Київський Національний університет імені Тараса Шевченка

М.В. Лавренюк

Алгоритми машинного навчання. Глибокі нейромережі в задачах механіки суцільних середовищ.

Київ 2024

ББК 22.161 + 22.17
УДК 539.3 + 517.95 + 004.8

Лавренюк М.В.

Алгоритми машинного навчання. Глибокі нейромережі в задачах механіки суцільних середовищ: Навчальний посібник. – Київ: КНУ ім. Тараса Шевченка, 2024. – 100 с.

В першій частині посібника розглянуто основні алгоритми машинного навчання: лінійну та логістичну регресію, метод опорних векторів, метод к найближчих сусідів, перцептрон та мультиперцептрон (нейромережі). Друга частина присвячена принципам роботи фізично-поінформованих нейромереж (PINN), зокрема бібліотеці DeepXDE, та автоматичному диференціюванню. В третій частині посібника аналізуються підходи до розв'язання задач механіки суцільних середовищ та рівнянь математичної фізики за допомогою фізично-поінформованих нейромереж.

Для студентів механіко-математичних факультетів університетів.

Рецензенти: *О.С. Лимарченко* — д-р. техн. наук, професор Київського національного університету імені Тараса Шевченка;

І.А.Улітко — канд. фіз.-мат. наук, доцент

Київського національного університету імені Тараса Шевченка

Рекомендовано до друку вченою радою
механіко-математичного факультету
Київського національного університету імені Тараса Шевченка
02.05.2024, протокол № 14

Зміст

Вступ.

Алгоритми машинного навчання відіграють все більш вирішальну роль у розв'язанні самого широкого спектру прикладних задач, таких як медицина, космічна галузь, біологія, автомобілебудування, системи безпеки тощо. Останнім часом набув популярності підхід до розв'язання крайових задач математичної фізики, що базується на поєднанні глибоких нейромереж і математичної фізики – PINN, фізично-поінформовані нейромепежі. Такий підхід дозволяє застосувати універсальний і комплексний підхід до розв'язання задач механіки суцільних середовищ, роблячи можливим уніфікований аналіз достатньо складних з точки зору фізичної постановки задач, для яких інакше потрібно було б застосовувати кожного разу індивідуальні підходи.

У даному посібнику розглядаються найпоширеніші алгоритми машинного навчання, що використовуються для розв'язання прикладних задач, такі як задачі регресії, лінійна регресія, поліноміальна регресія, метод опорних векторів, метод к найближчих сусідів, перцептрон, мультиперцептрон та глибокі нейромережі.

Методи на основі глибоких нейронних мереж − це новітній підхід до розв'язання задач механіки, який використовує штучні нейронні мережі для розв'язання крайових задач математичної фізики. Ці методи є досить новими, проте вони демонструють значний потенціал для аналізу задач з високою точністю і складними геометріями розглядуваних тіл, а також зниження обчислювальних витрат. Особливо перспективним видається підхід на основі глибоких нейромереж, відомий як фізично-поінформовані нейромережі. Цей підхід відноситься до безсіткових методів і використовує відому властивість нейромереж як універсальних апроксиматорів. При розв'язанні задач із використанням цього підходу задача зводиться до задачі оптимізації функції помилок, яка залежить від рівняння, яке описує задачу, геометрії області та початкових і граничних умов.

Ефективний аналіз задач математичної фізики в цілому і задач механіки суцільних середовищ зокрема за допомогою фізично-поінформованих нейромереж вимагає потужних інструментів для швидкої розробки програмного коду для широкого спектру задач – це

можуть бути задачі, що зводяться до звичайних диференціальних рівнянь (прикладом такої задачі є задача Ейлера про розрахунок балки, задачі, що зводяться до диференціальних рівнянь в частинних похідних та їх систем (прикладом такого типу задач є задача про поширення тепла у твердому тілі, що є частинним випадком задачі термопружності - якщо зміна об'єму мала, то система рівнянь термопружності стає незв'язаною і рівняння теплопровідності можна розв'язати окремо). Нарешті, це можуть бути крайові задачі теорії пружності – як плоска так і тривимірна задачі.

Викладення основних аспектів застосування фізично-поінформованих нейромереж, і, більш широко, алгоритмів машинного навчання, які можна використовувати при розв'язанны крайових задач математичної фізики у курсі "Рівняння математичної фізики", та при моделюванні поведінки суцільних середовищ у таких курсах як "Моделі та методи дослідження фізино-нелінійних середовищ", "Теорія пружності", "Чисельні методи задач механіки" є метою цього посібника.

1. Огляд основних алгоритмів машинного навчання

1.1. Задачі класифікації та регресії. Лінійні та нелінійні задачі.

Загальна ідея машинного навчання полягає в тому, щоб отримати модель для вивчення тенденцій на основі попередніх даних на будь-яку тему і мати можливість відтворювати ці тенденції на порівнянних даних у майбутньому. Наступна схема описує основний процес машинного навчання:

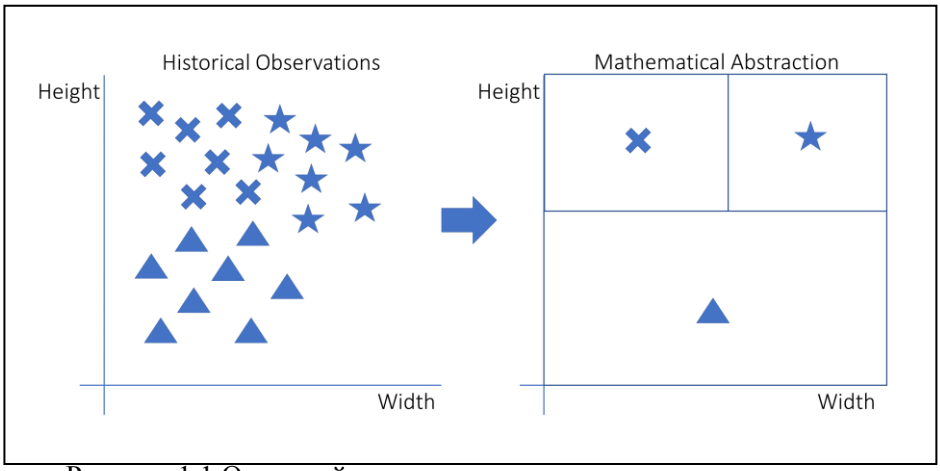


Рисунок 1.1 Основний процес машинного навчання

Рис. 1.1 візуально репрезентує модель машинного навчання, яка накладається на попередні відомі нам дані. Зліва - вихідні спостереження з трьома змінними: висотою, шириною та формою. Фігури - це зірочки, хрестики та трикутники.

Фігури розташовані в різних частинах графіка. Праворуч ми бачимо, як ці початкові спостереження були перетворені в правило прийняття рішень. Для нового спостереження нам потрібно знати ширину і висоту, щоб визначити, в який квадрат воно потрапляє. Квадрат, в який воно потрапляє, в свою чергу, визначає, яку форму воно, найімовірніше, матиме.

Для цієї задачі можна використати багато різних моделей. Модель - це математична формула, яку можна використовувати для опису точок даних. Одним із прикладів є лінійна модель, яка використовує лінійну функцію, що визначається формулою $y = ax + b$.

При підгонці (фітінгу) моделі, ми фактично знаходимо оптимальні значення для фіксованих параметрів за допомогою того чи іншого алгоритму. Для лінійної моделі такими параметрами є *a* і *b*.

Після того, як модель підігнана під дані, вона перетворюється на математичну формулу, в яку ми можете ввести значення незалежних змінних, щоб зробити прогноз для цільової змінної.

Першою визначальною властивістю алгоритмів машинного навчання є поділ на керовані та некеровані моделі (навчання з учителем та без учителя, supervised та unsupervised learning). Різниця між керованими та некерованими моделями полягає в постановці задачі.

У керованих моделях маєvj два типи змінних одночасно:

1) цільова змінна, яку також називають залежною змінною або змінною *y*.

2) незалежні змінні, які також відомі як змінні *x* або пояснювальні змінні.

Цільова змінна - це змінна, яку ми хочемо передбачити. Вона залежить від незалежних змінних і не є відомою заздалегідь. Незалежні змінні - це змінні, які ми знаєте заздалегідь. Ми можете підставити їх у рівняння, щоб передбачити цільову змінну. Таким чином, це відносно схоже на випадок $y = ax + b$.

На попередньому рисунку, і на наступних рисунках, цільовою змінною є форма точки даних, а незалежними змінними - висота і ширина. Ідею керованого навчання можна прослідкувати на рис. 1.2:

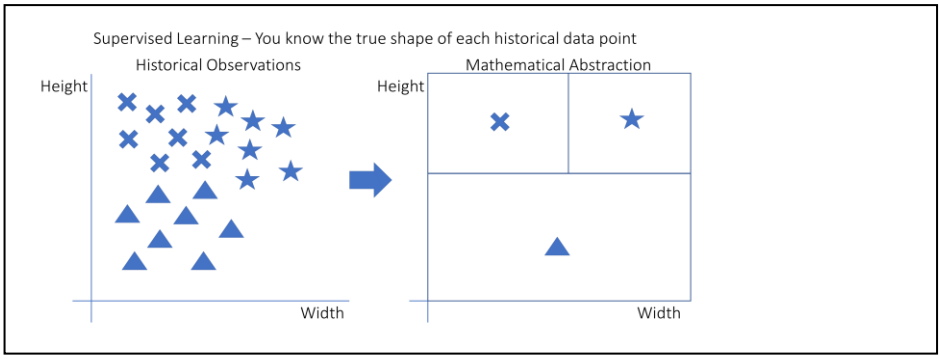


Рисунок 1.2. Типова схема навчання з вчителем

На цьому графіку кожна точка даних має висоту, ширину і форму. Тут є хрестики, зірочки та трикутники. Праворуч - правило прийняття рішень, яке могло б вивчити модель машинного навчання.

У цьому випадку спостереження, позначені хрестиком, є високими, але не широкими. Зірочки - і високі, і широкі. Трикутники - короткі, але можуть бути широкими або вузькими. По суті, модель вивчила правило, яке дозволяє вирішувати, чи є спостереження хрестиком, зірочкою або трикутником, ґрунтуючись лише на його висоті та ширині.

У некерованих моделях немає поділу на цільові та незалежні змінні. Неконтрольоване навчання (без учителя) намагається згрупувати точки даних, оцінюючи їхню схожість.

Як можна бачити по цьому прикладу, ми ніколи не можемо бути впевнені, що згруповані точки даних принципово належать до одного типу, але якщо групування має сенс, воно може бути дуже цінним на практиці. Ідею навчання без вчителя можна побачити на наступному графіку:

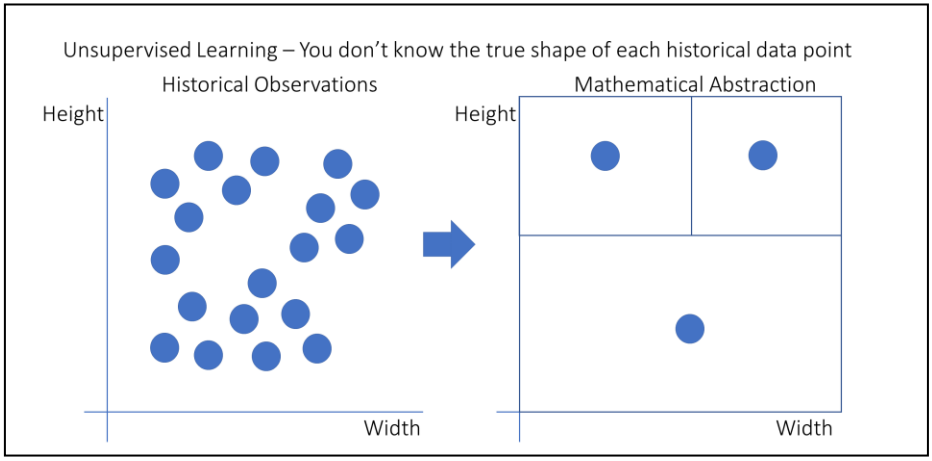


Рисунок 1.3. Типова схема навчання без вчителя

На цьому графіку спостереження більше не мають різної форми. Це все кола. Проте їх все ще можна об'єднати в три групи на основі відстані між точками. У цьому конкретному прикладі є три групи точок, які можна розділити на основі порожнього простору між ними.

Друга властивість, яка має велике значення для алгоритмів машинного навчання, - це те, чи можуть моделі оцінювати нелінійні

взаємозв'язки. Лінійні моделі - це моделі, які роблять передбачення за допомогою ліній або гіперплощин. На рис. ___ модель зображена у вигляді лінії, проведеної між точками. Модель $y = ax + b$ є класичним прикладом лінійної моделі. На наступному схематичному малюнку можна побачити, як лінійна модель може узгоджуватися із даними:

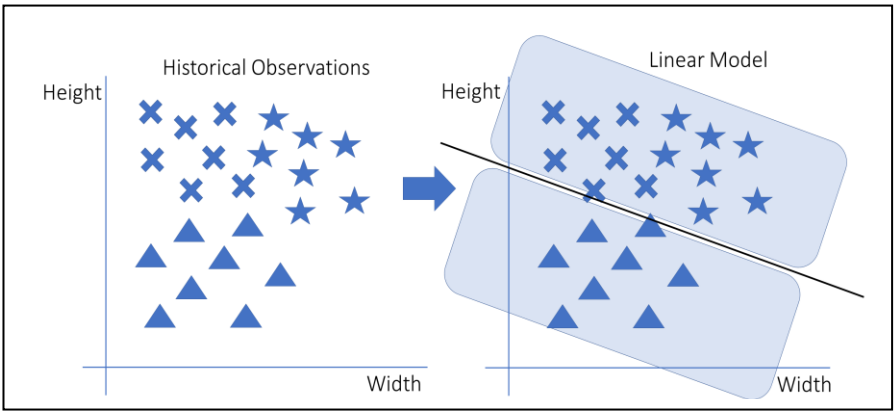


Рисунок 1.4. Узгодження лінійної моделі із даними

На цьому малюнку точки даних зображені зліва зірочками, трикутниками та хрестиками. Праворуч - лінійна модель, яка може відокремити трикутники від нетрикутників. Рішення - це лінія. Кожна точка над лінією є нетрикутником, а все, що нижче лінії - трикутником.

Якби ми захотіли додати ще одну незалежну змінну до попереднього графіка, вам довелося б намалювати її як додатковий вимір, створивши таким чином куб з фігурами всередині нього. Але лінія не зможе розрізати куб на дві частини. Багатовимірним аналогом лінії є гіперплощина. Отже, лінійна модель представлена гіперплощиною, яка у випадку двовимірного простору є лінією.

Нелінійні моделі - це моделі, які використовують будь-який інший підхід, окрім лінії, для розділення своїх випадків. Відомим прикладом є дерево рішень, яке, по суті, є довгим списком операторів if ... else. На нелінійному графіку оператори if ... else дозволять малювати квадрати або будь-яку іншу форму, яку ми захочемо намалювати.

На наступному графіку зображено нелінійну модель, застосовану до даних прикладу:

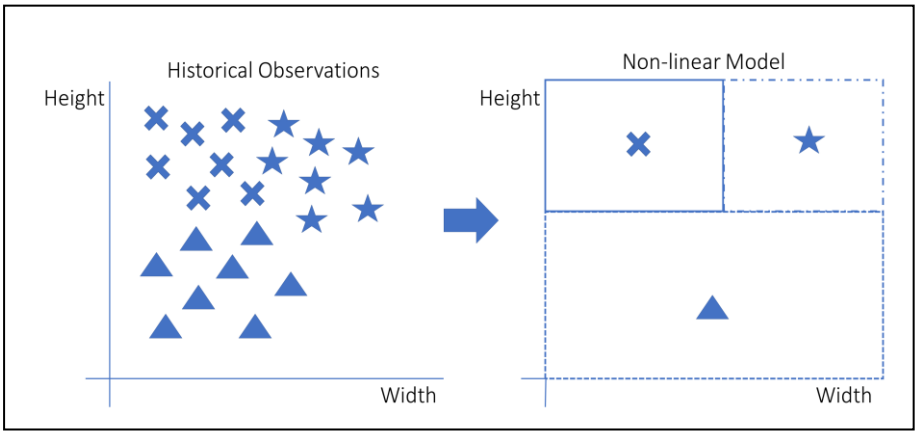


Рисунок 1.5. Нелінійна модель, застосована до класифікації на три класи

Цей графік показує, як рішення може бути нелінійним. Правило прийняття рішення складається з трьох квадратів. Квадрат, в який потрапляє нова точка даних, визначає її прогнозовану форму. Зауважимо, що неможливо зробити таку підгонку за допомогою лише однієї лінії: Потрібно дві лінії. Цю модель можна відтворити за допомогою операторів якщо ... інакше наступним чином:

Якщо висота, яка відповідає нашій точці даних, мала, то це трикутник. Інакше, якщо ширина , яка відповідає нашій точці даних, мала, то це хрестик. Інакше, якщо жодна з вищезазначених умов не відповідає дійсності, то це зірочка.

Алгоритми класичного машинного навчання з учителем можна розділити на дві групи залежно від типу цільової змінної, яку вони можуть передбачити:

Задачі класифікації - це задачі прогнозування з категоріальною цільовою змінною. Класифікаційні моделі вчаться класифікувати будь-яке нове спостереження. Присвоєний клас може бути або правильним, або неправильним, але не проміжним. Класичним прикладом класифікації є набір даних про ірис (iris dataset), в якому ви використовуєте фізичні виміри рослин, щоб передбачити їхній вид. Відомим алгоритмом, який можна використовувати для класифікації, є логістична регресія.

Задачі регресія - це задачі прогнозування, в якій цільова змінна є числовою. Відомим прикладом регресії є Housing Prices Challenge на Kaggle. У цьому змаганні з машинного навчання учасники намагаються передбачити ціни продажу будинків на основі числових незалежних змінних.

На наступному графіку можна побачити, як виглядатимуть регресія і класифікація на основі попереднього прикладу:

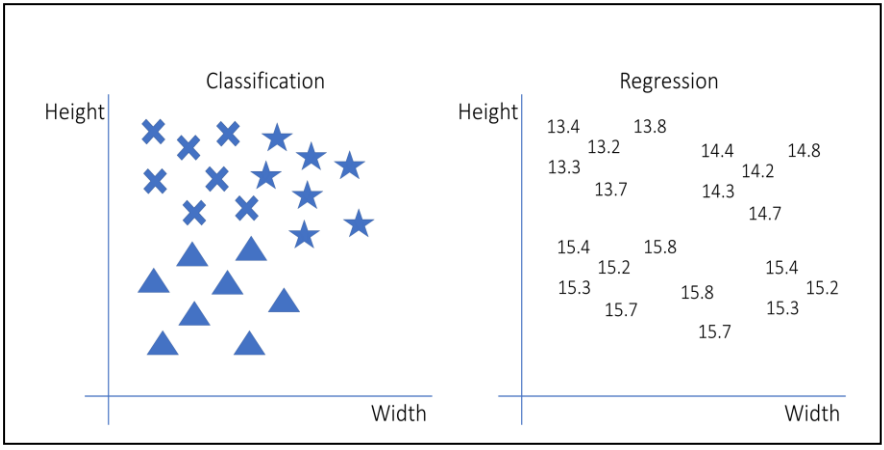


Рисунок 1.6. Порівняння результатів робрти регресійної і класифікаційної моделей

Ліва частина цього зображення - це класифікація. Цільовою змінною є форма спостереження, яка є категоріальною змінною. Права частина - це регресія. Цільова змінна є числовою. Тобто, правила прийняття рішень можуть бути абсолютно однаковими для цих двох прикладів, але їх інтерпретація відрізняється.

Для одного прогнозу класифікації є або правильними, або неправильними, тоді як регресії мають похибку на неперервній шкалі. Наявність числової міри помилки є більш практичною, тому багато моделей класифікації передбачають не лише клас, але й ймовірність потрапляння до будь-якого з класів. Деякі моделі можуть виконувати лише регресію, деякі - лише класифікацію, а деякі - і те, і інше.

В якості третього критерію для характеристики моделей машинного навчання потрібно враховувати складність моделі. Машинне навчання, а особливо штучний інтелект, зараз перебуває на стадії розквіту і

використовується для вирішення багатьох складних задач, таких як розпізнавання тексту, зображень і мови, або для самокерованих автомобілів.

Більш просунуті та складні моделі, такі як нейронні мережі, ймовірно, можуть навчитися всьому, чому може навчитись більш проста модель, наприклад, модель k-найближчих сусідів. Зрештою, ці просунуті моделі дуже добре навчаються. Однак ця складність також має свою ціну. Для того, щоб зробити моделі придатними для нашого прогнозу, нам, як правило, доведеться витратити набагато більше часу на їх розробку.

Окрім цього, нам також знадобиться набагато більше даних для більш складної моделі, а дані не завжди доступні. І останнє, але не менш важливе: складніші моделі складніше інтерпретувати нам, людям, а іноді ця інтерпретація може бути дуже цінною.

Існує кілька способів оцінювання моделей, але найпоширеніший з них - це розбиття набору даних на тренувальний і тестувальний набори. При використанні такого розбиття для оцінки моделі ми розбиваємо набір даних на дві частини:

1) навчальні дані використовуються для підбору (фітінгу) моделі.

2) тестові дані використовуються для оцінки моделі. Це означає, що ми будемо робити прогнози щодо значень змінної у у тестових даних і порівнювати ці результати з відомим істинним значенням цієї змінної (часто це значення називається ground truth).

1.2. Задачі регресії. Лінійна регресія. Поліноміальна регресія.

Лінійна регресія.

Якщо коротко, то задача регресії полягає у відшуканні взаємозв'язків між змінними. Наприклад, ви можете спостерігати за кількома працівниками певної компанії і спробувати зрозуміти, як їхні зарплати залежать від їхніх характеристик, таких як досвід, рівень освіти, посада, місто, в якому вони працюють, і так далі.

Це задача регресії, в якій дані, що стосуються кожного працівника, представляють одне спостереження. Передбачається, що досвід, освіта, посада та місто є незалежними характеристиками, а заробітна плата залежить від них.

Аналогічно можна спробувати встановити математичну залежність цін на житло від площі, кількості кімнат, відстані до центру міста тощо.

Загалом, у регресійному аналізі ми розглядаємо певне явище, яке нас цікавить, маючи при цьому ряд спостережень. Кожне спостереження має дві або більше ознак (фіч). Виходячи з припущення, що принаймні одна з ознак залежить від інших, намагаються встановити зв'язок між ними.

Іншими словами, потрібно знайти функцію, яка досить добре відображає одні ознаки або змінні на інші.

Залежні ознаки називаються залежними змінними, результатами або відповідями. Незалежні ознаки називаються незалежними змінними, входами, регресорами або предикторами.

Регресійні задачі зазвичай мають одну неперервну і незв’язану залежну змінну. Вхідні дані, однак, можуть бути неперервними, дискретними або навіть категоріальними, такими як стать, національність або бренд.

Зазвичай прийнято позначати виходи через y, а входи через x. Якщо є дві або більше незалежних змінних, то їх можна представити у вигляді вектора $\mathbf{x} = (x_1, ..., x_r)$, де r - кількість входів.

Зазвичай регресія потрібна, щоб відповісти на запитання, чи впливає одне явище на інше і як саме, або як пов'язані між собою кілька змінних. Наприклад, можна використовувати регресію, щоб визначити, чи впливає досвід або стать на заробітну плату, і якщо так, то в якій мірі.

Регресія також корисна, коли потрібно спрогнозувати реакцію за допомогою нового набору предикторів. Наприклад, можна спробувати спрогнозувати споживання електроенергії домогосподарством на наступну годину, враховуючи зовнішню температуру, час доби та кількість мешканців у цьому домогосподарстві.

Регресія використовується в багатьох різних галузях, включаючи економіку, комп'ютерні науки та соціальні науки. Її важливість зростає з кожним днем завдяки доступності великих обсягів даних та підвищенню обізнаності про практичну цінність даних.

Лінійна регресія, ймовірно, є одним з найбільш важливих і широко використовуваних методів регресії. Разом з тим, що не менш важливо, це також і один з найпростіших методів регресії, зокрема, однією з її основних переваг є простота інтерпретації результатів регресійного аналізу.

При реалізації лінійної регресії деякої залежної змінної y від набору незалежних змінних $\mathbf{x} = (x_1, ..., x_r)$, де r - кількість предикторів, припускається лінійний зв'язок між y та $\mathbf{x}$: $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_r x_r + \varepsilon$. Це рівняння називається регресійним рівнянням. $\beta_0, \beta_1, ..., \beta_r$ - коефіцієнти регресії, а ε - випадкова похибка (спричинена неточністю вимірювань вхідних даних).

Лінійна регресія обчислює оцінки коефіцієнтів регресії або, що те саме, прогнозовані ваги, позначені через $b_0, b_1, ..., b_r$. Ці оцінки визначають прогнозовану функцію регресії $f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$. Ця функція повинна досить добре відображати залежність між входами та виходами.

Прогнозований відгук, $f(\mathbf{x}_i)$, для кожного спостереження $i = 1, ..., n$, повинен бути якомога ближчим до відповідного фактичного відгуку y_i. Різниці $y_i - f(\mathbf{x}_i)$ для всіх спостережень $i = 1, ..., n$, називаються залишками. Задача регресії полягає у у визначенні найкращих прогнозованих ваг, тобто ваг, що відповідають найменшим залишкам.

Щоб отримати найкращі ваги, зазвичай мінімізують суму квадратів залишків (SSR) для всіх спостережень $i = 1, ..., n$: $\text{SSR} = \Sigma_i(y_i - f(\mathbf{x}_i))^2$. Такий підхід називається методом найменших квадратів.

Варіація фактичних відгуків y_i, $i = 1, ..., n$, відбувається частково через залежність від предикторів $\mathbf{x}_i$. Однак існує також додаткова

власна дисперсія результату.

Коефіцієнт детермінації, позначений як R^2, показує, яку частину варіації y можна пояснити залежністю від **x**, використовуючи ту чи іншу регресійну модель. Більший R^2 вказує на кращу відповідність і означає, що модель може краще пояснити варіацію вихідної змінної при різних вхідних даних.

Значення $R^2 = 1$ відповідає SSR = 0, що є ідеальною відповідністю, оскільки значення прогнозованих і фактичних відгуків повністю збігаються.

Розглянемо спочатку найпростішу модель **простої лінійної регресії**.

Проста або одновимірна лінійна регресія є найпростішим випадком лінійної регресії, оскільки вона має одну незалежну змінну, $\mathbf{x} = x$.

Наступний рисунок ілюструє просту лінійну регресію:

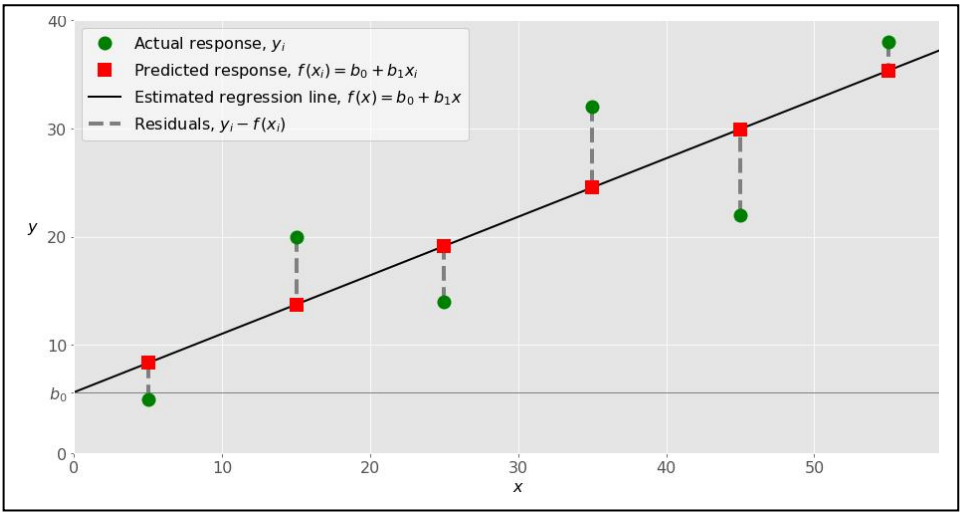


Рисунок 1.7. Результат використання моделі простої лінійної регресії.

При реалізації алгоритму простої лінійної регресії, зазвичай починають з заданого набору пар вхід-вихід (x-y). Ці пари є нашими спостереженнями, показаними на рисунку зеленими колами. Наприклад, крайнє ліве спостереження має вхід $x = 5$ і фактичний вихід, або реакцію, $y = 5$. Наступне спостереження має $x = 15$ і $y = 20$, і так далі.

Прогнозована функція регресії, представлена чорною лінією, має рівняння $f(x) = b_0 + b_1 x$. Наша мета - обчислити оптимальні значення

прогнозних ваг b_0 та b_1, які мінімізують SSR, та визначити прогнозовану функцію регресії.

Значення b_0, яке також називають перехопленням, показує точку, де прогнозована лінія регресії перетинає вісь y. Це значення прогнозованого відгуку $f(x)$ для $x = 0$. Значення b_1 визначає нахил прогнозованої лінії регресії (фактично це тангенс кута нахилу прямої регресії до осі фіч).

Прогнозовані реакції, показані у вигляді червоних квадратів, є точками на лінії регресії, які відповідають вхідним значенням. Наприклад, для вхідних даних $x = 5$, прогнозований відгук становить $f(5) = 8{,}33$, що відображає крайній лівий червоний квадрат.

Вертикальні пунктирні сірі лінії представляють залишки, які можна обчислити як $y_i - f(\mathbf{x}_i) = y_i - b_0 - b_1 x_i$ для $i = 1, \ldots, n$. Це відстані між зеленими колами та червоними квадратами. Коли ми застосовуємо лінійну регресію, ми по суті намагаємося мінімізувати ці відстані і зробити червоні квадрати якомога ближче до заданих зелених кіл.

Множинна лінійна регресія.

Множинна або багатовимірна лінійна регресія - це випадок лінійної регресії з двома або більше незалежними змінними.

Якщо є лише дві незалежні змінні, то функція регресії має вигляд $f(x_1, x_2) = b_0 + b_1 x_1 + b_2 x_2$. Це рівняння описує площину регресії у тривимірному просторі. Метою регресії є визначення значень вагових коефіцієнтів b_0, b_1 та b_2 таким чином, щоб ця площина була якомога ближчою до фактичних значень реакції, забезпечуючи при цьому мінімальний SSR.

Випадок з більш ніж двома незалежними змінними є подібним до простої лінійної регресії, але більш загальним. Функція регресії в цьому випадку має вигляд $f(x_1, \ldots, x_r) = b_0 + b_1 x_1 + \cdots + b_r x_r$, і містить $r + 1$ вагових коефіцієнтів, які необхідно визначити, при цьому кількість вхідних даних становить r.

Поліноміальна регресія.

Поліноміальну регресію можна розглядати як узагальнений випадок лінійної регресії, коли залежність між виходом і входами вважається поліноміальною, як і, відповідно, оцінка функції регресії.

Іншими словами, на додаток до лінійних членів, таких як b_1x_1, функція регресії f може включати нелінійні члени, такі як $b_2{x_1}^2$, $b_3{x_1}^3$ або навіть $b_4x_1x_2$, $b_5{x_1}^2x_2$.

Найпростіший приклад поліноміальної регресії має лише одну незалежну змінну, а функція регресії є квадратним тричленом: $f(x) = b_0 + b_1x + b_2x^2$.

Таким чином, у цьому випадку потрібно обчислити b_0, b_1 і b_2 (які є нашими невідомими), щоб мінімізувати SSR.

Порівняємо тепер цю функцію регресії з функцією $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2$, яка використовується для множинної лінійної регресії (випадок двох вхідних даних). Вони виглядають дуже схожими і є лінійними функціями відносно невідомих b_0, b_1 та b_2. Тому можливо розв'язати задачу поліноміальної регресії як лінійну задачу з членом x^2, що розглядається як вхідна змінна.

У випадку двох змінних і полінома другого степеня функція регресії має такий вигляд $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2 + b_3{x_1}^2 + b_4x_1x_2 + b_5{x_2}^2$.

Процедура розв'язання задачі ідентична до попереднього випадку. Застосовується лінійна регресія для п'яти вхідних даних: x_1, x_2, ${x_1}^2$, x_1x_2 та ${x_2}^2$. В результаті регресії ми одержуємо значення шести ваг, які мінімізують SSR: b_0, b_1, b_2, b_3, b_4 та b_5.

Метод градієнтного спуску.

Задача знаходження коефіцієнтів регресійної функції є задачею оптимізації, оскільки зводиться до пошуку мінімального значення суми квадратів залишків по всіх точках спостереження, і тому, при її розв’язанні зазвичай використовується метод градієнтного спуску та його різновиди і модифікації, як, наприклад, стохастичний метод градієнтного спуску, чи метод адаптивних моментів (ADAM).

Загалом, алгоритм градієнтного спуску - це наближений та ітераційний метод математичної оптимізації, і його можна використовувати для знаходження наближеного значення мінімуму будь-якої диференційованої функції.

Функція втрат або **функція витрат** - це функція, яку потрібно мінімізувати (або максимізувати), змінюючи цільові змінні. Багато методів машинного навчання призводять до необхідності розв’язання задач оптимізації. Вони намагаються мінімізувати різницю між

фактичними і прогнозованими результатами шляхом налаштування параметрів моделі (наприклад, ваг і зсувів для нейронних мереж, правил прийняття рішень для випадкового лісу або градієнтного бустінгу і так далі).

Для задачі регресії наша мета - мінімізувати різницю між прогнозом $f(\mathbf{x})$ та фактичними даними y (тобто, залишок). Тобто ми хочемо мінімізувати суму квадратів залишків (SSR), де SSR = $\Sigma_i(y_i - f(\mathbf{x}_i))^2$ для всіх спостережень $i = 1, ..., n$, де n - це загальна кількість спостережень. Замість SSR можна використовувати середню квадратичну похибку (MSE = SSR / n).

І SSR, і MSE використовують квадрат різниці між фактичними та прогнозованими результатами. Чим менша різниця, тим точніший прогноз. Нульова різниця означає, що прогноз дорівнює фактичним даним.

SSR або MSE мінімізується шляхом налаштування параметрів моделі. У лінійній регресії потрібно знайти функцію $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$, для цього потрібно визначити ваги b_0, b_1, ..., b_r, які мінімізують SSR або MSE.

У математиці похідна функції показує, наскільки змінюється значення функції при зміні її аргументу (або аргументів). Похідні важливі для оптимізації, оскільки нульові похідні можуть вказувати на мінімум, максимум або сідлову точку.

Градієнт функції C від декількох незалежних змінних v_1, ..., v_r позначається через $\nabla C(v_1, ..., v_r)$ і визначається як векторна функція частинних похідних C щодо кожної незалежної змінної: $\nabla C = (\partial C/\partial v_1, ..., \partial C/v_r)$. Ненульове значення градієнта функції C в даній точці визначає напрямок і швидкість найшвидшого зростання C. При роботі з градієнтним спуском вас цікавить напрямок найшвидшого зменшення функції втрат. Цей напрямок визначається від'ємним градієнтом $-\nabla C$.

Ідея градієнтного спуску наступна: починаємо з довільно обраної позиції точки або вектора $\mathbf{v} = (v_1, ..., v_r)$ і ітеративно переміщуємо її в напрямку найшвидшого зменшення функції втрат. А це напрямок вектора від'ємного градієнта, $-\nabla C$.

Отримавши випадкову початкову точку $\mathbf{v} = (v_1, ..., v_r)$, оновлюємо її або переміщуємо в нову позицію в напрямку від'ємного градієнта: $\mathbf{v} \rightarrow \mathbf{v}$

$-\eta\nabla C$, де η - це невелике додатне значення, яке називається **швидкістю навчання.**

Швидкість навчання визначає, наскільки великим є крок оновлення або переміщення. Це дуже важливий параметр. Якщо η занадто мале, то алгоритм може збігатися дуже повільно. Великі значення η також можуть спричинити проблеми зі збіжністю або взагалі зробити алгоритм розбіжним.

1.3. Задачі класифікації. Логістична регресія.

Алгоритми керованого машинного навчання (з учителем) визначають моделі, які фіксують взаємозв'язки між даними. Класифікація - це область керованого машинного навчання, в якій намагаються передбачити, до якого класу або категорії належить певний об'єкт на основі його характеристик.

Наприклад, можна проаналізувати працівників якоїсь компанії і спробувати встановити залежність між ними за певними ознаками або змінними, такими як рівень освіти, кількість років на поточній посаді, вік, заробітна плата, шанси на підвищення і так далі. Набір даних, що стосуються одного працівника, є одним спостереженням. Характеристики або змінні можуть мати одну з двох форм, або залежних (виходи, реакції), або незалежних змінних (входи, предиктори).

У наведеному вище прикладі, коли ми аналізуємо працівників, можна припустити, що рівень освіти, час перебування на поточній посаді та вік є незалежними факторами, і розглядати їх як вхідні дані. Зарплата і шанси на просування по службі можуть бути виходами, які залежать від входів.

По виду залежних змінних розрізняють задачі регресії та класифікації. В той час як задачі регресії мають неперервні і, як правило, незв’язані результати (наприклад, ми оцінюємо заробітну плату як функцію досвіду та рівня освіти), задачі класифікації мають дискретні та зв’язані результати, які називаються класами або категоріями. Наприклад, передбачення того, чи отримає працівник підвищення чи ні (істинне чи хибне) є задачею класифікації.

Існує два основних типи задач класифікації:

1) бінарна або біноміальна класифікація: є рівно два класи, поміж яких треба зробити вибір (зазвичай 0 і 1, істина і хибність, або позитивне і негативне);

2) багатокласова або мультиноміальна класифікація: три або більше класів результатів, поміж яких треба зробити вибір.

Коли потрібно застосовувати алгоритми класифікації? Класифікацію можна застосовувати в багатьох галузях науки і техніки. Наприклад, алгоритми класифікації тексту використовуються для відокремлення

нормальних листів від спаму, а також позитивних і негативних коментарів.

Логістична регресія є одним із фундаментальних методів класифікації. Вона належить до групи лінійних класифікаторів і дещо схожа на поліноміальну та лінійну регресію. Логістична регресія є швидкою і відносно нескладною, інтерпретувати її результати досить зручно. Хоча це, по суті, метод бінарної класифікації, його також можна застосовувати до багатокласових задач.

Логістична регресія є лінійним класифікатором, тому будемо використовувати лінійну функцію $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$, яку також називають логітом. Змінні b_0, b_1, ..., b_r є оцінками коефіцієнтів регресії, які також називають прогнозованими вагами або просто коефіцієнтами.

Функція логістичної регресії $p(\mathbf{x})$ є сигмоїдною функцією $f(\mathbf{x})$: $p(\mathbf{x}) = 1 / (1 + \exp(-f(\mathbf{x})))$. Як така, вона часто близька до 0 або 1. Функція $p(\mathbf{x})$ часто інтерпретується як прогнозована ймовірність того, що вихід для заданого $\mathbf{x}$ дорівнює 1. Отже, $1 - p(x)$ - це ймовірність того, що вихід дорівнює 0.

Логістична регресія визначає найкращі прогнозовані ваги b_0, b_1, ..., b_r такі, що функція $p(\mathbf{x})$ є максимально наближеною до всіх фактичних відповідей y_i, $i = 1, ..., n$, де n - кількість спостережень. Процес обчислення найкращих ваг на основі наявних спостережень називається навчанням моделі або підгонкою (фітінг).

Щоб отримати найкращі ваги, зазвичай максимізують **функцію правдоподібності** (LLF) для всіх спостережень $i = 1, ..., n$. Цей метод називається оцінкою максимальної правдоподібності і описується рівнянням $\text{LLF} = \Sigma_i(y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i)))$.

Коли $y_i = 0$, LLF для відповідного спостереження дорівнює $\log(1 - p(\mathbf{x}_i))$. Якщо $p(\mathbf{x}_i)$ близьке до $y_i = 0$, то $\log(1 - p(\mathbf{x}_i))$ близьке до 0. Це і є бажаний результат. Якщо $p(\mathbf{x}_i)$ далеко від 0, то $\log(1 - p(\mathbf{x}_i))$ значно зменшується. Наша ж мета - отримати максимальний LLF. Аналогічно, коли $y_i = 1$, LLF для цього спостереження дорівнює $y_i \log(p(\mathbf{x}_i))$. Якщо $p(\mathbf{x}_i)$ близьке до $y_i = 1$, то $\log(p(\mathbf{x}_i))$ близьке до 0. Якщо $p(\mathbf{x}_i)$ далеке від 1, то $\log(p(\mathbf{x}_i))$ - велике від'ємне число.

Визначивши найкращі ваги, які визначають функцію $p(\mathbf{x})$, можна отримати прогнозовані результати $p(\mathbf{x}_i)$ для будь-якого заданого входу $\mathbf{x}_i$.

Для кожного спостереження i = 1, ..., n прогнозований вихід дорівнює 1, якщо $p(\mathbf{x}_i)$ > 0.5, і 0 в іншому випадку. Поріг не обов'язково має дорівнювати 0.5, але зазвичай він дорівнює саме 0.5. Можна визначити менше або більше значення, якщо це зручніше для даної ситуації.

Між $p(\mathbf{x})$ і $f(\mathbf{x})$ існує ще одна важлива залежність, яка полягає в тому, що $\log(p(\mathbf{x}) / (1 - p(\mathbf{x})) = f(\mathbf{x})$. Ця рівність пояснює, чому $f(\mathbf{x})$ є логітом. Вона означає, що $p(\mathbf{x}) = 0.5$, коли $f(\mathbf{x}) = 0$, і що прогнозований вихід дорівнює 1, якщо $f(\mathbf{x}) > 0$ і 0 в іншому випадку.

Бінарна класифікація має чотири можливі типи результатів:
1) істинно негативні: правильно передбачені негативні (нулі);
2) хибнопозитивні: правильно передбачені позитивні результати (одиниці);
3) хибнонегативні: неправильно передбачені негативні результати (нулі);
4) хибнопозитивні: неправильно передбачені позитивні результати (одиниці).

Зазвичай оцінюється продуктивність класифікатора, порівнюючи фактичні та прогнозовані результати і підраховуючи правильні та неправильні прогнози. Найбільш простим показником точності класифікації є відношення кількості правильних прогнозів до загальної кількості прогнозів (або спостережень). Серед інших показників бінарних класифікаторів можна виділити наступні:

1) позитивна прогностична цінність - це відношення кількості істинно-позитивних результатів до суми істинних і хибно-позитивних результатів.

2) негативна прогностична цінність - це відношення кількості істинно-негативних результатів до суми істинно-негативних і хибно-негативних результатів.

3) чутливість (також відома як частота пригадування або істинно-позитивна частота) - це відношення кількості істинно-позитивних результатів до кількості фактичних позитивних результатів.

4) специфічність (або частота хибнонегативних результатів) - це відношення кількості хибнонегативних результатів до кількості реальних негативних результатів.

Найбільш підходящий показник залежить від конкретної розглядуваної задачі. Ми будемо використовувати найпростіший

показник точності класифікації.

Одновимірна логістична регресія є найпростішим випадком логістичної регресії. Існує лише одна незалежна змінна (або ознака), тобто **x** = x. Цей рисунок ілюструє одновимірну логістичну регресію:

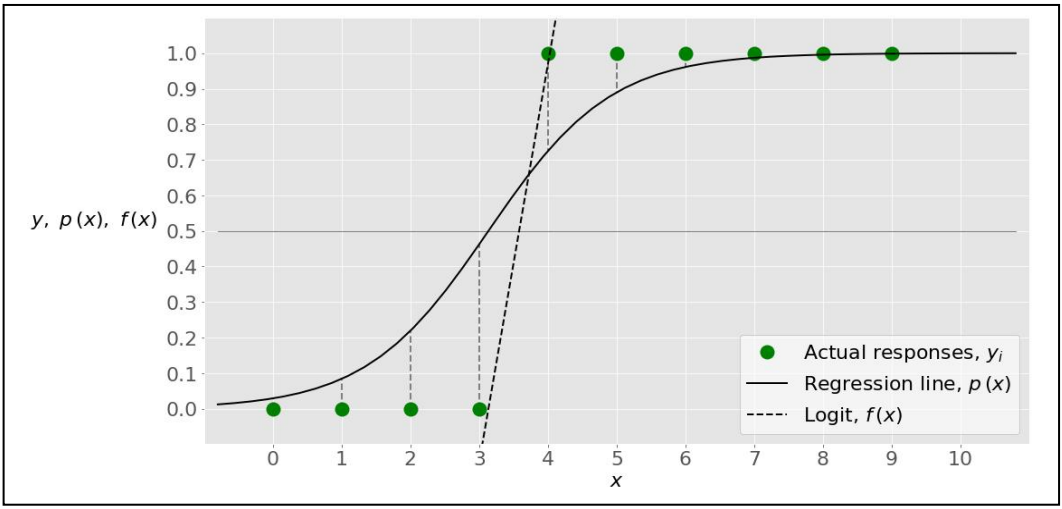


Рисунок 1.8. Результат використання моделі одновимірної логістичної регресії

Тут маємо набір пар вхід-вихід (або x-y), представлений зеленими кружечками. Це наші спостереження. Пам'ятайте, що y може набувати значень лише 0 або 1. Наприклад, крайній лівий зелений кружечок має вхід x = 0 і фактичний вихід y = 0. Крайній правий спостереження має x = 9 і y = 1.

Логістична регресія знаходить ваги b_0 та b_1, які відповідають максимальному LLF. Ці ваги визначають логіт $f(x) = b_0 + b_1x$, який є пунктирною чорною лінією. Вони також визначають прогнозовану ймовірність $p(x)$ = 1 / (1 + exp(-$f(x)$)), показану тут суцільною чорною лінією. У цьому випадку поріг $p(x)$ = 0.5 і $f(x)$ = 0 відповідає значенню x трохи більшому за 3. Це значення є межею між входами з прогнозованими виходами 0 та 1.

Багатовимірна логістична регресія має більше однієї вхідної змінної. На цьому рисунку показано класифікацію з двома незалежними змінними, x_1 та x_2:

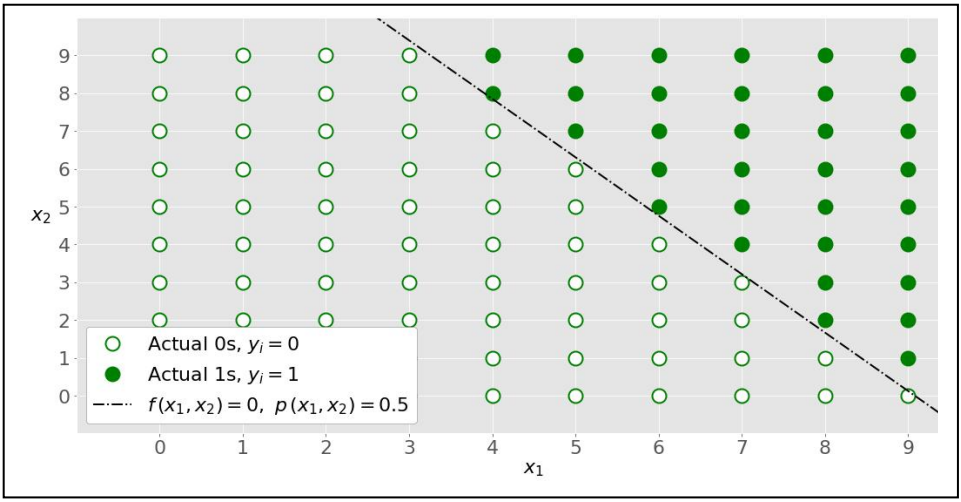


Рисунок 1.9. Результат використання моделі логістичної регресії із двома змінними

Цей графік відрізняється від одновимірного графіку тим, що обидві осі представляють входи. Виходи також відрізняються за кольором. Білі кола показують спостереження, класифіковані як нулі, тоді як зелені кола - як одиниці.

Логістична регресія визначає ваги b_0, b_1 та b_2, які максимізують LLF. Отримавши b_0, b_1 та b_2, зможемо отримати b_2:

Логіт $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2$. Ймовірність $p(x_1, x_2) = 1 / (1 + \exp(-f(x_1, x_2)))$. Штрих-пунктирна чорна лінія лінійно розділяє два класи. Ця лінія відповідає значенням $p(x_1, x_2) = 0.5$ та $f(x_1, x_2) = 0$.

Перенавчання - одна з найсерйозніших проблем, пов'язаних з машинним навчанням. Вона виникає, коли модель занадто добре вивчає навчальні дані. Тоді модель вивчає не тільки взаємозв'язки між даними, але й шум у наборі даних. Надмірно підігнані моделі, як правило, мають хорошу продуктивність з даними, які використовуються для їх підгонки (навчальними даними), але вони погано поводяться з невидимими

даними (або тестовими даними, тобто даними, які не використовуються для підгонки моделі).

Перенавчання зазвичай відбувається зі складними моделями. Регуляризація зазвичай намагається зменшити або “карати” (“штрафувати”) складність моделі. Методи регуляризації, що застосовуються до логістичної регресії, здебільшого мають тенденцію “карати” великі коефіцієнти b_0, b_1, ..., b_r:

- L1 регуляризація штрафує ЛНФ масштабованою сумою абсолютних значень ваг: $|b_0|+|b_1|+\cdots+|b_r|$.

- Регуляризація L2 штрафує LLF масштабованою сумою квадратів ваг: $b_0^2+b_1^2+\cdots+b_r^2$.

- Регуляризація еластичними сітками є лінійною комбінацією регуляризації L1 та L2.

Регуляризація може значно покращити продуктивність моделі на невидимих даних.

1.4. Опорно-векторні машини, метод опорних векторів.

Обґрунтування методу, приклад з одним предиктором. Розглянемо наступну ілюстрацію.

Рисунок 1.10. Розподіл коефіцієнта IQ опитаних осіб

На цьому графіку
- червоні кола представляють IQ осіб, які не встигли скласти тест вчасно
- зелені кола позначають IQ осіб, які встигли скласти тест вчасно

Тепер розглянемо абітурієнта, який подав заяву сьогодні, і чий IQ дорівнює 107. Чи повинні ми віднести цю людину до червоного чи зеленого кола? Чи є у цієї людини більша ймовірність завершити або не завершити тест вчасно? Щоб відповісти на це питання, на наступній діаграмі показано "розумну стратегію". Зокрема, як показано на цій діаграмі

- коротка вертикальна лінія позначає точку посередині між останнім червоним і першим зеленим колом - двома колами, які називаються краями,
- ця точка може використовуватися як поріг, тобто дані зліва від цього порогу класифікуються як червоні; дані справа від цього порогу класифікуються як зелені,
- чорне коло, що позначає абітурієнта, лежить праворуч від цього порогу і, таким чином, повинно бути класифіковане як зелене. Таким чином, ця особа, швидше за все, завершить тест вчасно.

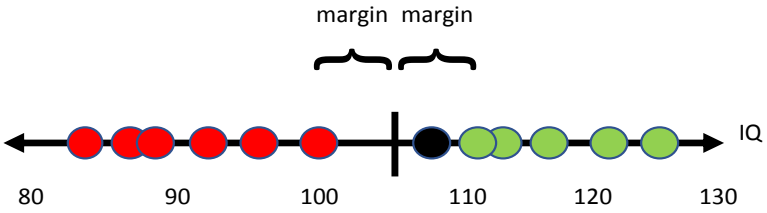


Рисунок 1.11. Вибір межі для "розумної стратегію" розділення класів

І хоча такий підхід є очевидним і простим, науковцям все ж вдалося розробити заплутані терміни для пояснення цих процедур. Наприклад:

- границя - це відстань між пороговою точкою та найближчими спостереженнями або колами,

- рішення вставити поріг посередині між двома краями називається класифікатором з максимальною межею, тому що цей поріг максимізує межі.

Викиди.

На жаль, на практиці рішення ускладнюють дві перешкоди. Перша складність полягає в тому, що більшість наборів даних містять викиди - точки даних, які відхиляються від очікуваного шаблону. Для ілюстрації, на наступному зображенні

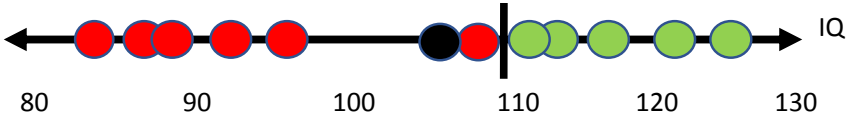


Рисунок 1.12. Неефективність найпростішої стратегії для вибірки з викидами

- одне з червоних кіл розташоване ближче до зеленого кола,

- якби ми застосували максимальний граничний класифікатор, то поріг знаходився б між цими двома краями,

- чорне коло, таким чином, було б класифіковане як червоне – тобто абітурієнт не завершить тест вчасно згідно нашого класифікатора,

- але цей висновок видається хибним, оскільки чорне коло загалом ближче до зелених кіл, ніж до червоних.

Щоб подолати цю проблему, нам може знадобитися відмовитися від максимально граничного класифікатора. Замість цього нам може знадобитися вставити поріг, який дозволяє одну або кілька помилкових класифікацій. Для ілюстрації, в наступному прикладі

- викид ігнорується,

- поріг вставляється посередині між червоним і зеленим колом, які знаходяться найближче один до одного - після ігнорування викиду

- Отже, чорне коло буде класифіковано як зелене – тобто абітурієнт класифікується як той, хто завершить тест.

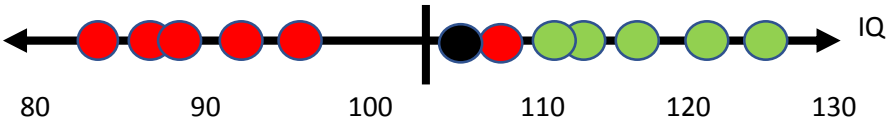


Рисунок 1.13. Неправильна класифікація викидів при “розумній стратегії”

Отже, як ми вирішуємо, які кола є викидами? По суті:

- можна спробувати різні порогові значення - після ігнорування одного, двох, трьох і більше кіл відповідно,

- можна використовувати перехресну перевірку (крос-валідацію), щоб вирішити, який поріг є найкращим,

- тобто можна вибрати поріг, який найточніше прогнозує результати в даних тестування,

- на практиці комп'ютерне програмне забезпечення саме виконає всі ці тести.

Зсув (упередження) проти дисперсії.

Цей приклад ілюструє життєво важливе протиріччя, що є центральною темою в машинному навчанні. У цьому прикладі, якщо застосовано максимальний граничний класифікатор і, таким чином, поріг вставляється між двома краями, то модель ідеально відображає наявні дані, тобто всі червоні кола з'являються ліворуч від цього порогу, а всі зелені кола - праворуч від цього порогу. Можна вибрати моделі машинного навчання, які точно відображають дані, тобто моделі з низьким рівнем упередження, однак така модель може неточно класифікувати майбутні випадки, тобто чорне коло може бути невірно класифіковане. На противагу, можна вибрати моделі машинного навчання, які не передбачають результати точно, тобто моделі з високою дисперсією, оскільки результати значно відрізняються від прогнозів.

Цікаво, що моделі з низьким рівнем упередженості часто мають високу дисперсію, і навпаки. Для ілюстрації розглянемо дві діаграми розсіювання на наступному зображенні:

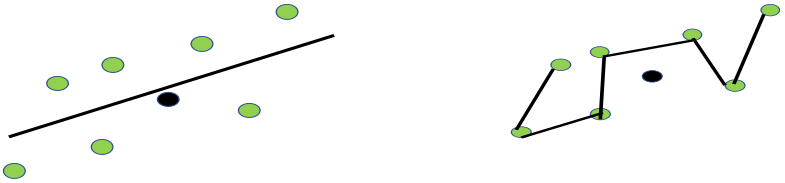


Рисунок 1.14. Ілюстрація проблеми компромісу упередженість/дисперсія

На лівій діаграмі розсіювання зелені кола помітно відхиляються від прямої лінії, отже, пряма лінія не відображає дані точно; відхилення є високим. Але пряма дуже близько передбачає чорне коло: дисперсія низька; більшість майбутніх випадків, ймовірно, будуть досить близькими до цієї прямої. Водночас, на правій діаграмі розсіювання червоні кола розташовані дуже близько до хвилястої лінії; таким чином, відхилення є низьким, однак, ця хвиляста лінія не дуже точно прогнозує чорне коло; багато майбутніх випадків, ймовірно, значно відхилятимуться від цієї хвилястої лінії.

Цей приклад наочно ілюструє важливу інтуїцію про машинне навчання:

- прості моделі, як правило, збільшують упередженість, але зменшують дисперсію: ці моделі передбачають майбутні випадки краще, ніж, можливо, вони представляють минулі дані,

- складні моделі, як правило, зменшують упередженість, але збільшують дисперсію; ці моделі не передбачають майбутні випадки так само добре, як вони представляють минулі дані.

Ця методика, за допомогою якої визначаються викиди і, таким чином, допускаються помилкові класифікації, називається:

- **класифікатором з м'якою межею**, оскільки відстань між порогом та найближчим колом, яке не є викидом, називається м'якою межею, а не межею

- або **класифікатором опорних вект**орів, оскільки ці найближчі кола також називаються опорними векторами, особливо якщо вони представлені в більш ніж одному вимірі.

Кластери, що перекриваються.

Друге ускладнення полягає в тому, що за багатьох обставин ми можемо бути не в змозі розділити два кластери - наприклад, червоні та зелені кола - однією точкою або лінією, навіть після виключення пропусків. Для ілюстрації, на наступному зображенні особи, які не

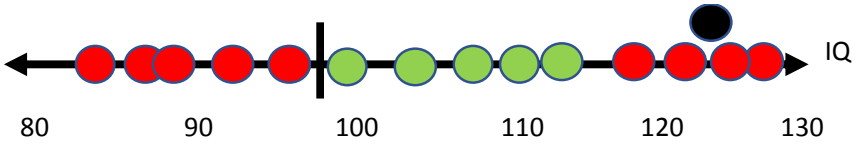


Рисунок 1.15. Приклад лінійно нероздільного набору даних

склали тест (червоні кола), мають або дуже низький, або дуже високий рівень IQ, Можливо, якщо їхній IQ занадто низький, вони не можуть зрозуміти матеріал. Якщо їхній IQ занадто високий, вони можуть відчувати нудьгу. Отже, незалежно від причини, незалежно від того, де встановлено поріг, багато кружечків будуть неправильно класифіковані. У цьому прикладі чорне коло буде класифіковано як зелене, але воно ближче до червоних.

На щастя, дослідники знайшли цікавий метод вирішення цієї проблеми. По суті, застосовуються різні формули, які збільшують кількість вимірів до тих пір, поки кластери даних не можна буде розділити прямою лінією, площиною або гіперплощиною. Для ілюстрації, на наступному графіку горизонтальна вісь *x* представляє IQ кожного кола, як і раніше, а вертикальна вісь *y* представляє квадрат цих значень IQ. На Рис. 1.16. IQ, що відповідає крайньому лі-

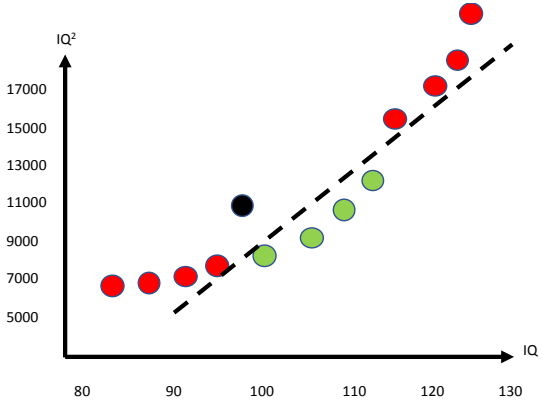


Рисунок 1.16. Розділення лінійно нероздільних даних

вому колу дорівнює 85, квадрат 85 дорівнює 7225. Отже, це коло з'являється з координатами 85 і 7225.

Зауважимо, що при перетворенні даних з одновимірного графіка у двовимірний, червоні та зелені кола тепер можна розділити однією лінією! Тобто кола над цією лінією, такі як чорне коло в цьому прикладі, будуть класифіковані як червоні або ті, хто, за прогнозами, не завершить тест. Кола нижче цієї лінії будуть класифіковані як зелені або ті, хто, за прогнозами, завершить тест.

Дійсно, в цьому прикладі ця проста лінія відображає всі існуючі дані і, таким чином, має низький рівень упередженості. Крім того, оскільки лінія є простою, відповідна формула, ймовірно, також правильно класифікує майбутні випадки і, таким чином, має низьку дисперсію.

Висновок з цього прикладу.

Навіть цей простий приклад ілюструє деякі важливі принципи:

1) після видалення викидів метод призначений для виявлення точки, лінії, площини або гіперплощини, яка розділяє два кластери точок даних - наприклад, абітурієнтів, які вчасно написали тест, і абітурієнтів, які не написали тест,

2) якщо жодна точка, лінія або площина не може розділити кластери точок даних, метод трансформує дані, суттєво збільшуючи кількість вимірів - наприклад, з одного виміру до двох,

3) метод продовжує збільшувати кількість вимірів доти, доки не буде знайдено лінію, площину, гіперплощину, яка розділяє два кластери точок даних,

4) цю лінію, площину або гіперплощину можна потім застосувати для класифікації майбутніх випадків.

Цей метод називається **опорно-векторними машинами**.

Поліноміальні ядра.

У попередньому прикладі дослідник або, принаймні, програма побудувала діаграму розсіювання, яка представляла як вихідні значення IQ, так і квадрати цих значень, а потім визначила лінію, яка розділяє червоні та зелені кола. Але ця формула, в якій вихідні значення підносяться до квадрату, не завжди буде ефективною. Як програма може визначити, яка формула буде ефективною? Можна спробувати наступний алгоритм:

1) програма застосовує формулу: **ядро** $= (a \times b + c)^d$,

2) програма продовжує варіювати c і збільшувати d - два параметри, до того часу, поки модель не стане ефективною, тобто, програма продовжує працювати до тих пір, поки модель або формула точно не класифікує дані.

Ця модель або підхід називається **поліноміальним ядром**.

Приклад ядра. Для ілюстрації розглянемо наступний приклад.

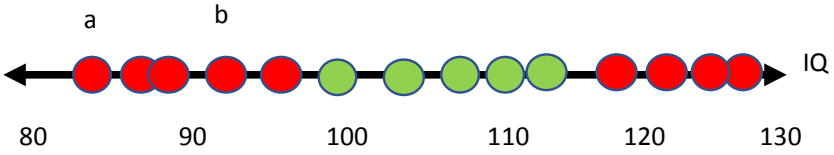


Рисунок 1.17. Лінійно нероздільна вибірка даних по IQ абітурієнтів

Тепер застосуємо формулу **ядро** $= (a \times b + c)^d$

В цій формулі a і b - це два кола, позначені a і b – їх числові значення приблизно 84 і 92 відповідно, c - параметр, який може змінюватися. Цей параметр відповідає за зміну масштабу, d - кількість вимірів.

Щоб застосувати цю формулу, ми спочатку вибираємо довільне значення c, наприклад, 0.5. Спочатку ми встановили d рівним 2. Отже, ядро

ядро = $(a \times b + 0.5)^2 = ab^2 + 0.5 \times ab + 05 \times ab + 0.25 = ab + ab^2 + 0.25$, після перетворень одержимо: ядро = $[a, a^2, 0.5] . [b, b^2, 0.5]$ як випливає з властивостей скалярного добутку: ядро = [84, 7056, 0.5] . [92, 8464, 0.5] бо a = 84 і b = 92. Ці два набори чисел [84, 7056, 0.5] та [92, 8464, 0.5] можна зобразити на діаграмі розсіювання.

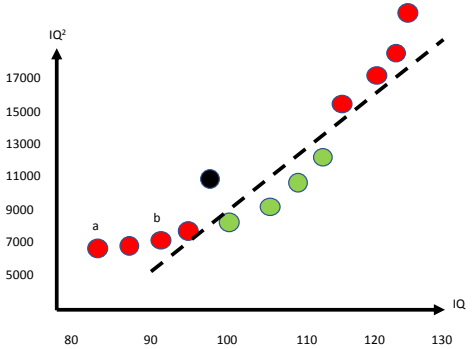


Рисунок 1.18. Застосування поліноміального ядра до лінійно нероздільної вибірки даних

Зокрема, кола *a* та *b* на наступній діаграмі розсіювання представляють ці два набори чисел. Зауважте, що набори чисел часто називають векторами. Ця фігура має лише два виміри. Вимір 0.5 не представлено. Можна припустити, що графік дорівнює 0.5 на осі z, яку ми не бачимо.

Досі формула лише перетворювала одновимірні дані, такі як числа 84 і 92, у двовимірні. Але, крім того, використовуючи формулу, ми можемо обчислити ядро - значення, яке приблизно вказує на відстань між двома точками. Тобто ядро або відстань дорівнює:

ядро = $(a \times b + 0.5)^2 = (84 \times 92 + 0.5)^2 = 59\ 729\ 712$.

Потім комп'ютер може застосувати цю процедуру до кожної пари кіл, щоб обчислити відстань між кожною парою кіл. Отже, чому комп'ютер обчислює відстань між кожною парою кіл? Яка користь від цієї інформації? Пам'ятаємо, що опорно-векторні машини намагаються визначити лінію, площину або гіперплощину, яка розділяє різні кластери - наприклад, червоні та зелені кола. Зокрема, ця машина призначена для максимізації відстані між цією лінією або площиною та найближчими точками у кожному кластері. Інформація про відстань між точками допомагає обчислити положення цієї лінії, площини або гіперплощини.

Отже, повторимо, для застосування алгоритму опорно-векторних машин, з використанням поліноміальних ядер, програма:

1) починає з конкретного значення *c* і *d* у формулі $(a \times b + c)^d$

2) використовує цю формулу для оцінки відстані між усіма точками даних - в кінцевому підсумку, щоб знайти лінію або площину, яка оптимально розділяє кластери точок даних

3) перевіряє, наскільки навчальні дані узгоджуються з цією моделлю

4) ітеративно і систематично змінює значення *c* і збільшує рівні *d* для покращення моделі.

Потім дослідник може використати перехресну перевірку (крос-валідацію) для оцінки цієї моделі.

Радіальне ядро та багато вимірність.

У попередніх прикладах IQ, єдиний предиктор, був представлений в одному вимірі, на осі *x*. Потім формули перетворили ці дані у двовимірні, потім у тривимірні і так далі, доки не вдалося відокремити кластери - наприклад, червоні та зелені кола. Зауважимо, що якщо дані містять два предиктори, потрібно побудувати діаграму розсіювання у двох вимірах, щоб представити ці дані, як показано на наступному зображенні. В цьому випадку ми не можемо провести лінію, яка б розділяла червоні та зелені кола, але, якщо дані потім трансформувати у три або більше вимірів, ми зможемо намалювати плоску поверхню, яка розділить червоні та зелені кола.

Також слід відмітити, що обґрунтування методу однакове, незалежно від того, скільки предикторів чи вимірів ми розрізняємо.

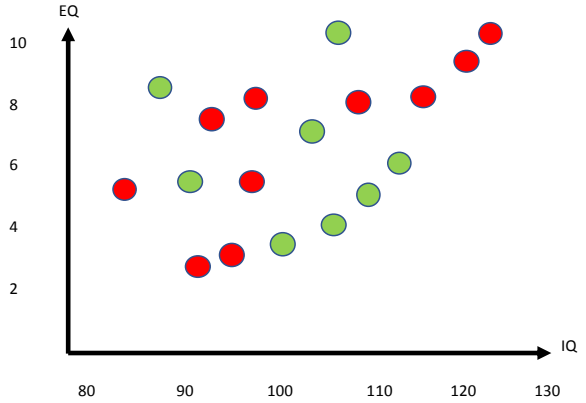


Рисунок 1.19. Застосування поліноміального ядра до лінійно нероздільної вибірки даних із двома фічами.

Якщо дані містять чотири або більше предикторів, нам потрібно побудувати графік у чотирьох або більше вимірах. Насправді, ніхто не може уявити собі чотири або більше вимірів. Але ті ж самі міркування та формули все одно застосовуються.

Радіальні ядра.

Замість поліноміальних ядер для моделювання даних програма може використовувати інші формули, які називаються радіальними ядрами. Ці ядра розширюють поліноміальні ядра до нескінченних розмірів. Радіальні ядра важче уявити, але вони, як правило, більш потужні, ніж поліноміальні ядра. Щоб проілюструвати принцип роботи радіальних ядер, розглянемо наступний набір даних (Рис. 1.20):

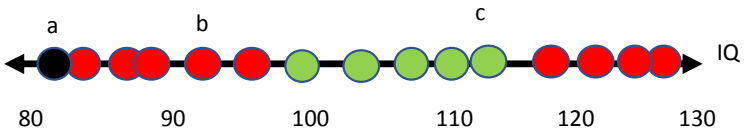


Рисунок 1.20. Нероздільна вибірка даних із однією фічею

Припустімо, ми хочемо класифікувати чорне коло. Тобто, ви хочете з'ясувати, чи слід класифікувати це коло як червоне, а отже,

малоймовірне завершення тези, чи як зелене, а отже, ймовірне завершення тези. Щоб відповісти на це питання радіальні ядра розглядають всі навчальні дані - всі інші кола - для класифікації цього чорного кола. Однак радіальні ядра надають більшу вагу тим даним, які знаходяться ближче до цього чорного кола. Наприклад, радіальні ядра нададуть значну вагу червоному колу, позначеному b, тому що це коло знаходиться досить близько до чорного кола. Зелене коло, позначене c, отримало б незначну вагу, тому що воно знаходиться далі від чорного кола. Після одержання цієї інформації радіальне ядро, швидше за все, класифікує чорне коло як червоне.

Щоб досягти цієї мети, радіальні ядра застосовують наступну формулу для зважування кіл:

радіальне ядро $= e^{-\gamma(a-b)^2}$.

Ключовою особливістю цієї формули є те, що зі збільшенням відстані між двома точками a і b вага зменшується.

1.5. Метод к найближчих сусідів.

Лінійна регресія працює в деяких випадках, але не завжди дає дуже точні прогнози. Тому математики розробили багато альтернативних моделей машинного навчання, які ми можемо використовувати. Алгоритм k-найближчих сусідів - одна з них.

Всі ці моделі мають свої особливості. Працюючи з алгоритмами машинного навчання, ми повинні мати глибоке розуміння кожної з них, щоб використовувати правильну модель у правильній ситуації. Щоб зрозуміти, чому і коли варто використовувати kNN, далі ми розглянемо, переваги і недоліки моделі kNN у порівнянні із іншими моделями машинного навчання.

Алгоритм kNN - це модель машинного навчання з учителем. Це означає, що він прогнозує цільову змінну, використовуючи одну або декілька незалежних змінних.

Алгоритм kNN легко адаптується як до класифікації, так і до регресії.

Алгоритм kNN трохи нетиповий у порівнянні з іншими алгоритмами машинного навчання. Кожна модель машинного навчання має свою специфічну формулу, яку потрібно обчислити. Специфіка алгоритму k-найближчих сусідів полягає в тому, що ця формула обчислюється не в момент фітингу моделі, а в момент прогнозування, а це не так для більшості інших моделей.

Коли надходить нова точка даних, алгоритм kNN, як видно з назви, починає з пошуку найближчих сусідів цієї нової точки даних. Потім він бере значення цих сусідів і використовує їх як прогноз для нової точки даних.

Як інтуїтивний приклад того, як це працює, подумайте про своїх сусідів. Ваші сусіди часто відносно схожі на вас. Можливо, вони належать до того ж соціально-економічної групи, що й ви. Можливо, у них така ж робота, як і у вас, можливо, їхні діти ходять до тієї ж школи, що й ваші, і так далі. Але для деяких завдань такий підхід не настільки корисний. Наприклад, немає сенсу дивитися на улюблений колір вашого сусіда, щоб передбачити ваш.

Алгоритм kNN заснований на тому, що ми можемо передбачити особливості точки даних на основі особливостей її сусідів. У деяких

випадках цей метод передбачення може бути успішним, а в інших - ні. Далі ми розглянемо математичний опис поняття "найближчий" для точок даних і методи об'єднання декількох сусідів в одне передбачення.

Щоб знайти точки даних, які знаходяться найближче до точки, яку потрібно передбачити, можна скористатися математичним визначенням відстані, а саме евклідовою відстанню. Щоб перейти до цього визначення, спочатку потрібно зрозуміти, що означає різниця двох векторів. Ось приклад:

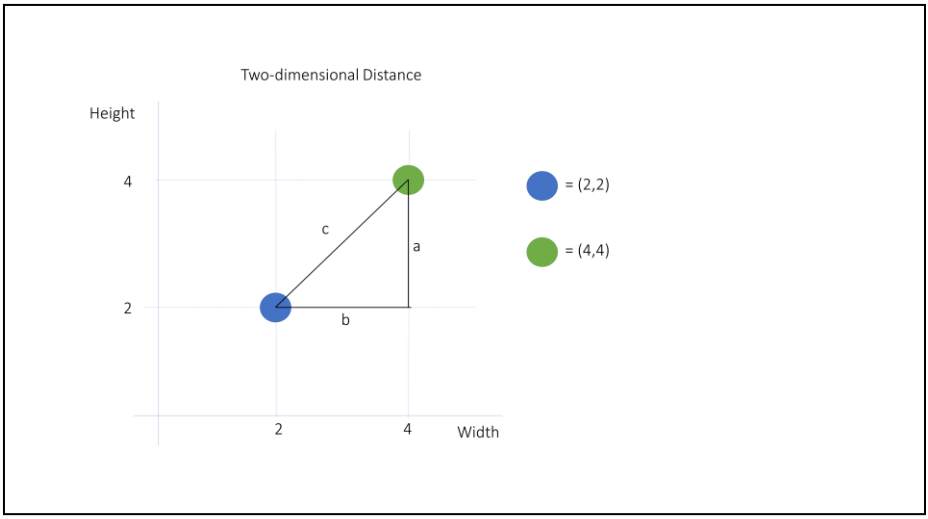


Рисунок 1.21. Обчислення відстані між двома точками даних із двома фічами

На Рис. 1.21 ми бачимо дві точки даних: синю в точці (2,2) і зелену в точці (4,4). Щоб обчислити відстань між ними, можна почати з додавання двох векторів. Вектор ***a*** йде з точки (4,2) в точку (4,4), а вектор ***b*** - з точки (4,2) в точку (2,2). Їх вершини позначені кольоровими точками. Зверніть увагу, що кут між ними складає 90 градусів.

Різниця між цими векторами - це вектор ***c***, який йде від вершини вектора ***a*** до вершини вектора ***b***. Довжина вектора ***c*** представляє собою відстань між двома нашими точками даних.

Довжина вектора називається нормою. Норма - це додатне значення, яке вказує на величину вектора. Норма вектора обчислюється за

допомогою формули Евкліда:

$$d(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$$

У цій формулі відстань обчислюється шляхом піднесення до квадрату різниць у кожному вимірі, а потім добуванням квадратного кореня з суми цих значень. У нашому випадку нам слід обчислити норму вектора різниці $\boldsymbol{c}$, щоб отримати відстань між точками даних.

Тепер, щоб застосувати це до наших даних, ми повинні розуміти, що наші точки даних насправді є векторами. Ми можемо обчислити відстань між ними, обчисливши норму вектора різниці.

Тепер, коли ми знаємо, як обчислити відстань від будь-якої точки до будь-якої точки, ми можемо використати цей спосіб для пошуку найближчих сусідів точки, для якої ми хочемо зробити прогноз.

Нам потрібно знайти певну кількість сусідів, і ця кількість позначається k. Мінімальне значення k дорівнює 1. Це означає використання лише одного сусіда для прогнозування. Максимальне - це кількість точок даних, які у вас є. Це означає використання всіх сусідів. Значення k визначає користувач, для чого можуть стати у пригоді інструменти оптимізації.

Зауважимо, що у задачах регресії цільова змінна є числовою. Тому ми об'єднуємо кілька сусідів в один прогноз, беручи середнє значення їхніх значень цільової змінної.

У задачах класифікації цільова змінна є категоріальною. Але для категоріальних змінних ми не можемо брати середні значення. Наприклад, яким буде середнє значення для трьох прогнозованих марок автомобілів? Очевидно, що ми не можемо застосувати середнє значення до прогнозів щодо класів.

Замість цього, у випадку класифікації, ми берете моду. Мода - це значення, яке зустрічається найчастіше. Це означає, що ми підраховуєте класи всіх сусідів і залишаємо найпоширеніший клас. Прогноз - це значення, яке зустрічається найчастіше серед сусідів.

Якщо існує декілька режимів, існує декілька можливих розв’язків. Ми можемо вибрати фінального переможця випадковим чином серед переможців. Ми також можемо прийняти остаточне рішення на основі відстаней між сусідами, в цьому випадку буде збережено моду найближчих сусідів.

1.6. Перцептрон, мультиперцептрон, нейронні мережі.

В цьому підрозділі ми розглядатимемо штучну нейронну мережу (ШНМ) - математичну функцію, призначену для імітації основної функції біологічного нейрона; вона використовується в багатьох застосунках, таких як прогнозування, класифікація вхідних даних і фільтрація даних. Можна дати ще таке визначення нейронної мережі - це машина, призначена для імітації роботи людського мозку, який складається з великої кількості нейронів, що об'єднуються для вирішення певної задачі.

Історія штучних нейронних мереж бере свій початок на початку 1940-х років. Перша важлива стаття про нейронні мережі була опублікована фізіологами Уорреном Маккалохом і Уолтером Піттсом в 1943 році, вони запропонували просту модель нейрона з електронним ланцюгом, яка складається з двох входів і одного виходу, в 1949 році Дональд Хебб запропонував закон навчання, який став відправною точкою для алгоритму навчання нейронних мереж, в 1950 і 1960 роках багато дослідників (Блок, Мінський, Паперт і Розенблат) працювали над перцептронами, де перший тип нейронних мереж отримав назву перцептронів. Перцептрон - це дуже просте математичне представлення нейрона, на якому базується більшість штучних нейронних мереж, як показано нижче на рисунку 1.22.

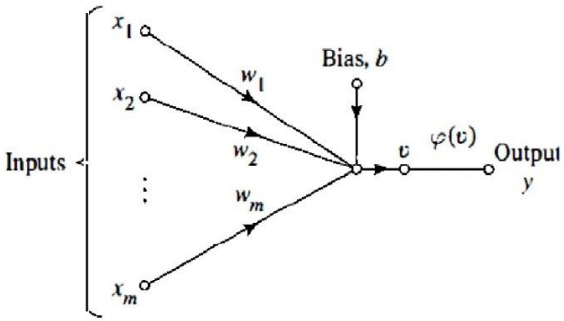


Рисунок 1.22 Перцептрон.

На цьому рисунку показано, що входи нейрона представлені X_1 , X_2 ... X_m , потім помножені на відповідну вагу W_1 , W_2 ... Wm подібно до синаптичної сили в біологічному нейроні, зовнішнє зміщення позначається через b. Сума цих входів з відповідними вагами та зміщенням "b" позначаєтьтся через V, де V обчислюється за рівнянням:
$V = \sum_{i=1}^{m} Wi * Xi + b$.

Після цього функція активації порівнюється зі значенням певного порогу. Якщо загальна сума входів, помножена на відповідну вагу, більша за поріг, то спрацьовує вихід (O), а якщо загальна сума входів, помножена на відповідну вагу, менша за поріг, то вихід (O) не спрацьовує.

Бернард Відроу та Марсіан Хофф у 1959 році розробили модель під назвою "АДАЛІН" (Adaptive Linier Neuron), а "МАДАЛІН" складається з "багатьох АДАЛІН" (Multilayer ADALINE). Відроу та Хофф у 1960 році розробили математичний метод адаптації ваги, де алгоритм залежав від мінімізації квадрату помилки, після чого цей алгоритм став називатися найменшою середньоквадратичною помилкою (LMS). У 1962 році Френк Розенблатт зміг продемонструвати збіжність алгоритму навчання. У 1969 році Марвін Мінскі та Сеймур Паперт опублікували книгу, в якій показали, що Персептрон не може навчатися на функціях, які не є лінійно відокремлюваними.

Як наслідок, це призвело до обмеження фінансування, доступного для досліджень штучних нейронних мереж, тому дослідження нейронних мереж занепали протягом 1970-х років і до середини 1980-х.

Алгоритм зворотного поширення був вперше розроблений Вербосом у 1974 році; найбільшого розвитку він набув у 1985-1986 роках, коли Румельхарт, Хінтон і Віллімас винайшли зворотнє поширення, яке є потужним інструментом для навчання багатошарових нейронних мереж. Поява методу зворотного поширення вражаюче розширила спектр задач, до яких можна застосовувати нейронні мережі.

Нейронні мережі та їх застосування.

Нейронна мережа - це складний математичний алгоритм, який певною мірою придатний для вирішення всіх питань, які не підпорядковуються мінімумам математичних констант і імітують роботу людського мозку з розпізнавання звуків, слів та зображень.

Більшість застосувань штучних нейронних мереж підпадають під три наступні розділи:

1) класифікація (використання вхідних значень для оцінки класифікації, наприклад, розпізнавання символів),

2) передбачення (використання вхідних даних для передбачення вихідних, наприклад, прогнозування погоди, вибір найкращих акцій на ринку),

3) Фільтрація даних (зробити вхідний сигнал більш плавним, наприклад, вилучити шум з телефонного сигналу).

Функція активації штучних нейронних мереж (ШНМ).

Штучна нейронна мережа (ШНМ) була введена Маккалохом і Піттсом, де ШНМ - це математична функція, призначена для імітації основної функції біологічного нейрона, який складається з великої кількості (нейронів), що працюють разом для вирішення конкретних завдань на основі навчальних даних, які складаються з входів, ваг входу і виходу.

Кожен вхід цього нейрона позначається X_1 , X_2 ...X_n , потім множиться на відповідну вагу W_1 , W_2 ...W_n, сума входів з відповідними вагами, і дає вихід, який називається "NET", як показано на рис. 1.23. Потім значення результату порівнюється з пороговим значенням:

$\text{net} = \sum_{i=1}^{n} \text{Wi} * \text{Xi}$

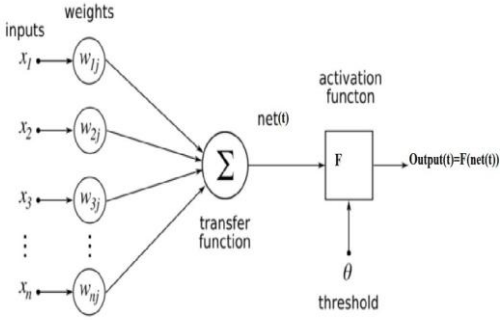


Рисунок 1.23: Принципова схема штучного нейрона.

Функція активації "F" називається передаточною функцією, де функція активації "F" діє як функція згладжування таким чином, що вихід нейрона в нейронній мережі знаходиться між певними значеннями.

Передавальна функція перетворює вхідні сигнали у вихідні. Існує багато функцій активації, таких як функція передачі з жорстким обмеженням, лінійна функція передачі та сигмоїдна функція передачі (логістична функція), як показано нижче на рис. 1.24.

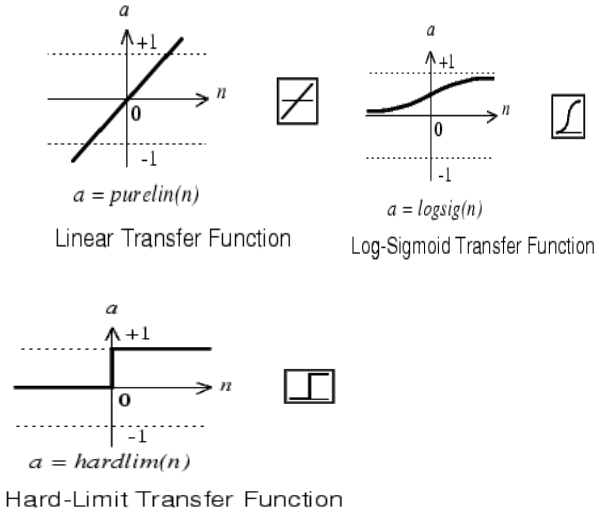


Рисунок 1.24: Типи функцій активації.

Вихід функції Hard-limit може бути як "0", так і "1" в залежності від порогу. В результаті неперервності цієї функції було виявлено, що вона не є достатньою для багатошарової штучної нейронної мережі, тому для перетворення рівня активації нейрона (зваженої суми входів) у вихідний сигнал використовується сигмоїдна функція, яка є найбільш поширеним типом активаційної функції, вона є прикладом логістичної функції, тому більшість штучних нейронних мереж (ШНМ) за замовчуванням використовують сигмоїдну функцію активації (логістичну функцію).

Логістична функція.

Логістична функція - це звичайна сигмоїдна крива, яка має "s-подібну форму", названа так у 1844 або 1845 році П'єром Франсуа Верхульстом, який досліджував її у зв'язку з ростом населення. Логістичні функції часто використовуються в нейронних мережах для введення нелінійності в модель і/або для обмеження сигналів в межах заданого діапазону. Логістична функція також відома як лог-сигмоїдна функція, яка є нелінійною криволінійною функцією s-подібної форми і

називається сигмоїдною функцією. Сигмоїдна функція є найпоширенішим типом активаційної функції (функція, що використовується для перетворення рівня активації нейрона (зваженої суми входів) у вихідний сигнал), що використовується для побудови нейронної мережі. Вона є всюди диференційованою і строго зростаючою функцією. Сигмоїдна передавальна функція може бути записана у вигляді**:**

Y(x) =1 / (1+exp (-αx)), де "α" =1.

Тут Y(x) - зважена сума всіх синаптичних входів плюс зсув нейрона "x", а "y" - вихід нейрона. Сигмоїдна функція описуєтьсяза допомогою експоненціального рівняння, а "α" є параметром нахилу і, змінюючи "α", можна отримати різні форми функції

Уніполярна сигмоїдна функція.

Функція активації уніполярної сигмоїдної функції описується за допомогою логарифмічної функції, де вихід обмежений між "0" та "1", логарифмічна сигмоїдна функція задається наступним чином:

G(x) =1 / (1+exp (-x)).

Діапазон вхідного сигналу уніполярної передавальної функції знаходиться між плюс нескінченністю та мінус нескінченністю, а вихідний сигнал знаходиться в діапазоні від "0" до "1", як показано нижче на рис. 1.25.

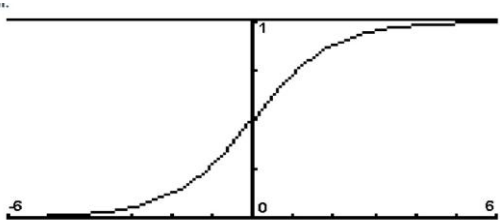


Рисунок 1.25: Однополярна сигмоїдна функція.

Біполярна сигмоїдна функція.

Біполярна сигмоїдна функція схожа на сигмоїдну функцію, але ця функція активації використовується, коли діапазон бажаного виходу обмежений між -1 та 1, коли вхід має будь-яке значення між плюс нескінченністю і мінус нескінченністю, як показано нижче на рис. 1.26.

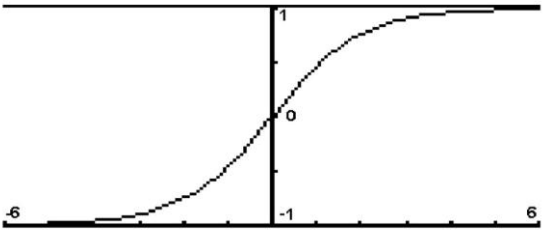


Рисунок 1.26: Біполярна сигмоїдна функція.

Сигмоїдна функція в нейронній мережі зі зворотним поширенням.

Функції активації є одним з найважливіших компонентів штучної нейронної мережі (ШНМ). Вони відповідають за надання пріоритету різним входам мережі. Раніше нейронні мережі будувалися з використанням жорстко обмежених (порогових) функцій активації, де вихід може бути або нульовим, або одиничним, але зростаюча потреба ШНМ в нелінійних системах змусила використовувати функції активації з нелінійними властивостями, які не базуються на перемикальних (жорстко обмежених) функціях і можуть давати пропорційний вихід до заданого входу.

Використання таких функцій для неперервних вихідних значень з обмеженим діапазоном зміни значень привернуло увагу дослідників у галузі ШНМ.

Сигмоїдна функція, яка була введена в 1844 році, була обрана для введення в функцію штучної нейронної мережі через свою нелінійність і безперервність виходу, що здається більш ефективним і корисним для використання в нейронній мережі зі зворотним поширенням, де зворотне поширення є ефективним і популярним методом, який був винайдений в 1986 році Румельхартом, Хінтоном та Вільямсом для навчання багатошарової нейронної мережі (мережа має вхідний шар та один або більше прихованих шарів і вихідний шар) для вирішення складних задач, процес навчання складається з двох проходів через шари мережі, прямого проходу та зворотного проходу.

При прямому проході сигмоїдна функція буде використана для визначення виходу прихованого шару та виходу вихідного шару для введення нелінійності в систему.

При зворотному проході похідна сигмоїдної функції використовується в налаштуваннях ваг вихідного шару та прихованого шару для обчислення похибки процесу зворотного поширення, тому похідна сигмоїдної функції зазвичай використовується при навчанні мережі.

Через свої властивості (диференційованість скрізь і нелінійність системи) логістична функція (сигмоїдна функція) є кращою для використання в нейронній мережі зі зворотним поширенням.

Одношаровий перцептрон (SLP).

Одношарова перцептронна мережа (SLP) є найпростішим типом нейронної мережі. Одношаровий перцептрон (SLP) складається з декількох зовнішніх входів, які потім множаться на відповідні ваги, після чого утворюється вихідний шар, як показано на рис. 1.27.

Одношарову перцептронну мережу можна вважати найпростішим видом мережі прямого поширення, де пряме поширення означає, що дані надходять від вхідного до вихідного шару в одному напрямку, вихід активується, коли сума добутків входів і відповідних ваг перевищує поріг, і де вихід деактивується, коли сума добутків відповідних входів і відповідних ваг нижча за поріг.

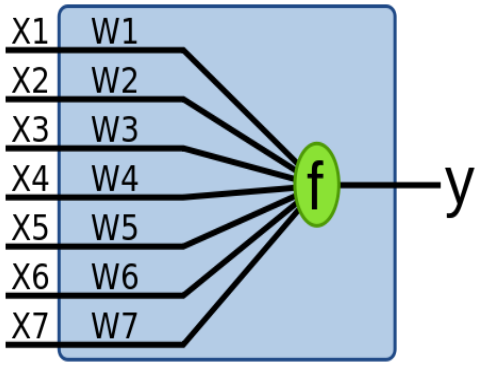


Рисунок 1.27: Одношаровий персептрон.

Багатошаровий персептрон (MLP).

Багатошаровий персептрон (MLP) є другим типом нейронної мережі прямого поширення, з одним або декількома шарами між вхідним і вихідним шарами, які називаються прихованими шарами, тому всі нейронні мережі мають вхідний шар і вихідний шар, як показано на рис. 1.28, кількість вхідних нейронів зазвичай відповідає кількості

незалежних змінних, які подаються на мережу, кількість прихованих шарів може змінюватися від мережі до мережі, а кількість вихідних нейронів залежить від того, яку задачу виконує мережа.

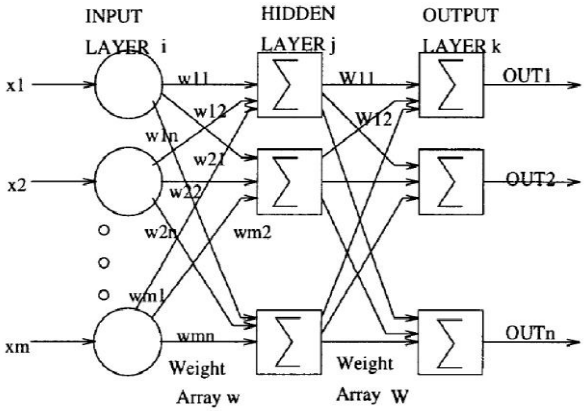


Рисунок 1.28: Багатошаровий персептрон.

Ця мережа складається з трьох шарів: вхідного шару зліва, одного прихованого шару посередині та вихідного шару справа.

Вхідний шар - це перший шар нейронної мережі, який отримує вхідні дані. У прямих нейронних мережах з одним або декількома нейронами може бути один або декілька прихованих шарів. У нейронних мережах прямого поширення є один вихідний шар. Вихідний шар розташований після вхідного та прихованого шарів, вихідний шар є третім і останнім шаром у штучній нейронній мережі.

Багатошаровий персептрон (MLP) може вирішувати більш складні задачі, ніж одношаровий, він може вирішувати задачі, які не піддаються лінійному розділенню, використовуючи алгоритм зворотного розповсюдження, який може використовуватися з будь-якою кількістю шарів.

Коли кожен вузол кожного шару штучної нейронної мережі з'єднаний з кожним вузлом сусіднього шару, в цьому випадку штучна нейронна мережа називається повнозв’язною мережею, тоді як штучна нейронна мережа називається частковозв’язною мережею, коли частина зв'язків забирається з мережі.

Нейронна мережа зворотного поширення (BPNN).

Алгоритм зворотного поширення був вперше запропонований Полом Вербосом у 1970-х роках. Зворотне поширення є потужним інструментом, створеним у 1986 році, коли різні дослідники винайшли систематичний спосіб навчання багатошарових штучних нейронних мереж для вирішення складних завдань, які вони назвали алгоритмом зворотного поширення помилки.

Зворотне поширення помилки складається з двох основних проходів через шари мережі: прямого і зворотного. У прямому проході вхідний сигнал подається на шари мережі, і його вплив поширюється через мережу шар за шаром, в результаті чого на виході формується фактичний вихід мережі.

Під час зворотного проходу синаптичні ваги коригуються (оновлюються) за правилом корекції помилок. Зокрема, фактичний вихід віднімається від бажаного виходу, який називається цільовим, щоб отримати помилку мережі, ця помилка поширюється назад через мережу, синаптичні ваги на вихідному шарі та прихованому шарі оновлюються (коригуються) таким чином, щоб зробити фактичний вихід мережі ближчим до бажаного (цільового).

Прямий та зворотний прохід алгоритму зворотного розповсюдження показано нижче на рис. 1.29.

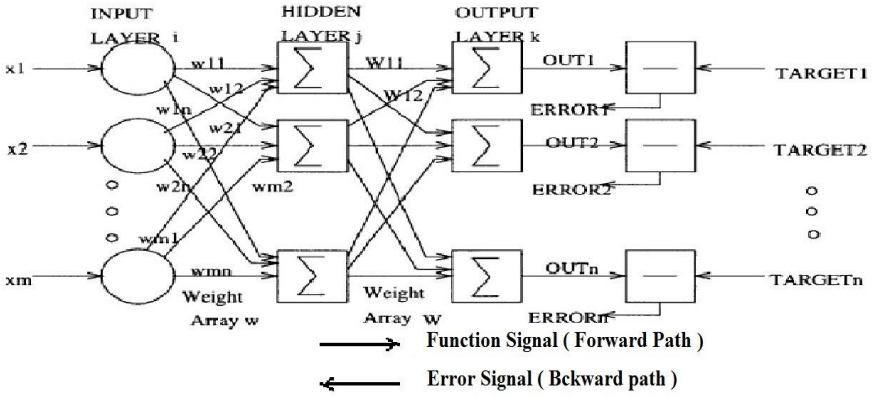


Рисунок 1.29: Архітектура нейронної мережі зі зворотним поширенням.

Прямий прохід і обчислення.

Процес прямого поширення (проходу) розпочинає навчання нейронної мережі за допомогою методу зворотного поширення, де просте тришарове зворотне поширення показано нижче на рис. 1.30.

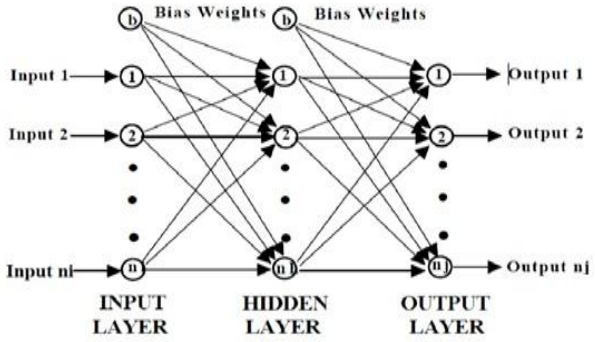


Рисунок 1.30: Структура мережі зворотного поширення.

Мережа зворотного поширення складається з трьох шарів: вхідного шару (i), прихованого шару (h) та вихідного шару (j), коли вхідні дані проходять вперед через шари, вихідні дані обчислюються за допомогою сигмоїдної функції активації, як показано на рис. 1.27.

Вихід будь-якого нейрона (кілька входів і вихід сигналу) у всіх шарах задається за допомогою цих рівнянь:

$\text{Net} = \sum_{i=1}^{n} Xi * Wi,$

Out = F (net),

F (Net) = 1 / 1 + exp (-net).

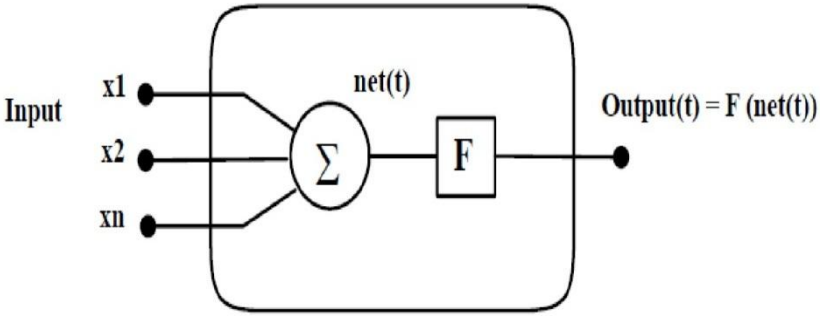


Рисунок 1.31: Штучний нейрон.

Тут "F" - сигмоїдна функція активації, похідна сигмоїдної функції зазвичай використовується для навчання мережі, її можна отримати наступним чином:

$\partial F(net)/\partial net = \exp(-net) / (1 + \exp(-net))^2 = (1+ \exp(-net)) * (\exp(-net) / (1 + \exp(-net)) = out(1- out) = F(net)(1- F(net))$.

Через свої властивості (диференційованість скрізь і нелінійність системи) логістична функція (сигмоїдна функція) є кращою для використання в нейронній мережі зі зворотним поширенням.

Вхідний шар (i), прихований шар (h) і вихідний шар (j) у прямому поширенні.

Шлях прямого поширення починається, коли вхідні дані передаються вперед через мережу, де вихід вхідного шару (O_i) дорівнює входу вхідного шару (I_i), як записано в цьому рівнянні:

Вхід вхідного шару (I_i) = Вихід вхідного шару (O_i).

Потім кожен вихід вхідного нейрона на виході вхідного шару (O_i) множиться на відповідну вагу і підсумовується, щоб отримати вхід прихованого шару (I_h), як описано в наступному рівнянні:

$(I_h) = \sum_i W_{hi} * O_i$, де (I_h) − вхідні дані прихованого шару.

Після цього кожен вихід прихованого нейрона у виході прихованого шару (O_h) обчислюється за допомогою логістичної функції (сигмоїдної функції), як записано в наступному рівнянні:

$(O_h) = 1 / (1+\exp(-I_h))$, де ($O_h$) − вихід прихованого шару

Кожен вихід вхідного нейрона на виході прихованого шару (O_h) множиться на відповідну вагу і підсумовується, щоб отримати вхід вихідного шару (I_j), як описано в наступному рівнянні:

$(I_j) = \sum_h W_{jh} * O_h$, де ($I_j$) − вхідні дані вихідного шару.

Потім кожен вихід нейрона на виході вихідного шару (O_j) обчислюється за допомогою логістичної функції (сигмоїдної функції), як записано в наступному рівнянні.

$(O_j) = 1 / (1+\exp(-I_j))$, де ($O_j$) – вихід вихідного шару.

З рівняння для вхідних даних вихідного шару випливає, що вихід вихідного шару (O_j) є функцією від входу вихідного шару f (I_j).

Рівняння, які були використані вище, є дуже важливими для обчислення виходу, який повністю відрізняється від бажаного виходу (цілі), оскільки всі ваги у всіх шарах мережі є невеликими випадковими

величинами, зазвичай між (-1 і +1) і (0 і +1) або іншими малими значеннями, тоді помилка кожного нейрона у вихідному шарі обчислюється для використання в іншому шарі мережі для оновлення ваг.

Розповсюдження зворотного поширення.

Після того, як фактичний вихід був обчислений при прямому поширенні, починається зворотній прохід шляхом обчислення помилки кожного нейрона у вихідному шарі.

Похибку мережі можна визначити як різницю між очікуваним вихідним значенням, яке називається цільовим і позначається "T_j ", і фактичним вихідним значенням, яке було обчислено при прямому поширенні, що позначається "O_j ", де маленька буква "j" позначає вихідний шар.

Наступне вносить похибку, яка позначається символом "E_p ":

$$E_p = \sum_{j=1}^{Nj} \left(T_{pj} - O_{pj}\right)^2.$$

Таким чином, помилка для кожної вихідної одиниці "j" базується на різниці між розрахунковим та бажаним виходом для цієї одиниці.

Маленька буква "p" вказує на значення для даної порції даних, метою навчання мережі є наближення фактичного виходу (O) кожного нейрона вихідного шару до його цільового (T), щоб згодом мінімізувати похибку.

З рівняння для вхідних даних вихідного шару випливає, що вихід вихідного шару (O_j) є функцією від входу вихідного шару f (I_j), як описано в рівнянні $O_j = f(I_j)$.

Перша похідна цієї функції виконує роль основної магістралі при зворотному поширенні помилки, у вихідному шарі сигнал помилки буде розраховуватися за допомогою наступних рівнянь, де сигнал помилки позначається "Δ_j ", а похідна цієї функції позначається "Ḟ":

$\Delta_j = \dot{F}(I_j) * (T_j - O_j)$,

$\Delta_j = O_j (1 - O_j) * (T_j - O_j)$.

Це значення помилки буде використано для визначення ваг у вихідному шарі (j) та прихованому шарі (h), тому значення помилки поширюється назад через шари, де відбувається цей процес.

При багаторазовому повторенні навіть ця похибка зменшиться.

Всі ці рівняння, які були використані вище, будуть коригувати ваги за допомогою наступних кроків:

1) завантажуємо порції даних в шари мережі і поширюємо їх через вхідні шари, повз приховані шари, до вихідного шару,

2) обчислюємо похибки шляхом порівняння між розрахунковим значенням виходу (фактичним виходом) та бажаним виходом (цільовим),

3) визначаємо похідну помилки для кожного вихідного нейрона,

4) використовуємо цю похідну для оновлення (коригування) ваг вихідного шару та прихованих шарів.

Тому структура будь-якої програми, яка використовує нейронну мережі зі зворотним поширенням, має вигляд, як зображено на рис. 1.32 нижче:

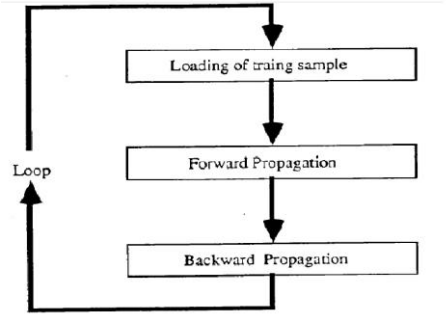


Рисунок 1.32: Структура будь-якої програми із використанням нейронної мережі зі зворотним поширенням.

Швидкість навчання та коефіцієнт імпульсу.

Ці два важливі параметри напряму впливають на здатність нейронної мережі до навчання.

Першим є коефіцієнт швидкості навчання (розмір кроку навчання), який позначається "η", де розмір кроку навчання (η) визначає, наскільки повинні змінюватися ваги, щоб зменшити функцію помилки (середньоквадратичну помилку, MSE). Якщо розмір кроку навчання (η) дуже малий, то процес навчання (збіжності) буде дуже повільним, якщо коефіцієнт швидкості навчання (η) занадто великий, то функція помилки буде збільшуватися і, ймовірно, це призведе до нестабільності і глобальні мінімуми будуть відсутні, тому коефіцієнт швидкості навчання (η) слід вибирати дуже ретельно, щоб прискорити збіжність і одночасно зберегти стабільність мережі.

Другим є коефіцієнт імпульсу, який позначається α, де коефіцієнт імпульсу (α) - це метод, вперше застосований Румельхартом, Хінтоном і Вільямсом для покращення часу навчання алгоритму зворотного розповсюдження шляхом вирішення конкретної задачі пошуку локального мінімуму, як показано нижче на рис. 1.33.

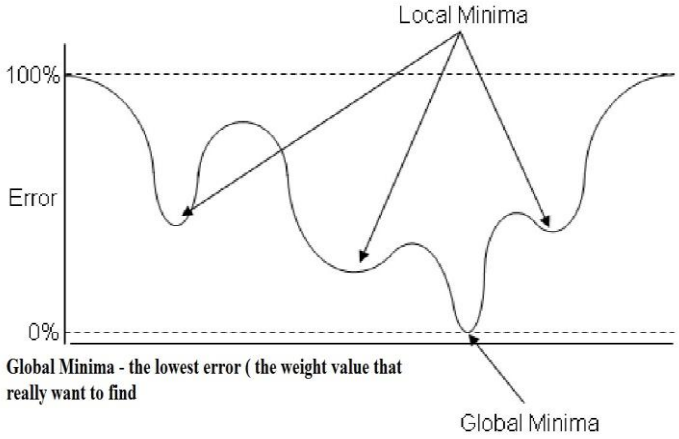


Рисунок 1.33: Області локальних мінімумів та глобальних мінімумів.

Локальний мінімум виникає тому, що алгоритм завжди змінює ваги таким чином, щоб викликати падіння функції помилки, але помилка може ненадовго зростати як частина більш загального падіння. У такому випадку алгоритм застрягне на якомусь місці, оскільки не зможе піднятися на вершину пагорба, і в цьому випадку похибка не зменшиться.

Коефіцієнт імпульсу (α) є дуже важливим коефіцієнтом, щоб уникнути падіння в яму локального мінімуму, де в цьому місці помилка більше нуля, з метою досягнення глобального мінімуму, де в цьому місці помилка приблизно дорівнює нулю.

Слово "імпульс" походить від аналогії з м'ячем, що котиться з великим імпульсом над вузькою ямою, якщо м'яч котиться повільно, то він впаде і застрягне в ямі. Якщо м'яч котиться досить швидко, він не застрягне в ямі.

Значення коефіцієнта імпульсу (α) і коефіцієнта швидкості навчання (η) зазвичай коливаються в межах від 0 до 1, і загалом ці два

параметри використовуються для прискорення процесу зворотного розповсюдження.

Підготовка вхідних даних.

Найкращий спосіб навчання мережі - це подати перший шаблон і оновити всі ваги у всіх шарах мережі, потім застосувати другий шаблон і змінити всі ваги у всіх шарах мережі (ті ж самі процедури, що і в першому шаблоні), потім третій шаблон і так далі до останнього шаблону, потім повернутися до першого шаблону і повторити ці процедури, поки помилка не стане дуже малою.

Однією з найпоширеніших помилок при запуску навчання шаблонів є подача першої частини даних через шари мережі, запуск алгоритму, після чого повторення його до тих пір, поки помилка не стане дуже малою, потім застосування другої частини даних і повторення цієї процедури, потім третьої частини і так далі до останньої частини.

Якщо це сталося, то мережа завершить свою роботу, вивчивши лише останню порцію даних, тобто, коли до мережі буде застосовано наступну порцію даних, мережа забуде попередню і так далі, доки не дійде до останньої порції даних.

Загальна помилка мережі буде оцінюватися шляхом підсумовування всіх помилок для кожного окремого нейрона, а потім для кожного набору даних, як показано нижче на рис. 1.34, тобто мережа продовжує тренувати всі вхідні дані мережі до тих пір, поки загальна помилка не впаде до значення бажаної (цільової), після цього алгоритм зупиняється. Коли мережа буде натренована, загалом мережа буде розпізнавати не тільки оригінальні вхідні дані, але й передбачати інші значення на основі вхідних даних, або в інших випадках, мережа буде розпізнавати не тільки оригінальні патерни (вхідні дані), але й розпізнавати пошкоджені та зашумлені патерни. Мережа буде використовувати скориговані (оновлені) ваги на етапі навчання як ваги на етапі тестування.

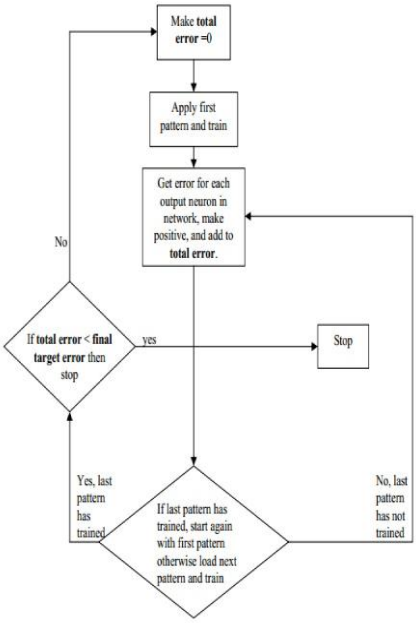


Рисунок 1.34: Процедура розрахунку сумарної похибки.

Налаштування ваг у вихідному шарі.

Всі ваги в різних шарах ініціалізуються невеликим випадковим числом, процес оновлення ваг починається з кінця прямого розповсюдження, тобто з вихідного шару в прямому розповсюдженні, а функція помилки буде оновлювати вагу в зворотному напрямку до інших шарів мережі.

Ваги (W_{jh}) між прихованими шарами (h) та вихідними шарами (j) оновлюються за допомогою рівняння

W_{jh} (нова) = W_{jh} (стара) + η*Δ *O_{jh},

щоб уникнути застрягання на локальному мінімумі, де в цьому місці похибка вище нуля. З метою досягнення глобального мінімуму, де в цьому місці похибка приблизно дорівнює нулю, можна додати імпульсний фактор (α), як у рівнянні

W_{jh} (нова) = W_{jh} (стара) + η*Δ *O_{jh} + α*[δ*W_{jh} (стара)].

Тут δ*W_{jh} означає попередню зміну ваги. Налаштування ваг для вихідного шару простіше, ніж для інших шарів, оскільки цільове значення кожного нейрона у вихідному шарі є доступним.

Налаштування ваг у прихованому шарі.

На відміну від нейронів вихідного шару, цільові вектори недоступні для нейронів у прихованих шарах. Доданок з помилкою для прихованого нейрона описується у рівнянні

$\Delta_h = \dot{F}(I_h) * \sum_{j=o}^{Nj} W_{jh} * \Delta_j,$

а потім у рівнянні

$\Delta_h = O_h * (1 - O_h) *. \sum_{j=o}^{Nj} W_{jh} * \Delta_j.$

Оновлення ваг між прихованим шаром та вхідним шаром обчислюється за допомогою рівняння

W_{hi} (нова) = W_{hi} (стара) + $\eta * \Delta_h * O_i + \alpha * [\delta W_{hi}$ (стара)].

Цей спосіб схожий на той, що використовувався для налаштування ваг у вихідному шарі.

Навчання за алгоритмом зворотного поширення.

Алгоритм навчання, який використовується в алгоритмі зворотного розповсюдження, показано нижче у вигляді наступних кроків:

1) ініціалізуємо ваги шарів невеликими випадковими значеннями,

2) вибираємо навчальний вектор (вхід і відповідний вихід),

3) поширюємо вхідні дані через мережу та обчислюємо фактичні вихідні дані при прямому поширенні,

4) обчислюємо похибки від різниці між фактичним значенням виходу і передбаченим,

5) зменшуємо функцію помилки, оновивши ваги у вихідному шарі та прихованому шарі на етапі зворотнього поширення,

6) переходимо до кроку 2 і повторюємо для наступної порції даних, поки похибка не стане прийнятно малою або не буде досягнуто максимальної кількості ітерацій (епох).

Завдяки простоті у використанні, вражаючій швидкості навчання та тренування штучних нейронних мереж (ШНМ), а також завдяки їхній відмінній здатності видобувати зміст зі складних даних і розпізнавати закономірності, окрім величезної здатності до прогнозування та фільтрації даних, все це зробило алгоритм навчання зі зворотним поширенням потужним інструментом і широко використовуваною технікою в навчанні та тренуванні штучних нейронних мереж (ШНМ).

2. Фізично-поінформорвані нейромережі (PINN).
2.1. Автоматичне диференціювання.

Автоматичне диференціювання (АД) є природним продовженням прагнення вчених та інженерів до механізації обчислень. Зрештою, ми вчимося обчислювати похідні, запам'ятовуючи набір правил. Чому тоді комп'ютери не можуть робити те саме? Проте, якщо просто дотримуватися правил і символічно розв'язувати похідні, це швидко призведе до значних обчислювальних труднощів. Проілюструємо це, розглянувши правило похідної від добутку двох функцій:

$$\frac{d}{dx}\big(f(x)g(x)\big) = \frac{d}{dx}\big(f(x)\big)g(x) + \frac{d}{dx}\big(g(x)\big)f(x).$$

Припустимо, що ми не проводимо жодних спрощень на цьому шляху. Просто дотримуючись правила добутку, ми перемножуємо всі спільні члени. По суті, це призводить до деревоподібної структури, де кількість членів збільшується експоненціально. У таблиці 2.1 наведено результати застосування символьного диференціювання за допомогою функції diff(f,x) MATLAB без спрощень. Зверніть увагу, як різко зростає кількість доданків, а також повторення тих самих доданків у виразах похідних.

$f(x)$	$\frac{df(x)}{dx}$ з функції diff(f, x) у MATLAB
4(x-1)(x-2)	8x-12
4(x-1)(x-2)(x-3)	4(x - 2)(x - 3) + (4x - 4)(x - 2) + (4x - 4)(x - 3)
4(x-1)(x-2)(x-3)(x-4)	(4x - 4)(x - 2)(x - 3) + (4x - 4)(x - 2)(x - 4) + (4x - 4)(x - 3)(x - 4) + 4(x - 2)(x - 3) (x - 4)

Таблиця 2.1: Похідні від символьного диференціювання.

Як альтернативний варіант, можна спробувати знаходити похідні чисельно. Зрештою, у більшості випадків застосувань похідних нас не цікавлять форми похідних, а лише їх кінцеві значення. Як найпростіший варіант, можна розглянути метод кінцевих різниць. Це, по суті, чисельне наближення до визначення градієнтів. Нехай задано скалярну багатовимірну функцію $f: R^n \to R$, ми можемо апроксимувати градієнти

як: $\frac{\partial \mathrm{f}}{\partial x_i} \approx \frac{\mathrm{f}(x+he_\mathrm{i})-\mathrm{f}(x)}{h}$, де e_i – i-й одиничний вектор, а h – мала величина. Це так званий метод прямих різниць.

У цього методу є кілька проблем. По-перше, він вимагає O(n) обчислень для n-вимірних градієнтів, що є надмірним для нейронних мереж з мільйонами параметрів, що навчаються. Крім того, йому притаманні чисельні помилки. Зокрема, помилки округлення (помилки, спричинені використанням скінченно-розрядних представлень з плаваючою комою) та помилки усікання (різниця між аналітичними градієнтами та числовими градієнтами). При малих h домінують похибки округлення. При великих h домінують похибки усікання. Такі помилки можуть бути значними у погано обумовлених задачах, спричиняючи чисельну нестабільність.

З іншого боку, автоматичне диференціювання, дозволяє успішно розв'язати задачу обчислення похідних без недоліків символьного диференціювання та скінченних різниць. Ключова ідея автоматичного диференціювання полягає в декомпозиції обчислень на елементарні кроки, які формують так званий **слід обчислень,** і об'єднанні похідних кожного кроку разом за допомогою ланцюгового правила (правила диференціювання складених функцій). Завдяки використанню слідів обчислень, автоматичне диференціювання може виконувати не тільки обчислення в замкненій формі, але й оператори потоку керування, що використовуються в програмах.

Незалежно від фактично вибраного шляху, в кінці чисельні обчислення сформують оціночний слід, який може бути використаний для автоматичного диференціювання.

Сліди оцінювання

Щоб розкласти функції на елементарні кроки, нам потрібно побудувати їхні **сліди обчислень**. Можна вважати ці сліди як запис кроків, які необхідно виконати, щоб досягти кінцевого результату. Розглянемо приклад [1111] функції f: $\mathrm{R}^n \to \mathrm{R}^m$, де m=2 а n=1:

$y = [\sin(x_1/x_2) + x_1/x_2 - e^{x_2}][x_1/x_2 - e^{x_2}]$.

Визначимо наступні змінні:

$v_{i-n} = x_i, i = 1, \dots, n$ – вхідні змінні,

$v_i, i = 1, \dots, l$ – проміжні змінні, $y_{m-i} = v_{1-i}, i = m-1, \dots, 0$ – вихідні змінні.

Тепер обчислимо значення функції при x=1 .5 і x=2.5, і запишемо всі проміжні значення.

Проміжні варіанти.	**Вирази**	**Значення**
v_{-1}	x_1	1.5
v_0	x_2	0.5
v_1	v_{-1}/v_0	3.0000
v_2	$\sin(v_1)$	0.1411
v_3	$\exp(v_0)$	1.6487
v_4	$v_1 - v_3$	1.3513
v_5	$v_2 + v_4$	1.4924
v_6	$v_5 * v_4$	2.0167

Таблиця 2.2: Сліди оцінювання функції $y = [\sin(x_1/x_2) + x_1/x_2 - e^{x_2}][x_1/x_2 - e^{x_2}]$

v_6 − кінцева вихідна змінна y, яка дорівнює 2.0167. Цей обчислювальний слід часто називають списком Венгерта.Графічно ми також можемо представити цей слід як спрямований ациклічний граф, де змінні є вершинами графа, а ребра представляють алгебраїчні зв'язки.

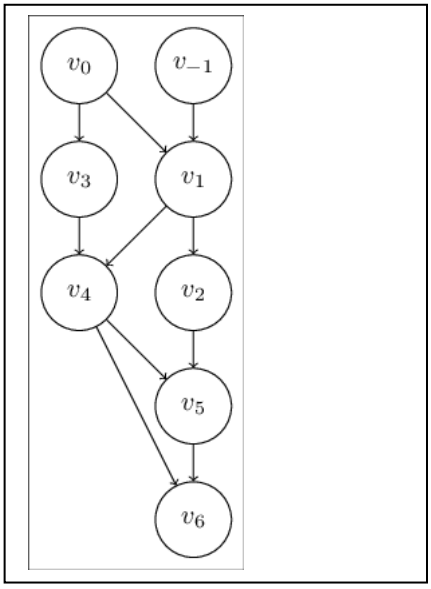


Рис. 2.3: Обчислювальний граф таблиці 2.2, для функції $y = [\sin(x_1/x_2) + x_1/x_2 - e^{x_2}][x_1/x_2 - e^{x_2}]$

Порівняно з табличним форматом, цей графік краще показує логічні залежності кожної з проміжних змінних вздовж сліду оцінювання. Наприклад, якщо ми хочемо обчислити v_5, ми повинні мати обчислені значення v_4 і v_2.

Логічним наступним кроком для знаходження похідних y по x_1 і x_2 є просте обчислення похідних по одній проміжній змінній за один раз, слідуючи за слідом обчислень. Це одразу приводить нас до алгоритму автоматичного диференціювання в прямому режимі.

Прямий режим. Накопичення сліду дотичної.

Припустимо, ми хочемо обчислити частинну похідну y по x_1 при $x_1 = 1.5$ і $x_2 = 0.5$. Ми можемо спробувати зробити це по одній проміжній змінній за раз. Зауважимо, що ми обчислюємо лише числове значення похідної. Для кожного v_i, ми обчислюємо $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$.

Спробуємо проробити для кількох змінних, щоб побачити, як це працює:

$$\dot{v}_{-1} = \frac{\partial x_1}{\partial x_1} = 1.0, \dot{v}_0 = \frac{\partial x_2}{\partial x_1} = 0,$$

$$\dot{v}_1 = \frac{\partial (v_{-1}/v_0)}{\partial x_1} = \dot{v}_{-1}(v_0^{-1}) + \dot{v}_0(-v_{-1}v_0^{-2}) = \frac{1.00}{0.50} = 2.00$$

$$\dot{v}_2 = \frac{\partial \big(\sin(v_1)\big)}{\partial x_1} = cos(v_1)\dot{v}_1 = -0.99 \times 2.00 = -1.98$$

Для цих обчислень ми просто застосовуємо правило диференціювання для складених функцій. Зауважимо, що $\dot{v}_i$ залежить лише від похідних та значень попередніх змінних. Тепер ми можемо доповнити таблицю 2 похідними.

Проміжні варіанти.	**Вирази**	**Значення**	**Похідні**
v_{-1}	x_1	1.5	1
v_0	x_2	0.5	0
v_1	v_{-1}/v_0	3.0000	2
v_2	$\sin(v_1)$	0.1411	-1.98
v_3	$\exp(v_0)$	1.6487	0
v_4	$v_1 - v_3$	1.3513	2
v_5	$v_2 + v_4$	1.4924	0.02
v_6	$v_5 * v_4$	2.0167	3.0118

Таблиця 2.4. Сліди оцінювання функції $y = [\sin(x_1/x_2) + x_1/x_2 - e^{x_2}][x_1/x_2 - e^{x_2}]$ разом з похідними за x_1

Значення проміжних змінних іноді називають первинним слідом, а значення похідних − дотичним слідом. Це добре узагальнюється до загальної векторно-значної функції виду $f: R^n \to R^m$. Припустимо, що ми намагаємося знайти значення функції при $x = a, a \in R^n$. У цьому випадку матриця Якобіана має вигляд:

$$J_f = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \Bigg|_{x=a}$$

а кожний стовпчик складається з

$$\dot{y}_j = \frac{\partial (y_j)}{\partial x_i} \Bigg|_{x=a}, j = 1, \ldots, m$$

які є частинними похідними від y_j за кожним x_i. Таким чином, ми можемо отримати стовпчики один за одним, задавши відповідні значення $\dot{x}_i = 1$ і зануливши інші записи. Це дозволяє застосувати деякі цікаві прийоми. Наприклад, якщо ми хочемо обчислити якобіан-векторний добуток з $\boldsymbol{r}$, ми можемо просто задати $\dot{x} = \boldsymbol{r}$ замість одиничного вектора:

$$J_f \boldsymbol{r} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix}.$$

З точки зору складності, прямий режим має порядок O(n), де n - розмірність вхідного вектора. Отже, він буде чудовим вибором для $f: R \to R^m$, але погано працювати для $f: R^n \to R$.

Одним з популярних способів реалізації прямого режиму автоматичного диференціювання є використання дуальних чисел. Дуальні числа були введені Вільямом Кліффордом у 1873 році в його статті "Попередній нарис бікватерніонів". Кліффорд вивчав, як розширити вектори Гамільтона, щоб врахувати обертання не лише навколо початку координат, але й довільних ліній у 3-вимірному просторі. Він назвав своє розширення векторів роторами, а суму таких

роторів - моторами. Потім він ввів символ ω для перетворення моторів у вектори, припустивши також, що $\omega^2 = 0$.

Однак, виявляється, що дуальними числами можна зручно представити операцію диференціювання. Формально дуальні числа можна подати у вигляді: $a + b\epsilon$. Вони підпорядковуються звичайному правилу покомпонентного додавання:

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon.$$

А їх множення працює подібно до множення комплексних чисел:

$$(a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon + db\epsilon^2 = ac + (ad + bc)\epsilon,$$

приймаючи до уваги, що $\epsilon^2 = 0$.

Зв'язок між дуальними числами та диференціюванням стає зрозумілим, коли ми розглянемо розклад в ряд Тейлора. Для довільної дійсної функції $f: R \to R$, ми можемо подати її розклад в ряд Тейлора у вигляді:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \mathrm{O}((x - x_0)^{n+1})$$

якщо функція диференційовна *n*+1разів і має обмежену *n*+1 похідну. Якщо ми розглянемо тепер дуальні числа у якості вхідних даних цього ряду, то отримаємо:

$$f(a + b\epsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a) b^n \epsilon^n}{n!} = f(a) + bf'(a)\epsilon.$$

Отже, якщо ми покладемо b=1, ми можемо отримати похідну в околі *x=a* , взявши коефіцієнт ϵ після обчислення:

$$f(a + b\epsilon) = f(a) + f'(a)\epsilon,$$

де ми розклали ряд Тейлора в околі точки a, а оскільки $\epsilon^2 = 0$, то всі члени, що включають ϵ^2 і вищі, зникають. Згідно з цим формулюванням, правило додавання буде наступним:

$$f(a + \epsilon) + g(a + \epsilon) = f(a) + f(a) + \big(f'(a) + g'(a)\big)\epsilon.$$

Правило множення буде наступним:

$$f(a + \epsilon)g(a + \epsilon) = f(a)g(a) + \big(f'(a)g(a) + g'(a)f(a)\big)\epsilon.$$

Правило диференціювання складених функцій (ланцюгове правило) запишеться у вигляді:

$$f\big(g(a + \epsilon)\big) = f(g(a) + g'(a)\epsilon) =$$

$$= f\big(g(a)\big) + f'\big(g(a)\big)g'(a)\epsilon.$$

Це, по суті, дає нам спосіб виконувати прямий режим автоматичного диференціювання: використовуючи дуальні числа, ми можемо отримати первісну і дотичну траєкторію одночасно. Отже, як ми узагальнити цей резщультат для випадку похідних старших за першу? Ми просто додаємо ϵ до кожної компоненти. Припустимо, що функція $f: R^n \to R$. Визначимо вектор $\boldsymbol{\epsilon} R^n$, такий, що $\epsilon_i^2 = \epsilon_i \epsilon_j = 0$. Якщо записати покомпонентні обчислення за рівнянням для $f(a+\epsilon)$, то одержимо, що:

$f(a+\epsilon) = f(a) + \nabla f(a) \cdot \epsilon.$

$\nabla f(a) \cdot \epsilon$ можна інтерпретувати як похідну за напрямком від f за напрямком ϵ. Додавання, множення та правило для складених функцій можна записати у вигляді:

$$f(a+\epsilon) + g(a+\epsilon) = f(a) + g(a) + \big(\nabla f(a) + \nabla g(a)\big)\epsilon,$$

$$f(a+\epsilon)g(a+\epsilon) = f(a)g(a) + \big(g(a)\nabla f(a) + f(a)\nabla g(a)\big)\epsilon,$$

$$f\big(g(a+\epsilon)\big) = f\big(g(a)\big) + \nabla f\big(g(a)\big)\nabla g(a)\epsilon.$$

Для $f: R^n \to R^m$, замість вектора $\boldsymbol{\epsilon}$, ми можемо використати матрицю $\boldsymbol{\epsilon}$. І для кожного рядка цієї матриці ми маємо дотримуємося того ж правила, що і для $f(a+\epsilon) = f(a) + \nabla f(a) \cdot \epsilon$.

Реверсний режим. Поширення з кінця.

Автоматичне диференціювання у зворотному режимі, також відоме як суміжний режим, обчислює похідну, рухаючись від кінця обчислювального сліду до початку. Розглянемо функцію $y = f\big(x(t)\big)$. З правила диференціювання складених функцій випливає, що:

$\frac{\partial y}{\partial t} = \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial t}.$

Два множники у правій частині можна розглядати як такі, що йдуть у зворотному напрямку: $\frac{\partial y}{\partial x}$ можна визначити, як тільки ми обчислимо *y* по *x*, а $\frac{\partial x}{\partial t}$ можна обчислити, к тільки ми обчислимо *x* по *t*. Поширюючи це правило на багатовимірні функції, отримаємо узагальнене правило диференціювання багатовимірних функцій: $D_a(f \circ g) = D_{g(a)}(f) D_a(g)$.

Застосувавши цю ідею до автоматичного диференціювання, отримаємо зворотний режим. Для обчислення похідних у цьому режимі потрібно зробити два проходи. Спочатку ми робимо прохід вперед, де отримуємо первинний слід (Таблиця 2.2). Потім ми поширюємо частинні похідні назад, щоб отримати шукані похідні (за правилом

диференціювання складених функцій). Повернемося до нашого прикладу, щоб побачити зворотний режим в дії. Знову розглянемо функцію $y = [\sin(x_1/x_2) + x_1/x_2 - e^{x_2}][x_1/x_2 - e^{x_2}]$.

Використовуючи той самий спосіб призначення проміжних змінних, що і в табл. 2 та на рис. 3, ми можемо сконструювати суміжний слід. Суміжний слід $\bar{v}_i$ визначається як:

$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$, що еквівалентне добутку всіх частинних від y_j до v_i. Почнемо з останньої змінної $v_6 = y$:

$$\bar{v}_6 = \frac{\partial y}{\partial v_6} = 1,$$

$$\bar{v}_5 = \frac{\partial y}{\partial v_5} = \bar{v}_6 \frac{\partial v_6}{\partial v_5} = 1 \times v_4 = 1.3513,$$

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} + \frac{\partial y}{\partial v_6}\frac{\partial v_6}{\partial v_4} = \bar{v}_5 \frac{\partial v_5}{\partial v_4} + \bar{v}_6 \frac{\partial v_6}{\partial v_4} = 1.3513 + 1.4914 = 2.8437,$$

Повторивши той самий процес для всіх проміжних змінних, ми отримаємо таблицю 2.5.

Проміжні варіанти.	**Значення**	**Суміжні значення**
v_{-1}	1.5	3.0118
v_0	0.5	-13.7239
v_1	3.0000	1.5059
v_2	0.1411	1.3513
v_3	1.6487	-2.8437
v_4	1.3513	2.8437
v_5	1.4924	1.3513
v_6	2.0167	1

Таблиця 2.5: Сліди оцінювання функції $y = [\sin(x_1/x_2) + x_1/x_2 - e^{x_2}][x_1/x_2 - e^{x_2}]$ разом з суміжними значеннями. (таблицю слід читати з нижнього рядка вгору).

Зверніть увагу на те, як ми змогли отримати суміжні значення до вхідних змінних x і y (які еквівалентні частинним похідним f за x і y) одночасно. Для функції $f: R^n \to R$, потрібно лише одне застосування зворотного режиму, щоб обчислити весь градієнт. Загалом, якщо розмірність вихідних даних значно менша за розмірність вхідних даних, зворотний режим є кращим вибором.

2.2. Основні принципи роботи та будівельні блоки PINN.

Глибокі нейромережі досягли успіху в таких завданнях, як комп'ютерний зір, обробка природної мови та теорія ігор. Глибоке навчання (ГН) змінило те, як виконуються завдання категоризації, розпізнавання образів і регресії в різних сферах застосування.

Глибокі нейронні мережі все частіше використовуються для розв'язання класичних прикладних задач математики, таких як диференціальні рівняння в частинних похідних (ДРЧП), з використанням підходів машинного навчання та штучного інтелекту. Деякі диференціальні рівняння в частинних похідних, як відомо, важко розв'язати за допомогою стандартних чисельних підходів. Наприклад, через значну нелінійність чи, наприклад, для рівнянь з домінуванням конвекції, які зазвичай мають сильно нерегулярні розв'язки (великі стрибки, розриви). Глибоке навчання нещодавно виникло як нова парадигма наукових обчислень завдяки універсальній апроксимації нейронних мереж та їхній великій експресивності. Нещодавні дослідження показали, що глибоке навчання є перспективним методом побудови мета-моделей для швидкого прогнозування динамічних систем. Зокрема, нейронні мережі виявилися здатними відображати основні нелінійні зв'язки між входом і виходом у складних системах. На жаль, робота з такими складними системами високої розмірності не позбавлена "прокляття розмірності", яке вперше описав Беллман у контексті задач оптимального керування. Однак алгоритми на основі машинного навчання є перспективними і для розв'язання ДРЧП.

Фізично-поінформовані нейронні мережі (PINN) - це науковий метод машинного навчання, який використовується для розв'язання задач, пов'язаних з диференціальними рівняннями в частинних похідних (ДРЧП). PINN апроксимують розв'язки ДРЧП, навчаючи нейронну мережу мінімізувати функцію втрат; вона включає члени, що відображають початкові та граничні умови вздовж границі просторово-часової області та залишок ДРЧП у вибраних точках області (так званих точках колокації). PINN − це мережі з глибоким навчанням, які, для будь-якої вхідної точки області інтегрування, після навчання знаходять наближений розв'язок диференціального рівняння в цій точці. Врахування в моделі залишкової мережі, яка кодує визначальні фізичні

рівняння, є значним нововведенням у PINN. Основна концепція навчання PINN полягає в тому, що його можна розглядати як машинне навчання без підкріплення, яке не потребує розмічених даних (labels), таких, як результати попередніх комп'ютерних моделювань або експериментів. Алгоритм PINN - це, по суті, безсіткова методика, яка знаходить розв'язки PDE шляхом перетворення задачі безпосереднього розв'язання визначальних рівнянь у задачу оптимізації функції втрат. Він працює шляхом інтеграції математичної моделі в мережу і посилення функції втрат залишковим членом з визначального рівняння, який діє як штрафний член для обмеження простору прийнятних рішень.

Фізично-поінформовані нейронні мережі можуть розв'язувати задачі, які описуються невеликою кількістю даних або зашумленими експериментальними спостереженнями. Оскільки вони можуть використовувати відомі дані, дотримуючись будь-якого фізичного закону, заданого загальними нелінійними диференціальними рівняннями в частинних похідних, PINN також можна вважати нейронними мережами, що базуються на алгоритмы навчання із підкріпленням. PINN дозволяють розв'язувати диференціальні рівняння, які в найзагальнішому вигляді можна записати як:

$$\begin{aligned} F\left(u\left(x_1,...,x_{d-1};t\right);\gamma\right) &= f\left(x_1,...,x_{d-1};t\right), \left(x_1,...,x_{d-1};t\right) \in \Omega, \\ B\left(u\left(x_1,...,x_{d-1};t\right);\gamma\right) &= g\left(x_1,...,x_{d-1};t\right), \left(x_1,...,x_{d-1};t\right) \in \partial\Omega, \end{aligned} \qquad (2.1)$$

визначені в області $\Omega \in \mathrm{R}^d$ з границею $\partial\Omega$.

Тут F - нелінійний диференційний оператор, а оскільки початкову умову фактично можна розглядати як умови типу граничної умови Діріхле в просторово-часовій області, відповідно B - оператор, що вказує на довільні початкові або граничні умови, пов'язані з задачею.

Рівняння (2.1) може описувати різноманітні фізичні системи, включаючи як прямі, так і обернені задачі. Метою прямої задачі є знаходження функції u для кожної точки $\left(x_1,...,x_{d-1};t\right)$, де γ − задані параметри. У випадку оберненої задачі, γ також має бути визначено з даних.

У методології PINN, $u\left(x_1,...,x_{d-1};t\right)$ обчислюється як передбачення нейромережі, параметризованої набором параметрів θ, що призводить до апроксимації:

$$\hat{u}_\theta(x_1,...,x_{d-1};t) \approx u(x_1,...,x_{d-1};t). \quad (2.2)$$

У цьому контексті, де прямі та обернені задачі аналізуються в одній і тій же структурі, і враховуючи, що PINN може адекватно розв'язувати обидві задачі, ми будемо використовувати θ для представлення як вектор всіх невідомих параметрів у нейронній мережі, що представляє сурогатну модель так і невідомих параметрів γ у випадку оберненої задачі.

У такому контексті нейромережа повинна навчитися апроксимувати диференціальні рівняння шляхом знаходження θ, які визначають нейромережеву модель шляхом мінімізації функції втрат, яка залежить від диференціального рівняння L_F, граничних умов L_B і, нарешті, деяких відомих даних L_{data}, кожні з яких відповідно зважені:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \left(\omega_F L_F(\theta) + \omega_B L_B(\theta) + \omega_d L_{data}(\theta)\right). \quad (2.3)$$

Загальна глибока нейронна мережа з L шарами може бути представлена як композиція L функцій $f_i(x_i,\theta_i)$, де x_i відомі як змінні стану, а θ_i позначає набір параметрів для i-го шару. Таким чином, функція $u(x)$ апроксимується як:

$$u_\theta(x) = f_L \circ f_{L-1} ... \circ f_1(x), \quad (2.4)$$

де кожна функція f_i визначена на двох просторах внутрішніх добутків E_i та H_i, причому $f_i \in E_i \times H_i$, а композицію шарів, позначену ◦, слід читати як $f_2 \circ f_1(x) = f_2(f_1(x))$.

З моменту створення оригінального "ванільного" варіанту PINN, більшість рішень використовували нейронні мережі прямого поширення. Однак деякі дослідники експериментували з різними типами нейронних мереж, щоб побачити, як вони впливають на загальну продуктивність PINN, зокрема із згортковими нейромережами (CNN), рекурентними нейромережами (RNN) і генеративно-змагальними нейромережами (GAN), і, схоже, ще не було повного дослідження інших мереж в рамках PINN.

Нейронна мережа прямого поширення (FFNN), також відома як багатошаровий персептрон (MLP), — це сукупність нейронів,

організованих у шари, які виконують обчислення послідовно через шари. Мережі прямого поширення — це глибокі нейромережі з декількома прихованими шарами, з рухом тільки вперед (без зворотного зв'язку). У повнозв'язній нейронній мережі нейрони в сусідніх шарах з'єднані, тоді як нейрони всередині одного шару не з'єднані.

Кожен нейрон - це математична операція, яка застосовує функцію активації до зваженої ї суми власних входів плюс коефіцієнт зсуву (bias).

Заданий вхід $x \in \Omega$ MLP перетворює на вихід через шар нейронів, які складаються з афінно-лінійних відображень між елементами (у послідовних шарах) та скалярних нелінійних функцій активації всередині елементів, що призводить до утворення композиції функцій. Таким чином, для MLP функцію f_i можна подати у вигляді:

$$f_i\left(x_i; W_i, b_i\right) = \alpha_i\left(W_i \cdot x_i + b_i\right), \tag{2.5}$$

поставивши у відповідність кожному E_i та H_i стандартний евклідів внутрішній добуток, тобто $E = H = R$, а α_i - скалярна (нелінійна) функція активації. Рівняння (2.4) можна також переписати у вигляді:

$$u_\theta = C_K \circ \alpha \circ C_{K-1} \ldots \circ \alpha \circ C_1\left(x\right), \tag{2.6}$$

де для кожного $1 \le k \le K$ задається $C_k\left(x_k\right) = W_k x_k + b_k$.

Таким чином, нейромережа складається з вхідного шару, вихідного шару та (K-2) прихованих шарів.

Використання нейронної архітектури додає методу PINN багато привабливих особливостей:

- Розв'язок за допомогою глибокої нейромережі є диференційованою, замкненою аналітичною формою, яку легко використовувати в будь-яких подальших обчисленнях. Більшість інших методів пропонують дискретний розв'язок (наприклад, предиктор-коректор або методи Рунге-Кутта) або розв'язок з обмеженою диференційованістю (наприклад, метод скінченних елементів).
- Такий розв'язок характеризується узагальнюючими властивостями нейронних мереж, які, як відомо, є найкращими.
- Необхідна кількість параметрів моделі набагато менша, ніж у будь-якому іншому методі розв'язування, а отже, отримуються

компактні моделі розв’язку, з дуже низькими вимогами до обсягу пам’яті.

- Метод є загальним і може бути застосований до ЗДР, систем ЗДР і до ДРЧП.
- Метод може бути реалізований апаратно, з використанням нейропроцесорів, і, отже, дає можливість розв’язувати в реальному часі складні задачі диференціальних рівнянь, що виникають у багатьох інженерних застосунках.

2.3. Бібліотеки, що реалізують технологію PINN. Бібліотека DeepXDE.

DeepXDE - це потужна бібліотека Python, призначена для вирішення математичних задач за допомогою методу фізично-обумовлених нейронних мереж (PINNs). Ця бібліотека надає гнучку і ефективну структуру для розв'язання диференціальних рівнянь в частинних похідних (ДРЧП) та інших математичних задач, поєднуючи техніки глибокого навчання з фізичними принципами. PINNs особливо корисні при розв'язанні задач, де основна фізика процесу відома, але рівняння складно сформулювати або розв'язати аналітично.

Основна ідея PINNs полягає в тому, щоб навчити нейронну мережу апроксимувати рішення ДРЧП, за допомогою вимоги, щоб вихід мережі відповідав як рівнянню (ДРЧП), так і граничним/початковим умовам. Цей підхід дозволяє навчитися рішенню безпосередньо з даних, що особливо корисно, коли доступно обмежена або зашумлена інформація.

DeepXDE надає зручний інтерфейс для визначення та розв'язання ПДР. Він підтримує широкий спектр ДРЧП, включаючи як часозалежні, так і часонезалежні проблеми, у одному, двох або трьох вимірах. Бібліотека також містить функціонал для обробки складних граничних умов, таких як періодичні, Неймана та Діріхле, Робіна, тощо.

Однією з ключових особливостей DeepXDE є її інтеграція з бібліотеками глибокого навчання, такими як TensorFlow та PyTorch, що дозволяє користувачам використовувати потужність цих бібліотек для тренування нейронних мереж. Ця інтеграція дозволяє ефективне тренування на GPU, що є важливим для розв'язання задач великих масштабів.

DeepXDE також надає засоби візуалізації результатів розв'язування ПДР, що полегшує аналіз та інтерпретацію результатів. Крім того, бібліотека включає приклади та навчальні матеріали, які допомагають користувачам розпочати розв'язання ДРЧП за допомогою PINN.

Окрім цього, DeepXDE є цінним інструментом для дослідників та інженерів, які працюють над математичним моделюванням та симуляцією. Його можливість поєднання глибокого навчання з моделюванням на основі фізики робить його добре пристосованим для

широкого спектра застосувань, від гідродинаміки до конструкційної механіки. Забезпечуючи простий у використанні інтерфейс та інтеграцію з популярними бібліотеками глибокого навчання, DeepXDE дозволяє користувачам ефективно розв'язувати складні математичні задачі та отримувати уявлення про основні фізичні процеси.

Особливості DeepXDE

У DeepXDE реалізовано багато алгоритмів і підтримується багато можливостей:

1) дозволяє компактизацію користувацького коду, можливість зробити його максимально наближеним до математичного формулювання,

2) складні геометрії областей без генерації великих сіток. Найпростішими типами геометрії є інтервал, трикутник, прямокутник, багатокутник, диск, еліпс, зірка, кубоїд, сфера, гіперкуб та гіперсфера. Інші геометричні фігури можна побудувати як конструктивну твердотільну геометрію (CSG) за допомогою трьох булевих операцій: об'єднання, різниці та перетину. DeepXDE також підтримує геометрію, представлену хмарою точок,

3) підтримка 5 типів граничних умов (ЗУ): Діріхле, Неймана, Робіна, періодичні та загальні, які можуть бути визначені на довільній області або на множині точок; а також наближені функції відстані для жорстких обмежень,

4) підтримка різних нейронних мереж: повнозв'язаних нейронних мереж (FNN), стекових нейрронних мереж FNN, залишкових нейронних мереж, (просторово-часових) багатомасштабних мереж Фур'є, тощо,

5) підтримка багатьох методів вибірки: рівномірна, псевдовипадкова, вибірка латинським гіперкубом, послідовність Гальтона, послідовність Хаммерслі та послідовність Соболя. Навчальні точки можуть залишатися незмінними під час навчання або передискретизуватися (адаптивно) через кожні певні ітерації,

6) 4 функціональні простори: степеневий ряд, поліном Чебишева, гаусівське випадкове поле (1D/2D),

7) паралельне навчання на декількох графічних процесорах.

8) можливість використання різних оптимізаторів, в тому числі і: найбільш популярних оптимізаторів для крайових задач − Adam та L-BFGS,

9) зручне збереження моделі під час навчання та завантаження натренованої моделі,

10) зворотні виклики для моніторингу внутрішніх станів та статистики моделі під час навчання: дострокова зупинка тощо,

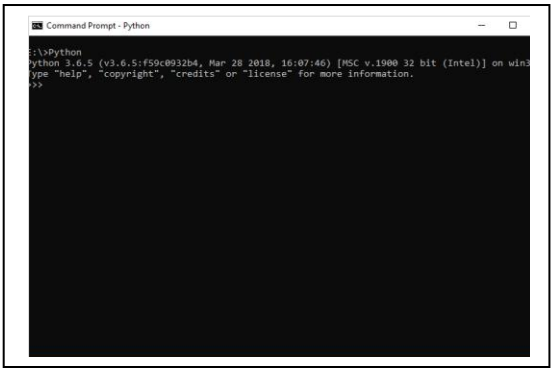
11) кількісна оцінка невизначеності за допомогою відсіву,

12) підтримка float16, float32 та float64,

А також DeepXDE підтримує багато інших корисних можливостей: різні (зважені) втрати, графіки швидкості навчання, метрики тощо.

Усі компоненти DeepXDE слабко пов'язані між собою, тому DeepXDE добре структурований і легко налаштовується. DeepXDE легко налаштувати під нові вимоги.

Одним з основних бек-ендів для бібліотеки DeepXDE є Tensorflow. Для встановлення TensorFlow важливо, щоб у системі був встановлений "Python". Python версії 3.4+ вважається найкращим для початку встановлення TensorFlow. Розглянемо наступні кроки для встановлення TensorFlow в операційній системі Windows.



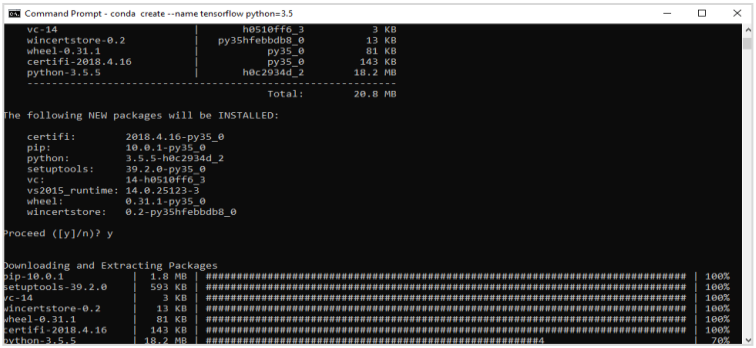
Крок 1: Перевірте версію python, яку ви встановлюєте.

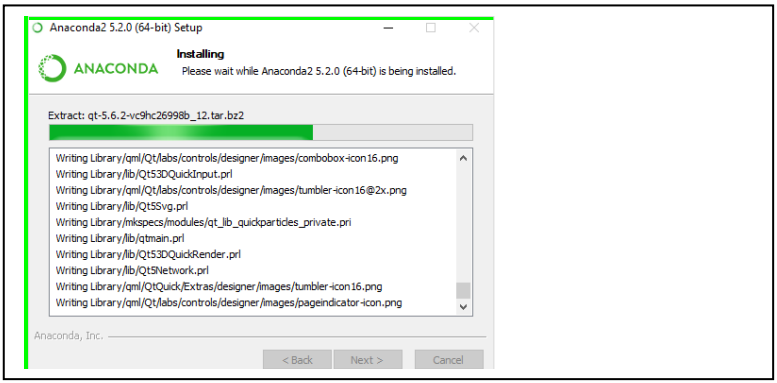
Крок 2: Користувач може обрати будь-який механізм для встановлення TensorFlow в систему. Ми рекомендуємо "pip" та "Anaconda". Pip - це команда, яка використовується для виконання та встановлення модулів у Python. Перш ніж встановити TensorFlow, нам потрібно встановити фреймворк Anaconda в нашу систему.

Крок 3: Виконайте наступну команду для ініціалізації встановлення TensorFlow:

```
conda create --name tensorflow python=3.5
```

Він завантажує необхідні пакети для встановлення TensorFlow.





Крок 4: Після успішного налаштування середовища важливо активувати модуль TensorFlow.

```
activate tensorflow
```

Крок 5: Використовуйте pip для встановлення "Tensorflow" у системі. Нижче наведено команду, яка використовується для встановлення:

```
pip install tensorflow
```

або

```
pip install tensorflow-gpu
```

3. Аналіз задач механіки за допомогою фізично-поінформованих нейромереж.

3.1. Задачі, що зводяться до звичайних диференціальних рівнянь.

Постановка задачі.

Розглянемо задачу прогину балки одиничної довжини:

$\frac{\mathrm{d}^4 u}{\mathrm{d}x^4} + 1 = 0, x \in [0,1],$

З двома граничними умовами на правому кінці:

$$u''(1) = 0, u'''(1) = 0,$$

однією умовою Дирихле на лівому кінці балки:

$$u(0) = 0$$

і однією умовою Неймана на лівому кінці балки:

$$u'(0) = 0.$$

Аналітичним розв'язком цієї задачі є наступний розв'язок:

$u(x) = -\frac{1}{24}x^4 + \frac{1}{6}x^3 - \frac{1}{4}x^2.$

Розв'язання задачі із використанням бібліотеки DeepXDE.

Перш за все, імпортуємо модуль DeepXDE і numpy:

```
import deepxde as dde
import numpy as np
```

Почнемо з визначення обчислювальної геометрії. Ми можемо використати вбудований клас `Interval` наступним чином:

```
geom = dde.geometry.Interval(0, 1)
```

Потім задаємо матрицю других частинних похідних (гессіан) та матрицю Якобіану для обчислення другої та третьої похідних відповідно:

```
def ddy(x, y):
  return dde.grad.hessian(y, x)
def dddy(x, y):
  return dde.grad.jacobian(ddy(x, y), x)
```

Далі, записуємо вираз для нев'язки оператора нашого рівняння:

```
def pde(x, y):
  dy_xx = ddy(x, y)
  dy_xxxx = dde.grad.hessian(dy_xx, x)
  return dy_xxxx + 1
```

Перший аргумент `pde` − це вхід мережі, тобто x-координата. Другий аргумент − це вихід мережі, тобто розв'язок $u(x)$, але тут ми використовуємо `y` як ім'я змінної.

Далі ми розглянемо ліву та праву граничні умови відповідно.

Зліва використовуються дві граничні умови, включаючи одну граничну умову Діріхле та одну граничну умову Неймана. Розташування лівої граничної умови визначається простою функцією Python. Функція повинна повертати значення `True` для тих точок, які задовольняють умовам $x = 0$ і `False` в іншому випадку (через помилки округлення часто доцільно використовувати `dde.utils.isclose` для перевірки еквівалентності двох значень з плаваючою точкою). У цій функції аргумент `x` для `boundary` є входом мережі і являє собою вектор розмірності d-dim, де d − розмірність і в даному випадку d=1. Потім як другий аргумент використовується булеве значення `on_boundary`. Якщо точка `x`, перший аргумент, знаходиться на межі області, в даному випадку, періодичної границі, коли вона досягає лівої кінцевої точки інтервалу, то `on_boundary` має значення `True`, інакше `on_boundary` має значення `False`.

```
def boundary_l(x, on_boundary):
  return on_boundary and dde.utils.isclose(x[0], 0)
```

Тепер прикладаємо дві граничні умови на правій границі. Розташування цих двох граничних умов визначається так, щоб функція повертала значення `True` для точок, що задовольняють умовам $x = 1$, і `False` в іншому випадку. Аргументи в цій функції подібні до аргументів функції `boundary_l`, з тією лише різницею, що в цьому випадку граничні умови загального оператора використовуються, коли він досягає правої кінцевої точки інтервалу.

```
def boundary_r(x, on_boundary):
  return on_boundary and dde.utils.isclose(x[0], 1)
```

Далі, визначаємо функцію, яка є точним розв'язком задачі:

```
def func(x):
  return -(x ** 4) / 24 + x ** 3 / 6 - x ** 2 / 4
```

Гранична умова Діріхле та гранична умова Неймана зліва визначаються наступним чином:

```
bc1 = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_l)
bc2 = dde.icbc.NeumannBC(geom, lambda x: 0, boundary_l)
```

Отже, ми визначили геометрію, нев’язку оператора та граничні умови задачі. Далі ми визначаємо функцію PDE нашої задачі у наступному вигляді:

```
data = dde.data.PDE(
    geom,
    pde,
    [bc1, bc2, bc3, bc4],
    num_domain=10,
    num_boundary=2,
    solution=func,
    num_test=100,
)
```

Число 10 - це кількість навчальних залишкових точок всередині області, а число 2 - кількість навчальних точок на границі. Аргумент `solution=func` є еталонним розв'язком для обчислення похибки нашого розв'язку і може бути проігнорований, якщо у нас немає еталонного розв'язку. Ми використовуємо 100 залишкових точок для тестування нев’язки оператора нашої задачі.

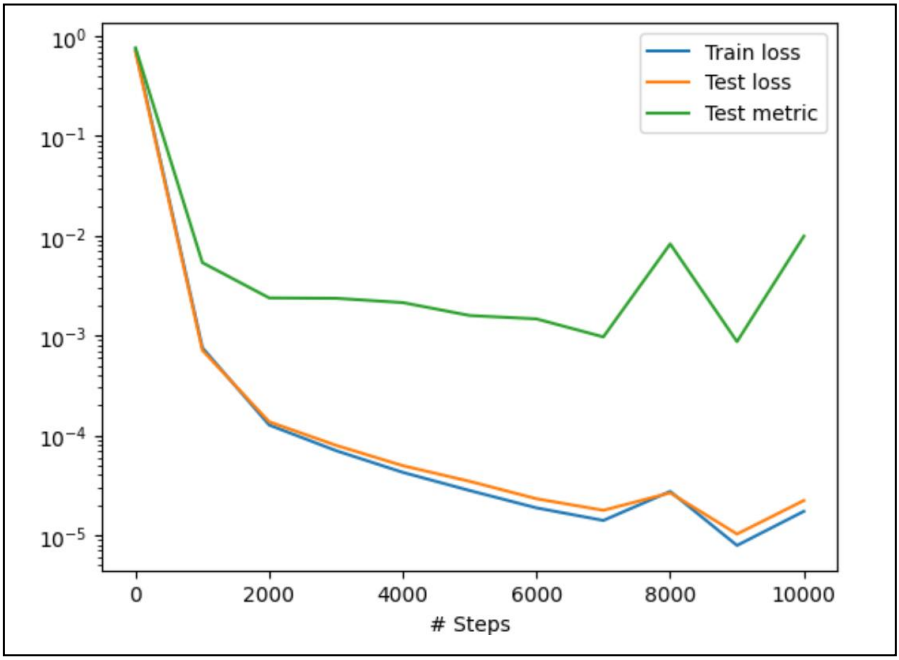


Рисунок 3.1. Графік функції втрат для тренувальної і тестувальної виборок

Далі ми обираємо тип нейромережі. Використаємо повнозв'язну нейронну мережу глибиною 4 (тобто з 3 прихованими шарами) і шириною 20:

```
layer_size = [1] + [20] * 3 + [1]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.FNN(layer_size, activation, initializer)
```

Отже, ми описали задачу в термінах функції PDE та відповідної нейромережі. Тепер побудуємо модель і виберемо відповідний метод оптимізації та задамо швидкість навчання:

```
model = dde.Model(data, net)
model.compile("adam", lr=0.001, metrics=["l2 relative error"])
```

Ми також обчислюємо відносну похибку L^2 як відповідну метрику під час навчання. Потім ми тренуємо модель протягом 10000 ітерацій (епох):

```
losshistory, train_state = model.train(iterations=10000)
```

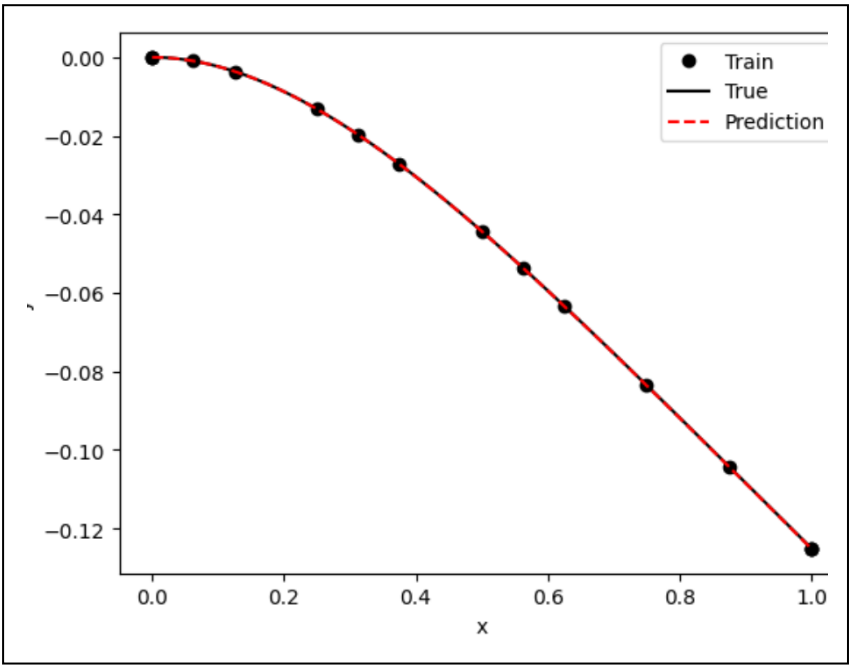


Рисунок. 3.2. Аналітичний розв’язок задачі і наближений розв’язок за допоиогою фізично-поінформованих нейромереж

3.2. Задачі, що зводяться до диференціальних рівнянь в частинних похідних та їх систем.

Постановка задачі.

Розглянемо наступну задачу поширення тепла в стержні:

$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, x \in [-1,1], t \in [0,1],$

де $\alpha = 0.4 -$ коефіцієнт теплопровідності.

На границі розглядуваної області задані умови Дирихле:

$u(0,t) = u(1,t) = 0,$

А в якості початкової умови вибрана періодична початкова умова:

$u(x,0) = \sin\left(\frac{n\pi x}{L}\right), 0 < x < L, n = 1,2, ...,$

де $L = 1$ – довжина стержня, а $n = 1$ – частота синусоїдних початкових умов.

Аналітичним розв'язком цієї задачі є:

$u(x,t) = e^{\frac{-n^2\pi^2\alpha t}{L^2}} \sin\left(\frac{n\pi x}{L}\right).$

Розв'язання задачі із використанням бібліотеки DeepXDE.

Перш за все, імпортуємо модуль DeepXDE:

```
import deepxde as dde
```

Потім визначаємо параметри рівняння:

```
a = 0.4
L = 1
n = 1
```

Далі ми визначаємо обчислювальну геометрію та часову область. Ми можемо використати вбудовані класи `Interval` та `TimeDomain` і об'єднати обидва домени за допомогою `GeometryXTime` наступним чином:

```
geom = dde.geometry.Interval(0, L)
timedomain = dde.geometry.TimeDomain(0, 1)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)
```

Далі ми запишемо вираз для нев'язки рівняння теплопровідності:

```
def pde(x, y):
    dy_t = dde.grad.jacobian(y, x, i=0, j=1)
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)
    return dy_t - a * dy_xx
```

Першим аргументом `pde` є двовимірний вектор, де перша компонента (x[:,0]) є x-координатою, а друга компонента (x[:,1]) є t-координатою. Другий аргумент - це вихід мережі, тобто розв'язок $u(x,t)$, але тут ми використовуємо `y` як ім'я змінної.

Далі ми розглядаємо граничну/початкову умову. `on_boundary` вибрано для того, щоб використовувати всю границю обчислювальної області як граничну умову. Ми включаємо простір `geomtime`, геометрію часу, створену вище, та `on_boundary` як граничні умови у функцію `DirichletBC` в DeepXDE. Ми також визначаємо початкову умову, яка є початковою умовою для рівняння Бюргерса, і використовуємо обчислювальну область, початкову функцію та `on_initial` для визначення початкової умови.

```
bc = dde.icbc.DirichletBC(geomtime, lambda x: 0, lambda _, on_boundary: on_boundary)
ic = dde.icbc.IC(
  geomtime,
  lambda x: np.sin(n * np.pi * x[:, 0:1] / L),
  lambda _, on_initial: on_initial,
)
```

Отже, ми визначили геометрію, нев'язку рівняння та граничні/початкові умови. Далі визначимо задачу `TimePDE` у вигляді:

```
data = dde.data.TimePDE(
  geomtime,
  pde,
  [bc, ic],
  num_domain=2540,
  num_boundary=80,
  num_initial=160,
  num_test=2540,
)
```

Число 2540 - це кількість навчальних залишкових точок, відібраних всередині області, а число 80 - кількість навчальних точок, відібраних на границі. Ми також включили 160 початкових залишкових точок для початкових умов.

Далі ми обираємо тип нейромережі. Тут ми використовуємо повнозв'язну нейронну мережу глибиною 4 (тобто з 3 прихованими шарами) і шириною 20:

```
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh", "Glorot normal")
```

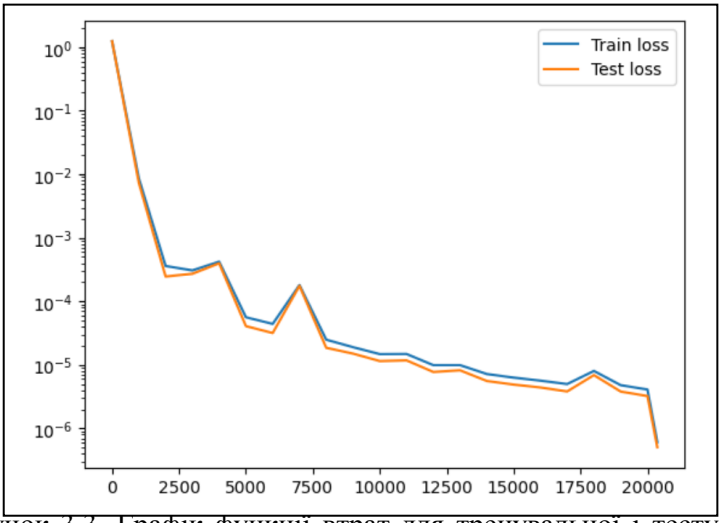


Рисунок 3.3. Графік функції втрат для тренувальної і тесту вальної виборок

Отже, ми маємо задачу для рівняння в частинних похідних та відповідну нейромережу. Тепер побудуємо модель і виберемо метод оптисізації та швидкість навчання:

```
model = dde.Model(data, net)
model.compile("adam", lr=1e-3)
```

Потім ми тренуємо модель протягом 20000 ітерацій:

```
losshistory, train_state = model.train(iterations=20000)
```

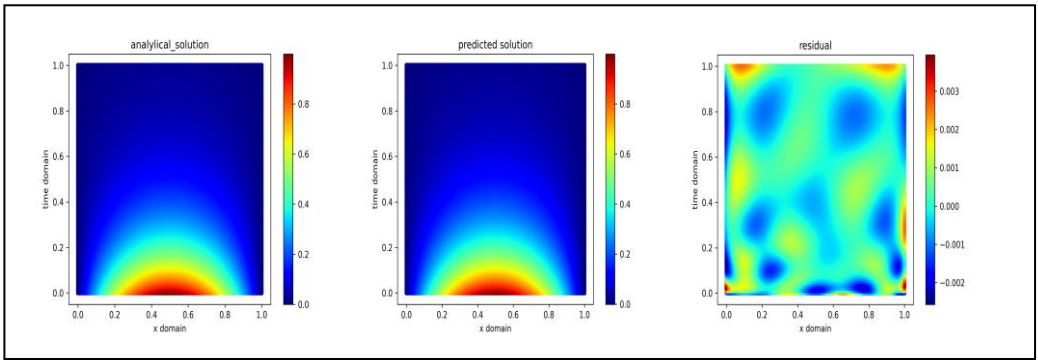


Рисунок 3.4. Аналітичний розв’язок задачі, наближений розв’язок за допоиогою фізично-поінформованих нейромереж та функція нев’язки розв’язку.

Після того, як ми навчили мережу за допомогою методу адаптивного моменту (ADAM), ми продовжуємо навчати мережу за допомогою методу квазіньютонівського методу Бройдела-Флетчера-Гольдфарба-Шанно з обмеженою пам’яттю (L-BFGS), щоб досягти ще менших втрат:

```
model.compile("L-BFGS-B")
losshistory, train_state = model.train().
```

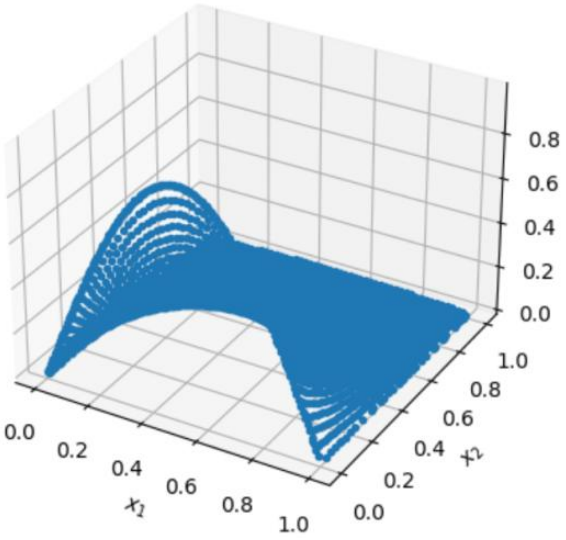


Рисунок 3.5. Наближений розв’язок задачі у просторі $t \times x$

3.3. Застосування фізично-поінформованих нейромереж до розв'язання двовимірної задачі теорії пружності.

Постановка задачі.

Розглянемо наступну двовимірну задачу лінійної теорії пружності:

$\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + f_x = 0, \frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + f_y = 0, x \in [0,1], y \in [0,1],$

де лінійна пружна модель задається наступним чином:

$\sigma_{xx} = (\lambda + 2\mu)\varepsilon_{xx} + \lambda\varepsilon_{yy}, \sigma_{yy} = (\lambda + 2\mu)\varepsilon_{yy} + \lambda\varepsilon_{xx}, \sigma_{xy} = 2\mu\varepsilon_{xy},$

з кінематичними співвідношеннями:

$\varepsilon_{xx} = \frac{\partial u_x}{\partial x}, \varepsilon_{yy} = \frac{\partial u_y}{\partial x}, \varepsilon_{xy} = \frac{1}{2}\left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x}\right).$

В квадратній області $[0,1] \times [0,1]$ задані масові сили вигляду:

$f_x = \lambda(4\pi^2 \cos(2\pi x) \sin(\pi y) - \pi \cos(\pi x) \mathrm{Q} y^3) +$

$+\mu(9\pi^2 \cos(2\pi x) \sin(\pi y) - \pi \cos(\pi x) \mathrm{Q} y^3),$

$f_y = \lambda(-3 \sin(\pi x)\, \mathrm{Q} y^2 + 2\pi^2 \sin(2\pi x) \cos(\pi y)) +$

$+\mu(-6 \sin(\pi x) \mathrm{Q} y^2 + 2\pi^2 \sin(2\pi x) \cos(\pi y) + \pi^2 \sin(\pi x) \mathrm{Q} y^4/4)$

разом із граничними умовами по переміщеннях:

$\mathrm{u}_x(x, 0) = \mathrm{u}_x(x, 1) = 0,$

$\mathrm{u}_y(0, y) = \mathrm{u}_x(1, y) = \mathrm{u}_y(x, 0) = 0,$

і напруженнях:

$\sigma_{xx}(0, y) = 0, \sigma_{xx}(1, y) = 0,$

$\sigma_{yy}(x, 1) = (\lambda + 2\mu)\mathrm{Q} \sin(\pi x).$

Приймемо значення параметрів λ, μ і Q рівними:

$\lambda = 1, \mu = 0.5, \mathrm{Q} = 4.$

Точним розв'язком цієї задачі є:

$\mathrm{u}_x(x, y) = \cos(2\pi x) \sin(\pi y),$

$\mathrm{u}_y(x, y) = \sin(\pi x) \mathrm{Q} y^4/4.$

Розв'язання задачі із використанням бібліотеки DeepXDE.

Перш за все, імпортуємо модулі DeepXDE, NumPy and PyTorch:

```
import deepxde as dde
import numpy as np
import torch
```

Почнемо з визначення загальних параметрів задачі:

```
lmbd = 1.0
mu = 0.5
Q = 4.0
```

```
pi = torch.pi
```

Далі ми визначаємо обчислювальну геометрію. Ми можемо використати вбудований клас `Rectangle` наступним чином:

```
geom = dde.geometry.Rectangle([0, 0], [1, 1])
```

Далі ми розглянемо ліву, праву, верхню та нижню граничні умови. На лівій границі застосовуються дві граничні умови. Розташування граничних умов визначається простою функцією Python. Функція повинна повертати True для тих точок, які задовольняють умовам x[0]=0, і False в іншому випадку (з-за помилок округлення часто доцільно використовувати dde.utils.isclose для перевірки того, чи є два значення з плаваючою точкою еквівалентними). У наведених нижче функціях аргумент x на boundary є входом мережі і являє собою вектор d-dim, де d - розмірність і d=1 в даному випадку. Потім в якості другого аргументу використовується булева функція on_boundary. Якщо точка x (перший аргумент) знаходиться на межі геометрії, то `on_boundary` приймає значення True, інакше `on_boundary` приймає значення `False`.

```
def boundary_left(x, on_boundary):
return on_boundary and dde.utils.isclose(x[0], 0.0)
```

Дві граничні умови застосовуються на правій границі. Подібно до `boundary_left`, розташування цих двох граничних умов визначається таким чином, щоб функція повертала `True` для точок, що задовольняють x[0]=1, і `False` в іншому випадку.

```
def boundary_right(x, on_boundary):
return on_boundary and dde.utils.isclose(x[0], 1.0)
```

На верхній границі задано дві граничні умови:

```
def boundary_top(x, on_boundary):
return on_boundary and dde.utils.isclose(x[1], 1.0)
```

На нижній границі задано також дві граничні умови:

```
def boundary_bottom(x, on_boundary):
return on_boundary and dde.utils.isclose(x[1], 0.0)
```

Далі, для можливості порівняння, ми визначаємо функцію, яка є точним розв'язком задачі:

```
def func(x):
    ux = np.cos(2 * np.pi * x[:, 0:1]) * np.sin(np.pi * x[:, 1:2])
    uy = np.sin(pi * x[:,0:1]) * Q * x[:, 1:2] ** 4 / 4

    E_xx = -2 * np.pi * np.sin(2 * np.pi * x[:,0:1]) * np.sin(np.pi * x[:,1:2])
    E_yy = np.sin(pi * x[:,0:1]) * Q * x[:,1:2] ** 3
    E_xy = 0.5 * (
        np.pi * np.cos(2 * np.pi * x[:,0:1]) * np.cos(np.pi * x[:,1:2])
        + np.pi * np.cos(np.pi * x[:,0:1]) * Q * x[:,1:2] ** 4 / 4
```

```
    )

    Sxx = E_xx * (2 * mu + lmbd) + E_yy * lmbd
    Syy = E_yy * (2 * mu + lmbd) + E_xx * lmbd
    Sxy = 2 * E_xy * mu

    return np.hstack((ux, uy, Sxx, Syy, Sxy))
```

Умови переміщення на границі визначаються наступним чином:

```
ux_top_bc = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_top, component=0)
ux_bottom_bc = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_bottom, component=0)
uy_left_bc = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_left, component=1)
uy_bottom_bc = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_bottom, component=1)
uy_right_bc = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_right, component=1)
```

Аналогічно, граничні умови по напруженнях визначаються як:

```
sxx_left_bc = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_left, component=2)
sxx_right_bc = dde.icbc.DirichletBC(geom, lambda x: 0, boundary_right, component=2)
syy_top_bc = dde.icbc.DirichletBC(
    geom,
    lambda x: (2 * mu + lmbd) * Q * np.sin(pi * x[:,0: 1]),
    boundary_top,
    component=3,
)
```

Далі, ми задаємо масові сили:

```
def fx(x):
    return (
        -lmbd
        * (
            4 * pi**2 * torch.cos(2 * pi * x[:,0:1]) * torch.sin(pi * x[:,1:2])
            - Q * x[:,1:2] ** 3 * pi * torch.cos(pi * x[:,0:1])
        )
        - mu
        * (
            pi**2 * torch.cos(2 * pi * x[:,0:1]) * torch.sin(pi * x[:,1:2])
            - Q * x[:,1:2] ** 3 * pi * torch.cos(pi * x[:,0:1])
        )
        - 8 * mu * pi**2 * torch.cos(2 * pi * x[:,0:1]) * torch.sin(pi * x[:,1:2])
    )

def fy(x):
    return (
        lmbd
        * (
            3 * Q * x[:,1:2] ** 2 * torch.sin(pi * x[:,0:1])
            - 2 * pi**2 * torch.cos(pi * x[:,1:2]) * torch.sin(2 * pi * x[:,0:1])
        )
        - mu
```

```
    * (
      2 * pi**2 * torch.cos(pi * x[:,1:2]) * torch.sin(2 * pi * x[:,0:1])
      + (Q * x[:,1:2] ** 4 * pi**2 * torch.sin(pi * x[:,0:1])) / 4
    )
    + 6 * Q * mu * x[:,1:2] **2 * torch.sin(pi * x[:,0:1])
  )
```

Далі ми виразимо нев'язки рівняння рівноваги та закону Гука:

```
def pde(x, f):
  E_xx = dde.grad.jacobian(f, x, i=0, j=0)
  E_yy = dde.grad.jacobian(f, x, i=1, j=1)
  E_xy = 0.5 * (dde.grad.jacobian(f, x, i=0, j=1) + dde.grad.jacobian(f, x, i=1, j=0))
  S_xx = E_xx * (2 * mu + lmbd) + E_yy * lmbd
  S_yy = E_yy * (2 * mu + lmbd) + E_xx * lmbd
  S_xy = E_xy * 2 * mu
  Sxx_x = dde.grad.jacobian(f, x, i=2, j=0)
  Syy_y = dde.grad.jacobian(f, x, i=3, j=1)
  Sxy_x = dde.grad.jacobian(f, x, i=4, j=0)
  Sxy_y = dde.grad.jacobian(f, x, i=4, j=1)
  momentum_x = Sxx_x + Sxy_y - fx(x)
  momentum_y = Sxy_x + Syy_y - fy(x)
  stress_x = S_xx - f[:, 2:3]
  stress_y = S_yy - f[:, 3:4]
  stress_xy = S_xy - f[:, 4:5]
  return [momentum_x, momentum_y, stress_x, stress_y, stress_xy]
```

Перший аргумент pde - це вхід мережі, тобто координати x та y. Другий аргумент - це вихід мережі, тобто розв'язок $u_x(x, y)$, але тут ми використовуємо f як ім'я змінної. Отже, ми визначили геометрію, нев'язку диференційного рівняння в частинних похідних та граничні умови. Далі ми визначаємо задачу у вигляді:

```
    data = dde.data.PDE(
      geom,
      pde,
      [
        ux_top_bc,
        ux_bottom_bc,
        uy_left_bc,
        uy_bottom_bc,
        uy_right_bc,
        sxx_left_bc,
        sxx_right_bc,
        syy_top_bc,
      ],
      num_domain=500,
      num_boundary=500,
      solution=func,
```

```
    num_test=100,
    )
```

Число 500 - це кількість навчальних залишкових точок всередині області, а число 500 - це кількість навчальних точок на границі. Аргумент `solution=func` є еталонним розв'язком для обчислення похибки нашого розв'язку, і може бути проігнорований, якщо у нас немає еталонного розв'язку. Використаємо 100 залишкових точок для тестування нев’язки ДРЧП.

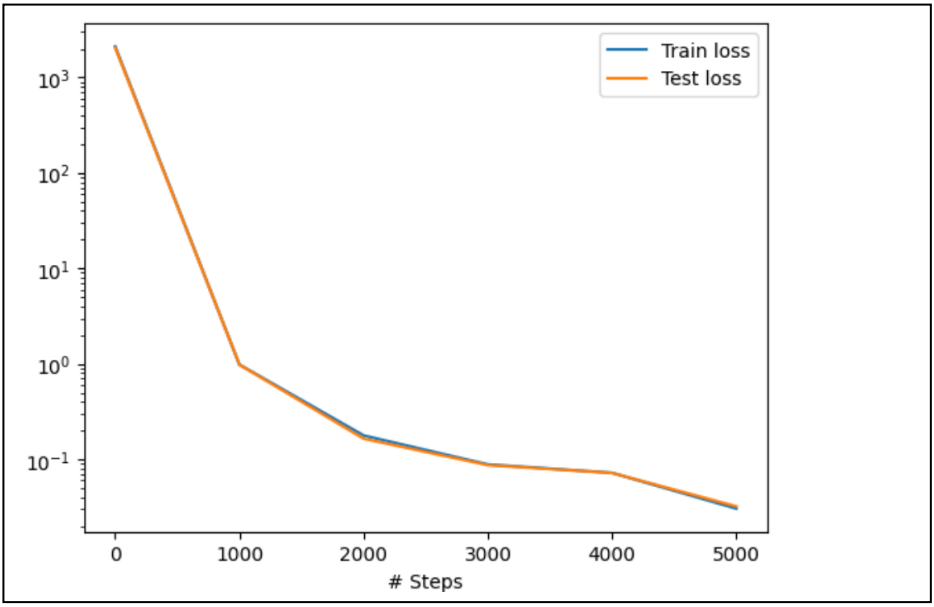


Рисунок 3.6. Графік функції втрат для тренувальної і тесту вальної виборок

Далі ми обираємо мережу. Використаємо 5 повно-зв'язних незалежних нейронних мереж глибиною 5 (тобто з 4 прихованими шарами) і шириною 40:

```
layers = [2, [40] * 5, [40] * 5, [40] * 5, [40] * 5, 5]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.PFNN(layers, activation, initializer)
```

Отже, ми описали крайову задачу, що зводиться до диференційних рівнянь в частиееих похідних та відповідну нейромережу. Залишилося

побудувати модель і вибрати метод оптимізації (виберемо метод адаптивного моменту) та швидкість навчання:

```
model = dde.Model(data, net)
model.compile("adam", lr=0.001)
```

Тренування моделі здійснюватимемо протягом 5000 ітерацій (епох):

```
losshistory, train_state = model.train(epochs=5000)
```

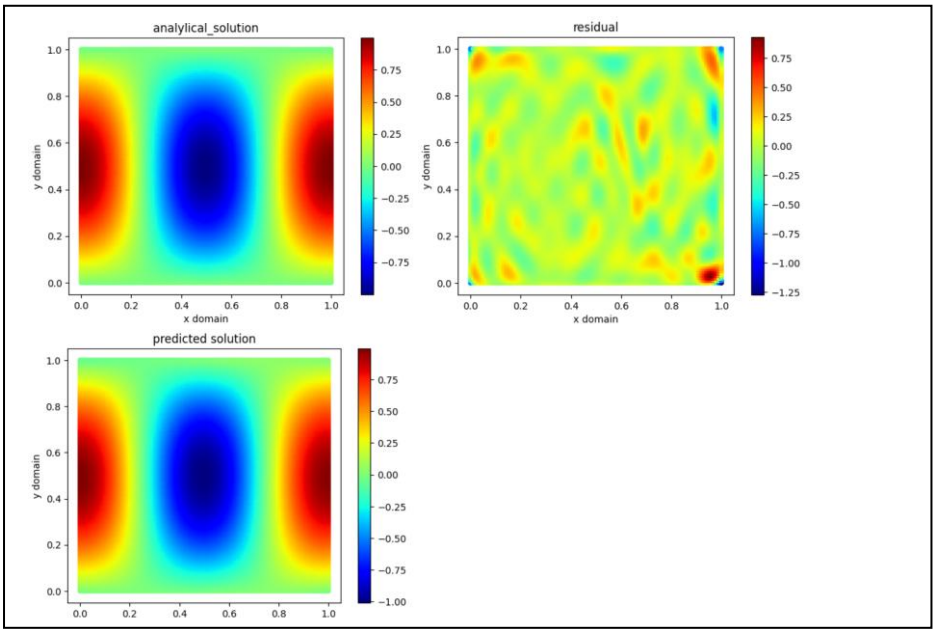


Рисунок 3.7. Аналітичний розв’язок задачі (переміщення u_x), наближений розв’язок за допоиогою фізично-поінформованих нейромереж та функція нев’язки розв’язку.

3.4. Застосування фізично-поінформованих нейромереж (PINN) до розв'язання крайових задач математичної фізики. Підхід Лагаріса.

В статті Лагаріса [7] представлено загальний метод розв'язання як звичайних диференціальних рівнянь (ЗДР), так і рівнянь з частинними похідними (РЧП), який спирається на можливості апроксимації функцій нейронними мережами із зворотним зв'язком і призводить до побудови розв'язку, записаного в диференційованій, замкненій аналітичній формі. Ця форма використовує нейронну мережу із зворотним зв'язком, як основний елемент апроксимації, параметри якої (ваги та зсуви) налаштовуються для мінімізації відповідної функції помилки. Для навчання мережі використовуються методи оптимізації, які, в свою чергу, вимагають обчислення градієнта похибки по відношенню до параметрів мережі.

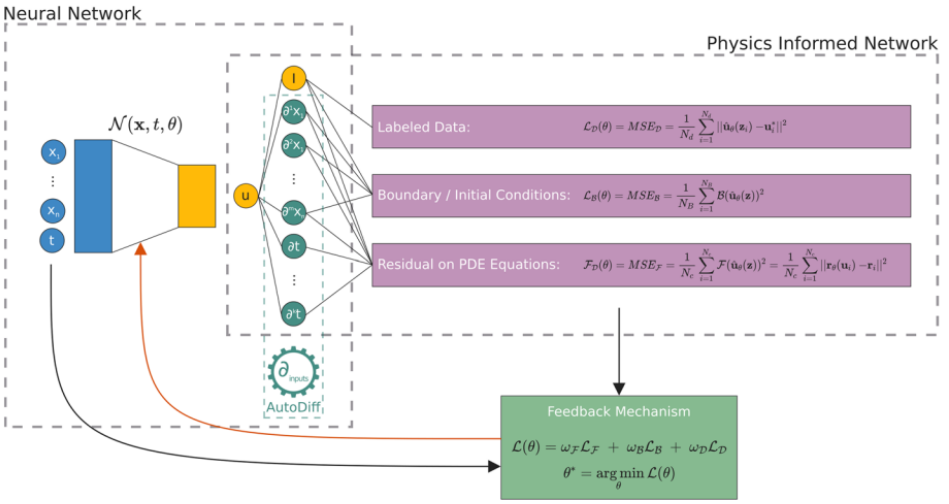


Рис. 3.8. Будівельні блоки PINN.

У підході Лагаріса функція моделі виражається як сума двох доданків: перший доданок задовольняє початковим/граничним умовам і не містить параметрів, які можна регулювати. Другий доданок включає в себе нейронну мережу прямого поширення, яку потрібно навчити

задовольняти диференціальне рівняння. Для ілюстрації підходу Лагаріс розглядає ДРЧП відносно двох змінних. Проте цей підхід легко розширюється на випадок і більшої кількості змінних. Для випадку рівняння Пуассона матимемо:

$$\frac{\partial^2\Psi(x,y)}{\partial x^2}+\frac{\partial^2\Psi(x,y)}{\partial y^2}=f(x,y),\quad x\in[0,1],y\in[0,1], \tag{3.1}$$

на границі виконуються умови Дирихле:

$$\Psi(0,y)=f_0(y),\Psi(1,y)=f_1(y) \text{ та } \Psi(x,0)=g_0(x),\Psi(x,1)=g_1(x). \tag{3.2}$$

Нейромережевий розв'язок підбирається у вигляді:

$$\Psi_t(x,y)=A(x,y)+x(1-x)y(1-y)N(x,y,\vec{p}), \tag{3.3}$$

де $A(x,y)$ вибирається таким чином, щоб задовольнити граничні умови, а саме:

$$\begin{aligned}A(x,y) &= (1-x)f_0(y)+xf_1(y)+\\ &+(1-y)\left\{g_0(x)-\left[(1-x)g_0(0)+xg_0(1)\right]\right\}+\\ &+y\left\{g_1(x)-\left[(1-x)g_1(0)+xg_1(1)\right]\right\}.\end{aligned} \tag{3.4}$$

Для змішаних граничних умов вигляду:

$$\Psi(0,y)=f_0(y),\Psi(1,y)=f_1(y),\Psi(x,0)=g_0(x),\frac{\partial\Psi(x,1)}{\partial y}=g_1(x) \tag{3.5}$$

(тобто коли на одній частині границі задані умови Дирихле, а на іншій - Неймана), пробний розв'язок можна записати у вигляд

$$\Psi_t(x,y)=B(x,y)+x(1-x)y\left[N(x,y,\vec{p})-N(x,1,\vec{p})-\frac{\partial N(x,1,\vec{p})}{\partial y}\right] \tag{3.6}$$

а функцію $B(x,y)$ вибираємо таким чином, щоб задовольнити граничні умови:

$$\begin{aligned}B(x,y)&=(1-x)f_0(y)+xf_1(y)+g_0(x)-\left[(1-x)g_0(0)+xg_0(1)\right]+\\ &+y\left\{g_1(x)-\left[(1-x)g_1(0)+xg_1(1)\right]\right\}\end{aligned} \tag{3.7}$$

Зауважимо, що другий член пробного розв'язку ніяк не впливає на граничні умови, оскільки він зникає на частині границі, де накладаються ГУ Діріхле, а його частина із нормальною похідною обертається в нуль на частині границі, де накладаються ГУ Неймана. У всіх наведених вище

задачах нев'язка ДРЧП, яку потрібно мінімізувати, визначається за допомогою:

$$E[\vec{p}] = \sum_i \left\{ \frac{\partial^2 \Psi(x_i, y_i)}{\partial x^2} + \frac{\partial^2 \Psi(x_i, y_i)}{\partial y^2} - f(x_i, y_i) \right\}, \tag{3.8}$$

де (x_i, y_i) точки на $[0,1]\times[0,1]$.

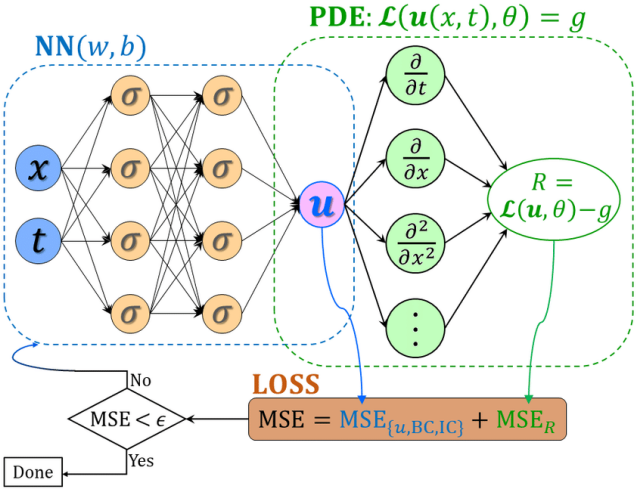


Рис. 3.9. Обчислювальна схема глибокої фізично-поінформованої нейронної мережі.

Нехай нам потрібно знайти розв'язок $u(\vec{x},\vec{t})$ крайової задачі

$$\begin{cases} L_1(\vec{x},\vec{t},u(\vec{x},\vec{t}),\nabla u(\vec{x},\vec{t}),\nabla^2 u(\vec{x},\vec{t})) = 0,\ \vec{x}\in D, \vec{t}\in T, \\ L_2(\vec{x},\vec{t},u(\vec{x},\vec{t}),\nabla u(\vec{x},\vec{t}))|_D = \varphi(\vec{x},\vec{t}), \\ u(\vec{x},\vec{t})|_{t=0} = f(\vec{x}), \\ \dfrac{\partial u(\vec{x},\vec{t})}{\partial t}|_{t=0} = F(\vec{x}). \end{cases} \tag{3.9}$$

Для розв'язання цієї крайової задачі можна використовувати неперервну чи дискретну за часом схему розв'язання. Ми будемо використовувати неперервну схему, згідно з якою для побудови розв'язку застосовується метод колокацій для дискретизації розглядуваної області D та її границі S. Задача зводиться до розв'язання системи рівнянь:

$$\begin{cases} L_1\left(\vec{x}_i,\vec{t}_j,u\left(\vec{x}_i,\vec{t}_j\right),\nabla u\left(\vec{x}_i,\vec{t}_j\right),\nabla^2 u\left(\vec{x}_i,\vec{t}_j\right)\right) = 0,\ \vec{x}\in\hat{D},\vec{t}\in\hat{T} \\ L_2\left(\vec{x}_i,\vec{t}_j,u\left(\vec{x}_i,\vec{t}_j\right),\nabla u\left(\vec{x}_i,\vec{t}_j\right)\right)\Big|_{\hat{D}} = \varphi\left(\vec{x}_i,\vec{t}_j\right) \\ u\left(\vec{x}_i,\vec{t}_j\right)\Big|_{\hat{T}} = f\left(\vec{x}_i\right) \\ \dfrac{\partial u\left(\vec{x}_i,\vec{t}_j\right)}{\partial t}\Big|_{\hat{T}} = F\left(\vec{x}_i\right). \end{cases} \tag{3.10}$$

Якщо $u_t\left(\vec{x},\vec{t},\vec{p}\right)$ – пробний розв'язок, що залежить від настроюваних параметрів $\vec{p}$, задача зводиться до задачі мінімізації функціоналу $\min_{\vec{p}} \sum_{\vec{x}_i\in\hat{D},\vec{t}_i\in\hat{T}} L_1\left(\vec{x}_i,\vec{t}_i,u_t\left(\vec{x}_i,\vec{t}_i\right),\nabla u_t\left(\vec{x}_i,\vec{t}_i\right),\nabla^2 u_t\left(\vec{x}_i,\vec{t}_i\right)\right)^2$ із обмеженнями, що накладаються граничними умовами. Це досягається за рахунок представлення шуканого розв'язку крайової задачі у вигляді: $u_t\left(\vec{x},\vec{t},\vec{p}\right) = A(\vec{x},\vec{t}) + F(\vec{x},\vec{t},N(\vec{x},\vec{t},\vec{p}))$, де функція $N(\vec{x},\vec{t},\vec{p})$ — нейронна мережа із параметрами $\vec{p}$ (ваги і зміщення) та n вхідними параметрами вхідних векторів $\vec{x}$ та $\vec{t}$, причому доданок $A(\vec{x},\vec{t})$ не містить настоюваних параметрів та задовольняє граничні і початкові умови, а другий доданок F будується таким чином, щоб обертатися на нуль у граничних та початкових умовах. Таким чином задача зводиться від початкової зв'язаної задачі оптимізації до незв'язаної.

Розглянемо спочатку крайову задачу для невідомої функції u, де на двох протилежних краях (наприклад, при $y=0$ та $y=1$) прямокутної області задані умови типу Неймана, а на двох інших – умови типу Дирихле. В цьому випадку ми можемо подати функцію u_t, що наближає шукану функцію u у наступному вигляді:

$$u_t(x,y) = B(x,y) + x(1-x)\Big[N(x,y,\vec{p}) - N(x,1-y,\vec{p}) - \\ - y\frac{\partial N(x,y,\vec{p})}{\partial y}\Big|_{y=1} + (1-y)\frac{\partial N(x,y,\vec{p})}{\partial y}\Big|_{y=0}\Big]. \tag{3.11}$$

Тоді її похідна за y матиме вигляд:

$$\frac{\partial u_t(x,y)}{\partial y}=\frac{\partial B(x,y)}{\partial y}+x(1-x)\left[\frac{\partial N(x,y,\vec{p})}{\partial y}+\frac{\partial N(x,1-y,\vec{p})}{\partial y}-\right.$$
$$\left.-\frac{\partial N(x,y,\vec{p})}{\partial y}\bigg|_{y=1}-\frac{\partial N(x,y,\vec{p})}{\partial y}\bigg|_{y=0}\right]. \qquad (3.12)$$

Цілком очевидно, що за умови, що функція $B(x,y)$ задовольняє граничним умовам на краях розглядуваної області, функція $u_t(x,y)$ в силу побудови також задовольнятиме цим самим умовам.

Не обмежуючи загальності вважатимемо, що функція задана на квадраті $[0,1]\times[0,1]$. Тоді, у випадку, коли умови Дирихле задані при $x=0$ та $x=1$, можна подати функцію $u_t(x,y)$ у вигляді:

$$u_t(x,y)=B(x,y)+y(1-y)\left[N(x,y,\vec{p})-N(1-x,y,\vec{p})-\right.$$
$$\left.-x\frac{\partial N(x,y,\vec{p})}{\partial x}\bigg|_{x=1}+(1-x)\frac{\partial N(x,y,\vec{p})}{\partial x}\bigg|_{x=0}\right]. \qquad (3.13)$$

В цьому випадку її похідна за x матиме вигляд:

$$\frac{\partial u_t(x,y)}{\partial x}=\frac{\partial B(x,y)}{\partial x}+y(1-y)\left[\frac{\partial N(x,y,\vec{p})}{\partial x}+\frac{\partial N(1-x,y,\vec{p})}{\partial x}-\right.$$
$$\left.-\frac{\partial N(x,y,\vec{p})}{\partial x}\bigg|_{x=1}-\frac{\partial N(x,y,\vec{p})}{\partial x}\bigg|_{x=0}\right]. \qquad (3.14)$$

Очевидно, що за умови, що функція $B(x,y)$ задовольняє граничним умовам на краях розглядуваної області, і в цьому випадку функція $u_t(x,y)$ в силу побудови також задовольнятиме цим самим умовам.

Підхід Лагаріса до побудови пробних функцій для крайової задачі із умовами Неймана на границі прямокутної області.

Розглянемо тепер випадок, коли потрібно відшукати невідому функцію u, де на кожній з пар протилежних країв (при $y=0$ та $y=1$, при $x=0$ та $x=1$) прямокутної області задані умови типу Неймана:

$$\begin{cases} \left.\dfrac{\partial u}{\partial y}\right|_{y=0} = g_0(x), \left.\dfrac{\partial u}{\partial y}\right|_{y=1} = g_1(x), \\ \left.\dfrac{\partial u}{\partial x}\right|_{x=0} = f_0(y), \left.\dfrac{\partial u}{\partial x}\right|_{x=1} = f_1(y). \end{cases} \tag{3.15}$$

В цьому випадку ми можемо подати функцію, що наближає шукану функцію u у наступному вигляді:

$$\begin{aligned} u_t(x,y) &= B(x,y) + f(x)\Big[N(x,y,\vec{p}) - N(x,1-y,\vec{p}) - \\ &- y\left.\frac{\partial N(x,y,\vec{p})}{\partial y}\right|_{y=1} + (1-y)\left.\frac{\partial N(x,y,\vec{p})}{\partial y}\right|_{y=0}\Big] + \\ &+ f(y)\Big[N(x,y,\vec{p}) - N(1-x,y,\vec{p}) - x\left.\frac{\partial N(x,y,\vec{p})}{\partial x}\right|_{x=1} + \\ &+ (1-x)\left.\frac{\partial N(x,y,\vec{p})}{\partial x}\right|_{x=0}\Big], \end{aligned} \tag{3.16}$$

де $f(x)$ — будь-яка гладка диференційована функція на відрізку [0,1], така, що обертається в нуль разом із своєю похідною в точках 0 та 1. Для простоти, шукатимемо цю функцію у вигляді поліному 4 порядку (оскільки маємо 4 умови: $f(0) = f(1) = f'(0) = f'(1) = 0$, $Ax^4 + Bx^3 + Cx^2 + Dx + E$.

З вказаних вище умов одержимо наступну систему лінійних алгебраїчних рівнянь для знаходження коефіцієнтів A, B, C, D, E:

$$\begin{cases} D = 0, E = 0, \\ A + B + C = 0, 4A + 3B + 2C = 0. \end{cases} \tag{3.17}$$

Система має безліч розв'язків, тому виберемо, наприклад $A = 1, B = -2, C = 1$, цьому розв'язку відповідає поліном $f(x) = x^2(1-x)^2$. Отже, остаточно ми можемо записати:

$$u_t(x,y) = B(x,y) + x^2(1-x)^2\left[N(x,y,\vec{p}) - N(x,1-y,\vec{p}) - \right.$$
$$\left. - y\frac{\partial N(x,y,\vec{p})}{\partial y}\bigg|_{y=1} + (1-y)\frac{\partial N(x,y,\vec{p})}{\partial y}\bigg|_{y=0}\right] +$$
$$+ y^2(1-y)^2\left[N(x,y,\vec{p}) - N(1-x,y,\vec{p}) - \right. \qquad (3.18)$$
$$\left. - x\frac{\partial N(x,y,\vec{p})}{\partial x}\bigg|_{x=1} + (1-x)\frac{\partial N(x,y,\vec{p})}{\partial x}\bigg|_{x=0}\right],$$

Розглянемо більш детально побудову функції $B(x,y)$.

Розглянемо випадок, коли функції $f_0(y), f_1(y), g_0(x), g_1(x)$ постійні f_0, f_1, g_0, g_1. Шукатимемо функцію $B(x,y)$ у вигляді: $B(x,y) = A + Bx + Cy + Dxy + Ex^2 + Fy^2$.

З граничних умов одержимо:

$$\frac{\partial B(x,y)}{\partial x}\bigg|_{x=0} = B + Dy = g_0, \frac{\partial B(x,y)}{\partial x}\bigg|_{x=1} = B + Dy + 2E = g_1,$$
$$\frac{\partial B(x,y)}{\partial y}\bigg|_{y=0} = C + Dx = f_0, \frac{\partial B(x,y)}{\partial y}\bigg|_{y=1} = C + Dx + 2F = f_1. \qquad (3.19)$$

Звідси знаходимо: $B = g_0, C = f_0, D = 0, E = \frac{g_1 - g_0}{2}, F = \frac{f_1 - f_0}{2}$, A не обмежуючи загальності покладемо рівною 0. Таким чином, в розглядуваному випадку шукана функція $B(x,y)$ матиме вигляд:

$$B(x,y) = g_0 x + f_0 y + \frac{g_1 - g_0}{2}x^2 + \frac{f_1 - f_0}{2}y^2. \qquad (3.20)$$

Підхід Лагаріса до побудови пробних функцій для крайової задачі плоскої теорії пружності.

Розглянемо тепер крайову задачу плоскої теорії пружності відносно двох невідомих функцій переміщень *u* і *v*, коли на кожному із країв прямокутної області задані граничні умови типу Неймана. Враховуючи, що напруження можуть бути виражені через поки що невідомі апроксимуючі функції $\Psi_1(x,y), \Psi_2(x,y)$, граничні умови на границі розглядуваної області можна подати у вигляді:

$$\begin{cases} (\lambda+2\mu)\dfrac{\partial\Psi_1}{\partial x}+\lambda\dfrac{\partial\Psi_2}{\partial y}\Big|_{x=0}=-p,(\lambda+2\mu)\dfrac{\partial\Psi_1}{\partial x}+\lambda\dfrac{\partial\Psi_2}{\partial y}\Big|_{x=1}=p, \\ (\lambda+2\mu)\dfrac{\partial\Psi_2}{\partial y}+\lambda\dfrac{\partial\Psi_1}{\partial x}\Big|_{y=0}=0,(\lambda+2\mu)\dfrac{\partial\Psi_2}{\partial y}+\lambda\dfrac{\partial\Psi_1}{\partial x}\Big|_{y=1}=0, \\ \mu\left(\dfrac{\partial\Psi_1}{\partial y}+\dfrac{\partial\Psi_2}{\partial x}\right)\Big|_{x=0}=0,\mu\left(\dfrac{\partial\Psi_1}{\partial y}+\dfrac{\partial\Psi_2}{\partial x}\right)\Big|_{x=1}=0, \\ \mu\left(\dfrac{\partial\Psi_1}{\partial y}+\dfrac{\partial\Psi_2}{\partial x}\right)\Big|_{y=0}=0,\mu\left(\dfrac{\partial\Psi_1}{\partial y}+\dfrac{\partial\Psi_2}{\partial x}\right)\Big|_{y=1}=0. \end{cases} \tag{3.21}$$

Шукатимемо невідомі функції $\Psi_1(x,y),\Psi_2(x,y)$ у вигляді:

$\Psi_1(x,y)=B_1(x,y)+F_1(N_1)$, $\Psi_2(x,y)=B_2(x,y)+F_2(N_2)$,

де $B_1(x,y),B_2(x,y)$ підбираються так, щоб задовольнити граничні умови, а $F_1(x,y),F_2(x,y)$ обертаються на нуль на межі розглядуваної прямокутної області ($x=0,x=1,y=0,y=1$).

Спираючись на результати, одержані для попередньої задачі, можемо записати:

$$\Psi_1(x,y)=B_1(x,y)+f(x)\left[N_1(x,y)-N_1(x,1-y)-y\frac{\partial N_1(x,1)}{\partial y}+\right.$$
$$\left.(1-y)\frac{\partial N_1(x,0)}{\partial y}\right]+f(y)\left[N_1(x,y)-N_1(1-x,y)-x\frac{\partial N_1(1,y)}{\partial x}+\right.$$
$$\left.+(1-x)\frac{\partial N_1(0,y)}{\partial x}\right],$$
$$\Psi_2(x,y)=B_2(x,y)+f(x)\left[N_2(x,y)-N_2(x,1-y)-y\frac{\partial N_2(x,1)}{\partial y}+\right.$$
$$\left.+(1-y)\frac{\partial N_2(x,0)}{\partial y}\right]+ \tag{3.22}$$
$$+f(y)\left[N_2(x,y)-N_2(1-x,y)-x\frac{\partial N_2(1,y)}{\partial x}+(1-x)\frac{\partial N_2(0,y)}{\partial x}\right].$$

Тут $f(t)=t^2(1-t)^2$, а $B_1(x,y),B_2(x,y)$ такі, що задовольняють (3.21):

$$\begin{cases} (\lambda+2\mu)\dfrac{\partial B_1}{\partial x}+\lambda\dfrac{\partial B_2}{\partial y}\Big|_{x=0}=-p,(\lambda+2\mu)\dfrac{\partial B_1}{\partial x}+\lambda\dfrac{\partial B_2}{\partial y}\Big|_{x=1}=p, \\ (\lambda+2\mu)\dfrac{\partial B_2}{\partial y}+\lambda\dfrac{\partial B_1}{\partial x}\Big|_{y=0}=0,(\lambda+2\mu)\dfrac{\partial B_2}{\partial y}+\lambda\dfrac{\partial B_1}{\partial x}\Big|_{y=1}=0, \\ \mu\left(\dfrac{\partial B_1}{\partial y}+\dfrac{\partial B_2}{\partial x}\right)\Big|_{x=0}=0,\mu\left(\dfrac{\partial B_1}{\partial y}+\dfrac{\partial B_2}{\partial x}\right)\Big|_{x=1}=0, \\ \mu\left(\dfrac{\partial B_1}{\partial y}+\dfrac{\partial B_2}{\partial x}\right)\Big|_{y=0}=0,\mu\left(\dfrac{\partial B_1}{\partial y}+\dfrac{\partial B_2}{\partial x}\right)\Big|_{y=1}=0. \end{cases} \quad (4.23)$$

Причому ці функції не є ітерованими, – їх не потрібно переобчислювати на кожному кроці оптимізації. Визначимо ці функції для розглядуваного прикладу крайової задачі плоскої теорії пружності.

Шукатимемо їх у вигляді:

$$\begin{cases} B_1(x,y)=A_1x+B_1y+C_1xy+D_1x^2+E_1y^2, \\ B_2(x,y)=A_2x+B_2y+C_2xy+D_2x^2+E_2y^2. \end{cases} \quad (3.24)$$

Підставляючи (3.24) у (3.23), одержимо систему рівнянь для визначення невідомих $A_1,A_2,B_1,B_2,C_1,C_2,D_1,D_2,E_1,E_2$:

$$\begin{cases} (\lambda+2\mu)(A_1+C_1y)+\lambda(B_2+2E_2y)\big|_{x=0}=-p \\ (\lambda+2\mu)(A_1+C_1y+2D_1)+\lambda(B_2+C_2+2E_2y)\big|_{x=1}=p \\ (\lambda+2\mu)(B_2+C_2x)+\lambda(A_1+2D_1x)\big|_{y=0}=0 \\ (\lambda+2\mu)(B_2+C_2x+E_2)+\lambda(A_1+C_1+2D_1x)\big|_{y=1}=0 \\ \mu(B_1+2E_1y+A_2+C_2y)\big|_{x=0}=0 \\ \mu(B_1+C_1+2E_1y+A_2+C_2y+2D_2)\big|_{x=1}=0 \\ \mu(B_1+C_1x+A_2+2D_2x)\big|_{y=0}=0 \\ \mu(B_1+C_1x+2E_1x+A_2+C_2+2D_2x)\big|_{y=1}=0. \end{cases} \quad (3.25)$$

Розв’язуючи одержану систему рівнянь, остаточно одержимо:

$$\begin{cases} A_1 = \dfrac{-(\lambda+2\mu)p}{4\mu(\lambda+\mu)}, A_2 = -1, B_1 = 1, B_2 = \dfrac{p\lambda}{4\mu(\lambda+\mu)}, \\ C_1 = 0, C_2 = -\dfrac{p\lambda}{2\mu(\lambda+\mu)}, D_1 = \dfrac{(\lambda+2\mu)p}{4\mu(\lambda+\mu)}, \\ D_2 = 0, E_1 = \dfrac{p\lambda}{4\mu(\lambda+\mu)}, E_2 = 0. \end{cases} \tag{3.26}$$

Отже,

$$\begin{cases} B_1(x,y) = \dfrac{-(\lambda+2\mu)p}{4\mu(\lambda+\mu)}x + y + \dfrac{(\lambda+2\mu)p}{4\mu(\lambda+\mu)}x^2 + \dfrac{p\lambda}{4\mu(\lambda+\mu)}y^2, \\ B_2(x,y) = -x + \dfrac{p\lambda}{4\mu(\lambda+\mu)}y - \dfrac{p\lambda}{2\mu(\lambda+\mu)}xy. \end{cases} \tag{3.27}$$

Розглянемо, нарешті, більш загальний випадок, коли на границі області функції $f_0(y), f_1(y), g_0(x), g_1(x)$ — довільні функції відповідно координат x та y. Покажемо, що в цьому випадку можна сконструювати функцію $B(x,y)$ яка буде задовольняти крайовим умовам (3.21). Щодо іншого доданку в представленні шуканої функції переміщень, він, очевидно, не буде залежати від граничних умов в тому сенсі, що буде обертатися на нуль на межі розглядуваної області.

Отже, шукану функцію $B(x,y)$ представимо у вигляді суперпозиції чотирьох функцій $B_1(x,y)$, $B_2(x,y)$, $B_3(x,y)$, $B_4(x,y)$ таких, що:

$$\left.\frac{\partial B_i(x,y)}{\partial x}\right|_{x=0} = \delta_{i1} g_0(x), \left.\frac{\partial B_i(x,y)}{\partial x}\right|_{x=1} = \delta_{i2} g_1(x),$$

$$\left.\frac{\partial B_i(x,y)}{\partial y}\right|_{y=0} = \delta_{i3} f_0(y), \left.\frac{\partial B_i(x,y)}{\partial y}\right|_{y=1} = \delta_{i4} f_1(y). \tag{3.28}$$

Тобто задача знаходження функції $B(x,y)$, що задовольняє граничним умовам (3.21), зводиться до знаходження чотирьох функцій $B_1(x,y)$, $B_2(x,y)$, $B_3(x,y)$, $B_4(x,y)$, що задовольняють (3.28) та в силу лінійності задачі в сумі дають функцію $B(x,y)$:

$B(x,y)=\sum_{i=1}^{4}B_i(x,y)$. Для знаходження кожної з функцій $B_i(x,y)$, наприклад $B_1(x,y)$ можемо записати:

$$B_1(x,y)=A_0+\sum_{n=1}^{\infty}A_n\cosh\left(\frac{\pi nx}{a}\right)\cos\left(\frac{\pi ny}{b}\right), \tag{3.29}$$

де

$$A_n=\frac{2}{\pi n\sinh\left(\frac{\pi na}{b}\right)}\int_0^b f(y)\cos\left(\frac{\pi ny}{b}\right)dy. \tag{3.30}$$

Інші функції $B_i(x,y)$ визначаються аналогічно.

Підхід Лагаріса до побудови пробних функцій для динамічної задачі поздовжних коливань стержня.

Для демонстрації викладеної методики розглянемо задачу розрахунку напружено-деформованого стану для одновимірної динамічної задачі поздовжніх коливань закріпленого на лівому кінці та вільного на правому стержня. Крайова задача в цьому випадку набуде вигляду:

$$\frac{\partial^2u}{\partial t^2}=\frac{E}{\rho}\frac{\partial^2u}{\partial x^2},\ u(t,x)\big|_{t=0}=f(x),\frac{\partial u(t,x)}{\partial t}\big|_{t=0}=F(x),$$

$$u(t,x)\big|_{x=0}=0,\frac{\partial u(t,x)}{\partial x}\big|_{x=1}=0. \tag{3.31}$$

В цьому випадку ми можемо подати функцію u_t, що наближає шукану функцію u у наступному вигляді:

$$u_t(x,t)=B(x,t)+x^2(1-x)^2\left[N(x,t,\vec{p})-N(x,0,\vec{p})-\right.$$
$$\left.-t\frac{\partial N(x,t,\vec{p})}{\partial t}\bigg|_{t=0}\right]+t^2\left[N(x,t,\vec{p})-N(0,t,\vec{p})-x\frac{\partial N(x,t,\vec{p})}{\partial x}\bigg|_{x=1}\right] \tag{3.32}$$

Цілком очевидно, що за умови, що функція $B(x,t)$ задовольняє граничним умовам на краях розглядуваної області, функція $u_t(x,t)$ в силу побудови також задовольнятиме цим самим умовам.

Було розглянуто випадок поздовжних коливань стержня з аерогелю із початковим профілем зміщень $f(x)=\sin \pi x$ та початковим профілем швидкостей $F(x)=0.$

Результати розрахунку порівнювалися із аналітичним розв'язком цієї задачі:

$$u_a(x,t)=\frac{\Phi(x+ct)+\Phi(x-ct)}{2}+\frac{1}{2c}\int_{x-ct}^{x+ct}\Psi(s)ds, \qquad (3.33)$$

де $\Phi(x),\Psi(x)$ — періодичні функції із періодом 2, продовжені непарним чином за межі інтервалу [0,1].

Архітектура мережі була побудована наступним чином: 1 вхідний шар на 32 нейрони, 4 прихованих шари по 64 нейрони кожен та 1 вихідний шар на 32 нейрони.

В якості методу оптимізації використовувалася комбінація двох найбільш популярних та ефективних методів – адаптивного моменту (ADAM) та методу Бройдена – Флетчера – Гольдфарба – Шанно у модифікації із обмеженим використанням пам'яті (L-BFGS). Цілком прогнозовано використання комбінації ADAM+L–BFGS дало найбільш високу точність одержаних розв'язків у порівнянні із методом адаптивного моменту чи методом L–BFGS.

На рис. 2–5 наведено наближений та аналітичний розв'язок вибраної задачі для $c=0,3;0,9;1,5;3$ разом із нев'язкою диференційного оператора рівняння.

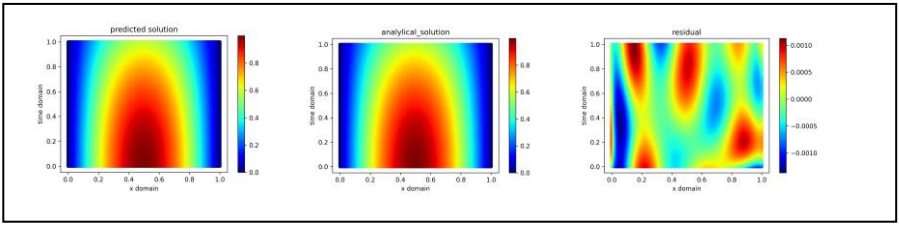


Рис 3.10. $c=0,3$

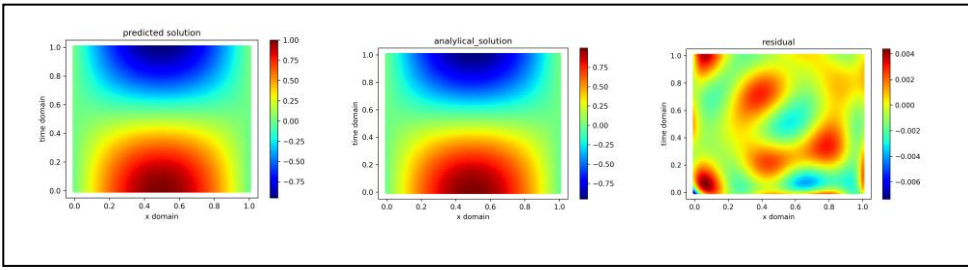


Рис. 3.11. $c = 0,9$

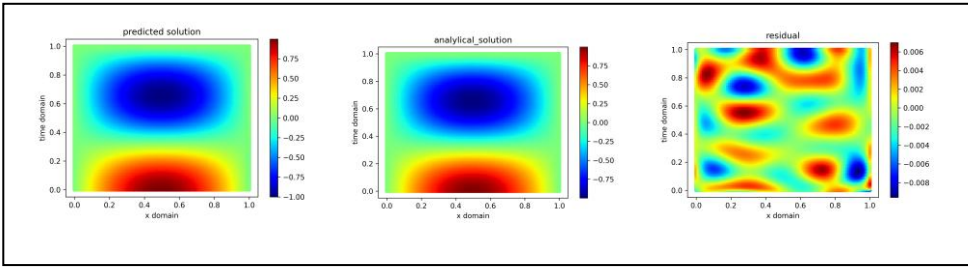


Рис. 3.12. $c = 1,5$

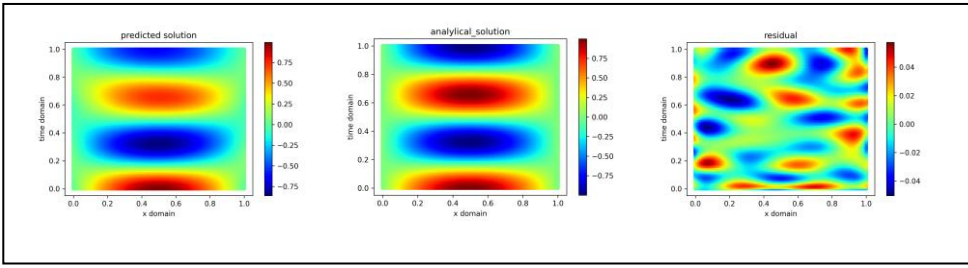


Рис. 3.13. $c = 3$.

Як видно з рис. 3.10–3.13, із збільшенням c $\left(\sqrt{\frac{E}{\rho}}\right)$ падає точність обчислень, і необхідно збільшувати кількість точок колокації та/або змінювати архітектуру нейромережі (збільшувати кількість нарів та кількість нейронів у шарах).

Список рекомендованої літератури.

1. Перестюк М.О., Маринець В.В. Теорія рівнянь математичної фізики. К.: Либідь, 2006. – 424 с.

2. Божидарник В.В., Сулим Г.Т. Елементи теорії пружності. — Львів: Світ, 1994. — 560 с.

3. Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. Automatic differentiation in machine learning: A survey // Journal of Machine Learning Research, – 2018. – 18(153), 1-43.

4. Griewank, Andreas, and Andrea Walther. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2008.

5. Blechschmidt, J., Ernst, O.G.: Three ways to solve partial differential equations with neural networks – A review. GAMM-Mitteilungen 44(2), (2021), https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.202100006

6. Karniadakis, G. E., & Perdikaris, P. Physics-informed neural networks: A seamless blend of physics and data // Nature Machine Intelligence, – 2020. – 2(7), 371-382.

7. Lagaris I., Likas A., Fotiadis D. Artificial neural networks for solving ordinary and partial differential equations // IEEE Transactions on Neural Networks. – 1998. – **9(5)**. – 987-1000. 10.1109/72.712178

8. Perdikaris, P., Raissi, M., & Karniadakis, G. E. Physics-informed neural networks: A comprehensive analysis of the contemporary approach and benchmarking of open-source solvers // Journal of Computational Physics, – 2021. – 423, 109798. doi: 10.1016/j.jcp.2020.109798

9. Raissi, M., Perdikaris, P., & Karniadakis, G. E. Physics-informed machine learning: Toward a new paradigm for solving physical inverse problems // Annual Review of Fluid Mechanics, – 2019. – 51, 557-577.

10. Raissi, M., Perdikaris, P., Karniadakis, G.E.: Physics-informed neural networks: A deep learning frame-work for solving forward and inverse problems involving nonlinear partial differential equations. J. Comput. Phys. 378, 686–707 (2019). https://doi.org/10.1016/j.jcp.2018.10.045, www.sciencedirect. com/science/article/pii/S0021999118307125