



Imię i nazwisko studenta: Bartosz Bieliński  
Nr albumu: 165430  
Studia pierwszego stopnia  
Forma studiów: stacjonarne  
Kierunek studiów: Automatyka i Robotyka  
Profil: Systemy decyzyjne i robotyka

Imię i nazwisko studenta: Piotr Winkler  
Nr albumu: 165504  
Studia pierwszego stopnia  
Forma studiów: stacjonarne  
Kierunek studiów: Automatyka i Robotyka  
Profil: Systemy decyzyjne i robotyka

## **PRACA DYPLOMOWA INŻYNIERSKA**

Tytuł pracy w języku polskim: Zastosowanie sieci neuronowych do edycji obrazów

Tytuł pracy w języku angielskim: Application of neural networks for image editing

Potwierdzenie przyjęcia pracy	
Opiekun pracy	Kierownik Katedry/Zakładu (pozostawić właściwe)
<i>podpis</i>	<i>podpis</i>
dr inż. Mariusz Domżalski	

Data oddania pracy do dziekanatu:

**POLITECHNIKA GDAŃSKA**

Katedra Systemów Decyzyjnych i Robotyki

**PRACA INŻYNIERSKA**

**Zastosowanie sieci neuronowych do edycji  
obrazów**

**Autorzy**

Piotr Winkler

Bartosz Bieliński

Gdańsk 2019

## **STRESZCZENIE**

Tematem pracy jest zbadanie możliwości zastosowania sieci neuronowych do edycji obrazów. Głównym celem było stworzenie narzędzi opartych na wyuczonych sieciach neuronowych, służących do odpowiedniego przetwarzania i modyfikowania obrazu. <Zakres pracy >

<Zastosowane metody badań ><wyniki ><najważniejsze wnioski >

**Słowa kluczowe:** sieć neuronowa, przetwarzanie obrazu, konwolucyjna sieć neuronowa, splotowa sieć neuronowa, model generatywny, głęboka sieć neuronowa,

**Dziedzina nauki i techniki zgodna z OECD:** Nauki inżynierijne i techniczne, Elektrotechnika, elektronika, inżynieria informatyczna, Sprzęt komputerowy i architektura komputerów

## **ABSTRACT**

The subject of the work is research on the possibilities of applying neural networks for image editing. The main goal was to create tools based on trained neural networks used for appropriate processing and modifying of images. <Zakres pracy >

<Zastosowane metody badań ><wyniki ><najważniejsze wnioski >

**Keywords:** neural network, image processing, convolutional neural network, generative adversarial network, deep neural network

**Field of science and technology in accordance with the requirements of the OECD:** Engineering and technology, Electrical engineering, Electronic engineering, Information engineering, Computer hardware and architecture

## WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

NN (ang. Neural Network) - Sieć neuronowa

ANN (ang. Artificial Neural Network) - Sztuczna sieć neuronowa

DNN (ang. Deep Neural Network) - Głęboka sieć neuronowa

FCL (ang. Fully Connected Layer) - Warstwa gestwa

CNN (ang. Convolutional Neural Network) - Splotowa sieć neuronowa

FCN (ang. Fully Convolutional Network) - Sieć w pełni splotowa

GAN (ang. Generative Adversarial Network) - Sieci o modelu generatywnym

VAE (ang. Variational Autoencoder) - Autoenkodery wariacyjne

ReLU (ang. Rectified Linear Unit) - Jednostronnie obcięta funkcja liniowa

BatchNorm (ang. Batch Normalization) - Normalizacja zbioru danych pogrupowanych w pakiety

YUV - Model barw, w którym Y odpowiada za luminancję obrazu, a UV są to dwa kanały chrominacji i kodują barwę

IcGAN (ang. Invertible conditional Generative Adversarial Network) - Odwracalny warunkowy model generatywny

cGAN (ang. conditional Generative Adversarial Network) - warunkowy model generatywny

Dropout (!!! ang. spadkowicz - tłumaczył Piotr Winkler) - technika regularyzacji mająca na celu ograniczać przeuczanie się sieci neuronowych

PReLU (ang. Parametric Rectified Linear Unit) - Parametryczna jednostronnie obcięta funkcja liniowa

RReLU (ang. Randomized Leaky Rectified Linear Unit) - Losowo nieszczelna jednostronnie obcięta funkcja liniowa

hiperparametry - parametry warunkujące przebieg procesu uczenia sieci neuronowych, takie jak np. długość kroku treningowego, czy ilość epok treningowych

CPU (ang. Central Processing Unit) - Centralna Jednostka Obliczeniowa

GPU (ang. Graphics Processing Unit) - Graficzna Jednostka Obliczeniowa

JSON (ang. JavaScript Object Notation) - lekki, tekstowy format wymiany danych komputerowych

AI (ang. Artificial Intelligence) - sztuczna inteligencja

OpenCV (ang. Open Source Computer Vision Library] - otwartoźródłowa biblioteka wizji komputerowej

tensor - macierzowa struktura danych biblioteki PyTorch

SGD (ang. Stochastic Gradient Descent) - Stochastyczny Spadek Gradientu

AdaGrad (ang. Adaptive Gradient Algorithm) - Adaptacyjny Algorytm Gradientowy

Adam (ang. Adaptive Moment Estimation) - Estymacja Momentu Adaptacyjnego

SAIL (Stanford Artificial Intelligence Laboratory) - Laboratorium Sztucznej Inteligencji Stanforda

# Spis treści

<b>WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW</b>	<b>4</b>
<b>1 Wstęp i cel pracy</b>	<b>8</b>
1.1 Cel pracy . . . . .	8
1.2 Dotychczasowe dokonania . . . . .	8
1.3 Założenia projektowe . . . . .	9
1.4 Układ pracy . . . . .	9
<b>2 Podstawy teoretyczne</b>	<b>10</b>
2.1 Sieci splotowe . . . . .	10
2.2 FCN . . . . .	12
2.3 Modele generatywne . . . . .	12
2.4 Autoenkodery . . . . .	13
<b>3 Przegląd rozwiązań</b>	<b>14</b>
3.1 Colorful image colorization . . . . .	14
3.2 Image Style Transfer Using Convolutional Neural Networks . . . . .	15
3.3 Invertible Conditional GANs for image editing . . . . .	16
3.4 Neural photo editing . . . . .	17
<b>4 Zaimplementowane rozwiązania</b>	<b>19</b>
4.1 TorchFrame . . . . .	20
4.1.1 PyTorch . . . . .	20
4.1.2 Architektura TorchFrame . . . . .	21
4.1.3 Konfiguracja w TorchFrame . . . . .	24
4.1.4 Testowanie w TorchFrame . . . . .	33
4.2 Filtry AI . . . . .	35
4.2.1 Filtr Sobela . . . . .	35
4.2.2 Sepia . . . . .	39
4.2.3 Filtr górnoprzepustowy . . . . .	42
4.3 Automatyczne kolorowanie czarno-białych obrazów . . . . .	43
4.3.1 Podejście . . . . .	43
4.3.2 Model końcowy . . . . .	44
4.3.3 BatchNorm . . . . .	46
4.3.4 Dropout . . . . .	47
4.3.5 Modyfikacja rozdzielczości . . . . .	47
4.3.6 Wykorzystywany zbiór treningowy . . . . .	48
4.3.7 Przetwarzanie wstępne danych . . . . .	49
4.3.8 Przetwarzanie końcowe danych . . . . .	50
4.3.9 Augmentacja danych . . . . .	51
4.3.10 Funkcje kosztów . . . . .	52
4.3.11 Funkcje aktywacji . . . . .	53
4.3.12 Algorytmy optymalizacyjne . . . . .	54
4.3.13 Trening . . . . .	54

4.3.14 Rezultaty . . . . .	55
<b>5 Podsumowanie</b>	<b>59</b>
<b>Bibliografia</b>	<b>60</b>
<b>Spis rysunków</b>	<b>62</b>
<b>Spis tabel</b>	<b>63</b>

# 1 WSTĘP I CEL PRACY

Sztuczne sieci neuronowe sięgają swym początkiem lat 40. XX wieku. Historia ich rozwoju odnotowała trzy okresy, w których rozwiązania te odbijały się szerokim echem w środowisku naukowym.

Pierwszy model neuronu, a potem perceptron zapoczątkowały rozwój tej dziedziny nauki, jednak pierwsze sieci jednowarstwowe nie były w stanie rozwiązywać złożonych problemów. Przeszkodę nie do pokonania stanowiła dla nich nawet prosta funkcja logiczna XOR. Z tego powodu badania sieci neuronowych zostały na długi czas porzucone.

Pojawienie się algorytmu wstępnej propagacji błędów pozwalającego skutecznie uczyć wielowarstwowe sieci neuronowe ponownie wzmożło zainteresowanie tematem, jednak tym razem na drodze postępowi stanęły ograniczenia technologiczne ówczesnych czasów.

Wreszcie wraz z nadaniemiem XXI wieku postępujący rozwój komputerów oraz internetu umożliwił sztucznym sieciom neuronowym rozwinięcie skrzydeł. Wejście w erę "big data" otworzyło dostęp do olbrzymich zbiorów danych niezbędnych do treningu sieci, a pojawienie się wysokowydajnych jednostek obliczeniowych pozwoliło znacznie ten proces przyspieszyć.

Zapoczątkowany w ten sposób rozwój trwa do dnia dzisiejszego. Sztuczne sieci neuronowe odnajdują zastosowanie w wielu dziedzinach życia i nauki. Grają w gry, przeprowadzają symulacje, przewidują i prognozują zachowania rynku, czy pogody, analizują i przetwarzają obrazy cyfrowe.

Z punktu widzenia niniejszej pracy największe znaczenie ma oczywiście ostatni z wymienionych punktów. Zdefiniowanie sieci neuronowych, jako matematycznych modeli obliczeniowych ujawnia ich naturalne predyspozycje do pracy na obrazach cyfrowych. W praktyce stanowią one bowiem zbiór liczb, wartości poszczególnych pikseli, który sieć neuronowa jest w stanie analizować, przetwarzać i modyfikować.

## 1.1 Cel pracy

Celem pracy jest stworzenie szeregu narzędzi programistycznych oferujących szeroki wachlarz możliwości edytowania obrazu. Narzędzia te oparte mają być na technologii sieci neuronowych. W szczególności przetestowana będzie skuteczność rozwiązań dedykowanych do przetwarzania obrazów, takich jak sieci konwolucyjne albo modele generatywne.

Po opracowaniu tychże narzędzi, opisane zostaną efekty pracy oraz zbadana zostanie skuteczność sieci neuronowych jako rozwiązania nakreślonej problematyki. Omówione zostaną także wykorzystane architektury zaimplementowanych modeli, wykorzystane funkcje kosztu, metody aktualizowania wag sieci oraz przebiegi treningu modeli.

## 1.2 Dotychczasowe dokonania

<Syntetyczny opis dotychczasowych dokonań w danej tematyce? >

### ***1.3 Założenia projektowe***

Główym założeniem pracy było zaprojektowanie i zaimplementowanie narzędzi programistycznych służących do edycji obrazu. Narzędzia te muszą wykorzystywać do swoich celów nauuczone sieci neuronowe. Na podstawie jakości działania tychże narzędzi, oceniona zostanie ich rzetelność oraz skuteczność.

Następnie przeprowadzona zostanie analiza słuszności zastosowania sieci neuronowych jako rozwiązania przedstawionej problematyki. Uzyskane rozwiązanie zostanie także zestawione ze znanyymi algorytmami do edycji obrazu nie opierającymi się na technologii sieci neuronowych.

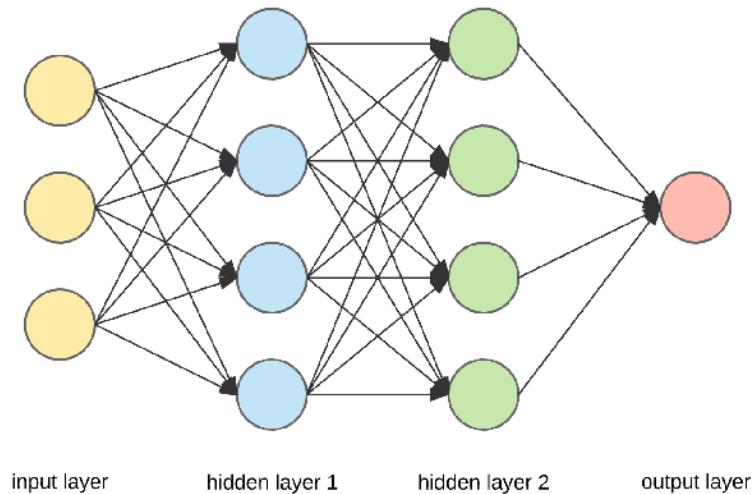
### ***1.4 Układ pracy***

W pierwszym rozdziale przedstawiono zarys rozwoju Sieci Neuronowych na przestrzeni lat oraz niezbędne podstawy teoretyczne związane z przedstawioną problematyką i wybranym dla niej rozwiązaniem.

W kolejnym rozdziale dokonano przeglądu już istniejących rozwiązań, opisano ich przeznaczenie, sposób działania oraz uzyskane rezultaty. Oceniono także wpływ danego rozwiązania na rozwój sieci neuronowych w dziedzinie przetwarzania i edytowania obrazów.

## 2 PODSTAWY TEORETYCZNE

W dzisiejszych czasach sieci neuronowe zajmują ważną pozycję na rynku narzędzi do edycji obrazu. Jest to głównie spowodowane ich umiejętnością do reprodukowania i modelowania niesłiniowych procesów, a także nowoczesnymi technikami przetwarzania plików graficznych. Jednak pierwsze architektury ANN (ang. artificial neural network) nie nadawały się do przetwarzania grafik. Było to częściowo spowodowane faktem, że obrazy, będące w rzeczywistości macierzami wartości pikseli, ciężko było skutecznie podać na wejście typowych architektur DNN (ang. deep neural network) zbudowanych pierwotnie z wielu warstw ukrytych, pomiędzy którymi połączenia są na zasadzie każdy z każdym oraz mają swoje wagi podlegające modyfikacji w trakcie procesu uczenia. Taka struktura pokazana została na Rysunku 2.1. Obrazy o niskiej rozdzielcości można było przekształcić w wektory wartości poszczególnych pikseli i w takiej postaci podawać na wejście sieci, jednak w przypadku obrazów o wyższej rozdzielcości to rozwiązanie, ze względu na znaczną długość powstających wektorów, nie oferowało dobrych rezultatów. Dopiero nowe architektury sieci spowodowały przełom w tej dziedzinie. Wprowadzenie do najistotniejszych i najciekawszych z nich zostanie przedstawione w poniższym rozdziale, a także rozwinięte w dalszej części tej pracy.



Rysunek 2.1: Struktura DNN

### 2.1 Sieci splotowe

Neuronowe sieci splotowe (CNN ang. convolutional neural network) stanowią podstawową strukturę w zakresie przetwarzania i analizowania obrazów cyfrowych. Są to sieci o hierarchicznej strukturze stanowiące podwaliny większości klasyfikatorów, detektorów, czy sieci segmentujących.

Autorzy jednego z artykułów traktujących o sieciach splotowych [3] opisują je następująco:

*'CNN to skuteczny algorytm poznawczy, stosowany powszechnie przy rozpoznawaniu wzorców i przetwarzaniu obrazów. Posiada wiele cech, takich jak prosta struktura, mniej parametrów treningowych, czy zdolność do adaptacji. CNN stały się gorącym tematem w zakresie analizy głosu i rozpoznawania obrazu. Ich struktura oparta na podziale wag czyni je bardziej podobnymi do biologicznych sieci neuronowych. Redukuje to złożoność modelu sieci oraz liczbę wag'.*

Na CNN składają się zazwyczaj trzy rodzaje warstw, z których każda posiada inne cechy.

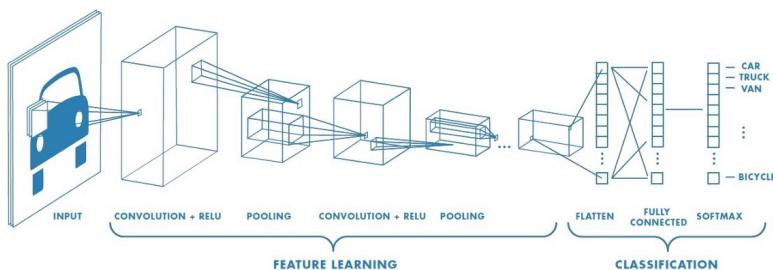
Podstawową warstwę stanowi warstwa splotowa. Składa się ona ze zbioru filtrów (neuronów) odpowiedzialnych za ekstrakcję cech z analizowanych obrazów poprzez dokonanie operacji konwolucji na obrazie poprzez przesuwanie zestawu filtrów wzdłuż niego. Na wyjściu filtrów otrzymuje się macierze o mniejszej rozdzielcości reprezentujące wyniki operacji konwolucji w danym punkcie. Każda kolejna warstwa splotowa wydobywa z obrazu cechy o wyższych poziomach abstrakcji bazując na wynikach obliczeń poprzednich warstw tego rodzaju. Dzięki temu procesowi kolejne warstwy filtrów uczą się rozpoznawać kluczowe cechy na obrazie, od drobnych elementów takich jak krawędzie albo kształty po bardziej złożone takie jak części ciała albo całe obiekty. Filtry te są zazwyczaj inicjowane losowymi wartościami i w miarę trenowania, dopasowują swoje parametry do wybranej problematyki.

Drugim istotnym elementem sieci splotowych jest warstwa poolingu. Może zostać opisana następująco [4]:

*'We wszystkich przypadkach pooling pomaga uczynić reprezentację w przybliżeniu niezmiennej w stosunku do małych tłumaczeń danych wejściowych. Niezmienność wobec tłumaczenia oznacza, że jeśli poddamy dane wejściowe nieznacznej translacji, to wartość większości wyników poddanych poolingu nie ulegnie zmianie'.*

Końcowy element CNN w większości przypadków stanowią warstwy gęste (FCL ang. Fully Connected Layer). Odpowiadają one za dokonanie odpowiedniej klasyfikacji obrazu na podstawie danych dostarczonych przez warstwy poprzedzające. Są przez to nieodzowne w przypadku zadań związanych z wszelkiego rodzaju klasyfikacją obrazów.

Wymienione tutaj elementy składowe sieci splotowych mogą przyjmować różne rozmiary i występować w różnych konfiguracjach, co przedstawiono na poniższym Rysunku 2.2.

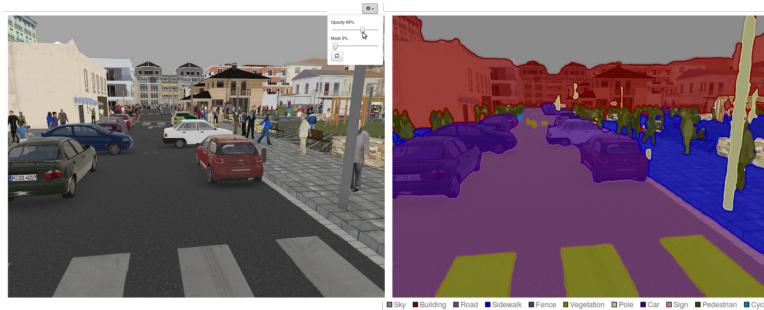


Rysunek 2.2: Przykładowa struktura CNN

Zapewnia to szerokie pole do eksperymentów i sprawia, że sieci te zdolne są rozwiązywać złożone, różnorodne problemy z wielu dziedzin codziennego życia.

## 2.2 FCN

Jednym z kluczowych problemów, jakie stawia przed badaczami edycja obrazów jest zagadnienie segmentacji semantycznej. Klasyczna klasyfikacja, polegająca na przypisywaniu obrazów do odpowiednich grup tematycznych, jest w tym przypadku sprowadzana do poziomu pojedynczych pikseli. Oznacza to, że sieci neuronowe przeznaczone do tego zadania są w stanie dokonać klasyfikacji dla każdego pojedynczego piksela analizowanego obrazu. Na tej podstawie uzyskiwany jest podział na segmenty, z których każdy reprezentuje inną klasę obiektów, jak na poniższym Rysunku 2.3.



Rysunek 2.3: Segmentacja semantyczna

W większości modele odpowiadające za przeprowadzanie segmentacji składają się z szeregowego połączenia enkodera oraz dekodera. Enkoder jest zazwyczaj pre-trenowaną siecią neuronową przeznaczoną do klasyfikowania obrazów. Dekoder odpowiada za semantyczne rzutowanie cech w niskiej rozdzielcości, wyuczonych przez enkoder, na wysoką rozdzielcość samych pikseli tworząc wspomniany wcześniej podział segmentowy.

FCN (ang. Fully Convolutional Networks) stanowią szczególny rodzaj sieci neuronowych przeznaczonych do segmentacji obrazów. Składają się one wyłącznie z kombinacji warstw splotowych oraz poolingu. Są w stanie przetwarzać obrazy o dowolnej, zmiennej wielkości, w odróżnieniu od innych typów modeli, w których zastosowanie warstw gęstych (FCL) wymusza z góry ustalone rozmiary danych wejściowych.

Naprzemienne przepuszczanie obrazów przez wspomniane warstwy splotowe oraz pooling może powodować niską rozdzielcość wyjściowych rezultatów pracy tych sieci oraz rozmycie granic poszczególnych obiektów. Z tego powodu w nowoczesnych rozwiązaniach stosuje się dodatkowe mechanizmy zapobiegające tego typu trendom.

## 2.3 Modele generatywne

Koncepcja modeli generatywnych, w skrócie GANów, przedstawiona została w 2014 roku przez Iana Goodfellow oraz jego współpracowników na uniwersytecie w Montrealu [1]. Modele te stanowią połączenie dwóch głębokich sieci neuronowych działających przeciwstawnie do siebie nawzajem.

Pierwsza sieć to tak zwany generator. W odniesieniu do tematu pracy, jego działanie polega na generowaniu nowych obrazów, lub ich fragmentów na podstawie wektora szumów.

Obrazy te przekazywane są, równolegle z zestawem obrazów prawdziwych, do dyskryminatora stanowiącego drugą część modelu GAN. Działanie tej sieci neuronowej polega na określeniu (w skali 0 do 1), w jakim stopniu produkty wyjściowe generatora odpowiadają obrazom rzeczywistym.

W opisany modelu występuje zatem podwójna pętla sprzężenia zwrotnego. Dyskryminator określa autentyczność obrazów porównując je ze zdefiniowaną odgórnie bazą danych. Z kolei generator otrzymuje informację o skuteczności swojego działania ze strony dyskryminatora.

Model generatywny znajduje się w stanie ciągłego konfliktu. Generator dąży do jak najdokładniejszego fałszowania obrazów w celu oszukania dyskryminatora, którego celem jest z kolei jak najdokładniejsze wykrywanie podróbek. Obie sieci neuronowe nieustannie dążą do osiągnięcia przewagi nad rywalem w procesie treningu. Ciągła rywalizacja sprawia, że zarówno generator, jak i dyskryminator zyskują coraz wyższą skuteczność działania.

W praktyce modele generatywne są w stanie naśladować dowolną dystrybucję danych. Są w stanie kreować światy podobne do naszego w zakresie obrazu, dźwięku czy mowy. Można powiedzieć, że są to prawdziwi syntetyczni artyści.

#### **2.4 Autoenkodery**

### 3 PRZEGŁĄD ROZWIĄZAŃ

Na przestrzeni ostatnich paru lat pojawiło się wiele rozwiązań zastosowania sieci neuronowych do edycji obrazu, duża część z nich była przełomowa w swojej dziedzinie. Powstawały rewolucyjne architektury sieci oraz technologie z nimi związane. Takie cechy sieci jak niezwykła zdolność do generalizacji zdobytej wiedzy na nowe przypadki oraz olbrzymia elastyczność sprawiły, że znalazły one wiele rzeczywistych zastosowań.

W tym rozdziale skupiono się na przedstawieniu kilku interesujących rozwiązań dla omawianej problematyki.

#### 3.1 *Colorful image colorization*

Wraz z rozwojem sieci neuronowych, rosło zainteresowanie możliwościami zastosowania ich do kolorowania czarno-białych obrazów. Jedno z dostępnych rozwiązań tego zagadnienia zostało przedstawione przez grupę pracowników Uniwersytetu w Berkeley [5]. Zamiarem ich pracy było stworzenie modelu, który niekoniecznie odtwarza oryginalne barwy obrazu, ale generuje barwy prawdopodobne, zdolne przekonać ludzkiego obserwatora o autentyczności obrazu. Uzyskane rezultaty zostały przedstawione na Rysunku 3.1.



Rysunek 3.1: Efekt kolorowanie czarno-białych zdjęć przez wytrenowany model.

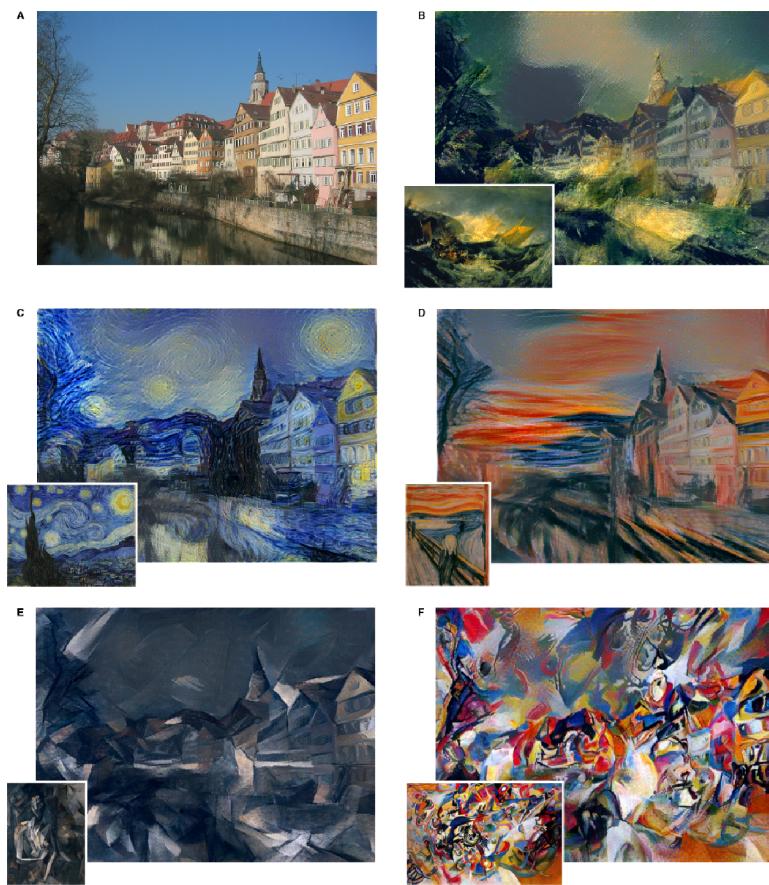
Wykorzystany model składa się z wielu warstw CNN, w których skład wchodzą warstwa filtrów konwolucyjnych, warstwa ReLU (ang. Rectified Linear Unit) oraz warstwa BatchNorm (ang. Batch normalization). Aby zapobiec utracie informacji przestrzennych, sieć nie posiada warstw łączących. Istotny był także sposób przygotowania zbioru danych do trenowania modelu. Obrazy ze zbioru uczącego były wpierw konwertowane do modelu YUV, a następnie kanał Y był podawany na wejście modelu, warstwy UV pełniły funkcję pożąданiej odpowiedzi w uczeniu nadzorowanym.

Ważnym aspektem zbadanym w artykule było także dobranie odpowiedniej funkcji kosztu. Nieuodpowiedni wybór skutkował desaturacją kolorowanych obrazów, jedną z potencjalnych przyczyn tego zjawiska może być tendencja sieci do tworzenia bardziej konserwatywnych odpowiedzi. Aby zniwelować ten efekt w modelu została zastosowana specjalna technika modyfikacji funkcji kosztu. Polega ona na przewidywaniu dystrybucji możliwych kolorów dla każdego piksela i zmienianiu kosztu dla modelu, w celu wyróżnienia rzadko spotykanych kolorów.

Powstałe rozwiązańe dowodzi olbrzymiego potencjału zastosowanie sieci neuronowych w dziedzinie pracy nad obrazami.

### 3.2 Image Style Transfer Using Convolutional Neural Networks

W roku 2016 został przedstawiony światu A Neural Algorithm of Artistic Style [6]. Wprowadził on przełom w dziedzinie przenoszenia stylu jednego obrazu na inny, a jego sukces opierał się na właściwym wykorzystaniu konwolucyjnych sieci neuronowych. Podstawą tego sukcesu było odkrycie przez Leona A. Gatys oraz jego współpracowników, że w CNN reprezentacja treści obrazu oraz jego stylu jest rozłączna. Umożliwia to wydobycie stylu przetwarzanego obrazu oraz połączenie go z treścią innego obrazu, czego dokonuje właśnie A Neural Algorithm of Artistic Style. Rezultaty takich operacji można zaobserwować na Rysunku 3.2



Rysunek 3.2: Obrazy będące kombinacją treści zdjęcia ze stylami kilku znanych dzieł sztuki.

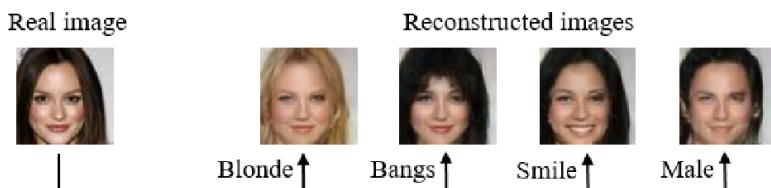
Do zbudowania modelu zostały użyte warstwy konwolucyjne oraz łączące z architektury VGG-Network [7], która została wytrenowana pod kątem rozpoznawanie obiektów i określania ich położenia. Dzięki temu sieć przetwarzając obraz tworzy jego reprezentację, która wraz z kolejnymi warstwami, przedstawia coraz wyraźniejszą informację o obiektach, a niekoniecznie o dokładnym wyglądzie obrazu. W modelu nie została użyta ani jedna warstwa gęsta, dzięki czemu na wyjściu możliwe jest otrzymanie dwuwymiarowego obrazu. Dla lepszej syntezy obrazów, w warstwach łączących zastosowano próbkowanie wartością średnią zamiast maksymalną. Takie zabiegi umożliwiają wyliczenie reprezentacji stylu z korelacji pomiędzy różnymi cechami w różnych warstwach konwolucyjnych.

Cały proces renderowania polega na odpowiednim zapisaniu w modelu treści oraz stylu obrazów otrzymanych przez wcześniejsze podanie na model tychże obrazów. Wpierw obraz, z którego pobierany jest styl, jest podawany na wejście sieci oraz przeliczany, reprezentacja stylu wyselekcjonowana z właściwych warstw jest przechowywana, obraz z treścią jest poddawany temu samemu procesowi, ale reprezentacja treści jest wyciągana z ostatnich warstw konwolucyjnych. W celu uzyskania fuzji obrazów, zapisane reprezentacje treści i stylu są zapisywane w tych warstwach modelu skąd zostały odczytane, a następnie na wejście podawany jest obraz składający się z losowego szumu białego. Następnie poprzez iteracyjną minimalizację funkcji kosztu, obraz wejściowy jest modyfikowany, co w rezultacie końcowym doprowadza do nałożenia zapisanego stylu na wczytaną treść.

A Neural Algorithm of Artistic Style jest świetnym przykładem, jak elastyczne mogą być interfejsy do modyfikacji obrazu oparte na technologii sieci neuronowych.

### 3.3 Invertible Conditional GANs for image editing

Edycja obrazów może być dokonywana na wielu różnych poziomach zaawansowania i abstrakcji, operacje takie jak nakładanie filtrów mogą być wykonywane przez proste algorytmy. Jednak w przypadku próby modyfikacji elementów na obrazie, algorytmy te nie będą w stanie dokonać semantycznych zmian ze względu na brak możliwości zrozumienia treści obrazu. Rozwiązanie tego problemu zostało przedstawione w postaci modelu IcGAN (ang. Invertible Conditional Generative Adversarial Network) w roku 2016 [8]. Zaprezentowany model był to enkoder z możliwością generowania wektora informacji o atrybutach obrazu połączony z warunkowym GAN zdolnym do kontrolowania cech generowanych obrazów na podstawie dodatkowej informacji warunkowej. Takie działanie umożliwia wprowadzania zmian w atrybutach generowanego obrazu uzyskiwanego na wyjście cGAN (ang. conditional Generative Adversarial Network). Rezultaty działania modelu można zaobserwować na Rysunku 3.3.



Rysunek 3.3: Obrazy generowane przez IcGAN.

Wykorzystany w IcGAN Ekonder w rzeczywistości składa się z dwóch podrzędnych Enkoderów, Enkoder  $E_z$  koduje wejściowy obraz do utajonego wektora  $z$  reprezentacji obrazu, natomiast Enkoder  $E_y$  generuje wektor informacji  $y$  oddających pewne kluczowe atrybuty obrazu. Enkody są trenowane z użyciem już wytrenowanego cGAN oraz obrazów rzeczywistych z etykietami ze zbioru uczącego. Zbadane zostały także różne podejścia interakcji między dwoma Enkoderami, wyróżnić można podejście, w którym Enkdodery są w pełni niezależne, podejście gdzie wyjście  $E_z$  jest zależne od wyjścia  $E_y$ , a także podejście gdzie  $E_z$  oraz  $E_y$  są połączone w jednej Enkoder o współdzielonych warstwach i dwóch wyjściach.

W przypadku cGAN możemy wyróżnić dwa najważniejsze czynniki, które trzeba mieć na uwadze. Pierwszym jest źródło wektora  $y$  podawanego na Generator. W przypadku Dyskryminatora  $y$  jest pobierany ze zbioru treningowego, jednakże w przypadku podawania tego samego wektora na Generator wystąpiła możliwość, że może dojść do niepożdanego przeuczenia modelu. Autorzy artykułu dokonali analizy tego rozwiązania, a także zbadali wydajność metody Bezpośredniej Interpolacji oraz Jądrowego estymatora gęstości. Wynikiem tych badań było stwierdzenie, że dla danej problematyki najlepiej sprawdza się podawanie wektora  $y$  ze zbioru uczącego, możliwość przeuczenia modelu została skomentowana następująco:

*'Jest to możliwe tylko, gdy informacje warunkowe są do pewnego stopnia unikatowa dla każdego obrazu. W tym przypadku, gdzie atrybuty obrazów są binarne, jeden wektor  $y$  może opisać wystarczająco duży i zróżnicowany podzbiór obrazów, zapobiegając nadmiernemu dopasowaniu się modelu do danego  $y$ .'*

Drugim czynnikiem jest warstwa Generatora i Dyskryminatora cGAN na którą podany jest wektor  $y$ . Guim Perarnau oraz jego współpracownicy ustalili, że najlepsze rezultaty otrzymuje się po podaniu wektora  $y$  na warstwę wejściową Generatora oraz pierwszą warstwę konwolucyjną Dyskryminatora.

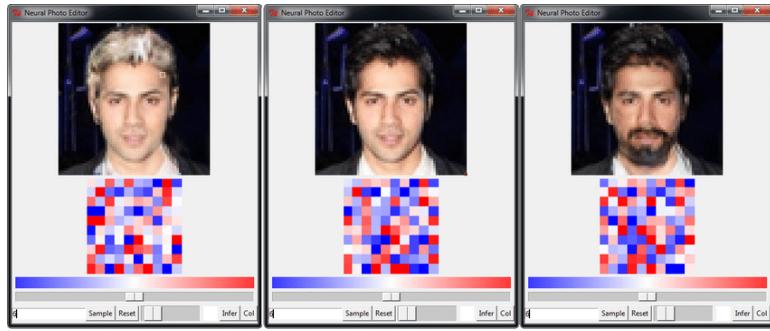
Ważnym spostrzeżeniem z analizy rozwiązania IcGAN jest obecność olbrzymiej ilości różnorodnych rozwiązań opartych na sieciach neuronowych, coraz to nowe architektury zostają wynalezione, aby udoskonalić zastosowania sieci neuronowych do przetwarzania i modyfikowania obrazów.

### 3.4 Neural photo editing

W 2017 roku Andrew Brock, Theodore Lim, J.M. Ritchie and Nick Weston zaprezentowali Neural Photo Editor [2], narzędzie do edytowania obrazu wyposażone w mechanizmy wykrywania kontekstu zmiany. Twórcy opisują swój twór następująco:

*'Interfejs wykorzystujący moc generatywnych sieci neuronowych do wprowadzania dużych, semantycznie spójnych zmian w istniejących obrazach.'*

Użytkowanie wygląda następująco: użytkownik pędzlem o określonym kolorze i rozmiarze maluje na wybranym obrazie, jednak zamiast zmieniać wartości pojedynczych pikseli, interfejs odczytuje kontekst edycji i wprowadza zmiany semantyczne w kontekście żądanej zmiany koloru. Efekt działania interfejsu został przedstawiony na Rysunku 3.4.



Rysunek 3.4: Efekt działania Neural Photo Editor.

Skuteczność NPE (Neural Photo Editor) polega na zastosowaniu IAN (ang. Introspective Adversarial Network), czyli sieci złożonej z połączonych VAE (ang. Variational Autoencoder) [9] oraz GAN, w taki sposób, że dekodująca sieć autoenkodera jest używana jako sieć generująca w GAN. Poprzez przechwytywanie przez model dalekosieżnych zależności, wykorzystanie bloku obliczeniowego bazującego na rozszerzonych splotach o współdzielonych wagach oraz dzięki zastosowaniu ulepszonej generalizacji, udało się osiągnąć dokładną rekonstrukcję obrazu bez strat na jakości detali.

Powstanie NPE utwierdza w przekonaniu, że aktualne możliwości sieci neuronowych do edycji obrazu znacznie przewyższają zwykłe algorytmy pod względem możliwości oraz uzależnienia od wkładu ludzkiego.

## **4 ZIMPLEMENTOWANE ROZWIĄZANIA**

W celu zbadania skuteczności sieci neuronowych jako narzędzi do edycji obrazu należało wybrać przykładowe zagadnienia z tej dziedziny, rozwiązać je z użyciem technik sztucznej inteligencji oraz ocenić ich skuteczność.

W tym rozdziale zostały przedstawione koncepcje rozwiązań wybranych zagadnień oraz ich implementacje.

#### 4.1 TorchFrame

Sztuczne sieci neuronowe stanowią dziedzinę nauki opartą w dużej mierze na eksperymentach. Dostarczane przez nie rozwiązania, choć często tak spektakularne, są mocno zawałowane, a droga do celu wiedzie przez kolejne treningi i doświadczalny dobór hiperparametrów sieci. Ogromne znaczenie w procesie uczenia ma również sposób przetworzenia danych wykorzystywanych do treningów, zarówno tych podawanych na wejście sieci, jak i tych otrzymywanych na jej wyjściu.

Niniejsza praca dotyczy przede wszystkim zastosowania sieci neuronowych w procesie przetwarzania obrazów cyfrowych. Wiąże się to z koniecznością przygotowywania zbiorów treningowych złożonych z ogromnej ilości danych wizyjnych poddanych odpowiedniemu przetworzeniu i filtracji, aby mogły właściwie spełnić swoją rolę w czasie treningu sieci.

Wspomniane zabiegi, jak również konieczność częstego powtarzania treningów, wymagają dużych nakładów pracy i czasu, aby mogły przynieść zamierzone efekty. W celu ułatwienia całego procesu przygotowany został framework TorchFrame stanowiący bazę pod eksperymenty podejmowane w ramach tej pracy i opisane w dalszych jej rozdziałach.

Sam framework umożliwia użytkownikom przeprowadzanie treningów sieci neuronowych, udostępniając wachlarz modyfikowalnych hiperparametrów oraz zestaw filtrów i metod przeznaczonych do obróbki danych treningowych. Właściwie użyty TorchFrame kontroluje przepływ danych uczących od początku do końca ograniczając konieczność ingerencji ze strony użytkownika do minimum, jednocześnie nie ograniczając przy tym potencjału eksperymentalnego sztucznych sieci neuronowych.

W ramach frameworka udostępniony został również prosty interfejs testowy umożliwiający ocenę efektów uzyskanych w procesie uczenia.

Niniejszy rozdział zostanie poświęcony analizie architektury TorchFrame'a oraz opisowi sposobu jego działania.

##### 4.1.1 PyTorch

U podstaw TorchFrame'a leżą mechanizmy innego frameworka, napisanego w języku Python i przeznaczonego do uczenia maszynowego o nazwie PyTorch. Jest to otwartoźródłowa biblioteka programistyczna stworzona przez oddział sztucznej inteligencji firmy Facebook. W jednym z artykułów [16] opublikowanych w ramach konferencji NIPS 2017 grupa badaczy opisuje ją następująco:

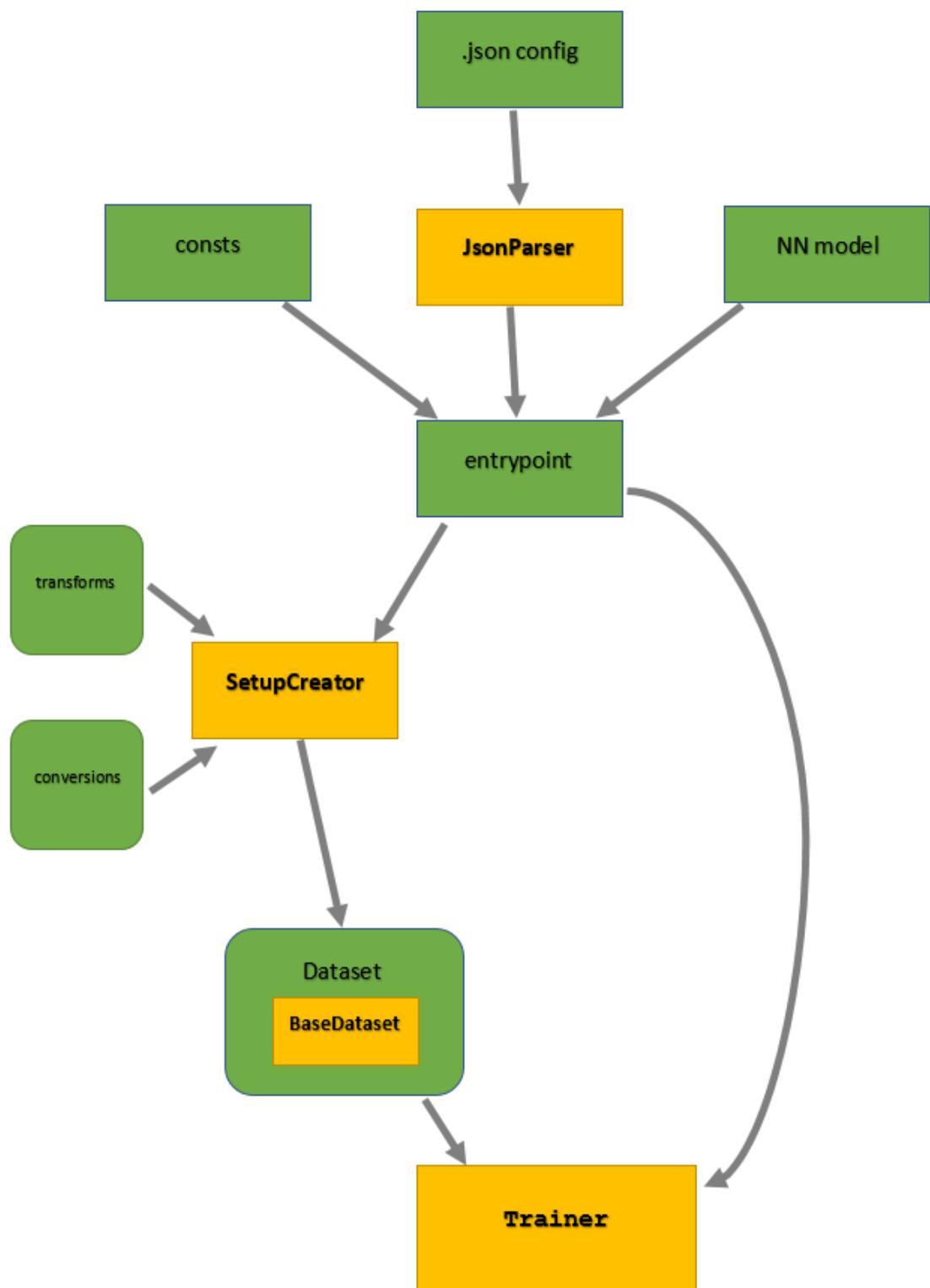
*'PyTorch - biblioteka zaprojektowana w celu umożliwienia szybkiego badania modeli uczenia maszynowego. Bazuje na kilku projektach, głównie Lua Torch, Chainer i HIPS Autograd, oraz dostarcza wysoko wydajnościowe środowisko z łatwym dostępem do automatycznego różnicowania modeli wykonywanych na różnych urządzeniach (CPU i GPU). Aby uczynić prototypowanie łatwiejszym, PyTorch nie podąża za podejściem symbolicznym używanym w wielu innych frameworkach do uczenia głębokiego, ale skupia się na różnicowaniu czysto imperatywnych programów, skupiając się na rozszerzalności i małym narzucie.' (...)*

*'PyTorch, podobnie jak większość innych bibliotek do uczenia głębokiego obsługuje automatyczne różnicowanie funkcji skalarnych w trybie wstecznym, czyli jedną z najważniejszych form automatycznego różnicowania dla aplikacji uczenia głębokiego, które zwykle różnicują skalarną funkcję celu.'*

TorchFrame wykorzystuje zdefiniowane w ramach PyTorch'a mechanizmy budowania sieci neuronowych korzystając z predefiniowanych metod opisu warstw sieci. Wykorzystuje ponadto zdefiniowane odgórnie hiperparametry, takie jak funkcje kosztu. Sam rdzeń TorchFrame'a bzuje na przytoczonym mechanizmie różnicowania, umożliwiając przeprowadzanie treningów na CPU oraz GPU.

#### 4.1.2 Architektura TorchFrame

TorchFrame podzielony został na bloki funkcjonalne, które powiązane ze sobą umożliwiają kontrolowany przepływ danych w procesie uczenia, a jednocześnie zapewniają elastyczność przy wprowadzaniu rozmaitych modyfikacji. Schemat funkcjonalny całego systemu przedstawiony został na Rysunku 4.1.



Rysunek 4.1: Architektura TorchFrame

Przepływ danych przez framework rozpoczyna się w obrębie trzech plików konfiguracyjnych, oznaczonych na przytoczonym schemacie odpowiednio jako: *consts*, *.json config* oraz *NN model*.

W pliku *consts* użytkownik TorchFrame definiuje zmienne środowiskowe, takie, jak ścieżki do plików kluczowych w procesie uczenia. Konieczne jest jedynie wskazanie położenia pliku konfiguracyjnego w formacie JSON. Pozostałe parametry są opcjonalne i mogą służyć do wskazania miejsca zapisu gotowego modelu sieci na dysku czy miejsca przechowywania danych treningowych, które zostaną następnie przetworzone i przekazane na wejście sieci w procesie uczenia.

*NN model* to skrypt języka Python, w którym użytkownik definiuje strukturę sieci neuronowej zgodnie z paradygmatem framework PyTorch przytoczonego w poprzednim rozdziale. Struktura TorchFrame narzuca na użytkownika konieczność zdefiniowania metody *forward*, która określa sposób obliczania wartości wyjściowych sieci na podstawie danych wejściowych między innymi poprzez określenie funkcji aktywacji poszczególnych warstw sztucznych neuronów.

Najważniejszym punktem zestawu konfiguracyjnego jest wspomniany już plik JSON. Udosępnia on szerokie możliwości manipulowania zarówno hiperparametrami sieci neuronowej, jak i szeregiem przekształceń możliwych do zaimplementowania na danych treningowych i testowych. Architektura TorchFrame zapewnia odpowiednie rozpropagowanie zgromadzonych tutaj danych w ramach procesu uczenia oraz podczas testów. Dokładna struktura tego pliku zostanie opisana w następnym rozdziale.

*JsonParser* jest klasą odpowiedzialną za odczytanie danych z pliku konfiguracyjnego, w formie słownika języka Python, oraz przekazanie ich w dalszą drogę w obrębie TorchFrame.

Centralnym punktem całej zaprezentowanej architektury jest *entrypoint*. Stanowi on punkt wejścia dla aplikacji użytkownika. Jego uruchomienie powoduje agregację danych ze wspomnianych już plików konfiguracyjnych, pogrupowanie ich oraz rozpropagowanie do dalszych komponentów odpowiedzialnych za przetwarzanie danych treningowych oraz przeprowadzanie samego treningu sieci neuronowej.

Część danych przekazywanych przez *entrypoint* trafia do *SetupCreator'a*. Komponent ten odpowiada za skomponowanie listy konwersji wymienionych w pliku konfiguracyjnym. Konwersje te zostaną następnie zaimplementowane na danych treningowych na etapie tworzenia zbioru uczącego. Źródłem danych do którego odwołuje się *SetupCreator* są skrypty *transforms* oraz *conversions*. Zawierają one gotową bazę przekształceń przeznaczonych do pracy na obrazach cyfrowych, a także metody formatowania danych do struktury *tensorów* (przypominających macierze z biblioteki *numpy* języka Python) wykorzystywanych przez bibliotekę PyTorch między innymi w mechanizmach automatycznego różnicowania modeli sieci neuronowych. Choć gotowa baza oferuje liczne przekształcenia, intuicyjna formuła pozwala użytkownikom w łatwy sposób definiować własne metody konwersji i implementować je zarówno na obrazach, jak i dowolnym innym rodzaju danych treningowych.

Tak skomponowany zestaw przekształceń wykorzystywany jest przy konstruowaniu zbioru uczącego w komponencie *Dataset*. Jego uniwersalny szkielet o nazwie *BaseDataset* odpowiada za sprawny przepływ danych w ramach TorchFrame zapewniając łatwy dostęp do zbioru implementowanych konwersji oraz umożliwiając ich zastosowanie poprzez dedykowane do tego metody. Posiada również funkcjonalność odczytu danych ze wskazanej przez użytkownika ścieżki w pliku *consts*, a także zdolność przetwarzanie ich na bieżąco, co pozwala zaoszczędzić pamięć w przypadku pracy na dużych zbiorach danych. W ramach tworzenia frameworka TorchFrame udostępnione zostały różne implementacje komponentu *Dataset* opierające się o *BaseDataset*. Ponownie jednak elastyczna struktura umożliwia użytkownikom skomponowanie własnych implementacji dostosowanych do indywidualnych potrzeb.

Ostatecznym elementem ścieżki treningowej, w którym skupiają się wszystkie zgromadzone dotąd dane jest komponent *Trainer*. Opiera się on o mechanizmy frameworka PyTorch w celu obliczania rezultatów pracy sieci, a także wyznaczanie kierunku uczenia w ramach funkcji celu i wstępnią propagację modyfikującą wagę sieci w czasie treningu. *Trainer* udostępnia na bieżąco dane pozwalające określić skuteczność uczenia sieci takie jak aktualny błąd sieci, ilość przetworzonych danych, czy numer epoki treningowej, w której aktualnie znajduje się model. W trakcie całego procesu możliwe jest również zapisywanie rezultatów w celu ich ponownego wykorzystania lub oceny.

#### 4.1.3 Konfiguracja w TorchFrame

Kluczem do właściwego przeprowadzenia treningu sztucznej sieci neuronowej jest odpowiedni dobór hiperparametrów. Ich modyfikacja w znaczący sposób wpływa na otrzymywane rezultaty przez co istotne jest, aby w ramach kolejnych eksperymentów można było w łatwy sposób dostosować je do potrzeb. Framework TorchFrame udostępnia pojedynczy interfejs użytkownika w postaci pliku konfiguracyjnego JSON skupiającego wszystkie najistotniejsze elementy w jednym miejscu i pozwalającego zachować przejrzystość stosowanej konfiguracji. W poniższym rozdziale opisane zostaną dostępne parametry wraz z wartościami, jakie mogą przyjmować.

1. **Net model** - ten parametr przyjmuje nazwę klasy, w której użytkownik zdefiniował strukturę sieci neuronowej na etapie inicjalizacji wraz z metodą *forward* definiującą sposób przepływu danych w ramach inferencji modelu.
2. **Criterion** - parametr ten pozwala zdefiniować funkcję kosztu używaną w procesie uczenia. W czasie treningu sieci neuronowych dąży się najczęściej do minimalizacji błędu określonego na bazie porównania rezultatów pracy sieci ze spodziewanym efektem. Zdefiniowana na tej podstawie funkcja błędu określana jest jako funkcja celu, a proces uczenia sieci sprowadza się do zdefiniowania zestawu wag, dla których jej wartość jest możliwie najmniejsza. Zagadnienie to opisane zostało w książce *Deep Learning* [4] z 2016 roku:

'Funkcja, którą chcemy minimalizować, lub maksymalizować nazywana jest funkcją celu lub kryterium. W przypadku minimalizacji możemy również nazywać ją funkcją kosztu, funkcją straty, lub funkcją błędu.'

Na funkcji kosztu spoczywa bardzo ważne zadanie. Musi ona wiernie destylować wszystkie aspekty modelu w jedną liczbę, w taki sposób, aby ulepszenia tej liczby były oznaką poprawy całego modelu. [17]

*'Funkcja kosztu redukuje wszystkie dobre i złe aspekty potencjalnie złożonego systemu do pojedynczej liczby, wartości skalarnej, która umożliwia klasyfikację i porównanie możliwych rozwiązań.'*

Dobór odpowiedniej funkcji kosztu może stanowić spore wyzwanie, ponieważ funkcja ta musi uchwycić właściwości danego problemu i być motywowaną założeniami istotnymi z punktu widzenia realizowanego projektu. [17]

*'(...) Dlatego ważne jest, aby funkcja wiernie reprezentowała nasze cele projektowe. Jeśli wybierzemy słabą funkcję błędu i uzyskamy niezadowalające wyniki, to wina za złe określenie celu poszukiwań spoczywa na nas.'*

Opisy rozmaitych funkcji kosztu dostępnych w TorchFrame zamieszczone zostały w tabeli 1.

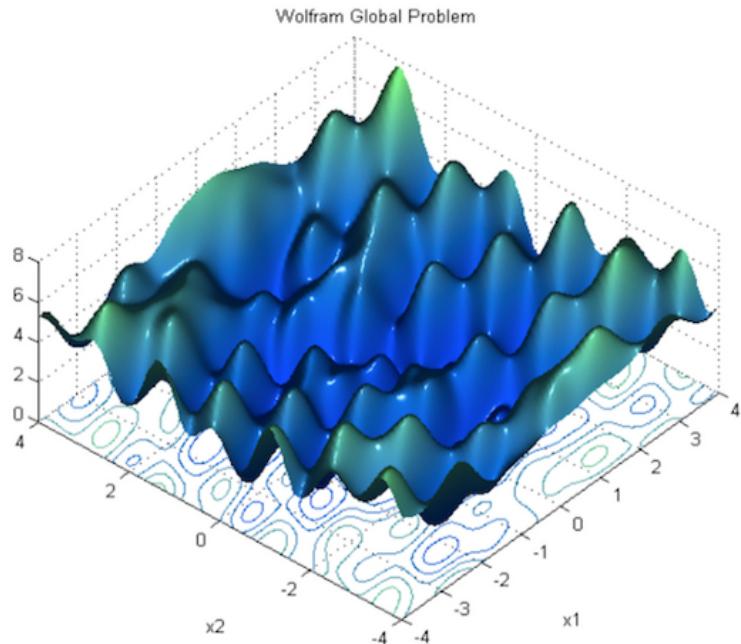
Funkcja kosztu	Opis
<b>L1Loss</b>	<p>Funkcja kosztu mierząca średni błąd bezwzględny pomiędzy odpowiadającymi sobie elementami ze zbioru wejściowego i docelowego. Można ją opisać za pomocą następującego wzoru:</p> $l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n =  x_n - y_n ,$ <p>gdzie <math>N</math> jest rozmiarem pojedynczego pakietu danych, a <math>x</math> i <math>y</math> to tensory o arbitralnym kształcie, z których każdy posiada <math>n</math> elementów.</p> <p>Parametry:</p> <ul style="list-style-type: none"> <li>reduction - parametr ten może przyjmować trzy wartości <i>none</i>, <i>mean</i> oraz <i>sum</i>. Pierwsza opcja spowoduje, że wartość funkcji celu zostanie wyznaczona zgodnie z podanym powyżej wzorem. Wartość <i>mean</i> spowoduje wyznaczenie średniej wartości elementów wyjściowych. <i>Sum</i> oznacza natomiast, że wyznaczona zostanie ich suma.</li> </ul>
<b>MSELoss</b>	<p>Funkcja kosztu mierząca średni błąd kwadratowy pomiędzy każdym elementem wejściowym <math>x</math> i celem <math>y</math>. Opisuje ją poniższy wzór:</p> $l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = (x_n - y_n)^2,$ <p>gdzie <math>N</math> jest rozmiarem pojedynczego pakietu danych, a <math>x</math> i <math>y</math> to tensory o arbitralnym kształcie, z których każdy posiada <math>n</math> elementów.</p> <p>Parametry:</p> <ul style="list-style-type: none"> <li>reduction - czytaj <i>reduction</i> dla funkcji <i>L1Loss</i>.</li> </ul>

<b>KLDivLoss</b>	<p>Funkcja kosztu nazywana rozbieżnością Kullback'a - Leibler'a. Jest użyteczną miarą dla rozkładów ciągłych i często jest przydatna podczas wykonywania bezpośredniej regresji w przestrzeni (dyskretnie próbko-wanych) ciągłych rozkładów wyjściowych.</p> <p>Kryterium to wymaga, aby rozmiary tensorów wejściowych i wyjściowych były identyczne.</p> <p>Wzór matematyczny opisujący rozbieżność KL przedstawiony został po-nizej:</p> $l(x, y) = L = \{l_1, \dots, l_N\}, l_n = y_n \cdot (\log y_n - x_n),$ <p>gdzie indeks <math>N</math> obejmuje wszystkie wymiary wejściowe, a <math>L</math> ma ten sam ksztalt, co wejście.</p> <p>Parametry:</p> <ul style="list-style-type: none"> <li>reduction - poza wartościami opisanymi dla funkcji <math>L1Loss</math> może przyjmować również parametr <math>batchmean</math>. Wymusza on sumowanie wartości wyjściowych, a następnie podział sumy przez rozmiar pakietu danych.</li> </ul>
<b>BCELoss</b>	<p>Kryterium wyznaczające wartość Binarnej Entropii Krzyżowej pomiędzy wyjściem sieci, a spodziewanymi rezultatami jej pracy.</p> <p>Opisuje to poniższy wzór:</p> $l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = -w_n[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)],$ $w_n = waga[n],$ <p>gdzie <math>N</math> jest rozmiarem pakietu danych.</p> <p>Parametry:</p> <ul style="list-style-type: none"> <li>weight - manualna wartość wagi przeskalowującej funkcję kosztu każdego elementu w pakiecie danych.</li> <li>reduction - czytaj reduction dla funkcji <math>L1Loss</math>.</li> </ul>
<b>SmoothL1-Loss</b>	<p>Kryterium przyjmujące postać funkcji <math>MSE</math> w przypadku, gdy wartość błędu bezwzględnego spada poniżej 1 oraz funkcji <math>L1</math> w przeciwnym wy-padku. Funkcja ta jest mniej czuła na wartości odstające niż <math>MSELoss</math>, a w niektórych przypadkach zapobiega zjawisku eksplodującego gradientu. Znana jest również jako funkcja kosztu <math>Huber'a</math>.</p> <p>Wzór opisujący:</p> $\text{loss}(x, y) = \frac{1}{n} \sum_i z_i,$ <p>gdzie <math>z_i</math> zdefiniowane jest następująco:</p> $y = \begin{cases} 0.5 \cdot (x_i - y_i)^2 & \text{gdy }  x_i - y_i  < 1 \\  x_i - y_i  - 0.5 & \text{gdy }  x_i - y_i  \geq 1 \end{cases}$ <p>Parametry:</p> <ul style="list-style-type: none"> <li>reduction - czytaj reduction dla funkcji <math>L1Loss</math>.</li> </ul>

Tabela 1: Funkcje kosztu w TorchFrame

3. **Optimizer** - pozwala wybrać rodzaj optymalizatora używanego w procesie uczenia sieci neuronowej. Jest to algorytm odpowiedzialny za aktualizowanie wag modelu. Korzysta on z wartości funkcji kosztu, jak z drogowskazu wskazującego kierunek prowadzący do osiągnięcia globalnego minimum w procesie minimalizacji, jakim jest trening sieci.

Zadanie zlokalizowania globalnego minimum nie jest zadaniem trywialnym. Optymalizacja sieci neuronowych jest optymalizacją niewypukłą. Oznacza to, że funkcja celu posiada wiele optimów, z czego tylko jedno jest poszukiwanym optimum globalnym. Przykładowa płaszczyzna funkcji celu przedstawiona została na Rysunku 4.2.



Rysunek 4.2: Przykładowa płaszczyzna funkcji celu

Można powiedzieć, że argumentami funkcji celu są wagi modelu sieci neuronowej. Każde kolejne połączenie w sieci posiadające własną wagę zwiększa wymiar płaszczyzny poszukiwań. Oznacza to, że dla modelu opisanego trzema wagami obszarem poszukiwań będzie płaszczyzna trójwymiarowa. Zazwyczaj modele sieci są jednak dużo większe. I tak dla modelu, na który składa się przykładowo sto wag poszukiwania optimum są w rzeczywistości prowadzone na hiperplaszczyźnie posiadającej sto wymiarów.

Najbardziej podstawowym algorytmem optymalizacji leżącym u podstaw innych metod jest tak zwany spadek gradientu. Najlepiej opisuje go programistyczna formuła aktualizacji wag sieci:

$$\Theta = \Theta - \eta \cdot \nabla J(\Theta),$$

gdzie  $\eta$  jest długością kroku treningowego, a  $\nabla J(\Theta)$  oznacza gradient funkcji kosztu zależnej od wag sieci  $\Theta$ . Warto zauważyć, że wartość gradientu wskazuje na płaszczyźnie funkcji celu położenie maksimum, dlatego w procesie minimalizacji musimy kierować się w przeciwną stronę i odejmować gradient od wag modelu.

Sam gradient wyznaczany jest z wykorzystaniem bardzo popularnego obecnie algorytmu nazywanego propagacją wsteczną. Jego ogólna idea jest stosunkowo prosta. Rezultaty pracy sieci ewaluowane są względem pożądanych wyników w ramach opisanej w poprzednim punkcie funkcji celu. Jeśli wyniki oceny nie są zadowalające wagi sieci są modyfikowane, a cała operacja powtarzana jest aż do momentu zakończenia treningu.

Raul Rojas w swojej książce [18] z 1996 roku tak opisuje etapy tego algorytmu:

*'Rozważmy sieć neuronową z pojedynczym wejściem rzeczywistym  $x$  i funkcją sieci  $F$ . Pochodna  $F(x)$  jest obliczana w dwóch fazach:*

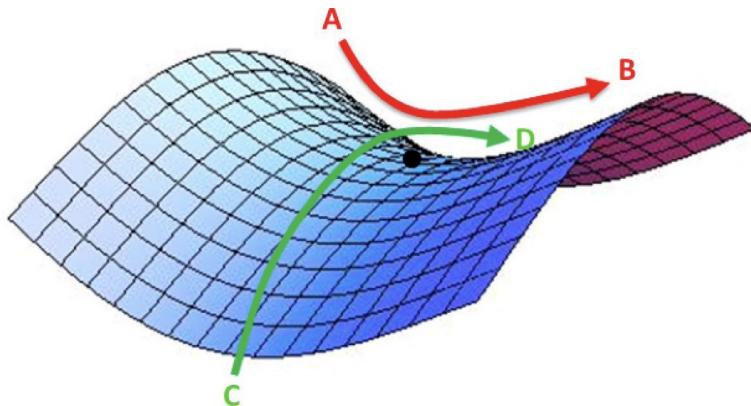
*Przekazywanie: wejście  $x$  jest podawane do sieci. Funkcje aktywacji i ich pochodne są oceniane w każdym węźle (neuronie) sieci. Pochodne są przechowywane.*

*Propagacja wsteczna: stała 1 jest podawana do jednostki wyjściowej i sieć biegnie wstecz. Informacje przychodzące do węzła są dodawane, a wynik jest mnożony przez wartość zmagażinowaną w lewej części jednostki. Rezultat jest transmitowany na lewo od jednostki. Wynik zgromadzony w jednostce wyjściowej stanowi pochodną funkcji sieci względem  $x$ .'*

Sam algorytm wstecznej propagacji działa poprawnie również dla sieci posiadających więcej niż jedną jednostkę wejściową, w których zaangażowanych jest więcej niezależnych zmiennych. Idea jego działania pozostaje wówczas ta sama.

Wynika z tego, że algorytm wstecznej propagacji, a co za tym idzie czerpiący z niego w czystej postaci spadek gradientu, cechują się dużą stałością i przewidywalnością działania. W wielu przypadkach jest to pożądana cecha, jednak zwiększąca się ilość parametrów sieci może powodować pojawianie się trudności wymagających od algorytmów optymalizacji większej elastyczności w podejmowaniu działań.

Do trudności takich zaliczyć można między innymi dobrą odpowiednią długością kroku treningowego tak, aby znaleźć złoty środek między szybkością wyznaczenia rozwiązania, a jego dokładnością. Dodatkowo w przypadku wspomnianych algorytmów wszystkie parametry sieci modyfikowane są w jednakowy sposób, co nie zawsze jest korzystne. Przykładowo, gdy dane treningowe są rzadkie, a różne cechy występują w nich z różnymi częstotliwościami, pożądanym zjawiskiem byłoby przeprowadzanie większych aktualizacji wag sieci dla cech występujących rzadziej. Pozostaje także niebezpieczeństwo związane z potencjalnym utknięciem modelu w jednym z minimów lokalnych, lub co gorsza w puncie siodłowym, którego przykładowy kształt przedstawia Rysunek 4.3. Posiada on jednocześnie cechy zarówno minimum, jak i maksimum lokalnego i jest najczęściej otoczony przez płaskowyż cechujący się tą samą wartością funkcji kosztu. Bardzo często uniemożliwia to ucieczkę takim algorytmom jak spadek gradientu, ponieważ wartość gradientu jest wówczas bliska zeru we wszystkich wymiarach.



Rysunek 4.3: Przykładowy kształt punktu siodłowego

W odpowiedzi na opisane problemy opracowana została technika tak zwanego pędu. Ogranicza ona oscylacje spadku gradientu w niewłaściwych kierunkach i przyspiesza proces zbiegania rozwiązania. Pęd sprowadza się do dodawania ułamka  $\gamma$  wektora aktualizacji z poprzedniego kroku algorytmu do bieżącego wektora aktualizacji. Przedstawia to poniższy wzór:

$$V(t) = \gamma \cdot V(t - 1) + \eta \cdot \nabla J(\Theta),$$

który prowadzi ostatecznie do następującej modyfikacji parametrów:

$$\Theta = \Theta - V(t).$$

Nazwa tej metody nawiązuje do pędu znanego z fizyki. Można powiedzieć, że w miarę uczenia modyfikacje wag sieci nabierają prędkości we właściwym kierunku, co sprawia, że nie są już tak podatne na ewentualne nieprawidłowe zmiany kierunku i szybciej osiągają właściwy cel. Badacz Yurii Nesterov zauważał jednak zasadniczą wadę związaną z metodą pędu. W sytuacji, gdy algorytm przemieszcza się w dół zbocza funkcji celu w żaden sposób nie kontroluje, czy znalazł się już na dnie. Gdy je osiąga wartość pędu jest dość wysoka, co może spowodować, że punkt optymalny zostanie pominięty lub osiągnięty z opóźnieniem. Nesterov zaproponował nieco inne rozwiązanie. W jego metodzie najpierw wykonywany jest duży skok bazujący na poprzedniej wartości pędu, a następnie w nowym, potencjalnym położeniu, obliczana jest wartość gradientu, która dokonuje korekcji miejsca docelowego. Dopiero wówczas dokonywana jest rzeczywista aktualizacja parametrów modelu. Metodę tę można przedstawić za pomocą następującego wzoru:

$$V(t) = \gamma \cdot V(t - 1) + \eta \cdot \nabla J(\Theta - \gamma \cdot V(t - 1)),$$

który prowadzi do aktualizacji:

$$\Theta = \Theta - V(t).$$

Argument  $\nabla J, \Theta - \gamma \cdot V(t - 1)$  odpowiada w tym przypadku za określenie przewidywanego punktu docelowego. Pozwala to wyznaczyć wartość gradientu nie dla obecnego położenia modelu na płaszczyźnie funkcji celu, ale dla domniemanego położenia osiągniętego w przyszłości. Czyni to algorytm optymalizacji bardziej responsywnym na ewentualne zmiany i ogranicza ryzyko ominięcia punktu optymalnego ze względu na zbyt dużą wartość pędu.

Bazując na opisanych tutaj algorytmach przygotowane zostały metody optymalizacji dostępne w TorchFrame, a zaczerpnięte z biblioteki PyTorch. Wiele z nich wprowadza również dodatkowe funkcjonalności, z których najistotniejsze opisane zostały w Tabeli 2.

Optymalizator	Opis
<b>SGD</b>	<p>Algorytm stochastycznego spadku gradientu. Stanowi wariację bazowego spadku gradientu polegającą na aktualizowaniu parametrów sieci dla każdego przykładu uczącego. Opisuje to następujący wzór:</p> $\Theta = \Theta - \eta \cdot \nabla J(\Theta; x(i); y(i)),$ <p>gdzie <math>\{x(i), y(i)\}</math> jest zbiorem poszczególnych par treningowych. Stochastyczny spadek gradientu cechuje się częstszymi aktualizacjami i zmiennymi oscylacjami na płaszczyźnie funkcji celu. Czyni go to wolniejszym od algorytmu bazowego w dążeniu do rozwiązania, jednak zwiększa prawdopodobieństwo odkrycia bardziej optymalnych minimów.</p> <p>Możliwe jest również zastosowanie opisanych wcześniej mechanizmów pędu oraz algorytmu Nesterov'a w celu poprawy wydajności działania tego algorytmu.</p>
<b>Adagrad</b>	<p>Algorytm pozwalający dopasować długość kroku treningowego do poszczególnych wag modelu. Parametry rzadko biorące udział w pracy sieci otrzymują wówczas większe aktualizacje niż te występujące stosunkowo często. Adagrad nadaje się przez to do pracy z rzadkimi zbiorami treningowymi. Wzór prezentujący sposób aktualizacji pojedynczego <math>i</math>-tego parametru w sieci prezentuje się następująco:</p> $\Theta_{t+1,i} = \Theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i},$ <p>gdzie <math>G_{t,ii}</math> jest diagonalną macierzą zawierającą sumy kwadratów gradientów wyznaczonych aż do chwili <math>t</math>, <math>g_{t,i}</math> jest bieżącą wartością gradientu dla parametru <math>\Theta_i</math>, a <math>\epsilon</math> oznacza współczynnik wygładzający, zabezpieczający przed ewentualnym dzieleniem przez 0. Wzór ten oznacza, że Adagrad modyfikuje długość kroku uczącego w każdej iteracji <math>t</math> dla każdego parametru <math>i</math> bazując na przeszłych wartościach gradientu wyznaczonych dla tej wagi.</p> <p>Istotną wadą tego optymalizatora jest fakt ciągłego zmniejszania długości kroku uczącego ze względu na rosnącą nieustannie wartość sumy kwadratów gradientów w mianowniku. W krytycznych przypadkach może to doprowadzić do całkowitego zaniku tego kroku, co w konsekwencji prowadzi do zablokowania możliwości dalszego treningu sieci.</p>

	<p>Algorytm ten stanowi rozszerzenie optymalizatora Adagrad. Rozwiązuje trąpiący go problem zanikającego kroku treningowego poprzez wprowadzenie ograniczenia, co do ilości przeszłych gradientów mających wpływ na aktualizację wag w bieżącej iteracji. Dodatkowo zamiast przechowywać określona liczbę przeszłych kwadratów gradientów, AdaDelta wyznacza ich sumę poprzez rekursywne definiowanie zanikającej średniej wartości przeszłych kwadratów tych gradientów. Sposób działania tego mechanizmu opisuje poniższy wzór:</p> $E[g^2]_t = \gamma \cdot E[g^2]_{t-1} + (1 - \gamma) \cdot g_t^2,$ <p>gdzie <math>E[g^2]_t</math> oznacza średnią wartość sumy kwadratów gradientów w bieżącej iteracji, <math>E[g^2]_{t-1}</math> definiuje tę średnią dla poprzedniego kroku, a <math>g_t^2</math> stanowi kwadrat bieżącej wartości gradientu. Parametr <math>\gamma</math> pełni tutaj podobną rolę, co w przypadku mechanizmu pędu. Ostatecznie parametry modelu aktualizowane są zgodnie z następującą formułą:</p> $\Delta\Theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$
<b>Adam</b>	<p>Optymalizator Adam podobnie jak AdaDelta pozwala wyznaczać adaptacyjne długości kroku uczącego dla każdego parametru sieci. Przechowuje on również eksponencjalnie zanikającą średnią wartość kwadratów przeszłych gradientów <math>v_t</math>. Poza tym przechowuje również eksponencjalnie zanikającą średnią przeszłych gradientów <math>m_t</math> podobną do wartości pędu.</p> <p>Wartości <math>m_t</math> oraz <math>v_t</math> są szacunkami odpowiednio pierwszego momentu (średnia) oraz drugiego momentu (wariancja niecentrowana), stąd nazwa metody.</p> <p>Autorzy Adama zauważyli, że <math>m_t</math> oraz <math>v_t</math>, które inicjalizowane są jako wektory zerowe dążą tendencyjnie do zera zwłaszcza podczas początkowych iteracji. Aby przeciwdziałać temu procesowi wyznaczane są odpowiednie estymaty korekcyjne obu momentów zgodnie ze wzorami:</p> $m'_t = \frac{m_t}{1 - \beta_1^t}$ $v'_t = \frac{v_t}{1 - \beta_2^t}$ <p>Na tej podstawie wyznaczana jest reguła aktualizacyjna Adama, postaci:</p> $\Theta_{t+1} = \Theta_t - \frac{\eta}{\sqrt{v'_t + \epsilon}} \cdot m'_t.$ <p>Wartości parametrów <math>\beta_1</math>, <math>\beta_2</math> oraz <math>\epsilon</math> mogą być dobierane eksperymentalnie, jednak rekomendowane jest przyjęcie następujących liczb: <math>\beta_1 = 0.9</math>, <math>\beta_2 = 0.999</math>, <math>\epsilon = 10^{-8}</math>.</p>

Tabela 2: Optymalizatory w TorchFrame

4. **Scheduler** - umożliwia wybór planisty. Odpowiada on za modyfikowanie długości kroku treningowego w zależności od aktualnego numeru epoki, w jakim znajduje się proces uczenia.

Jest to istotny element pozwalający dynamicznie dostosowywać krok do aktualnego położenia modelu na hiperplaszczyźnie funkcji celu. Przykładowo gdy sieć znajduje się daleko od optimum globalnego, lub gdy wpadnie w jedno z optimów lokalnych możliwe jest zwiększenie kroku treningowego, aby przyspieszyć proces optymalizacji. Gdy natomiast model znajduje się w pobliżu optimum globalnego długość kroku może zostać zmniejszona, co przełoży się na zwiększenie dokładności uzyskanych wyników.

5. ***Init epoch*** - parametr wskazujący numer epoki od którego rozpocznie się trening sieci neuronowej.
6. ***Training epochs*** - ilość epok, jaką przejdzie model w danej sesji treningowej. W każdej epoce modelowi przekazane zostaną wszystkie dane wskazane w zbiorze treningowym.
7. ***Training Monitoring Period*** - określa co ile pakietów danych obiekt *Trainer* będzie wyświetlał w konsoli dane treningowe, takie jak aktualna wartość funkcji kosztu.
8. ***Saving Period*** - parametr określający co ile epok obiekt *Trainer* będzie dokonywał zapisu pośrednich wyników treningu sieci.
9. ***Dataloader Parameters*** - zestaw trzech parametrów.

*Batch size* określa ilość pojedynczych próbek ze zbioru treningowego które zostaną zgromadzone w jeden pakiet danych. W trakcie procesu uczenia sieci wartość funkcji kosztu wyznaczana jest dla wszystkich elementów pakietu, a następnie uśredniana i dopiero wtedy wykorzystywana przez optymalizator do aktualizacji wag modelu.

*Shuffle* pozwala określić, czy dane ze zbioru treningowego zostaną posortowane, aby uniknąć powtarzalnej kolejności ich przekazywania na wejście sieci. Pozwala to lepiej przygotować sieć do pracy z danymi, których nie widziała w procesie uczenia.

*Num workers* to parametr określający ilość procesów równolegle przetwarzających dane i tym samym przyspieszających cały proces treningu.

10. ***Retrain*** - wartość *true* tego parametru powoduje, że TorchFrame podejmie próbę wczytania modelu sieci, optymalizatora oraz planisty zgodnie ze ścieżkami zdefiniowanymi w pliku *consts*. Brak którejkolwiek ścieżki spowoduje, że dany parametr zostanie zainicjowany losowo, co pozwala dotrenować sieć w dowolnej konfiguracji parametrów startowych.
11. ***Train on gpu*** - wartość *true* tego parametru sprawi, że TorchFrame podejmie próbę rozpoznania, czy dostępna jest graficzna jednostka obliczeniowa, a następnie przeniesie na nią dane umożliwiając przeprowadzanie za jej pomocą obliczeń w celu przyspieszenia procesu uczenia.
12. ***Dataset*** - stanowi zbiór parametrów określających sposób wstępnego przetwarzania danych w celu przygotowania zbioru uczącego.

*Name* - zawiera nazwę klasy definiującej sposób przetwarzania i przekazywania danych ze zbioru uczącego. Klasa ta powinna być oparta o szkielet *BaseDataset* zapewniający sprawny przepływ danych przez framework TorchFrame.

*Input conversions* - jest to lista konwersji przeznaczonych do przetwarzania danych treningowych przekazywanych na wejście sieci. Każda konwersja składa się z nazwy popartej definicją w pliku *conversions* oraz z parametrów, których używa w trakcie pracy na danych.

*Output conversions* - podobnie jak poprzedni parametr jest to lista konwersji. Ich implementacja pozwala przygotować zbiór danych wyjściowych wykorzystywanych jako punkt odniesienia dla rezultatów pracy sieci przy wyznaczaniu wartości funkcji celu.

*Transforms* - lista transformacji zdefiniowanych w pliku *transforms*, zapewniających między innymi formatowanie danych do postaci tensorów obsługiwanych przez bazowe mechanizmy biblioteki PyTorch.

13. **Additional params** - słownik języka Python, w którym użytkownik TorchFrame ma możliwość zdefiniowania dowolnych dodatkowych parametrów, jakie mogą wydać się pomocne w procesie treningu sieci neuronowej.

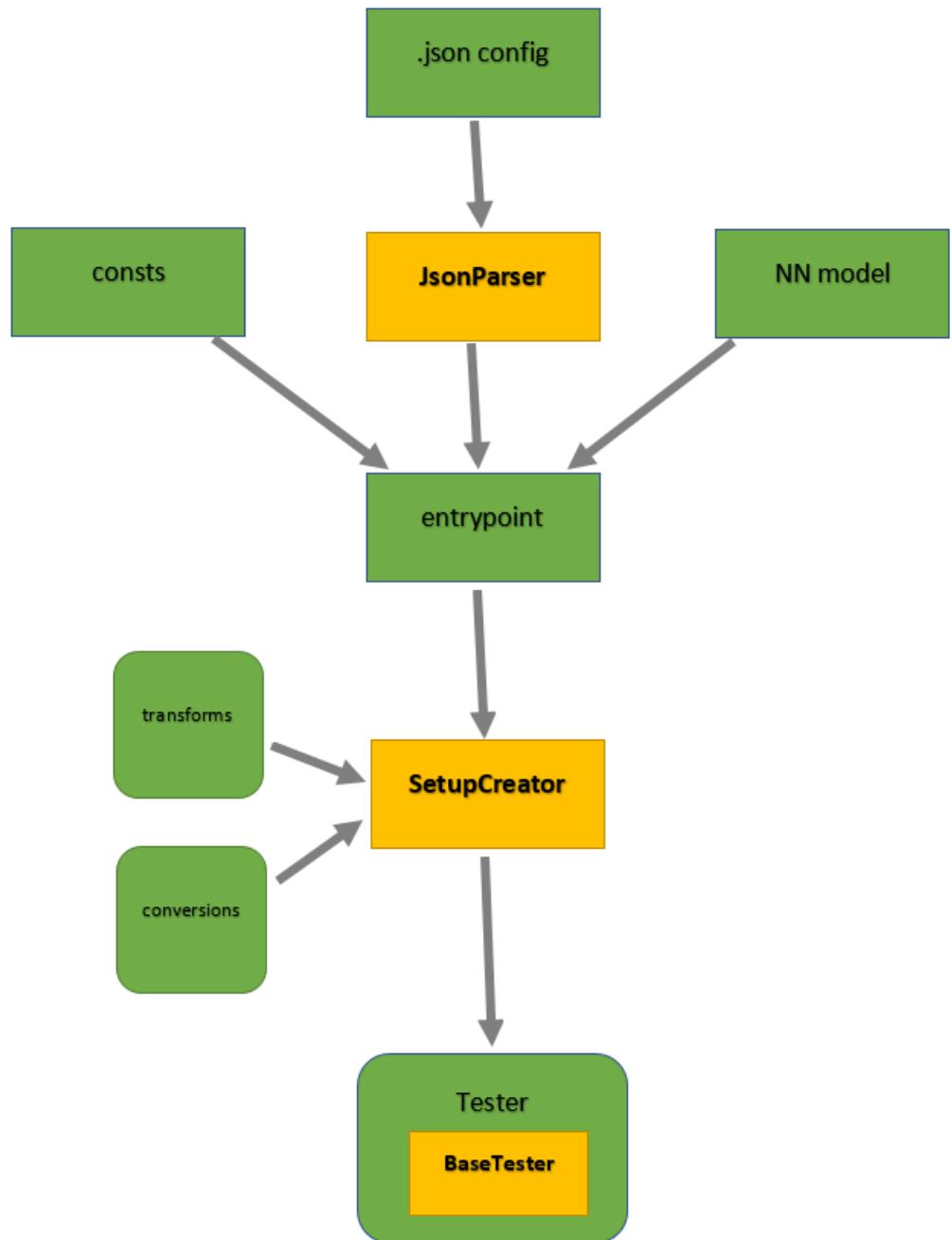
#### 4.1.4 Testowanie w TorchFrame

Podstawą do określenia skuteczności przeprowadzanych treningów sieci neuronowych jest nie tylko obserwacja parametrów takich jak wartość funkcji kosztu. Bardzo często konieczna jest wizualizacja wyników działania wytrenowanych modeli. Komponentowa budowa TorchFrame pozwala w łatwy sposób osiągnąć ten cel. Na bazie fragmentów wydzielonych z głównej architektury skomponowany został system testowy, którego schemat przedstawia Rysunek 4.4.

Większość przedstawionych na schemacie elementów zachowuje funkcjonalności opisane w rozdziale *Architektura TorchFrame*. Istotną różnicą jest pojawienie się obiektu *Tester*, którego zadaniem jest przeprowadzanie zdefiniowanych przez użytkownika testów. Podobnie jak obiekt *Dataset* bazuje on na szkielecie *BaseTester* odpowiadającym za funkcjonalne połączenie z resztą frameworka. W ramach tej pracy przygotowane zostały różne warianty testowe, jednak elastyczna forma szkieletu pozwala użytkownikom na definiowanie własnych scenariuszy.

Inaczej zachowuje się również *SetupCreator*. Element ten wyposażony został w specjalną metodę przeznaczoną do przygotowania środowiska testowego na bazie parametrów dostarczonych w pliku konfiguracyjnym JSON.

Plik ten różni się od tego omawianego w przypadku procesu treningowego. Jego zawartość została w znacznym stopniu ograniczona, gdyż proces testowy nie wymaga już tak wielu parametrów konfiguracyjnych. Pozwala to zachować przejrzystość środowiska testowego i ułatwia rozpropagowanie danych w systemie.



Rysunek 4.4: Architektura testowa TorchFrame

## 4.2 Filtry AI

Filtrowanie obrazów cyfrowych to bardzo popularny i powszechnie stosowany obecnie proces. Pozwala wyostrzyć niewyraźne zdjęcie, zmienić kontrast obrazu, czy zniwelować szumy tła. W rzeczywistości filtrowanie to nic innego, jak operacja matematyczna wykonywana na pikselach. Wykorzystywanie wartości wielu pikseli obrazu źródłowego w celu określenia wartości pojedynczego piksela w obrazie wynikowym. Sposób w jaki wartości te są pobierane oraz przetwarzane określają tak zwane maski. Przyjmują one postać macierzy kwadratowych różnych rozmiarów, a przechowywane w nich wartości decydują o wyniku filtracji.

Poniższy rozdział tej pracy spróbuje udzielić odpowiedzi na pytanie, czy sieci neuronowe mogą sprawnie posłużyć w procesie filtrowania obrazów. Składa się na niego seria eksperymentów, w których specjalnie dobrane modele sieci spróbowają odtworzyć wartości masek użytych do przygotowania danych treningowych, a następnie wykorzystają je do przetworzenia zupełnie nowych obrazów.

Dane referencyjne składają się z zestawu obrazów przetworzonych za pomocą filtrów wbudowanych w bibliotekę *OpenCV* takich, jak filtr Sobela, czy sepii.

Wszystkie modele wytrenowane zostały w oparciu o framework TorchFrame.

### 4.2.1 Filtr Sobela

Jednym z podstawowych i najbardziej znanych obecnie filtrów obrazu jest filtr Sobela-Feldmana, który zaprezentowany został po raz pierwszy w 1968 roku na konferencji Laboratorium Sztucznej Inteligencji uniwersytetu Stanforda (*SAIL*). Znajduje on przede wszystkim zastosowanie w procesie wykrywania krawędzi na obrazach cyfrowych. Sam Irwin Sobel opisuje ten filtr następująco [19]:

*Motywacją w rozwijaniu tego rozwiązania było stworzenie wydajnej obliczeniowo estymacji gradientu, która byłaby bardziej izotropowa niż popularny wówczas operator "Krzyża Roberts'a".*

Operator izotropowy to, w kontekście przetwarzania obrazów, operator, którego działanie jest równoważne dla wszystkich kierunków na obrazie. Filtr Sobela wyznacza przybliżenie gradientu funkcji natężenia obrazu. Dla każdego pojedynczego piksela wynikiem jego działania jest wektor gradientu (lub jego długość) wskazującego kierunek wzrostu intensywności obrazu, wyznaczony na bazie otaczających wartości ośmiu innych pikseli.

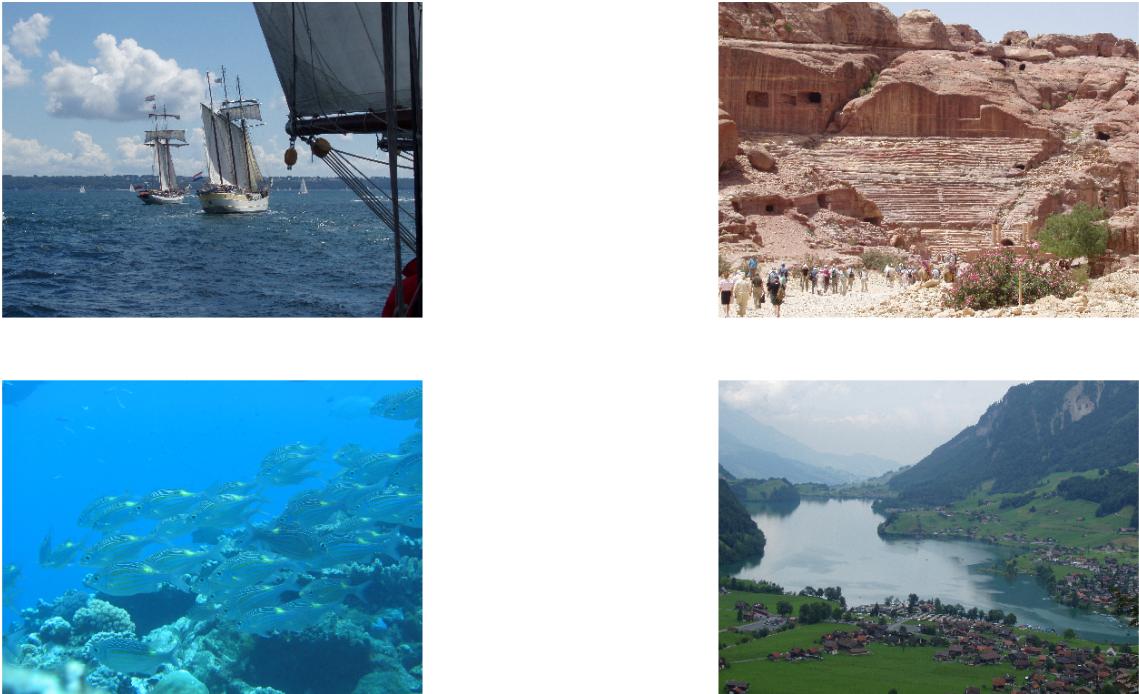
Na klasyczny filtr Sobela-Feldmana składają się dwie maski:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

$G_x$  odpowiada za filtrowanie krawędzi w pionie, a  $G_y$  w poziomie. Obie maski mogą być stosowane oddzielnie. Sam proces filtrowania bazuje na konwolucji opisanej w ramach sieci splotowych w rozdziale 2.1, polegającej na równomiernym przesuwaniu stosowanych filtrów wzdłuż analizowanego obrazu, przy jednoczesnym wykonywaniu zdefiniowanych w nich obliczeń w każdym punkcie. Można w tym miejscu dostrzec spore podobieństwo pomiędzy klasycznymi maskami i ich zastosowaniem, a neuronami wchodzącymi w skład warstw konwolucyjnych sztucznych sieci splotowych. Nie bez powodu neurony te nazywane są filtrami.

W przeprowadzonych doświadczeniach zastosowany został filtr Sobela z maską  $G_x$ . Zbiór uczący wykorzystywany w procesie treningu sieci składał się z różnorodnych obrazów dobieranych w sposób losowy. Przykładowe zdjęcia wchodzące w skład tego zbioru przedstawia Rysunek 4.5.



Rysunek 4.5: Przykładowe obrazy ze zbioru treningowego

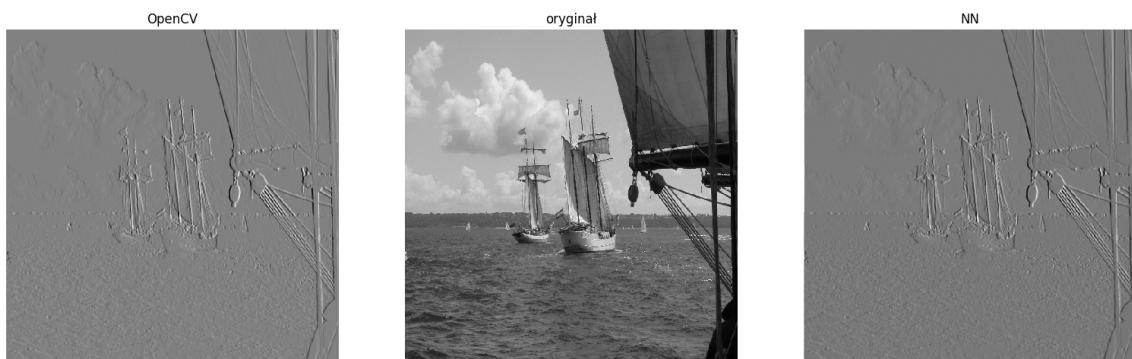
Aby mogły właściwie pełnić swoją rolę obrazy treningowe zostały w pierwszej kolejności podane odpowiedniemu przetworzeniu wstępнемu. W ramach tego procesu rozmiar każdego zdjęcia zmniejszony został do wymiarów  $256 \times 256$  pikseli, a wartości kolorów ograniczone do zakresu  $<0, 1>$  w celu usprawnienia obliczeń wykonywanych przez sieć m.in. poprzez zapobieganie zjawisku eksplodującego gradientu. Dodatkowo w przypadku tego filtru zastosowana została konwersja obrazów do formatu czarno-białego, co pozwoliło wyodrębnić pojedynczy kanał kolorystyczny z oryginałów. Tak przetworzony zbiór uczący podawany był na wejście sieci neuronowej, a rezultaty jej pracy porównywane z obrazami na które dodatkowo nałożony został filtr Sobela za pomocą biblioteki *OpenCV*. W przypadku obrazów referencyjnych po zastosowaniu filtracji konieczne okazało się również przeskalowanie wartości pikseli do przedziału  $<0, 1>$ , ponieważ w sieci zastosowana została funkcja aktywacji *ReLU*, która opisana została w rozdziale 4.3.11. Jej charakterystyka wyklucza pojawianie się wartości ujemnych, jako rezultatów pracy modelu, co w przypadku braku odpowiedniej normalizacji prowadziło do niepoprawnych wyników.

Sam model sieci składa się w tym przypadku z pojedynczego neuronu w warstwie konwolucyjnej, filtrującego obraz za pomocą macierzy kwadratowej stopnia trzeciego. Odzwierciedla to oryginalną macierz filtracji  $G_x$  w stosunku jeden do jednego, ponieważ każda z dziewięciu wag sieci odpowiada jednemu polu w tej macierzy. Takie podejście pozwala jednoznacznie ocenić stopień odwzorowania maski przez sieć poprzez analizę wartości jej parametrów.

W ramach przeprowadzonych eksperymentów, wypróbowane zostały różne konfiguracje hiperparametrów treningowych. Ostatecznie najlepszy rezultat udało się uzyskać przy zastosowaniu następującej konfiguracji:

- Funkcja kosztu: *SmoothL1Loss*
- Optymalizator: *Adam*
- Funkcja aktywacji: *ReLU*
- Ilość epok treningowych: 3
- Rozmiar pakietu danych: 8

Efekt działania wytrenowanego modelu przedstawia Rysunek 4.6.



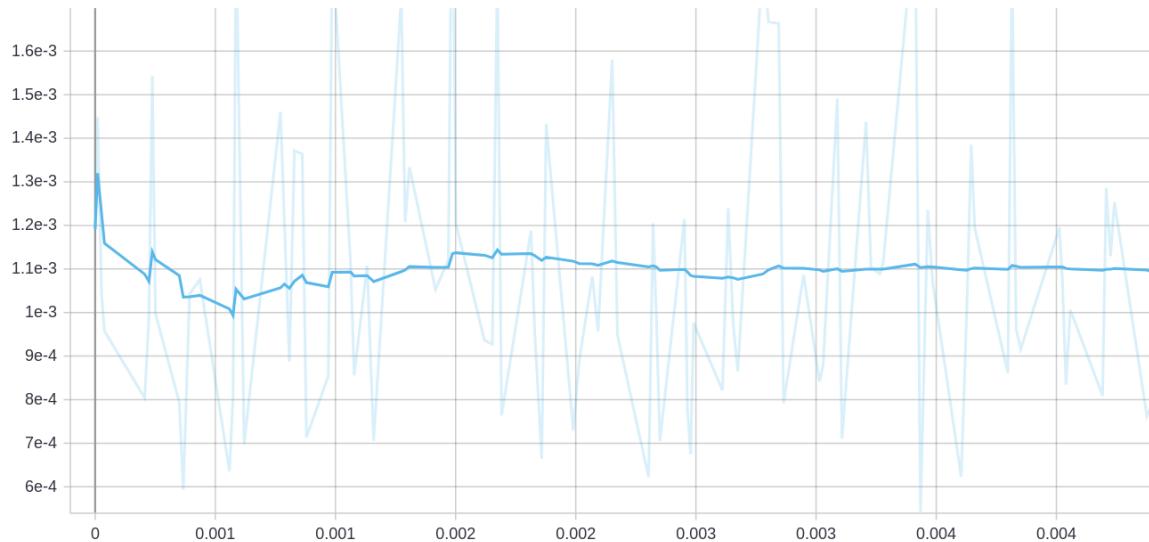
Rysunek 4.6: Działanie filtru Sobela

Zgodnie z opisem oryginalne, czarno-białe zdjęcie poddawane filtracji zamieszczone zostało na środku. Po lewej stronie przedstawiony został obraz przefiltrowany z wykorzystaniem biblioteki *OpenCV*, a po prawej obraz przetworzony przez wytrenowany model sieci neuronowej. Wizualnie otrzymane rezultaty są niemal identyczne. Zdjęcie wygenerowane przez sieć charakteryzuje się nieco ciemniejszą barwą, co może mieć związek z normalizacją danych przeprowadzaną w celu skuteczniejszego uczenia sieci. Poprawne rezultaty treningu najlepiej ocenić można analizując macierz wag modelu, która przedstawia się następująco:

$$G_{nn} = \begin{bmatrix} -0.1135 & -0.0144 & +0.1363 \\ -0.3239 & -0.0016 & +0.3340 \\ -0.1199 & -0.0058 & +0.1335 \end{bmatrix}$$

Porównując uzyskane rezultaty z oryginalną maską  $G_x$  łatwo zauważać można, że sieć neuronowa właściwie odtworzyła panujące w niej proporcje. Odpowiednio mniejszy rzad wielkości odzwierciedla normalizację danych na których uczyony był model. Środkowa kolumna składa się w całości z wartości bliskich零. Kolumna prawa zawiera wartości dodatnie z wyraźną dominacją elementu środkowego. Podobnie rozkładają się wartości w kolumnie lewej zawierającej wyłącznie wartości ujemne.

Wskazówką w określaniu poprawności przeprowadzanych treningów może być również wykres wartości funkcji kosztu w kolejnych krokach uczenia. Przebieg taki przedstawiony został na Rysunku 4.7.



Rysunek 4.7: Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych

W początkowej fazie przedstawiona charakterystyka cechuje się znaczącym spadkiem wartości, po czym utrzymuje się na stosunkowo stałym poziomie. Jest to spodziewany efekt, spowodowany niewielkimi rozmiarami modelu, co przelożyło się na szybkie zlokalizowanie globalnego minimum przez zastosowany optymalizator. Warto wspomnieć, że zastosowanie tej metryki do analizy działania sieci może być zwodnicze. Ciągły spadek wartości funkcji kosztu nie zawsze oznacza wzrost dokładności działania trenowanego modelu. Sieć neuronowa może zacząć w zbyt dużym stopniu dostosowywać się do dostępnych danych treningowych tracąc zdolność do generalizacji rozwiązania dla przykładów spoza tego zbioru. Zjawisko takie nazywane jest przeuczeniem i najczęściej objawia się spadkiem dokładności, przy jednoczesnym opadaniu wartości funkcji kosztu.

Liczne eksperymenty związane z zastosowaniem rozmaitych hiperparametrów wykazały, że pomimo niewielkich rozmiarów modelu zlokalizowanie minimum globalnego nie było zadaniem trywialnym. Prosty optymalizator, taki jak *SGD* nie był w stanie odnaleźć odpowiedniego punktu, a rezultaty jego działania w dużej mierze zależały od losowych wartości przypisanych do wag sieci na początku każdej sesji treningowej. Dopiero zastosowanie algorytmu adaptacyjnego, jakim jest *Adam* pozwoliło uzyskać powtarzalność w osiąganiu właściwych rezultatów. Algorytm *SGD* zastosowany został dopiero w końcowej fazie uczenia z bardzo małym krokiem treningowym rzędu  $\eta = 10^{-8}$ , co pozwoliło nieznacznie poprawić uzyskane wyniki.

Olbrzymi wpływ na rezultaty miały również zastosowane sposoby przetworzenia danych treningowych, między innymi wspomniane już ograniczenie wartości pikseli w przedziale  $<0, 1>$ , tak aby współgrały z zastosowaną funkcją aktywacji.

#### 4.2.2 Sepia

Sepia to filtr obrazu nadający zdjęciom charakterystycznego czerwonawo-brązowego koloru. Jego nazwa pochodzi od rodzaju mątwy, Sepii, z której pozyskiwany był atrament cechujący się takim właśnie zabarwieniem. Choć obecnie materiał ten nie jest już powszechnie wykorzystywany w malarstwie, to sepią wciąż cieszy się sporą popularnością w dziedzinie fotografii i cyfrowego przetwarzania obrazów.

Zastosowanie sepii, jako filtru, wiąże się z konwolucyjnym przetworzeniem zdjęcia za pomocą następującej maski:

$$G_x = \begin{bmatrix} +0.131 & +0.534 & +0.272 \\ +0.168 & +0.686 & +0.349 \\ +0.189 & +0.769 & +0.393 \end{bmatrix}$$

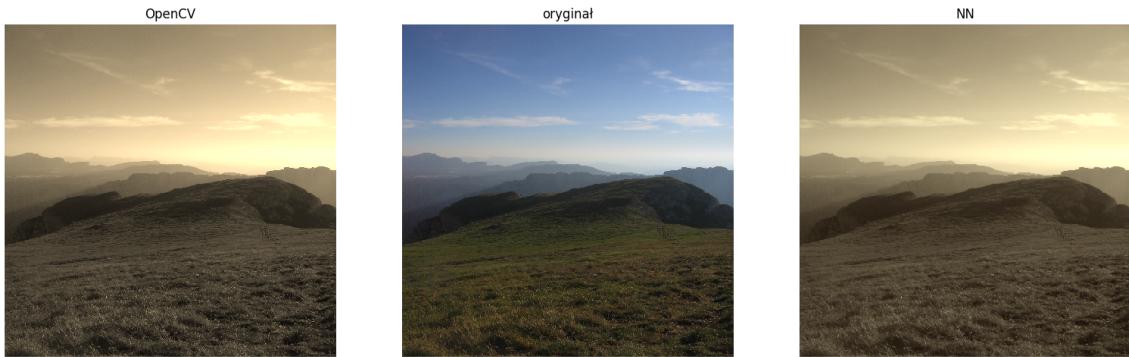
Sieć neuronowa użyta do jej odtworzenia składa się z trzech neuronów w warstwie splotowej. Jest to minimalna struktura niezbędna do odtworzenia trójkanałowego obrazu wyjściowego, jako że każdy filtr odpowiada za wygenerowanie pojedynczej warstwy kolorystycznej.

W procesie uczenia sieci zastosowany został identyczny zbiór treningowy, co w przypadku filtru Sobela, opisanego w poprzednim rozdziale. Ponownie w ramach wstępniego przetwarzania danych, wartości pikseli ograniczone zostały w przedziale  $<0,1>$ , a każdy obraz przeskalowany do wymiarów  $256x256$ . W przypadku obrazów referencyjnych zastosowany został oczywiście filtr zapewniający efekt sepia pochodzący z biblioteki *OpenCV*. Dodatkowo wartości, które w wyniku filtracji przekroczyły wartość 1, zostały ograniczone do jej poziomu. Główną różnicą w stosunku do opisanego uprzednio filtru Sobela jest fakt, że sieć trenowana była na obrazach posiadających pełny zakres trzech kanałów kolorystycznych, a nie jak poprzednio z wykorzystaniem formatu czarno-białego.

Optymalny rezultat działania sieci udało się uzyskać dla następującego zestawu hiperparametrów:

- Funkcja kosztu: *SmoothL1Loss*
- Optymalizator: *Adam*
- Funkcja aktywacji: *ReLU*
- Ilość epok treningowych: 5
- Rozmiar pakietu danych: 8

Większość parametrów nie uległa zmianie w stosunku do filtru Sobela. Świadczy to przed wszystkim o uniwersalności algorytmu *Adam* w odnajdywaniu optymalnych rozwiązań w procesie minimalizacji. Rezultaty pracy wytrenowanego modelu przedstawione zostały na Rysunku 4.8.

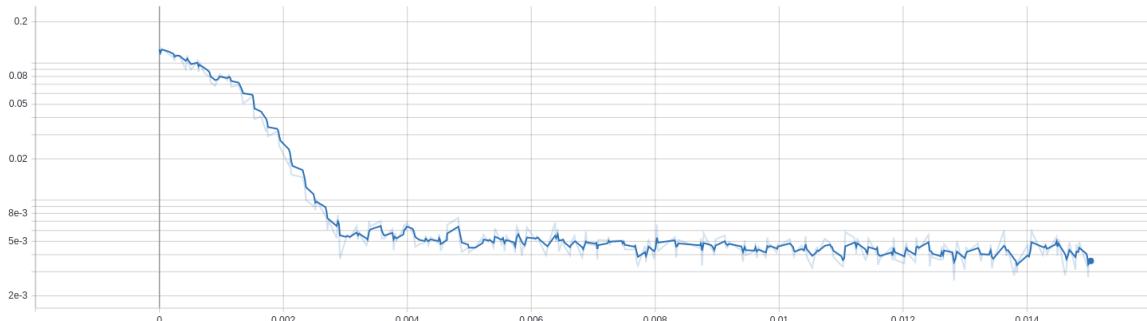


Rysunek 4.8: Działanie sepii

Wizualnie wyniki działania sztucznej sieci neuronowej cechują się nieco bardziej przytłumionymi kolorami w stosunku do obrazów wygenerowanych z wykorzystaniem oryginalnej maski. Powodem takiego zachowania jest najprawdopodobniej dążenie sieci do zminimalizowania wartości funkcji kosztu w obrębie całego zbioru treningowego, co wiąże się z pewnym mimowolnym uśrednieniem wag modelu. Przekłada się ono na trudności w odwzorowaniu obrazów cechujących się dużym kontrastem.

Dodatkowo w rozważanym przypadku rozmiar zastosowanej sieci, choć pozornie niewielki, nie pozwala jednoznacznie przełożyć uzyskanych parametrów modelu na referencyjną maskę  $G_x$ . Każdy z trzech zastosowanych filtrów przetwarza obraz za pomocą macierzy  $3 \times 3$  dla każdej z trzech warstw kolorystycznych. Oznacza to, że pojedynczy neuron powiązany jest z 27 wagami, a w całym modelu jest ich sumarycznie 81. Informacja o sposobie filtracji jest więc mocno rozproszona i ciężko jednoznacznie odczytać postać maski jaką w praktyce implementuje wytrenowany model.

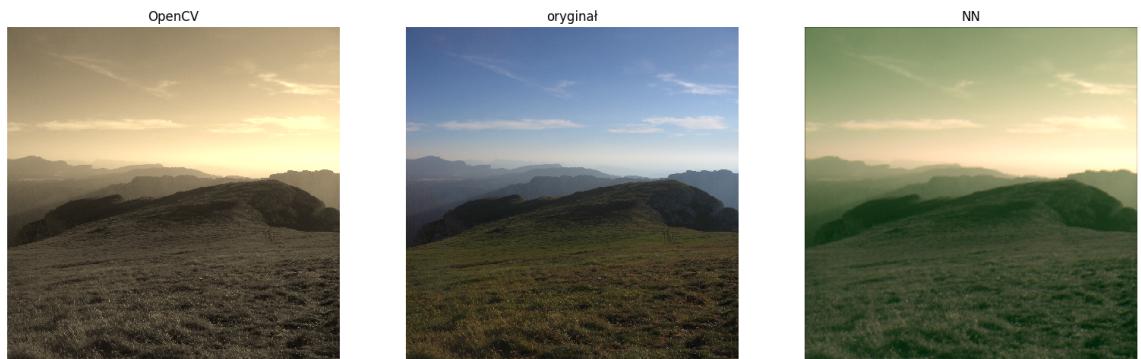
W takiej sytuacji, poza wizualną oceną działania sieci, przydatnych informacji dostarcza wykres wartości funkcji kosztu przedstawiony na Rysunku 4.9.



Rysunek 4.9: Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych

Wyraźne zbocze opadające w początkowej fazie treningu świadczy o poprawnym dążeniu sieci do rozwiązania optymalnego, a następujące po nim wypłaszczenie oznacza, że rozwiązanie to zostało osiągnięte. Porównując Rysunki 4.9 i 4.7 zauważać można, że w przypadku sepii długość zbocza opadającego jest znacznie większa. Może być to spowodowane większą złożonością funkcji kosztu w przypadku tego filtru, lub niekorzystnym początkowym umiejscowieniem algorytmu optymalizacji na płaszczyźnie celu, wynikającym z losowego dobioru startowych parametrów sieci. Najprawdopodobniej oba te czynniki wywarły swój wpływ na przebieg procesu uczenia.

W ramach przeprowadzonych eksperymentów ponownie testowane były rozmaite funkcje optymalizacji, jednak żadna nie pozwoliła osiągnąć takiej powtarzalności w osiąganiu poprawnych rezultatów, jak *Adam*. Nawet zastosowanie innych optymalizatorów adaptacyjnych bardzo często kończyło się na osiągnięciu jedynie jednego z minimów lokalnych. Przykładem takiego rozwiązania może być efekt działania algorytmu *Adagrad* przedstawiony na Rysunku 4.10.



Rysunek 4.10: Przykład działania sieci w minimum lokalnym

Choć na obrazie wygenerowanym przez sieć dostrzec można pewne podobieństwo do poprawnie działającego filtru, to wyraźnie widać że nie jest to zadowalający rezultat, a znalezione na hiperpłaszczyźnie funkcji celu minimum nie jest z pewnością minimum globalnym.

#### 4.2.3 *Filtr górnoprzepustowy*

### **4.3 Automatyczne kolorowanie czarno-białych obrazów**

Problem kolorowania czarno-białych obrazów cieszy się dużym zainteresowaniem z wielu powodów. Od potrzeb kulturowych takich jak możliwość lepszego zwizualizowania oraz zrozumienia przeszłości poprzez kolorowania zdjęć z czasów, kiedy występowały one jedynie w kolorach czerni i bieli, po potrzeby technologiczne takie jak rekonstrukcja filmów oraz poprawa obrazu cyfrowego.

Pomimo braku informacji o kolorze w czarno-białych zdjęciach, ludzie są w stanie określić potencjalne, rzeczywiste barwy obiektów na zdjęciach bazując na treści tych zdjęć oraz swoim doświadczeniu. Można z tego wywnioskować, że zdjęcia te zawierają informacje wystarczające do oszacowania potencjalnych kolorów. Pozwala to założyć, że do tego zagadnienia można skutecznie wykorzystać konwolucyjne sieci neuronowe, które cechują się niezwykłą umiejętnością do rozpoznawania wzorców oraz posiadają wyjątkowe zdolności do adaptacji. Z tego właśnie powodu sieci splotowe zostaną użyte w przedstawionym rozwiązaniu.

#### *4.3.1 Podejście*

Rozważając możliwe sposoby pokolorowania czarno-białego zdjęcia można spostrzec, że kiedy niektóre powierzchnie na zdjęciu mają zazwyczaj oczywiste barwy, niebo jest zazwyczaj niebieskie, a trawa zielona, to są też powierzchnie, które posiadają szeroki wachlarz możliwych kolorów, na przykład samochodów może być zarówno czerwony jak i niebieski albo zielony. Z tego powodu celem zaprezentowanego rozwiązania jest niekoniecznie odtworzenie rzeczywistych barw obrazu, a raczej wygenerowanie barw, które mogłyby być barwami rzeczywistymi.

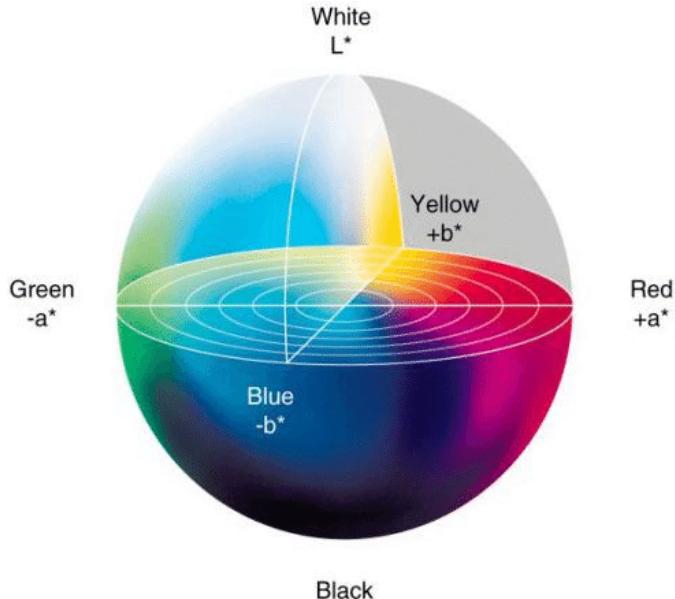
Aby zwiększyć efektywność uczenia wykorzystano przestrzeń barw CIELab. W tej przestrzeni barwę obrazu opisują 3 składowe:

- L -jasność (luminacja)
- A -barwa od zielonej do magenty
- B - barwa od niebieskiej do żółtej

Przestrzeń barw CIELab została przedstawiona na Rysunku 4.11

Zaletą zastosowania CIELab jest fakt, że jest ona najbardziej równomierną przestrzenią barw, co oznacza, że jeśli barwy znajdują się w jednakowej odległości od siebie w tej przestrzeni, to będą one postrzegane jako jednakowo różniące się od siebie. Powinno to zwiększyć skuteczność uczenia sieci oraz zapewnić bardziej realistyczne kolorowanie.

Składowa  $L$ , jako, że jest identyczna dla obrazu kolorowego jak i czarno-białego, stanowi w tym przypadku wejście sieci, na jej podstawie sieć odtwarza składowe  $A$  oraz  $B$ , które reprezentują przewidziane kolory dla obrazu wejściowego. W celu lepszego zrozumienia formatu przykładowa składowa  $L$  została przedstawiona na Rysunku 4.12. Jak widać zawiera ona informacje wystarczające do rozróżnienia między sobą powierzchni oraz wydobycia ich kluczowych cech.



Rysunek 4.11: Przestrzeń barw CIELab.

Jako rozwiązanie podanej problematyki wpierw został oceniony autorski model, na jego podstawie zostało przeprowadzone porównanie skuteczności różnych konfiguracji, w których uczyony był model. Do elementów poddanych testom należą algorytm optymalizacyjny, funkcja straty, funkcja aktywacją oraz sposób przetwarzania wstępne danych treningowych.

#### 4.3.2 Model końcowy

Opracowany przez nas model jest to FCN. Konwolucyjna część sieci składa się z 12 warstw splotowych mających na celu nauczyć się mapować składową wejściową  $L$  na wyjściowe składowe  $A$  i  $B$ . Składowe wyjściowe muszą mieć takie same wymiary jak składowa wejściowa, co oznacza, że kluczowym było odpowiednie dobranie parametrów warstw takich jak *padding* (pol. otoczka), *stride* (pol. krok) oraz wielkość filtrów. Pełna architektura sieci została przedstawiona w Tabeli 3. Pierwsza warstwa konwolucyjna rozkłada wejściowy kanał na 32 kanały, co pozwala wyciągnąć z niego jak najwięcej informacji o cechach obrazu. Warstwa ta ma wielkość filtra  $3 \times 3$ , tak więc aby zachować wymiar kanałów zostały zastosowane parametry *padding* = 1 oraz *stride* = 1.

Kolejne trzy warstwy ekstraktują z wejściowych 32 kanałów najbardziej istotne cechy związane z powiązaniem treści obrazu z szacowanym kolorem jego powierzchni. Warstwy te na swoje wyjście przekazują po 32 kanały zawierające wykryte powiązanie pomiędzy pikselami kanałów wejściowych.

Warstwa piąta rozciąga wejściowe 32 kanały na 64 kanały, dzięki temu kolejne 2 warstwy przyjmujące na wejście te 64 kanały i przekazujące je na wyjście są w stanie wydobyć z obrazu cechy o większym poziomie abstrakcji, co znacznie zwiększa skuteczność działania sieci.



Rysunek 4.12: Przykładowa składowa  $L$ .

Nr	Warstwa	Rozmiar filtra	Stride	Padding	Batch Normalization	Fun. aktywacji	Ilość kanałów wej./wyj.
1	Splotowa	3x3	1	1	Tak	RELU	1/32
2	Splotowa	3x3	1	1	Tak	RELU	32/32
3	Splotowa	3x3	1	1	Tak	RELU	32/32
4	Splotowa	3x3	1	1	Tak	RELU	32/32
5	Splotowa	3x3	1	1	Tak	RELU	32/64
6	Splotowa	3x3	1	1	Tak	RELU	64/64
7	Splotowa	3x3	1	1	Tak	RELU	64/64
8	Splotowa	3x3	1	1	Tak	RELU	64/32
9	Splotowa	3x3	1	1	Tak	RELU	32/32
10	Splotowa	1x1	1	0	Tak	RELU	32/32
11	Splotowa	1x1	1	0	Tak	RELU	32/32
12	Splotowa	1x1	1	0	Nie	-	32/2

Tabela 3: Architektura modelu podstawowego.

Warstwa ósma ogranicza ilość kanałów w sieci z 64 do 32 wyciągając z nich cechy najbardziej przydatne do rozwiązania danej problematyki. Kanały te są następnie ponownie przetwarzane przez warwę z wielkością filtru 3x3, co ma służyć agregacji rozłożonych cech w bardziej spójną całość, która może być już składana w pożądane wyjście.

Kolejne dwie warstwy w sieci są to warstwy konwolucyjne o wielkości filtru 1x1, odpowiadające one warstwom gęstym i mają na celu przekonwertowanie wartości funkcji aktywacji z poprzednich warstw na wartości kolorów odpowiadających pikseli w przestrzeni barw CIELab. Ostatnia warstwa, również z filtrem o wielkości 1x1 zwija 32 kanały otrzymywane na wejściu do 2 kanałów odpowiadających składowym  $A$  oraz  $B$ , które stanowią pożądany rezultat działania sieci.

Po wszystkich, oprócz ostatniej, warstwach konwolucyjnych znajdują się dodatkowo warstwa BatchNorm oraz warstwa funkcji aktywacji RELU mające na celu ustabilizować proces uczenia oraz zwiększyć jego efektywność.

#### 4.3.3 BatchNorm

Warstwa Batch Normalization została przedstawiona w 2015 roku przez S. Ioffe oraz C. Szegedy jako odpowiedź na problem zmieniającej się podczas uczenia dystrybucji wartości wejść każdej z warstw sieci [10]. Ma ona na celu usprawnić i ustabilizować trening sieci poprzez normalizację wartości podawanych na funkcje aktywacji. Zmienna dystrybucja tych wartości znacznie spowalnia i utrudnia proces uczenia poprzez potrzebę przemyślanego inicjowania wag sieci w celu zwiększenia prawdopodobieństwa nakierowania modelu na pożądane rozwiązania w trakcie procesu uczenia oraz przez konieczność używania mniejszych wartości współczynnika uczenia, aby przeciwdziałać problemom zanikającego oraz wybuchającego gradientu [11].

Problemy te zostały już zauważone i opisane w 1994 roku przez Y. Bengio oraz jego współpracowników [11]. Dowodzą oni, że:

*'Metoda gradientu prostego staje się coraz bardziej nieefektywna, gdy rośnie czasowy zakres zależności'*

Wskazują także, że problemy powstają podczas treningu DNN w fazie wstępnej propagacji błędu, kiedy to gradient pochodzący z głębszych warstw przechodzi wielokrotnie przez operacje mnożenia macierzowego. Jeśli wartość gradientu jest niewielka, to z każdą operacją mnożenia staje się jeszcze mniejsza, aż maleje do takich wartości, które nie umożliwiają modelowi uczenia się, a jeśli wartość ta jest wysoka to, wraz z przechodzeniem przez kolejne warstwy, rośnie jeszcze bardziej co przy bardzo dużych wartościach może doprowadzić do destabilizacji procesu uczenia. Są to zjawiska zdecydowanie niepożądane i z tego powodu powstało wiele rozwiązań, aby im przeciwdziałać takich jak ograniczanie maksymalnej wartości gradientu (ang. gradient clipping) albo zastosowanie warstw BatchNorm.

Zastosowanie warstw BatchNorm sprawia, że podczas uczenia metodą mini-batch (pol. małych paczek) każda paczka jest normalizowana w sposób zapewniający zerową wartość średnią oraz równą jedności wariancję na przestrzeni wszystkich kanałów wejściowych. Zaletą takiego podejścia jest poprawienie przepływu korygującego gradientu przez kolejne warstwy sieci podczas fazy wstępnej propagacji błędu. Ponadto warstwy BatchNorm zapewniają większą odporność sieci na niekorzystnie zainicjowane wagi początkowe modelu.

Użycie tych warstw w modelu podstawowym tuż za warstwami RELU pozwoliło uzyskać bardziej korzystną zbieżność modelu oraz lepsze rezultaty końcowe. Ocenione zostało też rozwiązania, w którym warstwy BatchNorm znajdują się przed warstwami funkcji aktywacji, lecz dało ono gorsze rezultaty, niż podejście wspomniane jako pierwsze.

#### 4.3.4 Dropout

W trakcie pracy nad ostatecznym modelem podstawowym sprawdzona została skuteczność zastosowania warstw Dropout [12]. Warstwy te w trakcie treningu dezaktywują część neuronów aby przeciwdziałać efektowi przeuczania się sieci oraz zapewniać wydajny sposób łączenia wielu różnych architektur sieci stworzonych do jednego celu w jednolitą całość o skuteczności większej niż poszczególne sieci osobno.

Wybór neuronów, dla których w danej iteracji treningu nie zostaną zaktualizowane wagie odbywa się z pewnym prawdopodobieństwem określonym podczas inicjalizacji modelu. Dezaktywowanie neuronów można interpretować jak przerywanie tymczasowo wszystkich połączeń danego neuronu, zarówno wejściowych jak i wyjściowych. Takie działania są równoznaczne z wyselekcjonowaniem z modelu mniejszej sieci i trenowaniu wyłącznie jej w aktualnej iteracji. Wagi tej podsieci są wtedy współdzielone z modelem źródłowym. Jako rezultat uzyskuje się model o znacznie ulepszonych zdolnościach generalizacji.

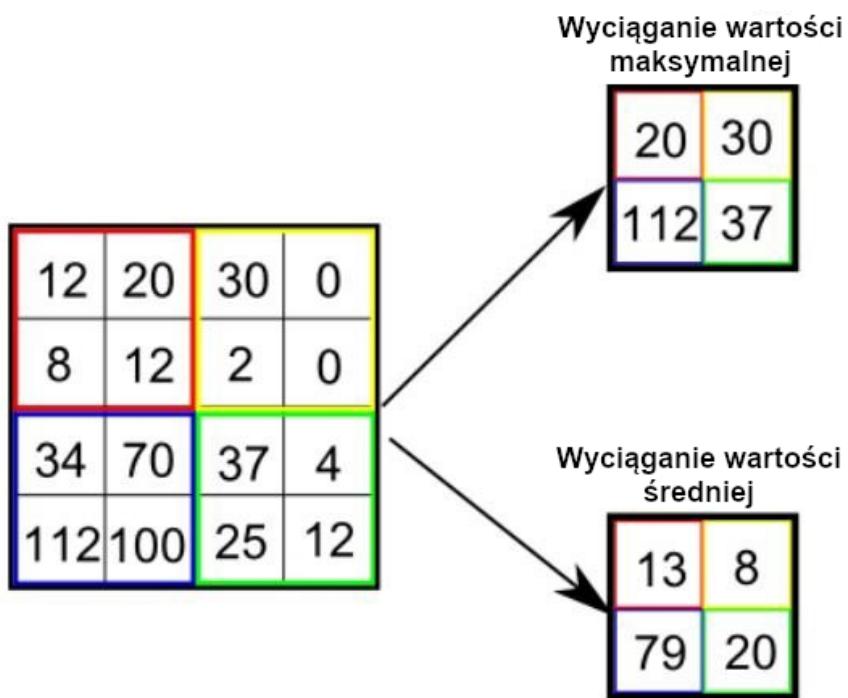
Zastosowanie tych warstw w modelu nie przyniosło wyraźnego polepszenia rezultatów sieci. W przypadku danej problematyki oraz obranego podejścia do jej rozwiązania, przeuczenie sieci nie stanowi wyraźnego zagrożenia, a zastosowanie Dropout wiąże się z utratą części informacji kluczowych do odpowiedniego generowania kolorów dla wejściowych obrazów. Model powinien nauczyć się jak największej różnorodności kolorów, a Dropout przeciwdziała uczeniu się nadmiernej ilości cech przez sieć, co wpływa niekorzystnie na otrzymywane rezultaty końcowe. Podczas testów skuteczności tych warstw zostały one umieszczone za warstwami BatchNorm. Wizualizacja skutków tej decyzji wraz z rozważeniem pozostałych czynników wpływających na efektywność sieci znajdują się w punkcie 4.3.14 związanym z rezultatami modelu podstawowego.

#### 4.3.5 Modyfikacja rozdzielczości

W modelach FCN powszechnie stosuje się różne metody zmiany rozdzielczości kanałów przechodzących przez sieć, najczęściej są to operacje poolingu mające na celu zredukować przestrzenną wielkość reprezentacji cech wyciągniętych z obrazu poprzez wyciągnięcie najbardziej istotnych wartości funkcji aktywacji z określonych obszarów reprezentacji w celu zredukowania rozmiaru sieci, a co za tym idzie, zmniejszenia ilości obliczeń koniecznych do wykonania przez sieć.

Ponadto pooling wspomaga adaptację modelu do zmennego położenia kluczowych wzorów rozpoznawanych przez sieć na obrazie wejściowym. Cechą ta zwana jest niezmiennością od translacji (ang. transaltion invariance). Jest ona rezultatem dokonywania operacji, takich jak wyliczanie wartości maksymalnej z poszczególnych obszarów, dzięki którym zmienne położenie wartości selekcyjowej, a co tym idzie, zmienne położenie ekstraktowanej cechy w obrębie danego obszaru nie wpływa na końcową postać reprezentacji przestrzennej obrazu wejściowego po przejściu przez warstwę poolingu. Przykładowe operacja poolingu została przedstawiona na Rysunku 4.13.

W modelu końcowym warstwy poolingu nie zostały zastosowane, aby uniknąć utraty kluczowych informacji przestrzennych koniecznych do właściwego wygenerowanie możliwych barw obrazu.



Rysunek 4.13: Przykładowe operacje warstwy poolingu

#### 4.3.6 Wykorzystywany zbiór treningowy

Do uczenia modelu został wykorzystany zbiór danych CIFAR-10 stworzony przez A. Krizhevsky oraz zaprezentowany w 2009 roku [13]. Składa się on z 60000 obrazów w przestrzeni kolorów RGB o rozdzielczości  $32 \times 32$  piksele. Spośród tych obrazów 10000 z nich zostało wykorzystanych jako zbiór walidacyjny do śledzenia skuteczności treningu modelu. Przykładowe obrazy ze zbioru danych zostały przedstawione na Rysunku 4.14.



Rysunek 4.14: Przykładowe obrazy z CIFAR-10

Zaletą CIFAR-10 jest niewielka rozdzielcość obrazów co pozwala na mniej złożony model oraz szybszy proces uczenia, który można skutecznie przeprowadzić nawet przy ograniczonych możliwościach obliczeniowych. Ponadto zbiór ten składa się z obrazów dzielących się na 10 klas, tak mała różnorodność klas, a co za tym idzie, stosunkowo niewielka ilość możliwych obiektów pojawiających się na zdjęciach powinna ułatwić sieci wyuczenie się właściwych barw dla identyfikowanych powierzchni. Z drugiej strony mała ilość klas może niekorzystnie wpływać na umiejętności generalizacji modelu, jednakże obrazy z CIFAR-10 przedstawiają zróżnicowane otoczenia zawierające obiekty nie kwalifikujące się do żadnej klasy, co powinno umożliwić wyuczenia się generalizacji przez sieć.

#### 4.3.7 Przetwarzanie wstępne danych

Przed podaniem na wejście sieci obrazy uczące były wpierw poddawane przetwarzaniu wstępemu mającemu na celu doprowadzić do szybszego oraz stabilniejszego treningu modelu. Przetwarzanie wstępne jest kluczowe, gdyż wartości o jakie aktualizowane są wagie neuronu zależą w dużej mierze od wartości wejść tego neuronu. W przypadku gdy przedziały wartości tych wejść nie są jednolite, to może wystąpić duża różnica w tempie aktualizacji wag sieci, niektóre wagie będą zmieniane o wiele szybciej niż inne co może spowodować destabilizację treningu. Przeskalowanie wszystkich wartości wejściowych do jednakowych przedziałów, o niewielkiej wartości maksymalnej i minimalnej oraz wartości średniej zbliżonej do zera zmniejsza możliwość wystąpienia tego problemu oraz sprzyja ujednoliceniu tempa uczenia się przez sieć rozpoznawania różnych cech. Ponadto brak przeskalowania wejść może doprowadzić do zjawisk wybuchającego oraz zanikającego gradientu.

W ramach badania rozwiązania przetestowane zostały różne metody przetwarzania wstępnego zarówno danych wejściowych jak i pożądanej odpowiedzi:

- Normalizacja danych na przestrzeni całego zbioru danych z użyciem wartości maksymalnej oraz minimalnej całego zbioru, tak aby wartości pikseli danej składowej dla każdego obrazu zawierały się w przedziale od -0.5 do 0.5.
- Standaryzacja danych na przestrzeni całego zbioru danych z użyciem wartości średniej oraz odchylenia standardowego całego zbioru, tak aby uzyskać wartość średnią równą w przybliżeniu zero oraz jednostkowe odchylenie standardowe.
- Zastosowanie rozmycia gaussowskiego(ang. Gaussian blur) o różnej wielkości filtra Gaussowskiego.

Rozmycie gaussowskie, zwane także wygładzaniem gaussowskim (ang. Gaussian smoothing), jest to operacja polegająca na modyfikacji obrazu z użyciem filtru Gaussa. Stosuje się je w celu rozmycia detali na przetwarzanym obrazie, a także by ograniczyć ilość występujących na nim zakłóceń oraz szumów. Jest ono powszechnie stosowane w fazie przetwarzania wstępnego danych graficznych. W praktyce operacja ta sprowadza się do dokonywania splotu kolejnych fragmentów obrazu z funkcją Gaussa. Zastosowanie jej na danych wejściowych miało na celu zmniejszenie znaczenia detali na obrazie oraz ułatwienie sieci nauczenia się rozróżniania rozmaitych powierzchni oraz wzorców.

Wymienione metody przetwarzania były stosowane w różnych połączeniach oraz konfiguracjach zarówno na składowej  $L$ , jak i składowych  $A$  oraz  $B$ , a uzyskane wyniki opisane zostały w punkcie 4.3.14

#### 4.3.8 Przetwarzanie końcowe danych

Jeśli w trakcie procesu uczenia były stosowane zabiegi przetwarzania wstępne danych wejściowych to podczas testowania modelu dane testowe również muszą być przetworzone w ten sam sposób, aby zapewnić poprawną pracę sieci. To samo dotyczy się danych wyjściowych sieci, jeśli podczas treningu z nadzorem pożądane wyjście było w pewien sposób przetworzone wstępnie to sieć uczy się odtwarzać te wyjście tak samo przetworzone, aby uzyskać oczekiwany efekt końcowy należy dokonać operacji odwrotnych do tych zastosowanych w fazie przetwarzania wstępnego. Przykładowo jeśli pożądana odpowiedź w procesie uczenia była normalizowana to podczas testów sieci jej wyście należy zdenormalizować, aby uzyskać oczekiwany rezultat. Jednakże my dla rozważanej problematyki proponujemy metodę alternatywną.

Polega ona na uwydatnianiu kolorów wygenerowanych przez sieć dla wysokich wartości natświetlenia - składowej  $L$ . Może być ona stosowana niezależnie od metody przetwarzania wstępnego danych wejściowych. Algorytm polega na przeskalowaniu pikseli składowych  $A$  i  $B$  tak, aby ich wartości mogły pokryć cały dostępny przedział wartości, ale dla danego piksela jego przedział jest ograniczony proporcjonalnie do stosunku wartości tego piksela w składowej  $L$  do maksymalnej możliwej wartości pikseli składowej  $L$ .

Dla przykładu założymy, że wartość danego piksela dla składowej  $A$  wynosi 70, dla składowej  $B$  wynosi -20, a dla składowej  $L$  80. Przedział wartości składowych  $A$  i  $B$  jest od -127 do 128, a składowej  $L$  od 0 do 100. W kolejnym kroku znajdowana jest maksymalna absolutna wartość składowych  $A$  i  $B$  dla aktualnego obrazu. Założymy, że dla  $A$  wynosi ona 90, a dla  $B$  60. Następnie piksele składowych  $A$  i  $B$  są dzielone przez swoje maksymalne wartości, a następnie rozciągnięte na cały dostępny przedział przez pomnożenie przez odpowiedni czynnik, czynnik ten jest z przedziału od 0 do 127 i jest wprost proporcjonalny do stosunku aktualnego piksela składowej  $L$  do maksymalnej wartości  $L$ , oznacza to, że jeśli dany piksel  $L$  jest równy 100 to czynnik jest równy 127, a jeśli dany piksel  $L$  jest równy 50 to wartość czynnika znajduje się w połowie swojego przedziału i równy jest 63,5.

Dla podanych założeń otrzymujemy następujące nowe wartości piksela składowej  $A$  ( $p_n^A$ ) i piksela składowej  $B$  ( $p_n^B$ ):

$$p_n^A = \frac{70}{90} * 127 * \frac{80}{100} = 79.02 \quad (1)$$

$$p_n^B = \frac{-20}{60} * 127 * \frac{80}{100} = -33.87 \quad (2)$$

Algorytm konwertowanie pikseli danej składowej można przedstawić wzorem:

$$S_{i,j}^n = \frac{S_{i,j}^p}{\max\{|S|\}} * 127 * \frac{L_{i,j}}{100} \quad (3)$$

Gdzie:

- $S$  - Konwertowana składowa będąca dwuwymiarowa macierzą pikseli.

- $S_{i,j}^n$  - Nowa wartość piksela (i, j) dla danej składowej.
- $S_{i,j}^p$  - Stara wartość piksela (i, j) dla danej składowej.
- $L_{i,j}$  - Wartość piksela (i, j) składowej jasności.

Zastosowanie powyższego algorytmu pozwoliło uzyskać bardziej zadawalające rezultaty działania modelu poprzez faworyzowanie kolorów tworzonych przez sieć dla powierzchni o dużej wartości jasności, jako, że wysoka jasność oferuje bardziej intensywne barwy. Kolorystyka powierzchni ciemnych pozostaje przytłumiona, jako, że brak naświetlenia ogranicza natężenie kolorów.

#### 4.3.9 Augmentacja danych

W celu skutecznego treningu modelu potrzebny jest odpowiednio duży i różnorodny zbiór treningowy. W obliczu problemu niewystarczającej ilości danych stosuje się metody zwane augmentacją danych pozwalającą poszerzyć zbiór obrazów uczących poprzez dodawanie nowych informacji do obrazów będących podstawą zbioru tworząc w ten sposób nowe obrazy, które mogą być wykorzystane w treningu. Augmentacja danych przeciwdziała uczeniu się przez sieć nieistotnych wzorów i cech takich jak orientacja obiektu, jego umiejscowienie albo rozmiar. Dzięki temu model jest w stanie dogłębniej analizować obrazy wejściowe ucząc się rozpoznawać cechy o coraz to większym poziomie abstrakcji co przekłada się na o wiele lepiej rozwiniętą zdolność modelu do generalizacji danej problematyki.

Do augmentacji stosuje się proste przekształcenia obrazu takie jak:

- Rotacja obrazu o wybrany kąt.
- Odbicie obrazu względem osi pionowej.
- Obcinanie skrajnych fragmentów obrazów (ang. crop).
- Zmiana odcienia oraz saturacji barw obrazu.
- Przybliżanie albo oddalenie treści obrazu.
- Modyfikacja rozdzielczości obrazu poprzez jego rozciąganie lub ściskanie.
- Tworzenie niewielkich ubytków w obrazach w losowych miejscach (ang. coarse dropout).

Należy jednak pamiętać, że nie wszystkie metody augmentacji nadają się do każdego zbioru danych, kluczowe jest, aby wybrać takie operacje, które poszerzają zbiór uczący o dane niosące znaczące oraz sensowne informacje patrząc z punktu wybranej problematyki oraz nie przesłaniają wzorców kluczowych do wyuczenia przez sieć. W przypadku zagadnienia kolorowania czarno-białych obrazów kluczową informacją jest kolor analizowanej powierzchni oraz jej cechy charakterystyczne, z tego powodu, do treningu modelu podstawowego nie zostały wykorzystane żadne metody augmentacji wpływające na barwy albo jasność obrazów. Wykluczone zostały również takie metody jak tworzeniu ubytków w obrazach, gdyż powoduje to niepotrzebną utratę informacji. W związku z tym, że wykorzystany zbiór CIFAR-10 posiada dużą ilość obrazów to w procesie uczenia została wykorzystana jedynie augmentacja poprzez odbicie obrazu względem osi pionowej z prawdopodobieństwem równym 50%.

#### 4.3.10 Funkcje kosztów

Kluczem do właściwego funkcjonowania modelu jest wybór odpowiedniej funkcji kosztu. W przypadku problematyki kolorowania czarno-białych obrazów ważne jest, aby wybrana funkcja kosztu uwzględniała specyficzną naturę problemu, gdzie dla niektórych przypadków właściwych jest wiele odpowiedzi. Jako przykład można podać taki obiekt jak samochód, który może być zarówno zielony, czerwony jak i żółty, każdy z tych kolorów powinien być oceniony jako właściwy, a wartość błędu wyliczona z użyciem funkcji kosztu dla tak wybranych przez sieć kolorów powinna odpowiednio to wskazywać. W ramach poszukiwań najbardziej odpowiedniej funkcji kosztu przetestowane zostały następujące funkcje.

1. MSELoss (ang. Mean Squared Error Loss) - koszt oparty na błędzie średniokwadratowym przedstawiony funkcją:

$$Koszt = \frac{1}{n} \sum_{i=0}^n (x_i - y_i)^2 \quad (4)$$

Po dopasowaniu równania MSELoss do rozważanej problematyki otrzyma się:

$$Koszt = \frac{1}{n} \frac{1}{m} \sum_{i=0}^n \sum_{j=0}^m ((A_{i,j}^P - A_{i,j}^R)^2 + (B_{i,j}^P - B_{i,j}^R))^2 \quad (5)$$

2. L1Loss zwany także MAELoss (ang. Mean Absolute Error Loss) - koszt oparty na średnim błędzie bezwzględnym danej funkcją:

$$Koszt = \frac{1}{n} \sum_{i=0}^n |x_i - y_i| \quad (6)$$

Równanie L1Loss dla rozważanego zagadnienia:

$$Koszt = \frac{1}{n} \frac{1}{m} \sum_{i=0}^n \sum_{j=0}^m (|A_{i,j}^P - A_{i,j}^R| + |B_{i,j}^P - B_{i,j}^R|) \quad (7)$$

3. SmoothL1Loss - zwany także *Huber loss*, odmiana L1Loss przedstawiona w 2015 roku przez R. Girshick [14]. Jej zaletami są mniejsza czułość na elementy odstające (ang. outlier) i zmniejszona szansa na wystąpienie zjawiska eksplodującego gradientu. SmoothL1Loss przedstawiony jest funkcją:

$$Koszt = \frac{1}{n} \sum_{i=0}^n z_i \quad (8)$$

gdzie  $z_i$  dane jest:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{jeśli } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{w pozostałych przypadkach} \end{cases} \quad (9)$$

Zastosowanie SmoothL1Loss do modelu podstawowego da następującą funkcję:

$$Koszt = \frac{1}{n} \sum_{i=0}^n (z_i + k_i) \quad (10)$$

gdzie  $z_i$  dane jest:

$$z_i = \begin{cases} 0.5(A_{i,j}^P - A_{i,j}^R)^2, \text{ jeśli } |A_{i,j}^P - A_{i,j}^R| < 1 \\ |A_{i,j}^P - A_{i,j}^R| - 0.5, \text{ w pozostałych przypadkach} \end{cases} \quad (11)$$

a  $k_i$  dane jest:

$$z_i = \begin{cases} 0.5(B_{i,j}^P - B_{i,j}^R)^2, \text{ jeśli } |B_{i,j}^P - B_{i,j}^R| < 1 \\ |B_{i,j}^P - B_{i,j}^R| - 0.5, \text{ w pozostałych przypadkach} \end{cases} \quad (12)$$

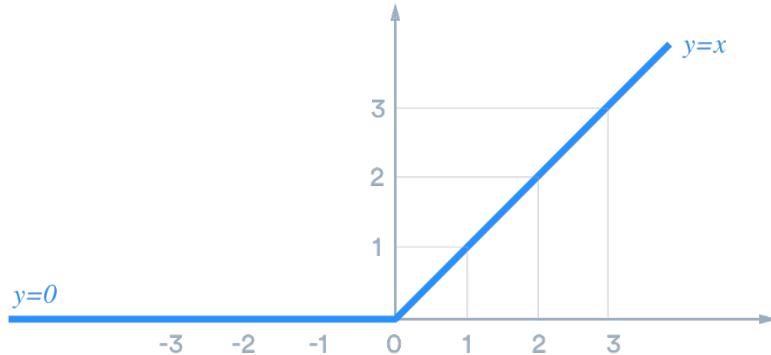
Gdzie:

- $x$  są to składowe  $A$  oraz  $B$  przewidziane przez sieć.
- $y$  to rzeczywiste składowe  $A$  i  $B$ .
- $A_{i,j}^R$  i  $B_{i,j}^R$  są to rzeczywiste wartości składowych  $A$  i  $B$  dla pikselu obrazu o współrzędnych  $(i, j)$ .
- $A_{i,j}^P$  i  $B_{i,j}^P$  są to wartości składowych  $A$  i  $B$  przewidziane przez sieć dla pikselu obrazu o współrzędnych  $(i, j)$ .

Szczegółowy opis skutków zastosowanie poszczególnych funkcji kosztów umieszczony został w punkcie 4.3.14.

#### 4.3.11 Funkcje aktywacji

W wyborze odpowiedniej funkcji aktywacji kierowaliśmy się badaniami przeprowadzonymi przez B. Xu oraz jego współpracowników [15]. Testowali oni skuteczność takich funkcji jak ReLU, Leaky ReLU (pol. przepuszczające ReLU), PReLU (ang. Parametric ReLU) oraz RReLU (ang. Randomized Leaky ReLU) w zagadnieniu klasyfikacji treści obrazów. Bazując na otrzymanych przez nich rezultatach zdecydowaliśmy się wykorzystać w rozważanej problematyce funkcję aktywacji ReLU, przedstawioną na Rysunku 4.15



Rysunek 4.15: Funkcja aktywacji ReLU

Funkcja ta posiada wiele zalet takich jak przyspieszenie procesu zbieganie się stanu sieci do stanu pożdanego poprzez brak ograniczeń na maksymalną wartość funkcji oraz niska złożoność obliczeniowa. Ponadto funkcja ta, w związku z zerową wartością dla ujemnych argumentów, zapewnia aktywację neuronów modelu tylko wtedy, gdy analizują one wzorce kluczowe dla nich samych oraz danego zagadnienia, zwiększa to odporność modelu na szумy wejściowe oraz przeuczanie, a także poprawia zdolności predykcyjne sieci.

Jednakże stosowanie ReLU tworzy zagrożenie blokowanie się procesu uczenia, jeśli dojdzie do sytuacji, w której wagi modelu dojdą do stanu, gdzie wartość aktywacji będzie zbliżona do zera. Spowoduje to zerową wartość gradientu podczas fazy wstępnej propagacji błędu powodując wstrzymanie się procesu aktualizowania wag modelu, a w rezultacie, nieefektywny proces uczenia. Pomimo to jednak zostaliśmy przy wyborze funkcji ReLU wierząc, że pozostałe zabiegi takie jak zastosowanie warstw BatchNorm pozwolą zminimalizować niepożądane efekty tego zjawiska.

#### 4.3.12 Algorytmy optymalizacyjne

Podczas planowania treningu modelu należy zdecydować się na odpowiedni algorytm optymalizacyjny, właściwa decyzja pozwala uniknąć zatrzymania się procesu uczenia w lokalnych minimach co zwiększa końcową dokładność oraz skuteczność modelu. Podczas badań przetestowane zostały różne algorytmy optymalizacyjne, a uzyskane rezultaty zostały szczegółowo opisane oraz porównane w punkcie 4.3.14.

Wykorzystane zostały następujące algorytmy:

- Adam (ang. Adaptive Moment Estimation)
- Adagrad (ang. Adaptive Gradient Algorithm)
- SGD (ang. Stochastic Gradient Descent)

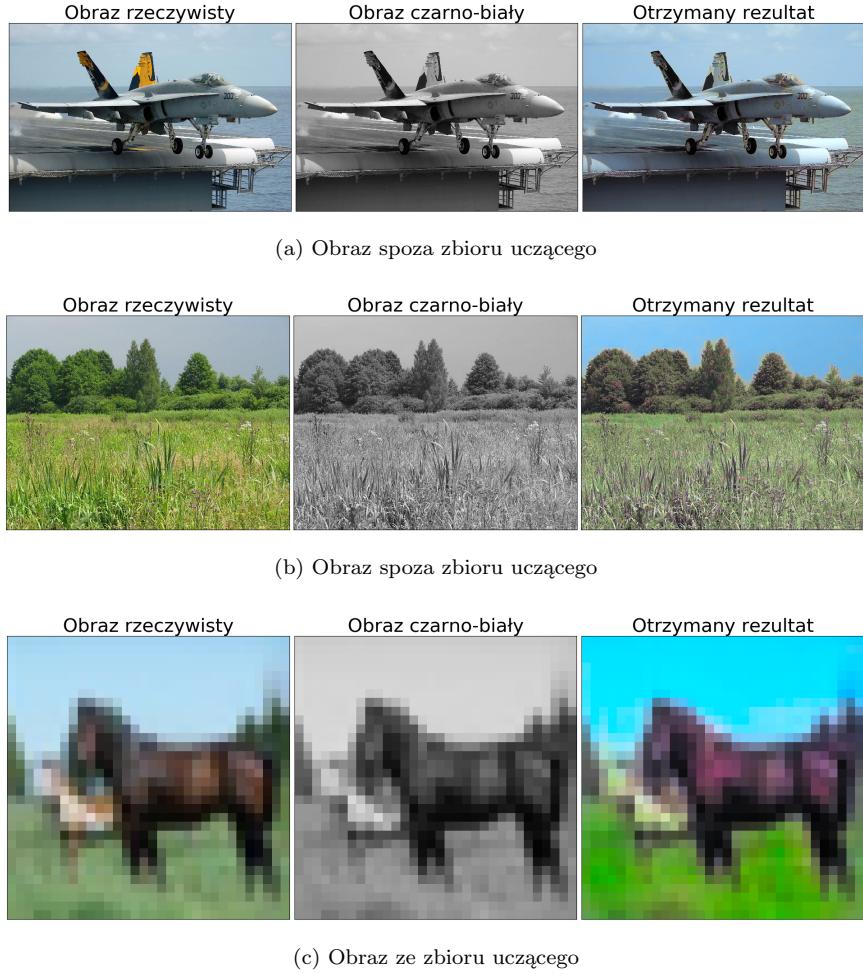
Przeprowadzone zostały także poszukiwania najbardziej odpowiednich hiperparametrów dla wymienionych algorytmów w celu osiągnięcia jak największej ich skuteczności w rozwiązaniu rozważanej problematyki.

#### 4.3.13 Trening

Model trenowaliśmy paczkami o wielkości 128 obrazów, cała epoka składała się z 50000 obrazów. Przed każdą epoką zbiór treningowy był przetasowywany, aby przeciwdziałać przeuczaniu się sieci. Aby znaleźć najbardziej zadowalające rozwiązanie przetestowane zostały różne konfiguracje treningowe, możliwe zmienne elementy konfiguracji to algorytmy optymalizacyjne opisane w punkcie 4.3.12, funkcje kosztów opisane w punkcie 4.3.10, rodzaje przetwarzania wstępnego opisane w punkcie 4.3.7 oraz skutki zastosowanie takich warstw jak Dropout (punkt 4.3.4) oraz BatchNorm (punkt 4.3.3). Jako funkcja aktywacji wybrana została funkcja ReLU. Dla różnych konfiguracji została także przedstawiona charakterystyka porównawcza.

#### 4.3.14 Rezultaty

Uzyskane wyniki są w znacznej mierze zależne od wybranej konfiguracji. Najbardziej satysfakcyjne rezultaty uzyskane przez model zostały przedstawione na Rysunku 4.16



Rysunek 4.16: Rezultaty.

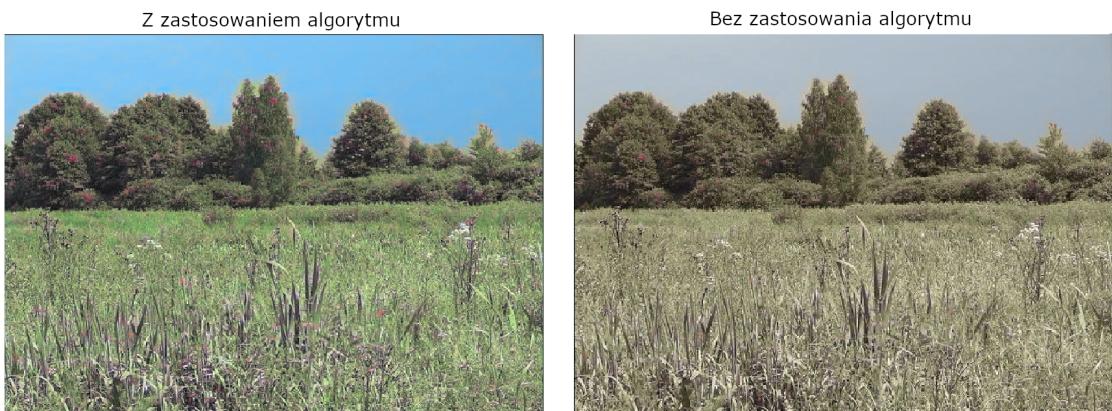
Efekt ten został uzyskany przy następującej konfiguracji:

- Funkcja kosztu MSELoss.
- Funkcja aktywacji ReLU.
- Algorytm optymalizacyjny Adam.
- Normalizacja składowej  $L$  podczas przetwarzania wstępnego.
- Standaryzacja składowych  $A$  i  $B$  podczas przetwarzania wstępnego.
- Nie zastosowanie rozmycia Gaussowskiego.
- Zastosowanie warstwy BatchNorm oraz nie zastosowanie warstwy Dropout.
- Zastosowanie zaproponowanego przez nas algorytmu przetwarzania końcowego.

Na podstawie obrazów na Rysunku 4.16 oraz przeprowadzonych testów można wywnioskować, że model nauczył się najlepiej kolorować powierzchnie o zazwyczaj stałej barwie takie jak niebo albo trawa, widać, że ich barwy są wyjątkowo intensywne oraz jednoznacznie wskazują co to są za powierzchnie. Gorzej rzeczą się ma z obiekta mi o zmiennych barwach takimi jak na przykład samochody, w związku z ich zmienną barwą na przestrzeni całego zbioru danych model, chcąc zminimalizować błąd liczony przez funkcję kosztu, uśredniał kolor tych obiektów co skutkuje ich przytłumionymi barwami przechodzącymi często w odcienie brązu. Z tego powodu pomniejsze detale na kolorowanych zdjęciach tracą swoje barwy, efekty tego zjawiska można zaobserwować analizując stateczniki odrzutowca ukazanego na obrazie (a) Rysunku 4.16.

W dalszej części sekcji opisany został wpływ poszczególnych elementów konfiguracji.

Największy wpływ na uzyskiwane rezultaty ma zastosowanie algorytmu przetwarzania końcowego opisanego w punkcie 4.3.8. Pomógł on znacznie wzbogacić kolory generowane przez sieć. Odpowiednie porównanie zostało przedstawione na Rysunku 4.17. Wadami tej metody są generowane momentami zbyt intensywne barwy zdjęć oraz bardziej widoczne czasami zdarzające się "wycieki" kolorów jednych powierzchni na inne, przykładowo czerwony kolor z samochodu wychodzi poza powierzchnię samochodu i barwi fragmenty trawy przylegające do samochodu na czerwono. Efekt ten jest znacznie mniej widoczny przy zdjęciach o wysokiej rozdzielczości, co sprawia, że w zastosowaniu praktycznym ta wada modelu jest do zaakceptowania. Na Rysunku 4.17 widać to poprzez drobne obszary żółci na granicy nieba i drzew.



Rysunek 4.17: Efekt zastosowania algorytmu przetwarzania końcowego

Kolejnym ważnym komponentem jest użyta funkcja kosztu, właściwy wybór znaczenie poprawia proces aktualizowania się wag sieci, co przekłada się na lepsze rezultaty. Kluczowe było, aby funkcja straty uwzględniała możliwą wielobarwność kolorowanych obiektów i nie karała sieci za barwy niezgodne z barwami na obrazie rzeczywistym, ale pasujące w kontekście kolorowanej powierzchni. Na Rysunku 4.18 zostało umieszczone porównanie rezultatów modeli uczonych z użyciem różnych funkcji kosztu, przedstawione obrazy zostały wygenerowane bez użycia algorytmu końcowego przetwarzania w celu uwydawnienia cech funkcji kosztu.



Rysunek 4.18: Porównanie rezultatów modeli uczonych z użyciem różnych funkcji kosztu.

**L1Loss:** Testując funkcję kosztu L1Loss oczekiwano, że uzyskiwane obrazy będą cechowały się niską saturacją, a generowane kolory będą przytłumione. Jest to spowodowane skłonnościami funkcji L1Loss do uśredniania wyjścia modelu w przypadku, kiedy kolorowana powierzchnia ma zmienną barwę na przestrzeni zbioru uczącego. Działanie L1Loss ma na celu zminimalizować błąd bezwzględny będący różnicą pomiędzy kolorami tych powierzchni, a kolorami generowanymi przez sieć, co skutkuje barwą o wartości znajdującej się pomiędzy wartościami wszystkich występujących kolorów dla powierzchni danego typu. W rezultacie otrzymywane są obrazy w kolorze sepii, co widać na Rysunku 4.18.

**SmoothL1Loss:** Funkcja ta bazuje na L1Loss, ale tworzy kryterium, które przy błędzie bezwzględnym o wartości mniejszej niż 1 używa kwadratu z wartością błędu, a w pozostałych przypadkach używa wartości bezwzględnej błędu. Wzór L1Loss jest przedstawiony w punkcie 4.3.10. Zabieg ten sprawia, że funkcja jest mniej czuła na elementy odstające oraz przeciwdziała zjawisku eksplodującego gradientu. Funkcja ta została zastosowana jako próba skuteczniejszego wyuczenia się przez model kolorów detali występujących na obrazach oraz aby przeciwdziałać uśrednianiu wartości barw przez sieć. Jednakże efekt ten nie został uzyskany, a przeciwdziałać elementom odstającym przez SmoothL1Loss dodatkowo negatywnie wpłynęło na kolory generowane przez sieć. Funkcja wylicza mniejszy błąd, gdy wartość różnicy bezwzględnej pomiędzy wyjściem sieci, a kolorem rzeczywistym jest wysoka, co skutkuje tendencją modelu do generowania składowych  $A$  i  $B$  o wartościach ze środka przedziału wartości, czyli w okolicy zera. W praktyce jednak wartości tych składowych są ograniczane przez składową  $L$  będącą jasnością obrazu, czego wynikiem są średnie wartości generowanych  $A$  i  $B$  lekko poniżej zera. Takie wartości odpowiadają głównie odcieniom niebieskiego i trochę odcieniom zielonego, co wyraźnie widać na Rysunku 4.18.

**MSELoss:** W związku z brakiem algorytmu, który wstrzymuje karanie modelu w sytuacji, gdy generowane kolory są niezgodne z rzeczywistymi, ale prawdopodobne dla kolorowanej powierzchni to funkcja MSELoss również ma tendencję do uśredniania generowanych kolorów jak funkcja L1Loss. Jednakże większa czułość funkcji na wysokie wartości błędów, powodowana podnoszeniem różnicy pomiędzy wartością przewidzianą a rzeczywistą do kwadratu, umożliwiła wytrenowanie modelu zdolnego do generowania barw z szerokiego zakresu wartości, co korzystnie wpłynęło na rezultaty. Pokolorowany obraz wygenerowane przez model uczony z użyciem funkcji MSELoss prezentuje się najlepiej spośród obrazów na Rysunku 4.18, z tego powodu oraz w związku z korzystnym efektem zastosowania MSELoss, właśnie ta funkcja została wybrana w procesie uczenia modelu końcowego.

<Skutki warstwy Dropout >

<Skutki różnych metod przetwarzania wstępnego >

<Skutki różnych algorytmów optymalizacyjnych >

<Skutki różnych algorytmów optymalizacyjnych >

## **5 PODSUMOWANIE**

## BIBLIOGRAFIA

- [1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio: *Generative Adversarial Networks*, arXiv ('2014)
- [2] Andrew Brock, Theodore Lim, J.M. Ritchie, Nick Weston: *NEURAL PHOTO EDITING WITH INTROSPECTIVE ADVERSARIAL NETWORKS*, arXiv ('2017)
- [3] Tianyi Liu, Shuang Sang Fang, Yuehui Zhao, Peng Wang, Jun Zhang: *Implementation of Training Convolutional Neural Networks*, arXiv ('2015), s.2.
- [4] Ian Goodfellow, Yoshua Bengio, Aaron Courville: *Deep Learning*, ('2016), s.342., s.82.
- [5] Richard Zhang, Phillip Isola, Alexei A. Efros: *Colorful Image Colorization*, arXiv ('2016)
- [6] Leon A. Gatys, Alexander S. Ecker, Matthias Bethge: *Image Style Transfer Using Convolutional Neural Networks*, IEEE ('2016)
- [7] Karen Simonyan, Andrew Zisserman: *Very Deep Convolutional Networks For Large-Scale Image Recognition*, ('2016)
- [8] Guim Perarnau, Joost van de Weijer, Bogdan Raducanu, Jose M. Álvarez: *Invertible Conditional GANs for image editing*, arXiv ('2016)
- [9] Diederik P. Kingma, Max Welling: *Auto-Encoding Variational Bayes*, arXiv ('2014)
- [10] Sergey Ioffe, Christian Szegedy: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv ('2015)
- [11] Y. Bengio, P. Simard, P. Frasconi: *Learning long-term dependencies with gradient descent is difficult*, IEEE ('1994)
- [12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research ('2014)
- [13] A. Krizhevsky: *Learning Multiple Layers of Features from Tiny Images*, Master's thesis, Department of Computer Science, University of Toronto ('2009)
- [14] R. Girshick: *Fast R-CNN*, arXiv ('2015)
- [15] B. Xu, N. Wang, T. Chen, M. Li: *Empirical Evaluation of Rectified Activations in Convolution Network*, arXiv ('2015)
- [16] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, Adam Lerer: *Automatic differentiation in PyTorch*, 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA
- [17] Russell Reed, Robert J MarksII: *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, Paperback – February 17, 1999, s.155.

- [18] Raul Rojas: *Neural Networks - A Systematic Introduction*, Springer-Verlag, Berlin, New-York, 1996, s.161.
- [19] Irwin Sobel: *An Isotropic 3x3 Image Gradient Operator*, ResearchGate, 2014, s.4.

## Spis rysunków

2.1	Struktura DNN - źródło: <a href="https://towardsdatascience.com/building-a-convolutional-neural-network-male-vs-female-50347e2fa88b">https://towardsdatascience.com/building-a-convolutional-neural-network-male-vs-female-50347e2fa88b</a> . . . . .	10
2.2	Przykładowa struktura CNN - źródło: <a href="https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html">https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html</a> . . . . .	11
2.3	Segmentacja semantyczna - źródło: <a href="https://devblogs.nvidia.com/image-segmentation-using-digits-5/">https://devblogs.nvidia.com/image-segmentation-using-digits-5/</a> . . . . .	12
3.1	Efekt kolorowanie czarno-białych zdjęć przez wytrenowany model - źródło: [5] . . . . .	14
3.2	Obrazy będące kombinacją treści zdjęcia ze stylami kilku znanych dzieł sztuki - źródło: [6] . . . . .	15
3.3	Obrazy generowane przez IcGAN - źródło: [8] . . . . .	16
3.4	Efekt działania Neural Photo Editor - źródło: [2] . . . . .	18
4.1	Architektura TorchFrame - źródło: Praca własna . . . . .	22
4.2	Przykładowa płaszczyzna funkcji celu - źródło: <a href="https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0">https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0</a> . . . . .	27
4.3	Przykładowy kształt punktu siodłowego - źródło: <a href="https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0">https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0</a> . . . . .	29
4.4	Architektura testowa TorchFrame - źródło: Praca własna . . . . .	34
4.5	Przykładowe obrazy ze zbioru treningowego - źródło: Praca własna . . . . .	36
4.6	Działanie filtra Sobela - źródło: Praca własna . . . . .	37
4.7	Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych - źródło: Praca własna . . . . .	38
4.8	Działanie sepii - źródło: Praca własna . . . . .	40
4.9	Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych - źródło: Praca własna . . . . .	41
4.10	Przykład działania sieci w minimum lokalnym - źródło: Praca własna . . . . .	42
4.11	Przestrzeń barw CIELab - źródło: <a href="https://www.flickr.com/photos/greenmambagreenmamba/4236391637">https://www.flickr.com/photos/greenmambagreenmamba/4236391637</a> . . . . .	44
4.12	Przykładowa składowa $L$ - źródło: Rysunek własny wykorzystujący: <a href="https://fr.m.wikipedia.org/wiki/Fichier:An_F-A-18C_Hornet_launches_from_the_flight_deck_of_the_conventionallyPowered_aircraft_carrier.jpg">https://fr.m.wikipedia.org/wiki/Fichier:An_F-A-18C_Hornet_launches_from_the_flight_deck_of_the_conventionallyPowered_aircraft_carrier.jpg</a> . . . . .	45
4.13	Przykładowe operacje warstwy poolingu - źródło: Rysunek własny . . . . .	48
4.14	Przykładowe obrazy z CIFAR-10 - źródło: Rysunek własny . . . . .	48
4.15	Funkcja aktywacji ReLU - źródło: <a href="https://pytorch.org/docs/stable/_images/ReLU.png">https://pytorch.org/docs/stable/_images/ReLU.png</a> . . . . .	53
4.16	Efekty działania sieci - źródło: Rysunek własny bazujący na: <a href="https://fr.m.wikipedia.org/wiki/Fichier:An_F-A-18C_Hornet_launches_from_the_flight_deck_of_the_conventionallyPowered_aircraft_carrier.jpg">https://fr.m.wikipedia.org/wiki/Fichier:An_F-A-18C_Hornet_launches_from_the_flight_deck_of_the_conventionallyPowered_aircraft_carrier.jpg</a> , <a href="https://pl.wikipedia.org/wiki/Plik:PL_Bagno_Całowanie_2.jpg">https://pl.wikipedia.org/wiki/Plik:PL_Bagno_Całowanie_2.jpg</a> , [13] . . . . .	55
4.17	Efekt zastosowania algorytmu przetwarzania końcowego - źródło rysunek własny na podstawie: <a href="https://pl.wikipedia.org/wiki/Plik:PL_Bagno_Całowanie_2.jpg">https://pl.wikipedia.org/wiki/Plik:PL_Bagno_Całowanie_2.jpg</a> . . . . .	56
4.18	Porównanie rezultatów modeli uczonych z użyciem różnych funkcji kosztu - źródło rysunek własny na podstawie: <a href="https://cdn.theearthhunters.com/wp-content/uploads/2013/06/bg-960x636.jpg">https://cdn.theearthhunters.com/wp-content/uploads/2013/06/bg-960x636.jpg</a> . . . . .	57

## **Spis tabel**

1	Funkcje kosztu w TorchFrame . . . . .	26
2	Optymalizatory w TorchFrame . . . . .	31
3	Architektura modelu podstawowego. . . . .	45