«Загальна теорія побудови нейромереж та людиноподібних роботів»

Від алгоритмів до втіленого інтелекту

Книга призначена для дослідників, інженерів та студентів, які працюють на стику штучного інтелекту, робототехніки та системного аналізу. Головна ідея: інтелект не може бути лише алгоритмом, він обов'язково має бути втіленим (embodied). Ми проведемо читача від математичних основ нейромереж через архітектури штучного мозку до інженерних принципів побудови роботів, здатних існувати та діяти в реальному світі.

Зміст

Глава 5. Мережі прямого поширення (Feedforward)
5.1. Багатошаровий перцептрон: теорія та практика
5.2. Завдання класифікації та регресії
5.3. Регуляризація: dropout, batch normalization
5.4. Межі та можливості повнозв'язних мереж

Глава 6. Згорткові нейронні мережі (CNN)
6.1. Операція згортки та її біологічний прототип
6.2. Ієрархія ознак у згорткових мережах
6.3. Відомі архітектури: LeNet, AlexNet, VGG, ResNet, Inception
6.4. Застосування: комп'ютерний зір, обробка зображень, медицина

Глава 7. Рекурентні нейронні мережі (RNN) та робота з послідовностями
7.1. Поняття часової залежності
7.2. Проблема довгострокових залежностей
7.3. LSTM та GRU: як запам'ятовувати надовго
7.4. Двонаправлені та глибокі рекурентні мережі
7.5. Застосування: NLP, машинний переклад, генерація тексту

Глава 8. Трансформери: революція в обробці послідовностей
8.1. Механізм уваги (attention): ідея та реалізація
8.2. Само-увага та багатоголова увага
8.3. Архітектура трансформера (кодер-декодер)
8.4. Позиційні кодування
8.5. BERT, GPT та їхні послідовники
8.6. Трансформери в комп'ютерному зорі (ViT)

Глава 9. Генеративні моделі
9.1. Автоенкодери та варіаційні автоенкодери (VAE)
9.2. Генеративно-змагальні мережі (GAN)
9.3. Дифузійні моделі
9.4. Моделі з нормалізуючими потоками (normalizing flows)
9.5. Оцінка якості генерації: метрики та методи

Глава 10. Графові нейронні мережі (GNN)
10.1. Чому графи? Дані зі складною структурою зв'язків
10.2. Згортки на графах
10.3. Архітектури: GCN, GraphSAGE, GAT
10.4. Застосування: соціальні мережі, молекули, логістика

Глава 11. Нейроморфні та спайкові мережі (SNN)
11.1. Біологічний нейрон: спайки, потенціали дії
11.2. Моделі спайкових нейронів (LIF, Izhikevich)
11.3. Навчання в спайкових мережах: STDP та його варіанти
11.4. Нейроморфні процесори: TrueNorth, Loihi, SpiNNaker
11.5. Енергоефективність та перспективи

Частина 3. Навчання та оптимізація

Глава 12. Методи оптимізації в глибокому навчанні
12.1. Градієнтний спуск та його варіанти: SGD, Momentum, Nesterov
12.2. Адаптивні методи: AdaGrad, RMSProp, Adam, AdamW
12.3. Вибір швидкості навчання та її розклад
12.4. Проблема сідлових точок та локальних мінімумів

Глава 13. Функції втрат та метрики якості
13.1. Функції втрат для регресії: MSE, MAE, Huber
13.2. Функції втрат для класифікації: перехресна ентропія (cross-entropy), hinge
13.3. Спеціалізовані функції втрат (для GAN, для сегментації)
13.4. Конструювання власних функцій втрат

Глава 14. Регуляризація та боротьба з перенавчанням
14.1. L1 та L2 регуляризація
14.2. Dropout та його варіанти
14.3. Batch Normalization, Layer Normalization
14.4. Data augmentation (аугментація даних)
14.5. Early stopping (рання зупинка)

Глава 15. Ініціалізація ваг та нормалізація
15.1. Проблема симетрії та її вирішення
15.2. Методи ініціалізації: Xavier, He
15.3. Вплив нормалізації на збіжність
15.4. Градієнтний кліпінг (gradient clipping)

Глава 16. Мета-навчання та few-shot learning
16.1. Як навчитися вчитися
16.2. Підходи: оптимізаційний, метричний, модельний
16.3. MAML та його варіанти
16.4. Прототипові мережі, Matching Networks

Глава 17. Навчання з підкріпленням (Reinforcement Learning)
17.1. Марковські процеси прийняття рішень
17.2. Q-навчання та Deep Q-Networks (DQN)
17.3. Policy Gradients та методи Actor-Critic
17.4. Навчання в неперервних просторах дій
17.5. Зворотний зв'язок та дослідження vs експлуатація
17.6. Багатоагентне навчання з підкріпленням

Частина 4. Інтеграція нейромереж у робототехнічні системи

Глава 18. Робот як втілена нейромережа
18.1. Від мозку до тіла: необхідність втілення
18.2. Сенсори: очі, вуха, шкіра робота
18.3. Актуатори: м'язи та приводи
18.4. Архітектура керування: централізована та розподілена

18.5. Реальний час та обмеження бортових обчислень

Глава 19. Сприйняття та комп'ютерний зір для роботів
19.1. Обробка візуальної інформації в реальному часі
19.2. Сегментація та виявлення об'єктів
19.3. Оцінка глибини та 3D-реконструкція
19.4. Слідкування за об'єктами та візуальна одометрія
19.5. Мультимодальна інтеграція: зір + лідар + радар

Глава 20. Слух та обробка мовлення
20.1. Перетворення звуку на спектрограми
20.2. Розпізнавання мовлення (ASR) на основі нейромереж
20.3. Синтез мовлення (TTS)
20.4. Локалізація джерела звуку
20.5. Розуміння інтонацій та емоцій

Глава 21. Планування рухів та керування
21.1. Кінематика та динаміка маніпуляторів
21.2. Планування шляху: A*, RRT, імовірнісні методи
21.3. Керування зусиллями та імпедансне керування
21.4. Навчання рухів (imitation learning)
21.5. Адаптація до мінливого середовища

Глава 22. Навігація та SLAM
22.1. Проблема одночасної локалізації та побудови карти
22.2. Візуальний SLAM (ORB-SLAM, LSD-SLAM)
22.3. SLAM на основі лідарів
22.4. Нейромережеві підходи до навігації
22.5. Дослідження незнайомої місцевості

Глава 23. Людино-роботна взаємодія
23.1. Розпізнавання жестів та поз
23.2. Розуміння намірів людини
23.3. Емоційний інтелект робота
23.4. Безпека при взаємодії
23.5. Соціальні роботи: етика та психологія

Частина 5. Вищий рівень: свідомість, самоусвідомлення, загальний інтелект (AGI)

Глава 24. Що таке свідомість з точки зору інженера
24.1. Проблема свідомості: філософські та наукові підходи
24.2. Інтегрована інформація (IIT) Джуліо Тононі
24.3. Глобальний робочий простір (GWT) Бернарда Баарса
24.4. Свідомість як властивість складних рекурентних мереж
24.5. Чи може машина володіти свідомістю? Критерії та тести

Глава 25. Самоусвідомлення та рефлексія в штучних системах
25.1. Моделі самого себе: навіщо агенту само-модель

Вступ: Запрошення до подорожі

Ми стоїмо на порозі епохи, коли межа між живим і штучним, між думкою та алгоритмом стає дедалі тоншою. Людиноподібні роботи, ще нещодавно головні герої фантастичних романів, впевнено входять у нашу реальність. Вони крокують на гідравлічних приводах, як Atlas від Boston Dynamics, вчаться спілкуватися, як мовні моделі нового покоління, й починають бачити світ крізь сенсори, які ми для них створюємо.

Але створення по-справжньому розумної машини, здатної не лише виконувати команди, а й розуміти, адаптуватися і, можливо, усвідомлювати, — це не просто інженерне завдання. Це виклик, що потребує синтезу знань із різних галузей: нейробіології, математики, фізики, програмування та філософії.

Ця книга — спроба такого синтезу.

Вона народилася з переконання, що інтелект не може існувати у вакуумі. Ми звикли сприймати нейромережі як абстрактні математичні моделі, що працюють у хмарі. Але справжній, «живий» інтелект, подібний до людського, обов'язково має бути втіленим (embodied). Йому потрібне тіло, щоб відчувати світ; йому потрібні сенсори, щоб сприймати цей світ; йому потрібні актуатори, щоб на нього впливати. Без тіла немає досвіду, без досвіду немає розуміння.

Тому наш шлях буде нелінійним. Ми пройдемо від базового математичного нейрона до найскладніших архітектур трансформерів, від алгоритмів зворотного поширення помилки до етичних проблем автономних систем. Ми поговоримо не лише про те, як працює градієнтний спуск, а й про те, чому мережа починає «розуміти» ієрархію ознак. Ми розглянемо не тільки, як запрограмувати рух робота, але й навіщо йому відчуття рівноваги та пропріоцепція.

Ця книга призначена для тих, хто не боїться міждисциплінарності. Для студентів, інженерів, дослідників і просто ентузіастів, які хочуть зазирнути за горизонт. Ми будемо використовувати строгу математичну мову, але не дозволимо їй заслонити головну ідею: ми створюємо не просто механізми, ми створюємо нову форму життя.

Ми навмисно не уникаємо складних питань. Говорити про свідомість, самоусвідомлення або ризики AGI (загального штучного інтелекту) так само важливо, як говорити про функції активації. Інженер майбутнього — це не тільки фахівець з коду чи «заліза», це мислитель, який розуміє наслідки своїх творінь.

Щасливої дороги, Читачу! Ми починаємо велике будівництво — від алгоритмів до втіленого інтелекту.

Частина 1. Фундаментальні основи штучного інтелекту

Глава 1. Що таке інтелект: природний та штучний

«Інтелект — це не привілей, а інструмент виживання. Питання не в тому, чи є він у нас, а в тому, як він влаштований, щоб ми могли відтворити його в іншій матерії.»

1.1. Визначення інтелекту: від психології до інженерії

Перш ніж будувати інтелект штучний, ми маємо домовитися про те, що ми взагалі будуємо. У цьому полягає перша і, мабуть, найскладніша методологічна проблема: поняття «інтелект» не має єдиного загальноприйнятого визначення.

У психологічній традиції XX століття склалося кілька підходів. Для Альфреда Біне, творця перших тестів IQ, інтелект був «здатністю правильно судити, розуміти та розмірковувати». Жан Піаже вбачав в інтелекті форму адаптації організму до середовища, вищий спосіб врівноваження суб'єкта з об'єктом. У рамках факторного аналізу Чарльз Спірмен виділив генеральний фактор інтелекту (g-factor), який передбачає існування певної загальної розумової енергії, що пронизує всі когнітивні здібності.

Однак для інженера, який приступає до створення системи, такі визначення надто розпливчасті. Нам потрібна операціоналізація — набір вимірюваних і конструктивних критеріїв. З інженерної точки зору, інтелект — це властивість системи набувати, обробляти та застосовувати знання й навички для досягнення цілей в умовах невизначеності та мінливості середовища.

Це визначення містить ключові компоненти:

1. Суб'єктність (наявність цілей, нехай і заданих ззовні).
2. Здатність до навчання (спроможність набувати знання, а не тільки використовувати закладені).
3. Адаптивність (здатність діяти в умовах, не повністю передбачених розробником).
4. Цілеспрямованість (застосування знань для модифікації середовища або свого становища в ньому).

Сучасна наука про ШІ оперує поняттям раціонального агента. Агент — це все, що може сприймати середовище через сенсори та впливати на нього через актуатори. Інтелект агента вимірюється ступенем успішності його дій: наскільки добре він обирає послідовності дій, що максимізують певну міру успіху (функцію корисності). Таке трактування відходить від антропоморфізму і дозволяє оцінювати інтелект мурахи, людини і програми для гри в шахи за єдиною шкалою ефективності.

1.2. Природний інтелект: уроки еволюції та нейробіології

Штучні нейронні мережі були натхненні біологічними, але еволюція створила рішення, до яких інженери дійшли лише через десятиліття, а багато — ще не дійшли зовсім.

Головні уроки еволюції:

1. Ієрархічність обробки. Зорова кора ссавців влаштована ієрархічно: прості клітини реагують на лінії та краї, складні — на абстрактніші форми, а гіперскладні — на цілі об'єкти. Ця архітектура стала прообразом згорткових нейромереж (CNN).

2. Пластичність і спеціалізація. Мозок здатний перебудовувати свої нейронні зв'язки під впливом досвіду (нейропластичність). При цьому різні зони спеціалізуються на різних функціях (зір, слух, мовлення), що нагадує модульну архітектуру сучасних робототехнічних систем.

3. Енергоефективність. Людський мозок, споживаючи близько 20 Вт, вирішує завдання, для яких суперкомп'ютерам потрібні мегавати. Спайковий характер передачі сигналу (рідкісні, але інтенсивні імпульси) та аналогова природа обчислень всередині нейрона дають уроки для нейроморфної інженерії.

4. Втіленість (Embodiment). Мозок не існує окремо від тіла. Наше сприйняття формується через рух: щоб зрозуміти об'єм предмета, ми повинні обійти його навколо або взяти в руки. Інтелект еволюціонував для керування тілом у просторі, і абстрактне мислення виросло із сенсомоторного досвіду.

Еволюція не проєктувала мозок «згори донизу». Вона методом спроб і помилок (природного відбору) нарощувала складність, надбудовуючи нові модулі над старими. Цей принцип модульності та нашарування також важливий для розуміння того, як будувати складні штучні системи.

1.3. Штучний інтелект: історія та парадигми

Історія ШІ — це історія зміни парадигм, кожна з яких намагалася відповісти на питання, як саме моделювати розум.

Символьний (логічний) підхід (1950-ті -- 1980-ті). Домінував на зорі ШІ. Інтелект постає як маніпуляція символами за формальними правилами. Логіка, семантичні мережі, експертні системи. Основоположники: Алан Тюрінг, Джон Маккарті, Марвін Мінскі. Сильна сторона: можливість роботи з абстрактними поняттями та складними логічними висновками. Слабкість: нездатність працювати із зашумленими даними та «здоровим глуздом» (фрейм-проблема).

Конекціоністський підхід (1980-ті -- теперішній час). Інтелект виникає із взаємодії безлічі простих елементів (нейронів), з'єднаних у мережу. Знання зберігаються не в явних правилах, а у вагах зв'язків. Основоположники: Френк Розенблатт (перцептрон), Джеффрі Гінтон, Ян Лекун. Сильна сторона: навчання на прикладах, стійкість до шуму, узагальнення. Слабкість: «чорна скринька», трудність інтерпретації рішень.

Поведінковий підхід (1990-ті). Інтелект — це властивість поведінки агента в середовищі, а не внутрішніх обчислень. Родні Брукс запропонував архітектуру «субсумпції», де складна поведінка складається з набору простих рефлексів (уникнення перешкод, рух на світло) без єдиного центру планування. Сильна сторона: робастність, швидкість реакції, зв'язок із фізичним світом. Слабкість: обмеженість абстрактного мислення.

Сучасний етап (2010-ті -- теперішній час) характеризується гібридизацією. Глибокі нейромережі (конекціонізм) інтегруються з модулями планування (символьний підхід) і працюють на реальних роботах (поведінковий підхід). Ми розуміємо, що ці парадигми не виключають, а доповнюють одна одну.

1.4. Сильний і слабкий ШІ: міфи та реальність

У дискусіях про ШІ важливо розрізняти два поняття, введені філософом Джоном Серлом.

Слабкий ШІ (Narrow AI / Weak AI) --- це системи, що моделюють інтелектуальну поведінку для вирішення конкретних завдань. Вони можуть обіграти чемпіона світу з Go, але не вміють заварити чай. Весь сучасний ШІ (розпізнавання облич, автопілоти, перекладачі, GPT) --- це слабкий ШІ. Вони --- інструменти, що імітують розуміння, але не володіють ним.

Сильний ШІ (Strong AI / General AI) --- це гіпотетична система, яка володіла б справжньою свідомістю, самоусвідомленням і здатністю мислити так само універсально, як людина. Це не просто програма, а «мисляча машина» в повному сенсі слова.

Серл у своєму уявному експерименті «Китайська кімната» показав, що маніпуляція символами за правилами (синтаксис) не породжує розуміння (семантики). Можна чудово відповідати на питання китайською, не знаючи китайської мови, якщо у вас є книга правил. Для Серла це доказ того, що навіть найдосконаліша програма залишиться слабким ШІ.

Однак заперечення полягає в тому, що система в цілому (кімната + людина + книга правил) може володіти розумінням, навіть якщо окремий її елемент не розуміє. Ця суперечка залишається відкритою, але для інженера важливо розуміти: створюючи складні

нейромережеві ансамблі, ми поки що не вирішили проблему переходу від синтаксису до семантики, від обчислення до переживання.

1.5. Інтелект як властивість системи, а не алгоритму

Ключовий тезис, який буде червоною ниткою проходити через всю книгу: інтелект — це не алгоритм, це властивість складної системи.

Алгоритм --- це послідовність інструкцій. Його можна записати на папері, виконати в умі. Але умовивід, що виникає в процесі виконання, не належить паперу чи чорнилу. Воно належить системі, що виконує алгоритм.

Так само інтелект не «живе» у вагах нейромережі чи в рядках коду. Він проявляється як емерджентна властивість динамічної системи, що включає:

· Архітектуру обробки інформації (нейромережа).
· Тіло (сенсори та актуатори), що взаємодіє із середовищем.
· Потік даних від середовища до системи і назад.

Емерджентність означає, що в системи з'являються властивості, яких немає в її компонентів окремо. Вода мокра, але ні атом водню, ні атом кисню окремо цією властивістю не володіють. Так і розуміння, намір, цілепокладання можуть виникати тільки на рівні системи в цілому, в її взаємодії зі світом.

Це накладає обмеження на методологію досліджень. Вивчати інтелект, розглядаючи тільки зрізи мозку (або тільки ваги навченої мережі) --- все одно що вивчати політ птаха за анатомією крила, ігноруючи повітря і гравітацію.

1.6. Втілений інтелект (embodied intelligence): чому тіло важливе

Ідея втіленого інтелекту (embodied cognition) перевертає класичне уявлення про те, що мозок --- це «генерал», а тіло --- «солдат», який виконує накази.

Тіло виконує три критично важливі функції:

1. Структурування сприйняття. Те, що ми бачимо, залежить від того, де ми знаходимося і куди дивимося. Рух тіла (поворот голови, переміщення в просторі) активно формує сенсорний вхід. Робот з нерухомою камерою бачить світ інакше, ніж робот, який може обійти об'єкт.

2. Зниження обчислювальної складності. Мудрість тіла в тому, що воно бере на себе частину обчислень. Нам не потрібно розраховувати кожен крок за законом Ньютона --- ми просто йдемо, і біомеханіка ніг забезпечує стійкість. Ходьба --- це не тільки команда мозку, це пасивна динаміка кістково-м'язової системи.

3. Формування абстрактних понять. Згідно з теорією когнітивної лінгвістики (Лакофф, Джонсон), наші абстрактні поняття (любов, справедливість, час) метафорично ґрунтуються на тілесному досвіді («близькі стосунки», «важка втрата», «попереду на нас чекає майбутнє»). Якщо в штучного агента немає тіла, у нього не може бути й таких понять у людському сенсі.

Таким чином, створення людиноподібного робота --- це не просто встановлення нейромережі на рухому платформу. Це спроба відтворити той тип втіленості, який зробив можливим людський інтелект. Ми не знаємо, чи можна створити AGI без тіла. Але ми точно знаємо, що людський інтелект без тіла не існує.

Підсумок Глави 1

Ми визначили інтелект як властивість цілеспрямованої системи адаптуватися до середовища. Ми побачили, що природа дала нам прототипи рішень, до яких інженерія тільки починає наближатися. Ми розрізнили слабкий і сильний ШІ, зрозумівши, що створення останнього --- задача, яка виходить за рамки простого масштабування моделей. І, нарешті, ми зафіксували головний методологічний принцип: інтелект не існує поза тілом і поза середовищем.

Наступна глава занурить нас у математичний апарат, без якого неможливо ні зрозуміти, ні побудувати жоден з компонентів нашого робота. Ми перейдемо від філософії та визначень до цифр і функцій --- до основ штучних нейронних мереж.

Глава 2. Математичні основи нейронних мереж

«Нейронна мережа — це не магія, це багатошарова композиція функцій, яка завдяки теоремі про універсальну апроксимацію може описати будь-яку залежність, якщо ми правильно підберемо коефіцієнти. Вся магія — у підборі цих коефіцієнтів.»

2.1. Штучний нейрон: від Мак-Каллока і Піттса до наших днів

Історія штучних нейронних мереж починається не з комп'ютерів, а зі спроби математично змоделювати роботу живої нервової клітини. У 1943 році Воррен Мак-Каллок (нейрофізіолог) і Волтер Піттс (логік) запропонували першу формальну модель нейрона.

2.1.1. Модель Мак-Каллока-Піттса

Біологічний нейрон отримує сигнали через дендрити, обробляє їх у тілі клітини (сомі) і при досягненні порогу збудження передає імпульс через аксон іншим нейронам. Мак-Каллок і Піттс запропонували наступну формалізацію:

1. Нейрон має кілька бінарних входів $x_i \in \{0, 1\}$.
2. Кожен вхід має вагу w_i, яка може бути збуджувальною ($w_i > 0$) або гальмівною ($w_i < 0$).
3. Нейрон обчислює зважену суму входів: $S = \sum w_i \cdot x_i$ (підсумовування за i від 1 до n).
4. Якщо сума перевищує поріг θ, нейрон видає 1, інакше 0.

Математично це записується як:

$$y = \Theta(\sum w_i \cdot x_i - \theta)$$

де $\Theta(\cdot)$ — функція Гевісайда (сходинка): $\Theta(z) = 1$ при $z \geq 0$, і $\Theta(z) = 0$ при $z < 0$.

Значення цієї моделі:

Вперше було показано, що мережа таких простих елементів може обчислювати будь-які логічні функції (І, АБО, НЕ) і, отже, є універсальним обчислювальним пристроєм (еквівалентним машині Тюрінга за певних припущень про пам'ять).

Обмеження:

· Бінарні входи та виходи.
· Відсутність навчання (ваги передбачалися заданими).
· Нечутливість до аналогових сигналів.

2.1.2. Перцептрон Розенблатта

У 1958 році Френк Розенблатт розвинув ідею Мак-Каллока-Піттса, створивши перцептрон. Головні нововведення:

· Вхідні сигнали стали дійсними числами (не тільки 0 і 1).
· Було запропоновано алгоритм навчання — ітеративна зміна ваг на основі помилки.

Модель нейрона в перцептроні залишилася попередньою (зважена сума + нелінійна функція активації), але тепер ваги могли налаштовуватися.

2.1.3. Сучасний формальний нейрон

Сьогодні ми використовуємо узагальнену модель, яку будемо позначати наступним чином.

Нехай на вхід нейрона надходить вектор ознак $x = [x_1, x_2, ..., x_n]^T$. Нейрон виконує два послідовних перетворення:

1. Лінійне перетворення (афінне): обчислюється зважена сума з додаванням зміщення (bias). У матричній формі:
$z = w^T x + b = \sum w_i x_i + b$ (підсумовування за і від 1 до n)
де $w = [w_1, w_2, ..., w_n]^T$ — вектор ваг, b — вільний член (bias), який також називають порогом (зі знаком мінус відносно історичної моделі).
2. Нелінійне перетворення (активація): результат z пропускається через нелінійну функцію активації φ(·):
$y = \varphi(z) = \varphi(w^T x + b)$

Чому потрібне нелінійне перетворення?
Якщо б ми використовували тільки лінійні перетворення ($y = w^T x + b$), то композиція будь-якої кількості шарів залишилася б лінійною функцією від входів. Лінійні моделі мають обмежену виражальну здатність — вони не можуть вирішувати задачі, де дані не є лінійно роздільними (наприклад, задача XOR). Нелінійність дозволяє мережі апроксимувати як завгодно складні функції.

2.2. Функції активації: лінійні, нелінійні, їхня роль

Вибір функції активації критично впливає на спроможності мережі та збіжність навчання. Розглянемо основні.

2.2.1. Сигмоїда (логістична функція)

$\sigma(z) = 1 / (1 + e^{-z})$

Властивості:

· Стискає вхід в інтервал (0, 1).
· Інтерпретується як ймовірність.
· Монотонна, диференційовна.
· Похідна: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Проблеми:

· Насичення (saturation): При великих додатних або від'ємних z похідна близька до нуля, що сповільнює або зупиняє навчання (зникаючий градієнт).
· Не центрована відносно нуля: Вихід завжди додатний, що може викликати зигзагоподібну збіжність градієнтного спуску.

2.2.2. Гіперболічний тангенс (Tanh)

$\tanh(z) = (e^z - e^{-z}) / (e^z + e^{-z})$

Властивості:

· Стискає вхід в інтервал (-1, 1).
· Центрований відносно нуля (краща збіжність, ніж у сигмоїди).
· Похідна: $\tanh'(z) = 1 - \tanh^2(z)$.

Проблеми:

· Ті ж проблеми насичення, що й у сигмоїди, хоча й менш виражені.

2.2.3. ReLU (Rectified Linear Unit)

$\mathrm{ReLU}(z) = \max(0, z)$

Властивості:

· Обчислювально дуже проста.
· Не насичує для додатних z, що дозволяє боротися зі зникаючим градієнтом.
· Сприяє розрідженості активацій (багато нейронів дають 0).

Проблеми:

· Вмираючий ReLU (dying ReLU): При від'ємних z градієнт дорівнює 0, і якщо нейрон перейшов у цю зону, він може ніколи не відновитися (ваги перестають оновлюватися).
· Не центрований відносно нуля.

2.2.4. Варіанти ReLU

Leaky ReLU:
$\mathrm{LeakyReLU}(z) = \max(\alpha z, z)$
де α — мала константа (зазвичай 0.01). Дозволяє зберігати невеликий градієнт при від'ємних значеннях.

ELU (Exponential Linear Unit):
$ELU(z) = \{ z, якщо\ z \geq 0; \alpha(e^z - 1), якщо\ z < 0 \}$
Згладжує перехід в нуль, покращуючи стійкість навчання.

Swish / SiLU (Sigmoid Linear Unit):
$Swish(z) = z \cdot \sigma(z)$
Виявлена компанією Google (Ramachandran et al., 2017). Гладка, немонотонна функція, яка часто перевершує ReLU в глибоких мережах.

2.2.5. Softmax (для вихідного шару класифікації)

Перетворює вектор дійсних чисел $z = [z_1, ..., z_k]$ у розподіл ймовірностей за K класами:

$Softmax(z_i) = e^{zi} / (\sum e^{zj})$ (підсумовування за j від 1 до K)

Властивості:

· Гарантує $\sum Softmax(z_i) = 1$ (підсумовування за i від 1 до K).
· Кожен елемент в інтервалі (0, 1).

2.3. Архітектури мереж: прямого поширення, рекурентні, згорткові

Тип архітектури визначається характером зв'язків між нейронами.

2.3.1. Мережі прямого поширення (Feedforward Neural Networks, FNN)

Інформація рухається тільки в одному напрямку: від входу через приховані шари до виходу. Граф обчислень не містить циклів. Це найпростіші мережі, які також називають багатошаровими перцептронами (MLP), якщо вони мають більше одного прихованого шару.

Математично:
Мережу з L шарів можна представити як композицію функцій:

$f(x) = f^{(L)}(f^{(L-1)}(... f^{(1)}(x) ...))$

де кожен шар l виконує перетворення:

$h^{(l)} = \varphi^{(l)}(W^{(l)} h^{(l-1)} + b^{(l)})$

· $h^{(0)} = x$ — вхідний вектор,
· $W^{(l)}$ — матриця ваг шару l,
· $b^{(l)}$ — вектор зміщень,
· $\varphi^{(l)}$ — нелінійна функція активації.

2.3.2. Рекурентні нейронні мережі (Recurrent Neural Networks, RNN)

Містять зворотні зв'язки, що дозволяють інформації зберігатися в часі. Це архітектура для обробки послідовностей (текст, часові ряди, мовлення).

У кожний момент часу t мережа отримує вхід x_t і прихований стан з попереднього кроку h_{t-1}:

$h_t = \varphi_h(W_{xh} x_t + W_{hh} h_{t-1} + b_h)$

$y_t = \varphi_\gamma(W_{h\gamma} h_t + b_\gamma)$

Тут W_{hh} — матриця рекурентних зв'язків, яка і створює «пам'ять» мережі.

2.3.3. Згорткові нейронні мережі (Convolutional Neural Networks, CNN)

Використовують операцію згортки замість повнозв'язного множення. Створені для роботи з даними, що мають сітчасту структуру (зображення, двовимірні ґратки).

Ключова ідея:

· Локальні зв'язки: кожен нейрон пов'язаний тільки з малою областю попереднього шару (рецептивним полем).
· Спільні ваги: один і той самий фільтр (ядро згортки) застосовується до всіх позицій входу.
· Багатоканальність: кілька фільтрів виділяють різні ознаки.

Згортковий шар для двовимірного входу обчислюється як:

$F_{ijk} = \varphi(\sum (\text{за c від 1 до C}) \sum (\text{за u від 1 до H}) \sum (\text{за v від 1 до W}) K_{uvck} \cdot I_{i+u-1,\ j+v-1,\ c} + b_k)$

де I — вхідний тензор (зображення), K — ядро згортки, k — індекс фільтра.

2.4. Навчання як оптимізація: функція втрат і градієнтний спуск

Навчання нейронної мережі — це процес підбору параметрів (ваг і зміщень) $\Theta = \{ W^{(l)}, b^{(l)} \}$ (для всіх l від 1 до L) таким чином, щоб мінімізувати деяку міру помилки — функцію втрат $L(\Theta)$.

2.4.1. Функція втрат як цільова функція

Ми маємо набір навчальних даних $D = \{ (x^{(i)}, y^{(i)}) \}$ (для i від 1 до N), де $x^{(i)}$ — вхід, $y^{(i)}$ — бажаний вихід (цільове значення). Мережа дає передбачення $\hat{y}^{(i)} = f(x^{(i)}; \Theta)$.

Функція втрат вимірює, наскільки передбачення відрізняються від істинних значень.

Приклади:

· Для регресії (передбачення числа): середньоквадратична помилка $L = (1/N) \sum (\hat{y}^{(i)} - y^{(i)})^2$ (підсумовування за i від 1 до N).
· Для бінарної класифікації: бінарна крос-ентропія $L = -(1/N) \sum [y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})]$ (підсумовування за i від 1 до N).
· Для багатокласової класифікації: категоральна крос-ентропія $L = -(1/N) \sum \sum y_c^{(i)} \log \hat{y}_c^{(i)}$ (підсумовування за i від 1 до N і за c від 1 до K).

2.4.2. Градієнтний спуск

Ми шукаємо мінімум функції втрат. У загальному випадку $L(\Theta)$ — складна невипукла функція багатьох змінних, і аналітично знайти мінімум неможливо. Використовується ітеративний метод градієнтного спуску.

Ідея: у кожній точці обчислюємо напрямок якнайшвидшого спадання функції — антиградієнт $-\nabla L(\Theta)$, і робимо крок у цьому напрямку.

Правило оновлення параметрів:

$$\Theta_{t+1} = \Theta_t - \eta \nabla L(\Theta_t)$$

де:

· Θ_t — поточні параметри,
· η — швидкість навчання (learning rate), додатний гіперпараметр, що визначає величину кроку,
· $\nabla L(\Theta_t)$ — градієнт функції втрат за параметрами.

Геометрична інтерпретація:
Уявімо поверхню функції втрат у багатовимірному просторі параметрів як горбисту місцевість. Ми знаходимося в деякій точці і хочемо спуститися в низину (мінімум). Градієнт показує напрямок найкрутішого підйому. Рухаючись у протилежному напрямку, ми спускаємося вниз. Швидкість навчання — це довжина нашого кроку.

2.4.3. Варіанти градієнтного спуску

· Пакетний градієнтний спуск (Batch GD): градієнт обчислюється по всій навчальній вибірці. Точно, але повільно і потребує багато пам'яті.
· Стохастичний градієнтний спуск (SGD): градієнт обчислюється по одному випадковому прикладу ($x^{(i)}$, $y^{(i)}$). Швидко, але шумно, траєкторія стрибає.
· Міні-пакетний градієнтний спуск (Mini-batch GD): компроміс — градієнт обчислюється по підмножині (батчу) з m прикладів. Це стандарт у глибокому навчанні.

2.5. Алгоритм зворотного поширення помилки

Як обчислити $\nabla L(\Theta)$ для глибокої мережі з мільйонами параметрів? Аналітично розписувати похідні для кожної ваги неможливо. На допомогу приходить алгоритм зворотного поширення помилки (backpropagation), який є ефективним застосуванням правила ланцюга (chain rule) з математичного аналізу.

2.5.1. Прямий прохід (forward pass)

Спочатку ми подаємо вхід x у мережу і обчислюємо значення всіх проміжних змінних (активацій) та вихід $\hat{y}$. Для кожного шару l ми запам'ятовуємо:

· Вхід у функцію активації: $z^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}$
· Вихід після активації: $h^{(l)} = \varphi^{(l)}(z^{(l)})$

2.5.2. Зворотний прохід (backward pass)

Ми обчислюємо градієнти, рухаючись від виходу мережі до входу.

Крок 1: Градієнт за виходом мережі.
Обчислюємо, як зміна виходу впливає на помилку: $\partial L/\partial \hat{y}$. Для різних функцій втрат цей вираз різний (наприклад, для MSE: $\partial L/\partial \hat{y} = (2/N)(\hat{y} - y)$).

Крок 2: Градієнт за перед-активацією останнього шару.
Нехай $a^{(L)} = \hat{y}$ (вихід останнього шару). Використовуємо правило ланцюга:

$\partial L/\partial z^{(L)} = (\partial L/\partial h^{(L)}) \odot (\partial h^{(L)}/\partial z^{(L)}) = (\partial L/\partial \hat{y}) \odot \varphi'^{(L)}(z^{(L)})$
де $\odot$ — поелементне множення (Адамарове).

Крок 3: Градієнти за параметрами останнього шару.
$\partial L/\partial W^{(L)} = (\partial L/\partial z^{(L)}) \cdot (h^{(L-1)})^T$
$\partial L/\partial b^{(L)} = \partial L/\partial z^{(L)}$

Крок 4: Поширення помилки на попередній шар.
Ми хочемо дізнатися $\partial L/\partial h^{(L-1)}$, щоб продовжити зворотний хід:

$\partial L/\partial h^{(L-1)} = (W^{(L)})^T \cdot (\partial L/\partial z^{(L)})$

Крок 5: Повторюємо кроки 2-4 для шару L-1.

Таким чином, помилка «просочується» назад через мережу, звідси й назва алгоритму.

2.5.3. Інтуїтивне розуміння

Зворотне поширення можна уявити як процес «розподілу відповідальності». Кожна вага робить свій внесок у підсумкову помилку. Ми обчислюємо цей внесок (часткову похідну) і потім оновлюємо вагу так, щоб зменшити помилку.

2.6. Проблеми навчання: затухання градієнта, перенавчання, ініціалізація

Градієнтний спуск і зворотне поширення дали нам інструмент, але на практиці виникає безліч проблем.

2.6.1. Проблема затухання та вибуху градієнта (Vanishing/Exploding Gradients)

При зворотному поширенні градієнти проходять через безліч шарів, множачись на похідні функцій активації та на вагові матриці.

· Затухання: Якщо похідні активації малі (як у зонах насичення сигмоїди), градієнти експоненційно спадають з кількістю шарів. Ранні шари навчаються вкрай повільно або не навчаються зовсім.
· Вибух: Якщо ваги великі, градієнти можуть експоненційно зростати, призводячи до чисельної нестабільності та «вильоту» параметрів у NaN.

Рішення:

· Використання ReLU (немає насичення для додатних значень).
· Використання залишкових зв'язків (ResNet), що дозволяють градієнту «обходити» шари.
· Нормалізація (Batch Normalization).
· Градієнтний кліпінг (обрізка градієнтів) для боротьби з вибухом.

2.6.2. Перенавчання (Overfitting)

Мережа вивчає навчальні дані «напам'ять», включаючи шум, і втрачає здатність узагальнювати на нові приклади.

Ознаки: Помилка на навчанні мала, але на валідації велика і зростає.

Рішення: Регуляризація (L1, L2), Dropout, аугментація даних, early stopping.

2.6.3. Проблеми ініціалізації ваг

Якщо всі ваги ініціалізувати однаково (наприклад, нулями), нейрони в шарі будуть симетричні і ніколи не розірвуть симетрію в процесі навчання.

Правильна ініціалізація критично важлива:

· Занадто малі ваги -> сигнал затухає.
· Занадто великі ваги -> сигнал вибухає.

Стандартні методи:

· Xavier (Glorot) ініціалізація: для сигмоїди і tanh. Ваги ініціалізуються з розподілу з дисперсією Var(W) = 2 / (n_in + n_out).
· He ініціалізація: для ReLU. Var(W) = 2 / n_in.
Тут n_in — кількість входів нейрона, n_out — кількість виходів.

2.6.4. Прокляття розмірності

Зі зростанням кількості вхідних ознак і глибини мережі обсяг необхідних даних для навчання зростає експоненційно. Простір можливих комбінацій ознак величезний, і мережа повинна побачити достатньо прикладів, щоб робити осмислені узагальнення.

Підсумок Глави 2

Ми заклали математичний фундамент:

1. Формальний нейрон як базовий елемент.
2. Функції активації, що привносять нелінійність.
3. Основні архітектури мереж.
4. Навчання як задача оптимізації через градієнтний спуск.
5. Алгоритм зворотного поширення — «серце» навчання глибоких мереж.
6. Ключові проблеми та методи їх подолання.

Цей апарат буде використовуватися в усіх наступних главах. Наступний крок — зрозуміти, як із простої математики виникає щось, схоже на розуміння.

Глава 3. Від статистики до сенсу: як мережа вчиться розуміти світ

«Нейронна мережа не розуміє світ так, як розуміє людина. Але вона будує внутрішню модель реальності, яка дозволяє їй успішно діяти в цьому світі. Питання в тому, де проходить межа між обчисленням і розумінням.»

3.1. Ієрархія ознак: від пікселів до образів

Одне з найглибших відкриттів у галузі глибокого навчання полягає в тому, що мережі природним чином вибудовують ієрархію абстракцій. Це не закладається інженером явно, а виникає автоматично в процесі навчання.

3.1.1. Принцип ієрархічного навчання

Коли ми навчаємо глибоку мережу на візуальних даних, перші шари завжди вчаться розпізнавати прості геометричні примітиви: межі, лінії, кути, кольорові переходи. Середні шари комбінують їх у складніші структури: текстури, прості форми (кола, прямокутники), частини об'єктів (очі, колеса, вікна). Верхні шари збирають ці частини в цілісні об'єкти та сцени.

Ця ієрархія не випадкова — вона відображає структуру фізичного світу. Світ складається з об'єктів, об'єкти — з частин, частини — з поверхонь, поверхні обмежені контурами. Мережа, стикаючись з безліччю прикладів, змушена відкрити цю ієрархію заново, тому що це найекономніший спосіб опису візуальної реальності.

3.1.2. Візуалізація ієрархії

Дослідники навчилися візуалізувати, що саме «бачать» нейрони на різних шарах. Для цього використовуються методи активаційної максимізації: ми шукаємо вхідне зображення, яке максимально збуджує конкретний нейрон.

Результати вражають:

· Нейрони першого шару реагують на прості орієнтовані градієнти.
· Нейрони середніх шарів — на текстури та патерни (очі, носи, колеса).
· Нейрони верхніх шарів — на цілі об'єкти (собаки, обличчя, автомобілі), причому з деякою інваріантністю до положення та ракурсу.

3.1.3. Інваріантність та еквіваріантність

У міру просування по ієрархії мережа набуває найважливішої властивості — інваріантності до несуттєвих змін входу.

Згорткові мережі досягають інваріантності до зсуву завдяки операції згортки та пулінгу. Глибші шари можуть бути інваріантними до повороту, масштабу, освітлення. Це і є початок «розуміння»: мережа впізнає об'єкт незалежно від того, де він знаходиться на зображенні, під яким кутом повернутий і як освітлений.

Математично інваріантність можна записати так:

Якщо T — деяке перетворення входу (зсув, поворот), то для ідеально інваріантної ознаки виконується:

$$f(T(x)) = f(x)$$

Тобто ознака не змінюється при перетворенні. На практиці мережі наближаються до цієї властивості, але рідко досягають її повністю.

3.2. Теорема про універсальну апроксимацію

Один із фундаментальних результатів, що пояснюють, чому нейронні мережі працюють — це теорема про універсальну апроксимацію.

3.2.1. Формулювання теореми

У найпростішій формулюванні (Cybenko, 1989; Hornik, 1991) теорема стверджує:

Нейронна мережа з одним прихованим шаром, що містить скінченне число нейронів, і нелінійною функцією активації (сигмоїдною або подібною) може апроксимувати будь-яку неперервну функцію багатьох змінних з будь-якою бажаною точністю за достатньої кількості нейронів.

Більш формально:

Для будь-якої неперервної функції $f: [0,1]^n \to \mathbb{R}$, визначеної на одиничному гіперкубі, і будь-якого $\varepsilon > 0$ існує така одношарова нейронна мережа з функцією активації φ і скінченним числом нейронів N, що:

$$| f(x) - \sum_{i=1}^{N} w_i \varphi(a_i^T x + b_i) | < \varepsilon$$ для всіх x з області визначення.

3.2.2. Наслідки та обмеження

Що теорема дає:

· Гарантію того, що архітектура «достатньо великої мережі» принципово здатна вивчити будь-яку залежність.
· Теоретичне обґрунтування, чому нейромережі такі універсальні.

Що теорема НЕ гарантує:

· Що ми зможемо знайти потрібні ваги (оптимізація може застрягти в локальному мінімумі).
· Що мережа буде узагальнювати (вона може просто запам'ятати навчальні приклади).
· Що достатньо одного шару — глибина потрібна для ефективності, а не для принципової можливості.

Глибокі мережі часто потребують експоненційно менше нейронів для апроксимації тієї ж функції порівняно з мілкими. У цьому їхня сила.

3.2.3. Інтуїція

Теорему можна зрозуміти так: нейронна мережа — це як «конструктор Лего». У нас є безліч простих цеглинок (нейронів з нелінійностями). Складаючи їх у правильних комбінаціях, ми можемо побудувати будь-яку форму (функцію). Теорема стверджує, що в принципі потрібні цеглинки завжди знайдуться, якщо їх досить багато. Але як саме їх скласти — це вже задача навчання.

3.3. Що таке «розуміння» з точки зору нейромережі?

Філософи та когнітивісти сперечаються про природу розуміння століттями. Але для інженера, який створює інтелектуальні системи, потрібне операційне визначення.

3.3.1. Розуміння як стиснення

Одна з впливових ідей (пов'язана з теорією інформації та роботами Юргенса Шмідгубера) полягає в тому, що розуміння — це здатність стискати дані, знаходячи в них регулярності та закономірності.

Уявіть, що вам потрібно запам'ятати мільйон фотографій собак. Якщо у вас немає «розуміння», що таке собака, ви будете зберігати кожен піксель кожної фотографії — колосальний обсяг даних. Якщо у вас є поняття «собаки» — чотириногої тварини з хвостом, вухами, шерстю — ви можете зберігати тільки відхилення від прототипу. Стиснення колосальне.

Нейронна мережа в процесі навчання саме цим і займається: вона знаходить компактне представлення (латентний простір), яке дозволяє реконструювати дані з мінімальними втратами.

3.3.2. Розуміння як передбачення

Інший підхід: розуміти — означає вміти передбачати. Якщо я розумію фізику, я можу передбачити, куди впаде м'яч. Якщо я розумію мову, я можу передбачити, яке слово найімовірніше з'явиться наступним.

Сучасні мовні моделі (GPT тощо) демонструють вражаючі здібності до передбачення наступного токена. Чи можна сказати, що вони «розуміють» текст? Частково так: вони вловлюють синтаксис, семантику, стиль, контекст. Але їхнє розуміння обмежене статистичними закономірностями текстового корпусу і не спирається на досвід взаємодії зі світом.

3.3.3. Розуміння як моделювання світу

Найбільш глибока точка зору: розуміння — це побудова внутрішньої моделі світу, яка дозволяє симулювати можливі сценарії.

Дитина, яка розуміє, що м'яч пружний, може уявно уявити, що станеться, якщо вона вдарить м'ячем об підлогу. Їй не потрібно реально кидати м'яч щоразу — модель працює «в голові».

Для робота розуміння означало б наявність такої внутрішньої моделі, що дозволяє прогнозувати наслідки дій і планувати поведінку. Це вже ближче до того, що ми шукаємо в AGI.

3.4. Латентний простір та його геометрія

Одне з найпотужніших понять у сучасному глибокому навчанні — це латентний простір (latent space). Це внутрішнє представлення даних, яке мережа будує у своїх прихованих шарах.

3.4.1. Що таке латентний простір?

Розглянемо задачу стиснення зображень за допомогою автоенкодера. Мережа складається з двох частин:

· Кодувальник (encoder): перетворює вхідне зображення високої роздільної здатності (скажімо, 1024 пікселі) у вектор невеликої розмірності (наприклад, 100 чисел). Це і є латентне представлення.
· Декодувальник (decoder): відновлює зображення з цього стиснутого вектора.

У процесі навчання мережа вчиться так стискати зображення, щоб декодувальник міг їх відновити з мінімальними втратами. В результаті латентний вектор захоплює найсуттєвіше в зображенні: форму об'єктів, їх розташування, колір, текстуру, але відкидає несуттєві деталі (шум, випадкові варіації).

3.4.2. Геометрична структура

Дивовижне спостереження: латентний простір часто має гладку геометричну структуру. Якщо взяти два зображення — наприклад, обличчя чоловіка та обличчя жінки — і знайти їхні латентні вектори z_1 та z_2, то точки на відрізку між ними (лінійна інтерполяція) при декодуванні даватимуть плавний перехід від чоловічого обличчя до жіночого.

Більше того, у латентному просторі можна виконувати «векторну арифметику». Класичний приклад зі словами:

$$z(\text{«король»}) - z(\text{«чоловік»}) + z(\text{«жінка»}) \approx z(\text{«королева»})$$

Це означає, що мережа вивчила не просто статистичні кореляції, а якусь внутрішню структуру смислів, де відношення між поняттями кодуються як вектори зсуву.

3.4.3. Розмірність латентного простору

Вибір розмірності латентного простору — важливий гіперпараметр. Якщо розмірність занадто мала, мережа не зможе передати всю необхідну інформацію (недостатня ємність). Якщо занадто велика, мережа може почати запам'ятовувати шум і втрачати здатність до узагальнення.

Існує гіпотеза многовиду (manifold hypothesis): реальні дані (зображення, тексти, звуки) зосереджені в околі многовиду значно меншої розмірності, ніж повний простір ознак. Завдання навчання — знайти цей многовид і представити дані в його координатах.

3.5. Емерджентність властивостей у глибоких мережах

У міру зростання масштабу мереж (кількості шарів, кількості параметрів, обсягу даних) у них починають проявлятися властивості, які не закладалися явно і не спостерігаються в маленьких мережах. Це явище називається емерджентністю.

3.5.1. Приклади емерджентних здібностей

У великих мовних моделях (GPT-3, GPT-4) дослідники виявили, що при досягненні певного порогу розміру (десятки мільярдів параметрів) модель раптово починає демонструвати здібності, яких не було в менших версій:

· Переклад мовами, якими модель спеціально не навчалася.
· Розв'язання арифметичних задач у кілька дій.
· Ланцюжки міркувань (chain-of-thought).
· Розуміння гумору, іронії, культурних відсилань.

Ці здібності не додавалися явно — вони «виросли» самі з простого завдання передбачення наступного токена.

3.5.2. Причини емерджентності

Точного пояснення поки немає, але є гіпотези:

1. Комбінаторний вибух взаємодій: Зі зростанням кількості параметрів експоненційно зростає число можливих комбінацій ознак. Деякі комбінації дають якісно нові можливості.
2. Ієрархічне навчання: Глибокі мережі вивчають ієрархію абстракцій, і на певній глибині абстракції стають досить загальними, щоб застосовуватися до нових, небачених раніше задач.
3. Стиснення світу: Щоб добре передбачати наступний токен у величезному корпусі текстів, модель змушена побудувати внутрішню модель світу, який описується цими текстами. Ця модель неминуче включає причинно-наслідкові зв'язки, знання про фізику, психологію, культуру.

3.5.3. Фазові переходи

Деякі дослідники проводять аналогію з фазовими переходами у фізиці. Як вода при нагріванні до 100°C різко перетворюється на пару, так і мережа при досягненні певного розміру може різко набувати нових властивостей. Ця аналогія поки що радше поетична, але вона допомагає думати про масштабування як про якісну, а не тільки кількісну зміну.

3.6. Межі здатності до навчання: чого мережа не може вивчити в принципі

При всіх вражаючих успіхах важливо розуміти принципові обмеження нейромережевого підходу.

3.6.1. Необчислювані функції

Алан Тюрінг показав, що існують функції, які принципово не можуть бути обчислені жодним алгоритмом (наприклад, проблема зупинки). Нейронна мережа, будучи скінченним обчислювальним пристроєм, також не може їх вивчити, тому що вони потребують нескінченного часу або пам'яті.

На практиці це означає, що мережа не може вирішити задачу, яка вимагає повного і точного знання про майбутнє або про нескінченні процеси.

3.6.2. Недостатність даних і прокляття розмірності

Навіть якщо функція обчислювана, для її вивчення може знадобитися експоненційно багато прикладів. Наприклад, щоб вивчити функцію від 1000 змінних, яка може набувати будь-яких комбінацій значень, потрібно більше прикладів, ніж атомів у Всесвіті.

Мережі рятуються тим, що реальні дані мають структуру (гладкість, симетрії, ієрархії). Але якщо такої структури немає, мережа безсила.

3.6.3. Причинність vs кореляція

Нейронні мережі навчаються на кореляціях у даних. Вони можуть чудово передбачати, що після спалаху блискавки буде грім, тому що в навчальних даних ці події часто слідують одна за одною. Але мережа не «розуміє», що блискавка є причиною грому через фізичний процес нагрівання повітря.

Це фундаментальне обмеження: мережа не бачить втручань, вона бачить тільки спостереження. Щоб вивчити причинно-наслідкові зв'язки, потрібні експерименти, активна взаємодія зі світом. Це повертає нас до теми втіленого інтелекту та навчання через дії.

3.6.4. Проблема узагальнення за межами розподілу (out-of-distribution)

Мережі прекрасно працюють на даних, схожих на навчальні. Але варто з'явитися прикладу, який лежить за межами розподілу навчальної вибірки, передбачення можуть стати беззмістовними.

Наприклад, мережа, навчена розрізняти собак і котів на фотографіях, може абсолютно неадекватно реагувати на малюнок собаки або на фотографію, зроблену під незвичним кутом.

Людина справляється з такими ситуаціями набагато краще, тому що в неї є загальні поняття і модель світу, що дозволяє екстраполювати. У мереж такої моделі в повному сенсі поки що немає.

Підсумок Глави 3

Ми простежили шлях від статистики до того, що можна назвати «розумінням» у штучних системах:

1. Мережі вибудовують ієрархію ознак, що відображає структуру світу.
2. Теорема про універсальну апроксимацію дає принципову можливість вивчити будь-яку функцію.
3. «Розуміння» можна операціоналізувати як стиснення, передбачення і побудову моделі світу.
4. Латентний простір має геометричну структуру, де смисли кодуються як напрямки.
5. При масштабуванні виникають емерджентні властивості, не закладені явно.
6. Існують принципові обмеження: необчислюваність, необхідність структури, проблема причинності та узагальнення.

Глава 4. Інформація та енергія в нейромережах перенесе нас зі світу смислів у світ фізичних обмежень — ми поговоримо про те, скільки коштує мислення в джоулях і бітах.

Глава 4. Інформація та енергія в нейромережах

«Інформація — це не абстракція. Кожен біт, який ми обробляємо, потребує енергії. Кожен джоуль, витрачений на обчислення, нагріває Всесвіт. Створюючи штучний інтелект, ми повинні розуміти цю фізичну ціну мислення.»

4.1. Нейромережа як система переробки інформації

Щоб зрозуміти місце нейромереж у фізичному світі, потрібно поглянути на них крізь призму теорії інформації та термодинаміки. Нейронна мережа — це перетворювач інформації: на вході в неї один сигнал (дані), на виході — інший (рішення, класифікації, передбачення).

4.1.1. Інформація за Шенноном

Клод Шеннон у 1948 році заклав основи теорії інформації, визначивши міру невизначеності — ентропію. Для дискретної випадкової величини X з розподілом ймовірностей p(x) ентропія H(X) обчислюється як:

$$H(X) = -\sum p(x) \log_2 p(x)$$

Одиниця вимірювання — біт. Ентропія показує, скільки біт інформації несе в середньому одне повідомлення з даного джерела.

Приклад: Якщо монета абсолютно симетрична (орел і решка по 0.5), ентропія дорівнює 1 біту. Якщо монета завжди падає орлом (ймовірність 1), ентропія дорівнює 0 — ніякої невизначеності, нове повідомлення не несе інформації.

4.1.2. Інформація в нейромережі

У нейромережі інформація існує в декількох формах:

1. Вхідні дані: сира інформація із зовнішнього світу (пікселі, звукові відліки, текстові токени). Їх ентропія визначається природою джерела.
2. Вага мережі: довготривала пам'ять, що зберігає вивчені закономірності. Обсяг інформації у вагах можна оцінити як кількість біт, необхідну для їх точного зберігання з урахуванням шуму квантування.
3. Активації: проміжні представлення, які мережа обчислює при обробці конкретного входу. Це «робоча пам'ять», поточний стан обчислень.
4. Вихідні дані: результат роботи мережі, зазвичай з меншою ентропією, ніж вхід (мережа «стискає» інформацію, відкидаючи несуттєві деталі).

4.1.3. Пропускна здатність

Важлива характеристика — скільки інформації мережа може переробити за одиницю часу. Для згорткової мережі, що обробляє відео в реальному часі, це можуть бути гігабіти за секунду. Для спайкової нейроморфної системи — значно менше, але з колосальною енергоефективністю.

Пропускна здатність обмежена як архітектурою мережі (числом нейронів, зв'язків), так і фізичними можливостями апаратури (тактовою частотою, шириною шини).

4.2. Енергетичні витрати навчання та інференсу

Навчання і робота нейромережі (інференс) мають зовсім різний енергетичний профіль.

4.2.1. Енергія навчання

Навчання — це екстремально енерговитратний процес. Чому?

1. Багаторазовий прохід по даних: Навчальна вибірка переглядається десятки і сотні разів (епохи). Кожен прохід вимагає прямого і зворотного поширення для всіх прикладів.
2. Обчислення градієнтів: Зворотне поширення потребує приблизно в 2-3 рази більше операцій, ніж прямий прохід. Потрібно зберігати проміжні активації для кожного шару, щоб обчислити градієнти.
3. Ітеративність: Навчання триває тисячі або мільйони ітерацій, поки функція втрат не зійдеться до мінімуму.

Приблизні цифри:

· Навчання сучасної великої мовної моделі (GPT-3) потребувало близько 1300 МВт·год електроенергії. Це приблизно стільки ж, скільки споживає 130 середніх домогосподарств за рік.
· Викиди CO_2 при такому навчанні порівнянні з викидами декількох трансатлантичних перельотів.

4.2.2. Енергія інференсу

Інференс (використання навченої мережі) значно дешевший, але теж потребує ресурсів.

· Для одного запиту до GPT-3 потрібно приблизно 3.5 Вт·год (ват-години) енергії. В масштабах мільйонів користувачів це дає величезні дата-центри з колосальним енергоспоживанням.
· Невелика мережа на мобільному пристрої може споживати мілівати.

4.2.3. Звідки беруться цифри

Енергоспоживання визначається трьома основними факторами:

1. Кількість операцій з рухомою комою (FLOPs). Кожна операція множення і додавання потребує енергії. У сучасних чипах енергія на одну операцію FLOP становить порядку 1-100 пДж (пікоджоулів) залежно від технології.
2. Переміщення даних. Зчитати дані з пам'яті часто дорожче, ніж виконати над ними операцію. Пересилання даних між чипами і рівнями кешу може споживати на порядки більше енергії, ніж саме обчислення.
3. Витоки та накладні витрати. Навіть коли обчислення не виконуються, транзистори споживають енергію (статичне енергоспоживання). Охолодження дата-центрів додає ще 30-50% до витрат.

4.2.4. Зелений ШІ

Усвідомлення енергетичної проблеми породило рух «зеленого ШІ» (Green AI). Основні напрямки:

· Розробка енергоефективних архітектур.
· Використання відновлюваної енергії для навчання.
· Повторне використання попередньо навчених моделей (transfer learning) замість навчання з нуля.
· Порівняння моделей не тільки за точністю, але й за енерговитратами (FLOPs, параметри).

4.3. Фізичні межі обчислень: Ландауер, фон Нейман

Існують фундаментальні фізичні обмеження, які не можна обійти жодною інженерною думкою.

4.3.1. Принцип Ландауера

Рольф Ландауер, фізик з IBM, у 1961 році сформулював принцип, що пов'язує інформацію і термодинаміку:

Стирання одного біта інформації в обчислювальному пристрої неминуче супроводжується розсіюванням енергії в кількості не менше kT ln 2, де k — стала Больцмана, T — температура системи в кельвінах.

Математично:

$E_{min} = kT \ln 2$

При кімнатній температурі (T = 300 K):

· $k = 1.38 \times 10^{-23}$ Дж/К
· $kT \ln 2 \approx 2.9 \times 10^{-21}$ Дж ≈ 0.018 еВ (електронвольт)

Це надзвичайно мала величина. Сучасні комп'ютери витрачають в мільйони разів більше енергії на одну операцію. Але принцип Ландауера встановлює абсолютну нижню межу, нижче якої опуститися неможливо, навіть на ідеальних квантових комп'ютерах.

Фізичний сенс: Інформація пов'язана з ентропією. Стираючи біт, ми зменшуємо ентропію системи, а за другим законом термодинаміки це потребує компенсації у вигляді зростання ентропії оточення — тобто виділення тепла.

4.3.2. Обернені обчислення

З принципу Ландауера випливає важливий висновок: якщо ми не стираємо інформацію, а тільки переставляємо її (обернені обчислення), теоретично можна обійтися без енергетичних витрат. Однак на практиці обернені обчислення надзвичайно складно реалізувати для алгоритмів, які ми використовуємо в нейромережах.

4.3.3. Межі фон Неймана

Джон фон Нейман описав архітектуру комп'ютера, яка домінує досі: процесор, пам'ять, шина даних. У цій архітектурі є фундаментальне обмеження — вузьке горлечко фон Неймана (von Neumann bottleneck).

Швидкість роботи обмежена не швидкодією процесора, а пропускною здатністю каналу між процесором і пам'яттю. Пересилання даних з пам'яті в процесор і назад займає час і потребує енергії.

Нейроморфні процесори намагаються обійти це обмеження, поєднуючи пам'ять і обчислення (in-memory computing), подібно до того, як це робить біологічний мозок, де синапси одночасно зберігають вагу і беруть участь в обчисленні.

4.3.4. Межі мініатюризації

Закон Мура (подвоєння числа транзисторів кожні два роки) підходить до фізичних меж. Коли розміри транзисторів досягають одиниць нанометрів, починають проявлятися квантові ефекти:

· Тунелювання електронів через затвор.
· Статистичні флуктуації числа домішкових атомів.
· Теплові шуми, порівнянні з сигналами.

Подальший прогрес буде пов'язаний не з мініатюризацією, а з новими архітектурами та принципами обчислень.

4.4. Коефіцієнт енергообміну (УЗЕ) для нейромереж

У попередніх роботах ми ввели поняття Універсального Закону Енергообміну (УЗЕ). Застосуємо його до нейромережевих систем.

4.4.1. Формулювання УЗЕ для обчислень

Для будь-якої системи переробки інформації можна ввести коефіцієнт енергообміну K_energy, що показує, скільки енергії витрачається на один біт осмисленої інформації:

K_energy = E_total / I_useful

де:

· E_total — повна енергія, спожита системою за час роботи.
· I_useful — кількість корисної інформації, виробленої системою (в бітах).

4.4.2. Що вважати корисною інформацією?

У контексті нейромереж це нетривіальне питання. Можна запропонувати декілька варіантів:

1. Інформація на виході: кількість біт у вихідному повідомленні. Але якщо мережа видає один біт (так/ні), виконавши при цьому гігантські обчислення, цей підхід недооцінює складність задачі.
2. Зниження ентропії: різниця ентропії входу і виходу. Мережа зменшує невизначеність, і ця різниця — міра корисної роботи.
3. Складність обчислювальної задачі: можна оцінювати через кількість операцій, необхідних для розв'язання задачі на ідеальному комп'ютері, і перераховувати в енергетичний еквівалент через принцип Ландауера.

4.4.3. Порівняння біологічних і штучних систем

Застосуємо УЗЕ для порівняння:

Система Енергоспоживання Продуктивність (операцій/с) K_energy (операцій/Дж)
Людський мозок ~20 Вт ~10^{15} (оцінка синапсів) ~5×10^{13}
GPU (сучасний) ~300 Вт ~10^{13} (FP32) ~3×10^{10}
Нейроморфний чип Loihi ~0.5 Вт ~10^{11} (спайків) ~2×10^{11}

Людський мозок ефективніший за сучасні GPU в тисячі разів! Це і є головний виклик для інженерів — наблизитися до біологічної енергоефективності.

4.4.4. Енергетична вартість навчання людини

Цікавий розрахунок: людський мозок «навчається» близько 20-25 років до дорослого стану. При середній потужності 20 Вт і ККД засвоєння інформації (дуже груба оцінка) отримуємо:

$E_{human} = 20 \text{ Вт} \times (20 \times 365 \times 24 \times 3600) \text{ секунд} \approx 6.3 \times 10^9 \text{ Дж}$

Це приблизно 1750 кВт·год — стільки ж, скільки споживає електричний чайник за кілька років роботи. При цьому обсяг засвоєної інформації колосальний. Штучні системи поки програють за енергоефективністю навчання з величезним відривом.

4.5. Нейроморфні обчислення як шлях до енергоефективності

Усвідомлення енергетичної проблеми призвело до розвитку альтернативного підходу — нейроморфних обчислень, які намагаються наслідувати біологічний мозок не тільки архітектурно, але й фізично.

4.5.1. Основні принципи нейроморфних систем

1. Асинхронність і спайковість: У біологічному мозку нейрони обмінюються рідкісними імпульсами (спайками). Коли спайка немає — немає й енерговитрат. Це радикально знижує середнє енергоспоживання.
2. Обчислення в пам'яті (in-memory computing): Синапс одночасно зберігає вагу і бере участь в обчисленні. Не потрібно пересилати дані з пам'яті в процесор — головне джерело енерговитрат в архітектурі фон Неймана.
3. Аналогові або змішані сигнали: Замість цифрових обчислень з високою точністю використовуються аналогові процеси, які від природи енергоефективні (але шумні).
4. Масивний паралелізм: Мільйони простих обчислювальних елементів працюють одночасно, як нейрони в мозку.

4.5.2. Приклади нейроморфних процесорів

· TrueNorth (IBM, 2014):
  · 4096 ядер, 1 мільйон програмованих нейронів, 256 мільйонів синапсів.
  · Енергоспоживання: близько 70 мВт при роботі.
  · Енергоефективність: 46 гігаопераций на ват (в тисячі разів вище звичайних GPU).
· Loihi (Intel, 2018):
  · 128 ядер, 131 тисяча нейронів, 130 мільйонів синапсів.
  · Підтримує навчання на чипі (STDP — Spike-Timing Dependent Plasticity).
  · Енергоефективність до 10^4 разів вища за стандартні процесори для деяких задач.
· SpiNNaker (Манчестерський університет):
  · Масив з мільйонів ARM-ядер, що моделюють нейрони.
  · Спроєктований для моделювання великих ділянок мозку в реальному часі.
  · Менша енергоефективність, але величезна гнучкість.

4.5.3. Проблеми та обмеження нейроморфного підходу

1. Навчання: Спайкові мережі (SNN) складно навчати стандартним зворотним поширенням. STDP та інші біоподібні правила поки поступаються в ефективності.

2. Точність: Аналогові обчислення шумні і не підходять для задач, що вимагають високої точності (наприклад, фінансові розрахунки).
3. Програмування: Екосистема інструментів для нейроморфних процесорів тільки формується. Писати для них складно, немає звичних фреймворків на кшталт PyTorch.
4. Універсальність: Нейроморфні чипи чудово підходять для задач, схожих на ті, що вирішує мозок (сенсорна обробка, патерни), але погано справляються з символьними обчисленнями та алгоритмами.

4.5.4. Перспективи

Нейроморфні обчислення — не заміна, а доповнення до класичних архітектур. Ймовірно, майбутнє за гібридними системами:

· Класичні GPU/CPU для навчання і складних символьних обчислень.
· Нейроморфні співпроцесори для енергоефективного інференсу на борту роботів і в edge-пристроях.

Саме такі гібридні системи дозволять наблизитися до енергоефективності біологічного мозку і створити по-справжньому автономних людиноподібних роботів.

Підсумок Глави 4

Ми розглянули нейромережі як фізичні системи, що підпорядковуються законам термодинаміки та теорії інформації:

1. Інформація має енергетичну вартість, і ця вартість вимірна.
2. Навчання сучасних мереж потребує колосальних енерговитрат, порівнянних із річним споживанням міст.
3. Принцип Ландауера встановлює абсолютну нижню межу енерговитрат на стирання біта — kT ln 2.
4. УЗЕ для нейромереж дозволяє порівнювати біологічні та штучні системи за енергоефективністю, і мозок поки виграє з величезним відривом.
5. Нейроморфні обчислення — найбільш перспективний шлях до енергоефективності, що наближає нас до біологічних показників.

Частина 2. Архітектури і типи нейронних мереж почнеться з наступної глави. Ми перейдемо від фундаментальних основ до конкретних архітектур, які сьогодні використовуються в промисловості та дослідженнях.

Частина 2. Архітектури та типи нейронних мереж

Глава 5. Мережі прямого поширення (Feedforward Neural Networks)

«Багатошаровий перцептрон — це не просто історичний курйоз. Це фундаментальний будівельний блок, на якому тримається все сучасне глибоке навчання. Зрозуміти його — означає зрозуміти, як із простих цеглинок складаються складні собори інтелекту.»

5.1. Багатошаровий перцептрон: теорія та практика

Мережі прямого поширення (Feedforward Neural Networks, FNN), також відомі як багатошарові перцептрони (MLP), є найбазовішим типом штучних нейронних мереж. Незважаючи на свою позірну простоту, вони становлять основу переважної більшості сучасних архітектур.

5.1.1. Визначення та структура

Мережа прямого поширення — це архітектура, в якій інформація рухається тільки в одному напрямку: від вхідного шару через один або декілька прихованих шарів до вихідного шару. У графі обчислень відсутні цикли та зворотні зв'язки.

Математично мережу з L шарів можна представити як композицію функцій:

$f(x) = f^{L}(f^{L-1}(... f^{1}(x) ...))$

де кожен шар l виконує афінне перетворення з подальшою нелінійною активацією:

$h^{(l)} = \varphi^{(l)}(W^{(l)} h^{(l-1)} + b^{(l)})$

Тут:

· $h^{(0)} = x$ — вхідний вектор
· $W^{(l)}$ — матриця ваг шару l (розмірності $\dim(h^{(l)}) \times \dim(h^{(l-1)})$)
· $b^{(l)}$ — вектор зміщень (bias) шару l
· $\varphi^{(l)}$ — нелінійна функція активації
· $h^{(l)}$ — вихід шару l (активації)

5.1.2. Історичний контекст: від перцептрона Розенблатта до багатошарових мереж

У 1958 році Френк Розенблатт представив перцептрон — одношарову нейронну мережу, здатну класифікувати лінійно роздільні образи. Перцептрон складався з вхідного шару (сенсорів) і вихідного нейрона з функцією активації у вигляді порога.

Алгоритм навчання перцептрона був простий і елегантний:

$\Delta w_i = \eta \cdot (y\_target - y\_pred) \cdot x_i$

де η — швидкість навчання.

Однак у 1969 році Марвін Мінський та Сеймур Пейперт опублікували книгу «Перцептрони», де показали фундаментальне обмеження одношарового перцептрона: він не може розв'язати задачу XOR (виключне АБО). Це призвело до так званої «першої зими ШІ» — періоду спаду інтересу до нейромережевих досліджень.

Порятунок прийшов з усвідомленням, що багатошарові мережі здатні вирішувати нелінійні задачі. Додавання хоча б одного прихованого шару з нелінійною активацією дозволяє апроксимувати будь-яку функцію (як ми обговорювали в Главі 3). Однак для навчання таких мереж знадобився новий алгоритм — зворотне поширення помилки (backpropagation), який був популяризований у 1980-х роках Румельхартом, Гінтоном та Вільямсом.

5.1.3. Універсальність багатошарового перцептрона

Багатошаровий перцептрон є універсальним апроксиматором. Теорема про універсальну апроксимацію (Глава 3, розділ 3.2) гарантує, що за достатньої кількості нейронів у прихованому шарі MLP може апроксимувати будь-яку неперервну функцію з будь-якою бажаною точністю.

Однак на практиці глибина (кількість шарів) часто важливіша за ширину (кількість нейронів у шарі). Глибокі мережі потребують експоненційно менше нейронів для апроксимації тих самих функцій, що й мілкі. Це явище відоме як ефективність глибини.

5.1.4. Архітектурні міркування

При проєктуванні MLP інженер стикається з кількома ключовими рішеннями:

1. Кількість шарів. Замало шарів — мережа не зможе вивчити складну залежність (underfitting). Забагато — ризик перенавчання і проблеми із затуханням градієнта.
2. Кількість нейронів у шарі. Визначає ємність (capacity) мережі. Емпіричне правило: починати з кількості, що трохи перевищує розмірність входу, і збільшувати при необхідності.
3. Тип функції активації. Вибір залежить від задачі та глибини мережі (див. розділ 5.1.5).
4. Ініціалізація ваг. Критично важлива для збіжності (Xavier для tanh/sigmoid, He для ReLU).

5.1.5. Вибір функцій активації для прихованих шарів

У сучасних MLP найбільш популярні такі функції активації для прихованих шарів:

· ReLU (Rectified Linear Unit): $\varphi(z) = \max(0, z)$. Стандартний вибір для більшості задач завдяки простоті та відсутності насичення для додатних z.
· Leaky ReLU: $\varphi(z) = \max(\alpha z, z)$ з $\alpha \approx 0.01$. Вирішує проблему «вмираючого ReLU».
· ELU (Exponential Linear Unit): $\varphi(z) = \{ z, \text{якщо } z \geq 0; \alpha(e^z - 1), \text{якщо } z < 0 \}$. Забезпечує гладкий перехід.
· Swish / SiLU: $\varphi(z) = z \cdot \sigma(z)$. Часто перевершує ReLU в глибоких мережах.

Для вихідного шару використовуються спеціалізовані функції залежно від задачі (див. розділ 5.2).

5.2. Завдання класифікації та регресії

MLP можуть вирішувати широкий спектр задач. Вибір архітектури вихідного шару та функції втрат визначається типом задачі.

5.2.1. Регресія (передбачення неперервних величин)

Задача: Передбачити одне або декілька дійсних чисел. Приклади: передбачення ціни будинку, температури, координат об'єкта.

Архітектура вихідного шару:

· Кількість нейронів дорівнює розмірності виходу (1 для скалярної регресії, k для багатовимірної).
· Функція активації: лінійна ($\varphi(z) = z$). Ніякого стиснення не потрібно, виходом може бути будь-яке дійсне число.

Функція втрат:

· MSE (Mean Squared Error): $L = (1/N) \sum (y^{(i)} - \hat{y}^{(i)})^2$. Найбільш поширена. Штрафує великі помилки квадратично.
· MAE (Mean Absolute Error): $L = (1/N) \sum |y^{(i)} - \hat{y}^{(i)}|$. Більш стійка до викидів.
· Huber Loss: Комбінація MSE та MAE. Для малих помилок працює як MSE, для великих — як MAE.

5.2.2. Бінарна класифікація (два класи)

Задача: Віднести об'єкт до одного з двох класів. Приклади: спам/не спам, здоровий/хворий.

Архітектура вихідного шару:

· Один нейрон.
· Функція активації: сигмоїда ($\sigma(z) = 1 / (1 + e^{-z})$). Стискає вихід в інтервал (0, 1), що інтерпретується як ймовірність належності до позитивного класу.

Функція втрат:

· Бінарна крос-ентропія (Binary Cross-Entropy, BCE): $L = -(1/N) \sum [y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})]$

5.2.3. Багатокласова класифікація (K класів)

Задача: Віднести об'єкт до одного з K взаємовиключних класів. Приклади: розпізнавання цифр (0-9), класифікація зображень (кіт/собака/птах).

Архітектура вихідного шару:

· K нейронів (по одному на кожен клас).
· Функція активації: Softmax. Перетворює вектор логітів $z = [z_1, ..., z_k]$ у розподіл ймовірностей:

$\text{Softmax}(z_i) = e^{z_i} / (\sum e^{z_j})$ (підсумовування за j від 1 до K)

Властивості: $\sum \text{Softmax}(z_i) = 1$, кожен елемент в (0, 1).

Функція втрат:

· Категориальна крос-ентропія (Categorical Cross-Entropy, CCE): $L = -(1/N) \sum \sum y_c^{(i)} \log \hat{y}_c^{(i)}$ (підсумовування за i від 1 до N і за c від 1 до K)
· Тут $y_c^{(i)}$ — індикатор (1, якщо приклад i належить класу c, інакше 0).

5.2.4. Багатокласова класифікація з кількома правильними відповідями (multi-label)

Задача: Об'єкт може належати одночасно кільком класам. Приклади: теги до фотографії (може містити і кота, і собаку).

Архітектура вихідного шару:

· K нейронів.
· Функція активації: сигмоїда для кожного нейрона незалежно. Кожен вихід інтерпретується як ймовірність належності до даного класу.

Функція втрат:

· Бінарна крос-ентропія, підсумована за всіма класами.

5.3. Регуляризація: dropout, batch normalization

MLP, особливо глибокі, схильні до перенавчання (overfitting) — запам'ятовування навчальних даних замість узагальнення. Регуляризація — це набір технік для боротьби з перенавчанням.

5.3.1. Dropout

Ідея: Під час навчання випадковим чином «відключати» (дропати) частину нейронів з імовірністю p. На кожній ітерації працює випадкова підмножина мережі.

Як працює:

· Для кожного нейрона в шарі генерується маска Бернуллі з ймовірністю p залишитися активним.
· Активації нейронів множаться на цю маску.
· Градієнти поширюються тільки через активні нейрони.
· Під час інференсу (тестування) всі нейрони працюють, але їх виходи множаться на p (або, що еквівалентно, ваги масштабуються).

Математично:

$r^{(l)} \sim \text{Bernoulli}(p)$

$\tilde{h}^{(l)} = r^{(l)} \odot h^{(l)}$

$z^{(l+1)} = W^{(l+1)} \tilde{h}^{(l)} + b^{(l+1)}$

Чому це працює:

1. Ансамблевий ефект: Dropout можна розглядати як навчання величезного ансамблю мереж із загальною ваговою пам'яттю.
2. Запобігання коадаптації: Нейрони не можуть покладатися на присутність інших, кожен вчиться витягувати корисні ознаки незалежно.

На практиці: p зазвичай вибирають 0.5 для прихованих шарів і 0.8-0.9 для вхідного шару.

5.3.2. Batch Normalization (пакетна нормалізація)

Ідея: Нормалізувати активації кожного шару так, щоб вони мали нульове середнє та одиничну дисперсію в межах міні-пакета (батчу).

Алгоритм:
Для міні-пакета $B = \{x_1, ..., x_m\}$ обчислюємо:

$\mu_B = (1/m) \sum x_i$ (середнє по батчу)
$\sigma^2_B = (1/m) \sum (x_i - \mu_B)^2$ (дисперсія по батчу)
$\hat{x}_i = (x_i - \mu_B) / \sqrt{(\sigma^2_B + \varepsilon)}$ (нормалізація, ε для числової стабільності)
$y_i = \gamma \hat{x}_i + \beta$ (масштабування та зсув, γ та β — навчальні параметри)

Переваги:

1. Прискорення збіжності: Нормалізовані входи дозволяють використовувати вищі швидкості навчання.
2. Зниження чутливості до ініціалізації: Мережа стає більш робастною до початкових значень ваг.
3. Слабкий регуляризуючий ефект: Оскільки статистики обчислюються по батчу, вноситься деякий шум, подібно до dropout.

Важливо: Під час інференсу використовуються не статистики батча, а ковзне середнє статистик, накопичене під час навчання.

5.3.3. Layer Normalization

Альтернатива Batch Normalization, особливо корисна для рекурентних мереж і трансформерів. Нормалізація проводиться не по батчу, а за ознаками для кожного окремого прикладу:

$\mu = (1/H) \sum x_i$ (за всіма H нейронами шару)
$\sigma^2 = (1/H) \sum (x_i - \mu)^2$
$\hat{x}_i = (x_i - \mu) / \sqrt{(\sigma^2 + \varepsilon)}$

5.3.4. L1 та L2 регуляризація

Додавання штрафу до функції втрат за величину ваг.

· L2 регуляризація (weight decay): $L_{total} = L_{original} + \lambda \sum w_i^2$. Змушує ваги бути маленькими, рівномірно розподіленими.
· L1 регуляризація: $L_{total} = L_{original} + \lambda \sum |w_i|$. Призводить до розрідженості ваг (багато стають нульовими).

5.3.5. Data Augmentation (аугментація даних)

Штучне збільшення навчальної вибірки шляхом модифікації існуючих прикладів. Для зображень: повороти, зсуви, зміна яскравості, додавання шуму. Мережа бачить більше варіацій і краще узагальнює.

5.3.6. Early Stopping (рання зупинка)

Навчання припиняється, коли помилка на валідаційній вибірці перестає зменшуватися (або починає зростати), навіть якщо помилка на навчанні продовжує падати.

5.4. Межі та можливості повнозв'язних мереж

Незважаючи на свою універсальність, MLP мають обмеження, які важливо розуміти.

5.4.1. Що MLP роблять добре

1. Робота з табличними даними: Для структурованих даних (ознаки в рядках) MLP часто є найкращим вибором.
2. Універсальність: Можуть використовуватися як частина складніших архітектур (останні шари в CNN, «голови» в трансформерах).
3. Інтерпретованість (відносна): Простіше аналізувати внесок ознак, ніж у згорткових мережах.

5.4.2. Обмеження MLP

1. Прокляття розмірності: Кількість параметрів зростає квадратично з розмірністю входу. Для зображення 256×256 пікселів (65536 входів) навіть один прихований шар з 1000 нейронів дасть 65 мільйонів ваг тільки на першому шарі. Це непрактично.
2. Ігнорування просторової структури: MLP не враховує, що пікселі, які знаходяться поруч, пов'язані. Для MLP піксель (i, j) і піксель (i, j+10) — такі ж різні входи, як і будь-які інші. Це робить навчання на зображеннях вкрай неефективним.
3. Відсутність інваріантності: MLP не інваріантний до зсувів. Зображення кота зліва і справа для MLP — два абсолютно різних приклади, якщо тільки мережа не побачить обидва варіанти в навчанні.
4. Проблеми з послідовностями: Для обробки послідовностей (текст, часові ряди) MLP неефективний, оскільки потребує фіксованого розміру входу і не враховує часову структуру.

5.4.3. Коли використовувати MLP у сучасних системах

Незважаючи на появу більш спеціалізованих архітектур (CNN, RNN, трансформери), MLP залишаються затребуваними:

1. Як «голова» класифікатора: Після того як CNN витягла ознаки із зображення, MLP приймає рішення на основі цих ознак.
2. В архітектурах MLP-Mixer: Нещодавні дослідження показали, що правильно спроєктовані MLP можуть конкурувати з трансформерами в деяких задачах комп'ютерного зору.
3. У задачах з табличними даними: Змагання на платформі Kaggle часто виграються градієнтним бустингом, але добре налаштовані MLP також показують відмінні результати.
4. У навчанні з підкріпленням: MLP часто використовуються для апроксимації функцій цінності та політик.

Підсумок Глави 5

Ми розглянули фундаментальну архітектуру — багатошаровий перцептрон:

1. Його структура — композиція афінних перетворень і нелінійностей.
2. Історичне значення — від перцептрона Розенблатта до розв'язання задачі XOR.
3. Застосування до задач регресії, бінарної та багатокласової класифікації.
4. Ключові методи регуляризації: dropout, batch normalization, L1/L2, аугментація, early stopping.
5. Обмеження повнозв'язних мереж та області, де вони залишаються найкращим вибором.

Глава 6. Згорткові нейронні мережі (CNN) покаже, як подолати обмеження MLP при роботі із зображеннями, використовуючи ідеї локальності, розподілу ваг та ієрархії ознак, запозичені у біологічного зору.

Глава 6. Згорткові нейронні мережі (Convolutional Neural Networks, CNN)

«Зір — це не просто реєстрація пікселів. Це активний процес витягування ієрархії ознак: від ліній та країв до текстур, частин об'єктів і, нарешті, до цілісних образів. Згорткові мережі навчилися робити те, що природа відточувала мільйони років — бачити світ крізь призму ієрархії.»

6.1. Операція згортки та її біологічний прототип

Щоб зрозуміти згорткові мережі, потрібно зробити екскурс у нейробіологію. У 1960-х роках Девід Г'юбел та Торстен Візель провели серію експериментів на зоровій корі котів, за які пізніше отримали Нобелівську премію. Вони відкрили два фундаментальних типи клітин:

· Прості клітини (simple cells): реагують на орієнтовані лінії та краї у певному місці зорового поля.
· Складні клітини (complex cells): також реагують на орієнтацію, але мають просторову інваріантність — реагують на лінію незалежно від її положення в межах рецептивного поля.

Ці відкриття стали біологічним натхненням для архітектури CNN.

6.1.1. Визначення операції згортки

У математиці згортка (convolution) — це операція, яка показує, як форма однієї функції змінюється під впливом іншої. Для дискретних сигналів одновимірна згортка визначається як:

$(f * g)[n] = \sum f[m] \cdot g[n - m]$ (підсумовування за m)

У контексті нейромереж ми використовуємо дискретну двовимірну згортку. Вхідне зображення I (розміром H × W) і ядро згортки K (розміром $k_h \times k_w$) породжують карту ознак F:

$F[i, j] = (I * K)[i, j] = \sum \sum I[i + u, j + v] \cdot K[u, v]$ (підсумовування за u від 0 до k_h-1, за v від 0 до k_w-1)

Інтуїція: Ядро згортки — це невеликий фільтр (наприклад, 3×3 або 5×5), який «ковзає» по зображенню, обчислюючи скалярний добуток зі своїм рецептивним полем. Результат — карта активацій, яка показує, де в зображенні присутній патерн, на який налаштований фільтр.

6.1.2. Ключові властивості згортки

1. Локальні зв'язки (local connectivity): Кожен нейрон пов'язаний тільки з малою областю попереднього шару (рецептивним полем). Це відображає біологічний факт: нейрони зорової кори реагують на стимули тільки в певній області простору.
2. Спільні ваги (shared weights): Один і той самий фільтр (ядро згортки) застосовується до всіх позицій входу. Це означає, що якщо фільтр навчився розпізнавати горизонтальну лінію

у верхньому лівому куті, він розпізнає її і в нижньому правому. Це дає інваріантність до зсуву (translation equivariance).
3. Розрідженість взаємодій (sparse interactions): Завдяки локальності та спільним вагам, кількість параметрів радикально скорочується порівняно з повнозв'язним шаром.

6.1.3. Багатоканальність

Реальні зображення мають три колірних канали (RGB). Крім того, на кожному згортковому шарі ми зазвичай застосовуємо не один, а безліч фільтрів (наприклад, 64, 128, 256). Кожен фільтр шукає свій патерн. Таким чином, вихід згорткового шару — це тривимірний тензор: висота × ширина × кількість фільтрів.

Математично для вхідного тензора I (з c каналів) і набору з K фільтрів K_k (кожен розміром c × k_h × k_w) вихідний тензор F має K каналів:

F_k[i, j] = ∑ (за c від 1 до C) ∑ (за u) ∑ (за v) I_c[i + u, j + v] · K_k[u, v, c] + b_k

де b_k — зміщення для k-го фільтра.

6.2. Ієрархія ознак у згорткових мережах

Одна з найкрасивіших властивостей CNN — автоматична побудова ієрархії ознак. Чим глибший шар, тим більш абстрактні концепції він представляє.

6.2.1. Ранні шари (низькорівневі ознаки)

Перші згорткові шари (близькі до входу) вчаться розпізнавати прості геометричні примітиви:

· Орієнтовані межі (горизонтальні, вертикальні, діагональні)
· Колірні плями
· Прості текстури (точки, смужки)

Ці фільтри універсальні і часто виглядають однаково незалежно від датасету — їх можна переносити між різними задачами.

6.2.2. Середні шари (ознаки середнього рівня)

Наступні шари комбінують прості примітиви у складніші структури:

· Кути та з'єднання ліній
· Контури простих форм (кола, прямокутники)
· Текстури середньої складності (хутро, трава, цегляна кладка)
· Частини об'єктів (очі, колеса, вікна, носи)

6.2.3. Глибокі шари (високорівневі ознаки)

Найглибші шари (близькі до виходу) представляють цілісні об'єкти та семантичні поняття:

· Цілі об'єкти (собаки, обличчя, автомобілі, будинки)
· Категорії сцен (пляж, місто, ліс)
· Абстрактні поняття, пов'язані із задачею

6.2.4. Візуалізація ієрархії

Дослідники розробили методи візуалізації того, що «бачать» нейрони:

· Активаційна максимізація (activation maximization): пошук вхідного зображення, яке максимально активує конкретний нейрон.
· Обернена згортка (deconvolution): проєктування активацій назад у простір пікселів.

Ці методи наочно демонструють ієрархію: нейрони ранніх шарів реагують на прості текстури, середніх — на частини об'єктів, глибоких — на цілі об'єкти з інваріантністю до пози та ракурсу.

6.3. Відомі архітектури: від LeNet до ResNet та Inception

Еволюція CNN — це історія поступового збільшення глибини та покращення архітектурних рішень.

6.3.1. LeNet-5 (1998, Ян Лекун)

Автори: Ян Лекун, Йошуа Бенжіо та ін.

Призначення: Розпізнавання рукописних цифр (MNIST).

Архітектура:

· Вхід: 32×32 (зображення цифри)
· C1: згортковий шар (6 фільтрів 5×5) → 28×28×6
· S2: субдискретизація (average pooling 2×2) → 14×14×6
· C3: згортковий шар (16 фільтрів 5×5) → 10×10×16
· S4: субдискретизація (average pooling 2×2) → 5×5×16
· C5: згортковий шар (120 фільтрів 5×5) → 1×1×120 (фактично повнозв'язний)
· F6: повнозв'язний шар (84 нейрони)
· Вихід: 10 нейронів (RBF-функції)

Значення: Перша практична CNN, що застосовувалася для читання чеків у банкоматах. Заклала базовий патерн: згортка → пулінг → згортка → пулінг → повнозв'язні шари.

6.3.2. AlexNet (2012, Алекс Кріжевський, Ілля Суцкевер, Джеффрі Гінтон)

Подія: Перемога на ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 з величезним відривом (помилка 15.3% проти 26.2% у другого місця). Початок ери глибокого навчання.

Ключові інновації:

· ReLU активація: Замість tanh, що прискорило навчання і допомогло із затуханням градієнта.
· Dropout: Для боротьби з перенавчанням.
· Аугментація даних: Випадкові зсуви, відбиття, зміни яскравості.
· Навчання на GPU (два GTX 580): Використання паралелізму.

· Local Response Normalization (LRN): Нормалізація активності сусідніх нейронів (зараз використовується рідко).

Архітектура (спрощено):

· Вхід: 227×227×3
· Conv1: 96 фільтрів 11×11, stride 4 → 55×55×96
· MaxPool1: 3×3, stride 2 → 27×27×96
· Conv2: 256 фільтрів 5×5 → 27×27×256
· MaxPool2: 3×3, stride 2 → 13×13×256
· Conv3: 384 фільтри 3×3 → 13×13×384
· Conv4: 384 фільтри 3×3 → 13×13×384
· Conv5: 256 фільтрів 3×3 → 13×13×256
· MaxPool3: 3×3, stride 2 → 6×6×256
· FC6: 4096 нейронів
· FC7: 4096 нейронів
· FC8: 1000 нейронів (Softmax)

Значення: Довела, що глибокі CNN можуть вирішувати складні задачі комп'ютерного зору.

6.3.3. VGGNet (2014, Карен Сімонян, Ендрю Зіссерман)

Ключова ідея: Використання дуже маленьких фільтрів (3×3) з малим кроком (stride 1) та збільшення глибини.

Чому 3×3?

· Два шари 3×3 мають ефективне рецептивне поле 5×5, але з меншою кількістю параметрів і більшою нелінійністю.
· Три шари 3×3 дають рецептивне поле 7×7.

Архітектури: VGG16 та VGG19 (16 і 19 шарів з навчальними вагами).

Достоїнства: Проста та однорідна архітектура, легко масштабується.

Недоліки: Величезна кількість параметрів (VGG16 — 138 млн), повільне навчання.

Значення: Стала стандартом для витягування ознак (feature extraction) у багатьох задачах.

6.3.4. ResNet (Residual Networks, 2015, Каймін Хе та ін.)

Проблема: Зі збільшенням глибини мережі помилка на навчанні починає зростати (не через перенавчання, а через затухання градієнта та труднощі оптимізації). Це називається деградацією (degradation problem).

Рішення: Залишкові зв'язки (residual connections / skip connections).

Ідея: Замість того щоб вчити відображення H(x) = output, вчимо залишок (residual) F(x) = H(x) − x. Тоді вихід шару стає:

y = F(x) + x

Градієнт може текти напряму через skip connection, минаючи шари з градієнтами, що затухають. Це дозволяє навчати мережі з сотнями і навіть тисячами шарів.

Архітектури: ResNet-50, ResNet-101, ResNet-152 (число — кількість шарів).

Bottleneck блок: Для зменшення обчислювальної складності використовується блок 1×1 → 3×3 → 1×1, де згортки 1×1 змінюють розмірність.

Значення: ResNet здійснила революцію, дозволивши створювати по-справжньому глибокі мережі. Переможець ILSVRC 2015 з помилкою 3.57% (нижче людської).

6.3.5. Inception (GoogLeNet, 2014, Крістіан Сегеді та ін.)

Ключова ідея: Використання фільтрів різного розміру (1×1, 3×3, 5×5) в одному шарі та об'єднання їх виходів. Це дозволяє мережі вибирати, який масштаб ознак важливий.

Inception module (наївний):

· Одночасна згортка 1×1, 3×3, 5×5 та пулінг 3×3.
· Конкатенація результатів по глибині.

Inception module зі зменшенням розмірності:

· Додавання згорток 1×1 перед 3×3 та 5×5 для зменшення кількості каналів (bottleneck), що радикально знижує обчислювальні витрати.

Версії: Inception-v1 (GoogLeNet), v2, v3, v4, Inception-ResNet.

Значення: Висока обчислювальна ефективність при відмінній точності.

6.4. Застосування: комп'ютерний зір, обробка зображень, медицина

CNN знайшли найширше застосування в задачах, де дані мають просторову структуру.

6.4.1. Класифікація зображень

Базова задача: віднести зображення до одного з класів. Архітектури, описані вище (VGG, ResNet, Inception), є стандартом для класифікації. Сучасні моделі (EfficientNet, ConvNeXt) продовжують покращувати співвідношення точність/ефективність.

6.4.2. Детекція об'єктів (object detection)

Задача: знайти на зображенні всі об'єкти певних класів і вказати їх bounding box'и (прямокутні рамки).

Два основні підходи:

1. Two-stage detectors:
   · Region Proposal: Спочатку пропонуються регіони, де можуть бути об'єкти (наприклад, Selective Search або Region Proposal Network).
   · Classification and Regression: Потім кожен регіон класифікується і уточнюється його положення.

· Приклади: R-CNN, Fast R-CNN, Faster R-CNN, Mask R-CNN (додає сегментацію).
2. One-stage detectors:
· Передбачають класи та bounding box'и одразу на всій сітці зображення за один прохід.
· Швидші, але історично трохи менш точні.
· Приклади: YOLO (You Only Look Once), SSD (Single Shot MultiBox Detector), RetinaNet.

6.4.3. Сегментація зображень

Задача: класифікувати кожен піксель зображення.

· Семантична сегментація: Кожному пікселю присвоюється клас (наприклад, дорога, пішохід, автомобіль). Різні об'єкти одного класу не розрізняються.
· Архітектури: FCN (Fully Convolutional Networks), U-Net, DeepLab.
· Інстанс-сегментація: Кожен окремий об'єкт виділяється індивідуально (наприклад, автомобіль №1, автомобіль №2).
· Архітектури: Mask R-CNN.

U-Net заслуговує на особливу згадку. Це архітектура з симетричним кодером (стискаючим) і декодером (відновлюючим), з skip-з'єднаннями між відповідними рівнями. Стала стандартом у медичній сегментації.

6.4.4. Генерація зображень

CNN використовуються в генеративних моделях:

· GAN (Generative Adversarial Networks): Генератор на основі деконволюції (транспонованої згортки) створює зображення з шуму.
· VAE (Variational Autoencoders): Кодер на CNN стискає зображення в латентний простір, декодер відновлює.
· Дифузійні моделі: Процес поступового зашумлення та подальшого видалення шуму з використанням U-Net.

6.4.5. Медицина

CNN здійснили революцію в медичній візуалізації:

· Аналіз рентгенівських знімків (виявлення пневмонії, переломів).
· Сегментація пухлин на МРТ та КТ.
· Аналіз гістологічних зрізів (виявлення ракових клітин).
· Класифікація патологій сітківки ока.

6.4.6. Відео та просторово-часові дані

Для обробки відео використовуються 3D згортки (3D CNN), де ядро згортки має розмірність глибина (час) × висота × ширина. Приклади: C3D, I3D.

6.4.7. Комп'ютерний зір для роботів

У робототехніці CNN використовуються для:

· Детекції перешкод та об'єктів маніпуляції.
· Семантичної сегментації сцени для навігації.

· Оцінки глибини за монокулярним зображенням.
· Візуальної одометрії (визначення руху за відеорядом).

Підсумок Глави 6

Ми поринули в архітектуру, яка змінила світ комп'ютерного зору:

1. Операція згортки, натхненна біологією, забезпечує локальність, спільні ваги та інваріантність до зсуву.
2. CNN автоматично будують ієрархію ознак — від ліній до об'єктів.
3. Еволюція архітектур (LeNet → AlexNet → VGG → ResNet → Inception) — це історія подолання обмежень глибини та ефективності.
4. Спектр застосувань величезний: від класифікації до сегментації, від медицини до робототехніки.

Глава 7. Рекурентні нейронні мережі (RNN) та робота з послідовностями перенесе нас із простору в час — ми навчимося обробляти дані, які розгортаються в динаміці.

Глава 7. Рекурентні нейронні мережі (RNN) та робота з послідовностями

«Час — це те, що відрізняє пам'ять від передбачення. Рекурентні мережі вчаться зберігати слід минулого, щоб передбачати майбутнє. Вони — перша спроба інженерії вдихнути в машину відчуття тривалості.»

7.1. Поняття часової залежності

Світ не статичний. Ми говоримо реченнями, де важливий порядок слів. Ми дивимося відео, де кадри пов'язані в часі. Ми слухаємо музику, де ноти утворюють мелодію. Всі ці дані мають структуру послідовності.

7.1.1. Чому повнозв'язні та згорткові мережі не підходять для послідовностей?

MLP та CNN мають фундаментальне обмеження: вони працюють із вхідними даними фіксованого розміру і не враховують порядок елементів. Для MLP речення «кіт з'їв мишу» та «миша з'їла кота» — це просто різні набори слів, але зв'язку між позиціями немає.

Можна, звісно, подавати в MLP вікно фіксованої довжини (наприклад, останні 10 слів). Але:

1. Довжина контексту обмежена розміром вікна.
2. Мережа не бачить залежності, що виходять за межі вікна.
3. Параметрів стає дуже багато (векторизація кожного слова у вікні).

7.1.2. Типи задач на послідовностях

Задачі з послідовностями можна класифікувати за співвідношенням входу та виходу:

1. Один-до-одного (one-to-one): Стандартна класифікація (наприклад, зображення → мітка). Не потребує RNN.

2. Один-до-багатьох (one-to-many): Один вхід породжує послідовність. Приклад: генерація підпису до зображення (зображення → послідовність слів).
3. Багато-до-одного (many-to-one): Послідовність на вході, один вихід. Приклад: класифікація тональності тексту (послідовність слів → позитивна/негативна).
4. Багато-до-багатьох (many-to-many):
· Синхронний варіант: Вихід на кожному кроці. Приклад: покадрова класифікація відео.
· Асинхронний варіант: Вхідна послідовність перетворюється на вихідну іншої довжини. Приклад: машинний переклад (послідовність однією мовою → послідовність іншою).

7.1.3. Ідея рекурентності

Рекурентні нейронні мережі (RNN) вводять поняття прихованого стану (hidden state) — вектора, який передається від одного кроку послідовності до наступного. Цей стан відіграє роль пам'яті: на кожному кроці мережа оновлює його, враховуючи новий вхід і попередній стан.

Математично це виражається рекурентною формулою:

$h_t = f(h_{t-1}, x_t)$

де h_t — прихований стан в момент t, x_t — вхід в момент t, f — деяка функція (зазвичай нейронна мережа).

7.2. Базова архітектура RNN

7.2.1. Прямий прохід (forward pass) у простій RNN

Найпростіша RNN (іноді називається мережею Елмана) визначається наступними рівняннями:

$h_t = \tanh(W_{xh} x_t + W_{hh} h_{t-1} + b_h)$

$y_t = \text{softmax}(W_{h\gamma} h_t + b_\gamma)$ (для класифікації на кожному кроці)

де:

· $x_t \in \mathbb{R}^d$ — вхідний вектор на кроці t
· $h_t \in \mathbb{R}^h$ — прихований стан
· $y_t \in \mathbb{R}^k$ — вихід (передбачення)
· $W_{xh} \in \mathbb{R}^{h \times d}$ — матриця ваг для входу
· $W_{hh} \in \mathbb{R}^{h \times h}$ — матриця рекурентних ваг (створює пам'ять)
· $W_{h\gamma} \in \mathbb{R}^{k \times h}$ — матриця ваг для виходу
· b_h, b_γ — зміщення

Функція tanh забезпечує нелінійність і утримує значення прихованого стану в інтервалі (-1, 1), що допомагає боротися з вибухом градієнтів (але не повністю).

7.2.2. Розгортання в часі (unfolding)

Ключовий прийом для розуміння та навчання RNN — розгортання (unfolding) в часі. Ми представляємо RNN як глибоку мережу з кількістю шарів, що дорівнює довжині послідовності, де на кожному шарі ваги W_{xh}, W_{hh}, $W_{h\gamma}$ — однакові для всіх кроків.

Це найважливіша властивість: RNN використовує одні й ті самі ваги на кожному часовому кроці. Це дозволяє обробляти послідовності будь-якої довжини.

7.2.3. Навчання RNN: Backpropagation Through Time (BPTT)

Для навчання RNN використовується алгоритм зворотного поширення помилки в часі (Backpropagation Through Time, BPTT).

Ідея:

1. Розгорнути RNN на всю довжину послідовності.
2. Обчислити втрати на кожному кроці (або тільки на останньому, залежно від задачі).
3. Підсумувати втрати за всіма кроками.
4. Застосувати стандартне зворотне поширення до розгорнутої мережі.
5. Градієнти для ваг підсумовуються за всіма кроками.

Проблема: Градієнти проходять через безліч кроків, множачись на матрицю W_{hh} і похідну tanh на кожному кроці. Це призводить до тих самих проблем, що й у глибоких мережах прямого поширення, але ускладнених багаторазовим множенням на ту саму матрицю.

7.3. Проблема довгострокових залежностей

7.3.1. Затухання та вибух градієнтів в RNN

Розглянемо градієнт втрат L на кроці T по відношенню до прихованого стану на кроці t (де T >> t). При зворотному поширенні через розгорнуту мережу:

$\partial L/\partial h_t = \partial L/\partial h\_T \cdot \partial h\_T/\partial h_t$

За правилом ланцюга:

$\partial h\_T/\partial h_t = \prod$ (від k = t+1 до T) $\partial h_k/\partial h_{k-1}$

Кожен множник $\partial h_k/\partial h_{k-1}$ включає в себе матрицю W_{hh} і діагональну матрицю похідних tanh:

$\partial h_k/\partial h_{k-1} = \text{diag}(\tanh'(W_{xh}\, x_k + W_{hh}\, h_{k-1} + b_h)) \cdot W_{hh}$

Проблеми:

· Якщо власні значення $W_{hh} < 1$, градієнт експоненційно затухає (vanishing gradient). Мережа не може вивчити залежності довші за кілька кроків.
· Якщо власні значення $W_{hh} > 1$, градієнт експоненційно вибухає (exploding gradient), що призводить до чисельної нестабільності.

7.3.2. Чому важливі довгострокові залежності

У реальних даних залежності можуть бути дуже довгими:

· У тексті: займенник може відноситися до іменника за 10-20 слів до нього.
· У музиці: основна тема може повертатися через хвилини.
· У відео: дія може бути пов'язана з подією, що сталася давно.

Прості RNN не здатні вловлювати такі залежності.

7.3.3. Методи боротьби з вибухом градієнтів

Градієнтний кліпінг (gradient clipping): Якщо норма градієнта перевищує поріг, масштабуємо градієнт:

if ||g|| > threshold: g = (threshold / ||g||) · g

Це проста та ефективна техніка проти вибуху градієнтів.

З затуханням градієнтів боротися складніше — для цього були розроблені спеціалізовані архітектури.

7.4. LSTM та GRU: як запам'ятовувати надовго

7.4.1. LSTM (Long Short-Term Memory) — 1997, Гохрейтер та Шмідгубер

LSTM — це архітектура, спеціально розроблена для запам'ятовування довгострокових залежностей. Ключова ідея: ввести стан комірки (cell state) C_t, який проходить через всю послідовність, та вентилі (гейти), які контролюють, яку інформацію додавати, видаляти або видавати.

Структура LSTM:

На кожному кроці t LSTM має:

· Вхід: x_t
· Попередній прихований стан: h_{t-1}
· Попередній стан комірки: C_{t-1}
· Вихід: h_t (новий прихований стан) і C_t (новий стан комірки)

Вентилі (Гейти):

1. Забуваючий вентиль (forget gate): Вирішує, яку інформацію з попереднього стану комірки забути.
   $f_t = \sigma(W\_f \cdot [h_{t-1}, x_t] + b\_f)$
2. Вхідний вентиль (input gate): Вирішує, яку нову інформацію записати в стан комірки.
   $i_t = \sigma(W\_i \cdot [h_{t-1}, x_t] + b\_i)$
3. Кандидат на новий стан (candidate cell state):
   $\tilde{C}_t = \tanh(W\_c \cdot [h_{t-1}, x_t] + b\_c)$
4. Оновлення стану комірки:
   $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$
   ($\odot$ — поелементне множення, Адамарове)
5. Вихідний вентиль (output gate): Вирішує, яку частину стану комірки видати як прихований стан.
   $o_t = \sigma(W\_o \cdot [h_{t-1}, x_t] + b\_o)$
6. Новий прихований стан:
   $h_t = o_t \odot \tanh(C_t)$

Інтуїція:

· Стан комірки C_t несе інформацію через всю послідовність. Вентилі регулюють потік.
· Забуваючий вентиль може «закрити» непотрібну інформацію.
· Вхідний вентиль додає нову важливу інформацію.
· Вихідний вентиль вирішує, що показувати на виході.

Завдяки цій архітектурі градієнти можуть текти через стан комірки з мінімальними спотвореннями (тільки через забуваючий вентиль), що дозволяє навчати залежності довжиною в сотні кроків.

7.4.2. GRU (Gated Recurrent Unit) — 2014, Чо та ін.

GRU — це спрощена версія LSTM, яка об'єднує деякі вентилі.

Структура GRU:

1. Вентиль оновлення (update gate): Комбінація забуваючого та вхідного вентилів LSTM.
   $z_t = \sigma(W\_z \cdot [h_{t-1}, x_t])$
2. Вентиль скидання (reset gate): Вирішує, яку частину попереднього стану забути.
   $r_t = \sigma(W\_r \cdot [h_{t-1}, x_t])$
3. Кандидат на новий прихований стан:
   $\tilde{h}_t = \tanh(W\_h \cdot [r_t \odot h_{t-1}, x_t])$
4. Новий прихований стан:
   $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

Переваги GRU:

· Менше параметрів (немає окремого стану комірки), швидше навчається.
· Часто показує результати, порівнянні з LSTM.
· Менш схильна до перенавчання на малих даних.

7.4.3. LSTM vs GRU: що вибрати?

· LSTM: Класичний вибір, коли потрібна максимальна здатність запам'ятовувати довгі залежності і є достатньо даних.
· GRU: Хороший вибір для середніх за розміром даних, коли важлива швидкість навчання і менше параметрів.

На практиці обидві архітектури широко використовуються, і вибір часто робиться емпірично.

7.5. Двонаправлені та глибокі рекурентні мережі

7.5.1. Двонаправлені RNN (Bidirectional RNN)

У деяких задачах контекст з обох боків важливий однаково. Наприклад, при розпізнаванні мовлення або заповненні пропусків у тексті.

Ідея: Запустити дві незалежні RNN:

· Одну, що читає послідовність зліва направо (forward RNN).
· Іншу, що читає справа наліво (backward RNN).

Приховані стани обох RNN конкатенуються на кожному кроці:

$h_t = [h_t(\text{forward}), h_t(\text{backward})]$

Таким чином, на кожному кроці мережа має інформацію як про минулий, так і про майбутній контекст.

7.5.2. Глибокі RNN (Deep RNN)

Аналогічно глибоким мережам прямого поширення, можна укладати кілька шарів RNN один на одного. Вихід нижнього шару стає входом для верхнього:

$h_t^{(l)} = \text{RNN}^{(l)}(h_t^{(l-1)}, h_{t-1}^{(l)})$

Глибокі RNN можуть вивчати ієрархію часових абстракцій: нижні шари — короткі патерни, верхні — довгострокові структури.

7.6. Застосування: NLP, машинний переклад, генерація тексту

RNN та їх варіанти (LSTM, GRU) знайшли найширше застосування в задачах, пов'язаних із послідовностями.

7.6.1. Мовне моделювання (Language Modeling)

Задача: Передбачити наступне слово в послідовності на основі попередніх.

Це базова задача для NLP. Навчена мовна модель може:

· Оцінювати ймовірність речення (perplexity).
· Генерувати новий текст (продовжувати послідовність).

Архітектура: Зазвичай використовується LSTM або GRU з вихідним шаром Softmax над словником.

7.6.2. Машинний переклад (Machine Translation)

Класична архітектура для перекладу — sequence-to-sequence (seq2seq) з кодером і декодером.

· Кодер: RNN (зазвичай двонаправлена) читає вихідне речення і стискає його в контекстний вектор (останній прихований стан або комбінацію станів).
· Декодер: Інша RNN генерує переклад, слово за словом, приймаючи на вхід попереднє згенероване слово і контекстний вектор.

Проблема: Контекстний вектор фіксованого розміру — вузьке місце для довгих речень.

Рішення: Механізм уваги (attention) — декодер на кожному кроці звертається до всіх прихованих станів кодера, вибираючи найбільш релевантні. Це революційне покращення, яке призвело до появи трансформерів (Глава 8).

7.6.3. Генерація тексту

Навчена мовна модель може генерувати текст, ітеративно передбачаючи наступне слово і подаючи його на вхід:

$x_{t+1} = \text{argmax}\ p(x \mid x_1, ..., x_t)$

Для різноманітності використовують не argmax (жадібний пошук), а ймовірнісну вибірку (sampling) або beam search.

7.6.4. Розпізнавання мовлення (ASR)

Аудіосигнал розбивається на короткі фрейми (10-30 мс), з яких витягуються ознаки (мел-кепстральні коефіцієнти, MFCC). RNN (зазвичай двонаправлена LSTM) обробляє послідовність ознак і видає ймовірності фонем або букв.

7.6.5. Класифікація тональності (Sentiment Analysis)

Послідовність слів → позитивний/негативний відгук. Використовується many-to-one RNN, де останній прихований стан подається в класифікатор.

7.6.6. Генерація музики

RNN можна навчити на послідовностях нот. Мережа вчиться музичної структури і може генерувати нові мелодії.

7.6.7. Передбачення часових рядів

Фінанси, погода, попит — скрізь, де дані йдуть у часі, RNN можуть передбачати майбутні значення на основі минулих.

Підсумок Глави 7

Ми поринули у світ послідовностей і пам'яті:

1. RNN вводять прихований стан як форму пам'яті, що дозволяє обробляти послідовності змінної довжини.
2. Навчання через BPTT страждає від проблеми затухання/вибуху градієнтів, що обмежує здатність простих RNN до запам'ятовування довгострокових залежностей.
3. LSTM та GRU вирішують цю проблему за допомогою вентилів, які контролюють потік інформації. LSTM додає окремий стан комірки, GRU — більш економна альтернатива.
4. Двонаправлені RNN використовують контекст з обох сторін, глибокі RNN будують ієрархію часових абстракцій.
5. Спектр застосувань величезний: від мовного моделювання та перекладу до розпізнавання мовлення та генерації музики.

Глава 8. Трансформери: революція в обробці послідовностей покаже, як відмова від рекурентності на користь механізму уваги дозволив подолати обмеження RNN і створити архітектуру, що лежить в основі сучасних великих мовних моделей.

Глава 8. Трансформери: революція в обробці послідовностей

«Увага — це все, що вам потрібно. Відмова від рекурентності на користь прямих зв'язків між усіма елементами послідовності дозволила моделювати контексти небаченої раніше довжини і створити архітектуру, що стала основою для великого мовного моделювання.»

8.1. Механізм уваги (attention): ідея та реалізація

Перш ніж говорити про трансформери, потрібно зрозуміти ключову ідею, яка зробила їх можливими — механізм уваги (attention mechanism).

8.1.1. Проблема вузького горлечка в seq2seq

В архітектурах кодер-декодер на основі RNN (див. Главу 7) весь вихідний текст стискався в один вектор фіксованого розміру — останній прихований стан кодера. Це створювало «вузьке горлечко»: для довгих речень інформація втрачалася, якість перекладу падала.

8.1.2. Ідея уваги

У 2014 році (Баджинау, Чо та Бенжіо) запропонували механізм уваги: на кожному кроці декодер може «зазирати» у всі приховані стани кодера і вибирати, яка частина вихідного тексту зараз найбільш важлива.

Інтуїція: Людина, яка перекладає речення, теж не тримає в голові все одразу. Вона дивиться на потрібні слова оригіналу, коли генерує переклад.

8.1.3. Математика уваги (загальна форма)

Загальна формула механізму уваги:

$$\text{Attention}(Q, K, V) = \text{softmax}(f(Q, K)) \cdot V$$

де:

· Q (Query — запит): Що ми шукаємо? У контексті перекладу — поточний стан декодера.
· K (Keys — ключі): На що ми дивимося? У контексті перекладу — всі приховані стани кодера.
· V (Values — значення): Що ми витягуємо? Зазвичай теж приховані стани кодера (або їх лінійна проєкція).
· f(Q, K): Функція сумісності (alignment score), яка показує, наскільки добре запит Q відповідає ключу K.

Варіанти f(Q, K):

· Скалярний добуток: $f(Q, K) = Q^T K$
· Масштабований скалярний добуток: $f(Q, K) = (Q^T K) / \sqrt{d_k}$
· Адитивна увага (з малою мережею): $f(Q, K) = v^T \tanh(W_1 Q + W_2 K)$

8.2. Само-увага та багатоголова увага

Трансформер вводить дві ключові інновації: само-увагу (self-attention) та багатоголову увагу (multi-head attention).

8.2.1. Само-увага (Self-Attention)

У само-увазі запити, ключі та значення беруться з одного й того самого джерела — з попереднього шару самої мережі. Це дозволяє кожному елементу послідовності «спілкуватися» з усіма іншими елементами.

Для речення з n слів ми отримуємо матрицю уваги n × n, де елемент (i, j) показує, наскільки слово i «звертає увагу» на слово j.

Математика само-уваги:

Для вхідної послідовності $X \in \mathbb{R}^{n \times d}$ (n токенів, кожен розмірності d) ми обчислюємо три матриці:

$Q = X \cdot W\_Q$
$K = X \cdot W\_K$
$V = X \cdot W\_V$

де $W\_Q, W\_K, W\_V \in \mathbb{R}^{d \times d_k}$ — навчальні матриці проєкцій.

Потім обчислюємо вихід само-уваги:

$$\text{Attention}(Q, K, V) = \text{softmax}((Q \cdot K^T) / \sqrt{d_k}) \cdot V$$

Масштабування $1/\sqrt{d_k}$: Необхідне, щоб при великих розмірностях d_k скалярні добутки не ставали занадто великими, відправляючи softmax в області з дуже маленькими градієнтами.

8.2.2. Багатоголова увага (Multi-Head Attention)

Замість одного механізму уваги трансформер використовує кілька «головок» уваги паралельно. Кожна головка вчиться звертати увагу на різні типи залежностей.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h) \cdot W\_O$$

де $\text{head}_i = \text{Attention}(Q \cdot W\_Q^i, K \cdot W\_K^i, V \cdot W\_V^i)$

Переваги:

· Різні головки можуть спеціалізуватися: одна на синтаксичні залежності, інша на семантичні, третя на анафору (займенники).
· Збільшується виражальна здатність моделі.

8.2.3. Позиційні кодування

У само-уваги є фундаментальна проблема: вона не враховує порядок слів. Для моделі «кіт з'їв мишу» і «миша з'їла кота» матриці уваги будуть однаковими, якщо не додати інформацію про позицію.

Рішення: Додати до вхідних ембеддингів позиційні кодування (positional encodings).

В оригінальному трансформері використовуються синусоїдальні функції:

PE(pos, 2i) = sin(pos / 10000^(2i/d_model))
PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

де pos — позиція токена, i — індекс розмірності.

Властивості:

· Значення знаходяться в [-1, 1], стабільні для навчання.
· Для будь-якого фіксованого зсуву k, PE(pos+k) можна виразити як лінійну функцію від PE(pos) — це дозволяє моделі легко вчитися враховувати відносні позиції.

У сучасних моделях часто використовують навчальні позиційні ембеддинги (кожній позиції зіставляється навчальний вектор).

8.3. Архітектура трансформера (кодер-декодер)

Оригінальний трансформер (Vaswani et al., 2017, «Attention Is All You Need») мав архітектуру кодер-декодер, призначену для машинного перекладу.

8.3.1. Загальна структура

Кодер:

· Складається з N ідентичних шарів (зазвичай N = 6 або 12).
· Кожен шар містить:
  1. Багатоголову само-увагу
  2. Позиційну повнозв'язну мережу (Feed-Forward Network, FFN) — два лінійних перетворення з нелінійністю ReLU між ними.
· Після кожного блоку — залишковий зв'язок (residual connection) та шар нормалізації (Layer Norm): LayerNorm(x + Sublayer(x))

Декодер:

· Також з N шарів.
· Кожен шар містить:
  1. Масковану багатоголову само-увагу — щоб декодер не «підглядав» у майбутні слова при генерації (маска забороняє звертати увагу на позиції > поточної).
  2. Багатоголову увагу кодер-декодер — запити йдуть з декодера, ключі та значення — з виходу кодера.
  3. Позиційну повнозв'язну мережу (FFN).
· Ті ж залишкові зв'язки та нормалізація.

8.3.2. Позиційна повнозв'язна мережа (FFN)

Це простий двошаровий перцептрон, який застосовується однаково до кожної позиції окремо:

$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$

Розмірність внутрішнього шару зазвичай в 4 рази більша за d_model (наприклад, 2048 при d_model=512).

8.3.3. Залишкові зв'язки та нормалізація

Залишкові зв'язки критично важливі для навчання глибоких трансформерів — вони дозволяють градієнтам текти напряму через шари.

Pre-Norm vs Post-Norm:

· Post-Norm (оригінал): Нормалізація після додавання з залишком: LayerNorm(x + Sublayer(x))
· Pre-Norm (сучасні моделі): Нормалізація перед підшаром: x + Sublayer(LayerNorm(x))

Pre-Norm забезпечує більш стабільне навчання для дуже глибоких моделей.

8.4. BERT, GPT та їхні послідовники

Оригінальний трансформер був encoder-decoder. Однак виявилося, що окремо взяті кодери та декодери теж надзвичайно корисні.

8.4.1. BERT (Bidirectional Encoder Representations from Transformers) — 2018, Google

Архітектура: Тільки кодер (стек трансформерних блоків без декодера).

Ключова ідея: Попереднє навчання (pre-training) на величезному корпусі тексту з двома задачами:

1. MLM (Masked Language Model): 15% токенів маскуються, модель повинна передбачити початкові токени за контекстом з обох сторін.
2. NSP (Next Sentence Prediction): Модель вчиться визначати, чи є друге речення логічним продовженням першого.

Вхід: Токен [CLS] на початку (використовується для класифікації), токен [SEP] між реченнями.

Застосування: Після попереднього навчання BERT можна донавчити (fine-tune) під конкретні задачі, додаючи невелику «голову» класифікатора.

· Класифікація тексту (використовуємо вихід [CLS])
· Відповіді на питання (Question Answering)
· Розпізнавання іменованих сутностей (NER)

Значення: BERT встановив новий стандарт в NLP, показавши, що глибоке двонаправлене попереднє навчання дає величезний приріст якості.

8.4.2. GPT (Generative Pre-trained Transformer) — 2018, OpenAI

Архітектура: Тільки декодер (з маскованою само-увагою).

Ключова ідея: Авторегресивне мовне моделювання — передбачення наступного токена.

Попереднє навчання: Модель вчиться передбачати наступний токен у послідовності на гігантському корпусі текстів.

Версії:

· GPT-1 (2018): 117 млн параметрів.
· GPT-2 (2019): 1.5 млрд параметрів. Показала, що масштабування дає емерджентні здібності (генерація зв'язних текстів).
· GPT-3 (2020): 175 млрд параметрів. Революція в NLP. Показала здатність до few-shot навчання — розв'язання задач без донавчання, тільки за кількома прикладами в промпті.
· GPT-4 (2023): Мультимодальна модель (працює з текстом і зображеннями). Точна кількість параметрів не розкривається.

Значення: GPT показала, що масштабування мовних моделей веде до появи нових здібностей (емерджентність). Запустила еру великих мовних моделей (LLM).

8.4.3. Інші важливі моделі

· RoBERTa (2019): Покращена версія BERT (більше даних, довше навчання, видалена задача NSP).
· XLNet (2019): Поєднує ідеї авторегресії та двонаправленості (permutation language modeling).
· T5 (Text-to-Text Transfer Transformer, 2019): Google, encoder-decoder архітектура, де всі задачі формулюються як «текст на вході — текст на виході».
· ELECTRA (2020): Більш ефективне попереднє навчання (задача replaced token detection).
· DeBERTa (2020): Покращена архітектура з розділеною увагою.
· LaMDA (2021): Google, модель для діалогу.
· PaLM (2022): Google, 540 млрд параметрів, з архітектурою SwiGLU.
· LLaMA (2023): Meta, відкрита модель, що показала можливість досягти якості GPT-3 з меншою кількістю параметрів при кращому навчанні.
· Mistral (2023): Ефективна відкрита модель з архітектурою sliding window attention.

8.4.4. Масштабування та закони масштабування

Дослідження OpenAI та DeepMind виявили закони масштабування (scaling laws):

· Продуктивність моделі (loss) покращується степеневим чином зі зростанням кількості параметрів, обсягу даних та обчислювальних ресурсів.
· Для оптимального використання ресурсів потрібно масштабувати всі три фактори одночасно.

Це призвело до гонки за дедалі більшими моделями, яка триває досі.

8.5. Трансформери в комп'ютерному зорі (ViT)

Успіх трансформерів в NLP надихнув дослідників застосувати їх до комп'ютерного зору.

8.5.1. ViT (Vision Transformer) — 2020, Google

Ідея: Розбити зображення на патчі фіксованого розміру (наприклад, 16×16 пікселів), лінеаризувати їх і подати як послідовність токенів у трансформер.

Архітектура:

· Зображення H×W×C розбивається на N патчів P×P.
· Кожен патч проєктується в ембеддинг (лінійний шар).
· Додаються позиційні ембеддинги.
· Додається спеціальний токен [CLS] (як у BERT) для класифікації.
· Вся послідовність подається в стандартний трансформер (кодер).
· Вихід [CLS] токена йде в класифікатор.

Результат: ViT показав, що за достатньої кількості даних (ImageNet-21k або JFT-300M) трансформер перевершує найкращі CNN.

Проблема: Трансформери менш ефективні при малих даних, оскільки у них немає вбудованої індуктивної вподобаності (локальність, трансляційна інваріантність), як у CNN.

8.5.2. Гібридні підходи та покращення

· DeiT (Data-efficient Image Transformers): Використання дистиляції знань від CNN для навчання на менших даних.
· Swin Transformer: Ієрархічний трансформер з локальною увагою у вікнах та механізмом зсуву вікон для cross-window з'єднань.
· CvT (Convolutional Vision Transformer): Вбудовування згорткових шарів у трансформер для кращого витягування низькорівневих ознак.

8.5.3. Застосування в робототехніці

Трансформери починають активно застосовуватися в робототехніці:

· RT-1 (Robotic Transformer, Google): Модель для керування роботом, навчена на величезному датасеті траєкторій.
· RT-2 (2023): Відео-мовна модель, яка може генерувати команди для робота на основі текстових інструкцій та візуальної сцени.
· Perceiver, Perceiver IO: Архітектури, які можуть обробляти довільні модальності (зображення, звук, текст) за допомогою крос-уваги.

Підсумок Глави 8

Ми поринули в архітектуру, яка визначає сучасний стан ШІ:

1. Механізм уваги дозволяє моделі динамічно фокусуватися на релевантних частинах вхідних даних.
2. Само-увага створює прямі зв'язки між усіма елементами послідовності, долаючи обмеження RNN.
3. Багатоголова увага дозволяє моделі враховувати різні типи залежностей.
4. Архітектура трансформера (кодер-декодер) стала основою для машинного перекладу та багатьох інших задач.
5. BERT (кодер) та GPT (декодер) стали двома основними гілками розвитку, які породили цілі сімейства моделей.
6. ViT показав, що трансформери можуть перевершувати CNN у комп'ютерному зорі за достатньої кількості даних.
7. Трансформери активно проникають у робототехніку, стаючи основою для систем керування та планування.

Глава 9. Генеративні моделі покаже, як нейромережі навчилися творити — створювати нові зображення, тексти, музику, яких не було в навчальних даних.

Глава 9. Генеративні моделі

«Творчість — це не магія. Це здатність вловити приховані закономірності в даних і використовувати їх для створення нових комбінацій, яких не було в навчальній вибірці. Генеративні моделі — це перший крок машин до уяви.»

9.1. Автоенкодери та варіаційні автоенкодери (VAE)

Автоенкодери — це найпростіші генеративні моделі, які вчаться стискати дані, а потім відновлювати їх.

9.1.1. Класичний автоенкодер

Ідея: Мережа вчиться копіювати вхід на вихід через «вузьке горлечко» — прихований простір меншої розмірності.

Архітектура:

· Кодувальник (encoder): $f: X \to Z$, стискає вхідні дані x в латентне представлення z (код).
· Декодувальник (decoder): $g: Z \to X$, відновлює дані $\hat{x}$ з коду z.

Навчання: Мінімізація помилки реконструкції (reconstruction loss):

$L = \| x - g(f(x)) \|^2$ (для неперервних даних) або бінарна крос-ентропія (для бінарних).

Застосування:

· Зменшення розмірності (альтернатива PCA).
· Шумопригнічення (denoising autoencoder — на вхід подається зашумлена версія, на виході — чиста).
· Попереднє навчання шарів (зараз використовується рідко).

Проблема: Класичний автоенкодер не є генеративною моделлю в повному сенсі. Він може стиснути і відновити дані, але не може згенерувати нові осмислені приклади, тому що латентний простір не регулярний — у ньому є «дірки».

9.1.2. Варіаційний автоенкодер (VAE) — 2013, Кінгма та Веллінг

VAE вирішує проблему нерегулярності латентного простору, змушуючи кодувальник видавати не точку, а розподіл.

Ключова ідея: Латентний простір повинен бути неперервним і повним. Кожна точка в латентному просторі повинна декодуватися в осмислений вихід.

Архітектура VAE:

1. Кодувальник видає параметри розподілу: середнє $\mu(x)$ та логарифм дисперсії $\log \sigma^2(x)$ (або $\sigma(x)$).
2. З цього розподілу семплюється латентний вектор z: $z \sim N(\mu(x), \sigma^2(x)I)$
3. Декодувальник відновлює $\hat{x}$ з z.

Проблема семплювання: Операція семплювання не диференційовна. Рішення — трюк з репараметризацією (reparameterization trick):

$z = \mu(x) + \sigma(x) \odot \varepsilon$, де $\varepsilon \sim N(0, I)$

Тепер градієнт може проходити через μ та σ, а випадковість винесена в незалежний шум ε.

Функція втрат VAE:

$L = L\_reconstruction + \beta \cdot L\_KL$

де:

· L_reconstruction: помилка відновлення (як у класичному автоенкодері).
· L_KL (KL-дивергенція): штраф за відхилення розподілу кодувальника від стандартного нормального розподілу N(0, I).

$L\_KL = D\_KL(N(\mu, \sigma^2) \| N(0, 1)) = -\frac{1}{2} \sum (1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2)$ (підсумовування за j від 1 до dim(z))

Інтуїція: KL-дивергенція змушує латентний простір бути гладким і неперервним. Точки, близькі в латентному просторі, декодуються в схожі виходи.

Генерація нових даних:

1. Семплюємо $z \sim N(0, I)$ (стандартний нормальний шум).
2. Подаємо z в декодер.
3. Отримуємо новий, раніше не бачений приклад.

Достоїнства VAE:

· Швидка генерація (один прохід через декодер).
· Неперервний латентний простір з хорошою структурою.
· Інтерполяція між точками дає плавні переходи.

Недоліки:

· Генерує розмиті зображення (через усереднення за розподілом).
· Тенденція до ігнорування деяких латентних розмірностей (posterior collapse).

9.1.3. Покращені VAE

· β-VAE: Збільшення ваги β для L_KL, що призводить до більш disentangled представлень (різні латентні розмірності кодують незалежні фактори варіації).
· VQ-VAE (Vector Quantised VAE): Використовує дискретне латентне простір (словник кодів). Ліг в основу багатьох сучасних генеративних моделей (VQ-GAN, DALL-E).

· NVAE (Nouveau VAE): Глибокий ієрархічний VAE зі складною архітектурою, що досягає якості, близької до GAN.

9.2. Генеративно-змагальні мережі (GAN)

GAN (Generative Adversarial Networks), запропоновані Ієном Ґудфеллоу в 2014 році, здійснили революцію в генерації зображень.

9.2.1. Ідея змагання

У GAN змагаються дві мережі:

· Генератор (Generator, G): Намагається створити реалістичні підробки. На вхід отримує випадковий шум z, на виході — зображення G(z).
· Дискримінатор (Discriminator, D): Намагається відрізнити реальні зображення з навчального набору від підробок, створених генератором. Це бінарний класифікатор.

Аналогія: Генератор — фальшивомонетник, дискримінатор — поліцейський, який перевіряє купюри. Фальшивомонетник вчиться робити дедалі якісніші підробки, поліцейський вчиться їх краще розпізнавати.

9.2.2. Мінімаксна гра

Навчання GAN формулюється як мінімаксна гра двох гравців:

$$\min_G \max_D V(D, G) = E_{x \sim p_{data}} [\log D(x)] + E_{z \sim p_z} [\log(1 - D(G(z)))]$$

· Дискримінатор хоче максимізувати V: правильно класифікувати реальні ($\log D(x) \to 1$) і підробки ($\log(1 - D(G(z))) \to 1$).
· Генератор хоче мінімізувати V: обманути дискримінатор, щоб той вважав підробки реальними ($\log(1 - D(G(z))) \to 0$).

На практиці генератор часто навчають максимізувати $\log D(G(z))$ замість мінімізації $\log(1 - D(G(z)))$ — це дає сильніші градієнти на ранніх етапах (non-saturating loss).

9.2.3. Проблеми навчання GAN

Навчання GAN сумно відоме своєю нестабільністю:

1. Незбіжність (non-convergence): Може не досягнути рівноваги Неша.
2. Колапс мод (mode collapse): Генератор знаходить кілька вдалих підробок і виробляє тільки їх, ігноруючи все різноманіття даних.
3. Зникаючі градієнти: Якщо дискримінатор надто сильний, градієнти генератора стають нульовими.
4. Чутливість до гіперпараметрів: Потребує ретельного налаштування.

9.2.4. Покращені архітектури GAN

· DCGAN (Deep Convolutional GAN, 2015):
  · Використання згорткових шарів у генераторі та дискримінаторі.
  · Batch normalization в обох мережах.
  · ReLU в генераторі (крім виходу, де tanh) і LeakyReLU в дискримінаторі.

· Задали стандарт для стабільного навчання.
· WGAN (Wasserstein GAN, 2017):
· Заміна дискримінатора на критика, який оцінює відстань Вассерштейна (Earth Mover's Distance) між розподілами.
· Прибирає сигмоїду на виході критика.
· Використовує weight clipping для дотримання умови Ліпшиця (пізніше замінено на градієнтний штраф — WGAN-GP).
· Значно стабільніший і вирішує проблему колапсу мод.
· LSGAN (Least Squares GAN): Використовує квадратичну функцію втрат замість бінарної крос-ентропії, що дає більш стабільне навчання.
· StyleGAN (2018-2020, NVIDIA):
· Революційна архітектура для генерації облич.
· Вводить карту стилів (mapping network), що перетворює шум z у проміжний вектор w.
· Адаптивна нормалізація екземпляра (AdaIN) для впровадження стилю на різних рівнях.
· Випадковий шум на різних масштабах для стохастичних деталей (ластовиння, зморшки).
· StyleGAN2 та StyleGAN3 покращили якість і прибрали артефакти.
· BigGAN (2018): Масштабування GAN до величезних розмірів і батчів, генерація високоякісних зображень ImageNet.

9.2.5. Conditional GAN (cGAN)

Дозволяє контролювати генерацію, подаючи додаткову інформацію y (клас, текст, зображення) і в генератор, і в дискримінатор.

$G(z, y) \rightarrow x$
$D(x, y) \rightarrow$ ймовірність

Застосування: генерація зображень за текстовим описом, переклад зображень (pix2pix).

9.2.6. pix2pix та CycleGAN

· pix2pix (2016): Умовний GAN для перекладу зображень (image-to-image translation). Потребує парних даних (наприклад, контур будівлі → фото будівлі).
· CycleGAN (2017): Дозволяє перекладати зображення з одного домену в інший без парних даних. Використовує цикл-послідовність (cycle-consistency loss): переклад з домену A в B і назад повинен повертати вихідне зображення.

Застосування: кінь → зебра, фото → картина Ван Гога, день → ніч.

9.3. Дифузійні моделі

Дифузійні моделі (diffusion models) — це новітній клас генеративних моделей, які в 2021-2023 роках витіснили GAN як state-of-the-art для генерації зображень.

9.3.1. Ідея дифузії

Процес дифузії складається з двох процесів:

1. Прямий процес (forward diffusion): Поступове додавання гауссового шуму до даних протягом T кроків, поки дані не перетворяться на чистий шум.
2. Зворотний процес (reverse diffusion): Навчання нейромережі поступово прибирати шум, відновлюючи дані з шуму.

9.3.2. Математика дифузійних моделей

Прямий процес:

$q(x_t \mid x_{t-1}) = N(x_t; \sqrt{(1-\beta_t)}\, x_{t-1}, \beta_t I)$

де β_t — невеликий коефіцієнт (зростає з t). Завдяки властивості гауссових розподілів можна одразу отримати x_t з x_0:

$x_t = \sqrt{(\bar{\alpha}_t)}\, x_0 + \sqrt{(1-\bar{\alpha}_t)}\, \varepsilon$, де $\alpha_t = 1-\beta_t$, $\bar{\alpha}_t = \prod$(від s=1 до t) α_s, $\varepsilon \sim N(0, I)$

Зворотний процес:

$p\_\theta(x_{t-1} \mid x_t) = N(x_{t-1}; \mu\_\theta(x_t, t), \sigma^2\_t I)$

Нейромережа (зазвичай U-Net) вчиться передбачати шум $\varepsilon\_\theta(x_t, t)$, доданий на кроці t, або безпосередньо μ_θ.

Функція втрат (спрощена):

$L\_simple = E\_\{t, x_0, \varepsilon\} [\| \varepsilon - \varepsilon\_\theta(\sqrt{(\bar{\alpha}_t)}\, x_0 + \sqrt{(1-\bar{\alpha}_t)}\, \varepsilon, t) \|^2]$

9.3.3. DDPM (Denoising Diffusion Probabilistic Models) — 2020, Ho et al.

Перша робота, що показала можливість дифузійних моделей генерувати зображення високої якості (на рівні GAN).

Процес генерації:

1. Семплюємо чистий шум $x\_T \sim N(0, I)$.
2. T кроків ітеративно прибираємо шум, використовуючи навчену мережу.

Недолік: Дуже повільна генерація (потрібно T кроків, часто T=1000).

9.3.4. DDIM (Denoising Diffusion Implicit Models) — 2020

Прискорює генерацію, роблячи процес детермінованим і дозволяючи пропускати кроки (можна генерувати за 50-100 кроків замість 1000).

9.3.5. Стабільна дифузія (Stable Diffusion) — 2022, Stability AI

Революційна модель, що зробила дифузію доступною для всіх.

Ключові інновації:

1. Латентна дифузія (Latent Diffusion): Дифузія відбувається не в просторі пікселів, а в стиснутому латентному просторі автоенкодера (VQ-VAE). Це радикально прискорює і здешевлює процес.
2. Крос-увага (cross-attention): Можливість керування генерацією текстом (text conditioning). Текст кодується CLIP-трансформером і через механізм крос-уваги впливає на процес дифузії.

3. Відкритість: Модель та ваги були відкриті, що породило величезну спільноту та незліченні варіації (LoRA, ControlNet, Dreambooth).

Архітектура:

· Текстовий енкодер: CLIP (або інший трансформер).
· Аутоенкодер: VQ-VAE, що стискає зображення 512×512 у латентний простір 64×64×4.
· U-Net з крос-увагою: Виконує дифузію в латентному просторі, приймаючи на вхід текст через крос-увагу.

9.3.6. DALL-E 2, Imagen, Midjourney

· DALL-E 2 (OpenAI, 2022): Використовує CLIP для ембеддингів зображень і тексту, потім дифузійна модель (unCLIP) генерує зображення.
· Imagen (Google, 2022): Використовує великий текстовий енкодер T5-XXL і каскад дифузійних моделей (спочатку 64×64, потім апскейлінг до 256×256 і 1024×1024).
· Midjourney: Комерційна модель, деталі архітектури не розкриваються, але ймовірно використовує дифузію.

9.3.7. Порівняння дифузійних моделей та GAN

Характеристика GAN Дифузійні моделі
Якість зображень Висока Дуже висока (часто краща за GAN)
Різноманітність Схильні до колапсу мод Високе різноманіття
Швидкість генерації Дуже швидка (один прохід) Повільна (багато кроків)
Стабільність навчання Нестабільне Стабільне
Контрольованість Потребує ускладнень Легко додається через крос-увагу

9.4. Моделі з нормалізуючими потоками (normalizing flows)

Normalizing flows — це клас генеративних моделей, які будують оборотне перетворення від простого розподілу (шуму) до складного (даних).

9.4.1. Ідея normalizing flows

Нехай $z \sim p_z(z)$ — простий розподіл (наприклад, стандартний нормальний). Ми хочемо знайти оборотну функцію f (з оберненою $g = f^{-1}$), таку що:

$x = f(z)$ має складний розподіл $p_x(x)$, близький до розподілу даних.

Ключова властивість: Завдяки оборотності можна точно обчислити ймовірність даних:

$$p_x(x) = p_z(f^{-1}(x)) \cdot |\det J_{f^{-1}}(x)|$$

де $|\det J|$ — абсолютне значення визначника Якобіана перетворення (множник, що враховує зміну об'єму при перетворенні).

9.4.2. Вимоги до перетворення

Щоб функція f була корисною:

1. Оборотність: Повинна існувати ефективна обернена функція.

2. Простий Якобіан: Визначник Якобіана повинен легко обчислюватися (щоб рахувати ймовірність).
3. Виражальність: Композиція простих перетворень може моделювати складні розподіли.

9.4.3. Приклади архітектур

· NICE (Non-linear Independent Components Estimation, 2014): Використовує адитивні перетворення з трикутним Якобіаном.
· RealNVP (2016): Використовує афінні перетворення (масштабування та зсув) з маскуванням, що розділяє вхідні розмірності.
· Glow (2018, OpenAI): Покращена версія з 1×1 згортками для перемішування каналів.

9.4.4. Застосування

· Генерація зображень (меншого розділення, ніж GAN/дифузія).
· Оцінка щільності ймовірності (можемо точно сказати, наскільки ймовірний приклад).
· Стиснення даних.
· Покращення зображень (super-resolution).

Достоїнства: Точне обчислення ймовірності, швидка генерація (один прохід), оборотність.

Недоліки: Складність архітектури, обмеження на розмір зображень.

9.5. Оцінка якості генерації: метрики та методи

Як виміряти якість згенерованих даних? Це нетривіальна задача, особливо для зображень.

9.5.1. Inception Score (IS)

Ідея: Використовувати попередньо навчений класифікатор (Inception-v3) для оцінки:

1. Якість: Класифікатор повинен бути впевненим у своїх передбаченнях (p(y|x) з низькою ентропією).
2. Різноманітність: Розподіл передбачених класів по всіх згенерованих зображеннях повинен бути рівномірним (висока ентропія).

$$IS = \exp(E_x [D_{KL}(p(y|x) \| p(y))])$$

Проблеми: Не враховує схожість з реальними даними, чутливий до попередньо навченої моделі.

9.5.2. FID (Fréchet Inception Distance)

Ідея: Порівняти статистику (середнє та коваріацію) ознак, витягнутих з реальних та згенерованих зображень за допомогою Inception-v3.

$$FID = \| \mu_r - \mu_g \|^2 + Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{(1/2)})$$

Чим менше FID, тим краще. FID краще корелює з людським сприйняттям, ніж IS.

9.5.3. Інші метрики

· Precision та Recall для генеративних моделей: Оцінюють якість (precision — частка реалістичних зображень) та різноманітність (recall — покриття різноманіття даних).
· LPIPS (Learned Perceptual Image Patch Similarity): Метрика перцептуальної схожості, заснована на ознаках глибокої мережі.
· CLIP score: Для text-to-image моделей — вимірює схожість між ембеддингами тексту та зображення в просторі CLIP.
· Human evaluation: Найнадійніший, але найдорожчий спосіб.

Підсумок Глави 9

Ми здійснили подорож світом генеративних моделей:

1. VAE створюють гладкий латентний простір, але дають розмиті зображення.
2. GAN через змагання двох мереж навчилися генерувати фотореалістичні зображення, але складні в навчанні.
3. Дифузійні моделі стали новим стандартом якості, особливо після Stable Diffusion, але повільні в генерації.
4. Normalizing flows пропонують точне обчислення ймовірності, але обмежені в масштабі.
5. Метрики оцінки (FID, IS, precision/recall) дозволяють порівнювати моделі.

Генеративні моделі — це не тільки мистецтво. Це інструмент для:

· Створення синтетичних даних для навчання (особливо в медицині, де дані рідкісні).
· Дизайну та творчості.
· Посилення людської уяви.

Глава 10. Графові нейронні мережі (GNN) покаже, як працювати з даними, де важливі не тільки самі об'єкти, але й зв'язки між ними — від соціальних мереж до молекул та логістики.

Глава 10. Графові нейронні мережі (Graph Neural Networks, GNN)

«Світ — це не просто множина об'єктів. Світ — це мережа зв'язків між ними. Атоми з'єднуються в молекули, люди — в спільноти, документи — в цитування. Графові нейронні мережі вчаться бачити структуру цих зв'язків і витягувати сенс із топології.»

10.1. Чому графи? Дані зі складною структурою зв'язків

Багато даних за своєю природою є графами — вони складаються з вузлів (вершин) та ребер (зв'язків між ними).

10.1.1. Приклади графових даних

· Соціальні мережі:
  · Вузли: користувачі.
  · Ребра: дружба, підписки, лайки, репости.
  · Задача: передбачити зв'язки (рекомендація друзів), класифікувати спільноти, виявити впливових користувачів.
· Хімія та біологія:
  · Вузли: атоми.

· Ребра: хімічні зв'язки.
· Задача: передбачити властивості молекул (токсичність, розчинність), спроєктувати нові ліки.
· Фізика:
· Вузли: частинки.
· Ребра: взаємодії (сили, зіткнення).
· Задача: моделювання динаміки систем (наприклад, симуляція рідин).
· Транспорт і логістика:
· Вузли: перехрестя, міста.
· Ребра: дороги, маршрути.
· Задача: оптимізація потоків, передбачення часу в дорозі.
· Рекомендаційні системи:
· Вузли: користувачі та товари.
· Ребра: покупки, перегляди, оцінки.
· Задача: рекомендувати товари, які сподобаються користувачеві.
· Цитування наукових статей:
· Вузли: статті.
· Ребра: цитування.
· Задача: класифікація статей за тематикою, передбачення впливовості.
· Інтернет:
· Вузли: веб-сторінки.
· Ребра: гіперпосилання.
· Задача: ранжування сторінок (PageRank), пошук інформації.
· Код програм:
· Вузли: оператори, змінні.
· Ребра: потоки даних, виклики функцій.
· Задача: аналіз вразливостей, генерація коду.

10.1.2. Чому не працюють стандартні архітектури?

· CNN потребують регулярної сітки (grid-like structure). У графах вузли можуть мати різну кількість сусідів, і немає природного порядку.
· RNN передбачають послідовність. Графи можуть мати довільну топологію (цикли, розгалуження).
· MLP можуть обробляти ознаки вузлів окремо, але ігнорують інформацію про зв'язки, яка часто критично важлива.

Потрібна архітектура, яка:

1. Враховує як ознаки вузлів, так і структуру графа.
2. Інваріантна до перестановок вузлів (permutation invariance).
3. Може працювати з графами різного розміру та топології.

10.2. Згортки на графах

Ключова ідея GNN — поширювати інформацію по графу, агрегуючи ознаки сусідніх вузлів. Це схоже на згортку, де рецептивне поле визначається не прямокутною областю, а структурою графа.

10.2.1. Основна парадигма: агрегація сусідів

На кожному шарі GNN кожен вузол оновлює своє представлення, агрегуючи інформацію від своїх сусідів:

h_v^(k) = COMBINE(h_v^(k-1), AGGREGATE({ h_u^(k-1) for u ∈ N(v) }))

де:

· h_v^(k) — представлення вузла v на шарі k.
· N(v) — множина сусідів вузла v.
· AGGREGATE — функція, що об'єднує інформацію від сусідів (сума, середнє, максимум, більш складні).
· COMBINE — функція, що об'єднує поточне представлення вузла з агрегованою інформацією від сусідів (часто конкатенація + лінійний шар + нелінійність).

10.2.2. Інваріантність до перестановок

Найважливіша властивість GNN: результат не повинен залежати від порядку перерахування сусідів. Тому AGGREGATE повинна бути симетричною функцією — інваріантною до перестановок. Підходять: сума, середнє, максимум.

10.2.3. Трансляційна інваріантність vs інваріантність до перестановок

· CNN мають трансляційну інваріантність — вони однаково реагують на патерн незалежно від його положення на сітці.
· GNN мають інваріантність до перестановок вузлів — якщо перенумерувати вузли, результат не зміниться. Це відповідає природі графів: у графі немає природного порядку вузлів.

10.3. Архітектури: GCN, GraphSAGE, GAT

10.3.1. Graph Convolutional Networks (GCN) — 2016, Кіпф та Веллінг

GCN — одна з перших і найвпливовіших архітектур. Вона спрощує агрегацію до зваженого середнього по сусідах з урахуванням степенів вузлів.

Правило оновлення GCN:

h_v^(k) = ReLU(W^(k) · ∑ (для u ∈ N(v) ∪ {v}) (h_u^(k-1) / √(deg(v)·deg(u))))

де deg(v) — степінь вузла v (кількість сусідів). Нормалізація за степенями запобігає вибуху значень для вузлів з високим степенем.

У матричній формі для всього графа:

H^(k) = ReLU(Â H^(k-1) W^(k))

де Â — нормалізована матриця суміжності з доданими петлями (self-loops): Â = D̃^(-1/2) Ã D̃^(-1/2), де Ã = A + I, D̃ — діагональна матриця степенів Ã.

Достоїнства: Простота, ефективність.
Недоліки: Трансдуктивність — не може працювати з новими вузлами, не баченими при навчанні (якщо не перераховувати граф цілком).

10.3.2. GraphSAGE (Graph SAmple and aggreGatE) — 2017, Hamilton, Ying, Leskovec

GraphSAGE вирішує проблему трансдуктивності, навчаючи функцію агрегації, яка може застосовуватися до нових вузлів.

Ключові ідеї:

1. Семплювання сусідів: Замість використання всіх сусідів (що може бути дорого для вузлів з високим степенем), семплюється фіксована кількість сусідів.
2. Гібридні агрегатори: Декілька варіантів агрегації:
   · Mean aggregator: Середнє по сусідах (як у GCN).
   · LSTM aggregator: Застосування LSTM до перемішаних сусідів (потребує порядку, тому сусіди випадково переставляються).
   · Pooling aggregator: Застосування MLP до кожного сусіда і потім поелементний max/sum.

Правило оновлення GraphSAGE:

$h_{N(v)}^{(k)} = AGGREGATE_k(\{ h_u^{(k-1)} \text{ for } u \in N(v) \})$

$h_v^{(k)} = \sigma(W^{(k)} \cdot CONCAT(h_v^{(k-1)}, h_{N(v)}^{(k)}))$

$h_v^{(k)} = h_v^{(k)} / \|h_v^{(k)}\|_2$ (нормалізація)

Достоїнства: Індуктивність (робота з новими вузлами), масштабованість (семплювання).

10.3.3. Graph Attention Networks (GAT) — 2017, Veličković et al.

GAT вводить механізм уваги в графи: кожен вузол може по-різному «звертати увагу» на своїх сусідів.

Ключова ідея: Використовувати само-увагу (self-attention) для обчислення ваг сусідів.

Механізм уваги:

$e_{ij} = a(W h_i, W h_j)$

де a — функція, що обчислює важливість сусіда j для вузла i (зазвичай невелика нейромережа з LeakyReLU). Потім ваги нормалізуються softmax по всіх сусідах:

$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \exp(e_{ij}) / \sum_{k \in N(i)} \exp(e_{ik})$

Нове представлення вузла — зважена сума сусідів:

$h_i' = \sigma(\sum_{j \in N(i)} \alpha_{ij} W h_j)$

Багатоголова увага (multi-head attention): Як у трансформерах, кілька незалежних головок уваги конкатенуються (або усереднюються).

Достоїнства:

· Різні ваги для різних сусідів (більш виразно, ніж GCN).

· Індуктивність.
· Інтерпретованість (можна візуалізувати ваги уваги).

10.3.4. Інші важливі архітектури

· GIN (Graph Isomorphism Network, 2018): Теоретично обґрунтована архітектура, максимально виразна у розрізненні не-ізоморфних графів (настільки ж потужна, як WL-тест).
· MPNN (Message Passing Neural Network, 2017): Загальна рамка для багатьох GNN, яка уніфікує різні підходи як «передачу повідомлень» між вузлами.
· RGCN (Relational Graph Convolutional Networks): Для графів з різними типами ребер (наприклад, у базах знань: «народився_в», «працює_в»).
· GNN з енкодером-декодером: Для задач, де потрібно передбачати зв'язки або генерувати графи.

10.4. Застосування: соціальні мережі, молекули, логістика

10.4.1. Соціальні мережі

Задачі:

· Класифікація вузлів: Визначити інтереси користувача, виявити ботів, передбачити стать/вік.
· Класифікація зв'язків: Передбачити, чи виникне дружба (link prediction), виявити приховані зв'язки.
· Кластеризація спільнот: Знайти групи тісно пов'язаних користувачів.
· Ранжування: Знайти впливових користувачів (аналог PageRank).

Приклад: Pinterest використовує GNN для рекомендації пінів користувачам на основі графа взаємодій.

10.4.2. Хімія та біоінформатика

Задачі:

· Передбачення властивостей молекул: Токсичність, розчинність, енергія зв'язування з білком.
· Генерація нових молекул: Створення молекул із заданими властивостями (drug discovery).
· Передбачення взаємодій білок-білок.
· Класифікація хімічних реакцій.

Приклад: DeepMind використовувала GNN для передбачення властивостей молекул у рамках конкурсу "MoleculeNet". AlphaFold також використовує графові представлення для передбачення структури білків (хоча там складніша архітектура).

10.4.3. Логістика та транспорт

Задачі:

· Оптимізація маршрутів: Знаходження найкоротших шляхів з урахуванням поточного завантаження доріг.

· Передбачення часу в дорозі: Врахування структури дорожньої мережі та історичних даних.
· Управління трафіком: Передбачення заторів, адаптивне управління світлофорами.
· Планування доставки: Оптимізація розподілу вантажів.

10.4.4. Рекомендаційні системи

Ідея: Будуємо граф, де вузли — користувачі та товари, а ребра — взаємодії (покупки, перегляди, лайки). GNN вчиться поширювати інформацію по графу:

· Від користувача до схожих користувачів через спільні товари.
· Від товару до схожих товарів через спільних користувачів.

Приклад: Pinterest (PinSAGE), Uber Eats, Alibaba використовують GNN для рекомендацій.

10.4.5. Фізика та моделювання

Задачі:

· Моделювання динаміки систем частинок: Передбачення траєкторій у рідинах, газах, плазмі.
· Open Graph Benchmark (OGB): Є задачі по передбаченню траєкторій у гравітаційних системах.

Приклад: Моделювання поведінки піску, рідин, деформації матеріалів.

10.4.6. Охорона здоров'я

Задачі:

· Аналіз медичних графів знань: Пошук зв'язків між симптомами, захворюваннями, ліками.
· Передбачення побічних ефектів ліків.
· Аналіз шляхів пацієнтів у системі охорони здоров'я.

10.4.7. Графи знань (Knowledge Graphs)

Графи знань — це величезні бази даних фактів у вигляді триплетів (суб'єкт, відношення, об'єкт). Приклади: Wikidata, Freebase, Google Knowledge Graph.

Задачі:

· Завершення графа знань (knowledge graph completion): Передбачення відсутніх зв'язків.
· Відповіді на питання за графом знань.
· Витягування інформації з тексту в структуру графа.

Моделі: RGCN, CompGCN, різні embedding-методи (TransE, DistMult) часто комбінуються з GNN.

10.4.8. Комп'ютерний зір

Хоча зображення — це сітки, їх можна розглядати як графи:

· 3D хмари точок (point clouds): Вузли — точки, ребра — найближчі сусіди. GNN застосовуються для сегментації та класифікації.
· Сцени як графи об'єктів: Вузли — об'єкти, ребра — відношення («над», «поруч», «частина»). Це допомагає розуміти сцени та взаємодії.

Підсумок Глави 10

Ми поринули у світ графових нейронних мереж:

1. Графи — природне представлення для багатьох типів даних (соціальні мережі, молекули, логістика).
2. Основна парадигма GNN — агрегація інформації від сусідів з інваріантністю до перестановок.
3. Ключові архітектури:
   · GCN — проста та ефективна спектральна згортка.
   · GraphSAGE — індуктивне навчання з семплюванням сусідів.
   · GAT — механізм уваги для зважування сусідів.
4. Застосування охоплюють практично всі галузі науки та індустрії: від пошуку ліків до рекомендацій у Pinterest, від логістики до аналізу соціальних мереж.

Глава 11. Нейроморфні та спайкові мережі (SNN) поверне нас до витоків — до біологічного натхнення. Ми побачимо, як можна наблизитися до енергоефективності мозку, використовуючи імпульсний характер передачі сигналів.

Глава 11. Нейроморфні та спайкові мережі (Spiking Neural Networks, SNN)

«Мозок працює не на тактовій частоті та безперервних сигналах. Його мова — це рідкісні, дискретні імпульси в часі. Спайкові нейронні мережі — це спроба почути цю мову і заговорити нею.»

11.1. Біологічний нейрон: спайки, потенціали дії

Перш ніж говорити про штучні спайкові мережі, потрібно зрозуміти, як працює біологічний нейрон. Його динаміка принципово відрізняється від штучних нейронів, які ми розглядали в попередніх главах.

11.1.1. Мембранний потенціал і потенціал дії

У стані спокою внутрішня частина нейрона має від'ємний потенціал відносно зовнішнього середовища (близько -70 мВ). Це мембранний потенціал спокою.

Коли нейрон отримує сигнали від інших нейронів через синапси, на його мембрані відбуваються зміни потенціалу:

· Збуджувальні постсинаптичні потенціали (EPSP): Підвищують потенціал (роблять менш від'ємним).
· Гальмівні постсинаптичні потенціали (IPSP): Знижують потенціал (роблять більш від'ємним).

Ці потенціали сумуються в часі та просторі. Якщо в результаті сумації мембранний потенціал досягає певного порогу (близько -55 мВ), відбувається потенціал дії (спайк) — різкий і швидкий викид потенціалу до +40 мВ, після чого потенціал повертається до рівня спокою (або навіть нижче, в стан рефрактерності).

Ключові характеристики:

· Спайки мають приблизно однакову форму й амплітуду. Інформація кодується не формою, а частотою та моментом виникнення спайків.
· Після спайка настає рефрактерний період (частки секунди), коли нейрон нечутливий до нових сигналів або потребує сильнішого стимулу.
· Комунікація є асинхронною — немає глобального тактового сигналу.

11.1.2. Синапси та пластичність

Нейрони з'єднані між собою синапсами. Коли спайк доходить до пресинаптичного закінчення, виділяються нейромедіатори, які впливають на постсинаптичний нейрон. Важливою властивістю синапсів є синаптична пластичність — здатність змінювати свою ефективність (вагу) залежно від активності. Це основа навчання в біологічних мережах.

11.2. Моделі спайкових нейронів

Для моделювання динаміки спайкових нейронів створено різні математичні моделі, які відрізняються за складністю та біологічною правдоподібністю.

11.2.1. Модель «інтеграція-та-спалах» (Integrate-and-Fire, IF)

Це найпростіша модель. Вона просто інтегрує вхідний струм I(t):

$C\, dV/dt = I(t)$

де C — ємність мембрани, V — мембранний потенціал.

Коли V досягає порогу V_{th}, генерується спайк, і потенціал скидається до початкового значення V_{reset}.

Недолік: Не враховує витік (leak) — природне повернення потенціалу до рівня спокою за відсутності сигналу.

11.2.2. Модель LIF (Leaky Integrate-and-Fire)

Це найпопулярніша модель завдяки балансу між простотою та біологічною правдоподібністю. Вона додає «витік»:

$C\, dV/dt = -g_L (V - E_L) + I(t)$ для $V < V_{th}$

де:

· g_L — провідність витоку,
· E_L — потенціал спокою (рівноважний потенціал).

Коли V досягає V_th, нейрон генерує спайк, і потенціал скидається до V_reset. Після цього настає рефрактерний період τ_ref, протягом якого нейрон не відповідає на стимули.

Рівняння LIF можна записати в дискретному часі для чисельного моделювання:

$V[t+1] = \beta V[t] + I[t+1]$, де β — коефіцієнт витоку (менший за 1).

11.2.3. Модель Іжикевича (Izhikevich model)

Модель, запропонована Євгеном Іжикевичем у 2003 році, поєднує біологічну правдоподібність (здатна відтворювати різноманітні типи спайкової активності реальних нейронів) та обчислювальну ефективність.

Вона описується двома диференціальними рівняннями:

$dv/dt = 0.04v^2 + 5v + 140 - u + I$
$du/dt = a(bv - u)$

з умовою скидання після спайка:

якщо $v \geq 30$ мВ, то $\{ v \leftarrow c, u \leftarrow u + d \}$

де:

· v — мембранний потенціал,
· u — змінна відновлення (враховує активацію калієвих каналів та інактивацію натрієвих),
· a, b, c, d — безрозмірні параметри, зміна яких дозволяє моделювати різні типи нейронів (регулярно спрацьовуючі, швидкі, bursting тощо).

11.2.4. Інші моделі

· Hodgkin-Huxley: Найбільш біологічно детальна модель, заснована на реальних іонних каналах. Дуже складна для обчислень, використовується в нейрофізіологічних дослідженнях.
· SRM (Spike Response Model): Модель, що описує стан нейрона як функцію від часу останнього спайка та вхідних стимулів.

11.3. Навчання в спайкових мережах: STDP та його варіанти

Навчання в SNN принципово відрізняється від зворотного поширення помилки, яке використовується в класичних штучних нейромережах. Воно натхненне біологічними механізмами.

11.3.1. STDP (Spike-Timing-Dependent Plasticity)

STDP — це біологічно правдоподібне правило навчання, яке змінює вагу синапсу залежно від відносного часу спайків пре- та постсинаптичного нейронів.

Основне правило:

· Якщо пресинаптичний нейрон (той, що передає сигнал) спрацьовує перед постсинаптичним (тобто його спайк сприяв виникненню спайка), то синапс підсилюється (potentiation). Кореляція «причина перед наслідком» зміцнює зв'язок.
· Якщо пресинаптичний нейрон спрацьовує після постсинаптичного (тобто його сигнал запізнився), то синапс послаблюється (depression).

Математично зміна ваги Δw описується функцією від різниці часів $\Delta t = t_{post} - t_{pre}$:

$\Delta w = f(\Delta t)$

Типова функція STDP має вигляд:

$f(\Delta t) = A_+ \exp(-\Delta t / \tau_+)$ для $\Delta t > 0$ (пре раніше пост) — підсилення
$f(\Delta t) = -A_- \exp(\Delta t / \tau_-)$ для $\Delta t < 0$ (пре пізніше пост) — послаблення

де A_+, A_-, τ_+, τ_- — параметри.

Властивості STDP:

· Локальне правило: для оновлення ваги потрібна інформація тільки від двох сусідніх нейронів та часу їх активації.
· Виникає змагання між синапсами: сильніші зв'язки, які систематично передують спайку, стають ще сильнішими.
· Призводить до стабілізації мережі.

11.3.2. Варіанти та узагальнення STDP

· Triplet STDP: Враховує не тільки пари спайків, але й потрійні взаємодії, що краще узгоджується з деякими експериментальними даними.
· STDP з добутвом (multiplicative STDP): Зміна ваги пропорційна поточному значенню ваги, що запобігає необмеженому зростанню.
· Remote supervised method (ReSuMe): Алгоритм, який намагається поєднати STDP з навчанням під наглядом, подаючи «цільові» спайки.

11.3.3. Проблеми навчання SNN

1. Недиференційовність: Спайк — це дискретна подія, тому стандартне зворотне поширення (яке потребує неперервних градієнтів) безпосередньо не застосовне.
2. Відсутність єдиного фреймворку: Не існує такого ж універсального та ефективного методу навчання для SNN, як backprop для ANN.
3. Повільне навчання: Симуляція SNN в часі (навіть з використанням подій) може бути повільнішою, ніж прямі обчислення в ANN.

11.3.4. Сучасні підходи до навчання SNN

· Ансамблеві методи: Використання ANN для навчання, а потім конвертація їх в SNN (шляхом заміни активацій на частоту спайків).
· Surrogate gradient: Використання «сурогатних» похідних під час зворотного поширення — заміна реальної (розривної) функції спайка гладкою функцією для обчислення градієнта.
· Навчання з підкріпленням: Застосування RL до SNN.

11.4. Нейроморфні процесори: TrueNorth, Loihi, SpiNNaker

Для ефективної роботи SNN потрібне спеціальне апаратне забезпечення, яке враховує їхні особливості: асинхронність, подієвість, розподіленість. Це і є нейроморфні процесори.

11.4.1. TrueNorth (IBM, 2014)

· Архітектура: 4096 ядер, кожне моделює 256 програмованих нейронів. Загалом 1 мільйон нейронів і 256 мільйонів синапсів.
· Принцип роботи: Асинхронний, подієвий. Нейрони обмінюються спайками через мережу пакетів.
· Енергоспоживання: Надзвичайно низьке — близько 70 мВт при роботі (для 1 млн нейронів). Це в тисячі разів ефективніше за звичайні процесори.
· Застосування: Розпізнавання образів, сенсорна обробка, але з обмеженою гнучкістю (програмується спеціальною мовою).

11.4.2. Loihi (Intel, 2018)

· Архітектура: 128 ядер, 131 тисяча нейронів, 130 мільйонів синапсів.
· Ключова особливість: Підтримує навчання на чипі за правилом STDP. Може адаптуватися без участі зовнішнього комп'ютера.
· Енергоефективність: До 10^4 разів вища за стандартні процесори для деяких задач (наприклад, розпізнавання запахів, жестів).
· Програмування: Підтримується через спеціальні бібліотеки та фреймворки (Lava), інтеграція з Python.

11.4.3. SpiNNaker (Manchester University)

· Архітектура: Масив з мільйонів ARM-процесорів, з'єднаних високошвидкісною мережею. Кожне ядро може моделювати кілька (до 1000) простих нейронів.
· Призначення: Моделювання великих (до мільярда нейронів) мереж в реальному часі. Використовується для нейронаукових досліджень.
· Гнучкість: Дуже висока — можна моделювати різні моделі нейронів (LIF, Izhikevich) та синапсів. Енергоефективність нижча, ніж у спеціалізованих чипів, але гнучкість набагато вища.

11.4.4. Інші проєкти

· BrainScaleS (Heidelberg University): Аналоговий нейроморфний пристрій, що працює в прискореному масштабі часу (у 10^4 разів швидше за біологічний).
· Akida (BrainChip): Комерційний нейроморфний процесор для edge-застосувань.
· NeuroGrid (Stanford): Аналогова система для моделювання великих мереж.

11.5. Енергоефективність та перспективи

Чому нейроморфні обчислення є настільки важливими для майбутнього, особливо для робототехніки?

11.5.1. Порівняння енергоефективності (згадаймо Главу 4)

Система Енергоспоживання Продуктивність (операцій/с) Енергоефективність (опер./Дж)
Людський мозок ~20 Вт ~10^{15} ~5×10^{13}
GPU (NVIDIA A100) ~300 Вт ~10^{14} (INT8) ~3×10^{11}

Нейроморфний чип Loihi ~0.5 Вт ~10^{11} ~2×10^{11}
Нейроморфний чип TrueNorth ~0.07 Вт ~10^{10} ~1.4×10^{11}

Висновок: Нейроморфні чипи наближаються до енергоефективності мозку, випереджаючи GPU на 2-3 порядки за енергією на операцію. Для робота, який має працювати від батареї, це критично важливо.

11.5.2. Переваги SNN та нейроморфних систем для робототехніки

1. Енергоефективність: Дозволяє створювати автономних роботів з тривалим часом роботи.
2. Низька затримка (low latency): Асинхронна обробка дозволяє реагувати на події майже миттєво, без очікування такту. Це важливо для уникнення перешкод, балансування.
3. Подієва обробка (event-based): Сенсори (наприклад, подієві камери — event cameras) генерують дані тільки тоді, коли відбувається зміна в сцені. Це ідеально поєднується з SNN, створюючи надзвичайно ефективний сенсорний цикл.
4. Природна адаптація: STDP та інші локальні правила дозволяють роботу адаптуватися до змін навколишнього середовища без участі зовнішнього навчання.

11.5.3. Виклики та перспективи

· Навчання: Як навчати SNN виконувати складні завдання (наприклад, планування шляху) так само добре, як ANN?
· Інструментарій: Потрібні кращі програмні засоби, бібліотеки, фреймворки, які б дозволили програмістам, звичним до PyTorch/TensorFlow, легко використовувати SNN.
· Гібридні системи: Найближче майбутнє, ймовірно, за гібридами: SNN для сенсорної обробки та низькорівневого контролю, а ANN (або класичні алгоритми) — для високорівневого планування та абстрактного мислення.

Підсумок Глави 11

Ми повернулися до витоків і розглянули нейроморфний підхід:

1. Біологічні нейрони спілкуються мовою спайків, кодуючи інформацію в часі.
2. Існують різні моделі спайкових нейронів: від простої LIF до біологічно детальної Hodgkin-Huxley.
3. Навчання в SNN базується на локальних правилах, найвідоміше з яких — STDP, що змінює ваги залежно від часу спайків.
4. Для ефективної роботи SNN створюються спеціальні нейроморфні процесори (TrueNorth, Loihi, SpiNNaker), які наближаються до енергоефективності мозку.
5. SNN та нейроморфне залізо мають величезний потенціал для робототехніки завдяки енергоефективності, низькій затримці та природній адаптації. Головний виклик — розвиток методів навчання та програмної екосистеми.

Частина 3. Навчання та оптимізація розпочнеться з наступної глави. Ми перейдемо від конкретних архітектур до глибшого розуміння процесів навчання, методів оптимізації та регуляризації.

Частина 3. Навчання та оптимізація

Глава 12. Методи оптимізації в глибокому навчанні

«Навчання нейронної мережі — це спуск з гори в тумані. Ви не бачите всієї місцевості, тільки схил під ногами. Мистецтво оптимізації — в тому, щоб обрати правильний розмір кроку і напрямок, щоб не впасти в прірву (вибух градієнтів) і не застрягти на плато (локальний мінімум).»

12.1. Градієнтний спуск та його варіанти: SGD, Momentum, Nesterov

У розділі 2.4 ми познайомилися з базовою ідеєю градієнтного спуску. Тут ми розглянемо його практичні варіації, які використовуються для навчання реальних моделей.

Нагадаємо основне правило оновлення параметрів:

$\Theta_{t+1} = \Theta_t - \eta \nabla L(\Theta_t)$

де η — швидкість навчання, а $\nabla L(\Theta_t)$ — градієнт функції втрат.

12.1.1. Повний пакетний градієнтний спуск (Batch Gradient Descent)

· Ідея: На кожному кроці градієнт обчислюється за всією навчальною вибіркою.
· Достоїнства: Стабільна збіжність, точний напрямок градієнта.
· Недоліки: Повільно на великих даних, потребує багато пам'яті, не може оновлювати параметри онлайн (у міру надходження даних).
· Використання: Практично не використовується в сучасному глибокому навчанні через величезні датасети.

12.1.2. Стохастичний градієнтний спуск (Stochastic Gradient Descent, SGD)

· Ідея: На кожному кроці градієнт обчислюється за одним випадковим прикладом $x^{(i)}$.
· Достоїнства: Дуже швидко (кожен крок — це один приклад), може оновлювати параметри онлайн, краще виходить із локальних мінімумів завдяки шуму.
· Недоліки: Дуже шумна траєкторія, повільна збіжність, не використовує переваги векторизації.
· Оновлення: $\Theta_{t+1} = \Theta_t - \eta \nabla L^{(i)}(\Theta_t)$

12.1.3. Міні-пакетний градієнтний спуск (Mini-batch Gradient Descent)

· Ідея: Компроміс між попередніми двома. Градієнт обчислюється за невеликим пакетом (batch) прикладів (типові розміри: 32, 64, 128, 256).
· Достоїнства:
  · Використовує ефективні матричні операції на GPU/CPU (векторизація).
  · Шум градієнта допомагає регуляризації та виходу з локальних мінімумів.
  · Швидкість і стабільність.
· Використання: Це стандартний підхід у глибокому навчанні.
· Оновлення: $\Theta_{t+1} = \Theta_t - \eta (1/|B|) \sum \nabla L^{(i)}(\Theta_t)$ для $i \in B$

12.1.4. SGD з інерцією (SGD with Momentum)

Проблема звичайного SGD: він може довго спускатися в пологих ярах (де градієнт маленький в одному напрямку і великий в іншому). Інерція (momentum) додає "фізики": ми запам'ятовуємо попередні оновлення і додаємо їх до поточного.

Ідея: Уявіть важку кулю, яка котиться з гори. Вона накопичує інерцію і може пройти невеликі ями (локальні мінімуми) та вирівнювати своє падіння в ярах.

Математика:

$v_t = \gamma\, v_{t-1} + \eta\, \nabla L(\Theta_t)$

$\Theta_{t+1} = \Theta_t - v_t$

де:

· v_t — вектор швидкості (імпульсу),
· γ — коефіцієнт інерції (зазвичай 0.9 або 0.99),
· η — швидкість навчання.

Ефект:

· Згладжує траєкторію.
· Прискорює збіжність у напрямках зі сталим знаком градієнта.
· Допомагає вийти з локальних мінімумів.

12.1.5. Прискорений градієнт Нестерова (Nesterov Accelerated Gradient, NAG)

Це вдосконалення методу інерції, запропоноване Юрієм Нестеровим. Ідея: спочатку зробити крок у напрямку накопиченої інерції, а потім обчислити градієнт і скоригувати його.

Інтуїція: Замість того щоб обчислювати градієнт у поточній точці Θ_t, ми дивимося, який градієнт буде в точці, куди ми збираємося прийти за інерцією ($\Theta_t - \gamma\, v_{t-1}$). Це дозволяє раніше реагувати на зміну ландшафту.

Математика:

$v_t = \gamma\, v_{t-1} + \eta\, \nabla L(\Theta_t - \gamma\, v_{t-1})$

$\Theta_{t+1} = \Theta_t - v_t$

На практиці NAG часто дає трохи кращі результати, ніж стандартний momentum.

12.2. Адаптивні методи: AdaGrad, RMSProp, Adam, AdamW

Підбір швидкості навчання η — критично важливе завдання. Адаптивні методи намагаються автоматично підлаштовувати швидкість навчання для кожного параметра окремо.

12.2.1. AdaGrad (Adaptive Gradient, 2011)

Ідея: Рідкісні параметри (ті, що оновлюються нечасто) повинні мати більшу швидкість навчання; часті — меншу. Для цього накопичується сума квадратів градієнтів для кожного параметра.

Оновлення:

$g_t = \nabla L(\Theta_t)$ (градієнт)

$G_t = G_{t-1} + g_t^2$ (накопичення квадратів градієнтів, поелементно)

$\Theta_{t+1} = \Theta_t - (\eta / \sqrt{(G_t + \varepsilon)}) \cdot g_t$

де ε — мала константа для стабільності (щоб уникнути ділення на нуль).

Достоїнства: Добре працює для розріджених даних (наприклад, у задачах NLP з великим словником).

Недоліки: Накопичена сума квадратів G_t монотонно зростає, тому швидкість навчання з часом стає дуже маленькою і навчання може зупинитися достроково.

12.2.2. RMSProp (Root Mean Square Propagation, 2012)

Ідея: Вирішує проблему AdaGrad, використовуючи експоненційно зважене ковзне середнє квадратів градієнтів, а не їх повну суму. Це дозволяє «забувати» старі градієнти.

Оновлення:

$g_t = \nabla L(\Theta_t)$

$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) g_t^2$ (експоненційне середнє)

$\Theta_{t+1} = \Theta_t - (\eta / \sqrt{(E[g^2]_t + \varepsilon)}) \cdot g_t$

де β — коефіцієнт затухання (зазвичай 0.9).

Достоїнства: Добре працює в нестаціонарних задачах, особливо в RNN. Став популярним завдяки роботі Джеффрі Гінтона з курсу Coursera.

12.2.3. Adam (Adaptive Moment Estimation, 2014)

Adam — один із найпопулярніших оптимізаторів сьогодні. Він поєднує ідеї інерції (momentum) та адаптивної швидкості навчання (RMSProp).

Ідея: Підтримує два експоненційних середніх:

· m_t (перший момент) — оцінка середнього градієнта (аналог інерції).
· v_t (другий момент) — оцінка дисперсії градієнта (квадратів, як у RMSProp).

Оновлення:

1. $g_t = \nabla L(\Theta_t)$
2. $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$

3. $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
4. $\hat{m}_t = m_t / (1 - \beta_1^t)$ (корекція зсуву для перших кроків)
5. $\hat{v}_t = v_t / (1 - \beta_2^t)$ (корекція зсуву)
6. $\Theta_{t+1} = \Theta_t - \eta \cdot \hat{m}_t / (\sqrt{(\hat{v}_t)} + \varepsilon)$

Параметри:

· η — швидкість навчання (часто 0.001).
· β_1 — коефіцієнт для першого моменту (зазвичай 0.9).
· β_2 — коефіцієнт для другого моменту (зазвичай 0.999).
· ε — мала константа (наприклад, 10^{-8}).

Достоїнства:

· Добре працює "з коробки" зі стандартними гіперпараметрами.
· Підходить для широкого класу задач.
· Ефективний з точки зору пам'яті.
· Інваріантний до масштабу градієнтів.

Недоліки: Іноді може гірше узагальнювати (давати вищу помилку на тесті), ніж SGD з інерцією.

12.2.4. AdamW (Adam with Decoupled Weight Decay, 2017)

Проблема Adam: L2-регуляризація (weight decay) в Adam реалізована некоректно. Вона додає $\lambda\theta$ до градієнта, але через адаптивну швидкість навчання цей штраф теж масштабується, що не є правильним.

Рішення: Розділити (decouple) weight decay від градієнта. Тобто спочатку оновлюємо параметри за правилом Adam (на основі градієнта втрат), а потім окремо застосовуємо L2-штраф, просто віднімаючи $\lambda\theta$.

$\Theta_{t+1} = \Theta_t - \eta \cdot \hat{m}_t / (\sqrt{(\hat{v}_t)} + \varepsilon) - \eta \lambda \Theta_t$

AdamW став стандартом для навчання трансформерів (зокрема, BERT, GPT) і вважається кращим вибором, ніж класичний Adam.

12.3. Вибір швидкості навчання та її розклад

Швидкість навчання (learning rate, η) — найважливіший гіперпараметр. Її вибір може визначати успіх чи невдачу навчання.

12.3.1. Наслідки неправильного вибору

· Занадто висока η: Функція втрат розбігається (loss → NaN), або модель "перестрибує" мінімум.
· Занадто низька η: Навчання дуже повільне, модель може застрягти в локальному мінімумі або так і не досягнути прийнятної якості.
· Оптимальна η: Швидка та стабільна збіжність.

12.3.2. Розклад швидкості навчання (Learning Rate Schedules)

Найкраща практика — не використовувати постійну η, а змінювати її за певним розкладом.

· Кроковий розклад (Step Decay): Зменшуємо η в певну кількість разів (наприклад, вдвічі) кожні N епох.
· Експоненційний розклад: $\eta = \eta_0 \cdot \exp(-kt)$
· Косинусний розклад (Cosine Annealing): η змінюється за косинусоїдою від початкового значення до майже нуля. Дуже популярний у трансформерах.
· Розклад із прогріванням (Warmup): Спочатку η лінійно збільшується від дуже малого значення до початкового протягом кількох епох, а потім зменшується за іншим розкладом. Це допомагає стабілізувати навчання на початку, особливо для великих моделей (трансформерів). Warmup став стандартом.

12.3.3. Пошук швидкості навчання (Learning Rate Finder)

Метод, популяризований у бібліотеці fast.ai: запускаємо навчання на кілька ітерацій, поступово збільшуючи η від дуже малої до дуже великої, і будуємо графік залежності функції втрат від η. Оптимальна η знаходиться в діапазоні, де втрати найшвидше падають (зазвичай трохи лівіше від точки, де вони починають зростати).

12.4. Проблема сідлових точок та локальних мінімумів

Чому ми взагалі можемо навчати глибокі мережі, незважаючи на те, що їхні функції втрат є вкрай невипуклими (мають безліч локальних мінімумів і сідлових точок)?

12.4.1. Локальні мінімуми vs сідлові точки

Раніше вважалося, що головна проблема — це локальні мінімуми. Але дослідження показали, що в просторах дуже високої розмірності (типових для глибокого навчання):

· Локальні мінімуми рідкісні. Більшість точок, де градієнт дорівнює нулю, є сідловими точками (saddle points).
· У сідловій точці функція має мінімум в одних напрямках і максимум в інших. Градієнт там нульовий, але це не мінімум.

12.4.2. Чому оптимізація все ж працює?

1. Шум градієнта: SGD та міні-пакетний GD вносять шум, який допомагає "виштовхнути" параметри з сідлових точок.
2. Інерція (momentum): Допомагає пролетіти через сідлові точки.
3. Розмірність: У високих розмірностях ймовірність того, що всі власні значення Гессіана будуть додатними (справжній локальний мінімум), експоненційно мала.
4. Якість локальних мінімумів: Виявилося, що більшість локальних мінімумів у глибоких мережах мають значення функції втрат, близьке до глобального мінімуму (вони "плоскі" та "добрі").

12.4.3. Плато (Plateaus)

Більшою проблемою, ніж локальні мінімуми, є плато — великі області, де градієнт дуже маленький. Навчання на плато майже зупиняється. Адаптивні методи (Adam) та інерція допомагають долати плато швидше.

Підсумок Глави 12

Ми розглянули основні методи оптимізації, які рухають сучасне глибоке навчання:

1. Від простого SGD через додавання інерції до NAG.
2. Адаптивні методи (AdaGrad, RMSProp), які підлаштовують швидкість навчання для кожного параметра.
3. Adam та AdamW — поєднання інерції та адаптивності, стандарт сьогодення.
4. Важливість розкладу швидкості навчання (step, cosine, warmup) та методів її пошуку.
5. Природа складнощів оптимізації: сідлові точки та плато, а не локальні мінімуми, є головними викликами.

Автор

Геннадій Буряк — кандидат технічних наук, винахідник, автор понад 60 патентів. Співавтор концепції DSS-асистента правосуддя та цифрової антикорупційної архітектури. Він продовжує досліджувати, викладати й створювати складні інтелектуальні моделі. Його погляд на ШІ — це поєднання інженерної точності та глибокої етики.

2026 рік