МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ПрАТ «ПРИВАТНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра Інформаційних технологій

ДО ЗАХИСТУ ДОПУЩЕНА

Зав.кафедрою _____
д.е.н.,доцент Левицький С.І.
«___» _____ 2023 р.

МАГІСТЕРСЬКА ДИПЛОМНА РОБОТА

РОЗРОБКА БІБЛІОТЕКИ ВИЯВЛЕННЯ РІЗКИХ ЗМІН СИГНАЛУ НА ОСНОВІ НЕЙРОННОЇ МЕРЕЖІ

Виконав
ст. гр. КІ-112м _____ В.І. Борщук
(підпис)

Керівник
к.т.н. _____ О.А. Хараджян
(підпис)

Запоріжжя
2023

ПРАТ «ПВНЗ «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ

ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра Інформаційних технологій

ЗАТВЕРДЖУЮ
Зав. кафедрою

_____
д.е.н.,доцент Левицький С.І.
«___» _____ 2023 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ ДИПЛОМНУ РОБОТУ

Студенту гр. *КІ – 112м*, спеціальності *«Комп'ютерна інженерія»*

Борщук Віталій Ігорович

1.Тема: *Розробка бібліотеки виявлення різких змін сигналу на основі нейронної мережі.*

затверджена наказом по інституту «___»_____ 2023 р. № ____

2. Термін здачі студентом закінченої роботи: «___»_____ 2023 р.

3. Перелік питань, що підлягають розробці:

1. Аналітичний огляд нейронних мереж
2. Огляд загальних принципів побудови нейронних мереж
3. Огляд бібліотек реалізації нейромереж
4. Розробка алгоритму виявлення фронтів в сигналі
5. Бібліотека математичних класів Alglib
6. Бібліотека для json-файлів nlohmann
7. Розробка програми виявлення фронтів в сигналі
8. Користувальницький інтерфейс програми
9. Розробка реалізації метода навчання нейронної мережі

4. Календарний графік підготовки кваліфікаційної роботи

№ етапу	Зміст	Терміни виконання	Готовність по графіку %, підпис керівника	Підпис керівника про повну готовність етапу, дата
1	Збір практичного матеріалу за темою кваліфікаційної бакалаврської роботи	04.09.23-17.10.23		
2	І атестація І розділ кваліфікаційної бакалаврської роботи	23.10.23-28.10.23		
3	ІІ атестація ІІ розділ кваліфікаційної бакалаврської роботи	20.11.23-25.11.23		
4	ІІІ атестація ІІІ розділ кваліфікаційної бакалаврської роботи, висновки та рекомендації, додатки, реферат	18.12.23-23.12.23		
5	Перевірка кваліфікаційної бакалаврської роботи на оригінальність	18.12.23-23.12.23		
6	Доопрацювання кваліфікаційної бакалаврської роботи, підготовка презентації, отримання відгуку керівника і рецензії	25.12.23-06.01.24		
7	Попередній захист кваліфікаційної бакалаврської роботи	08.01.24-13.01.24		
8	Подача кваліфікаційної бакалаврської роботи на кафедру	за 3 дні до захисту		
9	Захист кваліфікаційної бакалаврської роботи	15.01.24-20.01.24		

Дата видачі завдання: 16.01.2023 р.

Керівник кваліфікаційної бакалаврської роботи _____ (підпис) О.А. Хараджян (прізвище та ініціали)

Завдання отримав до виконання _____ (підпис) В.І. Борщук (прізвище та ініціали)

РЕФЕРАТ

Дипломна робота містить 66 стор., 7 рис., 0 таблиць, 2 додаток, 6 використаних джерел.

Об’єкт роботи: нейронні мережі.

Предмет роботи: аналіз сигналів з використанням нейронних мереж.

Мета роботи: розробка бібліотеки виявлення фронтів сигналів з використанням нейронних мереж.

Задачі роботи: аналітичний огляд нейронних мереж; огляд загальних принципів побудови нейронних мереж; огляд бібліотек реалізації нейромереж; розробка алгоритму виявлення фронтів в сигналі; бібліотека математичних класів Alglib; бібліотека для json-файлів nlohmann; розробка програми виявлення фронтів в сигналі; користувальницький інтерфейс програми; розробка реалізації метода навчання нейронної мережі; тестування розробленої системи.

В роботі розглянуто алгоритми визначенні різких змін у сигналах, які виникають у межах обмеженого часового інтервалу, з урахуванням завад та шуму в початкових сигналах. Для аналізу рівня палива в системах моніторингу автотранспорту використовуються спеціальні алгоритми, оскільки традиційні методи виявляють значні похибки через коливання палива та розходження у сигналах з датчиків. Застосування інструментів штучного інтелекту, таких як нейронні мережі, сприяє точному виявленню змін у рівні палива. Розроблено бібліотеку на основі нейронної мережі для визначення змін.

НЕЙРОННА МЕРЕЖА, ФРОНТ СИГНАЛУ, ALGLIB, NLOHMANN, QT

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Слово / словосполучення	Скорочення	Умови використання
Application programming interface	API	
Convolutional neural network	CNN	
JavaScript Object Notation	JSON	
Multilayer perceptron	MLP	
Recurrent neural network	RNN	

ВСТУП

Одною зі сфер застосування аналізу структури сигналів є аналіз рівня палива у системах моніторингу автотранспорту. Для такого аналізу пряме застосування звичайних алгоритмів дає значні похибки.

Визначення різких змін сигналів передбачає виявлення зміни сигнала за обмежений, наперед заданий інтервал часу. Це може бути визначення моменту зміни сигналу від нульового знвчення до максимально та навпаки (нарастаючий та спадаючий фронт), так і визначення наявності розтягнутого фронту сигналу. Задача виявлення змін сигналу ускладнюється наявністю завад та шуму в початковому сигналі. Загальні алгоритми визначення різких переходів сигналу в загальному випадку не можуть достовірно визначити наявність змін з незначною амплітудою.

Одним з засобів аналізу великої кількості різноманітних форм сигналів є нейронні мережі. Нейронні мережі відіграють важливу роль у сучасній обробці сигналів, забезпечуючи потужні можливості аналізу і розпізнавання. Завдяки своїй здатності вчитися на основі великої кількості даних, нейронні мережі можуть ефективно аналізувати різноманітні типи сигналів, такі як звукові, візуальні, радіочастотні та багато інших.

Застосування нейронних мереж у сфері аналізу сигналів дозволяє автоматизувати процеси фільтрації, класифікації та прогнозування. Вони можуть впевнено розпізнавати шаблони, виявляти аномалії і виконувати складні обчислення для отримання корисної інформації з сигналів.

Наприклад, у секторі телекомунікацій нейронні мережі використовуються для покращення якості передачі даних і роботи з радіосигналами.

Об’єкт роботи: нейронні мережі.

Предмет роботи: аналіз сигналів з використанням нейронних мереж.

Мета роботи: розробка бібліотеки виявлення фронтів сигналів з використанням нейронних мереж.

Задачі роботи:

- аналітичний огляд нейронних мереж;
- огляд загальних принципів побудови нейронних мереж;
- огляд бібліотек реалізації нейромереж;
- розробка алгоритму виявлення фронтів в сигналі;
- бібліотека математичних класів Alglib;
- бібліотека для json-файлів nlohmann;
- розробка програми виявлення фронтів в сигналі;
- користувальницький інтерфейс програми;
- розробка реалізації метода навчання нейронної мережі;
- тестування розробленої системи.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД НЕЙРОННИХ МЕРЕЖ

1.1. Загальні принципи побудови нейронних мереж

Визначення різких змін сигналів передбачає виявлення зміни сигнала за обмежений, наперед заданий інтервал часу. Це може бути визначення моменту зміни сигналу від нульового знвчення до максимально та навпаки (нарастаючий та спадаючий фронт), так і визначення наявності розтягнутого фронту сигналу. Задача виявлення змін сигналу ускладнюється наявністю завад та шуму в початковому сигналі. Загальні алгоритми визначення різких переходів сигналу в загальному випадку не можуть достовірно визначити наявність змін з незначною амплітудою.

Одною зі сфер застосування аналізу структури сигналів є аналіз рівня палива у системах моніторингу автотранспорту. Для такого аналізу пряме застосування звичайних алгоритмів дає значні похибки.

Основні джерела похибок це коливання палива, які пов’язані з прискореннями транспортного засобу, та зміна рівня при нахилах транспортного засобу. У зв’язку з особливостями реалізаціїї різних систем моніторингу сигнали з датчиків рівня палива, швидкості та нахилу можуть бути не синхронними і бути зміщеними на десятки секунд. Тому для надійного виявлення реальних змін рівня палива необхідно використовувати спеціальні алгоритми.

З урахуанням великої ріноманітності випадків спотворення сигналу, найбільш доцільним є використання засобів штучного інтелекту для виявлення цих змін.

Для вирішення таких задач використовуються два основні алгоритми: нейронна мережа та Random Forest Regressor. Також є дві додаткові напряму машинного навчання це вибір функцій і балансування даних, які мають великий вплив на результат алгоритму машинного навчання.

Нейронні мережі — це потужна група алгоритмів машинного навчання. Нейронна мережа створює мережу нейронів у розділених на шари. Кожен шар може складатися з одного або декількох нейронів, які з'єднані з іншими нейронними шарами через вагові коефіцієнти та зміщення. Зазвичай використовуються три різні типи шарів, перший з яких є вхідним. По-друге, вводиться один або більше шарів, що утворюють приховані шари. Нарешті йде вихідний рівень, який представляє вихід нейронної мережі. Залежно від типу нейронної мережі, алгоритми можуть створювати вихідні дані будь-якого з типів класифікаторів, створюючи вихідні дані, які вибирають один клас із кількох попередньо визначених класів. Навпаки, нейронні мережі також можуть бути регресорного типу, які використовуються для генерації вихідного значення замість вибору між попередньо визначеними класами. Що цікаво у зв'язку з цією тезою, так це тип нейронної мережі, яка використовується, і те, як вона використовується з набором даних. Простіше кажучи, є три різних типи нейронних мереж. Багатошарові персептрони (MLP) є досить простими мережами, які використовують принцип прямого зв'язку, як можна побачити на рис. 1.1. MLP є найпростішими нейронними мережами, і їх можна використовувати з чудовими результатами для більш простих завдань. По-друге, згорточні нейронні мережі (CNN) — це група нейронних мереж, які широко використовуються в обробці зображень і відео. CNN дуже схожі на MLP, але також додають тип фільтрації в процес, який підходить для обробки матриць, що представляють зображення. Нарешті, існує група рекурентних нейронних мереж (RNN). Ці типи нейронних мереж враховують вплив минулих даних, в основному зберігаючи інформацію про попередні екземпляри даних

часу для повторного використання під час створення нових прогнозів. Таким чином, RNN дуже корисні при аналізі даних часових рядів, таких як фондовий ринок, або накопичених сигналів системи. Було проведено багато досліджень і роботи в області RNN і здатності протидіяти залежним від часу наборам даних. Ефективно поєднання RNN і CNN для вирішення проблеми виявлення аномалій багаточасових рядів із чудовими результатами.

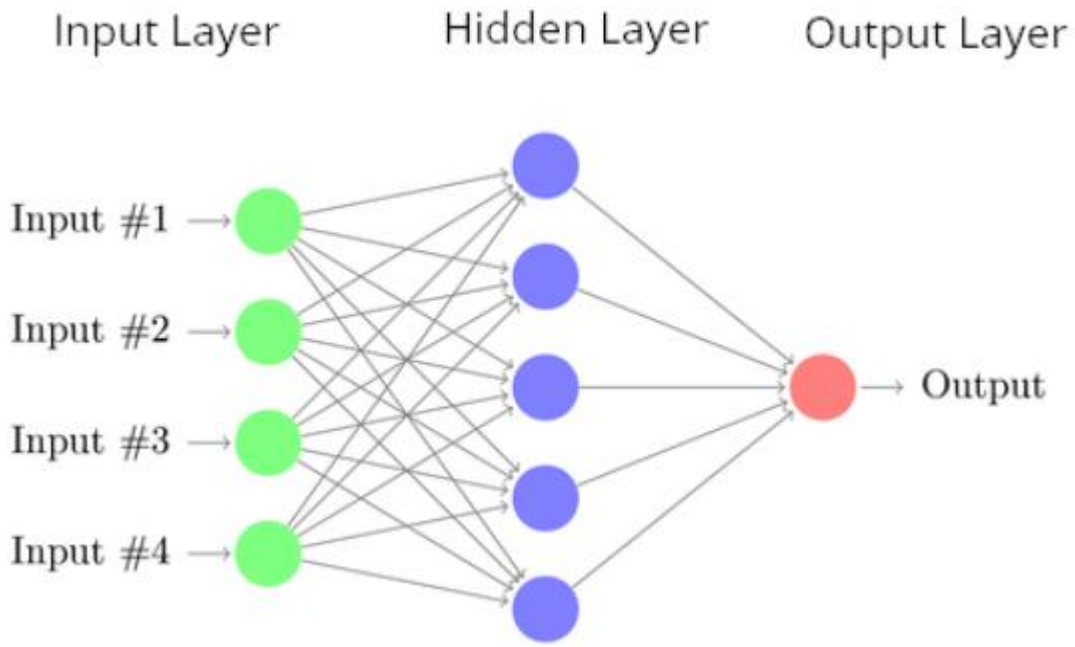

Рис. 1.1 Огляд базової структури нейронної мережі

Основну математичну структуру нейронної мережі можна побачити на рис. 1.2 Кожен нейрон шару з'єднаний з нейронами сусідніх шарів за допомогою вагових коефіцієнтів, позначених w, як показано на рис. 2.4. Потім вихід нейрона обчислюється шляхом взяття суми всіх нейронів, підключених до нього, де вихід кожного нейрона множиться на його відповідну вагу. Крім того, до суми додається зсув b. Нарешті, загальна сума та зсув на виході нейрона приймаються як вхідні дані для функції активації, яка дає можливість працювати з нелінійністю в мережі. Запровадження функції активації дає змогу мережі навчатися, що досягається шляхом попереднього навчання мережі. Існує

кілька різних функцій активації, включаючи лінійну функцію, сигмоподібну функцію та дотичну гіперболічну функцію

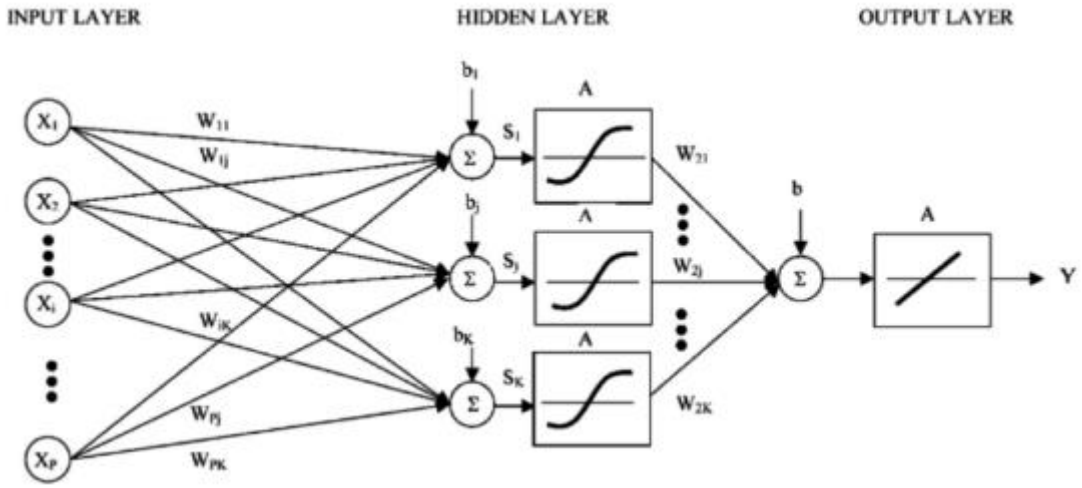

Рис. 2.2 Математична структури нейронної мережі.

Оскільки нейронні мережі належать до сімейства керованих алгоритмів машинного навчання, необхідне навчання нейронної мережі. Щоб налаштувати вагові коефіцієнти та зміщення мережі, використовується метод, який називається зворотним поширенням. По суті, на вихідному рівні вводиться термін помилки, який карає помилкові виходи. Потім помилки відстежуються, регулюючи ваги та зміщення в нейронах для кожного шару. Це робиться шляхом резервування великої частини набору вхідних даних для навчання нейронної мережі, в основному підключаючи вхідні сигнали до правильного виходу, завдяки чому мережа дізнається, які типи вхідних даних відповідають правильному виходу. Оскільки продуктивність нейронних мереж значною мірою залежить від фази навчання, важливо, щоб хороший навчальний набір містив багато різноманітних даних.

Random Forest Regressor — це алгоритм машинного навчання, який використовує ансамблеве навчання для поєднання результатів кількох дерев

рішень для підвищення продуктивності. Для ілюстрації дерева рішень можна описати як метод алгоритму, який порівнює точку даних X із встановленими межами A, B і C, як показано на рис. 1.3.

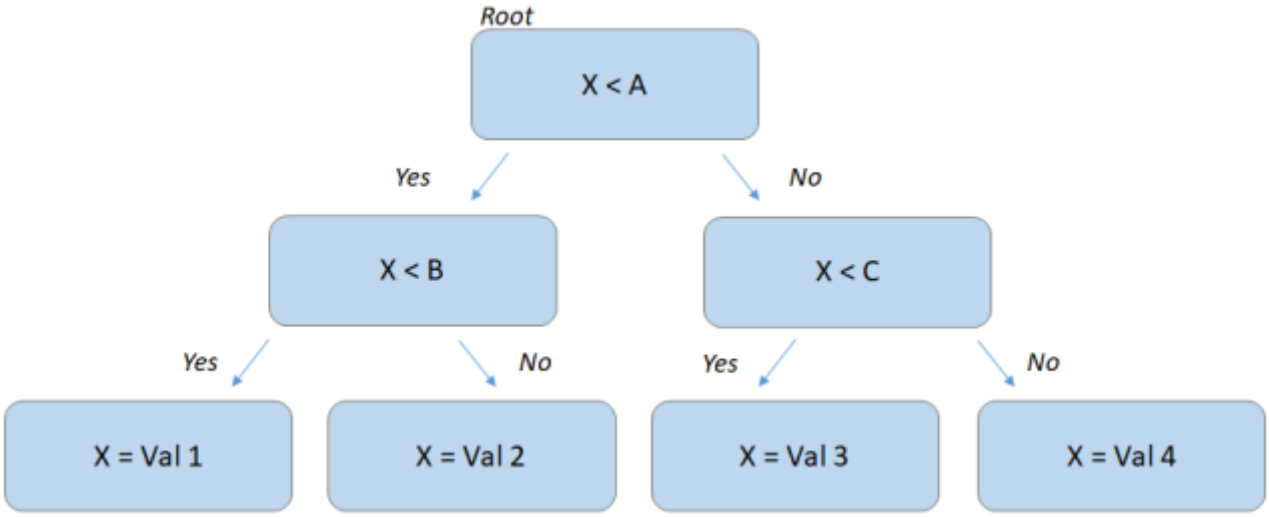

Рис. 1.3 Дерево рішень

Подібно до нейронної мережі або будь-якого іншого алгоритму машинного навчання, дерево рішень може бути типу регресії або класифікації. У типі класифікації дерев рішень виходом є один окремий клас із попередньо визначеного набору класів, а алгоритм називається Random Forest Classifer. Регресійний тип дерева рішень видає щось, що можна вважати дійсним числом, наприклад, ціна будинку або вихідний тиск перекачувального насоса. Random Forest Regressor схожий на частину нейронної мережі сімейства керованих алгоритмів машинного навчання. Таким чином, для налаштування меж дерева рішень потрібна фаза навчання. Навчання Random Forest Regressor включає повторне навчання всіх дерев рішень, доки не будуть отримані найточніші результати. Це означає, що межі A, B і C можна регулювати, доки не буде досягнуто найточніший результат відповідно до вихідних даних навчання.

Метод ансамблевого навчання використовується таким чином, що бере середній результат усіх дерев рішень, щоб зробити прогнози більш точними, де

проводиться більшість голосів усіх дерев рішень у Random Forest Regressor. У типі регресії Random Forest Regressor використовується середнє вихідне значення всіх дерев рішень, як показано на рис. 1.4. При роботі Random Forest Regressor, кожне дерево рішень починається з кореня, як показано на рис. 1.3. Щоб визначити кожну межу розглянуту як A, B і C у прикладі, вводиться функція штрафу. Часто використовується функція штрафу суми квадратів залишків. Випробовуючи всі можливі способи розділити дані шляхом коригування A, штраф отримано оцінку, яка в середньому показує, наскільки точним є результат для кожного значення обмеження A.

Потім вибирається межа, яка отримує найнижчий бал. Потім цей процес повторюється, де можна встановити мінімум точок даних, щоб вирішити, коли припинити повторення. Повторення цього процесу потім дає граничні значення A, B і C, які дають найкращі прогнози. У випадку, коли є кілька вхідних змінних, що є стандартною ситуацією, оцінка функції штрафу просто порівнюється між вхідними змінними, щоб знову вибрати межу з найнижчим показником для всіх вхідних змінних. Коли всі дерева рішень навчені, середній результат усіх дерев рішень вибирається як основний результат Random Forest Regressor, як показано на рис. 1.4.

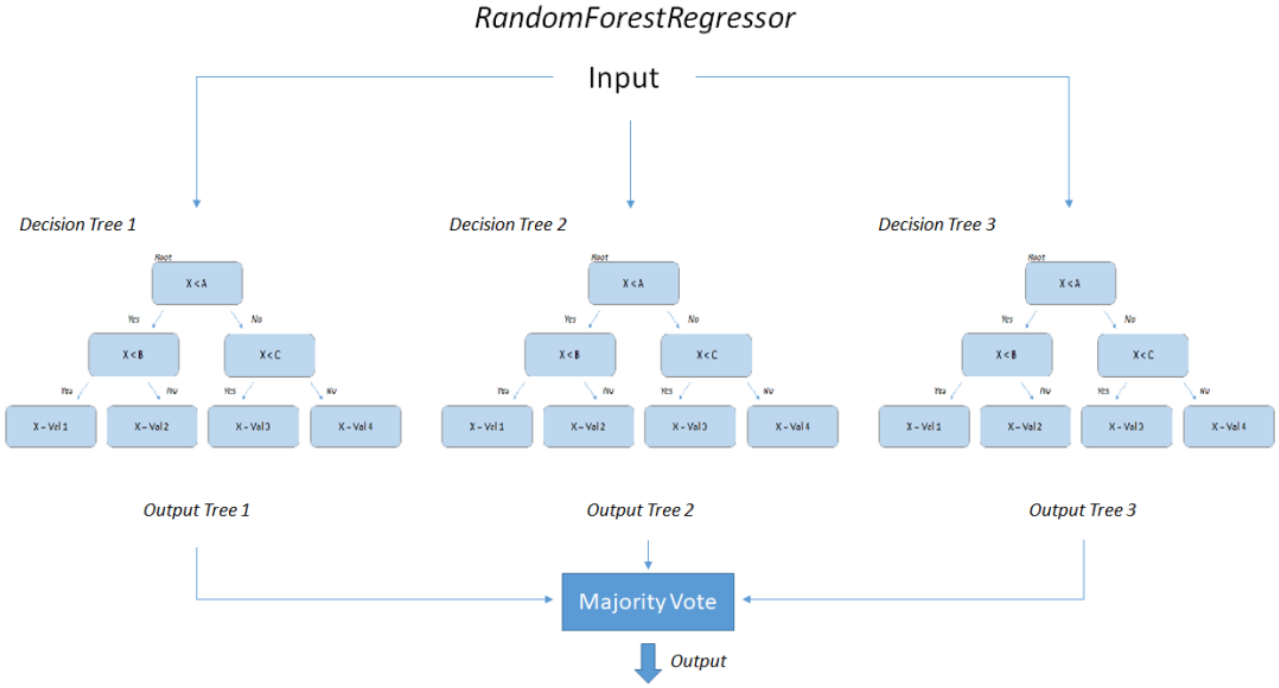

Рис. 1.4. Середній результат усіх дерев рішень

Значна частина продуктивності алгоритму ML полягає у виборі правильних функцій, які містять відповідну інформацію щодо цільового сигналу. Це особливо важливо, якщо набір даних містить сотні функцій на вибір. Багато функцій містять інформацію подібного типу, а деякі функції можуть навіть містити інформацію, яка абсолютно не має відношення до алгоритму ML. Методи вибору можуть виявитися дуже важливими при виборі функцій, оскільки вони можуть впливати на час обчислення, кореляцію та загальну ефективність. Використання двох функцій, які сильно корелюють одна з одною, може бути непотрібним.

Крім того, більшість функцій можуть мати низьку кореляцію з цільовим сигналом, що ускладнює правильний вибір правильних функцій. Це свідчить про те, що більш детальне вивчення фізичних і логічних зв’язків між функціями та цільовими сигналами може бути правильним шляхом. Використання двох ознак, які сильно корельовані одна з одною, не дає більше інформації для методу ML. Однак використання функцій, антикорельованих одна з одною, може допомогти змоделювати продуктивність. Крім того, також важливо пам’ятати, що вибір правильної підмножини функцій важливіший, ніж пошук окремих релевантних функцій. Функція, яка сама по собі може здатися нерелевантною, може бути дуже корисною в поєднанні з іншими функціями.

Новими способами вирішення проблеми вибору функцій і, таким чином, зменшення розмірів навчального набору є використання алгоритмів ML для вибору належних функцій.

Балансування даних. Дослідження показують, що алгоритми ML працюють погано, коли їх навчають на незбалансованих наборах даних. Велика кількість зібраних даних, як правило, містить інформацію з схожих середовищ і позицій, що може підвищити ефективність цих умов. З іншого боку,

прогностична потужність алгоритму значно знижується, коли представлених даних мало. Навчальні алгоритми на наборах даних, які містять дуже мало екземплярів певних типів точок даних, зазвичай створюють упереджені алгоритми ML, які мають вищу точність прогнозування для більшості даних, але нижчу точність прогнозування для меншості типів даних.

Важливим кроком, який може підвищити загальну продуктивність алгоритму машинного навчання, є балансування даних. Коли алгоритм ML навчається на незбалансованих наборах даних, модель легко перевищити, оскільки дані не завжди природно рівномірно розподіляються між різними функціями. Модель, навчена на збалансованому тренувальному наборі, також показала набагато кращі результати, використовуючи перевірку за винятком одного під час виявлення нових параметрів.

Нейронні мережі — це ряд алгоритмів, які імітують роботу мозку для розпізнавання зв’язків між величезними обсягами даних. Таким чином, вони, як правило, нагадують зв'язки нейронів і синапсів, які знаходяться в мозку.

Нейронні мережі з кількома рівнями процесів відомі як «глибокі» мережі та використовуються для алгоритмів глибокого навчання.

Хоча концепція інтегрованих машин, які можуть мислити, існувала століттями, нейронні мережі досягли найбільших успіхів за останні 100 років. У 1943 році Уоррен Маккаллох і Уолтер Піттс з Університетів Іллінойсу і Чикаго опублікували «Логічне обчислення ідей, властивих нервовій діяльності». В дослідженны проаналізовано, як мозок може виробляти складні шаблони та може бути спрощено до двійкової логічної структури лише з істинними/хибними зв’язками.

Френку Розенблату з Корнельської авіаційної лабораторії приписують розробку персептрона в 1958 році. Його дослідження ввели вагові коефіцієнти в роботу МакКолоха та Пітта, і Розенблат використав свою роботу, щоб

продемонструвати, як комп’ютер може використовувати нейронні мережі для виявлення уявлень і робити висновки.

Після спаду досліджень у 1970-х роках Джон Хопфілд представив Hopfield Net, статтю про рекурентні нейронні мережі в 1982 році. Крім того, знову виникла концепція зворотного поширення, і багато дослідників почали використовувати її потенціал для нейронних мереж.

У багатошаровому персептроні (MLP) персептрони розташовані у взаємопов’язаних шарах. Вхідний рівень збирає вхідні шаблони. Вихідний рівень має класифікації або вихідні сигнали, на які можуть відображатися вхідні моделі. Наприклад, шаблони можуть містити список величин для технічних індикаторів цінного паперу; потенційні результати можуть бути «купити», «тримати» або «продавати».

Приховані шари точно налаштовують вхідні ваги, доки межа похибки нейронної мережі не стане мінімальною. Існує гіпотеза, що приховані шари екстраполюють помітні характеристики у вхідних даних, які мають передбачувану силу щодо вихідних даних. Це описує вилучення ознак, яке виконує корисність, подібну до статистичних методів, таких як аналіз головних компонентів.

Нейронні мережі прямого зв’язку є одним із найпростіших типів нейронних мереж. Вони передають інформацію в одному напрямку через вхідні вузли; ця інформація продовжує оброблятися в цьому єдиному напрямку, поки не досягне виходу з мережі. Нейронні мережі прямого зв’язку можуть мати приховані рівні для функціональності, і цей тип найчастіше використовується для технологій розпізнавання обличчя.

Рекурентні нейронні мережі. Більш складний тип нейронних мереж, рекурентні нейронні мережі отримують вихідні дані вузла обробки та передають інформацію назад у мережу. Це призводить до теоретичного «навчання» та вдосконалення мережі. Кожен вузол зберігає історію процесу, і

ці історичні процеси повторно використовуються в майбутньому під час обробки.

Це стає особливо критичним для мереж, у яких передбачення є неправильним; система намагатиметься дізнатися, чому стався правильний результат, і відповідно коригуватиметься. Цей тип нейронної мережі часто використовується в програмах перетворення тексту в мовлення.

Згорткові нейронні мережі, також звані ConvNets або CNN, мають кілька рівнів, на яких дані сортуються за категоріями. Ці мережі мають вхідний рівень, вихідний рівень і приховану множину згорткових шарів між ними. Шари створюють карти функцій, які записують ділянки зображення, які далі розбиваються, доки вони не отримають результати. Ці рівні можуть бути об’єднані або повністю з’єднані, і ці мережі є особливо корисними для програм розпізнавання зображень.

Деконволюційні нейронні мережі просто працюють навпаки згорткових нейронних мереж. Застосування мережі полягає у виявленні елементів, які могли бути визнані важливими в згортковій нейронній мережі. Ці елементи, ймовірно, були б відкинуті під час процесу виконання згорткової нейронної мережі. Цей тип нейронної мережі також широко використовується для аналізу або обробки зображень.

Модульні нейронні мережі містять кілька мереж, які працюють незалежно одна від одної. Ці мережі не взаємодіють одна з одною під час процесу аналізу. Натомість ці процеси виконуються для більш ефективного виконання складних обчислювальних процесів.

Переваги нейронних мереж.

Часто може працювати ефективніше та довше, ніж люди.

Може бути запрограмований на навчання з попередніх результатів, щоб прагнути зробити більш розумні майбутні розрахунки.

Часто використовують онлайн-послуги, які зменшують (але не усувають) систематичний ризик.

Постійно розширюються в нових галузях із більш складними проблемами.

Недоліки нейронних мереж.

Усе ще покладайтеся на апаратне забезпечення, для обслуговування якого може знадобитися робота та досвід.

Розробка коду та алгоритмів може зайняти багато часу.

Може бути важко оцінити помилки або адаптацію до припущень, якщо система самонавчається, але їй бракує прозорості.

Зазвичай повідомляють приблизний діапазон або приблизну суму, яка може не відбутися.

Нейронна мережа має три основні компоненти: вхідний шар, внутрішні шари рівеня обробки та вихідний шар. Вхідний шар забезпечує нормалізацію даних на основі різних методів.

Також відома як мережа глибокого навчання, глибока нейронна мережа, у своєму найпростішому вигляді, включає два або більше рівнів обробки. Глибокі нейронні мережі покладаються на мережі машинного навчання, які постійно розвиваються, порівнюючи оцінені результати з фактичними результатами, а потім змінюючи майбутні прогнози.

Усі нейронні мережі мають три основні компоненти. По-перше, вхідні дані - це дані, введені в мережу, які підлягають аналізу. По-друге, рівень обробки використовує дані (і попередні знання про подібні набори даних) для формулювання очікуваного результату. Цей результат є третім компонентом, і цей третій компонент є бажаним кінцевим продуктом аналізу.

Нейронні мережі — це складні інтегровані системи, які можуть виконувати аналітику набагато глибше та швидше, ніж здатна людина. Існують різні типи нейронних мереж, які часто найкраще підходять для різних цілей і

цільових результатів. У фінансах нейронні мережі використовуються для аналізу історії транзакцій, розуміння руху активів і прогнозування результатів фінансового ринку.

1.2. Аналіз бібліотек реалізації нейромереж

Розглянемо деякі бібліотеки нейронних мереж з відкритим кодом.

Бібліотеки нейронних мереж зазвичай використовуються для реалізації нейронних мереж у комп'ютерних програмах. Протягом багатьох років багато з цих бібліотек було розроблено та вдосконалено, щоб полегшити реалізацію та використання функцій обробки нейронних мереж.

Бібліотека TensorFlow була вперше розроблена у 2015 році командою Google Brain, дослідницької групи Google з машинного навчання. Вона була розроблена для полегшення досліджень у створенні моделей машинного навчання та нейронних мереж. Таким чином, TensorFlow пропонує просте створення моделей і надійне середовище для розгортання моделей машинного навчання. TensorFlow можна використовувати на платформах Windows, Linux, Android і macOS.

Бібліотека CNTK також відома як Microsoft Cognitive Toolkit. CNTK, вперше випущена у 2016 році. Це бібліотека з відкритим кодом для навчання нейронних мереж для глибокого навчання. Це дозволяє користувачам створювати та поєднувати нейронні мережі, які часто використовуються, наприклад згорткові нейронні мережі (CNN) і рекурентні нейронні мережі (RNN). CNTK підтримується на платформах Linux, Windows і macOS.

Бібліотека PyTorch випущена у 2016 році PyTorch була розроблена дослідницькою лабораторією штучного інтелекту Facebook. PyTorch створено на основі Torch, бібліотеки машинного навчання та наукових обчислень. За

словами розробників PyTorch, це фреймворк машинного навчання, який прискорює створення дослідницького прототипу до розгортання у виробництві.

PyTorch має розподілене навчання моделей нейронних мереж, підтримку хмари, надійну екосистему та підтримується на платформах Windows, Linux і macOS.

Theano — це бібліотека з відкритим кодом, випущена в 2007 році Монреальським інститутом навчання алгоритмів (MILA). Theano пропонує підтримку для оцінки математичних виразів, які зазвичай використовуються для машинного навчання. Вона надає інтерфейс Python, який можна використовувати разом із Keras. Theano також підтримується в операційних системах Linux, Windows і macOS.

Caffe — це структура глибокого навчання, розроблена в 2013 році групою Berkeley AI Research. Вона була розроблена, щоб забезпечити швидкість і модульність у створенні нейронних мереж. Однак Caffe здебільшого підтримує нейронні мережі для класифікації та сегментації зображень. Зараз вона працює в операційних системах Linux, macOS і Windows.

Бібліотека Keras була розроблена Франсуа Шолле, інженером Google у 2015 році. Вона забезпечує зручний інтерфейс для створення нейронних мереж на Python. У Keras є такі модулі, як функції активації та рівні для впровадження нейронних мереж у декілька кроків для швидкого експериментування. Крім того, бібліотека може працювати поверх інших бібліотек, таких як TensorFlow і CNTK.

DeepLearning4J — це бібліотека глибокого навчання, написана для Java і Scala і вперше випущена в 2014 році. Вона містить розподілене обчислювальне навчальне середовище, яке може прискорити продуктивність. DeepLearning4J дозволяє користувачам гнучко створювати та поєднувати моделі нейронних мереж. Наразі DeepLearning4J підтримується на платформах Windows, Linux і

macOS. Крім того, вона добре працює разом з іншими бібліотеками нейронних мереж, такими як TensorFlow і Keras.

Особливості бібліотек нейронних мереж.

Бібліотеки нейронних мереж дозволяють інтуїтивно визначати обчислювальний графік (нейронну мережу) з меншою кількістю коду.

Використаний динамічний граф обчислень забезпечує гнучку побудову мережі під час виконання. Бібліотека може використовувати як парадигми статичного, так і динамічного графа.

Більшість коду бібліотеки написано мовою C++14. Використовуючи C++14 core API, ви можете розгорнути його на вбудованих пристроях.

У бвблвотеки є гарна абстракція функції, а також генератор шаблону коду для написання нової функції. Це дозволяє розробникам писати нову функцію з меншим кодуванням.

Новий код пристрою можна додати як плагін без жодних змін у коді бібліотеки. CUDA фактично реалізовано як розширення плагіна.

Чутливість до жестів. Бібліотека використовується в інтуїтивно зрозумілій функції чутливості до жестів Sony Mobile Communications «Xperia Ear». На основі даних кількох датчиків, вбудованих у Xperia Ear, ви можете просто кивнути головою, щоб підтвердити команду – відповісти так/ні, відповісти на телефонний дзвінок/відхилити телефонний дзвінок, скасувати читання тексту в мовлення сповіщень, пропускати/перемотувати назад трек пісні.

Розпізнавання зображень (ідентифікація користувача, відстеження обличчя тощо)

Бібліотека використовується для реалізації розпізнавання зображень розважального робота Sony "aibo" ERS-1000. Під час розпізнавання зображень за допомогою камер «риб’яче око», встановлених на носі, бібліотека активно використовується для ідентифікації користувача, відстеження обличчя,

розпізнавання зарядного пристрою, загального розпізнавання об’єктів тощо. Ці функції та різноманітні вбудовані датчики забезпечують її адаптивну поведінку.

Можена визначити обчислювальний графік (нейронну мережу) інтуїтивно з меншою кількістю коду.

Визначення двошарової нейронної мережі з softmax втратою вимагає лише наступних простих 5 рядків коду.

```
xx = nNn.Variable(input_shape)
tt = nNn.Variable(target_shape)
hh = F.tanh(PF.affine(xx, hidden_size, name='affine_1'))
yy = PF.affine(hh, target_size, name='affine_2')
loss = F.mean(F.softmax_cross_entropy(yy, tt))
```

Виклик обчислення прямого та зворотнього розрахунку виконує обчислювальний графік.

```
xx.d = some_data
tt.d = some_target
loss.forward()
loss.backward()
```

Система керування параметрами під назвою parameter_scope забезпечує гнучкий спільний доступ до параметрів.

Наступний блок коду демонструє, як ми пишемо просту рекурентну нейронну мережу Elman.

```
hh = h0 = nn.Variable(hidden_shape)
xx_list = [nn.Variable(input_shape) for i in range(input_length)]
```

```
    for xx in xx_list:
    # області дії параметрів повторно використовуються через цикл,
    # це означає, що параметри є спільними для всіх ітерацій.
        with nn.parameter_scope('rnn_cell'):
            hx = F.tanh(PF.affine(F.concatenate(xx, hh, axis=1),
hidden_size))
    yy = PF.affine(hh, target_size, name='fn_affine')
```

Підтримка графів динамічних обчислень.

Статичний граф обчислень, який використовувався загалом, є методом побудови графіка обчислень перед виконанням графіка. З іншого боку, граф динамічних обчислень забезпечує гнучку побудову мережі під час виконання. Бібліотека може використовувати як парадигми статичного, так і динамічного графа. Ось приклад графіка динамічного обчислення в бібліотеці.

```
    xx = nnn.Variable(input_shape)
    xx.d = some_data
    tt = nn.Variable(target_shape)
    tt.d = some_target
    with nnn.auto_forward():
        hh = F.relu(PF.convolution(xx, hidden_size, (3, 3),
pad=(1, 1), name='func_conv0'))
        for i in range(num_stochatic_layers):
            if np.random.rand() < layer_drop_ratio:
                continue  # Стохастичний шар.
            h2 = F.relu(PF.convolution(чх, hidden_size, (3, 3),
pad=(1, 1),
                       name='агтс_conv%d' % (i + 1)))
            hp = F.add2(hp, h2)
        y = PF.affine(ph, target_size, name='classification')
        loss = F.mean(F.softmax_cross_entropy(yy, tt))
    # Зворотні обчислення також можна виконати в динамічно
```

```
loss.backward()
```

Система кешування пам’яті, реалізована в бібліотеці, забезпечує швидке виконання без накладних витрат на розподіл пам’яті.

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМУ ВИЯВЛЕННЯ ФРОНТІВ В СИГНАЛІ

2.1. Бібліотека математичних класів ALGLIB

ALGLIB — це кросплатформна бібліотека числового аналізу та обробки даних. Вона підтримує п'ять мов програмування (C++, C#, Java, Python, Delphi) і кілька операційних систем (Windows і POSIX, включаючи Linux). Функції ALGLIB включають:

- Аналіз даних (класифікація/регресія, статистика)
- Оптимізація та нелінійні вирішувачі
- Інтерполяція та лінійне/нелінійне підбирання найменших квадратів
- Лінійна алгебра (прямі алгоритми, EVD/SVD), прямі та ітераційні лінійні розв’язувачі
- Швидке перетворення Фур'є та багато інших алгоритмів

Проект ALGLIB пропонує вам кілька версій ALGLIB:

ALGLIB Free Edition - доставляється безкоштовно за ліцензією GPL або Personal/Academic, пропонує повний набір числових функцій, широка алгоритмічна оптимізація, однопотоковий, ліцензійна угода не підходить для більшості комерційних програм.

ALGLIB Commercial Edition - гнучка комерційна ліцензія без роялті чи комісії за розповсюдження, широка алгоритмічна оптимізація, високопродуктивна версія C++ (SMP, комерційні ядра HPC), дві версії C# - керована та HPC (власний код, SMP/SIMD), комерційна підтримка та гарантії.

Загальні особливості ALGLIB:

- портативний - його можна скомпілювати практично будь-де за допомогою майже будь-якого компілятора;
- простий у використанні. Легка інтеграція, супроводжується великою документацією;
- ефективний. Глибокі алгоритмічні та низькорівневі оптимізації всередині.

Нейронні мережі є одним із найбільш гнучких і потужних методів інтелектуального аналізу даних. Вони можуть вирішувати проблеми регресії, класифікації, прогнозування. Нейронні мережі успішно застосовуються в багатьох сферах - від фінансових моделей до медичних проблем. Пакет ALGLIB включає одну з найкращих нейронних мереж на ринку: багато різних архітектур, ефективні алгоритми навчання, вбудовану підтримку перехресної перевірки. Як безкоштовна, так і комерційна версії можуть вирішувати однаковий набір обчислювальних проблем. Комерційне видання, однак, включає такі функції, пов’язані з продуктивністю, як оптимізований код із підтримкою SSE (для користувачів x86/x64) і підтримка багатопоточності. Ці функції відсутні у безкоштовному випуску.

Архітектури нейронних мереж у ALGLIB. ALGLIB підтримує нейронні мережі без прихованих шарів, з одним прихованим шаром і з двома прихованими шарами. «Shortcut» підключень із вхідного рівня безпосередньо до вихідного рівня не підтримуються.

Приховані шари мають одну зі стандартних сигмоподібних функцій активації, однак для вихідного рівня нейронної мережі може бути доступна більша різноманітність. Вихідний шар може бути:

- лінійні (такі мережі використовуються в апроксимаційних задачах)
- лінійний з SOFTMAX-нормализацией
- сигмоподібний, обмежений зверху ТА знизу (вихід мережі відображається у вказаному діапазоні [a,b])

– обмежений зверху АБО знизу. Найпростіший випадок, коли ця функція прагне до x, коли x прагне до +∞, і експоненціально прагне до нуля, коли x прагне до -∞.

Нейронні мережі з лінійним вихідним рівнем і вихідною SOFTMAX-нормалізацією становлять окремий випадок. Вони використовуються для завдань класифікації, де мережеві виходи повинні бути невід'ємними, а їхня сума повинна бути строго дорівнювати одиниці, що дозволяє використовувати їх як ймовірність того, що вхідний вектор буде віднесено до одного з класів. Кількість виходів у такій мережі завжди не менше двох (що є обмеженням елементарної логіки).

Такого набору архітектур, незважаючи на мінімалістичність, достатньо для вирішення більшості практичних завдань. Можна зосередитися на проблемі (класифікації чи наближенні), не приділяючи зайвої уваги деталям (наприклад, вибір певної функції активації прихованого шару зазвичай мало впливає на результат).

ALGLIB пропонує багату нейронну функціональність. Ці функції включають автоматичну нормалізацію даних, регулярізацію, навчання з випадковими перезапусками, перехресну перевірку.

Попередня обробка даних — це нормалізація навчальних даних — вхідні та вихідні дані нормалізуються, щоб мати одиничне середнє/відхилення. Попередня обробка має важливе значення для швидкої збіжності алгоритму навчання - вона може навіть не збігатися з погано масштабованими даними. Пакет ALGLIB автоматично аналізує набір даних і вибирає відповідне масштабування для входів і виходів. Вхідні дані автоматично масштабуються перед подачею в мережу, а вихідні дані мережі автоматично змінюються після обробки. Попередня обробка виконується прозоро для користувача, вам не потрібно про це турбуватися - просто передайте дані в навчальний алгоритм.

Регуляризація (відома також як спад ваги) є ще однією важливою функцією, реалізованою ALGLIB. Правильно підібраний коефіцієнт розпаду значно покращує як похибку узагальнення, так і швидкість збіжності.

Навчання з перезапусками — це спосіб подолати проблему поганих локальних мінімумів. У дуже нелінійних задачах алгоритм навчання може сходитися до стану мережі, який є локально оптимальним (тобто не може бути покращений невеликими кроками), але дуже далекий від найкращого можливого рішення (неоптимальний). У таких випадках можна виконати кілька перезапусків алгоритму навчання з випадкових позицій і вибрати найкращу мережу після навчання. ALGLIB робить це автоматично і дозволяє вказати бажану кількість перезапусків перед тренуванням.

Перехресна перевірка є добре відомою процедурою для отримання оцінок помилки узагальнення без окремого тестового набору. ALGLIB дозволяє виконувати перехресну перевірку лише одним викликом — вказується кількість згортань, а пакет обробляє все інше.

Обидві версії ALGLIB (безкоштовна та комерційна) включають розширені методи навчання з багатьма вдосконаленими алгоритмами. На додаток до цього, Commercial Edition включає наступні важливі оптимізації: багатопотоковість, внутрішні SSE (для користувачів C++), оптимізоване нативне ядро (для користувачів C#).

ALGLIB може розпаралелювати такі операції:

Обробка набору даних (дані розбиваються на пакети, які обробляються окремо),

Навчання з випадковими перезапусками (навчальні сеанси, що відповідають різним початковим точкам, можуть виконуватися в різних потоках),

Перехресна перевірка (паралельне виконання різних перехресних раунди перевірки).

Паралельні нейронні мережі можуть працювати на будь-якій багатоядерній системі - від x86/x64 до процесорів SPARC і ARM! Також підтримуються багатопроцесорні системи. Нейронні мережі легко розпаралелювати, тому слід очікувати майже лінійного прискорення від паралельного функціонування!

Робота з нейронними мережами. Функціональні функції, пов’язані з нейронними функціями, розташовані в двох підпакетах ALGLIB:

mlpbase - основні нейронні функції (обробка, модифікація) і структури даних, не пов'язані з навчанням;

mlptrain - функції та структури даних, що використовуються для навчання нейронних мереж.

Створення об'єкта тренера. Якщо необхідно навчити одну або кілька мереж, то потрібно почати зі створення об'єкта тренера. Об’єкт Trainer – це спеціальний об’єкт, який зберігає набір даних, налаштування навчання та тимчасові структури, які використовуються для навчання. Об’єкт Trainer створюється за допомогою функцій mlpcreatetrainer або mlpcreatetrainercls. Перша використовується, коли розв’язується задача регресії (прогноз числових залежних змінних), друга використовується для задач класифікації.

Об’єкт Trainer можна використовувати для навчання кількох мереж з однаковими наборами даних і налаштуваннями навчання. У цьому випадку мережі повинні навчатися по черзі - не можна поділитися об'єктом тренера між різними потоками.

Визначення набору даних. Наступним кроком є завантаження набору даних в об’єкт тренера. Перш за все, необхідно закодувати дані - перетворити їх із необробленого представлення (яке може містити як числові, так і категоричні дані) у числову форму.

Після того, як ваші дані були закодовані та збережені як двовимірна матриця, можна передати цю матрицю до функції mlpsetdataset. Якщо дані

розріджені, можна заощадити багато пам’яті, зберігаючи їх у структурі sparsematrix і передавайте його до mlpsetsparsedataset. Використання розрідженої матриці для зберігання даних може заощадити багато пам’яті, але не дасть додаткового прискорення.

ALGLIB виконує автоматичну попередню обробку даних перед навчанням, тож не потрібно зміщувати/масштабувати змінні, тому вони матимуть нульове середнє значення та одиничну дисперсію. Дані неявно масштабуються перед передачею в мережу. Мережевий вихід також автоматично змінюється, перш ніж його повернути.

Створення нейронної мережі. Після того як створено навчальний об’єкт і підготований набір даних, можно створити мережевий об’єкт. Об’єкт нейронної мережі зберігає таку інформацію:

- архітектура мережі,
- нейронні ваги.

Архітектура та ваги повністю описують нейронну мережу. Нейронна архітектура включає наступні компоненти:

- кількість входів,
- кількість і розміри прихованих шарів,
- розмір вихідного шару,
- тип вихідного шару.

Кількість вхідних даних визначається набором даних, необхідно вибрати рівно стільки вхідних даних, скільки є "вхідних" змінних у наборі даних.

Приховані шари можна вибрати довільно. Можна вибрати мережу без прихованих шарів (вхідний рівень підключається безпосередньо до вихідного), мережу з одним або двома прихованими шарами. Необхідо вибрати кількість нейронів у кожному шарі.

Кількість виходів знову визначається набором даних.

Вихідний рівень може бути лінійним (використовується для задач регресії) або SOFTMAX-нормализованим (використовується для задач класифікації). Ці два типи вихідного шару добре працюють у більшості випадків. Також можна створити мережі, виходи яких обмежені інтервалом або напівінтервалом (зазначеним під час створення мережі).

Після того як визначина мережева архітектура необхідно викликати функцію створення мережі.

Функція mlpcreate0, mlpcreate1 або mlpcreate2 для створення регресійної мережі (лінійного вихідного шару) з 0, 1 або 2 прихованими шарами.

Функція mlpcreatec0, mlpcreatec1 або mlpcreatec2 для створення мережі класифікатора (вихідний рівень SOFTMAX) з 0, 1 або 2 прихованими шарами.

Функція mlpcreater0, mlpcreater1 або mlpcreater2 для створення мережі регресії з 0, 1 або 2 прихованими шарами та вихідним шаром зі спеціальною функцією активації, обмеженою інтервалом [a,b].

Функція mlpcreateb0, mlpcreateb1 або mlpcreateb2 для створення мережі регресії з 0, 1 або 2 прихованими шарами та вихідним шаром зі спеціальною функцією активації, обмеженою напівінтервалом.

Навчання нейронної мережі. Поточна версія ALGLIB пропонує один алгоритм навчання - пакетний L-BFGS. Оскільки це пакетний алгоритм, він обчислює градієнт для всього набору даних перед оновленням ваг мережі. Внутрішньо цей алгоритм використовує метод оптимізації L-BFGS для мінімізації помилок мережі. L-BFGS — це метод вибору для задач нелінійної оптимізації — швидкий і потужний алгоритм оптимізації, придатний навіть для задач великого масштабу.

Окрему мережу можна навчити за допомогою функції mlptrainnetwork. Функція приймає як параметри об'єкт тренера S, мережевий об'єкт net і кількість перезапусків NRestarts. Якщо вказати NRestarts>1, об’єкт тренера виконає кілька тренувань, розпочатих із різних випадкових позицій, і вибере

найкращу мережу (таку, яка має мінімальну помилку в навчальному наборі). Комерційна версія ALGLIB може виконувати ці навчальні сеанси паралельно, безкоштовна версія виконує лише послідовне навчання.

Якщо NRestarts=1, мережа навчається з випадкового початкового стану. Якщо NRestarts=0, мережа навчається без рандомізації (вихідний стан використовується як початкова точка).

Пакет ALGLIB підтримує регулярізацію (також відому як розпад ваги). Правильно обраний коефіцієнт регуляризації покращує як швидкість збіжності, так і помилку узагальнення. Ви можете встановити коефіцієнт регулярізації за допомогою функції mlpsetdecay, яку слід викликати перед навчанням. Якщо невідомо, яке значення Decay вибрати, то необхідно експериментально вибрати значення в діапазоні від 0,001 (слабка регулярізація) до 100 (дуже сильна регулярізація). Значення потрібно шукати, починаючи з мінімального та збільшуючи значення Decay у 3-10 разів на кожному кроці, одночасно перевіряючи, шляхом перехресної перевірки або за допомогою тестового набору, помилку узагальнення мережі.

Крім того, перед тренуванням можена вказати критерії зупинки. Це можна зробити за допомогою функції mlpsetcond, яка замінює налаштування за замовчуванням. Ви можете вказати наступні критерії зупинки: досить мала зміна вагових коефіцієнтів WStep або перевищення максимальної кількості ітерацій (епох) MaxIts. Доцільно вибрати число порядку 0,001 як WStep. Іноді, якщо проблему дуже важко вирішити, її можна зменшити до 0,0001, але зазвичай достатньо 0,001.

Досить мале значення функції помилки служить критерієм зупинки в багатьох пакетах нейронних мереж. Проблема в тому, що, маючи справу з реальною проблемою, а не з освітньою, ви заздалегідь не знаєте, наскільки адекватно її можна вирішити. Деякі проблеми можна розв’язати з дуже низькою похибкою, тоді як 26% похибки класифікації вважається хорошим результатом

вирішення певних задач. Тому немає сенсу вказувати «досить незначну помилку» як критерій зупинки. Поки не вирішина проблема, невідомо про значення, яке слід вказати, тоді як після вирішення проблеми немає необхідності вказувати будь-який критерій зупинки.

Тепер до останнього моменту навчання окремих нейронних мереж. mlptrainnetwork дозволяє тренувати мережу за допомогою лише одного виклику, але всі деталі навчання приховані в цьому виклику. Ця функція викликається лише тоді, коли результат готовий. Однак інколи необхідно стежити за прогресом навчання. У цьому випадку можна використовувати пару функцій - mlpstarttraining і mlpcontinuetraining - для виконання нейронного навчання. Ці функції дозволяють виконувати тренування крок за кроком і стежити за його прогресом.

Набір тестів і перехресна перевірка. Після навчання мережі ви можете почати використовувати її для вирішення деяких реальних проблем. Але є ще одна річ, яку необхідно виконати – оцінка похибки узагальнення. Нейронна мережа може добре працювати на даних, які використовуються для навчання, але її продуктивність на нових даних зазвичай гірша. Можна сказати, що результати мережі на навчальному наборі є оптимістично упередженими.

Одним із способів оцінити похибку узагальнення мережі є використання тестового набору – абсолютно нового набору даних, який не використовувався для: навчання мережі, вибору найкращої мережі, вибору архітектури мережі тощо. Помилка мережі на тестовому наборі може бути обчислена за допомогою наступних функцій: mlpallerrorssubset або mlpallerrorssparsesubset (для розріджених наборів даних). Вони повертають декілька видів помилок (RMS, середнє, середнє відносне та інші) для частини набору даних (subsetsize≥0) або для повного набору даних (subsetsize<0).

Тестовий набір є найкращим рішенням - якщо є достатньо даних для створення окремого тестового набору, який більше ніде не використовується.

Але часто недостатньо даних - у цьому випадку можна використовувати перехресну перевірку.

Процедура перехресної перевірки повертає оцінку помилки тестового набору - оцінку, яка була отримана з використанням лише навчального набору. Результати перехресної перевірки майже неупереджені. K-кратна перехресна перевірка є дорогою процедурою – вона передбачає навчання K незалежних нейронних мереж на K дещо різних наборах даних.

На відміну від набору тестів, перехресна перевірка не оцінює властивості узагальнення однієї конкретної нейронної мережі. Вона оцінює властивості узагальнення нейронної архітектури в поєднанні з методом навчання (конкретний алгоритм, який використовується для навчання, критерії зупинки, регулярізація, кількість перезапусків, набір даних). Якщо виконується перехресна перевірка, дуже важливо використовувати ті самі параметри навчання (включно з кількістю перезапусків), які використовувалися для навчання мережі.

ALGLIB дозволяє виконувати K-кратну перехресну перевірку за допомогою функції mlpkfoldcv. В якості одного зі своїх параметрів ця функція приймає нейронні. Ця функція повністю вирішує всі проблеми, пов’язані з CV (розділення навчального набору, навчання окремих мереж, обчислення помилок).

Робота з нейронними мережами. Після того, як нейронна мережа навчена та перевірено її властивості узагальнення, можна почати її фактично використовувати! Більшість нейронних функцій знаходяться у підпакеті mlpbase.

Функція обробки - mlpprocess, яка повертає мережевий вихід у для входу x.

Функції серіалізації - mlpserialize і mlpunserialize, які дозволяють конвертувати мережу в/з рядка.

Функції перевірки/модифікації мережі - близько 20 функцій, які дозволяють досліджувати структуру мережі або змінювати її ваги.

2.2. Бібліотека для json-файлів nlohmann

Особливості бібліотеки

Тривіальна інтеграція. Весь код бібліотеки складається з одного файлу заголовків json.hpp. Клас написаний мовою C++11. Загалом, усе не потребує налаштування прапорів компілятора чи налаштувань проекту.

Ефективність пам'яті. Кожен об’єкт JSON має накладні витрати в один покажчик (максимальний розмір об’єднання) та один елемент перерахування (1 байт). Стандартне узагальнення використовує такі типи даних C++: std::string для рядків, int64_t, uint64_t або double для чисел, std::map для об’єктів, std::vector для масивів і bool для логічних значень. Однак ви можете створити шаблон узагальненого класу basic_json відповідно до ваших потреб.

Швидкість. Звичайно, існують швидші бібліотеки JSON. Однак якщо ваша мета — пришвидшити розробку, додавши підтримку JSON за допомогою єдиного заголовка, тоді ця бібліотека — це ваш шлях.

Розвинена система прикладів. Кожна функція API (задокументована в Документації API) має відповідний окремий файл прикладу. Наприклад, функція emplace() має відповідний файл прикладу emplace.cpp.

Клас json надає API для маніпулювання значенням JSON. Щоб створити об’єкт JSON шляхом читання файлу JSON:

```
#include <fstream>
```

```
#include <nlohmann/json.hpp>
using json = nlohmann::json;
std::ifstream f("example.json");
json data = json::parse(f);
```

Створення об’єктів json з літералів JSON. Припустімо, що ви хочете жорстко закодувати це літеральне значення JSON у файлі як об’єкт json:

```
{
  "pi": 3.141,
  "happy": true
}
```

Використання (необроблених) рядкових літералів і json::parse

```
json ex1 = json::parse(R"(
  {
    "pi": 3.141,
    "happy": true
  }
)");
```

Використання визначених користувачем (необроблених) рядкових літералів

```
using namespace nlohmann::literals;
json ex2 = R"(
  {
    "pi": 3.142,
    "flag": true
  }
```

```
)"_json;
```

Використання списків ініціалізаторів

```
json ex3 = {
  {"flag", true},
  {"pi", 3.142},
};
```

JSON як тип даних першого класу. Ось кілька прикладів, щоб зрозуміти, як використовувати клас. Припустімо, необхідно створити об’єкт JSON

```
{
  "pi": 3.14,
  "flag": true,
  "name": "abc",
  "nothing_val": null,
  "var_answer": {
    "everything_var": 42
  },
  "list": [4, 0, 3],
  "object": {
    "currency": "UA",
    "value": 32.89
  }
}
```

За допомогою цієї бібліотеки можно написати. Створити порожню структуру (null)

```
json j;
```

Додати число, яке зберігається як double (зверніть увагу на неявне перетворення j в об'єкт)

```
j["pi"] = 3.141;
```

Додати логічне значення, яке зберігається як bool

```
j["happy"] = true;
```

Додати рядок, який зберігається як std::string

```
j["name"] = "Niels";
```

Додати ще один нульовий об’єкт, передавши nullptr

```
j["nothing"] = nullptr;
```

Додати об'єкт всередині об'єкта

```
j["answer"]["everything"] = 42;
```

Додати масив, який зберігається як std::vector (з використанням списку ініціалізаторів)

```
j["list"] = { 3, 0, 2 };
```

Додати інший об'єкт (використовуючи список пар ініціалізатора)

```
j["object"] = { {"currency", "USD"}, {"value", 42.99} };
```

Замість цього можна написати (що дуже схоже на JSON вище)

```
json j2 = {
  {"pi", 3.141},
  {"happy", true},
  {"name", "Niels"},
  {"nothing", nullptr},
  {"answer", {
    {"everything", 42}
  }},
  {"list", {1, 0, 2}},
  {"object", {
    {"currency", "USD"},
    {"value", 42.99}
  }}
};
```

В усіх цих випадках не потрібно «повідомляти» компілятору, який тип значення JSON використовувати. Якщо необхідно явнио виразити деякі крайові випадки, то використовують функції json::array() і json::object().

Спосіб вираження порожнього масиву []

```
json empty_array  = json::array();
```

Способи вираження порожнього об'єкта {}

```
json empty_object  = json({});
json empty_object  = json::object();
```

Спосіб вираження _масиву_ пар ключ/значення [["currency", "USD"], ["value", 42.99]]

```
json array   = json::array({ {"currency", "USD"}, {"value", 42.99} });
```

Серіалізація / десеріалізація

Ви можете створити значення JSON (десеріалізація), додавши _json до рядкового літералу. Створити об'єкт із рядкового літералу

```
json j = "{ \"flag\": true, \"pi\": 3.142 }"_json;
```

або навіть краще з необробленим рядковим літералом

```
auto j2 = R"(
  {
    "happy": true,
    "pi": 3.141
  }
)"_json;
```

Без додавання суфікса _json переданий рядковий літерал не аналізується, а просто використовується як значення рядка JSON. Тобто json j = "{ \"happy\": true, \"pi\": 3.141 }" просто зберігатиме рядок "{ "happy": true, "pi": 3.141 }", а не розбирати фактичний об'єкт.

Рядковий літерал слід включити в область видимості за допомогою простору імен nlohmann::literals; (див. json::parse()).

Наведений вище приклад також можна виразити явно за допомогою json::parse():

розібрати явно

```
auto j3 = json::parse(R"({"happy": true, "pi": 3.141})");
```

Ви також можете отримати рядкове представлення значення JSON (серіалізувати):

явне перетворення в рядок

```
std::string s = j.dump();    // {"happy":true,"pi":3.141}
```

Серіалізація з форматованим друком. Передати кількість пробілів для відступу

```
std::cout << j.dump(4) << std::endl;
// {
//     "happy": true,
//     "pi": 3.141
// }
```

Зверніть увагу на різницю між серіалізацією та призначенням:

зберегти рядок у значенні JSON

```
json j_string = "this is a string";
```

отримати значення рядка

```
auto cpp_string = j_string.template get<std::string>();
```

отримати значення рядка (альтернатива, коли змінна вже існує)

```
std::string cpp_string2;
j_string.get_to(cpp_string2);
```

отримання серіалізованого значення (явна серіалізація JSON)

```
std::string serialized_string = j_string.dump();
```

вихід оригінального рядка

```
std::cout << cpp_string << " == " << cpp_string2 << " == " << j_string.template get<std::string>() << '\n';
```

вихід серіалізованого значення

```
std::cout << j_string << " == " << serialized_string << std::endl;
```

.dump() повертає початково збережене значення рядка.
Зауважте, що бібліотека підтримує лише UTF-8. Коли ви зберігаєте рядки з різними кодуваннями в бібліотеці, виклик dump() може викликати виключення, якщо json::error_handler_t::replace або json::error_handler_t::ignore не використовуються як обробники помилок.

Можна використовувати потоки для серіалізації та десеріалізації:

десеріалізація зі стандартного введення

```
json j;
std::cin >> j;
```

серіалізація до стандартного виводу

```
std::cout << j;
```

маніпулятор setw було перевантажено, щоб встановити відступ для форматованого друку

```
std::cout << std::setw(4) << j << std::endl;
```

Ці оператори працюють для будь-яких підкласів std::istream або std::ostream. Ось той самий приклад із файлами:

```
// read a JSON file
std::ifstream i("file.json");
json j;
i >> j;
```

записати оброблений JSON в інший файл

```
std::ofstream o("pretty.json");
o << std::setw(4) << j << std::endl;
```

Зверніть увагу, що встановлення біта винятку для failbit є недоречним для цього випадку використання. Це призведе до завершення програми через використання специфікатора noexcept.

SAX interface

Бібліотека використовує SAX-подібний інтерфейс із такими функціями:

Повернене значення кожної функції визначає, чи слід продовжувати аналіз.

Щоб реалізувати власний обробник SAX, виконайте такі дії:

1. Реалізація інтерфейсу SAX у класі. Ви можете використовувати клас nlohmann::json_sax<json> як базовий клас, але ви також можете використовувати будь-який клас, де описані вище функції реалізовані та публічні.

2. Створіть об’єкт вашого класу інтерфейсу SAX, напр. мій_сакс.

3. Виклик bool json::sax_parse(input, &my_sax); де перший параметр може бути будь-яким введенням, як-от рядок або вхідний потік, а другий параметр — вказівник на ваш інтерфейс SAX.

Зауважте, що функція sax_parse повертає лише логічне значення, яке вказує на результат останньої виконаної події SAX. Він не повертає значення json — вам вирішувати, що робити з подіями SAX. Крім того, жодні винятки не створюються у випадку помилки синтаксичного аналізу - ви вирішуєте, що робити з об’єктом винятку, переданим вашій реалізації parse_error. Внутрішньо інтерфейс SAX використовується для аналізатора DOM (клас json_sax_dom_parser), а також для акцептора (json_sax_acceptor), див. файл json_sax.hpp.

Basic usage

Щоб це працювало з одним із ваших типів, вам потрібно надати лише дві функції:

```
using json = nlohmann::json;
namespace ns {
    void to_json(json& j, const person& p) {
        j = json{{"name", p.name}, {"address", p.address},
{"age", p.age}};
    }

    void from_json(const json& j, person& p) {
        j.at("name").get_to(p.name);
        j.at("address").get_to(p.address);
        j.at("age").get_to(p.age);
    }
} // namespace ns
```

Під час виклику конструктора json із вашим типом буде автоматично викликано ваш спеціальний метод to_json. Подібним чином, під час виклику шаблону get<your_type>() або get_to(your_type&), буде викликано метод from_json.

Деякі важливі речі:

• Ці методи ПОВИННІ бути в просторі імен вашого типу (який може бути глобальним простором імен), інакше бібліотека не зможе їх знайти (у цьому прикладі вони знаходяться в просторі імен ns, де визначено person).

• Ці методи ПОВИННІ бути доступними (наприклад, належні заголовки повинні бути включені) всюди, де ви використовуєте ці перетворення. Подивіться на проблему 1108, щоб дізнатися про помилки, які можуть виникнути інакше.

• У разі використання шаблону get<your_type>(), your_type ПОВИНЕН бути Default Constructible. (Існує спосіб обійти цю вимогу, описаний пізніше.)

• У функції from_json використовуйте функцію at() для доступу до значень об’єкта, а не operator[]. Якщо ключ не існує, at створює виняток, який ви можете обробити, тоді як operator[] демонструє невизначену поведінку.

• Вам не потрібно додавати серіалізатори або десеріалізатори для типів STL, таких як std::vector: бібліотека вже реалізує ці.

Integration

json.hpp — єдиний необхідний файл у single_include/nlohmann або опублікований тут. Вам потрібно додати

```
#include <nlohmann/json.hpp>
```

// for convenience

```
using json = nlohmann::json;
```

до файлів, які потрібно обробити JSON, і встановіть необхідні перемикачі, щоб увімкнути C++11 (наприклад, -std=c++11 для GCC і Clang).

Ви також можете використовувати файл include/nlohmann/json_fwd.hpp для форвардних декларацій. Інсталяцію json_fwd.hpp (як частину кроку інсталяції cmake) можна здійснити, встановивши -DJSON_MultipleHeaders=ON.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМИ ВИЯВЛЕННЯ ФРОНТІВ В СИГНАЛІ

3.1. Користувальницький інтерфейс програми

Модуль налаштування та навчання для нейронної мережі виконано з використанням фреймворку Qt. Зовнішний вигляд інтерфейсу програми представлено на рис. 3.1.

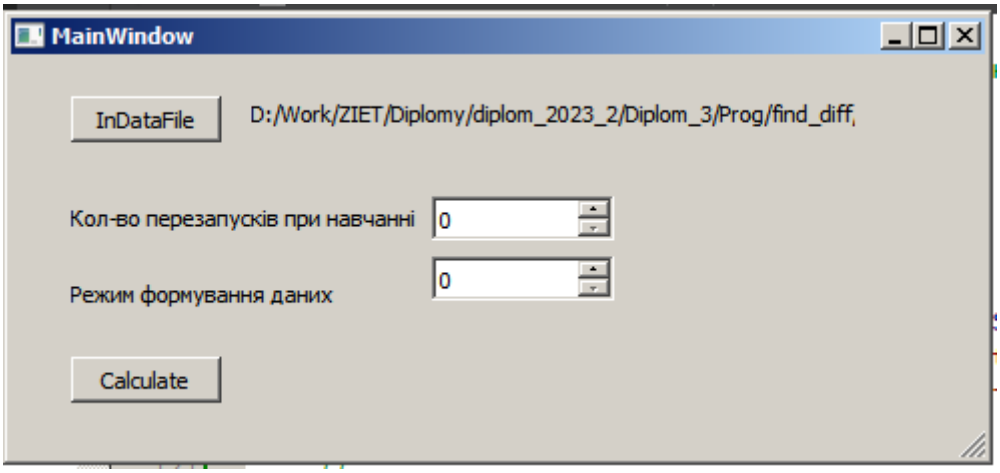

Рис. 3.1. Вигляд діалогу програми

Розглянемо структуру класу головного вікна програми.

```
QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE
class MainWindow : public QMainWindow
{
```

У приватній секції класу вказуємо наявність ресурсів для автоматичної роботи з об’єктами Qt.

```
    Q_OBJECT
```

Загальнодоступны конструктор та деструктор класу

```
public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
```

Слоти для сигналів від елементів форми

```
private slots:
    void on_pushButton_InFile_clicked();
    void on_pushButton_Calc_clicked();
```

Секція захищених елементів даних класу

```
private:
    Ui::MainWindow *ui; // Об’єкти віконного інтерфейсу
    QString file1Name; // Ім’я файлу з даними
    InputData* input_data; // Об’єкт обчислення даних
};
```

Реалізація методів класу головного вікна програми представлено нижче.

Конструктор класу MainWindow забезпечує створення елементів графічного інтерфейсу та стровення об’єкту для розрахунків.

```
MainWindow::MainWindow(QWidget *parent)
```

```
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    input_data = new InputData;
}
```

Деструктор класу видаляє усі динамічно створені об’єкти

```
MainWindow::~MainWindow()
{
    delete ui;
    delete input_data;
}
```

Функція-обробник для кнопки відкриття файлу викликає статичну функцію getOpenFileName класу QFileDialog.

```
void MainWindow::on_pushButton_InFile_clicked()
{
    file1Name = QFileDialog::getOpenFileName(this,
            tr("Open JSON File"), "/home", tr("JSON Files
(*.json)"));
    ui->label_FileName->setText(file1Name);
}
```

Основні обчислення виконуються у функції-обробнику кнопки «Calc».

```
void MainWindow::on_pushButton_Calc_clicked()
{
    input_data->set_mode( ui->spinBox_Mode->value() );
```

```
        input_data->set_cnt_training_iterations(              ui-
>spinBox_CntRestart->value() );
        input_data->LoadData(file1Name);
    }
```

Для читання файлу з даними у json-форматі використовується бібліотека nlohmann.

```
    #include "nlohmann/json.hpp"
    using json = nlohmann::json;
    #include <QMetaType>
    Q_DECLARE_METATYPE(json);
```

Для контрольованого перетворення значень із об’єкту json розроблено функцію контролю наявності об’єкта та його типу

```
    double JsonMy_toDouble(const json& obj, std::string name, long
no_err)
    {
        double res=0.0;
        if( obj.contains(name) )
        {
```

Виконується у разі наявності об’єкту

```
            if(obj[name].is_number())
                res = (double)obj[name];
            else if(obj[name].is_string())
                res = std::stod( (std::string)obj[name] );
        }  else {
```

Виконується у разі відсутності об’єкту або у раз помилки

```
        if(!no_err)
        {
            qWarning()<<"'"<<name.c_str()<<"' not found.";
        }   }
    return res;
}
```

Для спрощення використання функції у програмі використовуються макровизначення наступного виду

```
    double JsonMy_toDouble(const json& obj, std::string name, long
no_err=0);
    #define __JSONMY_SET_DOUBLE(v,o) (v) = JsonMy_toDouble((o),
#v);
    #define __JSONMY_SET_DOUBLE_NOERR(v,o) (v) =
JsonMy_toDouble((o), #v, JSONMY_NOERR);
    #define _JSONMY_SET_DOUBLE(v,o)
__JSONMY_SET_DOUBLE(v,o)
    #define _JSONMY_SET_DOUBLE_NOERR(v,o)
__JSONMY_SET_DOUBLE_NOERR(v,o)
    #define JSONMY_SET_DOUBLE(v,o) _JSONMY_SET_DOUBLE(v,o)
    #define JSONMY_SET_DOUBLE_NOERR(v,o)
_JSONMY_SET_DOUBLE_NOERR(v,o)
```

Аналогічний набір функцій та макровизначень розроблено для інших типів даних long, int.

3.2. Розробка реалізації метода навчання нейронної мережі

Вихідні дані програми представлені у вигляді json файлу. Формат даних наступний

```
{
"train_data":[
{"class":1,          "data":[{"dt":1.2,"Fuel":23,"Speed":34},
{"dt":1.2, "Fuel":23, "Speed":34}]},
{"class":2,   "data":[{"dt":1.2,   "Fuel":23,   "Speed":34},
{"dt":3.2, "Fuel":3, "Speed":54}]}
]
}
```

Блок train_data містить дані для навчання нейронної мережі. Блок є масивом різних навчальних прикладів. В кожному елементі масива міститься клас прикладу у полі class, та масиву даних приклада у полі data. В Усіх прикладах кількість елементів у полях data повинна бути однаковою.

Поле class може приймати наступні значення:

0 - ігнорувати зміну;

1 – позитивний фронт;

2 – негативний фронт.

Для забезпечення роботи алгоритму навчання нейронної мережі необхідні наступні визначення.

```
using namespace alglib;
#define COUNT_DATA_IN_ITEM 3 // Кількість елементів даних у
одному записі
```

```
#define NUM_ITEM_FOR_TRAIN 20 // Кількість записів для навчання
#define NUM_INPUTS (NUM_ITEM_FOR_TRAIN *COUNT_DATA_IN_ITEM) // Кількість входів нейронної мережі. Повинна бути більше або дорівнювати одному
#define NUM_CLASSES 3 //Кількість класів. Повинна бути більше або дорівнювати двом. Використовується наступна схема класів 0 - ігнорувати зміну, 1 – позитивний фронт, 2 – негативний фронт
```

Зчитування даних з файлу виконується у методі LoadData. Зчитані дані розміщуються у об’єкті-векторі

```
QVector<vec_data_t> block_itemdata;
```

Елементи вектора представляють навчальні приклади та мають наступну структуру

```
    typedef struct{
    double dT;
    double Fuel;
    double Speed;
} item_data_t;
typedef struct{
    QVector<item_data_t> data;
    int class_val;
}vec_data_t;
```

Метод зчитування даних виконує спочатки зчитування усіх даних у рядок

```
void InputData::LoadData(QString str)
{
```

```
QFile file(str);
file.open(QFile::ReadOnly);
QString data_txt = QLatin1String(file.readAll());
```

Після зчитування даних виконується синтаксичний аналіз даних у відповідності до структури json

```
load_data = json::parse(data_txt.toStdString(), nullptr, false);
file.close();
```

З об’єкту json виконується зчитування даних у об’єкт block_itemdata наступним чином

```
json train_data = load_data["train_data"];
unsigned int i,j;
for(i=0; i<train_data.size(); i++)
{
    vec_data_t vect_item;
    vect_item.class_val = JsonMy_toLong( train_data[i]["class"], 0 );
    json dat = train_data[i]["data"];
    for(j=0; j<dat.size(); j++)
    {
        item_data_t dd;
        dd.dT = JsonMy_toDouble( dat[j]["dT"], 0 );
        dd.Fuel = JsonMy_toDouble( dat[j]["Fuel"], 0 );
        dd.Speed = JsonMy_toDouble( dat[j]["Speed"], 0 );
        vect_item.data.append(dd);
    }
    block_itemdata.append( vect_item );
}
```

```
}
```

Метод навчання нейронної мережі та зберехення її параметрів представлено нижче

```
void InputData::Calc()
{
   if(block_itemdata.size()==0) return;
```

Навчальний набір представлен масивом, де один рядок відповідає одному запису [A => class(A)]. Класи позначаються цифрами від 0 до 2.

```
        real_2d_array xy;
        xy.setlength(block_itemdata.size(),          1          +
block_itemdata[0].data.size() * COUNT_DATA_IN_ITEM);
```

Навчальні набори можуть формуватися двома способами. В першому випадку набір формується з послідовним розміщенням вхідних сигналів у масиві.

```
        int i,j;
        if( mode == 0)
        {
            for(i=0; i<block_itemdata.size(); i++)
            {
                for(j=0; j<block_itemdata[i].data.size(); j++)
                {
                    xy[i][3*j+0] = block_itemdata[i].data[j].dT;
                    xy[i][3*j+1] = block_itemdata[i].data[j].Fuel;
```

```
                    xy[i][3*j+2]                                    =
block_itemdata[i].data[j].Speed;
                }
                xy[i][3*j]   =   block_itemdata[i].class_val;   //
приналежність до класу
            }
        }
```

У другому випадку усі дані кожного вхідного сигналу розміщуються одним блоком в масиві.

```
        else
        {
            for(i=0; i<block_itemdata.size(); i++)
            {
                for(j=0; j<block_itemdata[i].data.size(); j++)
                {
                    xy[i][j] = block_itemdata[i].data[j].dT;
                }
                for(; j<block_itemdata[i].data.size(); j++)
                {
                    xy[i][j] = block_itemdata[i].data[j].Fuel;
                }
                for(; j<block_itemdata[i].data.size(); j++)
                {
                    xy[i][3*j] = block_itemdata[i].data[j].Speed;
                }
                xy[i][3*j]   =   block_itemdata[i].class_val;   //
приналежність до класу
            }
        }
```

Загальна структура створення нейронної мережі. Оскільки вирішуємо проблему класифікації, то використовуємо функцію mlpcreatec1() замість mlpcreate1(). Ця функція створює мережу класифікаторів із вихідними даними, нормалізованими за методом SOFTMAX. Ця мережа повертає вектор ймовірностей членства у класі, які нормалізовані до невід'ємних значень та мають суму 1.0.

Для створення тренерського об'єкта використовується функція mlpcreatetrainercls(). Набір даних тренерського об'єкта та нейронна мережа обробляється трохи по-різному, щоб врахувати специфіку класифікаційних завдань.

Набір даних прикріплений до тренера. Формат набору даних відрізняється від формату, який використовується для регресії.

```
        mlptrainer trn;
        multilayerperceptron network;
        mlpreport rep;
        mlpcreatetrainercls(NUM_INPUTS, NUM_CLASSES, trn);
        mlpcreatec1(NUM_INPUTS,          cnt_training_iterations,
NUM_CLASSES, network);
        mlpsetdataset(trn, xy, cnt_training_iterations);
```

Мережа навчається з перезапусками cnt_training_iterations з випадкових позицій

```
        mlptrainnetwork(trn,   network,   cnt_training_iterations,
rep);
```

Навчання нейронної мережі виконується на даних, найбільш типові випадки представлені на рис. 3.2-3.1.

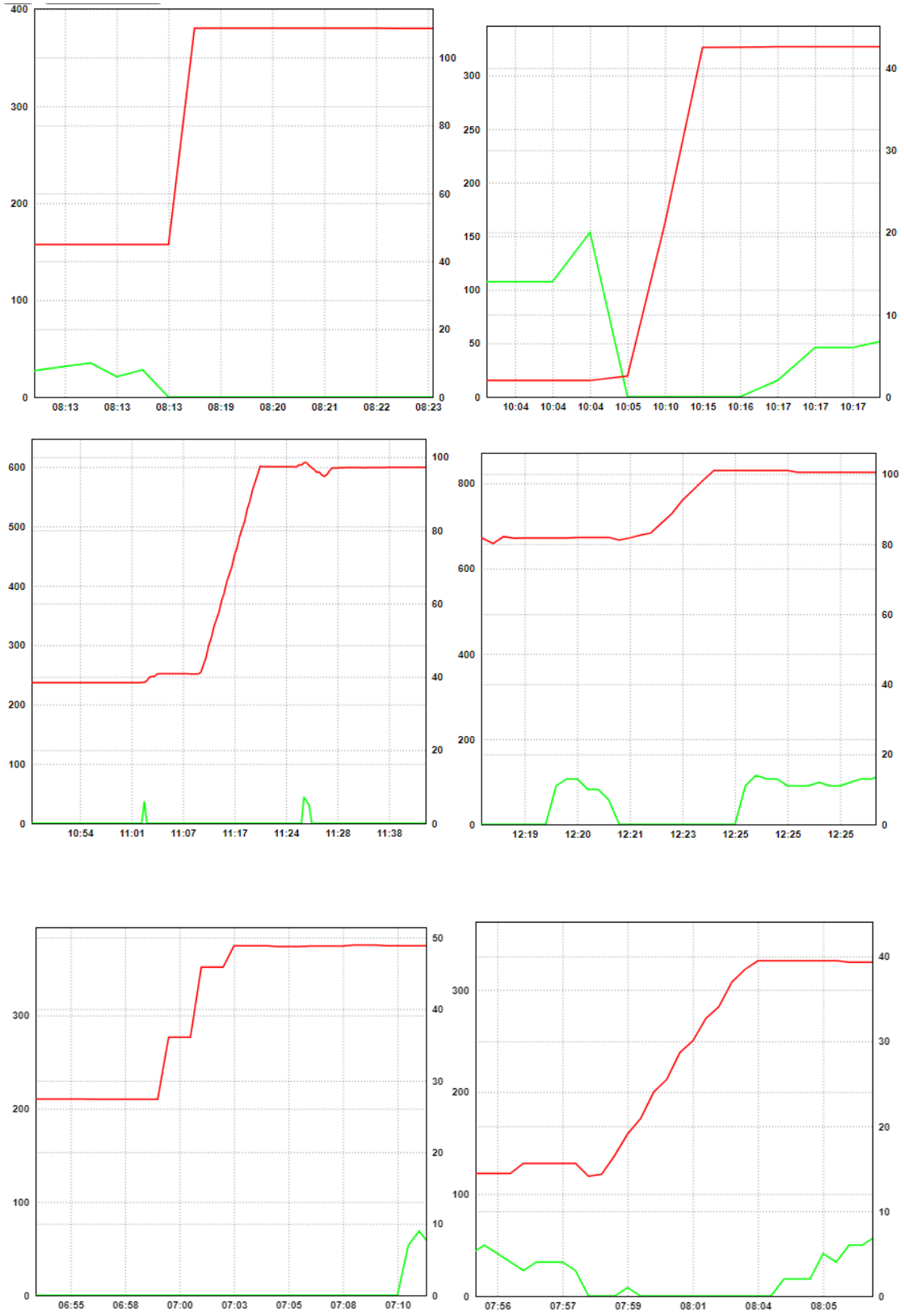

Рис. 3.2. Набір даних для зростаючого фронту сигналу

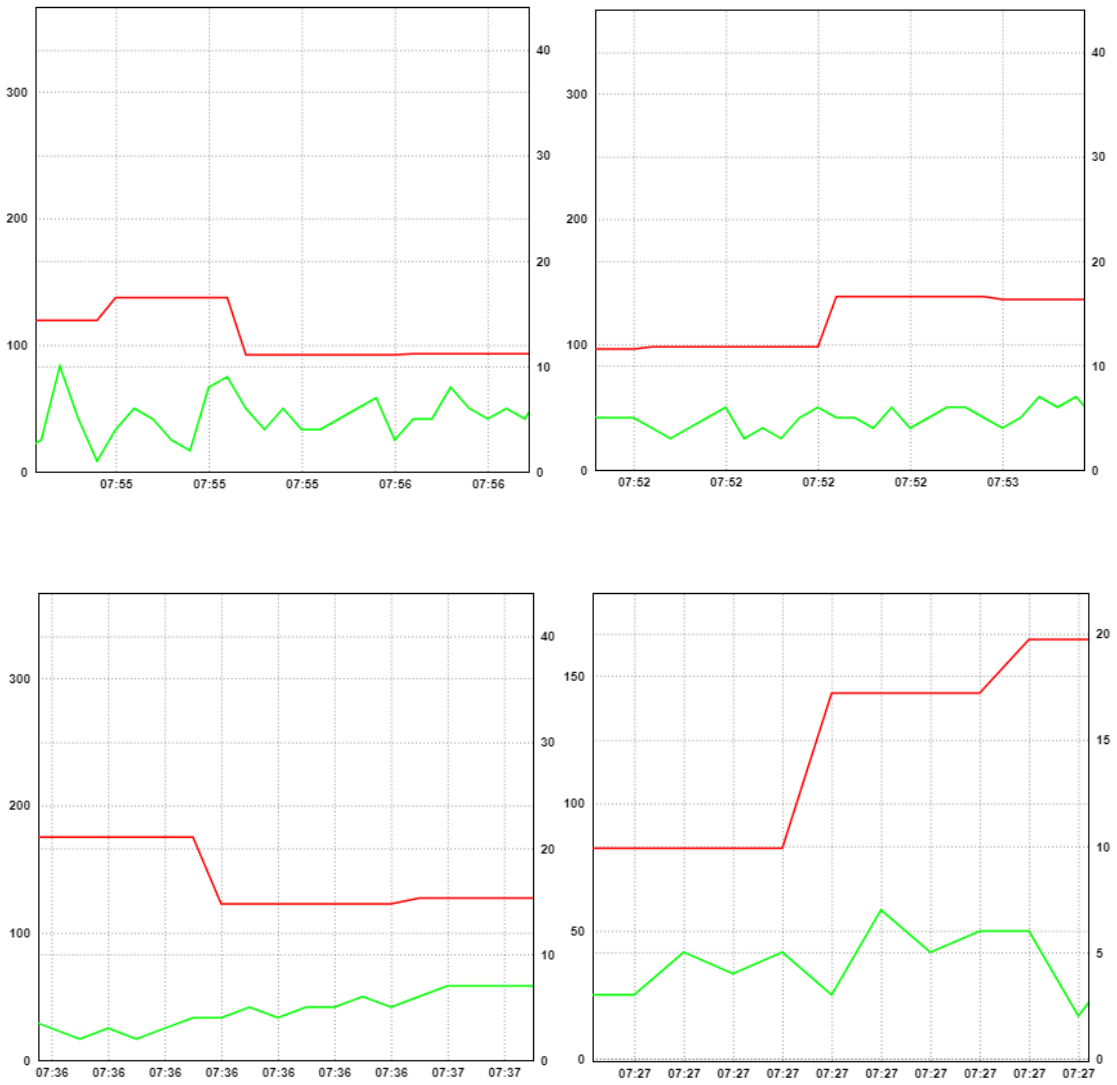

Рис. 3.3. Набір даних з завадами, які не можна розглядати, як зміна сигналу.

Розроблена мережа класифікаторів повертає ймовірність членства у класі замість індексів класів. Мережа повертає три значення (імовірності) замість одного (індекс класу).

Таким чином, для класу 0 очікуємо отримати [P0, P1, P2] = [1, 0, 0], для класу 1 - отримаємо [P0, P1, P2] = [0,1,0], а для класу 2 - отримаємо [P0, P1, P2] = [0, 0, 1].

Наступні властивості гарантуються мережевою архітектурою:

P0> = 0, P1> = 0, P2> = 0 невід'ємність

P0 + P1 + P2 = 1 нормалізація.

ВИСНОВКИ

В ході виконання цієї роботи було проведено аналітичний огляд нейронних мереж, їх загальних принципів побудови та переглянуто різноманітні бібліотеки для реалізації нейронних мереж. Цей огляд дозволив зрозуміти основні концепції та можливості застосування нейромереж у сфері обробки сигналів.

У рамках роботи було розроблено алгоритм виявлення фронтів в сигналі, який базується на використанні математичних класів Alglib та бібліотеки для json-файлів nlohmann. Також був реалізований користувальницький інтерфейс програми для зручного використання розробленого алгоритму.

Окрім цього, була проведена розробка реалізації методу навчання нейронної мережі для виявлення фронтів в сигналі. Цей метод використовує дані, отримані з вхідних сигналів, для побудови моделі, яка здатна точно визначати та класифікувати фронти.

Завершальним етапом було тестування розробленої системи, щоб переконатися у її ефективності та правильності функціонування. Отримані результати підтвердили працездатність розробленої системи для виявлення фронтів сигналів за допомогою нейронних мереж.

Робота підтверджує потенціал та ефективність використання нейронних мереж у сфері обробки сигналів, а отримані результати вказують на успішність розробленої системи для виявлення фронтів сигналів з використанням нейронних мереж.

РЕКОМЕНДАЦІЇ

Для подальшого удосконалення необхідна розробка інтерфейсу програми, що спростить користувачам використання розробленої бібліотеки та удосконалення засобів, спрямованих на виявлення фронтів сигналів.

Під час подальшої роботи необхідно удосконалювати методи навчання нейронних мереж та проводити дослідження впливу параметрів нейронних мереж на точність та швидкість виявлення різких змін у сигналах.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. About ALGLIB [Електронний ресурс]. – Режим доступу: www. URL: https://www.alglib.net/ (дата звернення: 20.09.23)
2. TensorFlow API Versions ALGLIB [Електронний ресурс]. – Режим доступу: www. URL: https://www.tensorflow.org/versions?hl=en (дата звернення: 05.10.23)
3. Theano ALGLIB [Електронний ресурс]. – Режим доступу: www. URL: https://github.com/Theano/Theano (дата звернення: 05.10.23)
4. М.А. Новотарський, Б.Б. Нестеренко. Штучні нейронні мережі: обчислення // Праці Інституту математики НАН України. – Т50. – Київ: Ін-т математики НАН України, 2004. – 408 с.
5. Теорія та практика нейронних мереж : навч. посіб. / Л. М. Добровська, І. А. Добровська. – К. : НТУУ «КПІ» Вид-во «Політехніка», 2015. – 396 с.
6. Штучні нейронні мережі: базові положення : навч. посіб. / І.А. Терейковський, Д.А. Бушуєв, Л.О. Терейковська . – К. : НТУУ «КПІ» Вид-во «Політехніка», 2022. – 128 с.

ДОДАТКИ

ДОДАТОК А

Вихідний код програми mainwindow.cpp