

Politechnika Warszawska

W Y D Z I A Ł M E C H A N I C Z N Y
E N E R G E T Y K I I L O T N I C T W A



Instytut Techniki Lotniczej i Mechaniki Stosowanej

Praca dyplomowa inżynierska

na kierunku Automatyka i robotyka
w specjalności Robotyka

System wykrywania twarzy

Maksymilian Szymanowicz

Numer albumu 285742

promotor
dr inż. Andrzej Kordecki

Warszawa, 2021

Streszczenie

Główny cel pracy to stworzenie opartego o metody głębokiego uczenia systemu detekcji twarzy, który potrafi z dużą dokładnością wykryć twarze na kolejnych klatkach nagranego już filmu, nawet wtedy, gdy twarze są niewyraźne, częściowo zasłonięte i jest ich kilkadziesiąt. Końcowy wynik detekcji dla danego obrazu składa się z prostokątnych ramek zawierających twarze wraz z przypisanymi im przez sieć wskaźnikami pewności.

Zaproponowano autorską metodę detekcji twarzy za pomocą zmodyfikowanej wersji sieci Faster R-CNN. Wyniki detekcji są bardzo dobre ($mAP @IoU=50: 0.8965$) i cel pracy został osiągnięty. Zmiany polegają przede wszystkim na zastąpieniu nowszymi metodami z sieci Mask R-CNN kilku niedoskonałych procedur oryginalnie stosowanych w Faster R-CNN. Na tak skonstruowanej sieci, przeprowadzono trzy eksperymenty, różniące się głównie rodzajami augmentacji. W każdym z nich, z nadatkiem spełniono założenia sformułowane w celu pracy. Zmodyfikowana sieć Faster R-CNN potrafi z dużą dokładnością wykryć twarze na obrazach, także wtedy, gdy są niewyraźne, częściowo zasłonięte lub występują w dużej ilości. Wykrywane są twarze o różnych rozmiarach, często w nietypowych pozach. Każdej z twarzy przypisywane są wskaźnik pewności, konfiguracyjnie większy od 0.7, oraz prostokątna ramka zdefiniowana czterema współrzędnymi.

Do treningu i ewaluacji działania sieci użyto wymagającej bazy danych Wider Face. Składa się ona z 16098 poprawnie anotowanych obrazów, wśród których wiele zawiera dziesiątki twarzy niewielkich rozmiarów, zdarzają się też twarze niewyraźne, bądź w niestandardowej pozycji oraz okluzje.

Najlepsze osiągnięte wyniki podczas eksperymentów to:

- mAP @IoU=50:	area=all	maxDet=100	0.8965
- mAP @IoU=50:5:95:	area=all	maxDet=100	0.5977
- mAR @IoU=50:	area=all	maxDet=100	0.9066
- mAR @IoU=50:5:95:	area=all	maxDet=100	0.6173

Zwracając uwagę na dużą liczbę twarzy o wysokiej skali trudności zawartych w bazie Wider Face, są to bardzo dobre rezultaty.

Oprócz wykrywania twarzy na pojedynczych obrazach, możliwa jest taka konwersja filmu, że po jej zakończeniu na kolejnych klatkach do twarzy przypisane są prostokątne ramki wraz ze wskaźnikami pewności, a ścieżka dźwiękowa zostaje zachowana. Ponadto, do użytku oddany jest tryb wykrywania twarzy na klatkach przechwytywanych z podłączonej do komputera kamery. Dla użytkowników posiadających najlepsze karty graficzne, detekcja twarzy może odbywać się niemal w czasie rzeczywistym.

Słowa kluczowe: sieci, neuronowe, uczenie, maszynowe, detekcja, twarzy, rcnn, fast, faster, mask, konwolucyjna, resnet, fpn, pyramid, roi, rois, pooling, roialign, anchors, widerface, confidence, funkcja straty

Abstract

The main purpose of the paper is to create a face detection system utilizing deep-learning methods. This system should be able to detect faces on consecutive frames of recorded video with high accuracy, even when there are several dozen of blurred, occluded faces in the picture. End result for the given image consists of bounding box predictions and confidence scores.

A custom face detection method utilizing a modified version of Faster R-CNN is proposed. Results are very good ($mAP @IoU=50: 0.8965$) and the design requirements are fulfilled. The changes mainly concern the replacement of a few imperfect procedures originally used in Faster R-CNN with newer methods from Mask R-CNN. Different types of augmentation were used in each of the three experiments conducted on the modified Faster R-CNN. In all of the experiments, requirements were exceeded. Modified Faster R-CNN can detect multiple blurred, occluded faces with high accuracy, even when their sizes are different or they are in an atypical position. Each displayed predicted bounding box is defined by four coordinates and confidence score, set in configuration section to be greater than 0.7.

Challenging Wider Face dataset was used for training and evaluation of network performance. It consists of 16098 images with correctly annotated ground truth boxes. Many pictures contain several dozen small faces, blurred or atypically positioned faces are also a common occurrence.

The best results achieved during the experiments:

- mAP @IoU=50:	area=all	maxDet=100	0.8965
- mAP @IoU=50:5:95:	area=all	maxDet=100	0.5977
- mAR @IoU=50:	area=all	maxDet=100	0.9066
- mAR @IoU=50:5:95:	area=all	maxDet=100	0.6173

Taking into consideration the large number of faces rated as difficult in Wider Face dataset, these results are very good.

Besides face detection in single images, a video conversion mode is offered. Neural network overlays prediction boxes and confidence scores on each frame, while an audio track is preserved. In addition, a camera face detection mode is also introduced. Frames from the camera connected to PC are captured, then processed accordingly by neural network. Users utilizing cutting-edge GPUs might even enjoy near real-time face detection.

key words: neural, networks, machine, learning, face, fast, faster, mask, rcnn, detection, roi, rois, pooling, roialign, fpn, pyramid, widerface, convolutional, anchors, confidence, loss function

Oświadczenie autora (autorów) pracy

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
data

.....
podpis autora (autorów) pracy

Oświadczenie

Wyrażam zgodę / nie wyrażam zgody^{*1} na udostępnianie osobom zainteresowanym mojej pracy dyplomowej. Praca może być udostępniana w pomieszczeniach biblioteki wydziałowej. Zgoda na udostępnienie pracy dyplomowej nie oznacza wyrażenia zgody na jej kopowanie w całości lub w części.

Brak zgody nie oznacza ograniczenia dostępu do pracy dyplomowej osób:

- reprezentujących władze Politechniki Warszawskiej,
 - członków Komisji Akredytacyjnych,
 - funkcjonariuszy służb państwowych i innych osób uprawnionych, na mocy odpowiednich przepisów prawnych obowiązujących na terenie Rzeczypospolitej Polskiej, do swobodnego dostępu do materiałów chronionych międzynarodowymi przepisami o prawach autorskich.
- Brak zgody nie wyklucza także kontroli tekstu pracy dyplomowej w systemie antyplagiatowym.

.....
data

.....
podpis autora (autorów) pracy

*1 - niepotrzebne skreślić

Spis treści

1. Wstęp	6
1.1 Wprowadzenie	6
1.2 Cel pracy	8
1.3 Przegląd metod	9
1.3.1 Region Based Convolutional Neural Networks (R-CNN)	9
1.3.2 You Only Look Once (YOLO)	11
1.3.3 Inne architektury	12
2. Podstawy teoretyczne	13
2.1 Opis najczęściej stosowanych funkcji aktywacji	13
2.2 Definicja, sposoby treningu i główne rodzaje sieci	16
2.3 Algorytm wstecznej propagacji błędów	19
2.4 Konwolucyjne sieci neuronowe	21
3. Procedura detekcji twarzy	24
3.1 Krótkie omówienie przeznaczenia podsieci i warstw	24
3.2 Sieć bazowa - ResNet-101	25
3.3 Feature Pyramid Network	27
3.4 Anchors	28
3.5 Region Proposal Network	28
3.6 Rol Pooling	30
3.7 Klasyfikator	32
3.8 Funkcja strat	33
3.9 Wczytane wagi i trening sieci	35
4. Część eksperymentalna	36
4.1 Modyfikacje bazy danych Wider Face	36
4.2 Miary oceny uczenia sieci	38
4.3 Parametry sieci	40
4.4 Eksperymenty	44
4.4.1 Badanie wpływu lustrzanego odbicia	44
4.4.2 Badanie wpływu rozmycia i obrotu obrazu	49
4.4.3 Badanie wpływu zwiększonej liczby detekcji	53
4.4.4 Kaskady Haara - porównanie	57
5. Podsumowanie	59
6. Dodatek A	60
7. Dodatek B	75

1. Wstęp

1.1. Wprowadzenie

Bardzo szybki rozwój technologii komputerowych, szczególnie w ostatniej dekadzie, pozwolił na znaczne obniżenie kosztów oraz stworzenie szeroko dostępnych, wydajnych procesorów, kart graficznych i od niedawna TPUs (ang. *Tensor Processing Units*), czyli specjalnych jednostek dedykowanych do komputacji związanych z uczeniem maszynowym. Łatwe i masowe korzystanie ze stosunkowo dużej mocy obliczeniowej spowodowało przełom w dziedzinie sztucznej inteligencji, przyczyniając się do rozwoju nowych technik, powstania bardzo interesujących prac naukowych, utworzenia intuicyjnych w użyciu bibliotek programistycznych i coraz szerszej aplikacji praktycznej projektów bazujących na uczeniu maszynowym.

Dziś nie sposób uciec od rozwiązań opracowanych z wykorzystaniem sieci neuronowych. W wielu firmach pozwoliły one znacznie podnieść efektywność, odciążyć pracowników. W życiu codziennym, intelligentne urządzenia w domu, samochodzie, kieszeni, czy na ręce podnoszą komfort oraz ułatwiają, poprawiają i skracają czas wykonywania niektórych zwykłych czynności. Wszystko wskazuje, że w przyszłości automatyczne maszyny wykorzystujące rozwiązania sztucznej inteligencji zostaną zaimplementowane w wielu miejscach, zastępując ludzi, podnosząc efektywność pracy i obniżając koszty. Sytuacja pandemiczna wyraźnie wskazuje na potrzebę takiej reorganizacji.

W ramach tej pracy poruszany jest problem z zakresu detekcji i klasyfikacji obiektów na obrazach. Dzięki milionom lat ewolucji optymalizującej ten proces, ludzki mózg jest bardzo szybki i wydajny przy rozpoznawaniu obiektów. Szczególnie dużą efektywność osiąga on w zadaniu rozróżniania twarzy, gdyż stopień rozwoju tej umiejętności mocno wpływał w przeszłości na sukces reprodukcyjny. Dla komputerów, problem detekcji i rozpoznawania twarzy jest jednak złożony i wieloetapowy, a jego rozwiązanie z użyciem uczenia maszynowego wymaga dostępu do dużej mocy obliczeniowej. Trening takiej sieci neuronowej zazwyczaj trwa około kilkudziesięciu godzin, w zależności od dostępu do zasobów obliczeniowych, bazy danych oraz typu sieci.

System detekcji twarzy to aplikacja, której celem jest osiągnięcie jak najwyższej precyzyji przy wyszukiwaniu lokacji wszystkich twarzy znajdujących się na danym zdjęciu, bądź klatce filmu, niezależnie od ich rozmiaru. Wykrycie twarzy jest dość wymagającym zadaniem, gdyż podczas tworzenia dobrej jakości systemu, należy wziąć pod uwagę czynniki, które potencjalnie mogą negatywnie wpływać na jego dokładność, np.:

- mnogość charakterystyk twarzy wśród populacji, przy czym niektóre z twarzy mogą przybierać niestandardowe wyrazy;
- jakość zrobionego zdjęcia, jego skala, a także liczba, złożoność oraz dystans fotografa do obiektów na obrazie (nie tylko twarzy);
- zmienne czynniki zewnętrzne: jasność, rozmycie obiektów podczas ruchu, cienie, kąt sfotografowania, a także okluzje twarzy (np. przez okulary, dłoń, włosy, kaptur, itp.) [1].

Systemy detekcji twarzy są m.in. implementowane w robotach reagujących na bliskość ludzi i prowadzących interakcję z nimi, wykorzystywane przy ustalaniu liczby osób przebywających

w danej chwili na terenie dużego obiektu [2], a także do monitorowania obiektów, w których jakakolwiek ludzka aktywność jest niepożądana. Ostatni cel wygodnie osiągnąć, wykorzystując autonomiczne drony, które po wykryciu człowieka natychmiast powiadamiają odpowiednie służby [3].

Po procesie detekcji twarzy, niektóre systemy szacują też płeć, wiek, kolor skóry i inne cechy, które mogą posłużyć np. w celu sprofilowania klientów. Można także zbudować system, który po detekcji twarzy, dokonuje ekstrakcji jej pojedynczych elementów (np. oczu, nosa, brody) do dalszego przetwarzania.

Najczęściej jednak, detekcja twarzy stanowi wstępny etap w procesie rozpoznawania twarzy i tu zastosowania mogą być dużo szersze:

- w automatycznych systemach instalowanych w korporacjach przy autoryzacji jedynie uprzednio wskazanych osób do przebywania w wyznaczonych pomieszczeniach [4];
- do odblokowywania smartfonów, tabletów, itp. [5];
- przy wyświetaniu spersonalizowanych reklam przez inteligentne urządzenia z kamerą lub media społecznościowe, które mają dostęp do zdjęcia twarzy użytkownika [6];
- w autonomicznych pojazdach nie tylko do uwierzytelnienia właściciela, ale też do odczytywania jego emocji, zwłaszcza gdy rolą kierowcy jest sprawowanie nadzoru nad wciąż jeszcze niedoskonałym SI [7];
- w policji i służbach specjalnych do identyfikacji podejrzanych, w urzędach do weryfikacji zdjęcia z oficjalnego dokumentu w bazie danych;
- w humanoidnych robotach skonstruowanych tak, by wyrażając emocje i rozmawiając z biologicznym człowiekiem, zachowywały się podobnie do zwyczajnych ludzi [8].

Jednym z najsłynniejszych humanoidnych robotów jest robot Sophia [8] (Rys. 1) skonstruowana przez Hanson Robotics, wykorzystująca kombinację zaawansowanej sztucznej inteligencji chatbota Alice, rozpoznawania oraz przetwarzania głosu i twarzy do konwersacji z ludźmi najczęściej o wartościach i emocjach. Pomimo zdolności do imitowania ponad sześćdziesięciu ekspresji twarzy, wielu gestów i często niezwykle trafnych odpowiedzi, a także otrzymania obywatelstwa Arabii Saudyjskiej, Sophii wciąż daleko do zaliczenia testu Turinga i osiągnięcia poziomu inteligencji zbliżonego do ludzkiego. [9]

Warto również wspomnieć o robotycznym psie Aibo (Rys. 2) masowo produkowanym przez Sony, który uczy się otoczenia, twarzy właścicieli oraz nawiązuje z nimi relacje. Używa małej kamery zamontowanej w nosie, aby odróżnić od siebie właścicieli i na każdego z nich zareagować inaczej. [10]



Rys. 1: Robot Sophia, źródło: [8]

Rys. 2: Robotyczny pies Aibo, źródło: [11]

Systemy rozpoznawania twarzy potrafią być bardzo dokładne i nie wymagają współpracy identyfikowanych ludzi. Dlatego też, mogą być w łatwy sposób użyte do naruszenia prywatności i kradzieży tożsamości. Niepokojące może być też użycie takiej technologii w państwowych systemach kredytów społecznych. [12]

1.2. Cel pracy

Głównym celem pracy jest stworzenie opartego o metody głębokiego uczenia systemu detekcji twarzy, który potrafi z dużą precyzją wykryć twarze na kolejnych klatkach nagranego już filmu, nawet wtedy, gdy twarze są niewyraźne, częściowo zasłonięte i jest ich kilkadziesiąt. System ten ma także znajdować twarze na niektórych klatkach pobieranych z kamery w czasie rzeczywistym. Końcowy wynik detekcji dla danego obrazu powinien zawierać prostokątne ramki, każda z nich zdefiniowana czterema współrzędnymi, wewnętrznej której zawiera się jedna twarz. Dodatkowo, dla każdej z ramek oblicza się wskaźnik pewności (ang. *confidence score*), który oddaje ocenę przez sieć prawdopodobieństwa, że twarz znajduje się wewnątrz tej ramki (Rys. 3).

W tym projekcie, system detekcji twarzy jest zmodyfikowaną siecią Faster R-CNN. Do treningu wykorzystano zmodyfikowaną bazę danych Wider Face o znacznym stopniu trudności [13]. Ważnymi fazami procesu budowy takiego systemu są:

- zmiana rozmiaru obrazów z bazy danych do wspólnego 1024×1024 ;
- generacja *Rois* (*Regions-of-Interest*) przy pomocy osobnej sieci RPN (*Region Proposal Network*), czyli specjalnych ramek, których mały ułamek zostanie użyty podczas treningu;
- 3-etapowy trening;
- badanie rezultatów dla otrzymanego modelu.

Wprowadzono kilka znanych z sieci Mask R-CNN usprawnień. Największe zmiany dotyczą zastosowania innej metody Rol-Pooling (*PyramidROIAlign*) oraz użycia odmiennej strategii treningowej. Ważną różnicą jest również obecność pięciu bloków sieci RPN, które przyjmują na wejściu macierze cech o różnej rozdzielczości.



Rys. 3: Otrzymany efekt końcowy – przykład, źródłowy reportaż: [14]

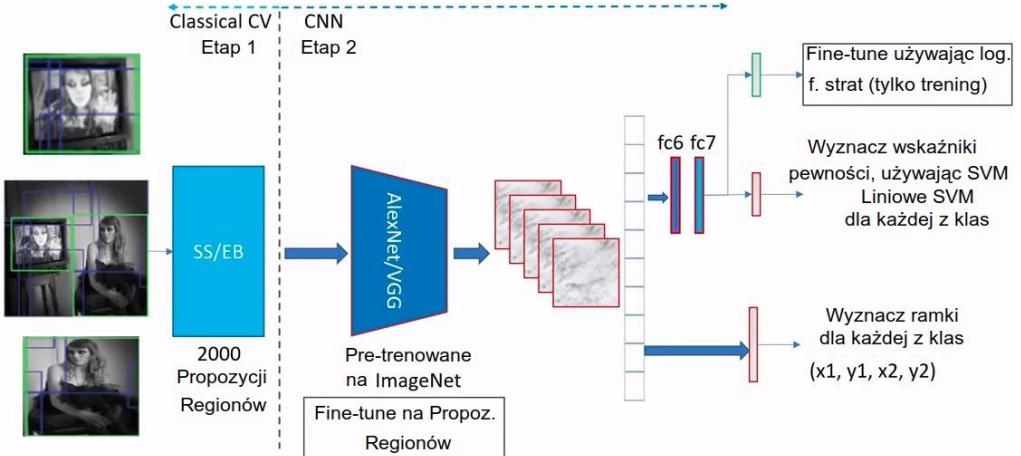
1.3. Przegląd metod

Intensywne badania w dziedzinie detekcji oraz klasyfikacji obiektów z wykorzystaniem sieci neuronowych doprowadziły do powstania wielu różnych metod rozwiązania problemu, każda z naciskiem na inne jego aspekty.

1.3.1. Region Based Convolutional Neural Networks (R-CNN)

Dla tego typu architektur, charakterystyczna jest obecność osobnego bloku odpowiedzialnego za generowanie propozycji współrzędnych ramek potencjalnie zawierających obiekt.

- 1) W pierwotnej wersji **R-CNN** [15] zastosowano *selective search algorithm*, który generuje ~2000 propozycji regionów o kształcie prostokąta i różnych rozmiarach. Następnie każda z nich jest zakrzywiana do jednakowych wymiarów (najczęściej kwadratu) i wczytywana do sieci konwolucyjnej. Tam wyznaczane są mapy cech (*feature maps*), które początkowo na warstwach w pełni połączonych poddawane są klasyfikacji. Ten odcinek sieci jest potem usprawniany przez zastosowanie na wyuczonej sieci klasyfikatora *SVM* (*Support Vector Machine*), zamiast warstw w pełni połączonych. Mapy cech przekształcone do postaci wektorów cech są klasyfikowane za pomocą wytrenowanych dla konkretnych klas *SVMs*, nadawany jest im także *confidence score*. Finalnie, procedura *greedy non-maximum suppression* używana jest do filtrowania ramek. Wadą tej wersji R-CNN jest bardzo wolna, kilkudziesięciosekundowa detekcja oraz znaczne użycie miejsca na dysku do przechowywania map cech obiektów każdej z ~2000 propozycji. Znaczącą trudnością jest także 3-stopniowy proces uczenia sieci (*fine-tuning* sieci konwolucyjnej, trening liniowych *SVMs* dla każdej z klas, trening regresora ramek) (Rys. 4).



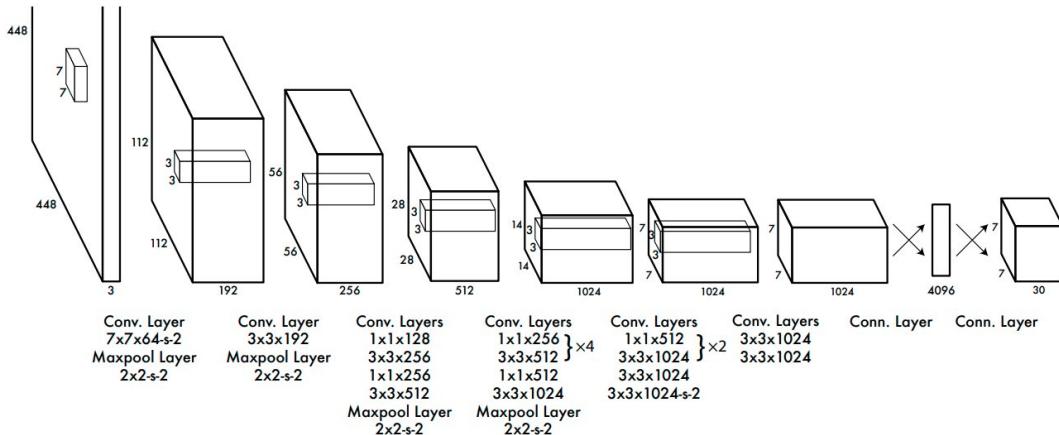
Rys. 4: Topologia sieci R-CNN, źródło: [16]

- 2) Przy **Fast R-CNN** [17], do sieci konwolucyjnej wczytywane jest pojedyncze zdjęcie, a macierz cech jest jednakowa dla wszystkich propozycji regionów wyznaczonych algorytmem *selective search*. Propozycje te są obliczane na bieżąco bez potrzeby ich przechowywania na dysku. Dodaje się też warstwę *RoI Pooling*, dzielącą fragmenty map cech odpowiadające propozycji regionów tak, by stworzyć siatkę o wymiarach $n \times n$. Z każdej komórki siatki wybierana jest maksymalna wartość i przekazywana do wyjścia z warstwy. Nie stosuje się klasyfikatorów *SVMs*. Dzięki architekturze *Fast R-CNN* (Rys. 38), trening może odbywać się jednoetapowo oraz odnawiać wagę we wszystkich warstwach. Skutkuje to zwiększoną *mAP*, dużo krótszym czasem treningu oraz czasem detekcji (VOC07) ~0.32s/im na NVIDIA K40 GPU.
- 3) **Faster R-CNN** [18] składa się z dwóch zintegrowanych modułów wykorzystujących tę samą macierz cech: modułu *Region Proposal Network (RPN)*, z którego otrzymujemy propozycje regionów oraz modułu *Fast R-CNN* dokonującego detekcji. Podczas treningu cała sieć uczy się jednocześnie, choć możliwe jest wyznaczenie strategii treningowej, gdzie w początkowych krokach nie są aktualizowane wagi niektórych warstw sieci bazowej. Faster R-CNN (Rys. 26) charakteryzuje się większą dokładnością oraz krótszym czasem detekcji ~0.2s/im na bazie danych *MS COCO 2014* przy użyciu NVIDIA K40 GPU.
- 4) **Mask R-CNN** [19] to metoda segmentacji instancji, ale jest tu uwzględniona, gdyż wprowadzone w niej zmiany mogą zwiększyć efektywność sieci *Faster R-CNN*. Detekcja za pomocą *Mask R-CNN* nie tylko klasyfikuje obiekt i wyświetla jego ramkę, ale wyznacza też jego kształt. Jest to możliwe dzięki dodaniu do architektury *Faster R-CNN* gałęzi odpowiedzialnej za wyznaczenie binarnej maski dla każdego *RoI*. Siecią bazową jest połączenie *ResNet+FPN*. Znaną z wcześniejszych wersji warstwę *RoI Pooling* ulepszono metodą *RoI Align*, która poprawia z 10% do 50% dokładność otrzymywanych masek. Trening *Mask R-CNN* wymaga jednak znajomości współrzędnych opisujących segmentację obrazu już w danych uczących.

1.3.2. You Only Look Once (YOLO)

Metody z rodziny YOLO zostały stworzone przede wszystkim z myślą o ich zastosowaniu przy detekcji w czasie rzeczywistym. Dlatego też charakteryzują się bardzo wysoką szybkością detekcji przy dość dużej precyzyji.

- 1) Schemat działania pierwotnej wersji **YOLOv1** [20] (Rys. 5) jest niezwykle prosty. Po zmianie rozmiaru obrazu, następuje jego przejście przez pojedynczą sieć konwolucyjną i wyświetlane są rezultaty powyżej pewnego *confidence score*. Dla sieci YOLOv1 występują jednak duże błędy lokalizacji oraz ograniczenie możliwości detekcji obiektów przez ilość komórek w siatce, zazwyczaj definiowanej jako 7×7 . Tylko jedna klasa może być przewidywana przez jedną komórkę. Zwykle dobiera się ilość przewidywanych ramek przez pojedynczą komórkę $B = 2$, wyznaczane jest też dla komórki C warunkowych prawdopodobieństw klas, gdzie C - ilość klas w bazie danych. Przewidywane ramki są generowane bez stosowania procedur takich jak wstępne propozycje regionów. Budowa ostatniej warstwy konwolucyjnej $7 \times 7 \times (B * 5 + C)$ jest bardzo charakterystyczna. Pojedyncza komórka, oprócz zestawu warunkowych prawdopodobieństw klas, zawiera informacje o współrzędnych oraz *confidence score* każdej z B przewidywanych ramek.



Rys. 5: Przykładowa topologia sieci YOLOv1 ($S = 7$, $C = 20$, $B = 2$), źródło: [20]

- 2) W wersji **YOLOv2** [21] jest już możliwe wykrycie nieco większej ilości obiektów, dzięki implementacji idei *anchor boxes* dla każdej z komórek. Generowanych jest k prostokątów o wymiarach uprzednio dobranych z użyciem danych uczących. Zastosowanie *anchor boxes* umożliwia ponadto dopasowanie klasy bezpośrednio do przewidywanych ramek, zamiast do komórek siatki. Podczas detekcji wykorzystywanych jest kilka rozmiarów map cech. Dodawana jest warstwa przechodnia (*passthrough layer*) o wymiarach 26×26 , która lepiej sprawdza się podczas detekcji małych obiektów niż powiązana z nią ostatnia warstwa konwolucyjna o wymiarach 13×13 . Sieć stosuje też *batch normalization* (normalizację próbki danych uczących), a także dostraja klasyfikator na obrazach o wyższej rozdzielcości (trening w wielu skalach, usunięto warstwy w pełni połączone), co zwiększa *mAP* o ~4%. Za sieć bazową (*backbone*) służy tu stale rozwijana przez autorów *DarkNet-19*.

3) **YOLOv3** [22] jest wolniejsze od YOLOv2, za to dużo lepiej radzi sobie z detekcją małych obiektów i ogółem daje dokładniejsze rezultaty. Detekcje są bowiem przeprowadzane w 3

różnych skalach. Obiekt może należeć do więcej niż jednej klasy. *Confidence scores* są obliczane przy pomocy regresji logistycznej. Jako *backbone* używana jest sieć *DarkNet-53*. YOLOv3 osiąga jednak gorsze wyniki dla średnich/dużych obiektów od swych poprzedników.

- 4) **YOLOv4** [23] to obecnie jedna z najszybszych i najbardziej dokładnych sieci używanych do detekcji obiektów. Składa się z trzech głównych modułów: *backbone* (tutaj: *CSPDarknet53*), *head*, który odpowiedzialny jest za predykcję ramek i klas oraz *neck*, czyli duża liczba warstw między dwoma modułami odpowiedzialna za przetwarzanie danych pozyskanych z map cech. YOLOv4 wykorzystuje najnowsze, dość skomplikowane strategie treningowe w ramach tzw. *Bag of freebies* oraz metody postprocessingu znacznie podnoszące dokładność w zamian za niewielkie zwiększenie czasu detekcji należące do *Bag of specials*.
- 5) **YOLOv5** [24] jest aktualnie najnowszą siecią z rodziny YOLO, znajdująca się w fazie wzmożonego rozwoju. Charakteryzuje się wysoką precyzją przy bardzo niskich czasach inferencji. Jak dotąd jednak, twórca nie opublikował artykułu zawierającego szczegóły zastosowanych metod w YOLOv5.

1.3.3. Inne architektury

- 1) **SSD: Single-Shot MultiBox Detector** [25] - użycie tej architektury eliminuje potrzebę generacji propozycji regionów za pomocą osobnego modułu, tak jak to było robione w rodzinie sieci R-CNN. Przyjmowane są obrazy zazwyczaj w rozdzielcości 300×300 lub 512×512 . Jako sieć bazową (*backbone*), wykorzystuje się najczęściej przyciętą VGG-16. Następnie dodaje się kolejne konwolucyjne warstwy o progresywnie malejącym rozmiarze. Umożliwia to dokonywanie predykcji detekcji na różnych skalach mapy cech (w przeciwieństwie do metody YOLOv1 - tam te predykcje dokonywane są dla pojedynczej skali). Do map cech o różnych rozdzielcościach przypisywane są *default boxes*, podobne do wcześniej omówionych *anchor boxes* w tym, że ich współrzędne są niezmienne dla obrazów o tej samej rozdzielcości, mają też ustalone proporcje. *Default boxes* pozwalają na łatwe otrzymanie przewidywanych ramek o różnych rozmiarach. Metodę SSD charakteryzuje balans między szybkością a dokładnością - ta ostatnia jest duża nawet dla obrazów o niewielkiej rozdzielcości.
- 2) **RetinaNet** [26] to sieć w pełni konwolucyjna (*FCN*), na którą składają się sieć bazowa *ResNet+FPN* oraz równolegle do siebie subsieć klas (*class subnet*) i subsieć ramek (*box subnet*). Wprowadzono też nową funkcję straty *focal loss* na miejsce znanej nam z wcześniejszych metod *cross-entropy loss*. Tuż po publikacji, metoda ta zarówno pod względem dokładności, jak też czasu detekcji była bardziej efektywna od najlepszych znanych wówczas wariantów *Faster R-CNN*.

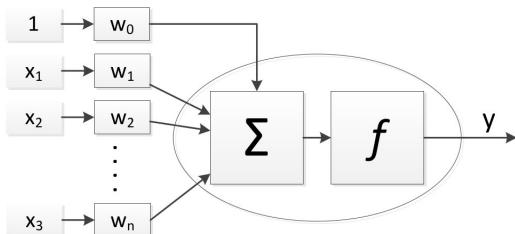
2. Podstawy teoretyczne

W tym rozdziale zostaną zdefiniowane podstawowe pojęcia z zakresu sieci neuronowych, co ułatwi zrozumienie zastosowanej do rozwiązania problemu metody *Faster R-CNN*.

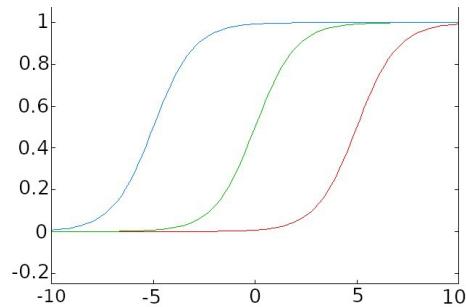
Neuron - podstawowy element składowy sztucznych sieci neuronowych (Rys. 6), nazwany tak dzięki podobieństwu jego zachowania do biologicznego neuronu, tj. przekształca sygnały wejściowe w pojedynczy sygnał wyjściowy. Matematycznie, najczęściej spotykany typ sztucznego neuronu można zdefiniować następująco:

$$y = f\left(\sum_{j=1}^n w_j x_j + b\right)$$

, gdzie $f(v)$ to funkcja aktywacji, \vec{w} to wektor wag, \vec{x} to wektor sygnałów wejściowych. Ponadto, prawie zawsze przyjmuje się $b \neq 0$. Parametr b (*bias*) umożliwia przesunięcie pionowe funkcji $v(\vec{x})$, wpływając na kształt funkcji aktywacji (np. Rys. 7). W konsekwencji, pozwala to na dużo większą elastyczność modelu, gdy ten dopasowuje się do danych. [27]



Rys. 6: Najczęściej spotykany typ neuronu



Rys. 7: Kształt sigmoidalnych funkcji aktywacji o różnych wartościach przesunięcia b

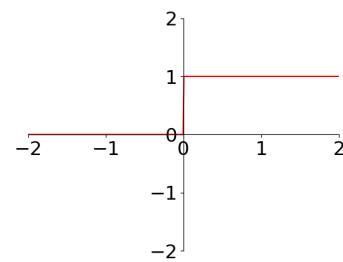
2.1. Opis najczęściej stosowanych funkcji aktywacji

W zależności od potrzeb, można dobrać funkcję aktywacji do neuronu:

- 1) Binarna funkcja skoku jednostkowego (Rys. 8)

$$f(v) = \begin{cases} 0, & v \leq 0 \\ 1, & v > 0 \end{cases}$$

Neuron zostaje aktywowany, gdy $v > 0$. Funkcja ta jest nieróżniczkowalna w 0, a $f'(v) = 0$ dla wszystkich innych wartości funkcji. Stała wartość pochodnej wyklucza zastosowanie technik optymalizacyjnych bazujących na gradiencie. Funkcja skoku jednostkowego daje dobre rezultaty przy binarnej klasyfikacji, jednak nie może być stosowana, jeśli neurony sieci reprezentują więcej niż dwie klasy.



Rys. 8: Wykres binarnej funkcji skoku jednostkowego

2) Funkcja liniowa (Rys. 9)

$$f(v) = av$$

Pochodna funkcji liniowej $f'(v) = a$, zatem tak jak przy funkcji skoku jednostkowego jest stała i nie zależy od v , zbyteczne jest więc stosowanie dla sieci w pełni liniowej metod optymalizacji bazujących na gradiencie. Ponadto, przy wielu warstwach sieci neuronowej z taką funkcją aktywacji, liniowa kombinacja funkcji liniowych pozostaje funkcją liniową - można je więc zastąpić pojedynczą warstwą. Taka sieć jest w rzeczywistości modelem regresji liniowej, a więc ma też ograniczone możliwości przy klasyfikacji złożonych danych. [28]

3) Funkcje sigmoidalne:

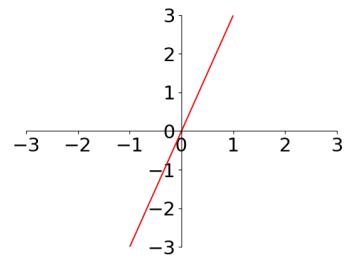
(3a) Funkcja logistyczna (Rys. 10)

$$f(v) = \frac{1}{1 + e^{-v}}, \quad f'(v) = \frac{e^{-v}}{(e^{-v} + 1)^2}$$

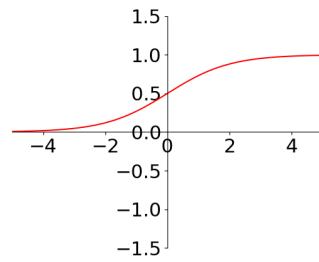
(3b) Tangens hiperboliczny (Rys. 11, 12)

$$f(v) = \tanh(v) = \frac{e^v - e^{-v}}{e^v + e^{-v}}, \quad f'(v) = 1 - \tanh^2(v)$$

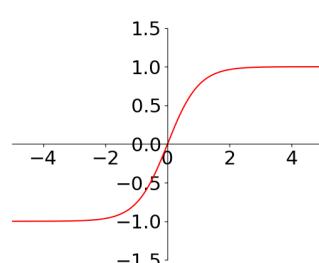
Funkcje sigmoidalne są funkcjami gładkimi, monotonicznymi, nieliniowymi i ograniczonymi. Własności pochodnej tych funkcji, która jest gładka i ograniczona (dla f. logistycznej $f'(v) \in (0, 0.25]$, dla f. tangensa hiperbolicznego $f'(v) \in (0, 1]$) zapobiegają skokowym zmianom wartości wyjściowych y oraz znacznie zmniejszają ryzyko eksplozji gradientu. Nieliniowość umożliwia efektywne stosowanie sieci wielowarstwowych. Duża stroność funkcji sigmoidalnych w sąsiedztwie zera powoduje, że drobne zmiany v skutkują dużymi zmianami wartości wyjściowych y , co finalnie skutkuje wartościami wyjściowymi położonymi bardzo blisko jednej z asymptot. Pozwala to na jednoznaczną klasyfikację. Jednakże, dla znacznie oddalonych od zera wartości v , występuje problem zanikającego gradientu, co powoduje bardzo wolne uczenie sieci lub jego brak. W zastosowaniach praktycznych funkcja tangensa hiperbolicznego jest preferowana przede wszystkim z powodu większych gradientów w sąsiedztwie zera, co przyspiesza minimalizację.



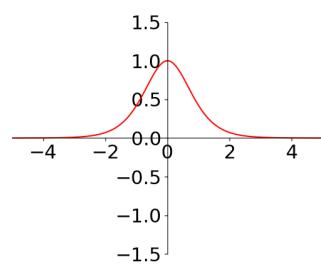
Rys. 9: Wykres funkcji liniowej



Rys. 10: Wykres funkcji logistycznej



Rys. 11: Wykres funkcji tangensa hiperbolicznego



Rys. 12: Wykres pochodnej funkcji tangensa hiperbolicznego

cją funkcji kosztów. Zastosowanie funkcji nieparzystej $f(v) = \tanh(v) \in (-1, 1)$ jest też najczęściej bardziej dogodne przy normalizacji wyjścia.

(4a) Funkcja ReLU (Rys. 13)

$$f(v) = \max(0, x), \quad f'(v) = \begin{cases} 0, & v < 0 \\ 1, & v > 0 \end{cases}$$

(4b) Funkcja przeciekającego ReLU (*leaky ReLU*)

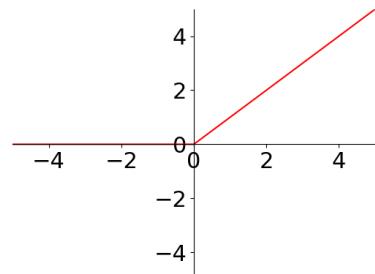
$$f(v) = \begin{cases} 0.01v, & v < 0 \\ v, & v \geq 0 \end{cases} \quad f'(v) = \begin{cases} 0.01, & v < 0 \\ 1, & v \geq 0 \end{cases}$$

Funkcja ReLU jest jedną z najczęściej używanych funkcji w konwolucyjnych sieciach neuronowych (CNN). Dodatkowo, najczęściej przyjmuje się arbitralnie $f'(0) = 0$. Chociaż matematycznie jest ona sklejeniem dwóch funkcji liniowych, przy optymalizacji bazującej na gradiencie, jej zachowanie bardziej zbliżone jest do metod nielinowych. Podczas klasyfikacji danych, obszar zostaje ograniczony wieloma hiperpłaszczyznami (Rys. 14). W przeciwieństwie do funkcji sigmoidalnych zawierających działanie wykładnicze, ReLU liczy się bardzo prosto, co znacznie przyspiesza proces uczenia. To jest właśnie główna zaleta funkcji ReLU - stosowana jako funkcja aktywacji w tysiącach neuronów w wielu warstwach znakomicie radzi sobie z zadaniami wymagającymi generalizacji. Wadą takiej sieci mogą być liczne, martwe neurony, permanentnie podające 0 jako wartość wyjściową. Częściowo rozwiązać ten problem można, zmniejszając współczynnik uczenia lub stosując przeciekające ReLU. Ta ostatnia z kolei może nie dawać jednoznacznych rezultatów dla ujemnych wartości v .

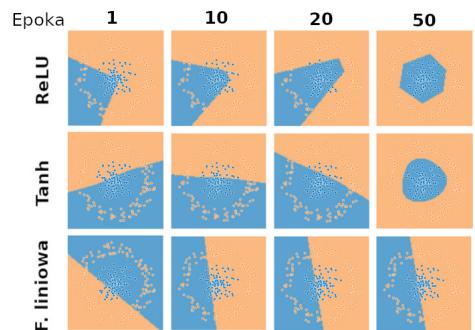
5) Funkcja Softmax

$$\sigma(\vec{v})_i = \frac{e^{v_i}}{\sum_{j=1}^K e^{v_j}}$$

Funkcja Softmax wyróżnia się tym spośród wyżej wymienionych funkcji, że stosowana jest niemal ekskluzywnie w warstwie wyjściowej (Rys. 15). Otrzymany z wcześniejszych warstw wektor logitów przekształca ona do postaci wektora dyskretnego rozkładu prawdopodobieństwa, odkodowując również przyporządkowane wynikom klasy. Po powyższej operacji, zazwyczaj oblicza się także wartość funkcji strat entropii krzyżowej (*cross-entropy loss*) i wykorzystuje ją do obliczania gradientu. Funkcji Softmax używa się tylko wtedy, gdy klasyfikowane dane/obiekty nie

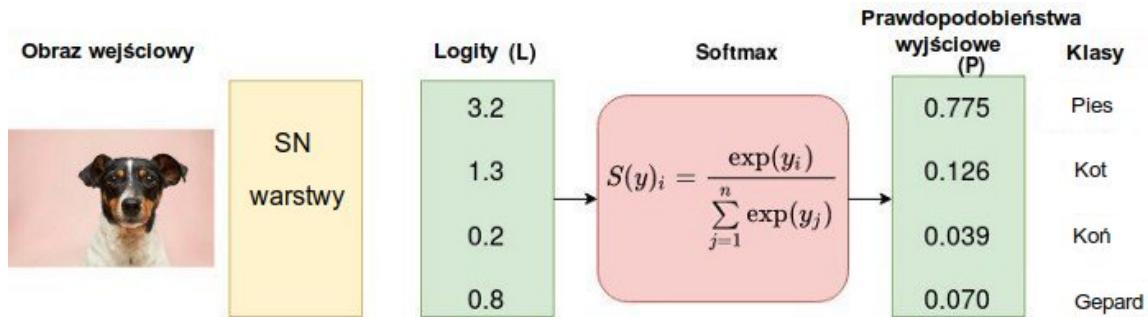


Rys. 13: Wykres funkcji ReLU



Rys. 14: Klasyfikacja danych na płaszczyźnie przez różne funkcje aktywacji, źródło: [29]

mogą należeć jednocześnie do więcej niż jednej klasy.

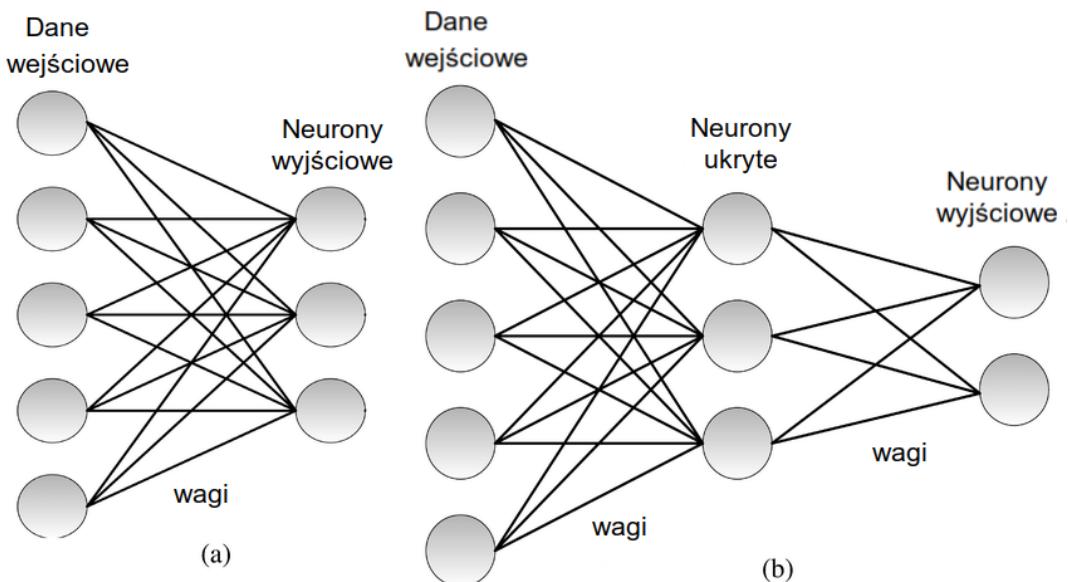


Rys. 15: Ilustracja działania warstwy Softmax, źródło: [30]

2.2. Definicja, sposoby treningu i główne rodzaje sieci

Sztuczna sieć neuronowa (*Artificial Neural Network, ANN*) w najogólniejszym znaczeniu to maszyna, którą zaprojektowano lub zaprogramowano tak, by naśladowała sposób, w jaki ludzki umysł uczy się, rozwiązuje dany problem lub ocenia otaczającą go rzeczywistość. Neurony i połączenia między nimi są zazwyczaj symulowane przy użyciu odpowiedniego oprogramowania na komputerach, zwykle z dostępem do dużej mocy obliczeniowej.[31] Jest ona bardzo potrzebna, gdyż w głębszych sieciach neuronowych, (*Deep Neural Networks, DNN*) liczba neuronów może wynosić kilkaset tysięcy lub nawet przekroczyć milion.

Sieć jednowarstwowa składa się tylko z jednej warstwy wejściowej i jednej warstwy wyjściowej. Sieć wielowarstwowa zawiera dodatkowo co najmniej jedną warstwę ukrytą (*hidden layer*) między nimi (Rys. 16). Warstwy te mogą, ale nie muszą być w pełni połączone.



Rys. 16: Schematy sieci jednokierunkowych: jednowarstwowej (a) i wielowarstwowej (b), źródło: [32]

Istnieją trzy główne sposoby treningu sieci.

Trening z nauczycielem (*supervised learning*) to technika stosowana w większości projektów używających sieci neuronowych. Posiadając bazę danych z poprawnymi wynikami Y dla przykładowych danych wejściowych X , celem jest aproksymacja mapującej funkcji $Y = \phi(X)$ tak, by z jak największą precyzją przewidyszała wyniki dla danych z tej samej kategorii. Do takiego treningu, najczęściej wykorzystywany jest algorytm wstecznej propagacji (*backpropagation*).

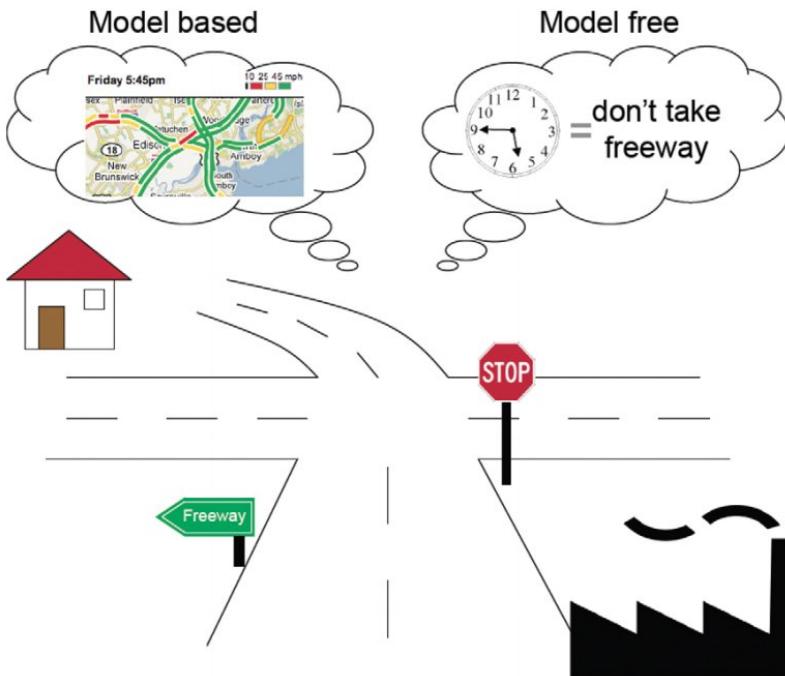
Trening bez nauczyciela - sieć nie ma dostępu do wyników dla danych uczących, więc musi sama nauczyć się ich struktury X . Technikę tą stosujemy:

- dla dużych baz treningowych, gdy ręczne anotacje są zbyt żmudne;
- do znajdowania podobnych wzorów lub podobnych przykładów w bazie danych (*clustering*). Wyniki można wykorzystać przy tworzeniu klasyfikatora do treningu z nauczycielem;
- do detekcji anomalii wśród danych. Sieć może się nauczyć np. wykrywania fałszywych transakcji;
- do oszacowania funkcji gęstości prawdopodobieństwa rozkładu danych w przestrzeni (*density estimation*).

Trening bez nauczyciela wymaga na ogół większej mocy obliczeniowej niż trening z nauczycielem. Największą jego wadą są jednak problemy z ewaluacją rezultatów po ich podzieleniu w klastry przez sieć. Nie ma możliwości ewaluacji wewnętrznej, jeśli użyta została baza danych bez anotacji, natomiast do ewaluacji zewnętrznej potrzebne jest ręczne przyporządkowanie klas, które mogą odbiegać od oczekiwanych, do klastrów, a następnie znalezienie bazy danych z odpowiednimi anotacjami do pomiaru precyzji. Sieci wykrywające obiekty trenowane bez nauczyciela są dużo mniej dokładne od trenowanych z nauczycielem, dlatego też nie mogą być stosowane tam, gdzie konsekwencje błędu mogą okazać się fatalne, np. w pojazdach autonomicznych.

Trening ze wzmocnieniem (*reinforcement learning, RL*) - podobnie jak podczas treningu bez nauczyciela, sieć uczy się w środowisku bez anotacji, jednak tu sieć (agent) uzależnia swoją reakcję na dane wejściowe od wyniku zaimplementowanej na etapie programowania funkcji nagrody (*reward function*). Celem sieci jest maksymalizacja skumulowanych nagród. Podczas projektowania sieci, ważne jest ustalenie relacji między eksploatacją, czyli podjęciem przez agenta najlepszej decyzji według znanych mu informacji, a eksploracją, czyli poszukiwaniem przez agenta nowych rozwiązań, które dostarczą mu potrzebne dane, a także mają potencjał okazać się jakościowo wyższe.[33][34] Metodę RL można podzielić na dwie grupy (Rys. 17):

- Model-based RL - agent tworzy model przyszłych decyzji na podstawie przeszłych doświadczeń. Składa się on z mapy prawdopodobieństw sukcesu w zależności od obranej przez niego ścieżki. Na tej podstawie, planuje on i wykonuje kolejne ruchy.
- Model-free RL - agent kompresuje prawdopodobieństwa do pojedynczych skalarów bez budowy modelu. Sposób ten jest dla agenta bardziej wydajny podczas podejmowania decyzji w czasie rzeczywistym, musi on jednak zakumulować bardzo wiele przeszłych doświadczeń, aby te skalary były dobrym oszacowaniem rzeczywistych prawdopodobieństw [35].



Rys. 17: Porównanie idei model-free RL do model-based RL, źródło: [35]

Ze względu na architekturę, sieci neuronowe dzieli się na dwie główne grupy.

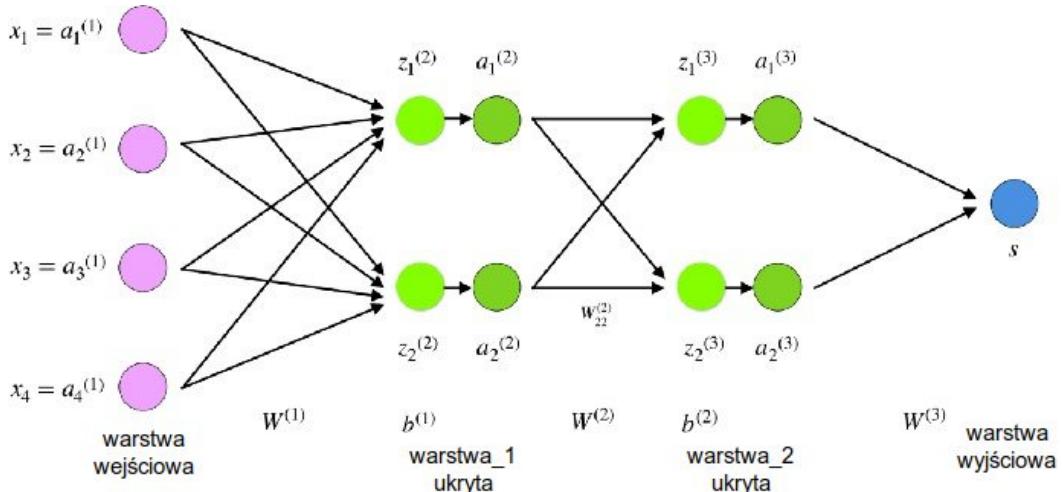
Sieci jednokierunkowe (*Feedforward Neural Networks, FFNN*) charakteryzują się przepływem informacji tylko w jednym kierunku jednocześnie, bez pętli sprzężenia zwrotnego. Są też statyczne - ich architektura podczas treningu lub inferencji nie podlega zmianom. Typowa topologia to sieć wielowarstwowa z nielinowymi funkcjami aktywacji. FFNN są przede wszystkim używane do uczenia sieci z nauczycielem. Celem jest ustalenie takich wag, by z dużą dokładnością aproksymować nieznaną funkcję przyporządkowującą wejściom sieci poprawne rezultaty. Najczęściej używa się do osiągnięcia tego efektu algorytmu wstecznej propagacji błędów (*Backpropagation*).

Sieci rekurencyjne (*Recurrent Neural Networks, RNN*) - dzięki pętlom sprzężenia zwrotnego informacje mogą przepływać w obu kierunkach. Ponadto, neurony z jednej warstwy mogą być ze sobą połączone. W przeciwieństwie do sieci FFNN, bardzo istotny jest dla rekurencji wymiar czasowy obliczeń i połączeń między neuronami. Taka konstrukcja sprawia, że sieć może przechować dane z poprzednich wejść, działając analogicznie do pamięci krótkotrwałej. Zjawisko zanikającego gradientu przy informacjach z bardzo dawnych wejść jest podobne do ludzkiego zapominania. Sygnał wyjściowy z sieci zależy więc nie tylko od obecnego sygnału wejściowego, ale też od sygnałów przeszłych. Umożliwia to sieci przewidywanie kolejnych danych wejściowych, ale też implikuje otrzymywanie różnych wyników dla tego samego sygnału wejściowego. Sieci RNN doskonale się sprawdzają przy przetwarzaniu powiązanych ze sobą sekwencji danych o nieznanej długości. W przypadku treningu z nauczycielem, stosuje się często algorytm wstecznej propagacji błędów przez czas (*Backpropagation through time*).

2.3. Algorytm wstecznej propagacji błędów

Algorytm wstecznej propagacji błędów jest bardzo popularną metodą nauki wielowarstwowej sieci FFNN (Rys. 18). Oznaczmy s jako wyjście sieci, \vec{x} jako wektor danych wejściowych, y jako znaną poprawną wartość wyjściową, W jako macierz wag, $C = c(s, y)$ jako funkcję strat, dla wagi w_{jk}^l : j to numer neuronu w danej warstwie, k to numer neuronu w poprzedniej warstwie, l to numer warstwy, m to liczba neuronów w warstwie $l - 1$.

$$\begin{aligned} a^{(1)} &= x, \quad l_0 = 2 \\ z^{(l)} &= W^{(l-1)} a^{(l-1)} + b^{(l-1)} \\ a^{(l)} &= f(z^{(l)}) \end{aligned}$$



Rys. 18: Stosowane w obliczeniach oznaczenia naniesione na przykładową wielowarstwową sieć FFNN, źródło: [35]

Do minimalizacji funkcji strat zastosowana zostanie metoda gradientu prostego (*gradient descent*), która jest zbieżna liniowo dla silnie wypukłych funkcji strat [36]. W kolejnych iteracjach, każda z wag i każde z obciążzeń będą aktualizowane według równania (1):

$$w_{jk}^{l[i+1]} = w_{jk}^{l[i]} - \eta \left(\frac{\partial C}{\partial w_{jk}^l} \right)^{[i]} \quad (1)$$

, gdzie parametr η nazywamy współczynnikiem uczenia.

$$\Delta w_{jk} = -\eta \frac{\partial C}{\partial w_{jk}^l}, \quad \Delta b_j = -\eta \frac{\partial C}{\partial b_j^l} \quad (2)$$

Wykonywane są działania na indeksach, aby znaleźć gradient z równania (2).

W równaniu (3) obliczana jest pochodna funkcji złożonej:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} \quad (3)$$

$$z_j^l = \sum_{k=1}^m w_{jk}^l a_k^{l-1} + b_j^l \quad (\text{z def.}) \quad (4)$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \quad (5)$$

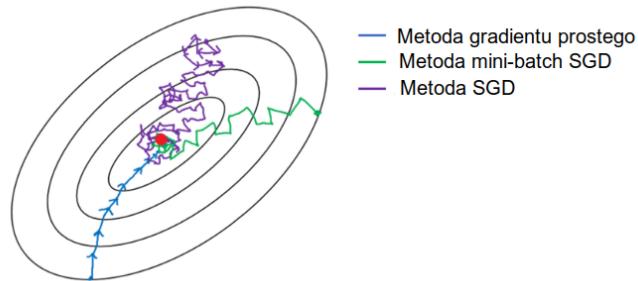
$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a_k^{l-1} \quad (6)$$

W równaniu (7) dla wygody wprowadzono oznaczenie gradientu lokalnego δ_j^l .

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \quad (7)$$

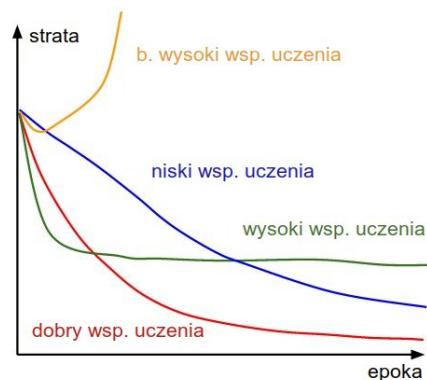
$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l * 1 = \delta_j^l \quad (8)$$

W praktyce dla dużych zbiorów treningowych, aktualizacja wag i obciążen po każdej epoce jest nieefektywna - minimalizacja funkcji strat jest bardzo powolna. Modyfikuje się więc tą metodę przez aktualizację wag po każdej, pojedynczej próbce przypadkowo wybranej ze zbioru treningowego. Taki wariant nazywany jest stochastycznym zejściem gradientu (*stochastic gradient descent, SGD*). Choć aktualizowany gradient ma wtedy większą tendencję do poruszania się po hiperpłaszczyznowym odpowiedniku linii zygzak (Rys. 19), udowodniono, że dla każdej wypukłej lub quasi-wypukłej funkcji strat, metoda SGD jest niemal zawsze zbieżna do globalnego minimum [37]. *Mini-batch SGD* jest metodą pośrednią, wagi i obciążenia są aktualizowane po $1 < M < full_batch$ próbek, gdzie M to liczba próbek w serii.



Rys. 19: Porównanie działania metod optimizacji, źródło: [38]

Należy dobrać odpowiedni współczynnik uczenia do sieci neuronowej (Rys. 20).



Rys. 20: Porównanie efektywności stałego współczynnika uczenia, źródło: [39]

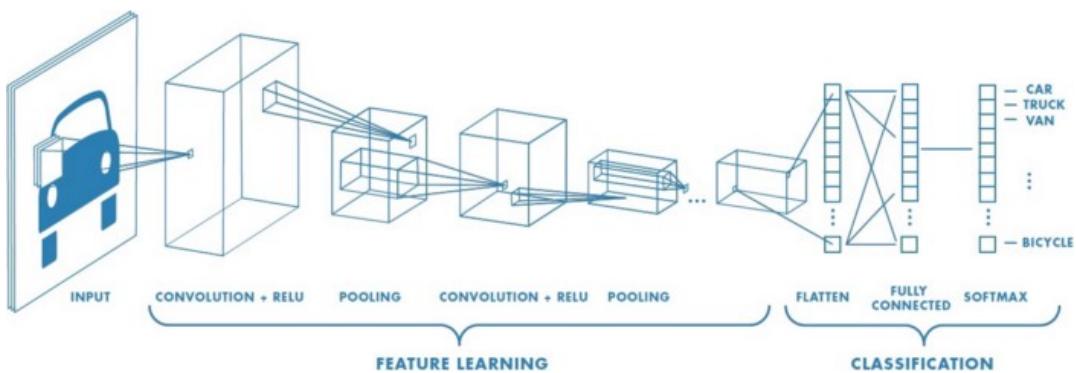
Dla uzyskania lepszych wyników, często współczynnik uczenia zmniejsza się skokowo (*step-based decay*) lub płynnie (*time-based decay*). Można także w celu przyspieszenia zbieżności wprowadzić momentum (γ):

$$\begin{cases} w_{jk}^{l[i+1]} = w_{jk}^{l[i]} - v^{[i]} \\ v^{[i]} = \gamma v^{[i-1]} + \eta \left(\frac{\partial C}{\partial w_{jk}^l} \right)^{[i]} \end{cases}$$

$\gamma \in [0, 1]$ to parametr pozwalający ustalić wpływ obecnego i przeszłych gradientów na zmianę wagi. Najczęściej przyjmuje się $\gamma = 0.9$.

2.4. Konwolucyjne sieci neuronowe

Konwolucyjne sieci neuronowe (*Convolutional Neural Networks, CNN*) to rodzina sieci stosowanych głównie do przetwarzania lub analizy obrazów, w szczególności do detekcji oraz klasyfikacji obiektów, choć używane są też do przetwarzania języka naturalnego (*NLP*), w genetyce [40], a także do predykcji na rynkach finansowych [41]. Architektura sieci konwolucyjnych (Rys. 21) została zainspirowana strukturami pól recepcyjnych, jakie istnieją w ludzkiej pierwszorzędnej korze wzrokowej [42].



Rys. 21: Typowa architektura sieci konwolucyjnej, źródło: [43]

Jeśli daną sieć konwolucyjną przystosowano do wczytywania kolorowych obrazów, to w przeciwieństwie do tradycyjnych sieci FCN, jej neurony ułożono trójwymiarowo. Obrazy wczytywane są jako mapa bitowa, gdzie kolor każdego piksela zakodowany jest w formacie RGB. W skład typowej sieci konwolucyjnej wchodzą warstwy konwolucji, aktywacji (najczęściej ReLU), poolingu.

Do operacji na obrazach, używa się konwolucji 2D. Warstwa konwolucyjna aplikuje do obrazu lub mapy cech, zestaw niewielkich przestrzennie filtrów (*kernels*), których głębokość równa jest głębokości danych wejściowych z poprzedniej warstwy. Filtry te stanowią analogię do wag obecnych w warstwach w pełni połączonych i tak też są traktowane przez algorytm wstecznej propagacji błędów, który umożliwia trening filtrów.

W konwolucji 2D (Rys. 22), filtry są trójwymiarowymi tensorami inicjalizowanymi najczęściej przypadkowo według rozkładu Gaussa. Konwolucja polega na nałożeniu filtra na dane odpowiadające jego rozmiarom i zsumowaniu iloczyń nałożonych na siebie liczb. Następnie dodawane jest obciążenie, również inicjalizowane tak jak filtry. W ten sposób, otrzymywana jest dana do kolejnej warstwy, a filtr przesuwa się wzduł wysokości oraz szerokości o zdefiniowaną wartość *Stride*. Na wyjściu z warstwy konwolucyjnej, mapa cech będzie miała mniejszą wysokość i szerokość niż poprzednio, jeśli nie zaimplementowano *paddingu*, czyli dodania sztucznych pikseli o wartości domyślnej 0 poza granicami obecnej mapy cech.

Po typowej operacji konwolucji, rozdzielcość kolejnej warstwy będzie mniejsza. Oznaczając W - rozmiar danych wejściowych na wysokość lub szerokość P - wielkość *paddingu*, S - *Stride*, K - rozmiar filtra, można łatwo obliczyć wysokość/szerokość otrzymanej mapy cech po przejściu przez warstwę konwolucyjną:

$$O = \frac{W - K + 2P}{S} + 1$$

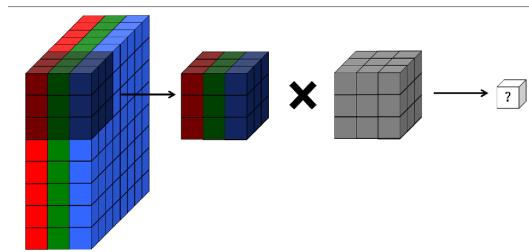
, a głębokość otrzymanej mapy cech równa jest ilości zastosowanych filtrów.

Po wytrenowaniu sieci, pierwsza warstwa konwolucyjna odpowiada za detekcję niskopoziomowych cech, takich jak proste krawędzie, okręgi. Kolejne warstwy konwolucyjne rozpoznają przy pomocy kombinacji cech niskiego poziomu coraz bardziej wysokopoziomowe cechy. Finalnie mogą to być np. tarcza zegara, czy samochód.

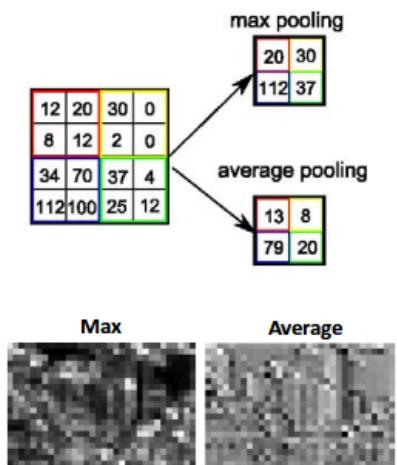
Warstwa ReLU zwykle stosowana jest po każdej z warstw konwolucyjnych. Jej celem jest wprowadzenie nieliniowości do modelu. W porównaniu do innych funkcji aktywacji, ReLU sprawia, że sieć CNN potrzebuje mniej zasobów obliczeniowych podczas treningu w porównaniu do funkcji sigmoidalnych i osiąga większą dokładność. [45]

Warstwy próbkowania (*pooling*) stosuje się między kolejnymi warstwami konwolucyjnymi, aby progresywnie zmniejszać rozmiar przestrzenny map cech. Niesie to ze sobą pozytywne skutki redukcji kosztów obliczeń i pozbycia się możliwości detekcji mało ważnych cech, co pozwala z kolei na większą generalizację (Rys. 23). Ponadto, przyczyniają się one wraz z warstwami konwolucyjnymi do umiejętności wykrywania przez sieć cech niezależnie od ich umiejscowienia na obrazie (*translation invariance*), choć większy wpływ na osiągnięcie tego efektu mają głębokość sieci CNN i augmentacja danych [46]. Dwa główne rodzaje warstw próbkowania to:

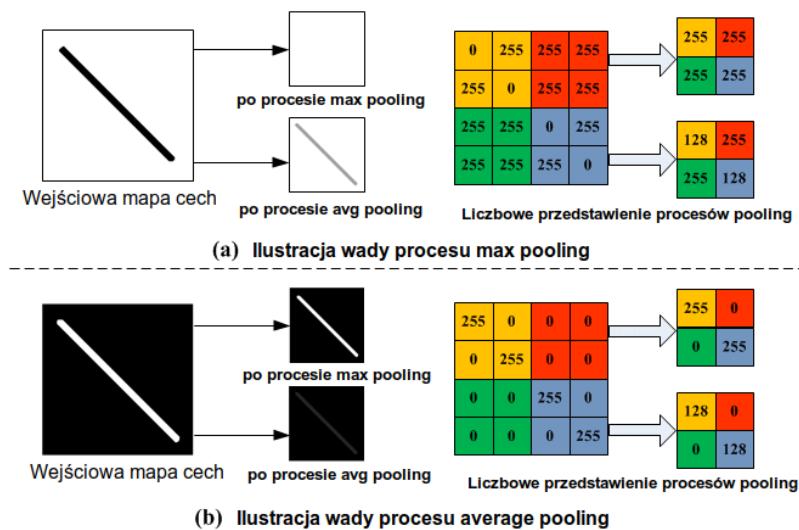
- *Max pooling* - wykorzystuje wartość maksymalną z danego regionu. Najbardziej użyteczne, gdy tło jest ciemne, a szukane cechy jaśniejsze, ostre, w dużym kontraste do otoczenia (Rys. 24). Zazwyczaj daje najlepsze rezultaty.
- *Average pooling* - wykorzystuje wartość średnią z danego regionu. Stosowane w niektórych przypadkach, gdy ważne jest przechowanie większej ilości cech kosztem ich ostrości. [47]



Rys. 22: Wizualizacja konwolucji 2D,
źródło: [44]



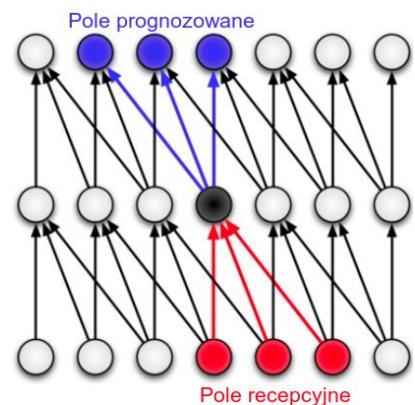
Rys. 23: Mapy cech powstałe po zastosowaniu różnych warstw pooling, źródło: [48]



Rys. 24: Działanie max i avg pooling w zależności od tła, źródło: [47]

Każdy neuron w warstwach opisywanych powyżej połączony jest jedynie z niewielkim regionem poprzedniej warstwy, który nazywany jest polem recepcyjnym (*receptive field*) dla tego neuronu. Wielkość pola recepcyjnego zależy od rozmiaru filtra. Fragment oryginalnego obrazu, który potencjalnie może wpływać na aktywację neuronu to efektywne pole recepcyjne. Prognozowane pole neuronu (*projective field*) obejmuje neurony w następnej warstwie zależne od wyników otrzymanych z rozpatrywanego neuronu (Rys. 25). [49]

Ostatnie warstwy sieci CNN to zazwyczaj warstwa spłaszczająca oraz kilka w pełni połączonych warstw, zakończonych warstwą softmax (Rys. 21), bądź alternatywą do niej. Taka topologia ostatnich warstw służy do dokonania ostatecznej klasyfikacji obiektów.



Rys. 25: Ilustracja pól oddziaływania neuronu, źródło: [49]

3. Procedura detekcji twarzy

Do detekcji twarzy zastosowano zmodyfikowaną wersję sieci Faster R-CNN (Rys. 26). Trening odbywa się na bazie danych Wider Face [13] z poprawionymi anotacjami. Wprowadzono kilka znanych z sieci Mask R-CNN usprawnień. Największe zmiany dotyczą zastosowania innej metody ROI-Pooling (*PyramidROIAlign*) oraz użycia odmiennej strategii treningowej. Ważną różnicą jest również obecność pięciu bloków sieci RPN, które przyjmują na wejściu macierze cech o różnej rozdzielcości.

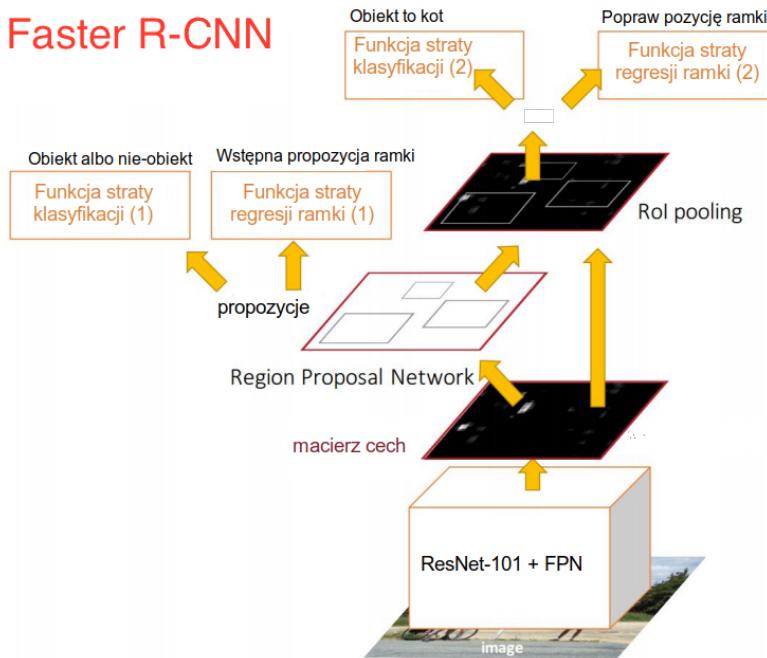
3.1. Krótkie omówienie przeznaczenia podsieci i warstw

Obrazy wczytywane są do bardzo głębokiej, rezydualnej sieci bazowej ResNet-101. Po dodaniu warstw Feature Pyramid Network, połączenie tych dwóch sieci charakteryzuje się dużą efektywnością w ekstrakcji cech.

Fragmenty obliczonych map cech kolejnych warstw piramidy wraz z wyznaczonymi dla różnych skal *anchors* są danymi wejściowymi do sieci Region Proposal Network, która znajdzie regiony o największym prawdopodobieństwie posiadania w swoim obszarze obiektu (Regions-of-Interest, Rols).

Warstwa ROI Pooling nakłada Rols na poziomy map cech piramidy i wycina odpowiadające im obszary. Przy użyciu interpolacji dwuliniowej są one zmniejszane do rozmiaru 7×7 .

Te powierzchniowo niewielkie mapy cech są wczytywane do modułu klasyfikatora, gdzie przy pomocy warstw konwolucyjnych i w pełni połączonych przewidywane są współrzędne ramek oraz klasy i wskaźniki pewności (*confidence scores*) obiektów.



Rys. 26: Topologia sieci Faster R-CNN, źródło: [18]

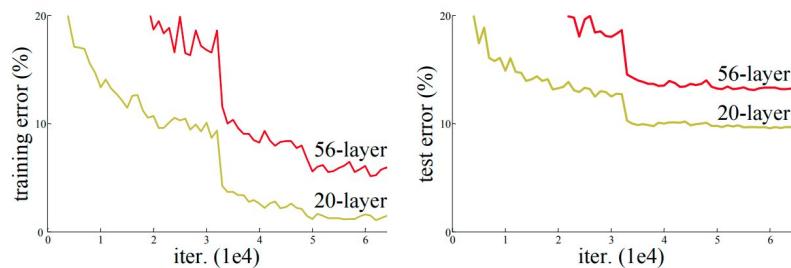
Kolejność operacji przedstawiona jest na Rys. 26. Program napisano w *Python3.6*, przy użyciu bibliotek *Tensorflow-gpu 1.14*, *Keras 2.2.4*. Najważniejsze fragmenty kodu dostępne są w **Dodatku A**, a całość wraz z opisem instalacji i użytkowania na stronie github.com/mszymanowicz/Mask_RCNN.

3.2. Sieć bazowa - ResNet-101

Jako sieć bazowa (*backbone*) zastosowana została sieć *ResNet-101* [50]. W porównaniu do sieci z rodziną ResNet (Rys. 27), wcześniej stosowane sieci konwolucyjne, w których połączone były jedynie kolejne warstwy, nie mogły być zbyt głębokie. Występowało zjawisko saturacji - zanikał gradient, gdy w kolejnych iteracjach do funkcji aktywacji przekazywane były skrajne wartości. Objawiało się ono degradacją przewidywań sieci dla zbiorów treningowego oraz testowego (Rys. 28). Takie sieci zazwyczaj zawierały więc jedynie ok. kilkunastu warstw konwolucyjnych, dobrym przykładem tego typu jest sieć VGG-16.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
				3×3 max pool, stride 2		
conv2_x	56×56	$\left[\begin{array}{l} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 2$	$\left[\begin{array}{l} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 3$	$\left[\begin{array}{l} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$	$\left[\begin{array}{l} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$	$\left[\begin{array}{l} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$
conv3_x	28×28	$\left[\begin{array}{l} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 2$	$\left[\begin{array}{l} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 4$	$\left[\begin{array}{l} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$	$\left[\begin{array}{l} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$	$\left[\begin{array}{l} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 8$
conv4_x	14×14	$\left[\begin{array}{l} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 2$	$\left[\begin{array}{l} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 6$	$\left[\begin{array}{l} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 6$	$\left[\begin{array}{l} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 23$	$\left[\begin{array}{l} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 36$
conv5_x	7×7	$\left[\begin{array}{l} 3 \times 3, 512 \\ 3 \times 3, 512 \end{array} \right] \times 2$	$\left[\begin{array}{l} 3 \times 3, 512 \\ 3 \times 3, 512 \end{array} \right] \times 3$	$\left[\begin{array}{l} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$	$\left[\begin{array}{l} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$	$\left[\begin{array}{l} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$
	1×1			average pool, 1000-d fc, softmax		
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Rys. 27: Wykaz odmian ResNet zastosowanych przez twórców do treningu na zbiorze ImageNet, źródło: [50]

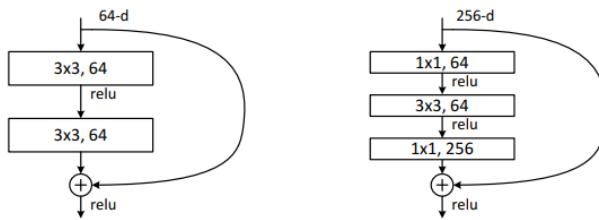


Rys. 28: Zjawisko degradacji na przykładzie sieci CIFAR-10, źródło: [50]

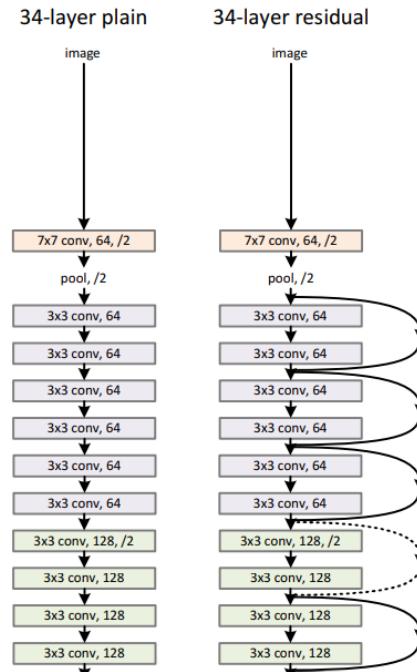
Okazało się, że zbudowanie sieci konwolucyjnej z bloków rezydualnych zapobiega niekorzystnemu zjawisku degradacji. Za pomocą funkcji tożsamościowej łączone są oddalone od siebie warstwy, a samo przekształcenie tożsamościowe nie dodaje złożoności, ani nie wpływa negatywnie na rezultat działania sieci. W praktyce, sieć łatwiej aproksymuje funkcję oryginalną

$H(x) = F(x) + x$ niż uczy się funkcji $H(x)$ bez tej pomocy. Taka topologia (Rys. 30) umożliwia budowę bardzo głębokich sieci o dużej dokładności. W istocie, stworzenie sieci ResNet doprowadziło do przełomu w dziedzinie klasyfikacji obiektów. Sieć ResNet-101 będzie jednak w ramach tego projektu wykorzystywana wyłącznie do ekstrakcji cech.

Twórcy sieci ResNet-101 w celu zredukowania czasu treningu oraz zmniejszenia złożoności sieci zastosowali w tym przypadku *bottleneck residual block* (Rys. 29). Przed konwolucją 3×3 tymczasowo redukowana jest czterokrotnie głębokość warstw. Sieć ResNet-101 zbudowana z takich bloków, mimo posiadania 101 warstw, nadal wymaga mniejszej ilości operacji zmienoprzecinkowych (*FLOPS*) niż konwencjonalna sieć VGG-19. [50]



Rys. 29: Porównanie topologii zwykłego bloku rezydualnego i *bottleneck residual block*, źródło: [50]



Rys. 30: Porównanie topologii zwykłej sieci konwolucyjnej z siecią rezydualną, źródło: [50]

W tym projekcie do sieci ResNet-101 będą wczytywane obrazy RGB z bazy danych WiderFace, każdorazowo ze zmienioną rozdzielczością do 1024×1024 . W zależności od rozdzielczości oryginalnego obrazu, schemat postępowania jest nieco inny:

- gdy jeden lub oba wymiary są większe od 1024px, obraz zostaje przeskalowany tak, aby największy wymiar wynosił teraz 1024px;
- gdy jeden lub oba wymiary są mniejsze od 800px, obraz jest skalowany tak, by większy wymiar osiągnął 1024px, jeśli poniższa nierówność jest spełniona. W przypadku niespełnienia nierówności, mniejszy wymiar osiągnie 800px:

$$\text{round} \left(\frac{\max(h, w) * 800}{\min(h, w)} \right) > 1024$$

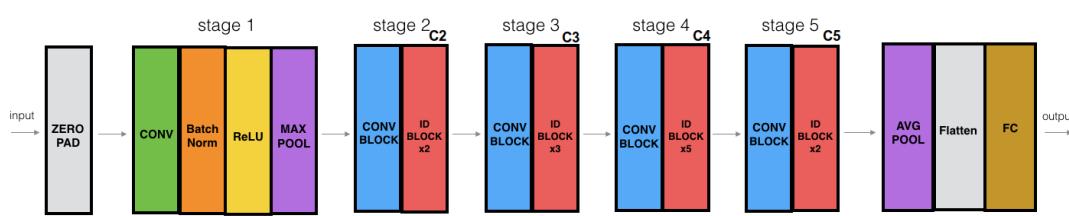
- gdy oba wymiary są większe lub równe 800px, lecz mniejsze lub równe od 1024px, obraz nie jest skalowany;
- najczęstszy przypadek w bazie danych Wider Face, dotyczący ponad połowy obrazów: jeden z wymiarów wynosi 1024px, a drugi wymiar jest równy lub mniejszy - obraz nie wymaga skalowania.

Obrazy są skalowane za pomocą interpolacji dwuliniowej (*bilinear interpolation*), korzystając z funkcji *resize()* pakietu *scikit-image*. Stosunek wymiarów po skalowaniu zostaje zachowany. Jeśli finalnie jeden lub oba wymiary są mniejsze od 1024px, wprowadza się zerowy *padding*.

3.3. Feature Pyramid Network

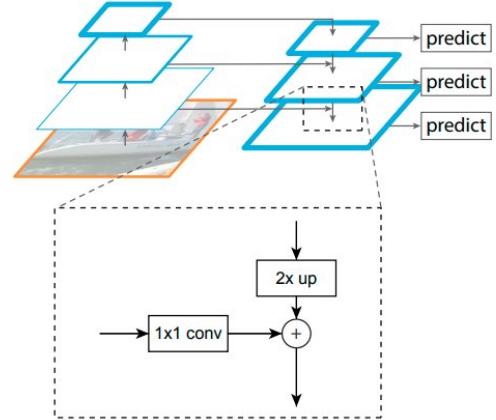
Feature Pyramid Network (FPN) [51] stanowi ulepszenie sieci bazowej ResNet-101. Połączenia w ResNet-101 stanowią przykład ścieżki *bottom-up* - otrzymywane są coraz mniejsze przestrzennie mapy o wysokopoziomowych cechach. FPN dodaje ścieżkę *top-down*, która zaczynając od ostatniej, najmniejszej warstwy ResNetu tworzy kolejne, nowe mapy cech o coraz wyższej rozdzielcości. Aby zapobiec częściowej utracie informacji o lokalizacji cech spowodowanej próbkowaniem *Pooling* i *Upsampling*, między warstwami o tej samej rozdzielcości z obu ścieżek stosuje się połączenia lateralne. W celu zmniejszenia ilości kanałów, w połączeniach lateralnych stosuje się warstwę konwolucji 1×1 , a mapę cech tworzy się przez zsumowanie tensorów, jak pokazano na schemacie bloku budującego FPN (Rys. 31). Są to bardzo proste operacje bazujące głównie na mapach cech obliczonych przez ResNet, więc nie wymagają dużego zużycia mocy obliczeniowej. Architektura sieci FPN zapewnia cechom z każdej warstwy dostęp zarówno do map wysokopoziomowych, jak też niskopoziomowych. [51]

Aby pokazać miejsca połączeń między ścieżkami *bottom-up* i *top-down*, warto podzielić sieć ResNet-101 na kilka etapów. Z map cech otrzymanych po każdym z etapów, za wyjątkiem pierwszego (C_2, C_3, C_4, C_5) (Rys. 32), wprowadzone są połączenia lateralne do odpowiadających im warstw ścieżki *top-down*. Warstwy te oznacza się kolejno P_5, P_4, P_3, P_2 (Rys. 33). Przykładowo, mapa cech z najwyższej warstwy P_5 ma wymiary $32 \times 32 \times 2048$.

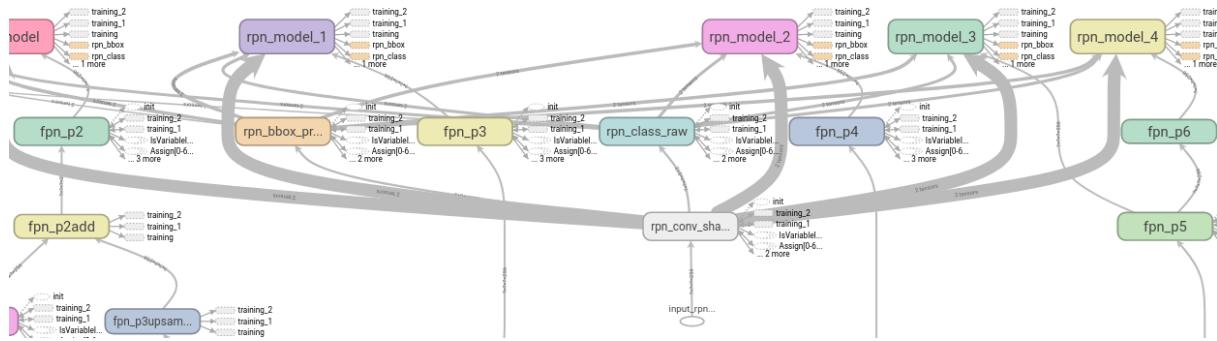


Rys. 32: Podział sieci ResNet-101 na 5 etapów, źródło: [52]

Dodatkowo, wprowadza się warstwę *MaxPooling P6*, używaną potem do generacji *anchors* w piątej skali w omówionej w kolejnym rozdziale sieci RPN. Bardzo ważną różnicą pomiędzy tą wersją sieci, a klasyczną Faster R-CNN jest obecność pięciu bloków RPN, które przyjmują na wejściu macierze cech o różnej rozdzielcości (Rys. 33).



Rys. 31: Blok budujący sieci FPN: sumacja tensoru powstałego po operacji *Upsampling* z czynnikiem 2 oraz tensoru otrzymanego na wyjściu z połączenia lateralnego, źródło: [51]



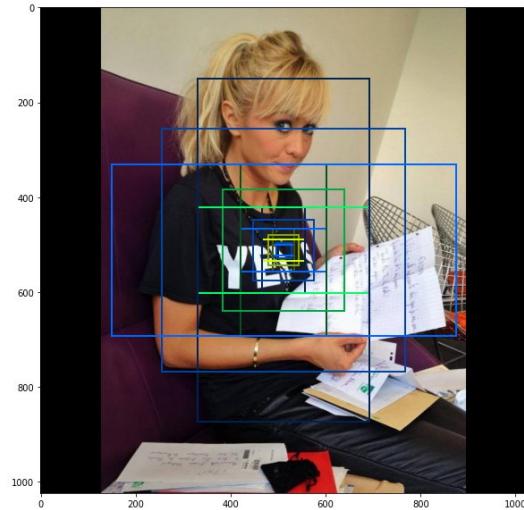
Rys. 33: Połączenia warstw FPN z modelem RPN na grafie Tensorboard

3.4. Anchors

Dla każdego obrazu, niezależnie od rezultatów treningu, generowanych jest w sumie ponad 200 tys. predefiniowanych ramek (*anchors*), w których mogą zawierać się obiekty. Dla *anchors* przewidziano 5 skal $[32, 64, 128, 256, 512]$ odpowiadających rozdzielczości poszczególnych warstw FPN: $\{P_6, P_5, P_4, P_3, P_2\}$. Skale te w rzeczywistości definiują jedynie pole *anchors*, np. skala 32 oznacza, że pole powierzchni tej ramki wynosi 32^2 . Dozwolone stosunki między bokami podaje parametr *aspect_ratios* = $[0.5, 1, 2]$. *Anchors* mają więc 15 dozwolonych rozmiarów ogólnie, a $k = 3$ dozwolonych rozmiarów dla każdej ze skal (Rys. 34).

W zależności od wielkości wspólnego pola powierzchni *anchor* z ramką ze zbioru treningowego, dzieli się *anchors* na pozytywne, neutralne i negatywne. *Anchor* uznawany jest za pozytywny, gdy jego Intersection-over-Union (IoU) przekracza 0.7 lub gdy dla danej ramki ten *anchor* posiada największe IoU. Negatywne *anchors* charakteryzują się IoU mniejszym od 0.3 dla wszystkich ramek. Pozostałe *anchors* oznaczane są jako neutralne i nie biorą udziału w treningu sieci. Istotnych dla treningu ramek jest zazwyczaj od kilkuset do nieco ponad tysiąca.

Level 0. Anchors: 196608 Feature map Shape: [256 256]
 Level 1. Anchors: 49152 Feature map Shape: [128 128]
 Level 2. Anchors: 12288 Feature map Shape: [64 64]
 Level 3. Anchors: 3072 Feature map Shape: [32 32]
 Level 4. Anchors: 768 Feature map Shape: [16 16]



Rys. 34: Anchors w różnych skalach nanieśione na obraz. Wyżej ilość anchors na każdym z poziomów

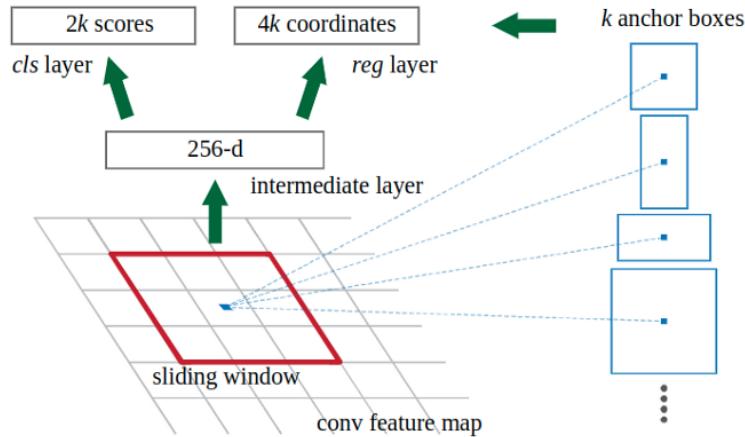
3.5. Region Proposal Network

Region Proposal Network (RPN) to niewielka, w pełni konwolucyjna sieć, której zadaniem jest wstępne wyznaczenie przypuszczalnych położen obiektów, w tym wypadku twarzy. Po każdej iteracji treningu, ustalone jest prawdopodobieństwo zawierania obiektów przez ramki, a parametry funkcji strat ulegają zmianie. Przy czym, te wstępne propozycje położenia obiektów modyfikowane są w procesie nauki całej sieci. Po wyuczeniu całej sieci, dobre wstępne propozycje ramek po przejściu przez moduł klasyfikatora, powinny pozwolić na dokładne dopasowanie ramek do obiektów w różnych położeniach i o różnej wielkości. Stąd też, propozycje

ramek obejmują cały obraz w różnych punktach i posiadają wiele skali.

RPN przesuwa się wzdłuż map cech ze ścieżki top-down obliczonych przez FPN, przyjmując na wejściu ich fragmenty o wymiarach 3×3 . Proces ten jest dobrze zoptymalizowany i wydajny. Wspomniana wyżej warstwa konwolucyjna o filtrze 3×3 stanowi wspólną podstawę dla dwóch warstw dokonujących konwolucji 1×1 . RPN dzieli się na pięć bloków odpowiadających innym rozdzielnictwom macierzy cech z FPN. Każdy blok posiada dwie gałęzie.

Pierwsza gałąź (Rys. 35) służy dokonaniu klasyfikacji pomiędzy tłem, a obiektemi. Po warstwie konwolucji 1×1 o ilości kanałów $2k$ i liniowej funkcji aktywacji, zastosowano warstwę softmax, na wyjściu której otrzymywane są przewidywane przez sieć RPN prawdopodobieństwa zawierania przez dany *anchor* obiektu lub jedynie tła.



Rys. 35: Początek i rozgałęzienie sieci RPN, źródło: [18]

Druga gałąź zajmuje się udoskonaleniem położenia pozytywnych *anchors*, które początkowo mogą nie być perfekcyjnie spozycjonowane. Zawiera jedynie warstwę konwolucji 1×1 o ilości kanałów $4k$ i liniowej funkcji aktywacji. Przewiduje ona delta, czyli zmiany współrzędnych potrzebne do najlepszego według RPN dopasowania *anchors*. We wzorach (9) ukazano zależności elementów wektora delt od współrzędnych ramek:

$$\vec{t} = [t_x \quad t_y \quad t_w \quad t_h]$$

$$t_x = \frac{x - x_a}{w_a} \quad t_x^* = \frac{x^* - x_a}{w_a}$$

$$t_y = \frac{y - y_a}{h_a} \quad t_y^* = \frac{y^* - y_a}{h_a}$$

$$t_w = \ln\left(\frac{w}{w_a}\right) \quad t_w^* = \ln\left(\frac{w^*}{w_a}\right)$$

$$t_h = \ln\left(\frac{h}{h_a}\right) \quad t_h^* = \ln\left(\frac{h^*}{h_a}\right)$$
(9)

, gdzie literami x, y, w, h oznaczono współrzędne (x, y) centrum ramki, jej szerokość oraz wysokość. Przykładowo: w - szerokość anotowanej ramki, w_a - szerokość prostokąta *anchor* powiązanego z anotowaną ramką. Symbol $*$ oznacza wartość przewidywaną przez RPN, brak tego symbolu to wartość prawdziwa.

Następnie, obliczany jest iloraz wektora delt i arbitralnie ustalonego wektora odchyлеń standardowych. W naszym przypadku wektor odchyłeń standardowych wynosi $[0.1, 0.1, 0.2, 0.2]^T$ - wartości te sprawdziły się podczas treningu na bazie danych COCO. Regresor działa bowiem najlepiej, jeśli wyjścia z niego (elementy znormalizowanych delt) posiadają średnią ≈ 0.0 oraz odchylenie standardowe ≈ 1.0 . Średniej nie trzeba korygować, gdyż *anchors* pokrywają obraz równomiernie - można więc z dużą pewnością założyć, że zmiany rozmiaru oraz udoskonalenia położenia negatywnych i pozytywnych *anchors* także rozkładają się równomiernie [53].

Wyznaczone w gałęziach bloków predykcje grupowane są w trzy duże zbiorcze tensory nieposiadające informacji o blokach: predykcji znormalizowanych delt, predykcji klas, predykcji logitów. Wartości zgromadzone w tych tensorach posłużą do obliczenia dwóch lokalnych funkcji strat dla RPN.

Tensory te służą także jako wejście do warstwy propozycji. Najpierw wybieranych jest co najwyżej 6000 *anchors* o najwyższym prawdopodobieństwie znalezienia obiektu, aby zredukować koszt obliczeniowy kolejnych procedur. Następnie, aplikowane są delta (pomnożone z powrotem przez wektor odchyłeń standardowych) oraz stosowany jest algorytm *Non-Maximum Suppression (NMS)* w celu wybrania najkorzystniejszych pozytywnych, niezachodzących na siebie *anchors*. W dalszej części pracy, nazywane będą one Regions-of-Interest (Rols). Liczba Rols selekcjonowanych przez NMS podczas treningu ograniczona jest do 2000, a podczas inferencji do 1000. Rol uznaje się za pozytywny, gdy jego IoU z oryginalną ramką (nie mylić z IoU dla *anchors* opisanym w poprzednim rozdziale) wynosi co najmniej 0.5, w przeciwnym wypadku oznacza się go jako negatywny.

W obrębie grafu detekcji celów, Rols są próbkiowane tak, aby w obrazie pozytywne Rols stanowiły 33% wszystkich Rols. Liczba Rols niemal zawsze jest równa maksymalnej liczbie Rols przekazywanych do modułu klasyfikatora (domyślnie nie może przekroczyć 200). W razie potrzeby, negatywne Rols są generowane. Regionom przyporządkowuje się konkretne ramki ze zbioru treningowego, obliczane są też ich prawdziwe delta. Gdyby nie próbować i pozostać w obrazie ~99% negatywnych Rols, to sieć nie nauczyłaby się detekcji przy tak ogromnej dysproporcji.

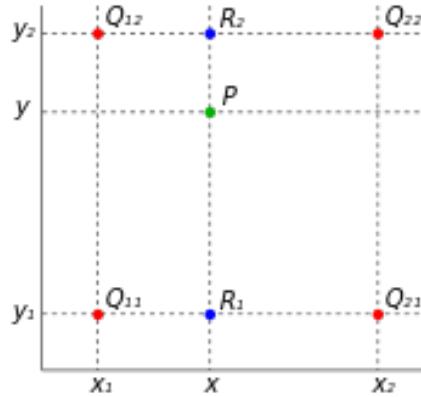
3.6. Rol Pooling

Rol pooling odnosi się do wycięcia fragmentu mapy cech ($P2, P3, P4$ lub $P5$) oraz zmniejszenia go do rozmiaru 7×7 . Takie działanie jest niezbędne, gdyż odpowiedzialne za klasyfikację warstwy w pełni połączone muszą mieć wejścia o stałym rozmiarze. Odbywa się ono w warstwie *PyramidROIAlign*. Jako wejście do niej przyjmowane są wyznaczone wcześniej Rols oraz mapy cech z FPN.

Najpierw, Rols są przyporządkowywane do jednego z dostępnych poziomów mapy cech na podstawie wzoru (9). Parametry h, w to znormalizowane wymiary Rol (np. jeśli wynoszą przed znormalizowaniem wynoszą one 224×224 , po normalizacji będzie to $224/1024 \times 224/1024$.

$$roi_level = \min\left(5, \max\left(2, 4 + \text{round}\left(\log_2\left(\frac{\sqrt{hw} * 1024}{224}\right)\right)\right)\right) \quad (9)$$

Można wyobrazić sobie mapę cech jako siatkę punktów na płaszczyźnie, gdzie do każdego punktu przyporządkowana jest pewna wartość. Fragment mapy cech zostaje wycięty przez ROI i następnie podzielony (zazwyczaj niekwadratową) siatką na 7×7 identycznych oczek. W centrum każdego z oczek znajduje się punkt, którego wartość zostaje obliczona przy użyciu interpolacji dwuliniowej. Ważna różnica: w oryginalnym projekcie *Mask R-CNN* użyto czterech punktów w każdym oczku. Tu jednak wykorzystywana jest funkcja `tf.image.crop_and_resize()`, która upraszcza to podejście, stosując pojedynczy punkt w oczku, co jest niemal tak samo efektywne [19]. W przypadku detekcji, która w przeciwieństwie do segmentacji nie wymaga pikselowej precyzji, mała strata na dokładności wynikająca z uproszczonego podejścia nie ma znaczenia.



Rys. 36: Czerwone kropki odpowiadają najbliższym punktom mapy cech, P to punkt w centrum jednego z oczek, dla którego szukana jest wartość, źródło: [54]

Znane są wartości dla czterech punktów mapy cech najmniej oddalonych od punktu z centrum oczka $f(x_1, y_1), f(x_2, y_1), f(x_1, y_2), f(x_2, y_2)$. Wzór na interpolowaną wartość dla punktu z centrum oczka [55] (Rys. 36):

$$f(x, y) \approx \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(x_1, y_1) & f(x_1, y_2) \\ f(x_2, y_1) & f(x_2, y_2) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix} \quad (10)$$

Punkty z centrum oczek wraz z obliczonymi w równaniu (10) wartościami stanowią wyjście o rozmiarze 7×7 z warstwy ROI Pooling.

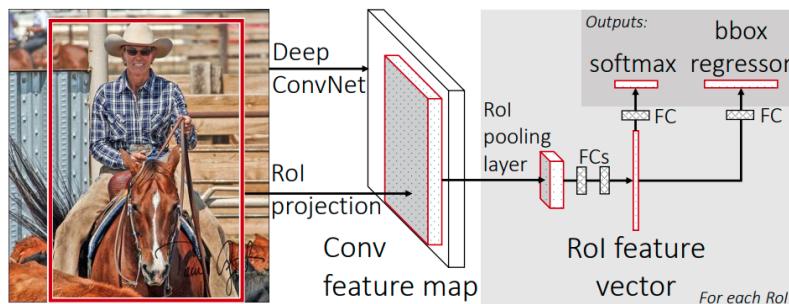
W porównaniu do rozwiązania zaproponowanego w klasycznym Faster R-CNN, polegających na kwantyzacji i przeprowadzaniu *Max Pooling* na mapach cech, jak też metody *ROI Warp*, znana z Mask R-CNN procedura *ROIAlign* zapewnia największą dokładność [19] (Rys. 37).

	<th>bilinear?</th> <th>agg.</th> <th>AP</th> <th>AP₅₀</th> <th>AP₇₅</th>	bilinear?	agg.	AP	AP ₅₀	AP ₇₅
RoIPool			max	26.9	48.8	26.4
RoIWarp		✓	max	27.2	49.2	27.1
		✓	ave	27.1	48.9	27.1
RoIAvg	✓	✓	max	30.2	51.0	31.8
	✓	✓	ave	30.3	51.2	31.5

Rys. 37: Porównanie metod RoI Pooling (ResNet-50-C4), źródło: [19]

3.7. Klasyfikator

Zastosowany tu moduł klasyfikatora ma taką samą budowę jak klasyfikator wcześniejszej wersji Fast R-CNN (Rys. 38). Wyjścia z warstwy *PyramidROIAlign* o stałej rozdzielczości 7×7 wczytywane są najpierw do warstwy konwolucyjnej o filtrze 7×7 , liczbie kanałów równej 1024, która pełni rolę warstwy wypłaszczającej. Po przejściu przez warstwę aktywacji ReLU, kolejną warstwę konwolucyjną o filtrze 1×1 i jeszcze jedną funkcję aktywacji ReLU, sieć rozgałęzia się.



Rys. 38: Topologia klasyfikatora Fast R-CNN, źródło: [17]

Pierwsza gałąź odpowiada za klasyfikację obiektów i ocenę prawdopodobieństwa ich przynależności do poszczególnych klas (*confidence score*). Składa się z warstwy w pełni połączonej i warstwy Softmax. Liczba wyjść jest równa ilości klas (w tym tła).

Druga gałąź zajmuje się poprawą współrzędnych wstępnych propozycji położenia ramek. Zawiera jedynie warstwę w pełni połączoną o liniowej funkcji aktywacji. Liczba wyjść to czterokrotność ilości klas.

Trzy otrzymane z gałęzi tensory: predykcji znormalizowanych delt między przewidywaną i anotowaną ramką, predykcji logitów, predykcji klas, posłużą do obliczenia dwóch lokalnych funkcji strat modułu klasyfikatora.

3.8. Funkcja strat

Wartość całkowitej funkcji strat to:

$$L_{total}(y, y^*) = L_{multitask}(y, y^*) + L_{L2reg} \quad (11)$$

, gdzie y - prawdziwe wyniki dla rozpatrywanej funkcji strat, y^* - predykcja.

Aby zmniejszyć ryzyko przeuczenia, stosuje się regularyzację L2. Szczególnie mocno penalizuje ona wagę o dużej wartości, przyczyniając się do mniejszej złożoności modelu i lepszej generalizacji. W tym przypadku, zastosowany parametr $\lambda = 0.0001/k_i$ opisujący jak bardzo penalizowane są wagi jest jednak stosunkowo mały i zmienny w zależności od ilości wag w warstwie. W ten sposób, zniwelowany zostanie wpływ na funkcję strat L_{L2reg} kilku dość dużych wag w warstwach mających kilkadziesiąt tysięcy parametrów, przy zachowaniu wpływu tych samych wag w warstwach mających parametrów kilkadziesiąt. Oznaczenia: i - numer warstwy, l - ilość trenowań warstw (wg pakietu Keras), k_i - ilość trenowań wag w danej warstwie, w_{ij} - pojedyncza waga w danej warstwie. Wartości funkcji strat L_{L2reg} są prawie zawsze rzędu $< 10^{-8}, 10^{-4} >$ (nikle w porównaniu z lokalnymi funkcjami strat), jednak w rzadkim przypadku nagłego pojawienia się bardzo dużych wag, wartości L_{L2reg} stają się znaczące, a penalizacja następuje natychmiast.

$$L_{L2reg} = \sum_{i=1}^l \left(\frac{0.0001}{k_i} \sum_{j=1}^{k_i} w_{ij}^2 \right) \quad (12)$$

Wielozadaniowa funkcja strat (*multi-task loss function*) to ogólnie suma iloczynów arbitralnie dobranych wag i lokalnych funkcji strat. Tutaj wszystkie wagę ustalone $W_j = 1$, więc jest to po prostu suma wszystkich czterech lokalnych funkcji strat.

$$L_{multitask} = L_{cls1} + L_{bbox1} + L_{cls2} + L_{bbox2} \quad (13)$$

Jeśli wczytywany jest w serii więcej niż jeden obraz (BATCH_SIZE > 1), ostateczna lokalna funkcja straty stanowi średnią arytmetyczną lokalnych funkcji strat liczonych dla pojedynczych obrazów (w obliczeniach oznaczono je falką, np. \tilde{L}_{bbox1}). Lokalne funkcje strat liczone są tak samo jak w oryginalnym projekcie Faster R-CNN, różnią się natomiast współczynnikami, które w [18] oznaczono jako λ . W tej implementacji sieci Faster R-CNN minimalizowane będą cztery lokalne funkcje strat, takie jak pokazano na Rys. 26. Są to odpowiednio:

- W pierwszej gałęzi RPN, funkcja straty kategorycznej entropii krzyżowej. Jest liczona zbiorczo dla wszystkich pięciu bloków RPN. Zarówno pozytywne *anchors* (z daną anotowaną ramką $IoU \geq 0.7$ lub największe), jak też negatywne ($IoU < 0.3$) mają wpływ na funkcję straty. N_{cls1} - ilość negatywnych i pozytywnych *anchors*, niemal zawsze osiąga maksymalną domyślną wartość 256. s_i - wyznaczone przez sieć prawdopodobieństwo znalezienia się Rol w klasie i (obiekt albo tło, stąd $C = 2$). s_i jest obliczane za pomocą funkcji *Softmax*. t_i - wartość logiczna: $t_i = 1$ dla poprawnej klasyfikacji.

$$\tilde{L}_{cls1} = \frac{1}{2 * N_{cls1}} \sum_{n=1}^{N_{cls1}} \sum_{i=1}^{C=2} (-\ln(s_i) * t_i) \quad (14)$$

- W drugiej gałęzi RPN, funkcja straty *smooth-L1*. Jest liczona zbiorczo dla wszystkich pięciu bloków RPN. Zachowuje się ona jak *L2-loss*, gdy wartość bezwzględna różnicy między współrzedną delty przewidywanej, a delty prawdziwej jest mniejsza od 1; w przeciwnym wypadku funkcja przyjmuje postać *L1-loss*. Tylko pozytywne *anchors* mają wpływ na tą funkcję straty. N_{bbox1} - ilość pozytywnych *anchors*, $y_{true}^{[n,i]}$ - element o indeksie i n-tego wektora prawdziwych delt pomiędzy anotowaną ramką, a przyporządkowanym jej pozytywnym *anchor* (o długości 4), y_{pred} - macierz przewidywanych delt między nimi.

$$L_{smoothL1}(x) = \begin{cases} 0.5x^2, & |x| < 1 \\ |x| - 0.5, & |x| \geq 1 \end{cases} \quad (15)$$

$$\tilde{L}_{bbox1} = \frac{1}{4 * N_{bbox1}} \sum_{n=1}^{N_{bbox1}} \sum_{i=1}^4 L_{smoothL1}(y_{true}^{[n,i]} - y_{pred}^{[n,i]}) \quad (16)$$

- Na pierwszym wyjściu z modułu klasyfikatora, również funkcja straty kategorycznej entropii krzyżowej. Zmieniają się jedynie klasy, gdyż sieć przyporządkowuje obiekt do klasy 'face' albo do tła (czyli liczba klas nadal $C = 2$, różnica ta miałaby większe znaczenie, gdyby zadaniem sieci było rozpoznanie obiektów z różnych kategorii). Zarówno pozytywne ($IoU \geq 0.5$ z anotowaną ramką), jak też negatywne Rols ($IoU < 0.5$) mają wpływ na tą funkcję straty. Brane są pod uwagę tylko predykcje dla klas występujących w serii obrazów. N_{cls2} - liczba wyselekcjonowanych wcześniej Rols, najczęściej domyślna maksymalna wartość 200. s_i - wyznaczone przez sieć prawdopodobieństwo znalezienia się przewidywanej ramki w klasie i . s_i jest obliczane za pomocą funkcji *Softmax*. t_i - wartość logiczna: $t_i = 1$ dla poprawnej klasyfikacji.

$$\tilde{L}_{cls2} = \frac{1}{2 * N_{cls2}} \sum_{n=1}^{N_{cls2}} \sum_{i=1}^{C=2} (-\ln(s_i) * t_i) \quad (17)$$

- Na drugim wyjściu z modułu klasyfikatora, również funkcja straty *smooth-L1*. Tylko pozytywne Rols mają wpływ na tą funkcję straty. Brane są pod uwagę tylko predykcje dla klas występujących w serii obrazów. N_{bbox2} - ilość wyselekcjonowanych wcześniej pozytywnych Rols, $y_{true}^{[n,i]}$ - element o indeksie i n-tego wektora prawdziwych delt pomiędzy anotowaną ramką, a przyporządkowanym jej pozytywnym Rol (o długości 4), y_{pred} - macierz przewidywanych delt między nimi.

$$\tilde{L}_{bbox2} = \frac{1}{4 * N_{bbox2}} \sum_{n=1}^{N_{bbox2}} \sum_{i=1}^4 L_{smoothL1}(y_{true}^{[n,i]} - y_{pred}^{[n,i]}) \quad (18)$$

Funkcje strat optymalizowane są metodą mini-batch SGD ze stałym parametrem uczenia $lr = 0.001$, $momentum = 0.9$ i normą skalowania gradientu $clipnorm = 1$.

Zaproponowany przez autorów Mask R-CNN[19] $lr = 0.02$ był za duży i często powodował eksplozje gradientu. Przed tym zjawiskiem chroni też $clipnorm = 1$, który po przekroczeniu normy L2 wektora gradientu, skaluje wartości tak, by ta norma wynosiła 1.

3.9. Wczytane wagi i trening sieci

Do nowego modelu wczytywane są wagi pretrenowane na bazie danych MS COCO. Epoka według biblioteki Keras to 1000 iteracji po seriach obrazów (*batches*), po upływie której można zapisać wagi. *Batch* w naszym przypadku zawiera 2 obrazy, trening odbywa się na pojedynczym GPU *NVIDIA Tesla T4* w trzech etapach:

- 1) Epoki 1-40: w treningu uczestniczą wszystkie możliwe warstwy, za wyjątkiem tych należących do sieci bazowej ResNet-101.
- 2) Epoki 41-120: w treningu nie uczestniczą jedynie pierwsze trzy bloki sieci ResNet-101
- 3) Epoki 121-160: w treningu uczestniczą wszystkie możliwe warstwy.

Podzespoły komputera: *NVIDIA Tesla T4*, 4vCPU *Intel Xeon Scalable Processor (Skylake)*, 15GB pamięci RAM, dysk SSD.

Trening na bazie danych Wider Face przy podanych podzespołach zajmuje ok. 48h.

4. Część eksperimentalna

4.1. Modyfikacje bazy danych Wider Face

Oryginalna baza Wider Face [13] zawiera 32203 zdjęcia wraz z 393703 oznaczonymi twarzami o dużej różnorodności w skali, pozycji, okluzji. Zdjęcia te podzielono w 61 grup w zależności od wydarzenia, na jakim zostały one wykonane. Z każdej z tych grup, przyporządkowano odpowiednio ~40%/10%/50% zdjęć do zbioru treningowego/walidacyjnego/testowego. Anotacje dla zbioru testowego nie są publicznie dostępne, więc parametry sieci będą ewaluowane jedynie na podstawie zbioru walidacyjnego.

Dla poprawienia dokładności sieci i uniknięcia ewentualnych problemów z zanikającym, bądź eksplodującym gradientem, autor tej pracy zdecydował się usunąć z bazy danych wszystkie obrazy niezawierające żadnych oznaczeń twarzy. Mogłyby one spowodować katastrofalne błędy, gdyż w bazie danych były oznaczone jako pojedyncza ramka o współrzędnych $[0, 0, 0, 0]$. Nazwy tych plików podano w **Dodatku B**.

Dodatkowo, autor tej pracy przy pomocy własnego programu znajdującego wadliwe anotacje, usunął je. Nazwy plików i wykaz tych anotacji zamieszczono w **Dodatku B**.

Tak przygotowana baza danych zawiera ostatecznie 12876 obrazów w zbiorze treningowym i 3222 obrazy w zbiorze walidacyjnym. W porównaniu z innymi bazami danych twarzy, ta jest wymagająca, gdyż może zawierać do kilkuset twarzy na jednym obrazie, często w małej skali i dość rozmazanych, częściowo zasłoniętych. Przykładowe obrazy poniżej (Rys. 39, 40, 41):



Rys. 39: 92 twarze małych rozmiarów
18_Concerts_Concerts_18_203.jpg



Rys. 40: 15 twarzy średnich rozmiarów
26_Soldier_Drilling_Soldiers_Drilling_26_20.jpg



Rys. 41: 2 twarze dużych rozmiarów
38_Tennis_Tennis_38_1029.jpg

4.2. Miary oceny uczenia sieci

Oznaczenia: FP - false positive, FN - false negative, TP - true positive, TN - true negative.

- Precyza - miara wskazująca jaka część poprawnie sklasyfikowanych instancji jest wśród wszystkich predykcji instancji. W przypadku, gdy dane są punktowe i możliwe do klasyfikacji należy/nie należy do instancji, jej definicja jest następująca:

$$Precision = \frac{TP}{TP + FP}$$

- Recall - miara wskazująca, jaką część poprawnie sklasyfikowanych instancji jest wśród wszystkich rzeczywistych instancji. W przypadku, gdy dane są punktowe i możliwe do klasyfikacji należy/nie należy do instancji, jego definicja jest następująca:

$$Recall = \frac{TP}{TP + FN}$$

Podczas detekcji obiektów, do instancji przyporządkowywany jest zestaw współrzędnych opisujący obszar. Predykcja polega na ich wyznaczeniu wraz z prawdopodobieństwem przynależności do klas.

W rozpatrywanej sieci Faster R-CNN obliczenie tych miar jest bardziej skomplikowane:

- 1) Sortuje się przewidywane ramki klasy 'face' według ich prawdopodobieństwa przynależności do tej klasy.
- 2) Tworzy się rzadką macierz $overlaps$, w której znajdują się obliczone IoUs wszystkich kombinacji par przewidywanych ramek z ramkami anotowanymi. Następnie, dla kolejnych przewidywanych ramek dobiera się nieprzyporządkowaną wcześniej anotowaną ramkę o największym IoU, pod warunkiem, że IoU jest wyższe od ustalonego progu. Podczas wyznaczania mAP/mAR często wykorzystuje się różne progi IoU.
- 3) Po tej operacji, powstają dwa wektory: wektor przyporządkowań indeksów przewidywanych ramek dla indeksów anotowanych ramek gt_match oraz wektor przyporządkowań indeksów anotowanych ramek dla indeksów przewidywanych ramek $pred_match$. Wektory te mogą mieć inną długość (odpowiednio m i n). W przypadku braku przyporządkowania którejś z ramek, przydziela się jej pozycji w wektorze wartość -1 . Przykładowe wektory:

$$gt_match = [8, 6, -1, 12, 2, 4, -1, 1, 3, 0, -1, -1, -1, -1, -1, -1]$$

$$pred_match = [9, 7, 4, 8, 5, -1, 1, -1, 0, -1, -1, -1, 3, -1]$$

(Do 0-ej anotowanej ramki przyporządkowano 8-mą przewidywaną ramkę. 2-ga anotowana ramka nie posiada odpowiadającej jej predykcji. 5-ta przewidywana ramka nie została przyporządkowana do żadnej anotowanej ramki.)

- 4) Tworzy się nowy wektor o długości wektora $pred_match$, zawierający wartości 1 na pozycjach, w których elementy $pred_match[i] > -1$. W przeciwnym wypadku, przypisuje się

wartości 0.

$$\vec{u} = [1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0]$$

Ogólnie:

$$\vec{u} = (u_0, u_1, \dots, u_{n-1}) \quad (19)$$

Oblicza się sumę kumulacyjną tego wektora. Wartość danego elementu to liczba znalezionych dotychczas TP .

$$v_{TP}^{[j]} = \sum_{j=0}^{n-1} u^{[j]} \quad (20)$$

$$v_{TP}^{\vec{u}} = [1 \ 2 \ 3 \ 4 \ 5 \ 5 \ 6 \ 6 \ 7 \ 7 \ 7 \ 7 \ 8 \ 8]$$

- 5) Oblicza się wektor precyzji \vec{p} i wektor recalls \vec{r} . Precyzję można zdefiniować jako stosunek ilości prawdziwie pozytywnych (przyporządkowanych, TP) ramek powyżej pewnego prawdopodobieństwa do ilości wszystkich przewidywanych powyżej tego prawdopodobieństwa ramek ($TP + FP$). Recall jest stosunkiem ilości prawdziwie pozytywnych (przyporządkowanych, TP) ramek powyżej pewnego prawdopodobieństwa do ilości m wszystkich anotowanych ramek ($TP + FN$).

$$\forall j \in \{0, 1, \dots, n-1\} \quad v_{TP+FP}^{[j]} = j + 1$$

$$\forall j \in \{0, 1, \dots, n-1\} \quad p^{[j]} = \frac{v_{TP}^{[j]}}{v_{TP+FP}^{[j]}}$$

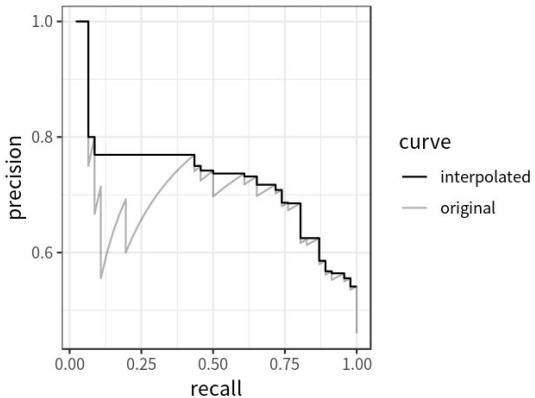
$$\forall j \in \{0, 1, \dots, n-1\} \quad r^{[j]} = \frac{v_{TP}^{[j]}}{m}$$

- 6) Dokonuje się interpolacji precyzji przez wybór maksymalnej precyzji ze wszystkich odpowiadających wyższym poziomom r Recall. Takie działanie zapewnia, iż krzywa Precision-Recall jest nierosnąca [56].

$$p_{interp}(r) = \max_{\tilde{r}: \tilde{r} > r} p(\tilde{r}) \quad (21)$$

- 7) Z definicji, mAP (*mean Average Precision*) dla pojedynczego obrazu to pole powierzchni pod krzywą Precision-Recall. Tu używane są interpolowane wartości precyzji. Pole pod krzywą P-R najlepiej obliczyć, sumując pola kolejnych prostokątów:

$$mAP = \int_0^1 p_{interp}(r) dr = \sum_{j=0}^{n-2} (r_{j+1} - r_j) * p_{interp}(r_{j+1}) \quad (22)$$



Rys. 42: Przykład krzywej P-R po interpolacji precyzji, źródło: [57]

- 8) mAR (*mean Average Recall*) dla pojedynczego obrazu zdefiniowane jest jako stosunek ilości wszystkich prawdziwie pozytywnych przyporządkowanych ramek powyżej pewnego prawdopodobieństwa do ilości wszystkich anotowanych ramek. Jest przedostatnim elementem wektora *recalls* z implementacji równania (21).

Będą używane zasady ewaluacji znane z baz danych i zawodów COCO. Dla pojedynczego obrazu podaje się miary Precision oraz Recall:

- dla różnych progów IoU. Najczęściej spotykane progi to $IoU = 0.5$, $IoU = 0.75$, $IoU = 0.5 : 0.05 : 0.95$ (tu liczymy średnią arytmetyczną miary z 10 progów),
- dla różnych rozmiarów anotowanych ramek. *area = small* oznacza, że brane są pod uwagę tylko anotowane ramki, których pole powierzchni $S < 32^2$,
 $area = medium : 32^2 < S < 96^2$,
 $area = large : S > 96^2$,
- dla różnych maksymalnych ilości obiektów, możliwe wartości dla parametru *maxDets*: $maxDets = 100$, $maxDets = 10$, $maxDets = 1$

Wszystkie te miary są następnie stosowane dla całego zbioru walidacyjnego i to właśnie one będą kluczowe dla oceny skuteczności metody, np:

$$mAP_{all} = \frac{1}{N} \sum_{j=1}^N mAP_j \quad (23)$$

4.3. Parametry sieci

Ustawienia ogólne:

- Wejściowa rozdzielcość obrazów: 1024×1024
- BATCH_SIZE = 2 (IMAGES_PER_GPU = 2, GPU_COUNT=1)
- STEPS_PER_EPOCH = 1000
- VALIDATION_STEPS = 50
- NUM_CLASSES = 1 + 1
- MEAN_PIXEL = `np.array([123.7, 116.8, 103.9])`
- IMAGE_CHANNEL_COUNT = 3
- LEARNING_RATE = 0.001

```

- LEARNING_MOMENTUM = 0.9
- WEIGHT_DECAY = 0.0001
- GRADIENT_CLIP_NORM = 5.0
- MAX_GT_INSTANCES = 100
- LOSS_WEIGHTS = {
    "rpn_class_loss": 1.,
    "rpn_bbox_loss": 1.,
    "mrcnn_class_loss": 1.,
    "mrcnn_bbox_loss": 1.,
    "mrcnn_mask_loss": 1.
}

```

Ustawienia sieci bazowej oraz FPN:

```

- BACKBONE = "resnet101"
- BACKBONE_STRIDES = [4, 8, 16, 32, 64]
- TOP_DOWN_PYRAMID_SIZE = 256

```

Ustawienia RPN:

```

- TRAIN_ROIS_PER_IMAGE = 200
- ROI_POSITIVE_RATIO = 0.33
- RPN_BBOX_STD_DEV = np.array([0.1, 0.1, 0.2, 0.2])
- RPN_ANCHOR_SCALES = (32, 64, 128, 256, 512)
- RPN_ANCHOR RATIOS = [0.5, 1, 2]
- RPN_ANCHOR_STRIDE = 1
- RPN_NMS_THRESHOLD = 0.7
- RPN_TRAIN_ANCHORS_PER_IMAGE = 256
- PRE_NMS_LIMIT = 6000
- POST_NMS_ROIS_TRAINING = 2000
- POST_NMS_ROIS_INFERENCE = 1000

```

Ustawienia warstwy RoI Pooling:

```
- POOL_SIZE = 7
```

Ustawienia klasyfikatora Faster R-CNN:

```

- FPN_CLASSIF_FC_LAYERS_SIZE = 1024
- DETECTION_MAX_INSTANCES = 100
- DETECTION_MIN_CONFIDENCE = 0.7
- DETECTION_NMS_THRESHOLD = 0.3
- BBOX_STD_DEV = np.array([0.1, 0.1, 0.2, 0.2])

```

Są to domyślne parametry sieci, wszelkie zmiany podczas eksperymentów zostaną podane.

Wyjaśnienia znaczenia parametrów:

- BATCH_SIZE - ilość obrazów w jednej serii. Po przetworzeniu każdej serii obrazów przez sieć funkcje strat i wagi są aktualizowane.
- STEPS_PER_EPOCH - liczba iteracji po seriach obrazów w epoce (definiowanej wg pakietu

Keras).

- VALIDATION_STEPS - ilość iteracji po każdej epoce na zbiorze walidacyjnym, wyłącznie w celu przeprowadzenia takich działań jak np. weryfikacja wartości funkcji strat na zbiorze walidacyjnym. Sieć wtedy nie trenuje.
- NUM_CLASSES - ilość klas, do których sieć ma klasyfikować obiekty (wraz z jedną dodatkową klasą zarezerwowaną na tło).
- MEAN_PIXEL - średnia wartość pikseli obrazów z bazy danych *ImageNet* [58], na których (między innymi) pretrenowano użytkę w modelu sieci bazowej.
- IMAGE_CHANNEL_COUNT - sieć przystosowana jest do obrazów RGB, więc wejściowe obrazy mają 3 kanały.
- LEARNING_RATE - wartość współczynnika uczenia, stałego podczas treningu.
- LEARNING_MOMENTUM - momentum uczenia, będące jednym z parametrów optymalizatora SGD.
- WEIGHT_DECAY - parametr używany podczas regularyzacji L2.
- GRADIENT_CLIP_NORM - parametr chroniący przed eksplozją gradientu, po przekroczeniu normy L2 wektora gradientu, skaluje wartości gradientu tak, by jego norma L2 wróciła do prewidzianego zakresu.
- MAX_GT_INSTANCES - maksymalna ilość anotowanych ramek na jednym obrazie, do jakiej przystosowana jest sieć.
- LOSS_WEIGHTS - wagi, przez które mnożone są lokalne funkcje strat.
- BACKBONE - rodzaj użytej sieci bazowej
- BACKBONE_STRIDES - parametr potrzebny do ustalenia rozdzielczości macierzy cech sieci bazowej.
- TOP_DOWN_PYRAMID_SIZE - głębokość warstw sieci FPN.
- TRAIN_ROIS_PER_IMAGE - ilość Rols dla jednego obrazu, jaka będzie przekazana do treningu w module klasyfikatora.
- ROI_POSITIVE_RATIO - oczekiwana proporcja pomiędzy pozytywnymi Rols, a wszystkimi Rols.
- RPN_BBOX_STD_DEV - wektor odchyлеń standardowych dla przewidywanych delt uzyskanych z RPN.
- RPN_ANCHOR_SCALES - skale *anchors* odpowiadające rozdzielczościom poszczególnych warstw FPN.
- RPN_ANCHOR RATIOS - dozwolone stosunki między bokami *anchors*.
- RPN_ANCHOR_STRIDE - parametr determinuje gęstość *anchors*. Domyślnie są tworzone dla każdego piksela mapy cech FPN.
- RPN_NMS_THRESHOLD - próg metody *Non-maximum suppression* podczas filtrowania Rols w warstwie propozycji.
- RPN_TRAIN_ANCHORS_PER_IMAGE - maksymalna ilość używanych *anchors* do treningu RPN.
- PRE_NMS_LIMIT - limit zachowanych Rols przed ich wczytaniem do metody NMS.
- POST_NMS_ROIS_TRAINING - liczba Rols selekcjonowanych przez NMS podczas treningu.
- POST_NMS_ROIS_INFERENCE - liczba Rols selekcjonowanych przez NMS podczas inferencji.
- POOL_SIZE - rozdzielczość (np. 7×7), do jakiej zmniejszany jest wycięty fragment mapy cech (P_2 , P_3 , P_4 lub P_5) w warstwie Rol Pooling.
- DETECTION_MAX_INSTANCES - maksymalna ilość detekcji dla pojedynczego obrazu.

- DETECTION_MIN_CONFIDENCE - minimalny wskaźnik pewności (obliczany w gałęzi modułu klasyfikatora dla danej klasy), aby zaakceptować przewidywaną ramkę.
- DETECTION_NMS_THRESHOLD - próg metody NMS podczas filtrowania Rols w module klasyfikatora.
- BBOX_STD_DEV - wektor odchyлеń standardowych dla przewidywanych delt uzyskanych z modułu klasyfikatora.

4.4. Eksperymenty

4.4.1. Badanie wpływu lustrzanego odbicia

Pierwszy eksperyment przeprowadzono na domyślnych ustawieniach, jednak w związku z parametrami `MAX_GT_INSTANCES = 100, DETECTION_MAX_INSTANCES = 100`, ze zbiorów treningowego i walidacyjnego usunięto odpowiednio 242 oraz 59 obrazów niespełniających powyższych warunków.

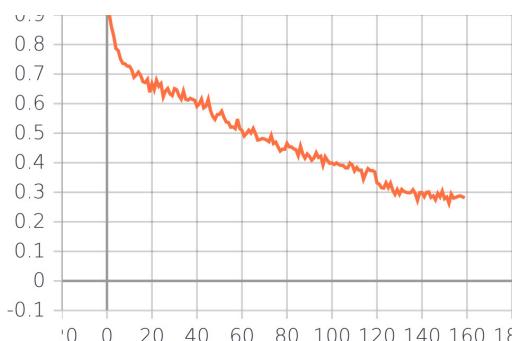
Augmentacja zbioru treningowego pozwala sztucznie zwiększyć ilość danych uczących, przy czym wygenerowane obrazy zawierają jeden lub więcej następujących efektów: zmiana położenia twarzy, zmiana kąta wykonania zdjęcia, manipulacje jasnością, ostrością. Włączenie do treningu pochodzących z augmentacji obrazów pozwala, by sieć osiągała większą dokładność podczas detekcji twarzy na zdjęciach, których specyficzne cechy rzadko występują w zbiorze treningowym.

Zastosowano augmentację on-line, korzystając z pakietu `imgaug` [59]. Tu wprowadzono jedynie pojedynczy efekt `fliplr`, czyli zamianę ze sobą prawej i lewej połowy obrazu poprzez symetryczne odbicie. Każdy wczytywany przez sieć oryginalny obraz zostaje z prawdopodobieństwem $p = 0.4$ wymieniony na wersję pochodzączą z augmentacji:

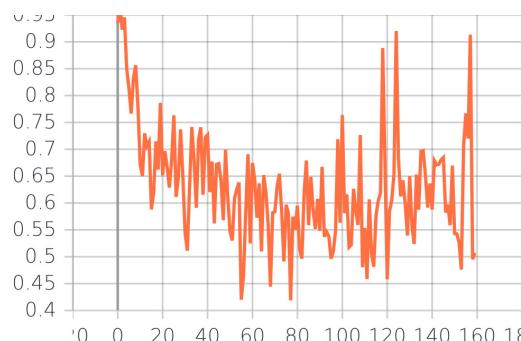
```
seq = iaa.Sequential([iaa.FlipLR(p=0.4)])
```

Implementacja tego rodzaju augmentacji sprawi, że obiekty o tych samych cechach będą podobnie często występować zarówno w prawej, jak i lewej połówce obrazów. Osiągnięte wyniki będą stanowiły dobrą bazę porównawczą dla bardziej skomplikowanych augmentacji w kolejnych eksperymentach.

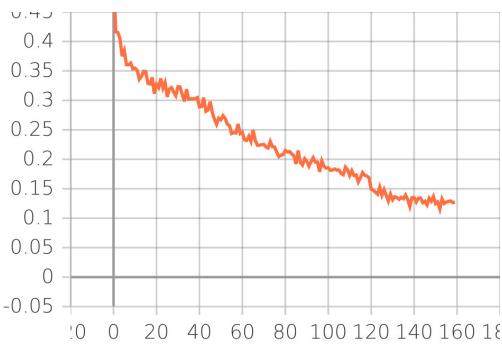
Wykresy funkcji strat w zależności od epoki:



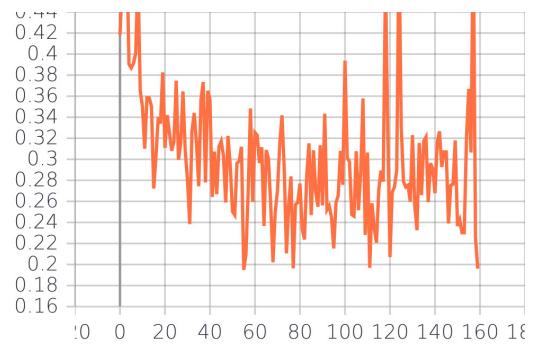
Rys. 43: Całkowita funkcja strat L_{total} dla zbioru treningowego



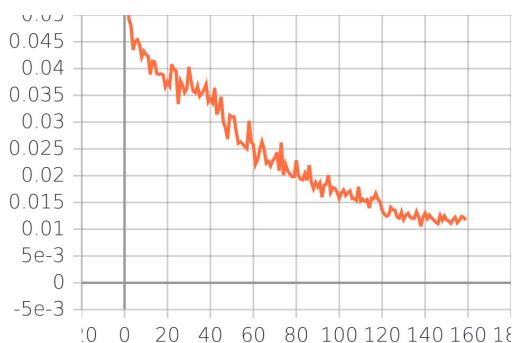
Rys. 44: Całkowita funkcja strat L_{total} dla zbioru walidacyjnego



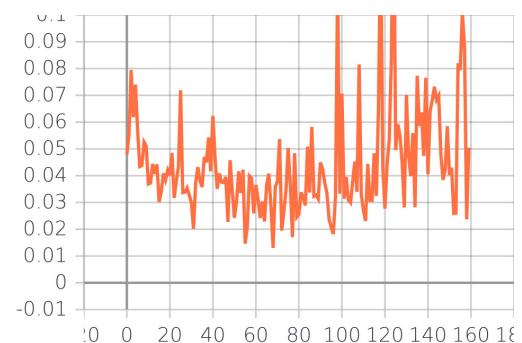
Rys. 45: Funkcja strat L_{bbox1} regresji Rols dla zbioru treningowego



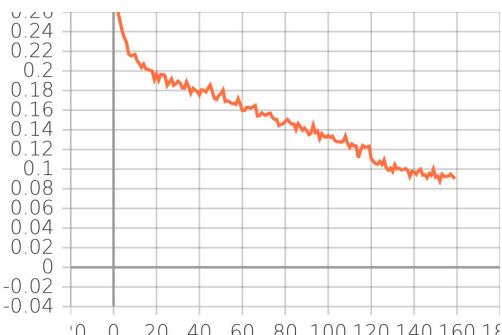
Rys. 46: Funkcja strat L_{bbox1} regresji Rols dla zbioru walidacyjnego



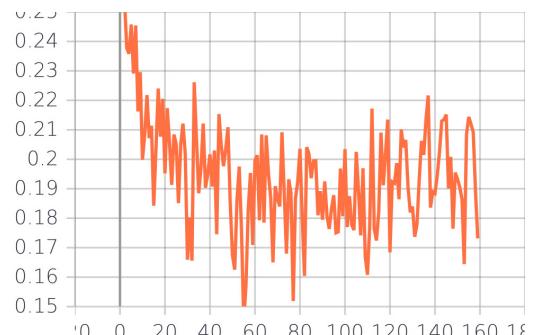
Rys. 47: Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru treningowego



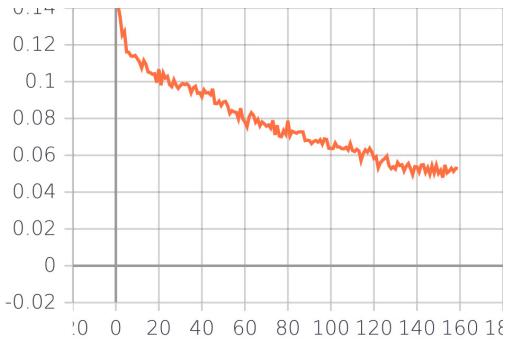
Rys. 48: Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru walidacyjnego



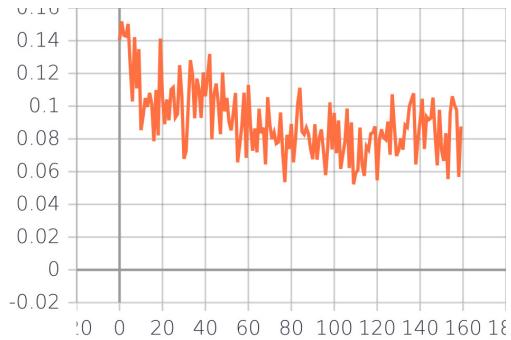
Rys. 49: Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru treningowego



Rys. 50: Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego



Rys. 51: Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru treningowego



Rys. 52: Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego

Wartości funkcji strat po 160 epoce:

$$L_{bbox1}(ep = 160, train) = 0.127$$

$$L_{cls1}(ep = 160, train) = 0.0117$$

$$L_{bbox2}(ep = 160, train) = 0.0902$$

$$L_{cls2}(ep = 160, train) = 0.05284$$

$$L_{total}(ep = 160, train) = 0.2818$$

$$L_{bbox1}(ep = 160, val) = 0.196$$

$$L_{cls1}(ep = 160, val) = 0.05042$$

$$L_{bbox2}(ep = 160, val) = 0.1732$$

$$L_{cls2}(ep = 160, val) = 0.08762$$

$$L_{total}(ep = 160, val) = 0.5072$$

Obliczenia zakończono po 160 epokach. Wraz ze wzrostem liczby epok, wszystkie funkcje strat z (Rys. 43-52) utrzymują tendencje do zmniejszania się, choć w drugiej połowie treningu można dostrzec kilka pików funkcji strat L_{bbox1} , L_{cls1} na zbiorze walidacyjnym. Najprawdopodobniej są spowodowane trafieniem podczas 50 kroków ewaluacyjnych w obrazy o wyjątkowo wysokim stopniu trudności. Na zbiorze walidacyjnym można dostrzec także znacznie większy szum niż na zbiorze treningowym. Są to zwykłe zjawiska dla sieci dokonujących detekcji. Warto także zauważyć, że końcowa wartość funkcji strat $L_{cls1}(ep = 160, val)$ jest 4.3 razy większa od końcowej wartości $L_{cls1}(ep = 160, train)$. Sieć znakomicie odróżnia obiekty od tła na zbiorze treningowym, a nieco gorzej, ale wciąż bardzo dobrze na zbiorze walidacyjnym.

Wytrenowany w tym eksperymencie model sieci osiąga na zbiorze walidacyjnym:

mAP/mAR	maxDet	area	IoU[%]	RESULT
mAP	100	all	50	0.8857
mAP	100	all	75	0.6593
mAP	100	all	50:5:95	0.5914
mAP	100	small	50:5:95	0.2794
mAP	100	medium	50:5:95	0.5666
mAP	100	large	50:5:95	0.6953
mAP	10	all	50	0.9400
mAP	10	all	75	0.7299
mAP	10	all	50:5:95	0.6443
mAP	10	small	50:5:95	0.3139
mAP	10	medium	50:5:95	0.5873
mAP	10	large	50:5:95	0.7027

mAP	1	all	50	0.9651
mAP	1	all	75	0.8242
mAP	1	all	50:5:95	0.7133
mAP	1	small	50:5:95	0.4778
mAP	1	medium	50:5:95	0.5897
mAP	1	large	50:5:95	0.7397
mAR	100	all	50	0.8966
mAR	100	all	75	0.6879
mAR	100	all	50:5:95	0.6121
mAR	100	small	50:5:95	0.3235
mAR	100	medium	50:5:95	0.5912
mAR	100	large	50:5:95	0.7031
mAR	10	all	50	0.9381
mAR	10	all	75	0.7502
mAR	10	all	50:5:95	0.6594
mAR	10	small	50:5:95	0.3460
mAR	10	medium	50:5:95	0.6076
mAR	10	large	50:5:95	0.7101
mAR	1	all	50	0.9687
mAR	1	all	75	0.8265
mAR	1	all	50:5:95	0.7156
mAR	1	small	50:5:95	0.4778
mAR	1	medium	50:5:95	0.5907
mAR	1	large	50:5:95	0.7402

Interpretacja wartości kluczowych miar dla oceny działania sieci:

- mAP @IoU=50: area=all maxDet=100 **0.88570** oznacza, iż większość predykcji zmodyfikowanej sieci Faster R-CNN jest poprawna, a fałszywie pozytywne predykcje zdarzają się stosunkowo rzadko.
- mAR @IoU=50: area=all maxDet=100 **0.8966** ukazuje, iż opracowana sieć wykrywa większość twarzy. Występujące na obrazie twarze są rzadko pomijane.

Te dwie najważniejsze miary świadczą o bardzo dobrych wynikach zmodyfikowanej wersji Faster R-CNN, a także o spełnieniu wszystkich założeń celu pracy przez opracowany system detekcji. Interpretując pozostałe rezultaty dla maxDet=100:

- mAP @IoU=50:5:95: area=all 0.5914 - porównując z mAP @IoU=50: area=all 0.8857, mniejsza wartość tej miary świadczy o odbiegającym od idealnej dokładności ustaleniu współrzędnych przewidywanej ramki. Jest ona jednak wciąż dosyć dobra.
- mAP @IoU=50:5:95: area=small 0.2794 - jest to niewielka wartość w porównaniu z mAP @IoU=50:5:95: area=medium 0.5666 oraz mAP @IoU=50:5:95: area=large 0.6953. Sieć ma problemy z dokładnym wykryciem twarzy o niewielkich rozmiarach. Dużo lepiej radzi sobie ona z detekcją twarzy o średnich i dużych rozmiarach.

Poniżej na Rys. 53, 54 pokazano predykcje sieci dla przykładowych obrazów:



Rys. 53: 14/14 poprawnie przewidzianych twarzy i 1 fałszywie pozytywna (na słupku). Pomimo licznych okluzji i masek policyjnych, sieć poradziła sobie niemal doskonale.

3 Riot Riot 3 522.jpg



Rys. 54: 49/59 poprawnie przewidzianych twarzy i 5 fałszywie pozytywnych.
Małe powierzchnie i okluzje wielu twarzy pogarszają wyniki.

18 Concerts Concerts 18 602.jpg

Dla mniejszej maksymalnej ilości detekcji `maxDet=10` sieć osiąga jeszcze lepsze rezultaty, `mAP @IoU=50 area=all` oraz `mAR @IoU=50 area=all` są większe o ponad 0.07, podobnie zachowują się pozostałe miary.

Obliczone miary dla obrazów zawierających jedną twarz (`maxDet=1`) są podane głównie po-głównie, gdyż w zbiorze walidacyjnym znajduje się bardzo mało takich obrazów. Zgodnie z oczekiwaniami, rezultaty są jeszcze wyższe niż dla `maxDet=10`, co więcej wartości `MAP` oraz `mAR` nie różnią się od siebie znacząco.

Czas detekcji osiągnięty na tym modelu, przy użyciu wcześniej podanych podzespołów komputerowych, wynosi ~0.315s/im.

4.4.2. Badanie wpływu rozmycia i obrotu obrazu

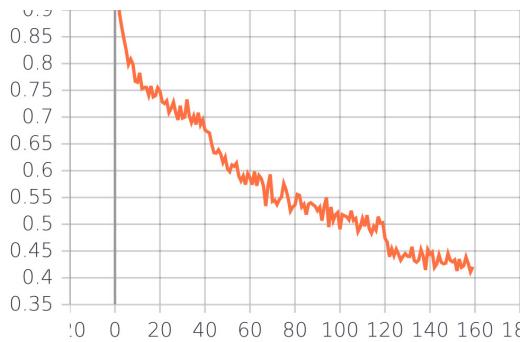
Podczas drugiego eksperymentu, tak jak w pierwszym, ograniczono liczbę anotowanych ramek na zdjęciu do stu stosując domyślne ustawienia, a ze zbiorów treningowego i walidacyjnego usunięto odpowiednio 242 oraz 59 obrazów niespełniających powyższego warunku. Drugi eksperyment jest próbą uzyskania lepszych rezultatów od pierwszego poprzez wprowadzenie wielu typów augmentacji w odpowiednich proporcjach.

Zastosowano kombinację różnych typów augmentacji, z czego każdy typ posiada symetrycznie trójkątny rozkład prawdopodobieństwa. Pozwoli to wytrenować sieć również na obrazach, których właściwości znacznie odbiegają od tych występujących na oryginalnych zdjęciach zbioru treningowego, jednakże większość augmentacji wprowadzi jedynie niewielkie zmiany. Jest to analogia do zdjęć i klatek filmowych tworzonych przez ludzi. Przeważająca większość z nich posiada standardowe właściwości, lecz w niewielkiej części można spotkać różne anomalie, które mogą utrudnić sieci detekcję obiektów. Wprowadzenie kombinacji augmentacji z symetrycznym rozkładem prawdopodobieństwa jest próbą złagodzenia tego niekorzystnego zjawiska.

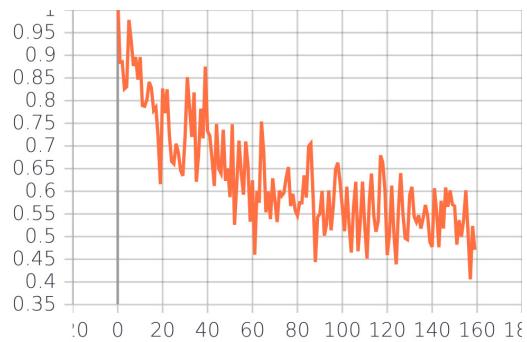
Każdy obraz zostanie poddany jednemu do czterech typów augmentacji. Skorzystano z zmiany prawej i lewej połowy obrazu poprzez symetryczne odbicie, zmiany jasności, rozmycia gaussowskiego oraz zmiany kąta położenia obrazu. Zbyt mała-duża jasność, niewyraźny obraz, zły kąt położenia obrazu to sytuacje, które zdarzają się zwłaszcza niewprawnym fotografom. Mimo niekorzystnie wykonanego obrazu, sieć nadal powinna być w stanie wykryć twarze. Implementacja idei kombinacji typów augmentacji:

```
seq = iaa.Sometimes(p=1, then_list=iaa.SomeOf(n=(1,4), children=[  
    iaa.Fliplr(p=1), iaa.MultiplyBrightness(mul=iap.Uniform((0.5,1),(1,1.5))),  
    iaa.GaussianBlur(sigma=iap.Uniform((0.0,1.5),(1.5,3.0))),  
    iaa.Rotate(rotate=iap.Uniform((-30,0),(0,30)))], random_order=False))
```

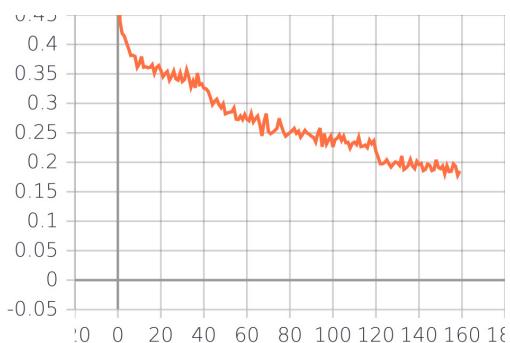
Wykresy funkcji strat w zależności od epoki:



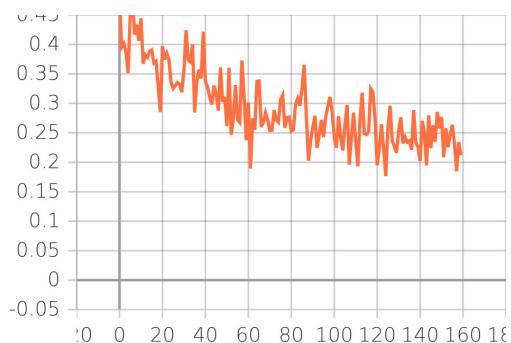
Rys. 55: Całkowita funkcja strat L_{total} dla zbioru treningowego



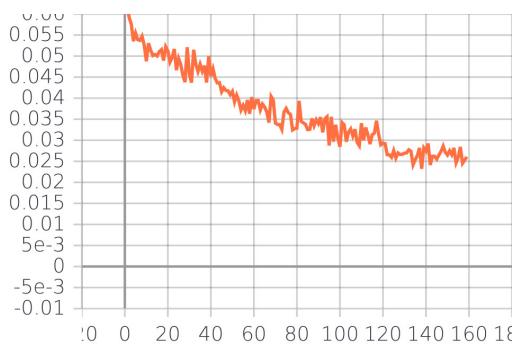
Rys. 56: Całkowita funkcja strat L_{total} dla zbioru walidacyjnego



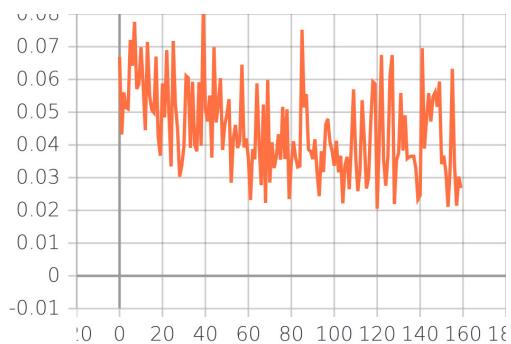
Rys. 57: Funkcja strat L_{bbox1} regresji Rols dla zbioru treningowego



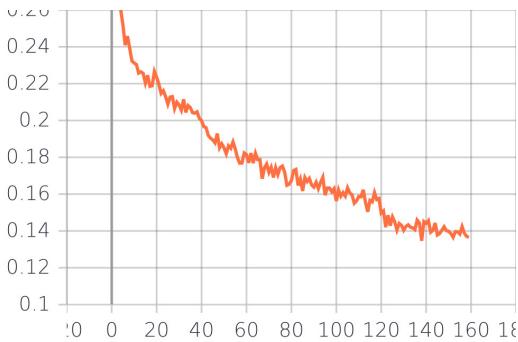
Rys. 58: Funkcja strat L_{bbox1} regresji Rols dla zbioru walidacyjnego



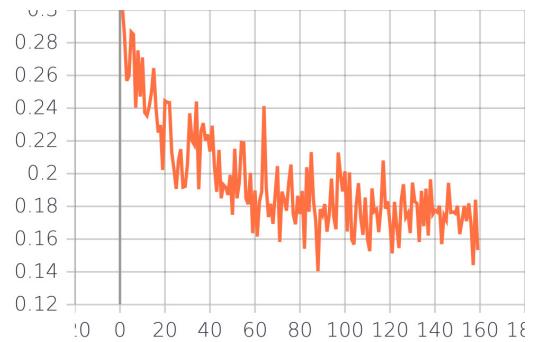
Rys. 59: Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru treningowego



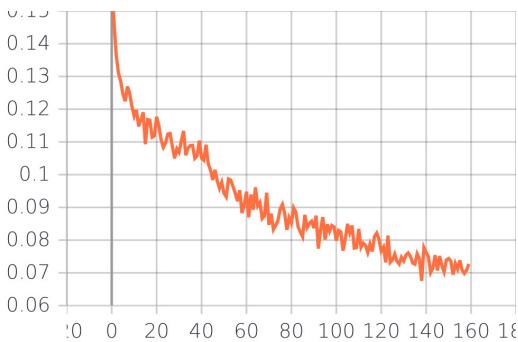
Rys. 60: Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru walidacyjnego



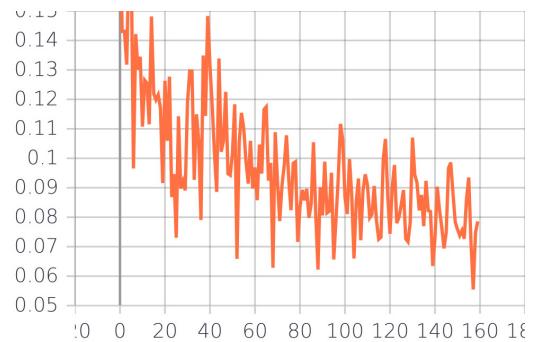
Rys. 61: Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru treningowego



Rys. 62: Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego



Rys. 63: Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru treningowego



Rys. 64: Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego

Wartości funkcji strat po 160 epoce:

$$L_{bbox1}(ep = 160, train) = 0.1852$$

$$L_{cls1}(ep = 160, train) = 0.02606$$

$$L_{bbox2}(ep = 160, train) = 0.1365$$

$$L_{cls2}(ep = 160, train) = 0.07271$$

$$L_{total}(ep = 160, train) = 0.4205$$

$$L_{bbox1}(ep = 160, val) = 0.2121$$

$$L_{cls1}(ep = 160, val) = 0.02686$$

$$L_{bbox2}(ep = 160, val) = 0.1533$$

$$L_{cls2}(ep = 160, val) = 0.07857$$

$$L_{total}(ep = 160, val) = 0.4709$$

Tak jak w poprzednim eksperymencie, wraz ze wzrostem liczby epok, wszystkie funkcje strat z (Rys. 55-64) utrzymują tendencje do zmniejszania się. Tu jednak, dla zbioru walidacyjnego tendencje te są wyraźniejsze dzięki nieco mniejszemu szumowi i braku dużych pików. Choć wartości wszystkich funkcji strat na zbiorze treningowym oraz $L_{bbox1}(ep = 160, val)$ są nieco wyższe od tych z eksperymentu dotyczącego lustrzanego odbicia, pozostałe funkcje strat na zbiorze walidacyjnym są zminimalizowane bardziej. Ma to pewne przełożenie na wyższe wartości mAP i mAR osiągnięte w tym eksperymencie.

Wytrenowany w tym eksperymencie model sieci osiąga na zbiorze walidacyjnym:

mAP/mAR	maxDet	area	IoU[%]	RESULT
mAP	100	all	50	0.8965
mAP	100	all	75	0.6728
mAP	100	all	50:5:95	0.5977
mAP	100	small	50:5:95	0.2851
mAP	100	medium	50:5:95	0.5760
mAP	100	large	50:5:95	0.6991
mAP	10	all	50	0.9450
mAP	10	all	75	0.7488
mAP	10	all	50:5:95	0.6537
mAP	10	small	50:5:95	0.3305
mAP	10	medium	50:5:95	0.5984
mAP	10	large	50:5:95	0.7057
mAP	1	all	50	0.9759
mAP	1	all	75	0.8377
mAP	1	all	50:5:95	0.7182
mAP	1	small	50:5:95	0.5370
mAP	1	medium	50:5:95	0.6018
mAP	1	large	50:5:95	0.7377
mAR	100	all	50	0.9066
mAR	100	all	75	0.6994
mAR	100	all	50:5:95	0.6173
mAR	100	small	50:5:95	0.3254
mAR	100	medium	50:5:95	0.6013
mAR	100	large	50:5:95	0.7060
mAR	10	all	50	0.9520
mAR	10	all	75	0.7664
mAR	10	all	50:5:95	0.6676
mAR	10	small	50:5:95	0.3607
mAR	10	medium	50:5:95	0.6191
mAR	10	large	50:5:95	0.7124
mAR	1	all	50	0.9785
mAR	1	all	75	0.8381
mAR	1	all	50:5:95	0.7190
mAR	1	small	50:5:95	0.5370
mAR	1	medium	50:5:95	0.6019
mAR	1	large	50:5:95	0.7377

Dla $\text{maxDet}=100$, $\text{area}=\text{all}$, wartości mAP @IoU=50: 0.8965, mAR @IoU=50: 0.9066, mAP @IoU=50:5:95: 0.5977 oraz mAR @IoU=50:5:95: 0.6173 są odpowiednio o 0.0108, 0.01, 0.0063, 0.0052 wyższe od wyników bazowych. Wszystkie pozostałe rezultaty za wyjątkiem mAP, mAR dla dużych obiektów na zdjęciach z pojedynczą twarzą, również są wyższe w tym eksper-

mencie. Może to świadczyć o poprawności rozumowania stojącego za zaproponowaną ideą augmentacji, wymagane są jednak szczegółowe badania na ten temat, aby stwierdzić to z całkowitą pewnością.

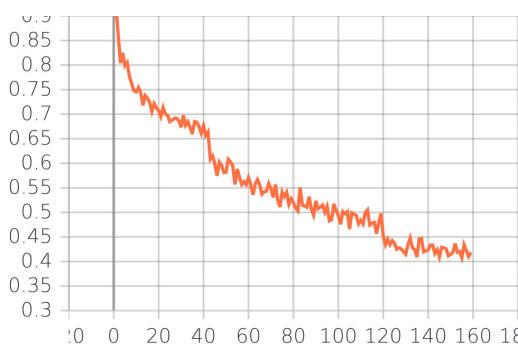
4.4.3. Badanie wpływu zwiększonej liczby detekcji

W tym eksperymencie zwiększoño możliwą liczbę detekcji przez sieć do 200 oraz związane z tym inne parametry. Zastosowano te same rodzaje augmentacji, jak w poprzednim eksperymencie, zmieniono jednak ich rozkład prawdopodobieństwa z symetrycznie trójkątnego na jednostajny w tych samych granicach. Zamiarem takiego działania jest zwiększenie dokładności sieci poprzez wprowadzenie znacznie większej liczby mocno modyfikujących obraz augmentacji do danych uczących.

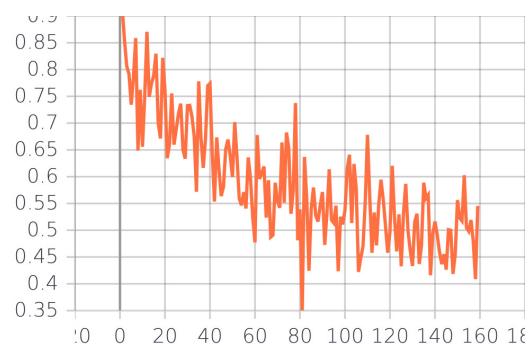
```
MAX_GT_INSTANCES = 200
DETECTION_MAX_INSTANCES = 200
TRAIN_ROIS_PER_IMAGE = 400
RPN_TRAIN_ANCHORS_PER_IMAGE = 512
```

```
seq = iaa.Sometimes(p=1, then_list=iaa.SomeOf(n=(1,4), children=[iaa.Fliplr(p=1),
    iaa.MultiplyBrightness(mul=iap.Uniform(0.5,1.5)),
    iaa.GaussianBlur(sigma=iap.Uniform(0.0, 3.0)),
    iaa.Rotate(rotate=iap.Uniform(-30, 30))], random_order=False))
```

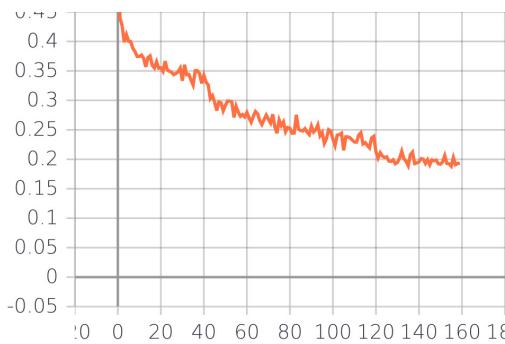
Wykresy funkcji strat w zależności od epoki:



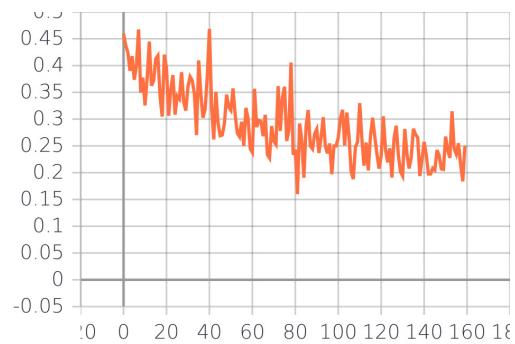
Rys. 65: Całkowita funkcja strat L_{total} dla zbioru treningowego



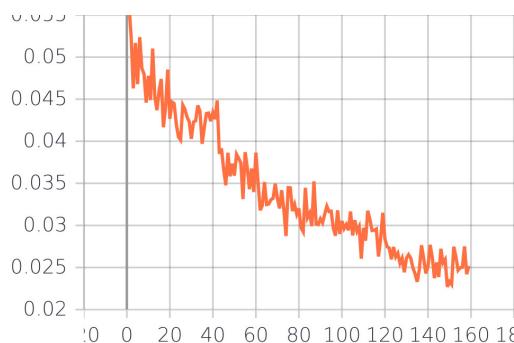
Rys. 66: Całkowita funkcja strat L_{total} dla zbioru walidacyjnego



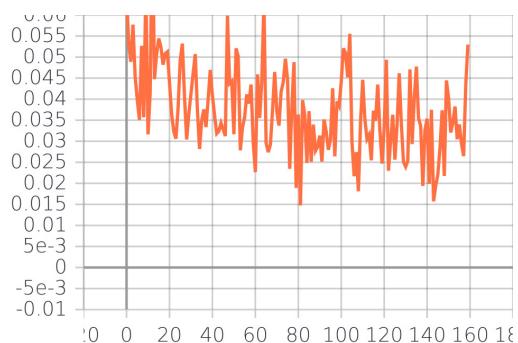
Rys. 67: Funkcja strat L_{bbox1} regresji Rols dla zbioru treningowego



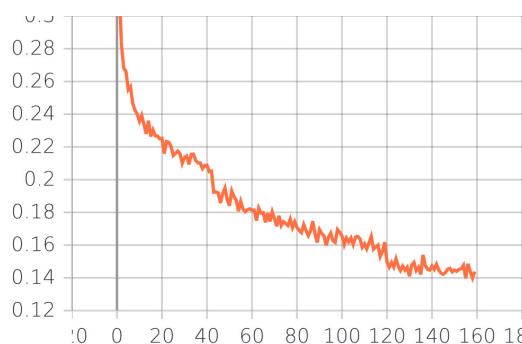
Rys. 68: Funkcja strat L_{bbox1} regresji Rols dla zbioru walidacyjnego



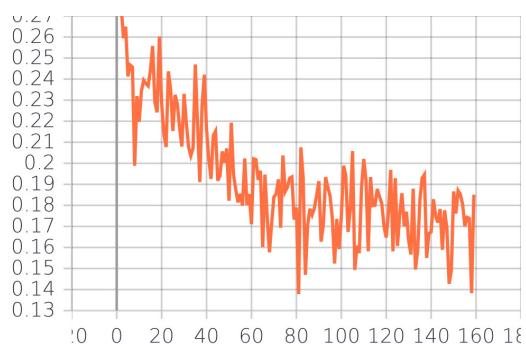
Rys. 69: Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru treningowego



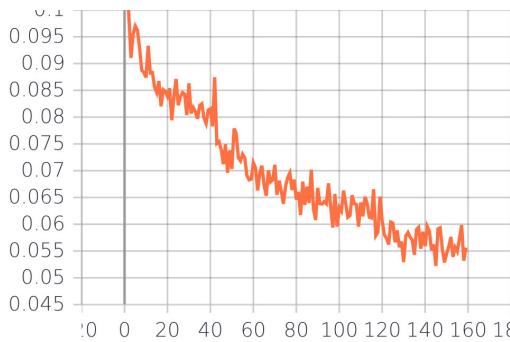
Rys. 70: Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru walidacyjnego



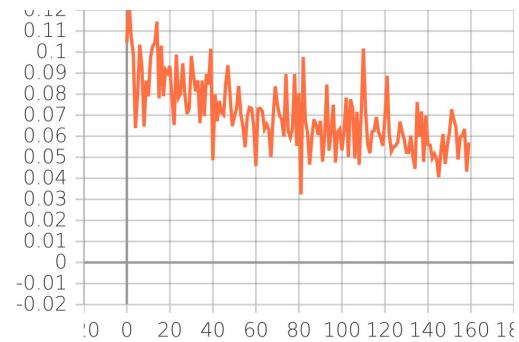
Rys. 71: Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru treningowego



Rys. 72: Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego



Rys. 73: Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru treningowego



Rys. 74: Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego

Wartości funkcji strat po 160 epoce:

$$L_{bbox1}(ep = 160, train) = 0.19308$$

$$L_{cls1}(ep = 160, train) = 0.02517$$

$$L_{bbox2}(ep = 160, train) = 0.1438$$

$$L_{cls2}(ep = 160, train) = 0.05563$$

$$L_{total}(ep = 160, train) = 0.4184$$

$$L_{bbox1}(ep = 160, val) = 0.2502$$

$$L_{cls1}(ep = 160, val) = 0.05297$$

$$L_{bbox2}(ep = 160, val) = 0.1851$$

$$L_{cls2}(ep = 160, val) = 0.05707$$

$$L_{total}(ep = 160, val) = 0.5454$$

Końcowe wartości funkcji strat na zbiorze treningowym są bardzo podobne do tych z poprzedniego eksperymentu, natomiast na zbiorze walidacyjnym są one wyższe (za wyjątkiem L_{cls2}). Jednakże, pomimo tej sytuacji, otrzymane rezultaty są bardzo podobne do wyników badania wpływu rozmycia i obrotu obrazów. Wyższe wartości funkcji strat często towarzyszą niższej dokładności modelu, jednak nie dzieje się tak zawsze - tu jest właśnie ten przypadek.

Wytrenowany w tym eksperymencie model sieci osiąga na zbiorze walidacyjnym:

mAP/mAR	maxDet	area	IoU[%]	RESULT
mAP	200	all	50	0.8917
mAP	200	all	75	0.6671
mAP	200	all	50:5:95	0.5938
mAP	100	all	50	0.8961
mAP	100	all	75	0.6722
mAP	100	all	50:5:95	0.5976
mAP	100	small	50:5:95	0.2843
mAP	100	medium	50:5:95	0.5748
mAP	100	large	50:5:95	0.7082
mAP	10	all	50	0.9451
mAP	10	all	75	0.7498
mAP	10	all	50:5:95	0.6539
mAP	10	small	50:5:95	0.3365
mAP	10	medium	50:5:95	0.5966
mAP	10	large	50:5:95	0.7134

mAP	1	all	50	0.9757
mAP	1	all	75	0.8393
mAP	1	all	50:5:95	0.7210
mAP	1	small	50:5:95	0.5111
mAP	1	medium	50:5:95	0.6029
mAP	1	large	50:5:95	0.7472
mAR	200	all	50	0.9015
mAR	200	all	75	0.6945
mAR	200	all	50:5:95	0.6139
mAR	100	all	50	0.9058
mAR	100	all	75	0.6992
mAR	100	all	50:5:95	0.6176
mAR	100	small	50:5:95	0.3190
mAR	100	medium	50:5:95	0.6003
mAR	100	large	50:5:95	0.7171
mAR	10	all	50	0.9519
mAR	10	all	75	0.7685
mAR	10	all	50:5:95	0.6688
mAR	10	small	50:5:95	0.3613
mAR	10	medium	50:5:95	0.6167
mAR	10	large	50:5:95	0.7219
mAR	1	all	50	0.9785
mAR	1	all	75	0.8408
mAR	1	all	50:5:95	0.7227
mAR	1	small	50:5:95	0.5111
mAR	1	medium	50:5:95	0.6045
mAR	1	large	50:5:95	0.7478

Dla `maxDet=200, area=all`, wartości miar `mAP @IoU=50:`, `mAR @IoU=50:`, `mAP @IoU=50:5:95:` oraz `mAR @IoU=50:5:95:` są niższe od swoich odpowiedników przy `maxDet=100` kolejno o 0.0044, 0.0043, 0.0038, 0.0037. Ten niekorzystny wpływ wywiera większa ilość małych twarzy na obrazach z dużą liczbą obiektów.

Dla `maxDet=100, area=all` wartości miar różnią się o mniej niż tysięczne części od tych z poprzedniego eksperymentu. Takie zjawisko spowodowane jest zapewne zastosowaniem tych samych efektów augmentacji (jedynie ze zmianą rozkładów prawdopodobieństwa).

Przy `maxDet=100`, to badany obecnie model nieco lepiej wykrywa duże obiekty: `mAP @IoU=50:5:95: area=large` jest wyższe o 0.0091 od tego z poprzedniego eksperymentu, natomiast `mAR @IoU=50:5:95: area=large` jest wyższe o 0.0111. Podobne zależności występują przy `maxDet=10`.

4.4.4. Kaskady Haara - porównanie

Warto pokazać, jak niewielką precyzyję posiadały starsze metody detekcji twarzy podczas ich stosowania na wymagających bazach danych.

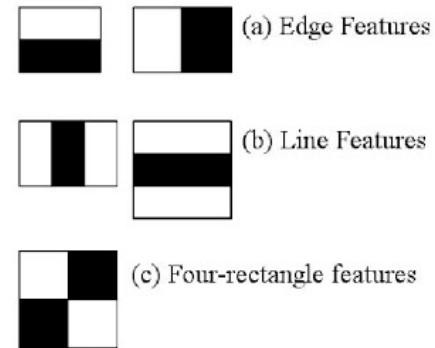
Kaskady Haara [61] do efektywnego zastosowania wymagają konwersji obrazu do skali szarości, co upraszcza komputerowe obliczenia. Przed przejściem do kolejnego etapu, klasyfikator Haara jest trenowany na bardzo dużym zbiorze obrazów, z których wiele twarzy nie zawiera. Następnie dokonuje się ekstrakcji cech interesujących nas części twarzy za pomocą cech Haara. Zachowują się one jak konwolucyjny filtr, ale w odróżnieniu od niego wartości numeryczne cech Haara są zdefiniowane ręcznie. Aby go nie nakładać na każdą grupę pikseli w obrazie twarzy, używany jest algorytm Adaboost, który pozwoli dokonać ekstrakcji najlepszych cech twarzy, tzn. takich, które pozwalają jak najdokładniej podzielić obrazy na zawierające i niezawierające twarze. Wykorzystuje on kaskadę klasyfikatorów pozwalającą na szybkie odrzucenie obszarów zdjęcia niezawierających twarzy. W pakiecie OpenCV istnieje klasyfikator Haara dla frontu twarzy, wytrenowany na bardzo dużym zbiorze danych i gotowy do wykorzystania przez użytkownika. Główną wadą zastosowania OpenCV kaskad Haara w ten sposób, jest brak możliwości dokonania transferu wiedzy i uczenia klasyfikatora na bazie danych użytkownika, w tym przypadku Wider Face.

Przy następujących ustawieniach (plik *haarcascade_frontalface_default.xml*):

```
faces = faceCascade.detectMultiScale(gray, scaleFactor=1.03, minNeighbors=6,
minSize=(24, 24))
```

Wyniki kaskad Haara dla zbioru walidacyjnego Wider Face:

mAP/mAR	maxDet	area	IoU[%]	RESULT
mAP	100	all	50	0.2426
mAP	100	all	75	0.0508
mAP	100	all	50:5:95	0.1042
mAP	100	small	50:5:95	0.0267
mAP	100	medium	50:5:95	0.0979
mAP	100	large	50:5:95	0.1987
mAP	10	all	50	0.2555
mAP	10	all	75	0.0559
mAP	10	all	50:5:95	0.1110
mAP	10	small	50:5:95	0.0290



Rys. 75: Graficzna reprezentacja cech Haara, źródło: [60]

mAP	10	medium	50:5:95	0.0927
mAP	10	large	50:5:95	0.2025
mAP	1	all	50	0.2792
mAP	1	all	75	0.0709
mAP	1	all	50:5:95	0.1261
mAP	1	small	50:5:95	0.0654
mAP	1	medium	50:5:95	0.551
mAP	1	large	50:5:95	0.2236
mAR	100	all	50	0.4422
mAR	100	all	75	0.1114
mAR	100	all	50:5:95	0.1983
mAR	100	small	50:5:95	0.0311
mAR	100	medium	50:5:95	0.1631
mAR	100	large	50:5:95	0.2722
mAR	10	all	50	0.4913
mAR	10	all	75	0.1207
mAR	10	all	50:5:95	0.2198
mAR	10	small	50:5:95	0.0320
mAR	10	medium	50:5:95	0.1584
mAR	10	large	50:5:95	0.2775
mAR	1	all	50	0.5787
mAR	1	all	75	0.1360
mAR	1	all	50:5:95	0.2600
mAR	1	small	50:5:95	0.0654
mAR	1	medium	50:5:95	0.0942
mAR	1	large	50:5:95	0.3080

Zastosowanie kaskad Haara daje dużo gorsze rezultaty od któregokolwiek z wytrenowanych wcześniej modeli Faster R-CNN.

5. Podsumowanie

Celem tej pracy było stworzenie opartego o metody głębokiego uczenia systemu detekcji twarzy, który potrafi z dużą dokładnością wykryć je na kolejnych klatkach filmu, nawet wtedy, gdy twarze są niewyraźne, częściowo zasłonięte i jest ich kilkadziesiąt.

Zaproponowano autorską metodę detekcji twarzy za pomocą zmodyfikowanej wersji sieci Faster R-CNN. Wyniki detekcji są bardzo dobre ($mAP @IoU=50: 0.8965$) i cel pracy został osiągnięty. Zmiany polegają przede wszystkim na zastąpieniu nowszymi metodami z sieci Mask R-CNN kilku niedoskonałych procedur oryginalnie stosowanych w Faster R-CNN. Na tak skonstruowanej sieci, przeprowadzono trzy eksperymenty, różniące się głównie rodzajami augmentacji. W każdym z nich, z naddatkiem spełniono założenia sformułowane w celu pracy. Zmodyfikowana sieć Faster R-CNN potrafi z dużą dokładnością wykryć twarze na obrazach, także wtedy, gdy są niewyraźne, częściowo zasłonięte lub występują w dużej ilości. Wykrywane są twarze o różnych rozmiarach, często w nietypowych pozach. Każdej z twarzy przypisywane są wskaźnik pewności, konfiguracyjnie większy od 0.7, oraz prostokątna ramka zdefiniowana czterema współrzędnymi.

Do treningu i ewaluacji działania sieci użyto wymagającej bazy danych Wider Face. Składa się ona z 16098 poprawnie anotowanych obrazów, wśród których wiele zawiera dziesiątki twarzy niewielkich rozmiarów, zdarzają się też twarze niewyraźne, bądź w niestandardowej pozycji oraz okluzje. Najlepsze wyniki osiągnięto podczas badania wpływu rozmycia i obrotu obrazu:

- $mAP @IoU=50: \text{area=all} \quad \text{maxDet}=100 \quad 0.8965$
- $mAP @IoU=50:5:95: \text{area=all} \quad \text{maxDet}=100 \quad 0.5977$
- $mAR @IoU=50: \text{area=all} \quad \text{maxDet}=100 \quad 0.9066$
- $mAR @IoU=50:5:95: \text{area=all} \quad \text{maxDet}=100 \quad 0.6173$

Zwracając uwagę na dużą ilość twarzy o wysokiej skali trudności zawartych w bazie Wider Face, są to bardzo dobre rezultaty.

Implementacja zmodyfikowanej wersji Faster R-CNN dostępna jest tu: github.com/mszymonowicz/Mask_RCNN. Oprócz wykrywania twarzy na pojedynczych obrazach, możliwa jest taka konwersja filmu, że po jej zakończeniu na kolejnych klatkach do twarzy przypisane są prostokątne ramki wraz ze wskaźnikami pewności, a ścieżka dźwiękowa zostaje zachowana. Ponadto, do użytku oddany jest tryb wykrywania twarzy na klatkach przechwytywanych z podłączonej do komputera kamery. Dla użytkowników posiadających najlepsze karty graficzne, detekcja twarzy może odbywać się niemal w czasie rzeczywistym.

W niedalekiej przyszłości, autor zamierza dużo bardziej szczegółowo zbadać wpływ różnych rodzajów augmentacji na otrzymywane wyniki, a także przeprowadzić eksperymenty dotyczące ablacji.

Autor zachęca Czytelników do przeprowadzania własnych eksperymentów i zgłaszania wszelkich problemów, nieścisłości, bądź propozycji usprawnień w zmodyfikowanej wersji Faster R-CNN.

6. Dodatek A

Program napisano w *Python3.6*, przy użyciu bibliotek *Tensorflow-gpu 1.14*, *Keras 2.2.4*. Jest on dostępny pod linkiem: github.com/mszymanowicz/Mask_RCNN. Duże podziękowania dla autora implementacji *Mask R-CNN*, której publicznie dostępne repozytorium można znaleźć tu: [62], a wiele fragmentów kodu zostało z niego zapożyczonych.

Są jednak między tym programem, a implementacją *Mask R-CNN* [62] znaczne różnice. Przede wszystkim usunięto wszystkie gałęzie i warstwy odpowiedzialne za przetwarzanie masek, efektywnie degradując sieć do postaci Faster R-CNN, zachowując jednak kilka ulepszeń opisanych w **Rozdziale 3**. Ponadto, dodano przełączanie między rodzajami augmentacji, usunięto usterki powodujące czasem niepoprawne przeskalowanie ramek podczas zmiany rozdzielczości obrazu przed wejściem do sieci. Stworzono także w pliku *video.py* możliwość przetworzenia filmu, gdzie na każdą klatkę nakładane są wyniki detekcji twarzy przez sieć oraz podjęcie próby detekcji twarzy w czasie rzeczywistym (do czego potrzebna jest bardzo dobra karta graficzna).

Z powodu znacznej objętości programu, w tym dodatku umieszczone są najistotniejsze z punktu opisu zastosowanej metody fragmenty kodu.

model.py to główny plik programu, w którym zawarto strukturę modelu, obliczenia dokonywane w warstwach i wywołania zwrotne (*callbacks*).

Budowa sieci bazowej ResNet-101:

```
def resnet_graph(input_image, architecture, stage5=False, train_bn=True):
    """Build a ResNet graph.
    architecture: Can be resnet50 or resnet101
    stage5: Boolean. If False, stage5 of the network is not created
    train_bn: Boolean. Train or freeze Batch Norm layers
    """
    assert architecture in ["resnet50", "resnet101"]

    # Stage 1
    x = KL.ZeroPadding2D((3, 3))(input_image)
    x = KL.Conv2D(64, (7, 7), strides=(2, 2), name='conv1', use_bias=True)(x)
    x = BatchNorm(name='bn_conv1')(x, training=train_bn)
    x = KL.Activation('relu')(x)
    C1 = x = KL.MaxPooling2D((3, 3), strides=(2, 2), padding="same")(x)

    # Stage 2
    x = conv_block(x, 3, [64, 64, 256], stage=2, block='a', strides=(1, 1),
                   train_bn=train_bn)
    x = identity_block(x, 3, [64, 64, 256], stage=2, block='b', train_bn=train_bn)
    C2 = x = identity_block(x, 3, [64, 64, 256], stage=2, block='c', train_bn=
                           train_bn)

    # Stage 3
    x = conv_block(x, 3, [128, 128, 512], stage=3, block='a', train_bn=train_bn)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='b', train_bn=
                           train_bn)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='c', train_bn=
                           train_bn)
    C3 = x = identity_block(x, 3, [128, 128, 512], stage=3, block='d', train_bn=
                           train_bn)
```

```

# Stage 4
x = conv_block(x, 3, [256, 256, 1024], stage=4, block='a', train_bn=train_bn)
block_count = {"resnet50": 5, "resnet101": 22}[architecture]
for i in range(block_count):
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block=chr(98 + i),
                        train_bn=train_bn)
C4 = x
# Stage 5
if stage5:
    x = conv_block(x, 3, [512, 512, 2048], stage=5, block='a', train_bn=train_bn)
    x = identity_block(x, 3, [512, 512, 2048], stage=5, block='b', train_bn=
                        train_bn)
    C5 = x = identity_block(x, 3, [512, 512, 2048], stage=5, block='c', train_bn
                            =train_bn)
else:
    C5 = None
return [C1, C2, C3, C4, C5]

```

```

def conv_block(input_tensor, kernel_size, filters, stage, block, strides=(2, 2),
               use_bias=True, train_bn=True):
    """conv_block is the block that has a conv layer at shortcut
    # Arguments
        input_tensor: input tensor
        kernel_size: default 3, the kernel size of middle conv layer at main path
        filters: list of integers, the nb_filters of 3 conv layer at main path
        stage: integer, current stage label, used for generating layer names
        block: 'a','b'..., current block label, used for generating layer names
        use_bias: Boolean. To use or not use a bias in conv layers.
        train_bn: Boolean. Train or freeze Batch Norm layers
    Note that from stage 3, the first conv layer at main path is with subsample=(2
        ,2)
    And the shortcut should have subsample=(2,2) as well
    """
    nb_filter1, nb_filter2, nb_filter3 = filters
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = KL.Conv2D(nb_filter1, (1, 1), strides=strides, name=conv_name_base + '2a',
                 use_bias=use_bias)(input_tensor)
    x = BatchNorm(name=bn_name_base + '2a')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same', name=
                  conv_name_base + '2b', use_bias=use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2b')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c', use_bias=
                  use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2c')(x, training=train_bn)

    shortcut = KL.Conv2D(nb_filter3, (1, 1), strides=strides,
                       name=conv_name_base + '1', use_bias=use_bias)(input_tensor)
    shortcut = BatchNorm(name=bn_name_base + '1')(shortcut, training=train_bn)

```

```

x = KL.Add()(x, shortcut)
x = KL.Activation('relu', name='res' + str(stage) + block + '_out')(x)
return x

```

```

def identity_block(input_tensor, kernel_size, filters, stage, block, use_bias=True, train_bn=True):
    """The identity_block is the block that has no conv layer at shortcut
    # Arguments
    input_tensor: input tensor
    kernel_size: default 3, the kernel size of middle conv layer at main path
    filters: list of integers, the nb_filters of 3 conv layer at main path
    stage: integer, current stage label, used for generating layer names
    block: 'a','b'..., current block label, used for generating layer names
    use_bias: Boolean. To use or not use a bias in conv layers.
    train_bn: Boolean. Train or freeze Batch Norm layers
    """
    nb_filter1, nb_filter2, nb_filter3 = filters
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = KL.Conv2D(nb_filter1, (1, 1), name=conv_name_base + '2a', use_bias=use_bias)(input_tensor)
    x = BatchNorm(name=bn_name_base + '2a')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same', name=conv_name_base + '2b', use_bias=use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2b')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c', use_bias=use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2c')(x, training=train_bn)

    x = KL.Add()(x, input_tensor)
    x = KL.Activation('relu', name='res' + str(stage) + block + '_out')(x)
    return x

```

Warstwy sieci FPN:

```

P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c5p5')(C5)
P4 = KL.Add(name="fpn_p4add")([KL.UpSampling2D(size=(2, 2), name="fpn_p5upsampled")(P5), KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c4p4')(C4)])
P3 = KL.Add(name="fpn_p3add")([KL.UpSampling2D(size=(2, 2), name="fpn_p4upsampled")(P4), KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c3p3')(C3)])
P2 = KL.Add(name="fpn_p2add")([KL.UpSampling2D(size=(2, 2), name="fpn_p3upsampled")(P3), KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c2p2')(C2)])
# Attach 3x3 conv to all P layers to get the final feature maps.
P2 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p2")(P2)

```

```

P3 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p3")(P3)
P4 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p4")(P4)
P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p5")(P5)
# P6 is used for the 5th anchor scale in RPN. Generated by
# subsampling from P5 with stride of 2.
P6 = KL.MaxPooling2D(pool_size=(1, 1), strides=2, name="fpn_p6")(P5)

# Note that P6 is used in RPN, but not in the classifier heads.
rpn_feature_maps = [P2, P3, P4, P5, P6]
mrcnn_feature_maps = [P2, P3, P4, P5]

```

Budowa sieci RPN:

```

def rpn_graph(feature_map, anchors_per_location, anchor_stride):
    """Builds the computation graph of Region Proposal Network.

    feature_map: backbone features [batch, height, width, depth]
    anchors_per_location: number of anchors per pixel in the feature map
    anchor_stride: Controls the density of anchors. Typically 1 (anchors for
    every pixel in the feature map), or 2 (every other pixel).

    Returns:
    rpn_class_logits: [batch, H * W * anchors_per_location, 2] Anchor classifier
        logits (before softmax)
    rpn_probs: [batch, H * W * anchors_per_location, 2] Anchor classifier
        probabilities.
    rpn_bbox: [batch, H * W * anchors_per_location, (dy, dx, log(dh), log(dw))]
        Deltas to be
        applied to anchors.
    """

    # Shared convolutional base of the RPN
    shared = KL.Conv2D(512, (3, 3), padding='same', activation='relu', strides=
        anchor_stride, name='rpn_conv_shared')(feature_map)

    # Anchor Score. [batch, height, width, anchors per location * 2].
    x = KL.Conv2D(2 * anchors_per_location, (1, 1), padding='valid', activation='
        linear', name='rpn_class_raw')(shared)

    # Reshape to [batch, anchors, 2]
    rpn_class_logits = KL.Lambda(lambda t: tf.reshape(t, [tf.shape(t)[0], -1, 2]))(x)

    # Softmax on last dimension of BG/FG.
    rpn_probs = KL.Activation("softmax", name="rpn_class_xxx")(rpn_class_logits)

    # Bounding box refinement. [batch, H, W, anchors per location * depth]
    # where depth is [x, y, log(w), log(h)]
    x = KL.Conv2D(anchors_per_location * 4, (1, 1), padding="valid", activation='
        linear', name='rpn_bbox_pred')(shared)

    # Reshape to [batch, anchors, 4]
    rpn_bbox = KL.Lambda(lambda t: tf.reshape(t, [tf.shape(t)[0], -1, 4]))(x)

```

```
    return [rpn_class_logits, rpn_probs, rpn_bbox]
```

Warstwa propozycji:

```
class ProposalLayer(KE.Layer):
    """Receives anchor scores and selects a subset to pass as proposals
    to the second stage. Filtering is done based on anchor scores and
    non-max suppression to remove overlaps. It also applies bounding
    box refinement deltas to anchors.

    Inputs:
    rpn_probs: [batch, num_anchors, (bg prob, fg prob)]
    rpn_bbox: [batch, num_anchors, (dy, dx, log(dh), log(dw))]
    anchors: [batch, num_anchors, (y1, x1, y2, x2)] anchors in normalized
             coordinates

    Returns:
    Proposals in normalized coordinates [batch, rois, (y1, x1, y2, x2)]
    """

    def __init__(self, proposal_count, nms_threshold, config=None, **kwargs):
        super(ProposalLayer, self).__init__(**kwargs)
        self.config = config
        self.proposal_count = proposal_count
        self.nms_threshold = nms_threshold

    def call(self, inputs):
        # Box Scores. Use the foreground class confidence. [Batch, num_rois, 1]
        scores = inputs[0][:, :, 1]
        # Box deltas [batch, num_rois, 4]
        deltas = inputs[1]
        deltas = deltas * np.reshape(self.config.RPN_BOX_STD_DEV, [1, 1, 4])
        # Anchors
        anchors = inputs[2]

        # Improve performance by trimming to top anchors by score
        # and doing the rest on the smaller subset.
        pre_nms_limit = tf.minimum(self.config.PRE_NMS_LIMIT, tf.shape(anchors)[1])
        ix = tf.nn.top_k(scores, pre_nms_limit, sorted=True, name="top_anchors").indices
        scores = utils.batch_slice([scores, ix], lambda x, y: tf.gather(x, y), self.config.IMAGES_PER_GPU)
        deltas = utils.batch_slice([deltas, ix], lambda x, y: tf.gather(x, y), self.config.IMAGES_PER_GPU)
        pre_nms_anchors = utils.batch_slice([anchors, ix], lambda a, x: tf.gather(a, x), self.config.IMAGES_PER_GPU,
                                            names=["pre_nms_anchors"])

        # Apply deltas to anchors to get refined anchors.
        # [batch, N, (y1, x1, y2, x2)]
        boxes = utils.batch_slice([pre_nms_anchors, deltas],
                                 lambda x, y: apply_box_deltas_graph(x, y),
                                 self.config.IMAGES_PER_GPU,
                                 names=["refined_anchors"])
```

```

# Clip to image boundaries. Since we're in normalized coordinates,
# clip to 0..1 range. [batch, N, (y1, x1, y2, x2)]
window = np.array([0, 0, 1, 1], dtype=np.float32)
boxes = utils.batch_slice(boxes,
lambda x: clip_boxes_graph(x, window),
self.config.IMAGES_PER_GPU,
names=["refined_anchors_clipped"])

# Filter out small boxes
# According to Xinlei Chen's paper, this reduces detection accuracy
# for small objects, so we're skipping it.

# Non-max suppression
def nms(boxes, scores):
    indices = tf.image.non_max_suppression(
        boxes, scores, self.proposal_count,
        self.nms_threshold, name="rpn_non_max_suppression")
    proposals = tf.gather(boxes, indices)
    # Pad if needed
    padding = tf.maximum(self.proposal_count - tf.shape(proposals)[0], 0)
    proposals = tf.pad(proposals, [(0, padding), (0, 0)])
    return proposals
    proposals = utils.batch_slice([boxes, scores], nms,
        self.config.IMAGES_PER_GPU)
    return proposals

def compute_output_shape(self, input_shape):
    return (None, self.proposal_count, 4)

```

Graf detekcji celów (w obrębie warstwy detekcji celów):

```

def detection_targets_graph(proposals, gt_class_ids, gt_boxes, config, gt_masks=None):
    """Generates detection targets for one image. Subsamples proposals and
    generates target class IDs, bounding box deltas for each.

    Inputs:
    proposals: [POST_NMS_ROIS_TRAINING, (y1, x1, y2, x2)] in normalized
    coordinates. Might
    be zero padded if there are not enough proposals.
    gt_class_ids: [MAX_GT_INSTANCES] int class IDs
    gt_boxes: [MAX_GT_INSTANCES, (y1, x1, y2, x2)] in normalized coordinates.

    Returns: Target ROIs and corresponding class IDs, bounding box shifts.
    rois: [TRAIN_ROIS_PER_IMAGE, (y1, x1, y2, x2)] in normalized coordinates
    class_ids: [TRAIN_ROIS_PER_IMAGE]. Integer class IDs. Zero padded.
    deltas: [TRAIN_ROIS_PER_IMAGE, (dy, dx, log(dh), log(dw))]

    Note: Returned arrays might be zero padded if not enough target ROIs.
    """
    # Assertions
    asserts = [tf.Assert(tf.greater(tf.shape(proposals)[0], 0), [proposals], name="roi_assertion")]
    with tf.control_dependencies(asserts):

```

```

proposals = tf.identity(proposals)

# Remove zero padding
proposals, _ = trim_zeros_graph(proposals, name="trim_proposals")
gt_boxes, non_zeros = trim_zeros_graph(gt_boxes, name="trim_gt_boxes")
gt_class_ids = tf.boolean_mask(gt_class_ids, non_zeros, name="trim_gt_class_ids")

# Handle COCO crowds
# A crowd box in COCO is a bounding box around several instances. Exclude
# them from training. A crowd box is given a negative class ID.
crowd_ix = tf.where(gt_class_ids < 0)[:, 0]
non_crowd_ix = tf.where(gt_class_ids > 0)[:, 0]
crowd_boxes = tf.gather(gt_boxes, crowd_ix)
gt_class_ids = tf.gather(gt_class_ids, non_crowd_ix)
gt_boxes = tf.gather(gt_boxes, non_crowd_ix)

# Compute overlaps matrix [proposals, gt_boxes]
overlaps = overlaps_graph(proposals, gt_boxes)

# Compute overlaps with crowd boxes [proposals, crowd_boxes]
crowd_overlaps = overlaps_graph(proposals, crowd_boxes)
crowd_iou_max = tf.reduce_max(crowd_overlaps, axis=1)
no_crowd_bool = (crowd_iou_max < 0.001)

# Determine positive and negative ROIs
roi_iou_max = tf.reduce_max(overlaps, axis=1)
# 1. Positive ROIs are those with >= 0.5 IoU with a GT box
positive_roi_bool = (roi_iou_max >= 0.5)
positive_indices = tf.where(positive_roi_bool)[:, 0]
# 2. Negative ROIs are those with < 0.5 with every GT box. Skip crowds.
negative_indices = tf.where(tf.logical_and(roi_iou_max < 0.5, no_crowd_bool))[:, 0]

# Subsample ROIs. Aim for 33% positive
# Positive ROIs
positive_count = int(config.TRAIN_ROIS_PER_IMAGE * config.ROI_POSITIVE_RATIO)
positive_indices = tf.random_shuffle(positive_indices)[:positive_count]
positive_count = tf.shape(positive_indices)[0]
# Negative ROIs. Add enough to maintain positive:negative ratio.
r = 1.0 / config.ROI_POSITIVE_RATIO
negative_count = tf.cast(r * tf.cast(positive_count, tf.float32), tf.int32) -
    positive_count
negative_indices = tf.random_shuffle(negative_indices)[:negative_count]
# Gather selected ROIs
positive_rois = tf.gather(proposals, positive_indices)
negative_rois = tf.gather(proposals, negative_indices)

# Assign positive ROIs to GT boxes.
positive_overlaps = tf.gather(overlaps, positive_indices)
roi_gt_box_assignment = tf.cond(tf.greater(tf.shape(positive_overlaps)[1], 0),
    true_fn = lambda: tf.argmax(positive_overlaps, axis=1), false_fn = lambda
    : tf.cast(tf.constant([]), tf.int64))
roi_gt_boxes = tf.gather(gt_boxes, roi_gt_box_assignment)
roi_gt_class_ids = tf.gather(gt_class_ids, roi_gt_box_assignment)

```

```

# Compute bbox refinement for positive ROIs
deltas = utils.box_refinement_graph(positive_rois, roi_gt_boxes)
deltas /= config.BBOX_STD_DEV

# Append negative ROIs and pad bbox deltas that
# are not used for negative ROIs with zeros.
rois = tf.concat([positive_rois, negative_rois], axis=0)
N = tf.shape(negative_rois)[0]
P = tf.maximum(config.TRAIN_ROIS_PER_IMAGE - tf.shape(rois)[0], 0)
rois = tf.pad(rois, [(0, P), (0, 0)])
roi_gt_boxes = tf.pad(roi_gt_boxes, [(0, N + P), (0, 0)])
roi_gt_class_ids = tf.pad(roi_gt_class_ids, [(0, N + P)])
deltas = tf.pad(deltas, [(0, N + P), (0, 0)])

return rois, roi_gt_class_ids, deltas

```

Warstwa ROI Pooling (metoda ROIAlign):

```

class PyramidROIAlign(KER.Layer):
    """Implements ROI Pooling on multiple levels of the feature pyramid.

    Params:
    - pool_shape: [pool_height, pool_width] of the output pooled regions. Usually
      [7, 7]

    Inputs:
    - boxes: [batch, num_boxes, (y1, x1, y2, x2)] in normalized
      coordinates. Possibly padded with zeros if not enough
      boxes to fill the array.
    - image_meta: [batch, (meta data)] Image details. See compose_image_meta()
    - feature_maps: List of feature maps from different levels of the pyramid.
      Each is [batch, height, width, channels]

    Output:
    Pooled regions in the shape: [batch, num_boxes, pool_height, pool_width,
      channels].
    The width and height are those specific in the pool_shape in the layer
    constructor.
    """

    def __init__(self, pool_shape, **kwargs):
        super(PyramidROIAlign, self).__init__(**kwargs)
        self.pool_shape = tuple(pool_shape)

    def call(self, inputs):
        # Crop boxes [batch, num_boxes, (y1, x1, y2, x2)] in normalized coords
        boxes = inputs[0]

        # Image meta
        # Holds details about the image. See compose_image_meta()
        image_meta = inputs[1]

        # Feature Maps. List of feature maps from different level of the
        # feature pyramid. Each is [batch, height, width, channels]

```

```

feature_maps = inputs[2:]

# Assign each ROI to a level in the pyramid based on the ROI area.
y1, x1, y2, x2 = tf.split(boxes, 4, axis=2)
h = y2 - y1
w = x2 - x1
# Use shape of first image. Images in a batch must have the same size.
image_shape = parse_image_meta_graph(image_meta)['image_shape'][0]
# Equation 1 in the Feature Pyramid Networks paper. Account for
# the fact that our coordinates are normalized here.
# e.g. a 224x224 ROI (in pixels) maps to P4
image_area = tf.cast(image_shape[0] * image_shape[1], tf.float32)
roi_level = log2_graph(tf.sqrt(h * w) / (224.0 / tf.sqrt(image_area)))
roi_level = tf.minimum(5, tf.maximum(2, 4 + tf.cast(tf.round(roi_level), tf.int32)))
roi_level = tf.squeeze(roi_level, 2)

# Loop through levels and apply ROI pooling to each. P2 to P5.
pooled = []
box_to_level = []
for i, level in enumerate(range(2, 6)):
    ix = tf.where(tf.equal(roi_level, level))
    level_boxes = tf.gather_nd(boxes, ix)

    # Box indices for crop_and_resize.
    box_indices = tf.cast(ix[:, 0], tf.int32)

    # Keep track of which box is mapped to which level
    box_to_level.append(ix)

    # Stop gradient propagation to ROI proposals
    level_boxes = tf.stop_gradient(level_boxes)
    box_indices = tf.stop_gradient(box_indices)

    # Crop and Resize
    # From Mask R-CNN paper: "We sample four regular locations, so
    # that we can evaluate either max or average pooling. In fact,
    # interpolating only a single value at each bin center (without
    # pooling) is nearly as effective."
    #
    # Here we use the simplified approach of a single value per bin,
    # which is how it's done in tf.crop_and_resize()
    # Result: [batch * num_boxes, pool_height, pool_width, channels]
    pooled.append(tf.image.crop_and_resize(feature_maps[i], level_boxes,
                                           box_indices, self.pool_shape, method="bilinear"))

# Pack pooled features into one tensor
pooled = tf.concat(pooled, axis=0)

# Pack box_to_level mapping into one array and add another
# column representing the order of pooled boxes
box_to_level = tf.concat(box_to_level, axis=0)
box_range = tf.expand_dims(tf.range(tf.shape(box_to_level)[0]), 1)
box_to_level = tf.concat([tf.cast(box_to_level, tf.int32), box_range], axis=1)

```

```

# Rearrange pooled features to match the order of the original boxes
# Sort box_to_level by batch then box index
# TF doesn't have a way to sort by two columns, so merge them and sort.
sorting_tensor = box_to_level[:, 0] * 100000 + box_to_level[:, 1]
ix = tf.nn.top_k(sorting_tensor, k=tf.shape(box_to_level)[0]).indices[::-1]
ix = tf.gather(box_to_level[:, 2], ix)
pooled = tf.gather(pooled, ix)

# Re-add the batch dimension
shape = tf.concat([tf.shape(boxes)[:2], tf.shape(pooled)[1:]], axis=0)
pooled = tf.reshape(pooled, shape)
return pooled

def compute_output_shape(self, input_shape):
    return input_shape[0][:2] + self.pool_shape + (input_shape[2][-1], )

```

Klasyfikator Faster R-CNN:

```

def fpn_classifier_graph(rois, feature_maps, image_meta, pool_size, num_classes,
                       train_bn=True, fc_layers_size=1024):
    """Builds the computation graph of the feature pyramid network classifier
    and regressor heads.

    rois: [batch, num_rois, (y1, x1, y2, x2)] Proposal boxes in normalized
    coordinates.
    feature_maps: List of feature maps from different layers of the pyramid,
    [P2, P3, P4, P5]. Each has a different resolution.
    image_meta: [batch, (meta data)] Image details. See compose_image_meta()
    pool_size: The width of the square feature map generated from ROI Pooling.
    num_classes: number of classes, which determines the depth of the results
    train_bn: Boolean. Train or freeze Batch Norm layers
    fc_layers_size: Size of the 2 FC layers

    Returns:
    logits: [batch, num_rois, NUM_CLASSES] classifier logits (before softmax)
    probs: [batch, num_rois, NUM_CLASSES] classifier probabilities
    bbox_deltas: [batch, num_rois, NUM_CLASSES, (dy, dx, log(dh), log(dw))] Deltas
        to apply to
    proposal boxes
    """

    # ROI Pooling
    # Shape: [batch, num_rois, POOL_SIZE, POOL_SIZE, channels]
    x = PyramidROIAlign([pool_size, pool_size], name="roi_align_classifier")([rois,
        , image_meta] + feature_maps)
    # Two 1024 FC layers (implemented with Conv2D for consistency)
    x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (pool_size, pool_size),
        padding="valid"), name="mrcnn_class_conv1")(x)
    x = KL.TimeDistributed(BatchNorm(), name='mrcnn_class_bn1')(x, training=
        train_bn)
    x = KL.Activation('relu')(x)
    x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (1, 1)), name="mrcnn_class_conv2")(x)
    x = KL.TimeDistributed(BatchNorm(), name='mrcnn_class_bn2')(x, training=
        train_bn)

```

```

x = KL.Activation('relu')(x)

shared = KL.Lambda(lambda x: K.squeeze(K.squeeze(x, 3), 2), name="pool_squeeze") (x)

# Classifier head
mrcnn_class_logits = KL.TimeDistributed(KL.Dense(num_classes), name='mrcnn_class_logits')(shared)
mrcnn_probs = KL.TimeDistributed(KL.Activation("softmax"), name="mrcnn_class")(mrcnn_class_logits)

# BBox head
# [batch, num_rois, NUM_CLASSES * (dy, dx, log(dh), log(dw))]
x = KL.TimeDistributed(KL.Dense(num_classes * 4, activation='linear'), name='mrcnn_bbox_fc')(shared)
# Reshape to [batch, num_rois, NUM_CLASSES, (dy, dx, log(dh), log(dw))]
s = K.int_shape(x)
mrcnn_bbox = KL.Reshape((s[1], num_classes, 4), name="mrcnn_bbox")(x)

return mrcnn_class_logits, mrcnn_probs, mrcnn_bbox

```

Funkcje strat w gałęziach sieci RPN:

```

def rpn_class_loss_graph(rpn_match, rpn_class_logits):
    """RPN anchor classifier loss.

    rpn_match: [batch, anchors, 1]. Anchor match type. 1=positive,
    -1=negative, 0=neutral anchor.
    rpn_class_logits: [batch, anchors, 2]. RPN classifier logits for BG/FG.
    """

    # Squeeze last dim to simplify
    rpn_match = tf.squeeze(rpn_match, -1)
    # Get anchor classes. Convert the -1/+1 match to 0/1 values.
    anchor_class = K.cast(K.equal(rpn_match, 1), tf.int32)
    # Positive and Negative anchors contribute to the loss,
    # but neutral anchors (match value = 0) don't.
    indices = tf.where(K.not_equal(rpn_match, 0))
    # Pick rows that contribute to the loss and filter out the rest.
    rpn_class_logits = tf.gather_nd(rpn_class_logits, indices)
    anchor_class = tf.gather_nd(anchor_class, indices)
    # Cross entropy loss
    loss = K.sparse_categorical_crossentropy(target=anchor_class, output=
        rpn_class_logits, from_logits=True)
    loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
    return loss

```

```

def rpn_bbox_loss_graph(config, target_bbox, rpn_match, rpn_bbox):
    """Return the RPN bounding box loss graph.

    config: the model config object.
    target_bbox: [batch, max positive anchors, (dy, dx, log(dh), log(dw))].
    Uses 0 padding to fill in unused bbox deltas.
    rpn_match: [batch, anchors, 1]. Anchor match type. 1=positive,
    -1=negative, 0=neutral anchor.
    rpn_bbox: [batch, anchors, (dy, dx, log(dh), log(dw))]

```

```

"""
# Positive anchors contribute to the loss, but negative and
# neutral anchors (match value of 0 or -1) don't.
rpn_match = K.squeeze(rpn_match, -1)
indices = tf.where(K.equal(rpn_match, 1))

# Pick bbox deltas that contribute to the loss
rpn_bbox = tf.gather_nd(rpn_bbox, indices)

# Trim target bounding box deltas to the same length as rpn_bbox.
batch_counts = K.sum(K.cast(K.equal(rpn_match, 1), tf.int32), axis=1)
target_bbox = batch_pack_graph(target_bbox, batch_counts, config.
    IMAGES_PER_GPU)

loss = smooth_l1_loss(target_bbox, rpn_bbox)

loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
return loss

```

Funkcje strat w gałęziach klasyfikatora Faster R-CNN:

```

def mrcnn_class_loss_graph(target_class_ids, pred_class_logits, active_class_ids
    ):
    """
    Loss for the classifier head of Mask RCNN.

    target_class_ids: [batch, num_rois]. Integer class IDs. Uses zero
        padding to fill in the array.
    pred_class_logits: [batch, num_rois, num_classes]
    active_class_ids: [batch, num_classes]. Has a value of 1 for
        classes that are in the dataset of the image, and 0
        for classes that are not in the dataset.
    """

    # During model building, Keras calls this function with
    # target_class_ids of type float32. Unclear why. Cast it
    # to int to get around it.
    target_class_ids = tf.cast(target_class_ids, 'int64')

    # Find predictions of classes that are not in the dataset.
    pred_class_ids = tf.argmax(pred_class_logits, axis=2)
    pred_active = tf.gather(active_class_ids[0], pred_class_ids)

    # Loss
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
        labels = target_class_ids, logits=pred_class_logits)

    # Erase losses of predictions of classes that are not in the active
    # classes of the image.
    loss = loss * pred_active

    # Computer loss mean. Use only predictions that contribute
    # to the loss to get a correct mean.
    loss = tf.reduce_sum(loss) / tf.reduce_sum(pred_active)
    return loss

```

```

def mrcnn_bbox_loss_graph(target_bbox, target_class_ids, pred_bbox):

```

```

"""Loss for Mask R-CNN bounding box refinement.

target_bbox: [batch, num_rois, (dy, dx, log(dh), log(dw))]
target_class_ids: [batch, num_rois]. Integer class IDs.
pred_bbox: [batch, num_rois, num_classes, (dy, dx, log(dh), log(dw))]

"""

# Reshape to merge batch and roi dimensions for simplicity.
target_class_ids = K.reshape(target_class_ids, (-1,))
target_bbox = K.reshape(target_bbox, (-1, 4))
pred_bbox = K.reshape(pred_bbox, (-1, K.int_shape(pred_bbox)[2], 4))

# Only positive ROIs contribute to the loss. And only
# the right class_id of each ROI. Get their indices.
positive_roi_ix = tf.where(target_class_ids > 0)[:, 0]
positive_roi_class_ids = tf.cast(tf.gather(target_class_ids, positive_roi_ix),
                                 tf.int64)
indices = tf.stack([positive_roi_ix, positive_roi_class_ids], axis=1)

# Gather the deltas (predicted and true) that contribute to loss
target_bbox = tf.gather(target_bbox, positive_roi_ix)
pred_bbox = tf.gather_nd(pred_bbox, indices)

# Smooth-L1 Loss
loss = K.switch(tf.size(target_bbox) > 0,
                smooth_l1_loss(y_true=target_bbox, y_pred=pred_bbox), tf.constant(0.0))
loss = K.mean(loss)
return loss

```

Zmiana rozdzielczości oryginalnego obrazu oraz przeskalowanie anotowanych ramek:

```

image, window, scale, padding, crop, bbox = utils.resize_image(image, min_dim=
    config.IMAGE_MIN_DIM, min_scale=config.IMAGE_MIN_SCALE, max_dim=config.
    IMAGE_MAX_DIM, mode=config.IMAGE_RESIZE_MODE, bbox=bbox)

```

```

def resize_image(image, min_dim=None, max_dim=None, min_scale=None, mode="square",
                 bbox=[]):
    """Resizes an image keeping the aspect ratio unchanged.

    min_dim: if provided, resizes the image such that it's smaller
    dimension == min_dim
    max_dim: if provided, ensures that the image longest side doesn't
    exceed this value.
    min_scale: if provided, ensure that the image is scaled up by at least
    this percent even if min_dim doesn't require it.
    mode: Resizing mode.
    square: Resize and pad with zeros to get a square image
    of size [max_dim, max_dim].

    Returns:
    image: the resized image
    window: (y1, x1, y2, x2). If max_dim is provided, padding might
    be inserted in the returned image. If so, this window is the
    coordinates of the image part of the full image (excluding
    the padding). The x2, y2 pixels are not included.
    scale: The scale factor used to resize the image

```

```

padding: Padding added to the image [(top, bottom), (left, right), (0, 0)]
"""

original_shape = image.shape
# Keep track of image dtype and return results in the same dtype
image_dtype = image.dtype
# Default window (y1, x1, y2, x2) and default scale == 1.
h, w = image.shape[:2]
window = (0, 0, h, w)
scale = 1
padding = [(0, 0), (0, 0), (0, 0)]
crop = None

if min_dim:
    # Scale up but not down
    scale = max(1, min_dim / min(h, w))
if min_scale and scale < min_scale:
    scale = min_scale

if max_dim and mode == "square":
    image_max = max(h, w)
    if round(image_max * scale) > max_dim:
        scale = max_dim / image_max

# Resize image using bilinear interpolation
if scale != 1:
    image = resize(image, (round(h * scale), round(w * scale)), preserve_range=True)

if mode == "square":
    # Get new height and width
    h, w = image.shape[:2]
    top_pad = (max_dim - h) // 2
    bottom_pad = max_dim - h - top_pad
    left_pad = (max_dim - w) // 2
    right_pad = max_dim - w - left_pad
    padding = [(top_pad, bottom_pad), (left_pad, right_pad), (0, 0)]
    image = np.pad(image, padding, mode='constant', constant_values=0)
    window = (top_pad, left_pad, h + top_pad, w + left_pad)

if np.any(bbox):
    scaled_bbox = np.copy(bbox).astype(np.float32)
    for i in range(len(scaled_bbox)):
        for j in range(4):
            if j == 0 or j == 2: #y
                scaled_bbox[i, j] = bbox[i, j] * ((window[2] - window[0]) / original_shape[0]) + padding[0][0]
            elif j == 1 or j == 3: #x
                scaled_bbox[i, j] = bbox[i, j] * ((window[3] - window[1]) / original_shape[1]) + padding[1][0]
    scaled_bbox = np.rint(scaled_bbox).astype(np.int32)
    return image.astype(image_dtype), window, scale, padding, crop,
           scaled_bbox
else:
    return image.astype(image_dtype), window, scale, padding, crop

```

Augmentacje oraz związane z nimi zmiany współrzędnych anotowanych ramek:

```
aug_type = 'fliplr'
if augment:
    if aug_type == 'fliplr':
        seq = iaa.Sequential([iaa.Fliplr(p=0.4)]) #construct aug pipeline
    elif aug_type == 'rotate':
        seq = iaa.Sequential([iaa.Rotate(rotate=iap.Uniform((-30,0),(0,30)))])
    elif aug_type == 'translate':
        seq = iaa.Sequential([iaa.TranslateX(px=(-100,100)), iaa.TranslateY(px=(-100,100))])
    elif aug_type == 'blur':
        seq = iaa.Sequential([iaa.GaussianBlur(sigma=iap.Uniform((0.0,1.5),(1.5,3.0)))])
    elif aug_type == 'brighten':
        seq = iaa.Sequential([iaa.MultiplyBrightness(mul=iap.Uniform((0.5,1),(1,1.5)))])
    elif aug_type == 'combine':
        seq = iaa.Sometimes(p=1, then_list=iaa.SomeOf(n=(1,4), children=[iaa.Fliplr(p=1), iaa.MultiplyBrightness(mul=iap.Uniform((0.5,1),(1,1.5))), iaa.GaussianBlur(sigma=iap.Uniform((0.0,1.5),(1.5,3.0))), iaa.Rotate(rotate=iap.Uniform((-30,0),(0,30)))]), random_order=False)
    elif aug_type == 'combine2':
        seq = iaa.Sometimes(p=1, then_list=iaa.SomeOf(n=(1,4), children=[iaa.Fliplr(p=1), iaa.MultiplyBrightness(mul=iap.Uniform(0.5,1.5)), iaa.GaussianBlur(sigma=iap.Uniform(0.0, 3.0)), iaa.Rotate(rotate=iap.Uniform(-30, 30))]), random_order=False)
bbox_formatted = np.copy(bbox)
for i in range(len(bbox)): #format bbox to normal format x1,x2,y1,y2 from y1, x1,y2,x2
    bbox_formatted[i,0] = bbox[i,1]
    bbox_formatted[i,1] = bbox[i,0]
    bbox_formatted[i,2] = bbox[i,3]
    bbox_formatted[i,3] = bbox[i,2]

bbs_to_aug = iaables.bbs.BoundingBoxesOnImage.from_xyxy_array(bbox_formatted, image.shape)
image_aug, bbox_aug = seq(image=image, bounding_boxes=bbs_to_aug)
bbox_after_aug = bbox_aug.to_xyxy_array(dtype=np.float32)
```

7. Dodatek B

Usunięte pliki z bazy danych Wider Face z powodu niezawierania żadnych oznaczeń twarzy:

- ze zbioru treningowego:

- 1) 0--Parade/0_Parade_Parade_0_452.jpg
- 2) 2--Demonstration/2_Demonstration_Political_Rally_2_444.jpg
- 3) 39--Ice_Skating/39_Ice_Skating_iceskiing_39_380.jpg
- 4) 46--Jockey/46_Jockey_Jockey_46_576.jpg

- ze zbioru walidacyjnego:

- 1) 0--Parade/0_Parade_Parade_0_275.jpg
- 2) 37--Soccer/37_Soccer_soccer_ball_37_281.jpg
- 3) 50--Celebration_Or_Party/50_Celebration_Or_Party_houseparty_50_715.jpg
- 4) 7--Cheering/7_Cheering_Cheering_7_426.jpg

Wykaz plików z wadliwymi anotacjami oraz ich rodzaj:

- ze zbioru treningowego:

- 1) 54--Rescue/54_Rescue_rescuepeople_54_29.jpg - przy rozdzielczości 1024×1024 ,
 $x_{min} = 1050$
- 2) 7--Cheering/7_Cheering_Cheering_7_17.jpg - $x_{max} = -11$

- ze zbioru walidacyjnego:

- 1) 39--Ice_Skating/39_Ice_Skating_iceskiing_39_583.jpg - przy rozdzielczości
 1024×577 , $x_{min} = 1026$
- 2) 7--Cheering/7_Cheering_Cheering_7_171.jpg - $y_{min} = y_{max} = 0$

8. Bibliografia

- [1] M. Yang, D. J. Kriegman, and N. Ahuja, "Detecting faces in images: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 1, pp. 34–58, 2002. [Online]. Available: <https://doi.org/10.1109/34.982883>
- [2] Face recognition and people counting boost customer service and profitability | Axis Communications. Accessed 2020-11-11. [Online]. Available: <https://www.axis.com/customer-story/4628>
- [3] Surveillance and Security Drone | AirborneDrones. Accessed 2020-11-12. [Online]. Available: <https://www.airbornedrones.co/surveillance-and-security/>
- [4] Secure universal access control | Biometric login. Accessed 2020-12-22. [Online]. Available: <https://www.bioid.com/secure-universal-access-control/>
- [5] (2020, Mar.) 5 best tablets with face recognition feature. Accessed 2020-12-22. [Online]. Available: <https://buybulktablets.com/5-best-tablets-with-face-recognition-feature/>
- [6] Facial Recognition Advertising Targets Customers - businessnewsdaily.com. Accessed 2020-12-22. [Online]. Available: <https://www.businessnewsdaily.com/15213-walgreens-facial-recognition.html>
- [7] M. Elgan and M. Elgan. (2019, Jun.) What happens when cars get emotional? Accessed 2020-12-22. [Online]. Available: <https://www.fastcompany.com/90368804/emotion-sensing-cars-promise-to-make-our-roads-much-safer>
- [8] Sophia | HansonRobotics. Accessed 2020-11-14. [Online]. Available: <https://www.hansonrobotics.com/sophia/>
- [9] AI robot confirmed as speaker | Malta Blockchain Summit. Accessed 2020-11-14. [Online]. Available: <https://www.maltablockchainsummit.com/news/ai-robot-confirmed-as-speaker-at-the-malta-blockchain-summit/>
- [10] Aibo | Aibo. Accessed 2020-11-14. [Online]. Available: <https://us.aibo.com/>
- [11] Sony gives a sneak peek of aibo, its new robotic dog. Accessed 2020-12-17. [Online]. Available: <https://asia.nikkei.com/Life-Arts/Life/Sony-gives-a-sneak-peek-of-aibo-its-new-robotic-dog>
- [12] B. Marr. Chinese Social Credit Score: Utopian Big Data Bliss Or Black Mirror On Steroids? Section: Innovation, Accessed 2020-11-14. [Online]. Available: <https://www.forbes.com/sites/bernardmarr/2019/01/21/chinese-social-credit-score-utopian-big-data-bliss-or-black-mirror-on-steroids/>
- [13] S. Yang, P. Luo, C. C. Loy, and X. Tang, "WIDER FACE: A Face Detection Benchmark," *arXiv:1511.06523 [cs]*, Nov. 2015, arXiv: 1511.06523. [Online]. Available: <http://arxiv.org/abs/1511.06523>
- [14] Arirang News. (2020, Oct.) Kim Jong-un's emotional tactics to solidify internal unity among people: Experts. [Online]. Available: <https://www.youtube.com/watch?v=8Cpeth4EWOs>

- [15] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *arXiv:1311.2524 [cs]*, Oct. 2014, arXiv: 1311.2524. [Online]. Available: <http://arxiv.org/abs/1311.2524>
- [16] Cogneethi, “C 6.5 | RCNN General Discussion - 2 Stage Network - 3 Stage Training - Results and some Questions,” Aug. 2019. [Online]. Available: <https://www.youtube.com/watch?v=7VkJClP9vJg>
- [17] R. Girshick, “Fast R-CNN,” *arXiv:1504.08083 [cs]*, Sep. 2015, arXiv: 1504.08083. [Online]. Available: <http://arxiv.org/abs/1504.08083>
- [18] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *arXiv:1506.01497 [cs]*, Jan. 2016, arXiv: 1506.01497. [Online]. Available: <http://arxiv.org/abs/1506.01497>
- [19] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN,” *arXiv:1703.06870 [cs]*, Jan. 2018, arXiv: 1703.06870. [Online]. Available: <http://arxiv.org/abs/1703.06870>
- [20] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” *arXiv:1506.02640 [cs]*, May 2016, arXiv: 1506.02640. [Online]. Available: <http://arxiv.org/abs/1506.02640>
- [21] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” *arXiv:1612.08242 [cs]*, Dec. 2016, arXiv: 1612.08242. [Online]. Available: <http://arxiv.org/abs/1612.08242>
- [22] ———, “YOLOv3: An Incremental Improvement,” *arXiv:1804.02767 [cs]*, Apr. 2018, arXiv: 1804.02767. [Online]. Available: <http://arxiv.org/abs/1804.02767>
- [23] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, “YOLOv4: Optimal Speed and Accuracy of Object Detection,” *arXiv:2004.10934 [cs, eess]*, Apr. 2020, arXiv: 2004.10934. [Online]. Available: <http://arxiv.org/abs/2004.10934>
- [24] G. Jocher, A. Stoken, J. Borovec, NanoCode012, ChristopherSTAN, L. Changyu, Laughing, tkianai, yxNONG, A. Hogan, lorenzomammana, AlexWang1900, A. Chaurasia, L. Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Durgesh, F. Ingham, Frederik, Guilhen, A. Colmagro, H. Ye, Jacobsolawetz, J. Poznanski, J. Fang, J. Kim, K. Doan, and L. Yu, “ultralytics/yolov5: v4.0 - nn.SiLU() activations, Weights & Biases logging, PyTorch Hub integration,” Jan. 2021. [Online]. Available: <https://doi.org/10.5281/zenodo.4418161>
- [25] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single Shot MultiBox Detector,” *arXiv:1512.02325 [cs]*, vol. 9905, pp. 21–37, 2016, arXiv: 1512.02325. [Online]. Available: <http://arxiv.org/abs/1512.02325>
- [26] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal Loss for Dense Object Detection,” *arXiv:1708.02002 [cs]*, Feb. 2018, arXiv: 1708.02002. [Online]. Available: <http://arxiv.org/abs/1708.02002>
- [27] A. Baldeschi, R. Margutti, and A. Miller, “Deep Learning: a new definition of artificial neuron with double weight,” *arXiv:1905.04545 [cs, stat]*, May 2019, arXiv: 1905.04545. [Online]. Available: <http://arxiv.org/abs/1905.04545>

- [28] 7 Types of Activation Functions in Neural Networks: How to Choose? Accessed 2020-11-25. [Online]. Available: <https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/>
- [29] A. Ye. (2020, Sep.) If Rectified Linear Units Are Linear, How Do They Add Nonlinearity? Accessed 2020-11-26. [Online]. Available: <https://towardsdatascience.com/if-rectified-linear-units-are-linear-how-do-they-add-nonlinearity-40247d3e4792>
- [30] K. E. Koehc. (2020, Oct.) Softmax Activation Function — How It Actually Works. Accessed 2020-11-26. [Online]. Available: <https://towardsdatascience.com/softmax-activation-function-how-it-actually-works-d292d335bd78>
- [31] S. S. Haykin and S. S. Haykin, *Neural networks and learning machines*, 3rd ed. New York: Prentice Hall, 2009, ch. 1, sec. 1, p. 2, oCLC: ocn237325326.
- [32] L. Camuñas-Mesa, B. Linares-Barranco, and T. Serrano-Gotarredona, “Neuromorphic spiking neural networks and their memristor-cmos hardware implementations,” *Materials*, vol. 12, p. 2745, 08 2019.
- [33] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Pérez, “Deep Reinforcement Learning for Autonomous Driving: A Survey,” *arXiv:2002.00444 [cs]*, Feb. 2020, arXiv: 2002.00444. [Online]. Available: <http://arxiv.org/abs/2002.00444>
- [34] C. Colas, O. Sigaud, and P.-Y. Oudeyer, “GEP-PG: Decoupling Exploration and Exploitation in Deep Reinforcement Learning Algorithms,” *arXiv:1802.05054 [cs]*, Sep. 2018, arXiv: 1802.05054. [Online]. Available: <http://arxiv.org/abs/1802.05054>
- [35] P. Dayan and Y. Niv, “Reinforcement learning: The Good, The Bad and The Ugly,” *Current Opinion in Neurobiology*, vol. 18, no. 2, pp. 185–196, Apr. 2008. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0959438808000767>
- [36] R. Meka, “CS289ML: Notes on convergence of gradient descent,” p. 7.
- [37] L. Bottou, “On-line Learning and Stochastic Approximations,” in *On-Line Learning in Neural Networks*, 1st ed., D. Saad, Ed. Cambridge University Press, Jan. 1999, pp. 17–26. [Online]. Available: https://www.cambridge.org/core/product/identifier/CBO9780511569920A009/type/book_part
- [38] Z. Little. (2020, Jul.) Gradient Descent: Stochastic vs. Mini-batch vs. Batch vs. AdaGrad vs. RMSProp vs. Adam. Accessed 2020-11-29. [Online]. Available: <https://xzz201920.medium.com/gradient-descent-stochastic-vs-mini-batch-vs-batch-vs-adagrad-vs-rmsprop-vs-adam-3aa652318b0d>
- [39] CS231n Convolutional Neural Networks for Visual Recognition. Accessed 2020-11-29. [Online]. Available: <https://cs231n.github.io/neural-networks-3/>
- [40] L. Flagel, Y. Brandvain, and D. R. Schrider, “The Unreasonable Effectiveness of Convolutional Neural Networks in Population Genetic Inference,” *Molecular Biology and Evolution*, vol. 36, no. 2, pp. 220–238, 12 2018. [Online]. Available: <https://doi.org/10.1093/molbev/msy224>

- [41] S. Mehtab and J. Sen, "Stock Price Prediction Using Convolutional Neural Networks on a Multivariate Timeseries," p. 7.
- [42] J. Kim, O. Sangjun, Y. Kim, and M. Lee, "Convolutional Neural Network with Biologically Inspired Retinal Structure," *Procedia Computer Science*, vol. 88, pp. 145–154, 2016. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S187705091631674X>
- [43] Prabhu. (2019, Nov.) Understanding of Convolutional Neural Network (CNN) — Deep Learning. Accessed 2020-11-30. [Online]. Available: <https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148>
- [44] N. Buduma and N. Locascio, "Fundamentals of deep learning," 2017.
- [45] Y. Wang, Y. Li, Y. Song, and X. Rong, "The influence of the activation function in a convolution neural network model of facial expression recognition," *Applied Sciences*, vol. 10, p. 1897, 03 2020.
- [46] E. Kauderer-Abrams, "Quantifying Translation-Invariance in Convolutional Neural Networks," *arXiv:1801.01450 [cs]*, Dec. 2017, arXiv: 1801.01450. [Online]. Available: <http://arxiv.org/abs/1801.01450>
- [47] D. Yu, H. Wang, P. Chen, and Z. Wei, "Mixed pooling for convolutional neural networks," 10 2014, pp. 364–375.
- [48] S. Bhandarkar, R. Prasad, V. Agarwal, R. Hebbar, D. Uma, Y. N. Reddy, Y. Raghuramulu, and K. Raj, "Deep learning and statistical models for detection of white stem borer disease in arabica coffee," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 423, pp. 443–451, 2019.
- [49] H. Le and A. Borji, "What are the Receptive, Effective Receptive, and Projective Fields of Neurons in Convolutional Neural Networks?" *arXiv:1705.07049 [cs]*, Apr. 2018, arXiv: 1705.07049. [Online]. Available: <http://arxiv.org/abs/1705.07049>
- [50] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs]*, Dec. 2015, arXiv: 1512.03385. [Online]. Available: <http://arxiv.org/abs/1512.03385>
- [51] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature Pyramid Networks for Object Detection," *arXiv:1612.03144 [cs]*, Apr. 2017, arXiv: 1612.03144. [Online]. Available: <http://arxiv.org/abs/1612.03144>
- [52] P. Dwivedi. (2019, Mar.) Understanding and Coding a ResNet in Keras. Accessed 2020-12-07. [Online]. Available: <https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33>
- [53] Bbox standard deviation usage in Mask R-CNN · Issue #270 · matterport/Mask_RCNN. Accessed 2020-01-11. [Online]. Available: https://github.com/matterport/Mask_RCNN/issues/270#issuecomment-367602954
- [54] (2020, May) Bilinear interpolation. Page Version ID: 957374633, Accessed 2020-12-19. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bilinear_interpolation&oldid=957374633

- [55] J. R. Winkler, “Numerical recipes in C: The art of scientific computing, second edition,” *Endeavour*, vol. 17, no. 4, pp. 123–128, Jan. 1993. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/016093279390069F>
- [56] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The Pascal Visual Object Classes (VOC) Challenge,” *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, Jun. 2010. [Online]. Available: <http://link.springer.com/10.1007/s11263-009-0275-4>
- [57] N. Zeng. (2018, Dec.) An Introduction to Evaluation Metrics for Object Detection. Accessed 2020-12-20. [Online]. Available: <https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/>
- [58] “ImageNet.” [Online]. Available: <http://www.image-net.org/>
- [59] imgaug — imgaug 0.4.0 documentation. Accessed 2020-12-28. [Online]. Available: <https://imgaug.readthedocs.io/en/latest/>
- [60] OpenCV: Cascade Classifier. Accessed 2020-12-28. [Online]. Available: https://docs.opencv.org/master/db/d28/tutorial_cascade_classifier.html
- [61] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1. Kauai, HI, USA: IEEE Comput. Soc, 2001, pp. I–511–I–518. [Online]. Available: <http://ieeexplore.ieee.org/document/990517/>
- [62] W. Abdulla, “Mask r-cnn for object detection and instance segmentation on keras and tensorflow,” https://github.com/matterport/Mask_RCNN, 2017.

Spis rysunków

1	Robot Sophia, źródło: [8]	8
2	Robotyczny pies Aibo, źródło: [11]	8
3	Otrzymany efekt końcowy – przykład, źródłowy reportaż: [14]	9
4	Topologia sieci R-CNN, źródło: [16]	10
5	Przykładowa topologia sieci YOLOv1 ($S = 7, C = 20, B = 2$), źródło: [20]	11
6	Najczęściej spotykany typ neuronu	13
7	Kształt sigmoidalnych funkcji aktywacji o różnych wartościach przesunięcia b	13
8	Wykres binarnej funkcji skoku jednostkowego	13
9	Wykres funkcji liniowej	14
10	Wykres funkcji logistycznej	14
11	Wykres funkcji tangensa hiperbolicznego	14
12	Wykres pochodnej funkcji tangensa hiperbolicznego	14
13	Wykres funkcji ReLU	15
14	Klasyfikacja danych na płaszczyźnie przez różne funkcje aktywacji, źródło: [29]	15
15	Ilustracja działania warstwy Softmax, źródło: [30]	16
16	Schematy sieci jednokierunkowych: jednowarstwowej (a) i wielowarstwowej (b), źródło: [32]	16
17	Porównanie idei model-free RL do model-based RL, źródło: [35]	18
18	Stosowane w obliczeniach oznaczenia naniesione na przykładową wielowarstwową sieć FFNN, źródło: [35]	19
19	Porównanie działania metod optymalizacji, źródło: [38]	20
20	Porównanie efektywności stałego współczynnika uczenia, źródło: [39]	20
21	Typowa architektura sieci konwolucyjnej, źródło: [43]	21
22	Wizualizacja konwolucji 2D, źródło: [44]	22
23	Mapy cech powstałe po zastosowaniu różnych warstw <i>pooling</i> , źródło: [48]	23
24	Działanie <i>max</i> i <i>avg pooling</i> w zależności od tła, źródło: [47]	23
25	Ilustracja pól oddziaływania neuronu, źródło: [49]	23
26	Topologia sieci Faster R-CNN, źródło: [18]	24
27	Wykaz odmian ResNet zastosowanych przez twórców do treningu na zbiorze ImageNet, źródło: [50]	25
28	Zjawisko degradacji na przykładzie sieci CIFAR-10, źródło: [50]	25
29	Porównanie topologii zwykłego bloku rezydualnego i <i>bottleneck residual block</i> , źródło: [50]	26
30	Porównanie topologii zwykłej sieci konwolucyjnej z siecią rezydualną, źródło: [50]	26
31	Blok budujący sieci FPN: sumacja tensoru powstałego po operacji <i>Upsampling</i> z czynnikiem 2 oraz tensoru otrzymanego na wyjściu z połączenia lateralnego, źródło: [51]	27
32	Podział sieci ResNet-101 na 5 etapów, źródło: [52]	27
33	Połączenia warstw FPN z modułem RPN na grafie Tensorboard	28
34	<i>Anchors</i> w różnych skalach naniesione na obraz. Wyżej ilość <i>anchors</i> na każdym z poziomów	28
35	Początek i rozgałęzienie sieci RPN, źródło: [18]	29

36	Czerwone kropki odpowiadają najbliższym punktom mapy cech, P to punkt w centrum jednego z oczek, dla którego szukana jest wartość, źródło: [54]	31
37	Porównanie metod <i>RoI Pooling</i> (ResNet-50-C4), źródło: [19]	32
38	Topologia klasyfikatora Fast R-CNN, źródło: [17]	32
39	92 twarze małych rozmiarów 18_Concerts_Concerts_18_203.jpg	36
40	15 twarzy średnich rozmiarów 26_Soldier_Drilling_Soldiers_Drilling_26_20.jpg	37
41	2 twarze dużych rozmiarów 38_Tennis_Tennis_38_1029.jpg	37
42	Przykład krzywej P-R po interpolacji precyzji, źródło: [57]	40
43	Całkowita funkcja strat L_{total} dla zbioru treningowego	44
44	Całkowita funkcja strat L_{total} dla zbioru walidacyjnego	44
45	Funkcja strat L_{bbox1} regresji Rols dla zbioru treningowego	45
46	Funkcja strat L_{bbox1} regresji Rols dla zbioru walidacyjnego	45
47	Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru treningowego	45
48	Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru walidacyjnego	45
49	Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru treningowego	45
50	Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego	45
51	Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru treningowego	46
52	Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego	46
53	14/14 poprawnie przewidzianych twarzy i 1 fałszywie pozytywna (na słupku). Po mimo licznych okluzji i masek policyjnych, sieć poradziła sobie niemal doskonale. 3_Riot_Riot_3_522.jpg	48
54	49/59 poprawnie przewidzianych twarzy i 5 fałszywie pozytywnych. Małe powierzchnie i okluzje wielu twarzy pogarszają wyniki. 18_Concerts_Concerts_18_602.jpg	48
55	Całkowita funkcja strat L_{total} dla zbioru treningowego	50
56	Całkowita funkcja strat L_{total} dla zbioru walidacyjnego	50
57	Funkcja strat L_{bbox1} regresji Rols dla zbioru treningowego	50
58	Funkcja strat L_{bbox1} regresji Rols dla zbioru walidacyjnego	50
59	Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru treningowego	50
60	Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru walidacyjnego	50
61	Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru treningowego	51
62	Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego	51
63	Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru treningowego	51
64	Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru walidacyjnego	51
65	Całkowita funkcja strat L_{total} dla zbioru treningowego	53
66	Całkowita funkcja strat L_{total} dla zbioru walidacyjnego	53
67	Funkcja strat L_{bbox1} regresji Rols dla zbioru treningowego	54

68	Funkcja strat L_{bbox1} regresji Rols dla zbioru walidacyjnego	54
69	Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru treningowego	54
70	Funkcja strat L_{cls1} klasyfikacji Rols dla zbioru walidacyjnego	54
71	Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru trenin- gowego	54
72	Funkcja strat L_{bbox2} regresji ramek klasyfikatora Faster R-CNN dla zbioru wali- dacyjnego	54
73	Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru tre- ningowego	55
74	Funkcja strat L_{cls2} klasyfikacji ramek klasyfikatora Faster R-CNN dla zbioru wa- lidacyjnego	55
75	Graficzna reprezentacja cech Haara, źródło: [60]	57