Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування вищого навчального закладу)
Навчально-науковий інститут інформаційних технологій та робототехніки
(повна назва інституту)
Кафедра комп'ютерних та інформаційних технологій і систем
(повна назва кафедри)

Пояснювальна записка
до дипломної роботи
магістра
(рівень вищої освіти)
на тему

Розпізнавання зображень за допомогою нейронних мереж з використанням бібліотеки TensorFlow

Виконав: студент 6 курсу, групи 6 ТН
спеціальності
122 Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)

Семенюта І.Г.
(прізвище та ініціали)
Керівник Мавріна М.О.
(прізвище та ініціали)
Рецензент
(прізвище та ініціали)

Полтава – 2021 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І РОБОТОТЕХНІКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

спеціальність 122 «Комп'ютерні науки»

на тему

«Розпізнавання зображень за допомогою нейронних мереж з використанням бібліотеки TensorFlow»

Студента групи 601-ТН Семенюти Ігора Григоровича

Керівник роботи
кандидат технічних наук
Мавріна М.О.

Завідувач кафедри
кандидат технічних наук,
доцент Головко Г.В.

Полтава – 2021

РЕФЕРАТ

Кваліфікаційна робота магістра: 88 с., 29 малюнків, 2 додатки, 37 джерел.

Об’єкт дослідження: процес розпізнавання зображення глибокою нейронною мережею.

Мета роботи: розроблення мобільного додатку із вбудованою нейронною мережею для розпізнавання об’єктів на зображенні.

Методи: проектування та розробка додатку, навчання нейронної мережі.

Ключові слова: класифікація об’єктів, нейронна мережа, глибоке навчання, алгоритм зворотного поширення помилок, TensorFlow, Theano, Keras.

ABSTRACT

Qualifying work of the master: 88 pages, 29 drawings, 2 appendices, 37 sources.

Object of research: the process of image recognition by a deep neural network.

Objective: To develop a mobile application with a built-in neural network for recognizing objects in the image.

Methods: application design and development, neural network training.

Keywords: object classification, neural network, deep learning, error backpropagation algorithm, TensorFlow, Theano, Keras.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект.

AI – Artificial Intelligence.

ANN – Artificial neural networks.

ООП – Об'єктно-орієнтоване програмування.

CNN – Convolutional neural network.

CPU – Central processing unit.

GPU – Graphics processing unit.

API – Application programming interface.

FFNN – Feed forward neural networks.

RNN – Recurrent neural network.

ResNet – Recurrent neural network.

LSTM – Long short-term memory.

ШНМ – Штучна нейронна мережа.

DCNN – Deep convolutional neural networks.

ВСТУП

Нині в умовах інформаційного суспільства конкурентна перевага вже не визначається розміром країни чи її природними ресурсами. Тепер усе визначається рівнем освіти та обсягом накопичених суспільством знань. Найближчим часом, за певних обставин, країна, яка зараз буде процвітати, зможе перевершити інші країни у створенні та засвоєнні нових знань. Особливу роль у цьому відіграли інформаційно-комунікаційні технології, особливо методи та засоби штучного інтелекту [1].

В останнє десятиліття машинне навчання стало набирати все більшої і більшої популярності і сьогодні займає одну з лідируючих позицій в сфері інформаційних технологій. Обсяг цифрових даних, використовуваний нами в повсякденному житті, збільшується з кожним днем, в зв'язку з цим, виникла необхідність в автоматичному аналізі смарт-даних для подальшого розвитку науково-технічного прогресу.

Сьогодні машинне навчання активно використовується для розпізнавання мови та образів, а також на його основі побудовано багато пошукових систем. Машинне навчання настільки міцно увійшло в наше повсякденне життя, що щоденне використання його методів залишається для нас, як правило, непоміченим.

На сьогодні існує позитивна динаміка розробки та впровадження елементів штучного інтелекту у більшість типів програмного забезпечення: мобільні додатки, інформаційні системи, електронні пристрої, тощо. мобільні додатки, інформаційні системи, електронні пристрої тощо. Цей «інтелектуальний» процес дозволяє говорити про поступове зростання інтелекту сучасних комп’ютерних систем, які можуть виконувати функції, які традиційно вважаються інтелектуальними: розуміння мови, логічне міркування, використання накопичених знань, навчання, розпізнавання образів і навчання і пояснювати своє рішення [1].

Основною метою машинного навчання є навчання комп'ютера таким чином, щоб він самостійно, без допомоги людини, міг приймати рішення, засновані на попередньому досвіді, для вирішення поставленого перед ним завдання. Існуючі алгоритми глибокого навчання допомагають аналізувати дані продажів в минулому для передбачення поведінки клієнтів в майбутньому, оптимізувати поведінку робота для мінімізації використовуваних ресурсів та видавати дані з біологічних джерел інформації.

Перевагою використання нейронних мереж є можливість їх застосування для добування інформації з великих або неточних даних, виявлення основних ознак і тенденцій, що не можуть бути розпізнані людиною і іншими методами. Навчена нейронна мережа може виступати в якості «експерта» для тієї категорії даних, для якої вона була підготовлена. Більш того, подібного роду структура в подальшому може допомогти прогнозувати нові результати.

РОЗДІЛ 1
ОГЛЯД ПІДХОДІВ ТА МЕТОДІВ, ЩО ЗАСТОСОВУЮТЬСЯ ДЛЯ КЛАСИФІКАЦІЇ ОБ'ЄКТІВ НА ЗОБРАЖЕННІ

1.1 Задача класифікації об'єктів

Масове виробництво та поширення різноманітних гаджетів, облаштованих високопродуктивними мікропроцесорами та високоякісними відеокамерами, а також зростаюча потреба в автоматичному розпізнаванні різноманітних об'єктів викликають науковий інтерес до вирішення задачі класифікації об'єктів із використанням сучасних програмних рішень.

Задача класифікації об'єктів – це задача розбиття множини об'єктів або спостережень на апріорно задані групи, називані класами, всередині кожного з яких вони вважаються схожими один на одного, та мають приблизно однакові властивості й ознаки. При цьому рішення здійснюється на основі аналізу значень атрибутів (ознак)[2].

Можна виділити наступні способи класифікації об'єктів з використанням різного типу ознак:

з використанням характеристик об'єктів:

− форма: ці методи використовують двовимірну просторову інформацію про об'єкти. Характерними ознаками, що використовуються у схемах класифікації за формою, є точки (центроїд, безліч точок), примітивні геометричні фігури (прямокутник або еліпс), скелет, силует та контур.

− рух: ці методи використовують тимчасові відстежувані особливості об'єктів для класифікації

на підставі використовуваної навчальної вибірки:

− контрольована класифікація: процес використання зразків відомих класів (навчальних наборів) для класифікації пікселів невідомої сутності. Приклади: алгоритм мінімальної відстані до середніх, алгоритм паралелепіпеда, алгоритм максимальної правдоподібності.

− неконтрольована класифікація: у цьому типі класифікації використовується метод, який досліджує велику кількість невідомих пікселів і ділить його на кількість класів на основі природних угруповань, які у зображенні. Комп'ютер визначає класи, що спектрально розділяються, а потім визначає їх значення для зображення. Жодних великих попередніх знань не потрібно. приклад: алгоритм кластеризації K-means.

на підставі припущення параметрів даних:

− параметричний класифікатор: використовуються такі параметри, як середня векторна та коваріаційна матриця. Існує припущення про гаусівський розподіл. Параметри, такі як середня векторна та коваріаційна матриця, часто генеруються з тренувальних даних. Приклади: максимальна правдоподібність, лінійний дискримінантний аналіз.

− непараметричний класифікатор: немає припущень щодо даних. Непараметричні класифікатори не використовують статистичні параметри обчислення поділу класів. Приклад: штучна нейронна мережа, метод опорних векторів, дерево рішень, експертна система.

на підставі інформації про пікселі:

− попіксельний класифікатор: цей класифікатор генерує підпис, використовуючи комбінацію спектрів всіх навчальних наборів пікселів заданої ознаки. Внесок усіх ознак, присутніх у пікселях навчального набору, присутній в отриманому підписі. Він може бути параметричним або непараметричним, точність може бути незадовільною через вплив проблеми змішаного пікселя. Приклади: максимальна ймовірність, ANN, метод опорних векторів та мінімальна відстань.

− субпіксельний класифікатор: спектральне значення кожного пікселя вважається лінійною чи нелінійною комбінацією певних чистих наборів, званих кінцевими елементами, забезпечуючи пропорційну відповідність кожного пікселя кожному кінцевому елементу. Субпіксельний класифікатор має можливість обробляти проблему змішаного пікселя, характерну для зображень

середнього та грубого просторового дозволу. Приклад: аналіз спектральної суміші, субпіксельний класифікатор, нечіткі класифікатори.

− класичний класифікатор: класифікатор по кожному полю призначений для вирішення проблеми гетерогенності навколишнього середовища, а також підвищення точності класифікації. Здебільшого використовуються підходи до класифікації на основі ГІС.

− об'єктно-орієнтований класифікатор: пікселі зображення об'єднані в об'єкти, а потім класифікація виконується на основі об'єктів. Вона включає в себе 2 етапи: сегментація зображення та класифікація зображень. Сегментація зображень поєднує пікселі в об'єкти, а потім класифікація реалізується на основі об'єктів. приклад: eCognition.

на основі кількості підсумкових категорій для кожного класифікованого елемента:

− тверда класифікація: також відома як чітка класифікація. У цьому випадку кожен об'єкт повинен мати членство в одному класі.

− м'яка класифікація: також відома як нечітка класифікація. У цьому випадку кожен об'єкт може виявляти множину та часткове членство у класі. Виходить точніший результат.

на підставі просторової інформації:

− спектральні класифікатори: ця класифікація зображень використовує чисту спектральну інформацію. Приклади: максимальна ймовірність, мінімальна відстань, штучна нейронна мережа.

− контекстуальні класифікатори: ця класифікація зображень використовує просторову інформацію про сусідні пікселі. Приклад: на основі частоти контекстуальний класифікатор.

− спектрально-контекстуальні класифікатори: ця класифікація використовує як спектральні, так і просторові дані, вихідні класифікаційні зображення генеруються з використанням параметричних чи непараметричних класифікаторів, а потім контекстуальні класифікатори використовуються у

класифікованих зображеннях. Приклад: комбінація параметричних або непараметричних та контекстуальних алгоритмів.

1.2 Методи класифікації об’єктів на зображеннях

У таблицях нижче наведено опис різних методів класифікації зображень, їх недоліки та переваги (табл. 1.1 та табл. 1.2).

Таблиця 1.1 – Методи класифікації зображень

Метод класифікації	**Опис**	**Характеристики**
1	2	3
Штучна нейронна мережа (ANN)	ANN – це тип штучного інтелекту, який імітує деякі функції людського розуму. ANN має нормальну тенденцію зберігання емпіричних знань. ANN складається з послідовності шарів, кожен шар складається з набору нейронів. Усі нейрони кожного шару пов'язані зваженими з'єднаннями з усіма нейронами на попередньому та наступному шарах.	Використовує непараметричний підхід. Продуктивність та точність залежать від структури мережі та кількості вхідних параметрів.
Дерево ухвалення рішень	Обчислює членство класу, повторно розбиваючи набір даних на рівномірні підмножини. Ієрархічний класифікатор дозволяє приймати та відхиляти мітки класів на кожному проміжний етап. Цей метод складається із трьох частин: Поділ вузлів, пошук термінальних вузлів та розподіл мітки класу на термінальні вузли	Засновано на ієрархічному правилі, використовують непараметричний підхід.

Продовження табл. 1.1

1	2	3
Метод опорних векторів	Будує гіперплощину або безліч гіперплощин у багатовимірному або нескінченномірному просторі, що використовується для класифікації. Гарний поділ досягається гіперплощиною, яка має найбільшу відстань до найближчої точки з тренувального набору будь-якого класу (функціональний запас), як правило, більше, ніж граничне значення, але нижче помилки узагальнення класифікатора.	Використовує непараметричний підхід з використанням бінарного класифікатора і може обробляти більше вхідних даних дуже ефективно. Продуктивність та точність залежать від вибору гіперплощини та параметрів ядра.
Нечітка міра	При нечіткій класифікації різні стохастичні асоціації визначаються для опису характеристик зображення. Різні типи стохастик поєднуються в набори властивостей, у яких члени цього набору властивостей є нечіткими по своїй природі. Це дає можливість описати різні категорії стохастичних характеристик у схожій формі.	Використовує стохастичний підхід. Продуктивність та точність залежать від вибору порога та нечіткого інтеграла.

Таблиця 1.2. – Переваги та недоліки методів класифікації зображень

Метод класифікації	**Переваги**	**Недоліки**
1	2	3
Штучна нейронна мережа (ANN)	− непараметричний класифікатор; − універсальний функціональний апроксиматор з довільною точністю; − датний, надає такі функції як OR, AND, NOT; − технологія, що самоналаштовується, заснована на даних; − ефективно обробляє помилкові вхідні дані; − висока швидкість обчислень.	− семантично слабка; − навчання вимагає суттєвих тимчасових витрат; − схильна до перенавчання; − важко вибрати тип архітектури мережі.

Продовження табл. 1.2

1	2	3
Дерево ухвалення рішень	− може обробляти непараметричні дані; − не вимагає докладного проектування та навчання; − забезпечує ієрархічні асоціації між вхідними змінними для прогнозування членства в класі та надає набір правил, які легко інтерпретувати; − просте і має досить хорошу обчислювальну продуктивність.	− використання меж гіперплощинних рішень, паралельних осям ознак, може обмежити їх використання у випадках, коли класи чітко помітні; − обчислення ускладнюються, коли різні значення не визначені та/або коли корелюють різні результати.
Метод опорних векторів	− надає можливість вибору порогового значення; − містить нелінійне перетворення; − забезпечує гарний ступінь узагальнення; − вирішує проблему перенавчання; − низька обчислювальна складність; − простота управління правилами прийняття рішень та частотою помилок.	− інтерпретованість результатів низька; − навчання займає багато часу; − структуру алгоритму важко зрозуміти; − визначення оптимальних параметрів складне, коли є нелінійний поділ даних навчання.
Нечітка міра	− ефективно справляється з невизначеністю; − властивості описуються шляхом визначення різних стохастичних відносин.	− без апріорних знань результат не точний; − точні рішення залежать від напряму рішення.

Отже, існує багато методів та запропоновано багато алгоритмів вирішення задачі класифікації зображень, проте усі ці ідеї поступаються у точності результату, простоті та швидкодії штучним нейронним мережам.

1.3 Застосування нейронних мереж у розпізнаванні зображень

Ідея моделювання роботи живих систем у технічних програмах з'явилася досить давно. Вже в середині XX ст. був відомий так званий перцептрон Розенблатта (рис. 1.1), який моделював роботу сітківки людського ока. Принцип дії елементів цієї моделі нагадує роботу сенсора-нейрона – нервової клітини мозку. На вхід такого сенсора подаються сигнали і залежно від значень, що надійшли, формується вихідний сигнал, який передається іншим сенсорам. Якщо сигнал посилюється – сенсор "збуджується", або, навпаки, зменшується – сенсор "пригнічується". Приблизно за таким же принципом передається сигнал за синоптичними волокнам наших нейронів.

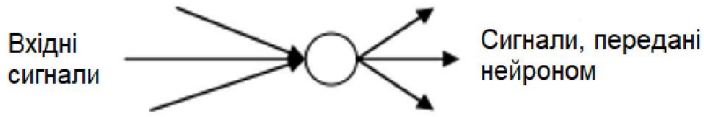

Рисунок 1.1 – Перцептрон Розенблатта

Можливості обчислювальних засобів середини XX ст. не дозволяли будувати достатньо ефективні схеми вирішення завдань на основі такої моделі, і про неї забули. Але з початку 90-х років, коли отримали широке поширення системи з паралельними обчисленнями, інтерес до таких моделей знову зріс, вони почали застосовуватися в технічних та інтелектуальних системах, зокрема, у системах розпізнавання.

Системою розпізнавання загалом називають інтерактивний програмно-технічний комплекс, здійснюючий процес розпізнавання об'єктів.

Системи розпізнавання можуть бути простими і складними, залежно від особливостей виділених класів об'єктів і використовуваних при цьому ознак. Прості системи виконують розпізнавання одним алгоритмом з урахуванням набору ознак одного типу. Найчастіше такі системи зустрічаються у технічних

додатках. Проте інколи доводиться мати справу зі складними системами розпізнавання, з використанням ознак різних типів та фізичної природи. Системи автоматизованого дешифрування для складання, наприклад, тематичних карт – це завжди складні інтерактивні системи розпізнавання, часто багаторівневі, коли результати, отримані одним алгоритмом розпізнавання, є вхідними даними іншого алгоритму.

Побудова складної системи розпізнавання вимагає попереднього аналізу всієї доступної інформації про об'єкти, що вивчаються. На цьому етапі необхідно:

− зібрати всі характеристики об'єктів, що вивчаються, які можна вилучити з доступних даних, що стосуються вирішуваної задачі;

− проаналізувати можливості формалізованого опису семантичних та непрямих ознак об'єктів, для кількісних характеристик, оцінити точності їх виміру чи розрахунку;

− сформувати повний набір формалізованих ознак об'єктів дослідження;

− визначити, за якими характеристиками розрізняються об'єкти, що підлягають виділенню тематичних категорій, та за якими характеристиками подібні об'єкти кожної категорії;

− вивчити поведінку отриманих ознак кожної тематичної категорії (класу) об'єктів та визначити набір найбільш інформативних ознак, що дозволяють надійно розділяти необхідні класи;

− проаналізувати можливість застосування до вибраних ознак описаних вище принципів розпізнавання та розробити загальну схему розв'язання задачі.

Якщо досліджуваний набір ознак та перелік класів не дає можливість вирішити задачу одним алгоритмом, потрібно розділити завдання на окремі процедури, тобто виконати декомпозицію системи розпізнавання. Наприклад, коли шукані класи відрізняються між собою за якимось ознаками, але всередині класів є групи, які за ними також відрізняються, можна спочатку виділити дрібніші класи, а потім згрупувати їх із використанням додаткових ознак. Є ще один шлях вирішення даної проблеми, якщо не вистачає ознак для надійного

поділу заданих класів, можна виділити більші, а потім аналізувати кожен із них з використанням іншої групи ознак. Інколи, навіть якщо розпізнавання може бути виконано на основі одного і того ж принципу, декомпозиція рішення необхідна для того, щоб аналітик даних мав можливість оцінити якість розпізнавання на окремих його етапах.

Отже, система розпізнавання включає дві основні функції: процес синтезу образів та процес аналізу образів. До процесу синтезу образів належить:

- формування набору ознак;
- визначення переліку класів;
- опис класів у вибраній системі ознак;
- оптимізація набору ознак щодо обраного переліку класів.

Остання процедура потрібна при надмірності вихідних даних, зокрема, мультиспектральних та особливо гіперспектральних зображень. Ці дані призначені для вирішення широкого кола тематичних завдань, тому при вирішенні конкретної задачі зазвичай потрібна лише частина з тих тематичних класів, які можуть бути виділено на таких даних навіть одним алгоритмом розпізнавання. Оптимізація набору ознак, яку іноді називають виділенням ознак, дозволяє значно підвищити ефективність системи розпізнавання. При цьому до поняття ефективності системи включається:

- якість результату розпізнавання;
- часові витрати на його одержання;
- витрати на розробку системи;
- витрати на навчання операторів – аналітиків даних.

Якість результату розпізнавання має оцінюватись окремо для кожного алгоритму. Для завдань тематичного дешифрування зображень воно включає два взаємопов'язані поняття: точність і достовірність. Точність визначається як частка правильно класифікованих пікселів, достовірність – як відповідність виділених класів їхньому тематичному змісту. Якщо точність в основному залежить від можливості поділу класів у заданій системі ознак і правила прийняття рішення, то достовірність залежить більшою мірою від повноти

системи ознак виділення обраних класів і репрезентативності вибірки образів при описі класу.

Аналіз образів – це, власне, сама процедура розпізнавання, тобто сукупність правил ухвалення рішення про віднесення образу до класу, що виконується нейромережею.

Відразу слід сказати, що навіть у грубому наближенні нейромережі не є моделлю біологічного процесу. Отже, її можливості не можна порівняти з можливостями мозку живих істот. Тим не менше, для багатьох конкретних практичних завдань така схема прийняття рішень виявляється ефективною.

Що ж є сучасна нейромережа? У найбільш загальному вигляді це система, що включає такі компоненти [5]:

- безліч простих процесорів (сенсорів);
- структуру зв'язків;
- правило розповсюдження сигналів;
- правило поєднання вхідних сигналів;
- правило обчислення сигналу активності;
- правило навчання, що коригує зв'язки.

Структура зв'язків мережі може бути представлена орієнтованим графом, який є багатополюсною мережею. Вузлами в ній є елементарні рецептори-процесори. Вхідні елементи – це безліч джерел, які отримують сигнали із зовнішнього середовища, а стоки – це елементи виходу. Структура зв'язків може бути різною: як довільною, так і впорядкованою за шарами. Можуть допускатися зв'язок між елементами одного шару, зв'язок елемента із собою, зворотний зв'язок між шарами. Якщо нейромережа – це орієнтований граф, то структуру зв'язків можна задати матрицею ваг w. Якщо елементи мережі впорядковані за шарами (згадаємо алгоритм Демукрона [6]), то для кожної пари суміжних шарів можна задати окрему вагову матрицю коефіцієнтів.

Правило поширення сигналів у мережі визначає, як відбувається оновлення стану елементів мережі. В одних випадках вибір моментів оновлення

проводиться випадковим чином, в інших стан груп певних елементів оновлюються лише після оновлення груп інших.

Найбільш поширеним способом комбінування сигналів є обчислення їх виваженої суми(1.1):

$$\boldsymbol{y_j = \sum_{i=1}^{n} x_i w_{ij}} \tag{1.1}$$

де y_j–повна величина сигналу на вході j-го сенсора, x_i–сигнал, вихідний від i-го сенсора попереднього шару, w_{ij}– вага зв'язку між i-м таj-м сенсорами, n – кількість задіяних зв'язків.

Часто використовується також квадратичне значення (1.2).

$$\boldsymbol{y_j = \sum_{i=1}^{n} (w_{ij} - x_{ij})^2} \tag{1.2}$$

Тобто сигнал від кожного i-го зв'язку для j-го сенсора оцінюється щодо максимально можливого.

Правило обчислення сигналу активності інакше називають функцією активності, а відповідне вихідне значення – активністю елемент. Якщо вихідний сигнал точно дорівнює комбінованому значенню вхідного сигналу, це тотожна функція. Але здебільшого нейронні мережі використовують нелінійні порогові функції. Порогова функція обмежує активність значенням 0 або 1 залежно від значення комбінованого входу Найчастіше віднімають значення порога із вхідного сигналу і розглядають «зміщення» або, інакше, «зсув» w_0, яке інтерпретується як зв'язок, що виходить від елемента, активність якого завжди дорівнює 1. Вираз для комбінованого вхідного сигналу тоді подають у вигляді (1.3).

$$y_j = w_0 + \sum_{i=1}^{n} x_i w_{ij} \tag{1.3}$$

Часто використовуваною нелінійною функцією активності є так звана сигмоїдальна функція, що безперервно заповнює діапазон від 0 до 1. Прикладом такої функції може бути логістична функція(1.4).

$$f(y) = \frac{1}{1 + e^{-y}} \tag{1.4}$$

Правило навчання, що коректує зв'язки – одна з головних переваг нейронної мережі, оскільки воно забезпечує її автоматичне налаштування на ухвалення рішення. Метою навчання є зміна вагових характеристик таким чином, щоб досягти необхідної поведінки мережі. Якість роботи нейронної мережі залежить від пред'явленого їй у процесі навчання набору навчальних даних, тобто від того, наскільки вони типові для розв'язуваного завдання. Найчастіше навчання – це тривалий процес, що вимагає численних експериментів. Метою навчання є мінімізація помилок розпізнавання.

Найчастіше використовується традиційний метод найменших квадратів. Тобто помилка для p-го навчального образу визначається за формулою (1.5),

$$E_p = \frac{1}{2}\sum_{j=1}^{n}(t_j - \tau_j)^2 \tag{1.5}$$

де t_j–необхідний вихід j-го елемента; τ_j–вихід, що спостерігається. Повна помилка за всіма пред'явленими образами є ΣE_p.

Для лінійних систем, коли $f(y_j) = y_j = w_0 + \sum x_i w_{ij}$, залежність квадрата помилки від ваг є параболічною, отже, існує мінімум, який можна

знайти, наприклад, методом градієнтного спуску. Загальне правило корекції ваг, що випливає з цього підходу – приріст ваг має відбуватися у напрямі, протилежному напрямку вектору градієнта.

З погляду сучасної теорії нейромереж, мережі з лінійною функцією активності – це стандартний випадок. Доведено, що нейромережі з лінійною функцією активності завжди можна тим чи іншим чином звести до одношарових [6]. Найчастіше в результаті виходить завдання лінійного дискримінантного аналізу – побудови оптимальних лінійних функцій, що розділяють, на основі певного алгоритму навчання. Найпростіший з них – алгоритм коригувальних прирощень або інакше принцип «підкріплення – покарання», який було запропоновано Розенблатом для розробленого ним перцептрону. У літературі системи розпізнавання такого типу раніше називали «самонавчальними» [7].

По-справжньому багатошаровими є лише мережі з нелінійною функцією активності. І, власне, проблемою, через яку у свій час загальмувався процес розробки нейромереж, була проблема навчання багатошарових мереж. Тобто надійного правила коригування ваг для багатошарових мереж не існувало. Такий алгоритм був розроблено лише 1986 р. Він називається алгоритмом зворотного поширення помилок.

1.4 Алгоритм зворотного поширення помилок

Розглянемо мережу з прямим зв'язком, тобто топологічно упорядковану за шарами. В алгоритмі зворотного розповсюдження помилок розглядаються два потоки: прямий потік – від входу до виходу та зворотний потік – від виходу до входу. Прямий потік відповідає процесу обчислення рішення, зворотний – просуває по мережі назад значення помилок. Тобто кожному значенню мережі, сигналу, що передається по дузі, відповідає зворотний сигнал, що визначає помилку. Таким чином, для кожного прямого проходу від входу до виходу будується зворотний прохід і розраховується величина помилки.

Спочатку треба розглянути, що є помилкою як функція ваг. Розглянемо довільний j-й елемент шару p.

Будемо, як і раніше, використовувати такі позначення:

x_i – сигнал, що виходить від і-го елемента p-1-го шару (тобто його функція активності);

w_{ij} – вага зв'язку між i-м елементом p-1-го шару та j-м елементом p-го шару;

w_0 – «поріг збудження» або зміщення – відповідає одиничній функції активності (зазвичай негативна величина);

y_j – сигнал на вході j-го елемента p-го шару.

Передбачається також, що комбінування сигналів на вході j-го елемента визначається як $y_j = w_0 + \sum_i x_i w_{ij}$.

В алгоритмі зворотного розповсюдження помилок передбачається, що функція активності не лінійна (зазвичай використовується сигмоїдальна функція). Позначатимемо її σ: $\sigma_j = f(y_j)$. Щоб визначити, як змінюється помилка при зміні ваги, треба розглянути її часткову похідну за відповідною змінною. Залежність помилки від ваг при довільній функції активності має вигляд(1.6):

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \sigma_j}\frac{\partial \sigma_j}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} \tag{1.6}$$

Як було сказано вище, збільшення вектора ваг при корекції ваг має йти в напрямку, протилежному градієнту помилки. Тобто в кожному вузлі мережі коригуюче збільшення δ_j можливо визначити як(1.7).

$$\delta_j = -\frac{\partial E}{\partial y_j} = -\frac{\partial E}{\partial \sigma_j}\frac{\partial \sigma_j}{\partial y_j} \tag{1.7}$$

Якщо правило комбінування сигналів на вході визначено лінійною функцією, тобто$y_j = \sum_i x_i w_{ij}$, то $\frac{\partial y_i}{\partial w_{ij}} = x_i$.

Звідси $\frac{\partial \mathrm{E}}{\partial w_{ij}} = -\delta_j x_i$, що, власне, і показує, як поширюється помилка через мережу. Звідси можна визначити збільшення ваги w_{ij} наступним чином (1.8):

$$\boldsymbol{\Delta w_{ij} = \eta \delta_j x_i} \tag{1.8}$$

де η – якась мала величина. Її називають нормою навчання, а вираз (1.8) – правилом Відроу-Хофф.

Залишилося подивитися, що являє собою величина *j*.

Якщо середньоквадратична помилка в *p*-му шарі$E_p = \frac{1}{2}\sum_{i=1}^{n}(t_j - \sigma_j)^2$, де t_j–потрібне значення функції активності *j*-го елемента з шару *p*,то

$\frac{d\mathrm{E}}{dw_{ij}} = -(t_j - \sigma_j)$.

Оскільки $\sigma_j = f(y_j)$, то $\frac{\partial \sigma_j}{\partial y_j} = f'(y_j)$.

Звідси $\frac{\partial E}{\partial w_{ij}} = -(t_j - \sigma_j) f'(y_j) x_i$.

Якщо як функція активності використовується логістична функція, то$f'(y_j) = [1 - f(y_j)]$.

Отже, зміна помилки в залежності від ваг може бути описана як

$\frac{\partial E}{\partial w_{ij}} = -(t_j - \sigma_j) f(y_j)[1 - f(y_j)] x_i = (\sigma_j - y_j) f(y_j)[1 - f(y_j)] x_j$,

тобто$\delta_j = (\sigma_j - t_j) f(y_i)[1 - f(y_j)]$.

Звідси отримуємо остаточний вираз для збільшення ваг Δw_{ij} (1.9).

$$\boldsymbol{\Delta w_{ij} = \eta(\sigma_j - t_j) f(y_j)[1 - f(y_j)] x_i} \tag{1.9}$$

Тепер розглянемо сам алгоритм зворотного розповсюдження помилок.

Зазвичай на нульовому кроці задають малі початкові ваги в інтервалі (-0,3, +0,3). З кожним значенням на вході зв'язується певний образ на виході. Навчання триває до того часу, поки сумарна середньоквадратична помилка по всіх входах не потрапить у задані межі.

У процесі навчання, крім норми навчання η (зазвичай η<1), використовується ще один параметр: інерційний член α, який залежить від ваг, отриманих на попередній ітерації. Цей параметр необхідний для того, щоб уникнути осциляцій на околиці рішення.

Крок 1. Прямий прохід (від входу до виходу).

Для кожного j-го елемента шарів p=1, розраховуємо значення на вході $y_j = w_0 + \sum_i x_i w_{ij}$ та функцію активації $\sigma_o = \frac{1}{1+\exp(-y_i)}$.

Крок 2. Значення $\tau_j = y_j$ на виході порівнюємо з очікуваними значеннями. Якщо значення збігаються, пред'являємо наступний образ. Якщо ні, то переходимо до кроку 3.

Крок 3. Обчислюємо помилку для елементів останнього прихованого шару p-1: $\delta_j = \sigma_j(1 - \sigma_j)\sum_k \delta_k w_{kj}$, k – індекси p-1 шару.

Повторюємо цю операцію для всіх шарів від p-2 до 1 (тобто проходимо по мережі в зворотному порядку і на всі вузли навішуємо обчислені для них значення δ).

Крок 4. Тепер робимо прямий прохід через мережу. Для всіх шарів оновлюємо ваги за правилом $\Delta w_{ij}(n+1) = \eta\delta_j\sigma_j + \alpha\Delta w_{ij}(n)$ де n означає номер ітерації навчання.

Висновки до першого розділу

У цьому розділі проведено теоретичний огляд та представлені відомості про різні підходи та методи до класифікації зображень. До найбільш поширених підходів класифікації зображень можуть бути віднесені контрольовані та неконтрольовані, параметричні та непараметричні або

об'єктно-орієнтовані, субпіксельні, попіксельні та рядкові або спектральні класифікатори, контекстуальні класифікатори та спектрально-контекстуальні класифікатори, а також жорсткі та м'які класифікатори. Розглянуті методи класифікації, а також їх переваги та недоліки, дозволяють обґрунтувати вибір на користь того чи іншого методу класифікації в залежності від типу задачі, що вирішується.

РОЗДІЛ 2
ГЛИБОКІ НЕЙРОННІ МЕРЕЖІ. ГЛИБОКЕ НАВЧАННЯ. ІНСТРУМЕНТИ ДЛЯ ГЛИБОКОГО НАВЧАННЯ: TENSORFLOW, THEANO, KERAS

2.1 Історія розвитку нейронних мереж

Штучні нейронні мережі були спроектовані так, щоб імітувати структуру мозку. Вони були вперше описані у 1943 році дослідниками Уорреном Маккаллохом та Уолтером Пітсом [3]. Тоді модель спочатку називалася пороговою логікою, яка розгалужувалася на два різні підходи: один більше надихався біологічними процесами людського мозку, а інший фокусувався на застосуваннях штучного інтелекту.

Хоча штучні нейронні мережі спочатку викликали великий інтерес і спонукали багато досліджень та розробок, їх популярність невдовзі знизилася, а дослідження сповільнилися через технічні обмеження. Вимоги до обчислювальних ресурсів у нейронних мереж були надто складними для комп'ютерів того часу. Комп'ютери не мали достатньої обчислювальної потужності і витрачали багато часу на навчання нейронних мереж. В результаті інші методи машинного навчання стали більш популярними, а штучні нейронні мережі переважно ігнорувалися.

Тим не менш, один важливий алгоритм, пов'язаний зі штучними нейронними мережами, був розроблений за цей час – «Метод зворотного поширення помилки», який вперше було описано 1974 р. А.І. Галушкиним, і навіть незалежно і водночас Полом Дж. Вербосом [4]. Даний алгоритм – це спосіб навчання штучних нейронних мереж, заснований на мінімізації помилок. Цей алгоритм дозволив вченим більше ефективно навчати штучні мережі.

Штучні нейронні мережі знову стали популярними наприкінці 2000-х років, коли такі компанії, як Google та Facebook, продемонстрували переваги використання цих методів машинного навчання у великих наборах даних,

зібраних у звичайних користувачів. Ці алгоритми в даний час в основному використовуються для глибокого навчання, що є областю машинного навчання, яка намагається моделювати складніші стосунки, наприклад, нелінійні стосунки.

Вони застосовуються для розпізнавання осіб, пішоходів, об'єктів, медичного аналізу, навігації автономних автомобілів та у багатьох інших сферах. У зв'язку зі зростанням обчислювальних потужностей та появою великих баз зображень стало можливим навчати глибокі нейронні мережі – нейронні мережі з великою кількістю прихованих шарів. У завданні розпізнавання образів особливого успіху досягли згорткові нейронні мережі (Convolutional Neural Networks), які щороку з 2012 року вигравали змагання ImageNet Large Scale Visual Classification Challenge (ILSVRC).

2.2 Принцип роботи штучних нейронних мереж

Кожна штучна нейронна мережа складається з безлічі прихованих шарів. Кожен прихований шар у штучній нейронній мережі складається з кількох вузлів. Кожен вузол пов'язаний з іншими вузлами, використовуючи вхідні та вихідні з'єднання. Кожне з'єднання може мати різну регульовану вагу. Дані передаються через ці приховані шари, і значення, отримані на виходах (вузлах кінцевого шару) в кінцевому результаті інтерпретується як різні результати. На малюнку (рис.2.1) представлена схема штучної нейронної мережі.

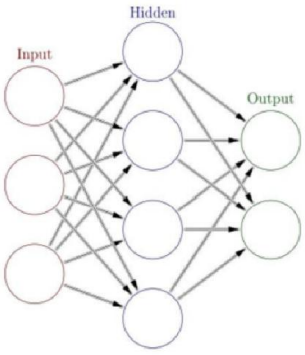

Рисунок 2.1 – Схема штучної нейронної мережі

На малюнку (рис.2.1) є три вхідні вузли, показані червоним. Кожен вхідний вузол являє собою параметр використовуваного набору даних. В ідеалі дані мають бути попередньо оброблені та нормалізовані до того, як буде подано на вхідні вузли.

У цьому прикладі для простоти наведено лише один прихований шар, який представлений синіми вузлами. Цей прихований шар має чотири вузли. Більшість штучних нейронних мереж мають більше одного прихованого шару.

Вихідний шар у прикладі показаний зеленим і має два вузли. З'єднання між усіма вузлами (представлені чорними стрілками на малюнку (рис.2.1) мають різні ваги під час процесу навчання.

2.3 Багатошарова нейронна мережа

У повністю підключеній нейронній мережі (рис. 2.2) всі нейрони передають свої вихідні сигнали іншим нейронам, і собі також. Всі вхідні сигнали надходять до всіх нейронів. Вихідний сигнал мережі може бути повністю рівний або частиною вихідного сигналу нейрона після кількох циклів роботи.

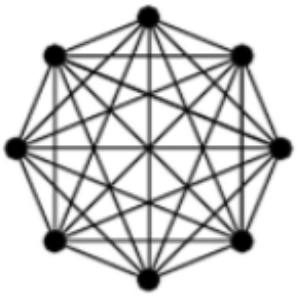

Рисунок 2.2 – Повнозв'язна нейронна мережа

У багатошаровій (ієрархічній) нейронній мережі (рис.2.3) нейрони формуються в шари. Цей шар містить набір нейронів з одним вхідним

сигналом. Кількість нейронів в шарі може бути довільною і не залежить від кількості в інших шарах. В такому випадку нейромережа складається з шарів, пронумерованих зліва направо. Зовнішній вхідний сигнал приходе на вхід цього шару нейронів (зазвичай нумерується), а вихід(мережі) є вихідним сигналом останнього шару. На додаток до вхідних і вихідних шарів у багатошаровій нейронній мережі (ієрархічній) є один або кілька прихованих шарів. З'єднання від виходу нейронів одного шару q до входу нейронів наступного шару ($q + 1$) називається послідовним з'єднанням.

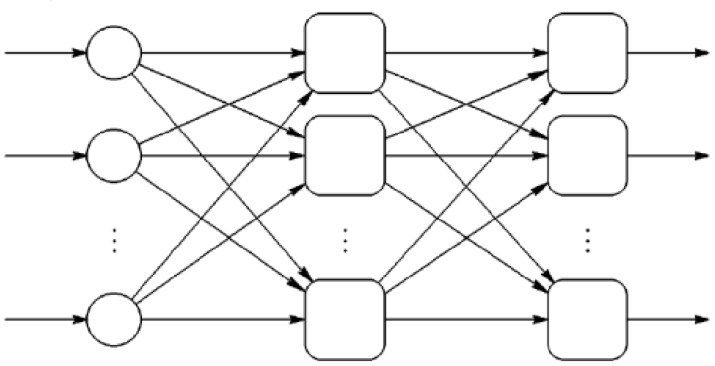

Рисунок 2.3 – Багатошарова нейронна мережа

2.4 Від нейронних мереж до глибокого навчання

Для того щоб навчати нейронну мережу використовуються різні варіанти градієнтного спуску, а щоб підрахувати градієнт, можна скористатися алгоритмами автоматичного диференціювання на графі обчислень. Всі ці методи, взагалі кажучи, не залежать від архітектури нейронної мережі і теоретично повинні працювати для будь-якого графа обчислень, аби ми вміли прокидати градієнти через вузли, тобто реалізовувати процедуру зворотного розповсюдження.

Ідея графа обчислень не здається такою вже складною, про те, як диференціювати композицію функцій, люди знають уже протягом кількох віків, та й безпосередньо алгоритм градієнтного спуску був відомий навіть раніше

XX століття. Сама ідея побудувати глибоку мережу, тобто нанизати один на одного кілька рівнів нейронів, теж не здається верхом винахідливості. Чи було так складно скомбінувати ці ідеї разом, що революція глибокого навчання почалося лише у XXI столітті, у 2005–2006 роках, а до цього глибокі архітектури та алгоритми були невідомі?

Звичайно, ні. Ідеї глибоких нейронних мереж мають майже таку ж довгу історію, як і самі штучні нейронні мережі. Перші глибокі мережі з'явилися ще в середині 1960-х років: перші справжні глибокі мережі у вигляді глибоких перцептронів були описані в роботах радянського вченого А. Г. Івахненка [8, 9]. Івахненко розробив так званий метод групового обліку аргументів [10]:

− спочатку вибираємо загальний вигляд, параметричне сімейство моделей, які навчатимемо (так звані поліноми Колмогорова-Габора, тобто, багаточлени з невідомими коефіцієнтами, але можуть бути й будь-які інші);

− будуємо та навчаємо різні варіанти обраних моделей;

− обираємо за допомогою метрики якості кілька найкращих моделей; якщо потрібну якість вже досягнуто, можна нічого далі не робити;

− якщо потрібну якість ще не досягнуто то починаємо будувати моделі наступного рівня, використовуючи виходи підібраних на попередньому кроці моделей як входи для наступних;

− цей процес можна рекурсивно повторювати доти, доки якість моделі або не досягне потрібного рівня, або не перестане покращуватись.

Метод групового обліку аргументів виглядає на подив сучасно. Якщо в ньому як базову модель вибрати перцептрон, отримаємо типову нейронну мережу з декількома шарами, яка навчається шар за шаром: спочатку перший, потім він фіксується і починається навчання другого, і т. д. Вже на початку 1970-х років цим методом цілком успішно навчалися моделі аж до семи рівнів у глибину [11], і це дуже схоже на процедуру навчання без вчителя.

Але все-таки нейронні мережі в результаті пішли трохи іншим шляхом. Першою глибокою нейронною мережею можна вважати вже згадуваний Neocognitron Куніхіро Фукусіми [12, 13], в якому з'явилися і згорткові мережі, і

активації, дуже схожі на ReLU. Однак ця модель не навчалася у сучасному сенсі цього слова: ваги мережі встановлювалися з локальних правил навчання без вчителя. Приблизно водночас з'явилися й глибокі моделі з урахуванням зворотного поширення. Першим застосуванням зворотного поширення помилки до довільних архітектур можна вважати роботи фінського тоді ще студента Сеппо Лінненмаа [14]: у 1970 році він побудував правила автоматичного диференціювання за графом обчислень, зокрема і зворотне поширення. Проте нейронними мережами та взагалі машинним навчанням Лінненмаа тоді не цікавився. Ці ідеї були застосовані в роботах Дрейфуса [15] та Вербоса [16], а починаючи з класичних робіт Румельхарта, Хінтона та Вільямса, що побачили світ у 1986 році, метод зворотного поширення став загальноприйнятим для навчання будь-яких нейронних архітектур.

А зараз перейдемо до питання, так би мовити, телеологічного характеру: навіщо взагалі потрібні глибокі мережі? Класична теорема Хорніка [17], заснована на ранніх роботах Колмогорова [18], стверджує, що будь-яку безперервну функцію можна як завгодно точно наблизити нейронною мережею з одним прихованим рівнем. Здавалося б, цього має бути достатньо. Навіщо плодити зайву складність на рівному місці?

Справа в тому, що глибокі архітектури часто дозволяють висловити те ж саме, наблизити ті ж функції набагато ефективніше, ніж неглибокі. Відомо, що схеми глибиною k+1 можуть ефективно висловити більше булевських функцій ніж схеми глибиною k. Більше того, можна навіть побудувати експоненційні поділи між різним числом рівнів, тобто для кожного k можна придумати функцію, яку можна виразити поліноміальною схемою глибини k+1, але яка потребує експоненційного розміру схеми глибини k [19]. З рівнями нейронної мережі виникає той самий ефект: ту саму функцію часто можна набагато краще наблизити глибшою мережею, ніж дрібною, навіть якщо загальне число нейронів у мережі залишити незмінним.

Відомо, що шар нейронів з ReLU-активацією фактично "згортає" простір, ототожнюючи деякі його частини між собою; і тому окремі поверхні,

побудовані у цьому «згорнутому» просторі, потім «розгортаються» у набагато складніші конструкції у просторі власне вхідних векторів.

Також глибока нейронна мережа створює непросто глибоке, а ще й розподілене уявлення. Тут йдеться про те, що кожен рівень глибокої мережі складається не з одного нейрона, а відразу з багатьох, і комбінації значень цих нейронів виробляють справжнісінький експонентний вибух у просторі входів, що зображено на малюнку(рис. 2.4). Уявімо, що можна розділяти точки на площині за допомогою трьох можливих ознак, кожен із яких – це лінійна функція: *f1*, *f2*, *f3*. На малюнку (рис. 2.4, а), зображена роздільна поверхня по одному з них, тобто пряма на площині, та розмічені частини, на які пряма ділить площину. Частин цих, зрозуміло, дві.

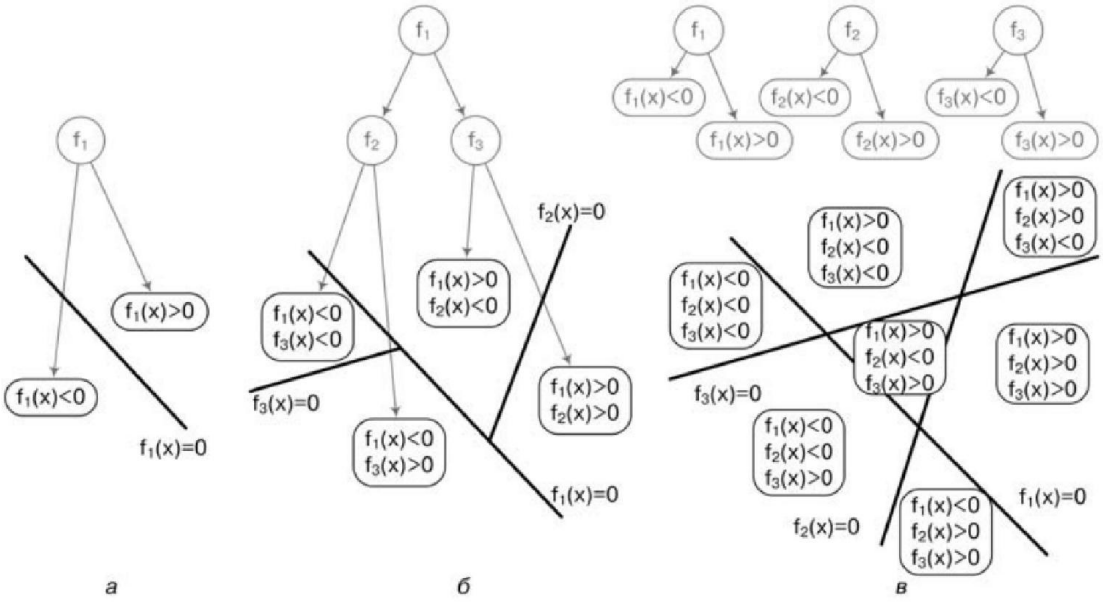

Рисунок 2.4 – Сила розподілених уявлень:

а)– розділяюча поверхня однієї ознаки;

б)– розділяюча поверхня дерева глибини 2 на 3 ознаках;

в)– розділяюча поверхня трьох ознак відразу

Як слід поєднувати ці ознаки у нашому класифікаторі? По перше, можна спробувати додати глибини та зробити дерево прийняття рішень; приклад

такого дерева і відповідна розділяюча поверхня показані на малюнку (рис. 2.4, б). Зверніть увагу, що дерево ділить площину на стільки ж елементів, скільки в нього листя: щоб розібрати чотири різні можливі випадки, нам потрібно було побудувати дерево з чотирма листками. А на малюнку (рис. 2.4, в) зображено розподілене уявлення: давайте уявімо, що наш «другий шар» може складатися із трьох нейронів відразу. Тоді всілякі комбінації цих трьох нейронів можуть розпізнати відразу. $2^3 = 8$ різних випадків (на малюнку їх реалізується 7), і це число різних варіантів зростає експоненційно при зростанні числа нейронів.

З наведеного вище, можна зробити висновок, що глибокі нейронні мережі дійсно потрібні. Але тоді виникає друге питання: Чому революція глибокого навчання відбулася в середині 2000-х, а чи не 1980-х років?

На це питання є дві відповіді. Перша – математична. Справа в тому, що глибокі нейронні мережі навчати, звичайно, можна тим самим алгоритмом градієнтного спуску, але в базовому варіанті, без додаткових хитрощів працювати це не буде. Уявіть, що ви почали навчати глибоку мережу алгоритмом зворотного поширення помилки. Останній, найближчий до виходів, рівень навчиться досить швидко і дуже добре. Але що буде далі? Далі виявиться, що більшість нейронів останнього рівня на всіх тестових прикладах вже «визначилися» зі своїм значенням, тобто їх вихід близький чи до нуля, чи до одиниці.

Таким чином, виходить, що останній шар нейронів, що навчився, «блокує» поширення градієнтів далі, тому ранні рівні глибокої мережі в результаті навчаються дуже повільно. Цей ефект називається проблемою загасаючих градієнтів (vanishing gradients).

З рекурентними мережами, які фактично по визначенню є дуже глибокими, виникає ще й зворотна проблема: іноді градієнти можуть почати «вибухати», експонентно, збільшуватися в міру «розгортання» нейронної мережі (exploding gradients). Обидві проблеми виникають часто, і досить довго дослідники не могли їх вирішити.

У середині 2000-х років з'явилися перші дійсно добре працюючі конструкції, що добре масштабуються... і вони дуже сильно нагадували вихідний механізм «глибоких моделей» Івахненка. Рішення, яке запропонували у групі Хінтона [20, 21, 22], полягало в тому, щоб передбачати нейронні мережі рівень за рівнем за допомогою спеціального випадку ненаправленої графічної моделі, так званої обмеженої машини Больцмана (restricted Boltzmann machine, RBM). Справа в тому, що градієнтний спуск, звичайно, добре знаходить локальний мінімум функції, а різні розумні варіанти градієнтного спуску здатні навіть вибиратися з невеликих локальних мінімумів і приходити до більш значного мінімуму.

Але все одно градієнтний спуск за своєю суттю локальний, він робить лише невеликі кроки в ту чи іншу бік від поточної точки, і результат сильно залежить від ініціалізації, від того, з якої точки ми починатимемо. Тому навчання без вчителя може призвести до дуже гарного ефекту: за рахунок того, що модель вже починає потихеньку «розбиратися» в структурі пропонованих даних, початкові значення ваг не випадкові, а досить розумні. Такі конструкції призвели до прориву спочатку в розпізнаванні мови, а потім і в обробці зображень.

Отже, прийшли до того, що просто так глибокі мережі навчати не вдається, і потрібно застосовувати різноманітні трюки, спочатку послідовно навчати окремі шари мережі зовсім іншими алгоритмами, а потім лише локально «докручувати» їх градієнтним спуском. Для виконання цієї задачі допоможе машина Больцмана.

Порівняно із серединою 2000-х років, і тим більше порівняно з початком 1990-х, навчати нейронні мережі можна більш ефективно. З'явився ряд важливих інструментів, які можна так чи інакше віднести або до регуляризації, або до різних модифікацій оптимізації в нейронних мережах. Цими інструментам будедропаут, міні-батчі, правильна ініціалізація ваг, нові модифікації градієнтного спуску… В цілому, хоча основні конструкції

нейронних мереж багато в чому прийшли до нас ще з 90-х, а то й 80-х років минулого століття, навчають їх зараз все-таки не випадковим способом.

Інша відповідь на питання про те, чому глибокі нейронні мережі було складно навчати – це те, що раніше комп'ютери були повільнішими, а доступних для навчання даних було набагато менше, ніж зараз.

Наприклад, навчання системи розпізнавання мови виглядало приблизно так: дослідники писали код і запускали навчання, яке тривало на тодішніх комп'ютерах тижнів зо два. Причому відбувалося це двотижневе навчання на класичному датасеті TIMIT, який зараз вважається досить маленьким та використовується для швидких експериментів. Через два тижні дивилися на результат, розуміли, що, мабуть, потрібно було підкрутити той чи інший параметр (а їх у нейронних мережах дуже багато, підкручували... і знову запускали навчання на два тижня. Звичайно, така обстановка не сприяла ні тонкому настроюванню мереж, ні своєчасним публікаціям до відповідних дедлайнів, ні просто технічній можливості успішно навчити нейронну мережу).

У середині 2000-х років все зійшлося докупи: комп'ютери стали досить потужними, щоб навчати великі нейронні мережі (більше того, обчислення в нейронних мережах незабаром навчилися делегувати відеокартам, що прискорило процес навчання ще на цілий порядок), набори даних стали достатньо великими, щоб навчання великих мереж мало сенс, а в математиці нейронних мереж відбулося чергове просування. І почалася та сама революція глибокого навчання.

2.5 Переваги глибокого навчання

Людський мозок не ідентифікує зображення попіксельно, а розкладає на підзадачі через кілька рівнів інтерпретацій. Як показано в [23], людський мозок обробляє візуальні сигнали через структуру декількох шарів, що добре моделюється нейронними мережами. Однією з перспектив розвитку глибокого навчання є заміна ознак, отриманих в результаті ручної роботи людини на

неконтрольоване або напівконтрольоване навчання ознак з подальшим ієрархічним вилученням цих ознак. Дослідження в цій галузі спрямовані на поліпшення структури нейронних мереж та створення моделей на навчання цих мереж великим обсягам даних. Одним з найбільш вражаючих фактів про нейронні мережі є те, що вони можуть обчислити взагалі будь-яку функцію. Тобто, незалежно від функції гарантується наявність такої нейронної мережі, що для кожного вхідного параметра *x*, значення *f(x)* або деяке наближення до нього є вихідним значенням мережі.

Теоретичні результати [24] показують, що архітектура з недостатньою глибиною може вимагати набагато більше обчислювальних елементів, число яких зростає експоненційно у відповідності до розмірності вхідних даних. Це також призводить до більш повільного навчання. Багаторівневі архітектури сприяють обміну та повторному використанню компонентів. Сенс наявності фільтрів на перших рівнях нейронних мереж глибокого навчання полягає в принципі використання загальних вагових коефіцієнтів та зсувів [25]. Це означає, що всі нейрони в першому прихованому шарі виявляють різні особливості у різних частинах вхідного зображення. З цієї причини іноді відображення вхідного шару на прихований шар називається картою властивостей. Вагові коефіцієнти, що визначають карту властивостей, називаються загальними ваговими коефіцієнтами, а зміщення – загальними зміщенням. Часто кажуть, що загальні вагові коефіцієнти та зміщення визначають ядро або фільтр згорткового шару.

Велика перевага використання загальних вагових коефіцієнтів і зміщень в тому, що вони значно зменшують кількість параметрів мережі. У той час, як з одного боку ядра дозволяють використовувати операцію згортки, з іншого боку мережі експлуатують шари об'єднання. Шар об'єднання – це шар, який на вхід приймає кожну вихідну картку властивостей згорткового шару та готує ущільнену карту властивостей, що сприяє успішному ієрархічному уявленню CNN.

2.6 Інструменти та бібліотеки для глибокого навчання

За останні кілька років розвиток машинного навчання досяг стрімких темпів, за рахунок бібліотек машинного навчання, глибокого навчання, які абстрагуються від складності скаффолдинга чи реалізації моделі. Машинне навчання та глибоке включає безліч математичних обчислень і операцій, особливо Matrix. За їх допомогою навіть простий новачок може розпочати роботу як професіонал.

Машинне навчання використовує математичні моделі загального призначення – відповіді на конкретні питання з допомогою даних. Протягом багатьох років машинне навчання використовувалося для виявлення спам-листів, створення розумних ракет, інтелектуальних роботів і будинків, виявлення об'єктів з допомогою комп'ютерного зору, розпізнавання мови, і навіть створення системи, яка може писати (романи, вірші тощо), рекомендувати продукти клієнтам та прогнозувати вартість товарів.

В даному підрозділі розглянемо інструменти машинного навчання, найперспективніші з них з погляду надійності, продуктивності та зручності використання.

2.6.1 Бібліотека TensorFlow. Google відкрив основу своєї системи машинного навчання, бібліотеку TensorFlow, наприкінці 2015 року з дозволу Apache 2.0. До цього бібліотека використовувалася Google як патентований засіб розпізнавання мовлення, пошуку, обробки фотографій та для електронної пошти Gmail разом з іншими програмами.

Бібліотека реалізована на основі мови C++ та має зручний інтерфейс прикладного програмування для Python, а також інтерфейс для C++, що користується меншою популярністю. TensorFlow можна швидко розгорнути в різних архітектурах.

Аналогічно Theano (популярна обчислювальна бібліотека для мови Python,) за допомогою якої обчислення описуються як блок-схеми,

відокремлюючи проектування від реалізації. Практично без жодних труднощів цей процес розподілу на частини дозволяє один і той же проект використовувати не тільки в великомасштабних навчальних системах з тисячами процесорів, а й на мобільних пристроях. Одна така система охоплює широкий діапазон платформ.

Однією з чудових властивостей TensorFlow є її здатність автоматичного диференціювання. Можна експериментувати з новими мережами без необхідності перевизначення багатьох ключових обчислень.

Автоматичне диференціювання спрощує виконання зворотного поширення помилки навчання, що призводить до складних обчислень під час використання методу нейронних мереж. TensorFlow приховує дрібні деталі зворотного поширення, що дає можливість звернути увагу на найважливіші питання.

Можна абстрагуватися від усіх математичних аспектів, оскільки вони знаходяться усередині бібліотеки. Це нагадує використання бази знань WolframAlpha для поставленої задачі розрахунку.

Іншою особливістю цієї бібліотеки є її інтерактивне середовище візуалізації, назване TensorBoard. Цей засіб показує блок-схему перетворення даних, відображає підсумкові журнали та відстежує хід виконання програми.

Створення прототипу TensorFlow значно швидше, ніж у Theano (програма ініціюється за секунди, а не за хвилини), оскільки багато операцій надходять попередньо скомпільованими. Завдяки виконанню підграфів стає легше налагоджувати програму; весь сегмент програми можна використовувати знову без повторних обчислень.

Оскільки TensorFlow використовується не тільки в нейронних мережах, вона також містить готовий пакет матричних обчислень та інструменти маніпулювання даними. Більшість бібліотек, таких як Torch та Caffe, призначено виключно для глибоких нейронних мереж, але TensorFlow є більш універсальною, а також має можливість масштабування.

Ця бібліотека добре задокументована та офіційно підтримується Google. Машинне навчання є складним предметом, тому лише компанія з винятково високою репутацією може підтримувати TensorFlow.

2.6.2 Theano та надбудови над Theano. Theano [26] та його високорівневі надбудови утворюють свого роду сімейство бібліотек, що мають аналогічні характеристики. Оскільки Theano насправді є не зовсім бібліотекою машинного навчання, а скоріше бібліотекою оптимізації низького рівня для обчислювальних графів, існує безліч інших бібліотек, в основі яких лежить Theano.

Theano – це бібліотека Python, що розробляється в Університеті Монреалю з 2008 року. Це оптимізована бібліотека маніпулювання тензорами, яка призначається для використання як "backend движка" високорівневих обгорток Theano. Theano зосереджується на ідеї обчислювальних графів: вона знає, як перетворити обчислювальну структуру графів в ефективний код за допомогою бібліотеки SciPy-NumPy [11], BLAS та інших рідних бібліотек і чистого коду C++ для запуску CPU або GPU так швидко, як можливо. Theano забезпечує оптимізацію швидкості та стабільності, оскільки він реорганізує та оптимізує обчислення всередині. Однією з її «фішок» є автоматична диференціація: потрібно лише реалізувати пряму (передбачувану) частину моделі, і Theano автоматично визначить, як обчислити градієнти у різних точках, дозволяючи користувачам виконувати градієнт спуску для навчання моделі. Іншим ключовим аспектом Theano є використання графічного процесора для обчислень. Використання графічного процесора в Theano насправді прозоре, у тому сенсі, що користувачі можуть писати той самий код і запускати його або на CPU, або на GPU. А конкретніше, Theano визначає, які частини обчислень мають бути перенесені на GPU.

Theano насправді не є бібліотекою машинного навчання, оскільки вона не надає користувачеві попередньо побудовані моделі, які можуть навчатись на наборах даних. Вона є математичною бібліотекою, яка надає інструменти для

створення власних моделей машинного навчання. Використання Theano спрощує реалізацію методу зворотного поширення помилки для згорткових мереж та рекурентних мереж. З цієї причини вона надає функції "if else" або "switch", що дозволяють використовувати умовний потік керування на графі. Цикли спрощуються за допомогою функції «Сканування».

Lasagne [27] – бібліотека на Python, побудована на основі Theano, по суті – обгортка високого рівня над Theano. У той час як можна використовувати Keras [28] (іншу обгортку Theano), ніколи не знаючи про те, що Theano лежить в основі, це не справедливо для Lasagne, яка призначена для використання з Theano та для взаємодії з нею. Lasagne була реалізована з особливим акцентом на прозорість: вона не приховує Theano за абстракціями, оскільки вона безпосередньо обробляє та повертає вирази Theano або типи даних Python numpy.Lasagne дозволяє створювати архітектури з кількома входами та декількома виходами та забезпечує методи оптимізації, включаючи стохастичний градієнт спуску, метод Нестерова, Adagrad, Adadelta та Adam. Функція втрат легко визначається і немає необхідності витягувати градієнти через символічну диференціації Theano. Ще однією корисною характеристикою Lasagne є її модульність: вона дозволяє використовувати всі частини (шари, вагові коефіцієнти, фільтри...) незалежно від Lasagne, оскільки їх легко зберегти та експортувати для повторного використання.

Інші обгортки Theano включають Pylearn2, Bloaks та Keras. Алгоритми Pylearn2 можуть бути записані з використанням виразів Theano, а Theano оптимізує та стабілізує вирази. Вона включає все необхідне для мереж багатошарових перцептронів, обмеженої машини Больцмана та ССП. Блоки – це фреймворк, що вводить поняття «Цеглини» для створення моделей. Цеглина– це параметризовані операції Theano. Блоки включають цікаві функції, такі як моніторинг та аналіз значень у міру просування навчання, та автоматичне збереження та поновлення навчання. Keras – вкрай модульна бібліотека нейронних мереж, написана мовою програмування Python і здатна працювати поверх як TensorFlow [29], так і Theano (підтримує бекенди Theano

та TensorFlow). Вона була розроблена з акцентом на можливість швидкого експериментування, оскільки мати можливість легко перейти від ідеї до результату – це ключ до проведення ефективних досліджень.

2.6.3 Бібліотека Keras. Keras – це платформа підтримки глибокого навчання Python, яка забезпечує зручний спосіб створення й вивчення практично будь-якої моделі глибокого навчання. Keras спочатку був розроблений для дослідників, щоб проводити швидкі експерименти.

Keras має такі основні характеристики:

- дозволяє виконувати той самий код на CPU або GPU;
- має дружній API, який допомагає розробляти прототипи моделей глибокого навчання;
- включає вбудовану підтримку згорткових мереж (використовується для розпізнавання образів), рекурентних мереж (використовується для обробки послідовності) та їх різних комбінацій;
- включає підтримку будь-якої архітектури мережі: моделі з кількома входами або виходами, спільне використання шарів, спільне використання моделей тощо. Це означає, що Keras підходить практично для будь-якої моделі глибокого навчання, від генеративних змагальних мереж до нейронних машин Тьюринга.

Фреймворк Keras розповсюджується на умовах вільної ліцензії та може безкоштовно використовуватись у комерційних проектах. Він сумісний із будь-якими версіями Python, від 2.7 до 3.6.

Налічується більше 200 000 користувачів Keras, починаючи з академічних дослідників та інженерів у стартапах і великих компаніях і закінчуючи аспірантами та аматорами. Keras використовується в Google, Uber, Netflix,Yelp CERN, Square та сотні стартапів, що вирішують широке коло завдань. Keras також користується великою популярністю серед учасників змагань з машинного навчання, проведених сайтом Kaggle, де майже всі недавні конкурси з глибокого навчання виграли з використанням моделей Keras.

2.7 Моделі глибокого навчання

Після вибору програмного забезпечення необхідно з'ясувати, яка архітектура є найкращою для класифікації зображень. Розглянемо наступні варіанти.

2.7.1 Згорткова нейронна мережа (CNN). CNN – це одна з нейромережевих моделей глибокого навчання, яка може бути описана трьома специфічними характеристиками, а саме: локально пов’язаними нейронами, загальними вагами та просторовими чи часовими вибірками. Архітектура CNN зосереджена на використанні двовимірної структури вхідного зображення (або іншого двовимірного входу, наприклад мовного сигналу). Це досягається за допомогою локального з'єднання нейронів і пов'язаних ваг, за яким слідує певний тип асоціації, що усуває інваріантність. Перевага CNN полягає в тому, що її легше вивчити, якщо параметри набагато менші, ніж у повністю підключеної нейронної мережі з такою ж кількістю прихованих шарів. Коротко описуючи характеристики архітектури, можна сказати, що CNN складається із згорткового шару та шару підвибору, який включає наступні (необов’язкові) повністю пов’язані шари.

Вхідний прийом згорткового шару виражається як: зображення $m * m * r$, де m– висота і ширина зображення, а r– кількість каналів. Наприклад, $r = 3$ для зображень RGB. Згортковий шар матиме k фільтрів розміру $n*n*q$, де n менше розміру зображення, а q може дорівнювати числу каналів r або менше і може змінюватися для кожного ядра. Розмір фільтра створює локальну структуру посилань, в якій кожен елемент переплетений із зображенням для створення k карт атрибутів розміру $mn + 1$. У згортковому шарі кожен нейрон локально підключений до входу з попереднього шару, і його функція – це двовимірна згортка з фільтром, тому його вагу можна розрахувати як результат нелінійного перетворення (2.1).

$$\boldsymbol{a_{ij} = p(f * x) = p(\sum_{i'=0}^{n} \sum_{j'=1}^{n} f_{i'j'} x_{i+i^i,j+j'} + b)} \quad (2.1)$$

де *f* – матриця вагових коефіцієнтів фільтра згортки розміру *n*n*, x відноситься до ваг вхідних нейронів, пов'язаних з нейронами (*i, j*) у наступному згортковому шарі. *ρ()* – нелінійна вагова функція (зазвичай сигмовидна або гіперболічна дотична), *b*– зсув, * є оператором згортки. Після згортки кожна карта властивостей може бути субдискретизована із середнім або максимальним об'єднанням по *p*p* суміжним областям, де *p* варіюється між 2 для невеликих зображень (наприклад, MNIST) і зазвичай не перевищує значення 5 для великих вхідних зображень. CNN довели свою ефективність [30], у тому числі на міжнародних змаганнях:

- розпізнавання рукописного введення (MNIST, арабська HWX-ISDIA);
- оптичне розпізнавання символів у міських умовах (2011),розпізнавання номерів вулиць (Нью-Йоркський університет);
- розпізнавання дорожніх знаків (2011), конкурс ГЦРБ (IDSIA, Нью-Йоркський університет);
- виявлення пішоходів на INRIA (2013) і інше (Нью-Йоркський університет);
- розпізнавання об'єктів (2012), конкурс ImageNet;
- визначення поведінки людини в наборі даних (2011) Hollywood II (Стенфорд) [18];
- і багато інших.

2.7.2 Рекурентна нейронна мережа (RNN). Суть RNN полягає у використанні послідовної інформації [22]. У традиційній нейронній мережі ми припускаємо, що всі входи (і виходи) не залежать від один одного. Але для багатьох завдань це є неправильним припущенням. Для передбачення наступного слова в пропозиції краще знати, які слова були перед ним. RNN

називаються рекурентними, оскільки вони виконують одне й те саме завдання для кожного елемента послідовності, а вихід залежить від попередніх обчислень. Інший підхід до розуміння RNN полягає у уявленні, що у них є «пам'ять», яка фіксує інформацію про попередні обчислення. Теоретично RNN можуть використовувати інформацію про послідовності довільної довжини, але на практиці вони обмежуються лише декількома попередніми кроками. На малюнку (рис. 2.5) представлена базова структура RNN:

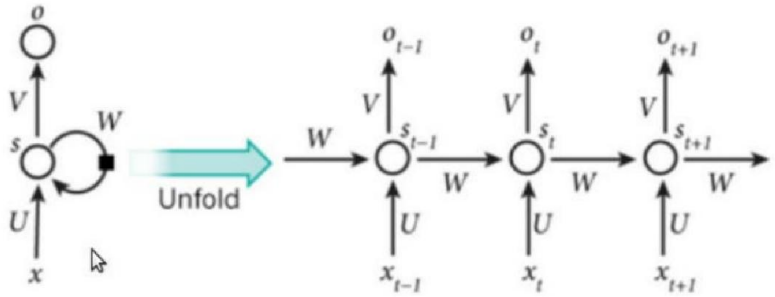

Рисунок 2.5 – Структура RNN

x_t – вхідне значення на кроці *t*. Наприклад, x_1 може бути вектором, відповідним другому кадру відео. s_t– прихований стан у момент часу *t*. Це пам'ять мережі. s_t розраховується на основі попереднього прихованого стану та входу на поточному кроці: $s_t=f(Ux_t+Ws_t\text{-}1)$. Функція f зазвичай нелінійна, така як тангенс чи ректифікатор. s_t-1, ..., s_t-*n*, які необхідні для обчислення першого прихованого стану, зазвичай ініціалізуються нулями. o_t – вихідне значення на кроці *t*. Наприклад, для прогнозування наступного кадру у відео, цей вихід буде вектором ймовірностей всіх можливих пікселів (дескриптори кадру), що використовуються у цій системі. Існує кілька різних видів RNN. В двонаправленій RNN [31] навчання здійснюється шляхом подачі даних одночасно у позитивному та негативному напрямку часу. Багатовимірні RNN розширюють одновимірні вхідні дані до багатовимірних даних, тим самим розширюючи потенційну застосовність RNN до застосування в завданнях комп'ютерного зору, обробці відео, розпізнавання медичних зображень та

багатьох інших областях. Повторні нейронні мережі із зворотним зв'язком (GFRNN) [32] розширюють існуючий підхід укладання кількох повторюваних шарів шляхом контролю сигналів, що йдуть від верхніх рекурентних шарів до нижніх шарів.

2.7.3 Залишкова нейронна мережа (ResNet). Варіацією стандартних CNN є глибокі залишкові нейронні мережі. ResNet беруть стандартну мережу прямого розповсюдження ConvNet і додають пропускні з'єднання, які пропускають (або обходять) кілька шарів згортки відразу. Кожен такий обхід створює залишковий блок, як показано на малюнку (рис.2.6), в якому шари згортки пророкують залишковий шар, який додається до входу наступного блоку. Такі мережі можуть підвищити точність за рахунок значного збільшення глибини [33]. Залишкова нейронна мережа у 8 разів глибша, ніж мережа VGG [34], але, як і раніше, має нижчу складність по порівняно з нею, посіла перше місце на змаганні ILSVRC 2015 року.

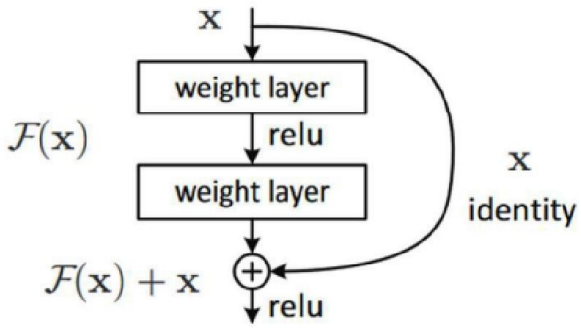

Рисунок 2.6 – Структурний елемент залишкової нейронної мережі

Залишкові шари призначені для вирішення проблеми деградації, яка виникає у міру збільшення глибини, але не пов'язана з перенавчанням [33]. Точність спочатку підвищується до своєї межі, а потім швидко погіршується. Коли додаткові шари виявляються непотрібними або навіть ведуть до деградації, тотожне відображення шарів є найефективнішим рішенням. Залишкове з'єднання дозволяє підштовхнути залишкове значення до нуля і

поширити по стеку тотожне відображення, таким чином «приховуючи» наявність зайвих (непотрібних) шарів. У крайньому випадку, якщо тотожне відображення було оптимальним для конкретного моменту навчання, буде легше підштовхнути залишкове значення до нуля, ніж пристосувати тотожне відображення до стеку нелінійних шарів.

Разом із деградацією у міру збільшення глибини, у ResNets була також розглянута проблема внутрішнього підступного зсуву. Внутрішнє зрушення коваріації викликане тим, що розподіл вхідних значень кожного шару змінюється під час навчання разом із зміною значень параметрів мережі. Внаслідок цього навчання сповільнюється, а також утруднюється навчання мереж із насиченням нелінійностей [35].

Нормалізація вибірки коригує розподіл вхідних значень шарів у міру навчання. Відбілювання вхідних значень (лінійне трансформування для досягнення нульового середнього значення та одиничною дисперсією) прискорює процес навчання мережі. Оскільки міні вибірки використовуються у стохастичному градієнтному навчанні, кожна міні-вибірка обчислює наближення середнього значення та дисперсії кожного вагового коефіцієнта. Потім статистика, що використовується при нормалізації, може бути використана для участі в процесі зворотного поширення градієнта (2.2):

$$\boldsymbol{y_{k+1} = y_k * norm(x) + \beta_k} \tag{2.2}$$

Крім того, розмір та зсув перетворення типу додається до кожного залишкового блоку навчання. Масштабування та зсувні перетворення дозволяють мережі мати можливість представляти тотожне перетворення.

2.7.4 Довга короткострокова пам'ять (LSTM). Однією із сильних сторін RNN є те, що вони дозволяють використовувати інформацію, отриману на попередньому етапі, для виконання поточного завдання, наприклад, використання попередніх відеокадрів може покращити розуміння поточного

кадру. RNN загалом можуть навчитися використати минулу інформацію. Але є також випадки, коли потрібно більше контексту. Спробуйте передбачити останнє слово в текст «Я виріс у Швеції. Я вільно говорю по …». Останні дані підказують, що наступне слово, ймовірно, є назвою мови, але щоб звузити можливий перелік варіантів до конкретної мови, потрібен контекст Швеції із попередньої пропозиції. Це цілком можливо, оскільки відстань між необхідною інформацією та місцем, в якому вона потрібна для передбачення, не дуже велика.

LSTM розроблено, щоб уникнути проблеми довгострокової залежності. У традиційній рекурентній нейронній мережі, під час процесу зворотного поширення градієнта, значення градієнта можуть бути в кінцевому підсумку багаторазово помножені на вагову матрицю коефіцієнтів, що відображає зв'язки між нейронами рекурентного прихованого шару. Це означає, що величини ваг в матриці переходу можуть вплинути на процес навчання. Якщо ваги у цій матриці малі (або, більш формально, якщо провідне власне значення вагової матриці менше 1,0), це може призвести до ситуації, що називається «зниклі градієнти», коли градієнтний сигнал стає таким невеликим, що навчання або стає дуже повільним, або зовсім перестає працювати. Це також може ускладнити завдання вивчення довгострокових залежностей у даних. І навпаки, якщо ваги у цій матриці великі (або провідне власне значення вагової матриці більше 1,0), це може призвести до ситуації, коли градієнтний сигнал настільки великий, що він може викликати відхилення у процесі навчання. Така подія часто називають градієнтами вибуху. На малюнку (рис.2.7) представлена структура комірки пам'яті.

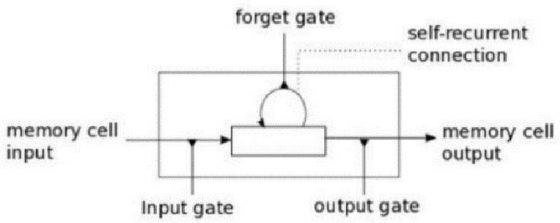

Рисунок 2.7 – Базова структура секції пам’яті

Вирішення цих проблем є основною метою моделі LSTM, вперше представленої у 1997 році [36]. Ця модель використовує нову структуру, звану осередком пам'яті. Осередок пам'яті складається з чотирьох основних елементів: «вхідний вентиль», «вихідний вентиль» та нейрон, з'єднаний із самим собою та «вентилем забування» (відповідно "input gate", "output gate", "self-recurrent connection" та "forget gate»).Зв'язок нейрона із самим собою має вагу 1.0 та гарантує, що, незважаючи на будь-які зовнішні впливи, стан осередку пам'яті може залишатися постійним від одного кроку до іншого. Вентилі служать для модулювання взаємодій між осередком пам'яті та середовищем. Вхідний вентиль може дозволити вхідним сигналам змінювати стан комірки або заборонити це. З іншого боку, вихідний вентиль може дозволити стану клітинки пам'яті впливати на інші нейрони або запобігти цьому. І, нарешті, вентиль забування може моделювати рекурентний зв'язок осередку з самим собою, дозволяючи запам'ятовувати або забувати свій попередній стан у міру потреби.

Висновки до другого розділу

У цьому розділі було розглянуто використання штучних нейронних мереж для класифікації зображень. Був зроблений наголос на вивченні методів та принципів глибокого навчання, а також інструментів та бібліотек, що існують у цій галузі. Як і звичайні нейронні мережі, глибокі нейронні мережі можуть моделювати складні нелінійні зв’язки між елементами. У процесі їх навчання глибока нейронна мережа намагається представити об’єкт у вигляді комбінації простих примітивів (наприклад, у задачі розпізнавання обличь такими примітивами можуть бути частини обличчя: ніс, очі, рот). Навчання глибинних нейронних мереж може бути здійснено за допомогою звичайного алгоритму зворотного поширення помилки і градієнтного спуску. Однак під час вивчення глибинних структур виникає кілька проблем, які необхідно

враховувати при оптимізації функцій у великому просторі: кількість обчислювальних елементів, початкові умови для ваг мережі, а також постійне коригування розміру кроку.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ НЕЙРОННОЇ МЕРЕЖІ ДЛЯ КЛАСИФІКАЦІЇ ЗОБРАЖЕНЬ

3.1 Вибір фреймворку для розробки мобільного додатку

Мобільна розробка – це вже не просто розробка iOS з використанням Swift або Android-розробка на Java. Ми знаходимося в поколінні гібридних, кросплатформових та прогресивних веб-додатків.

Коли справа доходить до розробки додатків, два популярні варіанти – це Ionic і React Native (рис. 3.1). Хоча обидві ці платформи відносяться до кросплатформної екосистеми, між ними є суттєві відмінності. Будь то React Native або Ionic; кожна має свій набір функцій, можливостей і цілей. Який краще? Який із них розглянути для нашого проекту? Ми намагатимемося визначити кожен з них індивідуально, а потім виділити плюси та мінуси, щоб отримати чітке уявлення.

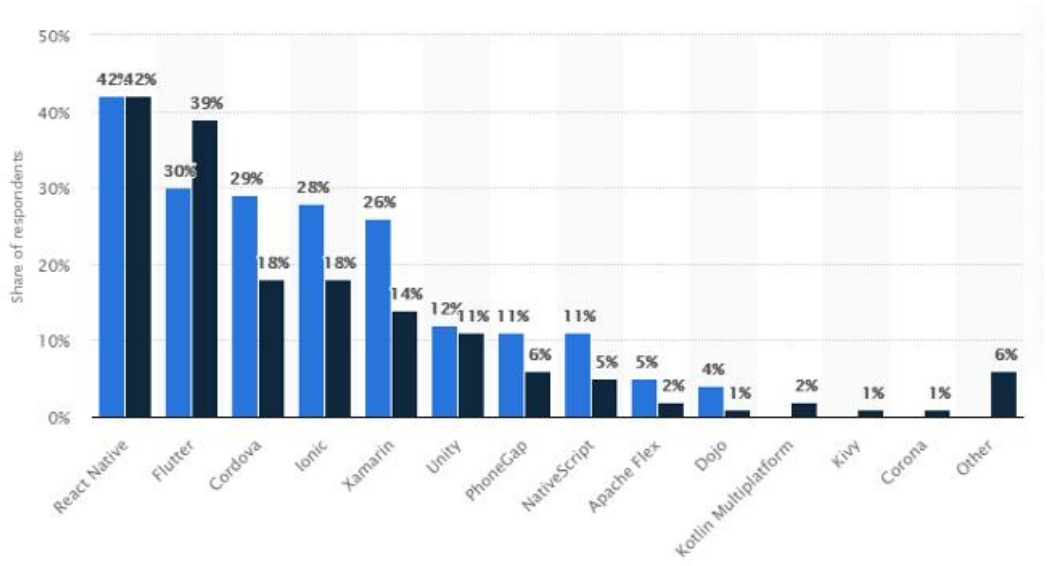

Рисунок 3.1 – Рейтинг використання серед розробників

React Native – це кросплатформне рішення, що дозволяє розробляти мобільні програми як на iOS, так і на Android. Він написаний на JavaScript з використанням React. React Native використовується для створення розробки мобільних програм з використанням Javascript і React як на iOS, так і на Android.

Ionic – це фреймворк для розробки гібридних мобільних додатків. Він дає можливість використовувати веб-технології, такі як HTML, Javascript і CSS для створення високоякісних кросплатформових додатків. Ionic Framework випустила безкоштовну електронну книгу, яка пояснює різницю між гібридною програмою та нативною програмою. Ідея полягає в тому, що ви пишете код один раз, і цей код можна розгорнути кроссплатформено на iOS, Android, в Інтернеті та на комп'ютері. Одна й та сама кодова база використовується на всіх платформах.

Чим вони схожі? Ionic дозволяє створювати мобільні, настільні та веб-додатки з використанням традиційних веб-технологій, таких як Javascript/CSS/HTML. Аналогічно, ReactNative дозволяє створювати нативні мобільні програми з використанням Javascript та React – популярного Javascript-фреймворку. Це означає, що вам не потрібно знати нативну розробку для iOS або Android, досить використовувати Ionic або ReactNative.

Чим вони відрізняються? React Native – це нативний фреймворк. React Native використовується для створення справді нативних кросплатформених додатків. Нативна програма побудована певною мовою програмування для платформи конкретного пристрою, iOS або Android.

Нативні програми для iOS написані на Swift або Objective-C, а нативні програми для Android написані на Java. З React Native всі базові віджети є нативними компонентами, що забезпечує зручність роботи з програмою. Він побудований з використанням Javascript та React, але всі компоненти є нативними компонентами iOS та Android. Перевага React Native в тому, що він створює нативні програми для iOS і Android з єдиною базою коду. Програми,

розроблені за допомогою React Native, мають чудовий інтерфейс користувача в порівнянні з фреймворками, що використовують Web View.

Ionic – це гібридна програма. Він використовує HTML, CSS та Javascript для створення програм, які можна використовувати в Інтернеті, на комп'ютері та на мобільному пристрої. Гібридні програми по суті використовують те, що називається Web View, для створення мобільних програм. Ionic-додатки створюються з використанням веб-технологій і відображаються з використанням Web View, які є повноекранним повнофункціональним веб-браузером.

Ідея тут полягає в тому, щоб повторно використати код на кількох платформах. Таким чином, гібридні програми не матимуть доступу до вбудованих функцій мобільного пристрою із коробки. Нативні функції, такі як камера, GPS, контакти тощо. Ionic використовує плагіни Cordova для інтеграції нативних функцій у вашу програму. Ionic відображає свої графічні елементи через браузер, який виконує кілька кроків, щоб розпочати показ компонента на екрані. Це може призвести до зниження продуктивності у великих програмах.
Але коли ви створюєте веб-програми з використанням гібридних технологій, таких як Ionic, їх легко перетворити на прогресивні веб-програми (PWA), які можна завантажити, як і будь-який інший мобільний додаток.

Якщо ваш продукт є виключно мобільним додатком, який має працювати на iOS та Android і має виглядати як нативний додаток, вам слід вибрати React Native. Якщо ви шукаєте елегантний інтерфейс користувача, який відповідає вашому веб-сайту у формі мобільного додатка, ви можете натомість розглянути Ionic.

Відрізняються розглянуті фреймворки й технологічним стеком.

React Native написано на Javascript з використанням популярного фреймворку React. Елементи інтерфейсу користувача написані на JSX, а не на HTML. JSX виглядає як будь-яка інша мова шаблонів, але постачається з усіма можливостями Javascript. Причина, через яку React Native використовує React,

полягає в тому, що обидві платформи були розроблені та створені з відкритим вихідним кодом Facebook.

Ви можете використовувати Angular, Vue або навіть React для створення програм Ionic. Технічний стек Ionic набагато більш гнучкий у порівнянні з React Native. З останньою версією Ionic 4 ви можете використовувати Ionic із будь-яким середовищем веб-розробки.

До минулого року React Native міг вважатися переможцем у цій категорії, оскільки програми Ionic створювалися з використанням лише Angular. Але з недавніми оновленнями для Ionic це відкриває величезну гнучкість у створенні програм Ionic. Будь-який веб-розробник на будь-якій сучасній веб-платформі може розробляти програми Ionic. Це величезний бонус для Ionic, що робить його переможцем у цій категорії [37].

3.1.1 Продуктивність фреймворків. Якщо ви дійсно хочете досягти максимальної продуктивності, найкраще писати нативні програми окремо для iOS та Android. Це пов'язано з тим, що нативні програми забезпечують доступ до нативних функцій платформи. Тут немає рівня абстракції, а пряма взаємодія із нативними модулями iOS та Android веде до підвищення продуктивності. І React Native, і Ionic не можуть відповідати нативному виконанню. Тим не менш, давайте подивимося, який з них є кращим.

React Native ближчий до нативної продуктивності, ніж Ionic. React Native, по суті, створює нативні програми з використанням Javascript. Він має зовнішній вигляд нативної програми та використовує ті ж елементи, що й нативні програми. Єдина відмінність полягає в тому, що він кроссплатформенний і використовує Javascript для обтікання цих власних будівельних блоків. React Native забезпечує відмінну продуктивність та швидкість реагування на ваші мобільні програми.

У Ionic з іншого боку, реалізується гібридний підхід. Він не створює нативні програми, і ви можете мати справу з проблемами продуктивності. Існує безліч зворотних викликів нативного коду, які можуть спричинити затримку.

Для Ionic також потрібен плагін Cordova, якщо вам потрібний доступ до нативних функцій. Хоча Ionic є відмінним рішенням для створення елегантного інтерфейсу користувача і більш швидкої розробки, він все ж таки має деякі проблеми з продуктивністю в порівнянні з такими середовищами, як React Native.

React Native забезпечує кращу продуктивність, ніж Ionic. Додатковий рівень в Ionic, який включає плагіни Cordova, додає повільності, оскільки він створює Web View, а не нативний додаток. React Native, з іншого боку, обертається навколо своїх компонентів, що забезпечує кращу продуктивність. На основі наведених фактів, логічніше буде обрати для розробки мобільного додатку – ReactNative [37].

3.2 Вибір інструментарію для власної реалізації класифікаційної моделі

При виборі мови програмування для реалізації поставленої завдання основними критеріями були швидкість роботи та наявність бібліотек для класифікації. Серед широко поширених та добре документованих мов програмування, найкращим чином поставленим завданням відповідає Python.

Python – високорівнева мова програмування загального призначення, орієнтований на підвищення продуктивності розробника та читання коду. Слід зазначити популярність цієї мови у науковому спільноті, адже Python з пакетами NumPy, SciPy та MatPlotLib активно використовується як універсальне середовище для наукових розрахунків як заміни поширеним спеціалізованим комерційним пакетам Matlab, IDL та іншим.

Ще один вагомий аргумент на користь використання Python – наявність великої кількості бібліотек та розширень для роботи у галузі машинного навчання, наприклад: Tensorflow, scikit-learn, Theano, Pylearn2, NuPIC.

У результаті вибір було зупинено на бібліотеці Tensorflow, оскільки вона добре документована (має докладний опис реалізованих класів і функцій, а також велика кількість прикладів, які значно прискорюють написання власного

коду), що робить її зручною та зрозумілою при використанні. Крім цього, ця бібліотека має відкритий вихідний код, доступний будь-якому користувачеві.

3.3 Огляд набору даних для нейронної мережі

Перед початком роботи з моделлю, потрібно підготувати масив зображень, на яких буде обличчя людини в масці та без. Так як у наявності є 686 зображень обличь без маски, а для навчання потрібно ці ж зображення, але в масці, то застосуємо власний класифікатор за допомогою невеликої нейронної мережі, і таким чином ми отримаємо два набори даних: перший – це датасет людей без масок, другий – в масках. Як виглядає зображення до і після можна побачити на малюнку (рис.3.2).

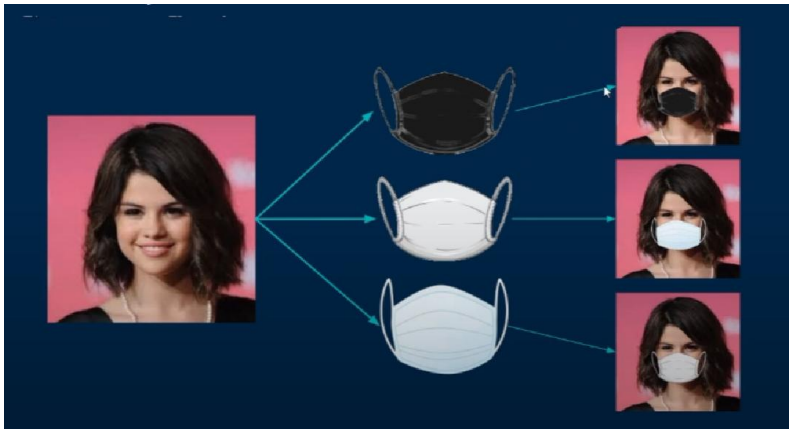

Рисунок 3.2 – Приклад зображення з тренувального набору даних

Для розпізнавання обличь та маніпулювання з ними використаємо бібліотеку Face Recognition, створену з використанням алгоритму розпізнавання обличчя dlib, що є прикладом глибокого навчання. Дана модель має точність 99,38%.

За допомогою бібліотеки отримаємо розташування та контури очей, носа, рота та підборіддя кожної людини (рис. 3.3). На основі знайдених рис обличчя нанесемо нашу медичну маску.

```
import face_recognition

face_image_np = face_recognition.load_image_file(self.face_path)
face_locations = face_recognition.face_locations(face_image_np, model=self.model)
face_landmarks = face_recognition.face_landmarks(face_image_np, face_locations)
```

Рисунок 3.3 – Вилучення основних ознак обличчя

Розглянемо основні методи бібліотеки. За допомогою методу face_locationотримаємо 2d масив обмежень людських обличь у зображенні за допомогою детектора обличьCNN. При використанні графічного процесор, це може дати набагато швидші результати, оскільки графічний процесор може обробляти пакети зображень одночасно.

Методом face_landmark отримаємо ознаки обличчя (очі, ніс тощо) для кожного обличчя на зображенні, а масив обмежувальних рамок людських обличь отримаємо методом face_landmarks. За отриманими ознаками додаємо на картинки маски.

3.4 Навчання нейронної мережі

Нейронна мережа сама по собі абсолютно нікчемна, якщо це не звучить дивно. Щоб це дало результати, його потрібно навчити. Взагалі, що таке навчання нейронної мережі? Це процес налаштування його параметрів за певним алгоритмом (такий алгоритм називається алгоритмом навчання), щоб він на виході дав правильний результат. Розрізняють алгоритми навчання з вчителем, без вчителя та з підкріпленням.

Процес навчання з вчителем застосовується тоді, коли вхідні дані вже розділені на певні класи (рис. 3.4). Тобто нейромережа знає правильні відповіді й на їх основі підлаштовує свої параметри.

Оскільки нейронна мережа знає правильну відповідь для кожної вхідної вибірки, ми можемо використовувати функцію втрат, щоб обчислити помилку,

тобто різницю між виходом нейронної мережі та правильним результатом, а потім налаштувати ваги. Повторюйте ці кроки, поки помилка не буде в межах допустимого діапазону (або до кінця кількості циклів). Якщо вхідні дані розбити на категорії, то ми вирішили задачу класифікації. Класичними алгоритмами навчання є алгоритм зворотного поширення помилок і метод градієнтного спуску.

Рисунок 3.4 – Процес навчання з вчителем

3.5 Перенавчання, регуляризація та dropout

Приступаючи до побудови моделі, необхідно згадати про дуже важливу пастку глибокого навчання – проблему перенавчання (overfitting). Дана проблема негативний помітно проявляється на мережах, подібних до тієї, що ми будуємо, а значить, необхідно захищатися від цього явища.

Перенавчання – це зайва точність відповідності нейронної мережі для конкретного набору навчальних прикладів, при якому мережа втрачає здатність до узагальнення. Іншими словами, наша модель могла вивчити навчальну множину (разом з шумом, який в ньому присутній), але вона не змогла розпізнати приховані процеси, які цю множину породили. Як приклад розглянемо задачу апроксимації синусоїди з адитивним шумом (рис. 3.5).

У нас є навчальна множина (сині кружки), отримане з вихідної кривої синуса, з деякою кількістю шуму. Якщо ми докладемо до цих даних графік

многочлена третього ступеня, ми отримаємо гарну апроксимацію вихідної кривої. Хтось заперечить, що многочлен 14-го ступеня підійшов би краще; дійсно, так як у нас є 15 точок, така апроксимація ідеально описала б навчальну вибірку. Проте, в цьому випадку введення додаткових параметрів в модель призводить до катастрофічних результатів: через те, що наша апроксимація враховує шуми, вона не збігається з вихідною кривою ніде, крім навчальних точок.

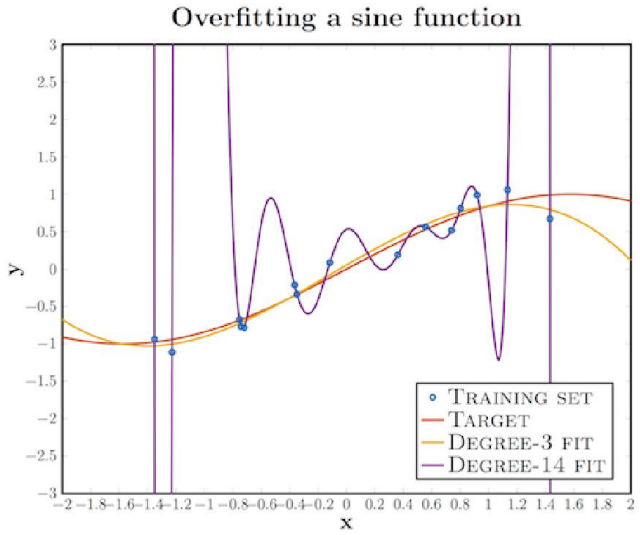

Рисунок 3.5 – Процес навчання з вчителем

У глибоких згорткових нейронних мережах маса різноманітних параметрів, особливо це стосується повнозв'язних шарів, перенавчання може проявити себе в такій формі: якщо у нас недостатньо навчальних прикладів, маленька група нейронів може стати відповідальною за більшість обчислень, а решта нейрони стануть надлишкові; або навпаки, деякі нейрони можуть нанести шкоду продуктивності, при цьому інші нейрони з їх шару не будуть займатися нічим, окрім виправлення їх помилок.

Щоб наша мережа не втратила здатність до узагальнення в цих ситуаціях, ми запровадили методи регуляризації: замість зменшення кількості параметрів ми обмежуємо параметри моделі під час навчання і не дозволяємо нейронам вивчати шум навчальних даних. Цю проблему можна вирішити шляхом відсіву,

що на перший погляд здається «чорною магією», але насправді допомагає усунути вищезгадану ситуацію. Зокрема, випадання з параметром p однієї ітерації навчання проходить через усі нейрони певного шару і повністю виключає їх із мережі з ймовірністю p під час ітерації. Це змусить мережу обробляти помилки замість того, щоб покладатися на існування конкретних нейронів (або груп нейронів), а на «консенсус» нейронів у межах одного шару. Це досить простий метод, який може ефективно вирішити проблему самостійного перенавчання без введення інших регуляризаторів. На малюнку нижче ілюструється цей метод (рисунок 3.6).

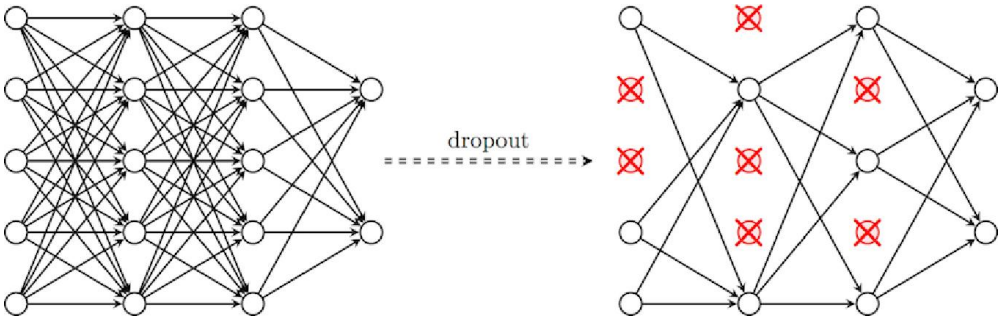

Рисунок 3.6 – Процес навчання з вчителем

3.6 Навчання нейронної мережі для пошуку осіб у медичних масках

Підготовка моделі для розпізнавання об’єктів на зображеннях виконується в такій послідовності:

- налаштування середовища;
- створення моделі;
- компіляція моделі;
- завантаження та підготовка даних;
- навчання нейронної мережі.

Перш ніж розпочати, потрібно імпортувати необхідні елементи з бібліотеки Keras та NumPy. Numpy – бібліотека мови Python, що додає підтримку великих багатовимірних масивів і матриць, разом з великою

бібліотекою високорівневих математичних функцій для операцій з цими масивами.

Перше, що нам знадобиться – це датасет (вибірка фотографій) людей з масками і без. Підготовлений раніше датасет має 686 фотографій людей у масках і 1983 без (одних і тих самих людей). Хотілося б відзначити один важливий момент, що нейромережа яку ми використовуємо – MobileNetV2, можна використовувати практично для будь-якої класифікації, наприклад, можна було навчити її для визначення статі, або спробувати автоматично визначати в окулярах людина чи ні.

Отже, розмістимо наш датасет із 686 фотографій у двох каталогах у папці dataset у масках у “with_mask” і без маски у “without_mask” відповідно. Ще одне важливе зауваження, це те, що навчаємо ми саме на обличчях, а не просто, що людина на зображенні в масці або без, хоча можна було й так.

Далі імпортуємо необхідні бібліотеки (рис. 3.7):

```
# import the necessary packages
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import os
```

Рисунок 3.7 – Перелік необхідних бібліотек

Вказуємо початкові гіперпараметри (рис.3.8). Перший – це швидкість навчання, він означає, з якою швидкістю ми рухаємося до мінімуму функції втрат – INIT_LR = 0,004.

Другий – це кількість епох, одна епоха – це один прохід навчання на всьому наборі даних – EPOCHS = 20.

Третій – це розмір пакету або батча, що означає кількість даних в одній партії. BS = 32.

Підбір гіперпараметрів у кожному разі треба підбирати емпірично.

```
# initialize the initial learning rate,
#number of epochs to train for,
# and batch size
INIT_LR = 1e-4
EPOCHS = 20
BS = 32
```

Рисунок 3.8 – Гіперпараметри

У цій папці зберігаються два каталоги з масками та без, які ми підготували раніше за допомогою скрипту (рис.3.9). Формуємо коректні шляхи до картинок, для подальшого використання.

```
# grab the list of images in our dataset directory, then initialize
# the list of data (i.e., images) and class images
print("[INFO] loading images...")
imagePaths = list(paths.list_images(dataset_path))
imagePaths = [imagePath.replace("\\","//",-1) for imagePath in imagePaths]
data = []
labels = []
```

Рисунок 3.9 – Формування шляху до навчальних наборів

Мітки класів потрібно перетворити в категорії. Тобто номера класів в мітках, ми маємо перетворити в представлення по категоріям. За допомогою циклу виконуємо вилучення класу з директорії (з маскою або без) та формуємо загальний масив зображень (рис. 3.10). Картинки отримуємо розміром (244*244).

```
# loop over the image paths
for imagePath in imagePaths:
    # extract the class label from the filename
    label = imagePath.split("//")[-2]

    # load the input image (224x224) and preprocess it
    image = load_img(imagePath, target_size=(224, 224))
    image = img_to_array(image)
    image = preprocess_input(image)

    # update the data and labels lists, respectively
    data.append(image)
    labels.append(label)
```

Рисунок 3.10 – Вилучення класів

Приводимо в NumPy масив та класи у бінарний вигляд, тобто – 0 без маски, 1 з маскою (рис. 3.11).

```
# convert the data and labels to NumPy arrays
data = np.array(data, dtype="float32")
labels = np.array(labels)

# perform one-hot encoding on the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
labels = to_categorical(labels)
```

Рисунок 3.11 – Класифікація навчального набору

Розіб'ємо датасет на тренувальний та тестовий (80% на 20%). При глибокому навчанні іноді можна зіткнутися із ситуацією, коли набір даних має обмежений розмір. Але щоб отримати кращі результати узагальнення моделі, необхідно мати більше даних, у тому числі різні їх варіації. Тобто необхідно збільшити розмір вихідного набору штучно, і це можна зробити за допомогою аугментації даних, що й ми робимо (рис. 3.12). Якщо коротко аугментація даних – це методика створення додаткових даних із наявних даних.

```
# partition the data into training and testing splits using 75% of
# the data for training and the remaining 25% for testing
(trainX, testX, trainY, testY) = train_test_split(data, labels,
    test_size=0.20, stratify=labels, random_state=42)

# construct the training image generator for data augmentation
aug = ImageDataGenerator(
    rotation_range=20,
    zoom_range=0.15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.15,
    horizontal_flip=True,
    fill_mode="nearest")
```

Рисунок 3.12 – Аугментація набору даних

Завантажуємо базову модель з попередньо навченими вагами. Вагу для моделі, можна вказати двома способами: указати параметр imagenet (попередньо навчені в ImageNet) або None(випадкова ініціалізація) (рис. 3.13).

Виконуємо операцію середньої вибірки (субдискретизації) для просторових даних (AveragePooling2D).

Вирівнювання тензора означає видалення всіх вимірів, крім одного. Шар Flatten змінює форму тензора, щоб мати форму, рівну кількості елементів, що містяться в тензорі. Це те саме, що і створення 1d-масиву елементів.

Виклик Dense(128, activation=”relu”) та Dense(2, activation=”softmax”) призведе до створення мережі Dense із 128 та 2 входами відповідно. У штучних нейронних мережах (ШНМ) функція активації є математичною залежністю вихідного сигналу штучного нейрона від вхідного, що йде на наступний шар. Функції активації є основою глибокого навчання. Вони визначають вихід моделі, її точність та обчислювальну ефективність. У деяких випадках функції активації мають значний вплив на здатність моделі до зближення та швидкість зближення. У нашому випадку ми використовуємо сигмоїду та ReLu.

Dropout застосовується у нейронних мережах для вирішення проблеми перенавчання.Dropout (0.5) означає, що нейрон буде включатись з ймовірність

50%. Іншим нейронам, під час навчання мережі, доведеться підбирати ваги так, щоб знаходити важливі ознаки самостійно, без участі сусідніх нейронів.

```
# load the MobileNetV2 network, ensuring the head FC layer sets are
# left off
baseModel = MobileNetV2(weights="imagenet", include_top=False,input_tensor=Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(7, 7))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(128, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(2, activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)
```

Рисунок 3.13 – Завантаження моделі

Також нам необхідно переконатися, що ми не змінимо заздалегідь навчену модель під час тренування. Рішення полягає у відключенні змінних попередньої моделі – ми просто заборонимо алгоритму оновлення значення при прямому і зворотному поширенні їх змінювати. Цей процес називається "заморожуванням моделі" (freezing the model) (рис. 3.14). Виконавши попередні операції компілюємо базову модель.

```
# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
    layer.trainable = False

# compile our model
print("[INFO] compiling model...")
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="binary_crossentropy", optimizer=opt,
    metrics=["accuracy"])
```

Рисунок 3.14 – Freezing the model

Створивши нейронну мережу її потрібно навчити. Для навчання мережі визиваємо метод fit (рис. 3.15). Навчання виконується на даних trainX – містять

зображення для навчання, trainY – правильні відповіді, уже перетворені в представлення по категоріям.

Так як наша мережа є глибокою, на її навчання необхідний певний час. В середньому, кожна епоха займає 70 секунд. Повне навчання мережі зайняло 23хв. Швидкість навчання сильно залежить від процесора, який використовувався. В нашому випадку використовувався тип ЦП Mobile QuadCore AMD Ryzen 5 3500U, 3700 MHz.

```
# train the head of the network
print("[INFO] training head...")
H = model.fit(
    aug.flow(trainX, trainY, batch_size=BS),
    steps_per_epoch=len(trainX) // BS,
    validation_data=(testX, testY),
    validation_steps=len(testX) // BS,
    epochs=EPOCHS)
```

Рисунок 3.15 – Навчання мережі

Для демонстрації якості навчання ми використаємо модуль sklearn.metrics, який надає набір простих функцій, що вимірюють помилку передбачення з урахуванням істинності та передбачення (рис. 3.16). В кінці навчання, після завершення 20-ї епохи, точність на даних становить 99% в масці.

	precision	recall	f1-score	support
with_mask	0.99	1.00	1.00	397
without_mask	1.00	0.99	0.99	137
accuracy			1.00	534
macro avg	1.00	0.99	1.00	534
weighted avg	1.00	1.00	1.00	534

Рисунок 3.16 – Навчання мережі

3.7 Оцінка якості моделі

Для детального розуміння процесу навчання моделі використаємо готовий інструмент мови Python: matplotlib – бібліотека для візуалізації даних. З її допомогою, можливо графічно відобразити можливість мережі вирішувати поставленні завдання в конкретний момент, тобто епоху (рис. 3.17).

Рисунок 3.17 – Точність моделі у визначений час

З графіку видно, що до 5-ї епохи модель активно навчалася, і точність швидко збільшувалась, тобто можна зробити висновок, що для навчання цієї моделі вистачило б 5 епох.

3.8 Інтеграція моделі в додаток

Отримавши готову модель, яка може розрізнити обличчя в масці, з досить високою точність, нам потрібно розгорнути її в ReactNative (рис. 3.18). З'являється запитання, чому ж розгортати модель з допомогою React Native? Приведемо три аргументовані переваги:

− доступність – оскільки розгорнувши модель на даній платформі, маємо доступ, як до GooglePlay так і до AppStore. Розмістивши додаток, ви матимете

доступ до мільйонів користувачів, які зможуть використовувати продукт на власних телефонах;

− децентралізована обробка – оскільки користувачі зможуть використати вашу модель на своєму мобільному телефоні, не потребуючи підключення до сервера для обробки даних для отримання прогнозу;

− варіант використання мобільної розробки–розгортаючи модель на мобільному пристрої, у вас є доступ до багатьох датчиків на телефоні (камера, GPSтощо).

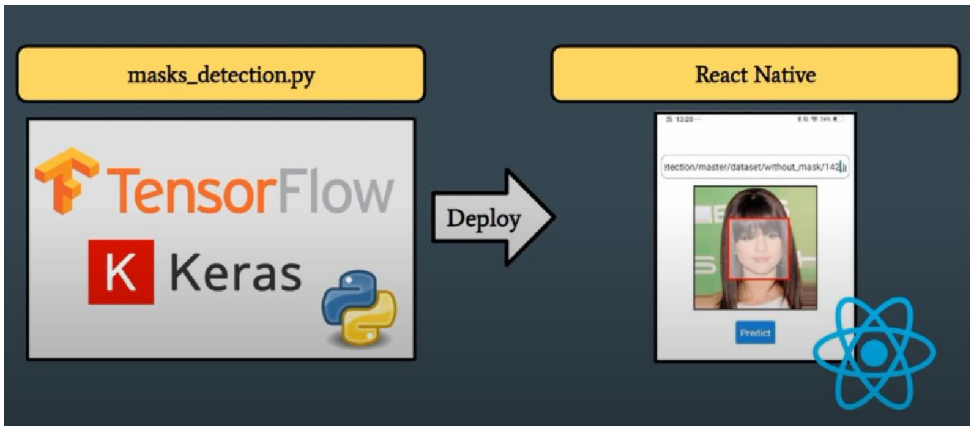

Рисунок 3.18 – Схема розгортання моделі

3.8.1 Розгортання готової моделі TensorFlow. Моделі для TensorFlow.js складаються з двох типів файлів: файл конфігурації моделі, що зберігається у форматі JSON, і ваги моделей, що зберігаються у двійковому форматі. Вага моделей часто фрагментуються на безліч частин для оптимізації кешування в браузерах.

Розгортання готової моделі можна розділити на три кроки:

− перетворення Python моделі зручної для використання мовою javascript (рис. 3.19). Тобто ми повинні спочатку навчити модель, зберегти її як файл .h5, і нарешті конвертувати модель. Як результат отримаємо файли – файл json та двійкові файли;

− комбінування отриманих двійкових файлів в один;

− завантаження моделі в ReactNative.

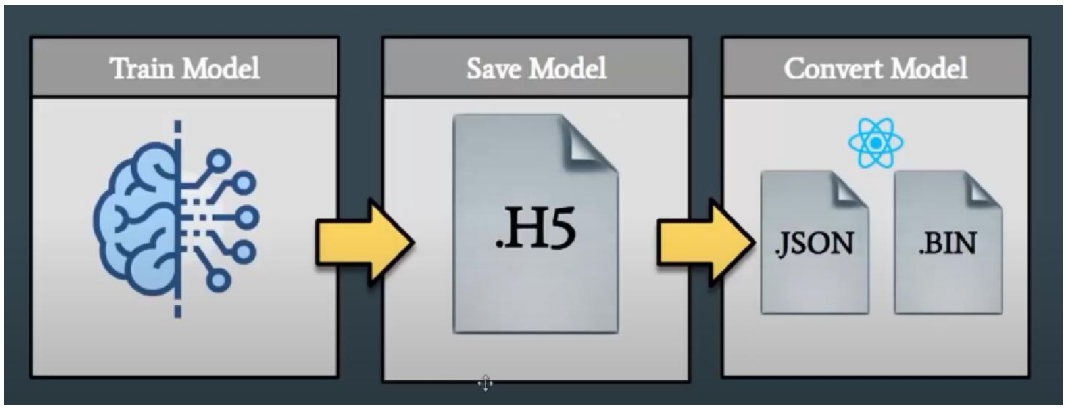

Рисунок 3.19 – Підготовка моделі

Наведені вище операції потребують використання бібліотеки tensorflow.js, яка надає нам доступ до її методів (рис. 3.20). Тож її потрібно імпортувати включно з іншими бібліотеками, які нам знадобляться для розпізнавання зображення обличчя людини в масці та без. Разом з нею ми використаємо відкриту бібліотеку blazeface– яка є легкою моделлю, що розпізнає обличчя на зображеннях. Blazeface використовує модифіковану архітектуру SingleShotDetector зі спеціальним кодером. Модель служити першим кроком для розпізнавання ключових точок обличчя.

Імпортувавши всі необхідні бібліотеки ми можемо завантажити модель та виконати необхідні операції (рис. 3.20).

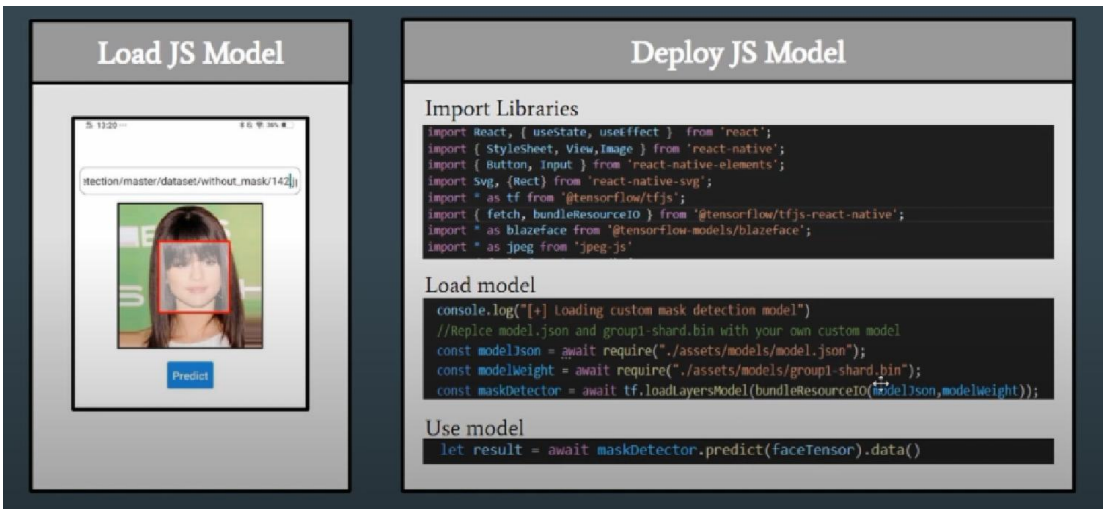

Рисунок 3.20 – Основні дії для інтеграції

3.8.2 Результат роботи програми. Функціонал програми є досить простий. На головному меню знаходиться область, в яку ми можемо вставити посилання на будь-яке зображення. Під текстовим полем розмістився блок з досліджуваною картинкою, під нею кнопка, яка запускає процес класифікації.

Натиснувши на кнопку з вказаною картинкою, ми отримаємо результат класифікації. Для обличчя без маски, з'явиться червоний блок навколо шуканого людського обличчя (рис. 3.21). Якщо людина в масці – зелений блок (рис. 3.21).

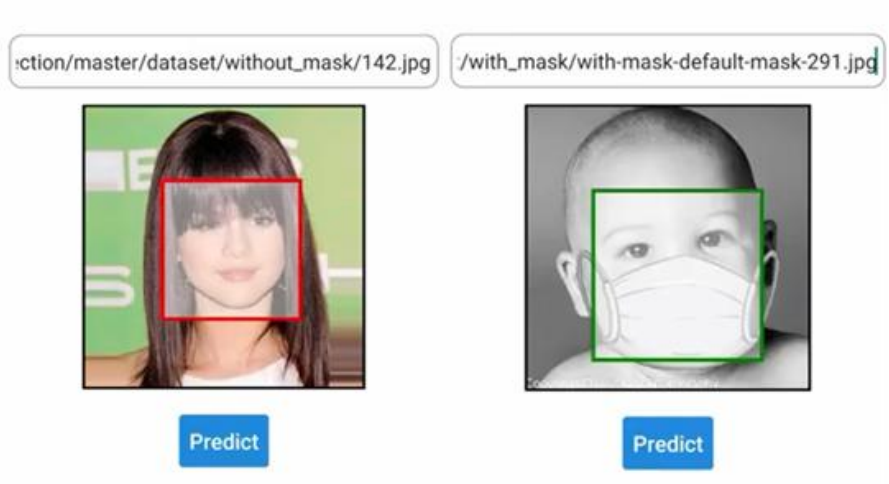

Рисунок 3.21 – Інтерфейс додатка

Висновки до третього розділу

У цьому розділі ми розгляну всі етапи інтеграції моделі в додаток, написаний на React Native. Сюди входить підготовка навчальних зображень, тобто отримання двох видів зображення, а саме людей в масці та без неї. Для цього було використано власний скрипт, який здатний визначити обличчя людей та нанести на зображення маску потрібного розміру. Маски додаються трьох різних видів.

Точність розпізнавання моделі залежить від кількості епох. В кінці навчання, після завершення 20-ї епохи, точність на перевірочних даних становила 99%. Було також досліджено, що збільшуючи кількість епох, ми б

збільшували якість нейронної мережі, але це відбувалося тільки до 5 епохи, наступні дії не давали великого приросту точності мережі.

Навчену мережу ми успішно інтегрували в додаток, і як результат змогли визначити з досить високою ймовірністю обличчя в масці та без.

ВИСНОВКИ

В ході виконання дипломної роботи було визначено поняття нейронної мережі та її великий вплив на сучасне суспільство, так як нейронні мережі являють собою систему, яка може допомогти вирішити завдання, які виходять за межі людських можливостей. Вони є тісно пов’язані з аналізом великих та складних для розуміння людини масивів даних. Це можуть бути: масиви даних у фінансовому секторі, астрономічні дані, дані для прогнозування курсу валют тощо.

У ході виконання дипломної роботи було розглянуто питання пов’язані з вибором типу нейронної мережі, яка краще справляється з завданням розпізнавання об’єктів на зображені. Наведено характеристику предметного середовища. Детально розглянуто алгоритми навчання мережі та методики для покращення точності прогнозування. Описано вимоги та технології для створення даної системи.

Для розробки системи розпізнавання об’єктів на зображенні використано мову програмування Python, середовищем для розробки обрано Jupyter Notebook. Також використано бібліотеку Tensorflow для навчання нейронної мережі та підготовки тренувального набору.

Дослідивши процес навчання моделі, можна з впевненістю сказати, що точність розпізнавання моделі залежить від кількості епох, тобто проходу по всіх елементах тренувального набору. В кінці навчання, після завершення 20-ї епохи, точність на перевірочних даних становила 99%. Було також досліджено, що збільшуючи кількість епох, ми б збільшували якість нейронної мережі, але це відбувалося тільки до 5 епохи, наступні дії не давали великого приросту точності мережі.

Результатом є додаток з успішно інтегрованою моделлю для розпізнавання людського обличчя в масці та без. Для інтеграції моделі був обраний фреймворк ReactNative, який є найбільш комфортним для інтеграції та має найкращу продуктивність серед розглянутих варіантів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Лубко Д. І. Методи та системи штучного інтелекту [Електронний ресурс] / Д. І. Лубко, С. В. Шаров // 2019 – Режим доступу до ресурсу: http://www.tsatu.edu.ua/kn/wp-content/uploads/sites/16/knyha.-msshy-v-byblyoteku.pdf.

2. Білан Н. І. Інформаційні системи та технології в управлінні [Електронний ресурс] / Н. І. Білан // 2014 – Режим доступу до ресурсу: http://eir.zntu.edu.ua/bitstream/123456789/342/1/met_vk_bila_3.pdf.

3. Pitts M. A Logical Calculus of the Ideas Immanent in Nervous Activity / M. Pitts, S. Warren, W. Pitts., 1943.

4. Paul J. W. Backpropagation Through Time: What It Does and How to Do It. / Werbos Paul., 1990.

5. Каллан Р. Основні концепції нейронних мереж / Р. Каллан., 2003.

6. Чабан Л. Н. Практикум по дискретній математиці / Л. Н. Чабан., 2010.

7. Гонсалес Р. Принципи розпізнавання образів / Р. Гонсалес., 1978.

8. Ivakhnenko A. G. Cybernetics and Forecasting Techniques / A. G. Ivakhnenko, V. G. Lapa, R. N. McDonough., 1967. – (American Elsevier).

9. Ивахненко А. Г. Кібернетичні пророчі пристрої / А. Г. Ивахненко, В. Г. Лапа., 1965. – (Наук. думка).

10. Ivakhnenko A. G. The Group Method of Data Handling – a Rival of the Method of Stochastic Approximation / Ivakhnenko., 1968.

11. Ivakhnenko A. G. Polynomial Theory of Complex Systems / Ivakhnenko., 1971.

12. Fukushima K. Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position – Neocognitron / Fukushima., 1979.

13. Fukushima K. A Self-Organizing Neural Network for a Mechanism of Pattern Recognition Unaffected by Shift in Position / Fukushima., 1980. – (Biological Cybernetics).

14. Linnainmaa S. The Representation of the Cumulative Rounding Error of an Algorithm as A Taylor Expansion of the Local Rounding Errors / Linnainmaa. – Univ. Helsinki, 1970. – (Master's thesis).

15. Dreyfus S. E. The Computational Solution of Optimal Control Problems with Time Lag / Dreyfus., 1973.

16. Werbos P. J. Applications of Advances in Nonlinear Sensitivity Analysis / Werbos., 1981.

17. Hornik K. Multilayer Feedforward Networks are Universal Approximators / K. Hornik, M. Stinchcombe, H. White., 1989.

18. Колмогоров А. Н. Про уявлення безперервних функцій декількох змінних у вигляді суперпозицій безперервних функцій однієї змінної та додавання / А. Н. Колмогоров., 1957.

19. Hastad J. Computational Limitations of Small-Depth Circuits / Hastad., 1987. – (Cambridge, MA, USA).

20. Hinton G. E. A Fast Learning Algorithm for Deep Belief Nets / G. E. Hinton, S. Osindero., 2006.

21. Salakhutdinov R. Learning Deep Generative Models / Salakhutdinov., 2006.

22. Salakhutdinov R. Deep Boltzmann Machines / R. Salakhutdinov, G. E. Hinton. – Florida, USA, 2009. – (AISTATS).

23. Man vs. computer: Bench marking machine learning algorithms for traffic sign recognition / J.Stallkamp, M. Schlipsing, J. Salmen, C. Igel., 2012.

24. LeCun Y. ICML 2013 tutorial / Y. LeCun, M. A. Ranzato., 2013.

25. A decision fusion and reasoning module for a traffic sign recognition system / [M. Meuter, C. Nunny,, S. M. Gormer та ін.]., 2011.

26. Библиотека машинного обучения Theano [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Theano/.

27. Библиотека машинного обучения Lasagne [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Lasagne/.

28. Библиотека машинного обучения Keras [Електронний ресурс] – Режим доступу до ресурсу: https://keras.io/.

29. Библиотека машинного обучения TensorFlow [Електронний ресурс] – Режим доступу до ресурсу: https://www.tensorflow.org/.

30. Hinton G. E. Reducing the Dimensionality of Data with Neural Networks / G. E. Hinton, R. R. Salakhutdinov., 2006.

31. Gated Feedback Recurrent Neural Networks / J.Chung, C. Gulcehre, K. Cho, Y. Bengio., 2015.

32. Krizhevsky A. Imagenet classification with deep convolutional neural networks / A. Krizhevsky, I. Sutskever, G. Hinton., 2012.

33. Deep Residual Learning for Image Recognition / K.He, X. Zhang, S. Ren, J. Sun., 2015.

34. Symonian K. Very deep convolutional networks for largescale image recognition / K. Symonian, A. Zisserman., 2015.

35. Ioffe S. Batch normalization: accelerating deep network training by reducing internal covariate shift / S. Ioffe, C. Szegedy., 2015.

36. Driver-activity recognition in the context of conditionally autonomous driving / C.Braunagel, E. Kasneci, W. Stolzmann, W. Rosenstiel., 2015.

37. React Native проти Ionic [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.linkedin.com/pulse/react-native/.

ДОДАТОК А. ТЕЗИСИ

MODERN TRENDS IN THE ARTIFICIAL INTELLIGENCE DEVELOPMENT

Shkarevskyi T.Y.[1], Semenyuta I.G.[1], Mavrina M.O.[2]

1) National University «Yuri Kondratyuk Poltava Polytechnic», Poltava, Ukraine;

2) Radics LLC, Kropyvnytskiy, Ukraine.

Technological innovations are a complex process with many interconnections, and the increasing speed of implementation requires the minimum amount of knowledge from the end-user for the possibility of their correct use and systematization. Therefore, theoretical research in the field of the development of artificial intelligence is a priorityscientific task [1,2]. Artificial intelligence allows us to integrate the learning process, the technological field, as well as modern medical and military developments into a single information space.

The purpose of the report is to identify key areas in the field of the development of artificial intelligence and the possibilities of its application in order to automate and improve security in various fields of human activity.

People tend to overestimate the effect of the latest technologies on the near future and underestimate their impact on the distant future. But this does not mean that the expert community, the state and business need a strategy for the development of innovative industries in the near future.

Even if we take into account the simplest industrial manipulators, in 2019 there are about 3.5 thousand people per robot [1], which indicates the need for a clear planning of needs and, accordingly, resources for their implementation in the form of final products.

In the report the trends of this scientific field was tested, which in the near future will have the greatest impact on the development of artificial intelligence and robotics [2].

These results can be used in further work for the study of intelligent systems [3].

List of references

1. Будущее за роботами: 11 трендов развития робототехники в ближайшие годы https://www.rbc.ru/trends/innovation/5d6feaba9a79479e9bfce47e.

2. Искусственный интеллект: современный подход (AIMA-2)// С. Рассел, П. Норвиг – М.: Диалектика-Вильямс, 2019. – 1408 с.

3. Искусственный интеллект и будущее человечества//Марк О'Коннелл, – М.: Форс, 2020. – 272 с.

ДОДАТОК Б. ЛІСТИНГ КОДУ

create_augmented_dataset.py

```
import os
from os import listdir
import sys
import random
import argparse
import numpy as np
from PIL import Image, ImageFile

__version__ = '0.3.0'



IMAGE_DIR = os.getcwd()+"\\images"
MASK_IMAGES=["default-mask.png","black-mask.png","blue-mask.png"]
MASK_IMAGE_PATHS =[os.path.join(IMAGE_DIR, mask_image) for mask_image in MASK_IMAGES]
FACE_FOLDER_PATH=os.getcwd()+"\\dataset\\without_mask"
AUGMENTED_MASK_PATH=os.getcwd()+"\\dataset\\with_mask"

def create_mask(face_path,mask_path,augmented_mask_path,color):
    show = False
    model = "hog"
    FaceMasker(face_path, mask_path,augmented_mask_path,color=color).mask()

class FaceMasker:
    KEY_FACIAL_FEATURES = ('nose_bridge', 'chin')

    def __init__(self, face_path, mask_path,augmented_mask_path, show=False,
model='hog',color="default"):
        self.face_path = face_path
        self.mask_path = mask_path
        self.show = show
        self.model = model
        self.augmented_mask_path=augmented_mask_path
        self._face_img: ImageFile = None
        self._mask_img: ImageFile = None
```

```
    self.color=color

def mask(self):
    import face_recognition

    face_image_np = face_recognition.load_image_file(self.face_path)
    face_locations = face_recognition.face_locations(face_image_np, model=self.model)
    face_landmarks = face_recognition.face_landmarks(face_image_np, face_locations)
    self._face_img = Image.fromarray(face_image_np)
    self._mask_img = Image.open(self.mask_path).convert("RGBA")

    found_face = False
    for face_landmark in face_landmarks:
        # check whether facial features meet requirement
        skip = False
        for facial_feature in self.KEY_FACIAL_FEATURES:
            if facial_feature not in face_landmark:
                skip = True
                break
        if skip:
            continue

        # mask face
        found_face = True
        self._mask_face(face_landmark)

    if found_face:
        if self.show:
            self._face_img.show()

        # save
        self._save()
    else:
        print('Found no face.')

def _mask_face(self, face_landmark: dict):
    nose_bridge = face_landmark['nose_bridge']
    nose_point = nose_bridge[len(nose_bridge) * 1 // 4]
```

```
        nose_v = np.array(nose_point)

        chin = face_landmark['chin']
        chin_len = len(chin)
        chin_bottom_point = chin[chin_len // 2]
        chin_bottom_v = np.array(chin_bottom_point)
        chin_left_point = chin[chin_len // 8]
        chin_right_point = chin[chin_len * 7 // 8]

        # split mask and resize
        width = self._mask_img.width
        height = self._mask_img.height
        width_ratio = 1.2
        new_height = int(np.linalg.norm(nose_v - chin_bottom_v))

        # left
        mask_left_img = self._mask_img.crop((0, 0, width // 2, height))
        mask_left_width = self.get_distance_from_point_to_line(chin_left_point, nose_point,
chin_bottom_point)
        mask_left_width = int(mask_left_width * width_ratio)
        mask_left_img = mask_left_img.resize((mask_left_width, new_height))

        # right
        mask_right_img = self._mask_img.crop((width // 2, 0, width, height))
        mask_right_width = self.get_distance_from_point_to_line(chin_right_point, nose_point,
chin_bottom_point)
        mask_right_width = int(mask_right_width * width_ratio)
        mask_right_img = mask_right_img.resize((mask_right_width, new_height))

        # merge mask
        size = (mask_left_img.width + mask_right_img.width, new_height)
        mask_img = Image.new('RGBA', size)
        mask_img.paste(mask_left_img, (0, 0), mask_left_img)
        mask_img.paste(mask_right_img, (mask_left_img.width, 0), mask_right_img)

        # rotate mask
        angle = np.arctan2(chin_bottom_point[1] - nose_point[1], chin_bottom_point[0] - nose_point[0])
        rotated_mask_img = mask_img.rotate(angle, expand=True)
```

```
        # calculate mask location
        center_x = (nose_point[0] + chin_bottom_point[0]) // 2
        center_y = (nose_point[1] + chin_bottom_point[1]) // 2

        offset = mask_img.width // 2 - mask_left_img.width
        radian = angle * np.pi / 180
        box_x = center_x + int(offset * np.cos(radian)) - rotated_mask_img.width // 2
        box_y = center_y + int(offset * np.sin(radian)) - rotated_mask_img.height // 2

        # add mask
        self._face_img.paste(mask_img, (box_x, box_y), mask_img)

    def _save(self):
        new_face_path=self.augmented_mask_path+"\\with-mask-"+self.color+"-"+self.face_path.split("\\")[-1]
        self._face_img.save(new_face_path)
        print(f'Save to {new_face_path}')

    @staticmethod
    def get_distance_from_point_to_line(point, line_point1, line_point2):
        distance = np.abs((line_point2[1] - line_point1[1]) * point[0] +
                          (line_point1[0] - line_point2[0]) * point[1] +
                          (line_point2[0] - line_point1[0]) * line_point1[1] +
                          (line_point1[1] - line_point2[1]) * line_point1[0]) / \
                   np.sqrt((line_point2[1] - line_point1[1]) * (line_point2[1] - line_point1[1]) +
                           (line_point1[0] - line_point2[0]) * (line_point1[0] - line_point2[0]))
        return int(distance)

if __name__ == '__main__':
    for MASK_IMAGE_PATH in MASK_IMAGE_PATHS:
        COLOR=MASK_IMAGE_PATH.split("\\")[-1].split(".")[0]
        FACE_IMAGE_PATHS=[os.getcwd()+"\\dataset\\without_mask\\"+path for path in
listdir(FACE_FOLDER_PATH)]
        for FACE_IMAGE_PATH in FACE_IMAGE_PATHS:
            print("face image path: ",FACE_IMAGE_PATH)
            create_mask(FACE_IMAGE_PATH,MASK_IMAGE_PATH
,AUGMENTED_MASK_PATH,COLOR)
```

train_mask_detector.py

```
# import the necessary packages
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import os

print(os.getcwd())
# construct the argument parser and parse the arguments
dataset_path=os.getcwd()+"//dataset"
model_path=os.getcwd()+"//model//mask_model"
plot_path=os.getcwd()+"//plot"


# initialize the initial learning rate, number of epochs to train for,
# and batch size
INIT_LR = 1e-4
EPOCHS = 20
BS = 32
```

```
# grab the list of images in our dataset directory, then initialize
# the list of data (i.e., images) and class images
print("[INFO] loading images...")
imagePaths = list(paths.list_images(dataset_path))
imagePaths = [imagePath.replace("\\","//",-1) for imagePath in imagePaths]
data = []
labels = []

# loop over the image paths
for imagePath in imagePaths:
        # extract the class label from the filename
        label = imagePath.split("//")[-2]
        # load the input image (224x224) and preprocess it
        image = load_img(imagePath, target_size=(224, 224))
        image = img_to_array(image)
        image = preprocess_input(image)

        # update the data and labels lists, respectively
        data.append(image)
        labels.append(label)

# convert the data and labels to NumPy arrays
data = np.array(data, dtype="float32")
labels = np.array(labels)

# perform one-hot encoding on the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
labels = to_categorical(labels)

# partition the data into training and testing splits using 75% of
# the data for training and the remaining 25% for testing
(trainX, testX, trainY, testY) = train_test_split(data, labels,
        test_size=0.20, stratify=labels, random_state=42)

# construct the training image generator for data augmentation
aug = ImageDataGenerator(
```

```
                rotation_range=20,
                zoom_range=0.15,
                width_shift_range=0.2,
                height_shift_range=0.2,
                shear_range=0.15,
                horizontal_flip=True,
                fill_mode="nearest")

# load the MobileNetV2 network, ensuring the head FC layer sets are
# left off
baseModel = MobileNetV2(weights="imagenet", include_top=False,input_tensor=Input(shape=(224, 224,
3)))

# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(7, 7))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(128, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(2, activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
                layer.trainable = False

# compile our model
print("[INFO] compiling model...")
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="binary_crossentropy", optimizer=opt,
                metrics=["accuracy"])

# train the head of the network
```

```
print("[INFO] training head...")
H = model.fit(
		aug.flow(trainX, trainY, batch_size=BS),
		steps_per_epoch=len(trainX) // BS,
		validation_data=(testX, testY),
		validation_steps=len(testX) // BS,
		epochs=EPOCHS)

# make predictions on the testing set
print("[INFO] evaluating network...")
predIdxs = model.predict(testX, batch_size=BS)

# for each image in the testing set we need to find the index of the
# label with corresponding largest predicted probability
predIdxs = np.argmax(predIdxs, axis=1)
# show a nicely formatted classification report
print(classification_report(testY.argmax(axis=1), predIdxs,
		target_names=lb.classes_))

# serialize the model to disk
print("[INFO] saving mask detector model... path: %s"%(model_path+".h5"))
model.save(model_path+".h5")



# plot the training loss and accuracy
N = EPOCHS
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig(plot_path)
```

App.js

```
import React, { useState, useEffect }  from 'react';
import { StyleSheet, View,Image } from 'react-native';
import { Button, Input } from 'react-native-elements';
import Svg, {Rect} from 'react-native-svg';
import * as tf from '@tensorflow/tfjs';
import { fetch, bundleResourceIO } from '@tensorflow/tfjs-react-native';
import * as blazeface from '@tensorflow-models/blazeface';
import * as jpeg from 'jpeg-js'
export default function App() {
   const [imageLink,setImageLink] =
useState("https://raw.githubusercontent.com/ohyicong/masksdetection/master/dataset/without_mask/142.jpg
")
   const [isEnabled,setIsEnabled] = useState(true)
   const [faces,setFaces]=useState([])
   const [faceDetector,setFaceDetector]=useState("")
   const [maskDetector,setMaskDetector]=useState("")
useEffect(() => {
     async function loadModel(){
console.log("[+] Application started")
       //Wait for tensorflow module to be ready
       const tfReady = await tf.ready();
console.log("[+] Loading custom mask detection model")
       //Replce model.json and group1-shard.bin with your own custom model
       const modelJson = await require("./assets/model/model.json");
       const modelWeight = await require("./assets/model/group1-shard.bin");
       const maskDetector = await tf.loadLayersModel(bundleResourceIO(modelJson,modelWeight));
console.log("[+] Loading pre-trained face detection model")
       //Blazeface is a face detection model provided by Google
       const faceDetector =  await blazeface.load();
       //Assign model to variable
       setMaskDetector(maskDetector)
       setFaceDetector(faceDetector)
console.log("[+] Model Loaded")
     }
loadModel()
   }, []);
```

```
    function imageToTensor(rawImageData){
      //Function to convert jpeg image to tensors
      const TO_UINT8ARRAY = true;
      const { width, height, data } = jpeg.decode(rawImageData, TO_UINT8ARRAY);
      // Drop the alpha channel info for mobilenet
      const buffer = new Uint8Array(width * height * 3);
      let offset = 0; // offset into original data
      for (let i = 0; i <buffer.length; i += 3) {
        buffer[i] = data[offset];
buffer[i + 1] = data[offset + 1];
buffer[i + 2] = data[offset + 2];
        offset += 4;
      }
      return tf.tensor3d(buffer, [height, width, 3]);
    }
    const getFaces = async() => {
try{
console.log("[+] Retrieving image from link :"+imageLink)
        const response = await fetch(imageLink, {}, { isBinary: true });
        const rawImageData = await response.arrayBuffer();
        const imageTensor = imageToTensor(rawImageData).resizeBilinear([224,224])
        const faces = await faceDetector.estimateFaces(imageTensor, false);
        var tempArray=[]
        //Loop through the available faces, check if the person is wearing a mask.
        for (let i=0;i<faces.length;i++){
          let color = "red"
          let width = parseInt((faces[i].bottomRight[1] - faces[i].topLeft[1]))
          let height = parseInt((faces[i].bottomRight[0] - faces[i].topLeft[0]))
          let
faceTensor=imageTensor.slice([parseInt(faces[i].topLeft[1]),parseInt(faces[i].topLeft[0]),0],[width,height,3]
)
          faceTensor = faceTensor.resizeBilinear([224,224]).reshape([1,224,224,3])
          let result = await maskDetector.predict(faceTensor).data()
          //if result[0]>result[1], the person is wearing a mask
          if(result[0]>result[1]){
            color="green"
          }
          tempArray.push({
```

```
id:i,
location:faces[i],
color:color
        })
      }
      setFaces(tempArray)
console.log("[+] Prediction Completed")
}catch{
console.log("[-] Unable to load image")
    }

  }
 return (
<View style={styles.container}>
<Input
      placeholder="image link"
      onChangeText = {(inputText)=>{
        console.log(inputText)
        setImageLink(inputText)
        const elements= inputText.split(".")
        if(elements.slice(-1)[0]=="jpg" || elements.slice(-1)[0]=="jpeg"){
          setIsEnabled(true)
}else{
          setIsEnabled(false)
        }
      }}
      value={imageLink}
      containerStyle={{height:40,fontSize:10,margin:15}}
      inputContainerStyle={{borderRadius:10,borderWidth:1,paddingHorizontal:5}}
      inputStyle={{fontSize:15}}

    />
<View style={{marginBottom:20}}>
<Image
        style={{width:224,height:224,borderWidth:2,borderColor:"black",resizeMode: "contain"}}
        source={{
          uri: imageLink
        }}
```

```
        PlaceholderContent={<View>No Image Found</View>}
      />
<Svg height="224" width="224" style={{marginTop:-224}}>
        {
         faces.map((face)=>{
          return (
<Rect
              key={face.id}
              x={face.location.topLeft[0]}
              y={face.location.topLeft[1]}
              width={(face.location.bottomRight[0] - face.location.topLeft[0])}
              height={(face.location.bottomRight[1] - face.location.topLeft[1])}
              stroke={face.color}
              strokeWidth="3"
              fill=""
            />
          )
         })
        }
</Svg>
</View>
<Button
        title="Predict"
        onPress={()=>{getFaces()}}
        disabled={!isEnabled}
      />
</View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```