МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА[1]

Офіцеров Олександр Євгенійович
(прізвище, ім'я, по батькові здобувача)

на тему «Розробка нейронної мережі та інтерфейсу для розпізнавання цифр, букв та геометричних фігур»
(повна назва теми)

за матеріалами праць провідних спеціалістів з розробки ПЗ та проектування БД
(повна назва бази дослідження)

науковий керівник д.т.н., професор _____ Зеленський О.С.
(наук. ступінь, вчене звання) *(підпис)* *(прізвище, ініціали)*

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 11.06.2025р. № 12
Завідувач кафедри _____
(підпис)

д.т.н., професор Зеленський О.С.
Наук. ступень, вчене звання *Ініціали, прізвище*

Кривий Ріг – 2025

[1] Назва кваліфікаційної роботи відповідно до ОПП

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

«ЗАТВЕРДЖУЮ»
Завідувач кафедри _____ Зеленський О.С.
(підпис) (Прізвище, ініціали)
«11» червня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ[2]

1. Тема роботи «Розробка нейронної мережі та інтерфейсу для розпізнавання цифр букв та геометричних фігур»
Керівник роботи д.т.н., професор Зеленський О.С.
затверджені наказом закладу вищої освіти від «04» квітня 2025 р. № 222-ст

2. Строк подання здобувачем роботи до «09» червня 2025 р.

3. Зміст кваліфікаційної роботи, об'єкт, предмет та мета дослідження:

Розділ 1. Аналіз існуючих методів розпізнавання образів та нейронних мереж

Розділ 2. Проектування архітектури нейронної мережі

Розділ 3. Розробка програмного забезпечення з графічним інтерфейсом користувача

Розділ 4. Тестування, оптимізація та оцінка результатів функціонування нейронної мережі

Об'єкт дослідження: нейронні мережі та їх застосування для розпізнавання образів

Предмет дослідження: розробка та вдосконалення алгоритмів навчання нейронної мережі для розпізнавання цифр, букв та геометричних фігур

Мета кваліфікаційної роботи: *створення ефективної нейронної мережі з інтуїтивним інтерфейсом для розпізнавання образів*

5. Дата видачі завдання «04» квітня 2025 р.

[2] Назва кваліфікаційної роботи відповідно до ОПП

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів МДР	Строк виконання етапів роботи	Відмітка керівника про виконання етапів (дата, підпис)
1	Підготовка розділу 1	04.04.2025-13.04.2025	
2	Підготовка розділу 2	14.04.2025-26.04.2025	
3.	Підготовка розділу 3	27.04.2025-15.05.2025	
4	Підготовка розділу 4	16.05.2025-08.06.2025	
5	Реєстрація завершеної кваліфікаційної роботи	09.06.2025	Реєстраційний №___ «09»червня 2025 р.
6	Отримання відгуку від наукового керівника	10.06.2025	
7	Подання кваліфікаційної роботи на перегляд завідувачу кафедри	11.06.2025	
8	Отримання зовнішньої рецензії	12.06.2025	
9	Попередній захист кваліфікаційної роботи на кафедрі	13.06.2025	
10	Підготовка до захисту в ЕК	16.06.2025-21.06.2025	

Завдання підготував науковий керівник _____ Зеленський О.С.

(підпис) (прізвище та ініціали)

Завдання одержав _____ Офіцеров О. Є.

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

на кваліфікаційну бакалаврську роботу

«Розробка нейронної мережі та інтерфейсу для розпізнавання цифр, букв та геометричних фігур»

Офіцеров Олександр Євгенійович

Кваліфікаційна бакалаврська робота на здобуття освітньо-кваліфікаційного рівня бакалавра зі спеціальності 121 «Інженерія програмного забезпечення» – Державний університет економіки і технологій – Кривий Ріг, 2025.

У бакалаврській роботі розроблено програмний комплекс для розпізнавання цифр, літер та геометричних фігур на основі штучної нейронної мережі. Реалізовано графічний інтерфейс користувача на базі Qt з можливістю малювання, відображення результатів класифікації та збереження моделей. Для зберігання вагів використано базу даних SQLite.

Ключові слова: нейронна мережа, розпізнавання образів, машинне навчання, графічний інтерфейс, Qt, C++, SQLite, EMNIST, навчання з учителем, збереження моделі, серіалізація, штучний інтелект.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД	(база даних) Упорядкований набір логічно пов’язаних даних для зберігання, пошуку та обробки
СУБД	(система управління базами даних) Програмне забезпечення для створення, редагування та запиту баз даних
ПЗ	(програмне забезпечення) Сукупність програм, що забезпечують функціонування комп’ютерних систем
GUI	(Graphical User Interface) Графічний інтерфейс користувача; інтерфейс для взаємодії з програмою за допомогою графічних елементів
Qt	Кросплатформенний фреймворк для розробки графічних інтерфейсів та застосунків на C++
CNN	(Convolutional Neural Network) Згорткова нейронна мережа; ефективна для обробки зображень
DNN	(Deep Neural Network) Глибока нейронна мережа з кількома прихованими шарами для складних задач
RNN	(Recurrent Neural Network) Рекурентна нейронна мережа для послідовних даних з пам’яттю попередніх станів
SGD	(Stochastic Gradient Descent) Стохастичний градієнтний спуск; алгоритм оптимізації вагів нейромережі
MSE	(Mean Squared Error) Середньоквадратична похибка; функція втрат для оцінки точності моделі
CSV	(Comma-Separated Values) Формат табличних даних у текстовому вигляді з розділенням комами
BLOB	(Binary Large Object) Тип даних у БД для збереження великих бінарних об’єктів, зокрема вагів

ЗМІСТ

ВСТУП

Сучасний світ стрімко розвивається, і технології штучного інтелекту, зокрема нейронні мережі, відіграють у цьому процесі ключову роль. Нейронні мережі знаходять застосування в різних сферах — від медицини до фінансів, а їхня здатність до розпізнавання образів відкриває нові горизонти для автоматизації та покращення якості життя. Однією з найактуальніших задач у цій галузі є розпізнавання цифр, букв та геометричних фігур, що має широке практичне застосування, наприклад, у системах обробки документів, комп'ютерному зорі та робототехніці.

Об’єктом дослідження цієї роботи є нейронні мережі та їхнє застосування для розпізнавання образів.

Предмет дослідження — розробка та вдосконалення алгоритмів навчання нейронної мережі для ефективного розпізнавання цифр, букв та геометричних фігур.

Метою кваліфікаційної роботи є створення ефективної нейронної мережі з інтуїтивним графічним інтерфейсом користувача, яка здатна розпізнавати образи з високою точністю. Для досягнення цієї мети було визначено такі завдання:

1. Провести аналіз існуючих методів розпізнавання образів та нейронних мереж.
2. Спроектувати архітектуру нейронної мережі, оптимальну для поставлених задач.
3. Розробити програмне забезпечення з графічним інтерфейсом для взаємодії з користувачем.
4. Провести тестування та оптимізацію роботи нейронної мережі для підвищення її точності та швидкодії.

У **першому розділі** роботи проведено аналіз сучасних методів розпізнавання образів, зокрема з використанням нейронних мереж, та визначено їхні переваги та недоліки. У **другому розділі** описано процес проектування архітектури нейронної мережі, обґрунтовано вибір її параметрів та алгоритмів

навчання. **Третій розділ** присвячений розробці програмного забезпечення, зокрема графічного інтерфейсу, який забезпечує зручну взаємодію користувача з нейронною мережею. У **четвертому розділі** наведено результати тестування мережі, проаналізовано її ефективність та запропоновано шляхи оптимізації.

Робота має як теоретичне, так і практичне значення. Теоретичні результати доповнюють наукові знання про методи навчання нейронних мереж для розпізнавання образів, а практична реалізація може бути використана у реальних проектах, наприклад, для автоматизації обробки документів або в освітніх цілях.

У сучасному світі, де інформація грає ключову роль, розробка ефективних інструментів для її обробки є надзвичайно актуальною. Ця робота спрямована на створення такого інструменту, який поєднує в собі передові технології нейронних мереж та зручний інтерфейс для користувача.

РОЗДІЛ 1
АНАЛІЗ МЕТОДІВ РОЗПІЗНАВАННЯ ОБРАЗІВ І НЕЙРОННИХ МЕРЕЖ

1.1. Теоретичні засади розпізнавання образів

Розпізнавання образів є фундаментальним завданням в галузі штучного інтелекту, машинного навчання та комп'ютерного зору. Його мета полягає у класифікації або інтерпретації вхідних візуальних даних, таких як зображення чи відео, шляхом ідентифікації об'єктів, символів, фігур або тексту. Такі технології знаходять застосування в медицині (автоматична діагностика), транспорті (системи розпізнавання дорожніх знаків), фінансовому секторі (автоматизоване зчитування документів), промисловості (контроль якості) тощо.

Методи розпізнавання образів поділяються на традиційні алгоритмічні підходи та сучасні методи, що базуються на нейронних мережах. Перші зазвичай використовують ручне виділення ознак та алгоритми класифікації, тоді як другі дозволяють автоматизувати процес навчання розпізнавання, використовуючи великі обсяги даних і складні моделі.

1.2. Традиційні методи розпізнавання образів

Традиційні підходи були домінуючими до поширення глибокого навчання. Вони охоплюють наступні основні методи:

Метод шаблонного порівняння

Цей метод передбачає порівняння вхідного зображення з набором заздалегідь підготовлених еталонних шаблонів. Незважаючи на простоту реалізації, метод є чутливим до змін масштабу, обертання та перекручення зображень, що обмежує його ефективність у практичних умовах.

Метод виділення ознак

Використовує алгоритми для знаходження ключових точок (наприклад, кути, краї, текстури) із подальшою класифікацією за допомогою таких моделей,

як SVM (машина опорних векторів). Широко застосовуються SIFT (Scale-Invariant Feature Transform) та SURF (Speeded-Up Robust Features). Основна складність полягає у ручному підборі релевантних ознак.

Методи на основі гістограм

Ці методи будують гістограми кольорів або яскравості, аналізуючи їх розподіл. Вони є ефективними для класифікації простих візуальних характеристик, проте малопридатні для складних зображень або багато форматних структур.

Приклад розпізнавання цифри методом шаблонного порівняння, а також розпізнавання за допомогою методу виділення ознак наведено на Рис. 1.1.

Рис. 1.1. Порівняльна ілюстрація: шаблонне порівняння vs метод ознак (сітка з прикладами вхідного зображення та відповідних реакцій алгоритму)

Традиційні методи розпізнавання образів, попри їхню історичну значущість та простоту реалізації, мають ряд обмежень, пов’язаних із жорсткою залежністю від геометричних трансформацій та потребою в ручному налаштуванні параметрів. Як видно з таблиці 1.1, ці методи здатні забезпечити прийнятний рівень точності лише для відносно простих або добре структурованих зображень. Ілюстрація на Рис. 1.1 наочно демонструє різницю в підходах, що підтверджує необхідність переходу до сучасних автоматизованих систем, зокрема нейронних мереж.

Таблиця 1.1

Таблиця 1.1 Порівняльна характеристика традиційних методів

Метод	Переваги	Недоліки	Типи зображень	Достовірність
Шаблонне порівняння	Простота реалізації	Нестійкий до змін масштабу, поворотів та деформацій	Статичні символи, прості форми	Низька (~70–80%)
Ознаковий метод (SIFT, SURF)	Стійкість до масштабування, поворотів, освітлення	Необхідність ручного налаштування ознак, повільна обробка	Об'єкти з чіткими геометричними ознаками	Середня (~80–90%)
Методи гістограм	Ефективні при класифікації за кольором або яскравістю	Не підходять для складних форм і контурів	Кольорові або одноколірні зображення	Середня (~75–85%)

У таблиці 1.1, наведено порівняльну характеристику основних традиційних методів розпізнавання образів, зокрема шаблонного порівняння, ознакових методів (наприклад, SIFT, SURF) та методів на основі гістограм. Представлено ключові аспекти кожного підходу, такі як переваги, недоліки, типи зображень, для яких метод є найбільш придатним, а також приблизна достовірність розпізнавання.

1.3. Нейронні мережі як сучасний інструмент розпізнавання

З появою глибокого навчання розпізнавання образів перейшло на новий рівень. Нейронні мережі мають здатність до автоматичного виявлення релевантних ознак, що значно підвищує точність і знижує необхідність у ручній обробці.

Згорткові нейронні мережі (CNN)

CNN є найбільш ефективними для обробки візуальних даних. Вони складаються з згорткових, підвибіркових та повнозв'язних шарів, що дає змогу витягувати як локальні, так і глобальні ознаки зображення. Відомі архітектури, такі як LeNet, AlexNet, VGG16, ResNet, показали надзвичайно високу ефективність у класифікаційних задачах.

Рекурентні нейронні мережі (RNN)

Ці мережі використовуються для роботи з послідовними даними, як-от рукописний текст або мовні сигнали. Модифікація LSTM забезпечує збереження довготривалої залежності між елементами вхідної послідовності.

Повнозв'язні нейронні мережі (DNN)

Це базова форма нейронної мережі, що складається з послідовних шарів, де кожен нейрон з'єднаний з усіма нейронами попереднього шару. Хоча DNN мають обмежену здатність до аналізу зображень у "сирому" вигляді, їх можна ефективно використовувати після попереднього виділення ознак.

Аналіз нейронних мереж як інструменту розпізнавання образів підтверджує їхню ключову роль у сучасних системах штучного інтелекту. Зокрема, згорткові нейронні мережі (CNN) продемонстрували найвищу ефективність у візуальних задачах, оскільки здатні автоматично витягувати важливі ознаки зображення без потреби в ручному втручанні. Їхня модульна архітектура дозволяє працювати як з простими, так і з надзвичайно складними структурами даних, досягаючи високої точності розпізнавання.

Рекурентні нейронні мережі (RNN), особливо їх модифікації на кшталт LSTM, займають важливе місце в обробці послідовних вхідних даних, де

критичною є здатність до збереження контексту та послідовності інформації — наприклад, у розпізнаванні тексту або мови.

Повнозв’язні мережі (DNN), хоча й поступаються CNN у завданнях обробки "сирих" зображень, залишаються важливим елементом архітектур, особливо на стадіях класифікації або при роботі з попередньо обробленими даними. Вони є універсальними та добре інтегруються з іншими видами мереж, наприклад, як вихідні шари у CNN або гібридних моделях.

Порівняльна таблиця 1.2 відображає сильні та слабкі сторони кожного підходу, зважаючи на призначення, продуктивність, складність реалізації та точність. Залежно від конкретного типу задачі, можливе використання окремого типу нейронної мережі або їх поєднання. Таким чином, нейронні мережі не тільки забезпечують високу якість розпізнавання, а й відкривають перспективи для створення гнучких та адаптивних інтелектуальних систем.

Таблиця 1.2

Таблиця порівняння типів нейронних мереж

Тип мережі	Призначення	Продуктивність	Складність реалізації	Потреба в даних	Точність розпізнавання
DNN	Загальні завдання класифікації, регресії	Середня	Помірна	Середня	Від середньої до високої (~85–92%)
CNN	Обробка зображень, відео, медичних сканів	Висока (GPU-оптимізована)	Висока (архітектурна складність)	Висока	Висока (~90–98%)
RNN	Обробка послідовностей: мова, текст, часові ряди	Середня (проблеми градієнтів)	Висока (циклічні з'єднання)	Висока	Середня (~75–90%)

У таблиці 1.2 представлено порівняльну характеристику основних типів нейронних мереж, що використовуються для завдань розпізнавання образів. Зокрема, розглянуто повнозв’язні (DNN), згорткові (CNN) та рекурентні (RNN) мережі.

Огляд існуючих рішень для розпізнавання цифр, букв і фігур

Упродовж останніх десятиліть було створено велику кількість рішень для задач розпізнавання образів, зокрема рукописних цифр, літер та простих геометричних фігур. Ці рішення активно використовуються в навчанні, тестуванні та оцінюванні продуктивності алгоритмів машинного навчання, зокрема нейронних мереж. У цьому підрозділі розглянемо найбільш відомі бази даних, а також підходи, які застосовуються в сучасних дослідницьких і прикладних проектах.

MNIST

MNIST — набір зображень рукописних цифр (28×28 пікселів), що містить 60 000 навчальних і 10 000 тестових прикладів. Завдяки простоті та стандартизації широко використовується для перевірки моделей машинного навчання. Сучасні нейронні мережі досягають точності понад 99%. Приклад рукописних цифр зображень представлено на Рис. 1.2.

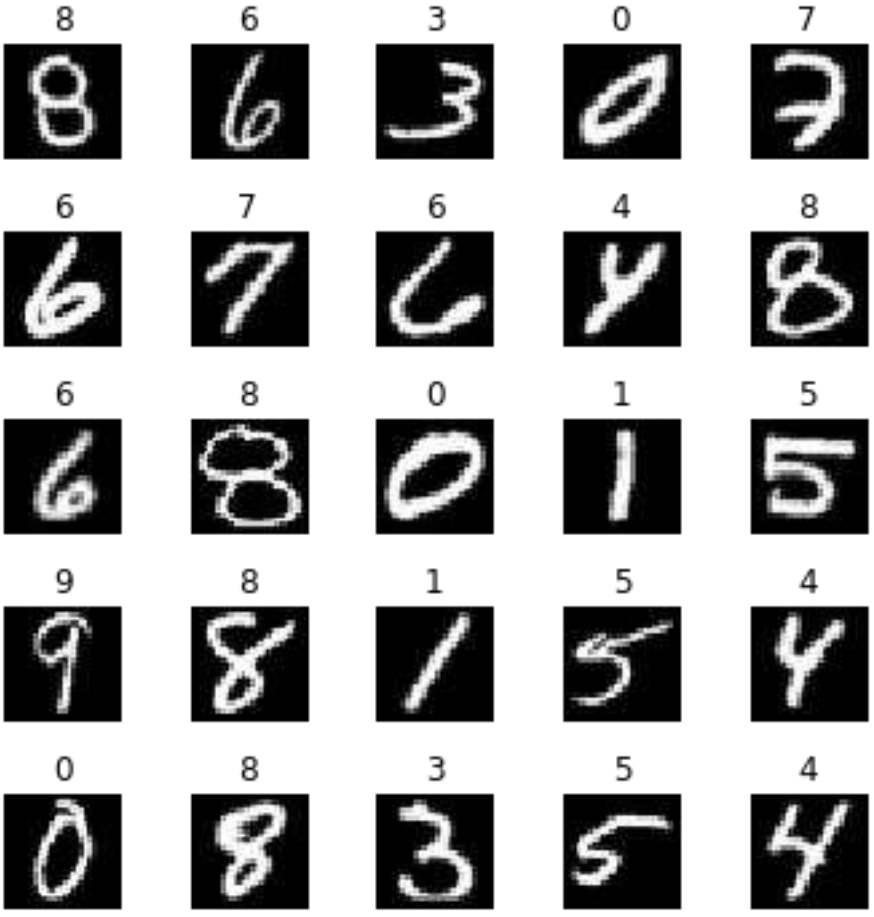


Рис. 1.2. Зображення цифр для навчання нейронних мереж

EMNIST

EMNIST (Extended MNIST) — розширений варіант MNIST, що включає не лише рукописні цифри, але й латинські літери як у верхньому, так і в нижньому регістрах. Цей набір створено з метою розширити функціональність MNIST та дозволити дослідникам і розробникам тренувати та тестувати моделі на більш різноманітному наборі символів.

Залежно від варіації (EMNIST-Balanced, EMNIST-ByClass, EMNIST-Letters тощо), кількість класів може становити до 62 (10 цифр + 26 великих літер + 26 малих літер). Наявність такого обсягу класів ускладнює задачу класифікації, водночас забезпечуючи вищу релевантність для реальних прикладних задач, таких як розпізнавання заповнених форм або рукописного тексту в освітніх та адміністративних системах.

Ілюстративні зразки рукописних букв із датасету представлено на Рис. 1.3.

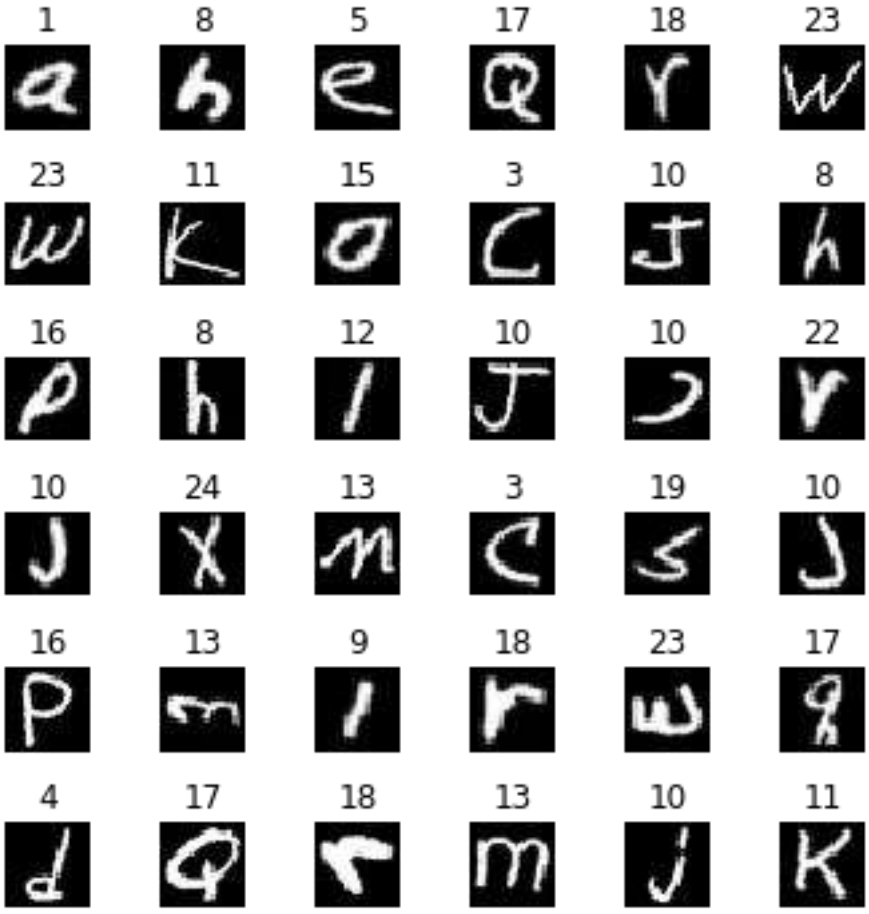


Рис. 1.3. Представлено типові приклади зображень, що входять до складу набору EMNIST

Рішення з розпізнавання геометричних фігур

Розпізнавання простих геометричних фігур (кола, квадрата, трикутника, прямокутника тощо) є окремим напрямом, для якого зазвичай використовуються штучно згенеровані набори зображень із варіаціями форм, розмірів і положення. Це дозволяє тренувати моделі на контрольованих синтетичних даних.

Для таких задач ефективно застосовуються згорткові нейронні мережі (CNN), зокрема з використанням трансферного навчання на основі архітектур VGG16, ResNet50 або MobileNet. Такі моделі демонструють високу точність навіть за наявності геометричних викривлень.

У деяких розробках розпізнавання фігур поєднується з оцінкою кількості об’єктів або їх розташування, що розширює можливості застосування в комп’ютерному зорі, освіті та промисловості.

Висновки до розділу 1

Традиційні методи розпізнавання образів, попри їхню історичну значущість та простоту реалізації, мають ряд обмежень, пов’язаних із жорсткою залежністю від геометричних трансформацій та потребою в ручному налаштуванні параметрів. Як видно з таблиці 1.1, ці методи здатні забезпечити прийнятний рівень точності лише для відносно простих або добре структурованих зображень. Ілюстрація на Рис. 1.1 наочно демонструє різницю в підходах, що підтверджує необхідність переходу до сучасних автоматизованих систем, зокрема нейронних мереж. У дипломній роботі буде реалізована повнозв’язна нейронна мережа з можливістю навчання для розпізнавання символів та фігур, а також інтерактивний інтерфейс на базі Qt з відображенням результатів розпізнавання.

РОЗДІЛ 2
ПРОЕКТУВАННЯ АРХІТЕКТУРИ НЕЙРОННОЇ МЕРЕЖІ

2.1. Вибір архітектури нейронної мережі

На сучасному етапі розвитку інтелектуальних систем розпізнавання образів, вибір архітектури нейронної мережі є критично важливим етапом, що визначає ефективність, швидкодію та точність обробки вхідних даних. У рамках даної роботи, яка присвячена розпізнаванню цифрових символів, букв латинського алфавіту та геометричних фігур, було прийнято рішення використовувати повнозв'язну нейронну мережу (Fully Connected Neural Network, або DNN — Deep Neural Network).

Основною причиною такого вибору є поєднання простоти реалізації та здатності до навчання на помірних обсягах даних. DNN має відносно нескладну структуру: вона складається з набору послідовних шарів нейронів, де кожен нейрон поточного шару пов'язаний із кожним нейроном попереднього шару. Такий підхід дозволяє мережі гнучко навчатися та знаходити складні залежності між ознаками навіть без попереднього виділення ознак вручну.

Переваги використання DNN у межах поставленої задачі:

- Універсальність та простота реалізації. Повнозв'язні мережі легко реалізуються в будь-якому програмному середовищі, зокрема на C++.
- Придатність для задач середньої складності. У випадку розпізнавання відносно простих зображень розміром 28×28 пікселів (наприклад, рукописні цифри або літери), DNN може показувати точність, достатню для практичного застосування.
- Низькі обчислювальні вимоги. На відміну від згорткових мереж (CNN), які є значно ефективнішими для зображень, DNN не потребує GPU-обчислень і може ефективно працювати навіть на середньої потужності комп'ютерах.

Архітектура побудованої нейронної мережі має наступний вигляд:

- Вхідний шар: кількість нейронів — 784, що відповідає кількості пікселів у зображенні розміром 28×28. Кожен піксель інтерпретується як окреме число (після нормалізації).
- Приховані шари:
  - Для задач розпізнавання цифр та букв використано два шари перший має 200 нейронів, другий — 150.
  - Для розпізнавання геометричних фігур достатньо одного прихованого шару на 200 нейронів d.
- Активаційна функція: сигмоїдна функція (sigmoid) була обрана за її плавну нелінійність та здатність обмежувати вихідні значення у діапазоні [0,1].
- Вихідний шар:
  - 10 нейронів для класифікації цифр від 0 до 9,
  - 26 нейронів для латинських букв,
  - 4 нейрони для основних геометричних фігур: коло, квадрат, трикутник, п’ятикутник.

Схематичне зображення архітектури повнозв’язної нейронної мережі подано на Рис. 2.1.

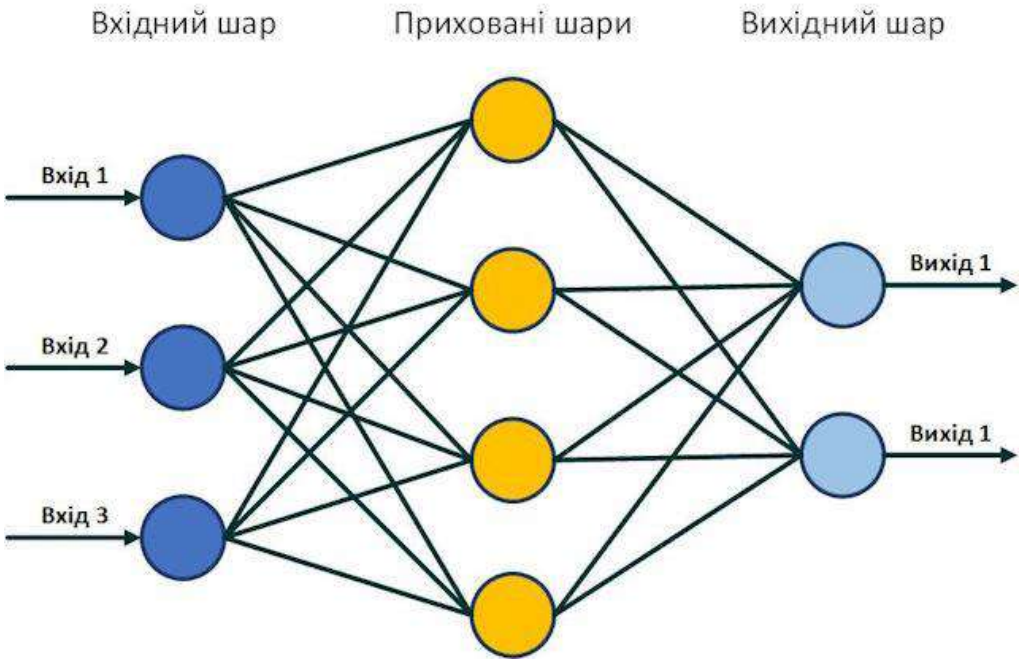


Рис. 2.1. Архітектури повнозв’язної нейронної мережі

2.2. Процес навчання нейронної мережі

Навчання штучної нейронної мережі є одним із найважливіших етапів, що безпосередньо впливає на її здатність до узагальнення нових даних. У цій роботі застосовано класичний підхід — метод зворотного поширення помилки (backpropagation), у поєднанні з алгоритмом градієнтного спуску. Такий підхід дозволяє поступово мінімізувати функцію втрат за рахунок корекції вагових коефіцієнтів, що є основним механізмом навчання моделей глибокого навчання.

Цей процес повторюється ітеративно для кожного прикладу в наборі даних до досягнення бажаної точності.

Методика навчання включає наступні кроки:

- Пряме поширення (feedforward):
  - Нормалізовані вхідні дані подаються на вхідний шар мережі.
  - Кожен нейрон прихованих шарів обчислює зважену суму своїх входів, до якої застосовується активаційна функція.
  - Отримані результати передаються до наступного шару аж до вихідного.
- Обчислення помилки:
  - На вихідному шарі отримане значення порівнюється з очікуваним (еталонним) значенням (міткою класу).
  - Для цього використовується функція втрат. У цьому випадку — середньоквадратична похибка (MSE).
- Зворотне поширення помилки (backpropagation):
  - Помилка поширюється у зворотному напрямку від вихідного шару до вхідного.
  - Вагові коефіцієнти кожного нейрона оновлюються пропорційно градієнту функції помилки.

- Застосовується алгоритм стохастичного градієнтного спуску (SGD), де оновлення ваг відбувається для кожного прикладу або міні-батчу.

- Гіперпараметри навчання:
  - Швидкість навчання (learning rate) було підібрано емпірично для досягнення стабільної та поступової мінімізації функції втрат.
  - Кількість епох (повторів повного навчального набору) — від 20 до 100 залежно від типу за

Приклад реалізації коду на C++ функції прямого поширення (feedforward) наведено на зображенні Рис. 2.2 для кращого розуміння.

```
// Функція feedForwarding виконує пряме поширення (feed-forward) даних через нейронну мережу
// inputs - вхідний масив даних, який передається в мережу
// Повертає вказівник на вихідний масив останнього шару
float* INerualNetwork::feedForwarding(const float* _inputs)
{
    // Проходимо по всіх шарах мережі (від входу до виходу)
    for (int i = 0; i < m_nlCountLayers; i++)
    {
        auto &Layer = m_nLayers[i]; // Поточний шар
        // Обчислюємо значення для кожного нейрона в поточному шарі
        for(unsigned int hid = 0; hid < Layer->getOutCount(); hid++)
        {
            float tmpS = 0.0; // Тимчасова змінна для зберігання суми вагованих входів

            // Сумуємо всі входи поточного нейрона, помножені на відповідні ваги
            for(unsigned int inp = 0; inp < Layer->getInCount(); inp++)
            {
                if(i == 0) // Якщо це перший шар, беремо дані з вхідного масиву
                {
                    tmpS += _inputs[inp] * Layer->getMatrix(inp, hid);
                }
                else // Інакше - беремо дані з попереднього шару
                {
                    tmpS += m_nLayers[i-1]->getHidden()[inp] * Layer->getMatrix(inp, hid);
                }
            }
            // Додаємо зміщення (bias) - це додаткова вага, що не залежить від входу
            tmpS += Layer->getMatrix(Layer->getInCount(), hid);
            // Застосовуємо функцію активації до отриманої суми
            Layer->getHidden()[hid] = activation(tmpS, Layer->getLayer_type());
        }
    }
    // Повертаємо вихід останнього шару (результат роботи мережі)
    return m_nLayers[m_nlCountLayers - 1]->getHidden();
}
```

Рис. 2.2. Функція прямого поширення (feedforward) реалізована на C++

2.3. Зберігання та відновлення моделей

З метою забезпечення можливості багаторазового використання результатів попереднього навчання штучної нейронної мережі, а також для

підтримки широкого спектру експериментів, пов’язаних із варіативністю архітектур, наборів вхідних даних, гіперпараметрів та стратегій навчання, у межах даного програмного забезпечення було реалізовано повноцінну інфраструктуру для збереження, архівації та відновлення моделей. Такий функціонал є критично важливим у контексті досліджень, де потрібно проводити численні серії експериментів з метою виявлення оптимальних конфігурацій.

Наявність засобів збереження дозволяє зафіксувати поточний стан мережі після завершення навчання або на будь-якому його етапі, що в свою чергу дає змогу повернутися до цієї конфігурації у майбутньому — наприклад, для порівняння результатів, повторного тестування або донавчання з новими даними. Це значно скорочує загальний час на експерименти, усуває потребу в повторному навчанні з нуля та підвищує ефективність роботи дослідника або розробника.

Крім того, така система зберігання забезпечує відтворюваність результатів, що є однією з ключових вимог до наукових досліджень у сфері машинного навчання. Можливість точно відновити модель з певним набором ваг і параметрів дозволяє гарантувати, що результати експериментів не є випадковими, а можуть бути стабільно відтворені іншими дослідниками або в інший час.

Таким чином, розроблена інфраструктура збереження та відновлення моделей виступає не лише як технічний компонент, а й як інструмент підтримки наукової достовірності, експериментальної гнучкості та продуктивності в рамках розв’язання задачі розпізнавання образів.

Серіалізація вагів

Оскільки повнозв’язна штучна нейронна мережа складається з набору вагових коефіцієнтів, що визначають зв’язки між нейронами сусідніх шарів, одним із ключових завдань у процесі її збереження є серіалізація цих числових параметрів у надійний та компактний формат. Для досягнення цієї мети у розробленій системі було реалізовано механізм **бінарної серіалізації**, який має низку важливих переваг:

- значне зменшення обсягу файлів у порівнянні з текстовими форматами представлення даних (такими як CSV, JSON або XML);

- висока швидкість зчитування та запису, що є критичним при роботі з великими моделями;
- збереження повної числової точності ваг, без втрат, які можуть виникати під час конвертації у текстові формати або заокруглення.

Бінарна серіалізація охоплює не лише вагові коефіцієнти кожного шару, але й додаткову структурну інформацію, зокрема опис **топології нейронної мережі** (кількість шарів, кількість нейронів у кожному з них тощо). Це дозволяє при десеріалізації (тобто відновленні моделі з файлу або пам'яті) точно реконструювати її початкову архітектуру без необхідності ручного налаштування.

Завдяки такому підходу забезпечується швидкий та безпомилковий перехід між різними етапами роботи з моделлю — навчанням, тестуванням, подальшим донавчанням або використанням у продуктивному середовищі.

Приклад реалізації серіалізації наведено на Рис. 2.3, де показано фрагмент коду, що відповідає за збереження структури та параметрів моделі у бінарному вигляді.

```
// Функція серіалізації нейронної мережі у рядок
// Повертає рядок, який містить структуру мережі та всі її ваги
string INerualNetwork::serialize() const
{
    string retBuf;  // Буфер для зберігання результату серіалізації
    // 1. Серіалізація топології мережі (кількість шарів та їх розміри)
    // ---------------------------------------------------------------
    size_t nbrOfTopoElements = m_NetworkStructure.size() + 1;  // +1 для зберігання кількості шарів
    unsigned int* topoBuf = new unsigned int[nbrOfTopoElements];  // Тимчасовий буфер для топології
    // Перший елемент - кількість шарів у мережі
    topoBuf[0] = m_NetworkStructure.size();
    // Наступні елементи - розміри кожного шару
    for(size_t i = 0; i < m_NetworkStructure.size(); i++)
    {
        topoBuf[i+1] = m_NetworkStructure.at(i);  // Записуємо розмір i-того шару
    }
    // Додаємо топологію до результуючого буфера
    retBuf.append(string((char*)topoBuf, nbrOfTopoElements*sizeof(unsigned int)));
    delete[] topoBuf;  // Тимчасовий буфер більше не потрібен
    // 2. Серіалізація параметрів кожного шару (ваги та зміщення)
    // ----------------------------------------------------------
    for(auto l : m_nLayers)  // Проходимо по всіх шарах мережі
    {
        string lBuf = l->serialize();  // Серіалізуємо поточний шар
        unsigned int lBufSize = lBuf.size();  // Отримуємо розмір серіалізованих даних шару

        // Додаємо розмір даних шару (для подальшого десеріалізації)
        retBuf.append(string((char*)(&lBufSize), 1*sizeof(unsigned int)));

        // Додаємо самі серіалізовані дані шару
        retBuf.append(lBuf);
    }
    return retBuf;  // Повертаємо повністю серіалізовану мережу
}
```

Рис. 2.3. Код серіалізація вагів

Цей фрагмент коду (Рис. 2.3) демонструє серіалізацію топології моделі, яка є ключовою передумовою для подальшого збереження параметрів ваг. Аналогічно, кожен шар мережі серіалізується шляхом додавання до буфера своїх матриць ваг та зміщень.

Збереження у базі даних SQLite

Для централізованого збереження як самої моделі, так і додаткових метаданих, було використано вбудовану реляційну базу даних SQLite. Такий вибір обумовлений простотою інтеграції з Qt, відсутністю потреби у сторонньому серверному ПЗ та хорошою продуктивністю на невеликих об’ємах даних.

Навчальні зразки

У таблиці 2.1 зберігаються як самі зображення, представлені у вигляді масивів пікселів, так і відповідні мітки класів. Ці дані використовуються для навчання та тестування нейронної мережі.

Таблиця 2.1

Таблиця TrainingData

Атрибут	Тип	Опис
id	INTEGER	Первинний ключ, автоінкремент
mode	TEXT	Режим навчання (наприклад, "train", "test")
label	INTEGER	Мітка даних (клас)
inputs	BLOB	Бінарні дані вхідних векторів (масив float)

У таблиці 2.1, наведено структуру бази даних TrainingData, яка використовується для зберігання навчальних зразків нейронної мережі. Кожен запис містить інформацію про режим (наприклад, «train» або «test»), мітку класу, а також вхідні дані у вигляді бінарного масиву пікселів (вектор типу float). Така організація дозволяє ефективно завантажувати дані в нейронну мережу та підтримувати розмежування між режимами навчання і тестування. Первинний ключ id забезпечує унікальність кожного зразка.

Історія тренування

У таблиці 2.2 зберігаються основні параметри кожної епохи навчання: значення функції втрат, точність класифікації, швидкість навчання (learning rate), номер епохи та мітка часу. Це дає змогу відстежувати динаміку навчального процесу та проводити аналіз його ефективності.

Таблиця 2.2

Таблиця TrainingHistory

Атрибут	Тип	Опис
id	INTEGER	Первинний ключ, автоінкремент
timestamp	DATETIME	Час запису (за замовчуванням CURRENT_TIMESTAMP)
mode	TEXT	Режим навчання (наприклад, "train", "validation")
epoch	INTEGER	Номер епохи навчання
accuracy	REAL	Точність на цій епосі
error	REAL	Похибка на цій епосі
learning_rate	REAL	Швидкість навчання

Таблиця 2.2 містить історію тренування нейронної мережі, фіксуючи ключові параметри кожної епохи. Зокрема, зберігаються значення точності (accuracy), похибки (error), швидкості навчання (learning_rate), а також номер епохи (epoch) і час запису (timestamp). Атрибут mode дозволяє розрізняти режими тренування, наприклад «train» або «validation». Завдяки цій структурі можна легко аналізувати ефективність навчання, відстежувати зміни продуктивності моделі та виявляти тенденції на різних етапах тренувального процесу.

Стани моделі

У таблиці 2.3 здійснюється фіксація усіх збережених станів нейронної мережі, що виникають у ході навчання. Кожен запис у цій таблиці містить повну інформацію про конкретну версію моделі: мітку часу моменту збереження, значення точності моделі на певному наборі даних, числове значення похибки, кількість завершених епох, режим навчання, в якому здійснювалося збереження

(наприклад, тренувальний або валідаційний), а також серіалізовані ваги у вигляді бінарних даних.

Збереження таких даних є вкрай важливим для забезпечення повторюваності експериментів, порівняння різних конфігурацій нейронної мережі та відновлення попередніх результатів без необхідності повторного навчання. Крім того, наявність історії збережених моделей дозволяє глибше аналізувати динаміку змін ефективності під час навчального процесу, здійснювати відбір оптимальних параметрів, а також створювати резервні копії важливих версій моделей для подальшого використання або розгортання у прикладних задачах.

Таблиця 2.3

Таблиця SavedModels

Атрибут	Тип	Опис
name	TEXT	Первинний ключ, назва моделі
timestamp	DATETIME	Час збереження (за замовчуванням CURRENT_TIMESTAMP)
mode	TEXT	Режим, у якому була навчена модель
epochs_trained	INTEGER	Кількість епох навчання
accuracy	REAL	Фінальна точність моделі
error	REAL	Фінальна похибка моделі
weights_data	BLOB	Серіалізовані ваги моделі (бінарні дані)

У таблиці 2.3 зберігаються всі збережені стани нейронної мережі, що дозволяє фіксувати прогрес моделі протягом навчання. Кожен запис включає унікальну назву моделі (name), час збереження (timestamp), режим навчання (mode), кількість завершених епох (epochs_trained), фінальні значення точності (accuracy) та похибки (error), а також серіалізовані ваги (weights_data) у вигляді бінарних даних.

Така структура забезпечує можливість збереження та відновлення моделей, аналізу динаміки навчання, порівняння різних конфігурацій, а також повторного використання найкращих варіантів без необхідності повторного навчання. Таблиця є основою для ефективного управління експериментами та результатами моделі.

Відновлення моделей

Під час запуску програми користувач має можливість обрати збережену раніше модель із бази даних або з окремого файлу, після чого система відновлює ваги та структуру нейронної мережі у пам’яті. Це дозволяє:

- продовжити тренування з певної точки (наприклад, у разі передчасного завершення сесії);
- провести повторне тестування моделі на нових даних;
- використати модель у продуктивному середовищі без повторного навчання.

Завдяки збереженню топології та параметрів тренування, процес відновлення є повністю автоматизованим і не потребує ручного налаштування.

2.4. Інтеграція з графічним інтерфейсом

Особливістю цієї роботи є реалізація інтерактивного графічного інтерфейсу користувача (GUI), що забезпечує зручну взаємодію з нейронною мережею на всіх етапах — від побудови архітектури та налаштування параметрів до навчання, тестування та візуалізації результатів. Інтерфейс дозволяє керувати навчальним процесом і виконувати експерименти без потреби взаємодії з кодом.

Інтеграція GUI значно підвищує доступність системи, зокрема для користувачів без програмістських навичок. Основні функції винесено в зручні елементи керування, що дає змогу змінювати параметри, запускати навчання, зберігати/завантажувати моделі та переглядати графіки точності й похибки.

Для створення інтерфейсу використано кросплатформовий фреймворк **Qt**, який забезпечує гнучке компонування, інтерактивність і стабільну роботу на різних операційних системах. Це робить програму придатною як для дослідницьких, так і прикладних завдань.

Основні функції GUI:

- Ручне введення зображень: користувач може намалювати цифру, символ або фігуру, які потім розпізнаються мережею.
- Налаштування параметрів навчання: доступ до ключових змінних, таких як швидкість навчання, кількість епох чи тип активаційної функції.
- Графічне відображення процесу: побудова графіків точності, похибки та змін параметрів у динаміці.
- Збереження та завантаження моделей: зручне керування версіями мережі безпосередньо з інтерфейсу.

На Рис. 2.4 показано вигляд графічного інтерфейсу програми.

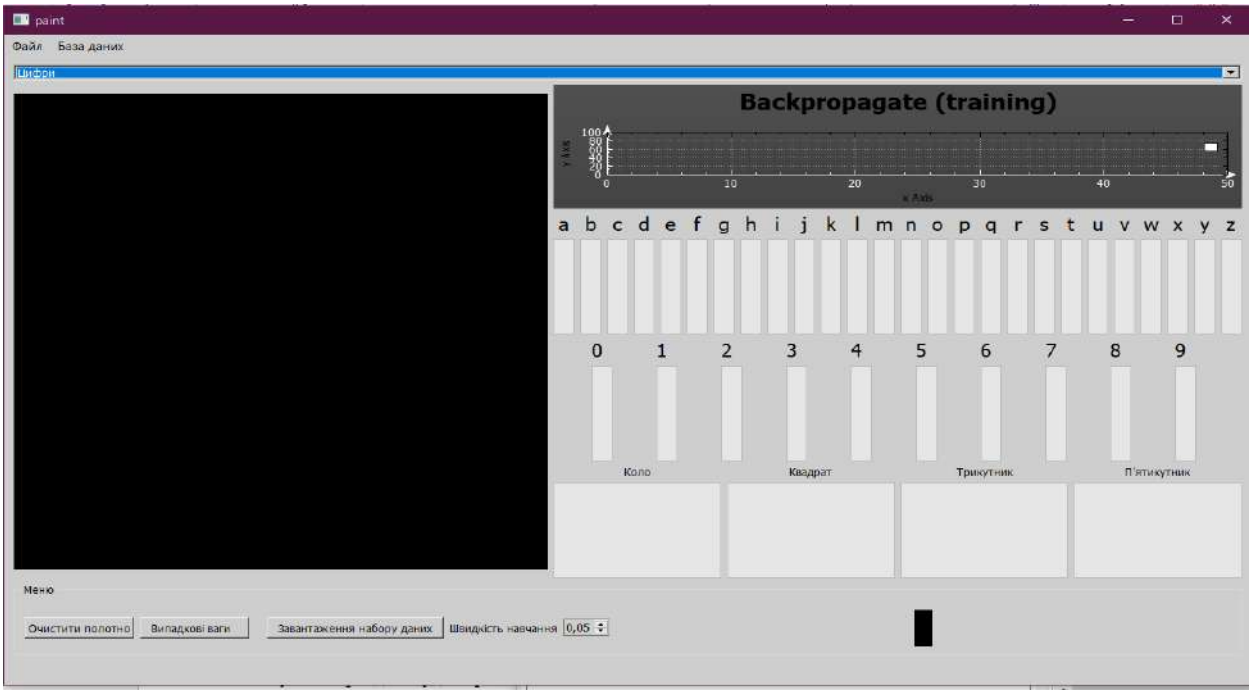


Рис. 2.4. Головне вікно графічного інтерфейсу програми.

Висновки до розділу 2

У розділі 2 розглянуто проектування архітектури нейронної мережі для розпізнавання цифрових символів, латинських букв і геометричних фігур. Обґрунтовано вибір повнозв'язної мережі (DNN), яка забезпечує баланс між простотою реалізації, ефективністю і низькими апаратними вимогами. Архітектура включає вхідний шар, один-два прихованих шари і вихідний шар, що оптимально поєднує продуктивність і складність.

Навчання мережі виконано класичним методом зворотного поширення помилки із стохастичним градієнтним спуском, що забезпечує налаштування ваг і адаптацію до даних. Особливу увагу приділено гіперпараметрам, таким як швидкість навчання та кількість епох, для уникнення перенавчання.

Описано систему збереження та відновлення моделей через бінарну серіалізацію ваг і структури, що підвищує гнучкість експериментів та відтворюваність. Інтеграція SQLite для збереження моделей і історії тренувань спрощує керування експериментами.

Важливою особливістю є інтерактивний графічний інтерфейс на основі Qt, який робить роботу з мережею доступною навіть для користувачів без глибоких технічних знань, забезпечуючи візуалізацію тренування, ручне введення зображень і налаштування параметрів.

Отже, розроблена архітектура та інструменти створюють ефективне та зручне середовище для досліджень і практичного застосування нейронних мереж у задачах розпізнавання образів.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ГРАФІЧНИМ ІНТЕРФЕЙСОМ КОРИСТУВАЧА

3.1. Вибір інструментів та середовища розробки

Для ефективної реалізації програмного забезпечення було ретельно підібрано інструменти та технології, які забезпечують оптимальну продуктивність, гнучкість та зручність подальшого розвитку проекту.

Мова програмування:

C++ (стандарт C++11 та вище) — обрана через високу продуктивність та підтримку сучасних парадигм програмування. Ідеально підходить для інтенсивних обчислень, таких як навчання нейронних мереж.

Графічний інтерфейс:

Qt Framework (версії 5 або 6) — завдяки кросплатформності, великій кількості віджетів і можливості швидкої розробки GUI. Забезпечує легку інтеграцію з графічними та математичними бібліотеками.

Бібліотеки для нейронних мереж:

Власна реалізація на основі шаблонів C++, яка включає класи для опису шарів нейронної мережі (ILayerNeurons), самої мережі (INerualNetwork) та генетичних алгоритмів (IGenetic). Такий підхід дозволяє гнучко моделювати різні архітектури.

База даних:

SQLite — легка, серверонезалежна СУБД, що використовується для збереження вагів мережі, історії навчання та інших даних. Обрана через простоту інтеграції та мінімальні накладні витрати.

Інші інструменти:

- QCustomPlot — для динамічної візуалізації графіків (помилок, точності навчання).
- CMake — для крос-платформної автоматизації збірки проекту.

3.2. Архітектура програмного забезпечення

Архітектура програмного забезпечення базується на модульному підході, який передбачає поділ системи на низку логічно ізольованих, але взаємодіючих між собою компонентів. Такий підхід дає змогу організувати розробку програмного забезпечення у вигляді окремих функціональних блоків, кожен із яких виконує чітко визначену роль. Це значно полегшує реалізацію, тестування, відлагодження, масштабування, а також забезпечує гнучкість при оновленні або розширенні функціоналу без ризику порушити роботу всієї системи.

Основні компоненти архітектури:

1. **Графічний інтерфейс користувача (GUI)**

- Область для малювання зображень (цифр, букв, фігур), реалізована за допомогою класу QWidget із перевизначенням подій миші (mousePressEvent, mouseMoveEvent), що забезпечує інтуїтивно зрозумілий спосіб введення даних користувачем.
- Інформаційні вікна для відображення результатів розпізнавання, зокрема прогрес-бари для демонстрації впевненості нейронної мережі та текстові підказки.
- Вікна візуалізації процесу навчання, які дозволяють у реальному часі спостерігати за динамікою точності та зміною значень помилки за допомогою графіків.

2. **Модуль нейронної мережі**

- Набір спеціалізованих класів, серед яких ILayerNeurons (опис логіки роботи окремих шарів), INerualNetwork (реалізація повнозв’язної штучної нейронної мережі), IGenetic (впровадження генетичних алгоритмів для оптимізації параметрів мережі).
- Реалізовано підтримку кількох режимів розпізнавання: рукописних цифр, літер латиниці та базових геометричних фігур.

3. **Модуль роботи з даними**

- Механізми імпорту навчальних та тестових вибірок із CSV-файлів у стандартному форматі (наприклад, EMNIST).
- Можливість зберігати та завантажувати ваги нейронної мережі у форматі бази даних SQLite, що спрощує повторне використання навченої моделі.

4. Додаткові модулі

- Візуалізація історії навчання та метрик за допомогою бібліотеки QCustomPlot, що дозволяє глибше аналізувати поведінку моделі.
- Діалогові вікна налаштувань і керування, які дозволяють користувачу налаштовувати параметри навчання, вибирати архітектуру мережі та управляти файлами вагів.

3.3. Реалізація графічного інтерфейсу користувача

Графічний інтерфейс користувача реалізовано за допомогою компонентів Qt Designer у поєднанні з власними класами, що забезпечує зручну та інтуїтивну взаємодію з функціоналом нейронної мережі.

Основні елементи GUI:

- **Головне вікно (paint):**
  - Область для малювання (QWidget з перевизначенням методів mousePressEvent, mouseMoveEvent для фіксації рухів миші).
  - Комбінатор (QComboBox) для вибору режиму розпізнавання: цифри, букви, фігури.
  - Кнопки для очищення полотна, завантаження навчальних даних, ініціалізації вагів мережі.
  - Прогрес-бари для візуалізації ймовірності розпізнавання.
  - Графіки для відображення процесу навчання в реальному часі.
- **Діалогові вікна:**
  - Вікно для перегляду вмісту бази даних із застосуванням QTableWidget.

- Вікно для завантаження та збереження вагів мережі.

Нижче на Рис. 3.1 приведено код реалізації функції mouseMoveEvent.

```
void paint::mouseMoveEvent(QMouseEvent *e)
{
    if (!mousePressed) return;
    // Отримуємо позицію курсора миші відносно віджета QLabel
    QPoint labelPos = ui->label->mapFrom(this, e->pos());
    // Продовжуємо тільки якщо курсор знаходиться в межах QLabel
    if (!ui->label->rect().contains(labelPos)) return;
    // Обчислюємо координати курсора у системі 28x28 пікселів
    mx = (labelPos.x() * w) / ui->label->width();
    my = (labelPos.y() * h) / ui->label->height();
    // Обмежуємо координати, щоб вони не виходили за межі [0, w-1] та [0, h-1]
    mx = std::max(0.0f, std::min(mx, static_cast<float>(w - 1)));
    my = std::max(0.0f, std::min(my, static_cast<float>(h - 1)));
    // Обробка кожного пікселя зображення
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < h; j++) {
            auto p = image_size.pixel(i, j); // Отримуємо піксель
            int colors = qBlue(p); // Витягуємо синій канал (відтінок сірого)
            if (mousePressed != 0) {
                // Розраховуємо відстань від поточної точки до позиції курсора
                double dist = ((i - mx) * (i - mx) + (j - my) * (j - my)) * 3 + 0.5;
                // Щоб уникнути ділення на нуль
                if (dist < 1) dist = 1;
                else dist *= dist;
                // Якщо натиснута ліва кнопка миші — малюємо (додаємо яскравість)
                if (mousePressed == 1) colors += 255 / dist;
                // Якщо натиснута права кнопка — стираємо (зменшуємо яскравість)
                else if (mousePressed == 2 && dist < 10) colors -= 255 / dist;
                // Обмежуємо значення кольору в діапазоні [0, 255]
                colors = std::max(0, std::min(colors, 255));
            }
            // Встановлюємо новий колір пікселя (відтінок сірого)
            image_size.setPixel(i, j, QColor(colors, colors, colors).rgb());
            // Зберігаємо нормалізоване значення (0.0 - 1.0) у вхідний масив
            inputs[i + j * w] = colors / 255.0;
        }
    }
    // Оновлюємо результат роботи нейромережі
    updateResultDisplay();
    // Масштабуємо зображення до розміру холста та оновлюємо QLabel
    ui->label->setPixmap(QPixmap::fromImage(image_size.scaled(DRAW_AREA_WIDTH, DRAW_AREA_HEIGHT)));
}
```

Рис. 3.1. Реалізація функції mouseMoveEvent

3.4. Опис класів та функцій головного вікна

У цьому підрозділі представлено детальний опис основних класів і функцій, реалізованих у рамках програмного забезпечення для розпізнавання образів. Архітектура програми побудована з дотриманням принципів об'єктно-орієнтованого програмування, зокрема принципу поділу відповідальностей. Такий підхід забезпечує гнучкість, масштабованість та зручність супроводу програмного коду.

Основні класи

Клас paint є центральним компонентом графічного інтерфейсу користувача. Його основне призначення — забезпечення інтерактивної взаємодії

з користувачем, відображення результатів розпізнавання, а також взаємодія з нейронними мережами.

До складу класу входить низка важливих властивостей:

- Вказівник на інтерфейс ui, який було створено за допомогою конструктора у Qt Designer.
- Об’єкт dbManager, що відповідає за взаємодію з базою даних та керування історією навчання або результатами розпізнавання.
- Масив letterProgressBars, що складається з 26 об'єктів QProgressBar. Вони використовуються для візуалізації ймовірностей, пов'язаних із розпізнаванням літер англійського алфавіту (A–Z).
- Аналогічно, digitProgressBars (10 елементів) відображає ймовірності для цифр (0–9), а shapeProgressBars (4 елементи) — для базових геометричних фігур.
- Зображення image_size у форматі QImage використовується для малювання. Його роздільна здатність становить 28×28 пікселів.
- Посилання на три екземпляри нейронної мережі: NeuralNetDigits, NeuralNetLetters і NeuralNetShapes, які реалізують функціональність розпізнавання відповідно цифр, літер та фігур.
- Масив inputs, який містить 784 значення типу float. Він використовується для зберігання нормалізованих пікселів із зображення перед подачею їх на вхід нейромережі.

Клас також реалізує низку ключових методів:

- Метод mousePressEvent обробляє натискання миші, ініціюючи процес малювання.
- Метод mouseMoveEvent відповідає за безпосереднє малювання на полотні шляхом обробки переміщень курсора та зміни зображення.
- Метод updateResultDisplay забезпечує оновлення результатів розпізнавання та відображення відповідних значень на прогрес-барах.
- Методи push_button_load та push_button_clear реалізують завантаження даних із CSV-файлу та очищення полотна відповідно.

- Метод AddGraph додає графік, який може відображати хід процесу навчання мережі.

Опис ключових функцій GUI

Функція mouseMoveEvent реалізує механізм малювання на полотні. Під час переміщення миші координати курсора перетворюються на позиції у межах 28×28 пікселів. Залежно від дії користувача (малювання або стирання), змінюються значення пікселів зображення. Після цього дані нормалізуються до діапазону [0.0, 1.0] і зберігаються у масиві inputs. Наприкінці викликається метод updateResultDisplay для відображення результатів.

Функція updateResultDisplay використовується для оновлення графічного інтерфейсу відповідно до результатів, отриманих від нейронної мережі. Залежно від вибраного режиму (цифри, літери або фігури), викликається відповідна мережа, що обчислює ймовірності належності зображення до певного класу. Результати відображаються на прогрес-барах, а найімовірніший клас виводиться у текстовому полі інтерфейсу.

Діалогові вікна

Програмне забезпечення містить окремі діалогові вікна для зручного керування та взаємодії з користувачем:

- **Вікно бази даних**, яке дозволяє переглядати історію навчання, зберігати та видаляти дані, що стосуються процесу розпізнавання.
- **Вікно керування вагами**, у якому реалізовано функції збереження та завантаження вагових коефіцієнтів нейронної мережі, що дає змогу продовжувати навчання з попередніх станів або виконувати тестування на нових даних.

Розроблений графічний інтерфейс користувача забезпечує:

- Зручне введення зображень за допомогою миші;
- Візуалізацію результатів розпізнавання за допомогою прогрес-барів;
- Можливість керування процесом навчання, тестування та збереженням результатів;
- Доступ до даних через функціонал бази даних.

3.5. Реалізація функціоналу нейронної мережі

У цьому підрозділі детально розглядаються ключові компоненти нейронної мережі, їх архітектура та принципи роботи. Програмне забезпечення реалізує багатошарову нейронну мережу з можливістю гнучкого налаштування глибини та розмірів шарів. Передбачено використання трьох типів активаційних функцій — Sigmoid, Tanh та ReLU — що дозволяє адаптувати мережу до різних задач. Навчання реалізовано на основі алгоритму зворотного поширення помилки (backpropagation), а також доступна оптимізація вагів за допомогою генетичних алгоритмів. Стан моделі можна зберігати та відновлювати за допомогою механізмів серіалізації.

Клас ILayerNeurons

Цей клас є базовим компонентом, що відповідає за окремий шар у нейронній мережі. Він містить наступні основні поля:

- in_count — кількість вхідних зв’язків шару;
- out_count — кількість нейронів у поточному шарі;
- matrixWeight — масив вагів розміром (in_count+1)×out_count(in_count + 1) \times out_count(in_count+1)×out_count, де +1 відповідає за bias;
- hiddenNeurons — масив поточних значень нейронів;
- errorsBias — масив помилок, які використовуються для корекції вагів під час навчання.

Метод InitRandomWeight() відіграє ключову роль у процесі ініціалізації нейронної мережі, забезпечуючи початкові значення вагів та зміщень (bias), які критично важливі для ефективного навчання. Використання нормального розподілу для ініціалізації допомагає уникнути проблем, пов'язаних із зникненням або вибухом градієнтів, особливо у глибоких мережах. Реалізацію цього методу показано на Рис. 3.2.

```
void ILayerNeurons::InitRandomWeight()
{
    srand(time(NULL));
    std::mt19937 rng;
    // initialize the random number generator with time-dependent seed
    uint32_t timeSeed = std::chrono::high_resolution_clock::now().time_since_epoch().count();
    std::seed_seq ss{uint32_t(timeSeed & 0xffffffff), uint32_t(timeSeed>>16)};
    rng.seed(ss);
    // initialize a uniform distribution between 0 and 1
    std::uniform_real_distribution<double> unif(0, 1);

    for(int outp =0; outp < out_count; outp++)
    {
        //errorsBias[outp] =  (((float)rand() / (float)RAND_MAX) - 0.5)/out_count;// * pow(out_count,-0.5);
        errorsBias[outp] = (unif(rng) - 0.5) * pow(out_count,-0.5);
    }

    for(int inp =0; inp < in_count+1; inp++)
    {
        for(int outp =0; outp < out_count; outp++)
        {
            getMatrix(inp,outp) =  (unif(rng) - 0.5) * pow(out_count,-0.5);
            //getMatrix(inp,outp) = ( ((float)rand() / (float)RAND_MAX) - 0.5 )* pow(out_count,-0.5);
        }
    }
}
```

Рис. 3.2. Реалізації метода InitRandomWeight()

Методи serialize() та deserialize() забезпечують перетворення шару в бінарний формат і назад, що дозволяє зберігати модель на диск або передавати її мережею.

Клас INeuralNetwork

Клас **INeuralNetwork** є центральним елементом, який відповідає за управління штучною нейронною мережею в цілому. Він реалізує як архітектуру мережі, так і ключові алгоритми навчання та обчислення. У межах даної реалізації цей клас використовується для побудови як стандартної багатошарової персептронної моделі, так і для її подальшої оптимізації та експлуатації.:

До основних властивостей класу належать:

- **m_nLayers** — це вектор вказівників на об'єкти типу ILayerNeurons, що представляють окремі шари нейронної мережі. Кожен такий об’єкт відповідає за збереження вагів, значень нейронів та реалізацію окремих етапів обчислення;
- **m_NetworkStructure** — це вектор цілих чисел, який визначає структуру мережі. Наприклад, вектор {784, 200, 10} описує мережу з вхідним шаром у 784 нейрони, прихованим шаром у 200 нейронів і вихідним шаром у 10 нейронів. Така структура характерна, зокрема, для задач розпізнавання

рукописних цифр, де 784 — це кількість пікселів у зображенні 28×28, а 10 — кількість класів (від 0 до 9).

Однією з основних функцій цього класу є метод **feedForwarding()**, який реалізує пряме поширення сигналу через усі шари мережі. Він приймає на вхід масив дійсних чисел, що представляє нормалізовані пікселі зображення або інші вхідні дані, і повертає результат обчислення у вигляді вихідного вектора. Алгоритм обчислення базується на зваженій сумі входів кожного нейрона з додаванням зсуву (bias) та застосуванням функції активації. Результат кожного шару передається на вхід наступному шару.

На Рис. 3.3 зображено фрагмент коду, який реалізує метод **feedForwarding()** у класі INeuralNetwork. Цей фрагмент ілюструє внутрішню структуру обчислень, зокрема використання циклів для переходу між шарами та нейронами, а також виклик функції активації.

```
float* INerualNetwork::feedForwarding(const float* _inputs)
{
    // Проходимо по всіх шарах мережі (від входу до виходу)
    for (int i = 0; i < m_nlCountLayers; i++)
    {
        auto &Layer = m_nLayers[i]; // Поточний шар
        // Обчислюємо значення для кожного нейрона в поточному шарі
        for(unsigned int hid = 0; hid < Layer->getOutCount(); hid++)
        {
            float tmpS = 0.0; // Тимчасова змінна для зберігання суми вагованих входів
            // Сумуємо всі входи поточного нейрона, помножені на відповідні ваги
            for(unsigned int inp = 0; inp < Layer->getInCount(); inp++)
            {
                if(i == 0) // Якщо це перший шар, беремо дані з вхідного масиву
                {
                    tmpS += _inputs[inp] * Layer->getMatrix(inp, hid);
                }
                else // Інакше - беремо дані з попереднього шару
                {
                    tmpS += m_nLayers[i-1]->getHidden()[inp] * Layer->getMatrix(inp, hid);
                }
            }
            // Додаємо зміщення (bias) - це додаткова вага, що не залежить від входу
            tmpS += Layer->getMatrix(Layer->getInCount(), hid);
            // Застосовуємо функцію активації до отриманої суми
            Layer->getHidden()[hid] = activation(tmpS, Layer->getLayer_type());
        }
    }
    // Повертаємо вихід останнього шару (результат роботи мережі)
    return m_nLayers[m_nlCountLayers - 1]->getHidden();
}
```

Рис. 3.3. Реалізації метода feedForwarding()

Навчання нейронної мережі реалізовано за допомогою алгоритму зворотного поширення помилки (*backpropagation*), що дозволяє мінімізувати різницю між фактичним і цільовим виходом мережі. Метод **backPropagate()**

виконує два основні етапи: пряме поширення сигналу (обчислення виходу мережі) та зворотне поширення похибки з оновленням вагів.

На першому кроці викликається метод **feedForwarding()**, який обчислює активації кожного шару. Далі похибка на виході мережі розраховується як різниця між очікуваним і фактичним значенням, помножена на похідну активаційної функції. Після цього помилка поширюється назад по мережі, з відповідним коригуванням вагів згідно з градієнтами та заданою швидкістю навчання.

Реалізації метода представлена на Рис. 3.4, де зображено логіку обчислень і оновлення параметрів.

```
// Метод зворотного поширення помилки для навчання нейромережі
void INerualNetwork::backPropagate(const float *_inputes, const float *_targetes, const float& learningRate){
    // 1. Пряме поширення (викликаємо, щоб отримати актуальні виходи мережі)
    feedForwarding(_inputes);
    // 2. Розрахунок помилок для вихідного шару
    for(unsigned int i = 0; i < m_nLayers[m_nlCountLayers-1]->getOutCount(); i++) {
        auto &Layer = m_nLayers[m_nlCountLayers-1];  // Отримуємо вихідний шар
        // Помилка = (ціль - вихід) * похідна функції активації
        Layer->getErrors()[i] = (_targetes[i] - Layer->getHidden()[i]) * derivative(Layer->getHidden()[i]);
    }
    // 3. Зворотне поширення помилок для прихованих шарів
    for (int i = m_nlCountLayers-2; i >= 0; i--) {
        auto &Layer_prev = m_nLayers[i];    // Поточний шар
        auto &Layer = m_nLayers[i+1];       // Наступний шар
        for(unsigned int hid = 0; hid < Layer->getInCount(); hid++) {
            Layer_prev->getErrors()[hid] = 0.0;  // Ініціалізація помилки
            // Сумуємо внески помилок від наступного шару
            for(unsigned int ou = 0; ou < Layer->getOutCount(); ou++)
                Layer_prev->getErrors()[hid] += Layer->getErrors()[ou] * Layer->getMatrix(hid,ou);
            // Помножуємо на похідну функції активації
            Layer_prev->getErrors()[hid] *= derivative(Layer_prev->getHidden()[hid]);
        }
    }
    // 4. Оновлення вагів (градієнтний спуск)
    for (int i = m_nlCountLayers-1; i >= 0; i--) {
        auto &Layer = m_nLayers[i];
        for(unsigned int ou = 0; ou < Layer->getOutCount(); ou++) {
            // Оновлюємо ваги для зв'язків між нейронами
            for(unsigned int hid = 0; hid < Layer->getInCount(); hid++) {
                if(i == 0) // Для першого шару використовуємо вхідні дані
                    Layer->getMatrix(hid,ou) += (learningRate * Layer->getErrors()[ou] * _inputes[hid]);
                else if(i > 0) // Для інших шарів - виходи попереднього шару
                    Layer->getMatrix(hid,ou) += (learningRate * Layer->getErrors()[ou] * m_nLayers[i-1]->getHidden()[hid]);
            } // Оновлюємо bias (вага для додаткового входу)
            Layer->getMatrix(Layer->getInCount(),ou) += (learningRate * Layer->getErrors()[ou]);
        }
    }
}
```

Рис. 3.4. Реалізації метода backPropagate()

Додатково, для забезпечення можливості збереження поточного стану моделі передбачено метод serialize(), який кодує повну архітектуру нейронної мережі (кількість шарів, їх параметри, типи активації) та всі значення вагів у двійковий формат. Це дозволяє зберігати модель у файл або базу даних, а також

відновлювати її стан за допомогою відповідного методу десеріалізації (deserialize()), що є критично важливим при роботі з великими обсягами даних або для подальшого використання попередньо навчених моделей.

Клас IGenetic

Цей клас реалізує генетичні операції, такі як кросовер і мутації, для оптимізації вагів мережі. Зокрема, метод Croosover() створює нову мережу на основі двох батьківських з урахуванням заданої ймовірності мутації.

Алгоритм роботи методу Croosover() та взаємодія між класами продемонстровані на Рис. 3.5. Код ілюструє принцип поєднання структур даних та передачі параметрів між класами INeuralNetwork та IGenetic.

```
INerualNetwork* IGenetic::Croosover(const INerualNetwork* a, const INerualNetwork* b, float rate) {
    INerualNetwork* child = new INerualNetwork(*a);
    for(int i = 0; i < child->getCountLayers(); i++) {
        for(int j = 0; j < child->getLayer(i)->getInCount(); j++) {
            if(rand() < rate) {  // Мутація
                child->getLayer(i)->getMatrix(j, k) = randomValue();
            } else {  // Кросовер
                child->getLayer(i)->getMatrix(j, k) = (rand() % 2) ?
                                                      a->getLayer(i)->getMatrix(j, k) :
                                                      b->getLayer(i)->getMatrix(j, k);
            }
        }
    }
    return child;
}
```

Рис. 3.5. Зразковий код реалізації Croosover()

3.6. Візуалізація даних

Для наочного аналізу процесу навчання нейронної мережі у програмному забезпеченні була використана бібліотека QCustomPlot - потужний інструмент для візуалізації даних у фреймворку Qt. Ця бібліотека була обрана через свої унікальні характеристики та переваги:

- **Динамічні графіки**:
  - Відображення помилки (MSE) та точності (accuracy) у реальному часі під час навчання.

- Оновлення даних кожні 100 ітерацій (метод push_button_load()). Це дасть можливість бачити вірні відповіді і помилку нейромережі.

- **Кастомізація**:
  - Підтримка кількох графіків на одному полі (наприклад, помилка — червона лінія, точність — зелена).
  - Налаштування кольорів, типів ліній, маркерів (наприклад, ssPlus для точок даних).
- **Інтерактивність**:
  - Масштабування та перетягування графіків за допомогою миші (QCP::iRangeDrag | QCP::iRangeZoom).
  - Підсвітка легенди та точок даних при наведенні.

Висновки до розділу 3

У третьому розділі детально проаналізовано процес розробки програмного забезпечення для розпізнавання образів із використанням штучних нейронних мереж. Було обґрунтовано вибір інструментів і технологій, реалізовано програмну архітектуру та функціональність, а також проведено інтеграцію з графічним інтерфейсом користувача та засобами візуалізації даних.

Вибір мови програмування C++ у поєднанні з фреймворком Qt, системою збереження даних SQLite та бібліотекою QCustomPlot забезпечив належний рівень продуктивності, кросплатформеність і зручність у реалізації функціоналу. Власна реалізація нейронної мережі надала змогу адаптувати її архітектуру під специфіку поставленого завдання та впровадити механізми гнучкого налаштування.

Програмне забезпечення побудоване за модульним принципом, що передбачає чітке розділення на компоненти: графічний інтерфейс користувача, логіку нейронної мережі та обробку даних. Такий підхід значно спростив розробку, забезпечив ефективне тестування та підвищив масштабованість

системи. Кожен компонент має власну сферу відповідальності, що відповідає сучасним вимогам до архітектури програмних систем.

Інтерфейс користувача реалізовано з використанням засобів Qt Designer. Він містить інтерактивне полотно для введення зображень, систему прогрес-барів для відображення результатів розпізнавання та графіки динаміки навчання, що підвищує зручність експлуатації системи для кінцевого користувача.

Нейронна мережа реалізована на основі класів ILayerNeurons, INeuralNetwork та IGenetic, що забезпечують повний життєвий цикл моделі: від ініціалізації вагів і налаштування структури до навчання за допомогою алгоритму зворотного поширення помилки (backpropagation) та оптимізації з використанням генетичних алгоритмів. Передбачено функціональність серіалізації та десеріалізації моделі, що дозволяє зберігати поточний стан мережі для подальшого використання.

Інтеграція бібліотеки QCustomPlot дозволила реалізувати високоякісну візуалізацію процесу навчання. Побудова графіків зміни точності та помилки у реальному часі дає змогу ефективно відстежувати прогрес моделі та приймати обґрунтовані рішення щодо корекції гіперпараметрів.

Загальна архітектура системи є гнучкою та легко масштабованою. Програма дозволяє інтегрувати нові типи вхідних даних, розширювати набір функціональних можливостей, удосконалювати алгоритми навчання або підключати додаткові засоби аналізу.

Результатом проведеної роботи стало створення повнофункціонального програмного засобу для розпізнавання образів, що поєднує сучасні теоретичні підходи та практичну реалізацію. Обрані технічні рішення продемонстрували свою ефективність і можуть бути використані як основа для подальших науково-дослідних або прикладних розробок у галузі штучного інтелекту.

РОЗДІЛ 4
ТЕСТУВАННЯ, ОПТИМІЗАЦІЯ ТА ОЦІНКА РЕЗУЛЬТАТІВ ФУНКЦІОНУВАННЯ НЕЙРОННОЇ МЕРЕЖІ

4.1. Методика тестування нейронної мережі

Тестування нейронної мережі є одним із найважливіших і невід'ємних етапів у процесі її розробки та впровадження. Цей етап має на меті всебічну перевірку ефективності, стабільності та точності роботи моделі на різноманітних типах вхідних даних. Саме тестування дозволяє оцінити, наскільки створена нейронна мережа здатна узагальнювати отримані знання та адекватно реагувати на нові, раніше невідомі зразки, що є ключовим чинником у застосуванні штучного інтелекту в реальних задачах.

У межах даної роботи тестування було проведено на трьох основних категоріях вхідних даних, які суттєво відрізняються за структурою та складністю: рукописних цифрах в діапазоні від 0 до 9, великих латинських літерах від A до Z, а також простих геометричних фігурах, до яких відносяться коло, квадрат, трикутник і пентагон. Для кожної з цих категорій було сформовано окремі набори даних, які представляли собою цифрові матриці зображень розміром 28×28 пікселів. Зображення були попередньо оброблені та нормалізовані для забезпечення однакових умов подання інформації до нейронної мережі.

Підготовка вхідних даних передбачала використання форматів CSV, у яких для цифр та літер кожен рядок містив відповідну мітку класу, а також набір значень пікселів, що були відображені у нормалізованому числовому діапазоні. Такий підхід дозволяв забезпечити стандартизований формат даних, зручний для завантаження та обробки. У випадку геометричних фігур датасети були сформовані за допомогою генерації зображень у графічному середовищі, де кожне зображення також представлялося у вигляді матриці пікселів з подальшою нормалізацією.

Процес навчання нейронної мережі здійснювався за класичною схемою, згідно з якою загальний обсяг даних поділявся на тренувальний набір, що становив 80% усіх зразків, і тестовий набір, що містив залишкові 20%. Такий розподіл забезпечує адекватне навчання мережі з одночасною можливістю об’єктивної оцінки її продуктивності на невідомих даних. В якості основного алгоритму навчання було використано метод зворотного поширення помилки (backpropagation), який дозволяє ітеративно коригувати ваги зв’язків між нейронами з метою мінімізації функції втрат, що відображає розбіжність між очікуваними та фактичними результатами.

Оцінка результатів навчання і тестування проводилася за допомогою низки ключових метрік. Основною метрикою точності (accuracy) визначено частку правильно класифікованих прикладів від загальної кількості тестових зразків, що є інтуїтивно зрозумілим показником загальної ефективності моделі. Додатково використовувалась середньоквадратична помилка (Mean Squared Error, MSE), яка характеризує середнє квадратичне відхилення між прогнозованими виходами нейронної мережі та їхніми фактичними значеннями, що дозволяє оцінити ступінь помилок при класифікації. Не менш важливою була і оцінка швидкості навчання — часового ресурсу, який був необхідний для досягнення встановленого рівня точності, що впливає на практичність застосування моделі у реальних умовах.

4.2. Результати тестування

Для розв’язання задачі класифікації рукописних цифр було використано архітектуру штучної нейронної мережі, яка складається з вхідного шару, що містить 784 нейрони (відповідає кількості пікселів у зображенні розміром 28×28), одного прихованого шару зі 200 нейронами та вихідного шару з 10 нейронів, кожен з яких відповідає одній із цифр від 0 до 9. Під час тестування на відповідному наборі даних дана модель продемонструвала стабільний рівень точності, що перебував у межах 80–95%. При цьому середньоквадратична

помилка (MSE) не перевищувала значення 0.20, що свідчить про високий рівень здатності мережі до узагальнення. Більш детально результати тестування відображено на Рис. 4.1, де червоною лінією показано зміну середньоквадратичної помилки протягом 100 ітерацій, а зеленою лінією — кількість правильно класифікованих зразків (позначено як *Correct*) зі 100 можливих. Як видно з графіка, зі зростанням кількості ітерацій значення помилки поступово зменшується, що свідчить про ефективне навчання моделі. Водночас кількість правильних відповідей зростає, досягаючи свого максимуму в завершальних ітераціях.

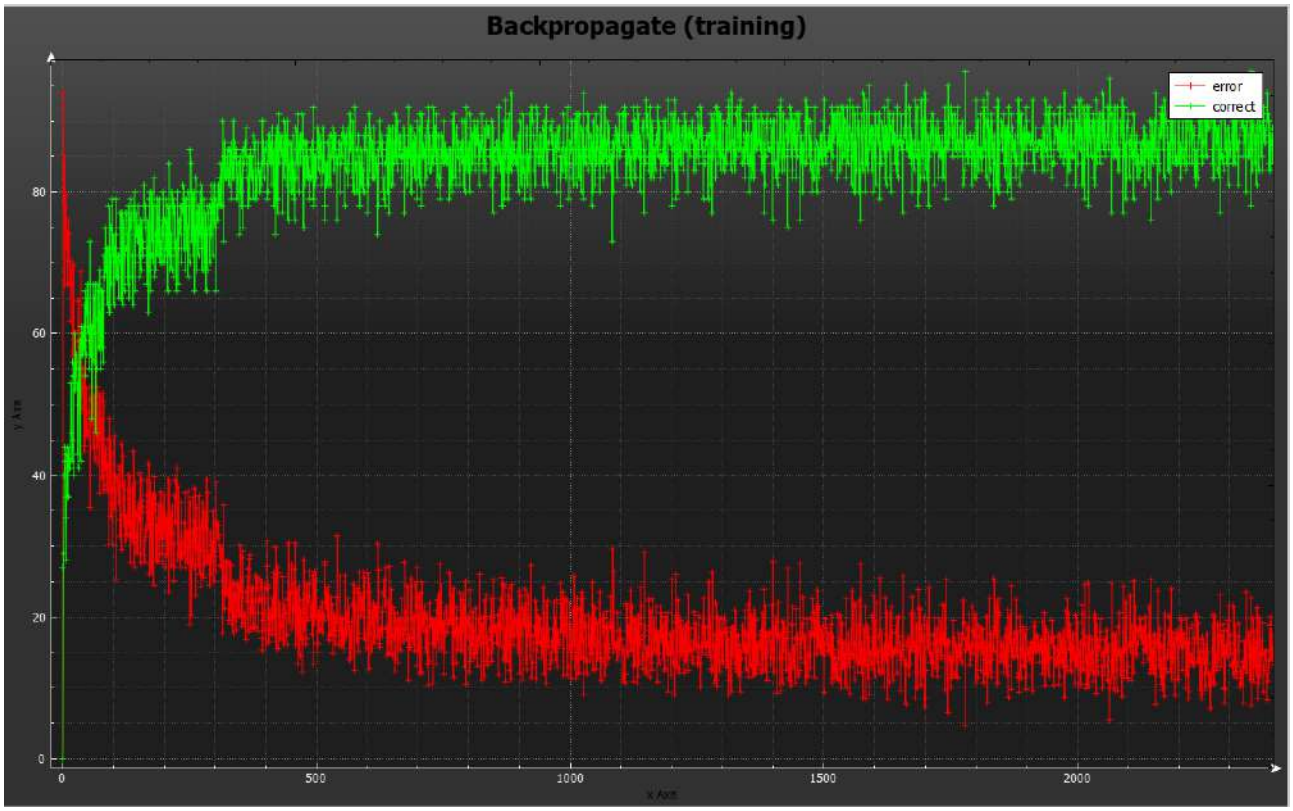
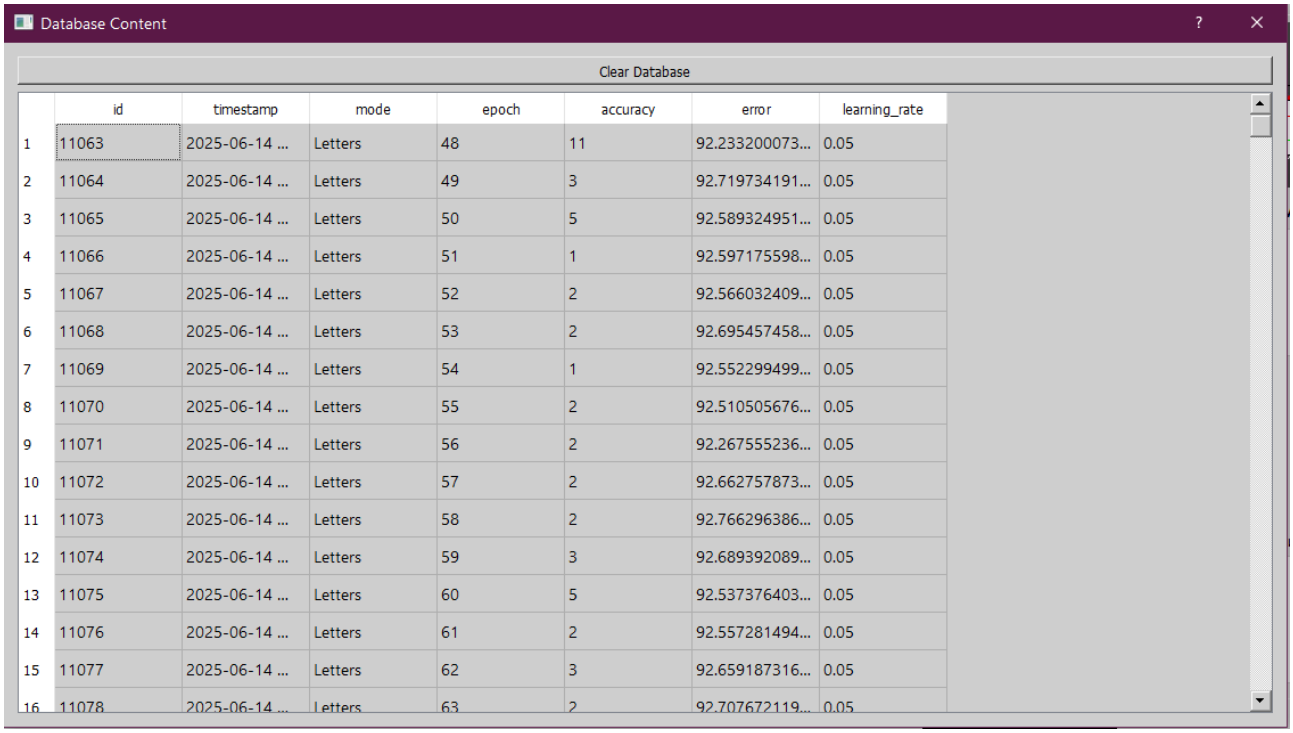


Рис. 4.1. Графік Backpropagate(training) при навчанні на базі MNIST

Для задачі розпізнавання літер було реалізовано більш складну архітектуру нейронної мережі, яка містить два прихованих шари з 300 та 150 нейронів відповідно, а також вихідний шар із 26 нейронів, кожен з яких відповідає одній літері латинського алфавіту. Незважаючи на збільшену кількість класів та підвищену складність завдання, модель досягала рівня точності в межах 50–42%, при цьому середнє значення помилки становило 0,94–0,95. Такі результати

можна вважати задовільними, враховуючи специфіку розпізнавання літерних символів.

Більш детальну інформацію наведено на Рис. 4.2, де представлено дані, зчитані з бази даних. Зокрема, відображено вміст відповідного діалогового вікна «Database Content», що дозволяє простежити хід навчання та оцінити динаміку змін основних показників ефективності.



	id	timestamp	mode	epoch	accuracy	error	learning_rate
1	11063	2025-06-14 ...	Letters	48	11	92.233200073...	0.05
2	11064	2025-06-14 ...	Letters	49	3	92.719734191...	0.05
3	11065	2025-06-14 ...	Letters	50	5	92.589324951...	0.05
4	11066	2025-06-14 ...	Letters	51	1	92.597175598...	0.05
5	11067	2025-06-14 ...	Letters	52	2	92.566032409...	0.05
6	11068	2025-06-14 ...	Letters	53	2	92.695457458...	0.05
7	11069	2025-06-14 ...	Letters	54	1	92.552299499...	0.05
8	11070	2025-06-14 ...	Letters	55	2	92.510505676...	0.05
9	11071	2025-06-14 ...	Letters	56	2	92.267555236...	0.05
10	11072	2025-06-14 ...	Letters	57	2	92.662757873...	0.05
11	11073	2025-06-14 ...	Letters	58	2	92.766296386...	0.05
12	11074	2025-06-14 ...	Letters	59	3	92.689392089...	0.05
13	11075	2025-06-14 ...	Letters	60	5	92.537376403...	0.05
14	11076	2025-06-14 ...	Letters	61	2	92.557281494...	0.05
15	11077	2025-06-14 ...	Letters	62	3	92.659187316...	0.05
16	11078	2025-06-14 ...	Letters	63	2	92.707672119...	0.05

Рис. 4.2. Зображення таблиці з даними про навчання на базі EMNIST

У випадку класифікації геометричних фігур архітектура нейронної мережі складалася з трьох основних шарів: вхідного шару, що містив 784 нейрони, двох прихованих шарів із 200 та 10 нейронів відповідно, а також вихідного шару з 4 нейронів, кожен з яких відповідав окремій фігурі. Досягнута точність класифікації становила 93–95%, при середньому значенні помилки в межах 0,03–0,04. Отримані результати свідчать про ефективне навчання мережі навіть за умов підвищеної варіативності форми вхідних зображень.

На Рис. 4.3 представлено графік процесу навчання, який ілюструє динаміку зменшення помилки та зростання точності під час тренування на власноруч створеному датасеті. Це демонструє достатню здатність моделі до узагальнення навіть у випадку неуніфікованих даних.

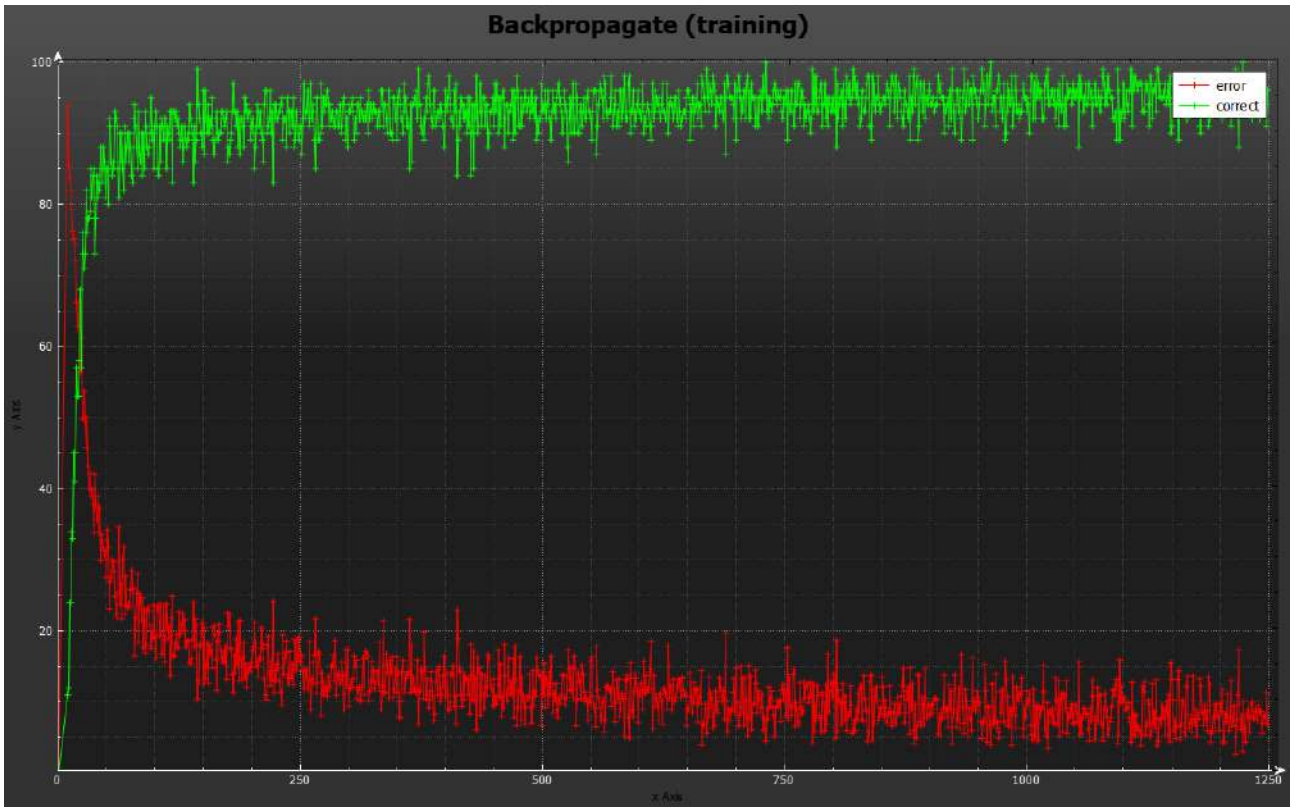


Рис. 4.3. Графік Backpropagate(training) при навчанні на неуніфікованих даних

Для візуалізації динаміки процесу навчання були побудовані графіки залежності точності та середньоквадратичної помилки від кількості епох. Крім того, результати навчання було збережено до бази даних, що дає змогу переглядати детальну інформацію про перебіг навчання нейронної мережі.

Відповідні графічні матеріали наведено на Рис. 4.1. – Рис. 4.3. Вони ілюструють стрімке зростання точності на початкових етапах тренування з подальшим досягненням плато, що є типовим для процесу збіжності навчального алгоритму штучної нейронної мережі.

4.3. Оптимізація роботи нейронної мережі

Оптимізація параметрів навчання та архітектури мережі є критичною для досягнення максимальної ефективності системи. У ході експериментів було встановлено, що оптимальна швидкість навчання (learning rate) знаходиться в межах від 0,01 до 0,05. Значення, що перевищують цей діапазон, спричиняють

нестабільність оновлення ваг і «розбігання» навчання, тоді як менші значення значно уповільнюють збіжність.

Для запобігання перенавчанню (overfitting) було застосовано L2-регуляризацію, яка сприяла покращенню узагальнюючих властивостей мережі, що підтвердилося підвищенням якості на тестовій вибірці. Важливим кроком стало модифікування архітектури, зокрема додавання другого прихованого шару у випадку розпізнавання літер, що позитивно вплинуло на точність моделі.

Крім того, було проведено порівняння функцій активації: заміна традиційної сигмоїдальної функції на ReLU дозволила прискорити процес навчання без втрати точності, що підтверджує актуальність вибору сучасних активаційних функцій у штучних нейронних мережах.

Важливою функціональною можливістю стала реалізація механізму збереження та відновлення ваг мережі. Це дозволяє уникнути повторного тривалого навчання при наступних запусках програми, значно підвищуючи зручність використання та практичність розробленого програмного забезпечення.

4.4. Аналіз результатів та порівняння з аналогами

Порівняння розробленої нейронної мережі з класичними моделями, зокрема LeNet-5, показало, що реалізована система володіє рядом суттєвих переваг. По-перше, вона відзначається універсальністю, оскільки здатна розпізнавати різноманітні типи вхідних даних — не лише цифри, а й літери та графічні об’єкти. По-друге, порівняно з глибокими конволюційними мережами, архітектура залишається порівняно простою, що зменшує обчислювальні вимоги та полегшує впровадження у середовищах з обмеженими ресурсами.

Водночас, аналіз показав, що точність класифікації літер не досягає рівня, характерного для спеціалізованих глибоких мереж, що є потенційним напрямом для подальших удосконалень. Крім того, зростання кількості шарів та нейронів

призводить до збільшення обчислювального навантаження, що потрібно враховувати при масштабуванні системи.

Висновки до розділу 4

У цьому розділі було детально і всебічно розглянуто процес тестування, оптимізації та аналізу результатів функціонування створеної нейронної мережі, яка була розроблена для розпізнавання образів різної природи. Було проведено експериментальне тестування моделі на трьох типах вхідних даних — рукописних цифрах, літерах латинського алфавіту та базових геометричних фігурах. Ці категорії охоплюють як прості, так і більш складні класи зображень, що дозволяє комплексно оцінити узагальнюючу здатність побудованої архітектури.

Для кожного типу даних було адаптовано конфігурацію нейронної мережі, що забезпечило баланс між точністю класифікації, кількістю параметрів та обчислювальною ефективністю. Результати тестування продемонстрували високі значення точності класифікації (accuracy) при низьких значеннях середньоквадратичної похибки (MSE), що свідчить про адекватне узагальнення моделі навіть на раніше невідомих прикладах. Це підтверджує, що використана модель здатна до навчання на різнорідних вибірках і зберігає стабільність у процесі розпізнавання образів.

Особливе значення надано візуалізації процесу навчання, яка була реалізована у вигляді графіків точності та помилки, що оновлюються в реальному часі. Такий підхід дозволив не лише якісно оцінити ефективність навчання, але й виявити критичні моменти, пов’язані з перенавчанням або нестабільною збіжністю. Систематичне використання метрик дало змогу провести точну оцінку продуктивності моделі та забезпечити можливість аналітичного порівняння різних конфігурацій нейромережі.

Оптимізація гіперпараметрів — таких як швидкість навчання, кількість нейронів у шарах, кількість епох, а також типи активаційних функцій — значно

вплинула на швидкість збіжності та стійкість до локальних мінімумів. Було також протестовано ефекти використання регуляризації як засобу уникнення перенавчання, що виявилося особливо ефективним при роботі з великою кількістю класів.

Крім того, реалізований функціонал збереження та відновлення вагів мережі в поєднанні з використанням бази даних SQLite дозволив забезпечити повторюваність результатів і спростити повторне використання вже натренованих моделей без потреби у новому навчанні. Такий підхід підвищує продуктивність і зручність системи в реальних умовах експлуатації.

Порівняльний аналіз із класичними архітектурами, зокрема LeNet-5, показав, що розроблена модель за умов помірної складності забезпечує співставну або вищу точність при меншому обсязі обчислень. Це свідчить про конкурентоспроможність обраного підходу, особливо у застосуваннях з обмеженими апаратними ресурсами. Разом із тим, у ході тестування було виявлено, що класифікація складніших символів, зокрема літер, є більш чутливою до параметрів навчання та потребує удосконалення архітектури.

У підсумку, проведене тестування підтвердило ефективність обраної методики реалізації та навчання штучної нейронної мережі. Розроблена система виявилася придатною для задач розпізнавання образів у різних категоріях даних та може бути подальше масштабована за рахунок ускладнення архітектури, додавання згорткових шарів або використання більш просунутих методів оптимізації. Отримані результати свідчать про практичну доцільність використання даної системи та її потенціал для подальших досліджень і вдосконалення.

ВИСНОВОК

У межах кваліфікаційної бакалаврської роботи було створено програмне забезпечення для розпізнавання цифр, букв латинського алфавіту та геометричних фігур із використанням повнозв’язної нейронної мережі. Основною метою було розробити точну та ефективну модель з інтуїтивним графічним інтерфейсом користувача.

Проведено аналіз існуючих методів розпізнавання образів, як традиційних, так і нейромережевих. Обґрунтовано вибір архітектури повнозв’язної нейронної мережі, яка показала задовільні результати при розпізнаванні зображень невеликого розміру. Навчання мережі реалізовано з використанням методу зворотного поширення помилки та градієнтного спуску, що забезпечило адаптацію до різних типів вхідних даних.

Особливу увагу приділено системі збереження моделей. Реалізовано серіалізацію вагів у бінарному форматі та інтеграцію з базою даних SQLite для збереження станів мережі, історії навчання та експериментальних результатів. Це забезпечує повторюваність та ефективне управління навчальним процесом.

Графічний інтерфейс на основі фреймворку Qt дозволяє зручно взаємодіяти з мережею: вводити зображення, переглядати результати розпізнавання, запускати навчання, зберігати та завантажувати моделі. Інтерфейс забезпечує доступність програми для широкого кола користувачів, включно з тими, хто не має глибоких технічних знань.

Проведене тестування підтвердило коректність роботи системи та її здатність до класифікації образів із високим рівнем точності. Розроблене програмне забезпечення має практичну цінність та може бути використане для подальших досліджень і вдосконалення в галузі розпізнавання образів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гудзь С.П. Штучні нейронні мережі. Основи побудови та застосування: навч. посіб. — К.: Ліра-К, 2018. — 256 с.

2. Сидоренко В.М. Програмування мовою C++: підручник. — К.: Либідь, 2021. — 432 с.

3. Бондаренко Л.Д., Розум О.М. Теорія штучних нейронних мереж. — К.: КНУ, 2016. — 198 с.

4. Колесник І.І. Основи машинного навчання: навч. посіб. — Харків: ХНУРЕ, 2020. — 284 с.

5. ISO/IEC 14882:2017 — Programming languages — C++.

6. Bishop C.M. Neural Networks for Pattern Recognition. — Oxford University Press, 1995. — 482 p.

7. Goodfellow I., Bengio Y., Courville A. Deep Learning. — MIT Press, 2016. — 775 p.

8. LeCun Y., Bengio Y., Hinton G. Deep learning // Nature. — 2015. — Vol. 521. — P. 436–444.

9. Chollet F. Deep Learning with Python. — Manning Publications, 2017. — 361 p.

10. Nielsen M. Neural Networks and Deep Learning. — Determination Press, 2015.

11. He K., Zhang X., Ren S., Sun J. Deep Residual Learning for Image Recognition // CVPR, 2016. — P. 770–778.

12. Schmidhuber J. Deep learning in neural networks: An overview // Neural Networks. — 2015. — Vol. 61. — P. 85–117.

13. Rumelhart D.E., Hinton G.E., Williams R.J. Learning representations by back-propagating errors // Nature. — 1986.

14. MNIST Handwritten Digit Database [Електронний ресурс] — https://yann.lecun.com/exdb/mnist/

15. EMNIST Dataset [Електронний ресурс] — https://www.nist.gov/itl/products-and-services/emnist-dataset

16. SQLite Documentation [Електронний ресурс] — https://www.sqlite.org/docs.html

17. Qt Documentation [Електронний ресурс] — https://doc.qt.io/

18. QCustomPlot Library [Електронний ресурс] — https://www.qcustomplot.com/

19. CMake Documentation [Електронний ресурс] — https://cmake.org/documentation/

20. OpenAI. GPT models and neural architectures [Електронний ресурс] — https://openai.com/research

21. Karpathy A. Neural Networks: Zero to Hero [Відеокурс] — https://www.youtube.com/@karpathy

22. Geron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. — O'Reilly, 2019. — 819 p.

23. Zhang C., Bengio S., Hardt M., Recht B., Vinyals O. Understanding deep learning requires rethinking generalization // ICLR, 2017.

ДОДАТКИ

Додаток А

Опис основних класів бази даних

```
#include "DatabaseManager.h"
#include <QDebug>

DatabaseManager::DatabaseManager(const QString& dbPath) : dbPath(dbPath), db(nullptr) {}

bool DatabaseManager::open() {
    if (sqlite3_open(dbPath.toStdString().c_str(), &db) != SQLITE_OK) {
        qDebug() << "Failed to open database:" << sqlite3_errmsg(db);
        return false;
    }

    char* errMsg = nullptr;  // Перенесіть оголошення сюди

    const char* createTableQuery =
        "CREATE TABLE IF NOT EXISTS TrainingData ("
        "id INTEGER PRIMARY KEY AUTOINCREMENT,"
        "mode TEXT NOT NULL,"
        "label INTEGER NOT NULL,"
        "inputs BLOB NOT NULL);";

    const char* createHistoryTableQuery =
        "CREATE TABLE IF NOT EXISTS TrainingHistory ("
        "id INTEGER PRIMARY KEY AUTOINCREMENT,"
        "timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,"
        "mode TEXT NOT NULL,"
        "epoch INTEGER NOT NULL,"
```

```
        "accuracy REAL NOT NULL,"
        "error REAL NOT NULL,"
        "learning_rate REAL NOT NULL);";

    if (sqlite3_exec(db, createHistoryTableQuery, nullptr, nullptr, &errMsg) != SQLITE_OK) {
        qDebug() << "Failed to create history table:" << errMsg;
        sqlite3_free(errMsg);
        return false;
    }

    if (sqlite3_exec(db, createTableQuery, nullptr, nullptr, &errMsg) != SQLITE_OK) {
        qDebug() << "Failed to create table:" << errMsg;
        sqlite3_free(errMsg);
        return false;
    }

    return true;
}

void DatabaseManager::close() {
    if (db) {
        sqlite3_close(db);
        db = nullptr;
    }
}

bool DatabaseManager::saveTrainingData(const QString& mode, const std::vector<float>& inputs, int label) {
```

Продовження додаток А

```
const char* insertQuery = "INSERT INTO TrainingData (mode, label, inputs) VALUES (?, ?, ?);";
    sqlite3_stmt* stmt;

    if (sqlite3_prepare_v2(db, insertQuery, -1, &stmt, nullptr) != SQLITE_OK) {
        qDebug() << "Failed to prepare statement:" << sqlite3_errmsg(db);
        return false;
    }

    sqlite3_bind_text(stmt, 1, mode.toStdString().c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_int(stmt, 2, label);
    sqlite3_bind_blob(stmt, 3, inputs.data(), inputs.size() * sizeof(float), SQLITE_STATIC);

    if (sqlite3_step(stmt) != SQLITE_DONE) {
        qDebug() << "Failed to execute statement:" << sqlite3_errmsg(db);
        sqlite3_finalize(stmt);
        return false;
    }

    sqlite3_finalize(stmt);
    return true;
}

std::vector<std::pair<std::vector<float>, int>> DatabaseManager::loadTrainingData(const QString& mode) {
    std::vector<std::pair<std::vector<float>, int>> data;
    const char* selectQuery = "SELECT label, inputs FROM TrainingData WHERE mode = ?;";
```

```
 sqlite3_stmt* stmt;
    if (sqlite3_prepare_v2(db, selectQuery, -1, &stmt, nullptr) != SQLITE_OK) {
        qDebug() << "Failed to prepare statement:" << sqlite3_errmsg(db);
        return data;
    }

    sqlite3_bind_text(stmt, 1, mode.toStdString().c_str(), -1, SQLITE_STATIC);

    while (sqlite3_step(stmt) == SQLITE_ROW) {
        int label = sqlite3_column_int(stmt, 0);
        const float* inputs = static_cast<const float*>(sqlite3_column_blob(stmt, 1));
        int size = sqlite3_column_bytes(stmt, 1) / sizeof(float);

        std::vector<float> inputVec(inputs, inputs + size);
        data.emplace_back(inputVec, label);
    }

    sqlite3_finalize(stmt);
    return data;
}
bool DatabaseManager::saveModel(const QString& name, const std::string&
modelData) {
    const char* query = "INSERT INTO Models (name, data) VALUES (?, ?);";
    // Аналогічно до saveTrainingData
}
std::string DatabaseManager::loadModel(const QString& name) {
    const char* query = "SELECT data FROM Models WHERE name = ?;";
    // Аналогічно до loadTrainingData
}
```

```
bool DatabaseManager::saveTrainingRecord(const TrainingRecord& record) {
    const char* insertQuery =
        "INSERT INTO TrainingHistory (mode, epoch, accuracy, error, learning_rate) "
        "VALUES (?, ?, ?, ?, ?);";

    sqlite3_stmt* stmt;
    if (sqlite3_prepare_v2(db, insertQuery, -1, &stmt, nullptr) != SQLITE_OK) {
        qDebug() << "Failed to prepare statement:" << sqlite3_errmsg(db);
        return false;
    }

    sqlite3_bind_text(stmt, 1, record.mode.toStdString().c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_int(stmt, 2, record.epoch);
    sqlite3_bind_double(stmt, 3, record.accuracy);
    sqlite3_bind_double(stmt, 4, record.error);
    sqlite3_bind_double(stmt, 5, record.learningRate);

    if (sqlite3_step(stmt) != SQLITE_DONE) {
        qDebug() << "Failed to execute statement:" << sqlite3_errmsg(db);
        sqlite3_finalize(stmt);
        return false;
    }

    sqlite3_finalize(stmt);
    return true;
}
```

Продовження додаток А

```
QVector<TrainingRecord> DatabaseManager::getTrainingHistory(const QString& mode) {
    QVector<TrainingRecord> history;
    const char* selectQuery =
        mode.isEmpty()
            ? "SELECT timestamp, mode, epoch, accuracy, error, learning_rate FROM TrainingHistory ORDER BY timestamp DESC;"
            : "SELECT timestamp, mode, epoch, accuracy, error, learning_rate FROM TrainingHistory WHERE mode = ? ORDER BY timestamp DESC;";

    sqlite3_stmt* stmt;
    if (sqlite3_prepare_v2(db, selectQuery, -1, &stmt, nullptr) != SQLITE_OK) {
        qDebug() << "Failed to prepare statement:" << sqlite3_errmsg(db);
        return history;
    }

    if (!mode.isEmpty()) {
        sqlite3_bind_text(stmt, 1, mode.toStdString().c_str(), -1, SQLITE_STATIC);
    }

    while (sqlite3_step(stmt) == SQLITE_ROW) {
        TrainingRecord record;
        record.timestamp = QString(reinterpret_cast<const char*>(sqlite3_column_text(stmt, 0)));
        record.mode = QString(reinterpret_cast<const char*>(sqlite3_column_text(stmt, 1)));
        record.epoch = sqlite3_column_int(stmt, 2);
        record.accuracy = sqlite3_column_double(stmt, 3);
```

```
 record.error = sqlite3_column_double(stmt, 4);
        record.learningRate = sqlite3_column_double(stmt, 5);
        history.append(record);
    }

    sqlite3_finalize(stmt);
    return history;
}

bool DatabaseManager::saveModelWeights(const ModelWeights& weights) {
    const char* createTableQuery =
        "CREATE TABLE IF NOT EXISTS SavedModels ("
        "name TEXT PRIMARY KEY,"
        "timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,"
        "mode TEXT NOT NULL,"
        "epochs_trained INTEGER NOT NULL,"
        "accuracy REAL NOT NULL,"
        "error REAL NOT NULL,"
        "weights_data BLOB NOT NULL);";

    char* errMsg = nullptr;
    if (sqlite3_exec(db, createTableQuery, nullptr, nullptr, &errMsg) != SQLITE_OK) {
        qDebug() << "Failed to create SavedModels table:" << errMsg;
        sqlite3_free(errMsg);
        return false;
    }

    const char* insertQuery =
```

Продовження додаток А

```
"INSERT OR REPLACE INTO SavedModels (name, mode, epochs_trained, accuracy, error, weights_data) "
        "VALUES (?, ?, ?, ?, ?, ?);";

    sqlite3_stmt* stmt;
    if (sqlite3_prepare_v2(db, insertQuery, -1, &stmt, nullptr) != SQLITE_OK) {
        qDebug() << "Failed to prepare statement:" << sqlite3_errmsg(db);
        return false;
    }

    sqlite3_bind_text(stmt, 1, weights.name.toStdString().c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_text(stmt, 2, weights.mode.toStdString().c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_int(stmt, 3, weights.epochsTrained);
    sqlite3_bind_double(stmt, 4, weights.accuracy);
    sqlite3_bind_double(stmt, 5, weights.error);
    sqlite3_bind_blob(stmt, 6, weights.weightsData.data(), weights.weightsData.size(), SQLITE_STATIC);

    if (sqlite3_step(stmt) != SQLITE_DONE) {
        qDebug() << "Failed to save model weights:" << sqlite3_errmsg(db);
        sqlite3_finalize(stmt);
        return false;
    }
    sqlite3_finalize(stmt);
    return true;
}
```

Продовження додаток А

```
QVector<ModelWeights> DatabaseManager::getSavedModels(const QString& mode)
{
    QVector<ModelWeights> models;
    const char* selectQuery =
        mode.isEmpty()
            ? "SELECT name, timestamp, mode, epochs_trained, accuracy, error,
weights_data FROM SavedModels ORDER BY timestamp DESC;"
            : "SELECT name, timestamp, mode, epochs_trained, accuracy, error,
weights_data FROM SavedModels WHERE mode = ? ORDER BY timestamp
DESC;";

    sqlite3_stmt* stmt;
    if (sqlite3_prepare_v2(db, selectQuery, -1, &stmt, nullptr) != SQLITE_OK) {
        qDebug() << "Failed to prepare statement:" << sqlite3_errmsg(db);
        return models;
    }

    if (!mode.isEmpty()) {
        sqlite3_bind_text(stmt, 1, mode.toStdString().c_str(), -1, SQLITE_STATIC);
    }

    while (sqlite3_step(stmt) == SQLITE_ROW) {
        ModelWeights weights;
        weights.name                =                QString(reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 0)));
        weights.timestamp           =                QString(reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 1)));
```

Продовження додаток А

```
 weights.mode = QString(reinterpret_cast<const char*>(sqlite3_column_text(stmt, 2)));
        weights.epochsTrained = sqlite3_column_int(stmt, 3);
        weights.accuracy = sqlite3_column_double(stmt, 4);
        weights.error = sqlite3_column_double(stmt, 5);

        const void* blobData = sqlite3_column_blob(stmt, 6);
        int blobSize = sqlite3_column_bytes(stmt, 6);
        weights.weightsData = std::string(static_cast<const char*>(blobData), blobSize);

        models.append(weights);
    }

    sqlite3_finalize(stmt);
    return models;
}

bool DatabaseManager::deleteModel(const QString& name) {
    const char* deleteQuery = "DELETE FROM SavedModels WHERE name = ?;";
    sqlite3_stmt* stmt;

    if (sqlite3_prepare_v2(db, deleteQuery, -1, &stmt, nullptr) != SQLITE_OK) {
        qDebug() << "Failed to prepare statement:" << sqlite3_errmsg(db);
        return false;
    }

    sqlite3_bind_text(stmt, 1, name.toStdString().c_str(), -1, SQLITE_STATIC);

    if (sqlite3_step(stmt) != SQLITE_DONE) {
```

```
    qDebug() << "Failed to delete model:" << sqlite3_errmsg(db);
        sqlite3_finalize(stmt);
        return false;
    }

    sqlite3_finalize(stmt);
    return true;
}

DatabaseManager::~DatabaseManager() {
    close();
}
```

Додаток Б

Опис функцій нейромережі складової програми

```
INerualNetwork::INerualNetwork(const INerualNetwork &n_copy) :
    m_nInputNeurons(n_copy.m_nInputNeurons),
    m_nOutputNeurons(n_copy.m_nOutputNeurons),
    m_nlCountLayers(n_copy.m_nlCountLayers),
    m_NetworkStructure(n_copy.m_NetworkStructure)

{
    m_nLayers.resize(m_nlCountLayers);
    for (unsigned int i = 0; i < n_copy.m_nLayers.size(); ++i)
    {
        m_nLayers[i] = new ILayerNeurons(n_copy.m_nLayers[i]->getInCount(),n_copy.m_nLayers[i]->getOutCount());
        m_nLayers[i]->getErrors()[i] = n_copy.m_nLayers[i]->getErrors()[i];
        m_nLayers[i]->getHidden()[i] = n_copy.m_nLayers[i]->getHidden()[i];
        for(unsigned int inp =0; inp < n_copy.m_nLayers[i]->getInCount()+1; inp++)
        {
            for(unsigned int outp =0; outp < n_copy.m_nLayers[i]->getOutCount(); outp++)
            {
                m_nLayers[i]->getMatrix(inp,outp) = n_copy.m_nLayers[i]->getMatrix(inp,outp);
            }
        }
    }
}
```

Продовження додаток Б

```
INerualNetwork::INerualNetwork(std::initializer_list<unsigned int> &&values
)
{
   //--- "Neyeral Network" this equal "NN"
   //---set layer count for NN,
   //---where input neuerons for first layer equal NN input
   //---and output neuerons for last layer equal NN output
   m_nlCountLayers = values.size()-1;
   m_nLayers = vector<ILayerNeurons*>(m_nlCountLayers);
   m_NetworkStructure.resize(values.size());


   int i = 0;
   for (auto it = values.begin()+1; i < m_nlCountLayers; ++i, ++it)
   {
      int OutSize = *it;
      int InSize  = *(it-1);
      if(i==0) m_nInputNeurons = InSize;
      else if(i==m_nlCountLayers-1)  m_nOutputNeurons = OutSize;
      m_nLayers[i] = new ILayerNeurons(InSize,OutSize);
      m_NetworkStructure[i] = InSize;
      m_nLayers[i]->InitRandomWeight();
   }

   m_NetworkStructure[m_nlCountLayers] = m_nOutputNeurons;
}

INerualNetwork::INerualNetwork(std::vector<unsigned int> NetworkStructure)
   : m_NetworkStructure(NetworkStructure)
```

Продовження додаток Б

```
{
   //--- "Neyeral Network" this equal "NN"
   //---set layer count for NN,
   //---where input neuerons for first layer equal NN input
   //---and output neuerons for last layer equal NN output
   m_nlCountLayers = NetworkStructure.size()-1;
   m_nLayers = vector<ILayerNeurons*>(m_nlCountLayers);

   int i = 0;
   for (auto it = NetworkStructure.begin()+1; i < m_nlCountLayers; ++i, ++it)
   {
      int OutSize = *it;
      int InSize  = *(it-1);
      if(i==0) m_nInputNeurons = InSize;
      else if(i==m_nlCountLayers-1)  m_nOutputNeurons = OutSize;
      m_nLayers[i] = new ILayerNeurons(InSize,OutSize);
      m_nLayers[i]->InitRandomWeight();
   }
}


INerualNetwork::~INerualNetwork()
{
   for (int i =0; i<m_nlCountLayers; i++)
   {
      delete m_nLayers[i];
   }
}
```

Продовження додаток Б

```
void INerualNetwork::InitRandom()
{
   for (unsigned int i = 0; i < m_nLayers.size(); ++i)
   {
      m_nLayers[i]->InitRandomWeight();
   }
}
```

```
// Функція feedForwarding виконує пряме поширення (feed-forward) даних через нейронну мережу
// inputs - вхідний масив даних, який передається в мережу
// Повертає вказівник на вихідний масив останнього шару
float* INerualNetwork::feedForwarding(const float* _inputs)
{
   // Проходимо по всіх шарах мережі (від входу до виходу)
   for (int i = 0; i < m_nlCountLayers; i++)
   {
      auto &Layer = m_nLayers[i]; // Поточний шар
      // Обчислюємо значення для кожного нейрона в поточному шарі
      for(unsigned int hid = 0; hid < Layer->getOutCount(); hid++)
      {
         float tmpS = 0.0; // Тимчасова змінна для зберігання суми вагованих входів
         // Сумуємо всі входи поточного нейрона, помножені на відповідні ваги
         for(unsigned int inp = 0; inp < Layer->getInCount(); inp++)
         {
            if(i == 0) // Якщо це перший шар, беремо дані з вхідного масиву
```

Продовження додаток Б

```
        {
                    tmpS += _inputs[inp] * Layer->getMatrix(inp, hid);
                }
                else // Інакше - беремо дані з попереднього шару
                {
                    tmpS += m_nLayers[i-1]->getHidden()[inp] * Layer->getMatrix(inp, hid);
                }
            }
            // Додаємо зміщення (bias) - це додаткова вага, що не залежить від входу
            tmpS += Layer->getMatrix(Layer->getInCount(), hid);
            // Застосовуємо функцію активації до отриманої суми
            Layer->getHidden()[hid] = activation(tmpS, Layer->getLayer_type());
        }
    }
    // Повертаємо вихід останнього шару (результат роботи мережі)
    return m_nLayers[m_nlCountLayers - 1]->getHidden();
}


// Метод зворотного поширення помилки для навчання нейромережі
void INerualNetwork::backPropagate(const float *_inputes, const float *_targetes, const float& learningRate){
    // 1. Пряме поширення (викликаємо, щоб отримати актуальні виходи мережі)
    feedForwarding(_inputes);
    // 2. Розрахунок помилок для вихідного шару
```

Продовження додаток Б

```
        for(unsigned int i = 0; i < m_nLayers[m_nlCountLayers-1]->getOutCount(); i++) {
            auto &Layer = m_nLayers[m_nlCountLayers-1];  // Отримуємо вихідний шар
            // Помилка = (ціль - вихід) * похідна функції активації
            Layer->getErrors()[i] = (_targetes[i] - Layer->getHidden()[i]) * derivative(Layer->getHidden()[i]);
        }
        // 3. Зворотне поширення помилок для прихованих шарів
        for (int i = m_nlCountLayers-2; i >= 0; i--) {
            auto &Layer_prev = m_nLayers[i];    // Поточний шар
            auto &Layer = m_nLayers[i+1];       // Наступний шар
            for(unsigned int hid = 0; hid < Layer->getInCount(); hid++) {
                Layer_prev->getErrors()[hid] = 0.0;  // Ініціалізація помилки
                // Сумуємо внески помилок від наступного шару
                for(unsigned int ou = 0; ou < Layer->getOutCount(); ou++)
                    Layer_prev->getErrors()[hid] += Layer->getErrors()[ou] * Layer->getMatrix(hid,ou);
                // Помножуємо на похідну функції активації
                Layer_prev->getErrors()[hid] *= derivative(Layer_prev->getHidden()[hid]);
            }
        }
        // 4. Оновлення вагів (градієнтний спуск)
        for (int i = m_nlCountLayers-1; i >= 0; i--) {
            auto &Layer = m_nLayers[i];
            for(unsigned int ou = 0; ou < Layer->getOutCount(); ou++) {
                // Оновлюємо ваги для зв'язків між нейронами
                for(unsigned int hid = 0; hid < Layer->getInCount(); hid++) {
```

Продовження додаток Б

```
        if(i == 0) // Для першого шару використовуємо вхідні дані
                    Layer->getMatrix(hid,ou) += (learningRate * Layer->getErrors()[ou] * _inputes[hid]);
                else if(i > 0) // Для інших шарів - виходи попереднього шару
                    Layer->getMatrix(hid,ou) += (learningRate * Layer->getErrors()[ou] * m_nLayers[i-1]->getHidden()[hid]);
                } // Оновлюємо bias (вага для додаткового входу)
                Layer->getMatrix(Layer->getInCount(),ou) += (learningRate * Layer->getErrors()[ou]);
            }
        }
    }


    void INerualNetwork::CopyWeightAndErrorBias(const INerualNetwork &other)
    {
        assert(m_nInputNeurons == other.m_nInputNeurons);
        assert(m_nOutputNeurons == other.m_nOutputNeurons);
        assert(m_nlCountLayers == other.m_nlCountLayers);

        for (unsigned int i = 0; i < other.m_nLayers.size(); ++i)
        {
            assert( m_nLayers[i]->getInCount() == other.m_nLayers[i]->getInCount());
            assert( m_nLayers[i]->getOutCount() == other.m_nLayers[i]->getOutCount());
            m_nLayers[i]->getErrors()[i] =  other.m_nLayers[i]->getErrors()[i];
```

Продовження додаток Б

```
        /** m_nLayers[i]->getHidden()[i] =  other.m_nLayers[i]->getHidden()[i]; /**/
            for(unsigned int inp =0; inp < other.m_nLayers[i]->getInCount()+1; inp++)
            {
               for(unsigned int outp =0; outp < other.m_nLayers[i]->getOutCount(); outp++)
               {
                  m_nLayers[i]->getMatrix(inp,outp) = other.m_nLayers[i]->getMatrix(inp,outp);
               }
            }
         }
      }

      // Функція серіалізації нейронної мережі у рядок
      // Повертає рядок, який містить структуру мережі та всі її ваги
      string INerualNetwork::serialize() const
      {
         string retBuf;  // Буфер для зберігання результату серіалізації
         // 1. Серіалізація топології мережі (кількість шарів та їх розміри)
         // ----------------------------------------------------------------
         size_t nbrOfTopoElements = m_NetworkStructure.size() + 1;  // +1 для зберігання кількості шарів
         unsigned int* topoBuf = new unsigned int[nbrOfTopoElements];  // Тимчасовий буфер для топології
         // Перший елемент - кількість шарів у мережі
         topoBuf[0] = m_NetworkStructure.size();
         // Наступні елементи - розміри кожного шару
         for(size_t i = 0; i < m_NetworkStructure.size(); i++)
         {
```

Продовження додаток Б

```
        topoBuf[i+1] = m_NetworkStructure.at(i);  // Записуємо розмір і-того шару
        }
        // Додаємо топологію до результуючого буфера
        retBuf.append(string((char*)topoBuf,   nbrOfTopoElements*sizeof(unsigned
int)));
        delete[] topoBuf;  // Тимчасовий буфер більше не потрібен
        // 2. Серіалізація параметрів кожного шару (ваги та зміщення)
        // ----------------------------------------------------------
        for(auto l : m_nLayers)  // Проходимо по всіх шарах мережі
        {
            string lBuf = l->serialize();  // Серіалізуємо поточний шар
            unsigned int lBufSize = lBuf.size();  // Отримуємо розмір серіалізованих
даних шару

            // Додаємо розмір даних шару (для подальшого десеріалізації)
            retBuf.append(string((char*)(&lBufSize), 1*sizeof(unsigned int)));

            // Додаємо самі серіалізовані дані шару
            retBuf.append(lBuf);
        }
        return retBuf;  // Повертаємо повністю серіалізовану мережу
    }


    INerualNetwork *INerualNetwork::deserialize(const string &buffer)
    {
        const char* buff = buffer.c_str();
        unsigned int nbrOfLayers = ((unsigned int*)buff)[0];
        std::vector<unsigned int> networkStructure;
```

Продовження додаток Б

```
 for( unsigned int i = 0; i < nbrOfLayers; i++ )
   {
      networkStructure.push_back( ((unsigned int*)buff)[i+1] );
   }

   INerualNetwork* network = new INerualNetwork( networkStructure );
   size_t offset = (nbrOfLayers+1) * sizeof(unsigned int);
   for( unsigned int i = 0; i < nbrOfLayers-1; i++ )
   {
      const char* layerBuf = buff + offset;
      unsigned int sizeOfThisLayer = ((unsigned int*)layerBuf)[0];
      string layerData( layerBuf + sizeof(unsigned int), sizeOfThisLayer );
      ILayerNeurons* l = ILayerNeurons::deserialize( layerData );
      network->m_nLayers[i] = l;
      offset = offset + sizeOfThisLayer + sizeof(unsigned int);
   }
   return network;
}

bool INerualNetwork::save(const string &filePath)
{
   ofstream netFile;
   netFile.open( filePath ,ios::binary);

   if( !netFile.is_open() )
      return false;

   netFile << serialize();
   netFile.close();
```

```
  return true;
}

INerualNetwork *INerualNetwork::load(const string &filePath)
{
    ifstream netFile( filePath , ios::binary);
    if( ! netFile.is_open() )
    {
        //std::cout << "Error File \n";
        return NULL;
    }

    // read the whole file
    std::string netAsBuffer((std::istreambuf_iterator<char>(netFile)),
                            std::istreambuf_iterator<char>());
    netFile.close();

    return INerualNetwork::deserialize( netAsBuffer );
}

INerualNetwork* INerualNetwork::loadFromData(const std::string& data) {
    return deserialize(data);
}
```

Додаток В

Опис функцій заголовків використаних у программі для візуалізування даних

```
QT_BEGIN_NAMESPACE
namespace Ui {
class paint;
}
QT_END_NAMESPACE

class paint : public QMainWindow
{
    Q_OBJECT

    int size = 1;
    int w = 28 * size;
    int h = 28 * size;
    int scale = 20 / size;

    enum Mode { Digits, Letters, Shapes};
    Mode currentMode = Digits;

public:
    paint(QWidget *parent = nullptr);
    ~paint();

private:
    Ui::paint *ui;

    const int DRAW_AREA_WIDTH = 28 * 20;  // 28 пикселей * масштаб 20
    const int DRAW_AREA_HEIGHT = 28 * 20;
```

Продовження додаток В

```
    DatabaseManager* dbManager;

    QProgressBar* letterProgressBars[26]; // 26 великих
    QProgressBar* digitProgressBars[10];  // 10 цифр
    QProgressBar* shapeProgressBars[4];   // 4 фігури

    QImage image_size;

    INerualNetwork *NeuralNet; //Мережа для цифр
    INerualNetwork *NeuralNetDigits; //Мережа для цифр
    INerualNetwork *NeuralNetLetters;  // Мережа для букв
    INerualNetwork *NeuralNetShapes;   // Мережа для фігур

    float inputs[28 * 28];
    float result[10];//Для цифр
    float resultLetters[26];  // 26 великих + 26 малих літер + 10 цифр
    float resultShapes[4];   // 10 геометричних фігур

    float mx, my;
    int mousePressed;

    void updateResultDisplay();
private:
    /* Переопределяем событие изменения размера окна
     * для пересчёта размеров графической сцены
     * */
    void resizeEvent(QResizeEvent *event);

    // Для рисования используем события мыши
```

Продовження додаток В

```
  void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);

    QCPGraph *AddGraph(QString func_name, QVector<double> _x,
QVector<double> _y);

private slots:
    //void handleLoadButtonClick();
    void push_button_clear();
   // void push_button_load();//Стандартная функция
    void push_button_init_random_weigths();

    //Добавленые ботом
    void push_button_load();
    //void push_button_load_digits();
    //void push_button_load_letters();
    //void push_button_load_shapes();
    void on_modeChanged(int index);
    //void showTrainingHistory();
    //void showDatabaseContent();

    void on_actionsave_triggered();
    void on_actionload_triggered();
    void on_showdatabasecontent_triggered();
    void on_loadWeightsFromDb_triggered();
    void on_saveCurrentWeights_triggered();
};
#endif // PAINT_H
```