyjnych w połączeniu z *max-pooling* oraz warstwami liniowymi. Ostatnią warstwą była funkcja aktywacyjna *softmax*. Funkcja ta służy do oceny przynależności do każdej klasy. Oznacza to, że wartości zwieracane mieszczą się w przedziale [0,1] i są to wartości prawdopodobieństwa przynależności do danej klasy [13].

2.1.3. Metody optymalizacyjne

W sieciach neuronowych istnieje funkcja kosztów, która informuje jak słaba jest wydajność modelu w chwili obecnej. Informacja o stratach jest potrzebna, aby sieć mogła polepszyć swoją wydajność podczas treningu. Należy minimalizować funkcję kosztów, ponieważ im mniejsza wartość, tym lepsze wyniki prezentuje model, taka operacja matematyczna nazywa się optymalizacją. Podczas projektowania modelu sieci neuronowej nie można zapomnieć o odpowiednim doborze metody, która będzie szukała globalnego minimum i jednocześnie zmniejszała funkcję kosztów. Dzięki temu model będzie aktualizować wagi oraz wartości odchylenia. Jedną z popularniejszych metod optymalizacji jest spadek wzdłuż gradientu przedstawiony w (2.4). Ta metoda pozwala na poprawę jakości modelu w kolejnych iteracjach, minimalizując funkcję kosztów w zależności od kierunku gradientu (wybierając przeciwny kierunek niż wskazuje gradient), licząc na znalezienie globalnego minimum. Jednak aby dokonać odpowiedniej minimalizacji, trzeba sprostać pewnym problemom takim, jak odpowiedni dobór kroku uczenia, które gradienty są istotne oraz jak poprawnie zainicjalizować wartości początkowe [14].

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} F(\theta_t) \quad (2.4)$$

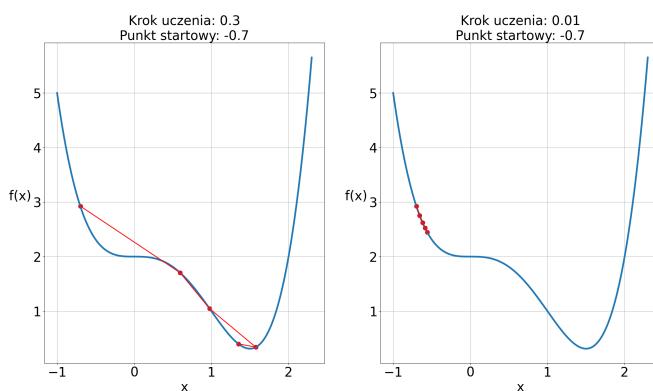
gdzie:

η - krok uczenia (ang. *learning rate*),

∇_{θ_t} - kierunek spadku,

$F(\theta_t)$ - funkcja wypukła, którą należy optymalizować.

Pierwszym problemem, któremu należy sprostać jest dobór odpowiedniego kroku uczenia. Jest to pierwsza przyczyna w momencie, w którym funkcja kosztów modelu ustabilizowała się na pewnej wartości, oznaczającą niską dokładność. Wartość takiego kroku można w zasadzie dobierać metodą prób i błędów, lecz będzie to bardzo nieefektywna i czasochłonna metoda. Rysunek 2.6 przedstawia porównanie optymalizacji funkcji $f(x) = x^4 - 2x^3 + 2$ dla dwóch różnych wartości kroku uczącego. Można zauważyć, że funkcja posiada dwa minima lokalne, co przy złym doborze wartości kroku uczącego może doprowadzić do zatrzymania optymalizacji przy pierwszym minimum. Wykres z prawej strony prezentuje właśnie taką sytuację, gdzie po pięciu iteracjach wartość optymalna wynosi około 2.5, natomiast wykres po lewej stronie osiągnął już praktycznie minimum globalne.



Rysunek 2.6: Prównanie działania optymalizacji funkcji z użyciem dwóch różnych wartości kroku uczenia

Jednym z rozwiązań dotyczącym wyboru odpowiedniego kroku uczącego jest jego adaptacyjna zmiana. Oznacza to zmianę jego wartości podczas treningu po pewnej liczbie epok lub można zastosować (2.5), który uwzględnia: wartość aktualnej epoki, zainicjowaną wartość kroku uczącego oraz rozkład współczynnika uczenia tzw. *decay* [15, 16].

$$lr = I_{lr} * \frac{1}{(1 + D * E)} \quad (2.5)$$

gdzie:

I_{lr} - zainicjalizowana wartość współczynnika uczenia,

D - rozkład współczynnika uczenia,

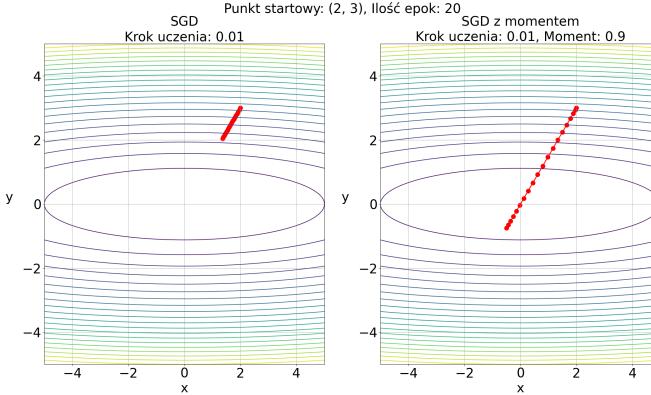
E - aktualna liczba epoki.

W standardowej metodzie spadku wzduż gradientu wymagane jest przetworzenie całego zbioru danych w każdym etapie optymalizacji. Wyliczany jest błąd dla każdego przykładu z osobna, a następnie liczona jest średnia z tych wartości. Niestety w większości przypadków nie będzie możliwe umieszczenie całego zbioru danych do pamięci karty graficznej. W związku z tym zbiór danych jest dzielony na mniejsze paczki, które to zmieszczą się w dostępnej pamięci sprzętu. Ta metoda wprowadza jednak pewne zaszumienia, ponieważ prawdziwy gradient jest aproksymowany. Operacja aproksymacji gradientu wprowadza przez próbę pewną losowość, więc tę metodę nazywa się stochastycznym spadem wzduż gradientu (ang. *Stochastic Gradient Descent - SGD*). Pomimo wprowadzonych zaszumień, co może wydawać się problematyczne w procesie uczenia, to jak się jednak okazuje szum ten pełni funkcję regularyzacyjną i pozwala osiągnąć lepsze rezultaty dla zbiorów testowych [14].

Inicjalizacja wag w sztucznej sieci neuronowej jest niezwykle istotna. Można dobierać odpowiednio parametry samodzielnie, dzięki czemu można uniknąć wielu problemów optymalizacyjnych. Takie rozwiązanie jest bardzo nieefektywne, ponieważ modele w dzisiejszych czasach posiadają miliony parametrów. Odpowiednia inicjalizacja wpływa na początkowy postęp w uczeniu się modelu. Przykładem złej inicjalizacji może być wykorzystanie w warstwie ukrytej funkcji aktywacyjnej *ReLU*. W związku z tym, że funkcja ucina ujemne argumenty i zwraca wartość 0, to w przypadku inicjalizacji parametrów sieci wartościami ujemnymi pojawi się problem z tzw. "martwymi neuronami". Oznacza to, że do neuronu nie będą dopływały żadne informacje, ponieważ wartości będą zerowe. Problem odpowiedniego doboru parametrów startowych jest o tyle istotny, że wpływa na proces uczenia się w propagacji w przód oraz w tył. Wprowadzenie zera jako wartości początkowej w każdej warstwie może spowodować naučenie się tych samych cech przez neurony. Wprowadzenie zbyt małych wartości może spowodować bardzo długi czas treningowy (gradient może zaniknąć), natomiast wprowadzenie zbyt dużych wartości może powodować rozbieżność wyniku (gradient może eksplodować). Jedna z sprawdzonych metod inicjalizacji zmiennych jest wprowadzenie losowego doboru wartości z rozkładu normalnego. Aby zapobiec eksplozji, jak i zanikania gradientu, średnia wartość aktywacyjna powinna być równa zero oraz wariancja w każdej warstwie powinna być taka sama [17].

Przez ostatnie lata naukowcy przedstawili wiele metod optymalizacji, dlatego też warto wyróżnić kilka najpopularniejszych i przedstawić ich wady i zalety. Najczęściej w modelach można spotkać metody optymalizacyjne takie jak *SGD*, *AdaGrad*, *AdaDelta* oraz *Adam*. Pierwsza z metod została wstępnie opisana wyżej, natomiast powstały dwa rozszerzenia tej metody, a mianowicie z wykorzystaniem mini paczek oraz z wykorzystaniem dodatkowego parametru "momentu". *Mini-SGD* polega na podzieleniu zbioru danych na mniejsze części w postaci paczek danych, a następnie na każdej paczce danych wykonanie standardowej operacji metody *SGD*. Po wyliczeniu wartości z danej paczki wyciąga się średnią wartość, którą to wykorzystuje się do aktualizacji całej funkcji. Ta metoda osiąga szybciej zbieżność niż

standardowy algorytm *SGD*. Wadą takiego rozwiązania jest ryzyko utknięcia w lokalnym minimum oraz aktualizacja funkcji wprowadza dodatkowe zaszumienie po każdej paczce [18, 19]. Drugie rozszerzenie dodaje nowy parametr do obliczeń, jakim jest moment. Główną wadą poprzedniej metody jest powstawanie dodatkowego szumu, dodanie momentu sprawia, że gradient nie jest zaszumiony, a więc usprawnia działanie optymalizacyjne. Przyspiesza zbieganie w odpowiednim kierunku, jednocześnie zmniejszając wahania w kierunkach nieistotnych. Rysunek 2.7 pokazuje optymalizację funkcji $f(x, y) = x^2 * y^2 + 20$ używając klasycznego *SGD* oraz z dodanym momentem o wartości 0.9. Wartość ta jest popularnym wyborem, dlatego została ona wykorzystana w owym porównaniu. Jak można zauważać klasyczne rozwiązanie bardzo powoli dąży do minimum, natomiast wykorzystując opcję z momentem osiągnięcie punktu minimum zajmuje mniej epok. To co można zauważać to przestrzeniowanie punktu minimum, jest to spowodowane dodaniem momentu. Jego działanie można porównać do zepchnięcia kuli ze wzgórza, która to akumuluje moment turlając się z góry coraz to szybciej zwiększać swoją prędkość. Równanie (2.6) prezentuje zapis matematyczny metody optymalizacji z momentem. Metoda ta posiada takie same zalety jak zwykłe *SGD*, a na dodatek tego zbiega szybciej do wartości minimum niż klasyczny spadek gradientu. Wadą, którą można dostrzec w tym rozwiązaniu jest dodatkowa zmienna do obliczania przy każdej aktualizacji [20, 21].



Rysunek 2.7: Porównanie działania standardowego *SGD* oraz z dodanym momentem

$$\begin{aligned} V_t &= \beta V_{t-1} + \alpha \nabla_w L(W, X, y) \\ W &= W - V_t \end{aligned} \tag{2.6}$$

gdzie:

β - moment,

α - krok uczenia,

L - funkcja kosztów.

Jednym z najnowszych algorytmów optymalizacyjnych jest *Adam*. Został on po raz pierwszy opublikowany w 2014 roku i zaprezentowany na przeszłej konferencji ICLR w 2015 roku [22]. W artykule przedstawiono bardzo obiecujące wykresy prezentujące znaczącą poprawę w szybkości treningu sieci neuronowych. Przedstawiony algorytm wykorzystuje w pełni zasadę pędu (momentu), która to została opisana w pewnym stopniu w poprzednim podrozdziale. Ta zasada opisuje właściwie tylko połowę mechanizmu działania optymalizatora *Adam*. W podejściu wykorzystującym zasadę pędu, normalizowany jest pierwszy moment procesu optymalizacyjnego. Twórcy algorytmu idą o krok dalej i normalizują drugi moment, czyli wariancję co prowadzi do wydajniejszej optymalizacji.

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta L \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_\theta L)^2 \\
\theta_{t+1} &= \theta_t - \eta \frac{m_t}{\sqrt{v_t}}
\end{aligned} \tag{2.7}$$

gdzie:

$m_t \in \mathbb{R}^N$ - reprezentuje pierwszy moment i $m_t = 0$,

$v_t \in \mathbb{R}^N$ - reprezentuje drugi moment i $v_t = 0$,

β_1 oraz β_2 - hiperparametry ważne, jak istotne mają być poprzednie kroki w porównaniu do aktualnego.

Równanie (2.7) przedstawia idee zastosowania drugiego momentu, najczęściej można spotkać użycie wartości $\beta_1 = 0.9$ oraz $\beta_2 = 0.999$. Opisany powyżej wzór posiada pewną wadę, która to została naprawiona w algorytmie *Adam*. Mianowicie w początkowej fazie uczenia, czyli dla małych wartości t , średnie kroczące m_t oraz v_t będą silnie związane ze swoją wartością zainicjalizowaną równą 0. Warto zwrócić uwagę na to, że małe wartości v_t w mianowniku równania mogą powodować bardzo duże kroki, co może doprowadzić do destabilizacji uczenia. W celu zapobiegania takim sytuacjom autorzy algorytmu *Adam* wprowadzili korekty, zaprezentowane w następującym wzorze:

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta L \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_\theta L)^2 \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned} \tag{2.8}$$

gdzie:

\hat{m}_t - skorygowany pierwszy moment,

\hat{v}_t - skorygowany drugi moment,

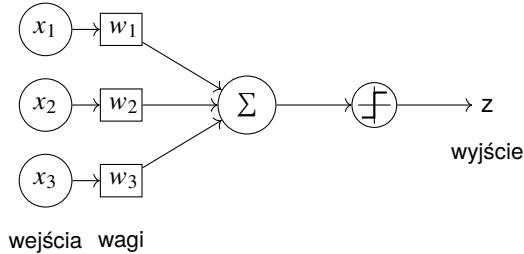
ϵ - bardzo mała stała (na przykład 10^{-12}) w celu uniknięcia dzielenia przez 0.

Przytoczone algorytmy optymalizacyjne są jednymi z najczęściej spotykanych w modelach głębokiego uczenia. Posiadają kilka swoich odmian w zależności od problematyki z jaką mają się zmagać, przykładowo można spotkać *AdamW* jako modyfikację klasycznego *Adam* w problematyce przetwarzania języka naturalnego (ang. *Natural Language Processing - NLP*). Pomimo bardzo dobrze przedstawionych aspektów teoretycznych oraz przytoczonych prostych obliczeniowo równań, to wybór odpowiedniego optymalizatora pozostaje w ujęciu praktycznym i doświadczalnym. Oznacza to, że nie jest łatwo dobrać odpowiedni algorytm do rozwiązywania danego problemu i jedną z lepszych metod ich doboru jest przetestowaniu kilku z nich z różnymi hiperparametrami.

2.1.4. Funkcje aktywacyjne

Tworząc model sieci neuronowej istotny jest odpowiedni dobór tzw. funkcji aktywacyjnych. Służą one do połączenia neuronów między warstwami oraz wprowadzają nieliniowość. Jest możliwe pominięcie użycia funkcji aktywacyjnych, lecz w takim wypadku powstałoby połączenie liniowe. Takie połączenie w prawdzie jest możliwe, ale nie jest specjalnie przydatne, zarówno w warstwie ukrytej jak i wyjściowej. W złożonych problemach taka sieć neuronowa nie poradziłaby sobie ze znalezieniem optymalnego

rozwiązań (o ile do jakiegoś rozwiązania by doszła). Tak naprawdę suma funkcji liniowych zawsze będzie dawać jednak funkcję liniową. Taki model będzie bardziej zbliżony do modelu regresji liniowej. Rysunek 2.8 przedstawia przykładowe połączenia z wagami do pojedynczego neurona wraz z funkcją aktywacyjną. Natomiast równanie (2.9) przedstawia sposób obliczania wartości wyjściowej wszystkich połączeń neuronowych poprzedniej warstwy dla neurona kolejnej warstwy [23].



Rysunek 2.8: Uproszczony schemat perceptronu

$$z = F \left(\sum_i w_i x_i \right) \quad (2.9)$$

gdzie:

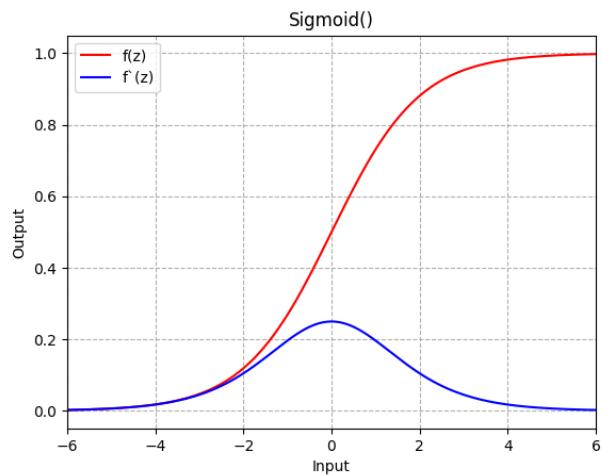
w_i - waga dla danego połączenia,

x_i - dana wejściowa neurona poprzedniej warstwy,

F - funkcja aktywacyjna.

Do rozwiązywania bardziej złożonych problemów niż klasyfikacja binarna, należy zastosować funkcję nieliniową. Na przestrzeni lat powstało kilka funkcji aktywacyjnych, które stosowane są do rozbudowanych modeli. Dobór odpowiedniej funkcji transferowej do danej problematyki jest kluczowy w celu znalezienia najlepszego rozwiązania. Istotny jest nie tylko przebieg takiej funkcji w propagacji w przód, ale również pochodna takiej funkcji, która wykorzystywana jest w propagacji wstecznej do aktualizacji wag. Najczęściej spotykanymi funkcjami transferowymi są *Sigmoid*, *Tanh* oraz *ReLU*.

Logistyczna funkcja aktywacyjna nazywana potocznie ang. *Sigmoid* (s-kształtna) z racji na charakterystyczny kształt wykresu. Funkcja ta przyjmuje wartości liczb rzeczywistych (tzw. wartości logitowe) i przekształca je w wartości z przedziału $[0, 1]$. Interpretuje się ją jako prawdopodobieństwo przynależności danej próbki, czyli nic innego jak prognozowanie. Dodatkowo można zauważać, że punkt przecięcia z osią rzędnych następuje w sytuacji, gdy $f(z) = 0.5$. Matematyczny zapis funkcji przedstawia (2.10). Na rysunku 2.9 kolorem czerwonym przedstawiono przykładowy przebieg funkcji. Taki przebieg funkcji wykorzystywany jest do propagacji w przód. Pozwala on wyznaczyć korekcje, jakie należy dokonać podczas aktualizacji wag. Jak już wcześniej wspomniano, ważny jest nie tylko przebieg takiej funkcji w przód, ale również podczas propagacji wstecznej. Oznacza to, że istotna jest pochodna takiej funkcji, której kształt często decyduje o wyborze przy projektowaniu modelu. Na rysunku 2.9 kolorem niebieskim przedstawiono przebieg pochodnej funkcji *Sigmoid*. Jak można zauważać traci ona swoją monotoniczność, przez co sieć neuronowa może generować nieoptimalne rozwiązania. Zazwyczaj wykorzystuje się ją w warstwie wyjściowej przykładowo w binarnej sieci klasyfikacyjnej, rzadziej spotyka się wykorzystanie tej funkcji w warstwach ukrytych.



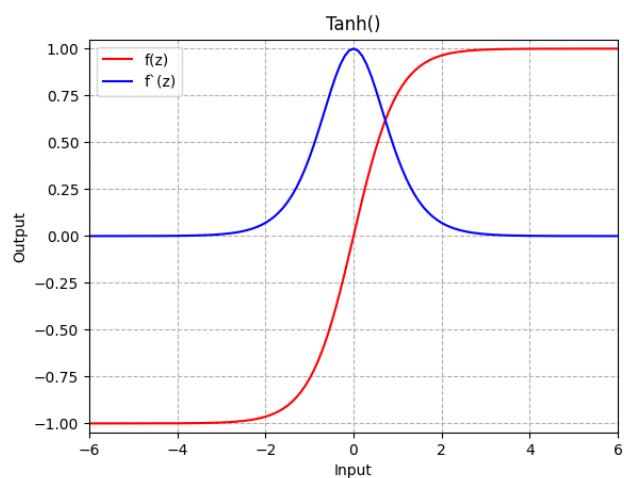
Rysunek 2.9: Przebieg funkcji *sigmoid* oraz jej pochodnej

$$f(z) = \frac{1}{1 + e^{-z}} \quad (2.10)$$

gdzie:

z - dyskretna wartość wejściowa.

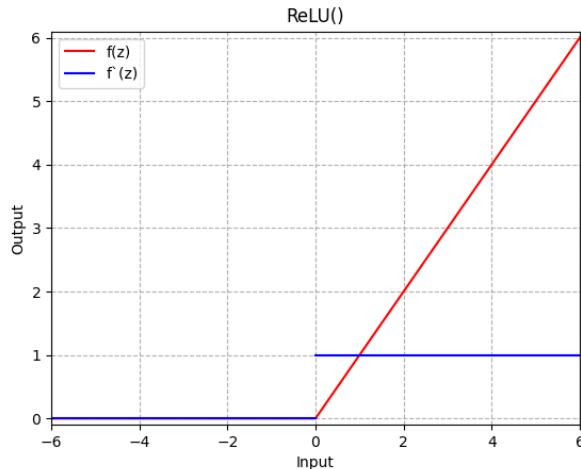
Funkcja tangensa hiperbolicznego tzw. *tanh* jest bardzo podobna do funkcji *sigmoid*, ale usprawnia działanie aktywacji w obszarze z ujemnymi argumentami. W funkcji *sigmoid* ujemne argumenty przyjmowały wartości bliskie zeru, co powodowało spadek wydajności uczenia się takiej sieci. Wartości funkcji *tanh* zawierają się w przedziale [-1,1]. Matematyczny zapis tej funkcji przedstawia (2.11). Na rysunku 2.10 przedstawiono kolorem czerwonym oryginalną funkcję *tanh*, natomiast kolorem niebieskim jej pochodną. Przebieg pochodnej jest bardziej stromy, niż w przypadku funkcji logistycznej. Oznacza to silniejszy gradient dla funkcji tangensa hiperbolicznego. Taką funkcję aktywacyjną można spotkać częściej w warstwach ukrytych niż funkcję *sigmoid*, ale nie jest to zalecane przy dużej ilości warstw ukrytych.



Rysunek 2.10: Przebieg funkcji *tanh* oraz jej pochodnej

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (2.11)$$

Prostowana jednostka liniowa (ang. *Rectified Linear Unit - ReLU*) wprowadza zupełnie nowe spojrzenie na funkcje transferowe w głębszych sieciach neuronowych. Główną zaletą tej funkcji jest eliminacja problemu z zanikającym gradientem, z którym to borykają się dwie poprzednie funkcje. Problem zanikającego gradientu polega na tym, że funkcja taka jak *sigmoid* ogranicza wartości wejściowe do zakresu [0,1]. Podczas wstępnej propagacji następuje wymnożenie wartości gradientu z każdej warstwy aż do warstwy wejściowej, co przy wielowarstwowym modelu może spowodować otrzymanie wartości bliskie zeru. Skutkiem takiego postępowania będzie brak postępów w nauce danego modelu, nazywanym zanikającym gradientem [24]. Dlatego istotny jest przebieg pochodnej funkcji aktywacyjnej. Funkcja *ReLU* pozwala rozwiązać problem zanikającego gradientu, ponieważ pochodna tej funkcji przyjmuje wartość 0 dla argumentów ujemnych oraz wartość 1 dla zera i argumentów dodatnich. Rysunek 2.11 przedstawia kolorem czerwonym przebieg funkcji *ReLU* oraz jej pochodną kolorem niebieskim. Sam wzór reprezentujący funkcję (2.12) nie jest skomplikowany obliczeniowo, co jest dodatkowym atutem.



Rysunek 2.11: Przebieg funkcji *ReLU* oraz jej pochodnej

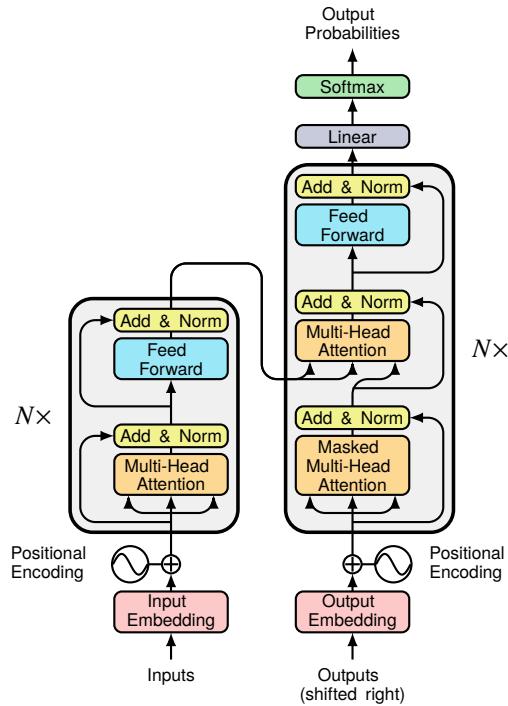
$$f(z) = \max(0, z) \quad (2.12)$$

Warto również wspomnieć o stosunkowo nowej funkcji aktywacyjnej zyskującej popularność, a mianowicie ang. *Gaussian Error Linear Unit - GELU*. Główne zastosowanie nowej funkcji transferowej jest w modelach, które wykorzystują technikę *dropout*. Technika *dropout* polega na wyłączeniu neuronu w danej warstwie ukrytej z pewnym prawdopodobieństwem. Stosowana jest w celu zmniejszenia zjawiska przeuczenia się modelu [25, 26].

2.1.5. Transformery

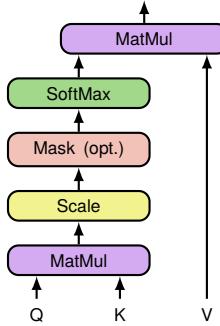
W 2017 roku został opublikowany artykuł zespołu naukowców z Google, zatytułowany "*Attention is all you need*". Przedstawiona została nowa architektura nazwana transformerem, która to miała poprawić rozwiązywanie problemów w NLP. Dotychczas stosowano rekurencyjne sieci neuronowe oraz sieci LSTM do analizy sekwencji danych. Jednakże te sieci nie nadawały się do długich ciągów danych.

Wynika to z faktu, że takie ciągi danych (w postaci zdań) mają pojedynczy ukryty wektor jako dane wyjściowe, natomiast ostatnia część sieci LSTM może nie być w stanie odpowiednio zrozumieć znaczenia całego zdania. Przez co danemu słowi wejściowemu nie jest poświecana odpowiednia uwaga. Dlatego powstała inicjatywa stworzenia nowej architektury, która opiera swoje działanie na tytułowej atencji. Rysunek 2.12 przedstawia ogólny schemat tworzenia modelu opartego na transformerze. Architektura podzielona została na enkoder oraz dekoder, podobnie jak to miało miejsce w sieci LSTM. Można zauważyć poszczególne charakterystyczne bloki dla głębszych sieci neuronowych. Bloki *Input/Output Embedding* mają za zadanie zmapowanie słów z ciągu danych do tzw. przestrzeni osadzeń [27, 28].



Rysunek 2.12: Przykładowa architektura modelu transformera

Blok przekazywania (ang. *Feed-Forward*) składa się z dwóch liniowych transformacji oraz funkcji transferowej *ReLU*. Służy on do ekstrakcji oraz transformacji cech sekwencji, dzięki czemu model jest zdolny do wykrywania zależności, jak i wzorców w danych tekstowych. W enkoderze oraz dekoderze znajduje się najistotniejszy blok opisywanej architektury, a mianowicie *Multi-Head Attention*. Składa się on z kilku równolegle działających warstw atencji, które to potem są łączone ze sobą i przepuszczane przez warstwę liniową. Rysunek 2.13 przedstawia schemat blokowy równania (2.13), który przedstawia przeskalowany wynik atencji. Taki pojedynczy blok nazywany jest głową uwagi (ang. *Head Attention*), jest on wykorzystywany w wielonagłówkowej warstwie atencji. Zapytania (Q), klucze (K) oraz wartości (V) reprezentują trzy różne aspekty sekwencji wejściowej. Są one wykorzystywane do obliczania wag atencji między pozycjami wejściowymi, a pozycją wyjściową. Dzięki czemu model efektywnie uwzględnia istotne informacje z całej sekwencji danych do wygenerowania odpowiedzi.



Rysunek 2.13: Schemat blokowy pojedynczej uwagi

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (2.13)$$

gdzie:

Q - zbiór kwerend spakowanych w macierz,

K - klucze spakowane w macierz,

V - wartości spakowane w macierz,

d_k - wymiarowość kluczy.

2.1.6. Vision in Transformer - ViT

Podczas gdy transformery wprowadziły nowy standard w przetwarzaniu języka naturalnego, to w problematyce wizji komputerowej w dalszym ciągu górowały splotowe sieci neuronowe. Zostały one przez ostatnie lata w znacznym stopniu usprawnione pod względem wydajności oraz złożoności. Ich główna idea pozostała bez zmian przez wiele lat. W momencie wprowadzenia architektury *transformer* w celach przetwarzania języka naturalnego, rozpoczęto jednocześnie badania z wykorzystaniem tejże architektury w problematyce wizyjnej. Próbowano połączyć splotowe sieci neuronowe z mechanizmem samo uwagi, w rozwiązywaniu problemów w detekcji jak i klasyfikacji obiektów. Powstały również modele, które to tworzyły napisy do filmów z opisem wykonywanych czynności [29].

W 2021 opublikowany został artykuł grupy badaczy z Google zatytułowany "An Image is Worth 16x16 Words: Transformers For Image Recognition at Scale". Przedstawia on model do klasyfikacji obrazów z wykorzystaniem ideologii architektury *transformer*. Twórcy projektując model starali się trzymać tak blisko jak to możliwe oryginalnej koncepcji *transformera*. Rysunek 2.14 przedstawia ogólny schemat blokowy działania modelu [30]. Można zauważyć charakterystyczny enkoder z oryginalnej koncepcji *transformera*, w którym znajdują się między innymi wielonagłówkowa warstwa atencji. Składa się on nie tylko z tej warstwy, ale również z warstwy normalizującej oraz wielowarstwowego perceptronu (ang. *Multilayer perceptron - MLP*). Składa się on z dwóch liniowych warstw ukrytych oraz jednej funkcji aktywacyjnej *GELU*. Równania (2.14) przedstawiają fundamentalną ideę przedstawioną na rysunku 2.14.

$$\begin{aligned}
 \mathbf{z}_0 &= [\mathbf{x}_{\text{class}} ; \mathbf{x}_p^1 \mathbf{E}; \mathbf{x}_p^2 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{\text{pos}}, \quad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D} \\
 \mathbf{z}'_\ell &= \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \quad \ell = 1 \dots L \\
 \mathbf{z}_\ell &= \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \quad \ell = 1 \dots L \\
 \mathbf{y} &= \text{LN}(\mathbf{z}_L^0)
 \end{aligned} \quad (2.14)$$

gdzie:

x_p - sekwencja spłaszczonych fragmentów obrazu $x_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$,

P - wielkość fragmentu obrazu,

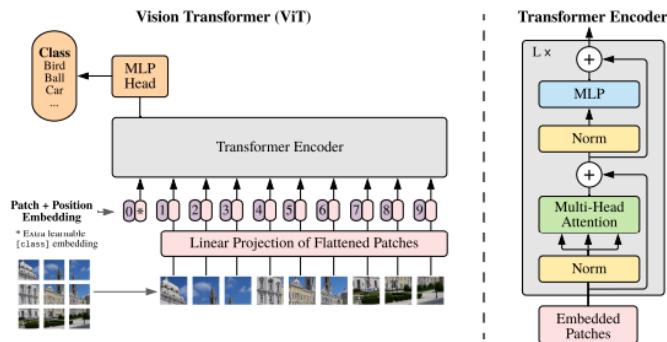
C - liczba kanałów koloru,

N - liczba fragmentów obrazu ($N = H \cdot W / P^2$),

E - osadzanie pozycji fragmentu obrazu,

MSA - wielonagłówkowa warstwa samouwagi,

LN - normalizacja warstwy.



Rysunek 2.14: Model Vision Transformer [30]

Model w pierwszej kolejności dzieli obraz na mniejsze części oraz pozycjonuje je w kolejności rosnącej uwzględniając kolejność wyodrębniania części od lewej do prawej. Utrzymanie odpowiedniej kolejności jest kluczowe w aspekcie mechanizmu uwagi, ponieważ przy złej kolejności przedstawienia podzielnego obrazu może się okazać, że uwagę przyciągnieżą zupełnie inną grupą pikseli. Składowa P ze wzoru (2.14) odpowiada za rozdzielczość jaką ma mieć poszczególny fragment podzielonego obrazu. Jest ona istotnym parametrem, który decyduje jak dobrze poradzi sobie model z wychwytywaniem odpowiednich cech w obrazie. Ważne jest, aby oryginalne wielkości obrazu były podzielne przez P , ponieważ nie może być sytuacji w której fragment obrazu ma mniejszą rozdzielczość niż pozostałe. W przypadku gdy nie zostanie spełniony ten warunek, należy zmniejszyć wartość składowej lub zmniejszyć wielkości oryginalnego obrazu. Alternatywnie zamiast wykorzystywać surowe informacje o obrazie, można wykorzystać mapę cech [31]. Można wykorzystać prosty blok składający się z jednej warstwy splotu i jednej warstwy spłaszczenia, w ten sposób otrzymane informacje tworzą wspomnianą mapę cech gotową do pozycjonowania przed enkoderem. W przypadku dobrania zbyt małej wartości, model może uczyć się bardzo powoli i nieefektywnie. Składowa D jest stałą wartością dla wszystkich warstw i oznacza ona wymiar spłaszczonej paczki podzielonego obrazu. Jest ona wynikową iloczynu powierzchni paczki z ilością kanału koloru. Na przykładzie paczki, której rozdzielczość wynosi 16, a obraz jest kolorowy to wynik będzie wynosił 768 przez pomnożenie $16 \times 16 \times 3$.

Autorzy przedstawili proces eksperymentalny trenowania i testowania takiego modelu. Trening modelu został przeprowadzony na kilku dużych zbiorach danych w celu wstępniego przetrenowania, aby potem móc dotrenować model do mniejszych zbiorów danych. Wykorzystano JFT [29], ImageNet oraz ImageNet-21k [32, 33] jako duże zbioru danych - posiadają one miliony obrazów w dużej rozdzielczości oraz kilkanaście tysięcy klas. Dzięki takiemu zabiegowi model zostaje wstępnie przetrenowany, co skutkuje krótszym czasem treningowym dla docelowego zbioru danych. W takim przypadku stosuje się technikę transferu wiedzy (ang. *transfer learning*), która pozwala na przeniesienie dotychczasowej

wiedzy modelu w postaci zapisanych wag ukrytych warstw, do modelu przygotowanego w celu nauki wzorców z nowego zbioru danych [34, 35]. Należy pamiętać, że takie przeniesienie wiedzy jest możliwe w przypadku, gdy model ma wykonać podobne zadanie (przykładowo klasyfikacja obrazu) wykorzystując nowy zbiór danych. Wykorzystując wcześniej wytrenowany model, został on dotrenowany na mniejszych zbiorach danych. Do tego celu wykorzystano zbiory CIFAR-10/100 (ang. *Canadian Institute for Advanced Research, 10/100 classes*) [36], Oxford-IIIT Pets [37] oraz Oxford Flowers-102 [38]. Zbiory te posiadają znacznie mniejszą ilość danych oraz maksymalnie około 100 klas.

Do eksperymentu zostały utworzone trzy modele o różnych parametrach wewnętrznych nazywanę "Base", "Large" oraz "Huge". Dwa pierwsze posiadają bazę konfiguracyjną na podstawie modelu BERT [39] natomiast twórcy dodali trzeci model największy spośród wszystkich. Główne różnice między nimi można dostrzec w liczbie paramterów, gdzie ten największy posiada ich ponad 632 miliony. Liczba parametrów odzwierciedla wielkość danego modelu, jednocześnie wprowadzając ograniczenia dla maszyn obliczeniowych w postaci wymaganej ilości pamięci.

2.2. Segmentacja obrazu

Segmentacja obrazu stanowi kluczową fazę w analizie i przetwarzaniu obrazów cyfrowych. Jej celem jest podział obrazu na różnorodne obszary zwane segmentami. Dzięki segmentacji możliwe jest wyodrębnienie istotnych informacji z obrazu oraz zrozumienie struktury i zawartości przedstawionej scenerii. Operowanie na segmentach zmniejsza złożoność analizy, ponieważ interpretowane są fragmenty zamiaszt całego obrazu. Działanie algorytmu segmentacji obrazu polega na przypisywaniu etykiet do pikseli w celu identyfikacji znajdujących się obiektów, osób lub innych istotnych elementów. Podział obrazu jest kluczowym elementem technologii i algorytmów widzenia komputerowego. Jest on wykorzystywana w wielu praktycznych zastosowaniach takich jak: analiza obrazów medycznych, pojazdach autonomicznych, rozpoznawanie twarzy oraz analizie obrazów satelitarnych [40].

2.2.1. Rodzaje segmentacji

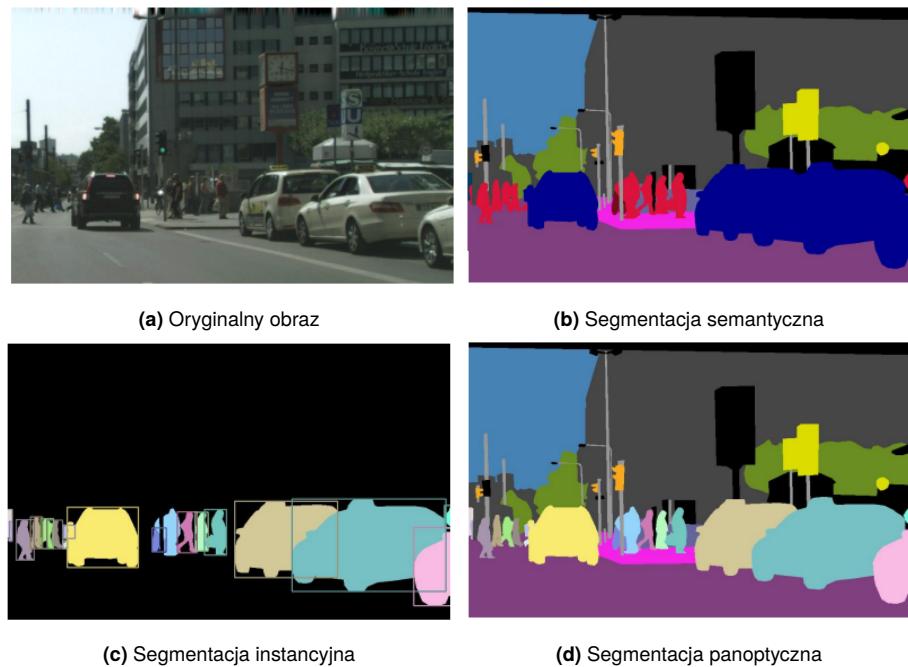
Segmentacja obrazu jest procesem podziału na części określane jako obszary, które są jednorodne. Takimi obszarami są zbiory pikseli, które posiadają wspólne własności rozróżniające je od innych obszarów jak np. barwa lub tekstura. Istnieje wiele różnych technik segmentacji, lecz w tym rozdziale wspomniane zostaną trzy najbardziej popularne - semantyczna, instancyjna oraz panoptyczna.

Segmentacja semantyczna polega na porządkowaniu pikseli na obrazie w oparciu o klasy semantyczne. Oznacza to, że każdy piksel należy do pewnej określonej klasy bez rozróżniania czy to jest któryś z kolei obiekt o tej samej klasie czy też nie. Na przykład segmentacja semantyczna przeprowadzona na obrazie z wieloma drzewami i pojazdami zapewni maskę, która kategoryzuje wszystkie typy drzew do jednej klasy (drzewa) i wszystkie typy pojazdów niezależnie od tego, czy są to autobusy czy też samochody do jednej klasy (pojazdy). Takie podejście powoduje, że problem jest często słabo zdefiniowany, zwłaszcza jeśli istnieje kilka obiektów zgrupowanych w tej samej klasie. Na przykład obraz zatłoczonej ulicy może segmentować cały obszar tłumu jako klasę "ludzie". Segmentacja semantyczna nie zapewnia dogłębnej analizy złożonych obrazów, dlatego też stale były poszukiwane lepsze rozwiązania [41].

Rozwinięciem segmentacji semantycznej i zarazem rozwiązaniem problemu z brakiem rozróżnienia obiektów jednej klasy jest segmentacja instancyjna. Obejmuje klasyfikację pikseli na podstawie instancji obiektu (w przeciwieństwie do klas obiektów). Algorytmy segmentacji instancji nie wiedzą, do

której klasy należy każdy region, zamiast tego oddzielają podobne lub nakładające się regiony na podstawie granic obiektów [42].

W 2019 roku zaprezentowana została segmentacja panoptyczna jako swego rodzaju połączenie dwóch poprzednich rodzajów segmentacji. Przewiduje tożsamość każdego obiektu, segregując każdą instancję każdego obiektu na obrazie. Segmentacja panoptyczna jest przydatna w przypadku wielu produktów, które wymagają ogromnych ilości informacji do wykonywania swoich zadań. Na przykład, samojezdne samochody muszą być w stanie szybko i dokładnie uchwycić i zrozumieć swoje otoczenie. Mogą to osiągnąć, przekazując strumień obrazów na żywo do algorytmu segmentacji panoptycznej [43]. Rysunek 2.15 przedstawia porównanie wizualne wszystkich trzech rodzajów segmentacji.



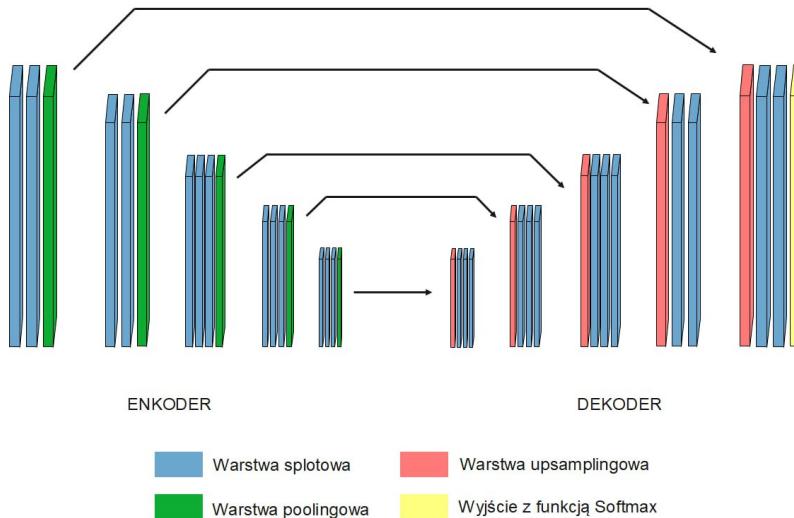
Rysunek 2.15: Porównanie rodzajów segmentacji [43]

2.2.2. Algorytm segmentacji obrazu

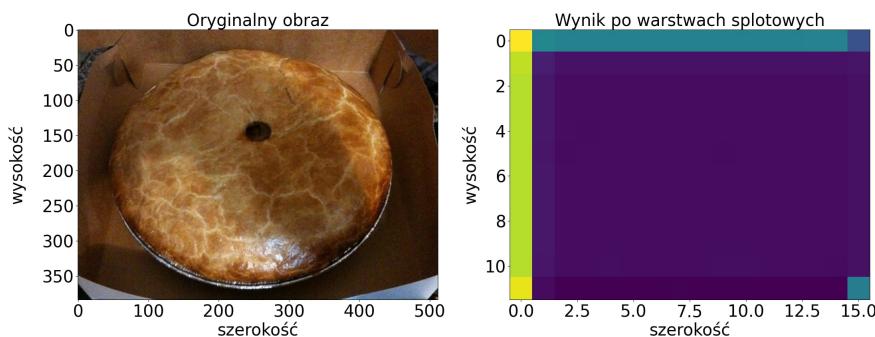
Pierwsze nie analityczne algorytmy wykorzystywały konwolucyjne sieci neuronowe w celu analizy i wprowadzenia ostatecznej mapy segmentacji. Analiza pełnej rozdzielczości obrazu przez całą sieć jest dosyć kosztowna obliczeniowo. Jednym z popularniejszych podejść przy tworzeniu modeli segmentacji obrazu jest wykorzystanie struktury enkodera/dekodera. Rysunek 2.16 przedstawia ogólny schemat struktury konwolucyjnego enkodera/dekodera.

W takiej strukturze najpierw zmniejszana jest rozdzielcość przestrzenna danych wejściowych. Na podstawie informacji w niższej rozdzielczości model jest uczony, aby skutecznie rozróżnić klasy. Problemem jest jednak wyjściowy wymiar obrazu, który po przejściu przez wszystkie warstwy splotowe oraz *max-pooling*, który jest znacznie mniejszy niż oryginalnie. Rysunek 2.17 przedstawia po lewej oryginalny obraz, natomiast po prawej stronie wynik wyjściowy enkodera modelu *VGG-16*. W tym celu używa się tzw. metody *upsamplingu*, jest to nic innego jak użycie techniki interpolacji, którą uzyskuje się przy pomocy warstw dekonwolucyjnych [44].

W 2015 roku przedstawiono *FCN* (ang. *Fully Convolutional Networks*) jeden z pierwszych modeli do segmentacji. Autorzy wykorzystali dotychczasowe rozwiązania sieci neuronowych wykorzysty-



Rysunek 2.16: Schemat działania konwolucyjnego enkodera i dekodera

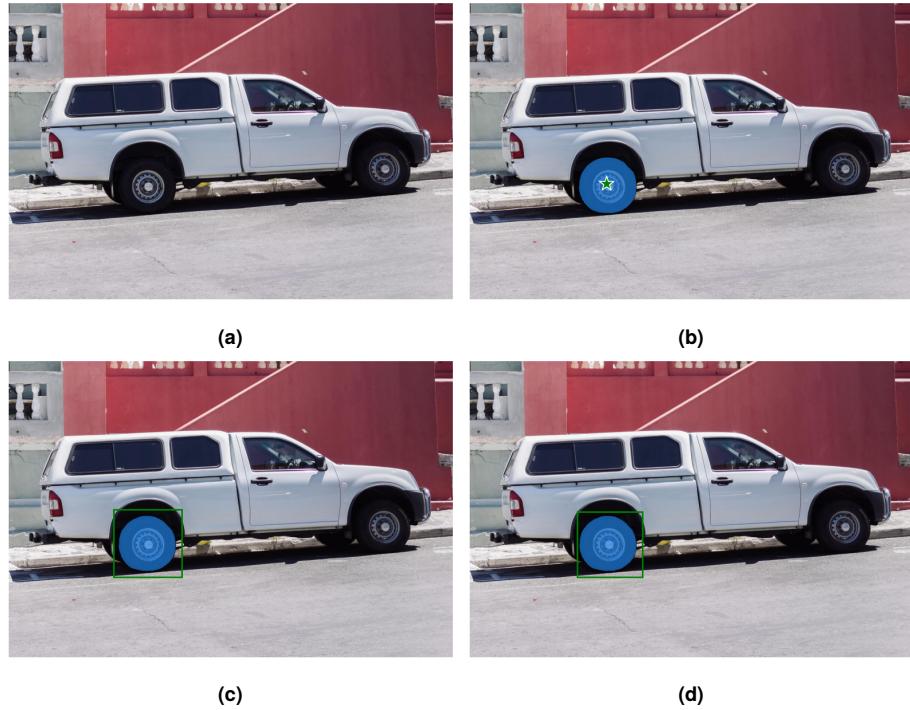


Rysunek 2.17: Porównanie obrazu przed i po wielowarstwowej konwolucji

wanych do klasyfikacji i urozmaicili je do rozwiązywania zadań z segmentacji. Wykorzystali do tego celu trzy bardzo popularne modele - *VGG net* [12], *AlexNet* [36] oraz *GoogLeNet* [45]. Główną cechą prezentowanego modelu jest fakt, że składa się on z warstw konwolucyjnych i nie zawiera warstw w pełni połączonych, które znajdują się w typowej architekturze *CNN* do klasyfikacji. Dzięki temu, że wszystkie warstwy są konwolucyjne to model jest w stanie obsłużyć obraz o różnych rozmiarach bez konieczności zmian architektury. Dodatkowo wprowadzona została koncepcja tzw. połączeń skokowych, które pozwalały na przekazywanie informacji z głębszych warstw do wyższych warstw o większej rozdzielczości. Takie rozwiązanie pozwala modelowi łączyć informacje o lokalnych cechach i globalnych kontekstach, co jest kluczowe w zadaniach segmentacji. W kolejnych latach modele do segmentacji zostały udoskonalane. Pojawiły się takie modele jak - *U-Net* [46], *PSPNet* (ang. *Pyramid Scene Parsing Network*) [47], jedną z ciekawszych oraz bardziej popularnych *MaskR-CNN* (ang. *Mask Region Convolutional Neural Network*) [48] czy też *SAM* (ang. *Segment Anything Model*) oparty na architekturze *transformera* [49].

Warto w tym momencie przyjrzeć się bliżej stosunkowo nowemu modelowi opartego na *transformerze*, a mianowicie *SAM* i jego odmiany. Model ten został zaprezentowany na początku 2023 roku

przez grupę badaczy z *Meta AI Research*. Zaprezentowany został nie tylko nowy model, ale i również największy jak dotąd zbiór danych z oznaczeniami posiadający ponad 1 miliard oznaczonych obiektów na 11 milionach zdjęć. Natomiast model składa się z trzech głównych komponentów - enkodera obrazu, enkodera poleceń oraz dekodera maski. Enkoder obrazu jest najistotniejszym i jednocześnie najpotężniejszym komponentem przedstawionego modelu. Zbudowany jest na *MAE* (ang. *Masked Autoencoder*) [50] oraz wstępnie przetrenowanym *ViT* i generuje jednorazowe osadzenie obrazów tak jak miało to miejsce w opisywanym *ViT* (ang. *Vision Transformer*). Drugim bardzo istotnym komponentem jest enkoder poleceń, który na wejściu może posiadać aż trzy różne dane, na podstawie których będzie realizowane działanie segmentacji. Pierwszą daną jaką można przekazać są punkty, którymi wskazuje się wybór segmentowanych obiektów, można wybrać punkt do segmentacji ale również punkt do separacji. Drugą opcją jest przekazanie koordynatów prostokąta, który obejmuje wybrany obiekt, gdzie wewnątrz niego dokonuje się segmentacja. Możliwe jest połączenie modelu do detekcji obiektów, które to w ostatnich czasach znacznie poprawiły swoje wyniki i przekazać wynik do wejścia modelu do segmentacji. Ostatnią możliwością wyznaczenia obiektu do segmentacji jest przekazanie polecenia w postaci tekstu, który to potem jest odpowiednio interpretowany i w rezultacie otrzymuje się porządkany wynik. W przypadku, w którym nie nie podane zostane jakiekolwiek polecenia do enkodera poleceń, obraz zostanie podzielone automatycznie na segmenty. Rysunek 2.18 przedstawia wyniki segmentacji dla poszczególnych poleceń.



Rysunek 2.18: Wyniki działania modelu *SAM*, (a) przedstawia oryginalny obraz, (b) przedstawia wynik działania z wybranym punktem, (c) przedstawia wynik działania w obszarze prostokąta, (d) przedstawia wynik działania po wprowadzeniu komendy "rear wheel"

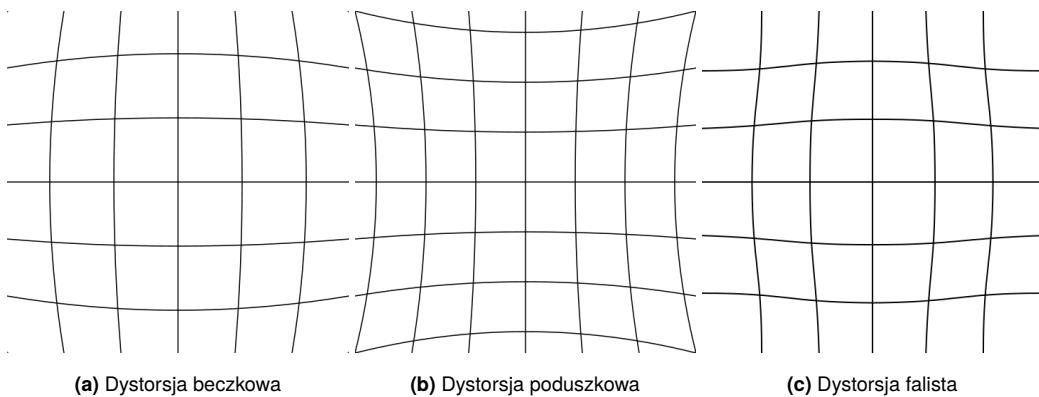
2.3. Przetwarzanie obrazu

Obrazy są jednymi z najczęściej wykorzystywanych danych w modelach głębokiego uczenia. Kamery stanowią rolę czujników do przechwytywania obrazów, pobierają one punkty ze świata rzeczywistego i rzutują je na płaszczyznę dwuwymiarową, którą reprezentuje obraz. Lokalizacja obiektów w obrazie jest

jednym z kroków do otrzymania informacji o położeniu obiektu w świecie rzeczywistym. Istnieje kilka opcji umożliwiających uzyskanie odległości obiektu od kamery, począwszy od umieszczenia w obszarze markera o znanych rozmiarach, aż po gotowe systemy kamer. Niemniej jednak w przypadku korzystania z kamer w domowych warunkach, bardzo ważna jest odpowiednia ich kalibracja w celu zniwelowania zniekształceń.

2.3.1. Zniekształcenia w obrazie

Wykonując zdjęcie prawie zawsze obraz ulega zniekształceniom w porównaniu z rzeczywistością. Zniekształcenia te określane są aberracjami optycznymi, które deformują i wyginają fizyczne proste linie i sprawiają, że wydają się one zakrzywione na obrazie. W uproszczeniu wada ta polega na różnym powiększaniu obrazu w zależności od jego odległości od osi optycznej. Wyróżnić można trzy główne rodzaje: dystorsja beczkowa, poduszkowa oraz falista. Ostatnia z nich jest rzadko spotykana i zazwyczaj świadczy o wadzie układu optycznego. Uzyskanie obrazu bez zniekształceń stosując jedynie obiektywy jest bardzo rzadkie, ponieważ większość obiektywów posiada co najmniej jeden rodzaj zniekształcenia. Bardzo dobrej jakości obiektywy redukują znaczco zniekształcenia przynajmniej w sposób wystarczający, żeby ludzkie oko nie dostrzegło zmian w obrazie.

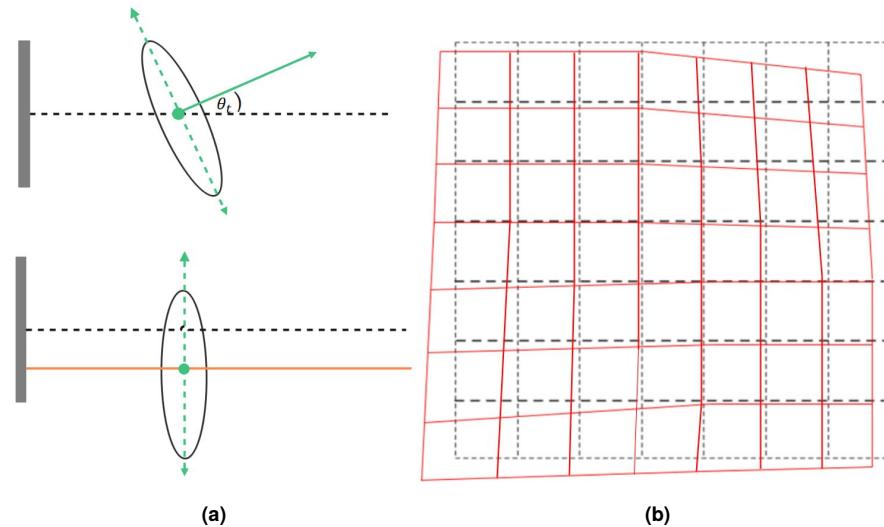


Rysunek 2.19: Rodzaje dystorsji obrazu

Zniekształcenie beczkowe jest rodzajem aberracji, w którym linie proste są zakrzywione do wewnętrz i kształtem przypominają beczkę. Dystorsja ta jest często spotykana w obiektywach szeroko-kątnych, ponieważ pole widzenia obiektywu jest znacznie szersze niż rozmiar przetwornika obrazu przez to obraz musi zostać "ściśnięty". W rezultacie proste linie są wyraźnie zakrzywione do wewnętrz, szczególnie w kierunku skrajnych krawędzi kadru, przykład takiego zniekształcenia przedstawiony został na rysunku 2.19a. Dystorsje poduszkowe są dokładnym przeciwnieństwem poprzedniej aberracji optycznej - linie są zakrzywione na zewnątrz od środka. Wada ta jest często spotykana w teleobiektywach i jest spowodowana powiększeniem obrazu w kierunku krawędzi kadru od osi optycznej. W tym przypadku pole widzenia jest mniejsze niż rozmiar przetwornika obrazu, przykład dystorsji poduszkowej przedstawiony został na rysunku 2.19b. Usuwanie opisanych zniekształceń jest zazwyczaj prostym procesem, ponieważ można wykorzystać oprogramowania do obróbki zdjęć, które posiadają odpowiednie narzędzia do tego celu. Ostatni rodzaj dystorsji jest najtrudniejszy do korekcji i próba jego korekcji może spowodować pogłębieniem zniekształcenia beczkowego lub poduszkowego. Zniekształcenie faliste zostało przedstawione na rysunku 2.19c. Opisane rozwiązanie jest możliwe w fotografii, gdzie taki obraz jest statyczny co oznacza, że nie wymaga poprawy w czasie rzeczywistym [51, 52].

Jeszcze jednym bardzo ważnym rodzajem zniekształceń, które należy scharakteryzować pod-

czas kalibracji jest efekt stycznych. Czasami nazywane jest zniekształceniem niecentrycznym, ponieważ jego główną przyczyną jest fakt, że zespół obiektywu nie jest wyśrodkowany i równoległy do płaszczyzny obrazu. Na rysunku 2.20b można zauważyć obrót, jak i pochylenie płaszczyzny obrazu od promienia środka obrazu, natomiast rysunku 2.20a przedstawia powód powstawania zniekształcenia. Współcześnie ten problem można pominąć, lecz podczas kalibracji kamery lub systemu wizyjnego, należy uwzględnić wszystkie opisane zniekształcenia [53].



Rysunek 2.20: Na rysunku (a) górnego obiektywu nie jest równoległy do płaszczyzny obrazu, który to powoduje efekt stycznych, natomiast rysunek (b) przedstawia zniekształconą płaszczyznę kolorem czerwonym

2.3.2. Parametry kamer

Kalibracja kamer jest istotną czynnością pozwalającą na pozbycie się niekorzystnych efektów w obrazie, jednocześnie zwiększając dokładność odtwarzanego ujęcia danej scenerii. Kalibracji mogą podlegać zewnętrzne parametry, które zależą od położenia oraz orientacji kamery i nie są powiązane z drugą grupą, a mianowicie parametrami wewnętrznymi, takimi jak ogniskowa, pole widzenia, rozdzielcość itd. Wpływ na zmiany parametrów wewnętrznych kamery można uzyskać zmieniając obiektyw lub przeslonę, natomiast parametry zewnętrzne można zmodyfikować odpowiednimi macierzami transformacji.

W wizji komputerowej powszechnie stosowanie są cztery układy współrzędnych - rzeczywisty układ współrzędnych (3D), układ współrzędnych kamery (3D), układ współrzędnych obrazu (2D) oraz układ współrzędnych pikseli (2D). Rzeczywisty układ współrzędnych jest to podstawowy kartezjański układ współrzędnych 3D, czego przykładem może być narożnik ściany - taki punkt może być oznaczony jako $P_w = (X_w, Y_w, Z_w)$. Następnie taki punkt jest interpretowany na układ współrzędnych kamery za pomocą operacji rotacji i translacji, przedstawia to równanie (2.15).

$$\begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = R \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} + T \quad (2.15)$$

gdzie:

x_w, y_w, z_w - współrzędne układu rzeczywistego,

x_c, y_c, z_c - współrzędne układu kamery,

$R = (r_{i,j})$ stanowi macierz rotacji 3x3, który definiuje orientację kamery,

$\mathbf{T} = (t_1, t_2, t_3)^T$ stanowi wektor translacji, który definiuje pozycję kamery.

Punkty z układu współrzędnych kamery rzutowane są na układ współrzędnych obrazu, jest to tzw. "Pin-hole camera model". Jest to transformacja stratna, oznacza to, że rzutowanie punktów z układu współrzędnych kamery na płaszczyznę 2D nie może zostać odwrócone. Informację o głębi zostają utracone, dlatego patrząc na obraz przechwycony przez kamerę, nie można określić rzeczywistej odległości. Omawiana płaszczyzna 2D znajduje się w odległości f od kamery. Współrzędne X_i, Y_i określa się stosując regułę trójkątów podobnych oraz wiedząc, że promień światła wchodzący do ogniskowej pod takim samym kątem pod jakim ten promień wychodzi. Sposób wyliczenia współrzędnych obrazu przedstawia równanie (2.16).

$$\begin{pmatrix} X_i \\ Y_i \\ Z_c \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} \quad (2.16)$$

Ostatnim elementem przetwarzania obrazu jest dyskretyzacja punktów w układzie współrzędnym obrazu do układu współrzędnych pikseli. W celu uzyskania wartości, które są przyjmowane podczas procesu dyskretyzacji, należy podzielić współrzędne obrazu przez szerokość i wysokość piksela (jest to jeden z parametrów kamery, jednostka metr/piksel). Układ współrzędnych pikseli ma swój początek w lewym górnym rogu, dlatego operator translacji (c_x, c_y) jest również wymagany podczas dyskretyzacji. Równanie (2.17) przedstawia wszystkie potrzebne macierze do obliczenia punktu z współrzędnych rzeczywistych do współrzędnych w postaci pikseli.

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{f}{\rho_u} & 0 & c_x & 0 \\ 0 & \frac{f}{\rho_v} & c_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} R_{3x3} & T_{3x1} \\ 0_{1x3} & 1_{1x1} \end{pmatrix}_{(4x4)} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \quad (2.17)$$

gdzie:

ρ_u, ρ_v - piksel z danej kolumny oraz wiersza,

u - kolumny pikseli,

v - wiersze pikseli.

Pierwsza macierz po znaku równości reprezentuje parametry wewnętrzne kamery, natomiast następna macierz reprezentuje parametry zewnętrzne kamery [54, 55, 56, 57].

3. PRZYGOTOWANIE ŚRODOWISKA

Po wprowadzeniu teoretycznym do podstawowych zagadnień obejmujących tematykę pracy oraz przedstawieniu rozwiązań dostępnych w literaturze, należało rozpocząć działania od przygotowania środowiska komputerowego. Odpowiednie przygotowanie tego etapu pozwoli na uniknięcie późniejszych komplikacji spowodowanych m.in. niekompatybilnością wykorzystanych bibliotek lub wersji oprogramowania. W tym rozdziale zostanie zaprezentowany wybór dostępnych narzędzi pozwalających na opracowanie wyników, jak i wybór systemu operacyjnego wraz z językiem programowania.

3.1. System operacyjny

Odpowiedni wybór systemu operacyjnego pozwala na sprawne poruszanie się między dostępymi rozwiązaniami, które wymagają pewnych bibliotek współpracujących z danym systemem. Na rynku dostępne są trzy najpopularniejsze systemy operacyjne - Windows, MacOS oraz Linux. O ile pierwsze dwa systemy są idealnymi wyborami do codziennego użytkowania, to jednak przy bardziej zaawansowanych działaniach pojawiają się problemy. W przypadku oprogramowania Windows ograniczenia pojawiają się już w przypadku korzystania z terminala. Problem polega na braku możliwości uruchomienia niektórych komend oraz w poszczególnych miejscach dostęp jest zablokowany. W porównaniu do Linux'a, który pozwala użytkownikowi wykonanie praktycznie wszystkich poleceń bezpośrednio z konsoli, jak i dostęp do każdego miejsca w systemie. W tym zestawieniu MacOS nie bierze brany pod uwagę, przez swoją integrację ze sprzętem firmy Apple. Olbrzymim atutem Linux'a jest fakt, że jest to system operacyjny typu "*open source*", co pozwala każdemu użytkownikowi na dostęp do całego systemu łącznie z jądrem. Sprawia to, że system posiada wiele zwolenników dbających nie tylko o aspekty bezpieczeństwa, ale również publikuje i rozwija szereg rozwiązań usprawniających działanie systemu i jego możliwości w różnych dziedzinach nauki. Zaletą systemu operacyjnego Linux dla rozwiązań z dziedziny głębokiego uczenia jest łatwy dostęp do procesora graficznego wykorzystywanego w kartach graficznych GPU (ang. "*Graphics Processing Unit*") za pomocą odpowiednich komend używając platformy CUDA (ang. *Compute Unified Device Architecture*) [58]. Jest to platforma obliczeniowa stworzona przez firmę NVIDIA, która pozwala wykorzystanie potencjału kart graficznych do przetwarzania równoległego. Co prawda CUDA jest wspierana na systemie Windows, jak i Linux, ale to na tym drugim jest bardziej stabilna podczas wykonywania działań na GPU. Osoby zajmujące się modelowaniem sieci neuronowych wybierają system operacyjny Linux. Jest to spowodowane wymienionymi powyżej zaletami tego systemu, jak i również zaleceniami twórców narzędzi.

Spośród trzech głównie dominujących systemów operacyjnych do projektu został wybrany Linux. Liczba dystrybucji systemu jest dosyć spora co sprawia, że wybór odpowiedniej nie należy do prosty. Po zaznajomieniu się z forami internetowymi związanymi z tworzeniem modeli uczenia maszynowego, użytkownicy polecają między innymi: Ubuntu, Linux Mint, Debiana oraz Arch Linux. Wybór konkretnej dystrybucji jest indywidualny i zależy od preferencji użytkownika, jak i poziomu doświadczenia. W projekcie została wykorzystana dystrybucja Ubuntu w wersji 22.04.

3.2. Język programowania

Przy tworzeniu modeli głębokich sieci neuronowych wybór języka programowania jest niezwykle istotny. Wybór ten wpływa na: wydajność tworzenia, zarządzanie kodem i sam model. Istnieje wiele popularnych

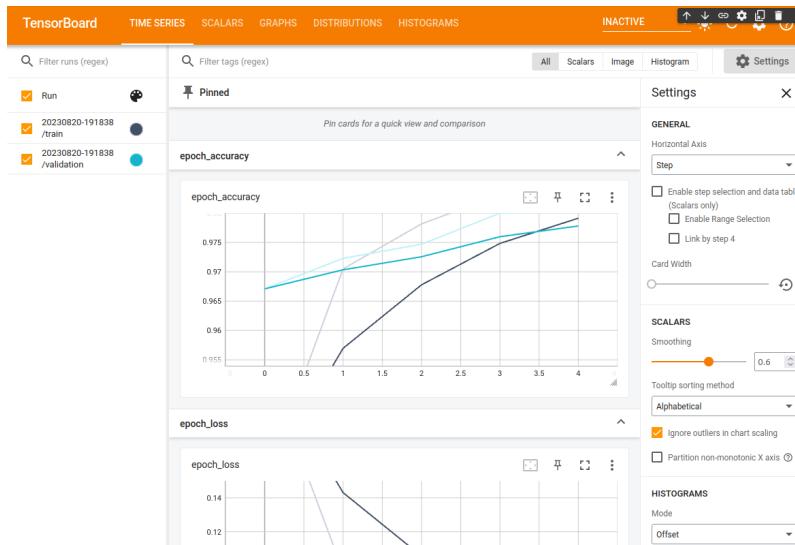
języków programowania, które można wykorzystać do wdrożenia modeli głębokich sieci neuronowych. Python jest popularnym językiem ze względu na bogate wsparcie dla bibliotek i narzędzi związanych z głębokim uczeniem maszynowym, takich jak *TensorFlow*, *PyTorch* i *Keras*. Język *Python* ma łatwą i zrozumiałą składnię, co ułatwia tworzenie i zarządzanie złożonym kodem modeli. Język R również jest używany w dziedzinie uczenia maszynowego i analizy danych, choć mniej powszechnie niż Python. Posiada szereg pakietów, takich jak "caret" czy "mlr", które ułatwiają pracę z modelami statystycznymi i uczeniem maszynowym. Warto również rozważyć języki niskopoziomowe, takie jak Julia czy C++. Wykorzystując C++ można znacznie lepiej zoptymalizować kod, jak i również można skorzystać z bibliotek takich jak *TensorFlow Serving*, aby wdrożyć wysokiej jakości model, szczególnie w zastosowaniach wymagających intensywnych obliczeń. W przeciwieństwie do tego Julia, stosunkowo młody język programowania, cieszy się popularnością w naukach przyrodniczych dzięki swojej wydajności i łatwości w tworzeniu wysokowydajnych numerycznych obliczeń. Wybór języka, który ostatecznie zostanie wybrany zależy od doświadczenia programistycznego, dostępności narzędzi i specyfiki projektu. Niezależnie od wyboru, kluczowe jest efektywne korzystanie z dostępnych bibliotek i narzędzi niż sam język programowania [59, 60].

Język C++ pomimo tego, że posiada wysoką wydajność dzięki komplikacji kodu, można również o wiele lepiej zarządzać pamięcią co jest szczególnie przydatne przy pracach z algorytmami wymagającymi dużej mocy obliczeniowej. W przeciwieństwie do *Pythona* jest to język dużo bardziej wymagający pod względem składni i zrozumienia. Podsumowując do wykonania projektu został wykorzystany język programowania *Python* w wersji 3.10.6.

3.3. Biblioteka sieci neuronowych

Istnieje wiele popularnych bibliotek do *Pythona*, które są szeroko stosowane w dziedzinie głębokiego uczenia. Oferują one m.in. narzędzia, moduły oraz interfejsy programistyczne do tworzenia, trenowania oraz analizy modeli. Spośród wszystkich dostępnych bibliotek można wyróżnić trzy najpopularniejsze oraz w dalszym ciągu rozwijane. *TensorFlow* jest to otwarte oprogramowanie do uczenia maszynowego stworzone przez firmę *Google*. W nawiązaniu do jego wszechstronności i jakości jego popularność wiąże się z narzędziem do szkolenia, analizy i tworzenia modeli głębokich sieci neuronowych i innych modeli uczenia maszynowego. Biblioteka opiera się na grafach obliczeniowych, gdzie węzły reprezentują różne operacje matematyczne, a krawędzie wskazują przepływ danych pomiędzy operacjami. Wydajne i równoległe przetwarzanie jest możliwe dzięki tej metodzie, co jest niezbędne przy pracy z dużymi zbiorami danych i skomplikowanymi modelami. Warto dodać, że biblioteka posiada wsparcie dla obliczeń na kartach graficznych (*GPU*), co pozwala na znaczne przyspieszenie treningu modelu. Ponadto biblioteka oferuje moduły i sprzęt, które ułatwiają tworzenie różnych typów sieci neuronowych, takich jak konwolucyjne sieci neuronowe (*CNN*), rekurencyjne sieci neuronowe lub generatywne sieci neuronowe. Dodatkowo *TensorFlow* dostarcza narzędzie *TensorBoard*, które można wykorzystać do analizy wydajności modeli oraz wizualizacji procesu uczenia się. Narzędzie to pozwala na wizualizację architektury modelu, a dokładniej na wizualizację grafu obliczeniowego modelu, co jest szczególnie przydatne przy analizie jego struktur oraz zrozumieniu przepływu danych pomiędzy warstwami. Bardzo istotnym aspektem przy testowaniu modelu jest weryfikacja jego wydajności, co przekłada się na przeanalizowanie trendów funkcji kosztów oraz dokładności. Rysunek 3.1 przedstawia fragment interfejsu oraz wyniki wydajnościowe modelu po 5 epokach. Posiada on również przydatną opcję porównywania modeli, co umożliwia sprawniejszą weryfikację i wybór najlepszego modelu. Ciekawą opcją jest wizualizacja zasobów obliczeniowych dla *GPU*, jak i *CPU* (ang. *Central Processing Unit*), dzięki czemu możliwa jest zmiana wartości

modelu na bardziej obciążające podzespoły [61].



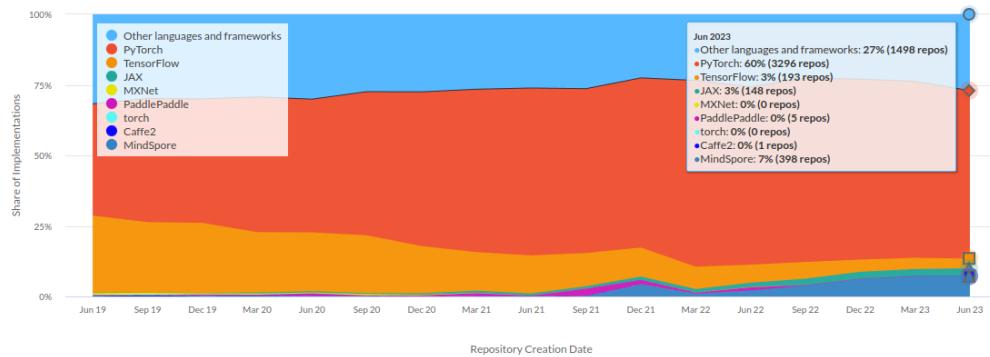
Rysunek 3.1: Fragment interfejsu narzędzia *TensorBoard*

Drugą równie popularną wysokopoziomową biblioteką do tworzenia i trenowania modeli sieci neuronowych jest *Keras*. Wyróżnia się on zaawansowanym interfejsem programistycznym, który został stworzony z myślą o prostocie i wysokim stopniu intuicyjności dla użytkownika. Dzięki temu przyciąga zarówno początkujących, jak i doświadczonych programistów operujących na modelach. Modele buduje się poprzez składanie warstw, są one zorganizowane w sekwencje, co pozwala na tworzenie złożonych modeli. Dzięki takiemu podejściu istnieje możliwość swobodnego eksperymentowania z różnymi architekturami. Podobnie jak w przypadku *TensorFlow*, tak i *Keras* pozwala na tworzenie różnorodnych modeli. Możliwe jest realizowanie modeli sekwencyjnych, modeli z wieloma wejściami i wyjściami oraz bardziej zaawansowanymi jak np. rekurencyjne sieci neuronowe (ang. *Recurrent Neural Network - RNN*) lub generatywna sieć przeciwnista (ang. *Generative Adversarial Network - GAN*). Biblioteka posiada już predefiniowane warstwy, z których to można utworzyć większość modeli, jednocześnie pozwala na tworzenie własnych w zależności od potrzeb. Z początku *Keras* był odrębnym projektem, ale w późniejszych wersjach stał się częścią biblioteki *TensorFlow*. Doczekał się również biblioteki *Keras Core*, która pozwala na zintegrowanie z dowolnym frameworkiem. Jest to pełne przepisanie kodu *Keras*, które opiera ją na modułowej architekturze zaplecza. Umożliwia uruchamianie przepływów pracy począwszy od *TensorFlow*, *JAX* oraz *PyTorch* [62].

PyTorch to zaawansowana biblioteka, która jest wykorzystywana do tworzenia i trenowania modeli uczenia maszynowego oraz głębokiego uczenia. Opiera się na bibliotece z otwartym kodem źródłowym o nazwie *Torch*. Jest łatwa w użyciu i wydajna, dzięki szybkiemu językowi skryptowemu *LuaJIT* oraz bazowej implementacji *C/CUDA*. W 2017 roku projekt *Torch* został przeniesiony na port biblioteki *Python* o nazwie *PyTorch*. Od tego czasu jest głównie rozwijana przez grupę badawczą *Meta AI*. Charakteryzuje się ona dynamicznym grafem obliczeniowym oraz przejrzystym i intuicyjnym interfejsem programistycznym. Wspomniany dynamiczny graf obliczeniowy to jest coś co odróżnia *Pytorcha* od innych bibliotek, takich jak *TensorFlow*. Pojęcie dynamicznego grafu obliczeniowego oznacza, że jest on tworzony i modyfikowany w trakcie działania programu, co ułatwia eksperymentowanie na modelach, jak i ich późniejsze debugowanie. Biblioteka umożliwia programowanie w sposób imperatywny, oznacza to, że można tworzyć kod w sposób bardziej naturalny i podobny do standardowego kodu *Pythona*. Podobnie jak w zbliżonych bibliotekach *PyTorch* również udostępnia szereg udogodnień w postaci modułów,

które pozwalają tworzyć modele warstwa po warstwie, dzięki czemu wprowadza dużą elastyczność w projektowaniu sieci neuronowych. Posiada również wsparcie dla przetwarzania na kartach graficznych, co jest szczególnie przydatne do głębokiego uczenia. *PyTorch* dostarcza bardzo ciekawy i zarazem przydatny moduł *autograd*, umożliwia on automatyczne obliczanie gradientów. Ułatwia to implementację algorytmów uczenia, takich jak propagacja wsteczna [63].

W kontekście wyboru biblioteki do głębokiego uczenia maszynowego, decyzja o wyborze *PyTorch* nad innymi alternatywnymi rozwiązaniami jest strategiczna i podyktowana pewnymi unikalnymi cechami tej biblioteki. Wyróżnia się dynamicznym grafem obliczeniowym, który umożliwia tworzenie i modyfikację modelu w sposób bardziej intuicyjny i naturalny. To znaczące wyróżnienie, szczególnie że językiem programowania w projekcie jest *Python*, a to oznacza, że programowanie imperatywne wspierane przez bibliotekę tworzy spójną całość. Ponadto, elastyczność *PyTorch* w tworzeniu i modyfikacji modeli stanowi istotny atut, zwłaszcza w fazie prototypowania, gdzie eksploracja różnych architektur jest kluczowa. Moduły umożliwiają tworzenie modeli warstwa po warstwie, dając pełną swobodę w projektowaniu złożonych sieci neuronowych. Wsparcie dla *GPU* oraz narzędzia *autograd* dodatkowo zwiększa atrakcyjność, umożliwiając efektywne wykorzystanie mocy obliczeniowej kart graficznych oraz automatyzację obliczeń gradientów, co jest kluczowe w procesie treningu modeli. Warto również podkreślić aktywną społeczność programistyczną, która przekłada się na dostępność różnorodnych źródeł informacji oraz wsparcie w rozwiązywaniu potencjalnych problemów. Na platformie www.paperswithcode.com dostępne są artykuły naukowe z kodem źródłowym do oglądu. Strona ta umożliwia przedstawienie w sposób graficzny, jak i *framework* jest najczęściej używany, przedstawia to rysunku 3.2. W ciągu ostatnich czterech lat *PyTorch* zdominował platformę, co dodatkowo wpływa na decyzję o jego wyborze.



Rysunek 3.2: Wykorzystanie różnych *frameworków* w dołączonych repozytoriach do artykułów [64]

4. IMPLEMENTACJA I EWALUACJA

W tym rozdziale zostaną omówione i zinterpretowane uzyskane wyniki, które pochodzą z przeprowadzonych eksperymentów oraz analizy danych. Celem jest ukazanie, w jaki sposób zastosowane metody segmentacji instancyjnej wpłynęły na jakość detekcji i rozpoznawania obiektów na obrazach, a także porównanie tych wyników z istniejącymi rozwiązaniami oraz omówienie ich implikacji. W pierwszej kolejności przedstawione zostanie utworzenie odpowiedniego zbioru danych składającego się z obiektów zapakowanych oraz omówiony zostanie wpływ jakości zdjęć i ich oznaczeń. Przedstawione zostaną wybrane metryki i wskaźniki jakości, które zostały zastosowane do oceny skuteczności modeli. Następnie zostanie przedstawiony proces tworzenia modelu klasyfikacji oraz detekcji, które to są kluczowe do dalszych omówień dotyczących segmentacji. Omówione zostaną kryteria, na podstawie których dokonano porównania wyników uzyskanych za pomocą zaimplementowanych metod z wynikami osiągniętymi przez inne modele lub podejścia. Ważnym aspektem tego rozdziału będzie również interpretacja wyników w kontekście specyficznych cech zbioru danych oraz wyzwań związanych z segmentacją instancyjną. Należy zwrócić uwagę na ograniczenia i trudności, które mogły wpływać na wynik w szczególnym odniesieniu do obszarów, w których zastosowane metody odniosły szczególny sukces. Po omówieniu modeli do detekcji oraz segmentacji zostanie przedstawione rozwiązanie dotyczące lokalizacji obiektów w przestrzeni. Zaprezentowane zostaną dostępne możliwości uzyskania głębi w obrazie, co pozwoli na lokalizację obiektów. Omówione zostaną powstałe trudności oraz ograniczenia. W końcowej części rozdziału przedstawione zostaną wnioski płynące z opracowania wyników. Te refleksje pozwolą na pełniejsze zrozumienie znaczenia i potencjalnych perspektyw dalszych badań w dziedzinie widzenia komputerowego.

4.1. Zbiór danych

Pierwsze zadanie, jakie należy wykonać przed zaprojektowaniem modelu do segmentacji instancyjnej, jest przygotowanie danych. Celem w pracy są obiekty zapakowane, mogą być one interpretowane w różny sposób m. in. jako przedmioty owinięte w papier pakunkowy lub włożone do kartonu. Przedmioty owinięte w papier nie zmieniają swojego kształtu, a jedynie zmieniają swój kolor. Identyfikacja takiego obiektu, może być utrudniona, ponieważ tekstura obiektu również ulega zmianie, przez co szczególnie przedmiotu ulegają zniekształceniu lub całkowicie zanikają. Zbiorem danych w pracy będą kartony, które to charakteryzują się prostopadłościennym kształtem oraz różnorakimi kolorami, dzięki czemu identyfikowane obiekty posiadają charakterystyczne cechy. Ilość zgromadzonych danych oraz ich różnorodność jest bardzo istotna, ponieważ im więcej informacji otrzyma model podczas treningu, tym lepsze będzie jego działanie. Dane te można pozyskać gromadząc zdjęcia kartonów dostępnych w internecie lub samodzielnie ustawić kartony i zrobić im zdjęcia. Drugi proces jest dużo bardziej czasochłonny i wymaga zgromadzenia znacznej liczby różnych kartonów. Zgromadzone dane należy odpowiednio rozdzielić na minimum dwa zbiory - zbiór treningowy oraz zbiór testowy. Warto również utworzyć trzeci zbiór tzw. walidacyjny, który to nie jest udostępniany modelowi podczas przebiegu treningu. Taki zbiór jest w stanie przedstawić prawdziwą dokładność utworzonego modelu. Utworzenie folderów ze zdjęciami nie mówi nic programowi, dlatego też ważnym etapem jest utworzenie oznaczeń do każdego zbioru.

4.1.1. Gromadzenie danych

Proces zbierania danych rozpoczęto od przeszukania forum internetowego oraz publikacji, z potencjalnym zbiorem zdjęć stert kartonów. Udało się znaleźć już przygotowany przez grupę badaczy sporej wielkości częściowo oznaczony zbiór. Zbiór ten został powiększony o dodatkowe zdjęcia oraz dodane zostały brakujące oznaczenia. Rysunek 4.1 przedstawia kilka przykładowych zdjęć znajdujących się w opisywanym zbiorze [65].



Rysunek 4.1: Przykładowe obrazy ze zbioru danych

Nagromadzenie wystarczającej ilości danych w zależności od potrzeb jest stosunkowo czasochłonne. W przypadku korporacji wydawane są olbrzymie środki, aby zbiory danych były jak największe przy jednoczesnym pisaniu adnotacji do każdego zdjęcia. Zbiory, które posiadają miliony zdjęć z kilkunastoma tysiącami klas, są tworzone przez grupy osób przez kilka miesięcy. Wykorzystanie już wcześniej zgromadzonego zbioru danych pozwoliło oszczędzić odpowiednią ilość czasu, która to została przełożona na zgłębianie teorii potrzebnej do dalszej części projektu. Omawiany zbiór danych składa się z 7400 zdjęć treningowych oraz 1000 zdjęć walidacyjnych. Na każde zdjęcie przepada od kilku do kilkunastu badanych obiektów. Łącznie w zbiorze walidacyjnym znajduje się około 20 tys. obiektów, natomiast w zbiorze treningowym około 148 tys. obiektów, co przekłada się na średnią wartość dwudziestu obiektów na zdjęcie.

4.1.2. Etykietowanie danych

Podczas tworzenia danych etykietowanie danych jest to drugą najważniejszą czynnością, jaką należy wykonać w celu utworzenia poprawnego zbioru danych. Do etykietowania danych służą specjalnie za-

projektowane aplikacje komputerowe, dzięki którym można w zależności od potrzeb: sklasyfikować obraz, dodać otoczkę wokół obiektu wraz z jego klasyfikacją lub wybrać zbiór punktów tworząc wielokąt wokół obiektu do segmentacji. Istnieją również strony internetowe, które pozwalają na dodanie adnotacji do wgranego zbioru danych, jedną z takich stron jest *Roboflow*. Platforma ta została stworzona z myślą o analizie obrazu, gdzie użytkownicy mogą dzielić się zbiorami danych, jak i utworzonymi modelami. Na stronie dostępne jest ponad 200 milionów zdjęć, ponad 200 tysięcy zbiorów danych oraz ponad 50 tysięcy wstępnie przetrenowanych modeli. Jednak najciekawszą opcją na stronie jest właśnie wspomniana możliwość etykietowania wprowadzonych danych. Rysunek 4.2 przedstawia układ strony z narzędziami do zaznaczenia obiektu oraz nadania mu odpowiedniej etykiety. Serwis umożliwia utworzenie oznaczeń do detekcji, jak i segmentacji. W przypadku detekcji do dyspozycji jest narzędzie do rysowania prostokątów, natomiast do segmentacji możliwe jest naniesienie punktów, które to następnie tworzą wielokąt. Możliwe jest skorzystanie z asystentów do tworzenia w szybki sposób wielokąta, dzięki systemowi wykrywania krawędzi oraz jest możliwe uruchomienie modelu do detekci obiektów w celu sprawniejszego etykietowania danych.



Rysunek 4.2: Zakładka strony *Roboflow* do etykietowania danych

Wspomniana strona internetowa posiada również bibliotekę do języka *Python* o takiej samej nazwie, która to umożliwia pobieranie zbioru danych z chmury serwisu. Rozwiązań to pomaga w tworzeniu przenośnego programu, który można uruchomić na dowolnej platformie sprzętowej. Jedną z takich platform jest *Google Colab*, serwis ten udostępnia przystępny dla użytkownika interfejs wraz z dostępem do chmury obliczeniowej *Google*. Strona opiera się na popularnym narzędziu *Jupyter Notebook*. Oznacza to możliwość pisania kodu w języku *Python* w jednej komórce, pod którą możliwe jest wyświetlenie wyniku w postaci tekstu lub obrazka. Rysunek 4.3 przedstawia otwarty projekt na platformie. W darmowej wersji dostarczane są zasoby obliczeniowe w postaci procesora graficznego (*GPU*), procesora tensorycznego (ang. *Tensor Processing Unit - TPU*) oraz procesora (*CPU*). Platforma ta sprawdza się bardzo dobrze dla użytkowników rozpoczynających swoją naukę z uczeniem maszynowym. Udogodniany procesor graficzny korzystnie wpływa na możliwości obliczeniowe w przypadku tworzenia i trenowania głębokich sieci neuronowych. Serwis posiada również swoje wady w postaci ograniczonego czasu pracy na jednej sesji. W przypadku wykorzystanego czasu sesja ulega zamknięciu, a co za tym idzie wszelkie dane tymczasowe zostają usunięte. Dlatego też należy pamiętać o zapisaniu uzyskanych wyników do lokalnej maszyny lub do zintegrowanego serwisu *Google Drive*.

```

import matplotlib.pyplot as plt
import torch
import torchvision

from PIL import Image
from torch import tqt
from torchfile import Path
from torchfile import Path
from models import VGG16
from torch import nn
from torchvision import transforms
from torch.utils.data import DataLoader
from torchsummary import summary

print("torch version (%s)" % torch.__version__)
print("torchvision version (%s)" % torchvision.__version__)

torch version 2.0.0+cu118
torchvision version 0.10.1+cu118

[ ] device = "cuda" if torch.cuda.is_available() else "cpu"
device
'cuda'

[ ] ROOT = Path("./data")
BATCH_SIZE = 256
EPOCH = 30

```

Rysunek 4.3: Strona główna z otwartym projektem na *Google Colab*

4.2. Projektowanie modelu

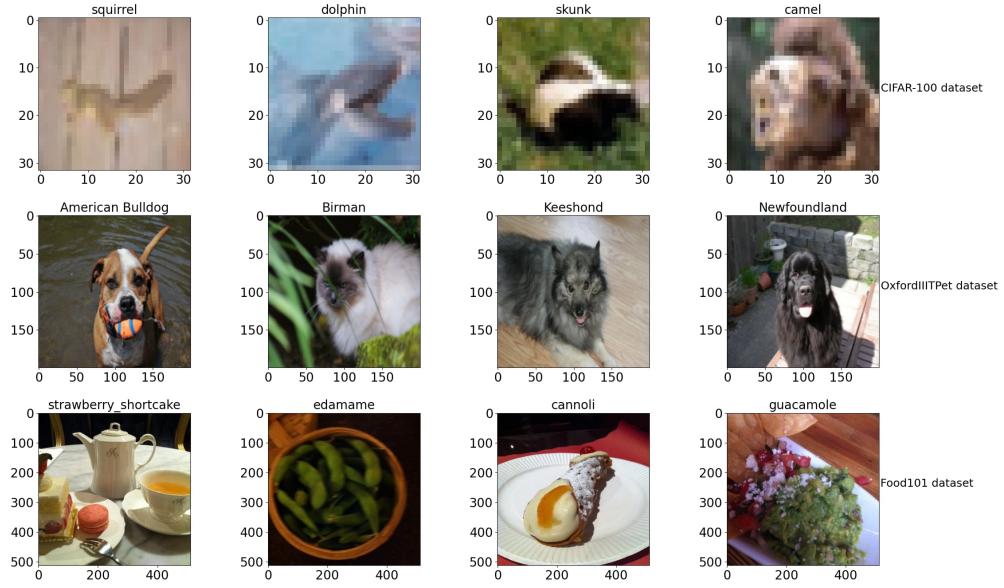
Projektowanie modelu do segmentacji instancyjnej wymaga wiedzy nie tylko z zakresu głębokiego uczenia, ale również umiejętności w pisaniu skryptów w wybranym języku, jak i znajomości popularnych frameworków. Połączenie umiejętności tworzenia programów w języku *Python*, jak i znajomość biblioteki *PyTorch*, pozwoliło na rozpoczęcie pracy nad modelami do rozwiązywania problemów wizji komputerowej. Początkowe prace składały się z utworzenia prostego modelu do klasyfikacji obrazu, zarówno wykorzystując architekturę konwolucyjnych sieci neuronowych, jak i transformera. Takie podejście pozwala na zrozumienie działania tak podstawowego zagadnienia, a zarazem kluczowego dla dalszych prac. Kolejne etapy pracy wymagały zapoznania się w tworzeniu modelu do detekcji obiektów bardzo istotnego zagadnienia, którego rezultaty wymagane są nie tylko w kontekście segmentacji, ale również w lokalizacji w przestrzeni. Po przeprowadzonych testach w utworzonych modelach wyciągnięte zostały wnioski, które wpływały na decyzję dotyczącą dalszych działań. W kolejnych etapach zdecydowano się na wykorzystanie istniejących rozwiązań, które były dostępne jako *open source*. Odpowiednie treningi i testy pozwoliły na wybranie konkretnego rozwiązania i zaimplementowanie go do całego programu.

4.2.1. Model do klasyfikacji

Rezultaty z opracowanego modelu do klasyfikacji obrazu są kluczowe do dalszych podejmowanych kroków. W związku z wieloma dostępnymi architekturami sieci neuronowych decyzja o wyborze najlepszej można wstępnie podjąć analizując wyniki klasyfikacji. Podczas tworzenia projektu, również stosowano taką zasadę, dlatego też w pierwszej kolejności zostały stworzone modele do klasyfikacji obrazu wykorzystując klasyczne konwolucyjne sieci neuronowe oraz ich wariacje, jak i transformery.

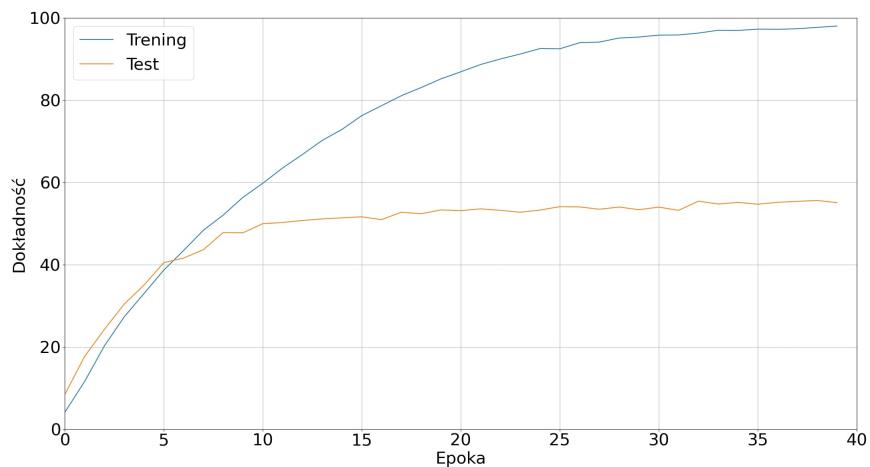
Pierwszą testowaną architekturą jest *VGG-16*, do testów wykorzystano zbiór danych *CIFAR-100*, *OxfordIIITPet* oraz *Food101*. Są to stosunkowo małe zbiory danych, a co za tym idzie pozwalają na szybszy trening modelu i jego weryfikację. Rysunek 4.4 przedstawia przykładowe obrazy wykorzystane do celów treningowych oraz testowych modelu. Wybrane zostały trzy zbiory, które różnią się od siebie głównie w wielkości obrazu, dzięki temu można określić wydajność modelu zarówno dla mniejszych jak i większych danych wsadowych.

Trening modeli obejmował czterdzieści epok oraz wykorzystanie algorytmów optymalizujących *SGD* oraz *Adam*. Przetestowanie działania dwóch optymalizatorów pozwoliło zdecydować, który z nich sprawdza się lepiej dla podanych wielkości obrazów. Rysunek 4.5a przedstawia wykresy wartości funkcji kosztów oraz wartości dokładności modelu na epokę dla zbioru danych *CIFAR-100* oraz optymalizatora *SGD*. Zbiór ten charakteryzuje się obrazami o wymiarach 32px na 32px, dlatego też można zauważać

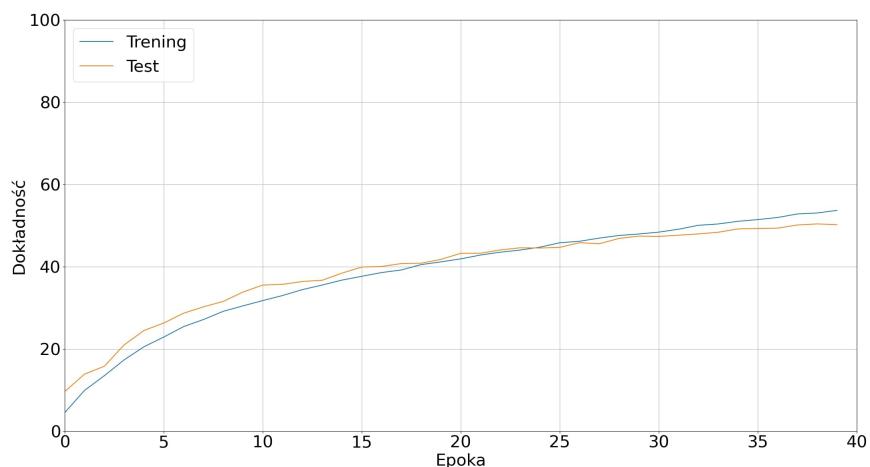


Rysunek 4.4: Przykładowe obrazy wraz z ich etykietami z trzech zbiorów testowych.

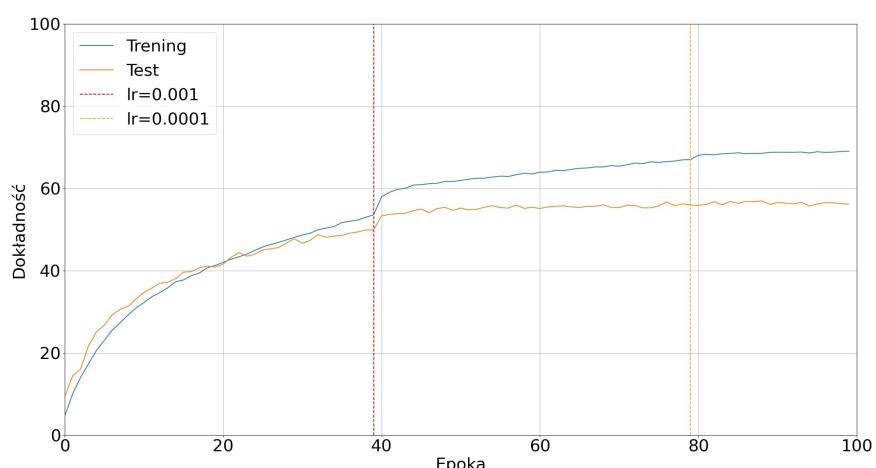
stosunkowo szybki wzrost dokładności na epokę. Jednocześnie pierwsze treningi modelu ukazały problemy, ponieważ jak można zauważyc wartości dokładności modelu w pętli treningowej osiągają 95%, natomiast w pętli testowej dla danych niewidzianych podczas treningu te wartości osiągają okolice 55%. Jest to związane z przeuczeniem się modelu, oznacza to, że model nauczył się idealnie dopasowywać do danych treningowych, ale ma problemy z generalizowaniem dobrze nowych niewidzianych wcześniej danych. W celu rozwiązania powyższego problemu zastosowano transformacje obrazu w postaci normalizacji oraz losowej rotacji. Pozwoliło to na zwiększenie generalizacji dla modelu, co poprawiło wyniki w pętli testowej, co przedstawia rysunek 4.5b dla tych samych danych oraz liczby epok. Można zauważyc, że model znacznie wolniej uczy się po osiągnięciu wyniku 50%, dlatego też zastosowano metodę zmiennej wartości lr . Wartość została zmniejszona dziesięcokrotnie w czterdziestej oraz osiemdziesiątej epoce, rezultaty można zauważyc na rysunku 4.5c. Porównując wykres pierwszy z dwoma kolejnymi, można zauważyc poprawę działania modelu. W dalszym ciągu nie jest on idealny, ale uzyskane wyniki pozwalają na zrozumienie powstałych problemu oraz im przeciwdziałaniu. Następnie należy wprowadzić większe augmentacje zbioru danych oraz wprowadzenie w odpowiednich momentach zmiany wartości tempa uczenia (lr). Podobne wyniki można zauważyc przy zbiorze z obrazami o większej rozdzielczości, tutaj również zachodzi efekt przeuczenia się modelu. Większa rozdzielcość obrazu wpływa w sposób znaczący na czas trwania treningu oraz na jego jakość. Rysunek 4.6a prezentuje pozytywny wpływ większej rozdzielczości danych na problem przeuczenia się, natomiast rysunku 4.6b pokazuje, że podczas pętli testowania zachodzą zmiany wartości skokowo.



(a) Oryginalny zbiór danych

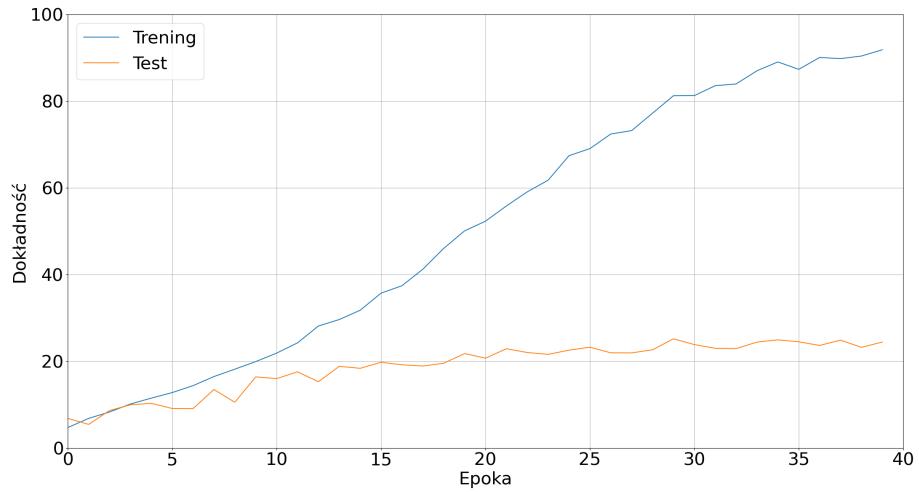


(b) Zbiór danych po zastosowaniu augmentacji

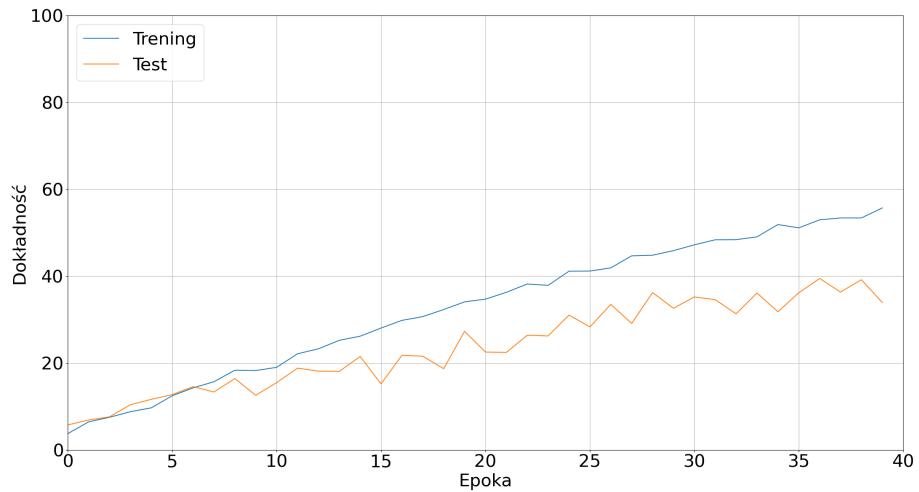


(c) Zbiór danych po zastosowaniu augmentacji, stu epok oraz metody zmiennego tempa uczenia

Rysunek 4.5: Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora *SGD* dla zbioru danych *CIFAR*



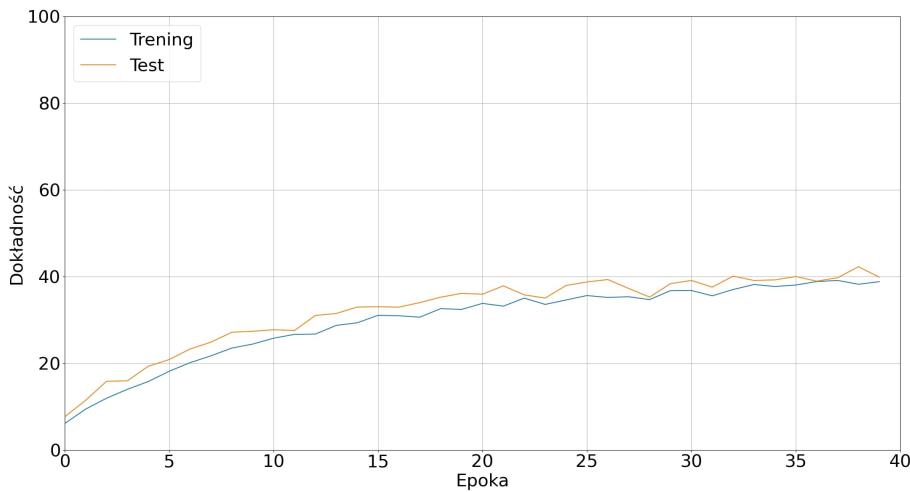
(a) Oryginalny zbiór danych



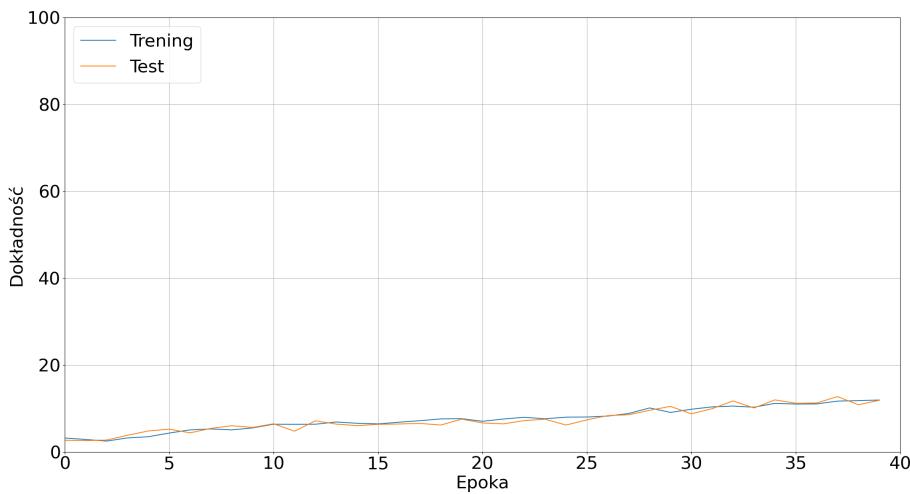
(b) Zbiór danych po zastosowaniu augmentacji

Rysunek 4.6: Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora SGD dla zbioru danych OXFORD

Rysunek 4.7 przedstawia wykresy dla obu zbiorów danych, lecz z wykorzystaniem optymalizatora Adam. Jak można zauważyć model uczący się zbiory obrazów o małej rozdzielczości, zaczyna się stabilizować w okolicy czterdziestu procent dokładności, co jest nie satysfakcyjnym wynikiem jak na powierzone mu zadanie. Z drugiej strony można zaobserwować jeszcze wolniejsze tempo uczenia się na danych z obrazami o dziesięciokrotnie większej rozdzielczości. Model w tym przypadku po czterdziestu epokach osiąga okolice szesnastu procent dokładności. Podczas treningu optymalizator SGD posiadał parametry odpowiednio tempo uczenia (lr) wynoszące 0.001 oraz momentum wynoszące 0.9, wyjątkiem stanowi wynik ze zmiennym tempem uczenia, gdzie wartość startowa to 0.01. W przypadku optymalizatora Adam wartości były stałe w każdym treningu i posiadały parametry odpowiednio tempo uczenia wynoszące 0.001, β_1 równe 0.9 oraz β_2 równe 0.999. Jak można zauważyć wyniki są dużo gorsze niż w przypadku optymalizatora SGD, dlatego też w dalszej części pracy wykorzystano optymalizator SGD z dodatkowym parametrem *momentum*.



(a) Zbiór danych *CIFAR* po zastosowaniu augmentacji

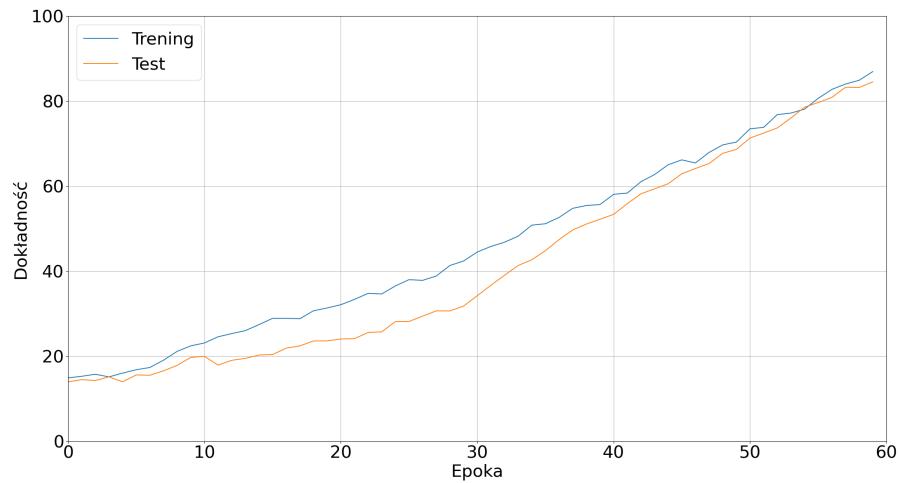


(b) Zbiór danych *OXFORD* po zastosowaniu augmentacji

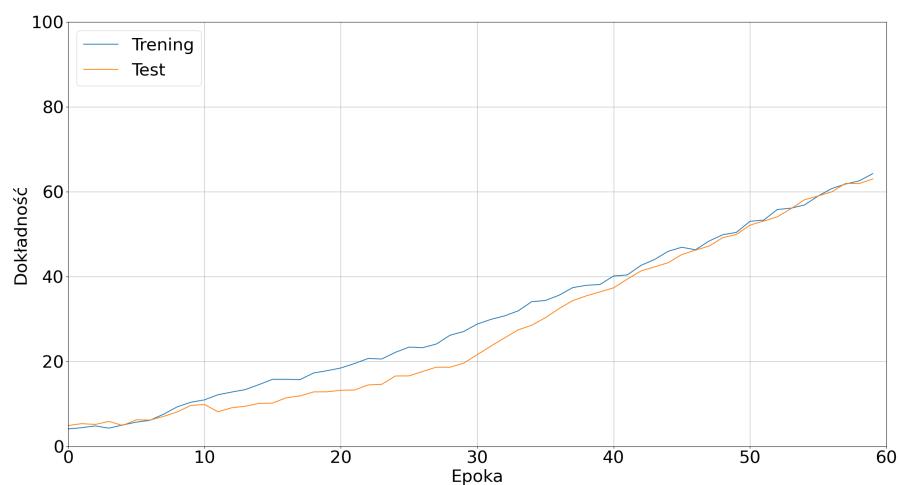
Rysunek 4.7: Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora *Adam*

Po dokonanych treningach dla różnych wariacji danych oraz metodach polepszenia wyników, rozpoczęto rozwijać testy nowszej architektury opartej na mechanizmie atencji, czyli transformerach. Same treningi dla sieci konwolucyjnej odbywały się z wykorzystaniem architektury *VGG16*, ze względu na swoją uniwersalność warstw oraz podobieństwach nowszych architektur w oparciu o splot macierzowy. Usyskane wyniki modelu oraz przeanalizowanie wyników w dostępnej literaturze, pozwoliły na podjęcie decyzji o przetestowaniu jednej z najnowszych architektur. Modele oparte na transformerze uzyskują bardzo dobre wyniki oraz wymagają stosunkowo małego czasu na przeanalizowanie obrazu, dlatego też utworzono model *ViT-B-16*. Jest to najmniejszy spośród przedstawionych modeli w literaturze, jednocześnie nie przekracza on możliwości dostępnego sprzętu. To co wyróżnia trenowanie modelu *ViT* to spore zapotrzebowanie na ilość danych oraz wymagana większa liczby epok. Podczas trenowania modelu wykorzystano te same zbiory danych co w poprzednim modelu. Początkowe próby treningu dla podobnej liczby epok kończyły się z wynikiem podobnym co najgorszy rezultat poprzedniego modelu, dlatego też wprowadzone zostały zmiany w postaci augmentacji danych. Dzięki takiemu działaniu mo-

model mógł rozpocząć swój trening zwracając przy tym dostateczne wyniki. Przetestowano wpływ zmiany optymalizatora, dla *SGD* przyjęto tempo uczenia o wartości $1e^{-4}$ oraz *momentum* 0.9, natomiast dla optymalizatora *Adam* przyjęto wartość tempa uczenia równą $1e^{-5}$ oraz pozostałe parametry takie same jak w modelu *VGG16*. Rysunek 4.8a przedstawia wykres dokładności modelu w pętli treningowej oraz w pętli testowej dla danej epoki dla zbioru danych *CIFAR-100*. Wcześniejsze transformacje danych nie skupiały się na zmianie wielkości obrazu, lecz w modelu *ViT* wprowadzenie zbyt małej wielkości obrazu będzie powodować bardzo powolne i nie efektywne uczenie. W związku z tym dla zbioru danych *CIFAR* zmieniono wielkość obrazu na 208px na 208px, wykorzystując do tego wbudowaną funkcję *Resize*. Wielkość obrazu jest również istotna, ponieważ zgodnie z wybraną architekturą musi ona być wielokrotnością liczby 16, taka liczba została założona jako wielkość obrazu w mechanizmie *PatchEmbedding*. Otrzymane wyniki po sześćdziesięciu epokach treningu są dużo bardziej obiecujące niż miało to miejsce w klasycznej sieci konwolucyjnej. Otrzymany wynik przekroczył 80% dokładności.



(a) Zbiór danych *CIFAR* po zastosowaniu augmentacji



(b) Zbiór danych *FOOD* po zastosowaniu augmentacji

Rysunek 4.8: Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora *Adam*

Natomiast rysunku 4.8b przedstawia wykres dokładności dla zbioru danych *FOOD101*. Tutaj również

można zauważać optymalne wyniki modelu, przekraczające 60% dokładności. Należy pamiętać, że zbiór *FOOD101* jak i *OxfordIIIITPet* posiadają obrazy o większej rozdzielcości, a co za tym idzie potrzeba większej liczby epok, aby uzyskać lepsze wyniki. Najgorzej spośród wszystkich zbiorów testowych wy padł *OxfordIIIITPet*. Jest to prawdopodobnie spowodowane tym, że składa się on ze zbioru zwierząt o różnym kolorze sierści często zbyt zbliżonym do innej klasy co powoduje pewną rozbieżność modelu. Tabela 4.1 przedstawia wyniki dokładności dla obu modeli wraz z odpowiednimi optymalizatorami oraz odpowiednią wariacją zbiorów danych. Warto w tym przypadku zwrócić uwagę na niewielkie różnice dokładności modelu *ViT* między optymalizatorem *SGD*, a optymalizatorem *Adam*.

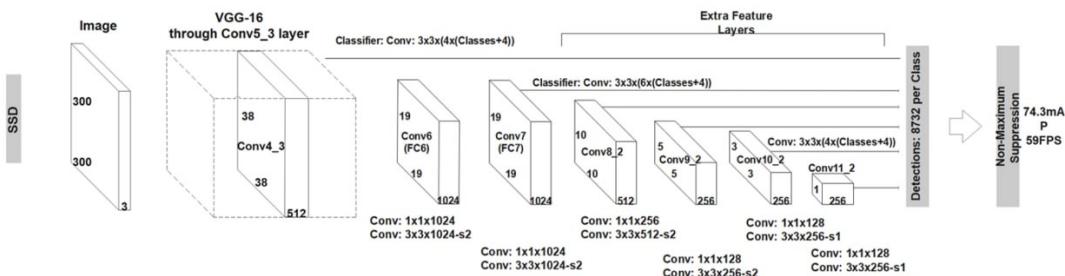
Tabela 4.1: Wyniki dla poszczególnych modeli oraz optymalizatorów

Model	Optymalizator	Zbiór danych	Dokładność [%]
VGG16	SGD	CIFAR-100	55.45
VGG16	SGD	CIFAR-100 z augmentacją	52.35
VGG16	SGD	CIFAR-100 z augmentacją i zmiennym lr	58.52
VGG16	SGD	OXFORD	25.34
VGG16	SGD	OXFORD z augmentacją	38.55
VGG16	SGD	OXFORD z augmentacją i zmiennym lr	45.38
VGG16	SGD	FOOD	35.14
VGG16	SGD	FOOD z augmentacją	69.53
VGG16	SGD	FOOD z augmentacją i zmiennym lr	72.24
VGG16	Adam	CIFAR-100	19.84
VGG16	Adam	CIFAR-100 z augmentacją	41.52
VGG16	Adam	OXFORD	9.24
VGG16	Adam	OXFORD z augmentacją	17.51
VGG16	Adam	FOOD	5.59
VGG16	Adam	FOOD z augmentacją	24.39
ViT	SGD	CIFAR-100 z augmentacją	83.23
ViT	SGD	CIFAR-100 z augmentacją i zmiennym lr	89.51
ViT	SGD	OXFORD z augmentacją	68.78
ViT	SGD	OXFORD z augmentacją i zmiennym lr	75.51
ViT	SGD	FOOD z augmentacją	62.54
ViT	SGD	FOOD z augmentacją i zmiennym lr	67.43
ViT	Adam	CIFAR-100 z augmentacją	84.44
ViT	Adam	OXFORD z augmentacją	64.51
ViT	Adam	FOOD z augmentacją	71.63

4.2.2. Model do detekcji

Klasyfikacja zdjęć na podstawie głównego obiektu jest mało wymagająca dla dzisiejszych jednostek obliczeniowych. Wymaga odpowiedniej segregacji zdjęć, ponieważ w przeciwnym wypadku model może być niepewny w rozpoznawaniu obiektu na zdjęciu. W tym celu wykorzystuje się model do identyfikacji obiektów widocznych na obrazie. Pozwala on na wykrycie wielu obiektów oraz oznaczenie ich w prostokątne ramy tzw. (ang. *bounding box*), a następnie przypisywanie obiekowi odpowiednią klasę. Detekcja obiektów w obrazie pozwalała na większy rozwój przetwarzania obrazu w wielu dziedzinach, m.in. w identyfikacji obiektów codziennego życia (produktów na półkach sklepowych, znaków drogowych, pojazdów czy też ludzi), w medycynie (w celu rozpoznania i lokalizacji zmian chorób wspiera tym podejmowanie decyzji przez lekarzy), samochodach autonomicznych (detekcja obiektów jest niezbędna do prawidłowego podejmowania decyzji i zwiększenia bezpieczeństwa). Początkowe modele zmagały się z dokładnością w okolicach 60% oraz bardzo wysokim czasem przetwarzania, co uniemożliwiło pracę w czasie rzeczywistym.

Bazując na artykule "SSD: Single shot multibox detector" [66] został utworzony własny model SSD-300, gdzie liczba 300 oznacza wielkość obrazu wejściowego 300px na 300px. Rysunek 4.9 przedstawia ogólny przebieg poszczególnych warstw w architekturze oraz wynik końcowy. Całość można rozbić na trzy główne składowe - ekstrakcja map cech (podobnie jak w modelach klasyfikacji), predykcja detekcji oraz warstwa tłumienia wykorzystująca algorytm NMS (ang. *Non-Maximum Suppression*).



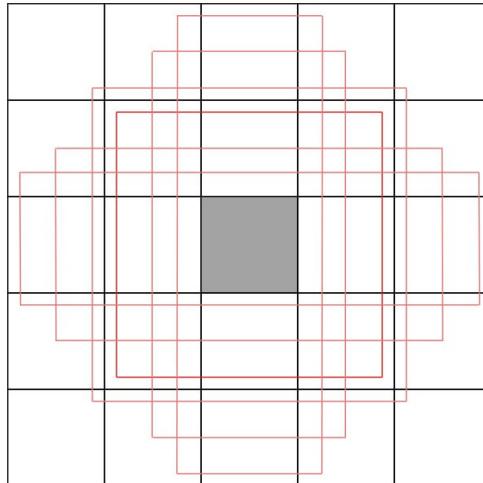
Rysunek 4.9: Diagram prezentujący ogólne założenia architektury SSD-300

Pierwsze warstwy mają za zadanie ekstrakcję niskopoziomowych map cech, dlatego też wykorzystano model VGG-16, z którego to wyciągane są cechy międzywarstwowe wykorzystane w późniejszym przewidywaniu prostokątnych ramek. Ten etap posiada tylko warstwy konwolucyjne. Następnie tworzonych jest kilka dodatkowych warstw, których zadaniem jest ekstrakcja wysokopoziomowych cech. Niskopoziomowe, jak i wysokopoziomowe mapy cech wykorzystywane są do wychwycenia małych, jak i dużych obiektów, nie mniej jednak warto tutaj nadmienić, że zbyt małe obiekty są bardzo trudne do wykrycia i przeanalizowania. Ekstrakcja map cech wyciągana jest z następujących warstw: *Conv4_3*, *Conv6*, *Conv7*, *Conv9_2*, *Conv10_2*, *Conv11_2*. Każda z tych warstw na wyjściu ma inny wymiar mapy cech, jednocześnie każda z nich jest wykorzystywana w kolejnym etapie, czyli detekcji. Predykcja prostokątnych ramek polega na nałożeniu kilku tysięcy prostokątów o różnej wielkości i wybranie spośród nich te o największym pokryciu danego obiektu. Tak zwane hipotezy to wstępne obliczone stałe pola, które wspólnie reprezentują zbiór prawdopodobnych i przybliżonych prognoz prostokątnych ramek. Do tworzenia prognoz prostokątów wykorzystuje się skale rosnącą liniowo dla wskazanych warstw oraz zmienną proporcje danej prognozy. Tabela 4.2 przedstawia wszystkie wartości jakie zostały wykorzystane do prognozowania prostokątnych ramek. Podczas nakładania ramek na obszar może dojść do przekroczenia granic, w takim przypadku prostokąty są ucinane do granic. Rysunek 4.10 przedstawia przykładowe

nałożenie prostokątów w warstwie *Conv9_2* w centralnym punkcie.

Tabela 4.2: Dane potrzebne do prognozy detekcji dla poszczególnych warstw

Mapa cech z warstwy	Wymiary mapy cech	Skala hipotez	Proporcje	Liczba hipotez na obszar	Sumaryczna wartość priors na całą mapę cech
<i>Conv4_3</i>	38 x 38	0.1	1:1, 2:1, 1:2, + extra	4	5776
<i>Conv7</i>	19 x 19	0.2	1:1, 2:1, 3:1, 1:2, 1:3, + extra	6	2166
<i>Conv8_2</i>	10 x 10	0.375	1:1, 2:1, 3:1, 1:2, 1:3, + extra	6	600
<i>Conv9_2</i>	5 x 5	0.55	1:1, 2:1, 3:1, 1:2, 1:3, + extra	6	150
<i>Conv10_2</i>	3 x 3	0.725	1:1, 2:1, 1:2, + extra	4	36
<i>Conv11_2</i>	1 x 1	0.9	1:1, 2:1, 1:2, + extra	4	4
SUMA	-	-	-	-	8732



Rysunek 4.10: Przykładowe nałożenie prostokąteków zgodnie z danymi dla warstwy *Conv9_2*

Ostatnim etapem jest wybór najlepszych ramek spośród wszystkich prognozowanych, wykorzystuje się do tego celu tzw. *ground truth*, czyli ręcznie oznaczone obiekty z umieszczonymi współrzędnymi w zbiorze danych. Do oceny jakości stosuje się współczynnik *Jaccarda* znany również jako IoU (ang. *Intersection over Union*). Jest to stosunek części wspólnej obszarów do sumy tych obszarów wartości są zbiorem liczb rzeczywistych w przedziale $<0,1>$ określa go następujący wzór:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (4.1)$$

Gdzie:

A, B - dwa różne zbiory.

Jeżeli wskaźnik Jaccarda jest mniejszy lub równy pewnemu progowi to taka ramka traktowana jest jako niepoprawne dopasowanie, natomiast jeżeli wartość będzie większa niż wartość progowa to taki prostokąt brany jest pod uwagę jako jeden z ostatecznych, który zawiera obiekt. Dodatkowo, aby model nie wyszukiwał stale obszarów, na których nie znajduje się obiekt, należy przygotować odpowiednią funkcję kosztów. Funkcja ta nazywana jest ang. *Multibox loss*, uwzględnia ona zarówno błędy w lokalizacji, jak i w klasyfikacji obiektów. Błędy w lokalizacji są wartościami współczynnika Jaccarda i to na ich podstawie wymierzana jest kara dla modelu, uwzględniane są tylko wartości z pozytywnych dopasowań i na ich podstawie model optymalizuje swoje predykcje, negatywne dopasowania zostają zignorowane. Wykorzystywana do tego celu jest funkcja *Smooth L1 loss*, równanie (4.2) opisuje matematyczny proces wyliczania błędów lokalizacji.

$$\begin{aligned} L_{loc}(x, l, g) &= \sum_{i \in \text{Pos}}^N \sum_{m \in \{cx, cy, w, h\}} x_{ij}^k \text{smooth}_{L1} \left(l_i^m - \hat{g}_j^m \right) \\ \hat{g}_j^{cx} &= \left(g_j^{cx} - d_i^{cx} \right) / d_i^w \quad \hat{g}_j^{cy} = \left(g_j^{cy} - d_i^{cy} \right) / d_i^h \\ \hat{g}_j^w &= \log \left(\frac{g_j^w}{d_i^w} \right) \quad \hat{g}_j^h = \log \left(\frac{g_j^h}{d_i^h} \right) \end{aligned} \quad (4.2)$$

$$x_{ij}^p = \begin{cases} 1 & \text{jeżeli } IoU > 0.5 \text{ między predykcją ramki } i \text{ a ramką z ground truth } j \text{ dla klasy } p \\ 0 & \text{otherwise} \end{cases}$$

Gdzie:

g^{cx}, g^{cy}, g^w, g^h - dane predykcji ramek odpowiednio współrzędne lewego rogu, szerokość oraz wysokość.

Natomiast błędy w klasyfikacji związane są z niepoprawną predykcją klasy obiektu. Wykorzystana została tzw. metoda *Hard Negative Mining*, która polega na jeszcze większym karaniu modelu, w których najtrudniej było rozpoznać brak obiektów. Najtrudniejsze błędy znajdywane są przez funkcję entropii krzyżowej dla każdej negatywnie dopasowanej prognozy i wybranie tych z najwyższą stratą. Następnie wynik kosztów klasyfikacji jest sumą strat entropii krzyżowej wśród pozytywnych i mocno negatywnych dopasowań. Równanie (4.3) przedstawia funkcję kosztów klasyfikacji obiektów. Równanie (4.4) prezentuje ostateczną funkcję kosztów, w opisywanym modelu użyto wartości $\alpha = 1$.

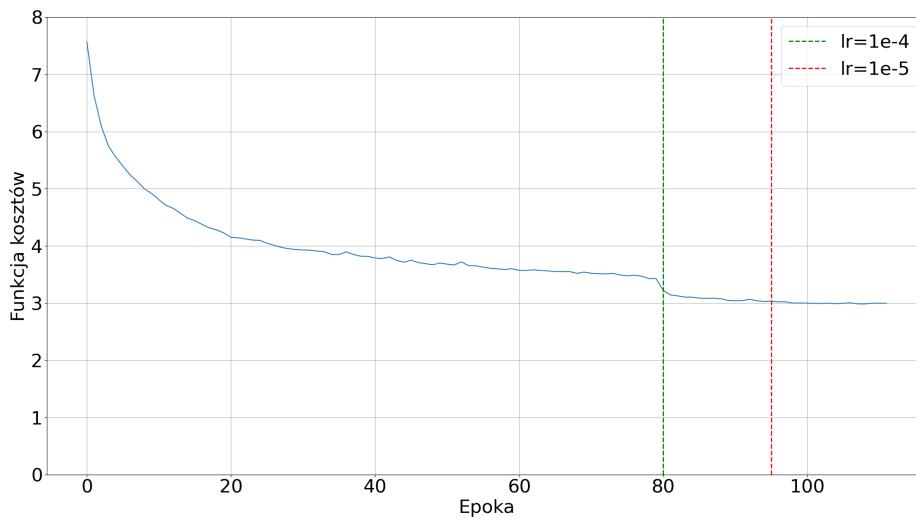
$$L_{conf}(x, c) = \sum_{i \in Pos}^N x_{ij}^p \log(\hat{c}_i^p) + \sum_{i \in Neg} \log(\hat{c}_i^0) \quad \text{gdzie} \quad \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)} \quad (4.3)$$

Gdzie:

N - oznacza liczbę dopasowanych ramek predykcji.

$$L(x, c, l, g) = \frac{1}{N} (L_{conf}(x, c) + \alpha L_{loc}(x, l, g)) \quad (4.4)$$

Model SSD-300 został wytrenowany na zbiorze danych VOC2012. Jest to zbiór danych składający się z obiektów codziennego życia, pojazdów lądowych, powietrznych oraz zwierząt. Zbiór nadaje się idealnie do testów, ponieważ posiada obiekty duże, średnie, jak i małe, dzięki czemu model będzie dokładniejszy dla obiektów każdej wielkości. Rysunek 4.11 przedstawia krzywą wartości funkcji kosztów dla każdej epoki, dodatkowo została zaznaczony moment zmiany współczynnika uczenia ($1e^{-4}, 1e^{-5}$) dla optymalizatora *SGD*. Rysunek 4.12 przedstawia przykładowe wyniki detekcji obiektów.



Rysunek 4.11: Krzywa funkcji kosztów dla poszczególnych epok



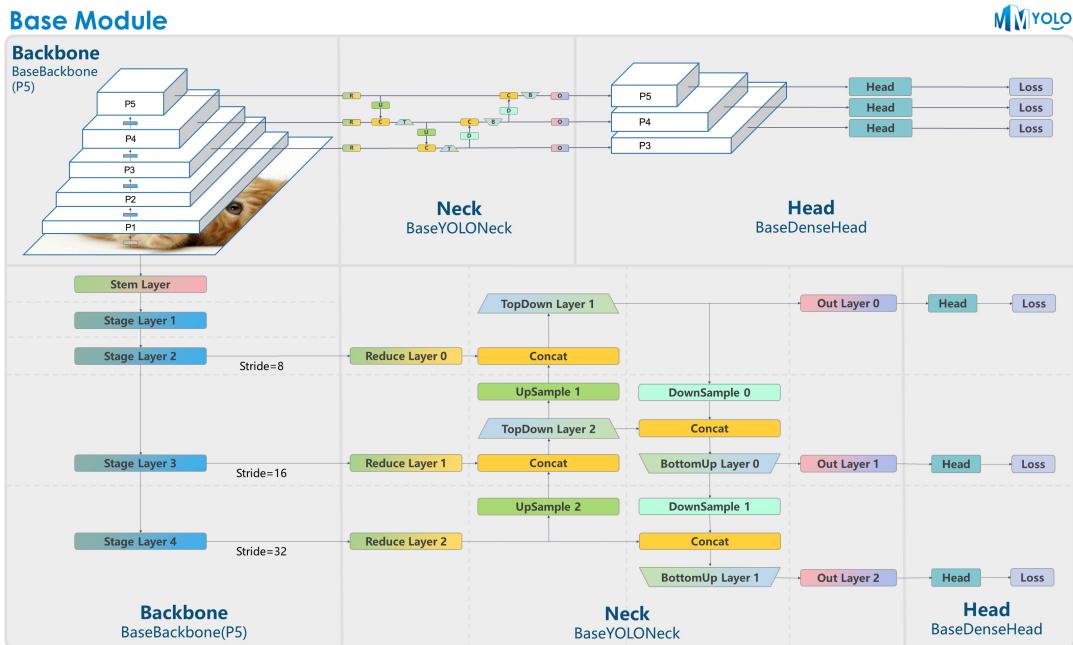
Rysunek 4.12: Przykładowe wyniki detekcji obiektów

Tabela 4.3 przedstawia wyniki dokładności dla poszczególnych klas, średnia ang. *Average Precision* - AP wynosi 79.1%, natomiast prędkość przetwarzania obrazu wyniosła 64 FPS.

Tabela 4.3: Wyniki dokładności dla poszczególnych klas

Klasa	aeroplane	bicycle	bird	boat	bottle
Dokładność [%]	88	84	77.1	71.5	45.7
Klasa	bus	car	cat	chair	cow
Dokładność [%]	87.6	85.3	89.3	64.1	86.3
Klasa	diningtable	dog	horse	motorbike	person
Dokładność [%]	70.2	88.7	88.2	86.2	77.7
Klasa	pottedplant	sheep	sofa	train	tvmonitor
Dokładność [%]	56.3	78.6	85	90	80.8

Przetestowany został jeszcze jeden z nowszych modeli do detekcji bazujący na sieciach spłotowych, a mianowicie YOLOv8 (ang. *You Only Look Once*). Jest to już ósma wersja rozwijanego projektu YOLO usprawniana przez wiele zespołów, natomiast najnowsza wersja została zaprezentowana przez grupę *Ultralytics* na początku 2023 roku. Podobnie, jak w architekturze SSD, YOLO również jest modelem jednoetapowy, który wykrywa obiekty w obrazie w jednym przebiegu. Architektura składa się z trzech głównych części: *BaseBackbone*, *Neck* oraz *Head*. Rysunek 4.13 przedstawia ogólny pogląd na całą architekturę.



Rysunek 4.13: Architektura YOLOv8 [67]

Podstawową siecią neuronową, która jest odpowiedzialna za ekstrakcję cech z obrazu jest sieć *Darknet-53* opracowana przez zespół firmy *Facebook*. Składa się ona z 53 warstw konwolucyjnych oraz warstw residual, jest ona bardzo popularna w zastosowaniach związanych z wizją komputerową. Część sieci nazwana sztyką jest tzw. łącznikiem informacji z różnych poziomów podstawy. W tej architekturze zastosowano sieć *FPN* (ang. *Feature Pyramid Networks*), która swoim kształtem przypomina piramidę, natomiast jej główną cechą charakterystyczną jest przepływ informacji w jednym kierunku z dołu do góry lub w przeciwną stronę. Z odpowiednich warstw wyciągane są mapy cech, które są analizowane przez głowę. Głowa modelu jest ostatnim etapem, który to generuje predykcje dla obiektów o różnych

rozmiarach, z tych warstw wyciągne są cztery predykcje dla każdego obiektu - prostokątny obszar, prawdopodobieństwo prawdziwego obiektu, klasyfikacja oraz ID obiektu.

Proces treningowy z użyciem już wcześniej przetrenowanego modelu jest stosunkowo prosty. W tym przypadku wykorzystano średni model o nazwie *YOLOv8m*, według dokumentacji udostępnionej przez twórców charakteryzuje się on przetwarzaniem pojedynczego obrazu w czasie poniżej 2 ms dla karty A100. Analizowane są obrazy o wielkości 640px na 640px, a więc obrazy w zbiorze często zwiększały swoją rozdzielczość na potrzeby modelu. Trening został przeprowadzony na karcie graficznej RTX3060Ti, natomiast listing 4.1 przedstawia cały skrypt potrzebny do przeprowadzenia treningu.

```
1 from ultralytics import YOLO
2
3 model = YOLO('yolov8m.pt')
4 results = model.train(data='cartons.yaml', epochs=50)
```

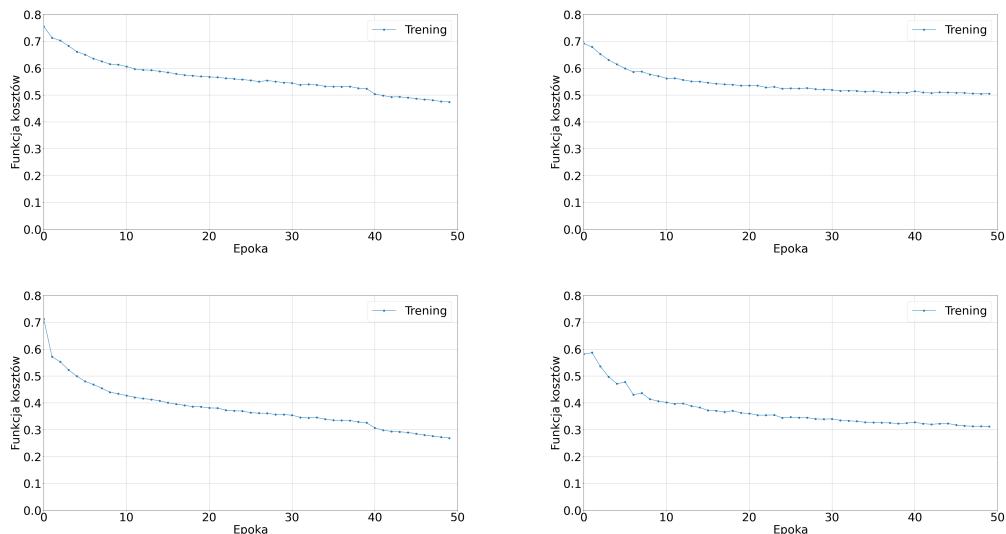
Listing 4.1: Skrypt treningowy dla architektury YOLO

Warto również objąć, jak wygląda wprowadzenie informacji gdzie znajduje się zbiór danych do przeprowadzenia treningu. Do tego celu służy plik cartons.yaml, listing 4.2 przedstawia całą zawartość pliku.

```
1 path: /Desktop/Praca magisterska/Program/data/OSCD/coco_carton/oneclass_carton/
2 train: train/images
3 val: val/images
4
5 names:
6 0: carton
```

Listing 4.2: Plik zawierający informację o zbiorze danych

Znajduje się w nim ścieżka do głównego folderu zbioru danych oraz ścieżka do zdjęć zarówno do celów treningowych, jak i walidacyjnych. Wyniki detekcji modelu YOLO są lepsze niż w przypadku modelu *SSD-300*, dlatego też przedstawione rezultaty będą już na docelowym zbiorze danych z kartonami.



Rysunek 4.14: Wyniki treningu oraz walidacji modelu *YOLOv8m* na docelowym zbiorze danych z jedną klasą.

Rysunek 4.14 przedstawia krzywe funkcji kosztów dla lokalizacji prostokątnych ramek oraz klasyfikacji obiektu zarówno podczas pętli treningowej, jak i pętli walidacyjnej. Trening przebiegł pomyślnie tym samym model osiągnął dokładność 93.5% co jest satysfakcyjnym wynikiem i dodatkowo czas potrzebny do przeanalizowania jednego obrazu na karcie RTX3060Ti wyniósł około 6.3 ms, co daje 166

FPS. Model ten posłuży w dalszej części pracy jako pomoc dla modelu do segmentacji obiektów. Rysunek 4.15 przedstawia przykładowe wyniki predykcji obiektów dla zbioru testowego.



Rysunek 4.15: Wyniki predykcji obiektów dla zbioru testowego

4.2.3. Model do segmentacji

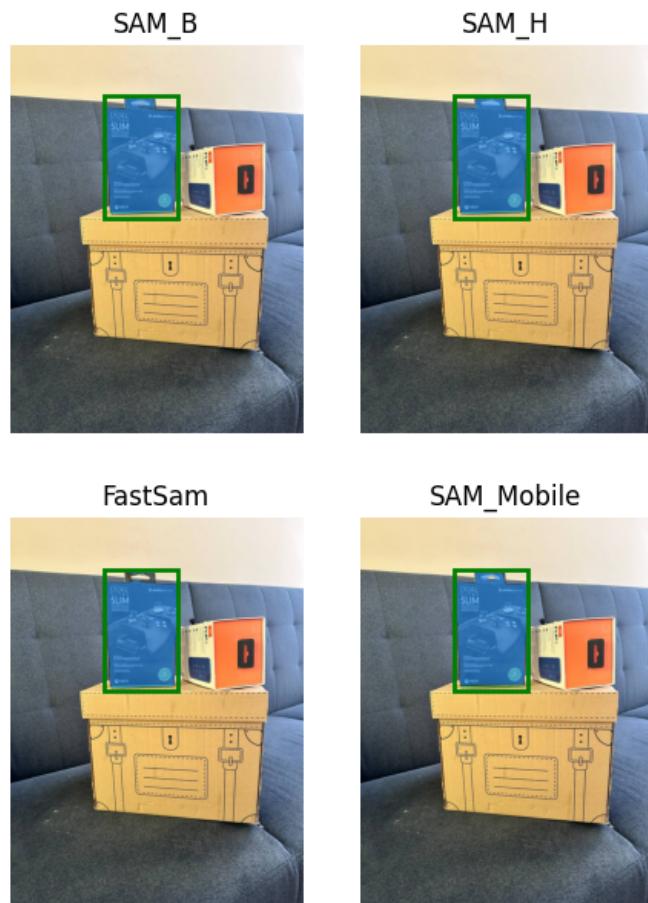
Detekcja obiektów jest jedną z metod wyszukiwania ich w obrazie, posiada ona jednak pewne wady w postaci szacunkowego obrębu obiektu. Predykcja prostokątnej ramki zależy również od sporządzonego zbioru danych, a konkretniej od *ground truth*, które to definiuje jak dokładnie obiekt został oznaczony. W przypadku potrzeby dokładnego wyodrębnienia obiektu konieczne jest wykorzystanie alternatywnego rozwiązania, takim rozwiązaniem jest właśnie segmentacja. Pozwala ona wyodrębnić obszar wyszukiwanego obiektu oddzielając przykładowo jego krawędź od pozostałego obszaru. W literaturze można znaleźć architektury oparte o sieci konwolucyjne oraz oparte na transformerach. Rozwiązania oparte na sieciach splotowych posiadają gorsze wyniki niż oparte na transformerach, dlatego też nie będą one rozważane. Stworzenie własnego rozwiązania opartego na transformerach jest korzystne dla rozważanego problemu, pozwala to na pełną kontrolę nad zawartością sieci. Niemniej jednak transformery wymagają bardzo dużych zbiorów danych, żeby otrzymać optymalne wyniki, jednocześnie wiąże się to z dużym zapotrzebowaniem na pamięć V-RAM procesora graficznego. W związku z powyższym postanowiono przetestować oraz wybrać najlepsze aktualnie dostępne rozwiązanie.

Na początku kwietnia 2023 roku przedstawiony został model *SAM*, a kilka miesięcy później dwie ulepszone wersje *FastSAM* oraz *MobileSAM*. Tabela 4.4 przedstawia porównanie liczby parametrów oraz czas wnioskowania wybranych modeli (czas zmierzony dla RTX3060Ti). Nie bez przyczyny podane zestawione zostały ze sobą modele wraz z ich liczbą parametrów oraz czasem wnioskowania. Liczba parametrów jest odpowiednikiem ilości potrzebnego miejsca w pamięci do poprawnego funkcjonowania. Jest to istotny parametr w przypadku wykorzystania modelu dla urządzeń mobilnych. W przypadku projek-

tów, gdzie jest zapotrzebowanie na szybki przepływ informacji to bardzo istotny jest czas wnioskowania, który określa potrzebną ilość czasu na przetworzenie jednej danej np. zdjęcia. Oczywiście nie jest łatwo zachować balans pomiędzy czasem wnioskowania, a otrzymaną dokładnością modelu, ale twórcy *MobileSAM* pokazali, że można otrzymać model o bardzo dobrej dokładności jednocześnie dosyć mocno kompresując. Rysunek 4.16 przedstawia porównanie predykcji maski dla obiektu objętego w ramkę prostokątną, dla poszczególnych modeli.

Tabela 4.4: Porównanie modeli *SAM-H*, *SAM-B*, *FastSAM* oraz *MobileSAM*

Model	Oryginalny SAM-H	Oryginalny SAM-B	FastSAM	MobileSAM
Liczba Parametrów	615M	136M	68M	9.66M
Czas wnioskowania	456ms	186ms	60ms	25ms



Rysunek 4.16: Wyniki predykcji obiektów dla zbioru testowego

Model *SAM_B*, *SAM_H* oraz *MobileSAM* oparte są na modelu *ViT*, a co za tym idzie są w pełni oparte na architekturze transformera. Różni ich wielkość modelu, na którym są oparte opowiadnie jest to *vit_b*, *vit_h* oraz *vit_t*. Wszystkie trzy modele korzystają z tego samego dekodera, natomiast ich enkodery

są zależne od wspomnianych wcześniej modeli *ViT* co wpływa na ilość wspomnianych w tabeli 4.4 parametrów. Jak można zauważać predykcja maski na wybranym obszarze nie różni się nadmiernie dla każdego modelu, dzięki temu można wybrać najmniejszy i jednocześnie najszybszy *MobileSAM*. Model *FastSAM* różni się od wspomnianych wcześniej modeli tym, że nie jest on oparty na *ViT*, a na *YOLOv8*. Model ten uzyskał lepsze wyniki czasowe od *SAM_B* oraz *SAM_H*, lecz gorszy od *MobileSAM*, dodatkowo jego predykcja maski jest najmniej dokładna. Niestety żaden z modeli nie oferuje możliwości treningu na własnym zbiorze danych, przez co nie można porównać uzyskanego wyniku z *ground truth*. Do dalszych rozważań został wybrany model *MobileSAM*.

4.3. Lokalizacja w przestrzeni

Lokalizacja obiektów w przestrzeni stosując do tego celu systemy wizyjne, jest bardzo przydatne m.in. w pojazdach autonomicznych oraz robotyce. Mając do dyspozycji jedną kamerę, można określić odległość obiektu od kamery oraz wielkość samego obiektu, dzięki markerom o znanej wielkości. Jest to dosyć prosta metoda niewymagająca większego nakładu obliczeniowego lub skomplikowanego algorytmu. Posiada jednak sporą wadę, a mianowicie marker musi być widoczny. W przypadku wykorzystania takiego rozwiązania dla robota przemysłowego, należy przewidzieć miejsca, w których to marker będzie potrzebny. Drugą możliwością jest zastosowanie specjalnych kamer przemysłowych, które to są przystosowane do odpowiedniego działania. Jest to jednak dosyć kosztowny sprzęt i nie każdą linię produkcyjną będzie stać na to przedsięwzięcie. Na rynku dostępna jest również kamera *Intel Realsense*. Jest to system kamer, który jest w stanie stosując odpowiednie algorytmy wyciągnąć głębie z obrazu i na jej podstawie określić odległości.

Do określenia lokalizacji obiektów w przestrzeni wykorzystano system kamer, składający się z dwóch kamer internetowych Logitech C270HD. Rysunek 4.17 przedstawia kamerę, która została wykorzystana do utworzenia systemu wizyjnego. Tabela 4.5 przedstawia najważniejsze parametry kamery.



Rysunek 4.17: Kamera Logitech C270HD [69]

Tabela 4.5: Parametry kamery Logitech C270HD

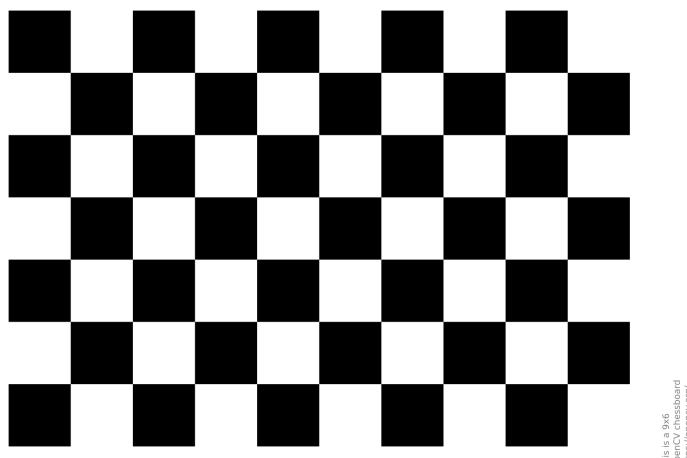
Rozdzielcość	720p (1280px x 720px) przy 30 FPS
Fokus	Stał fokus
Kąt widzenia	55°
Wielkość piksela	2.8 μ m
Wielkość sensora	3.58x2.02 mm
Ogniskowa obiektywu	4.2 mm

Wykorzystano stereowizje, dzięki której możliwa jest estymacja odległości od kamery. W tym celu został utworzony system wizyjny składający się z dwóch kamer ustawionych blisko siebie oraz zabezpieczonych przed swobodnym ruchem. Rysunek 4.18 przedstawia konstrukcję systemu wizyjnego, składającego się z dwóch kamer Logitech C270HD. Kamery zostały unieruchomione za pomocą desek i śrub, natomiast odległość między obiektywami wynosi 10cm. Stereowizja ma za zadanie wyodrębnienie informacji 3D z wielu dwuwymiarowych scen (w postaci zdjęć).



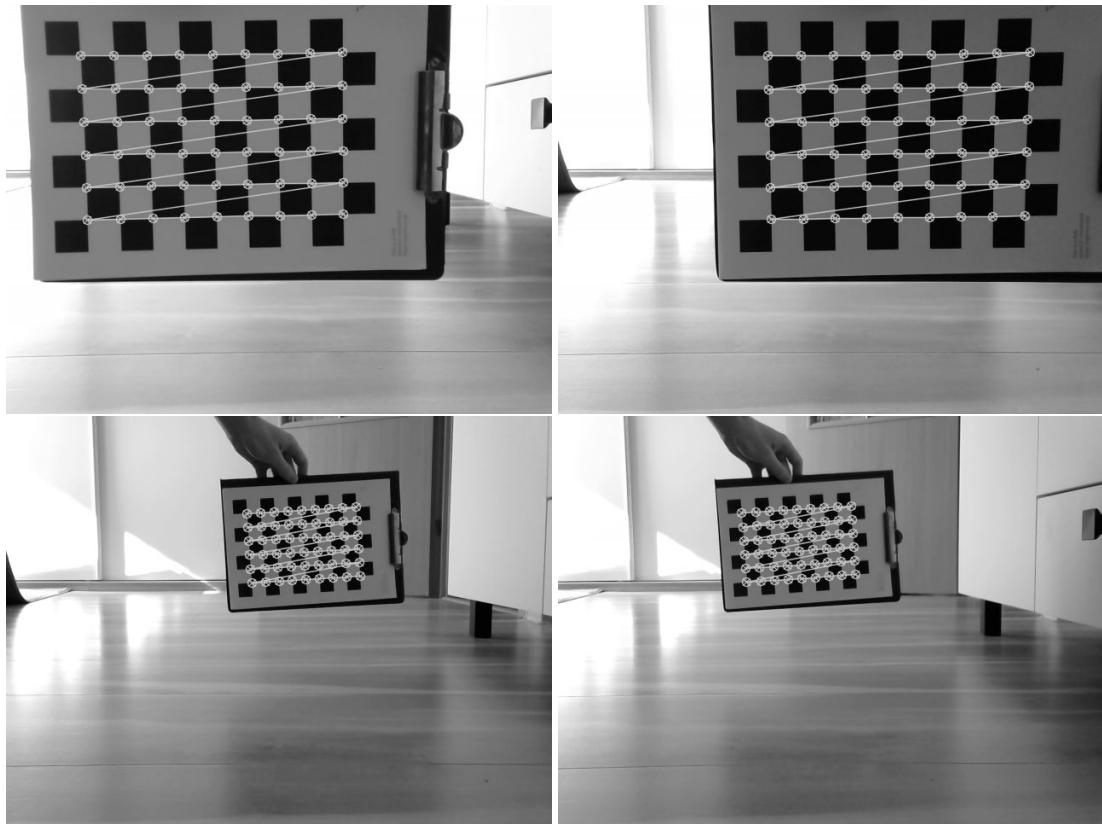
Rysunek 4.18: Opracowany system kamer do stereowizji

Przed rozpoczęciem prac nad wyodrębnianiem informacji 3D, należy odpowiednio skalibrować kamery. Kalibracja kamer stereo służy do określenia parametrów wewnętrznych i względnego położenia kamer, informacje te są wykorzystywane do rektyfikacji stereo i rekonstrukcji 3D. Dodatkowo kalibracja pozwala na zniwelowanie niekorzystnych zniekształceń. Kalibracji dokonano wykorzystując bibliotekę OpenCV oraz planszy przypominającej szachownicę rysunku 4.19.



Rysunek 4.19: Wzór szachownicy zastosowany do kalibracji [68]

W pierwszej kolejności należało wykonać zdjęcia szachownicy dla obu kamer. Do tego celu został stworzony skrypt, który w momencie wykrycia na szachownicy punktów na prostokątnym obszarze o wielkości 9 na 6 wyświetlał zdjęcia obu kamer z możliwością ich zapisania przez użytkownika. Łącznie wykonano 110 zdjęć po 55 na każdą kamerę. Rysunek 4.20 przedstawia przykładowe zdjęcia szachownicy wraz z naniesionymi punktami.



Rysunek 4.20: Przykładowe zdjęcia z obu kamer zastosowane do kalibracji

Uzyskane punkty są wykorzystywane do obliczenia nowych macierzy kamer, które pozwolą na zniwelowanie zniekształceń i przygotują system do pracy w stereowizji. Biblioteka OpenCV oferuje narzędzia umożliwiające w prosty sposób dokonanie obliczeń potrzebnych macierzy oraz wykorzystanie ich do skalibrowania kamer. Wykorzystując funkcję `calibrateCamera()` oraz `getOptimalNewCameraMatrix()` uzyskano parametry wewnętrzne lewej kamery przedstawione w tabeli 4.6, prawej kamery przedstawione w tabeli 4.8 oraz współczynniki zniekształceń odpowiednio dla lewej kamery w tabeli 4.7, jak i prawej kamery w tabeli 4.9.

Tabela 4.6: Macierz reprezentująca parametry wewnętrzne lewej kamery

1164.180	0.0	298.176
0.0	1163.316	282.914
0.0	0.0	1.0

Tabela 4.7: Współczynniki zniekształcenia lewej kamery

-0.256	6.578	0.008	0.041	65.904
--------	-------	-------	-------	--------

Tabela 4.8: Macierz reprezentująca parametry wewnętrzne prawej kamery

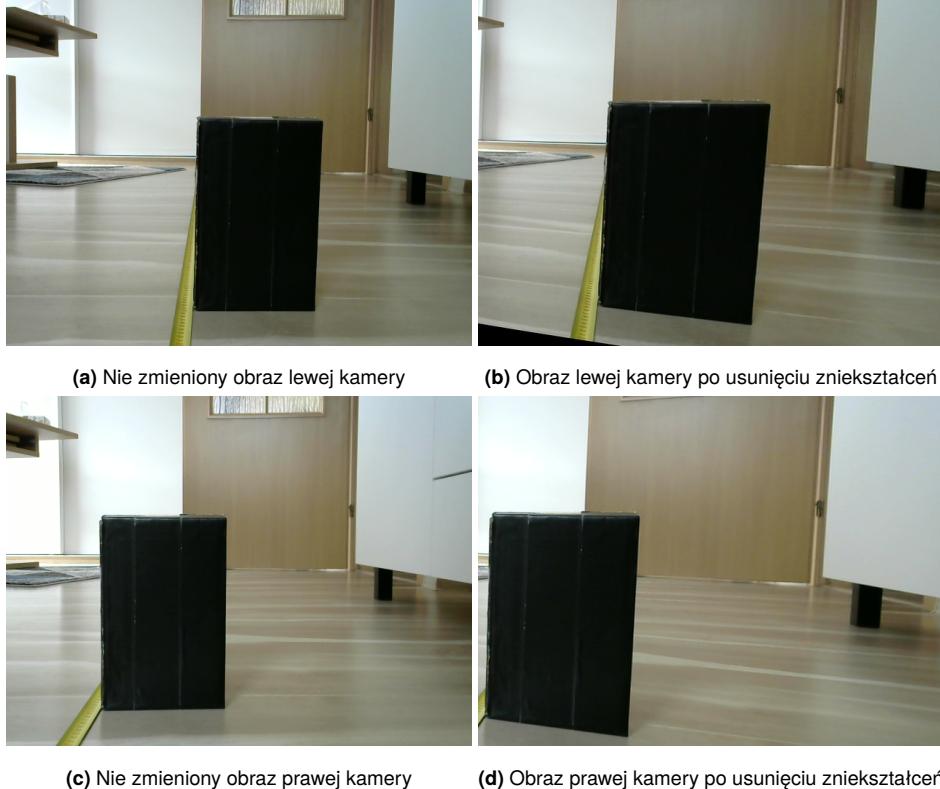
1331.344	0.0	386.429
0.0	1328.764	240.854
0.0	0.0	1.0

Tabela 4.9: Współczynniki znieksztalcania prawej kamery

0.781	-8.178	0.008	0.047	65.904
-------	--------	-------	-------	--------

Po udanej kalibracji obu kamer, należy dokonać kalibracji systemu do stereowizji w tym celu wykorzystane zostały funkcje *stereoCalibrate()* oraz *stereoRectify()*. Pierwsza z nich pozwala na m.in. wyliczenie parametrów zewnętrznych pomiędzy dwoma kamerami odpowiednio macierz rotacji i translacji. Dodatkowo została ustawiona flaga *CALIB_FIX_INTRINSIC*, która pozwala zachować istniejącą wiedzę na temat parametrów wewnętrznych oraz współczynników znieksztalceń z poprzednich funkcji. W ten sposób można znacznie przyspieszyć proces kalibracji, jak i uniknąć pogorszenia wyników. Macierz rotacji przedstawiono w tabeli 4.10, natomiast macierz translacji w tabeli 4.11. Pomimo usztywnienia konstrukcji systemu wizyjnego, należy liczyć się z nie równym umieszczeniem obiektywów, przez co obraz z obu kamer nie jest równy do horyzontu. Wspomniana wcześniej funkcja *stereoRectify()* ma za zadanie przekształcić obrazy pochodzące z dwóch kamer w taki sposób, aby linie epipolarne były równoległe do horyzontu i punkty na tych liniach były odwzorowane na tych samych rzędach. Jest to kluczowy etap w procesie kalibracji kamer przed zastosowaniem algorytmów do obliczenia mapy dysparacji.

Rysunek 4.21 przedstawia porównanie obrazu z kamer po usunięciu znieksztalceń.



Rysunek 4.21: Porównanie obrazu z kamer przed i po usunięciu znieksztalceń

Tabela 4.10: Macierz rotacji systemu wizyjnego

0.999	0.0107	-0.0511
-0.008	0.999	0.047
0.0516	-0.047	0.998

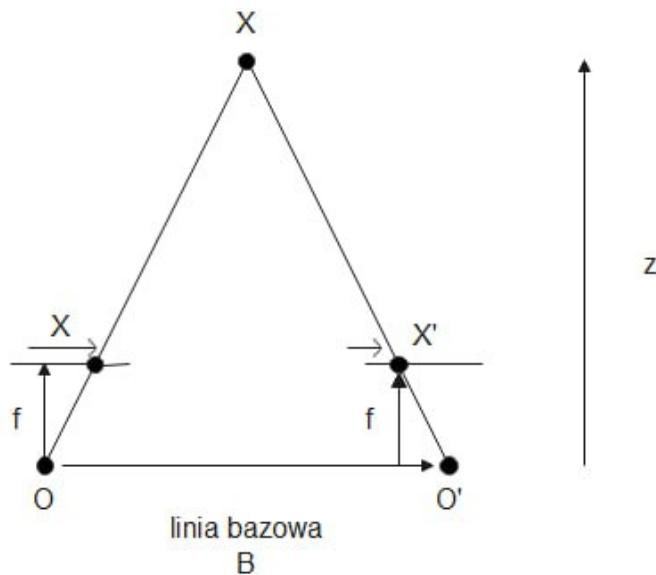
Tabela 4.11: Macierz translacji systemu wizyjnego

-5.464	-0.630	8.3647
--------	--------	--------

W celu potwierdzenia poprawnej kalibracji, można obliczyć normę wektora z tabeli 4.11, wynikiem powinna być odległość między kamerami. Zmierzona odległość między kamerami wynosi 10 cm, natomiast wyliczona:

$$\sqrt{-5.464^2 + -0.630^2 + 8.3647^2} = 10.01 [cm] \quad (4.5)$$

Po pomyślnej kalibracji systemu do działania w stereowizji należało wybrać algorytm, który dokona obliczeń mapy dysparycji, jednym z takich algorytmów jest SGBM (ang. *Semi-Global Block Matching*), czyli rozbudowany algorytm SGM (ang. *Semi-Global Matching*) [70]. Dysparycja odnosi się do odległości między dwoma odpowiadającymi sobie punktami na lewym i prawym obrazie z kamer stereo. Rysunek 4.22 przedstawia uproszczony schemat konfiguracji kamery dla punktów dysparycji. Równanie (4.6) mówi, że głębia punktu w scenie 3D jest odwrotnie proporcjonalna do różnicy odległości odpowiadających sobie punktów obrazu oraz punktu centralnego systemu wizyjnego.



Rysunek 4.22: Uproszczony schemat konfiguracji kamery dla punktów dysparycji

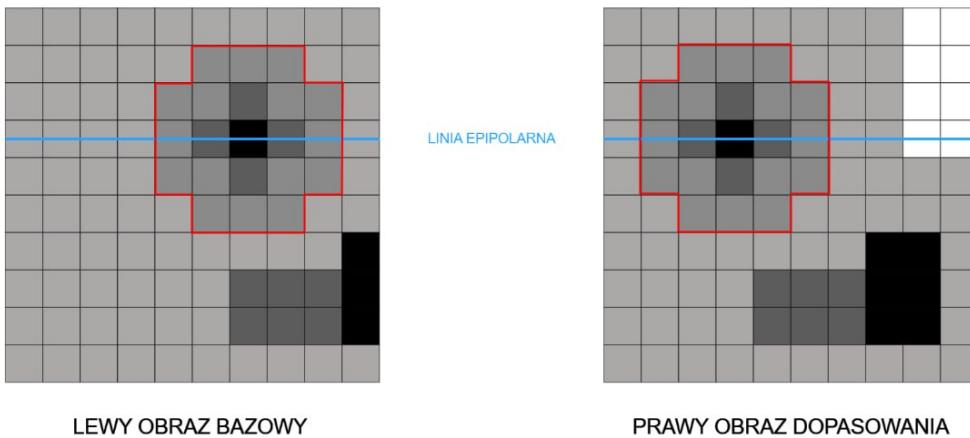
$$disparcja = x - x' = \frac{Bf}{Z} \quad (4.6)$$

gdzie:

X - punkt w przestrzeni trójwymiarowej,

x, x' - punkty na płaszczyźnie obrazu,
 f - długość ogniskowej,
 Z - gębia,
 B - odległość pomiędzy kamerami,
 O, O' - środki optyczne kamery.

Algorytm SGBM wykorzystuje obrazy pochodzące z dwóch kamer stereo i dzieli je na bloki o zdefiniowanym rozmiarze, dla każdego bloku w obrazie lewym algorytm próbuje odnaleźć odpowiadający mu blok w obrazie prawym. Uproszczony schemat działania algorytmu został przedstawiony na rysunku 4.23, wspomniany blok został zaznaczony czerwonym kolorem. Kolejnie algorytm próbuje znaleźć najbardziej spójne dopasowanie bloków między obiema kamerami na całym obszarze obrazu. Po optymalizacji algorytm wybiera dla każdego bloku lewego obrazu optymalną dysparcję, która odpowiada odległości między punktami na obu obrazach. Listing 4.3 przedstawia utworzenie instancji algorytmu SGBM z wybranymi optymalnymi parametrami następnie na tak powstałej instancji, należy uruchomić obliczenia odpowiednio wywołując komendę **stereo.compute**. Same parametry dobrane zostały metodą prób i błędów analizując wpływ zmiany wartości na prezentowaną mapę dysparycji.



Rysunek 4.23: Uproszczony schemat konfiguracji kamery dla punktów dysparycji

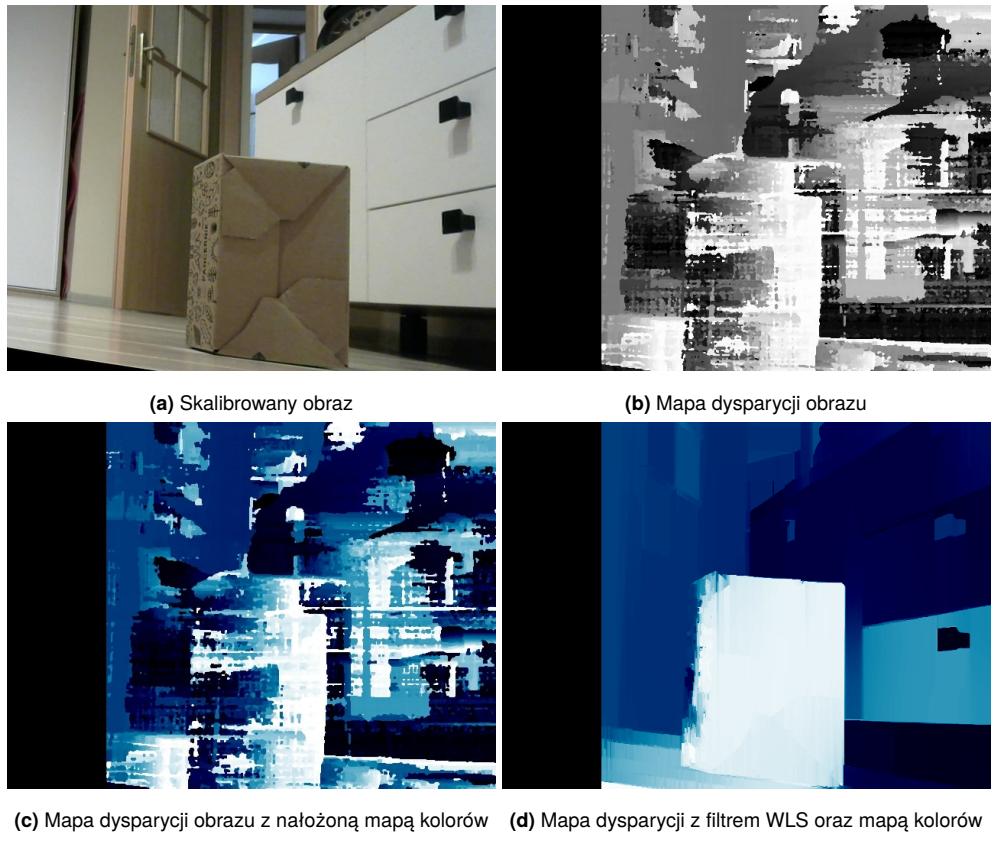
```

1 stereo = cv2.StereoSGBM_create(minDisparity = 0,
2 numDisparities = 128,
3 blockSize = 3,
4 uniquenessRatio = 5,
5 speckleWindowSize = 200,
6 speckleRange = 32,
7 disp12MaxDiff = 5,
8 P1 = 8*3*3**2,
9 P2 = 32*3*3**2,
10 mode=2) # MODE: 0-"MODE_SGBM", 1-"MODE_HH", 2-"MODE_SGBM_3WAY", 3-"MODE_HH4"

```

Listing 4.3: Skrypt przedstawiający parametry instancji StereoSGBM

Rysunek 4.24a przedstawia obraz bezpośrednio z kamery, natomiast rysunku 4.24b wynik algorytmu w postaci mapy dysparycji. Można zauważyć, że obraz dysparycji posiada dosyć sporą zaszumienie, dla tego też zastosowano filtr WLS (ang. *Weighted Least Squares*), aby lepiej dostrzec krawędzie obiektów. Wynik takiego działa przedstawia rysunek 4.24d, dla porównania została przedstawiona mapa dysparycji z taką samą mapą kolorów na rysunku 4.24c.

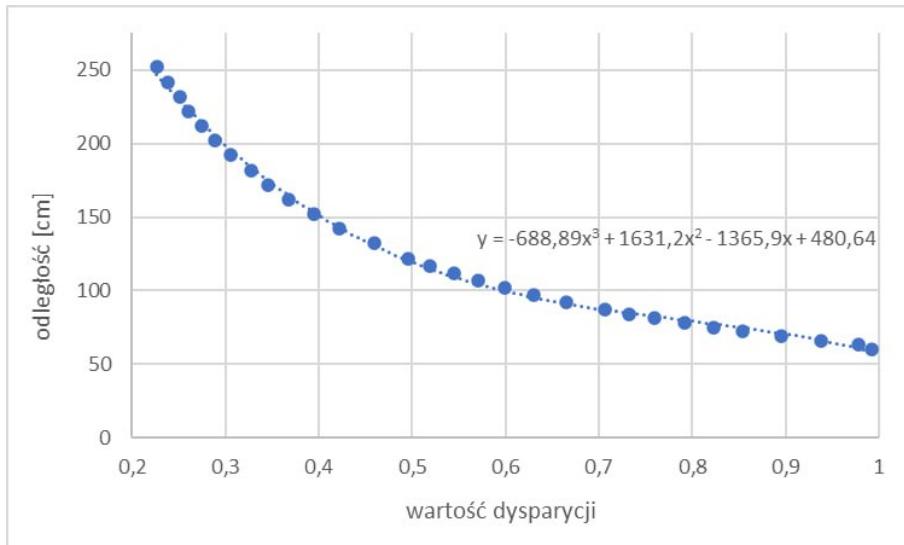


Rysunek 4.24: Porównanie niezmienionego obrazu z mapą dysparcji

Do określenia odległości od obiektu utworzono specjalną funkcję, która zamieni wartość dysparcji punktu z obrazu na wartość wyrażoną w metrach. Do tego celu przygotowano stanowisko pomiarowe, które zostało przedstawione na rysunku 4.25. Z okna przedstawiającego podgląd mapy dysparcji z filtrem WLS wybierano ręcznie punkt (środek obiektu), z którego dokonywano pomiaru średniej wartości dysparcji z obszaru o wielkości 5×5 . Obiekt został umieszczony 0,6 m od systemu wizyjnego i od tej odległości rozpoczęto pomiary przesuwając obiekt co 3 cm do odległości 88 cm, następnie co 5 cm do odległości 123 cm, następnie co 10 cm aż do 253 cm. Rysunek 4.26 przedstawia punkty pomiarowe wraz z linią trendu, która została utworzona w programie Excel wraz z równaniem wielomianowym trzeciego stopnia: $y = -688,89x^3 + 1631,2x^2 - 1365,9x + 480,64$. Takie rozwiązanie pozwala na stosunkowo precyzyjne otrzymanie wartości odległości obiektu od kamer.



Rysunek 4.25: Stanowisko do pomiaru odległości



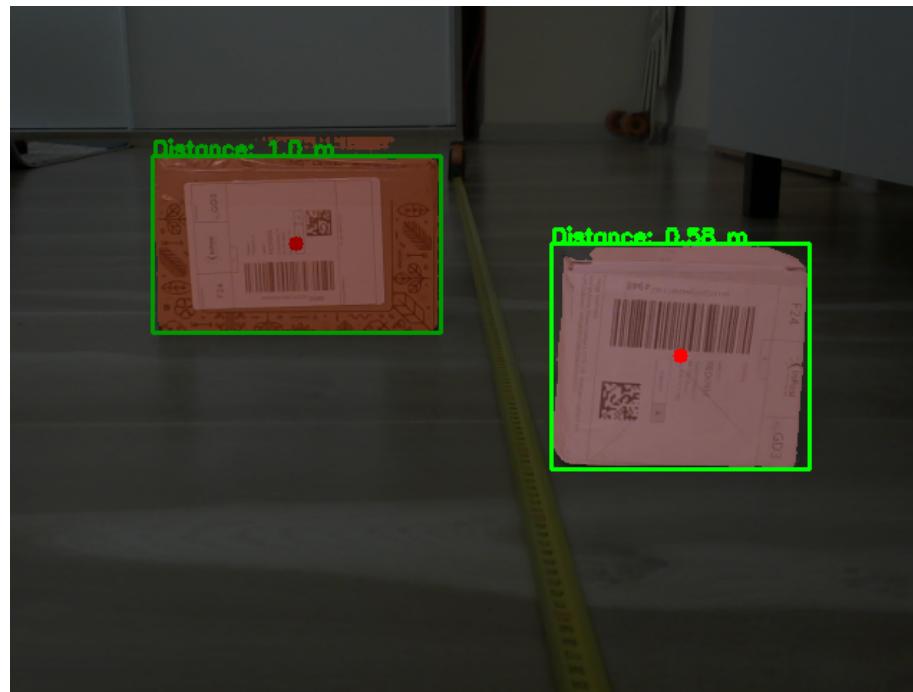
Rysunek 4.26: Wykres punktów pomiarowych oraz linii trendu

4.4. Ewaluacja algorytmów

Utworzony został skrypt sprzęgający ze sobą algorytm do detekcji, segmentacji oraz stereowizji. W pierwszej kolejności dokonywana jest kalibracja kamer, aby zniwelować zniekształcenia w obrazie, a następnie dokonać kalibracji stereo oraz wyliczeniu mapy dysparacji. W następnej kolejności inicjowany jest model do detekcji z wytrenowanymi wagami oraz model do segmentacji. W pętli uruchamiany jest odczyt obrazu z kamer, a następnie analizowany przez model YOLO. W momencie zlokalizowania w obrazie minimum jednego pudełka (pewność wykrycia musi wynieść co najmniej 51%) uruchamiany jest model SAM do wyodrębnienia maski segmentacyjnej w obszarze prostokątnej ramki wyodrębniając tym samym cały obiekt. Wykorzystując model do detekcji można w prosty sposób uzyskać punkt na wykrytym obiekcie w postaci środka ramki prostokątnej. W ten sposób określana jest odległość do wykrytego punktu. Rysunek 4.27 przedstawia wynik połączenia modeli i algorytmu lokalizacji w przestrzeni. Rzeczywista odległość obiektu z prawej strony wynosi 0,6 m, natomiast obiektu z lewej strony wynosi 0,9 m. Wyniki są stosunkowo dokładne w odległości do 120 cm, natomiast powyżej tej wartości estymacja odległości jest rozbieżna od wartości rzeczywistych. Działanie modelu do detekcji jest bardzo szybkie dzięki czemu nie ogranicza on wydajności całego systemu. Wraz z modelem do segmentacji uzyskują wynik w okolicach 15 FPS, natomiast w połączeniu z algorytmem do estymacji odległości wydajność drastycznie spada, ponieważ jest on obliczany przez CPU. Zweryfikowano dokładność estymowanej odległości, względem wartości zmierzonej wyniki tego eksperymentu zaprezentowano w tabeli 4.12, natomiast średni błąd wyniósł 16,52 cm. Wyniki pokazują, stosunkowo wysoką dokładność na małej odległości, natomiast im większa odległość tym gorsze wyniki estymacji.

Tabela 4.12: Wartości estymowane oraz zmierzone między kamerą, a obiektem

Wartość zmierzona [cm]	Wartość estymowana [cm]	Błąd [%]
60	62	3.33
65	60	7.69
70	64	8.57
75	70	6.66
80	75	6.25
85	78	8.23
80	81	10.00
100	87	13.00
110	95	13.63
115	97	15.65
120	101	15.83
130	110	15.38
140	120	14.28
150	130	13.33
160	136	15.00
170	141	17.05
180	155	13.88
190	140	26.31
200	178	11.00



Rysunek 4.27: Końcowy rezultat działania wszystkich algorytmów

5. PODSUMOWANIE

W niniejszej pracy zostało przeprowadzone badanie na temat segmentacji instancyjnej obiektów zapakowanych oraz określeniem ich lokalizacji w przestrzeni. Temat ten w ostatnim czasie staje się coraz popularniejszy, o czym może świadczyć liczba publikacji na temat detekcji, jak i segmentacji obiektów.

W pracy w ramach eksperymentu utworzono zbiór danych oznaczonych obiektów zapakowanych, który posłużył jako zbiór treningowy ostatecznego modelu. Przedstawiono również podejście do tworzenia własnych modeli opartych na pomysłach grup badawczych oraz sposoby doboru narzędzi programistycznych, jak i algorytmów. Dzięki tym informacjom stworzono pierwsze modele do klasyfikacji obrazów wyciągnięto wnioski, które pomogły w dalszym projektowaniu. Przedstawione zostały metody usprawniające trening modeli oraz radzenie sobie z powstałymi problemami np. przeuczenia się modelu. Zaprojektowano i wytrenowano model do detekcji obiektów bazujący na sieciach konwulcyjnych porównano jego wyniki z gotowymi rozwiązaniami oraz wyciągnięto potrzebne wnioski. Przedstawiono najnowsze rozwiązania do segmentacji obiektów oraz porównano działania tych modeli. Spośród wszystkich wybrano ten, który sprawdzał się najlepiej.

Przeprowadzono testy modelu do detekcji YOLOv8 na docelowym zbiorze danych oraz przedstawiono wyniki dokładności, jak i czasu inferencji dla pojedynczego zdjęcia, które wyniosły odpowiednio 93.5% oraz 6.3ms. Wykorzystano najlepszy model do detekcji i połączono jego rozwiązanie z modelem do segmentacji, aby uzyskać docelową segmentację instancyjną. Dużą rolę w projekcie spełniał model *MobileSAM*, który posiadał wysoką dokładność nałożonej maski, jednocześnie zdobywając znaczną przewagę czasu przetwarzania nad oryginalnym *SAM*. Model do segmentacji został wytrenowany przez twórców na ponad 10 milionach obrazów, dzięki czemu nie wymagał dodatkowego treningu. Uniemożliwiło to jednak weryfikację dokładności nałożonych masek z *ground true* zbioru obiektów zapakowanych.

Ostatnim etapem było przygotowanie systemu wizyjnego do lokalizacji obiektów w przestrzeni. Do tego celu wykorzystano algorytm *SGBM*, który jest powszechnie wykorzystywany w sterowizji. Po odpowiedniej kalibracji udało się uzyskać zamierzony efekt obliczania odległości obiektów od kamery. Przeprowadzone testy pokazały, że na bliskiej odległości do około metra dokładność obliczeniowa jest zadowalająca, natomiast na dalsze odległości powstają przekłamania.

Rezultaty przeprowadzonego badania osiągnęły założony cel oraz uzyskane wyniki są zadowalające. Potwierdzają, że segmentacja instancyjna obiektów w połączeniu z lokalizacją w przestrzeni, może być wykorzystywana w przemyśle, jak i autonomicznych pojazdach.

Potencjalne dalsze pracę nad rozwojem skupiają się nad możliwościami optymalizacyjnymi algorytmów m.in. lokalizacja obiektów w przestrzeni. Algorytm został uruchomiony na CPU, a istnieje możliwość modyfikacji skryptu tak, aby mógł on zostać uruchomiony na GPU. W ten sposób można uzyskać lepsze wyniki nie tylko w postaci mniejszych szumów mapy dysparcji, ale również wydajniejszego wyliczania potrzebnych danych. Można rozbudować skrypt wyliczający odległość obiektu od kamery na podstawie kilku punktów.

SPIS RYSUNKÓW

2.1 Diagram prezentujący połączenia między warstwami w necognitron [1]	12
2.2 Zastosowanie filtra na tensorze trzeciego rzędu	13
2.3 Splot macierzy dwuwymiarowej I stosując filtr K	14
2.4 Porównanie wielkości obrazu z wypełnieniem oraz bez	14
2.5 Działanie funkcji <i>max-pooling</i>	15
2.6 Prównanie działania optymalizacji funkcji z użyciem dwóch różnych wartości kroku uczenia	16
2.7 Porównanie działania standardowego <i>SGD</i> oraz z dodanym momentem	18
2.8 Uproszczony schemat perceptronu	20
2.9 Przebieg funkcji <i>sigmoid</i> oraz jej pochodnej	21
2.10 Przebieg funkcji <i>tanh</i> oraz jej pochodnej	21
2.11 Przebieg funkcji <i>ReLU</i> oraz jej pochodnej	22
2.12 Przykładowa architektura modelu transformera	23
2.13 Schemat blokowy pojedynczej uwagi	24
2.14 Model <i>Vision Transformer</i> [30]	25
2.15 Porównanie rodzajów segmentacji [43]	27
2.16 Schemat działania konwolucyjnego enkodera i dekodera	28
2.17 Porównanie obrazu przed i po wielowarstwowej konwolucji	28
2.18 Wyniki działania modelu <i>SAM</i> , (a) przedstawia oryginalny obraz, (b) przedstawia wynik działania z wybranym punktem, (c) przedstawia wynik działania w obszarze prostokąta, (d) przedstawia wynik działania po wprowadzeniu komendy "rear wheel"	29
2.19 Rodzaje dystorsji obrazu	30
2.20 Na rysunku (a) górný obiektý nie jest równoległy do płaszczyzny obrazu, który to powoduje efekt stycznych, natomiast rysunek (b) przedstawia zniekształconą płaszczyznę kolorem czerwonym	31
3.1 Fragment interfejsu narzędzia <i>TensorBoard</i>	35
3.2 Wykorzystanie różnych <i>frameworków</i> w dołączonych repozytoriach do artykułów [64] . .	36
4.1 Przykładowe obrazy ze zbioru danych	38
4.2 Zakłada strony <i>Roboflow</i> do etykietowania danych	39
4.3 Strona główna z otwartym projektem na <i>Google Colab</i>	40
4.4 Przykładowe obrazy wraz z ich etykietami z trzech zbiorów testowych.	41
4.5 Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora <i>SGD</i> dla zbioru danych <i>CIFAR</i>	42
4.6 Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora <i>SGD</i> dla zbioru danych <i>OXFORD</i>	43
4.7 Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora <i>Adam</i> .	44
4.8 Wykresy przedstawiające zmiany wartości dokładności modelu dla optymalizatora <i>Adam</i> .	45
4.9 Diagram prezentujący ogólne założenia architektury <i>SSD-300</i>	47
4.10 Przykładowe nałożenie prostokątów zgodnie z danymi dla warstwy <i>Conv9_2</i>	48
4.11 Krzywa funkcji kosztów dla poszczególnych epok	50
4.12 Przykładowe wyniki detekcji obiektów	50

4.13 Architektura <i>YOLOv8</i> [67]	51
4.14 Wyniki treningu oraz walidacji modelu <i>YOLOv8m</i> na docelowym zbiorze danych z jedną klasą.	52
4.15 Wyniki predykcji obiektów dla zbioru testowego	53
4.16 Wyniki predykcji obiektów dla zbioru testowego	54
4.17 Kamera Logitech C270HD [69]	55
4.18 Opracowany system kamer do stereowizji	56
4.19 Wzór szachownicy zastosowany do kalibracji [68]	56
4.20 Przykładowe zdjęcia z obu kamer zastosowane do kalibracji	57
4.21 Porównanie obrazu z kamer przed i po usunięciu zniekształceń	58
4.22 Uproszczony schemat konfiguracji kamery dla punktów dysparcji	59
4.23 Uproszczony schemat konfiguracji kamery dla punktów dysparcji	60
4.24 Porównanie niezmienionego obrazu z mapą dysparcji	61
4.25 Stanowisko do pomiaru odległości	61
4.26 Wykres punktów pomiarowych oraz linii trendu	62
4.27 Końcowy rezultat działania wszystkich algorytmów	63

SPIS TABEL

4.1 Wyniki dla poszczególnych modeli oraz optymalizatorów	46
4.2 Dane potrzebne do prognozy detekcji dla poszczególnych warstw	48
4.3 Wyniki dokładności dla poszczególnych klas	51
4.4 Porównanie modeli <i>SAM-H</i> , <i>SAM-B</i> , <i>FastSAM</i> oraz <i>MobileSAM</i>	54
4.5 Parametry kamery Logitech C270HD	56
4.6 Macierz reprezentująca parametry wewnętrzne lewej kamery	57
4.7 Współczynniki zniekształcenia lewej kamery	57
4.8 Macierz reprezentująca parametry wewnętrzne prawej kamery	58
4.9 Współczynniki zniekształcenia prawej kamery	58
4.10 Macierz rotacji systemu wizyjnego	59
4.11 Macierz translacji systemu wizyjnego	59
4.12 Wartości estymowane oraz zmierzone między kamerą, a obiektem	63

BIBLIOGRAFIA

- [1] Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biol. Cybernetics* 36, 193–202 (1980). <https://doi.org/10.1007/BF00344251>
- [2] HUBEL, D. H., WIESEL, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3), 574–591. <https://doi.org/10.1113/jphysiol.1959.sp006308>
- [3] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
- [4] Fang, X. (2017). Understanding deep learning via backtracking and Deconvolution. *Journal of Big Data*, 4(1). <https://doi.org/10.1186/s40537-017-0101-8>
- [5] Albawi, S., Mohammed, T. A., & Al-Zawi, S. (2017). Understanding of a convolutional neural network. 2017 International Conference on Engineering and Technology (ICET). <https://doi.org/10.1109/icengtechol.2017.8308186>
- [6] Wu, J. (2017). Introduction to convolutional neural networks - NJU. <https://cs.nju.edu.cn/wujx/paper/CNN.pdf>
- [7] O'Shea, K., & Nash, R. (2015, December 2). An introduction to Convolutional Neural Networks. arXiv.org. <https://arxiv.org/abs/1511.08458>
- [8] Xiao, H., Rasul, K., & Vollgraf, R. (2017, September 15). Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. arXiv.org. <https://arxiv.org/abs/1708.07747>
- [9] Raschka S. & Mirjalili V.: Python Uczenie maszynowe. Wydanie drugie, Helion, Gliwice 2019.
- [10] Liu (Hayden) Yuxi: Python Uczenie maszynowe w przykładach. Tensorflow 2, PyTorch i scikit-learn. Wydanie trzecie, Helion, Gliwice 2022.
- [11] Howard J., Gugger S. & Chintala S.: Deep learning dla programistów. Budowanie aplikacji AI za pomocą fastai i PyTorch. Helion, Gliwice 2021.
- [12] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.
- [13] Fenner E. M.: Uczenie maszynowe w pythonie dla każdego. Helion, Gliwice 2020.
- [14] Tabor J., Śmiejka M., Struski Ł., Spurek P. & Wołczyk M.: Głębokie Uczenie Wprowadzenie. Helion, Gliwice 2022.
- [15] Schaul, T., Zhang, S., & LeCun, Y. (2013, May). No more pesky learning rates. In International conference on machine learning (pp. 343-351). PMLR.
- [16] Schaul, T., & LeCun, Y. (2013). Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. arXiv preprint arXiv:1301.3764.
- [17] Katanforoosh & Kunin, "Initializing neural networks", deeplearning.ai, 2019. <https://www.deeplearning.ai/ai-notes/initialization/index.html> (data dostępu 15.07.2023)

- [18] Qian, Q., Jin, R., Yi, J., Zhang, L., & Zhu, S. (2015). Efficient distance metric learning by adaptive sampling and mini-batch stochastic gradient descent (SGD). *Machine Learning*, 99, 353-372.
- [19] Konečný, J., Liu, J., Richtárik, P., & Takáč, M. (2015). Mini-batch semi-stochastic gradient descent in the proximal setting. *IEEE Journal of Selected Topics in Signal Processing*, 10(2), 242-255.
- [20] Liu, C., & Belkin, M. (2018). Accelerating sgd with momentum for over-parameterized learning. arXiv preprint arXiv:1810.13395.
- [21] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks : the official journal of the International Neural Network Society*, 12 1, 145-151 .
- [22] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
- [23] Debes, K., Koenig, A., & Gross, H. M. (2005). Retrieved from <https://www.brains-minds-media.org/archive/151/supplement/bmm-debes-suppl-050704.pdf> (data dostępu 03.07.2023)
- [24] Wang, C.-F. (2019). Retrieved from <https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484> (data dostępu 13.07.2023)
- [25] Labach, A., Salehinejad, H., & Valaee, S. (2019). Survey of dropout methods for deep neural networks. arXiv preprint arXiv:1904.13310.
- [26] Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415.
- [27] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.
- [28] Sun, C., Myers, A., Vondrick, C., Murphy, K., & Schmid, C. (2019). Videobert: A joint model for video and language representation learning. In *Proceedings of the IEEE/CVF international conference on computer vision* (pp. 7464-7473).
- [29] Sun, C., Shrivastava, A., Singh, S., & Gupta, A. (2017). Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision* (pp. 843-852).
- [30] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.
- [31] Lecun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 541-551.
- [32] J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database,"*2009 IEEE Conference on Computer Vision and Pattern Recognition*, Miami, FL, USA, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.
- [33] Krizhevsky, A. (2009) Learning Multiple Layers of Features from Tiny Images. Technical Report TR-2009, University of Toronto, Toronto.

- [34] F. Zhuang et al., „A Comprehensive Survey on Transfer Learning,” in Proceedings of the IEEE, vol. 109, no. 1, pp. 43–76, Jan. 2021, doi: 10.1109/JPROC.2020.3004555.
- [35] S. J. Pan and Q. Yang, „A Survey on Transfer Learning,” in IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 10, pp. 1345–1359, Oct. 2010, doi: 10.1109/TKDE.2009.191.
- [36] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional Neural Networks. Communications of the ACM, 60(6), 84–90. <https://doi.org/10.1145/3065386>
- [37] Parkhi, O. M., Vedaldi, A., Zisserman, A., & Jawahar, C. V. (2012, June). Cats and dogs. In 2012 IEEE conference on computer vision and pattern recognition (pp. 3498–3505). IEEE.
- [38] Nilsback, M., & Zisserman, A. (2008). Automated Flower Classification over a Large Number of Classes. 2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing, 722–729.
- [39] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
- [40] Image Segmentation: The basics and 5 Key Techniques. <https://datagen.tech/guides/image-annotation/image-segmentation/> (data dostępu 02.03.2023)
- [41] Salamati, N., Larlus, D., Csurka, G., & Süsstrunk, S. (2014). Incorporating near-infrared information into semantic image segmentation. arXiv preprint arXiv:1406.6147.
- [42] Wu, Z., Shen, C., & Hengel, A. V. D. (2016). Bridging category-level and instance-level semantic image segmentation. arXiv preprint arXiv:1605.06885.
- [43] Kirillov, A., He, K., Girshick, R., Rother, C., & Dollár, P. (2019). Panoptic segmentation. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 9404–9413).
- [44] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). <https://doi.org/10.1109/cvpr.2015.7298965>
- [45] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1–9).
- [46] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In Medical Image Computing and Computer-Assisted Intervention—MICCAI 2015: 18th International Conference, Munich, Germany, October 5–9, 2015, Proceedings, Part III 18 (pp. 234–241). Springer International Publishing.
- [47] Zhao, H., Shi, J., Qi, X., Wang, X., & Jia, J. (2017). Pyramid scene parsing network. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2881–2890).
- [48] He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. In Proceedings of the IEEE international conference on computer vision (pp. 2961–2969).
- [49] Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., ... & Girshick, R. (2023). Segment anything. arXiv preprint arXiv:2304.02643.

- [50] He, K., Chen, X., Xie, S., Li, Y., Dollár, P., & Girshick, R. (2022). Masked autoencoders are scalable vision learners. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 16000-16009).
- [51] Prescott, B., & McLean, G. F. (1997). Line-based correction of radial lens distortion. Graphical Models and Image Processing, 59(1), 39-47.
- [52] Burns, J. B., Hanson, A. R., & Riseman, E. M. (1986). Extracting straight lines. IEEE transactions on pattern analysis and machine intelligence, (4), 425-455.
- [53] Liu, Y., Cheng, D., Wang, Q., Hou, Q., Gu, L., Chen, H., ... & Wang, Y. (2021). Optical distortion correction considering radial and tangential distortion rates defined by optical design. Results in Optics, 3, 100072.
- [54] Park, J., Byun, S. C., & Lee, B. U. (2009). Lens distortion correction using ideal image coordinates. IEEE Transactions on Consumer Electronics, 55(3), 987-991.
- [55] Juyang, Weng., P., Cohen., M., Herniou. (1992). Camera calibration with distortion models and accuracy evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 14(10):965-980. doi: 10.1109/34.159901
- [56] Bailey, D. G. (2002, December). A new approach to lens distortion correction. In Proceedings image and vision computing New Zealand (Vol. 2002, pp. 59-64).
- [57] Camera Modeling: Exploring Distortion and Distortion Models, Part I by Jeremy Steward. <https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i> (data dostępu 25.08.2023)
- [58] CUDA C++ Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (data dostępu 25.07.2023)
- [59] C++ documentation. <https://cppguide.readthedocs.io/en/latest/> (data dostępu 20.07.2023)
- [60] Python documentation. <https://docs.python.org/3.10/> (data dostępu 20.07.2023)
- [61] TensorFlow v2.10.1 documentation. https://www.tensorflow.org/versions/r2.10/api_docs/python/tf (data dostępu 10.08.2023)
- [62] Keras documentation. <https://keras.io/about/> (data dostępu 15.08.2023)
- [63] PyTorch 2.0 documentation. <https://pytorch.org/docs/stable/index.html> (data dostępu 15.08.2023)
- [64] Papers with codes trends. <https://paperswithcode.com/trends> (data dostępu 20.08.2023)
- [65] Yang, J., Wu, S., Gou, L., Yu, H., Lin, C., Wang, J., ... & Li, X. (2022). SCD: A stacked carton dataset for detection and segmentation. Sensors, 22(10), 3617.
- [66] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016). Ssd: Single shot multibox detector. In Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14 (pp. 21-37). Springer International Publishing.
- [67] https://github.com/open-mmlab/mmyolo/blob/main/docs/en/recommended_topics/model_design.md (data dostępu 19.09.2023)

- [68] <https://github.com/opencv/opencv/blob/4.x/doc/pattern.png> (data dostępu 22.09.2023)
- [69] <https://www.logitech.com/pl-pl/products/webcams/c270-hd-webcam.960-001063.html> (data dostępu 06.10.2023)
- [70] Hirschmüller, H. (2007). Stereo Processing by Semi-Global Matching and Mutual Information.

A. INSTRUKCJA DLA UŻYTKOWNIKA

A.1. Wymagania systemowe

W celu poprawnego funkcjonowania skryptów oraz potrzebnych bibliotek, użytkownik musi posiadać komputer z systemem operacyjnym *Linux*. Aby uzyskać wydajność zbliżoną do przedstawionej w pracy zaleca się posiadanie karty graficznej opracowanej przez firmę *NVIDIA*.

A.2. Język programowania oraz biblioteki

W celu ułatwienia korzystania z utworzonych skryptów wykorzystano *Conda*, czyli system do zarządzania środowiskami i pakietami *Python*. Należy zainstalować *Conda* zgodnie z instrukcją zamieszczoną na stronie producenta. Wersja *Pythona* oraz biblioteki zostały zapisane do pliku *conda_environment.yml* w folderze "*Install_Conda*". Znajduje się tam również plik *conda_env.py*, który pobierze potrzebne repozytorium z platformy *Github*. Po zakończeniu działania skryptu środowisko powinno być aktywne. Listing A.1 przedstawia komendę, jaką należy wpisać w konsoli.

```
1 python conda_env.py && conda activate stereovision
```

Listing A.1: Komenda inicjalizująca utworzenie środowiska *stereovision*

Istnieje prawdopodobieństwo wystąpienia błędu podczas uruchomiania właściwego skryptu, dlatego też zalecane jest wprowadzenie zmiany w pliku "MobileSAM/mobile_sam/predictor.py". Listing A.2 przedstawia zmianę, jaką należy dokonać w 10 linijce.

```
1 from mobile_sam.modeling import Sam      -->      from .modeling import Sam
```

Listing A.2: Zmiana w pliku *predictor.py*

A.3. Wykonywanie zdjęć do kalibracji

W celu wykonania zdjęć do kalibracji należy wykorzystać planszę zisaną jako "Plansza.png". Skrypt *taking_image.py* został przygotowany do zapisywania zdjęć gotowych do kalibracji. Bardzo ważne jest zweryfikowanie czy wyświetlany obraz jest z odpowiedniej kamery, innymi słowy lewy obraz musi pochodzić z lewej kamery. Aby zapisać zdjęcia należy umieścić planszę przed obiema kamerami, wtedy nastąpi wykrycie punktów i możliwość zapisania obrazu. Obraz można zapisać wciskając przycisk **s**, natomiast zakończyć program można za pomocą przycisku **q**. Aby uzyskać poprawne wyniki kalibracji zaleca się wykonanie minimum 20 zdjęć.

A.4. Uruchomienie głównego skryptu

Po wykonaniu wszystkich poprzednich kroków należy uruchomić skrypt *main.py*. W pierwszej kolejności należy sprawdzić czy wyświetlany obraz jest z odpowiedniej kamery, tak jak to miało miejsce w sekcji A.3. Jeżeli kamery zostały poprawnie przypisane, to można kontynuować działanie skryptu. Efektem powinny być dwa okienka z wyświetlonym obrazem z obu kamer, na których wyświetlona zostanie ramka wokół wykrytego obiektu wraz z maską segmentacyjną oraz odległością od kamer. Aby zakończyć działanie programu, należy wcisnąć przycisk **q**.