

Algorytmy sztucznej inteligencji w Przemyśle 4.0

Generative Adversarial Network jako narzędzie do łączenia obrazów satelitarnych - SatGAN

Autor dokumentu:
Bartłomiej Motłoch

Spis treści

Spis treści	2
1 Wstęp teoretyczny	3
1.1 Motywacja projektu	3
1.1.1 Wady tradycyjnych metod	4
1.1.2 Potencjalne zalety wykorzystania sieci GAN	4
1.2 Wykorzystanie scalania obrazów w Przemyśle 4.0	5
1.3 Podstawy teoretyczne sieci GAN	5
1.3.1 Zasada działania sieci GAN	6
1.3.2 Architektura UNet w kontekście GAN	6
1.3.3 Rola połączeń skip w UNet	6
1.3.4 Rola rozmiaru filtrów i ich hierarchiczność	6
2 Wybrane technologie	7
2.1 Biblioteki i frameworki	7
3 Dane treningowe, dane testowe	7
3.1 Źródło danych	7
3.2 Charakterystyka danych	8
3.3 Preprocessing danych	8
4 Opis funkcji i klas	9
4.1 plik padder.py	9
4.2 plik plotter.py	10
4.3 plik divider.py	10
4.4 plik test.py	11
4.5 Plik sift.py	12
4.6 pliki gan.py,gan_skip.py itp.	14
5 Architektury	15
5.1 Wprowadzenie do architektur modelu	15
5.1.1 Inicjalizacja wag	15
5.1.2 Warstwy normalizujące	15
5.1.3 Warstwy aktywacyjne	15
5.2 Architektura pierwsza - miniatura z pracy naukowej	16
5.2.1 Generator	16
5.2.2 Dyskriminator	16
5.3 Architektura druga - hierarchiczność rozmiaru filtrów w enkoderze	17
5.3.1 Generator	17
5.3.2 Dyskriminator	17
5.4 Architektura trzecia - zwiększenie pojemności generatora przy stałym rozmiarze filtrów	18
5.4.1 Generator	18
5.4.2 Dyskriminator	18
5.5 Architektura czwarta - standardowa pojemność generatora przy dodatkowej warstwie w dykryminatorze	19
5.5.1 Generator	19
5.5.2 Dyskriminator	19
6 Proces treningu	19
6.1 Parametry treningu	20
6.2 Problemy napotkane podczas treningu	20
6.3 Nieintuicyjność przebiegu funkcji strat	21
6.4 Mode collapse w architekturze trzeciej	23
6.5 Wybór najlepszej wersji modelu z treningu	25
6.5.1 MSE (Mean Squared Error)	25
6.5.2 PSNR (Peak Signal-to-Noise Ratio)	25
6.5.3 Praktyczne zastosowanie MSE i PSNR w projekcie	26

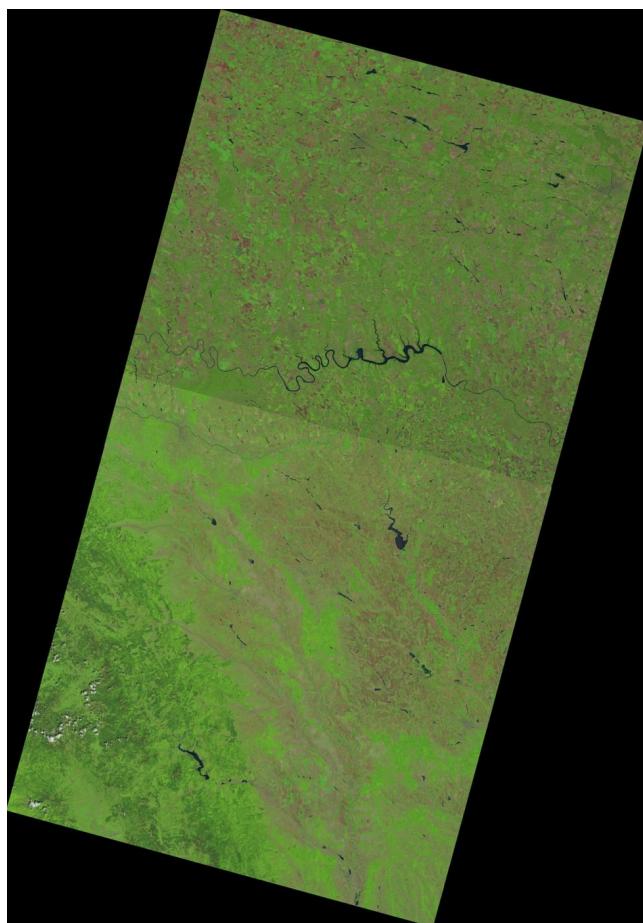
7 Ewaluacja najlepszego modelu	26
7.1 Przebiegi funkcji strat	26
7.2 Wykres średniej wartości MSE oraz PSNR	28
7.3 Porównanie czasowe obu metod	29
7.4 Wizualizacja przebiegu nauki dla wybranej architektury	31
8 Podsumowanie oraz wnioski	31
8.1 Najważniejsze osiągnięcia	31
Literatura	31

1 Wstęp teoretyczny

1.1 Motywacja projektu

Automatyzacja tworzenia panoram zdjęć satelitarnych stanowi istotny element w analizie danych geograficznych, monitorowaniu środowiska oraz tworzeniu precyzyjnych map. Zdjęcia satelitarne i lotnicze wykonywane są zazwyczaj w seriach, w których poszczególne obrazy częściowo nakładają się na siebie. Taki proces jest konieczny, aby uzyskać pełne pokrycie danego obszaru. Przykładem mogą być satelity takie jak Landsat 8 czy Landsat 9, które rejestrują zdjęcia tego samego obszaru w regularnych odstępach czasowych, średnio co 8 dni. Obrazy te są ustawione zgodnie z trajektorią przelotu satelity, najczęściej wzdłuż południków, co dodatkowo komplikuje ich późniejsze scalanie w jednolite bloki.

W przypadku zdjęć satelitarnych, gdzie długość boku pojedynczego obrazu może wynosić kilkanaście tysięcy pikseli, na znaczeniu zdobywa dokładne i szybkie dopasowanie sąsiadujących obrazów. Bez automatyzacji tego procesu manualne łączenie setek lub tysięcy obrazów byłoby niezwykle czasochłonne i podatne na błędy.



Rysunek 1: Przykład poprawnego złączenia dwóch obrazów satelitarnych przy użyciu algorytmu SIFT. Długość boku pojedynczego kwadratowego obrazu wynosi około 13 tysięcy pikseli.

Aby uzyskać spójne dane, które można wykorzystać do dalszych analiz lub tworzenia map, konieczne jest przekształcenie tych nakładających się zdjęć w taki sposób, aby utworzyć jednolite bloki obrazu, gdzie elementy

się nie powtarzają ani nie nakładają. Jest to ważne w wielu dziedzinach, takich jak geodezja, klimatologia, monitorowanie zmian w środowisku, a także w tworzeniu precyzyjnych modeli terenów.

1.1.1 Wady tradycyjnych metod

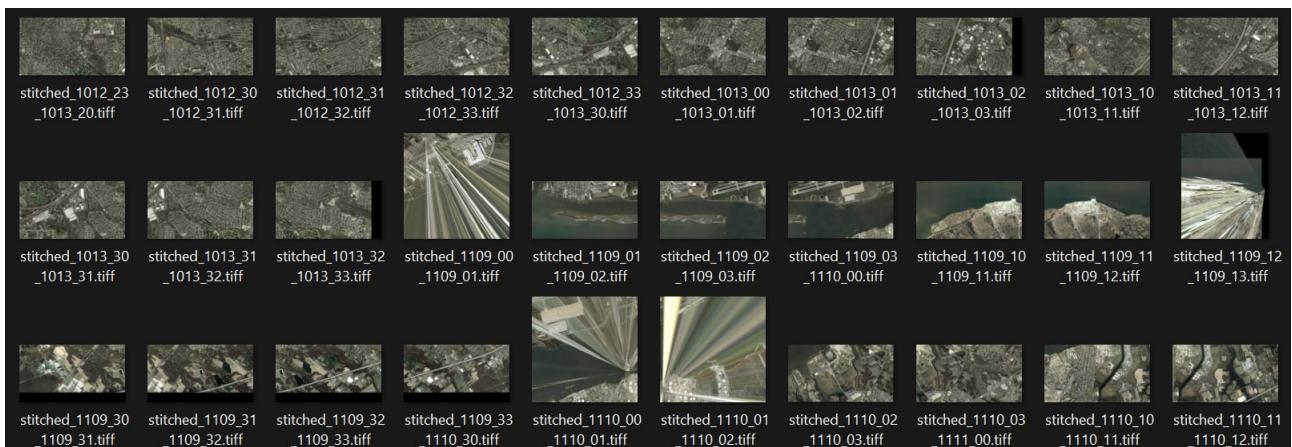
W przypadku klasycznych panoram istnieje wiele rozwiązań programistycznych, które realizują scalanie obrazów. Najczęściej proces ten opiera się na wyszukiwaniu punktów charakterystycznych między obrazami przeznaczonymi do połączenia. Punkty te są następnie dopasowywane, a jeden z obrazów poddawany jest transformacji homograficznej w celu uzyskania odpowiedniego wyrównania. Transformacja ta może być różnego rodzaju, od prostego przesunięcia aficznego aż po bardziej zaawansowane przekształcenia uwzględniające problemy z paralaksą, różnicami w skali czy rotacją.

Najpopularniejsze algorytmy używane do wyszukiwania punktów charakterystycznych to:

- **SIFT (Scale-Invariant Feature Transform)** - algorytm ten umożliwia wykrywanie punktów kluczowych, które są odporne na zmiany skali, rotację oraz inne transformacje geometryczne. Jednak SIFT jest zasobozerny i wolny, co ogranicza jego zastosowanie w dużych zestawach danych.
- **ORB (Oriented FAST and Rotated BRIEF)** - algorytm ten jest szybszy i mniej zasobozerny niż SIFT, jednak jego skuteczność w wykrywaniu punktów charakterystycznych jest ograniczona. W szczególności nie radzi sobie dobrze z obrazami o dużych różnicach w jasności, kontrastach czy perspektywach.

Wady tradycyjnych metod są szczególnie widoczne w przypadku przetwarzania obrazów satelitarnych:

- **Czasochłonność** – Proces wyszukiwania punktów charakterystycznych oraz obliczania transformacji homograficznych wymaga dużej mocy obliczeniowej. Przy przetwarzaniu dużych zbiorów obrazów satelitarnych staje się to wysoce nieefektywne.
- **Problemy ze skalą i rotacją** – Nawet zaawansowane algorytmy, takie jak SIFT, mogą mieć trudności z dopasowaniem obrazów, gdy występują znaczące różnice w skali, rotacji lub perspektywie. Te czynniki są często obecne w zdjęciach satelitarnych z różnych przelotów.
- **Niska jakość dopasowania w złożonych scenariuszach** – Algorytmy bazujące na punktach charakterystycznych zawodzą w przypadku obrazów o niewielkiej ilości detali, takich jak tereny wodne, pustynie czy jednolite pokrycie chmur. Podczas próby automatyzacji za pomocą SIFT w celu stworzenia zbioru *ground truth* dla dyskryminatora, zauważono, że algorytm często zawodził na obrazach z małą liczbą punktów charakterystycznych. Dodatkowo zmienne warunki atmosferyczne lub różnice w oświetleniu mogą znaczaco obniżyć jakość dopasowania.



Rysunek 2: Przykład obrazów błędnie scalonych przez SIFT

1.1.2 Potencjalne zalety wykorzystania sieci GAN

W obliczu ograniczeń tradycyjnych metod, takich jak SIFT, coraz większe zainteresowanie budzą nowe podejścia, w tym sieci generatywne (GAN). Dzięki swoim właściwościom, GAN mogą znaczaco poprawić zarówno jakość, jak i efektywność procesu tworzenia panoram. Szczególnie obiecujące jest połączenie sieci GAN z architekturą UNet, co pozwala na wykorzystanie zalet obu technologii. Potencjalne korzyści obejmują:

- **Automatyzacja procesu** – Sieci GAN uczą się bezpośrednio na danych treningowych, co umożliwia automatyczne przewidywanie i stosowanie optymalnych przekształceń obrazów. W ten sposób eliminowana jest konieczność ręcznego dostrajania parametrów czy wybierania punktów charakterystycznych.
- **Efektywność obliczeniowa** – Po odpowiednim wytrenowaniu, sieci GAN są w stanie generować wyniki szybciej niż klasyczne algorytmy. Proces generowania obrazu przez wybraną architekturę zależy głównie od rozmiaru obrazów, co czyni tę metodę skalową przy pracy z dużymi zbiorami danych satelitarnych.
- **Radzenie sobie z różnicami między obrazami** – GAN, dzięki zdolności do uczenia się wzorców i generowania danych, mogą skutecznie dopasowywać obrazy nawet w przypadku dużych różnic w skali, perspektywie, rotacji czy oświetleniu. Jest to szczególnie istotne w kontekście zdjęć satelitarnych, które są często wykonywane w zróżnicowanych warunkach.
- **Adaptacja do różnych warunków** – Modele GAN można dostosować do specyficznych rodzajów danych, takich jak obrazy satelitarne w pasmach RGB, IR (podczerwień) czy innych, a także do zdjęć wykonywanych o różnych porach dnia lub w różnych warunkach pogodowych. Taka elastyczność zwiększa uniwersalność tej technologii.

Wykorzystanie sieci GAN do automatyzacji tworzenia panoram stanowi więc obiecującą alternatywę dla tradycyjnych metod. Co ciekawe, analiza istniejących implementacji i publikacji naukowych pokazuje, że temat ten wciąż pozostaje stosunkowo słabo zbadany. Większość istniejących rozwiązań opiera się na głębokich sieciach wymagających dużych zasobów VRAM na pojedynczej karcie graficznej lub treningu w środowisku chmurowym. Taki stan rzeczy otwiera pole do eksploracji mniejszych, bardziej dostępnych podejść.

1.2 Wykorzystanie scalania obrazów w Przemyśle 4.0

Scalanie obrazów odgrywa ważną rolę w Przemyśle 4.0. Integracja automatyzacji, analizy danych oraz wizji komputerowej pozwala na efektywne rozwiązywanie problemów w różnych branżach. Można wymienić przykładowo:

- **Analiza gleby** - Scalanie obrazów może być stosowane w rolnictwie precyzyjnym do analizy właściwości gleby. Obrazy z dronów lub satelitów, wykonywane w różnych pasmach spektralnych (np. widzialnym, podczerwieni), często wymagają łączenia w celu uzyskania pełnego obrazu badanego obszaru. Dzięki automatyzacji tego procesu można:
 - Rozpoznawać obszary różniące się poziomem wilgotności, zawartością składników mineralnych oraz stopniem erozji gleby.
 - Generować precyzyjne mapy wspierające planowanie nawożenia oraz monitorowanie występowania chwastów.
- **Analiza tkanin** - W przemyśle tekstylnym scalanie obrazów pozwala na inspekcję jakości materiałów. Zastosowanie zaawansowanych algorytmów w połączeniu z kamerami o wysokiej rozdzielcości umożliwia:
 - Wykrywanie wad produkcyjnych, takich jak nierówności w strukturze tkaniny, przerwy w splotach czy przebarwienia.
 - Automatyczne porównywanie wzorców materiałów z oczekiwany standardem.
- **Kontrola trasy ruchu ramion robotycznych przy ograniczonym polu widzenia kamer** - W robotyce przemysłowej, szczególnie w przypadku zautomatyzowanych linii produkcyjnych, kamery o ograniczonym polu widzenia są często wykorzystywane do monitorowania i sterowania ruchem ramion robotów. Scalanie obrazów w tym kontekście:
 - Pozwala na stworzenie pełnej mapy środowiska roboczego, zwiększając precyzyję i bezpieczeństwo ruchu robotów.
 - Umożliwia dokładniejsze śledzenie i identyfikację obiektów znajdujących się w polu pracy, nawet jeśli są częściowo zasłonięte.
 - Minimalizuje ryzyko kolizji i optymalizuje trasy ruchu w ograniczonej przestrzeni.

1.3 Podstawy teoretyczne sieci GAN

Sieci Generatywne Adwersarialne (Generative Adversarial Networks, GAN) to technologia uczenia maszynowego, która pozwala generować realistyczne dane na podstawie wzorców obecnych w danych treningowych. Koncepcja GAN została wprowadzona przez Iana Goodfellowa w 2014 roku i opiera się na współprzewodnictwie dwóch modeli: generatora i dyskryminatora.

1.3.1 Zasada działania sieci GAN

GAN składa się z dwóch głównych komponentów:

- **Generator:** Jego zadaniem jest generowanie nowych danych, które mają przypominać dane rzeczywiste. W przypadku przetwarzania obrazów generator przekształca losowy szum (zwykle próbkę z rozkładu normalnego) w obraz, który powinien być nieodróżnialny od prawdziwych obrazów.
- **Dyskryminator:** Funkcjonuje jako klasyfikator, którego celem jest odróżnienie wygenerowanych obrazów od rzeczywistych. Dyskryminator próbuje maksymalizować swoją dokładność w klasyfikacji, działając na korzyść rozróżniania danych rzeczywistych i syntetycznych.

Oba modele rywalizują ze sobą w procesie optymalizacji, tworząc grę minimaksową:

- Generator stara się „oszukać” dyskryminatora, generując coraz bardziej realistyczne dane.
- Dyskryminator uczy się skuteczniejszego odróżniania danych rzeczywistych od fałszywych.

Ten proces prowadzi do wzajemnego doskonalenia obu modeli. Po osiągnięciu równowagi generator jest w stanie tworzyć dane niemal identyczne z rzeczywistymi.

1.3.2 Architektura UNet w kontekście GAN

UNet to rodzaj konwolucyjnej sieci neuronowej zaprojektowany pierwotnie do segmentacji obrazów medycznych, ale znalazł zastosowanie w wielu zadaniach związanych z przetwarzaniem obrazów, takich jak rekonstrukcja, stylizacja czy usuwanie szumów. UNet składa się z dwóch części:

- **Część kontrakcyjna (downsampling):** Wykonuje hierarchiczną ekstrakcję cech za pomocą kolejnych warstw konwolucyjnych i operacji maksymalnego spoolingowania (max-pooling) lub kombinacji stride oraz padding. Dzięki temu model uczy się globalnych właściwości obrazu, przechodząc od lokalnych szczegółów do bardziej abstrakcyjnych reprezentacji.
- **Część ekspansyjna (upsampling):** Ta faza odpowiada za rekonstrukcję szczegółów przestrzennych obrazu. Wykorzystuje operacje transponowanej konwolucji (ang. transposed convolution), które zwiększą rozdzielcość cech. Ważnym aspektem tej części jest łączenie (concatenation) z odpowiadającymi warstwami części kontrakcyjnej, co jest realizowane za pomocą połączeń skip (ang. skip connections).

W sieciach GAN architektura UNet może być szczególnie efektywna, gdy stosowana jest w generatorze. Dzięki połączonym skip model potrafi lepiej odwzorować zarówno globalne struktury obrazu, jak i jego lokalne detale, co prowadzi do uzyskania wyraźniejszych i bardziej realistycznych wyników.

1.3.3 Rola połączeń skip w UNet

Połączenia skip umożliwiają przekazywanie informacji z warstw kontrakcyjnych do odpowiadających im warstw ekspansyjnych. Pozwalają na:

- Zachowanie szczegółów lokalnych, które mogłyby zostać utracone w wyniku kompresji podczas procesu downsampleingu.
- Łatwiejsze rekonstrukcje drobnych detali obrazu, takich jak krawędzie i tekstury, dzięki ponownemu wykorzystaniu cech niskopoziomowych. Poprawę stabilności treningu, szczególnie w przypadku zadań wymagających precyzyjnej rekonstrukcji obrazu.

UNet z połączonymi skip znajduje zastosowanie w różnych zadaniach przetwarzania obrazów, w tym w generacji panoram zdjęć satelitarnych, gdzie dokładność rekonstrukcji i odwzorowanie szczegółów są kluczowe dla jakości wyników.

1.3.4 Rola rozmiaru filtrów i ich hierarchiczność

W sieciach konwolucyjnych, takich jak UNet, szczególne znaczenie ma projektowanie filtrów:

- **Rozmiar filtrów:** Mniejsze filtry (np. 3×3) są bardziej efektywne w ekstrakcji lokalnych wzorców i wymagają mniej parametrów, co pozwala na lepszą generalizację. Większe filtry mogą uchwycić bardziej globalne informacje, ale są bardziej kosztowne obliczeniowo. W praktyce często stosuje się mniejsze filtry wielokrotnie w hierarchicznych warstwach, aby uzyskać globalny kontekst przy jednoczesnym zachowaniu lokalnej precyzji.

- **Hierarchiczność:** W miarę przechodzenia do głębszych warstw sieci, filtry uczą się bardziej abstrakcyjnych cech, przechodząc od prostych krawędzi i tekstur w początkowych warstwach do bardziej złożonych struktur w głębszych warstwach. Taka hierarchiczność jest kluczowa w zadaniach generatywnych, ponieważ pozwala sieci na odtworzenie zarówno ogólnej struktury obrazu, jak i jego drobnych detali.

2 Wybrane technologie

W celu realizacji projektu wykorzystano różnorodne technologie, które wspierają zarówno implementację modeli sieciowych, jak i przetwarzanie oraz analizę danych.

2.1 Biblioteki i frameworki

Do implementacji i przeprowadzenia eksperymentów z sieciami generatywnymi (GAN) oraz przetwarzaniem danych wykorzystano następujące biblioteki:

- **PyTorch:** Jest to jeden z najpopularniejszych frameworków do uczenia maszynowego, szczególnie w zastosowaniach związanych z głębokim uczeniem. PyTorch zapewnia elastyczność w definiowaniu i trenowaniu modeli, dzięki czemu z łatwością zaimplementowano różne architektury GAN.
- **OpenCV:** Biblioteka służąca do przetwarzania obrazów, która została wykorzystana m.in. do klasycznej implementacji algorytmu SIFT (Scale-Invariant Feature Transform) w celu porównania wyników z metodą opartą na sieciach GAN. Dzięki OpenCV możliwe było wyszukiwanie punktów charakterystycznych oraz analiza dopasowania obrazów. Dodatkowo wykorzystana została do podzielenia dużych zdjęć satelitarnych na mniejsze fragmenty (tiles).
- **OS i re:** posłużyły do zarządzania plikami i katalogami, co było przydatne przy organizacji danych, generowaniu zbiorów treningowych oraz analizie struktur folderów.
- **Pandas i CSV:** Biblioteka Pandas umożliwiła łatwe zarządzanie danymi tabelarycznymi oraz ich analizę. Dane takie jak ścieżki do obrazów, wyniki ewaluacji czy statystyki zostały przechowywane i przetwarzane w formacie CSV.
- **NumPy:** Jest podstawowym narzędziem do obliczeń numerycznych w Pythonie. W projekcie NumPy użyto do operacji na tablicach i macierzach, co było istotne podczas preprocessingu obrazów oraz obliczania miar błędu (np. MSE, PSNR).
- **Matplotlib:** Biblioteka do tworzenia wizualizacji danych, używana w projekcie do generowania wykresów ilustrujących przebieg funkcji strat podczas treningu oraz porównywania wyników uzyskanych przez różne architektury.

3 Dane treningowe, dane testowe

3.1 Źródło danych

Dane użyte w projekcie pochodzą z publicznie dostępnych zasobów geograficznych na stronie USGS Explorer. Po utworzeniu konta i zapoznaniu się z funkcjonalnościami portalu umożliwiającymi eksplorację oraz eksport danych, zdecydowano się na wykorzystanie zdjęć lotniczych obejmujących obszar Stanów Zjednoczonych, a dokładniej metropolii Filadelfii.

Zbiór zawierał 20 zdjęć lotniczych o rozdzielczości 8574x8574 pikseli, przedstawiających miejskie krajobrazy. Zastosowanie zdjęć miejskich miało kilka zalet:

- Wysoka dynamika obrazu, wynikająca z różnorodności układu ulic, budynków, parków i innych elementów krajobrazu.
- Większa liczba szczegółów i punktów charakterystycznych w porównaniu do obszarów takich jak otwarte pola czy zbiorniki wodne, co czyni zadanie łatwiejszym dla klasycznych algorytmów takich jak SIFT.
- Zdjęcia w zbiorze były już wcześniej zorganizowane jako nienakładające się, co ułatwiło ich podział i przetwarzanie.

Wybór takich danych pozwalał na testowanie zarówno możliwości klasycznego algorytmu SIFT, jak i porównanie go z podejściem wykorzystującym sieci GAN.

3.2 Charakterystyka danych

Na podstawie wybranych zdjęć wykonano proces podziału na mniejsze obrazy, z których powstało 10800 fragmentów o wymiarach 512x512 pikseli. Każdy fragment miał następujące właściwości:

- **Rozmiar:** 512x512 pikseli.
- **Kanały:** RGB (3 kanały kolorów).
- **Nakładanie się sąsiednich obrazów:** 25% (obrazy w pewnym stopniu zachodziły na siebie, co miało symulować sytuacje, w których widoczne są wspólne obszary pomiędzy sąsiednimi zdjęciami).

Nakładanie obrazów było kluczowe, ponieważ umożliwiał modelowi zrozumienie relacji między fragmentami. Poniżej zamieszczono wizualizację ukazującą przykładowe sąsiednie fragmenty obrazu.



Rysunek 3: Przykład sąsiednich obrazów

3.3 Preprocessing danych

Przygotowanie danych obejmowało następujące kroki:

- **Podział obrazu na mniejsze fragmenty:**
 - Proces podziału dużych obrazów na fragmenty 512x512 pikseli został zrealizowany za pomocą skryptu Pythonowego z użyciem biblioteki OpenCV.
 - Skrypt dzielił obraz na mniejsze części, uwzględniając 25% nakładania się między fragmentami. Dodatkowo dobierał maksymalnie trzech sąsiednich fragmentów dla każdego obrazu (w prawo i w dół), co umożliwiał zachowanie ciągłości nakładania się między dwudziestoma oryginalnymi obrazami.
 - W celu zapewnienia spójnych wymiarów fragmentów skrajne obrazy (prawy i dolny brzeg mapy) otrzymywały *padding* w postaci czarnego wypełnienia, co gwarantowało rozmiar 512x512 pikseli dla każdego fragmentu.
 - Offset dla kolejnych grup obrazów zapewniał kontynuację podziału od miejsca zakończenia poprzedniego fragmentu.
- **Kontrola poprawności:**
 - Proces podziału był półautomatyczny. Użytkownik mógł ręcznie wybrać rząd i kolumnę obrazu do podziału. Ze względu na jednorazowy charakter tej operacji, podział był realizowany partiami w celu zapewnienia dokładności i pełnej kontroli nad procesem.
- **Przygotowanie danych do sieci GAN:**
 - Fragmenty wejściowe zostały odpowiednio przetworzone. Każdy obraz został rozszerzony o *padding*, aby dopasować rozmiar do wymagań panoramicznych:
 - * Rozmiar wejściowy wynosił 512x896 lub 896x512, w zależności od orientacji (horizontalnej lub wertykalnej).

- * Padding dodawano po prawej, lewej, górnej lub dolnej stronie, w zależności od krawędzi obrazu, która wymagała uzupełnienia.
- Powstały dwa osobne katalogi zawierające dane wejściowe: dla panoram horyzontalnych oraz wertykalnych.
- **Podział na zbiory treningowe i testowe:**
 - Ze względu na stosunkowo niewielki rozmiar zbioru (10 800 obrazów), dane zostały podzielone na zbiory:
 - * 90% danych treningowych, używanych w procesie uczenia sieci.
 - * 10% danych testowych, przeznaczonych do oceny modelu.
 - Zbiór testowy pozwalał na dokładne zbadanie jakości generowanych obrazów, w tym na obliczenie wskaźników MSE oraz PSNR.
- **Normalizacja i konwersja do tensorów:**
 - Każdy obraz został przekonwertowany do tensora przy użyciu biblioteki *PyTorch*, a następnie znormalizowany za pomocą funkcji:


```
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```
 - Normalizacja przekształca wartości pikseli do przedziału $[-1, 1]$, co jest zgodne z funkcją aktywacji *Tanh()* w wyjściu generatora, co jest standardowym zabiegiem w treningu sieci GAN.

• **Opcjonalne wykorzystanie binarnej maski:**

- W projekcie rozważano zastosowanie binarnej maski określającej wspólny obszar obu obrazów wejściowych, który miał być poddany operacji scalenia.
- W trakcie testowania architektury badano wpływ dodania takiej informacji na skuteczność modelu. Pomysł ten został zaczerpnięty z literatury [1].

4 Opis funkcji i klas

4.1 plik padder.py

plik `padder.py` realizuje proces dodawania paddingu, generowania masek oraz zarządzania zbiorami danych.

- **`extract_row_col_filename(filename)`**
Funkcja służy do wyodrębniania informacji o rzędzie i kolumnie z nazwy pliku. Przyjmuje nazwę pliku w formacie zawierającym wzorzec `subtile_(\d+)_(\d+)` (np. `subtile_12_34.png`) i zwraca liczby całkowite odpowiadające rzędowi oraz kolumnie. Jeśli nazwa nie pasuje do wzorca, funkcja informuje o błędzie i zwraca `None, None`.
- **`pad_images(input_dir, horizontal_output_dir, vertical_output_dir)`**
Funkcja realizuje dodawanie paddingu do obrazów w celu przygotowania ich do treningu modelu GAN. W szczególności:
 - Otwiera obrazy w formacie PNG z katalogu wejściowego (`input_dir`).
 - Sprawdza, czy rozmiar obrazów wynosi 512x512 pikseli, pomijając obrazy o innych rozmiarach.
 - Dla obrazów zgodnych z wymaganiami generuje:
 - * **Wersje z paddingiem pionowym:** Obrazy o wymiarach 512x896 pikseli z wypełnieniem czarnym (`RGB(0, 0, 0)`). Padding jest dodawany z góry (`_up`) lub z dołu (`_down`), a przetworzone obrazy są zapisywane w katalogu `vertical_output_dir`.
 - * Kod dla paddingu poziomego jest analogiczny do pionowego.
 - Wprowadza mechanizmy kontroli poprawności nazewnictwa oraz ostrzega przed błędami przetwarzania.
- **`create_mask(width, height, output_path)`**
Funkcja generuje binarne maski określające obszar wspólny dwóch obrazów wejściowych. Maski te mogą być wykorzystane w procesie scalenia obrazów:
 - **Parametry:** Przyjmuje wymiary obrazu (`width, height`) oraz ścieżkę zapisu maski (`output_path`).

- **Działanie:**

- * Jeśli szerokość obrazu jest większa od wysokości (horizontalna maska), funkcja ustawia biały obszar w zakresie 128 pikseli w środku szerokości.
- * Dla obrazów wertykalnych maska ustawia biały obszar w środku wysokości.
- Generuje maski jako obrazy w skali szarości (L) i zapisuje je w zadanej ścieżce.

- **move_images(src_dir, dest_dir, percentage=10, seed=42)**

Funkcja realizuje losowe przenoszenie określonego procentu obrazów z jednego katalogu do drugiego. Jest użyteczna podczas podziału danych na zbiory treningowe i testowe:

- **Parametry:**

- * **src_dir:** Ścieżka katalogu źródłowego.
- * **dest_dir:** Ścieżka katalogu docelowego.
- * **percentage:** Procent obrazów do przeniesienia (domyślnie 10%).
- * **seed:** Wartość ziarna dla losowego wyboru obrazów (domyślnie 42).
- Funkcja identyfikuje pliki graficzne (.png, .jpg, .jpeg, .tiff) w katalogu źródłowym, losowo wybiera z nich określony procent i przenosi je do katalogu docelowego.
- Mechanizm unika nadpisywania istniejących plików, wyświetlając ostrzeżenia w przypadku konfliktów.

4.2 plik plotter.py

Plik `plotter.py` zawiera funkcję do wizualizacji strat podczas procesu treningu modelu na podstawie danych zapisanych w pliku CSV. Funkcja jest szczególnie przydatna do monitorowania postępu treningu i analizy zachowania modelu w różnych etapach.

- **plot_losses_with_custom_x(csv_file)**

Funkcja generuje wykres strat (`Loss_G`, `Loss_Fake`, `Loss_Real`, `Loss_D`) na podstawie danych z pliku CSV. Wykres wykorzystuje niestandardową oś X (`Custom_X`), która łączy informacje o epoce (`Epoch`) i kroku (`Step`).

- **Parametry:**

- * **csv_file:** Ścieżka do pliku CSV zawierającego dane dotyczące strat i kroków treningu.

- **Opis działania:**

1. Dane są wczytywane z pliku CSV za pomocą biblioteki `pandas`.
2. Tworzona jest kolumna `Custom_X`, która oblicza niestandardowy indeks X na podstawie wzoru:

$$\text{Custom_X} = (\text{Epoch} - 1) \times 562 + (\text{Step} - 1)$$

Liczba 562 reprezentuje liczbę kroków w jednej epoce.

3. Następnie za pomocą biblioteki `matplotlib` generowany jest wykres przedstawiający zmiany poszczególnych strat (`Loss_G`, `Loss_Fake`, `Loss_Real`, `Loss_D`)

4.3 plik divider.py

Plik `divider.py` służy do generowania mniejszych fragmentów obrazów (subtiles) na podstawie głównych płytka mapy i ich sąsiednich fragmentów. Jest to istotny etap preprocessingu danych do dalszej analizy lub wykorzystania w modelach uczenia maszynowego.

- **load_image(image_dir, row, col)**

Funkcja wczytuje obraz na podstawie jego indeksów wiersza (`row`) i kolumny (`col`). Jeśli plik o określonej nazwie nie istnieje, funkcja zwraca czarny obraz o rozmiarze 8574x8574 pikseli.

- **Parametry:**

- * **image_dir:** Ścieżka do katalogu zawierającego obrazy.
- * **row, col:** Indeksy wiersza i kolumny obrazu.

- **Zwraca:** Wczytany obraz lub czarny placeholder.

- **save_subtile(output_dir, subtile, global_row, global_col)**

Funkcja zapisuje fragment obrazu (**subtile**) do określonego katalogu z nazwą pliku odzwierciedlającą globalne indeksy wiersza i kolumny.

- **Parametry:**

- * **output_dir:** Ścieżka do katalogu wyjściowego.
- * **subtile:** Fragment obrazu do zapisania.
- * **global_row, global_col:** Globalne indeksy wiersza i kolumny.

- **stack_and_generate_subtiles(...)**

Funkcja generuje subtiles (fragmenty obrazu) o określonym rozmiarze (**subtile_size**) i przesunięciu (**stride**) na podstawie aktualnie wybranego obrazu i jego sąsiednich fragmentów.

- Obrazy są łączone w większy obraz, z którego wyodrębniane są subtiles o nakładaniu się 25%.

- **Parametry:**

- * **main_tile, right_tile, bottom_tile, bottom_right_tile:** Główny obraz i jego sąsiedzi.
- * **subtile_size:** Rozmiar subtiles (domyślnie 512x512).
- * **stride:** Wartość przesunięcia (domyślnie 384 piksele).
- * **output_dir:** Ścieżka do katalogu wyjściowego.

- **load_neighbors(image_directory, row, col)**

Funkcja wczytuje główny obraz oraz jego sąsiednie fragmenty (prawy, dolny i prawy-dolny).

- **Parametry:**

- * **image_directory:** Ścieżka do katalogu z obrazami.
- * **row, col:** Indeksy wiersza i kolumny głównego obrazu.

- **Zwraca:** Wczytane obrazy w formie obiektów `numpy.ndarray`.

- **main()**

Główna funkcja programu konfiguruje proces podziału obrazów na subtiles. Użytkownik może wybrać obraz startowy za pomocą indeksów wiersza (**selected_row**) i kolumny (**selected_col**).

- Tworzy katalog wyjściowy, jeśli nie istnieje.
- Ładuje główny obraz i jego sąsiadów za pomocą `load_neighbors`.
- Generuje subtiles za pomocą `stack_and_generate_subtiles`.

- **Obsługa parametrów:**

- **image_directory:** Ścieżka do katalogu z obrazami wejściowymi (domyślnie: "Philadelphia tiles").
- **output_directory:** Ścieżka do katalogu wyjściowego (domyślnie: "Philadelphia subtiles 512").
- **subtile_size:** Rozmiar subtiles (domyślnie: 512).
- **stride:** Przesunięcie dla subtiles (domyślnie: 384).

4.4 plik test.py

Plik `test.py` zawiera kod do testowania wytrenowanego generatora obrazu. W szczególności pozwala na wczytanie dwóch wejściowych obrazów, ich przetworzenie za pomocą generatora, a następnie zapisanie wyniku w formie obrazu.

- **extract_epoch_step(filename)**

Funkcja do ekstrakcji numeru epoki (**epoch**) i kroku (**step**) z nazwy pliku modelu.

- **Parametry:**

- * **filename:** Nazwa pliku modelu (np. `generator_epoch_10_step_200.pth`).

- **Zwraca:** epoch oraz step

- **test_single_input(generator, inputs, device='cuda', output_path='output.png')**

Funkcja wykonująca pojedynczy test na podanym generatorze. Wczytuje obrazy wejściowe, generuje wyjście i zapisuje wygenerowany obraz na dysku.

- **Parametry:**

- * **generator:** Wytrenowany model generatora.
- * **inputs:** Tensor wejściowy (połączone obrazy wejściowe).
- * **device:** Urządzenie, na którym ma działać generator (domyślnie 'cuda').
- * **output_path:** Ścieżka do pliku, w którym zapisany zostanie wynikowy obraz.
- Przebieg funkcji:
 - * Przenosi generator i dane wejściowe na odpowiednie urządzenie.
 - * Generuje obraz wyjściowy w trybie `no_grad()`.
 - * Normalizuje dane wyjściowe (np. z zakresu [-1, 1] do [0, 1]).
 - * Zapisuje wygenerowany obraz jako plik PNG.
- **main()**
Funkcja główna definiująca przepływ programu, w tym wczytanie modelu, danych wejściowych i wykonanie testu.
 - **Wejścia:**
 - * `input_path1, input_path2`: Ścieżki do obrazów wejściowych.
 - * `mask_path`: Ścieżka do opcjonalnej maski obrazu (komentarz w kodzie).
 - **Proces:**
 1. Wczytuje obrazy wejściowe za pomocą `Pillow (Image.open)`.
 2. Przetwarza obrazy za pomocą transformacji (`ToTensor`, `Normalize`).
 3. Łączy obrazy wejściowe w jeden tensor.
 4. Wczytuje wytrenowany generator z katalogu `model_checkpoints`.
 5. Sortuje pliki modelu według epok i kroków.
 6. Ładuje stan generatora z ostatniego pliku modelu.
 7. Wywołuje funkcję `test_single_input`, aby wygenerować obraz wyjściowy.
 - **Wyjście:** Zapisany wygenerowany obraz (`test_output.png`).

- **Obsługa generatora:**
 - Wczytywany generator to model klasy `Generator()`, który musi być zdefiniowany w odpowiednim miejscu w kodzie (np. w innym pliku).
 - Generator ładowany jest z plików o nazwie `generator_epoch_<epoch>_step_<step>.pth`.
- **Transformacje:**
 - Obrazy wejściowe są przekształcane na tensorzy za pomocą transformacji `ToTensor` i normalizowane do zakresu [-1, 1].

4.5 Plik sift.py

Plik `sift.py` zawiera funkcje do wykrywania i dopasowywania cech obrazów z użyciem techniki SIFT (Scale-Invariant Feature Transform), a także funkcje do przetwarzania i łączenia obrazów w kontekście tworzenia dużych mozaik z mniejszych płytek (np. w przypadku obrazów satelitarnych). Plik zawiera również funkcje pomocnicze do obliczania macierzy homografii oraz maskowania obrazów.

- `extract_row_col_filename(filename)` Funkcja do ekstrakcji numerów wiersza i kolumny z nazwy pliku subtile (np. `subtile_0_9.png`).
 - **Parametry:**
 - * `filename`: Nazwa pliku (np. `subtile_0_9.png`).
 - **Zwraca:** (`row, col`) jako liczby całkowite odpowiadające numerom wiersza i kolumny. Jeśli nazwa pliku nie pasuje do wzorca, funkcja zwraca `None, None`.
- `find_neighbours_sub(y_index, x_index, tile_files, input_dir, mode='row')`
Funkcja wyszukująca sąsiadów dla danego indeksu wiersza i kolumny w zbiorze plików.
 - **Parametry:**
 - * `y_index, x_index`: Indeksy wiersza i kolumny, dla których szukamy sąsiadów.
 - * `tile_files`: Lista plików zawierających obrazy.

- * **input_dir:** Ścieżka do katalogu z obrazami.
* **mode:** Tryb wyszukiwania sąsiadów ('row' lub 'col').
– **Zwraca:** Listę plików z obrazami sąsiadów.
- **match_features(des1, des2, method='BF', cross_check=True)**
Funkcja dopasowująca cechy pomiędzy dwoma zestawami deskryptorów za pomocą wybranej metody.

 - **Parametry:**
 - * **des1, des2:** Deskryptory cech dla dwóch obrazów.
 - * **method:** Metoda dopasowania ('BF' dla `BFMatcher` lub 'FLANN' dla `FlannBasedMatcher`).
 - * **cross_check:** Flaga wskazująca, czy używać cross-checking podczas dopasowywania.
 - **Zwraca:** Posortowaną listę dopasowanych cech.

- **compute_homography(kp1, kp2, matches)**
Funkcja obliczająca macierz homografii na podstawie dopasowanych punktów kluczowych.

 - **Parametry:**
 - * **kp1, kp2:** Listy punktów kluczowych dla dwóch obrazów.
 - * **matches:** Dopasowania punktów kluczowych.
 - **Zwraca:** Macierz homografii (H) oraz maskę (`mask`).

- **mask_images(img1, img2, mode)**
Funkcja stosująca maskowanie na dwóch obrazach w zależności od trybu łączenia ('row' dla poziomego łączenia, 'col' dla pionowego).

 - **Parametry:**
 - * **img1, img2:** Obrazy do zamaskowania.
 - * **mode:** Tryb łączenia ('row' lub 'col').
 - **Zwraca:** Zamaskowane wersje `img1` i `img2`.

- **validate_stitched_dimensions(stitched_width, stitched_height, mode)**
Funkcja weryfikująca, czy wymiary łączonego obrazu odpowiadają oczekiwany wartościom na podstawie trybu.

 - **Parametry:**
 - * **stitched_width, stitched_height:** Wymiary łączonego obrazu.
 - * **mode:** Tryb łączenia ('row' lub 'col').
 - **Zwraca:** `True` jeśli wymiary są zgodne z oczekiwaniemi, w przeciwnym przypadku `False`.

- **compute_image_corners(img, homography=None)**
Funkcja obliczająca rogi obrazu, opcjonalnie transformując je za pomocą macierzy homografii.

 - **Parametry:**
 - * **img:** Obraz, którego rogi mają zostać obliczone.
 - * **homography:** Opcjonalna macierz homografii do zastosowania na rogach obrazu.
 - **Zwraca:** Współrzędne rogów obrazu (po transformacji homografią, jeśli podana).

- **monster_detector(img1, img2)**
Funkcja do wykrywania punktów kluczowych za pomocą kilku różnych detektorów (Harris, Shi-Tomasi, Dense Grid, FAST, MSER) i łączenia ich w jeden zestaw punktów kluczowych.

 - **Opis:** Detektor łączy różne techniki wykrywania punktów kluczowych, aby zwiększyć dokładność detekcji.
 - **Zwraca:** Dopasowane punkty kluczowe i ich opisy z dwóch obrazów.

- **stitch_images_sift(img1, img2, mode)**
Funkcja łącząca dwa obrazy na podstawie dopasowanych cech SIFT. W przypadku problemów z homografią, próbuje zastosować alternatywne metody. Jeśli wszystkie próby zawiodą, stosuje prostą konkatenację obrazów.

- **Opis:** Funkcja ta próbuje najpierw dopasować obrazy za pomocą standardowego SIFT, a jeśli to się nie powiedzie, stosuje alternatywną kosztowną metodę. W ostateczności używa metody prostego łączenia.
 - **Zwraca:** Złożony obraz wynikowy (po dopasowaniu lub prostym połączeniu).
- **process_tiles_sift(input_dir, output_dir, mode='row')**
Funkcja ta przetwarza obrazy zapisane w określonym katalogu wejściowym, próbując połączyć je w większe obrazy na podstawie detekcji punktów kluczowych. Działa na obrazach w formacie TIFF lub PNG.
 - **Opis:** Przechodzi przez wszystkie obrazy w katalogu wejściowym, wczytuje je, a następnie dla każdej płytki próbuje znaleźć jej sąsiadów. Następnie obrazy są łączone w jeden obraz za pomocą wcześniej zdefiniowanej funkcji `stitch_images_sift`.
 - **Zwraca:** Zapisane obrazy połączenia w katalogu wyjściowym.

4.6 pliki gan.py, gan_skip.py itp.

Plik `gan.py` zawiera implementację Generative Adversarial Network (GAN) służącej do składania obrazów przy użyciu PyTorch. Projekt wykorzystuje sieć generatora do generowania syntetycznych obrazów oraz sieć dyskryminatora do oceny autentyczności wygenerowanych obrazów. W pliku zaimplementowano również funkcje pomocnicze, klasę niestandardowego zbioru danych oraz główną pętlę treningową.

- **weights_init_normal(m)**
Funkcja inicjalizuje wagi warstw sieci, aby poprawić stabilność i szybkość uczenia.
 - **Opis:** W zależności od typu warstwy, inicjalizuje wagi za pomocą rozkładu normalnego (`nn.init.normal_`) i ustawia wartości biasów na zero. Obsługuje warstwy konwolucyjne, BatchNorm oraz InstanceNorm.
- **Generator()**
Klasa definiująca generator dla GAN (Generative Adversarial Network), który tworzy obrazy wyjściowe na podstawie obrazów wejściowych.
 - **Opis:** Generator implementuje architekturę typu *encoder-decoder* z połączaniami skip-connection. Składa się z czterech warstw enkodera, warstwy wąskiego gardła (*bottleneck*), czterech warstw dekodera oraz warstwy wyjściowej, która normalizuje dane do przedziału $[-1, 1]$.
 - **Zwraca:** Tensor obrazu o rozmiarze $[3, H, W]$.
- **Discriminator()**
Klasa definiująca dyskryminator dla GAN, którego zadaniem jest odróżnianie obrazów rzeczywistych od wygenerowanych.
 - **Opis:** Dyskryminator to sieć konwolucyjna z użyciem normalizacji spektralnej (`nn.utils.spectral_norm`) w celu poprawy stabilności. Przyjmuje obrazy jako wejście i zwraca ocenę dla każdego obrazu.
 - **Zwraca:** Płaski tensor, zawierający ocenę dla każdego obrazu wejściowego.
- **StitchDataset(input_dir, stitched_dir, transform=None, mode='row')**
Klasa datasetu do załadowania par obrazów wejściowych i odpowiadających im obrazów połączonych.
 - **Opis:** Dataset przetwarza obrazy w katalogach wejściowym i wyjściowym, łącząc pary obrazów zgodnie z nazwami plików. Obsługuje dwa tryby: `row` (łączenie poziome) i `column` (łączenie pionowe).
 - **Zwraca:** Każdy element to para (`inputs, stitched_img`), gdzie `inputs` to tensor wejściowy złożony z dwóch obrazów i maski, a `stitched_img` to obraz wynikowy.
- **train_gan(generator, discriminator, dataloader, num_epochs=10, device='cuda', last_epoch=0, save_dir="checkpoints", log_file="training_log_paper.csv")**
Funkcja trenuje sieć GAN, korzystając z podanych generatora i dyskryminatora.
 - **Opis:** Przechodzi przez zestaw danych, trenując na przemian dyskryminator i generator. Używa strat opartych na funkcji `nn.BCEWithLogitsLoss` oraz technik akumulacji gradientów i mieszanej precyzji (`torch.amp.GradScaler`).
 - **Zwraca:** Zapisuje modele generatora i dyskryminatora do katalogu `save_dir` oraz zapisuje logi w pliku CSV.

5 Architektury

5.1 Wprowadzenie do architektur modelu

Projektowanie architektur sieci neuronowych dla GAN (Generative Adversarial Networks) wymaga starannego rozważenia wielu decyzji, takich jak inicjalizacja wag, wybór warstw normalizujących i aktywacyjnych. Sieci GAN są znane ze swojej złożoności w procesie uczenia, co wynika z interakcji między dwoma komponentami: generatorem i dyskryminatorem.

Trudności w uczeniu GAN:

- **Niebezpieczeństwo dominacji dyskryminatora:** Gdy dyskryminator zbyt szybko uczy się odróżniać obrazy rzeczywiste od generowanych, generator nie otrzymuje wystarczająco dokładnych informacji zwrotnych, co może prowadzić do zahamowania procesu uczenia.
- **Mode collapse:** Generator może nauczyć się wytwarzanie ograniczony zestaw obrazów, które skutecznie oszukują dyskryminator, ale nie odzwierciedlają pełnej różnorodności danych.
- **Niestać gradientów:** Wzajemna optymalizacja generatora i dyskryminatora może prowadzić do problemów z eksplodującymi lub zanikającymi gradientami.

Poniżej przedstawiono decyzje podjęte w celu złagodzenia tych problemów i poprawy stabilności uczenia.

5.1.1 Inicjalizacja wag

Wagi są inicjalizowane za pomocą metody `weights_init_normal`, co zapewnia stabilność trenowania sieci oraz umożliwia szybszą zbieżność:

- **Rozkład normalny wag:** Inicjalizacja wag konwolucyjnych (`Conv`) rozkładem normalnym ($\mu = 0, \sigma = 0.02$) ogranicza ryzyko zanikającego lub eksplodującego gradientu w trakcie propagacji wstecznej.
- **Stałe wartości dla biasów:** Inicjalizacja biasów stałą wartością 0 minimalizuje początkowe przesunięcia, umożliwiając bardziej zrównoważone uczenie.
- **Normalizacja dla BatchNorm/InstanceNorm:** Wagi w warstwach normalizacyjnych (`BatchNorm`, `InstanceNorm`) są inicjalizowane tak, aby miały średnią 1.0 i odchylenie 0.02, co przyspiesza stabilizację wartości normalizowanych w pierwszych iteracjach.

5.1.2 Warstwy normalizujące

Normalizacja jest podstawowym elementem stabilizacji procesu uczenia i różni się w generatorze i dyskryminatorze:

- **InstanceNorm dla generatora:** Generator korzysta z `InstanceNorm`, która normalizuje cechy każdej instancji (obrazu) oddzielnie. Dzięki temu:
 - Redukowana jest zależność między obrazami w trakcie normalizacji.
 - Zwiększa jest zdolność do generowania różnorodnych obrazów, co jest szczególnie istotne w aplikacjach takich jak stylizacja obrazów.
- **Spektralna normalizacja dla dyskryminatora:** Dyskryminator wykorzystuje spektralną normalizację wag (`SpectralNorm`), która stabilizuje proces uczenia przez kontrolowanie norm wag w warstwach konwolucyjnych. Zapobiega to eksplozji gradientów, jednocześnie poprawiając zdolność modelu do identyfikowania obrazów rzeczywistych i wygenerowanych.

5.1.3 Warstwy aktywacyjne

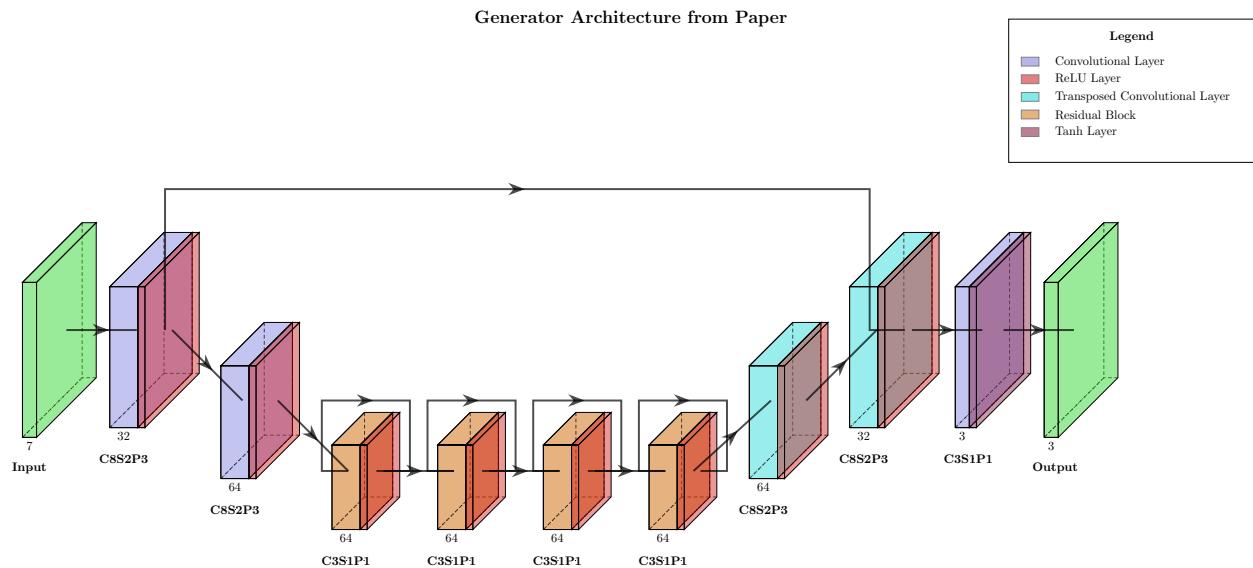
Wybór warstw aktywacyjnych odgrywa kluczową rolę w modelowaniu nieliniowości:

- **ReLU w generatorze:** Funkcja `ReLU` (Rectified Linear Unit) jest używana w generatorze z uwagi na jej zdolność do wprowadzenia nieliniowości oraz efektywność obliczeniową. Wykorzystywana w trybie `inplace=True`, co redukuje zużycie pamięci.
- **LeakyReLU w dyskryminatorze:** Funkcja `LeakyReLU` jest stosowana w dyskryminatorze, co pozwala uniknąć problemu "martwych neuronów" dzięki przepuszczaniu niewielkiego gradientu w przypadku ujemnych wartości (np. $\alpha = 0.2$).

5.2 Architektura pierwsza - miniatura z pracy naukowej

W pracy [1] zaproponowano sieć GAN z trzema warstwami konwolucyjnymi w enkoderze, 16 blokami resztkowymi w wąskim gardle oraz dwoma transponowanymi konwolucjami w dekoderze i jedną zwykłą konwolucją. Jednak ograniczenia sprzętowe – 8 GB VRAM – uniemożliwiły przetwarzanie nawet batcha o rozmiarze 1. W celu redukcji wymagań pamięciowych zmniejszono liczbę bloków resztkowych w wąskim gardle z 16 do 4, co pozwoliło na trenowanie modelu przy ograniczonych zasobach.

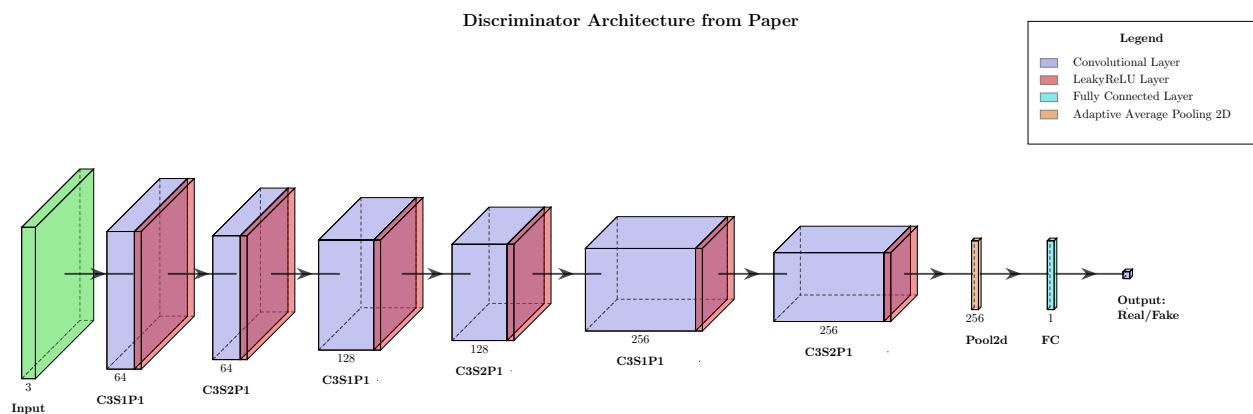
5.2.1 Generator



Rysunek 4: Architektura generatora wzorowana na pracy naukowej

Generator wykorzystuje skip-connection pomiędzy wyjściem z pierwszego bloku enkodera a wejściem drugiego bloku dekodera. Na wejściu model stosuje binarną maskę (7 kanałów). Filtry konwolucyjne o rozmiarze 3 są używane w warstwach resztkowych, podczas gdy w enkoderze i dekoderze (z wyjątkiem ostatniego bloku dekodera) stosowane są filtry o rozmiarze 8. Z powodu długiego czasu uczenia batcha, ta pierwotna architektura została porzucona.

5.2.2 Dyskryminator



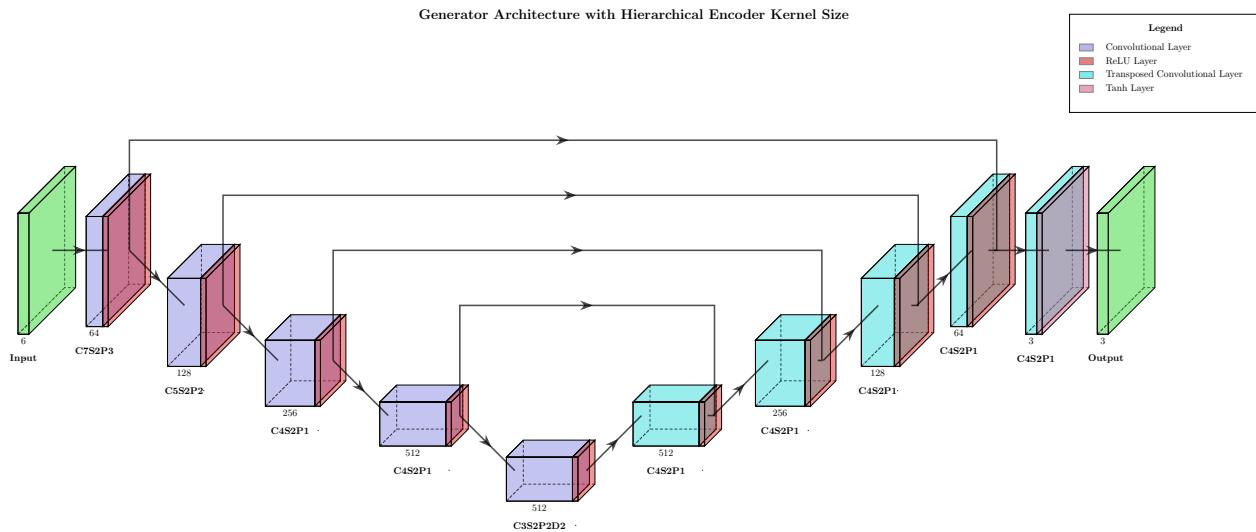
Rysunek 5: Architektura dyskryminatora wzorowana na pracy naukowej

Architektura dyskryminatora jest klasycznym rozwiązaniem w przypadku GAN. Zastosowanie adaptive average pooling umożliwia przetwarzanie obrazów o dowolnym rozmiarze.

5.3 Architektura druga - hierarchiczność rozmiaru filtrów w enkoderze

Przy drugiej architekturze postanowiono częściowo wykorzystać budowę UNet w postaci dużej liczby skip-connection oraz hierarchiczność rozmiaru filtrów tylko w enkoderze. Dodatkowo zrezygnowano z dodatkowej maski binarnej.

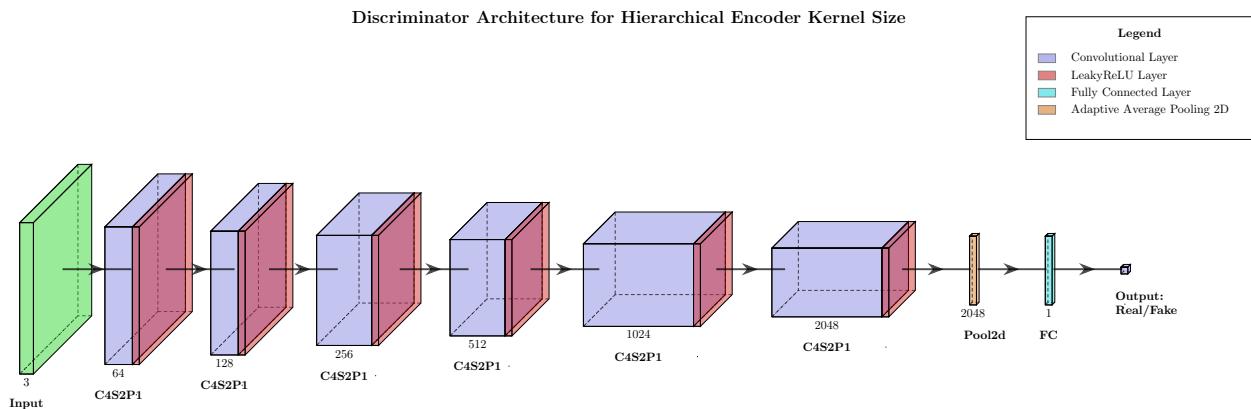
5.3.1 Generator



Rysunek 6: Architektura generatora z hierarchiczną rozmiarem filtrów w enkoderze

Można zauważyć, że liczba filtrów w generatorze jest potęgami dwójki im głębszej w literze U. Jest to typowy zabieg dobrze skalujący się na GPU. Bottleneck posiada warstwę konwolucyjną z dylatacją $D=2$, co pozwala na zwiększenie pola recepcyjnego bez znaczącego zwiększenia liczby parametrów. Dylatacja ta umożliwia modelowi lepsze uchwycenie zależności przestrzennych w danych wejściowych.

5.3.2 Dyskryminator

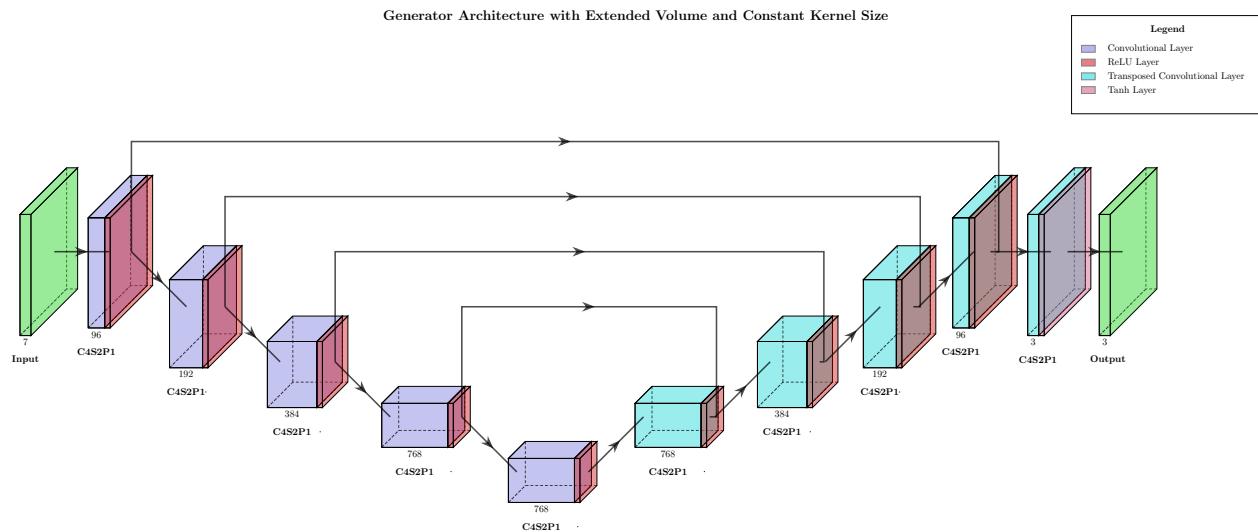


Rysunek 7: Architektura dyskryminatora z hierarchiczną rozmiarem filtrów w enkoderze

Architektura dyskryminatora różni się od poprzedniej wyłącznie zwiększeniem liczby filtrów wraz z pogłębianiem się sieci. Wprowadzenie większej liczby filtrów w kolejnych warstwach pozwala na lepsze przechwytywanie szczegółów i złożonych cech w obrazach, co może prowadzić do bardziej efektywnego rozróżniania między obrazami rzeczywistymi a generowanymi.

5.4 Architektura trzecia - zwiększenie pojemności generatora przy stałym rozmiarze filtrów

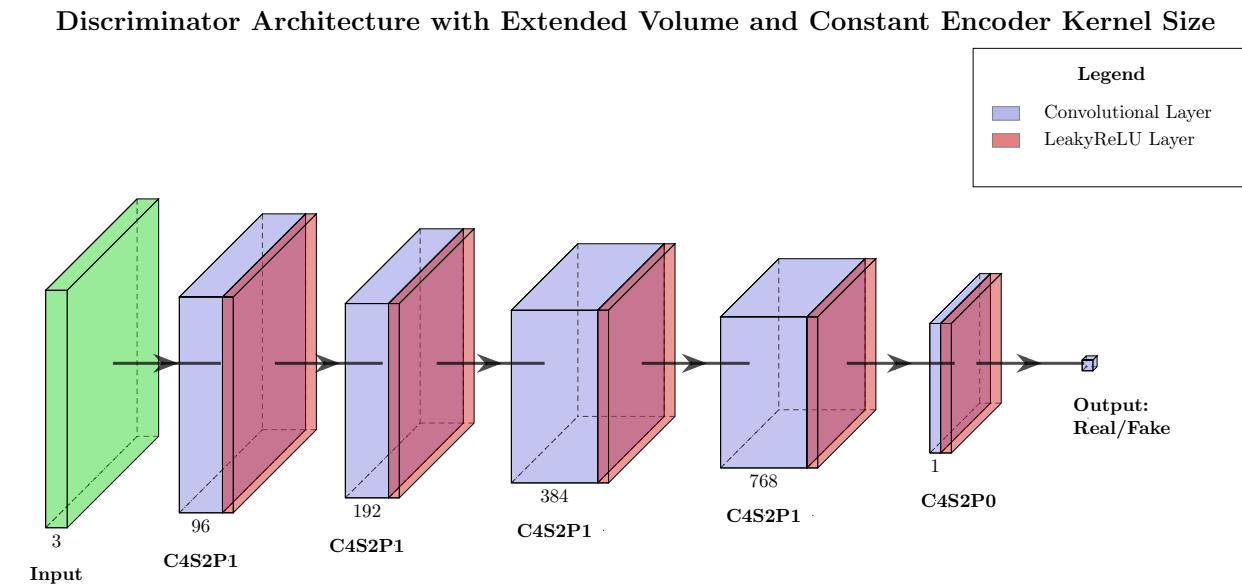
5.4.1 Generator



Rysunek 8: Architektura generatora z powiększoną pojemnością przy stałym rozmiarze filtrów

Stał rozmiar filtrów odchodzi od architektury UNet, ale pozostawia połączenia skip. Zwiększenie pojemności modelu w teorii pozwoli na lepsze zrozumienie relacji między obrazami. W tym modelu wykorzystano maskę binarną, która powinna pomagać w ukierunkowaniu generacji obrazów na określone obszary.

5.4.2 Dyskryminator

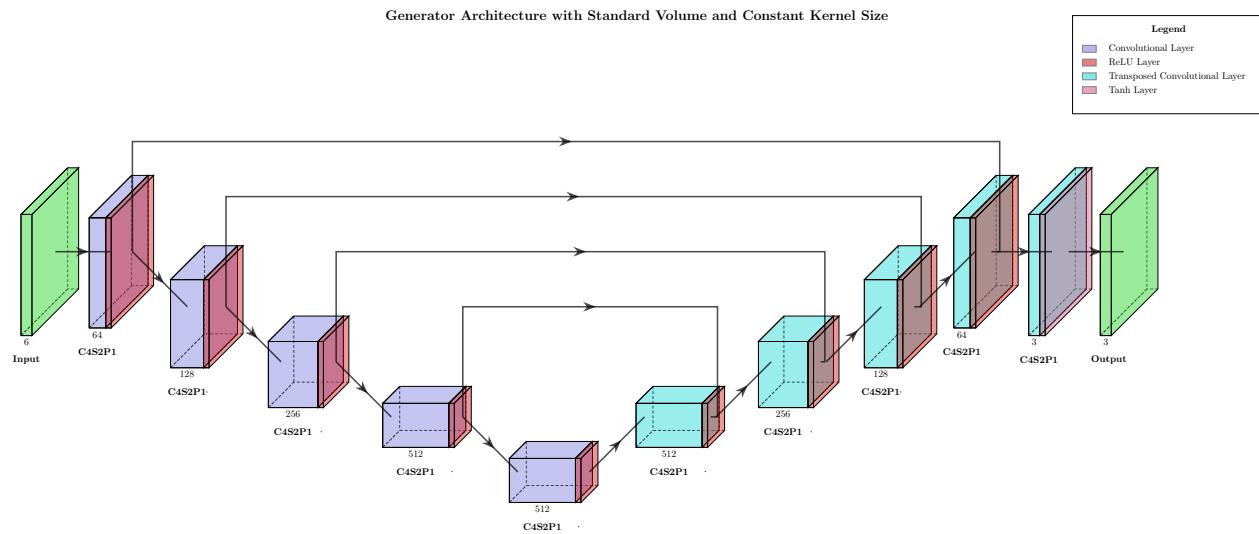


Rysunek 9: Architektura dyskryminatora z powiększoną pojemnością przy stałym rozmiarze filtrów

Jak można zauważyć, dyskryminator posiada o jedną warstwę konwolucyjną mniej oraz nie posiada warstwy poolingowej ani fully connected. Nadrabia za to powiększoną liczbą filtrów, które pozwalają na lepsze przechwytywanie szczegółów i złożonych cech w obrazach. Wprowadzenie większej liczby filtrów w kolejnych warstwach może prowadzić do bardziej efektywnego rozróżniania między obrazami rzeczywistymi a generowanymi.

5.5 Architektura czwarta - standardowa pojemność generatora przy dodatkowej warstwie w dyskryminatorze

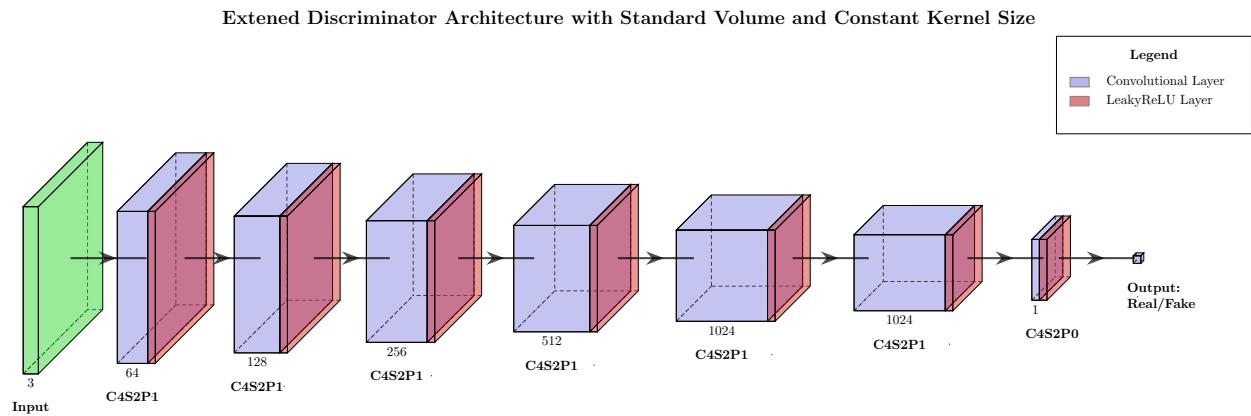
5.5.1 Generator



Rysunek 10: Architektura generatora ze standardową pojemnością przy stałym rozmiarze filtrów

W generatorze nie wykorzystano maski binarnej. Architektura ta zachowuje standardową pojemność, co może być korzystne w przypadku ograniczonych zasobów obliczeniowych.

5.5.2 Dyskriminator



Rysunek 11: Powiększona architektura dyskryminatora ze standardową pojemnością przy stałym rozmiarze filtrów

Dodanie dwóch bliźniaczych warstw o liczbie filtrów 1024 pogłębiło model kosztem pojemności modelu. Wprowadzenie dodatkowych warstw pozwala na lepsze przechwytywanie złożonych cech w obrazach, co może prowadzić do bardziej efektywnego rozróżniania między obrazami rzeczywistymi a generowanymi.

6 Proces treningu

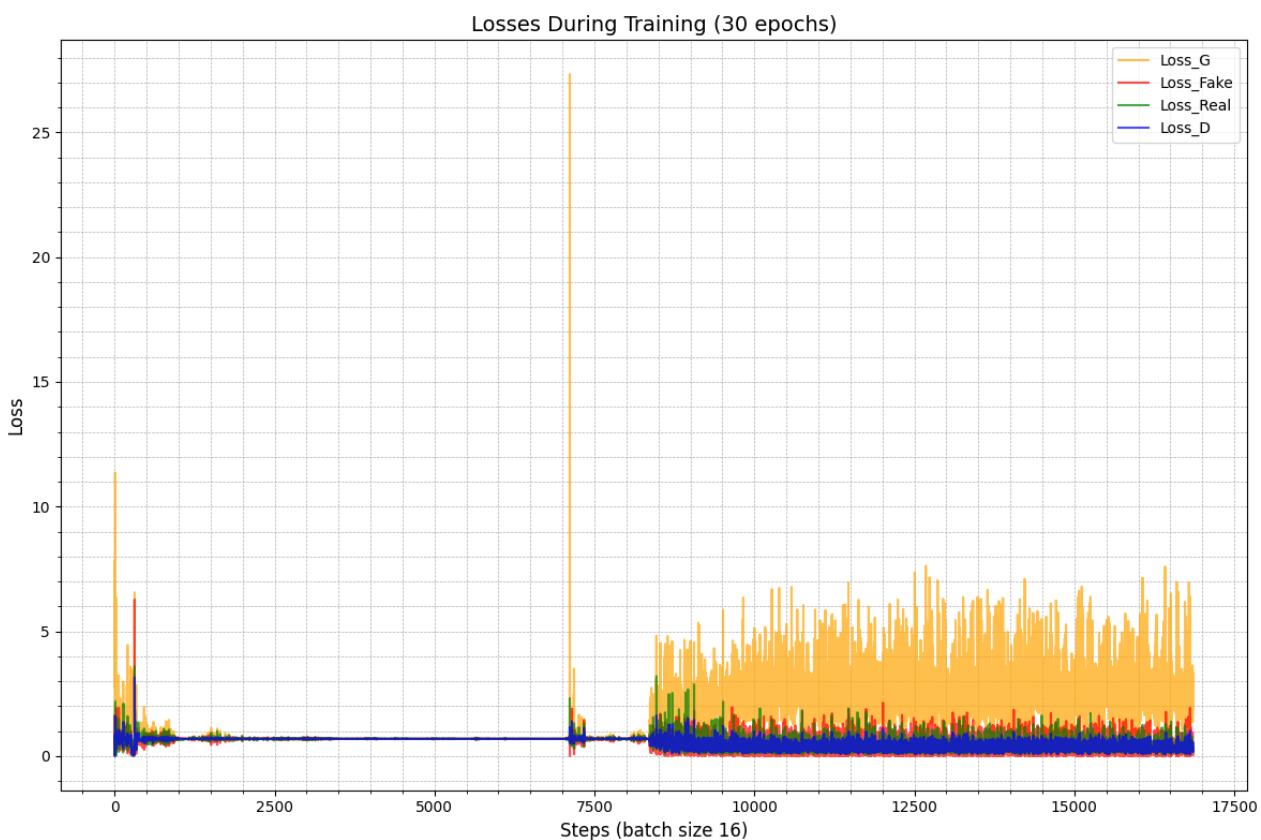
Co epokę lub pół epoki zapisywano checkpointy dla generatora i dyskryminatora. Wartości funkcji strat dla generatora, dyskryminatora dla obrazu syntetycznego, obrazu docelowego oraz średniej arytmetycznej z obu zapisywano do pliku CSV co 16 próbek (step). Trening sieci GAN nie należy do szybkich procesów, a czas potrzebny na osiągnięcie zadowalających wyników jest trudny do oszacowania. Sposobami na monitorowanie postępów są wizualna ocena generowanych obrazów oraz analiza przebiegów funkcji strat.

6.1 Parametry treningu

Większość badanych architektur była trenowana przez 30-90 epok. Najbardziej obiecujące efekty zaobserwowano dla architektury trzeciej. Jako funkcję straty wykorzystano binarną entropię krzyżową (`nn.BCEWithLogitsLoss()` w PyTorchu). Optymalizator Adam został zastosowany z początkowym współczynnikiem uczenia wynoszącym 0.0002 lub 0.0001 oraz współczynnikami beta równymi (0.5, 0.999). Takie parametry sprzyjają stabilizacji treningu, kosztem wolniejszej zbieżności. Starano się utrzymać rozmiar batcha na poziomie 16, jednak dla niektórych architektur konieczne było zastosowanie sumowania gradientów z dwóch batchów po 8 próbek. Wykorzystanie narzędzia `torch.amp` pozwoliło na zaoszczędzenie pamięci VRAM.

6.2 Problemy napotkane podczas treningu

Na początku treningu sieć często otrzymywała gradienty znacznie przekraczające typowe wartości. Następnie oczekiwano obniżenia wartości funkcji straty dla dyskryminatora do poziomu poniżej 0.69 (co w skali logarytmicznej odpowiada 50% poprawnej klasyfikacji) oraz wzrostu wartości funkcji straty dla generatora. Przebiegi obu funkcji strat były nieregularne, a amplituda wału zależała od współczynnika uczenia. W trakcie pracy nad projektem zauważono, że stopniowe zmniejszanie współczynnika uczenia wraz z postępem treningu może pomóc w rozwiązaniu niektórych problemów, takich jak pojawianie się czarnych lub białych plam na generowanych obrazach. Należy jednak pamiętać, że zbyt duży współczynnik uczenia może prowadzić do niestabilności treningu i utrudniać precyzyjne dostosowanie wag w późniejszych etapach. Jednocześnie brak jest dobrej metryki, która wskazywałaby optymalny moment na redukcję współczynnika uczenia. Wartości takie jak IS score (Inception Score) czy FID (Fréchet Inception Distance) mogą być pomocne, ale ich uzyskanie jest kosztowne obliczeniowo i nie zawsze dostarcza jednoznacznego wskaźnika.



Rysunek 12: Wykres funkcji strat dla architektury drugiej

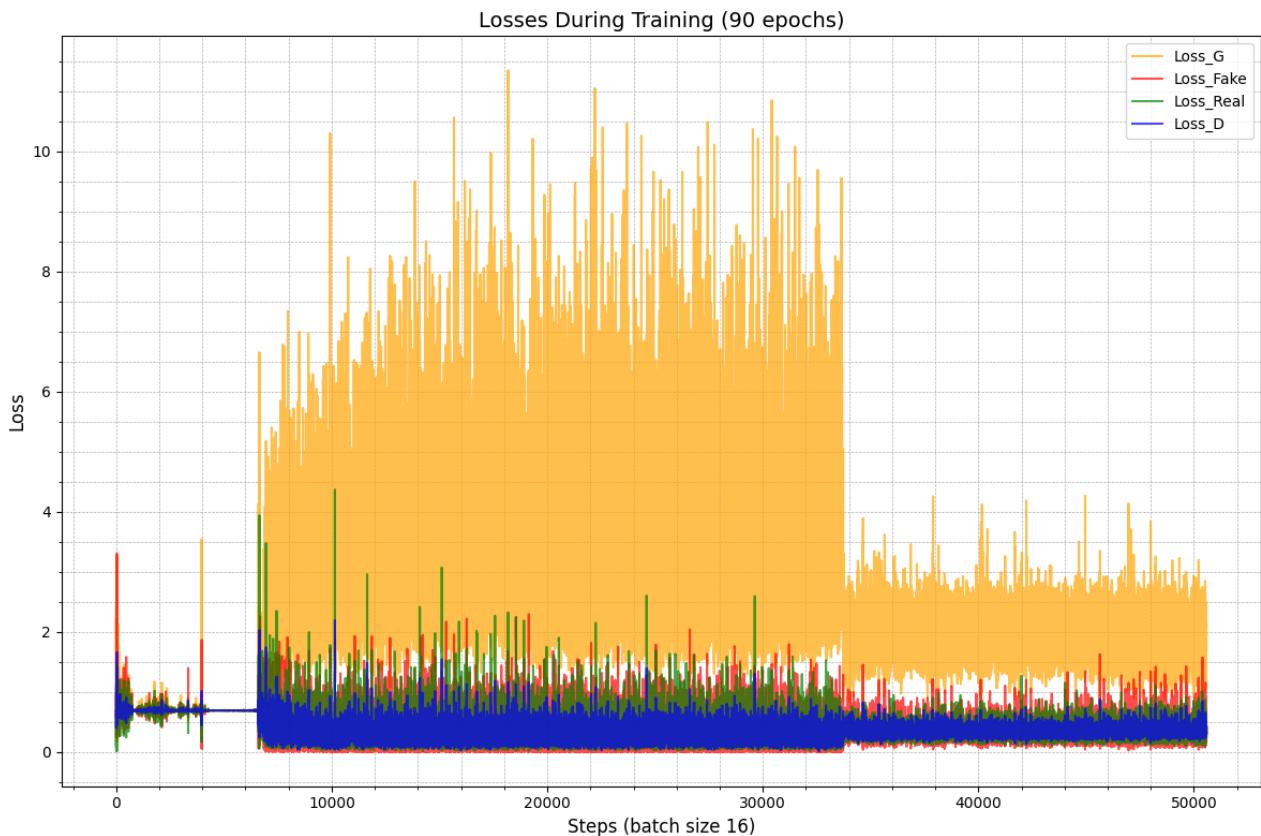
Wykres przedstawia jak niestabilny jest proces nauki sieci GAN. Na początku treningu, gdy generator i dyskryminator są jeszcze nie wytrenowane, wartość straty (loss_fake lub loss_real) będzie bliska 0.69. Oznacza to, że model nie potrafi jeszcze rozróżnić między prawdziwymi a fałszywymi danymi i działa na poziomie zgadywania. Następnie zauważamy gwałtowny skok wartości funkcji straty dla generatora, przy jednoczesnym spadku dla obrazów syntetycznych do wartości bliskim zera. Może to oznaczać:

- Zbyt silny dyskryminator: Jeśli dyskryminator jest zbyt dobry w rozpoznawaniu fałszywych obrazów, generator otrzymuje duże gradienty, co prowadzi do gwałtownego wzrostu wartości straty.

- Eksplozję gradientów: Zbyt agresywne aktualizacje wag generatora mogą prowadzić do niestabilności.
- Zmianę dynamiki treningu: W pewnym momencie dyskryminator mógł się znacząco poprawić, co zaburzyło równowagę między modelami.

W ostateczności model zaczął oscylować w wartości dyskryminatora od 0.3 do 0.5. Oscylacje wartości straty dyskryminatora w tym zakresie wskazują na równowagę między generatorem a dyskryminatorem. Oznacza to, że:

- Dyskryminator jest na tyle silny, że potrafi skutecznie odróżniać prawdziwe obrazy od fałszywych, ale nie jest na tyle dominujący, aby całkowicie zahamować naukę generatora.
- Generator stopniowo uczy się oszukiwać dyskryminator, co prowadzi do wzrostu wartości loss_fake i spadku loss_real.



Rysunek 13: Wykres funkcji strat dla architektury czwartej

Na wykresie przebiegu wartości funkcji strat zauważamy podobne zachowanie, jak dla architektury drugiej. Jedyną różnicą są zdecydowanie większe wartości strat dla generatora. Od 60. epoki zredukowano współczynnik uczenia z 0.0001 do 0.00005, co poprawiło proces nauki. Dyskryminator lepiej zaczął odróżniać obrazy rzeczywiste, co zmniejszyło zakres wartości straty dyskryminatora od dołu. Generator miał większe trudności z oszukiwaniem dyskryminatora:

- Wyższe wartości Loss_G wskazują, że generator nie był w stanie skutecznie oszukiwać dyskryminatora, co prowadziło do większych błędów.
- Może to wynikać z różnic w architekturze dyskryminatora oraz generatora.

Mniejszy współczynnik uczenia pozwolił na bardziej precyzyjne aktualizacje wag, co zmniejszyło fluktuacje wartości strat.

6.3 Nieintuicyjność przebiegu funkcji strat

Uczenie sieci GAN nie jest łatwym zadaniem ze względu na rywalizacyjny charakter treningu. Przebiegi funkcji strat dla generatora i dyskryminatora często są niestabilne i trudne do interpretacji. W przeciwieństwie do tradycyjnych sieci neuronowych, gdzie spadek funkcji straty zazwyczaj wskazuje na poprawę modelu, w przypadku

GAN-ów niskie wartości funkcji straty nie zawsze oznaczają lepszą jakość generowanych obrazów. Na przykład, jeśli dyskryminator stanie się zbyt silny, funkcja straty generatora może gwałtownie wzrosnąć, co niekoniecznie przekłada się na pogorszenie jakości generowanych obrazów. Dlatego ważne jest równoczesne monitorowanie zarówno wartości funkcji strat, jak i jakości generowanych obrazów.



Rysunek 14: Wygenerowany obraz przez 30 epokę architektury drugiej

Brak maski binarnej często wiąże się z inwersją koloru dróg dla jednego lub obu obrazów wejściowych. Można zauważać istniejące czarne plamy na brzegach obrazu i ogólnie intensywniejsze barwy na brzegach. Jest to spowodowane tym, że generator nie jest w pełni wytrenowany, aby dokładnie odwzorować szczegóły na krawędziach obrazu, co prowadzi do artefaktów i niejednorodności w kolorystyce. W miarę postępu treningu, generator powinien stopniowo poprawiać jakość generowanych obrazów, redukując te niepożądane efekty.



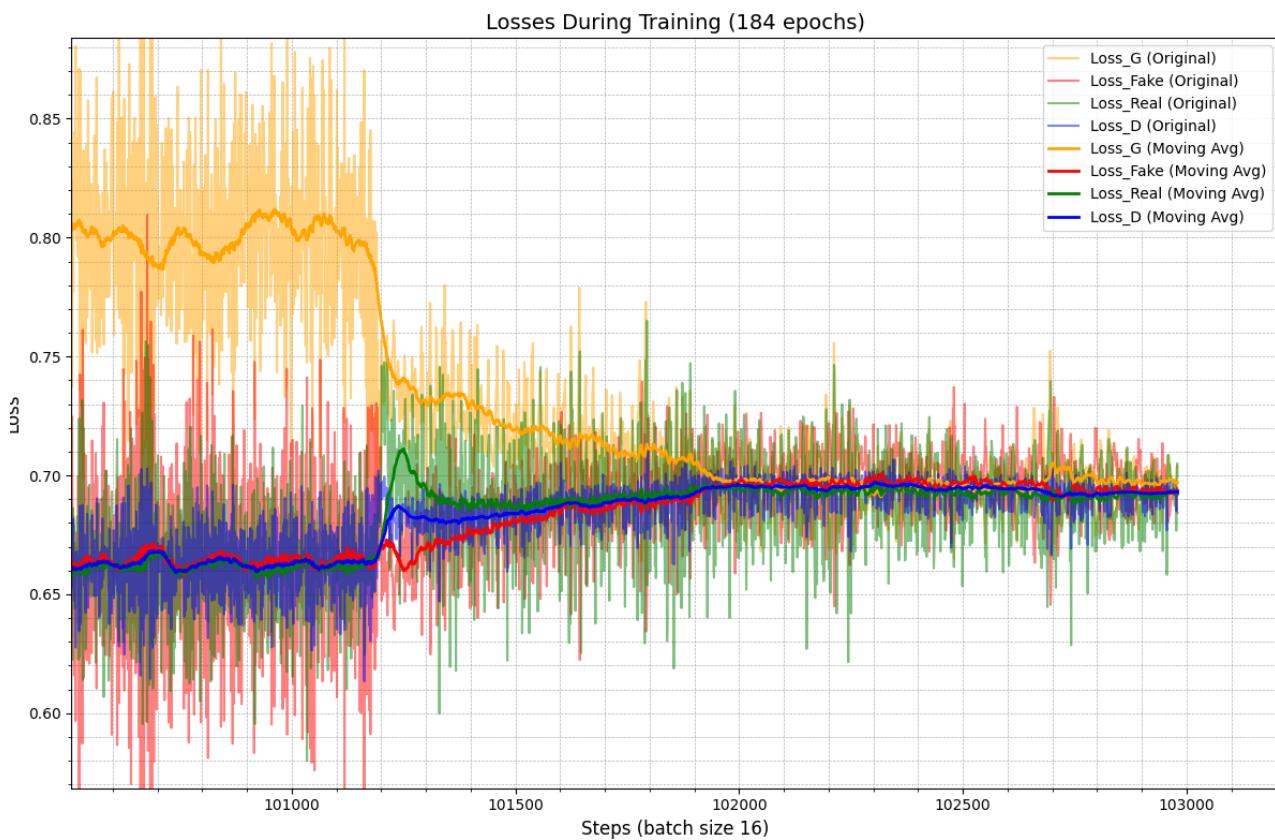
Rysunek 15: Wygenerowany obraz przez 90 epokę architektury czwartej

Dla 90. epoki architektury czwartej brak maski binarnej podkreśla jej kluczową rolę w procesie łączenia obrazów. Można zaobserwować lepsze blendowanie się wspólnego obszaru obu obrazów, co świadczy o postępie w nauce generatora. Niemniej jednak sieć nadal boryka się z problemami, takimi jak czarne rogi oraz intensywne barwy na brzegach, co sugeruje, że generator nie jest jeszcze w pełni wytrenowany do precyzyjnego odwzorowania szczegółów na krawędziach obrazu.

W kontekście projektu poszukiwano architektury, która w stosunkowo krótkim czasie byłaby w stanie generować obrazy o zadowalającej jakości. Istnieje prawdopodobieństwo, że poprzednie dwie architektury mogłyby osiągnąć lepsze wyniki przy dłuższym treningu. Jednak czas potrzebny na przetestowanie tej hipotezy przekracza dostępne zasoby obliczeniowe i czasowe.

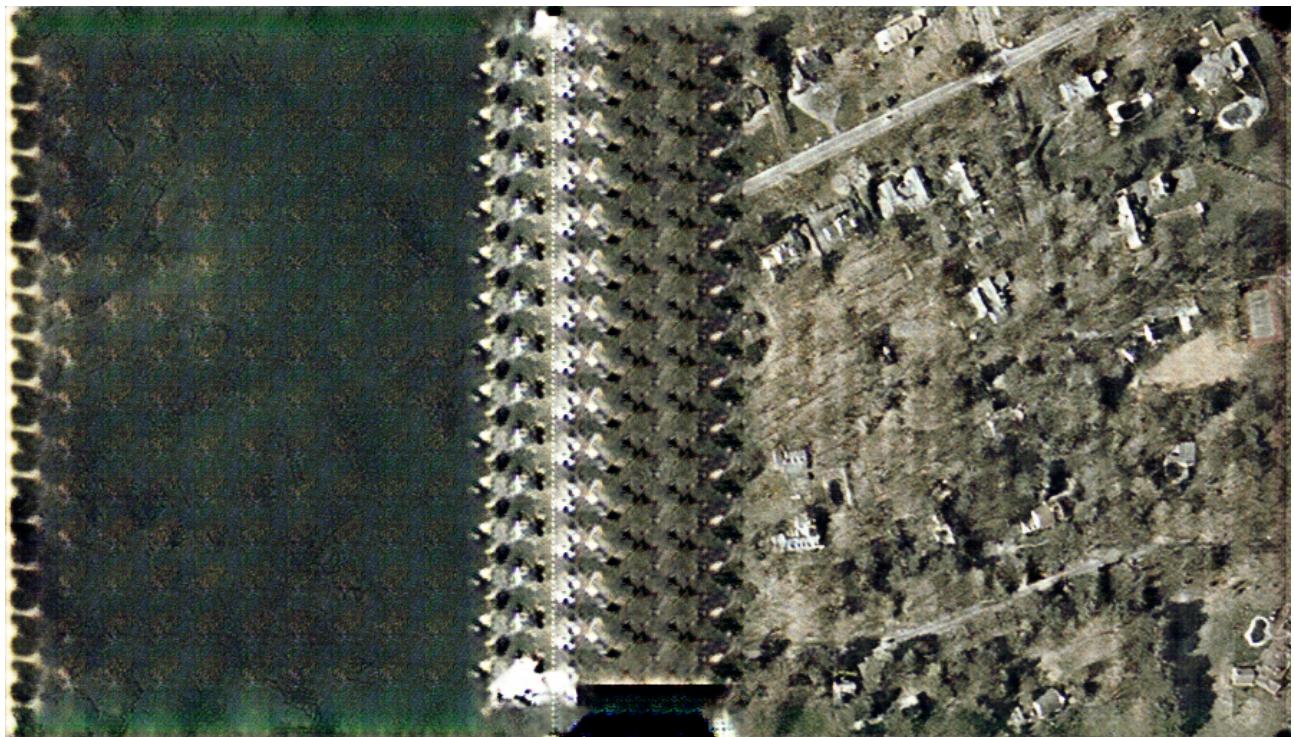
6.4 Mode collapse w architekturze trzeciej

Po 180 epokach zdecydowano o dwukrotnym zmniejszeniu współczynnika uczenia z wartości 0.0001 do 0.00005. Decyzja ta została podjęta na podstawie wcześniejszego sukcesu, gdy współczynnik uczenia został zredukowany z 0.0002 do 0.0001 w 90. epoce. Poniższy wykres przedstawia zbliżenie na przebieg funkcji strat, w którym zaobserwowano zjawisko mode collapse:



Rysunek 16: Zbliżenie na fragment wykresu przedstawiający przebieg funkcji strat dla współczynnika uczenia 0.00001. Szerokość okna średniej ruchomej dla 50 pomiarów.

Jak można zauważyc, dyskryminator nie jest w stanie odróżnić obrazów syntetycznych od rzeczywistych. Może to wskazywać na dwie możliwości: albo doszło do konwergencji, a generator osiągnął poziom efektywności porównywalny z algorytmem SIFT, albo generator nauczył się generować obrazy, które skutecznie oszukują dyskryminator. Poniżej przedstawiono obraz wygenerowany w epoce 184:



Rysunek 17: Wygenerowany obraz dla epoki 184

Można zaobserwować, że generator nauczył się generować wzorce, które mogą występować w rzeczywistych obrazach, jednak zaczął je powtarzać w miejscu jednego z obrazów wejściowych. W związku z tym postanowiono cofnąć się do epoki 180 i ustawić współczynnik uczenia dla dyskryminatora na wartość 0.00005. Doświadczenie to sugeruje, że wprowadzenie hierarchicznosci rozmiaru filtrów w dyskryminatorze mogłoby przynieść korzyści, potencjalnie zmniejszając ryzyko wystąpienia mode collapse.

6.5 Wybór najlepszej wersji modelu z treningu

Wybór najlepszej wersji modelu w trakcie treningu GAN-ów jest trudnym zadaniem ze względu na niestabilność procesu uczenia oraz brak jednoznacznych metryk jakościowych. W przypadku tego projektu postanowiono wykorzystać dwie metryki: Mean Squared Error (MSE) oraz Peak Signal-to-Noise Ratio (PSNR), które są powszechnie stosowane do oceny jakości rekonstrukcji obrazów. Te metryki pozwalają na ilościowe porównanie generowanych obrazów z obrazami docelowymi, co ułatwia wybór najlepszego modelu.

6.5.1 MSE (Mean Squared Error)

MSE to jedna z najprostszych i najczęściej stosowanych metryk do oceny różnicy między dwoma obrazami. Oblicza się ją jako średnią kwadratów różnic między wartościami pikseli obrazu generowanego a obrazu docelowego. Formuła MSE jest następująca:

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

gdzie:

- y_i to wartość piksela w obrazie docelowym,
- \hat{y}_i to wartość piksela w obrazie generowanym,
- N to całkowita liczba pikseli w obrazie.

Zalety MSE:

- Jest prosta w obliczeniu i interpretacji.
- Wrażliwa na duże różnice między obrazami, co może pomóc w wykrywaniu poważnych błędów rekonstrukcji.

Wady MSE:

- Nie uwzględnia percepcji wzrokowej człowieka – nawet niewielkie różnice w tekście lub detalach mogą prowadzić do wysokich wartości MSE, mimo że obraz wygląda dobrze.

MSE jest szczególnie przydatna w początkowych etapach treningu, gdy model uczy się ogólnej struktury obrazu, ale może nie być wystarczająco precyzyjna do oceny jakości detali.

6.5.2 PSNR (Peak Signal-to-Noise Ratio)

PSNR to metryka oparta na MSE, która wyraża stosunek między maksymalną możliwą wartością sygnału (np. 255 dla obrazów 8-bitowych) a mocą szumu (błędu) reprezentowanego przez MSE. Im wyższa wartość PSNR, tym lepsza jakość rekonstrukcji obrazu. PSNR oblicza się za pomocą wzoru:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX^2}{MSE}\right)$$

gdzie:

- MAX to maksymalna możliwa wartość piksela (np. 255 dla obrazów 8-bitowych),
- MSE to wartość Mean Squared Error.

Zalety PSNR:

- Jest łatwa do obliczenia i interpretacji.

- Jest powszechnie stosowana w przetwarzaniu obrazów, co ułatwia porównywanie wyników z innymi parami.

Wady PSNR:

- Podobnie jak MSE, nie uwzględnia percepacji wzrokowej człowieka.
- Wysokie wartości PSNR nie zawsze oznaczają lepszą jakość wizualną, szczególnie w przypadku obrazów o złożonych teksturach lub detaliach.

PSNR jest szczególnie przydatna do porównywania modeli pod kątem ogólnej jakości rekonstrukcji, ale podobnie jak MSE, może nie być wystarczająco czuła na subtelne różnice w detalach.

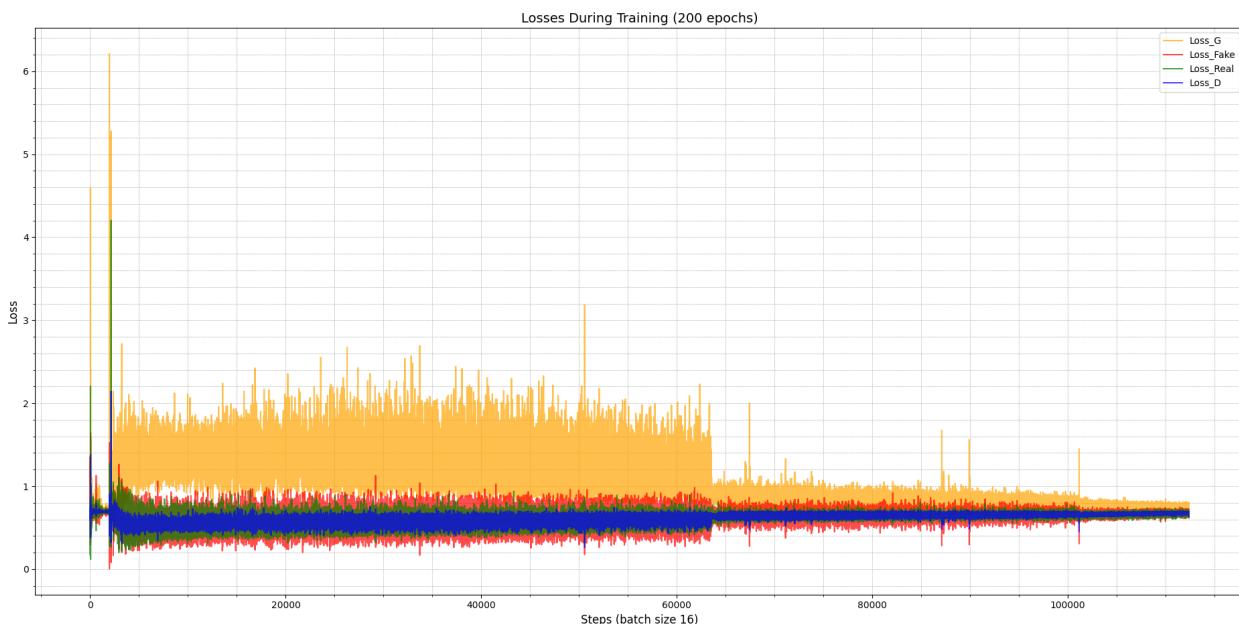
6.5.3 Praktyczne zastosowanie MSE i PSNR w projekcie

W trakcie treningu GAN-ów wartości MSE i PSNR zostały obliczone dla zbioru testowego dla każdej epoki modelu. Pozwoliło to na zobrazowanie postępów w jakości generowanych obrazów oraz na wybór najlepszej wersji modelu na podstawie obiektywnych kryteriów. Wyniki zostały zapisane do pliku CSV.

7 Ewaluacja najlepszego modelu

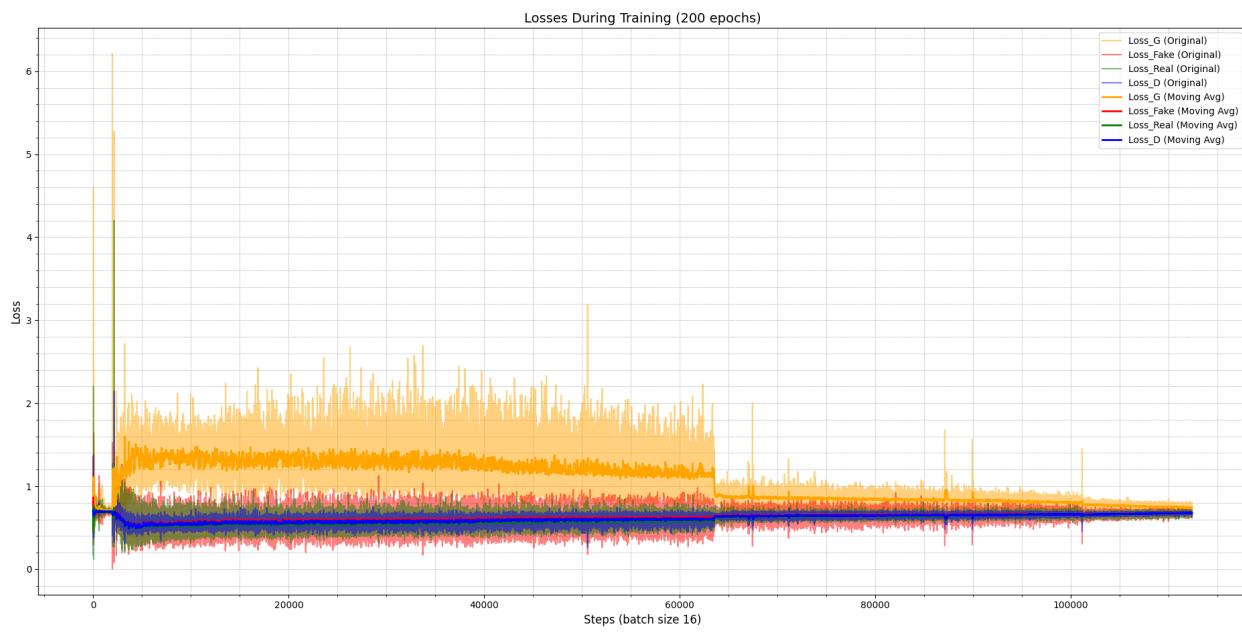
Spośród badanych architektur najlepsze rezultaty uzyskano dla architektury trzeciej. Model trenowano przez 200 epok, aż do naturalnego powrotu wartości funkcji straty dyskryminatora w okolice 0.69.

7.1 Przebiegi funkcji strat



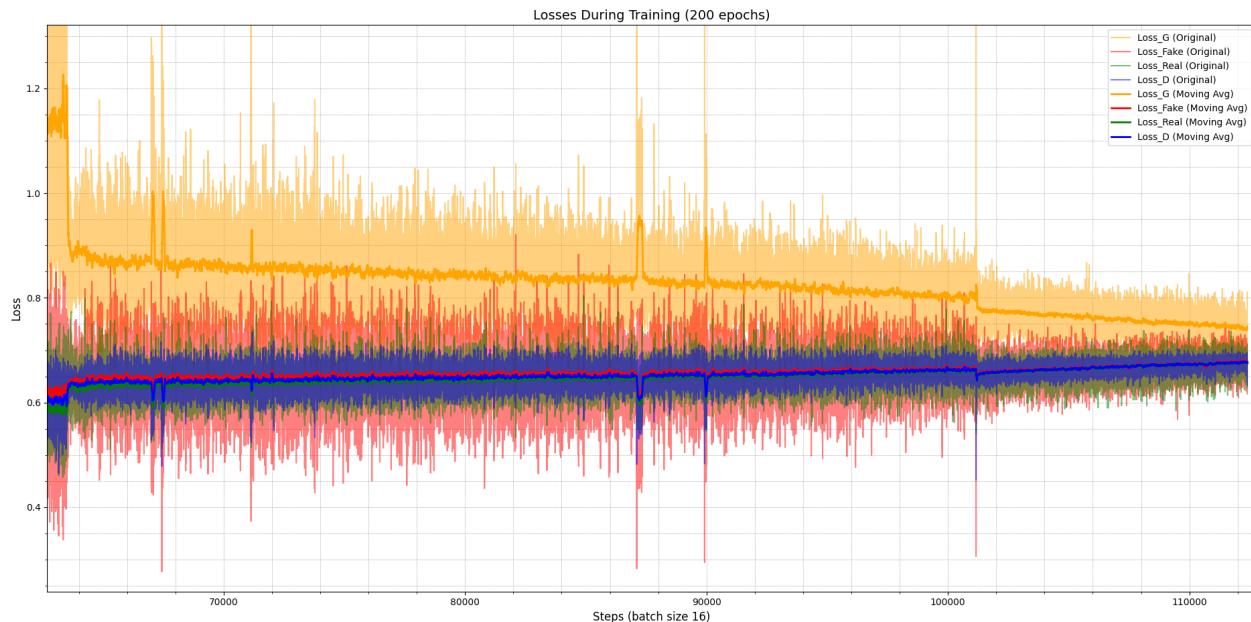
Rysunek 18: Przebieg funkcji strat dla architektury trzeciej w 200 epokach

Jak można zauważyć, dyskryminator stosunkowo szybko nauczył się odróżniać obrazy rzeczywiste od syntetycznych. Wraz z postępem treningu wariancja funkcji straty dla obrazów rzeczywistych maleje. Pojedyncze skoki funkcji straty generatora wskazują na momenty przerwania treningu. Aby zapewnić ciągłość procesu, należałoby dodatkowo zapisywać słownik dla scalera oraz obu optymalizatorów Adam. Widoczny jest stopniowy powrót wartości straty dyskryminatora (linia niebieska) w okolice 0.7.



Rysunek 19: Przebieg funkcji strat dla architektury trzeciej w 200 epokach ze średnią ruchomą (50 pomiarów)

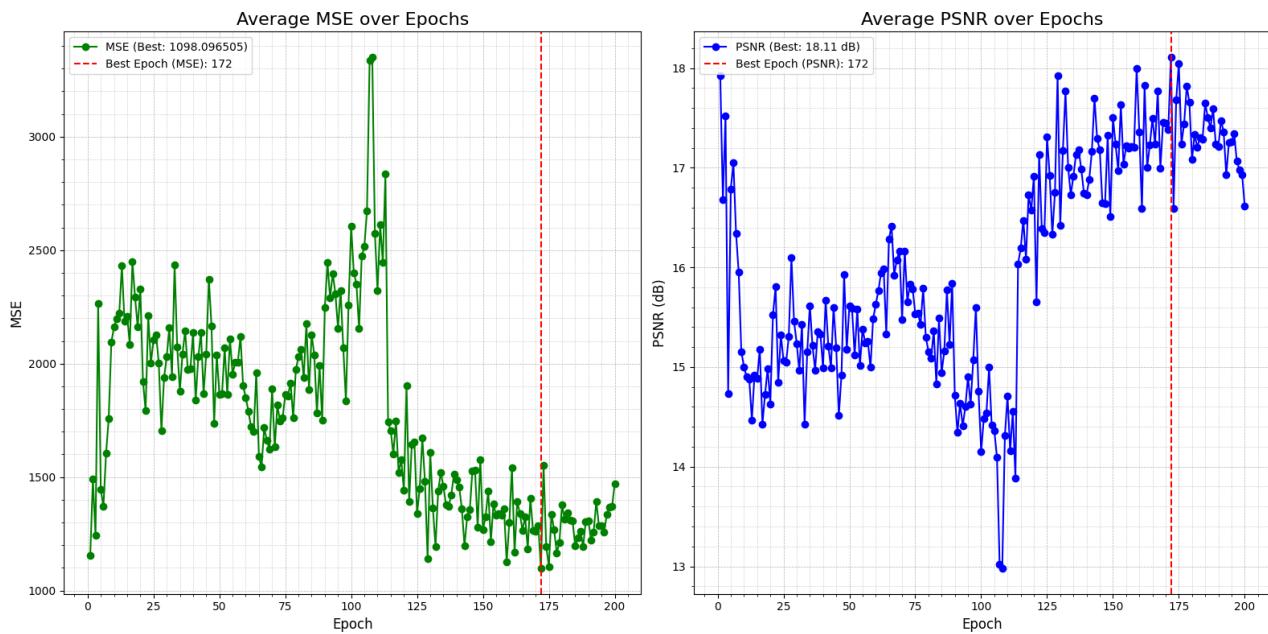
Średnia ruchoma pozwala lepiej zaobserwować trend zmian wartości funkcji strat. Widać, że dyskryminator gorzej klasyfikuje obrazy syntetyczne jako fałszywe, co jest oczekiwanym zachowaniem podczas treningu sieci GAN. Średnia ruchoma umożliwia również dokładniejszą obserwację spadku wartości funkcji straty generatora, nawet przy współczynniku uczenia wynoszącym 0.0002.



Rysunek 20: Fragment przebiegu funkcji strat z ruchomym oknem

Po 100 000. kroku widoczna jest zmiana wartości współczynnika uczenia dla dyskryminatora do 0.00005. Miało to na celu spowolnienie procesu uczenia dyskryminatora, aby zapobiec zbyt szybkiemu osiągnięciu 50% dokładności klasyfikacji.

7.2 Wykres średniej wartości MSE oraz PSNR



Rysunek 21: Wykres średniej wartości MSE oraz PSNR dla każdej epoki generatora na zbiorze testowym

Jak widać na wykresie, najlepszy model pochodzi z epoki 172, mimo że przewidywano, iż optymalne wyniki osiągnięte zostaną w okolicach 180. Dla współczynnika uczenia 0.0001 generator osiągał najlepsze rezultaty, jednak wynikalo to z wcześniejszego zastosowania wyższego współczynnika, który przyspieszył proces treningu. W okolicach 105. epoki zauważalny jest przełom, po którym wartości obu metryk zaczęły ponownie podążać za trendem liniowym wzrostu jakości. Po 180. epoce dalszy trening nie przyniósł znaczącej poprawy wyników.



Rysunek 22: Wygenerowana panorama po 172. epoce

Na wygenerowanym obrazie widać mniejszy kontrast oraz miejscowe artefakty, szczególnie w rogach i w miejscach łączenia obrazów. Charakterystyczny jest również czarny obszar na dolnym środku. Jest to efekt podziału obrazu na fragmenty z czarnym dopełnieniem na brzegach. Aby zredukować ten problem, można by zapisać fragmenty

obrazu w taki sposób, aby ich krawędzie były przesunięte w stronę środka.

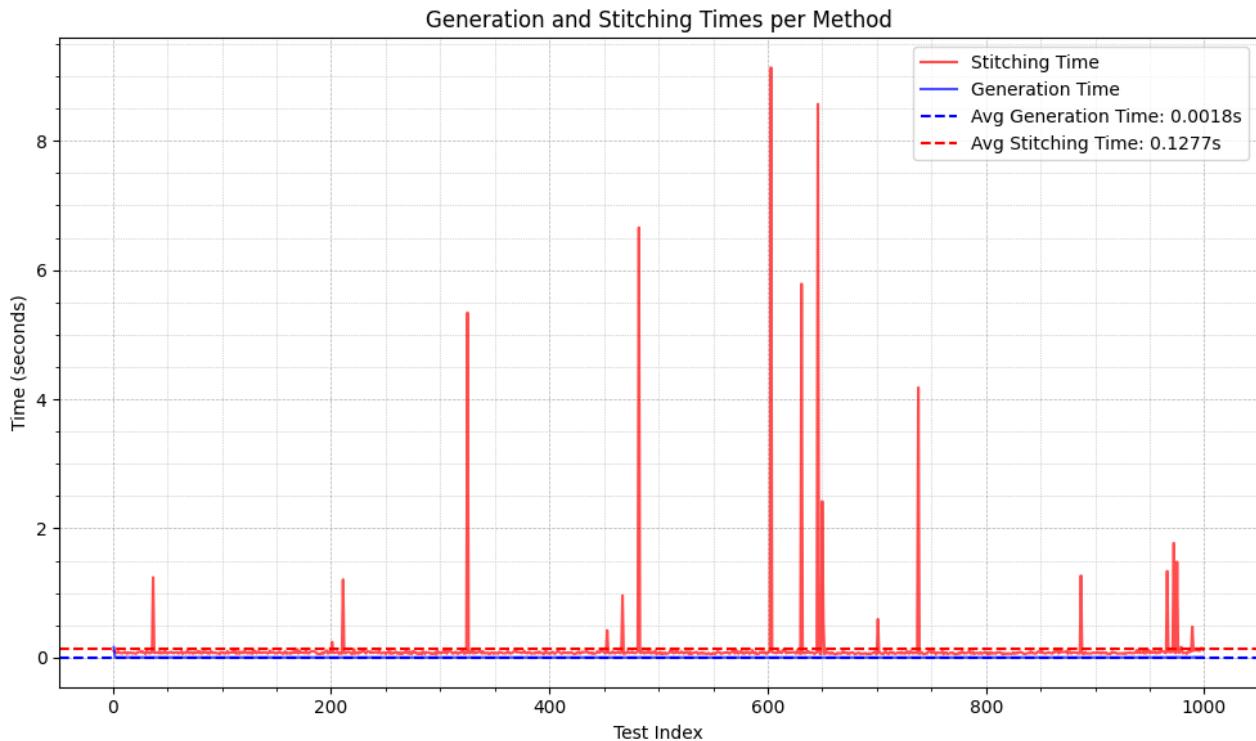
Dla porównania, poniżej przedstawiony jest obraz docelowy, stworzony za pomocą algorytmu SIFT:



Rysunek 23: Obraz docelowy stworzony na podstawie algorytmu SIFT

7.3 Porównanie czasowe obu metod

Jednym z celów na początku projektu było sprawdzenie, czy generator jest w stanie przyspieszyć proces analizy obrazów.

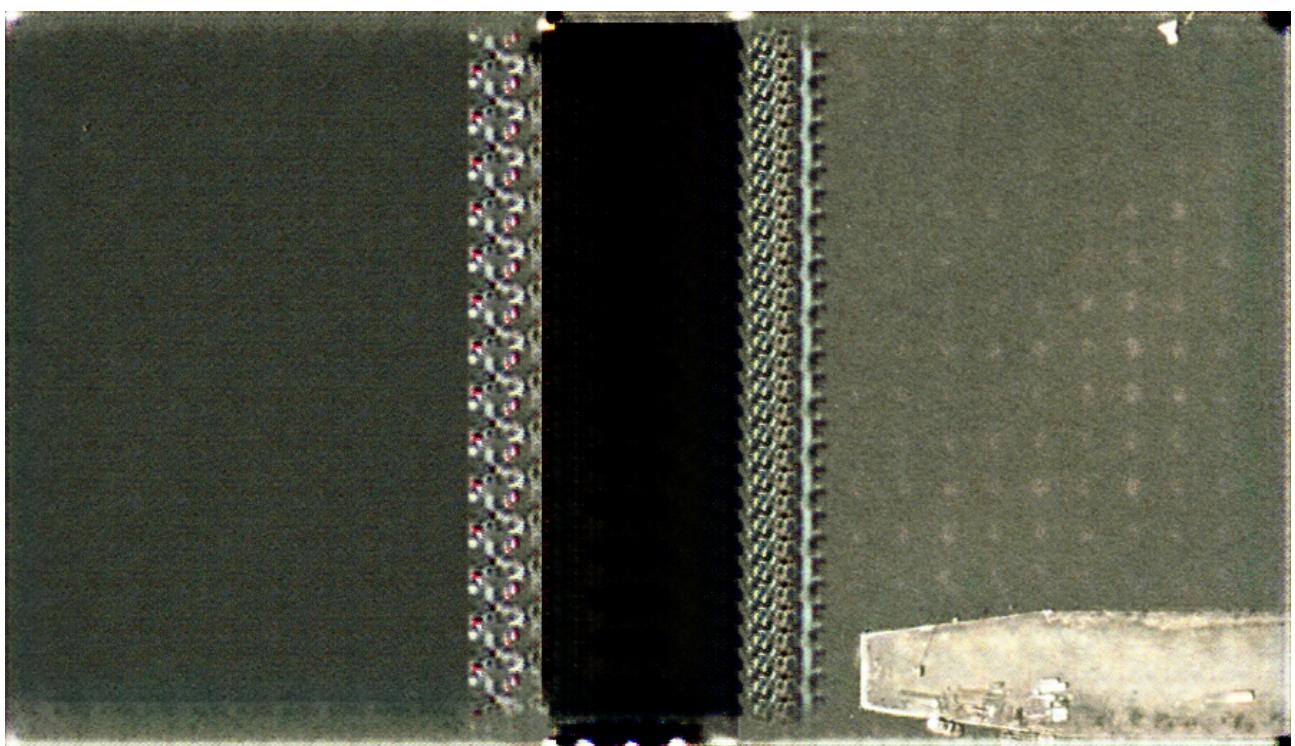


Rysunek 24: Porównanie czasowe obu metod na zbiorze testowym oraz średni czas działania

Jak można zauważyc, metoda oparta na generatorze działa w sposób niemal stały czasowo, podczas gdy metoda bazująca na algorytmie SIFT może nie znaleźć wystarczającej liczby wspólnych punktów charakterystycznych między obrazami wejściowymi, co zmusza ją do zastosowania bardziej kosztownych obliczeniowo operacji.



Rysunek 25: Obraz o najdłuższym czasie przetwarzania metodą bazującą na SIFT



Rysunek 26: Obraz wygenerowany przez model

Jak wspomniano wcześniej, największe trudności obie metody napotykają w przypadku obrazów pozbawionych wyraźnych punktów charakterystycznych, takich jak obszary wodne o jednolitej teksturze. Eksperyment potwierdza, że zastosowanie transformatora zamiast tradycyjnego algorytmu jest znacznie bardziej wydajne czasowo, jednak obecny model nie generuje obrazów o wysokiej jakości.

7.4 Wizualizacja przebiegu nauki dla wybranej architektury

Stworzono gif-a z postępów generatora przez epoki dla obrazu najczęściej wykorzystanego w sprawozdaniu.

8 Podsumowanie oraz wnioski

Podczas realizacji projektu udało się osiągnąć wiele istotnych rezultatów. Stworzony model sieci GAN wykazał potencjał w generowaniu panoram w kontrolowanych warunkach, choć wymaga dalszych usprawnień w zakresie jakości generowanych obrazów. Kluczową rolę w procesie łączenia obrazów odegrała maska binarna. Problem ograniczenia generowania obrazów o znanej proporcji wspólnych obszarów można rozwiązać poprzez wykorzystanie większej kanwy z czarnym wypełnieniem, z której za pomocą algorytmów przetwarzania cyfrowego można wyciąć największy prostokąt reprezentujący panoramę. To rozwiązanie wiąże się jednak z dłuższym i bardziej wymagającym procesem treningowym, na który zabrakłyby zasobów i czasu.

Dodatkowo, analiza porównawcza wykazała znaczną przewagę czasową metody opartej na GAN nad tradycyjnymi algorytmami, takimi jak SIFT.

W celu dalszego rozwoju warto byłoby zbadać architektury UNet z hierarchicznym doborem rozmiaru filtrów, zarówno z wykorzystaniem maski binarnej, jak i bez niej. Warto również przetestować dyskryminator z hierarchicznym doborem filtrów, co mogłoby wpłynąć na poprawę jakości generowanych obrazów.

Pomimo pewnych ograniczeń, projekt pozwolił na zdobycie cennej wiedzy dotyczącej trenowania i architektur sieci GAN oraz algorytmów scalania obrazów. Wyniki eksperymentów potwierdzają, że wykorzystanie transformatorów w procesie generowania panoram może być obiecującym kierunkiem badań.

8.1 Najważniejsze osiągnięcia

- Częściowy sukces w stworzeniu sieci GAN tworzącej panoramy w kontrolowanych warunkach
- Zdobycie obszernej wiedzy w zakresie treningu i architektur sieci GAN
- Poszerzenie wiedzy z algorytmów scalania obrazów
- Udowodnienie, że proces tworzenia panoram można znaczco przyspieszyć przez wykorzystanie sieci GAN

Literatura

- [1] Canwei Shen, Xiangyang Ji, and Changlong Miao. Real-time image stitching with convolutional neural networks. In *2019 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 192–197, 2019.