МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
Тернопільський національний економічний університет
Факультет комп'ютерних інформаційних технологій
Кафедра комп'ютерної інженерії

Лящинський Павло Борисович

Синтез структур згорткових нейронних мереж для класифікації біомедичних зображень / Synthesis of the convolution neural networks structures for biomedical images classification

спеціальність:123 – Комп'ютерна інженерія
освітньо-професійна програма – Комп'ютерна інженерія

Випускна кваліфікаційна робота

Виконав студент групи КІм-21
П. Б. Лящинський
_____

Науковий керівник:
д.т.н., професор, О. М. Березький
_____

ТЕРНОПІЛЬ - 2019

РЕЗЮМЕ

Випускова кваліфікаційна робота на тему “Синтез структур згорткових нейронних мереж для класифікації біомедичних зображень” на здобуття освітньо-кваліфікаційного рівня “Магістр” зі спеціальності “Комп’ютерна інженерія” написана обсягом 89 сторінок і містить 40 ілюстрацій, 4 таблиці, 7 додатків та 63 джерела за переліком посилань.

Метою ВКР є розробка алгоритмів синтезу структур згорткових нейронних мереж та їх програмної реалізації для підвищення точності класифікації біомедичних зображень.

У випусковій кваліфікаційній роботі на основі аналізу методів і засобів синтезу структур ЗНМ програмно реалізовано автоматичний синтез структур згорткових нейронних мереж. При цьому використано алгоритми повного перебору та випадкового пошуку.

Суть розробленого програмного продукту полягає в можливості автоматизованого синтезу структур згорткових нейронних мереж для класифікації біомедичних зображень.

Ключові слова: АЛГОРИТМ, СИНТЕЗ, АНАЛІЗ, ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ.

RESUME

The initial qualification work on the topic "Synthesis of the convolution neural networks structures for biomedical images classification" for obtaining the Master's degree in Computer Engineering specialty is written 89 pages and contains 40 illustrations, 4 tables, 7 appendices and 63 sources by the list of links.

The method of IQW is to send out algorithms for the synthesis of the broadest neuronal measures produced and their programming, which is used to classify classifications of biomedical images.

Using professional activities that are relevant to important methods and use synthesis of CNN technology, the program actually uses automatic synthesis of the produced neutron measurements. In doing so, algorithms are constantly reviewed and searched.

The sum of the developed software product was in modern automated synthesis of many neural measures for classification of biomedical images.

Keywords: ALGORITHM, SYNTHESIS, ANALYSIS, CONVOLUTIONAL NEURAL NETWORKS.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

CNN, ЗНМ – згорткові нейронні мережі;

ШНМ – штучні нейронні мережі;

MDP – Марковський процес прийняття рішень;

GPU – графічний процесор;

БО – Байєсова оптимізація;

GA – генетичний алгоритм;

API – прикладний програмний інтерфейс;

CUDA – програмно-апаратна архітектура паралельних обчислень;

МОВ – метод опорних векторів;

NAS – пошук нейронної архітектури;

DAG – ациклічний графік;

DDR – подвійна швидкість передачі даних;

GDDR – графічна подвійна швидкість передачі даних;

AWS – Amazon Web Service.

ВСТУП

Актуальність теми. Випускна кваліфікаційна робота виконана згідно вимог [1,2] і присвячена дослідженню питань пов’язаних з синтезом структур згорткових нейронних мереж. Значна популярність згорткових нейронних мереж для задач класифікації зображень та ріст складності обчислень вимагає автоматизації процесу синтезу структур ЗНМ для більш точного, подальшого, опрацювання інформації, отриманої з їх допомогою.

Згорткова нейронна мережа, CHC, CNN – основний інструмент для класифікації та розпізнавання об'єктів, облич на фотографіях, розпізнавання мови. Є безліч варіантів застосування CNN, такі як Deep Convolutional Neural Network (DCNN), Region-CNN (R-CNN), Fully Convolutional Neural Networks (FCNN), Mask R-CNN та інші.

Першим і, по суті, найбільш тривіальним завданням, яке навчилися вирішувати за допомогою нейронних мереж, стала класифікація зображень.

Згорткові нейронні мережі можуть швидко працювати на послідовній машині і швидко навчатися за рахунок чистого розпаралелювання процесу згортки по кожній карті, а також за рахунок зворотної згортки при поширенні помилки по мережі [3].

Класифікації за допомогою CNN активно застосовуються в медицині: можна навчити нейронну мережу класифікації хвороб або симптомів, наприклад, для МРТ-діагностики.

В агробізнесі розробляється і впроваджується методика аналізу та розпізнавання зображень, при якій дані отримують від відкритих супутників, таких як LSAT, і використовують для прогнозування майбутньої врожайності конкретних земель.

Розпізнавання об'єктів на фото і відео за допомогою нейронних мереж застосовується в безпілотному транспорті, відеоспостереженні, системах контролю доступу, системах "розумного будинку" і так далі.

Розпізнавання облич означає можливість виділяти обличчя на зображеннях. А потім, за допомогою нейромереж CNN, розпізнавати обличчя конкретної людини.

Також нейронні мережі можна використовувати для виділення людей або окремих частин тіла людини на фото або відео, для побудови їхніх скелетів, поз. Такий підхід застосовується, наприклад, для відеоаналітики.

На даний момент існує кілька конкуруючих моделей, що дозволяють отримати тривимірні моделі особи (3DMM) всього по одній фотографії. Крім реконструкції облич згорткові мережі застосовують також для реконструкції інших тривимірних об'єктів по фото.

Згорткові нейронні мережі можна застосовувати не тільки для вирішення завдань комп'ютерного зору. Наприклад, недавно Facebook AI Research виклала у відкритий доступ wav2letter ++ – свою технологію розпізнавання мови, засновану на CNN.

Також існують статті та дослідження фахівців AI-сфери, в яких пропонується застосовувати CNN для аналізу емоційної тональності тексту, тобто області, взагалі далекої від сигналів і зображень.

Мета і завдання дослідження. Метою роботи є програмна реалізація автоматизованого синтезу структур згорткових нейронних мереж для класифікації біомедичних зображень.

Об’єкт дослідження – процес синтезу структур ЗНМ.

Предмет дослідження – алгоритми синтезу структур згорткових нейронних мереж.

Для досягнення поставленої мети необхідно розв’язати такі задачі:

– проаналізувати структури згорткових нейронних мереж;

– проаналізувати засоби та технології синтезу структур згорткових нейронних мереж;

– проаналізувати методи та алгоритми синтезу структур ЗНМ;

– програмно реалізувати автоматизований синтез структур ЗНМ на основі проаналізованих алгоритмів;

– провести тестування реалізованих алгоритмів.

Методи досліджень базуються на використанні методів комп’ютерного зору, методів аналізу зображень та методів синтезу структур ЗНМ.

Наукова новизна одержаних результатів. Розроблено генератор структур згорткових нейронних мереж для класифікації біомедичних зображень, що дозволило синтезувати структури ЗНМ з мінімальним втручанням користувача.

Практичне значення отриманих результатів. Розроблено програмне забезпечення, що автоматизує синтез структур згорткових нейронних мереж для подальшої класифікації біомедичних зображень і потребує мінімального втручання користувача. Експериментально доведена ефективність запропонованого програмного продукту.

Публікації та апробація випускної кваліфікаційної роботи. Отримані результати апробовані в межах міжнародної науково практичної конференції «Problèmes et perspectives d'introduction de la recherche scientifique innovante» та в журналі «Journal of Machine Learning». Опубліковано тези доповіді та статті по темі роботи [4,5,6,7,8,9].

Впровадження результатів ВКР. Результати роботи використані в госпдоговірній науково-дослідній роботі на тему «Нейромережеві методи і засоби класифікації зображень ауто- та ксеногенних тканин» (державний реєстраційний номер 0119U103227) (додаток А).

Випускна кваліфікаційна робота складається із трьох розділів, висновків, списку використаної літератури та додатків.

У першому розділі на основі публікацій [10,11,12] проведено аналіз засобів та технологій синтезу структур ЗНМ, також проаналізовано бібліотеки для досягнення згаданих цілей.

У другому розділі проаналізовано та описано алгоритми синтезу структур згорткових нейронних мереж.

Третій розділ містить опис та тестування програмної реалізації завдання.

1 АНАЛІЗ ЗАСОБІВ ТА ТЕХНОЛОГІЙ СИНТЕЗУ СТРУКТУР ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ

1.1 Аналіз структур згорткових нейронних мереж

Згорткова нейронна мережа (ЗНМ) складається з шарів входу та виходу, також має кілька прихованих шарів. Приховані шари зазвичай складаються зі згорткових, агрегувальних, повноз'єднаних шарів та шарів нормалізації.

Цей процес описують в нейронних мережах як згортку. З математичної точки зору він є швидше взаємною кореляцією, ніж згорткою [13].

Топологія згорткової нейронної мережі представлена на рисунку 1.1.

Рисунок 1.1 – Структура ЗНМ

Її детальніший варіант представлений на рисунку 1.2. Перші два типи шарів, згортковий та субдискретизований, чергуючись між собою, формують вхідний вектор ознак для багатошарового перцептрона [3].

Згорткові шари застосовують для вхідних даних, передаючи результат до наступного шару.

Кожен згортковий нейрон обробляє дані лише для свого рецептивного поля.

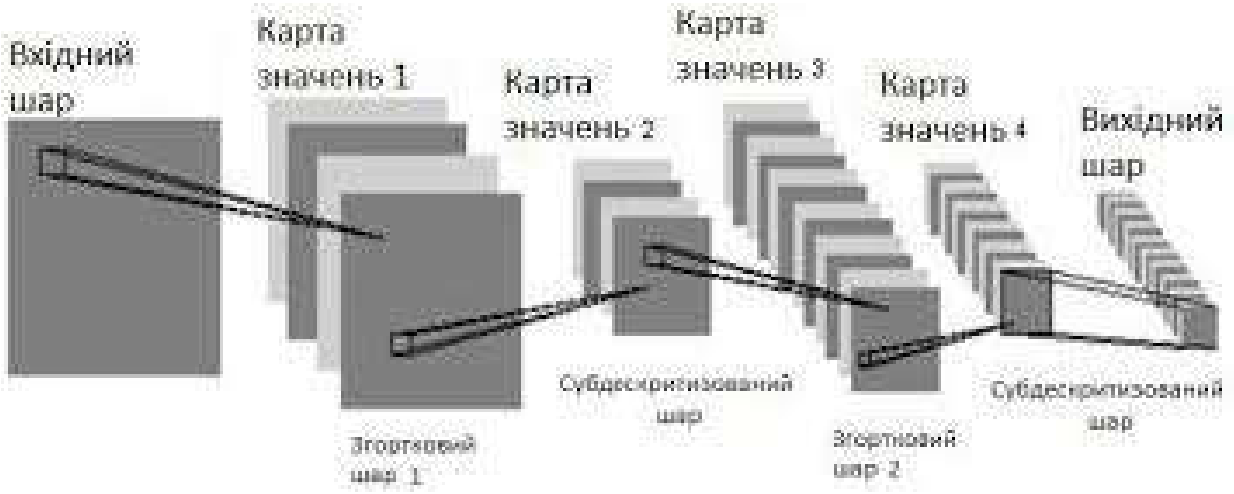

Рисунок 1.2 –Детальніша структура ЗНМ

Повноз'єднані нейронні мережі прямого поширення можна застосовувати як для навчання ознак, так і для класифікування даних, але застосування цієї архітектури до зображень є непрактичним. Наприклад, повноз'єднаний шар для (маленького) зображення розміром 100 × 100 має 10 000 ваг. Операція згортки дає змогу розв'язати цю проблему, оскільки вона зменшує кількість вільних параметрів, дозволяючи мережі бути глибшою за меншої кількості параметрів. Наприклад, незалежно від розміру зображення, області розміру 5 × 5, кожна з одними й тими ж спільними вагами, вимагають лише 25 вільних параметрів. Таким чином, це розв'язує проблему зникання або вибуху градієнтів у тренуванні традиційних багатошарових нейронних мереж з багатьма шарами за допомогою зворотного поширення.

Згортковий шар являє собою набір карт (інша назва – карти ознак або матриці), у кожної карти є синаптичне ядро (в різних джерелах його називають по-різному: скануюче ядро або фільтр). Кількість карт визначається вимогами до задачі, якщо взяти велику кількість карт, то підвищиться якість розпізнавання, але збільшюється обчислювальна складність. Виходячи з аналізу наукових статей, в більшості випадків пропонується брати співвідношення один до двох, тобто кожна карта попереднього шару (наприклад, у першого згорткового шару, попереднім є вхідний) пов'язана з двома картами згорткового шару. Приклад наведено на рисунку 1.3.

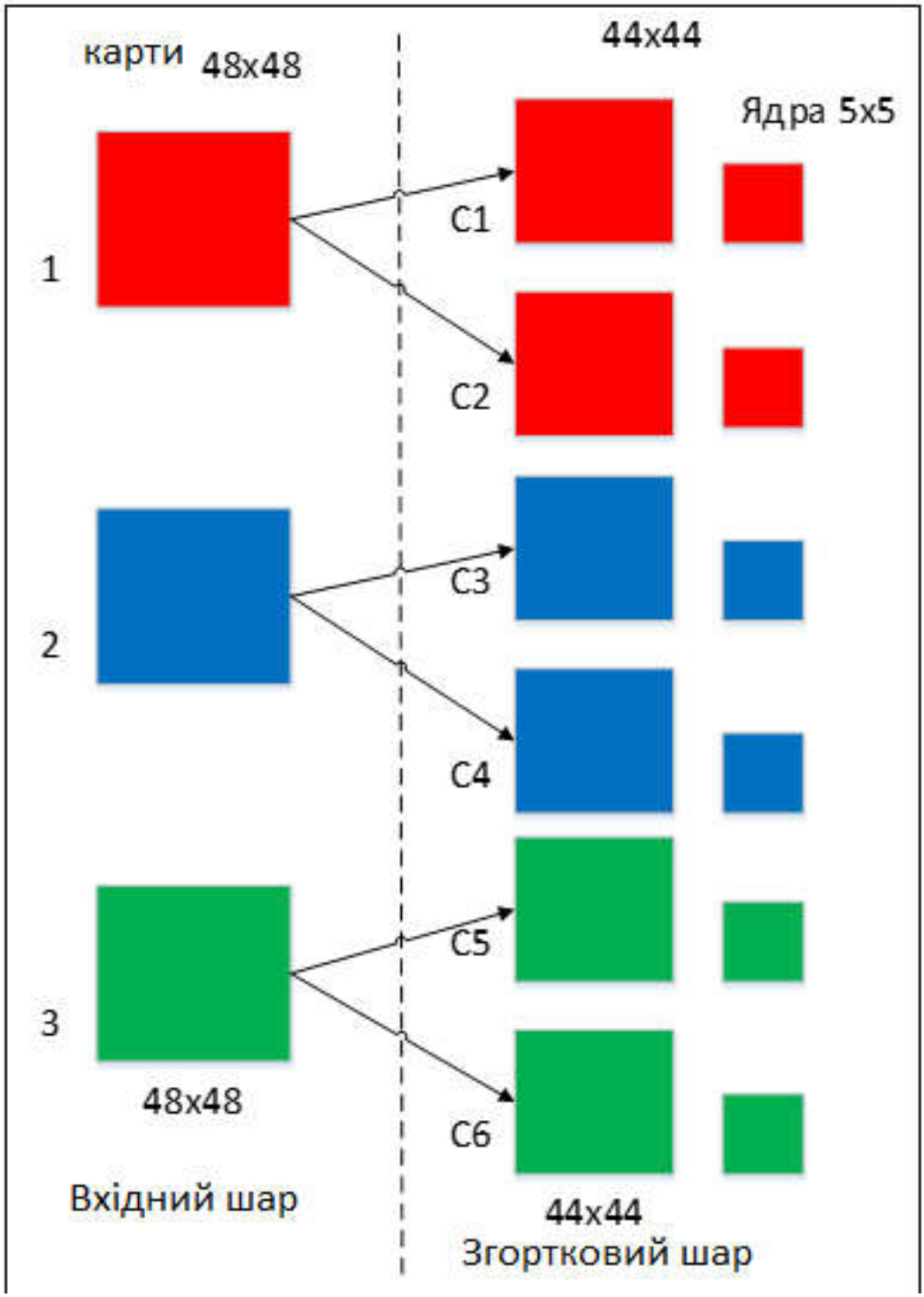

Рисунок 1.3 – Організація зв’язків між картами згорткового шару та попереднього

Розміри у всіх карт згорткового шару однакові і обчислюються за формулою (1.1).

$$(w,h) = (mW - kW + 1, mH - kH + 1), \quad (1.1)$$

де (w,h) – обчислюваний розмір згорткової карти,

mW – ширина попередньої карти,

kW – ширина ядра,

mH – висота попередньої карти,

kH – висота ядра.

Ядро являє собою фільтр або вікно, яке «ковзає» по всій області карти і знаходить певні ознаки об'єктів. Наприклад, якщо мережу навчали на безлічі осіб, то одне з ядер в процесі навчання може видавати найбільший сигнал в області очей, рота, інше ядро може виявляти інші ознаки. Розмір ядра зазвичай беруть в межах від 3x3 до 7x7. Якщо розмір ядра маленький, то воно не зможе виділити будь-які ознаки, якщо занадто велике, то збільшується кількість зв'язків між нейронами. Також розмір ядра вибирається таким, щоб розмір карт згорткового шару був парним, це дозволяє не втрачати інформацію при зменшенні розмірності в підвибірному шарі.

Ядро являє собою систему поділюваних ваг або синапсів, це одна з головних особливостей згорткової нейронної мережі. У звичайній багатошаровій мережі дуже багато зв'язків між нейронами, тобто синапсів, що вельми уповільнює процес детектування. У згорковій нейронній мережі – навпаки, загальні ваги дозволяє скоротити число зв'язків і дозволити знаходити одну і ту саму ознаку по всій області зображення [3]. Тут застосовується операціє згортки, яка обчислюється за формулою (1.2).

$$(f \times g)[m,n] = \sum_{k.l} f[m-k, m-l] \times g[k,l], \quad (1.2)$$

де f – вихідна матриця зображення,

g – ядро згортки.

Згортка – це операція обчислення нового значення обраного пікселя, що враховує значення оточуючих його пікселів. Для обчислення значення використовується матриця, що називається ядром згортки або матрицею згортки. Зазвичай ядро згортки є квадратною матрицею $n \times n$, де n – непарне число. Під час обчислення нового значення обраного пікселя ядро згортки «прикладається» своїм центром до даного пікселя. Навколишні пікселі так само накриваються ядром. Далі вираховується сума, де складовими є помножені значення пікселів на значення комірки ядра, що накрила даний піксель. Сума ділиться на коефіцієнт нормування (div) або на суму всіх елементів ядра згортки. Отримане значення якраз і є новим значенням обраного пікселя. Якщо застосувати згортку до кожного пікселя зображення, то в результаті вийде певний ефект, що залежить від обраного ядра згортки.

Операція згортки представлена на рисунку 1.4.

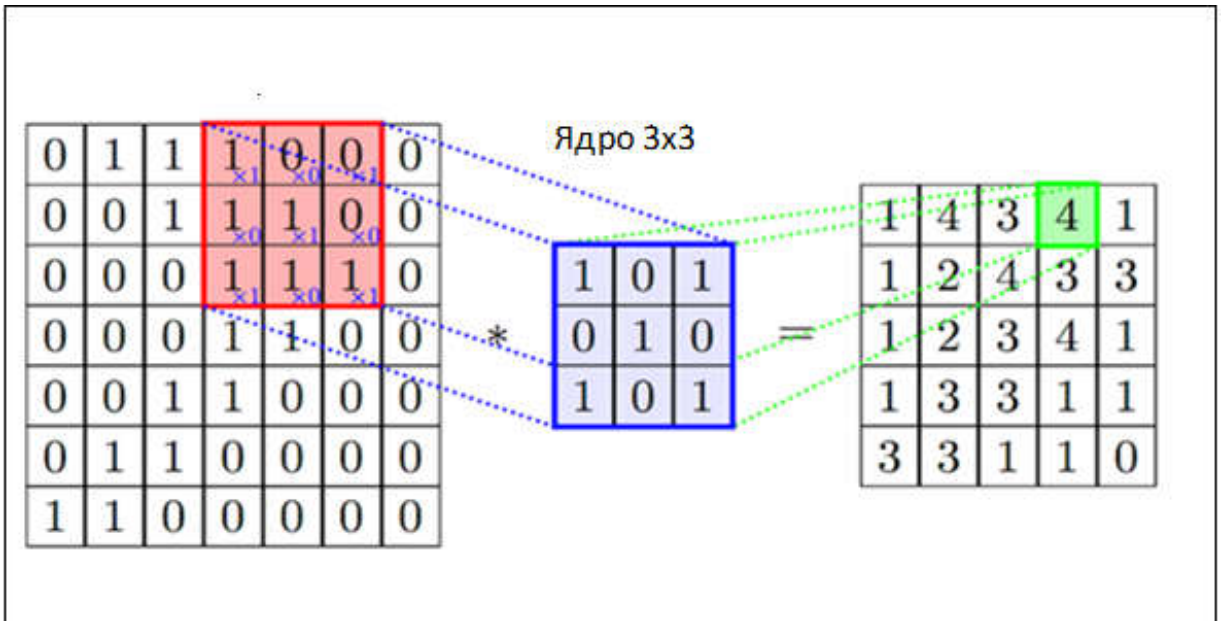

Рисунок 1.4 – Операція згортки

Іншим важливим поняттям ЗНМ є агрегування (субдискретизація), яке є різновидом нелінійного зниження дискретизації. Існує декілька нелінійних функцій для реалізації агрегування, серед яких найпоширенішою є максимізаційне агрегування (max pooling). Воно розділяє вхідне зображення на

набір прямокутників без перекриттів, і для кожної такої підобласті виводить її максимум. Ідея полягає в тому, що точне положення ознаки не так важливе, як її грубе положення відносно інших ознак. Агрегувальний шар слугує поступовому скороченню просторового розміру представлення для зменшення кількості параметрів та об'єму обчислень у мережі, і відтак також для контролю перенавчання. В архітектурі ЗНМ є звичним періодично вставляти агрегувальний шар між послідовними згортковими шарами. Операція агрегування забезпечує ще один різновид інваріантності відносно паралельного перенесення.

Агрегувальний шар діє незалежно на кожен зріз глибини входу, і зменшує його просторовий розмір. Найпоширенішим видом є агрегувальний шар із фільтрами розміру 2×2, що застосовуються з кроком 2, який знижує дискретизацію кожного зрізу глибини входу в 2 рази як за шириною, так і за висотою, відкидаючи 75 % збуджень. В цьому випадку кожна операція взяття максимуму діє над 4 числами. Розмір за глибиною залишається незмінним.

На додачу до максимізаційного агрегування, агрегувальні вузли можуть використовувати й інші функції, такі як усереднювальне агрегування (average pooling) та L^2-нормове агрегування. Історично усереднювальне агрегування застосовувалася часто, але останнім часом втратило популярність у порівнянні з дією максимізаційного агрегування, робота якого на практиці виявилася кращою. Через агресивне скорочення розміру представлення, тенденція йде до менших фільтрів, або відмови від агрегувального шару взагалі.

Субдискретизований шар також, як і згортковий має карти. Мета субдискретизованого шару – зменшення розмірності карт попереднього шару. Якщо на попередній операції згортки вже були виявлені деякі ознаки, то для подальшої обробки настільки докладне зображення вже не потрібно, і воно ущільнюється до менш докладного. До того ж, фільтрація вже непотрібних деталей допомагає не перенавчатися.

У процесі сканування ядром субдискретизованого шару (фільтром) карти попереднього шару, скануюче ядро не перетинається на відміну від згорткового

шару. Зазвичай, кожна карта має ядро розміром 2x2, що дозволяє зменшити попередні карти згорткового шару в 2 рази. Вся карта ознак поділяється на осередки 2×2 елементи, з яких вибираються максимальні за значенням.

Зазвичай, в субдискретизованому шарі застосовується функція активації ReLU (шар зрізаних лінійних вузлів). ReLU є абревіатурою від англ. Rectified Linear Units. Цей шар застосовує ненасичувальну передавальну функцію $f(x) = \max(0, x)$. Він посилює нелінійні властивості функції ухвалення рішення і мережі в цілому, не зачіпаючи рецептивних полів згорткового шару. Для посилення нелінійності застосовуються й інші функції, наприклад, насичувальні гіперболічний тангенс $f(x) = \tanh(x)$, $f(x) = |\tanh(x)|$ та сигмоїдна функція $f(x) = (1 + e^{-x})^{-1}$. Зрізаному лінійному вузлові ReLU часто віддають перевагу перед іншими функціями, оскільки він тренує нейронну мережу в декілька разів швидше без значної розплати точністю узагальнення.

Формування нової карти субдискретизованого шару на основі попередньої карти згорткового шару приведено на рисунку 1.5.

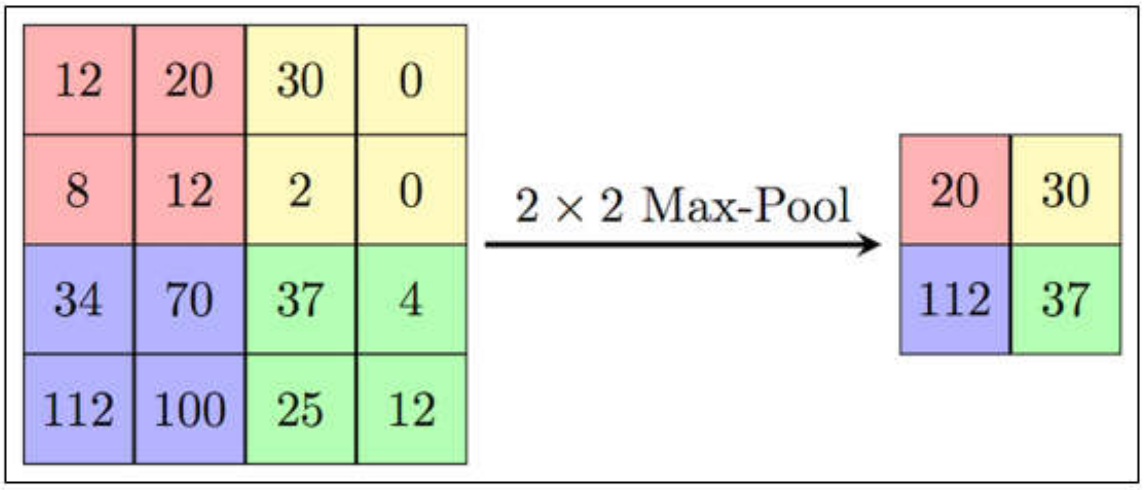

Рисунок 1.5 – Операція підвибірки (Max Pooling)

Субдискретизований шар можна описати формулою (1.3).

$$x^l = f(a^l \times subsample(x^{l-1}) + b^l), \tag{1.3}$$

де x^l – вихід шару l,

$f()$ – функція активації,

a^l, b^l – коефіцієнти зсуву шару l,

subsample() – операція вибірки локальних максимальних значень.

Останній з типів шарів – це шар звичайного багатошарового перцептрона, або, як його ще називають – повнозв'язний шар. Мета шару – класифікація.

Повнозв'язний шар моделює складну нелінійну функцію, оптимізуючи яку, покращується якість розпізнавання.

Нейрони кожної карти попереднього субдискретизованого шару пов'язані з одним нейроном прихованого шару. Таким чином, число нейронів прихованого шару дорівнює числу карт субдискретизованого шару, але зв'язки можуть бути не обов'язково такими, наприклад, тільки частина нейронів будь-якої з карт субдискретизованого шару може бути пов'язана з першим нейроном прихованого шару, а частина, що залишилася з другим, або всі нейрони першої карти пов'язані з нейронами 1 і 2 прихованого шару. Обчислення значень нейрона можна описати формулою (1.4).

$$x_j^l = f(\sum_i x_i^{l-1} \times w_{i,j}^{l-1} + b_j^{l-1}), \tag{1.4}$$

де x_j^l – карта ознак j (вихід шару l),

$f()$ – функція активації,

b^l – коефіцієнт зсуву шару l,

$w_{i,j}^l$ – матриця вагових коефіцієнтів шару l.

В наш час існує багато архітектур згорткових нейронних мереж. Найпопулярнішими можна назвати такі:

– LeNet. Перше успішне застосування згорткової нейронної мережі вдалося розробити Яну Лекуну в 1990-і роки. Архітектура LeNet застосовувалася для зчитування поштових індексів, цифр і т. д.

– AlexNet. Це робота Алекса Крижевського, Іллі Суцкевера і Джеффа Хинтона, яка зіграла істотну роль в популяризації ЗНМ в області комп'ютерного зору. Архітектура AlexNet була представлена на ImageNet ILSVRC Challenge в 2012 році і обійшла всі роботи конкурентів (16% помилок проти 26% у архітектури, яка зайняла друге місце).

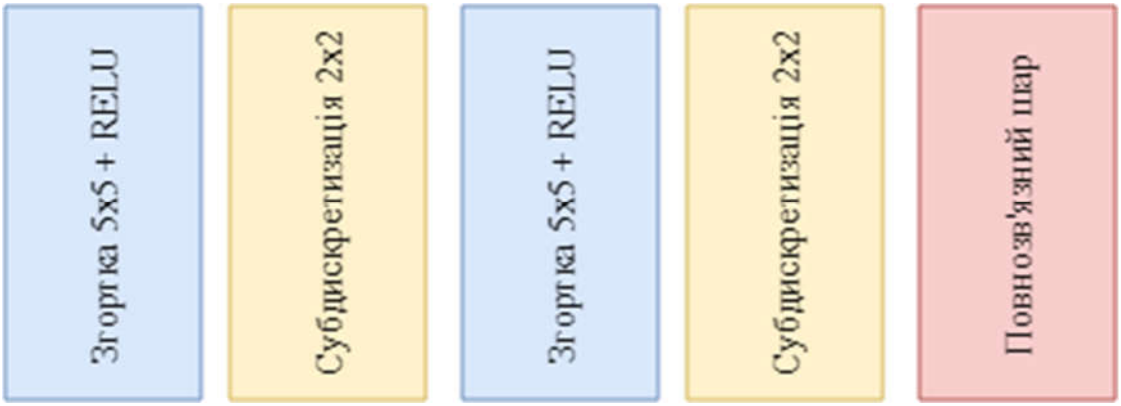

Рисунок 1.6 – Архітектура LeNet

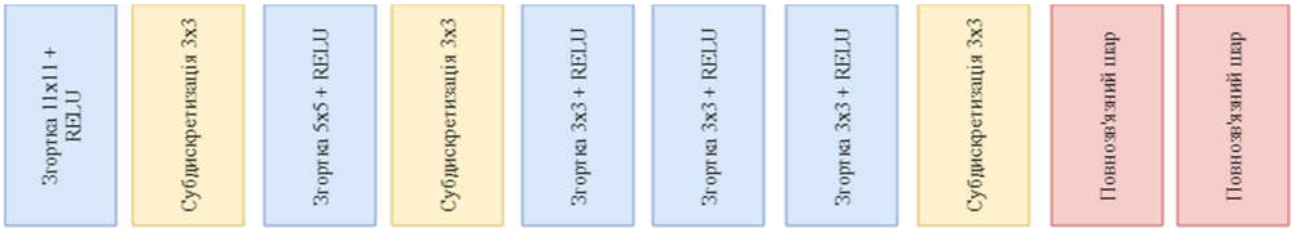

Рисунок 1.7 – Архітектура AlexNet

– ZF Net. А ось переможцем ILSVRC 2013 стала згорткова нейронна мережа Метью Зеллера і Роба Фергюса, яка відома як ZF Net (абревіатура від Zeiler і Fergus). Дана архітектура була поліпшеною версією AlexNex: тут збільшили розміри середніх згорткових шарів і зменшили крок і розмір фільтра на першому шарі.

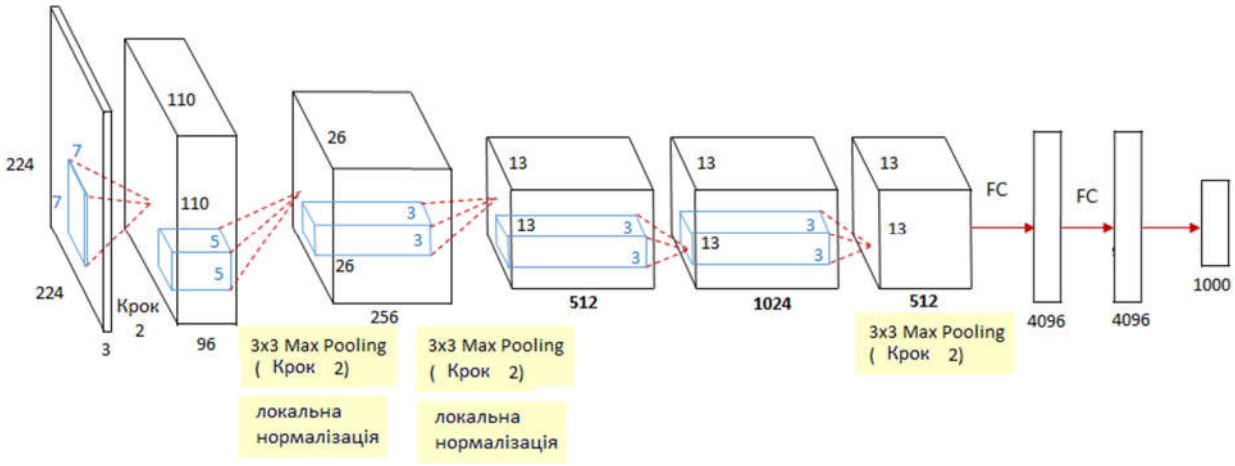

Рисунок 1.8 – Архітектура ZF Net

– GoogLeNet. У 2014 році вищезгаданий конкурс виграла ЗНМ розробки Шегеда і інших - співробітників корпорації Google. Основна заслуга даної архітектури полягає в розробці та впровадженні вхідного модуля (Inception Module), що дозволило різко скоротити число параметрів до 4 млн з 60 млн. Скорочення параметрів відбувається також завдяки заміні повнозв'язних шарів у верхній частині мережі шарами середнього пулінгу.

– VGGNet. Відразу за GoogLeNet на ILSVRC 2014 перемогла мережа Карена Симоняна і Ендрю Ціссермана, яка стала відома як VGGNet. Розробникам вдалося наочно продемонструвати, що глибина є ключовим фактором для продуктивності. Їх мережа містить 16 згорткових і повнозв'язних шарів і має надзвичайно однорідну архітектуру, яка виконує згортку 3x3 і пулінг 2x2 від початку до кінця. Вихідна модель доступна в режимі Plug and Play у фреймворку для глибокого навчання Caffe. Недоліком VGGNet є те, що потрібно оцінювати і використовувати набагато більше пам'яті і параметрів (140M). Більшість цих параметрів знаходяться в першому повнозв’язному шарі, і з тих пір було виявлено, що ці FC-шари можуть бути видалені без зниження продуктивності, значно зменшуючи кількість необхідних параметрів.

Рисунок 1.9 – Архітектура VGGNet

– ResNet. Залишкова мережа (Residual Network), розроблена Каймінгом Хе і іншими, стала переможцем ILSVRC 2015. Ключові особливості - інтенсивне використання пакетної нормалізації і спеціальні скіп-з'єднання. В кінці архітектури відсутні повнозв'язні шари. ResNet станом на сьогоднішній день є справжнім витвором мистецтва в світі згорткових нейронних мереж і використовується найчастіше.

Найбільші перешкоди при організації ЗНМ може чинити пам'ять. Сучасні графічні процесори мають ліміт пам'яті в 3, 4 або 6 Гб, в той час як кращі GPU мають цілих 12 Гб. Є 3 основних джерела пам'яті, за якими обов'язково потрібно стежити:

– кількість активацій на кожному шарі нейронної мережі;

– кількість параметрів нейронної мережі;

– будь-яка реалізація згорткової нейронної мережі повинна підтримувати різнорідну пам'ять: пакет даних зображень, можливо, їх вдосконалені реалізації.

1.2 Аналіз методів та алгоритмів синтезу структур згорткових нейронних мереж

Машинне навчання дало можливість досягти значного прогресу за останні роки в різних завданнях, наприклад розпізнавання зображень, розпізнавання мови та машинного перекладу. Важливий аспектом для цього прогресу є нові нейронні архітектури. В даний час працюючі архітектури здебільшого розроблялися вручну фахівцями-людьми, що є трудомістким та помилковим

процесом. Через це зростає інтерес до методів пошуку автоматизованої нейронної архітектури [14].

У праці [15] описано процес автоматизації вибору архітектури згорткової нейронної мережі за допомогою процедури метамоделювання, заснованої на основі навчання з підсиленням – MetaQNN. Автори розробили новий агент Q-навчання, мета якого виявляти архітектури згорткових нейронних мереж, що виконують задачу машинного навчання без втручання людини. Після дискретизації та обмеження параметрів вибору, розроблений навчальний агент залишається обмеженим, але має великий простір модельних архітектур для пошуку. Агент вчиться випадковим чином і повільно починає використовувати свої висновки для вибору високоефективних моделей.

Завданням навчання агента є знайти оптимальні шляхи як Марковський процес прийняття рішень (MDP) у середовищі з кінцевим горизонтом. Обмеження середовища кінцевим горизонтом гарантує, що агент детерміновано припиниться з обмеженою кількістю часових кроків. Крім того, автори роботи обмежують середовище, щоб мати дискретний та кінцевий простір, а також простір дій.

Завданням тренування навчального агента автори вважають навчання послідовно вибирати шари нейронної мережі.

Вони моделюють процес вибору шару як Марковський процес прийняття рішення з припущенням, що якісний рівень в одній мережі також повинен добре працювати в іншій мережі. Ми робимо це припущення, виходячи з ієрархічної природи деяких спроб навчання нейронних мереж з багатьма прихованими шарами.

Агент послідовно вибирає шари за допомогою стратегії greedy до досягнення стану завершення.

Архітектура CNN, визначена шляхом агента, навчається на обраній навчальній проблемі, агент отримує винагороду, рівну точності перевірки. Точність перевірки та опис архітектури зберігається в пам’яті, що повторюється,

а періодичні вибірки періодично відбираються з пам'яті для оновлення Q-значень.

Описаний метод вимагає трьох основних варіантів проектування:

– зведення визначень рівня CNN до простого стану;

– кортежі, визначаючи набір дій, які може здійснити агент, тобто набір шарів, який агент може вибрати наступним з урахуванням його поточного стану;

– врівноваження розміру простору дії-стану, і відповідно ємність моделі із кількістю потрібної розвідки.

Методи автоматичного проектування глибоких нейромережевих архітектур, такі як підходи, засновані на навчанні з підкріпленням, показують перспективні результати. Однак, їхній успіх базується на величезних обчислювальних ресурсах (наприклад сотні графічних процесорів), що ускладнює їх широке використання.

Помітним обмеженням є те, що вони все одно розробляють та навчають кожну мережа з нуля під час дослідження архітектурного простору, що є дуже неефективним. У роботі [16] автори пропонують нову основу для ефективного пошуку архітектури на основі поточної мережі і повторного використання ваг. В якості метаконтролера використовується навчальний агент для посилення, дія якого полягає у зростанні глибини мережі або ширини шару з перетвореннями, що зберігають функцію. Таким чином, раніше перевірені мережі можуть підлягати повторному використанню для подальшої розвідки, тим самим економиться велика кількість обчислювальної вартості. Автори застосовують свій метод для дослідження архітектурного простору рівнинних згорткових нейронних мереж (відсутність пропускних з'єднань, розгалуження тощо) на еталонному наборі даних (CIFAR-10, SVHN) з обмеженими обчислювальними ресурси (5 GPU).

У роботі [17] автори пропонують ефективний пошук нейронної архітектури (ENAS), швидкий і недорогий підхід до автоматичного проектування моделі. У системі ENAS контролер виявляє архітектури нейронної мережі шляхом пошуку для оптимального підграфу у великому

обчислювальному графіку. Контролер навчається з градієнтом політики, щоб вибрати підграф, який максимізує очікувана винагорода на валідаційному наборі. Обмін параметрами серед дочірніх моделей дозволяють ENAS забезпечити потужні емпіричні показники, використовуючи при цьому набагато менше GPU годин, ніж існуючі підходи до автоматичного проектування моделі, і, зокрема, в 1000 разів дешевше, ніж стандартний пошук по нейронній архітектурі.

Центральним у ідеї ENAS є спостереження, що всі з графіків, над якими може закінчуватися ітерація NAS, розглядати як підграфіки більшого графіка.

Байєсова оптимізація (БО) відноситься до класу методів глобальної оптимізації. Загальний випадок використання БО в машинному навчання – це вибір моделей, де неможливо аналітично моделювати результати узагальнення статистичної моделі.

Вибір моделі дозволяє лише налаштувати скалярні гіперпараметри алгоритмів машинного навчання. Однак зі сплеском інтересу до глибокого навчання виникає зростаючий попит на налаштування архітектур нейронної мережі. У праці [18] автори розвивають NASBOT, Гауссову базову технологічну основу для пошуку нейронної архітектури.

Для цього автори розробили метрику відстані в просторі архітектури нейронної мережі, які можна ефективно обчислити за допомогою оптимальної транспортної програми.

У роботі продемонстровано, що NASBOT переважає інші альтернативи пошуку архітектури в декількох моделях на основі перехресної перевірки завдання вибору на багатошарових перцептронах та згорткових нейронних мережах.

За останні роки згорткові нейронні мережі (CNN) отримали надзвичайний успіх у багатьох проблемах реального світу.

Однак продуктивність CNN дуже покладається на їх архітектуру.

У праці [19] автори пропонують метод автоматичного проектування архітектури CNN за допомогою генетичних алгоритмів, який здатний виявити

перспективну архітектуру CNN при виконанні завдання класифікації зображень. Запропонований алгоритм не потребує попередньої обробки перед його роботою, а також будь-якої післяобробки, а це означає, що він повністю автоматичний. Запропонований алгоритм затверджений на основі широко використовуваних наборів даних бенчмарків порівнюючи з найсучаснішими конкурентами на рівні однолітків, що охоплюють вісім CNN-мереж, створених вручну, чотири напівавтоматичні CNN і додаткові чотири автоматично розроблені CNN. Результати експериментів свідчать, що запропонований алгоритм досягає найкращої точності класифікації, послідовно серед CNN, що розробляються вручну та автоматично. Крім того, запропонований алгоритм також показує конкурентоспроможність класифікаційної точності напівавтоматичних конкурентів однолітків, зменшуючи параметри в 10 разів.

Також, запропонований алгоритм в середньому займає лише один відсоток обчислювального ресурсу порівняно з рівнем всіх інших алгоритмів пошуку архітектури.

У роботі [20] автори пропонують використовувати генетичний алгоритм (GA) для оптимізації точності згорткової нейронної мережі (CNN). GA змінює структуру CNN, наприклад, кількість згорткови фільтрів, кроки, розмір ядра, вузли, параметри навчання тощо. Кожна модифікація мережі проходить навчання та оцінку. Мутація розвинених мереж створює більш успішні мережі протягом декількох поколінь. Кінцева розвинута мережа на 4,77% точніша, ніж мережа, запропонована в попередній літературі. Крім того, розвинена мережа на 13,4% є менш складною в обчисленнях.

1.3 Аналіз бібліотек для синтезу структур згорткових нейронних мереж

В наш час найбільш популярними бібліотеками для побудови архітектури нейронних мереж, задачі синтезу структур та оптимізації гіперпараметрів ЗНМ є

Keras та TensorFlow, підмодуль model_selection у бібліотеці sklearn, Hyperopt відповідно.

Keras – відкрита нейромережева бібліотека, написана мовою Python. Вона здатна працювати поверх Deeplearning4j, TensorFlow та Theano. Її було створено як частину дослідницьких зусиль проекту ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), а її основним автором та підтримувачем є Франсуа Шоллє (François Chollet), інженер Google [21].

Нейронніі шари, функції втрат, оптимізатори, схеми ініціалізації, функції активації та схеми регуляризації – все це окремі модулі, які можна комбінувати для створення нових моделей. Нові модулі додаються просто, як нові класи та функції. Моделі визначені в коді Python, а не в окремих файлах конфігурації моделі.

Keras було задумано скоріше як інтерфейс, ніж як наскрізну систему машинного навчання. Вона представляє високорівневий, інтуїтивніший набір абстракцій, який робить простим формування нейронних мереж незалежно від тилової бібліотеки наукових обчислень. Microsoft працює над доданням до Keras і тилу CNTK.

Keras містить численні реалізації широко вживаних будівельних блоків нейронних мереж, таких як шари, цільові та передавальні функції, оптимізатори, та безліч інструментів для спрощення роботи із зображеннями та текстом.

Keras належним чином не робить власних операцій низького рівня, таких як тензорні вироби та згортки; для цього він покладається на бекенд. Незважаючи на те, що Keras підтримує кілька двигунів заднього рівня, його основним (і типовим) тилом є TensorFlow, а основним його прихильником є Google. API Keras поставляється упакованим у TensorFlow як tf.keras [22].

Класи KerasClassifier та KerasRegressor в Keras беруть аргумент build_fn, який називає функцію, яку потрібно викликати для отримання вашої моделі.

Ви повинні визначити функцію, яка визначає вашу модель, компілює її та повертає.

У прикладі нижче (рисунок 1.9) ми визначаємо функцію create_model (), яка створює просту багатошарову нейронну мережу.

Ми передаємо цю назву функції класу KerasClassifier аргументом build_fn. Ми також передаємо додаткові аргументи nb_epoch = 150 і batch_size = 10. Вони автоматично вбудовуються і передаються функції fit (), яка викликається внутрішньо класом KerasClassifier [23].

У цьому прикладі ми використовуємо scikit-learn StratifKFold для виконання 10-кратної стратифікованої перехресної перевірки. Це техніка перекомпонування, яка може забезпечити надійну оцінку ефективності моделі машинного навчання за небаченими даними.

Ми використовуємо функцію scikit-learn cross_val_score () для оцінки нашої моделі за допомогою схеми перехресної перевірки та друку результатів.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import numpy

# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
# evaluate using 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Рисунок 1.10 – Приклад використання scikit-learn

Запуск прикладу відображає майстерність моделі для кожної епохи. Всього створено та оцінено 10 моделей та відображається кінцева середня точність.

В описаному випадку вона становить 0.646838691487.

TensorFlow – відкрита програмна бібліотека для машинного навчання в цілій низці задач, розроблена компанією Google для задоволення її потреб у системах, здатних будувати та тренувати нейронні мережі для виявлення та розшифровування образів та кореляцій, аналогічно до навчання й розуміння, які застосовують люди (рисунок 1.11). TensorFlow є системою машинного навчання Google Brain другого покоління, випущеною як відкрите програмне забезпечення 9 листопада 2015 року. В той час як еталонна реалізація працює на одиничних пристроях, TensorFlow може працювати на декількох центральних та графічних процесорах (включно з додатковими розширеннями CUDA для обчислень загального призначення на графічних процесорах).

TensorFlow доступна для 64-розрядних Linux, macOS, Windows, та для мобільних обчислювальних платформ, включно з Android та iOS.

Обчислення TensorFlow виражаються як станові графи потоків даних. В червні 2016 року Джефф Дін з Google заявив, що TensorFlow згадували 1 500 репозиторіїв на GitHub, лише 5 з яких були від Google [24].

Архітектура Tensorflow складається з трьох частин:

– попереднє оброблення даних;

– побудова моделі;

– тренування та оцінка моделі.

ЇЇ називають Tensorflow, оскільки вона приймає вхід як багатовимірний масив, також відомий як тензори. З її допомогою можна побудувати своєрідну блок-схему операцій. Вхідні дані надходять в один кінець, а потім проходять через цю систему декількох операцій і виходять з іншого кінця як вихідні дані [25].

Модель може бути навчена та використана як на графічних процесорах, так і на звичайних процесорах. Спочатку графічні процесори були розроблені для

відеоігор. Наприкінці 2010 року дослідники Стенфорда виявили, що GPU також дуже добре виконує матричні операції. Глибоке навчання покладається на безліч множин матриць. TensorFlow дуже швидко обчислює множення матриці, оскільки він написаний на C ++. Незважаючи на те, що він реалізований на C ++, до TensorFlow можна отримати доступ і керувати, головним чином, іншими мовами, зокрема Python.

Важливою особливістю TensorFlow є TensorBoard. TensorBoard дозволяє графічно та візуально відстежувати, що робить TensorFlow [25].

Вибір моделі – це завдання вибору статистичної моделі з набору моделей-кандидатів за даними даних. У найпростіших випадках розглядається попередньо набір даних. Однак завдання може також включати розробку експериментів таким чином, щоб зібрані дані добре відповідали проблемі вибору моделі. З огляду на кандидатські моделі подібної прогнозованої чи пояснювальної сили, найпростіша модель, швидше за все, буде найкращим вибором [26].

Вибір моделі також може стосуватися проблеми вибору декількох репрезентативних моделей з великого набору обчислювальних моделей з метою прийняття рішень або оптимізації в умовах невизначеності [27].

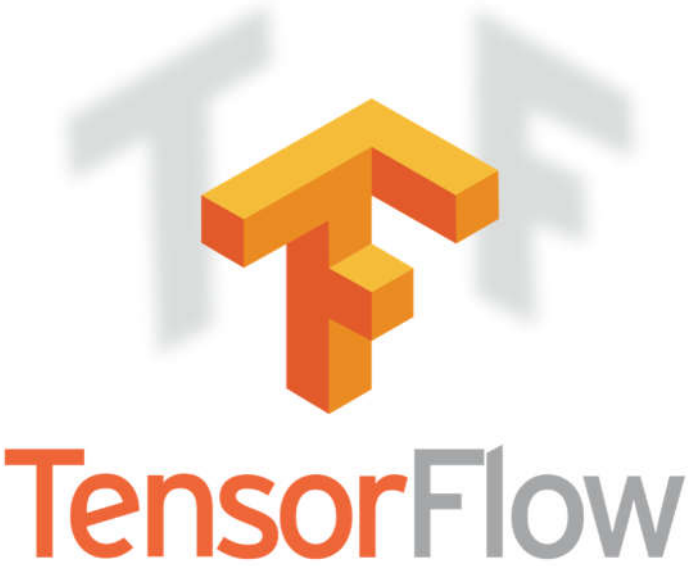

Рисунок 1.11 – Логотип TensorFlow

1.4 Постановка задач дослідження

Метою магістерської роботи є автоматизований синтез структур згорткових нейронних мереж.

Для досягнення поставленої мети необхідно:

– проаналізувати існуючі структури згорткових нейронних мереж;

– проаналізувати методи та алгоритми синтезу структур ЗНМ;

– проаналізувати бібліотеки для синтезу структур згорткових нейронних мереж;

– програмно реалізувати автоматизований синтез структур ЗНМ.

У першому розділі було проведено аналіз засобів та технологій синтезу структур згорткових нейронних мереж. Також було проаналізовано існуючі структури згорткових нейронних мереж. Крім того, було проведено опис та аналіз бібліотек і алгоритмів, що дозволяють виконати поставлені завдання.

Аналіз існуючих алгоритмів синтезу структур згорткових нейронних мереж дав змогу вибрати найкращий та найбільш підходящий алгоритм для досягнення поставленої мети магістерської роботи. Також, у першому розділі було проаналізовано статті, в яких описані існуючі проблеми синтезу структур згорткових нейронних мереж та способи їх вирішення.

2 АЛГОРИТМИ СИНТЕЗУ СТРУКТУР ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ

2.1 Алгоритм повного перебору

Найпопулярнішим методом оптимізації гіперпараметрів є пошук по ґратці, який просто робить повний перебір по заданій вручну підмножині простору гіперпараметрів навчального алгоритму (рисунок 2.1). Пошук по ґратці повинен супроводжуватися деякою мірою продуктивності.

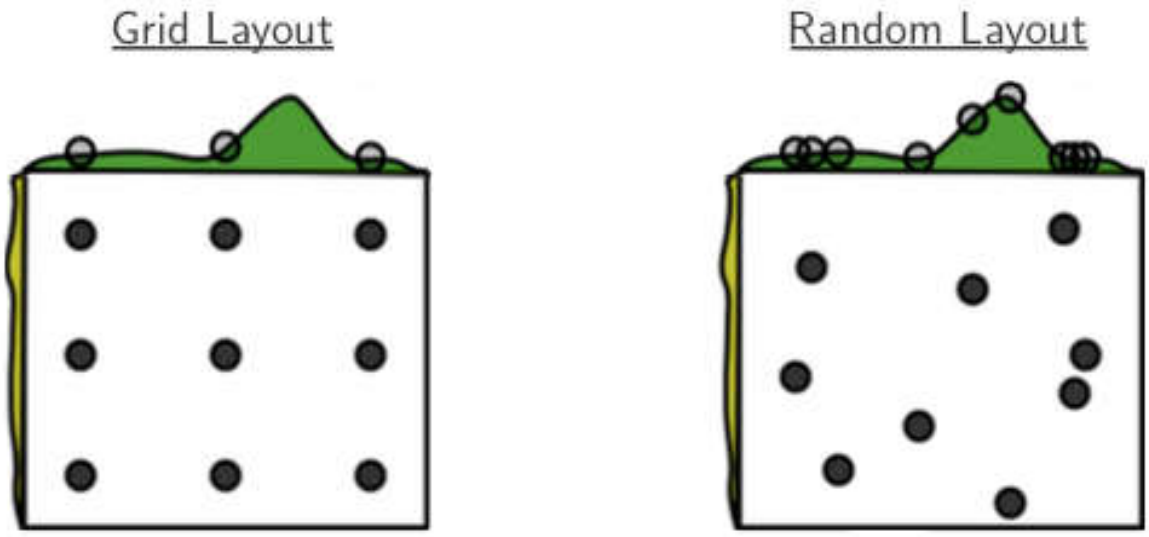

Рисунок 2.1 – Схематичне зображення пошуку по ґратці та рандомного пошуку

Оскільки простір параметрів алгоритму машинного навчання для деяких параметрів може включати простори з дійсними або необмеженими значеннями, тому можлива ситуація, коли необхідно задати границю і дискретизацію до застосування пошуку по ґратці.

Наприклад, типовий класифікатор з не щільним зазором на основі методу опорних векторів (МОВ) та з ядровою радіально-базисною функцією має принаймні два гіперпараметри, які необхідно налаштувати для високої продуктивності на недоступних даних – константа C регуляризації і гіперпараметр ядра γ. Обидва параметри є неперервними, так що для пошуку по

ґратці вибирають скінченну множину «обґрунтованих» значень, скажімо: $C \in \{10,100,1000\}, \gamma \in \{0.1,0.2,0.5,1.0\}$.

Пошук по ґратці проганяє МОВ для кожної пари (C,γ) по декартовому добутку цих двох множин і перевіряє продуктивність на кожній парі вибраних параметрів на фіксованому перевірочному наборі (або за допомогою внутрішньої перехресної перевірки на тренувальному наборі і в цьому випадку кілька МОВ проганяють попарно). Нарешті, алгоритм пошуку по ґратці видає на виході найвищий результат, який було досягнуто на процедурі перевірки.

Пошук по ґратці страждає від прокляття розмірності, але часто легко розпаралелюється, оскільки зазвичай гіперпараметричні величини, з якими алгоритм працює, не залежать одна від одного [28].

Задачі машинного навчання, які передбачають навчання «природному стану» на скінченній кількості зразків даних у просторі властивостей з високим числом вимірів, зазвичай, потребують величезної кількості навчальних даних для того, щоб забезпечити хоча б декілька зразків з різною комбінацією значень. Типове правило полягає в тому, що у кожному вимірі повинно бути щонайменше 5 навчальних прикладів. З фіксованою кількістю навчальних зразків прогностична потужність класифікатора або регресора спочатку збільшується, оскільки кількість використовуваних розмірів/функцій збільшується, але потім зменшується, що відомо, як феномен Хьюза або явище піка [29].

Пошук по ґратці – це модель оптимізації гіперпараметрів.

У scikit-learn ця методика надається в класі GridSearchCV.

При побудові цього класу ви повинні надати словник гіперпараметрів для оцінки в аргументі param_grid. Це карта імені параметра моделі та масиву значень, які слід спробувати.

За замовчуванням точність – це оцінка, яка оптимізована, але інші показники можуть бути визначені в аргументі конструктора GridSearchCV.

За замовчуванням для по ґратці буде використовуватися лише один потік. Встановивши аргумент n_jobs в конструкторі GridSearchCV на -1, процес

використовуватиме всі ядра на вашій машині. Це може перешкоджати основній навчальній роботі з нейронної мережі.

Потім процес GridSearchCV побудує та оцінить одну модель для кожної комбінації параметрів. Перехресне підтвердження використовується для оцінки кожної окремої моделі, а за замовчуванням використовується трикратна перехресна перевірка, хоча це може бути замінено, якщо вказати аргумент CV для конструктора GridSearchCV [30].

Нижче, на рисунку 2.2, наведено приклад визначення простого пошуку по сітці.

```
1 param_grid = dict(epochs=[10,20,30])
2 grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
3 grid_result = grid.fit(X, Y)
```

Рисунок 2.2 – Приклад пошуку по сітці

Після завершення можна отримати доступ до результату пошуку по ґратці в об’єкті результату, поверненому з grid.fit (). Об’єкт best_score_ забезпечує доступ до найкращого результату, який спостерігається під час процедури оптимізації, а best_params_ описує поєднання параметрів, які досягли найкращих результатів.

Докладніше про клас GridSearchCV можна дізнатися в документації API scikit-learn.

Попередній приклад (див. рисунок 1.9) показав, наскільки легко перетворити свою модель глибокого навчання від Keras і використовувати її у функціях з бібліотекою scikit-learn [31].

У цьому прикладі ми йдемо на крок далі. Функція, яку ми задаємо аргументу build_fn при створенні оболонки KerasClassifier, може приймати аргументи. Ми можемо викоритовувати ці аргументи для подальшого налаштування конструкції моделі. Крім того, ми знаємо, що можемо навести аргументи функції fit ().

У цьому прикладі ми використовуємо пошук по ґратці, щоб оцінити різні конфігурації для нашої моделі нейронної мережі та повідомити про комбінацію, яка забезпечує найкраще оцінену ефективність.

Функція create_model () визначена для прийому двох аргументів оптимізатора та init, обидва повинні мати значення за замовчуванням. Це дозволить оцінити ефект використання різних алгоритмів оптимізації та схем ініціалізації ваги для нашої мережі.

Після створення нашої моделі ми визначаємо масиви значень для параметра, який ми хочемо шукати, зокрема:

– оптимізатори пошуку для різних значень ваги;

– ініціалізатори для підготовки мережевих ваг за різними схемами;

– епохи для навчання моделі для різної кількості експозицій до навчального набору даних;

– пакети для зміни кількості проб перед оновленням ваги.

Параметри задаються в словнику і передаються в конфігурацію класу scikit-learn GridSearchCV. Цей клас оцінить версію нашої моделі нейронної мережі для кожної комбінації параметрів ($2 \times 3 \times 3 \times 3$ для комбінацій оптимізаторів, ініціалізацій, епох та пакетів). Кожна комбінація потім оцінюється, використовуючи за замовчуванням 3-кратну стратифіковану перехресну перевірку [32].

Це багато моделей і багато обчислень. Може бути корисно зробити невеликі експерименти з меншою підмножиною даних. Це доречно в цьому випадку через малу мережу та малий набір даних (менше 1000 екземплярів та 9 атрибутів).

Нарешті, відображається продуктивність та комбінація конфігурацій для найкращої моделі з подальшим виконанням усіх комбінацій параметрів (рисунок 2.3).

```
# MLP for Pima Indians Dataset with grid search via sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
import numpy

# Function to create model, required for KerasClassifier
def create_model(optimizer='rmsprop', init='glorot_uniform'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, kernel_initializer=init, activation='relu'))
    model.add(Dense(8, kernel_initializer=init, activation='relu'))
    model.add(Dense(1, kernel_initializer=init, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, verbose=0)
# grid search epochs, batch size and optimizer
optimizers = ['rmsprop', 'adam']
init = ['glorot_uniform', 'normal', 'uniform']
epochs = [50, 100, 150]
batches = [5, 10, 20]
param_grid = dict(optimizer=optimizers, epochs=epochs, batch_size=batches, init=init)
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Рисунок 2.3 – Модель для пошуку по ґратці

На робочій станції, для виконання на процесорі, може знадобитися близько 5 хвилин. Результати нижче (рисунок 2.4).

Ми можемо побачити, що пошук по ґратці, за допомогою єдиної схеми ініціалізації, оптимізатора rmsprop, 150 епох та розміру блоку 5, досяг найкращого бала перехресної перевірки – приблизно 75% у цій проблемі.

Із вищеописаних прикладів можна побачити, що використання scikit-learn для стандартних операцій машинного навчання, таких як оцінювання моделі та оптимізація гіперпараметрів моделі, може заощадити багато часу на реалізацію цих схем самостійно.

```
Best: 0.752604 using {'init': 'uniform', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 150}
0.707031 (0.025315) with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 50}
0.589844 (0.147095) with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 50}
0.701823 (0.006639) with: {'init': 'normal', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 50}
0.714844 (0.019401) with: {'init': 'normal', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 50}
0.718750 (0.016573) with: {'init': 'uniform', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 50}
0.688802 (0.032578) with: {'init': 'uniform', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 50}
0.657552 (0.075566) with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 100}
0.696615 (0.026557) with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 100}
0.727865 (0.022402) with: {'init': 'normal', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 100}
0.736979 (0.030647) with: {'init': 'normal', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 100}
0.739583 (0.029635) with: {'init': 'uniform', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 100}
0.717448 (0.012075) with: {'init': 'uniform', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 100}
0.692708 (0.036690) with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 150}
0.697917 (0.028940) with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 150}
0.727865 (0.030647) with: {'init': 'normal', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 150}
0.747396 (0.016053) with: {'init': 'normal', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 150}
0.729167 (0.007366) with: {'init': 'uniform', 'optimizer': 'rmsprop', 'batch_size': 5, 'epochs': 150}
0.752604 (0.017566) with: {'init': 'uniform', 'optimizer': 'adam', 'batch_size': 5, 'epochs': 150}
0.662760 (0.035132) with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'batch_size': 10, 'epochs': 50}
...
```

Рисунок 2.4 –Результати пошуку по ґратці

2.2 Алгоритм рандомного або випадкового пошуку

Випадковий пошук замінює повний перебір всіх комбінацій на їх випадковий вибір. Це можна легко застосувати до дискретних випадків, але метод можна узагальнити на неперервні та змішані простори. Випадковий пошук може перевершити пошук по ґратці, особливо, якщо лише мала кількість гіперпараметрів впливає на продуктивність алгоритму машинного навчання (рисунок 2.5). У цьому випадку кажуть, що завдання оптимізації має низьку внутрішню розмірність. Випадковий пошук також легко паралелізується і, крім того, можливе використання попередніх даних через вибір розподілу для вибірки випадкових параметрів [33].

Припустимо, що $f : Rn \to R -$ це фітнес-функція, або функція втрат, яку потрібно мінімізувати. Позначимо через $x \in Rn$ позицію або можливий варіант розв'язку в пошуковому просторі. Тоді базовий алгоритм випадкового пошуку можна описати наступним чином:

а) ініціювати x випадковою позицією в пошуковому просторі;

б) до тих пір поки критерій переривання не досягнений (наприклад, виконано максимальну кількість ітерацій або досягнено необхідного фітнесу) повторювати наступне:

в) вибирати нову позицію y з рівномірно розподілених точок на гіперсфері заданого радіусу d що оточує поточну позицію x. Для цього потрібно зробити наступне:

1) згенерувати n-вимірний гаусівський вектор з мат. сподіванням в точці 0 і довільною дисперсією: $x = (x_1, x_2, ..., x_n)$;

2) обрахувати радіус цього вектора (відстань від початку координат): $r = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$;

3) тоді рівномірно розподілений вектор заданого радіусу d можна знайти як $\frac{d}{r}x$;

г) якщо $(f(y) < f(x))$ – переходити на нову позицію заданням $x = y$;

д) тепер x має найкращу позицію.

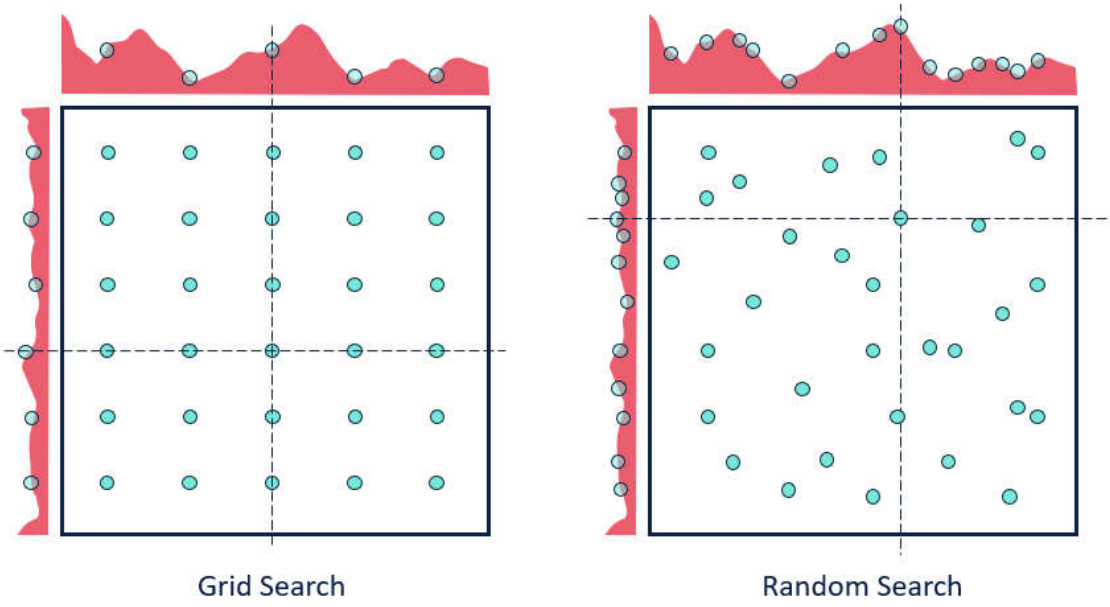

Рисунок 2.5 – Візуальне представлення методу пошуку по ґратці та рандомного пошуку

Адаптивний алгоритм випадкового пошуку – одна з найчастіше вживаних модифікацій алгоритму – використовує змінний крок пошуку залежно від досягненого успіху на попередніх кроках. Якщо дві послідовних ітерації дають покращення цільової функції, крок збільшується в a_s разів; якщо M послідовних ітерацій не дають покращення, крок зменшується в a_f разів. В загальному алгоритмі величина кроку d обчислюється наступним чином:

а) ініціювати довільне d, наприклад, $d = 1$ та лічильник невдалих ітерацій $m = 0$;

б) якщо нова позиція y є кращою за позицію x, збільшити d в a_s разів: $d = a_s d$;

в) якщо нова позиція y є гіршою за позицію x, збільшити лічильник на одиницю: $m = m + 1$;

г) якщо виконується умова $m \geq M$, зменшити d в a_f разів: $d = a_f d$ та обнулити лічильник: $m = 0$.

Допустимими є, наприклад, параметри $a_s = 1.618, a_f = 0.618, M = 3n$, де n – розмірність пошукового простору.

Використання алгоритму випадкового пошуку з адаптивним вибором кроку можливо в найнесподіваніших ситуаціях – наприклад, при виборі оптимального розміщення туристів в автобусі.

В літературі існує декілька варіантів випадкового пошуку:

– випадковий пошук із фіксованим кроком – базовий алгоритм Растригіна який обирає нові позиції із гіперсфери із заданим фіксованим радіусом;

– випадковий пошук з оптимальним кроком (Schumer and Steiglitz) – теоретична викладка з визначення оптимального радіусу гіперсфери для пришвидшеної конвергенції з оптимумом. Використання цього методу вимагає апроксимації цього оптимального радіусу шляхом багаторазової дискретизації, тому є занадто вимогливим до ресурсів для практичного застосування;

– випадковий пошук з адаптивним кроком (Schumer and Steiglitz) – алгоритм що евристично адаптує радіус гіперсфери. Проте, алгоритм дещо ускладнений;

– випадковий пошук із оптимізованим відносним кроком (Schrack and Choit) – апроксимує оптимальний розмір кроку шляхом простого експоненційного зменшення. Проте, формула для обчислення коефіцієнту згладжування дещо ускладнена.

Пошук нейронної архітектури (NAS) – це багатообіцяючий напрямок досліджень, який може замінити розроблені експертом мережі з вивченими, специфічними для архітектури завданнями. У праці [34] автори пропонують, щоб допомогти обґрунтувати емпіричні результати у цій галузі, нові базові лінії NAS, які формують такі спостереження:

– NAS – це спеціалізована проблема оптимізації гіперпараметрів;

– випадковий пошук є конкурентною базовою лінією для оптимізації гіперпараметрів.

Використовуючи ці спостереження, автори оцінюють як випадковий пошук з ранньою зупинкою, так і новий випадковий пошук з алгоритмом розподілу ваги. Їх результати показують, що випадковий пошук з ранньою зупинкою є конкурентоспроможною базовою лінією NAS, наприклад, він працює як мінімум так само, як ENAS [35], провідний метод NAS, на обох орієнтирах. Крім того, випадковий пошук за допомогою обміну вагою перевершує випадковий пошук з ранньою зупинкою.

Загальна проблема оптимізації гіперпараметрів має три компоненти, кожен з яких може мати специфічні для NAS підходи.

Пошуковий простір. Оптимізація гіперпараметрів включає визначення належної конфігурації гіперпараметра з набору можливих конфігурацій. Простір пошуку визначає цей набір конфігурацій і може включати безперервні або дискретні гіперпараметри структурованим або неструктурованим способом [35, 36, 37, 38]. Пошукові простори, специфічні для NAS зазвичай включають дискретні гіперпараметри з додатковою структурою, які можуть бути захоплені

спрямованим ациклічним графіком (DAG) [39, 17]. Крім того, оскільки пошукового простору для проектування всієї архітектури було б занадто багато, вузли та краї просторів пошуку зазвичай визначаються над деяким меншим будівельним блоком, тобто блок комірок деяким чином повторюється за допомогою заданої або вивченої мета-архітектури для формування більшої архітектури [14]. У статті автори описали проектування свого алгоритму випадкового пошуку NAS.

Метод пошуку. Враховуючи простір пошуку, існують різні способи пошуку для вибору можливих конфігурацій оцінки. Випадковий пошук – найосновніший підхід і він досить ефективний на практиці [36, 40]. Різні загальні і NAS-специфічні адаптаційні методи також були введені, всі вони намагаються змістити пошук шляху до конфігурацій, які швидше працюють. У традиційній оптимізації гіперпараметрів вибір способу пошуку може залежати від місця пошуку. Байєсівські підходи, засновані на Гауссових процесах [41, 42, 38, 43] та підходи на основі градієнтів [44, 45], як правило, застосовні лише для просторів безперервного пошуку.

На противагу цьому, Байєсівські дерева [46, 47], еволюційні стратегії [35] та випадковий пошук є більш гнучкими та можуть бути застосованими до будь-якого простору пошуку. Методи пошуку, специфічні для NAS, також можна класифікувати до одних і тих самих категорій.

Метод оцінювання. Для кожної конфігурації гіперпараметра, розглянутої методом пошуку, ми повинні оцінити її якість. Підхід за замовчуванням для проведення такої оцінки передбачає повне навчання моделі із заданими гіперпараметрами та, згодом, виміру його якості, наприклад, його точність прогнозування на валідаційному наборі.

Перше покоління методів NAS покладалося на повне оцінювання навчання, і тому потрібно дуже багато годин GPU для досягнення бажаного результату [48, 49, 50, 51]. Часткові методи навчання використовують швидку зупинку на швидкості вдосконалення процесу оцінювання за рахунок галасливих оцінок якості конфігурації. Ці методи використовують Байєсівську оптимізацію

[52, 41, 42], прогнозування продуктивності [53, 54] для адаптації розподілу ресурсів для різних конфігурацій. Методи оцінки, специфічні для NAS, експлуатують структуру нейроннної мережі для надання ще дешевших, евристичних оцінок якості. Багато з цих методів орієнтуються на обмін та повторне використання: мережеві морфізми будуються на попередньо навчених архітектурах [55, 56, 57]; гіпермережі та прогнозування продуктивності кодує інформацію з раніше розглянутих архітектур [58, 59, 60]; методи спільного розподілу ваг [61, 62, 39, 17, 60] використовують єдиний набір ваг для всіх можливих архітектур.

2.3 Генетичні алгоритми

Генетичний алгоритм (genetic algorithm) – це алгоритм пошуку, що використовується для вирішення задач оптимізації і моделювання шляхом послідовного підбору, комбінування і варіації параметрів з використанням механізмів, що нагадують біологічну еволюцію.

Особливістю генетичного алгоритму є акцент на використання оператора «схрещення», який виконує операцію рекомбінацію рішень-кандидатів, роль якої аналогічна ролі схрещення в живій природі. «Батьком-засновником» генетичних алгоритмів вважається Джон Голланд (John Holland), книга якого «Адаптація в природних і штучних системах» (Adaptation in Natural and Artificial Systems) є фундаментальною в цій сфері досліджень [63].

Задача кодується таким чином, щоб її вирішення могло бути представлено в вигляді масиву подібного до інформації складу хромосоми. Цей масив часто називають саме так «хромосома». Випадковим чином в масиві створюється деяка кількість початкових елементів «осіб», або початкова популяція. Особи оцінюються з використанням функції пристосування, в результаті якої кожній особі присвоюється певне значення пристосованості, яке визначає можливість

виживання особи. Після цього з використанням отриманих значень пристосованості вибираються особи, допущені до схрещення (селекція). До осіб застосовується «генетичні оператори» (в більшості випадків це оператор схрещення (crossover) і оператор мутації (mutation)), створюючи таким чином наступне покоління осіб [63]. Особи наступного покоління також оцінюються застосуванням генетичних операторів і виконується селекція і мутація. Так моделюється еволюційний процес, що продовжується декілька життєвих циклів (поколінь), поки не буде виконано критерій зупинки алгоритму.

Таким критерієм може бути:

– знаходження глобального, або надоптимального вирішення;

– вичерпання числа поколінь, що відпущені на еволюцію;

– вичерпання часу, відпущеного на еволюцію.

Генетичні алгоритми можуть використати для пошуку рішень в дуже великих і важких просторах пошуку.

Схема роботи генетичного алгоритму представлена на рисунку 2.6.

Серед етапів генетичного алгоритму можна виділити такі:

а) створення початкової популяції;

б) обчислення функції пристосованості для осіб популяції (оцінювання);

в) повторювання до виконання критерію зупинки алгоритму;

1) вибір індивідів із поточної популяції (селекція);

2) схрещення або/та мутація;

3) обчислення функції пристосовуваності для всіх осіб;

4) формування нового покоління.

Перед першим кроком необхідно випадковим чином створити деяку початкову популяцію. Навіть якщо популяція виявиться абсолютно неконкурентоздатною, генетичний алгоритм все одно достатньо швидко переведе її в придатну для життя популяцію. Таким чином, на першому кроці можна не старатися зробити надто пристосованих осіб, достатньо, щоб вони відповідали формату осіб популяції, і на них можна було порахувати функцію

пристосованості. Наслідком першого кроку є популяція H, що налічує N осіб.

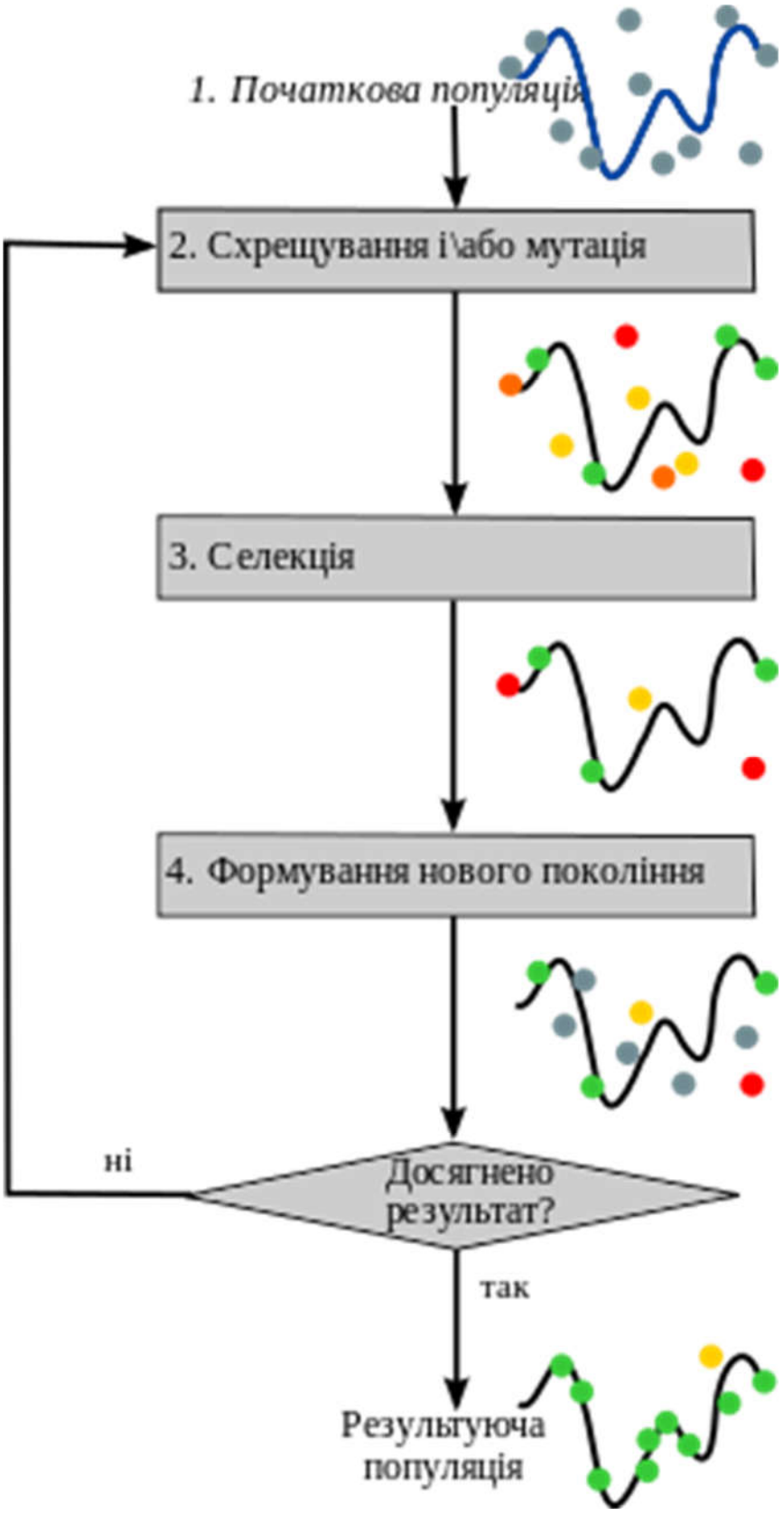

Рисунок 2.6 – Схема роботи генетичного алгоритму

На етапі відбору необхідно із всієї популяції вибрати її певну долю, яка залишиться в «живих» на цьому етапі популяції. Є декілька способів провести відбір. Ймовірність виживання особи h повинна залежати від значення її пристосованості $Fitness(h)$. Сама ж доля відібраних s зазвичай є параметром генетичного алгоритму, і її просто задають заздалегідь. Внаслідок відбору із N осіб популяції H повинні залишитись sN осіб, які ввійдуть в наступну популяцію H'. Решта осіб «загине».

Заснований на принципі колеса рулетки метод селекції вважається для генетичних алгоритмів основним методом відбору особин для батьківської популяції з метою подальшого їх перетворення генетичними операторами, такими як схрещування і мутація. Незважаючи на випадковий характер процедури селекції, батьківські особини вибираються пропорційно значенням їх функцій пристосованості: кожній хромосомі зіставлений сектор колеса рулетки, величина якого встановлюється пропорційною значенню функції пристосованості даної хромосоми:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_i}, \tag{2.1}$$

де p_i – величина сектора рулетки (імовірність, з якою особина буде відібрана для схрещування),

i – номер особини, що досліджується,

N – кількість особин початкової популяції,

f_i – значення цільової функції для i-ої особини.

Розмноження в генетичних алгоритмах зазвичай “статеве” – щоб «народити» нащадка, необхідно декілька батьків, зазвичай потрібна участь двох. Розмноження в різних алгоритмах описується по різному – воно, звісно, залежить від формату осіб. Головна вимога до розмноження – щоб нащадок чи

нащадки мали можливість успадкувати риси всіх батьків, «змішавши» їх якимось достатньо розумним чином.

Ну тут все як у людей, для отримання нащадка потрібно двоє батьків. Головне, щоб нащадок (дитина) міг успадкувати від батьків їх риси. При цьому розмножуються всі, а не тільки ті, що вижили (ця фраза особливо абсурдна, але так як у нас все в сферичному вакуумі, то можна все), в іншому випадку виділиться один альфа самець, гени якого перекриють всіх інших, а нам це принципово не прийнятно.

Розмноження або оператор рекомбінації застосовують відразу ж після оператора відбору батьків для отримання нових особин-нащадків. Сенс рекомбінації полягає в тому, що створені нащадки повинні наслідувати генну інформацію від обох батьків. Розрізняють дискретну рекомбінацію і кросинговер.

Приклад операції розмноження: вибрати $(1-s)p/2$ пар гіпотез із H і провести з ними розмноження, отримавши по два нащадка від кожної пари (якщо розмноження описано так, щоб давати одного нащадка, необхідно вибрати ($(1-s)p$ пар), і додати цих нащадків в H'. В результаті H' буде складатися з N осіб.

Особи для розмноження зазвичай вибираються із всієї популяції H, а не із тих, що вижили на першому кроці (хоча останній варіант теж має право на існування). Справа в тому, що головна проблема генетичних алгоритмів – недостача різноманітності (diversity) в особах. Достатньо швидко виділяється єдиний генотип, який являє собою локальний максимум і згодом всі елементи популяції програють йому в відборі, і вся популяція «забивається» копіями цієї особи. Існують різні способи боротьби із таким небажаним ефектом; один з них – вибір для розмноження не з самих «пристосованих», а взагалі зі всіх осіб.

До мутацій відноситься все те ж, що і до розмноження: є деяка доля мутантів m, що є параметром генетичного алгоритму, і на кроці мутацій

необхідно вибрати mN осіб, а згодом змінити їх згідно із заздалегідь заданими операціями мутації.

На рисунку 2.7 представлена проста реалізація генетичного алгоритму на мові програмування С++.

```
# include <iostream.h>
# include <algorithm.h>
# include <numeric.h>
using namespace std;
int main()
{
    //початковий масив (популяція) з 1000 елементів (осіб).
    const int N = 1000;
    int a[N];
    //заповнимо елементи нулями
    fill(a, a+N, 0);
    for (;;)
    {
        //Мутація кожного елемента.
        //Випадково збільшуємо або зменшуємо значення елементу на один;
        for (int i = 0; i < N; ++i)
            if (rand()%2 == 1)
                a[i] += 1;
            else
                a[i] -= 1;
        //відсортуванням по зростанню вибираємо більші за значенням...
        sort(a, a+N);
        //... і тоді більші за значенням виявляться в другій частині масиву.
        //скопіюємо більші в першу половину, коли вони залишили нащадків, а перші померли:
        copy(a+N/2, a+N,    a /*куди*/);
        //тепер поглянемо на середнє значення елементу популяції. Як бачимо, середнє значення все більше і більше.
        cout << accumulate(a, a+N, 0) / N << endl;
    }
}
```

Рисунок 2.7 – Реалізація генетичного алгоритму на С++

Шматок коду на рисунку 2.7 реалізовує програму пошуку в одномірному просторі, без схрещення. Ця програма вважає більші за значенням елементи представлені цілими числами найбільш життєздатними.

Оскільки алгоритм сам навчається, то спектр застосування вкрай широкий:

– завдання на графи;

– завдання компонування;

– складання розкладів;

– створення «штучного інтелекту».

Генетичний алгоритм – це, в першу чергу, еволюційний алгоритм, іншими словами, основна фішка алгоритму – схрещування (комбінування). Як нескладно

здогадатися, ідея алгоритму нахабним чином взята у природи. Так ось, шляхом перебору і, найголовніше, відбору виходить правильна «комбінація».

Алгоритм ділиться на три етапи:

– схрещування;

– селекція (відбір);

– формування нового покоління.

Якщо результат нас не влаштовує, ці кроки повторюються до тих пір, поки результат не почне нас задовольняти або поки не станеться одна з нижче перерахованих умов:

– кількість поколінь (циклів) досягне заздалегідь обраного максимуму;

– вичерпається час на мутацію.

Існує кілька варіантів генетичного алгоритму, а саме:

– класичний генетичний алгоритм;

– генітор (Genitor);

– гібридний генетичний алгоритм.

Характеристиками класичного генетичного алгоритму є:

– фіксований розмір популяції;

– фіксована розрядність генів;

– пропорційний відбір;

– особини для схрещування вибираються випадковим чином;

– одноточковий кросовер і одноточкова мутація;

– наступне покоління формується з нащадків поточного покоління без “елітизму”, нащадки займають місця своїх батьків;

– бінарне кодування змінних.

Вид генетичного алгоритму Генітор (Genitor) володіє наступними характеристиками:

– фіксований розмір популяції;

– фіксована розрядність генів;

– особини для схрещування вибираються випадковим чином;

– обмежень на тип кросовера і мутації немає;

– внаслідок схрещування особин виходить один нащадок, який займає місце найменш пристосованої особи;

– дійсне та бінарне кодування змінних.

Гібридний генетичний алгоритм має такі характеристики:

– фіксований розмір популяції;

– фіксована розрядність генів;

– будь-які комбінації стратегій відбору та формування наступного покоління;

– обмежень на тип кросовера і мутації немає;

– використання ГА на початковій стадії вирішення задач оптимізації, а потім в роботу включаються класичні методи оптимізації;

– дійсне та бінарне кодування змінних.

Давайте розглянемо генетичний алгоритм на прикладі Діофантових рівнянь (рівняння з цілочисельними коренями).

Наше рівняння: $a + 2b + 3c + 4d = 30.$

Ви напевно вже здогадались, що корені цього рівняння лежать на відрізку [1;30], тому ми беремо 5 випадкових значень a, b, c, d. Обмеження в 30 взято спеціально для спрощення завдання.

І так, у нас є перше покоління:

1) (1,28,15,3);

2) (14,9,2,4);

3) (13,5,7,3);

4) (23,8,16,19);

5) (9,13,5,2);

Для того, щоб обчислити коефіцієнти виживання, підставимо кожне рішення у вираз. Відстань від отриманого значення до 30 і буде потрібним значенням.

1) $114 - 30 = 84$;

2) $54 - 30 = 24$;

3) $56 - 30 = 26$;

4) $163 - 30 = 133$;

5) $58 - 30 = 28$.

Менші значення ближче до 30, відповідно є більше бажаними. Виходить, що великі значення матимуть менший коефіцієнт виживання. Для створення системи обчислимо ймовірність вибору кожної хромосоми. Але рішення полягає в тому, щоб взяти суму зворотних значень коефіцієнтів, і виходячи з цього обчислювати відсотки (0.135266 – сума зворотних коефіцієнтів).

1) $(1/84)/0{,}135266 = 8{,}80\%$;

2) $(1/24)/0{,}135266 = 30{,}8\%$;

3) $(1/26)/0{,}135266 = 28{,}4\%$;

4) $(1/133)/0{,}135266 = 5{,}56\%$;

5) $(1/28)/0{,}135266 = 26{,}4\%$.

Далі будемо вибирати п'ять пар батьків, у яких буде рівно по одній дитині. Давати місце випадку ми будемо давати рівно п'ять разів, кожен раз шанс стати батьком буде однаковим і буде дорівнювати шансу на виживання.

Як було сказано раніше, нащадок містить інформацію про гени батька і матері. Це можна забезпечити різними способами, але в даному випадку буде використовуватися «кросовер» рисунок 2.8.

Х.-батько a1 | b1,c1,d1 **Х.- мати** a2 | b2,c2,d2 **Х.- нащадок** a1,b2,c2,d2 or a2,b1,c1,d1

Х.-батько a1,b1 | c1,d1 **Х.- мати** a2,b2 | c2,d2 **Х.- нащадок** a1,b1,c2,d2 or a2,b2,c1,d1

Х.-батько a1,b1,c1 | d1 **Х.- мати** a2,b2,c2 | d2 **Х.- нащадок** a1,b1,c1,d2 or a2,b2,c2,d1

Рисунок 2.8 – Обчислення

Є дуже багато шляхів передачі інформації нащадкові, а кросовер – тільки один з безлічі. Розташування розділювача може бути абсолютно довільним, як і те, хто з батьків буде ліворуч від межі.

А тепер зробимо те ж саме з нащадками:

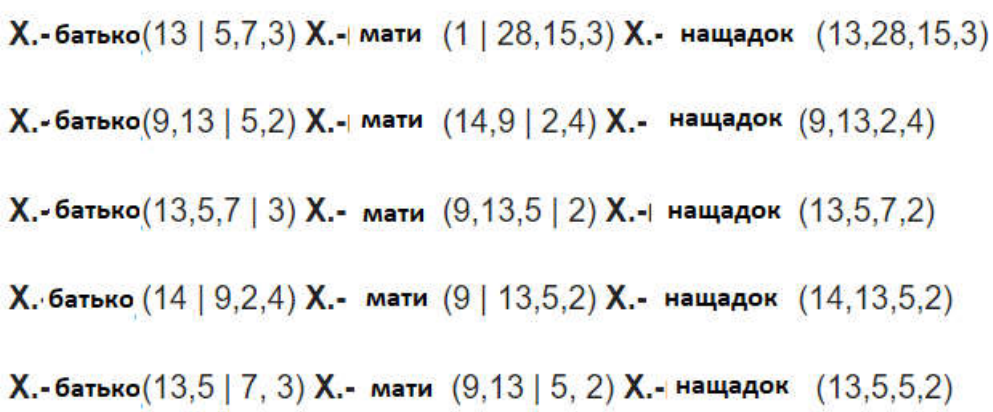

Рисунок 2.9 – Обчислення нащадків

Тепер обчислимо коефіцієнти виживання нащадків (рисунок 2.10).

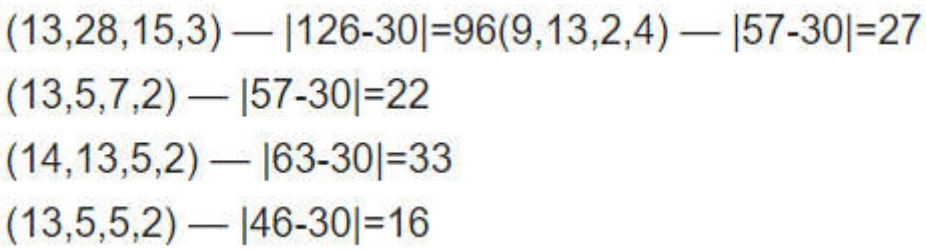

Рисунок 2.10 – Коефіцієнти виживання нащадків

Сумно, так як середня пристосованість (fitness) нащадків виявилася 38,8, а у батьків цей коефіцієнт дорівнював 59,4. Саме в цей момент доцільніше використовувати мутацію, для цього замінимо один або більше значень на випадкове число від 1 до 30.

Алгоритм буде працювати до тих пір, поки коефіцієнт виживання не буде дорівнювати нулю. Тобто буде вирішенням рівняння.

Системи з більшою популяцією. Наприклад, 50 замість 5 сходяться до бажаного рівня (0) більш швидко і стабільно.

Клас на C ++ вимагає 5 значень при ініціалізації: 4 коефіцієнти і результат. Для вище наведеного прикладу це буде виглядати так: CDiophantine dp (1,2,3,4,30);

Потім, щоб вирішити рівняння, треба викликати функцію Solve (), яка поверне аллель, що містить рішення. Потрібно викликати GetGene (), щоб отримати ген з правильними значеннями a,b,c,d. Стандартна процедура main.cpp, що використовує цей клас, приведена на рисунку 2.11.

```
#include "<iostream.h>"
#include "diophantine.h"

void main() {

   CDiophantine dp(1,2,3,4,30);

   int ans;
   ans = dp.Solve();
   if (ans == -1) {
      cout << "No solution found." << endl;
   } else {
      gene gn = dp.GetGene(ans);

      cout << "The solution set to a+2b+3c+4d=30 is:\n";
      cout << "a = " << gn.alleles[0] << "." << endl;
      cout << "b = " << gn.alleles[1] << "." << endl;
      cout << "c = " << gn.alleles[2] << "." << endl;
      cout << "d = " << gn.alleles[3] << "." << endl;
   }
}
```

Рисунок 2.11 – Стандартна процедура main.cpp

У другому розділі було проаналізовано алгоритми синтезу структур згорткових нейронних мереж, зокрема: алгоритм повного перебору, алгоритм рандомного пошуку та генетичні алгоритми. Також було детально описано їх роботу із наведенням прикладів.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ЗАВДАННЯ

3.1 Системні вимоги

Для правильного функціонування розробленого програмного модуля на персональному комп’ютері він повинен відповідати наступним мінімальним системним вимогам:

– процесор з тактовою частотою не менш ніж 2,2ГГц;

– оперативна пам’ять обсягом 8Гб та поколінням пам’яті DDR4 або DDR3 (наприклад, Kingston DDR4-2400 HyperX Fury);

– відеокарта з підтримкою технології CUDA та обсягом пам’яті не менш ніж 8Гб та з поколінням пам’яті не нижче GDDR5 (наприклад, GeForce GTX 1070 8GB);

– жорсткий диск розміром 50Гб.

Також на комп’ютері користувача має бути встановлене наступне програмне забезпечення:

– операційна система Windows 10 x64 або Ubuntu x64;

– середовище Python 3.6

– середовище Anaconda із встановленим Tensorflow ≥ 1.13 (рисунок 3.1);

– остання версія драйверів для відеокарти.

CUDA – програмно-апаратна архітектура паралельних обчислень, яка дозволяє істотно збільшити обчислювальну продуктивність завдяки використанню графічних процесорів (GPUs) фірми Nvidia (рисунок 3.2).

CUDA SDK надає можливість включати в текст програм на C виклик підпрограм, що виконуються на графічних процесорах Nvidia. Це реалізовано шляхом команд, які записуються на особливому діалекті C. Архітектура CUDA дає розробнику можливість на свій розсуд організовувати доступ до набору інструкцій графічного прискорювача й керувати його пам'яттю.

Існують, навіть, спеціальні відеокарти сімейства QUADRO та TESLA «заточені» для паралельних обчислень.

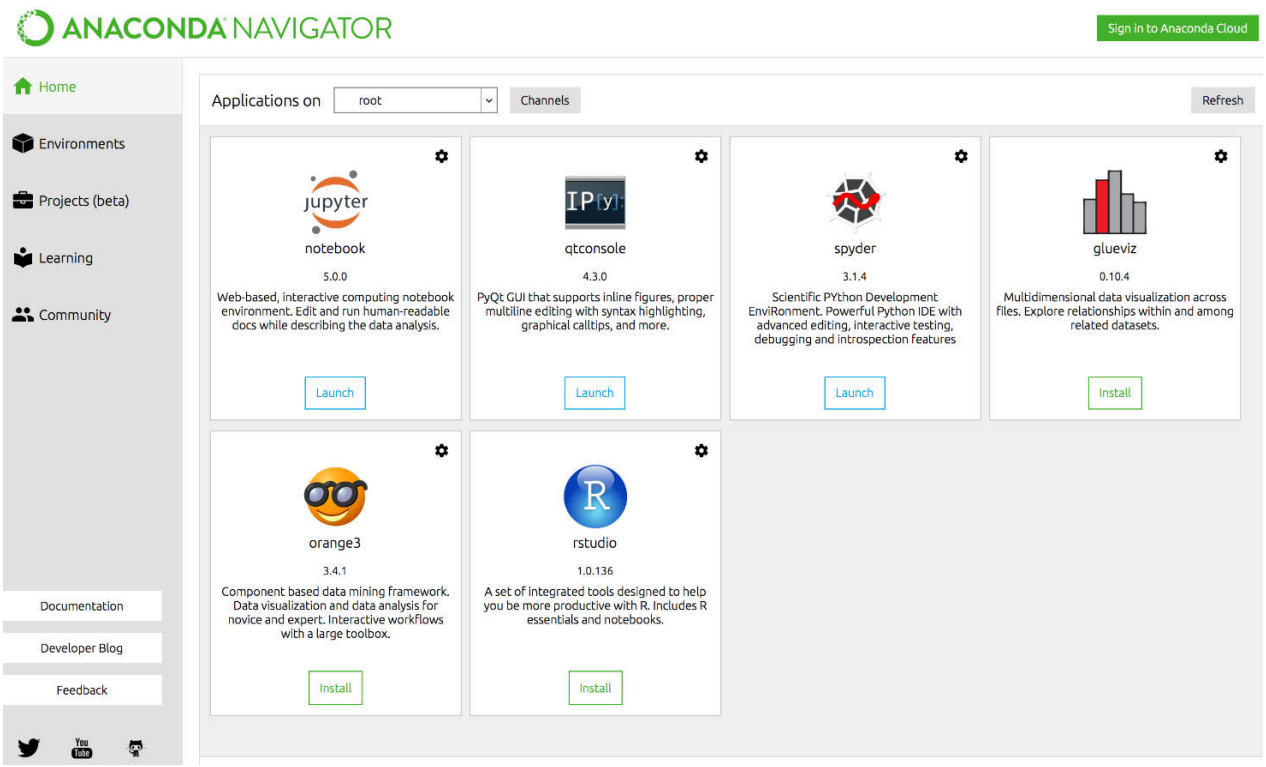

Рисунок 3.1 – Початковий інтерфейс Anaconda

Рисунок 3.2 – Логотип CUDA

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

– інтерфейс програмування додатків CUDA базується на стандартній мові програмування С з деякими обмеженнями;

– спільна між потоками пам'ять розміром в 16 Кб може бути використана під організований користувачем кеш з більш широкою смугою пропускання, ніж при вибірці зі звичайних текстур;

– більш ефективні транзакції між пам'яттю центрального процесора і відеопам'яттю;

– повна апаратна підтримка цілочисельних і побітових операцій.

Різні відеокарти мають різні обчислювальні можливості. У таблиці 3.1 наведено перелік сучасних відеокарт, що підтримують технологію CUDA.

Таблиця 3.1 – Список підтримуваних відеокарт

Версія специфікації	GPU	Відеокарта
1	2	3
1.0	G80, G92, G92b, G94, G94b	GeForce 8800GTX/Ultra, Tesla C/D/S870, FX4/5600, 360M, GT 420
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b	GeForce 8400GS/GT, 8600GT/GTS, 8800GT/GTS, 9400GT, 9600 GSO, 9600GT, 9800GTX/GX2, 9800GT, GTS 250, GT 120/30/40, FX 4/570, 3/580, 17/18/3700, 4700x2, 1xxM, 32/370M, 3/5/770M, 16/17/27/28/36/37/3800M, NVS420/50

Продовження таблиці 3.1

1.2	GT218, GT216, GT215	GeForce 210, GT 220/40, FX380 LP, 1800M, 370/380M, NVS 2/3100M
1.3	GT200, GT200b	GeForce GTX 260, GTX 275, GTX 280, GTX 285, GTX 295, Tesla C/M1060, S1070, Quadro CX, FX 3/4/5800
2.0	GF100, GF110	GeForce (GF100) GTX 465, GTX 470, GTX 480, Tesla C2050, C2070, S/M2050/70, Quadro Plex 7000, Quadro 4000, 5000, 6000, GeForce (GF110) GTX 560 TI 448, GTX570, GTX580, GTX590
2.1	GF104, GF114, GF116, GF108, GF106	GeForce 610M, GT 430, GT 440, GT 640, GTS 450, GTX 460, GTX 550 Ti, GTX 560, GTX 560 Ti, 500M, Quadro 600, 2000
3.0	GK104, GK106, GK107	GeForce GTX 690, GTX 680, GTX 670, GTX 660 Ti, GTX 660, GTX 650 Ti, GTX 650, GeForce GTX 680MX, 645M, GeForce GT 640M
3.5	GK110, GK208	GeForce GTX TITAN, GeForce GTX TITAN Black, GeForce GTX 780 Ti, GeForce GTX 780
5.0	GM107, GM108	GeForce GTX 750 Ti, GeForce GTX 750 , GeForce GTX 860M, GeForce GTX 850M, GeForce 840M, GeForce 830M
5.2	GM200, GM204, GM206	GeForce GTX Titan X, GeForce GTX 980 Ti, GeForce GTX 980, GeForce GTX 970, GeForce GTX 960, GeForce GTX 950, GeForce GTX 970M, GeForce GTX 965M

Графічні процесори сьогодні використовуються в широкому діапазоні застосувань, головним чином тому, що вони можуть різко прискорити

паралельні обчислення, бути доступними та енергоефективними. У галузі медичної візуалізації графічні процесори у деяких випадках є вирішальними для забезпечення практичного використання алгоритмів, що вимагають обчислень.

Різниця між типами пам’яті приведена на рисунку 3.3.

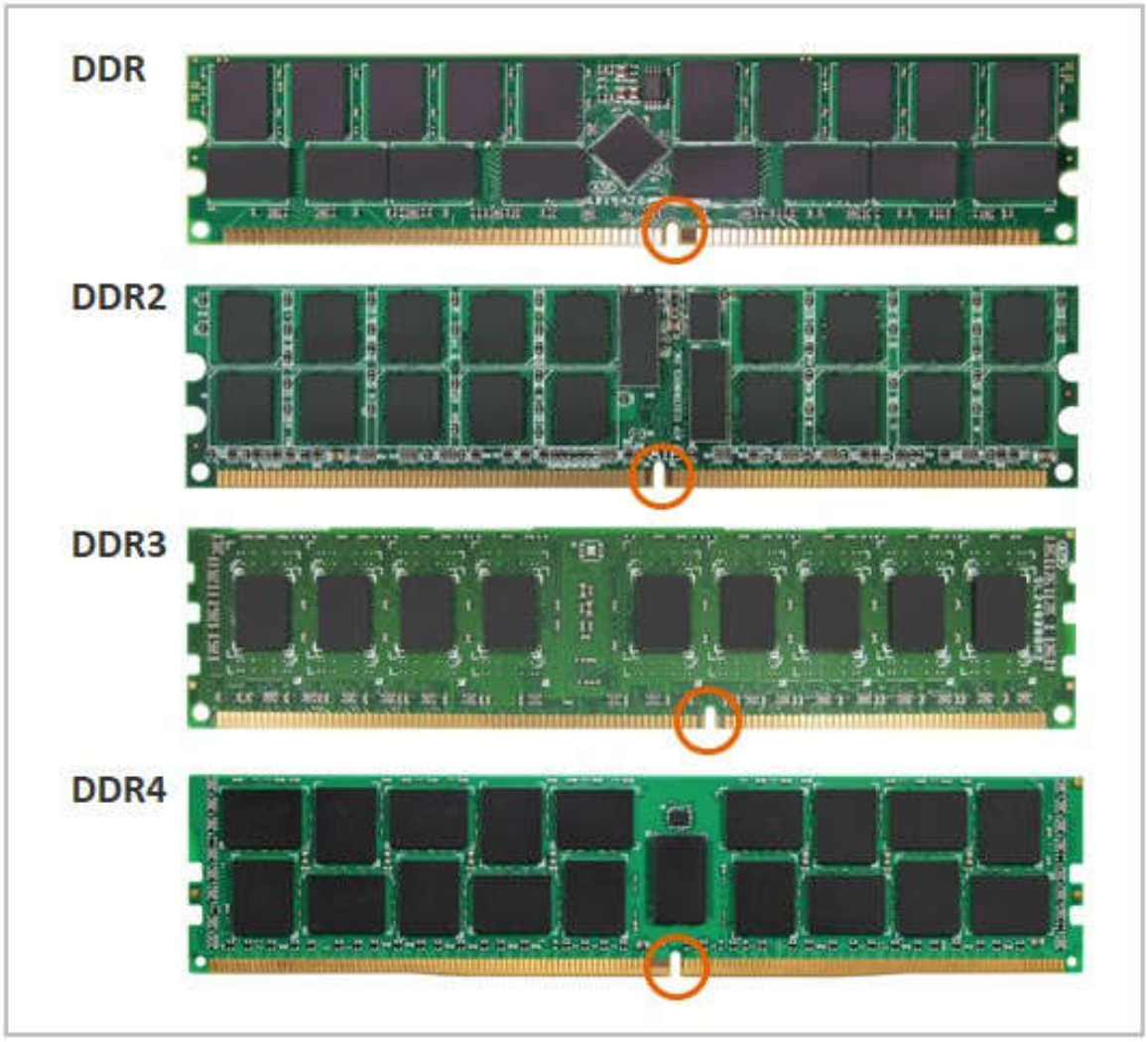

Рисунок 3.3 – Різниця між типами пам'яті

Яка різниця між DDR3 та DDR4? DDR3 мав досить успішний запуск, поки DDR4 все ще залишається новим типом пам’яті. DDR3 був вперше випущений у 2007 році та використовувався у всьому, від LG1313, LGA1151 (лише 6-й / 7-й Gen Core), а також AMD / AM3 / AM3 + та FM1 / 2/2 +. Однак усі сучасні платформи (починаючи з 2017 року) перейшли на режим DDR4, і більшість

платформ Intel відійшли від DDR3 за допомогою процесорів 6-го покоління Skylake.

DDR4 працює при меншій напрузі, ніж DDR3. DDR4 зазвичай працює на 1,2 вольта, що менше, ніж живлення в 1,5 у DDR3. Різниця в напрузі може призвести до економії 15 Вт в порівнянні з DDR3 – не багато для домашнього користувача.

Ще одна велика різниця між DDR3 та DDR4 – швидкість. Специфікації DDR3 офіційно починаються від 800 MT/c (або мільйони переказів в секунду) і закінчуються на DDR3-2133. Деякі модулі розгону піднімалися на DDR3-3200 і більше, але це неофіційна швидкість. Тим часом DDR4 запускається на частоті 1600 МГц, з офіційною підтримкою до DDR4-3200 – і набори розгону можуть набирати стільки ж, скільки й DDR4-4800. Підвищена швидкість означає загальне збільшення пропускної здатності.

3.2 Підготовка віртуальної машини та програмного середовища

Перед початком реалізації проекту мій домашній комп’ютер було протестовано шляхом навчання класифікатора (згорткової нейронної мережі) для кольорових гістологічних зображень розміром 128x128 пікселів. В ході тестування було отримано помилки типу Out of memory error. Ці помилки виникають, коли на комп’ютері недостатньо пам’яті, щоб виділити її під завантажені вхідні дані та модель. Отже, домашній комп’ютер не підходить для реалізації проекту.

Існує три можливі варіанти вирішення проблеми.

Перший варіант полягає у створенні віртуальної машини у сервісі Google Cloud Platform – запропонований компанією Google набір хмарних служб, які виконуються на тій же самій інфраструктурі, яку Google використовує для своїх продуктів призначених для кінцевих споживачів, таких як Google Search та

YouTube. Окрім інструментів для керування, також надається ряд модульних хмарних служб, таких як обчислення, зберігання даних, аналіз даних та машинне навчання. Для реєстрації потрібно мати банківську карту або банківський рахунок.

Таблиця 3.2 – Конфігурація домашнього комп'ютера

Процесор	AMD Athlon 64 x2 Dual Core, 2.51ГГц
Оперативна пам'ять	4ГБ, DDR2
Відеокарта	Nvidia GTX 950 Asus, серія Strix, 2Гб, GDDR5
Операційна система	Windows 10 Pro

Для того, щоб створити віртуальну машину потрібно перейти в Google Cloud Console, створити новий проект та обрати пункт Compute Engine у меню. Після чого відкриється вікно, зображене на рисунку 3.4, де можна обрати характеристики віртуальної машини.

Також у користувача є можливість збільшити або зменшити обсяг оперативної пам'яті та кількості ядер (рисунок 3.5).

Підсумуємо характеристики віртуальної машини, яка буде оптимальним варіантом для реалізація поставленого завдання, у таблиці 3.3.

Як бачимо з таблиці 3.3, віртуальна машина із наведеною конфігурацією буде обходитися нам у майже 300 американських доларів. Під ціною за місяць мається на увазі, що віртуальна машина (VM) буде працювати 7 днів на тиждень та 24 години на добу, тобто буде працювати неперервно. Для того, щоб уникнути зайвих витрат не потрібно забувати, що VM потрібно вимикати, коли немає необхідності у її використанні. Ціна за одну годину роботи становить 0,5 долара.

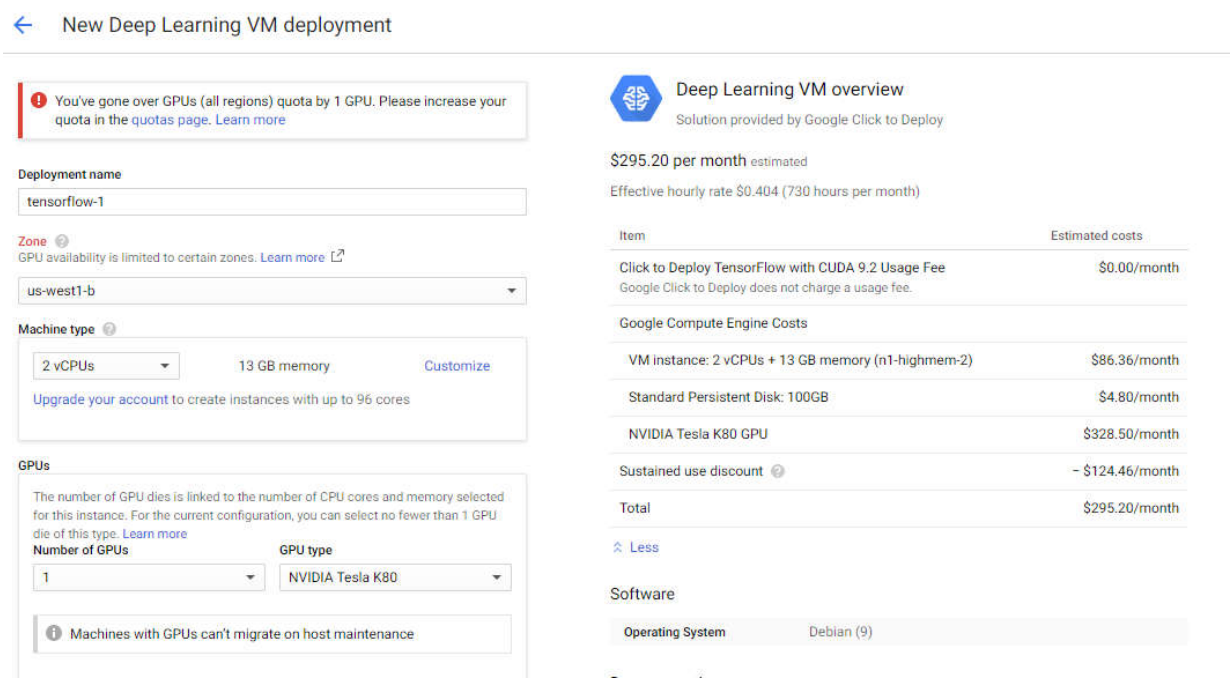

Рисунок 3.4 – Створення віртуальної машини в Google Cloud

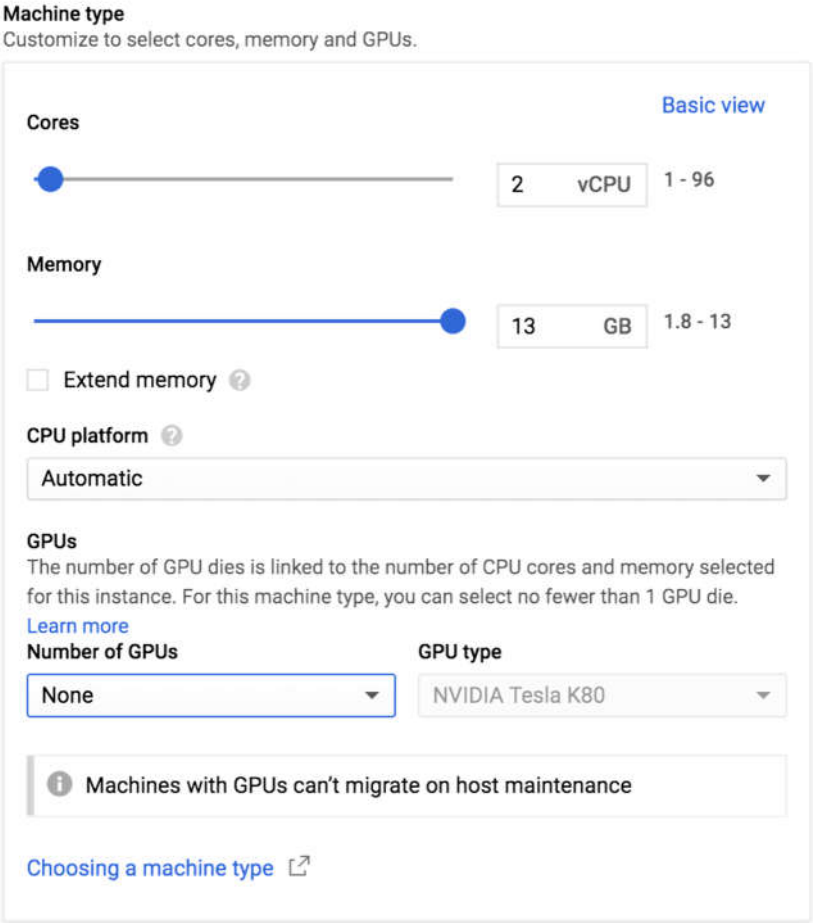

Рисунок 3.5 – Конфігурація VM

Таблиця 3.3 – Характеристики віртуальної машини

Процесор	2 ядра + 13Гб пам’яті
Диск	100Гб
Графічний процесор	Nvidia Tesla K80, 12Гб
Ціна за місяць	296 доларів
Ціна за годину	0.5 долара

Для того, щоб користувачі мали змогу порахувати ціну використання віртуальної машини, команда Google пропонує використовувати онлайн Google Cloud Pricing Calculator, де користувачу необхідно вказати бажані характеристики VM, після чого можна переглянути ціну обраної віртуальної машини (рисунок 3.6).

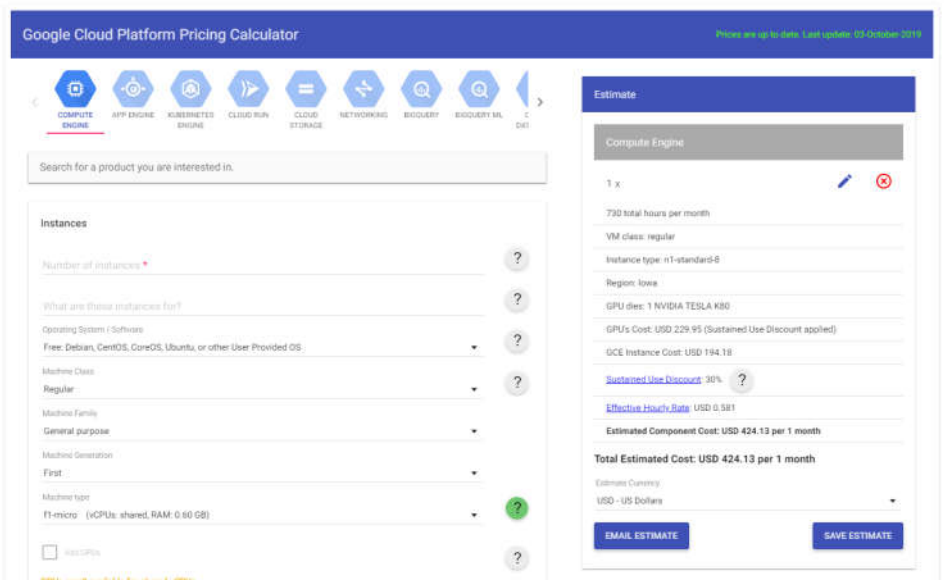

Рисунок 3.6 – Google Cloud Pricing Calculator

Другий варіант також полягає у створенні віртуальної машини, але вже на сервісі AWS (Amazon Web Services) – надає платформу хмарних обчислень в оренду приватним особам, компаніям та урядам на основі платної підписки. Існує і безкоштовна підписка, яка доступна протягом перших 12 місяців. Технологія дозволяє абонентам мати у своєму розпорядженні повноцінний

віртуальний кластер комп'ютерів, який завжди доступний через Інтернет. Віртуальні комп'ютери AWS мають більшість атрибутів реального комп'ютера, включаючи апаратні пристрої (процесор, відеокарту, локальну та оперативну пам'ять, жорсткий диск або SSD-накопичувач); операційну систему на вибір; мережу; і попередньо встановлені прикладні програми, такі як веб-сервер, база даних, CRM і т. д. Кожна система AWS також віртуалізує консольний ввід/вивід (клавіатура, дисплей і миша), що дозволяє користувачам AWS підключитися до своєї системи AWS за допомогою браузера. Браузер виступає як вікно у віртуальний комп'ютер, дозволяючи користувачу входити в систему, налаштовувати та використовувати свої віртуальні системи так само, як справжній, фізичний комп'ютер. Це дозволяє їм налаштувати систему так, щоб надавати інтернет-орієнтовані сервіси та послуги своїм клієнтам.

Віртуальна машина на AWS, що є схожою за конфігурацією до VM (рисунок 3.4) на Google Cloud обійдеться у 0,9 долара за годину, що майже у два рази дорожче. Проте, варто врахувати, що у цій VM процесор має 4 ядра, а не два, та кількість оперативної пам’яті становить 61Гб, а не 13.

Name	GPUs	vCPUs	RAM (GiB)	Network Bandwidth	Price/Hour*	RI Price / Hour**
p2.xlarge	1	4	61	High	$0.900	$0.425
p2.8xlarge	8	32	488	10 Gbps	$7.200	$3.400
p2.16xlarge	16	64	732	20 Gbps	$14.400	$6.800

Рисунок 3.7 – VMs з підтримкою GPU на AWS та їх ціна за годину

Третій варіант полягає у використанні сервісу Google Colaboratory, який надає змогу користувачеві запускати Jupyter Notebook («ноутбук», інтерактивне виконання коду на мові Python) та можливість використовувати графічний процесор Nvidia Tesla K80 безкоштовно. Проте, є один нюанс – середовище перезапускається кожних 12 годин. Тобто всі дані, які користувач завантажить у

свій «ноутбук» зникнуть через 12 годин а всі виконувані процеси (в тому числі і навчання нейромережі, якщо користувач використовує «ноутбук» для цього) припиняться. Даний варіант підійде тим, хто не має бажання витрачати кошти на дорогі графічні процесори, встановлювати драйвери та налаштовувати середовище програмування самостійно.

Технічні характеристики такого «ноутбука» наступні:

Таблиця 3.4 – Технічні характеристики Google Colab

Процесор	Intel Xeon 2,3ГГц, 1 ядро
Диск	320Гб
Пам’ять	12,6Гб
Графічний процесор	Nvidia Tesla K80, 12Гб
Кожні 12 годин дані із диска, оперативної пам’яті, кешу процесора тощо, які будуть розміщені на виділеній віртуальній машині, будуть стерті	

Плюсом даного сервісу є те, що користувачу непотрібно самостійно встановлювати популярні бібліотеки для машинного навчання, такі як Tensorflow, PyTorch – вони встановлені за замовчуванням. Користувач зосереджений тільки на написанні коду. Також є можливість запускати не тільки Jupyter Notebook, а й звичайні файли коду Python.

Скористаємося третім варіантом.

3.3 Засоби та платформа реалізації

Проаналізувавши поставлену задачу, зрозуміло, що перевага має надаватися тій мові програмування, яка найбільш пристосована до роботи із машинним навчанням та великими даними. По суті, машинне навчання – це

технологія, яка допомагає додаткам на основі штучного інтелекту навчатися і видавати результати автоматично, без людського втручання. Спеціаліст з машинного навчання повинен збирати, систематизувати і аналізувати дані, а потім на основі отриманої інформації створювати алгоритми для штучного інтелекту. Python найкраще підходить для виконання таких завдань, тому що він досить зрозумілий в порівнянні з іншими мовами. Більш того, у нього відмінна продуктивність при обробці даних.

Одна з основних причин, чому Python використовується для машинного навчання полягає в тому, що у нього є безліч фреймворків, які спрощують процес написання коду і скорочують час на розробку. До таких фреймворків входять Tensorflow, Keras, PyTorch, Theano, Caffe та багато інших. Плюсом є те, що зараз кожна така бібліотека підтримує обчислення на графічних процесорах із коробки. Користувачу залишається тільки встановити програмний пакет CUDA та найсвіжіші драйвери для своєї відеокарти.

Для реалізації поставленого завдання обрано фреймворк Tensorflow версії 1.14 – це відкрита програмна бібліотека для машинного навчання, розроблена компанією Google, де зараз її і застосовують для досліджень та розробки власних продуктів.

В якості віртуальної машини обрано сервіс Google Colaboratory та мову програмування Python 3.6.

3.4 Початкова архітектура та вхідні параметри

В якості початкової моделі використовується наступна модель, яка складається із таких шарів:

– згортка;

– активація ReLU;

– згортка;

– активація ReLU;

– макс-пулінг 2d;

– dropout;

– згортка;

– активація ReLU;

– згортка;

– активація ReLU;

– dropout;

– flatten;

– повнозв’язний шар;

– повнозв’язний шар.

Вхідними даними для мережі обрано кольорові цитологічні зображення 64 $\times$ 64 пікселі, що розділені на чотири класи.

Приклад цитологічного зображення приведено на рисунку 3.8. На рисунку 3.9 наведено усю вибірку цитологічних зображень. Дана вибірка складається з 78 зображень, але, на жаль, цього недостатньо для навчання глибокої згорткової нейронної мережі. Тому нову вибірку було отримано шляхом генерування зображень, за допомогою GAN-мережі.

Нова вибірка складається, приблизно, із 1900 зображень (рисунок 3.10).

3.5 Опис роботи програмної частини

Програма генерує структури згорткових нейронних мереж за допомогою алгоритму Grid Search (пошуку по ґратці), який був описаний у розділах вище (див. рисунок 2.5) [4].

Користувач може задавати такі параметри (рисунок 3.11):

– кількість повнозв'язних шарів;

– кількість фільтрів;

– кількість згорткових шарів.

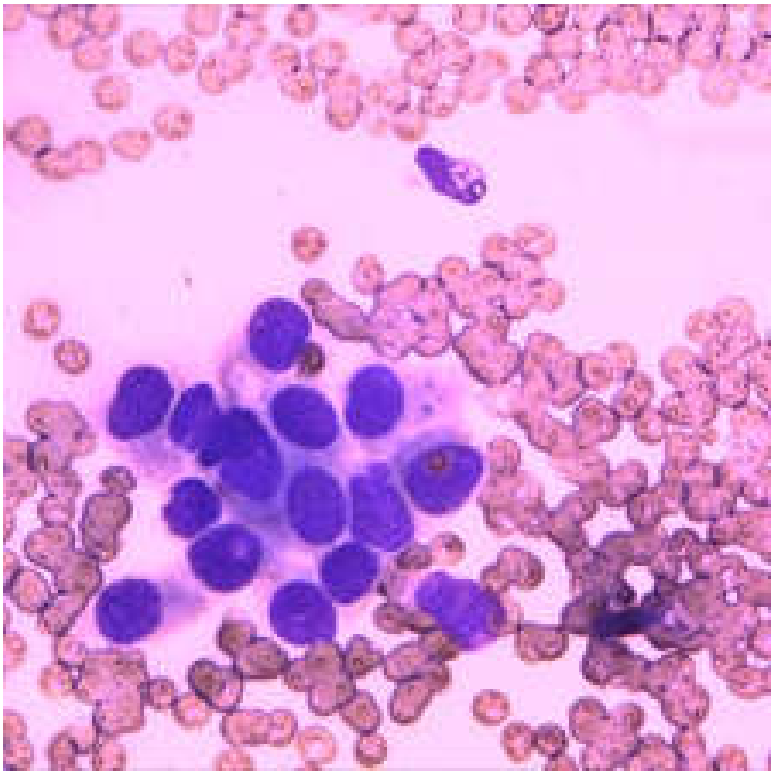

Рисунок 3.8 – Цитологія

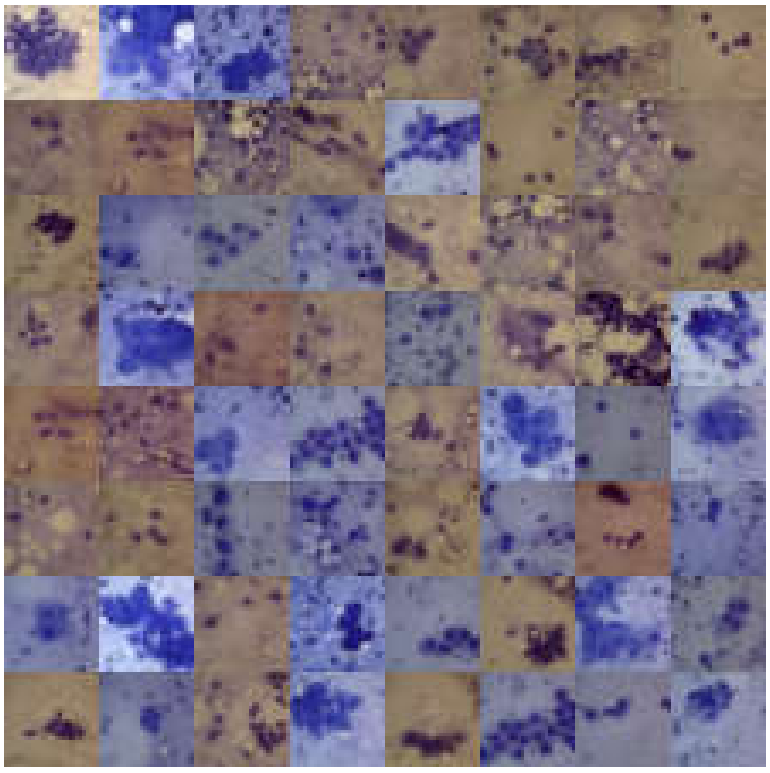

Рисунок 3.9 – Вибірка цитологічних зображень

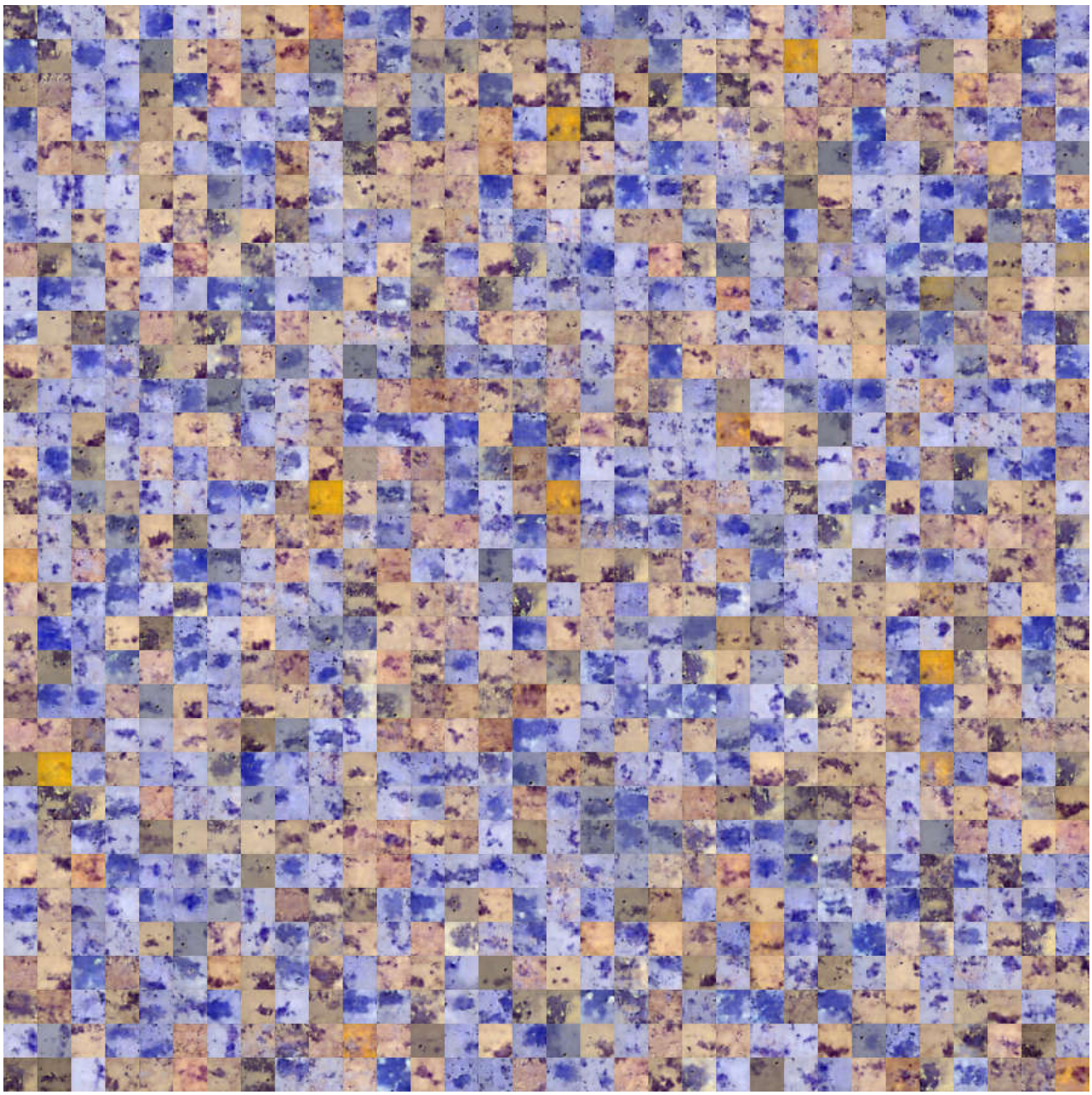

Рисунок 3.10 – Нова вибірка

```
dense_layers = [2]
filters = [64, 128, 256]
conv_layers = [2, 3]
```

Рисунок 3.11 – Параметри для генерування структури

Програма генерує структури згорткових нейронних мереж для всіх комбінацій, після чого, кожна з отриманих мереж навчається протягом двадцяти епох на даних, описаних вище (див. рисунок 3.9).

Кожна з моделей оцінюється на тестових даних.

Для кожної моделі виводиться точність та ROC-крива. Після завершення навчання всіх моделей, користувачу виводиться комбінація параметрів, що показали найкращі результати.

Алгоритм роботи програми представлений на рисунку 3.12.

3.6 Тестування програмної частини

Початкові параметри були задані згідно з рисунком 3.11. Дані комбінації передбачають шість структур згорткових нейронних мереж ($1 \times 3 \times 2 = 6$).

Після навчання кожна модель отримала таку точність:

– 2-conv-with-64-filters-2-dense-with-64-units – точність 0.9538;

– 2-conv-with-128-filters-2-dense-with-128-units – точність 0.9612;

– 2-conv-with-256-filters-2-dense-with-256-units – точність 0.9530;

– 3-conv-with-64-filters-2-dense-with-64-units – 0.9460;

– 3-conv-with-128-filters-2-dense-with-128-units – 0.9499;

– 3-conv-with-256-filters-2-dense-with-256-units – 0.9643.

ROC-крива для моделі №1 зображена на рисунку 3.13.

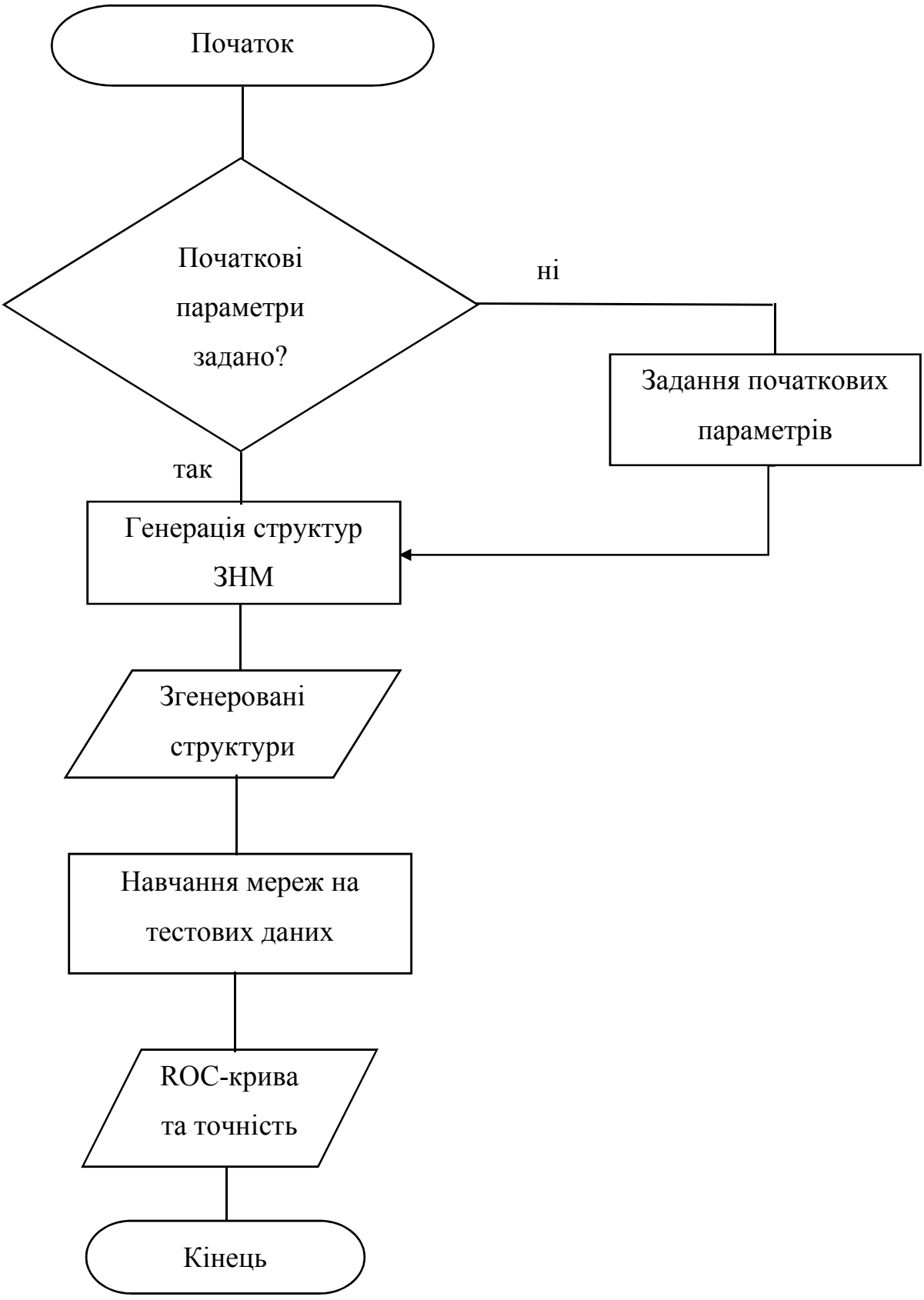

Рисунок 3.12 – Схема роботи алгоритму

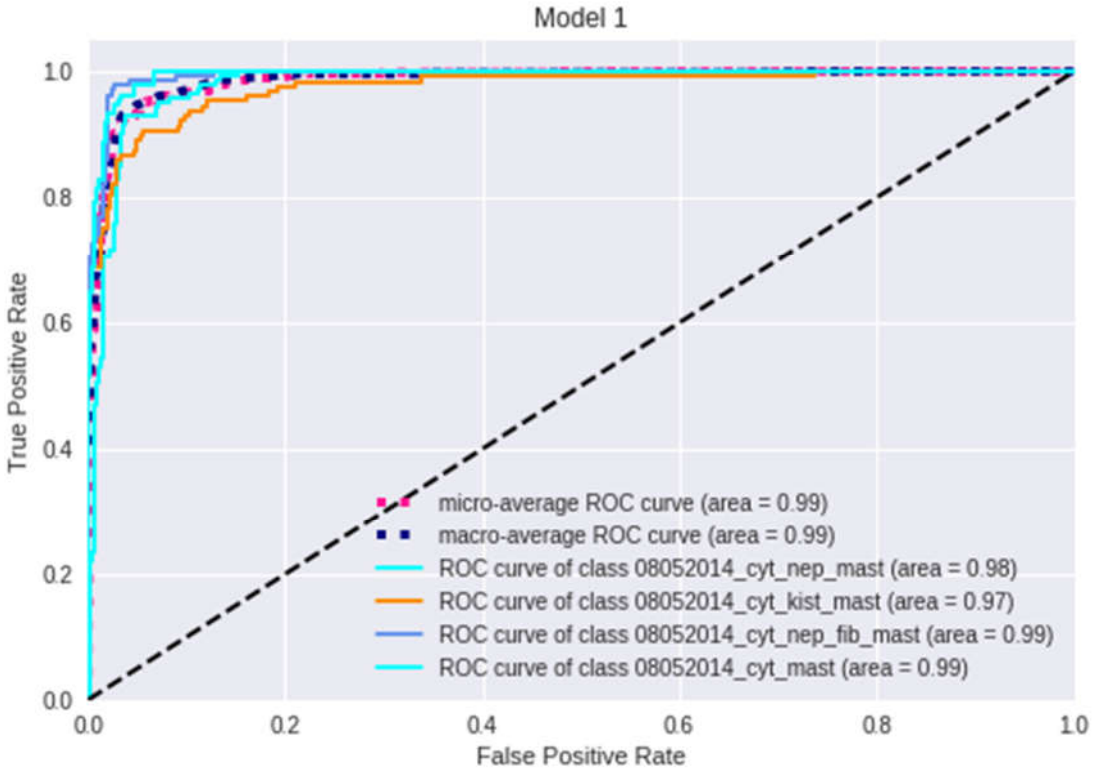

Рисунок 3.13 – Крива для моделі №1

ROC-крива для моделі №2 зображена на рисунку 3.14.

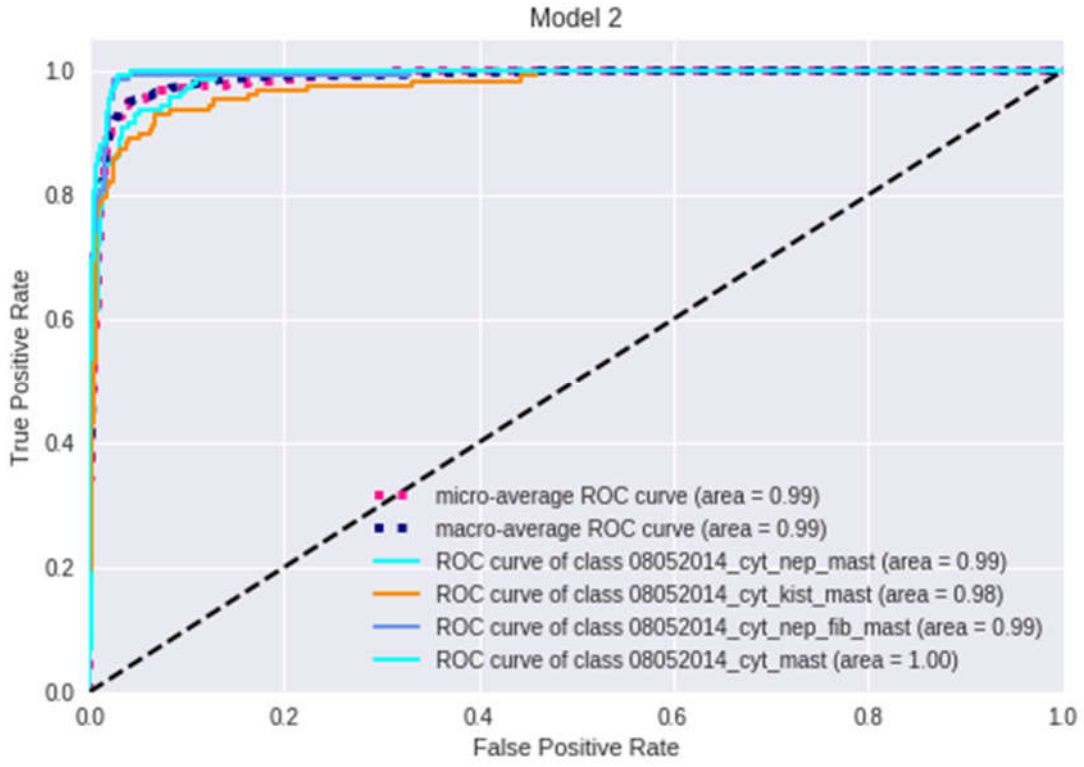

Рисунок 3.14 – Крива для моделі №2

ROC-крива для моделі №3 зображена на рисунку 3.15.

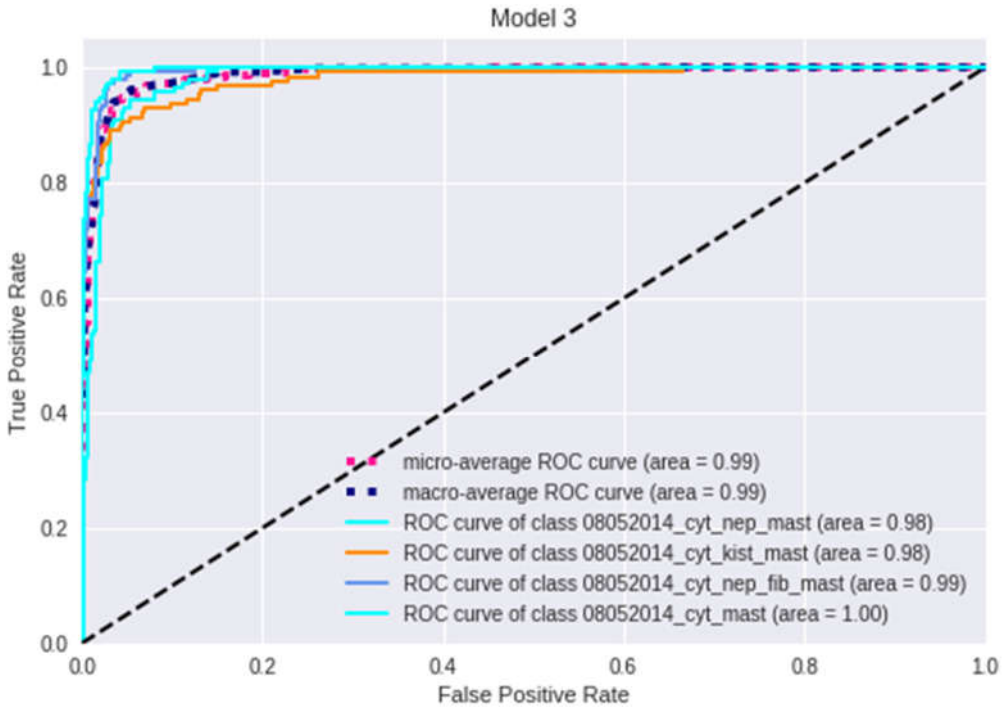

Рисунок 3.15 – Крива для моделі №3

ROC-крива для моделі №4 зображена на рисунку 3.16.

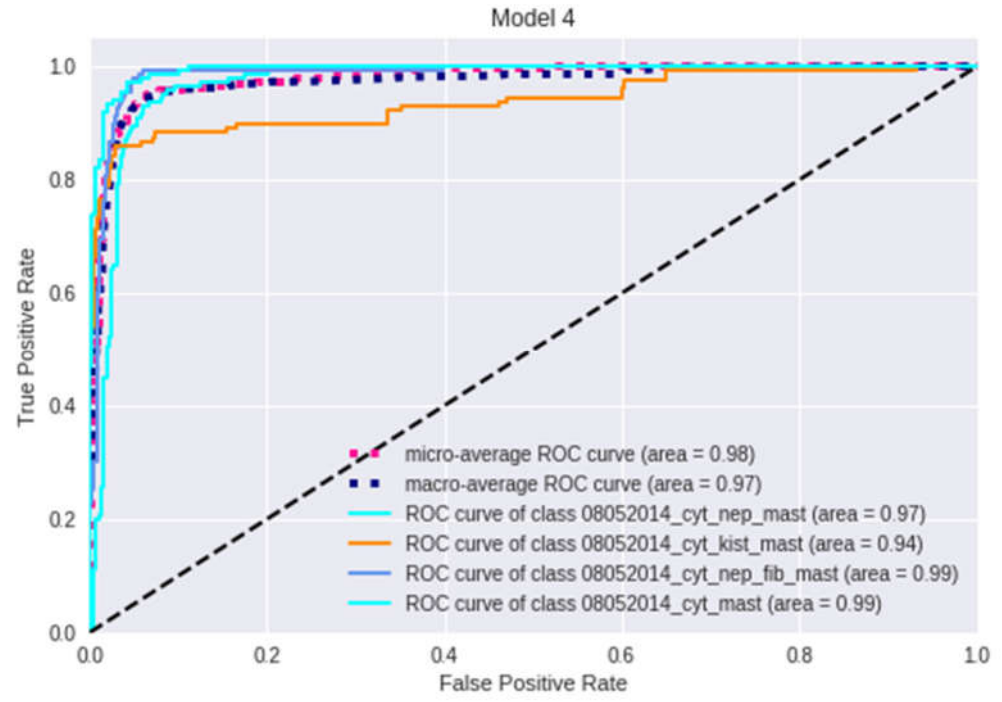

Рисунок 3.16 – Крива для моделі №4

ROC-крива для моделі №5 зображена на рисунку 3.17.

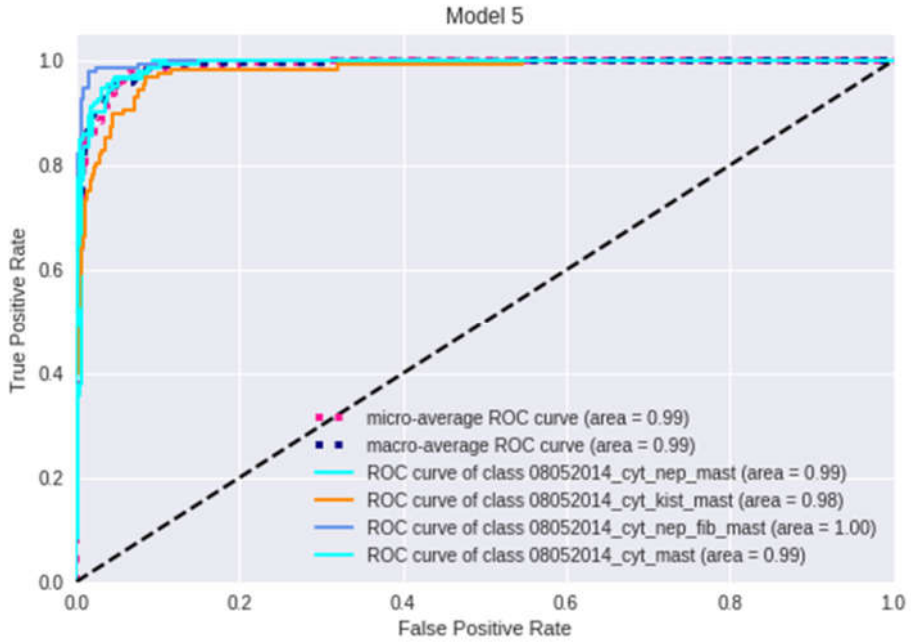

Рисунок 3.17 – Крива для моделі №5

ROC-крива для моделі №6 зображена на рисунку 3.18.

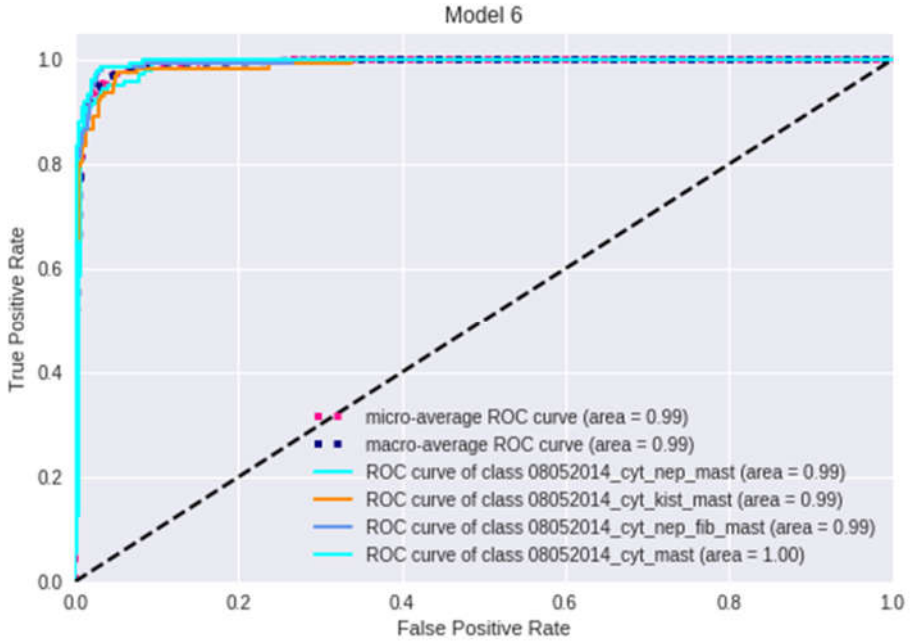

Рисунок 3.18 – Крива для моделі №6

У третьому розділі було проведено опис програмної реалізації поставленого завдання.

ВИСНОВКИ

1. Здійснено аналіз засобів та технологій, алгоритмів та методів синтезу структур згорткових нейронних мереж, що показало актуальність програмної реалізації синтезу структур ЗНМ.

2. Проаналізовано алгоритми синтезу структур ЗНМ: алгоритм повного перебору, алгоритм рандомного (випадкового) пошуку та генетичні алгоритми, що дало змогу реалізувати їх програмно.

3. Розроблено генератор структур ЗНМ.

4. Програмно реалізовано автоматизований синтез структур згорткових нейронних мереж, що дало змогу синтезувати структури ЗНМ з мінімальним втручанням користувача.

5. Проведено тестування розробленого програмного продукту, яке дало точність класифікації, в середньому, 95%.

6. Результати роботи використані в госпдоговірній науково-дослідній роботі на тему «Нейромережеві методи і засоби класифікації зображень ауто- та ксеногенних тканин» (державний реєстраційний номер 0119U103227).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Березький О.М., Дубчак Л.О., Мельник Г.М. Методичні рекомендації до виконання випускної кваліфікаційної роботи з освітнього ступеня “Магістр”. Спеціальність: 123 - Комп’ютерна інженерія. Магістерська програма - Комп’ютерна інженерія". Під ред. О.М. Березького. Тернопіль: ТНЕУ, 2019. 41 с.

2. Гураль І.В., Дубчак Л.О. Методичні вказівки до оформлення курсових проектів, звітів про проходження практики, випускних кваліфікаційних робіт для студентів спеціальності «Комп’ютерна інженерія». Під ред. О.М. Березького. Тернопіль: ТНЕУ, 2019. 33 с.

3. Сверточная нейронная сеть, часть 1: структура, топология, функции активации и обучающее множество: веб-сайт. URL: https://habr.com/ru/post/348000/ (дата звернення: 16.10.2018).

4. Лящинський П. Б., Лящинський П.Б. Автоматизований синтез структур згорткових нейронних мереж. Problèmes et perspectives d'introduction de la recherche scientifique innovante. 2019. Вип. 2. С. 104–105.

5. Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS: веб-сайт. URL: https://arxiv.org/abs/1912.06059 (дата звернення: 12.12.2019).

6. Березький О. М., Піцун О.Й., Лящинський П.Б., Лящинський П.Б., Мельник Г.М. Інтелектуальна система автоматизованої мікроскопії аналізу гістологічних та цитологічних зображень. Штучний інтелект, Київ, 2017. Вип. 2 (76). С. 128-140.

7. Berezsky O., Pitsun O., Dubchak L., Liashchynskyi P., Liashchynskyi P. GPU-based biomedical image processing. 14th International Conference on Perspective Technologies and Methods in MEMS Design, MEMSTECH 2018 – Proceedings – 96-99 pp.

8. Свідоцтво про реєстрацію авторського права на твір №75360. Комп'ютерна програма «Інтелектуальна система діагностування передракових станів молочної залози на основі аналізу гістологічних та цитологічних зображень "HIAMS"». / О.М. Березький, О.Й. Піцун, Г.М. Мельник, П.Б. Лящинський, П.Б. Лящинський. Дата реєстрації 14.12.2017 р.

9. Березький О.М., Лящинський П.Б., Лящинський П.Б, Сухович А.Р., Долинюк Т.М. Синтез біомедичних зображень на основі генеративно-змагальних мереж. УЖІТ. 2019. Вип. 1. С. 123-132.

10. Березький О.М., Піцун О.Й., Боднар А.Р., Долинюк Т.М. Класифікація гістологічних та цитологічних зображень на основі згорткових нейронних мереж. Штучний інтелект. Київ, 2017. Вип. 1 (75). С. 33-42.

11. Березький О. М., Батько Ю.М., Березька К.М., Вербовий С.О., Дацко Т.В., Дубчак Л.О., Ігнатєв І.В., Мельник Г.М., Николюк В.Д., Піцун О.Й. Методи, алгоритми і програмні засоби опрацювання біомедичних зображень. Тернопіль: Економічна думка, ТНЕУ, 2017. 330 с.

12. Березький О.М., Піцун О.Й. Засоби класифікації біомедичних зображень на основі нейронних мереж. Науковий вісник НЛТУ України, 2018, Т. 28, Вип. 9.

13.Згорткова нейронна мережа: веб-сайт. URL: https://uk.wikipedia.org/wiki/Згорткова_нейронна_мережа (дата звернення: 21.10.2018).

14. Elsken T., Metzen J., Hutter F. Neural Architecture Search: A Survey. Journal of Machine Learning Research. 2019. No.20. P. 1–21.

15. Baker B., Gupta O., Naik N., Raskar R. Designing Neural Network Architectures Using Reinforcement Learning. ICLR. 2017.

16. Cai H., Chen T., Zhang W. Efficient Architecture Search by Network Transformation. AAAI. 2017. No.18.

17. Pham H., Guan M., Zoph B. Efficient Neural Architecture Search via Parameter Sharing. 2018.

18. Kandasamy K., Neiswanger W., Schneider J. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. Carnegie Mellon University.

19. Sun Y., Xue B., Zhang M., Yen G. Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification. arXiv. 2019.

20. Reiling A. J., Convolutional neural network optimization using genetic algorithms. The School of Engineering of the University of Dayton. 2017.

21.Keras: веб-сайт. URL: https://uk.wikipedia.org/wiki/Keras (дата звернення: 30.11.2018).

22. What is Keras? The deep neural network API explained: веб-сайт. URL: https://www.infoworld.com/article/3336192/what-is-keras-the-deep-neural-network-api-explained.html (дата звернення: 25.12.2018).

23. Brock A., Lim T. SMASH: One-shot model architecture search through hypernetworks. International Conference on Learning Representations. 2018.

24. TensorFlow: веб-сайт. URL: https://uk.wikipedia.org/wiki/TensorFlow (дата звернення: 13.01.2019).

25. What is TensorFlow? Introduction, Architecture & Example: веб-сайт. URL: https://www.guru99.com/what-is-tensorflow.html (дата звернення: 25.01.2019).

26.Model selection: веб-сайт. URL: https://en.wikipedia.org/wiki/Model_selection (дата звернення 12.02.2019).

27. Aho K., Derryberry D., Peterson T. Model selection for ecologists: the worldviews of AIC and BIC. Ecology. 2014. No.95. P. 631–636.

28.Пошук по ґратці: веб-сайт. URL: https://uk.wikipedia.org/wiki/Оптимізація_гіперпараметрів#Пошук_по_ґратці (дата звернення: 28.02.2019).

29.Прокляття розмірності: веб-сайт. URL: https://uk.wikipedia.org/wiki/Прокляття_розмірності (дата звернення: 28.02.2019).

30. How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras: веб-сайт. URL: https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/ (дата звернення: 5.03.2019).

31.Scikit Learn: веб-сайт. URL: https://scikitlearn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html#sklearn.grid_search.GridSearchCV (дата звернення: 15.03.2019).

32. Use Keras Deep Learning Models with Scikit-Learn in Python: веб-сайт. URL: https://machinelearningmastery.com/use-keras-deep-learning-models-scikit-learn-python/ (дата звернення: 15.03.2019).

33.Випадковий пошук: веб-сайт. URL: https://uk.wikipedia.org/wiki/Оптимізація_гіперпараметрів#Випадковий_пошук (дата звернення: 20.03.2019).

34. Li L., Talwalkar A. Random Search and Reproducibility for Neural Architecture Search. 2019.

35. Olson R. S., Moore J. H. Tpot: A tree-based pipeline optimization tool for automating machine learning. Workshop on Automatic Machine Learning. 2016.

36. Bergstra J., Bengio Y. Random search for hyper-parameter optimization. Journal of Machine Learning Research. 2012. No.13. P. 281–305.

37. Feurer M., Klein A., Blum M. Efficient and robust automated machine learning. Advances in Neural Information Processing Systems. 2015.

38. Snoek J., Adams R. Practical bayesian optimization of machine learning algorithms. Advances in Neural Information Processing Systems. 2012.

39. Liu H., Yang Y. DARTS: Differentiable architecture search. International Conference on Learning Representations. 2019.

40. Li L., Jamieson K. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. International Conference on Learning Representation. 2017. No.17.

41. Kandasamy K., Dasarathy G. Gaussian process bandit optimisation with multi-fidelity evaluations. Advances in Neural Information Processing Systems. 2016.

42. Klein A., Falkner S. Fast bayesian optimization of machine learning hyperparameters on large datasets. International Conference on Artificial Intelligence and Statistics. 2017.

43. Swersky K., Snoek J. Multi-task bayesian optimization. Advances in Neural Information Processing Systems. 2013.

44. Bengio Y. Gradient-based optimization of hyperparameters. Neural Computation. 2000.

45. Maclaurin D., Duvenaud D., Adams R. Gradient-based hyperparameter optimization through reversible learning. International Conference on Machine Learning. 2015.

46. Bergstra J. Algorithms for hyper-parameter optimization. Advances in Neural Information Processing Systems. 2011.

47. Hutter F., Hoos H., Leyton-Brown K. Sequential model-based optimization for general algorithm configuration. Proc. of LION-5. 2011.

48. Real E., Moor S. Large-scale evolution of image classifiers. ICML. 2017.

49. Real E., Aggarwal A. Regularized Evolution for Image Classifier Architecture Search. 2018.

50. Zoph B., Le Q. V. Neural Architecture Search with Reinforcement Learning. International Conference on Learning Representation. 2017.

51. Zoph B., Vasudevan V. Learning transferable architectures for scalable image recognition. Conference on Computer Vision and Pattern Recognition. 2018.

52. Falkner S., Klein A., Hutter F. Bohb: Robust and efficient hyperparameter optimization at scale. International Conference on Machine Learning. 2018.

53. Domhan T., Springenberg J.T., Hutter F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. International Joint Conferences on Artificial Intelligence. 2015.

54. Golovin D., Sonik B. Google vizier: A service for black-box optimization. SIGKDD Conference on Knowledge Discovery and Data Mining. 2017.

55. Cai H., Yang J. Path-level network transformation for efficient architecture search. International Conference on Machine Learning. 2018.

56. Elsken T., Metzen J.H., Hutter F. Multi-objective Architecture Search for CNNs. 2018.

57. Jin H., Song Q., Hu X. Auto-Keras: Efficient Neural Architecture Search with Network Morphism. 2018.

58. Brock A., Lim T. SMASH: One-shot model architecture search through hypernetworks. International Conference on Learning Representations. 2018.

59. Liu C., Zoph B., Neumann M. Progressive Neural Architecture Search. European Conference on Computer Vision. 2018.

60. Zhang C., Ren M., Urtasun R. Graph hypernetworks for neural architecture search. International Conference on Learning Representations. 2019.

61. Bender G., Kindermans P.J., Zoph B. Understanding and simplifying one-shot architecture search. International Conference on Machine Learning. 2018.

62. Cai H., Zhu I., Han S. ProxylessNAS: Direct neural architecture search on target task and hardware. International Conference on Learning Representations. 2019.

63.Генетичний алгоритм: веб-сайт. URL: https://uk.wikipedia.org/wiki/Генетичний_алгоритм (дата звернення: 14.08.2019).