

**Politechnika Warszawska**

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



Instytut Radioelektroniki i Technik Multimedialnych

# Praca dyplomowa

na studiach: Głębokie Sieci Neuronowe – Zastosowania w Mediach Cyfrowych

Zastosowanie głębokich sieci neuronowych do detekcji  
budynków na zdjęciach lotniczych

Mateusz Gomulski

opiekun pracy dyplomowej  
prof. dr hab. inż. Władysław Skarbek

WARSZAWA 2020

# **Zastosowanie głębokich sieci neuronowych do detekcji budynków na zdjęciach lotniczych**

**Streszczenie.** Niniejsza praca dyplomowa podejmuje problematykę detekcji budynków na zdjęciach lotniczych przy wykorzystaniu głębokich sieci neuronowych. Zastosowana konwolucyjna sieć neuronowa o nazwie *GML-Net* posiada architekturę typu *U-Net* z enkoderem wywodzącym się z rodziny *ResNet* oraz z blokami *BottleNeck* zapewniającymi odczyt i agregację map cech z przekroju różnych skal. Wykorzystane w bieżącej pracy punktowo i wgłębiennie separowalne konwolucje pozwalają sieci neuronowej efektywnie nauczyć się korelacji kanałów przestrzennych, uniknąć nadmiernego dopasowania oraz uzyskać większą efektywność obliczeniową. Jako funkcję straty zapewniającą efektywne uczenie się sieci, zdecydowano się użyć ważonej sumy *Binary Cross-Entropy Loss*, *Dice Loss* oraz *Lovász hinge loss*. Niniejsza praca dyplomowa pokazuje, iż tak zbudowana sieć neuronowa pozwala uzyskać zadowalające wyniki przy zdjęciach lotniczych zawierających zaledwie trzy kanały przestrzenne (RGB). Wyniki te są tylko nieznacznie gorsze od wyników uzyskanych przez najlepsze modele wytrenowane na zbiorze danych *Inria Aerial Image Labeling Dataset*.

**Słowa kluczowe:** Głębokie sieci neuronowe, Zdjęcia lotnicze, Rozpoznawanie budynków, Segmentacja semantyczna, *ResNet*, *U-Net*, *BottleNeck*

## **Buildings detection in aerial images using deep neural networks**

**Abstract.** The diploma thesis deals with the problem of detecting buildings in aerial images using deep neural networks. The *GML-Net* convolutional neural network that is used in this work has a *U-Net* architecture with an encoder derived from the *ResNet* family and *BottleNeck* blocks that provide reading and aggregation of feature maps from a cross-section of various scales. Pointwise and depthwise convolutions used in this diploma thesis allow neural network to effectively learn spatial channel correlation, avoid overfitting and obtain greater computational efficiency. Effective network learning is ensured by loss function defined as a weighted sum of *Binary Cross-Entropy Loss*, *Dice Loss* and *Lovász hinge Loss*. This diploma thesis shows that a neural network constructed in such a way allows to obtain satisfactory results for aerial images with only three spectral channels (RGB). These results are only slightly worse than the results achieved by the best models trained on the *Inria Aerial Image Labeling Dataset*.

**Keywords:** Deep neural networks, Aerial images, Buildings detection, Semantic segmentation, *ResNet*, *U-Net*, *BottleNeck*



# **Spis treści**

<b>1. Wstęp . . . . .</b>	6
<b>2. Podstawy teoretyczne głębokiego uczenia . . . . .</b>	8
2.1. Głębokie uczenie - podstawowe definicje . . . . .	8
2.2. Ewolucja architektur głębokiego uczenia . . . . .	13
2.2.1. Cybernetyka (1943 - 1969) . . . . .	13
2.2.2. Konekcjonizm (1980 - 1998) . . . . .	15
2.2.3. Głębokie uczenie (2006 - 2020) . . . . .	20
2.3. Proces uczenia się głębokich sieci neuronowych . . . . .	35
<b>3. Przegląd literatury z zakresu detekcji budynków na zdjęciach lotniczych . . . . .</b>	41
<b>4. Konstrukcja sieci <i>GML-Net</i> . . . . .</b>	47
4.1. Najważniejsze hiperparametry sieci . . . . .	47
4.2. Dane uczące i ich transformacje . . . . .	49
4.3. Architektura sieci . . . . .	60
4.4. Trenowanie sieci . . . . .	78
<b>5. Analiza wyników . . . . .</b>	90
5.1. Generowanie finalnych predykcji . . . . .	90
5.2. Pomiar efektywności sieci <i>GML-Net</i> . . . . .	98
5.3. Uzyskane wyniki na tle literatury badawczej . . . . .	103
<b>6. Zakończenie . . . . .</b>	105
<b>Bibliografia . . . . .</b>	106
<b>Wykaz symboli i skrótów . . . . .</b>	109
<b>Spis rysunków . . . . .</b>	112
<b>Spis tabel . . . . .</b>	112

# 1. Wstęp

Głębokie uczenie zrewolucjonizowało dziedzinę przetwarzania sygnałów cyfrowych w niespotykanej dotąd skali. Przed erą głębokich sieci neuronowych komputery nie były w stanie zbliżyć się do skuteczności z jaką przeciętny człowiek radzi sobie z takimi zagadnieniami z dziedziny cyfrowego przetwarzania sygnałów jak na przykład: rozpoznawanie / segmentacja obiektów na zdjęciach cyfrowych, rozpoznawanie pozy / emocji na podstawie analizy filmu wideo czy detekcja poleceń / słów kluczowych na podstawie nagrania audio. Nadejście ery głębokich sieci neuronowych, której początek można datować na 2006 rok (a intensywny rozwój na czas po 2012 roku), spowodowało nie tylko dorównanie przez komputery, w wielu zadaniach, średniej ludzkiej skuteczności, ale często istotne przewyższenie jej (ang. *superhuman performance*). Dobrym przykładem ponadludzkiej skuteczności głębokich sieci neuronowych jest konkurs *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)*, który polegał na automatycznym generowaniu podpisów do obrazów znajdujących się w zbiorze *ImageNet*. Jeszcze w 2011 roku zwycięskie rozwiązanie w tym konkursie charakteryzowało się stopą błędów top-5<sup>1</sup> na poziomie 26%. Przełomowe rozwiązanie zaproponowane przez Krizhevsky A., Sutskever I. i Hinton G. z 2012 roku [18], wykorzystujące głębokie uczenie się, wygrało ten konkurs osiągając stopę błędów top-5 na poziomie 15%. W 2017 roku organizatorzy konkursu *ILSVRC* zapowiedzieli zakończenie jego organizacji (przy najlepszym wyniku top-5 równym 2,251% uzyskanym przez sieć *SENet*), gdyż rezultaty osiągane na zbiorze testowym przez większość zgłoszonych do konkursu modeli zaczęły istotnie przekraczać wyniki przeciętnego człowieka (około 5%) - zapowiedzieli oni jednocześnie organizację dużo bardziej wymagającego konkursu wiążącego się z klasyfikacją trójwymiarowych obiektów.

Generowanie automatycznych podpisów do zdjęć jest jednak zadaniem dużo mniej skomplikowanym niż detekcja obiektów na zdjęciach lotniczych czy też satelitarnych. Jest tak, gdyż zdjęcia lotnicze / satelitarne charakteryzują się o wiele większą złożonością pod względem skali, jakości, rozdzielczości czy też zaszumienia niż zdjęcia pochodzące z takich zbiorów jak *ImageNet* a głównym zadaniem sieci nie jest wybranie jednego z tysiąca podpisów, ale wskazanie piksel po pikselu, do jakiego obiektu dany piksel z konkretnego zdjęcia należy. Stąd też rezultaty osiągane przez głębokie sieci neuronowe przy tego typu zadaniach są w dalszym ciągu dużo gorsze niż rezultaty osiągane przez człowieka. Biorąc pod uwagę stale wzrastającą liczbę zdjęć oraz nagrań wideo rejestrowanych codziennie przez satelity / samoloty / drony, śmiało można stwierdzić, iż istnieje duże zapotrzebowanie / pole do rozwoju i ulepszeń głębokich metod stosowanych do analizy tego typu danych, tak żeby w przyszłości głębokie sieci neuronowe mogły z powodzeniem zastąpić w tym

<sup>1</sup> Stopa błędów top-5 to miara stosowana przy ocenie modeli klasifikacji w ramach uczenia maszynowego, zakłada ona, że model zakwalifikował dany obraz poprawnie gdy wśród pięciu najbardziej prawdopodobnych kategorii wskazanych przez model znajduje się poprawna kategoria

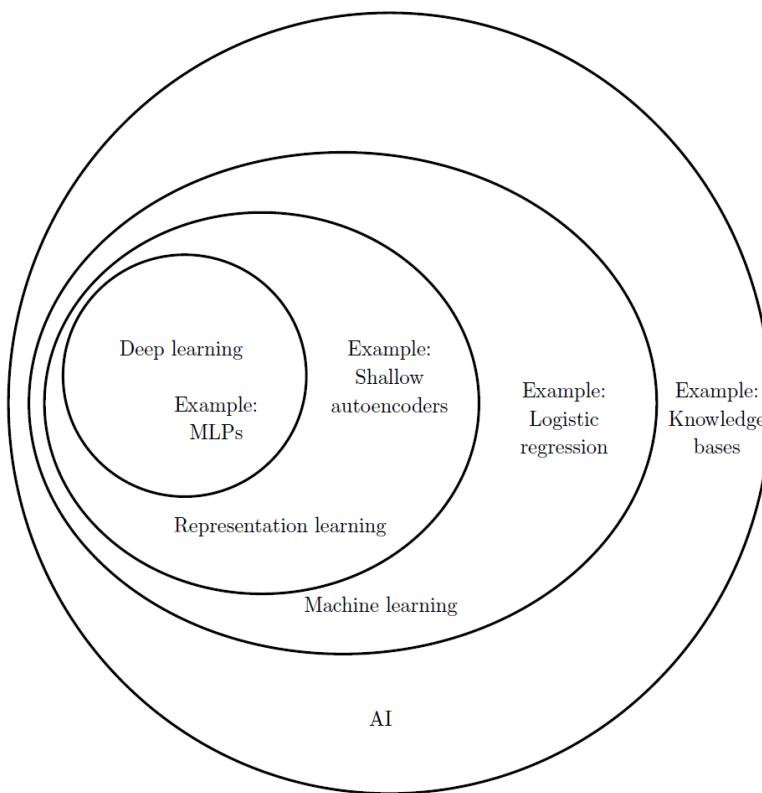
obszarze człowieka. W szczególności, istotną kwestią przy analizie zdjęć lotniczych i satelitarnych, jest detekcja budynków, gdyż świadomość dokładnego położenia zabudowań na mapach miast jest niezbędna w takich dziedzinach życia społeczno-ekonomicznego jak: urbanistyka, socjologia, bezpieczeństwo publiczne, ubezpieczenia majątkowe czy też reagowanie kryzysowe w obliczu klęsk żywiołowych.

Niniejsza praca podejmuje problematykę rozpoznawania budynków na zdjęciach lotniczych przy pomocy głębokich sieci neuronowych, gdzie jako zbiór testowy wykorzystywany jest zbiór *Inria Aerial Image Labeling Dataset*. Sieć neuronowa przedstawiona w niniejszej pracy korzysta z architektury typu *U-Net*, jej enkoder jest zasilany wagami modelu *Wide ResNet-50-2* [35], całość łączą bloki typu *Bottleneck* (w wersji OSNet [36]) gwarantujące agregację map cech z przekroju różnych skal. Aby efektywnie nauczyć się korelacji kanałów przestrzennych, uniknąć nadmiernego dopasowania oraz uzyskać większą efektywność obliczeniową, zaimplementowana sieć neuronowa wykorzystuje punktowo i wgębnie separowalne konwolucje. Efektywność głębokiego uczenia się jest gwarantowana przez funkcję straty złożoną z ważonej sumy: Binary Cross-Entropy Loss, Dice Loss oraz Lovász hinge loss. W kolejnych rozdziałach niniejszej pracy opisywane są podstawy teoretyczne głębokich sieci neuronowych, analizowana jest literatura dotycząca detekcji budynków na zdjęciach lotniczych i satelitarnych oraz szczegółowo definiowane są: architektura użytej sieci neuronowej, zbiór danych i jego transformacje, funkcje strat oraz metody oceny skuteczności działania zaimplementowanej sieci. Całość zakończona jest porównaniem uzyskanych wyników z wynikami przedstawionymi w literaturze.

## 2. Podstawy teoretyczne głębokiego uczenia

### 2.1. Głębokie uczenie - podstawowe definicje

Głębokie uczenie (ang. *deep learning*) jest jedną z najbardziej dynamicznie rozwijających się gałęzi sztucznej inteligencji (ang. *artificial intelligence*). Zadaniem głębokiego uczenia jest wychwycenie jak największej liczby zależności (cech) występujących w analizowanym zbiorze danych i przedstawienie tych zależności w prostszy sposób - przy pomocy sieci neuronowej z odpowiednio dobranymi wagami. Głębokie uczenie można sklasyfikować jako podzbiór uczenia reprezentacji (ang. *representation learning*), które z kolei jest podzbiorem uczenia maszynowego (ang. *machine learning*), a to w końcu stanowi podzbiór metod sztucznej inteligencji. Rysunek 2.1 prezentuje diagram zależności pomiędzy wyżej wymienionymi dziedzinami.



Rysunek 2.1. Diagram Venna ukazujący zależności pomiędzy głębokim uczeniem, uczeniem reprezentacji, uczeniem maszynowym a sztuczną inteligencją [10]

Głębokie uczenie jest implementowane przy pomocy głębokich sieci neuronowych (ang. *deep neural networks*), których architektura opiera się na połączonych ze sobą neuronach. Każde połączenie pomiędzy neuronami jest reprezentowane przy pomocy wagi, która wskazuje jak ważna jest wartość danego neuronu (w konkretnym połączeniu z innym neuronem) z perspektywy całej sieci. Neurony są grupowane w warstwy (w liczbie zazwyczaj od kilku do nawet kilkuset), a te z kolei grupuje się w bloki, czyli funkcjonalne

jednostki przetwarzania danych. Głębokie sieci neuronowe są implementowane przy pomocy bardzo zróżnicowanych architektur jednak architektury te zazwyczaj składają się z kilku podstawowych elementów (warstw). Do najważniejszych z nich należą: warstwa konwolucyjna, warstwa liniowa, warstwa grupująca, warstwa aktywacji, warstwa *dropout* oraz warstwa normalizacji partiami. Najbardziej popularnymi klasami głębokich sieci neuronowych są konwolucyjne sieci neuronowe (ang. *convolutional neural networks*) oraz rekurencyjne sieci neuronowe (ang. *recurrent neural networks*). Konwolucyjne sieci neuronowe charakteryzują się tym, iż ich architektura jest oparta o warstwy splotowe, które pozwalają na efektywną ekstrakcję cech. Natomiast rekurencyjne sieci neuronowe charakteryzuje występowaniem cyklicznych połączeń pomiędzy neuronami, które umożliwiają analizę danych o charakterze sekwencyjnym.

### Warstwa konwolucyjna (splotowa)

Warstwa konwolucyjna (ang. *convolution layer*) jest podstawową jednostką przetwarzania w głębokich sieciach neuronowych. Składa się ona z szeregu filtrów (ang. *filter / kernel*), o rozmiarach zazwyczaj nieprzekraczających kilku pikseli, które przesuwając się po danych wejściowych, reprezentowanych zazwyczaj w formie wielowymiarowej macierzy, wykonują operację iloczynu Hadamarda, po którym następuje sumowanie wszystkich elementów uzyskanej macierzy do pojedynczych liczb (ekstrakcja cech). Liczby te następnie tworzą nową macierz, która jest nazywana mapą cech (ang. *features map*). Każdy z filtrów zastosowanych w warstwie konwolucyjnej składa się początkowo z losowych wartości, w trakcie trenowania sieci, w kolejnych iteracjach obliczeń (zwanych powszechnie epokami (ang. *epochs*)), następuje aktualizacja wartości filtrów, aby sieć nabierała pożądanych od niej własności. Każdy filtr warstwy splotowej tworzy własną mapę cech, więc deklarując liczbę filtrów w danej warstwie, deklarujemy ile map cech (unikalnych atrybutów danego zbioru danych) na wyjściu z danej warstwy chcemy uzyskać. W ramach warstwy konwolucyjnej, poza rozmiarem filtrów (ich kształtem), należy zdefiniować jeszcze kilka ważnych parametrów (hiperparametrów<sup>2</sup>), takich jak:

- krok (ang. *stride*) o jaki filtr będzie przesuwany w czasie ekstrakcji cech,
- *padding* czyli wskazanie czy chcemy by tworzone mapy cech utrzymały rozdzielcość danych wejściowych, jeśli tak to jakimi wartościami należy uzupełnić ewentualne braki powstałe na skutek różnic pomiędzy rozmiarem danych wejściowych a rozmiarem filtra (domyślnie braki są uzupełniane wartością zero),
- dylacja (ang. *dilation*) czyli odległość pomiędzy punktami filtra - domyślnie filtr jest nakładany na sąsiadujące ze sobą elementy macierzy wejściowej, ale można przy

---

<sup>2</sup> Hiperparametrami sieci neuronowej nazywamy najważniejsze parametry tej sieci, które są definiowane przez jej architekta przed przystąpieniem do procesu trenowania.

pomocy dylacji ustawić by był on nakładany na przykład na co drugi jej element (dylacja równa 2).

Szczególnym przypadkiem warstwy konwolucyjnej jest **konwolucja transponowana** (ang. *transposed convolution*), którą można określić jako operację odwrotną do operacji splotu. O ile celem konwolucji jest uzyskanie uproszczonej reprezentacji danych wejściowych, o tyle konwolucja transponowana stara się przetransformować dane wejściowe o mniejszej wymiarowości, do danych o większym wymiarze. Stąd też konwolucję transponowaną często określa się jako nadpróbkowanie (ang. *upsampling*) i jest ona najczęściej wykorzystywana przy zadaniach związanych z rekonstrukcją danych. Innym szczególnym przypadkiem warstwy konwolucyjnej jest **konwolucja z filtrem o rozmiarze 1x1**, której najczęstszym zadaniem jest redukcja liczby map cech w danym miejscu sieci neuronowej, dzięki czemu dochodzi do istotnego zmniejszenie rozmiaru tej sieci bez dużego uszczerbku dla jej efektywności.

### Warstwa liniowa

Warstwa liniowa (ang. *linear layer / dense layer / fully-connected layer*) to najprostsza ze wszystkich warstw stosowanych w głębokich sieciach neuronowych. Jej zadaniem jest zastosowanie transformacji liniowej na danych pochodzących z poprzedzającej ją warstwy. Hiperparametrami warstwy liniowej są: liczba wejść (ang. *input size*), liczba wyjść (ang. *output size*) oraz flaga wskazująca czy przy przeprowadzaniu transformacji linowej należy stosować przesunięcie (ang. *bias*).

### Warstwa grupująca

Warstwa grupująca (ang. *pooling layer*) to jednostka, której zadaniem jest redukcja wymiarowości danych wejściowych poprzez zastosowanie wybranej funkcji grupującej (najczęściej funkcji maximum lub funkcji średniej). Podobnie jak w przypadku warstwy splotowej, po danych wejściowych warstwy grupującej przesuwany jest filtr o wymiarach kilku pikseli, którego zadaniem jest wybranie największej wartości z danego zakresu danych (*max pooling*) lub wyliczenie średniej dla tego zakresu (*average pooling*). W ten sposób, dla filtra (okna) warstwy grupującej o wymiarach na przykład 2x2, cztery wartości danych wejściowych są redukowane do jednej wartości - maksimum lub średniej z nich. W ramach warstwy grupującej definiowany jest taki sam zbiór hiperparametrów jak w przypadku warstwy konwolucyjnej czyli: rozmiar filtra, krok filtra, *padding* oraz dylacja - mają one dokładnie takie samo zastosowanie jak w przypadku warstwy splotowej. Poza zdefiniowanymi powyżej podstawowymi operacjami grupującymi często wyróżnia się jeszcze operację globalnego grupowania do maksimum (ang. *global max pooling*) oraz operację globalnego grupowania do średniej (ang. *global average pooling*), które polegają

na zastosowaniu funkcji maksimum / średniej nie na poziomie filtrów, ale na poziomie całych kanałów danych wejściowych.

### Warstwa aktywacji

Warstwa aktywacji (ang. *activation function*) przetwarza dane pochodzące z map cech wygenerowanych w poprzednich warstwach po to by nadać im niezbędną nielinowość. Jest ona konieczna po to by sieci neuronowe były w stanie nauczyć się skomplikowanych wzorców występujących w analizowanych przez nie danych. Do najpopularniejszych warstw (funkcji) aktywacji wykorzystywanych w głębokich sieciach neuronowych można zaliczyć:

- *sigmoid*:

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

- *tangens hiperboliczny*:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2)$$

- *ReLU*:

$$ReLU(x) = \max(x, 0) \quad (3)$$

- *Leaky ReLU*:

$$LReLU(x) = \begin{cases} x & \text{dla } x > 0 \\ 0,01 \cdot x & \text{dla } x \leq 0 \end{cases} \quad (4)$$

- *Parametric ReLU*:

$$PReLU(x) = \begin{cases} x & \text{dla } x > 0 \\ \alpha \cdot x & \text{dla } x \leq 0 \end{cases} \quad (5)$$

- *Exponential Linear Unit*:

$$ELU(x) = \begin{cases} x & \text{dla } x > 0 \\ \alpha \cdot (e^x - 1) & \text{dla } x \leq 0 \end{cases} \quad (6)$$

- *softmax*<sup>3</sup>:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} \quad (7)$$

Funkcje *sigmoid* oraz *tangens hiperboliczny*, ze względu na swoją złożoność obliczeniową i fakt, iż mogą się przyczyniać do występowania problemu zanikającego gradientu, są stosunkowo rzadko wykorzystywane w bieżących architekturach głębokich sieci neuronowych. Najczęściej stosowaną funkcją aktywacji jest *ReLU*, głównie ze względu na swoją prostotę i skuteczność. Często zarzuca się jej jednak brak różniczkowalności w punkcie 0 oraz zerową wartość pochodnej dla ujemnych argumentów, która może powodować wymieranie neuronów (*dead ReLU*). Stąd też opracowane zostały różne modyfikacje funkcji *ReLU*, które starają się rozwiązać te problemy przy okazji zachowując wszystkie najważniejsze zalety tej funkcji aktywacji (*Leaky ReLU*, *Parametric ReLU*, *Exponential Linear Unit*).

### Warstwa *dropout*

Głębokie sieci neuronowe ze względu na ogromną liczbę trenowalnych parametrów (wag) są szczególnie podatne na problem zwany przeuczeniem (ang. *overfitting*), który polega na zbyt dokładnym dopasowaniu się sieci neuronowej do konkretnych danych uczących, przez co sieć ta staje się mniej skuteczna w generowaniu prawidłowych predykcji dla danych z poza zbioru uczącego (rośnie jej błąd generalizacji). Zadaniem warstwy *dropout* jest zmiana funkcjonowania sieci neuronowej w taki sposób by jak najbardziej zminimalizować błąd generalizacji modelu. Rozwiązaniem tego problemu, uzyskiwanym w ramach warstwy *dropout*, jest losowe wyłączanie (zerowanie aktywacji) niektórych neuronów sieci w każdej epoce jej uczenia, po to by miała ona zdolność wyłapywania jak najbardziej uniwersalnych cech danego zbioru danych, niezwiązanych z konkretnymi wartościami poszczególnych neuronów. To jaki procent wszystkich neuronów zostanie wyłączonych w konkretnej epoce jest definiowane przy pomocy hiperparametru  $p$ , czyli prawdopodobieństwa, że dany neuron zostanie wyzerowany.

Warstwę *dropout* w związku z jej właściwością zapobiegania przeuczeniu sieci neuronowej nazywa się często techniką regularyzacji. Inną równie często stosowaną techniką regularyzacji modelu jest **augmentacja danych** (ang. *data augmentation*), czyli technika transformacji danych treningowych sieci neuronowej w taki sposób by jak najbardziej je zróżnicować, uzyskując tym samym większą liczbę przypadków uczących. Jednymi z najpopularniejszych sposobów przekształcania danych uczących są: skalowanie (ang. *scale*),

---

<sup>3</sup> Softmax jest specyficzną funkcją aktywacji, gdyż jej głównym zadaniem jest normalizacja danych wyjściowych sieci neuronowej do rozkładu prawdopodobieństwa

przesuwanie (ang. *shift*), losowe obracanie (ang. *random rotation*), losowe wycinanie fragmentów (ang. *random crop*) czy normalizacja (ang. *normalization*).

### Warstwa normalizacji partiami

Zadaniem warstwy normalizacji partiami (ang. *batch normalization layer*) jest zaradzenie problemowi ciągłej zmiany rozkładów wartości generowanych przez poszczególne warstwy głębokich sieci neuronowych wraz z postępem ich uczenia. W związku z tym zjawiskiem poszczególne warstwy modelu dużą część swojej pracy poświęcają na dostosowanie się do wewnętrznego przesunięcia rozkładów (ang. *internal covariate shift*), zamiast spożytkować tę pracę na efektywną naukę. Rozwiązaniem tego problemu było dodanie nowej warstwy do architektury modelu, której zadaniem była odpowiednia standaryzacja wartości wyjściowych z poprzedniej warstwy poprzez odjęcie średniej wartości w danej partii (ang. *batch*) i podzielenie przez odchylenie standardowe z tej partii. Zastosowanie warstwy normalizacji partiami pozwala na efektywniejszą naukę sieci neuronowej - sieć w czasie treningu szybciej minimalizuje funkcję strat.

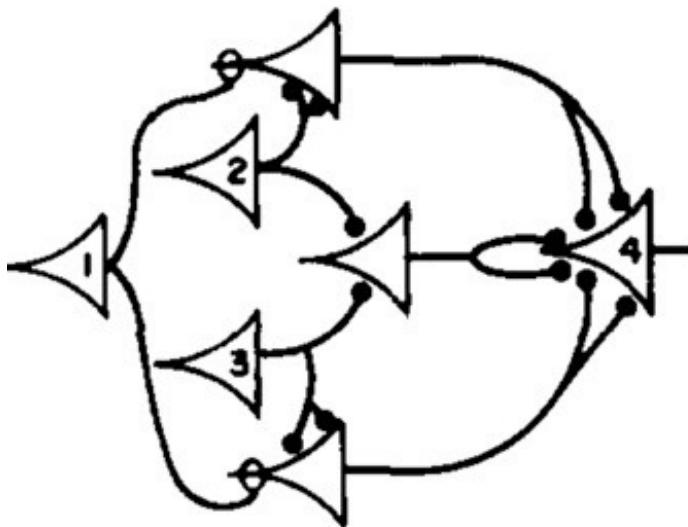
## 2.2. Ewolucja architektur głębokiego uczenia

Termin głębokie uczenie zyskał w ostatnim czasie ogromną popularność, głównie dzięki mnogości zastosowań jaką ta gałąź sztucznej inteligencji znalazła w szerokiej gamie dziedzin życia codziennego - od samochodów autonomicznych, przez asystentki głosowe w urządzeniach mobilnych / *smart home*, aż po systemy rozpoznawania twarzy na lotniskach. Jednak termin ten nie jest nowy w świecie nauki, był on używany przez naukowców od wielu lat, a jego korzenie można datować na lata 40-ste dwudziestego wieku.

### 2.2.1. Cybernetyka (1943 - 1969)

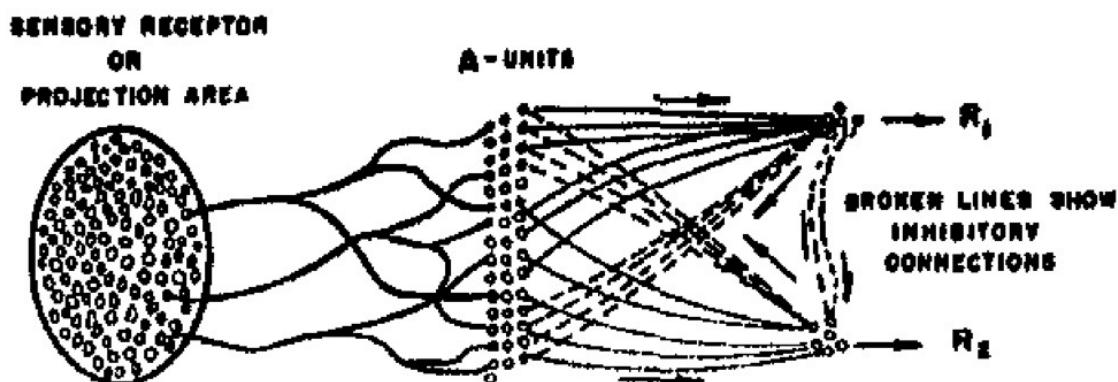
Praprzodkiem głębokiego uczenia, rozwijanym w latach 40., 50. i 60. XX wieku, była cybernetyka, czyli nauka badająca mechanizmy kontroli i komunikacji u zwierząt i maszyn [32]. W ramach cybernetyki naukowcy starali się stworzyć modele mogące naśladować biologiczną pracę mózgu. Najważniejszymi osiągnięciami tamtego okresu były następujące modele liniowe:

- **model neuronu** stworzony w 1943 roku przez Warrena McCullocha i Waltera Pittsa (przez autorów nazywany *Threshold Logic Unit*) [21], który oparty był na równaniu liniowym z arbitralnie dobranymi wagami przy zmiennych wejściowych (patrz rysunek 2.2),



Rysunek 2.2. Przykładowy schemat neuronu zaprezentowany w 1943 roku przez Warrena McCullocha i Waltera Pittsa [21]

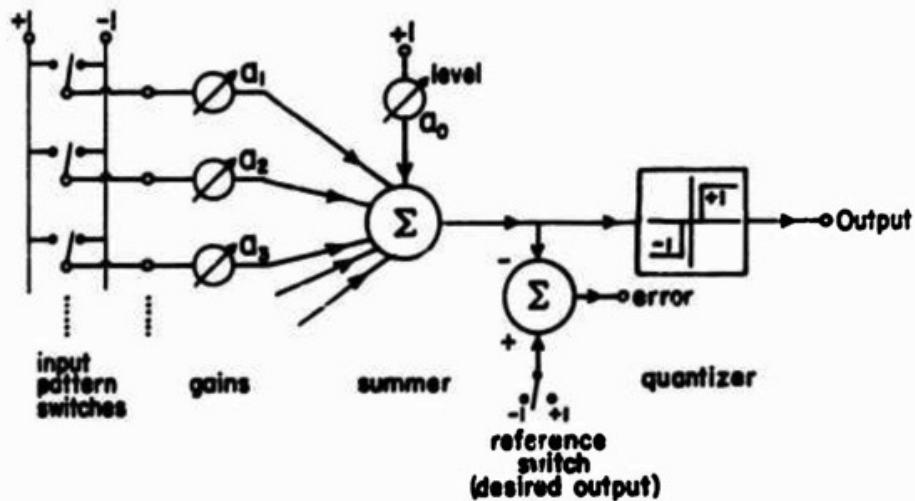
- **model perceptronu** zaprezentowany w 1958 roku przez Franka Rosenblatta [26], który był rozwinięciem neuronu o możliwość samodzielnej nauki wartości wag przy poszczególnych zmiennych na podstawie danych treningowych<sup>4</sup> (patrz rysunek 2.3),



Rysunek 2.3. Schematyczna reprezentacja połączeń w prostym perceptronie zaprezentowana w 1958 roku przez Franka Rosenblatta [26]

- model adaptacyjnego neuronu liniowego stworzony przez Bernarda Widrowa i Teda Hoffa, określany skrótem **ADALINE** (ang. *Adaptive Linear Neuron*), który od perceptronu różnił się między innymi tym, że w czasie uczenia jego wagi były dostosowywane do ważonej sumy wartości wejściowych (patrz rysunek 2.4).

<sup>4</sup> Wielu naukowców uznaje model perceptronu zaprezentowany przez Franka Rosenblatta za pierwszą współczesną sieć neuronową (*feedforward neural network*).



**Rysunek 2.4.** Schemat ADALINE zaprezentowany w 1960 roku przez Bernarda Widrowa i Teda Hoffa [31]

Pod koniec lat 60. XX wieku popularność modeli tworzonych w nurcie cybernetyki istotnie zmalała za sprawą licznych ograniczeń modeli liniowych wskazywanych przez część naukowców. W 1969 roku ukazała się słynna książka Marvin Minsky'ego i Seymoura Paperta pod tytułem *Perceptrons: an introduction to computational geometry* [22], która wskazywała na liczne ograniczenia modelu pojedynczego perceptronu Rosenblatta, takie jak na przykład fakt, iż nie jest on w stanie nauczyć się prostej operacji logicznej XOR (ani żadnej innej operacji nieseparowalnej liniowo). Minksy i Papert wskazywali jednocześnie, iż takich operacji mogłyby nauczyć się sieć złożona z kilku perceptronów, lecz nie ma efektywnych sposobów uczenia takich sieci. Ta publikacja istotnie przyczyniła się do spadku popularności modeli uczenia inspirowanych biologicznie.

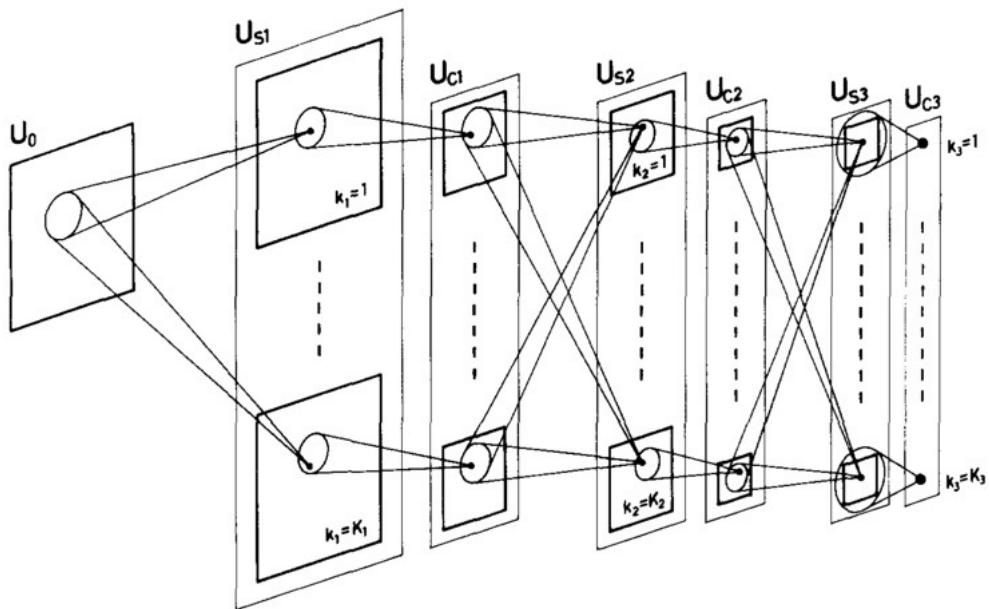
### 2.2.2. Konekcjonizm (1980 - 1998)

Kolejnym przedkiem głębokiego uczenia, który najdynamiczniej rozwijał się w latach 80. i 90. XX wieku, był konekcjonizm, czyli podejście z dziedziny kognitywistyki, które miało za zadanie wyjaśnić zjawiska umysłowe przy pomocy modeli neuronowych. Główną ideą tego nurtu sztucznej inteligencji było przekonanie, iż duża liczba prostych jednostek obliczeniowych połączonych w sieć może umożliwić uzyskanie intelligentnego zachowania u maszyny.

#### Neocognitron

Jednymi z najważniejszych artykułów początkowej fazy konekcjonizmu były artykuły Kunihiko Fukushima wprowadzające pojęcia **cognitronu** [5] i **neocognitronu** [6], czyli samoorganizującej się sieci neuronowej (wzorowanej na systemie widzenia ssaków), która jest w stanie rozpoznawać wzorce według geometrycznego podobieństwa ich kształtu.

*Neocognitron* był rozwinięciem *cognitronu* wprowadzającym do sieci neuronowej charakter splotowy, co pozwoliło na uniezależnienie odpowiedzi modelu od umiejscowienia wzorca.

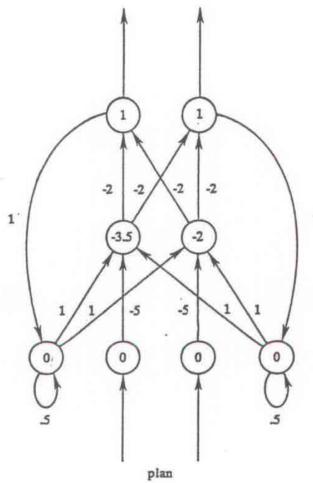


Rysunek 2.5. Diagram pokazujący połączenia pomiędzy warstwami w neocognitronie zaprezentowanym w 1980 roku przez Kunihiko Fukushimę [6]

*Neocognitron* można więc uznać za jeden z pierwszych modeli z dziedziny **konwolucyjnych sieci neuronowych**. Składał się on z sześciu warstw splotowych przeplatanych warstwami grupującymi (patrz rysunek 2.5). Wejściem do modelu *neocognitronu* były obrazy o rozmiarach  $16 \times 16$ , a każda warstwa splotowa zawierała 24 filtry o wymiarach  $5 \times 5$ . Po wytrenowaniu model Fukushimy był w stanie rozpoznać pięć cyfr („0”, „1”, „2”, „3”, „4”) i cztery litery („X”, „Y”, „T”, „Z”).

### Recurrent Neural Network (RNN)

Prace Fukushimy przyczyniły się do rozwoju konwolucyjnych sieci neuronowych, które dawały bardzo obiecujące wyniki w dziedzinie analizy obrazu, jednak miały one ograniczony potencjał zastosowania w dziedzinach, w których liczy się analiza całych sekwencji danych, jak na przykład analiza mowy, dźwięku, wideo czy pisma odręcznego. Do tego typu zastosowań potrzebna jest sieć, która dysponuje wewnętrzną pamięcią, po to by móc wyłapywać zależności nie tylko wewnątrz bieżącego fragmentu danych, ale także pomiędzy kolejnymi pojawiającymi się sekwencjami. Głębokie sieci neuronowe, które posiadają taką własność, nazywamy **rekurencyjnymi sieciami neuronowymi**. Jednym z pierwszych artykułów z nurtu konekcionizmu, wprowadzających rekurencyjne połączenia do sieci neuronowych, i w ten sposób tworzących sieci z dynamiczną pamięcią, był artykuł *Serial order: A parallel distributed processing approach* napisany w 1986 roku przez Michaela Irwina Jordana [17].

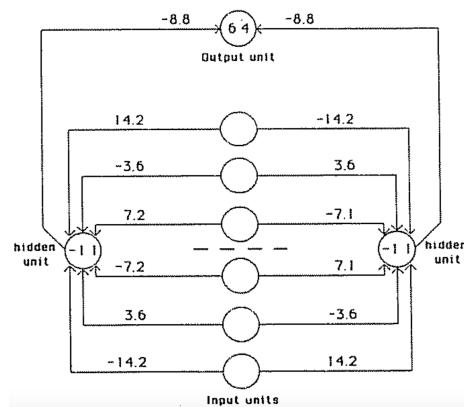


**Rysunek 2.6.** Diagram pokazujący połączenia w sieci rekurencyjnej zaprezentowanej w 1986 roku przez Michaela Irwina Jordana [17]

W swoim artykule Jordan proponuje nowatorskie podejście, które umożliwia nauczenia się sekwencji zdarzeń przez sieć neuronową, gdzie każda ukryta komórka (ang. *hidden cell*) otrzymuje swoje własne dane wyjściowe z odpowiednim, ustalonym opóźnieniem (patrz rysunek 2.6).

### Backpropagation

Dynamiczny rozwój konekcjonizmu nie byłby możliwy bez opracowania efektywnego **algorytmu wstecznej propagacji błędu** (ang. *backpropagation algorithm*). Jako jego twórcę uznaje się Paula Werbosa, który opisał go już w 1974 roku w swojej rozprawie doktorskiej. Jednak algorytm ten nie zyskał dużego rozgłosu aż do 1986 roku, kiedy to David Rumelhart, Geoffrey Hinton i Ronald Williams w swoim artykule *Learning representations by back-propagating errors* [27] pokazali zastosowanie algorytmu wstecznej propagacji błędu do trenowania **wielowarstwowego perceptronu** (ang. *Multilayer Perceptron*).



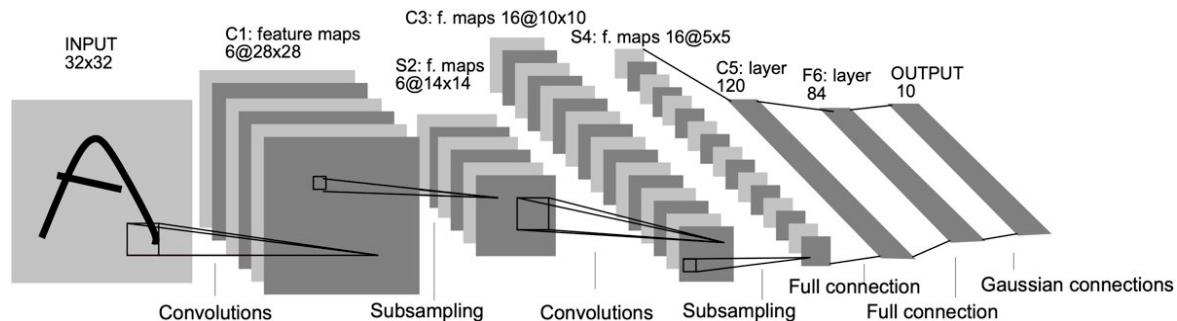
**Rysunek 2.7.** Schemat wielowarstwowego perceptronu zaprezentowany w 1986 roku przez Davida Rumelharta, Geoffreya Hintona i Ronaldsa Williamsa [27]

## 2. Podstawy teoretyczne głębokiego uczenia

W czasie treningu sieci neuronowej algorytm wstępnej propagacji błędu wylicza gradient funkcji strat w odniesieniu do wag tej sieci przy pomocy reguły łańcuchowej (ang. *chain rule*). Gradient ten jest wyliczany dla każdej warstwy sieci neuronowej poczynając od najgłębszych warstw przechodząc iteracyjnie ku warstwom początkowym.

### LeNet-5

Kontynuatorem koncepcji Kunihiko Fukushima, który do wytrenowania splotowej sieci neuronowej o architekturze wzorowanej na *neocognitronie* zastosował algorytm wstępnej propagacji błędu przedstawiony przez Davida Rumelharta, Geoffreya Hintona i Ronaldą Williamsa, był francuski naukowiec Yann André LeCun. W swoich dwóch artykułach *Back-propagation Applied to Handwritten Zip Code Recognition* z 1989 roku [19] i *Gradient-based learning applied to document recognition* z 1998 roku [20] zaprezentował on konwolucyjną sieć neuronową, która była w stanie zadowalającą, jak na tamten czas, skutecznością rozpoznawać znaki pisma odręcznego. Sieć z 1998 roku zyskała nazwę **LeNet-5**, składała się z około 60 tysięcy trenowalnych parametrów podzielonych na siedem warstw (cztery splotowo-grupujące i trzy warstwy gęste) o filtrach w rozmiarze 5 x 5, a jako wejście przyjmowała obrazy o rozdzielcości 32 x 32 (patrz rysunek 2.8). Co ciekawe, żeby ograniczyć liczbę trenowanych parametrów / połączeń niektóre warstwy modelu LeCunna łączyły się jedynie z wybranymi filtrami z warstw poprzednich.



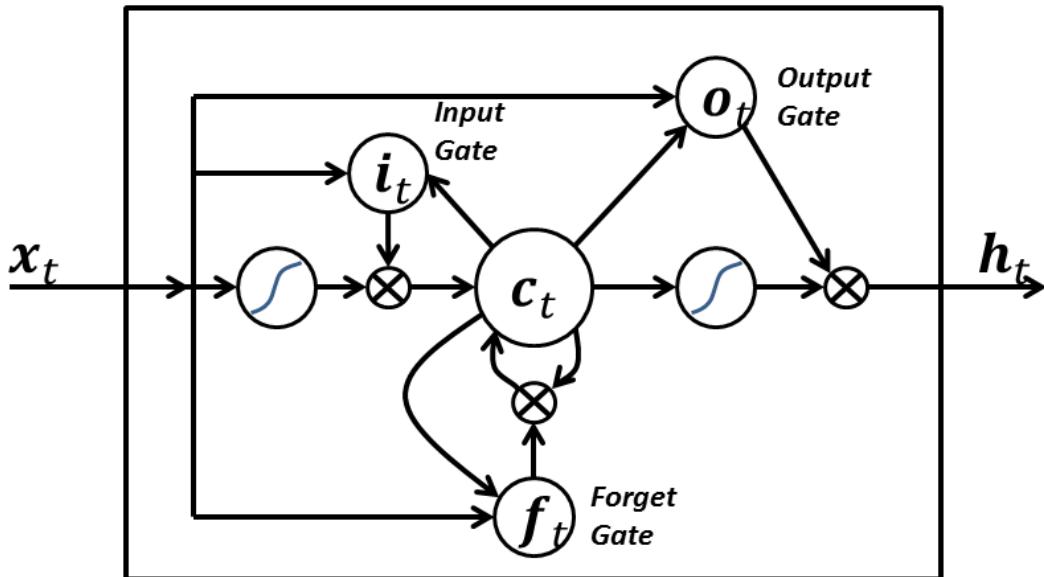
Rysunek 2.8. Architektura sieci LeNet-5 stworzona w 1998 roku przez Yanna LeCuna [20]

Sieć LeNet-5 stała się kamieniem milowym w rozwoju sieci neuronowych, w szczególności konwolucyjnych sieci neuronowych, gdyż pokazała, iż wielowarstwowe modele mogą osiągać w niektórych zastosowaniach lepsze rezultaty niż inne dostępne w tamtym czasie techniki rozpoznawania pisma odręcznego. Artykuł Yanna LeCunna z 1998 roku przyczynił się również do powstania sprawdzonego schematu trenowania sieci neuronowej z warstwami splotowo-grupującymi zakończonymi warstwami gęstymi, schemat ten w kolejnych latach był wielokrotnie powielany i rozszerzany przez wielu autorów, lecz zasadniczy trzon architektury splotowych sieci neuronowych pozostał praktycznie niezmieniony.

### Long Short-Term Memory (LSTM)

Drugim kluczowym artykułem, który powstał u schyłku konekcyjizmu był artykuł z 1997 roku zatytułowany *Long short-term memory* napisany przez Seppa Hochreitera i Jürgena Schmidhubera [14]. Wprowadził on nową architekturę **LSTM** w dziedzinie rekurencyjnych sieci neuronowych, która pozwalała rozwiązać istotny problem dotykający w tamtym czasie sieci *RNN*, a mianowicie problem zanikającego gradientu (ang. *vanishing gradient problem*)<sup>5</sup>. Sieć *LSTM* wprowadzała wewnętrzne mechanizmy zwane bramkami (ang. *gates*), które pozwalały kontrolować przepływ informacji przetwarzanych przez sieć neuronową. Bramki te razem z komórkami pamięci (ang. *memory cells*) miały za zadanie nauczyć się, które informacje z poprzednich sekwencji danych należy zatrzymać (uznając je za ważne), a które można pominąć (nieważne). W ramach modelu *LSTM* wyróżniano trzy bramki:

- bramkę zapomnienia (ang. *forget gate*) - bramka ta decydowała, którą informację należy zapomnieć, a którą zachować,
- bramkę wejścia (ang. *input gate*) - bramka ta aktualizowała wewnętrzny stan komórki pamięci,
- bramkę wyjścia (ang. *output gate*) - bramka ta decydowała jaki powinien być następny ukryty stan (ang. *hidden state*) sieci.



**Rysunek 2.9.** Architektura sieci *LSTM* zaprezentowanej w 1997 roku przez Seppa Hochreitera i Jürgena Schmidhubera [14]

<sup>5</sup> Problem zanikającego gradientu pojawia się gdy, po przekroczeniu pewnej głębokości (długości przetwarzanej sekwencji), sieci neuronowe nie są w stanie efektywnie uczyć się (aktualizować swoich wag), gdyż gradient błędów dla głębokich warstw (odległych sekwencji) staje się zbyt mały.

## 2. Podstawy teoretyczne głębokiego uczenia

---

Istotnymi ograniczeniami rozwoju sieci neuronowych tworzonych u schyłku XX wieku były skromne, z dzisiejszej perspektywy, zasoby obliczeniowe ówczesnych komputerów, które powodowały, iż wytrenowanie nawet najprostszej sieci neuronowej trwało wiele dni, stąd też metody stosowane przez przedstawicieli nurtu konektywizmu stopniowo traciły na popularności.

### 2.2.3. Głębokie uczenie (2006 - 2020)

Za swoisty renesans nurtu konektywizmu i właściwy początek głębokiego uczenia, można uznać rok 2006 kiedy to Hinton G., Osindero S. i Teh Y. W. w swoim artykule *A Fast Learning Algorithm for Deep Belief Nets* [12] pokazali, że sieci głębokiej wiarygodności mogą być efektywnie trenowane przy pomocy strategii o nazwie chciwe warstwowe trenowanie wstępne (ang. *greedy layer-wise pretraining*), która pozwalała zmniejszyć problem zanikającego gradientu oraz wzmacnić generalizację na zbiorze testowym poprzez efektywniejszą inicjalizację wag. Kolejni autorzy, wzorując się na pracy zespołu Geoffreya Hintona, szybko pokazali, że strategia chciwego warstwowego trenowania wstępnego może być zastosowana też do innych rodzajów głębokich sieci neuronowych, co otworzyło drogę do tworzenia modeli o niedostępnych dotąd głębokościach.

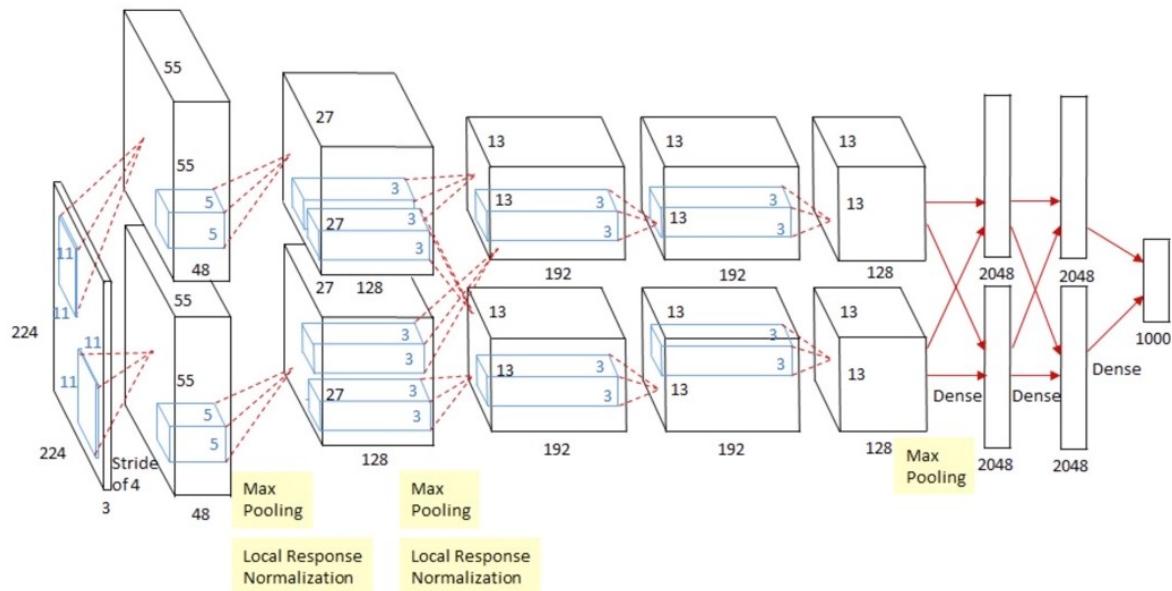
#### AlexNet

W tym samym czasie, dynamiczny rozwój przemysłu gier komputerowych i związany z nim rozwój kart graficznych, umożliwił bezprecedensowy wzrost efektywności trenowania głębokich sieci neuronowych - zamiast tradycyjnego trenowania sieci na jednostkach CPU (ang. *central processing unit*) zaczęto je trenować na jednostkach GPU (ang. *graphics processing unit*). Przełomowym artykułem w zakresie trenowania modelu na wielu kartach graficznych był artykuł *ImageNet classification with deep convolutional neural networks* opublikowany w 2012 roku przez Krizhevsky A., Sutskever I. i Hinton G. [18]. Co prawda nie był to pierwszy artykuł ukazujący ogromne możliwości GPU w zakresie trenowania głębokich sieci neuronowych<sup>6</sup>, jednak stał się jednym z najczęściej cytowanych artykułów w dziedzinie głębokiego uczenia, gdyż łączył w sobie wiele innowacyjnych na tamten czas rozwiązań takich jak:

- trening sieci na ogromnym zbiorze danych - *ImageNet* z 15 milionami podpisanych obrazów dla ponad 22 tysięcy kategorii,
- architektura modelu w postaci konwolucyjnej sieci neuronowej,
- trening sieci neuronowej jednocześnie na dwóch kartach graficznych,
- *ReLU* jako funkcja aktywacji przyspieszająca zbieżność procesu uczenia,

<sup>6</sup> Pierwszym takim artykułem był *High Performance Convolutional Neural Networks for Document Processing* napisany w 2006 roku przez Chellapilla K., Puri S. i Simard P.

- augmentacja danych treningowych realizowana poprzez translację, obroty horyzontalne i wycinanie fragmentów obrazów,
- zastosowanie warstw *dropout* w celu zapobiegnięcia przetrenowaniu sieci,
- trenowanie modelu w pakietach przy użyciu stochastycznego spadku gradientu (ang. *batch stochastic gradient descent*) z jasno określonymi parametrami momentum i spadku wagi (ang. *weight decay*).



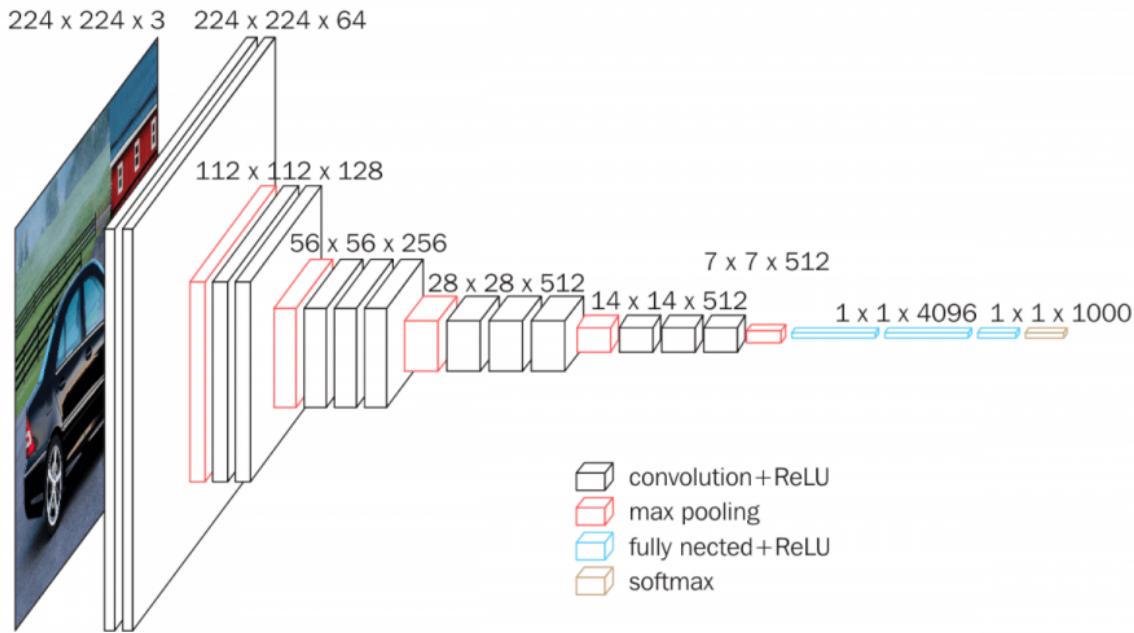
**Rysunek 2.10.** Architektura sieci *AlexNet* zaprezentowana w 2012 roku przez Alexa Krizhevsky'ego, Ilya Sutskevera i Geoffrey'a Hintona [18]

Sieć stworzona przez Krizhevsky'ego, Sutskevera i Hintona swoją architekturą bardzo przypominała sieć *LeNet-5*, była jednak od niej wielokrotnie większa pod względem liczby parametrów. Składała się ona bowiem z około 60 milionów trenowalnych parametrów podzielonych na osiem warstw (pięć warstw konwolucyjno-grupujących i trzy warstwy gęste) o filtrach rozmiarów 11 x 11, a jako wejście przyjmowała obrazy RGB o rozdzielczości 224 x 224 (patrz rysunek 2.10). Sieć ta jest obecnie powszechnie znana pod nazwą ***AlexNet*** (od imienia jej głównego autora), wygrała ona w 2012 roku prestiżowy konkurs *ImageNet Large Scale Visual Recognition Challenge* pokonując drugi najlepszy model o ponad 10 punktów procentowych (pod kątem stopy błędów top-5), zapoczątkowując tym samym swoistą rewolucję w dziedzinie głębokiego uczenia.

### VGGNet

Kolejnym istotnym modelem z perspektywy rozwoju głębokich sieci neuronowych był model ***VGGNet*** zaprojektowany w 2014 roku przez Karen Simonyan i Andrew Zissermana. W swoim artykule *Very deep convolutional networks for large-scale image recognition* [28] zaproponowali oni konwolucyjną sieć neuronową o niespotykanej dotąd głębokości

liczącej 19 warstw i o filtrach wielkości  $3 \times 3$  (patrz rysunek 2.11). Użycie tak małych filtrów, autorzy argumentowali tym, iż zastosowanie modułu składającego się z trzech filtrów  $3 \times 3$ , ma efektywne pole postrzegania jak jeden filtr  $7 \times 7$ , za to wymaga zdecydowanie mniejszej liczby trenowalnych parametrów i pozwala zastosować trzykrotnie funkcję aktywacji. Pomimo tych zabiegów sieć *VGGNet* posiadała bardzo duże rozmiary - składała się bowiem z około 144 milionów trenowalnych parametrów.

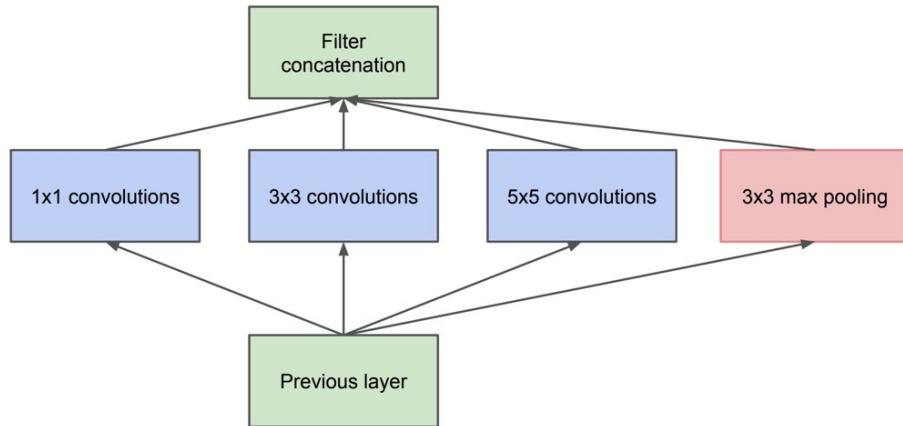


**Rysunek 2.11.** Architektura sieci *VGGNet* zaprezentowana w 2014 roku przez Karen Simonyan i Andrew Zissermana [28]

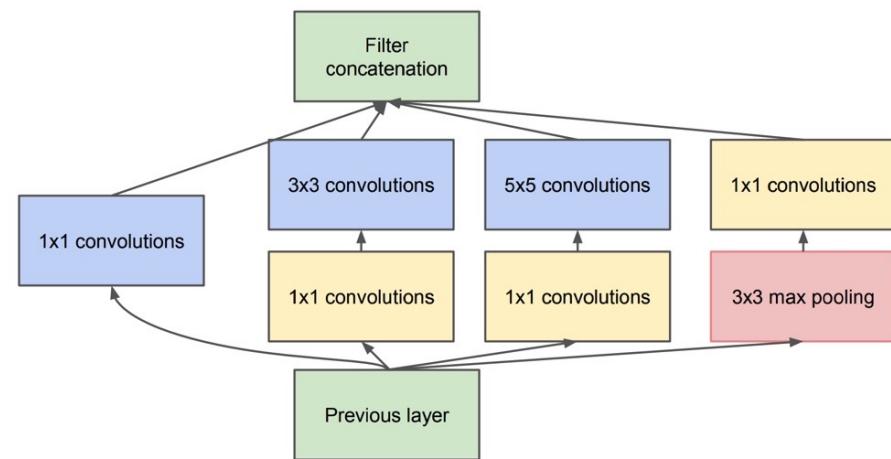
### GoogleNet

Sieć *VGGNet* zajęła drugie miejsce w konkursie *ImageNet Large Scale Visual Recognition Challenge 2014* osiągając stopę błędów top-5 na poziomie 7,32%. Pierwsze miejsce w tym konkursie zajęła sieć ***GoogleNet***, osiągając wynik równy 6,67%. Sieć ta została zaproponowana w październiku 2014 roku przez Szegedy Ch., Liu W., Jia Y., Sermanet P., Reed S., Anguelov D., Erhan D., Vanhoucke V., Rabinovich A. w artykule *Going deeper with convolutions*. Do momentu powstania *GoogleNet* większość głębokich sieci neuronowych była realizowana przy pomocy sekwencyjnego nakładania na siebie warstw splotowych i grupujących, sieć zaprojektowana przez pracowników *Google Inc.* jako pierwsza wprowadziła przetwarzanie równoległe realizowane w blokach *Inception*. Bloki te były realizowane jako równoległe warstwy konwolucyjne o filtrach różnych rozmiarów ( $1 \times 1$ ,  $3 \times 3$ ,  $5 \times 5$ ), których wynik był scalany (przy pomocy operacji konkatenacji) w jedną warstwę na wyjściu z danego bloku (patrz rysunek 2.12). Zastosowanie w blokach *Inception* filtrów o kilku różnych rozmiarach miało umożliwić detekcję cech o dużym zróżnicowaniu przestrzennym. Zastosowanie filtrów  $3 \times 3$  służyło redukcji złożoności sieci, dzięki czemu

finalny model zawierał około pięć milionów trenowalnych parametrów, czyli mniej więcej 12 razy mniej niż dużo mniej efektywna sieć *AlexNet* z 2012 roku. Cała architektura modelu *GoogleNet* składała się z 22 połączonych ze sobą sekwencyjnie bloków, z czego dziewięć stanowiły moduły *Inception*, łącznie w całej sieci można było wyróżnić około 100 różnych warstw (patrz rysunek 2.13).



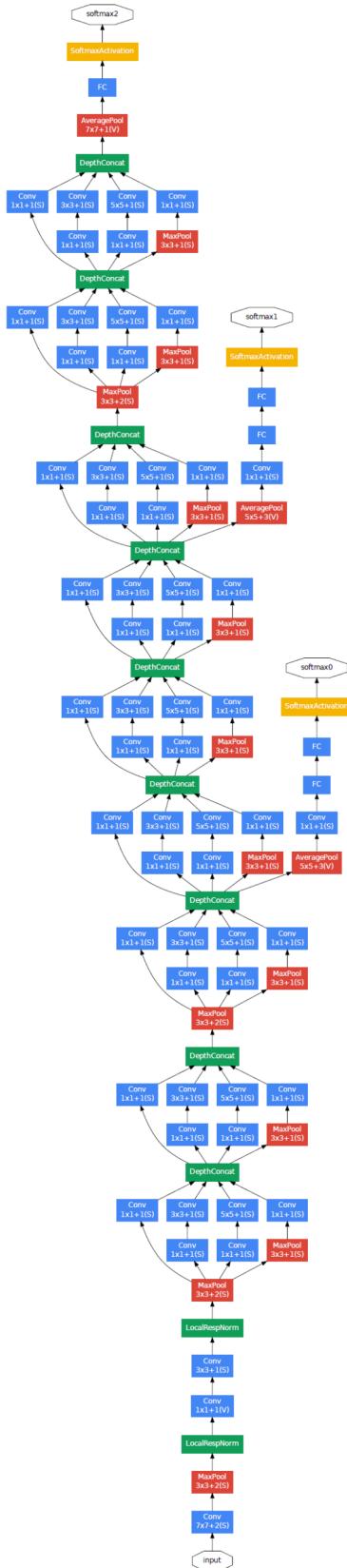
(a) Inception module, naïve version



(b) Inception module with dimension reductions

Rysunek 2.12. Architektura bloku *Inception* zaprezentowana w 2014 roku w artykule *Going deeper with convolutions* [30]

W kolejnych latach autorzy modelu *GoogleNet* wprowadzili liczne poprawki do opisywanej przez siebie architektury, dodali m.in. operację normalizacji partiami, faktoryzację konwolucji w celu poprawienia efektywności obliczeniowej, usprawnili architekturę bloków *Inception* poprzez dodanie połączeń *residual connections* oraz zastosowali konwolucje separowalne wgłębiście (ang. *depthwise separable convolutions*) i punktowo (ang. *pointwise separable convolutions*). W ten sposób powstały kolejne, jeszcze bardziej skuteczne, architektury wyrastające wprost z *GoogleNet* (zwanej też *Inception-v1*), były nimi: *Inception-v2* [2015], *Inception-v3* [2015], *Inception-v4* [2016] oraz *Xception* [2017].

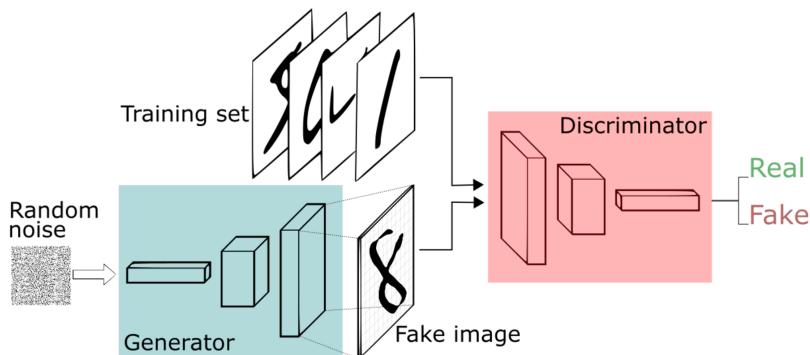


**Rysunek 2.13.** Architektura sieci *GoogleNet* zaprezentowana w 2014 roku w artykule *Going deeper with convolutions* [30]

### Generative Adversarial Nets (GAN)

W czasie gdy konwolucyjne sieci neuronowe ewoluowały w stronę coraz większej liczby warstw i coraz większej głębokości, naukowcy z Uniwersytetu w Montrealu: Goodfellow I., Pouget-Abadie J., Mirza M., Xu B., Warde-Farley D., Ozair S., Courville A. i Bengio Y. w swojej pracy z 2014 roku pod tytułem *Generative Adversarial Nets* [10] zaproponowali zupełnie nową architekturę głębokich sieci neuronowych określanych mianem sieci generatywnych z adwersarzem - **GAN**. W ramach GAN jednocześnie trenowane są dwa modele:

- model generatywny (ang. *generative model*), który stara się odwzorować dystrybucję danych w zbiorze treningowym, tak, żeby zmaksymalizować prawdopodobieństwo, że dyskryminator popełni błąd,
- model dyskryminacyjny (ang. *discriminative model*), który szacuje prawdopodobieństwo, że próbka pochodzi z danych uczących, a nie z generatora.



**Rysunek 2.14.** Wizualizacja architektury modelu GAN (Źródło: <https://jrmerwin.github.io/deep-learning4j-docs/generative-adversarial-network.html>)

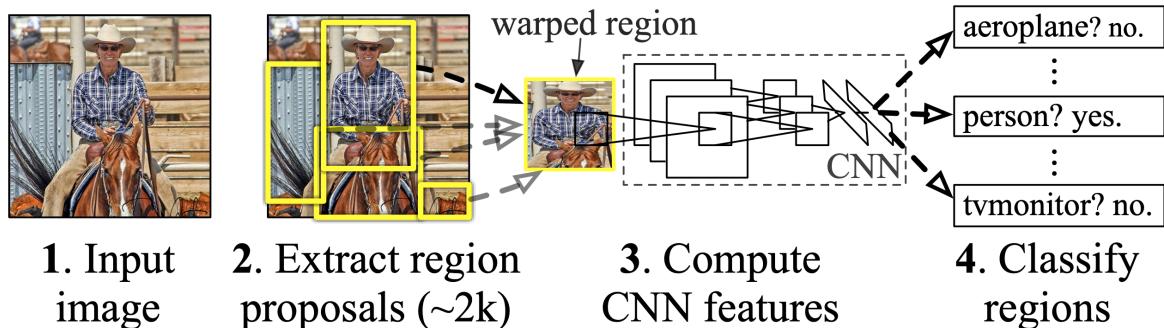
W ten sposób pomiędzy generatorem a dyskryminatorem dochodzi do gry o sumie zerowej, której celem jest jak najlepsze wytrenowanie generatora w ekstrakcji i odwzorowaniu cech ze zbioru treningowego. GAN jest dobrym przykładem uczenia nienadzorowanego (ang. *unsupervised learning*), które w przeciwieństwie do uczenia nadzorowanego (ang. *supervised learning*) nie potrzebuje posiadać w zbiorze treningowym etykiet wskazujących jaki jest oczekiwany wynik, który ma zostać wygenerowany przez model.

### Region Based Convolutional Neural Networks (R-CNN)

Rok 2014 był bardzo istotnym rokiem w dziejach rozwoju głębokich sieci neuronowych, gdyż w tym właśnie roku poza tak istotnymi architekturami jak *VGGNet*, *GoogleNet* czy GAN, powstała jeszcze jedna bardzo ważna architektura o nazwie *Region Based Convolutional Neural Networks (R-CNN)*. Została ona zaproponowana przez Rossa Girshicka, Jeffa Donahue'a, Trevora Darrella i Jitendra Malika w artykule *Rich feature hierarchies for accurate object detection and semantic segmentation* [8]. Celem sieci R-CNN była detekcja

## 2. Podstawy teoretyczne głębokiego uczenia

obiektów na zdjęciach, czyli nie tylko wskazanie, że na danym zdjęciu jest określony obiekt, ale też zaznaczanie przy pomocy ramki, gdzie dokładnie dany obiekt się znajduje (patrz rysunek 2.15)



Rysunek 2.15. Schemat działania modelu *R-CNN* zaprezentowany w 2014 roku przez Rossa Girshicka, Jeffa Donahue'a, Trevora Darrella i Jitendra Malika [8]

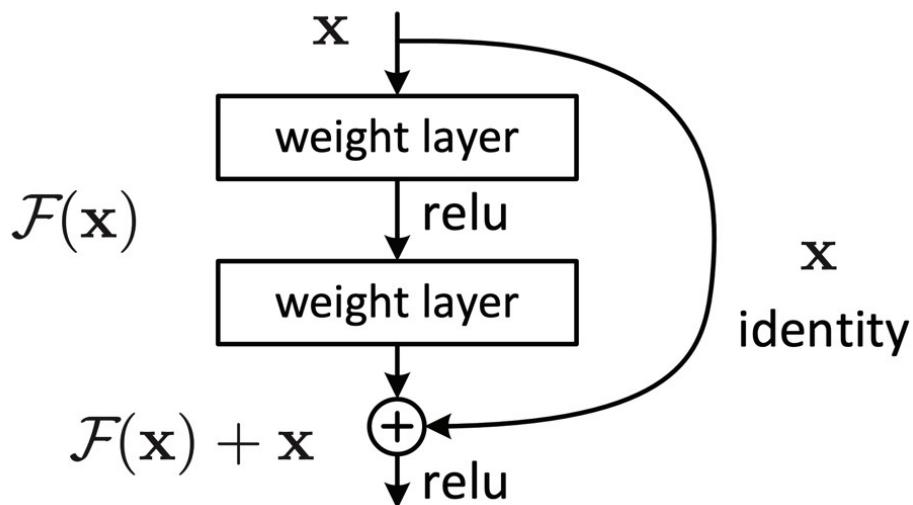
Oryginalna sieć *R-CNN* bazowała na algorytmie o nazwie *Selective Search*, który dzielił obraz wejściowy na około dwa tysiące regionów zainteresowania (ang. *regions of interest*), które następnie były przepuszczane przez konwolucyjną sieć neuronową, która ekstraktywała najważniejsze cechy danego *ROI* i przydzielała obiekt, który znajdował się w danym regionie, do odpowiedniej kategorii. Taka konstrukcja sieci *R-CNN* miała jednak wiele wad - wieloetapowość, nieefektywność obliczeniową czy bardzo wolne działanie (detekcja obiektów na jednym obrazie trwała około 47 sekund). Stąd też w kolejnych latach autorzy oryginalnego artykułu zaproponowali bardziej efektywne obliczeniowo implementacje *R-CNN*:

- sieć *Fast R-CNN* [2015] - była rozwinięciem sieci *R-CNN*, w którym konwolucyjna sieć neuronowa nie ekstraktywała cech z poszczególnych regionów zainteresowania tylko z całego obrazu wejściowego a algorytm *Selective Search* był stosowany dopiero na wyekstraktowanych mapach cech, co znaczco przyspieszyło działanie sieci, gdyż ekstrakcja cech przy pomocy sieci CNN miała miejsce tylko jeden raz (zamiast dwóch tysięcy razy),
- sieć *Faster R-CNN* [2016] - była rozwinięciem *Fast R-CNN*, w którym algorytm *Selective Search* został zastąpiony przez oddzielną sieć neuronową, która uczyła się przewidywania regionów zainteresowania,

Obie powyższe implementacje znaczco przyspieszyły detekcję obiektów - sieć *Fast R-CNN* była ponad 20 razy, a sieć *Faster R-CNN* ponad 200 razy, szybsza w detekcji obiektów niż oryginalna sieć *R-CNN*. Uzyskanie tak dużej efektywności sieci *Faster R-CNN* sprawiło, iż mogła ona być stosowana do detekcji obiektów w czasie rzeczywistym (ang. *real-time object detection*).

### ResNet

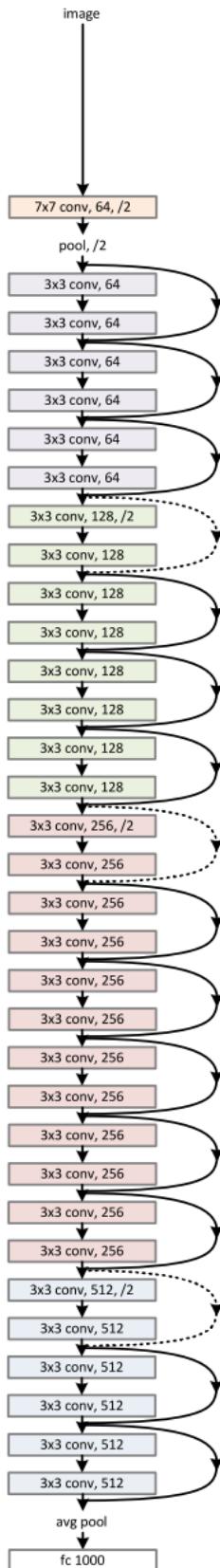
W 2015 roku konkurs *ILSVRC* wygrał model konwolucyjny o niespotykanej dotąd głębokości 152 warstw - był to model zaprojektowany przez pracowników *Microsoft Research Asia* o nazwie **ResNet**. Na zbiorze *ImageNet* osiągnął on fenomenalną stopę błędów top-5 równą 3,57%, uzyskując tym samym nadludzką skuteczność (dla przeciętnego człowieka stopa błędów na tym zbiorze wynosi około 5%). Autorami tego sukcesu byli He K., Zhang X., Ren S. i Sun J., którzy w swoim artykule *Deep Residual Learning for Image Recognition* z 2015 roku [11] zaprezentowali efektywną metodę radzenia sobie z problemem zanikającego gradientu w bardzo głębokich sieciach neuronowych. Rozwiązaniem zaproponowanym przez pracowników *Microsoft Research Asia* był blok rezydualny (ang. *residual block*), który był realizowany przy pomocy połączeń rezydualnych (ang. *residual (skip) connections*) zakończonych operacją sumowania. Połączenia te, pomijając niektóre warstwy konwolucyjno-aktywacyjne, przekazywały gradient błędu bezpośrednio do wyższych warstw sieci w ten sposób zachowując istotny wpływ na zmienność ich wag (patrz rysunek 2.16). Zwycięska sieć *ResNet-152*<sup>7</sup> miała około 60 milionów trenowalnych parametrów, czyli o ponad połowę mniej niż sieć *VGGNet*, ale za to kilkakrotnie razy więcej niż sieć *GoogleNet*.



Rysunek 2.16. Blok rezydualny zaprezentowany w 2015 roku przez He K., Zhang X., Ren S. i Sun J. [11]

Sieć *ResNet-152* doczekała się licznych modyfikacji w kolejnych latach, w których to była ona uzupełniana między innymi o: bloki wzorowane na blokach *Inception* sieci *GoogleNet* (*Inception-v4* [2016], *ResNext* [2017], *Wide ResNet* [2017]), większą liczbę połączeń rezydualnych (*DenseNet* [2016]) czy po prosu większą liczbę warstw (*ResNet-1202* [2015], *ResNet-164* [2016]).

<sup>7</sup> Liczba przy nazwie sieci *ResNet* wskazuje z ilu warstw dana sieć rezydualna się składa.

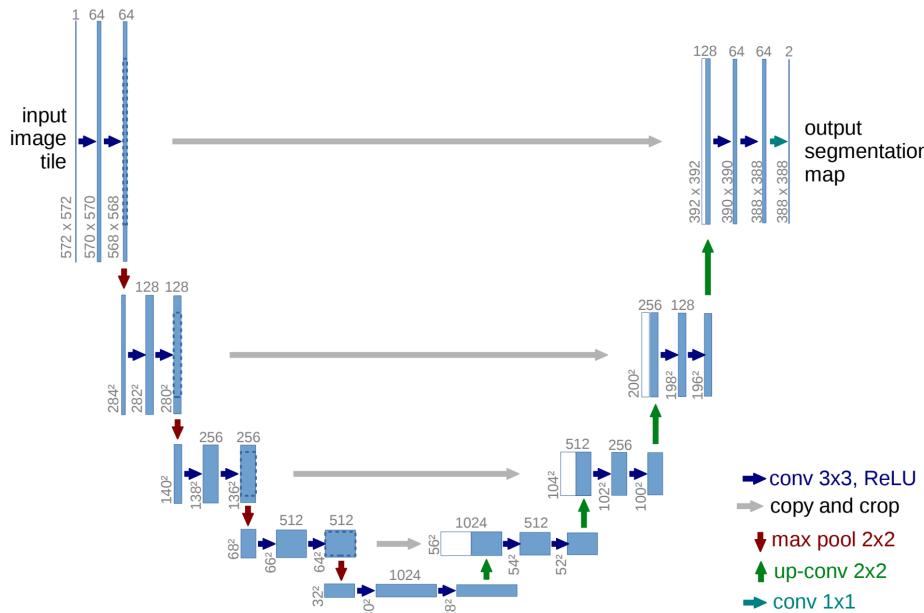


Rysunek 2.17. Przykładowa architektura sieci *ResNet-34* zaprezentowana w 2015 roku przez He K., Zhang X., Ren S. i Sun J. [11]

## U-Net

W tym samym roku co sieć *ResNet* zaprezentowana została inna konwolucyjna sieć neuronowa o bezprecedensowej dotąd architekturze - w kształcie litery „U”. Sieć ta zyskała nazwę ***U-Net*** i została szczegółowo opisana przez Olafa Ronnebergera, Philippa Fischera i Thomasa Broxa w artykule *U-Net: Convolutional Networks for Biomedical Image Segmentation* [25]. Sieć *U-Net* składa się z dwóch części:

- enkodera, w ramach którego przy pomocy warstw splotowo-grupujących następuje ekstrakcja cech wejściowego obrazu na różnych poziomach rozdzielczości przestrzennej (ścieżka analizy),
- dekodera, w ramach którego przy pomocy nadpróbkowania (ang. *upsampling*) następuje propagacja informacji kontekstowej do warstw o wyższej rozdzielczości przestrzennej (ścieżka syntezy),



**Rysunek 2.18.** Architektura sieci *U-Net* zaprezentowana w 2015 roku przez Olafa Ronnebergera, Philippa Fischera i Thomasa Broxa [25]

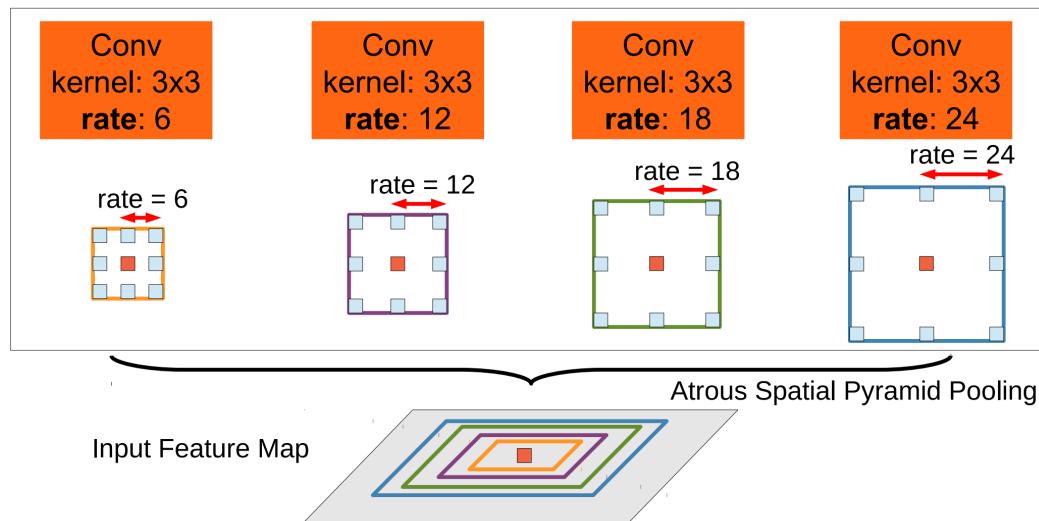
Pomiędzy każdą z warstw enkodera i dekodera, w architekturze *U-Net* występują bezpośrednie połączenia rezydualne<sup>8</sup>, które transferują informacje z niskich warstw enkodera do wysokich warstw dekodera, dzięki którym możliwa jest szczegółowa rekonstrukcja obrazu na kolejnych poziomach rozdzielczości przestrzennej. Sieć *U-Net* można więc zdefiniować jako transformację w pełni konwolucyjnej sieci neuronowej (ang. *Fully Convolutional Network*), czyli konwolucyjnej sieci neuronowej niezawierającej warstw gęstych, uzupełnioną o bezpośrednie połączenie pomiędzy ścieżką analizy a syntezy. Sieć *U-Net*

<sup>8</sup> W odróżnieniu od sieci *ResNet*, połączenia rezydualne w sieci *U-Net* były finalizowane przy pomocy operacji konkatenacji z warstwami dekodera (jak w blokach *Inception*), a nie operacji sumowania (jak w blokach rezydualnych)

była początkowo wykorzystywana przede wszystkim do semantycznej segmentacji (ang. *semantic segmentation*) pikseli na zdjęciach biomedycznych, co oznacza iż jej zadaniem było przypisanie każdemu pikselowi ze zdjęcia wejściowego odpowiedniej klasie, tak aby uzyskać zdjęcie podzielone na spójne regiony odpowiadające poszczególnym klasom. Sieć ta była szczególnie efektywna w takich zastosowaniach, gdyż nie wymagała dużej liczby danych treningowych, brakujące dane treningowe można było z powodzeniem zastąpić augmentacją już dostępnych zdjęć.

### DeepLab

Kolejną istotną siecią neuronową szeroko stosowaną do semantycznej segmentacji obrazów, która również została zaprezentowana w 2015 roku, była sieć **DeepLab** (znana obecnie pod nazwą *DeepLabv1*). Została ona po raz pierwszy opisana w artykule *Semantic image segmentation with deep convolutional nets and fully connected CRFs* [4], którego autorami byli Chen L., Papandreou G., Kokkinos I., Murphy K. oraz Yuille A. L. Sieć *DeepLabv1* była zwykłą konwolucyjną siecią neuronową, podobnie jak *U-Net* składającą się z enkodera i dekodera, jednak jej wyróżnikiem było zastosowanie rozszerzonych konwolucji (ang. *dilated convolution / atrous convolution*)<sup>9</sup>. Umożliwiają one łatwiejszą kontrolę rozdzielczości map cech oraz zapewniają większy obszar „widzenia” filtrów, co pozwala uchwycić większy zakres informacji kontekstowej. Dzięki zastosowaniu rozszerzonych konwolucji, sieć *DeepLabv1* posiadała mniej trenowalnych parametrów niż inne sieci służące do semantycznej segmentacji takie jak *U-Net* czy *FCN* osiągając przy tym często lepsze wyniki.



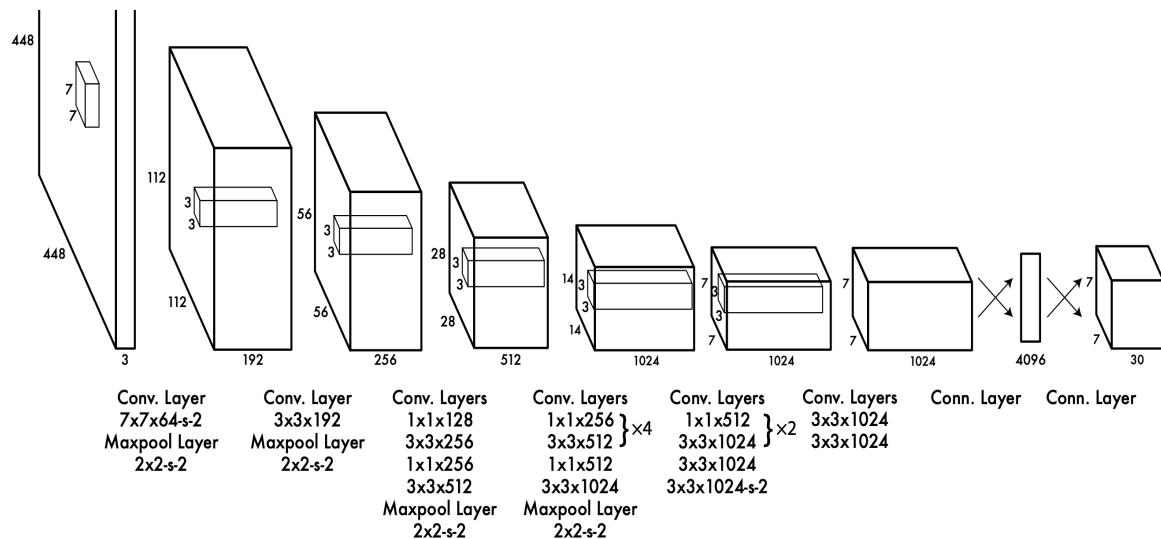
**Rysunek 2.19.** Architektura bloku ASSP zaprezentowana w 2017 roku przez Chen L., Papandreou G., Kokkinos I., Murphy K. oraz Yuille A. L. [3]

<sup>9</sup> Rozszerzone konwolucje to operacje splotowe, w których filtr nie jest nakładany na sąsiadujące ze sobą piksele tylko na piksele oddalone od siebie o pewną stałą odległość.

W 2017 roku autorzy sieci *DeepLabv1* zaproponowali duże ulepszenie tej sieci, w swoim artykule *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs* [3] opisali sieć *DeepLabv2* wykorzystującą blok o nazwie ASPP (Atrous Spatial Pyramid Pooling), który zapewniał efektywną segmentację obiektów o różnych skalach, poprzez zastosowanie rozszerzonych konwolucji z filtrami o różnych częstotliwościach próbkowania i różnych efektywnych polach widzenia. Kolejne ulepszenia sieci *DeepLab*, o nazwach *DeepLabv3* [2017] i *DeepLabv3+* [2018], nie niosły ze sobą przełomowych rozwiązań, wprowadzały na przykład zmianę enkodera z sieci *ResNet-50*, na sieć *ResNet-101*, czy dodawały konwolucje separowalne wgłębnie.

### You Only Look Once (YOLO)

W 2016 roku zaprezentowany został przełomowy model w dziedzinie rozpoznawania obiektów w czasie rzeczywistym - **YOLO**. Model ten przewyższał wszystkie wcześniejsze modele nie tylko pod kątem precyzji, ale także szybkości przeliczeń. Jego autorami byli Joseph Redmon, Santosh Divvala, Ross Girshick i Ali Farhadi, którzy w swoim artykule *You Only Look Once: Unified, Real-Time Object Detection* [24] zaprezentowali konwolucyjną sieć neuronową, która jednocześnie prognozowała lokalizację obiektu (w postaci ramki (ang. *bounding box*)) oraz prawdopodobieństwo klas do jakich dany obiekt według modelu należy. Model YOLO, był w stanie w ciągu jednej sekundy rozpoznać obiekty na 45 zdjęciach (45 klatkach filmu), osiągając średnią precyzję (ang. *mean average precision*) na poziomie 63%.



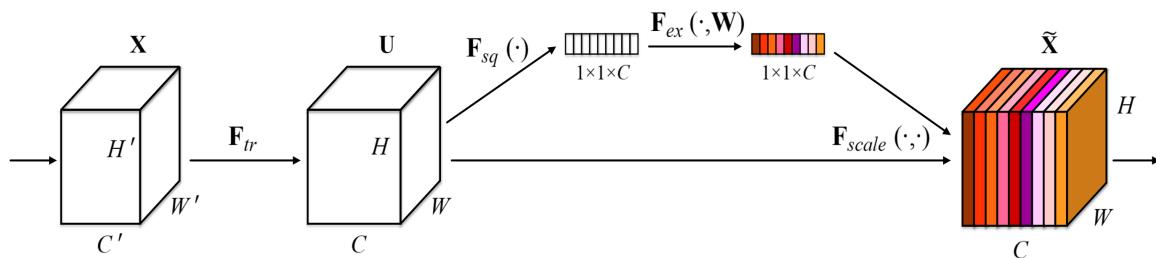
**Rysunek 2.20.** Architektura sieci YOLO zaprezentowana w 2016 roku przez Josepha Redmona, Santosha Divvalsa, Rossa Girshicka i Aliego Farhadiego [24]

Sieć stworzona przez Redmona, Divvalsa, Girshicka i Farhadiego składała się z 24 warstw konwolucyjnych zakończonych dwoma warstwami gęstymi (patrz rysunek 2.20). Tworząc ją autorzy zainspirowali się siecią *GoogleNet*, przy czym bloki *inception* zostały zastąpione

przez nich redukcyjnymi warstwami konwolucyjnymi o rozmiarze filtra 1 x 1, po których następowały warstwy z filtrami 3 x 3. Sieć YOLO miała jednak kilka wad: nie była w stanie efektywnie rozpoznawać bardzo małych obiektów, nie umiała efektywnie generalizować obiektów ze względu na rozmiar (jej skuteczność zależała od rozmiarów obiektów w zbiorze treningowym) oraz miała trudności z poprawną detekcją obiektów, które znajdowały się bardzo blisko siebie. Wszystkie te problemy zostały rozwiązane w kolejnych wersjach modelu: YOLO v2 [2016], YOLO v3 [2018] oraz YOLO v4 [2020]. Implementacje te nie tylko rozwiązały większość problemów oryginalnej architektury, ale także poprawiły statystki pod kątem precyzji i szybkości detekcji.

### Squeeze-and-Excitation Network

W 2017 roku konkurs *ImageNet Large Scale Visual Recognition Challenge* wygrała sieć *Squeeze-and-Excitation Network (SENet)* uzyskując stopę błędów top-5 na poziomie 2,251% (co było o 25% lepszym wynikiem niż najlepszy wynik z roku 2016<sup>10</sup>). Autorami zwycięskiej sieci byli Hu J., Shen L., Sun G., którzy opisali ją w swoim artykule pod tytułem *Squeeze-and-Excitation Networks* [15]. Ich celem było stworzenie sieci, która położy większy nacisk na wychwycenie relacji pomiędzy kanałami obrazu wejściowego, w miejsce relacji przestrzennych, na których skupiała się większość tworzonych w tamtym czasie modeli. Blok *Squeeze-and-Excitation* zastosowany w sieci SENet kładzie nacisk na adaptacyjne dostosowanie wag poszczególnych kanałów, żeby odzwierciedlić istotność każdego z nich. Jest to osiągane w następującym procesie: warstwa grupująca do średniej (ang. *average pooling*) kompresuje wszystkie kanały przestrzenne do pojedynczych wartości. Wartości te następnie uzyskują nieliniowość przez zastosowanie warstwy gęstej i funkcji aktywacji *ReLU* oraz są wygładzane przy pomocy drugiej warstwy gęstej i funkcji aktywacji *Sigmoid*. Tak przetworzone przez blok SE wartości służą jako finalne wagi dostosowujące istotność poszczególnych kanałów przestrzennych.



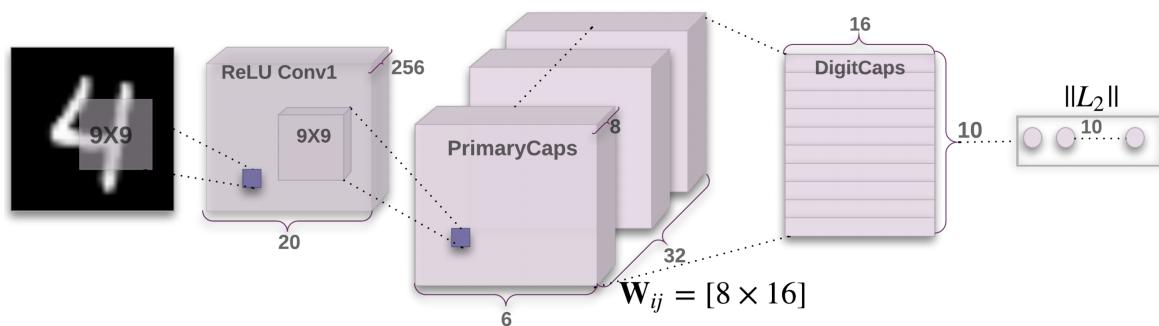
**Rysunek 2.21.** Architektura bloku *Squeeze-and-Excitation* zaprezentowana w 2017 roku przez Hu J., Shen L. i Sun G.[15]

<sup>10</sup> Zwycięski model z 2016 roku o nazwie *Trimp-Soushen* nie został opisany w bieżący podródziale, gdyż jego autorzy nie opublikowali żadnego artykułu czy raportu technicznego przedstawiającego jego architekturę.

Prostota bloków *SE* sprawia, iż mogą one być z łatwością implementowane w innych wiodących sieciach neuronowych. Jako przykład takiej implementacji autorzy sieci *SENet* wskazywali sieć *ResNet-50*, która uzupełniona o bloki *Squeeze-and-Excitation*, uzyskuje skuteczność zbliżoną do skuteczności sieci *ResNet-101*, nie zwiększaając istotnie liczby trenowalnych parametrów oraz kosztów obliczeniowych. Rok 2017 był ostatnim rokiem organizacji konkursu *ILSVRC* - został on zakończony, gdyż rezultaty osiągane na zbiorze testowym przez większość zgłaszanych do konkursu modeli zaczęły istotnie przekraczać wyniki przeciętnego człowieka.

### Capsule Neural Network (CapsNet)

Kolejną istotną architekturą głębokich sieci neuronowych, która zostanie zaprezentowana w bieżącym podrozdziale, jest kapsułowa sieć neuronowa (ang. *capsule neural network*), która oznaczana jest skrótem ***CapsNet***. Została ona zaprezentowana w 2017 roku przez Geoffrey'a Hintona, Sarę Sabour i Nicholasa Frossta w artykule *Dynamic Routing Between Capsules* [13].

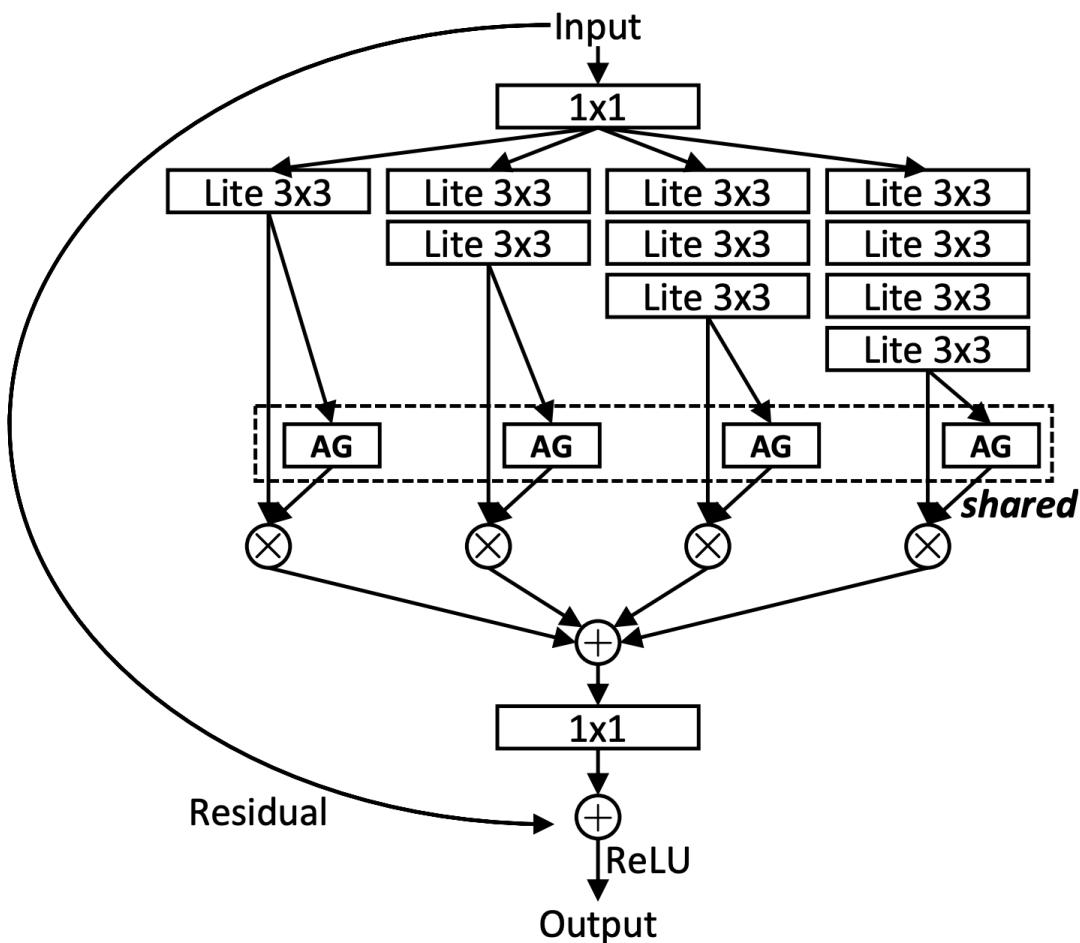


Rysunek 2.22. Architektura sieci *CapsNet* zaprezentowana w 2017 roku przez Geoffrey'a Hintona, Sarę Sabour i Nicholasa Frossta [13]

Autorzy omawianego artykułu zaproponowali dodanie do tradycyjny konwolucyjnych sieci neuronowych bloków, które nazwali kapsułami, po to by te ekstraktowały z obrazów informacje przestrzenne (informacje dotyczące pozycji) oraz prawdopodobieństwo występowania określonych obiektów na tych obrazach. Kapsuły te następnie przekazują te informacje do odpowiednich kapsuł umieszczonych powyżej nich, w ramach procesu dynamicznego trasowania (ang. *dynamic routing*), a następnie dochodzi do zwiększenia współczynnika sprzężenia (ang. *coupling coefficient*) dla kapsuł, których wyniki najsilniej rezonują z wynikami kapsuł z poziomu niżej oraz osłabienia tego współczynnika dla pozostałych kapsuł. W ten sposób sieć *CapsNet* adresuje problem obrazów, które składają się z odpowiednich części, ale części te nie zachowują odpowiednich relacji przestrzennych pomiędzy sobą oraz jest w stanie lepiej rozpoznawać obiekty, na zdjęciach robionych z bardzo różnych kątów widzenia.

### Omni-Scale Network

W 2019 roku naukowcy związani z *Samsung AI Center*, w swojej pracy *Omni-Scale Feature Learning for Person Re-Identification* [36], zaproponowali nowy rodzaj bloku rezydualnego, wykorzystywanego w ramach sieci *Omni-Scale Network (OSNet)*, o nazwie *Bottleneck*. Blok ten był złożony z kilku niezależnych ścieżek konwolucji, z których każda wychwytywała cechy o innej skali. Ścieżki te były łączone przy pomocy nowatorskich bramek agregacji, które dynamicznie scalają cechy o różnych skalach z wagami zależnymi od kanałów wejścia (patrz rysunek 2.23)



**Rysunek 2.23.** Architektura bloku *Bottleneck* zaprezentowana w 2019 roku przez Zhou K., Yang Y., Cavallaro A. i Xiang T. [36]

Sieć *OSNet* została wykorzystana przez Zhou K., Yang Y., Cavallaro A. i Xiang T. w zadaniach związanych z ponownym rozpoznawaniem osób na zdjęciach z publicznego monitoringu i uzyskała bardzo dobre wyniki, pomimo relatywnie niewielkiej liczby trenowań parametrów, które ta sieć posiadała. Wydaje się, iż bloki *Bottleneck* zaimplementowane w ramach sieci *OSNet* mogłyby stanowić dobre uzupełnienie architektury służącej do rozpoznawania budynków na zdjęciach lotniczych, gdyż mogłyby one pomóc uzyskać odporność modelu na skalę w jakiej prezentowane są budynki na tego typu zdjęciach.

### 2.3. Proces uczenia się głębokich sieci neuronowych

Uczenie (trenowanie) głębokich sieci neuronowych jest najważniejszą częścią procesu ich projektowania, gdyż od jego efektywności zależy jak dobre predykcje uzyskamy z opracowanego przez nas modelu. Może ono przyjmować kilka podstawowych form, do których najczęściej zaliczamy:

- uczenie nadzorowane (ang. *supervised learning*) - uczenie, w którym wszystkie dane w zbiorze uczącym mają odpowiadające im etykiety (ang. *labels*), więc sieć neuronowa uczy się starając się wygenerować predykcje jak najbliższe dostępnym etykietom,
- uczenie nienadzorowane (ang. *unsupervised learning*) - uczenie, w ramach którego dane treningowe nie mają etykiet (są nieoznaczone), więc sieć neuronowa stara się nauczyć wewnętrznej struktury danych treningowych,
- uczenie pół-nadzorowane (ang. *semi-supervised learning*) - uczenie, w którym większość danych treningowych nie posiada etykiet, ma je tylko część danych, stanowi ono miks uczenia nadzorowanego i nienadzorowanego,
- uczenie ze wzmacnieniem (ang. *reinforcement learning*) - uczenie, w ramach którego oszacowany błąd służy do nakładania kar lub dawania nagród sieci neuronowej, więc sieć uczy się metodą prób i błędów po to by zmaksymalizować zyski i zminimalizować kary.

W bieżącym podrozdziale skupimy się na uczeniu nadzorowanym, gdyż to właśnie uczenie zostało wykorzystane do wytrenowania opracowanej na potrzeby niniejszej pracy sieci neuronowej. Proces nadzorowanego uczenia się głębokich sieci neuronowych możemy podzielić na dwie fazy: przechodzenie danych treningowych przez sieć neuronową w przód (ang. *forward propagation*) oraz przechodzenie informacji o wartości funkcji strat w tył (ang. *backward propagation*). W czasie fazy przechodzenia w przód dane treningowe są przetwarzane, warstwa po warstwie, w taki sposób, że wszystkie neurony nakładają swoje transformacje na informacje, którą uzyskują z poprzednich neuronów i tak przetworzona informacja jest następnie przekazywana dalej w głąb sieci. Po przejściu przez całą sieć neuronową, ostatnia jej warstwa generuje predykcję dla konkretnego przykładu uczącego. Następnie taka predykcja, wraz z prawidłową etykietą trafiają do, zdefiniowanej dla danej sieci, funkcji straty (ang. *loss function*), która wylicza błąd tej predykcji na podstawie odpowiedniej miary podobieństwa prognozowanej przez sieć etykiety oraz rzeczywistej etykiety. Tak wyliczona wartość błędu jest następnie wstecznie propagowana do wszystkich neuronów sieci, zaczynając od neuronów znajdujących się w warstwie końcowej aż do neuronów z warstw początkowych. Każdy z neuronów otrzymuje tylko fragment łącznej wartości funkcji strat, proporcjonalnie do tego jaki wkład miał konkretny

neuron do wygenerowania predykcji etykiety. Po rozpropagowaniu informacji o finalnej wartości funkcji strat po całej sieci, rozpoczyna się proces dostosowywania wag połączeń pomiędzy neuronami w taki sposób, żeby jak najbardziej zminimalizować wartość funkcji strat. Proces ten nosi nazwę spadku wzduż gradientu (ang. *gradient descent*) i polega na zmianie wag sieci o małe wartości po to by sprawdzić jak te zmiany wpłyną na finalną wartość funkcji strat - w którym kierunku powinny one zachodzić, żeby zbliżyć się do globalnego minimum funkcji strat.

Proces uczenia się sieci neuronowej jest dokonywany zazwyczaj na danych treningowych połączonych w partie, które przepuszczane są przez model patria po partii w następujących po sobie iteracjach (epokach). Stąd też **rozmiar partii** (ang. *batch size*) oraz **liczba epok** (ang. *number of epochs*) to jedne z najważniejszych hiperparametrów sieci neuronowej i od ich odpowiedniego wyboru często zależy finalna skuteczność naszego modelu. Zanim przystąpi się do trenowania głębokiej sieci neuronowej łączny zbiór danych należy podzielić na trzy rozłączne podzbiory: **zbiór treningowy** (ang. *training set*), **zbiór walidacyjny** (ang. *validation set*) oraz **zbiór testowy** (ang. *test set*). Proporcje podziału łącznego zbioru danych na te trzy zbiory zależą wyłącznie od autora danego modelu, jednak bardzo często w literaturze przyjmuje się następujący podział (kolejno): 70%, 15% i 15%. Sieć neuronowa uczy się korzystając wyłącznie ze zbioru treningowego. Zbiór walidacyjny służy do monitorowania wyników sieci na danych z poza zbioru treningowego oraz do odpowiedniego zmieniania hiperparametrów modelu w taki sposób, aby uzyskiwać coraz lepsze wyniki. Wreszcie zbiór testowy pozwala na ocenę skuteczności predykcji finalnego modelu na niezależnym zbiorze danych bez obawy o obciążenie takiej oceny przez odpowiedni dobór hiperparametrów.

Poza właściwym podziałem zbioru danych, kolejnym istotnym elementem procesu przygotowania modelu do głębokiego uczenia jest dobór odpowiedniej **funkcji straty**. Funkcja ta wpływa bezpośrednio na efektywność treningu głębokich sieci neuronowych, stąd też na przestrzeni lat zaproponowanych zostało bardzo wiele różnych rodzajów funkcji strat, które były mniej lub bardziej efektywne w zależności od tego do jakiego zadania i w ramach jakiej architektury były wykorzystywane. Do najpopularniejszych z nich można zaliczyć:

- **błąd średniokwadratowy** (ang. *Mean Square Error / Quadratic Loss / L2 Loss*) - jedna z najprostszych funkcji strat, która mierzy średnią różnicę podniesioną do kwadratu pomiędzy predykcjami oszacowanymi przez sieć ( $y_i$ ) a oczekiwaniymi wartościami ( $\hat{y}_i$ ):

$$MSE = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n} \quad (8)$$

- średni błąd absolutny (ang. *Mean Absolute Error / L1 Loss*) - funkcja ta mierzy średnią absolutną różnicę pomiędzy predykcjami oszacowanymi przez sieć ( $y_i$ ) a oczekiwanyimi wartościami ( $\hat{y}_i$ ):

$$MAE = \frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{n} \quad (9)$$

- entropia skrośna (ang. *Cross-Entropy / Log Loss*) - mierzy ona różnicę pomiędzy dwoma rozkładami prawdopodobieństwa dla danych zmiennych losowych, jej szczególnym przypadkiem jest binarna entropia skrośna (ang. *Binary Cross-Entropy*), która jest uproszczeniem entropii skrośnej dla dwóch klas:

$$CE = \sum_{i=1}^n p_i \cdot \log \frac{1}{q_i} \quad (10)$$

$$BCE = -(p \cdot \log q + (1-p) \cdot \log(1-q)) \quad (11)$$

W bieżącej pracy wykorzystane zostaną jeszcze dwie inne, mniej popularne funkcje straty, ale znajdujące swoje zastosowanie w semantycznej segmentacji obrazów:

- Dice Loss* - funkcja straty wywodząca się od współczynnika podobieństwa Sørensen, której zadaniem jest porównanie podobieństwa dwóch próbek, gdzie  $p_i$  i  $g_i$  reprezentują pary odpowiadających sobie danych z predykcji uzyskanej z modelu i ze zbioru oczekiwanych wartości:

$$DL = 1 - \frac{2 \cdot \sum_i^N p_i \cdot q_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2} \quad (12)$$

- Lovász hinge loss* - funkcja straty, która jest gładkim rozszerzeniem współczynnika podobieństwa Jaccarda (znanego też pod nazwą *Intersection over Union*) - została ona szeroko omówiona w pracy *The Lovasz Hinge: A Novel Convex Surrogate for Submodular Losses* napisanej przez Jiaqiana Yu i Matthew B. Blaschko [34].

Duży wpływ na efektywność głębokiego uczenia ma również wybór właściwego **optymalizatora** (ang. *optimizer*), czyli algorytmu poszukującego globalnego minimum funkcji strat. Większość obecnych optymalizatorów stosuje metodę spadku wzdłuż gradientu, jednak różnią się one sposobem aktualizacji wag w kolejnych etapach uczenia, część optymalizatorów liczy gradient błędu dla pojedynczych danych treningowych, część dla całych partii, niektóre z nich adaptują stopę uczenia do właściwości wag. Do najczęściej używanych optymalizatorów należą:

## 2. Podstawy teoretyczne głębokiego uczenia

---

- *Batch Gradient Descent* - najbardziej podstawowy optymalizator, który liczy gradient funkcji strat i aktualizuje wagi sieci neuronowej dla całego zbioru treningowego co zapewnia względnie stabilną zbieżność do globalnego minimum, kosztem efektywności obliczeniowej,
- *Stochastic Gradient Descent* - optymalizator, który liczy gradient funkcji strat i aktualizuje wagi sieci neuronowej dla każdego przykładu uczącego się osobno, stąd też jest dużo bardziej efektywny obliczeniowo od optymalizatora *BGD*, kosztem występowania dużej zmienności przy poszukiwaniu globalnego minimum,
- *Mini-Batch Gradient Descent* - optymalizator, który łączy w sobie zalety *BGD* i *SGD*, czyli efektywność obliczeniową i względnie stabilną zbieżność do globalnego minimum - proces liczenia gradientu i aktualizacji wag w *MBGD* jest przeprowadzany na przypadkach uczących połączony w niewielkich rozmiarów partie,
- *SGD z momentum* - optymalizator, który łączy w sobie zalety *MBGD* oraz rozwiązuje najważniejszy problem wcześniej przedstawionych optymalizatorów, czyli fakt, iż mają one tendencję do zatrzymywania się w punkcie siodłowym płaszczyzny gradientu, *SGD z momentum* rozwiązuje ten problem poprzez akumulowanie przeszłych gradientów w celu wyliczenie wartości obecnego gradientu,
- *AdaGrad* - optymalizator, który dostosowuje stopę uczenia się do częstości występowania cech, stosuje on mniejszą stopę uczenia dla wag cech często występujących i większą stopę uczenia dla wag cech rzadziej występujących, w ten sposób wszystkie wagi konvergują w podobnym tempie,
- *RMSprop* - optymalizator, który adresuje problem różnic w wielkościach gradientów dla poszczególnymi wag modelu, poprzez zastosowanie indywidualnych kroków uczenia dla każdej wagi w sieci neuronowej,
- *Adam (Adaptive Moment Estimation)* - najbardziej popularny optymalizator szeroko wykorzystywany do trenowania głębokich sieci neuronowych łączący w sobie najważniejsze zalety optymalizatorów *SGD z momentum* (akumulowanie przeszłych wykładniczo malejących średnich gradientów) oraz *RMSprop* (indywidualne kroki uczenia dla poszczególnych wag modelu),

Wraz z wyborem optymalizatora, przed przystąpieniem do trenowania modelu, dobrze jest zdefiniować związane z nim hiperparametry głębokiego uczenia m.in.: **stopę uczenia** (ang. *learning rate*), **momentum** (ang. *momentum*), **spadek wagi** (ang. *weight decay*), **spadek stopy uczenia** (ang. *learning rate decay*). Chociaż jakość wytrenowania modelu ocenia się zazwyczaj przez pryzmat wartości funkcji straty to często w czasie trenowania wylicza się też dodatkowe metryki, które mają za zadanie uchwycić inne istotne właściwości modelu, a które mogą służyć do porównywania pomiędzy sobą modeli o bardzo

różnych architekturach i funkcjach strat. W bieżącej pracy zdefiniowano następujące dodatkowe metryki, które posługują w dalszej części pracy do oceny jakości modelu:

- ogólna dokładność (ang. *Overall Accuracy*) - najprostsza miara oceny predykcji modelu definiowana jako:

$$OA = \frac{\text{Liczba poprawnych predykcji}}{\text{Łączna liczba wszystkich predykcji}} = \frac{TP + TN}{TP + FP + TN + FN} \quad (13)$$

gdzie:

TP - liczba wszystkich poprawnie zakwalifikowanych predykcji dodatnich,  
 FP - liczba wszystkich niepoprawnie zakwalifikowanych predykcji dodatnich,  
 TN - liczba wszystkich poprawnie zakwalifikowanych predykcji ujemnych,  
 FN - liczba wszystkich niepoprawnie zakwalifikowanych predykcji ujemnych.

- wynik F1 (ang. *F1 Score*) - miara precyzji definiowana w zadaniach klasyfikacji binarnej jako średnia harmoniczna precyzji (ang. *precision*) i czułości (ang. *recall*):

$$F1S = \frac{2 \cdot \text{Precyzja} \cdot \text{Czułość}}{\text{Precyzja} + \text{Czułość}} \quad (14)$$

$$\text{Precyzja} = \frac{TP}{TP + FP} \quad (15)$$

$$\text{Czułość} = \frac{TP}{TP + FN} \quad (16)$$

- współczynnik podobieństwa Jaccarda (ang. *Jaccard Similarity Coefficient / Intersection over Union*) - miara podobieństwa pomiędzy dwoma zbiorami zdefiniowana jako iloraz mocy części wspólnej tych zbiorów i mocy sumy tych zbiorów:

$$IoU = \frac{|A \cap B|}{|A \cup B|} \quad (17)$$

- indeks podobieństwa strukturalnego (ang. *Structural Similarity Index Measure*) - miara strukturalnego podobieństwa, które występuje pomiędzy dwoma obrazami  $X$  i  $Y$ :

$$SSIM(X, Y) = \frac{(2\mu_X\mu_Y + c_1) \cdot (2\sigma_{XY} + c_2)}{(\mu_X^2 + \mu_Y^2 + c_1) \cdot (\sigma_X^2 + \sigma_Y^2 + c_2)} \quad (18)$$

## 2. Podstawy teoretyczne głębokiego uczenia

gdzie:

$\mu_X$  - średnia z  $X$ ,

$\mu_Y$  - średnia z  $Y$ ,

$\sigma_X^2$  - wariancja  $X$ ,

$\sigma_Y^2$  - wariancja  $Y$ ,

$\sigma_{XY}$  - kowariancja  $X$  i  $Y$ ,

$c_1 = (0,01 \cdot L)^2$  - zmienna stabilizująca dzielenie ze słabym mianownikiem,

$c_2 = (0,03 \cdot L)^2$  - zmienna stabilizująca dzielenie ze słabym mianownikiem,

$L$  - rozpiętość tonalna wartości pikseli.

W bieżącym rozdziale zarysowane zostały podstawy teoretyczne głębokiego uczenia, których znajomość jest niezbędna do efektywnej analizy dalszych rozdziałów niniejszej pracy. W kolejnym rozdziale przedstawione zostaną najważniejsze artykuły naukowe związane z detekcją budynków na zdjęciach lotniczych i satelitarnych, tak by czytelnik mógł się zapoznać ze specyfiką problemu, poznać najważniejsze wyzwania w tej dziedzinie oraz zglebić obecnie proponowane, przez świat nauki i biznesu, rozwiązania.

### **3. Przegląd literatury z zakresu detekcji budynków na zdjęciach lotniczych**

W bieżącym rozdziale zaprezentowane zostaną wybrane badania naukowe z zakresu detekcji budynków na zdjęciach lotniczych i satelitarnych. Najważniejszym kluczem doboru artykułów badawczych była zbieżność tematyczna z badaniem własnym, w dalszej kolejności liczyła się deklarowana skuteczność opisywanych architektur oraz ich różnorodność tak by uzyskać jak najszerzy przegląd dostępnych metod oraz stosowanych hiperparametrów. W ten sposób wyselekcjonowano siedem najbardziej istotnych artykułów z dziedziny semantycznej segmentacji budynków na zdjęciach lotniczych i satelitarnych. Artykuły te zostaną przeanalizowane w bieżącym rozdziale w kolejności zgodnej z datami ich powstania. Wnioski z tak przeprowadzonej analizy stanowić będą nieocenioną pomoc przy doborze właściwych hiperparametrów modelu tworzonego w ramach badania własnego, który zostanie zaprezentowany w kolejnym rozdziale bieżącej pracy.

Pierwszym analizowanym artykułem jest *Satellite Imagery Feature Detection using Deep Convolutional Neural Network: A Kaggle Competition* [16]. Jego autorami są Iglovikov V., Mushinskiy S. i Osin V., został on przez nich opublikowany w czerwcu 2017 roku. W artykule tym autorzy opisują stworzoną przez siebie sieć neuronową, którą zgłosili do konkursu *DSTL Satellite Imagery Feature Detection challenge*. Celem tego konkursu było stworzenie modelu pozwalającego na efektywną detekcję dziesięciu różnych klas obiektów (w tym budynków) na zdjęciach satelitarnych, o rozdzielczości  $3600 \times 3600$  pikseli i dziewięciu kanałach przestrzennych. Rozwiązaniem zaproponowanym przez Iglovikova, Mushinskiy'ego. i Osina była sieć o architekturze *U-Net* (z funkcją aktywacji *ELU*), trenowana przy pomocy funkcji straty zdefiniowanej jako różnica pomiędzy binarną entropią skrośną a indeksem Jaccarda przez optymalizator *Nadam* o startowej stopie uczenia równej 0,001 i momentum 0,9. Z każdego zdjęcia satelitarnego, autorzy omawianego artykułu, wybierali losowo 128 fragmentów o rozmiarach  $112 \times 112$  pikseli, fragmenty te były następnie łączone w partie po 400 na każdą epokę (trening trwał łącznie 100 epok). W rezultacie autorzy uzyskali skuteczność w detekcji budynków na poziomie **62,9% IoU**.

Kolejnym artykułem badawczym, który zostanie omówiony w bieżącym rozdziale, jest *Multi-Task Learning for Segmentation of Building Footprints with Deep Neural Networks* citebischke opublikowany we wrześniu 2017 roku. Jego autorami są Bischke B., Helber P., Folz J., Borth D. oraz Dengel A., którzy pisząc go starali się zaadresować problem słabej jakości predykcji granic budynków przez sieci neuronowe, a ich rozwiązaniem tego problemu było zastosowanie nowatorskiej funkcji strat o nazwie *Uncertainty Based Multi-Task Loss*. Funkcja ta łączyła w sobie nie tylko część semantyczną, ale też geometryczną, która miała dbać o uzyskanie z sieci predykcji maski o odpowiednim, pożdanym kształcie. Jako zbiór danych autorzy omawianego artykułu wybrali *Inria Aerial Image Labeling Dataset*,

### 3. Przegląd literatury z zakresu detekcji budynków na zdjęciach lotniczych

złożony ze 180 zdjęć lotniczych pięciu miast: Austin, Chicago, Hrabstwo Kitsap, Tyrol oraz Wiedeń (po 36 zdjęć każdego z miast). Zdjęcia te miały rozdzielcość  $5000 \times 5000$  pikseli, trzy kanały przestrzenne, każde z nich pokrywało teren o powierzchni  $1500 \times 1500 \text{ m}^2$ . Zbiór walidacyjny został wydzielony ze zbioru treningowego poprzez wzięcie pięciu pierwszych zdjęć każdego z miast. Każde zdjęcie lotnicze, autorzy omawianego w tym akapicie artykułu, dzielili na 10 części, z każdej z tych części wybierali oni po cztery losowe fragmenty o rozmiarach  $384 \times 384$  pikseli. Fragmenty te były następnie poddawane augmentacji w postaci losowego odwrócenia w poziomie lub pionie, a następnie były łączone w partie po dziesięć sztuk i w takiej formie zasilały finalną sieć neuronową. Sieć zaproponowana przez Bischke, Helbera, Folza, Bortha i Dengela miała architekturę *SegNet* z *VGG-16* jako enkoderem, była trenowana przy pomocy optymalizatora *SGD* z początkową stopą uczenia równą 0,01, spadkiem wagi 0,0005 i momentum 0,9. Tak skonstruowana sieć uzyskała skuteczność na poziomie **95,17% OA** oraz **70,14% IoU**.

W artykule *Automatic Building Segmentation of Aerial Imagery Using Multi-Constraint Fully Convolutional Networks* [33] z marca 2018 roku, autorzy Wu G., Shao X., Guo Z., Chen Q., Yuan W., Shi X., Xu Y. i Shibasaki R. prezentują sieć *MC-FCN*, która jest ulepszeniem sieci *U-Net* o dodanie trzech dodatkowych wieloskalowych połączeń pomiędzy nadpróbkowanymi warstwami a przykładami uczącymi, które są wykorzystywane w czasie uczenia się modelu przy pomocy binarnej entropii skrośnej. Trenowanie tak powstałej sieci neuronowej było przeprowadzane na zbiorze zdjęć lotniczych o wysokiej rozdzielcości wykonanych w Nowej Zelandii dla ponad  $18 \text{ km}^2$  terenu, w tym 17 tysięcy budynków. Każde ze zdjęć lotniczych zostało w pierwszej fazie podzielone na drobne fragmenty o rozdzielcości  $224 \times 224$  pikseli, a fragmenty te następnie posłużyły do zasilenia sieci *MC-FCN* w przykłady uczące. Łącznie trenowanie finalnego modelu trwało 100 epok i zostało ono przeprowadzone przy pomocy optymalizatora *Adam* o startowej stopie uczenia równej 0,001. W wyniku tak przeprowadzonych obliczeń autorzy artykułu *Automatic Building Segmentation of Aerial Imagery Using Multi-Constraint Fully Convolutional Networks* uzyskali sieć neuronową o skuteczności **97,6% OA** oraz **83,3% IoU**.

W czerwcu 2018 roku opublikowany został kolejny artykuł podejmujący problematykę detekcji budynków na zdjęciach satelitarnych *Building Detection from Satellite Imagery Using a Composite Loss Function* [9]. Jego autorami byli Golovanov S., Kurbanov R., Artamonov A., Davydow A. oraz Nikolenko S., w swoim artykule zaproponowali oni sieć neuronową o architekturze *LinkNet*<sup>11</sup> z funkcją aktywacji *ELU*, pretrenowanym enkoderem o architekturze SE-ResNeXt-50 oraz funkcją strat zdefiniowaną jako ważona suma binarnej entropii skrośnej, *Lovasz hinge loss* oraz błędu średniokwadratowego (o wagach odpowiednio: 0,8, 0,2 i 10). Dodatkowo w celu lepszej separacji budynków, autorzy omawianego w bieżącym akapicie artykułu, przy liczeniu wartości *BCE* zastosowali różne wagi dla poszczególnych

<sup>11</sup> LinkNet ma architekturę niemal identyczną jak *U-Net* tylko w miejscu konkatenacji warstw stosowane jest sumowanie.

pikseli, wzmacniając piksele blisko krawędzi budynków i osłabiając piksele wewnętrz budynków. Cały model był trenowany na zbiorze danych o nazwie *SpaceNet* złożonym z trzykanałowych zdjęć satelitarnych takich miast jak Las Vegas, Paryż, Szanghaj czy Chartum. Z każdego zdjęcia wycinane były fragmenty o rozdzielcości 256x256 pikseli, które podlegały standardowym augmentacjom takim jak: rotacje, obroty, losowe skalowania, losowe przesunięcia oraz zmiany w jasności i kontraste. Tak przygotowane fragmenty zdjęć trafiały następnie w partiach po 32 sztuki do sieci neuronowej, która podlegała trenowaniu przez 120 epok przez optymalizator *SGD* z momentum Nestorova równym 0,95 oraz początkową stopą uczenia na poziomie 0,01. Finalnie autorom artykułu *Building Detection from Satellite Imagery Using a Composite Loss Function* udało się uzyskać skuteczność modelu na poziomie **76,79% F1S**.

Kolejnym ważnym artykułem, z dziedziny semantycznej segmentacji budynków na zdjęciach lotniczych, który zostanie omówiony w bieżącym rozdziale, jest artykuł *Semantic Segmentation from Remote Sensor Data and the Exploitation of Latent Learning for Classification of Auxiliary Tasks* [2], którego autorami są Chatterjee B. oraz Poullis Ch. W swoim artykule zaprezentowali oni sieć o nazwie *ICT-Net*, której bazową architekturę stanowiła sieć *U-Net* uzupełniona o bloki *Dense* oraz *Squeeze-and-Excitation*, w celu poprawy jakości predykcji modelu. Tak skonstruowana sieć była trenowana na zbiorze *Inria Aerial Image Labeling Dataset*, który podobnie jak w artykule *Multi-Task Learning for Segmentation of Building Footprints with Deep Neural Networks* został podzielony na zbiór walidacyjny oraz treningowy poprzez przypisanie do zbioru walidacyjnego pięciu pierwszych zdjęć każdego z miast reprezentowanych w tym zbiorze. Zdjęcia lotnicze pobrane ze zbioru *Inria Aerial Image Labeling Dataset* poddawane były, przez autorów omawianego w tym akapicie artykułu, wstępnej augmentacji (obroty / rotacje / skalowanie) a następnie wycinano z nich losowe fragmenty o rozdzielcości 256x256 pikseli, które trafiały do sieci *ICT-Net* w partiach po cztery sztuki. Trenowanie finalnego modelu było przeprowadzane na przestrzeni 100 epok, przy pomocy entropii skrośnej jako funkcji strat, za pomocą optymalizatora *RMSProp* o początkowej stopie uczenia równej 0,001. Co ciekawe autorzy omawianego w tym akapicie artykułu, po przeprowadzeniu badań empirycznych, zdecydowali się na zastosowanie 40-sto procentowego progu definiującego czy dany piksel należy do budynku czy nie, co oznacza, iż kwalifikowali oni jako piksele należące do budynków wszystkie piksele, dla których sieć *ICT-Net* zwróciła prawdopodobieństwo bycia budynkiem na poziomie 40% lub wyższym (zazwyczaj w literaturze próg ten wynosi 50%). Wyniki uzyskane przez Chatterjeego i Poullisa na zbiorze *Inria Aerial Image Labeling Dataset* przewyższały o około 1,5% najlepsze w tamtym czasie wyniki i do dnia pisania niniejszej pracy żadna inna sieć neuronowa nie uzyskała lepszych wyników dla tego zbioru (ang. *state of art*) - te wyniki to **97,14% OA** oraz **80,32% IoU**.

W kwietniu bieżącego roku, na łamach czasopisma *IEEE Transactions on Geoscience and Remote Sensing* ukazał się artykuł *Building Footprint Generation by Integrating Conv-*

*lution Neural Network with Feature Pairwise Conditional Random Field (FPCRF)* [37], którego autorami są Zhu X., Li X. Q., Shi Y. oraz Huang X. W artykule tym przedstawione jest kompleksowe podejście do detekcji budynków na zdjęciach lotniczych i satelitarnych, które jest realizowane przy pomocy sieci *FC-DenseNet* jako ekstraktora cech oraz metody *Feature Pairwise Conditional Random Field* mającej służyć jako grafowa korelacja pikseli poprawiająca jakość semantycznej segmentacji poprzez generowanie ostrych krawędzi budynków. Autorzy testują stworzony przez siebie model na kilku wiodących zbiorach danych:

- zdjęciach satelitarnych ze zbioru *Planetscope* dla miast Monachium, Paryż, Rzym i Zurych,
- zdjęciach lotniczych *ISPRS benchmark* dla miasta Poczdam,
- zdjęciach satelitarnych *WorldView3* ze zbioru *Dstl Kaggle dataset*,
- zdjęciach lotniczych *Inria Aerial Image Labeling Dataset* dla miast Austin, Chicago, Kitsap County, Western Tyrol, and Vienna.

Dla każdego z wyżej wymienionych zbiorów danych, autorzy omawianego w bieżącym akapicie artykułu, uzyskują satysfakcyjujące wyniki. Stworzona przez nich sieć jest zasilana fragmentami zdjęć o rozmiarze  $256 \times 256$  pikseli w partiach po cztery sztuki. Jako funkcja straty wykorzystywany jest *NLLLoss*, który jest minimalizowany przez optymalizator *SGD* ze startową stopą uczenia o wartości 0,0001. Wyniki uzyskane przez Zhu X., Li X. Q., Shi Y. oraz Huang X. dla zbioru *Inria Aerial Image Labeling Dataset* przedstawiają się następująco: **95,81% OA, 87,65% F1S** oraz **74,79% IoU**.

Ostatnim artykułem, który zostanie zaprezentowany w bieżącym rozdziale, jest artykuł *Polygonal Building Segmentation by Frame Field Learning* [7]. Artykuł ten został opublikowany w kwietniu bieżącego roku, a jego autorami są Girard N., Smirnov D., Solomon J. oraz Tarabalka Y. Autorzy Ci w swoim artykule postanowili skupić się na nauczeniu sieci neuronowej nie tylko odpowiedniej klasyfikacji pikseli, ale również odpowiedniego dopasowania do ramek poszczególnych budynków (ang. *frame field*). Takie rozwiązanie pozwoliło uzyskać predykcje z modelu, które w sposób efektywny mogły być konwertowane z obrazów rastrowych do wielokątów (ang. *polygon*), w szczególności przy użyciu zaprezentowanego w artykule algorytmu. Autorzy omawianego w bieżącym akapicie artykułu jako wiodące architektury swojego modelu wybrali dwie sieci: *U-Net* oraz *DeepLabV3*, a zdefiniowana przez nich funkcja straty stanowiła ważoną sumę binarnej entropii skrośnej oraz *Dice Loss*. Trenowanie modelu odbywało się na zbiorze danych *Inria Aerial Image Labeling Dataset*, gdzie z każdego zdjęcia lotniczego wycinane były fragmenty o rozdzielczości  $512 \times 512$  pikseli, które następnie były łączone w partie po 16 sztuk i w takiej formie trafiały do finalnego modelu. Tak zdefiniowana sieć pozwoliła uzyskać autorom zadowalające dla nich wyniki na poziomie **78% IoU**.

### 3. Przegląd literatury z zakresu detekcji budynków na zdjęciach lotniczych

L.p.	Tytuł artykułu	Wiodący zbiór danych	Architektura modelu	Funkcja strat	Optymalizator	Rozmiar partii	Uzyskane wynik
1.	Satellite Imagery Feature Detection using Deep Convolutional Neural Network: A Kaggle Competition [16]	DSTL Satellite Imagery	U-Net	Różnica pomiędzy: BCE i IoU	Nadam (LR: 0.001)	400	IoU: 62.9%
2.	Multi-Task Learning for Segmentation of Building Footprints with Deep Neural Networks [1]	Inria Aerial Image Labeling Dataset	SegNet	Uncertainty Based Multi-Task Loss	SGD (LR: 0.001)	10	OA: 95.17% IoU: 70.14%
3.	Automatic Building Segmentation of Aerial Imagery Using Multi-Constraint Fully Convolutional Networks [33]	Aerial images taken by Land Information of New Zealand	MC-FCN U-Net	BCE	Adam (LR: 0.001)	-	OA: 97.60% IoU: 83.30%
4.	Building Detection from Satellite Imagery Using a Composite Loss Function [9]	SpaceNet	LinkNet SE-ResNeXt-50	Ważona suma: BCE, LHL i MSE	SGD (LR: 0.01)	32	F1S: 76.79%
5.	Semantic Segmentation from Remote Sensor Data and the Exploitation of Latent Learning for Classification of Auxiliary Tasks [2]	Inria Aerial Image Labeling Dataset	ICT-Net	CE	RMSProp (LR: 0.001)	4	OA: 97.14% IoU: 80.32%
6.	Building Footprint Generation by Integrating Convolutional Neural Network with Feature Pairwise Conditional Random Field (FPCRFF) [37]	Inria Aerial Image Labeling Dataset	FC-DenseNet FPCRFF	NLLoss	SGD (LR: 0.0001)	4	OA: 95.81% F1S: 87.65% IoU: 74.79%
7.	Polygonal Building Segmentation by Frame Field Learning [7]	Inria Aerial Image Labeling Dataset	U-Net DeepLabV3	Ważona suma: BCE i Dice Loss	-	64	IoU: 78.00%

**Tabela 3.1.** Podsumowanie najważniejszych informacji pochodzących z artykułów omówionych w przeglądzie literatury

### 3. Przegląd literatury z zakresu detekcji budynków na zdjęciach lotniczych

Tabela 3.1 prezentuje podsumowanie najważniejszych informacji pochodzących z omówionych w bieżącym rozdziale artykułów naukowych z zakresu detekcji budynków na zdjęciach lotniczych i satelitarnych. Analiza literatury wskazuje, iż najczęściej wybieranym zborem danych, na którym przeprowadza się trenowanie modelu w zadaniach detekcji budynków na zdjęciach lotniczych jest zbiór *Inria Aerial Image Labeling Dataset*. Najczęściej stosowaną architekturą dla tego typu zadań jest architektura typu *U-Net* oraz jej modyfikacje, najczęściej stosowaną funkcją straty jest binarna entropia skrośna, a najczęściej wykorzystywanym optymalizatorem jest *SGD*. Dodatkowo warto zwrócić uwagę, iż w przeważającej większości artykułów, rozdzielcość zdjęć uczących sieć neuronową wynosiła  $256 \times 256$  pikseli. Powyższe wnioski płynące z zaprezentowanego w niniejszym rozdziale przeglądu literatury posłużą w dalszej części pracy do właściwego doboru początkowych wartości hiperparametrów tworzonej głębszej sieci neuronowej.

## 4. Konstrukcja sieci *GML-Net*

W bieżącym rozdziale szczegółowo opisany zostanie proces tworzenia autorskiej głębokiej sieci neuronowej o nazwie *GML-Net*. Sieć ta została skonstruowana przez autora niniejszej pracy w celu przeprowadzenia efektywnej detekcji budynków na zdjęciach lotniczych pochodzących ze zbioru *Inria Aerial Image Labeling Dataset*. W pierwszej części rozdziału przedstawione zostaną najważniejsze hiperparametry modelu oraz proces wczytywania, podziału i obróbki danych treningowych i walidacyjnych. W kolejnej części przedstawiona zostanie architektura sieci *GML-Net*, warstwy przetwarzania (bloki), które tę architekturę tworzą oraz sposób trenowania tej sieci.

### 4.1. Najważniejsze hiperparametry sieci

Pierwszym krokiem procesu tworzenia modelu finalnej sieci neuronowej jest definicja bibliotek języka Python, które zostaną wykorzystane w trakcie obliczeń na poszczególnych etapach tworzenia modelu. Główną biblioteką, która została zastosowana w bieżącej pracy do implementacji głębokiego uczenia jest biblioteka *PyTorch*. Została ona wybrana, gdyż pozwala ona na efektywne i elastyczne posługiwanie się podstawowymi jednostkami przetwarzania w architekturze głębokich sieci neuronowych, zapewnia pełną integrację z biblioteką *NumPy* oraz umożliwia wygodną definicję zbiorów danych uczących i walidacyjnych.

#### Kod źródłowy 1.

```
1 import os
2 import time
3 import copy
4 import glob
5 import torch
6 import random
7 import numpy as np
8 import pandas as pd
9 from math import exp
10 from PIL import Image
11 import torch.nn as nn
12 import torch.optim as optim
13 from datetime import datetime
14 from google.colab import drive
15 import matplotlib.pyplot as plt
16 from torchsummary import summary
17 from torch.utils.data import Dataset
```

```
18 | from torchvision import transforms, models  
19 | from numpy.lib.stride_tricks import as_strided  
20 | from collections import defaultdict, OrderedDict
```

Kolejnym krokiem tworzenia sieci *GML-Net* była definicja jej najważniejszych hiperparametrów, którymi są:

- rozdzielcość fragmentów obrazów, na których trenowana będzie sieć *GML-Net* (*sample\_res*),
- rozdzielcość obrazów na jakie będzie podzielony zostanie obraz wejściowy przed pobraniem z niego próbek *sample\_res* x *sample\_res* (*sub\_arr\_size*),
- rozmiar batcha w data loaderze (*batch\_s*),
- najważniejsze parametry optymalizatora (*learn\_rate*, *momt*, *wght\_dec*),
- najważniejsze parametry schedulera (*step\_s*, *gamm*),
- wagi poszczególnych składowych funkcji strat (*wghts\_dict*),
- wartość, powyżej której uznajemy że dany piksel został zakwalifikowany jako będący częścią budynku (*loss\_thres*),
- liczba epok trenowania (*epoch\_num*),
- minimalna wartość stopy uczenia (*optim\_eps*).

## Kod źródłowy 2.

```
1 google_path = "/content/drive/My Drive/GSN/Thesis/Dane/AID/"  
2 train_path = "/content/AerialImageDataset/"  
3 mdl_path = "/content/drive/My Drive/GSN/Thesis/Najlepszy model/"  
4 mdl_name = "ResNet50_wide_UNet_depth_3_BottleNeck_BS_18"  
5 org_h_dim = 5000  
6 org_v_dim = 5000  
7 sub_arr_size = 1000  
8 sample_res = 256  
9 recalc_flag = False  
10 batch_s = 18  
11 learn_rate = 0.01  
12 momt = 0.9  
13 wght_dec = 0.0005  
14 step_s = 5  
15 gamm = 0.5
```

```

16 wghts_dict = {"bce": 0.3, "lhl": 0.3, "dice": 0.4}
17 loss_thres = 0.5
18 epoch_num = 200
19 optim_eps = 1e-08

```

Jak widać wartości zastosowanych hiperparamterów sieci *GML-Net* w dużej części pokrywają się z wartościami hiperparapetrów obserwowanych w literaturze badawczej - zdecydowano się na obrazy o rozdzielcości  $256 \times 256$  pikseli, początkową stopę uczenia na poziomie 0,01, momentum wartości 0,9 i spadek wag równy 0,0005.

### Kod źródłowy 3.

```

1 params_dict = {}
2 params_dict["sub_arr_size"] = sub_arr_size
3 params_dict["sample_res"] = sample_res
4 params_dict["recalc_flag"] = recalc_flag
5 params_dict["batch_s"] = batch_s
6 params_dict["learn_rate"] = learn_rate
7 params_dict["momt"] = momt
8 params_dict["wght_dec"] = wght_dec
9 params_dict["step_s"] = step_s
10 params_dict["gamm"] = gamm
11 params_dict["bce"] = wghts_dict["bce"]
12 params_dict["lhl"] = wghts_dict["lhl"]
13 params_dict["dice"] = wghts_dict["dice"]
14 params_dict["loss_thres"] = loss_thres
15 params_dict["epoch_num"] = epoch_num
16 params_dict["org_h_dim"] = org_h_dim
17 params_dict["org_v_dim"] = org_v_dim

```

Wartości wszystkich hiperparmetrów modelu zostały zapisane w słowniku *params\_dict*, po to by zawsze mieć pewność przy użyciu jakich hiperparameatrów dany model był trenowany.

#### 4.2. Dane uczące i ich transformacje

Pierwszym krokiem przygotowania danych treningowych i walidacyjnych sieci *GML-Net* był import zbioru treningowego *Inria Aerial Image Labeling Dataset* z dysku *Google* na dysk *Coloba*.

**Kod źródłowy 4.**

```

1 drive.mount('/content/drive')
2
3 if os.path.exists("sample_data"):
4     !rm -r sample_data
5
6 if not os.path.exists(train_path):
7     !cp -R "$google_path" $train_path
8
9 curr_wd = os.getcwd()

```

Pobrany zbiór treningowy składa się ze 180 zdjęć lotniczych pięciu miast: Austin, Chicago, Hrabstwo Kitsap, Tyrol oraz Wiedeń - po 36 zdjęć każdego z miast. Każde z tych zdjęć ma przypisaną do siebie maskę *ground truth*, która wskazuje które piksele zdjęcia bazowego reprezentują budynki. Zdjęcia bazowe mają rozdzielcość  $5000 \times 5000$  pikseli, trzy kanały przestrzenne, każde z nich pokrywa teren o powierzchni  $1500 \times 1500 \text{ m}^2$ . Zbiór walidacyjny został wydzielony ze zbioru treningowego poprzez wybranie pięciu pierwszych zdjęć każdego z miast (zgodnie z podejściem zastosowanym w literaturze badawczej) - w ten sposób powstał finalny zbiór treningowy składający się ze 155 zdjęć oraz finalny zbiór walidacyjny składający się z 25 zdjęć. Opisany tu podział jest realizowany przy pomocy funkcji *create\_meta\_info*, która tworzy pliki CSV zawierające ścieżki do zdjęć z poszczególnych zbiorów.

**Kod źródłowy 5.**

```

1 def create_meta_info(root_directory):
2     l_meta = []
3     l_meta_t = []
4     l_meta_v = []
5     os.chdir(root_directory)
6
7     for file in glob.glob("*_GT.tif"):
8         mask_path = os.path.join(root_directory, file)
9         im_path = os.path.join(root_directory, file[:-7] + ".tif")
10        l_meta.append((im_path, mask_path))
11        if not im_path[-6].isdigit() and im_path[-5] in ["1", "2", "3", "4", "5"]:
12            l_meta_v.append((im_path, mask_path))
13        else:
14            l_meta_t.append((im_path, mask_path))
15
16    os.chdir(curr_wd)
17    if l_meta:
18        pd.DataFrame(l_meta).to_csv('m_inf.csv', index=False, header=False)

```

```

19 pd.DataFrame(l_meta_t).to_csv('m_inf_t.csv', index=False, header=False)
20 pd.DataFrame(l_meta_v).to_csv('m_inf_v.csv', index=False, header=False)

```

Posiadając już zdjęcia podzielone na zbiór treningowy i walidacyjny, kolejnym etapem obróbki danych było wydzielenie odpowiednich próbek z tych zdjęć. W związku z tym, iż zbiór *Inria Aerial Image Labeling Dataset* składa się ze zdjęć o bardzo wysokiej rozdzielczości i niemożliwe oraz bardzo nieefektywne byłoby przeprowadzenie treningu sieci neuronowej na zdjęciach o rozdzielczości  $5000 \times 5000$  zdecydowano się na podział tych zdjęć (oraz odpowiadających im masek) na 25 równych części, każda o rozdzielczości  $1000 \times 1000$  pikseli. Dodatkowo, żeby nie utracić informacji znajdujących się na krawędziach podzielonych zdjęć, ze zdjęć bazowych (oraz z odpowiadających im masek) zdecydowano się wydzielić kolejnych 16 części (również o rozdzielczości  $1000 \times 1000$  pikseli) znajdujących się na łączeniach każdych czterech sąsiadujących ze sobą bazowych części. W ten sposób z każdego zdjęcia znajdującego się w bazowym zbiorze treningowym poznano 41 fragmentów (oraz 41 fragmentów odpowiadających im masek), co przekłada się na łączną liczbę 6355 fragmentów trenujących ( $155 * 41$ ). Dodając do tego fakt, iż z każdego takiego fragmentu wybranych zostanie 18 losowych okien (rozmiar partii równy 18) o rozmiarach  $256 \times 256$  pikseli, uzyskujemy finalnie 114390 przykładów uczących. Pobieranie fragmentów z poszczególnych zdjęć (masek) zostało zrealizowane przy pomocy funkcji *resample*, która z kolei opiera się na funkcji *as\_strided* pochodzącej z biblioteki *numpy.lib.stride\_tricks*.

### Kod źródłowy 6.

```

1 def resample(image, N, org_h_dim, org_v_dim):
2     wind_h = wind_w = N
3     img_shp = image.shape
4     nm_cls = img_shp[-1] if len(img_shp) > 2 else 1
5     n_arrs1 = int((img_shp[0] // wind_h))
6     n_arrs2 = int((img_shp[1] // wind_w))
7     s_l = wind_h * org_h_dim * nm_cls
8     s_k = wind_w * nm_cls
9
10    if nm_cls == 1:
11        new_strds = (s_l, s_k, org_v_dim * nm_cls, nm_cls)
12        new_shape = (n_arrs1, n_arrs2, wind_h, wind_w)
13        img_winds = as_strided(image, shape=new_shape, strides=new_strds)
14        fin_winds = img_winds.reshape((n_arrs1 * n_arrs2, wind_h, wind_w))
15    else:
16        new_strds = (s_l, s_k, org_v_dim * nm_cls, nm_cls, 1)
17        new_shape = (n_arrs1, n_arrs2, wind_h, wind_w, nm_cls)
18        img_winds = as_strided(image, shape=new_shape, strides=new_strds)

```

```

19     fin_winds = img_winds.reshape((n_arrows1*n_arrows2, wind_h, wind_w, nm_cls))
20     return fin_winds

```

Pozyskane fragmenty zdjęć i masek zostały następnie połączone w listy, w celu ułatwienia ich przetwarzania na kolejnych etapach - łączenie to zostało zrealizowane przy pomocy funkcji *get\_img\_smpl*.

### Kod źródłowy 7.

```

1 def get_img_smpl(idx, sub_arr_size, img_arr, org_h_dim, org_v_dim):
2     image = np.array(Image.open(img_arr[idx, 0]))
3     mask = np.array(Image.open(img_arr[idx, 1]))
4     img_sub_arrows1 = resample(image, sub_arr_size, org_h_dim, org_v_dim)
5     msk_sub_arrows1 = resample(mask, sub_arr_size, org_h_dim, org_v_dim)
6     s1 = list(zip(img_sub_arrows1, msk_sub_arrows1))
7     half_wind = int(sub_arr_size // 2)
8     img_hv = image[half_wind:-half_wind, half_wind:-half_wind, ...]
9     mask_hv = mask[half_wind:-half_wind, half_wind:-half_wind, ...]
10    img_sub_arrows2 = resample(img_hv, sub_arr_size, org_h_dim, org_v_dim)
11    msk_sub_arrows2 = resample(mask_hv, sub_arr_size, org_h_dim, org_v_dim)
12    s2 = list(zip(img_sub_arrows2, msk_sub_arrows2))
13    return s1, s2

```

Aby ułatwić proces pobierania fragmentów ze zdjęć i masek bazowych oraz umożliwić ich efektywną augmentację postanowiono połączyć te operacje przy pomocy jednej klasy *ResampledImageDataset*.

### Kod źródłowy 8.

```

1 class ResampledImageDataset(Dataset):
2     def __init__(self, csv_file, org_h, org_v, sub_arr_s, transform=None):
3         self.img_arr = pd.read_csv(csv_file, header=None).values
4         self.transform = transform
5         self.org_h = org_h
6         self.org_v = org_v
7         self.sub_arr_s = sub_arr_s
8
9     def __len__(self):
10        return len(self.img_arr)
11
12    def __getitem__(self, idx):
13        s1, s2 = get_img_smpl(idx, self.sub_arr_s, self.img_arr, self.org_h,
14                               self.org_v)

```

```

16     if self.trasform is not None:
17         s1 = self.trasform(s1)
18         s2 = self.trasform(s2)
19     return s1 + s2

```

Dodatkowo, aby wyliczyć średnią i odchylenie standardowe dla poszczególnych kanałów zdjęć bazowych po to by móc zastosować te statystyki do normalizacji, stworzono klasę *BasicImageDataset*.

### Kod źródłowy 9.

```

1 class BasicImageDataset(Dataset):
2     def __init__(self, csv_file, transform=None):
3         self.images_arr = pd.read_csv(csv_file, header=None).values
4         self.trasform = transform
5
6     def __len__(self):
7         return len(self.images_arr)
8
9     def __getitem__(self, idx):
10        image = np.array(Image.open(self.images_arr[idx, 0]))
11        mask = np.array(Image.open(self.images_arr[idx, 1]))
12        sample = [image, mask]
13
14        if self.trasform is not None:
15            sample = self.trasform(sample)
16
17    return sample

```

Dla każdego zdjęcia pochodzącego z klasy *BasicImageDataset* zapisywane są wartości jego kolorów w poszczególnych kanałach, a po przejściu przez wszystkie zdjęcia wyliczane są globalne średnie i odchylenia standardowe dla wszystkich kanałów.

### Kod źródłowy 10.

```

1 create_meta_info(train_path)
2 basic_ds = BasicImageDataset("m_inf.csv")
3 basic_ds_len = len(basic_ds)
4 sngl_img_pxl_num = basic_ds[0][0].size
5 tot_pxl_num = (sngl_img_pxl_num * basic_ds_len) / 3
6
7 if recalc_flag:
8     mean_rgbs = (np.array([sample[0].sum((0, 1)) for sample in
9                           basic_ds]).sum(0) / tot_pxl_num) / 255

```

```

10 std_rgbs = np.sqrt(np.array([np.power(sample[0] - mean_rgbs, 2)].sum((0, 1)))
11                                     for sample in basic_ds]).sum(0) /
12                                     tot_pxl_num) / 255
13 else:
14     mean_rgbs = np.array([0.40483995, 0.42726254, 0.39271341])
15     std_rgbs = np.array([0.45071778, 0.4634665 , 0.42900409])

```

Kolejnym etapem tworzenia zbiorów treningowego i walidacyjnego jest definicja augmentacji, które zostaną zastosowane na tych zbiorach.

### Kod źródłowy 11.

```

1 class RandomHorizontalFlip(object):
2     def __init__(self, p=0.5):
3         super().__init__()
4         self.p = p
5
6     def __call__(self, all_samples):
7         trans_list = []
8
9         for sample in all_samples:
10            image, mask = sample[0], sample[1]
11            image = transforms.functional.to_pil_image(image)
12            mask = transforms.functional.to_pil_image(mask)
13
14            if torch.rand(1) < self.p:
15                image = transforms.functional.hflip(image)
16                mask = transforms.functional.hflip(mask)
17
18            trans_list.append((image, mask))
19
20     return trans_list

```

Pierwszą z nich jest augmentacja realizowana przez losowy horyzontalny obrót zdjęcia (maski) wzdłuż środkowej osi - losowość polega na tym, iż taka augmentacja dochodzi do skutku z prawdopodobieństwem 50% - czyli dla co drugiego zdjęcia.

### Kod źródłowy 12.

```

1 class RandomVerticalFlip(object):
2     def __init__(self, p=0.5):
3         super().__init__()
4         self.p = p
5

```

```

6  def __call__(self, all_samples):
7      trans_list = []
8
9      for sample in all_samples:
10         image, mask = sample[0], sample[1]
11
12         if torch.rand(1) < self.p:
13             image = transforms.functional.vflip(image)
14             mask = transforms.functional.vflip(mask)
15
16         trans_list.append((image, mask))
17
18     return trans_list

```

Drugą zdefiniowaną augmentacją jest losowy wertykalny obrót zdjęcia (maski) wzdłuż środkowej osi, który również jest realizowany z prawdopodobieństwem 50%.

### Kod źródłowy 13.

```

1 class RandomRotation(object):
2     def __call__(self, all_samples):
3
4         trans_list = []
5
6         for sample in all_samples:
7             image, mask = sample[0], sample[1]
8             angle_mult = random.randint(0, 12)
9             rot_img = transforms.functional.rotate(image, angle_mult * 30, expand=True)
10            rot_mask = transforms.functional.rotate(mask, angle_mult * 30, expand=True)
11            trans_list.append((rot_img, rot_mask))
12
13     return trans_list

```

Kolejną augmentacją zdefiniowaną dla zbiorów treningowego i walidacyjnego jest rotacja zdjęcia (maski) o losowy kąt będący wielokrotnością kąta 30 stopni.

### Kod źródłowy 14.

```

1 class RandomCrop(object):
2     def __init__(self, out_s):
3         assert isinstance(out_s, (int, tuple))
4         self.out_s = out_s
5
6     def __call__(self, all_samples):

```

```

7   trans_list = []
8
9   for sample in all_samples:
10    image, mask = np.array(sample[0]), np.array(sample[1])
11    out_s = (self.out_s, self.out_s) if isinstance(self.out_s, int) else out_s
12    new_h, new_w = out_s
13    top = np.random.randint(0, image.shape[0] - new_h)
14    left = np.random.randint(0, image.shape[1] - new_w)
15    image = image[top:(top + new_h), left: left + new_w]
16    mask = mask[top:(top + new_h), left: left + new_w]
17    trans_list.append((image, mask))
18
19  return trans_list

```

Następną augmentacją wykorzystywaną przy budowie sieci *GML-Net* jest wybór losowego okna o rozmiarach 256x256 pikseli.

### Kod źródłowy 15.

```

1  class Normalize(object):
2      def __init__(self, mean, std, inplace=False):
3          self.mean = mean
4          self.std = std
5          self.inplace = inplace
6
7      def __call__(self, all_samples):
8          trans_list = []
9
10     for sample in all_samples:
11         image, mask = sample[0], sample[1]
12         norm_img = transforms.functional.normalize(image / 255.0, self.mean,
13                                         self.std, self.inplace)
14
15         norm_mask = mask / 255.0
16         trans_list.append((norm_img, norm_mask))
17
18     return trans_list

```

Ostatnią augmentacją wykorzystywaną w bieżącej pracy jest normalizacja zdjęć przy użyciu globalnych średnich i odchyлеń standardowych, których sposób wyliczenia został opisany powyżej.

## Kod źródłowy 16.

```

1 class ToTensor(object):
2     def __call__(self, all_samples):
3
4         trans_list = []
5
6         for sample in all_samples:
7             image, mask = np.array(sample[0]), np.array(sample[1])
8             trans_list.append((torch.from_numpy(image).permute(2, 0, 1),
9                               torch.from_numpy(mask)))
10
11     return trans_list

```

Po zastosowaniu wszystkich augmentacji, należy przekonwertować zdjęcia (maski) z formatu macierzy *NumPy* do formatu tensora *PyTorch* tak, żeby mogły one być wykorzystane do trenowania finalnej sieci neuronowej - odpowiedzialną za ten proces jest klasa *ToTensor*.

## Kod źródłowy 17.

```

1 trn_trans = transforms.Compose([RandomHorizontalFlip(),
2                                RandomVerticalFlip(),
3                                RandomRotation(),
4                                RandomCrop(sample_res),
5                                ToTensor(),
6                                Normalize(mean_rgbs, std_rgbs)])
7
8 train_dataset = ResampledImageDataset("m_inf_t.csv", org_h_dim,
9                                       org_v_dim, sub_arr_size,
10                                      transform=trn_trans)
11
12 vld_trans = transforms.Compose([RandomCrop(sample_res), ToTensor(),
13                                Normalize(mean_rgbs, std_rgbs)])
14
15 valid_dataset = ResampledImageDataset("m_inf_v.csv", org_h_dim,
16                                       org_v_dim, sub_arr_size,
17                                       transform=vld_trans)

```

Finalnym krokiem tworzenia zbiorów treningowego i walidacyjnego było zdefiniowanie kolejności poszczególnych augmentacji. Jak widać w przypadku zbioru walidacyjnego augmentacje zostały ograniczone do wyboru losowego okna i normalizacji, po to by móc oceniać jakość modelu na jak najbardziej stabilnych danych.

**Kod źródłowy 18.**

```

1 org_data_set = ResampledImageDataset("m_inf.csv", org_h_dim,
2                                     org_v_dim, sub_arr_size)
3 ds_len = len(org_data_set)
4 rnd_idx = random.randint(0, ds_len - 1)
5 rnd_idx2 = random.randint(0, len(org_data_set[0]) - 1)
6 org_img0, org_mask0 = org_data_set[rnd_idx][rnd_idx2]
7
8 org_img0_shp = org_img0.shape
9 org_img = np.zeros((org_img0_shp[0], org_img0_shp[1], 4), dtype=np.uint8)
10 org_img[..., :3] = org_img0
11 org_img[..., 3] = 255
12
13 plt_org_mask = np.zeros((org_img0_shp[0], org_img0_shp[1], 4),
14                         dtype=np.uint8)
15 plt_org_mask[..., 0] = org_mask0.squeeze()
16 plt_org_mask[org_mask0.squeeze() > 0, 3] = 180

```

Kolejnym etapem obróbki i wczytywania danych było wyświetlenie losowego zdjęcia wraz z jego maską za zbioru treningowego po to by sprawdzić czy proces generowania danych uczących przebiega poprawnie.

**Kod źródłowy 19.**

```

1 ds_len2 = len(train_dataset)
2 rnd_idx21 = random.randint(0, ds_len2 - 1)
3 rnd_idx22 = random.randint(0, len(train_dataset[0]) - 1)
4 trans_img0, trans_mask0 = train_dataset[rnd_idx21][rnd_idx22]
5
6 trans_img0 = torch.clamp(trans_img0, 0, 1).permute(1, 2, 0).cpu().numpy()
7 trans_img0_shp = trans_img0.shape
8 trans_img = np.zeros((trans_img0_shp[0], trans_img0_shp[1], 4))
9 trans_img[..., :3] = trans_img0
10 trans_img[..., 3] = 1.0
11
12 trans_mask0 = torch.clamp(trans_mask0, 0, 1).cpu().numpy()
13 plt_trans_mask = np.zeros((trans_img0_shp[0], trans_img0_shp[1], 4))
14 plt_trans_mask[..., 0] = trans_mask0.squeeze()
15 plt_trans_mask[trans_mask0.squeeze() > 0.5, 3] = 0.7

```

Aby móc efektywnie wyświetlić zdjęcie uczące wraz z odpowiadającą mu maską, zdecydo-

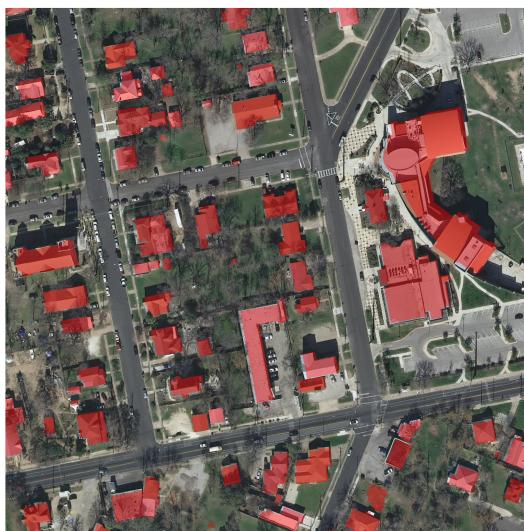
wano się przekonwertować je do formatu RBGA po to by następnie móc je zwizualizować przy pomocy funkcji *plt* z biblioteki *matplotlib.pyplot*.

### Kod źródłowy 20.

```

1 f2, axarr2 = plt.subplots(nrows=1, ncols=2, figsize=(20, 30))
2 axarr2[0].set_title("Oryginalne zdjecie\n", color='black', fontweight="bold",
3                      fontsize=25)
4 axarr2[0].imshow(org_img)
5 axarr2[0].imshow(plt_org_mask, alpha=0.6)
6 axarr2[0].axis('off')
7 axarr2[1].imshow(trans_img)
8 axarr2[1].imshow(plt_trans_mask, alpha=0.6)
9 axarr2[1].axis('off')
10 axarr2[1].set_title("Przetransformowane zdjecie\n", color='black',
11                      fontweight="bold", fontsize=25)
12 plt.savefig('Org_trans_img.png', dpi=300, bbox_inches='tight')
```

**Oryginalne zdjecie**



**Przetransformowane zdjecie**



**Rysunek 4.1.** Porównanie oryginalnego oraz przetransformowanego zdjęcia

Ostatnim etapem przygotowywania zbiorów treningowych i walidacyjnych było utworzenie dla nich obiektów ładujących dane (ang. data loader), które posłużyły bezpośrednio do zasilenia finalnego modelu w przykłady uczące / walidacyjne.

### Kod źródłowy 21.

```

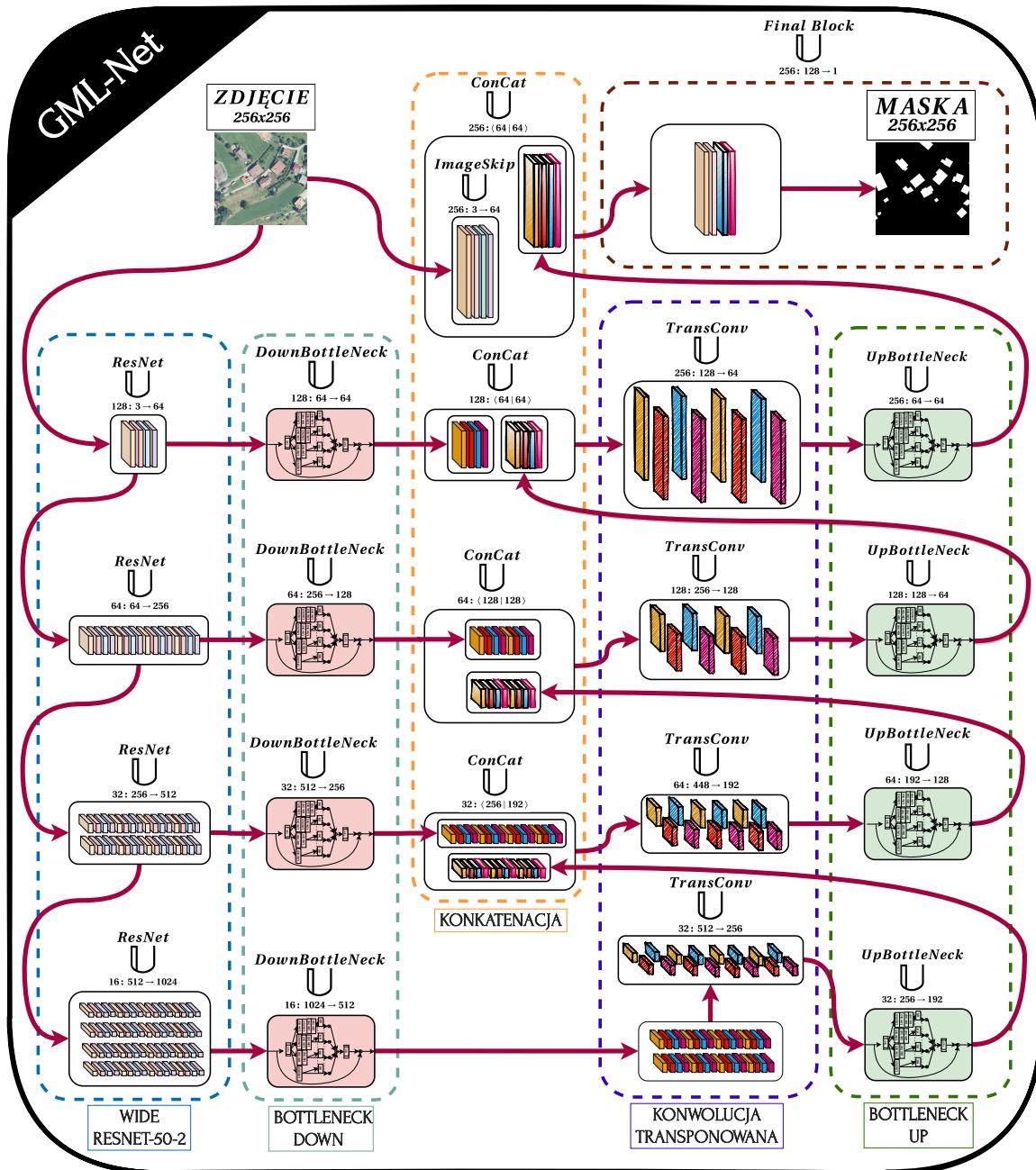
1 train_data_gen = torch.utils.data.DataLoader(train_dataset, shuffle=True,
2                                               batch_size=batch_s,
3                                               num_workers=2,
4                                               pin_memory=True,
```

```
5                         drop_last=True)
6
7 valid_data_gen = torch.utils.data.DataLoader(valid_dataset,
8                                              batch_size=batch_s,
9                                              num_workers=2,
10                                             pin_memory=True,
11                                             drop_last=True)
12
13 dataset_sizes = {"Train": len(train_data_gen.dataset), "Valid":
14                           len(valid_data_gen.dataset)}
15
16 data_loaders = {"Train": train_data_gen, "Valid": valid_data_gen}
```

Po ustrukturyzowaniu procesu wczytywania i obróbki danych, kolejnym krokiem w budowie modelu detekcji budynków na zdjęciach lotniczych, była definicja finalnej architektury sieci *GML-Net*.

### 4.3. Architektura sieci

Architektura sieci *GML-Net* wzorowana jest na architekturze *U-Net*, jest jednak od niej płytsza o jeden poziom ekstrakcji cech / upsamplingu, stąd też występują w niej tylko trzy połączenia rezydualne. Rolę enkodera w sieci *GML-Net* pełni sieć *Wide ResNet-50-2*, która tym różni się od zwykłej sieci *ResNet-50*, że posiada dwa razy więcej map cech w blokach rezydualnych, co czyni ją sporo szerszą i w efekcie skuteczniejszą w wychwytywaniu ukrytych zależności. Cechy wyekstraktowane przy pomocy *Wide ResNet-50-2* na danym poziomie rozdzielczości przestrzennej są następnie przetwarzane przez blok *BottleNeck* (wzorowany na blokach sieci *OSNet*) po to by uzyskać zagregowany zbiór cech z przekroju różnych skal. Zbiór ten jest następnie transferowany, przy pomocy połączeń rezydualnych, bezpośrednio do warstw dekodera. Rekonstrukcja cech w dekoderze przeprowadzana jest przy pomocy konwolucji transponowanej. Zrekonstruowane cechy są przetwarzane przez blok *BottleNeck*, a następnie przeprowadzana jest ich konkatenacja z cechami uzyskanymi z enkodera - tak połączone mapy cech stają się wsadem do kolejnych, wyższych warstw dekodera. W całej sieci powszechnie wykorzystywane są konwolucje separowalne wgęebnie i punktowo, co pozwala sieci *GML-Net* efektywnie nauczyć się korelacji kanałów przestrzennych, uniknąć nadmiernego dopasowania oraz uzyskać większą efektywność obliczeniową. Stąd też pomimo wykorzystania stosunkowo ciężkich architektur jakimi są bez wątpienia *U-Net* i *Wide ResNet-50-2*, udało się ograniczyć liczbę trenowalnych parametrów finalnego modelu *GML-Net* do 27 milionów. Rysunek 4.2 przedstawia architekturę omawianej w bieżącym rozdziale sieci:



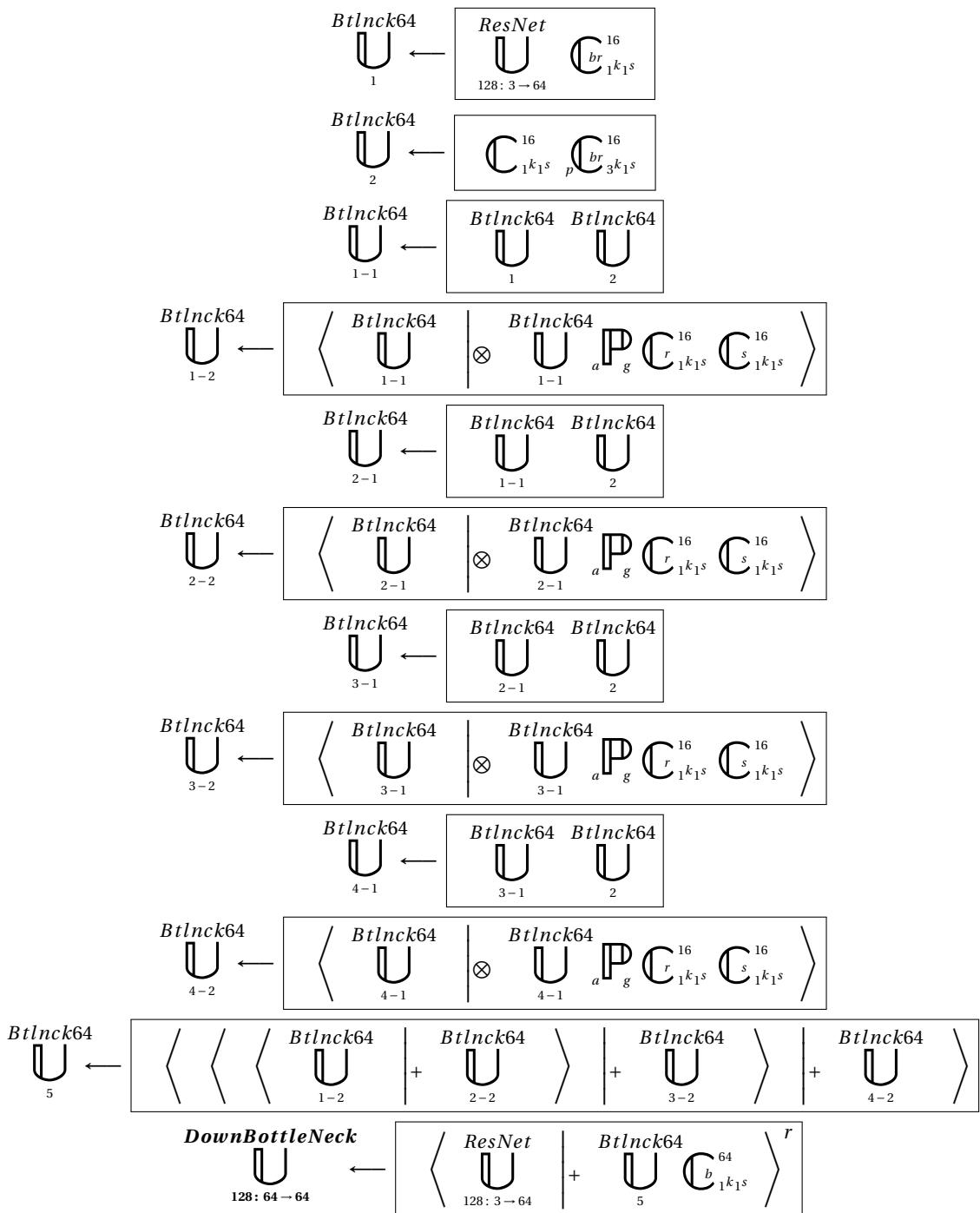
Rysunek 4.2. Architektura sieci GML-Net

Symboly nad poszczególnymi elementami sieci przedstawionej na powyższym rysunku zostały definiowane zgodnie z zapisem STNN dla architektur głębokich sieci neuronowych, który został opracowany przez prof. dr. hab. inż. Władysława Skarbka w artykule *Symbolic Tensor Neural Networks for Digital Media – from Tensor Processing via BNF Graph Rules to CREAMS Applications* [29]. Poniżej zaprezentowano szczegółową definicję każdego z tych symboli:

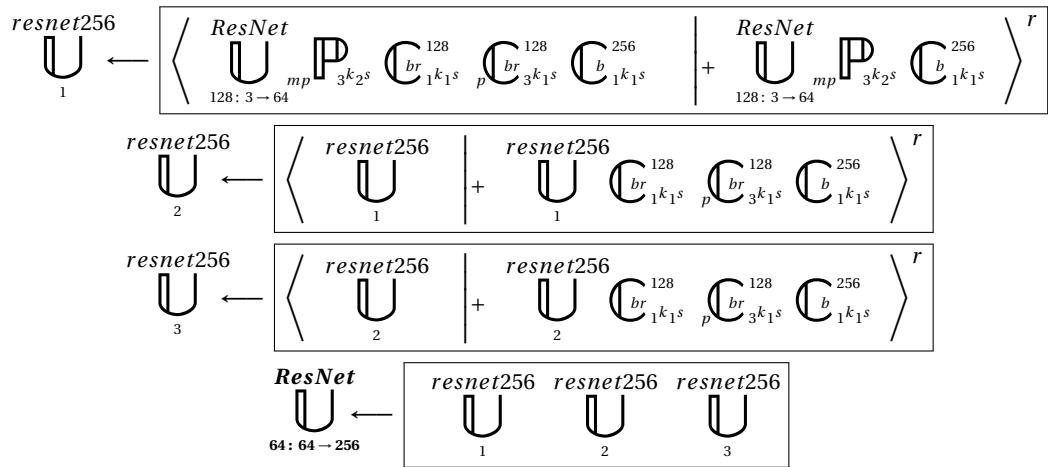
**Ekstrakcja cech nr 1 - 64 mapy cech, rozdzielcość przestrzenna  $128 \times 128$  (*ResNet64*):**



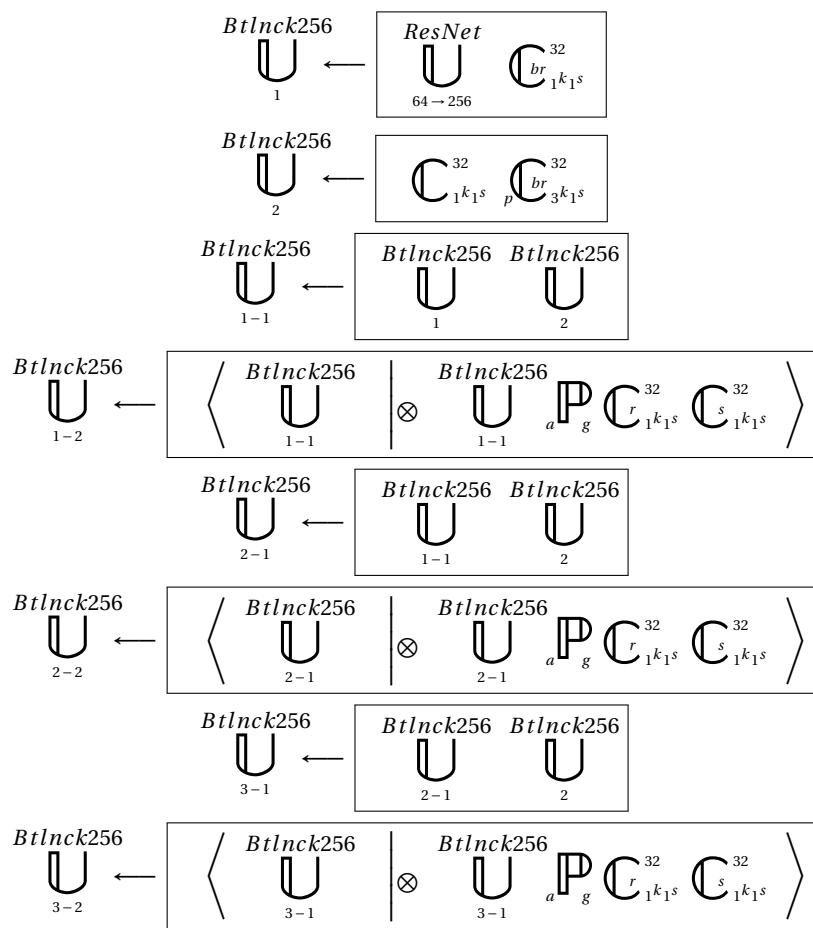
**Przetworzenie wyekstrahowanych cech *ResNet64* przez blok *DownBottleNeck64*:**

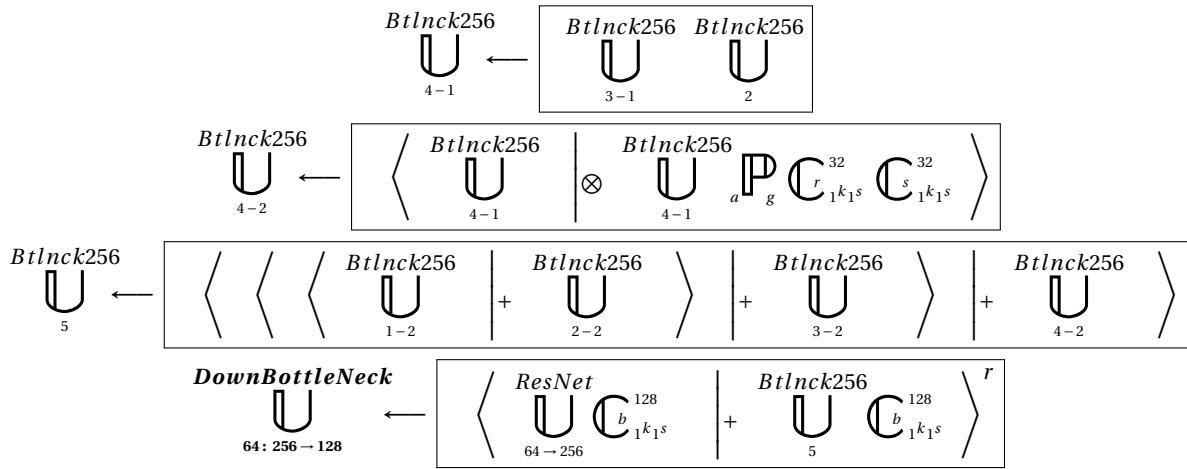


**Ekstrakcja cech nr 2 - 256 map cech, rozdzielcość przestrzenna 64x64 (ResNet256):**

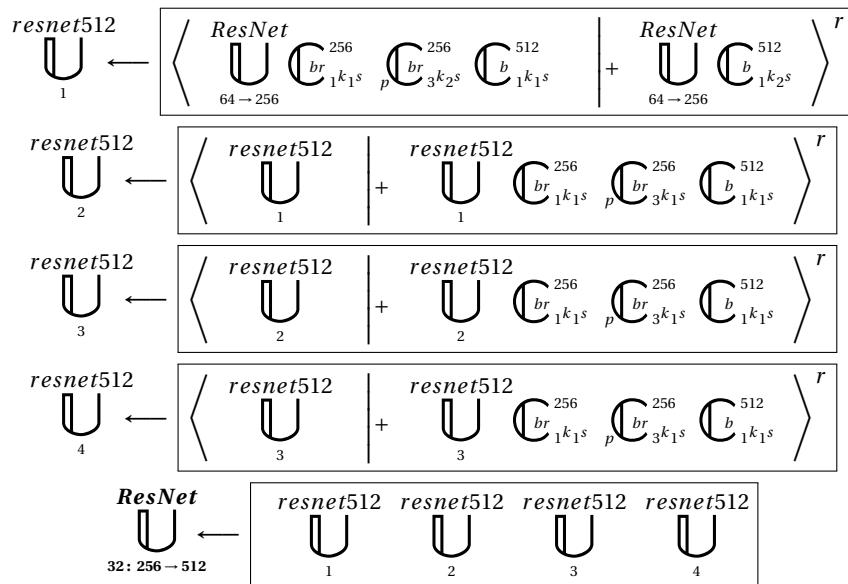


**Przetworzenie wyekstrahowanych cech ResNet256 przez blok DownBottleNeck128:**

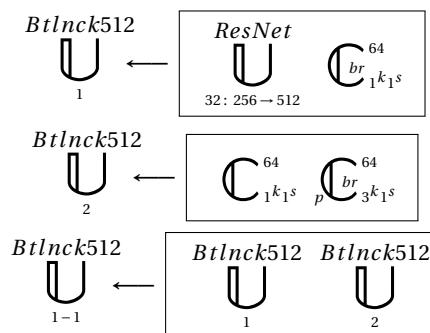


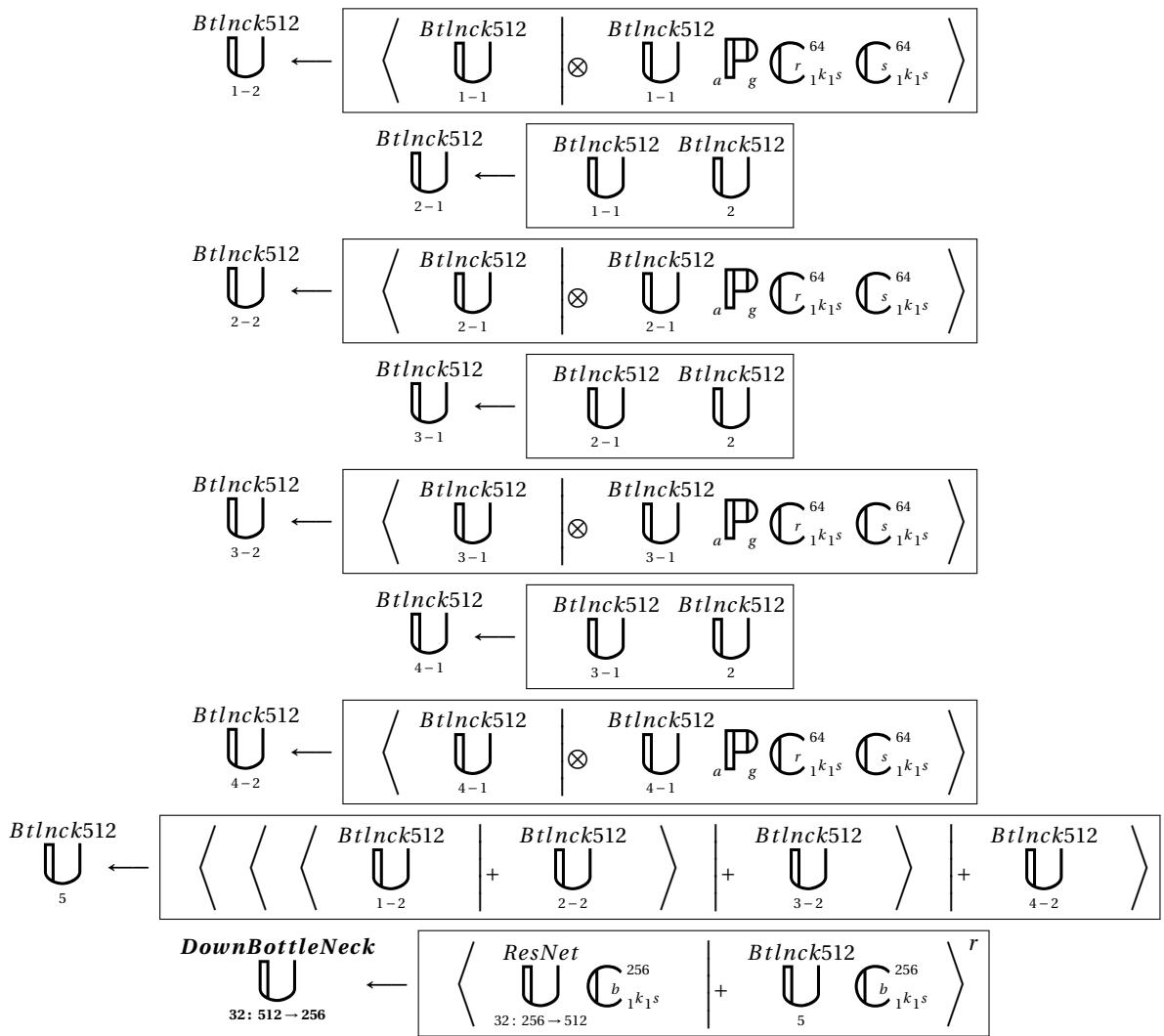


**Ekstrakcja cech nr 3 - 512 map cech, rozdzielcość przestrzenna 32x32 (ResNet512):**

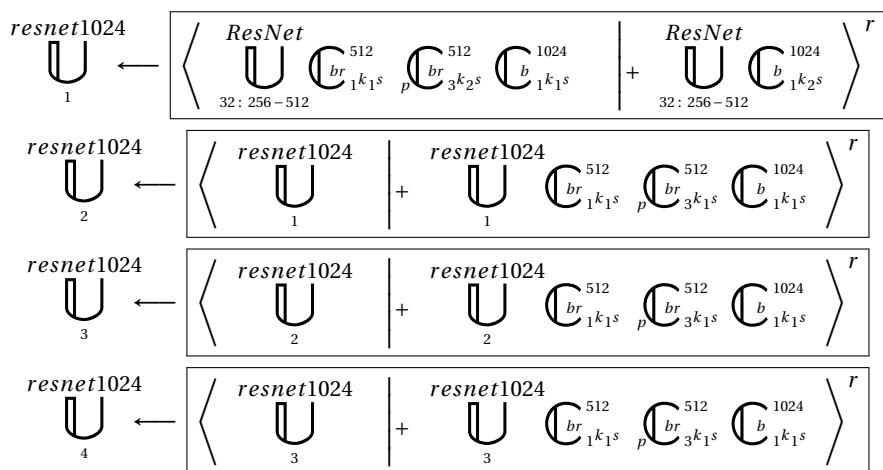


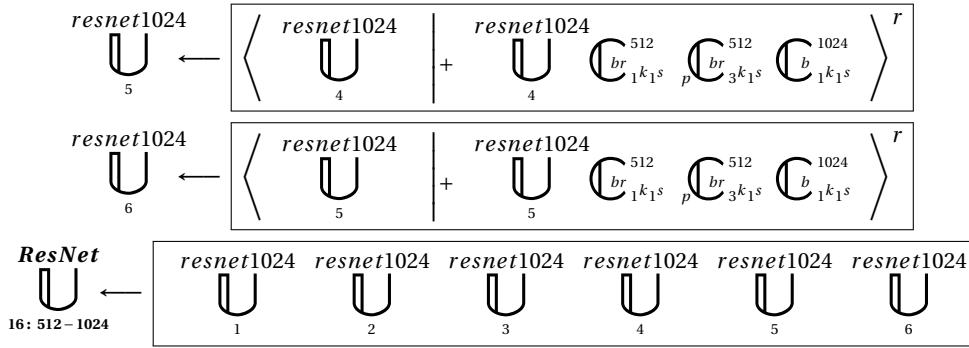
**Przetworzenie wyekstrahowanych cech *ResNet512* przez blok *DownBottleNeck256*:**



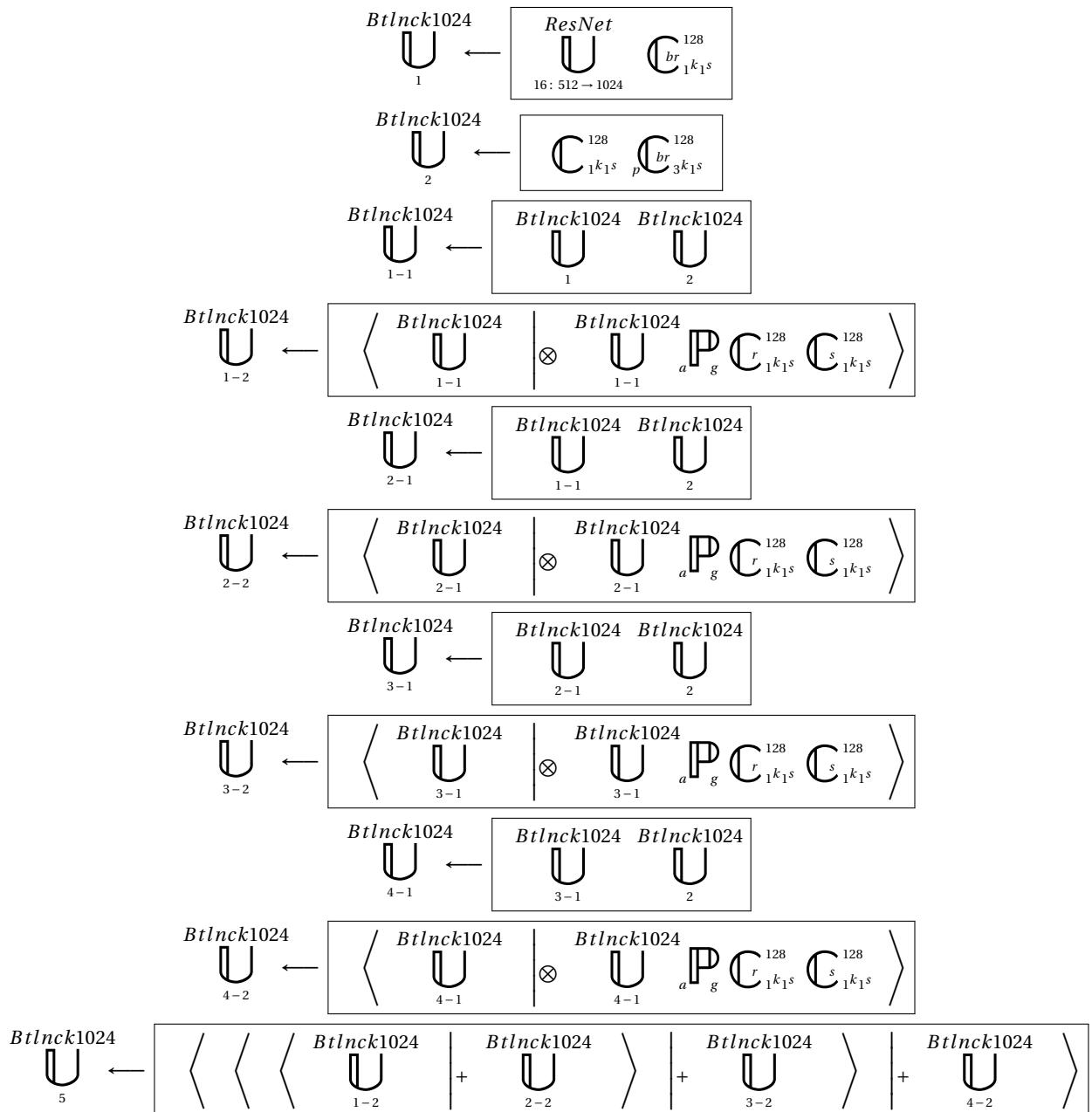


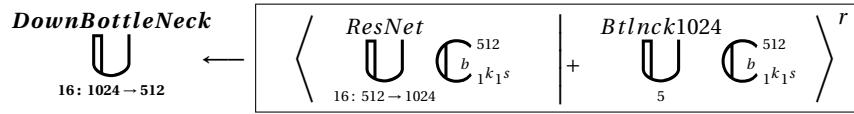
**Ekstrakcja cech nr 4 - 1024 mapy cech, rozdzielcość przestrzenna 16x16 (ResNet1024):**





**Przetworzenie wyekstrahowanych cech ResNet1024 przez blok DownBottleNeck512:**

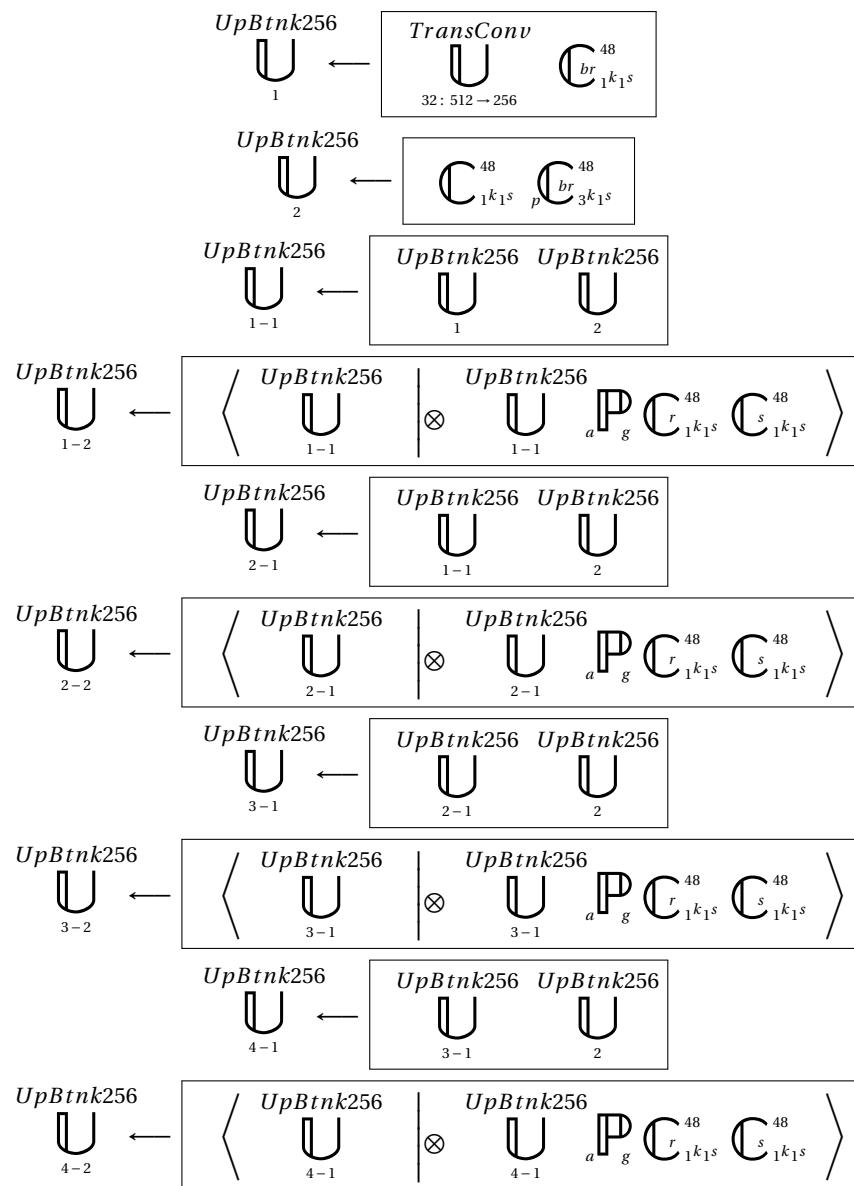


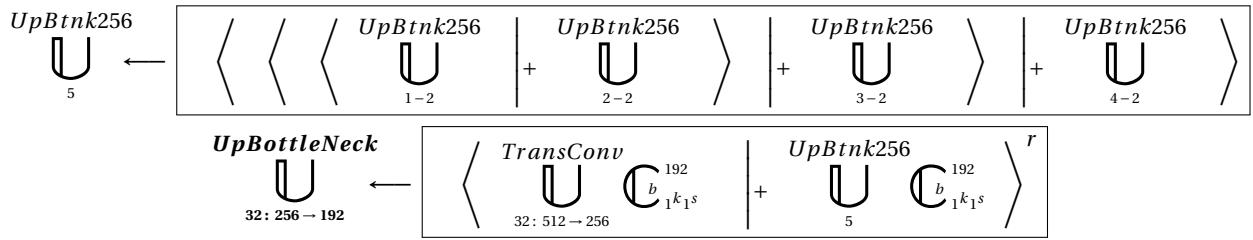


**Rekonstrukcja nr 1 - 256 map cech, rozdzielcość przestrzenna 32x32 (TransConv256):**



**Przetworzenie nadpróbkowanych cech TransConv256 przez blok UpBottleNeck192:**

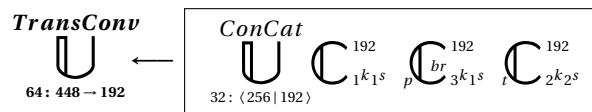




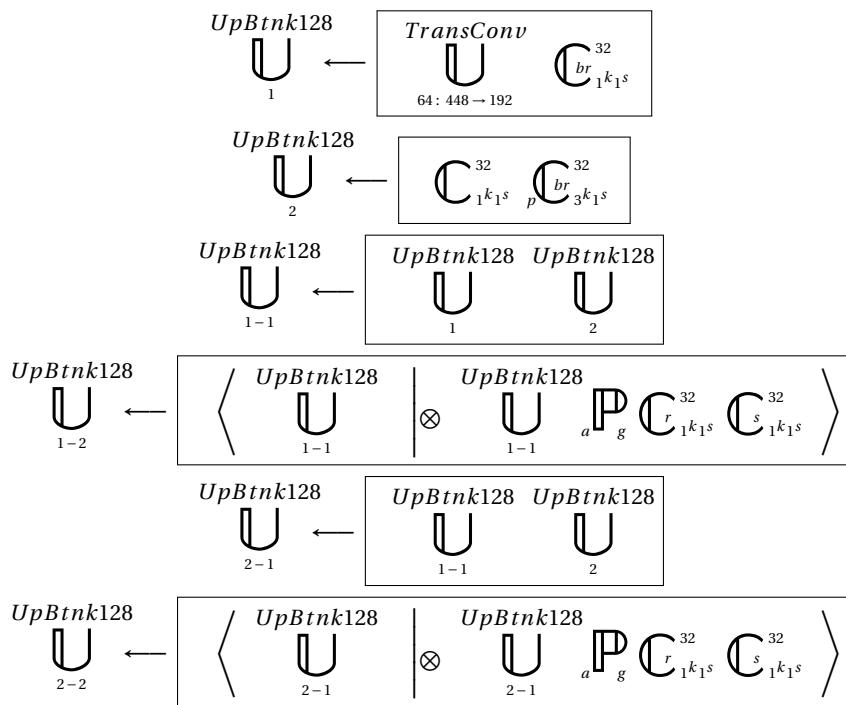
Konkatenacja nr 1 - 448 map cech, rozdzielcość przestrzenna 32x32 (*ConCat448*):

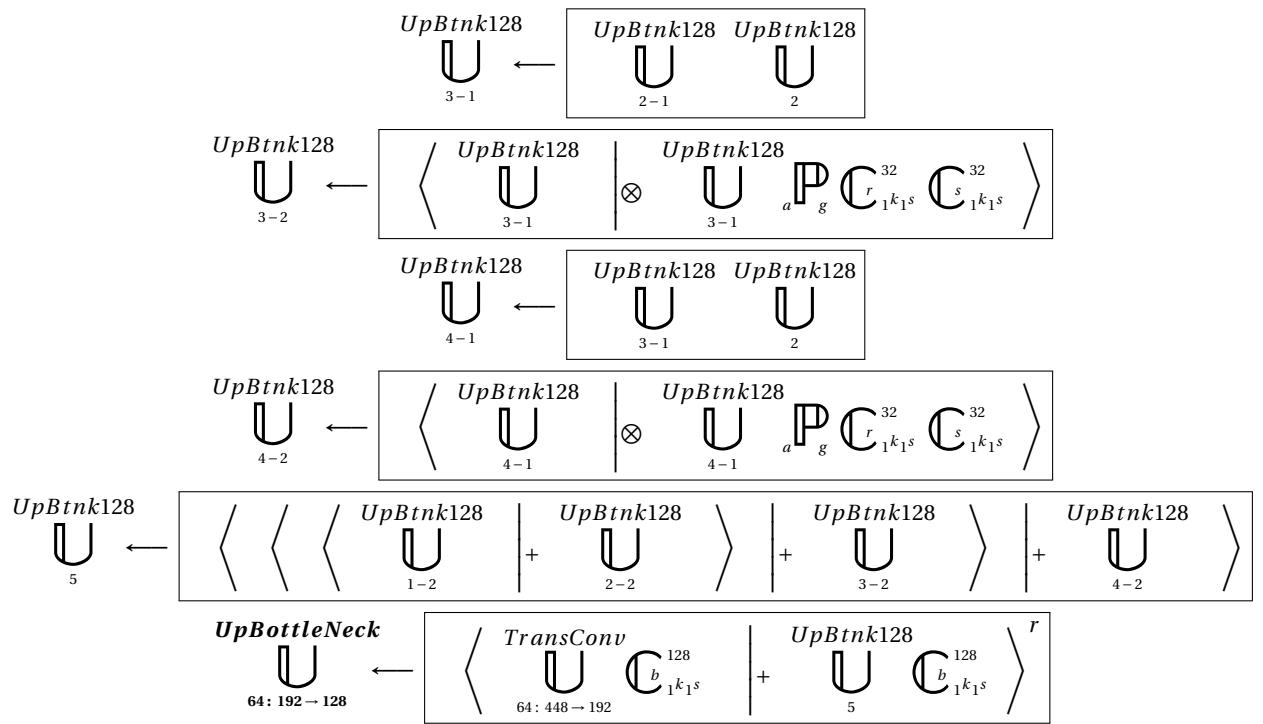


Rekonstrukcja nr 2 - 192 mapy cech, rozdzielcość przestrzenna 64x64 (*TransConv192*):

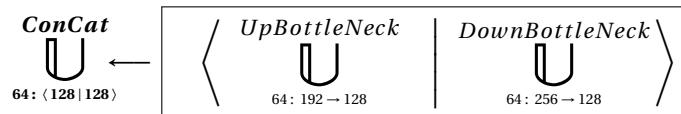


Przetworzenie nadpróbkowanych cech *TransConv192* przez blok *UpBottleNeck128*:

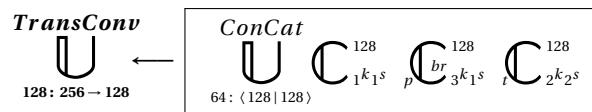




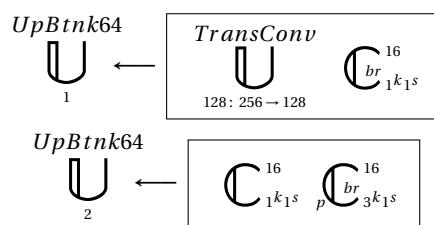
**Konkatenacja nr 2 - 256 map cech, rozdzielcość przestrzenna  $64x64$  (ConCat256):**

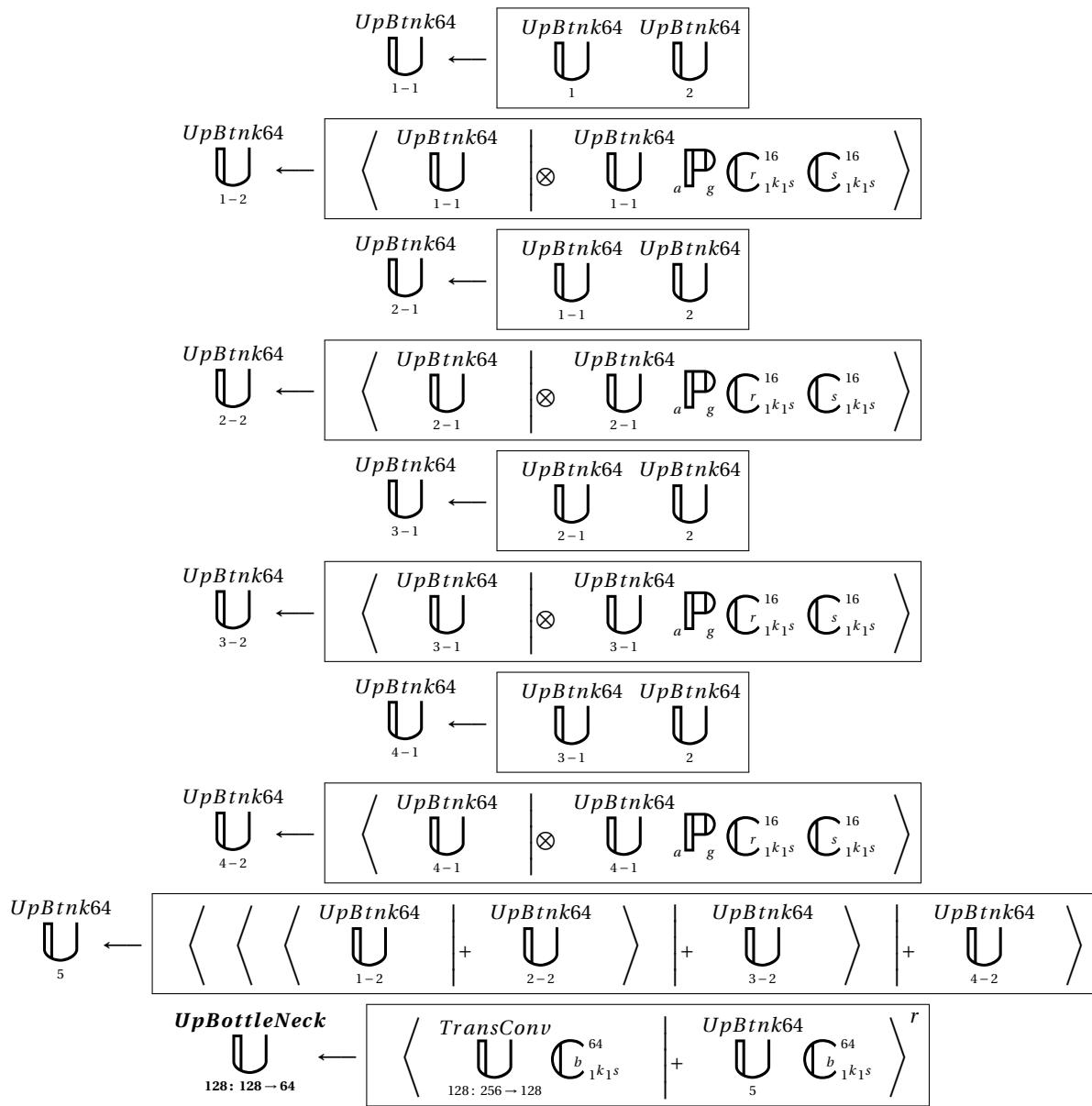


**Rekonstrukcja nr 3 - 128 map cech, rozdzielcość przestrzenna  $128x128$  (TransConv128):**

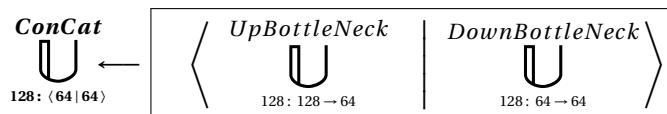


**Przetworzenie nadpróbkowanych cech  $TransConv128$  przez blok  $UpBottleNeck64$ :**

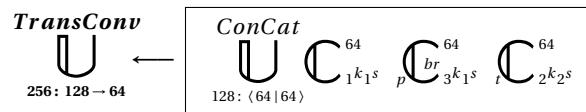




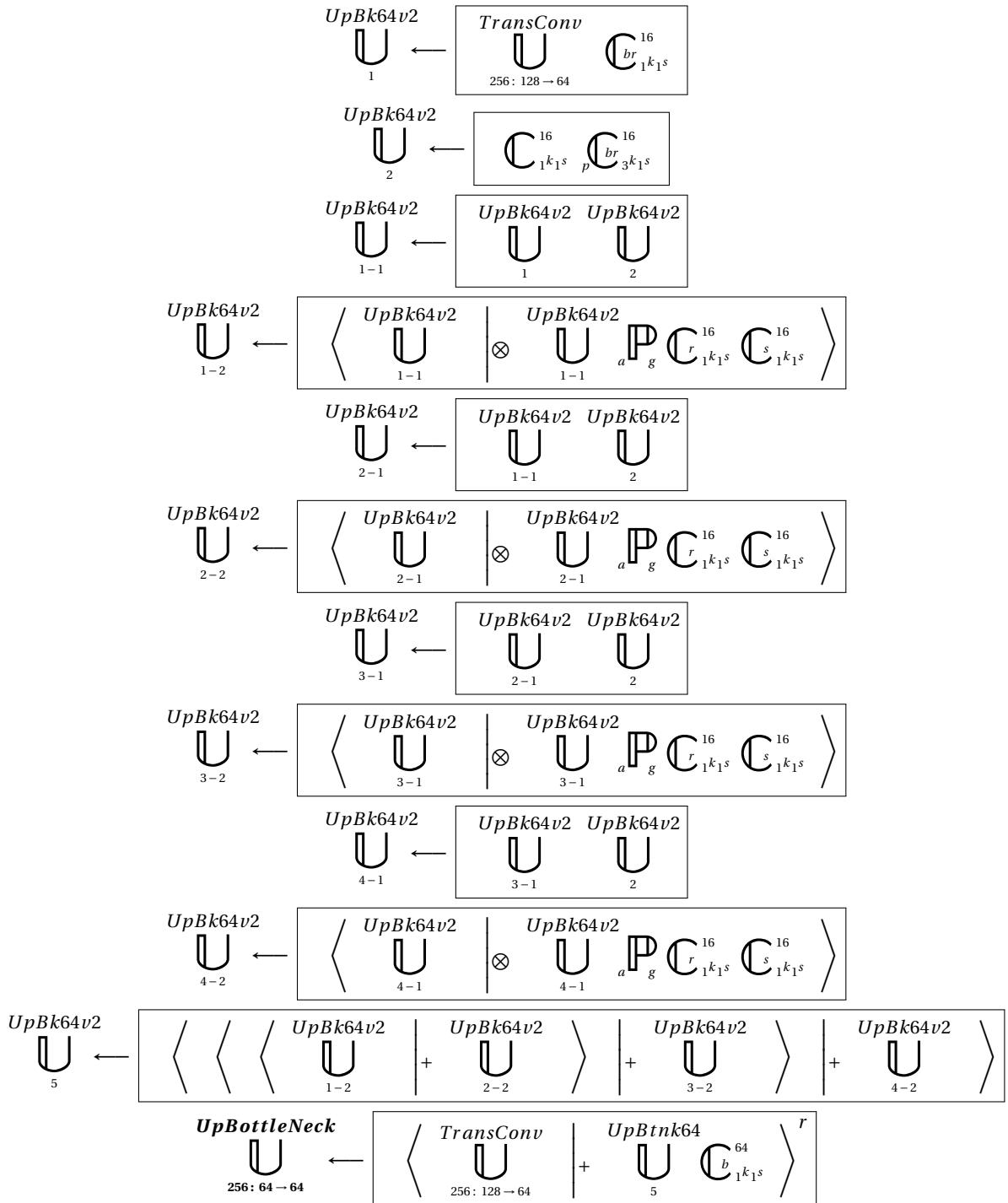
**Konkatenacja nr 3 - 128 map cech, rozdzielcość przestrzenna 128x128 (ConCat128):**



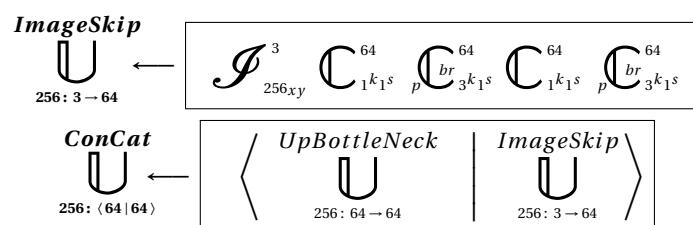
**Rekonstrukcja nr 4 - 64 mapy cech, rozdzielcość przestrzenna 256x256 (TransConv64):**



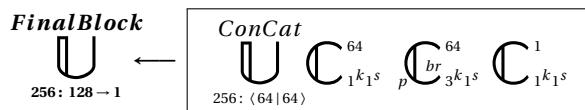
**Przetworzenie nadpróbkowanych cech *TransConv64* przez blok *UpBottleNeck64v2*:**



**Konkatenacja nr 4 - 128 map cech, rozdzielcość przestrzenna 256x256 (*ConCat128*):**



**Finalny blok - jedna mapa cech, rozdzielcość przestrzenna 256x256 (*FinalBlock1*) :**



Sieć *GML-Net* składa się z kilku podstawowych bloków przetwarzania stworzonych z wartstw konwolucyjnych, grupujących, aktywacyjnych i normalizacyjnych. Pierwszym z nich i najbardziej podstawowym jest *LightConvBlock*, czyli blok splotowy, wykorzystujący konwolucję separowalną wgębnie oraz konwolucję z filtrem o rozmiarze  $1x1$  w celu efektywnego obliczeniowo tworzenia map cech.

### Kod źródłowy 22.

```

1 class LightConvBlock(nn.Module):
2     def __init__(self, in_ch, out_ch, k_size, pad, bn_flag):
3         super(LightConvBlock, self).__init__()
4         self.conv1x1 = nn.Conv2d(in_ch, out_ch, 1, stride=1, padding=0, bias=False)
5         self.conv2 = nn.Conv2d(out_ch, out_ch, k_size, stride=1, padding=pad,
6                             bias=False, groups=out_ch)
7         self.bn = nn.BatchNorm2d(out_ch)
8         self.relu = nn.ReLU(inplace=True)
9         self.bn_flag = bn_flag
10
11    def forward(self, in_tensor):
12        out_tens = self.conv1x1(in_tensor)
13        out_tens2 = self.conv2(out_tens)
14        out_tens2 = self.bn(out_tens2) if self.bn_flag else out_tens2
15        return self.relu(out_tens2)

```

Drugim podstawowym blokiem sieci *GML-Net* jest *UpsamplBlock*, czyli blok wykorzystywany w dekoderze, którego zadaniem jest nadpróbkowanie tensorów o niższej rozdzielcości przestrzennej do wyższej rozdzielcości. W tym celu blok ten korzysta z zaprezentowanego powyżej bloku *LightConvBlock* oraz z konwolucji transponowanej o filtrze rozmiaru 2 i o kroku filtra równym 2.

### Kod źródłowy 23.

```

1 class UpsamplBlock(nn.Module):
2     def __init__(self, in_ch, out_ch, k_size, pad, bn_flag):
3         super(UpsamplBlock, self).__init__()

```

```

4 self.cb = LightConvBlock(in_ch, out_ch, k_size, pad, bn_flag)
5 self.conv_up = nn.ConvTranspose2d(out_ch, out_ch, kernel_size=2, stride=2)
6
7 def forward(self, in_tensor):
8     return self.conv_up(self.cb(in_tensor))

```

Kolejnym blokiem wykorzystywanym w sieci opisywanej w bieżącym rozdziale jest blok *Conv1x1*, w ramach którego zaimplementowana została konwolucja z filtrem o rozmiarze *1x1*, normalizacja partiami oraz funkcja aktywacji *ReLU*. Głównym zadaniem realizowanym przez blok *Conv1x1* jest redukcja liczby map cech w odpowiednich częściach sieci neuronowej *GML-Net*

### Kod źródłowy 24.

```

1 class Conv1x1(nn.Module):
2     def __init__(self, in_channels, out_channels, stride=1, groups=1):
3         super(Conv1x1, self).__init__()
4         self.conv1x1 = nn.Conv2d(in_channels, out_channels, 1, stride=stride,
5                               padding=0, bias=False, groups=groups)
6         self.bn = nn.BatchNorm2d(out_channels)
7         self.relu = nn.ReLU(inplace=True)
8     def forward(self, x):
9         return self.relu(self.bn(self.conv1x1(x)))

```

Blok *Conv1x1Linear* jest modyfikacją bloku *Conv1x1* nieimplementującą funkcji aktywacji oraz niedopuszczającą do zmiany parametru *groups* konwolucji *1x1*.

### Kod źródłowy 25.

```

1 class Conv1x1Linear(nn.Module):
2     def __init__(self, in_channels, out_channels, stride=1, bn=True):
3         super(Conv1x1Linear, self).__init__()
4         self.conv = nn.Conv2d(in_channels, out_channels, 1, stride=stride,
5                           padding=0, bias=False)
6         self.bn = nn.BatchNorm2d(out_channels) if bn else None
7     def forward(self, x):
8         x = self.conv(x)
9         x = self.bn(x) if self.bn is not None else x
10        return x

```

**Kod źródłowy 26.**

```

1 class LightConv3x3(nn.Module):
2     def __init__(self, in_channels, out_channels):
3         super(LightConv3x3, self).__init__()
4         self.conv1x1 = nn.Conv2d(in_channels, out_channels, 1, stride=1,
5                               padding=0, bias=False)
6         self.conv2 = nn.Conv2d(out_channels, out_channels, 3, stride=1, padding=1,
7                               bias=False, groups=out_channels)
8         self.bn = nn.BatchNorm2d(out_channels)
9         self.relu = nn.ReLU(inplace=True)
10    def forward(self, x):
11        return self.relu(self.bn(self.conv2(self.conv1x1(x))))

```

Blok *ChannelGate* implementuje bramkę agregacji sieci *OSNet*, jego zadaniem jest połączenie w sposób dynamiczny cech pochodzących z różnych skal poprzez zastosowanie oddzielnej sieci neuronowej o trenowalnych wagach.

**Kod źródłowy 27.**

```

1 class ChannelGate(nn.Module):
2     def __init__(self, in_ch, num_gates=None, gate_act='sigmoid', red=16,
3                  lyr_norm=False):
4         super(ChannelGate, self).__init__()
5         num_gates = in_ch if num_gates is None else num_gates
6         self.global_avgpool = nn.AdaptiveAvgPool2d(1)
7         self.fc1 = nn.Conv2d(in_ch, in_ch // red, kernel_size=1)
8         self.norm1 = nn.LayerNorm((in_ch // red, 1, 1)) if lyr_norm else None
9         self.relu = nn.ReLU(inplace=True)
10        self.fc2 = nn.Conv2d(in_ch // red, num_gates, kernel_size=1)
11        self.gate_act = nn.Sigmoid() if gate_act == 'sigmoid' else \
12          nn.ReLU(inplace=True) if gate_act == 'relu' else None
13    def forward(self, x):
14        input = x
15        x = self.fc1(self.global_avgpool(x))
16        x = self.norm1(x) if self.norm1 is not None else x
17        x = self.fc2(self.relu(x))
18        x = self.gate_act(x) if self.gate_act is not None else x
19        return input * x

```

Ostatnim istotnym elementem sieci *GML-Net* jest blok *BottleNeck\_Block*, który analizuje

cechy pochodzące z sieci *Wide ResNet-50-2* (lub z bloku *UpsamplBlock*) dla różnych skal, a następnie agreguje je przy pomocy bramki *ChannelGate*.

### Kod źródłowy 28.

```

1 class BottleNeck_Block(nn.Module):
2     def __init__(self, in_ch, o_ch, IN=False, btlnc_k_red=4, **kwargs):
3         super(BottleNeck_Block, self).__init__()
4         mch = o_ch // btlnc_k_red
5         self.conv1 = Conv1x1(in_ch, mch)
6         self.conv2a = LightConv3x3(mch, mch)
7         self.conv2b = nn.Sequential(LightConv3x3(mch, mch),
8                                     LightConv3x3(mch, mch))
9         self.conv2c = nn.Sequential(LightConv3x3(mch, mch),
10                                LightConv3x3(mch, mch),
11                                LightConv3x3(mch, mch))
12        self.conv2d = nn.Sequential(LightConv3x3(mch, mch),
13                                LightConv3x3(mch, mch),
14                                LightConv3x3(mch, mch),
15                                LightConv3x3(mch, mch))
16        self.gate = ChannelGate(mch)
17        self.conv3 = Conv1x1Linear(mch, o_ch)
18        self.downsample = Conv1x1Linear(in_ch, o_ch) if in_ch != o_ch else None
19        self.IN = nn.InstanceNorm2d(o_ch, affine=True) if IN else None
20        self.relu = nn.ReLU(inplace=True)

```

### Kod źródłowy 29.

```

1 def BottleNeck_forward(self, x):
2     inpt = x
3     x1 = self.conv1(x)
4     x2a = self.conv2a(x1)
5     x2b = self.conv2b(x1)
6     x2c = self.conv2c(x1)
7     x2d = self.conv2d(x1)
8     x2 = self.gate(x2a) + self.gate(x2b) + self.gate(x2c) + self.gate(x2d)
9     x3 = self.conv3(x2)
10    inpt = self.downsample(inpt) if self.downsample is not None else inpt
11    out = x3 + inpt
12    out = self.IN(out) if self.IN is not None else out
13    return self.relu(out)
14 BottleNeck_Block.forward = BottleNeck_forward

```

Jak już wcześniej wspominano sieć *GML-Net* w części związanej z ekstrakcją cech opiera się na sieci *Wide ResNet-50-2* wytrenowanej na zbiorze *ImageNet*. Nie korzysta jednak z całości jej architektury - zatrzymuje się na poziomie 1024 map cech, po to aby ograniczyć liczbę trenowalnych parametrów. Proces ekstrakcji cech przebiega w czterech etapach:

1. Ekstrakcja są 64 map cech o rozdzielczości przestrzennej  $128x128$ .
2. Ekstrakcja są 256 map cech o rozdzielczości przestrzennej  $64x64$ .
3. Ekstrakcja są 512 map cech o rozdzielczości przestrzennej  $32x32$ .
4. Ekstrakcja są 1024 map cech o rozdzielczości przestrzennej  $16x16$ .

W ramach każdego z tych etapów wyekstraktowane cechy są przesyłane przy pomocy połączeń rezydualnych do bloków *BottleNeck\_Block* po czym tak przetworzone są konkatenowane z blokami *BottleNeck\_Block* uzyskanymi w ramach pracy dekodera.

### Kod źródłowy 30.

```

1 class GML_Net(nn.Module):
2     def __init__(self, layrs_dict):
3         super().__init__()
4         [setattr(self, l_name, l_def) for l_name, l_def in layrs_dict.items()]
5
6     rs_lyrs = list(models.wide_resnet50_2(pretrained=True).children())
7     net_ls = OrderedDict({})
8     net_ls['l0'] = nn.Sequential(*rs_lyrs[:3])
9     net_ls['l1'] = nn.Sequential(*rs_lyrs[3:5])
10    net_ls['l2'] = rs_lyrs[5]
11    net_ls['l3'] = rs_lyrs[6]
12
13    net_ls['bt_nk0'] = BottleNeck_Block(64, 64)
14    net_ls['bt_nk1'] = BottleNeck_Block(256, 128)
15    net_ls['bt_nk2'] = BottleNeck_Block(512, 256)
16    net_ls['bt_nk3'] = BottleNeck_Block(1024, 512)

```

### Kod źródłowy 31.

```

1 def GML_Net_fwd_enc(self, input):
2     l0 = self.l0(input)
3     bt_l0 = self.bt_nk0(l0)
4     l1 = self.l1(l0)
5     bt_l1 = self.bt_nk1(l1)

```

```

6   l2 = self.l2(l1)
7   bt_l2 = self.bt_nk2(l2)
8   l3 = self.l3(l2)
9   bt_l3 = self.bt_nk3(l3)
10
11  return bt_l0, bt_l1, bt_l2, bt_l3

```

W części dekodera związanej z rekonstrukcją, mapy cech są nadpróbkowane przy pomocy *UpsamplBlock* po czym są przetwarzane przez *BottleNeck\_Block* i konkatenowane z cechami pochodząymi z enkodera i w ten sposób trafiają do wyższych warstw nadpróbkujących.

### Kod źródłowy 32.

```

1 n_class = 1
2 net_ls ['up_smpl_0'] = UpsamplBlock(512, 256, 1, 0, False)
3 net_ls ['up_smpl_1'] = UpsamplBlock(448, 192, 3, 1, True)
4 net_ls ['up_smpl_2'] = UpsamplBlock(256, 128, 3, 1, True)
5 net_ls ['up_smpl_3'] = UpsamplBlock(128, 64, 3, 1, True)
6 net_ls ['up_bt_nk0'] = BottleNeck_Block(256, 192)
7 net_ls ['up_bt_nk1'] = BottleNeck_Block(192, 128)
8 net_ls ['up_bt_nk2'] = BottleNeck_Block(128, 64)
9 net_ls ['up_bt_nk3'] = BottleNeck_Block(64, 64)
10 net_ls ['cb_in0'] = LightConvBlock(3, 64, 3, 1, True)
11 net_ls ['cb_in1'] = LightConvBlock(64, 64, 3, 1, True)
12 net_ls ['cb_fin0'] = LightConvBlock(128, 64, 3, 1, True)
13 net_ls ['out_conv'] = nn.Conv2d(64, n_class, 1)

```

### Kod źródłowy 33.

```

1 def GML_Net_fwd_dec(self, bt_l0, bt_l1, bt_l2, bt_l3):
2     up_l0 = self.up_smpl_0(bt_l3)
3     cat_lyr0 = torch.cat([self.up_bt_nk0(up_l0), bt_l2], dim=1)
4     up_l1 = self.up_smpl_1(cat_lyr0)
5     cat_lyr1 = torch.cat([self.up_bt_nk1(up_l1), bt_l1], dim=1)
6     up_l2 = self.up_smpl_2(cat_lyr1)
7     cat_lyr2 = torch.cat([self.up_bt_nk2(up_l2), bt_l0], dim=1)
8     up_l3 = self.up_smpl_3(cat_lyr2)
9
10    return up_l3

```

Po przejściu przez wszystkie warstwy dekodujące sieci *GML-Net* liczba map cech zostaje zredukowana, przy pomocy bloków *LightConvBlock*, do jednej. Dzieje się tak, gdyż celem sieci jest wygenerowanie maski, która po przetworzeniu przy pomocy funkcji *sigmoid*, wskaże prawdopodobieństwo, że dany piksel reprezentuje budynek.

### Kod źródłowy 34.

```

1 def GML_Net_forward(self, input):
2     bt_l0, bt_l1, bt_l2, bt_l3 = GML_Net_fwd_enc(self, input)
3
4     up_l3 = GML_Net_fwd_dec(self, bt_l0, bt_l1, bt_l2, bt_l3)
5     cat_lyr3 = torch.cat([self.up_bt_nk3(up_l3), self.cb_in1(self.cb_in0(input))], dim=1)
6
7     fin_lyr = self.out_conv(self.cb_fin0(cat_lyr3))
8     fin_pred = torch.squeeze(fin_lyr)
9     return fin_pred
10
11 GML_Net.forward = GML_Net_forward

```

Po ustrukturyzowaniu procesu przetwarzania danych uczących (walidacyjnych) i po zdefiniowaniu finalnej architektury sieci *GML-Net*, kolejnym istotnym krokiem w procesie tworzenia głębokiej sieci neuronowej jest usystematyzowanie procesu jej trenowania.

#### 4.4. Trenowanie sieci

Pierwszym krokiem definicji procesu uczenia się głębokiej sieci neuronowej jest wybór odpowiedniej, do zadania realizowanego przez daną sieć neuronową, funkcji strat. W przypadku sieci *GML-Net* zdecydowano się, po przeanalizowaniu dostępnej literatury badawczej, iż sieć ta zostanie wytrenowana przy pomocy funkcji strat stanowiącej ważoną sumę binarnej entropii skrośnej, *Dice Loss* oraz *Lovász Hinge Loss*. Po przeprowadzeniu licznych eksperymentów, ustalono iż optymalne wagie poszczególnych części funkcji strat to: 0,3 dla *BCE*, 0,4 dla *DL* oraz 0,3 dla *LHL*. W ten sposób powstała finalna funkcja strat o następującej definicji:

$$LF = 0,3 \text{ BCE} + 0,4 \text{ DL} + 0,3 * LHL \quad (19)$$

### Kod źródłowy 35.

```

1 def bce_loss_func(pred, target, stats, bce_wght, f_loss):
2     if bce_wght > 0.0:
3         bce_loss = nn.functional.binary_cross_entropy_with_logits(pred, target)

```

```

4 stats['BCE Loss'] += bce_loss.data.cpu().numpy() * target.size(0)
5 f_loss += (bce_loss * bce_wght)
6
7 return f_loss

```

Binarna entropia skrośna jest wyliczana przy pomocy funkcji pochodzącej z biblioteki `torch.nn.functional` o nazwie `binary_cross_entropy_with_logits`, zgodnie ze wzorem zaprezentowanym w podrozdziale 2.3 niniejszej pracy.

### Kod źródłowy 36.

```

1 def dice_loss_func(pred, target, smth, stats, dice_wght, f_loss):
2     pred = torch.sigmoid(pred)
3     prd2 = pred > loss_thres
4     trgt2 = target > loss_thres
5     intrs = (prd2 * trgt2).sum([1, 2])
6     f1s = torch.True_divide(2. * intrs + smth, prd2.sum([1, 2]) + trgt2.sum([1, 2]) + smth)
7
8     if dice_wght > 0.0:
9         dice_loss = 1 - f1s
10        dice_mean_loss = dice_loss.mean()
11        stats['Dice Loss'] += dice_loss.sum().data.cpu().numpy()
12        f_loss += (dice_mean_loss * dice_wght)
13
14 return f_loss, f1s, prd2, trgt2, intrs

```

Implementacja funkcji *Dice Loss* zastosowana w bieżącym rozdziale opiera się na wyliczeniu metryki *F1 Score* a następnie odjęciu jej od wartości 1, po to by sieć ucząc się maksymalizowała wartość współczynnika podobieństwa Sørensena.

### Kod źródłowy 37.

```

1 def lovasz_grad(gt_sorted):
2     p = len(gt_sorted)
3     gts = gt_sorted.sum()
4     intersection = gts - gt_sorted.float().cumsum(0)
5     union = gts + (1 - gt_sorted).float().cumsum(0)
6     jaccard = 1.0 - intersection / union
7
8     if p > 1:
9         jaccard[1:p] = jaccard[1:p] - jaccard[0:-1]
10    return jaccard

```

**Kod źródłowy 38.**

```

1 def lovasz_hinge_loss(org_pred, org_target, stats, lhl_wght, f_loss):
2     if lhl_wght > 0.0:
3         pred = org_pred.view(-1)
4         target = org_target.view(-1)
5
6     if len(target) == 0:
7         return pred.sum() * 0.
8
9     signs = 2. * target.float() - 1.0
10    errors = (1. - pred * signs)
11    errors_sorted, perm = torch.sort(errors, dim=0, descending=True)
12    perm = perm.data
13    gt_sorted = target[perm]
14    grad = lovasz_grad(gt_sorted)
15    lh_loss = torch.dot(nn.functional.relu(errors_sorted), grad)
16    stats['LH Loss'] += lh_loss.data.cpu().numpy() * org_target.size(0)
17    f_loss += (lh_loss * lhl_wght)
18
return f_loss

```

Funkcja *Lovász Hinge Loss* została zaimplementowana zgodnie z jej definicją szeroko opisaną w pracy *The Lovasz Hinge: A Novel Convex Surrogate for Submodular Losses* napisanej przez Jiaqiana Yu i Matthew B. Blaschko.

**Kod źródłowy 39.**

```

1 def gaussian(w_s, sigma):
2     gauss = torch.Tensor([exp(-(x - w_s // 2) ** 2 /
3                           float(2 * sigma ** 2)) for x in range(w_s)])
4
return gauss/gauss.sum()
5
6 def create_window(w_s, channel, img1):
7     _1D_window = gaussian(w_s, 1.5).unsqueeze(1)
8     _2D_window = _1D_window.mm(_1D_window.t()).float()
9     _2D_window = _2D_window.unsqueeze(0).unsqueeze(0)
10    window = _2D_window.expand(channel, 1, w_s, w_s).contiguous()
11
12    if img1.is_cuda:
13        window = window.cuda(img1.get_device())
14    window = window.type_as(img1)
15
16
return window

```

Poza funkcją strat, istotnym elementem treningu głębskiej sieci neuronowej, jest też właściwa definicja metryk oceniających skuteczność generowanych przez tą sieć predykcji. W przypadku sieci *GML-Net*, po analizie literatury dotyczącej detekcji budynków na zdjęciach lotniczych, zdecydowano się na następujące metryki:

- ogólna dokładność (*OA*),
- współczynnik podobieństwa Jaccarda (*IoU*)
- *F1 Score (F1S)*,
- indeks podobieństwa strukturalnego (*SSIM*).

### Kod źródłowy 40.

```

1 def ssim_index(img1, img2, w_s=11):
2     img1 = img1[...].float()
3     img2 = img2[...].float()
4     _, channel, _, _ = img1.size()
5     window = create_window(w_s, channel, img1)
6     mu1 = nn.functional.conv2d(img1, window, padding = w_s // 2,
7                               groups=channel)
8     mu2 = nn.functional.conv2d(img2, window, padding = w_s // 2,
9                               groups=channel)
10    sigma1_sq = nn.functional.conv2d(img1 * img1, window, padding = w_s // 2,
11                                    groups=channel) - mu1.pow(2)
12    sigma2_sq = nn.functional.conv2d(img2 * img2, window, padding = w_s // 2,
13                                    groups=channel) - mu2.pow(2)
14    sigma12 = nn.functional.conv2d(img1 * img2, window, padding = w_s // 2,
15                                  groups=channel) - mu1 * mu2
16    ssim_map = ((2 * mu1 * mu2 + 0.01 ** 2) * (2 * sigma12 + 0.03 ** 2)
17 ) / ((mu1.pow(2) + mu2.pow(2) + 0.01 ** 2) * (sigma1_sq + sigma2_sq +
18                                     0.03 **2))
19    return ssim_map.mean()
```

Każda z przedstawionych powyżej metryk została zaimplementowana zgodnie z odpowiednimi wzorami, które zostały szeroko omówione w podrozdziale 2.3. niniejszej pracy.

### Kod źródłowy 41.

```

1 def calc_net_metrics(prd2, trgt2, intrs, smth, stats, f_loss, target):
2     union = (prd2 + trgt2).sum([1, 2])
3     iou = torch.True_divide(intrs + smth, union + smth)
4     acc = torch.True_divide(intrs + (~prd2 * ~trgt2).sum([1, 2]),
5                             prd2[0, ...].numel())
```

```

6   ssim = ssim_index(prd2, trgt2)
7
8   stats['Final Loss'] += f_loss.data.cpu().numpy() * target.size(0)
9   stats['OA'] += acc.sum().cpu().numpy()
10  stats['IoU'] += iou.sum().cpu().numpy()
11  stats['SSIM'] += ssim.data.cpu().numpy() * trgt2.size(0)

```

Wyliczone wartości komponentów łącznej funkcji strat oraz wartości poszczególnych metryk podlegają zapisowi w słowniku (*stats*). Taki zapis jest realizowany dla każdej partii (*batcha*) przykładów uczących / walidacyjnych w danej epoce obliczeniowej.

### Kod źródłowy 42.

```

1 def loss_calc(pred, target, stats, wghts_dict, loss_thres):
2     f_loss = 0.0
3     bce_wght = wghts_dict['bce']
4     lhl_wght = wghts_dict['lhl']
5     dice_wght = wghts_dict['dice']
6     smth = 1.0
7
8     f_loss = bce_loss_func(pred, target, stats, bce_wght, f_loss)
9     f_loss = lovasz_hinge_loss(pred, target, stats, lhl_wght, f_loss)
10    f_loss, f1s, prd2, trgt2, intrs = dice_loss_func(pred, target, smth, stats,
11                                                    dice_wght, f_loss)
12    stats['F1S'] += f1s.sum().cpu().numpy()
13
14    calc_net_metrics(prd2, trgt2, intrs, smth, stats, f_loss, target)
15
16 return f_loss

```

Po zakończenie danej epoki wartości komponentów funkcji strat są sumowane, a wartości metryk uśredniane, tak żeby móc je zaprezentować jako miernik średniej jakości predykcji w danej epoce obliczeniowej.

### Kod źródłowy 43.

```

1 def epoch_stats(stats, epoch_samples, phase, res_dict):
2     prnt_text = phase + ": "
3
4     for k in list(stats.keys())[:-4]:
5         curr_stat = stats[k] / epoch_samples
6         if k in res_dict[phase]:

```

```

7     res_dict[phase][k].append(curr_stat)
8 else:
9     res_dict[phase][k] = [curr_stat]
10    prnt_text += "{}: {:.4f}, ".format(k, curr_stat)
11    prnt_text = prnt_text[:-2] + "\n"
12
13 for k in list(stats.keys())[-4:]:
14     curr_stat = stats[k] / epoch_samples
15     if k in res_dict[phase]:
16         res_dict[phase][k].append(curr_stat)
17     else:
18         res_dict[phase][k] = [curr_stat]
19     prnt_text += "{}: {:.2%}, ".format(k, curr_stat)
20 print(prnt_text[:-2])

```

Jeżeli w danej epoce na zbiorze walidacyjnym uda się osiągnąć najniższą odnotowaną dotąd wartość łącznej funkcji strat, to stan sieci *GML-Net*, który umożliwił osiągnięcie takiego rezultatu, jest zapisywany na dysku *Google* wraz z hiperparametrami danego obliczenia.

#### Kod źródłowy 44.

```

1 def save_model(trnd_model, epoch, res_dict):
2     curr_date = datetime.now().strftime('%Y_%m_%d')
3     curr_mdl_name = mdl_name + " - przeliczenie na " + curr_date
4     dict_out = {'model_params': trnd_model.state_dict(), 'optimizer_params':
5                 optimizer_ft.state_dict(), 'scheduler_params':
6                 scheduler_ft.state_dict(), 'configuration': params_dict,
7                 'stats': res_dict, 'Num of epochs': epoch}
8
9     torch.save(dict_out, mdl_path + curr_mdl_name + ".pth")
10
11    dummy_input = torch.randn(1, 3, sample_res, sample_res).cuda()
12    input_names = ["Input tensor"]
13    output_names = ["Output mask"]
14
15    torch.onnx.export(trnd_model, dummy_input, mdl_path + curr_mdl_name +
16                      ".onnx", input_names=input_names,
17                      output_names=output_names, export_params=True)

```

Dzięki zapisowi stanu najlepszego modelu możliwe jest późniejsze uzyskiwanie z niego predykcji oraz w razie potrzeby kontynuacja trenowania.

### Kod źródłowy 45.

```
1 def save_best_res(phase, sched, epoch_loss, best_loss, model, epoch, res_dict,
2                     best_model_wts):
3
4     if phase == 'Valid' and sched:
5         sched.step(epoch_loss)
6
7     print("")
8
9     if phase == 'Valid' and epoch_loss < best_loss:
10        print("Najlepszy jak dotad model!")
11        best_loss = epoch_loss
12        best_model_wts = copy.deepcopy(model.state_dict())
13        save_model(model, epoch, res_dict)
14
15    return best_loss, best_model_wts
```

Rozpoczęcie każdej epoki obliczeniowej wiąże się z deklaracją wartości początkowych komponentów funkcji strat oraz metryk jakości predykcji.

### Kod źródłowy 46.

```
1 def phase_init(phase, optim, model):
2     stats = defaultdict(float)
3     stats['BCE Loss'] = 0.0
4     stats['LH Loss'] = 0.0
5     stats['Dice Loss'] = 0.0
6     stats['Final Loss'] = 0.0
7     stats['OA'] = 0.0
8     stats['IoU'] = 0.0
9     stats['F1S'] = 0.0
10    epoch_samples = 0
11
12    if phase == 'Train':
13        for param_group in optim.param_groups:
14            print("Learning Rate: " + str(param_group['lr']) + '\n')
15            model.train()
16    else:
17        model.eval()
18
19    return stats, epoch_samples
```

Funkcja *calc\_epoch\_loss* w ramach trenowania sieci *GML-Net* odpowiada za bieżące generowanie predykcji z tej sieci oraz porównywanie ich do oczekiwanych predykcji poprzez wyliczenie wartości łącznej funkcji strat oraz wartości poszczególnych metryk.

### Kod źródłowy 47.

```

1 def calc_epoch_loss(phase, optim, model, stats, wghts_dict, loss_thres,
2                     epoch_samples, res_dict):
3     for package in data_loaders[phase]:
4         for inputs, masks in package:
5             inputs = inputs.to(device)
6             masks = masks.to(device)
7             optim.zero_grad()
8
9             with torch.set_grad_enabled(phase == 'Train'):
10                outputs = model(inputs)
11                loss = loss_calc(outputs, masks, stats, wghts_dict, loss_thres)
12
13                if phase == 'Train':
14                    loss.backward()
15                    optim.step()
16
17                epoch_samples += inputs.size(0)
18
19    epoch_stats(stats, epoch_samples, phase, res_dict)
20    return stats['Final Loss'] / epoch_samples

```

Funkcje *train\_init*, *epoch\_init* oraz *print\_epoch\_time* są funkcjami pomocniczymi w procesie trenowania sieci *GML-Net*, odpowiadają one za inicjalizację niektórych parametrów oraz wydruk statystyk przeliczeń w poszczególnych epokach.

### Kod źródłowy 48.

```

1 def train_init(model):
2     best_model_wts = copy.deepcopy(model.state_dict())
3     best_loss = 1e10
4     phases_list = ['Train', 'Valid']
5     res_dict = {'Train': {}, 'Valid': {}}
6
7     return best_loss, phases_list, res_dict
8
9 def epoch_init(epoch, num_ep):
10    print('\nEpoka {}/{}.format(epoch, num_ep))

```

```

11 print(‘-’ * 10)
12 since = time.time()
13 return since
14
15 def print_epoch_time(since):
16     time_e = time.time() – since
17     print(‘Czas przeliczenia bieżacej epoki: {:.0f}m {:.0f}s’.format(time_e // 60,
18                                         time_e % 60))
19     print(‘-’ * 10)

```

Funkcja *train\_model*, jest główną funkcją realizującą trenowanie omawianej w bieżącym rozdziale głębokiej sieci neuronowej. Jej głównym zadaniem jest kierowanie kolejnością obliczeń w ramach poszczególnych epok i faz obliczeniowych (treningowej i walidacyjnej).

### Kod źródłowy 49.

```

1 def train_model(model, optim, sched, device, wghts_dict, loss_thres, num_ep):
2     best_loss, phases_list, res_dict = train_init(model)
3     best_model_wts = model.state_dict()
4
5     for epoch in range(1, num_ep + 1):
6         since = epoch_init(epoch, num_ep)
7         for phase in phases_list:
8             stats, epoch_samples = phase_init(phase, optim, model)
9             epoch_loss = calc_epoch_loss(phase, optim, model, stats, wghts_dict,
10                                         loss_thres, epoch_samples, res_dict)
11            best_loss, best_model_wts = save_best_res(phase, sched, epoch_loss,
12                                         best_loss, model, epoch,
13                                         res_dict, best_model_wts)
14            print_epoch_time(since)
15
16            print(‘\nNajniższa strata wyliczona na zbiorze walidacyjnym: {:.4f}’.format(
17                best_loss))
18
19            model.load_state_dict(best_model_wts)
20            return model, res_dict

```

Po zdefiniowaniu funkcji treningowej, przed przystąpieniem do właściwego procesu trenowania, należy jeszcze wybrać optymalizator, który takie trenowanie będzie realizował a także *scheduler*, który będzie odpowiedzialny za odpowiednie obniżanie stopy uczenia

w trakcie trenowania. Po przeprowadzeniu eksperymentów z kilkoma różnymi optymalizatorami oraz po przeanalizowaniu literatury badawczej, jako optymalizator najlepiej realizujący trenowanie sieci *GML-Net* wybrano optymalizator *SGD* z początkową stopą uczenia na poziomie 0,01, momentum o wartości 0,9 i spadkiem wag równym 0,0005. Jako *scheduler* wybrano *ReduceLROnPlateau*, ze spadkiem uczenia realizowanym poprzez przemnożenie aktualnej stopy uczenia przez 0,5 przy pięciu epokach bez poprawy stopy uczenia na zbiorze walidacyjnym.

### Kod źródłowy 50.

```

1 model_ft = GML_Net(layrs_dict=net_ls)
2 model_ft = model_ft.cuda()
3 optimizer_ft = optim.SGD(model_ft.parameters(), lr=learn_rate,
4                           momentum=momt, weight_decay=wght_dec)
5
6 scheduler_ft = optim.lr_scheduler.ReduceLROnPlateau(optimizer_ft, 'min',
7                                                       factor=gamm,
8                                                       patience=step_s,
9                                                       eps=optim_eps)
10
11 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
12 trnd_model, res_dict = train_model(model_ft, optimizer_ft, scheduler_ft,
13                                     device, wghts_dict, loss_thres,
14                                     epoch_num)
```

### Kod źródłowy 51.

```

1 Epoka 155/200
2 -----
3 Learning Rate: 4.8828125e-06
4
5 Train: BCE Loss: 0.1203, LH Loss: 0.4504, Dice Loss: 0.1532, Final Loss: 0.2325
6     OA: 96.88%, IoU: 78.87%, F1S: 84.68%, SSIM: 95.31%
7
8 Valid: BCE Loss: 0.1165, LH Loss: 0.4679, Dice Loss: 0.1178, Final Loss: 0.2224
9     OA: 97.08%, IoU: 82.42%, F1S: 88.22%, SSIM: 95.46%
10
11 Najlepszy jak dotad model!
12 Czas przeliczenia bieżacej epoki: 7m 31s
```

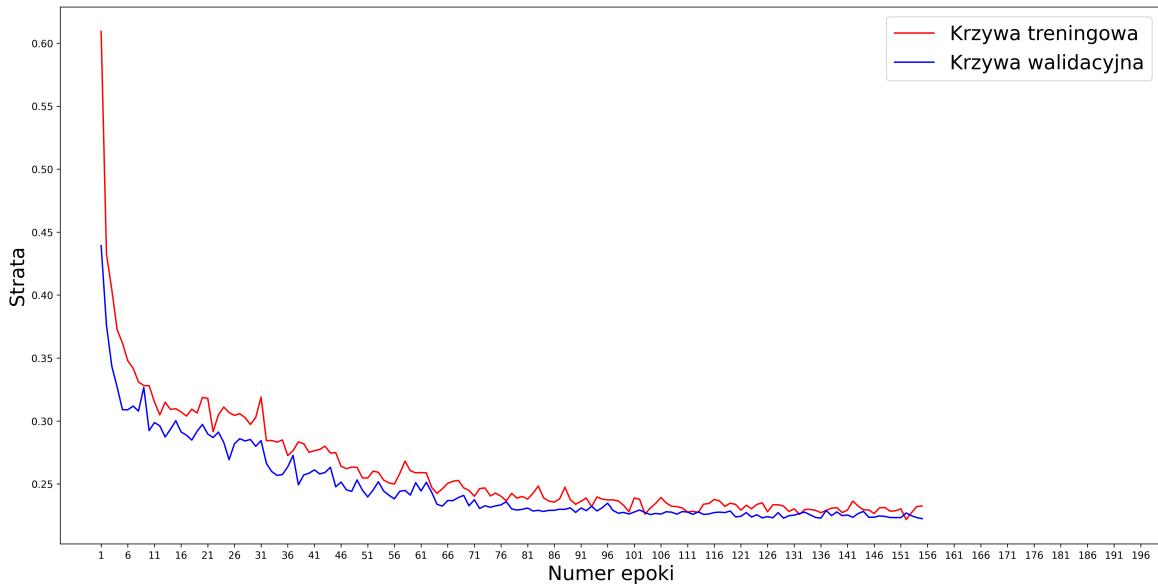
Najlepszy wynik pod kątem łącznej straty na zbiorze walidacyjnym, sieć *GML-Net* uzyskała po 155 epokach obliczeniowych. Wartość łącznej funkcji strat wyniosła wtedy 0,2224,

a uzyskane metryki były na poziomie:  $OA = 97,08\%$ ,  $IoU = 82,42\%$ ,  $F1S = 88,22\%$ ,  $SSIM = 95,46\%$ . Uzyskane wartości metryk można uznać za satysfakcyjne patrząc przez pryzmat wyników uzyskiwanych przez modele zaprezentowane w literaturze dotyczącej detekcji budynków na zdjęciach lotniczych.

### Kod źródłowy 52.

```

1 res_dict = model_dict['stats']
2 train_loses = res_dict['Train']['Final Loss']
3 valid_losses = res_dict['Valid']['Final Loss']
4
5 plt.plot(*zip([(i + 1, loss) for i, loss in enumerate(train_loses)]), '-r')
6 plt.plot(*zip([(i + 1, loss) for i, loss in enumerate(valid_losses)]), '-b')
7
8 plt.legend(['Krzywa treningowa', 'Krzywa walidacyjna'], fontsize=20)
9 plt.title('Krzywe uczenia sie sieci GML-Net', fontsize=22)
10 plt.xlabel('Numer epoki', fontsize=20)
11 plt.xticks(range(1, epoch_num + 1))
12 plt.ylabel('Strata', fontsize=20)
13 plt.gcf().set_facecolor("white")
14 plt.gcf().set_size_inches(20, 10)
15 plt.gca().set_xticks(plt.gca().get_xticks()[::5])
16 plt.savefig('Krzywe uczenia.png', dpi=300, bbox_inches='tight')
```



Rysunek 4.3. Krzywe uczenia sieci *GML-Net*

Jak widać na powyższym rysunku 4.3, przez cały okres trenowania sieci *GML-Net*, wartość łącznej funkcji straty na zbiorze walidacyjnym utrzymywała się poniżej wartości łącznej

funkcji straty na zbiorze treningowym. Można to tłumaczyć tym, iż zbiór walidacyjny jest kilkukrotnie mniejszy od zbioru treningowego, może więc charakteryzować się mniejszą zmiennością i stąd generować nieznacznie lepsze wyniki. Co istotne w czasie treningu nie dochodzi do przetrenowania sieci - zastosowanie silnej augmentacji danych, przyczyniło się do uzyskania wystarczającej odporności modelu na ten problem.

W niniejszym rozdziale szczegółowo opisany został proces konstrukcji głębszej sieci neuronowej o nazwie *GML-Net* od wczytywania i przetwarzania danych, przez architekturę modelu aż po sposób jego trenowania. W kolejnym rozdziale zaprezentowana zostanie analiza wyników generowanych przez sieć *GML-Net*, w tym sposób łączenia predykcji masek fragmentów obrazu w jedną łączną maskę dla całego obrazu o wysokiej rozdzielczości.

## 5. Analiza wyników

Zaprezentowany w poprzednim rozdziale model *GML-Net* można by uznać za ukończony, gdyby operował on na obrazach o rozdzielczości  $5000 \times 5000$ . Niestety trenowanie głębokiej sieci neuronowej na obrazach o tak wysokiej rozdzielczości nie jest możliwe ze względu na ograniczoną pojemność kart graficznych, stąd też konieczne było wytrenowanie sieci *GML-Net* na licznych podpróbkach obrazów oryginalnej rozdzielczości. Aby wygenerować finalną predykcję masek dla obrazów o rozdzielczości  $5000 \times 5000$  postanowiono więc wygenerować predykcje dla ich podpróbek o rozdzielczości  $256 \times 256$ , a następnie te podpróbki miały zostać połączone w jedną łączną maskę o wysokiej rozdzielczości. Taki proces należałoby powtórzyć dla wszystkich zdjęć znajdujących się w zbiorze walidacyjnym po to aby móc dokładnie określić finalną jakość predykcji sieci *GML-Net*.

### 5.1. Generowanie finalnych predykcji

Pierwszym etapem generowania finalnych predykcji było wczytanie zapisu najlepszego uzyskanego modelu wraz z odpowiednimi jego hiperparametrami oraz ustawienie go w trybie ewaluacji.

#### Kod źródłowy 53.

```
1 mdl_from_path = True
2
3 if mdl_from_path:
4     mdl_n = "ResNet50_wide_UNet_depth_3_BottleNeck_BS_18.pth"
5     model_dict = torch.load(mdl_path + mdl_n)
6     state_dict = model_dict['model_params']
7     trnd_model = GML_Net(layrs_dict=net_ls)
8     trnd_model.load_state_dict(state_dict)
9     trnd_model.cuda()
10    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
11    trnd_model.eval()
```

#### Kod źródłowy 54.

```
1 def g_prms(imgs_arr, img_num):
2     mask_path = imgs_arr[img_num, 1]
3     image = np.array(Image.open(imgs_arr[img_num, 0]))
4     mask = np.array(Image.open(mask_path))
5     wind_h = sample_res
6     wind_w = sample_res
```

```

7 img_shp = image.shape
8 nm_cls = img_shp[-1] if len(img_shp) > 2 else 1
9 img_h = img_shp[0]
10 img_w = img_shp[1]
11 S_y = sample_res // 2
12 S_x = sample_res // 2
13 return image, mask, wind_h, wind_w, nm_cls, img_h, img_w, S_y, S_x

```

Przy pomocy funkcji *g\_prms* ustalone najważniejsze parametry podziału zdjęć pochodzących ze zbioru walidacyjnego na podpróbki o rozdzielczości  $256 \times 256$ . Postanowiono, iż każde zdjęcie zostanie podzielone na 1444 nakładające się podpróbki, szerokość nakładania się ustalono na połowę wymiaru podpróbki.

### Kod źródłowy 55.

```

1 def get_wndts(img, msk, wnd_h, wnd_w, nm_cls, img_h, img_w, Sy, Sx):
2     G_x = 1 + (img_w - wnd_w) // Sx
3     G_y = 1 + (img_h - wnd_h) // Sy
4     s_l = Sy * img_w * nm_cls
5     s_k = Sx * nm_cls
6     s_b = img_w * nm_cls
7     s_a = 1 * nm_cls
8     img_windows = as_strided(img, shape=(G_y, G_x, wnd_h, wnd_w, nm_cls),
9                               strides=(s_l, s_k, s_b, s_a, 1))
10    return img_windows.reshape((G_y * G_x, wnd_h, wnd_w, nm_cls)), G_x, G_y

```

Przy pomocy funkcji *as\_strided* pochodzącej z biblioteki *numpy.lib.stride\_tricks* przeprowadzono podział każdego ze zdjęć walidacyjnych.

### Kod źródłowy 56.

```

1 imgs_arr = pd.read_csv("m_inf_v.csv", header=None).values
2 img_n = 1
3 img, msk, wnd_h, wnd_w, nm_cls, img_h, img_w, Sy, Sx = g_prms(imgs_arr,
4                                                               img_n)
5 wnd_img, Gx, Gy = get_wndts(img, msk, wnd_h, wnd_w, nm_cls, img_h, img_w,
6                             Sy, Sx)
7 fig, axs = plt.subplots(Gx, Gy)
8 plt.subplots_adjust(wspace=0.1, hspace=0.1)
9 fig.set_size_inches(20, 20)
10
11 for i, ax in enumerate(axs.flat):

```

## 5. Analiza wyników

---

```
12 |     ax.imshow(wnd_img[i])
13 |     ax.axis('off')
14 | plt.savefig('Podprobki.png', dpi=300, bbox_inches='tight')
```

Aby mieć pewność, że podział został przeprowadzony prawidłowo postanowiono zilustrować go przy pomocy mozaiki składającej się z wszystkich podpróbek wybranego zdjęcia (patrz rysunek 5.1).



**Rysunek 5.1.** Wybrane zdjęcie pochodzące ze zbioru walidacyjnego rozbite na 1444 podpórki

Mając do dyspozycji wszystkie 1444 podpróbki zdjęcia bazowego można było przystąpić do generowania predykcji masek dla każdej z tych podpróbek.

### Kod źródłowy 57.

```

1 def get_all_preds(reshp_img):
2     fin_trans = transforms.Compose([transforms.ToTensor(),
3                                     transforms.Normalize(mean_rgbs, std_rgbs)])
4     fin_preds = np.zeros((reshp_img.shape[0], sample_res, sample_res))
5
6     for i in range(0, reshp_img.shape[0], batch_s):
7         if i + batch_s > reshp_img.shape[0]:
8             continue
9         input = reshp_img[i:(i + batch_s), ...]
10        trans_input = torch.zeros((input.shape[0], input.shape[3], input.shape[1],
11                                   input.shape[2]))
12        for j in range(input.shape[0]):
13            trans_input[j, ...] = fin_trans(input[j, ...])
14        with torch.no_grad():
15            pred_base = trnd_model(trans_input.to(device))
16
17        pred_sigm = torch.sigmoid(pred_base).data.cpu().numpy()
18        fin_preds[i:(i + batch_s), ...] = pred_sigm
19
20    return fin_preds

```

Aby móc efektywnie połączyć uzyskane predykcje w jedną łączną maskę o rozdzielczości  $5000 \times 5000$  postanowiono skorzystać z dwuwymiarowych okien Hanna zdefiniowanych szerzej w artykule Nicolasa Pielawskiego oraz Caroliny Wählby *Introducing Hann windows for reducing edge-effects in patch-based image segmentation* [23].

### Kod źródłowy 58.

```

1 hann_wnd_b = np.zeros((sample_res, sample_res), dtype=float)
2 hann_wnd_up = np.zeros((sample_res, sample_res), dtype=float)
3 hann_wnd_down = np.zeros((sample_res, sample_res), dtype=float)
4 hann_wnd_left = np.zeros((sample_res, sample_res), dtype=float)
5 hann_wnd_right = np.zeros((sample_res, sample_res), dtype=float)
6 hann_wnd_up_l = np.zeros((sample_res, sample_res), dtype=float)
7 hann_wnd_down_l = np.zeros((sample_res, sample_res), dtype=float)
8 hann_wnd_up_r = np.zeros((sample_res, sample_res), dtype=float)
9 hann_wnd_down_r = np.zeros((sample_res, sample_res), dtype=float)

```

Autorzy tego artykułu zaproponowali algorytm łączenia podpróbek zdjęć w jedno łączne zdjęcie w czasie realizowania zadań z zakresu semantycznej segmentacji. Ich pomysł

## 5. Analiza wyników

---

opierał się na przemnażaniu predykcji uzyskanych dla poszczególnych podpróbek przez dwuwymiarowe okna Hanna, stanowiące *de facto* macierz wag, która główny nacisk kładzie na predykcje znajdujące się w środku podpróbki - w ten sposób redukowany jest efekt złych predykcji krawędzi.

### Kod źródłowy 59.

```
1 def hann_up_down_left_right(i, j, h_part, v_part):
2     if j < sample_res / 2:
3         hann_wnd_up[j, i] = 0.5 * h_part
4     else:
5         hann_wnd_up[j, i] = 0.25 * h_part * v_part
6
7     if j > sample_res / 2:
8         hann_wnd_down[j, i] = 0.5 * h_part
9     else:
10        hann_wnd_down[j, i] = 0.25 * h_part * v_part
11
12    if i < sample_res / 2:
13        hann_wnd_left[j, i] = 0.5 * v_part
14    else:
15        hann_wnd_left[j, i] = 0.25 * h_part * v_part
16
17    if i > sample_res / 2:
18        hann_wnd_right[j, i] = 0.5 * v_part
19    else:
20        hann_wnd_right[j, i] = 0.25 * h_part * v_part
```

Nicolas Pielawski i Caroline Wählby zaproponowali również modyfikacje bazowego dwuwymiarowego okna Hanna, dla przypadków gdy predykcja pochodzi z podpróbki znajdującej się na krawędzi zdjęcia bazowego, tak żeby w takich przypadkach położyć nacisk na predykcje znajdujące się na krawędziach podpróbki.

### Kod źródłowy 60.

```
1 def hann_up_l_up_r(i, j, h_part, v_part):
2     if i <= sample_res / 2 and j <= sample_res / 2:
3         hann_wnd_up_l[j, i] = 1
4     elif i > sample_res / 2 and j < sample_res / 2:
5         hann_wnd_up_l[j, i] = 0.5 * h_part
6     elif i < sample_res / 2 and j > sample_res / 2:
7         hann_wnd_up_l[j, i] = 0.5 * v_part
8     else:
```

```

9  hann_wnd_up_l[j, i] = 0.25 * h_part * v_part
10
11 if i >= sample_res / 2 and j <= sample_res / 2:
12     hann_wnd_up_r[j, i] = 1
13 elif i < sample_res / 2 and j < sample_res / 2:
14     hann_wnd_up_r[j, i] = 0.5 * h_part
15 elif i > sample_res / 2 and j > sample_res / 2:
16     hann_wnd_up_r[j, i] = 0.5 * v_part
17 else:
18     hann_wnd_up_r[j, i] = 0.25 * h_part * v_part

```

W ten sposób uzyskano 9 wariantów dwuwymiarowego okna Hanna, których zastosowanie zależało od tego gdzie dana podpróbka znajduje się na obrazie bazowym. Funkcje *hann\_up\_down\_eft\_right*, *hann\_up\_l\_up\_r* oraz *hann\_down\_r\_down\_l* zawierają implementacje poszczególnych dwuwymiarowych okien Hanna.

### Kod źródłowy 61.

```

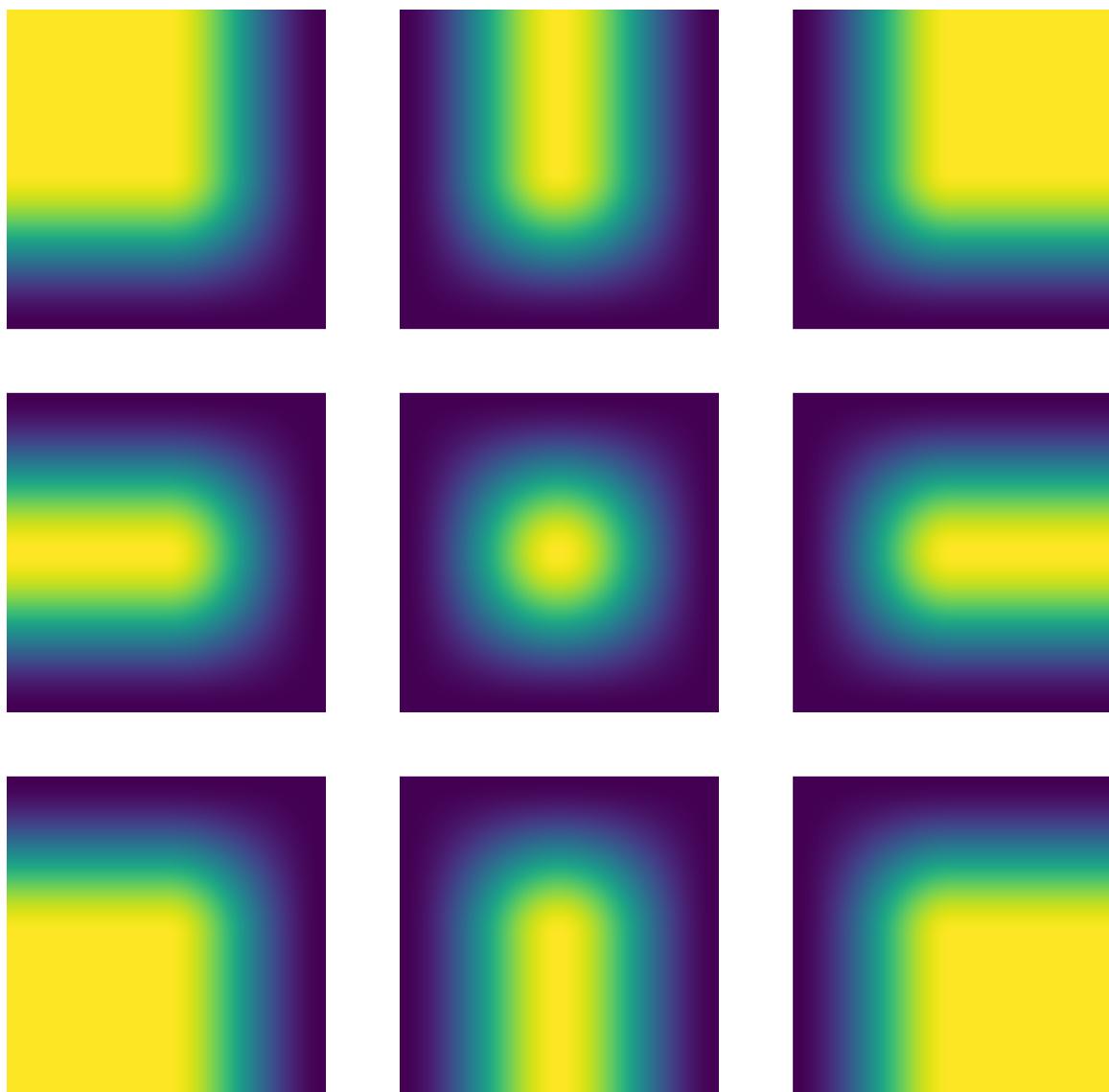
1 def hann_down_r_down_l(i, j, h_part, v_part):
2     if i >= sample_res / 2 and j >= sample_res / 2:
3         hann_wnd_down_r[j, i] = 1
4     elif i < sample_res / 2 and j > sample_res / 2:
5         hann_wnd_down_r[j, i] = 0.5 * h_part
6     elif i > sample_res / 2 and j < sample_res / 2:
7         hann_wnd_down_r[j, i] = 0.5 * v_part
8     else:
9         hann_wnd_down_r[j, i] = 0.25 * h_part * v_part
10    if i <= sample_res / 2 and j >= sample_res / 2:
11        hann_wnd_down_l[j, i] = 1
12    elif i > sample_res / 2 and j > sample_res / 2:
13        hann_wnd_down_l[j, i] = 0.5 * h_part
14    elif i < sample_res / 2 and j < sample_res / 2:
15        hann_wnd_down_l[j, i] = 0.5 * v_part
16    else:
17        hann_wnd_down_l[j, i] = 0.25 * h_part * v_part

```

Rysunek 5.2 prezentuje wizualizację wszystkich dziewięciu okien Hanna, które w dalszej części niniejszego podrozdziału posłużą do połączenia predykcji uzyskanych z podprobek w jedną łączną maskę predykcji.

### Kod źródłowy 62.

```
1 for i in range(sample_res):
2     for j in range(sample_res):
3         h_part = (1 - np.cos((2 * np.pi * i) / (sample_res - 1)))
4         v_part = (1 - np.cos((2 * np.pi * j) / (sample_res - 1)))
5         hann_wnd_b[j, i] = 0.25 * h_part * v_part
6         hann_up_down_left_right(i, j, h_part, v_part)
7         hann_up_l_up_r(i, j, h_part, v_part)
8         hann_down_r_down_l(i, j, h_part, v_part)
```



Rysunek 5.2. Wizualizacja dziewięciu okien Hanna 2D służących do właściwego ważenia predykcji poszczególnych fragmentów zdjęcia bazowego

### Kod źródłowy 63.

```

1 f2, axarr2 = plt.subplots(nrows=3, ncols=3, figsize=(25, 25))
2 axarr2[0, 0].imshow(hann_wnd_up_l, cmap='viridis')
3 axarr2[1, 0].imshow(hann_wnd_left, cmap='viridis')
4 axarr2[2, 0].imshow(hann_wnd_down_l, cmap='viridis')
5 axarr2[0, 1].imshow(hann_wnd_up, cmap='viridis')
6 axarr2[1, 1].imshow(hann_wnd_b, cmap='viridis')
7 axarr2[2, 1].imshow(hann_wnd_down, cmap='viridis')
8 axarr2[0, 2].imshow(hann_wnd_up_r, cmap='viridis')
9 axarr2[1, 2].imshow(hann_wnd_right, cmap='viridis')
10 axarr2[2, 2].imshow(hann_wnd_down_r, cmap='viridis')
11 axarr2[0, 0].axis('off')
12 axarr2[1, 0].axis('off')
13 axarr2[2, 0].axis('off')
14 axarr2[0, 1].axis('off')
15 axarr2[1, 1].axis('off')
16 axarr2[2, 1].axis('off')
17 axarr2[0, 2].axis('off')
18 axarr2[1, 2].axis('off')
19 axarr2[2, 2].axis('off')
20 plt.savefig('Okna Hanna.png', dpi=300, bbox_inches='tight')
```

Funkcja *choose\_proper\_hann* służy do wyboru odpowiedniego dwuwymiarowego okna Hanna, w zależności od tego, z której części bazowego zdjęcia walidacyjnego dana próbka została pobrana.

### Kod źródłowy 64.

```

1 def choose_proper_hann(i, j, max_shp):
2     if j == 0 and i == 0:
3         curr_hann = hann_wnd_up_l
4     elif i == 0 and j != 0 and j != max_shp - 1:
5         curr_hann = hann_wnd_up
6     elif i == 0 and j == max_shp - 1:
7         curr_hann = hann_wnd_up_r
8     elif j == 0 and i != 0 and i != max_shp - 1:
9         curr_hann = hann_wnd_left
10    elif j == 0 and i == max_shp - 1:
11        curr_hann = hann_wnd_down_l
12    elif j == max_shp - 1 and i != 0 and i != max_shp - 1:
13        curr_hann = hann_wnd_right
14    elif j == max_shp - 1 and i == max_shp - 1:
```

## 5. Analiza wyników

```
15 curr_hann = hann_wnd_down_r
16 elif j != 0 and j != max_shp - 1 and i == max_shp - 1:
17     curr_hann = hann_wnd_down
18 else:
19     curr_hann = hann_wnd_b
20 return curr_hann
```

Przy pomocy funkcji *get\_fin\_mask* realizowane jest generowanie finalnej maski dla poszczególnych zdjęć ze zbioru walidacyjnego. Funkcja ta przechodzi przez predykcje wygenerowane dla wszystkich podpróbek, przemnaża je przez odpowiednie okna Hanna i nanosi we właściwym miejscu finalnej maski *5000x5000*.

### Kod źródłowy 65.

```
1 def get_fin_mask(fin_preds, Gy, Gx, wnd_h, wnd_w, img_h, img_w, Sy, Sx):
2     rshp_fin_preds = fin_preds.reshape((Gy, Gx, wnd_h, wnd_w))
3     fin_image_mask = np.zeros((img_h, img_w))
4     end_x = sample_res
5     strt_x = 0
6     for i in range(Gy):
7         end_y = sample_res
8         strt_y = 0
9         for j in range(Gx):
10            curr_p = rshp_fin_preds[i, j]
11            curr_h = choose_proper_hann(i, j, Gy)
12            fin_image_mask[strt_x:end_x, strt_y:end_y] += np.multiply(curr_p, curr_h)
13            strt_y += Sy
14            end_y += Sy
15            strt_x += Sx
16            end_x += Sx
17     return fin_image_mask
```

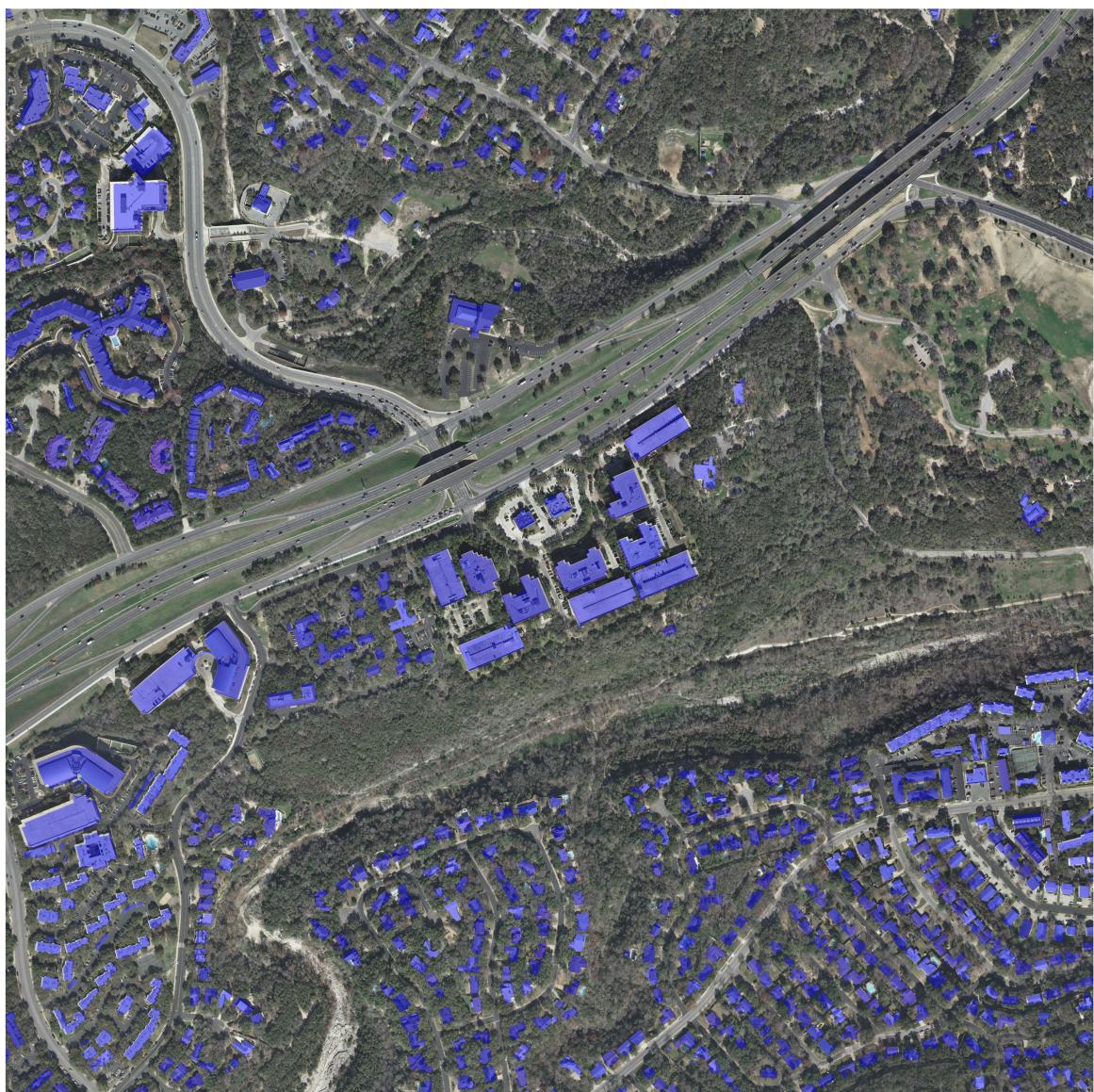
## 5.2. Pomiar efektywności sieci *GML-Net*

Poniższe obrazy wizualizują skuteczność sieci *GML-Net* w detekcji budynków na zdjęciach lotniczych na przykładzie losowo wybranego zdjęcia ze zbioru walidacyjnego.

### Kod źródłowy 66.

```
1 imgs_arr = pd.read_csv("m_inf_v.csv", header=None).values
2 img_n = 1
```

```
3 final_f1s = []
4 final_iou = []
5 final_acc = []
6 final_ssim = []
7 pred_time = []
8 img_time = []
9 img, msk, wnd_h, wnd_w, nm_cls, img_h, img_w, Sy, Sx = g_prms(imgs_arr, img_n)
10 wnd_img, Gx, Gy = get_wnd(img, msk, wnd_h, wnd_w, nm_cls, img_h, img_w, Sy, Sx)
11 fin_preds = get_all_preds(wnd_img)
12 f_img_msk = get_fin_mask(fin_preds, Gy, Gx, wnd_h, wnd_w, img_h, img_w, Sy, Sx)
```



Rysunek 5.3. Oryginalna maska (*ground truth*)

## 5. Analiza wyników

---

Porównanie wizualne rysunku 5.3 z rysunkiem 5.4 wskazuje, iż różnice w predykcji maski przez sieć *GML-Net* a faktyczną maską *ground truth* nie są istotne.



**Rysunek 5.4.** Maska wygenerowana przez sieć *GML-Net*

### Kod źródłowy 67.

```
1 img_rgba = np.ones((img.shape[0], img.shape[1], 4), dtype=np.uint8) * 255
2 img_rgba[:, :, :3] = img
3 mask_rgba = np.zeros((img.shape[0], img.shape[1], 4), dtype=np.uint8)
4 mask_rgba[msk == 255, :] = [0, 0, 255, 180]
5 pred_mask_rgba = np.zeros((img.shape[0], img.shape[1], 4), dtype=np.uint8)
6 pred_mask_rgba[f_img_msk > loss_thres, :] = [128, 0, 128, 180]
7
8 f2, axarr2 = plt.subplots(nrows=1, ncols=2, figsize=(25, 25))
9 axarr2[0].set_title("Maska ground truth\n", color='black', fontweight="bold",
```

```

10         fontsize=25)
11 axarr2[0].imshow(img_rgba)
12 axarr2[0].imshow(mask_rgba, alpha=0.6)
13 axarr2[0].axis('off')
14 axarr2[1].imshow(img_rgba)
15 axarr2[1].imshow(pred_mask_rgba, alpha=0.6)
16 axarr2[1].axis('off')
17 axarr2[1].set_title("Maska uzyskana z sieci GML-Net\n", color='black',
18                     fontweight="bold", fontsize=25)
19
20 plt.savefig('Predykcja vs. maska.png', dpi=300, bbox_inches='tight')

```

### Kod źródłowy 68.

```

1 def calc_metrics(f_img_msk, msk):
2     prd2 = f_img_msk > loss_thres
3     trgt2 = msk == 255
4     intrs = (prd2 * trgt2).sum()
5
6     f1s = torch.True_divide(2. * intrs, prd2.sum() + trgt2.sum())
7     f1s = f1s.cpu().numpy().item(0)
8     union = (prd2 + trgt2).sum()
9
10    iou = torch.True_divide(intrs, union).cpu().numpy().item(0)
11    acc = torch.True_divide(intrs + (~prd2 * ~trgt2).sum(), prd2.size)
12    acc = acc.cpu().numpy().item(0)
13
14    prd3 = torch.tensor(np.expand_dims(prd2, axis=0)).cuda()
15    trgt3 = torch.tensor(np.expand_dims(trgt2, axis=0)).cuda()
16    ssim = ssim_index(prd3, trgt3).cpu().numpy().item(0)
17
18    return f1s, iou, acc, ssim
19
20 f1s, iou, acc, ssim = calc_metrics(f_img_msk, msk)

```

Wartości poszczególnych metryk wyliczone dla powyższego losowego zdjęcia pochodzącego ze zbioru walidacyjnego potwierdzają obserwacje wizualne wskazujące na dobrą jakość predykcji. Wartości tych metryk są następujące:  $OA = 97,66\%$ ,  $IoU = 80,10\%$ ,  $F1S = 88,95\%$  oraz  $SSIM = 96,26\%$ .

**Kod źródłowy 69.**

```

1 for i in range(len(imgs_arr)):
2     since0 = time.time()
3     img, msk, wnd_h, wnd_w, nm_cls, img_h, img_w, Sy, Sx = g_prms(imgs_arr, i)
4     wnd_i, Gx, Gy = get_wnds(img, msk, wnd_h, wnd_w, nm_cls, img_h, img_w,
5                               Sy, Sx)
6     fin_p = get_all_preds(wnd_i)
7     since1 = time.time()
8     f_img_msk = get_fin_mask(fin_p, Gy, Gx, wnd_h, wnd_w, img_h, img_w,
9                               Sy, Sx)
10    pred_time += [time.time() - since1]
11    img_time += [time.time() - since0]
12    f1s, iou, acc, ssim = calc_metrics(f_img_msk, msk)
13    final_f1s += [f1s]
14    final_iou += [iou]
15    final_acc += [acc]
16    final_ssimm += [ssim]
17
18 joined_time = sum(img_time)
19 av_img_time = sum(img_time) / len(imgs_arr)
20 avg_pred_time = sum(pred_time) / (len(imgs_arr) * Gx * Gy)

```

Sieć *GML-Net* osiągnęła zadowalającą finalną skuteczność na zbiorze walidacyjnym przy predykcji masek o rozmiarach  $5000 \times 5000$  - uzyskano następujące finalne wartości metryk:  $OA = 96,44\%$ ,  $IoU = 75,97\%$ ,  $F1S = 86,07\%$  oraz  $SSIM = 94,55\%$ . Wyniki te zostały uzyskane przy średnim czasie generowania predykcji jednej maski  $256 \times 256$  na poziomie 0,0003 sekundy oraz średnim czasie generowania predykcji maski łącznego obrazka  $5000 \times 5000$  na poziomie 15 sekund.

Rodzaj metryki	Wyniki dla masek $256 \times 256$	Wyniki dla masek $5000 \times 5000$
<b>OA</b>	97,08%	96,44%
<b>IoU</b>	82,42%	75,97%
<b>F1S</b>	88,22%	86,07%
<b>SSIM</b>	95,46%	94,55%

**Tabela 5.1.** Podsumowanie różnic pomiędzy metrykami jakości predykcji sieci *GML-Net* dla masek  $256 \times 256$  a tymi samymi metrykami dla masek  $5000 \times 5000$ .

Jak można się było spodziewać metryki jakości predykcji modelu wyliczane na poziomie pełnych zdjęć o wysokiej rozdzielczości są gorsze od metryk wyliczanych na poziomie podpróbek o rozdzielczości  $256 \times 256$ . Ogólna dokładność pogarsza się o 0,64 punktu

procentowego, współczynnik podobieństwa Jaccarda spada aż o 6,45 p.p., wynik  $F1$  jest niższy o 2,35 p.p. a indeks podobieństwa strukturalnego jest gorszy o 0,91 p.p.

### 5.3. Uzyskane wyniki na tle literatury badawczej

Tabela 5.2 przywołuje ponownie wyniki uzyskane na zbiorze *Inria Aerial Image Labeling Dataset* przez autorów, których prace zostały omówione w przeglądzie literatury. Uzyskane przez nich wyniki dotyczą zbioru testowego *IAILD*, natomiast opisywane, jak dotąd, wartości metryk jakości predykcji sieci *GML-Net* dotyczyły wyłącznie zbioru walidacyjnego. Aby więc móc porównywać skuteczność modelu opisywanego w bieżącej pracy na tle skuteczności innych modeli, należałoby policzyć odpowiednie metryki jakości predykcji sieci *GML-Net* na zbiorze testowym.

Tytuł artykułu	Uzyskane wyniki
<i>Multi-Task Learning for Segmentation of Building Footprints with Deep Neural Networks</i> [1]	OA: 95.17% IoU: 70.14%
<i>Semantic Segmentation from Remote Sensor Data and the Exploitation of Latent Learning for Classification of Auxiliary Tasks</i> [2]	OA: 97.14% IoU: 80.32%
<i>Building Footprint Generation by Integrating Convolution Neural Network with Feature Pairwise Conditional Random Field (FPCRNF)</i> [37]	OA: 95.81% F1S: 87.65% IoU: 74.79%
<i>Polygonal Building Segmentation by Frame Field Learning</i> [7]	IoU: 78.00%

**Tabela 5.2.** Podsumowanie wyników uzyskanych na zbiorze *Inria Aerial Image Labeling Dataset* przez autorów innych badań zajmujących się problematyką detekcji budynków na zdjęciach lotniczych.

Niestety nie można zrobić tego samodzielnie, gdyż autorzy zbioru *Inria Aerial Image Labeling Dataset* nie udostępnili masek *ground truth* dla zbioru testowego. Przygotowali oni jednak formularz na stronie internetowej *IAILD* (<https://www.lri.fr/~gcharpia/aeria>

## 5. Analiza wyników

---

I\_benchmark), w ramach którego możliwe jest przesłanie, do autorów wykorzystywanego w bieżącej pracy zbioru danych, predykci 180 masek dla zbioru testowego. Po wysłaniu takich danych, po pewnym czasie, autorzy *Inria Aerial Image Labeling Dataset* odsyłają, na podany w formularzu adres *e-mail*, wyliczone przez nich wartości metryk *IoU* oraz *OA*, w podziale na wyniki dla poszczególnych miast ze zbioru testowego.

Miasto	OA	IoU
Bellingham	97,11%	71,39%
Bloomington	97,34%	71,83%
Innsbruck	96,99%	74,89%
San Francisco	91,72%	74,98%
Tyrol Wschodni	98,01%	77,85%
<b>Łącznie:</b>	<b>96,23%</b>	<b>74,42%</b>

**Tabela 5.3.** Podsumowanie wyników sieci *GML-Net* dla zbioru testowego *IAILD*

Tabela 5.3 przedstawia wskaźniki jakości predykci sieci *GML-Net* dla zbioru testowego *IA-ILD*, wyliczone przez autorów tego zbioru po przesłaniu im 180 predykci masek. Jak widać omawiana w bieżącym rozdziale sieć osiągnęła łączną skuteczność mierzoną przy pomocy metryki *OA* na poziomie 96,23% a mierzoną przy pomocy metryki *IoU* na poziomie 74,42%. Najlepsze wyniki sieć *GML-Net* uzyskała dla miasta Tyrol Wschodni, a najgorsze dla miast San Francisco (pod kątem *OA*) oraz Bellingham (pod kątem *IoU*).

Porównując uzyskane wyniki do wyników zaprezentowanych w przeglądzie literatury można śmiało stwierdzić, iż sieć *GML-Net* jest w stanie generować wyniki o zbliżonej jakości do modeli przedstawionych w literaturze, odstając o zaledwie niecały punkt procentowy od najlepszego wyniku pod kątem metryki *OA* i o niecałe sześć punktów procentowych pod kątem metryki *IoU*. Wyniki uzyskane przez sieć *GML-Net* można również porównać do wyników prezentowanych przez autorów zbioru *Inria Aerial Image Labeling Dataset* na ich stronie internetowej (<https://project.inria.fr/aerialimagelabeling/leaderboard>). Znajduje się tam tabela przedstawiająca wartości uzyskiwanych metryk *OA* i *IoU* dla 110 modeli, których autorzy zgodzili się na publikację wskaźników jakości predykci ich sieci. Średnia wartość ogólnej dokładności dla tych 110 modeli wynosi 95,46%, a średnia wartość indeksu Jaccarda wynosi 70,02%, co oznacza, iż sieć *GML-Net* uzyskuje wyniki istotnie wyższe od średnich dla tego zbioru - mówiąc dokładniej, zajmuje 29. miejsce pod kątem *OA* oraz 30 pod kątem *IoU*.

## 6. Zakończenie

Niniejsza praca podejmowała problematykę detekcji budynków na zdjęciach lotniczych przy użyciu głębokich sieci neuronowych. Praca ta w pierwszych rozdziałach skupiała się na zdefiniowaniu najważniejszych pojęć związanych z głębokim uczeniem, przedstawieniem jego podstaw teoretycznych, historii oraz najistotniejszych architektur z nim związanych. Kolejne rozdziały zostały poświęcone przeglądowi literatury badawczej z zakresu badanej problematyki oraz definicji własnej głębokiej sieci neuronowej o nazwie *GML-Net*. Całość została zwieńczona rozdziałem opisującym sposób generacji finalnych predykcji oraz porównującym uzyskane wyniki do wyników przedstawionych literaturze badawczej.

Zaprezentowana w bieżącej pracy sieć *GML-Net* pozwoliła na uzyskanie zadowalającej skuteczności w detekcji budynków na zdjęciach lotniczych pochodzących ze zbioru *Inria Aerial Image Labeling Dataset*. Nie udało się w prawdzie osiągnąć lepszych rezultatów niż aktualne wyniki *state of art* zaprezentowane w pracy *Semantic Segmentation from Remote Sensor Data and the Exploitation of Latent Learning for Classification of Auxiliary Tasks* [2], ale mimo to wyniki uzyskane przy pomocy sieci *GML-Net* można uznać za satysfakcyjujące. Za duże pole do dalszego rozwoju uzyskanej sieci można uznać sposób łączenia predykcji mask o rozdzielczości  $256 \times 256$  w jedną łączną maskę o rozdzielczości  $5000 \times 5000$ , gdyż w tym procesie skuteczność predykcji istotnie się pogarszała, szczególnie patrząc przez pryzmat metryki *Intersection over Union*.

Za duże zalety sieci *GML-Net* można natomiast uznać jej ciekawą architekturę wykorzystującą elementy sieci *ResNet*, *U-Net* oraz *ICT-Net*, a także nowatorską funkcję straty stanowiąca ważoną sumę *Binary Cross-Entropy Loss*, *Dice Loss* oraz *Lovász hinge loss*. Stąd też, za uprawnione wydaje się stwierdzenie, iż zaprezentowana w niniejszej pracy głęboka sieć neuronowa *GML-Net* wnosi nowe, ciekawe spojrzenie do literatury podejmującej problematykę detekcji budynków na zdjęciach lotniczych.

## Bibliografia

- [1] Bischke B., Helber P., Folz J., Borth D. i Dengel A., "Multi-Task Learning for Segmentation of Building Footprints with Deep Neural Networks", *2019 IEEE International Conference on Image Processing (ICIP)*, s. 1480–1484, 2017.
- [2] Chatterjee B. i Poullis Ch., "Semantic Segmentation from Remote Sensor Data and the Exploitation of Latent Learning for Classification of Auxiliary Tasks", Dostęp zdalny (30.09.2020): <https://arxiv.org/pdf/1912.09216.pdf>, 2019.
- [3] Chen L., Papandreou G., Kokkinos I., Murphy K. i Yuille A. L., "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, t. 40, nr. 4, s. 834–848, 2017.
- [4] Chen L., Papandreou G., Kokkinos I., Murphy K. i Yuille A. L., "Semantic image segmentation with deep convolutional nets and fully connected CRFs", Dostęp zdalny (30.09.2020): <https://arxiv.org/pdf/1412.7062.pdf>, 2015.
- [5] Fukushima K., "Cognitron: A Self-Organizing Multilayered Neural Network", *Biological cybernetics*, t. 20, nr. 4, s. 121–136, 1975.
- [6] Fukushima K., "Neocognitron: A Self Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", *Biological cybernetics*, t. 36, nr. 4, s. 193–202, 1980.
- [7] Girard N., Smirnov D., Solomon J. i Tarabalka Y., "Polygonal Building Segmentation by Frame Field Learning", Dostęp zdalny (30.09.2020): <https://arxiv.org/pdf/2004.14875.pdf>, 2020.
- [8] Girshick R., Donahue J., Darrell T. i Malik J., "Rich feature hierarchies for accurate object detection and semantic segmentation", *2014 IEEE Conference on Computer Vision and Pattern Recognition*, s. 580–587, 2014.
- [9] Golovanov S., Kurbanov R., Artamonov A., Davydow A. i Nikolenko S., "Building Detection from Satellite Imagery Using a Composite Loss Function", *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2018.
- [10] Goodfellow I., Bengio Y. i Courville A., *Deep Learning*. Cambridge, Massachusetts: The MIT Press, 2016.
- [11] He K., Zhang X., Ren S. i Sun J., "Deep Residual Learning for Image Recognition", *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*, s. 770–778, 2016.
- [12] Hinton G., Osindero S. i Teh Y. W., "A Fast Learning Algorithm for Deep Belief Nets", *Neural Computation*, t. 18, nr. 7, s. 1527–1554, 2006.
- [13] Hinton G., Sabour S. i Frosst N., "Dynamic Routing Between Capsules", *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems*, s. 3859–3869, 2017.

- [14] Hochreiter S. i Schmidhuber J., "Long Short-Term Memory", *Neural Computation*, t. 9, nr. 8, s. 1735–1780, 1997.
- [15] Hu J., Shen L. i Sun G., "Squeeze-and-Excitation Networks", *IIEEE/CVF Conference on Computer Vision and Pattern Recognition*, s. 7132–7141, 2017.
- [16] Iglovikov V., Mushinskiy S. i Osin V., "Satellite Imagery Feature Detection using Deep Convolutional Neural Network: A Kaggle Competition", Dostęp zdalny (30.09.2020): <https://arxiv.org/pdf/1706.06169.pdf>, 2017.
- [17] Jordan M. I., "Serial order: A parallel distributed processing approach", Dostęp zdalny (30.09.2020): <http://cseweb.ucsd.edu/~gary/258/jordan-tr.pdf>, 1986.
- [18] Krizhevsky A., Sutskever I. i Hinton G., "ImageNet classification with deep convolutional neural networks", Dostęp zdalny (30.09.2020): <http://www.cs.toronto.edu/~hinton/absps/imagenet.pdf>, 2012.
- [19] LeCun Y., Boser B., Denker J. S., Henderson D., Howard R. E., Hubbard W. i Jackel L. D., "Backpropagation Applied to Handwritten Zip Code Recognition", *Neural computation*, t. 1, nr. 4, s. 541–551, 1989.
- [20] LeCun Y., Bottou L., Bengio Y. i Haffner P., "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, t. 86, nr. 11, s. 2278–2324, 1998.
- [21] McCulloch W. i Pitts W., "A logical calculus of the ideas immanent in nervous activity", *Bulletin of Mathematical Biophysics*, t. 5, nr. 4, s. 115–113, 1943.
- [22] Minsky M. i Seymour P., *Perceptrons: an introduction to computational geometry*. Cambridge, Massachusetts: The MIT Press, 1969.
- [23] Pielawski N. i Wählby C., "Introducing Hann windows for reducing edge-effects in patch-based image segmentation", Dostęp zdalny (30.09.2020): <https://arxiv.org/pdf/1910.07831.pdf>, 2019.
- [24] Redmon J., Divvala S., Girshick R. i Farhadi A., "You Only Look Once: Unified, Real-Time Object Detection", *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, s. 779–788, 2016.
- [25] Ronneberger O., Fischer P. i Brox T., "U-Net: Convolutional Networks for Biomedical Image Segmentation", *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, t. 9351, s. 234–241, 2015.
- [26] Rosenblatt F., "The perceptron: a probabilistic model for information storage and organization in the brain", *Psychological Review*, t. 65, nr. 6, s. 386–408, 1958.
- [27] Rumelhart D., Hinton G. i Williams R., "Learning representations by back propagating errors", *Nature*, t. 323, nr. 9, s. 533–536, 1986.
- [28] Simonyan K. i Zisserman A., "Very deep convolutional networks for large-scale image recognition", Dostęp zdalny (30.09.2020): <https://arxiv.org/pdf/1409.1556.pdf>, 2014.

## 6. Bibliografia

---

- [29] Skarbek W., "Symbolic Tensor Neural Networks for Digital Media – from Tensor Processing via BNF Graph Rules to CREAMS Applications", *Fundamenta Informaticae*, t. 168, nr. 2–4, s. 89–184, 2019.
- [30] Szegedy Ch., Liu W., Jia Y., Sermanet P., Reed S., Anguelov D., Erhan D., Vanhoucke V. i Rabinovich A., "Going deeper with convolutions", *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*, s. 1–9, 2015.
- [31] Widrow B. i Hoff T., "Adaptive switching circuits", *1960 IRE WESCON Convention Record*, s. 96–104, 1960.
- [32] Wiener N., *Cybernetics: Or Control and Communication in the Animal and the Machine*. Cambridge, Massachusetts: The MIT Press, 1948.
- [33] Wu G., Shao X., Guo Z., Chen Q., Yuan W., Shi X., Xu Y. i Shibasaki R., "Automatic Building Segmentation of Aerial Imagery Using Multi-Constraint Fully Convolutional Networks", *Remote Sensing*, t. 10, nr. 3, s. 407–425, 2018.
- [34] Yu J. i Blaschko M., "The Lovász Hinge: A Novel Convex Surrogate for Submodular Losses", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, t. 42, nr. 3, s. 735–748, 2020.
- [35] K. Zagoruyko S., "Wide Residual Networks", Dostęp zdalny (30.09.2020): <https://arxiv.org/pdf/1605.07146.pdf>, 2017.
- [36] Zhou K., Yang Y. i C. andXiang T., "Omni-Scale Feature Learning for Person Reidentification", *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, s. 3701–3711, 2019.
- [37] Zhu X., Li X. Q., Shi Y. i Huang X., "Building Footprint Generation by Integrating Convolution Neural Network with Feature Pairwise Conditional Random Field (FPCRF)", *IEEE Transactions on Geoscience and Remote Sensing*, s. 1–18, 2020.

## Wykaz symboli i skrótów

**ADALINE** – ang. *Adaptive Linear Neuron*

**AI** – ang. *Artificial Intelligence*

**BCE** – ang. *Binary Cross-Entropy*

**BGD** – ang. *Batch Gradient Descent*

**BSGD** – ang. *Batch Stochastic Gradient Descent*

**BS** – ang. *Batch Size*

**CNN** – ang. *Convolutional Neural Network* / ang. *Capsule Neural Network*

**CPU** – ang. *Central Processing Unit*

**CE** – ang. *Cross-Entropy*

**DL** – ang. *Deep Learning* / ang. *Dice Loss*

**DNN** – ang. *Deep Neural Network*

**ELU** – ang. *Exponential Linear Unit*

**F1S** – ang. *F1 Score*

**FNN** – ang. *Feedforward Neural Network*

**FCN** – ang. *Fully Convolutional Network*

**GAN** – ang. *Generative Adversarial Networks*

**GPU** – ang. *Graphics Processing Unit*

**GSN** – Głębokie Sieci Neuronowe

**GT** – ang. *Ground Truth*

**ILSVRC** – ang. *ImageNet Large Scale Visual Recognition Challenge*

**IAILD** – ang. *Inria Aerial Image Labeling Dataset*

**IoU** – ang. *Intersection over Union*

**LF** – ang. *Loss Function*

**LHL** – ang. *Lovasz Hinge Loss*

**LR** – ang. *Learning Rate*

**LSTM** – ang. *Long Short-Term Memory*

**MAE** – ang. *Mean Absolute Error*

**MAP** – ang. *Mean Average Precision*

**MBGD** – ang. *Mini-Batch Gradient Descent*

**ML** – ang. *Machine Learning*

**MLP** – ang. *Multilayer Perceptron*

**MSE** – ang. *Mean Square Error*

**OA** – ang. *Overall Accuracy*

**RL** – ang. *Representation Learning* / ang. *Reinforcement Learning*

**R-CNN** – ang. *Region Based Convolutional Neural Networks*

**RNN** – ang. *Recurrent Neural Network*

**ROI** – ang. *Regions of Interest*

**SGD** – ang. *Stochastic Gradient Descent*

**SL** – ang. *Supervised Learning*

**SoA** – ang. *State of Art*

**SSIM** – ang. *Structural Similarity Index Measure*

**SSL** – ang. *Semi-Supervised Learning*

**TLU** – ang. *Threshold Logic Unit*

**UL** – ang. *Unsupervised Learning*

# Spis rysunków

2.1	Diagram Venna ukazujący zależności pomiędzy głębokim uczeniem, uczeniem reprezentacji, uczeniem maszynowym a sztuczną inteligencją [10] . . . . .	8
2.2	Przykładowy schemat neuronu zaprezentowany w 1943 roku przez Warrena McCullocha i Waltera Pittsa [21] . . . . .	14
2.3	Schematyczna reprezentacja połączeń w prostym perceptronie zaprezentowana w 1958 roku przez Franka Rosenblatta [26] . . . . .	14
2.4	Schemat ADALINE zaprezentowany w 1960 roku przez Bernarda Widrowa i Teda Hoffa [31] . . . . .	15
2.5	Diagram pokazujący połączenia pomiędzy warstwami w neocognitronie zaprezentowanym w 1980 roku przez Kunihiko Fukushima [6] . . . . .	16
2.6	Diagram pokazujący połączenia w sieci rekurencyjnej zaprezentowanej w 1986 roku przez Michaela Irwina Jordana [17] . . . . .	17
2.7	Schemat wielowarstwowego perceptronu zaprezentowany w 1986 roku przez Davida Rumelharta, Geoffreya Hintona i Ronaldsa Williamsa [27] . . . . .	17
2.8	Architektura sieci <i>LeNet-5</i> stworzona w 1998 roku przez Yanna LeCuna [20] . .	18
2.9	Architektura sieci <i>LSTM</i> zaprezentowanej w 1997 roku przez Seppa Hochreitera i Jürgena Schmidhubera [14] . . . . .	19
2.10	Architektura sieci <i>AlexNet</i> zaprezentowana w 2012 roku przez Alexa Krizhevsky'ego, Ilya Sutskevera i Geoffrey'a Hintona [18] . . . . .	21
2.11	Architektura sieci <i>VGGNet</i> zaprezentowana w 2014 roku przez Karen Simonyan i Andrew Zissermana [28] . . . . .	22
2.12	Architektura bloku <i>Inception</i> zaprezentowana w 2014 roku w artykule <i>Going deeper with convolutions</i> [30] . . . . .	23
2.13	Architektura sieci <i>GoogleNet</i> zaprezentowana w 2014 roku w artykule <i>Going deeper with convolutions</i> [30] . . . . .	24
2.14	Wizualizacja architektury modelu GAN (Źródło: <a href="https://jrmerwin.github.io/deeplearning4j-docs/generative-adversarial-network.html">https://jrmerwin.github.io/deeplearning4j-docs/generative-adversarial-network.html</a> ) . . . . .	25
2.15	Schemat działania modelu <i>R-CNN</i> zaprezentowany w 2014 roku przez Rossa Girshicka, Jeffa Donahue'a, Trevora Darrella i Jitendra Malika [8] . . . . .	26
2.16	Blok rezydualny zaprezentowany w 2015 roku przez He K., Zhang X., Ren S. i Sun J. [11] . . . . .	27
2.17	Przykładowa architektura sieci <i>ResNet-34</i> zaprezentowana w 2015 roku przez He K., Zhang X., Ren S. i Sun J. [11] . . . . .	28
2.18	Architektura sieci <i>U-Net</i> zaprezentowana w 2015 roku przez Olafa Ronnebergera, Philippa Fischera i Thomasa Broxa [25] . . . . .	29
2.19	Architektura bloku <i>ASSP</i> zaprezentowana w 2017 roku przez Chen L., Papandreou G., Kokkinos I., Murphy K. oraz Yuille A. L. [3] . . . . .	30

2.20 Architektura sieci <i>YOLO</i> zaprezentowana w 2016 roku przez Josepha Redmona, Santosha Divwalsa, Rossa Girshicka i Aliego Farhadiego [24] . . . . .	31
2.21 Architektura bloku <i>Squeeze-and-Excitation</i> zaprezentowana w 2017 roku przez Hu J., Shen L. i Sun G.[15] . . . . .	32
2.22 Architektura sieci <i>CapsNet</i> zaprezentowana w 2017 roku przez Geoffrey'a Hintona, Sarę Sabour i Nicholasa Frossa [13] . . . . .	33
2.23 Architektura bloku <i>Bottleneck</i> zaprezentowana w 2019 roku przez Zhou K., Yang Y., Cavallaro A. i Xiang T. [36] . . . . .	34
4.1 Porównanie oryginalnego oraz przetransformowanego zdjęcia . . . . .	59
4.2 Architektura sieci <i>GML-Net</i> . . . . .	61
4.3 Krzywe uczenia sieci <i>GML-Net</i> . . . . .	88
5.1 Wybrane zdjęcie pochodzące ze zbioru walidacyjnego rozbite na 1444 podpórki	92
5.2 Wizualizacja dziewięciu okien Hanna 2D służących do właściwego ważenia predykcji poszczególnych fragmentów zdjęcia bazowego . . . . .	96
5.3 Oryginalna maska ( <i>ground truth</i> ) . . . . .	99
5.4 Maska wygenerowana przez sieć <i>GML-Net</i> . . . . .	100

## Spis tabel

3.1 Podsumowanie najważniejszych informacji pochodzących z artykułów omówionych w przeglądzie literatury . . . . .	45
5.1 Podsumowanie różnic pomiędzy metrykami jakości predykcji sieci <i>GML-Net</i> 256x256 a tymi samymi metrykami dla masek $5000x5000$ . . . . .	102
5.2 Podsumowanie wyników uzyskanych na zbiorze <i>Inria Aerial Image Labeling Dataset</i> przez autorów innych badań zajmujących się problematyką detekcji budynków na zdjęciach lotniczych. . . . .	103
5.3 Podsumowanie wyników sieci <i>GML-Net</i> dla zbioru testowego <i>IAILD</i> . . . . .	104