

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ**  
Навчально-науковий інститут Інформаційних технологій  
Кафедра інженерії програмного забезпечення

**Пояснювальна записка**

до бакалаврської роботи  
на ступінь вищої освіти бакалавр

на тему: **«РОЗРОБКА КОНСТРУКТОРА НЕЙРОННИХ МЕРЕЖ ІЗ  
КОНТРОЛЬОВАНИМ НАВЧАННЯМ МОВОЮ C++»**

Виконав: студент 4 курсу, групи ПД-41  
спеціальності

121 Інженерія програмного забезпечення

(шифр і назва спеціальності)

Гапей М.Ю.

(прізвище та ініціали)

Керівник \_\_\_\_\_

Фесенко М.А.

(прізвище та ініціали)

Рецензент \_\_\_\_\_

(прізвище та ініціали)

Київ – 2023



- 5.2 Задачі дипломної роботи
- 5.3 Аналіз аналогів
- 5.4 Вимоги до програмного забезпечення
- 5.5 Програмні та технічні засоби реалізації
- 5.6 Діаграма варіантів використання
- 5.7 Діаграма класів
- 5.8 Діаграма діяльності
- 5.9 Екранні форми
- 5.10 Апробація результатів дослідження
- 5.11 Висновки

Дата видачі завдання «25» лютого 2023 року

### КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	25.02.2023-05.03.2023	Виконано
2	Аналіз та дослідження існуючих аналогів	06.03.2023-09.03.2023	Виконано
3	Дослідження програмних засобів	10.03.2023-12.03.2023	Виконано
4	Моделювання об'єкта проектування	13.03.2023-19.03.2023	Виконано
5	Розробка функціонала додатка	20.03.2023-14.05.2023	Виконано
6	Вступ, висновки, реферат	15.05.2023-18.05.2023	Виконано
7	Розробка обов'язкових демонстраційних матеріалів	19.05.2023-21.05.2023	Виконано
8	Попередній захист роботи	22.05.2023	Виконано
9	Здача роботи	01.06.2023	Виконано

Студент \_\_\_\_\_ Гапей М.Ю.  
( підпис ) (прізвище та ініціали)

Керівник роботи \_\_\_\_\_ Фесенко М.А.  
( підпис ) (прізвище та ініціали)





## РЕФЕРАТ

Текстова частина бакалаврської роботи: 48 с., 1 табл., 23 рис., 30 джерела  
C++, QT FRAMEWORK, SERIALIZATION FIXTURE, QT CREATOR,  
NEURAL NETWORK, FULLY CONNECTED, CONVOLUTIONAL.

Об'єкт дослідження – процес створення нейронних мереж із контрольованим навчанням на базі Qt framework.

Предмет дослідження – комп'ютерний додаток для створення нейронних мереж із контрольованим навчанням.

Мета роботи – спрощення процесу конструювання нейронних мереж із контрольованим навчанням за допомогою комп'ютерного додатку, розробленого мовою C++.

Методи дослідження – методи розробки та проектування програмного забезпечення, методи обробки вхідних даних, методи тестування програмного забезпечення.

В роботі розглянуті та проаналізовані існуючі аналоги програмного забезпечення для конструювання нейронних мереж. Досліджено методи створення багат шарових нейронних мереж із керованим навчанням. Розроблено спосіб представлення топології та її експлуатації для кінцевого користувача.

Галузь використання – складова штучного інтелекту, аналіз даних.

## ЗМІСТ

<b>ВСТУП</b> .....	<b>9</b>
<b>1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ</b> .....	<b>10</b>
<b>1.1 Нейронні мережі</b> .....	<b>10</b>
1.1.1 Що таке нейронна мережа.....	10
1.1.2 Штучний нейрон.....	11
1.1.3 Види шарів.....	12
1.1.4 Активація сигналів .....	13
1.1.5 Види нейронних мереж .....	14
1.1.6 Види навчання .....	16
1.1.7 Лінійна та поліноміальна регресії.....	18
<b>1.2 Порівняння аналогів</b> .....	<b>19</b>
1.2.1 Deep Learning Toolbox .....	19
1.2.2 Java Neural Network Framework Neuroph .....	20
1.2.3 Easy Neural Network Writer .....	21
<b>2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ</b> .....	<b>23</b>
2.1 Вибір стеку технологій.....	23
2.2 Середовище розробки Qt Creator .....	24
2.3 Мова програмування C++ .....	26
2.4 Фреймворк Qt .....	28
2.5 Бібліотека Serialization Fixture .....	29
<b>3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ КОНСТРУКТОРА НЕЙРОННИХ   МЕРЕЖ ІЗ КОНТРОЛЬОВАНИМ</b> .....	<b>30</b>
3.1 Патерни проєктування .....	30
3.2 API архітектура.....	32
3.3 Проєктування GUI .....	40
3.2 Реалізація та тестування .....	41
<b>ВИСНОВКИ</b> .....	<b>48</b>
<b>ПЕРЕЛІК ПОСИЛАНЬ</b> .....	<b>49</b>
<b>ДОДАТОК А</b> .....	<b>52</b>
<b>ДОДАТОК Б</b> .....	<b>59</b>

## **ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ**

NN – neural network

CNN – convolutional neural network

GUI – graphical user interface

ШІ – штучний інтелект

ПЗ – програмне забезпечення



## ВСТУП

З часів зародження програмування, складність постановлених задач поступово зростала, як і методи їх вирішення. Деякі із них було неможливо реалізувати традиційним (алгоритмічним) способом. Тому на заміну стандартних алгоритмів прийшли нейронні мережі. Інтерфейс взаємодії однаковий, лише за одним виключенням – реалізація алгоритму делегується нейронній мережі.

Однак технічно нейронні мережі є похідними від багатовимірних математичних функцій, процес її створення потребує базових теоретичних знань, розуміння принципів роботи, а також набір необхідних інструментів розробки нейронних мереж.

Саме простота інструментів – є вагомим фактором вибору, оскільки грає ключову роль, коли необхідно швидко досягнути поставленої мети.

Популярність нейронних мереж зростає з кожним днем, через свою многогранність та практичність. Процес їх створення повинен бути таким же зручним, як і використання.

Підводячи підсумки проблеми, наведеної вище, щоб прискорити процес розробки, було вирішено розробити комп'ютерний додаток для створення нейронних мереж із контрольованим навчанням на базі Qt framework.

Була поставлена мета створити комп'ютерний додаток, який дає змогу швидко та просто створювати власні нейронні мережі із керованим навчанням, без необхідності вивчати сторонні бібліотеки та фреймворки.

Опираючись на поставлену мету, були визначені наступні задачі:

- аналіз обраної предметної області;
- порівняння існуючих аналогів програмного забезпечення;
- проектування та реалізація конструктора нейронних мереж із контрольованим;
- тестування отриманого додатка.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1 Нейронні мережі

Нейронні мережі мають багату історію, яка сягає середини 20 століття. Концепція нейронних мереж була вподобана біологічною структурою [2] та функціями людського мозку. Ідея полягала в тому, щоб розробити обчислювальні моделі, які могли б імітувати те, як мозок обробляє інформацію та навчається на досвіді.

Однією з перших моделей нейронних мереж був перцептрон, розроблений психологом Френком Розенблаттом у 1958 році [3]. Перцептрон був типом прямої нейронної мережі (Fully connected NN) [7], яка могла навчитися класифікувати шаблони на основі їхніх особливостей. Однак він мав обмеження у здатності вирішувати складніші проблеми.

Одним із головних досягнень цього часу стала розробка глибоких нейронних мереж, що дозволило створювати складні моделі з багатьма шарами. Це стало можливим частково завдяки наявності великих обсягів даних і розробці TUP (Tensor Processing Unit) та GPU (Graphisc Processing Unit), які могли б виконувати матричні обчислення набагато швидше, ніж традиційні CPU (Central processing unit).

### 1.1.1 Що таке нейронна мережа

Нейронна мережа (англ. Neural network) — це тип комп'ютерної програми, яка навчається на основі даних [1]. Для обробки інформації, вона імітує роботу людського мозку за допомогою взаємопов'язаних вузлів, які називаються нейронами.

Мережа організована на шари, кожен з яких виконує певне завдання. Перший шар приймає вхідні дані, наприклад зображення чи текст, а наступні шари обробляють і перетворюють дані для створення вихідних прогнозів (дивитися рисунок 1.1).

Під час навчання, мережа регулює інтенсивність зв'язків між нейронами на основі отриманих даних. Це дозволяє мережі визначати закономірності та робити прогнози на основі нових даних, яких вона раніше не бачила.

Нейронні мережі використовуються в різноманітних сферах, таких як розпізнавання образів звуків, обробка людської мови, прогнозування та ШІ.



Рисунок 1.1 – Представлення нейронної мережі у вигляді чорної коробки

### 1.1.2 Штучний нейрон

Штучний нейрон — це основна обчислювальна одиниця, створена для імітації поведінки біологічного нейрона в мозку людини [14]. Він приймає один або кілька вхідних даних, виконує математичні розрахунки і повертає результат обчислення (дивитися рисунок 1.2).

Основними одиницями для виконання математичних обрахунків є ваги (weight) та зміщення (bias). Для підвищення варіативності результатів, обрахунки передають функціям активації, які можуть посилювати або послаблювати значимість вхідних даних. Регулювати ваги та зміщення штучних нейронів — означає навчати нейронну мережу.

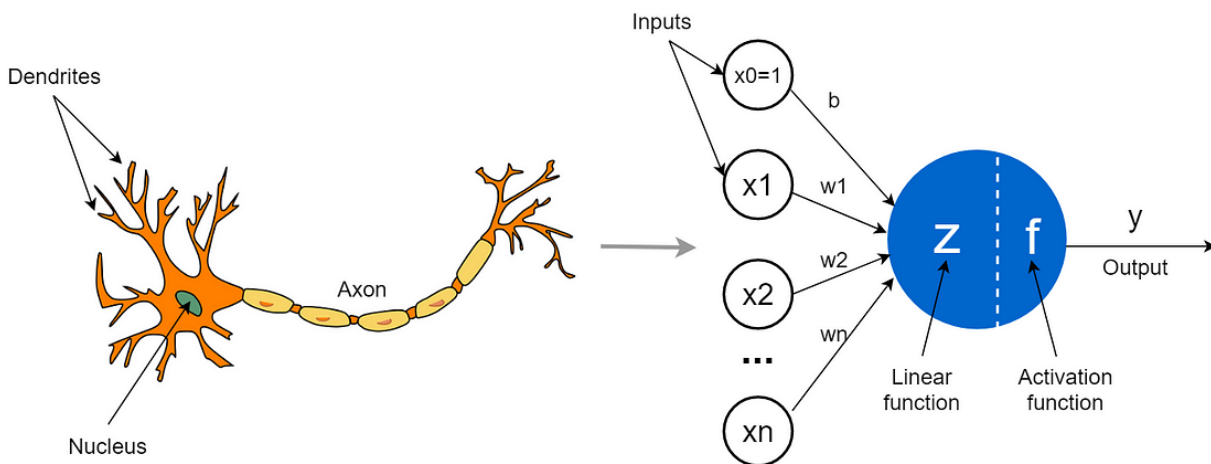


Рисунок 1.2 – Візуальне представлення біологічного та штучного нейрона

### 1.1.3 Види шарів

Нейронна мережа складається мінімум з двох або більше шарів штучних нейронів з'єднаних між собою [6]. Кожен шар мережі відповідає за різний тип обчислень.

Вхідний шар є першим рівнем мережі та відповідає за отримання вхідних даних, таких як зображення, текст чи звук. Всі дані форматуються в числові, для можливості бути переданими в нейронну мережу.

Приховані шари — це шари, розташовані між вхідним і вихідним шарами та виконують обчислення над вхідними даними. Кожен прихований шар складається з набору штучних нейронів, які обробляють вхідні дані попереднього рівня та передають вихідні дані на наступний рівень.

Вихідний шар є останнім рівнем мережі, який створює прогнози або вихідні дані мережі на основі обчислень, виконаних попередніми рівнями.

Кількість прихованих шарів і число нейронів в них, може змінюватися в залежності від складності розв'язуваної задачі. Нейронні мережі з більшою кількістю прихованих шарів, як правило, краще підходять для вирішення складних проблем, тоді як мережі з меншою кількістю прихованих шарів можуть бути достатніми для вирішення простих. На рисунку 1.3 показано модель з двома прихованими шарами.

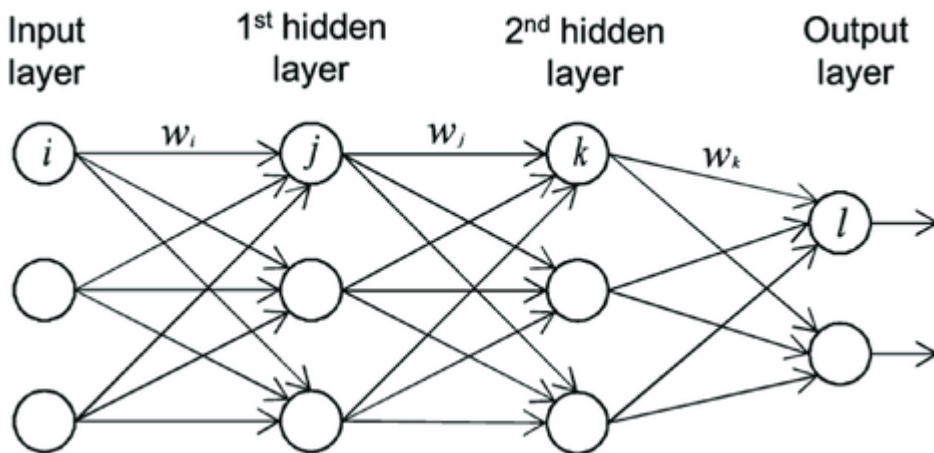


Рисунок 1.3 – Нейронна мережа з трьома повнозв'язними шарами

### 1.1.4 Активація сигналів

В рамках нейронної мережі функція активації — це математична функція, яка застосовується для додаткової обробки даних кожного нейрона. Функція активації визначає, чи має бути активований нейрон, і яке вихідне значення має бути отримано. Метою функції активації є введення нелінійності на рівні нейронної мережі. Без функцій активації, результат обчислень нейронів міг суттєво обмежити можливості нейронної мережі [27].

Існує декілька основних типів функцій активації, які можна використовувати в нейронній мережі, зокрема:

- Функція ReLU (англ. Rectified Linear Unit): функція, яка повертає 0 для від'ємних вхідних значень і вихід рівний вхідним для додатних вхідних значень. Ця функція зазвичай використовується в мережах глибокого навчання, оскільки значно прискорює процес обчислень.
- Сигмоїдна функція (англ. Sigmoid): функція, яка створює S-подібну криву та зазвичай використовується в задачах двокласової (бінарної) класифікації.
- Функція гіперболічного тангенса (англ. Tanh): подібна до сигмоїдної функції, використовується в прихованих шарах, є особливо корисною у глибоких нейронних мережах, де вхідні значення для нейрона можуть бути дуже великими або дуже малими.

На рисунку 1.4 представлено графіки наведених функцій на області значень дійсних чисел  $[-6; 6]$ .

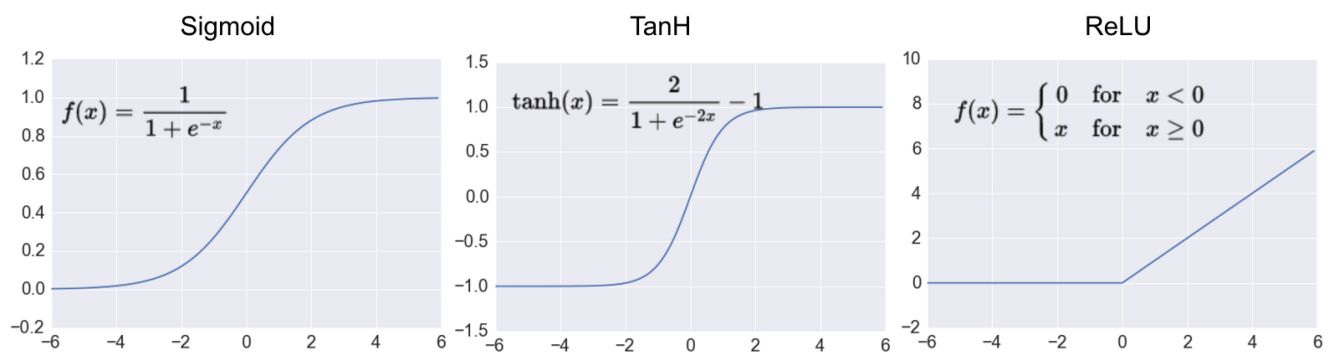


Рисунок 1.4 – Графіки функцій активації та їх математичне представлення

Також використовують функцію розподілу ймовірностей Softmax - функція зазвичай використовується в задачах багатокласової класифікації. На рисунку 1.5 проілюстровано визначення функції для кожного вихідного нейрона, де  $Z_j$  – значення  $j$ -того вихідного нейрона,  $K$  – кількість класів класифікації.

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \text{ for } j = 1, \dots, K$$

Рисунок 1.5 – Математичне представлення функції Softmax

Зазвичай ReLU і Tanh використовують у прихованих шарах, а сигмоїдну та Softmax функції у вихідному шарі. Функція активації для кожного шару нейронної мережі може мати значний вплив на її продуктивність, тому вибір відповідної функції для конкретної проблеми - є важливою частиною проектування нейронної мережі.

### 1.1.5 Види нейронних мереж

Нейронні мережі прямого зв'язку (Fully connected NN): це найпростіший і найпоширеніший тип нейронних мереж, де дані проходять лише в одному напрямку від входу до виходу [7]. Вони можуть мати один або кілька прихованих шарів між вхідним і вихідним (дивитися рисунок 1.6). Мережі прямого зв'язку зазвичай використовуються для завдань класифікації та регресії.

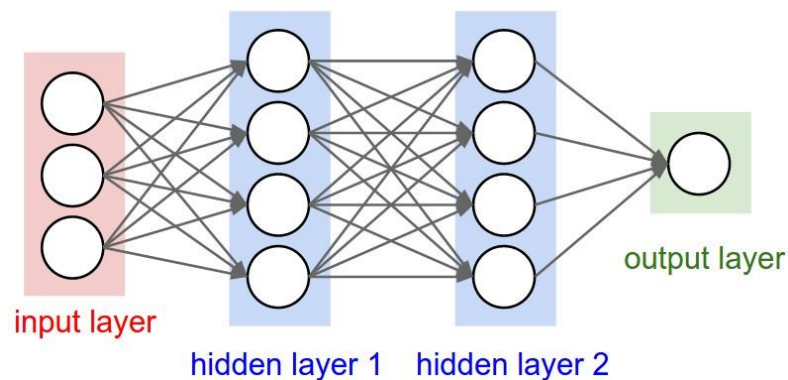


Рисунок 1.6 – Схематичний приклад нейронної мережі прямого зв'язку

Згорткові нейронні мережі (англ. Convolutional NN) - це спеціалізований тип нейронної мережі, який зазвичай використовується для завдань обробки зображень і відео (дивитися рисунок 1.7) [8]. CNN використовують згорткові шари для виявлення локальних шаблонів та особливостей у вхідних даних, а також шари об'єднання (англ. Pooling) для зменшення просторових розмірів виходу. CNN досягли найсучаснішої продуктивності в задачах обробки багатовимірних даних.

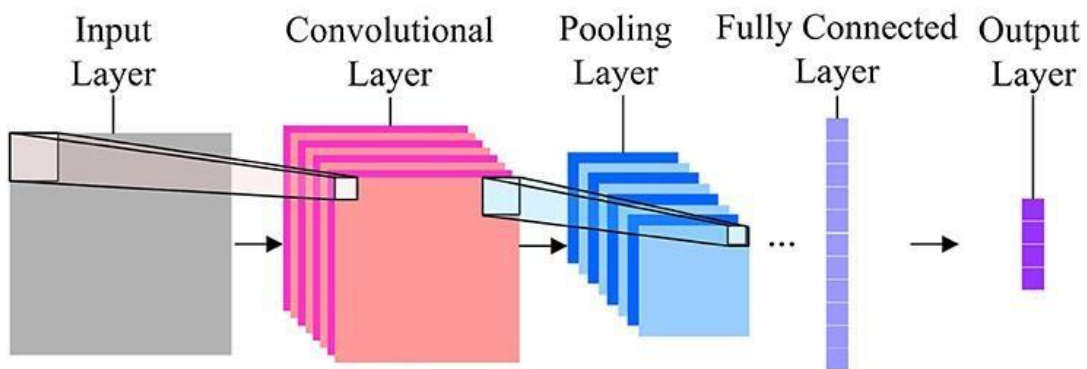


Рисунок 1.7 – Схематичний приклад згорткової нейронної мережі

Генеративні змагальні мережі (англ. Generative adversarial networks) - це тип нейронної мережі, яка складається з двох частин: генератора та дискримінатора [13]. Генератор генерує подробилені дані, а дискримінатор намагається відрізнити ці дані від справжніх. Обидві частини навчаються разом, щоб генератор створював дані, які неможливо відрізнити від реальних даних. GAN зазвичай використовуються для створення реалістичних зображень, відео та аудіо.

Повторювані нейронні мережі (англ. Recurrent NN) - це тип нейронної мережі, яка може обробляти послідовності вхідних даних, наприклад аудіо та текст [11]. RNN використовують форму зворотного зв'язку, яка дозволяє передавати інформацію від одного кроку послідовності до наступного (дивитися рисунок 1.8). Цей механізм зворотного зв'язку дозволяє RNN моделювати тимчасові залежності, що значно покращує розпізнавання та моделювання мови або інших завдань, пов'язаних із послідовністю, що не під силу іншим типам нейронних.

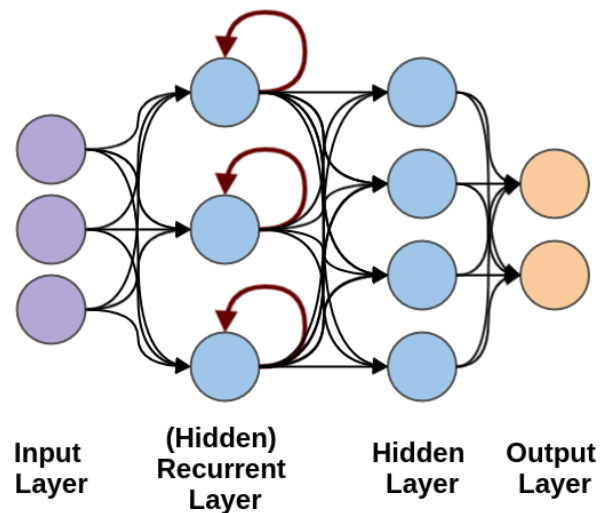


Рисунок 1.8 – Схематичної приклад рекурентної нейронної мережі

Наведені архітектури нейронних є одні з найпоширеніших, вибір залежить від конкретної задачі, що вирішується, і даних, що використовуються. На практиці, використовують змішані типи нейронних мереж для досягнення більшої гнучкості.

### 1.1.6 Види навчання

Контрольоване навчання (англ. Supervised learning) - це найпоширеніший тип навчання нейронної мережі, який відбувається за допомогою позначених даних [4]. Вхідними дані пов'язані відповідними бажаними виходами, тому процес навчання мережі супроводжується мінімізацією різниці між ними. Контрольоване навчання зазвичай використовується для таких завдань, як класифікація зображень, виявлення об'єктів і розпізнавання мовлення.

Неконтрольоване навчання (англ. Unsupervised learning) - цей тип навчання нейронної мережі передбачає навчання мережі за допомогою не позначених даних [4]. Процес навчання полягає у знаходженні нейронною мережею закономірності даних без будь-яких попередніх знань про них. Неконтрольоване навчання зазвичай використовується для таких завдань, як кластеризація (автоматична класифікація), зменшення розмірності (зменшення випадкових значень) та виявлення аномалій (дивитися рисунок 1.9).



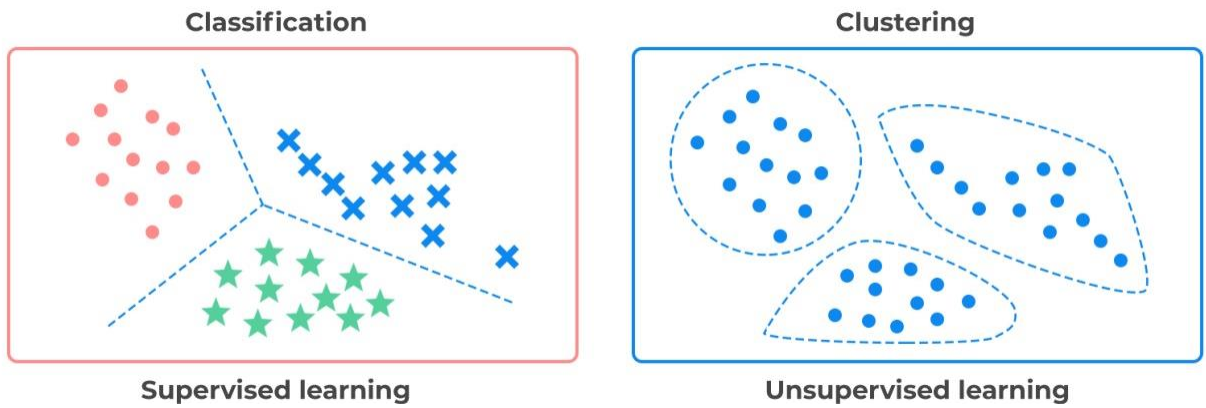


Рисунок 1.9 – Різниця між контрольованим та неконтрольованим навчанням

Навчання з підкріпленням (англ. Reinforcement learning) – цей тип навчання нейронної мережі передбачає навчання мережі приймати рішення на основі зворотнього зв'язку з її середовища [5]. Мережа представлена набором можливих дій (для взаємодії із середовищем), і отримує винагороди або покарання на основі тих, які вона виконує. Мережа вчиться максимізувати винагороди, які вона отримує з часом. Навчання з підкріпленням зазвичай використовується для таких завдань, як ігри, робототехніка та автономне керування (водіння).

На рисунку 1.10 схематично показано зв'язок між нейронною мережею (агентом) та середовищем. На практиці середовищем може бути комп'ютерна гра або програма, яка симулює деякі елементи з реального або віртуального світу. На прикладі гри 2048 – середовищем буде ігровий двигун, який реагує на можливі дії користувача: вліво, вправо, вгору та вниз. В свою чергу дані з ігрового поля розміром 4x4 передаються в модель, яка на виході повертає ймовірності цих дій на поточному кроці.

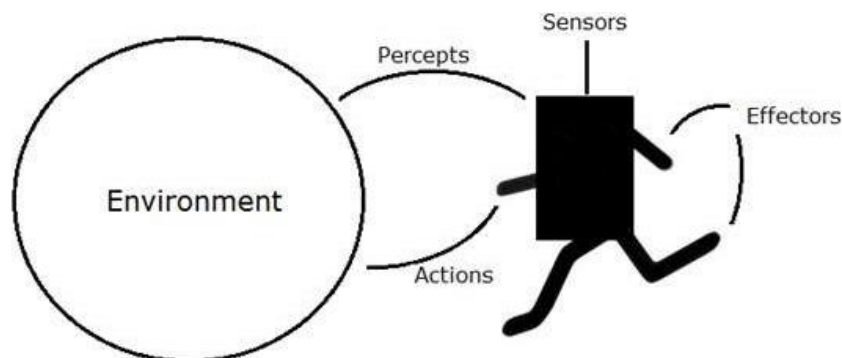


Рисунок 1.10 – Ілюстрація середовища та її впливу на агента

### 1.1.7 Лінійна та поліноміальна регресії

На практиці прості нейронні мережі для класифікації даних можна замінити на лінійну та поліноміальну регресії [12]. Основна особливість яких полягає у швидкості навчання (знаходження значень коефіцієнтів регресії), оскільки для корегування внутрішніх параметрів регресії достатньо однієї ітерації.

Лінійну регресію слід використовувати для багатьох показників які відповідають одному деякому значенню. Наприклад за допомогою регресії можна визначити стать особи за критеріями росту та ваги. Для лінійної регресії є ймовірність отримати гіршу апроксимацію ніж у випадку з поліноміальною регресією (читати далі).

Поліноміальну регресію слід застосовувати для апроксимації деякого набору точок. У випадку з нейронними мережами – точка задає input (деякий критерій) і target (відповідна критерію величина). Потужністю регресії називають максимальну степінь многочлена для рівняння апроксимації (кількість точок перегину рівної  $k-1$ ). Простими словами, ця величина задає максимально-допустиму кривизну лінії.

На рисунку 1.11 наочно показано точність апроксимації лінійної та поліноміальної функцій регресії.

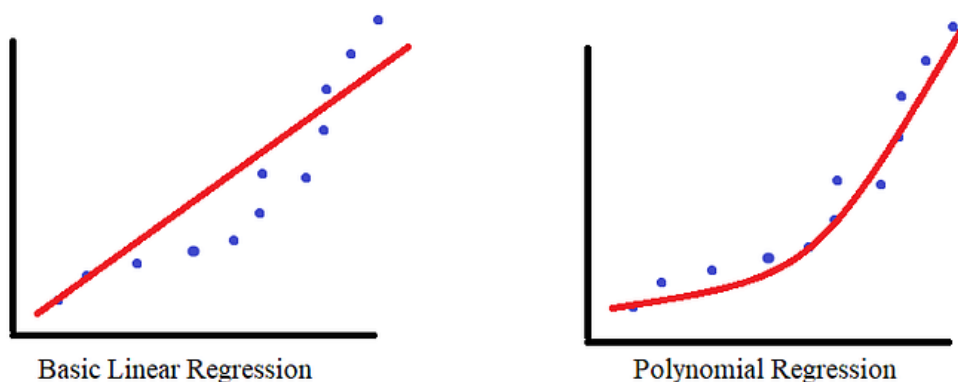


Рисунок 1.11 – Різниця між лінійною та поліноміальною регресією

Процес знаходження коефіцієнтів для обох видів регресії відбувається шляхом знаходження мінімуму функції залишкової сума квадратів RSS (Residual sum of squares).

## 1.2 Порівняння аналогів

Для максимально ефективного створення та навчання нейронних мереж повинно виконуватися ряд умов, які представляють користувачеві широкий набір інструментів, а саме:

- Формат вхідних даних: 1D, 2D та 3D;
- Імпорт та експорт нейронних мереж;
- Шарів різних типів: Fully Connected, Convolutional, MaxPooling;
- Функції активації, нормалізації та похибки;
- Алгоритми навчання: batch, mini-batch, stochastic;
- Оптимізатори алгоритмів навчання.

Більша частина існуючих додатків відповідають вище зазначеним умовам, тому було обрано найпоширеніші із них (дивитися рисунок 1.12), а саме: Deep Learning Toolbox (бібліотека широкого призначення на базі MATLAB), Java Neural Network Framework Neuroph (фреймворк з відкритим кодом для підприємств на основі Java розробки) та Easy Neural Network Writer (декстопний застосунок для операційної системи Windows) Проаналізуємо та порівняємо вище згадані програми.

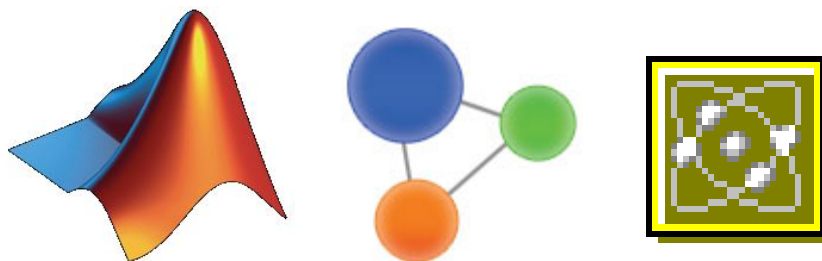


Рисунок 1.12 – Логотипи програм Deep Learning Toolbox, Neuroph та Easy Neural Network Writer

### 1.2.1 Deep Learning Toolbox

Deep Learning Toolbox — це бібліотека для MATLAB, середовище чисельних обчислень, яке надає повний набір інструментів для проектування,

навчання та розгортання (deploy) глибоких нейронних мереж. Офіційний сайт бібліотеки [15].

Набір інструментів містить широкий спектр готових архітектурних рішень нейронних мереж, таких як: згорткові нейронні мережі (Convolutional neural network), рекурентні нейронні мережі (Recurrent neural network) і генеративні змагальні мережі (Generative adversarial network).

Бібліотека також містить багато популярних функцій активації, а також функцій похибок та алгоритмів оптимізації (для навчання).

Набір інструментів надає простий у використанні графічний інтерфейс користувача для створення, навчання та тестування нейронних мереж. Графічний інтерфейс дозволяє користувачам візуально проектувати нейронні мережі, налаштовувати їх параметри та контролювати прогрес навчання в режимі реального часу. Крім того, інструментарій надає інтерфейс програмування, що дозволяє користувачам створювати власні архітектурні рішення для нейронних мереж і навчальні алгоритми за допомогою коду MATLAB.

Загалом Deep Learning Toolbox надає потужну та гнучку платформу для вивчення та розробки моделей (нейронних мереж) глибокого навчання.

### **1.2.2 Java Neural Network Framework Neuroph**

Neuroph — це Java фреймворк з відкритим кодом для конструювання та навчання нейронних мереж. Він був розроблений факультетом електротехніки та обчислювальної техніки Загребського університету в Хорватії. Офіційний сайт додатка [16].

Фреймворк надає набір класів Java для створення, навчання та тестування нейронних мереж. Підтримує низку архітектур нейронних мереж, включаючи нейронні мережі прямого зв'язку (Fully connected neural networks), багат шарові перцептрони (Multilayer perceptrones), згорткові нейронні мережі (Convolutional neural networks) та самоорганізуючі карти (Self-organizing maps).

Neuroph надає простий у використанні графічний інтерфейс користувача для створення та навчання нейронних мереж, а також інтерфейс командного рядка

для професійних користувачів. Графічний інтерфейс дозволяє відслідковувати стан моделі та налаштовувати її параметри, такі як: кількість нейронів, швидкість навчання, функції активації тощо.

Neuroph підтримує імпорт та експорт моделей в різних форматах, таких як XML та JSON, які можна використовувати для інтеграції навчених моделей в інші програми.

В цілому Neuroph є корисним інструментом для дослідників та розробників, які хочуть експериментувати з нейронними мережами та створювати додатки, які їх використовують.

### **1.2.3 Easy Neural Network Writer**

Easy Neural Network Writer (скорочено EasyNNW) — це програма, призначена для створення та навчання штучних нейронних мереж. Вона доступний для операційних систем Windows та розроблене Neural Planner Software. Офіціальний сайт додатка [18].

Додаток EasyNNW надає зручний інтерфейс для створення та налаштування моделей нейронної. Набір інструментів містить широкий спектр готових архітектурних рішень нейронних мереж, таких як: згорткові нейронні мережі (Convolutional neural networks), рекурентні нейронні мережі (Recurrent neural network) та мережі Хопфілда (Hopfield networks) [25] та багатошарові нейронні мережі (Deep neural networks). А оскільки реалізація нейронних мереж відбувається у вигляді незалежних компонентів, а наявність відкритого коду дозволяє легко створювати власні, а також інтегрувати їх в дослідницькі програми та системи.

Також додаток містить інструменти для попередньої обробки, вилучення та візуалізації даних, що полегшує роботу з великими наборами даних даючи можливість аналізувати роботу моделі.

Однією з визначних особливостей EasyNNW - це здатність генерувати код C із навчених моделей нейронної мережі, що дозволяє розробникам легко інтегрувати нейронні мережі у власні програми.

EasyNNW — не тільки потужний та універсальний інструмент для конструювання та навчання нейронних мереж, він підходить як для початківців, так і для досвідчених користувачів, що є великою перевагою серед аналогів.

Перелік ключових відмінностей усіх наведених вище аналогів продемонстровано в таблиці 1.1.

Таблиця 1.1 – Зведені результати аналізу програмного забезпечення для конструювання нейронних мереж

Показник	Deep Learning Toolbox	Java Neural Network Framework Neuroph	Easy Neural Network Writer	Neural Network Constructor
Платформи	MATLAB	Віртуальна машина Java	Windows	Windows, Linux, macOS
Категорії витрат	Платний	Безкоштовний	Безкоштовний / Платний	Безкоштовний
Формати зберігання	MATLAB, HDF5, ONNX	XML та JSON	CSV, Excel, MATLAB	Бінарний формат
Архітектури нейронних мереж	NN, CNN, RNN, GAN	NN, CNN, MLP, SOM	NN, CNN, RNN, Hopfield Network, DNN	NN, CNN, Deep NN
Зручність використання	Комплексний	Простий та інтуїтивний	Послідовний	Мінімалістичний
Візуалізація моделі	Tile формат	Tile формат із нейронними зв'язками	Немає	Blueprint формат
Особливості	Інтерфейс програмування	Архітектура плагіна	Генерація коду C із навчених моделей	Відкритий вихідний код ядра

## 2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ

### 2.1 Вибір стеку технологій

Навчання нейронних мереж потребує ефективного використання апаратних ресурсів комп'ютера. При цьому важливу роль відіграє гнучкість використовуваних інструментів. Оскільки граничні обмеження можуть значно уповільнити або зупинити процес розробки, що призведе до значних втрат часу та ресурсів. Вагомим фактором також є не лише гнучкість, але й простота. Надмірна складність може демотивувати процес розробки.

Виходячи з цього, для написання програми обрано одну із найпоширеніших мов низькорівневого програмування - мову C++. Для представлення візуальної частини було обрано фреймворк Qt, а для серілізації просту бібліотеку широкого призначення Serialization Fixture (скорочено SF).

На відміну від C++, мова C хоч і є її предком - головним недоліком вважається відсутність концепції RAII (Resource Acquisition Is Initialization), яка вважається головною складовою мови C++. Цей внутрішній механізм, гарантує прозору та прогнозовану тривалість життя усіх об'єктів. Оскільки створення тісно пов'язано зі знищенням, то у випадку, коли об'єктом є користувальницький тип – процес його знищення супроводжується поверненням усіх використаних ресурсів, назад до операційної системи. У випадку з C - складність розроблюваних програм гарантовано підвищується, тому що розробникам доводиться самостійно стежити за звільненням пам'яті, що часто призводить до її витоку (leak).

Фреймворк Qt обрано в першу чергу за наявності власного інтегрованого середовища розробки - Qt Creator, що розширює палітру можливостей розробників. Проаналізуємо обрані технічні та програмні засоби реалізації.

## 2.2 Середовище розробки Qt Creator

В якості середовища розробки, обрано інтегроване середовище розробки Qt Creator. Воно має мінімаліний набір компонентів та простий інтерфейс – це дозволяє новим користувачам швидко освоїти її. Програма дозволяє створювати як звичайні програми на C/C++ та і націлена на розробку з використанням Qt framework . Приклад інтерфейсу програми наведено на рисунку 2.1.

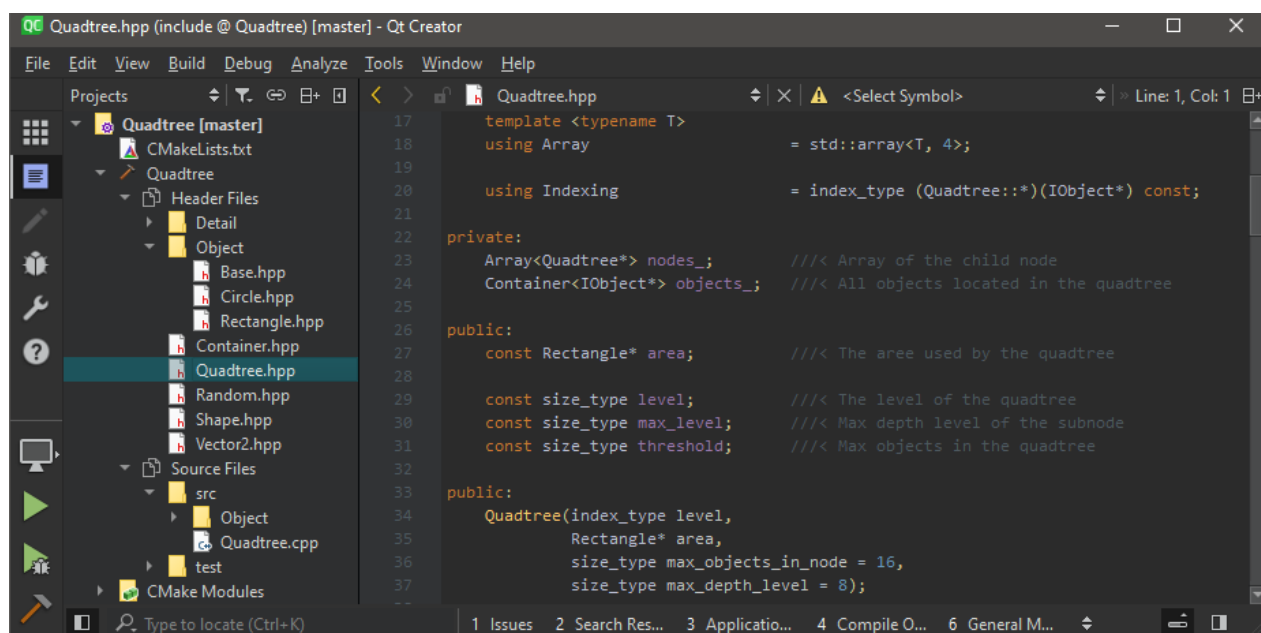


Рисунок 2.1 – Приклад інтерфейсу інтегрованого середовища розробки Qt Creator

Qt Creator має компонент Qt Designer - інструмент розробки графічного інтерфейсу користувача (GUI), який дозволяє розробникам створювати та проектувати форми для своїх програм [21]. Qt Designer надає інтерфейс перетягування (drag-and-drop) для створення елементів GUI, таких як кнопки, мітки, текстові поля та багато іншого. Потім ці елементи можна об'єднувати в більш комплексні віджети, щоб створити бажаний макет. Qt Designer дозволяє налаштовувати зовнішній вигляд інтерфейсу користувача за допомогою таблиць стилів, а також підтримує створення користувацьких віджетів.

Однією з ключових переваг Qt Designer є те, що він дозволяє розробникам відокремити дизайн інтерфейсу користувача від логіки програми. Це полегшує



підтримку та зміну інтерфейсу користувача без необхідності змінювати базовий код. Qt Designer також може генерувати код (як наприклад у Windows Form) кількома мовами програмування, такими як C++, Python і Java. Це заощаджує багато часу та зусиль розробників, оскільки в згенерований код інтерфейсу можна з легкістю інтегрувати бізнес логіку. На рисунку 2.2 зображено один із можливих варіантів інтерфейсу Qt Designer.

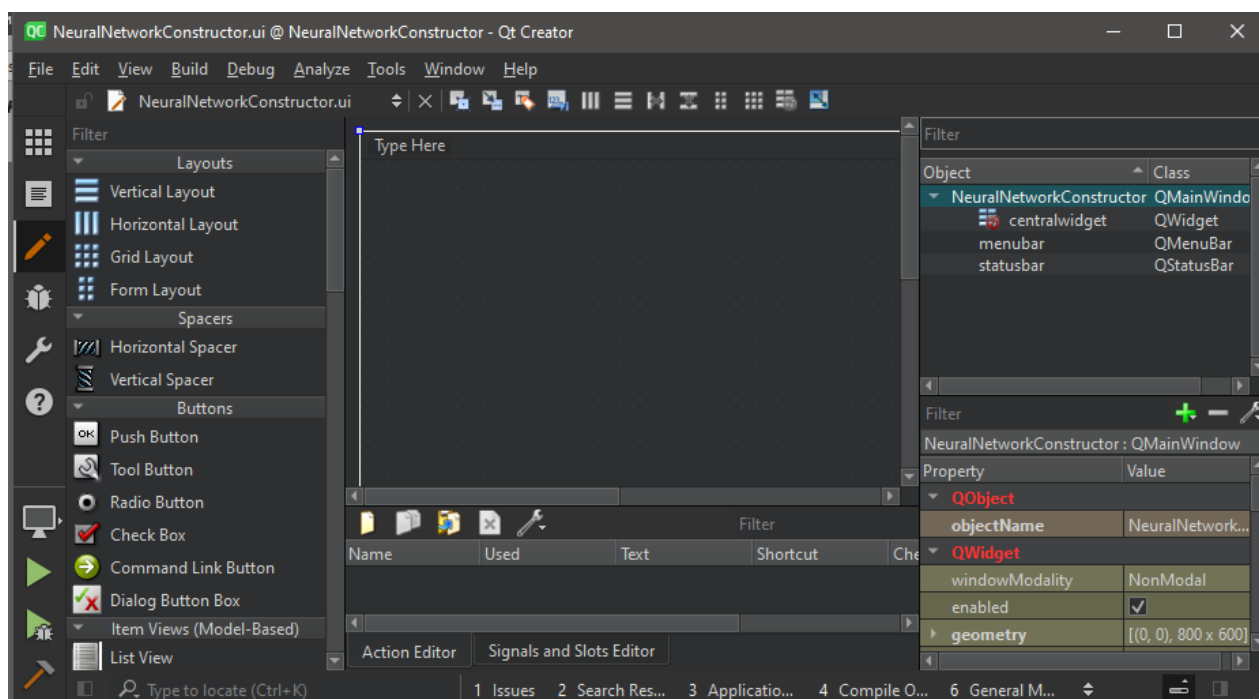


Рисунок 2.2 – Приклад інтерфейсу Qt Designer

Незважаючи на те, що Qt Creator є потужним і універсальним інтегрованим середовищем розробки, він має деякі недоліки, про які слід знати розробникам. Нижче наведені деякі із них:

- Qt Creator є власним програмним продуктом - це означає, що він не є абсолютно безкоштовним у використанні;
- Creator підтримує C++ та частково Python мови програмування - це може бути недоліком для розробників, які працюють з іншими мовами.
- Qt Creator може бути складним для початківців, які не знайомі з Qt framework або мовою програмування C++. Для ефективного використання потрібен певний рівень знань.

## 2.3 Мова програмування C++

C++ - мова програмування загального призначення, яка є вдосконаленням мови програмування C. Вона була розроблена Б'ярне Страуструпом у Bell Labs на початку 1980-х років. Офіційний сайт [17].

Мова підтримує концепцію як низькорівневого так і об'єктно-орієнтованого програмування. Поєднуючи ці дві концепції в одну – ми отримуємо гнучкий та потужний інструмент розробки, для такого спектру ПЗ як: ігри, операційні та вбудовані системи, комп'ютерні та мобільні застосунки, тощо.

Низькорівнева концепція підтримує такі елементи, як вказівники, керування пам'яттю та вбудоване програмування. Це дозволяє розробляти драйвера, мікроконтролери та навантажені системи, де важливу роль відіграє продуктивність виконуючого коду.

Об'єктно-орієнтована концепція підтримує такі елементи, як класи, об'єкти, успадкування, інкапсуляцію та поліморфізм, що полегшує процес написання складних та об'ємних програм.

Мова C++ має зворотну сумісність із C, що дозволяє розробникам використовувати код C у програмах на C++ і навпаки (для транслявання коду C++ в C використовують конструкцію extern "C").

C++ є потужною, але складною для вивчення мовою програмування. Однак ефективність та гнучкість роблять її популярним вибором для розробників, яким потрібен високопродуктивний код, і які хочуть скористатися розширеними можливостями мови (наприклад: placement new, type\_traits, atomic).

Деякі ключові обмеження мови станом на C++23 (це останній актуальний стандарт мови C++):

- На відміну від високорівневих мов програмування (Java, C#, Python), які мають механізм Garbage collection (збірка сміття), C++ пропонує розумні вказівники, які автоматично звільняють використану пам'ять;
- Відсутня підтримка Reflection (відображення) – розробники змушені створювати власні реалізації (хоч і сильно обмежені) або навіть втручатися у

роботу компілятора, щоб отримати інформацію з AST (Abstract Syntax Tree);

Також складність мови полягає у її багатозначності, деякі конструкції мови мають різну семантику, наприклад:

- Ключове слово `static`: воно може вказувати на тривалість життя об'єкта або змінювати тип зв'язування. Функція або змінна учасник, яка є статичною належить лише класу, статична функція яка не є учасником або глобальна змінна має внутрішній тип зв'язування. Доступ до всіх полів або функцій класу, які є статичними виконується через сам клас, або об'єкт (що не є обов'язковим);
- Ключове слово `inline` – вирішує проблему одного визначення (one definition rule), а також дає підказку для компілятора вбудовувати код програми в місці використання, замість того, щоб посилатися на нього.

Більшість функцій та можливостей мови не використовуються в повній мірі, оскільки це підвищує загальний рівень складності, але щоб писати хороший код, не потрібно досконало знати мову C++.

У порівнянні з іншими мовами програмування, станом на 2023 рік, C++ посідає лише 4 місце за популярністю (дивитися рисунок 2.3) [22].




May 2023	May 2022	Change	Programming Language	Ratings	Change
1	1		 Python	13.45%	+0.71%
2	2		 C	13.35%	+1.76%
3	3		 Java	12.22%	+1.22%
4	4		 C++	11.96%	+3.13%
5	5		 C#	7.43%	+1.04%

Рисунок 2.3 – Рейтинг популярності мов програмування

Конкурентом C++ є доволі молода, але перспективна мова Rust, яка поки що займає 17 місце у рейтингу популярних мов програмування. Вона позбавлена помилок, які виникали в процесі розвитку C++, а саме: зворотна сумісність, генерація спец функції, `auto_ptr` тощо. Не зважаючи на це, мова C++ надає великі перспективи та можливості молодим розробникам, які хочуть себе випробувати.

## 2.4 Фреймворк Qt

Фреймворк Qt — популярний міжплатформенний інструмент для розробки додатків із відкритим кодом. Він був розроблений компанією Qt (раніше Trolltech) і використовується для створення широкого спектру програм для настільних (декстопних), мобільних і вбудованих систем. Офіціальний сайт [19].

Фреймворк Qt надає набір бібліотек та інструментів C++ для розробки додатків із графічним інтерфейсом користувача (GUI), мережевого програмування, доступу до бази даних, мультимедіа тощо. Ключові особливості фреймворку:

- Мульти платформенність - Windows, macOS, Linux, Android та iOS;
- Модульний дизайн - дозволяє розробникам використовувати лише ті компоненти (бібліотеки), які їм необхідні, що суцільно зменшує розмір результуючої програми;
- Розширювана архітектура - дозволяє розробникам легко унаслідувати та додавати нові функції до своїх програм;
- Прості у використанні інструменти проектування - дозволяють розробникам створювати графічні інтерфейси користувача за допомогою інтерфейсу drag-and-drop;

Qt - це перш за все фреймворком C++, але він також забезпечує підтримку інших мов програмування, таких як: Python, Java та .Net C#.

Незважаючи на вище перераховані позитивні сторони Qt фреймворк також має ряд недоліків, ось деякі із них:

- Фреймворк Qt має комплексу кодову основу - це може заплутати нових розробників, які планують освоїти інструмент на прикладному рівні;
- Хоча фреймворк Qt загалом швидкий і ефективний, іноді можуть виникати проблеми продуктивності, потребує оптимізації або модифікації коду програми;
- Широка документація – це не тільки плюс, але й мінус – деяким розробникам може бути складно орієнтуватися в ній чи знаходити потрібну їм інформацію.

## 2.5 Бібліотека `Serialization Fixture`

SF (`Serialization Fixture`) — гнучка та розширювана бібліотека C++ із багатьма функціями, які дозволяють швидко та легко перетворювати об'єкти в послідовність байт. Офіційний сайт [20].

Гнучкість полягає у її модульній простоті та різноманітності використання. Основною перевагою, над аналогами, такими як `boost::serialization`, `cereal` та `zpp::bits` – є зрозумілий та простий інтерфейс використання, відсутня внутрішня залежність, проста інтеграція, мінімальна або відсутня необхідність використання макросів, підтримка потокових адаптерів, розділеної та серіалізації шаблонів, серіалізація агрегатних типів, `std::any` – `type erasure` (стирання типу), що дозволяє зберігати дані різних типів однією змінною.

Стандартні функції бібліотеки дозволяють максимально ефективно розпоряджатися ресурсами комп'ютера. А завдяки вбудованій системі хешування типів - мати можливість серіалізувати та десеріалізувати об'єкти на різних платформах.

Недоліками бібліотеки слід вважати відсутність підтримки популярних форматів збереження даних: JSON (`JavaScript Object Notation`) та XML (`Extensible Markup Language`). А оскільки бібліотека серіалізує дані у бінарному форматі, це може призвести до неможливості, підтримувати розповсюдження серіалізованих даних на різних платформах, де розміри стандартних інтегральних типів відрізняються (дробові числа, або числа з плаваючою комою не мають фіксованої версії, як у випадку з інтегральними типами, наприклад `int32_t`, `uint8_t`, `char16_t` та інші). Також бібліотека не має вбудованих засобів роботи з популярним інтернет протоколом RPC (`remote procedure call`). Це в свою чергу потребує від розробників власної реалізації, а також розуміння роботи внутрішнього механізму бібліотеки.

Не зважаючи на недоліки бібліотека має доволі просту і зрозумілу архітектуру, мінімальний набір незалежних компонентів, робить її чудовим вибором для більшості C++ розробників.

## 3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ КОНСТРУКТОРА НЕЙРОННИХ МЕРЕЖ ІЗ КОНТРОЛЬОВАНИМ

### 3.1 Патерни проєктування

При створенні програми було обрано один із найпопулярніших патернів проєктування - Model-View-Controller (скорочено MVC). Його основна ціль – відділити залежність двох основних компонентів рівня Model (модель) та рівня View (представлення). В свою чергу компонент Controller – це зв’язна ланка моделі та представлення. Реагуючи на дії користувача контролер посилає моделі нагадування про те, що щось змінилося. Модель хоч і нічого не знає про представлення, має сповіщати всіх наглядачів (observers) про зміни. Зазвичай події (events) впливають на декілька рівнів одночасно. На рисунку 3.1 представлено типовий шаблон взаємодії усіх трьох компонентів.

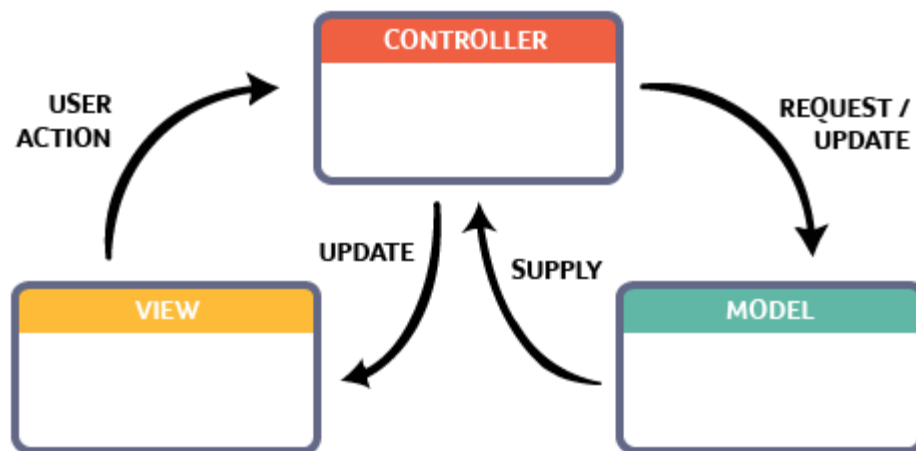


Рисунок 3.1 – Патерн проєктування MVC

Патерн «Стратегія» (англ. Strategy) є агрегатним компонентом системи і дозволяє створювати власну реалізацію маючи спільний інтерфейс. Засобами мови програмування C++ даний патерн можна реалізувати декількома способами, а саме через вказівник на функцію, `std::function`, лямбда вираз або віртуальну функцію. Деякі способи поступаються по швидкості, в деяких випадках через обмеження, перевага надається об'єктному стилю. В першу чергу, це пов'язано із

неможливістю серіалізувати вказівники на функції, оскільки функцію неможливо створити та зберегти.

Патерни «Ітератор» (англ. Iterator) та «Адаптер» (англ. Adapter) дозволяють імплементувати уніфіковану та варіативну взаємодію із об'єктами. Стандартна бібліотека шаблонів (англ. Standard Template Library) мови програмування C++ надає декілька типів ітераторів таких як: `input`, `output`, `random_access`, `forward` та `bidirectional` [28], а також адаптерів таких як `stack`, `queue`, `priority_queue` [29].

Патерн «Фабрика» (англ. Factory) забезпечує конструювання об'єктів різних типів, які об'єднані спільним інтерфейсом або абстракцією. Це дозволяє специфікувати поведінку створення об'єктів відповідно до потреб користувача.

Патер «Стан» (англ. State) специфікує поведінку об'єктів при виконанні послідовності деяких дій в залежності від стану цих об'єктів.

В процесі розробки, наведені вище патерни проектування [30] активно використовувалися для забезпечення гнучкої та модульної архітектури додатка. Патерн «Стратегія» дозволив створювати власні реалізації функцій активації та похибки, а також інтегрувати власні алгоритми розподілу не змінюючи при цьому код програми. Патерни «Адаптер» та «Ітератор» допомогли зберігати різні типи тензорів в одному контейнерному об'єкті, а також надав можливість уніфікувати взаємодію обробки їхніх даних. Також патерн «Адаптер» використано для створення власних потокових адаптерів серіалізації, а також спеціальних типів, які забезпечують відкритий та безпечний інтерфейс топології нейронної мережі. Патерн «Фабрика» надав простий інтерфейс створення різноманітних оптимізаторів навчання. Патер «Стан» дозволив відокремити різні поведінки розробляючої програми в залежності від обраного режиму та дій користувача.

Під час проектування також було використано деякі принципи SOLID [24]. Оскільки повне дотримання усім принципам може значно ускладнити архітектуру додатка, використовувалися лише деякі із них, а саме:

- Single Responsibility Principle – усі класи мають одну зону відповідальності. Це доволі простий та зрозумілий принцип, перешкоджає утворенню супер класів (швейцарських ножів);

- Open closed Principle – означає, що для розширення функціоналу не потрібно втручатися в логіку базових класів. Проблема, яку вирішує даний принцип, полягає у тому, що для розробки нового функціоналу погано спроектовані системи доводиться змінювати, що збільшує загальні витрати на розробку;

- Dependency Inversion Principle – це голосне правило забороняє утворювати залежності високих рівнів від низьких. Простими словами, взаємодія з кожним елементом системи, повинна відбуватися за допомогою інтерфейсів (тобто спільного шаблону).

Дотримання наведеним принципам дозволило спроектувати модульну та розширювальну архітектуру, де кожен компонент є незалежним від інших і може змінюватися в залежності від бізнес потреб.

### **3.2 API архітектура**

Створюючи додаток, було поставлено мету спростити процес створення нейронних мереж, що повинно бути можливим, не лише засобами графічного інтерфейсу (GUI), тому архітектура розробляючої програми була поділена на два рівні: ядро – бібліотека для машинного навчання, та представлення – графічна оболонка. В цьому розділі детально описується принцип роботи ядра програми.

Ядро - це проста та розширювана бібліотека, для машинного навчання, дозволяє навчати нейронні мережі різних типів будь-якої складності, а також користуватися перевагами регресійного аналізу. Але чи вистачить комп'ютерних ресурсів для досягнення бажаної мети? Під розширюваністю мається на увазі вдало спроектована архітектура бібліотеки, яка далася не відразу. Головною метою створення була мотивація зробити не просто бібліотеку як аналог іншим, а додати можливість налаштування не змінюючи код бібліотеки та втрат продуктивності. У цьому й полягає вся суть гнучкості. Ми не просто можемо використовувати готові компоненти, які надає бібліотека, але й створювати свої і з легкістю їх інтегрувати у існуючі. Простий приклад: будь-яка нейронна мережа



під час навчання весь можливий час витрачає на матричні обчислення, а саме множення матриць та векторів. Профілювання це підтверджує, більш ніж достатньо написати 3-х шарову повнозв'язну нейронну мережу середнього розміру приблизно з 1500 нейронами і відчутти, наскільки повільно просувається процес навчання (особливо на CPU). Тому в бібліотеці надається механізм створення своїх типів векторів і матриць, а також загального класу зв'язування операцій з них. І це лише мала частина того, що можна налаштувати.

*Поверхневий опис ключових нутроців бібліотеки.*

Як і будь-яка інша об'ємна бібліотека - має опорні класи для використання. В даному випадку, найголовнішим класом є TrixuNet. Створення нейронних мереж цього класу досить гнучке, тому що допускає різні види топології, яка може включати різні види шарів.

При цьому через те, що користувачеві дозволена будь-які втручання - бібліотека захищає від неправильної інтеграції цих компонентів, надаючи спеціальні класи Require. Бонусом є те, що потенційному користувачеві не доведеться працювати з ними на пряму, за всім потурбується бібліотека.

Головною функцією - є відкритість інтерфейсу нейронної мережі, оскільки ми можемо створити її динамічно та використовувати її топологію зі своїми алгоритмами навчання. Але щоб не порушити роботу нейронної мережі, використовуються спеціальні Locker типи. Вони є обгортками для інших типів, і можуть обмежувати доступ для змін. На найнижчому рівні Locker типи дозволяють використовувати об'єкт для читання та запису, але забороняють будь-яке перевиділення (reallocation) пам'яті для цього об'єкта. Наприклад контейнер із 8-ми елементів, можна використовувати як звичайний контейнер, але обмежені будь-які операції видалення, вставки, зміни розміру цього контейнера тощо. З'явилася така потреба в Locker типах тільки тому, що будь-яка неправильна зміна топології нейронної мережі, може призвести до поганих наслідків. На першому місці ставилася продуктивність, тому зайві перевірки могли сильно уповільнити і без того доволі ресурсозатратний процес навчання.

Процес налаштування відбувається за допомогою спеціального ядра типів

(TypeSet), що передається типу нейронної мережі, що у свою чергу перевіряє валідність цих типів. Чому не можна перевіряти ці типи одразу в ядрі? Відповідь – можна, але тепер для кожного нового ядра потрібно постійно прописувати валідацію, замість того, щоб надати це нейронній мережі, яка сама знає, які типи її потрібні.

*Другорядні але не менш важливі компоненти.*

Щоб нейронна мережа швидше навчалася - важливо правильно підібрати не тільки потрібну топологію під розв'язуване завдання, але і алгоритм оптимізації. Ця важлива складова сильно впливає швидкість навчання моделей. Адже навчальні дані, що надаються, мають шум і невпорядкованість, а внутрішня топологія нейронної мережі погані початкові ваги та зміщення. Так само важливо підібрати відповідні функції активації - вони надають нейронній мережі рельєфність (якщо дивитись на неї як на складну багатовимірну математичну функцію - якою вона насправді).

Крім функцій активації, необхідно використовувати функцію нормалізації. Зазвичай вона розташовується в останньому шарі нейронної мережі, але ніхто не забороняє використовувати її в середині або навіть на початку. Але варто пам'ятати - чим складніше функція та її похідна, тим довше триває процес навчання. Але як нейронна мережа знає, як навчатися? На допомогу приходять найвідоміший метод навчання – градієнтний спуск. Оптимізатори про які згадувалося раніше – стабілізують градієнтний спуск і роблять його більш передбачуваним. Для того, щоб модель навчалася, найважливіше мати предикат, на основі якого буде проходити навчання. У випадку з класичними нейронними мережами – це функція помилки. Мінімізація помилки на основі градієнтного спуску і навчання.

Топологія нейронної мережі має великий вплив на швидкість збіжності алгоритму, оскільки погано проініціалізована топологія ймовірніше за все уповільнить процес навчання. Тому за ініціалізацію топології відповідають спеціальні генератори, якщо необхідно обмежити або розширити розподіл коефіцієнтів. У бібліотеці два типи генераторів: перші для генерації чисел з

плаваючою точкою – використовується для ініціалізації топології, другий для генерації цілих чисел – зазвичай застосовується у парі зі стохастичним градієнтним спуском.

Не варто забувати, що нейронні мережі представляються у вигляді шарів, а кожен шар має свої внутрішні компоненти. Часто використовуються: вектори і матриці, але також присутній тензор - це двомірна матриця, або тривимірний вектор. Зазначимо, що дані поняття взаємозамінні як словесно а й суто технічно. Будь-який вектор можна як тензор, а будь-яку матрицю як вектор. Але і на цьому все не закінчується - будь-який вектор/матрицю/тензор можна представити у вигляді Range (діапазону). Такий тип служить для впровадження common (загальних) типів їх зберігання у контейнерах і надання функцій загальної сигнатури. Наведену архітектуру винесено в окрему бібліотеку і використовується композитивно. В ній є не тільки тензорні класи, але й різні алгоритми, написані для навчання нейронних мереж з підкріпленням. Все, що перераховано в цьому розділі, присутнє в бібліотеці повною мірою і може бути налаштовано користувачем.

Компоненти для навчання, серіалізації, тестування нейронної мережі: Модульна архітектура та дотримання принципам SOLID, дає чітке розуміння, що жоден клас не робить більше, ніж від нього просять. Нейронні мережі нічого не вміють, крім як отримати дані та повернути результат. Тому як не складно здогадатися для їхнього навчання використовуються спеціальні класи Training (вчителі). Вони приймають тип нейронної мережі, і на основі типу знають як навчати нейронну мережу, а також надають стандартний набір алгоритми навчання. Кожен алгоритм отримує оптимізатор, тому в процесі навчання, їх можна змінювати, а також комбінувати з іншими алгоритмами навчання. Training також дозволяє встановлювати функцію похибки, що робить навчання ще більш гнучкішим.

Серіалізація дозволить користуватися вже навченими моделями, не витрачаючи час на повторне навчання. Ніхто не забороняє розбити цикл навчання на кілька ітерацій, користувач завжди може продовжити навчання коли захоче.

Оскільки серіалізація є бінарною, що підриває підтримку на різних платформах, при серіалізації перші кілька байт використовуються для кодування мета даних про розмір типів. Це дає можливість серіалізувати дані на одній архітектурі та десеріалізувати на іншій. Вся магія полягає в обгорткових класах, вони отримують мета дані і відповідно до розміру, обробляють інформацію без потенційних втрат.

Процес навчання може тривати скільки завгодно, але нейронна мережа не знає, чи добре вона відповідає поставленим завданням. Необхідно перевірити якість її навчання. Клас Training може також надавати інформацію щодо стану навченої моделі, але з виконанням цих завдань краще впораються класи Accuracy та Checker. Accuracy має кілька режимів перевірки - ці свідчення дають уявлення про те, наскільки локально чи глобально прогнозує відповідь нейронна мережа. Checker надає інформацію про відхилення бажаного результату від реального. Загалом досить важко змусити велику нейронну мережу не помилятися зовсім. Але якщо вона ніколи не помиляється отже вона перенавчена. Це також погано. Тому за цим коефіцієнтом можна дати об'єктивну оцінку внутрішнього стану нейронної мережі.

#### *Лінійна та поліноміальна регресії.*

Крім нейронної мережі можна також задовольнятися регресійним аналізом даних. У бібліотеці присутні два типи регресії: лінійна для багатовимірного випадку та поліноміальна для одновимірного. Чому немає поліноміальної регресії для багатовимірного випадку? Тому що функція такої складності буде рівною нейронній мережі. Регресія підпорядковується тим же правилам, як і нейронні мережі. За винятком того, що процес навчання відбувається миттєво, а заповнювати випадковими значеннями топологію не потрібно.

#### *Висновки.*

Даною бібліотекою була навчена повнозв'язна нейронна мережа з точністю 0.998% на тренувальному та 0.981% на тестовому пакетах, що вміє розпізнавати рукописні цифри на картинці 28x28 пікселів з одним каналом квітів, тобто тільки чорно-біла картинка. Хоч і повнозв'язна нейронна мережа не має такої великої



включаючи клас `Shape`, а також `Linear` (ціль класу `Shape` – узагальнити інтерфейс представлення розмірності усіх тензорних типів).

З точки зору продуктивності найменшою одиницею моделі (при побудові нейронної мережі) є тензор, вони групуються в шари, які вже свою чергу утворюють нейронну мережу. Тензори поділяються на два типи: `own` і `view`. `Own` – відповідальний за виділений ресурс пам'яті, зазвичай це найпоширеніший тип використання. `View` – не володіє ресурсом, і використовується лише для запису та зчитування даних з ресурсу.

Окрім типів тензори також поділяються на види – `Vector`, `Matrix`, `Tensor`. Кожен вид може конвертуватися в інший, тому матрицю можна представити у вигляді вектору, і навпаки. Всі види тензорів відносяться до одного із наведених вище типів.

Тензор, як незалежна одиниця - не може взаємодіяти з іншими видами тензорів і виконувати складні операції, тому для вирішення цих проблем використовується клас `Linear`. Він дозволяє виконувати різні види обчислень специфічного характеру, наприклад: множення вектора на матрицю, тензорний добуток, транспонування, знаходження оберненої матриці та інші.

Кожен шар нейронної мережі складається з деякої кількості тензорів - в якості внутрішніх параметрів та змінних для кешування. Шари мають спільний інтерфейс взаємодії, та поділяються на два типи `raw` і `train`. `Raw` – використовується лише для готових моделей. `Train` – є розширеною версією `raw` типу, оскільки підтримує зворотне розповсюдження помилки. На відміну від цього, в кожному типі може бути функція активації, яка підтримує пряме, а також зворотне розповсюдження у випадку з `train` шаром.

Пакет `train` містить клас `Training` та усі реалізації інтерфейса `IOptimizer`. Більша частина оптимізаторів включає в себе використання `Range` класів. Це пояснюється необхідністю конвертувати матриці, вектори та тензори в один уніфікований тип даних, який може приймати функція інтерфейсу (а також додаткова можливість збереження усіх цих об'єктів в контейнері). Проблема такої необхідності полягає в обмеженнях мови програмування `C++`. Оскільки в

стандарті мови не має такого терміну, як інтерфейс, зазвичай розробники використовують стандартні засоби його реалізації – через чисті віртуальні функції (pure virtual function). Віртуальні функції (в загальному), дозволяють реалізовувати принцип ООП – поліморфізм. Більшість розробників компіляторів, реалізують віртуальні функції через віртуальні таблиці (vtable). Virtual table – це масив вказівників на функції, а оскільки вказівник на функцію, це не прямий виклик, функція не може бути шаблоном. Припустимо, що ця таблиця може зберігати вказівники на вже проінстанційовані функції – виникає проблема, яку саме реалізацію функції і для якого класу використовувати.

Пакет `utility` містить клас для перетворення тензора в діапазон (це необхідно для узагальнення інтерфейсу роботи з тензорами). Також цей пакет містить генератори чисел: `RandomInteger`, `RandomFloating`.

Пакет `layer` включає в себе усі реалізації інтерфейсів `ILayer` та `ITrainLayer`, який розширює інтерфейс `ILayer`, це означає, що тренований шар може бути перетворений в звичайний шар, але не навпаки. Кожен шар може агрегатно зберігатися в нейронній мережі.

В пакеті `functional` наведено лише частину спроектованого функціоналу, тому деякі елементи не були представлені в даній діаграмі (функції активації: `ELU`, `LReLU`, `SELU`, `GELU`, `SoftSign`, `SoftPlus`, `Swish`, `ModRelu`, `ModTanh` та функції похибки: `MAE` - `mean_absolute_error`, `MSLE` - `mean_squared_log_error`, `NLL` - `negative_log_likelihood`, `LC` - `logcosh`). Кожен клас в пакетах `loss` та `activation`, реалізує необхідний функціонал, який згодом використовується в шарах та оптимізаторах.

Крім класу `TrixyNet`, глобальному пакеті належать компоненти `Serializer`, а також `Checker`, який включає в собі `Accuracy` та `Guide`. Останній елемент визначає рівень похибки нейронної мережі (`loss`).

Усі наведені класи є шаблонами (`template`), тому для спрощення загального семантичного навантаження в наведеній діаграмі їх було опущено. Під час інстанціювання класу `TrixyNet`, вимагає від користувача забезпечити його типом `TypeSet`, який містить набір усіх необхідних типів для роботи - це потребує

передавати TypeSet або тип нейронної мережі іншим класам, які її використовують та мають до неї агрегатне чи композиційне відношення.

### 3.3 Проєктування GUI

Користувач повинен мати можливість створювати, редагувати та видаляти шари створеної топології нейронної мережі. Під час створення можна задавати лише специфікований список параметрів, наприклад: розмір входу, виходу, функція активація тощо. Оскільки кожен шар нейронної мережі не пов'язаний з будь-яким іншим, процес видалення та редагування не повинен супроводжуватися незручностями з боку користувача.

Тренування мережі може бути можливим лише тоді, коли будуть виконані наступні умови, а саме: підібраний та відформатований набір даних, нейронна мережа створена та проініціалізована. Далі необхідно встановити функцію похибки, обрати один із доступних методів навчання і передати в нього оптимізатор щ власними налаштованими параметрами. Під час тренування користувач може відстежувати процес та втручатися в нього, коли це необхідно. Процес відстеження відбувається в числовому форматі, де представлені значення точності (accuracy) та похибки (loss).

Кожну створену модель можна зберігати та завантажувати. На даному рівні проєктування, програма не підтримує уніфіковані формати зберігання нейронних мереж, такі як HDF5, ONNX, XML та JSON [23]. Програма повинна мати два режими: режим конструювання, усі вимоги до якого наведені вище, а також режим тестування. В ньому користувач повинен завантажити існуючу модель, задати формат вхідних даних. Після чого подати на вхід дані і отримати деякий результат від нейронної мережі.

Для досвідчених користувачів а також розробників, пропонується доволі простий механізм інтеграції усього функціоналу завдяки незалежності рівнів ядра (model) та представлення (view).



### 3.2 Реалізація та тестування

На рисунку 3.3 зображено діаграму засобів використання (use case diagram). Взаємодія з системою ділиться на 4 прецеденти (управління моделями, розробка, навчання та тестування), вони в свою чергу включають інші прецеденти.

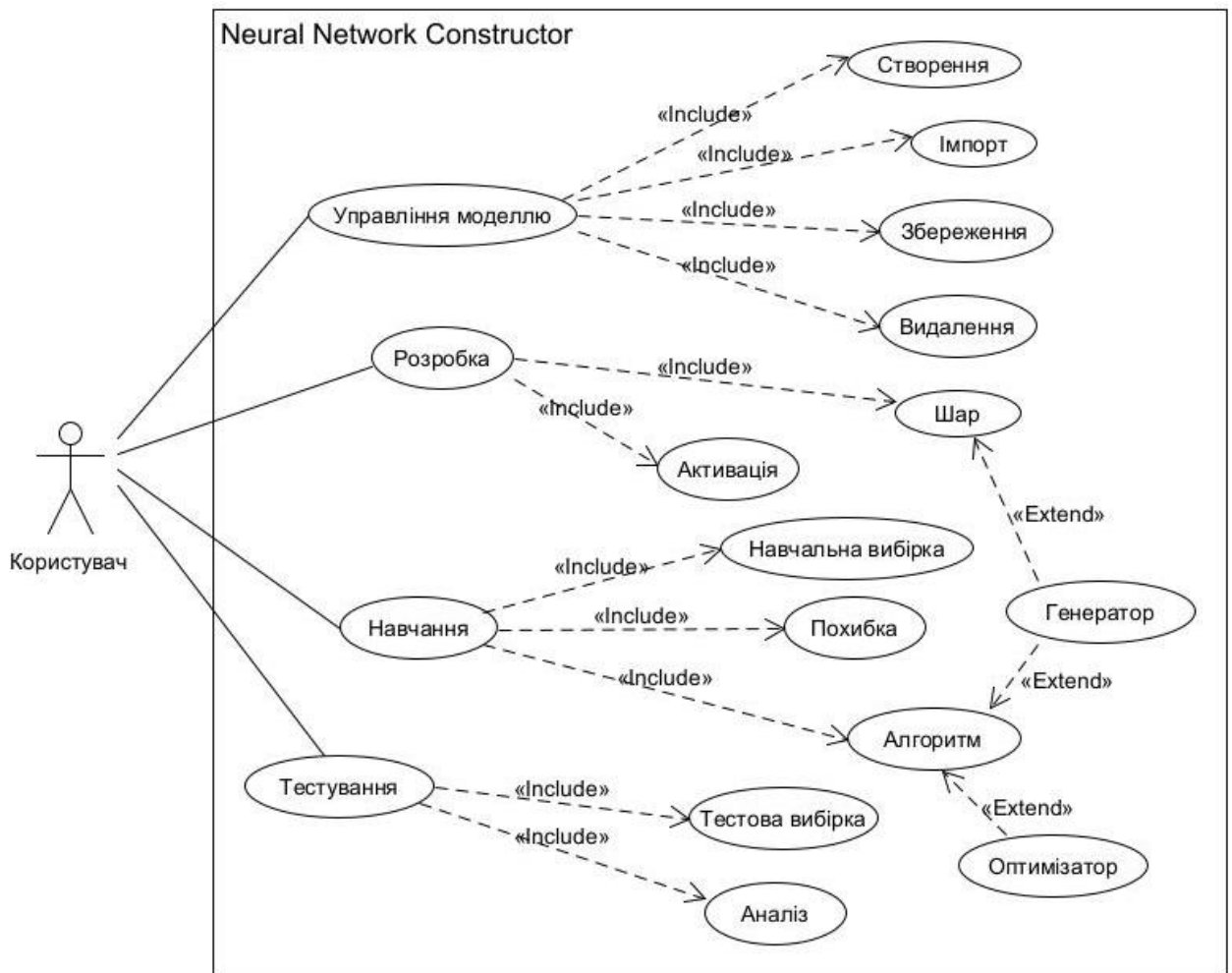


Рисунок 3.3 – Засоби використання системи

При вході в програму, користувачеві представлено форму зі списком усіх створених та завантажених нейронних мереж, а також три кнопки: exit (вихід), create (створення) та load (експорт). Кнопка create показує користувачеві форму в якій можна ввести ім'я нової нейронної мережі та шлях, куди потрібно її зберегти, кнопка завантаження відкриє explorer (провідник системи) для вибору необхідного файлу. Обраний файл перевіряється на валідність та у випадку успіху

додається до списку. Завантажена чи створена нейронна мережа або подвійний клік по елементу зі списку закриває представлену форму.

Верхня панель загального вікна містить у собі три відділені вкладки різного типу, а саме development (розробка), train (тренування) та testing (тестування). Перемикання між ними вмикає відповідні режими і відкриває нижню панель (під даною), де знаходяться необхідні опції.

В режимі розробника (development), користувачеві доступні вкладки: prototype (готові прототипи), layer (доступні шари), activation (доступні функції активації), generator (генератор дробових чисел). Кожен елемент створюється незалежно, і для компонування потребує утворення зв'язку. Наприклад можна спочатку створити функцію активації, потім повнозв'язний шар і підключити функцію активації (помістити) до створеного шару. Шари як і функції активації необхідно між собою з'єднувати, під час з'єднання, перевіряється відповідність виходів та входів. Оскільки не дозволяється підключати шар із виходом розміру 10-ти нейронів до шару із входом розміром 15-ти нейронів. У разі помилкового з'єднання необхідно показувати користувачеві відповідне повідомлення. При створенні шару задаються його розміри, але редагувати параметри шару можна для існуючих.

В режимі тренування (train), користувачу доступні такі вкладки як: prototype (готові шаблони для тренування), data (тренувальний набір даних), algorithm (алгоритми навчання), loss (функції похибки), optimizer (оптимізатори), generator (генератор інтегральних чисел). Для того щоб навчити нейронну мережу, необхідно створити функцію похибки і обрати алгоритм (або алгоритми) навчання. До функції помилок підключається алгоритм, в який можна помістити оптимізатор, а також генератор, якщо це необхідно. Алгоритми навчання можуть працювати в парі з іншим, достатньо з'єднати їх між собою у бажаному порядку. При створенні алгоритму задаються відповідні параметри, їх можна змінювати в процесі навчання. Навчання нейронної мережі буде неможливим у випадку відсутності тренувального набору даних. Порушення даної вимоги, буде супроводжуватися відповідним повідомленням.

Режим тестування (testing) надає можливість обирати вхідні дані із завантаженого набору даних. У даному режимі користувачеві доступно три вкладки - data (тестовий набір даних), using (використання нейронної мережі), analysis (аналіз дозволить отримати інформацію щодо точності та похибки, відповідно до обраного набору даних).

Для запуску навчання або тестування необхідно натиснути відповідну трикутну кнопку run (запуск). При необхідності можна призупинити та продовжити процес. У разі невалідності даних, користувач отримає відповідне повідомлення з підказкою.

Збереження нейронної мережі відбувається автоматично при виході із програми, натисканні спеціальної комбінації клавіш Ctrl+S або при виборі опції File -> Save/Load.

Також програма має додаткові опції, такі як settings, help, version. Опція Settings відкриває вікно у якому можна змінювати мову, тему, обмежене використання ресурсів та інше. Опція Help дозволяє вибрати tutorial/support. Version відкриває форму з інформацією про програму та контактними даними розробників.

В загальному інтерфейс взаємодії з усіма компонентами для конструювання та навчання нейронної мережі відбувається на полотнах, для кожного режиму використовуються незалежні один від одного полотна. На полотні можна перетягувати елементи та з'єднувати їх між собою - це нагадує роботу з blueprints ігрового двигуна Unreal Engine, в даній статті [26] гарно описані ключові положення та принципи роботи з ними. Деяких елементи на полотнах мають внутрішні параметри, які можна редагувати. Процес редагування супроводжується інтерактивним меню, яке викликається подвійним натисканням лівої кнопки миші. В перспективі планується змінювати внутрішні параметри на самому елементі, без необхідності відкривати додаткові форми.

На рисунку 3.4 зображено діаграму діяльності взаємодії з графічним інтерфейсом. Ключові операції є паралельними, що надає можливість взаємодіяти з системою зручним для користувача способом.



На рисунку 3.5 зображено приклад створення нейронної мережі з чотирьма повнозв'язними шарами. Перші три шари мають функцію активації ReLU, в останньому шарі використовується функція активації SoftMax, оскільки нейронна мережа класифікує три класи.

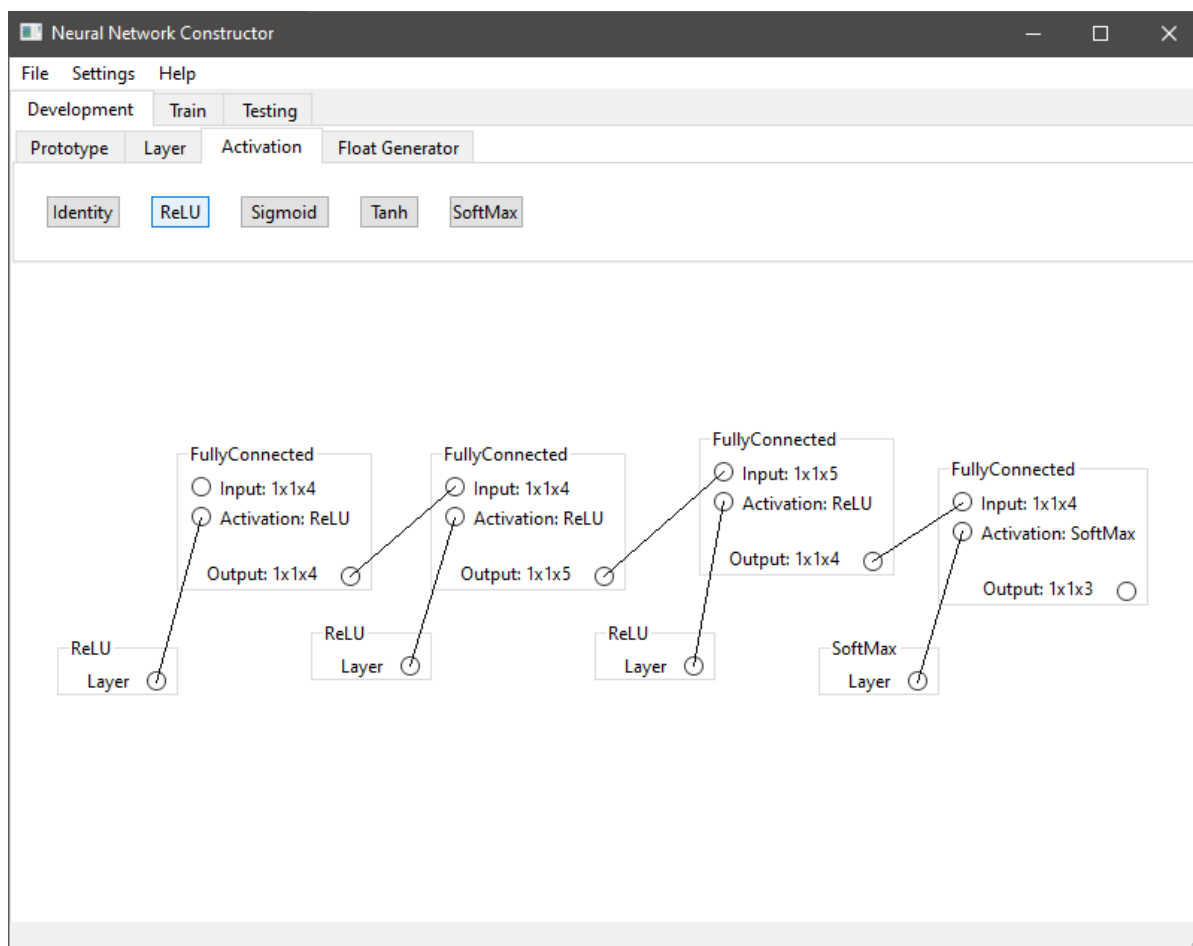


Рисунок 3.5 – Розробка нейронної мережі

Варто зазначити, що уразі створення декількох моделей на одному полотні, користувач отримає повідомлення про помилку, оскільки всі створені шари повинні бути з'єднані між собою. Виключенням є шари входу та виходу. Оскільки вони з'єднуються з одним із сусідніх шарів. У перспективі планується додати спеціальний одиничний компонент, який буде з'єднуватися з вхідним шаром, що дозволить створювати декілька нейронних мереж на одному полотні. Поточна активна нейронна мережа буде визначатися за допомогою зв'язку з цим компонентом.

На рисунку 3.6 зображено приклад створення тренера для навчання. Варто зазначити, що формат вхідних та вихідних даних налаштовується при виборі. Кожне значення для зразка записується в файлі та відділяється крапкою з комою, а кожне значення для зразка просто комою, дробові числа позначаються через крапку. У перспективі планується додати можливість автоматичної нормалізації.

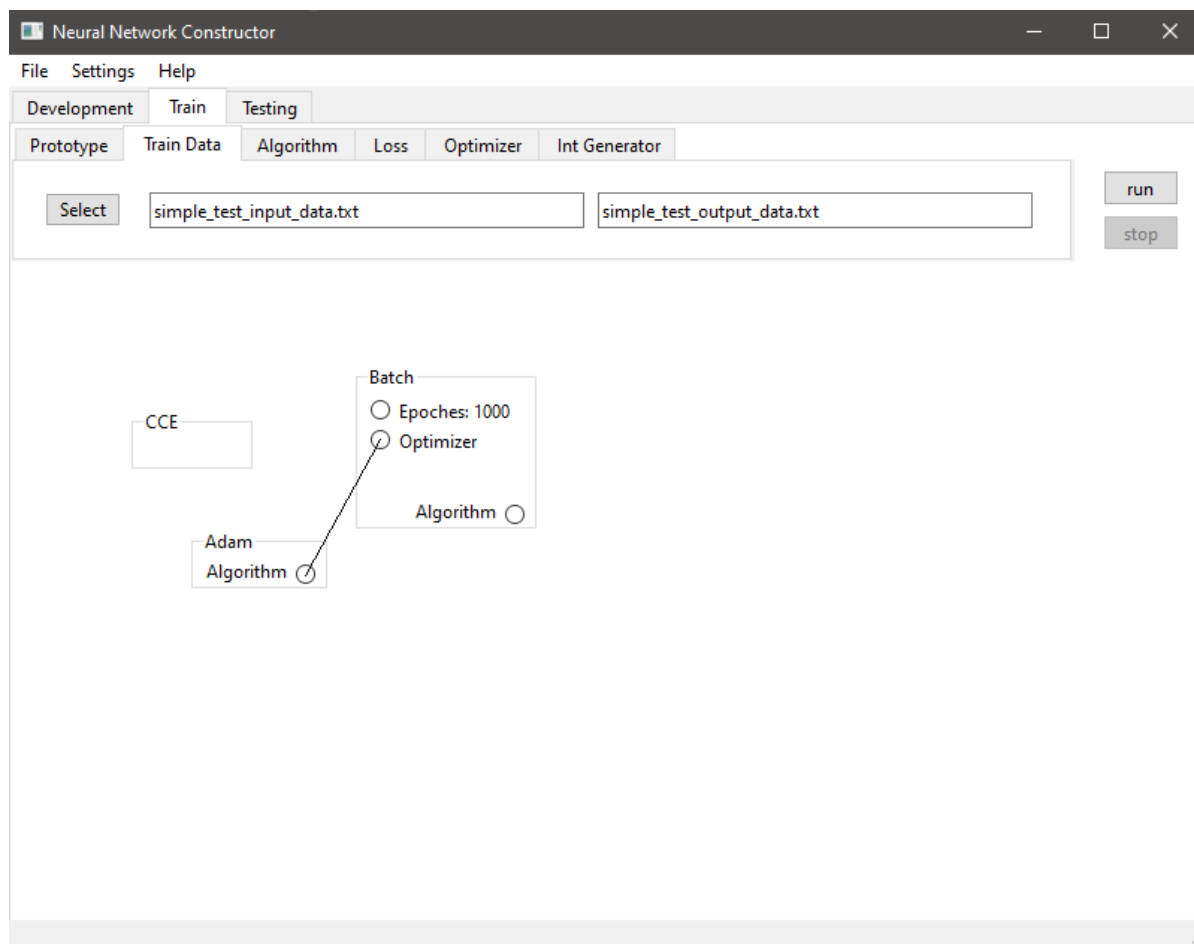


Рисунок 3.6 – Створення тренера для навчання

Для повноти створення тренера, необхідно створити функцію похибки, в даному випадку було створено функцію SSE. Оскільки використовуватися може лише одна функція похибки, то немає необхідності її підключати до інших компонентів.

Валідація навчальних та тестових даних відбувається під час натискання кнопки run, оскільки нейронна мережа може змінюватися вже після вибору вхідних даних.

На рисунку 3.7 зображено результат навчання нейронної мережі.

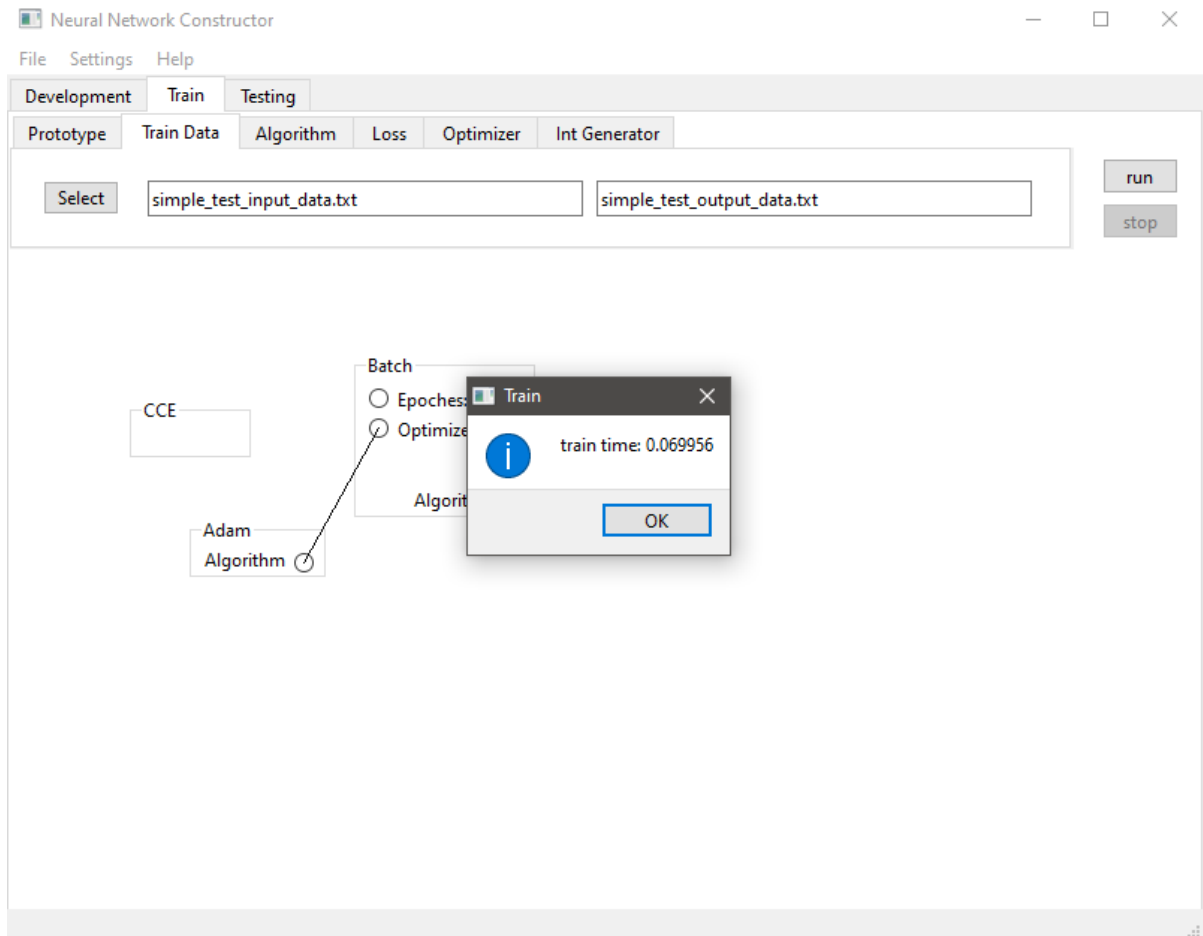


Рисунок 3.7 – Результат навчання нейронної мережі

На рисунку 3.8 представлено результат тестування навченої моделі. Для тестування було використано дані для навчання.

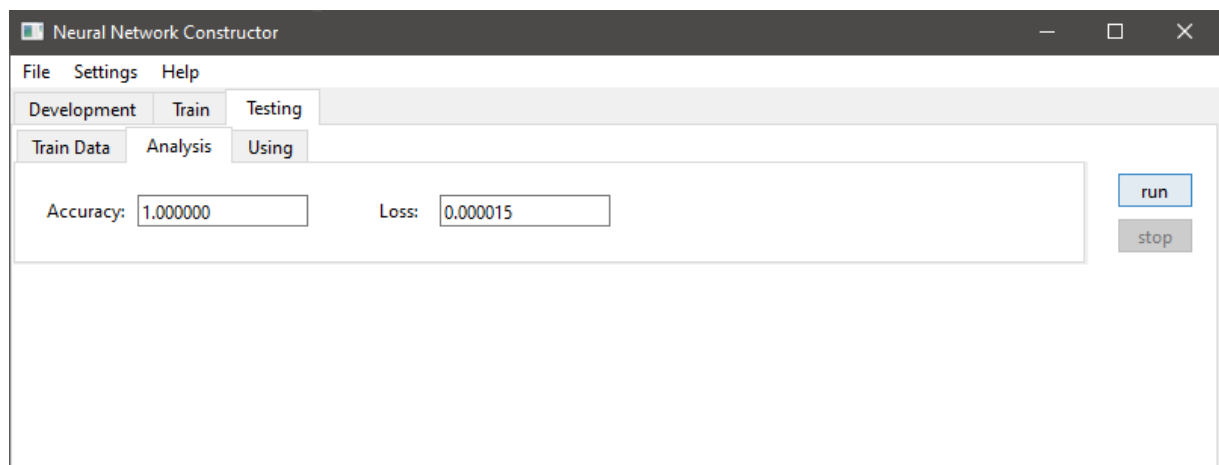


Рисунок 3.8 – Результат тестування нейронної мережі

## ВИСНОВКИ

Проаналізовано існуючі додатки, оцінено їх функціонал, користувальницький інтерфейс та загальну зручність. Виявлено переваги та недоліки: обширний функціонал, гнучкість та швидкість, а також обмежений функціонал, неінтуїтивний інтерфейс та обмежена доступність. Вище перелічені переваги та недоліки враховано при розробці власного додатку.

Проведено аналіз технічних засобів для розробки додатку. У результаті аналізу враховано критерії простоти, зручності та доступності.

Проведено аналіз використання аналогічних додатків. На основі цього спроектовано та розроблено додаток Neural Network Constructor, що відповідає потребам та вимогам користувачів, а також враховано критерії зручності та простоти у використанні.

Проведено тестування, під час якого виявлено деякі помилки, більшу частину яких було усунуто.

Додаток є актуальним для розробників ШІ, навчальних закладів (з метою популяризації нейронних мереж), а також для рядових користувачів. Даний конструктор створений з метою поліпшення часових витрат на створення нейронних мереж на основі вхідних та вихідних даних.

У подальшому розвитку проєкта планується:

- Розширена підтримка форматів зберігання моделі;
- Покращення візуалізації та моніторингу моделі;
- Інтеграція нових видів шарів та алгоритмів навчання;
- Розширення мовної локалізації;
- Підвищення продуктивності програми;
- Підтримка мануального розширення компонентів додатка.



## ПЕРЕЛІК ПОСИЛАНЬ

1. Difference between brain and machine [Електронний ресурс] – Режим доступу до ресурсу: <https://sysblok.ru/knowhow/mozg-protiv-kompjutera/>
2. Overview of neuron structure and function - Khan Academy [Електронний ресурс] – Режим доступу до ресурсу: <https://www.khanacademy.org/science/biology/human-biology/neuron-nervous-system/a/overview-of-neuron-structure-and-function>
3. Professor's perceptron paved the way for AI – 60 years too soon [Електронний ресурс] – Режим доступу до ресурсу: <https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon>
4. Supervised-vs-unsupervised-learning [Електронний ресурс] – Режим доступу до ресурсу: <https://www.linkedin.com/pulse/supervised-vs-unsupervised-learning-whats-difference-smriti-saini>
5. Agent and environment [Електронний ресурс] – Режим доступу до ресурсу: <https://coderlessons.com/tutorials/akademicheskii/izuchit-iskusstvennyi-intellekt/ai-agenty-i-sreda>
6. Topology of neural network [Електронний ресурс] – Режим доступу до ресурсу: <https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/>
7. Fully connected neural network [Електронний ресурс] – Режим доступу до ресурсу: <https://deeplearningmath.org/general-fully-connected-neural-networks.html>
8. Convolution neural network [Електронний ресурс] – Режим доступу до ресурсу: <https://developersbreach.com/convolution-neural-network-deep-learning/>
9. Geron A. Hands-On Machine Learning with Scikit-Learn and TensorFlow / Aurelien Geron. – Beijing, Boston, Farnham, Sebastopol, Tokyo: O'REILLY, 2017
10. Goodfellow I. Deep Learning / I. Goodfellow, Y. Bengio, A. Courville., 2016. – 800 с.

11. What are Recurrent Neural Networks? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.ibm.com/topics/recurrent-neural-networks>
12. Introduction to Linear Regression and Polynomial Regression [Электронный ресурс] – Режим доступа до ресурсу: <https://jashrathod.github.io/2021-06-03-diving-deep-into-linear-regression-and-polynomial-regression/>
13. GAN Deep Learning: A Practical Guide [Электронный ресурс] – Режим доступа до ресурсу: <https://datagen.tech/guides/computer-vision/gan-deep-learning/>
14. Artificial Neural Network - Basic Concepts [Электронный ресурс] – Режим доступа до ресурсу: [https://www.tutorialspoint.com/artificial\\_neural\\_network/artificial\\_neural\\_network\\_basic\\_concepts.htm](https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_basic_concepts.htm)
15. Deep Learning Toolbox [Электронный ресурс] – Режим доступа до ресурсу: <https://www.mathworks.com/products/deep-learning.html>
16. Java Neural Network Framework Neuroph [Электронный ресурс] – Режим доступа до ресурсу: <https://neuroph.sourceforge.net/>
17. Standard C++ [Электронный ресурс] – Режим доступа до ресурсу: <https://isocpp.org>
18. Easy Neural Network Writer [Электронный ресурс] – Режим доступа до ресурсу: <http://www.easynn.com>
19. Qt framework [Электронный ресурс] – Режим доступа до ресурсу: <https://www.qt.io/product/framework>
20. SerializationFixture [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/Sigma-Ryden/SerializationFixture.git>
21. Qt designer - manual [Электронный ресурс] – Режим доступа до ресурсу: <https://doc.qt.io/qt-6/qtdesigner-manual.html>
22. TIOBE Index [Электронный ресурс] – Режим доступа до ресурсу: <https://www.tiobe.com/tiobe-index>
23. File formats for machine learning [Электронный ресурс] – Режим доступа до ресурсу: <https://www.hopsworks.ai/post/guide-to-file-formats-for-machine-learning>
24. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship / Robert Cecil Martin., 2008. – 464 с.

25. Artificial Neural Network - Hopfield Networks [Электронный ресурс] – Режим доступа до ресурсу: [https://www.tutorialspoint.com/artificial\\_neural\\_network/artificial\\_neural\\_network\\_hopfield.htm](https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_hopfield.htm)
26. Blueprint Fundamentals [Электронный ресурс] – Режим доступа до ресурсу: [https://michaeljcole.github.io/wiki.unrealengine.com/Blueprint\\_Fundamentals](https://michaeljcole.github.io/wiki.unrealengine.com/Blueprint_Fundamentals)
27. How Activation Functions Work in Deep Learning [Электронный ресурс] – Режим доступа до ресурсу: <https://www.kdnuggets.com/2022/06/activation-functions-work-deep-learning.html>
28. Iterator cppreference [Электронный ресурс] – Режим доступа до ресурсу: <https://en.cppreference.com/w/cpp/header/iterator>
29. Adaptors cppreference [Электронный ресурс] – Режим доступа до ресурсу: [https://en.cppreference.com/w/cpp/container#Container\\_adaptors](https://en.cppreference.com/w/cpp/container#Container_adaptors)
30. Design patterns [Электронный ресурс] – Режим доступа до ресурсу: <https://refactoring.guru/design-patterns/catalog>

## ДОДАТОК А



ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ  
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ  
ТЕХНОЛОГІЙ  
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



### Розробка конструктора нейронних мереж із контрольованим навчанням мовою C++

Виконав студент 4 курсу  
групи ПД-41

Гапєй Максим Юрійович  
Керівник роботи

Доцент кафедри ІІІ Фесенко Максим Анатолійович

Київ – 2023

### МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ




- **Мета роботи** – спрощення процесу конструювання нейронних мереж із контрольованим навчанням за допомогою комп'ютерного додатку, розробленого мовою C++.
- **Об'єкт дослідження** – процес створення нейронних мереж із контрольованим навчанням на базі Qt framework.
- **Предмет дослідження** – комп'ютерний додаток для створення нейронних мереж із контрольованим навчанням.

# ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Аналіз обраної предметної області;
2. Порівняння існуючих аналогів програмного забезпечення;
3. Проектування та реалізація конструктора нейронних мереж із контрольованим;
4. Тестування отриманого додатка.

3

## АНАЛІЗ АНАЛОГІВ

Показник	 <b>Deep Learning Toolbox</b>	 <b>Java Neural Network Framework Neuroph</b>	 <b>Easy Neural Network Writer</b>	<b>Neural Network Constructor</b>
Платформи	MATLAB	Віртуальна машина Java	Windows	Windows, Linux, macOS
Формати зберігання	MATLAB, HDF5, ONNX	XML та JSON	CSV, Excel, MATLAB	Бінарний формат
Архітектури нейронних мереж	Fully Connected NN, Convolutional NN, Recurrent NN, Generative Adversarial Network	Fully Connected NN, Convolutional NN, Multi Layer Perceptron, Self Organizing Map	Fully Connected NN, Convolutional NN, Recurrent NN, Hopfield Network, Deep NN	Fully Connected NN, Convolutional NN, Deep NN
Зручність використання	Комплексний	Простий та інтуїтивний	Послідовний	Мінімалістичний
Візуалізація моделі	Tile формат	Tile формат із нейронними зв'язками	Немає	Blueprint формат
Особливості	Інтерфейс програмування	Архітектура плагіна	Генерація коду C із навчених моделей	Відкритий вихідний код ядра

4

# ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## Функціональні вимоги:

- Формат вхідних даних: 1D, 2D та 3D;
- Імпорт та експорт нейронних мереж;
- Шарів різних типів: Fully Connected, Convolutional, MaxPooling;
- Функції активації, нормалізації та похибки;
- Алгоритми навчання: batch, mini-batch, stochastic;
- Оптимізатори алгоритмів навчання.

## Нефункціональні вимоги:

- Операційна система: Windows / Linux / MacOS;
- Процесор: Core i3 / Ryzen 5;
- Оперативна пам'ять: 2GB;
- Вільне місце на жорсткому диску: 10MB.

5

# ПРОГРАМНІ ТА ТЕХНІЧНІ ЗАСОБИ РЕАЛІЗАЦІЇ



C++ Programming language



Qt Framework

SF

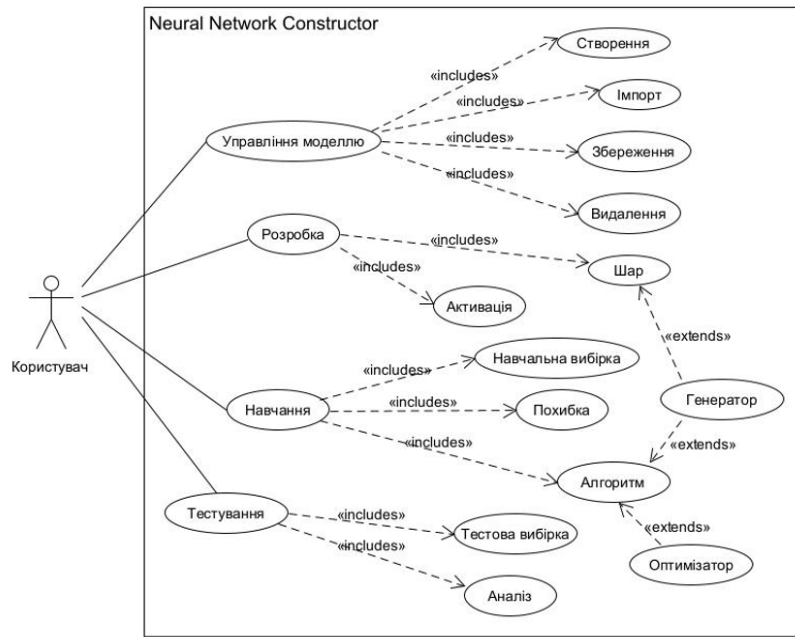
Serialization fixture library



Qt Creator (IDE)

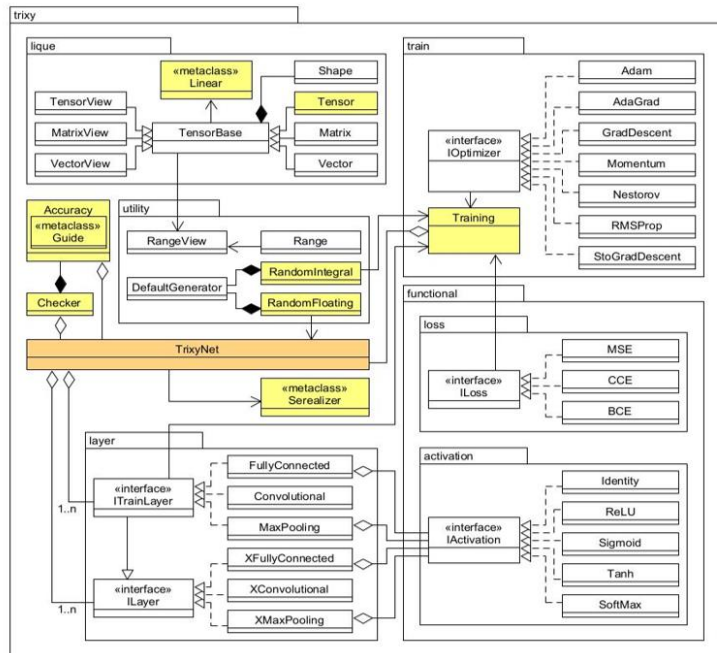
6

# ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



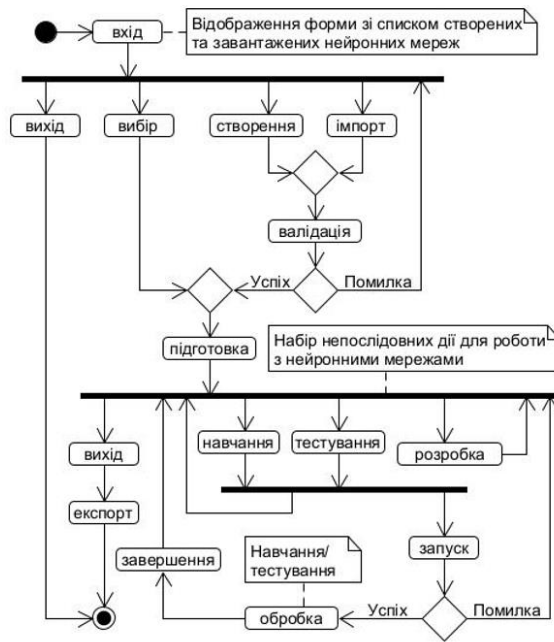
7

# ДІАГРАМА КЛАСІВ



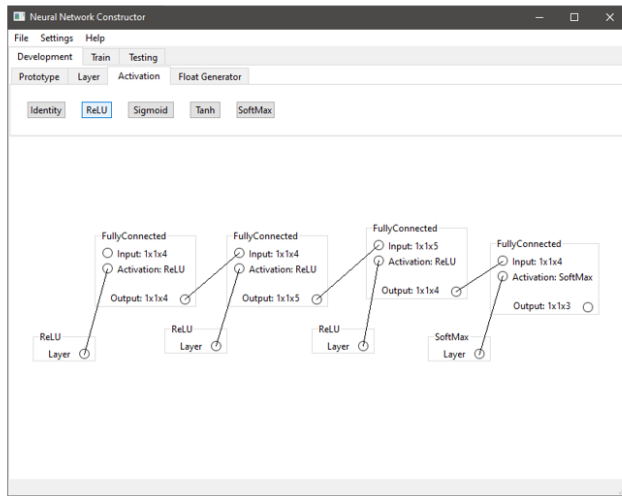
8

# ДІАГРАМА ДІЯЛЬНОСТІ

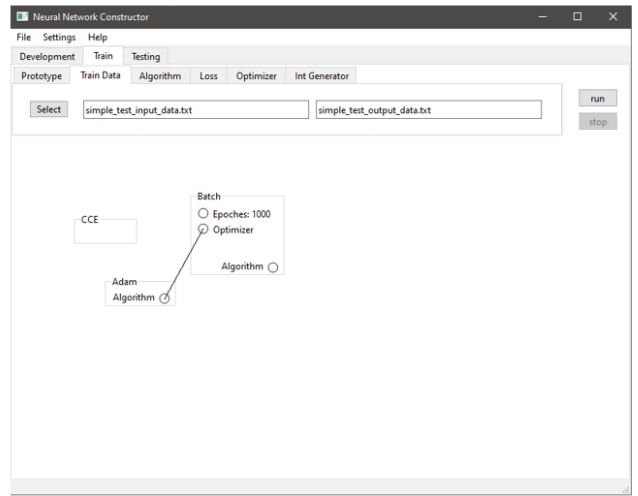


9

# ЕКРАННІ ФОРМИ



Створення нейронної мережі

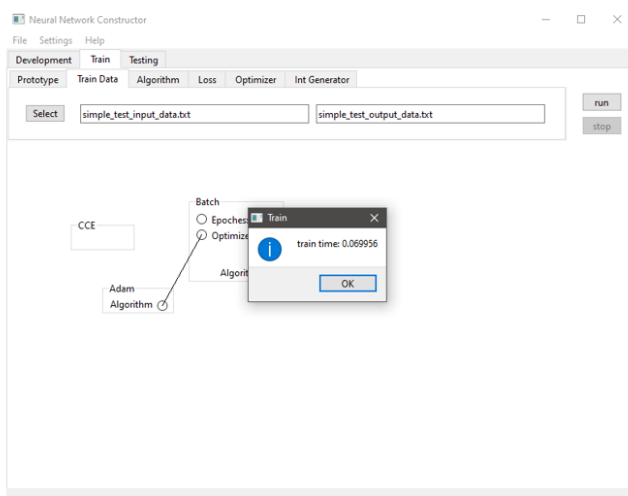


Створення тренера для навчання

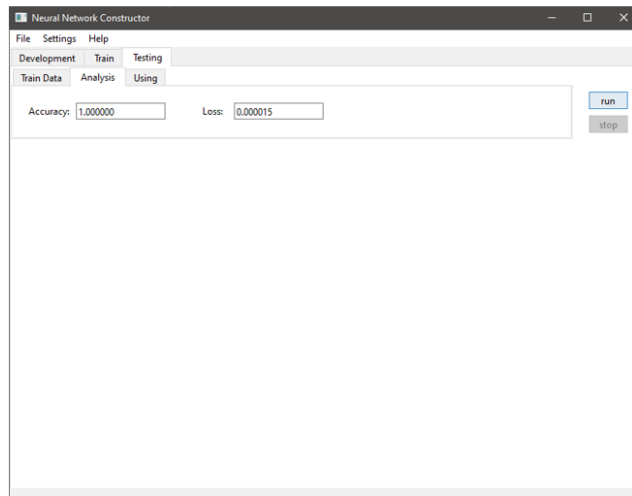
10



## ЕКРАННІ ФОРМИ



Результат навчання



Результати тестування

11

## АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Гапєй М.Ю. Розробка конструктора для нейронних мереж із контрольованим навчанням / Гапєй М.Ю., Фесенко М.А. // Сучасні інтелектуальні інформаційні технології в науці та освіті: Матеріали всеукраїнської науково-практичної конференції. Збірник тез. 16.05.2023, ДУТ, м. Київ – К.: ДУТ, 2023.

Гапєй М.Ю. Покращення медичної діагностики за допомогою інтелектуального аналізу даних / Гапєй М.Ю., Фесенко М.А. // Сучасні інтелектуальні інформаційні технології в науці та освіті: Матеріали всеукраїнської науково-практичної конференції. Збірник тез. 16.05.2023, ДУТ, м. Київ – К.: ДУТ, 2023.

12

## ВИСНОВКИ

1. Проаналізовано існуючі додатки, оцінено їх функціонал, користувацький інтерфейс та загальну зручність. Виявлено переваги та недоліки: обширний функціонал, гнучкість та швидкість, а також обмежений функціонал, неінтуїтивний інтерфейс та обмежена доступність. Вище перелічені переваги та недоліки враховано при розробці власного додатку.
2. Проведено аналіз технічних засобів для розробки додатку. У результаті аналізу враховано критерії простоти, зручності та доступності.
3. Проведено аналіз використання аналогічних додатків. На основі цього спроектовано та розроблено додаток Neural Network Constructor, що відповідає потребам та вимогам користувачів, а також враховано критерії зручності та простоти у використанні.
4. Проведено тестування, під час якого виявлено деякі помилки, більшу частину яких було усунуто.

**ДЯКУЮ ЗА УВАГУ!**

## ДОДАТОК Б

### Інтерфейс основних компонентів програми:

```
namespace trixy
{
    namespace lique
    {
        template <typename T>
        struct Shape
        {
        private:
            static constexpr bool require = std::is_integral<T>::value;

            static_assert(require, "'T' should be an integral type.");

        public:
            using size_type = T;

        public:
            size_type depth;
            size_type height;
            size_type width;

            size_type size;

        public:
            Shape();
            explicit Shape(size_type d, size_type h, size_type w);
            explicit Shape(size_type h, size_type w);
            explicit Shape(size_type w);
        };

        template <typename Precision>
        using TensorBase = Tensor<Precision, TensorType::base, TensorMode::view>;

        template <typename Precision>
        class Tensor<Precision, TensorType::base, TensorMode::view> : public
        TensorType::base
        {
            SERIALIZATION_ACCESS()

            static constexpr bool require = std::is_arithmetic<Precision>::value;

            static_assert(require, "'Precision' should be an arithmetic type.");

        public:
            using size_type          = std::size_t;
            using precision_type     = Precision;
            using value_type         = Precision;
            using shape_type         = Shape<std::size_t>;

            using pointer            = Precision*;
            using const_pointer      = const Precision*;

            using reference          = Precision&;
            using const_reference    = const Precision&;

            using range_view        = utility::Range<Precision>;

        protected:
    
```

```

    pointer data_;
    shape_type shape_;

public:
    Tensor() noexcept;
    ~Tensor();

    explicit Tensor(const shape_type& shape, pointer data = nullptr) noexcept;

    Tensor(size_type d, size_type h, size_type w, pointer data = nullptr)
noexcept;
    Tensor(size_type h, size_type w, pointer data = nullptr) noexcept;
    Tensor(size_type w, pointer data = nullptr) noexcept;

    Tensor(const Tensor& tensor) noexcept;
    Tensor(Tensor&& tensor) noexcept;

    Tensor& operator= (const Tensor& tensor) noexcept;
    Tensor& operator= (Tensor&& tensor) noexcept;

    Tensor& copy(const_pointer src) noexcept;
    Tensor& copy(const Tensor&) noexcept;
    Tensor& copy(std::initializer_list<precision_type>) noexcept;

    template <class Generator,
              trixy::meta::require<trixy::meta::is_callable<Generator>::value> =
0>
    Tensor& fill(Generator generator) noexcept;

    Tensor& fill(precision_type value) noexcept;

    template <class Function>
    Tensor apply(Function func) const;

    template <class Function>
    Tensor& apply(Function func) noexcept;

    template <class Function>
    Tensor& apply(Function func, const_pointer src) noexcept;

    template <class Function>
    Tensor& apply(Function func, const Tensor& rhs) noexcept;

    Tensor add(const Tensor& rhs) const;
    Tensor& add(const Tensor& rhs) noexcept;
    Tensor& add(const Tensor& lhs, const Tensor& rhs) noexcept;

    Tensor sub(const Tensor& rhs) const;
    Tensor& sub(const Tensor& rhs) noexcept;
    Tensor& sub(const Tensor& lhs, const Tensor& rhs) noexcept;

    Tensor mul(const Tensor& rhs) const;
    Tensor& mul(const Tensor& rhs) noexcept;
    Tensor& mul(const Tensor& lhs, const Tensor& rhs) noexcept;

    Tensor join(precision_type value) const;
    Tensor& join(precision_type value) noexcept;
    Tensor& join(precision_type value, const Tensor&) noexcept;

    pointer data() noexcept;
    const_pointer data() const noexcept;

    size_type size() const noexcept;
    const shape_type& shape() const noexcept;

```

```

void swap(Tensor& tensor) noexcept;

operator range_view() const noexcept;

pointer at(size_type i) noexcept;
const_pointer at(size_type i) const noexcept;

reference operator() (size_type i) noexcept;
const_reference operator() (size_type i) const noexcept;
};

template <typename Precision>
using TensorView = Tensor<Precision, TensorType::tensor, TensorMode::view>;

template <typename Precision>
class Tensor<Precision, TensorType::tensor, TensorMode::own>
    : public TensorBase<Precision>, public TensorType::tensor
{
    LIQUE_TENSOR_BASE_BODY()

public:
    Tensor() noexcept = default;
    ~Tensor();

    explicit Tensor(const shape_type& shape, const_pointer data);
    explicit Tensor(const shape_type& shape, precision_type value);
    explicit Tensor(const shape_type& shape);

    Tensor(size_type depth, size_type height, size_type width, const_pointer
data);
    Tensor(size_type depth, size_type height, size_type width, precision_type
value);
    Tensor(size_type depth, size_type height, size_type width);

    Tensor(const_pointer first, const_pointer last);

    Tensor(std::initializer_list<precision_type> list);

    Tensor(const Tensor& tensor);
    Tensor(Tensor&& tensor) noexcept;

    Tensor& operator= (const Tensor& tensor);
    Tensor& operator= (Tensor&& tensor) noexcept;

    pointer at(size_type i, size_type j, size_type k) noexcept;
    const_pointer at(size_type i, size_type j, size_type k) const noexcept;

    reference operator() (size_type i, size_type j, size_type k) noexcept;
    const_reference operator() (size_type i, size_type j, size_type k) const
noexcept;

    Tensor& resize(const shape_type& shape);
    Tensor& resize(size_type depth, size_type height, size_type width);

    Tensor& reshape(size_type depth, size_type height, size_type width) noexcept;
};

template <typename Precision>
class Tensor<Precision, TensorType::tensor, TensorMode::view>
    : public TensorBase<Precision>, public TensorType::tensor
{
    LIQUE_TENSOR_BASE_BODY()

```

```

public:
    Tensor() noexcept = default;
    ~Tensor() = default;

    Tensor(const shape_type& shape, pointer data) noexcept;
    Tensor(size_type depth, size_type height, size_type width, pointer data)
noexcept;
    Tensor(pointer first, pointer last) noexcept;

    Tensor(const Tensor& tensor) noexcept = default;
    Tensor(Tensor&& tensor) noexcept = default;

    Tensor& operator= (const Tensor& tensor) noexcept = default;
    Tensor& operator= (Tensor&& tensor) noexcept = default;

    pointer at(size_type i, size_type j, size_type k) noexcept;
    const_pointer at(size_type i, size_type j, size_type k) const noexcept;

    reference operator() (size_type i, size_type j, size_type k) noexcept;
    const_reference operator() (size_type i, size_type j, size_type k) const
noexcept;

    Tensor& reshape(size_type depth, size_type height, size_type width) noexcept;
};

template <typename Precision>
class Linear
{
private:
    template <class T>
    using as_flat_iterate =
        trixy::meta::require<not meta::is_matrix<T>::value and
meta::is_iterate<T>::value>;

public:
    using size_type = std::size_t;
    using precision_type = Precision;

public:
    template <class Iterable, lique::meta::as_iterate<Iterable> = 0>
    auto first(Iterable& it) const noexcept -> decltype(it.data());

    template <class Iterable, lique::meta::as_iterate<Iterable> = 0>
    auto last(Iterable& it) const noexcept -> decltype(it.data() + it.size());

    template <class Vector1, class Vector2, class Matrix,
        as_flat_iterate<Vector1> = 0,
        as_flat_iterate<Vector2> = 0,
        meta::as_matrix<Matrix> = 0>
    void dot(Vector1& result, const Vector2& row_vector, const Matrix& matrix)
const noexcept;

    template <class Vector1, class Vector2, class Matrix,
        as_flat_iterate<Vector1> = 0,
        as_flat_iterate<Vector2> = 0,
        meta::as_matrix<Matrix> = 0>
    void dot(Vector1& result, const Matrix& matrix, const Vector2& col_vector)
const noexcept;

    template <class Matrix1, class Matrix2, class Matrix3,
        meta::as_matrix<Matrix1> = 0,
        meta::as_matrix<Matrix2> = 0,
        meta::as_matrix<Matrix3> = 0>

```

```

    void dot(Matrix1& result, const Matrix2& lhs, const Matrix3& rhs) const
noexcept;

    template <class Vector1, class Vector2, class Matrix,
              as_flat_iterate<Vector1> = 0,
              as_flat_iterate<Vector2> = 0,
              meta::as_matrix<Matrix> = 0>
    void tensordot(Matrix& result, const Vector1& col_vector, const Vector2&
row_vector) const noexcept;

    template <class Matrix1, class Matrix2,
              meta::as_matrix<Matrix1> = 0,
              meta::as_matrix<Matrix2> = 0>
    void transpose(Matrix1& result, const Matrix2& matrix) const noexcept;

    template <class Matrix1, class Matrix2,
              meta::as_matrix<Matrix1> = 0,
              meta::as_matrix<Matrix2> = 0>
    void inverse(Matrix1& result, Matrix2& matrix) const noexcept;

    template <class Vector1, class Vector2,
              as_flat_iterate<Vector1> = 0,
              as_flat_iterate<Vector2> = 0>
    precision_type dot(const Vector1& lhs, const Vector2& rhs) const noexcept;

    template <class Tensor1, class Tensor2,
              meta::as_iterate<Tensor1> = 0,
              meta::as_iterate<Tensor2> = 0>
    void add(Tensor1& result, const Tensor2& rhs) const noexcept;

    template <class Tensor1, class Tensor2,
              meta::as_iterate<Tensor1> = 0,
              meta::as_iterate<Tensor2> = 0>
    void sub(Tensor1& result, const Tensor2& rhs) const noexcept;

    template <class Tensor1, class Tensor2,
              meta::as_iterate<Tensor1> = 0,
              meta::as_iterate<Tensor2> = 0>
    void mul(Tensor1& result, const Tensor2& rhs) const noexcept;

    template <class Tensor1,
              meta::as_iterate<Tensor1> = 0>
    void join(Tensor1& result, precision_type value) const noexcept;
};

} // namespace lique

namespace utility
{
    struct RandomType
    {
        template <typename T = long long,
                  typename = meta::when<std::is_integral<T>::value>>
        struct Integral { using type = T; };

        template <typename T = double,
                  typename = meta::when<std::is_floating_point<T>::value>>
        struct Floating { using type = T; };
    };
} // namespace utility

namespace meta

```

```

{

template <typename>
struct is_integral_random_type : std::false_type {};
template <typename T>
struct is_integral_random_type<utility::RandomType::Integral<T>> : std::true_type
{};

template <typename>
struct is_floating_random_type : std::false_type {};
template <typename T>
struct is_floating_random_type<utility::RandomType::Floating<T>> : std::true_type
{};

} // namespace meta

namespace utility
{

class DefaultGenerator
{
public:
    using size_type = std::size_t;
    using Generator = int (* )();

public:
    DefaultGenerator() noexcept;
    DefaultGenerator(size_type seed) noexcept;

    void seed(size_type seed) noexcept;

    int operator() () noexcept;

    static std::size_t seed() noexcept;

    static constexpr int min() noexcept;
    static constexpr int max() noexcept;
};

template <typename RandomType, class Generator = DefaultGenerator, typename enable
= void>
class Random;

template <typename RandomType, class Generator>
class Random<RandomType, Generator,
    TRWHEN(meta::is_integral_random_type<RandomType>::value and
    trixy::meta::is_callable<Generator>::value)>
{
public:
    using integral_type = typename RandomType::type;
    using size_type = std::size_t;

private:
    Generator gen;

public:
    Random() noexcept;
    Random(size_type seed) noexcept;

    void seed(size_type seed) noexcept;

    integral_type operator() () noexcept;
    integral_type operator() (integral_type min, integral_type max) noexcept;
};

```



```

template <typename RandomType, class Generator>
class Random<RandomType, Generator,
    TRWHEN(meta::is_floating_random_type<RandomType>::value and
        trixy::meta::is_callable<Generator>::value)>
{
public:
    using floating_type = typename RandomType::type;
    using size_type = std::size_t;

private:
    Generator generator_;

public:
    Random() noexcept;
    Random(size_type seed) noexcept;

    void seed(size_type seed) noexcept;

    floating_type operator() () noexcept;
    floating_type operator() (floating_type min, floating_type max) noexcept;
};

template <typename T = RandomType::Integral<>::type, class Generator =
DefaultGenerator>
using RandomIntegral = Random<RandomType::Integral<T>, Generator>;

template <typename T = RandomType::Floating<>::type, class Generator =
DefaultGenerator>
using RandomFloating = Random<RandomType::Floating<T>, Generator>;

template <typename T, typename RangeType = RangeType::View>
struct Range;

template <typename T>
struct Range<T, RangeType::Unified>
{
public:
    using pointer          = T*;
    using const_pointer    = const T*;

    using value_type       = T;
    using size_type        = std::size_t;
    using difference_type  = std::ptrdiff_t;

private:
    pointer first_;
    size_type size_;

public:
    Range();
    Range(size_type size);
    Range(size_type size, value_type value);
    ~Range();

    void fill(value_type value);

    void resize(size_type size);
    void resize(size_type size, value_type value);

    pointer data() noexcept;
    const_pointer data() const noexcept;
};

```

```

        difference_type size() const noexcept;

        pointer first() noexcept;
        pointer last() noexcept;

        const_pointer first() const noexcept;
        const_pointer last() const noexcept;
};

template <typename T>
struct Range<T, RangeType::View>
{
public:
    using pointer          = T*;
    using const_pointer    = const T*;

    using value_type       = T;
    using size_type        = std::size_t;
    using difference_type  = std::ptrdiff_t;

private:
    pointer first_;
    pointer last_;

public:
    Range(pointer first, pointer last);
    Range();
    Range(pointer first, pointer last, value_type value);

    void fill(value_type value);

    pointer data() noexcept;
    const_pointer data() const noexcept;

    difference_type size() const noexcept;

    pointer first() noexcept;
    pointer last() noexcept;

    const_pointer first() const noexcept;
    const_pointer last() const noexcept;
};

} // namespace utility

namespace functional
{
    struct ActivationType {};

    namespace activation
    {
        template <typename Precision>
        class IActivation : public ActivationType, public sf::Instantiable
        {
            SERIALIZABLE(IActivation)

        public:
            using precision_type = Precision;
            using Range = utility::Range<Precision>;

        public:
            virtual ~IActivation() = default;
    };
}

```

```

    virtual void f(Range result, const Range input) noexcept = 0;
    virtual void df(const Range result, const Range input) noexcept = 0;
};

} // namespace activation

struct LossType {};

namespace loss
{
    template <typename Precision>
    class ILoss : public LossType, public sf::Instantiable
    {
        SERIALIZABLE(ILoss)

    public:
        using precision_type = Precision;
        using Range = utility::Range<Precision>;

    public:
        virtual ~ILoss() = default;

        virtual void f(precision_type& result, const Range y_true, const Range y_pred)
noexcept = 0;
        virtual void df(Range result, const Range y_true, const Range y_pred) noexcept
= 0;
    };

} // namespace loss

} // namespace functional

namespace train
{
    template <class Optimizeriable>
    class IOptimizer<Optimizeriable,
        meta::when<meta::is_trixy_net<Optimizeriable>::value>>
    {
    public:
        using Net = Optimizeriable;

        using precision_type = typename Net::precision_type;
        using size_type = typename Net::size_type;

        using Range = utility::Range<precision_type>; // default view
range
        using RangeUnified = utility::Range<precision_type, RangeType::Unified>;

    private:
        template <typename Ret, typename... Args>
        using Func = Ret (*)(void* const, Args...);

    private:
        Func<void, precision_type> f_set_learning_rate = nullptr;
        Func<precision_type> f_get_learning_rate = nullptr;

        Func<void, Range, Range> f_update = nullptr;

    protected:
        template <class Derived> void initialize() noexcept;

```

```

public:
    virtual ~IOptimizer() = default;

    void learning_rate(precision_type value) noexcept;

    precision_type learning_rate() const noexcept;

    void update(Range param, Range grad) noexcept;

protected:
    template <class Table>
    static RangeUnified& get(Table& table, Range range);
};

} // namespace train

namespace layer
{

template <typename LayerType, class Net, typename LayerMode>
class Layer;

template <class Net>
class ILayer : public sf::Instantiable
{
    SERIALIZABLE(ILayer)

public:
    template <typename T>
    using Container = typename Net::template Container<T>;

    using Vector = typename Net::Vector;
    using Matrix = typename Net::Matrix;
    using Tensor = typename Net::Tensor;

    using XVector = typename Net::XVector;
    using XMatrix = typename Net::XMatrix;
    using XTensor = typename Net::XTensor;

    using size_type = typename Net::size_type;
    using precision_type = typename Net::precision_type;
    using shape_type = typename Net::Tensor::shape_type;

    using Linear = typename Net::Linear;

    using Generator = std::function<precision_type()>; // type erasing
    using IActivation =

functional::activation::IActivation<precision_type>;

    using IOptimizer = train::IOptimizer<Net>;

public:
    virtual ~ILayer() = default;

    virtual void init(Generator& generator) noexcept { /*pass*/ }
    virtual void connect(IActivation* activation) = 0;

    virtual void forward(const Tensor& input) noexcept = 0;
    virtual const Tensor& value() const noexcept = 0;

    virtual const shape_type& isize() const noexcept = 0;
    virtual const shape_type& osize() const noexcept = 0;
};

```

```

template <class Net>
class ITrainLayer : public ILayer<Net>
{
    SERIALIZABLE(ITrainLayer)

    using Base = ILayer<Net>;

public:
    template <typename T>
    using Container = typename Base::template Container<T>;

    using typename Base::Vector;
    using typename Base::Matrix;
    using typename Base::Tensor;

    using typename Base::XVector;
    using typename Base::XMatrix;
    using typename Base::XTensor;

    using typename Base::size_type;
    using typename Base::precision_type;
    using typename Base::shape_type;

    using typename Base::Linear;

    using typename Base::Generator;
    using typename Base::IActivation;

    using typename Base::IOptimizer;

public:
    virtual ~ITrainLayer() = default;

public:
    virtual void backward(const Tensor& input, const Tensor& idelta, bool full =
true) noexcept = 0;
    virtual const Tensor& delta() const noexcept = 0;

    virtual void update(IOptimizer& optimizer, precision_type alpha) noexcept {
/*pass*/ }

    virtual void accumulate() noexcept { /*pass*/ }
    virtual void reset() noexcept { /*pass*/ }
};

} // namespace layer

TRIXY_NET_TEMPLATE()
using UnifiedNed = TrixyNet<TypeSet>;

TRIXY_NET_TEMPLATE()
class TrixyNet<TypeSet>
    : public guard::TrixyNetRequire<TypeSet>::type
{
    SERIALIZATION_ACCESS()

public:
    template <typename T>
    using Container = typename TypeSet::template Container<T>;

    using Vector = typename TypeSet::Vector;
    using Matrix = typename TypeSet::Matrix;
    using Tensor = typename TypeSet::Tensor;

```

```

template <typename T>
using XContainer = memory::ContainerLocker<Container<T>>;

using XVector = memory::VectorLocker<Vector>;
using XMatrix = memory::MatrixLocker<Matrix>;
using XTensor = memory::TensorLocker<Tensor>;

using precision_type = typename TypeSet::precision_type;
using size_type = typename TypeSet::size_type;

using Linear = typename TypeSet::Linear;

using ILayer = layer::ILayer<TrixyNet>;
using ITrainLayer = layer::ITrainLayer<TrixyNet>;

using Topology = Container<ILayer*>;

private:
    Topology inner_;

public:
    Linear linear;

public:
    TrixyNet(size_type reserve_size = 8);
    TrixyNet(const Topology& topology);
    ~TrixyNet();

    TrixyNet& add(ILayer* layer);

    const Topology& inner() const noexcept;
    ILayer& layer(size_type i) noexcept;
    size_type size() const noexcept;

    const Tensor& feedforward(const Tensor& sample) noexcept;
    const Tensor& operator() (const Tensor& sample) noexcept;

    template <class FloatGenerator>
    void init(FloatGenerator generator) noexcept;
};

TRIXY_SERIALIZER_TEMPLATE()
class Serializer
{
public:
    template <class OutStream,
              class SreamWrapper = sf::wrapper::OFileStream<OutStream>>
    static void serialize(OutStream& out, Serializable& serializable);

    template <class InStream,
              class SreamWrapper = sf::wrapper::IFileStream<InStream>>
    static void deserialize(InStream& in, Serializable& serializable);
};

template <class Checkable>
class Checker
{
public:
    using Net = Checkable;

    using precision_type = typename Net::precision_type;
    using size_type = typename Net::size_type;

private:

```



```
    TRIXY_ACCURACY_TEMPLATE()
    long double operator() (const Container<Sample>& idata,
                           const Container<Target>& odata) noexcept;
};

} // namespace trixy
```