Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Факультет комп'ютерних наук
Кафедра теоретичної та прикладної системотехніки

«Затверджую»
Зав. кафедри теоретичної та
прикладної системотехніки
_____________ д.т.н., проф. С. І. Шматков
«__» ________ 2022 р

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «МЕТОДИ НАВЧАННЯ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ НА ОСНОВІ ПОПУЛЯЦІЙНОГО АЛГОРИТМУ»

Захищено на засіданні
Атестаційної комісії № 45
протокол № __ від __.12.2022 р.
Оцінка _____ / _______________
Голова Атестаційної комісії
_________________ Мінухін С. В.
(підпис) (прізвище та ініціали)

Виконав:
студент 2 курсу, групи КУ– 61
Галузь знань: 15 – Автоматизація та приладобудування
Спеціальність: 151 – «Автоматизація та комп'ютерно-інтегровані технології»
Безсмертний Денис Русланович
(прізвище, ім'я та по батькові)
____
(підпис)

Керівник: Шматков Сергій Ігоревич
(прізвище, ім'я та по батькові)
____
(підпис)

Рецензент: Хруслов Максим Михайлович
____
(підпис)

Харків – 2022

АНОТАЦІЯ

Пояснювальна записка до магістерської атестаційної роботи складається зі вступу, трьох розділів, висновків, списку використаних джерел і чотирьох додатків. Загальний обсяг роботи складає 93 сторінки, із яких 52 сторінки основної частини з 29 рисунками, 1 таблицею, 8 формулами та 23 найменуваннями списку використаних джерел та чотирма додатками.

Метою кваліфікаційної роботи є аналіз методів навчання згорткової нейронної мережі та розробка програмної моделі згорткової нейронної мережі на основі популяційного алгоритму на мові програмування Python з використанням хмарової платформи Google Colaboratory.

Об'єкт дослідження – процес навчання згоркової нейронної мережі на основі популяційного алгоритму.

Предмет дослідження – методи та алгоритми навчання згорткової нейронної мережі на основі популяційного алгоритму.

Проблема, яка вирішується в кваліфікаційній роботі є комбінування популяційного алгоритму та згорткову нейронну мережу.

Область застосування – використання нейронної мережі у різних цілях (розпізнавання образів, аналіз даних). Розроблений програмний продукт може широко використовуватися в сфері аналізу або розпізнавання.

Ключові слова: методи навчання, згорткова нейронна мережа, популяційний алгоритм, Python, Google Colaboratory, програмна модель.

ABSTRACT

The explanatory note to the master's attestation work consists of an introduction, three sections, conclusions, a list of used sources and four appendices. The total volume of the work is 93 pages, of which 52 pages are the main part with 29 figures, 1 table, 8 formulas and 23 names of the list of used sources and four appendices.

The purpose of the qualification work is the analysis of convolutional neural network learning methods and the development of a software model of a convolutional neural network based on a population algorithm in the Python programming language using the Google Colaboratory cloud platform.

The object of research – is the learning process of a convolutional neural network based on a population algorithm.

The subject of the research – methods and algorithms for training a convolutional neural network based on a population algorithm.

The problem that is solved in the qualification work is the combination of the population algorithm and the convolutional neural network.

The field of application – is the use of a neural network for various purposes (pattern recognition, data analysis). The developed software product can be widely used in the field of analysis or classification.

Keywords: learning methods, convolutional neural network, population algorithm, Python, Google Colaboratory, software model.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ 6

ВСТУП 7

РОЗДІЛ 1 АНАЛІЗ МЕТОДІВ НАВЧАННЯ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ НА ОСНОВІ ПОПУЛЯЦІЙНОГО АЛГОРИТМУ 9

1.1 Аналіз нейронних мереж 9

1.1.1 Принципи побудови штучних нейронних мереж 9

1.1.2 Архітектура нейронних мереж 11

1.1.3 Згорткові нейронні мережі 15

1.2 Аналіз методів навчання нейронних мереж 17

1.3 Аналіз популяційних алгоритмів 20

1.3.1 Історія та види популяційних алгоритмів 20

1.3.2 Аналіз окремих видів популяційних алгоритмів 21

Висновок за розділом 1 26

РОЗДІЛ 2 РОЗРОБКА МОДЕЛІ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ НА ОСНОВІ ПОПУЛЯЦІЙНОГО АЛГОРИТМУ 28

2.1 Опис вхідних даних 28

2.2 Аналіз структури згорткової нейронної мережі 29

2.3 Аналіз алгоритму рою часток 33

Висновок за розділом 2 39

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ МОДЕЛІ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ НА ОСНОВІ ПОПУЛЯЦІЙНОГО АЛГОРИТМУ 40

3.1 Інструментальні засоби реалізації моделі 40

3.2 Програмна реалізація моделі згорткової нейронної мережі на основі популяційного алгоритму 42

3.3 Дослідження моделі згорткової нейронної мережі на основі популяційного алгоритму 52

Висновок за розділом 3 54

ВИСНОВОК 55

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 57

ДОДАТОК А 61

ДОДАТОК Б 64

ДОДАТОК В 67

ДОДАТОК Г 70

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

PSO – рій часток

CNN – згорткова нейронна мережа

НМ - нейронна мережа

ГА – генетичний алгоритм

ПА – популяційний алгоритм

ПМ – програмна модель

MNIST – Modified National Institute of Standards and Technology

ВСТУП

На сьогоднішньому етапі розвитку суспільства створення комп’ютерних інтерфейсів, систем прийняття швидких рішень, аналізу даних та класифікації за допомогою нейронних мереж набуває максимально швидких обертів, це спрощує користування складною системою та обробку великих обсягів даних.

Актуальність роботи. Використання нейронних мереж у різних галузях набирає великих обертів, обробка великих обсягів даних та їх класифікація найчастіше робиться за допомогою нейронних мереж. Це обумовлено тим, що можна навчити модель класифікувати образи на основі навчальних даних і потім завантажувати любий об’єм даних такого ж типу та отримувати швидкий результат по кожному об’єкту. Також існують різні способи навчання: звичайні, гібридні та кастомні з використанням різних алгоритмів, кожен метод має свої плюси та мінуси при використанні різних даних, ситуацій, а також швидкість навчання залежить від обраного методу.

Варто відмітити, що останнім часом розпізнавання образів користується все більшою популярністю, адже розпізнавання зображення, тексту чи мови, а також різноманітних явищ сприяє спрощенню комунікативного зв’язку людини з комп’ютером, допомагає застосовувати різні системи штучного інтелекту, в тому числі в системах відеоспостереження. Можливість сприймати зовнішній світ у формі образів сприяє передумовам дослідження властивостей величезної кількості об’єктів завдяки ознайомленню з кінцевою їх кількістю, а об’єктивна ознака засадничої властивості образів допомагає створювати модель їх розпізнавання. Крім того, системи відеоспостереження, що використовують штучний інтелект, на сьогоднішній день активно розвиваються та з успіхом

починають застосовуватись на масштабному рівні. Покращується як наукова база штучних нейронних мереж, так і обчислювальні потужності технічного обладнання.

У зв'язку з вищевикладеним, тема кваліфікаційної роботи спрямована на аналіз методів навчання згорткової нейронної мережі і розробку моделі згорткової нейронної мережі на основі популяційного алгоритму.

Метою дослідження є аналіз методів навчання згорткової нейронної мережі та розробка програмної моделі згорткової нейронної мережі на основі популяційного алгоритму.

Об'єкт дослідження – процес навчання згоркової нейронної мережі на основі популяційного алгоритму.

Методи дослідження: методи навчання нейронних мереж,методи побудови нейронних мереж, основи популяційних алгоритмів, методи комбінування нейронних мереж та популяційних алгоритмів.

Предмет дослідження – методи та алгоритми навчання згорткової нейронної мережі на основі популяційного алгоритму.

Завдання дослідження:

1. Виконати аналіз нейронних мереж.
2. Виконати аналіз існуючих методів навчання згорткової нейронної мережі.
3. Виконати аналіз популяційних алгоритмів.
4. Розробити програмну модель згорткової нейронної мережі на основі популяційного алгоритму та генетичного алгоритму.
5. Проведення порівняльного аналізу двох моделей.

РОЗДІЛ 1
АНАЛІЗ МЕТОДІВ НАВЧАННЯ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ НА ОСНОВІ ПОПУЛЯЦІЙНОГО АЛГОРИТМУ

1.1 Аналіз нейронних мереж

1.1.1 Принципи побудови штучних нейронних мереж

Незважаючи на значні відмінності, різні типи нейронних мереж (НМ) мають кілька спільних рис, які описують їх роботу. Основа будь-якої НМ досить проста, здебільшого однотипна, її елементи імітують роботу нейронів мозку. Далі нейрон буде означати штучний нейрон, а саме комірку нейронної мережі. Кожен нейрон має власний поточний стан, аналогічний нервовим клітинам у мозку, які можуть бути активними або неактивними. Він має набір синапсів – односторонніх вхідних з’єднань, які з’єднуються з виходами інших нейронів, а також має аксон - вихідне з’єднання цього нейрона, з якого сигнал (активний чи ні) надходить до синапсів інших нейронів[1].

Структура найпростішої НМ представлена на рисунку нижче (рис. 1.1). Блакитним кольором позначені нейрони вхідного шару, зеленим кольором – нейрони прихованого шару, а червоним – нейрон вихідного шару.

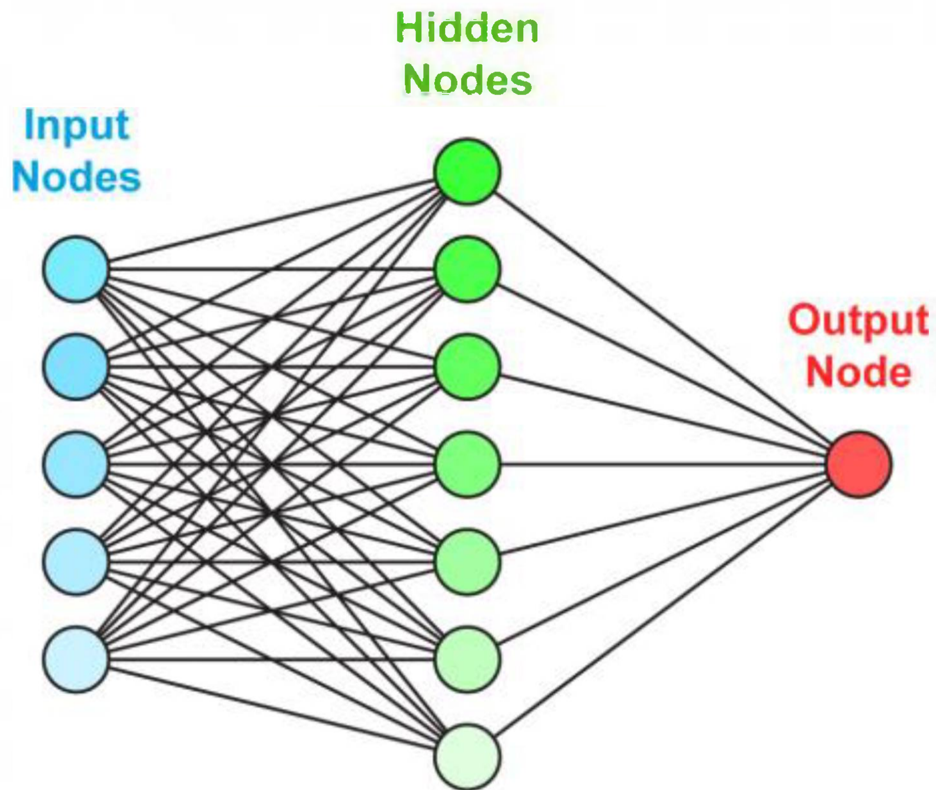


Рисунок 1.1 – Найпростіша нейронна мережа.

Кожен штучний нейрон, повинен мати вхід, через який він отримує сигнал. А також повинні бути ваги, на які помножуються вхідні сигнали, що проходять по зв'язку. На рисунку, який можна побачити вище ваги – це лінії від кожного нейрона.

Сигнали, що надходять на входи, помножуються на їх вагу. Сигнал першого входу x_1 множиться на вагу w_k, відповідну цьому входу. В результаті отримуємо x_1w_k. І так до m-го входу. У підсумку на останній вхід отримуємо x_mw_{km}.

Після цих дій усі результати передаються в суматор. Він рахує всі вхідні сигнали які були помножені на їх відповідні ваги (1.1).

$$x_1 w_k + x_2 w_{k2} + \cdots + x_m w_{km} = \sum_{i=1}^{m} x_i w_i \tag{1.1}$$

В результаті роботи суматора отримується число, яке називається зваженою сумою. Зважена сума (net) – це сума вхідних сигналів, помножених на відповідні їм ваги (1.2).

$$net = \sum_{i=1}^{m} x_i w_i \tag{1.2}$$

Не дуже доцільно подавати зважену суму на вихід. Для побудови нормального вихідного сигналу нейрону необхідно якимось чином обробити цю зважену суму. Саме для цього і існує функція активації.

Вона конвертує зважену суму в число, що є виходом нейрона (вихід нейрона можна позначити змінною out).

У різних моделях нейронних мереж використовуються різні функції активації. У загальному випадку їх можна позначити символом ϕ(net). Зважений сигнал в дужках означає, що функція активації приймає зважену суму як параметр.

Функція активації ϕ (net) – функція, яка приймає зважену суму як параметр. Значення цієї функції і є виходом нейрона (1.3) [2].

$$out = \phi(net) \tag{1.3}$$

1.1.2 Архітектура нейронних мереж

Абсолютно кожна нейронна мережа має включати в себе вхідний шар нейронів (тобто перший шар). Цей шар не виконує жодних перетворень чи обчислень, його завдання інше: приймати та розподіляти вхідні сигнали іншим нейронам. І цей шар є єдиним правилом для всіх типів нейронних мереж, а критерієм поділу є подальша структура.

1. Моношарова структура нейронної мережі. Це структура взаємодії нейронів, при якій сигнали з вхідного шару негайно надходять на вихідний, який, строго кажучи, не тільки перетворює сигнал, але й негайно дає відповідь. Як вже було сказано, 1-й вхідний шар виконує прийняття і розподіл, а обчислення які необхідні вже відбуваються на другому шарі. Вхідні нейрони з’єднані з основним шаром за допомогою синапсів різної ваги, що забезпечує якість зв’язків [3].

2. Багатошарова нейронна мережа. У даному типі структури крім вхідного і вихідного шарів, є декілька прихованих (проміжних) шарів. Кількість цих шарів залежить від складності нейронної мережі. Така структура більше нагадує тип біологічної нейронної мережі. Такі типи були розроблені зовсім недавно, раніше всі процеси реалізовувалися за допомогою моношарових нейронних мереж. Відповідні рішення мають більші можливості в порівнянні зі звичайними рішеннями, тому що в процесі обробки даних кожен проміжний рівень є проміжним етапом, на якому інформація обробляється і поширюється на наступні шари [4].

Додатково, крім значення кількості шарів, можна розділити нейронні мережі по напрямку руху інформації.

1. Нейронні мережі прямого поширення. При такій структурі сигнал поширюється строго в напрямку від вхідного шару до вихідного. Переміщення сигналу в зворотному напрямку не виконується і є принципово неможливим. На сьогоднішній день розробки моделей на основі такої технології мають широке поширення і на сьогоднішній день успішно вирішують завдання розпізнавання образів, прогнозування та групування [5].

2. Нейронні мережі зі зворотними зв’язками. У цьому випадку інформація йде в прямому і зворотному напрямках. Тому вихідний результат можна відстежити назад до вхідного. Вихід нейрона визначається ваговими

властивостями та вхідними сигналами, а також доповнюється попередніми виходами, які відстежуються до входу. Ці нейронні мережі характеризуються функцією короткочасної пам'яті, на основі якої сигнали відновлюються і доповнюються в міру їх обробки [6].

Аналіз деяких типів нейронних мереж:

1. Машина Больцмана має схожість з мережею Хопфілда, але в ній деяка частина нейронів, позначена як вхідні, а інші як приховані. Потім відбувається перетворення вхідних нейронів у вихідні. Машина Больцмана є стохастичною мережею. Навчання такого типу мережі виконується на основі зворотного поширення помилки або порівняльному алгоритму дивергенції. Загалом навчання відбувається за схожим сценарієм як у мережі Хопфілда [7].

2. Мережі радіальних базисних функцій (radial basis functions networks) це спеціальний тип нейронних мереж із прямими зв'язками. Основне їхнє призначення – апроксимація багатовимірних функцій. Вони запропоновані 1985 р. Повеллом. Їхню математичну основу складає теорія апроксимації багатовимірних функцій. Наскільки завгодно точна апроксимація функцій досягається при цьому шляхом комбінації радіально симетричних функцій.

Ці мережі мають такі властивості: архітектура мереж з прямими зв'язками 1-го порядку (FF-мережі), швидке навчання, відсутність "патологій" збіжності, на відміну від Backpropagation-мереж не виникає проблеми локальних мінімумів, Більш тривалий час їх підготовки та налаштування з -за необхідності обчислення складніших розрахунків, хороші апроксиматори функцій. Структура цих мереж містить один вхідний шар, один прихований шар нейронів, число яких зазвичай відповідає числу елементів навчальної послідовності, і один вихідний шар з одного або декількох нейронів [5].

3. Мережа типу deep belief – це структура глибокої мережі, де вона складається з взаємопов'язаних RBM або VAE деякої кількості. Цей тип мережі вивчається опосередковано, де кожен блок повинен мати можливість кодувати попередній. Такий тип навчання ще називають "жадібним навчанням", суть якого є вибір локальних та оптимальних рішень, але вони не будуть гарантувати оптимального кінцевого результату. Крім того, такий тип мережі можна навчати за допомогою метода зворотного поширення помилки, щоб представити дані як імовірнісні моделі. Якщо використовується неконтрольоване навчання (без вчителя), готову модель можна буде використовувати для виявлення нових даних [8].

4. Згорткові нейронні мережі сильно відрізняються від інших типів мереж, тому що вони були розроблені під один тип задач. В багатьох випадках вони використовуються для обробки і розпізнавання зображень, набагато рідше для аудіо. Взагалі згорткові нейронні мережі використовуються для однієї задачі – розпізнавання зображень: наприклад є малюнок кота, мережа визначить, що на малюнку є кіт, якщо на малюнку буде собака, вона визначить, що в цьому випадку на малюнку вже собака. Особливістю такого типу мереж є використання так званого "фільтру", який не обробляє всі дані за раз. Наприклад буде зображення 200x200 пікселів, 40 000 пікселів не буде оброблено за один раз. Замість цього береться масив розміром 20 x 20 (як правило починаючи з верхнього лівого кута), потім фільтр переміщується на один піксель і обробляє новий квадрат. Потім оброблені масиви переходять через згорткові шари нейронної мережі, на яких не всі вузли з'єднані між собою. Глибина цих шарів як правило зменшується глибиною використовуючи ступінь двійки: 64, 32, 16, 8, 4, 2, 1. На практиці до кінця згорткової нейронної мережі додається FFNN для того, щоб надалі обробляти дані [9].

Можна зробити висновок, що кожен тип нейронної мережі спеціалізується в якійсь певній області. Саме згорткові нейронні мережі (CNN) мають найбільші перспективи, оскільки розпізнавання зображень стає все більш актуальною задачею. Тому було обрано саме цей тип нейронних мереж.

1.1.3 Згорткові нейронні мережі

Згорткові нейронні мережі працюють за схемою прямого поширення (односпрямовані).

Згорткові нейронні мережі – це алгоритм глибокого навчання, який може сприймати вхідне зображення, призначати важливість (вагові значення та зміщення) різним аспектам/об’єктам зображення та мати можливість відрізнити один від іншого. Попередня обробка, яка необхідна в згортковій нейронній мережі, набагато менша порівняно з іншими алгоритмами класифікації. У той час як у примітивних методах фільтри розробляються вручну, після достатнього навчання, згорткова нейронна мережа має можливість вивчати ці фільтри/характеристики.

CNN може успішно фіксувати просторові та часові залежності в зображенні за допомогою застосування відповідних фільтрів. Архітектура забезпечує кращу адаптацію до набору даних зображення завдяки зменшенню кількості залучених параметрів і можливості повторного використання вагових коефіцієнтів. Іншими словами, мережу можна навчити краще розуміти складність зображення.

CNN тепер забезпечують більш масштабований підхід до задач класифікації зображень і розпізнавання об’єктів, використовуючи принципи лінійної алгебри, зокрема множення матриць, для ідентифікації шаблонів у зображенні. Тим не менш, вони можуть бути вимогливими до обчислень, вимагаючи графічних процесорів для навчання моделей.

CNN відрізняються від інших нейронних мереж своєю чудовою продуктивністю з зображенням, мовленням або аудіосигналом. Вони мають три основні типи шарів, а саме:

1. Convolutional layer – шар здійснить набір вихідних нейронів, які будуть пов'язані з локальною областю вхідного вихідного зображення. Кожен такий нейрон буде обчислювати скалярний твір між своїми вагами і невеликою частиною вихідного зображення з яким він пов'язаний.
2. Pooling layer – шар здійснить операцію семплювання зображення за двома вимірами - висотою і шириною, що в результаті дасть нам нове 3D-подання.
3. Fully-connected (FC) layer – підраховує оцінки за класами.

Згортковий шар є першим шаром згорткової мережі. У той час як за згортковими шарами можуть слідувати додаткові згорткові шари або шари об’єднання, повнозв’язковий шар є останнім. З кожним шаром згорткової нейронної мережі зростає у своїй складності, ідентифікуючи більші частини зображення. Попередні шари зосереджені на простих функціях, таких як кольори та краї. Коли дані зображення просуваються між шарами згорткової нейронної мережі, вони починають розпізнавати більші елементи або форми об’єкта, поки нарешті не ідентифікують запланований об’єкт. Структуру згорткової нейронної мережі можна побачити на рисунку (рис. 1.2).

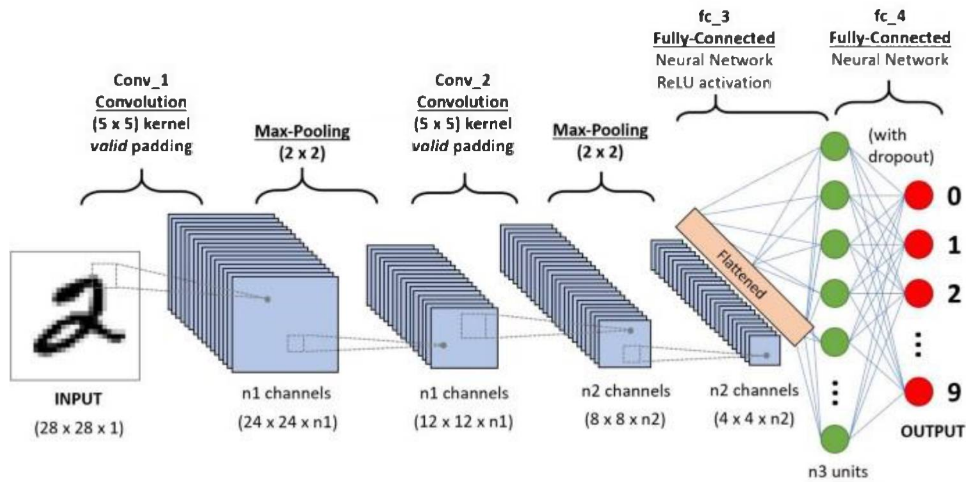


Рисунок 1.2 – Приклад згорткової мережі.

Архітектура згорткових нейронних мереж явно передбачає отримання на вході зображень, що дозволяє врахувати певні властивості вхідних даних у самій архітектурі мережі. Ці властивості дозволяють реалізувати функцію прямого поширення ефективніше та сильно зменшують загальну кількість параметрів у мережі. Тому можна провести експеримент і перевірити, як згорткова нейронна мережа впорається з розпізнаванням образів на основі популяційного алгоритму [9].

1.2 Аналіз методів навчання нейронних мереж

Існує кілька методів навчання нейронної мережі, тому можна виділити чотири найцікавіших та найпопулярніших.

1. Метод зворотного поширення (Backpropagation).

Метод зворотного поширення помилки – один із відомих методів, який використовують для глибокого навчання нейронних мереж прямого поширення (такі мережі називають багатошаровими персептронами). Цей метод відповідає методу навчання з учителем, тому необхідно ставити в навчальних прикладах цільові значення.

Мета навчання нейронної мережі під час використання алгоритму зворотного поширення помилки – це процес підстроювання вагів нейронної мережі, який дозволить при додатку певної безлічі входів отримати необхідну кількість виходів нейронів (вихідних нейронів). Можна назвати цю групу множини входів і виходів - векторами. У процесі навчання передбачається, що з будь-якого вхідного вектора існує цільовий вихідний вектор, парний вхідному і який задає необхідний вихід. Цю пару називають навчальною [10].

2. Метод гнучкого поширення (Resilient propagation або Rprop).

Алгоритм Rprop, або пороговий алгоритм зворотного розповсюдження помилки, реалізує наступну евристичну стратегію зміни кроку збільшення параметрів для багатошарових нейронних мереж.

Багатошарові мережі зазвичай використовують функції сигмоїдної активації в прихованих шарах. Ці функції відносяться до класу функцій зі стисненим відображенням, оскільки вони відображають нескінченний діапазон значень аргументу кінцевий діапазон значень функції. Сигмоїдальні функції характеризуються тим, що їхній нахил наближається до нуля, коли значення входу нейрона суттєво зростають. Наслідком цього є те, що при використанні методу якнайшвидшого спуску величина градієнта стає малою і призводить до малих змін параметрів, що настроюються, навіть якщо вони далекі від оптимальних значень.

Мета порогового алгоритму зворотного поширення помилки Rprop (Resilient propagation) полягає в тому, щоб підвищити чутливість методу при великих значеннях входу функції активації. І тут замість значень самих похідних використовується лише їх знак [11].

3. Генетичний Алгоритм (ГА).

ГА – це адаптивний евристичний метод пошуку, що є імовірнісним алгоритмом пошуку, заснований на механіці природного відбору та природній генетиці. Він застосовується у двох випадках: для підстроювання ваг прихованих та вихідних шарів нейронної мережі або знаходити параметри нейронної мережі, такі як кількість шарів, кількість нейронів, функція активації, тобто виконується структурна оптимізація.

Цей алгоритм містить такі процедури: формування початкової популяції, оператор кросинговера, мутація, оцінка пристосованості особин, селекція. Населення містить безліч альтернативних рішень, представлених у вигляді особин популяції. Алгоритм завершує свою роботу, якщо значення похибки розпізнавання найкращої особи популяції не змінюється n популяцій. Чим більше n, тим менша похибка розпізнавання і точніше нейронна мережа [12].

4. Популяційний алгоритм (ПА).

Єдиний більш-менш відомий підхід використання ПА з нейронними мережами це пошук гіперпараметрів нейронної мережі. Цей спосіб нагадує генетичний алгоритм, де він вибудовує структуру нейронної мережі, у разі популяційного алгоритму використовується для знаходження оптимальних параметрів шару нейронної мережі. Дослідження показали, що при такому комбінуванні застрявання в глобальному мінімумі виникає набагато менше.

Через свою складність оптимізація гіперпараметрів глибоких нейронних мереж виявляється складною та вимагає підходу, який виходить за межі того, що використовується для оптимізації класичного машинного навчання. Потрібно охопити великий простір пошуку, щоб гарантувати, що алгоритм не застрягне в локальному мінімумі, і водночас потрібно обмежити кількість моделей, які навчаються, до прийнятої кількості. Для такої ситуації на сьогоднішній день підходи стохастичного пошуку на основі багатьох

популяцій визнані найбільш перспективними. До них належать генетичні алгоритми, диференціальна еволюція, оптимізація плодової мушки, оптимізація колонії мурашок і оптимізація рою частинок.

Потрібно визначити гіперпараметри, які являють собою набір, який містить, по-перше, параметри згорткового шару, який в свою чергу містить кількість фільтрів та розмір фільтра, по-друге, параметри Fully-connected шару, який містить розмір шару. Запропонований метод оптимізації гіперпараметрів спрямований на виявлення гіперпараметрів, які максимізують точність класифікації згорткової нейронної мережі [13].

Після аналізу деяких типів навчання нейронної мережі, був обраний метод на основі популяційних алгоритмів, так як використання такого підходу вирішує проблеми звичайних типів навчання, та дослідження показують, що якість розпізнавання збільшується.

1.3 Аналіз популяційних алгоритмів

1.3.1 Історія та види популяційних алгоритмів

Ще з незапам’ятних часів людей цікавила поведінка тварин у групі, так звана поведінка у стані рою, яким способом функціонують птахи, коли зграя прямує в теплі країни, як забезпечує себе їжею рій бджіл, яким способом виживає колонія мурах, створюючи складні структури, як поводяться риби. косяку, адже так синхронізовано їхню поведінку. Організація особин у соціумі, як у присутності єдиного управляючого розуму, попри децентралізацію управління, деякі закономірності злагодженого цілісного організму, підглянуті взаємозв'язки у природи спонукають створення та розвитку нових ідей у сфері алгоритмічної оптимізації.

Ройовий інтелект (Swarm intelligence) описує імітацію колективної поведінки системи, що самоорганізується. Існує досить велика кількість

таких алгоритмів. У канонічному варіанті, написаному 1995р. Кеннеді (J.Kennedy) та Еберхартом (R.Eberhart), модель, покладена в основу цього методу, була отримана спрощенням моделі Рейнолдса. В результаті цього спрощення окремі особи популяції стали представлятися окремими об'єктами, що не мають розміру, але мають деяку швидкість.

З огляду на крайньої схожості з матеріальними частками прості об'єкти стали називатися частками, а їх популяція – роєм. У кожний момент часу (на кожній ітерації) частинки мають у просторі деяке положення та вектор швидкості. Для кожного положення частинки обчислюється відповідне значення цільової функції, і на цій основі за певними правилами частка змінює своє положення і швидкість в просторі пошуку, при визначенні наступного положення частки враховується інформація про найкраще становище з усіх інших сусідніх частинок, що відповідає завданням фітнес-функції [14].

Приклади роєвих алгоритмів: метод рою часток, мурашиний алгоритм**,** бджолиний алгоритм, штучна імунна система, алгоритм сірих вовків, алгоритм кажанів, алгоритм гравітаційного пошуку, алгоритм альтруїзму, і багато інших.

1.3.2 Аналіз окремих видів популяційних алгоритмів

1. Метод рою часток.

Класична модель МРЧ була створена лише в 1995 році Расселом Еберхартом і Джеймсом Кеннеді. Їх модель відрізняється тим, що частинки-агенти рою, крім підпорядкування деяким правилам обмінюються інформацією друг з одним, а поточний стан кожної частки характеризується місцем розташування частки у просторі рішень та швидкістю переміщення. Якщо проводити аналогію зі зграєю, то можна сказати, що всі агенти

алгоритму (частки), у зграї вони можуть бути птахами або рибами, ставлять для себе три досить простих правила:

1. Усі агенти повинні уникати перетину з оточуючими їх агентами.
2. Кожна частка повинна коригувати свою швидкість відповідно до швидкостей оточуючих її частинок.
3. Кожен агент повинен намагатися зберігати досить малу відстань між собою та оточуючими його агентами.

Алгоритм рою часток - ітеративний процес, що постійно перебуває в зміні. Щоб зрозуміти, як функціонує алгоритм МРЧ, можна розглянути область пошуку як багатовимірний простір з агентами алгоритму. Спочатку всі агенти знаходяться у випадкових місцях простору та з випадковим вектором швидкості. У кожній з точок, яку частка відвідує, вона розраховує задану функцію і фіксує найкраще значення функції, яку шукає. Так само всі частинки знають місце розташування найкращого результату пошуку у всьому рої і з кожною ітерацією агенти коригують вектори своїх швидкостей та їх напрями, намагаючись наблизитися до найкращої точки рою і при цьому бути ближче до свого індивідуального максимуму. При цьому постійно відбувається розрахунок шуканої функції та пошук найкращого значення [15].

2. Мурашиний алгоритм.

Оптимізаційний алгоритм з наслідуванням колонії мурах – один з найефективніших алгоритмів для вирішення задач з пошуку маршрутів у графах та знаходження приблизних рішень для задачі комівояжера. Суть полягає в тому, що використовується алгоритм поведінки мурах, для вирішення різних задач.

Першим, хто зумів застосувати поведінку мурах для вирішення задачі про найкоротші шляхи, став Марко Доріго на початку 90-х років XX століття.

Після цього було вирішено дуже багато оптимізаційних завдань за допомогою мурашиних алгоритмів. На сьогодняшній день ці алгоритми показують найкращі результати у деяких задачах.

Концепція алгоритму полягає у здатності мурах знаходити найкоротший шлях вкрай швидко та адаптуватися до різних зовнішніх умов. Під час руху кожна мурашка позначає свій шлях феромоном, що надалі використовується іншими мурахами. Це і є простим алгоритмом одного агента, який у сумі всіх агентів колонії дозволяє знаходити найкоротший шлях або змінювати його при виявленні перешкоди [16].

3. Алгоритм бджолиної колонії.

Алгоритм бджолиної колонії – це алгоритм роєвого інтелекту, що ґрунтується на поведінці колонії бджіл, може використовуватись у різних завданнях оптимізації. Умовою для коректного його застосування потрібна наявність топологічної відстані його аналога на області рішень.

Для збору нектару в бджолиній колонії застосовується два види бджіл: бджоли-розвідники та бджоли-робітники. Перші проводять дослідження території, що оточує вулик, щодо наявності нектару. Після повернення у вулик, бджоли-розвідники повідомляють інформацію про кількість нектару, напрям його розташування та відстань до нього. Далі, в найбільш відповідні області вилітають робітники, причому, чим більше нектару в цій області, тим більше бджіл вилітає в неї. Крім збору меду, до їхнього завдання входить оновлення інформації про дану та прилеглі галузі [17].

Колонія використовує алгоритм подібний до видобутку нектару медоносними бджолами. Замість поля із квітами розглянемо область рішень. Замість нектару використовуємо критерії задач оптимізації, цільову функцію. На кожній ітерації алгоритму вибирається n областей з кращим значенням

цільової функції, вони називаються “кращі”, з тих, що залишилися, вибирається ще кращих, званих “перспективними”. Можна встановити певну мінімальну відстань між двома сусідніми областями. У цьому випадку при виникненні накладення, область з гіршим значенням цільової функції відсікається. Замість неї вибирається інша область. Дані області запам'ятовуються і за наступної ітерації до них посилається певна кількість бджіл.

Порівняння цих алгоритмів можна побачити у таблиці 1.

Таблиця 1

	Метод рою часток	**Мурашиний Алгоритм**	**Алгоритм бджолиної колонії**
Переваги	- Вкрай низька алгоритмічна складність у реалізації. - Досить ефективний для глобальної оптимізації.	- Досить ефективний для TSP (Traveling Salesman Problem) із невеликою кількістю вузлів. - Використовує додатки, які можуть адаптуватися до змін. - Завдяки пам'яті всієї колонії та випадковому вибору шляху не так сильно схильний до невдалих початкових рішень.	- Можливість ефективного поділу на паралельні процеси. - Висока швидкість роботи.
Застосування	- Завдання машинного навчання. - Завдання оптимізації функцій багатьох параметрів, форм, розмірів та топологій; - Область проектування - Біоінженерія, біомеханіка, біохімія.	- Розрахунки комп'ютерних та телекомунікаційних мереж. - Завдання комівояжера. - Завдання розмальовки графа. - Завдання оптимізації мережевих трафіків.	- Оптимізація керування. - Оптимізація класифікаторів.
Розвиток	- Подання МРЧ як багатоагентної обчислювальної системи. - Можливості включення інших, складніших методів РІ.	- Гібридизація з генетичними алгоритмами. - Використання бази нечітких правил.	- Зниження залежності від параметрів, що встановлюються. - Поєднання з генетичними алгоритмами.

Були розглянуті основні алгоритми роєвого інтелекту, що використовуються: метод рою часток, мурашиний алгоритм, алгоритм бджолиної колонії. Було проведено їх порівняльний аналіз, в результаті якого були виявлені основні сильні сторони методів, сфери застосування, а також перспективи розвитку кожного з трьох аналізованих.

Як можна побачити з таблиці 1, різні алгоритми потрібні для вирішення різних задач, та для машинного навчання більше всього підходить рій часток.

Так як рій часток дуже гарно підходить до цієї задачі, тому потрібно проаналізувати головні фактори. Вибір відповідного розміру рою є, ймовірно, найважливішим фактором, що визначає продуктивність алгоритму PSO (рій часток). Якщо рій вибрано замалий, алгоритм, швидше за все, буде спрямований до локального мінімуму, не зустрівши глобально гарного рішення. Зі збільшенням розміру рою ймовірність знайти хороше рішення зростає, однак вимоги до обчислень також зростають лінійно з кількістю частинок.

Оптимізація за допомогою рою часток для гіперпараметрів глибоких нейронних мереж є дуже цікавим і багатообіцяючим підходом. Рій часток довів свою здатність ефективно проходити великий простір пошуку рішень і надавати узгоджені, високопродуктивні рішення без попереднього знання, явно перевершуючи людські здібності в цьому відношенні. Тому саме цей алгоритм буде використовуватись для навчання згорткової нейронної мережі.

Висновок за розділом 1

Був проведений аналіз нейронних мереж, їх архітектуру та типи. Після аналізу типів нейронних мереж було виявлено, що згорткові нейронні мережі краще за інших розпізнають образи, так як ця задача дуже актуальна, було обрано саме цей тип нейронної мережі.

Також для коректного виконання поставленого завдання було проведено аналіз методів навчання нейронної мережі та найбільш перспективним виявився менш популярний метод навчання на основі популяційних алгоритмів, так як пошук гіперпараметрів нейронної мережі є дуже перспективним напрямком у вирішенні проблеми застрявання в локальному мінімумі.

На основі аналізу популяційних алгоритмів, їх порівняння для поставленого завдання, було прийняте рішення використовувати рій часток, саме цей алгоритм краще за всіх підходить для машинного навчання.

РОЗДІЛ 2
РОЗРОБКА МОДЕЛІ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ НА ОСНОВІ ПОПУЛЯЦІЙНОГО АЛГОРИТМУ

2.1 Опис вхідних даних

В якості вхідних даних використовуватиметься датасет MNIST (Modified National Institute of Standards and Technology), вибір обумовлений тим, що його часто використовують для навчання нейронної мережі і досить багато досліджень проводилося саме з ним. MNIST dataset – це база даних, де зберігаються зразки написання рукописних цифр.

У датасеті 70 тисяч картинок із цифрами від 0 до 9, наведених до однакового вигляду. Усі вони переведені у формат CSV та мають розмір 28×28 пікселів. У них чорне фон, на якому зображена біла цифра. Цифра поміщена в середині так, щоб її центр мас збігався з центром зображення. Сама вона трохи менша за цілу картинку – її розмір становить 20×20 пікселів [18]. Приклад даних з датасету можна побачити на рисунку (рис. 2.1).

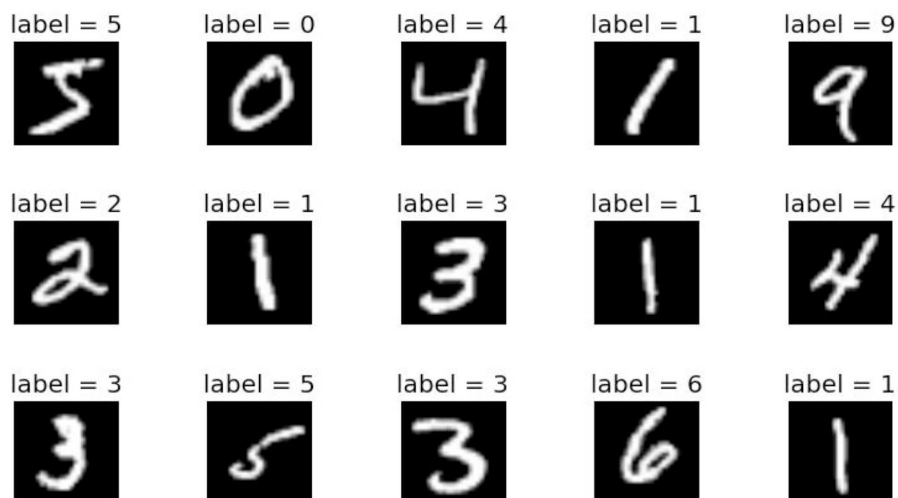


Рисунок 2.1 – Приклад даних з датасету.

2.2 Аналіз структури згорткової нейронної мережі

Згорткові нейронні мережі мають можливість забезпечувати не повну стійкість до змін розміру, зсувів, зміни ракурсу, зміни наклону та інших аномалій. Згорткові нейронні мережі поєднують в собі три архітектурні ідеї, для забезпечення адаптації до змін.

1. Поля локальних рецепторів (забезпечення локального двовимірного зв'язку між нейронами).

2. Синаптичні коефіцієнти (дають змогу детектування деяких рис в різних місцях зображення та зменшують загальну кількість вагових коефіцієнтів).

3. Ієрархічна організація із просторовими підвиборками.

Згортковий тип нейронних мереж та її модифікації вважаються найкращими за точністю та швидкістю для знаходження (розпізнавання) образів на сцені.

У згортковій нейронній мережі в операції згортки використовується лише невелика матриця ваг, яку “рухають” по всьому шару (на самому початку – безпосередньо по вхідному зображенню), формуючи після кожного зсуву сигнал активації для нейрона наступного шару з аналогічною позицією. Тому для різних вихідних нейронів використовується однакова матриця ваг, яку найчастіше називають ядром згортки (фільтр). Тоді наступний шар, був отриманий в результаті операції згортки такою матрицею ваг, показує наявність даної ознаки в шарі, що обробляється, і її координати, формуючи так звану карту ознак. Як це працює можна побачити на рисунку (рис. 2.2).

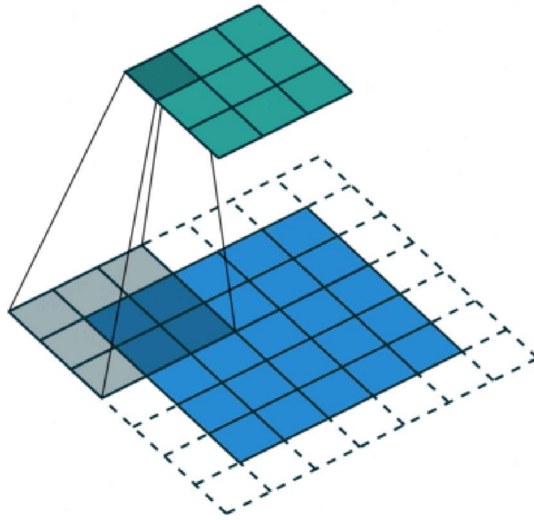


Рисунок 2.2 – Процес конволюції.

У ядра є центральний елемент. Причому центральний елемент може бути необов'язково у центрі. Наприклад, центральний елемент ядра на рисунку вище знаходиться в позиції матриці (1,1), і відносно цього елемента відбувається операція згортки.

Залежно від способу згортки - конволюції або крос-кореляції, різної величини кроку та вибору центрального елемента ядра - розмірність вихідної матриці може змінюватись. Формула для розрахунку розмірності матриці (2.1).

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor \quad (2.1)$$

Найчастіше використовують функцію активації Sigmoid або ReLU. Формула функції активації ReLU (2.2).

$$f_{ReLU} = max(0, x) \tag{2.2}$$

Формула функції активації Sigmoid (2.3)

$$f_{sigmoid} = \frac{1}{1 + e^{-x}} \tag{2.3}$$

У роботі буде використовуватись функція активації ReLU, тому що ReLU не потребує великої кількості обчислювальних ресурсів, ніж сигмоїда, тому що виконує більш прості математичні операції. Тому є сенс використовувати ReLu при створенні глибоких нейронних мереж.

Сігмоїда використовується тільки якщо класів (для задачі класифікації) не більше двох: вихід моделі буде числом від нуля (перший клас) до одиниці (другий клас). Для більшого числа класів, щоб вихід моделі відображав ймовірність цих класів (і сума ймовірностей по виходах мережі дорівнювала одиниці), використовується softmax.

Далі потрібно проаналізувати типи шарів у згортковій нейронній мережі.

Згортковий шар. Згортковий (convolutional) шар дозволяє об’єднувати значення пікселів розташованих поруч та виділяти більш узагальнені ознаки зображення. Приклад коду цього шару можна побачити на рисунку (рис. 2.3).

```
model.add(Conv2D(32,kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
```

Рисунок 2.3 – Приклад згорткового шару.

Основні параметри згорткового шару це filters (кількість фільтрів), а другий kernel_size (розмір ядра кожного фільтра), також є параметр strides (задає крок сканування фільтрів по осях площини, 1 піксель), функція активації, та розмір вхідного зображення (якщо говориться про вхідний шар).

Шар макспулінгу. Цей шар дозволяє виділяти важливі особливості на картах ознак, дає інваріантність до знаходження об'єкта на картах, а також знижує розмір карт, прискорюючи час роботи мережі. Приклад коду цього шару можна побачити на рисунку (рис. 2.4).

```
model.add(MaxPooling2D(pool_size=(sp, sp), strides=(1, 1)))
```

Рисунок 2.4 – Приклад шару макспулінгу.

Код дуже схожий на згортковий шар, причому навіть збереглися ті ж параметри: вибір страйда, та Pool Size (повертає матрицю sp на sp). Але, звичайно, тут не відбувається поелементного перемноження матриць, а лише вибір максимального значення із заданого вікна. “Класичні” значення параметрів макспулінгу у параметрах згортки – це крос-кореляція та позиція центрального елемента у лівому верхньому кутку.

Повнозв’язковий шар. Основним завданням повнозв’язного шару є моделювання складної нелінійної функції, яка найчастіше використовується для класифікації. Ця опція оптимізується в процесі навчання мережі, що дозволяє покращувати якість розпізнавання. Можна вважати, що шари, що йдуть до повнозв’язкового, є засобами попередньої обробки зображення, а

подальша класифікація виконується звичайною мережею прямого поширення. Тобто все, що йде до повного шару, використовується для виділення різних ознак, які потім подаються на вхід класифікатору. Приклад коду цього шару можна побачити на рисунку (рис. 2.5).

```
model.add(Dense(num_classes, activation='softmax'))
```

Рисунок 2.5 – Приклад повнозв'язкового шару.

Функція втрат. Завершальний етап мережі – функція, що оцінює якість роботи всієї моделі. Функція втрат знаходиться в самому кінці після всіх шарів мережі. Відповіддю функції втрат є число, що характеризують якість відповіді нейронної мережі. Зазвичай з метою оцінки якості класифікаторів застосовують категоріальну кросентропійну функцію втрат (2.4).

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i \tag{2.4}$$

Тепер необхідно поєднати згорткову нейронну мережу з роєм часток, але для цього потрібно зрозуміти сам алгоритм рою часток [19].

2.3 Аналіз алгоритму рою часток

Основна ідея алгоритму рою частинок полягає в колективній поведінці деяких агентів, які можуть взаємодіяти один з одним та з навколишнім середовищем для вирішення певного завдання. Завдяки взаємодії частинок виходить успішно вирішувати завдання.

При вирішенні задачі знаходження екстремуму функції кожна частка характеризується трьома параметрами: координатами у просторі та

значенням цільової функції. При цьому на кожному кроці частинки пам'ятають точку, де значення цільової функції було найкращим і намагаються туди повернутися. Кожна частка знає координати кращої точки, в якій була будь-яка інша частка. Також на переміщення кожної впливає випадкове відхилення та швидкість. Схему алгоритму можна побачити на рисунку (рис. 2.6).

Рисунок 2.6 – Схема алгоритму рою часток.

В канонічному алгоритмі положення частки регулює формула (2.5).

$$v_{i+1} = \chi \cdot [v_i a_1 \cdot rnd() \cdot (pbest_i - x_i)\, a_2 \cdot rnd() \cdot (gbest_i - x_i)] \tag{2.5}$$

Насамперед потрібно реалізувати клас рою, який на вхід прийматиме розмір рою, коефіцієнти пріоритетів зміщення частинок до різних точок, кількість ітерацій алгоритму, цільову функцію та область пошуку екстремуму. Приклад коду цього класу можна побачити на рисунку (рис. 2.7).

```
class Pso(object):
    """Обёртка PSO
       Этот класс содержит частицы и предоставляет абстракцию для хранения всего контекста алгоритма PSO.

       Args:
           swarmsize (int): Количество частиц в рое
           maxiter (int): Максимальное количество поколений, которое будет выполняться роем
    """
    def __init__(self, swarmsize=100, maxiter=100):
        self.max_generations = maxiter
        self.swarmsize = swarmsize

        self.omega = 0.5
        self.phip = 0.5
        self.phig = 0.5

        self.minstep = 1e-4
        self.minfunc = 1e-4

        self.best_position = [None]
        self.best_function_value = [1]

        self.particles = []

        self.retired_particles = []
```

Рисунок 2.7 – Клас рою часток.

Далі реалізувати клас для окремої частки, основні параметри якого будуть: вектор нижньої межі кордонів, вектор верхньої межі кордонів, Кількість вимірювань простору пошуку, функція чорної скриньки для оцінки. Приклад коду цього класу можна побачити на рисунку (рис. 2.8).

```python
class Particle(object):
    """Класс частиц для PSO
    Этот класс инкапсулирует поведение каждой частицы в PSO и обеспечивает эффективный сı

    Args:
        lower_bound (np.array): Вектор нижних пределов границ размеров частиц
        upper_bound (np.array): Вектор верхних пределов границ размеров частиц
        dimensions (int): Количество измерений пространства поиска
        objective function (function): Функция черного ящика для оценки
    """
    def __init__(self,
                 lower_bound,
                 upper_bound,
                 dimensions,
                 objective_function):
        self.reset(dimensions, lower_bound, upper_bound, objective_function)
```

Рисунок 2.8 – Клас для окремих часток.

Клас оновлення швидкості часток. Приклад коду можна побачити на рисунку (рис. 2.9).

```python
def update_velocity(self, omega, phip, phig, best_swarm_position):
    """Обновление скорости частиц

      Args:
        omega (float): Константа уравнения скорости
        phip (float): Константа уравнения скорости
        phig (float): Константа уравнения скорости
        best_swarm_position (np.array): Лучшее положение частицы
    """
    random_coefficient_p = np.random.uniform(size=np.asarray(self.position[-1]).shape)
    random_coefficient_g = np.random.uniform(size=np.asarray(self.position[-1]).shape)

    self.velocity.append(omega
                         * np.asarray(self.velocity[-1])
                         + phip
                         * random_coefficient_p
                         * (np.asarray(self.best_position[-1])
                            - np.asarray(self.position[-1]))
                         + phig
                         * random_coefficient_g
                         * (np.asarray(best_swarm_position)
                            - np.asarray(self.position[-1])))

    self.velocity[-1] = self.velocity[-1].astype(int)
```

Рисунок 2.9 – Оновлення швидкості часток.

Клас оновлення положення часток. Приклад коду можна побачити на рисунку (рис. 2.10).

```
def update_position(self, lower_bound, upper_bound, objective_function):
    """Обновление положения частиц

       Args:
          lower_bound (np.array): Вектор нижних пределов границ размеров частиц
          upper_bound (np.array): Вектор верхних пределов границ размеров частиц
          objective function (function): Функция черного ящика для оценки
    """
    new_position = self.position[-1] + self.velocity[-1]

    if np.array_equal(self.position[-1], new_position):
        self.function_value.append(self.function_value[-1])
    else:
        mark1 = new_position < lower_bound
        mark2 = new_position > upper_bound

        new_position[mark1] = lower_bound[mark1]
        new_position[mark2] = upper_bound[mark2]

        self.function_value.append(objective_function(self.position[-1]))

    self.position.append(new_position.tolist())

    if self.function_value[-1] < self.best_function_value[-1]:
        self.best_position.append(self.position[-1][:])
        self.best_function_value.append(self.function_value[-1])
```

Рисунок 2.10 – Оновлення положення часток.

Реалізація цього алгоритму буде спрямована на знаходження гіперпараметрів шару згорткової нейронної мережі [20].

Розроблену модель навчання згорткової нейронної мережі на основі популяційного алгоритму (рою часток) можна представити у вигляді блок-схеми (рис.2.11).

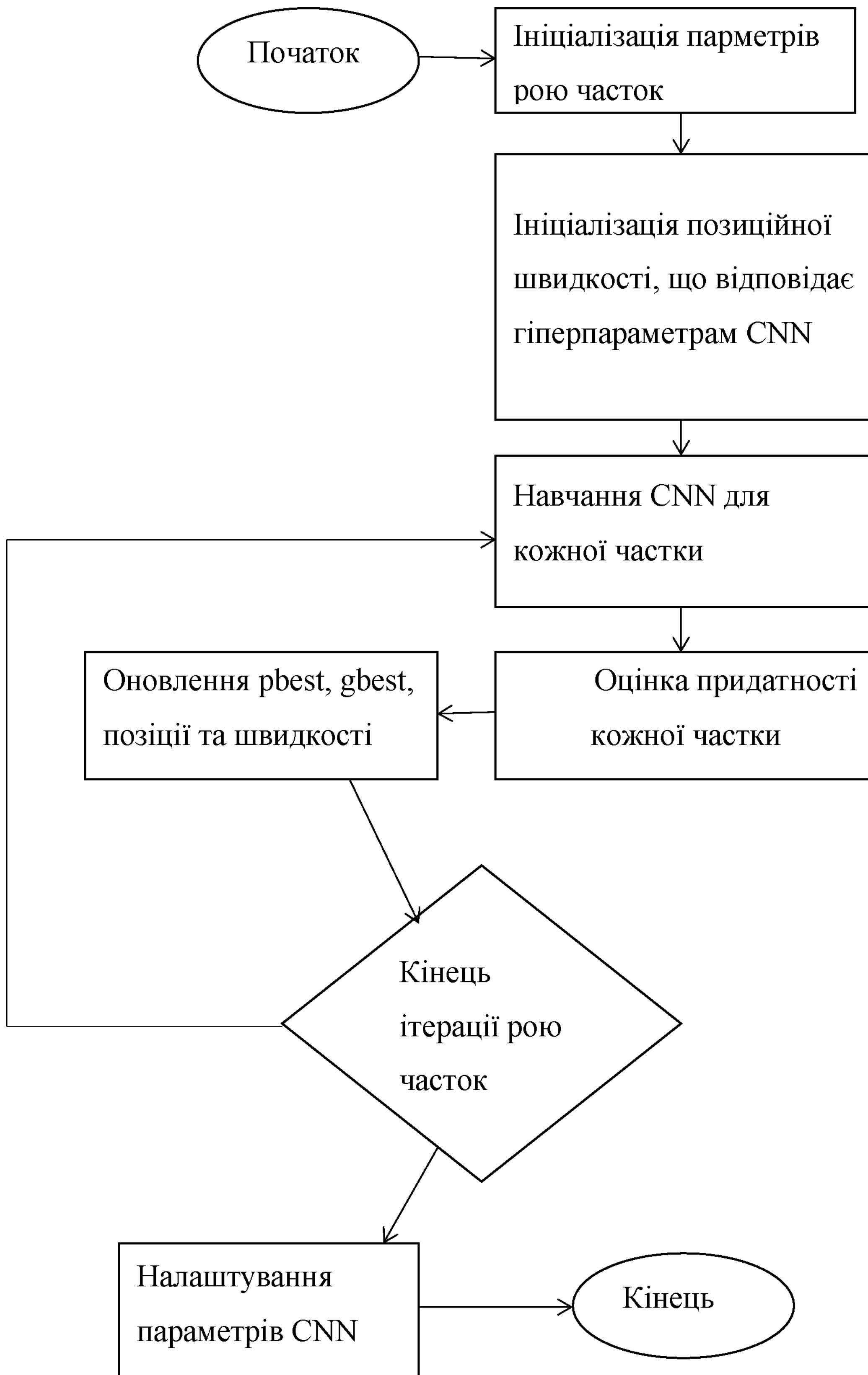


Рисунок 2.11 – Схема виконання етапів алгоритму.

Висновок за розділом 2

В цьому розділі було проведено опис вхідних даних для тестування моделі.

Був проведений аналіз згорткової нейронної мережі, її архітектури, типів шарів, функції активації, розглянуто процес навчання згорткової нейронної мережі.

А також розроблена модель згорткової нейронної мережі на основі популяційного алгоритму (рою часток) для розпізнавання образів. Проводився аналіз алгоритму рою часток, який буде приймати участь у знаходженні гіперпараметрів нейронної мережі та буде вирішувати проблеми навчання звичайним способом, опис якого був у розділі 1.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ МОДЕЛІ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ НА ОСНОВІ ПОПУЛЯЦІЙНОГО АЛГОРИТМУ

3.1 Інструментальні засоби реалізації моделі

Для вирішення поставленого завдання використовувалися наступні технології та засоби.

1. Мова програмування Python 3.7.
2. Нейромережева бібліотека Keras 2.2.4. Backend бібліотеки Keras представлений нейромережевою бібліотекою TensorFlow 1.13.1;
3. Робоче середовище розробки Google Colabaratory.

Python – це мова комп’ютерного програмування, яка часто використовується для створення веб-сайтів і програмного забезпечення, автоматизації завдань і аналізу даних. Python є мовою загального призначення, тобто її можна використовувати для створення різноманітних програм і не спеціалізується на конкретних проблемах. Ця універсальність, а також зручність для початківців зробили її однією з найбільш використовуваних мов програмування сьогодні. Опитування, проведене галузевою аналітичною компанією RedMonk, показало, що це була друга за популярністю мова програмування серед розробників у 2021 році [22].

Keras – це високорівневий API глибокого навчання, розроблений Google для впровадження нейронних мереж. Він написаний на Python і використовується для полегшення впровадження нейронних мереж. Він також підтримує численні серверні обчислення нейронної мережі.

Keras відносно легко освоїти та працювати з ним, тому що він забезпечує інтерфейс Python з високим рівнем абстракції, маючи можливість використовувати кілька серверних інтерфейсів для обчислень. Це робить Keras повільнішим, ніж інші фреймворки глибокого навчання, але надзвичайно зручним для початківців.

Keras дозволяє перемикатися між різними серверами. Keras підтримує такі фреймворки: Tensorflow, Теано, PlaidML, MXNet, CNTK (Microsoft Cognitive Toolkit).

З цих п’яти фреймворків TensorFlow прийняв Keras як офіційний API високого рівня. Keras вбудовано в TensorFlow і може використовуватися для швидкого глибокого навчання, оскільки він надає вбудовані модулі для всіх обчислень нейронної мережі. У той же час, обчислення з використанням тензорів, обчислювальних графіків, сеансів, можна створювати на замовлення за допомогою Tensorflow Core API, який дає повну гнучкість і контроль над програмою та дозволяє реалізувати ідеї за відносно короткий час [23].

Google Collaboratory – це хмарний сервіс, який спрямований на спрощення дослідження в галузі машинного та глибокого навчання. Colaboratory дозволяє абсолютно безкоштовно віддалено отримати доступ до машини з підключеною графічною картою, що значно полегшує роботу, коли потрібно навчити глибокі нейронні мережі. Можна сказати, що він є аналогом Google Docs для Jupyter Notebook.

Tensorflow і майже всі необхідні бібліотеки Python встановлені в Collaboratory. Якщо пакет відсутній, його легко встановити за допомогою pip або apt-get [24].

3.2 Програмна реалізація моделі згорткової нейронної мережі на основі популяційного алгоритму

Для реалізації моделі згорткової нейронної мережі для розпізнавання образів на основі популяційного алгоритму потрібно спочатку розробити структуру згорткової нейронної мережі. Структуру можна побачити на рисунку (рис. 3.1).

```
def func(x):
  n, sf, sp, l = x[0], x[1], x[2], x[3]

  model = Sequential()
  model.add(Conv2D(32,kernel_size=(3, 3),
                   activation='relu',
                   input_shape=input_shape))
  model.add(Conv2D(64, (3, 3), activation='relu'))
  model.add(Conv2D(n, (sf, sf), activation='relu'))
  model.add(MaxPooling2D(pool_size=(sp, sp), strides=(1, 1)))
  model.add(Flatten())
  model.add(Dense(num_classes, activation='softmax'))
  model.compile(loss=keras.losses.categorical_crossentropy,
                optimizer=keras.optimizers.Adam(),
                metrics=['accuracy'])

  cp = [keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, verbose=0, mode='auto')]

  model.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            verbose=0,
            validation_data=(x_test, y_test),
            callbacks=cp)

  score = model.evaluate(x_test, y_test, verbose=0)

  # loss, val
  print('current config: ', x, ' val: ', score[1])
  return score[1]
```

Рисунок 3.1 – структура згорткової нейронної мережі.

Як можна побачити рій часток буде оптимізувати два шари нейронної мережі, а саме шар Conv2D – це шар згортки та шар макспулінгу, для отримання максимальних результатів, рій часток буде знаходити 4 параметри, які будуть приймати участь у структурі шарів.

Сама структура нейронної мережі містить 6 шарів, 3 шари згортки, 1 шар макспулінгу, 1 шар флаттен, та повнозв’язний шар.

Після того, як всі шари були додані необхідно скомпілювати модель, це можна зробити наступним чином (рис. 3.2).

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Рисунок 3.2 – Компіляція розробленої моделі.

На рисунку, який зображений вище, можна побачити функцію втрат (categorical_crossentropy), функцію оптимізації (adam) та метрику (accuracy).

Функція втрат – це функція яка визначає наскільки близько передбачений розподіл до істинного.

Оптимізатор Adam – це один з найбільш ефективних алгоритмів оптимізації в навчанні нейронних мереж. Він поєднує в собі ідеї RMSProp і оптимізатора імпульсу. Замість того щоб адаптувати швидкість навчання параметрів на основі середнього першого моменту (середнього значення), як в RMSProp, Adam також використовує середнє значення других моментів градієнтів.

Далі необхідно завантажити початкові дані та привести їх у коректний вид, рисунку (рис. 3.3).

```
batch_size = 128
num_classes = 10
epochs = 100

# размеры входного изображения
img_rows, img_cols = 28, 28

# данные mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'записей для тренировки')
print(x_test.shape[0], 'записей для теста')
```

```
x_train shape: (60000, 28, 28, 1)
60000 записей для тренировки
10000 записей для теста
```

Рисунок 3.3 – Завантаження та препроцессінг вхідних даних.

Далі потрібен сам оптимізатор – рій часток. Код класу часток можна побачити на рисунку (рис. 3.4).

```
class Particle(object):
    """Класс частиц для PSO
    Этот класс инкапсулирует поведение каждой частицы в PSO и обеспечивает эффективный способ ведения учета состояния роя в любой заданной итерации.

    Args:
        lower_bound (np.array): Вектор нижних пределов границ размеров частиц
        upper_bound (np.array): Вектор верхних пределов границ размеров частиц
        dimensions (int): Количество измерений пространства поиска
        objective function (function): Функция черного ящика для оценки
    """
    def __init__(self,
                 lower_bound,
                 upper_bound,
                 dimensions,
                 objective_function):
        self.reset(dimensions, lower_bound, upper_bound, objective_function)

    def reset(self,
              dimensions,
              lower_bound,
              upper_bound,
              objective_function):
        """Сброс частиц
        Позволяет сбросить частицу без перераспределения

        Args:
          lower_bound (np.array): Вектор нижних пределов границ размеров частиц
          upper_bound (np.array): Вектор верхних пределов границ размеров частиц
          dimensions (int): Количество измерений пространства поиска
        """
        position = []
        for i in range(dimensions):
            if lower_bound[i] < upper_bound[i]:
                position.extend(np.random.randint(lower_bound[i], upper_bound[i] + 1, 1, dtype=int))
            elif lower_bound[i] == upper_bound[i]:
                position.extend(np.array([lower_bound[i]], dtype=int))
            else:
                assert False

        self.position = [position]

        self.velocity = [np.multiply(np.random.rand(dimensions),
```

Рисунок 3.4 – Клас частинок рою часток.

В цьому коді написані основні параметри часток, а також функція ресету часток без перерозподілу.

Наступним етапом було написання функції оновлення швидкості часто та функції оновлення позиції часток, код можна побачити на рисунку (рис. 3.5 та 3.6).

```python
def update_velocity(self, omega, phip, phig, best_swarm_position):
    """Обновление скорости частиц

      Args:
        omega (float): Константа уравнения скорости
        phip (float): Константа уравнения скорости
        phig (float): Константа уравнения скорости
        best_swarm_position (np.array): Лучшее положение частицы
    """
    random_coefficient_p = np.random.uniform(size=np.asarray(self.position[-1]).shape)
    random_coefficient_g = np.random.uniform(size=np.asarray(self.position[-1]).shape)

    self.velocity.append(omega
                         * np.asarray(self.velocity[-1])
                         + phip
                         * random_coefficient_p
                         * (np.asarray(self.best_position[-1])
                            - np.asarray(self.position[-1]))
                         + phig
                         * random_coefficient_g
                         * (np.asarray(best_swarm_position)
                            - np.asarray(self.position[-1])))

    self.velocity[-1] = self.velocity[-1].astype(int)
```

Рисунок 3.5 – Функція оновлення швидкості часток.

```python
def update_position(self, lower_bound, upper_bound, objective_function):
    """Обновление положения частиц

      Args:
        lower_bound (np.array): Вектор нижних пределов границ размеров частиц
        upper_bound (np.array): Вектор верхних пределов границ размеров частиц
        objective function (function): Функция черного ящика для оценки
    """
    new_position = self.position[-1] + self.velocity[-1]

    if np.array_equal(self.position[-1], new_position):
        self.function_value.append(self.function_value[-1])
    else:
        mark1 = new_position < lower_bound
        mark2 = new_position > upper_bound

        new_position[mark1] = lower_bound[mark1]
        new_position[mark2] = upper_bound[mark2]

        self.function_value.append(objective_function(self.position[-1]))

    self.position.append(new_position.tolist())

    if self.function_value[-1] < self.best_function_value[-1]:
        self.best_position.append(self.position[-1][:])
        self.best_function_value.append(self.function_value[-1])
```

Рисунок 3.6 – Функція оновлення позиції часток.

Останні кроки для виконання алгоритму це написання основного класу рою часток, який зберігає основні константи для виконання, ініціалізація часток для рою та сам алгоритм оптимізації.

Ініціалізацію часток для рою можна побачити на рисунку (рис. 3.7).

```
def initialize_particles(self,
                         lower_bound,
                         upper_bound,
                         dimensions,
                         objective_function):
    """Инициализация частиц для роя

        Args:
            objective_function (function): функция, которую нужно минимизировать
            lower_bound (np.array): Вектор нижних пределов границ размеров частиц
            upper_bound (np.array): Вектор верхних пределов границ размеров частиц
            dimensions (int): Количество измерений пространства поиска
        Returns:
            particles (list): частицы роя
    """
    particles = []
    for _ in range(self.swarmsize):
        particles.append(Particle(lower_bound,
                                  upper_bound,
                                  dimensions,
                                  objective_function))
        if particles[-1].best_function_value[-1] < self.best_function_value[-1]:
            self.best_function_value.append(particles[-1].best_function_value[-1])
            self.best_position.append(particles[-1].best_position[-1])


    self.best_position = [self.best_position[-1]]
    self.best_function_value = [self.best_function_value[-1]]

    return particles
```

Рисунок 3.7 – Код ініціалізації часток для рою.

Основний клас рою часток з константами можна побачити на рисунку (рис. 3.8).

```
class Pso(object):
    """Обёртка PSO
       Этот класс содержит частицы и предоставляет абстракцию для хранения всего контекста алгоритма PSO.

       Args:
           swarmsize (int): Количество частиц в рое
           maxiter (int): Максимальное количество поколений, которое будет выполняться роем
    """
    def __init__(self, swarmsize=100, maxiter=100):
        self.max_generations = maxiter
        self.swarmsize = swarmsize

        self.omega = 0.5
        self.phip = 0.5
        self.phig = 0.5

        self.minstep = 1e-4
        self.minfunc = 1e-4

        self.best_position = [None]
        self.best_function_value = [1]

        self.particles = []

        self.retired_particles = []
```

Рисунок 3.8 – Основний клас рою часток.

Сам алгоритм оптимізації гіперпараметрів за допомогою рою часток можна побачити на рисунку (рис. 3.9).

```
generation = 1
while generation <= self.max_generations:
    for particle in self.particles:
        particle.update_velocity(self.omega, self.phip, self.phig, self.best_position[-1])
        particle.update_position(lower_bound, upper_bound, objective_function)

        if particle.best_function_value[-1] == 0:
            self.retired_particles.append(copy.deepcopy(particle))
            particle.reset(dimensions, lower_bound, upper_bound, objective_function)
        elif particle.best_function_value[-1] < self.best_function_value[-1]:
            stepsize = np.sqrt(np.sum((np.asarray(self.best_position[-1])
                                       - np.asarray(particle.position[-1])) ** 2))

            if np.abs(np.asarray(self.best_function_value[-1])
                      - np.asarray(particle.best_function_value[-1])) \
                    <= self.minfunc:
                return particle.best_position[-1], particle.best_function_value[-1]
            elif stepsize <= self.minstep:
                return particle.best_position[-1], particle.best_function_value[-1]
            else:
                self.best_function_value.append(particle.best_function_value[-1])
                self.best_position.append(particle.best_position[-1][:])

    generation += 1

return self.best_position[-1], self.best_function_value[-1]
```

Рисунок 3.9 – Алгоритм оптимізації.

Далі необхідно натренувати нашу нейронну мережу на основі тренувальної вибірки та перевірити її працездатність на основі тестових даних, а також задати кількість епох (рис. 3.10).

```
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=0,
          validation_data=(x_test, y_test),
          callbacks=cp)

score = model.evaluate(x_test, y_test, verbose=0)

# loss, val
print('current config: ', x, ' val: ', score[1])
return score[1]
```

Рисунок 3.10 – Код виконання програмної моделі.

Результат виконання коду нейронної мережі можна побачити на рисунку (рис. 3.11).

```
current config:  [14, 7, 2, 4]  val:  0.9908000230789185
current config:  [13, 8, 3, 3]  val:  0.9901000261306763
current config:  [16, 2, 2, 2]  val:  0.9879000186920166
current config:  [5, 3, 3, 3]  val:  0.9872999787330627
current config:  [14, 7, 2, 4]  val:  0.9886999726295471
current config:  [16, 2, 2, 2]  val:  0.9882000088691711
current config:  [5, 3, 3, 3]  val:  0.9860000014305115
current config:  [14, 4, 2, 2]  val:  0.9885000058174133
current config:  [12, 5, 3, 3]  val:  0.9891999959945679
current config:  [13, 8, 3, 3]  val:  0.9891999959945679
current config:  [13, 4, 2, 2]  val:  0.9894999861717224
current config:  [15, 6, 3, 3]  val:  0.9896000027656555
current config:  [16, 7, 2, 4]  val:  0.9891999959945679
current config:  [13, 6, 3, 3]  val:  0.9901000261306763
current config:  [15, 7, 2, 4]  val:  0.9900000095367432
current config:  [12, 5, 3, 3]  val:  0.9911000013256073
```

```
print('Test loss: ', bp)
print('Test accuracy: ', value, v)
```

```
Test loss:  [12, 5, 3, 3]
Test accuracy:  0.9860000014305115 0.9911000013256073
```

Рисунок 3.11 – Результат виконання моделі.

Після розробки моделі було прийняте рішення написати модель згорткової нейронної мережі на основі генетичного алгоритму. Це

обумовлено тим, що навчання за допомогою генетичного алгоритму такої моделі має майже максимальну точність розпізнавання датасету MNIST, тому буде достатньо цікаво подивитися порівняння цих двох моделей.

Генетичний алгоритм формує структуру згорткової нейронної мережі, для отримання максимальних результатів.

Основні функції цього алгоритму:

1. Генерація потомства поточної популяції, рисунку (рис. 3.12).

```
def generate_offsprings(self) -> List[CNN]:
    """
    Генерирует потомство текущей популяции.
    Шаги:
    1. В то время как популяция потомства меньше, чем текущая численность популяции
        1. случайным образом выберем 2-х разных особей из текущей популяции и оставим ту, у которой наибольшая приспособленность
        2. сделаем то же самое, что и на шаг 1, убедимся, что они разные
        3. выберем случайное число между (0, 1)
        4. Если число меньше, чем crossover_probability
            1. случайным образом разделим две родительские CNN
            2. создадим два новых потомка, поменяв местами два родительских слоя
            3. добавить их к потомству популяции
        5. В противном случаем
            1. добавим двух родителей к потомкам
    2. Пройдём через мутации
        1. За каждое сгенерированное потомство
            1. выберем случайное число между (0, 1)
            2. Если число меньше, чем mutation_probability
                1. случайно выберем одну из 4 возможных мутаций, учитывая mutation_operation_distribution distribution
                2. add_skip: добавляет слой со случайным пропуском, увеличивает сложность и глубину сети
                3. add_pooling: добавляет случайный слой пула, увеличивает глубину, но снижает сложность
                4. remove: удаляет слой, уменьшает сложность и глубину
                5. change: изменяет параметры слоя (например, размер фильтра, максимальное или среднее объединение)
    :return: потомков нынешнего поколения
    """
    offsprings = []

    while len(offsprings) < len(self.population):
        p1 = self.select_two_individuals(self.population)
        p2 = self.select_two_individuals(self.population)

        while p1.hash == p2.hash:
            p2 = self.select_two_individuals(self.population)

        r = random.random()
```

Рисунок 3.12 – Генерація потомства.

2. Оцінка придатності згорткової нейронної мережі, рисунок (рис. 3.13).

```python
def evaluate_individual_fitness(self, cnn: CNN) -> None:
    """
    Оценивает пригодность CNN. Пригодность основана на точности CNN на тестовых данных.
    Не выполняет повторную оценку пригодности, поскольку она уже находится в кэше пригодности.
    После расчета фитнеса он добавляется в кэш фитнеса и сохраняется в кэше json.
    :param cnn: CNN, чтобы найти пригодность
    """
    try:
        cnn.generate()

        cnn.train(self.dataset, epochs=self.epoch_number)
        loss, accuracy = cnn.evaluate(self.dataset)
    except ValueError as e:
        print(e)
        accuracy = 0

    self.fitness[cnn.hash] = accuracy

    if self.fitness_cache is not None:
        with open(self.fitness_cache, 'w') as json_file:
            json.dump(self.fitness, json_file)
```

Рисунок 3.13 – Функція оцінки придатності.

3. Алгоритм початку роботи моделі, прохід через генетичний алгоритм та формування структури згорткової нейронної мережі, рисунок (рис. 3.14).

```python
def run(self) -> CNN:
    """
    Проходит через весь метод генетического алгоритма
    Шаги:
    1. инициализирует популяцию
    2. Для maximal_generation_number генераций:
        1. оценим приспособленность населения
        2. произведем потомство
        3. оценим приспособленность потомства
        4. выберем новую популяцию из потомков и текущей популяции
    :return: лучшая CNN, найденная после всех поколений
    """
    print("Initializing Population")
    self.initialize()
    print("Population Initialization Done:", self.population)

    for i in range(self.maximal_generation_number):
        print("Generation", i)

        print("Evaluating Population fitness")
        self.evaluate_fitness(self.population)
        print("Evaluating Population fitness Done:", self.fitness)

        print("Generating Offsprings")
        offsprings = self.generate_offsprings()
        print("Generating Offsprings Done:", offsprings)

        print("Evaluating Offsprings")
        self.evaluate_fitness(offsprings)
        print("Evaluating Offsprings Done:", self.fitness)

        print("Selecting new environment")
        new_population = self.environmental_selection(offsprings)
        print("Selecting new environment Done:", new_population)

        self.population = new_population

    best_cnn = sorted(self.population, key=lambda x: self.fitness[x.hash])[-1]
    print("Best CNN:", best_cnn, "Score:", self.fitness[best_cnn.hash])
```

Рисунок 3.15 - Алгоритм початку роботи програмної моделі.

Були показані основні розроблені функції моделі розпізнавання образів згорткової нейронної мережі на основі генетичного алгоритму. Приклади такого програмного продукту можна знайти у різних дослідженнях оптимізації згорткової нейронної мережі.

3.3 Дослідження моделі згорткової нейронної мережі на основі популяційного алгоритму

Етапи проведення дослідження:

1. Завантаження початкових даних у модель на основі генетичного алгоритму.
2. Завантаження початкових даних у модель на основі популяційного алгоритму.
3. Навчання цих моделей.
4. Тестування моделей.
5. Порівняння швидкості навчання та якості розпізнавання.

Для початку потрібно завантажити дані, 60 000 для навчання та 10 000 для тесту. Потім виконати весь програмний код, хмарна платформа Google Colaboratory дозволяє зробити це не навантажуючи власний ПК та використовувати хмарні обчислення.

Після виконання коду було виявлено, що структура і метод навчання на основі рою часток має перевагу над моделлю з генетичним алгоритмом, але був знайдений один мінус – це дуже довгий час навчання, він може сягати від 1 до 6 годин. Модель на основі рою часток навчилася на 1 годину раніше.

Результати виконання програмної моделі (ПМ) на основі генетичного алгоритму, рисунок (рис. 3.16).

```
Epoch 1/5
750/750 [==============================] - 41s 52ms/step - loss: 0.1910 - accuracy: 0.9528 - val_loss: 0.0479 - val_accuracy: 0.9858
Epoch 2/5
750/750 [==============================] - 39s 51ms/step - loss: 0.0410 - accuracy: 0.9869 - val_loss: 0.0499 - val_accuracy: 0.9843
Epoch 3/5
750/750 [==============================] - 39s 52ms/step - loss: 0.0295 - accuracy: 0.9908 - val_loss: 0.0334 - val_accuracy: 0.9901
Epoch 4/5
750/750 [==============================] - 39s 52ms/step - loss: 0.0230 - accuracy: 0.9925 - val_loss: 0.0345 - val_accuracy: 0.9892
Epoch 5/5
750/750 [==============================] - 36s 48ms/step - loss: 0.0185 - accuracy: 0.9941 - val_loss: 0.0390 - val_accuracy: 0.9896
157/157 [==============================] - 2s 16ms/step - loss: 0.0345 - accuracy: 0.9906
256-64-32-64-max-256-512-32-64-max-128-32 0.9905999898910522
Evaluating Offsprings Done: {'128-64': 0.9801999926567078, '32-512-256-32': 0.11349999904632568, '64-512': 0.9787999987602234, '64-512-256-512-32-6
Selecting new environment
Best CNN: 256-64-32-64-max-256-512-32-64-max-128-32 Score: 0.9905999898910522
Selecting new environment Done: [256-64-32-64-max-256-512-32-64-max-128-32, 64-512-256-512-32-64-max-128-32, 64-512-256-512-32-64-max-128-32, 64-51
Best CNN: 256-64-32-64-max-256-512-32-64-max-128-32 Score: 0.9905999898910522
256-64-32-64-max-256-512-32-64-max-128-32
```

Рисунок 3.16 – Результати виконання.

Як можна побачити, генетичний алгоритм знайшов краще рішення архітектури та точність розпізнавання є 0.9905.

Результати виконання ПА на основі рою часток, рисунок (рис. 3.17).

```
current config:  [13, 4, 2, 2]  val:  0.9894999861717224
current config:  [15, 6, 3, 3]  val:  0.9896000027656555
current config:  [16, 7, 2, 4]  val:  0.9891999959945679
current config:  [13, 6, 3, 3]  val:  0.9901000261306763
current config:  [15, 7, 2, 4]  val:  0.9900000095367432
current config:  [12, 5, 3, 3]  val:  0.991100013256073
```

```
[ ]  print('Test loss: ', bp)
     print('Test accuracy: ', value, v)
```

```
Test loss:  [12, 5, 3, 3]
Test accuracy:  0.9860000014305115 0.991100013256073
```

Рисунок 3.17 – Результати виконання.

На цьому рисунок можна побачити якість розпізнавання в 0.9911, тобто навчання (оптимізація) за допомогою рою часток має перевагу над генетичним методом навчання. Багато досліджень підтверджує, що саме такий метод навчання (за допомогою популяційних алгоритмів) у майбутньому буде мати більший спектр використання, тому що має більш кращі показники у використанні зі згортковою нейронною мережею. Проблема довгого навчання буде вирішена більшою кількістю обчислюваних потужностей або розпаралелюванням алгоритму.

Висновок за розділом 3

Був проведений аналіз програмних засобів для реалізації моделі, вибрана мова програмування Python і середовище розробки Google Colaboratory, такий вибір обумовлений простим інтерфейсом середовища розробки, безліччю вбудованих функцій, які допомагають працювати з нейронними мережами.

Була розроблена програмна модель згорткової нейронної мережі на основі популяційного та генетичного алгоритмів, описані основні функції та був проведений тест працездатності цих моделей. Було виявлено, що модель на основі популяційного алгоритму має перевагу над моделлю з генетичним алгоритмом на основі дослідження якості розпізнавання та швидкості навчання.

ВИСНОВОК

В ході виконання кваліфікаційної роботи була розроблена програмна модель згорткової нейронної мережі на основі популяційного алгоритму для розпізнавання образів, модель навчається на 60 000 образах та тестується на 10 000 образах, в ході виконання модель показує точність розпізнавання.

У першому розділі була розглянута структура нейронної мережі. Було проаналізовано основні типи архітектури нейронних мереж, аналіз методів навчання нейронної мережі та аналіз популяційних алгоритмів. На основі аналізу нейронних мереж було обрано згорткову нейронну мережу, був обраний метод навчання за допомогою популяційного алгоритму, а також був обраний рій часток в якості популяційного алгоритму, так як дослідження показують, що за допомогою нього можна значно покращити показники моделі з нейронною мережею.

У другому розділі були описані основні параметри мережі.

Був проведений аналіз згорткової нейронної мережі, її архітектури, роботи шарів, функції активації та розглянуто процес навчання згорткової нейронної мережі на основі популяційного алгоритму, було виявлено, що оптимізація роєм часток має великі перспективи в оптимізації нейронної мережі.

У третьому розділі був проведений аналіз програмних засобів для реалізації моделі, вибрана мова програмування Python і середовище розробки Google Colaboratory, такий вибір обумовлений простим інтерфейсом середовища розробки, безліччю вбудованих функцій, які допомагають працювати з нейронними мережами.

Була розроблена програмна модель згорткової нейронної мережі на основі популяційного алгоритму для розпізнавання образів, описані основні

функції та був проведений тест працездатності моделі. Також було проведено порівняння з іншою моделлю на основі генетичного алгоритму, результат показав перевагу моделі на основі популяційного алгоритму.

Дану модель та результати, отримані в ході розробки та тестування, можуть використовувати для дослідження на більш складних датасетах та в подальшому використовувати в багатьох галузях, наприклад в камерах фіксування автомобільних номерів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Some basic principles behind neural network [Електронний ресурс]. Режим доступу: https://www.navixy.com/blog/some-basic-principles-behind-neural-networks/ (Дата звернення – 01.09.2022)
2. Neural Network Principles [Електронний ресурс]. Режим доступу: https://www.webology.org/data-cms/articles/20220315050405pmWEB19261.pdf (Дата звернення – 03.09.2022)
3. The mostly complete chart of Neural Networks, explained [Електронний ресурс]. Режим доступу: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464 (Дата звернення – 05.09.2022)
4. Multi-Layer Neural Network [Електронний ресурс]. Режим доступу: http://deeplearning.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/ (Дата звернення – 07.09.2022)
5. Architecture and fundamentals of Radial Basis neural network [Електронний ресурс]. Режим доступу: https://studbooks.net/1160955/informatika/seti_radialnyh_bazisnyh_funktsiy/architecture_and_fundamentals/neural_networks_of_radial_basis_functions (Дата звернення – 09.09.2022)
6. Recurrent Neural Networks [Електронний ресурс]. Режим доступу: https://www.ibm.com/cloud/learn/recurrent-neural-networks (Дата звернення – 11.09.2022)

7. Beginners Guide to Boltzmann Machine [Електронний ресурс]. Режим доступу: https://analyticsindiamag.com/beginners-guide-to-boltzmann-machines/ (Дата звернення – 14.09.2022)
8. An Overview of Deep Belief Network (DBN) in Deep Learning [Електронний ресурс]. Режим доступу: https://www.analyticsvidhya.com/blog/2022/03/an-overview-of-deep-belief-network-dbn-in-deep-learning/ (Дата звернення – 19.09.2022)
9. CNN network [Електронний ресурс]. Режим доступу: https://www.ibm.com/cloud/learn/convolutional-neural-networks (Дата звернення – 23.09.2022)
10. Back Propagation in Neural Network: Machine Learning Algorithm [Електронний ресурс]. Режим доступу: https://www.guru99.com/backpropogation-neural-network.html (Дата звернення – 26.09.2022)
11. How To Use Resilient Back Propagation To Train Neural Networks [Електронний ресурс]. Режим доступу: https://visualstudiomagazine.com/articles/2015/03/01/resilient-back-propagation.aspx (Дата звернення – 29.09.2022)
12. Using Genetic Algorithms to Train Neural Networks [Електронний ресурс]. Режим доступу: https://towardsdatascience.com/using-genetic-algorithms-to-train-neural-networks-b5ffe0d51321 (Дата звернення – 30.09.2022)
13. Training Neural Networks with PSO [Електронний ресурс]. Режим доступу: https://vortarus.com/training-neural-networks-with-pso/ (Дата звернення – 01.10.2022)

14. Introduction to Swarm Intelligence [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/introduction-to-swarm-intelligence/ (Дата звернення – 05.10.2022)
15. A Gentle Introduction to Particle Swarm Optimization [Електронний ресурс]. Режим доступу: https://machinelearningmastery.com/a-gentle-introduction-to-particle-swarm-optimization/ (Дата звернення – 10.10.2022)
16. Ant colony optimization [Електронний ресурс]. Режим доступу: http://www.scholarpedia.org/article/Ant_colony_optimization (Дата звернення – 14.10.2022)
17. Artificial bee colony algorithm [Електронний ресурс]. Режим доступу: http://www.scholarpedia.org/article/Artificial_bee_colony_algorithm (Дата звернення – 18.10.2022)
18. MNIST dataset [Електронний ресурс]. Режим доступу: https://www.tensorflow.org/datasets/catalog/mnist (Дата звернення – 25.10.2022)
19. Python CNN [Електронний ресурс]. Режим доступу: https://www.datacamp.com/tutorial/convolutional-neural-networks-python (Дата звернення – 28.10.2022)
20. Hyper-parameter optimization of convolutional neural network based on particle swarm optimization algorithm [Електронний ресурс]. Режим доступу: https://beei.org/index.php/EEI/article/view/3257 (Дата звернення – 10.11.2022)
21. Python [Електронний ресурс]. Режим доступу: https://www.python.org/doc/essays/blurb/ (Дата звернення – 15.11.2022)
22. Keras [Електронний ресурс]. Режим доступу: https://keras.io/about/ (Дата звернення – 19.11.2022)

23.Google Colaboratory [Електронний ресурс]. Режим доступу: https://research.google.com/colaboratory/faq.html (Дата звернення – 23.11.2022)

ДОДАТОК А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет комп'ютерних наук
Кафедра теоретичної та прикладної системотехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) Магістр
галузь знань 15 – Автоматизація та приладобудування
спеціальність 151 – Автоматизація та комп'ютерно-інтегровані технології

ЗАТВЕРДЖУЮ
Завідувач кафедри теоретичної
та прикладної системотехніки

______________**д.т.н., проф. Шматков С. І.**

«_10_» ___12___ 2021 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Безсмертний Денис Русланович
(прізвище, ім'я, по батькові студента)

1. Тема роботи « Методи навчання згорткової нейронної мережі на основі популяційного алгоритму »____________________________

керівник роботи Шматков Сергій Ігоревич, професор, д. т. н.

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "___" ___________20__року
№_____________

2.Строк подання студентом роботи __30 листопада 2022________________
3. Перелік питань, які потрібно розробити
1. Аналіз методів навчання згорткової нейронної мережі.____________________________

2. Розробка програмної моделі згорткової нейронної мережі на основі популяційного алгоритму.
3. Оцінка працездатності та порівняння з іншою моделлю для отримання висновків щодо якості виконання.

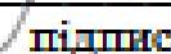
4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Проведення аналізу літературних джерел по методам навчання згорткової нейронної мережі	10.12.2021 – 12.01.2022
2	Розробка алгоритму навчання згорткової нейронної мережі на основі популяційного алгоритму	13.01.2022 – 31.01.2022
3	Програмна реалізація моделі згорткової нейронної мережі на основі популяційного алгоритму	01.02.2022 – 27.02.2022
4	Проведення тесту працездатності	28.02.2022 – 29.03.2022
5	Оцінка якості роботи моделі та порівняння з іншою	30.03.2022 – 30.04.2022
6	Розробка висновків та практичних рекомендацій	01.05.2022 –

		31.05.2022
7	Написання пояснювальної записки	01.06.2022 – 30.08.2022
8	Представлення роботи науковому керівнику	31.08.2022 – 01.09.2022
9	Написання звіту з науково-дослідної практики	01.09.2022 – 12.10.2022
10	Написання звіту з преддипломної практики	13.10.2022 – 20.11.2022

5. Дата видачі завдання ___________10.12.2021__________________________

Студент ____Д. Р. Безсмертний ____________________
ініціали, прізвище підпис

Керівник роботи __С. І. Шматков ____________________
ініціали, прізвище підпис

ДОДАТОК Б

Технічне завдання

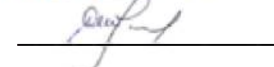
на розробку програмного виробу «Методи навчання згорткової нейронної мережі на основі популяційного алгоритму»

1.	Введення	1.1. Назва: Методи навчання згорткової нейронної мережі на основі популяційного алгоритму. 1.2. Галузь застосування: класифікація, аналіз.
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 151 – Автоматизація та комп'ютерно-інтегровані технології 2.2. Завдання на кваліфікаційну роботу магістра № ___________ від «___» ___________2022 (представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3.	Призначення розробки	3.1. Мета розробки: дослідження можливості комбінування популяційного алгоритму із згортковою нейронною мережею та розробка програмної моделі. 3.2. Призначення розробки надає можливість розпізнавати вхідний потік даних за допомогою програмної моделі. 3.3. Вихідні дані розробки: статистичні експериментальні дані, отримані від програмної моделі.
4.	Технічні вимоги до програмного виробу	4.1. Вимоги до функціональних характеристик: популяційний алгоритм повинен приймати участь у навчанні нейронної мережі 4.2. Вимоги до надійності: забезпечувати точність отриманих результатів 4.3.Вимоги до умов експлуатації: немає 4.4. Вимоги до складу і параметрів технічних засобів: ПК, вихід до інтернету. 4.5. Вимоги до інформаційної та програмної сумісності: немає 4.6. Вимоги до маркування та упаковки: немає 4.7. Вимоги до транспортування і зберігання: на звичайних носіях інформації 4.8. Спеціальні вимоги: немає.
5.	Вимоги до	Програмною документацією до виробу «Методи навчання згорткової нейронної мережі на основі

	програмної документації	популяційного алгоритму» вважати: 1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Методику дослідження (у вигляді глави 3.1.3 пояснювальної записки до кваліфікаційної роботи). 3) Опис моделі (представити в Розділі 3 пояснювальної записки до кваліфікаційної роботи). 4) Код програми (представити в Додатку Г до пояснювальної записки до кваліфікаційної роботи).	
6.	Вимоги до техніко-економічних показників	1) Оцінка економічної ефективності – непотрібна.	
7.	Стадії і етапи розробки	Дата	Назва етапу
		від 10 грудня 2021 до 12 січня 2022	Проведення аналізу літературних джерел по методам навчання згорткової нейронної мережі.
		від 13 січня 2022 до 31 січня 2022	Розробка алгоритму навчання згорткової нейронної мережі на основі популяційного алгоритму
		від 1 лютого 2022 до 27 лютого 2022	Програмна реалізація моделі згорткової нейронної мережі на основі популяційного алгоритму.
		від 28 лютого 2022 до 29 березня 2022	Проведення тесту працездатності.

		від 30 березня 2022 до 30 квітня 2022	Оцінка якості роботи моделі та порівняння з іншою.
		від 1 травня 2022 до 31 травня 2022	Розробка висновків та практичних рекомендацій.
		від 1 червня 2022 до 30 серпня 2022	Написання пояснювальної записки.
		від 31 серпня 2022 до 1 вересня 2022	Представлення роботи науковому керівнику.
		від 1 вересня 2022 до 12 жовтня 2022	Написання звіту з науково-дослідної практики.
		від 13 жовтня 2022 до 20 листопада 2022	Написання звіту з преддипломної практики.
8.	Порядок контролю і приймання програмного продукту (моделі)	1. Перевірку ходу розробки програми виконувати раз в 3 тижні. 2. Захист розробленої моделі провести на засіданні Атестаційної комісії. 3. Пояснювальну записку подати на паперових носіях в 1 примірнику і в електронному вигляді в 1 примірнику на CD-R компакт-диску.	

Виконавець
студент групи КУ- 61
Безсмертний Д. Р.

Замовник
д. т. н.,
Шматков С.І.


ДОДАТОК В

ПРОГРАМА І МЕТОДИКА ВИПРОБУВАНЬ

програмного виробу «Модель згорткової нейронної мережі на основі популяційного алгоритму»

1. Об'єкт випробувань

1.1 Об'єктом випробувань є модель згорткової нейронної мережі на основі популяційного алгоритму.

2. Мета випробувань

Перевірка відповідності функціональності програмної реалізації заявленим функціональним можливостям в технічному завданні (Додаток Б до пояснювальної записки до кваліфікаційної роботи).

3. Загальні положення

3.1 Підставою для проведення випробувань є наказ про призначення атестаційної комісії.

3.2 Місце і тривалість випробувань

Приймальні (приймально-здавальні) випробування проводяться на базі комп'ютерного класу кафедри в період роботи атестаційної комісії.

3.3 Обсяг випробувань

Приймальні випробування програмного виробу проводяться в обсязі відповідному цієї Програми і методики випробувань.

3.4 Організації, які беруть участь у випробуваннях

Приймальні випробування проводяться атестаційною комісією напередодні засідання (або в процесі засідання) за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

Програма повинна:

1. Представляти з себе модель згорткової нейронної мережі на основі популяційного алгоритму.
2. Забезпечувати максимальну точність навчання.
3. Програма повинна розпізнавати образи датасету MNIST.
4. Для виконання програми необхідний ПК з доступом до мережі

Інтернет та браузером.

5. Програма працює на різних операційних системах: Linux, Windows та інших.

6. Вимоги до маркування та упаковки (не висуваються).

7. Вимоги до транспортування і зберігання (не висуваються).

8. Спеціальні вимоги (не пред'являються).

5. Вимоги до програмної документації

Склад програмної документації, що подається на випробування, включає:

1) Справжнє Технічне завдання на розробку програмного виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи).

2) Програму і методику випробувань розробленого програмного виробу (представити у вигляді Додатку В до пояснювальної записки до кваліфікаційної роботи).

3) Опис моделі (представити в Розділі 3 пояснювальної записки до кваліфікаційної роботи).

4) Код програми (представити в Додатку Г до пояснювальної записки до кваліфікаційної роботи).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Для виконання програми необхідний ПК з доступом до мережі Інтернет та браузером.

Програма працює на різних операційних системах: Linux, Windows та інших.

Для проведення випробувань необхідно зайти в хмарну платформу Google Colaboratory.

6.2 Порядок проведення випробувань

1. Перевірка програмної документації

1.1. Перевірка складу програмної документації. Перевірку здійснювати за критерієм наявності, представленої в ТЗ документації.

1.2. Перевірка якості програмної документації. Перевірку здійснювати за критерієм відповідності вимогам ГОСТ 19.301-79 ЕСПД. «Програма і методика випробувань».

2. Перевірка працездатності моделі

Тест 1

2.1. Перевірка працездатності

2.1.1. Перевірку здійснювати за критерієм аналізу отриманих даних.

```
current config:  [14, 4, 2, 2]  val:  0.9889000058174133
current config:  [12, 5, 3, 3]  val:  0.9891999959945679
current config:  [13, 8, 3, 3]  val:  0.9891999959945679
current config:  [13, 4, 2, 2]  val:  0.9894999861717224
current config:  [15, 6, 3, 3]  val:  0.9896000027656555
current config:  [16, 7, 2, 4]  val:  0.9891999959945679
current config:  [13, 6, 3, 3]  val:  0.9901000261306763
current config:  [15, 7, 2, 4]  val:  0.9900000095367432
current config:  [12, 5, 3, 3]  val:  0.991100013256073
```

Рисунок В.1 – Робота моделі

Тест 2

2.2. Перевірка отриманих результатів.

2.2.1. Перевірку здійснювати за критерієм аналізу якості розпізнавання.

```
print('Test loss: ', bp)
print('Test accuracy: ', value, v)

Test loss:  [12, 5, 3, 3]
Test accuracy:  0.9860000014305115 0.991100013256073
```

Рисунок В.2 – Результати розпізнавання

Висновки: випробування вважаються успішно пройденими, якщо були виконані всі перевірки з пунктів 1,2.

Виконавець

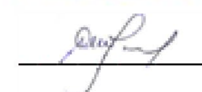
студент групи КУ-61

Безсмертний Д.Р.

Замовник

д.т.н., професор

Шматков С.І.



ДОДАТОК Г

Лістинг коду моделі на основі популяційного алгоритму

```
# для скорейшего выполнения данного кода надо зайти в Runtime -
> Change runtime type -> GPU

from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

import copy
import numpy as np


batch_size = 128
num_classes = 10
epochs = 100

# размеры входного изображения
img_rows, img_cols = 28, 28

# данные mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'записей для тренировки')
```

```
print(x_test.shape[0], 'записей для теста')
# преобразовываем векторы классов в бинарные матрицы классов
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
def func(x):
  n, sf, sp, l = x[0], x[1], x[2], x[3]

  model = Sequential()
  model.add(Conv2D(32,kernel_size=(3, 3),
                   activation='relu',
                   input_shape=input_shape))
  model.add(Conv2D(64, (3, 3), activation='relu'))
  model.add(Conv2D(n, (sf, sf), activation='relu'))
  model.add(MaxPooling2D(pool_size=(sp, sp), strides=(l, l)))
  model.add(Flatten())
  model.add(Dense(num_classes, activation='softmax'))
  model.compile(loss=keras.losses.categorical_crossentropy,
                optimizer=keras.optimizers.Adam(),
                metrics=['accuracy'])

  cp = [keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, verb
ose=0, mode='auto')]

  model.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            verbose=0,
            validation_data=(x_test, y_test),
            callbacks=cp)

  score = model.evaluate(x_test, y_test, verbose=0)

  # loss, val
  print('current config: ', x, ' val: ', score[1])
  return score[1]
class Particle(object):
    """Класс частиц для PSO
    Этот класс инкапсулирует поведение каждой частицы в PSO и обеспечивает
 эффективный способ ведения учета состояния роя в любой заданной итерации.

    Args:
        lower_bound (np.array): Вектор нижних пределов границ размеров час
тиц
        upper_bound (np.array): Вектор верхних пределов границ размеров ча
стиц
        dimensions (int): Количество измерений пространства поиска
```

```
        objective function (function): Функция черного ящика для оценки
    """
    def __init__(self,
                 lower_bound,
                 upper_bound,
                 dimensions,
                 objective_function):
        self.reset(dimensions, lower_bound, upper_bound, objective_functio
n)

    def reset(self,
              dimensions,
              lower_bound,
              upper_bound,
              objective_function):
        """Сброс частиц
        Позволяет сбросить частицу без перераспределения

        Args:
          lower_bound (np.array): Вектор нижних пределов границ размеров ч
астиц
          upper_bound (np.array): Вектор верхних пределов границ размеров
частиц
          dimensions (int): Количество измерений пространства поиска
        """
        position = []
        for i in range(dimensions):
            if lower_bound[i] < upper_bound[i]:
                position.extend(np.random.randint(lower_bound[i], upper_bo
und[i] + 1, 1, dtype=int))
            elif lower_bound[i] == upper_bound[i]:
                position.extend(np.array([lower_bound[i]], dtype=int))
            else:
                assert False

        self.position = [position]

        self.velocity = [np.multiply(np.random.rand(dimensions),
                                     (upper_bound -
 lower_bound)).astype(int)]

        self.best_position = self.position[:]

        self.function_value = [objective_function(self.best_position[-1])]
        self.best_function_value = self.function_value[:]
```

```
    def update_velocity(self, omega, phip, phig, best_swarm_position):

        random_coefficient_p = np.random.uniform(size=np.asarray(self.posi
tion[-1]).shape)
        random_coefficient_g = np.random.uniform(size=np.asarray(self.posi
tion[-1]).shape)

        self.velocity.append(omega
                             * np.asarray(self.velocity[-1])
                             + phip
                             * random_coefficient_p
                             * (np.asarray(self.best_position[-1])
                                - np.asarray(self.position[-1]))
                             + phig
                             * random_coefficient_g
                             * (np.asarray(best_swarm_position)
                                - np.asarray(self.position[-1])))

        self.velocity[-1] = self.velocity[-1].astype(int)

    def update_position(self, lower_bound, upper_bound, objective_function
):
        new_position = self.position[-1] + self.velocity[-1]

        if np.array_equal(self.position[-1], new_position):
            self.function_value.append(self.function_value[-1])
        else:
            mark1 = new_position < lower_bound
            mark2 = new_position > upper_bound

            new_position[mark1] = lower_bound[mark1]
            new_position[mark2] = upper_bound[mark2]

            self.function_value.append(objective_function(self.position[-
1]))

        self.position.append(new_position.tolist())

        if self.function_value[-1] < self.best_function_value[-1]:
            self.best_position.append(self.position[-1][:])
            self.best_function_value.append(self.function_value[-1])

class Pso(object):
    """Обёртка PSO
       Этот класс содержит частицы и предоставляет абстракцию для хранения
 всего контекста алгоритма PSO.
```

```
        Args:
            swarmsize (int): Количество частиц в рое
            maxiter (int): Максимальное количество поколений, которое будет
 выполняться роем
    """
    def __init__(self, swarmsize=100, maxiter=100):
        self.max_generations = maxiter
        self.swarmsize = swarmsize

        self.omega = 0.5
        self.phip = 0.5
        self.phig = 0.5

        self.minstep = 1e-4
        self.minfunc = 1e-4

        self.best_position = [None]
        self.best_function_value = [1]

        self.particles = []

        self.retired_particles = []

    def run(self, function, lower_bound, upper_bound, kwargs=None):
        if kwargs is None:
            kwargs = {}

        objective_function = lambda x: function(x, **kwargs)
        assert hasattr(function, '__call__'), 'Invalid function handle'

        assert len(lower_bound) == len(upper_bound), 'Invalid bounds lengt
h'

        lower_bound = np.array(lower_bound)
        upper_bound = np.array(upper_bound)

        assert np.all(upper_bound > lower_bound), 'Invalid boundary values
'


        dimensions = len(lower_bound)

        self.particles = self.initialize_particles(lower_bound,
                                                   upper_bound,
                                                   dimensions,
```

```
                                                      objective_function)

        # Начнём эволюцию
        generation = 1
        while generation <= self.max_generations:
            for particle in self.particles:
                particle.update_velocity(self.omega, self.phip, self.phig,
 self.best_position[-1])
                particle.update_position(lower_bound, upper_bound, objecti
ve_function)

                if particle.best_function_value[-1] == 0:
                    self.retired_particles.append(copy.deepcopy(particle))
                    particle.reset(dimensions, lower_bound, upper_bound, o
bjective_function)
                elif particle.best_function_value[-
1] < self.best_function_value[-1]:
                    stepsize = np.sqrt(np.sum((np.asarray(self.best_positi
on[-1])
                                                          -
 np.asarray(particle.position[-1])) ** 2))

                    if np.abs(np.asarray(self.best_function_value[-1])
                              - np.asarray(particle.best_function_value[-
1])) \
                            <= self.minfunc:
                        return particle.best_position[-
1], particle.best_function_value[-1]
                    elif stepsize <= self.minstep:
                        return particle.best_position[-
1], particle.best_function_value[-1]
                    else:
                        self.best_function_value.append(particle.best_func
tion_value[-1])
                        self.best_position.append(particle.best_position[-
1][:])

            generation += 1

        return self.best_position[-1], self.best_function_value[-1]

    def initialize_particles(self,
                             lower_bound,
                             upper_bound,
                             dimensions,
                             objective_function):
```

```
        particles = []
        for _ in range(self.swarmsize):
            particles.append(Particle(lower_bound,
                                      upper_bound,
                                      dimensions,
                                      objective_function))
            if particles[-1].best_function_value[-
1] < self.best_function_value[-1]:
                self.best_function_value.append(particles[-
1].best_function_value[-1])
                self.best_position.append(particles[-1].best_position[-1])


        self.best_position = [self.best_position[-1]]
        self.best_function_value = [self.best_function_value[-1]]

        return particles
pso = Pso(swarmsize=4, maxiter=14)
# n, sf, sp, l
bp, value = pso.run(func, [1, 2, 2, 2], [16, 8, 4, 4])

v = func(bp)
```

Лістинг коду моделі на основі генетичного алгоритму

```
# для скорейшего выполнения данного кода надо зайти в Runtime -
> Change runtime type -> GPU

import os
import json
import random
import numpy as np
import tensorflow as tf

from abc import abstractmethod, ABC
from typing import Iterable, Callable, Union, Sequence, Dict, Any, Tuple,
List
from tensorflow.python.keras.optimizer_v2.optimizer_v2 import OptimizerV2
from typing import Dict, Callable, Iterable, Union, Tuple, Sequence, Any,
List


class Layer(ABC):
    @abstractmethod
    def tensor_rep(self, inputs: tf.keras.layers.Layer) -
> tf.keras.layers.Layer:
```

```
        """
        Возвращает представление слоя keras объекта
        :param inputs: Предыдущий слой будет передан в качестве входных да
нных для следующего
        :return: представление слоя keras объекта
        """
        pass


class SkipLayer(Layer):
    GROUP_NUMBER = 1

    def __init__(self, feature_size1: int,
                 feature_size2: int,
                 kernel: Tuple[int, int] = (3, 3),
                 stride: Tuple[int, int] = (1, 1),
                 convolution: str = 'same'):
        """
        Инициализирует параметры слоя пропуска
        Слой Skip (пропуск) состоит из:
        1. Свёртки:
            * filter size: feature_size1
            * kernel size: kernel
            * stride size: stride
        2. Пакетной нормализации
        3. ReLU активации
        4. Свёртки:
            * filter size: feature_size2
            * kernel size: kernel
            * stride size: stride
        5. Пакетной нормализации
        6. Вход + предыдущие выходы
        7. ReLU активация
        :param feature_size1: размер фильтра первой свертки должен быть ст
епенью 2
        :param feature_size2: размер фильтра второй свертки должен быть ст
епенью 2
        :param kernel: размер ядра для всех сверток, по умолчанию: (3, 3)
        :param stride: размер шага для всех сверток, по умолчанию: (1, 1)
        :param convolution: тип заполнения для всех сверток должен быть ли
бо "valid", либо "same"
        """
        self.convolution = convolution
        self.stride = stride
        self.kernel = kernel
        self.feature_size2 = feature_size2
```

```
        self.feature_size1 = feature_size1

    def tensor_rep(self, inputs: tf.keras.layers.Layer) -
> tf.keras.layers.Activation:
        group_name = f'SkipLayer_{SkipLayer.GROUP_NUMBER}'
        SkipLayer.GROUP_NUMBER += 1

        skip_layer = tf.keras.layers.Conv2D(self.feature_size1, self.kerne
l, self.stride, self.convolution,
                                            name=f'{group_name}/Conv1')(in
puts)

        skip_layer = tf.keras.layers.BatchNormalization(name=f'{group_name
}/BatchNorm1')(skip_layer)
        skip_layer = tf.keras.layers.Activation('relu', name=f'{group_name
}/ReLU1')(skip_layer)

        skip_layer = tf.keras.layers.Conv2D(self.feature_size2, self.kerne
l, self.stride, self.convolution,
                                            name=f'{group_name}/Conv2')(sk
ip_layer)
        skip_layer = tf.keras.layers.BatchNormalization(name=f'{group_name
}/BatchNorm2')(skip_layer)

        # Следит за тем, чтобы размерность на пропускаемых слоях была один
аковой
        inputs = tf.keras.layers.Conv2D(self.feature_size2, (1, 1), self.s
tride, name=f'{group_name}/Reshape')(inputs)

        outputs = tf.keras.layers.add([inputs, skip_layer], name=f'{group_
name}/Add')
        return tf.keras.layers.Activation('relu', name=f'{group_name}/ReLU
2')(outputs)

    def __repr__(self) -> str:
        return f'{self.feature_size1}-{self.feature_size2}'


class PoolingLayer(Layer):
    pooling_choices = {
        'max': tf.keras.layers.MaxPool2D,
        'mean': tf.keras.layers.AveragePooling2D
    }

    def __init__(self, pooling_type: str, kernel: Tuple[int, int] = (2, 2)
, stride: Tuple[int, int] = (2, 2)):
```

```
        """
        Слой объединения, это либо слой MaxPooling, либо слой AveragePooli
ng.
        :param pooling_type: либо "max", либо "mean", это определяет тип с
лоя объединения
        :param kernel: размер ядра для слоя пула, по умолчанию: (2, 2)
        :param stride: размер шага для слоя объединения, по умолчанию: (2,
 2)
        """
        self.stride = stride
        self.kernel = kernel
        self.pooling_type = pooling_type

    def tensor_rep(self, inputs: tf.keras.layers.Layer) -> Union[
        tf.keras.layers.MaxPool2D, tf.keras.layers.AveragePooling2D]:
        return PoolingLayer.pooling_choices[self.pooling_type](pool_size=s
elf.kernel, strides=self.stride)(inputs)

    def __repr__(self) -> str:
        return self.pooling_type


class CNN:
    def __init__(self, input_shape: Sequence[int],
                 output_function: Callable[[tf.keras.layers.Layer], tf.ker
as.layers.Layer],
                 layers: Sequence[Layer],
                 optimizer: OptimizerV2 = None,
                 loss: Union[str, tf.keras.losses.Loss] = 'sparse_categori
cal_crossentropy',
                 metrics: Iterable[str] = ('accuracy',),
                 load_if_exist: bool = True,
                 extra_callbacks: Iterable[tf.keras.callbacks.Callback] =
None,
                 logs_dir: str = './logs/train_data',
                 checkpoint_dir: str = './checkpoints') -> None:
        """
        Инициализирует CNN.

        :param input_shape: входная форма ввода CNN должна быть размером н
е менее 2
        :param output_function: функция вывода для присоединения в конце с
лоев, это будет определять, что выводит CNN
        :param layers: список слоев для определения CNN
        :param optimizer: тип оптимизатора для использования при обучении
CNN
```

```
        :param loss: функция потерь для использования при обучении CNN
        :param metrics: метрика, используемая для количественной оценки то
го, насколько хороша CNN
        :param load_if_exist: если модель уже находится в checkpoint_dir,
используйте эти веса
        :param extra_callbacks: любые другие обратные вызовы для использов
ания при обучении режима, это может быть планировщик скорости обучения
        :param logs_dir: каталог, где хранить журналы Tensorboard
        :param checkpoint_dir: каталог, в котором можно хранить контрольны
е точки модели
        """
        self.checkpoint_dir = checkpoint_dir
        self.logs_dir = logs_dir
        self.load_if_exist = load_if_exist
        self.loss = loss

        if optimizer is None:
            self.optimizer = tf.keras.optimizers.Adam()
        else:
            self.optimizer = optimizer

        self.metrics = metrics
        self.output_function = output_function
        self.input_shape = input_shape
        if layers is None:
            self.layers = []
        else:
            self.layers = layers

        self.hash = self.generate_hash()

        self.model: tf.keras.Model = None

        self.checkpoint_filepath = f'{self.checkpoint_dir}/model_{self.has
h}/model_{self.hash}'
        model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
            filepath=self.checkpoint_filepath,
            save_weights_only=True,
            monitor='val_accuracy',
            mode='max',
            save_best_only=True)

        tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=f"{s
elf.logs_dir}/model_{self.hash}",
                                                              update_freq=
'batch', histogram_freq=1)
```

```
        self.callbacks = [model_checkpoint_callback, tensorboard_callback]

        if extra_callbacks is not None:
            self.callbacks.extend(extra_callbacks)

    def generate(self) -> tf.keras.Model:
        """
        Сгенерирует tf.keras.Model CNN на основе списка слоев, функции пот
ерь, оптимизатора и показателей
        :return: скомпилированная tf.keras.Model
        """

        print(self.layers)

        if self.model is None:
            tf.keras.backend.clear_session()
            SkipLayer.GROUP_NUMBER = 1
            inputs = tf.keras.Input(shape=self.input_shape)

            outputs = inputs

            for i, layer in enumerate(self.layers):
                outputs = layer.tensor_rep(outputs)

            outputs = self.output_function(outputs)

            self.model = tf.keras.Model(inputs=inputs, outputs=outputs)
            self.model.compile(self.optimizer, loss=self.loss, metrics=sel
f.metrics)

            SkipLayer.GROUP_NUMBER = 1
        return self.model

    def evaluate(self, data: Dict[str, Any], batch_size: int = 64) -
> Tuple[float, float]:
        """
        Оценивает модель - вычисляет точность модели на тестовых данных
        :param data: данные для тестирования, использует значения "x_test"
 и "y_test" данных для тестирования
        :param batch_size: размер партии для тестирования
        :return: потери и точность модели на тестовых данных
        """

        return self.model.evaluate(data['x_test'], data['y_test'], batch_s
ize=batch_size)
```

```
    def train(self, data: Dict[str, Any], batch_size: int = 64, epochs: in
t = 1) -> None:
        """
        Обучает определенную модель, cnn.generate() должна быть вызвана до
 запуска этой функции.

        Модель разделит обучающие данные, при этом 20% будут проходить про
верку, модель сохранит данные с лучшим результатом проверки после каждой э
похи.

        Если модель уже существует в определенном каталоге checkpoint_dir,
 вместо этого она просто загрузит сохраненные веса.

        При обучении модель использует обратные вызовы TensorBoard и Model
Checkpoint для автоматической регистрации и сохранения контрольных точек м
одели. Вы также можете добавить любые другие обратные вызовы через парамет
ры extra_callback в CNN __init__
        :param data: данные для обучения сети, это dict с параметрами 'x_t
rain' и 'y_train', которые содержат данные, которые будут использоваться в
 период обучения модели.
        :param batch_size: размер пакета для обучения сети
        :param epochs: количество эпох для обучения сети
        """

        if self.load_if_exist and os.path.exists(f'{self.checkpoint_dir}/m
odel_{self.hash}/'):
            self.model.load_weights(self.checkpoint_filepath)
        else:
            if self.model is not None:
                self.model.fit(data['x_train'], data['y_train'], batch_siz
e=batch_size, epochs=epochs,
                               validation_split=.2,
                               callbacks=self.callbacks)

    def generate_hash(self) -> str:
        """
        Генерирует хэш CNN на основе содержащихся в нем слоев:
          SkipLayer представлен как feature_size1-feature_size2
          PoolingLayer представлен как pooling_type
        Пример:
        '32-32-mean-max-256-32'
        :return: хэш CNN на основе структуры слоев
        """

        return '-'.join(map(str, self.layers))
```

```
    def __repr__(self) -> str:
        return self.hash


def get_layer_from_string(layer_definition: str) -> List[Layer]:
    """
    Создаем список слоев из хэша строки, чтобы его можно было передать в C
NN.

    Приклад:
    '128-64-mean-max-32-32'

    Будет сконвертирован:
    [SkipLayer(128, 64), PoolingLayer('mean'), PoolingLayer('max'), SkipLa
yer(32, 32)]
    SkipLayer представлен как feature_size1-feature_size2
    PoolingLayer представлен как pooling_type
    :param layer_definition: строковое представление слоев
    :return: список преобразованных слоев
    """

    layers_str: list = layer_definition.split('-')

    layers = []

    while len(layers_str) > 0:
        if layers_str[0].isdigit():
            f = SkipLayer(int(layers_str[0]), int(layers_str[0 + 1]))
            layers_str.pop(0)
            layers_str.pop(0)
        else:
            f = PoolingLayer(layers_str[0])
            layers_str.pop(0)
        layers.append(f)

    return layers
class AutoCNN:
    def get_input_shape(self) -> Tuple[int]:
        """
        Определяет входную форму с учетом входных данных
        :return: Возвращает форму ввода, если ввод представляет собой изоб
ражение в градациях серого, он добавляет еще одно измерение
        """
        shape = self.dataset['x_train'].shape[1:]
```

```
        if len(shape) < 3:
            shape = (*shape, 1)

        return shape

    def get_output_function(self) -
> Callable[[tf.keras.layers.Layer], tf.keras.layers.Layer]:
        """
        Создает выходной слой по умолчанию, он использует выход с one hot
encoding  для сети.
        Функция находит количество уникальных значений в 'y_train' данных,
 а затем генерирует функцию, которая:
        1. сглаживает предыдущий вывод
        2. использует плотный слой с N уникальными выходами 'y_train', доб
авляет активацию softmax в конце
        :return: Простая функция вывода
        """
        output_size = np.unique(self.dataset['y_train']).shape[0]

        def output_function(inputs):
            out = tf.keras.layers.Flatten()(inputs)

            return tf.keras.layers.Dense(output_size, activation='softmax'
)(out)

        return output_function

    def __init__(self, population_size: int,
                 maximal_generation_number: int,
                 dataset: Dict[str, Any],
                 output_layer: Callable[[tf.keras.layers.Layer], tf.keras.
layers.Layer] = None,
                 epoch_number: int = 1,
                 optimizer: OptimizerV2 = tf.keras.optimizers.Adam(),
                 loss: Union[str, tf.keras.losses.Loss] = 'sparse_categori
cal_crossentropy',
                 metrics: Iterable[str] = ('accuracy',),
                 crossover_probability: float = .9,
                 mutation_probability: float = .2,
                 mutation_operation_distribution: Sequence[float] = None,
                 fitness_cache: str = 'fitness.json',
                 extra_callbacks: Iterable[tf.keras.callbacks.Callback] =
None,
                 logs_dir: str = './logs/train_data',
                 checkpoint_dir: str = './checkpoints'
                 ) -> None:
```

```
        """
        Инициализирует AutoCNN
        :param population_size: количество CNN для каждого поколения
        :param maximal_generation_number: максимальное количество поколени
й
        :param dataset: данные для обучения и тестирования, это словарь с
ключами, 'x_train', 'y_train', 'x_test', 'y_test'
        :param output_layer: слой для добавления в конец CNN, этот слой во
зврашает выходные данные CNN
        :param epoch_number: количество эпох для обучения каждой CNN
        :param optimizer: оптимизатор для использования CNN
        :param loss: функция потерь, используемая для CNN
        :param metrics: метрики, используемые для количественной оценки то
го, насколько хороша CNN
        :param crossover_probability: вероятность смешивания двух родитель
ских CNN
        :param mutation_probability: вероятность мутировать потомство CNN
        :param mutation_operation_distribution: распределение вероятности
выбора одной из 4 операций мутации
        :param fitness_cache: расположение файла для сохранения значений п
ригодности
        :param extra_callbacks: любые другие обратные вызовы для использов
ания при режиме обучения, это может быть планировщик скорости обучения
        :param logs_dir: каталог, где хранить журналы Tensorboard
        :param checkpoint_dir: каталог, в котором можно хранить контрольны
е точки модели
        """
        self.logs_dir = logs_dir
        self.checkpoint_dir = checkpoint_dir
        self.extra_callbacks = extra_callbacks
        self.fitness_cache = fitness_cache

        if self.fitness_cache is not None and os.path.exists(self.fitness_
cache):
            with open(self.fitness_cache) as cache:
                self.fitness = json.load(cache)
        else:
            self.fitness = dict()

        if mutation_operation_distribution is None:
            self.mutation_operation_distribution = (.7, .1, .1, .1)
        else:
            self.mutation_operation_distribution = mutation_operation_dist
ribution

        self.mutation_probability = mutation_probability
```

```
        self.crossover_probability = crossover_probability
        self.epoch_number = epoch_number
        self.metrics = metrics
        self.loss = loss
        self.optimizer = optimizer
        self.dataset = dataset
        self.maximal_generation_number = maximal_generation_number
        self.population_size = population_size
        self.population = []

        self.population_iteration = 0

        if output_layer is None:
            self.output_layer = self.get_output_function()
        else:
            self.output_layer = output_layer

        self.input_shape = self.get_input_shape()

    def initialize(self) -> None:
        """
        Инициализирует популяцию CNN
        Генерирует CNN population_size с помощью:
        1. выбора начальной случайной глубины между [1-5] слоями
        2. для каждого слоя
            1. выбирает число между (0-1)
            2. если число < .5
                * Создаем SkipLayer со случайными размерами фильтра
            3. если число >= .5
                * Выбираем между случайным максимальным объединением или с
редним слоем объединения
        """

        self.population.clear()

        for _ in range(self.population_size):
            depth = random.randint(1, 5)

            layers = []

            for i in range(depth):
                r = random.random()

                if r < .5:
                    layers.append(self.random_skip())
                else:
```

```
                layers.append(self.random_pooling())

            cnn = self.generate_cnn(layers)

            self.population.append(cnn)

    def random_skip(self) -> SkipLayer:
        """
        Генерирует случайно созданный SkipLayer
        Случайным образом выбирает два размера фильтра для слоев свертки,
это степень двойки и от 32 до 512.
        :return: случайно сгенерированный SkipLayer
        """

        f1 = 2 ** random.randint(5, 9)
        f2 = 2 ** random.randint(5, 9)
        return SkipLayer(f1, f2)

    def random_pooling(self) -> PoolingLayer:
        """
        Случайным образом выбирает с равной вероятностью либо максимальный
, либо средний объединяющий слой
        :return: случайно выбранный объединяющий слой
        """
        q = random.random()

        if q < .5:
            return PoolingLayer('max')
        else:
            return PoolingLayer('mean')

    def evaluate_fitness(self, population: Iterable[CNN]) -> None:
        """
        Оценивает приспособленность каждого из индивидуумов в популяции
        :param population: население CNN для оценки пригодности
        """

        for cnn in population:
            if cnn.hash not in self.fitness:
                self.evaluate_individual_fitness(cnn)

            print(cnn, self.fitness[cnn.hash])

    def evaluate_individual_fitness(self, cnn: CNN) -> None:
        """
```

```
        Оценивает пригодность CNN. Пригодность основана на точности CNN на
 тестовых данных.
        Не выполняет повторную оценку пригодности, поскольку она уже наход
ится в кэше пригодности.
        После расчета фитнеса он добавляется в кэш фитнеса и сохраняется в
 кэше json.
        :param cnn: CNN, чтобы найти пригодность
        """
        try:
            cnn.generate()

            cnn.train(self.dataset, epochs=self.epoch_number)
            loss, accuracy = cnn.evaluate(self.dataset)
        except ValueError as e:
            print(e)
            accuracy = 0

        self.fitness[cnn.hash] = accuracy

        if self.fitness_cache is not None:
            with open(self.fitness_cache, 'w') as json_file:
                json.dump(self.fitness, json_file)

    def select_two_individuals(self, population: Sequence[CNN]) -> CNN:
        """
        Случайным образом выбирает двух особей без замены и оставляет ту,
у которой наибольшая приспособленность.
        :param population: CNN для выбора из двух CNN
        :return: CNN с самой высокой пригодностью
        """
        cnn1, cnn2 = random.sample(population, 2)

        if self.fitness[cnn1.hash] > self.fitness[cnn2.hash]:
            return cnn1
        else:
            return cnn2

    def split_individual(self, cnn: CNN) -
> Tuple[Sequence[Layer], Sequence[Layer]]:
        """
        Случайным образом разбивает слои CNN на две части
        :param cnn: CNN, чтобы разделить слои
        :return: две части слоев CNN
        """
        split_index = random.randint(0, len(cnn.layers))
```

```
        return cnn.layers[:split_index], cnn.layers[split_index:]

    def generate_offsprings(self) -> List[CNN]:
        """
        Генерирует потомство текущей популяции.
        Шаги:
        1. В то время как популяция потомства меньше, чем текущая численно
сть популяции
            1. случайным образом выберем 2-
х разных особей из текущей популяции и оставим ту, у которой наибольшая пр
испособленность
            2. сделаем то же самое, что и на шаг 1, убедимся, что они разн
ые
            3. выберем случайное число между (0, 1)
            4. Если число меньше, чем crossover_probability
                1. случайным образом разделим две родительские CNN
                2. создадим два новых потомка, поменяв местами два родител
ьских слоя
                3. добавить их к потомству популяции
            5. В противном случаем
                1. добавим двух родителей к потомкам
        2. Пройдём через мутации
            1. За каждое сгенерированное потомство
                1. выберем случайное число между (0, 1)
                2. Если число меньше, чем mutation_probability
                    1. случайно выберем одну из 4 возможных мутаций, учиты
вая mutation_operation_distribution distribution
                    2. add_skip: добавляет слой со случайным пропуском, ув
еличивает сложность и глубину сети
                    3. add_pooling: добавляет случайный слой пула, увеличи
вает глубину, но снижает сложность
                    4. remove: удаляет слой, уменьшает сложность и глубину
                    5. change: изменяет параметры слоя (например, размер ф
ильтра, максимальное или среднее объединение)
        :return: потомков нынешнего поколения
        """
        offsprings = []

        while len(offsprings) < len(self.population):
            p1 = self.select_two_individuals(self.population)
            p2 = self.select_two_individuals(self.population)

            while p1.hash == p2.hash:
                p2 = self.select_two_individuals(self.population)

            r = random.random()
```

```
            if r < self.crossover_probability:
                p1_1, p1_2 = self.split_individual(p1)
                p2_1, p2_2 = self.split_individual(p2)

                o1 = [*p1_1, *p2_2]
                o2 = [*p2_1, *p1_2]

                offsprings.append(o1)
                offsprings.append(o2)
            else:
                offsprings.append(p1.layers)
                offsprings.append(p2.layers)

        choices = ['add_skip', 'add_pooling', 'remove', 'change']

        for cnn in offsprings:
            cnn: list

            r = random.random()

            if r < self.mutation_probability:
                if len(cnn) == 0:
                    i = 0
                    operation = random.choices(choices[:2], weights=self.m
utation_operation_distribution[:2])[0]
                else:
                    i = random.randint(0, len(cnn) - 1)
                    operation = random.choices(choices, weights=self.mutat
ion_operation_distribution)[0]

                if operation == 'add_skip':
                    cnn.insert(i, self.random_skip())
                elif operation == 'add_pooling':
                    cnn.insert(i, self.random_pooling())
                elif operation == 'remove':
                    cnn.pop(i)
                else:
                    if isinstance(cnn[i], SkipLayer):
                        cnn[i] = self.random_skip()
                    else:
                        cnn[i] = self.random_pooling()

        offsprings = [self.generate_cnn(layers) for layers in offsprings]

        return offsprings
```

```
    def generate_cnn(self, layers: Sequence[Layer]) -> CNN:
        """
        Генерирует CNN с параметрами AutoCNN и требуемыми слоями
        :param layers: слои, чтобы добавить к CNN
        :return: созданная CNN
        """
        return CNN(self.input_shape, self.output_layer, layers, optimizer=
self.optimizer, loss=self.loss,
                   metrics=self.metrics, extra_callbacks=self.extra_callba
cks, logs_dir=self.logs_dir,
                   checkpoint_dir=self.checkpoint_dir)

    def environmental_selection(self, offsprings: Sequence[CNN]) -
> Iterable[CNN]:
        """
        Выберает новую популяцию из потомков и текущей популяции
        Шаги:
        1. До тех пор, пока новый размер популяции не будет таким же, как
предыдущий:
            * Выберем двух особей для замены
            * добавим лучшую из двух к новой популяции
        2. Убедимся, что предыдущий лучший CNN находится в новой популяции
            * если лучшего нет в популяции, удалим худшего из новой популя
ции и заменим его
        :param offsprings: потомки поколения
        :return: новое население для использования в следующем поколении
        """
        whole_population = list(self.population)
        whole_population.extend(offsprings)

        new_population = []

        while len(new_population) < len(self.population):
            p = self.select_two_individuals(whole_population)

            new_population.append(p)

        best_cnn = max(whole_population, key=lambda x: self.fitness[x.hash
])

        print("Best CNN:", best_cnn, "Score:", self.fitness[best_cnn.hash]
)

        if best_cnn not in new_population:
```

```
            worst_cnn = min(new_population, key=lambda x: self.fitness[x.h
ash])
            print("Worst CNN:", worst_cnn, "Score:", self.fitness[worst_cn
n.hash])
            new_population.remove(worst_cnn)
            new_population.append(best_cnn)

        return new_population

    def run(self) -> CNN:
        """
        Проходит через весь метод генетического алгоритма
        Шаги:
        1. инициализирует популяцию
        2. Для maximal_generation_number генераций:
            1. оценим приспособленность населения
            2. произведем потомство
            3. оценим приспособленность потомства
            4. выберем новую популяцию из потомков и текущей популяции
        :return: лучшая CNN, найденная после всех поколений
        """
        print("Initializing Population")
        self.initialize()
        print("Population Initialization Done:", self.population)

        for i in range(self.maximal_generation_number):
            print("Generation", i)

            print("Evaluating Population fitness")
            self.evaluate_fitness(self.population)
            print("Evaluating Population fitness Done:", self.fitness)

            print("Generating Offsprings")
            offsprings = self.generate_offsprings()
            print("Generating Offsprings Done:", offsprings)

            print("Evaluating Offsprings")
            self.evaluate_fitness(offsprings)
            print("Evaluating Offsprings Done:", self.fitness)

            print("Selecting new environment")
            new_population = self.environmental_selection(offsprings)
            print("Selecting new environment Done:", new_population)

            self.population = new_population
```

```
        best_cnn = sorted(self.population, key=lambda x: self.fitness[x.ha
sh])[-1]
        print("Best CNN:", best_cnn, "Score:", self.fitness[best_cnn.hash]
)
        return best_cnn
import tensorflow as tf
import random

# Устанавливает случайные seed-
ы, чтобы сделать тестирование более последовательным
random.seed(42)
tf.random.set_seed(42)

# Загружает данные для теста и обучения
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Помещает данные в словарь для использования алгоритмом
data = {'x_train': x_train, 'y_train': y_train, 'x_test': x_test, 'y_test'
: y_test}

# Устанавливаем желаемые параметры
a = AutoCNN(population_size=5, maximal_generation_number=4, dataset=data,
epoch_number=5)

# Запускаем алгоритм до тех пор, пока не будет достигнуто значение maximal
_generation_number
best_cnn = a.run()
print(best_cnn)
```