

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу**

До захисту допущено:
Завідувач кафедри
_____ Оксана ТИМОЩУК
«___» _____ 20__ р.

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системний аналіз і управління»
спеціальності 124 «Системний аналіз»
на тему: «Система створення художньої стилізації зображень на основі
генеративних змагальних мереж»

Виконав:

студент IV курсу, групи КА-71
Правосуд Руслан Олександрович

Керівник:

доцент, к.т.н., доцент кафедри ММСА
Дідковська М.В.

Консультант з економічного розділу:

доцент, к. е. н., доцент кафедри ТТРЕ
Рощина Н. В.

Консультант з нормконтролю:

доцент, к.т.н., доцент кафедри ММСА
Коваленко А. Є.

Рецензент:

доцент, к.т.н., доцент кафедри ПЗКС
Заболотня Т.М.

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.

Студент _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 124 «Системний аналіз»

Освітньо-професійна програма «Системний аналіз і управління»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Оксана ТИМОЩУК

« ____ » _____ 20__ р.

ЗАВДАННЯ

на дипломну роботу студенту

Правосуду Руслану Олександровичу

1. Тема роботи « Система створення художньої стилізації зображень на основі генеративних змагальних мереж», керівник роботи Дідковська М.В., доцент, к.т.н., затверджені наказом по університету від «26» травня 2021 р. № 1344-с

2. Термін подання студентом роботи 7 червня 2021р.

3. Вихідні дані до роботи: система створення художньої стилізації зображень на основі генеративних змагальних мереж

4. Зміст роботи: реалізація процесу навчання моделі; розробка системи медичного діагностування за вхідним зображенням, прогнозування наявності онкологічного захворювання крові.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
4	доцент, к. е. н. Рощина Н. В.		

7. Дата видачі завдання: 12.04.2021

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Затвердження теми ДР	12.04.2021-20.04.2021	виконано
2	Ознайомлення зі структурою БДР згідно з Положенням про державну атестацію студентів НТУУ «КПІ ім. І. Сікорського»	12.04.2020-27.04.2021	виконано
3	Ознайомлення з ДСТУ 3008-95 та стандарти ЄСПД	20.04.2021-27.04.2021	виконано
4	Проведення дослідження за темою БДР під керівництвом керівника	20.04.2021-04.05.2021	виконано
5	Завершення роботи над першим варіантом частини БДР	05.05.2021-12.05.2021	виконано
6	Проведення роботи над експериментальною частиною БДР	05.05.2021-16.05.2021	виконано
7	Проведення роботи над програмним продуктом	16.05.2021-25.05.2021	виконано
8	Оформлення БДР та аналіз отриманих результатів	16.05.2021-25.05.2021	виконано

Студент

Руслан Правосуд

Керівник

Марина Дідковська

РЕФЕРАТ

Дипломна робота: 104 с., 29 рис., 2 додатки, 11 джерел.

ПЕРЕНЕСЕННЯ СТИЛЮ, ГЕНЕРАТИВНІ ЗМАГАЛЬНІ НЕЙРОННІ МЕРЕЖІ.

Об'єкт дослідження: система перенесення стилю, що дозволяє переносити загальний стиль деякого набору зображень до окремого зображення, без втрати семантичного наповнення.

Предмет дослідження: методи перенесення стилю, що базуються на генеративних змагальних нейронних мережах

Мета дослідження: створити інструмент контрольованого перенесення стилю, що дозволяє дослідницькі експерименти та інші застосування

Постановка задачі: розробити систему перенесення стилю на базі математичного апарату генеративних змагальних нейронних мереж

Програмний продукт було розроблено з використанням мови програмування Python, на основі Jupyter Notebook.

ABSTRACT

Thesis: 104 p., 29 fig., 2 appendices, 11 sources.

STYLE TRANSFER, GENERATIVE ADVERSARIAL NEURAL NETWORKS.

The Object of research: a style transfer system that allows transferring the overall style of a set of images to a single image, without losing semantic content.

The Subject of research: methods of style translation based on general adversarial neural networks

The purpose of the study: to create a controlled style transfer tool that allows research experiments and other applications

Problem statement: to develop a system of style transfer on the basis of the mathematical apparatus of generative adversarial neural networks

The programmable product was developed using Python language programming based on Jupyter Notebook.

ЗМІСТ

ВСТУП.....	8
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ МЕТОДІВ ПЕРЕНЕСЕННЯ СТИЛЮ МІЖ ДВОМА НАБОРАМИ ЗОБРАЖЕНЬ	10
1.1 Задача перенесення стилю між наборами зображень	10
1.2 Огляд методів стилізації зображення з використанням нейронних мереж та деякі приклади застосування	13
1.3 Застосування нейронних мереж	17
1.4 Застосування генеративних змагальних мереж для вирішення задачі перенесення стилю	18
Висновки до розділу 1.....	19
РОЗДІЛ 2 ТЕОРЕТИЧНІ ОСНОВИ ТА РОЗВИТОК ЗГОРТКОВИХ ТА ГЕНЕРАТИВНИХ ЗМАГАЛЬНИХ НЕЙРОННИХ МЕРЕЖ.....	20
2.1 Нейронні мережі.....	20
2.2. Згорткові нейронні мережі	24
2.3 Генеративні змагальні мережі	28
2.4 Циклічні генеративні змагальні мережі	30
2.5 Огляд архітектури мереж генеративної та дискримінаційної частин циклічних генеративних змагальних мереж	35
2.5.1 U-Net.....	35
2.5.2 PatchGAN	37
Висновки до розділу 2.....	39
РОЗДІЛ 3 АРХІТЕКТУРА ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ ПОБУДОВАНОГО ПРОГРАМНОГО ПРОДУКТУ	40
3.1 Вибір мови програмування.....	40
3.2 Використання бібліотек та модулів	42
3.2.1 Вбудовані модулі.....	42
3.2.2 PIL	42
3.2.3 Numpy	44
3.2.4 TensorFlow	46
3.3 Архітектура програми	48
3.3.1 Навчання моделі.....	48
3.3.2 Використання моделі.....	50
3.4 Результати	51
Висновки до розділу 3.....	53
РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ	54
4.1 Постановка задачі проектування	54

4.2 Обґрунтування функцій програмного продукту.....	55
4.3 Обґрунтування системи параметрів ПП.....	58
4.4 Аналіз експертного оцінювання параметрів.....	60
4.5 Аналіз рівня якості варіантів реалізації функцій.....	65
4.6 Економічний аналіз варіантів розробки ПП.....	66
4.7 Вибір кращого варіанту ПП техніко-економічного рівня.....	72
Висновки до розділу 4.....	73
ВИСНОВКИ	74
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	75
ДОДАТОК А КОД ПРОГРАМНОГО ПРОДУКТУ	77
ДОДАТОК Б ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	99

ВСТУП

Будь-яке зображення можна розділити на дві складові: його зміст та стиль. При цьому, для деяких наборів зображень можна виявити схожий стиль його складових, при абсолютно довільному семантичному наповненні. Одним із таких прикладів є роботи художників.

Людський зір, як система розпізнавання зображень, може досить точно класифікувати зображення за стилістичним наповненням. У той же час, дана задача досить складно формалізується у математичних термінах. Тим не менш, за останні декілька десятиліть спостерігається стрімкий розвиток технологічних засобів розв'язання задач комп'ютерного бачення. За допомогою, зокрема, згорткових та генеративних змагальних нейронних мереж.

Однією з таких задач є задача стилістичного відображення між двома наборами зображень, кожному з яких притаманний власний, спільний для всіх його складових стиль. При цьому, вимагається збереження семантичного наповнення вихідного зображення, при зміні його стилістичного наповнення.

Об'єктом даної роботи є дослідження та побудова системи стилістичного наповнення зображення стилем заданого набору зображень, що мають спільний стиль, у даному випадку - картин художника Клод Моне.

Дипломна робота складається з чотирьох розділів.

У першому розділі проводиться аналіз актуальності проблеми, наводяться приклади використання систем створення художньої стилізації зображень, проводиться огляд класичних інструментів розв'язання задачі перенесення стилю.

Другий розділ присвячено огляду теоретичних основ та архітектури генеративних змагальних та згорткових мереж.

У третьому розділі наводиться опис розробленого програмного продукту, обраного алгоритму вирішення задачі та результатів роботи програми.

Четвертий розділ виконує функціонально-вартісний аналізу отриманого програмного продукту.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ МЕТОДІВ ПЕРЕНЕСЕННЯ СТИЛЮ МІЖ ДВОМА НАБОРАМИ ЗОБРАЖЕНЬ

1.1 Задача перенесення стилю між наборами зображень

Людському зір здатний досить точно розрізняти як смислове так і стилістичне наповнення зображення. Зокрема, для людських очей досить простою є задача розрізнення фотознімків та написаних художниками картин.

У основі мистецтва полягає принцип перенесення власних вражень у створювану роботу. При цьому вони можуть мати деякий контраст із її семантичним наповненням.

У той же час роботи фіксованого художника будуть мати певну стилістичну закономірність, яку фахівці здатні легко виявити серед безлічі інших робіт.

Що побачив Клод Моне, ставлячи свій мольберт біля берегу Сієни поблизу Аргентейла в чудовий весняний день в 1873 (Рисунок 1.1, вгорі)? Кольорова фотографія, якби вона була винайдена на той момент, можливо, задокументувала б чітке блакитне небо та річку, що його віддзеркалює.

Моне передав свої враження від цього пейзажу за допомогою м'яких мазків пензля та яскравої палітри. Що якби Моне трапилося опинитись в маленькій гавані в Кассі у прохолодний літній вечір (Рисунок 1.1, знизу)? Коротка прогулянка через галерею картин Моне дозволяє легко уявити як він намалював би пейзаж: можливо, в пастельних тонах відтінки, з різкими нальотами фарби і дещо стиснений кольоровий діапазон відтворили б побачений ним пейзаж.

Ми можемо уявити все це, незважаючи на те, що ніколи не бачили приклад картини Моне поряд з фотографією місця, що він малював.

Натомість ми маємо знання про набір картин Моне та набір пейзажних фотографій. Ми можемо зрозуміти стилістичні відмінності між цими двома наборами, і тим самим уявити, як би могло виглядати зображення, якби ми могли «перекласти» його з одного набору в інший.

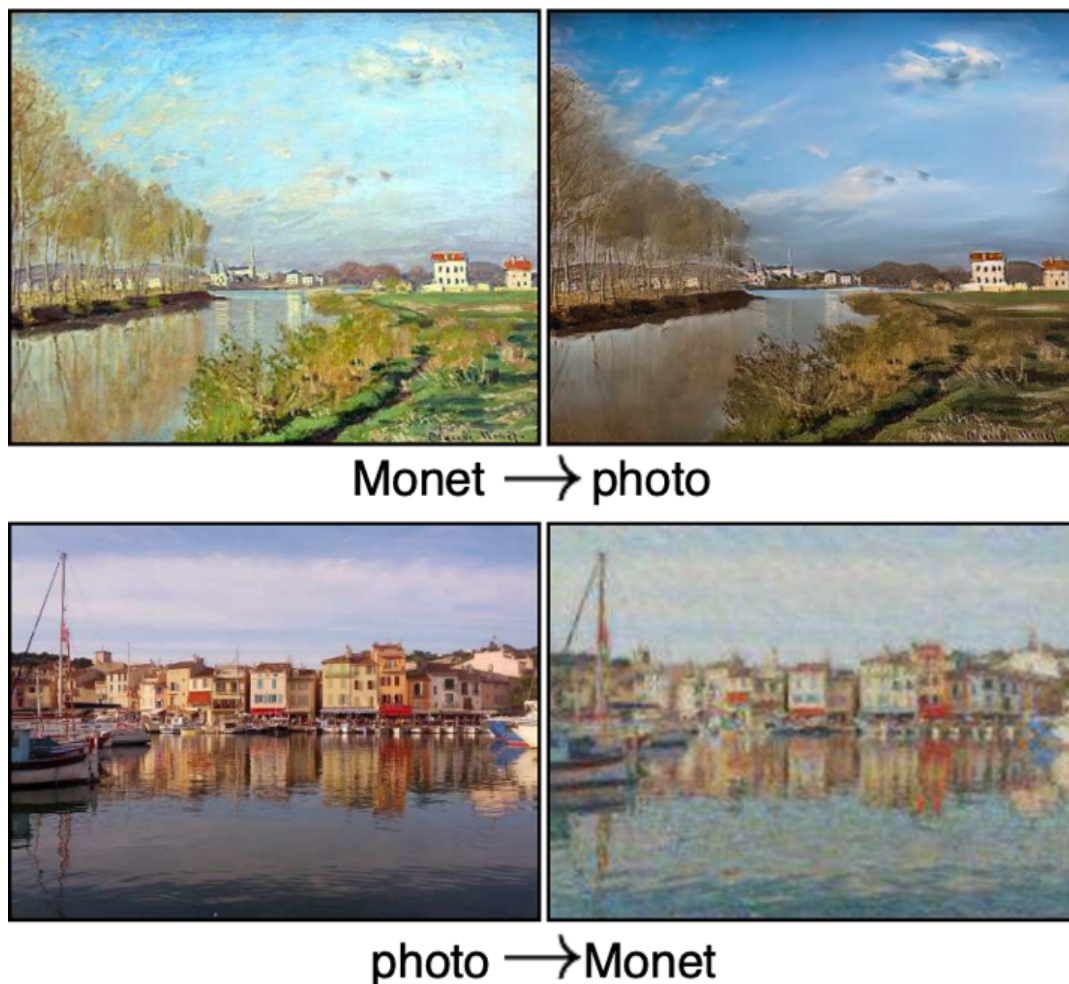


Рисунок 1.1 Стилiстичне вiдображення мiж двома наборами зображень

За остання десятиліття технології комп'ютерного бачення значно просунулися у своєму розвитку. Зокрема, значну частину досліджень склали задачі генерування нових зображень та задачі перенесення стилю з використанням генеративних змагальних мереж.

Серед задач перенесення стилю можна виокремити задачу перенесення стилю між двома наборами зображень. У кожному з яких існує загальний стиль характерний для всіх зображень набору.

У даній роботі вирішується задача перенесення стилю між набором звичайних довільних фотографій та набором картин Клода Моне.

1.2 Огляд методів стилізації зображення з використанням нейронних мереж та деякі приклади застосування

Класична задача перенесення стилю вирішується для двох конкретних зображень на основі нейронної мережі VGG-Network. При цьому у одного з зображень виокремлюється семантичний зміст, у іншого стильова репрезентація. Після цього ці дві складові поєднуються у єдине зображення.

Відкриття того, що семантика та стиль зображення є роздільними величинами було описано у статті «A Neural Algorithm of Artistic Style», написаній Leon A. Gatys, Alexander S. Ecker та Matthias Bethge у 2015 році.

Зокрема, у цій же статті, були наведені приклади роботи подібної системи (Рисунок 1.2).



Рисунок 1.2 Приклад поєднання семантичної та стильової складової зображень. а) оригінал, б)-д) результати стилізації оригіналу

Ще одним з підходів до розв'язання даної задачі є використання умовних генеративних змагальних мереж. При цьому для навчання мережі необхідна наявність пар взаємно відповідних зображень з двох наборів даних, що не підходить для розв'язання поставленої у даній роботі задачі.

Задача стилістичного відображення між двома наборами зображень, без семантичного зв'язку між довільними двома зображення з різних наборів даних, зветься задачею непарного відображення зображень (Рисунок 1.3).

Створення саме такої системи є метою даної роботи. Методом її розв'язання є використання циклічних генеративних змагальних мереж (CycleGAN), які описані її авторами (Jun-Yan Zhu, Taesung Park, Phillip Isola, Alexei A. Efros) у статті «Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks» 2017 року.

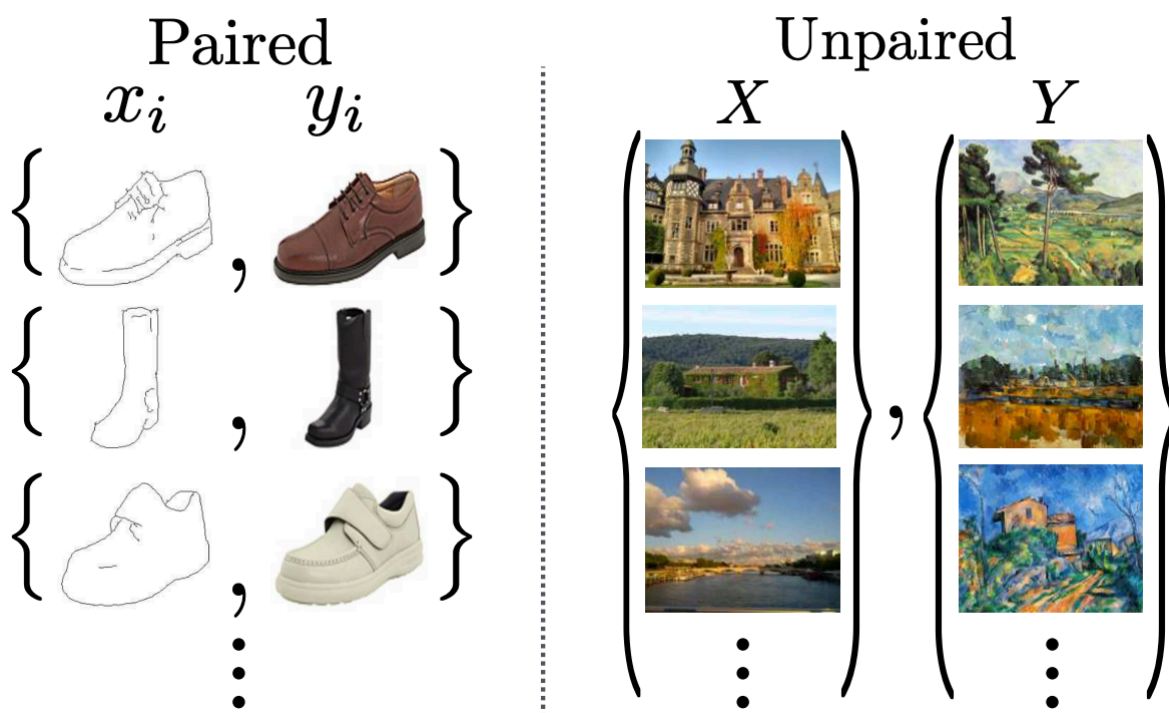


Рисунок 1.3 Зліва – парний набір зображень. Справа – непарний набір зображень

Серед прикладів застосування стилізації непарних наборів зображень слід зазначити вирішення задачі генерування зображень іншої пори року, що може бути використано для розширення наборів вхідних даних у інших задачах комп'ютерного бачення (Рисунок 1.4).

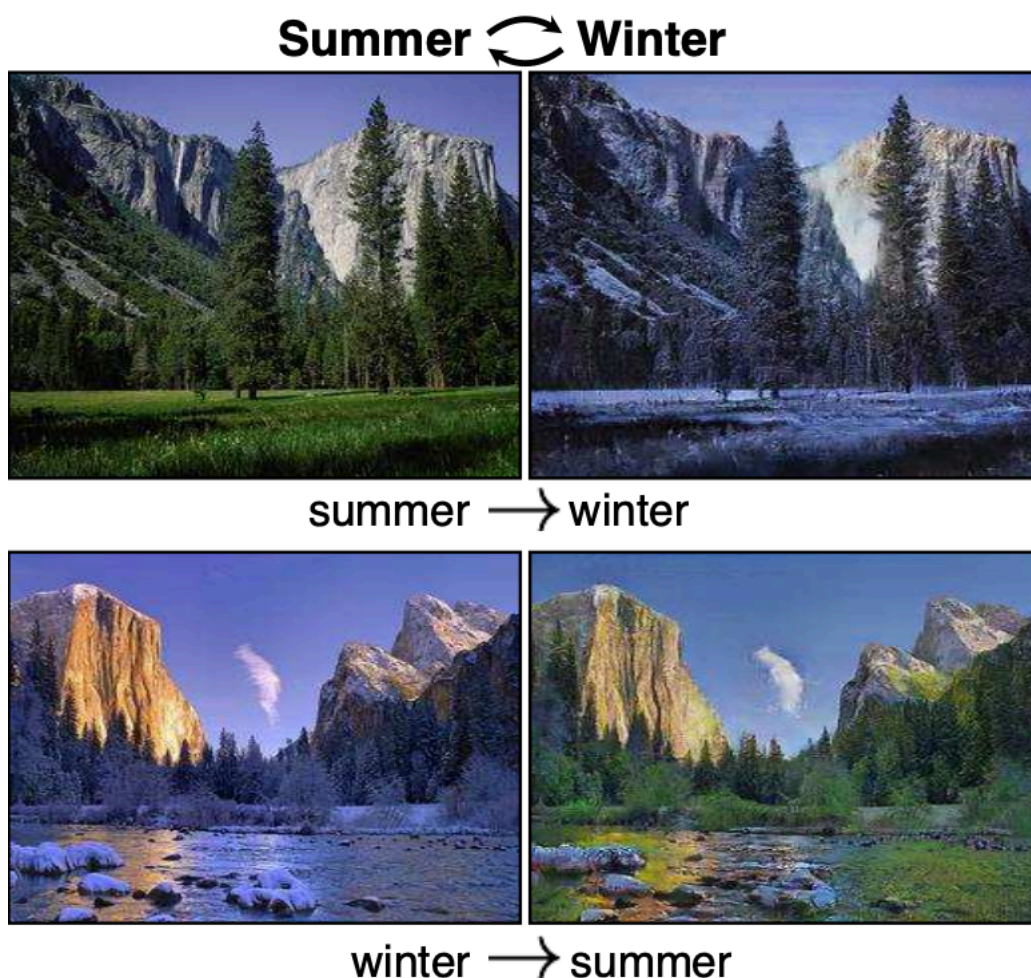


Рисунок 1.4 Стильове оформлення зображення відповідно до певної пори року

Ще однією цікавою задачею, що вирішується зазначеним методом є перехід від супутникового зображення ландшафту до схематичного і навпаки (Рисунок 1.5).



Рисунок 1.5 Відображення супутникового зображення у схематичне

1.3 Застосування нейронних мереж

Основним засобом розв'язання складних задач комп'ютерного зору є використання нейронних мереж.

В силу особливостей архітектури, згорткові нейронні мережі здатні знаходити в зображеннях складні характеристики та використовувати їх для вирішення поставленої в конкретному випадку задачі.

Особливе значення мають генеративні змагальні мережі, які, на відміну від інших, не вирішують задачу розпізнавання, а виконують генерування зображень певного, в залежності від власної архітектури та поставленої задачі, виду.

Наразі, багато досліджень у сфері комп'ютерного зору проводиться з метою знаходження способів розширення класу задач, що вирішуються нейронними мережами, та збільшення рівня якості вирішення уже відомих задач.

Зокрема, стрімкий прогрес спостерігається у якості роботи генеративних змагальних мереж (Рисунок 1.6).



Рисунок 1.6 Розвиток якості генерування людських обличь генеративними змагальними мережами 2014-2017 роки

1.4 Застосування генеративних змагальних мереж для вирішення задачі перенесення стилю

Як було зазначено у пункті 1.2, існує декілька типів задач перенесення стилю від одного зображення до іншого.

При цьому для різних задач існують різні способи їх вирішення, різні архітектури відповідних нейронних мереж. Для всіх них, оскільки вирішується задача обробки цифрових зображень, характерна наявність згорткових шарів в архітектурі.

У нашій задачі, найбільш доцільно використовувати циклічну генеративну змагальну мережу. Оскільки, вона найкраще підходить для вирішення задачі перенесення стилю між двома наборами зображень, що не мають впорядкованої семантичної відповідності між довільними двома представниками з різних наборів.

Крім того, в силу симетричності вирішуваної задачі (з точки зору циклічної генеративної змагальної мережі), встановлюється обернена відповідність між картинами Клода Моне та звичайними фотографіями, що є досить цікавою особливістю, хоча і виходить за рамки поставленої задачі.

Висновки до розділу 1

Задача перенесення стилю була досліджена і доведена до сучасного стану розуміння менш, ніж за останнє десятиліття і є досить молодого задачею, вирішеною у сфері комп'ютерного зору.

В залежності від особливостей поставленої задачі існують різні підходи для її вирішення. Велику у роль у вирішенні подібних задач відіграють згорткові та генеративні змагальні мережі.

Результати їх роботи знаходять застосування у широкому колі інших задач пов'язаних з комп'ютерним зором.

РОЗДІЛ 2 ТЕОРЕТИЧНІ ОСНОВИ ТА РОЗВИТОК ЗГОРТКОВИХ ТА ГЕНЕРАТИВНИХ ЗМАГАЛЬНИХ НЕЙРОННИХ МЕРЕЖ

2.1 Нейронні мережі

Нейронні мережі - це набір алгоритмів, змодельованих на зразок людського мозку, призначених для розпізнавання певних закономірностей. Вони інтерпретують отримані дані через своєрідне машинне сприйняття, класифікуючи, ставлячи у відповідність деякі числові дані або кластеризуючи сирі вхідні дані.

Закономірності, які вони розпізнають, є числовими, та зберігаються у векторах, в які повинні бути перекладені всі дані з реального світу, наприклад, зображення, звук, текст чи часові ряди.

Шари мережі складаються зроблені з вузлів. Вузол - це елемент, де відбуваються обчислення, що моделює нейрон людського мозку, який спрацьовує, коли зустрічає достатню кількість подразників. Вузол поєднує вхідні дані з набором коефіцієнтів, або вагів, які або підсилюють, або послаблюють цей вхідний сигнал, тим самим надаючи значення вхідним даним щодо завдання, якому намагається навчити алгоритм.

Ці добутки вхідних даних та вагів підсумовуються, а потім сума передається через так звану функцію активації вузла, щоб визначити, чи повинен і в якій мірі цей сигнал прогресувати далі через мережу, щоб вплинути на кінцевий результат, скажімо, на процес класифікації. Якщо сигнали проходять, нейрон «активується» (Рисунок 2.1).

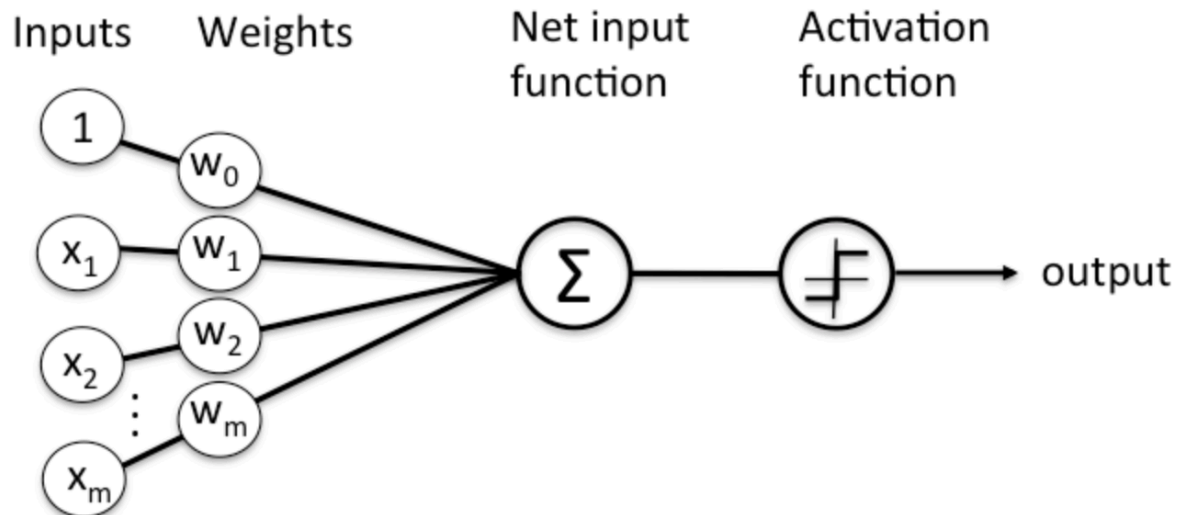


Рисунок 2.1 Модель вузла (нейрона) нейронної мережі

Шар вузлів - це ряд нейроноподібних перемикачів, які вмикаються або вимикаються, коли вхід подається через мережу.

Вихідні дані кожного шару є одночасно вхідними даними наступного шару, починаючи з початкового вхідного шару, що приймає вхідні дані відповідної задачі.

Окремо можна виділити вхідний шар, що приймає початкові дані та вихідний шар, що видає результат роботи алгоритму. Усі інші шари нейронної мережі зветься внутрішніми (Рисунок 2.2).

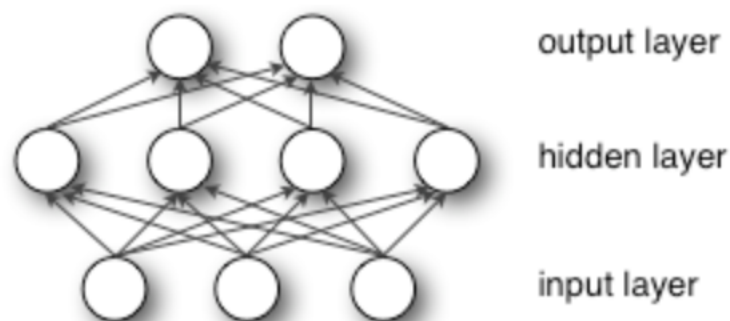
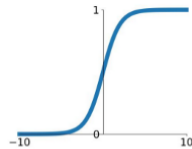


Рисунок 2.2 Схематичне зображення архітектури класичної нейронної мережі

В залежності від поставленої задачі та типу шару нейронної мережі обираються відповідні функції активації (Рисунок 2.3). Зокрема, стандартною функцією активації для згорткових мереж є ReLu або Leaky ReLu.

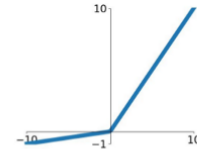
Sigmoid

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



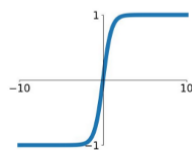
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

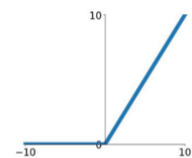


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

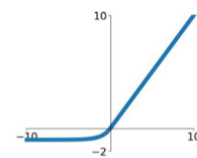


Рисунок 2.3 Приклади функцій активації

Для навчання нейронної мережі, вхідні дані проходять по нейронній мережі від початкового шару до вихідного і порівнюються з очікуваними даними, шляхом складання деякої функції помилок, яку нейронна мережа намагається мінімізувати.

Для цього обчислюються градієнт функції помилок по функціям активації та вагам. Значення ваг коригується, використовуючи значення координат градієнту. Такий процес навчання продовжується доки значення функції помилок не стане достатньо задовільним (Рисунок 2.4).

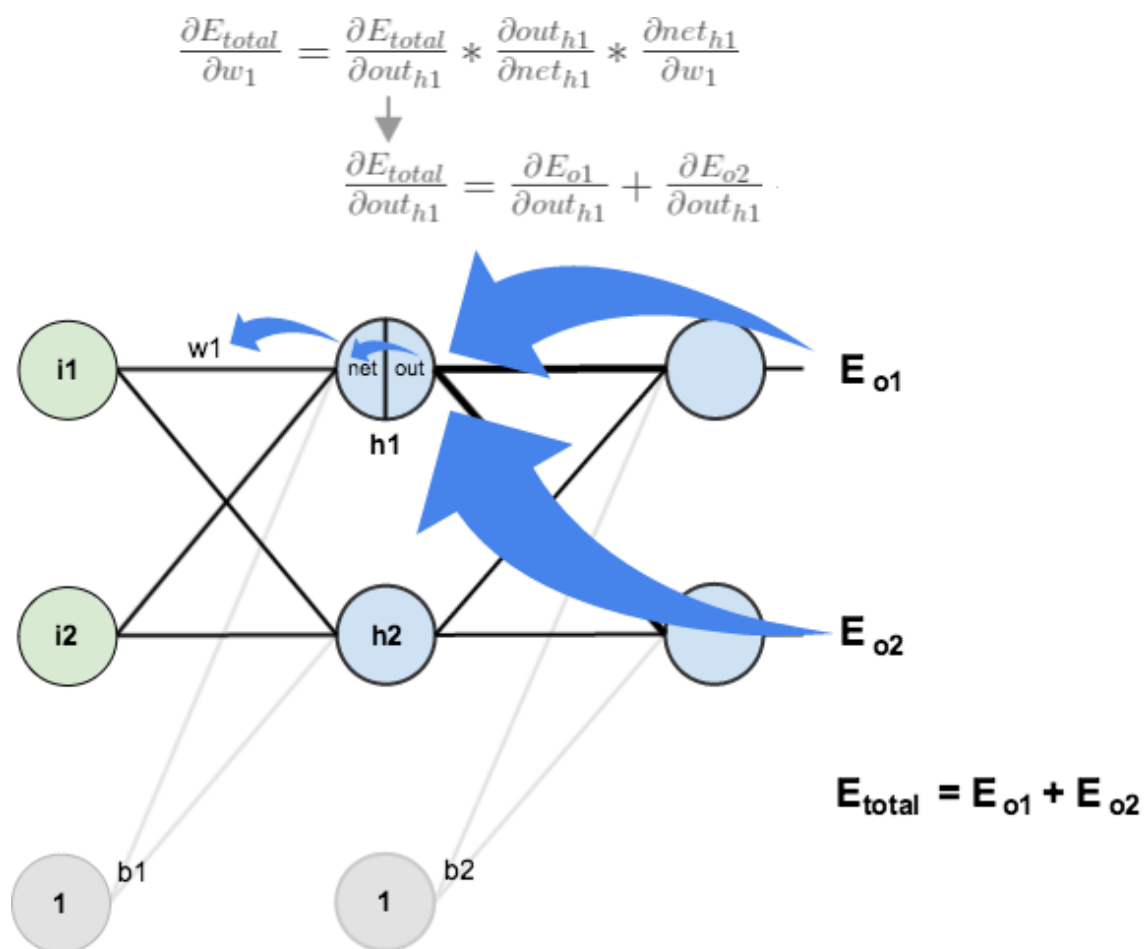


Рисунок 2.4 Приклад поширення помилки – оновлення значень ваг нейронної мережі

2.2. Згорткові нейронні мережі

Згорткові нейронні мережі спеціалізуються на роботі із зображеннями. Їх ефективність у даній сфері досягається за рахунок наявності в архітектурі, так званих, згорткових шарів, що виконують фільтрацію та виявлення певних характеристик у зображеннях (Рисунок 2.5).

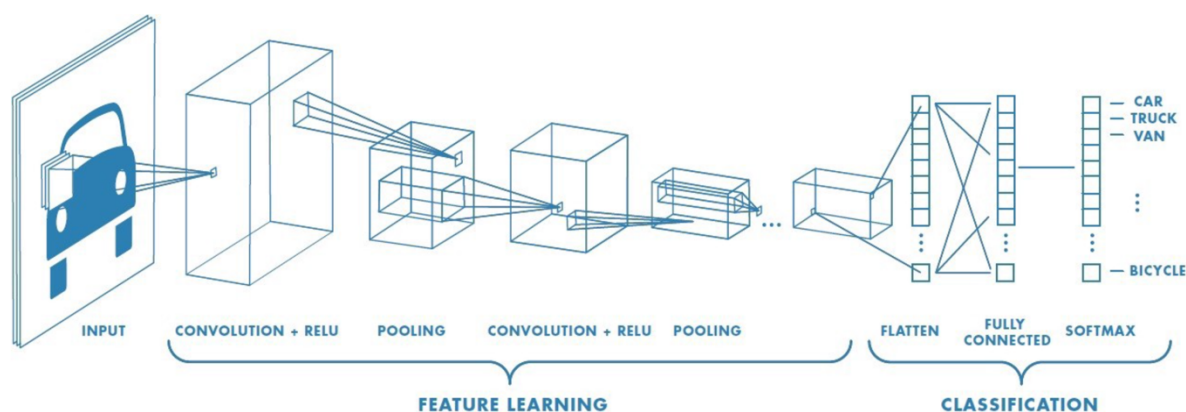


Рисунок 2.5 Приклад архітектури згорткової нейронної мережі

Фільтри влаштовані таким чином, що деякі регіони, представленого у цифровому виді зображення, множаться на певні ваги та підсумовуються, передаючи нову інформацію на наступний шар нейронної мережі. Ця операція зветься згорткою.

Кольорове зображення є тривимірним об'єктом, що має висоту, ширину та 3 кольорових канали (Рисунок 2.6).

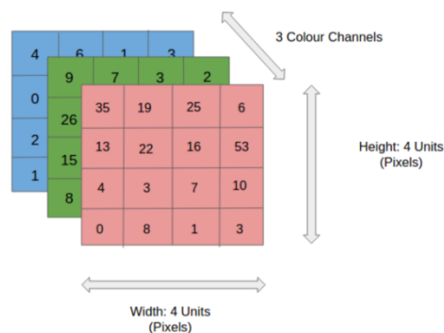


Рисунок 2.6 Складові цифрового зображення

Згортка відбувається шляхом руху фільтра уздовж зображення, та відображення виконаної операції на нове зображення (Рисунок 2.7).

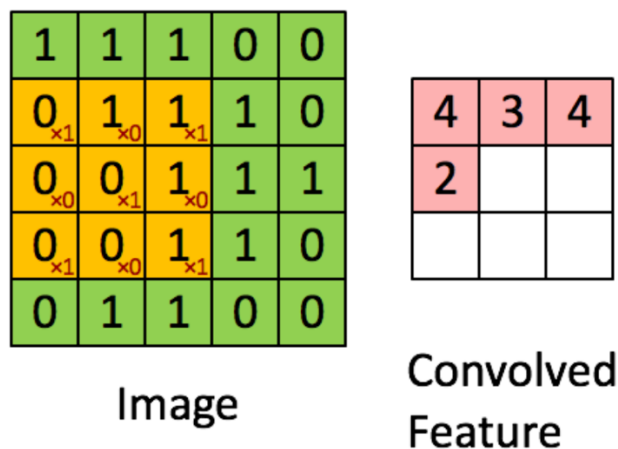


Рисунок 2.7 Приклад роботи згортки

У випадку кольорового зображення, подібний рух фільтра уздовж зображення відбувається аналогічно (Рисунок 2.8).

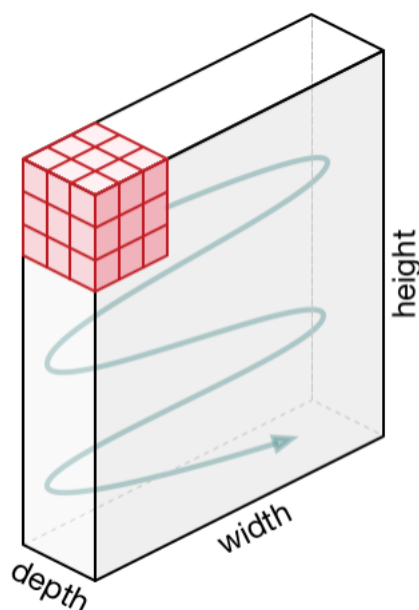


Рисунок 2.8 Рух фільтра у випадку кольорового зображення.

Результатом застосування згортки до кольорового зображення є матриця чисел (Рисунок 2.9).

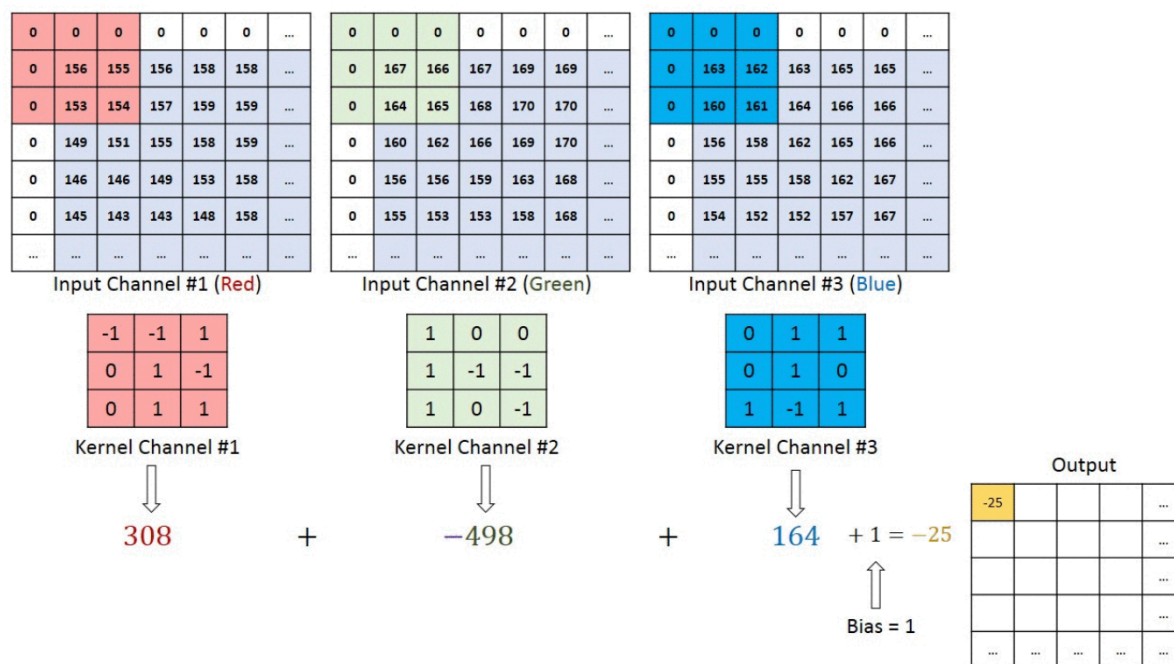


Рисунок 2.9 Застосування згортки до кольорового зображення

Після виконання згортки до отриманих даних застосовується функція активації (зазвичай ReLU або Leaky ReLU).

Іноді застосовується згортка, що збільшує розмір поточного зображення за рахунок додання «рамки» з нейтральних елементів та подальшого застосування фільтру.

Крім того у архітектурі згорткової мережі можуть бути наявні, так звані, шари пулінгу, які схожі за принципом роботи на згортку. Їх задача полягає в зменшенні розміру зображення, що призводить до зменшення часу на обчислення та вибір найбільш значущих характеристик зображення. На відміну від згортки, у даному випадку обирається найбільше або середнє значення у регіоні, що потрапляє в фільтр.

По всіх інших параметрах процес навчання згорткової нейронної мережі нічим не відрізняється від процесу навчання у звичайному випадку. При цьому, слід зазначити, що у процесі навчання змінюється і значення вагів фільтрів, що дозволяє мережі адаптувати процес відбору

характеристик у регіонах зображень відповідно до потреб поставленої задачі.

2.3 Генеративні змагальні мережі

У системі генеративних змагальних мереж генеративна модель протиставляється супернику: дискримінаційній моделі, яка навчається визначати, чи належать подані їй дані розподілу згенерованого генеративною моделлю чи реальному розподілу даних.

Генеративну модель можна вважати аналогічною групі фальшивомонетників, які намагаються виготовити фальшиву валюту та використовувати її без виявлення, тоді як дискримінаційна модель є аналог поліції, що намагається виявити фальшиву валюту. Змагання в цій грі підштовхує обидві команди вдосконалювати свої методи, доки підробки не стане неможливо відрізнити від оригіналу.

Отже, загальний принцип роботи генеративної змагальної мережі полягає в тому, що є дві мережі, одна з яких генерує зображення певного виду, намагаючись зробити їх схожими на зображення з множини деякого відомого набору зображень. При цьому, інша нейронна мережа намагається відрізнити згенеровані зображення від оригінальних (Рисунок 2.10). Навчання дискримінаційної частини, принципово нічим не відрізняється від навчання бінарного класифікатора. У той же час, генеруюча мережа отримує зворотній зв'язок від дискримінаційної та реагує на них відповідним чином, змінюючи свої ваги.

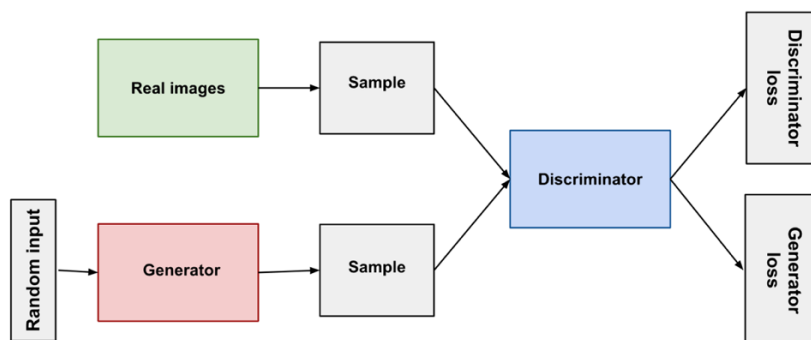


Рисунок 2.10 Схематичне зображення генеративної змагальної нейронної мережі

Задачу генеративної змагальної мережі можна формалізувати наступною формулою:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))] \quad (2.3.1)$$

де G та D – диференційовані функції

G – відображає вхідні дані у бажаний набір даних

$D(\mathbf{x})$ – функція, що визначає імовірність, що $\mathbf{x}$ належить реальному набору даних, а не згенерована генеративною частиною моделі

$p_{\text{data}}(\mathbf{x})$ – набір реальних даних, які навчається генерувати генеративна модель

$p_z(\mathbf{z})$ – дані, що подаються на вхід генеративній моделі

Для навчання дискримінаційної та генеративної частини мережі, беруться відповідні градієнти цільової функції та з їх використанням оновлюються значення відповідних ваг.

Як правило, процес навчання складається з декількох ітерацій. При цьому, кожную ітерацію можна розділити на дві частини: перша – оновлення ваг дискримінаційної моделі (Рисунок 2.11), друга – генеративної.

При цьому оновлення генеративної моделі відбувається з залученням дискримінаційної. Оскільки для осмисленого навчання генеративна модель має знати як на її поточний стан реагує дискримінаційна модель.

На Рисунку 2.11 та Рисунку 2.12 зображено які частини генеративної змагальної мережі залучено під час відповідних фаз ітерації роботи нейронної мережі.

Оскільки у нашій задачі робота відбувається із зображеннями, генеративна та дискримінаційна частина нейронної мережі є згортковими нейронними мережами.

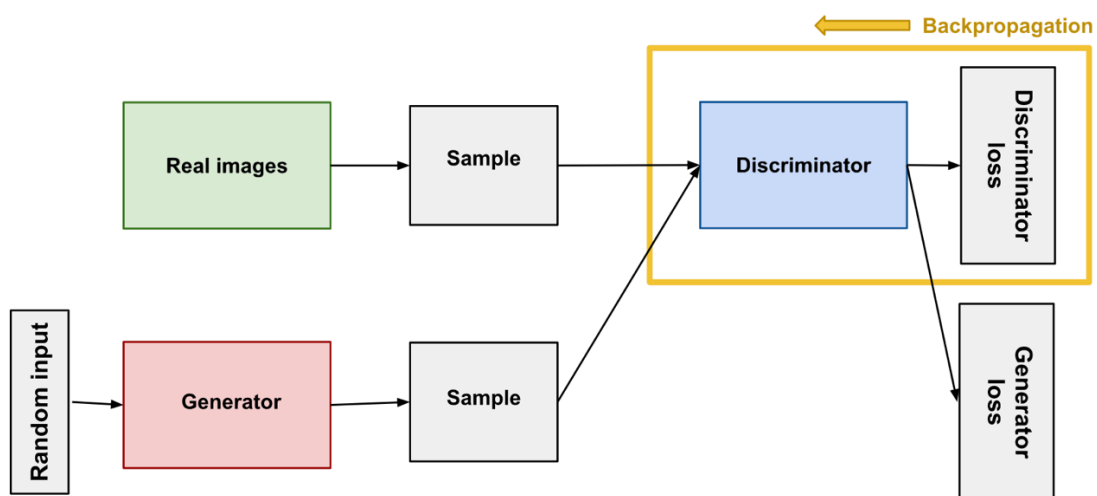


Рисунок 2.11 Фаза навчання дискримінаційної частини нейронної мережі

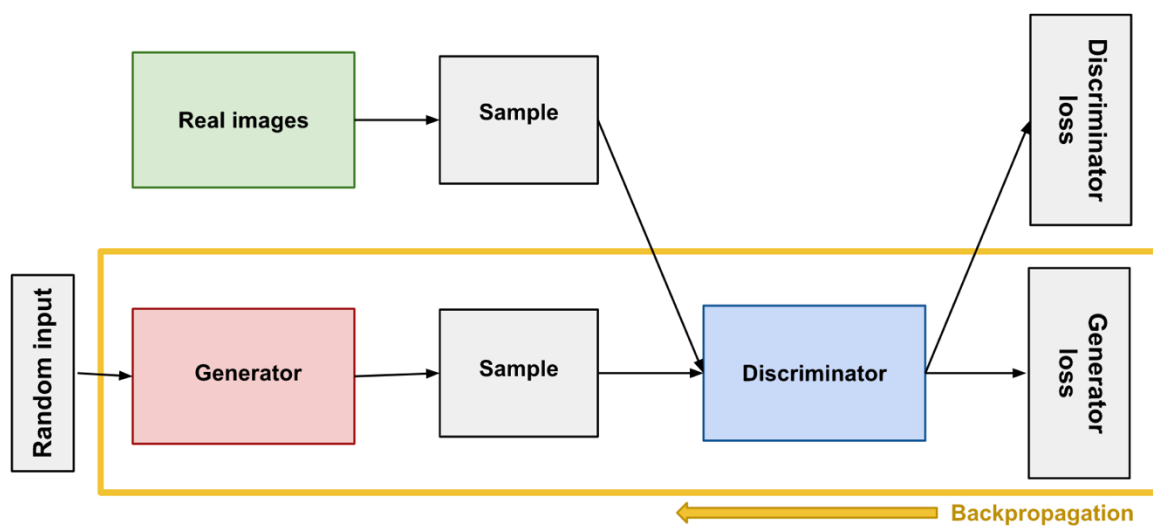


Рисунок 2.12 Фаза навчання генеративної частини нейронної мережі

2.4 Циклічні генеративні змагальні мережі

Для вирішення поставленої задачі найбільш доцільним є використання циклічних генеративних змагальних мереж, оскільки на відміну від інших методів вирішення задачі перенесення стилю, циклічні нейронні мережі здатні працювати при відсутності будь-якого зв'язку між наборами даних. Тобто, коли для довільного зображення невідомо який стиль воно б мало мати у іншому наборі даних та не відбувається перенесення стилю з конкретного зображення на інше, а стоїть задача виокремлення загального стилю з деякого набору даних.

Циклічні генеративні змагальні мережі реалізують вирішення задачі знаходження відображення зображення з початкової множини зображень X до цільової множини Y за відсутності спарених прикладів. Їх мета полягає у вивченні відображення $G: X \rightarrow Y$ такого, що розподіл зображень із $G(X)$ неможливо відрізнити від розподілу Y , використовуючи змагальну втрату. Оскільки це відображення дуже обмежене, його поєднують із відображенням $F: Y \rightarrow X$ і вводять втрату консистенції циклу для забезпечення $F(G(X)) \approx X$ (і навпаки).

Тобто, у даній архітектурі існує два генератора та два дискримінатори для множин X та Y відповідно (Рисунок 2.13).

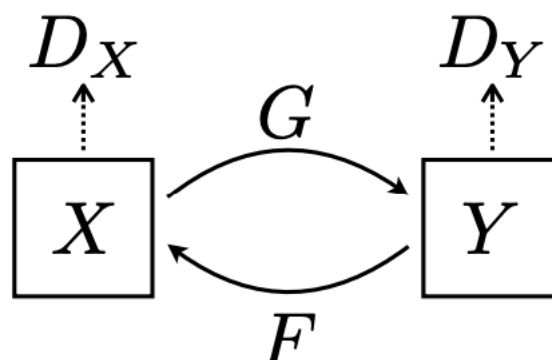


Рисунок 2.13 Схематичне зображення архітектури циклічної генеративної змагальної мережі

Кожний дискримінатор та генератор мають відповідно власну функцію втрат, але крім цього вводиться функція циклічних втрат, яка відображає наскільки точно виконуються умови $F(G(X)) \approx X$ та $G(F(Y)) \approx Y$ для зображень X та Y із відповідних наборів даних. Тобто, вимагається, щоб зображення, відображене в іншу множину, було схоже на себе, після відображення назад іншим генератором (Рисунок 2.14). Дана умова змушує генератори G та F бути «відповідними» один одному.

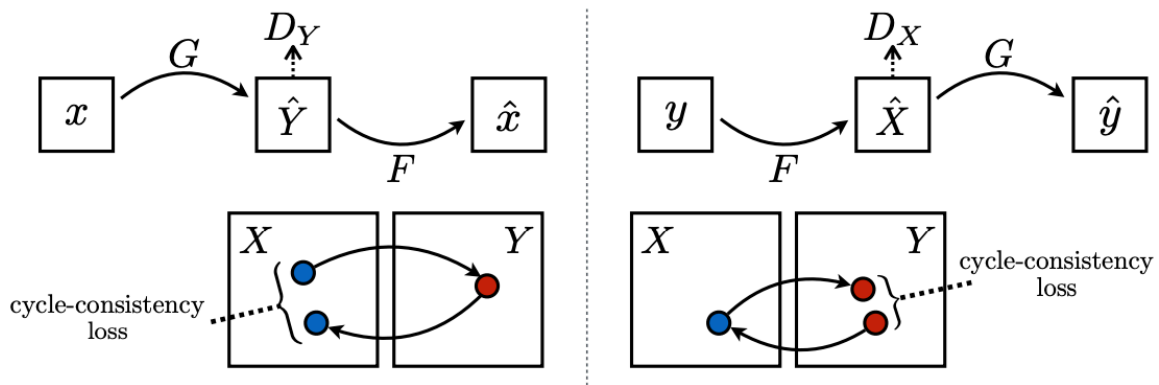


Рисунок 2.14 Ілюстрація умови консистенції циклового відображення

Окрім різниці в архітектурі нейронної мережі, циклічна генеративна змагальна мережа відрізняється від звичайної генеративної змагальної мережі функцією втрат. Одним із її доданків є функція втрат, представлена у пункті 2.3 (та її симетричний аналог для відображення з Y в X):

$$\begin{aligned} \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = & \mathbb{E}_{y \sim p_{\text{data}}(y)} [\log D_Y(y)] \\ & + \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log(1 - D_Y(G(x)))] \end{aligned} \quad (2.4.1)$$

Як вже було зазначено раніше, для забезпечення відповідності між генераторами G та F у функцію втрат вноситься наступний доданок:

$$\begin{aligned} \mathcal{L}_{\text{cyc}}(G, F) = & \mathbb{E}_{x \sim p_{\text{data}}(x)} [\|F(G(x)) - x\|_1] \\ & + \mathbb{E}_{y \sim p_{\text{data}}(y)} [\|G(F(y)) - y\|_1]. \end{aligned} \quad (2.4.2)$$

Загалом, функція втрат для циклічної генеративної змагальної мережі має наступний вигляд:

$$\begin{aligned} \mathcal{L}(G, F, D_X, D_Y) = & \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ & + \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \quad (2.4.3) \\ & + \lambda \mathcal{L}_{\text{cyc}}(G, F), \end{aligned}$$

де λ - гіперпараметр, що регулює відносну значимість функції циклічних втрат

Задача мережі полягає у знаходженні таких G^* та F^* , що:

$$G^*, F^* = \arg \min_{G, F} \max_{D_X, D_Y} \mathcal{L}(G, F, D_X, D_Y) \quad (2.4.4)$$

Кожна ітерація навчання циклічної генеративної змагальної мережі складається з двох симетричних частин: на вхід подається зображення з множини X (Рисунок 2.15) – знаходиться згенероване відображення, функції втрат і т.д., аналогічно – подається зображення з множини Y (Рисунок 2.16).

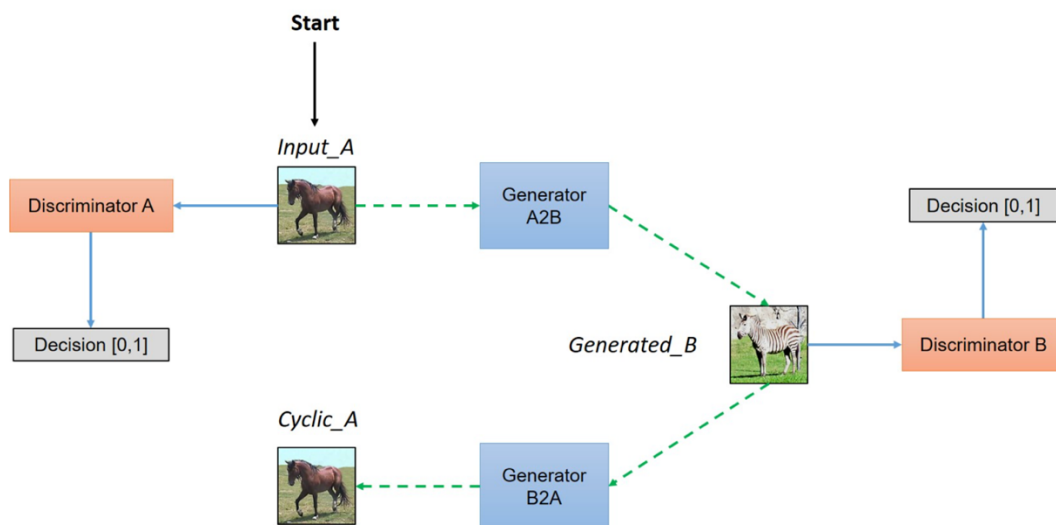


Рисунок 2.15 Перша частина ітерації навчання циклічної генеративної змагальної мережі

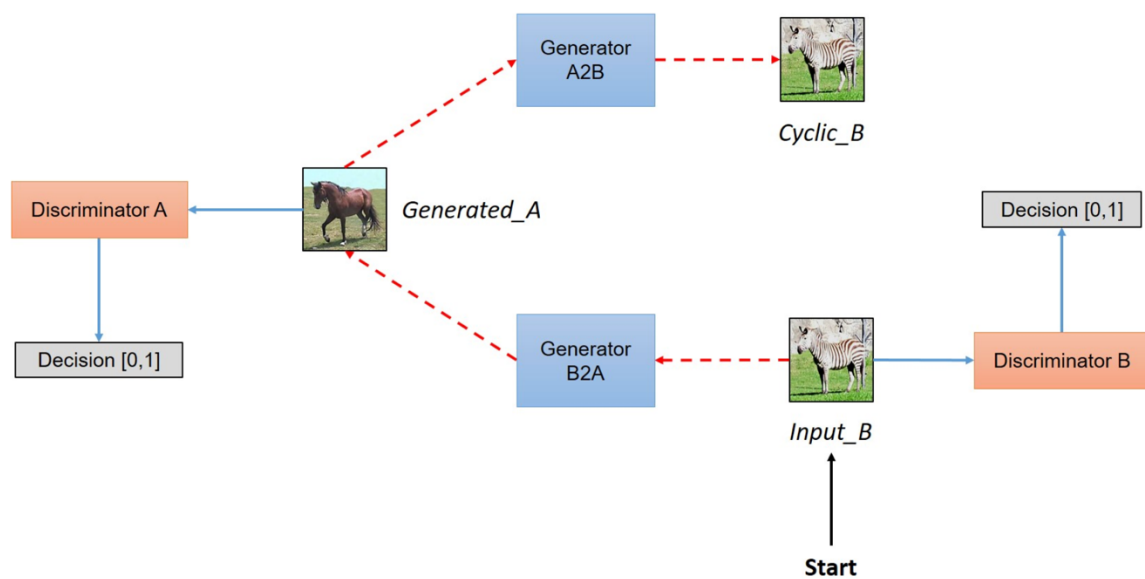


Рисунок 2.16 Друга частина ітерації навчання циклічної генеративної змагальної мережі

2.5 Огляд архітектури мереж генеративної та дискримінаційної частин циклічних генеративних змагальних мереж

2.5.1 U-Net

U-net (Рисунок 2.17) був винайдений і вперше використаний для біомедичної сегментації зображень. Його архітектуру можна у широкому сенсі розглядати як мережу кодера, за якою йде мережа декодера. На відміну від класифікації, де кінцевий результат глибинної мережі є єдиним важливим, семантична сегментація вимагає не лише дискримінації на рівні пікселів, а й механізму проектування дискримінаційних особливостей, засвоєних на різних етапах кодера, на піксельний простір.

Кодер - перша половина на діаграмі архітектури.

Декодер - друга половина архітектури. Мета полягає в семантичному проектуванні дискримінаційних ознак засвоєних кодером, на простір пікселів щоб отримати щільну класифікацію. Декодер складається з передискретизації та конкатенації з подальшим регулярним згортанням.

Основною задачею генеративної частини циклічної генеративної змагальної мережі є знаходження як можна більшої кількості семантичних та стилістичних ознак, з використанням яких відбувається генерація нових зображень.

Згорткова нейронна мережа U-Net в силу особливостей своєї архітектури здатна ефективно знаходити необхідні для цього ознаки в зображеннях, тому чудово підходить на роль генератора для рішення поставленої задачі.

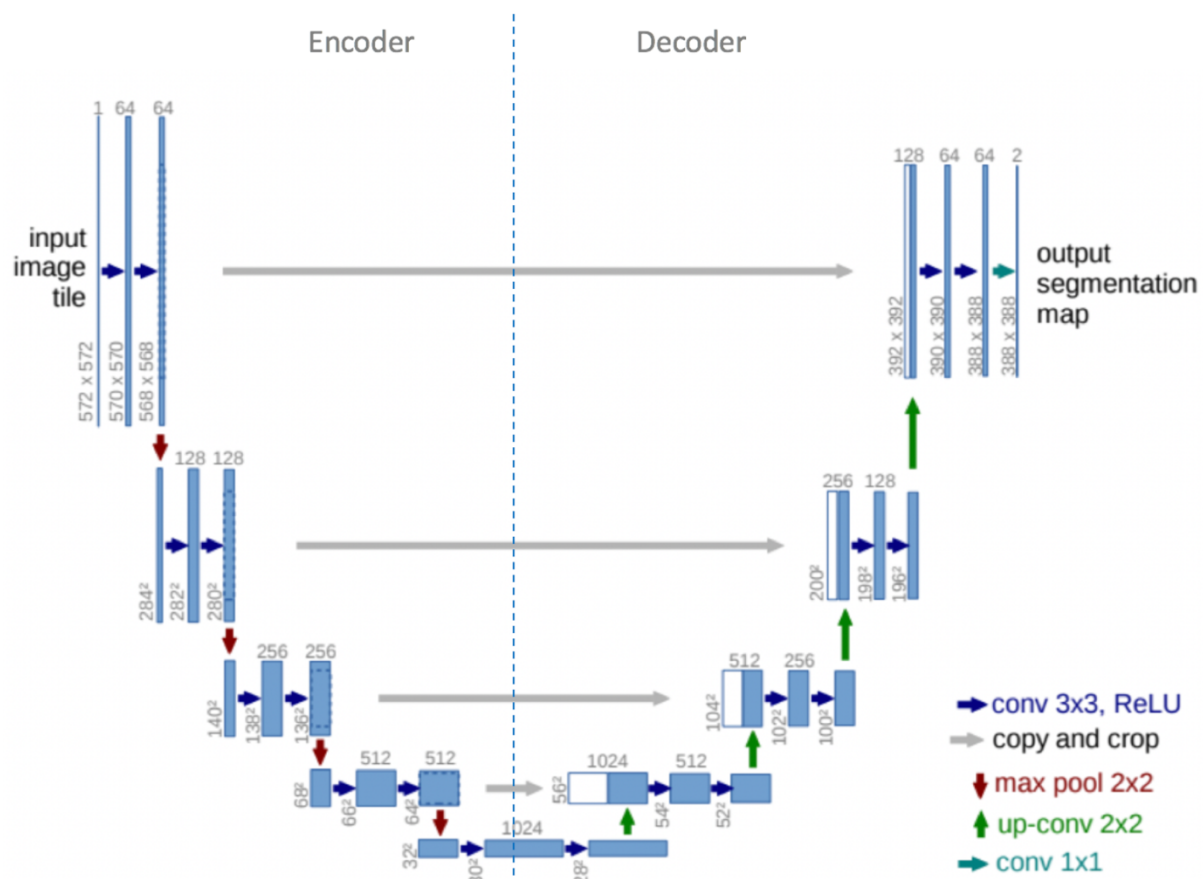


Рисунок 2.17 Архітектура згорткової нейронної мережі U-Net

2.5.2 PatchGAN

Задача дискримінатора полягає у розрізненні згенерованих генератором зображень від справжніх.

Ця задача вирішується з використанням згорткової нейронної мережі під назвою PatchGAN, що має досить просту архітектуру, яка складається з послідовного застосування згорткових шарів та шарів з функцією активації ReLU (Рисунок 2.18).

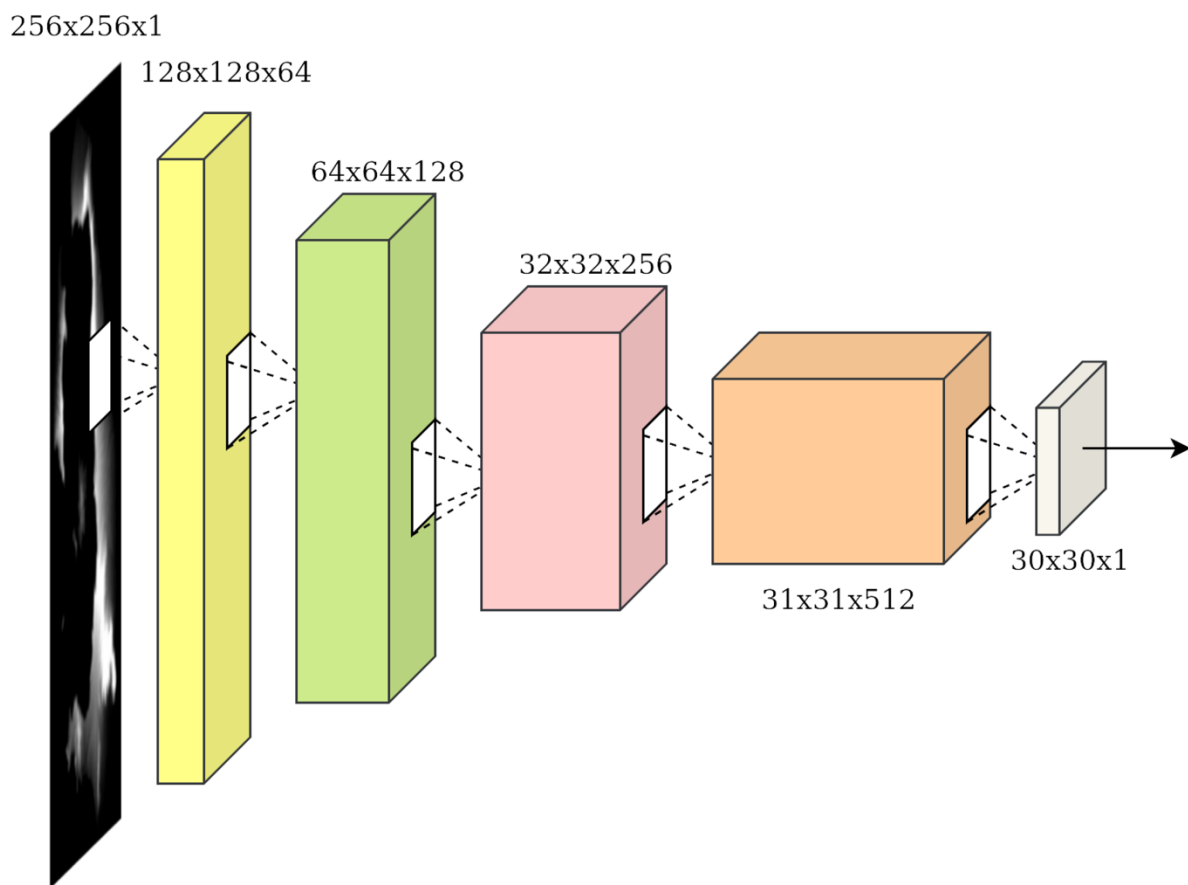


Рисунок 2.18 Архітектура згорткової нейронної мережі PatchNet

При цьому значення кожного елементу вихідної матриці відображає ймовірність, що відповідний кольоровий сегмент фотозображення розміру 70x70 належить зображенню зі справжнього набору даних, а не згенеровано генеративною частиною нейронної мережі (Рисунок 2.19).

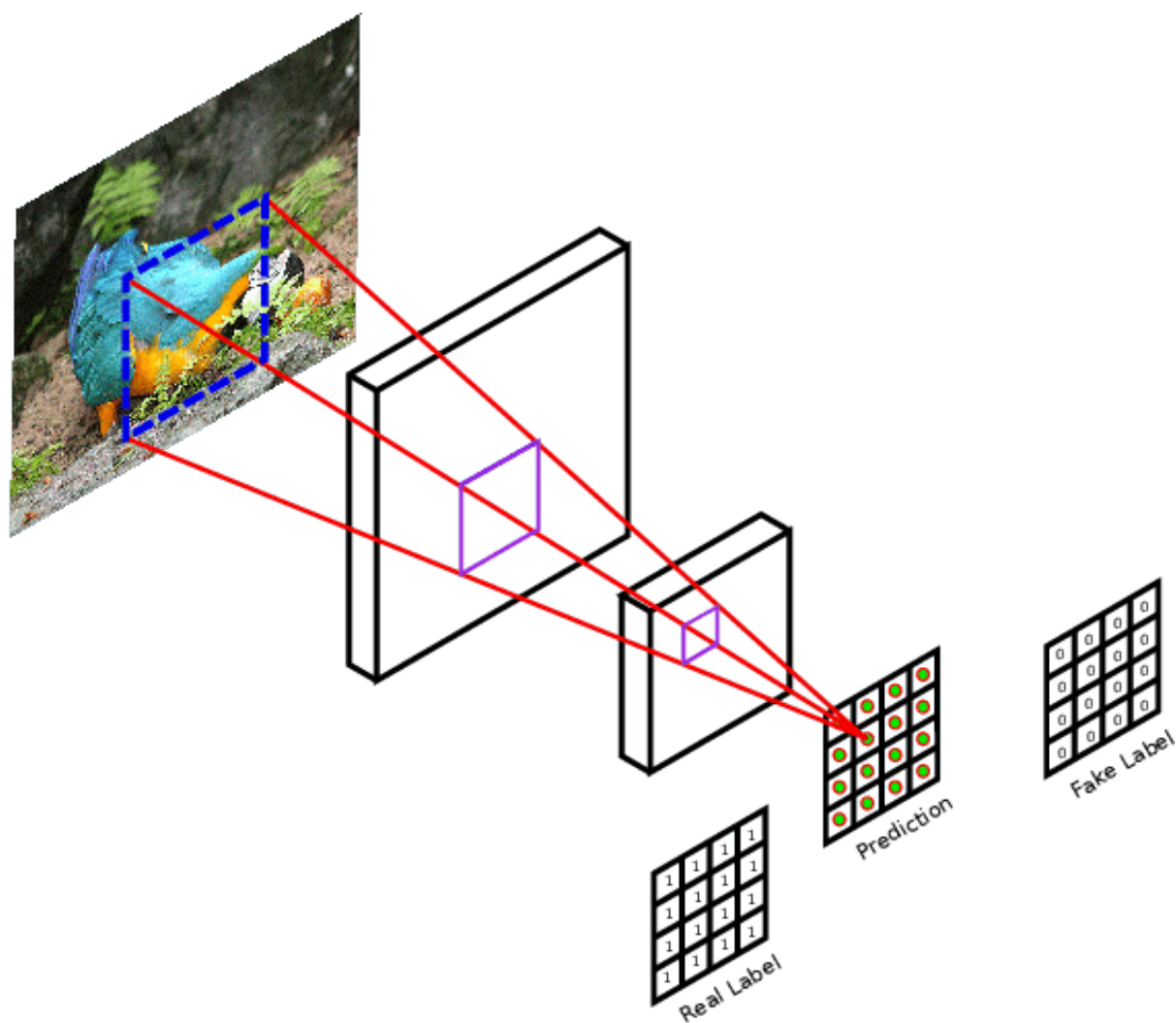


Рисунок 2.19 Відображення частини вхідного зображення розміру 70x70 у елемент вихідної матриці нейронної мережі архітектури PatchGAN

Висновки до розділу 2

Існує безліч різновидів нейронних мереж. Для вирішення задач комп'ютерного зору найчастіше використовуються згорткові нейронні мережі.

Було зроблено огляд генеративних змагальних мереж. Для вирішення поставленої задачі було обрано архітектуру циклічної генеративної змагальної мережі (CycleGAN), генеративна частина якої реалізує архітектуру U-Net, а дискримінативна - PatchGAN.

РОЗДІЛ 3 АРХІТЕКТУРА ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ ПОБУДОВАНОГО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вибір мови програмування

У якості найбільш вдалої мови програмування для вирішення поставленої задачі було обрано мову Python версії 3.

Python має декілька суттєвих переваг перед іншими мовами. Найголовніша з них полягає у тому, що для мови Python розроблено безліч бібліотек та фреймворків для аналізу даних, машинного навчання, проектування архітектури та впровадження нейронних мереж довільної складності. Крім того, існують засоби виконання коду на мові Python у більш ефективний спосіб, ніж на інших мовах з залучення сторонніх обчислювальних потужностей.

Серед стандартних характеристик мови слід зазначити, що у мові Python:

- Керування пам'яттю - цілком автоматичне — не потрібно хвилюватися щодо розподілу або звільнення пам'яті. Немає загрози “небезпечного посилання”.
- Типи зв'язані з об'єктами, а не зі змінними. Це означає, що змінній може бути призначене значення будь-якого типу, і що (наприклад) масив може містити об'єкти різних типів. Традиційні мови не надають такої можливості.
- Операції звичайно виконуються в більш високому рівні абстракції. Це частково результат того, як написана мова, і частково результат розширеної стандартної бібліотеки кодів, що поставляється разом з Python.

Незважаючи на те, що мова Python є інтерпретованою мовою з механізмом збору сміття, вона є достатньо швидкодіючою при роботі з нейронними мережами, оскільки, мова Python підтримує бібліотеки написані на мові C, що гарантує ефективну роботу з пам'яттю та швидкість виконання обчислень.

Оскільки Python є інтерпретованою мовою, це дозволяє виконувати окремі сегменти коду без необхідності перекомпіляції всієї програми, що робить більш зручною розробку програмного програмну та тестування окремих частин коду.

Лаконічність команд мови Python дозволяє писати менше коду для вирішення поставленої задачі, ніж було б затрачено з використанням будь-якої іншої мови подібного призначення.

3.2 Використання бібліотек та модулів

3.2.1 Вбудовані модулі

Окрім стандартних засобів мови Python у процесі виконання роботи було використано модуль OS, який забезпечує функції взаємодії з операційною системою. ОС підпадає під стандартні модулі Python. Цей модуль забезпечує портативний спосіб використання функціональних можливостей, що залежать від операційної системи.

У процесі виконання роботи даний модуль було використано для взаємодії з файловою системою, для завантаження та зберігання зображень, що були використані у ході роботи з циклічною генеративною змагальною мережею.

3.2.2 PIL

Бібліотека зображень Python (PIL) додає можливості обробки зображень до інтерпретатора Python.

Ця бібліотека забезпечує широку підтримку формату файлів, ефективне внутрішнє представлення та досить потужні можливості обробки зображень.

Основна бібліотека зображень розроблена для швидкого доступу до даних, що зберігаються у декількох основних піксельних форматах. Це забезпечує міцну основу для загального інструменту обробки зображень.

PIL пропонує кілька стандартних процедур для роботи із зображеннями.

До них належать:

- робота з окремими пікселями зображення
- маскування та обробка прозорості,

- фільтрація зображень, зокрема розмиття, контур, згладжування або знаходження країв об'єктів зображення
- покращення зображення, наприклад, регулювання різкості, регулювання яскравості, контрасту або кольору,
- додавання тексту до зображень та багато іншого.

Разом з OS, PIL була використана для зчитування, запису за файлової системи та обробки зображень.

3.2.3 Numpy

NumPy - основний пакет наукових обчислень на Python. Це бібліотека Python, яка забезпечує підтримку багатовимірних масивів, різні похідні об'єкти (наприклад, масковані масиви та матриці) та асортимент підпрограм для швидких операцій над масивами, включаючи математичні, логічні, маніпуляції з розмірностями об'єкту, сортування, вибір, введення / виведення, дискретні перетворення Фур'є, основні операції та структури лінійної алгебра, основні статистичні операції, моделювання випадкових подій та багато іншого.

В основі пакета NumPy лежить об'єкт `ndarray`, який інкапсулює n -вимірні масиви однорідних типів даних, причому багато операцій виконуються в скомпільованому коді для збільшення продуктивності. Існує кілька важливих відмінностей між масивами NumPy та стандартними послідовностями Python:

Масиви NumPy мають фіксований розмір при створенні, на відміну від списків Python (які можуть динамічно зростати). Зміна розміру `ndarray` створить новий масив і видалить оригінал.

Всі елементи масиву NumPy повинні мати однаковий тип даних, і, отже, матимуть однаковий розмір у пам'яті. Виняток: можна мати масиви об'єктів (Python, включаючи NumPy), що дозволяє мати масиви різних за розміром елементів.

Масиви NumPy сприяють більш швидкому виконанню математичних та інших типів операцій над великою кількістю даних. Як правило, такі операції виконуються швидше і потребують меншу кількість написаного коду, ніж це можливо за допомогою вбудованих послідовностей Python.

Зростаюча кількість науково-математичних пакетів на основі Python використовує масиви NumPy; хоча вони, як правило, підтримують і вбудовані масиви Python, вони перетворюють такі входи в масиви NumPy

перед обробкою, і часто виводять масиви NumPy в якості результату обчислень.

3.2.4 TensorFlow

Створена командою Google Brain, TensorFlow - це бібліотека з відкритим кодом для чисельних обчислень та масштабованого машинного навчання. TensorFlow поєднує в собі безліч моделей машинного навчання та глибокого навчання (нейронних мереж) та алгоритмів. Вона використовує Python для забезпечення зручного інтерфейсного API для побудови програм із фреймворком, виконуючи ці програми у високопродуктивному C ++.

TensorFlow може навчати та запускати глибокі нейронні мережі для рукописної класифікації цифр, розпізнавання зображень, вбудовування слів, рекурентних нейронних мереж, моделей послідовності в послідовність для машинного перекладу, обробки природної мови і т.п.

TensorFlow дозволяє розробникам створювати графи потоків даних - структури, що описують, як дані рухаються через графи або серію вузлів обробки. Кожен вузол на графі представляє математичну операцію, а кожне з'єднання або ребро між вузлами є багатовимірним масивом даних або тензором.

TensorFlow забезпечує все це для програміста за допомогою мови Python. Python простий у вивченні та роботі з ним, і він пропонує зручні способи висловити, як абстракції високого рівня можуть поєднуватися. Вузли та тензори в TensorFlow - це об'єкти Python, а програми TensorFlow - самі програми Python.

Однак фактичні математичні операції в Python не виконуються. Бібліотеки перетворень, доступні через TensorFlow, записані як високопродуктивні двійкові файли C ++. Python просто спрямовує трафік між фрагментами та забезпечує абстракції програмування високого рівня, щоб з'єднати їх між собою.

Додатки TensorFlow можна запускати на будь-якій зручній цілі: локальній машині, кластері в хмарі, пристроях iOS та Android, центральних процесорах або графічних процесорах. Якщо ви використовуєте власну хмару Google, ви можете запустити TensorFlow на спеціальному пристрої Google TensorFlow Processing Unit (TPU) для подальшого прискорення. Отримані в результаті моделі, створені TensorFlow, можуть бути розгорнуті на більшості будь-яких пристроїв, де вони будуть використовуватися для обслуговування прогнозів.

3.3 Архітектура програми

3.3.1 Навчання моделі

Програмний продукт складається з двох Jupyter Notebook. У першому з них відбувається побудова на навчання описаної циклічної генеративної змагальної мережі.

Для збільшення якості та швидкості навчання нейронної мережі застосовуються деякі техніки обробки наборів даних.

По кожному з кольорових каналів зображень відбувається нормування значення інтенсивності кольору з проміжку $[0, 255]$ у проміжок $[-1, 1]$.

Для боротьби з перенавчанням до кожного з зображень з деякою ймовірністю застосовуються деякі перетворення.

Таким чином кожне зображення з ймовірністю 0.5 буде “кропнуто” – збільшено в розмірі, а потім з нового зображення вирізається шматок меншого розміру. При цьому з ймовірністю 0.4 воно буде збільшено з розміру 256×256 до 286×286 , після чого буде взята частина зображення розміру 256×256 . З ймовірністю 0.1 окрім описаного перетворення, буде застосовано також збільшення в розмірі до 300×300 і вирізано частину зображення розміру 256×256 .

З ймовірністю 0.1 зображення буде повернуто на 90° проти годинникової стрілки, з ймовірністю 0.2 на 180° , з ймовірністю 0.2 на 270° .

Також серед перетворень були застосовані віддзеркалення, з ймовірністю 0.4. При цьому, зображення симетрично відображалось відносно вертикалі на горизонталі. До того ж, з ймовірністю 0.1, до зображення застосовувалося транспонування у сенсі матричних операцій.

Описані перетворення застосовувалися до обох наборів даних: звичайних фотозображень та картин Моне, перед початком навчання нейронної мережі.

Далі методами бібліотеки TensorFlow описувалась архітектура циклічної змагальної генеративної нейронної мережі, згідно обраній у минулому розділі.

Після навчання нейронної мережі, ваги зберігаються в окремому файлі і будуть використані для швидкого налаштування моделі в майбутньому без необхідності витрачання часу на навчання.

3.3.2 Використання моделі

У іншому файлі будується архітектура моделі, як і в попередньому, але на відміну від нього не відбувається навчання, а завантажуються ваги збережені з результату роботи попереднього етапу. Після чого можна обираючи зображення з набору даних звичайних фотографій, отримувати результат перенесення стилю картин Моне, використовуючи натреновану циклічну генеративну змагальну мережу.

3.4 Результати

Наведемо результати роботи програми на деяких зображеннях. Зліва – оригінал, справа – результат роботи нейронної мережі.



Рисунок 3.1 Приклад 1

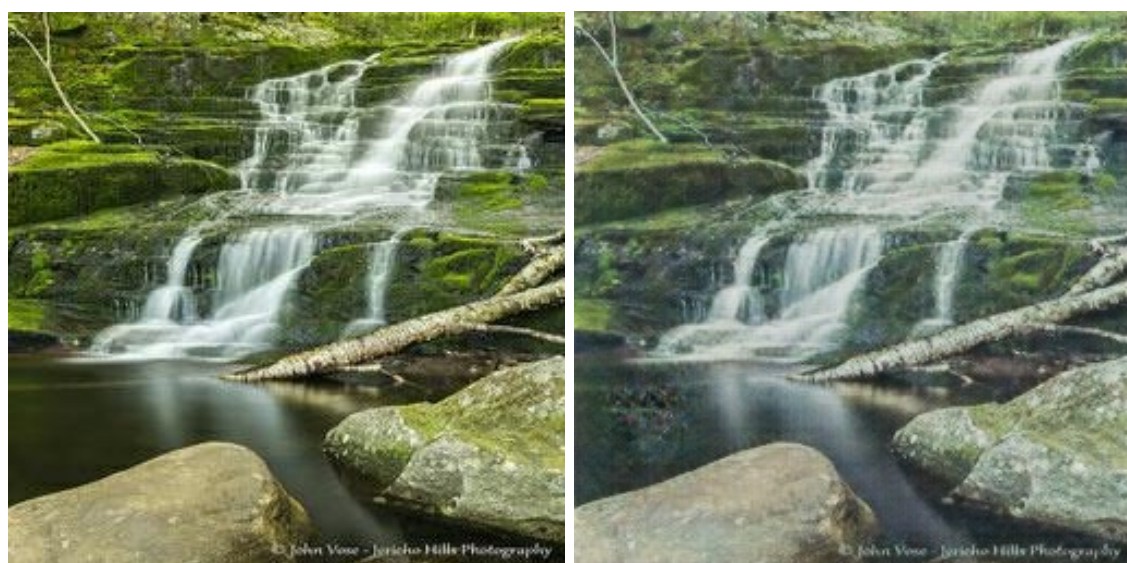


Рисунок 3.2 Приклад 2



Рисунок 3.3 Приклад 3



Рисунок 3.4 Приклад 4

Висновки до розділу 3

У даному розділі було описано обрані для реалізації програмного продукту інструменти вирішення поставленої задачі: вибір мови програмування, необхідні бібліотеки та модулі, архітектура програми. Продемонстровано результати роботи програмного продукту.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У даному розділі проводиться оцінка основних характеристик програмного продукту, розробленого для вирішення задачі художньої стилізації зображень за допомогою генеративних змагальних нейронних мереж.

Нижче наведено аналіз різних варіантів реалізації модулю з метою вибору оптимальної, з огляду при цьому як на економічні фактори, так і на характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. ФВА проводиться з метою виявлення резервів зниження витрат за рахунок ефективніших варіантів виробництва, кращого співвідношення між споживчою вартістю виробу та витратами на його виготовлення. Для проведення аналізу використовується економічна, технічна та конструкторська інформація.

Алгоритм функціонально-вартісного аналізу включає в себе визначення послідовності етапів розробки продукту, визначення повних витрат (річних) та кількості робочих часів, визначення джерел витрат та кінцевий розрахунок вартості програмного продукту.

4.1 Постановка задачі проектування

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки системи. Оскільки рішення стосовно проектування та реалізації компонентів, що розробляється, впливають на

всю систему, кожна окрема підсистема має її задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для збору, обробки та проведення аналізу масивів даних цифрових зображень.

Технічні вимоги до програмного продукту є наступні:

- функціонування на персональних комп'ютерах із стандартним набором компонентів;
- зручність та зрозумілість для користувача;
- швидкість обробки даних та доступ до інформації в реальному часі;
- можливість зручного масштабування та обслуговування;
- мінімальні витрати на впровадження програмного продукту.

4.2 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукту, який вирішує задачу художньої стилізації зображення. Беручи за основу цю функцію, можна виділити наступні:

F_1 – вибір мови програмування;

F_2 – вибір фреймворку машинного навчання;

F_3 – вибір середовища розробки.

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1 :

а) Python

б) C++

Функція F_2 :

а) Tensorflow;

б) Caffe

Функція F_3 :

- a) Jupyter Notebook;
- б) Visual Studio.

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1).

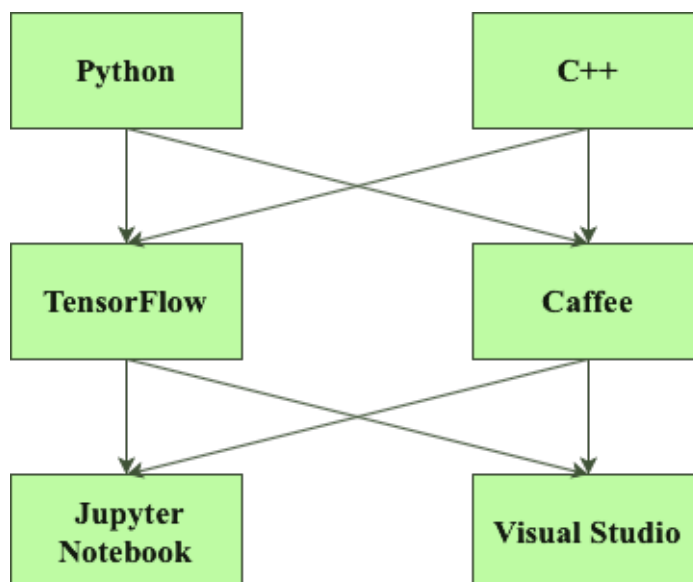


Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображає множину всіх можливих варіанти основних функцій.

Таблиця 4.2 - Позитивно-негативна матриця

Функції	Варіанти реалізації	Переваги	Недоліки
F_1	A	Швидка розробка програми, доступність бібліотек, кросплатформеність	Низька швидкість роботи, особливо, якщо потрібно обробляти велику кількість даних
	B	Код швидко виконується	Іде багато часу на розробку програми

F_2	A	Надійно працює з складними проектами	Не підтримується багатьма мовами
	B	Надійність	Додатковий час на інсталяцію та вивчення
F_3	A	Підтримується багатьма мовами програмування, легко запускається на будь-якому сервері	Відсутня можливість роботи без інтернету
	B	Багато інструментів, безпечна	Підтримує одночасно лише одну мову програмування

На основі цієї карти будуюмо позитивно-негативну матрицю варіантів основних функцій (Таблиця 4.2). Робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція F_1 :

Перевагу віддаємо швидкості вивчення, простоті використання та наявності стандартних бібліотек для обчислення. Для спрощення роботи по написанню коду варіант Б має бути відкинтий.

Функція F_2 :

Обидва варіанти можна використовувати в розробці.

Функція F_3 :

Віддаємо перевагу варіанту А в разі вибору мови програмування Python.

Таким чином, будемо розглядати такі варіанти реалізації ПП:

$F_1a - F_2a - F_3a$

$$F_1a - F_2b - F_3a$$

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.3 Обґрунтування системи параметрів ПП

На основі даних, розглянутих вище, визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- $X1$ – швидкодія мови програмування;
- $X2$ – об'єм пам'яті для обчислень та збереження даних;
- $X3$ – час навчання даних;
- $X4$ – потенційний об'єм програмного коду.

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у таблиці 4.4.

Таблиця 4.4 - Основні параметри ПП

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Швидкодія мови програмування	$X1$	оп/мс	10000	14000	19000
Об'єм пам'яті	$X2$	Мб	420	128	64
Час попередньої обробки даних	$X3$	мс	4	3	2

Потенційний об'єм програмного коду	X4	кількість рядків коду	4000	2500	1000
------------------------------------	----	-----------------------	------	------	------

За даними таблиці 4.3 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.5.

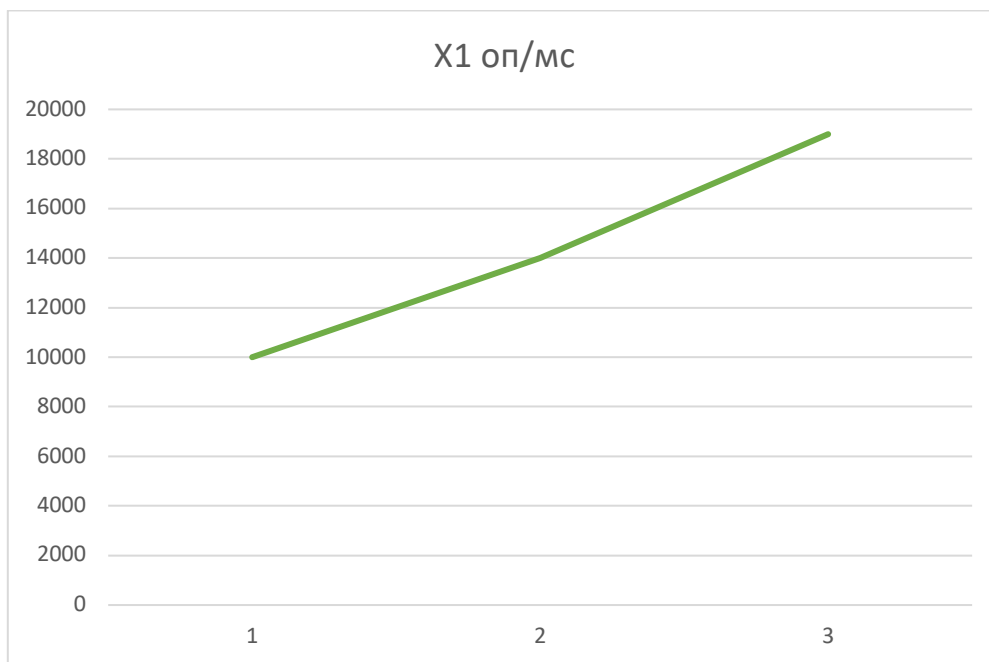


Рисунок 4.2 – X1, швидкодія мови програмування

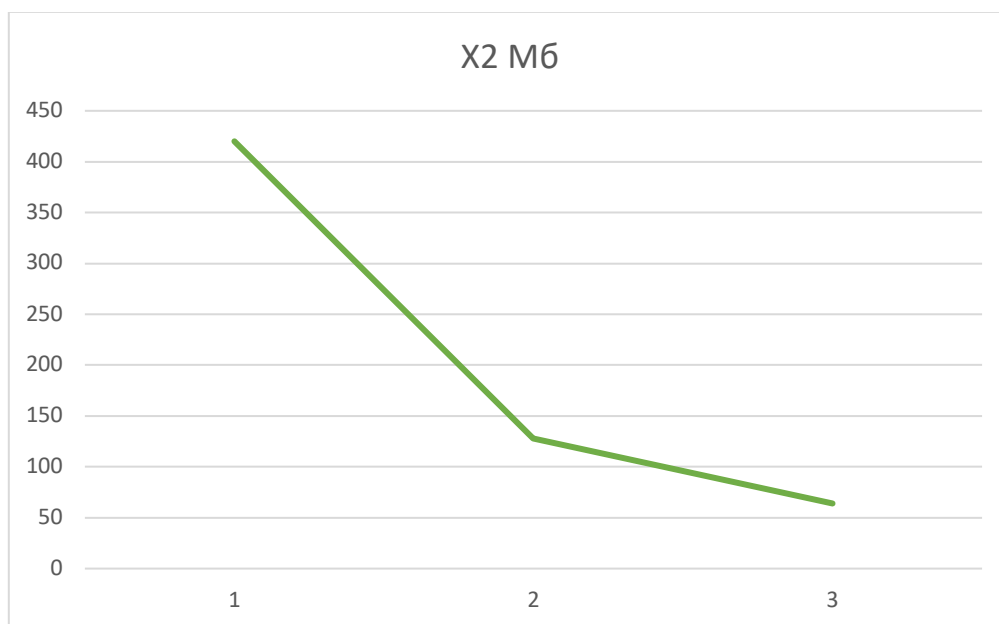


Рисунок 4.3 – X2, об'єм пам'яті

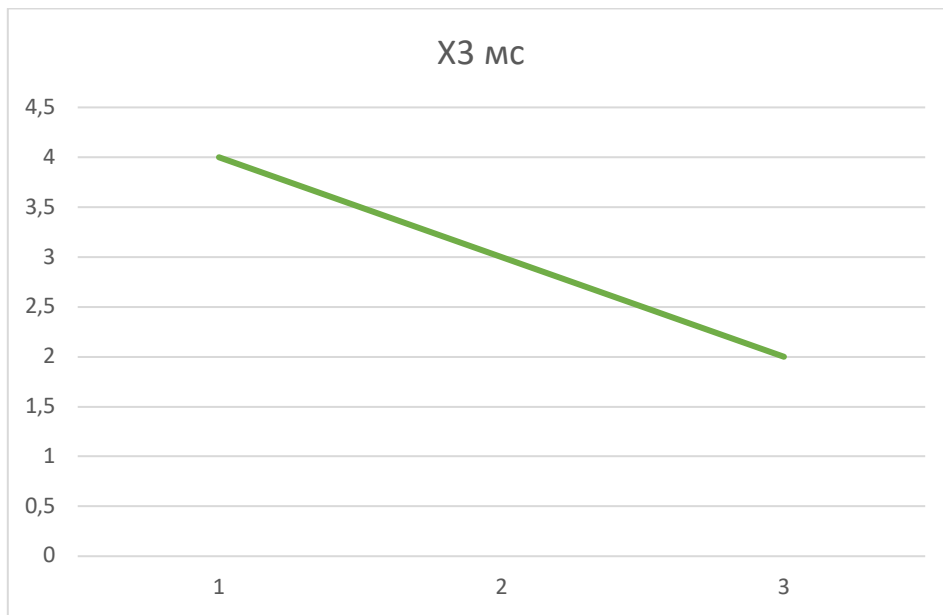


Рисунок 4.4 – X3, час попередньої обробки даних

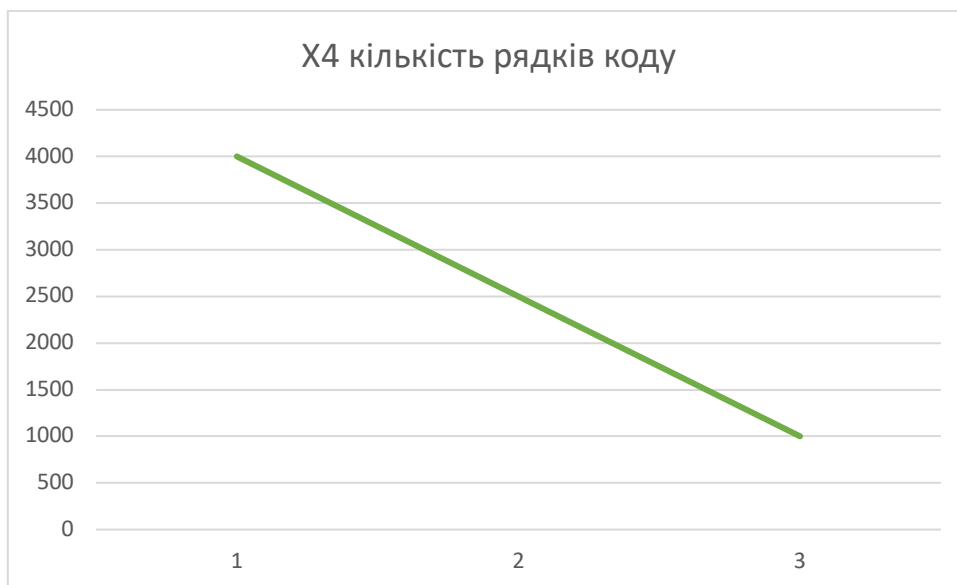


Рисунок 4.5 – X4, потенційний об'єм програмного коду

4.4 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при

знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.4.

Таблиця 4.4 - Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія мови програмування	Оп/мс	5	5	5	5	5	5	5	35	8,75	76,5625
X2	Об'єм пам'яті	Мб	3	3	2	2	3	1	3	17	-9,25	85,5625
X3	Час попередньої обробки даних	мс	2	2	3	3	3	4	3	20	-6,25	39,0625
X4	Потенційний об'єм програмного коду	Кількість рядків коду	5	5	5	5	4	5	4	33	6,75	45,5625

	Разом		15	15	15	15	15	15	15	105	0	246,75
--	-------	--	----	----	----	----	----	----	----	-----	---	--------

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^N r_{ij} R_{ij} = 105, \quad (4.1)$$

де N – число експертів,
 n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 26,25 \quad (4.2)$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T. \quad (4.3)$$

Сума відхилень по всіх параметрам повинна дорівнювати 0;

г) загальна сума квадратів відхилення:

$$S = \sum_{i=1}^N \Delta_i^2 = 246,75. \quad (4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 246,75}{7^2(4^3 - 4)} = 1,01 > W_k = 0,67. \quad (4.5)$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 5.5.

Таблиця 4.5 - Попарне порівняння параметрів.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	>	>	>	>	>	>	>	>	1,5
X1 і X3	>	>	>	>	>	>	>	>	1,5
X1 і X4	=	=	=	=	>	=	>	=	1,0
X2 і X3	>	>	<	<	=	<	=	<	0,5
X2 і X4	<	<	<	<	<	<	<	<	0,5
X3 і X4	<	<	<	<	<	<	<	<	0,5

Числове значення, що визначає ступінь переваги i -го параметра над j -тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases} \quad (4.6)$$

З отриманих числових оцінок переваги складемо матрицю $A = \|a_{ij}\|$.

Для кожного параметра зробимо розрахунок вагомості K_{vi} за наступними формулами:

$$K_{Bi} = \frac{b_i}{\sum_{i=1}^n b_i} \quad (4.7)$$

$$b_i = \sum_{j=1}^N a_{ij} \quad (4.8)$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятись від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{Bi} = \frac{b'_i}{\sum_{i=1}^n b'_i}, \quad (4.9)$$

$$b'_i = \sum_{j=1}^N a_{ij} b_j \quad (4.10)$$

Як видно з таблиці 4.6, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.6 - Розрахунок вагомості параметрів

Параметрих _i	Параметрих _j				Перша ітер.		Друга ітер.	
	X1	X2	X3	X4	b_i	K_{Bi}	b_i^1	K_{Bi}^1
X1	1,0	1,5	1,5	1,0	5,0	0,31	19	0,32
X2	0,5	1,0	0,5	0,5	2,5	0,16	9,25	0,16
X3	0,5	1,5	1,0	0,5	3,5	0,22	12,25	0,2
X4	1,0	1,5	1,5	1,0	5,0	0,31	19	0,32
Всього:					16	1	59.5	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів X_2 (Об'єм пам'яті), X_3 (час попередньої обробки даних) та X_4 (потенційний об'єм програмного коду) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра X_1 (швидкість роботи мови програмування) обрано не найгіршим.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 5.7):

$$K_K(j) = \sum_{i=1}^n K_{ei,j} B_{i,j}, \quad (4.11)$$

де n – кількість параметрів;

K_{ei} – коефіцієнт вагомості i -го параметра;

B_i – оцінка i -го параметра в балах.

Таблиця 4.7 - Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	A	X1	10000	6	0,32	1,92
F2	A	X2	64	5	0,16	0,8
	B	X2	128	2	0,2	0,4
F3	A	X3	1000	8	0,32	2,56

За даними з таблиці 4.7 за формулою:

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \dots + K_{\text{ТУ}}[F_{zk}], \quad (4.12)$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 1,92 + 0,8 + 2,56 = 5,28,$$

$$K_{K2} = 1,92 + 0,4 + 2,56 = 4,88.$$

Як видно з розрахунків, кращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;
2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як

$$T_0 = T_P \cdot K_{\Pi} \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}, \quad (4.13)$$

де T_P – трудомісткість розробки ПП;

K_{Π} – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

K_M – коефіцієнт рівня мови програмування;

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $T_P = 90$ людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_{\Pi} = 1.7$. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $K_{СК} = 1$. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $K_{СТ} = 0.8$. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 90 \cdot 1.7 \cdot 0.8 = 122.4 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $T_P = 27$ людино-днів, $K_{\Pi} = 0.9$, $K_{СК} = 1$, $K_{СТ} = 0.8$:

$$T_2 = 27 \cdot 0.9 \cdot 0.8 = 19.44 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$T_I = (122.4 + 19.44 + 4.8 + 19.44) \cdot 8 = 1328,64 \text{ людино-годин.}$$

$$T_{II} = (122.4 + 19.44 + 6.91 + 19.44) \cdot 8 = 1345.52 \text{ людино-годин.}$$

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь два програмісти з окладом 13550 грн., один аналітик в області даних з окладом 9850. Визначимо середню зарплату за годину за формулою:

$$C_q = \frac{M}{T_m \cdot t} \text{ грн.,} \quad (4.14)$$

де M – місячний оклад працівників;

T_m – кількість робочих днів тиждень;

t – кількість робочих годин в день.

$$C_q = \frac{13550 + 13550 + 9850}{3 \cdot 21 \cdot 8} = 73,31 \text{ грн.} \quad (4.15)$$

Тоді, розрахуємо заробітну плату за формулою:

$$C_{зп} = C_q \cdot T_i \cdot K_d, \quad (4.16)$$

де C_q – величина погодинної оплати праці програміста;

T_i – трудомісткість відповідного завдання;

K_d – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

$$\text{I.} \quad C_{\text{ЗП}} = 73,31 \cdot 1328.64 \cdot 1.2 = 116883,12 \text{ грн.}$$

$$\text{II.} \quad C_{\text{ЗП}} = 73,31 \cdot 1345.52 \cdot 1.2 = 118368,09 \text{ грн.}$$

Відрахування на єдиний соціальний внесок становить 22%:

$$\text{I.} \quad C_{\text{ВД}} = C_{\text{ЗП}} \cdot 0.22 = 116883,12 \cdot 0.22 = 25714,29 \text{ грн.}$$

$$\text{II.} \quad C_{\text{ВД}} = C_{\text{ЗП}} \cdot 0.22 = 118368,09 \cdot 0.22 = 26041,04 \text{ грн.}$$

Тепер визначимо витрати на оплату однієї машино-години. (C_M)

Так як одна ЕОМ обслуговує одного програміста з окладом 13550 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$C_M = 12 \cdot M \cdot K_3 = 12 \cdot 13550 \cdot 0,2 = 32520 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$C_{\text{ЗП}} = C_M \cdot (1 + K_3) = 32520 \cdot (1 + 0.2) = 39024 \text{ грн.}$$

Відрахування на соціальний внесок:

$$C_{\text{ВД}} = C_{\text{ЗП}} \cdot 0.22 = 39024 \cdot 0,22 = 8585,28 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 20000 грн.

$$C_A = K_{\text{ТМ}} \cdot K_A \cdot Ц_{\text{ПР}} = 1.15 \cdot 0.25 \cdot 20000 = 5750 \text{ грн.,}$$

де K_{TM} — коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

K_A — річна норма амортизації;

C_{PP} — договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$C_P = K_{TM} \cdot C_{PP} \cdot K_P = 1.15 \cdot 20000 \cdot 0.05 = 1150 \text{ грн.},$$

де K_P — відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$T_{EF} = (D_K - D_B - D_C - D_P) \cdot t_z \cdot K_B = (365 - 104 - 12 - 16) \cdot 8 \cdot 0.9 = 1677.6 \text{ годин},$$

де D_K — календарна кількість днів у році;

D_B, D_C — відповідно кількість вихідних та святкових днів;

D_P — кількість днів планових ремонтів устаткування;

t —кількість робочих годин в день;

K_B — коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$C_{EL} = T_{EF} \cdot N_C \cdot K_3 \cdot C_{EH} = 1677.6 \cdot 0.3 \cdot 0.3 \cdot 3.52 = 531.46 \text{ грн.},$$

де N_C — середньо-споживча потужність приладу;

K_3 — коефіцієнтом зайнятості приладу;

$\text{Ц}_{\text{ЕН}}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$C_H = \text{Ц}_{\text{ПР}} \cdot 0.67 = 20000 \cdot 0.67 = 13400 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$C_{\text{ЕКС}} = C_{\text{ЗП}} + C_{\text{ВІД}} + C_A + C_P + C_{\text{ЕЛ}} + C_H, \quad (4.17)$$

$$C_{\text{ЕКС}} = 39024 + 8585,28 + 5750 + 1150 + 531,46 + 13400 = 68440,74 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{\text{М-Г}} = C_{\text{ЕКС}} / T_{\text{ЕФ}} = 68440,74 / 1677.6 = 40,80 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов'язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$C_M = C_{\text{М-Г}} \cdot T, \quad (4.18)$$

$$\text{I.} \quad C_M = 40,80 \cdot 1328,64 = 54086,11 \text{ грн.}$$

$$\text{II.} \quad C_M = 40,80 \cdot 1345.52 = 54897,22 \text{ грн.}$$

Накладні витрати складають 67% від заробітної плати:

$$C_H = C_{\text{ЗП}} \cdot 0.67, \quad (4.19)$$

$$\text{I.} \quad C_H = 116883,12 \cdot 0.67 = 78311,69 \text{ грн.}$$

$$\text{II. } C_H = 118368,09 \cdot 0,67 = 79306,62 \text{ грн.}$$

Отже, вартість розробки ПП за варіантами становить:

$$C_{\text{ПП}} = C_{\text{ЗП}} + C_{\text{ВІД}} + C_{\text{М}} + C_{\text{Н}}, \quad (4.20)$$

$$\text{I. } C_{\text{ПП}} = 116883,12 + 25714,29 + 54086,11 + 78311,69 = 274995,21 \text{ грн.}$$

$$\text{II. } C_{\text{ПП}} = 118368,09 + 26041,04 + 54897,22 + 79306,62 = 278612,97 \text{ грн.}$$

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$K_{\text{ТЕР}j} = K_{\text{К}j} / C_{\text{Ф}j}, \quad (4.21)$$

$$K_{\text{ТЕР}1} = 5,28 / 274995,21 = 1,92 \cdot 10^{-5},$$

$$K_{\text{ТЕР}2} = 4,88 / 278612,97 = 1,75 \cdot 10^{-5}.$$

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $K_{\text{ТЕР}1} = 1,92 \cdot 10^{-5}$.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-

економічного

рівня

якості

$$K_{\text{TEP}} = 1,92 \cdot 10^{-5}.$$

Цей варіант реалізації програмного продукту має такі параметри:

- мова програмування – Python;
- Використання моделей з великою ємністю
- Використання стандартного інтерфейсу візуалізації, швидкість розробки

Даний варіант виконання програмного комплексу дає користувачу зручний інтерфейс, непоганий функціонал і швидкодію.

Висновки до розділу 4

Проведено повний функціонально-вартісний аналіз програмного продукту. Визначено та проведено оцінку основних функцій програмного продукту. Визначено параметри, які характеризують програмний продукт. Проведено експертне оцінювання параметрів та аналіз якості варіантів реалізації функцій.

Проведено економічний аналіз варіантів розробки – трудомісткість, витрати на заробітну плату та інші витрати.

На основі аналізу вибрано варіант реалізації програмного продукту.

ВИСНОВКИ

Перенесення стилю між зображеннями є складною для формалізації задачею. Методи її вирішення ґрунтуються на розкладенні зображення на семантичну та стильову складові. Для вирішення задачі перенесення стилю між множинами незалежних зображень застосовуються методи генеративних змагальних мереж, зокрема циклічні генеративні змагальні мережі.

У процесі виконання роботи було розроблено програмну реалізацію циклічної генеративної змагальної мережі для вирішення задачі перенесення стилю на мові програмування Python, на основі фреймворку проектування та реалізації нейронних мереж TensorFlow засобами Jupyter Notebook.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ian J. Goodfellow, Jean Pouget-Abadie*, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, Generative Adversarial Nets // Universite de Montr'eal, 2014.
2. Leon A. Gatys, Alexander S. Ecker, Matthias Bethge, A Neural Algorithm of Artistic Style // Baylor College of Medicine, 2015
3. Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros, Image-to-Image Translation with Conditional Adversarial Networks // Berkeley AI Research Laboratory, 2018
4. Jun-Yan Zhu, Taesung Park, Phillip Isola Alexei A. Efros, Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks // Berkeley AI Research laboratory, 2020
5. A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks // University of Toronto, 2012.
6. K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition", 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.
7. C. Szegedy et al., "Going deeper with convolutions", 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9, doi: 10.1109/CVPR.2015.7298594.
8. Y. Jing, Y. Yang, Z. Feng, J. Ye, Y. Yu and M. Song, "Neural Style Transfer: A Review", 2019 IEEE Transactions on Visualization and Computer Graphics, doi: 10.1109/TVCG.2019.2921336.
9. L. A. Gatys, A. S. Ecker, M. Bethge, A. Hertzmann and E. Shechtman, "Controlling Perceptual Factors in Neural Style Transfer," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 3730-3738, doi: 10.1109/CVPR.2017.397.

- 10.Z. Wang et al., "Diversified Arbitrary Style Transfer via Deep Feature Perturbation," 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 7786-7795, doi: 10.1109/CVPR42600.2020.00781.
- 11.D. Chen, L. Yuan, J. Liao, N. Yu and G. Hua, "StyleBank: An Explicit Representation for Neural Image Style Transfer," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 2770-2779, doi: 10.1109/CVPR.2017.296.

ДОДАТОК А КОД ПРОГРАМНОГО ПРОДУКТУ

Train.ipynb

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import cv2
import os
import tensorflow_addons as tfa

try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    print('Device:', tpu.master())
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
except:
    strategy = tf.distribute.get_strategy()

print('Number of replicas:', strategy.num_replicas_in_sync)

AUTO = tf.data.experimental.AUTOTUNE
print(tf.__version__)
!gdown --id 1QwqcGHYeqNjETTMahUb4uhU1Lnhj3-aO
!unzip dataset.zip
folder_monet = '/content/monet_jpg'
def load_images_from_folder(folder):
    images = []
    for filename in os.listdir(folder):
        img = cv2.imread(os.path.join(folder,filename))
        if img is not None:
```

```

        images.append(img)
    return images
monet_ds = load_images_from_folder(folder_monet)
plt.imshow(monet_ds[2])
folder_photo = '/content/photo_jpg'
photo_ds = load_images_from_folder(folder_photo)
plt.imshow(photo_ds[2500])
def normalize_image(image):
    image = (tf.cast(image, tf.float32) / 127.5) - 1
    return image
monet_ds = list(map(normalize_image, monet_ds))
photo_ds = list(map(normalize_image, photo_ds))
photo_ds = tf.data.Dataset.from_tensor_slices(photo_ds)
monet_ds = tf.data.Dataset.from_tensor_slices(monet_ds)
def data_augment(image):
    p_rotate = tf.random.uniform([], 0, 1.0, dtype=tf.float32)
    p_spatial = tf.random.uniform([], 0, 1.0, dtype=tf.float32)
    p_crop = tf.random.uniform([], 0, 1.0, dtype=tf.float32)

    #cropping
    if p_crop > .5:
        image = tf.image.resize(image, [286, 286])
        image = tf.image.random_crop(image, size=[256, 256, 3])
        if p_crop > .9:
            image = tf.image.resize(image, [300, 300])
            image = tf.image.random_crop(image, size=[256, 256, 3])

    #rotating
    if p_rotate > .9:
        image = tf.image.rot90(image, k=3)
    elif p_rotate > .7:
        image = tf.image.rot90(image, k=2)
    elif p_rotate > .5:
        image = tf.image.rot90(image, k=1)

```

```

# mirroring
if p_spatial > .6:
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)
    if p_spatial > .9:
        image = tf.image.transpose(image)

return image

def preprocessing(monet_ds, photo_ds, augment=None, repeat=True, shuffle=True,
batch_size=1):

    if augment:
        monet_ds = monet_ds.map(augment, num_parallel_calls=AUTO)
        photo_ds = photo_ds.map(augment, num_parallel_calls=AUTO)

    if repeat:
        monet_ds = monet_ds.repeat()
        photo_ds = photo_ds.repeat()
    if shuffle:
        monet_ds = monet_ds.shuffle(2048)
        photo_ds = photo_ds.shuffle(2048)

    monet_ds = monet_ds.batch(batch_size, drop_remainder=True)
    photo_ds = photo_ds.batch(batch_size, drop_remainder=True)
    monet_ds = monet_ds.cache()
    photo_ds = photo_ds.cache()
    monet_ds = monet_ds.prefetch(AUTO)
    photo_ds = photo_ds.prefetch(AUTO)

    gan_ds = tf.data.Dataset.zip((monet_ds, photo_ds))

    return gan_ds
BATCH_SIZE = 4

```

```

full_ds = preprocessing(monet_ds, photo_ds, augment=data_augment, repeat=True,
                        shuffle=True, batch_size=BATCH_SIZE)
example_monet, example_photo = next(iter(full_ds))
plt.subplot(121)
plt.title('Real photo')
plt.imshow(example_photo[0] * 0.5 + 0.5)
plt.subplot(122)
plt.title('Monet painting')
plt.imshow(example_monet[0] * 0.5 + 0.5)
OUTPUT_CHANNELS = 3

```

```

def downsample(filters, size, apply_instancenorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    result = keras.Sequential()
    # Convolutional layer
    result.add(layers.Conv2D(filters, size, strides=2, padding='same',
                             kernel_initializer=initializer, use_bias=False))
    # Normalization layer
    if apply_instancenorm:
        result.add(tf.layers.InstanceNormalization(gamma_initializer=gamma_init))
    # Activation layer
    result.add(layers.LeakyReLU())

    return result

def upsample(filters, size, apply_dropout=False):
    # Normalization layer
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    # Transpose convolutional layer
    result = keras.Sequential()

```

```

result.add(layers.Conv2DTranspose(filters, size, strides=2,
                                padding='same',
                                kernel_initializer=initializer,
                                use_bias=False))

#Instance Normalization
result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))

# Dropout layer
if apply_dropout:
    result.add(layers.Dropout(0.5))

# Activation layer
result.add(layers.ReLU())

return result

def Generator():
    inputs = layers.Input(shape=[256,256,3])

    # bs = batch size
    down_stack = [
        downsample(64, 4, apply_instancenorm=False), # (bs, 128, 128, 64)
        downsample(128, 4), # (bs, 64, 64, 128)
        downsample(256, 4), # (bs, 32, 32, 256)
        downsample(512, 4), # (bs, 16, 16, 512)
        downsample(512, 4), # (bs, 8, 8, 512)
        downsample(512, 4), # (bs, 4, 4, 512)
        downsample(512, 4), # (bs, 2, 2, 512)
        downsample(512, 4), # (bs, 1, 1, 512)
    ]

    up_stack = [
        upsample(512, 4, apply_dropout=True), # (bs, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 8, 8, 1024)
        upsample(512, 4), # (bs, 16, 16, 1024)
        upsample(256, 4), # (bs, 32, 32, 512)
    ]

```

```

        upsample(128, 4), # (bs, 64, 64, 256)
        upsample(64, 4), # (bs, 128, 128, 128)
    ]
    initializer = tf.random_normal_initializer(0., 0.02)
    last = layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                   strides=2,
                                   padding='same',
                                   kernel_initializer=initializer,
                                   activation='tanh') # (bs, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        x = layers.Concatenate()([x, skip])

    x = last(x)

    return keras.Model(inputs=inputs, outputs=x)

generator_g = Generator()
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    inp = layers.Input(shape=[256, 256, 3], name='input_image')

```

```

x = inp

down1 = downsample(64, 4, False)(x) # (bs, 128, 128, 64)
down2 = downsample(128, 4)(down1) # (bs, 64, 64, 128)
down3 = downsample(256, 4)(down2) # (bs, 32, 32, 256)

zero_pad1 = layers.ZeroPadding2D()(down3) # (bs, 34, 34, 256)
conv = layers.Conv2D(512, 4, strides=1,
                    kernel_initializer=initializer,
                    use_bias=False)(zero_pad1) # (bs, 31, 31, 512)

norm1 = tf.layers.InstanceNormalization(gamma_initializer=gamma_init)(conv)

leaky_relu = layers.LeakyReLU()(norm1)

zero_pad2 = layers.ZeroPadding2D()(leaky_relu) # (bs, 33, 33, 512)

last = layers.Conv2D(1, 4, strides=1,
                    kernel_initializer=initializer)(zero_pad2) # (bs, 30, 30, 1)

return tf.keras.Model(inputs=inp, outputs=last)
discriminator_y = Discriminator()
class CycleGan(keras.Model):
    def __init__(
        self,
        monet_generator,
        photo_generator,
        monet_discriminator,
        photo_discriminator,
        lambda_cycle=15,
    ):
        super(CycleGan, self).__init__()
        self.m_gen = monet_generator

```

```

self.p_gen = photo_generator
self.m_disc = monet_discriminator
self.p_disc = photo_discriminator
self.lambda_cycle = lambda_cycle

def compile(
    self,
    m_gen_optimizer,
    p_gen_optimizer,
    m_disc_optimizer,
    p_disc_optimizer,
    gen_loss_fn,
    disc_loss_fn,
    cycle_loss_fn,
    identity_loss_fn
):
    super(CycleGan, self).compile()
    self.m_gen_optimizer = m_gen_optimizer
    self.p_gen_optimizer = p_gen_optimizer
    self.m_disc_optimizer = m_disc_optimizer
    self.p_disc_optimizer = p_disc_optimizer
    self.gen_loss_fn = gen_loss_fn
    self.disc_loss_fn = disc_loss_fn
    self.cycle_loss_fn = cycle_loss_fn
    self.identity_loss_fn = identity_loss_fn

def train_step(self, batch_data):
    real_monet, real_photo = batch_data

    with tf.GradientTape(persistent=True) as tape:
        # photo to monet back to photo
        fake_monet = self.m_gen(real_photo, training=True)
        cycled_photo = self.p_gen(fake_monet, training=True)

```



```

# monet to photo back to monet
fake_photo = self.p_gen(real_monet, training=True)
cycled_monet = self.m_gen(fake_photo, training=True)

# generating itself
same_monet = self.m_gen(real_monet, training=True)
same_photo = self.p_gen(real_photo, training=True)

# discriminator used to check, inputing real images
disc_real_monet = self.m_disc(real_monet, training=True)
disc_real_photo = self.p_disc(real_photo, training=True)

# discriminator used to check, inputing fake images
disc_fake_monet = self.m_disc(fake_monet, training=True)
disc_fake_photo = self.p_disc(fake_photo, training=True)

# evaluates generator loss
monet_gen_loss = self.gen_loss_fn(disc_fake_monet)
photo_gen_loss = self.gen_loss_fn(disc_fake_photo)

# evaluates total cycle consistency loss
total_cycle_loss = self.cycle_loss_fn(real_monet, cycled_monet,
self.lambda_cycle) + self.cycle_loss_fn(real_photo, cycled_photo, self.lambda_cycle)

# evaluates total generator loss
total_monet_gen_loss = monet_gen_loss + total_cycle_loss +
self.identity_loss_fn(real_monet, same_monet, self.lambda_cycle)
total_photo_gen_loss = photo_gen_loss + total_cycle_loss +
self.identity_loss_fn(real_photo, same_photo, self.lambda_cycle)

# evaluates discriminator loss
monet_disc_loss = self.disc_loss_fn(disc_real_monet, disc_fake_monet)
photo_disc_loss = self.disc_loss_fn(disc_real_photo, disc_fake_photo)

```

```

# Calculate the gradients for generator and discriminator
monet_generator_gradients = tape.gradient(total_monet_gen_loss,
                                          self.m_gen.trainable_variables)
photo_generator_gradients = tape.gradient(total_photo_gen_loss,
                                          self.p_gen.trainable_variables)

monet_discriminator_gradients = tape.gradient(monet_disc_loss,
                                              self.m_disc.trainable_variables)
photo_discriminator_gradients = tape.gradient(photo_disc_loss,
                                              self.p_disc.trainable_variables)

# Apply the gradients to the optimizer
self.m_gen_optimizer.apply_gradients(zip(monet_generator_gradients,
                                         self.m_gen.trainable_variables))

self.p_gen_optimizer.apply_gradients(zip(photo_generator_gradients,
                                         self.p_gen.trainable_variables))

self.m_disc_optimizer.apply_gradients(zip(monet_discriminator_gradients,
                                         self.m_disc.trainable_variables))

self.p_disc_optimizer.apply_gradients(zip(photo_discriminator_gradients,
                                         self.p_disc.trainable_variables))

return {
    "monet_gen_loss": total_monet_gen_loss,
    "photo_gen_loss": total_photo_gen_loss,
    "monet_disc_loss": monet_disc_loss,
    "photo_disc_loss": photo_disc_loss
}

with strategy.scope():
    def discriminator_loss(real, generated):
        real_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True,
                                                         reduction=tf.keras.losses.Reduction.NONE)(tf.ones_like(real), real)

```

```

generated_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True,
reduction=tf.keras.losses.Reduction.NONE)(tf.zeros_like(generated), generated)

total_disc_loss = real_loss + generated_loss

return total_disc_loss * 0.4
with strategy.scope():
    def generator_loss(generated):
        return tf.keras.losses.BinaryCrossentropy(from_logits=True,

reduction=tf.keras.losses.Reduction.NONE)(tf.ones_like(generated), generated)
    with strategy.scope():
        def calc_cycle_loss(real_image, cycled_image, LAMBDA):
            loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))

            return LAMBDA * loss1
        with strategy.scope():
            def identity_loss(real_image, same_image, LAMBDA):
                loss = tf.reduce_mean(tf.abs(real_image - same_image))
                return LAMBDA * 0.5 * loss
            with strategy.scope():
                monet_generator_optimizer = tf.keras.optimizers.Adam(0.002, decay = 0,
beta_1=0.5)
                photo_generator_optimizer = tf.keras.optimizers.Adam(0.002, beta_1=0.5)

                monet_discriminator_optimizer = tf.keras.optimizers.Adam(0.002, beta_1=0.5)
                photo_discriminator_optimizer = tf.keras.optimizers.Adam(0.002, beta_1=0.5)
            with strategy.scope():
                monet_generator = Generator() # transforms photos to Monet-esque paintings
                photo_generator = Generator() # transforms Monet paintings to be more like photos

                monet_discriminator = Discriminator() # differentiates real Monet paintings and
generated Monet paintings

```

```

photo_discriminator = Discriminator() # differentiates real photos and generated
photos
with strategy.scope():
    cycle_gan_model = CycleGan(
        monet_generator, photo_generator, monet_discriminator, photo_discriminator
    )

    cycle_gan_model.compile(
        m_gen_optimizer = monet_generator_optimizer,
        p_gen_optimizer = photo_generator_optimizer,
        m_disc_optimizer = monet_discriminator_optimizer,
        p_disc_optimizer = photo_discriminator_optimizer,
        gen_loss_fn = generator_loss,
        disc_loss_fn = discriminator_loss,
        cycle_loss_fn = calc_cycle_loss,
        identity_loss_fn = identity_loss
    )

n_monet_samples = 300
n_photo_samples = 7038
EPOCHS_NUM = 50
cycle_gan_model.fit(
    full_ds,
    epochs=EPOCHS_NUM,
    steps_per_epoch=(max(n_monet_samples, n_photo_samples)//BATCH_SIZE),
)

example_monet , example_photo = next(iter(full_ds))
example_gen_output_monet_fake = monet_generator(example_photo, training=False)
example_gen_output_photo_cycle =
photo_generator(example_gen_output_monet_fake, training=False)
example_gen_output_photo_real = photo_generator(example_photo, training=False)

# We execute the discriminator for Photo (real)
example_disc_out_photo_real = photo_discriminator(example_photo, training=False)

```

```

# We run the discriminator for Monet (fake)
example_disc_out_monet_fake =
monet_discriminator(example_gen_output_monet_fake, training=False)

# from Monet we generate photo (fake) and regenerate Monet (cycle) again
example_gen_output_photo_fake = photo_generator(example_monet, training=False)
example_gen_output_monet_cycle =
monet_generator(example_gen_output_photo_fake, training=False)
example_gen_output_monet_same = monet_generator(example_monet,
training=False)

# We run the discriminator for Monet (real)
example_disc_out_monet_real = monet_discriminator(example_monet,
training=False)

# We execute the discriminator for Photo (fake)
example_disc_out_photo_fake =
photo_discriminator(example_gen_output_photo_fake, training=False)
plt.figure(figsize=(10,10))

# photo
plt.subplot(4,4,1)
plt.imshow(example_photo[0] * 0.5 + 0.5)

# Monet
plt.subplot(4,4,2)
plt.imshow(example_gen_output_monet_fake[0] * 0.5 + 0.5)

# Photo Cycle
plt.subplot(4,4,3)
plt.imshow(example_gen_output_photo_cycle[0] * 0.5 + 0.5)

```

```

# Photo Same
plt.subplot(4,4,4)
plt.imshow(example_gen_output_photo_same[0] * 0.5 + 0.5)

# Discriminador Photo (real)
plt.subplot(4,4,5)
m = example_disc_out_photo_real[0,...,-1].numpy()
plt.imshow(m, vmin=np.min(m), vmax=np.max(m), cmap='RdBu_r')
#plt.colorbar(im,fraction=0.046, pad=0.04)

# Discriminador Monet (fake)
plt.subplot(4,4,7)
m = example_disc_out_monet_fake[0,...,-1].numpy()
plt.imshow(m, vmin=np.min(m), vmax=np.max(m), cmap='RdBu_r')
#plt.colorbar(im,fraction=0.046, pad=0.04)

# Monet
plt.subplot(4,4,9)
plt.imshow(example_monet[0] * 0.5 + 0.5)

# Foto generado
plt.subplot(4,4,10)
plt.imshow(example_gen_output_photo_fake[0] * 0.5 + 0.5)

# Monet Cycle
plt.subplot(4,4,11)
plt.imshow(example_gen_output_monet_cycle[0] * 0.5 + 0.5)

# Monet Same
plt.subplot(4,4,12)
plt.imshow(example_gen_output_monet_same[0] * 0.5 + 0.5)

# Discriminador Monet (real)
plt.subplot(4,4,13)
m = example_disc_out_monet_real[0,...,-1].numpy()

```

```

plt.imshow(m, vmin=np.min(m), vmax=np.max(m), cmap='RdBu_r')
#plt.colorbar(im,fraction=0.046, pad=0.04)
# Discriminador photo(fake)
plt.subplot(4,4,15)
m = example_disc_out_photo_fake[0,...,-1].numpy()
plt.imshow(m, vmin=np.min(m), vmax=np.max(m), cmap='RdBu_r')
#plt.colorbar(im,fraction=0.046, pad=0.04)

plt.show()
example_monet , example_photo = next(iter(full_ds))
example_gen_output_monet_fake = monet_generator(example_photo, training=False)
plt.subplot(1,2,1)
plt.imshow(example_photo[0] * 0.5 + 0.5)

# Monet
plt.subplot(1,2,2)
plt.imshow(example_gen_output_monet_fake[0] * 0.5 + 0.5)

plt.show()
path_to_photos = '/content/drive/MyDrive/Colab Notebooks/diplom/photo_jpg'
output_path = '/content/drive/MyDrive/Colab Notebooks/diplom/predicted/'
import PIL
arr = os.listdir(path_to_photos)
for i, filename in enumerate(arr):
    photo = cv2.imread(os.path.join(path_to_photos,filename))
    photo = normalize_image(photo)
    result = monet_generator(tf.expand_dims(photo, axis=0), training=False)
    result = (result * 127.5 + 127.5).numpy().astype(np.uint8)
    im = PIL.Image.fromarray(result[0])
    im.save(f'{output_path}{filename}')
cycle_gan_model.save_weights(os.path.join('/content/drive/MyDrive/Colab
Notebooks/diplom', 'weights.h5'), overwrite=True)

```

Results.ipynb

```

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import cv2
import os
import tensorflow_addons as tfa
import PIL
from PIL import Image
OUTPUT_CHANNELS = 3

def downsample(filters, size, apply_instancenorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    result = keras.Sequential()
    # Convolutional layer
    result.add(layers.Conv2D(filters, size, strides=2, padding='same',
                             kernel_initializer=initializer, use_bias=False))
    # Normalization layer
    if apply_instancenorm:
        result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))
    # Activation layer
    result.add(layers.LeakyReLU())

    return result

def upsample(filters, size, apply_dropout=False):
    # Normalization layer
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    # Transpose convolutional layer
    result = keras.Sequential()
    result.add(layers.Conv2DTranspose(filters, size, strides=2,
                                       padding='same',
                                       kernel_initializer=initializer,
                                       use_bias=False))
    #Instance Normalization
    result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))
    # Dropout layer
    if apply_dropout:
        result.add(layers.Dropout(0.5))
    # Activation layer
    result.add(layers.ReLU())

```



```

return result

def Generator():
    inputs = layers.Input(shape=[256,256,3])

    # bs = batch size
    down_stack = [
        downsample(64, 4, apply_instancenorm=False), # (bs, 128, 128, 64)
        downsample(128, 4), # (bs, 64, 64, 128)
        downsample(256, 4), # (bs, 32, 32, 256)
        downsample(512, 4), # (bs, 16, 16, 512)
        downsample(512, 4), # (bs, 8, 8, 512)
        downsample(512, 4), # (bs, 4, 4, 512)
        downsample(512, 4), # (bs, 2, 2, 512)
        downsample(512, 4), # (bs, 1, 1, 512)
    ]

    up_stack = [
        upsample(512, 4, apply_dropout=True), # (bs, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 8, 8, 1024)
        upsample(512, 4), # (bs, 16, 16, 1024)
        upsample(256, 4), # (bs, 32, 32, 512)
        upsample(128, 4), # (bs, 64, 64, 256)
        upsample(64, 4), # (bs, 128, 128, 128)
    ]
    initializer = tf.random_normal_initializer(0., 0.02)
    last = layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                  strides=2,
                                  padding='same',
                                  kernel_initializer=initializer,
                                  activation='tanh') # (bs, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        x = layers.Concatenate()([x, skip])

    x = last(x)

```

```

    return keras.Model(inputs=inputs, outputs=x)
generator_g = Generator()
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    inp = layers.Input(shape=[256, 256, 3], name='input_image')

    x = inp

    down1 = downsample(64, 4, False)(x) # (bs, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (bs, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (bs, 32, 32, 256)

    zero_pad1 = layers.ZeroPadding2D()(down3) # (bs, 34, 34, 256)
    conv = layers.Conv2D(512, 4, strides=1,
                        kernel_initializer=initializer,
                        use_bias=False)(zero_pad1) # (bs, 31, 31, 512)

    norm1 = tf.layers.InstanceNormalization(gamma_initializer=gamma_init)(conv)

    leaky_relu = layers.LeakyReLU()(norm1)

    zero_pad2 = layers.ZeroPadding2D()(leaky_relu) # (bs, 33, 33, 512)

    last = layers.Conv2D(1, 4, strides=1,
                        kernel_initializer=initializer)(zero_pad2) # (bs, 30, 30, 1)

    return tf.keras.Model(inputs=inp, outputs=last)
discriminator_y = Discriminator()
class CycleGan(keras.Model):
    def __init__(
        self,
        monet_generator,
        photo_generator,
        monet_discriminator,
        photo_discriminator,
        lambda_cycle=15,
    ):
        super(CycleGan, self).__init__()
        self.m_gen = monet_generator
        self.p_gen = photo_generator
        self.m_disc = monet_discriminator
        self.p_disc = photo_discriminator
        self.lambda_cycle = lambda_cycle

    def compile(
        self,
        m_gen_optimizer,

```

```

p_gen_optimizer,
m_disc_optimizer,
p_disc_optimizer,
gen_loss_fn,
disc_loss_fn,
cycle_loss_fn,
identity_loss_fn
):
    super(CycleGan, self).compile()
    self.m_gen_optimizer = m_gen_optimizer
    self.p_gen_optimizer = p_gen_optimizer
    self.m_disc_optimizer = m_disc_optimizer
    self.p_disc_optimizer = p_disc_optimizer
    self.gen_loss_fn = gen_loss_fn
    self.disc_loss_fn = disc_loss_fn
    self.cycle_loss_fn = cycle_loss_fn
    self.identity_loss_fn = identity_loss_fn

def train_step(self, batch_data):
    real_monet, real_photo = batch_data

    with tf.GradientTape(persistent=True) as tape:
        # photo to monet back to photo
        fake_monet = self.m_gen(real_photo, training=True)
        cycled_photo = self.p_gen(fake_monet, training=True)

        # monet to photo back to monet
        fake_photo = self.p_gen(real_monet, training=True)
        cycled_monet = self.m_gen(fake_photo, training=True)

        # generating itself
        same_monet = self.m_gen(real_monet, training=True)
        same_photo = self.p_gen(real_photo, training=True)

        # discriminator used to check, inputting real images
        disc_real_monet = self.m_disc(real_monet, training=True)
        disc_real_photo = self.p_disc(real_photo, training=True)

        # discriminator used to check, inputting fake images
        disc_fake_monet = self.m_disc(fake_monet, training=True)
        disc_fake_photo = self.p_disc(fake_photo, training=True)

        # evaluates generator loss
        monet_gen_loss = self.gen_loss_fn(disc_fake_monet)
        photo_gen_loss = self.gen_loss_fn(disc_fake_photo)

        # evaluates total cycle consistency loss
        total_cycle_loss = self.cycle_loss_fn(real_monet, cycled_monet, self.lambda_cycle) +
self.cycle_loss_fn(real_photo, cycled_photo, self.lambda_cycle)

```

```

    # evaluates total generator loss
    total_monet_gen_loss = monet_gen_loss + total_cycle_loss +
self.identity_loss_fn(real_monet, same_monet, self.lambda_cycle)
    total_photo_gen_loss = photo_gen_loss + total_cycle_loss +
self.identity_loss_fn(real_photo, same_photo, self.lambda_cycle)

    # evaluates discriminator loss
    monet_disc_loss = self.disc_loss_fn(disc_real_monet, disc_fake_monet)
    photo_disc_loss = self.disc_loss_fn(disc_real_photo, disc_fake_photo)

    # Calculate the gradients for generator and discriminator
    monet_generator_gradients = tape.gradient(total_monet_gen_loss,
                                              self.m_gen.trainable_variables)
    photo_generator_gradients = tape.gradient(total_photo_gen_loss,
                                              self.p_gen.trainable_variables)

    monet_discriminator_gradients = tape.gradient(monet_disc_loss,
                                                  self.m_disc.trainable_variables)
    photo_discriminator_gradients = tape.gradient(photo_disc_loss,
                                                  self.p_disc.trainable_variables)

    # Apply the gradients to the optimizer
    self.m_gen_optimizer.apply_gradients(zip(monet_generator_gradients,
                                              self.m_gen.trainable_variables))

    self.p_gen_optimizer.apply_gradients(zip(photo_generator_gradients,
                                              self.p_gen.trainable_variables))

    self.m_disc_optimizer.apply_gradients(zip(monet_discriminator_gradients,
                                              self.m_disc.trainable_variables))

    self.p_disc_optimizer.apply_gradients(zip(photo_discriminator_gradients,
                                              self.p_disc.trainable_variables))
    return {
        "monet_gen_loss": total_monet_gen_loss,
        "photo_gen_loss": total_photo_gen_loss,
        "monet_disc_loss": monet_disc_loss,
        "photo_disc_loss": photo_disc_loss
    }
}

def discriminator_loss(real, generated):
    real_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True,
reduction=tf.keras.losses.Reduction.NONE)(tf.ones_like(real), real)

    generated_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True,
reduction=tf.keras.losses.Reduction.NONE)(tf.zeros_like(generated), generated)

    total_disc_loss = real_loss + generated_loss

    return total_disc_loss * 0.4
def generator_loss(generated):

```

```

return tf.keras.losses.BinaryCrossentropy(from_logits=True,
reduction=tf.keras.losses.Reduction.NONE)(tf.ones_like(generated), generated)
def calc_cycle_loss(real_image, cycled_image, LAMBDA):
    loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))

    return LAMBDA * loss1
def identity_loss(real_image, same_image, LAMBDA):
    loss = tf.reduce_mean(tf.abs(real_image - same_image))
    return LAMBDA * 0.5 * loss
monet_generator_optimizer = tf.keras.optimizers.Adam(0.002, decay = 0, beta_1=0.5)
photo_generator_optimizer = tf.keras.optimizers.Adam(0.002, beta_1=0.5)

monet_discriminator_optimizer = tf.keras.optimizers.Adam(0.002, beta_1=0.5)
photo_discriminator_optimizer = tf.keras.optimizers.Adam(0.002, beta_1=0.5)
monet_generator = Generator()
photo_generator = Generator()

monet_discriminator = Discriminator()
photo_discriminator = Discriminator()
cycle_gan_model = CycleGan(
    monet_generator, photo_generator, monet_discriminator, photo_discriminator
)

cycle_gan_model.compile(
    m_gen_optimizer = monet_generator_optimizer,
    p_gen_optimizer = photo_generator_optimizer,
    m_disc_optimizer = monet_discriminator_optimizer,
    p_disc_optimizer = photo_discriminator_optimizer,
    gen_loss_fn = generator_loss,
    disc_loss_fn = discriminator_loss,
    cycle_loss_fn = calc_cycle_loss,
    identity_loss_fn = identity_loss
)
monet_generator(np.zeros((1,256,256,3)))
cycle_gan_model.load_weights('kaggle results/weights.h5')
def normalize_image(image):
    image = (tf.cast(image, tf.float32) / 127.5) - 1
    return image
def predict(filename, show_big = False, path_to_photos = 'gan-getting-started/photo_jpg/'):
    photo_ = cv2.imread(os.path.join(path_to_photos,filename))
    photo_ = cv2.cvtColor(photo_, cv2.COLOR_BGR2RGB)
    photo = normalize_image(photo_)
    result = monet_generator(tf.expand_dims(photo, axis=0), training=False)
    result = (result * 127.5 + 127.5).numpy().astype(np.uint8)
    im = PIL.Image.fromarray(result[0])
    if show_big:
        im.show()
        PIL.Image.fromarray(photo_).show()
    plt.subplot(1,2,1)

```

```
plt.imshow(photo_, vmin=0, vmax=255)

plt.subplot(1,2,2)
plt.imshow(im, vmin=0, vmax=255)

plt.show()
predict('3e31646935.jpg', True)
```

ДОДАТОК Б ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

Система створення художньої стилізації зображення на основі генеративних змагальних мереж

Виконав:
студент 4 курсу
групи КА-71
Правосуд Руслан Олександрович

Дипломний керівник:
доцент, к.т.н.
Дідковська Марина Віталіївна

Рисунок Б.1

Генеративні змагальні мережі

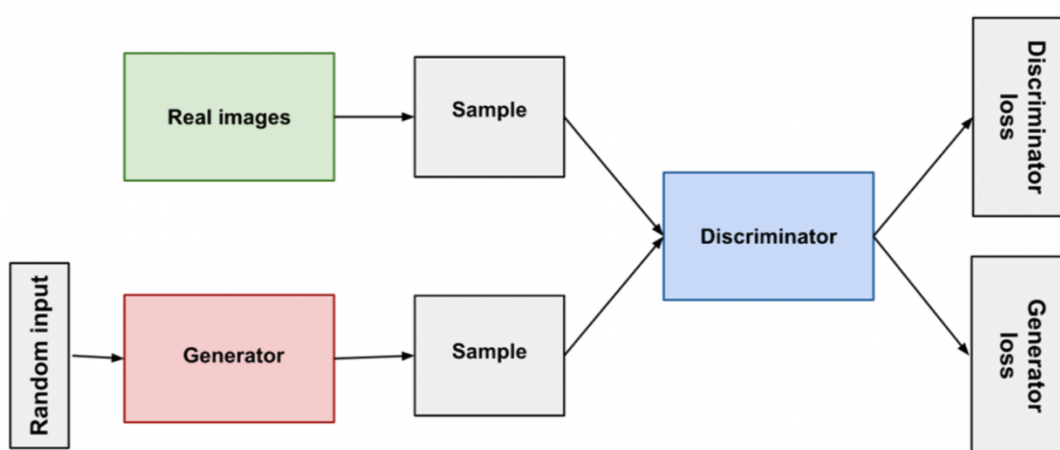


Рисунок Б.2

Функція втрат генеративної змагальної мережі

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

Де G та D – диференційовані функції

G – відображає вхідні дані у бажаний набір даних

$D(\mathbf{x})$ – функція, що визначає імовірність, що $\mathbf{x}$ належить реальному набору даних, а не згенерована генеративною частиною моделі

$p_{\text{data}}(\mathbf{x})$ – набір реальних даних, які навчається генерувати
генеративна модель

$p_{\mathbf{z}}(\mathbf{z})$ – дані, що подаються на вхід генеративній моделі

Рисунок Б.3

Циклічні генеративні змагальні мережі

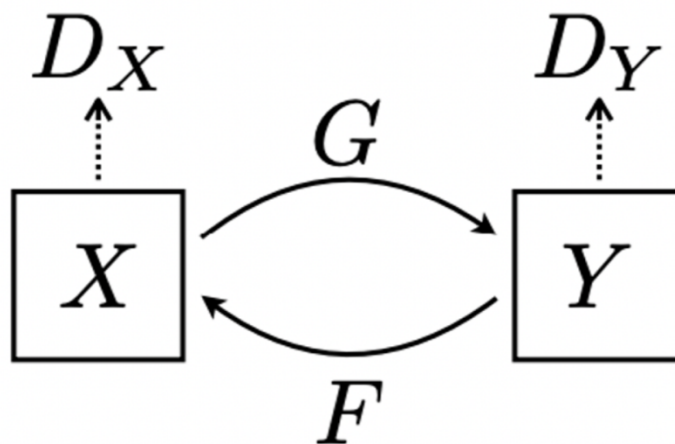


Рисунок Б.4

Умова консистентності циклічного відображення

$$\mathcal{L}_{\text{cyc}}(G, F) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{\text{data}}(y)} [\|G(F(y)) - y\|_1].$$

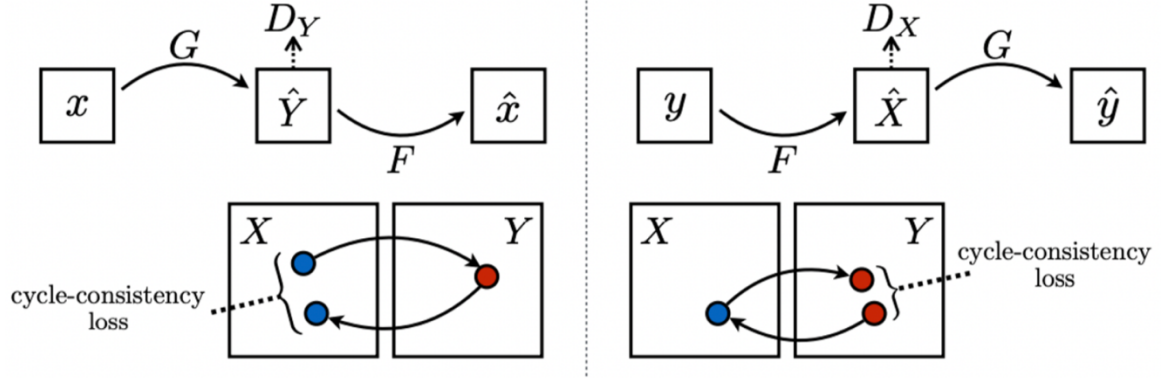


Рисунок Б.5

Функція втрат циклічної генеративної змагальної мережі

$$\begin{aligned} \mathcal{L}(G, F, D_X, D_Y) = & \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ & + \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ & + \lambda \mathcal{L}_{\text{cyc}}(G, F), \end{aligned}$$

$$\begin{aligned} \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = & \mathbb{E}_{y \sim p_{\text{data}}(y)} [\log D_Y(y)] \\ & + \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log(1 - D_Y(G(x)))] \end{aligned}$$

$$G^*, F^* = \arg \min_{G, F} \max_{D_x, D_Y} \mathcal{L}(G, F, D_X, D_Y)$$

Рисунок Б.6

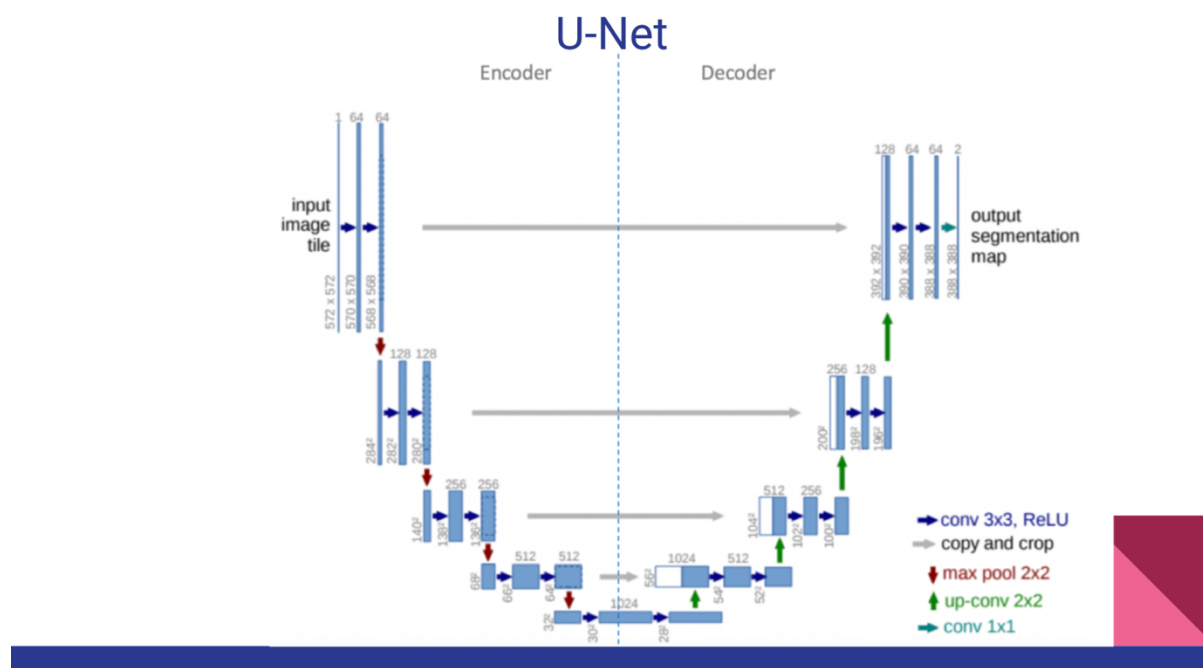


Рисунок Б.7

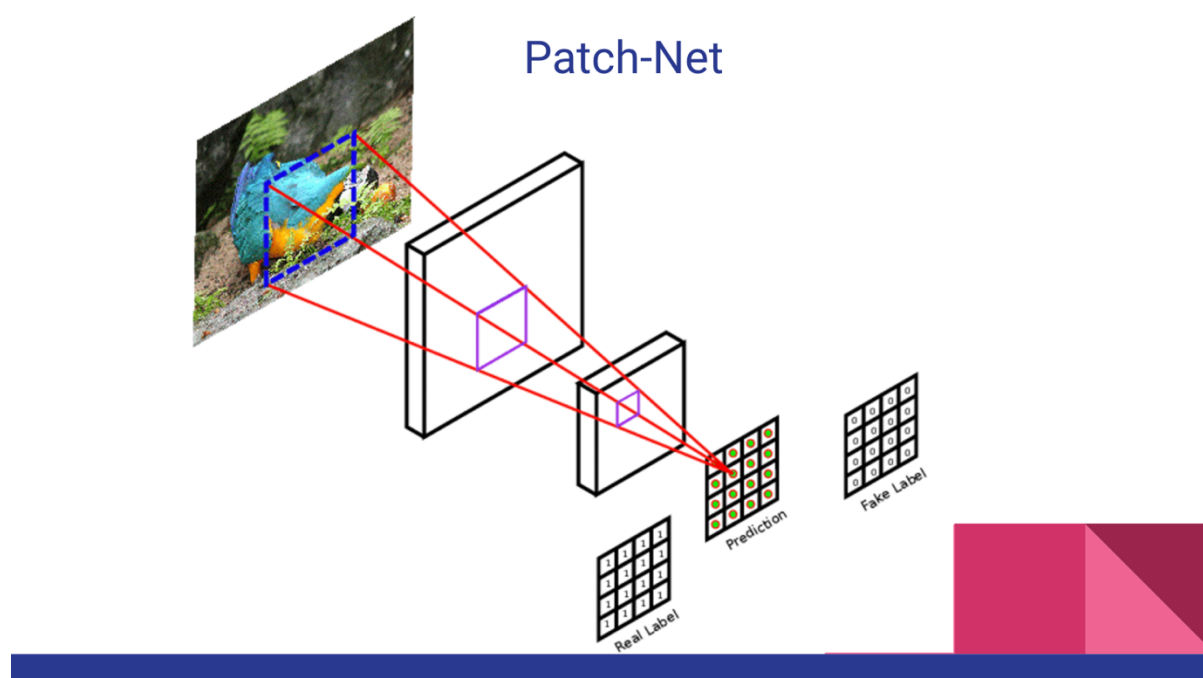


Рисунок Б.8

Використані бібліотеки



Рисунок Б.9

Результати роботи

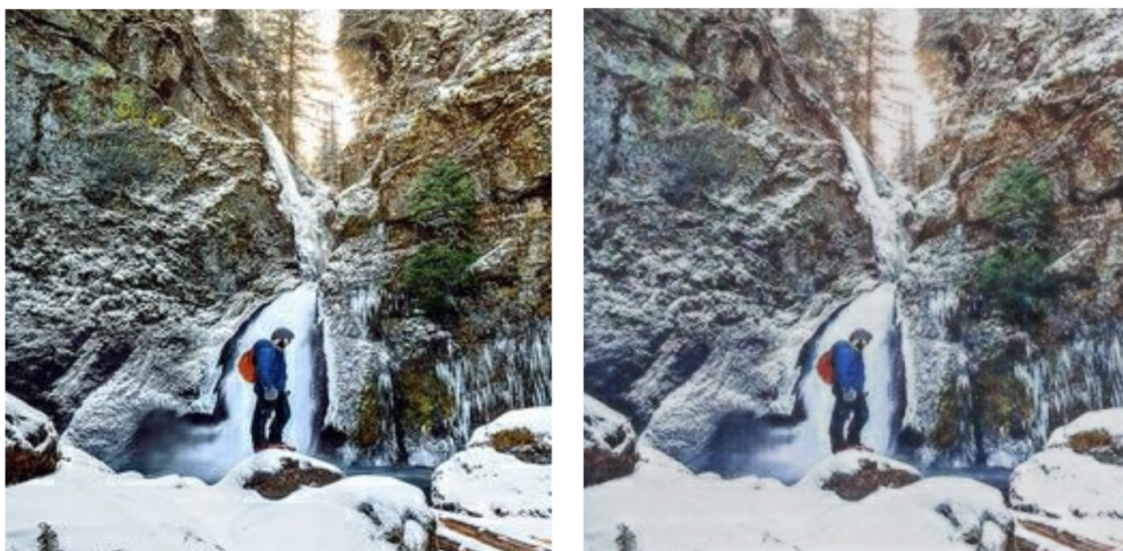


Рисунок Б.10

Результати роботи



Рисунок Б.11

Дякую за увагу!



Рисунок Б.12