Міністерство освіти і науки України
Львівський національний університет ім. Івана Франка
Факультет електроніки та комп’ютерних технологій
Кафедра системного проектування

КУРСОВА РОБОТА

на тему: “ Семантична сегментація зображень”

Виконав:
студен групи ФеІм 12с
Богатирьов О. О.
Науковий керівник:
доцент Демків Л. С.

Львів - 2023р.

АНОТАЦІЯ

У даній курсовій роботі розглядається задача семантичної сегментації зображень із застосуванням архітектури U-Net. Проводиться огляд літератури, в якому досліджуються вже існуючі підходи до мультикласової сегментації зображень. Зокрема, детально розглядаються статті "Fully Convolutional Networks for Semantic Segmentation", "U-Net: Convolutional Networks for Biomedical Image Segmentation" та "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". На основі огляду літератури обрано підхід U-Net та розглядається його застосування в задачах сегментації зображень. Крім того, описано архітектуру UNET, реалізовану в рамках даної роботи.

ЗМІСТ

ВСТУП 5

1. СЕМАНТИЧНА СЕГМЕНТАЦІЯ 7

1.1. Визначення семантичної сегментації. 7

1.2. Відмінності семантичної сегментації від інших задач комп'ютерного зору. 8

2. ОГЛЯД ЛІТЕРАТУРИ 10

2.1. Дослідження, що вже виконані у галузі мультикласової сегментації зображень. 10

2.1.1. "Fully Convolutional Networks for Semantic Segmentation" 10

2.1.2. "U-Net: Convolutional Networks for Biomedical Image Segmentation" 11

2.1.3. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" 12

2.2. Підхід U-Net та його застосування в задачах сегментації. 12

3. АРХІТЕКТУРА U-Net 14

3.1. Опис моєї реалізації U-Net. 14

4. ЗБІР ТА ПОПЕРЕДНЯ ОБРОБКА ДАНИХ 16

4.1. Вибір набору даних для навчання та тестування. 16

4.2. Попередня обробка даних 16

5. НАВЧАННЯ МОДЕЛІ 18

5.1. Завантаження зображень. 18

5.2. Розрахунок ваг класів. 18

5.3. Навчання U-Net нейромережі з використанням тренувального набору даних. 19
6. АНАЛІЗ РЕЗУЛЬТАТІВ 23
6.1. Метрики для оцінки точності. 23
6.2. Результати для навчання без ваг класів. 24
6.3. Результати навчання з вагами. 26
ВИСНОВКИ 34
ЛІТЕРАТУРА 35
ДОДАТКИ 36

ВСТУП

У сучасному світі обробка зображень відіграє важливу роль у багатьох галузях, таких як медицина, автономні транспортні засоби, робототехніка тощо. Одним із ключових завдань обробки зображень є семантична сегментація, яка полягає в призначенні кожному пікселю зображення відповідного класу або категорії. Досягнення точності та ефективності у семантичній сегментації є важливим завданням для подальшого розвитку цієї галузі.

У даному курсовому проекті досліджується підхід U-Net для семантичної сегментації зображень. Метою роботи є розробка та тренування моделі з використанням архітектури U-Net та аналіз результатів її навчання. Для досягнення цієї мети, виконано огляд літератури, в якому вивчено раніше проведені дослідження у галузі мультикласової сегментації зображень. Особлива увага приділена статтям "Fully Convolutional Networks for Semantic Segmentation", "U-Net: Convolutional Networks for Biomedical Image Segmentation" та "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift".

У роботі також описано архітектуру U-Net, яка використовується у даному проекті. Детально описано реалізацію цієї архітектури та зазначено особливості її застосування у задачах сегментації зображень. Також розглянуто процес збору та попередньої обробки даних перед навчанням моделі.

У подальших розділах описано процес навчання моделі з використанням тренувального набору даних, а також аналізуються отримані результати. Проводиться оцінка точності моделі за допомогою певних метрик та порівняння результатів навчання з використанням ваг класів та без них.

Загальний висновок, отриманий у цій роботі, дає можливість зробити висновки про ефективність підходу U-Net у семантичній сегментації зображень та про можливі напрямки подальших досліджень у цій галузі.

1. СЕМАНТИЧНА СЕГМЕНТАЦІЯ

1.1. Визначення семантичної сегментації.

Семантична сегментація - це задача комп'ютерного зору, яка має на меті призначити кожному пікселю зображення мітку або класову категорію згідно його семантичного значення. У простих словах, це процес, за допомогою якого комп'ютер класифікує кожен піксель на зображенні відповідно до конкретної семантичної категорії, такої як людина, автомобіль, дерево тощо. Задача полягає в тому, щоб точно визначити межі та форму кожного об'єкта на зображенні.

Одним з відмінних аспектів семантичної сегментації є те, що вона надає докладну інформацію про положення та форму об'єктів на зображенні. У порівнянні з іншими задачами комп'ютерного зору, такими як класифікація чи об'єктне виявлення, семантична сегментація забезпечує піксельну точність, даючи змогу точно визначати кожен об'єкт окремо та відрізняти його від фону або інших об'єктів на зображенні.

Оскільки семантична сегментація використовується в багатьох сферах, таких як медицина, автомобільна промисловість, аналіз зображень та багато інших, вона має велике значення для розв'язання реальних завдань. Наприклад, вона може бути використана для автоматичного визначення ракових клітин на медичних зображеннях, для розпізнавання об'єктів на дорогах у системах безпеки автомобілів або для аналізу супутникових зображень для визначення змін у вегетації чи забудові.

Розв'язання задачі семантичної сегментації вимагає використання складних алгоритмів та глибоких нейромереж, таких як U-Net, які можуть здійснювати ефективне та точне віднесення кожного пікселя до його семантичної категорії. Такі методи дають змогу отримати високу якість сегментації зображень і мають великий потенціал для рішення різноманітних завдань у сфері комп'ютерного зору.

1.2. Відмінності семантичної сегментації від інших задач комп'ютерного зору.

Окрім семантичної сегментації, у сфері комп'ютерного зору існує багато інших задач, які займаються аналізом зображень. Ось декілька таких задач:

1. Класифікація зображень: Задача класифікації зображень полягає в тому, щоб призначити зображенню одну з попередньо визначених категорій або класів. Вона не враховує просторових відношень між об'єктами на зображенні і не здатна виділяти окремі області зображення.
2. Локалізація об'єктів: Локалізація об'єктів полягає в тому, щоб виявити та відмітити присутність об'єктів на зображенні та вказати їхні місцезнаходження. Вона зазвичай пов'язана з відміткою меж об'єктів або розташуванням прямокутних рамок навколо об'єктів.
3. Детекція об'єктів: Детекція об'єктів схожа на локалізацію, але вона також визначає кількість та класифікує об'єкти на зображенні. Завдання полягає в знаходженні об'єктів та визначенні їхніх меж та класів без необхідності точного виділення всіх деталей внутрішньої структури.
4. Визначення ключових точок: Ця задача полягає в знаходженні та локалізації ключових точок або особливостей на зображенні, які можуть бути використані для подальшого аналізу або реконструкції.
5. Відстеження об'єктів: Відстеження об'єктів передбачає визначення руху та місцезнаходження об'єктів на послідовних зображеннях. Вона часто використовується в системах відеоспостереження та аналізу руху.

Кожна з цих задач має свої унікальні особливості та використовує різні підходи та методи для аналізу зображень. Семантична сегментація зосереджена на виділенні та класифікації окремих областей зображення, що робить її особливо корисною для багатьох областей, де інші методи комп’ютерного зору є недостатніми, включаючи медичну діагностику, автономні автомобілі, аналіз земельних даних та багато інших.

Також слід зазначити наступні особливості семантичної сегментації:

1. Докладніша інформація про об'єкти: У семантичній сегментації кожен піксель на зображенні призначається конкретному класу або об'єкту. Це дає змогу отримати більш детальну інформацію про положення та форму кожного об'єкту на зображенні, що є важливим для багатьох додатків, наприклад, в медицині чи автомобільній промисловості.
2. Піксельна точність: У семантичній сегментації кожен піксель має свою класифікацію, що дає змогу здійснити точне виділення меж об'єктів на зображенні. Це відрізняється від задачі класифікації, де визначається тільки загальна категорія зображення, а не деталізована інформація про окремі об'єкти.
3. Заповнення областей: У семантичній сегментації кожна область на зображенні повністю заповнена одним класом. Це відрізняється від задачі детекції об'єктів, де потрібно виявити лише присутність об'єктів на зображенні без деталізації їх меж і форм.
4. Збереження контексту: У семантичній сегментації зберігається контекстуальна інформація про зображення, оскільки кожен піксель враховується у контексті його оточення. Це дає змогу враховувати сусідні пікселі та їхні взаємозв'язки при класифікації, що може покращ

2. ОГЛЯД ЛІТЕРАТУРИ

2.1. Дослідження, що вже виконані у галузі мультикласової сегментації зображень.

2.1.1. "Fully Convolutional Networks for Semantic Segmentation"

Дослідження "Fully Convolutional Networks for Semantic Segmentation", яке було опубліковане J. Long, E. Shelhamer та T. Darrell у рамках конференції IEEE Conference on Computer Vision and Pattern Recognition (CVPR) у 2015 році, презентує інноваційний підхід до задачі семантичної сегментації зображень з використанням повністю згорткових мереж (Fully Convolutional Networks - FCN).

Автори дослідження зрозуміли, що традиційні згорткові нейромережі, які використовуються для класифікації зображень, не можуть передбачати піксельну сегментацію, тобто присвоїти клас кожному пікселю зображення. Тому вони розробили архітектуру FCN, яка здатна вирішувати задачу семантичної сегментації, зберігаючи просторову інформацію та здатність прогнозувати клас кожного пікселя.

Основною ідеєю дослідження є заміна повністю зв'язаних шарів (fully connected layers) у згорткових нейромережах на звичайні згорткові шари. Це дає змогу зберігати просторові розміри вихідних зображень і отримувати карту сегментації з таким же розміром.

Архітектура FCN складається з енкодер-частини та декодер-частини. Енкодер використовує попередньо навчену згорткову нейромережу, яка перетворює зображення на векторні ознаки. Декодер використовує звичайні згорткові шари для розширення вихідних ознак та отримання карти сегментації. Для поліпшення точності сегментації, в декодері використовуються додаткові з'єднувальні шляхи, які передають інформацію з енкодера.

Дослідження також використовує метод зведення розмірності (dimensionality reduction), який дає змогу зменшити розмірність вихідних ознак для поліпшення обчислювальної ефективності.

Результати дослідження показали, що FCN має високу точність в задачі семантичної сегментації зображень і може ефективно використовуватись у різних сферах, включаючи комп'ютерний зір, медичне зображення та автономну навігацію.

2.1.2. "U-Net: Convolutional Networks for Biomedical Image Segmentation"

У дослідженні "U-Net: Convolutional Networks for Biomedical Image Segmentation" авторами O. Ronneberger, P. Fischer та T. Brox, опублікованому в рамках конференції Medical Image Computing and Computer-Assisted Intervention (MICCAI) у 2015 році, представлено архітектуру мережі U-Net, спеціально розроблену для задач сегментації біомедичних зображень.

Автори вперше застосували цю архітектуру для вирішення завдань сегментації біомедичних зображень, таких як сегментація клітин, органів або пухлин. Автори спираються на раніше описану FCN архітуктуру [1], і в статті описано особливості використання реалізації U-Net.

Архітектура U-Net вдало використовується для сегментації біомедичних зображень, оскільки вона може добре впоратись з об'єктами різних розмірів та форм, забезпечує точну локалізацію об'єктів та має низьку кількість параметрів для ефективного тренування.

Дослідження впровадило новий стандарт у галузі сегментації біомедичних зображень та стало основою для подальшого розвитку та вдосконалення архітектур нейромереж для сегментації зображень.

2.1.3. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"

Дослідження "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" було написане Сергеєм Іоффе та Крістіаном Зегеді. Ця робота була представлена на конференції машинного навчання (International Conference on Machine Learning) у 2015 році.

У цьому дослідженні автори висвітлюють проблему внутрішнього коваріаційного зсуву (internal covariate shift), яка виникає під час навчання глибоких нейромереж. Вони розглядають цю проблему як один із факторів, що уповільнюють процес навчання та знижують ефективність моделей глибокого навчання.

У дослідженні автори пропонують методологію пакетної нормалізації (batch normalization), яка дає змогу знизити вплив внутрішнього коваріаційного зсуву. Цей метод включає нормалізацію активаційних значень в кожному міні-пакеті даних шляхом центрування та масштабування. Пакетна нормалізація дає змогу забезпечити стабільність розподілу активацій, що поліпшує швидкість навчання та здатність моделі до узагальнення.

Результати дослідження показують, що використання пакетної нормалізації може значно прискорити навчання глибоких нейромереж та покращити їхню ефективність. Автори проводять експерименти на різних архітектурах нейромереж та демонструють переваги пакетної нормалізації в порівнянні з іншими методами нормалізації.

Пакетна нормалізація в подальшому використовується як покращення U-Net архітектури багатьма дослідниками.

2.2. Підхід U-Net та його застосування в задачах сегментації.

З моменту публікації оригінальної статті з описом архітектури Підхід U-Net є одним з найпопулярніших підходів у сегментації зображень і використовується для досягнення високої точності в завданнях

мультикласової сегментації. U-Net був спеціально розроблений для сегментації біомедичних зображень, але його застосування також розширилося на інші галузі, включаючи комп'ютерний зір та обробку зображень.

Архітектура U-Net складається з двох основних компонентів: енкодеру та декодеру. Енкодер має структуру згорткової нейромережі, яка послідовно зменшує розмірність зображення та зберігає контекстну інформацію про об'єкти на різних рівнях абстракції. Завдяки цьому, енкодер може виділити різні особливості зображення на різних шарах.

Декодер, навпаки, використовує транспоновані згорткові шари для поступового збільшення розмірності зображення та відновлення деталей. Декодер поєднує вихідні ознаки з енкодера з кожним шаром, використовуючи операцію конкатенації або з'єднувальні шляхи (skip connections). Це дає змогу передавати більш детальну інформацію про місцезнаходження об'єктів на різних рівнях.

Застосування U-Net в задачах сегментації дає змогу досягнути точності визначення меж об'єктів, зберігаючи контекстуальну інформацію та враховуючи глобальний контекст зображення. Цей підхід добре працює з різними типами даних, включаючи медичні зображення [2], супутникові знімки [4], зображення урбаністичного середовища [5] тощо.

U-Net є гнучким і розширюваним підходом, і може бути модифікований та адаптований відповідно до конкретної задачі сегментації. Це дає змогу дослідникам та розробникам впроваджувати різні покращення та оптимізації, щоб покращити результати сегментації зображень для конкретних вимог та доменів застосування.

3. АРХІТЕКТУРА U-Net

3.1. Опис моєї реалізації U-Net.

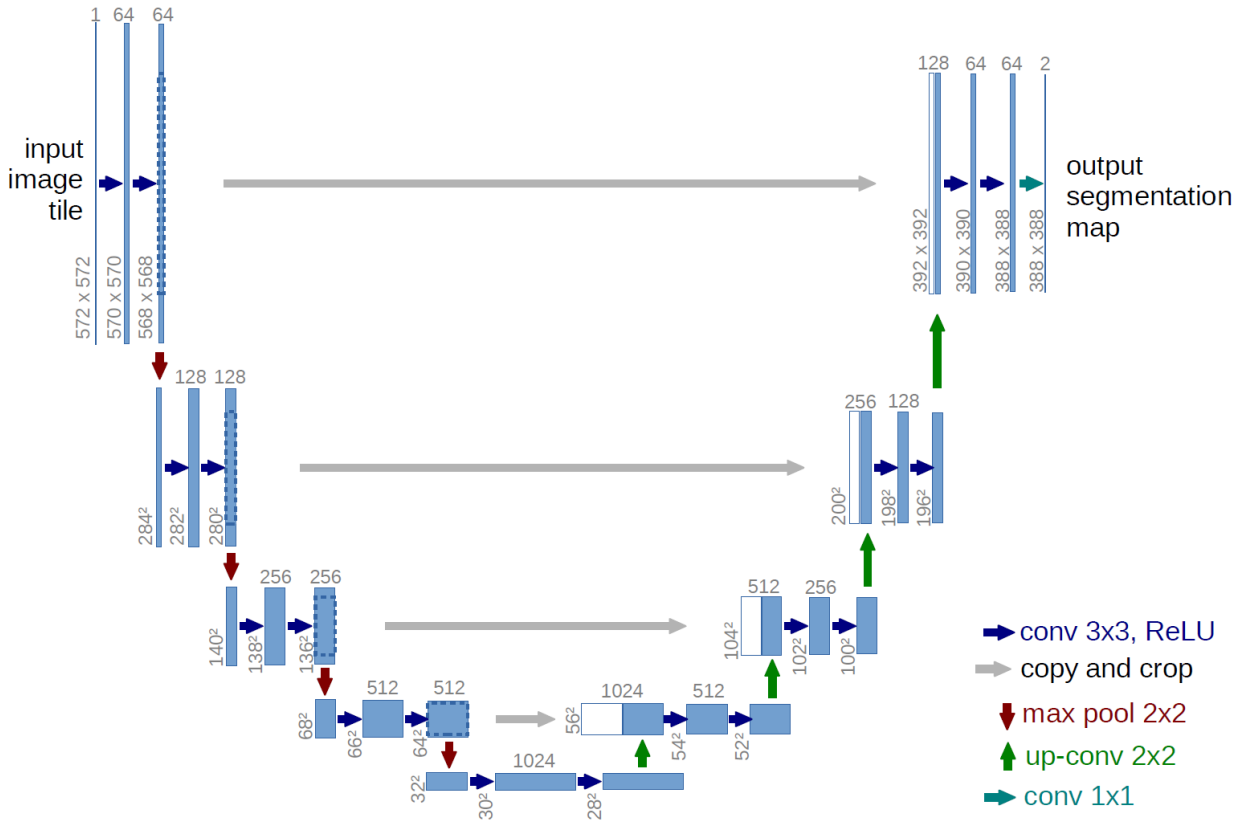

Рис.1. Оригінальна схема архітектури U-Net [2]

Оригінальна архітектура U-Net приведена на Рис.1, з моменту публікації опубліковано багато робіт які адаптують і покращують запропоновану архітектуру, так, наприклад, базуючись на загальних дослідженнях впливу пакетної нормалізації на якість навчання нейромереж [3] є роботи де ця методика показує свою ефективність і при застосуванні у U-Net [6], то ж модель використана в цій роботі теж буде використовувати цей метод.

Для розробки моделі і її подальшого навчання було використано python пакет PyTorch, який містить реалізації базових абстракцій для різних типів машинного навчання: n-вимірні тензори, тензорні операції, згорткові шари та інші. [7]

Повний код моделі наведено в Додатку 1, для простоти описання моделі вводиться клас DoubleConv (скорочення від “подвійна згортка”), який описує

2 згортки і ReLu з оригінальної структури(Рис.1.), а також додає пакетну нормалізацію, переваги якої було згадно раніше, між ними.

Повна U-Net модель знаходиться у класі U-Net (Додаток 1), в цьому класі спочатку в методі __init__ створюються базові шари: спуск, підйом, шар для об'єднання, "шийка пляшки" (в англомовний літературі - bottleneck) і фінальний згортковий шар.

В методі forward класу U-Net, описується покрокова передача даних між усіма шарами. Спочатку вхідний тензор пропускається через усі згорткові шари які збирають властивості на різних рівнях (ліва частина на Рисунку 1) і на кожній ітерації зберігається результат шару, щоб в подальшому бути використаним у підйомі. Далі застосовується згортка "шийка пляшки", яка на Рисунку 1 знаходиться в самому низу. Далі починається збільшення тензору, але перед кожною подвійною згорткою поточний тензор об'єднується з результатами паралельної подвійної згортки під час зменшення (сірі горизонтальні стрілки на Рисунку 1). В кінці до тензора застосовується згортка 1x1, яка зменшує кількість класів до умов задачі і використовується функція softmax, яка перетворює вихідні дані нейромережі у ймовірності класів:

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \quad (1)$$

Також важливою відмінністю моєї реалізації є вихідний тензор. який матиме більшу розмірність, яка буде відповідати кількості класів набору даних.

4. ЗБІР ТА ПОПЕРЕДНЯ ОБРОБКА ДАНИХ

4.1. Вибір набору даних для навчання та тестування.

Для навчання і тестування було обрано набор підводних фотографій (Додаток 2), Цей набір даних складається з 1525 фотографій для навчання і 110 для тренування. В наборі виділяється 8 класів об’єктів для класифікації, також не всі зображення містять всі класи і розподіл класів є доволі незбалансованим, що дає змогу дослідити вплив такого розподілу на навчання.

4.2. Попередня обробка даних.

Для попередньої обробки даних використовується бібліотека Albumentations, яка допомагає попередньо обробити зображення. Оскільки нейромережа буде опрацьовувати тренувальний і валідаційний набори даних. Код для трансформацій наведено в Додатку 3.

Тренувальні трансформації, визначені за допомогою train_transform, мають наступні етапи:

1. Resize: Змінює розмір зображення на висоту IMAGE_HEIGHT та ширину IMAGE_WIDTH.
2. HorizontalFlip: Застосовує горизонтальне відображення до зображення з ймовірністю 0.5 (50%).
3. VerticalFlip: Застосовує вертикальне відображення до зображення з ймовірністю 0.1 (10%).
4. Normalize: Нормалізує значення пікселів зображення, віднімаючи середнє значення [0.0, 0.0, 0.0] і ділячи на стандартне відхилення [1.0, 1.0, 1.0]. Також вказує, що максимальне значення пікселів у зображенні становить 255.0.
5. ToTensorV2: Конвертує зображення у формат тензору PyTorch.

Трансформації для валідаційного набору даних, визначені за допомогою val_transforms, мають наступні етапи:

1. Resize: Змінює розмір зображення на висоту IMAGE_HEIGHT та ширину IMAGE_WIDTH.
2. Normalize: Нормалізує значення пікселів зображення, віднімаючи середнє значення [0.0, 0.0, 0.0] і ділячи на стандартне відхилення [1.0, 1.0, 1.0]. Також вказує, що максимальне значення пікселів у зображенні становить 255.0.
3. ToTensorV2: Конвертує зображення у формат тензору PyTorch.

Перевагою використання цієї бібліотеки є те, що трансформації застосовуються як до оригінального зображення, так і до маски. Але кожна трансформація має різний вплив на оригінал і маску, так ця бібліотека має набір трансформацій для додавання шуму, зміни тону зображення та інші [8], але такі трансформації будуть застосовані тільки до оригінального зображення, оскільки перерахунок кольорів маски буде впливати на результати навчання, що неприпустимо, але змін розміру і положення пікселів оригінального зображення відповідно змінить маску.

Тренувальні зображення мають додаткові трансформації, у порівнянні з валідаційними. Так HorizontalFlip і VerticalFlip дають змогу збільшити розмір тренувального набору даних, оскільки досліджуваний домен дає змогу такі перетворення, але оскільки в наборі даних вже доволі великий обсяг даних, то застосовані лише наведені перетворення.

5. НАВЧАННЯ МОДЕЛІ

5.1. Завантаження зображень.

Для зчитування зображень під час навчання і валідації було використано базові класи PyTorch: Dataset і DataLoader. Dataset - це клас у бібліотеці PyTorch, який використовується для створення користувацьких наборів даних для навчання моделей глибокого навчання. Цей клас визначає інтерфейс, який дає змогу отримувати доступ до даних та їх міток у структурованому форматі, що сприяє зручності та ефективності роботи з ними.

Клас Dataset має два основних методи, які потрібно перевизначити при створенні власного набору даних:

- len(): Цей метод повертає загальну кількість елементів у наборі даних. Це дає змогу отримати інформацію про кількість прикладів даних, що мають бути використані для навчання моделі.
- getitem(index): Цей метод приймає індекс та повертає пару даних і міток для заданого індексу. Це дає змогу отримати доступ до конкретного прикладу даних у наборі.

Після створення класу Dataset (Додаток 4) можна використовувати його разом з класом DataLoader для зручного завантаження та обробки даних під час навчання моделі. DataLoader автоматично застосовує паралельне завантаження даних, перемішування, попередню обробку та розподіл даних на пакети, що спрощує процес навчання.

Власний клас Dataset може бути налаштований для роботи з різними типами даних, такими як зображення, текст або числові дані. Це дає змогу легко створювати набори даних для різних завдань і використовувати їх у моделях глибокого навчання.

5.2. Розрахунок ваг класів.

Оригінальна стаття вказує на важливість використання ваги класів при розрахунку похибки навчання [2]. Оскільки сам по собі набір даних не містив

значення ваг класів було розроблено підпрограму яка б допомогла з їх обрахунком.

Оскільки класи обраного набору даних не є збалансованими, то цей крок був необхідним. Концептуально вага дає змогу значно сильніше впливати неправильно поміченим пікселям класу, який представлено найменше.

Розрахунок ваг зводиться до банального підрахунку кількості пікселів кожного класу в усіх зображеннях набору даних з подальшою нормалізацією і оберненням за рахунок ділення 1 на нормалізовані кількості. Код цієї підпрограми наведено у Додатку 5.

Результатом роботи підпрограми є значенні наведені в Таблиці 1.

Номер класу	0	1	2	3	4	5	6	7
Кількість пікселів	189688979	12313599	11940788	37353994	2917692	186100654	39309599	78415339
Нормалізована кількість	1	0.0649	0.0629	0.1969	0.0154	0.9811	0.2072	0.4134
Вага	1	15.4048	15.8858	5.0781	65.0134	1.0193	4.8255	2.4190

Таблиця 1. Результати роботи підпрограми для розрахунку ваг.

З ваг наведених у Таблиці 1 стає зрозуміло що набір даних є дуже незбалансованим, так відношення між 0 класом і 4 дорівнює 1:65.0134, якщо виключити ваги з розрахунку похибки ми ризикуємо “застрягти” під час навчання у локальному мінімумі. Оскільки не класифікувати 4 клас взагалі буде набагато логічніше з точки зору зменшення похибки.

5.3. Навчання U-Net нейромережі з використанням тренувального набору даних.

Цикл навчання наведено у Додатку 3 і деякі функції для зручності винесено у окремий файл, вміст якого наведено у Додатку 4. До запуску циклу проводиться створення необхідних об’єктів. Спочатку описуються

перетворення для зображень, створюється модель U-Net, функція втрат CrossEntropyLoss з вагами і оптимізатор.

Оптимізатор використовує алгоритм Adam. Adam (Adaptive Moment Estimation) є адаптивним методом оптимізації градієнтного спуску, який використовується для налаштування параметрів моделей глибокого навчання. Він поєднує в собі ідеї з двох інших алгоритмів оптимізації - AdaGrad і RMSprop.

Основна ідея Adam полягає в тому, щоб адаптивно налаштувати швидкість навчання для кожного параметра в залежності від історії градієнтів та їх квадратів. Він використовує два експоненційно зважені середні: перше експоненційно зважене середнє (exponentially weighted average, EMA) градієнтів, і друге експоненційно зважене середнє квадратів градієнтів.

Конкретні дії, які виконує Adam під час оптимізації, включають:

1. Обчислення градієнтів моделі за допомогою алгоритму зворотнього поширення похибки.
2. Ініціалізація змінних, що тримають історію градієнтів (перше експоненційно зважене середнє) та квадратів градієнтів (друге експоненційно зважене середнє).
3. Оновлення змінних, обчисленням EMA градієнтів та квадратів градієнтів на кожному кроці.
4. Обчислення коригуючих коефіцієнтів для збереження сталості швидкості навчання на різних етапах оптимізації.
5. Оновлення параметрів моделі з урахуванням налаштованої швидкості навчання та квадратного кореня з EMA градієнтів.
6. Повторення кроків 1-5 для кожної ітерації навчання.

Далі завантажується модель, яка попередньо була збережена. Це дає змогу продовжити навчання з попередніх експериментів. Ще додатково створюється об’єкт GradScaler.

У фреймворку PyTorch для навчання моделей на графічних процесорах (GPU) існує пакет torch.cuda.amp, який надає підтримку автоматичного зміщення градієнтів (Gradient Scaling). Градієнтне зміщення є технікою, яка допомагає уникнути проблем з чисельною стійкістю (наприклад, великого випливу або занурення значень) під час навчання моделі.

Використання GradScaler особливо корисне в глибоких нейромережах, де можуть виникати чисельні проблеми через великі значення градієнтів. GradScaler автоматично масштабує значення градієнтів під час їх обчислення, забезпечуючи чисельну стійкість і запобігаючи великому випливу або зануренню значень.

Далі для кожної епохи ми тренуємо модель на повному тренувальному наборі даних і зберігаємо стан моделі і результати обробки валідаційного набору у вигляді зображень.

Ітерацію тренування винесено у окрему функцію, вона має наступну логіку:

Для кожного пакета, який отримується від loader:

1. Переміщує дані та мітки на робочий пристрій.
2. Застосовує перетворення target_transform(Додаток 6) до міток targets_raw і також переміщує їх на робочий пристрі. Перетворення дає змогу отримати з зчитаного формату зображення у формі (ширина, висота, колір) у форму виходу нейромережі (клас, ширина, висота).
3. Виконує прямий прохід моделі на вхідних даних. Застосовується автоматичне масштабування градієнтів з контекстним менеджером torch.cuda.amp.autocast(), що дає змогу використовувати поліпшену точність обчислень, наприклад, при використанні змінних з плаваючою комою в навчанні на GPU.
4. Обчислює втрату (помилку) між передбаченими значеннями та цільовими значеннями за допомогою функції втрат.

5. Обнуляє градієнти оптимізатора optimizer перед здійсненням зворотнього поширення.
6. Здійснює зворотнє поширення градієнтів за допомогою функції backward() та масштабує градієнти за допомогою об'єкта scaler.
7. Виконує крок оптимізатора optimizer для оновлення параметрів моделі згідно з обчисленими градієнтами.
8. Оновлює стан масштабування градієнтів scaler.
9. Оновлює прогресну панель tqdm loop, що відображає втрату для поточного пакету даних.

6. АНАЛІЗ РЕЗУЛЬТАТІВ

6.1. Метрики для оцінки точності.

Для аналізу результатів навчання було використано наступні метрики: піксельна точність, IoU для кожного класу і dice score. Кожна з цих метрик має свою власну сутність і особливості, які розглянемо детальніше.

1. Піксельна точність (Pixel Accuracy):

   Піксельна точність - це проста метрика, яка вимірює відсоток пікселів, що були класифіковані правильно відносно загальної кількості пікселів у зображенні. Вона вказує на загальну точність сегментації. Ця метрика особливо корисна, коли класи мають незбалансовану розподіленість, оскільки вона враховує кількість правильно класифікованих пікселів незалежно від класу. Однак піксельна точність не враховує просторову структуру сегментації і може бути вразлива до шуму або незначних зсувів у сегментаційному результаті.

2. Intersection over Union (IoU):

   Intersection over Union вимірює ступінь перекриття між прогнозованим сегментом і правильною міткою. Він обчислюється як відношення площі об'єднання двох областей до їх перетину. IoU надає інформацію про точність та повноту сегментації для кожного класу окремо. Використання IoU для кожного класу дає змогу детальніше оцінити ефективність моделі в розрізі окремих класів. Ця метрика особливо корисна для вимірювання якості сегментації, коли об'єкти різних класів можуть мати перекриття або розташовуватися поруч. Однак IoU також може бути чутливим до незначних зміщень у прогнозованих сегментах та не враховує чутливість до форми сегментів.

3. Dice Score:

   Dice Score, також відомий як F1 Dice Score, вимірює ступінь схожості між прогнозованим сегментом і правильною міткою. Він обчислюється як подвоєне співвідношення площі перетину до загальної площі двох

областей. Dice Score також надає інформацію про точність та повноту сегментації для кожного класу окремо. Ця метрика часто використовується в задачах сегментації, оскільки вона чутлива до форми сегментів і перетинів між ними. Однак Dice Score може бути нечутливим до незначних зміщень у прогнозованих сегментах і також може бути впливованим розміром об'єктів.

У випадку мультикласової сегментації зображень, використання цих метрик дає змогу оцінити якість сегментації для кожного класу окремо, а також загальну точність моделі. Вони допомагають виявити сильні та слабкі сторони моделі у вирішенні завдання сегментації. Однак важливо враховувати їхні обмеження. Всі ці метрики не враховують контекст сегментації та просторову структуру зображення. Вони можуть бути вразливі до шуму, зміщень та незначних змін у формі сегментів. Також слід пам'ятати, що результати метрик можуть бути спотворені, якщо використовується неправильна мітка або некоректно вказані контури сегментів.

Узагальнюючи, піксельна точність, IoU та Dice Score є корисними метриками для оцінки якості мультикласової сегментації зображень. Використання цих метрик дає змогу отримати розгорнуту інформацію про ефективність моделі в розрізі окремих класів та загальної точності. Однак їхні обмеження варто враховувати при інтерпретації результатів.

6.2. Результати для навчання без ваг класів.

Необхідність використання ваг класів описано у 5.2, але для демонстрації різниці метрик точності і проблематики сегментації наборів даних з незбалансованими класами було проведено наступний експеримент.

Для кращої візуалізації результатів було створено скрипт наведений у Додатку 7, а кодування класів кольорами – у Додатку 8.

Після навчання моделі без ваг класів було досягнуто піксельної точності 90.41. Але також були отримані наступні значення IoU та Dice Score:

Номер класу	0	1	2	3	4	5	6	7
IoU	0.757	0	0	0	0	0.401	0.187	0.269
Dice score	0.861	0	0	0	0	0.565	0.308	0.421

Таблиця 2. Результати незваженого навчання.

Як видно з таблиці 2, в певний момент модель обрала лише 4 класи для класифікації, а інші – ігнорувала. Також це добре видно з результатів роботи скрипту з Додатку 7.

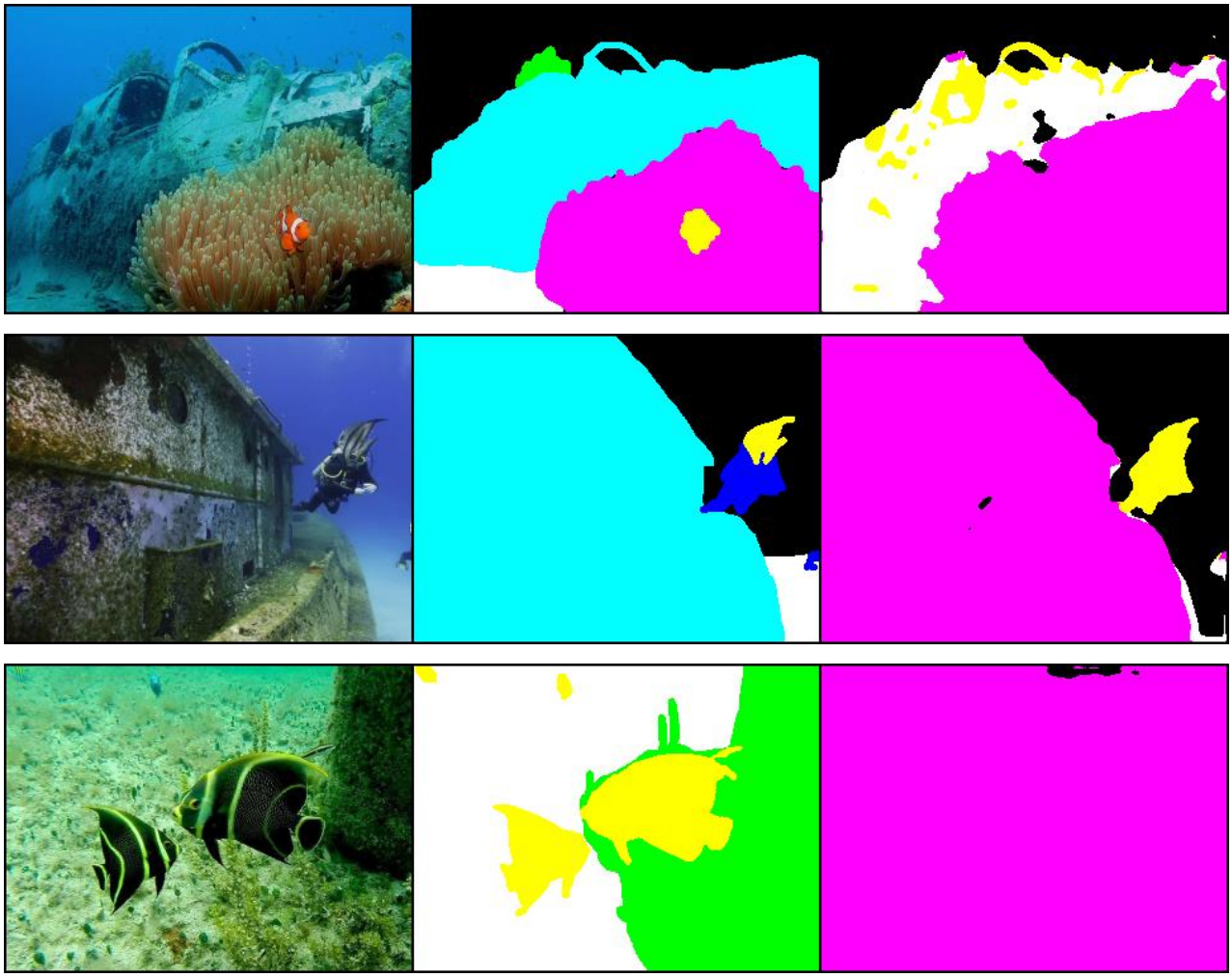

Рис.2.1. Візуалізація роботи навчання незваженої моделі.

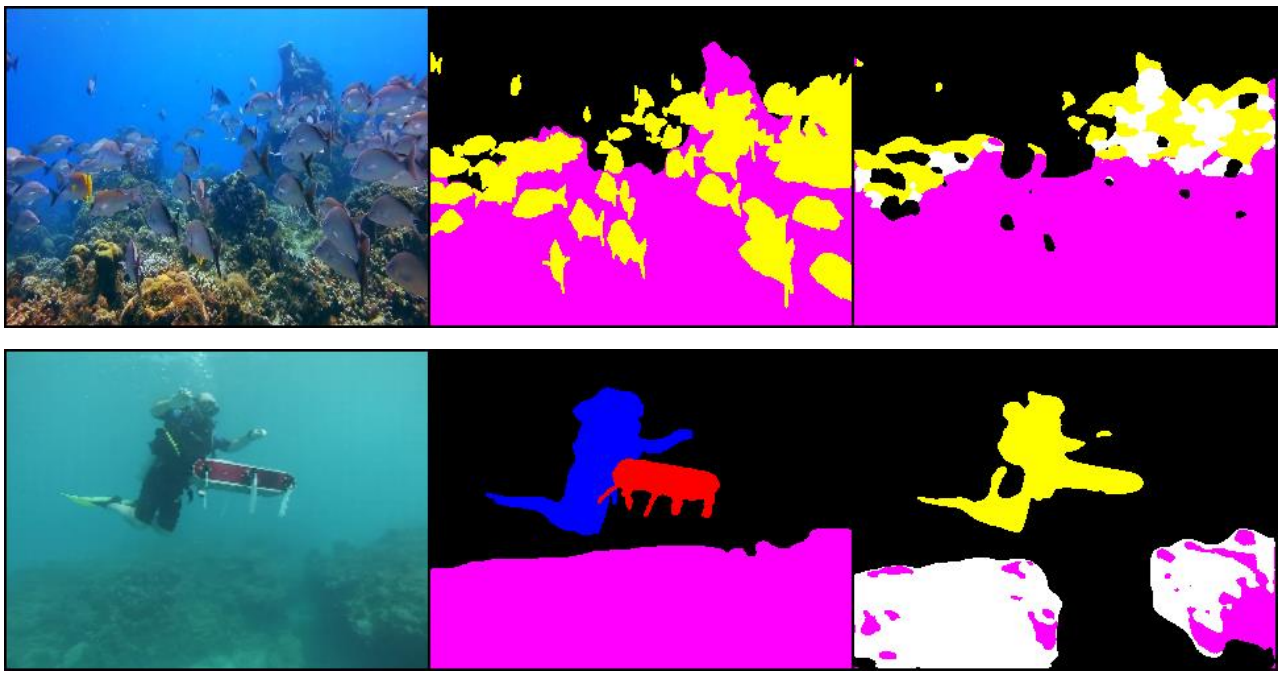

Рис.2.2. Візуалізація роботи навчання незваженої моделі.

Як видно з Рис.2.1. та Рис.2.2. та Таблиці 2 модель повністю ігнорує 4 класи, а саме 1-4, і повертаючись до Таблиці 1 це пояснюється кількісним представлення цих класів в навчальному наборі даних. Тут модель знайшла найкращі вихідні значення, які скоріше за все будуть відповідати дійсності, тобто локальний мінімум. Це добре ілюструє необхідність використання ваги класів, а також проблематику використання піксельної точності для оцінки прогресу моделі для мультикласової сегментації.

6.3. Результати навчання з вагами.

Для наступного експерименту було використано підхід описаний у 5.2. Так результати першої ітерації мають наступний вигляд:

Номер класу	0	1	2	3	4	5	6	7
IoU	0.489	0.080	0.111	0.101	0.026	0.255	0.092	0.312
Dice score	0.655	0.145	0.192	0.161	0.050	0.404	0.168	0.475

Таблиця 3. Перше покоління зваженого навчання.

Піксельна точність першої ітерації становила 85.45, що для інших задач комп’ютерного зору є відносно високим значенням.

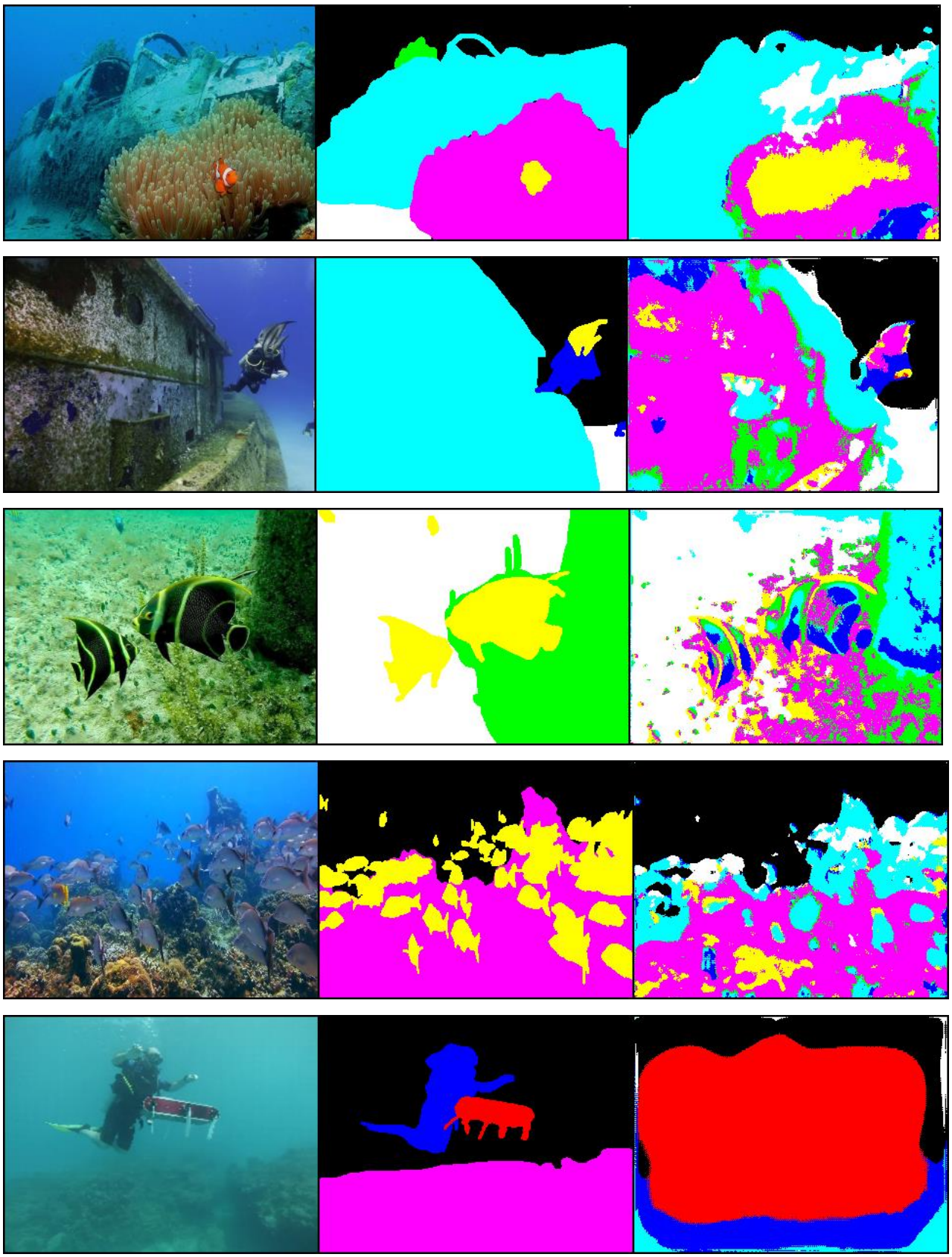

Рис.3. Результати першого покоління.

Як видно з Таблиці 3 і Рисунку 3 додавання ваг суттєво змінює те, як буде розвиватися подальше навчання моделі. Також слід зазначити що клас

4(червоний колір), який згідно з Таблицею 1 є найменш представленим, тепер з'являється. Це видно з останнього прикладу на Рисунку 3. Також цікавим є те, як цей класифікація цього класу візуально відображається протягом навчання – усі зображення, які містили 4 клас мали значну частину зображення червоним, оскільки якщо під час розрахунку функції втрат реальна маска містить червоний, а вихід нейромережі – ні, буде сильно збільшувати значення функції. Це продемонстровано на Рисунку 4.

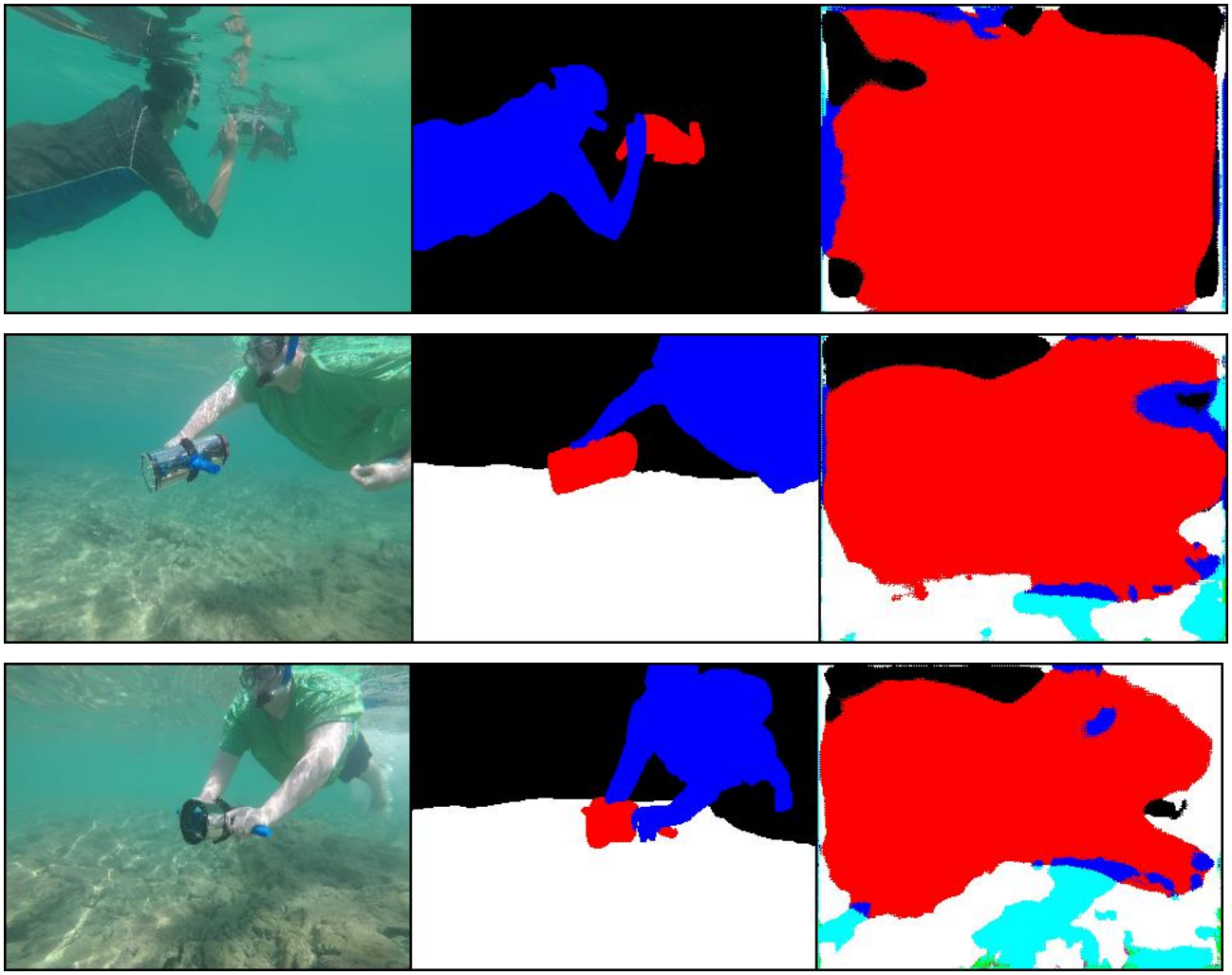

Рис.4. Специфіка передбачення 4 класу(червоний) на першому поколінні.

Під час наступних ітерацій для зменшення значення функції втрат 4 клас буде локалізуватися саме там, де він представлений на реальних масках.

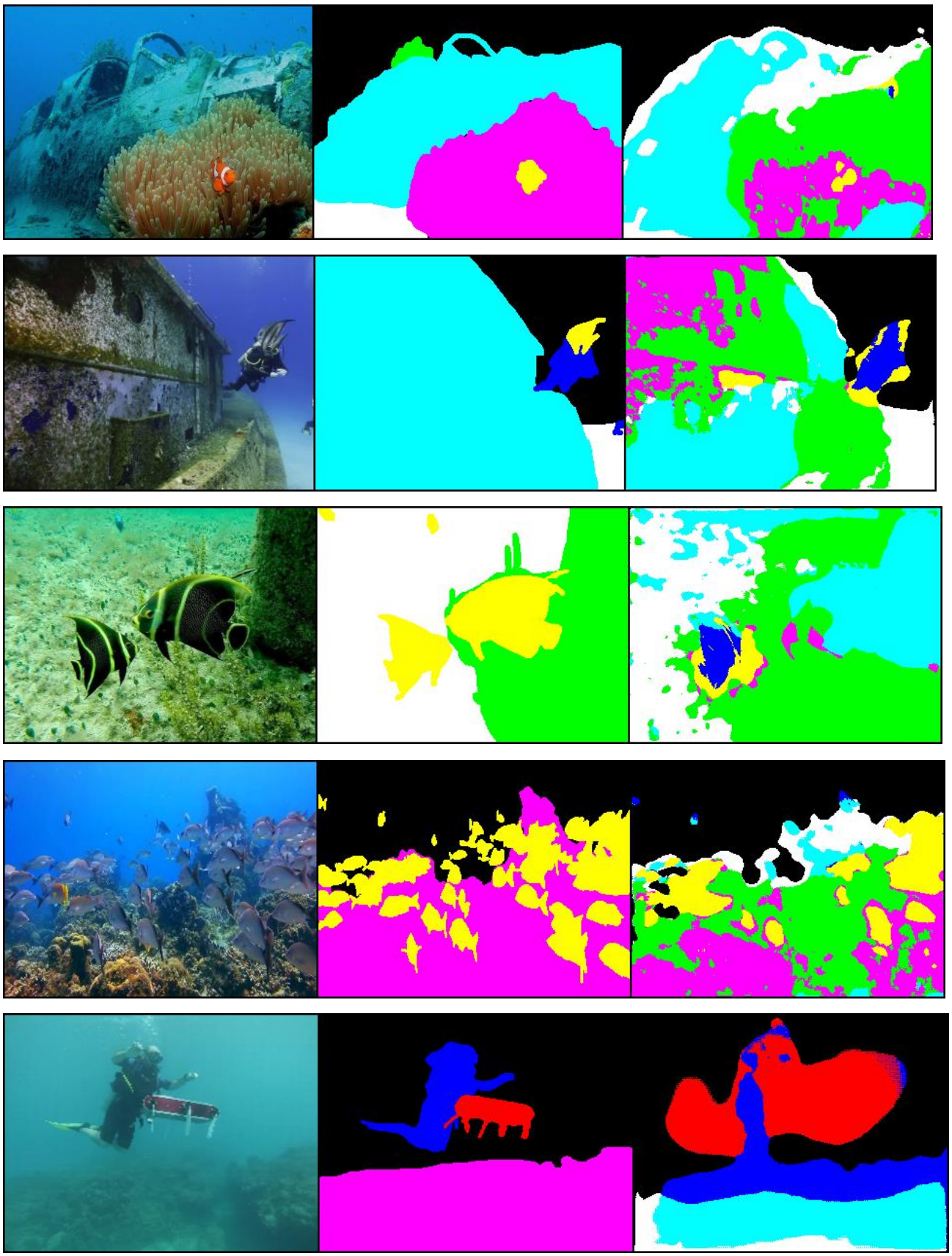

Рис.5. П’яте покоління.

З Рисунку 5 видно що модель починає відтворювати форми і локалізувати краще певні класи. Піксельна точність цього покоління 88.42, а IoU та Dice Score поступово збільшується, що видно з Таблиці 4.

Номер класу	0	1	2	3	4	5	6	7
IoU	0.665	0.128	0.168	0.118	0.057	0.250	0.145	0.389
Dice score	0.797	0.212	0.276	0.182	0.104	0.399	0.248	0.557

Таблиця 4. П’яте покоління зваженого навчання.

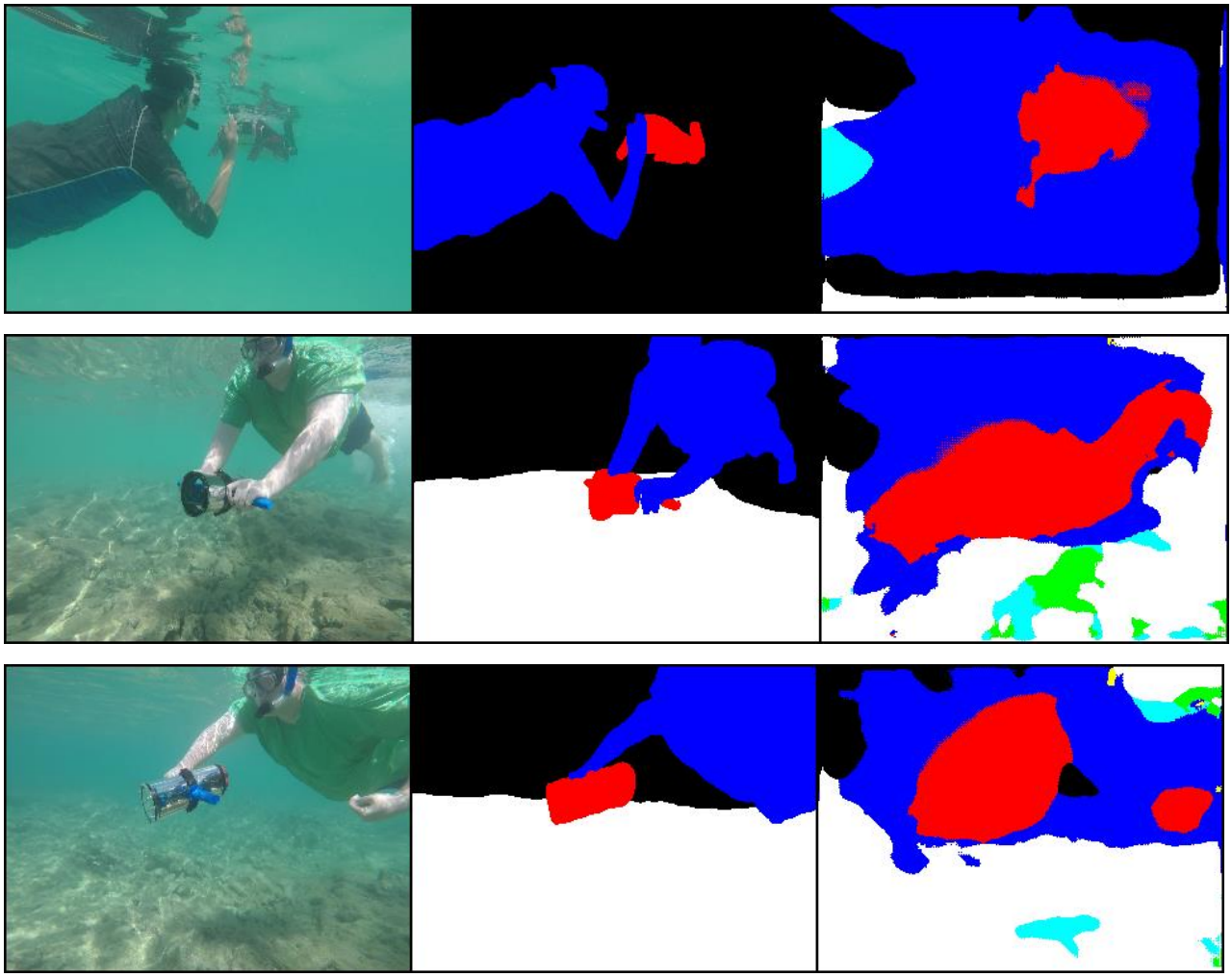

Рис.6. Специфіка передбачення 4 класу(червоний) на п’ятому поколінні.

Порівнюючи Рисунки 4 і 6 можна побачити поступову локалізацію 4 класу(червоного).

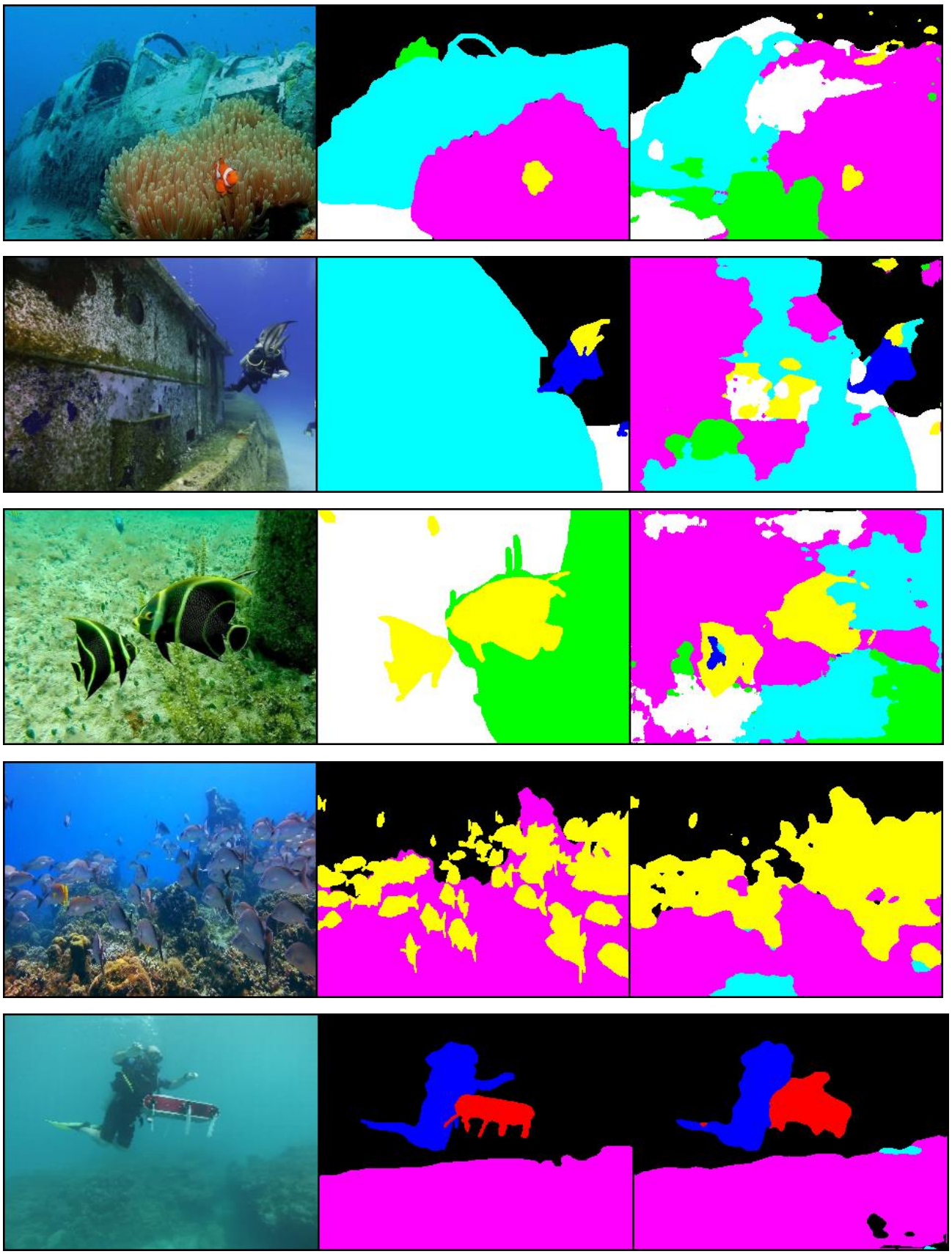

Рис.7. Результати 30 покоління.

Результати навчання 30 поколінь наведено на Рисунку 7. Піксельна точність цього покоління дорівнює 94.18.

Номер класу	0	1	2	3	4	5	6	7
IoU	0.818	0.500	0.226	0.202	0.488	0.568	0.432	0.581
Dice score	0.900	0.650	0.354	0.282	0.572	0.715	0.589	0.735

Таблиця 4. 30 покоління зваженого навчання.

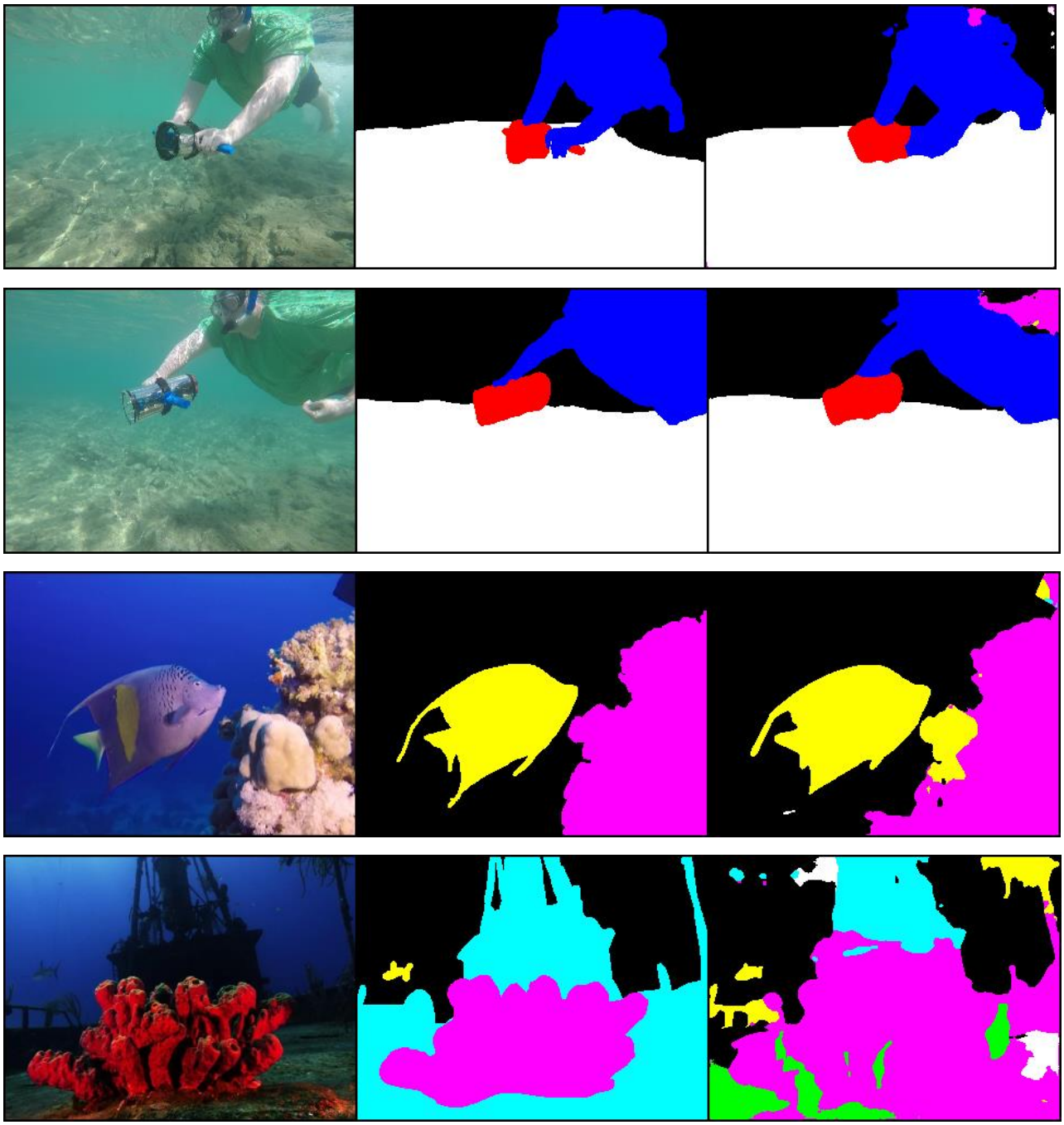

Рис.8.1. Результати 30 покоління.

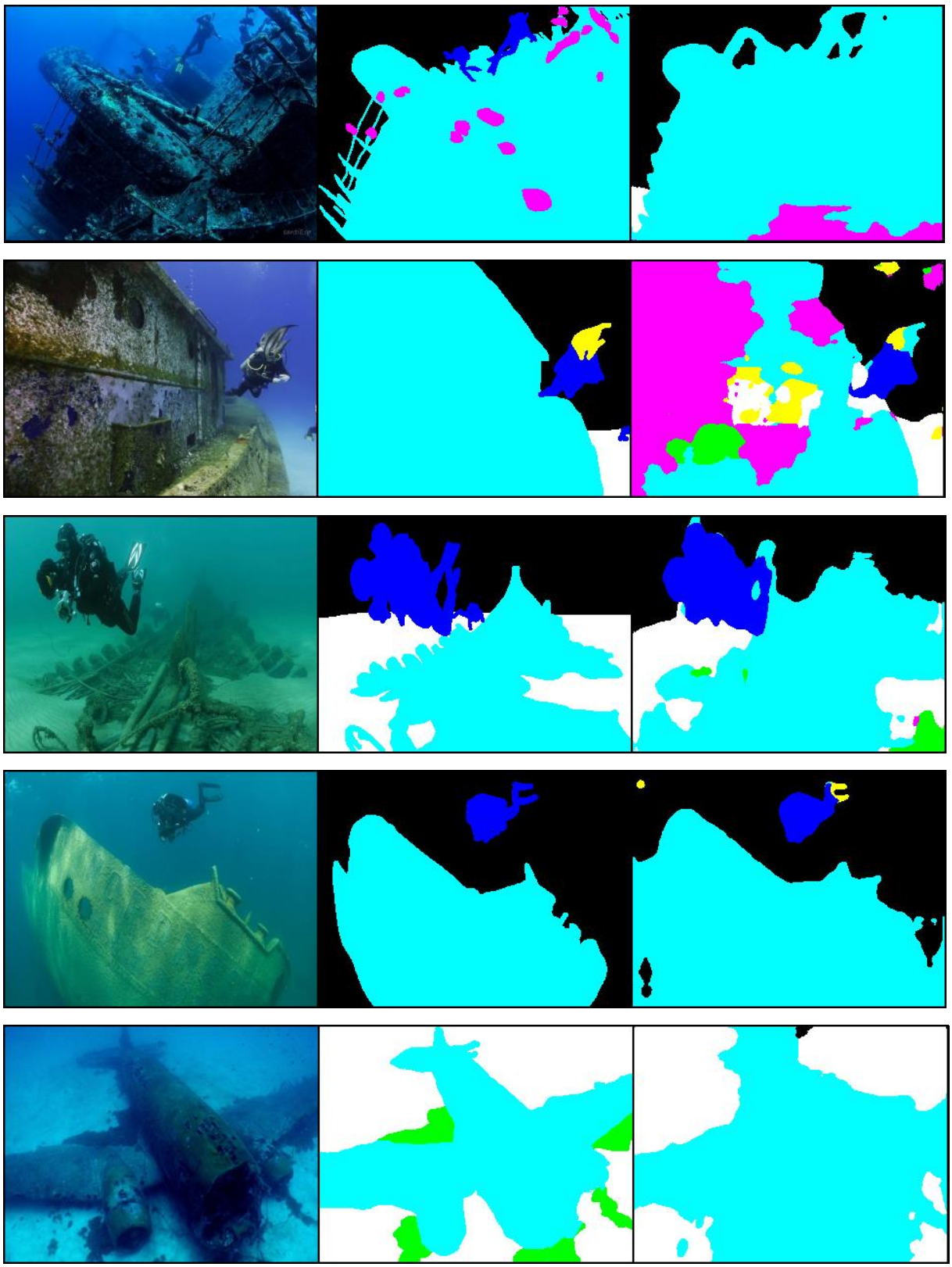

Рис.8.2. Результати 30 покоління.

Як видно з Рисунків 8.1. і 8.2. і Таблиці 4 модель досягла доволі високої якості сегментації.

ВИСНОВКИ

У даній курсовій роботі було проведено дослідження задачі семантичної сегментації зображень з використанням архітектури UNET. Одним з особливостей даної роботи було використання набору даних з зображеннями підводної зйомки, який є нестандартним доменом для сегментації зображень.

Проведений огляд літератури показав, що існують дослідження та підходи до мультикласової сегментації зображень, але здебільшого вони зосереджені на класичних доменах, таких як медицина або загальне зорове сприйняття. Використання датасету з підводною зйомкою вносить новий вимір у дослідження, оскільки він вимагає адаптації під особливості цього домену, такі як відображення кольорів, зміна освітлення та присутність підводних об'єктів.

Під час розробки та тренування моделі на датасеті з підводною зйомкою, було отримано прийнятні результати семантичної сегментації. Аналіз результатів показав, що модель може успішно впоратися з особливостями підводного домену та точно виділяти об'єкти на зображеннях. Оцінка точності моделі засвідчує її ефективність в контексті підводних зображень.

Незважаючи на те, що використання підводного датасету у сегментації зображень є викликом, результати даної роботи демонструють перспективи використання архітектури UNET у цьому домені. Це відкриває нові можливості для подальшого дослідження та вдосконалення методів сегментації зображень у підводному середовищі.

У подальших дослідженнях можливо розглянути додаткові аспекти підводної сегментації, такі як використання різних подібних наборів даних або розробка спеціалізованих алгоритмів попередньої обробки для підвищення точності моделі. Все це може сприяти подальшому розвитку сегментації зображень у підводному середовищі та в інших нестандартних доменах.

ЛІТЕРАТУРА

1. J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.
2. O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI), 2015.
3. Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. Iinternational conference on machine learning (2015), pp. 448–456.
4. A. B. L. Larsen, S. K. Darkner, and R. Larsen, "Semantic Segmentation of High-Resolution Satellite Images using Fully Convolutional Neural Networks," 2016.
5. A. B. L. Larsen, S. K. Darkner, and R. Larsen, "Evaluation of Deep Learning Architectures for Semantic Segmentation of Urban Scenes," in Proceedings of the IEEE International Conference on Image Processing (ICIP), 2016.
6. Granli, Petter. "Semantic segmentation of seabed sonar imagery using deep learning." (2019).
7. Документація PyTorch [Електронний ресурс] – Режим доступу до ресурсу: https://pytorch.org/docs/stable/index.html.
8. Документація Albumentations [Електронний ресурс] – Режим доступу до ресурсу: https://albumentations.ai/docs/.

ДОДАТКИ

1. model.py

```
import torch
import torch.nn as nn
import torchvision.transforms.functional as TF

class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DoubleConv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        return self.conv(x)

class UNET(nn.Module):
    def __init__(
            self, in_channels=3, out_channels=1, features=[64, 128, 256, 512],
    ):
        super(UNET, self).__init__()
        self.ups = nn.ModuleList()
        self.downs = nn.ModuleList()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Down part of U-Net
        for feature in features:
```

```
        self.downs.append(DoubleConv(in_channels, feature))
        in_channels = feature

    # Up part of U-Net
    for feature in reversed(features):
        self.ups.append(
            nn.ConvTranspose2d(
                feature*2, feature, kernel_size=2, stride=2,
            )
        )
        self.ups.append(DoubleConv(feature*2, feature))

    self.bottleneck = DoubleConv(features[-1], features[-1]*2)
    self.final_conv = nn.Conv2d(features[0], out_channels, kernel_size=1)

def forward(self, x):
    skip_connections = []

    for down in self.downs:
        x = down(x)
        skip_connections.append(x)
        x = self.pool(x)

    x = self.bottleneck(x)
    skip_connections = skip_connections[::-1]

    for idx in range(0, len(self.ups), 2):
        x = self.ups[idx](x)
        skip_connection = skip_connections[idx//2]

        if x.shape != skip_connection.shape:
            x = TF.resize(x, size=skip_connection.shape[2:])
```

```
        concat_skip = torch.cat((skip_connection, x), dim=1)
        x = self.ups[idx+1](concat_skip)

    return torch.softmax(self.final_conv(x), dim=1)
```

2. https://www.kaggle.com/datasets/ashish2001/semantic-segmentation-of-underwater-imagery-suim
3. train.py

```
import torch
import albumentations as A
from albumentations.pytorch import ToTensorV2
from tqdm import tqdm
import torch.nn as nn
import torch.optim as optim
from model import UNET
from utils import (
    load_checkpoint,
    save_checkpoint,
    get_loaders,
    check_accuracy,
    save_predictions_as_imgs,
    target_transform
)

LEARNING_RATE = 1e-4
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
BATCH_SIZE = 32
NUM_EPOCHS = 6
NUM_WORKERS = 2
IMAGE_HEIGHT = 240
IMAGE_WIDTH = 320
PIN_MEMORY = True
LOAD_MODEL = True
```

```
TRAIN_IMG_DIR = "data/train_images/"
TRAIN_MASK_DIR = "data/train_masks/"
VAL_IMG_DIR = "data/val_images/"
VAL_MASK_DIR = "data/val_masks/"


def train_fn(loader, model, optimizer, loss_fn, scaler):
    loop = tqdm(loader)

    for batch_idx, (data, targets_raw) in enumerate(loop):
        data = data.to(device=DEVICE)
        targets = target_transform(targets_raw.to(device=DEVICE))
        # forward
        with torch.cuda.amp.autocast():
            predictions = model(data)
            loss = loss_fn(predictions, targets)

        # backward
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        loop.set_postfix(loss=loss.item())


def main():
    train_transform = A.Compose(
        [
            A.Resize(height=IMAGE_HEIGHT, width=IMAGE_WIDTH),
            A.HorizontalFlip(p=0.5),
            A.VerticalFlip(p=0.1),
            A.Normalize(
```

```
            mean=[0.0, 0.0, 0.0],
            std=[1.0, 1.0, 1.0],
            max_pixel_value=255.0,
        ),
        ToTensorV2(),
    ],
)

val_transforms = A.Compose(
    [
        A.Resize(height=IMAGE_HEIGHT, width=IMAGE_WIDTH),
        A.Normalize(
            mean=[0.0, 0.0, 0.0],
            std=[1.0, 1.0, 1.0],
            max_pixel_value=255.0,
        ),
        ToTensorV2(),
    ],
)

model = UNET(in_channels=3, out_channels=8).to(DEVICE)
loss_fn = nn.CrossEntropyLoss(weight=torch.tensor([ 1.0000, 15.4048, 15.8858, 5.0781, 65.0134, 1.0193, 4.8255, 2.4190]))
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

train_loader, val_loader = get_loaders(
    TRAIN_IMG_DIR,
    TRAIN_MASK_DIR,
    VAL_IMG_DIR,
    VAL_MASK_DIR,
    BATCH_SIZE,
    train_transform,
    val_transforms,
```

```
        NUM_WORKERS,
        PIN_MEMORY,
    )

    if LOAD_MODEL:
        load_checkpoint(torch.load("my_checkpoint.pth.tar"), model)

    scaler = torch.cuda.amp.GradScaler()

    for epoch in range(NUM_EPOCHS):
        train_fn(train_loader, model, optimizer, loss_fn, scaler)

        # save model
        checkpoint = {
            "state_dict": model.state_dict(),
            "optimizer":optimizer.state_dict(),
        }
        save_checkpoint(checkpoint, epoch)

        # check accuracy
        check_accuracy(val_loader, model, device=DEVICE)

        # print some examples to a folder
        save_predictions_as_imgs(
            val_loader, model, folder="saved_images/", device=DEVICE
        )

if __name__ == "__main__":
    main()
```

4. dataset.py

```
import os
from PIL import Image
from torch.utils.data import Dataset
```

```
import numpy as np

class UnderwaterDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.images = os.listdir(image_dir)

    def __len__(self):
        return len(self.images)



    def __getitem__(self, index):
        img_path = os.path.join(self.image_dir, self.images[index])
        mask_path = os.path.join(self.mask_dir, self.images[index].replace(".jpg", ".bmp"))
        image = np.array(Image.open(img_path).convert("RGB"))
        mask = np.array(Image.open(mask_path).convert("RGB"), dtype=np.float32)

        if self.transform is not None:
            augmentations = self.transform(image=image, mask=mask)
            image = augmentations["image"]
            mask = augmentations["mask"]

        return image, mask
```

5. calculate_weights.py

```
import os
import numpy as np
from PIL import Image
from utils import target_transform
import torch
```

```
import datetime

def main():
    dir = "data/train_masks/"
    images = os.listdir(dir)
    all_count = torch.zeros((8), dtype=torch.int)
    for index, image in enumerate(images):
        img_path = os.path.join(dir, image)
        image = torch.stack([torch.tensor(np.array(Image.open(img_path).convert("RGB")))])
        hot_vector = target_transform(image)
        argmax = torch.argmax(hot_vector, dim=1, keepdim=True).int().flatten()
        all_count += torch.bincount(argmax.int(), minlength=8)
    weights = 1. / all_count
    print(weights)
    norm = counts / torch.max(counts)
    print(counts)
    print(norm)
    print(1. / norm)

if __name__ == "__main__":
    main()
```

6. utils.py

```
import numpy as np
import torch
import torchvision
from dataset import UnderwaterDataset
from torch.utils.data import DataLoader
from classes import class_colors

def save_checkpoint(state, epoch):
    print("=> Saving checkpoint")
```

```
    filename = f"my_checkpoint {epoch}.pth.tar"
    torch.save(state, filename)

def load_checkpoint(checkpoint, model):
    print("=> Loading checkpoint")
    model.load_state_dict(checkpoint["state_dict"])

def get_loaders(
    train_dir,
    train_maskdir,
    val_dir,
    val_maskdir,
    batch_size,
    train_transform,
    val_transform,
    num_workers=4,
    pin_memory=True,
):
    train_ds = UnderwaterDataset(
        image_dir=train_dir,
        mask_dir=train_maskdir,
        transform=train_transform,
    )

    train_loader = DataLoader(
        train_ds,
        batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=pin_memory,
        shuffle=True,
    )

    val_ds = UnderwaterDataset(
```

```
        image_dir=val_dir,
        mask_dir=val_maskdir,
        transform=val_transform,
    )

    val_loader = DataLoader(
        val_ds,
        batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=pin_memory,
        shuffle=False,
    )

    return train_loader, val_loader


def calculate_iou(outputs, targets, smooth=1e-6):
    outputs = outputs.flatten()
    targets = targets.flatten()
    intersection = (outputs * targets).sum()
    union = (outputs + targets).sum() - intersection
    iou = (intersection + smooth) / (union + smooth)
    return iou


def calculate_dice(outputs, targets, smooth=1e-6):
    outputs = outputs.flatten()
    targets = targets.flatten()
    intersection = (outputs * targets).sum()
    union = outputs.sum() + targets.sum()
    dice_score = (2 * intersection + smooth) / (union + smooth)
    return dice_score
```

```
def check_accuracy(loader, model, device="cuda"):
    num_correct = 0
    num_pixels = 0
    ious_per_class = np.zeros(8, dtype=float)
    dice_per_class = np.zeros(8, dtype=float)
    total_items = 0
    model.eval()
    i = 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device)
            y = target_transform(y.to(device))

            raw_preds = model(x)
            preds = torch.zeros_like(raw_preds,  requires_grad=True)
            preds.scatter_(1, torch.argmax(raw_preds, dim=1, keepdim=True), 1)

            num_correct += torch.sum(preds == y)
            num_pixels += torch.numel(preds)

            batch_size, num_classes, _, _ = y.shape

            ious = []
            dice_scores = []
            for class_idx in range(num_classes):
                output_class = preds[:, class_idx, :, :]
                target_class = y[:, class_idx, :, :]

                iou_class = calculate_iou(output_class, target_class)
                dice_score_class = calculate_dice(output_class, target_class)

                ious_per_class[class_idx] += iou_class * batch_size
                dice_per_class[class_idx] += dice_score_class * batch_size
```

```
            total_items += batch_size

    mean_iou = ious_per_class / total_items
    mean_dice_score = dice_per_class / total_items
    np.set_printoptions(suppress=True)
    print("IoU:", mean_iou)
    print("Dice score:", mean_dice_score)

    print(
        f"Got {num_correct}/{num_pixels} with acc {num_correct/num_pixels*100:.2f}"
    )
    model.train()

def save_predictions_as_imgs(
    loader, model, folder="saved_images/", device="cuda"
):
    model.eval()
    for idx, (x, y) in enumerate(loader):
        x = x.to(device=device)
        with torch.no_grad():
            raw_preds = model(x)
            preds = torch.zeros_like(raw_preds)
            argmax = torch.argmax(raw_preds, dim=1, keepdim=True)
            preds.scatter_(1, argmax, 1)

            print(torch.bincount(argmax.int().flatten()))
        images = reverse_transform(preds)



        torchvision.utils.save_image(
            images, f"{folder}/pred_{idx}.bmp", normalize=True
        )
        original = y.transpose(1, 3).transpose(2, 3)
```

```
        torchvision.utils.save_image(original, f"{folder}{idx}.bmp")

    model.train()

def target_transform(tensors):
    num_classes = len(class_colors)
    def process(input_tensor):
        one_hot_tensor = torch.zeros(input_tensor.shape[0], input_tensor.shape[1], num_classes)
        for h in range(input_tensor.shape[0]):
            for w in range(input_tensor.shape[1]):
                color = tuple(input_tensor[h, w].tolist())
                if color in class_colors:
                    class_index = class_colors[color]
                    one_hot_tensor[h, w, class_index] = 1
        return one_hot_tensor.transpose(0, 2).transpose(1, 2)

    return torch.stack([process(tensor) for tensor in tensors])

def reverse_transform(tensor):
    batch_size, _, height, width = tensor.shape
    argmax_tensor = torch.argmax(tensor, dim=1)

    image_tensor = torch.zeros(batch_size, 3, height, width)

    for color, index in class_colors.items():
        indices = argmax_tensor == index
        for i, c in enumerate(color):
            image_tensor[:, i, :, :][indices] = c
    return image_tensor
```

7. eval.py

```
import os
import albumentations as A
```

```
import numpy as np
import torch
import torchvision
from PIL import Image
from albumentations.pytorch import ToTensorV2

from model import UNET
from utils import reverse_transform

IMAGE = "d_r_465_.jpg"
IMAGE_HEIGHT = 240
IMAGE_WIDTH = 320

img_path = os.path.join("data/val_images/", IMAGE)
mask_path = os.path.join("data/val_masks/", IMAGE.replace(".jpg", ".bmp"))
image_original = np.array(Image.open(img_path).convert("RGB"))
mask = np.array(Image.open(mask_path).convert("RGB"), dtype=np.float32)

val_transforms = A.Compose(
    [
        A.Resize(height=IMAGE_HEIGHT, width=IMAGE_WIDTH),
        A.Normalize(
            mean=[0.0, 0.0, 0.0],
            std=[1.0, 1.0, 1.0],
            max_pixel_value=255.0,
        ),
        ToTensorV2(),
    ],
)

augmentations = val_transforms(image=image_original, mask=mask)
image = augmentations["image"].unsqueeze(0)
mask = augmentations["mask"].unsqueeze(0).transpose(1, 3).transpose(2, 3)
```

```
model = UNET(in_channels=3, out_channels=8)
model.load_state_dict(torch.load("unweighted predictions\my_checkpoint.pth unweighted.tar", map_location=torch.device('cpu'))["state_dict"])
model.eval()

with torch.no_grad():
    raw_preds = model(image)
    preds = torch.zeros_like(raw_preds)
    argmax = torch.argmax(raw_preds, dim=1, keepdim=True)
    preds.scatter_(1, argmax, 1)
pred_masks = reverse_transform(preds)

original_transform = A.Compose(
    [
        A.Resize(height=IMAGE_HEIGHT, width=IMAGE_WIDTH),
        ToTensorV2(),
    ],
)

image_original = original_transform(image=image_original)["image"].unsqueeze(0)

combined_tensor = torch.cat((image_original, mask, pred_masks), dim=0)



torchvision.utils.save_image(
    combined_tensor, f"eval\{IMAGE}.bmp", normalize=True
)
```

8. classes.py

```
class_colors = {
    (0.0, 0.0, 0.0): 0, # black - Background (waterbody)
    (0.0, 0.0, 255.0): 1, # blue - Human divers
    (0.0, 255.0, 0.0): 2, # green - Aquatic plants and sea-grass
```

```
    (0.0, 255.0, 255.0): 3, # sky - Wrecks and ruins
    (255.0, 0.0, 0.0): 4, # red - Robots (AUVs/ROVs/instruments)
    (255.0, 0.0, 255.0): 5, # pink - Reefs and invertebrates
    (255.0, 255.0, 0.0): 6, # yellow - Fish and vertebrates
    (255.0, 255.0, 255.0): 7 # white - Sea-floor and rocks
}
```