

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КІЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”



Перше видання

Машинне навчання

10 квітня 2024 р.



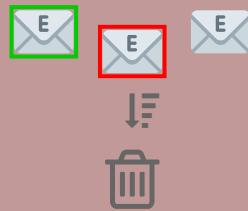
Штучний інтелект

Будь-яка техніка, яка дозволяє комп’ютерам імітувати поведінку людини



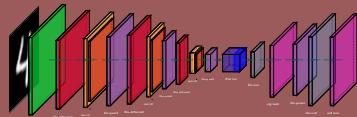
Машинне навчання

Можливість комп’ютера учитися не будучи явно запрограмованим



Глибинне навчання

Пошук шаблону в даних за допомогою нейронних мереж



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КІЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Кочура Ю. П., Гордієнко Ю. Г.

Машинне навчання

Занурення в машинне навчання

Навчальний посібник

Рекомендовано для здобувачів ступеня “магістр”, які навчаються за освітніми програмами «Комп’ютерні системи та мережі» спеціальність 123 «Комп’ютерна інженерія» та «Інженерія програмного забезпечення комп’ютерних систем» спеціальність 121 «Інженерія програмного забезпечення»

Електронне видання

Ми були б дуже вдячні за рекомендації та пропозиції щодо наповнення та покращення цього документа. Повідомлення про проблеми надсилайте через github: <https://github.com/dml-book/dml/issues>

ЗАТВЕРДЖЕНО
на засіданні кафедри обчислювальної техніки
протокол № x від dd.mm.yy

Перше видання. Версія 0.1.3

© 2024 Юрій Петрович Кочура
Юрій Григорович Гордієнко

Зміст

1 Вступ	4
1.1 Що таке штучний інтелект?	4
1.2 Що таке машинне навчання?	5
1.2.1 Навчання з учителем	6
1.2.2 Навчання без учителя	9
1.2.3 Навчання з частковим залученням учителя	10
1.2.4 Навчання з підкріпленням	10
1.3 Термінологія машинного навчання	11
1.3.1 Дані, що використовуються прямо та опосередковано	11
1.3.2 Первинні та акуратні дані	12
1.3.3 Навчальні та зарезервовані дані	13
1.3.4 Орієнтир	14
1.3.5 Конвеєр даних	14
1.3.6 Гіперпараметри та параметри	14
1.3.7 Навчання на основі моделі та екземплярів	14
1.3.8 Поверхневе та глибоке навчання	15
1.3.9 Навчання та оцінка	15
1.4 Становлення глибинного навчання	15
1.5 Біологічний та штучний нейрон	17
1.5.1 Функції активації	19
1.6 Функції втрат	23
1.6.1 Евклідові втрати	23
1.6.2 L1 втрати	24
1.6.3 Відстань Кульбака – Лейблера	24
1.6.4 Перехресна втрата ентропії: бінарна	25
1.6.5 Перехресна втрата ентропії: багатокласова	25
1.6.6 Перехресна втрата ентропії із декількома мітками	26
1.7 Метрики оцінки	26
Задача класифікації	26
Виявлення об'єктів	29
1.8 Мотивація та інтуїція	31
1.8.1 Інженіринг ознак	32
2 Стратегії навчання та ініціалізації	34
2.1 Пряме та зворотне поширення	34
2.1.1 Одновимірна регресія	34
Пряме поширення	35

Зворотне поширення	35
Оновлення параметрів	35
2.1.2 Багатовимірна регресія	35
Пряме поширення	36
Зворотне поширення	36
2.1.3 Лінійна двошарова мережа	36
Пряме поширення	37
Зворотне поширення	37
2.1.4 Нелінійна двошарова мережа	37
Пряме поширення	38
Зворотне поширення	39
2.2 Ініціалізація нейронних мереж	39
2.2.1 Важливість ефективної ініціалізації	39
2.2.2 Проблема зникнення та вибуху градієнтів	40
2.2.3 Пошук оптимальних значень для ініціалізації	42
2.2.4 Ініціалізація Ксав'є (Xavier)	43
2.3 Оптимізація параметрів нейронних мереж	46
2.3.1 Особливість оптимізації	46
2.3.2 Виклики оптимізації	47
Локальний мінімум	47
Сідлові точки	48
2.3.3 Опуклість	49
2.3.4 Властивості опуклих функцій	50
Локальний мінімум – глобальний мінімум	50
Другі похідні та опуклість	50
2.3.5 Пакетний градієнтний спуск	50
Одновимірний пакетний градієнтний спуск	51
Багатовимірний пакетний градієнтний спуск	54
2.3.6 Стохастичний градієнтний спуск	55
2.3.7 Міні-пакетний градієнтний спуск	56
2.3.8 Імпульс	57
2.3.9 Адаптивний градієнт (AdaGrad)	60
2.3.10 Середнє квадратичне поширення (RMSProp)	60
2.3.11 Адаптивна оцінка моментів (Adam)	61
2.4 Деякі методи регуляризації нейронних мереж	62
2.4.1 Рання зупинка (Early stopping)	63
2.4.2 L2 регуляризація	64
2.4.3 L1 регуляризація	65
2.4.4 Багінг (Bagging)	66
2.4.5 Дропаут (Dropout)	67
2.5 Перехресна перевірка	70
2.6 Зменшення розмірності	71
2.6.1 Метод головних компонент	72
Література	75

Розділ 1

Вступ

“Теоретично між теорією та практикою нема ніякої різниці. Але на практиці вона є.”

– Бенджамін Брюстер

1.1 Що таке штучний інтелект?

Термін “штучний інтелект” (англ. Artificial Intelligence, AI) був запропонований та прийнятий на літній конференції 1956 року [1] в коледжі Дартмут (місто Гановер, США), яку організували Джон Маккарти (John McCarthy), Марвін Мінський (Marvin Minsky), Натаніель Рочестер (Nathaniel Rochester) та Клод Шеннон (Claude Shannon). З того часу штучний інтелект прийнято розглядати як галузь науки. До напрямків досліджень в галузі штучного інтелекту відноситься: машинне навчання (Machine Learning, ML), робототехніка (Robotics), експертні системи (Expert Systems), обробка природної мови (Natural Language Processing, NLP), мовлення (Speech) та комп’ютерний зір (Computer Vision).

Штучний інтелект у широкому розумінні – будь-яка техніка, яка дозволяє комп’ютерам імітувати поведінку людини. У більш вузькому розумінні штучний інтелект – здатність інженерної системи¹ здобувати, обробляти та застосовувати знання та вміння [2].

На рисунку 1.1 показано машинне навчання як один з напрямків досліджень в галузі штучного інтелекту та одна з технік машинного навчання – глибинне навчання.



Рисунок 1.1 – Штучний інтелект, машинне та глибинне навчання

¹Інженерна система – поєднання взаємодіючих елементів, організованих для досягнення однієї або декількох поставлених цілей.

1.2 Що таке машинне навчання?

Машинне навчання (Machine Learning, ML) – це галузь штучного інтелекту, що вивчає розробку та застосування алгоритмів, які дозволяють комп’ютеру вчитися робити прогнози чи приймати рішення на основі вхідних даних не будучи при цьому явно запрограмованим. Ці дані можуть бути отримані з реального середовища за допомогою сенсорів, створені вручну або згенеровані іншим алгоритмом.

Машинне навчання також можна розглядати як процес вирішення деякого практичного завдання шляхом алгоритмічного навчання **статистичної моделі** на наборі даних, що характеризує шаблон поставленого завдання.

Комп’ютери та обчислення допомагають нам досягати більш складних цілей і кращих результатів у вирішенні поставлених задач, ніж ми могли б досягти самі. Однак, багато сучасних завдань вийшли за рамки просто обчислень через один основний обмежуючий фактор: традиційно, комп’ютери можуть дотримуватися лише конкретних вказівок/інструкцій, які їм дають.

Вирішення задач з програмування вимагає написання конкретних покрокових інструкцій, які має виконувати комп’ютер. Ми називаємо ці кроки алгоритмами. У цьому випадку, комп’ютери можуть допомогти нам там, де ми:

1. Розуміємо як вирішити задачу.
2. Можемо описати задачу за допомогою чітких покрокових інструкцій, які комп’ютер може зрозуміти.

Методи контролюваного машинного навчання дозволяють комп’ютерам “учитися” на прикладах. Вирішення задач із застосуванням машинного навчання вимагає виявлення деякого шаблону², а потім, коли такий шаблон готовий, дозволяють, наприклад, нейронні мережі вивчити карту переходів між вхідними та вихідними даними. Ця особливість відкриває нові типи задач, де комп’ютери можуть допомогти нам у їх розв’язанні, за умови, коли ми:

1. Визначили шаблон задачі.
2. Маємо достатньо даних, що ілюструють даний шаблон.

Принципова відмінність парадигм: класичного програмування (символьний штучний інтелект) та машинного навчання, показана на рисунку 1.2. Таким чином, для вирішення задачі за допомогою класичного програмування нам потрібно прописати чіткі покрокові інструкції нашого алгоритму. Далі, коли алгоритм реалізовано, ми можемо подати йому на вхід деякі дані, що характеризують окремий стан поставленої задачі, а на виході він згенерує нам деякі вихідні дані (відповіді). Коли мова йде про контрольоване машинне навчання, тут на вхід комп’ютерній програмі подаються приклади, що висвітлюють обидві сторони шаблону поставленої задачі: вхідні та очікувані вихідні дані (відповіді), а на виході отримуємо правило (алгоритм) за яким буде здійснюватись переход між вхідними та вихідними даними.

Для стисливості, далі будуть використовуватись терміни **навчання** та **машинне навчання** як синоніми. З тієї ж причини часто буде використовуватись термін **модель**, під яким будемо мати на увазі **статистичну модель**.

За характером навчальних даних навчання буває з учителем (контрольоване), без учителя (неконтрольоване), з частковим залученням учителя (напівконтрольоване) та з підкріпленням (див. рис. 1.3).

²Пошук прикладів, що висвітлюють обидві сторони шаблону: вхід і вихід.



Рисунок 1.2 — Відмінність класичного програмування від машинного навчання

1.2.1 Навчання з учителем

Термін “навчання з учителем” походить від ідеї учителя, який тренує машину для виконання певного завдання, надаючи їй правильні відповідь, після того, як машина обчислить свою власну. Надання правильних відповідей учителем здійснюється для того, щоб дати можливість машині зрозуміти, де були зроблені помилки. Отже, ідея навчання з учителем полягає в наступному: на основі прикладів вхідних даних та пов’язаних з ними результатів, потрібно вивчити правило прийняття рішень, яке надає найкращі показники відповідно до метрики, обраної для їх оцінки. Але що ми маємо на увазі під терміном “машинне навчання”? Том Мітчелл (1997) наводить загальне визначення машинного навчання в обчислювальній техніці: “Комп’ютерна програма, яка учається з досвіду E по відношенню до деякого класу задач T та міри продуктивності P , називається машинним навчанням, якщо її продуктивність у задачах з T , що вимірюється за допомогою P , покращується з досвідом E .“ У нашому контексті, задачами будуть, наприклад, класифікація або регресія, показник ефективності – деяка метрика, яку ми оберемо, а досвід відповідатиме прикладам з набору даних на яких нашому алгоритму доведеться учитися.

У цьому випадку ми працюємо з множиною даних, яку представлено наступним чином:

$$d = \{(\mathbf{X}^{(1)}, y^{(1)}), (\mathbf{X}^{(2)}, y^{(2)}), \dots, (\mathbf{X}^{(n)}, y^{(n)})\}, \quad (1.1)$$

де n – загальна кількість прикладів у наборі, $y^{(i)}$ – мітка i -го прикладу, як правило, $y^{(i)} \in \mathbb{R}$ або $y^{(i)} \in \mathbb{N}$.

Кожен елемент $\mathbf{X}^{(i)}$ множини (1.1) називається вектором вхідних ознак або прикладом. Вектор – це одновимірний масив. Одновимірний масив, у свою чергу, є упорядкованою та проіндексованою послідовністю значень. Довжина цієї послідовності, m , називається розмірністю вектора: $\mathbf{X}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_m^{(i)})$.

Вектор ознак – це вектор, у якому кожен елемент $j = 1, \dots, m$ містить значення, що характеризує приклад. Кожне таке значення називається ознакою та позначається x_j . Запис $x_j^{(i)}$ означає, що ми розглядаємо ознакою j для i -го прикладу.

Мітка $y^{(i)}$ може бути або елементом кінцевої множини класів $\{1, 2, \dots, C\}$ або дійсним числом, або складнішою структурою, такою як вектор, матриця, дерево чи граф. У задачах класифікації, клас – категорія даних, до якої належить розглянутий приклад. Скажімо, якщо прикладами є повідомлення електронної пошти, а наше

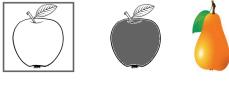
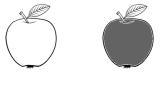
Контрольоване навчання Supervised learning	Напівконтрольоване навчання Semi-supervised learning	Неконтрольоване навчання Unsupervised learning	Навчання з підкріплення Reinforcement learning
Дані: (X, y) X – приклад, y – мітка	Дані: (X, y) та X , $ (X, y) < X $ X – приклад, y – мітка	Дані: X X – приклад, немає мітки!	Дані: пари стан-дія
Мета – знайти функцію відображення $X \rightarrow y$	Мета – знайти функцію відображення або категорію $X \rightarrow y$	Мета – знайти правильну категорію.	Мета – максимізація загальної винагороди, отриманої агентом при взаємодії з навколошнім середовищем.
Приклад  Це є яблуко.	Приклад  Це є яблуко.	Приклад  Цей об'єкт схожий на інший.	Приклад  Їжте це, бо це зробить вас сильнішим.

Рисунок 1.3 – Види навчання за характером досвіду (даних)

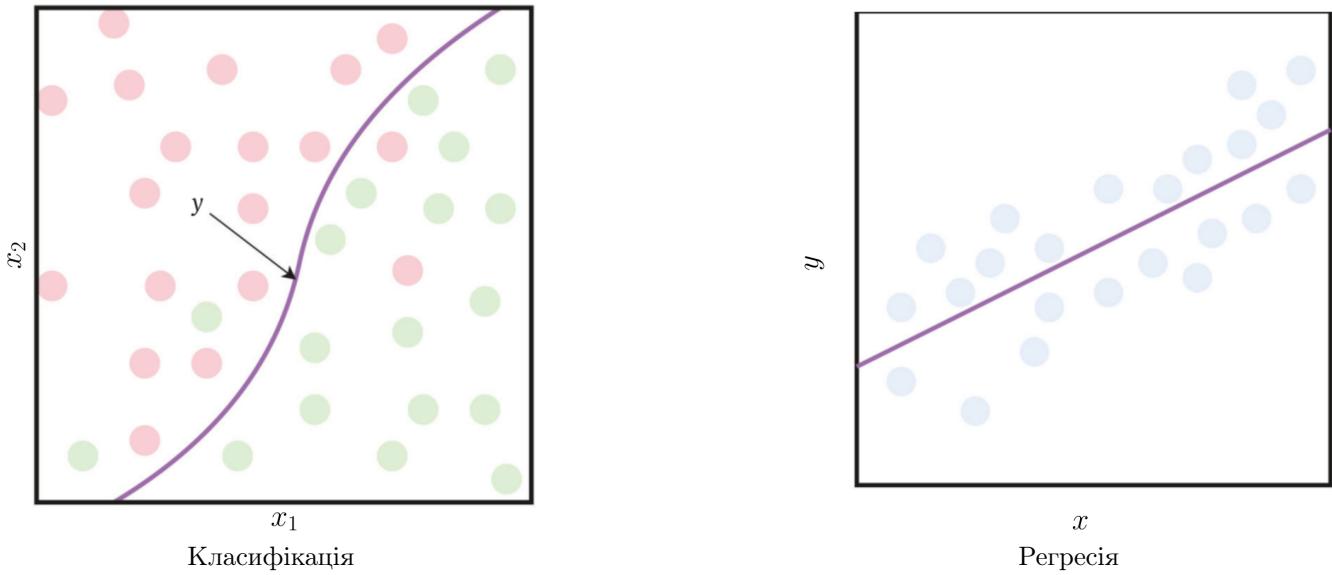


Рисунок 1.4 – Відмінність між класифікацією та регресією

завдання полягає у виявленні спаму, тоді усі повідомленя потрібно розділити на два класи: спам, не спам.

Завдання передбачення класу називається класифікацією, а завдання передбачення дійсного числа – регресією. Числове значення мітки, яке має бути передбачене моделлю, навченої з учителем, називається цільовим показником або метою. Прикладом регресії є завдання передбачення заробітної плати співробітника на основі його досвіду роботи та знань. Прикладом класифікації є завдання, коли модель повертає висновок (здоровий, хворий) на основі введених лікарем показників, які характеризують стан пацієнта.

Відмінність між класифікацією та регресією показано на рис. 1.4. У разі класифікації алгоритм навчання шукає лінію або, у загальному випадку, гіперповерхню, яка розділяє приклади різних класів. З іншого боку, у разі регресії алгоритм навчання прагне знайти лінію чи гіперповерхню, яка добре відповідає навчальним прикладам.

Мета алгоритму навчання з учителем – знайти екстремум цільової функції³ на навчальному наборі даних і скоротити розрив між прогнозами моделі та мітками реальних спостережень. Модель на вході приймає вектор ознак, а на виході видає прогноз, який дозволяє визначити мітку для цього вектора ознак. Наприклад, модель,

³Функція загальних втрат або емпіричний ризик.

створена з використанням набору даних про пацієнтів, може на вхід приймати вектор ознак, що характеризує стан пацієнта, а на виході видавати прогноз у вигляді ймовірності про наявності у пацієнта ознак деякого патологічного стану.

Те, що знаходитьться всередині методів машинного навчання, зокрема, глибоких нейронних мереж, може бути складним, але за своєю суттю це просто функції. Вони беруть деякі вхідні дані (вектор ознак \mathbf{X}) і генерують деякі дані виходу $f(\mathbf{X})$ – прогноз. Графічно це можна відобразити так як показано на рис. 1.5. Іншими словами, ми можемо говорити, що модель “дивиться” на вектор вхідних ознак і на основі досвіду роботи з аналогічними прикладами, виводить прогноз.

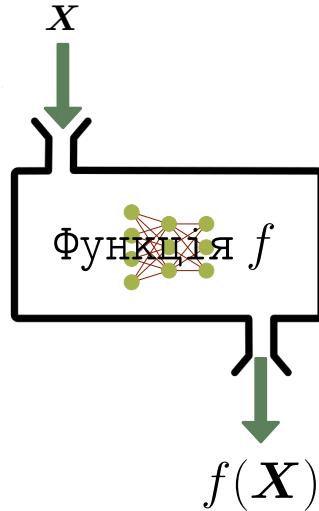


Рисунок 1.5 — Представлення моделі як функції

Статистичне контролюване навчання. Більшість алгоритмів засновані на оцінці розподілу ймовірностей $p(y|x)$. Найбільш поширилою стратегією для досягнення цього є спочатку параметризація проблеми (задання), в результаті чого вона зводиться до оцінки $p(y|x, w)$, а потім відбувається пошук оптимального параметра w шляхом максимізації функції правдоподібності (або функції логарифмічної правдоподібності для зручності обчислень) з використанням градієнтного спуску. Це стосується, наприклад, лінійної регресії та логістичної регресії. Обидва ці методи більш детально будуть розглянуті в наступних розділах.

Машини опорних векторів. Оскільки це один з найпоширеніших методів навчання з учителем⁴, ми детально його розглянемо. Цей метод полягає у пошуку найкращих розділювальних гіперплощин для наших даних. На відміну від вище згаданих методів, машина опорних векторів не надає ймовірностей, щоб оцінити ідентичність класу. Натомість, ідентичність класу визначається лише за положенням, де знаходиться приклад даних відносно розділяючої гіперплощини. Математично це відповідає запису $w^\top x + b$, значення цього рівняння може бути додатним або від'ємним, де w – вектор ваг та b – зсув. Дійсно, w , будучи ортогональним розділювальним гіперплосчині, тоді дані, що знаходяться на позитивній стороні гіперплосчини – це точки, скалярний добуток яких з w буде додатнім, а дані, які знаходяться на протилежній стороні гіперплосчини будуть мати від'ємний скалярний добуток з вектором w . Ідея машин опорних векторів походить від трюку ядра. Щоб пояснити трюк, потрібно розглянути теорему про представника (Representer theorem):

$$\exists (\alpha_i)_{i=1..n}, \quad w = \sum_{i=1}^n \alpha_i x_i, \quad (1.2)$$

де $x_i = i$ -ий приклад.

Ми можемо інтуїтивно зрозуміти цю теорему, коли виконуємо градієнтний спуск із завісною втратою (hinge

⁴Контрольоване навчання – синонім для навчання з учителем, часто використовується на практиці.

loss) в якості цільової функції, оновлення параметрів будуть відповідати або 0, або $-y_i x_i$ у процесі збіжності. Отже, врешті-решт, отримується w як лінійна комбінація $(x_i)_{i=1..n}$. Виходячи з цього ми отримуємо:

$$\exists(\alpha_i)_{i=1..n}, w^\top x + b = b + \sum_{i=1}^n \alpha_i x^\top x_i \quad (1.3)$$

Тепер використання функцій ядра для отримання додаткових функцій може забезпечити певні переваги, оскільки дає додаткову інформацію. Припустимо, ми вибрали для цього функцію відображення ознак ϕ , тоді нашими «новими» точками даних є $\phi(x)$. Якщо розмірність $\phi(x_i)$ величезна, внутрішні результати в сумі попередніх обрахунків буде дуже складно обчислювати. Трюк ядра дозволяє зробити ці обчислення набагато менш витратними: це відповідає тому факту, що для кожного вибору ϕ завжди існує білінійна форма k така, що $k(x, x_i) = \phi(x)^\top \phi(x_i)$ і це відповідає функції, обчисленій у набагато менш розмірному просторі!

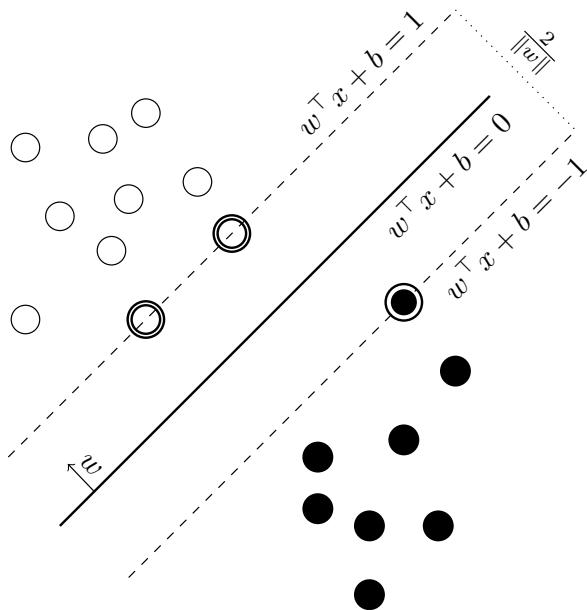


Рисунок 1.6 — Машина опорних векторів

k-найближчі сусіди. Цей метод може бути використаний для класифікації та регресії. Є непараметричним і працює в будь-яких ситуаціях, коли ми можемо визначити середнє значення за мітками. Модель визначає ідентичність класу кожної точки даних від її k-найближчих сусідів.

Дерево рішень. Цей метод полягає у розбитті вхідних даних на різні ділянки та визначення мети кожної точки даних, розглядаючи, в яку область вона потрапляє. На кожному кроці побудови дерева, модель спочатку знаходить, за яким об'єктом слід зробити розбиття, а потім за цією вибраною ознакою визначає найкраще розбиття, використовуючи метрику приросту інформації.

1.2.2 Навчання без учителя

У цьому випадку ми працюємо з множиною нерозмічених прикладів⁵, яку представлено наступним чином:

$$d = \{\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, \dots, \mathbf{X}^{(n)}\}, \quad (1.4)$$

де n — загальна кількість прикладів у наборі.

⁵Відсутні мітки.

Як і раніше, $\mathbf{X}^{(i)}$ – вектор вхідних ознак. Мета алгоритму навчання без учителя – створити модель, яка приймає на вхід вектор ознак \mathbf{X} і перетворює його на інший вектор або значення, яке можна використати для вирішення прикладного завдання. Наприклад, у задачах кластеризації, модель повертає ідентифікатор кластера для кожного вхідного вектора ознак з набору даних. Кластеризація використовується для пошуку груп схожих екземплярів у великому наборі даних, наприклад серед зображень або текстових документів.

У задачах зменшення розмірності модель повертає новий вектор ознак, який має меншу кількість елементів порівняно з вхідним вектором \mathbf{X} . Наприклад, дослідник має вектор ознак, який надто складний для його візуалізації та розуміння, оскільки число вимірів може бути дуже великим. Модель зменшення розмірності перетворює цей вектор великої розмірності на інший вектор меншої розмірності, який буде зберігати частину найбільш важливої інформації вхідного вектора і буде простішим для візуалізації та розуміння.

У задачах виявлення аномалій виходом є дійсне число, яке вказує, наскільки \mathbf{X} відрізняється від типового приклада з набору даних, який використовувався для навчання моделі. Виявлення аномалій може використовуватися для розпізнавання підозрілих або несанкціонованих дій у системах безпеки, таких як виявлення атак на комп’ютерні мережі або виявлення зловживань кредитними картками. Крім того, виявлення аномалій може бути корисним для визначення якості даних. Аналіз викидів може допомогти виявити та виправити помилки у даних, які можуть виникнути через неправильну розмітку, помилки вимірювань або інші фактори.

1.2.3 Навчання з частковим залученням учителя

У цьому випадку набір даних представлено множиною з розмічених та нерозміченых прикладів:

$$d = \{(\mathbf{X}^{(1)}, y^{(1)}), \mathbf{X}^{(2)}, \dots, \mathbf{X}^{(n)}\}, \quad (1.5)$$

де n – загальна кількість прикладів у наборі, $y^{(i)}$ – мітка i -го прикладу.

Як і у попередніх випадках, $\mathbf{X}^{(i)}$ – вектор вхідних ознак. Зазвичай кількість нерозміченых прикладів набагато більша ніж кількість розмічених. Навчання з частковим залученням учителя поєднує елементи навчання з учителем та навчання без учителя. Основна мета цього підходу полягає в тому, щоб використати невелику кількість прикладів з мітками для навчання моделі, а велику кількість нерозміченых прикладів – для отримання додаткової інформації та поліпшення універсальності моделі. У випадку, коли кількість прикладів для деяких класів обмежена, а для інших – значно більша, навчання з частковим залученням учителя може допомогти боротися з дисбалансом класів та покращити прогнози для екземплярів класів, які зустрічаються у наборі даних значно рідше за інші.

1.2.4 Навчання з підкріпленням

Навчання з підкріпленням – це розділ машинного навчання, де агент⁶ вивчає оптимальну стратегію для своїх дій з метою максимізації загальної винагороди, отриманої внаслідок переходів між дозволеними станами середовища⁷. Цикл взаємодії агента з середовищем показано на рисунку 1.7. Стан середовища, який доступний агенту для спостережень – вектор вхідних ознак. Оптимальна стратегія – це функція, яка на вход приймає вектор ознак стану, а на виході видає оптимальну дію, яку слід виконати агенту у цьому стані. Дія є оптимальною, якщо вона максимізує середню загальну винагороду агента.

Базується навчання з підкріпленням на гіпотезі винагороди. Гіпотеза винагороди звучить наступним чином: будь-яка мета може бути формалізована як результат максимізації сукупної винагороди. Головна ідея гіпотези винагороди полягає у тому, що агент спрямований на вивчення оптимальної стратегії, тобто такої послідовності дій, яка дозволить агенту отримати у кінцевому підсумку максимальну загальну винагороду.

Навчання з підкріпленням фундаментально відрізняється від контролюваного (з учителем) та неконтрольованого (без учителя) / напівконтрольованого навчання, оскільки ми тренуємо модель управляти діями нашого

⁶Агент – сутність (програма чи об’єкт), що існує та діє окремо в деякому середовищі.

⁷Стochasticний та невизначений світ у якому існує та діє агент.

агента з метою знаходження ним оптимальної послідовності дій, які приведуть його до вирішення поставленого завдання з максимальним кінцевим значенням загальної винагороди. Іншими словами, метою навчання агента є визначення найкращого наступного стану у який має перейти агент для отримання найбільшої кінцевої винагороди.

Задачі, де необхідно приймати рішення або запровадити певну поведінку (виконати певні дії), прийнято називати задачами керування. Навчання з підкріпленим займається вирішенням задач в яких прийняття рішень часто є послідовним, а мета – довгостроковою. Такі завдання виникають в іграх, робототехніці, управлінні ресурсами чи логістикою.

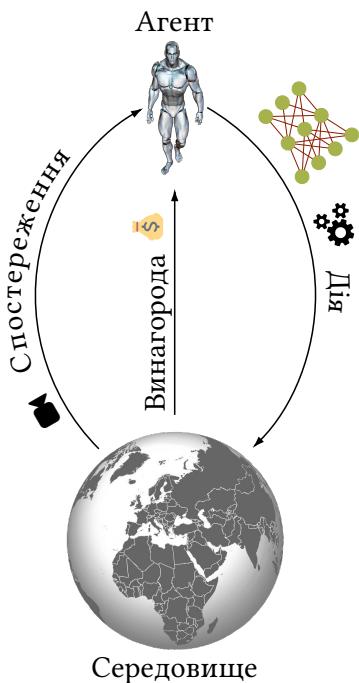


Рисунок 1.7 — Цикл взаємодії агента з середовищем

1.3 Термінологія машинного навчання

У цьому параграфі Ви познайомитеся із загальноприйнятою термінологією, яка відноситься до самих даних (наприклад, дані, що використовуються прямо і опосередковано, первинні та акуратні дані, навчальні та зарезервовані дані) та до машинного навчання (наприклад, орієнтири, гіперпараметр, конвеєр та ін.).

1.3.1 Дані, що використовуються прямо та опосередковано

Дані у проекті машинного навчання можуть використовуватися для формування прикладів прямо чи опосередковано.

Припустимо, що ми будуємо систему для розпізнавання частин мови. На вход моделі надходить послідовність слів, а на виході видається послідовність міток тієї ж довжини, що й вхід. Приклад розпізнавання частин мови у реченні показано на рисунку 1.8. Для того, щоб алгоритм машинного навчання міг прочитати дані, ми повинні перетворити кожне слово природної мови на зрозумілий для машини масив атрибуутів⁸, який ми називаємо вектором ознак. Для створення таких атрибуутів ми можемо вдатися до використання словників, довідкових таблиць тощо. Ви, ймовірно, уже зрозуміли, що послідовність слів – це дані, що використовуються

⁸Терміни “атрибут” та “ознака” використовуються як синоніми для опису конкретної властивості прикладу.

для формування навчальних прикладів безпосередньо, тоді як дані, що містяться у словниках та довідкових таблицях, використовуються опосередковано для формування вектора ознак.

Успіх — це вміння рухатись від невдачі до невдачі, не втрачаючи ентузіазму. Вінston Черчилль

Compute

Computation time on cpu: cached

Успіх NOUN — PUNCT Це PRON вміння NOUN рухатись VERB від ADP невдачі NOUN , PUNCT не PART втрачаючи VERB ентузіазму NOUN . PUNCT Вінston PROPN Черчилль PROPN

Рисунок 1.8 — Приклад розпізнавання частин мови у реченні [3]

1.3.2 Первинні та акуратні дані

Як щойно ми обговорили, безпосередньо використовувані дані – це формат даних у якому здійснюється збір екземплярів для формування набору, проте, не завжди такий формат можна використати прямо для навчання моделей. Тому кожен екземпляр з такого набору даних часто доводиться перетворювати у вектор ознак, який буде зрозумілим для машини та використаний для навчання. **Первинні дані** – це набір ознак у справжній формі збору, часто такі дані є неструктурованими⁹. Наприклад, PDF-документ або JPEG-файл є первинними неструктурованими даними; алгоритм машинного навчання неспроможний використати їх напряму.

Необхідно (але не достатньо) умовою використання даних у машинному навчанні є їх акуратність¹⁰. Прикладом структурованих даних може бути таблиця, рядки якої представляють окремі екземпляри (приклади) спостережень, а стовпці – атрибути прикладів, як показано на рисунку 1.9. Інколи первинні дані можуть бути структурованими, наприклад, якщо вони вже представлені у вигляді таблиці. Однак на практиці, щоб отримати структуровані дані з первинних, аналітики нерідко вдаються до процедури конструювання ознак, яка застосовується безпосередньо до первинних та інколи опосередкованих даних з метою перетворення кожного первинного прикладу у структурований вектор ознак.

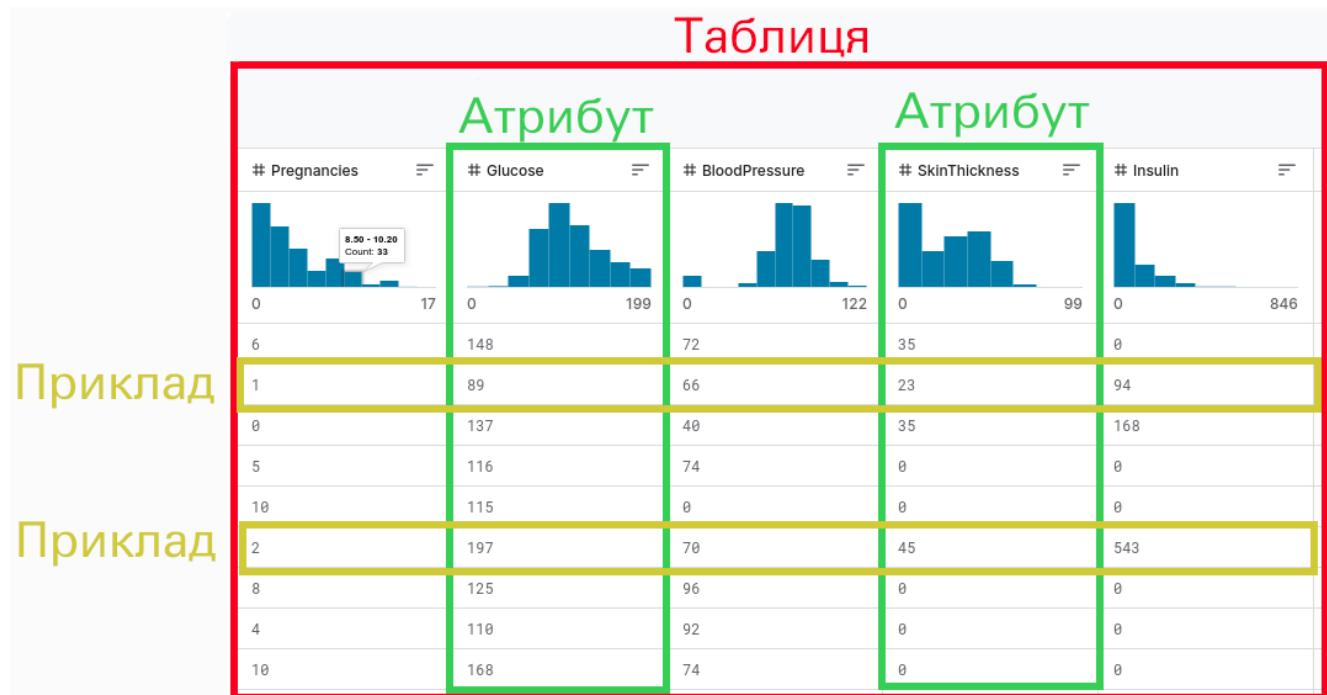


Рисунок 1.9 — Структуровані дані подані у вигляді таблиці

⁹Неструктуровані дані – це дані, які не мають чіткої організації або формату. Прикладами неструктурованих даних є текстові документи, зображення, аудіо- та відеофайли, електронні листи, дані з соціальних мереж, веб-сторінки тощо.

¹⁰Термін “акуратні дані” використовується для позначення даних, які є точними, достовірними та структурованими, тобто організовані в спосіб, який сприяє ефективному використанню та аналізу.

Важливо зазначити, що в деяких задачах приклад, який використовується алгоритмом навчання, може бути представлений атрибутами, які утворюють послідовність векторів, матрицю або послідовність матриць. Поняття структурованості для таких даних визначається наступним чином: необхідно лише замінити рядок фіксованої ширини в таблиці матрицею фіксованої ширини та висоти або узагальненням матриці на більшу кількість вимірів – тензором.

Термін “акуратні дані” було уведено Хедлі Уїкхемом (Hadley Wickham) в однойменній статті [4].

Варто зазначити, що дані можуть бути акуратними, але все одно непридатними для використання конкретним алгоритмом машинного навчання. Слід звернути увагу, що в академічній літературі з машинного навчання слово “приклад” зазвичай стосується акуратних даних. Однак на етапі збору даних та їх розмітки приклади можуть бути ще представлені в первинній формі: зображення, тексти або рядки з категоріальними атрибутами в електронній таблиці. Коли важливо підкреслити різницю, ми будемо використовувати поняття “первинний приклад”, щоб вказати, що частина даних ще не була перетворена на структурований вектор ознак. В іншому випадку ми припускатимемо, що приклади мають структуровану форму.

Проміжний тип між структурованими та неструктуркованими даними займають так звані напівструктурковані дані. Вони мають деяку форму або організацію, але не повністю відповідають стандартам структурованих даних. До напівструктуркованих даних можна віднести файли журналів, текстові файли із комами або табуляторами, а також документи у форматах JSON та XML.

1.3.3 Навчальні та зарезервовані дані

На практиці інженери машинного навчання зазвичай працюють із трьома різними наборами прикладів:

1. навчальний;
2. контрольний¹¹ (валідаційний або розробки)
3. тестовий

Отримавши дані у вигляді набору прикладів, перше, що потрібно зробити – перемінати приклади та розділити дані на три окремі набори: навчальний, контрольний та тестовий. Навчальний набір зазвичай є найбільшим; він використовується в алгоритмі навчання для створення моделі. Контрольний та тестовий набори мають приблизно одинаковий, набагато менший розмір. Алгоритму навчання забороняється використовувати приклади з контрольного чи тестового набору для навчання моделі. Тому обидва ці набори іноді називають зарезервованими.

Причина наявності трьох наборів, а не одного, проста: у процесі навчання ми не хочемо, щоб модель добре передбачала мітки лише тих прикладів, які алгоритм вже зустрічав. Тривіальний алгоритм, який просто запам'ятає усі навчальні приклади, не робитиме жодних помилок, коли його попросять передбачити мітки з навчальної вибірки. Однак на практиці такий алгоритм буде некорисним. Нам потрібна модель, яка буде добре передбачати мітки прикладів, які не брали участі у процесі навчання. Іншими словами ми хочемо мати гарну якість передбачень на зарезервованому наборі. Тобто, ми хочемо, щоб модель добре працювала на більшості випадкових вибірок із статистичного розподілу, до якого належать наші дані. Ми виходимо з того, що якщо модель добре працює на зарезервованому наборі, який випадково вибрано з невідомого розподілу до якого належать наші дані, то існують високі шанси, що така модель буде добре працювати і на інших випадково вибраних прикладах. У такому випадку говорять, що модель має високу здатність узагальнення.

Два зарезервованих набори, а не один, потрібні, оскільки контрольний набір використовується, щоб вибрати алгоритм навчання та відшукати оптимальні конфігураційні параметри цього алгоритму, які прийнято називати гіперпараметрами. Тестовий набір використовується для оцінювання моделі перед її передачею замовнику або розгортанням у виробничому середовищі. Ось чому так важливо, щоб ніяка інформація з

¹¹Іноді, якщо навчальних прикладів замало, аналітик може обійтися без контрольного набору, скориставшись при цьому перехресною перевіркою.

контрольного та тестового наборів не була показана алгоритму на етапі навчання. Інакше результати контролю та тестування, швидше за все, будуть надто оптимістичними. Така проблема іноді трапляється через просочування даних – явище, яке буде детальніше розглянуто пізніше.

1.3.4 Орієнтир

У машинному навчанні орієнтиром (baseline) називається простий алгоритм розв'язання задачі, зазвичай заснований на евристіці, простій зведеній статистиці або дуже простому алгоритмі машинного навчання. Наприклад, якщо задача пов'язана з класифікацією, ми можемо обрати класифікатор, який буде слугувати орієнтиром, та виміряти його якість. Потім, ми можемо порівнювати з цим орієнтиром будь-яку наступну модель, яка зазвичай буде отримана із застосуванням більш витончених підходів.

1.3.5 Конвеєр даних

Конвеєр даних – це послідовність операцій із набором даних, яка веде від початкового стану до готового формату для використання у моделі. Сюди відносяться наступні три операції більшості конвеєрів даних:

1. **Екстракція** передбачає процес вилучення даних із кількох однорідних або різномірних джерел.
2. **Трансформація** стосується очищення та маніпулювання даними з метою перетворення їх у належний формат: розбиття даних, заміна пропущених даних, виділення ознак, збільшення даних, зменшення незбалансованості класів, зменшення розмірності.
3. **Завантаження** – це введення перетворених даних у пам'ять процесорів¹², які здійснюють навчання.

На практиці, розгортаючи модель у виробничому середовищі, ми зазвичай розгортаємо увесь конвеєр. Крім того, при налаштуванні гіперпараметрів зазвичай оптимізується також увесь конвеєр повністю.

1.3.6 Гіперпараметри та параметри

Гіперпараметри – це вхідні дані алгоритму машинного навчання чи конвеєра, які впливають на якість моделі. Вони не належать до навчальних даних, тому не можуть бути використані для навчання моделі. Наприклад, максимальна глибина дерева в алгоритмі дерева рішень, штраф за неправильну класифікацію у методі опорних векторів, величина k в алгоритмі k найближчих сусідів, цільова розмірність в алгоритмі зменшення розмірності, розмір фільтрів у згорткових мережах, кількість нейронів у кожному шарі нейронної мережі, швидкість навчання, епохи навчання – це приклади гіперпараметрів.

Параметри, з іншого боку, є змінними, які уточнюються у ході навчання та визначають модель. На кожній ітерації навчання параметри оновлюються оптимізаційним алгоритмом за визначенім правилом. Оновлення параметрів здійснюється з метою знаходження такої конфігурації параметрів моделі, які будуть відповідати глобальному мінімуму цільової функції втрат на заданому наборі даних. Прикладами параметрів є вагові коефіцієнти w та зсув b у рівнянні лінійної регресії: $\hat{y} = wx + b$, де x – вектор вхідних ознак моделі, а \hat{y} – вихід моделі (передбаченням).

1.3.7 Навчання на основі моделі та екземплярів

Більшість алгоритмів навчання з учителем ґрунтуються на моделях. Типовою моделлю є метод опорних векторів (Support Vector Machine – SVM). В алгоритмах навчання на основі моделей навчальні дані використовуються для створення моделі з оптимальною конфігурацією параметрів. Навчену модель можна зберегти на диску, а навчальні дані видалити.

В алгоритмах навчання на основі екземплярів весь набір даних використовується як частина моделі. Одним із найчастіше використовуваних на практиці алгоритмів такого роду є метод k найближчих сусідів (kNN).

¹²CPU, GPU, TPU тощо.

Для передбачення мітки вхідного прикладу, алгоритм kNN розглядає його окіл у просторі векторів ознак і виводить мітку, що зустрічається у цьому околі найчастіше.

1.3.8 Поверхневе та глибоке навчання

Алгоритм поверхневого навчання вивчає параметри безпосередньо з вхідних ознак навчальних прикладів. Більшість алгоритмів машинного навчання є поверхневими, наприклад, машина опорних векторів (SVM), лінійна та логістична регресія, перцептрон, дерева рішень тощо. Коли мова йде про глибоке навчання – мається на увазі пошук шаблону в даних за допомогою нейронної мережі, що має кілька прихованих шарів, які розташовані між входом та виходом мережі. У глибокому нейромережевому навчанні або просто глибокому навчанні, на відміну від поверхневого, більшість параметрів моделі вивчаються не з вхідних ознак навчальних прикладів, а з активацій, тобто виходів попередніх шарів.

1.3.9 Навчання та оцінка

Коли алгоритм машинного навчання застосовується до набору даних для одержання моделі, говорять про навчання моделі або просто навчання.

Коли навчена модель застосовується до деякого одного вхідного прикладу (іноді до деякої послідовності прикладів) з метою отримання передбачення (або кількох передбачень), говорять про оцінювання моделі або просто оцінювання.

1.4 Становлення глибинного навчання

Глибинне навчання – це потужний інструмент штучного інтелекту, який використовує багатошарові штучні нейронні мережі для досягнення передової (SOTA) точності в таких завданнях як виявлення (ідентифікація) об'єктів, класифікація та сегментація об'єктів, розпізнавання мовлення, машинний переклад. Використовуючи глибинне навчання, комп'ютери можуть вивчати та розпізнавати приховані закономірності у даних, які вважаються западто складними або неочевидними, щоб їх можна було класично запрограмувати.

Батьком глибинного навчання вважається український вчений Олексій Григорович Івахненко [5, 6], який опублікував перший загальний робочий алгоритм контролюваного навчання для глибинних багатошарових перцепtronів прямого поширення [7, Івахненко та Лапа, 1965]. Нейрони багатошарового перцептрана мали поліноміальні функції активації, що поєднували додавання і множення в поліномах Колмогорова-Габора. Пізніше у 1971 Івахненко уже описав глибоку мережу з 8 шарами [8], навчену за допомогою розробленого ним методу групового урахування аргументів, який все ще популярний дотепер [5].

Ранні нейронні мережі [9, Мак-Каллок (McCulloch) & Піттс (Pitts), 1943] взагалі не вчилися. Згодом у 1949 році Хебб (Hebb) у своїй роботі [10] опублікував ідеї про навчання без учителя (неконтрольоване навчання). У наступних десятиліттях з'явилися неглибокі неконтрольовані та контролювані нейронні мережі (наприклад, [11, Розенблatt (Rosenblatt), 1958]). Ранні контролювані нейронні мережі були по суті варіантами лінійних регресорів, що датуються двома попередніми століттями (Гаусс, Лежандр) [5].

Не зважаючи на те, що більшість методів глибинного навчання є нещодавніми винаходами, основна ідея програмування з даними та нейронними мережами (назви багатьох моделей глибинного навчання) вивчалась століттями [12, ст. 34-36]. Основні причини, чому глибинне навчання зазнало успіху та показує хороші результати у багатьох задачах пов'язано з появою великих наборів даних у відкритому доступі, появою компаній, що обслуговують сотні мільйонів користувачів онлайн, поширенням дешевих високоякісних датчиків, дешевим зберіганням даних (закон Крайдера) та дешевим обчисленням (закон Мура), зокрема на графічних процесорах. Усе це дало можливість алгоритмам та моделям, які були раніше обчислювально нездійсненними, стати актуальними. Найкраще це продемонстровано у таблиці 1.1 [12, ст. 36-38].

Табл. 1.1: Зростання розміру наборів даних, оперативної пам'яті комп'ютерів та обчислювальної потужності [12, ст. 36]

Десятиліття	Датасет	Пам'ять	FLOPS
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (Nvidia C2050)
2020	1 T (social network)	100 GB	1 PF (Nvidia DGX-2)

Очевидно, що оперативна пам'ять не встигає за зростанням даних. У той же час зростання обчислювальної потужності випереджає рівень доступних даних. Це означає, що статистичні моделі повинні стати більш ефективними з точки зору використання пам'яті (це зазвичай досягається шляхом додавання нелінійності), водночас мати можливість витрачати більше часу на оптимізацію цих параметрів через збільшений бюджет обчислень. Отже, основний акцент у машинному навчанні та статистиці перемістився від (узагальненіх) лінійних моделей і методів ядра (функції подібності) до глибинних нейронних мереж. Це також є однією з причин того, чому багато фундаментальних розробок глибинного навчання, такі як багатошарові персептронони [9, Мак-Каллок (McCulloch) & Піттс (Pitts), 1943], згорткові нейронні мережі [13, Лекун (LeCun), 1998], довга короткочасна пам'ять [14, Хохрайтер (Hochreiter) & Шмідхубер (Schmidhuber), 1997] та Q-навчання [15, Воткінс (Watkins) & Даян (Dayan), 1992] були по суті заново “відкриті” за останнє десятиліття, після того як вони пролежали практично бездіяльно протягом значного часу [12, ст. 36-38].

Нижче наведено перелік ідей, які допомогли дослідникам досягти значного прогресу в глибинному навчанні за останнє десятиліття:

1. Нові методи контролю пропускної здатності, такі як дропаут [16, Шривастава (Srivastava) та ін., 2014], допомогли зменшити ризик перенавчання. Це було досягнуто шляхом введенням шуму [17, Бішоп (Bishop), 1995] по всій нейронній мережі, змінюючи ваги випадковим чином для навчальних цілей.
2. Механізм уваги вирішив другу проблему, яка не давала спокою статистиці понад століття: збільшення пам'яті та складності системи не збільшуючи при цьому кількості навчальних параметрів. Дослідники знайшли елегантне рішення, використовуючи те, що можна розглядати лише як структуру вказівника, яку можна вивчати [18, Bahdanau et al., 2014]. Замість того, щоб запам'ятувати усю текстову послідовність, наприклад, для машинного перекладу у представленні вектора фіксованої довжини, усе, що потрібно було зберегти – це вказівник на проміжний стан процесу перекладу. Це дозволило значно підвищити точність для довгих послідовностей, оскільки моделі більше не потрібно було запам'ятувати всю послідовність перед початком створення нової послідовності.
3. Генеративні змагальні мережі [19, Гудфеллоу (Goodfellow) та ін., 2014]. Традиційно статистичні методи оцінки щільності та генеративні моделі зосереджені на пошуку правильних розподілів ймовірностей і (часто наблизених) алгоритмів вибірки з них. У результаті ці алгоритми були значною мірою обмежені відсутністю гнучкості, яка властива статистичним моделям. Основним нововведенням у генеративних змагальніх мережах була заміна семплера довільним алгоритмом з диференційованими параметрами. Потім ці параметри коригуються таким чином, щоб дискримінатор (фактично тест з двома вибірками) не міг відрізнити підроблені дані від реальних.
4. Вдосконалено алгоритми для паралельного та розподіленого навчання моделей. Крім того, здатність розпаралелювати обчислення також сприяла значним успіхам у навчанні з підкріпленням, принаймні для тієї частини, яка стосується моделювання середовища існування агента. Це призвело до значного прогресу в симуляції фізики (наприклад, за допомогою MuJoCo) та дало можливість комп'ютерам досягти надлюдської продуктивності в ряді завдань: відео ігри [20], покер [21], а також у настільних іграх, включаючи го та шахи [22, 23, 24, 25].

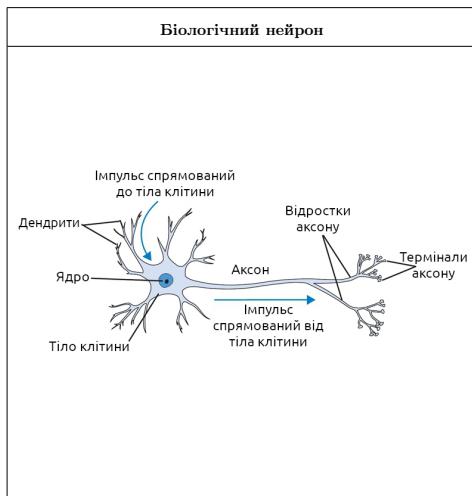
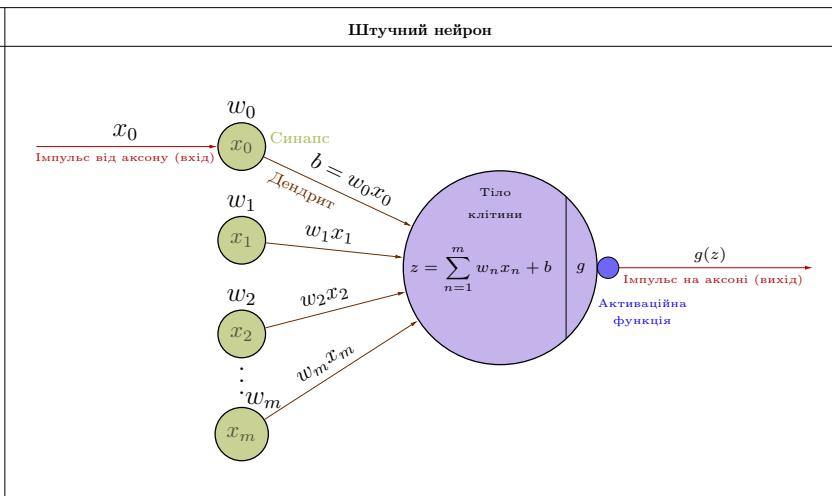
5. Фреймворки глибинного навчання відіграли вирішальну роль у поширенні ідей. Перше покоління фреймворків, що дозволяли легко моделювати, включало: Caffe, Torch та Theano. За допомогою цих інструментів було написано багато основоположних робіт. На даний момент їх замінили TensorFlow (часто використовується через високорівневий API Keras), CNTK, Caffe 2 та Apache MXNet. Третє покоління інструментів, а саме імперативних інструментів для глибинного навчання, ймовірно, очолив Chainer, який використовує синтаксис, подібний до Python NumPy для опису моделей. Цю ідею взяли на озброєння PyTorch, API Gluon MXNet та Jax [12, ст. 36-38].

1.5 Біологічний та штучний нейрон

Створення штучних нейронних мереж спочатку було мотивовано потребою моделювання біологічних нервових систем, а сьогодні це стало проблемою інженерії для досягнення продуктивності на рівні людини і вище (надлюдська продуктивність) у ряді задач машинного навчання.

Базовою обчислювальною одиницею мозку є нейрон. Мозок дорослої людини містить приблизно 86 мільярдів нейронів [26], які з'єднані приблизно $10^{14} - 10^{15}$ синапсами. Біологічний нейрон та його спрощена математична модель (штучний нейрон) графічно представлені у таблиці 1.2. Кожен нейрон, отримавши вхідне значення сигналу, спрямовує його по дендритах до тіла клітини, після чого нейрон активується та продукує вихідний сигнал через аксон. Аксон у свою чергу має відгалуження, які через синапси з'єднані з дендритами інших сусідніх нейронів. У математичній моделі сигнал, що надходить від аксона (*наприклад, x_0*) взаємодіє мультиплікативно (w_0x_0) з дендритами сусідніх нейронів. Результат такої взаємодії залежить від синаптичної сили у цьому синапсі (для розглянутого випадку – це w_0 , див. таблицю 1.2). Ідея полягає у тому, що синаптичні сили (вагові коефіцієнти або просто ваги w) є навчальними параметрами, за допомогою яких можна контролювати силу впливу одного нейрона на інший та його напрямок (збудливий – додатні ваги та гальмівний – від'ємні ваги). У біологічній моделі дендрити передають вхідні сигнали до тіла клітини, де вони усі додаються, збільшуючи напругу на мембрани нейрона. Вхідний сигнал часто описується як іонний струм через мембрани нейрона, який виникає, коли нейротрансмітери призводять до активації іонні канали клітини. Якщо сумарна величина сигналу перевищує деякий поріг, нейрон спрацьовує (*активується*), посилаючи коротко-часний сплеск напруги (*потенціал дії*) уздовж свого аксона, після чого вольтаж на мембрани скидається до потенціалу спокою. У математичній моделі штучного нейрона припускається, що точна часова залежність вхідних сигналів не має значення і лише частота активації нейрона передає інформацію.

Табл. 1.2: Біологічний та штучний нейрон

Біологічний нейрон	Штучний нейрон
 <p>Імпульс спрямований до тіла клітини Дендрити Ядро Аксон Відростки аксону Термінали аксону Імпульс спрямований від тіла клітини</p>	 <p>x_0 x_1 x_2 \vdots x_m</p> <p>w_0 w_1 w_2 \vdots w_m</p> <p>Синапс Дендрит $b = w_0 \cdot x_0$</p> <p>$z = \sum_{n=1}^m w_n x_n + b$</p> <p>$g(z)$ $g(z) = \text{Імпульс на аксоні (вихід)}$</p> <p>Активаційна функція</p>

Однією з найперших моделей, яка описує роботу біологічного нейрона є ідеальна модель інтегрування і акти-

вації (perfect integrate-and-fire), запропонована у 1907 році Луї Лапіком (Louis Lapicque):

$$I(t) = \sum_{k=1}^m i_k(t) = C \frac{dV}{dt}, \quad (1.6)$$

де $I(t)$ – сумарна вхідна сила струму від m синапсів як функція часу; C – ємність мембрани нейрона; V – напруга на мембрані нейрона, також є функцією від часу.

Частота активації нейрона як функція від постійної сили струму має наступний вираз:

$$g(I) = \frac{I}{CV_{\text{пор.}} + It_{\text{від.}}}, \quad (1.7)$$

де $V_{\text{пор.}}$ – порогове значення напруги на мембрані нейрона; $t_{\text{від.}}$ – період відновлення, який обмежує частоту активації нейрона.

Враховуючи деякі спрощення в інтерпретації математичної моделі штучного нейрона, інтенсивність спрацьовування штучного нейрона розраховується за допомогою активаційної функції g , яка представляє частоту сплесків напруги вздовж аксону для біологічного нейрона. Приклади деяких активаційних функцій представлено у таблиці 1.3. Під спрощеннями в інтерпретації слід розуміти те, що існує велика кількість різних нейронів, кожен з різними властивостями, дендрити у біологічних нейронах виконують складні нелінійні обчислення, синапси – це не просто вагові коефіцієнти, а є нелінійною динамічною системою.

Приклад коду для прямого поширення сигналу через один штучний нейрон (далі просто нейрон) може виглядати наступним чином:

```

1 import numpy as np
2 m = 3
3
4 def input_signal(m):
5     """
6         Ця функція створює вектор випадкових чисел розмірністю (m, 1)
7         Аргументи:
8             m -- розмір вектора X або кількість вхідних сигналів
9         Повертає:
10            X -- вектор випадкових чисел розмірністю (m, 1)
11            """
12
13     X = np.random.randn(m, 1)
14     return X
15
16 X = input_signal(m)
17 print("X = " + str(X))

X = [[ 0.36910181]
 [ 0.49501913]
 [-0.3970563 ]]
```

```

1 def parameters_INITIALIZATION(m):
2     """
3         Ця функція створює вектор випадкових чисел розмірністю (m, 1)
4         для ш та ініціалізує в
5         Аргументи:
6             m -- розмір вектора w або кількість вагових коефіцієнтів
7         Повертає:
```

```

8     w -- вектор випадкових чисел розмірністю (m, 1)
9     b -- скаляр (відповідає за зміщення (bias))
10    """
11
12    w = np.random.randn(m, 1)
13    b = 2
14    return w, b
15
16 w, b = parameters_initialization(m)
17 print("w.T = " + str(w.T))
18 print("b = " + str(b))

```

```
w.T = [[-0.40707941  0.32829049 -0.86712231]]
b = 2
```

```

1 def forwardPropagate(X, w, b):
2     """
3     Ця функція знаходить сумарний вхідний сигнал та активаційне
4     значення для прямого поширення
5     Аргументи:
6     X -- вектор вхідних сигналів розмірністю (m, 1)
7     w -- вектор вагових коефіцієнтів (m, 1)
8     b -- скаляр, зміщення (bias)
9     Повертає:
10    z -- сумарний вхідний сигнал на нейроні
11    g -- активаційне вихідне значення сигмоїди
12    """
13
14    z = np.dot(w.T, X) + b
15    g = 1 / (1 + np.exp(-z))
16    return z, g
17
18 z, g = forwardPropagate(X, w, b)
19 print("Сумарний сигнал = " + str(z))
20 print("Активація = " + str(g))

```

```
Сумарний сигнал = [[2.3565527]]
Активація = [[0.91345366]]
```

1.5.1 Функції активації

Функції активації використовуються у кінці кожного прихованого шару та на шарі виходу нейронної мережі для введення нелінійності в модель. Кожна така функція приймає одне число і виконує над ним певну задану математичну операцію. Деякі загальнозвичайні функції активації представлено у таблиці 1.3.

Табл. 1.3: Деякі загальновживані функції активацій

Сигмоїда	Гіперболічний тангенс	ReLU	Leaky ReLU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ $\epsilon \ll 1$

Сигмоїда приймає дійсне число z і переводить його у діапазон значень від 0 до 1. Використовується сигмоїда для бінарної чи багатоміткової класифікації на останньому шарі мережі (вихід). Зокрема, великі від'ємні z наближають значення сигмоїди усе близьче до 0, а великі додатні – до 1. Сигмоїда має добре зрозумілу інтерпретацію – вихідне значення можна розглядати як інтенсивність активації нейрона від стану спокою (0) до насиченої інтенсивності активації з максимальною частотою (1). На практиці, як правило, сигмоїду використовують на вихідному шарі нейронної мережі, а отримане значення інтерпретують як ймовірність:

$$P(y=1|z) = \sigma(z) = \frac{1}{1+e^{-z}}, \quad (1.8)$$

де y – очікуване значення моделі.

Сигмоїду не рекомендовано використовувати у прихованих шарах нейронної мережі через спричинення двох основних проблем:

1. Зникнення градієнтів (*vanishing gradients*) для насичених значень активації (0 або 1).
2. Отримання лише додатних значень через відсутнію симетрію функції відносно нуля.

Зникнення градієнтів призводить до повільного навчання глибинних нейронних мереж або навіть до повної зупинки, оскільки навчальні параметри мережі розраховані за допомогою зворотного поширення, такі як ваги та зсуви, не будуть оновлюватись. Графічне представлення сигмоїди разом з її градієнтом показано на рисунку 1.10. Як можна побачити з цього рисунку для насичених значень активації нейрона (на хвості 0 або 1), значення градієнта наближається до нуля. Зникнення градієнтів може виникати, коли ваги сигмоїдних нейронів ініціалізовані погано – встановлено дуже великі позитивні або негативні значення. Це призводить до насичення активації сигмоїди та зникнення градієнта. Проте, навіть якщо ваги були ініціалізовані добре, а градієнт обраний близьким до максимуму (наприклад, 0.2) зі збільшенням кількості прихованих шарів нейронної мережі, проблема зникнення градієнтів буде виражатися усе сильніше. Так для мережі лише з 4-х шарів та значення градієнта на кожному шарі 0.2 отримаємо $0.2^4 = 0.0016$. Деякі приклади навчання нейронних мереж за допомогою зворотного поширення (*backpropagation*) розглянуто у параграфі 2.1. Інша проблема, яка викликана відсутністю симетрії сигмоїди відносно нуля, також негативно впливає на процес навчання нейронної мережі так як дані, які будуть надходити у нейрон завжди будуть додатні ($x > 0$ для $z = w^T x + b$) – це означає, що усі градієнти ваг під час зворотного поширення стануть додатними або від'ємними, що призведе до небажаного зигзагоподібного оновлення. Однак слід зауважити, що після того, як ці градієнти складаються у пакет даних, остаточне оновлення ваг може мати змінні знаки, дещо пом'якшуючи цю проблему. Не зважаючи на те, що проблема, яка викликана відсутністю симетрії сигмоїди відносно нуля хоч і негативно впливає на навчання нейронної мережі, проте має менш серйозні наслідки порівняно з проблемою зникнення градієнтів.

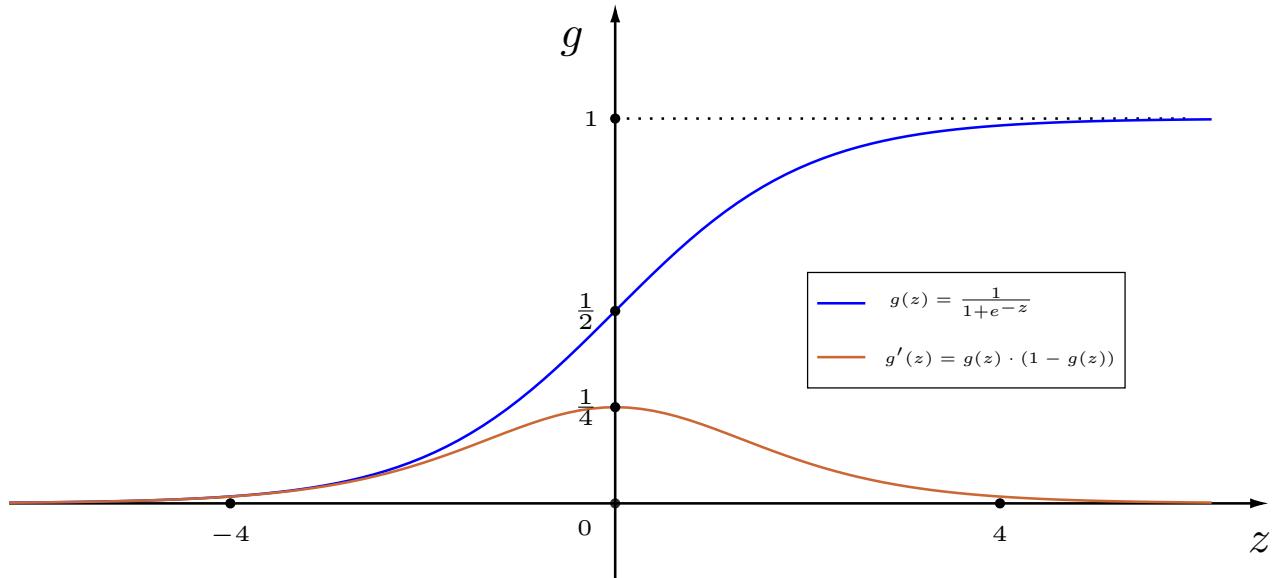


Рисунок 1.10 — Сигмоїда та її градієнт

Softmax – узагальнена форма сигмоїдної активаційної функції, використовується на останньому шарі нейронної мережі для задач класифікації даних за декількома категоріями (три і більше). Після отримання n необмежених значень ($z = [z_1, z_2, \dots, z_i, \dots, z_n]$), softmax нормалізує ці значення на n так, щоб у сумі отримати 1. Потім, нормалізовані значення можна розглядати як ймовірності. Кожне значення ймовірності відповідає за певний клас. Функція softmax, яка вказує на ймовірність i -го класу, має таке математичне визначення:

$$\text{softmax} = p(y = i|z) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}} \quad (1.9)$$

Гіперболічний тангенс приймає дійсне число z і переводить його у діапазон значень від -1 до 1. Графічне представлення гіперболічного тангенса разом з його градієнтом показано на рисунку 1.11. На відміну від сигмоїди, гіперболічний тангенс є симетричним відносно нуля, але так само як і для сигмоїди, значення активації досягають насищення, що призводить до зникнення градієнта. На практиці гіперболічний тангенс сходитьться швидше ніж сигмоїда.

Для уникнення локальних мінімумів корисно додати лінійний вираз до гіперболічного тангенса:

$$g(z) = \tanh(z) + az, \quad (1.10)$$

де a – деяка константа.

Слід зазначити, що гіперболічний тангенс є масштабованою функцією сигмоїди:

$$\tanh(z) = 2\sigma(2z) - 1 \quad (1.11)$$

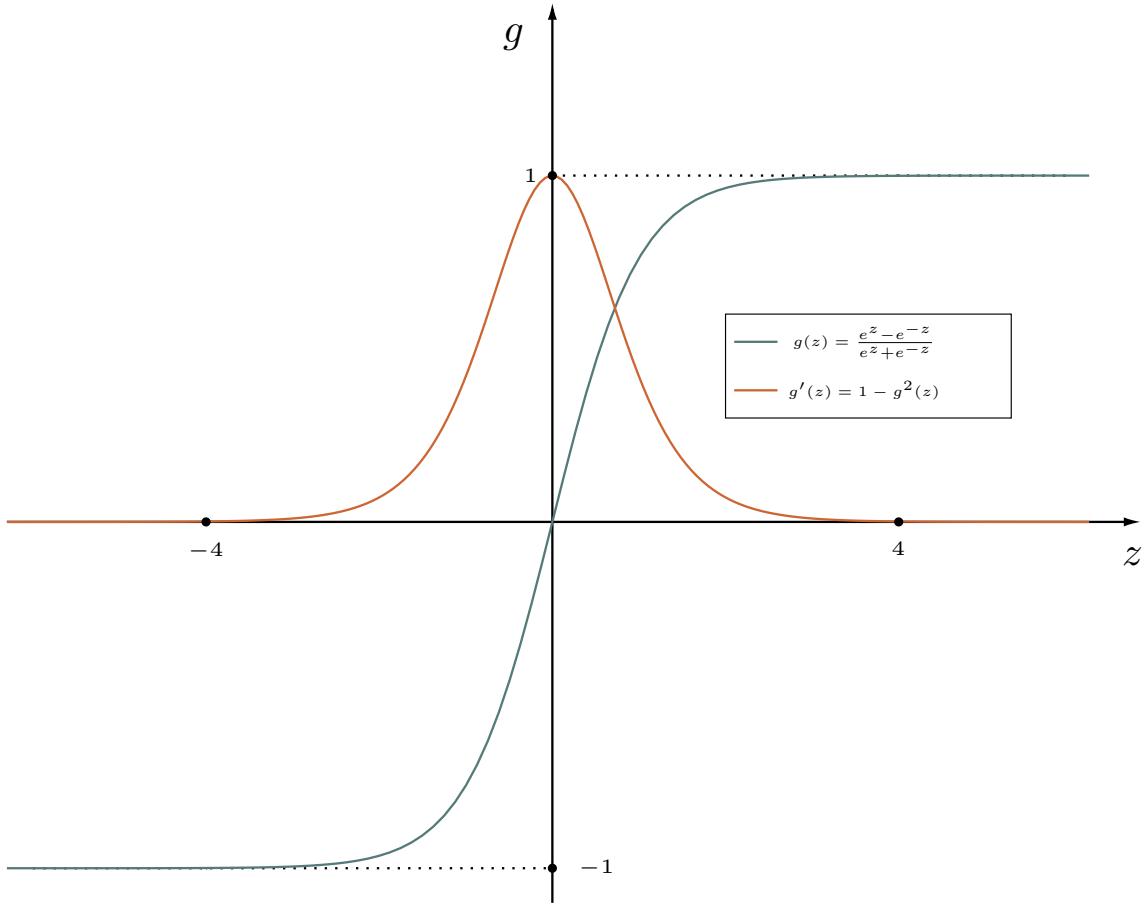


Рисунок 1.11 — Гіперболічний тангенс та його градієнт

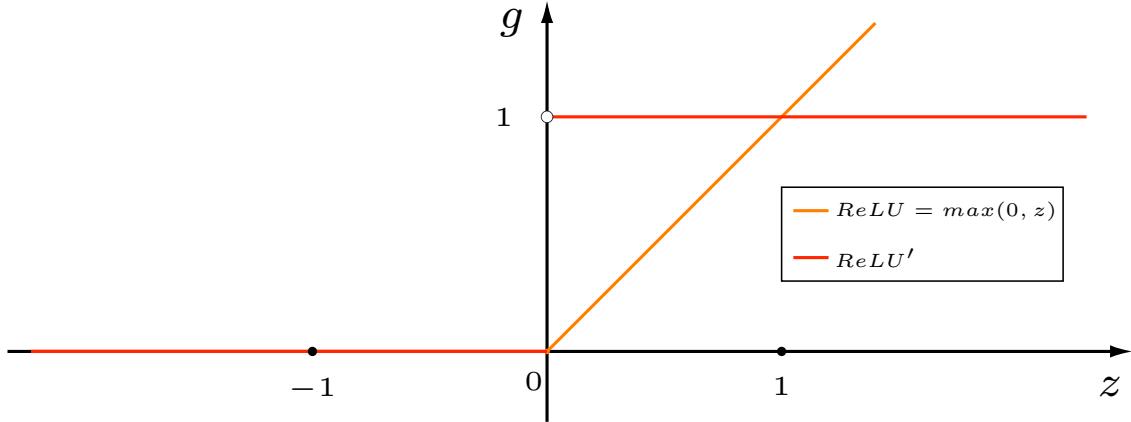


Рисунок 1.12 — ReLU та градієнт

ReLU та її модифікації (PReLU, ELU, leaky ReLU, Maxout) мають важливе значення для найсучасніших архітектур нейронних мереж [27, 28] у досягненні точності моделей на рівні людини. ReLU є просто пороговою функцією: для від'ємних вхідних значень сигналу активація функції дорівнює 0, а для додатних – значенню вхідного сигналу. Графічне представлення ReLU разом з своїм градієнтом показано на рисунку 1.12.

ReLU є ефективною з точки зору обчислень, оскільки не потребує експоненційних розрахунків, також вирішує проблему зникнення градієнтів для додатних вхідних сигналів. Однак, як можна побачити з рисунку 1.12, ReLU не є симетричною відносно 0 і для від'ємних вхідних значень градієнт дорівнює 0. Експериментально було показано, що навчання глибинних згорткових нейронних мереж з ReLU відбувається у декілька разів швидше (~ 6 разів) порівняно з використанням сигмоїди або гіперболічного тангенса [29].

Leaky ReLU використовується для вирішення проблеми зникнення градієнта в ReLU для від'ємних вхідних числових значень. Замість того, щоб для усіх від'ємних вхідних значень встановлювати градієнт в 0, leaky ReLU дозволяє малий ненульовий градієнт, коли нейрон є неактивним.

$$\text{Leaky ReLU} = \max(\varepsilon z, z), \quad (1.12)$$

де $0 < \varepsilon \ll 1$ – константа.

У випадку параметризації ReLU (PReLU) заміняється константа ε на змінну, яка є навчальним параметром разом з іншими параметрами нейронної мережі.

ELU (Exponential Linear Unit) має такий самий вигляд як ReLU для додатних вхідних значень і приймає експоненційну залежність для від'ємних вхідних значень. ELU є ще однією спробою вирішення проблеми зникнення градієнта в ReLU для від'ємних вхідних числових значень та симетрії відносно нуля. Функція має наступний математичний запис:

$$\text{ELU} = \max(\varepsilon[e^z - 1], z), \quad (1.13)$$

де $0 < \varepsilon \ll 1$ – константа.

Maxout є узагальненням функцій ReLU та leaky ReLU [28], повертає максимум від вхідних сигналів x , що надходять на нейрон, призначена для використання разом із дропаутом (dropout). Кожний Maxout нейрон здійснює таке обчислення:

$$\text{Maxout} = \max(w_1^\top x + b_1, w_2^\top x + b_2) \quad (1.14)$$

Слід зазначити, що Maxout об'єднує переваги ReLU (лінійний режим операцій, відсутні насичення) та вирішує проблему зникнення градієнтів в ReLU. Однак, основним недоліком Maxout є подвоєння кількості навчальних параметрів для кожного нейрона, що призводить до збільшення загальних витрат під час розрахунків.

1.6 Функції втрат

1.6.1 Евклідові втрати

Евклідові втрати також відомі як L2 втрати, найчастіше використовуються для задачі регресії. Математичний запис представляє собою квадрат відстані між прогнозованим значенням моделі \hat{y} та очікуваним y :

$$J = \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \|\hat{y} - y\|_2^2 \quad (1.15)$$

Усереднене значення L2 втрат (mean squared error - MSE):

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{m} \|\hat{y} - y\|_2^2 \quad (1.16)$$

Слід зазначити, що квадратичний вираз робить втрати L2 особливо сприйнятливими до викидів. Для того, щоб уникнути цю чутливість часто використовують середнє квадратичне відхилення (root-mean-square error

- RMSE), яке є квадратним коренем MSE:

$$\mathcal{L} = \sqrt{\frac{\sum_{i=1}^m (\hat{y}_i - y_i)^2}{m}} = \frac{\|\hat{y} - y\|_2}{\sqrt{m}} \quad (1.17)$$

Інший поширений варіант – логарифмічна середньоквадратична втрата, яка використовується для уникнення викидів, коли прогнозовані значення моделі та очікувані є великими числами:

$$J = \frac{1}{m} \sum_{i=1}^m (\log(\hat{y}_i + 1) - \log(y_i + 1))^2 \quad (1.18)$$

1.6.2 L1 втрати

L1 втрати визначають абсолютну відстань між прогнозованим значенням моделі \hat{y} та очікуваним y :

$$J = \sum_{i=1}^m |\hat{y}_i - y_i| = \|\hat{y} - y\|_1 \quad (1.19)$$

Усереднене значення L1 втрат (mean absolute error - MAE):

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^m |\hat{y}_i - y_i| = \frac{1}{m} \|\hat{y} - y\|_1 \quad (1.20)$$

L1 втрати іноді мають переваги над L2 втратами, особливо коли розмір простору ознак великий. Це пов'язано з тим, що L1 втрати є стійкішими до викидів порівняно з L2 втратами, а також є швидшими в обчисленнях. Однак, варто звернути увагу на те, що похідна L1 втрат не існує в нулі, тому часто використовується згладжений її варіант:

$$J = \sum_{i=1}^m \begin{cases} \frac{1}{2}(\hat{y}_i - y_i)^2, & \text{якщо } \hat{y}_i - y_i < 1 \\ \hat{y}_i - y_i - \frac{1}{2}, & \text{інакше} \end{cases} \quad (1.21)$$

Слід зазначити, що такий варіант функції втрат L1, як зазначено у рівнянні (1.21), дозволяє усунути проблему вибуху градієнтів (*exploding gradients*) [30].

Однак, слід пам'ятати, що використання L1 та L2 втрат для оцінок ймовірностей від активації сигмоїди або softmax призводить до появи немонотонних часткових похідних від функцій L1 та L2 відносно виходу кінцевого шару [31]. На неправильно класифікованих прикладах навчання мережі буде значно сповільнено через зникнення градієнта на кінцях сигмоїд. Саме через це L1 та L2 втрати зазвичай не використовуються разом із активацією сигмоїди або softmax, а частіше використовуються з ReLU.

1.6.3 Відстань Кульбака – Лейблера

На відміну від норми L2, яка вимірює відстань між двома точками в евклідовому просторі, відстань Кульбака–Лейблера (Kullback-Leibler divergence) вимірює різницю між двома розподілами ймовірностей $p(x)$, $q(x)$:

$$KL(p||q) = \sum_i p(x_i) \log \frac{p(x_i)}{q(x_i)} \quad (1.22)$$

Слід відмітити, що відстань KL є невід'ємною, асиметричною мірою ($KL(p||q) \neq KL(q||p)$). Походить з теорії інформації та поняття ентропії. Ентропія розподілу ймовірності $p(x)$ є кількісною оцінкою величини інформації, що міститься в даних:

$$H = - \sum_i p(x_i) \log p(x_i) \quad (1.23)$$

За величиною ентропії можна кількісно визначити, скільки інформації було втрачено, якщо замінити деякий розподіл параметризованим наближенням, тим самим встановивши на скільки згадане наближення відхиляється від очікуваного розподілу ймовірностей.

1.6.4 Перехресна втрата ентропії: бінарна

Перехресна втрата ентропії (cross-entropy loss) дозволяє визначити відстань між емпіричним розподілом даних та прогнозованим розподілом за моделлю. На практиці для вирішення задачі бінарної класифікації за допомогою нейронних мереж, на вихідному шарі мережі використовують, як правило, сигмоїду з метою спочатку оцінити ймовірності приналежності вхідних даних до певного класу як p та $1 - p$, а уже потім здійснити кількісне визначення перехресних втрат ентропії.

Наприклад, нехай ми маємо навчальний набір даних $D_{train} = (x_i, y_i)_{i=1}^m$ з дискретним розподілом ймовірності $q(x)$ для ановованих міток $y \in \{0, 1\}$. Якщо навчена модель прийме розподіл ймовірності прогнозу $p(x)$, тобто $p(y=1|x) = \hat{y}(x)$, ми можемо обчислити відстань KL між $p(x)$ і $q(x)$ як

$$\begin{aligned} KL(q||p) &= \sum_{j=1}^2 q(y_j|x) \log \frac{q(y_j|x)}{p(y_j|x)} = \\ &= \sum_{j=1}^2 \left[q(y_j|x) \log q(y_j|x) - q(y_j|x) \log p(y_j|x) \right] = \\ &= - \sum_{j=1}^2 q(y_j|x) \log p(y_j|x) + S_q = \\ &= -q(y=1|x) \log p(y=1|x) - q(y=0|x) \log p(y=0|x) + S_q = \\ &= -\left[y \log \hat{y} + (1-y) \log(1-\hat{y}) \right] + S_q, \end{aligned} \quad (1.24)$$

де S_q – константа.

Вираз бінарної перехресної втрати ентропії для одного навчального прикладу:

$$L = -\left[y \log \hat{y} + (1-y) \log(1-\hat{y}) \right] \quad (1.25)$$

Таким чином, як можна побачити з рівняння (1.24), мінімізація бінарної перехресної втрати ентропії еквівалентна мінімізації відстані KL.

1.6.5 Перехресна втрата ентропії: багатокласова

Перехресну втрату ентропії можна узагальнити на задачу багатокласової класифікації, де серед усіх можливих класів лише один клас присутній на кожному прикладі. Нехай маємо m прикладів для кожного з яких створено свій вектор y , який складається з n ановованих міток, де кожна мітка відповідає певному класу $c = \{c_1, c_2, \dots, c_n\}$, а цільові значення виражуються двійково:

$$y_i = \begin{cases} 1, & \text{якщо } i = c_i \\ 0, & \text{інакше} \end{cases} \quad (1.26)$$

Деякі приклади вектора y можна подати так:

$$y = \begin{bmatrix} y_{c_1} \\ y_{c_2} \\ \vdots \\ y_{c_n} \end{bmatrix} \quad y_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad y_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \dots \quad y_k = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \quad (1.27)$$

Якщо p_i – ймовірність для кожного класу, тоді $\sum_i p_i = 1$. Для отримання такої ймовірності оцінки, на вихідному шарі мережі використовують активаційну функцію softmax (рівняння (1.9)) замість сигмоїди.

Багатокласова перехресна втрата ентропії для одного навчального прикладу визначається таким чином:

$$L = - \sum_{i=1}^n y_i \log \hat{y}_i \quad (1.28)$$

1.6.6 Перехресна втрата ентропії із декількома мітками

Ще один випадок визначення перехресної втрати ентропії застосовується до задачі багатокласової класифікації, де кілька міток можуть приймати значення 1 одночасно (наприклад, на зображені одноважно присутні кілька об'єктів, що відносяться до різних категорій). У цьому випадку функція softmax більше не застосовується. Деякі приклади вектора y можна подати так:

$$y = \begin{bmatrix} y_{c_1} \\ y_{c_2} \\ \vdots \\ y_{c_n} \end{bmatrix} \quad y_1 = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad y_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad \dots \quad y_k = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \quad (1.29)$$

Типовою практикою є повернення до використання сигмоїди на вихідному шарі мережі, щоб передбачувані ймовірності кожного класу були незалежними від ймовірностей інших класів. У цьому випадку перехресна втрата ентропії для одного прикладу визначається таким чином:

$$L = - \sum_{i=1}^n \left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right] \quad (1.30)$$

1.7 Метрики оцінки

Завдання з глибинного навчання можуть бути досить складними та важко вимірюваними з точки зору визначення ефективності: яка мережа є кращою за іншу? У деяких простих випадках, таких як регресія, функція втрат, що використовується для навчання мережі, може бути хорошою метрикою для визначення ефективності роботи мережі. Однак, для багатьох завдань існують метрики оцінки, які одним числом вказують, наскільки добре працює мережа з точки зору вирішення поставленого завдання. Ці метрики оцінки дозволяють швидко побачити якість моделі та легко порівняти різні моделі для одних і тих самих завдань [32].

Задача класифікації

Давайте розглянемо просту задачу бінарної класифікації, де ми намагаємося передбачити здорових пацієнтів та хворих на пневмонію. У нас є тестовий набір із 10 пацієнтів (див. рисунок 1.13), 9 з яких здорові (показано зеленими прямокутником), а 1 пацієнт має пневмонію (червоний прямокутник).



Рисунок 1.13 — Істинні мітки для тестового набору даних [32]

Для вирішення цього завдання ми навчили 3 моделі (Модель1, Модель2, Модель3), після чого хотіли б порівняти їх ефективність. Прогнози для кожної моделі на тестовому наборі показані нижче на рисунку 1.14.

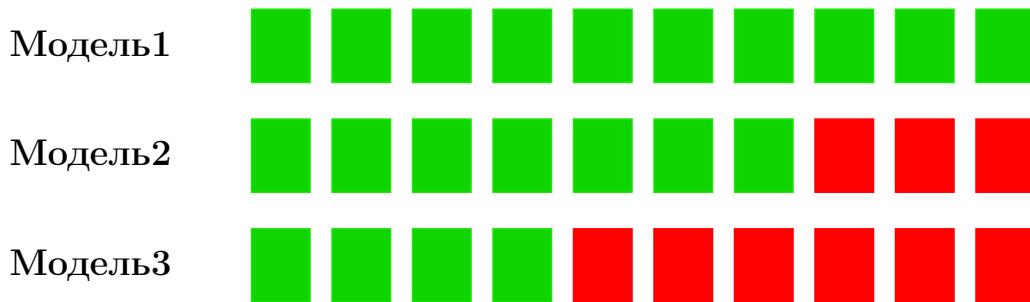


Рисунок 1.14 — Прогнози трьох моделей на тестовому наборі даних [32]

Точність (Accurasy). Для порівняння цих моделей, використаємо спочатку точність, яка дозволяє визначити відсоток правильно класифікованих прикладів із тестового набору:

$$\text{Точність}(f) = \frac{\sum_{x_i \in X_{test}} 1\{f(x_i) = y_i\}}{X_{test}} \quad (1.31)$$

Якщо ми використовуємо точність як метрику оцінки, отримаємо, що найкращою моделлю є Модель1:

$$\text{Точність}(M_1) = \frac{9}{10}, \quad \text{Точність}(M_2) = \frac{8}{10}, \quad \text{Точність}(M_3) = \frac{5}{10} \quad (1.32)$$

Загалом, коли тестовий набір є незбалансованим (для одного класу представлено значно більше даних ніж для іншого класу, такий сценарій характерний переважній більшості випадків!), точність не є хорошию метрикою для визначення ефективності моделей [32].

Матриця невідповідностей (Confusion Matrix). Метрика точності не розрізняє помилок зроблених моделлю в сторону хибно позитивних або хибно негативних значень. Матриця невідповідностей – це табличний формат, який показує більш детальну інформацію про кількість правильних та неправильних прогнозів моделі. Графічне представлення матриці невідповідностей подано на рисунку 1.15.

		Фактичні	
		+	-
Предбачені	+	Істинно позитивні	Хибно позитивні
	-	Хибно негативні	Істинно негативні

Рисунок 1.15 — Матриця невідповідностей для бінарної класифікації [32]

Повнота (Recall). Для визначення пневмонії надзвичайно важливо знайти усіх хворих на цю недугу. Модель, яка відносить хворих на пневмонію до категорії здорових є неприпустимою, оскільки пацієнти, яким потрібна допомога, залишатимуться без лікування. Отже, природне питання, яке слід задати при оцінці наших моделей: який відсоток серед усіх пацієнтів із пневмонією модель класифікувала як хворих на пневмонію? Метрика, яка дозволяє дати відповідь на це питання називається повнотою:

$$\text{Повнота} = \frac{\text{Істинно позитивні}}{\text{Істинно позитивні} + \text{Хибно негативні}} \quad (1.33)$$

Таким чином, повнота для розглянутих моделей прийме наступні значення:

$$\text{Повнота}(M_1) = \frac{0}{1}, \quad \text{Повнота}(M_2) = \frac{1}{1}, \quad \text{Повнота}(M_3) = \frac{1}{1} \quad (1.34)$$

Влучність (Precision). Припустимо, що лікування пневмонії є дуже дорогим і тому ви також хотіли б переконатися, що лікування отримують лише хворі на пневмонію. Інше логічне запитання, яке слід задати при оцінці наших моделей: який відсоток серед усіх пацієнтів, яких віднесено до категорії хворих на пневмонію насправді мають пневмонію? Метрика, яка дозволяє дати відповідь на це питання називається влучністю:

$$\text{Влучність} = \frac{\text{Істинно позитивні}}{\text{Істинно позитивні} + \text{Хибно позитивні}} \quad (1.35)$$

Влучність розглянутих моделей прийме наступні значення:

$$\text{Влучність}(M_1) = \frac{0}{0}, \quad \text{Влучність}(M_2) = \frac{1}{3}, \quad \text{Влучність}(M_3) = \frac{1}{6} \quad (1.36)$$

Міра F1 (F1 Score). Повнота і влучність є обидві корисні метрики, але використання кількох оцінювальних метрик ускладнює пряме порівняння моделей (з книги Ендрю Нг “Machine Learning Yearning”).

Міра F1 – це метрика, яка поєднує повноту та влучність, шляхом знаходження їх середньо гармонічного значення:

$$F1 = \frac{2}{\frac{1}{\text{Повнота}} + \frac{1}{\text{Влучність}}} \quad (1.37)$$

Таким чином, міра F1 для кожної моделі прийме наступні значення:

$$F1(M_1) = 0, \quad F1(M_2) = \frac{1}{2}, \quad F1(M_3) = \frac{2}{7} \quad (1.38)$$

Виявлення об'єктів

Intersection over Union (IoU) також відомий як коефіцієнт Жаккарда (Jaccard index) – це найпопулярніша метрика оцінки продуктивності моделей для таких завдань як семантична сегментація, виявлення та відстеження об'єктів. Задача виявлення об'єктів (детекція) складається з двох підзадач [32, 33, 34]:

1. Пошук та ідентифікація об'єктів (присвоєння мітки знайденому об'єкту)
2. Локалізація (визначення положення об'єкта на зображені шляхом розміщення його в обмежуючу рамку - bounding box).

Обмежувальні рамки та математичний вираз за яким визначають кількісну величину IoU показано на рисунку 1.16. Чим вище значення IoU, тим краще модель визначає положення та розмір знайденого об'єкта [32]. IoU є досить зручною метрикою, оскільки вона добре працює для будь-якого розміру та форми об'єктів. IoU разом з повнотою та влучністю лежать в основі повної метрики для виявлення об'єктів – mean average precision (mAP) [32].

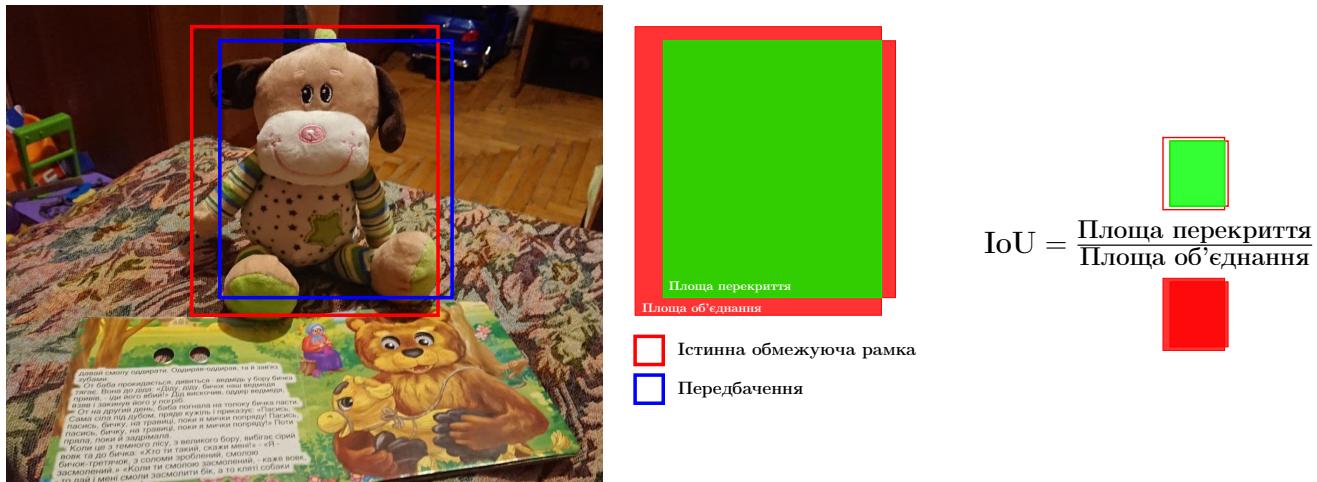


Рисунок 1.16 — Intersection over Union (IoU) [32, 33]

Average Precision (AP): the Area Under Curve (AUC). Детектори об'єктів створюють багато передбачень: для кожного зображення з тестового набору створюється своє передбачення, водночас кожне зображення може мати декілька передбачуваних об'єктів. Кожному передбачуваному об'єкту присвоюється значення довіри: показник, який вказує на скільки детектор упевнений у своєму передбаченні [32].

Ми можемо обирати різні порогові значення довіри для того, щоб визначити, які прогнози прийняті від детектора. Наприклад, якщо ми встановимо порогове значення 0.7, тоді кожен прогноз моделі зі значенням довіри більше 0.7 буде прийнято, а решта відхилено. Так як існує велика кількість різних порогових значень, які можна обрати, виникає логічне питання: як ми можемо оцінити ефективність детектора для певного порогового значення довіри? Щоб дати відповідь на це питання потрібно розглянути криву влучність-повнота

(precision-recall curve). Ми можемо виміряти влучність та повноту детектора для кожного порогового значення та прогнозу, після чого отримаємо одну точку даних. Для серії прогнозів буде отримано сукупність таких точок, після з'єднання яких отримаємо криву влучність-повнота [32]. Приклад кривої влучність-повнота показано на рисунку 1.17. Для різних порогових значень довіри буде отримано свою таку криву.

2-х класова крива влучність-повнота: $AP = 0.68$

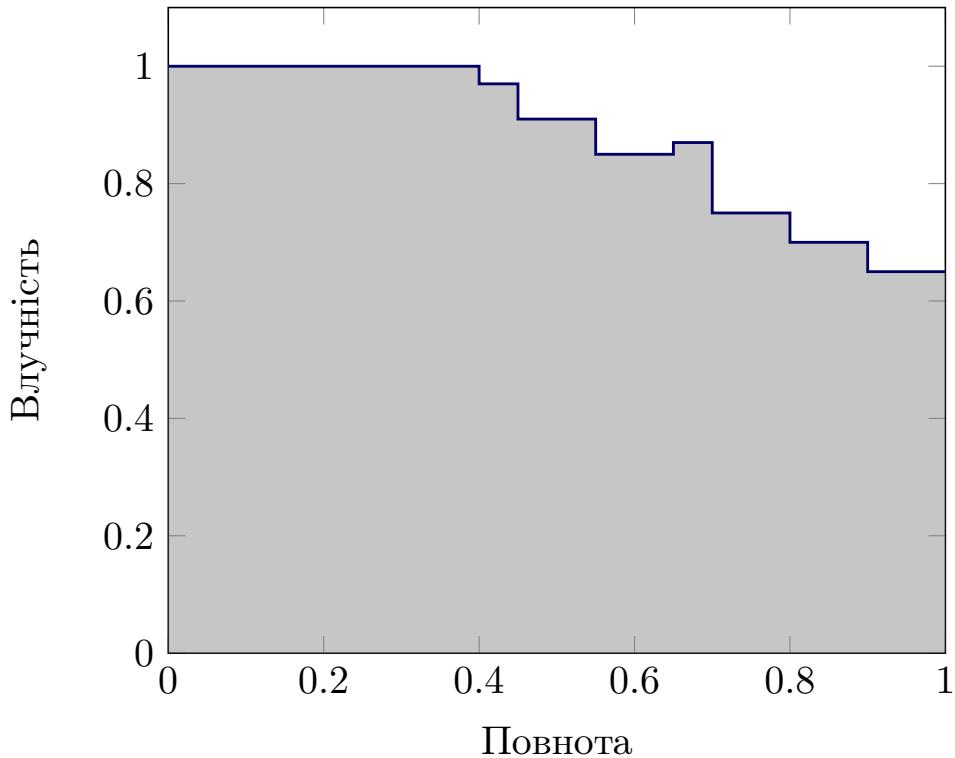


Рисунок 1.17 — Крива влучність-повнота. Темна лінія є кривою, а сіра зона – площа під кривою (area under the curve – AUC) [32]

Чим краща модель, тим вищі її значення влучності та повноти: це зміщує межу кривої (темну лінію) вгору та праворуч (рисунок 1.17). Ми можемо підсумувати ефективність моделі за однією метрикою, розрахувавши площину під кривою (показано сірим кольором – рисунок 1.17). Це дасть нам число від 0 до 1, де чим вище значення буде отримано, тим краще. Ця метрика широко відома як середня влучність (Average Precision - AP) [32].

Mean Average Precision (mAP). Виявлення об'єктів – це комплексне завдання: ми хочемо знайти усі об'єкти на зображенні, намалювати навколо кожного точні обмежувальні рамки та точно передбачити клас кожного об'єкта. Для оцінки ефективності моделі, яка вирішує таку задачу використовують метрику mean average precision (mAP). Для того, щоб обчислити mAP на всьому наборі даних, потрібно усереднити AP для кожного зображення та класу. Ця метрика використовується для загальних тестів моделей з виявлення об'єктів [32].

1.8 Мотивація та інтуїція

Для більшості задач машинного навчання вибір хороших ознак¹³ має першочергове значення. Навіть найкращий алгоритм не зможе продемонструвати хороших результатів, якщо для його навчання було використано погані ознаки. З іншого боку, вибір ознак часто є вкрай нетривіальним завданням. Наприклад, розглянемо кольорове зображення обличчя людини. Нехай у цьому випадку зображення обличчя людини буде представлено інтенсивністю пікселя для трьох кольорів: червоного, зеленої та синього (якщо використовується RGB-кодування). Тому, 1М піксельне кольорове зображення буде мати 3М ознак за якими ми можемо навчати нашу модель. З іншого боку, вираз обличчя людини, ймовірно, можна охарактеризувати ≤ 56 ознаками (на обличчі людини є ~ 56 м'язів). Отже, для ідентифікації конкретних виразів обличчя, дані великої розмірності можуть бути представлені відповідними ознаками меншої розмірності.

Таким чином, ідеальний екстрактор (видобувач) ознак буде приймати на вхід зображення обличчя, а на вихід видавать видобуті ознаки, що характеризують вираз обличчя людини. Однак, на теперішній час, не існує ідеальних способів як це зробити. У традиційних методах розпізнавання образів, які були розроблені починаючи з 50-х років, ці екстрактори ознак були жорстко закодовані на основі суб'ективної інтуїції дослідників. Основна ідея, яка прийшла до нас разом з нейронними мережами, полягає в тому, що хороші ознаки можуть бути вивчені мережею безпосередньо з даних, таким чином більше непотрібно досліднику видобувати їх вручну.

З іншого боку, базова модель слабо розвинулася з часу її створення починаючи з 1950-х років. Перша машина («Марк-1», Френк Розенблат в 1957 р.), яка реалізовувала алгоритм перцептрона (булева модель нейрона Мак-Калока&Пітса) була лінійним класифікатором, побудованим поверх простого жорстко прописаного екстрактора ознак. До сьогоднішнього дня на практиці для вирішення прикладних задач машинного навчання використовують ручне видобування ознак.

Звичайно, якби нейронна мережа була представлена лише лінійними шарами (нейронами), сукупний ефект також був би лінійним і ми могли б згорнути всю архітектуру мережі лише в один шар (нейрон). Це пояснюється тим, що результат комбінації лінійних перетворень залишається лінійним перетворенням. З іншого боку, введення нелінійності відкриває можливість для побудови глибоких мереж з кількох шарів, оскільки їх неможливо лінійно об'єднати, таким чином кожен окремий нейрон та шар буде вивчати різні ознаки.

Загалом, у цьому підручнику ми будемо переважно розглядати методи контролюваного навчання на основі градієнта, де результат прогнозу для деякого вхідного вектора ознак $\mathbf{X}^{(i)}$ представлений функцією:

$$\hat{y}^{(i)} = f(\mathbf{W}, b, \mathbf{X}^{(i)}), \quad (1.39)$$

де \mathbf{W} – деякий вектор/матриця вагових коефіцієнтів (навчальні параметри), b – зсув моделі (скаляр/вектор, також навчальний параметр), f – деяка функція аргументів (може бути нелінійною). Якість моделі визначається на основі деякої цільової функції втрат J :

$$L(\mathbf{W}, b, \mathbf{X}^{(i)}, y^{(i)}) = L(f(\mathbf{W}, b, \mathbf{X}^{(i)}), y^{(i)}) = L(\hat{y}^{(i)}, y^{(i)}) \quad (1.40a)$$

$$J(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^n L(\hat{y}^{(i)}, y^{(i)}), \quad (1.40b)$$

де $L(\hat{y}^{(i)}, y^{(i)})$ – міра невідповідності (функція втрат) між істинною величиною y та оцінюваною величиною \hat{y} для одного навчального прикладу, тут розглянуто деякий i -й приклад з набору даних. Наша мета – знайти таку конфігурацію навчальних параметрів моделі, які будуть відповідати глобальному мінімуму цільової

¹³Ознака (фіча) – це окрема властивість чи характеристика у даних, від якої безпосередньо залежить вихідний результат передбачення моделі.

функції втрат ¹⁴ J на валідаційній/тестовій вибірці даних. Після того як модель зробить передбачення, ми розраховуємо втрати та оновлюємо ваги і зсув нашої моделі в найкращому напрямку, для цього обчислюємо градієнти цільової функції втрат відносно навчальних параметрів:

$$\frac{\partial J(\mathbf{W}, b, \mathbf{X}, y)}{\partial \mathbf{W}} = \frac{\partial J(f(\mathbf{W}, b, \mathbf{X}), y)}{\partial \hat{y}} \frac{\partial f(\mathbf{W}, b, \mathbf{X})}{\partial \mathbf{W}} \quad (1.41a)$$

$$\frac{\partial J(\mathbf{W}, b, \mathbf{X}, y)}{\partial b} = \frac{\partial J(f(\mathbf{W}, b, \mathbf{X}), y)}{\partial \hat{y}} \frac{\partial f(\mathbf{W}, b, \mathbf{X})}{\partial b} \quad (1.41b)$$

Градієнт показує напрямок найкрутішого зростання у заданій точці кривої. Оскільки нам потрібно мінімізувати втрати моделі, маємо взяти градієнт з від'ємним знаком – це буде напрямок найкрутішого спадання цільової функції. Далі виконуємо оновлення навчальних параметрів за правилом:

$$W = W - \alpha \frac{\partial J(\mathbf{W}, b, \mathbf{X}, y)}{\partial \mathbf{W}} \quad (1.42a)$$

$$b = b - \alpha \frac{\partial J(\mathbf{W}, b, \mathbf{X}, y)}{\partial b}, \quad (1.42b)$$

де α – крок навчання (швидкість навчання). Залежно від того, скільки навчальних прикладів використовується для цього оновлення, ми отримуємо різні алгоритми оптимізації: стохастичний, пакетний та мініпакетний градієнтний спуск (до цього ми повернемося пізніше).

З поданого вище шаблону виникають три запитання:

1. Яку архітектуру, наприклад, нейронної мережі, ми повинні використовувати для вивчення $f(\mathbf{W}, b, \mathbf{X})$?
2. Яку функцію втрат $L(\hat{y}, y)$ слід використовувати?
3. Чи варто використовувати простий градієнтний спуск (1.42) чи якийсь більш досконалій алгоритм оптимізації?

1.8.1 Інжиніринг ознак

Інжиніринг (конструювання) ознак є дуже важливим етапом для створення моделі. Він передбачає видобування та вибір ознак. Під час видобування ознак витягаються з даних усі ознаки, які характеризують поставлену задачу. Під час вибору – визначаються усі найбільш важливі ознаки з метою покращення продуктивності моделі.

На рисунку 1.18 розглянуто інжиніринг ознак на прикладі класифікації зображень. Ручне вилучення ознак з даних вимагає глибоких знань як задачі, яка вирішується, так і предметної галузі. Крім того, цей спосіб є трудомістким. Ми можемо автоматизувати процес конструювання ознак за допомогою глибинного навчання!

¹⁴Усереднені втрати моделі для всієї вибірки.

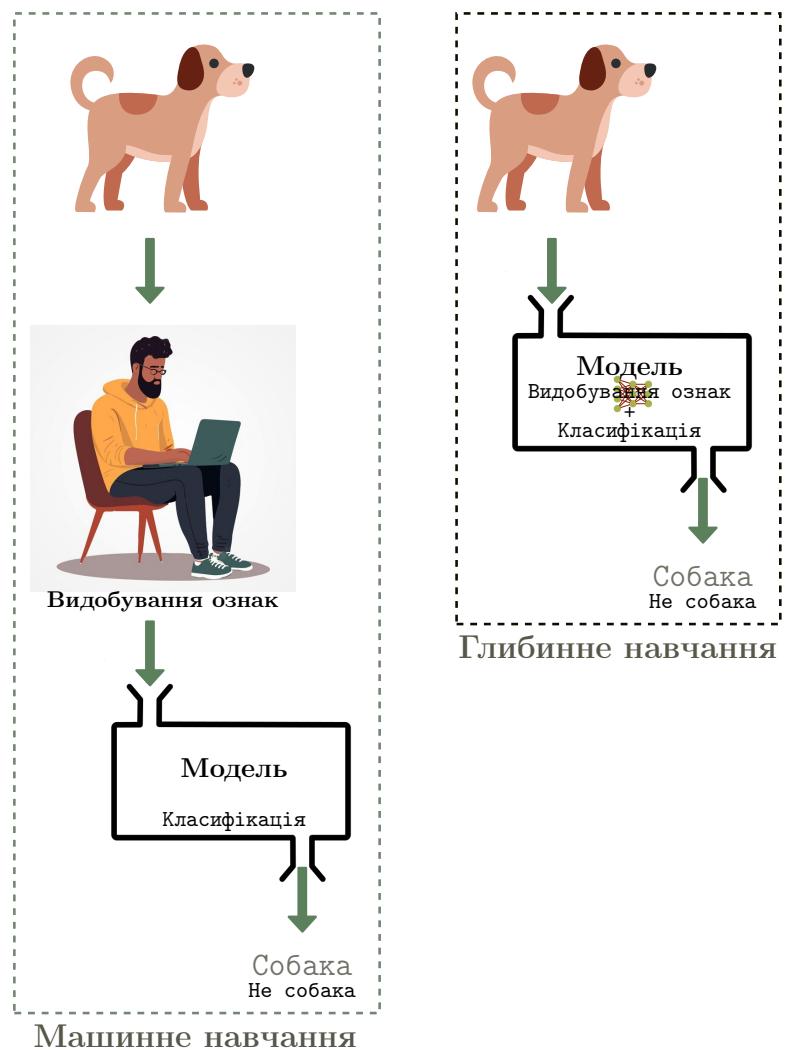


Рисунок 1.18 — Інженіринг ознак в машинному навчанні та глибинному навчанні

Розділ 2

Стратегії навчання та ініціалізації

“Грам власного досвіду коштує дорожче тонни чужих повчань.”

– Магатма Ганді

2.1 Пряме та зворотне поширення

У цьому параграфі розглянуто пряме та зворотне поширення на прикладі одновимірної та багатовимірної регресії, двошарової лінійної та нелінійної нейронної мережі.

2.1.1 Одновимірна регресія

Нехай ми маємо набір даних: $D = (x, y) = (x_i, y_i)_{i=1}^m$, який складається з m прикладів. Іншими словами, $x = (x_1, x_2, \dots, x_m)$ та $y = (y_1, y_2, \dots, y_m)$ – вектори-рядки з m скалярних прикладів. Завдання полягає у пошуку таких параметрів w та b , щоб отримана на виході функція $\hat{y} = wx + b$ якомога оптимальніше описувала обраний набір даних. Це завдання може бути вирішено за допомогою оптимізаційних алгоритмів (наприклад, градієнтний спуск).

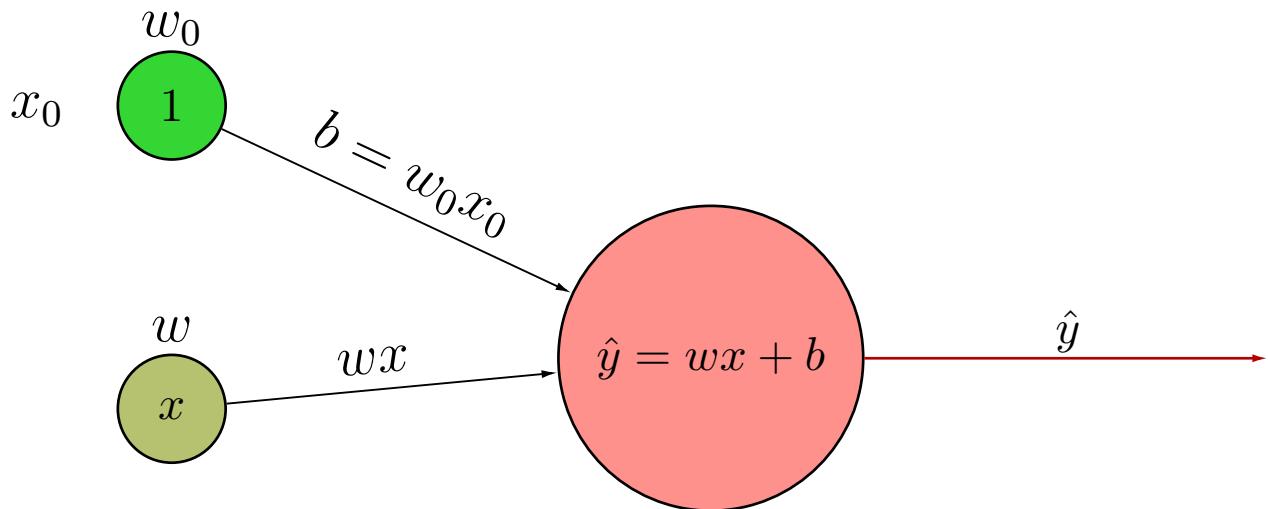


Рисунок 2.1 – Одновимірна регресія

Пряме поширення

Перший крок оптимізаційного алгоритму полягає у визначенні втрат моделі. Для цього потрібно знати вихідне значення моделі \hat{y} та очікуване y . Для визначення втрат використано MSE, формула (1.16).

$$\hat{y} = wx + b \quad (2.1a)$$

$$\mathcal{L}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{m} \|\hat{y} - y\|_2^2 \quad (2.16)$$

Зворотне поширення

Другий крок полягає у визначенні градієнту від функції втрат за навчальними параметрами w та b . Це означає, що потрібно обчислити похідні. Слід зазначити, що значення отримані унаслідок прямого поширення будуть використані у виразах градієнтів.

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{2}{m} (\hat{y} - y) x^\top \quad (2.2a)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{m} (\hat{y} - y) \quad (2.2b)$$

Оновлення параметрів

Третій крок полягає в оновленні навчальних параметрів w та b у напрямку спадання функції втрат. Правила за якими здійснюється оновлення цих параметрів мають наступний математичний запис:

$$w \leftarrow w - \alpha \frac{\partial \mathcal{L}}{\partial w} \quad (2.3a)$$

$$b \leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b}, \quad (2.3b)$$

де α – швидкість навчання.

2.1.2 Багатовимірна регресія

Багатовимірна регресія використовує дві і більше ознаки, які впливають на вихідний результат моделі. Тобто, розглянемо випадок, коли X – матриця ознак розміром $(n \times m)$, а y – усе ще вектор-рядок розміром $(1 \times m)$. Замість одного скалярного значення ваг як у випадку з одновимірною регресією, ваги багатовимірної регресії є вектором-рядком (один елемент на ознаку) розміром $(1 \times n)$. Параметр зсуву усе ще є скаляром.

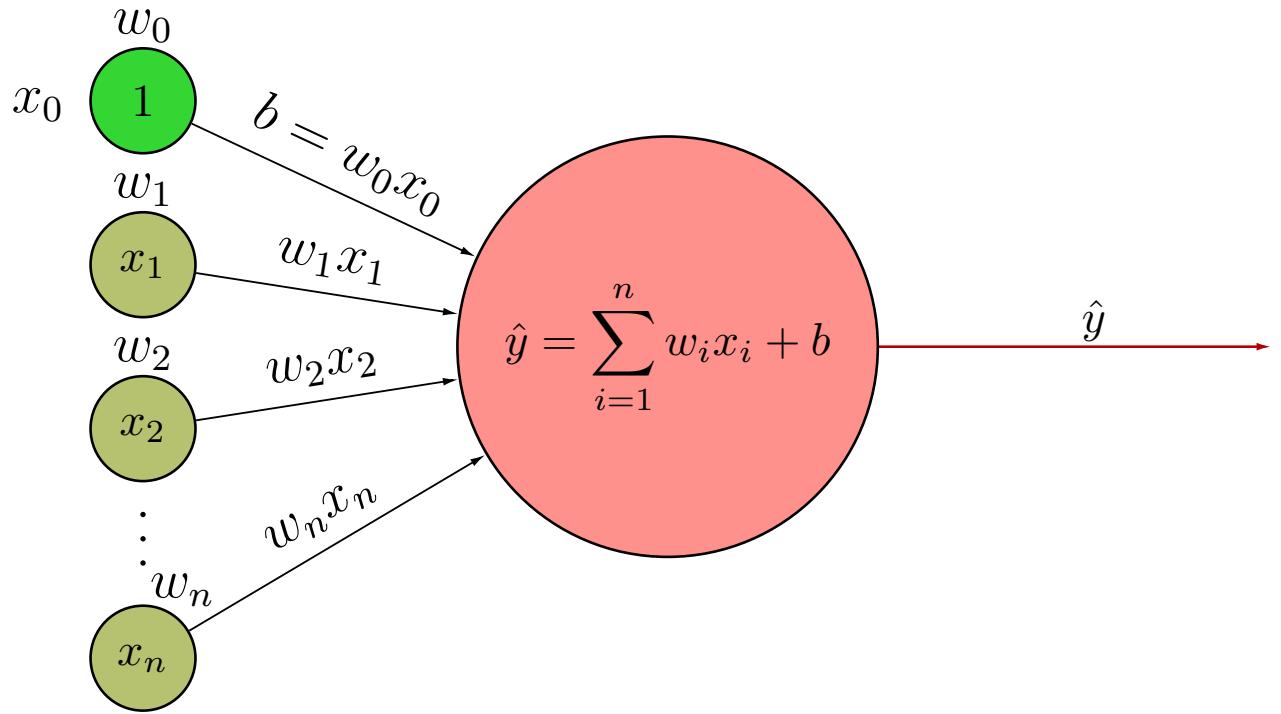


Рисунок 2.2 — Багатовимірна регресія

Пряме поширення

$$\hat{y} = wX + b \quad (2.4a)$$

$$\mathcal{L}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{m} \|\hat{y} - y\|_2^2 \quad (2.4b)$$

Зворотне поширення

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{2}{m} (\hat{y} - y) X^\top \quad (2.5a)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{m} (\hat{y} - y) \quad (2.5b)$$

2.1.3 Лінійна двошарова мережа

Розглянемо випадок двох послідовних лінійних шарів. Введемо приховану змінну Z розміром $(k \times m)$, яка представляє вихідні значення першого прихованого лінійного шару. Перший шар параметризується матрицею ваг W_1 розміром $(k \times n)$ та зсувом b_1 розміром $(k \times 1)$. Другий шар (вихідний) вибрано таким самим як для багатовимірної регресії, але цього разу вхідним значенням для нього буде Z замість X .

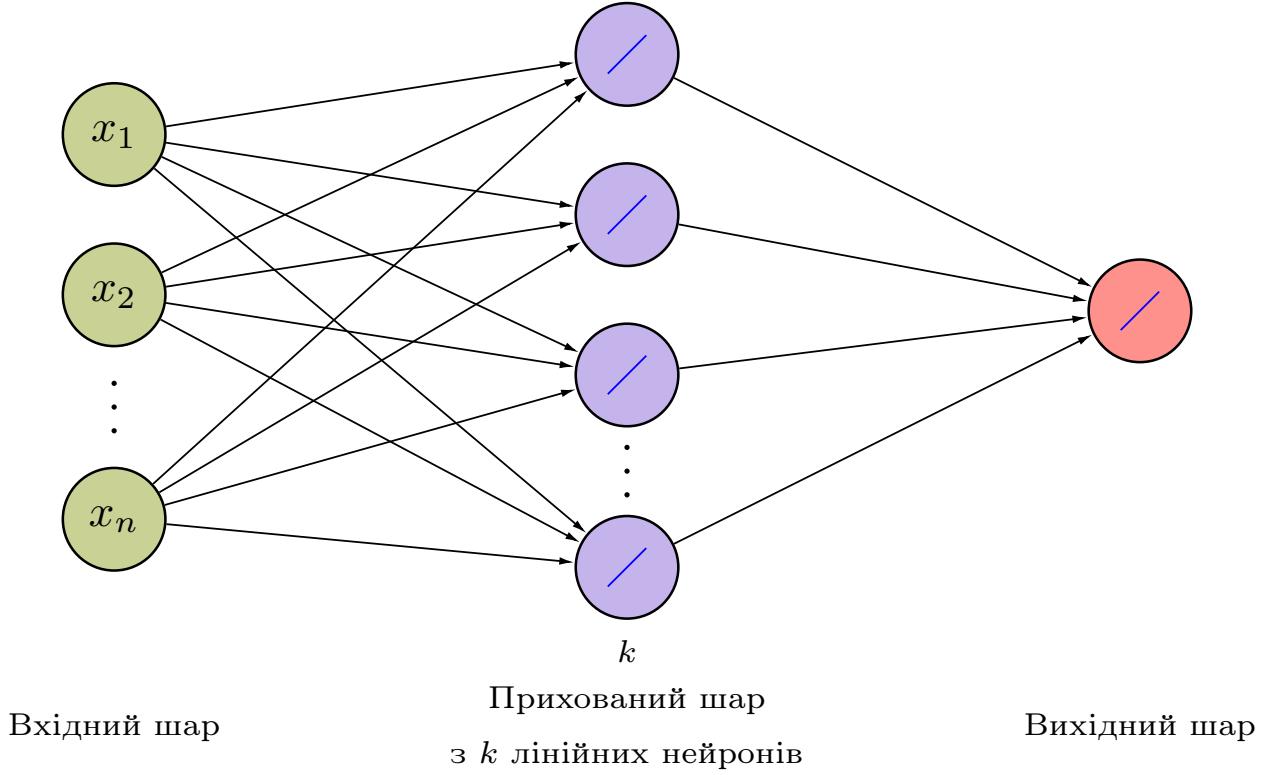


Рисунок 2.3 — Лінійна двошарова мережа

Пряме поширення

$$Z = W_1 X + b_1 \quad (2.6a)$$

$$\hat{y} = w_2 Z + b_2 \quad (2.6b)$$

$$\mathcal{L}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{m} \|\hat{y} - y\|_2^2 \quad (2.6c)$$

Зворотне поширення

$$\frac{\partial \mathcal{L}}{\partial W_1} = w_2^\top \frac{2}{m} (\hat{y} - y) X^\top \quad (2.7a)$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = w_2^\top \frac{2}{m} (\hat{y} - y) \quad (2.7b)$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{2}{m} (\hat{y} - y) Z^\top \quad (2.7c)$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{2}{m} (\hat{y} - y) \quad (2.7d)$$

2.1.4 Нелінійна двошарова мережа

У нелінійних нейронних мережах до отриманих значень на нейронах застосовується функція активації, яка вносить нелінійність у систему. Так, наприклад, для мережі, яка зображена на рисунку 2.4, перш ніж просто

подати отримані значення Z з першого прихованого шару на вхід другому шару мережі, спочатку має бути застосована до цих значень функція активації. Нехай для нашого випадку такою активаційною функцією буде сигмоїда. Деякі загально уживані функції активації подані у таблиці 1.3. На виході після активації отримаємо нові значення, які позначимо A , після чого ці значення будуть передані на вхід наступному шару. Шар виходу залишимо таким самим як для багатовимірної регресії, хоча на практиці часто до вихідного шару мережі може бути застосована своя функція активації.

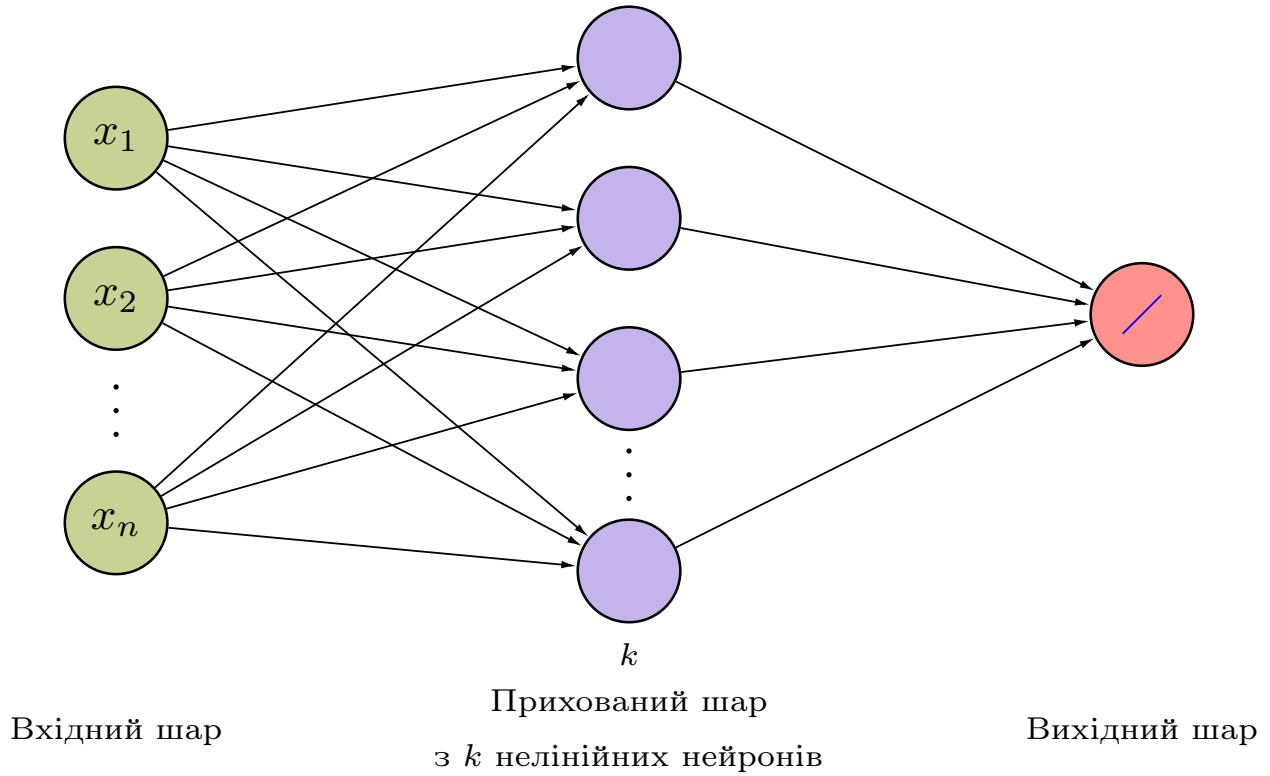


Рисунок 2.4 — Нелінійна двошарова мережа

Пряме поширення

$$Z = W_1 X + b_1 \quad (2.8a)$$

$$A = g(Z) = \frac{1}{1 + e^{-Z}} \quad (2.8b)$$

$$\hat{y} = w_2 A + b_2 \quad (2.8c)$$

$$\mathcal{L}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{m} \|\hat{y} - y\|_2^2 \quad (2.8d)$$

Зворотне поширення

$$\frac{\partial \mathcal{L}}{\partial W_1} = \left(\left(w_2^\top \frac{2}{m} (\hat{y} - y) \right) \odot A \odot (1 - A) \right) X^\top \quad (2.9a)$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \left(\left(w_2^\top \frac{2}{m} (\hat{y} - y) \right) \odot A \odot (1 - A) \right) \quad (2.9b)$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{2}{m} (\hat{y} - y) A^\top \quad (2.9c)$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{2}{m} (\hat{y} - y) \quad (2.9d)$$

\odot – математичний символ для добутку Адамара¹ (Hadamard product).

2.2 Ініціалізація нейронних мереж

Ініціалізація параметрів нейронної мережі може мати значний уплив на збіжність оптимізаційного алгоритму під час навчання. Відомо, що прості підходи ініціалізації дозволяють прискорити навчання, але вони потребують певної обережності [35]. У цьому параграфі розглянуто та пояснено як можна ефективно ініціалізувати параметри нейронної мережі.

2.2.1 Важливість ефективної ініціалізації

Для побудови алгоритму машинного навчання, зазвичай ми маємо визначити архітектуру (наприклад, логістичну регресію, машину опорних векторів, нейронну мережу) і навчити її на заданому наборі даних. Загальний навчальний процес для нейронних мереж уключає наступні кроки:

1. Ініціалізація параметрів мережі
2. Вибір оптимізаційного алгоритму
3. Цикл навчання:
 - (а) Пряме поширення вхідних даних.
 - (б) Обчислення функції втрат.
 - (в) Обчислення градієнтів функції втрат відносно параметрів мережі, використовуючи зворотне поширення.
 - (г) Оновлення кожного параметру через градієнти, що визначені відповідно до алгоритму оптимізації.

Після завершення процесу навчання ми можемо використовувати нові вхідні дані для визначення продуктивності отриманої моделі. Ініціалізація параметрів мережі може мати вирішальний уплив на кінцеву продуктивність моделі, тому для цього потрібно визначити правильний метод.

Якщо ініціалізувати усі ваги як нуль – це приведе до того, що нейрони будуть вивчати одні і ті ж ознаки під час навчання. Насправді, будь-який константний підхід ініціалізації параметрів буде працювати дуже погано. Для цього давайте розглянемо нейронну мережу з двома прихованими шарами ReLU, яка показана на рисунку 2.5. Припустимо, що ми ініціалізували усі зміщення як нуль, а усі ваги визначили деякою константою γ . Якщо ми прямо поширимо вхідні дані (x_1, x_2) через цю мережу, то на виході обох прихованих шарів отримаємо $relu(\gamma x_1 + \gamma x_2)$. Таким чином нейрони обох шарів будуть еволюціонувати симетрично протягом усього навчання, тим самим не даючи можливість різним нейронам вивчати різні ознаки в даних.

¹Добуток Адамара – поелементний добуток матриць.

²Якщо усі ваги мережі ініціалізувати (i) дуже малими або (ii) дуже великими значеннями, незважаючи на порушення симетрії, це приведе до (i) повільного навчання або (ii) розбіжності оптимізаційного алгоритму. І перший, і другий варіант не є оптимальними, саме тому вибір належних значень параметрів під час ініціалізації буде відігравати значну роль на ефективність навчання. Це буде досліджено та обговорено детальніше у параграфі 2.2.2.

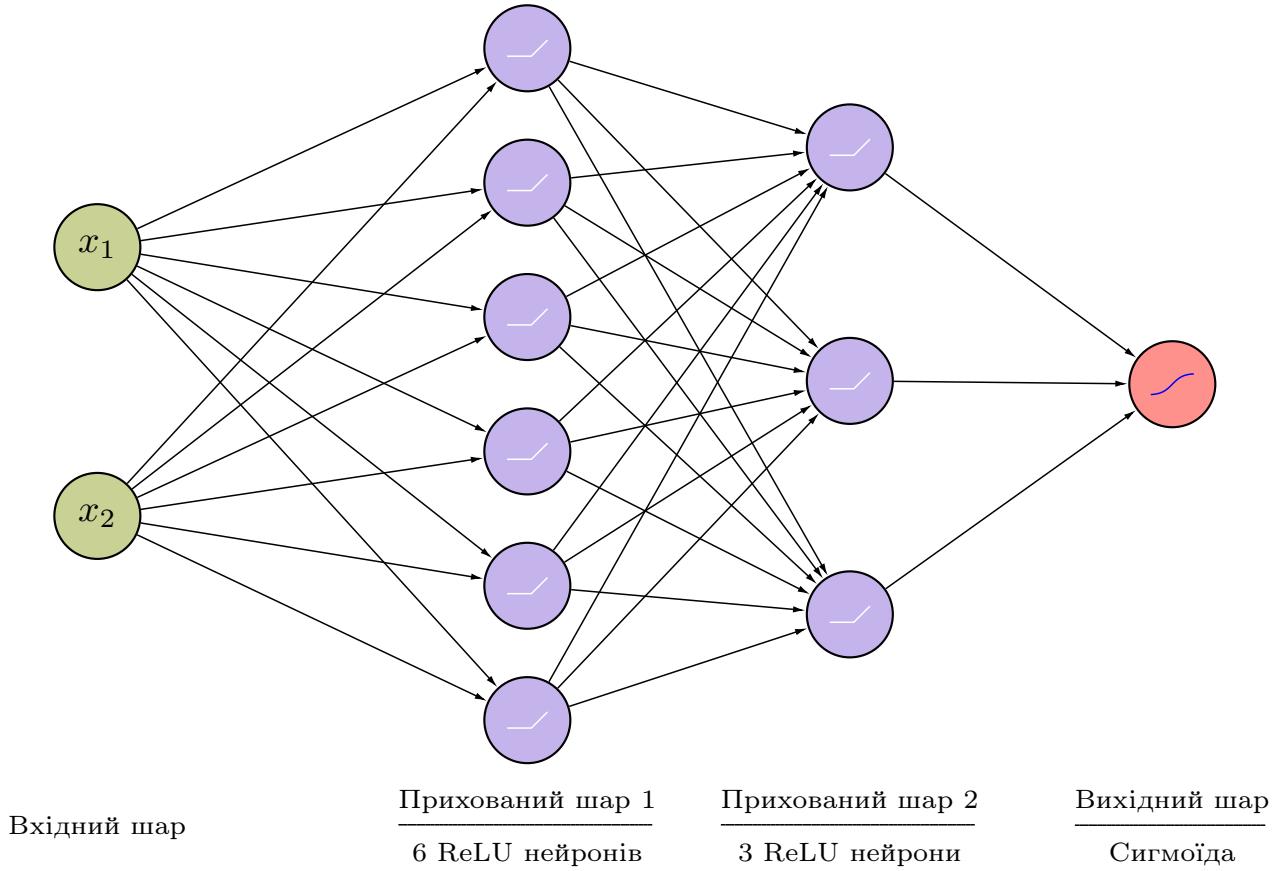


Рисунок 2.5 — Нейронна мережа з двома прихованими шарами ReLU та сигмоїдою на виході [35]

2.2.2 Проблема зникнення та вибуху градієнтів

Розглянемо нейронну мережу, яка складається з k прихованих шарів, кожний прихований шар має по два нейрони. На виході розміщено один нейрон. Графічне представлення цієї мережі подано на рисунку 2.6.

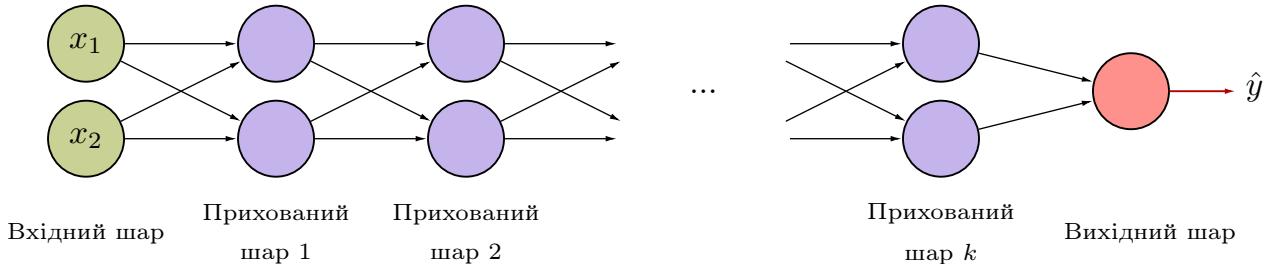


Рисунок 2.6 — Нейронна мережа з k прихованих шарів [35]

²Усі параметри нейронної мережі будуть ініціалізовані таким чином: зміщення на кожному нейроні дорівнюють нулю, а ваги отримано з нормальногоподілу $w_{ij} \sim \mathcal{N}(\mu = 0, \sigma^2)$.

На кожній ітерації циклу навчання (пряме поширення, обчислення втрат, зворотне поширення, оновлення параметрів) ми можемо спостерігати зворотне поширення градієнтів, яке посилюється або зводиться до мінімуму при переході від вихідного до вхідного шару. У цьому можна переконатись, розглянувши наступний приклад. Нехай усі активаційні функції нейронів цієї мережі будуть лінійними ($g(z) = z$). Тоді значення активації на вихідному шарі буде визначатись таким чином:

$$\hat{y} = a^{[L]} = W^{[L]}W^{[L-1]}W^{[L-2]} \dots W^{[2]}W^{[1]}x, \quad (2.10)$$

де $L = k + 1$ – глибина мережі, k – кількість прихованих шарів, $W^{[1]}, W^{[2]}, \dots, W^{[k]}$ – матриці вагових коефіцієнтів розміром $(2, 2)$, оскільки маємо два вхідні значення та по два нейрони у кожному прихованому шарі. З огляду на це та для ілюстративних цілей, якщо ми припустимо:

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = W \quad (2.11)$$

На виході отримаємо:

$$\hat{y} = a^{[L]} = W^{[L]}W^{L-1}x, \quad (2.12)$$

де $W^{[L]}$ – матриця вагових коефіцієнтів на вихідному шарі мережі, W^{L-1} – матриця вагових коефіцієнтів W у степені $L - 1$.

Який вплив на ефективність навчання нейронної мережі має ініціалізація ваг?

Випадок 1: Ініціалізація ваг дуже величими значеннями призводить до вибуху градієнтів. Нехай кожний ваговий коефіцієнт ініціалізовано значенням, яке трохи більше ніж в одиничній матриці:

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 1.6 & 0 \\ 0 & 1.6 \end{bmatrix} \quad (2.13)$$

Відповідно до рівняння (2.12) та враховуючи рівняння (2.13) отримаємо:

$$\hat{y} = a^{[L]} = W^{[L]}1.6^{L-1}x \quad (2.14)$$

Таким чином, кінцеве значення активації на вихідному шарі мережі $a^{[L]}$ експоненційно зростає зі збільшенням глибини мережі L . Використання цих значень активації під час зворотного поширення призводить до проблеми вибуху градієнтів. Тобто, градієнти функції втрат відносно параметрів приймають досить великих значення, а це призводить до флуктуацій (коливань) функції втрат навколо локального або глобального мінімуму.

Випадок 2: Ініціалізація ваг дуже малими значеннями призводить до зникнення градієнтів. Аналогічно розглянемо випадок, коли кожний ваговий коефіцієнт ініціалізовано значенням, яке трохи менше ніж в одиничній матриці:

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 0.6 & 0 \\ 0 & 0.6 \end{bmatrix} \quad (2.15)$$

Відповідно до рівняння (2.12) та враховуючи рівняння (2.15) отримаємо:

$$\hat{y} = a^{[L]} = W^{[L]}0.6^{L-1}x \quad (2.16)$$

Таким чином, кінцеве значення активації на вихідному шарі мережі $a^{[L]}$ експоненційно спадає зі збільшенням глибини мережі L . Використання цих значень активації під час зворотного поширення призводить до проблеми зникнення градієнтів. Тобто, градієнти функції втрат відносно параметрів приймають досить малі значення, а це призводить до збіжності функції втрат перш ніж вона досягне свого мінімуму.

Загалом, ініціалізація ваг досить великими або малими значеннями призведе до розбіжності або уповільнення навчання вашої мережі. Хоча проблему вибуху / зникнення градієнтів було продемонстровано на прикладі простих симетричних матриць вагових коефіцієнтів, спостереження подані вище можуть бути узагальнені на будь-які значення ініціалізації вагових коефіцієнтів, які є замалими або занадто великими.

2.2.3 Пошук оптимальних значень для ініціалізації

Для того, щоб градієнти активацій мережі не зникали та не вибухали потрібно дотримуватись таких основних правил:

1. Середнє значення активацій має дорівнювати нулю.
2. Дисперсія активацій повинна залишатися незмінною на кожному шарі.

Враховуючи ці два правила досягається те, що градієнти під час зворотного поширення не будуть приймати досить великих або досить малих значень у будь-якому шарі. Таким чином градієнт буде поширюватись до вихідного шару не зникаючи і не вибухаючи. Для прикладу розглянемо шар l . Пряме поширення для цього шару:

$$a^{[l-1]} = g^{[l-1]}(z^{[l-1]}) \quad (2.17a)$$

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \quad (2.17b)$$

$$a^{[l]} = g^{[l]}(z^{[l]}) \quad (2.17c)$$

При цьому має виконуватись:

$$\mathbb{E}[a^{[l-1]}] = \mathbb{E}[a^{[l]}] \quad (2.18a)$$

$$Var(a^{[l-1]}) = Var(a^{[l]}) \quad (2.18b)$$

Забезпечуючи середнє значення активацій нуль та підтримуючи дисперсію на кожному шарі рівною дисперсії вихідного шару гарантує відсутність проблем зникнення та вибуху градієнтів. Цей підхід застосовується як для прямого поширення (для активацій), так і для зворотного поширення (для градієнтів функції втрат відносно активацій). Рекомендований метод для ініціалізації параметрів мережі є ініціалізація Ксав'є (Xavier) [36] або будь-який похідний метод, який для кожного шару l :

$$W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}}\right) \quad (2.19a)$$

$$b^{[l]} = 0 \quad (2.19b)$$

Іншими словами, усі ваги для шару l обираються випадково з нормальногорозподілу з середнім $\mu = 0$ та дисперсією $\sigma^2 = \frac{1}{n^{[l-1]}}$, де $n^{[l-1]}$ – кількість нейронів у шарі $l - 1$. Зсуви ініціалізуються нулями.

2.2.4 Ініціалізація Ксав'є (Xavier)

Ініціалізація Ксав'є дозволяє підтримувати однакову дисперсію на кожному шарі. Для спрощення та кращого розуміння математичного апарату, який лежить в основі ініціалізації Ксав'є, будемо вважати, що значення активації для обраного шару нормально розподілені відносно нуля. Шар l нейронної мережі з яким будемо працювати зображене на рисунку 2.7. Він складається з $n^{[l]}$ нейронів, кожен нейрон активується за допомогою гіперболічного тангенса. Вхідне значення до шару l позначено $a^{[l-1]}$, а вихідне значення — $a^{[l]}$.

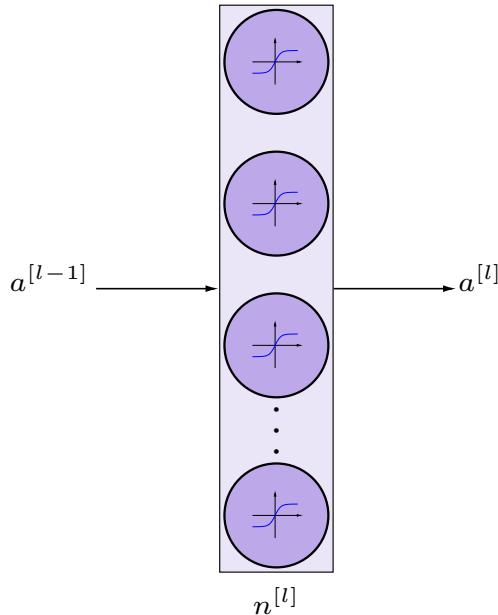


Рисунок 2.7 — Шар l нейронної мережі

Пряме поширення для шару l :

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \quad (2.20a)$$

$$a^{[l]} = \tanh(z^{[l]}) \quad (2.20b)$$

Мета полягає в отриманні взаємозв'язку між $\text{Var}(a^{[l-1]})$ та $\text{Var}(a^{[l]})$. Тоді буде зрозуміло як слід ініціалізувати ваги, щоб виконувалась рівність (2.18б).

Припустимо, що ваги $W^{[l]}$ ініціалізовано малими значеннями, зсуви $b^{[l]}$ ініціалізовано нулями, а вхідні дані є нормалізованими. На початку навчання модель знаходиться у лінійному режимі \tanh^3 , оскільки вираз (2.20a) є малим. Це означає, що значення активацій є також малими, а отже можна розкласти \tanh у ряд Тейлора та обмежитись членом першого порядку:

$$\tanh(z^{[l]}) \approx z^{[l]} \quad (2.21)$$

Враховуючи вираз (2.21) отримаємо:

$$\text{Var}(a^{[l]}) = \text{Var}(z^{[l]}) \quad (2.22)$$

³Важливими властивостями є парність ($\tanh(-z) = -\tanh(z)$) та лінійність поблизу нуля ($\tanh'(0) = 1$).

Більш того, $z^{[l]}$ – вектор:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} = \begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \\ \vdots \\ z_{n^{[l]}}^{[l]} \end{bmatrix}, \quad (2.23)$$

де $z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]} + b_k^{[l]}$.

Для простоти припустимо, що $b^{[l]} = 0$ (це в підсумку буде істинним припущенням, враховуючи вибір ініціалізації, який ми оберемо). Таким чином, дивлячись поелементно⁴ на рівняння (2.22), отримаємо:

$$Var(a_k^{[l]}) = Var(z_k^{[l]}) = Var\left(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}\right) \quad (2.24)$$

Основний математичний прийом полягає у винесенні знака суми за межи дисперсії у рівнянні (2.24). Для цього мають виконуватись наступні три припущення:

1. Ваги незалежні та однаково розподілені.
2. Вхідні дані незалежні та однаково розподілені.
3. Ваги та вхідні дані є взаємно незалежними.

Таким чином, отримаємо:

$$Var(a_k^{[l]}) = Var(z_k^{[l]}) = Var\left(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}\right) = \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]} a_j^{[l-1]}) \quad (2.25)$$

Ще один математичний прийом полягає у перетворенні дисперсії добутку у добуток дисперсій⁵. Дисперсію добутку можна виразити так:

$$Var(XY) = \mathbb{E}[X]^2 Var(Y) + Var(X) \mathbb{E}[Y]^2 + Var(X) Var(Y) \quad (2.26)$$

Підставляючи у рівняння (2.26) $X = w_{kj}^{[l]}$ та $Y = a_j^{[l-1]}$, отримаємо:

$$\begin{aligned} Var(w_{kj}^{[l]} a_j^{[l-1]}) &= \mathbb{E}[w_{kj}^{[l]}]^2 Var(a_j^{[l-1]}) + Var(w_{kj}^{[l]}) \mathbb{E}[a_j^{[l-1]}]^2 + \\ &\quad + Var(w_{kj}^{[l]}) Var(a_j^{[l-1]}) \end{aligned} \quad (2.27)$$

З першого припущення випливає, що $\mathbb{E}[w_{kj}^{[l]}]^2 = 0$, а з другого $\mathbb{E}[a_j^{[l-1]}]^2 = 0$, оскільки ваги ініціалізуються з середнім нуль, а вхідні дані є нормалізованими. У результаті отримаємо:

$$\begin{aligned} Var(z_k^{[l]}) &= \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]}) Var(a_j^{[l-1]}) = \\ &= \sum_{j=1}^{n^{[l-1]}} Var(W^{[l]}) Var(a_j^{[l-1]}) = n^{[l-1]} Var(W^{[l]}) Var(a^{[l-1]}) \end{aligned} \quad (2.28)$$

⁴Дисперсія вектора така ж як і дисперсія будь-якого його елемента, оскільки усі ці елементи були отримані незалежно з одного розподілу.

⁵Це можливо лише для незалежних випадкових величин.

Наведена вище рівність (2.28) випливає з першого припущення, згідно з яким:

$$Var(w_{kj}^{[l]}) = Var(w_{11}^{[l]}) = Var(w_{12}^{[l]}) = \dots = Var(W^{[l]}) \quad (2.29)$$

Аналогічно з другого припущення випливає:

$$Var(a_j^{[l-1]}) = Var(a_1^{[l-1]}) = Var(a_2^{[l-1]}) = \dots = Var(a^{[l-1]}) \quad (2.30)$$

Враховуючи рівняння (2.29) та (2.30), отримаємо:

$$Var(z^{[l]}) = Var(z_k^{[l]}) \quad (2.31)$$

Зрештою, враховуючи (2.22), (2.28) та (2.31), отримуємо таку рівність:

$$Var(a^{[l]}) = n^{[l-1]} Var(W^{[l]}) Var(a^{[l-1]}) \quad (2.32)$$

Для того, щоб дисперсія залишалась однаковою від шару до шару ($Var(a^{[l]}) = Var(a^{[l-1]})$) має виконуватись:

$$Var(W^{[l]}) = \frac{1}{n^{[l-1]}} \quad (2.33)$$

Слід зазначити, що на попередніх кроках не вибиралася якийсь конкретний шар мережі l . Таким чином, було показано, що вираз (2.32) справедливий для кожного шару мережі. Нехай L – вихідний шар мережі. Використовуючи вираз (2.32) на кожному шарі, можна встановити зв'язок між дисперсією вхідного шару та дисперсією вихідного шару:

$$\begin{aligned} Var(a^{[L]}) &= n^{[L-1]} Var(W^{[L]}) Var(a^{[L-1]}) = \\ &= n^{[L-1]} Var(W^{[L]}) n^{[L-2]} Var(W^{[L-1]}) Var(a^{[L-2]}) = \\ &= \dots = \\ &= \left[\prod_{l=1}^L n^{[l-1]} Var(W^{[l]}) \right] Var(x) \end{aligned} \quad (2.34)$$

Залежно від того, як будуть ініціалізовані ваги мережі, буде сильно змінюватись співвідношення між дисперсією вхідного шару та дисперсією вихідного шару. Розглянемо наступні три випадки:

$$n^{[l-1]} Var(W^{[l]}) \begin{cases} < 1 & \Rightarrow \text{Зникнення градієтів} \\ = 1 & \Rightarrow Var(a^{[L]}) = Var(x) \\ > 1 & \Rightarrow \text{Вибух градієтів} \end{cases} \quad (2.35)$$

Таким чином, щоб уникнути зникнення чи вибуху градієнтів, ми повинні встановити $n^{[l-1]} Var(W^{[l]}) = 1$ шляхом ініціалізації ваг з дисперсією $Var(W^{[l]}) = \frac{1}{n^{[l-1]}}$. На практиці використовують обидва наступні підходи для ініціалізацію Ксав'є: ініціалізують ваги як $\mathcal{N}(0, \frac{1}{n^{[l-1]}})$ або як $\mathcal{N}(0, \frac{2}{n^{[l-1]} + n^{[l]}})$. Дисперсія останнього розподілу є середнім гармонічним $\frac{1}{n^{[l-1]}}$ та $\frac{1}{n^{[l]}}$.

Слід ще раз підкреслити, що ініціалізація Ксав'є працює для активаційних функцій $tanh$. На практиці існує безліч інших методів для ініціалізації ваг мережі. Наприклад, якщо в прихованих шарах мережі використовується ReLU активація – прийнятим методом для цього випадку є ініціалізація Хі (Хе) [27], під час якого ваги ініціалізуються з розподілу $\mathcal{N}(0, \frac{2}{n^{[l-1]}})$. Як можна помітити, під час ініціалізації Хі береться подвоєна дисперсія з ініціалізації Ксав'є.

2.3 Оптимізація параметрів нейронних мереж

Початок вирішення проблеми за допомогою методів машинного навчання починається з визначення завдання, збору набору даних та навчання моделі. Модель складається з архітектури та параметрів. Параметри для заданої архітектури визначають, наскільки точно модель виконує поставлене завдання. Тому на практиці перед кожним інженером машинного навчання постає задача, яка полягає у знаходженні оптимальних значень цих параметрів. Для цього потрібно визначити функцію втрат, яка буде показувати наскільки добре працює модель. Мета оптимізації полягає у тому, щоб мінімізувати функцію втрат (похибку навчання) і знайти значення параметрів, які будуть відповідати цим втратам [37]. Часто у задачах оптимізації функцію втрат називають цільовою функцією. Зазвичай більшість алгоритмів стосуються мінімізації цільової функції, але якщо потрібно її максимізувати для цього є просте рішення: потрібно просто змінити знак цільової функції на протилежний.

Алгоритми оптимізації важливі для глибинного навчання. З одного боку, навчання складної моделі може тривати години, дні чи навіть тижні. Ефективність алгоритму оптимізації безпосередньо впливає на ефективність навчання моделі. З іншого боку, розуміння принципів різних алгоритмів оптимізації та ролі їх гіперпараметрів дозволить інженеру цілеспрямовано налаштовувати гіперпараметри для покращення продуктивності моделей глибинного навчання [12].

Майже усі проблеми оптимізації, що вирішуються під час навчання глибинних моделей, стосуються цільових функцій, які мають неопуклу просторову структуру з декількома локальними мінімумами та/або сідловими точками. Тим не менше, розробка та аналіз алгоритмів у контексті опуклих задач є дуже повчальними. У цьому параграфі буде детально розглянуто деякі оптимізаційні алгоритми, які використовуються на практиці для знаходження оптимальних параметрів нейронних мереж.

2.3.1 Особливість оптимізації

Похибка навчання та похибка узагальнення⁶ моделі фундаментально відрізняються між собою. Як було уже раніше зазначено, мета оптимізації полягає у тому, щоб мінімізувати похибку навчання, у той час як мета глибинного навчання полягає у мінімізації похибки узагальнення. Для того, щоб мінімізувати похибку узагальнення потрібно слідкувати за процесом навчання моделі та уникати випадки, коли модель ще є недонаученою або уже починає перенавчатись.

Для ілюстрації вищезгаданих відмінностей між похибками навчання та узагальнення моделі розглянемо емпіричний ризик та реальний ризик, рисунок 2.8. Під емпіричним ризиком будемо розуміти середню похибку моделі на навчальному наборі, тоді як реальний ризик – очікувана похибка моделі для всієї сукупності даних з якими може зіштовхнутись модель під час вирішення поставленого завдання.

Будемо вважати, що маємо обмежений навчальний набір даних, у результаті чого крива для емпіричного ризику є менш гладкою ніж для реального ризику [12]. На рисунку 2.8 показано, що глобальний мінімум емпіричного ризику на навчальному наборі даних може знаходитися в іншому місці, ніж мінімум реального ризику (похибка узагальнення).

⁶Похибка узагальнення – похибка моделі на даних, яких вона раніше не бачила.

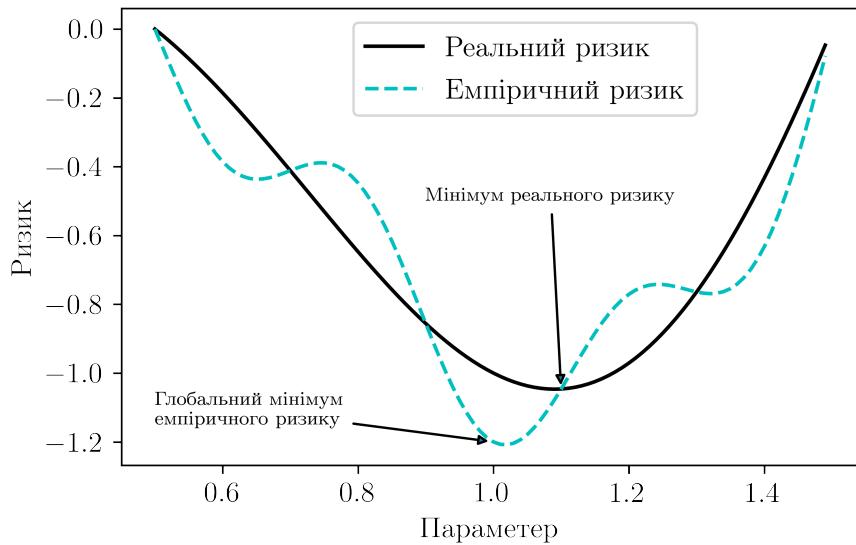


Рисунок 2.8 — Емпіричний та реальний ризик [12]

2.3.2 Виклики оптимізації

Більшість цільових функцій у задачах глибинного навчання є досить складними і часто не мають аналітичних розв'язків. Саме тому на практиці користуються числовими алгоритмами оптимізації. У процесі оптимізації може виникати багато проблем: локальні мініуми, сідлові точки тощо. Розглянемо детальніше деякі з них.

Локальний мінімум

Для будь-якої цільової функції $f(x)$, якщо значення $f(x)$ в точці x менше, ніж значення $f(x)$ в околі точки x , тоді $f(x)$ може бути локальним мініумом. Якщо значення $f(x)$ в точці x є мінімальним на усій області визначення, тоді $f(x)$ — глобальний мінімум.

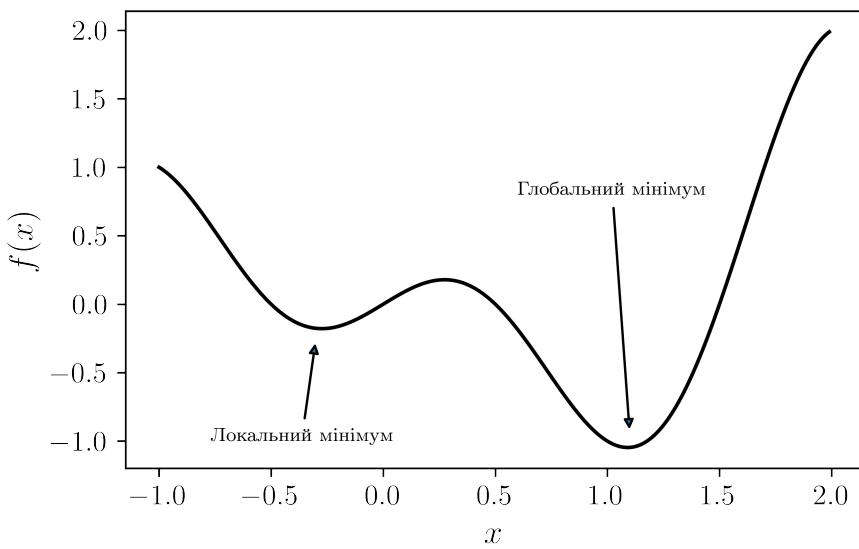


Рисунок 2.9 — Локальний та глобальний мінімум [12]

Зазвичай цільова функція моделей глибинного навчання має багато локальних мініумів. На кожній ітерації чисельного оптимізаційного алгоритму здійснюється мінімізація цільової функції лише локально, а не глобально. Таким чином розв'язок, отриманий на останній ітерації, може відповідати локальному мініму-

му, оскільки градієнт цільової функції в локальному та глобальному мінімуму наближається або стає нулем. Лише деякий ступінь шуму, внесений у процес навчання, може вивести розв'язок оптимізації з локального мінімуму. Фактично це є однією з корисних властивостей стохастичного та міні-пакетного градієнтного спуску, де зміна градієнтів від міні-пакету до міні-пакету здатна вивести параметри з локального мінімуму.

Сідлові точки

Ще однією причиною зникнення градієнтів крім локальних мінімумів є сідлові точки (точки перегину). Сідлова точка – це будь-яке місце, де усі градієнти функції зникають, але яке не є ні глобальним, ні локальним мінімумом. Розглянемо функцію $f(x) = x^3$, рисунок 2.10. Перша та друга похідна дорівнюють нулю у точці $x = 0$. На цьому етапі оптимізація може зупинитися, хоча це і не мінімум.

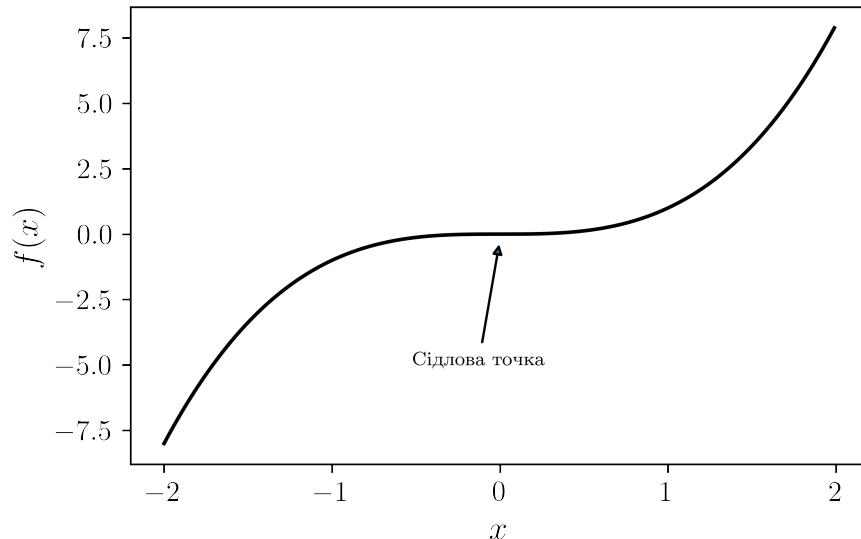


Рисунок 2.10 — Сідлова точка [12]

Як можна побачити з наступного прикладу, сідлові точки у багатовимірному просторі можуть бути ще більш підступнішими. Розглянемо функцію $f(x, y) = x^2 - y^2$, рисунок 2.11. Вона має сідлову точку в $(0, 0)$. Ця точка є максимумом відносно y та мінімумом відносно x . Більше того, ця функція схожа на сідло, тому ця математична властивість і отримала таку назву.

Ми припускаємо, що вхідні дані функції є k -мірним вектором, а її вихід є скаляром, тому гессіан функції матиме k власних значень. Розв'язком оптимізації може бути локальний мінімум, локальний максимум або сідлова точка у точці, де градієнт функції дорівнює нулю:

- Коли отримано усі позитивні власні значення матриці гессе для функції в точці, де градієнт дорівнює нулю, ми маємо локальний мінімум.
- Коли отримано усі негативні власні значення матриці гессе для функції в точці, де градієнт дорівнює нулю, ми маємо локальний максимум.
- Коли отримано позитивні та негативні власні значення матриці гессе для функції в точці, де градієнт дорівнює нулю, ми маємо сідлову точку.

Для задач з високою розмірністю ймовірність того, що принаймні деякі власні значення будуть від'ємні також є досить великою. Це робить сідлові точки більш ймовірними ніж локальні мінімуми. Деякі винятки для такої ситуації будуть розглянуті у наступному параграфі, коли буде вводитись опуклість.

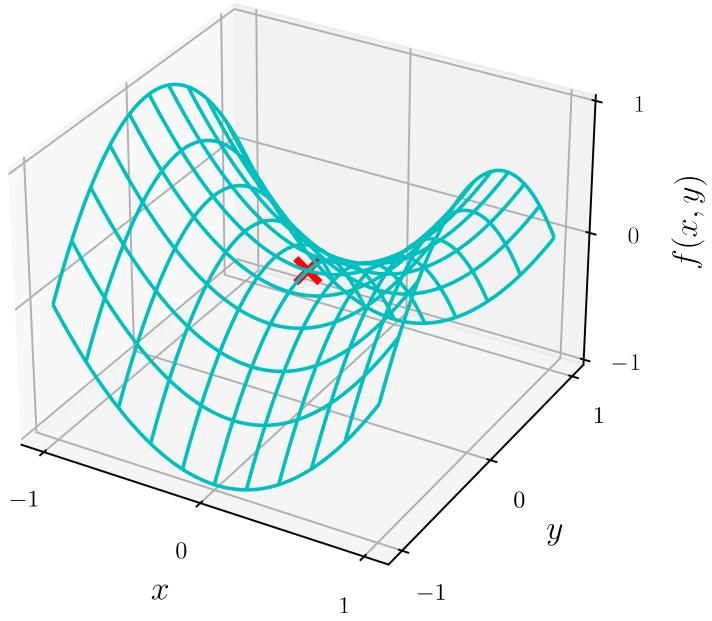


Рисунок 2.11 — Сідлова точка 2D [12]

2.3.3 Опуклість

Опукла функція – це така функція для якої власні значення гессіана є невід'ємними. На жаль, більшість проблем глибинного навчання не належать до цієї категорії, проте опуклість відіграє важливу роль у розробці алгоритмів оптимізації. Це багато в чому пов'язано з тим, що аналізувати та перевіряти алгоритми в такому контексті набагато легше. Іншими словами, якщо алгоритм працює погано навіть для простої просторової структури цільової функції (опуклої), сподіватися побачити кращі результати в іншому випадку, як правило, узагалі не вийде. Більше того, хоча задачі оптимізації глибинних нейронних мереж, як правило, мають складну просторову структуру (неопуклу), вони часто володіють деякими властивостями опуклих функцій поблизу локальних мінімумів. Це може призвести до появи нових варіантів оптимізації, наприклад [38].

Поняття опуклої множини лежить в основі визначення опуклої функції. Множина \mathcal{X} у векторному просторі є опуклою, якщо для будь-яких $a, b \in \mathcal{X}$, відрізок, що з'єднує a та b також знаходиться в \mathcal{X} . Математично це означає, що для усіх $\lambda \in [0, 1]$ маємо:

$$\lambda a + (1 - \lambda)b \in \mathcal{X} \quad (2.36)$$

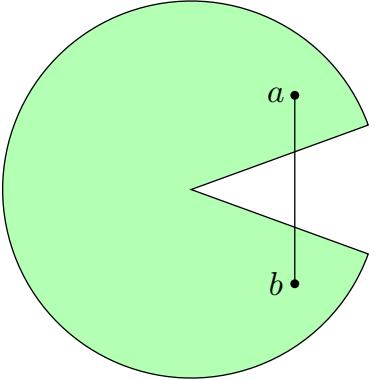
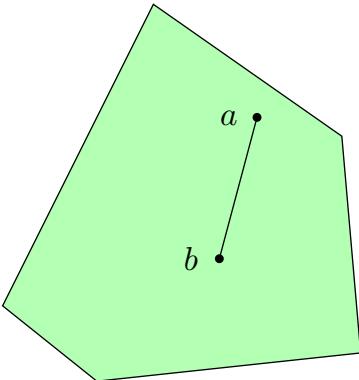
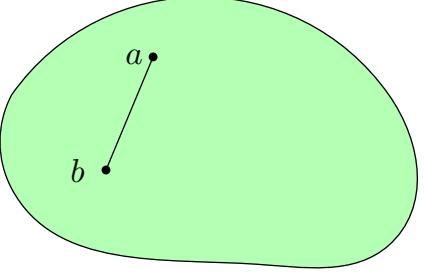
Розглянемо таблицю 2.1. Перша множина є неопуклою, оскільки існує відрізок, який не лежить у цій множині. Інші дві множини є опуклими, оскільки для будь-яких двох точок відрізок буде лежати у межах множини.

Перетин двох опуклих множин є опуклою множиною, проте об'єднання опуклих множин – необов'язково опукла множина [12].

Визначення опуклої функції можна дати на основі поняття опуклої множини. Нехай дано опуклу множину \mathcal{X} , тоді функція $f : \mathcal{X} \rightarrow \mathbb{R}$ є опуклою, якщо для будь-яких $a, b \in \mathcal{X}$ та для усіх $\lambda \in [0, 1]$ виконується:

$$\lambda f(a) + (1 - \lambda)f(b) \geq f(\lambda a + (1 - \lambda)b) \quad (2.37)$$

Табл. 2.1: Неопуклі та опуклі множини [12]

Неопукла множина	Опукла множина	Опукла множина
		

2.3.4 Властивості опуклих функцій

Опуклі функції мають багато корисних властивостей. Розглянемо деякі з цих властивостей.

Локальний мінімум – глобальний мінімум

Локальні мінімуми опуклих функцій є глобальними мінімумами. Доведемо це твердження від супротивного. Розглянемо опуклу функцію f визначену на опуклій множині \mathcal{X} . Нехай $x_0 \in \mathcal{X}$ – локальний мінімум. Тоді існує мале позитивне значення p , яке для $x \in \mathcal{X}$ задовільняє $0 < |x - x_0| \leq p$ і у результаті маємо $f(x_0) < f(x)$.

Припустимо, що локальний мінімум x_0 – це не глобальний мінімум функції f : тобто існує $x_g \in \mathcal{X}$ для якого $f(x_g) < f(x_0)$. Також існує $\lambda \in [0, 1)$ таке як $\lambda = 1 - \frac{p}{|x_0 - x_g|}$ для якого виконується $0 < |\lambda x_0 + (1 - \lambda)x_g - x_0| \leq p$. Проте, за визначенням опуклої функції, отримаємо:

$$\begin{aligned} f(\lambda x_0 + (1 - \lambda)x_g) &\leq \lambda f(x_0) + (1 - \lambda)f(x_g) < \\ &< \lambda f(x_0) + (1 - \lambda)f(x_0) = f(x_0) \end{aligned} \tag{2.38}$$

Нерівність 2.38 суперечить умові, що x_0 є локальним мінімумом ($f(x_0) < f(x)$). Таким чином, не існує такого $x_g \in \mathcal{X}$ для якого $f(x_g) < f(x_0)$. Локальний мінімум x_0 є також глобальним мінімумом.

Той факт, що локальний мінімум для опуклих функцій є також глобальним мінімумом, дуже зручний. Це означає, що у процесі мінімізації функції, ми не можемо “застрягти”. Однак, слід зауважити, що це не означає, що не може існувати більше одного глобального мінімуму або що такого глобального мінімуму може взагалі не існувати [12]. Наприклад, функція $f(x) = \max(|x| - 1, 0)$ досягає свого мінімуму в інтервалі $[-1, 1]$, тоді як функція $f(x) = \exp(x)$ не досягає мінімального значення на \mathbb{R} : тобто для $x \rightarrow -\infty$ функція асимптотично прямує до нуля, але не існує такого значення x для якого $f(x) = 0$.

Другі похідні та опуклість

Одновимірна функція $f : \mathbb{R} \rightarrow \mathbb{R}$ опукла тоді і тільки тоді, коли її друга похідна $f'' \geq 0$. Будь-яка двічі диференційована багатовимірна функція $f : \mathbb{R}^n \rightarrow \mathbb{R}$ є опуклою тоді і тільки, коли гессіан $\nabla^2 f \geq 0$ [12].

2.3.5 Пакетний градієнтний спуск

Пакетний градієнтний спуск рідко використовується на практиці у задачах глибинного навчання через повільну збіжність та надмірні обчислення, які потрібно виконати для здійснення одного кроку у напрямку

мінімізації цільової функції. Надмірність обчислень особливо помітна на великих навчальних наборах даних. Проте теоретичний аналіз ваг та коефіцієнтів збіжності пакетного градієнтного спуску є інтуїтивно зрозумілим, тому його розуміння є ключовим для аналізу інших методів оптимізації. Розмір пакету у пакетному градієнтному спуску дорівнює усьому навчальному набору. Іншими словами, якщо навчальний набір складається з n прикладів, спочатку потрібно подати на вход мережі n прикладів та знайти цільову функцію для цього пакету. Потім за допомогою зворотного поширення обчислити градієнти цільової функції відносно кожного параметра. Далі на основі знайдених градієнтів оновити параметри мережі. Графічна процедура для оновлення параметрів мережі представлена на рисунку 2.12.

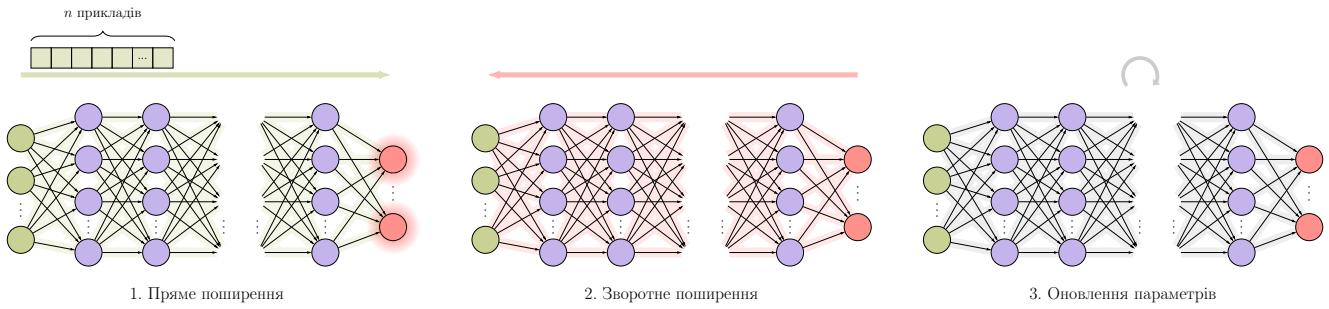


Рисунок 2.12 — Оновлення параметрів

Одновимірний пакетний градієнтний спуск

Одновимірний пакетний градієнтний спуск є чудовим прикладом для пояснення та демонстрації того, як цей алгоритм може мінімізувати цільову функцію. Розглянемо деяку неперервно диференційовану функцію $f : \mathbb{R} \rightarrow \mathbb{R}$. Розкладши у ряд Тейлора до першого порядку функцію $f(x + \varepsilon)$, отримаємо:

$$f(x + \varepsilon) = f(x) + \varepsilon f'(x) + \mathcal{O}(\varepsilon^2) \quad (2.39)$$

Рівняння (2.39) є наближенням першого порядку функції $f(x + \varepsilon)$, що задається значенням функції $f(x)$ та першою похідною $f'(x)$ у точці x . Для малого ε , що рухається у напрямку негативного градієнта, значення функції також зменшується. Для простоти оберемо фіксований розмір кроку $\alpha > 0$ та позначимо $\varepsilon = -\alpha f'(x)$. Підставляючи це у рівняння (2.39), отримаємо:

$$f(x - \alpha f'(x)) = f(x) - \alpha f'^2(x) + \mathcal{O}(\alpha^2 f'^2(x)) \quad (2.40)$$

До тих пір, поки похідна $f'(x) \neq 0$, значення функції $f(x - \alpha f'(x))$ буде зменшуватись, оскільки $\alpha f'^2(x) > 0$. Більш того, ми завжди можемо обрати α досить малим, щоб не враховувати вирази вищих порядків. Таким чином приходимо ненервінності:

$$f(x - \alpha f'(x)) \lesssim f(x) \quad (2.41)$$

Це означає, якщо ми будемо використовувати наступне правило для оновлення x , значення функції $f(x)$ може зменшитися:

$$x \leftarrow x - \alpha f'(x) \quad (2.42)$$

Для пакетного градієнтного спуску спочатку ми маємо обрати початкове значення x і константу $\alpha > 0$. Потім, коли ці значення були обрані, ми використовуємо їх для ітеративного оновлення параметра x . Ітеративне оновлення закінчується тоді, коли було досягнуто умову зупинки, наприклад, коли величина градієнта $|f'(x)|$ досить мала або кількість ітерацій досягла певного значення.

Для ілюстрації пакетного градієнтного спуску, розглянемо просту цільову функцію $f(x) = x^2$. Хоча ми знаємо, що $x = 0$ – це розв'язок для задачі мінімізації $f(x)$, використавши цю функцію ми подивимось як буде змінюватись x на кожному кроці під час мінімізації $f(x)$.

```

1 import numpy as np
2
3 def f(x): # Цільова функція
4     return x**2
5
6 def f_grad(x): # Градієнт (похідна) цільової функції
7     return 2 * x
8
9 def bgd(alpha, f_grad):
10    x = 6.0          # Початкове значення x
11    results = [x]
12    epoch = 8        # Максимальне число ітерацій
13    for i in range(epoch):
14        x -= alpha * f_grad(x)
15        results.append(float("%.6f" % x))
16    print(f'ітерація {epoch}, x: {x:.6f}')
17    return results
18
19 results = bgd(0.25, f_grad)
20 print(results)

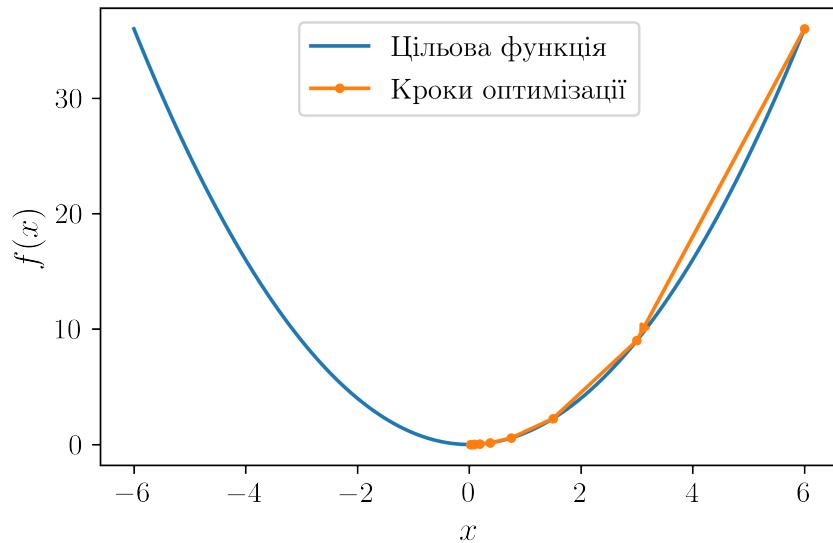
```

```

ітерація 8, x: 0.023438
[6.0, 3.0, 1.5, 0.75, 0.375, 0.1875, 0.09375, 0.046875, 0.023438]

```

Візуалізація цільової функції $f(x) = x^2$ та значення параметрів x на кожному кроці одновимірного пакетного градієнтного спуску показано на рисунку 2.13. Як можна бачити з цього рисунку, для фіксованого кроку навчання $\alpha = 0.25$ та початкового параметра $x = 6$ за вісім кроків оптимізаційного алгоритму було отримано $x = 0.023438$, що є близьким значенням до $x = 0$.



Крок навчання (швидкість навчання) α задається безпосередньо користувачем або розробником алгоритму оптимізації. Якщо використовувати досить малий крок навчання – це приведе до повільного оновлення

параметра x і як наслідок потрібно буде зробити значно більше ітерацій для отримання кращого розв'язку. Щоб показати це графічно, розглянемо той самий процес оптимізації для $\alpha = 0.06$. Як можна бачити з рисунка 2.14, після 8 ітерацій ми все ще знаходимось далеко від оптимального розв'язку.

```
1 results = bgd(0.06, f_grad)
```

```
ітерація 8, x: 2.157807
```

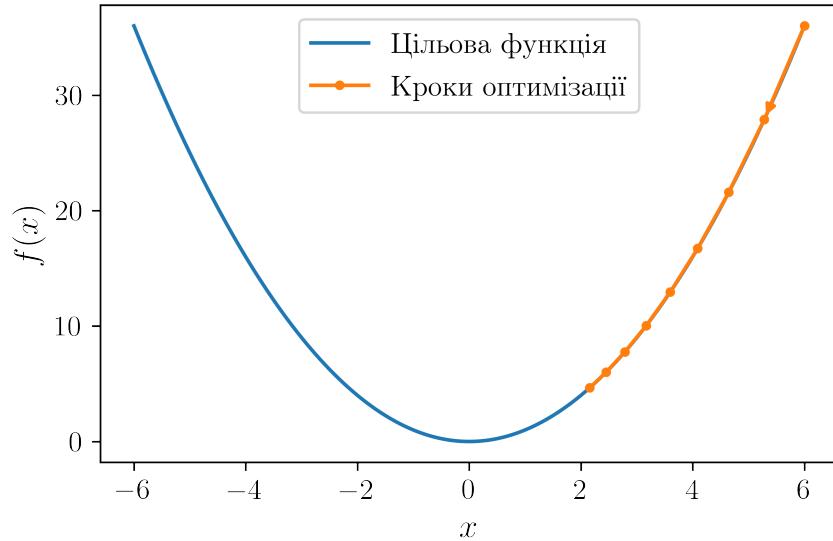


Рисунок 2.14 — Одновимірний пакетний градієнтний спуск ($\alpha = 0.06$) [12]

Якщо ж використовувати досить великий крок навчання, вираз $|\alpha f'(x)|$ може бути досить великим у ряді Тейлора першого порядку. Тобто, $\mathcal{O}(\alpha^2 f'^2(x))$ у (2.40) може стати суттєвим. У цьому випадку немає гарантії того, що ітеративне оновлення x буде мінімізувати значення $f(x)$. Наприклад, якщо установити крок навчання $\alpha = 1.1$, ітеративне оновлення параметра x буде розходитись, тобто віддалятись від оптимального розв'язку $x = 0$. Ілюстрація цього випадку показана на рисунку 2.15.

```
1 results = bgd(1.1, f_grad)
```

```
ітерація 8, x: 25.798902
```

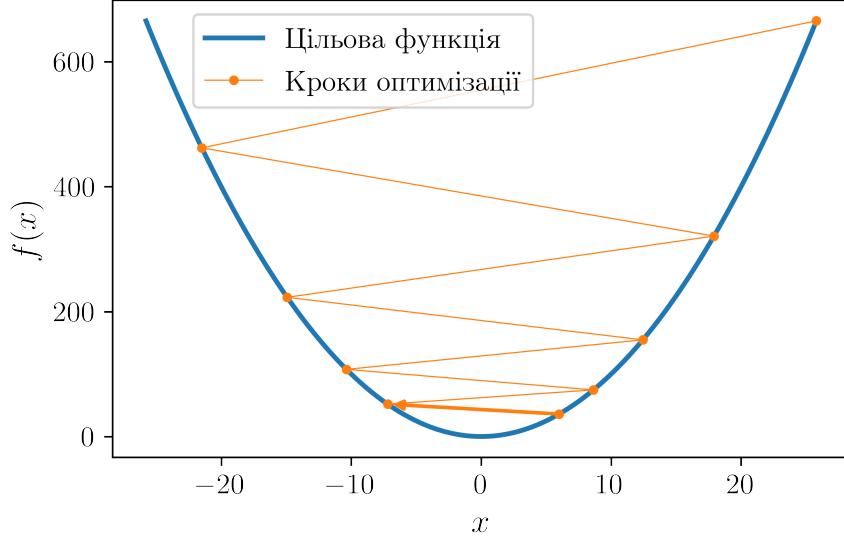


Рисунок 2.15 — Одновимірний пакетний градієнтний спуск ($\alpha = 1.1$) [12]

Багатовимірний пакетний градієнтний спуск

Тепер, коли ми маємо краще розуміння того як працює одновимірний пакетний градієнтний спуск, розглянемо ситуацію, де $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$. Тобто, цільова функція є відображенням з d -вимірного в одновимірний простір дійсних чисел $f : \mathbb{R}^d \rightarrow \mathbb{R}$. Тому градієнт цієї функції теж є багатовимірним і представляє собою вектор, який складається з d часткових похідних:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right]^\top \quad (2.43)$$

Кожна часткова похідна градієнта $\frac{\partial f(\mathbf{x})}{\partial x_i}$ показує швидкість зміни $f(\mathbf{x})$ залежно від вхідного значення x_i . Як і для одновимірного випадку, використаємо розклад багатовимірної функції у ряд Тейлора:

$$f(\mathbf{x} + \varepsilon) = f(\mathbf{x}) + \varepsilon^\top \nabla f(\mathbf{x}) + \mathcal{O}(|\varepsilon|^2) \quad (2.44)$$

Найкрутіший спуск задається від'ємним градієнтом, тому позначимо $\varepsilon = -\alpha \nabla f(\mathbf{x})$. Будемо вважати крок навчання α досить малим, щоб не враховувати вирази вищих порядків:

$$f(\mathbf{x} - \alpha \nabla f(\mathbf{x})) \lesssim f(\mathbf{x}) \quad (2.45)$$

Нерівність (2.45) означає, якщо ми будемо використовувати наступне правило для оновлення \mathbf{x} , значення функції $f(\mathbf{x})$ може зменшитися:

$$\mathbf{x} \leftarrow \mathbf{x} - \alpha \nabla f(\mathbf{x}) \quad (2.46)$$

Графічне представлення багатовимірного пакетного градієнтного спуску показано на рисунку 2.16.

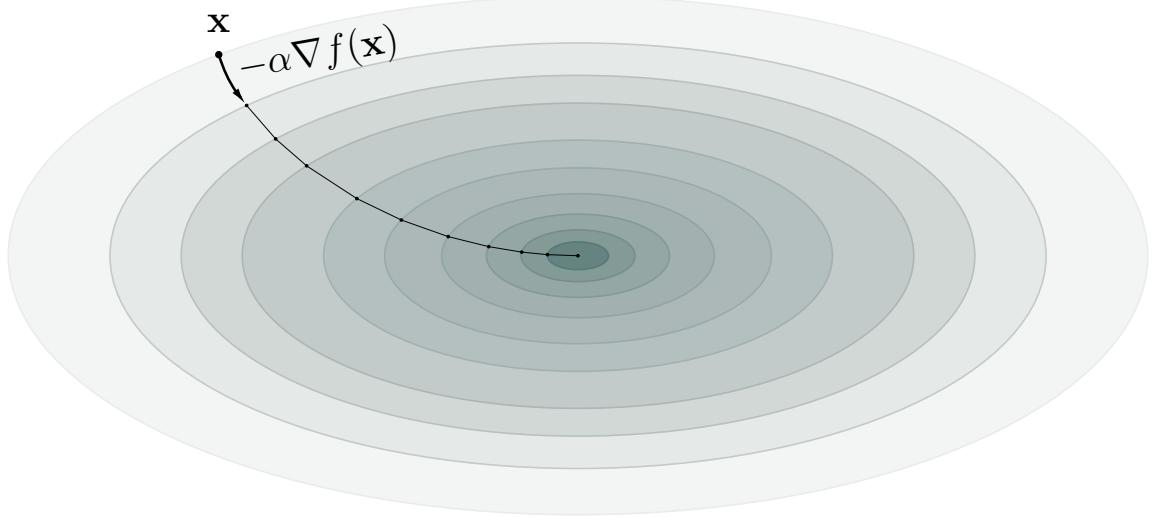


Рисунок 2.16 — Багатовимірний пакетний градієнтний спуск

2.3.6 Стохастичний градієнтний спуск

У глибинному навчанні зазвичай цільовою функцією є усереднене значення втрат для кожного прикладу з навчального набору даних. Тобто, якщо навчальний набір складається з n прикладів, тоді $f_j(\mathbf{x})$ – функція втрат для j -го прикладу з навчального набору, а \mathbf{x} – вектор параметрів. Таким чином, цільова функція прийме наступний вигляд:

$$f(\mathbf{x}) = \frac{1}{n} \sum_{j=1}^n f_j(\mathbf{x}) \quad (2.47)$$

Градієнт цільової функції 2.47 відносно \mathbf{x} обчислюється наступним чином:

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{j=1}^n \nabla f_j(\mathbf{x}) \quad (2.48)$$

Як можна побачити з рівняння 2.48, обчислювальні витрати для кожної незалежної ітерації пакетного градієнтного спуску становлять $\mathcal{O}(n)$, тобто лінійно зростають з n . Іншими словами, чим більший буде навчальний набір даних, тим більшими будуть обчислювальні витрати для кожної ітерації пакетного градієнтного спуску.

Стохастичний градієнтний спуск (SGD) дозволяє зменшити обчислювальні витрати на кожній ітерації. Це досягається за рахунок того, що на кожному кроці SGD рівномірно (з однаковою ймовірністю) обирається один приклад з індексом $j \in \{1, 2, \dots, n\}$ серед переміщеного набору даних та обчислюється для нього градієнт $\nabla f_j(\mathbf{x})$ для оновлення \mathbf{x} :

$$\mathbf{x} \leftarrow \mathbf{x} - \alpha \nabla f_j(\mathbf{x}), \quad (2.49)$$

де α – крок навчання. Можна побачити, що обчислювальні витрати для кожної ітерації зменшились з $\mathcal{O}(n)$ для пакетного градієнтного спуску до константи $\mathcal{O}(1)$ для стохастичного градієнтного спуску. Слід зазначити, що усереднене значення стохастичного градієнту $\nabla f_j(\mathbf{x})$ є хорошим наближенням повного градієнта $\nabla f(\mathbf{x})$:

$$\mathbb{E}_j \nabla f_j(\mathbf{x}) = \frac{1}{n} \sum_{j=1}^n \nabla f_j(\mathbf{x}) = \nabla f(\mathbf{x}) \quad (2.50)$$

Графічне представлення багатовимірного стохастичного градієнтного спуску показано на рисунку 2.17.

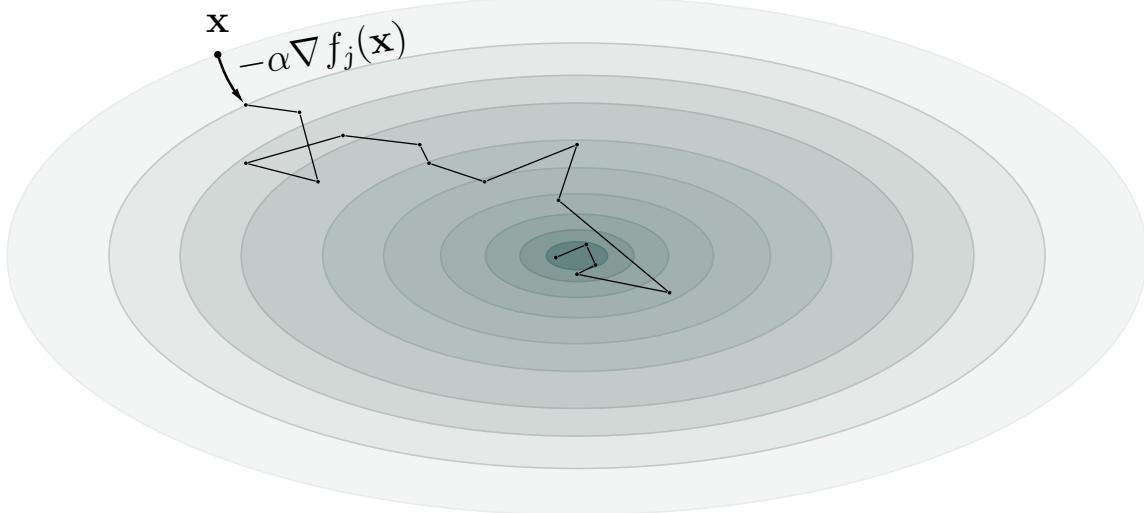


Рисунок 2.17 — Багатовимірний стохастичний градієнтний спуск

Як можна бачити з цього рисунку, траєкторія оновлення параметрів стохастичного градієнтного спуску нагадує випадкове блукання у той час як для пакетного градієнтного спуску траєкторія спрямована вниз за локальним нахилом ландшафту цільової функції. Через стохастичний характер градієнта розв'язок стохастичного градієнтного спуску буде осцилювати навколо глобального мінімуму і не буде покращуватись зі збільшенням кількості кроків. Для вирішення цієї проблеми потрібно керувати кроком навчання α , проте, якщо крок навчання обрати досить малим оновлення параметрів буде незначним, а оптимізація повільною, особливо, якщо кривина цільової функції мала. З іншого боку, якщо обрати крок навчання досить великим, оновлення параметрів буде досить великим, а розв'язок оптимізації, ймовірно, буде розходитись, особливо, якщо кривина цільової функції велика. Єдиний спосіб вирішити ці дві проблеми – динамічно змінювати крок навчання у ході оптимізації.

У роботі [39] подано довідковий матеріал та корисні рекомендації щодо використання стохастичного градієнтного спуску, а також пояснено, чому SGD є хорошим алгоритмом навчання для великих навчальних наборів даних.

2.3.7 Міні-пакетний градієнтний спуск

Розмір міні-пакету – це кількість прикладів, які використовуються для навчання моделі на кожній ітерації. Як правило, розмір міні-пакету обирають пропорційно степені двійки 2^k : 16, 32, 64, 128, ... Вибір правильного розміру міні-пакету є важливим для забезпечення збіжності цільової функції та значень параметрів, а також для узагальнення моделі. На практиці зазвичай для вибору міні-пакету використовують пошук по гратці. У деяких дослідженнях, наприклад [40, 41], розглядається, як обрати розмір міні-пакету, але загального консенсусу як це робити правильно для різних задач наразі немає.

Нехай на кожній ітерації оптимізаційного алгоритму міні-пакет буде складатись з b прикладів. Тоді цільова функція для міні-пакету буде визначатись наступним чином:

$$g(\mathbf{x}) = \frac{1}{b} \sum_{j=1}^b g_j(\mathbf{x}) \quad (2.51)$$

Градієнт цільової функції 2.51 відносно \mathbf{x} обчислюється наступним чином:

$$\nabla g(\mathbf{x}) = \frac{1}{n} \sum_{j=1}^n \nabla g_j(\mathbf{x}) \quad (2.52)$$

Оновлення вектора параметрів \mathbf{x} на міні-пакеті здійснюється аналогічно до (2.46) та (2.49), тільки береться своє значення градієнта, яке було отримано на цьому міні-пакеті:

$$\mathbf{x} \leftarrow \mathbf{x} - \alpha \nabla g(\mathbf{x}) \quad (2.53)$$

Графічне представлення багатовимірного міні-пакетного градієнтного спуску показано на рисунку 2.18. Як можна бачити з цього рисунку, траєкторія оновлення параметрів має менший шум порівняно з стохастичним градієнтним спуском. Це означає, що на кожній ітерації міні-пакетного градієнтного спуску здійснюється точніше обчислення градієнта відносно параметрів у напрямку локального мінімума цільової функції.

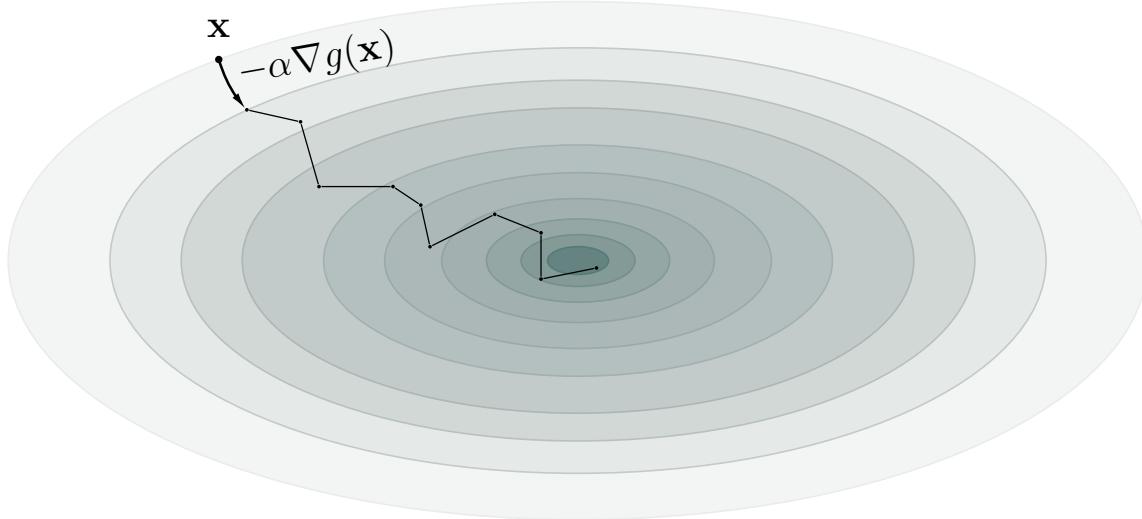


Рисунок 2.18 – Багатовимірний міні-пакетний градієнтний спуск

Збільшення розміру міні-пакету до тих пір, поки він поміщається в пам'яті графічного процесора, як правило, дозволяє покращити ефективність розпаралелювання та прискорити навчання моделей.

2.3.8 Імпульс

Більшість цільових функцій у задачах глибинного навчання, як уже раніше зазначалось, мають складну просторову структуру. Крім локальних мінімумів та сідлових точок можуть також зустрічатись патологічні викривлення, простіше кажучи області цільової функції, які неправильнно масштабовані. Зазвичай ландшафт таких областей має вигляд долини, траншеї, каналу або яру. Приклад такого ландшафту показано на рисунку 2.19. Значення цільової функції на кожній ітерації оптимізаційного алгоритму або стрибає між краями долини, або наближається до оптимуму досить повільно. Прогрес у заданому напрямку може взагалі зупинитись. Пакетний, стохастичний та міні-пакетний градієнтний спуск поводять себе досить погано у таких областях.

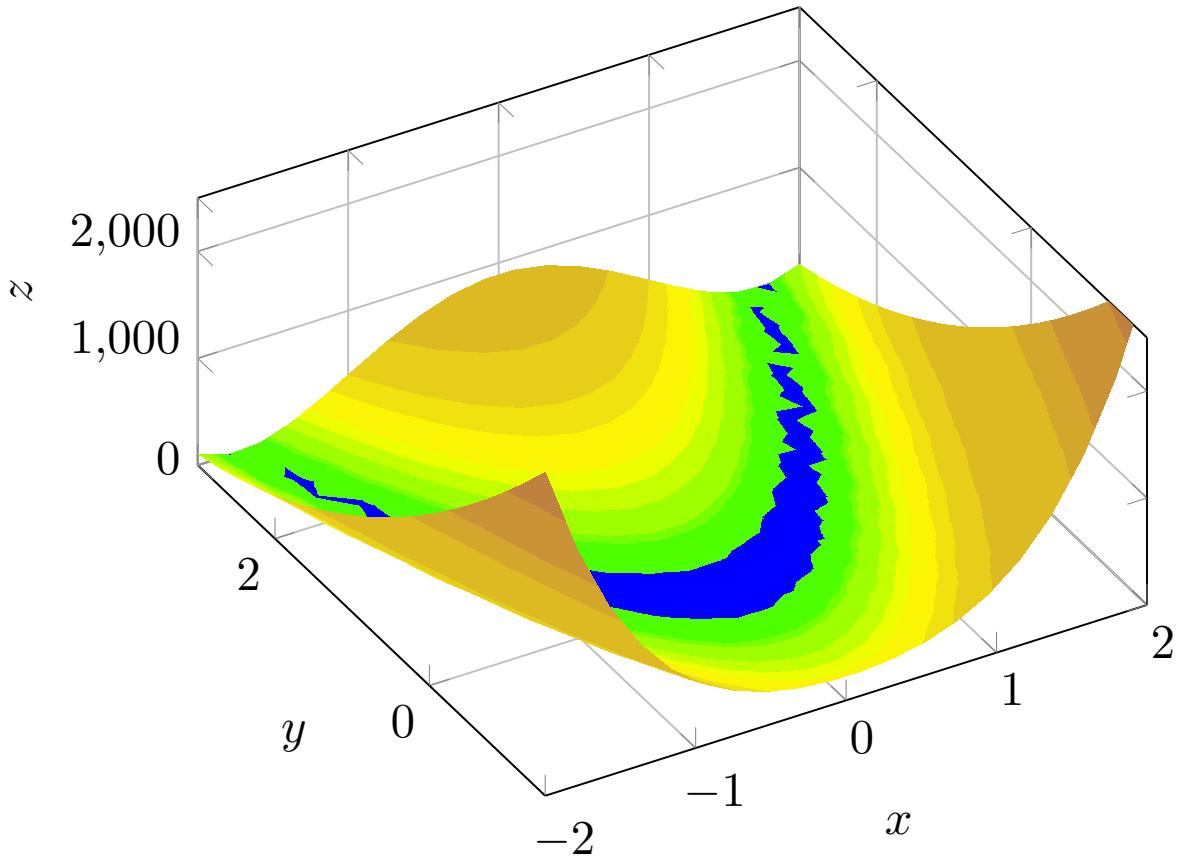


Рисунок 2.19 — Цільова функція з ландшафтом долини

Поняття імпульсу у задачах оптимізації параметрів нейронних мереж можна розглядати з фізичної точки зору, це буде пояснено трохи пізніше. Імпульс надає оптимізаційному алгоритму короткочасну пам'ять про параметри мережі і дозволяє суттєво збільшити швидкість збіжності алгоритму [42]. Іншими словами, він дозволяє врахувати параметри мережі з попередньої ітерації.

Розглянемо міні-пакетний градієнтний спуск (рисунок 2.18). Було б непогано, якби ми могли отримати вигоду від ефекту зменшення дисперсії навіть за межами усереднення градієнтів на міні-пакеті. Один із способів для зменшення дисперсії є заміна обчислення градієнта на експоненційно ковзаюче середнє (EKC):

$$\mathbf{p}^{(t+1)} \leftarrow \beta \mathbf{p}^{(t)} + (1 - \beta) \nabla g(\mathbf{x}^{(t)}) \quad (2.54)$$

або спрощений варіант EKC:

$$\mathbf{p}^{(t+1)} \leftarrow \beta \mathbf{p}^{(t)} + \nabla g(\mathbf{x}^{(t)}), \quad (2.55)$$

де t – номер ітерації, $\beta \in (0, 1)$. Тепер оновлення параметрів буде здійснюватись з урахуванням усереднення градієнтів, отриманих за (2.54) або (2.55):

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \alpha \mathbf{p}^{(t+1)} \quad (2.56)$$

Підставивши у (2.56) вираз для імпульсу (2.55), отримаємо:

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \alpha \nabla g(\mathbf{x}^{(t)}) + \beta (\mathbf{x}^{(t)} - \mathbf{x}^{(t-1)}) \underset{\text{імпульс}}{\text{імпульс}} \quad (2.57)$$

Параметр \mathbf{p} прийнято називати імпульсом, а β – коефіцієнтом імпульсу. Імпульс акумулює градієнти з передніх ітерацій. Фактично він представляє собою суму миттєвого градієнта та усередненого градієнта за декілька минулих ітерацій. Щоб переконатись у цьому, розширимо запис $\mathbf{p}^{(t+1)}$ з (2.55) рекурсивно:

$$\mathbf{p}^{(t+1)} \leftarrow \beta^2 \mathbf{p}^{(t-1)} + \beta \nabla g(\mathbf{x}^{(t-1)}) + \nabla g(\mathbf{x}^{(t)}) = \sum_{\tau=0}^t \beta^\tau \nabla g(\mathbf{x}^{(t-\tau)}) \quad (2.58)$$

Таким чином, велике значення β відповідає усередненню для великого діапазону значень градієнтів, тоді як мале β має незначний ефект на корекцію градієнта. Тобто, коли $\beta = 0$, ми отримуємо звичайний міні-пакетний градієнтний спуск. Але для $\beta = 0.9$ або $\beta = 0.99$ імпульс дозволяє прискорити збіжність оптимізаційного алгоритму і майже завжди працює добре для цих значень гіперпараметра. Іноді для отримання незначної додаткової вигоди можна цей гіперпараметр налаштувати.

Вищезазначені міркування лягли в основу того, що сьогодні називають прискореними методами градієнта. Такі методи мають додаткову перевагу: вони набагато ефективніше працюють для неправильно масштабованої просторової структури цільової функції⁷. Тобто там, де є напрямки для яких прогрес оптимізаційного алгоритму є набагато повільніший, ніж в інших.

Імпульс завдяки своїй ефективності є досить добре вивченою темою в оптимізації параметрів глибинних мереж та не тільки⁸. Цей підхід для задач лінійної алгебри запропонував у роботі [43] Стенлі Франкель (Stanley Frankel), а Борис Поляк у роботі [44] детально описав теоретичні аспекти збіжності методів з імпульсом. Нестеров у роботі [45] запропонував модифікований варіант для обчислення імпульсу, шляхом зміни точки, де буде здійснюватись обчислення миттєвого градієнта. Цей підхід називають прискореним градієнтом Нестерова або просто імпульсом Нестерова:

$$\mathbf{p}^{(t+1)} \leftarrow \beta \mathbf{p}^{(t)} + \nabla g(\mathbf{x}^{(t)}) + \beta \mathbf{p}^{(t)} \quad (2.59a)$$

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \alpha \mathbf{p}^{(t+1)} \quad (2.59b)$$

Більш детальний огляд імпульсу Нестерова можна знайти у PhD дисертації Іллі Суцкевера [46].

Фізична природа імпульсу розглядається у роботі [47]. Нехай частинка масою m рухається у в'язкому середовищі з коефіцієнтом тертя μ під дією консервативних сил з потенціальною енергією $g(\mathbf{x})$:

$$m \frac{d^2 \mathbf{x}}{dt^2} = -\nabla g(\mathbf{x}) - \mu \frac{d\mathbf{x}}{dt}, \quad (2.60)$$

де \mathbf{x} – вектор координат частинки. Дискретизувавши рівняння (2.60), отримаємо:

$$m \frac{\mathbf{x}^{(t+\Delta t)} - 2\mathbf{x}^{(t)} + \mathbf{x}^{(t-\Delta t)}}{\Delta t^2} + \mu \frac{\mathbf{x}^{(t+\Delta t)} - \mathbf{x}^{(t)}}{\Delta t} = -\nabla g(\mathbf{x}) \quad (2.61)$$

Після приведення до спільного знаменника та групування доданків, перейдемо до такого виду:

$$(m + \mu \Delta t) \frac{\mathbf{x}^{(t+\Delta t)} - \mathbf{x}^{(t)}}{\Delta t^2} + m \frac{\mathbf{x}^{(t-\Delta t)} - \mathbf{x}^{(t)}}{\Delta t^2} = -\nabla g(\mathbf{x}) \quad (2.62)$$

Помноживши ліву та праву частину рівняння (2.62) на $\frac{\Delta t^2}{m + \mu \Delta t}$ та здійснивши перестановку доданків, отрима-

⁷Ландшафт цільової функції має форму долини, траншеї, каналу або яру.

⁸Імпульс також широко використовується у лінійній алгебрі для прискорення збіжності ітеративних методів вирішення лінійних рівнянь.

ємо:

$$\mathbf{x}^{(t+\Delta t)} = \mathbf{x}^{(t)} - \frac{\Delta t^2}{(m + \mu\Delta t)} \nabla g(\mathbf{x}) + \frac{m}{(m + \mu\Delta t)} (\mathbf{x}^{(t)} - \mathbf{x}^{(t-\Delta t)}) \quad (2.63)$$

Як можна помітити, рівняння (2.63) еквівалентне рівнянню (2.57). Крок навчання α та коефіцієнт імпульсу β відіграють роль коефіцієнта тертя та маси відповідно:

$$\alpha = \frac{\Delta t^2}{(m + \mu\Delta t)} \quad (2.64a)$$

$$\beta = \frac{m}{(m + \mu\Delta t)} \quad (2.64b)$$

Слід підкреслити, що у випадку, коли $\beta = 0$, маса частинки $m = 0$, відповідно до рівняння (2.64b). Таким чином коефіцієнт імпульсу відіграє роль маси в ітеративних алгоритмах оптимізації.

2.3.9 Адаптивний градієнт (AdaGrad)

Як було уже раніше зазначено, оптимізаційні алгоритми з імпульсом дозволяють прискорити збіжність і показують хороші результати для неправильного масштабованої структури цільової функції. Але це досягається за рахунок введення додаткового гіперпараметра (коефіцієнт імпульсу). З огляду на це, природно постає питання, чи існує інший спосіб, який би дозволив підтримувати процес навчання і унеможливив затухання для ділянок цільової функції з малою кривиною (у долинах, ярах, траншеях). Тобто, якщо ми потрапляємо у таку область, оновлення параметрів починає здійснюватись у напрямку оптимуму досить маленькими кроками, таким чином прогрес досить суттєво уповільнюється. Це наштовхує на думку, можливо є сенс динамічно управлюти кроком навчання на кожній ітерації. Алгоритм AdaGrad індивідуально адаптує крок навчання на кожній ітерації завдяки масштабуванню. Масштабування здійснюється шляхом ділення глобального кроku навчання на квадратний корінь від суми усіх попередніх квадратних значень градієнта. AdaGrad було запропоновано у роботі [48].

Для прикладу розглянемо міні-пакетний градієнтний спуск з градієнтом цільової функції, який визначається рівнянням (2.52). Тоді загальна процедура для оновлення параметрів мережі за допомогою алгоритму AdaGrad буде мати наступний вигляд:

$$\mathbf{r}^{(t+1)} \leftarrow \mathbf{r}^{(t)} + \nabla g(\mathbf{x}^{(t)}) \odot \nabla g(\mathbf{x}^{(t)}) \quad (2.65a)$$

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \frac{\alpha}{\sqrt{\mathbf{r}^{(t+1)}} + \varepsilon} \odot \nabla g(\mathbf{x}^{(t)}), \quad (2.65b)$$

де ε – мала константа, зазвичай порядку 10^{-7} , яка унеможливлює ділення на нуль. Вираз (2.65a) акумулює квадрат градієнта. Чим більше буде значення градієнта цільової функції, тим швидшим буде зменшення кроку навчання, тоді як малі значення градієнта будуть мати незначний уплів на зменшення кроку навчання. В контексті опуклої оптимізації алгоритм AdaGrad має деякі бажані теоретичні властивості. Однак емпірично для навчання глибинних нейронних мереж накопичення квадратичних градієнтів з початку навчання може привести до передчасного та надмірного зниження кроку навчання. AdaGrad добре працює для деяких, але не для всіх моделей глибинного навчання [49, Chapter 8, pp. 302-307].

2.3.10 Середнє квадратичне поширення (RMSProp)

Алгоритм середнього квадратичного поширення (Root Mean Square Propagation, RMSProp) є методом оптимізації на основі градієнта, що використовується для навчання нейронних мереж. Цей підхід запропонований Джейфрі Гіntonом (Geoffrey Hinton) [50]. Його ідея полягає у модифікації алгоритму AdaGrad для кращої

роботи на неопуклих задачах, замінивши накопичення градієнта на експоненційно ковзаюче середнє. Проблема AdaGrad полягає у тому, що цей алгоритм зменшує крок навчання відповідно до усієї історії квадрата градієнта і може зробити крок навчання занадто малим ще до моменту підходу до області глобального мінімуму. Для задач з складною структурою цільової функції таке затухання кроку навчання зазвичай має негативний ефект. RMSProp використовує експоненційно ковзаюче середнє, щоб відкинути градієнти з крайнього минулого для кращої збіжності алгоритму, коли ми підходимо до області оптимуму. Процедура для оновлення параметрів у RMSProp має наступний вид:

$$\mathbf{r}^{(t+1)} \leftarrow \rho \mathbf{r}^{(t)} + (1 - \rho) \nabla g(\mathbf{x}^{(t)}) \odot \nabla g(\mathbf{x}^{(t)}) \quad (2.66a)$$

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \frac{\alpha}{\sqrt{\mathbf{r}^{(t+1)} + \varepsilon}} \odot \nabla g(\mathbf{x}^{(t)}) \quad (2.66b)$$

Порівняно з AdaGrad, використання експоненційно ковзаючого середнього вводить новий гіперпараметр (ρ), який контролює шкалу довжини ковзаючої середньої. Емпірично доведено, що RMSProp є ефективним та практичним алгоритмом оптимізації глибинних нейронних мереж [49, Chapter 8, pp. 302-307].

2.3.11 Адаптивна оцінка моментів (Adam)

Алгоритм оптимізації, який поєднує усі переваги імпульсу та RMSProp, називають адаптивною оцінкою моментів (Adaptive Moments Estimation, Adam) [51]. Adam став досить популярним і зарекомендував себе як одного із найбільш надійних та ефективних алгоритмів оптимізації для глибинного навчання. Однак цей алгоритм також має свої недоліки. Зокрема у роботі [52] показано, що бувають ситуації, коли Adam може розходитись через поганий контроль дисперсії. У іншій роботі [53] запропоновано корекцію алгоритма Adam, яка називається Yogi і направлена на виправлення цієї проблеми.

Однією з ключових складових алгоритму Adam є те, що він використовує експоненційно ковзаюче середнє для отримання оцінок як моменту першого порядку (вираз для імпульсу), так і моменту другого порядку (накопичення квадрата градієнтів). Тобто, змінні стану визначаються наступним чином:

$$\underset{\text{момент 1-го порядку}}{\mathbf{p}^{(t+1)}} \leftarrow \beta \mathbf{p}^{(t)} + (1 - \beta) \nabla g(\mathbf{x}^{(t)}) \quad (2.67a)$$

$$\underset{\text{момент 2-го порядку}}{\mathbf{r}^{(t+1)}} \leftarrow \rho \mathbf{r}^{(t)} + (1 - \rho) \nabla g(\mathbf{x}^{(t)}) \odot \nabla g(\mathbf{x}^{(t)}) \quad (2.67b)$$

Крім простого поєднання імпульсу та RMSProp, Adam включає нормалізацію (масштабування) моментів першого та другого порядку, щоб врахувати їх ініціалізацію на початку. Іншими словами, якщо ми ініціалізуємо $\mathbf{p}^{(0)} = \mathbf{r}^{(0)} = 0$, ми отримаємо значне зміщення на початку до менших значень. Цю проблему можна вирішити, використавши той факт, що $\sum_{\tau=0}^t \beta^\tau = \frac{1-\beta^{t+1}}{1-\beta}$. Таким чином нормалізовані змінні стану визначаються так:

$$\hat{\mathbf{p}}^{(t+1)} \leftarrow \frac{\mathbf{p}^{(t+1)}}{1 - \beta^{t+1}} \quad (2.68a)$$

$$\hat{\mathbf{r}}^{(t+1)} \leftarrow \frac{\mathbf{r}^{(t+1)}}{1 - \rho^{t+1}} \quad (2.68b)$$

Оновлення параметрів з урахуванням вищезазначених оцінок буде здійснюватись за цим правилом:

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \frac{\alpha}{\sqrt{\hat{\mathbf{r}}^{(t+1)} + \varepsilon}} \odot \hat{\mathbf{p}}^{(t+1)} \quad (2.69)$$

На відміну від RMSProp, це оновлення використовує імпульс $\hat{\mathbf{p}}^{(t+1)}$, а не сам градієнт. Більше того, є ще одна невелика косметична різниця, оскільки масштабування кроку навчання відбувається за допомогою $\frac{1}{\sqrt{\hat{\mathbf{r}}^{(t+1)} + \varepsilon}}$ замість $\frac{1}{\sqrt{\hat{\mathbf{r}}^{(t+1)} + \varepsilon}}$. Перший варіант зазвичай на практиці працює трохи краще. Константу ε обирають порядку 10^{-7} .

2.4 Деякі методи регуляризації нейронних мереж

Регуляризація нейронних мереж використовується, щоб запобігти перенавчанню моделі на тренувальному наборі даних та отримати краще узагальнення на даних, яких модель раніше не бачила (тестовий набір даних). Іншими словами, регуляризація – це будь-які зміни, які вносяться в алгоритм навчання з метою зменшення втрат узагальнення. Здатність моделі узагальнювати залежить від двох факторів [54]:

- Розміру навчального (тренувального) набору даних.
- Складності нейронної мережі. Під складністю мається на увазі степінь поліному, за допомогою якого отримана модель описує певну проблему.

Перш ніж перейти до розгляду основних методів регуляризації потрібно розділи набір даних на окремі частини. Це робиться для того, щоб можна було оцінити здатність отриманої моделі узагальнювати. Зазвичай розділяють набір даних на три частини [54]: навчальний (Train), перевірки (Val або Dev) та тестовий (Test) набори. Класичний підхід поділу та підхід поділу для ери великих даних показано на рисунку 2.20.

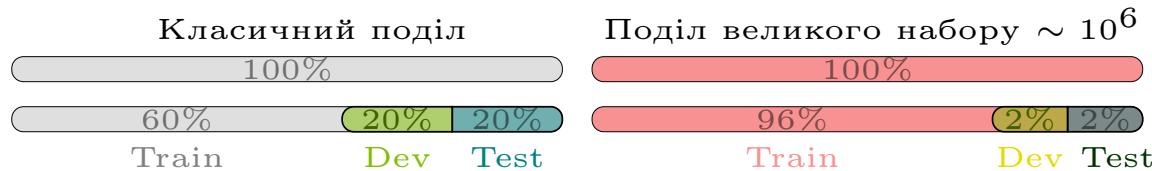


Рисунок 2.20 – Поділ даних на навчальний (Train), перевірки (Dev) та тестовий (Test) набори: класичний підхід та підхід для ери великих даних

На навчальному наборі навчають модель шукати приховані закономірності у даних, на наборі перевірки можна додатково налаштовувати гіперпараметри для кращої роботи моделі і на тестовому наборі перевіряють її спроможність успішно узагальнювати. Якщо модель запам'ятала так добре навчальний набір даних, що погано узагальнює на даних, яких раніше не бачила – говорять про перенавчання моделі. Якщо модель показує погані результати і на навчальному, і на перевірочному та тестовому наборах – говорять про те, що модель є недонавчененою. Якщо ж модель показує хороші результати на навчальному, перевірочному та тестовому наборах – говорять про правильний компроміс зсуву та розкиду. Для демонстрації цих трьох ситуацій розглянемо таблицю 2.2 у якій подано точність (accuracy) моделей на різних частинах набору.

Табл. 2.2: Точність трьох моделей на різних частинах набору [54]

Train Accuracy	Dev Accuracy	Test Accuracy	Примітка
68%	65%	63%	Недонавчання (високий зсув)
99%	70%	65%	Перенавчання (високий розкид)
95%	93%	92%	Доречний рівень навчання (належний баланс між зсувом та розкидом)

Регуляризація дозволяє скоротити розрив між продуктивністю моделі на тестовому наборі та її продуктивністю на перевірочному та навчальному наборах, зберігаючи при цьому продуктивність на навчальному наборі якомога більшою.

2.4.1 Рання зупинка (Early stopping)

Одним з популярних методів регуляризації є рання зупинка. Слід пам'ятати, що оптимізація параметрів нейронної мережі є ітеративним процесом. Оцінюючи втрати моделі після кожної епохи на навчальному та перевірочному наборах даних, можна спостерігати криві, які будуть схожі до кривих з рисунку 2.21.

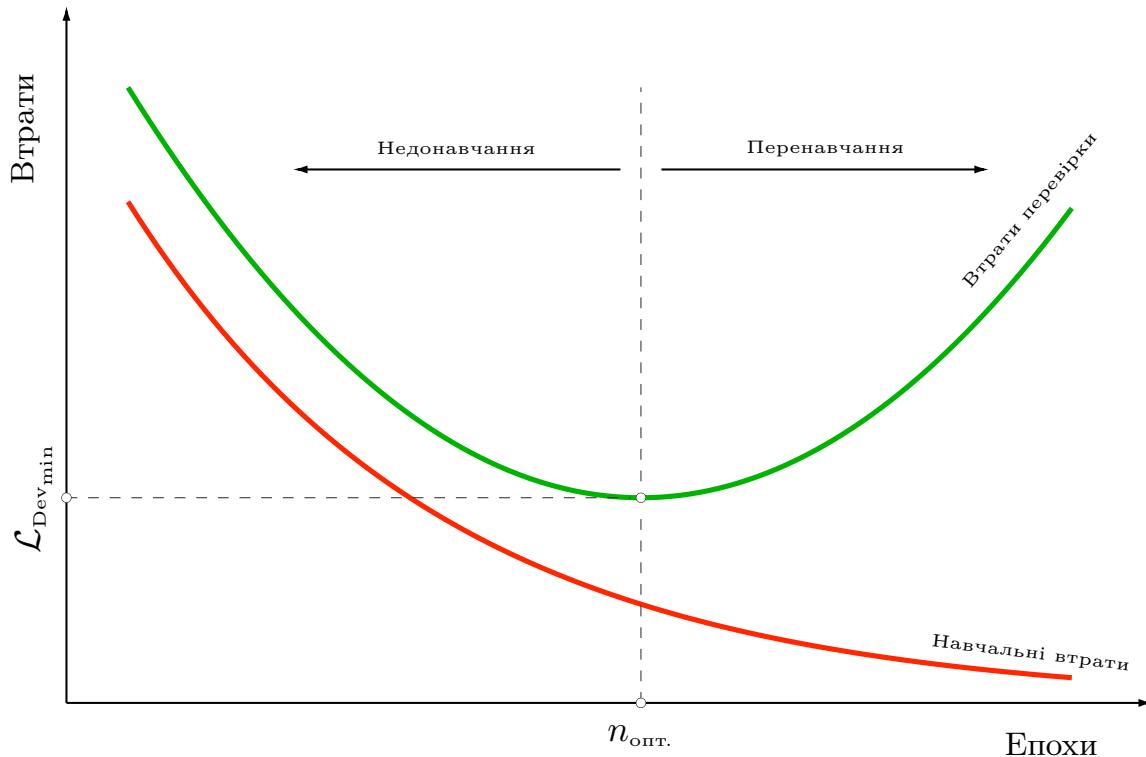


Рисунок 2.21 — Баланс між навчальними втратами та втратами перевірки [54]

У результаті можна буде помітити, що з початку навчання і аж до певної кількості епох $n_{\text{опт.}}$, втрати моделі будуть зменшуватись як на навчальному, так і на перевірочному наборах. Далі зі збільшенням кількості епох навчальні втрати будуть продовжувати зменшуватись, в той час як втрати на перевірочному наборі почнуть зростати (з цього моменту починається перенавчання моделі). Рання зупинка дозволяє зберегти параметри моделі з тієї ітерації оптимізаційного алгоритму, коли втрати на перевірочному наборі були найменші. Тобто, якщо розглядати приклад з рисунку 2.21 – рання зупинка збереже параметри моделі з епохи $n_{\text{опт.}}$. Таким чином, збережена модель буде найефективнішою на перевірочному наборі та ймовірно буде краще узагальнювати на тестовому наборі. Псевдокод для реалізації ранньої зупинки можна записати так [49, Chapter 7, pp. 224-270]:

Нехай p – кількість епох, щоб спостерігати зменшення втрат моделі на перевірочному наборі даних, перш ніж припинити навчання.

Нехай \mathbf{W}_0 та b_0 – ініціалізовані параметри.

- 1: $\mathbf{W} \leftarrow \mathbf{W}_0$
- 2: $b \leftarrow b_0$
- 3: $n \leftarrow 0$
- 4: $j \leftarrow 0$
- 5: $\mathcal{L}_{\text{Dev}}^* \leftarrow \infty$
- 6: $\mathbf{W}^* \leftarrow \mathbf{W}$
- 7: $b^* \leftarrow b$
- 8: $n_{\text{опт.}} \leftarrow n$

```

9: while  $j < p$  do
10:   Оновити параметри мережі  $\mathbf{W}$  та  $b$  для поточної епохи
11:    $n \leftarrow n + 1$ 
12:    $\mathcal{L}_{\text{Dev}} \leftarrow \text{DevError}(\mathbf{W}, b)$ 
13:   if  $\mathcal{L}_{\text{Dev}} < \mathcal{L}^*_{\text{Dev}}$  then
14:      $j \leftarrow 0$ 
15:      $\mathbf{W}^* \leftarrow \mathbf{W}$ 
16:      $b^* \leftarrow b$ 
17:      $n_{\text{опт.}} \leftarrow n$ 
18:      $\mathcal{L}^*_{\text{Dev}} \leftarrow \mathcal{L}_{\text{Dev}}$ 
19:   else
20:      $j \leftarrow j + 1$ 
21:   end if
22: end while

```

Найкращими параметрами будуть \mathbf{W}^* та b^* , які отримані для епохи $n_{\text{опт.}}$. Алгоритм ранньої зупинки є досить простим у реалізації. Крім цього, такі фреймворки глибинного навчання як Tensorflow або Pytorch уже містять готову реалізацію цього алгоритму. Слід зазначити, що рання зупинка є швидшою за інші методи регуляризації, оскільки немає ніякого гіперпараметра регуляризації, який потрібно додатково налаштовувати. Для порівняння, регуляризація L2 вимагає налаштування гіперпараметра регуляризації. Для цього може знадобитися кілька експериментів, щоб регуляризувати модель за допомогою L2, тоді як для ранньої зупинки потрібен лише один запуск.

2.4.2 L2 регуляризація

Теорема “безкоштовних сніданків не існує” означає, що не існує універсальної глобальної оптимізації, а єдиний шлях для однієї стратегії бути продуктивнішою за іншу – це бути спеціально налаштованою для розв’язання певного класу задач. Іншими словами, ми розробляємо алгоритми машинного навчання, які б добре виконували конкретне завдання. Це досягається шляхом вбудовування набору переваг у алгоритм навчання. Коли ці переваги узгоджуються з проблемою, яку ми просимо алгоритм вирішити, він працює краще. Одним із способів для такої модифікації навчального алгоритму є збільшення або зменшення репрезентативної спроможності моделі, шляхом додавання або видалення функцій з простору гіпотез розв’язків, серед яких навчальний алгоритм має змогу обрати найкращий. Інший спосіб модифікації навчального алгоритму полягає у наданні переваги одному розв’язку над іншим у просторі гіпотез алгоритму. Це означає, що обидві функції є прийнятними, але одна з них є кращою. Небажаний розв’язок буде обрано, лише якщо він відповідає навчальним даним значно краще, ніж бажаний розв’язок [49, Chapter 5, pp. 116-117].

Як раніше було уже зазначено, одним із факторів від якого залежить здатність моделі узагальнювати є її складність. Тому, щоб запобігти перенавчанню моделі можна зменшити її складність. Це можна зробити двома способами:

1. Деактивувати окремі шари або нейрони мережі тим самим зменшивши кількість навчальних параметрів.
2. Обмежити зростання ваг за допомогою додавання норми параметрів до цільової функції: норму параметрів також прийнято називати виразом для зниження ваг (weight decay) [55].

Якщо б ми навчали нейронну мережу, скажімо за допомогою пакетного градієнтного спуску, мінімізувати цільову функцію:

$$J_{b. \text{cross-entropy}} = -\frac{1}{m} \sum_{i=1}^m \left[y \log \hat{y} + (1 - y) \log (1 - \hat{y}) \right] \quad (2.70)$$

Тоді б оновлення ваг здійснювалось за правилом:

$$w \leftarrow w - \alpha \frac{\partial J_{b. \text{cross-entropy}}}{\partial w} \quad (2.71)$$

L2 регуляризація полягає у додаванні L2 норми параметрів до цільової функції з метою обмеження зростання ваг:

$$J = J_{b. \text{cross-entropy}} + \lambda J_{L2}, \quad (2.72)$$

де λ – гіперпараметр регуляризації ($\lambda \in \mathbb{R}$), $J_{L2} = \|w\|_2^2 = \sum_k |w_k|^2$.

Таким чином, правило оновлення ваг з урахуванням L2 регуляризації прийме наступний вид:

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w} = w - \alpha \left[\frac{\partial J_{b. \text{cross-entropy}}}{\partial w} + \frac{\partial J_{L2}}{\partial w} \right] \quad (2.73a)$$

$$w \leftarrow w - 2\alpha\lambda w - \alpha \frac{\partial J_{b. \text{cross-entropy}}}{\partial w} \quad (2.73b)$$

На кожному кроці L2 регуляризації ваги зменшуються до дещо нижчого значення, оскільки загалом $2\alpha\lambda << 1$, що спричиняє зниження абсолютноного значення ваг.

Вплив гіперпараметра регуляризації λ на оновлення ваг мережі [54]:

- Дуже мале значення λ : відсутній явний ефект впливу.
- Належне значення λ : значення ваг зменшуються до нуля, нагадуючи центрований розподіл, який стає все більш і більш піковим протягом навчання.
- Занадто велике значення λ : усі ваги швидко стають близькими до нуля, таким чином модель буде явно не донавчатись, оскільки ваги будуть занадто обмеженими.

Як зниження ваг допомагає моделі краще узагальнювати?

- Зниження ваг стримує будь-які недоречні значення, дозволяє у процесі оптимізації знайти найменші значення ваг, які б вирішували конкретну задачу.
- Зниження ваг послаблює уплив викидів на оптимізацію параметрів. Іншими словами, на вихідний результат моделі менше впливають зміни у вхідних даних.

В академічній спільноті L2 регуляризація також відома як гребінева регресія (ridge regression) або регуляризація Тихонова.

2.4.3 L1 регуляризація

L1 регуляризація або регресія LASSO [56] (Least Absolute Shrinkage and Selection Operator) полягає у додаванні L1 норми параметрів до цільової функції з метою обмеження зростання ваг. Нехай такою цільовою функцією буде (2.70). Тоді нова цільова функція з урахуванням L1 норми прийме вид:

$$J = J_{b. \text{cross-entropy}} + \lambda J_{L1}, \quad (2.74)$$

де λ – гіперпараметр регуляризації ($\lambda \in \mathbb{R}$), $J_{L1} = \|w\|_1 = \sum_k |w_k|$.

Правило оновлення ваг з урахуванням L1 регуляризації буде записуватись наступним чином:

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w} = w - \alpha \left[\frac{\partial J_{b. \text{cross-entropy}}}{\partial w} + \frac{\partial J_{L1}}{\partial w} \right] \quad (2.75a)$$

$$w \leftarrow w - \alpha \lambda sign(w) - \alpha \frac{\partial J_{b. \text{cross-entropy}}}{\partial w} \quad (2.75b)$$

Вплив гіперпараметра регуляризації λ на оновлення ваг мережі [54]:

- Дуже мале значення λ : відсутній явний ефект впливу.
- Належне значення λ : багато ваг стають нулями. Це називається “роздіженням ваг”. Оскільки зниження ваг в L1 регуляризації не залежить від їх величини ($\alpha \lambda \text{sign}(w)$), тому ваги зі значенням 0.001 будуть зменшуватись на таку ж саму величину як і ваги зі значенням 100. Зазвичай величина зниження ваг є дужку малою ($\alpha \lambda \ll 1$), тому ваги з малими значеннями швидше наблизяться до 0.
- Занадто велике значення λ : спостерігається плато, а це означає, що ваги рівномірно приймають значення навколо нуля. Якщо λ занадто велике, тоді зниження ваг $\alpha \lambda \text{sign}(w)$ буде значно більшим ніж градієнт $\alpha \frac{\partial J_{b, \text{cross-entropy}}}{\partial w}$. Таким чином, кожне оновлення ваг буде здійснюватись на величину $\approx \alpha \lambda \text{sign}(w)$ у напрямку, протилежному знаку ваг. Наприклад, якщо w є близьким до нуля, але при цьому трохи більше нуля, тоді значення w буде зміщено на $-\alpha \lambda$. Таким чином ширина плато повинна бути $2\alpha \lambda$.

Властивість розрідженності, породжена регуляризацією L1, широко використовується як механізм вибору ознак. Вибір ознак спрощує проблему машинного навчання, шляхом вибору підмножини доступних ознак, які слід використовувати [49, Chapter 7, pp. 232-233].

2.4.4 Багінг (Bagging)

Багінг (англ. Bagging = Bootstrap aggregating) – це метод зменшення втрат узагальнення шляхом об’єднання кількох моделей [57]. Ідея полягає в тому, щоб навчити кілька різних моделей окремо, а потім за допомогою цих моделей передбачити вихідний результат для тестових даних. Це приклад загальної стратегії машинного навчання, яка називається усередненням моделей (model averaging). Методи, що використовують цю стратегію, відомі як ансамблеві методи. Причина того, що усереднення моделей працює, полягає в тому, що різні моделі зазвичай не допускають однакових помилок на тестовому наборі [49, Chapter 7, pp. 253-255].

Для прикладу розглянемо ансамбль з k регресійних моделей. Припустимо, що кожна модель допускає помилку φ_i на кожному тестовому прикладі. Ці помилки відповідають багатовимірному нормальному розподілу з середнім нуль, дисперсіями $\mathbb{E}[\varphi_i^2] = v$ та коваріаціями $\mathbb{E}[\varphi_i \varphi_j] = c$. Тоді середня помилка, зроблена прогнозом усіх моделей ансамблю, дорівнюватиме $\frac{1}{k} \sum_i \varphi_i$, а середня квадратична помилка ансамблю визначатиметься так:

$$\mathbb{E} \left[\left(\frac{1}{k} \sum_i \varphi_i \right)^2 \right] = \frac{1}{k^2} \mathbb{E} \left[\sum_i \left(\varphi_i^2 + \sum_{i \neq j} \varphi_i \varphi_j \right) \right] = \frac{1}{k} v + \frac{k-1}{k} c \quad (2.76)$$

У випадку, коли помилки моделей ідеально корелують між собою і $v = c$, середня квадратична помилка ансамблю дорівнює v , таким чином усереднення за ансамблем жодним чином не допомагає зменшити втрати. У випадку, коли помилки моделей абсолютно не корелують між собою і $c = 0$, середня квадратична помилка ансамблю становить $\frac{1}{k} v$. Це означає, що середня квадратична помилка ансамблю обернено пропорційна його розміру. Іншими словами, ансамбль в середньому буде демонструвати таку ж саму помилку як і будь-яка окрема його модель, але, якщо моделі роблять незалежні помилки, ансамбль буде працювати значно краще, ніж окремо взяті моделі з цього ансамблю [49, Chapter 7, pp. 253-255].

Різні ансамблеві методи створюють ансамбль моделей по-різному. Наприклад, кожен учасник ансамблю може бути отриманий унаслідок навчання абсолютно різного типу моделей з використанням різних алгоритмів навчання чи цільових функцій. Багінг – це метод, який дозволяє використовувати декілька разів одну і ту ж модель, алгоритм навчання та цільову функцію. Зокрема, багінг включає створення k різних наборів даних. Кожен набір даних має таку ж кількість прикладів як і оригінальний набір, будується шляхом вибірки із заміною з оригінального набору. Це означає, що з великою ймовірністю в кожному наборі даних будуть присутні кілька повторюваних прикладів. Потім модель i тренується на наборі i . Відмінності між прикладами, які включені до кожного набору даних, призводять до відмінностей між навченими моделями [49, Chapter 7, pp. 253-255].

Нейронні мережі досягають досить широкого розмаїття розв’язків, тому від усереднення різних моделей часто можна отримати користь, навіть якщо усі вони навчались на одному наборі даних. Відмінності в ініціалізації

параметрів, виборі міні-пакетів та гіперпараметрах часто бувають достатніми, щоб різні учасники ансамблю робили частково незалежні помилки [49, Chapter 7, pp. 253-255].

2.4.5 Дропаут (Dropout)

Дропаут – метод регуляризації, запропонований у роботі [16], дозволяє ефективно поєднувати безліч різних архітектур шляхом випадкової деактивації нейронів у прихованих шарах мережі та вхідних ознак під час навчання. Дропаут є більш ефективним методом для регуляризації великих та складніших мереж [54], ніж методи регуляризації, які були розглянуті вище: багінг, L1 та L2 регуляризація.

Поєднання декількох моделей майже завжди покращує продуктивність методів машинного навчання. Однак, усереднення результатів багатьох окремо навчених великих нейронних мережах є надзвичайно ресурсно-затратною задачею. Як уже зазначалось раніше, поєднання кількох моделей є найбільш корисним, коли окремі моделі відрізняються одна від одної, а для того, щоб отримати ансамблі різних моделей потрібно, щоб вони мали різну архітектуру, налаштування або були навчені на різних даних, що характеризують певну задачу. Навчити багато різних архітектур – це непросте завдання, оскільки пошук оптимальних гіперпараметрів для кожної архітектури забирає, як правило, багато часу, а саме навчання кожної великої мережі вимагає багато як обчислювальних ресурсів, так і часу для навчання. Більше того, глибинні нейронні мережі зазвичай вимагають великих обсягів навчальних даних (уплив кількості навчальних даних на продуктивність різних моделей показано на рисунку 2.22), але дуже часто буває так, що навчальних даних, які характеризують певну задачу, є не настільки багато, щоб можна було навчити декілька різних мереж на різних підмножинах даних. Навіть, якщо б можна було навчити багато різних великих мереж, використовувати їх усіх в додатку, де важливо отримати швидку відповідь (прогноз) на реальних даних, було б неможливо [16].

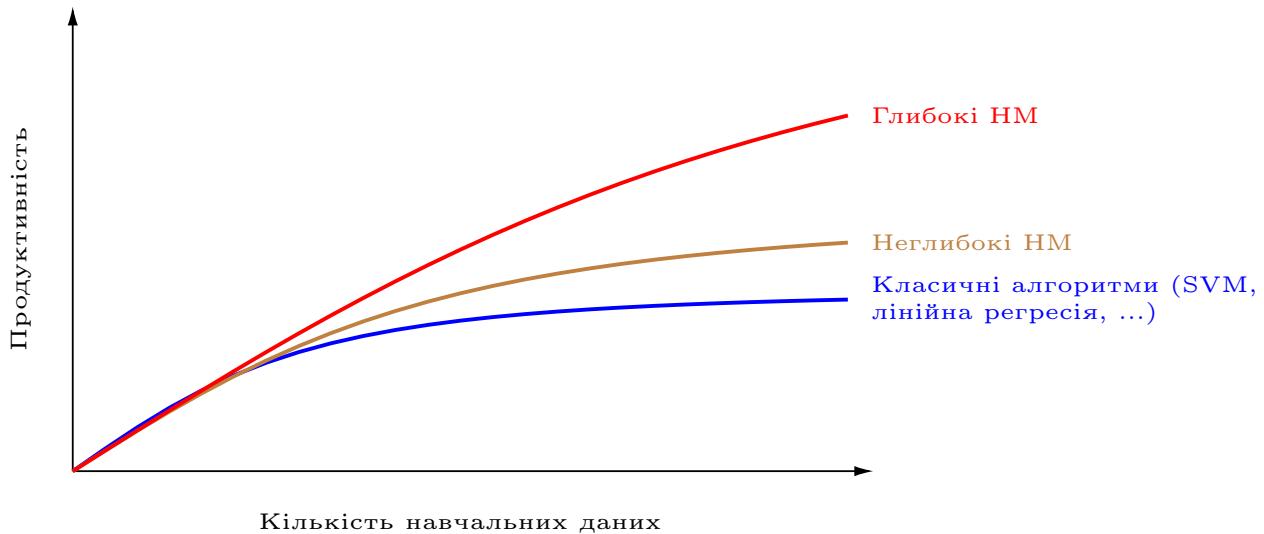


Рисунок 2.22 – Кількість навчальних даних vs продуктивність моделей [58]

Дропаут дозволяє вирішити зазначені вище проблеми. Цей метод запобігає перенавчанню та забезпечує ефективний спосіб поєднання експоненційної кількості різних архітектур нейронних мереж. Термін “дропаут” позначає виключення окремих частин (нейронів у прихованих шарах та вхідних ознак) нейронної мережі. Виключення або деактивація окремих частин означає тимчасове їх вилучення з мережі разом з усіма їх вхідними та вихідними зв’язками. Вибір, які вузли мережі слід деактивувати, є випадковим і залежить від їх ймовірності збереження. У найпростішому випадку кожен нейрон зберігається з фіксованою ймовірністю p незалежно від інших нейронів. Належне значення p може бути обране за допомогою перевірочного набору або просто встановити його рівним 0.5, що буде близьким значенням до оптимального для широкого кола мереж

та задач. Однак, для ознак, які подаються на вхід мережі, оптимальна ймовірність збереження зазвичай має бути близькою до 1, ніж до 0.5 [16].

Застосування дропаута до нейронної мережі породжує ансамбль підмереж. Якщо мережа складається з n частин (нейронів у прихованих шарах та вхідних ознак), тоді її можна розглядати як сукупність з 2^n можливих підмереж. Усі ці підмережі мають спільні ваги, так що загальна кількість параметрів все ще становить $\mathcal{O}(n^2)$ або менше. Для кожної нової ітерації вибирається нова підмережа для навчання. Іншими словами, навчання нейронної мережі з дропаутом можна розглядати як навчання ансамблю з 2^n підмереж [16]. Залежно від ймовірності збереження p , деякі підмережі будуть навчатись (якщо взагалі будуть) частіше за інші. Для прикладу розглянемо базову нейронну мережу, яка складається з двох вхідних ознак, трьох нейронів у прихованому шарі та одного нейрона на виході. Якщо до прихованого шару базової мережі застосувати дропаут, отримаємо ансамбль з $2^3 = 8$ підмереж, які можуть бути потенційно обрані для навчання. Графічне представлення базової мережі та ансамблю підмереж для описаного вище прикладу показано на рисунку 2.23. Слід зазначити, що у цьому прикладі не розглядається застосування дропаута до вхідного шару ознак, проте у загальному випадку дропаут може застосовуватись і до нейронів у прихованих шарах, і до вхідних ознак. Остання підмережа ансамблю (рисунок 2.23) має деактивовані усі нейрони у прихованому шарі, таким чином вхідний шар ознак не зв'язаний з вихідним шаром, тому ця підмережа взагалі навчатись не буде.

Під час тестування неможливо точно усереднити прогнози за експоненціальною кількістю моделей. Однак є дуже простий метод наближеного усереднення, який добре працює на практиці. Ідея полягає у використанні нейронної мережі під час тестування без деактивації окремих її частин. Ваги цієї мережі – це зменшені у масштабі значення навчених ваг. Якщо вузол (вхідна ознака або нейрон у прихованому шарі) обирається з ймовірністю p під час навчання, тоді вихідні ваги цього вузла множаться на p під час тестування, як показано у таблиці 2.3. Це гарантує, що для будь-якого прихованого нейрона очікуваний вихід (за розподілом, який використовується для деактивації нейронів під час навчання) буде таким самим, як фактичний вихід під час тестування. Таке масштабування дозволяє об'єднати 2^n підмереж зі спільними вагами в одну мережу для використання під час тестування [16].

Ідея, яка закладена в основу дропаут, не обмежується лише використанням в мережах прямого поширення. Загалом, дропаут можна застосовувати до графових моделей, наприклад, машини Болыцмана [16]. Зокрема, у роботі [16] наголошується на тому, що дропаут дозволяє використовувати значно вищий рівень вхідного шуму порівняно з шумопоглинальними автокодувальниками⁹ (DAEs). Також у цій роботі зазначається, що для дропаут часто оптимально виявлялась деактивація на рівні 20% вхідних та 50% прихованих вузлів мережі.

⁹DAEs додають трохи шуму на вхідні вузли. Зазвичай найкраще працюють з рівнем шуму 5%.

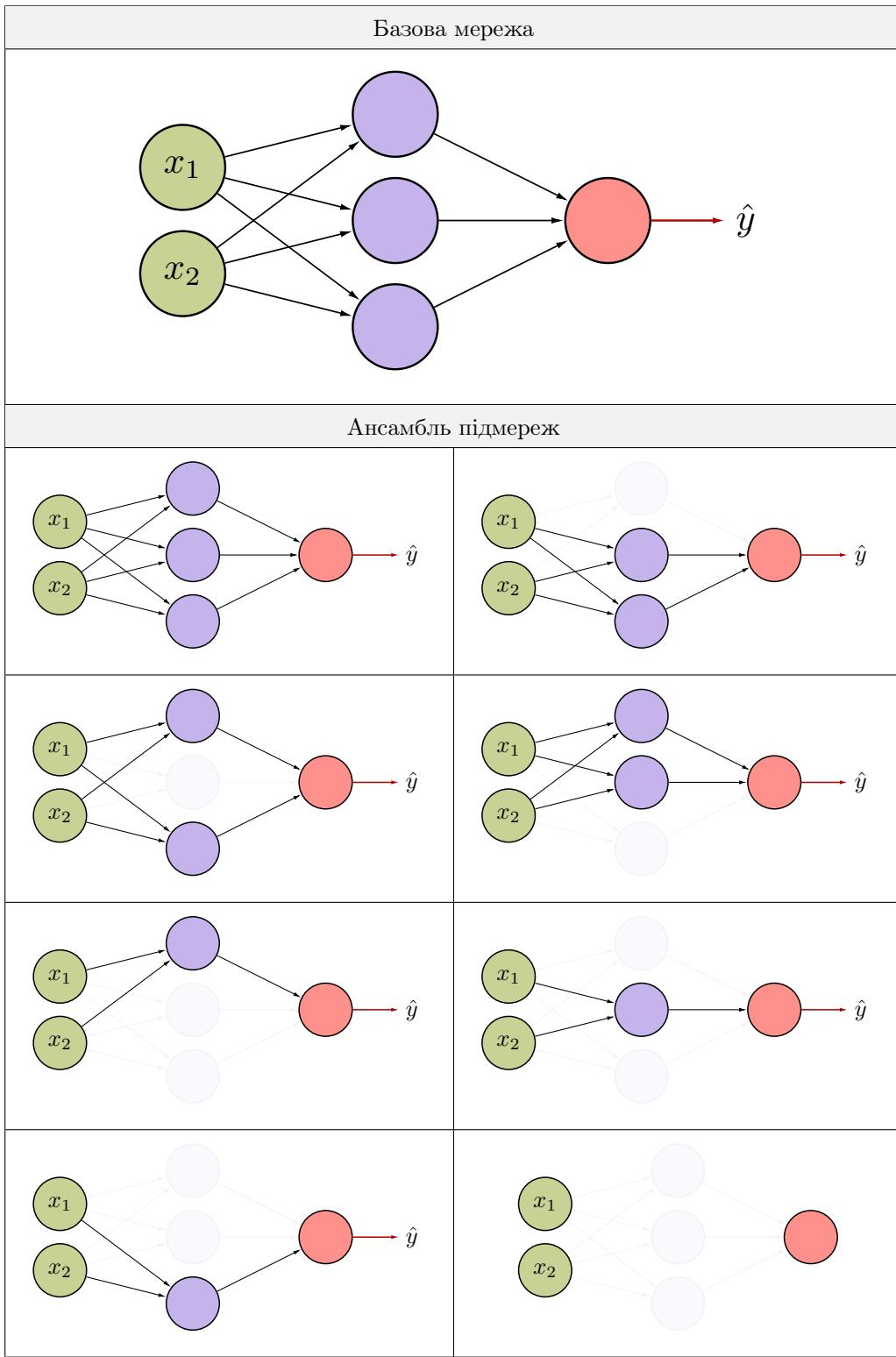
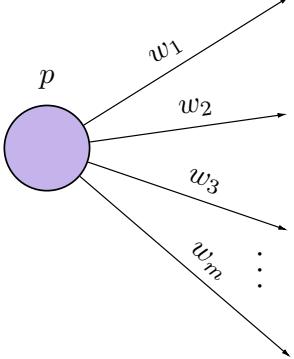
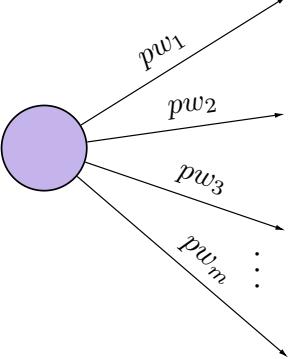


Рисунок 2.23 — Ансамбль підмереж

Табл. 2.3: Ідея наближеного усереднення ансамблю навчених підмереж під час тестування [16]

Під час навчання	Під час тестування
<p>Нейрон присутній з ймовірністю p. З'єднаний з нейронами в наступному шарі з вагою $\mathbf{w} = \{w_1, w_2, w_3, \dots, w_m\}$.</p> 	<p>Нейрон завжди присутній, а ваги множаться на p. Вихідний результат під час тестування такий же, як і очікуваний результат під час навчання.</p> 

2.5 Перехресна перевірка

Гіперпараметри – це вхідні значення алгоритму машинного навчання чи конвеєра, які впливають на якість моделі. Вони відіграють важливу роль у процесі навчання моделі. Деякі гіперпараметри впливають на швидкість навчання, але більшість контролює два компроміси: між зсувом та дисперсією і між точністю та повнотою. Як правило, гіперпараметри не оптимізуються алгоритмом навчання, проте в деяких стратегіях навчання може використовуватись адаптивна зміна значень гіперпараметрів, наприклад, кроку навчання, коли після певної кількості ітерацій не спостерігається прогрес оновлення навчальних параметрів моделі, тоді зменшують крок навчання. У кожної моделі машинного навчання та кожного оптимізаційного алгоритму є свій набір гіперпараметрів. Для налаштування гіперпараметрів часто використовують метод пошуку на сітці, випадковий пошук, пошук з подрібненням¹⁰, Байесівські методи та генетичні методи оптимізації. Усі ці методи налаштування гіперпараметрів використовуються лише тоді, коли є досить великий контрольний набір даних (валідаційна вибірка). На практиці часто виникає проблема, коли навчальних прикладів може бути замало, таким чином, ми не можемо дозволити собі зарезервувати значну частину цих даних для тестового і контрольного наборів. У такій ситуації було б доцільніше використовувати більше даних для навчання моделі, тому датасет розділяють лише на дві вибірки: навчальну та тестову. Для налаштування гіперпараметрів ми можемо на навчальному наборі здійснити імітацію контрольного набору за допомогою перехресної перевірки.

Перехресна перевірка працює наступним чином. Спочатку потрібно зафіксувати значення гіперпараметрів, які ми хочемо оцінити. Потім ділимо навчальний набір на кілька піднаборів однакового розміру. Кожен такий піднабір називається групою (fold). Зазвичай на практиці використовується п'ятигрупова перехресна перевірка, тобто випадково розділяється навчальна вибірка на п'ять груп однакового розміру: $\{F_1, F_2, \dots, F_5\}$ як показано на рисунку 2.24. У такому випадку, кожна група F_i , де $i = 1, \dots, 5$ буде містити 20% даних з навчальної вибірки. Далі потрібно навчити п'ять моделей на підмножині груп наступним чином. Для навчання першої моделі, f_1 , будуть використовуватись усі приклади з груп F_2, F_3, F_4, F_5 як навчальний набір, а приклади з групи F_1 – як контрольний набір. Аналогічним чином буде здійснюватись навчання другої моделі f_2 , потрібно лише змістити контрольний набір на одну групу праворуч, тобто: F_1, F_3, F_4, F_5 – навчальний набір,

¹⁰Пошук з подрібненням – комбінація пошуку на сітці та випадкового пошуку.

а група F_2 – контрольний набір. Тобто, у загальному випадку n -групової перехресної перевірки, модель f_n буде навчатись на усіх групах, крім n . Процес навчанняожної моделі f_i може здійснюватись послідовно або паралельно. Дляожної навченої моделі f_i ми визначаємо на контрольному наборі метрику, яка нас цікавить. Потім обчислюємо середнє значення для отриманих метрик, яке буде показувати кінцевий результат роботи цих моделей для заданих гіперпараметрів.



Рисунок 2.24 – Розбиття навчального набору даних для 5-групової перехресної перевірки

Для знаходження найкращих значень гіперпараметрів можна використовувати будь-який вище згаданий метод: пошук на сітці, випадковий пошук тощо. Після того, як було визначено оптимальні значення гіперпараметрів під час перехресної перевірки, зазвичай використовують увесь навчальний набір, щоб навчити остаточну модель з використанням знайдених гіперпараметрів. Потім отриману модель оцінюють на тестовому наборі.

2.6 Зменшення розмірності

Зменшення розмірності може бути корисними під час роботи з даними великої розмірності, оскільки призводить до більш ефективного моделювання, візуалізації та кращої продуктивності узагальнення. Методи зменшення розмірності дозволяють дослідити багатовимірний набір даних із сотнями ознак чи змінних та отримати краще представлення про використовувану множину даних з метою знаходження прихованих шаблонів та кращого розуміння датасету шляхом зменшення розмірності набору даних з довільного числа до двохвимірного чи трьохвимірного представлення, щоб можна було візуалізувати дані на екрані.

2.6.1 Метод головних компонент

Метод головних компонент (Principal Component Analysis – PCA) [59] – лінійний алгоритм, який використовують для зменшення розмірності, візуалізації даних, зменшення шуму та видобування нових ознак, які охоплюють важливу інформацію з оригінального багатовимірного набору. На практиці дуже часто доводиться працювати з складними даними, тобто багатовимірними. Візуалізація даних великої розмірності у просторі меншої розмірності дозволяє спростити пошук шаблонів чи кластерів у цих даних під час навчання моделей машинного навчання. PCA досягає цього шляхом перетворення багатовимірних даних у нову систему координат, де осі, які називаються головними компонентами, є лінійними комбінаціями ознак багатовимірного набору даних. Геометрична трансформація даних з простору ознак у простір головних компонент показано на рисунку 2.25.

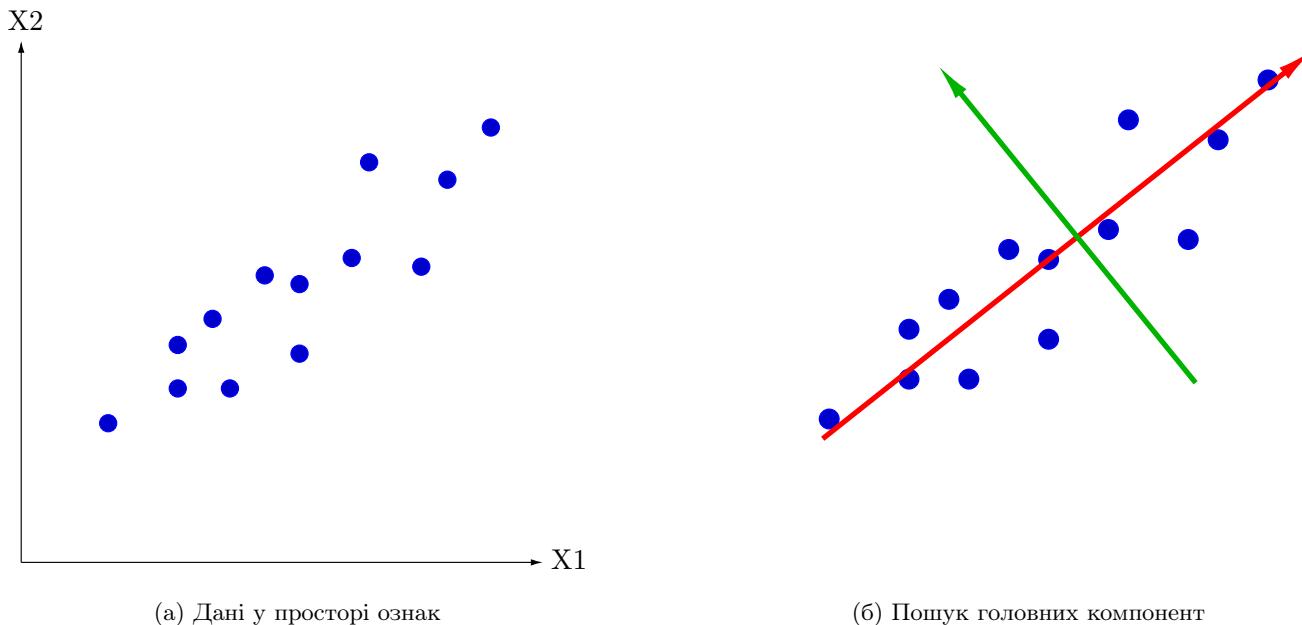


Рисунок 2.25 — Метод головних компонент: представлення даних у просторі ознак (а), пошук головних компонент (б) та представлення даних у просторі головних компонент (в)

PCA знаходить нову множину вимірів (набір базисних представлень), щоб усі виміри були ортогональними і, отже, лінійно незалежними та ранжувалися у порядку спадання дисперсії відносно цих вимірів. Таким чином, чим важливіший головний компонент (вісь), тим більшою буде дисперсія відносно цієї осі (ширший розподіл

даних) і тим вище буде знаходитись головний компонент у новому базисному представленні. Далі потрібно обрати перші k власних вектори та спроектувати n -вимірний набір ознак у новий k -вимірний простір.

PCA починається з центрування та нормалізації даних. Потрібно відцентрувати дані, розрахувавши відхилення кожної ознаки з набору даних відносно середнього значення, щоб отримати відносно центру середнє значення 0. Для того, щоб стандартне відхилення розподілу було рівне 1, потрібно відцентроване значення для усіх прикладів з набору даних розділити на величину стандартного відхилення σ_j :

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j}, \quad (2.77)$$

де x_j – ознака типу j з набору даних, $x_j^{(i)}$ – значення ознаки типу j для i -го прикладу з набору даних. Середнє значення μ_j та дисперсія σ_j^2 розраховуються таким чином:

$$\mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)} \quad (2.78a)$$

$$\sigma_j^2 = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2, \quad (2.78b)$$

де m – кількість прикладів у наборі даних.

Наступним кроком потрібно розрахувати коваріаційну матрицю для всіх ознак з набору даних:

$$A = \frac{1}{m} \sum_{i=1}^m x^{(i)} x^{(i)\top} \in \mathbb{R}^{n \times n} \quad (2.79)$$

Коваріаційна матриця A є симетричною матрицею:

$$\begin{pmatrix} Var(x_1) & Cov(x_1, x_2) & \dots & Cov(x_1, x_{n-1}) & Cov(x_1, x_n) \\ Cov(x_1, x_2) & Var(x_2) & \dots & Cov(x_2, x_{n-1}) & Cov(x_2, x_n) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Cov(x_1, x_{n-1}) & Cov(x_2, x_{n-1}) & \dots & Var(x_{n-1}) & Cov(x_{n-1}, x_n) \\ Cov(x_1, x_n) & Cov(x_2, x_n) & \dots & Cov(x_{n-1}, x_n) & Var(x_n) \end{pmatrix} \quad (2.80)$$

Потім потрібно знайти власні значення коваріаційної матриці та власні вектори:

$$(A - \lambda I)\mathbf{z} = 0, \quad (2.81)$$

де λ – власні значення, $\mathbf{z} \in \mathbb{R}^n \setminus \{0\}$ – власні вектори.

Далі відсортувати власні вектори за власними значеннями у порядку спадання та обчислити k ортогональніх головних власних векторів коваріаційної матриці для k найбільших власних значень. У кінці потрібно спроектувати n -вимірні дані у новий k -вимірний простір. Ця процедура максимізує дисперсію для всього k -вимірного простору.

PCA – це лінійний алгоритм, таким чином він не зможе інтерпретувати складний (нелінійний) поліноміальний зв'язок між ознаками. Лінійні алгоритми зосереджені на пошуку лінійних комбінацій вихідних ознак, які пояснюють найбільшу дисперсію в даних, але вони не враховують внутрішні нелінійні зв'язки, які можуть існувати між даними. Основною проблемою лінійних алгоритмів зменшення розмірності є те, що вони зосереджені на розміщенні схожих точок даних якомога далі одна від одної у просторі з меншою розмірністю. Таким чином, лінійні алгоритми зменшення розмірності обмежені у своїй здатності представляти подібні

точки даних близько одна до одної. Але для того, щоб представити дані високої розмірності на низькорозмірному нелінійному многовиді, важливо, щоб подібні точки даних були представлені близько один до одного, що зазвичай неможливо за допомогою лінійних алгоритмів зменшення розмірності. Тому на практиці часто використовуються нелінійні алгоритми, наприклад, t-SNE, ізометричне відображення (Isomap) тощо.

Література

- [1] P. McCorduck and C. Cfe, *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. CRC Press, 2004.
- [2] (2020) ISO/IEC TR 24028:2020(en) Information technology – Artificial intelligence – Overview of trustworthiness in artificial intelligence. Online Browsing Platform (OBP). [Online]. Available: <https://www.iso.org/obp/ui/#iso:std:iso-iec:tr:24028:ed-1:v1:en>
- [3] K. Yasuoka, “Roberta model pre-trained on Корпук ubertext for pos-tagging and dependency-parsing,” <https://huggingface.co/KoichiYasuoka/roberta-base-ukrainian-upos>, accessed: 2024-03-24.
- [4] H. Wickham, “Tidy data,” *Journal of statistical software*, vol. 59, pp. 1–23, 2014.
- [5] J. Schmidhuber. (2015) Deep learning. [Online]. Available: http://www.scholarpedia.org/article/Deep_Learning
- [6] ——. (2015) Critique of paper by “deep learning conspiracy”(nature 521 p 436). [Online]. Available: <https://people.idsia.ch/~juergen/deep-learning-conspiracy.html>
- [7] Івахненко and Лапа, “Кибернетические предсказывающие устройства,” *Київ:«Наукова думка»*, 1965.
- [8] A. G. Ivakhnenko, “Polynomial theory of complex systems,” *IEEE transactions on Systems, Man, and Cybernetics*, no. 4, pp. 364–378, 1971.
- [9] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.
- [10] D. Hebb, “The organization of behavior. a neuropsychological theory,” 1949.
- [11] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*, vol. 65, no. 6, p. 386, 1958.
- [12] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2021, <https://d2l.ai>.
- [13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [14] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [15] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
- [16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [17] C. M. Bishop, “Training with noise is equivalent to tikhonov regularization,” *Neural computation*, vol. 7, no. 1, pp. 108–116, 1995.
- [18] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv preprint arXiv:1409.0473*, 2014.

- [19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” *Advances in neural information processing systems*, vol. 27, 2014.
- [20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [21] M. Moravcik, M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, “Deepstack: Expert-level artificial intelligence in heads-up no-limit poker,” *Science*, vol. 356, no. 6337, pp. 508–513, 2017.
- [22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, no. 7587, pp. 484–489, 2016.
- [23] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv preprint arXiv:1712.01815*, 2017.
- [24] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of go without human knowledge,” *nature*, vol. 550, no. 7676, pp. 354–359, 2017.
- [25] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play,” *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.
- [26] F. A. Azevedo, L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, W. J. Filho, R. Lent, and S. Herculano-Houzel, “Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain,” *Journal of Comparative Neurology*, vol. 513, no. 5, pp. 532–541, 2009.
- [27] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.
- [28] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, “Maxout networks,” in *International conference on machine learning*. PMLR, 2013, pp. 1319–1327.
- [29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, 2012. [Online]. Available: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>
- [30] R. Girshick, “Fast r-cnn,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.
- [31] K. Janocha and W. M. Czarnecki, “On loss functions for deep neural networks in classification,” *arXiv preprint arXiv:1702.05659*, 2017.
- [32] A. Ng. CS230 Deep Learning. Advanced Evaluation Metrics. Stanford. [Online]. Available: <https://cs230.stanford.edu/section/8/>
- [33] J. Hui. mAP (mean Average Precision) for Object Detection. [Online]. Available: <https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173>
- [34] A Metric and A Loss for Bounding Box Regression. Stanford. [Online]. Available: <https://giou.stanford.edu/#home>
- [35] K. Katanforoosh and D. Kunin. (2019) Initializing neural networks. deeplearning.ai. [Online]. Available: <https://www.deeplearning.ai/ai-notes/initialization/>

- [36] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.
- [37] K. Katanforoosh, D. Kunin, and J. Ma. (2019) Parameter optimization in neural networks. deeplearning.ai. [Online]. Available: <https://www.deeplearning.ai/ai-notes/optimization/>
- [38] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson, “Averaging weights leads to wider optima and better generalization,” *arXiv preprint arXiv:1803.05407*, 2018.
- [39] L. Bottou, “Stochastic gradient descent tricks,” vol. 7700, pp. 430–445, January 2012. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/stochastic-gradient-tricks/>
- [40] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, “Don’t decay the learning rate, increase the batch size,” *arXiv preprint arXiv:1711.00489*, 2017.
- [41] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, large minibatch sgd: Training imagenet in 1 hour,” *arXiv preprint arXiv:1706.02677*, 2017.
- [42] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *International conference on machine learning*. PMLR, 2013, pp. 1139–1147.
- [43] S. P. Frankel, “Convergence rates of iterative treatments of partial differential equations,” *Mathematics of Computation*, vol. 4, no. 30, pp. 65–75, 1950.
- [44] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” *Ussr computational mathematics and mathematical physics*, vol. 4, no. 5, pp. 1–17, 1964.
- [45] Y. Nesterov, “A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$,” in *Doklady an ussr*, vol. 269, 1983, pp. 543–547.
- [46] I. Sutskever, *Training recurrent neural networks*. University of Toronto Toronto, Canada, 2013.
- [47] N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.
- [48] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *Journal of machine learning research*, vol. 12, no. 7, 2011.
- [49] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [50] G. Hinton, N. Srivastava, and K. Swersky. (2012) Neural networks for machine learning. [Online]. Available: <http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf>
- [51] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [52] S. J. Reddi, S. Kale, and S. Kumar, “On the convergence of adam and beyond,” *arXiv preprint arXiv:1904.09237*, 2019.
- [53] S. Reddi, M. Zaheer, D. Sachan, S. Kale, and S. Kumar, “Adaptive methods for nonconvex optimization,” in *Proceeding of 32nd Conference on Neural Information Processing Systems (NIPS 2018)*, 2018.
- [54] K. Katanforoosh and D. Kunin. Regularizing your neural networks. [Online]. Available: <https://rawgit.com/danielkunin/Deeplearning-Visualizations/master/regularization/index.html>
- [55] A. Krogh and J. A. Hertz, “A simple weight decay can improve generalization,” in *Advances in neural information processing systems*, 1992, pp. 950–957.
- [56] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.
- [57] L. Breiman, “Bagging predictors,” *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

- [58] A. Ng. Neural Networks And Deep Learning. Why is deep learning taking off? Deeplearning.ai. [Online]. Available: <https://www.youtube.com/watch?v=xflCLdJh0n0>
- [59] H. Hotelling, “Analysis of a complex of statistical variables into principal components.” *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933.