МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет Компютерних наук,
управління та адміністрування

Кафедра Інформаційних технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: Категоризація та клеймування зображень за допомогою API з використанням нейронної мережі TENSORFLOW

Виконав студент 2 курсу групи МІС-20
спеціальності 122 Комп'ютерні науки

Доскач Денис Володимирович

Керівник д.т.н., професор
Мещеряков Володимир Іванович

Рецензент д.т.н., професор
Казакова Надія Феліксівна

Одеса 2021

АНОТАЦІЯ

Тема магістерської роботи «Категоризація та клеймування зображень за допомогою API з використанням нейронної мережі TENSORFLOW».

Об’єкт дослідження – процес розробки системи керування та створення класифікаторів зображень на підставі динамічного контенту.

Метод роботи – генерація та опрацювання динамічних моделей побудованих на TensorFlow використовуючи платформу ML.NET.

Мета роботи – створення універсального програмного забезпечення, яке зможе автоматизувати процес створення класифікаторів на основі переданих файлів, розбитих на категорії.

Практична цінність магістерської роботи полягає в тому, що є можливість модульного вбудовування в систему нейронного клеймування зображень, завдяки зовнішньому API. Ця задача досить часто виникає, коли будуються платформи для автоматичного сортування контенту. Як один з прикладів – системи керування зображеннями. Наприклад, інтернет-магазини, сховища зображень, тощо. Використовуючи таку систему, можна налагодити комунікацію та отримання категорії зображень, та спростити внутрішні бізнес процеси.

Ключові слова: ДИНАМІЧНА МОДЕЛЬ, КЛЕЙМУВАННЯ ЗОБРАЖЕНЬ, МАСИВ ДАНИХ, МАШИННЕ НАВЧАННЯ, ПЛАТФОРМА, СИСТЕМА.

Магістерська кваліфікаційна робота містить 71 сторінки, 40 рисунків та 16 джерел.

SUMMARY

Theme of master's work " Categorization and Marking of Images by Means of the API using the TENSORFLOW Neural Network".

The object of research is the process of developing a control system and creating image classifiers based on dynamic content.

Method of work - generation and processing of dynamic models built on TensorFlow using the ML.NET platform.

The purpose of the work is to create universal software that can automate the process of creating classifiers based on transferred files, divided into categories.

The practical value of the master's thesis is that it is possible to modularly integrate into the system of neural image branding, thanks to an external API. This problem often arises when building platforms for automatic content sorting. One example is image management systems. For example, online stores, image repositories, etc. Using such a system, you can establish communication and image categories, and simplify internal business processes.

Keywords: DYNAMIC MODEL, IMAGING OF IMAGES, DATA ARRAYS, MACHINE LEARNING, PLATFORM, SYSTEM.

The master's thesis contains 71 pages, 40 figures and 16 sources.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – база даних

ОС – операційна система

ПЗ – програмне забезпечення

СУБД – сукупність програмних та лінгвістичних засобів загального або спеціального призначення, що забезпечують управління створенням та використанням баз даних

API – програмний інтерфейс програми

ASP.NET – технологія створення веб-застосунків і веб-сервісів

AutoML – процес автоматизації наскрізного процесу застосування машинного навчання до завдань реального світу

EF Core – проста, кросплатформова версія популярної технології доступу до даних Entity Framework з відкритим вихідним кодом, що розширюється.

ML.NET – крос-платформна та відкрита система машинного навчання для розробників

MS SQL – система управління базами даних

MySQL – система керування базами даних (СКБД) з відкритим кодом

NET Core – вільне та відкрите програмне забезпечення каркаса веб-застосунків, з продуктивністю вищою ніж у ASP.NET

NuGet – це вільна система керування пакунками, розроблена для Microsoft development platform

ORM – технологія програмування, яка пов'язує бази даних із концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних»

ReLU – функція активації при глибокому навчанні

ResNet – класична нейронна мережа, яка використовується як основа для багатьох завдань комп'ютерного зору

SQL – декларативна мова програмування для взаємодії користувача з ба-зами даних

UI – користувальницький інтерфейс

ВСТУП

Розпізнавання візуальних образів є одним із найважливіших компонентів систем управління та обробки інформації, автоматизованих систем та систем прийняття рішень.

Завдання, пов’язані з класифікацією та розпізнаванням об’єктів, явищ і сигналів, що характеризуються кінцевим набором деяких властивостей, з’являються в області робототехніки, пошуку інформації, моніторингу та аналізу візуальних даних, досліджень штучного інтелекту.

Алгоритмічна обробка та класифікація зображень використовуються в системах безпеки, контролю та управління доступом, системах відеоспостереження, системах віртуальної реальності, системах пошуку інформації. Системи розпізнавання рукописного тексту, номери транспортних засобів, відбитків пальців або обличчя широко використовуються у виробництві та використовуються в інтерфейсах програмних продуктів, системах безпеки та розпізнавання та ін.

Якщо глянути на сучасний рівень доступу до інформації, то можна виявити, що знайти потрібну інформацію без будь-яких спеціальних засобів майже не можливо. У багатьох випадках, щоб якось працювати з цими даними, доводиться спочатку їх класифікувати.

Кластеризація (або кластерний аналіз) – завдання розбиття безлічі об'єктів на групи, які називаються кластерами. Усередині кожного кластера повинні виявитися схожі об'єкти, а об'єкти різних груп повинні бути якомога іншими.

Класифікація – завдання полягає у присвоєнні кожному об'єкту із заданої множини, однієї із заздалегідь заданих груп. Цілі класифікації:

– Розуміння структури даних нового об'єкта. Присвоєння групи об'єкту явно визначає його властивості та структуру. Ці властивості можна врахувати за подальшої обробки.

– Передбачення даних. На основі відомих даних можна передбачати можливість приналежності об'єкта класифікації до якогось класу.

Відмінністю кластеризації від класифікації є те, що перелік груп чітко не заданий і визначається в процесі роботи алгоритму. Багато великих корпорацій, таких як Яндекс або Google для обробки даних, використовують класифікацію. Класифікація є досить простою, але при цьому часто вирішуваним завданням Data Mining.

Загальні характеристики кваліфікаційної роботи:

– повний обсяг сторінок пояснювальної записки – 71;
– кількість рисунків – 40;
– кількість посилань – 16.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПРОГРАМ АНАЛОГІВ

Один з основних підходів, що найбільш широко використовувався в області розпізнавання зображень, є застосуванням класичних моделей-класифікаторів, що навчаються з учителем. Для навчання таких моделей використовуються маркована вибірка даних, що складається з масиву зображень і відповідного їм масиву міток, що визначають категорію, до якої відноситься зображення [1][1)].

1.1 Аналіз предметної області

У процесі навчання масив даних поділяється на дві нерівні частини – навчальну вибірку і тестову вибірку, потім за допомогою специфічного для конкретного алгоритму правила навчання параметри моделі налаштовуються з використанням навчальної вибірки таким чином, щоб отримавши як вхідні дані зображення, модель на виході виробляла мітку відповідного класу. Цей метод представлений безліччю моделей, серед яких найбільш широко використовуються регресійні моделі, штучні нейронні мережі (багатошарові персептрони), методи опорних векторів, а також дерева рішень та моделі ансамблі, що представляють собою поєднання деяких перелічених моделей. Багатошарові персептрони з зворотним поширенням широко використовуються для розпізнавання різних категорій зображень, таких як рукописні цифри, почерк, людські особи та дані зорових сенсорів робото-технічних систем. Багатошарова модель персептрона – це набір штучних нейронів – обчислювальної одиниці моделі – об'єднаних рівні (шари), задані в ієрархічному порядку (рис. 1.1). Штучний нейрон являє собою модель біологічного нейрона (нервової клітини), представлену одним або декількома

[1)] [1] A Probabilistic Perspective (Adaptive Computation and Machine Learning series), ISBN-13: 978-0262018029, ISBN-10: 0262018020. (дата звернення 03.09.2021).

входами, одним виходом і функцією активації. Основною метою цієї роботи – є створення універсального програмного забезпечення, яке зможе автоматизувати процес створення класифікаторів на основі переданих файлів, розбитих на категорії. Таке програмне забезпечення може бути встановлене в будь-яку інтернет систему, або будь-яку іншу інформаційну платформу, що дозволить розширити можливості сортування, фільтрування зображень.

Оскільки часто важко сформулювати правила аналізу для класифікації зображень за категоріями розпізнавання часто представляється скрутним, здатність навчатися на базі вибірки робить нейронні мережі і споріднені їм моделі придатними для розпізнавання природних зображень навколишнього світу, які мають нечіткі структури та багато варіацій в межах класу [2][1)].

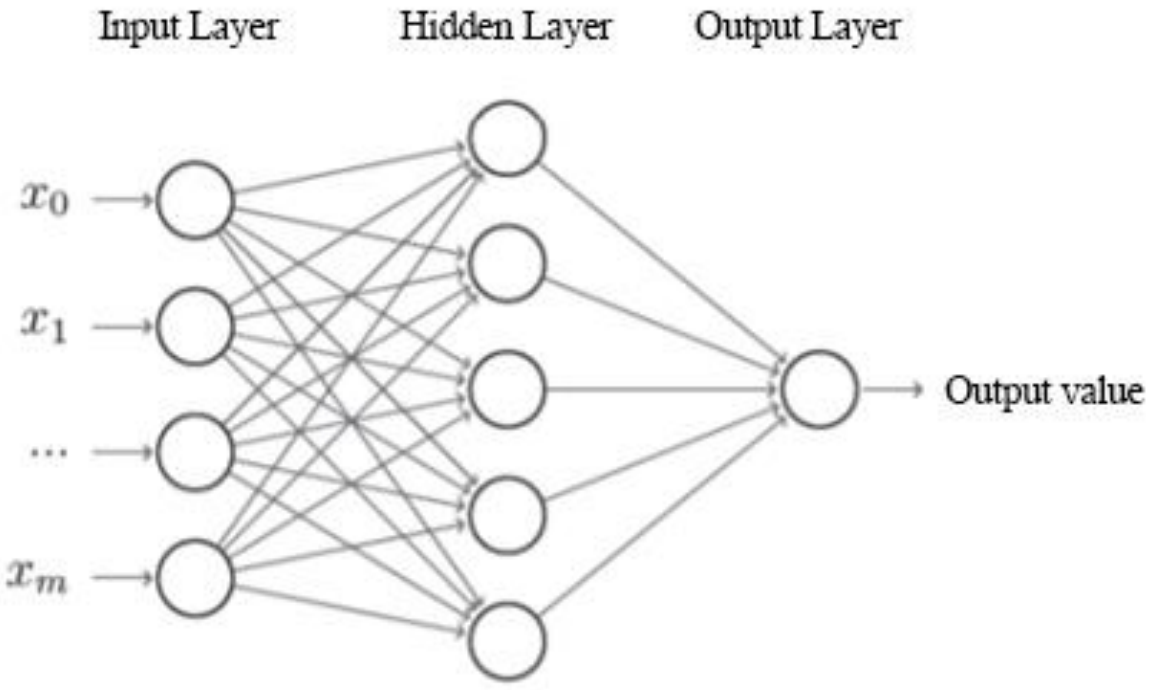

Рисунок 1.1 – Схема штучної нейронної мережі із трьома шарами

[1)] [2] Perceptrons: An Associative Learning Network. URL: https://ei.cs.vt.edu/~history/Perceptrons.Estebon.html. (дата звернення 04.09.2021).

Багатошарові персептрони успішно використовують їх для розпізнавання окремих обмежених категорій зображень, таких як символи природної мови, рукописні цифри та почерк. Наразі в більшості додатків, які використовують безпосереднє навчання викладача для розпізнавання зображень, нейронні мережі замінені методами опорних векторів, які забезпечують більш ефективне рішення з точки зору обчислювальних ресурсів. Особливістю завдань розпізнавання зображень є те, що дані як візуальні сигнали демонструють надзвичайно низьку інформаційну ємність, тобто більшість точок растрового зображення (наприклад, що відповідають одному кольору або рівномірно розподіленої фонової області) не містять інформації, яка впливає на розпізнавання.

При цьому розмірність зображень, що використовуються в системах обробки інформації, зазвичай дуже велика, сучасні мультимедіа, графічні дисплеї та датчики забезпечують велику кількість розподілених зображень (фото, відеокадри, комп'ютерна графіка) з високою роздільністю, з розмірністю в мільйони . Для класичного методу розпізнавання образів він характеризується прямим зв’язком між розмірністю (кількістю параметрів) вибіркових даних і часом навчання, а також індексом оптимізації та збіжності моделі.

Негативно впливає на продуктивність моделі наявність великої кількості параметрів, більшість з яких не містить необхідної для розпізнавання інформації, крім дуже високих вимог до обчислювальних ресурсів, це також призведе до перенавчання. Коли функція розпізнавання апроксимується моделлю на навчальній вибірці А, задовільна класифікація. Немає узагальнень і показує низьку точність у тестовому зразку.

Для вирішення цієї проблеми використовується підхід пошуку компактного представлення зображення [3][1)] – виділення обмеженої кількості

[1)] [3] Neural Networks and Deep Learning: A Textbook 1st ed. 2018 Edition, ISBN-13: 978-3319944623, ISBN-10: 3319944622. (дата звернення 13.09.2021).

генералізованих ознак, що містять основну інформацію, необхідну для розпізнавання.

1.2 Склад програмного продукту

ML.NET – це кросплатформний фреймворк машинного навчання для розробників .NET, а Model Builder – інструмент інтерфейсу користувача у Visual Studio, який використовує автоматизоване машинне навчання (AutoML), щоб легко вчити та використовувати власні моделі ML.NET. За допомогою ML.NET і Model Builder можна створювати власні моделі машинного навчання для таких сценаріїв, як аналіз настроїв, прогнозування цін, класифікатори та інші типи моделей.

Історично класифікація зображень – це проблема, яка популяризувала глибокі нейронні мережі, особливо візуальні типи нейронних мереж – згорткові нейронні мережі (CNN). Проте можна сказати, що CNN були популяризовані після того, як у 2012 році побили рекорд у The ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Цей конкурс широко оцінює алгоритми виявлення об’єктів та класифікації зображень. Набір даних, який вони надають, містить 1000 категорій зображень і понад 1,2 мільйона зображень. Мета алгоритму класифікації зображень – правильно передбачити, до якого класу належить об’єкт. З 2012 року кожен переможець цього конкурсу використовував CNN.

Навчання глибоких нейронних мереж може бути обчислювальним і займати багато часу. Щоб отримати дійсно хороші результати, вам потрібна велика обчислювальна потужність, а це означає багато графічних процесорів, а це означає великі витрати. Звісно, кожен міг би навчати ці великі архітектури та отримувати результати SOTA в хмарних середовищах, але це також досить дорого.

Якийсь час ці архітектури були недоступні для звичайних розробників. Однак концепція трансферного навчання змінила це. Особливо для

вирішуваної сьогодні проблеми – класифікації зображень. Сьогодні можна використовувати найсучасніші архітектури, які перемогли на конкурсі, завдяки навчанню з перенесенням і попередньо підготовленим моделям [4][1)].

По суті, попередньо навчена модель – це збережена мережа, яка раніше була навчена на великому наборі даних, наприклад, на наборі даних. Оскільки великі набори даних зазвичай використовуються для деяких глобальних рішень, можна налаштувати попередньо підготовлену модель і спеціалізувати її для певних проблем. Таким чином можна використовувати деякі з найвідоміших нейронних мереж, не втрачаючи занадто багато часу та ресурсів на навчання. Крім того, можна точно налаштувати ці моделі, змінивши поведінку вибраних шарів. Вся ідея зводиться до використання нижніх рівнів попередньо навченої моделі CNN і додавання додаткових шарів, які налаштують архітектуру для конкретних проблем.

По суті, серйозні рішення щодо класифікації зображень зазвичай складаються з двох частин. Вони називаються їх хребтом і головою. Основна архітектура, як правило, є глибокою архітектурою, яка була попередньо навчена на наборі даних без верхніх шарів. Голова є частиною моделі класифікації зображень, яка використовується для передбачення користувацьких класів.

Ці шари додаються поверх попередньо підготовленої моделі. З цими системами маємо дві фази: вузьке місце і етап навчання. Під час фази вузького місця зображення певного набору даних проходять через архітектуру магістралі, а результати зберігаються. Під час фази навчання збережений вихід із основної мережі використовується для навчання користувацьких шарів (рис. 1.2).

[1)] [4] VGG16 – Convolutional Network for Classification and Detection. URL: https://neurohive.io/en/popular-networks/vgg16/. (дата звернення 16.09.2021).

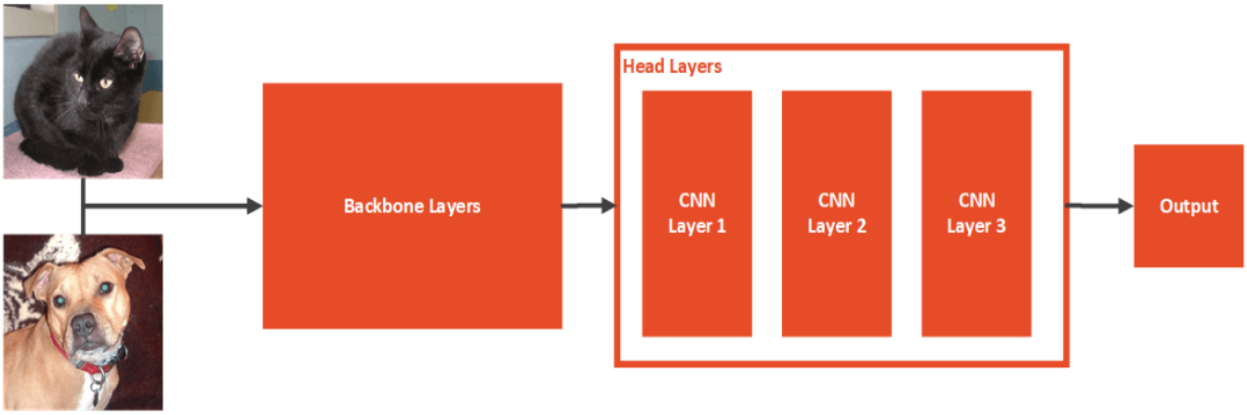

Рисунок 1.2 – Процес класифікації зображень

Протягом багатьох років кілька архітектур, які виграли конкурс ImageNet, стали досить популярними. Деякі з них: VGG16, GoogLeNet (Inception), ResNet [5][1)], [6][2)]. Однак не тільки ці архітектури популярні для трансферного навчання. Наприклад, MobileNet також часто використовується, тому що його легко навчити.

VGG16 – це велика згортка нейронна мережа, запропонована К. Симоняном та А. Зіссерманом у роботі «Дуже глибокі згорткові мережі для широкомасштабного розпізнавання зображень» (рис. 1.3). Ця мережа досягає 92,7% точності тестів топ-5 у наборі даних. Однак його тренували тижнями.

Ось огляд високого рівня моделі:

[1)] [5] Deep Learning: GoogLeNet Explained. URL: https://towardsdatascience.com/deep-learning-googlenet-explained-de8861c82765. (дата звернення 22.09.2021).

[2)] [6] ResNet (34, 50, 101): «остаточные» CNN для классификации изображений. URL: https://neurohive.io/ru/vidy-nejrosetej/resnet-34-50-101/. (дата звернення 22.09.2021).

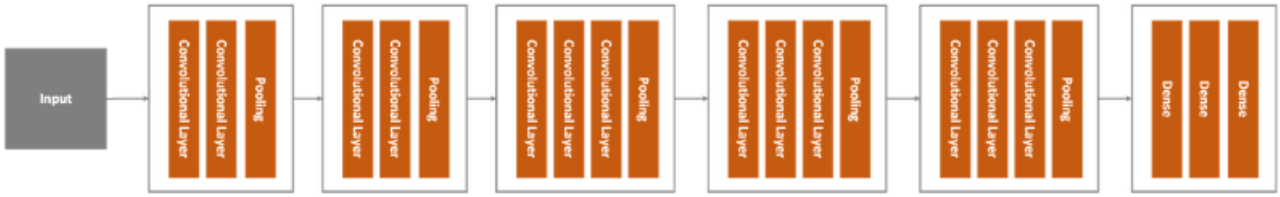

Рисунок 1.3 – Модель архітектури типу VGG16

GoogLeNet також називають Inception (рис. 1.4). Це тому, що він використовує дві концепції: 1×1 Convolution і Inception Module. Перша концепція, 1×1 Convolution, використовується як модуль зменшення розмірів. При зменшенні кількості вимірів кількість обчислень також зменшується, а це означає, що глибину та ширину мережі можна збільшити. Замість використання фіксованого розміру для кожного шару згортки, GoogLeNet використовує початковий модуль:

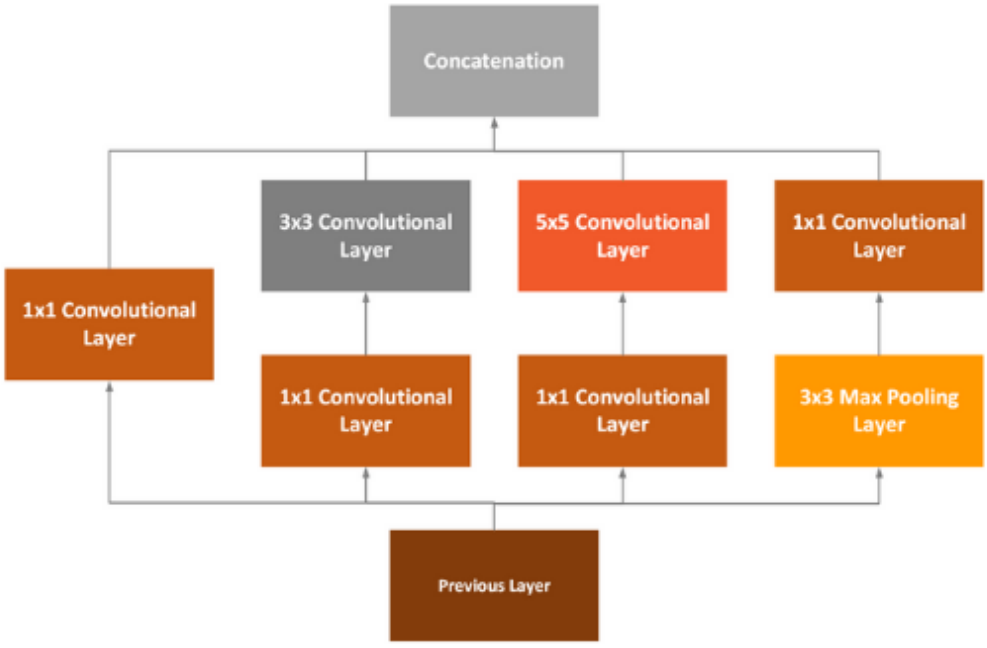

Рисунок 1.4 – Модель архітектури типу GoogLeNet

Як можна бачити, шар згортки 1×1, шар згортки 3×3, шар згортки 5×5 і шар об'єднання 3×3 максимального об'єднання виконують свої операції разом, а потім їх результати знову сумуються разом на виході. Всього GoogLeNet має 22 шари, і він виглядає приблизно так (рис. 1.5):

Рисунок 1.5 – Розгорнута послідовність виконання шарів GoogLeNet

Залишкові мережі або ResNet – це остаточна архітектура (рис. 1.6). Проблема попередньої архітектури полягає в тому, що вони дуже глибокі. Вони мають багато шарів, і через це їх важко тренувати (зникаючий градієнт).

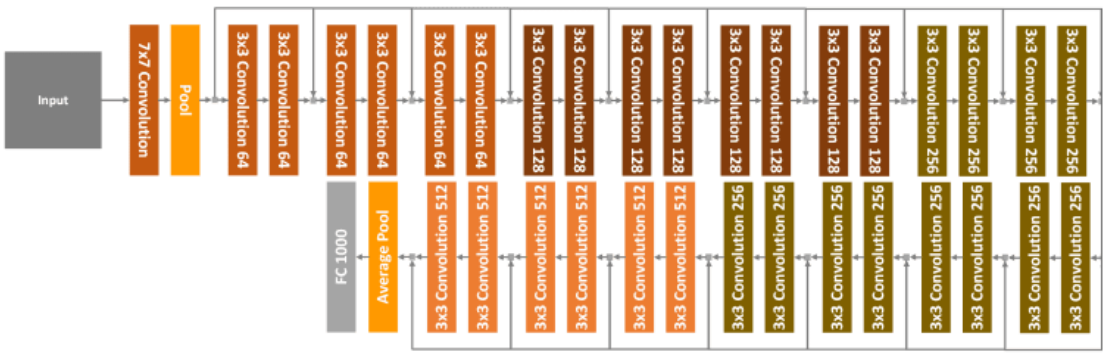

Рисунок 1.6 – Повна архітектура ResNet

Отже, ResNet вирішив цю проблему за допомогою так званого «підключення до ідентифікаційного ярлика» або залишкових блоків (рис. 1.7).

По суті, ResNet дотримується дизайну згорткового шару VGG 3×3, де за кожним згортковим шаром слідує шар пакетної нормалізації та функція активації ReLU. Однак різниця полягає в тому, що перед остаточним ReLu ResNet вводить вхідні дані. Один із варіантів полягає в тому, що вхідне значення проходить через шар згортки 1×1.

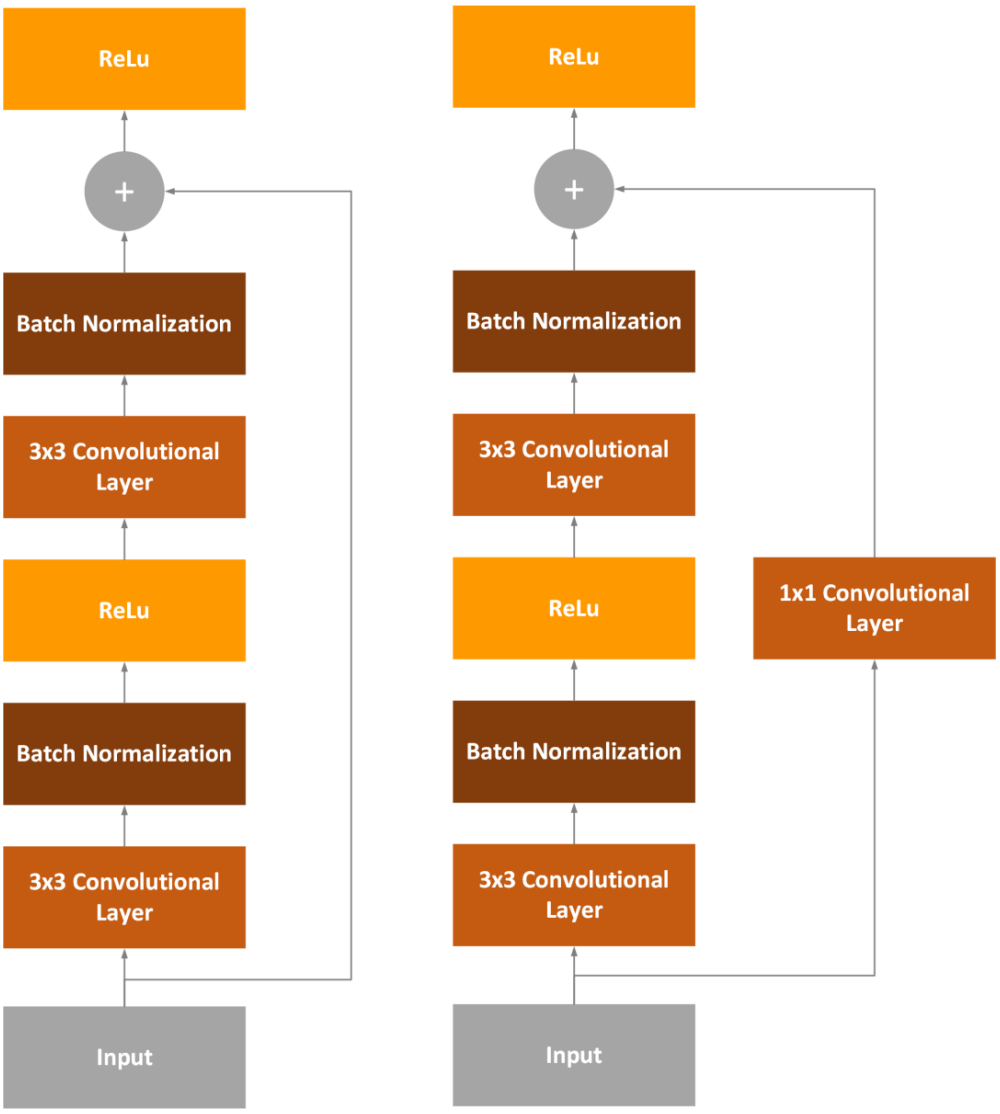

Рисунок 1.7 – Модульна система ResNet

Основна ідея полягає в тому, що більш глибока мережа не повинна створювати помилку навчання вище, ніж дрібна мережа. Автори ResNet припускають, що якщо ви додаєте шари, які нічого не впливають на мережеву помилку, вони повинні залишитися незмінними. Це означає, що дозволити мережі підігнати залишки легше, ніж дозволити їм безпосередньо вмістити всі необхідні дані. Це досягається за допомогою залишкових блоків.

MobileNet – це сімейство нейронних мереж комп'ютерного зору загального призначення, розроблених для мобільних пристроїв для підтримки класифікації, виявлення тощо.

1.2.1 ML.NET та TensorFlow

ML.NET не надає способу побудови нейронної мережі, на основі простого персептрона. По суті, це не інструмент для цього, як TensorFlow і Pythorch є в Python. Однак можна використовувати ML.NET у поєднанні з TensorFlow (точніше TensorFlow.NET), щоб використовувати попередньо навчені моделі, які надає TensorFlow. Як показано на наступній діаграмі, додаємо посилання на пакети ML.NET NuGet у програми .NET Core або .NET Framework. Під обкладинкою ML.NET містить і посилається на рідну бібліотеку TensorFlow, яка дозволяє писати код, який завантажує наявний навчений файл моделі TensorFlow [7][1)].

Таким чином, додаючи посилання на пакети ML.NET NuGet в додатках .NET, а ML.NET містить і посилається на рідну бібліотеку TensorFlow. Це дає можливість використовувати попередньо навчені моделі TensorFlow (рис. 1.8).

[1)] [7] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems 2nd Edition, ISBN-13: 978-1492032649, ISBN-10: 1492032646. (дата звернення 02.10.2021).

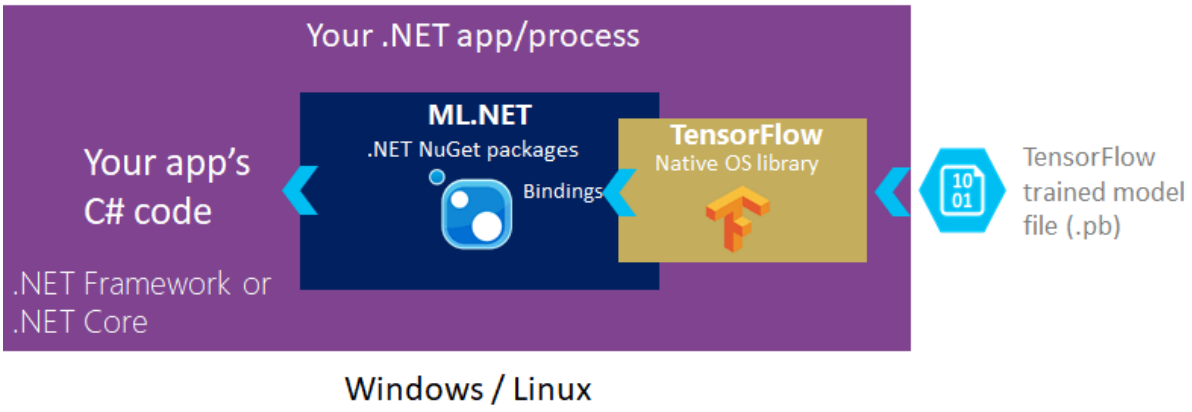

Рисунок 1.8 – Взаємодія тренованих моделей з платформою

1.2.2 Архітектура ML.NET високого рівня

Давайте розглянемо високорівневу архітектуру цієї реалізації. Тут створюється рішення, яке дозволяє швидко змінити архітектуру, яку використовуємо для класифікації зображень. Ідея полягає в тому, щоб розділити різні частини програми на основі роботи, яку вони виконують. Структура папок рішення виглядає так (рис. 1.9):

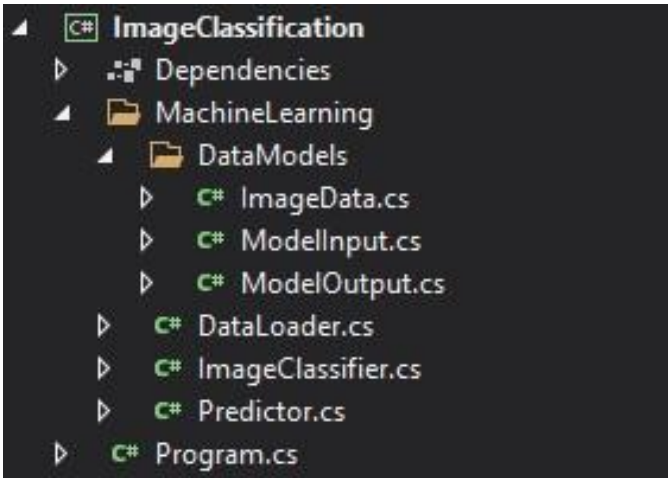

Рисунок 1.9 – Структура проекту

Папка DataModel містить класи, які моделюють дані. Клас DataLoader завантажує зображення з папки, попередньо обробляє їх і розбиває на набори даних для навчання та тесту. Клас ImageClassifier обгортає модель (рис.1.10).

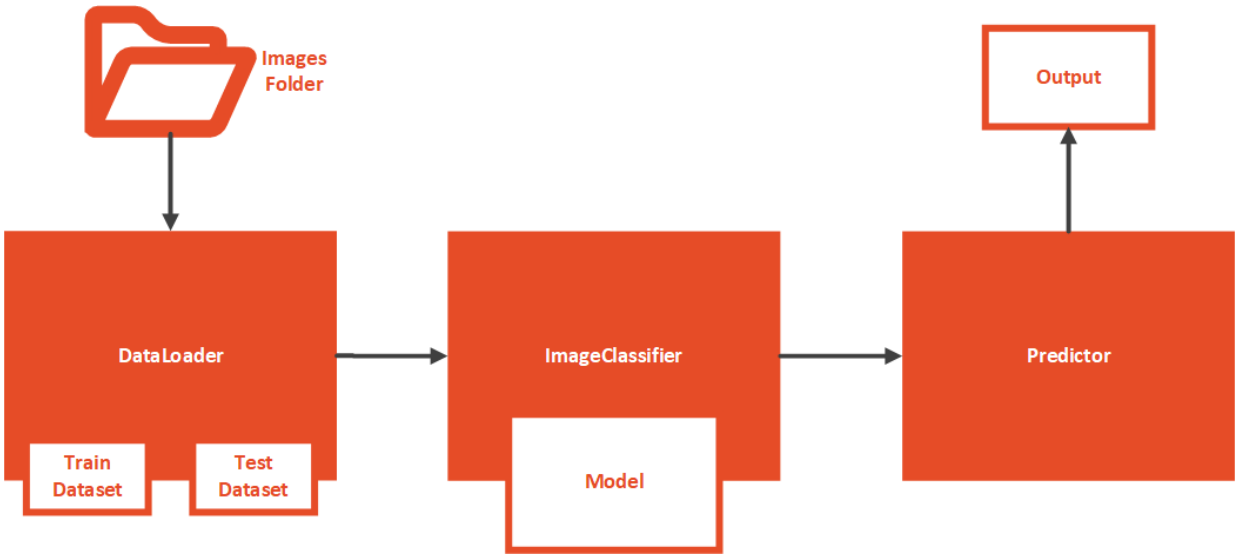

Рисунок 1.10 — Огляд архітектури

Він використовує дані, надані DataLoader, для навчання моделі. Нарешті, клас Predictor використовує тестовий набір даних і модель навчання для прогнозування.

Для того, щоб завантажувати дані з зображень і використовувати їх з алгоритмами ML.NET, необхідно реалізувати класи, які збираються моделювати ці дані. У папці даних можна знайти три файли: ImageData, ModelInput і ModelOutput. Клас InputData містить основну інформацію про зображення:

```
namespace ImageClassification.MachineLearning.DataModels
{
    public class ImageData
    {
        public string ImagePath { get; set; }

        public string Label { get; set; }
```

```
    }
}
```

Цей клас містить шлях до зображення та мітку. Інформація про мітку витягується з імені файлу.

Клас ModelInput моделює вихідні дані, які використовуються як вхідні дані для моделі:

```
namespace ImageClassification.MachineLearning.DataModels
{
    public class ModelInput
    {
        public byte[] Image { get; set; }

        public UInt32 LabelAsKey { get; set; }

        public string ImagePath { get; set; }

        public string Label { get; set; }
    }
}
```

Властивість Image – це масив байтів і байтове представлення зображення. Модель очікує, що дані зображення будуть такого типу для навчання. LabelAsKey є закодованим значенням Label.

Нарешті, вихід моделі представляє вихід моделі:

```
namespace ImageClassification.MachineLearning.DataModels
{
    public class ModelOutput
    {
        public string ImagePath { get; set; }

        public string Label { get; set; }

        public string PredictedLabel { get; set; }
    }
}
```

1.2.3 Обґрунтування вибору ML.NET

ML.NET або машинне навчання завжди обіцяє вирішити проблему стислим способом і надати багато переваг користувачеві, роблячи правильні прогнози та допомагаючи їм приймати правильні рішення. Нижче наведені основні переваги ML.NET:

- ML.NET може перевірити великий обсяг набору даних, а потім надати можливі тенденції та моделі результатів відповідно до цих даних.
- Це повний автоматизований процес для аналізу даних. Під час виконання проектів ML.NET не потрібна взаємодія з людьми.
- Зі збільшенням досвіду щодо обсягу вибірки даних алгоритм, написаний у ML. NET, стане набагато точнішим та ефективнішим.
- Алгоритми ML.NET дуже ефективні для обробки багатовимірних і різноманітних даних.
- Універсальність програм на базі ML.NET дуже велика. Можна використовувати програми ML.NET у будь-якій галузі, як охорона здоров'я, маркетинг, продажі тощо, для аналізу вимог або вибору клієнтів.

Також слід зазначити функції та покращення системи, які компанія Microsoft постійно впроваджує в свою систему. Завдяки тому, що позаду ML.NET стоїть така велика корпорація, стає зрозуміло, чому цю платформу обирають тисячі розробників.

Microsoft випустила ML.Net як фреймворк з відкритим вихідним кодом після майже 10-річних досліджень, щоб надати розробнику можливість створити сучасний додаток із налаштованим машинним навчанням [8][1)]. Випуск ML.Net 1.0 включає наступні функції:

[1)] [8] Programming ML.NET 1st Edition, ISBN-13: 978-0137383658, ISBN-10: 0137383657. (дата звернення 07.10.2021).

- ML.Net – це фреймворк з відкритим вихідним кодом і крос-платформний фреймворк.
- Перша версія ML.Net надає навчені та прогнозні моделі.
- Крім того, ML.Net підтримує деякі основні компоненти, такі як перетворення, структуру даних на основі базового машинного навчання, а також деякі зразки алгоритму навчання.
- ML.Net забезпечує підвищення ефективності додатків.
- ML.Net набагато швидший і надійніший за конкурентів.

1.3 Альтернативні способи навчання класифікаторів

На просторах інтернету ми можемо познайомитися з багатьма аналогами, наприклад як утиліта Keras та класифікація зображень з використанням згорткових нейронних мереж.

1.3.1 Утиліта Keras

Бібліотека глибокого навчання, що представляє собою високоуровневий API, написаний на Python і здатний працювати поверх TensorFlow, Theano або CNTK. Він був розроблений за розрахунком на швидке навчання. Спосіб переходу від гіпотезу до результатів з найменшими тимчасовими витратами є ключем до проведення успішних досліджень [9][1)].

Keras використовують якщо потрібна бібліотека глибокого навчання, яка:

- дозволяє легко і швидко створювати прототипи (благодаря зручності, модульності та масштабованості);
- підтримує як сверточні та рекурентні мережі, так і їх комбінації;

[1)] [9] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, 2nd Edition. (дата звернення 10.10.2021).

– без проблем працює як з процесором (CPU), так і з графічним процесором (GPU).

1.3.2 Класифікація зображень з використанням згорткових нейронних мереж

Згорткові нейронні мережі входять у піддомен машинного навчання, тобто глибоке навчання. Алгоритми глибокого навчання обробляють інформацію так само, як людський мозок, але, очевидно, у дуже невеликому масштабі, оскільки наш мозок надто складний (у нашому мозку близько 86 мільярдів нейронів).

Класифікація зображень включає вилучення функцій зображення для спостереження за деякими закономірностями в наборі даних. Використання ІНС для класифікації зображень може виявитися дуже дорогим з точки зору обчислень, оскільки параметри, що навчаються, стають надзвичайно великими.

2 ТЕХНОЛОГІЇ ТА СТРУКТУРА РОЗРОБКИ

Далі будуть визначені загальні положення щодо початку імплементації, набор технологій, а також процесів, які залучені під час написання проекту. Також, визначено переваги кожної з технологій та обґрунтовано включення її до створюваної системи.

2.1 .NET 5 – уніфікована платформа

На риунку 2.1 показано, що платформа об'єднує DESKTOP, WEB, CLOUD, MOBILE, GAMING, IoT і AI. Раніше в офіційному блозі Microsoft писали про реліз ML.Net, і Model Builder зокрема.

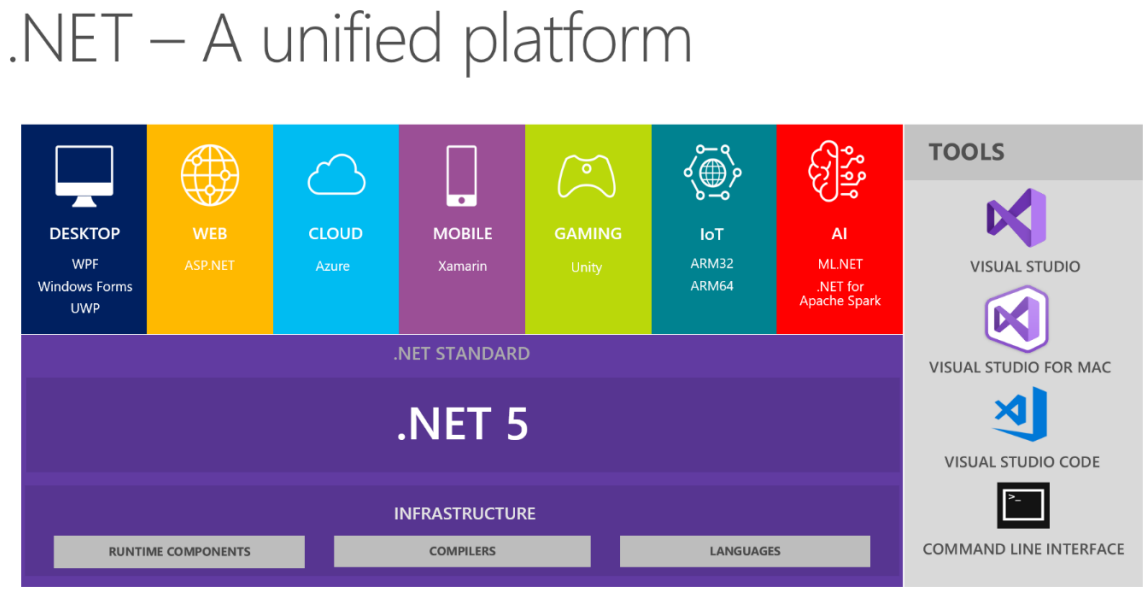

Рисунок 2.1 – Архітектура платформи .NET

Отже є спеціальне розширення для Visual Studio під назвою Model Builder, який дозволяє додати машинне навчання в проект правою кнопкою миші.

На даний момент доступні 4 сценаріїв навчання:

- Sentiment analysis – аналіз тональності, binary classification (бінарна класифікація), по тексту визначається його емоційне забарвлення, позитивний або негативний.
- Issue classification – multiclass classification (многоклассовая класифікація), цільова мітка для issue (тікета, помилки, звернення на підтримку і т.д.) може бути обрана як один з трьох взаємовиключних варіантів.
- Price prediction-regression, класична задача регресії, коли вихідним результатом є безперервне число; в прикладі це оцінка квартири.
- Image classification-multiclass classification (класифікація), але вже для зображень.

Зауважимо, що немає multilabel-класифікації, коли цільових метод може бути багато одночасно. Для зображень немає можливість вибору завдання сегментації. Я припускаю, що за допомогою фреймворка вони в цілому можна вирішити, однак сьогодні фокусуємося саме на билдер. Здається, що масштабування візарду на розширення кількості завдань не є важким завданням, тому варто очікувати в їх майбутньому [10][1)].

Варто зауважити, що швидкість навчання обмежена ресурсами локальної машини. Після цього починається тренування моделі: на цьому кроці послідовно навчаються різні моделі, для кожної виводиться скор, і в кінці вибирається найкраща. І наприкінці проводиться оцінка моделі: на даному етапі можна подивитися на те, які цільові метрики були досягнуті, а так же поганяти модель наживо (рис. 2.2).

Як тільки модель є протестованою та підготовленою для наступного кроку починається генерація коду. На цьому кроці згенеруються і додаються в рішення (solution) два проекти. В одному з них є повноцінний приклад використання моделі, а в інший слід заглядати тільки якщо цікаві деталі реалізації.

[1)] [10] Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming 10th ed. Edition, ISBN-13: 978-1484269381, ISBN-10: 1484269381. (дата звернення 21.10.2021).

Після всього процесу буде можливо використовувати модель в реальному житті:

```
|                                Top 2 models explored
|
--------------------------------------------------------------------------------
----
|     Trainer                 MicroAccuracy  MacroAccuracy  Duration #Iterat
ion|
|1    SdcaMaximumEntropyMulti        0,7475         0,5426     176,7
1|
|2    AveragedPerceptronOva          0,7128         0,4492      42,4
2|
--------------------------------------------------------------------------------
----
```

Рисунок 2.2 – Висновки щодо навчання моделей

Висновки щодо ML Builder: білдер пропонує на вибір кілька сценаріїв, в яких не виявлено критичних місць, що вимагають занурення в ML. З цієї точки зору він прекрасно підходить як інструмент "getting started" або вирішення типових простих завдань тут і зараз.

2.2 UI (інтерфейс користувача) на базі Vuejs

Vue – це прогресивний фреймворк для створення інтерфейсів користувача. На відміну від монолітного фреймворка, Vue розроблений для поступового впровадження. Його ядро в основному вирішує проблеми на рівні перегляду та спрощує інтеграцію з іншими бібліотеками та існуючими проектами. З іншого боку, у поєднанні з сучасними інструментами та додатковими бібліотеками, Vue ідеально підходить для створення складних односторінкових програм (SPA, Single-Applications).

Компоненти є важливою концепцією Vue. Ця абстракція дозволяє з маленьких «фрагментів» збирати великі програми. Вони є об’єктами

багаторазового використання. Якщо подумати, майже будь-який інтерфейс можна вважати деревом компонентів (рис. 2.3):

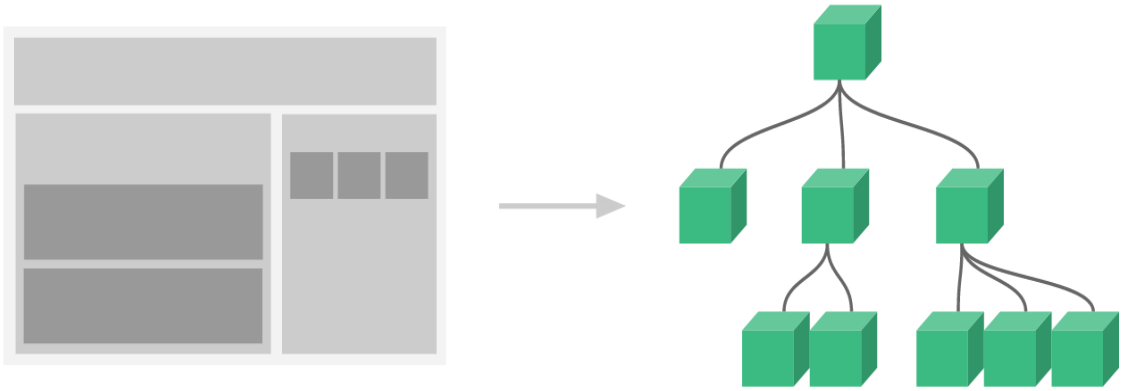

Рисунок 2.3 – Деревоподібна структура поєднання компонентів

Додаток Vue складається з кореневого примірника Vue, створюваного за допомогою new Vue, опціонально організованого в дерево вкладених, повторно використовуваних компонентів [11][1)]. Наприклад, дерево компонентів для додатка TODO-списку може виглядати так:

```
Кореневий екземпляр
└─ TodoList
   ├─ TodoItem
   │  ├─ TodoButtonDelete
   │  └─ TodoButtonEdit
   └─ TodoListFooter
      ├─ TodosButtonClear
      └─ TodoListStatistics
```

[1)] [11] Front-End Development Projects with Vue.js: Learn to build scalable web applications and dynamic user interfaces with Vue 2, ISBN-13: 978-1838984823, ISBN-10: 1838984828. (дата звернення 27.10.2021).

Самостійний додаток складається з наступних частин:

– Стан – «джерело істини», керуючий додатком;
– Уявлення – відображення стану заданий декларативно;
– Дії – можливі шляхи зміни стану програми у відповідь на взаємодію користувача з поданням.

На рисунку 2.4 найпростіше уявлення концепції «односпрямованого потоку даних»:

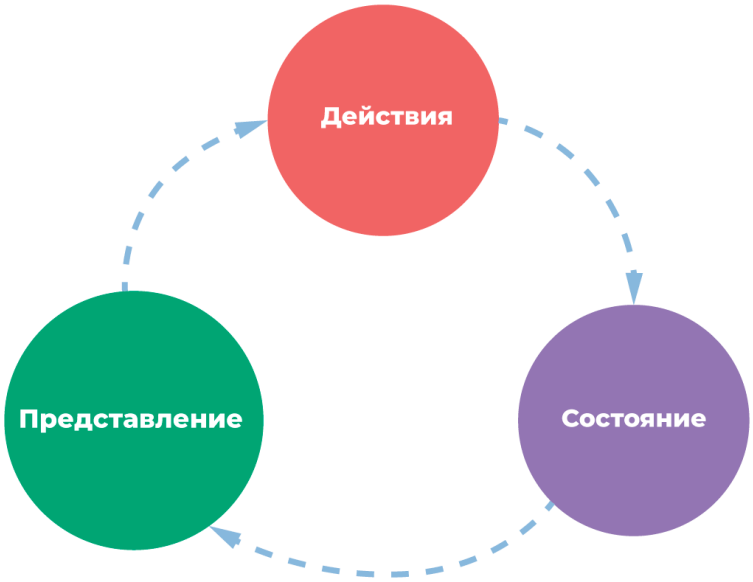

Рисунок 2.4 – Односпрямований потік даних

Однак простота швидко зникає, коли у нас з'являється кілька компонентів, що ґрунтуються на одному і тому ж стані:

– Кілька уявлень можуть залежати від однієї і тієї ж частини стану програми.
– Дії з різних уявлень можуть впливати на одні і ті ж частини стану програми.

Вирішуючи першу проблему, доведеться передавати одні й ті ж дані вхідними параметрами в глибоко вкладені компоненти. Це часто складно і

утомливо, а для сусідніх компонентів таке і зовсім не спрацює. Вирішуючи другу проблему, можна прийти до таких рішень, як звернення по посиланнях до батьківських або дочірніх екземплярів або спробам змінювати і синхронізувати кілька копій стану через події. Обидва підходи тендітні і швидко призводять до появи коду, який неможливо підтримувати (рис. 2.5).

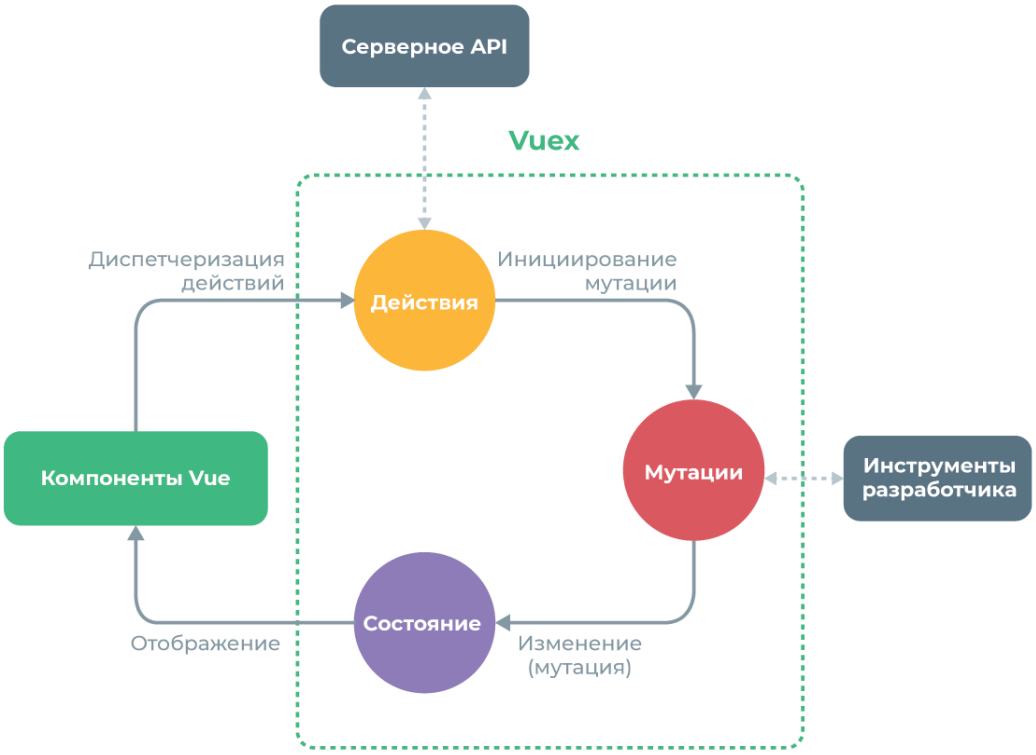

Рисунок 2.5 – Життєвий цикл глобального API Vuex

Чітко визначаючи і розділяючи концепції, що виникають при управлінні станом, і вимагаючи дотримання певних правил, які підтримують незалежність між уявленнями і станами, можна краще структурувати код і полегшувати його підтримку.

2.3 Серверна частина

Entity Framework Core (EF Core) – це об’єктно-орієнтована, легковажна та розширювана технологія від Microsoft для доступу до даних. EF Core – це інструмент ORM (об’єктно-реляційне відображення даних з реальними об’єктами). Іншими словами, EF Core дозволяє використовувати базу даних, але з більш високим рівнем абстракції: EF Core дозволяє абстрагуватися від самої бази даних і її таблиць і працювати з даними незалежно від типу сховища (рис. 2.6).

Якщо на фізичному рівні ми оперуємо таблицями, індексами, первинними і зовнішніми ключами, але на концептуальному рівні, який нам пропонує Entity Framework [12][1)], ми вже працюємо з об'єктами.

Entity Framework Core підтримує багато різних систем баз даних. Тому ми можемо використовувати EF Core з будь-якою базою даних, якщо вона має відповідного постачальника.

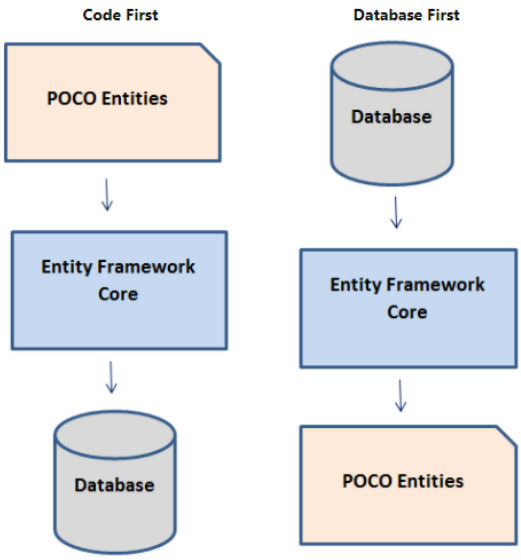

Рисунок 2.6 – Підходи створення БД

[1)] [12] Entity Framework Core. URL: https://docs.microsoft.com/en-us/ef/core/. (дата звернення 29.10.2021).

EF Core підтримує два підходи до розробки – Code-First та Database-First. EF Core в основному націлений на підхід, що ґрунтується на першому коді, і надає незначну підтримку підходу на основі бази даних, оскільки візуальний конструктор або майстер моделі БД не підтримуються з EF Core 2.0.

У підході на основі коду EF Core API створює базу даних і таблиці за допомогою міграції на основі умов та конфігурації, наданих у ваших класах домену. Цей підхід корисний у дизайні, керованому доменом (DDD).

У підході на основі бази даних EF Core API створює домен і класи контексту на основі наявної бази даних за допомогою команд EF Core. Це має обмежену підтримку в EF Core, оскільки не підтримує візуальний дизайнер або майстер.

За замовчуванням на даний момент Microsoft надає багато вбудованих постачальників: для MS SQL Server, для SQLite і для PostgreSQL. Існують також сторонні постачальники, такі як MySQL.

Також варто відзначити, що EF Core надає загальний API для обробки даних. І, наприклад, якщо ми вирішимо змінити цільову базу даних, основні зміни в проекті будуть в основному включати конфігурацію та конфігурацію підключення до відповідного провайдера.

А код, який безпосередньо працює з даними, отримує дані, додає їх в БД і т.д., залишиться колишнім.

Термін «Onion Architecture» ввів Джеффрі Палермо в 2008 році. З роками ця концепція стала дуже популярною і є одним з найбільш часто використовуваних типів архітектури при створенні програм на ASP.NET. Onion Architecture ділить програму на кілька рівнів (рис. 2.7). Архітектура залежить не від рівня даних, як у класичних багаторівневих архітектурах, а від реальних моделей домену. Незалежний рівень, який є ядром архітектури. Другий шар залежить від першого шару, третій шар залежить від другого шару і так далі. Тобто виходить, що навколо першого незалежного рівня знаходиться друга залежність ієрархії. Третій знаходиться навколо другого,

але також може залежати від першого. Образно кажучи, це можна виразити у вигляді цибулі, у неї також є серцевина, а всі інші шари знаходяться навколо серцевини до зовнішньої оболонки.

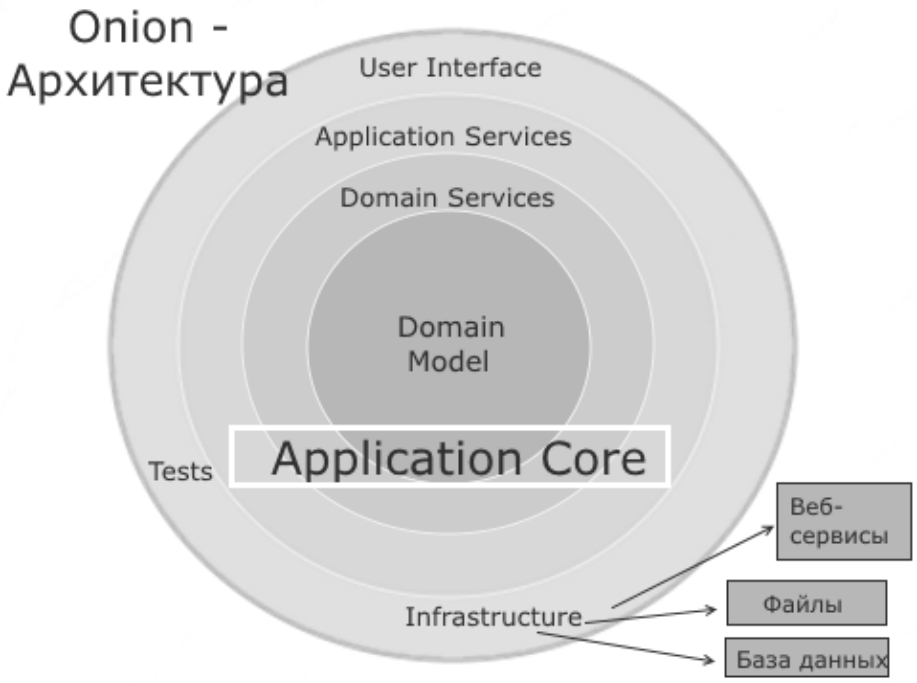

Рисунок 2.7 – Onion-архітектура

Кількість рівнів може відрізнятися, але в центрі завжди знаходиться модель домену (Domain Model), тобто ті класи моделей, які використовуються в додатку і об'єкти яких зберігаються в базі даних.

Переваги цибулевої архітектури:

- Шари Onion Architecture підключаються через інтерфейси. Імплантація здійснюється під час виконання.
- Архітектура програми побудована на основі моделі домену.
- Усі зовнішні залежності, такі як доступ до бази даних і виклики служб, представлені на зовнішніх рівнях.
- Немає залежностей внутрішнього шару із зовнішніми шарами.
- Муфти розташовані до центру.

– Гнучка, стійка та портативна архітектура.
– Немає необхідності створювати спільні проекти.
– Можна швидко протестувати, оскільки ядро програми ні від чого не залежить.

Кілька недоліків цибулевої архітектури:

– Нелегко зрозуміти для початківців, пов’язане з навчанням. Архітектори здебільшого плутають, розподіляючи обов’язки між шарами.
– Інтенсивно використовувані інтерфейси

3 ПРОЕКТУВАННЯ СИСТЕМИ ТА ЇЇ РЕАЛІЗАЦІЯ

Класифікація зображень – це завдання комп’ютерного зору, яке відноситься до категорії глибокого навчання. Для того щоб навчати модель потрібно позначати вхідне зображення одним із заданих цільових класів на основі вже позначених зображень навчального набору. Таким чином необхідно мати набір даних із зображеннями різних категорій. Завдання полягає в тому, щоб створити програму, яка застосовує навчання з перенесенням, попередньо навчену модель TensorFlow і API класифікації зображень ML.NET, щоб ідентифікувати структури з різних категорій та отримувати результати щодо обрання коректної класифікації.

3.1 Розробка конвеєру постачання моделей нейронних мереж

На початку розробки необхідно визначити яким чином будуть формуватися зображення та їх категоризація для тренувань. Можливими варіантами є:

- Мануальний збір інформації;
- Використання платних підписок архівів зображень;
- Використання відкритих постачальників зображень;
- Написання програми-парсера, яка могла б дати змогу автоматизувати процес.

В даному випадку, було обрано останній варіант, який дасть змогу зекономити час та ресурси, а також спростити можливість підбору зображень для автоматичного налаштування користувачів, які не мають часу на підбір колекцій власноруч.

Для реалізації програми-парсера необхідно визначити які ресурси та методики є для реалізації. По-перше, важливим буде розглянути процес пагінації та як прискорити процес завантаження великих масивів даних

паралельно. По-друге, необхідно визначити що дає використання асинхронності під час роботи процесів введення-виводу. А також визначити алгоритми оптимізації багато-поточних процесів та як запобігти виняткових ситуацій з недостатньої кількістю пам’яті.

Пагінація – це розбиття довгого звіту на сторінки з заголовками та службовою інформацією (рис. 3.1). Зміщене розбиття на сторінки найпростіше реалізувати. Однак розбивка сторінок має свої обмеження, наприклад обмеження великих значень зміщення та неточності, які виникають через зсув сторінки. Іноді розбивка сторінок може бути дуже складною для внутрішньої системи.

Розбивка API на сторінки є важливим етапом, якщо наявна велика кількість даних і кінцевих точок. Розбиття на сторінки автоматично передбачає додавання порядку в результат запиту. Ідентифікатор об’єкта є результатом за замовчуванням, але результати можна впорядковувати й іншими способами. Розбивка на сторінки також використовується для швидкого отримання даних із малим часом очікування.

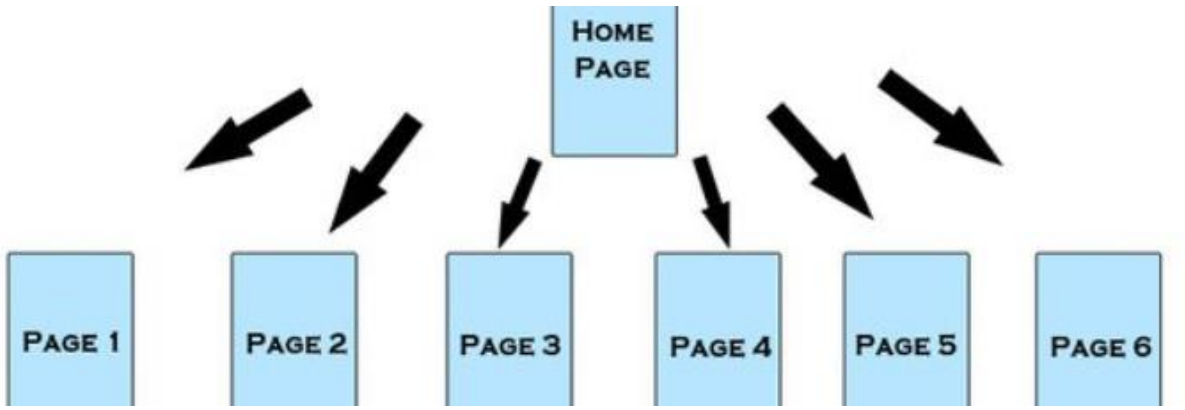

Рисунок 3.1 – Процес пагінації

Уявімо, що у API є кінцева точка, яка потенційно може повернути мільйони записів одним запитом. Скажімо, є сотня користувачів, які збираються використовувати цю кінцеву точку, запитуючи всі дані за один

прийом одночасно. В більшості випадків сервер не зможе витримати таку напругу, що призведе до деяких проблем, включаючи проблему безпеки.

Ідеальна кінцева точка API дозволить споживачам отримувати лише певну кількість записів за один раз. Таким чином, замість повернення всіх даних, система не завантажує сервер бази даних, центральний процесор, на якому розміщено API, або пропускну здатність мережі. Така функція є дуже важливою для будь-якого API, особливо публічного.

Тобто пагінація – це є розбиття на сторінки в методі, за допомогою якого користувач отримує сторінку. Це означає, що запитуючи номер сторінки та розмір сторінки, програма повертає тільки ті дані, які потрібні.

Знаючи як працює пагінація, можна визначити яким чином можна отримувати результати паралельно. Для виконання паралельних процесів, будемо використовувати клас, який пропонує платформа .NET – Task.

Tasks – це конструкції, які використовуються для реалізації Promise Model of Concurrency. Коротше кажучи, вони пропонують обіцянку, що робота буде завершена пізніше, дозволяючи координувати роботу з обіцянкою за допомогою чистого API. Завдання являє собою одну операцію, яка не повертає значення. А Task<T> представляє одну операцію, яка повертає значення типу T.

Важливо розуміти Task як абстракції роботи, що виконуються асинхронно, а не як абстракцію над потоками. За замовчуванням завдання виконуються в поточному потоці та делегують роботу операційній системі відповідно до потреб. За бажанням, Task можна виконувати в окремому потоці через API Task.Run.

Завдання надають протокол API для моніторингу, очікування та доступу до значення результату (у випадку Task<T>) завдання. Інтеграція з ключовим словом await забезпечує абстракцію вищого рівня для використання завдань [13][1)] [14][2)].

[1)] [13] Understanding async/await State Machine in .NET. URL: https://mykkon.work/async-state-machine/. (дата звернення 03.11.2021).

Використання await дозволяє програмі або службі виконувати роботу під час виконання завдання, передаючи контроль її виклику, доки завдання не буде виконано. Вашому коду не потрібно покладатися на зворотні виклики або події, щоб продовжити виконання після завершення завдання. Інтеграція мови та API завдань робить це автоматично. Якщо використовується Task<T>, ключове слово await додатково «розгорне» значення, повернуте після завершення завдання (рис. 3.2).

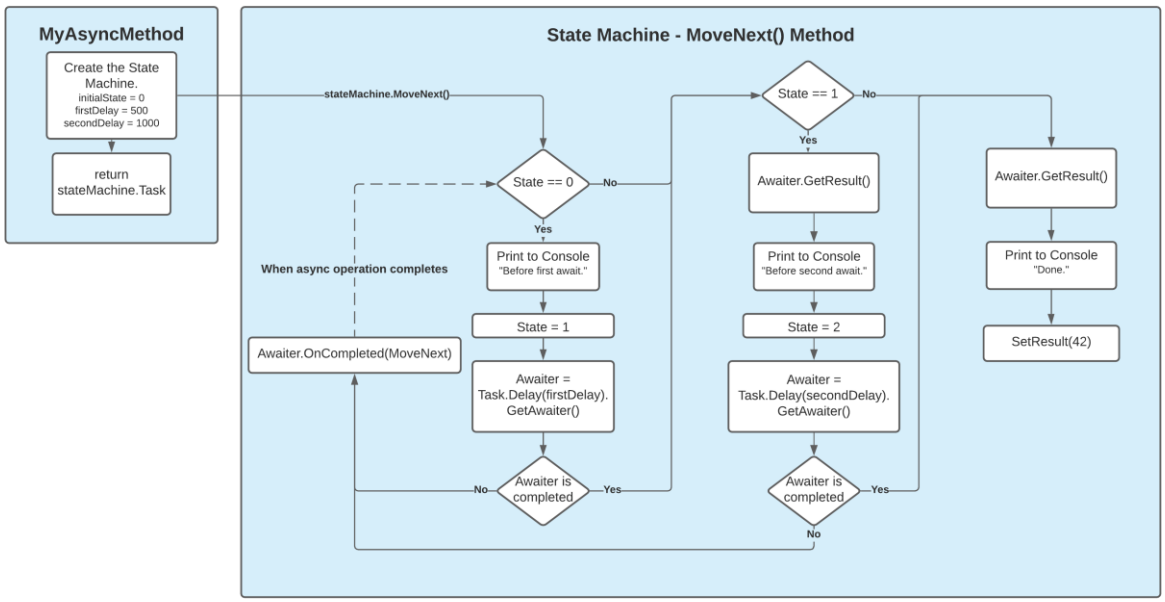

Рисунок 3.2 – Процес використання Async State Machine

Протягом усього цього процесу ключовим висновком є те, що жоден потік не призначений для виконання завдання. Хоча робота виконується в певному контексті (тобто ОС має передавати дані драйверу пристрою та відповідати на переривання), немає потоку, призначеного для очікування повернення даних із запиту. Це дозволяє системі виконувати набагато більший обсяг роботи, а не чекати завершення виклику введення-виводу.

[2)] [14] Exploring the async/await State Machine – Main Workflow and State Transitions. URL: https://vkontech.com/exploring-the-async-await-state-machine-main-workflow-and-state-transitions/. (дата звернення 09.11.2021).

Хоча може здатися що потрібно виконати для цього чимало роботи, але якщо виміряти час, це мізерно порівняно з часом, необхідним для виконання фактичної роботи введення-виводу. Тому під час написання сценарію парсеру було обрано шлях формування асинхронних запитів під час завантаження файлів. Нижче зазначений лістинг коду для виконання цього сценарію:

```
    public static async Task Main(string[] args)
    {
        Console.WriteLine("Scenario `{0}` has been started",
typeof(Program).Assembly.GetName().Name);

        #region Initialization
        var categories = new List<Category>
        {
            new Category{ Name = "portraits", Keywords = new
List<string>{ "human", "face", "man", "woman" } },
            new Category{ Name = "scenery", Keywords = new
List<string>{ "scenery", "landscape", "sky", "city", "street",
"architecture" } },
            new Category{ Name = "food", Keywords = new
List<string>{ "food", "fruits", "vegetables", "meal" } },
            new Category{ Name = "cars", Keywords = new
List<string>{ "cars" } },
            new Category{ Name = "interior", Keywords = new
List<string>{ "interior" } },
            new Category{ Name = "flowers", Keywords = new
List<string>{ "flowers" } },
            new Category{ Name = "dogs", Keywords = new
List<string>{ "dogs" } },
            new Category{ Name = "cats", Keywords = new
List<string>{ "cats" } },
        };

        var parseRequest = new ParseRequest
        {
            Categories = categories,
            EstimatedCount = estimatedCountOfImages
        };
        #endregion

        #region Paths

        #region Archive Settings
        const string defaultArchiveName = "data-set.zip";
        (char Left, char Right) indexBracers = ('(', ')');
        var pattern = defaultArchiveName.Split('.');
        #endregion

        #region Directories
```

```
        var currentDirectory = Directory.GetCurrentDirectory();
        var projectDirectory =
Path.GetFullPath(Path.Combine(currentDirectory, "..", "..",
".."));
        var imagesDirectory = Path.Combine(projectDirectory,
"assets", "inputs", "images");
        #endregion

        Directory.CreateDirectory(imagesDirectory);

        #region Indexing
        var files = Directory.GetFiles(imagesDirectory);
        var currentIndex = files.Any()
            ? (int?)files.Max(path => GetIndexFromPath(path,
pattern, indexBracers))
            : null;
        var nextIndex = currentIndex.HasValue ?
(int?)currentIndex.Value + 1 : null;
        var indexString = nextIndex.HasValue ? $"
{indexBracers.Left}{nextIndex}{indexBracers.Right}" :
string.Empty;
        var archiveName =
$"{pattern[0]}{indexString}.{pattern[^1]}";
        var archive = Path.Combine(imagesDirectory,
archiveName);
        #endregion

        #endregion

        var stopwatch = Stopwatch.StartNew();
        var context = new ParsingContext();

        var progress = GetProgress(stopwatch);
        var parsedImages =
context.ParseImagesAsync(parseRequest, progress);

        using (var fs = new FileStream(archive,
FileMode.CreateNew))
        {
            using var zip = new ZipArchive(fs,
ZipArchiveMode.Update);
            var indexes = categories.SelectMany(x =>
x.Keywords).ToDictionary(x => x, x => default(int));

            await foreach (var parsedImage in parsedImages)
            {
                if (!indexes.TryGetValue(parsedImage.Keyword,
out int index))
                {
                    throw new ArgumentException("There is no
such category in search-list");
                }

                var image = parsedImage.Image;
                var rawFormat = image.RawFormat;
                if (maxWidthOfImage < image.Width)
                {
                    image =
image.ProportionalResizeImageWidth(maxWidthOfImage);
                }
```

```
                var format = new
ImageFormatConverter().ConvertToString(rawFormat).ToLower();
                var entryName = $"{parsedImage.Keyword}{(index
== default ? string.Empty : $"-{index}")}.{format}";
                var entryPath =
Path.Combine(parsedImage.Category, entryName);
                var entry = zip.CreateEntry(entryPath,
CompressionLevel.Optimal);

                using var stream = entry.Open();
                try
                {
                    image.Save(stream, rawFormat);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                    Console.WriteLine(ex.StackTrace);
                    throw;
                }

                if (useLog)
                {
                    Console.WriteLine("Image {0}, Category:
`{1}`, Keyword: `{2}` was parsed",
                                      index,
                                      parsedImage.Category,
                                      parsedImage.Keyword);
                }

                indexes[parsedImage.Keyword] += 1;

                image.Dispose();
                if (!image.Equals(parsedImage.Image))
                {
                    parsedImage.Image.Dispose();
                }
            }
        }

        stopwatch.Stop();

        Console.WriteLine();
        Console.WriteLine("Scenario `{0}` has been finished",
typeof(Program).Assembly.GetName().Name);
        Console.WriteLine("Total process took:");
        Console.WriteLine(stopwatch.Elapsed);
        Console.WriteLine();
    }
```

Як видно з цього листінгу кількість сторінок із зображеннями розбивається на окремі завдання та починається виконання паралельно. Таким чином досягається максимально продуктивність.

Як можна побачити на рисунку 3.3, в консолі виводиться поступовий прогрес. Для цього був використаний шаблон Progress Reporting [15][1)].

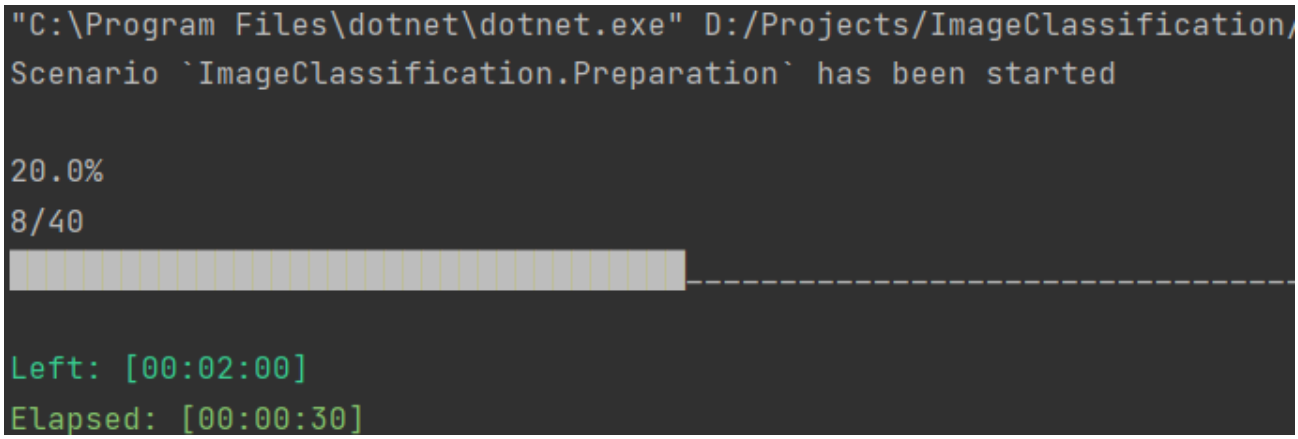

Рисунок 3.3 – Виконання парсингу зображень з відкритих джерел

Після виклику System API запит знаходиться в просторі ядра, потрапляючи до мережевої підсистеми ОС (наприклад, /net у ядрі Linux). Тут ОС оброблятиме мережевий запит асинхронно. Деталі можуть відрізнятися залежно від використовуваної ОС (виклик драйвера пристрою може бути запланований як сигнал, надісланий назад до середовища виконання, або може бути здійснено виклик драйвера пристрою, а потім сигнал надіслано назад), але в кінцевому підсумку під час виконання буде повідомлено, що запит на мережу виконується. У цей час робота для драйвера буде запланована, виконується або вже закінчена (запит уже надходить «по проводу»), але оскільки все це відбувається асинхронно, драйвер пристрою може займатися чимось іншим.

Наприклад, у Windows потік ОС здійснює виклик до драйвера мережевого пристрою та просить його виконати мережеву операцію через пакет запиту на переривання (IRP), який представляє операцію. Драйвер пристрою отримує IRP, здійснює виклик до мережі, позначає IRP як

[1)] [15] Task-based asynchronous pattern. URL: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap. (дата звернення 12.11.2021).

очікуваний і повертається назад до ОС. Оскільки потік ОС тепер знає, що IRP очікує, він не має більше роботи для цього завдання і повертається назад, щоб його можна було використовувати для виконання іншої роботи.

На основі того, що під одночасно під ОС може виконуватися велика кількість паралельних процесів, будемо розглядати алгоритм, який використовує SemaphoreSlim. Цей підхід дозволяє обмежувати використання завдань, та дозволяє пропускати лише необхідну оптимальну кількість. Така кількість, яка може бути оброблено одночасно на будь-якому комп’ютері.

```
async IAsyncEnumerable<ParsedImage> ParseAsync(ParseRequest
request, IProgress<ParseProgress> progress)
{
    if (request is null)
    {
        ThrowHelper.ArgumentNull(nameof(request));
    }

    if (request.EstimatedCount < 1)
    {

ThrowHelper.ArgumentOutOfRange(nameof(request.EstimatedCount),
request.EstimatedCount, "Value must be 1 or greater!");
    }

    var imagesPerCategory = (double)request.EstimatedCount /
request.Categories.Count();
    var capacity = (int)Math.Ceiling(imagesPerCategory /
_pageSize) * request.Categories.Sum(x => x.Keywords.Count());

    var throttler = new SemaphoreSlim(MaxThreads);
    var currentCount = 0;
    foreach (var category in request.Categories)
    {
        var keywordsCount = category.Keywords.Count();
        var imagesPerKeyword =
(int)Math.Ceiling(imagesPerCategory / keywordsCount);

        var allTasks = new List<Task>();
        var parsedImages = new
List<ParsedImage>(imagesPerKeyword);

        var total = (int)Math.Ceiling((double)imagesPerKeyword /
_pageSize);
        var pages = Enumerable.Range(_startFrom, total);

        var disposables = new List<IDisposable>();
        foreach (var keyword in category.Keywords)
        {
```

```
            await throttler.WaitAsync();
            allTasks.Add(
                Task.Run(async () =>
                {
                    try
                    {
                        var uri = new
Uri(_url).AddParameter("query", keyword);

                        foreach (var page in pages)
                        {
                            var pageUri =
uri.AddParameter("per_page", _pageSize)

.AddParameter("page", page);

                            var take = imagesPerKeyword - (page
- 1) * _pageSize;

                            var response = await
_httpClient.GetAsync<Response>(pageUri);

disposables.Add(response.Disposable);
                            var results =
response.Result.Results.Take(take);

                            foreach (var result in results)
                            {
                                var download = await
_httpClient.GetAsync(result.Links.Download);
                                var stream = await
download.Content.ReadAsStreamAsync();
                                var image =
Image.FromStream(stream);
                                var parsedImage = new
ParsedImage
                                {
                                    Category = category.Name,
                                    Image = image,
                                    Keyword = keyword
                                };
                                parsedImages.Add(parsedImage);
                                disposables.Add(download);
                                disposables.Add(stream);
                            }
                        }
                    }
                    finally
                    {
                        throttler.Release();
                    }
                }));
        }
        await Task.WhenAll(allTasks);

        foreach (var parsedImage in parsedImages)
        {
            var data = new ParseProgress
            {
                CurrentCount = ++currentCount,
```

```
                EstimatedCount = request.EstimatedCount
            };
            progress?.Report(data);
            yield return parsedImage;
        }

        foreach (var disposable in disposables)
        {
            disposable.Dispose();
        }
    }
}
```

Коли запит виконується і дані повертаються через драйвер пристрою, він повідомляє центральному процесору про нові дані, отримані через переривання. Спосіб обробки цього переривання залежить від ОС, але в кінцевому підсумку дані будуть передаватися через ОС, поки не досягнуть виклику взаємодії системи (наприклад, в Linux обробник переривань планує нижню половину IRQ для передачі даних через ОС асинхронно). Це також відбувається асинхронно. Результат ставиться в чергу, поки наступний доступний потік не зможе виконати асинхронний метод і розгорнути результат виконаного завдання.

Клас MLContext є відправною точкою для всіх операцій ML.NET. Ініціалізація MLContext створює нове середовище ML.NET, яке можна спільно використовувати між об’єктами робочого процесу створення моделі. Концептуально це схоже на DBContext в Entity Framework.

API класифікації зображень починає процес навчання (рис. 3.4), завантажуючи попередньо навчену модель TensorFlow [16][1)]. Навчальний процес складається з двох етапів:

- Фаза вузького місця
- Тренувальний етап

[1)] [16] Train a deep learning image classification model with ML.NET and TensorFlow. URL: https://docs.microsoft.com/en-us/samples/dotnet/machinelearning-samples/mlnet-image-classification-transfer-learning/. (дата звернення 18.11.2021).

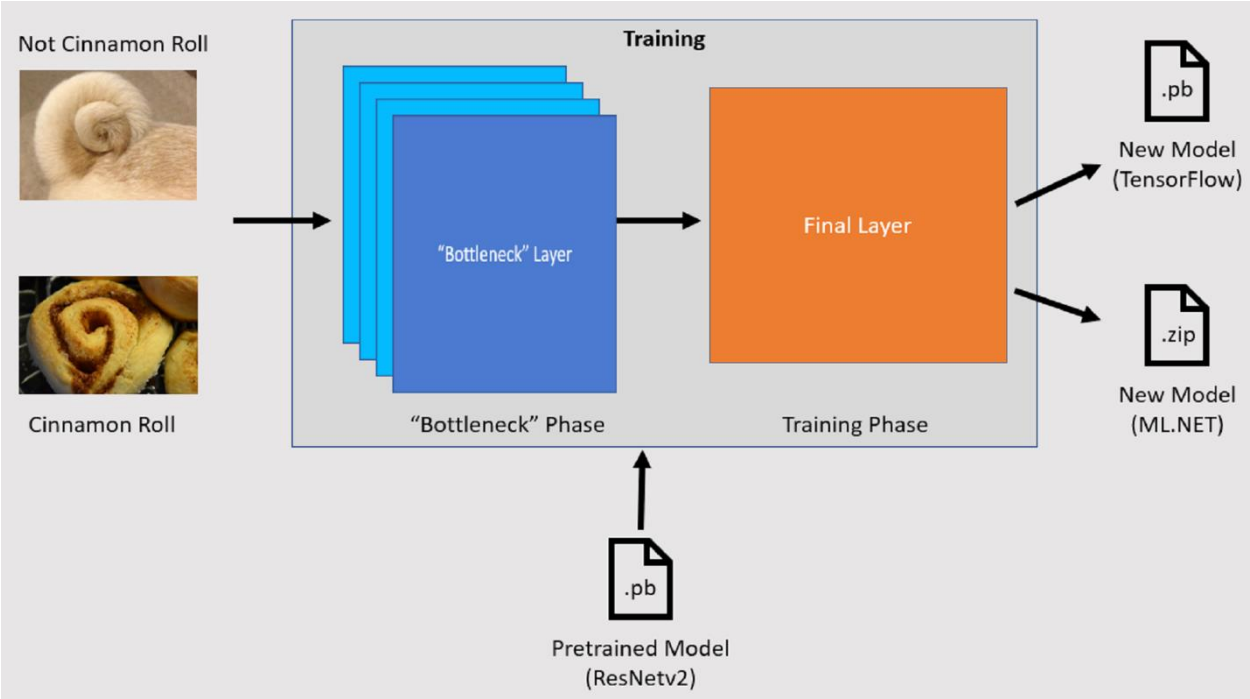

Рисунок 3.4 – Процес тренування моделі

Під час фази вузького місця завантажується набір навчальних зображень, а значення пікселів використовуються як вхідні дані або функції для заморожених шарів попередньо навченої моделі. Заморожені шари включають всі шари нейронної мережі аж до передостаннього шару, неофіційно відомого як шар вузького місця. Ці шари називаються замороженими, оскільки навчання на цих шарах не відбуватиметься, а операції передаються. Саме на цих заморожених шарах обчислюються шаблони нижнього рівня, які допомагають моделі розрізняти різні класи. Чим більша кількість шарів, тим більш інтенсивним є цей крок. На щастя, оскільки це одноразовий розрахунок, результати можна кешувати та використовувати в подальших запусках під час експериментування з різними параметрами.

Після обчислення вихідних значень фази вузького місця вони використовуються як вхідні дані для перенавчання останнього шару моделі. Цей процес є ітеративним і виконується кількість разів, визначену

параметрами моделі. Під час кожного запуску оцінюються втрати та точність. Потім вносяться відповідні коригування для покращення моделі з метою мінімізації втрат і максимальної точності. Після завершення навчання виводяться два формати моделі. Одна з них - це версія моделі .pb, а інша - серійна версія моделі .zip ML.NET. При роботі в середовищах, які підтримуються ML.NET, рекомендується використовувати версію моделі .zip. Однак у середовищах, де ML.NET не підтримується, у вас є можливість використовувати версію .pb.

Попередньо навчена модель, яка використовується в цьому посібнику, є 101-шаровим варіантом моделі залишкової мережі (ResNet) v2. Оригінальна модель навчена класифікувати зображення на тисячу категорій. Модель бере як вхідні дані зображення розміром 224 x 224 і виводить ймовірності класів для кожного з класів, на яких вона навчається. Частина цієї моделі використовується для навчання нової моделі за допомогою спеціальних зображень для прогнозування між двома класами.

Вище зазначений алгоритм підготовки зображень для розробки нейронної мережі. Цей алгоритм створює архів на зазначені теми, а також формує всі зображення по директоріям.

Після кроку підготовки необхідно зазначити наступний крок – крок тренування. Вихід першої фази подається як вхід на кінцевий шар моделі для її перенавчання. Кількість разів, коли це відбувається, визначається параметрами моделі. Кожна ітерація обчислює точність моделі та значення втрат залежно від того, які подальші оптимізації моделі можуть бути виконані. Наприкінці етапу навчання ми отримуємо формати .zip і .pb моделі. Переважно використовувати версію .zip у середовищі, що підтримує ML.NET.

Під час тренування використовуються та виконуються такі процеси:

– Ініціалізація;
– Розархівування;
– Підготовка набору даних;

– Завантаження зображень;

– Розділення даних;

– Визначення моделі;

– Навчання;

– Оцінення модель;

– Збереження моделі.

Дані завантажуються в тому ж порядку, в якому вони зчитуються з підкаталогів даних. Потрібно перемішувати дані, щоб додати дисперсію. Процес конвеєру виглядає таким чином:

```
/// <summary>
/// Trains neurone network and saves it.
/// </summary>
/// <param name="stream">Stream where is saved trained
model.</param>
/// <returns>Boolean flag of success.</returns>
public async Task<IEnumerable<string>> TrainAsync(Stream stream)
{
    var totalElapsed = new TimeSpan();

    if (MeasureTime)
    {
        Stopwatch = Stopwatch.StartNew();
    }

    if (File.Exists(Archive))
    {
        var unarchiving = StepCollection.GetStep<(string,
string), Task>(StepName.Unarchiving);
        await unarchiving.Execute((Archive, Folder));
        totalElapsed += Stopwatch.RestartPull();
    }

    var prepareDataSet = StepCollection.GetStep<(string,
MLContext), (IDataView DataSet, IEnumerable<string>
Classifications)>(StepName.PreparingDataSet);
    var shuffledDataSet = prepareDataSet.Execute((Folder,
mlContext));
    totalElapsed += Stopwatch.RestartPull();

    var loadImages = StepCollection.GetStep<(string, MLContext,
IDataView), IDataView>(StepName.LoadingImages);
    var inMemoryDataSet = loadImages.Execute((Folder, mlContext,
shuffledDataSet.DataSet));
    totalElapsed += Stopwatch.RestartPull();
```

```
    var splitData = StepCollection.GetStep<(MLContext,
IDataView, double), TrainTestData>(StepName.SplittingData);
    var trainTestData = splitData.Execute((mlContext,
inMemoryDataSet, TestFraction));
    totalElapsed += Stopwatch.RestartPull();

    Options.ValidationSet = trainTestData.TestSet;
    var define = StepCollection.GetStep<(MLContext, Options),
IEstimator<ITransformer>>(StepName.DefiningModel);
    var pipeline = define.Execute((mlContext, Options));
    totalElapsed += Stopwatch.RestartPull();

    var train =
StepCollection.GetStep<(IEstimator<ITransformer>, IDataView),
ITransformer>(StepName.Trainning);
    var trainedModel = train.Execute((pipeline,
trainTestData.TrainSet));
    totalElapsed += Stopwatch.RestartPull();

    if (UseEvaluation)
    {
        var evaluate = StepCollection.GetStep<(MLContext,
IDataView, ITransformer),
MulticlassClassificationMetrics>(StepName.EvaluatingModel);
        var metrics = evaluate.Execute((mlContext,
trainTestData.TestSet, trainedModel));
        MulticlassMetricsUpdated?.Invoke(metrics);
        totalElapsed += Stopwatch.RestartPull();
    }

    var save = StepCollection.GetStep<(MLContext, ITransformer,
DataViewSchema, Stream), bool>(StepName.SavingModel);
    var success = save.Execute((mlContext, trainedModel,
trainTestData.TrainSet.Schema, stream));
    totalElapsed += Stopwatch.RestartPull();

    if (MeasureTime)
    {
        Stopwatch.Stop();
        _progress?.Invoke(new TrainProgress
        {
            Message = $"Total training took: {totalElapsed}",
            Status = StepStatus.Finished
        });
        Stopwatch = null;
    }

    var classifications =
shuffledDataSet.Classifications.ToHashSet();
    using (var zip = new ZipArchive(stream,
ZipArchiveMode.Update))
    {
        var entry = zip.CreateEntry(ClassificationsFileName);
        using var entryStream = entry.Open();
        using var writer = new StreamWriter(entryStream);
        writer.Write(string.Join(Environment.NewLine,
classifications));
    }
    return classifications;
}
```

Model Builder створює навчену модель разом із кодом, необхідним для завантаження моделі та початку прогнозування. Моделі ML.NET зберігаються як файл .zip, а код для завантаження та використання моделі додається як новий проект у ваше рішення. ML.NET Model Builder також додає зразок програми, яку можна використовувати, щоб побачити свою модель у дії.

Він також завантажує код для перенавчання моделі з новим набором даних, якщо потрібно перенавчати цю модель з коду, не використовуючи інтерфейс ML.NET Model Builder.

Після використання сценарію можна побачити журнал записів, в якому описаний кожний крок, а також зібрані статистичні та аналітичні дані:

```
Scenario `ImageClassification.Train` has been started
Downloading archive:
D:\Projects\ImageClassification\ImageClassification.Train\assets
\inputs\images\data-set.zip already exist.
-------------------------------
[12/7/2021 10:04:39 PM]
Current step: Unarchiving
Status: Started
Message: Started unarchiving process for archive:
`D:\Projects\ImageClassification\ImageClassification.Train\asset
s\inputs\images\data-set.zip`
-------------------------------
-------------------------------
[12/7/2021 10:04:39 PM]
Current step: Unarchiving
Status: Finished
Message: Finished unarchiving
Step took: 00:00:00.2024021
-------------------------------
-------------------------------
[12/7/2021 10:04:39 PM]
Current step: PreparingDataSet
Status: Started
Message: Preparing data set
-------------------------------
-------------------------------
[12/7/2021 10:04:39 PM]
Current step: PreparingDataSet
Status: Finished
Message: Data set is prepared
Step took: 00:00:00.0941261
-------------------------------
-------------------------------
```

```
[12/7/2021 10:04:39 PM]
Current step: LoadingImages
Status: Started
Message: Loading images in memory
------------------------------
------------------------------
[12/7/2021 10:04:40 PM]
Current step: LoadingImages
Status: Finished
Message: Loading images in memory has been finished
Step took: 00:00:00.2239529
------------------------------
------------------------------
[12/7/2021 10:04:40 PM]
Current step: SplittingData
Status: Started
Message: Data splitting is started
------------------------------

------------------------------
[12/7/2021 10:04:40 PM]
Current step: SplittingData
Status: Finished
Message: Data has been splitted by formula Train/Test - 90/10
Step took: 00:00:00.0257715
------------------------------
------------------------------
[12/7/2021 10:04:40 PM]
Current step: DefiningModel
Status: Started
Message: Model will be defined with options:
Epoch=200
Architecture=InceptionV3
BatchSize=10
LearningRate=0.01
FeatureColumnName=Image
LabelColumnName=Image

------------------------------
------------------------------
[12/7/2021 10:04:40 PM]
Current step: DefiningModel
Status: Finished
Message: Model is defined
Step took: 00:00:00.5714504
------------------------------
------------------------------
[12/7/2021 10:04:40 PM]
Current step: Trainning
Status: Started
Message: Started training process
------------------------------
------------------------------
[12/7/2021 10:04:56 PM]
Current step: Trainning
Status: Finished
Message: Finished training process
Step took: 00:00:15.8300634
------------------------------
------------------------------
```

```
[12/7/2021 10:04:56 PM]
Current step: EvaluatingModel
Status: Started
Message: Started evaluating model...
------------------------------
------------------------------
[12/7/2021 10:04:58 PM]
Current step: EvaluatingModel
Status: Finished
Message: Finished evaluating model
Step took: 00:00:02.1462656
------------------------------
==============================

Confusion table
          ||=======================
PREDICTED ||  cars |  cats |  dogs | Recall
TRUTH     ||=======================
     cars ||     1 |     0 |     0 | 1.0000
     cats ||     0 |     0 |     0 | 0.0000
     dogs ||     0 |     1 |     0 | 0.0000
          ||=======================
Precision ||1.0000 |0.0000 |0.0000 |

==============================
------------------------------
[12/7/2021 10:04:58 PM]
Current step: SavingModel
Status: Started
Message: Started saving model...
------------------------------
------------------------------
[12/7/2021 10:05:02 PM]
Current step: SavingModel
Status: Finished
Message: Finished saving model
Step took: 00:00:03.9898888
------------------------------
------------------------------
Message: Total training took: 00:00:23.1143638
------------------------------
Scenario `ImageClassification.Train` has been finished
successfully
```

Щоб покращити модель, можна спробувати покращити її продуктивність, спробувавши деякі з наступних підходів:

– Більше даних: чим більше прикладів вивчає модель, тим краще вона працює. Можна використовувати повний набір даних SDNET2018 для навчання.

– Збільшення даних: поширеною технікою внесення різноманітності до даних є збільшення даних шляхом отримання зображення та застосування різних перетворень (поворот, перевертання, зсув, обрізання). Це додає більше різноманітних прикладів, на яких можна вчитися моделі.

– Довше тренування: чим довше модель тренується, тим більш налаштованим вона буде. Збільшення кількості епох може покращити продуктивність моделі.

– Змінення параметрів: на додаток до вказаних параметрів, можна налаштувати інші параметри, щоб потенційно підвищити продуктивність. Зміна швидкості навчання, яка визначає величину оновлень моделі після кожної епохи, може значно покращити продуктивність.

– Використання іншої архітектури моделі: залежно від того, як виглядають дані, модель, яка найкраще вивчає її функції, може відрізнятися. Якщо продуктивність моделі низька, можливо потрібно буде змінити архітектуру.

4 ІНСТРУКЦІЯ КОРИСТУВАННЯ СЕРВІСОМ

Після проведення ряду робіт аналізу була виконана мета розробки даного проекту – реалізоване API для категоризації та клеймування зображень. Створений проект є повноцінним програмним продуктом, який повністю готовий до користування.

Для початку роботи тереба відкрити папку Publish та запустити файл ImageClassification.API.exe (рис. 4.1). Відкриється консоль, яку не можна закривати:

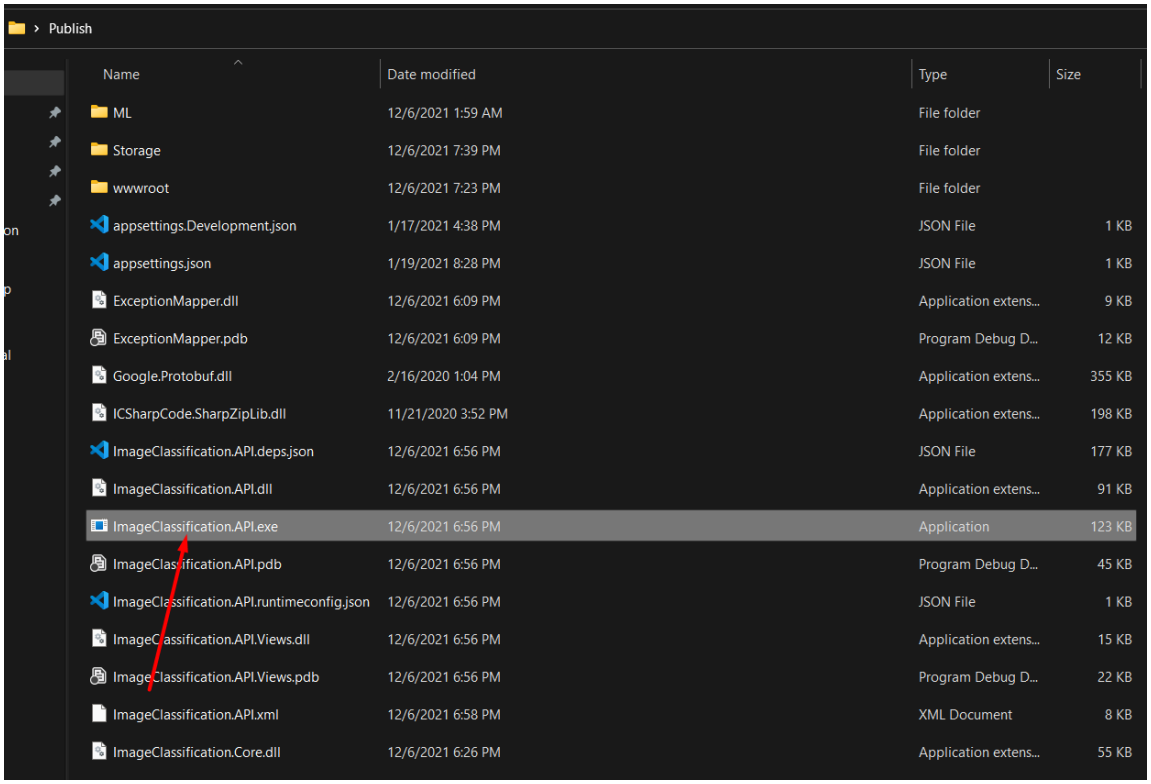

Рисунок 4.1 – Папка Publish

Далі треба перейти за посиланням – http://localhost:5000, після чого відкриється головне вікно додатку на вкладці Storage (рис. 4.2).

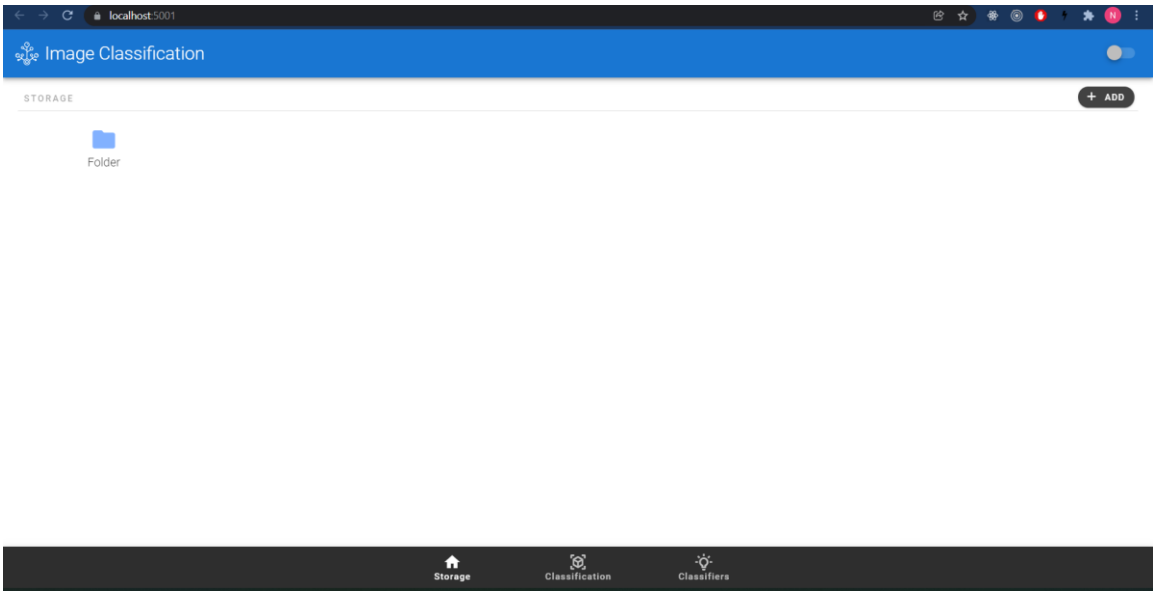

Рисунок 4.2 – Головне вікно додатку

На цій сторінці є можливість управління папками (редагування, створення нових папок). Натиснувши правою кнопкою, можна відкрити меню керування папками (рис. 4.3):

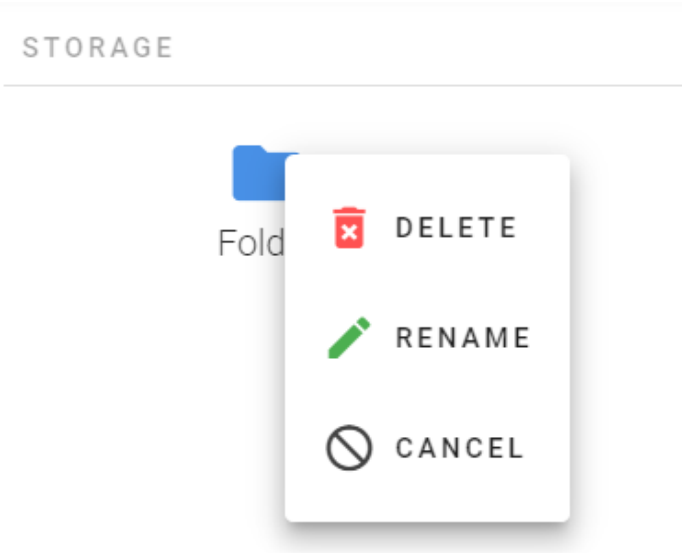

Рисунок 4.3 – Меню керування папками

Щоб додати нову папку для тренувань – потрібно натиснути на чорну кнопку Add, після чого з'явиться нова іконка та текстове поле, в яке потрібно ввести ім'я та натиснути Enter або галочку (рис. 4.4). Після чого буде створено нову папку, яка буде відображатись у списку (рис. 4.5).

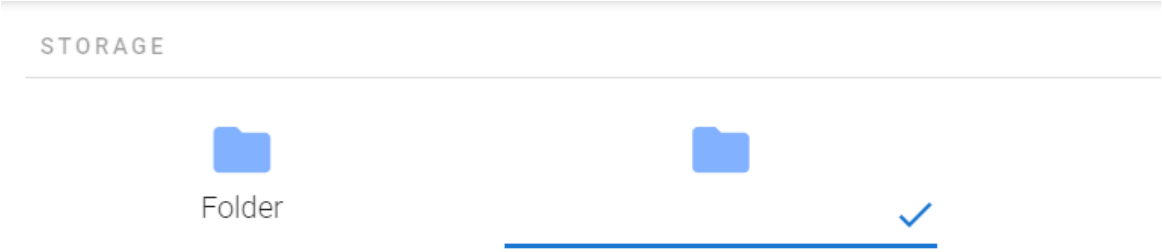

Рисунок 4.4 – Створення нової папки

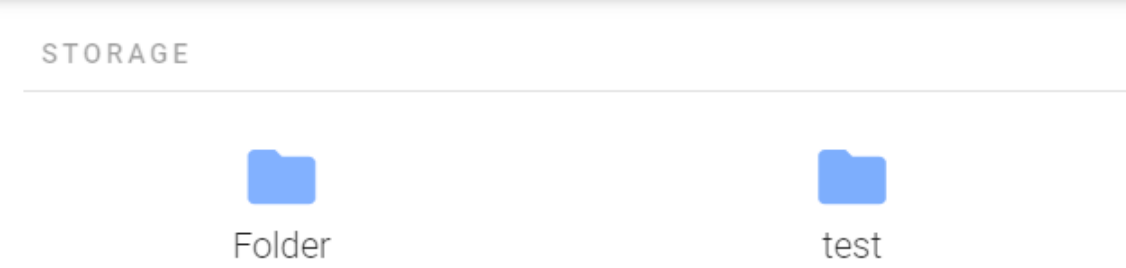

Рисунок 4.5 – Перелік доступних папок

У цьому вікні папки Folder можна створювати папки із зображеннями, які використовуються для тренування нейронної мережі. За замовчуванням створено папку Folder – вона вже готова для роботи, там знаходяться зображення для трьох категорій - собаки, машини, кішки (рис. 4.6):

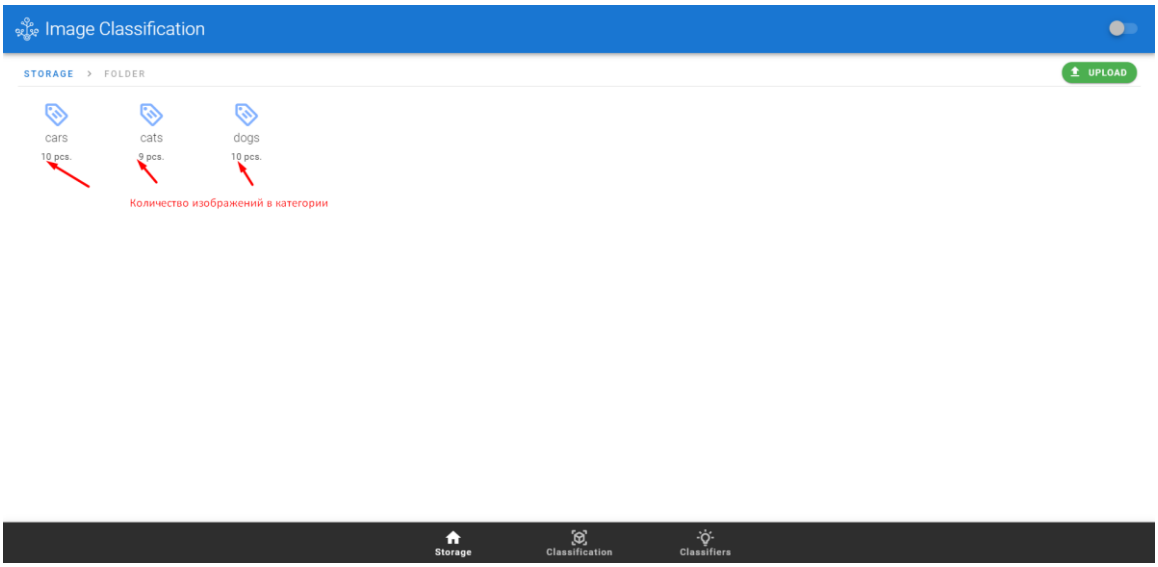

Рисунок 4.6 – Папка Folder

Для додавання нової категорії та зображення в ній потрібно натиснути на зелену кнопку Upload у верхньому правому куті (рис. 4.7).

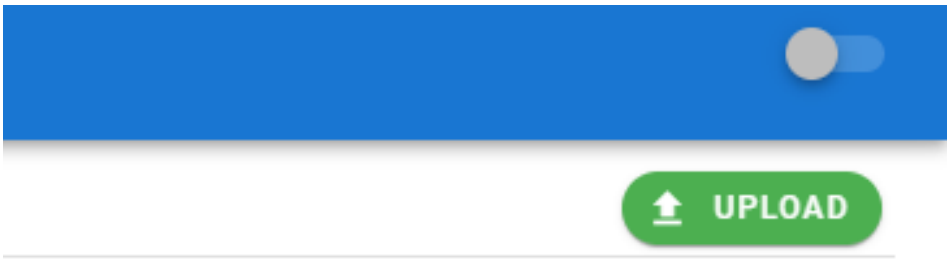

Рисунок 4.7 – Кнопка Upload

У вікні вибираємо назву категорії (Label) і зображення (може бути більше одного) і натискаємо Upload. У такий спосіб створюється категорія з новою назвою (рис. 4.8). Якщо ми хочемо доповнити категорію, досить просто вказати ім'я.

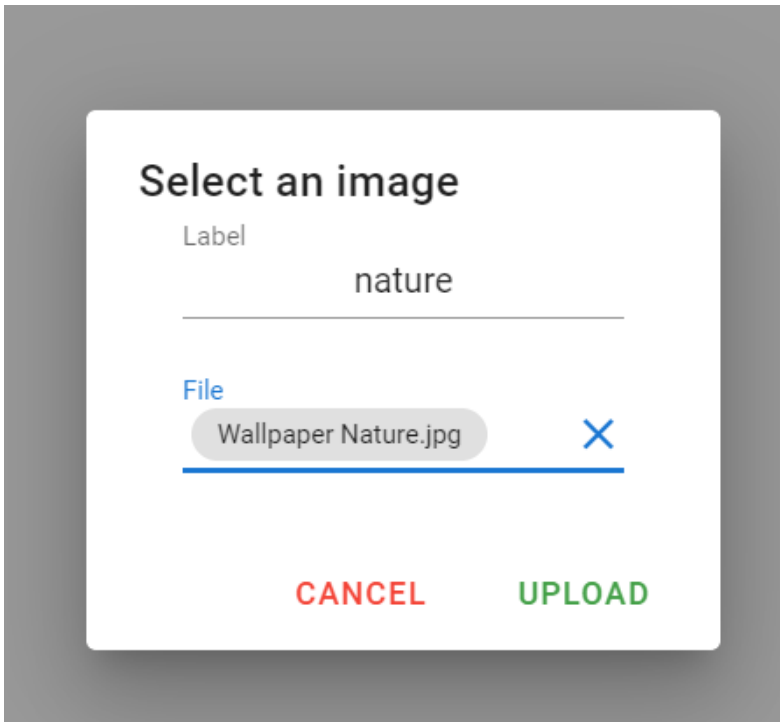

Рисунок 4.8 – Вікно створення нової категорії

Після додавання бачимо, що було додано нову категорію з 1 зображенням (рис. 4.9). Категорії можна видаляти та створювати нові без обмежень. Видалення відбувається натисканням правої кнопки миші на обрану категорію та натиснувши Delete.

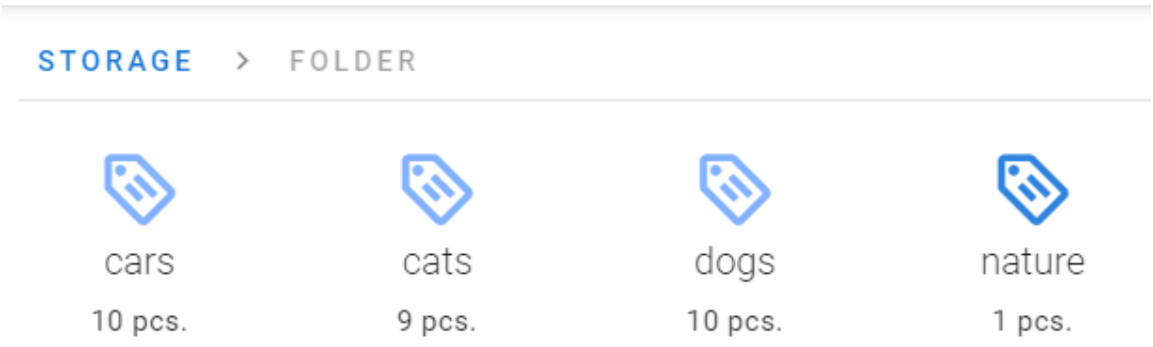

Рисунок 4.9 – Створені категорії для навчання нейронної мережі

Перейшовши на вкладку Classifiers (рис. 4.10) є змога створювати моделі нейронних мереж, ґрунтуючись на папках з категоріями та зображеннями. Щоб створити нову, потрібно натиснути кнопку Prepare (рис. 4.8):

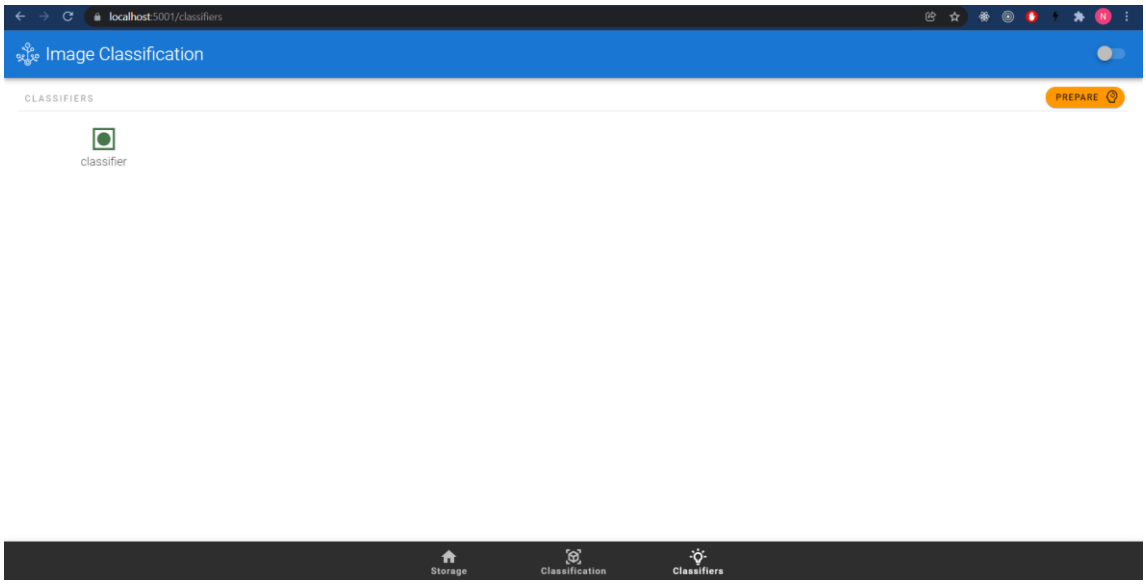

Рисунок 4.10 – Вкладка Classifiers

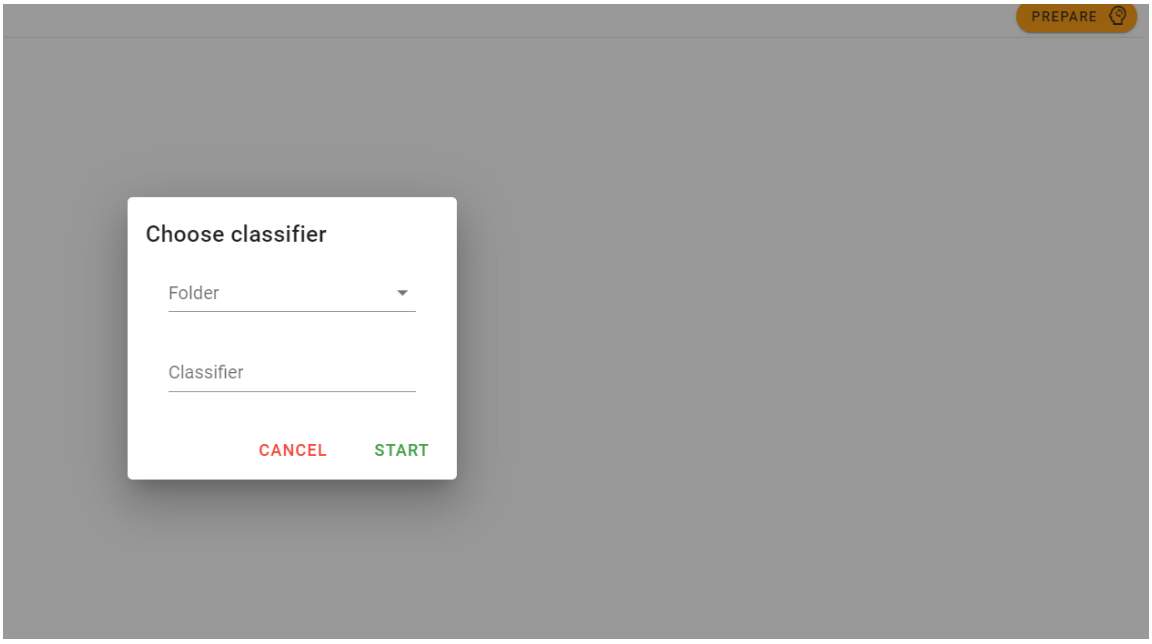

Рисунок 4.11 – Створення моделі нейронних мереж

Після цього необхідно вибрати папку, яка використовуватиметься для тренування моделі, та ім'я нової моделі. Ім'я повинне бути унікальним, тому не повинно повторюватися (рис. 4.12). Після чого натискаємо Start і з'являється вікно завантаження (рис. 4.13).

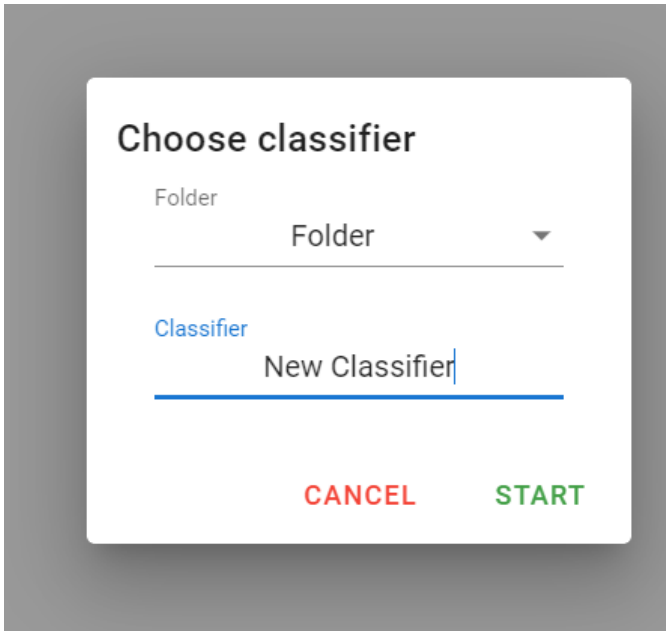

Рисунок 4.12 – Приклад заповнення форми для створення моделі нейронних мереж

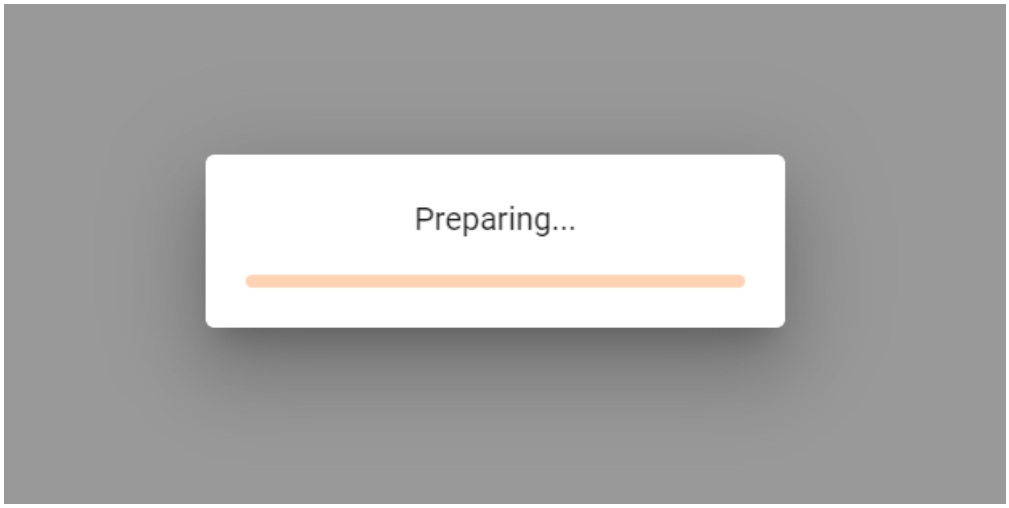

Рисунок 4.13 – Вікно завантаження нової моделі

Для того, щоб переглянути логи тренування (рис. 4.14), потрібно перейти за адресою – https://localhost:5001/trainlogs. І як тільки в логах з'явилося повідомлення Finished (рис. 4.15) – модель готова до роботи.

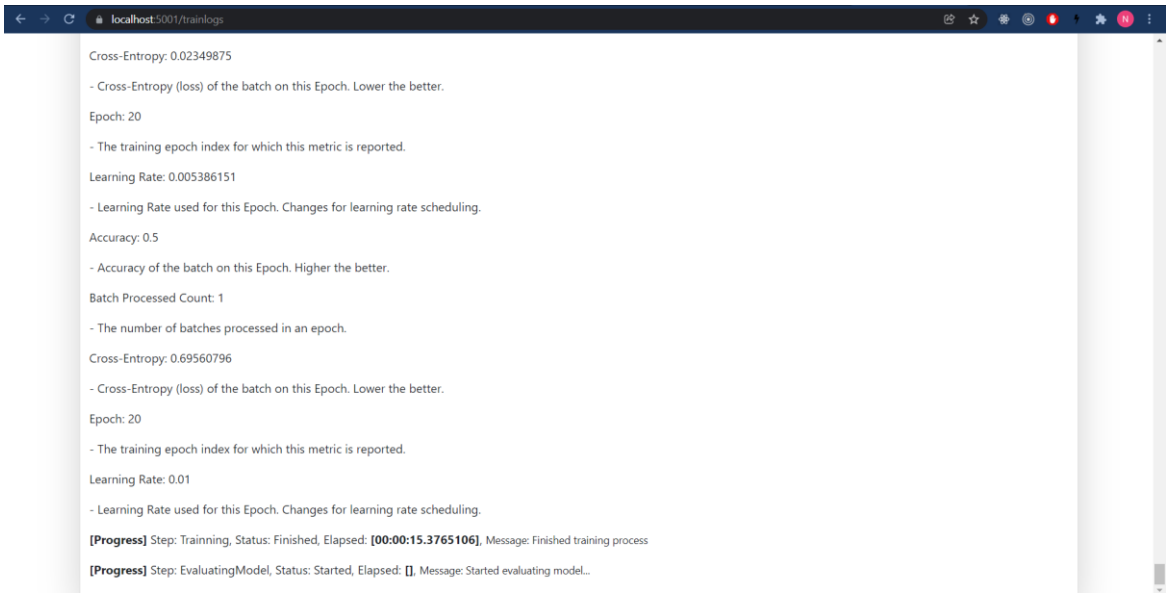

Рисунок 4.14 – Логи тренування нейронної мережі

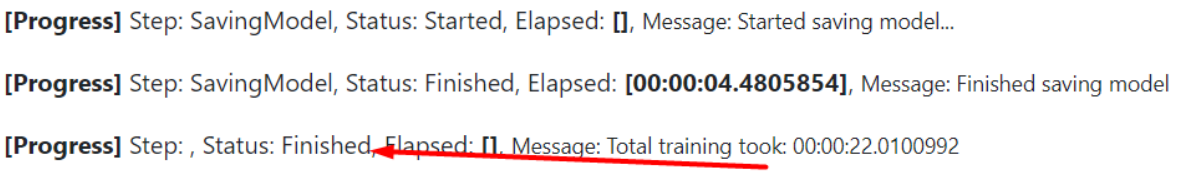

Рисунок 4.15 – Сповіщення про готовність моделі до роботи

Після чого новий класифікатор буде додано до списку (рис. 4.16).

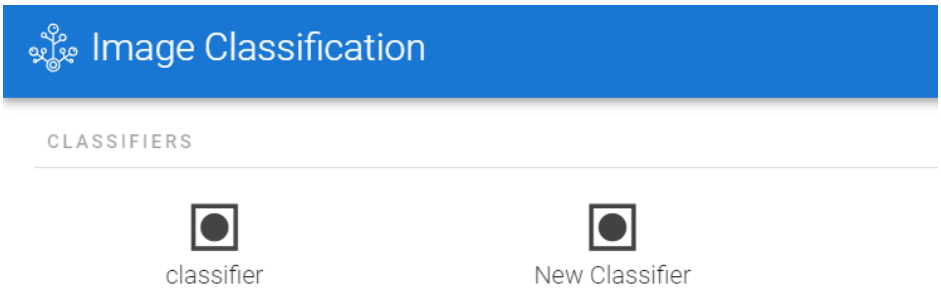

Рисунок 4.16 – Доступні моделі нейронних мереж

Потім заходимо на вкладку Classification (рис. 4.17). Вибираємо classifier і зображення (можна зробити з телефону, завантажити з комп'ютера або зробити в прямому ефірі), після чого потрібно натиснути кнопку Recognize, яка з'явилася вгорі (рис. 4.18). Після чого починається процес сканування:

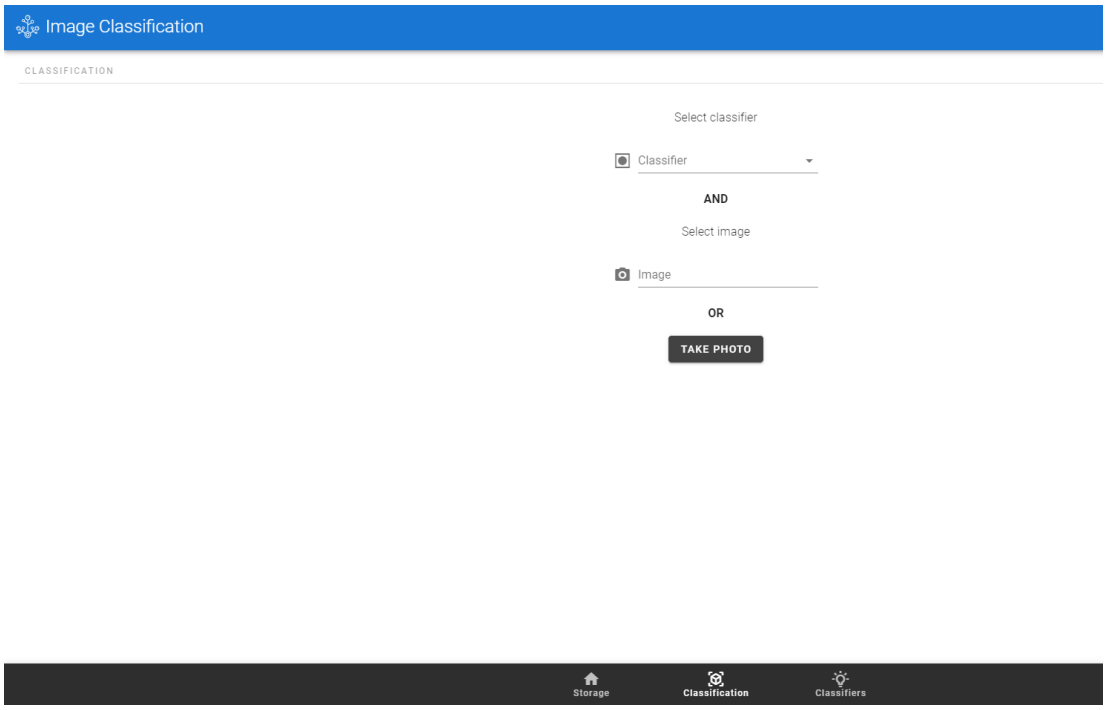

Рисунок 4.17 – Вкладка Classification

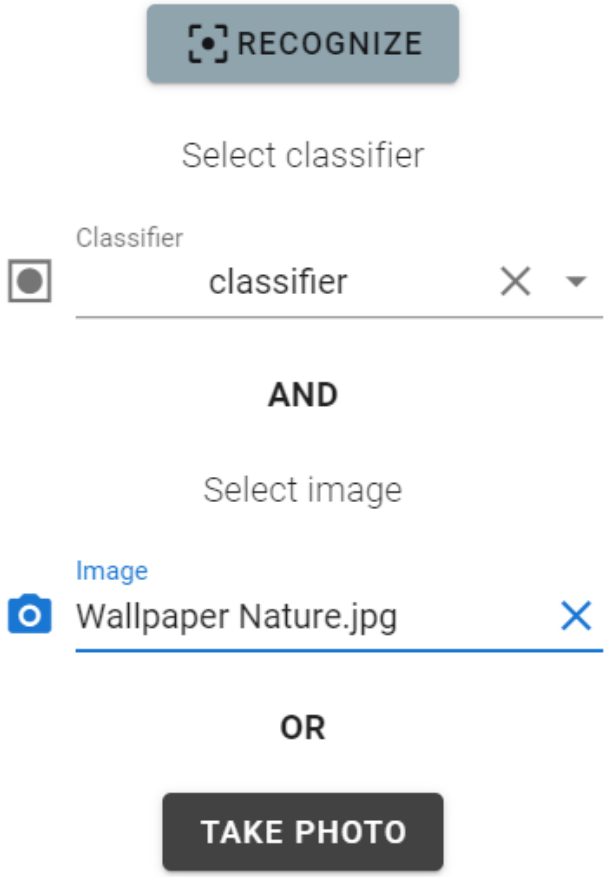

Рисунок 4.18 – Вкладку Classification

Після того як завершиться процес сканування – повертається результат (4.19).

Важливо: перший раз система може дуже довго вантажити результат – вона завантажує всі необхідні пакети для роботи, після чого буде в рази швидше (загалом 3-5 секунд). Вперше може тривати до 15 хвилин.

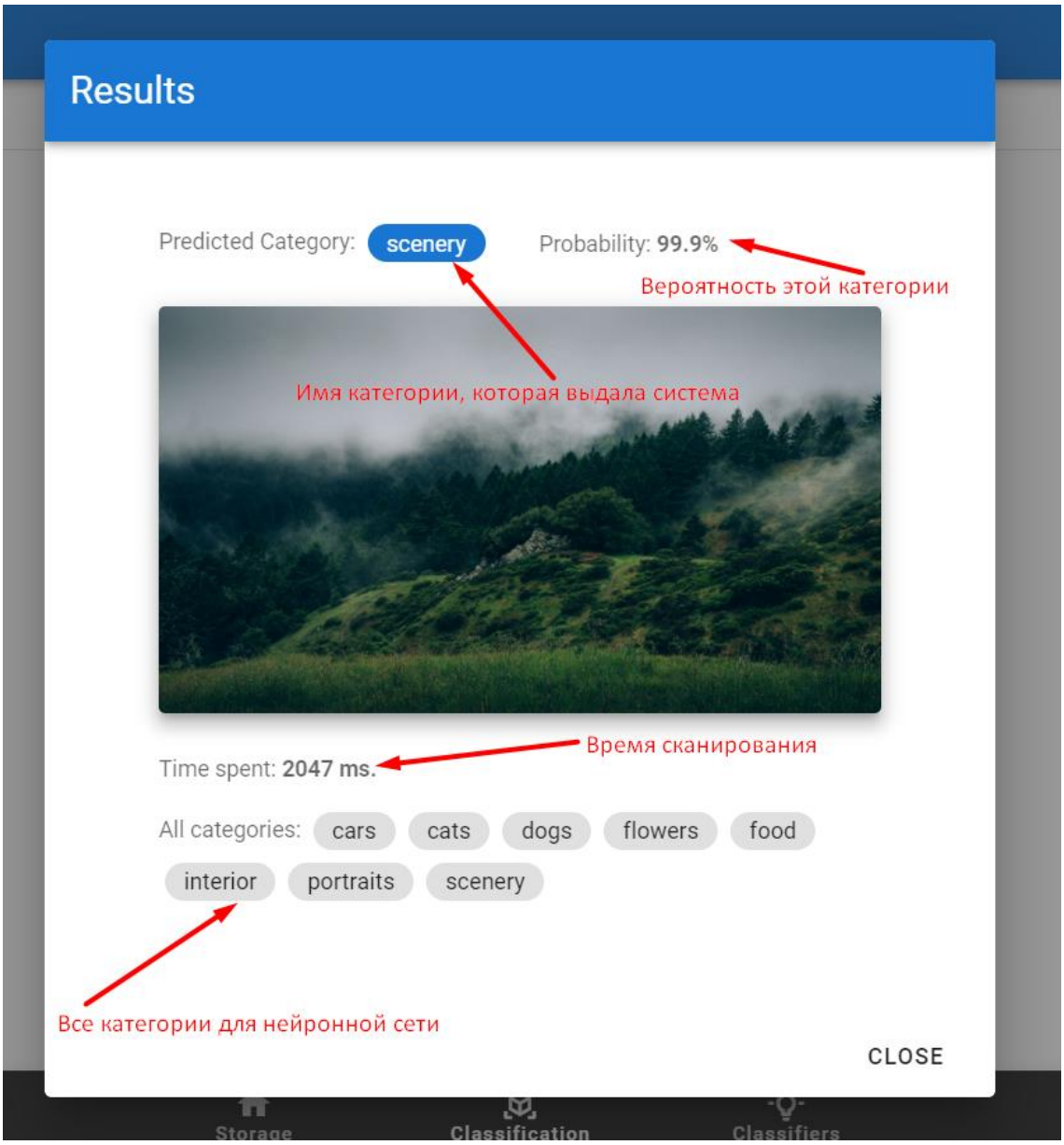

Рисунок 4.19 – Результат сканування

У додатку також реалізована функція зміни режиму фону – світлий (рис. 4.20) або темний (рис. 4.21). це надає змогу користувачеві обрати більш комфортний для варіант для використання додатку.

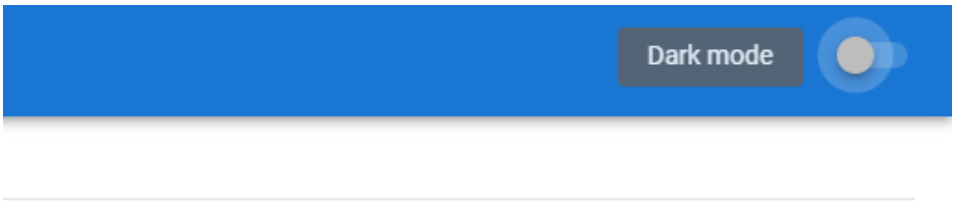

Рисунок 4.20 – Світлий режим

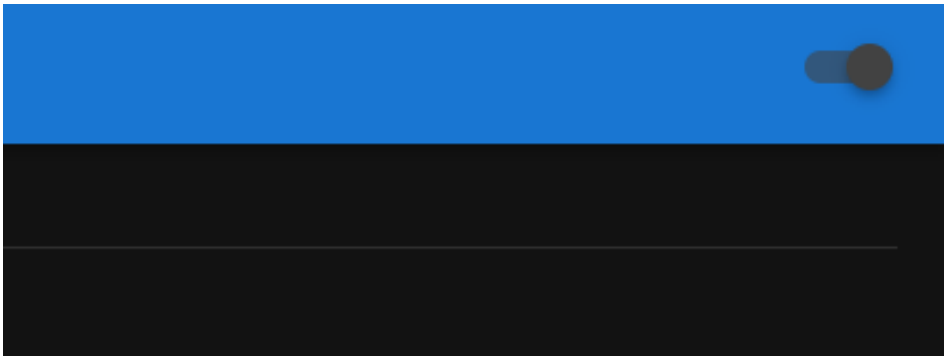

Рисунок 4.21 – Темний режим

ВИСНОВКИ

Розпізнавання зображень використовується для виконання багатьох візуальних завдань на основі машин, таких як позначення вмісту зображень мета-тегами, виконання пошуку вмісту зображень та керування автономними роботами, самокерованими автомобілями та системами уникнення аварій.

У той час як мозок людини і тварин легко розпізнає об’єкти, комп’ютери мають труднощі з цим завданням. Програмне забезпечення для розпізнавання зображень вимагає глибокого навчання.

Нинішні та майбутні застосування розпізнавання зображень включають розумні бібліотеки фотографій, цільову рекламу, інтерактивність медіа, доступність для людей з вадами зору та розширені дослідницькі можливості.

Під час реалізації проекту, було зазначено мету: створити системи для динамічного створення моделей для праці множини нейронних мереж, які змогли би класифікувати зображення та видавати теги. Ця мета була цілком виконана, крім того були створені додаткові функції, такі як: розширення створених баз знань, завантаження та автоматичне формування архівів зображень для тренування, визначення точності та доцільності моделі.

Також було відтворено комунікаційну модель, завдяки якій можна гнучко налаштовувати процес тренування та оцінення. Сам процес створення моделі включає в себе дев’ять кроків:

– Ініціалізація;
– Розархівування;
– Підготовка набору даних;
– Завантаження зображень;
– Розділення даних;
– Визначення моделі;
– Навчання;
– Оцінювальна модель;

– Збереження моделі.

Кожен крок дозволяє автоматизувати процес навчання моделі та зробити легшим процес налаштування для звичайного користувача, підвищує ефективність тренування, а також знижує час формування необхідної кількості зображень.

Середній час завантаження та архівації моделі, яка складається з тисячі зображень складає менше однієї хвилини. Такі показники можливі завдяки паралелізації завдань, а також багатопотоковості. Крім того, слід зазначити результати тренування та аналізу моделей: середній час тренування моделі, яка складається з однієї тисячі зображень сягає 3 хвилин, такий час дозволяє створювати до 500 моделей на добу в послідовному процесі, або до 2000 моделей на обчислювальних машинах з 16 потоками.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series), ISBN-13: 978-0262018029, ISBN-10: 0262018020. (дата звернення 03.09.2021).
2. Perceptrons: An Associative Learning Network. URL: https://ei.cs.vt.edu/~history/Perceptrons.Estebon.html. (дата звернення 04.09.2021).
3. Neural Networks and Deep Learning: A Textbook 1st ed. 2018 Edition, ISBN-13: 978-3319944623, ISBN-10: 3319944622. (дата звернення 13.09.2021).
4. VGG16 – Convolutional Network for Classification and Detection. URL: https://neurohive.io/en/popular-networks/vgg16/. (дата звернення 16.09.2021).
5. Deep Learning: GoogLeNet Explained. URL: https://towardsdatascience.com/deep-learning-googlenet-explained-de8861c82765. (дата звернення 22.09.2021).
6. ResNet (34, 50, 101): «остаточные» CNN для классификации изображений. URL: https://neurohive.io/ru/vidy-nejrosetej/resnet-34-50-101/. (дата звернення 22.09.2021).
7. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems 2nd Edition, ISBN-13: 978-1492032649, ISBN-10: 1492032646. (дата звернення 02.10.2021).
8. Programming ML.NET 1st Edition, ISBN-13: 978-0137383658, ISBN-10: 0137383657. (дата звернення 07.10.2021).
9. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, 2nd Edition. (дата звернення 10.10.2021).

10. Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming 10th ed. Edition, ISBN-13: 978-1484269381, ISBN-10: 1484269381. (дата звернення 21.10.2021).
11. Front-End Development Projects with Vue.js: Learn to build scalable web applications and dynamic user interfaces with Vue 2, ISBN-13: 978-1838984823, ISBN-10: 1838984828. (дата звернення 27.10.2021).
12. Entity Framework Core. URL: https://docs.microsoft.com/en-us/ef/core/. (дата звернення 29.10.2021).
13. Understanding async/await State Machine in .NET. URL: https://mykkon.work/async-state-machine/. (дата звернення 03.11.2021).
14. Exploring the async/await State Machine – Main Workflow and State Transitions. URL: https://vkontech.com/exploring-the-async-await-state-machine-main-workflow-and-state-transitions/. (дата звернення 09.11.2021).
15. Task-based asynchronous pattern. URL: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap. (дата звернення 12.11.2021).
16. Train a deep learning image classification model with ML.NET and TensorFlow. URL: https://docs.microsoft.com/en-us/samples/dotnet/machinelearning-samples/mlnet-image-classification-transfer-learning/. (дата звернення 18.11.2021).