



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

PROJEKT DYPLOMOWY INŻYNIERSKI

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

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

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

Data oddania projektu do dziekanatu:



OŚWIADCZENIE dotyczące projektu dyplomowego zatytułowanego: Zastosowanie sieci neuronowych do edycji obrazów

Imię i nazwisko studenta: Piotr Winkler

Data i miejsce urodzenia: 29.03.1997, Czuchów

Nr albumu: 165504

Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki

Kierunek: automatyka i robotyka

Poziom kształcenia: pierwszy

Forma studiów: stacjonarne

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. 2018 poz. 1191 z późn. zm i konsekwencji dyscyplinarnych określonych w ustawie z dnia 20 lipca 2018 r. Prawo szkolnictwie wyższym i nauce (Dz. U. 2018 poz. 1668 z późn. zm.),¹ a także odpowiedzialność cywilnoprawnej oświadczam, że przedkładany projekt dyplomowy został opracowany przez mnie samodzielnie.

Niniejszy projekt dyplomowy nie był wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

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

Potwierdzam zgodność niniejszej wersji projektu dyplomowego z załączoną wersją elektroniczną

Gdańsk, dnia

.....

podpis studenta

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

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

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

STRESZCZENIE

Tematem projektu 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. Następnie skuteczność tych narzędzi została oceniona i porównana z rozwiązaniami opartymi na klasycznych metodach przetwarzania obrazu.

Zbadane zostały dwie problematyki, automatyczne kolorowanie czarno-białych obrazów oraz nakładanie na obraz prostych filtrów takich jak wykrywający krawędzie filtr Sobela-Feldmana.

W celu ułatwienia przeprowadzania badań został opracowany autorski framework *TorchFrame*. Ma on za zadanie kontrolować przepływ danych w procesie uczenia i testowania badanych modeli, a co za tym idzie, ograniczyć konieczność ingerencji ze strony użytkownika, przy jednoczesnym zapewnieniu swobody w prowadzonych eksperymentach i implementowanych modyfikacjach danych.

W przypadku prostych filtrów obrazu rozwiązania oparte na sieciach splotowych okazały się wolniejsze i bardziej pracochłonne w implementacji niż metody klasyczne przy porównywalnych rezultatach. Jednakże z ich pomocą udało się udowodnić olbrzymią uniwersalność sieci splotowych mogących znaleźć zastosowanie w zagadnieniach związanych z prostym przetwarzaniem obrazu. Ukażane zostało ponadto podobieństwo między tradycyjnymi metodami filtracji, a sposobem działania neuronów tworzących sieci konwolucyjne.

Do rozwiązania problematyki automatycznego kolorowania czarno-białych obrazów zostały przyjęte dwa różne podejścia. Pierwsze obejmowało zastosowanie prostego modelu autorskiego treningowego z użyciem wielu różnych hiperparametrów oraz metod przygotowania danych treningowych. Otrzymane wyniki były zadowalające, jednakże są one w dużej mierze zależne od wybranych hiperparametrów oraz konfiguracji, w których uczona była sieć.

Drugie podejście obejmowało zaimplementowanie bardziej złożonego modelu z użyciem techniki przeniesienia uczenia. Podejście to oparte było na integracji cech obrazu średniego oraz wysokiego poziomu. Tak uzyskane informacje były następnie wykorzystywane do predykcji prawdopodobnych barw obrazu. W rezultacie przełożyło się to na większą realistyczność otrzymanych kolorów, a co za tym idzie, wysoce satysfakcjonujące wyniki.

Badania przeprowadzone w ramach tego projektu dyplomowego dowodzą, że sieci neuronowe mogą być skutecznie zastosowane jako narzędzia do wszechstronnej edycji obrazu. Jednakże, pomimo ich niezwykłych możliwości, sukces zależy w dużej mierze od dobrze przemyślanego wyboru stosowanej architektury oraz poprawnie przeprowadzonego procesu uczenia.

Słowa kluczowe: sieć neuronowa, przetwarzanie obrazu, konwolucyjna sieć neuronowa, splotowa sieć neuronowa, głęboka sieć neuronowa, filtry obrazu, automatyczne kolorowanie czarno-białych obrazów, platforma programistyczna

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

ABSTRACT

The subject of this project is to explore the possibilities of using neural networks for image editing. The main goal was to create tools, based on trained neural networks, used for proper processing and modifying of images. Then the effectiveness of these tools was evaluated and compared with solutions based on classic image processing methods.

Two approaches were investigated, automatic coloring of black and white images and applying simple filters on images such as Sobel-Feldman filter detecting edges.

To make the research easier, an original framework named TorchFrame has been created. It has the task of controlling data flow in the process of teaching and testing the examined models and hence reduce the need for user interference while ensuring freedom in conducted experiments and implemented data modifications.

For simple image filters, solutions based on convolutional neural networks proved to be slower and more labor-intensive in implementation than classical methods, while presenting comparable results. However, with their help it was possible to prove the enormous universality of convolutional neural networks, which can be used in issues related to simple image processing. Furthermore, the similarity between traditional filtration methods and the way, how the neurons in convolutional networks works, has been shown.

To solve the problem of black and white images colorization, two different approaches were proposed. The first one included the application of simple author's model trained with many different hiperparameters and data preparation methods. Obtained results were satisfactory but they are greatly dependent from chosen hiperparameters and training configuration.

The second approach included implementing more complex model using transfer learning technique. This approach was based on the integration of medium and high level features of the image. The information obtained that way was then used to predict possible colors of the image. At the end, this method resulted in more realistic colors and hence highly satisfying results.

Experiments, carried out as part of this project, proves that neural networks can be successfully used as tools in general image editing process. However, despite their remarkable possibilities, success highly depends on the well considered choice of used architecture and correctly carried out training process.

Keywords: neural network, image processing, convolutional neural network, generative adversarial network, deep neural network, image filters, automatic image colorizing, framework

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

SPIS TREŚCI

| | |
|--|----|
| WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW | 6 |
| 1. Wstęp i cel pracy | 8 |
| 1.1. Cel pracy | 8 |
| 1.2. Założenia projektowe | 9 |
| 1.3. Układ pracy | 9 |
| 2. Podstawy teoretyczne (Piotr Winkler) | 10 |
| 2.1. Sieci splotowe | 10 |
| 2.2. FCN | 12 |
| 2.3. Modele generatywne | 13 |
| 3. Przegląd rozwiązań (Bartosz Bieliński) | 14 |
| 3.1. Colorful image colorization | 14 |
| 3.2. Image Style Transfer Using Convolutional Neural Networks | 15 |
| 3.3. Invertible Conditional GANs for image editing | 17 |
| 3.4. Neural photo editing | 18 |
| 4. Zaimplementowane rozwiązania | 20 |
| 4.1. <i>TorchFrame</i> (Piotr Winkler) | 21 |
| 4.1.1. <i>PyTorch</i> | 21 |
| 4.1.2. Architektura <i>TorchFrame</i> | 22 |
| 4.1.3. Konfiguracja w <i>TorchFrame</i> | 24 |
| 4.1.4. Testowanie w <i>TorchFrame</i> | 35 |
| 4.2. Filtry AI (Piotr Winkler) | 37 |
| 4.2.1. Filtr Sobela | 37 |
| 4.2.2. Sepia | 41 |
| 4.2.3. Filtr górnoprzepustowy | 44 |
| 4.2.4. Podsumowanie | 46 |
| 4.3. Automatyczne kolorowanie czarno-białych obrazów (Bartosz Bieliński) | 48 |
| 4.3.1. Podejście | 48 |
| 4.3.2. Model autorski | 50 |
| 4.3.3. BatchNorm | 51 |
| 4.3.4. Dropout | 52 |
| 4.3.5. Modyfikacja rozdzielczości | 52 |

| | |
|--|----|
| 4.3.6. Wykorzystywany zbiór treningowy | 53 |
| 4.3.7. Przetwarzanie wstępne danych | 54 |
| 4.3.8. Przetwarzanie końcowe danych | 55 |
| 4.3.9. Augmentacja danych | 56 |
| 4.3.10. Funkcje kosztów | 57 |
| 4.3.11. Funkcje aktywacji | 58 |
| 4.3.12. Algorytmy optymalizacyjne | 59 |
| 4.3.13. Trening | 60 |
| 4.3.14. Rezultaty | 60 |
| 4.3.15. Model autorski - wnioski | 67 |
| 4.3.16. Model z wykorzystaniem przeniesienia uczenia | 68 |
| 4.3.17. Podsumowanie | 72 |
| 5. Podsumowanie | 74 |
| WYKAZ LITERATURY | 76 |
| SPIS RYSUNKÓW | 78 |
| SPIS TABEL | 80 |

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 gęsta
- CNN (ang. Convolutional Neural Network) - Splotowa sieć neuronowa
- FCN (ang. Fully Convolutional Network) - Sieć w pełni splotowa
- GAN (ang. Generative Adversarial Network) - Generatywne sieci przeciwnostawne
- VAE (ang. Variational Autoencoder) - Autoenkodery wariancyjne
- ReLU (ang. Rectified Linear Unit) - Jednostronnie obcięta funkcja liniowa
- BatchNorm (ang. Batch Normalization) - Normalizacja zbioru danych pogrupowanych w pakiety
- YUV - Model barw, w którym kanał Y odpowiada za luminancję obrazu, a UV są to dwa kanały chrominacji kodujące barwy
- IcGAN (ang. Invertible conditional Generative Adversarial Network) - Odwracalne, warunkowe, generatywne sieci przeciwnostawne
- cGAN (ang. conditional Generative Adversarial Network) - Warunkowe, generatywne sieci przeciwnostawne
- Dropout (pol. algorytm odrzucania) - 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) - Biblioteka wizji komputerowej z licencją zgodną z zasadami wolnego oprogramowania
- tensor - Macierzowa struktura danych biblioteki PyTorch
- SGD (ang. Stochastic Gradient Descent) - Stochastyczny Spadek Gradientu
- AdaGrad (ang. Adaptive Gradient Algorithm) - Adaptacyjny Algorytm Gradientowy

AdaDelta (ang. Adaptive Delta) - Rozszerzenie Adaptacyjnego Algorytmu Gradientowego

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

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

p - Prawdopodobieństwo dezaktywowania neuronu sieci przez warstwę Dropout

ResNet (ang. Residual Networks) - Sieć szcątkowa

RGB (ang. Red Green Blue) - Model przestrzeni barw

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 wzmogło zainteresowanie tematem, jednak tym razem na drodze postępowi stanęły ograniczenia technologiczne ówczesnych czasów.

Wreszcie, wraz z nadaniem 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 odnoszą zastosowanie w wielu dziedzinach życia i nauki. Grają w gry, przeprowadzają symulacje, przewidują i prognozują zachowania rynku czy pogody, a także analizują i przetwarzają obrazy cyfrowe.

Z punktu widzenia niniejszej pracy największe znaczenie ma oczywiście ostatnie z wymienionych zastosowań. 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 niniejszej pracy jest zbadanie możliwości zastosowania sieci neuronowych do edycji obrazu. Wiąże się to ze stworzeniem narzędzi programistycznych umożliwiających implementowanie, trenowanie oraz testowanie sztucznych sieci neuronowych przeznaczonych do przetwarzania obrazów cyfrowych. W ramach podjętej tematyki szczególny nacisk położony zostanie na przetestowanie rozwiązań dedykowanych do pracy z grafiką, takich jak sieci splotowe.

Po opracowaniu tychże narzędzi, opisane zostaną osiągnięte efekty pracy oraz zbadana zostanie skuteczność sieci neuronowych jako rozwiązań nakreślonej problematyki. Otrzymane rezultaty zostaną także porównane z rezultatami rozwiązań opartych na klasycznych metodach przetwarzania obrazów nie wykorzystujących technologii sztucznych sieci neuronowych. Omówione zostaną również wykorzystane architektury zaimplementowanych modeli oraz przetestowane konfiguracje procesu uczenia, zwłaszcza wykorzystane hiperparametry oraz sposoby przygotowania danych treningowych.

1.2. Założenia projektowe

Głównym założeniem pracy jest zaprojektowanie i zaimplementowanie służących do edycji obrazu narzędzi programistycznych, które powinny wykorzystywać do swoich celów odpowiednio wytrenowane sieci neuronowe. Na podstawie przeprowadzonych testów oceniona zostanie skuteczność wykorzystanych modeli oraz poprawność otrzymanych wyników.

Wykonana zostanie również analiza słuszności zastosowania sieci neuronowych jako rozwiązania przedstawionej problematyki, a uzyskane rezultaty zestawione zostaną ze znanyimi metodami edycji obrazu nie opierającymi się na technologii sztucznej inteligencji.

1.3. Układ pracy

W pierwszym rozdziale opisane zostaną podstawy teoretyczne, obejmujące najważniejsze zagadnienia związane z omawianą w pracy tematyką edycji obrazów z wykorzystaniem sieci neuronowych. Znaczący fragment tego rozdziału dedykowany jest sieciom splotowym stanowiącym most pomiędzy sztuczną inteligencją, a zagadnieniami związanymi z szeroko pojętym przetwarzaniem grafiki.

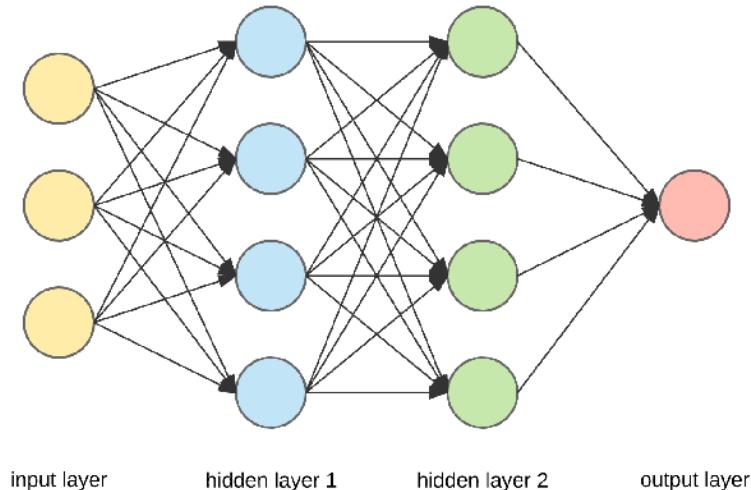
Następnie zaprezentowane zostaną dotychczasowe dokonania i rezultaty pracy grup badawczych specjalizujących się w graficznych zastosowaniach sztucznej inteligencji. Przeanalizowana zostanie idea kryjąca się za każdym z analizowanych rozwiązań oraz uzyskane z ich pomocą wyniki. Przedstawione w tym rozdziale badania stanowią inspirację dla przygotowanych rozwiązań autorskich.

Kolejny rozdział pracy stanowić będzie szczegółowy przegląd przygotowanego oprogramowania oraz zaimplementowanych za jego pomocą sieci neuronowych przeznaczonych do edycji obrazów. Zawarty zostanie opis prostych filtrów, demonstrujących ideę zastosowania sieci splotowych, oraz zaawansowanego modelu przeznaczonego do automatycznego kolorowania czarno-białych obrazów.

W ostatnim rozdziale zawarte zostanie podsumowanie uzyskanych rezultatów wraz z krytyczną analizą zasadności zastosowania sztucznej inteligencji w zakresie grafiki cyfrowej.

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 nielinowych 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. Neurony w nich zawarte połączone były na zasadzie każdy z każdym, a każde takie połączenie opisane było za pomocą wagi modyfikowanej 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 głębokiej sieci neuronowej składającej się wyłącznie z warstw gęstych

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 [1] opisują je następująco:

'CNN to skuteczny algorytm poznawczy, stosowany powszechnie przy rozpoznawaniu wzorów 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.

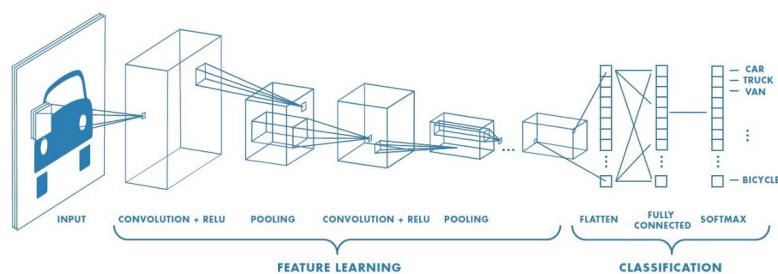
Podstawową warstwą stanowi warstwa splotowa. Składa się ona ze zbioru filtrów (neuronów) odpowiedzialnych za ekstrakcję cech z analizowanych obrazów w ramach operacji konwolucji. Proces ten polega na równomiernym przesuwaniu macierzy filtrujących, składających się z wag neuronów, wzdłuż przetwarzanych obrazów. Na wyjściu filtrów otrzymuje się macierze o mniejszej rozdzielcości reprezentujące wyniki operacji splotu. Każda kolejna warstwa konwolucyjna 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 [2]:

'We wszystkich przypadkach pooling pomaga uczynić reprezentację w przybliżeniu niezmienną w stosunku do małych translacji danych wejściowych. Niezmiennałość wobec translacji oznacza, że jeśli poddamy dane wejściowe nieznaczemu przesunięciu, to wartość większości wyników poddanych poolingowi 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 wcześniej elementy składowe sieci splotowych mogą przyjmować różne rozmiary i występować w różnych konfiguracjach, co przedstawiono na Rysunku 2.2.

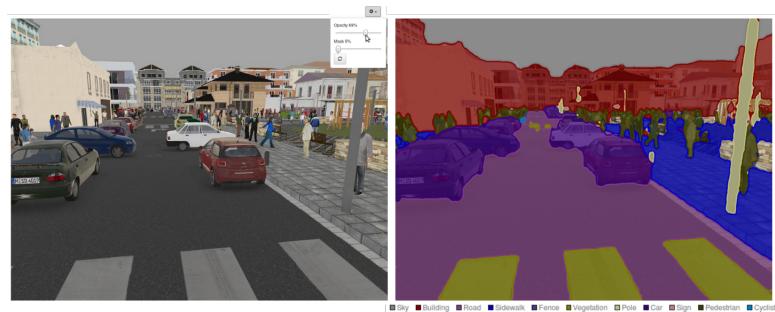


Rysunek 2.2: Przykładowa struktura CNN

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. Zostało to przedstawione na 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 rozdzielczości, wygenerowanych przez enkoder, na wysoką rozdzielczość 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.

Naprzemiennie przepuszczanie obrazów przez wspomniane warstwy splotowe oraz pooling może powodować niską rozdzielczość końcowych 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 [3]. Modele te stanowią połączenie dwóch głębokich sieci neuronowych działających przeciwnie 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 od 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.

3. PRZEGŁĄD ROZWIĄZAŃ

Na przestrzeni ostatnich paru lat pojawiło się wiele innowacyjnych technologii opartych na sieciach neuronowych. Takie cechy sieci, jak niezwykłe zdolności do generalizacji zdobytej wiedzy na nowe przypadki oraz olbrzymia elastyczność sprawiły, że znalazły one wiele rzeczywistych zastosowań, zwłaszcza do problemów nieszablonowych, dla których metody nie oparte na uczeniu maszynowym nie przynosiły zadowalających wyników. Zastosowania te często były przełomowe w swojej dziedzinie i do czasów dzisiejszych uważane są za prekursorów pewnych idei.

W tym rozdziale skupiono się na przedstawieniu kilku interesujących rozwiązań stosujących sieci neuronowe do edycji obrazu, które są zarazem kluczowe do lepszego zapoznania się z omawianą problematyką.

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 [4]. Celem 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 obrazów przez wytrenowany model.

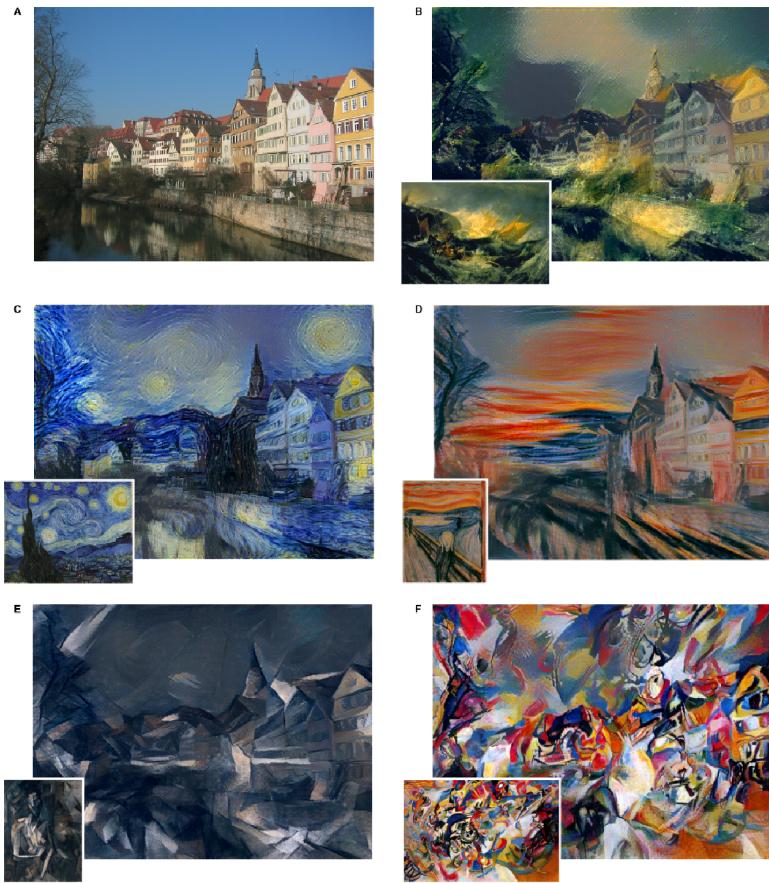
Wykorzystany model składa się z wielu warstw CNN, w których skład wchodzą warstwa filtrów konwolucyjnych, warstwa ReLU (ang. Rectified Linear Unit) oraz warstwa BatchNorm (ang. Batch normalization). Aby zapobiec utracie informacji przestrzennych, sieć nie posiada warstw poolingu. Istotny był także sposób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

Ważnym aspektem zbadanym w artykule było także dobranie odpowiedniej funkcji kosztu. Nieodpowiedni 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 poprawie wartości wyliczanego dla modelu błędu, w celu wyróżnienia rzadko spotykanych kolorów.

Powstałe rozwiązanie dowodzi olbrzymiego potencjału zastosowania sieci neuronowych w dziedzinie pracy nad obrazami, efekty uzyskiwane za ich pomocą są niemożliwe do odtworzenia z użyciem tradycyjnych algorytmów.

3.2. *Image Style Transfer Using Convolutional Neural Networks*

W roku 2016 został przedstawiony światu *A Neural Algorithm of Artistic Style* [5]. Wprowadził on przełom w dziedzinie przenoszenia stylu jednego obrazu na inny, a jego sukces opierał się na właściwym wykorzystaniu konwolucyjnych sieci neuronowych. Podstawą tego osiągnięcia 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.

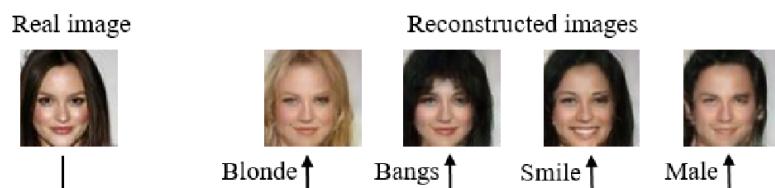
Do zbudowania modelu zostały użyte warstwy konwolucyjne oraz poolingu z architektury VGG-Network [6], która została wytrenowana pod kątem rozpoznawania obiektów i określania ich położenia. Dzięki temu sieć przetwarzając obraz tworzy jego reprezentację, która wraz z kolejnymi warstwami, przedstawia coraz pełniejszą 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 najczęściej stosowaną wartością maksymalną. Takie zabiegi umożliwiają wyliczenie reprezentacji stylu z korelacji pomiędzy poszczególnymi cechami w kolejnych warstwach konwolucyjnych.

Cały proces renderowania polega na odpowiednim przechowaniu w modelu treści oraz stylu wejściowych obrazów i składa się z wielu następujących po sobie etapów. Wpierw obraz, z którego pobierany jest styl, jest podawany na wejście sieci oraz przetwarzany, reprezentacja stylu, wyselekcjonowana z właściwych warstw, jest odpowiednio przechowywana. Następnie temu samemu procesowi poddawany jest obraz z treścią, ale reprezentacja treści jest wyciągana z ostatnich warstw konwolucyjnych. W celu wykonania fuzji obrazów, uzyskane reprezentacje treści oraz stylu są zapisywane w tych warstwach modelu skąd zostały odczytane, po czym 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 niezłożone, 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 [7]. Zaprezentowany model to enkoder z możliwością generowania wektora informacji o atrybutach obrazu, połączony z warunkowym GANem zdolnym do kontrolowania cech generowanych obrazów na podstawie dodatkowej informacji warunkowej. Takie działanie umożliwia wprowadzanie zmian w atrybutach generowanego obrazu uzyskiwanego na wyjściu 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 podzielnych enkoderów. Enkoder E_z koduje wejściowy obraz do utajonego wektora z reprezentacji obrazu, natomiast enkoder E_y generuje wektor informacji y , oddający pewne kluczowe atrybuty obrazu. Enkodery 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 enkodery 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 jeden enkoder o współdzielonych warstwach i dwóch wyjściach.

W przypadku cGAN można 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, pojawia się możliwość wystąpienia niepożądanej przeuczenia modelu. Autorzy artykułu dokonali analizy tego rozwiązania, a także zbadali wydajność metod 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.

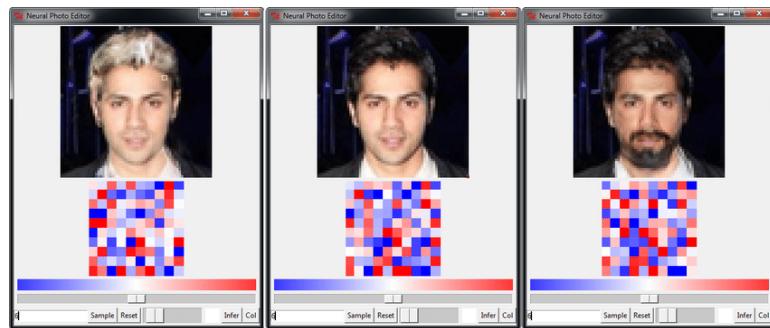
3.4. Neural photo editing

W 2017 roku Andrew Brock, Theodore Lim, J.M. Ritchie and Nick Weston zaprezentowali *Neural Photo Editor* [8], narzędzie do edytowania obrazu wyposażone w mechanizmy wykrywania kontekstu zmian. Twórcy opisują swoje dzieło 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 wprowadzanej edycji. Następnie na jego podstawie wprowadza zmiany semantyczne w kontekście żądanej zmiany koloru. Efekt działania interfejsu został przedstawiony na Rysunku 3.4.

Skuteczność NPE (ang. Neural Photo Editor) polega na zastosowaniu IAN (ang. Introspective Adversarial Network), czyli sieci złożonej z połączonych VAE (ang. Variational Autoencoder) [9] oraz GANów, w taki sposób, że dekodująca sieć autoenkodera jest używana jako sieć generująca w GANie. Poprzez przechwytywanie przez model dalekosiążnych 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.



Rysunek 3.4: Efekt działania Neural Photo Editor.

Powstanie NPE utwierdza w przekonaniu, że aktualnie istniejące sieci neuronowe do edycji obrazu znacznie przewyższają zwykłe algorytmy pod względem możliwości, a także są zauważalnie słabiej uzależnione od wkładu ludzkiego.

4. ZAIMPLEMENTOWANE 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.

Zadania te zostały zrealizowane, a szczegółowe opisy rozwiązań umieszczone w następującym rozdziale.

W pierwszej kolejności zaprezentowany został autorski framework (pol. platforma programistyczna) *TorchFrame*, stworzony w języku programistycznym Python i przeznaczony do przyspieszenia i uproszczenia pracy z sieciami neuronowymi. Z jego pomocą wykonane zostały modele przeznaczone do różnego rodzaju filtrowania i przetwarzania obrazów. Rozwiązania te skupiły się na rozległej tematyce, poczynając od prostych filtrów demonstrujących działanie sztucznych sieci splotowych, a kończąc na złożonym modelu przeznaczonym do intelligentnego kolorowania czarno-białych obrazów.

W ramach przedstawionych rozwiązań zamieszczone zostały porównania wyników osiągniętych rozmaitymi metodami treningowymi wraz z rozważaniami teoretycznymi związanymi z zastosowaniem różnorodnych hiperparametrów w procesie uczenia sieci. Ponadto, w ramach kolejnych podrozdziałów, opisane zostały istotne elementy składające się na proces projektowania i tworzenia sztucznych sieci neuronowych.

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 uczenia 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 do tego celu 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 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 biblioteka programistyczna stworzona przez oddział sztucznej inteligencji firmy Facebook. W jednym z artykułów [10] 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 modelu. Wykorzystuje ponadto zdefiniowane odgórnie hiperparametry, takie jak funkcje kosztu. Sam rdzeń *TorchFrame'a* bazuje 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.

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

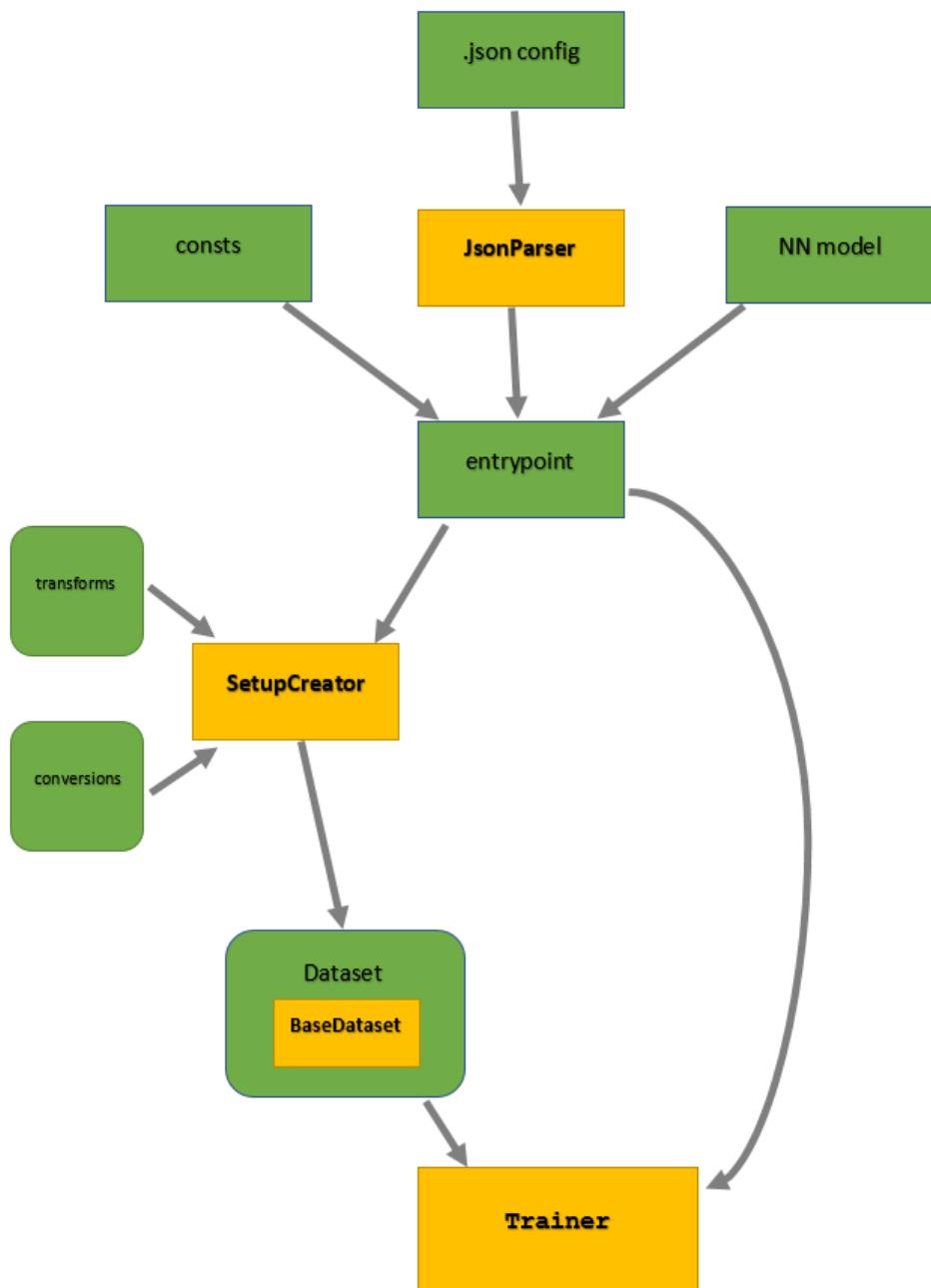
W pliku *consts* użytkownik *TorchFrame* definiuje zmienne środowiskowe, takie jak ścieżki do plików kluczowych w procesie uczenia. Konieczne jest jedynie wskazanie położenia pliku konfiguracyjnego w formacie *JSON*. Pozostałe parametry są opcjonalne i mogą służyć do wskazania miejsca zapisu gotowego modelu 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 frameworka *PyTorch* przytoczonego w poprzednim punkcie podrozdziału. 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*. Udostępnia 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 danych w ramach procesu uczenia oraz podczas testów. Dokładna struktura tego pliku zostanie opisana w następnym punkcie.

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.



Rysunek 4.1: Architektura *TorchFrame*

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 obrębie *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 przetwarzania 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 architektury 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 wyznaczania kierunku uczenia w ramach funkcji celu i wstępnej propagacji błędu modyfikującej wagę sieci w czasie treningu. *Trainer* udostępnia na bieżąco dane pozwalające określić skuteczność uczenia sieci, takie jak aktualny błąd, ilość产生的 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 dostosowywać 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* [2] 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 poprawa wartości tej liczby była oznaką poprawy całego modelu. [11]

'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. [11]

'(...) 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*, zaczerpniętych z biblioteki *PyTorch*, zamieszczone zostały w tabeli 4.1.

Tabela 4.1: Funkcje kosztu w *TorchFrame*

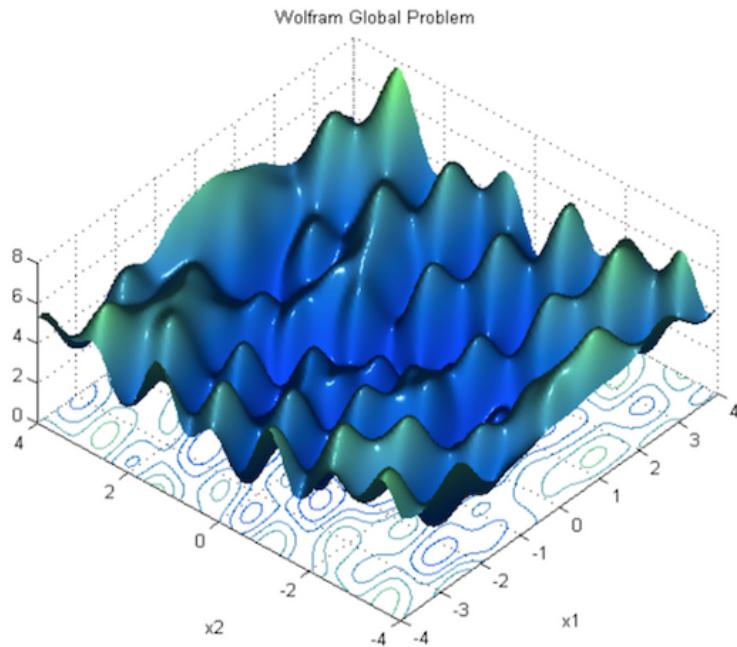
| Funkcja kosztu | Opis |
|----------------|---|
| L1Loss | <p>Funkcja kosztu mierząca błąd bezwzględny pomiędzy odpowiadającymi sobie elementami ze zbioru wyjściowego i docelowego. Można ją opisać za pomocą następującego wzoru:</p> $l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = x_n - y_n ,$ <p>gdzie N jest rozmiarem pojedynczego pakietu danych, a x i y to tensory o arbitralnym kształcie, z których każdy posiada n elementów.</p> <p>Parametry:</p> <ul style="list-style-type: none"> • 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 wyliczenie średniej wartości elementów wyjściowych. <i>Sum</i> oznacza natomiast, że wyznaczona zostanie ich suma. |

| | |
|------------------|--|
| MSELoss | <p>Funkcja kosztu mierząca błąd kwadratowy pomiędzy każdym elementem wyjściowym x i celem y. Opisuje ją poniższy wzór:</p> $l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = (x_n - y_n)^2,$ <p>gdzie N jest rozmiarem pojedynczego pakietu danych, a x i y to tensory o arbitralnym kształcie, z których każdy posiada n elementów.</p> <p>Parametry:</p> <ul style="list-style-type: none"> • <i>reduction</i> - czytaj <i>reduction</i> dla funkcji <i>L1Loss</i>. |
| KLDivLoss | <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 próbkiwanych) ciągłych rozkładów wyjściowych. Kryterium to wymaga, aby rozmiary tensorów wejściowych i wyjściowych były identyczne.</p> <p>Wzór matematyczny opisujący rozbieżność KL przedstawiony został poniżej:</p> $l(x, y) = L = \{l_1, \dots, l_N\}, l_n = y_n \cdot (\log y_n - x_n),$ <p>gdzie indeks N obejmuje wszystkie wymiary wejściowe, a L ma ten sam kształt, co dane wejściowe.</p> <p>Parametry:</p> <ul style="list-style-type: none"> • <i>reduction</i> - poza wartościami opisanymi dla funkcji <i>L1Loss</i> może przyjmować również parametr <i>batchmean</i>. Wymusza on sumowanie wartości wyjściowych, a następnie podzielenie sumy przez rozmiar pakietu danych. |
| BCELoss | <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 N jest rozmiarem pakietu danych.</p> <p>Parametry:</p> <ul style="list-style-type: none"> • <i>weight</i> - manualna wartość wagi przeskalowującej funkcję kosztu każdego elementu w pakiecie danych. • <i>reduction</i> - czytaj <i>reduction</i> dla funkcji <i>L1Loss</i>. |

| | |
|----------------------|--|
| SmoothL1-Loss | <p>Kryterium przyjmujące postać funkcji <i>MSE</i> w przypadku, gdy wartość błędu bezwzględnego spada poniżej 1 oraz funkcji <i>L1</i> w przeciwnym wypadku. Funkcja ta jest mniej czuła na wartości odstające niż <i>MSELoss</i>, a w niektórych przypadkach zapobiega zjawisku eksplodującego gradientu. Znana jest również jako funkcja kosztu <i>Huber'a</i>.</p> <p>Wzór opisujący:</p> $loss(x, y) = \frac{1}{n} \sum_i z_i,$ <p>gdzie z_i zdefiniowane jest następująco:</p> $y = \begin{cases} 0.5 \cdot (x_i - y_i)^2 & \text{gdy } x_i - y_i < 1 \\ x_i - y_i - 0.5, & \text{w pozostałych przypadkach} \end{cases}$ <p>Parametry:</p> <ul style="list-style-type: none"> • reduction - czytaj <i>reduction</i> dla funkcji <i>L1Loss</i>. |
|----------------------|--|

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 wagи 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 η jest długością kroku treningowego, a $\nabla J(\Theta)$ oznacza gradient funkcji kosztu zależnej od wag sieci Θ . Warto zauważyć, że wartość gradientu wskazuje położenie maksimum na płaszczyźnie funkcji celu, 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, wagи sieci są modyfikowane, a cała operacja powtarzana jest aż do momentu zakończenia treningu.

Raul Rojas w swojej książce [12] 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:

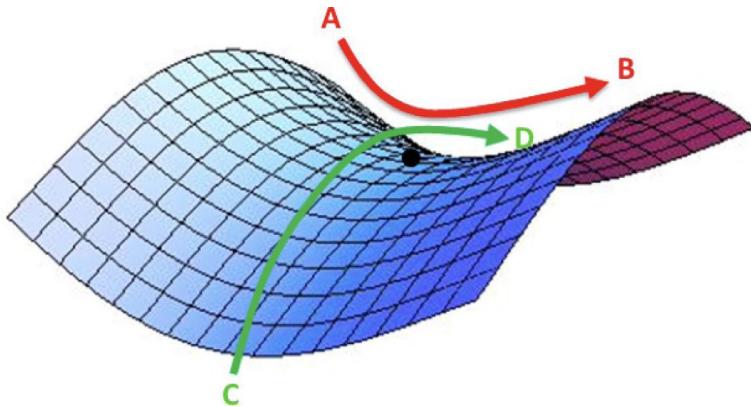
Przekazywanie: wejście x jest podawane do sieci. Funkcje aktywacji i ich pochodne są oceniane w każdym węźle (neuronie) sieci. Pochodne są przechowywane.

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 zmagazynowaną w lewej części jednostki. Rezultat jest transmitowany na lewo od jednostki. Wynik zgromadzony w jednostce wejściowej 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 zwiększająca 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. Pozostaże także niebezpieczeństwo związane z potencjalnym utknięciem modelu w jednym z minimów lokalnych, lub co gorsza w punkcie 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 algorytmem jak spadek gradientu, ponieważ wartość gradientu jest wówczas bliska零 we wszystkich wymiarach.



Rysunek 4.3: Przykładowy kształt punktu siodłowego

W odpowiedzi na opisane problemy opracowana została technika tak zwanego pędu [13]. Ogranicza ona oscylacje spadku gradientu w niewłaściwych kierunkach i przyspiesza proces zbiegania rozwiązania. Pęd sprowadza się do dodawania ułamka γ 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 zauważył 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 [14]. 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 ∇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 4.2.

Tabela 4.2: Optymalizatory w *TorchFrame*

| Optymalizator | Opis |
|---------------|---|
| SGD [15] | <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 $\{x(i), y(i)\}$ jest zbiorem poszczególnych par treningowych. Stochastyczny spadek gradientu cechuje się częstymi 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. Możliwe jest również zastosowanie opisanych wcześniej mechanizmów pędu oraz algorytmu Nesterov'a w celu poprawy wydajności działania tego algorytmu.</p> |

| | |
|----------------------|--|
| AdaGrad [16] | <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. <i>AdaGrad</i> (ang. <i>Adaptive Gradient Algorithm</i>) nadaje się przez to do pracy z rzadkimi zbiorami treningowymi. Wzór prezentujący sposób aktualizacji pojedynczego i-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 $G_{t,ii}$ jest diagonalną macierzą zawierającą sumy kwadratów gradientów wyznaczonych aż do chwili t, $g_{t,i}$ jest bieżącą wartością gradientu dla parametru Θ_i, a ϵ oznacza współczynnik wygładzający, zabezpieczający przed ewentualnym dzieleniem przez 0.</p> <p>Wzór ten oznacza, że <i>AdaGrad</i> modyfikuje długość kroku uczącego w każdej iteracji t dla każdego parametru i bazując na przeszłych wartościach gradientów 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> |
| AdaDelta [17] | <p>Algorytm ten stanowi rozszerzenie optymalizatora <i>AdaGrad</i>. Rozwiązuje trapiący go problem zanikającego kroku treningowego poprzez wprowadzenie ograniczenia, co do ilości przeszłych gradientów mających wpływ na aktualizację wag w bieżącej iteracji.</p> <p>Dodatkowo, zamiast przechowywać określona liczbę przeszłych kwadratów gradientów, <i>AdaDelta</i> (ang. <i>ang. Adaptive Delta</i>) wyznacza ich sumę poprzez rekursywne definiowanie zanikającej średniej wartości przeszłych kwadratów tych gradientów.</p> <p>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 $E[g^2]_t$ oznacza średnią wartość sumy kwadratów gradientów w bieżącej iteracji, $E[g^2]_{t-1}$ definiuje tę średnią dla poprzedniego kroku, a g_t^2 stanowi kwadrat bieżącej wartości gradientu. Parametr γ pełni tutaj podobną rolę, co w przypadku mechanizmu pędu.</p> <p>Ostatecznie parametry modelu aktualizowane są zgodnie z następującą formułą:</p> $\Delta\Theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$ |

| | |
|------------------|---|
| Adam [18] | <p>Optymalizator <i>Adam</i> (ang. <i>Adaptive Moment Estimation</i>), podobnie jak <i>Ada-Delta</i>, pozwala wyznaczać adaptacyjne długości kroku uczącego dla każdego parametru sieci. Przechowuje on eksponencjalnie zanikającą średnią wartość kwadratów przeszłych gradientów v_t. Poza tym zapisuje również eksponencjalnie zanikającą średnią przeszłych gradientów m_t, podobną do wartości pędu.</p> <p>Wartości m_t oraz v_t są szacunkami odpowiednio pierwszego momentu (średnia) oraz drugiego momentu (wariancja niecentrowana), stąd nazwa metody.</p> <p>Autorzy <i>Adama</i> zauważyli, że m_t oraz v_t, które inicjowane są jako wektory zerowe, dążą tendencyjnie do zera, szczególnie 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 β_1, β_2 oraz ϵ mogą być dobierane eksperymentalnie, jednak rekomendowane jest przyjęcie następujących liczb: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.</p> |
|------------------|---|

Wybór odpowiedniego algorytmu optymalizacji nie jest prostym zagadnieniem. Ostateczną decyzję powinna poprzedzać matematyczna analiza postawionego problemu lub eksperymentalne porównanie dostępnych możliwości. Należy pamiętać, że czynnik ten ma często kluczowy wpływ na uzyskiwane rezultaty. W przypadku prostych zadań optymalizacji najczęściej, choć nie zawsze, wystarczające powinny okazać się bazowe algorytmy, takie jak *SGD* (ang. *Stochastic Gradient Descent*). Z kolei bardziej złożone problemy z reguły wymagać będą zastosowania metod adaptacyjnych takich jak *Adam*, czy *AdaGrad*. Często oferują one lepsze rezultaty, jednak za cenę większej złożoności obliczeniowej. Wiele zależy również od samego kontekstu użycia. Przykładowo w przypadku douczania sieci neuronowej w celu poprawy ostatecznych rezultatów, zalecane jest użycie algorytmów ze stałą długością kroku treningowego, które w odróżnieniu od algorytmów adaptacyjnych mają w takich przypadkach mniejszą tendencję do destabilizacji rozwiązania.

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

Jest to istotny element pozwalający dynamicznie dostosowywać krok do aktualnego położenia modelu na hiperpłaszczyźnie 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 rozpoczęcie się trening sieci neuronowej.
6. ***Training epochs*** - Liczba epok, jaką przejdzie model w danej sesji treningowej. W każdej eoce 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 liczbę pojedynczych próbek ze zbioru treningowego, które zostaną zgrupowane 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 przetasowane, 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 którykolwiek ścieżki spowoduje, że powiązany z nią 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.

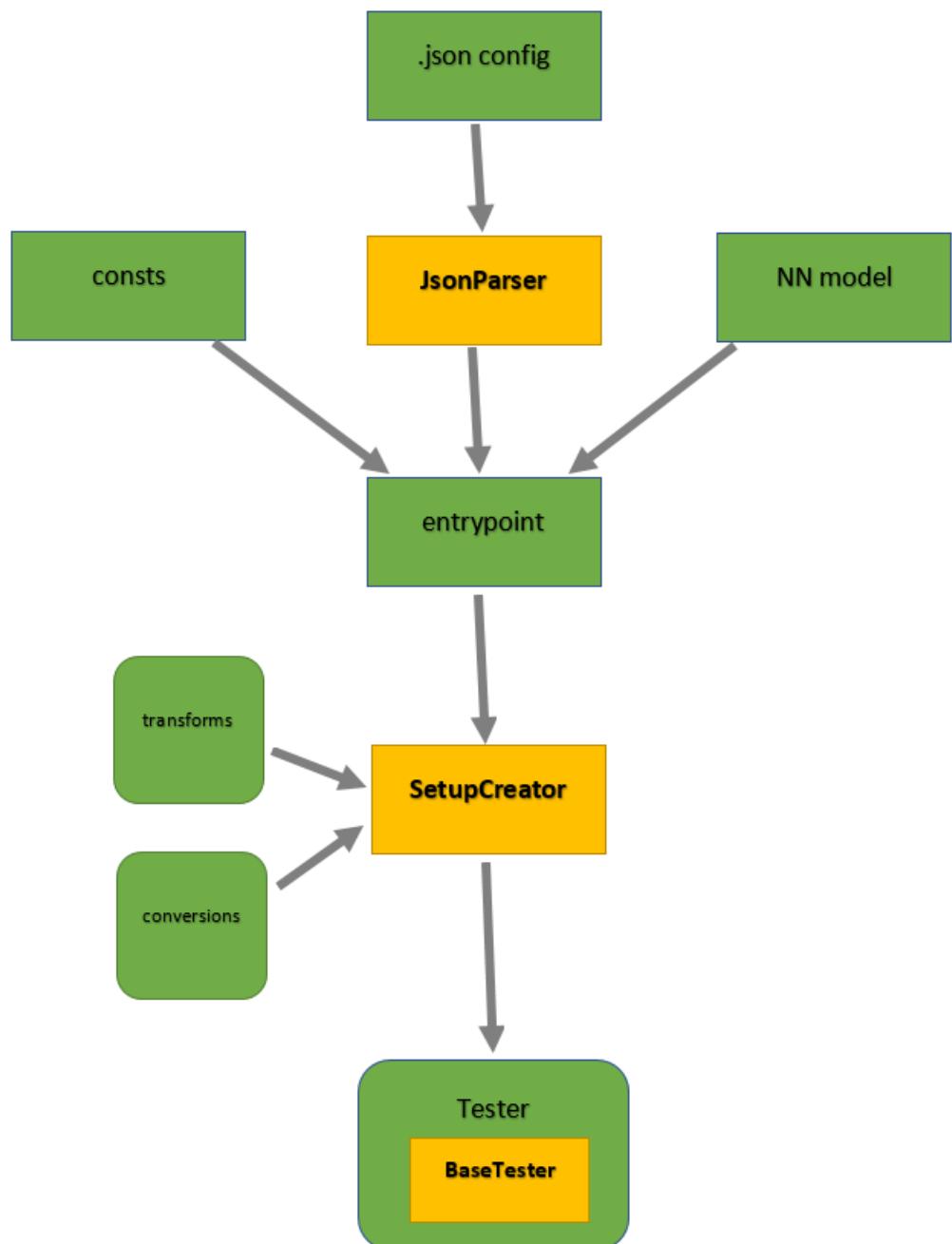
4.1.4. Testowanie w *TorchFrame*

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 odpowiednim szkielecie, 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 szумy 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 spróbowią 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* (ang. Open Source Computer Vision Library), 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* ang. *Stanford Artificial Intelligence Laboratory*). 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.

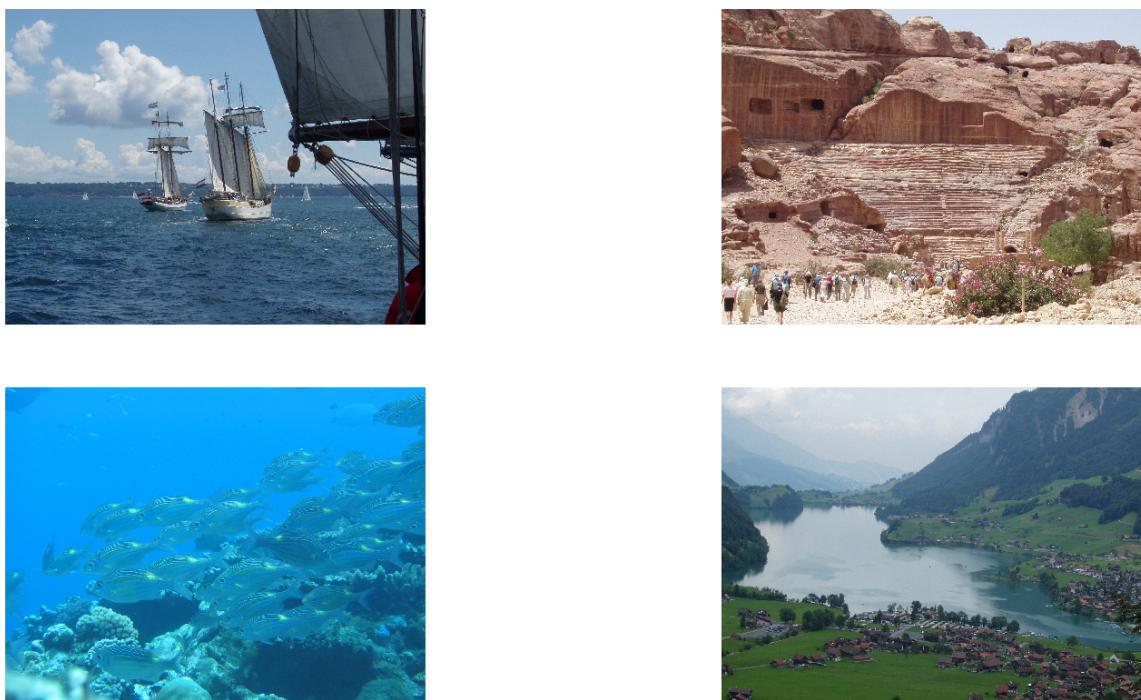
Na klasyczny filtr Sobela-Feldmana składają się dwie maski:

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

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

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 wchodzącyymi 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

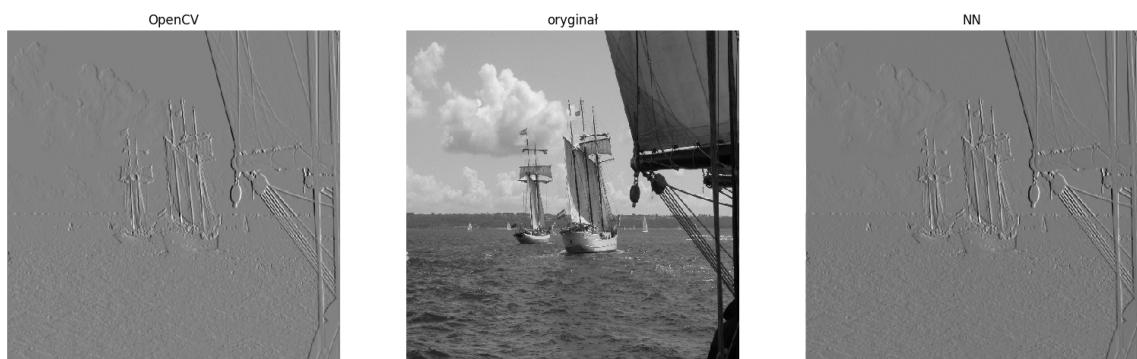
Aby mogły właściwie pełnić swoją rolę, obrazy treningowe zostały w pierwszej kolejności podane odpowiedniemu przetworzeniu wstępemu. W ramach tego procesu rozdzielnosc każdego zdjęcia zmniejszona została do wymiarów 256x256 pikseli, a wartości kolorów przeskalowane do zakresu $< 0,1 >$ w celu usprawnienia 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.



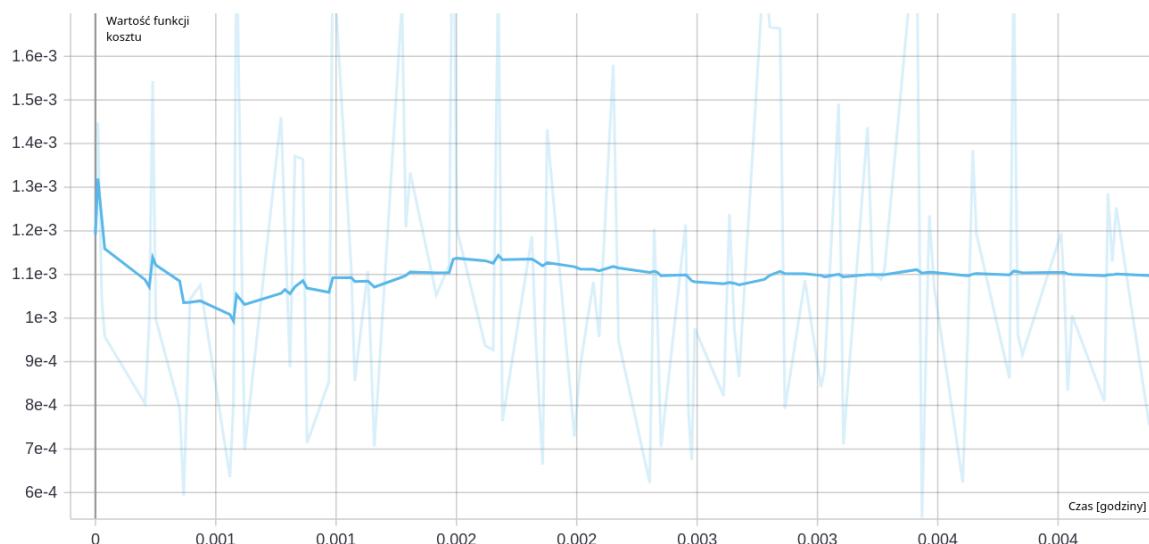
Rysunek 4.6: Działanie filtru Sobela

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. Poprawność rezultatów 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 zauważyc można, że sieć neuronowa właściwie odtworzyła panujące w niej proporcje. Odpowiednio mniejszy rząd wielkości odzwierciedla normalizację danych na których uczyony był model. Środkowa kolumna składa się w całości z wartości bliskich zeru. 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 przełożyło 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 rozwiązań 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 wstępnego przetworzenia danych treningowych, między innymi wspomniane już przeskalowanie wartości pikseli do przedziału $< 0,1 >$, 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 sepii 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 użytą 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 filtra Sobela, opisanego w poprzednim punkcie podrozdziału. Ponownie, w ramach wstępniego przetwarzania danych, wartości pikseli przeskalowane zostały do przedziału $< 0,1 >$, a każdy obraz przeskalowany do wymiarów 256×256 . W przypadku obrazów referencyjnych zastosowany został oczywiście filtr zapewniający efekt sepii 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 filtra Sobela jest fakt, że sieć trenowana była na obrazach posiadających pełny zakres trzech kanałów kolorystycznych RGB (ang. Red Green Blue), 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 filtra Sobela. Świadczy to przede wszystkim o uniwersalności algorytmu *Adam* w odnajdywaniu optymalnych rozwiązań w procesie minimalizacji. Rezultaty pracy wytrenowanego modelu przedstawione zostały na Rysunku 4.8.

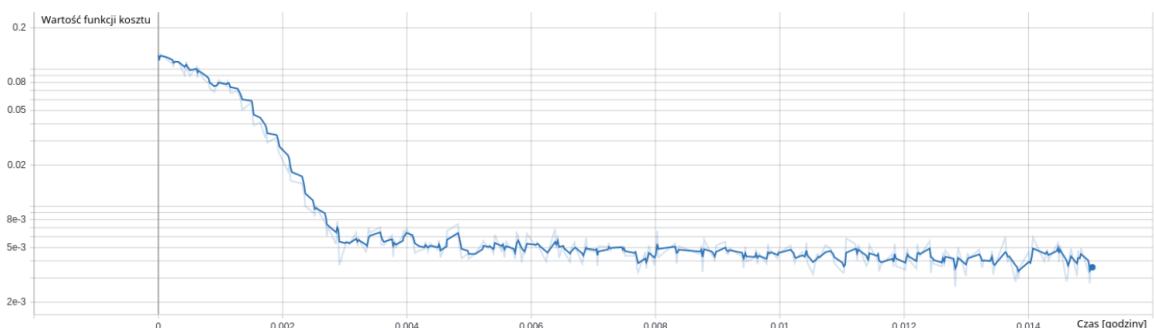


Rysunek 4.8: Działanie sepii

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 pozorne niewielki, nie pozwala jednoznacznie przełożyć uzyskanych parametrów modelu na referencyjną maskę G_x . Każdy z trzech zastosowanych filtrów przetwarza obraz za pomocą macierzy 3×3 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.

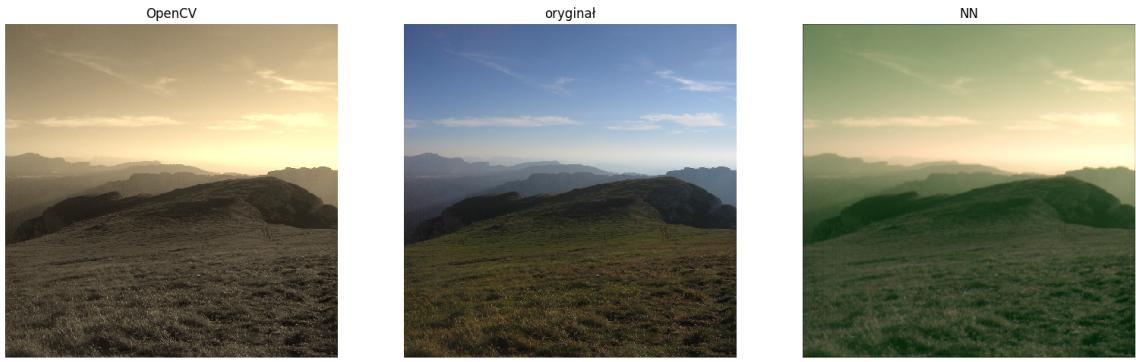
W takiej sytuacji, poza wizualną oceną działania sieci, przydatnych informacji dostarcza wykres wartości funkcji kosztu przedstawiony na Rysunku 4.9.



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 doboru 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

Filtryle górnoprzepustowe używane są w celu uwypuklenia szczegółów występujących na obrazie. Tłumią one elementy o niskiej częstotliwości, a wzmacniają te cechujące się częstotliwościami wysokimi poprzez zwiększenie ich jasności lub barwy. Efektem zastosowania filtru górnoprzepustowego jest najczęściej zwiększenie kontrastu poprzez podkreślenie ostrych krawędzi obiektów. W pracy "*Image Enhancement Techniques using Highpass and Lowpass Filters*" [20] znaleźć można następujący opis:

'Filtr górnoprzepustowy to filtr, który dobrze przepuszcza wysokie częstotliwości, ale tłumi częstotliwości niższe niż częstotliwość graniczna. Ostrzenie jest zasadniczo operacją górnoprzepustową w dziedzinie częstotliwości.'

Istnieje kilka standardowych form filtrów górnoprzepustowych, takich jak filtr Butterworth'a, czy Gauss'a. Wszystkie filtry górnoprzepustowe (H_{hp}) są zazwyczaj reprezentowane poprzez ich relację z filtrami dolnoprzepustowymi (H_{lp}):

$$H_{hp} = 1 - H_{lp}.$$

W ramach niniejszego eksperymentu zastosowany został filtr górnoprzepustowy reprezentowany przez maskę następującej postaci:

$$G_x = \begin{bmatrix} -1 & -1 & -1 \\ -1 & +9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

W celu odwzorowania działania filtru zastosowana została sztuczna sieć splotowa złożona z trzech neuronów, podobnie jak miało to miejsce w przypadku sepii. Model sieci uczyony był na tych samych danych, z tą różnicą, że dane referencyjne zostały przetworzone z wykorzystaniem filtru górnoprzepustowego G_x .

Wyniki działania wytrenowanego modelu przedstawia Rysunek 4.11.



Rysunek 4.11: Działanie filtru górnoprzepustowego

Parametry dla których uzyskany został przedstawiony efekt są następujące:

- Funkcja kosztu: *SmoothL1Loss*
- Optymalizator: *Adam*
- Funkcja aktywacji: *ReLU*
- Ilość epok treningowych: 5
- Rozmiar pakietu danych: 2

Na uwagę zasługuje tutaj przede wszystkim rozmiar pakietu danych odróżniający ten zbiór hiperparametrów od dwóch poprzednich. Został on zmniejszony w celu ograniczenia wpływu uśrednienia wartości funkcji kosztu na modyfikację wag modelu. Pomimo tego zabiegu wyraźnie widać, że w ramach wszystkich przeprowadzonych eksperymentów to właśnie efekt działania sieci górnoprzepustowej najbardziej odbiega od obrazu referencyjnego. Jest to spowodowane charakterem działania badanego filtru. Jego celem jest uwypuklanie pojedynczych elementów obrazu, takich jak krawędzie, co stoi w niejakkiej sprzeczności z procesem treningu, którego celem jest osiągnięcie jak najmniejszej wartości błędu w obrębie wszystkich danych treningowych. Oznacza to, że sieć będzie dążyła do jak najlepszego uśrednienia procesu filtracji górnoprzepustowej. Efekt ten został niewątpliwie osiągnięty. Obraz wygenerowany przez model cechuje się wyraźniejszymi konturami, jednak nie w takim stopniu jak obraz docelowy.

Wykres wartości funkcji kosztu przedstawia Rysunek 4.12.



Rysunek 4.12: Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych

Podobnie jak w dwóch poprzednich przypadkach, początkowy spadek wartości prowadzi model do punktu optymalnego rozwiązania. W punkcie tym algorytm optymalizacji zaczyna nieznacznie oscylować podejmując kolejne próby poprawy uzyskanych wyników.

W ramach testów różnych rozwiązań na uwagę zasługują rezultaty osiągnięte z wykorzystaniem optymalizatora *AdaDelta* przedstawione na Rysunku 4.13.



Rysunek 4.13: Przykład działania sieci w minimum lokalnym

Wygenerowany obraz został wyostrzony w stosunku do oryginału, jednak nieznacznie wypaczone zostały przy tym kolory tła, co jest zjawiskiem niekorzystnym i może wskazywać na osiągnięcie przez model jednego z lokalnych minimów funkcji celu.

4.2.4. Podsumowanie

Przeprowadzone w ramach tego rozdziału eksperymenty dowodzą, że sztuczne sieci splotowe z powodzeniem mogą posłużyć w procesie filtrowania obrazów. Należy zadać sobie jednak pytanie, czy rezultaty ich działania są w stanie zrekompensować stosunkowo czasochłonny proces treningu związany z odpowiednią obróbką danych oraz eksperymentalnym doborem właściwych hiperparametrów.

W przypadku prostych filtrów, jak sepia czy filtr górnoprzepustowy korzystniejsze okazuje się zastosowanie klasycznych metod przetwarzania w postaci odpowiednich masek konwolucyjnie nakładanych na docelowe obrazy. Rozwiązań te są często pozbawione typowej dla sieci neuronowych tendencji do uśredniania wyników, co przekłada się na lepsze rezultaty w sytuacjach, w których celem jest na przykład osiągnięcie dużego kontrastu przetwarzanych obrazów.

Nie oznacza to jednak, że sieci neuronowe mogą zostać zastąpione w każdej sytuacji. Ich zdolności adaptacyjne sprawiają, że znajdują one zastosowanie w rozwiązaniach, w których klasyczne metody zawodzą. Dowodem na to będą kolejne rozdziały tej pracy.

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 występujących na nich obiektów bazując jedynie na treści tych zdjęć oraz swoim doświadczeniu. Można z tego wywnioskować, że zdjęcia te zawierają informacje wystarczające do oszacowania prawdopodobnych kolorów. Pozwala to założyć, że do tego zagadnienia można skutecznie wykorzystać konwolucyjne sieci neuronowe, które cechują się niezwykłą umiejętnością 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 przeważnie oczywiste barwy, niebo jest zazwyczaj niebieskie, a trawa zielona, to są też powierzchnie, które posiadają szeroki wachlarz możliwych kolorów. Samochód, na przykład, 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.

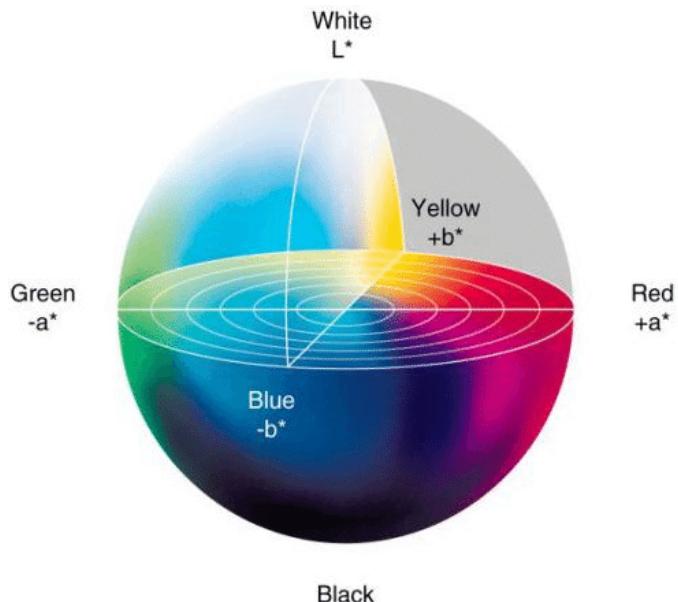
Aby zwiększyć efektywność uczenia wykorzystano przestrzeń barw CIELab. W przestrzeni tej barwę obrazu opisują 3 składowe:

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

Przestrzeń barw CIELab została przedstawiona na Rysunku 4.14.

Zaletą zastosowania CIELab jest fakt, że jest to najbardziej równomierna przestrzeń 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 model odtwarza składowe *A* oraz *B*, które reprezentują przewidziane kolory dla obrazu wejściowego. W celu lepszego zrozumienia formatu przykładowa składowa *L* została przedstawiona na Rysunku 4.15. Jak widać, zawiera ona informacje wystarczające do rozróżnienia między sobą powierzchni oraz wydobycia ich kluczowych cech.



Rysunek 4.14: Przestrzeń barw CIELab.



Rysunek 4.15: Przykładowa składowa L .

Jako rozwiązanie podanej problematyki wpierw oceniony został model autorski. Z jego użyciem przeprowadzone zostało porównanie skuteczności różnych konfiguracji, w których uczona była sieć. Do elementów poddanych testom należą algorytm optymalizacyjny, funkcja straty, funkcja aktywacji oraz sposób przetwarzania wstępnego danych treningowych.

Zarówno trening, jak i testy modelu autorskiego odbywały się z użyciem frameworka *Torch-Frame* opisanego w podrozdziale 4.1.

4.3.2. Model autorski

W ramach rozwiązania autorskiego opracowany został model 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 te muszą mieć takie same wymiary jak składowa wejściowa, co oznacza, że kluczowym było odpowiednie dobranie parametrów takich jak *padding* (pol. otoczka), *stride* (pol. krok) oraz wielkość filtrów. Pełna architektura sieci została przedstawiona w Tabeli 4.3.

Tabela 4.3: Architektura modelu autorskiego.

| 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 |

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 3x3, tak więc, aby zachować niezmieniony 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 powiązania 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 zdolności analityczne sieci.

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 warstwę z wielkością filtra 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 filtra 1x1. Odpowiadają one warstwom gęstym i mają na celu przekonwertowanie wartości funkcji aktywacji z poprzednich warstw na wartości kolorów odpowiednich 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 [21]. 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.

Problemy te zostały zauważone i opisane już w 1994 roku przez Y. Bengio oraz jego współpracowników [22]. 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 zmala do wartości, które uniemożliwiają modelowi dalsze uczenie się. Z kolei 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 tych warstw 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 wagę początkowe modelu.

Użycie tych warstw w modelu autorskim tuż za warstwami ReLU pozwoliło uzyskać bardziej korzystną zbięźność modelu oraz lepsze rezultaty końcowe. Ocenione zostało też rozwiązanie, 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 ostateczną wersją modelu autorskiego sprawdzona została skuteczność zastosowania warstw Dropout (pol. algorytm odrzucania) [23]. Autorzy tej techniki opisują ją następująco:

'Technika ta przeciwdziała efektowi przeuczenia się sieci oraz zapewnia wydajny sposób łączenia wielu różnych architektur sieci neuronowych. Termin "odrzucenie" nawiązuje do odrzucania neuronów (widocznych i ukrytych) z sieci neuronowej.'

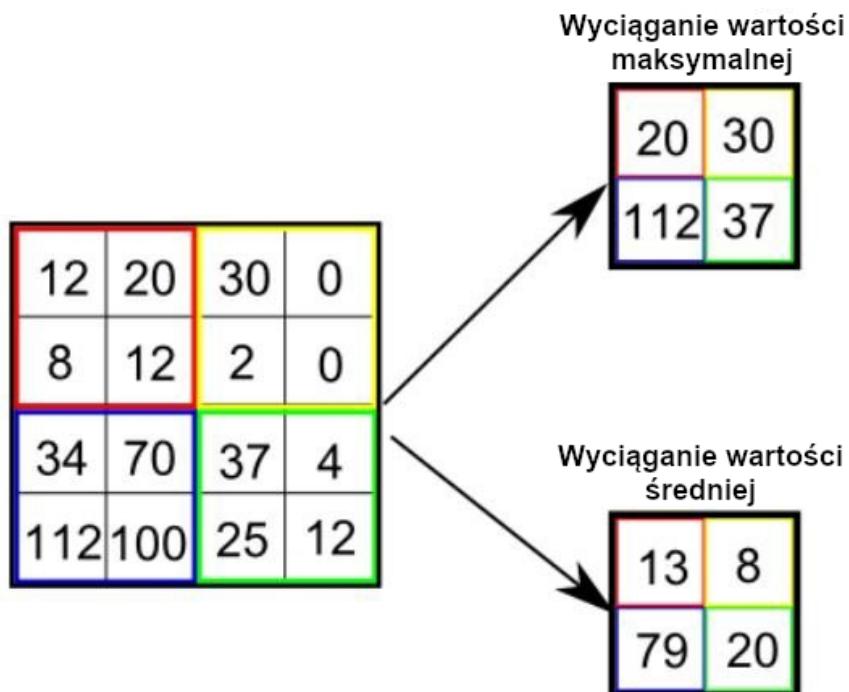
Warstwy te w trakcie treningu dezaktywują część neuronów w uczonej sieci neuronowej, co jest równoznaczne ze wstrzymaniem aktualizacji ich wag w danej iteracji treningu. Wybór neuronów do dezaktywowania odbywa się z pewnym prawdopodobieństwem określonym podczas inicjalizacji modelu. Dezaktywowanie neuronów, zwane również odrzuceniem, można interpretować jak przerwanie tymczasowo wszystkich połączeń danego neuronu, zarówno wejściowych jak i wyjściowych. Takie działania są równoznaczne z wyselekcjonowaniem z modelu mniejszej sieci i trenowaniem 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 działania sieci. W przypadku danej problematyki oraz obranego do jej rozwiązania podejścia, 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 dla różnych wartości parametru p (prawdopodobieństwo dezaktywowania dla każdego neuronu sieci), wraz z rozważeniem pozostałych czynników wpływających na efektywność sieci, znajduje się w punkcie 4.3.14 związanym z rezultatami modelu autorskiego.

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 zredukowanie przestrzennej wielkości reprezentacji cech wyciągniętych z obrazu poprzez wyciągnięcie najbardziej istotnych wartości funkcji aktywacji z określonych obszarów reprezentacji. Celem tego działania jest zredukowanie rozmiaru sieci, a co za tym idzie, zmniejszenie ilości obliczeń koniecznych do wykonania przez sieć.

Ponadto pooling wspomaga adaptację modelu do zmiennego położenia kluczowych wzorów rozpoznawanych przez sieć na obrazie wejściowym. Cechą ta zwana jest niezmiennością od translacji (ang. translation invariance). Jest ona rezultatem dokonywania operacji, takich jak wyliczanie wartości maksymalnej z poszczególnych obszarów. Dzięki temu zmienne położenie wartości selekcjonowanej, 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 operacje poolingu zostały przedstawione na Rysunku 4.16.

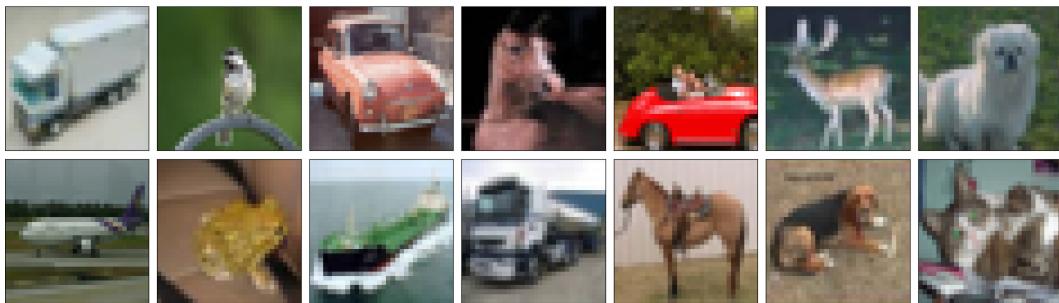


Rysunek 4.16: Przykładowe operacje warstwy poolingu

W modelu autorskim warstwy poolingu nie zostały zastosowane, aby uniknąć utraty kluczowych informacji przestrzennych koniecznych do właściwego wygenerowania prawdopodobnych barw obrazu.

4.3.6. Wykorzystywany zbiór treningowy

Do uczenia modelu został wykorzystany zbiór danych CIFAR-10 stworzony przez A. Krizhevsky oraz zaprezentowany w 2009 roku [24]. Składa się on z 60000 obrazów w przestrzeni kolorów RGB o rozdzielcości 32×32 piksele. Spośród tych obrazów, 10000 zostało wykorzystanych jako zbiór walidacyjny do śledzenia skuteczności treningu modelu. Przykładowe obrazy ze zbioru danych zostały przedstawione na Rysunku 4.17.



Rysunek 4.17: 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, 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 umiejętność generalizacji modelu, jednakże obrazy z CIFAR-10 przedstawiają zróżnicowane otoczenia zawierające obiekty nie kwalifikujące się do żadnej klasy, co powinno umożliwić osiągnięcie wysokiej uniwersalności rozwiązania.

4.3.7. Przetwarzanie wstępne danych

Przed podaniem na wejście sieci obrazy uczące były wpierw poddawane przetwarzaniu wstępemu mającemu na celu doprowadzenie 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, może wystąpić duża różnica w tempie aktualizacji wag sieci. Niektóre wagie będą modyfikowane 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 badanego rozwiązania przetestowane zostały różne metody przetwarzania wstępnego zarówno danych wejściowych, jak i pochodzącej 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 filtru 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 wstępniego 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 wstępniego 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 modelu. Jeśli podczas treningu z nadzorem pożądane wyjście było w pewien sposób przetworzone, to sieć uczy się odtwarzać to wyjście przetworzone w taki sam sposób. 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 wyjście należy zdenormalizować, aby uzyskać oczekiwany rezultat. Jednakże, w ramach opracowanego rozwiązania, zaproponowana została metoda alternatywna.

Polega ona na uwydatnianiu kolorów wygenerowanych przez sieć dla wysokich wartości naświetlenia - 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, ale dla danego piksela jego przedział jest dodatkowo 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 rozciąga się od -127 do 128, a składowej L od 0 do 100. W pierwszym 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 wartości pikseli składowych A i B są dzielone przez swoje odpowiednie, maksymalne wartości znalezione w kroku poprzednim i rozciągane na cały dostępny przedział poprzez 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 , co oznacza, 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 dostępnego przedziału i równa się 63,5.

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

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

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

Formułę konwertowania 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 dwuwymiarową 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 zadowalające 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 przytumiona, jako że brak naświetlenia ogranicza natężenie kolorów.

4.3.9. Augmentacja danych

W celu skutecznego uczenia modelu potrzebny jest odpowiednio duży i różnorodny zbiór treningowy, składający się, w przypadku tego rozwiązania, wyłącznie z obrazów. W obliczu problemu nie wystarczającej ilości danych stosuje się metody zwane augmentacją danych pozwalające poszerzyć zbiór uczący poprzez dodawanie nowych informacji do danych będących podstawą zbioru, tworząc w ten sposób nowe dane, które mogą być wykorzystane w treningu. Augmentacja danych przeciwdziała uczeniu się przez sieć nieistotnych wzorców 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 wyższym 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 oddalanie 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 poszerzą 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 autorskiego 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 liczbę 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 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 straty przetestowane zostały następujące możliwości.

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 bezwzględny, dany 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** - odmiana L1Loss przedstawiona w 2015 roku przez R. Girshick [25], opisana funk-

cja:

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

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 autorskiego 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:

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

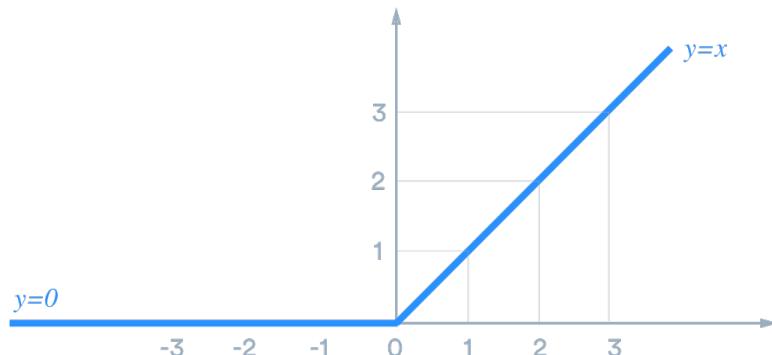
Gdzie:

- 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 piksela obrazu o współrzędnych (i, j) .

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

4.3.11. Funkcje aktywacji

W wyborze odpowiedniej funkcji aktywacji kierowano się badaniami przeprowadzonymi przez B. Xu oraz jego współpracowników [26]. Testowali oni skuteczność takich funkcji jak ReLU, Leaky ReLU (pol. przepuszczające ReLU), PReLU (ang. Parametric ReLU) oraz RReLU (ang. Randomized Leaky ReLU) w zagadnienniu klasyfikacji treści obrazów. Bazując na otrzymanych przez nich rezultatach zdecydowano się wykorzystać w rozważanej problematyce funkcję aktywacji ReLU, przedstawioną na Rysunku 4.18



Rysunek 4.18: Funkcja aktywacji ReLU

Funkcja ta posiada wiele zalet, takich jak niska złożoność obliczeniowa oraz przyspieszenie procesu zbieganie się stanu sieci do stanu pożądanego poprzez brak ograniczeń na maksymalną wartość funkcji. 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 blokowania się procesu uczenia w sytuacji, w której wag modelu dojdą do stanu o wartości aktywacji zbliżonej 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, pozostało jednak 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 sieci. 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*
- *AdaGrad*
- *SGD*

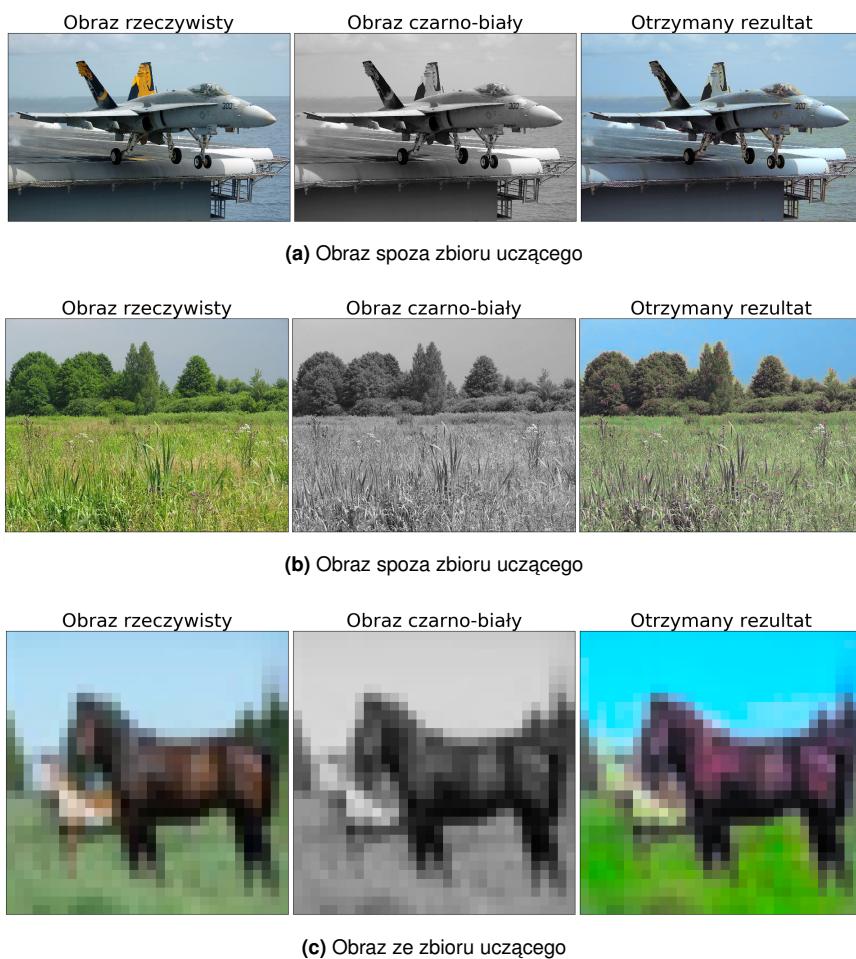
Przeprowadzone zostały także poszukiwania najbardziej odpowiednich parametrów dla wymienionych algorytmów w celu osiągnięcia jak największej ich skuteczności w rozwiązyaniu rozważanej problematyki.

4.3.13. Trening

Model trenowany był paczkami o wielkości 128 obrazów, cała epoka składała się z 50000 obrazów. Przed każdą epoką zbiór treningowy był przetasowywany, aby przeciwdziałać przeuczaniu się sieci. W celu znalezienia najbardziej zadawalającego rozwiązania przetestowane zostały różne konfiguracje treningowe. Możliwe zmienne elementy tych 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 przetestowanych 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 4.19

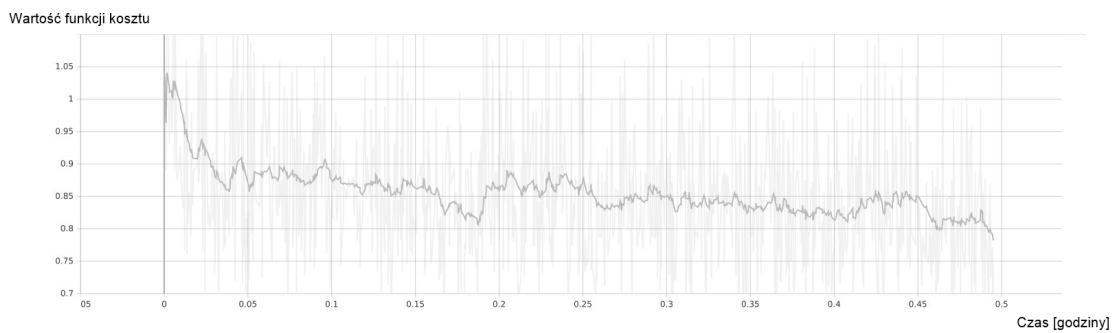


Rysunek 4.19: Efekt działania modelu autorskiego.

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.
- Brak rozmycia Gaussowskiego.
- Zastosowanie warstwy BatchNorm oraz brak warstwy Dropout.
- Zastosowanie zaproponowanego algorytmu przetwarzania końcowego, opisanego w punkcie 4.3.8.

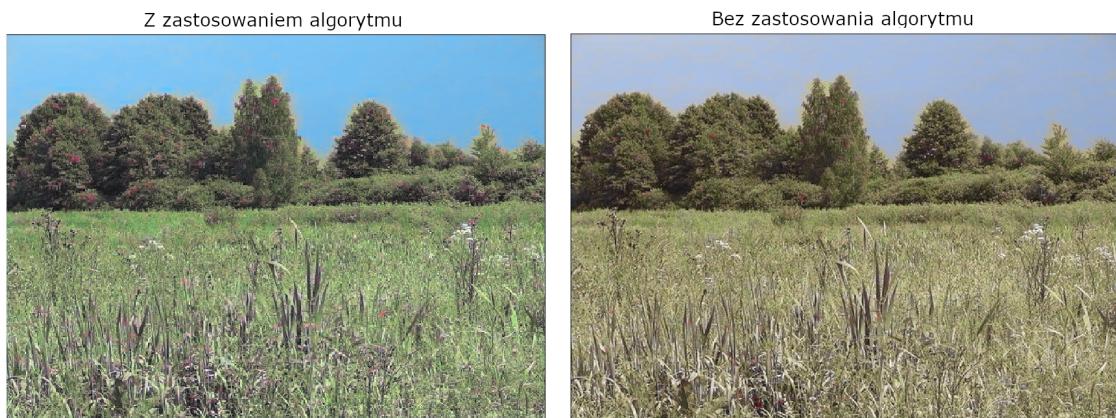
Na podstawie obrazów na Rysunku 4.19 oraz przeprowadzonych testów można wywnioskować, że model nauczył się najlepiej kolorować powierzchnie o zazwyczaj stałej barwie, takie jak niebo albo trawa. Widać, że ich barwy są wyjątkowo intensywne oraz jednoznacznie wskazują na rodzaj powierzchni. Gorzej rzeczą 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 przeodczącymi często w odcieniu 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 4.19. Wykres uczenia modelu w przedstawionej konfiguracji został zaprezentowany na Rysunku 4.20. Można zaobserwować stopniowy spadek wartości funkcji kosztu, co oznacza poprawne zbieganie się modelu do właściwego rozwiązania. W końcowych fazach wykresu widać, że wartość błędu zmienia się o bardzo małe wartości, co znaczy, że proces uczenia wpadł już w minimum będące jednym z dostępnych rozwiązań.



Rysunek 4.20: Wykres uczenia modelu autorskiego

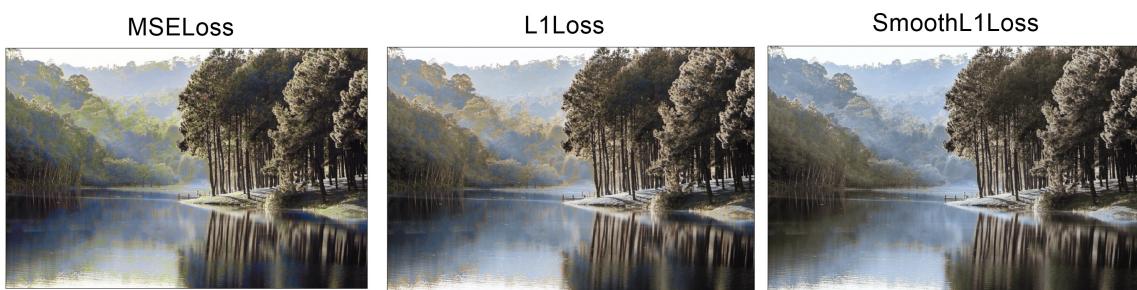
W dalszej części sekcji opisany został wpływ poszczególnych elementów konfiguracji.

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 4.21. Wadami tej metody są generowane momentami zbyt intensywne barwy obrazy oraz bardziej widoczne, czasami zdarzające się "wycieki" kolorów jednych powierzchni na inne, przykładowo czerwony kolor z samochodu wychodzi poza powierzchnię pojazdu i barwi fragmenty trawy przylegające do niego 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 4.21 widać to poprzez drobne obszary żółci na granicy nieba i drzew.



Rysunek 4.21: Efekt zastosowania algorytmu przetwarzania końcowego

Kolejnym ważnym komponentem jest użyta funkcja kosztu. Właściwy wybór znacznie poprawia proces aktualizowania 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 4.22 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 4.22: 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. Skutkuje to 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 4.22.

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, aby przeciwdziałać uśrednianiu wartości barw przez sieć. Jednakże efekt ten nie został uzyskany, a przeciwdziałanie elementom odstającym przez SmoothL1Loss dodatkowo negatywnie wpłynęło na kolory generowane przez model. 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 tych 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 zielonego, co wyraźnie widać na Rysunku 4.22.

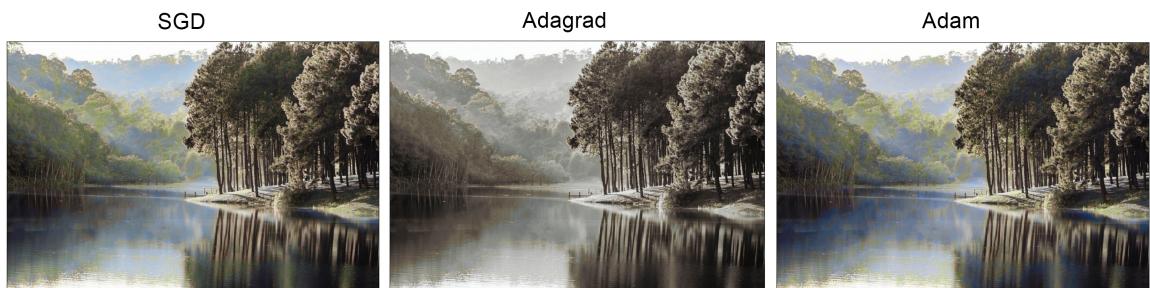
MSELoss: W związku z brakiem odpowiedniego algorytmu, który wstrzymuje karanie modelu w sytuacji, gdy generowane kolory są niezgodne z rzeczywistymi, ale prawdopodobne dla kolorowanej powierzchni, funkcja MSELoss również ma tendencję do uśredniania generowanych kolorów jak funkcja L1Loss. Jednakże większa czułość funkcji na wysokie wartości błędów, powodowana podnoszeniem do kwadratu różnicy pomiędzy wartością przewidzianą a rzeczywistą, umożliwiła wytrenowanie modelu zdolnego do generowania barw z szerokiego zakresu wartości, co korzystnie wpłynęło na rezultaty. Pokolorowany obraz, wygenerowany przez model uczony z użyciem funkcji MSELoss prezentuje, się najlepiej spośród obrazów na Rysunku 4.22. Z tego powodu właśnie ta funkcja została wybrana w procesie uczenia modelu autorskiego.

Wybór odpowiedniego algorytmu optymalizacyjnego może znacznie poprawić rezultaty modelu poprzez odpowiednie nakierowanie jego wag, aby wartość straty wyliczana przez funkcję kosztu była jak najmniejsza. Minimalizowanie wartości straty jest zagadniением złożonym, w trakcie poszukiwania optimum dla wag sieci występują liczne minima lokalne, które utrudniają odnalezienie minimum globalnego, zapewniającego najbardziej satysfakcjonujące wyniki. Aby przeciwdziałać zatrzymaniu się procesu uczenia w lokalnych minimach należy zastosować odpowiedni algorytm optymalizacyjny. Przykładowe obrazy pokolorowane przez model uczony z użyciem różnych algorytmów optymalizacyjnych zostały przedstawione na Rysunku 4.23

Dla algorytmu Adam zostały wybrane parametry: $\beta_1 = 0.9$, $\beta_2 = 0.999$, długość kroku treningowego = 0.1, współczynnik zaniku wag = $1e - 10$ oraz $eps = 1e - 08$.

Dla algorytmu AdaGrad zostały wybrane parametry: długość kroku treningowego = 0.1, współczynnik zaniku długości kroku treningowego = 0.999, współczynnik zaniku wag = $1e - 10$ oraz $eps = 1e - 10$.

Dla algorytmu SGD zostały wybrane parametry: długość kroku treningowego = 0.1, pęd = 0.9, tłumienie = 0 oraz współczynnik zaniku wag = 0. Zmianę długości kroku treningowego dla SGD nadzorował planista o parametrach: wielkość kroku = 1 oraz gamma = 0.999.



Rysunek 4.23: Porównanie rezultatów modeli uczonych z użyciem różnych algorytmów optymalizacyjnych

Jako algorytm optymalizacyjny dla finalnej wersji modelu autorskiego został wybrany Adam. Decyzja ta jest oparta na wielu kryteriach takich jak wartości końcowe funkcji kosztu oraz wizualne efekty działania sieci. Przykładowa ocena wizualna została opisana bazując na Rysunku 4.23. Porównując pokolorowane obrazy można zauważać, że najgorzej prezentuje się rezultat modelu stosujący algorytm AdaGrad. Na obrazie tym nie występuje wiele kolorów i dominują barwy szarości. Algorytm ten jest przeznaczony do pracy z rzadkimi zbiorami treningowymi, a zastosowany zbiór CIFAR-10 takim nie jest. Można więc podejrzewać, że podczas procesu uczenia wagi sieci na płaszczyźnie funkcji celu wpadły w minimum lokalne i nie były w stanie z niego wyjść, co poskutkowało pominięciem minimum globalnego będącego najlepszym możliwym rozwiązaniem. Analizując obrazy dla algorytmów Adam oraz SGD można zauważać, że są one znacznie lepiej pokolorowane niż obraz dla algorytmu AdaGrad. Można na nich wyraźnie wyróżnić roślinność pokolorowaną na zielono oraz taflę wody w odcieniach niebieskiego. Jednakże, pomimo znacznego podobieństwa pomiędzy tymi dwoma obrazami, można zauważać, że tafla wody jest znacznie bardziej wypełniona kolorem na obrazie dla algorytmu Adam, co świadczy o tym, że algorytm ten był w stanie zapewnić lepsze minimum dla wag sieci. Zagwarantowało to modelowi lepiej rozwiniętą zdolność predykcji kolorów dla różnorodnych powierzchni.

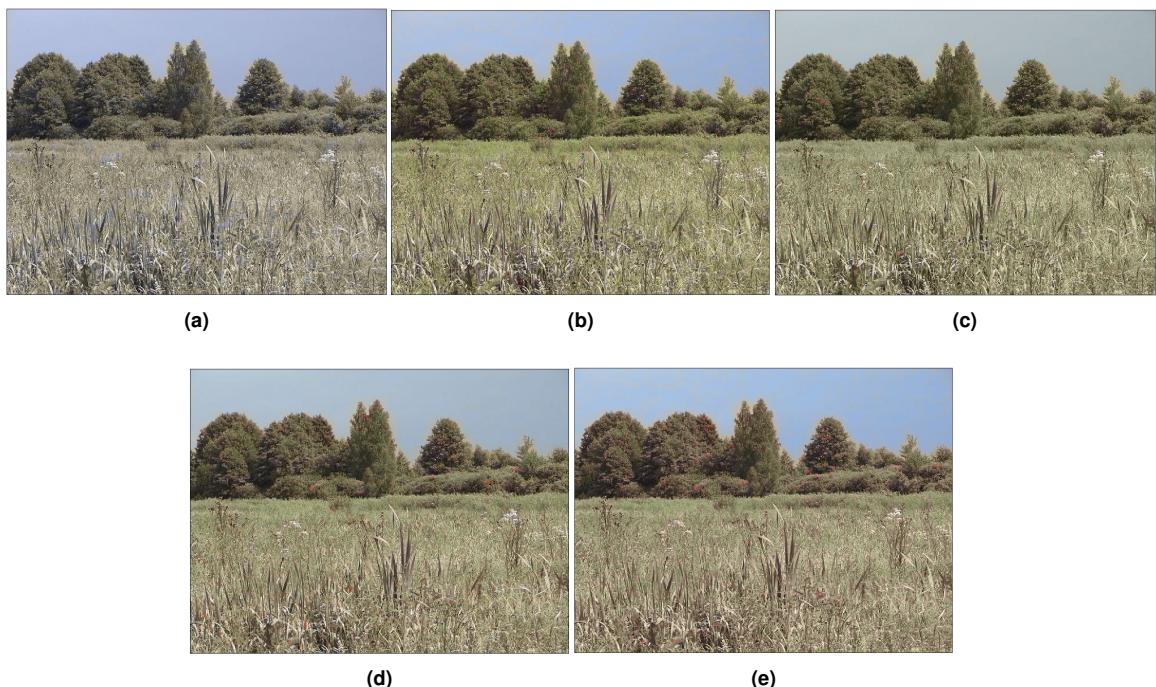
Z powodów opisanych w punkcie 4.3.4 warstwa Dropout nie została użyta w modelu autorskim. Na Rysunku 4.24 został pokazany wpływ zastosowania tej warstwy w modelu na kolorowane obrazy, w zależności od wartości parametru p (prawdopodobieństwo dezaktywowania dla każdego neuronu sieci).

Jak widać, zastosowanie tej warstwy nie przyniosło żadnych korzyści. Dla małych wartości parametru p barwy na kolorowanym obrazie są mniej żywe niż bez zastosowania Dropout, a dla większych wartości widać, że model traci w dużej mierze swoje zdolności do predykcji kolorów. Dowodzi to, że w przypadku rozważanej problematyki większym zagrożeniem jest utrata kluczowych informacji o obrazie niż niebezpieczeństwo przeuczenia sieci, co sprawia, że warstwa Dropout staje się niepożądana.



Rysunek 4.24: Wpływ zastosowania warstwy Dropout na kolorowanie obrazów

Duże znaczenie dla jakości treningu modelu miało także wybór metod przetwarzania wstępne- go, zarówno obrazów wejściowych, jak i obrazów referencyjnych. Przyczyny zastosowania takich za- biegów zostały dokładnie opisane w punkcie 4.3.7. Porównanie efektów zastosowania różnych metod połączonych w różne konfiguracje zostało umieszczone na Rysunku 4.25, a dokładny opis konfiguracji został umieszczony w Tabeli 4.4.



Rysunek 4.25: Skutki zastosowania różnych metod przetwarzania wstępnego

Tabela 4.4: Szczegóły konfiguracji przetwarzania wstępnego.

| Konfiguracja | Przetwarzanie danych wejściowych | Przetwarzanie danych referencyjnych |
|--------------|---|-------------------------------------|
| a | Normalizacja | Normalizacja |
| b | Normalizacja | Standaryzacja |
| c | Standaryzacja | Standaryzacja |
| d | Normalizacja oraz rozmycie gaussowskie | Standaryzacja |
| e | Standaryzacja oraz rozmycie gaussowskie | Standaryzacja |

Rezultaty przetwarzania wstępnego:

- (a) W pierwszej przetestowanej konfiguracji została zastosowana najprostsza metoda przetwarzania wstępnego - normalizacja. Po jej zastosowaniu wartości wejścia sieci (składowa L) oraz wartości referencyjne (rzeczywiste składowe A i B) były przeskalowane do przedziału od -0.5 do 0.5. Zabieg ten miał na celu usprawnić trening modelu z przyczyn opisanych w punkcie 4.3.7. Jednakże zauważać można, że rezultaty uzyskane przy tej konfiguracji prezentują się najgorzej w porównaniu z pozostałymi konfiguracjami. Barwy na obrazie są przytłumione oraz mało różnorodne. Dowodzi to, że w przypadku rozważanej problematyki najprostsze rozwiązanie jest niewystarczające.
- (b) W tym przypadku można zaobserwować znacznie lepsze rezultaty niż dla poprzedniej konfiguracji. Zastosowanie standaryzacji na danych referencyjnych znacznie usprawniło proces aktualizacji wag sieci w trakcie szukania minimum globalnego na płaszczyźnie funkcji celu. Generowane barwy są kontekstowo prawdopodobne dla kolorowanych powierzchni oraz cechują się wysokim stopniem nasycenia, co pozytywnie przekłada się na ich odbiór przez obserwatora. Ta konfiguracja została wybrana jako dająca najlepsze efekty z przedstawionych opcji.
- (c) Kierując się sukcesem standaryzacji w konfiguracji (b) ocenione zostały rezultaty zastosowanie tej metody również dla danych wejściowych. Niestety zabieg ten nie przyniósł oczekiwanej poprawy wyników modelu. Barwy na obrazie wyjściowym są mniej różnorodne oraz o niższej saturacji niż dla konfiguracji (b), ale za to o wyższej saturacji niż dla konfiguracji (a). Świadczy to, że w rozważanym zagadnieniu standaryzacja składowej luminacji nie wpływa pozytywnie na proces uczenia. Może to być spowodowane utratą kluczowych informacji o powiązaniach pomiędzy jasnością, a barwą, w przypadku przeskalowania składowej L , tak aby uzyskać zerową wartość średnią oraz jednostkowe odchylenie standardowe.

- (d) W tej konfiguracji została podjęta próba polepszenia rezultatów uzyskanych dla konfiguracji (b) poprzez zastosowanie dodatkowo rozmycia gaussowskiego na znormalizowanych danych wejściowych. Operacja ta miała na celu zmniejszyć wyrazistość detali na obrazie, aby uwydątnić kluczowe cechy charakterystyczne powierzchni identyfikowanych przez warstwy konwolucyjne. Dzięki temu trenowany model miał się nauczyć rozpoznawać prawidłowo powierzchnie bez względu na przesłaniające je drobne obiekty oraz zawarte w tych powierzchniach detale nie zmieniające ich rodzaju. Niepożądane było, aby model inaczej nauczył się kolorować niebo w zależności czy jego tle występuje jakiś drobny obiekt, taki jak na przykład ptak. Jednakże, analizując Rysunek 4.25, można dojść do wniosku, że nie przyniosło to oczekiwanych rezultatów. Wręcz przeciwnie, obniżyło to jakość kolorowania względem konfiguracji bez rozmycia gaussowskiego, co wyraźnie widać po mniej intensywnych barwach na obrazie.
- (e) Zastosowanie rozmycia gaussowskiego opisane dla konfiguracji (d) zostało także przetestowane w połączeniu z konfiguracją (c). W tym przypadku widać, że odniosło to pożądanego efektu. Generowane kolory są znacznie bardziej nasycone niż dla konfiguracji (c) oraz pasują kontekstowo do powierzchni, na którą zostały nałożone. Wynika z tego, że choć zastosowanie standaryzacji na składowej L może mieć niekorzystne skutki, to można je zredukować używając właśnie rozmycia gaussowskiego. Aczkolwiek, porównując wyniki tej konfiguracji z wynikami konfiguracji (b), widać, że nie jest ona konfiguracją najbardziej odpowiednią dla testowanego rozwiązania.

Bazując na efektach wizualnych różnych konfiguracji, do końcowej wersji modelu autorskiego została zastosowana konfiguracja (b) składająca się z operacji normalizacji składowej wejściowej sieci L oraz standaryzacji referencyjnych składowych A oraz B .

4.3.15. Model autorski - wnioski

Analizując uzyskane rezultaty dla modelu autorskiego, można stanowczo stwierdzić, że sieci FCN zdecydowanie nadają się jako rozwiązanie problematyki kolorowania czarno-białych obrazów. Ich niezwykłe zdolności do adaptacji oraz wyjątkowa biegłość w zapamiętywaniu i identyfikowaniu istotnych wzorców świetnie współgrają przy generowaniu prawdopodobnych barw pasujących do kolorowanych powierzchni. Jednakże przedstawione rozwiązanie nie jest idealne. Jego dużą wadą jest słabo rozwinięta umiejętności kolorowania przez model drobnych detali na obrazach oraz mała liczba rozpoznawanych przez sieć obiektów, spowodowana wyborem słabo zróżnicowanego zbioru uczącego. Aby zniwelować tę wadę należałyby znacząco zmodyfikować model oraz podejście do danej problematyki.

Jedna z potencjalnych modyfikacji mogłaby obejmować wprowadzenie segmentacji obrazu oraz klasyfikacji jego segmentów w celu zawężenia zakresu potencjalnych barw prawdopodobnych dla analizowanego fragmentu obrazu. Model w trakcie treningu uczyłby się możliwych kolorów oddzielnie dla wszystkich występujących na obrazach klas zamiast dla całego obrazu jako całości. Zmniejszyłoby to wpływ obiektów na siebie nawzajem i prawdopodobnie zwiększyło zdolności sieci do kolorowania pomniejszych detali występujących na obrazach. Ponadto umożliwiłoby to wykorzystanie większych oraz bardziej różnorodnych zbiorów treningowych, co zwiększyłoby uniwersalność tego rozwiązania. Jednakże takie podejście wymagałoby sieci o znacznie większej architekturze, co przekłada się na wyższe zapotrzebowanie na moc obliczeniową często ciężko dostępną bez wyspecjalizowanego sprzętu.

Pomimo niedoskonałości przedstawionego modelu autorskiego należy podkreślić jego użyteczność w półautomatycznym kolorowaniu czarno-białych obrazów. Jak można było zauważyć, zadowalająco radzi sobie on z kolorowaniem tła obrazów oraz powierzchni o mało różnorodnej kolorystyce. Można wykorzystać to do wstępnego pokolorowania czarno-białego obrazu, a następnie innymi metodami uzupełnić ubytki koloru oraz poprawić barwy wymagających tego detali. Takie podejście jest znacznie bardziej efektywne niż przykładowo kolorowanie obrazu ręcznie od początku do końca.

4.3.16. Model z wykorzystaniem przeniesienia uczenia

Olbrzymią zaletą sieci neuronowych, sprawiającą, że są one wyjątkowo uniwersalne, jest możliwość przenoszenia uczenia. Polega to na zastosowaniu w wybranym modelu wiedzy, którą inny model posiadł w trakcie treningu. Dzięki temu sieć trenowana do pewnego zadania z użyciem obszernych zbiorów uczących może być wykorzystana w zadaniu podobnym, do którego nie posiada się zbyt wielu danych treningowych. Odbywa się to poprzez wyciągnięcie z wybranego modelu wyuczonych warstw specjalizujących się w pewnych operacjach oraz zastosowanie tych warstw w innym modelu, dzięki czemu można uzyskać, bez konieczności treningu, przykładowo warstwy konwolucyjne wytrenowane do wyciągania z obrazu jego kluczowych cech i identyfikowania charakterystycznych wzorców. Przenoszenie uczenia odbywa się poprzez zaimplementowanie w tworzonym modelu warstw sieci, z której przenosimy wiedzę wraz z opisującymi te warstwy wagami. Takie działania dają olbrzymie możliwości konstruowania wyjątkowo złożonych i zaawansowanych modeli bez konieczności przeprowadzania długotrwałego treningu. Przenoszenie uczenia jest jedną z głównych metod znajdywania rozwiązań różnych problematyk z użyciem sieci neuronowych przy ograniczonych zasobach dostępnej mocy obliczeniowej oraz danych treningowych.

Z związku z licznymi zaletami przenoszenia uczenia, została podjęta próba zastosowania tej metody w zagadnieniu kolorowania czarno-białych obrazów. Do tego celu wykorzystane zostało podejście zaprezentowane przez L. Melas-Kyriazi oraz G. Han w 2016 roku [27]. Wykorzystali oni głębkę sieć splotową będącą połączeniem wyuczonej sieci typu ResNet (ang. Residual Networks) [28] z opracowanymi przez nich warstwami dekonwolucyjnymi. Model ten został zaimplementowany, a jego rezultaty zostały porównane z rezultatami modelu zaprezentowanego w punkcie 4.3.2.

Podobnie jak w modelu autorskim, w procesie uczenia przeprowadzonym przez autorów omawianego rozwiązania została wykorzystana przestrzeń barw CIELab. Wejściowy obraz konwertowany jest oddzielnie do skali szarości oraz do przestrzeni barw CIELab. Na wejście modelu podawana jest wersja obrazu w skali szarości, a danymi referencyjnymi dla wyjścia sieci są dwie składowe koloru A i B obrazu wejściowego w przestrzeni barw CIELab.

Idea testowanej architektury polega na integracji cech wysokiego oraz średniego poziomu z przetwarzanego obrazu. Założenie jest takie, że cechy wysokiego poziomu mogą wskazywać na typ otoczenia. Jeśli wykryte zostanie np. pomieszczenie, to będzie to miało bezpośredni wpływ na cechy średniego poziomu, tak aby wskazywały, że w danej sytuacji niekorzystne jest kolorowanie powierzchni w barwach nieba albo trawy, a bardziej prawdopodobne będą barwy powiązane z wnętrzem budynku. Wykorzystany model składa się z 3 komponentów: sieci ResNet-Gray stanowiącej początkową część testowanego modelu, bloku fuzji oraz sieci dekonwolucyjnej odpowiedzialnej za wygenerowanie oczekiwanych kolorów.

Model ResNet-Gray został opracowany przez autorów testowanego rozwiązania poprzez wytrenowanie sieci ResNet-18 do zadania klasyfikacji czarno-białych obrazów z użyciem zbioru danych ImageNet [29]. Osiągnięta dokładność na zbiorze testowym wyniosła 79.9% w porównaniu z ResNet-18 klasyfikującym kolorowe obrazy z dokładnością 85%. Pozwala to założyć, że model ResNet-Gray poprawnie nauczył się rozpoznawać najistotniejsze cechy obiektów na obrazach, co powinno przełożyć się na satysfakcjonującą jakość kolorowania. Zadanie ResNet-Gray w testowanym rozwiążaniu polega na identyfikacji na obrazie wejściowym kluczowych wzorców wymaganych do wyciągnięcia cech poszczególnych poziomów. Początkowe warstwy wyciągają cechy niskopoziomowe, takie jak położenie krawędzi na obrazie. Warstwy środkowe służą do odpowiedniego przetworzenia tych cech, aby uzyskać cechy średniego poziomu. Są one następnie podawane na kolejne warstwy ResNet-Gray, a dodatkowo trafiają również na blok fuzji. Zadaniem ostatnich warstw jest wydobycie cech wysokopoziomowych, które również są przekazywane do bloku odpowiedzialnego za fuzję.

Blok fuzji służy do przestrzennego połączenia podawanych na niego cech, a następnie przetworzenia ich przez sieć złożoną z jednej warstwy realizującej odpowiednią funkcję, której współczynniki poddawane są optymalizacji w procesie uczenia.

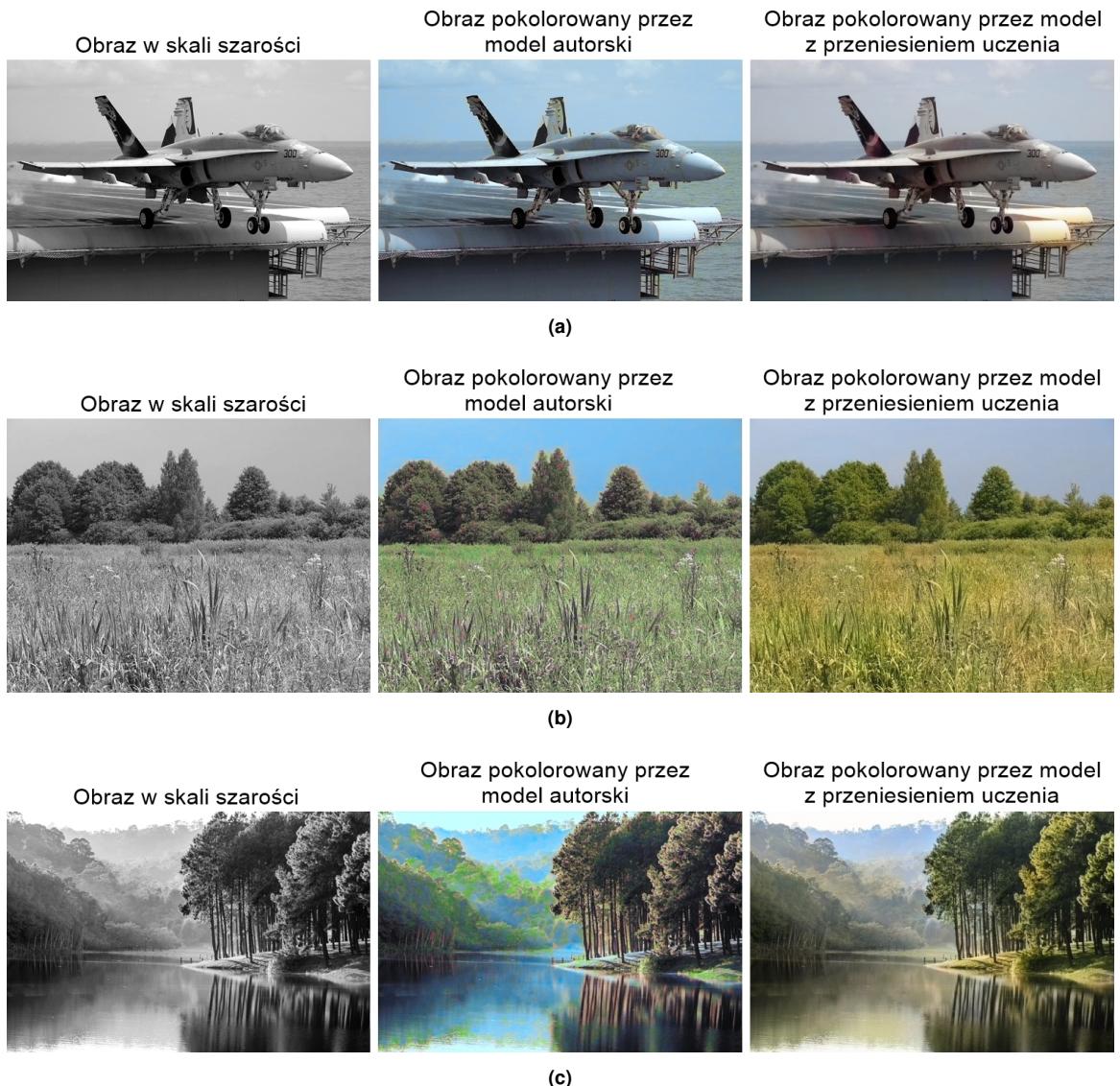
Wyjście bloku fuzji jest następnie przekazywane na sieć dekonwolucyjną, która ma na celu przetworzyć wejściowy zbiór cech, otrzymany w procesie fuzji, na składowe A i B przestrzeni kolorów CIELab dla obrazu wejściowego. Sieć ta składa się z zestawu członów konwolucyjnych i warstw nadpróbkujących (ang. upsampling). Każdy człon konwolucyjny, oprócz dwóch ostatnich, składa się z pojedynczej warstwy konwolucyjnej połączonej z warstwą BatchNorm i warstwą funkcji aktywacji ReLU. Przedostatni człon złożony jest z warstwy konwolucyjnej, po której umieszczona jest warstwa funkcji aktywacji Sigmoid, mająca na celu wygenerowanie oczekiwanych wartości dla składowych kolorów. Ostatni człon konwolucyjny składa się z pojedynczej warstwy konwolucyjnej, której wyjście przekazywane jest na ostatnią warstwę nadpróbkującą będącą zarazem wyjściem sieci dekonwolucyjnej. Warstwy nadpróbkujące w sieci służą do modyfikacji rozdzielcości kanałów przechodzących przez sieć. W tym przypadku zwiększą one dwukrotnie szerokość i wysokość każdego kanału wejściowego używając algorytmu najbliższego sąsiada.

Przedstawiona architektura została zaimplementowana i do każdego z 3 komponentów została zastosowana metoda przeniesienia uczenia polegająca na wczytaniu do wszystkich wykorzystanych warstw wag udostępnionych przez autorów architektury. Tak przygotowana sieć została następnie douczona na zbiorze danych CIFAR-10 w celu zwiększenia rzetelności porównania jej rezultatów z rezultatami modelu autorskiego. Jednakże w procesie uczenia nie wszystkie wagi warstw sieci były aktualizowane. Możliwość aktualizacji wag komponentu ResNet-Gray została zablokowana, aby przeciwdziałać traceniu wyuczonej wiedzy przez ten model. Zjawisko takie może wystąpić, ponieważ model ResNet-Gray był trenowany do zadania klasyfikacji, które jest znacznie odmienne od zagadnienia kolorowania czarno-białych obrazów. W związku z tym, w problematyce kolorowaniu, stosowana jest inna funkcja celu, która wskazuje kierunek optymalizacji wag wymagany do lepszego generowania potencjalnych barw dla obrazów wejściowych. Jeśli wagi ResNet-Gray byłyby aktualizowane zgodnie ze wskazaniami funkcji kosztu, to doprowadziłoby to do zmiany wyuczonej specjalizacji modelu i utracenia przez niego zdolności do prawidłowego wyciągania pożądanych cech z obrazu, co przy założonej idei rozwiązania byłoby niepożądane.

Wybór funkcji kosztu wykorzystanej w procesie uczenia był poparty badaniami przeprowadzonymi przez L. Melas-Kyriazi oraz G. Han. Pomyśladawcy rozwiązania przetestowali skuteczność takich funkcji celu jak L1Loss i MSELoss. Dowiedli oni, że MSELoss pozwala uzyskać żywse barwy w kolorowanych obrazach, dlatego właśnie ta funkcja została wykorzystana w treningu modelu. Kierując się testami przeprowadzonymi w punkcie 4.3.12 oraz zaleceniami autorów ResNet-Gray, jako algorytm optymalizacyjny został wybrany algorytm Adam o parametrach: $\beta_1 = 0.9$, $\beta_2 = 0.999$, długość kroku treningowego = 0.1, współczynnik zaniku wag = $1e - 10$ oraz $eps = 1e - 08$.

Możliwości wyboru metody wstępniego przetwarzania danych są znacznie ograniczone z powodu stosowania przeniesienia uczenia. Wynika to z faktu, że wykorzystując już wyuczone warstwy należy im zapewnić dane wejściowe w takiej samej postaci, jaką była wykorzystana podczas treningu tych warstw. Pierwszym komponentem testowanej architektury jest model ResNet-Gray, model ten samodzielnie dokonuje odpowiedniego przygotowania danych wejściowych, w związku z czym nie ma potrzeby wykonywać tego samodzielnie. Natomiast referencyjne składowe A i B muszą być znormalizowane na przestrzeni całego zbioru danych do przedziału $< 0, 1 >$ tak, jak odbywało się to podczas pierwotnego treningu modelu. Ma to bezpośredni związek z zastosowaniem warstwy funkcji aktywacji Sigmoid, której przedział wartości równy jest $(0, 1)$.

Testowana architektura, po odpowiednim douczeniu, została wykorzystana do pokolorowania wybranych obrazów. Porównanie otrzymanych rezultatów z rezultatami uzyskanymi z użyciem modelu autorskiego zostało przedstawione na Rysunku 4.26.



Rysunek 4.26: Porównanie skuteczności modelu autorskiego i modelu z zastosowaniem metody przeniesienia uczenia.

Analizując pokolorowane obrazy na Rysunku 4.26 można dojść do wniosku, że bardziej satysfakcyjną jakość kolorowania została uzyskana z użyciem modelu, dla którego zastosowano metodę przeniesienia uczenia. Porównując wygenerowane barwy można zauważyć, że dla modelu z przeniesieniem uczenia są one bardziej realistyczne i nadzwyczaj dobrze przypominają barwy prawdopodobne dla powierzchni, na które zostały nałożone. Ponadto, widać, że model autorski ma niekorzystną tendencję do generowanie kolorów o zbyt wysokiej saturacji, co obniża wiarygodność rezultatów tego modelu. Świadczy to, że podejście zaproponowane przez L. Melas-Kyriazi oraz G. Han wykazuje się wysoką skutecznością jako rozwiązanie zagadnienia predykcji kolorów, a integracja cech wysokiego oraz średniego poziomu przynosi oczekiwane rezultaty.

Wielką zaletą zaimplementowanej architektury jest jej modułowość dającą znacznie większe możliwości rozprowadzania w sieci danych z poszczególnych etapów przetwarzania. Sieć dekonwolucyjna dostając dodatkowe informację o kolorowanych obiektach, zwłaszcza o ich klasie, jest w stanie lepiej przewidzieć prawdopodobne barwy dla tych obiektów. Wadą tej architektury jest jednak narzucona stała rozdzielcość obrazów wyjściowych równa 512×512 pikseli. W przypadku kolorowania obrazów o wysokiej rozdzielcości dochodzi do znacznej utraty ich jakości. Model autorski cechuje się większą elastycznością w tym obszarze działań umożliwiając kolorowanie dowolnych obrazów bez modyfikacji ich rozdzielcości.

Jak widać, zastosowanie w przedstawionym rozwiążaniu sieci neuronowej specjalizującej się w operacjach nie do końca związanych z rozwiązywaną problematyką, może pozytywnie wpływać na skuteczność całego rozwiązania. Utwierdza to w przekonaniu, że przeniesienie uczenia jest podejściem o wielu zaletach, które znacznie zwiększa ilość możliwych zastosowań sieci neuronowych. Jest ono szczególnie przydatne w rozwiązaniach, które wymagają zastosowania sieci neuronowej o złożonej architekturze przy jednoczesnym ograniczeniu zasobów obliczeniowych. Ponadto metoda przeniesienia uczenia pozwala, z użyciem douczania, zwiększyć skuteczność końcową modelu poprzez poszerzenie ogólnej wiedzy już wytrenowanej sieci o wiedzę związaną z docelową problematyką. Umożliwia to skuteczne dopracowanie efektywności pracy sieci. Dużą zaletą jest także znacznie ułatwione tworzenie modularnych rozwiązań cechujących się większą uniwersalnością oraz elastycznością.

4.3.17. Podsumowanie

Sieci neuronowe słyną ze swoich niezwykłych zdolności adaptacyjnych sprawiających, że mogą być, z licznymi sukcesami, stosowane do wielu różnych celów. Badania przeprowadzone w tym rozdziale dowodzą, że nadają się one znakomicie także jako rozwiązanie problematyki kolorowania czarno-białych obrazów. Warto zaznaczyć, że jest to problematyka bardzo złożona, dla której nie istnieje proste rozwiązanie używające klasycznych metod edycji obrazu. W rozdziale tym zostały przedstawione dwa funkcjonujące rozwiązania. Posiadają one swoje zalety i wady, jednakże oba prezentują zadowalające rezultaty.

Model autorski przedstawiony w punkcie 4.3.2 jest dowodem na to, że wstępnie pozytywne wyniki można osiągnąć stosując nawet proste sieci splotowe. Model taki, ze względu na nieskomplikowaną architekturę, jest mało problematyczny w implementacji, a jego skuteczny trening można przeprowadzać bez dużych nakładów mocy obliczeniowej. Sprawia to, że jest on wyjątkowo łatwo dostępny i stanowi znakomitą bazę pod różnorakie eksperymenty i potencjalne rozszerzenia.

Inne podejście zostało zaprezentowane w przypadku modelu opisanego w punkcie 4.3.16. Fundamentem tego rozwiązania jest wykorzystanie techniki przeniesienia uczenia redukującej ograniczenia wynikające z niewystarczających zasobów obliczeniowych. Wykorzystany model cechuje się znacznie większą złożonością zastosowanej architektury niż model autorski. Bazuje on na efektywnej współpracy pomniejszych komponentów składających się na całokształt rozwiązania. Charakter tego podejścia umożliwił stworzenie znacznie bardziej zaawansowanego modelu uwzględniającego większą ilość czynników kluczowych dla rozważanego zagadnienia, co przekłada się na wysoce zadowalające rezultaty.

Zaprezentowane w tym rozdziale wyniki dowodzą wysokiej użyteczności i nadzwyczajnej skuteczności zastosowania sieci neuronowych do rozwiązywania niepospolitych oraz złożonych problemów, tak w zakresie edycji obrazu jak i w innych dziedzinach.

5. PODSUMOWANIE

Celem zaprezentowanego projektu inżynierskiego było stworzenie oprogramowania, opartego na technologii sztucznych sieci neuronowych, umożliwiającego wszechstronną edycję obrazów cyfrowych. Przygotowane rozwiązania należało przetestować, a zastosowane implementacje przeanalizować pod kątem skuteczności działania ze względu na różnorodny dobór parametrów oraz architektury sieci.

Dodatkowo, w celu uproszczenia i przyspieszenia wykonywanych eksperymentów, postawione zostało zadanie przygotowania framework'a w języku Python, umożliwiającego łatwe komponowanie oraz testowanie narzędzi opartych o mechanizmy sieci neuronowych.

Cel ten został zrealizowany, a zaimplementowane rozwiązanie otrzymało nazwę *TorchFrame*. Szczegółowy opis działania framework'a zawarty został w podrozdziale 4.1. Opracowane oprogramowanie kontroluje przepływ danych w procesie uczenia sieci neuronowych udostępniając użytkownikom szeroki zakres gotowych do użycia metod konwersji obrazu. Ułatwia to znacznie zagadnienia wstępnego oraz wtórnego przetwarzania danych treningowych. Dodatkowo uproszczone zostało zagadnienie parametryzacji całego procesu. Użytkownik *TorchFrame* definiuje wszystkie istotne elementy wykorzystując pojedynczy plik JSON, którego struktura została szczegółowo opisana w ramach tej pracy. Komponentowa budowa framework'a umożliwiła wydzielenie części dedykowanej szeroko pojętym testom zaimplementowanych rozwiązań. W ramach przeprowadzonych eksperymentów zdefiniowane zostały przykładowe struktury programistyczne dostosowujące *TorchFrame* do dedykowanych problemów. Jego elastyczna struktura umożliwia jednak intuicyjne implementowanie rozmaitych modyfikacji ograniczonych jedynie wyobraźnią użytkowników.

Przygotowane w ten sposób narzędzie posłużyło w realizacji wspomnianego oprogramowania przeznaczonego do edycji obrazów cyfrowych. Realizacja tego zadania przedstawiona została w dwóch fazach.

W podrozdziale 4.2 opracowana została seria filtrów wzorowanych na klasycznych metodach przetwarzania obrazów, takich jak sepia, czy wykrywający krawędzie filtr Sobela-Feldmana. Każda z oryginalnych metod odtworzona została z wykorzystaniem jak najprostszych modeli neuronowych sieci splotowych. Uzyskane rezultaty przeanalizowane zostały pod kątem skuteczności działania w odniesieniu do zastosowanych hiperparametrów oraz metod przetwarzania danych treningowych. Kluczowym elementem tego rozdziału było wskazanie podobieństwa między klasycznymi metodami filtracji, a używanymi powszechnie sieciami splotowymi, zbudowanymi z warstw neuronów nazywanych właśnie filtrami.

Rozdział 4.3 przedstawia próby automatycznego kolorowania czarno-białych obrazów. Zagadnienie to zostało rozwiążane z użyciem dwóch różnych modeli, autorskiego modelu prostego oraz bardziej zaawansowanego modelu zaimplementowanego z użyciem technik przeniesienia uczenia. Dla modelu autorskiego zostały przeprowadzone szczegółowe badania dotyczące zależności wyników od poszczególnych parametrów architektury sieci, jak i konfiguracji procesu uczenia. Rezultaty końcowe uzyskane z użyciem tego modelu były zadowalające, ale mocno zależne od wybranych parametrów. Dowodzi to, że odpowiednio skonfigurowane sieci splotowe mogą być skutecznie zastosowane w tym zagadnieniu, aczkolwiek to model złożony pozwolił osiągnąć największy sukces. Idea tego rozwiązania opiera się na integracji cech obrazu średniego oraz wysokiego poziomu uzyskiwanych z użyciem złożonej sieci splotowej wytrenowanej pierwotnie do zadania klasyfikacji. Cechy te, po przejściu przez proces fuzji, są następnie wykorzystywane przez sieć dekonwolucyjną do predykcji prawdopodobnych barw dla obrazu wejściowego. Dzięki tym dodatkowym informacjom o obrazie udało się znacznie zwiększyć efektywność procesu kolorowania, a co za tym idzie, wiarygodność generowanych barw. Pokolorowane obrazy w wielu przypadkach wyglądały tak realistycznie, że niemożliwym było zauważenie, że ich kolory zostały wygenerowane automatycznie, co świadczy o nadzwyczaj wysokiej skuteczności bardziej zaawansowanego modelu.

Przeprowadzone eksperymenty i uzyskane w ich wyniku rezultaty jasno wskazują na użyteczność sztucznych sieci neuronowych w procesie edycji obrazów. Ich zdolności adaptacyjne sprawiają, że są one w stanie odtworzyć niemal każde klasyczne rozwiązanie, a umiejętność rozpoznawania i wykorzystywania skomplikowanych wzorców czyni je ponadto niezastąpionymi w sytuacjach, w których zawodzą tradycyjne metody.

Należy jednak pamiętać, że sieci neuronowe wciąż znajdują się w fazie rozwoju, a skuteczność ich działania bardzo często nie dorównuje stosowanym powszechnie algorytmom. Dobrym przykładem takiego stanu rzeczy są zaprezentowane filtry AI. Zastosowanie prostych masek filtrujących dało w tym przypadku lepsze rezultaty niż, wymagające znacznych nakładów pracy, sieci splotowe, mimo przeprowadzenia licznych eksperymentów i prób dla różnorodnych zestawów hiperparametrów.

Z drugiej strony, zaprezentowane rozwiązanie inteligentnego kolorowania czarno-białych obrazów pokazało, że zastosowanie złożonych architektur sieci neuronowych, a także odpowiednio duży nakład pracy włożony w proces treningowy, mogą doprowadzić do rozwiązania nieosiągalnego innymi metodami.

Sztuczna inteligencja bez wątpienia stanowi fenomen w świecie nauki. Nie należy jednak traktować jej jak młotka, dla którego każdy problem wygląda jak gwóźdź, ale warto nieustannie śledzić dynamiczny rozwój tej dziedziny wiedzy. Doświadczenie pokazuje, że znajduje ona zastosowanie w coraz szerszym gronie aspektów życia codziennego. Bardzo często nie jesteśmy nawet świadomi jej obecności wokół nas. W związku z zachodzącymi zmianami nie należy pytać czy sztuczna inteligencja dorówna kiedyś innym, konkurencyjnym metodom. Należy zadać sobie pytanie kiedy to się stanie.

WYKAZ LITERATURY

- [1] T. Liu, S. Fang, Y. Zhao, P. Wang, J. Zhang: *Implementation of Training Convolutional Neural Networks*, arXiv ('2015), s.2.
- [2] I. Goodfellow, Y. Bengio, A. Courville: *Deep Learning*, ('2016), s.342., s.82.
- [3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio: *Generative Adversarial Networks*, arXiv ('2014)
- [4] R. Zhang, P. Isola, A. A. Efros: *Colorful Image Colorization*, arXiv ('2016)
- [5] L. A. Gatys, A. S. Ecker, M. Bethge: *Image Style Transfer Using Convolutional Neural Networks*, IEEE ('2016)
- [6] K. Simonyan, A. Zisserman: *Very Deep Convolutional Networks For Large-Scale Image Recognition*, ('2016)
- [7] G. Perarnau, J. van de Weijer, B. Raducanu, J. M. Álvarez: *Invertible Conditional GANs for image editing*, arXiv ('2016)
- [8] A. Brock, T. Lim, J.M. Ritchie, N. Weston: *Neural Photo Editing With Introspective Adversarial Networks*, arXiv ('2017)
- [9] D. P. Kingma, M. Welling: *Auto-Encoding Variational Bayes*, arXiv ('2014)
- [10] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer: *Automatic differentiation in PyTorch*, 31st Conference on Neural Information Processing Systems ('2017), Long Beach, CA, USA
- [11] R. Reed, R. J MarksII: *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, Paperback – February 17 ('1999), s.155.
- [12] R. Rojas: *Neural Networks - A Systematic Introduction*, Springer-Verlag, Berlin, New-York ('1996), s.161.
- [13] J. López, Á. Barbero, J. R. Dorronsoro: *Momentum Acceleration of Least–Squares Support Vector Machines*, International Conference on Artificial Neural Networks ('2011)
- [14] I. Sutskever, J. Martens, G. Dahl, G. Hinton: *On the importance of initialization and momentum in deep learning*, University of Toronto ('2013)
- [15] J. Kiefer, J. Wolfowitz: *Stochastic Estimation of the Maximum of a Regression Function*, Ann. Math. Statist. 23 ('1952), no. 3, s.462-466.
- [16] J. Duchi, E. Hazan, Y. Singer: *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research 12 ('2011) s.2121-2159.
- [17] M. D. Zeiler: *AdaDelta: An Adaptive Learning Rate Method*, arXiv ('2012)

- [18] D. P. Kingma, J. L. Ba: *Adam: A Method for Stochastic Optimization*, arXiv ('2014)
- [19] I. Sobel: *An Isotropic 3x3 Image Gradient Operator*, ResearchGate ('2014), s.4.
- [20] A. Makandar, B. Halalli: *Image Enhancement Techniques using Highpass and Lowpass Filters*, International Journal of Computer Applications ('2015), s.13.
- [21] S. Ioffe, C. Szegedy: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv ('2015)
- [22] Y. Bengio, P. Simard, P. Frasconi: *Learning long-term dependencies with gradient descent is difficult*, IEEE ('1994)
- [23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research ('2014)
- [24] A. Krizhevsky: *Learning Multiple Layers of Features from Tiny Images*, Master's thesis, Department of Computer Science, University of Toronto ('2009)
- [25] R. Girshick: *Fast R-CNN*, arXiv ('2015)
- [26] B. Xu, N. Wang, T. Chen, M. Li: *Empirical Evaluation of Rectified Activations in Convolution Network*, arXiv ('2015)
- [27] L. Melas-Kyriazi, G. Han: *Combining Deep Convolutional Neural Networks with Markov Random Fields for Image Colorization*, arXiv ('2016)
- [28] K. He, X. Zhang, S. Ren, J. Sun: *Deep Residual Learning for Image Recognition*, arXiv ('2015)
- [29] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei.: *ImageNet: A large-scale hierarchical image database*, Conference on Computer Vision and Pattern Recognition, nieefektywna ('2009)

SPIS RYSUNKÓW

| | |
|--|----|
| 2.1 Struktura głębokiej sieci neuronowej składającej się wyłącznie z warstw gęstych - źródło: https://towardsdatascience.com | 10 |
| 2.2 Przykładowa struktura CNN - źródło: https://www.mathworks.com | 11 |
| 2.3 Segmentacja semantyczna - źródło: https://devblogs.nvidia.com | 12 |
| 3.1 Efekt kolorowanie czarno-białych obrazów przez wytrenowany model - źródło: [4] | 14 |
| 3.2 Obrazy będące kombinacją treści zdjęcia ze stylami kilku znanych dzieł sztuki - źródło: [5] | 16 |
| 3.3 Obrazy generowane przez IcGAN - źródło: [7] | 17 |
| 3.4 Efekt działania <i>Neural Photo Editor</i> - źródło: [8] | 19 |
| 4.1 Architektura <i>TorchFrame</i> - źródło: Rysunek własny | 23 |
| 4.2 Przykładowa płaszczyzna funkcji celu - źródło: https://towardsdatascience.com . | 28 |
| 4.3 Przykładowy kształt punktu siodłowego - źródło: https://towardsdatascience.com | 30 |
| 4.4 Architektura testowa <i>TorchFrame</i> - źródło: Rysunek własny | 36 |
| 4.5 Przykładowe obrazy ze zbioru treningowego - źródło: Rysunek własny | 38 |
| 4.6 Działanie filtru Sobela - źródło: Rysunek własny | 39 |
| 4.7 Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych - źródło: Rysunek własny | 40 |
| 4.8 Działanie sepii - źródło: Rysunek własny | 42 |
| 4.9 Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych - źródło: Rysunek własny | 43 |
| 4.10 Przykład działania sieci w minimum lokalnym - źródło: Rysunek własny | 44 |
| 4.11 Działanie filtru górnoprzepustowego - źródło: Rysunek własny | 45 |
| 4.12 Wykres wartości funkcji kosztu w zależności od ilości kroków treningowych - źródło: Rysunek własny | 46 |
| 4.13 Przykład działania sieci w minimum lokalnym - źródło: Rysunek własny | 46 |
| 4.14 Przestrzeń barw CIELab - źródło: https://www.flickr.com | 49 |
| 4.15 Przykładowa składowa L - źródło: Rysunek własny wykorzystujący: https://fr.m.wikipedia.org | 49 |
| 4.16 Przykładowe operacje warstwy poolingu - źródło: Rysunek własny | 53 |
| 4.17 Przykładowe obrazy z CIFAR-10 - źródło: Rysunek własny | 54 |
| 4.18 Funkcja aktywacji ReLU - źródło: https://pytorch.org | 59 |
| 4.19 Efekt działania modelu autorskiego - źródło: Rysunek własny bazujący na: https://fr.m.wikipedia.org , https://pl.wikipedia.org , [24] | 60 |

| | |
|--|----|
| 4.20 Wykres uczenia modelu autorskiego - źródło: Rysunek własny | 61 |
| 4.21 Efekt zastosowania algorytmu przetwarzania końcowego - źródło rysunek własny na podstawie: https://pl.wikipedia.org | 62 |
| 4.22 Porównanie rezultatów modeli uczonych z użyciem różnych funkcji kosztu - źródło rysunek własny na podstawie: https://cdn.theearthhunters.com | 62 |
| 4.23 Porównanie rezultatów modeli uczonych z użyciem różnych algorytmów optymalizacyjnych - źródło rysunek własny na podstawie: https://cdn.theearthhunters.com | 64 |
| 4.24 Wpływ zastosowania warstwy Dropout na kolorowanie obrazów - źródło: rysunek własny na podstawie: https://pl.wikipedia.org | 65 |
| 4.25 Skutki zastosowania różnych metod przetwarzania wstępnego - źródło: rysunek własny na podstawie https://pl.wikipedia.org | 65 |
| 4.26 Porównanie skuteczności modelu autorskiego i modelu z zastosowaniem metody przenesienia uczenia - źródło: Rysunek własny bazujący na: https://fr.m.wikipedia.org , https://pl.wikipedia.org , https://cdn.theearthhunters.com | 71 |

SPIS TABEL

| | |
|---|----|
| 4.1 Funkcje kosztu w <i>TorchFrame</i> | 25 |
| 4.2 Optymalizatory w <i>TorchFrame</i> | 31 |
| 4.3 Architektura modelu autorskiego. | 50 |
| 4.4 Szczegóły konfiguracji przetwarzania wstępnego. | 66 |