МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

Литвиненко О.Є

“_____”_____2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

“МАГІСТР”

Тема: «Програмний засіб навігації БПЛА на основі *OpenCV* та *Tensorflow*»

Виконавець: Базельцев Кирило Олександрович

Керівник: Глазок Олексій Михайлович

Нормоконтролер: Тупота Євгеній Вікторович

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра комп'ютеризованих систем управління

Спеціальність 123 «Комп'ютерна інженерія»
(шифр, найменування)

Освітньо-професійна програма «Системне програмування»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____Литвиненко О. Є.

«_____»_____2020 р.

ЗАВДАННЯ

на виконання дипломної роботи

Базельцева Кирила Олександровича

(прізвище, ім'я, по батькові)

1.Тема роботи:«Програмний засіб навігації БПЛА на основі *OpenCV* та *Tensorflow*»

затверджена наказом ректора від «27» серпня 2020 р. № 1203/ст

2.Термін виконання роботи: з 05 жовтня 2020 р. по 31 грудня 2020 р.

3.Вихідні дані до проекту (роботи): середовище програмування *Python*, програмна бібліотека *TensorFlow*, програмна бібліотека *OpenCV*, ігровий рушій *Unity*, середовище програмування *C#*

4.Зміст пояснювальної записки (перелік питань, що підлягають розробці):
Огляд нейронних мереж та аналіз способів їх застосування для реалізації комп'ютерного зору. Побудова програмного модуля комп'ютерного зору з можливостю розпізнавання об'єктів. Побудова додатку, що виконує навігацію БПЛА на основі даних з модуля комп'ютерного зору.

5.Перелік обов'язкового графічного матеріалу:
1. Програма для вирішення задачі про отримання карти глибини з вхідного відеопотоку (схема алгоритму);
2. Модуль навігації (схема алгоритму);
3. Граф зв'язків шарів згорткової нейронної мережі;
4. Приклад результату роботи модуля комп'ютерного зору.

6. Календарний план

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Проаналізувати існуючі системи автономної навігації безпілотних літальних апаратів	05.10.2020 – 10.10.2020	
2	Проаналізувати особливості роботи з *OpenCV*	10.10.2020 – 14.10.2020	
3	Проаналізувати особливості роботи з *DepthNet* та *Unity*	14.10.2020 – 28.10.2020	
4	Побудувати та натренувати нейронну мережу для вирішення проблеми аналізу глибини зображення	28.10.2020 – 14.11.2020	
5	Розробити модуль симуляції польоту БПЛА та корегування його напрямку руху	15.11.2020 – 01.12.2020	
6	Проаналізувати результати роботи та визначити способи покращення роботи розробленої системи	01.12.2020 – 05.12.2020	
7	Оформити пояснювальну записку	05.12.2020 – 10.12.2020	
8	Оформити графічний та ілюстративний матеріал	10.12.2020 – 13.12.2020	

7. Дата видачі завдання: « 05 » жовтня 2020 р.

Керівник дипломного проекту Глазок О.М.

(підпис)

Завдання прийняв до виконання Базельцев К.О.

(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломного проекту «Програмний засіб навігації БПЛА на основі *OpenCV* та *Tensorflow*»: 80 сторінок, 20 рисунків, 1 таблиця, 20 літературних джерел, 2 додатки.

НЕЙРОННІ МЕРЕЖІ, КОМП'ЮТЕРНИЙ ЗІР, БПЛА

Об'єкт дослідження – штучні нейронні мережі та комп'ютерний зір.

Предмет дослідження – використання нейронної мережі та комп'ютерного зору для автономної навігації БПЛА.

Мета дипломної роботи – дослідження можливості використання згорткової нейронної мережі для розв'язання задачі автономної навігації.

Результати виконання дипломної роботи рекомендується використовувати при подальшому покращенні автономності навігації безпілотних літальних апаратів з монокулярною камерою.

ЗМІСТ

HYPERLINK \l "_Toc9224764" РОЗДІЛ 1 ОГЛЯД ТЕХНОЛОГІЇ

1.1. Комп’ютерний зір з використанням бібліотеки *OpenCV*.....9

1.2. Бібліотека *TensorFlow*.....13

1.3. Побудова нейронних мереж за допомогою *TensorFlow* 16

1.4. Топології нейронних мереж 19

1.5. Висновки до розділу 23

РОЗДІЛ 2 ПОБУДОВА МОДУЛЯ КОМП’ЮТЕРНОГО ЗОРУ 25

2.1. Імплементація *OpenCv* у програмному додатку.....25

2.2. Бібліотека *DepthNet* 27

2.3. Побудова моделі та тренування *DepthNet* 29

2.4. Використання *DepthNet* для побудови карти глибини.....45

2.5. Висновки до розділу 53

РОЗДІЛ 3 ПОБУДОВА МОДУЛЯ НАВІГАЦІЇ БПЛА З ВИКОРИСТАННЯМ МОДУЛЯ КОМП’ЮТЕРНОГО ЗОРУ 55

3.1 Симуляція польоту дрону за допомогою *Unity* 55

3.2. Отримання карти глибини.....64

3.3. Аналіз карти глибини та надання команд БПЛА 66

3.4. Використана апаратура.....72

3.5. Висновки до розділу 75

ВИСНОВКИ..... 77

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ..... 79

ДОДАТОК А 81

ДОДАТОК Б..... 91

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ШНМ – штучна нейронна мережа;

БПЛА – безпілотний літальний апарат;

MLN – multi-layer neural network;

LSTM – long short-term memory;

ТП – тензорний процесор;

РНМ – рекурентна нейронна мережа;

RGB – red, green, blue;

ОС – операційна система;

CuDnn – Cuda Deep Neural Network library,

GD – Gradient descent,

SGD – Stochastic gradient descent,

ШІ – штучний інтелект.

ВСТУП

У дипломній роботі розглядається використання модуля комп’ютерного зору та нейронних мереж для навігації БПЛА.

Актуальність теми роботи визначається зростанням популярності безпілотних літальних апаратів та необхідністю наявності системи автономної навігації у багатьох випадках використання БПЛА.

Мета дипломної роботи – дослідження можливості використання комп’ютерного зору та нейронної мережі для розв’язання задачі автономної навігації БПЛА.

Для досягнення цієї мети необхідно виконати наступні завдання:

– виконати огляд технології комп’ютерного зору та розглянути способи її реалізації;

– описати загальну архітектуру нейронних мереж та провести порівняльний аналіз топологій з метою вибору топології, що дозволить найбільш ефективно вирішувати задачу побудови маршруту, що ґрунтується на результатах роботи модуля комп’ютерного зору;

– розробити і програмно реалізувати алгоритм роботи модуля комп’ютерного зору, що має отримувати на вхід відеопотік та повертати покадрову карту глибини;

– розробити і програмно реалізувати алгоритм роботи модуля навігації БПЛА, що має отримувати відеопотік з симуляції польоту дрону, передавати його модулю комп’ютерного зору, отримувати карту глибини, аналізувати її та корегувати напрямок руху БПЛА відповідно.

Об’єкт дослідження – штучні нейронні мережі.

Предмет дослідження – використання нейронної мережі для автономної навігації БПЛА.

Методи дослідження – розробка алгоритмів, написання коду, відладка програмного продукту.

Практичне значення отриманих результатів – можливість використання для подальшого покращення автономності літальних апаратів.

РОЗДІЛ 1
ОГЛЯД ТЕХНОЛОГІЇ КОМП'ЮТЕРНОГО ЗОРУ ТА ТОПОЛОГІЙ НЕЙРОННИХ МЕРЕЖ

За останнє десятиріччя область використання безпілотних літальних апаратів (БПЛА) стрімко збільшилася. Їх використовують у різноманітних умовах, таких як розвідка, зйомка, порятунок та картографування. БПЛА спритні в повітрі, ними можна керувати за допомогою пульта дистанційного керування, вони можуть досягати великих висот і відстаней. Багато безпілотних літальних апаратів оснащені приєднаною камерою, наприклад екшн-камерою, яка дозволяє дрону знімати фотографії та відео з різноманітних ракурсів.

Однак є деякі недоліки: керувати безпілотником може бути досить важко. Навіть при застосовуванні останніх досягнень у програмних засобах керування пілот має бути дуже обережними, оскільки втрата контролю над безпілотником може означати втрату самого БПЛА.

Ось чому останнім часом було докладено багато зусиль для створення автономних безпілотників. Автономний безпілотник може самостійно літати, не потребуючи пульта дистанційного керування: наприклад, слідуючи за людиною чи необхідним об'єктом, а також уникаючи перешкоди.

Незалежно від цілей використання, автономне орієнтування – одна з ключових бажаних особливостей БПЛА як у приміщенні, так і на відкритому повітрі. Є кілька напрямків рішення цієї проблеми. Було проведено значну кількість дослідів з метою використання додаткових спеціальних засобів зондування, такі як РАДАР та *LIDAR*, які забезпечують високу точність навігації та можливість уникнення перешкод, тим самим забезпечуючи можливість автономних польотів *[4]*. Недоліком є те, що такі радіолокаційні системи дорогі і мають значну вагу. Це робить використання таких систем майже неможливим разом з дешевими мікро безпілотними літальними апаратами (*MAV*). Ультразвуковий *SONAR* є більше дешевою альтернативою, але він страждає від недостатньої точності і зменшеного поля зору.

Альтернативою до таких методів є використання технології комп'ютерного зору та нейронних мереж для побудови оптимального маршруту. Для цього у даній роботі пропонується використовувати програмні модулі, основані на *OpenCV* або *Tensorflow.*

1.1. Комп'ютерний зір з використанням бібліотеки *OpenCV*

Комп'ютерний зір або комп'ютерне бачення – теорія та технологія створення машин, які можуть проводити виявлення, стеження та визначення об'єктів.

Як наукова дисципліна, комп'ютерний зір належить до теорії та технології створення штучних систем, які отримують інформацію у вигляді зображень. Відеодані можуть бути представлені у вигляді багатьох форм, таких як відеопотік, зображення з різних камер або тривимірними даними з медичного сканера.

Як технологічна дисципліна, комп'ютерний зір прагне застосувати теорії та моделі комп'ютерного зору до створення систем комп'ютерного зору. Прикладами таких систем можуть бути:

– системи керування процесами (промислові роботи, автономні транспортні засоби);

– системи відеоспостереження;

– системи організації інформації (наприклад, для індексації баз даних зображень);

– системи моделювання об'єктів або навколишнього середовища (аналіз медичних зображень, топографічне моделювання);

– системи взаємодії (наприклад, пристрої введення для систем людино-машинної взаємодії).

Ще в середині минулого століття обробка зображень була здебільшого аналоговою і виконувалась оптичними пристроями. Подібні оптичні методи досі важливі, в таких областях як, наприклад, голографія. Тим не менш, з різким зростанням продуктивності комп'ютерів, ці методи все в більшій мірі витіснялися методами цифрової обробки зображень. Методи цифрової обробки зображень

зазвичай є більш точними, надійними, гнучкими і простими в реалізації, ніж аналогові методи. У цифровій обробці зображень широко застосовується спеціалізоване обладнання, таке як процесори з конвеєрною обробкою інструкцій та багатопроцесорні системи. Особливою мірою це стосується систем обробки відео. Обробка зображень виконується також за допомогою програмних засобів комп'ютерної математики, наприклад, *MATLAB, Mathcad, Maple, Mathematica* і ін Для цього в них використовуються як базові засоби, так і пакети розширення *Image Processing*.

Комп'ютерний зір також може бути описаний як доповнення (але не обов'язково протилежність) біологічному зору. У біології вивчається зорове сприйняття людини і різноманітних тварин, в підсумку створюються моделі роботи таких систем в термінах фізіологічних процесів. Комп'ютерний зір, з іншого боку, вивчає і описує системи комп'ютерного зору, які виконано апаратно або програмно. Міждисциплінарний обмін між біологічним та комп'ютерним зором виявився досить продуктивним для обох наукових галузей.

Підрозділи комп'ютерного зору охоплюють відтворення дій, виявлення подій, стеження, розпізнавання образів, відновлення зображень.

Комп'ютерний зір, Обробка зображень, Аналіз зображень, Зір робота і Машинний зір — тісно пов'язані області. Але досі точно не визначено, чи є вони розділами однієї, ширшої галузі. При докладному аналізі може здатись, що це лише різні назви однієї і тієї ж області. Щоб не виникло плутанини, прийнято розрізняти їх як напрямки, зосереджені на певному предметі вивчення. Нижче наведено опис деяких з них, найбільш важливих.

Обробка зображень і Аналіз зображень в основному зосереджені на роботі з двовимірними зображеннями, тобто як перетворити одне зображення на інше. Наприклад, попіксельні операції збільшення контрастності, операції з виділення країв, усунення шумів чи геометричні перетворення, такі як обертання зображення. Дані операції припускають, що обробка/аналіз зображення діють незалежно від вмісту самих зображень.

Комп'ютерний зір зосереджується на обробці тривимірних сцен, спроєктованих на одне чи декілька зображень. Наприклад, відновлення структури

чи іншої інформації про тривимірну сцену по одному чи декільком зображенням. Комп'ютерний зір часто залежить від більш чи менше складних припущень відносно того, що представлено на зображеннях.

Машинний зір зосереджується на застосуванні, в основному промисловому, наприклад, автономні роботи і системи зорової перевірки та вимірювання. Це означає, що технології давачів зображення і теорії керування пов'язані з обробкою відеоданих для керування роботом і обробка даних в реальному часі здійснюється апаратно чи програмно.

Типові завдання обробки зображень:

– Розпізнавання тексту;
– Обробка супутникових знімків;
– Машинний зір;
– Обробка даних для виділення різних характеристик;
– Обробка зображень у медицині;
– Ідентифікація особи (по обличчю, райдужці, дактилоскопічним даними);
– Автоматичне управління автомобілями;
– Визначення форми об'єкта;
– Визначення переміщення об'єкта;
– Накладення фільтрів.

OpenCV – бібліотека функцій та алгоритмів комп'ютерного зору, обробки зображень і чисельних алгоритмів загального призначення з відкритим кодом. Бібліотека надає засоби для обробки і аналізу вмісту зображень, у тому числі розпізнавання об'єктів на фотографіях (наприклад, осіб і фігур людей, тексту тощо), відстежування руху об'єктів, перетворення зображень, застосування методів машинного навчання і виявлення загальних елементів на різних зображеннях.

Бібліотека розроблена *Intel* і нині підтримується *Willow Garage* та *Itseez*. Вихідній код бібліотеки написаний мовою *C++* і поширюється під ліцензією *BSD*. Біндинги підготовлені для різних мов програмування, таких як *Python, Java, Ruby, Matlab, Lua* та інших. Може вільно використовуватися в академічних та комерційних цілях.

Офіційно проект *OpenCV* був запущений у 1999 році за ініціативою *Intel Research* з ціллю розвивати *CPU*-ресурсомісткі додатки *[5]*. Основними вкладниками у проект була *Intel's Performance Library Team* та певна кількість експертів з чисельної оптимізації у *Inter Russia*. На перших етапах розвитку *OpenCV* основними задачами бібліотеки були:

– розвивати дослідження у напрямку комп'ютерного зору, забезпечуючи добре оптимізований та відкритий код бібліотеки;

– поширювати знання у сфері комп'ютерного зору, забезпечуючи загальну інфраструктуру, яку б могли розвивати розробники, таким чином код ставатиме більш легким для сприйняття та обміну;

– розвивати засновані на роботі з комп'ютерним зором комерційні додатки, створюючи не залежну від платформи, оптимізовану та безкоштовну бібліотеку. Для цього використовувалася ліцензія, яка не вимагала від таких комерційних додатків бути відкритими.

Перша альфа-версія *OpenCV* була оприлюднена на *IEEE* конференції з комп'ютерного зору й розпізнавання образів у 2000 році. П'ять бета-версій було випущено у період між 2001 і 2005 роками. Перша версія 1.0 була випущена у 2006 році. У середині 2008 року *OpenCV* отримала корпоративну підтримку від *Willow Garage* і знову перейшла у стадію активної розробки. «Пре-релізна» версія 1.1 була випущена у жовтні 2008 року.

Другий великий випуск *OpenCV* відбувся у жовтні 2009 року. *OpenCV* 2 включала у себе серйозні зміни у інтерфейсі *C++*. Ці зміни спрямовані на більш прості, тип-безпечні моделі, додавання нових функцій, і кращу реалізацію існуючих моделей в плані швидкодії (особливо на багатоядерних системам). Офіційні релізи надалі відбуваються кожні 6 місяців і розробкою займається незалежна команда з Росії, яка підтримується комерційними корпораціями.

У серпні 2012 року підтримку *OpenCV* було передано некомерційній організації *OpenCV.org.*

Бібліотека містить понад 2500 оптимізованих алгоритмів, серед яких повний набір як класичних так і практичних алгоритмів машинного навчання і комп'ютерного зору. Алгоритми *OpenCV* застосовують у таких сферах:

– аналіз та обробка зображень;

– системи з розпізнавання обличчя;

– ідентифікації об’єктів;

– розпізнавання жестів на відео;

– відстежування переміщення камери;

– побудова *3D*-моделей об’єктів;

– створення *3D*-хмар точок зі стерео камер;

– склеювання зображень між собою, для створення зображень всієї сцени з високою роздільною здатністю;

– система взаємодії людини з комп’ютером;

– пошуку схожих зображень із бази даних;

– усування ефекту червоних очей при фотозйомці зі спалахом;

– стеження за рухом очей;

– аналіз руху;

– ідентифікація об’єктів;

– сегментація зображення;

– трекінг відео;

– розпізнавання елементів сцени і додавання маркерів для створення доповненої реальності, та інші.

OpenCV написана на *C++* і її основний інтерфейс також реалізовано на *C++*, але бібліотека і досі надає старіший *C* інтерфейс. Наразі реалізовано інтерфейс мовами *Python, Java* і *MATLAB* / *OCTAVE* (починаючи з версії 2.5). *API* для цих інтерфейсів можна знайти в онлайн документації. Оболонки для інших мов, таких як *C#, CH, Ruby* були розроблені з метою охоплення ширшої аудиторії.

Всі нові розробки та алгоритми *OpenCV* наразі розробляються у *C++* інтерфейсі.

1.2. Бібліотека *TensorFlow*

TensorFlow – відкрита програмна бібліотека для машинного навчання цілій низці задач, розроблена компанією *Google* для задоволення її потреб у системах,

здатних будувати та тренувати нейронні мережі для виявлення та розшифрування образів та кореляцій, аналогічно до навчання й розуміння, які застосовують люди.

Її наразі застосовують як для досліджень, так і для розробки продуктів *Google*, часто замінюючи на його ролі її закритого попередника *DistBelief*. *TensorFlow* було початково розроблено командою *Google Brain* для внутрішнього використання в *Google*, поки її не було випущено 9 листопада 2015 року під відкритою ліцензією (*Apache* 2.0).

Починаючи з 2011 року, *Google Brain* будувала *DistBelief* як власницьку систему машинного навчання на основі нейронних мереж глибинного навчання. Її використання швидко росло в різноманітних компаніях *Alphabet* як у дослідницьких, так і в комерційних застосуваннях. *Google* призначила декількох інформатиків, включно з Джеффом Діном, яким було доручено спростити та переробити кодову основу *DistBelief* на швидшу, надійнішу бібліотеку рівня застосунків, якою стала *TensorFlow*. 2009 року команда під проводом Джефрі Хінтона реалізувала узагальнене зворотне поширення та інші вдосконалення, які дозволити породжувати нейронні мережі з суттєво вищою точністю, наприклад, це дозволило на 25% знизити похибки в розпізнаванні мови.

TensorFlow є системою машинного навчання *Google Brain* другого покоління, випущеною як відкрите програмне забезпечення 9 листопада 2015 року. В той час як еталонна реалізація працює на одиничних пристроях, *TensorFlow* може працювати на декількох центральних та графічних процесорах (включно з додатковими розширеннями *CUDA* для обчислень загального призначення на графічних процесорах) *[6]*. *TensorFlow* доступна для 64-розрядних *Linux, macOS, Windows*, та для мобільних обчислювальних платформ, включно з *Android* та *iOS*.

Обчислення *TensorFlow* виражаються як станові графи потоків даних. Назва *TensorFlow* походить від операцій, що такі нейронні мережі виконують над багатовимірними масивами даних. Ці багатовимірні масиви називають «тензорами». В червні 2016 року Джефф Дін з *Google* заявив, що *TensorFlow* згадували 1500 репозиторіїв на *GitHub*, лише 5 з яких були від *Google*.

У травні 2016 року *Google* анонсувала свій тензорний процесор (ТП) – спеціалізовану мікросхему, побудовану спеціально для машинного навчання, й підігнану під *TensorFlow*. ТП є програмованим ШІ-прискорювачем, розробленим для забезпечення високої продуктивності в арифметиці низької точності (наприклад, 8-бітній), і спрямованим радше на використання або виконання моделей, аніж на їхнє тренування. *Google* оголосила, що вони використовували ТП у своїх центрах обробки даних понад рік, і виявили, що вони забезпечують для машинного навчання на порядок кращу оптимізовану продуктивність на ват споживаної потужності, порівняно з серверами загального використання.

TensorFlow забезпечує ППІ для *Python*, а також для *C++*, *Haskell, Java* та *Go*. Платформа спочатку розроблена командою *Google Brain* і використовуються в сервісах *Google* для розпізнавання мови, виділення облич на фотографіях, визначення схожості зображень, відсіювання спаму в *Gmail*, підбору новин у *Google News* і організації перекладу з урахуванням смислу. Розподілені системи машинного навчання можна створювати на типовому обладнанні, завдяки вбудованій підтримці в *TensorFlow* рознесення обчислень на кілька *CPU* або *GPU*.

Серед застосувань, для яких *TensorFlow* є основою, є програмне забезпечення автоматизованого опису зображень, таке як *DeepDream*. 26 жовтня 2015 року *Google* офіційно реалізувала *RankBrain*, який підтримує *TensorFlow*. *RankBrain* тепер обробляє суттєве число пошукових записів, замінюючи та доповнюючи традиційні статичні алгоритми на основі результатів пошуку.

Іншими застосуванням є використання у складі програм *FakeApp* з метою безшовного поєднання фото- та відеозображень для створення підробних, але правдоподібних відео, відомих під назвою *Deepfake*.

TensorFlow надає бібліотеку готових алгоритмів чисельних обчислень, реалізованих через графи потоків даних (*data flow graphs*). Вузли в таких графах реалізують математичні операції або точки вводу/виводу, в той час як ребра графа представляють багатовимірні масиви даних (тензори), які перетікають між вузлами. Вузли можуть бути закріплені за обчислювальними пристроями і виконуватися асинхронно, паралельно обробляючи разом все підходящі до них

тензори, що дозволяє організувати одночасну роботу вузлів в нейронної мережі за аналогією з одночасною активацією нейронів в мозку.

Інтеграція *TensorFlow* з *Python* забезпечується не лише через *pip*, а й у дистрибутиві *Anaconda*.

TensorFlow також може перекладати частину роботи свого штучного інтелекту з центру обробки даних повністю на смартфони. Як правило, коли користувач використовує на своєму телефоні програму, що у свою чергу використовує нейронні мережі, вона не може працювати без надсилання інформації назад до центру обробки даних. Але *Google* також вдосконалили свій механізм нейронних мереж, щоб у деяких випадках він міг працювати на самому телефоні. Так компанія створила свій додаток *Google Translate*. *Google* навчає програму розпізнавати слова та перекладати їх на іншу мову всередині своїх центрів обробки даних, але після навчання програма може працювати самостійно – без з'єднання з Інтернетом.

1.3. Побудова нейронних мереж за допомогою *TensorFlow*

TensorFlow надає набір інструментів для побудови архітектур нейронних мереж, а також навчання та обслуговування моделей. Ця бібліотека пропонує різні рівні абстракції, тому її можливо використовувати для банальних процесів машинного навчання на високому рівні або пройти більш глибокий опис і самостійно написати розрахунки, що відбуваються на більш низьких рівнях.

TensorFlow пропонує багато видів шарів у своєму пакеті *tf.layers*. Модуль спрощує створення шару для моделі нейронної мережі, навіть без конфігурації багатьох параметрів. На даний момент він підтримує типи шарів, що використовуються здебільшого в конволюційних мережах. Для інших типів мереж, таких як *RNN*, може знадобитися переглянути *tf.contrib.rnn* або *tf.nn*. Найбільш базовим типом шару є повністю зв'язаний шар. Для його реалізації потрібно лише налаштувати вхідні параметри та розмір даних у класі *Dense*. Для інших типів шарів може знадобитися більше параметрів, але вони реалізовані

таким чином, щоб впровадити поведінку за замовчуванням та заощадити час розробників.

Існують певні розбіжності щодо того, що слід вважати шаром нейронної мережі. Одна думка стверджує, що шар повинен зберігати навчені параметри (наприклад, ваги). Це означає, наприклад, що при застосуванні функції активації не використовується додатковий шар. Дійсно, *tf.layers* реалізує таку функцію, використовуючи параметр активації. Однак шари, введені в модуль, не завжди суворо дотримуються цього правила.

Типовою згортковою мережею є послідовність згортки та об'єднання пар, за якою йдуть кілька повністю зв'язаних шарів. Згортка схожа на невелику нейронну мережу, яка застосовується неодноразово. В результаті мережеві шари стають набагато меншими, але збільшуються в глибину. Пул – це операція, яка зазвичай зменшує розмір вхідного зображення. Максимальне об'єднання – це найпоширеніший алгоритм об'єднання, який виявився ефективним у багатьох завданнях комп'ютерного зору.

Для побудови нейронних мереж за допомогою *TensorFlow* рекомендується використовувати високорівневий *API Keras*. *Keras* – відкрита нейромережна бібліотека, написана мовою *Python*. Вона здатна працювати поверх *TensorFlow, Microsoft Cognitive Toolkit, R, Theano* та *PlaidML*. Спроєктовану для уможливлення швидких експериментів з мережами глибинного навчання, її зосереджено на тому, щоби вона була зручною в користуванні, модульною та розширюваною. *Keras* містить численні втілення широко вживаних нейромережних будівельних блоків, таких як шари, цільові та передавальні функції, оптимізувальники та безліч інструментів для спрощення роботи із зображеннями та текстом, щоби спрощувати кодування, потрібне для написання глибинно-нейромережного коду. Її код розміщено на *GitHub*, а до форумів спільнотної підтримки належать сторінка питань *GitHub* та канал *Slack*. На додачу до стандартних нейронних мереж, *Keras* містить підтримку згорткових та рекурентних нейронних мереж.

Перед побудовою моделі необхідно отримати вибірку даних та передобробити її. Наприклад для зображень передобробка буде включати в себе наступні шаги:

– зміна кольору зображення на чорно-білий,

– перетворення зображень у масив чисел зі значеннями від 0 до 255,

– нормалізація чисел діленням кожного числа на 255, щоб отримати числа від 0 до 1.

З таким набором масивів може працювати модель нейронної мережі. Побудова моделі нейронної мережі вимагає правильної конфігурації кожного шару, і подальшої компіляції моделі. Базовим будівельним блоком нейронної мережі є шар. Шари витягають образи з даних, що в них подаються. Велика частина навчання нейронної мережі складається із з'єднання в послідовність простих шарів. Більшість шарів мають параметри, які налаштовуються під час навчання.

Перший шар мережі для обробки та класифікації зображень буде перетворювати формат зображення з двовимірного масиву в одновимірний. Цей шар не має параметрів для навчання, він тільки переформатує дані. Після цього, нейронна мережа містить два шари. Це повнозв'язні нейронні шари. Кожен вузол містить оцінку, що вказує ймовірність приналежності зображення до одного з класів.

Ще кілька параметрів моделі конфігуруються на етапі компіляції моделі:

– Функція втрат (*Loss function*) – вимірює точність моделі під час навчання. Ціль навчання нейронної мережі в мінімізації результатів цієї функції щоб "направити" модель в правильному напрямку.

– Оптимізатор (*Optimizer*) – показує яким чином оновлюється модель на основі вхідних даних і функції втрат.

– Метрики (*Metrics*) – використовуються для моніторингу тренування і тестування моделі. Приклад з зображеннями використовує метрику *accuracy*, що дорівнює частині правильно класифікованих зображень.

Навчання моделі нейронної мережі вимагає виконання наступних кроків:

– подання тренувальних даних в модель;

– навчання моделі асоціювати зображення з правильними класами;
– прогнозування моделлю результатів для перевірочних даних.

В процесі навчання моделі відображаються метрики втрати (*loss*) і точності (*accuracy*). Отримана на перевірочному наборі даних точність може виявитися трохи нижче, ніж на тренувальному. Цей розрив між точністю на тренуванні і тесті є прикладом перенавчання (*overfitting*) *[7]*. Перенавчання виникає, коли модель показує на нових даних гірший результат, ніж на тих, на яких вона навчалася, тому що було побудовано занадто багато асоціацій з занадто великою вагою під час обробки тренувальних даних і тепер їй важко класифікувати нові дані.

На цьому етапі модель можна використовувати для створення прогнозів, що будуть являти собою масив чисел, що представляють відсоток “впевненості” нейронної мережі у класі зображення.

1.4. Топології нейронних мереж

Нейронні мережі розрізняють за топологічними типами відповідно до структури зв’язків між нейронами мережі, а також за типом використаних формальних нейронів.

Нейронна мережа прямого поширення, нейромережа прямого розповсюдження (рис. 1.1.) – вид нейронної мережі, в якій сигнали поширюються в одному напрямку, починаючи від вхідного шару нейронів, через приховані шари до вихідного шару і на вихідних нейронах отримується результат опрацювання сигналу. В мережах такого виду немає зворотних зв’язків. Протилежним видом нейронних мереж із зворотними зв’язками є рекурентні нейронні мережі. Прикладом нейронної мережі прямого поширення є перцептрон Розенблатта, від якого і беруть свій початок нейромережі прямого розповсюдження. В літературі часто термін перцептрон, багатошаровий перцептрон та нейромережа прямого поширення застосовуються синонімічно. Власне, між різними видами перцептронів спільне одне – вони усі є нейромережами з прямим поширенням

сигналу, різняться в основному кількістю шарів, функцією активації та методом навчання.

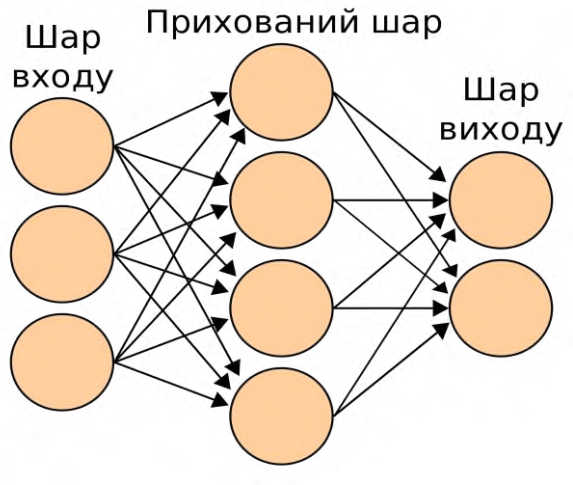

Рис. 1.1. Архітектура нейронної мережі прямого поширення

Рекурентні нейронні мережі (РНМ) (рис. 1.2) – це клас штучних нейронних мереж, у якому з'єднання між вузлами утворюють граф орієнтований у часі. Це створює внутрішній стан мережі, що дозволяє їй проявляти динамічну поведінку в часі. На відміну від нейронних мереж прямого поширення, РНМ можуть використовувати свою внутрішню пам'ять для обробки довільних послідовностей входів. Це робить їх застосовними до таких задач, як розпізнавання несегментованого неперервного рукописного тексту та розпізнавання мовлення.

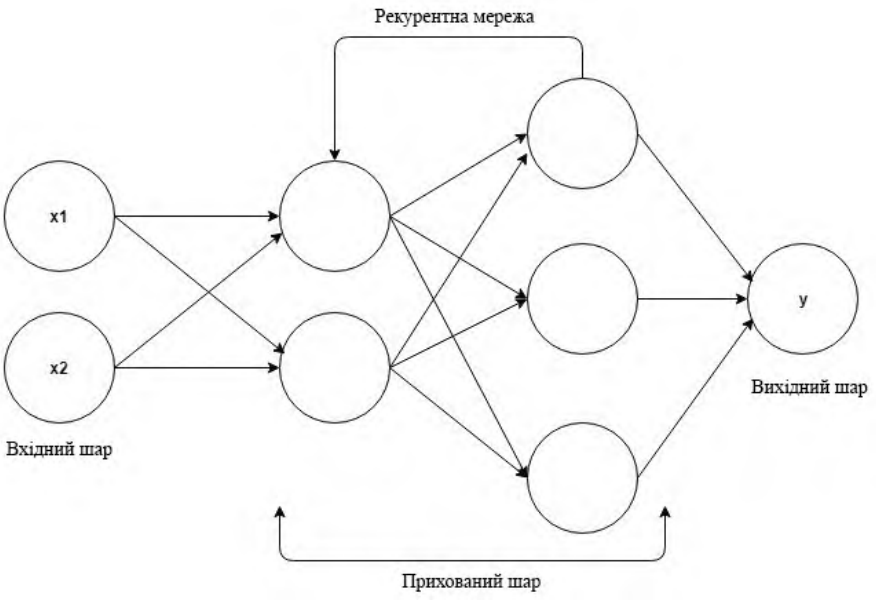

Рис. 1.2. Архітектура рекурентної нейронної мережі

Мережа радіально базисних функцій (рис. 1.3) – це штучна нейронна мережа, яка використовує радіальні базисні функції у якості функції активації. Виходом мережі є лінійна комбінація радіальних базисних функцій входу та параметрів нейрона. Мережі радіальних базисних функцій мають багато застосувань, зокрема, такі як апроксимацію функції*[*, прогнозування часових рядів, задачі класифікації та керування системою.

Рекурсивні нейронні мережі (рис 1.4) – це клас глибинних нейронних мереж, створюваних рекурсивним застосуванням одного й того ж набору ваг до структури, щоби здійснювати структурне передбачування вхідних структур мінливого розміру, або скалярне передбачування на них, шляхом обходу заданої структури в топологічній послідовності.

РНМ були успішними, наприклад, в навчанні послідовнісних та деревних структур в обробці природної мови, головним чином неперервних представлень фраз та речень на основі векторного представлення слів.

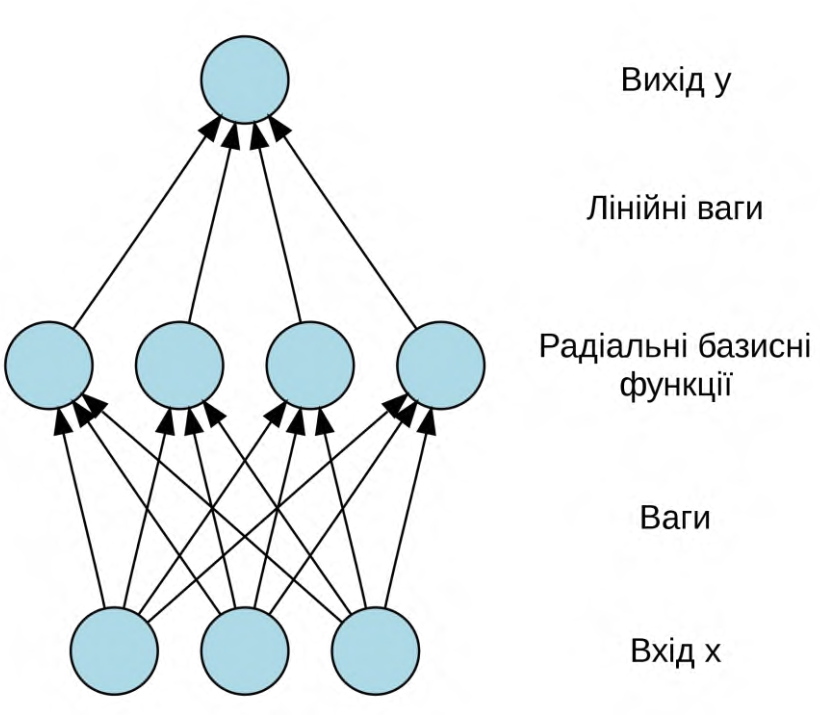

Рис. 1.3. Архітектура мережі радіальних базисних функцій

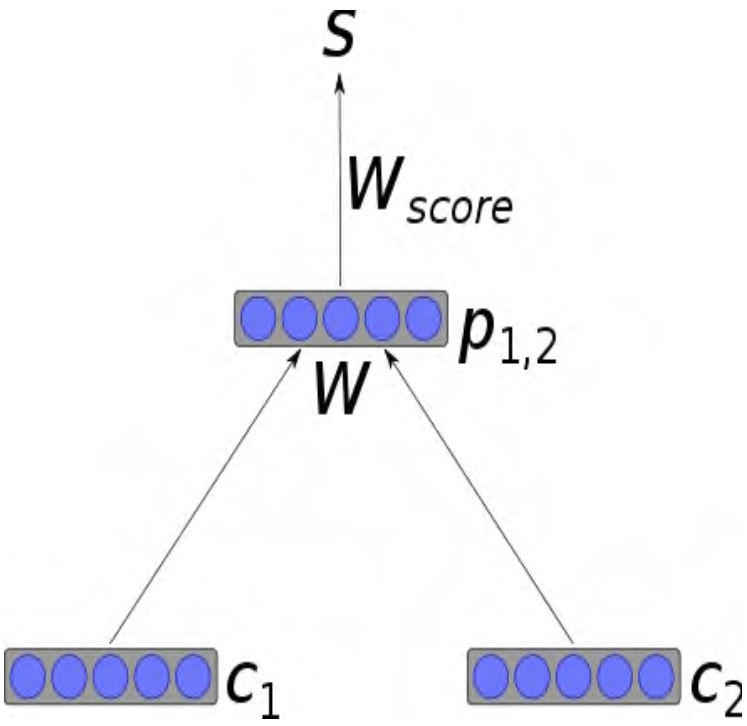

Рис. 1.4. Архітектура рекурсивної нейронної мережі

Самоорганізаційна карта Кохонена – нейронна мережа з некерованим навчанням, яка використовується для конструювання багатовимірного простору в простір з нижчою розмірністю (найчастіше, двовимірний). Створює дискретне представлення вхідних просторів навчальних вибірок, які називаються картою, і тому використання цього типу нейронної мережі є методом для зниження розмірності. Самоорганізовані карти відрізняються від інших штучних нейронних мереж, оскільки вони застосовують конкурентне навчання, яке є протилежним до навчання з виправленням помилок (наприклад, метод зворотного поширення помилки з градієнтним спуском), і в тому сенсі, що вони використовують функцію сусідства для збереження топологічних властивостей вхідного простору.

Нейронна мережа Гопфілда – це тип рекурентної, повнозв’язної, штучної нейронної мережі з симетричною матрицею зв’язків. У процесі роботи динаміка таких мереж сходиться до одного з положень рівноваги. Ці положення рівноваги є локальними мінімумами функціоналу, що називається енергія мережі (у найпростішому випадку – локальними мінімумами негативно певної квадратичної форми на *n*-вимірному кубі). Така мережа може бути використана як автоасоціативна пам’ять, як фільтр, а також для розв’язання деяких завдань оптимізації. На відміну від багатьох нейронних мереж, що працюють до

отримання відповіді через певну кількість тактів, мережі Хопфілда працюють до досягнення рівноваги, коли наступний стан мережі дорівнює попередньому.

Згорткові нейронні мережі (рис. 1.5) в машинному навчанні – це клас глибинних штучних нейронних мереж прямого поширення, який успішно застосовувався до аналізу візуальних зображень. ЗНМ використовують різновид багатошарових перцептронів, розроблений так, щоби вимагати використання мінімального обсягу попередньої обробки.

Рис. 1.5. Архітектура згорткової нейронної мережі

Вони відомі також як інваріантні відносно зсуву або просторово інваріантні штучні нейронні мережі, виходячи з їхньої архітектури спільних ваг та характеристик інваріантності відносно паралельного перенесення.

Згорткові мережі взяли за основу біологічний процес, а саме схему з'єднання нейронів зорової кори тварин. Окремі нейрони кори реагують на стимули лише в обмеженій області зорового поля, відомій як рецептивне поле. Рецептивні поля різних нейронів частково перекриваються таким чином, що вони покривають усе зорове поле.

Таким чином для вирішення задачі про автономну навігацію БПЛА, а саме для створення модуля комп'ютерного зору доцільно використовувати згорткову нейронну мережу, бо саме такі нейронні мережі мають найбільшу точність для роботи з зображеннями *[8]*.

1.5. Висновки до розділу

У першому розділі дипломного проекту було обґрунтовано актуальність теми – розв'язання задачі навігації безпілотного літального апарату за допомогою

модулів комп’ютерного зору та навігації на основі *OpenCV* і нейронної мережі, що побудована з використанням *TensorFlow.* У розділі будо проведено огляд методології побудови модулів комп’ютерного зору на основі *OpenCV,* а також способів побудови нейронних мереж за допомогою *TensorFlow.* Було розглянуто топології нейронних мереж, таких як: нейронна мережа прямого поширення, рекурентна нейронна мережа, рекурсивна нейронна мережа, згорткова нейронна мережа; та обґрунтовано вибір згорткової нейронної мережі для побудові модуля комп’ютерного зору.

РОЗДІЛ 2
ПОБУДОВА МОДУЛЯ КОМП'ЮТЕРНОГО ЗОРУ

Глибиною у контексті роботи модуля комп'ютерного зору є відстань між розпізнаними об'єктами та камерою безпілотного літального апарату. Відстань можна буде вичислити за допомогою нейронної мережі, що буде повертати прогноз відстаней базуючись на зсуві зображення попіксельно між двома кадрами відеопотоку. Таким чином модуль комп'ютерного зору повинен вирішувати задачу розпізнання глибини оточення БПЛА. Тобто алгоритм роботи модуля має виконувати наступні кроки:

1. Отримання відеопотоку.
2. Оброблення відеопотоку таким чином, щоб на виході отримувати карту глибини зображення.
3. Виведення оброблених зображень.

Алгоритм роботи модуля комп'ютерного зору представлено на рис. 2.1.

2.1. Імплементація *OpenCV* у програмному додатку

Проект *OpenCV* має бібліотеку *opencv-python*, що дозволяє використовувати можливості обробки зображень з мовою програмування *python*.

Python – інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять її корисною для швидкої розробки програм, а також як засіб поєднання наявних компонентів. *Python* підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор *Python* та стандартні бібліотеки доступні як у скомпільований, так і у вихідній формі на всіх основних платформах. В мові програмування *Python* підтримується кілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована.

Серед основних її переваг можна назвати такі:

– чистий синтаксис (для виділення блоків слід використовувати відступи);

– переносність програм (що властиве більшості інтерпретованих мов);

– стандартний дистрибутив має велику кількість корисних модулів (включно з модулем для розробки графічного інтерфейсу);

– можливість використання *Python* в діалоговому режимі (дуже корисне для експериментування та розв'язання простих задач);

– стандартний дистрибутив має просте, але разом із тим досить потужне середовище розробки, яке зветься *IDLE* і яке написане мовою *Python*;

– зручний для розв'язання математичних проблем (має засоби роботи з комплексними числами, може оперувати з цілими числами довільної величини, у діалоговому режимі може використовуватися як потужний калькулятор);

– відкритий код (можливість редагувати його іншими користувачами).

Python має ефективні структури даних високого рівня та простий, але ефективний підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис *Python*, динамічна обробка типів, а також те, що це інтерпретована мова, роблять її ідеальною для написання скриптів та швидкої розробки прикладних програм у багатьох галузях на більшості платформ.

Інтерпретатор мови *Python* і багата Стандартна бібліотека (як вихідні тексти, так і бінарні дистрибутиви для всіх основних операційних систем) можуть бути отримані з сайту *Python www.python.org*, і можуть вільно розповсюджуватися. Цей самий сайт має дистрибутиви та посилання на численні модулі, програми, утиліти та додаткову документацію.

Інтерпретатор мови *Python* може бути розширений функціями та типами даних, розробленими на *C* чи *C++* (або на іншій мові, яку можна викликати із *C*). *Python* також зручна як мова розширення для прикладних програм, що потребують подальшого налагодження.

Для використання *OpenCV* з *python* необхідно завантажити модуль *opencv-python* командою *sudo pip3 install opencv-python.* Після цього у початку файлу можна імпортувати модуль *cv2* командою *import cv2* та використовувати його для отримання, обробки, виводу зображень.

Наприклад, цей модуль буде використовуватися для виводу наборів цифр у вигляді зображення та його фарбування у форматі *RGB* для кращого сприйняття.

Рис. 2.1. Алгоритм роботи модуля комп’ютерного зору

2.2. Бібліотека *DepthNet*

DepthNet – бібліотека з відкритим вихідним кодом для виведення глибини на основі монокулярних відеозаписів, заснована на наборі даних для навігації, який імітує аерофотознімки зі стабілізованої монокулярної камери. Бібліотека у собі використовує згорткову нейронну мережу. Бібліотеку було протестовано як на модульованих сценах, так і на реальних даних польоту БПЛА. Хоча з реальними даними точність роботи дещо нижча.

Для тренування моделі *DepthNet* було використано датасет *StillBox*. *StillBox* – це синтетичний набір даних із глибиною, побудований за допомогою *Blender*,

що містить жорсткі об’єкти зі стабілізованими зображеннями *[9]*. Він націлений на імітацію польоту безпілотного літака із дуже неоднорідною складовою сцени із випадковими текстурами та розмірами. Таким чином, глибину дуже важко отримати виключно з контексту, а алгоритми глибини, засновані на структурі від руху, орієнтовані на надійність, повинні отримати кращу перевагу, ніж алгоритми на основі одного кадру. Приклад подано на рис 2.2.

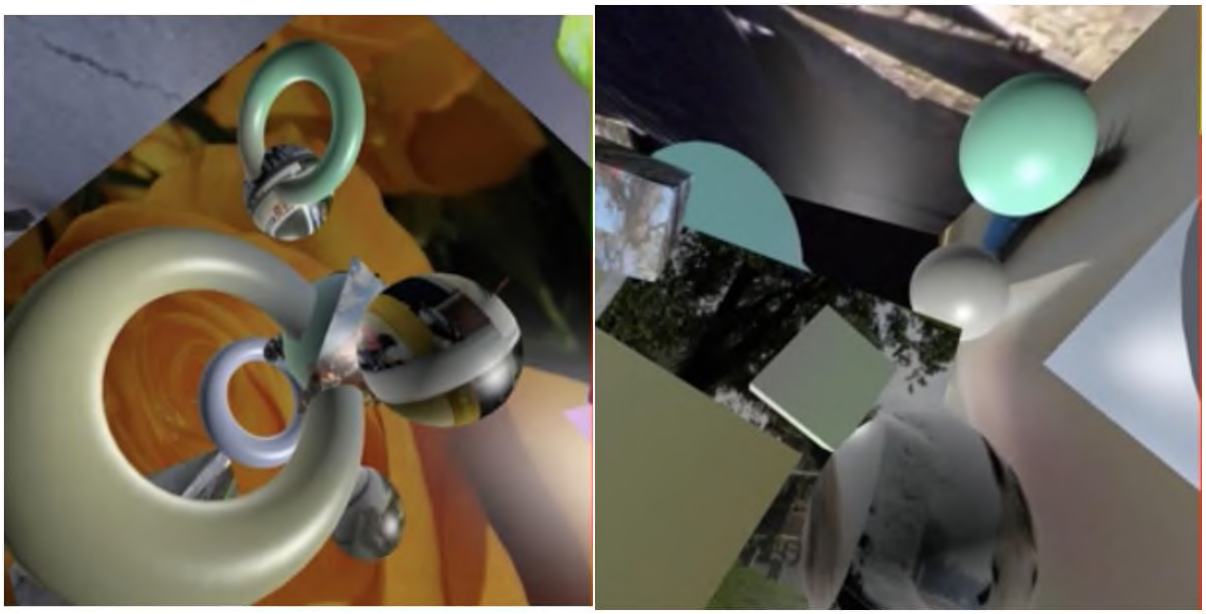

Рис. 2.2. Приклад датасету *StillBox*

Після обробки тестових зображень нейронною мережею *DepthNet* буде отримано набір зображень, що будуть, з деякою точністю, репрезентувати глибину зображення. Приклад наведено на рис. 2.3.

Проаналізувавши лише одне зображення неможливо встановити відстань до від камери до об’єкту, тому що розмір об’єкту невідомий. Тому *DepthNet* аналізує 10 послідовних зображень з відепотоку та порівнює результати їх аналізу. Це дозволяє отримати дані про зміну розміру об’єкту в полі зору камери при наближені. Крім того під час запуску роботи нейронної мережі користувач може надати їй значення коефіцієнту зсуву між кадрами. Ця змінна буде впливати на результати аналізу нейронної мережі. Коригуючи цей коефіцієнт, можна на ходу змінювати точність роботи модуля комп’ютерного зору.

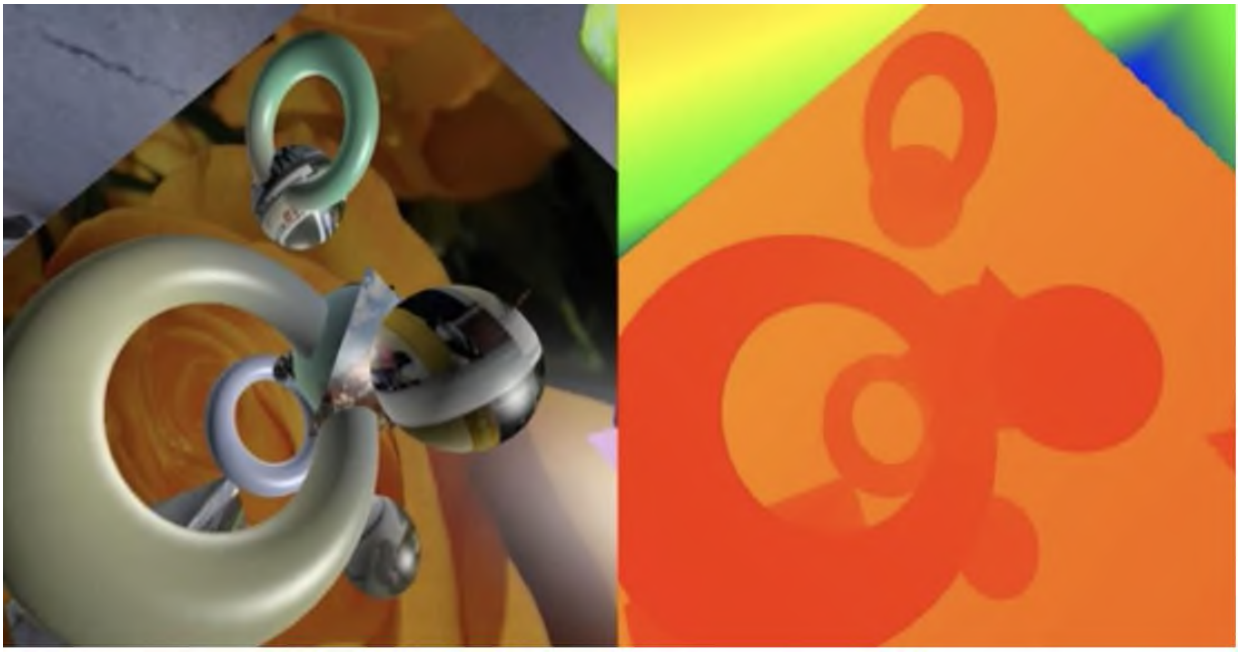

Рис. 2.3. Приклад виводу нейронної мережі *DepthNet*

2.3. Побудова моделі та тренування *DepthNet*

Метою тренування нейронної мережі є навчити її виділяти глибину в зображені ґрунтуючись на різниці між двома кадрами відео. Необхідно побудувати згорткову нейронну модель. Було використано бібліотеку *TensorFlow*, а також фреймворк *Keras*.

Імпортування необхідних модулів та бібліотек:

import keras.nn as nn

from models.utils import conv, deconv, predict_depth, post_process_depth, adaptative_cat, init_modules

Конструктор класу *class DepthNet(nn.Module),* що успадковується від модуля нейронної мережі фреймворку *Keras*:

def __init__(self, batch_norm=False, with_confidence=False, clamp=False, depth_activation=None):

super(DepthNet, self).__init__()

У якості функції активації буде використано лямбда функцію *elu* з фреймворку *Keras*. Це варіація функції *ReLU*, що є передавальною функцією, яка визначена таким чином:

де x – вхідне значення нейрона.

ELU – експоненціально-лінійна *ReLU* робить середнє значення передавача ближчим до нуля, що прискорює навчання *[*10*]*. Було візначено, що *ELU* може отримати більш високу точність класифікації, ніж *ReLU*. Визначається ця функція наступним чином:

self.depth_activation = lambda x: nn.functional.elu(x) + 1

Нейронна мережа буде складатися з 10 згорткових шарів, та чотирьох розгорткових шарів, а також 5 шарів, що будуть прогнозувати глибину та 4 шарів, що підготовляють прогноз до виводу:

self.conv1 = conv(6, 32, stride=2, batch_norm=batch_norm)
self.conv2 = conv(32, 64, stride=2, batch_norm=batch_norm)
self.conv3 = conv(64, 128, stride=2, batch_norm=batch_norm)
self.conv3_1 = conv(128, 128, batch_norm=batch_norm)
self.conv4 = conv(128, 256, stride=2, batch_norm=batch_norm)
self.conv4_1 = conv(256, 256, batch_norm=batch_norm)
self.conv5 = conv(256, 256, stride=2, batch_norm=batch_norm)
self.conv5_1 = conv(256, 256, batch_norm=batch_norm)
self.conv6 = conv(256, 512, stride=2, batch_norm=batch_norm)
self.conv6_1 = conv(512, 512, batch_norm=batch_norm)
self.deconv5 = deconv(512, 256, batch_norm=batch_norm)
self.deconv4 = deconv(513, 128, batch_norm=batch_norm)
self.deconv3 = deconv(385, 64, batch_norm=batch_norm)
self.deconv2 = deconv(193, 32, batch_norm=batch_norm)
self.predict_depth6 = predict_depth(512, with_confidence)
self.predict_depth5 = predict_depth(513, with_confidence)
self.predict_depth4 = predict_depth(385, with_confidence)
self.predict_depth3 = predict_depth(193, with_confidence)
self.predict_depth2 = predict_depth(97, with_confidence)

self.upsampled_depth6_to_5 = nn.ConvTranspose2d(1, 1, 4, 2, 1, bias=False)

self.upsampled_depth5_to_4 = nn.ConvTranspose2d(1, 1, 4, 2, 1, bias=False)

self.upsampled_depth4_to_3 = nn.ConvTranspose2d(1, 1, 4, 2, 1, bias=False)

self.upsampled_depth3_to_2 = nn.ConvTranspose2d(1, 1, 4, 2, 1, bias=False)

init_modules(self)

Структуру нейронної мережі *DepthNet* наведено на рис. 2.4.

Кожна згортка (крім модуля прогнозування глибини) супроводжується просторовою пакетною нормалізацією та шаром активації *ReLU*. Нормалізація партії допомагає конвергенції та стабільності під час тренування, нормалізуючи вихід згортки за партією декількох входів. Основна ідея цієї мережі полягає в тому, що пробні карти функцій поєднуються з відповідними попередніми результатами згортки. Тоді вища семантична інформація асоціюється з інформацією, тісніше пов'язаною з пікселями (оскільки вона пройшла через менш поетапні згортки), яка потім використовується для реконструкції. Важливо також зазначити, що мережі не потрібно повністю аналізувати структуру оптичного потоку, оскільки їй будуть надаватися пари кадрів із сцен з жорсткими об'єктами, що різко знижує розмірність проблеми, яка розглядається.

Можна помітити, що, хоча мережа все ще повністю згорнута, розміри карт функцій зменшуються до 1×1, а потім поводяться точно так само, як повністю зв'язаний шар, який може слугувати для неявного визначення напрямку руху та розповсюдження цієї інформації на виходах. Другим помітним фактом є те, що поблизу центру кадру мережа не має труднощів у виведенні глибини, що означає, що вона використовує нерівність сусідів та інтерполює, коли іншої інформації немає.

Це можна інтерпретувати як ідентифікацію фігур у форматі *3D* разом із їх збільшенням: пікселі, що належать до однієї форми, вважаються такими, що

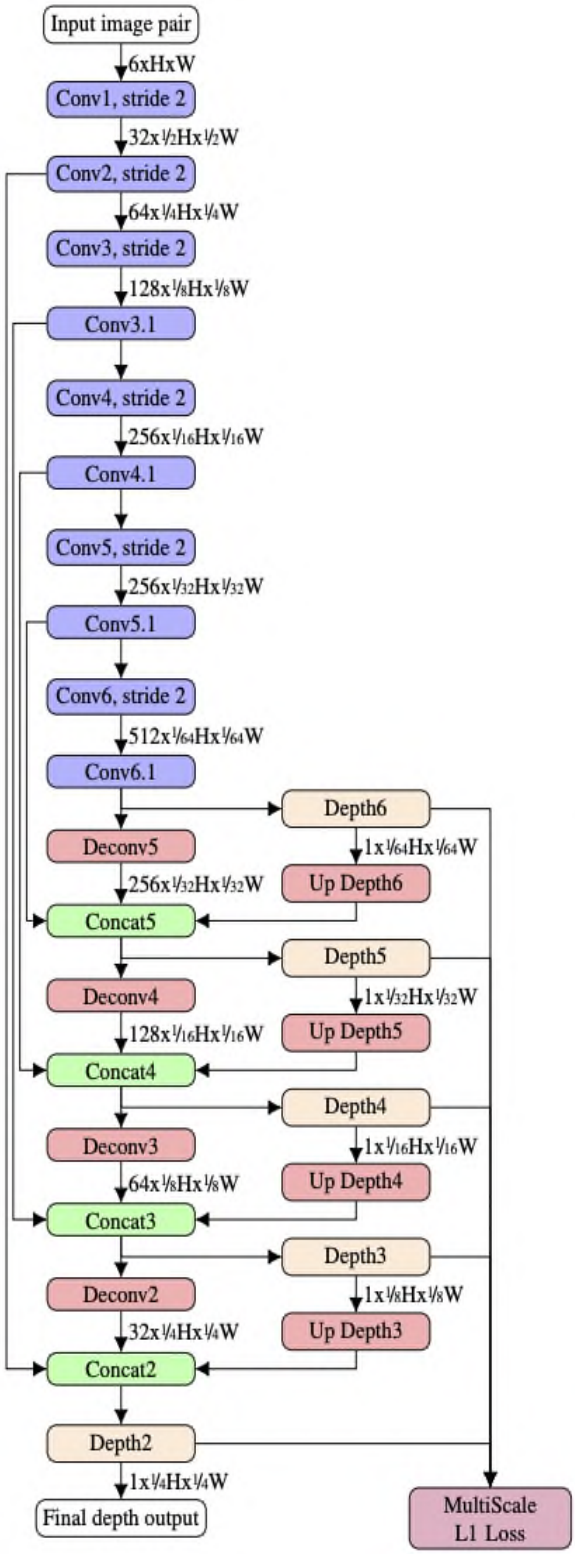

Рис. 2.4. Структура нейронної мережі *DepthNet*

мають близькі та безперервні значення глибини, що призводить до висновку про відстань розпізнаних об’єктів до БПЛА.

Для тренування моделі імпортуються наступні бібліотеки: бібліотека *Keras* для використання фреймворку бібліотеки *TensorFlow*; *Keras.backends.cudnn* для

тренування і використання нейронної мережі за допомогою графічного адаптера; *keras.optim* для використання алгоритмів оптимізації; *keras.utils.data* для ітерації через масиви даних, підготованих до використання нейронною мережею; *kerasvision.transforms* для використання функцій трансформування даних у необхідний для нейронної мережі вигляд; *co_transforms, models* та *datasets* необхідні для легшої передобробки даних до подання їх нейронній мережі; *depth_metric_reconstruction_loss* необхідна для виміру похибки при обчислені глибини під час роботи моделі нейронної мережі; *TermLogger* необхідна для виводу інформації у термінал; *SummaryWriter* необхідна для отримання результатів тренування нейронної мережі.

```
import keras
import keras.backends.cudnn as cudnn
import keras.optim
import keras.utils.data
import kerasvision.transforms as transforms
import co_transforms
import models
import datasets
from loss import depth_metric_reconstruction_loss as metric_loss
from terminal_logger import TermLogger
from tensorboardX import SummaryWriter
```

Модуль *TensorBoard* забезпечує візуалізацію та інструментарій, необхідні для експериментів із машинним навчанням. Він дозволяє:

– Відстеживати та візуалізувати такі показники, як втрата та точність;

– візуалізувати графік моделі (функції активації та шари);

– переглядати гістограми ваг, прихильності та інших тензорів, які змінюються з часом;

– відображати зображення, текст та звукові дані.

Якщо графічний адаптер комп’ютера дозволить – краще використовувати для тренування його, ніж процесор. Це може значно знизити час навчання:

device = keras.device("cuda") if keras.cuda.is_available() else keras.device("cpu")

Функція для тренування буде виглядати наступним чином:

def train(train_loader, model, optimizer, epoch_size, term_logger, train_writer):

global n_iter, args

Ініціалізація необхідних змінних. *batch_time* буде використано для виміру часу тренування однієї партії даних, *data_time* – для виміру часу тренування усього набору даних, *losses* – для виміру значення втрати нейронної мережі під час тренування, *depth2_metric_errors* та *depth2_normalized_errors* – для отримання значення похибки нейронної мережі, під час її навчання.

batch_time = AverageMeter()

data_time = AverageMeter()

losses = AverageMeter()

depth2_metric_errors = AverageMeter()

depth2_normalized_errors = AverageMeter()

Переключення моделі у режим тренування:

model.train()

end = time.time()

Вимір часу завантаження тренувальних даних:

for i, (input, target, _) in enumerate(train_loader):

data_time.update(time.time() - end)

target = target.to(device)

input = keras.cat(input,1).to(device)

Отримання виводу моделі:

output = model(input)

У контексті алгоритму оптимізації функція, яка використовується для оцінки рішення набору ваг, називається цільовою функцією. Під час тренування нейронної мережі необхідно визначити чи рішення набору ваг має буду максимальним або мінімальним *[11]*. Зазвичай приймається рішення намагатися мінімізувати похибку нейронної мережі. Таким чином, цільову функцію можна назвати функцією втрат, а значення, розраховане функцією втрати, називається просто «втратою».

Отримання значення втрати:

loss = metric_loss(output, target, weights=(0.32, 0.08, 0.02, 0.01, 0.005), loss=args.loss)

depth2_norm_error = metric_loss(output[0], target, normalize=True)

depth2_metric_error = metric_loss(output[0], target, normalize=False)

Запис значення втрати:

losses.update(loss.item(), target.size(0))

train_writer.add_scalar('train_loss', loss.item(), n_iter)

depth2_metric_errors.update(depth2_metric_error.item(), target.size(0))

depth2_normalized_errors.update(depth2_norm_error.item(), target.size(0))

Побудова графіку для відображення втрати:

optimizer.zero_grad()

loss.backward()

optimizer.step()

Вимірювання часу, що зайняло тренування:

batch_time.update(time.time() - end)

end = time.time()

Створення, або відкриття файлу, якщо він вже існує для запису результатів тренування:

with open(os.path.join(args.save_path, args.log_full), 'a') as csvfile:

```
writer = csv.writer(csvfile, delimiter='\t')
writer.writerow([loss.item(), depth2_metric_error.item()])
```

Вивід інформації про хід тренування у термінал:

```
term_logger.train_bar.update(i+1)
```

За допомогою бібліотеки *TermLogger* можна створювати полоси з інформацією про хід виконання роботи програмою у терміналі:

```
if i % args.print_freq == 0:
    term_logger.train_writer.write(
        'Train: Time {batch_time.val:.3f} ({batch_time.avg:.3f}) '
        'Data {data_time.val:.3f} ({data_time.avg:.3f}) '
        'Loss {loss.val:.4f} ({loss.avg:.4f}) '
        'Depth error {depth2_error.val:.3f} ({depth2_error.avg:.3f})\r'
        .format(batch_time=batch_time, data_time=data_time,
                loss=losses, depth2_error=depth2_metric_errors))
if i >= epoch_size - 1:
    break
n_iter += 1
```

Вивід результату роботи функції:

```
return losses.avg, depth2_metric_errors.avg, depth2_normalized_errors.avg
```

Функція валідації моделі після тренування:

```
def validate(val_loader, model, epoch, logger, output_writers=[]):
```

Ініціалізація необхідних змінних. *batch_time* буде використано для виміру часу тренування однієї партії даних, *depth2_metric_errors* та *depth2_norm_errors* – для отримання значення похибки нейронної мережі, під час її валідації.

```
batch_time = AverageMeter()
depth2_metric_errors = AverageMeter()
```

depth2_norm_errors = AverageMeter()

Переведення моделі у режим валідації:

model.eval()

end = time.time()

Завантаження даних для валідації:

for i, (input, target, _) in enumerate(val_loader):

target = target.to(device)

input = keras.cat(input, 1).to(device)

Отримання виводу моделі:

output = model(input)

Вивід перших трьох результатів партії даних:

if log_outputs and i < len(output_writers):

if epoch == 0:

output_writers[i].add_image('GroundTruth', util.tensor2array(target[0], max_value=100), 0)

output_writers[i].add_image('Inputs', util.tensor2array(input[0,:3]), 0)

output_writers[i].add_image('Inputs', util.tensor2array(input[0,3:]), 1)

output_writers[i].add_image('DepthNet Outputs', util.tensor2array(output[0], max_value=100), epoch)

depth2_norm_error = metric_loss(output, target, normalize=True)

depth2_metric_error = metric_loss(output, target, normalize=False)

Збереження похибки обчислення глибини:

depth2_norm_errors.update(depth2_norm_error.item(), target.size(0))

depth2_metric_errors.update(depth2_metric_error.item(), target.size(0))

Отримання часу затраченого на валідацію:

```
batch_time.update(time.time() - end)
end = time.time()
logger.test_bar.update(i+1)
```

Вивід інформації про хід валідації нейронної мережі за допомогою бібліотеки *TermLogger:*

```
if i % args.print_freq == 0:
   logger.test_writer.write(
      'Validation: '
      'Time {batch_time.val:.3f} ({batch_time.avg:.3f}) '
      'Depth error {depth2_error.val:.3f} ({depth2_error.avg:.3f})'
      .format(batch_time=batch_time,
            depth2_error=depth2_metric_errors))
```

Вивід функції:

```
return depth2_metric_errors.avg, depth2_norm_errors.avg
```

Основна функція для тренування моделі:

```
def main():
   global args, best_error, viz
   args = util.set_params(parser)
```

Ініціалізація необхідних змінних. *train_writer* та *val_writer* будуть використовиватись для виводу результатів тренування та валідації нейронної мережі:

```
train_writer = SummaryWriter(args.save_path/'train')
val_writer = SummaryWriter(args.save_path/'val')
output_writers = []
```

Якщо через командний рядок було задано аргумент для виводу логів під час тренування моделі – вони будуть виведені у окремий файл:

```
if args.log_output:
    for i in range(3):
        output_writers.append(SummaryWriter(args.save_path/'val'/str(i)))
keras.manual_seed(args.seed)
```

Ініціалізація зміних для нормалізаціх даних для тренування:

```
mean = [0.5, 0.5, 0.5]
std = [0.2, 0.2, 0.2]
```

Нормалізація даних для тренування

```
normalize = transforms.Normalize(mean=mean, std=std)
```

Нормалізація вхідних даних:

```
input_transform = transforms.Compose([
    co_transforms.ArrayToTensor(),
    transforms.Normalize(mean=[0, 0, 0], std=[255, 255, 255]),
    normalize
])
```

Нормалізація даних для валідації:

```
target_transform = transforms.Compose([
    co_transforms.Clip(0, 100),
    co_transforms.ArrayToTensor()
])
```

Нормалізація вихідних даних:

```
co_transform = co_transforms.Compose([
    co_transforms.RandomVerticalFlip(),
    co_transforms.RandomHorizontalFlip()
])
```

Вивід інформації про завантажженя тренувальних даних:

```
print("=> fetching scenes in '{}'".format(args.data))
train_set, val_set = datasets.still_box(
    args.data,
    transform=input_transform,
    target_transform=target_transform,
    co_transform=co_transform,
    split=args.split,
    seed=args.seed
)
```

Вивід інформації про дані для тренування:

```
print('{} samples found, {} train scenes and {} validation samples '.format(len(val_set)+len(train_set), len(train_set), len(val_set)))
```

Завантаження даних для тренування у модель:

```
train_loader = keras.utils.data.DataLoader(
    train_set, batch_size=args.batch_size, shuffle=True,
    num_workers=args.workers, pin_memory=True)
```

Завантаження даних для валідації моделі нейронної мережі:

```
val_loader = keras.utils.data.DataLoader(
    val_set, batch_size=args.batch_size,
    shuffle=False,
    num_workers=args.workers, pin_memory=True)
if args.epoch_size == 0:
    args.epoch_size = len(train_loader)
```

Завантажженя моделі нейронної мережі та створення її об'єкту, якщо було вказано використовувати вже натреновану модель:

```
if args.pretrained:
```

data = keras.load(args.pretrained)

assert(not data[‘with_confidence’])

model = models.DepthNet(batch_norm=data[‘bn’], clamp=args.clamp, depth_activation=args.activation_function)

model.load_state_dict(data[‘state_dict’])

Створення нової моделі нейронної мережі, якщо не було вказано використовувати вже натреновану:

else:

print("=> creating model ‘{}’".format(args.arch))

model = models.DepthNet(batch_norm=args.bn, clamp=args.clamp, depth_activation=args.activation_function)

Ініціалізація моделі:

model = model.to(device)

model = keras.nn.DataParallel(model)

cudnn.benchmark = True

Оптимізатори – це алгоритми або методи, що використовуються для зміни атрибутів нейронної мережі, таких як ваги та швидкість навчання, щоб зменшити втрату. Оптимізатори використовуються для вирішення задач оптимізації шляхом мінімізації функції. Оптимізатором визначається те, як слід змінити вагу або швидкість навчання нейронної мережі, щоб зменшити втрати *[12]*. Вага ініціалізується за допомогою деяких стратегій ініціалізації та оновлюється з кожною ітерацією навчання відповідно до рівняння оновлення. Протягом останніх декількох років досліджуються різні оптимізатори, кожен із яких має свої переваги та недоліки.

Градієнтний спуск (*GD*) – це найосновніший алгоритм оптимізації першого порядку, який залежить від похідної першого порядку функції втрат. Він обчислює, яким чином слід змінювати ваги, щоб функція могла досягти мінімуму. Завдяки зворотному розповсюдженню втрати передаються з одного шару на

інший, а параметри моделі, також відомі як ваги, змінюються залежно від втрат, щоб втрати можна було мінімізувати *[*13*]*. Цей алгоритм є дуже простим у реалізації, проте він одночасно приймає цілий набір даних з для обчислення похідної для оновлення ваг, що вимагає великих затрат пам'яті, мінімуми функції досягаються через тривалий час або взагалі ніколи не досягаються.

Алгоритм стохастичного градієнтного спуску (*SGD*) є розширенням алгоритму *GD* і долає деякі його недолік. Однім з недоліків *GD* є те, що йому потрібно багато пам'яті для завантаження всього набору даних одночасно. У випадку алгоритму *SGD* похідна обчислюється, беручи одну точку за раз. Потреба в пам'яті такого алгоритму менша порівняно з алгоритмом *GD*, оскільки похідна обчислюється, приймаючи лише 1 точку, проте через це час, необхідний для завершення 1 ітерації навчання росте, порівняно з алгоритмом *GD [*14*]*.

Для обох алгоритмів *GD* та *SGD* швидкість навчання залишається незмінною. Отже, ключовою ідеєю алгоритму *Adam* є встановлення адаптивного рівня навчання для кожної з ваг. Швидкість навчання ваги буде зменшуватися із збільшенням кількості ітерацій *[*15*]*.

У бібліотеці *Keras* реалізовані алгоритми оптимізації *Adam* та *SGD.*

Ініціалізація алгоритму оптимізації нейронної мережі, ґрунтуючись на переданих аргументах:

```
assert(args.solver in ['adam', 'sgd'])
print('=> setting {} solver'.format(args.solver))
if args.solver == 'adam':
```

Завантажженя алгоритму *Adam,* якщо було передано відповідний аргумент:

```
optimizer = keras.optim.Adam(model.parameters(), args.lr,
                             betas=(args.momentum, args.beta),
                             weight_decay=args.weight_decay)
```

Завантажженя алгоритму *SGD,* якщо було передано відповідний аргумент:

```
elif args.solver == 'sgd':
    optimizer = keras.optim.SGD(model.parameters(), args.lr,
```

momentum=args.momentum,

weight_decay=args.weight_decay,

dampening=args.momentum)

Створення файлів для виводу результатів тренування нейронної мережі:

with open(os.path.join(args.save_path, args.log_summary), 'w') as csvfile:

writer = csv.writer(csvfile, delimiter='\t')

writer.writerow(['train_loss', 'train_depth_error', 'normalized_train_depth_error', 'depth_error', 'normalized_depth_error'])

with open(os.path.join(args.save_path, args.log_full), 'w') as csvfile:

writer = csv.writer(csvfile, delimiter='\t')

writer.writerow(['train_loss', 'train_depth_error'])

Ініціалізація змінної для виводу інформації про ход тренування нейронної мережі у термінал:

term_logger = TermLogger(n_epochs=args.epochs, train_size =min(len(train_loader), args.epoch_size), test_size=len(val_loader))

term_logger.epoch_bar.start()

Ініціалізація валідатора нейронної мережі, якщо параметр валідації було передано як один з аргументів:

if args.evaluate:

depth_error, normalized = validate(val_loader, model, 0, term_logger, output_writers)

*term_logger.test_writer.write(' * Depth error : {:.3f}, normalized : {:.3f}'.format(depth_error, normalized))*

return

Тренування нейронної мережі:

for epoch in range(args.epochs):

term_logger.epoch_bar.update(epoch)

scheduler.step()

Оновлення інформації про тренування нейронної мережі у терміналі:

term_logger.reset_train_bar()

term_logger.train_bar.start()

Отримання значень втрати та похибки:

train_loss, train_error, train_normalized_error = train(train_loader, model, optimizer, args.epoch_size, term_logger, train_writer)

Вивід значень втрати та похибки у термінал:

*term_logger.train_writer.write(' * Avg Loss : {:.3f}, Avg Depth error : {:.3f}, normalized : {:.3f}'*

.format(train_loss, train_error, train_normalized_error))

train_writer.add_scalar('metric_error', train_error, epoch)

train_writer.add_scalar('metric_normalized_error', train_normalized_error, epoch)

Оновлення інформації про валідації нейронної мережі у терміналі:

term_logger.reset_test_bar()

term_logger.test_bar.start()

Отримання значення похибки нейронної мережі після валідації:

depth_error, normalized = validate(val_loader, model, epoch, term_logger, output_writers)

Вивід інформації про похибку після валідації у термінал:

*term_logger.test_writer.write(' * Depth error : {:.3f}, normalized : {:.3f}'.format(depth_error, normalized))*

val_writer.add_scalar('metric_error', depth_error, epoch)

```
        val_writer.add_scalar('metric_normalized_error', normalized, epoch)
        if best_error < 0:
            best_error = depth_error
```

Збереження найменшого значення похибки:

```
        is_best = depth_error < best_error
        best_error = min(depth_error, best_error)
        util.save_checkpoint(
            args.save_path, {
                'epoch': epoch + 1, 'arch': args.arch,
                'state_dict': model.state_dict(),
                'best_error': best_error,
                'bn': args.bn,
                'with_confidence': False,
                'activation_function': args.activation_function,
                'clamp': args.clamp,
                'mean': mean,
                'std': std
            },
            is_best)
```

Збереження найменшого значення похибки у файл:

```
        with open(os.path.join(args.save_path, args.log_summary), 'a') as csvfile:
            writer = csv.writer(csvfile, delimiter='\t')
            writer.writerow([train_loss, train_error, depth_error])
    term_logger.epoch_bar.finish()
```

2.4. Використання *DepthNet* для побудови карти глибини

Для використання тренованої моделі нейронної мережі необхідно завантажити наступні бібліотеки:

import keras

import PIL, skimage, imageio, matplotlib

import numpy as np

from imageio import imread, imwrite

from path import Path

import argparse

from tqdm import tqdm

import keras.nn.functional as F

from models import DepthNet

from util import tensor2array

Imageio – це бібліотека *Python*, яка забезпечує простий інтерфейс для читання та запису широкого спектру даних зображень, включаючи анімовані зображення, об'ємні дані та наукові формати. *Imageio* має відносно просте ядро, яке забезпечує загальний інтерфейс для різних форматів файлів. Це ядро дбає про читання з різних джерел (наприклад, *http*) та надає простий *API* для плагінів для доступу до вихідних даних. Усі формати файлів реалізовані в плагінах. Додаткові плагіни можна легко зареєструвати.

Numpy – розширення мови *Python*, що додає підтримку великих багатовимірних масивів і матриць, разом з великою бібліотекою високорівневих математичних функцій для операцій з цими масивами. Попередник *Numpy*, *Numeric*, був спочатку створений *Jim Hugunin*. *Numpy* – відкрите програмне забезпечення і має багато розробників. Оскільки *Python* – інтерпретована мова, математичні алгоритми, часто працюють в ньому набагато повільніше ніж у компільованих мовах, таких як *C* або навіть *Java*. *NumPy* намагається вирішити цю проблему для великої кількості обчислювальних алгоритмів забезпечуючи підтримку багатовимірних масивів і безліч функцій і операторів для роботи з ними. Таким чином будь-який алгоритм, який може бути виражений в основному як послідовність операцій над масивами і матрицями, працює так само швидко, як еквівалентний код, написаний на *C*. *NumPy* можна розглядати як гарну вільну альтернативу *MATLAB*, оскільки мова програмування *MATLAB* зовні нагадує

NumPy: обидві вони інтерпретовані, і обидві дозволяють користувачам писати швидкі програми поки більшість операцій проводяться над масивами або матрицями, а не над скалярами. Перевага *MATLAB* у великій кількості доступних додаткових тулбоксів, включаючи такі як пакет *Simulink*. Основні пакети, що доповнюють *NumPy*, це: *SciPy* – бібліотека, що додає більше *MATLAB*-подібної функціональності; *Matplotlib* – пакет для створення графіки в стилі *MATLAB*. Внутрішньо як *MATLAB*, так і *NumPy* базується на бібліотеці *LAPACK*, призначеної для вирішення основних задач лінійної алгебри.

Запуск програми тренування буде виконано з командного рядку з викликом деяких аргументів. Аргументи необхідно проаналізувати. Вони будуть показувати який файл нейронної мережі необхідно використовувати для побудови карти глибини, звідки брати файли для обробки, куди зберігати вивід нейронної мережі, та інше.

Модуль *argparse* дозволяє легко писати зручні інтерфейси командного рядка. Програма визначає, які аргументи вона вимагає, і *argparse* зрозуміє, як проаналізувати їх із *sys.argv*. Модуль *argparse* також автоматично генерує повідомлення про довідку та використання та видає помилки, коли користувачі надають програмі недійсні аргументи. Заповнення *ArgumentParser* інформацією про аргументи програми здійснюється за допомогою викликів методу *add_argument*. Як правило, ці виклики говорять *ArgumentParser*, як взяти рядки в командному рядку і перетворити їх на об’єкти. Ця інформація зберігається та використовується при виклику *parse_args*.

parser = argparse.ArgumentParser(description=‘Inference script for DepthNet img must be with no rotation’, formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument("--output-disp", action=‘store_true’, help="save disparity img")

parser.add_argument("--output-depth", action=‘store_true’, help="save depth img")

parser.add_argument("--output-raw", action=‘store_true’, help="save raw numpy depth array")

Вибір, чи використовувати вже натреновану модель нейронної мережі, чи тренувати нову:

parser.add_argument("--pretrained", required=True, type=str, help="pretrained DepthNet path")

Вибір кількості кадрів з відеопотоку, між якими буде вирахувано зсув фокальної площини. З використанням цього значення буде аналізуватися швидкість руху БПЛА, та, відповідно, зсув найближчої та найдальшої точки між БПЛА та жорстким об’єктом:

parser.add_argument("--frame-shift", default=1, type=int, help="temporal shift between imgs of the pairs feeded to the network")

Явне ініціалізування висоти відеопотоку з командного рядку:

parser.add_argument("--img-height", default=512, type=int, help="Image height")

Явне ініціалізування ширини відеопотоку з командного рядку:

parser.add_argument("--img-width", default=512, type=int, help="Image width")

Вибір, чи варто змінювати розмір вхідних зображень для нейронної мережі:

parser.add_argument("--no-resize", action=‘store_true’, help="no resizing is done")

Вибір директорії для тренувальних даних:

parser.add_argument("--dataset-list", default=None, type=str, help="Dataset list file")

parser.add_argument("--dataset-dir", default=‘.’, type=str, help="Dataset directory")

Вибір директорії для виводу результатів роботи нейронної мережі:

parser.add_argument("--output-dir", default='output', type=str, help="Output directory")

parser.add_argument("--img-exts", default=['png', 'jpg', 'bmp'], nargs='', type=str, help="images extensions to glob")*

Основна функція для використання нейронної мережі:

def main():

Парсування аргументів:

args = parser.parse_args()

Використання процесора замість графічного адаптера для обчислень нейронної мережі:

```
device = keras.device("cpu")
if not(args.output_disp or args.output_depth):
    print('You must at least output one value !')
    return

weights = keras.load(args.pretrained, map_location='cpu')
```

Ініціалізація моделі нейронної мережі:

```
depth_net = DepthNet(batch_norm=weights['bn'],
                     depth_activation=weights['activation_function'],
                     clamp=weights['clamp']).to(device)
print("running inference with {} ...".format(weights['arch']))
depth_net.load_state_dict(weights['state_dict'])
depth_net.eval()
```

Ініціалізація шляху до директорії з даними для обробки та директорії для виводу результатів обробки:

dataset_dir = Path(args.dataset_dir)

output_dir = Path(args.output_dir)

output_dir.makedirs_p()

Завантаження файлів для обробки:

if args.dataset_list is not None:

with open(args.dataset_list, ‘r’) as f:

test_files = [dataset_dir/file for file in f.read().splitlines()]

else:

test_files = sorted(sum([dataset_dir.files(‘.{}’.format(ext)) for ext in args.img_exts], []))*

print(‘{} files to test’.format(len(test_files)))

Ітерація через файли для обробки:

for file1, file2 in tqdm(zip(test_files[:-args.frame_shift], test_files[args.frame_shift:])):

Ініціалізація двох файлів попереднього та наступного кадру відеопотоку. При чому інформація про пікселі зображення буде зберігатися у вигляді масиву *numpy*:

img1 = imread(file1).astype(np.float32)

img2 = imread(file2).astype(np.float32)

h,w,_ = img1.shape

assert(img1.shape == img2.shape), "img1 and img2 must be the same size"

if (not args.no_resize) and (h != args.img_height or w != args.img_width):

img1 = numpy.array(Image.fromarray(img1).resize(size=(args.img_height, args.img_width))).astype(np.float32)

img2 = numpy.array(Image.fromarray(img2).resize(size=(args.img_height, args.img_width))).astype(np.float32)

imgs = np.concatenate([np.transpose(img1, (2, 0, 1)), np.transpose(img2, (2, 0, 1))])

Надання зображень нейронній мережі:

tensor_imgs = keras.from_numpy(imgs).unsqueeze(0).to(device)

tensor_imgs = ((tensor_imgs/255 - 0.5)/0.2)

Вивід значення глибини:

output_depth = depth_net(tensor_imgs)

upscaled_output = F.interpolate(output_depth.unsqueeze(1), (h,w), mode='bilinear', align_corners=False)[0,0]

Вивід обробленого зображення у обраному вигляді:

if args.output_disp:

disp = 1/upscaled_output

*disp = (255*tensor2array(disp, max_value=None, colormap='bone')).astype(np.uint8)*

imwrite(output_dir/'{}_disp{}'.format('.'.join(file2.name.split('.')[:-1]), f".{file2.name.split('.')[-1]}"), disp.transpose(1,2,0))

if args.output_depth:

*depth = (255*tensor2array(upscaled_output, max_value=100, colormap='rainbow')).astype(np.uint8)*

print(f".{file2.name.split('.')[-1]}")

imwrite(output_dir/'{}_depth{}'.format('.'.join(file2.name.split('.')[:-1]), f".{file2.name.split('.')[-1]}"), depth.transpose(1,2,0))

if args.output_raw:

np.save(output_dir/'{}_depth.npy'.format(file2.namebase), output_depth.cpu())

На рис. 2.5 наведено приклад даних, використаних для тестування.

Рис. 2.5. Приклад даних, використаних для тестування нейронної мережі

Приклад виведення (результату роботи нейронної мережі) у відтінках сірого наведено на рис. 2.6.

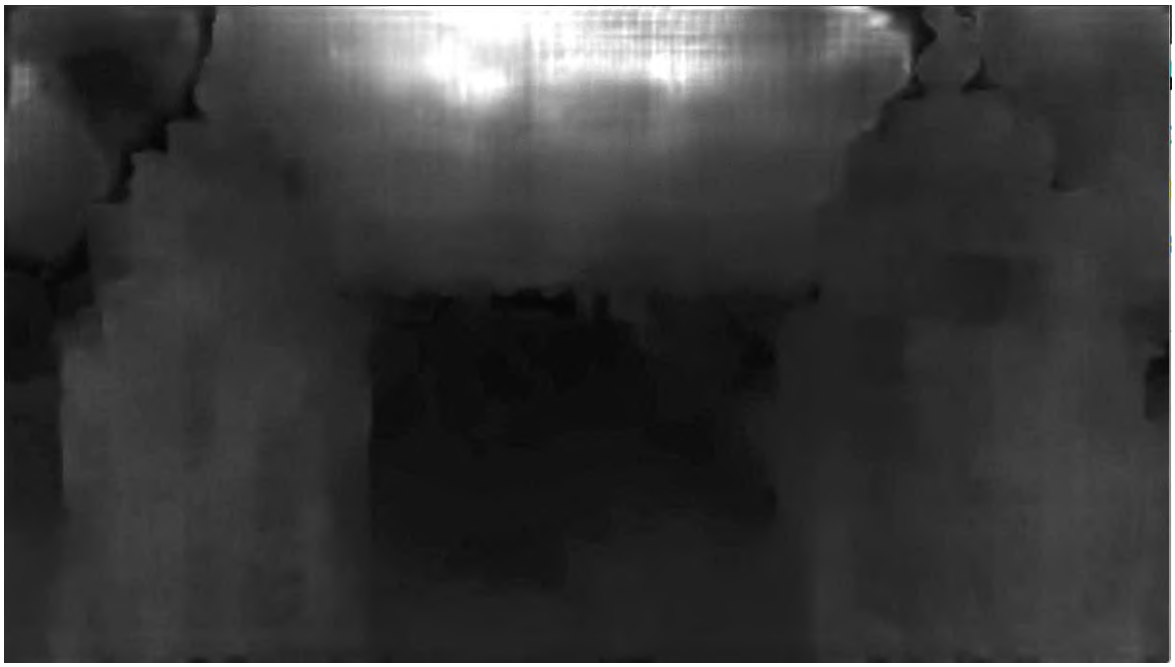

Рис. 2.6. Приклад результату роботи нейронної мережі у відтінках сірого

На рис. 2.7 наведено приклад кольорового виведення (результату роботи нейронної мережі).

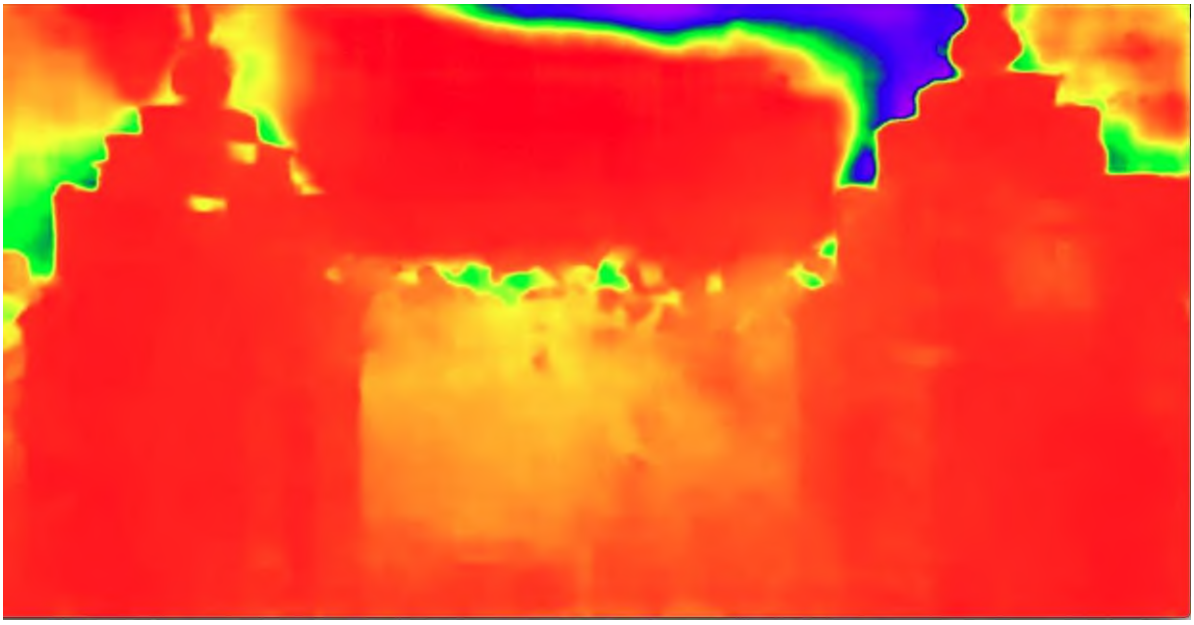

Рис. 2.7. Приклад кольорового результату роботи нейронної мережі

У кольоровому виведенні червоним кольором позначені найближчі об’єкти, фіолетовим – найдальші, помаранчевим – об’єкти на середній відстані. Як можна зауважити, нейронна мережа погано визначила відстань до хмар. Це можна пояснити тим, що вона була тренована на даних з яскравим освітленням. Необхідна подальша робота для покращення роботи нейронної мережі у похмуру погоду.

2.5. Висновки до розділу

В даному розділі була досліджена бібліотека *OpenCV,* та бібліотека *DepthNet,* що були використані для побудови модуля комп’ютерного зору. Було розроблено, та програмно реалізовано алгоритм роботи комп’ютерного зору. Програмно реалізовано тренування нейронної мережі з використанням бібліотеки *DepthNet* та датасету *StillBox.* Було проаналізовано результати тренування нейронної мережі на основі похибки кожної ітерації. Було розглянуто приклади роботи модуля комп’ютерного зору при різному освітленні.

Проведено аналіз коду, що дозволяє створити шар нейронної мережі. Розглянуто типи шарів.

Також розглянуто процес компіляції нейронної мережі, функцію втрат, яка використовується під час навчання нейронної мережі, оптимізатор та метрики, що використовуються під час збереження натренованої моделі.

РОЗДІЛ 3
ПОБУДОВА ПРОГРАМНОГО ЗАСОБУ НАВІГАЦІЇ БПЛА З ВИКОРИСТАННЯМ МОДУЛЯ КОМП'ЮТЕРНОГО ЗОРУ

Мета модуля навігації БПЛА полягає в коригуванні напрямку руху БПЛА ґрунтуючись на карті глибини, побудованій модулем комп'ютерного зору. Схему алгоритму роботи модуля навігації БПЛА подано на рис. 3.1. Основні елементи цього алгоритму наступні:

1. Отримання відеопотоку з БПЛА.
2. Передача відеопотоку модулю комп'ютерного зору.
3. Отримання з модуля комп'ютерного зору карти глибини.
4. Аналіз карти глибини.
5. Передача команди про зміну напрямку руху БПЛА, ґрунтуючись на результаті аналізу карти глибини.

3.1. Симуляція польоту дрону за допомогою *Unity*

Для симуляції польоту дрону та отримання відеопотоку з нього було використано ігровий рушій *Unity*. *Unity* – багатоплатформний інструмент для розробки дво- та тривимірних додатків та ігор, що працює на операційних системах *Windows* і *OS X*. Є можливість створювати інтернет-додатки за допомогою спеціального під'єднуваного модуля для браузера *Unity*, а також за допомогою експериментальної реалізації в межах модуля *Adobe Flash Player*.

Технічні характеристики *Unity*:

– Ігрова логіка пишеться за допомогою *C#*, раніше також була можливість використовувати *Boo* та *JavaScript*, але розробники відмовились від їх підтримки.

– Ігровий рушій повністю пов'язаний із середовищем розробки. Це дозволяє випробовувати гру прямо в редакторі.

– Робота з ресурсами можлива через звичайний *Drag&Drop*.

– Підтримка імпортування великої кількості форматів файлів.

– Вбудований генератор ландшафтів.

– Вбудована підтримка мережі.

– Інтерфейс редактора дуже гнучкий, є можливість писати свої вікна редактора та різноманітні розширення для нього.

– Існує рішення для спільної розробки — *Asset Server*. Також можна використовувати зручний для користувача спосіб контролю версій. Наприклад, *SVN* або *Source Gear*.

Рис. 3.1. Схема алгоритму роботи модуля навігації БПЛА

Додатки, створені за допомогою *Unity*, підтримують *DirectX* та *OpenGL*. Редактор *Unity* має простий *Drag & Drop* інтерфейс, який легко налаштовувати, що складається з різних вікон, завдяки чому можна проводити налагодження гри прямо в редакторі *[*16*]*. Рушій підтримує три сценарних мови: *C#, JavaScript* (модифікація). Проект в *Unity* ділиться на сцени (рівні) – окремі файли, що містять свої ігрові світи зі своїм набором об'єктів, сценаріїв, і налаштувань. Сцени можуть містити в собі як, об'єкти (моделі), так і порожні ігрові об'єкти – тобто ті, які не мають моделі. Об'єкти, в свою чергу містять набори компонентів, з якими і взаємодіють скрипти. Також у них є назва (в *Unity* допускається наявність двох і більше об'єктів з однаковими назвами), може бути тег (мітка) і шар, на якому він повинен відображатися. Так, у будь-якого предмета на сцені обов'язково присутній компонент *Transform* – він зберігає в собі координати місця розташування, повороту і розмірів по всіх трьох осях. У об'єктів з видимою геометрією також за умовчанням присутній компонент *Mesh Renderer*, що робить модель видимою (рис. 3.2).

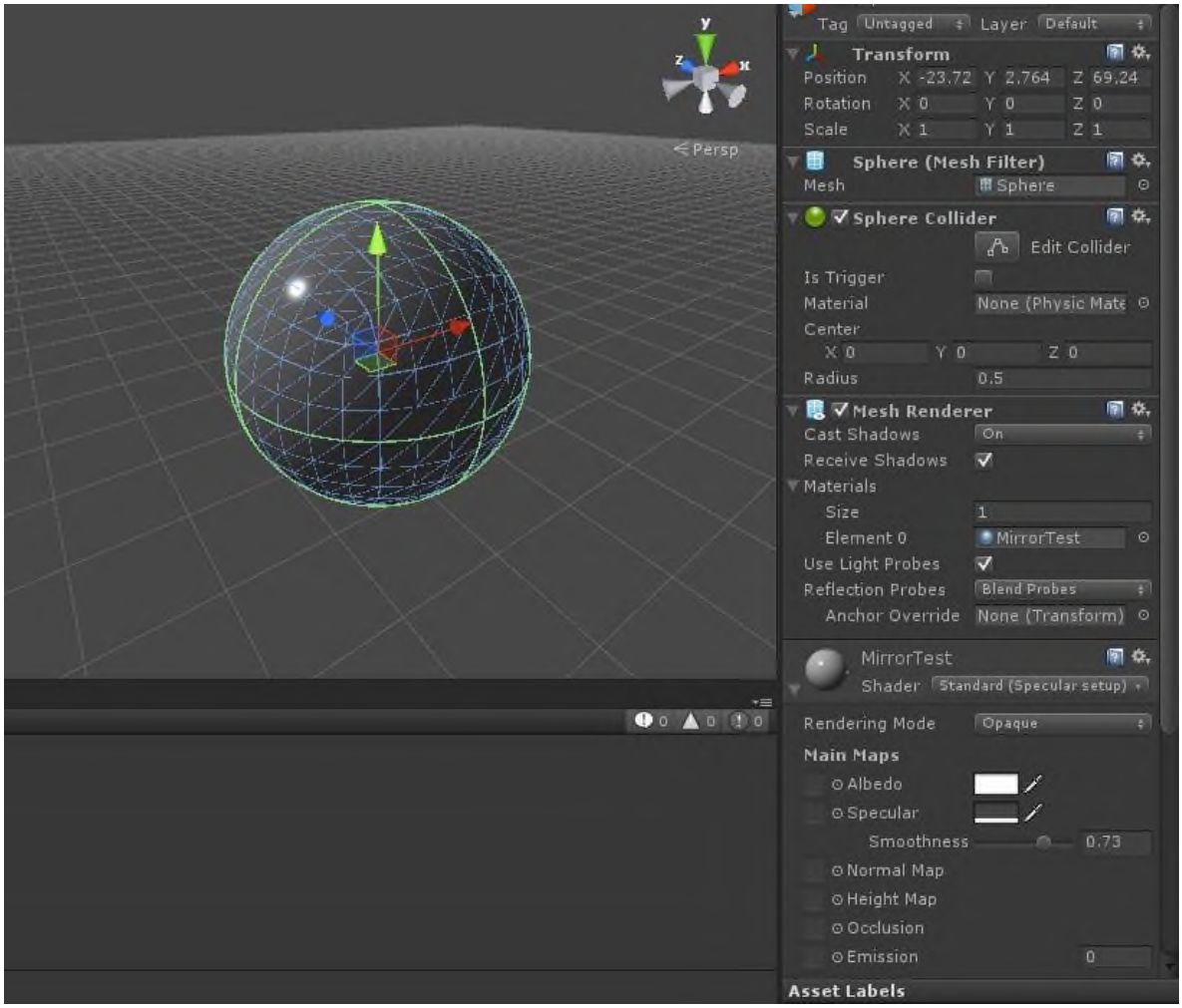

Рис. 3.2. Приклад відображення зеркальної сфери у *Unity*

Також *Unity* підтримує фізику твердих тіл і тканини, фізику типу *Ragdoll* (ганчіркова лялька) *[17]*. У редакторі є система успадкування об’єктів; дочірні об’єкти будуть повторювати всі зміни позиції, повороту і масштабу батьківського об’єкта. Скрипти в редакторі прикріплюються до об’єктів у вигляді окремих компонентів.

При імпорті текстури в рушій можна згенерувати *alpha*-канал, *mip*-рівні, *normal-map*, *light-map*, карту відображень, проте безпосередньо на модель текстуру прикріпити не можна — буде створено матеріал, з яким буде призначений шейдер, і потім матеріал прикріпиться до моделі. Редактор *Unity* підтримує написання і редагування шейдерів *[18]*. Крім того він містить компонент для створення анімації, яку також можна створити попередньо в *3D*-редакторі та імпортувати разом з моделлю, а потім розбити на файли.

Після створення у *Unity* нового проекту буде відкрито вікно сцени (рис. 3.3).

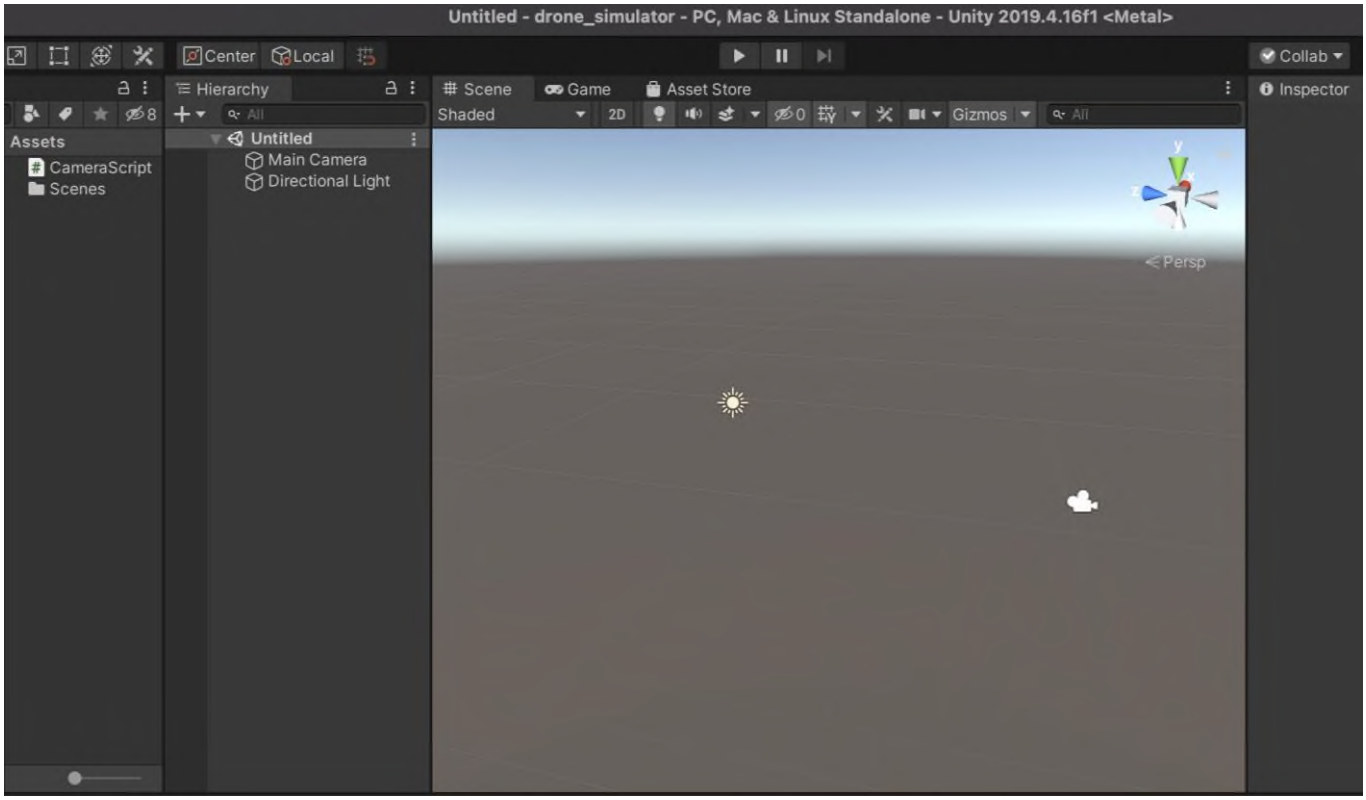

Рис. 3.3. Приклад нової сцени у *Unity*

Сцени містять середовища та меню гри. Під час стандартної розробки гри кожен унікальний файл сцени виконує роль унікального рівня. У кожній сцені розміщується середовище проекту, об’єкти, камери, освітлення та перешкоди.

Графічний рушій використовує *DirectX* (*Windows*), *OpenGL* (*Mac, Windows, Linux*), *OpenGL ES* (*Android, iOS*), та спеціальне власне *API* для *Wii*. Також підтримуються *bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion* (*SSAO*), динамічні тіні з використанням *shadow maps, render-to-texture* та повноекранні ефекти *post-processing*. *Unity* підтримує файли 3*ds Max, Maya, Softimage, Blender, modo, ZBrush, Cinema* 4*D, Cheetah*3*D, Adobe Photoshop, Adobe Fireworks* та *Allegorithmic Substance*. В ігровий проект *Unity* можна імпортувати об’єкти цих програм та робити налаштовування за допомогою графічного інтерфейсу. Для написання шейдерів використовується *ShaderLab*, що підтримує шейдерні програми написані на *GLSL* або *Cg*. Шейдер може включати декілька варіантів реалізації, що дозволяє *Unity* визначати найкращий варіант для конкретної відеокарти. *Unity* також має вбудовану підтримку фізичного рушія *Nvidia PhysX* (колишнього *Ageia*), підтримку симуляції одягу в системі реального часу на довільній та прив’язаній полігональній сітці (починаючи з *Unity* 3.0), підтримку системи *ray casts* та шарів зіткнення.

Скриптова система ігрового рушія зроблена на *Mono* — вільний відкритий проект з реалізації *.NET Framework*. Програмісти можуть використовувати *UnityScript* (власна скриптова мова, подібна до *JavaScript* та *ECMAScript*), *C#* або *Boo* (мова програмування, подібна до *Python*). Починаючи з версії 3.0, до *Unity* входить перероблена версія *MonoDevelop* для зневадження скриптів.

З виходом версії 5.2 передбачається вбудована можливість редагувати скрипти у середовищі *Visual Studio*

Для побудови симуляції було створено просту сцену (рис. 3.4), на якій було розміщено декілька площин, що повинні виступати у ролі землі, та декілька кубів, що будуть виступати у ролі перешкод для БПЛА. У якості самого БПЛА було використано основну камеру сцени, що допоможе отримувати відеопотік наче з камери БПЛА.

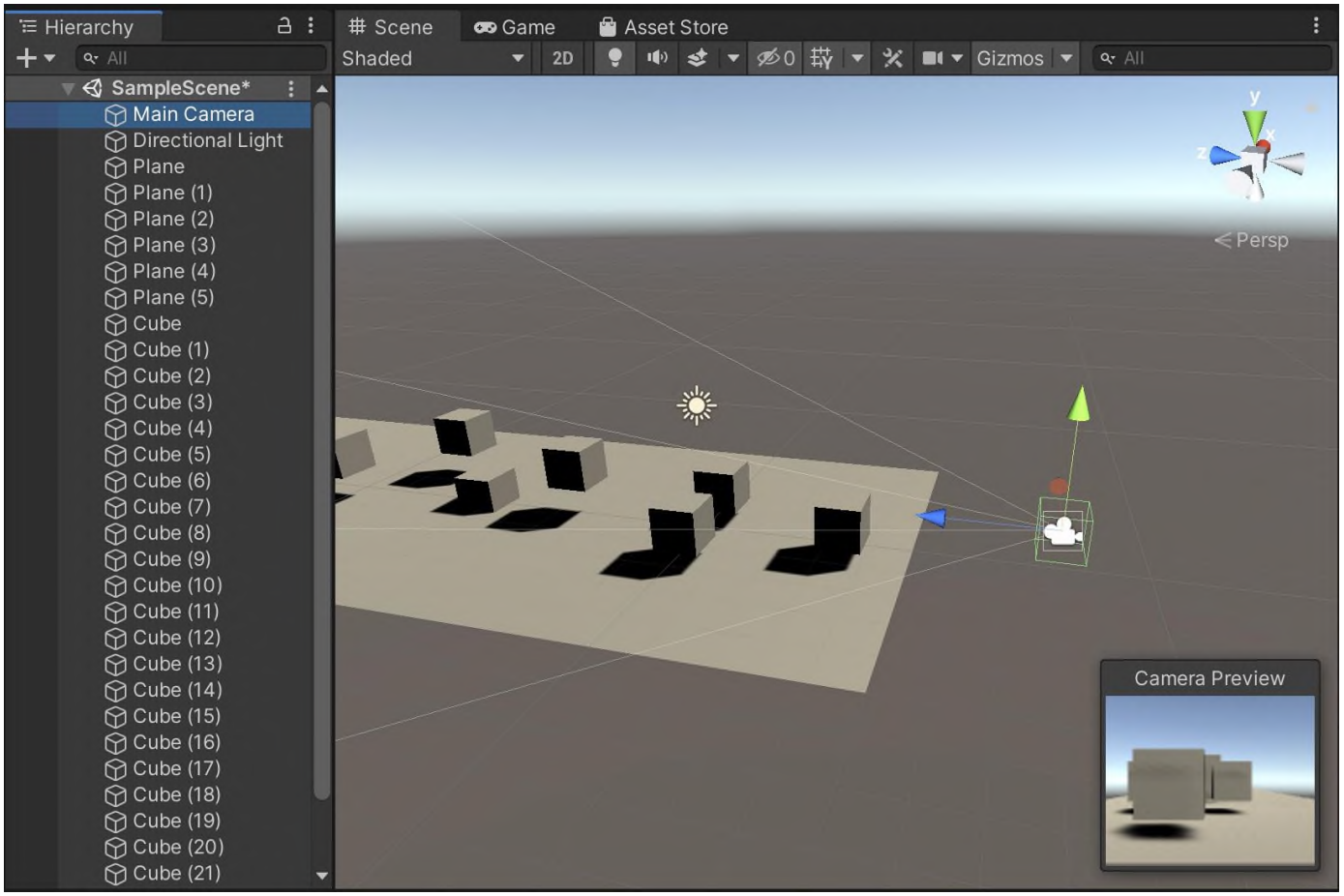

Рис. 3.4. Сцена у *Unity*

Для руху камери уперед було написано простий скрипт, що зміщує камеру на 2 пікселя уперед кожен кадр:

```
transform.Translate(Vector3.forward * sppedMultiplier * Time.deltaTime);
```

Якщо камера зіткнеться з одним з кубів вона повинна зупинитися та вивести повідомлення у консоль:

```
private void OnTriggerEnter(Collider collider)
    {
        Debug.Log("Fail");
        sppedMultiplier = 0;
    }
```

Також було написано скрипт для отримання відеопотоку з камери. Він повинен робити знімок того, що бачить камера кожен кадр та зберігати його.

Основний клас для скрипта головної камери проекту:

public class CameraScript : MonoBehaviour

{

Ініціалізації необхідних зміних:

public float sppedMultiplier = 2f;

public Vector3 startPos;

private Camera myCamera;

private static ScreenshotHandler instance;

private int scrrenShotCount = 0;

*Vector*3 – використовується для представлення 3*D*-векторів і точок. Ця структура використовується в *Unity* для передачі 3*D*-позицій та напрямків. Цей клас також містить функції для виконання загальних векторних операцій. Окрім *Vector*3 для маніпулювання векторами та точками також можна використовувати інші класи. Наприклад, класи *Quaternion* і *Matrix*4*x*4 корисні для обертання або перетворення векторів і точок. Структура *Vector*3 забезпечує підтримку апаратного прискорення. Для перетворень матриць екземпляри *Vector*2, *Vector*3 та *Vector*4 представляються у вигляді рядків: вектор *v* перетворюється матрицею *M* із множенням *vM*. Приклад проектування об’єкту *Vector3* на сцену в *Unity* наведено на рис. 3.5.

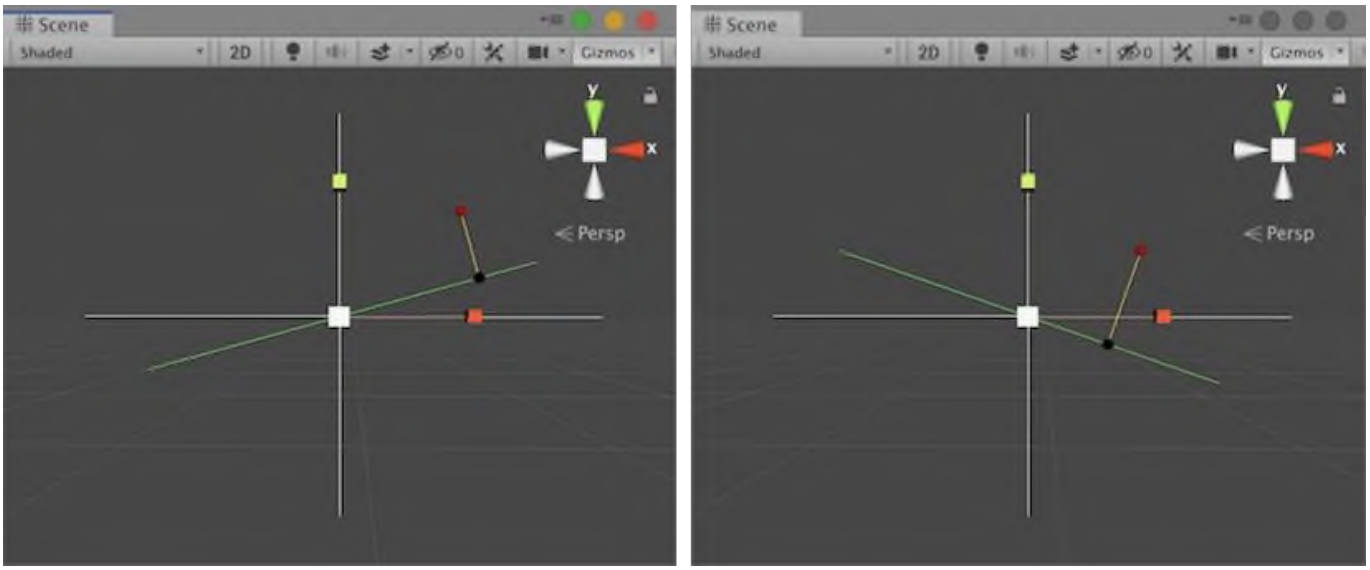

Рис. 3.5. Приклад проектування об’єкту *Vector3* на сцену в *Unity*

Об’єкт *Camera* – це пристрій, за допомогою якого користувач буде дивитися на своє оточення. Точка простору екрана визначається у пікселях. У нижньому лівому куті екрана (0,0); справа вгорі (*pixelWidth, pixelHeight*). Точка простору області перегляду нормалізована відносно об’єкту камери. Позиція *z* знаходиться у світових одиницях від Камери.

Функція *Start* буде виконуватися тільки раз на початку виконання програми. Вона повинна лише присвоїти компонент камери змінній:

```
void Start()
{
    myCamera = gameObject.GetComponent<MainCamera>();
}
```

Функція *Update* буде виконуватися кожен кадр:

```
void Update()
{
```

Переміщення камери:

```
    transform.Translate(Vector3.forward * sppedMultiplier * Time.deltaTime);
```

Створення знімку того, що бачить головна камера:

```
    ScreenshotHandler.TakeScreenshot_Static(Screen.width, Screen.height);
}
```

Функція рендеру області бачення головної камери, та збереження зображення:

```
    private void OnPostRender()
{
    RenderTexture renderTexture = myCamera.targetTexture;
    Texture2D renderResult = new Texture2D(renderTexture.width, renderTexture.height, TextureFormat.ARGB32, false);
```

Rect rect = new Rect(0, 0, renderTexture.width, renderTexture.height);

Спочатку зображення буде отримано як масив цифрових значень пікселей:

renderResult.ReadPixels(rect, 0, 0);

Масив цифрових значень пікселей буде преобразовано у масив байтів у форматі *png:*

byte[] byteArray = renderResult.EncodeToPng();

Запис масиву байтів у новий файл:

System.IO.File.WriteAllBytes("~/Personal/photo/screenshot" + scrrenShotCount.toString() + ".png", byteArray);

RenderTexture.ReleaseTemporary(renderTexture);

myCamera.targetTexture = null;

}

}

Приклад отриманих зображень наведено на рис. 3.6.

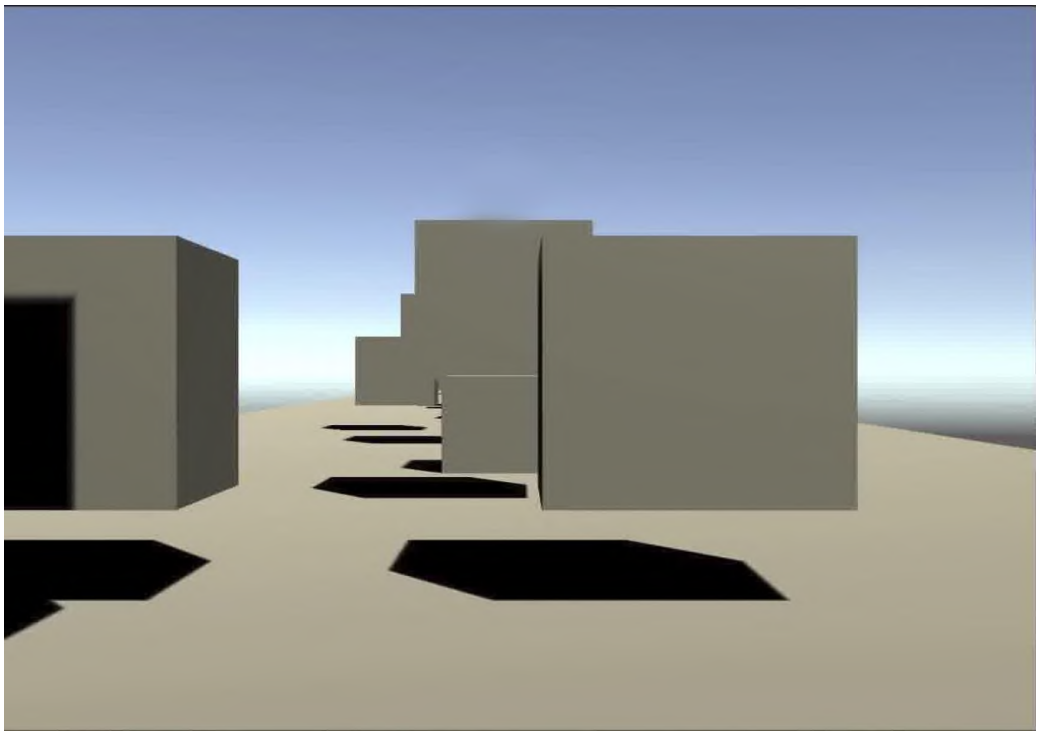

Рис. 3.6. Приклад зображень отриманих під час виконання проекту в *Unity*

3.2. Отримання карти глибини

Після збереження зображень отриманих з головної камери проекту у *Unity* зображення будуть оброблені нейронною мережею *DepthNet.* Один з процесів у фоні буде перевіряти, чи було збережено новий файл у директорію, та надавати його моделі нейронної мережі:

Ініціалізація лічильника кількості файлів:

files_count = 0

Процес буде перебувати у нескінченному циклі:

while true:

directory = '~/Personal/photos/'

Обчислення кількості файлів у директорії:

number_of_files = len([name for name in os.listdir(directory) if os.path.isfile(os.path.join(directory, name))])

Якщо кількість файлів збільшена – будуть обрані останні 2 файла у директорії для обробки:

if files_count < number_of_files:

files_count = number_of_files

test_files = sorted(sum([dataset_dir.files('.{}'.format(ext)) for ext in args.img_exts], []))*

files_array = tqdm(zip(test_files[:-args.frame_shift], test_files[args.frame_shift:]))

file1 = files_array[:-1]

file2 = files_array[len(files_array):]

Файли будуть зчитані для надання нейронній мережі:

img1 = imread(file1).astype(np.float32)

img2 = imread(file2).astype(np.float32)

h,w,_ = img1.shape

if (not args.no_resize) and (h != args.img_height or w != args.img_width):

Зображення будуть конвертовані у багатовимірний масив пікселів:

img1 = numpy.array(Image.fromarray(img1).resize(size=(args.img_height, args.img_width))).astype(np.float32)

img2 = numpy.array(Image.fromarray(img2).resize(size=(args.img_height, args.img_width))).astype(np.float32)

imgs = np.concatenate([np.transpose(img1, (2, 0, 1)), np.transpose(img2, (2, 0, 1))])

Нормалізація масивів пікселів для нейронної мережі:

tensor_imgs = keras.from_numpy(imgs).unsqueeze(0).to(device)

tensor_imgs = ((tensor_imgs/255 - 0.5)/0.2)

output_depth = depth_net(tensor_imgs)

upscaled_output = F.interpolate(output_depth.unsqueeze(1), (h,w), mode='bilinear', align_corners=False)[0,0]

Отримання результатів роботи нейронної мережі:

*depth = (255*tensor2array(upscaled_output, max_value=100, colormap='rainbow')).astype(np.uint8)*

print(f".{file2.name.split('.')[-1]}")

Виведення результатів роботи нейронної мережі у якості файлів в нову директорію:

imwrite(output_dir/'{}_depth{}'.format('.'.join(file2.name.split('.')[:-1]), f".{file2.name.split('.')[-1]}"), depth.transpose(1,2,0))

У цей час, кожен кадр під час виконання проекту у *Unity* буде зчитуватись інформація з останнього файлу у директорії результатів роботи моделі нейронної мережі, якщо кількість файлів було змінено:

Ініціалізація лічильника кількості файлів:

```
private int filesCount = 0;
private Texture2D ReadLastFile()
  {
```

Отримання всіх файлів з директорії з результатами роботи нейронної мережі:

```
var files = System.IO.Directory.GetFiles("~/Personal/result/");
```

Якщо кількість файлів збільшилась – останній файл буде взято з директорії та конвертовано у *2D* текстуру для подальшого аналізу:

```
if(files.Length() > filesCount) {
  filesCount = files.Length();
  file = files[files.Length() - 1];
  byte[] byteArray = System.IO.File.ReadAllBytes(file);
  return Texture2D.LoadImage(byteArray);
  }
}
```

3.3. Аналіз карти глибини та надання команд БПЛА

Аналіз карти глибини (рис. 3.7) буде виконуватися грунтуючись на дев’яти зонах отриманого зображення (рис. 3.8). Для кожного сектора буде обчислено середню інтенсивність червоного кольору усіх пікселів у цьому секторі. Буде обрано сектор з найменшою середньою інтенсивністю, та подано сигнал БПЛА про зміну напрямку руху таким чином, щоб обраний сектор став середнім. При чому буде віддано перевагу зміну напрямку руху у горизонтальній площині ніж

вертикальній, а при зміні напрямку руху по вертикальній площині буде віддано перевагу зміні напрямку униз, ніж вверх, тому що інакше БПЛА буде кожен раз злітати в небо. Тобто якщо середній сектор (сектор 1) буде відображати, що від дрона до об'єкта ще достатньо відстані – безпілотний літальний апарат буде продовжувати рух у попередньо заданому напрямку. Якщо модуль навігації визначить, що дрон у поточному напрямку продовжувати рух не може, то курс буде змінено.

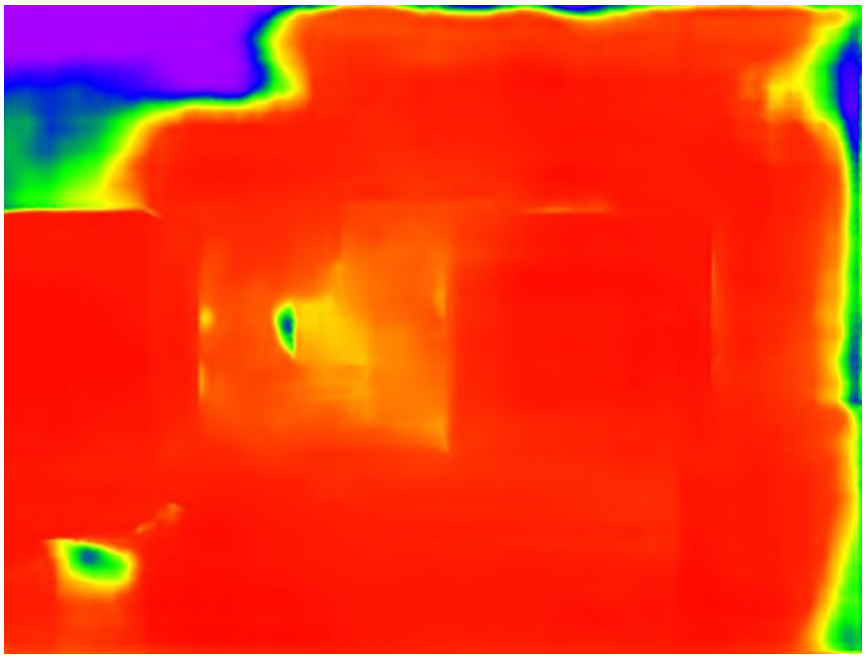

Рис. 3.7. Зображення отримане після обробки нейронною мережею

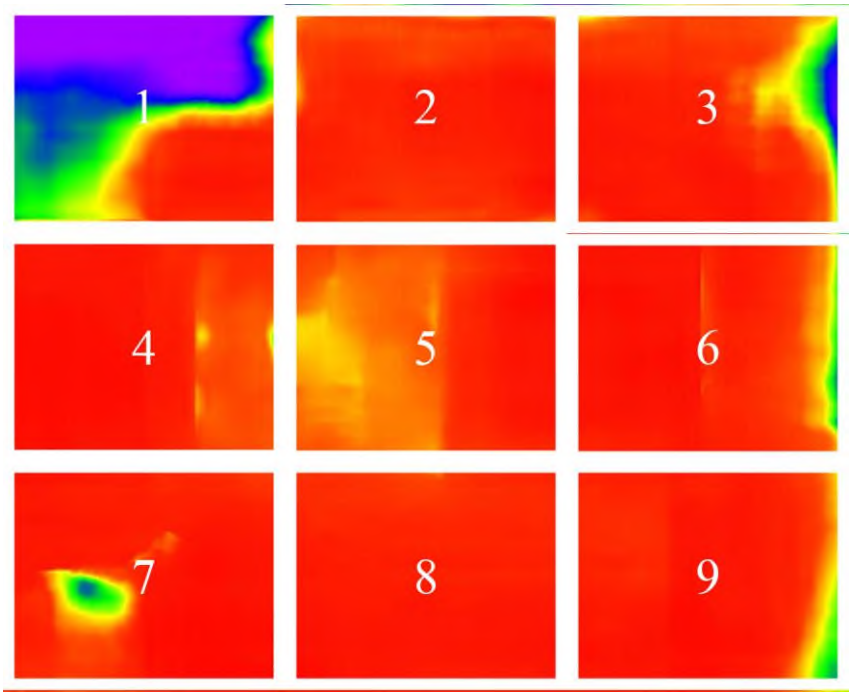

Рис. 3.8. Зображення після обробки нейронною мережею умовно поділене на дев'ять зон

Метод *AnalizeDepthMap* буде аналізувати карту глибини:

```
private int AnalizeDepthMap()
   {
```

Читання останнього файлу з директорії для результатів роботи нейронної мережі:

```
Texture2D image = ReadLastFile();
```

Ініціалізація необхідних змінних:

```
List<int> pixelList = new List<int>();
List<int> averagesList = new List<int>();
```

Вихідне зображення буде розміром 512×512 пікселів. Отримати дані про середнє значення інтенсивності червоного кольору можна у циклі, що буде ітеруватися від 0 до 600:

```
for (int i = 0; i < 600; i++)
{
   for (int j = 0; j < 600; j++)
   {
      if (j == 199 || j == 399 || j == 599)
      {
```

Якщо зовнішній цикл дійде до краю сектора – необхідно записати середнє значення інтенсивності червоного кольору у цьому секторі у список та перемістити ітератор у початок наступного сектора:

```
if (i == 199 || i == 399 || i == 599)
            {
               i = i - 199;
               j ++;
               averagesArray.Add(pixelList.Average());
               pixelList = new List<int>();
```

```
                break;
                }
                    j = j - 199;
                    i++;
                }
```

Запис значення інтенсивності червоного кольору поточного пікселя у список:

```
                    pixelList.Add(image.GetPixel(i, j)[0]);
                }
            }
```

Якщо середнє значення інтенсивності червоного кольору у середньому секторі менше 200 – значить до об’єкту ще достатньо відстані, більше ніж 5 метрів. У такому випадку БПЛА може продовжувати пересуватися у цьому напрямку:

```
            if (averagesList[4] < 200) {
                return 5;
```

Якщо середнє значення інтенсивності червоного кольору у секторах 3 або 6 менше за 150 можна змінювати напрямок БПЛА вліво або вправо. Напрямок буде обрано ґрунтуючись на мінімальному значенні інтенсивності червоного кольору у цих секторах. Ці напрямки будуть обрані з пріоритетом перед усіма іншими:

```
            } else if (averagesList[3] < 150 || averagesList[5] < 150) {
                int[] array = new int[] { averagesList[3], averagesList[5] };
                return array.AsSmartEnumerable().MinBy(entry => entry.Value).Index +
1;
```

Якщо середнє значення інтенсивності червоного кольору у секторах 6, 7 або 8 менше за 150 можна змінювати напрямок БПЛА вниз, вниз вправо або вниз вліво. Напрямок буде обрано ґрунтуючись на мінімальному значенні

інтенсивності червоного кольору у цих секторах. Ці напрямки будуть обрані з пріоритетом перед верхніми секторами (1, 2, 3):

```
} else if (averagesList[6] < 150 || averagesList[7] < 150 || averagesList[8] < 150) {
    int[] array = new int[] { averagesList[6], averagesList[7], averagesList[8] };
    return array.AsSmartEnumerable().MinBy(entry => entry.Value).Index + 1;
```

Якщо середнє значення інтенсивності червоного кольору у секторах 1, 2 або 3 менше за 150 можна змінювати напрямок БПЛА вгору, вгору вправо або вгору вліво. Напрямок буде обрано ґрунтуючись на мінімальному значенні інтенсивності червоного кольору у цих секторах. Ці напрямки обираються з найменшим пріоритетом:

```
} else if (averagesList[0] < 150 || averagesList[1] < 150 || averagesList[2] < 150) {
    int[] array = new int[] { averagesList[0], averagesList[1], averagesList[2] };
    return array.AsSmartEnumerable().MinBy(entry => entry.Value).Index + 1;
```

Якщо жоден з секторів не задовольняє умову його вибору – БПЛА краще зупинитись:

```
} else {
    return 0;
}   }
```

Метод *ChangeCourse* відповідає за зміну курсу БПЛА, ґрунтуючись на результатах аналізу інтенсивності червоного кольору у секторах результуючого зображення:

```
private void changeCourse()
   {
```

Отримання результату аналізу результуючої карти глибини:

```
int imageSectore = AnalizeDepthMap();
```

Ініціалізація масиву напрямків, що будуть обрані ґрунтуючись на результаті аналізу карти глибини:

```
String[,] directions = {
{‘stop’, ‘’}, {‘left’, ‘up’}, {‘up’, ‘’}, {‘right’, ‘up’},
{‘left’, ‘’}, {‘’, ‘’}, {‘right’, ‘’},
{‘left’, ‘down’}, {‘down’, ‘’}, {‘right’, ‘down’}
};
```

Якщо аналіз повернув напрямок *‘stop’* – БПЛА повинен зупинитись:

```
if (directions[imageSectore][0] == ‘stop’) {
   sppedMultiplier = 0;
```

Якщо аналіз повернув напрямок *‘’ (порожній рядок)* – БПЛА може продовжувати політ у поточному напрямку:

```
} else if (directions[imageSectore][0] == ‘’) {
   return;
```

Для інших напрямків БПЛА отримає команду про переміщення у відповідному напрямку:

```
} else {
   foreach (var direction in directions)
   {
      if (direction == ‘’) {
         continue;
      }
      transform.Translate(Vector3.GetMethod(direction).Invoke(null, null) *
sppedMultiplier * Time.deltaTime);
```

```
        }
    }
}
```

3.4. Використана апаратура

Було проведено тестування роботи модулей комп'ютерного зору та навігації БПЛА на двох ноутбуках: *Dell G3 15* та *MacBook Pro 2018.* Найважливішими пристроями для роботи розглянутих модулей є процесор та відеоадаптор. Також на швидкість обробки даних може впливати операційна система. Порівняння комплектації ноутбуків наведено у таблиці 3.1.

Таблиця 3.1.

Порівняльна таблиця комплектації ноутбуків, використаних для тестування розроблених програмних модулей

Ноутбук	Процесор	Відеоадаптор	ОС
Dell G3 15	*Intel Core i7 2.20GHz × 6*	*GeForce GTX 1050 Ti*	*Ubuntu 16.04 LTS*
MacBook Pro 2018	*Intel Core i5 1.40GHz × 4*	*Intel Iris Plus Graphics 1536 MB*	*MacOS Big Sur*

Для будь-якої нейронної мережі фаза навчання моделі є найбільш ресурсоємним завданням. Під час тренування нейронна мережа приймає вхідні дані, які потім обробляються прихованими шарами за допомогою ваг, які регулюються під час тренування, і модель потім повертає прогноз. Ваги коригуються, щоб знаходити закономірності, щоб робити кращі прогнози. Це ресурсоємне завдання, особливо враховуючи те, що на вхід нейронної мережі подається інформація про кожен піксель з відеопотоку

Відеоадаптер – це спеціалізований процесор із виділеною пам'яттю, який звичайно виконує операції з плаваючою комою, необхідні для рендерингу графіки. Іншими словами, це одночиповий процесор, що використовується для

великих графічних та математичних обчислень, що звільняє цикли процесора для інших завдань. Відеокарта зазвичай є платою розширення і вставляється у слот розширення, універсальний (*PCI-Express, PCI, ISA, VLB, EISA, MCA*) або спеціалізований (*AGP*). Проте відеокарта може бути і вбудованою у материнську плату як у вигляді окремого елемента, так і як складової частини північного мосту чипсету або центрального процесора. Відповідно вставлювана називається дискретною, а вбудована — інтегрованою. Сучасні відеокарти не обмежуються лише звичайним виведенням зображень, вони мають вбудований графічний мікропроцесор, котрий може здійснювати додаткову їх обробку, звільняючи від цих задач центральний процесор. Також процесор і відеокарта працюють разом і є залежними один від одного. Наприклад, усі сучасні відеокарти, що застосовують відеопроцесори *AMD/ATi* і *NVIDIA* підтримують *OpenGL* на апаратному рівні. Останнім часом, разом зі зростанням обчислювальних потужностей графічних процесорів має місце тенденція використовувати обчислювальні можливості графічного процесора для вирішення неграфічних задач (див. *OpenCL*).

Основна відмінність між графічними процесорами та центральними процесорами полягає в тому, що графічні процесори приділяють пропорційно більше транзисторів арифметичним логічним одиницям і менше кешам і контролю потоку в порівнянні з центральними процесорами.

До того ж компанія *Nvidia* розробляє бібліотеку для навчання нейронних мереж. *Cuda* (*cuDNN*) – це прискорена *GPU* бібліотека для глибоких нейронних мереж. *cuDNN* прискорює широко використовувані бібліотеки глибокого навчання, включаючи *Caffe2, Chainer, Keras, MATLAB, MxNet, PyTorch* та *TensorFlow*.

Первісна версія *CUDA SDK* була представлена 15 лютого 2007. В основі інтерфейсу програмування додатків *CUDA* лежить мова C з деякими розширеннями. Для успішної трансляції коду цією мовою до складу *CUDA SDK* входить власний C-компілятор командного рядка *nvcc* компанії *Nvidia*. Компілятор *nvcc* створений на основі відкритого компілятора *Open*64 і призначений для трансляції *host*-коду (головного, керуючого коду) і *device*-

коду (апаратного коду) (файлів з розширенням *.cu*) в об'єктні файли, придатні в процесі складання кінцевої програми або бібліотеки у середовищі програмування, наприклад, в *NetBeans*. В архітектурі *CUDA* використовується модель пам'яті *GRID*, кластерне моделювання потоків і *SIMD*-інструкції. Застосовна не тільки для високопродуктивних графічних обчислень, але і для різних наукових обчислень з використанням відеокарт *nVidia*. Учені і дослідники широко використовують *CUDA* в різних областях, включаючи астрофізику, обчислювальну біологію та хімію, моделювання динаміки рідин, електромагнітних взаємодій, комп'ютерну томографію, сейсмічний аналіз і багато іншого. У *CUDA* є можливість підключення до додатків, що використовують *OpenGL* і *Direct3D*. *CUDA* – кросплатформленість для таких операційних систем як *Linux, Mac OS X* і *Windows*.

У порівнянні з традиційним підходом до організації обчислень загального призначення допомогою можливостей графічних *API*, у архітектури *CUDA* відзначають наступні переваги в цій області:

– Інтерфейс програмування додатків *CUDA* (*CUDA API*) заснований на стандартній мові програмування Сі з деякими обмеженнями. На думку розробників, це повинно спростити і згладити процес вивчення архітектури *CUDA*.

– Колективна між потоками пам'ять (*shared memory*) розміром в 16 Кб може бути використана під організований користувачем кеш з більш широкою смугою пропускання, ніж при вибірці зі звичайних текстур.

– Більш ефективні транзакції між пам'яттю центрального процесора і відеопам'яттю.

– Повна апаратна підтримка цілочисельних і побітових операцій.

– Підтримка компіляції *GPU* коду засобами відкритого *LLVM*.

Але для використання *cuDnn* необхідно мати відеадаптер від *Nvidia* починаючи з версії більшої за *GeForce 410M [*19*]*. Тому, навіть без порівняння у часі, можна зробити висновок, що *Dell* набагато швидше виконає тренування нейронної мережі. Так з *MacBook* на тренування знадобилось 5 годин 13 хвилин, з *Dell* на тренування знадобилось 46 хвилин.

Проте використання натренованої моделі нейронної мережі не відрізнялось настільки сильно. В середньому на отримання карти глибини з одного кадру *MacBook* витрачав 1.13 секунди, у той час як *Dell* витрачав 0.87 секунди.

Модуль симуляції польоту безпілотного літального апарату виявилось неможливим розробляти з використанням *Ubuntu,* так як ігровий рушій *Unity* не оптимізований для роботи з *Linux* системами. Є можливість скомпілювати проект у *Unity* для *Linux* системи, але робота такого проекту буде дуже повільною, знову через недостатню оптимізацію *[20]*. Тому програмний засіб навігації БПЛА було протестовано лише на *MacBook*. У середньому час отримання модулем комп'ютерного зору відеопотоку з модуля симуляції польоту дрону, отримання карти глибини, аналіз її та коригування напрямку польоту займає близько 4 секунд. Швидкість польоту дрону потрібно корегувати відповідно.

До того ж з реальним дроном час від передачі відеопотоку до отримання інструкцій, щодо коригування напрямку руху буде ще дещо більшим, через те, що дрону необхідно буде передавати відеопотік через Інтернет на ноутбук, або сервер. Можна провести тести з використанням *Raspberry Pi,* але це, скоріш за все, не буде доцільним, оскільки апаратна частина *Raspberry Pi* не розрахована на використання моделей нейронної мережі. Найкращим способом зменшення часу від передачі відеопотоку до отримання інструкцій щодо коригування напрямку руху БПЛА буде використання комп'ютеру з відеоадаптером *Nvidia Geforce RTX 2080,* що має підтримку *cuDNN* версії 7.5, що зменшить час на обробку відеопотоку та отримання карти глибини, а також використання *4G* інтернету.

3.5. Висновки до розділу

У третьому розділі було розглянуто побудову програмного засобу, який виконує симуляцію польоту безпілотного літального апарату та виконує

обробку результатів роботи модуля комп’ютерного зору і відповідно викликає зміну напрямку руху БПЛА.

Розглянуто принцип роботи з ігровим рушієм *Unity* та спосіб побудови середовища для симуляції польоту БПЛА з перешкодами. Програмно реалізовано передачу відеопотоку з камери симульованого БПЛА у модуль комп’ютерного зору, побудованого за допомогою бібліотеки *DepthNet*, що використовує згорткову нейронну мережу.

Програмно реалізовано скрипт аналізу отриманої карти глибини та зміни напрямку руху БПЛА. Спроектовано та реалізовано алгоритм роботи програмного засобу навігації БПЛА.

ВИСНОВКИ

У дипломному проекті було розглянуто використання нейронних мереж для побудови модуля автономної навігації безпілотного літального апарату, що складається з модуля комп'ютерного зору та модуля навігації.

Було розглянуто поняття «нейронна мережа», архітектури простої та згорткової нейронної мережі. Розглянуто згорткові нейронні мережі – це клас глибинних штучних нейронних мереж прямого поширення, який успішно застосовувався до аналізу візуальних зображень. Згорткові мережі взяли за основу біологічний процес, а саме схему з'єднання нейронів зорової кори тварин. Окремі нейрони кори реагують на стимули лише в обмеженій області зорового поля, відомій як рецептивне поле. Рецептивні поля різних нейронів частково перекриваються таким чином, що вони покривають усе зорове поле.

Було розглянуто реалізацію комп'ютерного зору за допомогою бібліотеки *OpenCV*. *OpenCV* – бібліотека функцій та алгоритмів комп'ютерного зору, обробки зображень і чисельних алгоритмів загального призначення з відкритим кодом. Бібліотека надає засоби для обробки і аналізу вмісту зображень, у тому числі розпізнавання об'єктів на фотографіях, відстежування руху об'єктів, перетворення зображень, застосування методів машинного навчання і виявлення загальних елементів на різних зображеннях.

Було створено програмний засіб, що приймає в якості аргументу відеопотік з камери безпілотного літального апарату, та на його основі будує покадрову карту глибини, що відображає відстань від спостерігача до об'єкту. Для цього було використано бібліотеку *OpenCV* та бібліотеку *DepthNet,* що використовує згорткову нейронну мережу.

Створено проект за допомогою ігрового рушія *Unity* для симуляції польоту БПЛА з перешкодами. Було написано скрипти для передачі відеопотоку з головної камери проетку у *Unity* модулю комп'ютерного зору та аналізу результуючої карти глибини, на основі чого корегується політ БПЛА.

Практичним значенням виконаної розробки є можливість використання результатів роботи для подальшого покращення автономності літальних

апаратів. Можливими напрямками подальших досліджень є: генерування та використання більш широкого, різноманітного та реалістичного об’єму тренувальних даних для нейронної мережі у бібліотеці *DepthNet,* використання реального дрону для випробувань, використання стереокамери замість монокулярної камери.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. – К.: НАУ, 2017, 63 с.

2. ГОСТ 2.301-68. Единая система конструкторской документации. Форматы. – Введ. 2002–01–01. – М. : Вид.-во стандартов, 2006. – 27 с.

3. ДСТУ 3008-95. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення / Держстандарт України. – Вид. офіц. – *[*Чинний від 1995-02-23*]*. – Київ, 2007. – 86с.

4. *Bhadeshia H. K. D. H., Neural Networks in Materials Science,* 1999 – 28 *с.*

5. *Hitomi, E. E., Silva, J. V. and Ruppert, G. C., 3d scanning using rgbd imaging devices: A survey,* 2015 – 379 *с.*

6. *Kendall, A., Martirosyan, H., Dasgupta, S., Henry, P., Kennedy, R., Bachrach, A. and Bry, A., End-to-End Learning of Geometry and Context for Deep Stereo Regression,* 2017 – 238 *с.*

7. *Coombs, D., Herman, M., Hong, T.-H. and Nashman, M., Real-time obstacle avoidance using central flow divergence, and peripheral flow,* 1998 – 59 *с*.

8. *Saxena, A., Chung, S. H. and Ng, A. Y., Learning depth from single monocular images,* 2005 – 8 *с*.

9. *Hadsell, R., Sermanet, P., Ben, J., Erkan, A., Scoffier, M., Kavukcuoglu, K., Muller, U. and LeCun, Y., Learning long-range vision for autonomous off-road driving,* 2009 – 149 *с*.

10. *Tinghui Zhou, Matthew Brown, Noah Snavely, and David G. Lowe, Unsupervised learning of depth and ego-motion from video,* 2017 – 154 *с*.

11. *Wenjie Luo, Alexander G Schwing, and Raquel Urtasun, Efficient deep learning for stereo matching,* 2016 – 248 *с*.

12. *Ravi Garg, Vijay Kumar B. G, and Ian D. Reid, Unsupervised CNN for single view depth estimation: Geometry to the rescue,* 2016 – 173 *с*.

13. *Wei Zhang, Kazuyoshi Itoh, Jun Tanida, and Yoshiki Ichioka. Parallel distributed processing model with local space-invariant interconnections and its optical architecture // Applied Optics. 1990, Vol. 29, Issue 32. P. 4790 – 4846.*

14. *Bengio, Yoshua; Lamblin, Pascal; Popovici, Dan; Larochelle, Hugo, Greedy Layer-Wise Training of Deep Networks,* 2007 – 160 с.

15. *Kalchbrenner, Nal; Grefenstette, Edward; Blunsom, A Convolutional Neural Network for Modelling Sentences,* 1995 – 203 с.

16. *Chollet, François; Allaire, J. J., Deep Learning with R. Manning,* 2018 – 334 *с*.

17. *Rosenblatt. F., The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain. Psychological Review,* 1958 – 408 *с*.

18. *Drayton, Peter; Albahari, Ben; Neward, Ted, C# Language Pocket Reference,* 2002 – 413 *с*.

19. *Michels, J., Saxena, A. and Ng, A. Y., High speed obstacle avoidance using monocular vision and reinforcement learning //ICML '05: Proceedings of the 22nd international conference on Machine learning. – August 2005. – P. 593–600.*

20. *Pinard, C. and Chevalley, L. and Manzanera, A. and Filliat, D., End-to-end depth from motion with stabilized monocular videos,* 2017 – 74 *с*.

Додаток А

Уривок коду програми тренування нейронної мережі обробки зображення та побудови карти глибини

```
import argparse
import time
import csv
import os

import keras
import keras.backends.cudnn as cudnn
import keras.optim
import keras.utils.data
import kerasvision.transforms as transforms
import co_transforms
import models
import datasets
from loss import depth_metric_reconstruction_loss as metric_loss
from terminal_logger import TermLogger
from tensorboardX import SummaryWriter

import util
from util import AverageMeter


parser = argparse.ArgumentParser(description='Pykeras DepthNet Training on Still Box dataset')
util.set_arguments(parser)

best_error = -1
n_iter = 0
```

```
device = keras.device("cuda") if keras.cuda.is_available() else keras.device("cpu")


def main():
    global args, best_error, viz
    args = util.set_params(parser)

    train_writer = SummaryWriter(args.save_path/'train')
    val_writer = SummaryWriter(args.save_path/'val')
    output_writers = []
    if args.log_output:
        for i in range(3):
            output_writers.append(SummaryWriter(args.save_path/'val'/str(i)))
    keras.manual_seed(args.seed)

    # Data loading code
    mean = [0.5, 0.5, 0.5]
    std = [0.2, 0.2, 0.2]
    normalize = transforms.Normalize(mean=mean,
                                     std=std)
    input_transform = transforms.Compose([
        co_transforms.ArrayToTensor(),
        transforms.Normalize(mean=[0, 0, 0], std=[255, 255, 255]),
        normalize
    ])
    target_transform = transforms.Compose([
        co_transforms.Clip(0, 100),
        co_transforms.ArrayToTensor()
    ])
    co_transform = co_transforms.Compose([
        co_transforms.RandomVerticalFlip(),
        co_transforms.RandomHorizontalFlip()
```

```
    ])

    print("=> fetching scenes in '{}'".format(args.data))
    train_set, val_set = datasets.still_box(
        args.data,
        transform=input_transform,
        target_transform=target_transform,
        co_transform=co_transform,
        split=args.split,
        seed=args.seed
    )
    print('{} samples found, {} train scenes and {} validation samples
'.format(len(val_set)+len(train_set),
                                                        len(train_set),
                                                        len(val_set)))
    train_loader = keras.utils.data.DataLoader(
        train_set, batch_size=args.batch_size, shuffle=True,
        num_workers=args.workers, pin_memory=True)
    val_loader = keras.utils.data.DataLoader(
        val_set, batch_size=args.batch_size,
        shuffle=False,
        num_workers=args.workers, pin_memory=True)
    if args.epoch_size == 0:
        args.epoch_size = len(train_loader)
    # create model
    if args.pretrained:
        data = keras.load(args.pretrained)
        assert(not data['with_confidence'])
        print("=> using pre-trained model '{}'".format(data['arch']))
        model = models.DepthNet(batch_norm=data['bn'], clamp=args.clamp,
depth_activation=args.activation_function)
```

```
            model.load_state_dict(data['state_dict'])
        else:
            print("=> creating model '{}'".format(args.arch))
            model = models.DepthNet(batch_norm=args.bn, clamp=args.clamp,
depth_activation=args.activation_function)

        model = model.to(device)
        model = keras.nn.DataParallel(model)
        cudnn.benchmark = True

        assert(args.solver in ['adam', 'sgd'])
        print('=> setting {} solver'.format(args.solver))
        if args.solver == 'adam':
            optimizer = keras.optim.Adam(model.parameters(), args.lr,
                                         betas=(args.momentum, args.beta),
                                         weight_decay=args.weight_decay)
        elif args.solver == 'sgd':
            optimizer = keras.optim.SGD(model.parameters(), args.lr,
                                        momentum=args.momentum,
                                        weight_decay=args.weight_decay,
                                        dampening=args.momentum)

        scheduler = keras.optim.lr_scheduler.MultiStepLR(optimizer,
                                                         milestones=[19,30,44,53],
                                                         gamma=0.3)

        with open(os.path.join(args.save_path, args.log_summary), 'w') as csvfile:
            writer = csv.writer(csvfile, delimiter='\t')
            writer.writerow(['train_loss', 'train_depth_error',
'normalized_train_depth_error', 'depth_error', 'normalized_depth_error'])

        with open(os.path.join(args.save_path, args.log_full), 'w') as csvfile:
```

```
        writer = csv.writer(csvfile, delimiter='\t')
        writer.writerow(['train_loss', 'train_depth_error'])

    term_logger = TermLogger(n_epochs=args.epochs,
train_size=min(len(train_loader), args.epoch_size), test_size=len(val_loader))
    term_logger.epoch_bar.start()

    if args.evaluate:
        depth_error, normalized = validate(val_loader, model, 0, term_logger,
output_writers)
        term_logger.test_writer.write(' * Depth error : {:.3f}, normalized :
{:.3f}'.format(depth_error, normalized))
        return

    for epoch in range(args.epochs):
        term_logger.epoch_bar.update(epoch)
        scheduler.step()

        # train for one epoch
        term_logger.reset_train_bar()
        term_logger.train_bar.start()
        train_loss, train_error, train_normalized_error = train(train_loader, model,
optimizer, args.epoch_size, term_logger, train_writer)
        term_logger.train_writer.write(' * Avg Loss : {:.3f}, Avg Depth error : {:.3f},
normalized : {:.3f}'
                                       .format(train_loss, train_error, train_normalized_error))
        train_writer.add_scalar('metric_error', train_error, epoch)
        train_writer.add_scalar('metric_normalized_error', train_normalized_error,
epoch)

        # evaluate on validation set
        term_logger.reset_test_bar()
```

```
        term_logger.test_bar.start()
        depth_error, normalized = validate(val_loader, model, epoch, term_logger,
output_writers)
        term_logger.test_writer.write(' * Depth error : {:.3f}, normalized :
{:.3f}'.format(depth_error, normalized))
        val_writer.add_scalar('metric_error', depth_error, epoch)
        val_writer.add_scalar('metric_normalized_error', normalized, epoch)

        if best_error < 0:
            best_error = depth_error

        # remember lowest error and save checkpoint
        is_best = depth_error < best_error
        best_error = min(depth_error, best_error)
        util.save_checkpoint(
            args.save_path, {
                'epoch': epoch + 1,
                'arch': args.arch,
                'state_dict': model.state_dict(),
                'best_error': best_error,
                'bn': args.bn,
                'with_confidence': False,
                'activation_function': args.activation_function,
                'clamp': args.clamp,
                'mean': mean,
                'std': std
            },
            is_best)

        with open(os.path.join(args.save_path, args.log_summary), 'a') as csvfile:
            writer = csv.writer(csvfile, delimiter='\t')
            writer.writerow([train_loss, train_error, depth_error])
```

```
    term_logger.epoch_bar.finish()


def train(train_loader, model, optimizer, epoch_size, term_logger, train_writer):
    global n_iter, args
    batch_time = AverageMeter()
    data_time = AverageMeter()
    losses = AverageMeter()
    depth2_metric_errors = AverageMeter()
    depth2_normalized_errors = AverageMeter()

    # switch to train mode
    model.train()

    end = time.time()

    for i, (input, target, _) in enumerate(train_loader):
        # measure data loading time
        data_time.update(time.time() - end)
        target = target.to(device)
        input = keras.cat(input,1).to(device)

        output = model(input)

        loss = metric_loss(output, target, weights=(0.32, 0.08, 0.02, 0.01, 0.005),
loss=args.loss)
        depth2_norm_error = metric_loss(output[0], target, normalize=True)
        depth2_metric_error = metric_loss(output[0], target, normalize=False)
        # record loss and EPE
        losses.update(loss.item(), target.size(0))
        train_writer.add_scalar('train_loss', loss.item(), n_iter)
        depth2_metric_errors.update(depth2_metric_error.item(), target.size(0))
```

```
        depth2_normalized_errors.update(depth2_norm_error.item(), target.size(0))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        batch_time.update(time.time() - end)
        end = time.time()

        with open(os.path.join(args.save_path, args.log_full), 'a') as csvfile:
            writer = csv.writer(csvfile, delimiter='\t')
            writer.writerow([loss.item(), depth2_metric_error.item()])
        term_logger.train_bar.update(i+1)
        if i % args.print_freq == 0:
            term_logger.train_writer.write(
                'Train: Time {batch_time.val:.3f} ({batch_time.avg:.3f}) '
                'Data {data_time.val:.3f} ({data_time.avg:.3f}) '
                'Loss {loss.val:.4f} ({loss.avg:.4f}) '
                'Depth error {depth2_error.val:.3f} ({depth2_error.avg:.3f})\r'
                .format(batch_time=batch_time, data_time=data_time,
                        loss=losses, depth2_error=depth2_metric_errors))
        if i >= epoch_size - 1:
            break
        n_iter += 1

    return losses.avg, depth2_metric_errors.avg, depth2_normalized_errors.avg


@keras.no_grad()
def validate(val_loader, model, epoch, logger, output_writers=[]):
    batch_time = AverageMeter()
    depth2_metric_errors = AverageMeter()
```

```
        depth2_norm_errors = AverageMeter()
        log_outputs = len(output_writers) > 0
        # switch to evaluate mode
        model.eval()

        end = time.time()

        for i, (input, target, _) in enumerate(val_loader):
            target = target.to(device)
            input = keras.cat(input, 1).to(device)
            # compute output
            output = model(input)
            if log_outputs and i < len(output_writers):  # log first output of 3 first batches
                if epoch == 0:
                    output_writers[i].add_image('GroundTruth', util.tensor2array(target[0],
max_value=100), 0)
                    output_writers[i].add_image('Inputs', util.tensor2array(input[0,:3]), 0)
                    output_writers[i].add_image('Inputs', util.tensor2array(input[0,3:]), 1)
                output_writers[i].add_image('DepthNet Outputs',
util.tensor2array(output[0], max_value=100), epoch)
            depth2_norm_error = metric_loss(output, target, normalize=True)
            depth2_metric_error = metric_loss(output, target, normalize=False)
            # record depth error
            depth2_norm_errors.update(depth2_norm_error.item(), target.size(0))
            depth2_metric_errors.update(depth2_metric_error.item(), target.size(0))

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()
            logger.test_bar.update(i+1)
            if i % args.print_freq == 0:
                logger.test_writer.write(
```

```
            'Validation: '
            'Time {batch_time.val:.3f} ({batch_time.avg:.3f}) '
            'Depth error {depth2_error.val:.3f} ({depth2_error.avg:.3f})'
            .format(batch_time=batch_time,
                depth2_error=depth2_metric_errors))

    return depth2_metric_errors.avg, depth2_norm_errors.avg


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

Додаток Б

Уривок коду програми використання нейронної мережі *DepthNet* для отримання карти глибини

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraScript : MonoBehaviour
{
    public float sppedMultiplier = 2f;
    public Vector3 startPos;
    private Camera myCamera;
    private static ScreenshotHandler instance;
    private int scrrenShotCount = 0;
    private int filesCount = 0;

    void Start()
    {
        myCamera = gameObject.GetComponent<MainCamera>();
    }

    void Update()
    {
        transform.Translate(Vector3.forward * sppedMultiplier * Time.deltaTime);
        ScreenshotHandler.TakeScreenshot_Static(Screen.width, Screen.height);
    }

    private void OnTriggerEnter(Collider collider)
    {
        Debug.Log("Fail");
        sppedMultiplier = 0;
```

```
        }

        private void OnPostRender()
        {
            RenderTexture renderTexture = myCamera.targetTexture;
            Texture2D renderResult = new Texture2D(renderTexture.width,
renderTexture.height, TextureFormat.ARGB32, false);
            Rect rect = new Rect(0, 0, renderTexture.width, renderTexture.height);
            renderResult.ReadPixels(rect, 0, 0);

            byte[] byteArray = renderResult.EncodeToPng();
            System.IO.File.WriteAllBytes("~/Personal/photo/screenshot" +
scrrenShotCount.toString() + ".png", byteArray);
            RenderTexture.ReleaseTemporary(renderTexture);
            myCamera.targetTexture = null;
        }

        private void changeCourse()
        {
            int imageSectore = AnalizeDepthMap();
            String[,] directions = {
            {‘stop’, ‘’}, {‘left’, ‘up’}, {‘up’, ‘’}, {‘right’, ‘up’},
            {‘left’, ‘’}, {‘’, ‘’}, {‘right’, ‘’},
            {‘left’, ‘down’}, {‘down’, ‘’}, {‘right’, ‘down’}
            };
            if (directions[imageSectore][0] == ‘stop’) {
                sppedMultiplier = 0;
            } else if (directions[imageSectore][0] == ‘’) {
                return;
            } else {
                foreach (var direction in directions)
                {
```

```
                if (direction == '') {
                    continue;
                }
                transform.Translate(Vector3.GetMethod(direction).Invoke(null, null) *
sppedMultiplier * Time.deltaTime);
            }
        }
    }

    private int AnalizeDepthMap()
    {
        Texture2D image = ReadLastFile();
        List<int> pixelList = new List<int>();
        List<int> averagesList = new List<int>();
        for (int i = 0; i < 600; i++)
        {
            for (int j = 0; j < 600; j++)
            {
                if (i == 199 || i == 399 || i == 599)
                {
                    i = i - 199;
                    j ++;
                    averagesArray.Add(pixelList.Average());
                    pixelList = new List<int>();
                }
                if (j == 199 || j == 399 || j == 599)
                {
                    j = j - 199;
                    i++;
                }
                pixelList.Add(image.GetPixel(i, j)[0]);
```

```
            }
        }
        if (averagesList[4] < 200) {
            return 5;
        } else if (averagesList[3] < 150 || averagesList[5] < 150) {
            int[] array = new int[] { averagesList[3], averagesList[5] };
            return array.AsSmartEnumerable().MinBy(entry => entry.Value).Index + 1;
        } else if (averagesList[6] < 150 || averagesList[7] < 150 || averagesList[8] <
150) {
            int[] array = new int[] { averagesList[6], averagesList[7], averagesList[8]
};
            return array.AsSmartEnumerable().MinBy(entry => entry.Value).Index + 1;
        } else if (averagesList[0] < 150 || averagesList[1] < 150 || averagesList[2] <
150) {
            int[] array = new int[] { averagesList[0], averagesList[1], averagesList[2]
};
            return array.AsSmartEnumerable().MinBy(entry => entry.Value).Index + 1;
        } else {
            return 0;}}

    private Texture2D ReadLastFile()
    {
        var files = System.IO.Directory.GetFiles("~/Personal/result/");
        if(files.Length() > filesCount) {
            filesCount = files.Length();
            file = files[files.Length() - 1];
            byte[] byteArray = System.IO.File.ReadAllBytes(file);
            return Texture2D.LoadImage(byteArray);
        }
    }
    private void TakeScreenshot(int width, int height)
```

```
{
    myCamera.targetTexture = RenderTexture.GetTemporary(width, height, 16);
}

private void TakeScreenshot_Static(int width, int height)
{ instance.TakeScreenshot(width, height); }}
```