МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ВСТУП ДО ТЕХНОЛОГІЙ ШТУЧНОГО ІНТЕЛЕКТУ

Курс лекцій

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інженерія програмного забезпечення комп’ютерних систем»
спеціальності 121 Інженерія програмного забезпечення

Укладачі: В. І. Таран, Ю. Г. Гордієнко, С. Г. Стіренко

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2024

УДК 004.8
В50

Укладачі: *Таран Владислав Ігорович*, д. філософ., ст. викладач кафедри ОТ ФІОТ
Гордієнко Юрій Григорович, д.ф.-м.н., с.н.с., професор кафедри ОТ ФІОТ
Стіренко Сергій Григорович, д.т.н., проф., завідувач кафедри ОТ ФІОТ

Рецензент *Ролік О. І.*, д.т.н., професор, завідувач кафедри ІСТ ФІОТ

Відповідальний редактор *Новотарський М. А.*, д.т.н., проф., професор кафедри ОТ ФІОТ

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 7 від 09.05.2024 р.)
за поданням вченої ради Факультету інформатики та обчислювальної техніки
(протокол № 10 від 29.04.2024 р.)

В50 Вступ до технологій штучного інтелекту [Електронний ресурс] : курс лекцій : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Інженерія програмного забезпечення комп’ютерних систем» спец. 121 Інженерія програмного забезпечення / КПІ ім. Ігоря Сікорського ; уклад.: В. І. Таран, Ю. Г. Гордієнко, С. Г. Стіренко. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2024. – 202 с.

Навчальний посібник розроблено для ознайомлення здобувачів із сучасними підходами та методами побудови систем зі штучним інтелектом, а також розробкою інтелектуальних сутностей — агентів, які здатні навчатись та ефективно вирішувати складні задачі. Курс лекцій містить інформацію щодо теоретичних та практичних основ штучного інтелекту та роботи інтелектуальних систем із використанням різних методів, таких як: логічних, ймовірнісних, машинного та глибокого навчання.

Навчальний посібник призначений для здобувачів ступеня бакалавра напряму підготовки за освітньою програмою “Інженерія програмного забезпечення комп’ютерних систем” спеціальності 121 - “Інженерія програмного забезпечення”.

УДК 004.8

Реєстр. № НП 23/24-527. Обсяг 9,1 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ВСТУП

Курс лекцій “Вступ до технологій штучного інтелекту” призначений для викладання в рамках нормативної дисципліни “Технології штучного інтелекту” бакалаврату 4-го курсу, а також вибіркової дисципліни “Вступ до штучного інтелекту”. Матеріал спрямований на вивчення студентами сучасних підходів та методів побудови систем зі штучним інтелектом (ШІ), а також розробки інтелектуальних сутностей — агентів, які здатні навчатись та ефективно вирішувати складні задачі. В дисципліні розглядається: область та поняття ШІ, структури інтелектуальних агентів, типи та властивості навколишніх середовищ де діють агенти, типи представлення даних в інтелектуальних агентах, принципи обрання агентами дій та прийняття рішень, задача пошуку із використанням алгоритмів локального пошуку, еволюційних та генетичних алгоритмів, бази знань інтелектуального агента та процесу логічного судження, логіка висловлювань, ймовірнісні судження та обрання дій агентом в умовах невизначеності, задача навчання інтелектуального агента методами машинного та глибокого навчання, глибокі нейронні мережі. Даний курс лекцій дозволить майбутніми фахівцями набути важливих компетенцій в сфері інтелектуальних систем та штучного інтелекту.

Метою курсу лекцій є підготовка фахівців, здатних розв’язувати комплексні задачі у сфері розробки інтелектуальних систем та використовувати сучасні засоби і технології ШІ.

Предметом курсу лекцій є:

• теоретичні та практичні основи штучного інтелекту та інтелектуальних систем;

• методи та засоби побудови інтелектуальних систем — агентів;

• методи логічних суджень;

• методи ймовірнісних суджень;

• методи машинного та глибокого навчання.

ТЕМА 1
ПОНЯТТЯ ШТУЧНОГО ІНТЕЛЕКТУ

Область штучного інтелекту охоплює широку сферу знань, що включає в себе не тільки дослідження того як працює людський мозок, але і також розробку інтелектуальних сутностей — агентів, які будуть здатні вирішувати як їм ефективно діяти у найрізноманітніших, і навіть незнайомим їм ситуаціях. Наразі активно ведуться дослідження в області штучного інтелекту, що робить її однією з областей науки, що стрімко розвиваються.

Тематика штучного інтелекту на даний час охоплює великий перелік наукових напрямів, від загальних задача таких, як навчання, міркування, сприйняття, до більш конкретних прикладних застосунків, як доведення математичних теорем, водіння автомобілю або діагностика захворювань. Дослідження в області штучного інтелекту можуть бути використані при вирішенні будь-якої інтелектуальної задачі, оскільки цю область можна розглядати як універсальну та міждисциплінарну.

Розглянемо поняття “штучний інтелект” (ШІ) і що під ним криється. Історично склалось, що дослідники розглядали декілька різновидів штучного інтелекту:

1. **Перший підхід** зосереджувався на визначенні інтелекту з точки зору його відповідності поведінці людини;

2. **Другий підхід** використовував більш абстрактне, формальне визначення інтелекту, який в подальшому отримав назву “раціональність”.

Поняття раціональності в широкому сенсі розуміється як властивість “поступати правильно”.

Предмет інтелекту також розглядається з двох сторін:

1. Перша фокусуються на внутрішніх властивостях процесів мислення;

2. В той час другий — досліджує зовнішні характеристики інтелекту, а саме інтелектуальну поведінку.

З вище сказаного можна побачити, що підходи до того як мислити і як діяти штучному інтелекту спрямовані на різні аспекти інтелекту. Перший намагається досягти подібності із людиною, в той час як другий — можливість раціональної поведінки. Методи, що використовуються в обох випадках також різняться. Дослідження інтелекту, який буде подібний до людського, проводиться в рамках емпіричних наук, пов'язаних із психологією, що включають спостереженням та гіпотези про фактичну людську поведінку та процеси мислення. В той час, **раціоналістичний підхід** до інтелекту припускає певний синтез математики та техніки, що застосовує методи статистики, теорії ймовірності, теорії управління і економіки та інші.

1.1 Тест Тьюрінгу

Тест Тьюрінгу, запропонований Аланом Тьюрінгом у 1950 році, був розроблений як експеримент мислення, який дозволив би відповісти на філософське питання **“Чи може машина мислити?”.** Щоб пройти цей тест комп’ютер має відповісти на декілька письмових питань, поставлених людиною-експертом, таким чином, щоб людина не змогла б визначити від кого була отримана відповідь, від людини чи комп’ютера. Для того щоб запрограмувати комп’ютер у відповідності із вимогами тесту знадобиться величезний обсяг роботи, оскільки комп’ютер потребуватиме реалізацію функцій у перелічених нижче напрямах:

1. **Обробка природної мови** для спілкування із людиною;

2. **Представлення знань** для того, щоб зберігати інформацію про те що комп’ютер “дізнається” та “почує”;

3. **Автоматичне міркування** для відповіді на поставлені питання та виведення нових міркувань;

4. **Машинне навчання** для адаптації до нових обставин, а також для виявлення та екстраполяції моделей.

5. **Комп’ютерний зір** і розпізнавання мови для сприйняття навколишнього середовища;

6. **Робототехніка** для маніпуляції об’єктами та переміщення у просторі.

Тут не слід змішувати поняття “штучного інтелекту” і “машинного навчання”. **Машинне навчання** — це область штучного інтелекту, в якій вивчається здатність покращувати свої знання базуючись на досвіді. В одних системах штучного інтелекту використовуються методи машинного навчання для досягнення певного рівня знань, в інших — даний підхід не використовується.

Ці **шість перелічених вище напрямів** складають основну частину досліджень в **області штучного інтелекту.**

1.2 Підходи при побудові систем штучного інтелекту

1.2.1 Підхід когнітивного моделювання

Щоб стверджувати, що програма мислить, як людина, потрібно розуміти “як саме людина мислить”. Процес мислення людини вивчається трьома способами:

1. **Самоаналіз** — спроба зрозуміти як і коли думки приходять у нашу свідомість;

2. **Психологічні експерименти** — спостереження за діями людини;

3. **Візуалізація роботи мозку** — спостереження за роботою мозку.

Якщо з’явиться достатньо точна теорія роботи людської свідомості, тоді стане можливим виразити цю теорію у вигляді комп’ютерної програми. Якщо поведінка системи вводу-виводу програми відповідатиме фактичній поведінці людини, це буде свідчити, що комп’ютерна програма може працювати як людина. Наприклад, Аллен Ньюедд та Герберт Саймон у 1961 році розробили програму *General Problem Solver* — універсальний вирішувач задач, метою

якого в більшій мірі було імітувати етапи розв’язку задачі таким чином, як це робила б людина. В подальшому це стало досліджуватись когнітивною наукою. **Когнітивна наука** — це міждисциплінарна область, яка об’єднує комп’ютерні моделі зі штучним інтелектом та експериментальні методи з психології для побудови теорій роботи людського мозку. Останнім часом комбінування методів нейровізуалізації із технологіями машинного навчання з метою аналізу даних, вже привели до появи можливості “читати думки”, тобто можливість визначати семантичний зміст думок у свідомості людини.

1.2.2 Підхід на основі законів мислення

Грецький філософ Арістотель був одним з перших, хто намагався визначити **закони “правильного мислення”,** як процесу формування незаперечних суджень. Його **силогізми** стали взірцем для створення процедур доведення**,** які завжди дозволяють прийти до вірного висновку, при умові якщо дано вірні передумови. В 19 ст. вчені створили точну систему логічних визначень для стверджувань про будь-які предмети, що зустрічаються у навколишньому світі, та про співвідношення між ними. До 1965 року вже було розроблено програми, які могли вирішувати будь-яку проблему, що має вирішення, описану в системі **логічних визначень**. Дослідники в області штучного інтелекту, що притримуються так званої **теорії логіцизму**, припускають, що вони зможуть створити інтелектуальні системи на основі подібних програм. Але **логіка**, в своєму розумінні, вимагає, щоб знання про оточуваний світ були точними. Ці умови в реальності важко досягти. **Теорія ймовірності** заповнює ці проблеми, дозволяючи проводити строгі судження, володіючи неточною інформацією.

1.2.3 Підхід з використанням раціонального агента

Підводячи підсумок всього вищесказаного можна узагальнити, що **агент** (лат. “діяти”) — це щось, **що діє**. Звісно, всі комп’ютерні програми щось роблять, але вважається, що **комп’ютерні агенти** будуть робити більше, а саме:

1. **Працювати автономно**;
2. **Сприймати** навколишнє середовище;
3. **Зберігати знання**, протягом свого існування;
4. **Адаптуватися** до змін;
5. **Встановлювати** та **досягати** певних цілей.

Раціональним агентом називається агент, який діє таким чином, щоб досягати найкращого результату. Якщо раціональний агент знаходиться в **умовах невизначеності**, тоді він має намагатись досягти **найкращого очікуваного результату**.

У підході створення штучного інтелекту на основі **законів мислення** акцент був зроблений на формування правильних логічних висновків. Безумовно, іноді формування правильних **логічних висновків** стає частиною функціонування раціонально агенту. Один зі способів раціональної організації дій полягає в отримані логічним шляхом - висновку, що конкретна дія дозволяє досягти поставленої мети, і потім діяти згідно цього прийнятого рішення. З іншого боку, існують способи діяти раціонально, що не передбачають формування логічних висновків. Наприклад, **рефлекторні дії**, можуть бути більш успішними, порівняно із більш повільними логічно продуманими діями.

Підхід до побудови штучного інтелекту із використанням **раціонального агенту** має дві важливі переваги у порівняні з іншими підходами:

1. Він є **більш загальним** у порівняні із законами мислення. Правильний висновок є тільки однім з багатьох можливих механізмів досягнення раціональності;

2. Краще **піддається науковому дослідженню та розвитку**. Визначення раціональності в цілому можна добре визначити математично. В багатьох

випадках можна спроектувати агента, який гарантовано досягне мети. Але це практично неможливо, якщо метою є імітувати людську поведінку або процес мислення. Це звісно не свідчить про те, що людина ірраціональна, але з точки зору математики вона може приймати не завжди оптимальні рішення.

Загалом підхід у створенні штучного інтелекту із використанням раціонального агента переважав протягом більшої частини історії проведення досліджень у цій області. Спочатку **раціональні агенти** будувались на **логічних принципах** і формували певні плани для досягнення конкретної мети. Пізніше методи, що базуються на **теорії ймовірності** і технологіях **машинного навчання**, дозволили створювати агентів, які були здатні приймати рішення в умовах невизначеності, маючи за мету досягнення найкращого очікуваного результату. Тобто, більшість робіт в сфері штучного інтелекту фокусувались на дослідженні та створенні агентів, які будуть здатні **поступати правильно**. Що вважати правильним визначалось самою задачею, поставленою перед агентом. Ця загальна парадигма була настільки розповсюдженою, що її стали називати **стандартною моделлю**. Ця модель переважала не тільки в області штучного інтелекту, але і в **теорії управляння** — де під “правильністю” розумівся процес мінімізація функції ціни, або максимізації цільової функції. Але в такій моделі був певний недолік. Він полягав у тому, що в разі **ідеальної раціональності** завжди обирається саме оптимальне рішення. Цього не завжди можна досягти в складних умовах, оскільки вимоги до обчислювальних ресурсів, щоб знайти оптимальне рішення, можуть бути занадто великими. Тим не менш, ідеальна раціональність часто є гарною відправною точкою при проєктуванні систем штучного інтелекту.

1.3 Особливості проєктування систем штучного інтелекту

Проблема при проєктуванні систем штучного інтелекту полягає у тому, що необхідно знаходити **компроміс** між досягненням **поставленої мети** та виконанням **правильних дій**. В задачах де чітко сформульовані правила, а

звідси і поставлена чітка мета, підходи, викладені у стандартній моделі, є задовільними. Але в разі розробки інтелектуальних систем, що працюватимуть в умовах **реального світу**, чіткі правила і мету стає в рази складніше визначити. Наприклад, уявімо розробку **самокерованого автомобіля**, який має **безпечно рухатись** від пункту А в пункт Б. Можна припустити, що в цьому випадку метою є лише досягти зазначеного пункту призначення. Однак рух по дорозі супроводжується ризиком зіткнення з іншими автомобілями, чого звісно не можна досягати. Також існують ризики виходу із ладу обладнання автомобіля, нанесення травм пішоходам, та багато іншого. Тому, в разі **жорсткого задання мети** повної безпеки, інтелектуальний агент може прийняти єдине вірне рішення — просто не виїжджати на дорогу. Протилежним прикладом можна привести гру в шахи, де **штучний інтелект** йде на хитрощі, щоб досягти перемоги. Отже, для успішної роботи подібних інтелектуальних систем потрібен компроміс між досягненням мети та діями, які треба виконати для досягнення цієї мети. Це називається проблемою **вирівнювання цінностей** — цінності та цілі, що передаються інтелектуальній системі мають бути **узгодженні із цінностями людини**. Загалом, основна мета інтелектуального агента бути корисним людині.

1.4 Міждисциплінарність штучного інтелекту

Як раніше зазначалось область штучного інтелекту є багатогранною та **міждисциплінарною**. Досягнення в одній області науки, просувають вперед іншу і навпаки. До областей, які внесли свій вклад у розвиток області штучного інтелекту можна віднести:

1. **Філософія**:

- формальні правила для отримання обґрунтованих висновків;
- як виникає думка у людському мозку;
- звідки походять знання;
- яким чином знання призводить до дії.

2. **Математика**:

- формальні правила формування правильних висновків (*формальна логіка*);

- як виконувати судження на основі неточної інформації (*теорія імовірності, статистика*).

3. **Економіка**:

- як приймати рішення згідно з поставлених переваг (*теорія прийняття рішень , оптимізація цінової функції або функції втрат*);

- що робити, коли інші перешкоджають нам (*теорія гри, багатоагентні інтелектуальні системи*);

- як діяти у випадках, коли винагорода може бути отримана у віддаленому майбутньому (*марковські процеси прийняття рішень, навчання з підкріпленням*).

4. **Нейронауки**:

- як інформація обробляється у мозку (*дослідження нейронів мозку, перцептрон, інтерфейс між людським мозком та комп'ютером*).

5. **Психологія**:

- як люди думають та діють (*когнітивні науки*).

6. **Комп'ютерна техніка**:

- створення ефективних та потужних обчислювальних систем (*закон Мура, квантові обчислення*).

7. **Кібернетика**:

- як системи можуть працювати під власним управлінням (*процес навчання, теорія управління*).

8. **Лінгвістика**:

- як мова пов'язана із мисленням (*представлення знань, обробка природних мов*).

1.5 Історія розвитку штучного інтелекту

На завершення теми коротко прослідкуємо розвиток технологій, які були пов’язані із інтелектуальними системами. Перші роботи в цій області базувались здебільшого на досягненнях у математичному моделюванні, що спирались на знання фізіології, функцій нейронів мозку, формального аналізу логіки висловлювань та теорії обчислень:

1. Були запропоновані перші мережеві структури штучних нейронів (Уоррен Мак-Каллок, Уолтер Піттс, 1943), які могли знаходитись у стані “увімкнено” або “вимкнено” в залежності від стимуляції с боку сусідній нейронів. Це дозволяло моделювати логічні зв’язки (І, АБО, НЕ, та інші);

2. Розробка системи доведення математичних теорем — *Logic Theorist* (Аллен Ньюэлл, Герберт Саймон, Кліффорд Шоу, 1956). Програма була здатна вести автоматичні міркування;

3. Розробка універсального вирішувача задача — *General Problem Solver* (Герберт Саймон, Кліффорд Шоу, Аллен Ньюэлл, 1959). Програма була призначена для моделювання процедури вирішення задач подібно до того, як це б робила людина;

4. Створення перцептрону Розенблата (Френк Розенблат, 1962);

5. Програма *SHRDLU* — світ блоків (Террі Виноград, 1968). Перша програма, яка могла вирішувати задачі, поставлені на природній мові.

Але наявні на той час **обчислювальні потужності** не давали можливості вирішувати складні задачі з реального світу, оскільки це потребувало б занадто багато часу на пошук оптимального вирішення задачі. Це призвело до уповільненню розвитку в області штучного інтелекту. Їх розвиток зсунувся у сторону експертних систем. Але розвиток концепцій штучних нейронних мереж продовжувався.

6. Марвін Мінский та Сеймур Паперт у 1969р. довели, що перцептрони (найпростіша форма нейронної мережі) може бути навчана для вирішення певних задач;

7. Навчання з підкріпленням (Річ Саттон, 1988).

Пізніше розвиток глобальної мережі Інтернет (2000 роки) та поява великих обсягів даних (тексти, зображення, аудіо, відео) дав новий поштовх у розвитку штучного інтелекту та машинного навчання. Це дозволило розробляти інтелектуальні системи у різних сферах знань, маючи **великі набори даних для навчання**. Але чим складніші ставали задачі, тим складніші та більші потребувались нейронні мережі. Це призвело до неможливості їх ефективного навчання на поставлених у той час задачах.

Створення концепції **глибокого навчання** (2011 роки) дозволило подолати цю перешкоду. Джефрі Хінтон (2013 р.) продемонстрував значні покращення, порівняно з попередніми системами, які здебільшого мали задані вручну функції. Значних успіхів було досягнуто у задачах розпізнавання мови, машинному перекладі, комп’ютерного зору та інших. З того часу активно почали розвиватись застосунки із **глибокими нейронними мережами**.

ТЕМА 2
ІНТЕЛЕКТУАЛЬНІ АГЕНТИ

В даній темі буде розглянуто основні властивості агентів, різновиди навколишнього середовища та типи агентів, які в них працюють. Використовуючи концепцію раціональності, будуть показані певні набір принципів для побудови агентів, яких цілком справедливо буде назвати інтелектуальними.

2.1 Поняття інтелектуального агенту

Як вже раніше зазначалось, агент — це те, що діє. Але у більш розгорнутому варіанті, можна дати наступне визначення цього поняття. **Агент** — певний об’єкт, що може **сприймати** навколишнє середовище за допомогою датчиків та **взаємодіяти** із середовищем, використовуючи виконавчі механізми. Цей процес можна спрощено представити наступним чином (рис. 2.1).

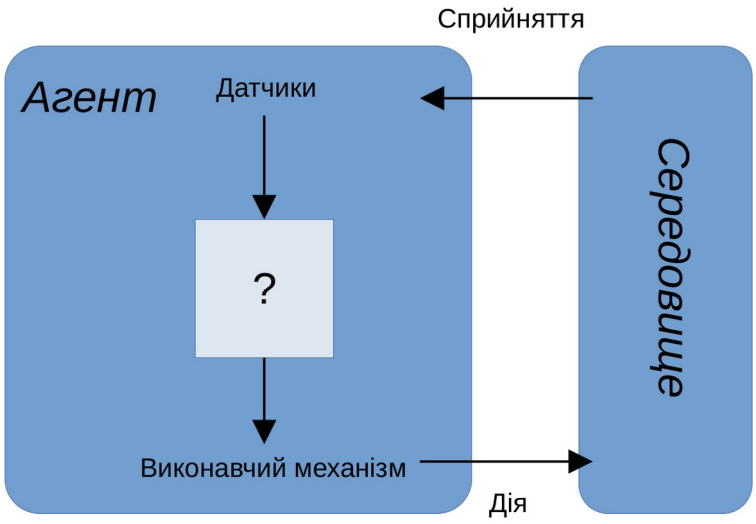


Рисунок 2.1 — Базова схема взаємодії агента із середовищем

Деяка роботизована платформа, яку згідно визначення можна вважати агентом, в якості **датчиків** може мати відеокамери, інфрачервоні датчики відстані, лідари, гіроскопи, акселерометри, тощо, а **виконавчими механізмами**

можуть слугувати, наприклад, різні двигуни. Іншим прикладом агента можна вважати звичайну комп’ютерну програму, яка в якості сенсорних даних на вхід отримує вміст файлу, мережевий пакет, ввід з клавіатури, тощо і виконує певні дії (взаємодіє) шляхом запису файлу, відправки пакету даних або відображення інформації на екрані. Але в цілому факт можливості взаємодії із середовищем не свідчить про те, що кожен агент є інтелектуальним, оскільки будь-який агент може **сприймати власні дії**, але **не кожен** агент **сприймає результат** власних **дій**.

Також дамо визначення того, що можна вважати **навколишнім середовищем**. Теоретично навколишнім середовищем може бути все що завгодно в межах відомого Всесвіту. На практиці навколишнім середовищем називається та частина всесвіту, стан якої ми приймаємо до уваги, створюючи певного агента. Іншими словами, це та частина, яку **сприймає** та на яку **може впливати** агент своїми діями.

Тут використовується термін сприйняття, як процес реєстрації вхідних даних або станів, за допомогою сенсорів агента. **Послідовністю сприйняття** агента називається повна історія всього, що реєструвалось сенсорами агента протягом його життєвого циклу. Отже, можна говорити про те, що вибір агентом будь-якої дії у будь-який конкретний момент часу може залежати від його **вбудованих знань** та послідовності **сприйнятої інформації**, накопиченої до цього моменту. При цьому, дії агента ніколи не можуть залежати від інформації, яку він не сприймав. У відповідь на певну послідовність сприйняття, агентом буде виконана певна обрана дія. Це можна виразити у вигляді **функції агента**, яка відображає певну послідовність сприйняття на деяку дію.

В найпростішому варіанті функцію агента можна представити у вигляді певної таблиці, що описуватиме **поведінку** конкретного агента. Але в реальності подібна таблиця матиме нескінченно великий обсяг, якщо тільки не вводити обмеження на максимальну послідовність сприйняття. Проводячи експерименти з певним агентом, таку таблицю в теорії можна створити,

перевіривши всі можливі варіанти послідовності сприйняття і реєструючи відповідні дії агента. Така **таблиця** — є **зовнішнім** описом функцій агента. **Внутрішнім** описом функцій агента є **програма агента**. Функції агента представляють собою певний абстрактний **математичний опис** дій, у той час як програма агента — це **конкретна реалізація** цих функцій в рамках певної фізичної системи.

Щоб проілюструвати дані ідеї розглянемо простий побутовий приклад поєднання агента-робота із функцією прибирання. Навколишнє середовище для цього прикладу складатиметься із двох кімнат А і Б. Робот, що виконуватиме **функції агента**, сприймає інформацію про своє місце знаходження і в разі, якщо в поточній кімнаті є бурд, прибирає його — дією “прибрати”. Для простоти будемо вважати, що робот може переміщуватись між двома кімнатами шляхом виконання дії “ліворуч” або “праворуч”. Також агент може не виконувати жодної дії. Звісно в реальності, дії робота будуть набагато складніші, наприклад, обертати колеса, визначити поточне положення у просторі, тощо. Послідовність сприйняття для такого прикладу буде містити різноманітні комбінації вхідних даних та дій, які може виконати агент в кожному випадку.

2.2 Показники продуктивності

Раніше вже було дано визначення **раціональному агенту**, як такого, який виконує **правильні дії**, але залишається незрозумілим як можна оцінити **чи дія** була **правильною.** Для цього використовуються **показники продуктивності** агента. Вони дозволяють **оцінити** на скільки **дії**, що виконував агент були **корисними**. Коли агент виконує поставлені перед ним задачі у певному середовищі, він генерує певну послідовність дій**,** базуючись на своєму сприйнятті. Послідовність дій агента призводить до певної зміни станів навколишнього середовища. В разі, якщо ці стани були **корисними,** наприклад з точки зору людини, тоді вважається, що агент діяв **добре** або **правильно**.

Саме поняття **правильності** фіксується **показниками продуктивності**, за допомогою яких можна оцінити будь-яку послідовність станів навколишнього середовища.

Людина, у виборі своїх дій, керується власними бажаннями та наданими перевагами, тому поняття раціональності в даному випадку відображає певну успішність у виборі дії, з точки зору самої людини. З іншого боку, **машина** не має власного бажання, тому показник продуктивності для неї в самому початку є лише у думках розробника цієї машини або користувача, який її використовую. Згадаємо поняття **вирівнювання цінностей**, де результат дій агента мав узгоджуватись із поглядами людини. Повертаючись до прикладу з роботом прибиральником, можна задати показник продуктивності як — “прибрати максимальну кількість сміття” за годину. З точки зору **раціонального агента**, він може **максимізувати** цей показник шляхом прибирання сміття, з подальшим його викидом на підлогу, щоб прибрати його повторно. Для агента це буде правильно, оскільки він діє в поставлених йому умовах. Але для користувача малоймовірно, що така поведінка агента буде корисною. Щоб уникнути такої поведінки агента, можна задати показник продуктивності трохи іншим чином, а саме: “підтримка кімнат чистими”. В такому випадку можна давати агенту винагороду у вигляді балів за кожну чисту кімнату у заданому проміжку часу. Як **загальне правило**, краще обирати **показник продуктивності** з точки зору **вимоги досягти поставленої мети**, а не з точки зору **поведінки агента**.

2.3 Раціональність

Загалом у будь-який момент часу **оцінка раціональності** дій агента визначається, враховуючи чотири фактори:

1. **Показники продуктивності**, які визначають критерії успішності;
2. **Знання агента** про навколишнє середовище, отримані протягом існування;

3. **Дії**, які можуть бути виконані агентом;

4. **Послідовність сприйняття** на поточний момент.

Враховуючи ці фактори, можна сформулювати наступне визначення **раціонального агента** — для кожної можливої послідовності сприйняття, раціональний агент повинен обирати дію, яка, як очікується, **максимізує** його **показники продуктивності**, враховуючи вхідну інформацію від поточної послідовності сприйняття та всіх вбудованих знань, якими він володіє.

Знову повернемось до прикладу робота, який прибирає поточну кімнату, в разі наявності сміття, або переміщується в іншу кімнату. Чи є даний агент раціональним? Для відповіді на таке питання спочатку треба визначити показники продуктивності, інформацію про навколишнє середовище, які датчики та виконавчі механізми має агент. Припустимо, що:

1. Використовується **показник продуктивності**, що передбачає винагороду в один бал за кожну чисту кімнату в кожен інтервал часу життя агента, який складатиме 1000 інтервалів часу;

2. Інформація про **навколишнє середовища** відома завчасно, окрім інформації про розподіл сміття. Дія “прибрати” призводить до очищення поточної кімнати, де знаходиться агент. Дії “праворуч” та “ліворуч” призводять до переміщення агента до іншої кімнати, окрім випадків виходу за межі середовища. В такому разі агент не переміщується;

3. Агент може виконувати лише **дії** “праворуч”, “ліворуч” та “прибрати”;

4. Агент завжди **вірно сприймає інформацію** про поточне положення та наявність сміття в кімнаті.

В даних конкретних умовах можна стверджувати, що агент дійсно раціональний і його продуктивність більш-менш задовільна. Однак при інших умовах можна легко переконатись, що цей самий агент може стати **ірраціональним**. Наприклад, після того як всі кімнати будуть прибрані, агент почне постійно безглуздо **переміщуватись** між кімнатами. В такому разі, якщо за кожне переміщення будуть передбачені окремі **штрафні бали**, такий агент **не зможе** досягти задовільного результату. Тому, **кращім агентом** в таких умовах

буде той, який буде залишатись на місці до тих пір, поки не буде впевнений, що чисті кімнати знову стали брудними**.** І тільки після цього перемішуватись та прибирати сміття. Але якщо інформація про середовище не відома, тоді такому агенту потрібно буде **досліджувати** своє навколишнє середовище.

2.4 Досконалість, навчання, автономність

Треба чітко розуміти різницю між **раціональністю** та **досконалістю.** Агент, який все знає, за визначенням, має інформацію про результат будь-яких своїх дій та може діяти відповідним чином — **досконало**. В реальності така ситуація майже недосяжна, окрім окремих обмежених задач з чітко визначеними правилами. **Раціональність** — це максимізація **очікуваної** продуктивності, в той час як **досконалість** — це максимізація **фактичної** продуктивності при умові, що агент знає все. Загалом задача проектування досконалого агента є нездійсненною, оскільки вкрай важко розробити агента, який буде виконувати найкращі дії, про успішність яких можна буде стверджувати лише після виконання цієї дії. Тобто, такий агент вимагатиме знання про майбутнє. Визначення **раціональності**, на противагу досконалості, не потребує знання про все. Вона залежить лише від **послідовності сприйняття**, що була сформована на поточний момент. В певній мір це можна назвати **накопиченим досвідом** агента.

До цього моменту робилось спрощувальне припущення, що формування послідовності сприйняття не вимагає від агента ніяких дій. Напротивагу, **збором інформації** називається виконання певної дії з метою зміни послідовності сприйняття. Це є елементом **процесу дослідження**, яке має виконувати агент, потрапивши у **невідоме** йому навколишнє середовище. Також раціональний агент, окрім можливості збору інформації, повинен мати можливість **навчатись** на тих даних, які він отримує. Початкова конфігурація агента первинними знаннями може не в повній мірі відображати реальні умови навколишнього середовища в якому працює агент. Тому, протягом свого

існування, агент має накопичувати досвід та **розширювати** свої **знання**, базуючись на ньому. Існують окремі крайні випадки, коли середовище завчасно **відоме** та **передбачуване**. В такому середовищі агенту не потрібно сприймати інформацію або навчатись, він завжди діє правильно. Але такі агенти є вкрай **вразливими** до будь-яких незначних змін у середовищі або послідовності їх дій. Не маючи можливості сприймати зміни, такі агенти продовжуватимуть діяти в рамках запрограмованої поведінки, яка при зміні середовища може стати не правильною. Але агент про це не дізнається, оскільки **не сприймає** середовище. Про таких агентів говорять, що їм **не вистачає автономності,** бо вони в більшій мірі спираються на **знання**, закладені в них **проектувальниками**, ніж на результати свого власного сприйняття та процесу навчання. Щоб бути автономним, раціональний агент повинен **навчатись**, для **адаптації** до середовища та **компенсації** неповних або невірно закладених початкових знань. Наприклад, агент прибиральник, який матиме можливість спрогнозувати де з’явиться сміття, буде краще ніж агент, який на це не здатен.

З практичної точки зору, перед інтелектуальним агентом рідко ставиться вимога, щоб він був автономним із самого початку. Якщо агенту не вистачає досвіду, він повинен буде діяти **випадковим** чином або за певними закладеними проектувальниками початковими діями. Це є подібним тому, як **еволюція** надала тваринам природжені **рефлекси**, які дозволяють їм вижити протягом достатньо тривалого часу, щоб навчитись та стати самостійними. Як результат, після достатньо довгого існування в своєму навколишньому середовищі, агент може стати **незалежним** від своїх початкових знань.

2.5 Визначення проблемного середовища

Наразі вже були визначили окремі терміни, які використовуються при побудові раціонального агента. Цим термінами є: **показники продуктивності**, **навколишнє середовище**, **виконавчі механізми** та **датчики агента**. Всі ці терміни **об’єднується** у поняття **проблемне середовище** в якому працює

інтелектуальний агент. З точки зору того, **як агент взаємодіє** з цим середовищем, розрізняють декілька **класифікацій проблемного середовища**.

2.5.1 Середовище, що можна спостерігати повністю або частково

Якщо датчики агента дозволяють отримати **доступ до повної інформації** про стан середовища в кожен момент часу, проблемне середовище є таким, що можна **спостерігати повністю**. Фактично середовище можна спостерігати повністю, якщо датчики дозволяють аналізувати всі його аспекти, які важливі для вибору агентом дії. “Важливість” в свою чергу визначається заданим критерієм продуктивності. Таке повністю спостережуване середовище є дуже зручним, оскільки агенту не потрібно зберігати внутрішню інформацію про поточний стан, щоб знати все, що відбувається навколо.

Середовище можна **спостерігати частково** в разі, якщо агент має **неточні датчики**, присутні певні шуми або потрібна інформація просто не сприймається наявними датчиками. Наприклад, робот може мати датчик наявності сміття безпосередньо під собою, але подібний датчик ніяким чином не дозволяє визначити засміченість сусідньої кімнати.

Якщо агент взагалі **не має датчиків**, тоді середовище **не можна спостерігати** взагалі.

2.5.2 Одноагентне та мультиагентне середовище

На перший погляд різниця між даними середовищами є очевидною, а саме в **кількості агентів**, які перебувають і працюють в **одному середовищі**. Тут треба розуміти, що вважати **іншим агентом** окрім нашого агента. В цілому, якщо інша сутність також виконує певні дії для досягнення своєї мети, таку сутність можна сприймати в якості іншого агента. При чому, якщо цей **інший агент** своїми діями **максимізує свої показники** продуктивності і це призводить до **мінімізації показників** продуктивності **нашого агента**, таке середовище

називається **конкурентним мультиагентним середовищем**. З іншого боку, якщо **покращення показників** продуктивності **іншого агента** призводить до **покращення показників** продуктивності **нашого** і можливо додаткових **агентів**, таке середовище називається **кооперативним мультиагентним середовищем.** Часто в мультиагентних середовищах одним із показників раціональності є **взаємодія та підтримка комунікації** з іншими агентами.

2.5.3 Детерміноване та недетерміноване середовище

Якщо **наступний стан** середовища **визначається** його **поточним станом** та діями, які виконав агент, то таке середовище називається **детермінованим**. В іншому випадку, середовище є **недетермінованим**. Теоретично, в детермінованому середовищі, яке можна спостерігати повністю, агенту не потрібно буде діяти в умовах невизначеності. Але якщо середовище можна спостерігати лише частково, тоді може складатись враження, що таке середовище є недетермінованим. Більшість реальних ситуацій є настільки складними, що не можливо слідкувати за всіма аспектами навколишнього середовища, тому з практичних міркувань слід розглядати більшість **реальних середовищ** як **недетерміновані**. Середовище для прикладу із роботом прибиральником є детермінованим, але можливі його недетерміновані варіанти. В разі якщо сміття з’являтиметься випадковим чином або буде ймовірність ненадійної роботи функцій робота, таке середовище можна буде розглядати як недетерміноване.

2.5.4 Епізодичне та послідовне середовище

В **епізодичному проблемному середовищі** досвід агента складається із нерозривних епізодів. Кожен епізод включає в себе спочатку сприйняття агентом середовища, а потім виконання однієї дії. При цьому важливо, що наступний епізод ніяким чином не залежить від дій у попередньому епізоді.

Епізодичними є більшість задач **класифікації**. Наприклад, агент, який розпізнає дефекти на конвеєрі, формує кожне рішення для поточної деталі, незалежно від попередньо прийнятих рішень. Більше того, від поточного рішення не залежить наявність дефектів у наступних деталях. На противагу, у **послідовному середовищі** поточне рішення може впливати на майбутні. Прикладом послідовних проблемних середовищ можна назвати гру у шахи, де дії агента можуть нести **довготривалі наслідки**. Епізодичні середовища набагато легше програмувати, оскільки агенту не потрібно думати на перед.

2.5.5 Статичне та динамічне середовище

Якщо середовище може змінюватись за той час поки агент приймає рішення, то таке середовище називається **динамічним** по відношенню до агента. В іншому випадку, воно є **статичним**. Діяти в умовах статичного середовища набагато простіше, оскільки агенту не потрібно спостерігати за навколишнім середовищем в процесі прийняття рішення виконати певну діє. Також не має необхідності слідкувати за часом прийняття рішення. Якщо середовище з плином часу не змінюється, але змінюються показники продуктивності, тоді таке середовище називається **напів динамічним**.

2.5.6 Дискретне та неперервне середовище

Різницю між дискретним та неперервним середовищами слід розглядати відносно стану самого середовища, способу обліку часу, а також сприйняття та дій агента. Наприклад, гра в шахи (без обліку часу) пов’язана з певною множиною **дискретних** сприйняттів та дій. З іншого боку, керування автомобілем — це проблемне середовище із **неперервно** змінюваними станами та неперервним плином часу, оскільки швидкість, положення автомобіля та оточуваних транспортних засобів змінюється в певному діапазоні неперервних величин. При чому, ці зміни відбуваються плавно.

2.5.7 Відоме та невідоме середовище

Ця властивість більше стосується не самого середовища, а стану знань агента про закони функціонування середовища в якому він працює. У **відомому** середовищі результати або ймовірний результат всіх дій є визначеним. Очевидно, що якщо середовище **невідоме**, то агенту потрібно буде досліджувати як воно функціонує, щоб приймати вірні рішення.

2.6 Структура агентів

До цього часу огляд агентів проводився з точки зору аналізу їх поведінки — дій, що виконуються агентом після отримання будь-якої заданої послідовності сприйняття. Тепер розглянемо як можна організувати **внутрішні функції агента**. Задача при роботі із системами штучного інтелекту полягає у розробці програми агента, яка реалізує функції агента та дозволяє відобразити певну послідовність сприйняття на конкретні дії агента. Подібні програми працюють в обчислювальному пристрої, що має фізичні датчики для сприйняття навколишнього середовища та виконавчі механізми для взаємодії із середовищем — виконання дій. Такий набір компонентів називається **архітектурою агента**. Кожна архітектура обирається таким чином, щоб дозволити програмі агента виконувати поставлені задачі у цільовому середовищі. Тобто, архітектура агента має забезпечити передачу результатів сприйняття від датчиків у програму агента та передачу, обраних програмою дій, на виконавчі механізми. Виділяють декілька **основних видів** програм агентів, що реалізують принципи, які лежать в більшості систем штучного інтелекту:

1. Агент із заданою поведінкою на **основі таблиці**;
2. Агент із простою **рефлекторною** поведінкою;
3. Агент із поведінкою на **основі моделі**;
4. Агент із поведінкою, що враховує **мету**;
5. Агент із поведінкою, що враховує **корисність**;

6. Агент, що може **навчатись**.

2.6.1 Агент із заданою поведінкою на основі таблиці

Найпростіший спосіб розробки **раціонального агента** це створення таблиці, яка буде відображати певні дії на будь-яку можливу послідовність сприйняття агента. Така таблиця представляє функції агента, які реалізуються його програмою.

Таблиця 2.1 — Послідовність сприйняття для прикладу робота прибиральника

Послідовність сприйняття	Дія
[А, Чисто]	Праворуч
[А, Брудно]	Прибрати
[В, Чисто]	Ліворуч
[В, Брудно]	Прибрати
…	
[А, Чисто], [А, Чисто]	Праворуч
[А, Чисто], [А, Брудно]	Прибрати
…	
[Б, Чисто], [Б, Чисто], [Б, Брудно]	Прибрати
…	

Агент, який працюватиме за такою таблицею, повинен сприймати черговий стан навколишнього середовища і за його результатом **шукати потрібну дію**, яка відповідає поточній послідовності сприйняття агента.

В цілому даний підхід може працювати в задачах, де можна повністю та правильно заповнити таблицю дій. Але на практиці в реальних умовах, майже неможливо спроектувати подібну таблицю через наявність невідомих станів або через занадто великі обсяги подібної таблиці. Оскільки ключовою задачею штучного інтелекту є створення програм, які в межах поставлених перед агентом задач, забезпечуватимуть його раціональну поведінку і вимагатимуть

невеликий обсяг програмного коду, а не величезні таблиці із всіма можливими варіантами станів та дій.

2.6.2 Агент із простою рефлекторною поведінкою

Іншим різновидом простого агента є рефлекторний агент. Такі агенти обирають свої дії, базуючись лише на поточному сприйнятті середовища. При цьому вони ігнорують всю іншу історію сприйняття. Порівняно із попереднім варіантом, такі агенти будуються на **основі правил**. Ці правила визначають, яку дію має виконати агент у відповідь на певне сприйняття — умову.

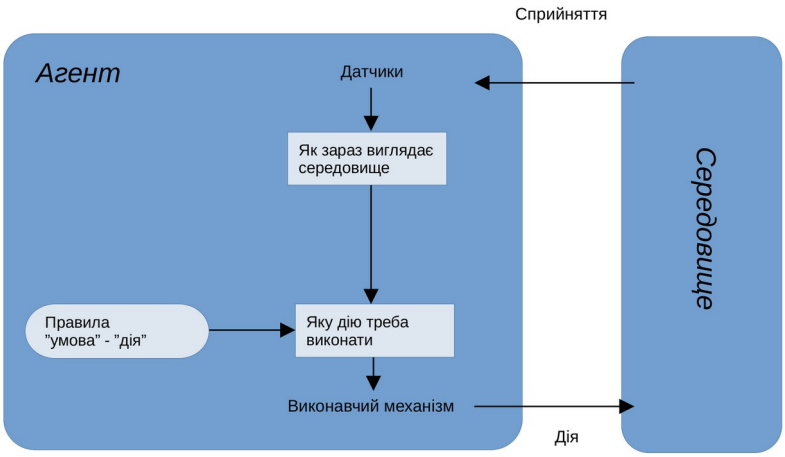


Рисунок 2.2 — Схема взаємодії для агента із простою рефлекторною поведінкою

Для прикладу робота подібними правилами можуть бути: “прибрати”, якщо є сміття; якщо робот знаходиться у кімнаті А, тоді виконати дію “праворуч”, результат — перемістити у кімнату Б; якщо робот знаходиться у кімнаті Б, тоді виконати дію “ліворуч”, результат — перемістити у кімнату А. Загалом для такого простого прикладу цих правил буде достатньо, щоб зробити раціонального агента робота. Можна припустити, що подібна програма агента буде містити ці три правила, які перевіряються конструкціями *if-then-else*, і можливо деякий допоміжний код. Порівняно із таблицями дій, із попереднього варіанту, програма агента в даному випадку буде досить невеликою.

В більш загальному вигляді, подібний підхід у побудові агента може використовувати певний інтерпретатор, який буде **опрацьовувати** результати вхідного сприйняття, **шукати** потрібну умову і **обирати** відповідну дію. Простота даного підходу, що використовую **логічні оператори**, дозволяє реалізувати подібні функції у вигляді апаратних логічних схем або, як програмна альтернатива, нелінійних елементів штучних нейронних мереж. Прості рефлекторні агенти мають основну перевагу у вигляді насправді **простої реалізації**, але їх недоліком є **обмежений інтелект**. Такі агенти можуть бути успішними тільки, якщо правильне рішення може бути прийнято, базуючись лише на поточному сприйнятті. Тобто для середовища, яке можна повністю спостерігати. Поява навіть незначної кількості аспектів середовища, які не можна спостерігати, призводить до значних ускладнень у роботі агента. Наприклад, вихід з ладу окремого датчика агента, яке призведе до неможливості сприймати певний аспект середовища. Якщо у робота вийде з ладу датчик визначення положення у просторі, тоді “яку дію слід виконувати агенту в разі необхідності переміститись в іншу кімнату”? Виконання дії “ліворуч” може призвести до відмови, в разі якщо робот вже знаходиться у лівій кімнаті А. Рефлекторні агенти, які працюють у середовищах із **обмеженим спостереженням**, часто можуть потрапляти у подібні нескінченні цикли. Виходом з такої проблеми може стати введення певної **випадковості** при обранні агентом дій.

2.6.3 Агент із поведінкою на основі моделі

Більш ефективним варіантом дій агента в умовах середовища, яке можна частково спостерігати, це слідкувати за тією частиною середовища, що в даний момент не можна сприйняти. Це означаю, що агент має підтримувати інформацію про певний **внутрішній стан**, який буде залежати від історії сприйняття. Для оновлення такої внутрішньої інформації про стан з плином часу, в програмі агента мають в певній формі бути закодовані знання двох видів:

1. По-перше потрібна інформація про те, як навколишнє середовище **змінюється** з плином часу. Така інформація додатково ділиться на дві частини:

1.1. Результати дії агента;

1.2. Як середовище змінюється, незалежно від агента.

Розглянемо декілька прикладів:

• знання про те, що відбувається, коли **агент виконує дію** “праворуч” — він переміщується у кімнату Б;

• знання факту, що в процесі роботи агента його датчики можуть забруднюватись. Подібні знання про те, як “працює навколишнє середовище”, називається **моделлю середовища**. Ця модель може реалізовуватись у вигляді простих логічних схем або складних наукових теорій.

2. По-друге, потрібна інформація про те, як стан середовища **впливає** на сприйняття агента. Наприклад, якщо датчики агента забруднені, тоді агент отримуватиме спотворенні дані, або якщо агент працює у середовищі з низьким освітленням, то зображення з камер будуть темніші. Цей вид знань називається **моделлю сенсорів**.

Комбінування моделей середовища та сенсорів, дозволяє агенту зберігати необхідну історію стану навколишнього середовища, враховуючи обмеження його датчиків.

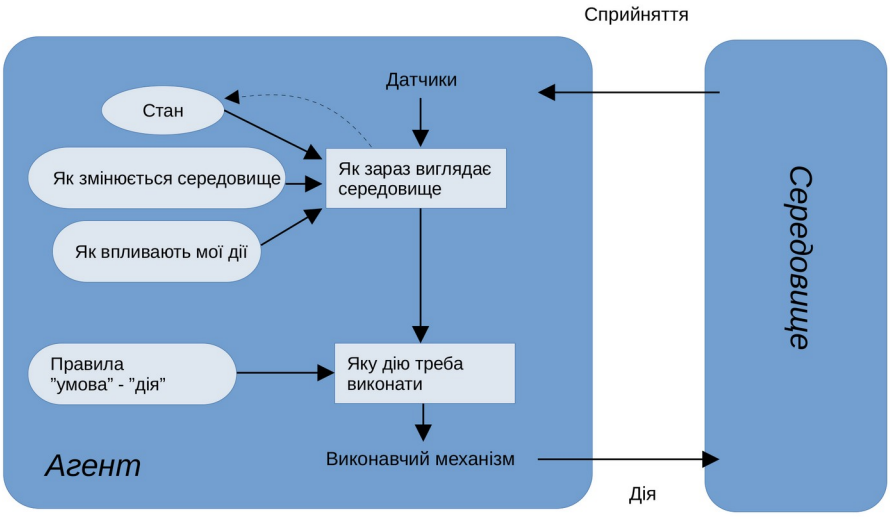


Рисунок 2.3 — Схема взаємодії для агента із поведінкою на основі моделі

2.6.4 Агент із поведінкою, що враховує мету

Простого знання про поточний стан середовища не завжди може бути достатньо для прийняття правильного рішення. Наприклад, якщо перед агентом роботом стоїть вибір повернути праворуч чи ліворуч, правильне рішення буде залежати від того куди йому треба потрапити. Іншими словами, агенту необхідно мати не лише інформацію про поточний стан середовища, але і **інформацію про мету**, яку він має досягти. Програма агента може комбінувати цю інформацію із інформацією про результати можливих дій, щоб зробити вибір, який дозволить досягти поставленої мети. Агентів, які працюють таким чином, називають **цілеспрямованими агентами**.

Іноді вибір дії є простим, наприклад, коли мета досягається після виконання єдиної дії. Але в **реальних умовах** агенту потрібно буде розглядати певну послідовність можливих дія і необхідних умов, щоб знайти спосіб досягнути мети. Для вирішення таких задач, агент може використовувати **методи пошуку та планування**, що є підобластю штучного інтелекту.

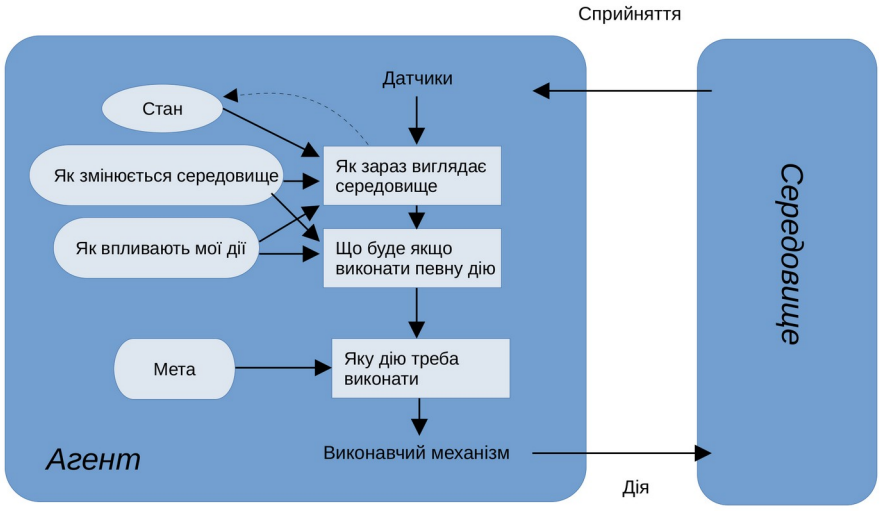


Рисунок 2.4 — Схема взаємодії для агента із поведінкою, що враховує мету

2.6.5 Агент із поведінкою, що враховує корисність

В більшості видів середовищ для виконання правильних дій не завжди достатньо лише визначеної мети. Мета, сама по собі, дозволяє розрізнити лише два можливі стани “добре” або “погано”. Більш загальний показник продуктивності повинен дозволяти **розрізнити** різні стани середовища та дій, відповідно до того, “наскільки” добрими вони будуть для агента. Цей показник “наскільки добре” називається **корисність**. До цього вже розглядався варіант, коли за кожну дію агента, які призводили до зміни послідовності стану середовища, агенту нараховувались бали. Такий підхід дозволяє легко розрізняти більш або менш бажані варіанти досягнення мети. Наприклад, для робота прибиральника більш бажаним є робота у вільній кімнаті, а ніж у кімнаті де перебуває людина, щоб не заважати їй своїм шумом.

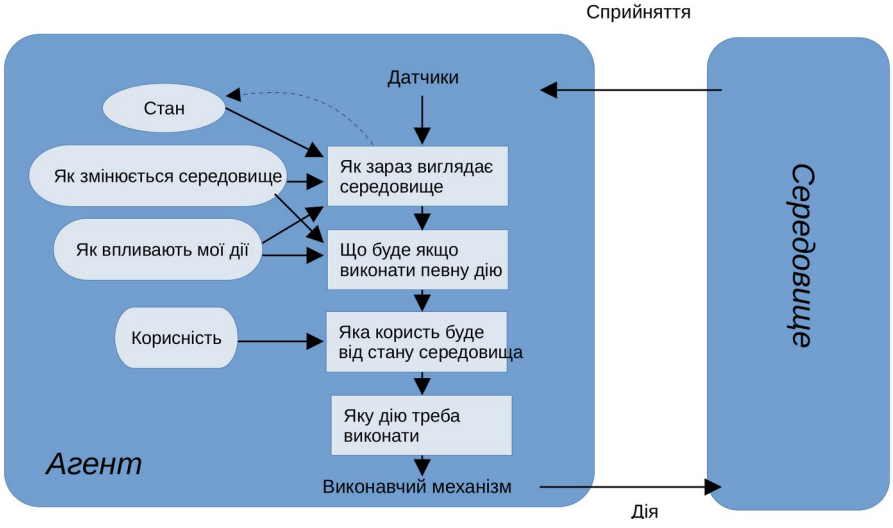


Рисунок 2.5 — Схема взаємодії для агента із поведінкою, що враховує корисність

Функція корисності агента по суті представляє собою внутрішнє відображення його показників продуктивності. В тих випадках, коли внутрішня функція корисності узгоджена із показниками продуктивності, агент обирає дії, які призводять до **максимізації функції корисності**. Такий агент буде проявляти раціональну поведінку згідно обох цих показників. Агент в цьому

випадку називається **практичним агентом**. Використання даного підходу дозволяє знаходити **компроміс** в умовах неоднозначних завдань або цілей, що протирічать одна одній. В реальних умовах недетермінованого середовища, яке частково спостерігається, агенту потрібно приймати рішення в умовах невизначеності. В цьому випадку агент буде обирати дії, які дозволять максимізувати **очікувану** корисність.

2.6.6 Агент, що може навчатись

Вище розглядались програми агентів, в яких обирались різні методи вибору дій. Але постає питання як створити такого агента. В простому розумінні, відповіддю може бути запрограмувати поведінку інтелектуального агента вручну. Але дуже швидко можна прийти до висновку, що потрібен більш продуктивні методи, а ніж написання всього вручну. Ідея таких методів полягає у тому, що потрібно створювати агентів, які матимуть **можливість навчатись** і потім, відповідно, проводити їх навчання. Зараз подібні методи стали домінуючими в різних задачах де є потреба в системах зі штучним інтелектом. Будь-який тип агента, розглянутого вище, наприклад із поведінкою на основі моделі, мети, корисності, та інші, можуть бути доповнені **функціями навчання**.

Навчання має ще одну перевагу, а саме: воно дозволяє агенту функціонувати в **початково невідомих** середовищах та ставати більш успішним та компетентним, порівняно із використанням лише вбудованих початкових знань.

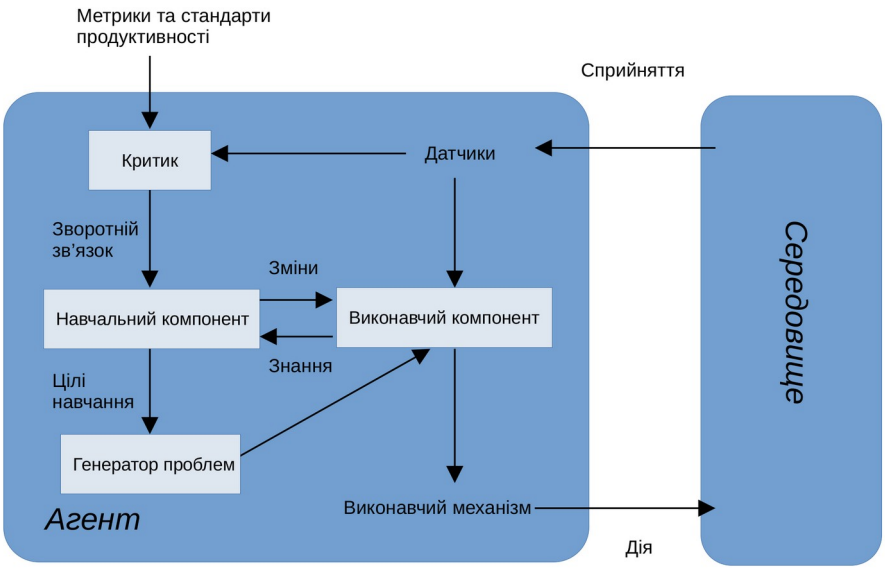


Рисунок 2.6 — Схема взаємодії для агента, що може навчатись

В структурі агента, що може навчатись можна виділити чотири основних компоненти:

1. **Навчальний компонент**, що відповідає за внесення покращень;

2. **Виконавчий компонент**, що забезпечує обрання певної дії;

3. **Критик**, який надає навчальному компоненту інформацію про те, наскільки успішно працює агент в межах обраних показників продуктивності і визначає, як слід змінити виконавчий компонент для його покращення в майбутньому;

4. **Генератор проблем**, який дозволяє агенту досліджувати результати різних дій.

Конструкція самого **навчального компоненту** залежить від **виконавчого компоненту**. І при розробці навчального компоненту, важливо починати його проектування лише після завершення проєктування самого агента. Тобто, головною метою має бути не задача навчання агента, а **правильне проєктування** виконавчої частини агента, щоб він міг ефективно виконувати поставлені задачі.

Під **виконавчим компонентом** наразі розглядається все, що до цього складало структуру агента в цілому. Це процеси сприйняття даних з датчиків, обрання дій та передача команд на виконавчі механізми.

Критик потрібен тому, що сам по собі результат сприйняття не дає ніякої інформації про те “наскільки успішно діє агент”. Для цього використовуються певні метрики або стандарти, які є зовнішніми по відношенню до самого агента.

Генератор проблем використовується для того, щоб пропонувати агенту дії, які дозволять отримати новий інформативний досвід. Сам по собі виконавчий компонент агента прагне виконувати відомі дії, які з його точки зору є найкращими. Але якщо агент може **експериментувати** та виконувати інші не зовсім оптимальні дії, він може знайти кращі дії, з точки зору довготривалої перспективи. Щоб дати агенту таку можливість, використовується генератор проблем.

Навчальний компонент може вносити зміни у будь-які знання агента. В найпростіших випадках, навчання виконується безпосередньо за результатами послідовності актів сприйняття. Спостереження за парами послідовних станів середовища дає можливість агенту отримати інформацію про те, “як його дії впливають на середовище” або “як змінюється середовище”. До прикладу, робот прибирає сміття із різних поверхонь і по результатам того, наскільки чистими ці поверхні стали, він може дізнатись, яку потужність в подальшому слід обирати для кожної поверхні, щоб повністю її прибирати.

Також в процесі навчання метрики та стандарти продуктивності дозволяють виділяти певні вхідні результати сприйняття як винагороди або штрафи. Це забезпечує **зворотній зв’язок** агента із середовищем та дозволяє з’ясувати якість його поведінки.

В цілому, процес навчання інтелектуального агента можна охарактеризувати як **процес модифікації** кожного компонента агента з метою забезпечення більш точної відповідності цих компонентів наявній інформації зворотного зв’язку. Цей процес дозволяє покращити загальну продуктивність агента.

ТЕМА 3
ОБРАННЯ АГЕНТОМ ДІЙ

У попередній темі розглядались програми агентів з точки зору їх **структури**, що складається із різних компонентів. Функції цих компонентів полягають в обранні рішень: “Який стан зараз має середовище”, “Яку дію необхідно виконати агенту в поточний момент”, “Який результат матиме обрана дія”. В цій темі давайте розглянемо питання “Як працюють ці компоненти”.

3.1 Представлення даних в агенті

Для відповіді на це питання розглянемо окремий компонент агента, а саме той, який дозволяє визначити, що відбудеться в результаті виконання певної дії. Подібні результати або стани можуть представлятись у різних **ступенях деталізації** таких, як атомарне, розгорнуте або структурне.

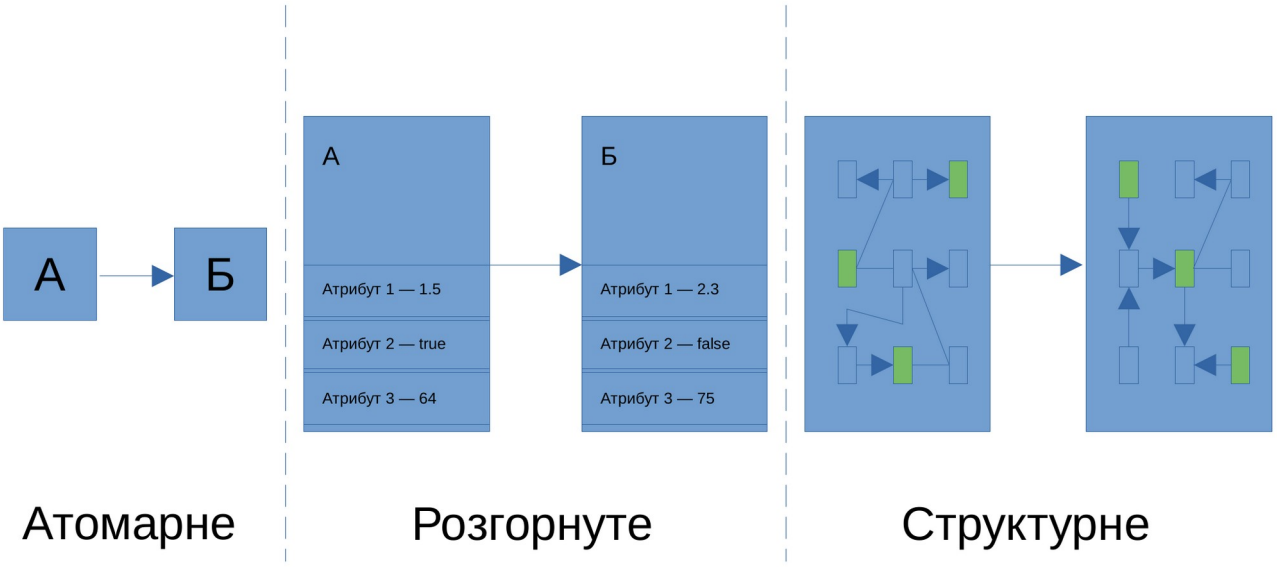


Рисунок 3.1 — Види представлення станів

В **атомарному представленні** — кожен стан середовища є неподільним і не має внутрішньої структури. Наприклад, розглядаючи задачу пошуку маршруту від одного пункту А до іншого — Д (через проміжні пункти Б-В-Г), для такої задачі кожен наступний стан або положення в маршруті буде

представлятись назвою чергового проміжного пункту. Цей черговий поточний пункт, в якому знаходиться агент, буде представлятись у вигляді єдиного атому знання. Це знання — пункт перебування агента буде містити лише певний **ідентифікатор**, що буде єдиною різницею між іншими пунктами, в які може потрапити агент. Всі стандартні алгоритми, що лежать в основі **процедур пошуку**, ігрових процесів, а також процесів прийняття рішень, працюють саме із атомарним представленням станів.

В **розгорнутому представлені** кожен стан може бути розкладений у фіксований набір змінних або **атрибутів**, кожна з яких має своє значення. Для прикладу пошуку маршруту, подібними атрибутами, окрім самого положення в певному проміжному пункті, може бути координати місце знаходження, кількість витрачених коштів на пересування, тощо. В атомарному представленні два різних стани завжди є різними і не мають нічого спільного один з одним. Напротивагу, в розгорнутому представлені різні стани можуть мати однакові значення певних параметрів. Таке представлення набагато спрощує процес обрання рішення як перейти з одного стану в інший. Більшість задач в області **штучного інтелекту** використовують розгорнуте представлення станів. Це дозволяє використовувати алгоритми задоволення обмежень (*constrain sutisfaction*), наприклад — задача розфарбування графу, логіку висловлювань, планування, а також різні алгоритми машинного навчання.

В більшості реальних випадків є потреба представляти оточуване середовище у вигляді певних об’єктів або **сутностей**, які мають певне відношення або **зв’язки** між собою, а не тільки змінні із певними значеннями. Наприклад, плануючи маршрут, навряд при описі чергового стану — пункту, буде передбачено певний атрибут, який показуватиме, що шлях між пунктом Б та В на певному кілометрі був блокований через небезпечні погодні умови. В даному випадку, такі умови або об’єкти зі своїми атрибутами, а також своїм співвідношеннями, зручніше та інформативніше буде описувати у явному вигляді. **Структурне представлення** лежить в основі реляційних баз даних, логіки першого порядку, імовірнісних моделях та більшості методів обробки

природної мови. В дійсності, більшість з того, що людина виражає природною мовою, стосується об’єктів та їх відношення.

Як результат, кожен **наступний вид** представлення (атомарне — розгорнуте — структурне) **збільшує деталізацію** представлення. Більш детальний вид представлення, в певній мірі, містить інформацію, яку включає менш детальний вид. Чим більше деталізоване представлення, тим більш стисло можна виразити в ньому можливі стани середовища. Але з іншого боку, це ускладнює структуру представлення знань, формування суджень та процеси навчання.

3.2 Обрання дій шляхом пошуку

До цього розглядались дії агента як такі, що базуються на поточному сприйнятті та дозволяють отримати бажаний результат вже після виконання однієї дії. Простому табличному або рефлекторному агенту цілком достатньо буде використання таких простих дій, які не мають послідовності, для вирішення поставленої задачі. Але як буде працювати агент, що діє **на основі мети**, у випадку коли виконання лише однієї дії не призводить до очікуваного результату. Розглянемо яким чином такий агент може знайти послідовність дій, які дозволять йому досягнути поставленої мети. Коли правильна дія, яку потрібно виконати агенту, не повністю очевидна, потрібно розробити **план дій** наперед. Тобто, необхідно визначити таку послідовність дій, яка забезпечить досягнення поставленої мети. Такий агент називається **агентом, який вирішує задачі**, а процес, який він виконує — називається **пошуком**.

Агенти, які вирішують задачі, використовують **атомарне представлення**. В такому випадку стан навколишнього середовища розглядається у вигляді певного цілого, що не має внутрішньої структури, видимої для алгоритму пошуку. Розглянемо деякі **алгоритми пошуку** для прикладу найпростішого середовища, а саме: епізодичного, одноагентного, детермінованого, статичного, дискретного, відомого середовища, яке можна спостерігати повністю.

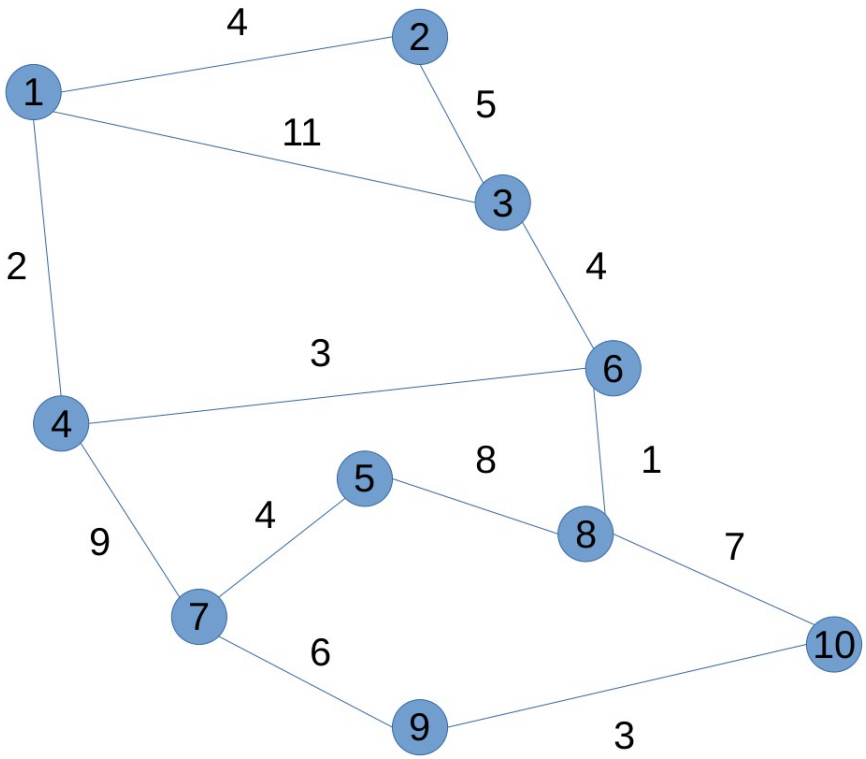


Рисунок 3.2 — Приклад атомарного представлення станів — граф для прикладу задачі пошуку маршруту

В якості прикладу задачі буде взято задачу пошуку маршруту (рис. 3.2) від певного початкового пункту до певного кінцевого пункту. Подібне середовище можна представити у вигляді **графа**, в якому **вершинами** є певні стани, а **ребра** між ними є діями. У випадку, якщо агенту необхідно проходити певні проміжні пункти — стани середовища, щоб досягти потрібного пункту, тоді йому необхідно виконувати певну послідовність дій для досягнення потрібної мети. В цьому прикладі агент має всю необхідну інформацію, тому він може реалізувати певний процес вирішення задачі, що включає в себе наступні етапи:

1. **Вибір мети**. Агент ставить за мету досягти пункту 10. Ця мета визначатиме поведінку агента, задає обмеження та дії, які слід розглядати;

2. **Формулювання задачі.** Агенту потрібно розробити певний опис станів та дій, необхідних для досягнення мети. Ця абстрактна модель буде відповідати частині навколишнього середовища де працює агент. Для даного прикладу агента однією із зручних моделей буде розгляд дій в контексті пересування від

одного пункту до наступного. Це означає, що єдиним фактом, що описуватиме зміну стану середовища для агента, буде ідентифікатор чергового пункту, в який перемістився агент;

3. **Пошук.** Перед тим як виконувати будь-яку дію в середовищі, агент моделює послідовність дій у своїй моделі. Він виконує пошук рішень до тих пір, поки не виявить **послідовність дій**, які дозволяють досягти поставленої мети — для даного прикладу це певний кінцевий пункт. Така послідовність дій називається рішенням. Можливо, агенту потрібно буде змоделювати декілька послідовностей дій, які не дозволять досягти поставленої мети, або визначити, що рішення для досягнення поставленої мети не існую.

4. **Виконання**. Після виконання попередніх етапів, агент може почати послідовно виконувати дії, які він виявив в ході моделювання.

Також треба звернути увагу на важливу властивість. В детермінованому, відомому середовищі, що можна спостерігати повністю, вирішення будь-якої задачі представляється у вигляді фіксованої послідовності дій, яку виявив агент. Якщо модель була вірною, то після того, як агент знайде рішення, він зможе ігнорувати своє сприйняття середовища в ході виконання потрібних дій. Це стає можливим тому, що знайдена послідовність дій гарантовано приведе агента до поставленої мети. В теорії управління таку ситуацію називають **системою без зворотного зв’язку.** Це система, в якій агент взагалі не виконує сприйняття середовища для контролю результату своїх дій. На противагу, у **системах із зворотним зв’язком** агент завжди виконує сприйняття середовища для контролю результатів, незалежно від ситуації. Такий підхід забезпечує більш **стабільну роботу** агента в разі, якщо побудована модель була невірною або середовище є недетермінованим. В середовищі, що можна спостерігати лише частково або недетермінованому середовищі, рішення задачі може представлятись у вигляді розгалужених стратегій дій, що передбачатиме можливість виконання різних або інших дій, базуючись на інформації сприйняття отриманої від датчиків агента.

3.2.1 Задача пошуку рішення

Формальне визначення задачі пошуку включає в себе наступні елементи:

1. **Множину можливих станів**, в яких може перебувати середовище. Ця множина станів називається **простором станів**;

2. **Початковий стан**, із якого агент починає діяти. Наприклад, перебування в певній пункті — вершині, для прикладу задачі пошук маршруту;

3. **Мета** — набір з одного або декількох цільових станів. В конкретній задачі може бути визначена одна мета, наприклад, досягти певного кінцевого пункту, але іноді для задачі може визначатись певна кількість альтернативних кінцевих станів — цілей. Також метою може визначатись певна властивість, яка стосуватиметься множини певних станів. Наприклад, для приклада із роботом прибиральником із минулих тем, метою може бути “відсутність сміття” в будь-якому місце знаходженні агента, незалежно від інших фактів про стан навколишнього середовища;

4. **Дії**, які доступні агенту. При знаходженні середовища і агента у певному стані, йому може бути доступна певна множина дій, які можна виконати. В такому випадку говорять, що кожна така дія може бути застосована до конкретного стану. Наприклад, для задачі із маршрутом (рис. 3.2), якщо агент знаходиться у вершині 4, йому доступні три дії: переміситись у вершину 1, 6 або 7;

5. **Функція визначення результату** — описує, що конкретно виконує кожна дія і в який новий стан перейде агент і середовище після виконання цієї дії у поточному стані;

6. **Функція визначення ціни дії** — визначає числове значення виконання певної дії у поточному стані для досягнення наступного стану. Коли агент вирішує задачу, використання функції ціни дозволяє визначити показники продуктивності агента. Для агента, який шукає маршрут, така функція може повертати відстань між вершинами — пунктами або час пересування між вершинами, тощо.

Послідовність дій формується у **шлях** агента**.** А **рішення** — це шлях від початкового стану до кінцевого — цільового стану. Вважається, що **вартість повного шляху** складається із суми вартостей виконання кожної окремої дії, яка його складає. **Оптимальне рішення** повинно мати найменшу вартість шляху серед усіх можливих рішень.

3.2.2 Приклади задач

Описані раніше підходи до вирішення задач можуть бути застосованими для багатьох варіантів експериментальних середовищ. В цілому подібні задачі можна умовно поділити на стандартизовані та реальні задачі. **Стандартизовані задачі** призначені для ілюстрації або перевірки різних методів вирішення поставлених задач. Такій задачі можна дати короткий та чіткий опис, що дозволяє використовувати її для різних досліджень та порівняння продуктивності різних алгоритмів. Напротивагу, **реальні задачі**, як правило, є **індивідуальними** та **нестандартизованими,** оскільки кожен агент може мати свій набір датчиків та виконавчих механізмів, що надають йому унікальне сприйняття середовища та можливість виконувати дії.

Щоб проілюструвати описані вище підходи розглянемо формальний опис процесу пошуку рішення для прикладу із роботом із минулої теми. Середовище складається із двох кімнат — лівої та правої, в кімнаті може бути присутнім сміття, агент може переміщуватись між кімнатами двома діями “ліворуч” та “праворуч”, також дія “прибрати” — видаляє сміття в поточній кімнаті. Опис для задачі матиме наступний вигляд:

1. **Середовище має 8 станів** (рис. 3.3) — робот може перебувати в одній із двох кімнат, кожна кімната може містити сміття;

2. **Початковий стан** може бути положення у будь-якій з двох кімнат**;**

3. Доступні **три дії**: “прибрати”, “праворуч”, “ліворуч”;

4. **Функція визначення результату.** Дія **“прибрати”** видаляє сміття в поточній кімнаті**, “праворуч”** або **“ліворуч”** переміщую агента у сусідню кімнату;

5. **Мета** — стан, в якому сміття відсутнє в усіх кімнатах;

6. **Функція визначення ціни дії**. Кожна дія коштує 1 бал.

Виходячи із зазначеного формулювання задачі, в якості **показника продуктивності** можна обрати “прибрати все сміття і витратити мінімальну кількість балів”. Відповідно, в ході пошуку рішення даної задачі необхідно **мінімізувати витрати балів**.

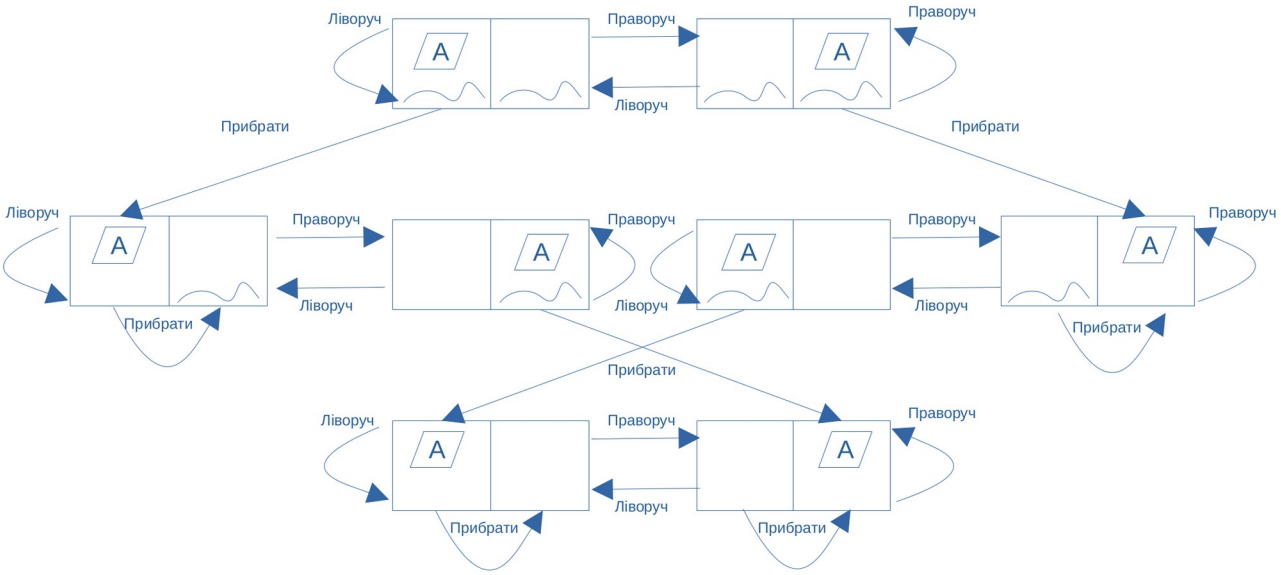


Рисунок 3.3 — Можливі стани навколишнього середовища для прикладу агента-робота

3.2.3 Алгоритми пошуку

Алгоритми пошуку в якості вхідних даних приймають **задачу пошуку** та результатом повертають її рішення. В подібних алгоритмах на граф, що описує середовища, накладається **дерево пошуку.** Це дерево представляє різні шляхи із початкового стану, обрані в ході пошуку, у кінцевий — цільовий стан. Кожна **вершина** в дереві пошуку відповідає певному **стану** в просторі станів, а **ребра** — представляються у вигляді дії. **Корінь** дерева відповідає початковому стану в ході вирішення задачі.

Важливо розуміти різницю між **простором станів** та **деревом пошуку**. **Простір станів** описує **множину станів та дій**, які дозволяють переходити від одного стану в інше. **Дерево пошуку** описує **шляхи між тими станами**, які будуть використані в ході досягнення поставленої мети, наприклад, бажаного стану. Дерево пошуку (від кореня) може мати декілька шляхів, що проходять через різні вершини, до будь-якого заданого стану, але кожна вершина дерева (проміжна або листова) має унікальний шлях у зворотному напрямку до кореня дерева. Розглянемо приклад дерева пошуку для задачі пошуку маршруту (рис. 3.4).

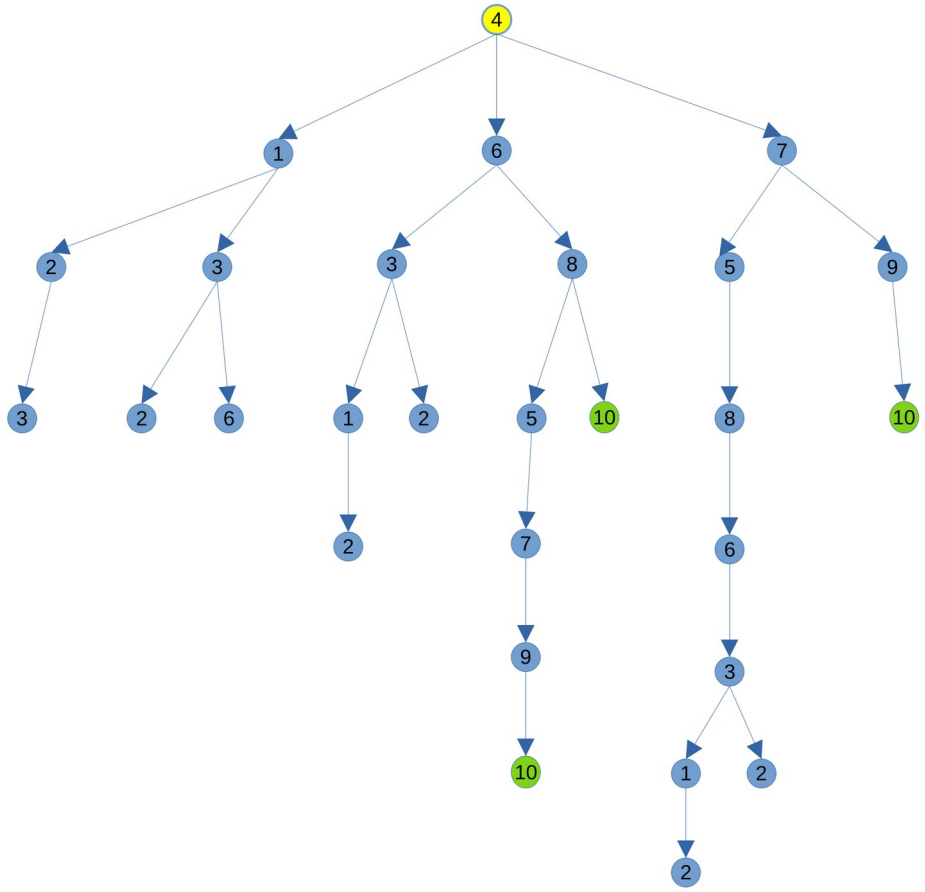


Рисунок 3.4 — Дерево пошуку для прикладу задачі пошуку маршруту

В ході пошуку можна **розгорнути** поточну вершину, щоб отримати інформацію про доступні **дії** (ребра) та **стани** (вершини) до яких вони приводять. Для кожного нового отриманого стану формуються **дочірні**

вершини (вершини на кінці ребра). Множина таких дочірніх вершин називається **периферією пошукового дерева**.

Тепер можна визначити, яку із сформованих дочірніх вершин слід розглянути наступною. В цьому якраз полягає задача пошуку і кожен алгоритм обирає наступні вершини певним чином. Найпростішим варіантом є обрання наступною дочірньою вершиною ту, яка має найменше значення деякої **функції ціни**. Це називається пошуком по **першому кращому збігу**. Його приклад буде розглянуто трохи пізніше.

3.2.4 Структура даних в ході процесу пошуку

Для використання алгоритмів пошуку необхідно організувати дані в певні структури, що дозволять зберігати інформацію про шлях у дереві пошуку. Для представлення окремої вершини дерева можна використовувати структуру типу **“вершина”**, яка матиме чотири компоненти:

1. **Стан.** Зберігає певний стан в загальному просторі станів, який відповідає даній вершині;

2. **Попередня або батьківська вершина**. Вершина із дерева пошуку, яка використовувалась для формування даної дочірньої вершини;

3. **Дія.** Дія, яка була застосована до батьківської вершини для формування даної дочірньої вершини;

4. **Вартість.** Загальна вартість шляху від початкової вершини (стану) до даної вершини.

Слідування з вершини в **зворотному напрямі** за посиланнями на батьківські вершини дозволяє **відновити** стани та дії на шляху до даної вершини. Виконання цієї процедури з **цільової вершини** представляє собою **шукане рішення**.

Також, використовується певна структура даних, яка дозволяє зберігати інформацію про периферію дерева. Вона містить інформацію про дочірні

вершини до яких можна перейти і представляю собою певну **чергу із вершин**. В алгоритмах пошуку можуть використовуватись декілька видів черг:

1. **Черга з пріоритетами.** В початок черги поміщається елемент із **мінімальною вартістю**, яка визначається згідно **певної функції ціни.** Така черга використовується в **пошуку по першому кращому збігу**;

2. **Черга типу FIFO.** Черга функціонує по принципу “перший зайшов — перший вийшов”. Подібна черга використовується при **пошуку у ширину**.

3. **Черга типу LIFO.** Черга функціонує по принципу “перший зайшов — останній вийшов” або стеку. Подібна черга використовується при **пошуку у глибину**.

В графах можуть бути присутні **надлишкові зв’язки** (петлі) — це такі, які приводять у повторювані стани в дереві пошуку. Тому, навіть якщо **граф** містить всього **десять вершин**, наявність подібних петель або циклів призводить до того, що **дерево пошуку** матиме **нескінченну кількість вершин** станів, якщо не передбачено певних обмежень. Для виходу із подібної ситуації необхідно або вводити обмеження, наприклад видаляти петлі, або передбачити можливість зберігати інформацію про вже відвідувані стани — вершини, щоб пошук не потрапляв у них повторно. Алгоритми пошуку прийнято називати **пошуком у графі**, якщо виконується перевірка надлишкових шляхів, та **пошуком по дереву**, якщо перевірка не виконується.

3.2.5 Показники продуктивності пошукового алгоритму

Оцінити продуктивність алгоритму можна за наступними показниками:

1. **Повнота.** Показує чи гарантує алгоритм знаходження рішення, якщо воно мається, або інформування про помилку в разі, якщо рішення не існує;

2. **Оптимальність витрат.** Показує чи забезпечує алгоритм знаходження рішення із мінімальною вартістю шляху із всіх можливих варіантів;

3. **Часова складність.** Показує за який час алгоритм знайде рішення. Реальний час або абстрактний — кількість виконаних операцій;

4. **Просторова складність.** Показує який обсяг пам’яті потрібен для виконання повного пошуку.

3.3 Пошук в ширину

Коли всі стани мають **однакові вартість**, відповідна стратегія називається пошуком у ширину. Це проста стратегія, в якій першим розгортається коренева вершина, далі розгортаються всі дочірні вершини по черзі, далі — їх дочірні вершини, і так далі. Такий підхід є **систематичною стратегією пошуку**, і відповідно, вона буде повною навіть для нескінчених просторів станів. Щоб збільшити ефективність пошуку можна використовувати критерій **раннього досягнення мети**, щоб завершувати пошук в разі потрапляння у цільовий стан.

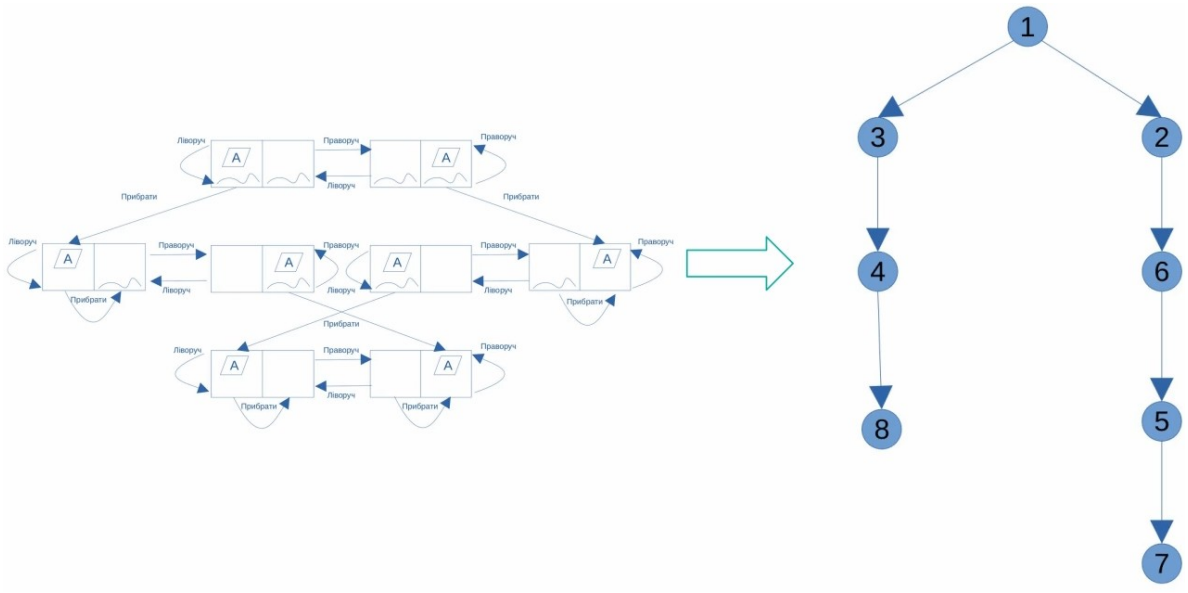


Рисунок 3.5 — Приклад пошуку в ширину

3.4 Пошук по першому кращому збігу

В даному алгоритмі пошуку наступною розгортається вершина, яка має найменше значення вартість виконання дії серед всіх дочірніх вершин в периферії дерева, але при цьому загальна вартість рішення не враховується. Наприклад, в задачі пошуку маршруту наступною обереться вершина, яка має меншу відстань до наступної вершини (рис. 3.6).

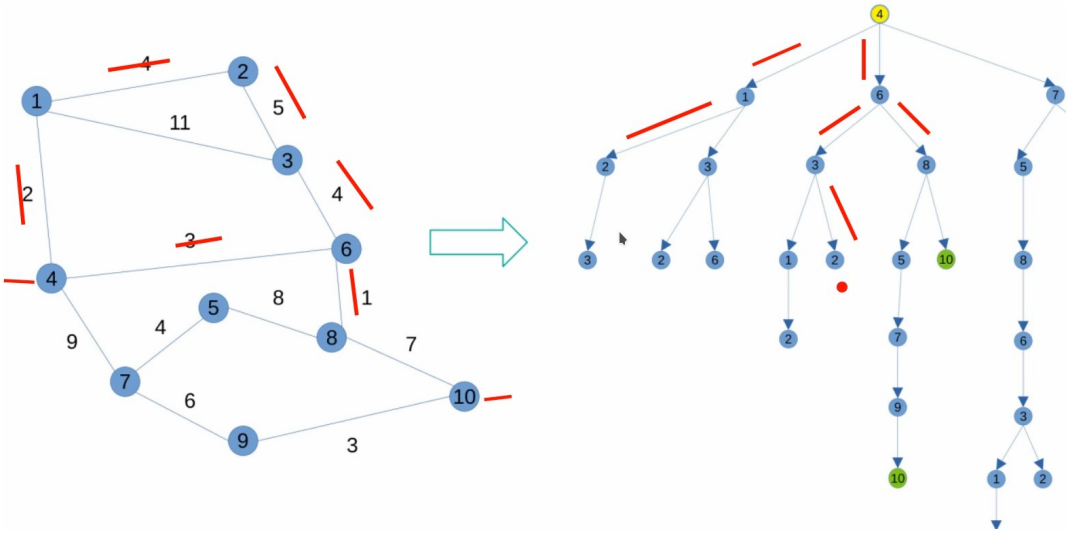


Рисунок 3.6 — Приклад пошуку по першому кращому збігу

3.5 Пошук за алгоритмом Дейкстри (за критерієм вартості)

Коли дії мають різну ціну, логічний є використовувати пошуку по кращому збігу, де функція оцінки буде визначати вартість шляху від кореневого вузла до поточного. Такий підхід називається алгоритмом Дейкстри або в області штучного інтелекту — **пошуком по критерію вартості**. Наприклад, пошук в ширину розповсюджується хвилями в просторі станів одної глибини, а в пошуку по критерію вартості він розповсюджується хвилями по вартості доступних шляхів.

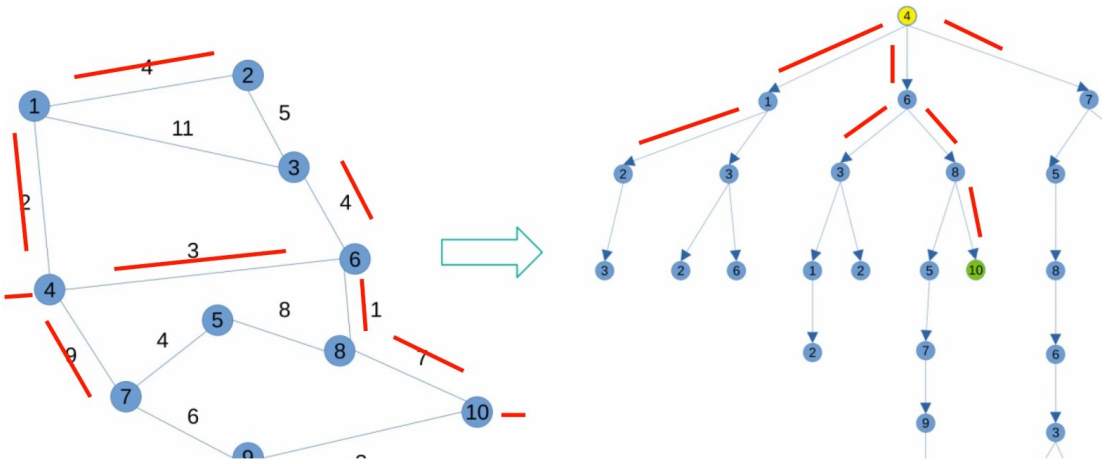


Рисунок 3.7 — Приклад пошуку за алгоритмом Дейкстри

ТЕМА 4
ЗНАХОДЖЕННЯ КРАЩОГО СТАНУ

У попередній темі розглядались алгоритми та приклади задача, в яких рішення представлялось у вигляді послідовності дій агента. Такі підходи в побудові агента можуть застосовуватись в статичних детермінованих середовищах, які можна спостерігати повністю. Попередні приклади пошукових задач потребували переходи у певні проміжні стани від яких залежало досягнення кінцевого цільового стану — мети. Тобто, в таких задачах важливим елементом досягнення мети був правильних шлях при досягнені мети. В цій темі будуть розглянуті алгоритми та приклади задач, в яких шлях не є настільки важливим, оскільки в багатьох задачах важливим є саме досягнення мети, а самим шляхом до неї можна знехтувати.

В таких задачах використовуються **алгоритми локального пошуку**. Такі алгоритми, при своїй роботі, враховують лише поточний стан і розглядають переходи в інші стани із числа суміжних по відношенню до поточного стану. Ці алгоритми не відстежують а ні шлях, а ні множину станів, які вже були досягнуто. Це означає, що такі алгоритми **не є систематичними**, а отже не забезпечують повного дослідження можливих станів тої частини простору пошуку в якому шукається рішення. Тим не менш вони мають декілька важливих переваг, порівняно із попередніми алгоритмами пошуку:

1. їх використання вимагає значно **меншого обсягу пам’яті**;

2. вони зазвичай дозволяють знаходити задовільні рішення у **великих або неперервних просторах станів**, для яких систематичні алгоритми пошуку не підходять.

4.1 Алгоритми локального пошуку

Алгоритми локального пошуку можуть застосовуватись для вирішення **задач оптимізації**, в яких метою є знаходження стану, яке є кращім з точки зору

певної цільової функції. Щоб проілюструвати ідею локального пошуку розглянемо приклад деякої задачі у вигляді ландшафту простору станів (рис. 4.1).

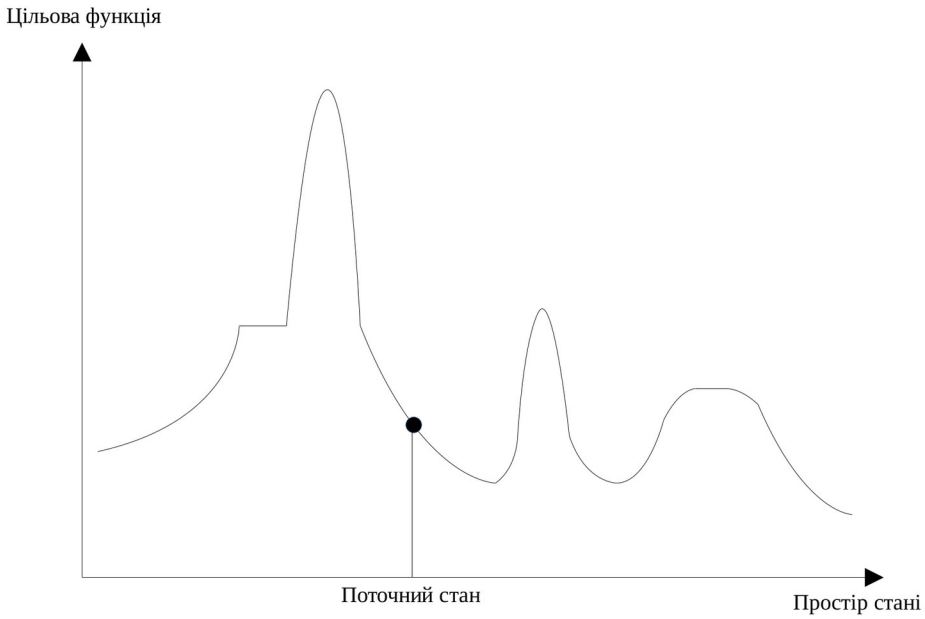


Рисунок 4.1 — Приклад ландшафту простору станів

Кожна точка (стан) в даному просторі характеризується певним значенням, що визначається **цільовою функцією** або функції вартості. Задача в даному випадку полягає у пошуку найбільшого із всіх значень цієї цільової функції. Тобто, шукається точка — найвищий пік, який називається **глобальним максимумом**. Цей процес пошуку називається **сходження до вершини**. Напротивагу, якщо ці значення відповідають функції вартості, тоді така задача полягає у пошуку найглибшої низини. Така точка називається **глобальним мінімумом**, процес її пошуку називається **градієнтним спуском**.

4.1.1 Пошук шляхом сходження до вершини

Алгоритм пошуку шляхом сходження до вершини полягає у відстеженні єдиного поточного стану та переході до наступного стану із найбільшим значенням цільової функції. Цей перехід — рух відбувається у напрямку, який

забезпечує **найбільш крутий підйом**. Робота цього алгоритму завершується після досягнення піку — коли жоден із сусідніх станів не має більшого значення цільової функції. Такий алгоритм пошуку при своїй роботі не аналізує стани за межами найближчих сусідніх станів. Пошук шляхом сходження до вершини іноді називають **жадібним локальним пошуком**, оскільки в ньому передбачено перехід у найкращий сусідній стан, попередньо не розглядаючи інші варіанти у який стан краще перейти далі. На практиці подібний підхід часто демонструє достатньо високі показники продуктивності. Зазвичай, такий пошук дозволяє швидко отримати перехід у сторону покращення рішення, бо покращити початковий гірший результат завжди достатньо легко. Недоліком даного алгоритму є можливість виникнення ситуацій, коли в ході пошуку алгоритм потрапляє у певні **тупикові ситуації** (рис. 4.2) — стани, із яких він не може вийти.

До таких ситуацій можна віднести потрапляння у:

1. **Локальний максимум**, який представляє собою певний пік, більший по відношенню до кожної сусідньої точки — станом, але менший ніж глобальний максимум. Алгоритм сходження до вершини, досягнувши околиць цього локального максимуму, забезпечує пересування вгору до вершини такого піка, але в кінці кінців заходить у тупик, із якого не може вийти;

2. **Хребти**, що представляють собою послідовність локальних максимумів, які досить важко пройти для звичайних жадібних алгоритмів;

3. **Плато** — це така область у ландшафті простору станів, де значення функції оцінки майже однакові — **плоскі**. Ця область може представляти собою плоский локальний максимум із якого не існує виходу у напрямку збільшення значення цільової функції. Також подібна область може мати вигляд виступу із якого теоретично можна успішно перемістити вгору. Але зазвичай алгоритми пошуку шляхом сходження до вершини не здатні вийти за межі плато.

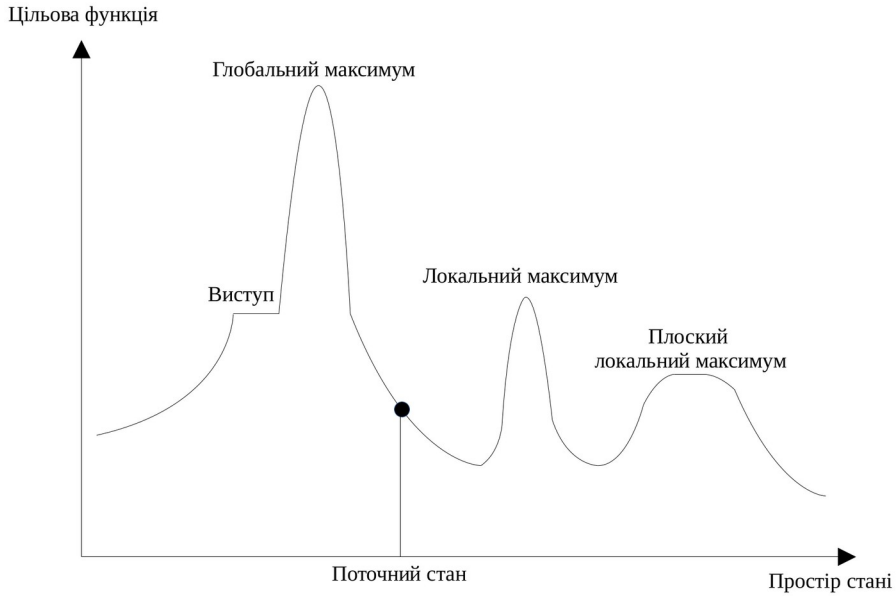


Рисунок 4.2 — Приклади можливих тупикових ситуацій

4.1.2 Модифікації алгоритмів сходження до вершини

Тепер розглянемо, які зміни можна внести в алгоритм сходження, щоб **покращити** його можливості виходити із подібних тупикових ситуацій.

1. Найбільш простим способом вирішення проблеми із проходженням плато є **продовження переміщуватись** у попередньому напрямі в надії, що ця плоска область була насправді виступом перед областю збільшення значень цільової функції. В цілому такий підхід в певній мірі покращує алгоритм, але також і погіршу його у випадку, якщо все таки ця область була плоским локальним максимумом. В результаті це призведе до нескінченного циклу переміщень і алгоритм назавжди застрягне в цій області. В такому випадку можна додатково ввести **обмеження на кількість переміщень**, що дозволить завершувати роботу алгоритму після певної кількості ітерацій. Це дозволяє збільшити відсоток успішно вирішуваних задача за допомогою даного алгоритму, але також збільшує кількість кроків алгоритму при неуспішних рішеннях.

2. Також було розроблено багато варіацій алгоритмів пошуку шляхом сходження до вершини. Наприклад, **стохастичне сходження** до вершини, в якому обрання напряму переміщення вгору виконується випадковим чином.

При цьому ймовірність обрання певного напряму може залежати від крутизни руху в гору. Як правило, такий алгоритм має повільніше сходження, порівняно із попереднім варіантом алгоритму, але на деяких ландшафтах станів він частіше знаходить кращі рішення.

3. Сходження до вершини із **вибором першого кращого** варіанту реалізується за допомогою стохастичного пошуку сходження до вершини шляхом обрання наступного стану випадковим чином до тих пір, поки не буде знайдено кращій стан, порівняно із поточним. Даний підхід використовується у випадках, коли будь-який стан має велику кількість можливих наступних стані.

4. Пошук шляхом сходження до вершини із **випадковим перезапуском**. В цьому варіанті виконується серія пошуків із різних початкових станів, обраних випадковим чином. Такий пошук виконується поки не буде досягнуто шуканого кращого результату. В цілому, якщо форма ландшафту простору станів має декілька локальних максимумів та плато, такий алгоритм дозволяє достатньо швидко досягти непоганого результату.

5. Алгоритм сходження до вершини, який виключає **можливість переміщення вниз** до станів із гіршою оцінкою — більшою вартістю, завжди мають ризик зайти у тупикову ситуацію, з якої вони не зможуть вийти. Алгоритм із можливістю **випадкового блукання** обирають наступний стан куди переміститись, не враховуючи оцінку цього наступного стану. В результаті, такий алгоритм досягне глобального максимуму, але при цьому процес пошуку буде вкрай неефективним.

6. Логічним є доповнення алгоритму сходження до вершини можливістю виконувати перехід у гірші стани, з подальшою перспективою знайти стани навіть краще ніж знайдені до цього. Ідея такого алгоритму полягає у **зміні розміру кроків** переміщення алгоритму, щоб давати можливість вийти із локальних максимумів. При цьому величина цих кроків має бути достатньо великою, щоб покинути локальний максимум, але достатньо малою, щоб не пропустити глобальний. Для досягнення такої поведінки алгоритму, величина кроків поступово зменшується на подальших етапах роботи алгоритму — з

плином часу. Загальна робота описаного алгоритму подібна до стохастичного сходження до вершини, але при цьому перехід до наступного стану відбувається за наступним правилом:

1. якщо оцінка наступного стану **покращує результат**, тоді відбувається перехід в цей стан;

2. якщо оцінка наступного стану **погіршує результат**, тоді алгоритм обере його із певною ймовірністю меншою за одиницю. Ця ймовірність експоненційно зменшується, враховуючи величину погіршення оцінки наступного стану. Також ймовірність переходу у гірший стан, порівняно із поточним, зменшується зі збільшенням часу роботи алгоритму. Тобто, інтенсивність переходів у гірші стани, щоб дати алгоритму можливість вийти із локальних максимумів, є більш ймовірним на початкових етапах роботи алгоритму, а ніж на пізніх.

4.2 Еволюційні алгоритми

Напротивагу попереднім алгоритмам, розробка еволюційних алгоритмів в певній мірі була мотивована **концепцією природного відбору** у природі. Під **популяцією** тут розуміється певна множина особин — станів із певною оцінкою, яка в термінах алгоритму називається **пристосованістю**. Далі найбільш пристосовані особини, тобто із більшим значенням оцінки цільової функції, формують наступні стани — особин, шляхом **рекомбінації**, і формують наступне покоління популяції. Існує велика кількість різних еволюційних алгоритмів, які відрізняються за принципом формування популяцій.

4.2.1 Генетичний алгоритм

Щоб продемонструвати подібні алгоритми розглянемо принцип їх роботи на прикладі генетичного алгоритму. В даному алгоритмі кожна особина представляється у вигляді певної строки літер, аналогічно до представлення **молекули ДНК**, або послідовністю чисел. Для формування наступного покоління із поточної популяції обирається певна кількість **генів** — станів із ймовірністю пропорційній їх функції пристосованістю (значення цільової функції) і попарно відбувається обмін елементами генів шляхом **рекомбінація**. Процедура рекомбінації полягає у виборі точки перетину, яка називається **кросинговером**. Це дозволяє розділити представлення певного гену на дві частини і об’єднати їх, сформувавши нові гени. Також в алгоритм вводиться певний елемент ймовірності, який називається **частотою мутації**, і дозволяє змінювати певні частини в новостворених генах. Наступне покоління включає в себе новостворені гени і деяку кількість попередніх генів, які мали найбільшу функцію пристосованості. Це називається **елітарністю**, і вона гарантує, що пристосованість чергових популяцій не буде зменшуватись на наступних етапах роботи алгоритму. Також може виконуватись **відбракування**, коли видаляються гени або особини із пристосованістю нижчою певного рівня.

Якщо порівнювати продуктивність та ефективність даного алгоритму із попередніми, то популяції на ранніх етапах роботи алгоритму є досить різноманітними, тому процедура кросинговеру дозволяє виконувати **значні переміщення** про простору станів на початку пошуку. Але після **селекції** наступних поколінь популяції в бік збільшення пристосованості, відбувається зменшення різноманіття, що відповідає **зменшенню величини кроків** переміщення.

4.2.2 Приклад розрахунків для генетичного алгоритму

Розрахунки, що передбачаються за даним алгоритмом наведені нижче на рисунках 4.3-4.7.

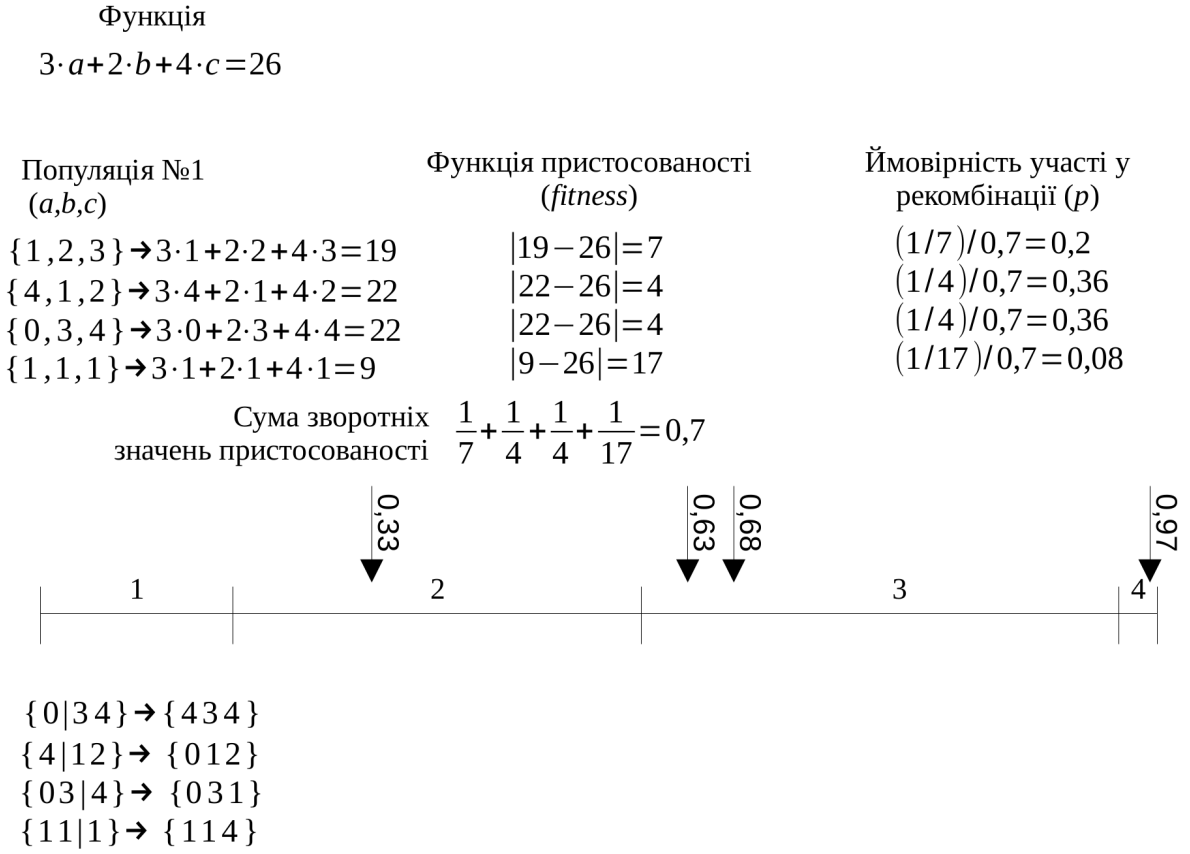
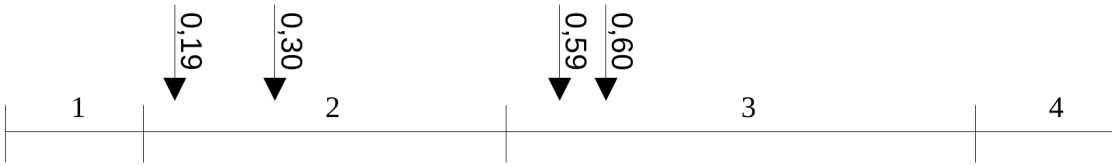


Рисунок 4.3 — Розрахунок 1-ї популяції

Популяція №2 (*a*,*b*,*c*)	Функція пристосованості (*fitness*)	Ймовірність участі у рекомбінації (*p*)
$\{0,1,2\} \rightarrow f(a,b,c)=10$	$\lvert 10-26\rvert=16$	$(1/16)/0{,}45=0{,}14$
$\{4,3,4\} \rightarrow f(a,b,c)=34$	$\lvert 34-26\rvert=8$	$(1/8)/0{,}45=0{,}28$
$\{1,1,4\} \rightarrow f(a,b,c)=21$	$\lvert 21-26\rvert=5$	$(1/5)/0{,}45=0{,}44$
$\{0,3,1\} \rightarrow f(a,b,c)=10$	$\lvert 10-26\rvert=16$	$(1/16)/0{,}45=0{,}14$

Сума зворотніх значень пристосованості $\sum_i (\frac{1}{fit_i})=0{,}45$

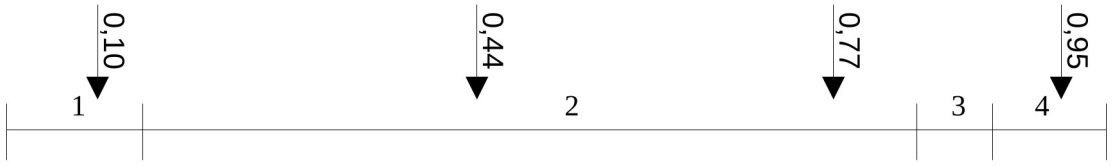


$\{1|14\} \rightarrow \{414\}$
$\{4|34\} \rightarrow \{134\}$
$\{11|4\} \rightarrow \{4(4)4\}$
$\{43|4\} \rightarrow \{114\}$

Рисунок 4.4 — Розрахунок 2-ї популяції

Популяція №3 (*a*,*b*,*c*)	Функція пристосованості (*fitness*)	Ймовірність участі у рекомбінації (*p*)
$\{4,1,4\} \rightarrow f(a,b,c)=30$	$\lvert 30-26\rvert=4$	0,16
$\{1,3,4\} \rightarrow f(a,b,c)=25$	$\lvert 25-26\rvert=1$	0,65
$\{4,4,4\} \rightarrow f(a,b,c)=36$	$\lvert 36-26\rvert=10$	0,06
$\{1,1,4\} \rightarrow f(a,b,c)=21$	$\lvert 21-26\rvert=5$	0,13

$\sum_i (\frac{1}{fit_i})=1{,}55$



$\{1|34\} \rightarrow \{1(4)4\}$
$\{1|34\} \rightarrow \{134\}$
$\{41|4\} \rightarrow \{1(2)4\}$
$\{11|4\} \rightarrow \{4(2)4\}$

Рисунок 4.5 — Розрахунок 3-ї популяції

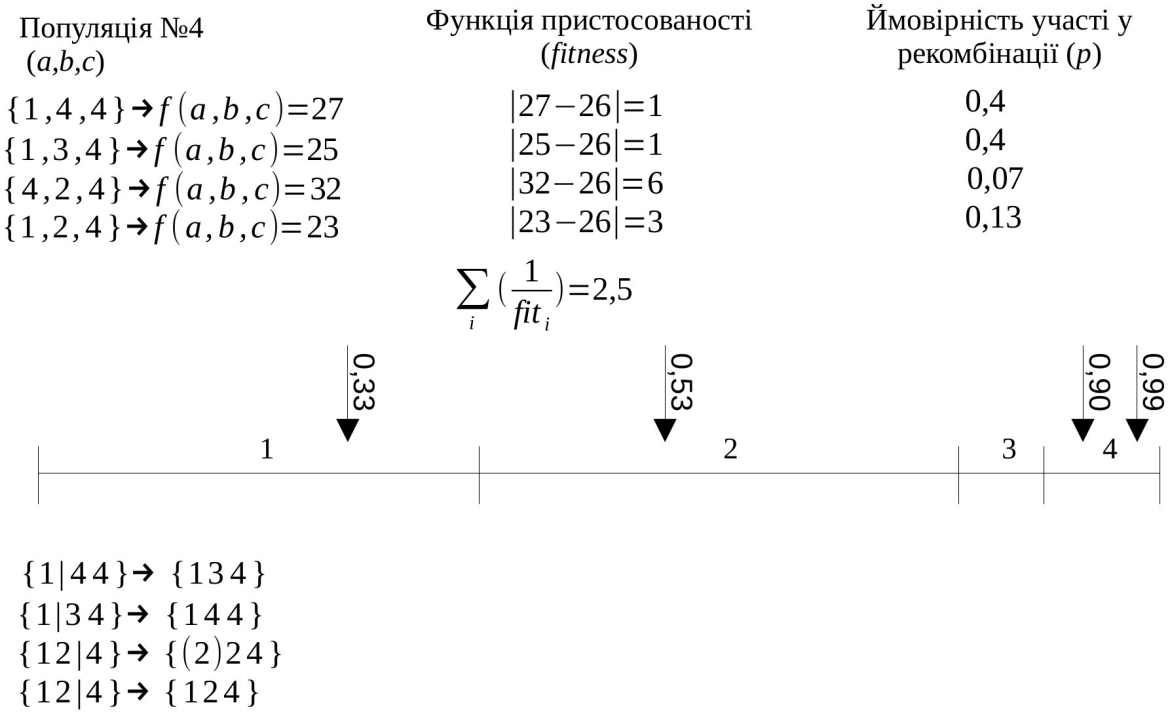


Рисунок 4.6 — Розрахунок 4-ї популяції

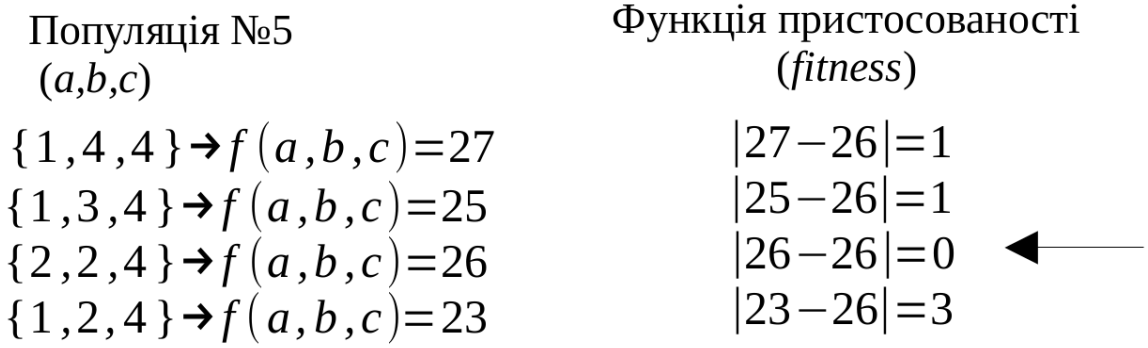


Рисунок 4.7 — Розрахунок 5-ї популяції

4.3 Пошук в оперативному режимі в невідомому середовищі

Розглянуті раніше агенти використовували алгоритми пошуку в **автономному режимі**. Такі агенти обчислювали повне рішення перед тим як виконувати будь-яку дію. Напротивагу такому агенту існують агенти, які виконують пошук в **оперативному режимі**. Вони функціонують за принципом почергового обчислення і виконання чергової дії. В такому випадку, спочатку виконується певна дія, потім аналізується її результат, а після обчислюється

наступна дія. Пошук в оперативному режимі доцільно використовувати в **динамічному середовищі**, де можуть передбачатися певні штрафи за простій або затримку агента. Також даний режим пошуку буде корисним у недетермінованих середовищах, оскільки це дозволить агенту зосередитись на непередбачуваних обставинах, які можуть мати місце в цьому середовищі.

Але в такому режимі роботи потрібен певний компроміс, оскільки чим більше агент планує наперед, тим менше він потраплятиме у складні або **тупикові ситуації**. В разі, якщо оперативний режим пошуку застосовується в невідомому середовищі, де агент не знає стан середовища або результати своїх дій, тоді агенту необхідно виконувати певні дії, спрямовані на **дослідження** подібного середовища. Іншими словами, агент має **експериментувати**. Типовим прикладом застосування пошуку в оперативному режимі може слугувати задача відображення, коли робот знаходиться у невідомій йому будівлі. Робот має дослідити цю будівлю, щоб побудувати карту, яку надалі можна буде використати для переміщення між кімнатами цієї будівлі.

4.3.1 Задача пошуку в оперативному режимі

Загалом будь-яка задача пошуку в оперативному режимі вирішується шляхом почергового виконання обчислень, сприйняття та дії. Для цього задається:

1. **Функція дій**, яка визначає список певних дій, які можна виконати із поточного стану;

2. **Функція ціни**, яка визначає вартість виконання певної дії в поточному стані для переходу у наступний стан. При чому така функція може бути використана тільки після, того як агент дізнається, що результат цієї дії приводить його у цей черговий стан;

3. **Функція мети**, яка перевіряє досягнення цільового стану.

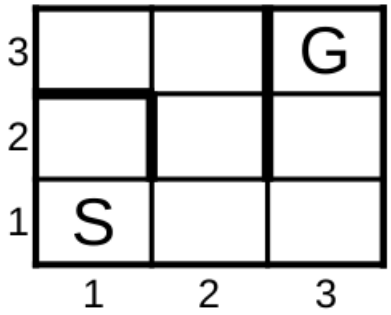


Рисунок 4.8 — Просте навколишнє середовище для агента

Окремо слід відмітити, що агент не може визначити результат виконання певної дії із певного стану в разі, якщо він фактично не знаходиться в цьому стані і не виконує цю дію. Прикладом такого випадку може бути задача із лабіринтом (рис. 4.8). На початку агент (*G*) може бути необізнаним, що виконання дії “вгору” переводить його у верхній стан. Або, що після цього, виконання дії “вниз” приведе його у минулий стан. Таку необізнаність можна в певній мірі зменшити, надавши агенту знання стосовно того, як працюють його дії/функції по переміщенню. Тоді залишиться необізнаність лише стосовно місце знаходження перешкод у середовищі. Нарешті агенту може бути доступна певна функція оцінки, яка дозволить йому визначати відстань до поставленої мети (*S*).

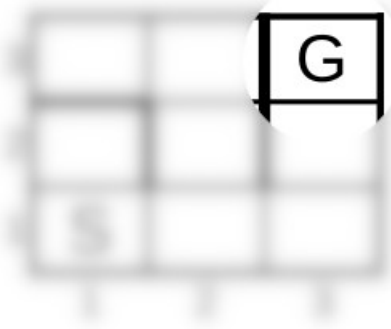


Рисунок 4.9 — Можливі стани, які доступні агенту в оперативному режимі

Як правило, задача агента полягає у тому, щоб досягти мети при мінімальних витратах. Вартістю, в таких випадках, виступає загальна вартість шляху, який пройде агент у пошуках цілі. Зазвичай ця вартість **порівнюється** із

вартістю **оптимального шляху**, який би розрахував агент у відомому середовищі. В термінах алгоритмів пошуку в оперативному режимі це відношення називається **коефіцієнтом конкурентоздатності**. Бажано, щоб цей коефіцієнт був як можна меншим.

Дослідження і пошук, який проводиться в оперативному режимі, часто приводять до потрапляння агента у тупикові ситуації із яких він вже не зможе потрапити у цільовий стан. Такі тупикові ситуації представляють собою реальну небезпеку для агентів, наприклад вулиця для реальних агентів-роботів, оскільки в подібних середовищах можуть бути **невідворотні дії**, після виконання яких вже не можна буде повернутись у попередній стан. В такий середовищах агент має виконувати дослідження достатньо обережно. Напротивагу є середовища, які безпечно досліджувати, оскільки при виконанні неправильної дії, агент зможе повернутись до попереднього стану і спробувати іншу дію.

4.3.2 Агенти, які виконують пошук в оперативному режимі

Після кожної дії агент, який виконує пошук в оперативному режимі, отримує інформацію сприйняття про стан в якому він перебуває. Базуючись на цій інформації, агент може **доповнити** свою карту навколишнього середовища. Потім ця карта буде їм застосовуватись для **планування**, яку дію виконати наступною. Таке чергування етапів планування та дії в алгоритмах оперативного пошуку, доволі сильно відрізняє їх від алгоритмів автономного пошуку.

1. При **автономному пошуку** досліджується модель навколишнього середовища;

2. При **оперативному пошук** досліджується реальне навколишнього середовище.

Розглянутий раніше алгоритм пошуку шляхом сходження до вершини можна вважати алгоритмом пошуку в оперативному режимі, оскільки він також враховує інформацію тільки для одного поточного стану. Але в своїй

найпростішій формі не є досить корисним через те, що не дозволить агента покинути локальний максимум і перейти в інші стани. Більше того даний алгоритм не може використовувати перезапуск випадковим чином тому, що фізично агент не здатен перенести себе у новий випадковий стан — положення. Замість цього, може бути використаний **метод випадкового блукання**, який дозволить агенту досліджувати середовище. В даному методі випадковим чином обирається наступна дія, що доступна в поточному стані. При чому пріоритет надається тим діям, які ще не були спробувані. В цілому такий метод дозволить агенту досягти бажаної мети, проте процес її досягнення може бути доволі повільним. Той факт, що агенти, які виконують пошук в оперативному режимі, початково знаходяться в повній необізнаності дає певні можливості для **навчання**. По перше, агент досліджує навколишнє середовище, реєструючи свій досвід, і може побудувати певну карту або зберегти результати виконання кожної дії із кожного стану. По друге, агент, виконуючи локальний пошук, отримує все більш точні оцінки кожного зі станів, які в кінці кінців дозволять йому приймати все більш оптимальні рішення.

ТЕМА 5
ЛОГІЧНІ СУДЖЕННЯ

В цій темі буде розглянуто проектування агентів, які здатні формувати власні представлення про складне навколишнє середовище. Для цього агент буде використовувати процес **формування логічних висновків**, щоб сформувати нові представлення про середовище. Базуючись на цих нових представленнях, агент зможе приймати рішення про те, які дії слід виконувати далі.

Доволі очевидно, що людина володіє знаннями і саме ці знання дають їй можливість діяти. Цей факт може також розповсюджуватись на область штучного інтелекту. Агенти, що **базуються на знаннях**, використовують процес судження на основі внутрішнього представлення знань, щоб прийняти рішення, яку дію слід виконати на наступному кроці.

Агенти, які розглядались минулими темами, також володіли певними знаннями, але ці знання були досить обмеженими і негнучкими. Такі агенти знали, які дії їм доступні і до якого результату приводить виконання певної дії із певного стану. Але такі агенти не знають загальних фактів. Наприклад, агент, який прокладає маршрут, не знає, що певна ділянка його маршруту не може мати довжину, що виражена від’ємною величиною. Або агент, який грає у шахи, не знає, що дві фігури не можуть займати одне і те саме поле на гральній дошці. Знання, якими володіли такі агенти були корисними для виконання ними поставленої вузької задачі, але вони ніяким чином не могли допомогти в будь-яких інших задачах.

Атомарне представлення, яке використовується агентами, що вирішують задачі, також має **певні обмеження**. Наприклад, в середовищі, яке можна спостерігати лише частково, у агента, який вирішує задачі, єдиним доступним варіантом представлення знання про поточний стан буде певна множина або список всіх можливих станів, до який можна потрапити. Проводячи аналогію із подорожами і маршрутами то, якщо перед людиною поставити задачу відвідати

будь-яке місто із населенням менше, наприклад, 10 тис., то в принципі для цього буде достатнім задати їй лише цю умову. Але, якщо подібну задачу поставити перед агентом, який вирішує задачу і використовую атомарне представлення, тоді формально знадобиться описати йому мету у вигляді певного списку, можливо досить великого, який задовільнятиме дану умову. Перехід на **розгорнуте представлення**, коли певний стан задається множиною певних параметрів, дозволяє в певній мірі покращити таку ситуацію і дає можливість агентам працювати більш ефективно. Але все одно можливості таких агентів будуть обмежуватись областю визначення конкретної задачі.

Для ще більшого розширення можливостей інтелектуальних агентів розглянемо процес **формування логічних висновків** і використаємо логіку як загальний клас представлення для агентів, які базуються на знаннях. Застосовуючи такий підхід, агенти отримають здатність комбінувати та **рекомбінувати накопичену інформацію**, щоб досягати різної мети. При чому така мета може відрізнятись від поточної і змінюватись в ході роботи. Агенти, які базуються на знаннях, можуть приймати нові задачі, що описані в певній явній формі. Такі агенти можуть доволі швидко досягати компетентності, коли їм надають нові знання про навколишнє середовище або проводять їх навчання. Вони також можуть успішно адаптуватись до змін у навколишньому середовищі, шляхом оновлення відповідних знань.

5.1 Агенти, які базуються на знаннях

Центральним компонентом будь якого агента, який базується на знаннях, є його **база знань** або скорочено ***KB*** *(Knowledge Base)*. База знань представляє собою множину **висловлювань**. Під “висловлюванням” тут розуміється термін, який наближений але не ідентичний до висловлювань у природних мовах. В базі знань кожне висловлювання виражається на мові, яка називається **мовою представлення знань**. Висловлювання представляється у вигляді певного **твердження** про навколишнє середовище. Якщо висловлювання сприймається

як даність, без процесу його виведення із іншого висловлювання, тоді таке висловлювання є **аксіомою**.

Для роботи із базою знань в агента має бути передбачена певні **операції** для **додавання** до бази нових висловлювань, а також для **запиту** із неї відомої інформації. Такі дві операції будуть складовою процесу формування **логічних висновків**, тобто отримання нових висловлювань із старих, які вже відомі. Логічний висновок повинен задовольняти вимогу того, що будь-яка відповідь на запит інформації із бази знання, має слідувати виключно з інформації, яка на поточний момент присутня або додана в базу знань.

В загальному вигляді програма подібного агента, який базується на знаннях включатиме три основні етапи:

1. **додавання** до бази знань результатів поточного сприйняття середовища датчиками агента;

2. **запит** із бази знань інформації про те, яку чергову дію слід виконати. В процесі пошуку відповіді на запит можуть бути проведений певні **процедури судження** про поточний стан середовища, наслідки та результати певних дій;

3. **обрання** дії та додавання до бази знань факту виконання агентом обраної дії.

Описані вище етапи в програмі агента в певній мірі робить його подібним до агента, який використовує модель для представлення внутрішніх станів. Однак, враховуючи описані операції для роботи із базою знань, програми агентів, які базуються на знаннях, вже не можна розглядати як лише програму для обчислення чергової дії. Така програма починає працювати вже на вищому **рівні знань**, коли для визначення поведінки та дій агента потрібно лише певні **знання**, які будуть відомі агенту, та його **мету**. Наприклад, агенту, який керує автомобілем, може бути поставлена мета досягти пункту Д. При чому цей агент може знати або йому додадуть до бази знань факт, що пункт Г є безальтернативним проміжним пунктом на шляху між початковим і кінцевими пунктами. В такому випадку можна очікувати, що агент включить цей проміжний пункт до свого маршруту тому, що він знатиме про відсутність

інших маршрутів до кінцевого пункту. При чому такий аналіз дій агента не буде залежати від того, як агент діє на рівні реалізації.

Агента, який базується на знаннях, можна створити шляхом передачі йому знань потрібних для виконання задачі. Розпочавши із пустої бази знань, розробник агента може почергово передавати йому висловлювання, поки агент не отримає повні знання про те, як йому слід діяти у навколишньому середовищі. Описаний варіант називається **декларативним підходом** при побудові такого агента. Напротивагу йому є **процедурний підхід**, коли розробник кодую бажану поведінку в програмному забезпеченні агента. Наразі, при конструюванні агентів використовується **комбінування** декларативних та процедурних підходів. Агентів, що базуються на знаннях, також можна забезпечити механізмами для **навчання**. Це теоретично дозволить створити агента, який буде самостійно навчатись, і він зможе стати повністю автономним.

5.2 Приклад тестового навколишнього середовища

Для того, щоб продемонструвати роботу агента, який базується на знаннях, розглянемо приклад простої **гри із пошуком скарбу**. Навколишнє середовище для гри буде складатись із лабіринту в якому випадковим чином будуть розмішуватись пастки — ями, в які може потрапити агент, а також хижий звір, якого також краще оминати. Десь в лабіринті буде заховано скарб, який матиме знайти агент. Хоча це доволі проста гра, вона дозволить продемонструвати основні моменти процесу формування **логічних висновків,** які використовуватиме інтелектуальний агент, який базуються на знаннях, для обрання своїх дій.

5.2.1 Умови навколишнього середовища

Основні умови навколишнього середовища (рис. 5.1):

1. **Показники продуктивності.** За знаходження скарбу агент отримує 1000 балів, за потрапляння у пастку або до лап звіра втрачає 1000 балів. Кожна дія агента коштує 1 бал. Гра завершується коли агент потрапляє у пастку або покидає лабіринт (початкова позиція 1,1);

2. **Навколишнє середовище.** Лабіринт складається із решітки 4x4. Агент завжди розпочинає рух з квадрату (1,1), і направлений праворуч. Місце знаходження скарбу, пасток та звіра обирається випадковим чином. Квадрат (1,1) завжди безпечний; Агент програє якщо потрапляє до пастки або лап звіра; Звір не переміщується по лабіринту.

3. **Виконавчі механізми.** Агент може “рухатись” у напряму куди він направлений. Повертати на 90 градусів “ліворуч” та “праворуч”. Може “взяти скарб” у квадраті де він знаходиться та “вийти” із лабіринту у початковому квадраті (1,1).

4. **Датчики сприйняття.** Агент володіє чотирма датчиками:

- В суміжних квадратах (але не по діагоналі) із квадратом із хижим звіром агент чує його “гарчання”;

- В суміжних квадратах із пасткою — ямою агент відчуває рух повітря “вітер”;

- В квадраті де знаходиться скарб, агент бачить “блиск”;

- Коли агент натикається на стіну, він фіксує “удар”.

В результаті процесу сприйняття агента формується список із чотирьох символів, які відповідають наведеній вище інформації із датчиків.

Описане середовище можна класифікувати як **дискретне, детерміноване, статичне** та **одно агентне**. Воно також є **послідовним**, оскільки знаходження скарбу вимагає виконання певної послідовності пов’язаних дій. Середовище є таким, яке можна **спостерігати частково**, бо можна сприймати його аспекти лише у місці перебування агента. Також середовище є **небезпечним** для дослідження, оскільки чергова дія агента може стати невідворотною, в разі його потрапляння у пастку. Тому, агент має **досліджувати** середовище **обережно**, враховуючи інформацію зі своїх датчиків. В такому випадку виявлені квадрати

із небезпеками будуть **доповнювати базу знань** агента і дозволять йому досягти поставленої мети.

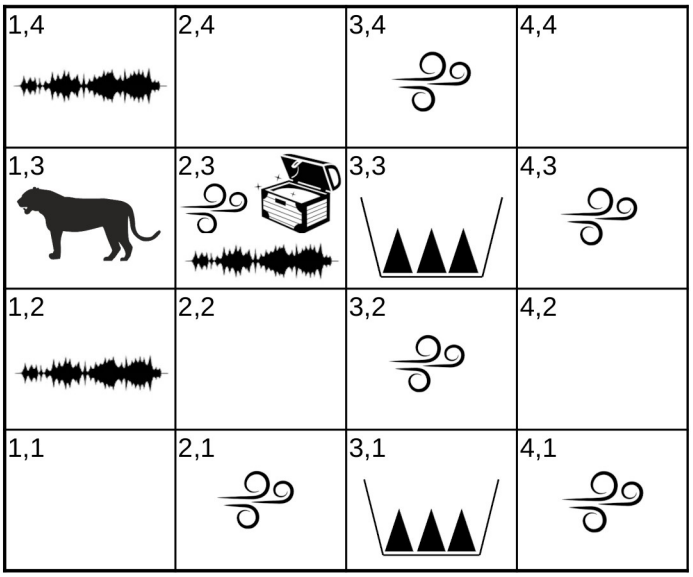


Рисунок 5.1 — Приклад навколишнього середовища-лабіринт в якому діє інтелектуальний агент

Для агента в такому середовищі головною проблемою є його початкова **необізнаність** про конфігурацію навколишнього середовища. Щоб подолати цю необізнаність, агенту, що базується на знаннях, потрібно проводити певний **процес судження**, щоб отримувати **логічні висновки** та обирати свої дії.

5.2.2 Моделювання поведінки обережного агента

Тепер розглянемо як такий агент буде діяти в описаному середовищі. Для цього використаємо неформальну мову представлення знань у вигляді запису спеціальних позначок у квадрати лабіринту.

Агент (A) розпочинає з квадрату (1,1). За умовою він є безпечним, тому отримує позначку (OK). Перше сприйняття агента в даному квадраті є пустим, з чого слідує, що суміжні квадрати (1,2) та (2,1) є безпечними, тому вони отримують позначку (OK). Обережний агент буде переміщуватись у квадрати, які відомі як абсолютно безпечні. Припустимо, що агент перемістився у квадрат

(2,1). Тут за допомогою своїх датчиків сприйняття він відчуває “вітер”. Квадрат отримує позначку (В), а суміжні квадрати (2,2) та (3,1) як ті, які потенційно мають пастку — яму (Я?). На даному етапі (рис. 5.2) єдиним абсолютно безпечним, із позначкою (ОК), і невідвіданим квадратом відомим агенту залишається (1,2), тому агент повертається і прямує до нього.

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 ОК	2,2 Я?	3,2	4,2
1,1 ОК	2,1 В (А) ОК	3,1 Я?	4,1

Рисунок 5.2 — Приклад 1-го етапу процесу обрання дій агентом

В цьому квадраті агент чує “гарчання”, що свідчить про наявність звіра у суміжних квадратах, тому даний квадрат отримує позначку (Г).

На цьому етапі (рис. 5.3) починається певний **процес судження** і отримання **логічних висновків** агентом, базуючись на наявних знаннях. А саме те, що в квадраті (1,1) не може знаходитися звір, оскільки це суперечить правилам, а також факту, що квадрат вже має позначку (ОК). В квадраті (2,2) також не може його бути, бо інакше агент почув би “гарчання” в квадраті (2,1). Відповідно агент може із абсолютною впевненістю стверджувати, що звір знаходиться в квадраті (1,3), оскільки це єдиний квадрат, що залишився і відповідає правилам і інформації сприйняття. Отже, квадрат (1,3) за результатом логічного висновку отримує позначку звір (З!). Крім того, слід відмітити, що агент не відчуває “вітер” у квадраті (1,2). Це свідчить, що в квадратах (1,3), (2,2) та (1,1) не має пастки. Але на попередньому етапі квадрат (2,2) вже отримав

позначку (Я?) як такий, що потенційно має пастку. Враховуючи наявні на даному етапі знання агента, він може зробити **логічний висновок**, що у квадраті (2,2) не має ями і змінити його позначку на (ОК). Додатково агент може зробити логічний висновок і з абсолютною впевненістю стверджувати, що тепер яма знаходиться у квадраті (3,1) і змінити його позначку на (Я!).

1,4	2,4	3,4	4,4
1,3 З!	2,3	3,3	4,3
1,2 Г ОК	2,2 ОК	3,2	4,2
1,1 ОК	2,1 В ОК	3,1 Я!	4,1

Рисунок 5.3 — Приклад 2-го етапу процесу обрання дій агентом

Це доволі складний логічний висновок, оскільки він об’єднує знання, отримані агентом у різний час та різних місцях, а також покладається на відсутність сприйняття в даний момент, при обрані наступної дії. Тепер агенту відомий ще один абсолютно безпечний квадрат (2,2) до якого він переміщується. Щоб не затягувати ілюстрацію (рис. 5.4), будемо вважати, що в даному квадраті агент нічого не сприйняв і направився у квадрат (2,3). Там агент почув “гарчання”, відчув “вітер” і побачив “блиск”. Згідно мети, поставленої перед агентом, він “бере” скарб та покидає лабіринт відомим йому безпечним шляхом.

1,4	2,4	3,4	4,4
1,3 З!	2,3 Г В **Б** (А) ОК	3,3	4,3
1,2 Г ОК	2,2 ОК	3,2	4,2
1,1 ОК	2,1 В ОК	3,1 Я!	4,1

Рисунок 5.4 — Приклад 3-го етапу процесу обрання дій агентом

Слід відмітити, що в кожний конкретний момент, коли агент робить **логічний висновок**, враховуючи наявну в нього інформацію і знання, цей висновок гарантовано буде вірним при умові наявності **вірної інформації**. Це фундаментальна властивість процесу логічних суджень.

5.3 Логічне представлення

Раніше вже зазначалось, що база знань складається із висловлювань. Ці висловлювання мають виражатись у відповідності до **синтаксису мови** представлення. Тепер розглянемо ці основні деталі логічного представлення на прикладів із звичайною арифметикою. Поняття синтаксису можна проілюструвати на прикладі звичайного виразу “x + y = 4”. Відповідно до синтаксису прийнятого в арифметиці це правильне висловлювання. Напротивагу, “x2y=+” — не вірне.

Логіка також має визначати **семантичний сенс** мови або яке значення несе висловлювання. Семантика мови визначає **істинність** кожного висловлювання стосовно певних аспектів середовища або його станів. Наприклад, висловлювання у вигляді виразу “x + y = 4” істинне при “x = 2”, та “y = 2”. Але воно також може бути хибним, коли “x = 1”, та “y = 1”. В

стандартній логіці кожне висловлювання має бути **істиною** для кожного можливого стану або аспекту навколишнього середовища. В ній не може бути **проміжних** варіантів, **неточностей** або **невідповідності**. В цілому під поняттям “стани або аспекти навколишнього середовища” розуміється певна **модель**. Під терміном модель розуміється певна **математична абстракція**, яка встановлює **істинність** або **хибність** для кожного висловлювання, яке стосується даної моделі. Різні моделі, в певній мірі відображають різні варіанти станів та аспектів реальних навколишніх середовищ, в який може перебувати агент. Наприклад, якщо певне висловлювання є істиною в певній моделі, тоді вважається, що ця певна модель задовільняє це певне висловлювання. Процес **логічного судження** полягає у формуванні певних відношень між висловлюваннями. Такі відношення формуються, коли з одного висловлювання може **логічно слідувати** інше. Наприклад, із висловлювання “x = 0” слідує висловлювання “xy = 0”. Очевидно, що у будь-якій моделі де “x = 0”, при будь-якому значенні “y”, вираз “xy” завжди дорівнюватиме нулю.

5.4 Перевірка по моделям

Одним із алгоритмів, який можна використовувати у процесі **логічних суджень**, є алгоритм із **перевіркою по моделям.** В ньому виконується **перебір** всіх можливих моделей для перевірки того, що певне **висловлювання** є істиною в усіх моделях, в яких **база знань** агента також є істиною. Для ілюстрації застосуємо даний алгоритм до згаданого прикладу гри із лабіринтом. Розглянемо ситуація, коли агент відмітив, що квадрат (1,1) є безпечним, а перейшовши до квадрату (2,1) відчув “вітер”, що свідчить про наявність поряд пастки — ями. Це сприйняття, наряду із відомими правилами гри, складає базу знань агента. Припустимо, що агенту потрібно з’ясувати чи присутня пастка у квадратах (1,2), (2,2), (3,1). В кожному з цих квадратів може знаходитись або не знаходитись пастка. Отже для такої ситуації **існує 8** можливих **моделей**.

Базу знань агента можна розглядати як сукупність висловлювань або як одне висловлювання, яка стверджує всі окремі висловлювання. База знань буде хибною або помилковою в моделях, які протирічать поточним знанням агента. Наприклад, наявна база знань агента буде хибною в будь-якій моделі в якій квадрат (1,2) містить пастку, оскільки в квадраті (1,1) агент не сприймав “вітер”. Отже таке висловлювання протирічить базі знань. Фактично існує лише **три моделі**, в яких база знань агента є **істинною**. На ілюстрації вони виділені **зеленим**. Тепер розглянемо перевірку двох можливих висловлювань (рис. 5.5):

1. В квадраті (1,2) не має пастки;
2. В квадраті (2,2) не має пастки.

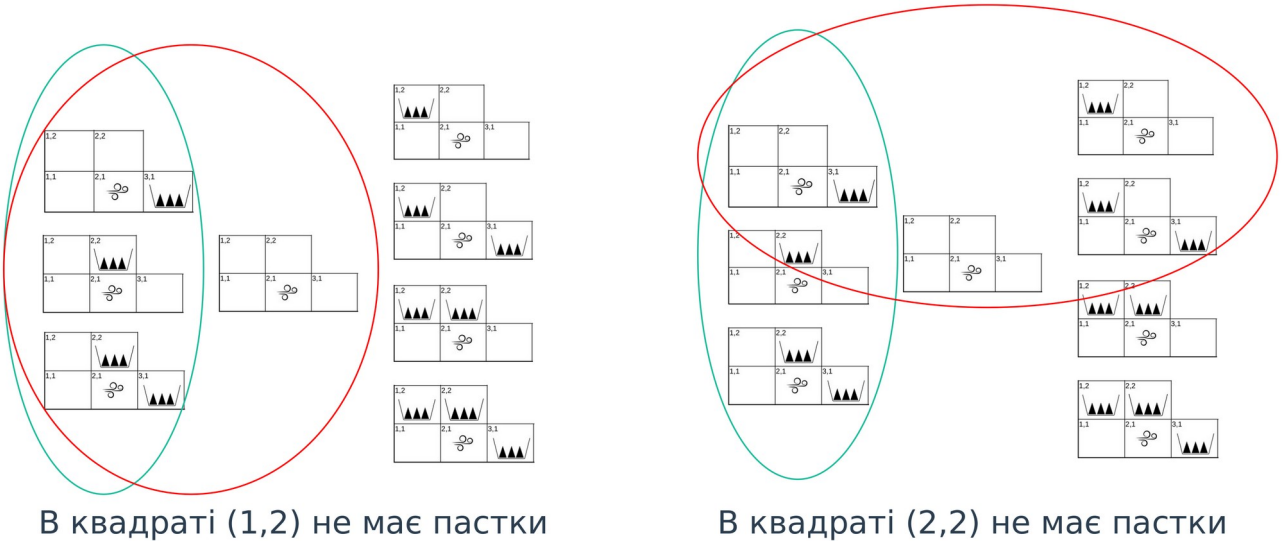


Рисунок 5.5 — Визначення істинності двох висловлювань за алгоритмом із перевіркою по моделям

Моделі, які задовільняють **перше та друге** висловлювання виділені **червоним**. Проаналізувавши ситуацію, можна встановити, що в кожній моделі, в якій **база знань** є **істинною**, **перше** висловлювання також є **істиною** (рис. 5.5 лівий). Відповідно, із **бази знань** агента слідує, що в квадраті (1,2) не має пастки. Також можна побачити, що в деяких моделях, в який **база знань** є **істиною**, **друге** висловлювання є **хибним** (рис. 5.5 правий). Відповідно, із бази знань не слідує, що в квадраті (2,2) не має пастки. Також не можна зробити логічний висновок, базуючись на наявних знаннях, що квадраті (2,2) є пастка.

На даному прикладі було продемонстровано процес **логічного судження**, а також як логічні **слідування між висловлюваннями** застосовуються для отримання **логічного висновку** за допомогою **алгоритму перевірки по моделям**.

Алгоритм логічного висновку називається таким, який **не має протиріччя**, коли він дозволяє отримати такі висловлювання, які дійсно **слідують** із **бази знань**. Також алгоритм логічного висновку є **повним,** якщо він дозволяє вивести будь-яке висловлювання, яке слідує із **бази знань**.

Описаний процес логічних суджень формував висновки, які є гарантовано істинними в будь-якому середовищі, в якому істині передумови. Тобто, якщо база знань є істинною в реальному середовищі, тоді будь-яке висловлювання, отримане логічним шляхом з цієї бази даних за допомогою процедури логічних висновків **без протиріч**, також буде істинною в реальному середовищі. Тому, незважаючи на те, що процес логічного судження оперує певним синтаксисом, цей процес **відповідає зв’язкам** у реальному середовищі.

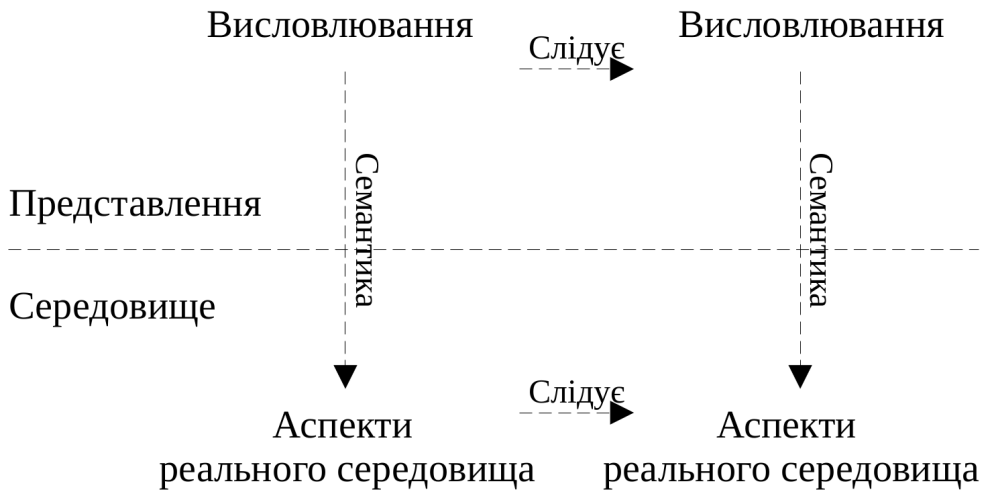


Рисунок 5.6 — Взаємозв’язок представлення агента та реального середовища

Останній момент в процесі логічних суджень є **обґрунтування.** Воно встановлює відповідність між процесами логічного судження та реальним середовищем, в якому працює агент. Може постати питання: “як дізнатись, що база знань агента є істинною в реальному середовищі”. Однією із відповідей на

дане питання є те, що **цей зв’язок** із реальним середовищем агент встановлює за допомогою своїх датчиків. Коли агент отримує певну інформацію сприйняття зі своїх датчиків, він додає цю інформацію, у вигляді висловлювання, у свою базу знань. Відповідно, коли дане висловлювання знаходиться у базі знань, воно є істинним у реальному середовищі. Тому в цілому, **істинність висловлювань**, в яких виражаються результати сприйняття, визначаються процесами формування сприйняття та створення висловлювання на основі цього сприйняття. Простими словами, істинність таких висловлювань залежить від точності датчиків та правильності інтерпретації даних з цих датчиків.

ТЕМА 6
ЛОГІКА ВИСЛОВЛЮВАНЬ

В даній темі розглянемо більш **формалізований** процес **логічного судження** та формування **логічних висновків**. Для цього використаємо **просту форму** логіки, яка називається **логікою висловлювань** або обчисленням висловлювань. Також буде розглянуто її **синтаксис** — структуру висловлювань та **семантику** — спосіб визначення істинності висловлювань. Після цього розглянемо **синтаксичний алгоритм логічного висновку**, який реалізує **семантичне** поняття **логічного наслідку**. Для ілюстрації будемо використовувати приклад лабіринту з минулої теми.

6.1 Синтаксис

Синтаксис логіки висловлювань визначає допустимі форми висловлювань. **Атомарні висловлювання** є **неподільними** синтаксичними елементами. Вони складаються із одного **символу** у вигляді змінної. Кожен такий символ — змінна визначає **висловлювання**, яке може бути або істинним, або хибним. Зазвичай для запису таких символів використовують імена або букви, іноді із індексами. Такі символи обираються довільно або частіше вони мають певне **мнемонічне** значення. Наприклад, для прикладу лабіринту символ “Я” буде означати присутність ями. Додатково можна включити індекс “Я_3,1” для того, щоб конкретизувати, що яма знаходиться в квадраті (3,1). Символ може мати **два значення**: “істина (*True*)” — для висловлювань, які завжди істинні або “хибність (*False*)” — висловлювань, які завжди хибні.

Складні висловлювання будуються із більш простих висловлювань за допомогою **логічних зв’язків,** використовуючи дужки та **логічні оператори**. Широко застосовуються п’ять логічних операторів:

1. **Заперечення**. Таке висловлювання $\neg Я_{3,1}$ або $\overline{Я_{3,1}}$ (риска зверху) називається запереченням — **логічне “НЕ”** висловлювання $Я_{3,1}$;

2. **Кон’юнкція**. Висловлювання, яке базується на зв’язці, наприклад, $Я_{3,1} \wedge B_{2,1}$ називається кон’юнкцією і є **логічним “І”**. Його частини називаються кон’юнктами;

3. **Диз’юнкція**. Висловлювання, в якому використовується зв’язка **логічне “АБО”**, наприклад, $(Я_{3,1} \wedge З_{1,3}) \vee З_{2,2}$. Його частини називаються диз’юнктами;

4. **Імплікація.** Таке висловлювання, як $(Я_{3,1} \wedge З_{1,3}) \Rightarrow \neg З_{2,2}$ називається імплікацією або **слідуванням**. Його **передумовою** є $Я_{3,1} \wedge З_{1,3}$, а $\neg З_{2,2}$ є **наслідком**. Імплікація також називається **правилом “якщо-тоді”**. В мовах програмування воно задається конструкцією *if-then*.

5. **Двостороння імплікація.** Висловлювання $З_{1,3} \Leftrightarrow \neg З_{2,2}$ називається двосторонньою імплікацією або **еквівалентністю**. Таке висловлювання буде істинним тоді і тільки, коли **обидві** його частини є **істинними**, або **обидві** будуть **хибними**.

6.2 Граматика

Формальна граматика логіки висловлювань приведена у формі **нотацій Бекуса-Наура**. Це формальна система опису синтаксису, коли одна синтаксична категорія послідовно визначається через інші.

$$
\begin{array}{lll}
\text{Висловлювання} & ::= & \text{АтомарнеВисловлювання} \mid \text{СкладневисловлюванняВисловлювання} \\
\text{АтомарнеВисловлювання} & ::= & \text{Істинна} \mid \text{Хибність} \mid \text{P} \mid \text{Q} \mid \text{R} \ldots \\
\text{СкладневисловлюванняВисловлювання} & ::= & \textit{Висловлювання} \\
 & & \mid \neg \textit{Висловлювання} \\
 & & \mid \text{Висловлювання} \wedge \textit{Висловлювання} \\
 & & \mid \text{Висловлювання} \vee \textit{Висловлювання} \\
 & & \mid \text{Висловлювання} \Rightarrow \textit{Висловлювання} \\
 & & \mid \text{Висловлювання} \Leftrightarrow \textit{Висловлювання}
\end{array}
\qquad (6.1)
$$

Порядок виконання операцій : $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$

Ця граматика доповнена списком, який визначає **порядок виконання** операторів і дозволяє усунути невизначеність при наявності у виразі декількох операторів. Оператор **логічного “НЕ”** має найбільший пріоритет, тому

висловлювання виду $\neg З_{2,2} \wedge Я_{3,1}$ слід розуміти як $(\neg З_{2,2}) \wedge Я_{3,1}$, а не $\neg(З_{2,2} \wedge Я_{3,1})$. Іншою формою запису для подібного висловлювання є $\overline{З_{2,2}} \wedge Я_{3,1}$.

6.3 Семантика

Після визначення синтаксису та граматики логіки висловлювань дамо визначення семантиці. Семантика задає **правила** для визначення **істинності висловлювань** по відношенню до конкретної **моделі**. В логіці висловлювань будь-яка модель просто **фіксує значення** для кожного символу: “істина” або “хибність”. Для істинних висловлювань будемо вважати, що вони мають значення “*true*”, для хибних висловлювань — “*false*”. Для прикладу розглянемо гру з лабіринтом. Якщо в висловлюваннях бази знань використовується символи $Я_{1,2}$, $Я_{2,2}$, $Я_{3,1}$, тоді однією із можливих моделей буде: $m_1 = \{Я_{1,2} = false, Я_{2,2} = false, Я_{3,1} = true\}$ (рис. 6.1). При наявності трьох символів існує 8 можливих моделей, що відповідає числу моделей, які розглядались минулої теми. Однак тут слід відмітити що, оскільки вже визначено синтаксис, ці моделі вже стали чисто математичними об’єктами, які не обов’язково мають бути пов’язані із цим прикладом гри.

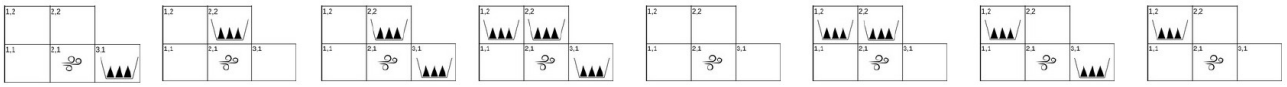


Рисунок 6.1 — Можливі моделі при наявності трьох символів

Семантика логіки висловлювань повинна визначати, як слід обчислювати істинність значення будь-якого висловлювання при наявній моделі. Ця процедура виконується рекурсивно. Всі висловлювання формуються із **атомарних висловлювань** та **п’яти операторів,** тому спочатку слід зазначити як обчислювати істинність атомарних висловлювань, а потім — як обчислювати істинність висловлювань, сформованих за допомогою кожного з п’яти

операторів. Задача обчислення істинності атомарного висловлювання є доволі простою. Наприклад:

1. Висловлювання “*true*” завжди істинне в будь-якій моделі, “*false*” — завжди хибне;

2. Фактичне значення будь-якого символу, який представляє певне висловлювання, повинно буди безпосередньо визначено в моделі. Наприклад, в наведеній моделі (рис. 6.1, крайній лівий) m_1 висловлювання $Я_{1,2}$ є хибним.

Для складних висловлювань існує **п’ять правил**, які справедливі для будь-якого вхідного висловлювання *P* та *Q* (атомарного або складного) в будь-якій моделі *m*.

1. $\neg P$ має значення “істини” тоді і тільки тоді, коли *P* є хибним в моделі *m;*

2. $P \wedge Q$ - “істина” тоді і тільки тоді, коли і *P,* і *Q* є істинними в моделі *m;*

3. $P \vee Q$ - “істина” тоді і тільки тоді, коли або *P,* або *Q* є істинними в моделі *m;*

4. $P \Rightarrow Q$ - “істина” за виключенням випадку, коли *P* “істина”, а *Q* “хибне” в *m;*

5. $P \Leftrightarrow Q$ - “істина” тоді і тільки тоді, коли обидва *P* і *Q* є істинними або хибними в моделі *m.*

6.4 Таблиця істинності

Ці правила також можна представити у вигляді таблиці. За допомогою цієї таблиці істинності можна розрахувати значення для певного висловлювання по відношенню до певної моделі *m*. Для ілюстрації розглянемо висловлювання $\neg Я_{1,2} \wedge (Я_{2,2} \vee Я_{3,1})$, яке розраховується в моделі m_1 , і як результат воно матиме наступне значення: $\neg false \wedge (false \vee true) = true \wedge (false \vee true) = true \wedge true = true$.

Таблиця 6.1 — Значення істинності для логічних операторів

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Таблиця істинності для операторів логічного **“І”**, **“АБО”** та **“НЕ”** майже повністю відповідає інтуїтивному розумінню таких же слів в природній мові. Однією із можливих ситуацій, яка може неоднозначно розумітись, є логічне “АБО” де висловлювання $P \vee Q$ є “істинним”, коли P або Q, або вони обидва мають “істинне” значення. Є ще один логічний оператор, який називається **“виключне АБО**”. Вона часто пишеться наступним чином $P \oplus Q$ і приймає “хибне” значення, коли обидва P та Q є істинними. В певній мірі запис цього логічного оператора в інший спосіб $P \neq Q$ дозволяє покращити розуміння принципу роботи даного оператора. Висловлювання із **двосторонньою імплікацією** $P \Leftrightarrow Q$ буде “істиною”, якщо обидва висловлювання $P \Rightarrow Q$ та $Q \Rightarrow P$ будуть “істинами”.

Висловлювання із двосторонньою імплікацією дозволяють описати більшість правил із гри з лабіринтом. Розглянемо ситуацію із гри, коли в квадраті відчувається “вітер”, якщо у сусідніх квадратах присутня пастка-яма. Відповідно матимемо висловлювання, “вітер” в деякому квадраті може бути, **тоді і тільки тоді** коли в сусідньому квадраті є яма. Із цього твердження зрозуміло, що для нього добре підходить саме **двостороння імплікація.** Будемо мати запис: $B_{1,1} \Leftrightarrow (Я_{1,2} \vee Я_{2,1})$. Може постати питання, чому використовується **логічне “АБО”** для задання наявності ями в сусідніх квадратах. Адже за правилами гри, пастка може знаходитись в обох квадратах. Тут слід відмітити, що це висловлювання більше стосується наявності “вітру”, а не кількості пасток-ям в сусідніх квадратах. І, якщо згадати таблицю істинності, можна побачити, що навіть при наявності ям в обох сусідніх квадратах, права частина висловлювання зі зв’язкою “АБО” буде “істиною”. Тому ніяких протиріч, стосовно цього оператору не виникає. А із самого висловлювання в цілому

слідує, що для того щоб в квадраті відчувався “вітер” потрібно, щоб хоча б в одному суміжному квадраті була яма. Але допустима ситуація із наявністю ям у всіх сусідніх квадратах.

6.5 Проста база знань

Тепер після визначення семантики логіки висловлювань, можна розпочати формувати базу знань інтелектуального агента для прикладу гри із лабіринтом та пошуком скарбу. На початковому етапі потрібно задати для кожного квадрату висловлювання стосовно правил гри. Відповідно до аспектів, що описуватимуть правила, оберемо перші їх букви в якості символів для позначення висловлювань.

1. Висловлювання $Я_{x,y}$ є “істиною”, якщо в квадраті (x,y) присутня яма;

2. Висловлювання $З_{x,y}$ є “істиною”, якщо в квадраті (x,y) присутній хижий звір;

3. Висловлювання $В_{x,y}$ є “істиною”, якщо в квадраті (x,y) відчувається “вітер”;

4. Висловлювання $Г_{x,y}$ є “істиною”, якщо в квадраті (x,y) чується “гарчання” звіра;

5. Висловлювання $А_{x,y}$ є “істиною”, якщо в квадраті (x,y) знаходиться агент.

Даних висловлювань буде цілком достатньо, щоб вивести висловлювання або зробити логічний висновок, наприклад, $\neg Я_{1,2}$ - що у квадраті (1,2) не має ями. Кожне нове висловлювання, яке буде додаватись до бази знань, будемо помічати P_i (від слова “результат”).

Тепер заповнимо базу знань агента про певні аспекти середовища, які він сприймає в ході дослідження середовища, а також аспекти, які відповідають правилам гри. Матимемо наступні висловлювання:

1. $P_1 : \neg Я_{1,1}$ - в квадраті (1,1) не має ями (згідно правил);

2. $P_2: B_{1,1} \Leftrightarrow (Я_{1,2} \vee Я_{2,1})$
$P_3: B_{2,1} \Leftrightarrow (Я_{1,1} \vee Я_{2,2} \vee Я_{3,1})$ - в певному квадраті відчувається “вітер” **тоді тільки тоді**, коли в сусідньому квадраті є яма. Таке висловлювання потрібно сформувати для кожного квадрату у лабіринті. Але на даний момент обмежимось, лише тими квадратами через які перемістився агент.

3. $P_4: \neg B_{1,1}$
$P_5: B_{2,1}$ - висловлювання про сприйняття “вітру” у перших двох квадратах, які пройшов агент.

Після виконання даного етапу маємо базу знань, яка містить висловлювання $P_1 - P_5$ і всі окремі висловлювання в ній є “істинами”.

6.6 Проста процедура логічного висновку

Наразі метою є визначити, чи буде із **бази знань** агента **слідувати** певне висловлювання. Наприклад, чи слідує з бази знань висловлювання $\neg Я_{1,2}$? Перший алгоритм логічного висновку, який ми розглянемо, представляє собою **безпосередню реалізацію** визначення логічного слідування. Він перевіряє кожну модель, щоб визначити чи певне висловлювання є істинним у кожній моделі, в якій база знань є істинною. Для логіки виловлювань моделі представляються у вигляді множини варіантів значень “*true*” або “*false*” для кожного символу у висловлюванні. Згідно нашої бази знань маємо символи: $B_{1,1}, B_{2,1}, Я_{1,1}, Я_{1,2}, Я_{2,1}, Я_{2,2}, Я_{3,1}$. Для цих 7 символів може існувати $2^7 = 128$ можливих моделей.

Таблиця 6.2 — Проста процедура логічного висновку за таблицею істинності

B1,1	*B2,1*	*Я1,1*	*Я1,2*	*Я2,1*	*Я2,2*	*Я3,1*	*P1*	*P2*	*P3*	*P4*	*P5*	*KB*
false	false	false	false	false	false	false	true	true	true	true	false	false
false	false	false	false	false	false	true	true	true	false	true	false	false
…	…	…	…	…	…	…	…	…	…	…	…	…
false	true	false	false	false	false	false	true	true	false	true	true	false
false	true	false	false	false	false	true	true	true	true	true	true	true
false	true	false	false	false	true	false	true	true	true	true	true	true
false	true	false	false	false	true	true	true	true	true	true	true	true
false	true	false	false	true	false	false	true	false	false	true	true	false
…	…	…	…	…	…	…	…	…	…	…	…	…
true	true	true	true	true	true	true	false	true	true	false	true	false

Згідно таблиці, лише для трьох з них база знань (*KB*) буде істинною. В цих трьох моделях істинними є висловлювання $\neg Я_{1,2}$, тому можна зробити логічний висновок, що в квадраті (1,2) не має ями. З іншого боку, висловлювання $Я_{2,2}$ істинне в двох моделях і хибне в одній. Отже з поточною базою знань не можна визначити наявність ями в квадраті (2,2).

Приведений алгоритм не містить **протиріч**, оскільки він безпосередньо реалізує логічне слідування згідно його визначення. Також він є **повним** тому, що може застосовуватись для будь-якої бази знань і будь-якого висловлювання. Він завжди має **скінчену** кількість ітерацій при пошуку рішення при умові, що кількість моделей, які перевіряються, також скінченна.

ТЕМА 7
ДОВЕДЕННЯ ТЕОРЕМ ЛОГІКИ ВИСЛОВЛЮВАННЯ

На поточний момент було показано, як отримати логічний висновок, шляхом перевірки моделей. Для цього виконувалась послідовна перевірка моделей і пошук висловлювань, які присутні в усіх моделях. Тепер розглянемо, як логічний висновок можна зробити через доведення теорем логіки висловлювання. Для цього будемо застосовувати **правила логічно слідування** безпосередньо до висловлювань у базі знань, з метою доведення істинності певного висловлювання без застосування моделей. Якщо кількість моделей є достатньо великою, а довжина доказу — обмежена, тоді доведення теорем може стати **більш ефективним** методом, порівняно із перевіркою по моделям.

7.1 Логічні еквівалентності

Перед розглядом алгоритму доказу логічних висловлювань, давайте розглянемо декілька додаткових концепцій, які використовуються в процесі формування логічного наслідку. Першим є поняття **логічної еквівалентності** — два висловлювання є логічно еквівалентними, якщо вони мають значення істинності в **одній і той самій** множині моделей. Це твердження записується у наступному вигляді $P \wedge \neg P$. Наприклад, можна легко показати, що висловлювання $P \wedge Q \equiv Q \wedge P$ - еквівалентні за допомогою таблиці істинності із минулої теми. В логіці подібні еквівалентності відіграють практично таку саму роль, яку арифметичні рівності відіграють в звичайній математиці. Ще одним визначенням поняття еквівалентності є те, що будь-які два висловлювання є еквівалентними тоді і тільки тоді, коли кожне з них є логічним наслідком іншого.

$$
\begin{aligned}
&(A \wedge B) = (B \wedge A) : \textit{комутативність логічного } I \\
&(A \vee B) = (B \vee A) : \textit{комутативність логічного АБО} \\
&(A \wedge B) \wedge C = A \wedge (B \wedge C) : \textit{асоціативність логічного } I \\
&(A \vee B) \vee C = A \vee (B \vee C) : \textit{асоціативність логічного АБО} \\
&\neg(\neg A) = A : \textit{усунення подвійного заперечення} \\
&A \Rightarrow B = \neg A \vee B : \textit{усунення імплікації} \\
&A \Leftrightarrow B = (A \Rightarrow B) \wedge (B \Rightarrow A) : \textit{усунення подвійної імплікації} \\
&\neg(A \wedge B) = \neg A \vee \neg B : \textit{правило де Моргана} \\
&\neg(A \vee B) = \neg A \wedge \neg B : \textit{правило де Моргана} \\
&A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C) : \textit{дистрибутивність логічного } I \textit{ по АБО} \\
&A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C) : \textit{дистрибутивність логічного АБО по } I
\end{aligned}
\qquad (7.1)
$$

7.2 Поняття допустимості та здійсненності

Важливим визначення, яке нам знадобиться, це **допустимість.** Висловлювання є допустимим, якщо воно є істинним **в усіх моделях**. Наприклад, висловлювання $P \wedge \neg P$ є допустимим. Допустимі висловлювання також відомі під назвою **тавтології** — яка обов'язково є істинною.

Також розглянемо поняття **здійсненості**. Деяке висловлювання є здійсненним тоді і тільки тоді, коли воно істинне в **деякій моделі**. Сформована минулої теми база знань - $P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5$ є здійсненною, оскільки існує не одна, а навіть три моделі, в яких ця база знань є істинною. Здійсненність можна перевірити, перебираючи всі можливі моделі до тих пір, поки не буде знайдена хоча б одна модель, яка задовільняє це висловлювання.

Безумовно, поняття **допустимості** та **здійсненності** пов'язані один з одним. Висловлювання P **є допустимим** тоді і тільки тоді, коли висловлювання $\neg P$ - **нездійсненно.** І навпаки, висловлювання P є здійсненним тоді і тільки тоді, коли висловлювання $\neg P$ - недопустиме.

Доведення висловлювання Q базуючись на істинності висловлювання P полягає у **перевірці нездійсненності** виразу $(P \wedge \neg Q)$. Такий підхід відповідає стандартному математичному методу доведення шляхом **приведення до абсурду**. Іншою назвою цього методу є **доведення шляхом спростування** або **доведення за допомогою виявлення протиріч**. За даним методом доведення робиться припущення, що висловлювання Q є хибним і робиться

спроба показати, що таке припущення призводить до протиріч із відомими **аксіомами**. Наприклад, таким чином можна довести, що в квадраті (2,2) не має ями при відсутності сприйняття вітру у квадраті (1,2).

$$\begin{gathered} P = \neg B_{1,2} \\ Q = \neg Я_{2,2} \\ \left(\neg B_{1,2} \wedge Я_{2,2}\right) \end{gathered} \tag{7.2}$$

7.3 Правила логічного висновку

Для того, щоб можна було формувати логічні висновки розглянемо правила логічного висновку, які можуть застосовуватись для формування ланцюжка висновків, які дозволять досягти мети, а саме довести певне висловлювання. Найбільш широко застосовуваним для цього процесу правилом є **правило відділення**

$$\frac{P \Rightarrow Q, P}{Q} \tag{7.3}$$

Цей запис означає, що в разі якщо дано будь-які два висловлювання $P \Rightarrow Q$ та, P тоді з них можна вивести нове висловлювання Q . Наприклад, дано два висловлювання:

(1) відчувається “вітер” “І” яма знаходиться у квадраті (1,2) тоді “слідування” йти в квадрат (2,1) та

(2) відчувається “вітер” “І” яма знаходиться у квадраті (1,2).

Із цих двох висловлювання, які знаходяться в базі знань, тобто є істинними, можна вивести висловлювання (3) — йти в квадрат (2,1).

$$\frac{\left(B_{1,1} \wedge Я_{1,2}\right) \Rightarrow A_{2,1}, \left(B_{1,1} \wedge Я_{1,2}\right)}{A_{2,1}} \tag{7.4}$$

Ще одним корисним правилом логічного висновку є правило **видалення зв’язки “І”**. Це правило стверджує, що із кон’юнкції можна вивести будь-який із її кон’юнктів.

$$\frac{P \wedge Q}{P} \tag{7.5}$$

Наприклад, застосувавши дане правило до аксіоми про сприйняття “блиску” при наявності в поточному квадраті скарбу, агент може зробити висновок про присутність скарбу в квадраті де він сприйняв “блиск”:

$$Б_{2,3} \Leftrightarrow C_{2,3} = (Б_{2,3} \Rightarrow C_{2,3}) \wedge (C_{2,3} \Rightarrow Б_{2,3})$$
$$\frac{(Б_{2,3} \Rightarrow C_{2,3}),\ Б_{2,3}}{C_{2,3}} \qquad (7.6)$$

В якості правил логічного висновку також можна використовувати всі **логічні еквівалентності**. Наприклад, із **еквівалентності усунення двосторонньої імплікації** можна отримати два правила логічного висновку:

$$\frac{P \Leftrightarrow Q}{(P \Rightarrow Q) \wedge (Q \Rightarrow P)} \qquad \frac{(P \Rightarrow Q) \wedge (Q \Rightarrow P)}{P \Leftrightarrow Q} \qquad (7.7)$$

Однак не всі логічні правила діють в обох напрямах. Не можна застосувати **правило відділення** в зворотному напряму, щоб отримати два висловлювання $P \Rightarrow Q$ та P лише з одного висловлювання Q .

7.4 Процес формування логічного висновку

Тепер застосуємо розглянуті правила та еквівалентності і розглянемо як можна їх використати для прикладу гри із лабіринтом. В результаті ми зможемо отримати нові висловлювання, які відображатимуть певний аспект навколишнього середовища. Почнемо із бази знань, яка містить п’ять висловлювань, доданих минулої теми. Їх можна використати, щоб довести інше нове висловлювання $\neg P_{1,2}$, тобто що в квадраті (1,2) не має ями.

1. Спочатку застосуємо правило **усунення двосторонньої імплікації** до висловлювання (2), щоб отримати наступне:

$$P_6 : (B_{1,1} \Rightarrow (Я_{1,2} \vee Я_{2,1})) \wedge ((Я_{1,2} \vee Я_{2,1}) \Rightarrow B_{1,1}) \qquad (7.8)$$

2. Далі застосуємо до нового висловлювання (6) правило **видалення зв’язки “І”:**

$$P_7 : (Я_{1,2} \vee Я_{2,1}) \Rightarrow B_{1,1} \qquad (7.9)$$

3. Із логічної **еквівалентності заперечення** слідує

$$P_8 : \neg B_{1,1} \Rightarrow \neg (Я_{1,2} \vee Я_{2,1}) \qquad (7.10)$$

4. Тепер можна застосувати **правило відділення** до висловлювання (8) і до висловлювання (4) — стосовно сприйняття вітру:

$$\frac{\neg B_{1,1} \Rightarrow \neg(Я_{1,2} \vee Я_{2,1}), \neg B_{1,1}}{\neg(Я_{1,2} \vee Я_{2,1})} \tag{7.11}$$

$$P_9 : \neg(Я_{1,2} \vee Я_{2,1}) \tag{7.12}$$

5. Нарешті, застосовуємо **правило де Моргана,** яке дозволяє отримати такий логічний висновок:

$$P_{10} : \neg Я_{1,2} \wedge \neg Я_{2,1} \tag{7.13}$$

В результаті цього процесу було доведено, що (1) “в квадраті (1,2) не має ями” та (2) “в квадраті (2,1) не має ями”.

Таким чином, пошук доказу шляхом формування логічного висновку, використовуючи **логічні правила та еквівалентності**, можна застосовувати як альтернативу методу із перевіркою по моделям. В багатьох практичних випадках **пошук доказу** може виявитись більш ефективним методом, оскільки в процесі доведення можна ігнорувати всі нерелевантні висловлювання, незважаючи на скільки багато їх мається. Для попереднього прикладу доведення, яке привело нас до нового висловлювання $\neg Я_{1,2} \wedge \neg Я_{2,1}$, не використовувались наявні у базі знань висловлювання про $B_{2,1}, Я_{1,1}, Я_{2,2}, Я_{3,1}$. Їх можна не розглядати, оскільки цільове висловлювання — символ $Я_{1,2}$ був присутнім лише в складі висловлювання P_4, а інші символи в складі даного висловлювання містились лише в P_4, P_2. З цього слідує, що висловлювання P_1, P_3, P_5 не мають ніякого відношення до процесу доведення шуканого виловлювання $\neg Я_{1,2}$. Це саме **міркування залишиться справедливим** в разі, якщо база знань буде доповнена новими висловлюваннями. Наприклад, інтелектуальний агент в процесі своєї роботи додасть до бази тисячу нових висловлювань, які відображатимуть певну інформацію, яку він сприйняв або вивів. В такому разі звичайний алгоритм перебору рядків у таблиці істинності — перевірки по моделям стикнеться з експоненційним збільшенням кількості моделей, які треба буде перевіряти. Напротивагу, **процес доведення**, наведеного вище висловлювання, навіть для збільшеної бази знань, залишиться без змін.

Останньою фундаментальною властивістю логічних систем є **монотонність.** Це означає, що множина висловлювань, які можуть бути отримані шляхом формування логічного висновку, буде збільшуватись лише при додаванні до бази знань нової інформації. Наприклад, припустимо, що база знань для лабіринту буде містити нову інформацію про те, що в лабіринті може бути присутньо лише п’ять пасток — ям. Ці знання можуть допомогти агенту прийти до нових додаткових висновків. Але це ні як не може поставити під сумнів вже зроблений висновок, що в квадраті (1,2) не має ями. Монотонність означає, що правило логічного висновку може застосовуватись кожен раз, коли в базі знань виявляється підходящі передумови. В цьому разі отримане результуюче висловлювання буде наслідком застосування даного правила, незалежно від того, яка інша інформація знаходиться в базі знань.

7.5 Доведення методом резолюцій

Всі розглянуті до цього правила логічного висновку були такими, які **не мали протиріч.** Тобто, в ході формування логічного висновку не виникали ситуації, коли було присутньо декілька висловлювань, які могли заперечувати одне одне. Тепер розглянемо правило логічного висновку, а саме **правило резолюцій**. Для цього почнемо із наочного прикладу із базою знань для гри з лабіринтом. Розглянемо етап, коли агент повертається з квадрату (2,1) в (1,1) та переходить у квадрат (1,2). В ньому він отримує інформацію про сприйняття “гарчання”, але не фіксує сприйняття “вітру”. Введемо в базу знань інтелектуального агента ці нові факти. В цій демонстрації обмежимось сприйняттям “вітру”:

1. $P_{11}: \neg B_{1,2}$
2. $P_{12}: B_{1,2} \Leftrightarrow (Я_{1,1} \vee Я_{2,2} \vee Я_{1,3})$

Застосувавши аналогічний процес, який був використаний для отримання висловлювання P_{10} , тепер можна зробити висновок про відсутність ям у квадратах (2,2) та (1,3). За правилами, в квадраті (1,1) не може бути ями.

3. $P_{13}:(B_{1,2}\Rightarrow(Я_{1,1}\vee Я_{2,2}\vee Я_{1,3}))\wedge((Я_{1,1}\vee Я_{2,2}\vee Я_{1,3})\Rightarrow B_{1,2})$ **усунення двосторонньої імплікації**

4. $P_{14}:(Я_{1,1}\vee Я_{2,2}\vee Я_{1,3})\Rightarrow B_{1,2}$ **видалення зв'язки "І"**

5. $P_{15}:\neg B_{1,2}\Rightarrow\neg(Я_{1,1}\vee Я_{2,2}\vee Я_{1,3})$ **еквівалентність заперечення**

6. $\dfrac{\neg B_{1,2}\Rightarrow\neg(Я_{1,1}\vee Я_{2,2}\vee Я_{1,3}),\neg B_{1,2}}{\neg(Я_{1,1}\vee Я_{2,2}\vee Я_{1,3})}$ **правило відділення**

   $P_{16}:\neg(Я_{1,1}\vee Я_{2,2}\vee Я_{1,3})$

7. $P_{17}:\neg Я_{1,1}\wedge\neg Я_{2,2}\wedge\neg Я_{1,3}$ **правило де Моргана**

8. $P_{18}:\neg Я_{2,2}$

   $P_{19}:\neg Я_{1,3}$ **виведення будь-якого висловлювання із кон'юнкції** (видалення зв'язки "І")

Далі можна застосувати до висловлювання P_3 правило усунення двосторонньої імплікації, після чого застосувати до отриманого результату разом із висловлюванням P_5 правило відділення. Це дозволить встановити той факт, що найменше в одному з квадратів (1,1), (2,2) або (3,1) присутня яма.

9. $P_{20}:(B_{2,1}\Rightarrow(Я_{1,1}\vee Я_{2,2}\vee Я_{3,1}))\wedge((Я_{1,1}\vee Я_{2,2}\vee Я_{3,1})\Rightarrow B_{2,1})$

10. $P_{21}:B_{2,1}\Rightarrow(Я_{1,1}\vee Я_{2,2}\vee Я_{3,1})$

11. $\dfrac{B_{2,1}\Rightarrow(Я_{1,1}\vee Я_{2,2}\vee Я_{3,1}),B_{2,1}}{Я_{1,1}\vee Я_{2,2}\vee Я_{3,1}}$

    $P_{22}:Я_{1,1}\vee Я_{2,2}\vee Я_{3,1}$

Тепер в нас з'явилась можливість скористатись правилом резолюцій або **правилом усунення протиріч.** Висловлювання літерал (символ) $\neg Я_{2,2}$ в висловлювані (18) є протилежним до літералу $Я_{2,2}$ у висловлюванні (22), тому його можна усунути, через що отримаємо наступне нове висловлювання:

12. $P_{23}:Я_{1,1}\vee Я_{3,1}$

Природною мовою такий хід думок можна виразити наступним чином: якщо щонайменше в одному з квадратів (1,1), (2,2) або (3,1) присутня пастка — яма, але її не має в квадраті (2,2), тоді яма повинна бути в квадраті (1,1) або (3,1). Аналогічним чином усувається літерал $Я_{1,1}$ із висловлювання (22), який протирічить літералу $\neg Я_{1,1}$ із висловлювання (1). Матимемо:

13. $P_{24}: Я_{3,1}$

Ці два останні кроки логічного висновку представляють собою приклад правила формування логічного висновку, який називається **одиничною резолюцією:**

$$\frac{L_1 \vee ... \vee L_k, M}{L_1 \vee ... \vee L_{i-1} \vee L_{i+1} \vee ... \vee L_k} \tag{7.14}$$

де кожне L представляє собою літерал, а L_i та M являються **взаємно зворотними** літералами (один заперечує інший). Таким чином, у правилі одиничної резолюції береться **диз'юнкція літералів** і ще один **літерал (протилежний)** після чого формується новий вираз. Окремо слід відмітити, що один літерал може розглядатись як **диз'юнкція з одного літералу.** Це називається **одиничним виразом**. Отже правило одиничної резолюції може бути узагальнене до **повного правила резолюцій**:

$$\frac{L_1 \vee ... \vee L_k, M_1 \vee ... \vee M_n}{L_1 \vee ... \vee L_{i-1} \vee L_{i+1} \vee ... \vee L_k \vee M_1 \vee ... \vee M_{j-1} \vee M_{j+1} \vee ... \vee M_n} \tag{7.15}$$

де L_i та M_j взаємно зворотні літерали. Це означає, що в **правилі резолюцій** беруться два вирази і формується новий вираз, який містить всі літерали двох початкових виразів, за винятком двох взаємно зворотних літералів.

Наприклад, розглянемо застосування даного правило для гри лабіринту:

$$\frac{Я_{1,1} \vee Я_{3,1}, \neg Я_{1,1} \vee \neg Я_{2,2}}{Я_{3,1} \vee \neg Я_{2,2}} \tag{7.16}$$

Використання **правила резолюцій** також **вимагає**, щоб результуюче висловлювання містило лише по одній копії кожного літералу. Видалення зайвої копії літералів називається **факторизацією**. Тому, наступні два вирази після застосування правила резолюцій буде мати наступний вигляд.

$$\begin{array}{l} 1: A \vee B \\ 2: A \vee \neg B \\ \hline A \vee B \vee A \vee \neg B = A \vee A = A \end{array} \tag{7.17}$$

7.6 Кон’юнктивна нормальна форма

Правило резолюцій може застосовуватись лише до виразів у вигляді **диз’юнкції літералів.** Через це на перший погляд можна вирішити, що це правило застосовується тільки до баз знань і запитів, які складаються лише з диз’юнкцій. Але у логіці висловлювань будь-яке висловлювання логічно еквівалентне кон’юнкції виразів. Будь-яке висловлювання, представлене у вигляді кон’юнкції виразів із диз’юнкцією літералів, називається висловлюванням у **кон’юнктивній нормальній формі — КНФ.**

$$\begin{aligned} &\textit{ВисловлюванняКНФ} \rightarrow \textit{Вираз}_1 \wedge \ldots \wedge \textit{Вираз}_n \\ &\textit{Вираз} \rightarrow \textit{Літерал}_1 \vee \ldots \vee \textit{Літерал}_m \\ &\textit{Літерал} \rightarrow \textit{Символ} \mid \neg \textit{Символ} \\ &\textit{Символ} \rightarrow P \mid Q \mid \ldots \\ &\textit{Факт} \rightarrow \textit{Символ} \end{aligned} \tag{7.18}$$

Давайте розглянемо цю процедуру перетворення у КНФ на прикладі висловлювання із сформованої раніше бази знань — $B_{1,1} \Leftrightarrow (Я_{1,2} \vee Я_{2,1})$ для цього:

1. Усуваємо двосторонню імплікацію

$$(B_{1,1} \Rightarrow (Я_{1,2} \vee Я_{2,1})) \wedge ((Я_{1,2} \vee Я_{2,1}) \Rightarrow B_{1,1}) \tag{7.19}$$

2. Усуваємо імплікацію

$$(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}) \wedge (\neg (Я_{1,2} \vee Я_{2,1}) \vee B_{1,1}) \tag{7.20}$$

3. Кон’юнктивна нормальна форма вимагає, щоб “заперечення” було лише біля літералів — символів (не виразів), тому вводимо її всередину виразу з правої частини за **правилом де Моргана**:

$$(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}) \wedge ((\neg Я_{1,2} \wedge \neg Я_{2,1}) \vee B_{1,1}) \tag{7.21}$$

4. Тепер висловлювання містить лише вкладені зв’язки “І” та “АБО”, що застосовуються до літералів. Використаємо закон **дистрибутивності**, щоб розподілити зв’язки “АБО” по зв’язкам “І”.

$$(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}) \wedge (\neg Я_{1,2} \vee B_{1,1}) \wedge (\neg Я_{2,1} \vee B_{1,1}) \tag{7.21}$$

В результаті цих перетворень висловлювання було приведено до **кон’юнктивної нормальної форми**. Його стало складніше читати, але тепер його можна застосувати в якості вхідних даних для **алгоритму резолюцій**.

7.7 Алгоритм резолюцій

Процедура формування **логічного висновку,** із використанням правила резолюцій, базується на використанні принципу доказу шляхом встановлення протиріч. Таким чином, щоб з'ясувати, що із бази знань агента слідує певне висловлювання *A*, треба показати, що висловлювання $БЗ \wedge \neg A$ є **нездійсненними**. Тобто, **не має жодної** моделі де це висловлювання є істиною. Це можна довести показавши, що має місце протиріччя.

Спочатку висловлювання перетворюється у **кон'юнктивну нормальну форму,** а потім до отриманого виразу застосовується **правило резолюцій**. В кожній парі виразів, які містять взаємно протилежні літерали, виконується видалення цих двох літералів. Отримується новий вираз, який додається до існуючої множини виразів, в разі якщо такого виразу ще не було додано. Вказаний процес повторюється до тих пір, поки не досягнеться одна з двох умов:

1. більше не створюється будь-який новий вираз, який можна додати до існуючих. В такому випадку із бази знань **не слідує** висловлювання *A;*

2. резолюція двох виразів приводить до отримання **пустого** виразу. В цьому випадку, із бази знань **слідує** висловлювання *A.*

Пустий вираз — диз'юнкція без диз'юнктів еквівалентне висловлюванню "хибність" або "*False*". Крім того, пустий вираз утворюється лише після видалення двох взаємно протилежних одиничних літералів, наприклад, $P \vee \neg P$.

Процедуру резолюцій можна використати, щоб сформувати логічний висновок для прикладу лабіринту. Припустимо, що агент знаходиться в квадраті (1,1), він не сприймає "вітер" з чого слідує, що в сусідніх квадратах не має ями. Відповідна база знань буде мати наступний вигляд:

$$БЗ = P_2 \wedge P_4 = (B_{1,1} \Leftrightarrow (Я_{1,2} \vee Я_{2,1})) \wedge \neg B_{1,1} \qquad (7.22)$$

Потрібно довести наступне висловлювання, що в квадраті (1,2) не має ями - $\neg Я_{1,2}$. Отже треба довести наступне виловлювання:

$$БЗ \wedge \neg A \tag{7.23}$$

$$(B_{1,1} \Leftrightarrow (Я_{1,2} \vee Я_{2,1})) \wedge \neg B_{1,1} \wedge Я_{1,2} \tag{7.24}$$

Перетворимо це висловлювання у КНФ. Першу частину вже було перетворено на попередньому кроці:

$$(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}) \wedge (\neg Я_{1,2} \vee B_{1,1}) \wedge (\neg Я_{2,1} \vee B_{1,1}) \wedge \neg B_{1,1} \wedge Я_{1,2} \tag{7.25}$$

Далі видаляємо взаємно протилежні літерали для кожної пари виразів:

1. $\dfrac{(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}), (\neg Я_{2,1} \vee B_{1,1})}{(\neg B_{1,1} \vee Я_{1,2} \vee B_{1,1})}$ еквівалентно “True”
2. $\dfrac{(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}), (\neg Я_{2,1} \vee B_{1,1})}{(Я_{1,2} \vee Я_{2,1} \vee \neg Я_{2,1})}$ еквівалентно “True”
3. $\dfrac{(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}), (\neg Я_{1,2} \vee B_{1,1})}{(\neg B_{1,1} \vee Я_{2,1} \vee B_{1,1})}$ еквівалентно “True”
4. $\dfrac{(\neg B_{1,1} \vee Я_{1,2} \vee Я_{2,1}), (\neg Я_{1,2} \vee B_{1,1})}{(Я_{1,2} \vee Я_{2,1} \vee \neg Я_{1,2})}$ еквівалентно “True”
5. $\dfrac{(\neg Я_{2,1} \vee B_{1,1}), \neg B_{1,1}}{\neg Я_{2,1}}$
6. $\dfrac{(\neg Я_{1,2} \vee B_{1,1}), \neg B_{1,1}}{\neg Я_{1,2}}$
7. $\dfrac{\neg Я_{1,2}, Я_{1,2}}{-}$

В результаті отримуємо **пустий** вираз (7), що свідчить про успішне доведення істинності запиту. Тобто, із бази знань слідує висловлювання “в квадраті (1,2) не має ями”.

7.8 Гібридний агент

Здатність формувати **логічні висновки** про різні аспекти навколишнього середовища можна доволі легко поєднати із поведінкою агента, який використовує **принцип “правило-дія”**, а також **алгоритми вирішення задач**. Це дає можливість **скомбінувати** дані підходи для побудови **гібридного агента**. Програма агента буде підтримувати і оновлювати базу знань, а також поточний план дій. Початкова база знань буде містити правила функціонування навколишнього середовища, аналогічно до аксіом в прикладі із лабіринтом.

Наприклад, таким правилом буде сприйняття в поточному квадраті “вітру”, коли в суміжних квадратах присутня яма. На кожному черговому етапі, в ході життєвого циклу агента, в базу будуть додаватись чергові висловлювання стосовно поточного сприйняття агента. Після цього агент буде звертатись до бази знань, щоб отримати інформацію про наявні безпечні квадрати, які потрібно відвідати. Основана частина програми агента будуватиме план, враховуючи декілька можливих цілей за зниженням їх пріоритетів. Наприклад, першою метою з найвищим пріоритетом може бути взяти “скарб” в разі сприйняття “блиску” і далі прокласти безпечний маршрут із лабіринту. Планування маршруту виконуватиметься за допомогою певного алгоритму пошуку, наприклад, які розглядались минулими темами. Якщо досягти цієї мети не вийде, програма зможе перейти до наступної мети, наприклад, знайти безпечний квадрат, який ще не був відвіданим. В такому випадку агент може звернутись до своєї бази знань, щоб провести процедуру доведення і зробити логічний висновок про безпечність квадрату, а саме “довести, що в потрібному йому квадраті відсутня небезпека” - $БЗ \wedge \neg OK_{x,y} = false$. Якщо такого квадрату не виявиться, тоді агентом буде відмічено, що поставлену задачу неможливо виконати. В цьому разі, єдиним рішенням для агента буде повернутись до квадрату (1,1) та покинути лабіринт.

7.9 Логіка висловлювань та логіка першого порядку

Логіка висловлювань дозволяє проілюструвати основні поняття логіки та процедури формування логічного висновку. Але також є більш розширений спосіб представлення інформації про навколишнє середовище у більш виразному стислому вигляді.

Мови програмування є найбільшим класом формальних мов, які отримали широке застосування. В програмах окремі факти про середовище представляються у вигляді певних структур даних, масивів, списків, тощо. Наприклад, масив можна використати для представлення розглянутого

прикладу лабіринту. Але в цілому мовам програмування не вистачає загального механізму виводу нових фактів із вже існуючих. Будь-яке оновлення структури даних виконується за допомогою певних процедур, які є специфічними для конкретної проблемної області. Їх реалізація та деталі визначаються розробником, враховуючи його власні знання про цю саму проблемну область. Такий процедурний підхід може бути протиставлено декларативній природі логіки висловлювань, в якій знання та методи логічного висновку не змішуються і формування логічного висновку виконується незалежно від проблемної області. В якомусь сенсі процес логіки є універсальним. Для прикладу, поєднання декларативних та процедурних підходів застосовуються у база даних SQL.

***Л*огіка висловлювань** — це **декларативна мова**, оскільки її семантика базується на значеннях істинності відношень між певними фактами та можливими моделями навколишнього середовища. Також вона дозволяє оброблювати інформацію, яка є частково заданою, використовуючи для цього поєднання наявних фактів за допомогою логічних операторів. Ще однією її особливістю є **композиційність**, коли загальна семантика висловлювання представляється у вигляді певної функції від семантики окремих частин цього висловлювання. Однак в цілому логіці висловлювань не вистачає виразності, щоб коротко описувати певні аспекти середовища. Наприклад, для опису правила із лабіринту, необхідно було створювати окреме висловлювання для кожного квадрату.

$$B_{1,1} \Leftrightarrow (Я_{1,2} \vee Я_{2,1}) \tag{7.26}$$

Природною мовою, таке правило можна було достатньо легко описати одним реченням для всього лабіринту.

Напротивагу, мова **логіки першого порядку**, її синтаксис та семантика базується на поняттях **об’єкту** та **відношень** між ними. Це дозволяє виражати факти про навколишнє середовище в більш стислому вигляді. Основною її відмінністю від логіки висловлювань, де присутні лише факти зі значенням *True-False* про середовище, є більш широке представлення про середовище як

таке, що складається із об’єктів та відношень між ними, які можуть існувати або не існувати.

Основними синтаксичними елементами логіки першого порядку є символи, які позначають об’єкти, відношення та функції (рис. 7.1). Тому самі символи діляться на три типи: **символи констант** — позначають об’єкти. **Символи предикатів** — позначають відношення між об’єктами. **Символи функцій** — позначають певні функції над об’єктами.

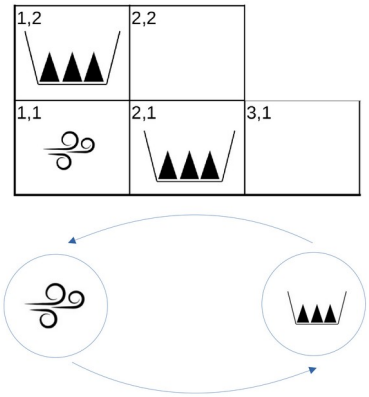


Рисунок 7.1 — Представлення фактів/правил у логіці висловлювань (зверху) та логіці першого порядку (знизу)

7.10 Логічне програмування

Логічне програмування — це технологія, яка дозволяю застосовувати **декларативний підхід** при описі знань та виведенні нових знань. Розглянуті до цього підходи із процесом формування **логічного висновку** у логіці висловлювань можна реалізувати, застосувавши **логічне програмування**. В цілому подібні логічні системи конструюються шляхом представлення знань про навколишнє середовище на певній формальній мові. Далі поставлена задача вирішується за допомогою застосування процесів логічного висновку до цих знань. Такий принцип було виражено в рівнянні Роберта Ковальскі:

$$\text{Алгоритм} = \text{Логіка} + \text{Управління} \tag{7.27}$$

Розповсюдженою мовою логічного програмування є мова *Prolog*. Наразі ***SWI-Prolog*** є відкритою реалізацією даної мови логічного програмування. Вона застосовується в більшій мірі в якості мови для швидкої розробки **прототипів,** роботи із **семантичними мережами**, а також для вирішення задач, пов'язаних із **маніпуляцією над символами**, наприклад, при написанні **компіляторів** і **синтаксичному аналізі** текстів на природних мовах. На мові *Prolog* було написано багато **експертних систем** для юридичних, медичних, фінансових та інших проблемних областей.

Програми на мові *Prolog* представляють собою множину певних виразів записаних у системі позначень, яка в незначній мірі відрізняється від позначень у стандартній **логіці першого порядку**. В мові *Prolog* малі літери застосовуються для позначення змінних, а великі — для позначення констант. Коми використовуються, щоб розділяти кон'юнкти у виразах. Самі вирази в даній мові записують в зворотному напрямі, порівняно із записом, який розглядався для логіки висловлювань. Вираз в такій формі, як $A \wedge B \Rightarrow C$, на мові *Prolog* буде мати наступний запис: $C :- A, B$.

В *Prolog* запис $[E|L]$ означає список, першим елементом якого є — *E*, а інша частина цього списку позначена як *L*. Наприклад, така програма для предикату *append(X, Y, Z)* буде успішно виконуватись — повертати значення *True*, якщо список *Z* буде являтись результатом додавання списку *X* в кінець списку *Y*.

```
append([], Y, Y).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

Природною мовою ці два вирази можна прочитати наступним чином.

1. Доповнення списку *Y* пустим списком призводить до отримання того ж самого списку *Y*.

2. *[A|Z]* — це результат доповнення списку *Y* елементами списку*[A|X]* , при умові, що *Z* — це результат доповнення списку *Y* елементами списку *X*.

В більшості мов програмування високого рівня можна написати схожу функцію, яка визначатиме як додати один список у кінець іншого. Однак

подібне визначення на мові *Prolog* є більш потужним, оскільки цей запис **описує відношення** між трьома аргументами, а не функцію, яка обчислюється з двох аргументів.

Щоб проілюструвати це відношення можна вести в дану програму запит у вигляді *append(X, Y, [1,2,3])*. Іншими словами це буде звучати як: “які два списки можна доповнити один одним, щоб отримати результуючий список *[1,2,3]*”. Програма на *Prolog* в якості відповіді поверне наступні рішення:

X = [] *Y = [1,2,3];*

X = [1] *Y = [2,3];*

X = [1,2] *Y = [3];*

X = [1,2,3] *Y = [].*

7.10.1 Принцип виконання програми на мові Prolog

Виконання програм *Prolog* відбувається за принципом **зворотного логічного висновку** із **пошуком у глибину**, при якому спроба застосувати вираз виконується в тому порядку, в якому вони були записані в **базу знань**. Конструктивні рішення, закладені в основу *Prolog*, представляють собою компроміс між декларативністю та ефективністю роботи. В результаті деякі аспекти мови *Prolog* виходять за рамки стандартного логічного висновку.

1. В мові *Prolog* використовується семантика баз даних, а не семантика логіки першого порядку. Це можна буде чітко побачити із трактування рівності та заперечення.

2. В *Prolog* передбачена велика кількість вбудованих функцій для виконання арифметичних операцій. Для літералів, в яких використовуються відповідні функціональні символи, виконується **доведення** шляхом виконання коду замість виконання подальшої процедури формування логічного висновку. Наприклад, запит “*X* is 4+3” успішно виконується після встановлення зв’язку змінної *X* із значенням 7. Напротивагу, спроба виконати запит “5 is X+Y”

завершиться безуспішно тому, що вбудовані арифметичні функції не забезпечують вирішення довільних виразів.

3. В мові *Prolog* присутні вбудовані предикати, запит яких дозволяє виконувати певні функції, наприклад, предикати **вводу-виводу** та предикати assert/retract для **модифікації** бази знань.

4. В системі *Prolog* використовується зворотний процес формування логічного висновку із пошуком у глибину без перевірок на нескінченні рекурсії. Так було зроблено для зручності використання мови програмування, яка працює доволі швидко при правильному використанні. Однак це також означає, що логічна програма, яка побудована на першого погляду цілком допустимо, може в результаті ніколи не завершити своє виконання.

7.10.2 Надлишковість логічного висновку та нескінченні цикли

Неузгодженість між пошуком в глибину та деревами пошуку, які включають повторювані стани та нескінченні шляхи, є певним недоліком при роботі із *Prolog*. Продемонструвати дану особливість роботи мови можна розглянувши логічну програму, що дозволяє визначити чи існує шлях між двома вершинами в орієнтованому графі (рис. 7.2). В цілому така програма матиме два **правила** — висловлювання-предикат:

path(X, Z) :- link(X, Z)

path(X, Z) :- path(X, Y), link(Y, Z)

Щоб описати простий граф, який складається із трьох вершин, потрібно буде додати в базу знань два **факти** — висловлювання:

link(a, b)

link(b, c)

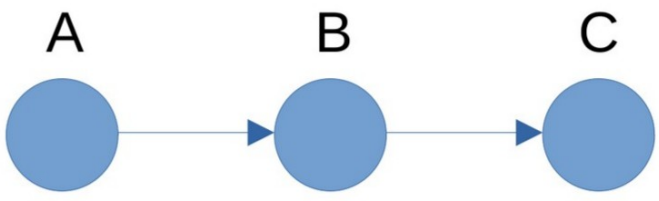


Рисунок 7.2 — Простий граф із трьох вершин

При виконані в цій програмі запиту “*path(a, c)*” сформується дерево доказу показане на рис. 7.3. З іншого боку, якщо поміняти місцями зазначені правила, це призведе до того, що система *Prolog* буде слідувати по нескінченному циклу (рис. 7.4). Через це для виконання правильних обчислень, необхідно вказувати правильну послідовність висловлювань.

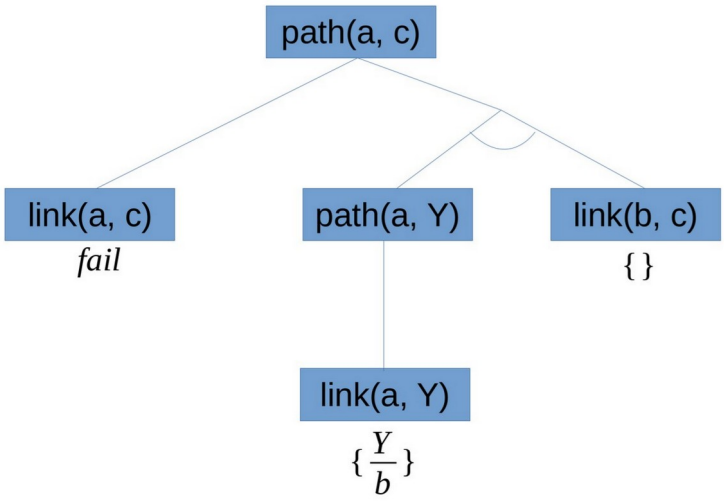


Рисунок 7.3 — Дерево доказу для запиту при правильній послідовності фактів

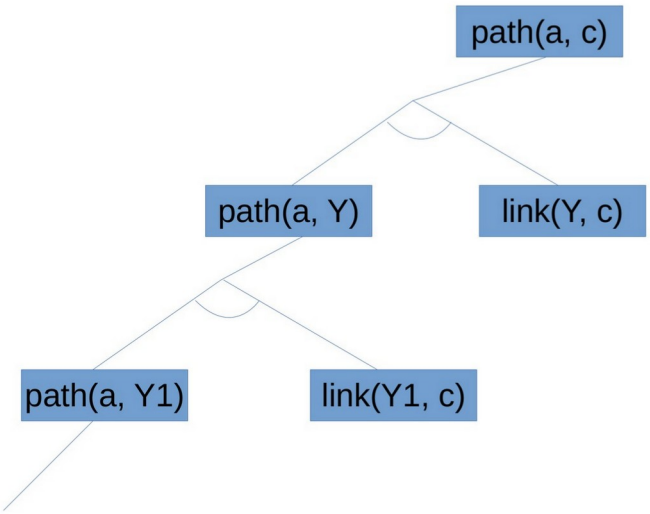


Рисунок 7.4 — Нескінченне дерево доказу для запиту при неправильно заданій послідовності фактів

```
%Список товарів та їх категорія
item('Player','Electronic').
item('Laptop','Electronic').
item('TV','Electronic').
item('HDMI Cable','Accessory').
item('USB Hub','Accessory').
item('Headphones','Accessory').
item('Bluetooth mouse','Accessory').
%Гарантія на категорію товару
warranty('6 month','Accessory').
warranty('2 year','Electronic').
%Виробник товару
manufacturer('Noname','HDMI Cable').
manufacturer('Noname','USB Hub').
manufacturer('Noname','Headphones').
manufacturer('Someone','Bluetooth mouse').
manufacturer('LG','Player').
manufacturer('Samsung','TV').
manufacturer('Dell','Laptop').
%Товар в магазині
shopitem('Player','MarketPlace').
shopitem('HDMI Cable','MarketPlace').
shopitem('Laptop','MarketPlace').
shopitem('Bluetooth mouse','MarketPlace').
shopitem('TV','eShop').
shopitem('HDMI Cable','eShop').
shopitem('Laptop','eShop').
shopitem('Player','eShop').
```

```
%Правила
manufactInShop(Shop,Manufacturer):-
shopitem(ItemName,Shop),manufacturer(Manufacturer,ItemName).

shopAccessoryList(Shop,ItemName):-shopitem(ItemName,Shop),
item(ItemName,Category),Category='Accessory',Name=ItemName.


whatWarranty(ItemName,Warranty):-item(ItemName,Category),
warranty(Warranty,Category).
```

Рисунок 7.5 — Приклад запису фактів, що імітують базу товарів, у програмі на мові *Prolog*

```
?- shopAccessoryList('eShop',ItemName).
ItemName = 'HDMI Cable'
yes
?- whatWarranty('Laptop',Warranty).
Warranty = '2 year'
yes
?- manufactInShop('MarketPlace',HasManufacturer).
HasManufacturer = 'LG' ;
HasManufacturer = 'Noname' ;
HasManufacturer = 'Dell' ;
HasManufacturer = 'Someone' ;
no
```

Рисунок 7.6 — Результати виконання запитів, що імітують базу товарів, для прикладу програми на мові *Prolog*

ТЕМА 8
ОБРАННЯ АГЕНТОМ ДІЙ В УМОВАХ НЕВИЗНАЧЕНОСТІ

Агенти, які діють в реальному середовищі, повинні справлятись із **невизначеністю**. Невизначеність може виникати через властивість реального середовища, аспекти якого у більшості випадках можна спостерігати лише частково. Також невизначеність виникаю у зв’язку із **недетермінованістю** дій самого агента або в разі якщо агент діє у **багатоагентному** середовищі. До цього розглядались агенти, що вирішували задачі та агентів, які використовували логіку. Вони могли в певній мірі справлятись із невизначеністю шляхом відстеження **можливих станів** та формування умовних планів дій, за допомогою певних алгоритмів або враховуючи наявні знання чи інформацію сприйняття із датчиків.

Для простих задач це може бути корисним, але в такому випадку:

1. Агент повинен враховувати **велику кількість ситуацій**, які можуть виникати в ході його роботи.

2. **План дій**, який враховуватиме будь-які **можливі ситуації**, може стати нескінченно великим.

3. Іноді може **не існувати** плану дій, який гарантовано дозволить досягти **поставленої мети**. Але в будь-якому випадку агент повинен діяти.

8.1 Приклад невизначеності

Давайте, розглянемо звичайну ситуацію, яка скоріше за все всім відома — “прибути на важливу зустріч вчасно”. Найпростішим рішенням цієї задачі буде, наприклад, “вийти із дому раніше, припустимо за 60 хв.”. Агент, який вирішуватиме дану задачу в цілому не можу бути абсолютно впевненим, що “вийшовши завчасно за 60 хв.” дозволить йому “прибути на зустріч вчасно”, навіть якщо йому потрібно проїхати декілька кілометрів. Замість цього агент в ході оцінки плану може прийти до висновку, що план “вийти за 60 хв.”

дозволить йому “прибути на зустріч вчасно” **в разі, якщо** “на дорозі не буде заторів, якщо автомобіль не зламається, записав правильну адресу, не забув перевести годинник”, і багато інших можливих ситуацій. По жодній з цих умов **не можна** зробити **гарантовано** вірне судження, з чого слідує, що не можна гарантувати успішність самого плану. Тим не менше, можна припустити що із усіх можливих планів, дозволить максимізувати **показники продуктивності**. Це може бути найочевидніше, “прибути вчасно”, “мінімізувати час очікування, якщо прибути раніше потрібного часу”, і т.д. Наявні знання агента можуть не дозволити йому гарантувати досягнення будь-якої із цих зазначених умов при виконанні обраного план. Але з певної долею впевненості можна стверджувати, що обрана мета буде досягнута. Інші плани, як для прикладу “вийти за 2 год.” можуть збільшити впевненість “прибути вчасно”, але також можуть привести до більшого очікування. Звідси слідую, що обрання правильного способу дій — **раціональне рішення** залежить не тільки від відносної **важливості** поставленої мети, а й також від **впевненості**, що поставленої мети буде досягнуто.

8.2 Судження в умовах невизначеності

Розглянемо простий приклад судження про погоду **в умовах невизначеності**. Спробуємо записати правила для визначення дощу, використовуючи **логіку висловлювань**. В подальшому це дозволить проілюструвати **труднощі**, які виникають при звичайному логічному підході. Запишемо просте правило, що при холодній погоді буде дощ.

$$\text{Холодна погода} \Rightarrow \text{Дощ} \tag{8.1}$$

Проблема полягає в тому, що подібне правило є невірним, оскільки холодна погода не завжди буде через те, що йде дощ. Можливо зараз просто холодна пора року. При холодній погоді, також може бути хмарно, вітряно, і т.д.

$$\text{Холодна погода} \Rightarrow \text{Дощ} \vee \text{Хмарно} \vee \text{Вітряно} \vee \text{Холодна пора року} \tag{8.2}$$

Щоб з абсолютною впевненістю стверджувати про істинність цього правила, потрібно буде ввести в нього велику кількість ще інших можливих умов. Також можна спробувати переписати це правило.

$$\text{Дощ} \Rightarrow \text{Холодна погода} \tag{8.3}$$

Проблема полягає в тому, що подібне правило також є невірним, оскільки не завжди йде дощ, коли на вулиці холодна погода. Влітку може бути і “тепла погода”, але все одно може йти дощ.

Звідси слідує, що спроба використати логіку для вирішення задачі в даній проблемній області матиме декілька значних недоліків:

1. **Велика кількість зусиль**. Для формування повної множини передумов та наслідків для написання подібного правила знадобиться велика кількість зусиль, а використання таких правил буде доволі складним.

2. **Неповнота теоретичних знань.** Для проблемної області може бракувати теоретичних знань, для врахування всіх аспектів.

3. **Неповнота практичних знань.** Навіть якщо відомі всі теоретичні правила, все одно може мати місце невизначеність стосовно певних станів певної проблемної області.

Описана проблема типова не тільки для даного прикладу. В цьому випадку знання агента в кращій мірі дозволяють сформулювати релевантні судження лише з певним **ступенем впевненості.** Основним інструментом для роботи із цим є **теорія ймовірності**.

Можна порівнювати різні види агентів таких, як логічного агента та ймовірнісного агента. **Логічний агент** розглядає окремі аспекти середовища як певні висловлювання, які мають бути або істинними, або хибними. Напротивагу, **ймовірнісний агент** може мати чисельну оцінку ступеня впевненості від 0 — для висловлювань, які точно хибні, до 1 — для висловлювань, які точно істині.

Теорія ймовірності надає спосіб для **розрахунку невизначеності** в разі, наприклад, неповноти знань агента. Можна не знати із повною впевненістю за яких умов буде “холодна погода”, але можна стверджувати, що в 70% випадків

— ймовірність 0.7, що “холодна погода” буде через те що йде “дощ”. Подібна впевненість може базуватись на статистичних даних, певних знаннях, тощо. Єдиним неоднозначним моментом наразі є те, що в **реальному середовищі** не може бути місця невизначеності. Погода або холодна, або тепла. Дощ йде або ні. В цілому твердження із ймовірністю робляться по відношенню до **знань агента**, а не аспектів реального середовища.

8.3 Невизначеність та раціональні рішення

Розглядаючи приклад із “прибути на важливу зустріч вчасно” і “виїздом завчасно за 60 хв.”, можна припустити, що агент матиме шанс у 95% “прибути вчасно”. Чи означає це, що подібний план є раціональним? Необов’язково. Можуть існувати інші варіанти дій, наприклад, “вийти завчасно за 2 год.”, які матимуть більшу ймовірність “прибути вчасно”. Якщо важливішим є саме не запізнитись на зустріч, тоді другий план може бути кращім, незважаючи на можливість прибути зарано і чекати.

Щоб **зробити вибір**, агент перед усім повинен керуватись інформацією про **переваги** при різних результатах. Будь-який результат — це визначений стан, що включає потрібні аспекти середовища, наприклад, “не запізнитись” та “менше чекати”. **Теорія корисності** використовується для представлення переваг та кількісних суджень про них. Ця теорія стверджує, що для інтелектуального агента кожен стан або послідовність станів має певну величину корисності і агент завжди має віддавати **перевагу** станам, які мають **більшу корисність**. Переваги, виражені у вигляді корисності, комбінуються із ймовірністю в загальній теорії раціонального рішення, яка називається **теорією прийняття рішень.**

Теорія прийняття рішень = теорія ймовірності + теорія корисності

Фундаментальна ідея теорії прийняття рішень полягає у тому, що будь-який агент є раціональним тоді і тільки тоді, коли він обирає дії, які дозволяють

досягти **найбільшої очікуваної корисності**, усередненій по всім можливим результатам обраної дії.

8.4 Ймовірнісні твердження

Як і логічні твердження, ймовірнісні твердження стосуються можливих моделей — станів середовища в якому працює агент. Ймовірність показує величину “на скільки можливою” є певна модель. В теорії ймовірності множина всіх можливих моделей називається **простором елементарних подій**. Повністю визначена ймовірнісна модель характеризується чисельною величиною ймовірності. Для кожної моделі ймовірність лежить в межах від 0 до 1. Загальна сума ймовірностей для всіх моделей рівна 1.

$$0 \leqslant P(\omega) \leqslant 1 \tag{8.4}$$

$$\sum P(\omega) = 1 \tag{8.5}$$

Ймовірності, такі як *P(x)* називаються безумовними або **апріорними ймовірностями**. Така ймовірність стосується ступеня довіри до певного твердження при умові **відсутності** будь-якої іншої інформації. Але частіше ми володіємо іншою інформацією, яка пов’язані із твердженням, що нас цікавить. В таких випадках нас буде цікавити не апріорна ймовірність певної події, а умовна або **апостеріорна ймовірність** певної події, при умові **наявності іншої**. Апостеріорна ймовірність записується як *P(x | y)*. Для прикладу, апріорна ймовірність, що буде дощ може складати *P(дощ) = 0,2*. Але при рішенні чи брати парасольку, чи ні буде важливіша апостеріорна ймовірність *P(дощ | хмарна погода) = 0,6.* Важливо, що апріорна ймовірність залишається актуальною і після того, як ми побачимо, що на дворі хмарна погода. Але, враховуючи цю інформацію, вона вже не є настільки корисною. Приймаючи рішення, агент повинен враховувати всі аспекти, які він може спостерігати. Твердження *P(дощ | хмарна погода)* звісно не означає, що “всякий раз, коли надворі хмарна погода, можна прийти до висновку, що буде дощ із ймовірністю 0,6”. В дійсності мається на увазі трохи інше, а саме, що “всякий раз коли на

дворі хмарна погода і ми **не маємо** ніякої **додаткової інформації**, можна зробити висновок, що буде дощ із ймовірність 0,6”.

Формула апостеріорної ймовірності має наступний вигляд

$$P(a|b)=\frac{P(a\wedge b)}{P(b)} \tag{8.6}$$

В іншому записі ця формула називається **правилом множення ймовірностей:**

$$P(a\wedge b)=P(a|b)P(b) \tag{8.7}$$

Вона базується на тому факті, що для того, щоб твердження $a\wedge b$ було істинним, потрібно, щоб *b* було істинним, а також *a* при умові *b*.

Властивість незалежності

В разі якщо, події *a* та *b* не пов’язані, тоді:

$$P(a|b)=P(a) \text{ або } P(b|a)=P(b) \text{ та } P(a\wedge b)=P(a)P(b) \tag{8.8}$$

Твердження про **незалежність** певних подій між собою зазвичай базуються на **знаннях** про конкретну проблемну область.

8.5 Правило Байеса

Вище було представлено правило множення ймовірностей. Фактично його можна записати в двох формах:

$$P(a\wedge b)=P(a|b)P(b) \tag{8.9}$$

$$P(a\wedge b)=P(b|a)P(a) \tag{8.10}$$

прирівнявши праві частини цих двох рівнянь та розділивши їх на $P(a)$ отримаємо:

$$P(b|a)=\frac{P(a|b)P(b)}{P(a)} \tag{8.11}$$

Дане рівняння називається **правилом** або **теоремою Байеса.** Це рівняння лежить в основі сучасних систем **штучного інтелекту** для формування **ймовірнісних висновків**. У більш загальному вигляді для багатозначних змінних, правило Байеса виглядає наступним чином:

$$P(Y|X)=\frac{P(X|Y)P(Y)}{P(X)} \tag{8.12}$$

8.5.1 Застосування правила Байеса

Дане правило дозволяє розраховувати ймовірність події *b* при умові події *a*, маючи при цьому три інші ймовірності $P(a|b), P(b), P(a)$. Зазвичай дані сприйняття вказують на ***наслідок***, який було викликано невідомою ***причиною***, і за допомогою розрахунків необхідно **визначити** цю причину. В такому випадку правило Байеса приймає наступний вигляд:

$$P(\textit{причина}|\textit{наслідок})=\frac{P(\textit{наслідок}|\textit{причина})P(\textit{причина})}{P(\textit{наслідок})} \tag{8.13}$$

Умовна ймовірность $P(\textit{наслідок}|\textit{причина})$ представляє кількісну оцінку взаємозв'язку двох подій у **причинному напрямі,** а умовна ймовірність $P(\textit{причина}|\textit{наслідок})$ описує взаємозв'язок у **діагностичному напрямі —** зворотному.

Наприклад, розглянемо ситуацію з автомобілем, який не завівся у зв'язку із можливою відсутністю палива. Яка буде ймовірність відсутності палива при цьому? Вважаємо, що подія А — автомобіль не заводиться (симптом, наслідок), Б — не має палива (причина). Логічно, що автомобіль точно не заведеться в разі відсутності в ньому палива. Отже, матимемо апостеріорну ймовірність $P(A|Б)=1$. Також припустимо, що апріорні ймовірності, що машина не заведеться в цілому $P(A)=0{,}02$ (2%), а ймовірність, що в ній не буде палива, дорівнюватиме $P(Б)=0{,}01$ (1%). Тепер ми можемо розрахувати, з якою ймовірністю "в автомобілі не буде палива в разі, якщо він не завівся".

$$\begin{aligned} P(Б|A)&=\frac{P(A|Б)P(Б)}{P(A)}= \\ &=\frac{1*0{,}01}{0{,}02}=0{,}5 \end{aligned} \tag{8.14}$$

Правило Байеса може використовуватись для отримання відповіді на ймовірнісні запити, в яких враховується один із наслідків, наприклад, "автомобіль не заводиться". Часто інформація доступна саме в формі

$P(\text{наслідок}|\text{причина})$. Але в разі, якщо наслідків буде більше, потрібно буде враховувати це при розрахунках. Для цього можна застосовувати поняття **незалежності** при розгляді різних наслідків для певного твердження. Наприклад, можна вважати, що наслідок “індикатор низького рівня палива” прямо не пов’язаний із наслідком що “автомобіль не заводиться”, оскільки цей другий наслідок може бути по причині того, що існують певні “проблеми із електрикою автомобіля”. Отже перший наслідок “автомобіль не заводиться” є незалежним від другого наслідку “індикатор низького рівня палива”.

8.5.2 Наївні байесовські моделі

Приведений приклад ілюструє доволі часту ситуацію, коли одна причина безпосередньо впливає на ряд наслідків, при чому, всі ці наслідки є **умовно незалежними**, коли має місце ця причина. Повний **взаємний розподіл ймовірностей** для подібної ситуації може бути записаний у вигляді даної формули:

$$P(\text{причина}, \text{наслідок}_1, \ldots, \text{наслідок}_n) = P(\text{причина}) \prod_i P(\text{наслідок}_i|\text{причина}) \quad (8.15)$$

Такий розподіл ймовірностей називається наївною байесовською моделлю. Термін “наївний” тут застосовується тому, що робиться спрощувальне **припущення** про незалежність “наслідків”, хоча строго вони можуть і не бути повністю не пов’язаними між собою при заданій “причині”. Наївну байесовську модель на практиці іноді називають **байесовським класифікатором**.

8.5.3 Класифікація тексту за допомогою наївної байесовської моделі

Розглянемо, як наївну байесовську модель можна застосовувати для **класифікації текстів**. Припустимо, що дано текст і необхідно визначити до якого завчасно заданого класу або категорії він відноситься. Тут в якості **“причини”** виступає **категорія** тексту, а наявність в тексті **певних ключових слів** представляється у вигляді **“наслідку”**. Розглянемо такий текст: “У вівторок

в Києві буде сонячна погода. Температура повітря до 10 градусів Цельсія”. Задача полягає у тому, щоб віднести кожне речення до конкретної категорії: новини, спорт, бізнес, погода або розваги.

Наївна байесовська модель включатиме:

1. Апріорні ймовірності $P(категорія)$

2. Апостеріорні ймовірності $P(слово_i|категорія)$

Ймовірність для кожної категорії виражається як частка документів, які відносяться до цієї категорії, від загального числа документів, які до цього розглядались. Якщо в 15% розглянутих документів мова йшла про “погоду”, тоді $P(категорія=погода)=0{,}3$. Аналогічним чином ймовірність $P(слово_i|категорія)$ оцінюється як частка документів, в яких присутнє *i*-слово. Для прикладу якщо, 37% документів про “погоду” містять слово “градусів”, то $P(слово=градусів|категорія=погода)=0{,}37$. Зазначені ймовірності розраховуються в ході **тренування класифікатору** на вже розмічених документах. Щоб класифікувати новий документ, потрібно перевірити, які ключові слова в ньому присутні, а потім застосувати рівняння для наївною байесовської моделі, щоб отримати апостеріорні ймовірності $P(категорія|ключове\,слово)$. Якщо необхідно вказати лише одну категорію, тоді обирається та, яка буде мати **найбільшу** апостеріорну ймовірність.

ТЕМА 9
ЙМОВІРНІСНІ СУДЖЕННЯ

У попередній темі було розглянуто основні елементи теорії ймовірності і зазначено важливі властивість **незалежності** та **умовної незалежності** для спрощення ймовірнісного представлення навколишнього середовища. В цій темі буде розглянуто систематичний підхід представлення подібних залежностей у формі **байесовських мереж**. Для цього буде визначено **синтаксис** і **семантика** цих мереж і показано, як вони можуть застосовуватись для представлення знань у більш ефективний спосіб.

9.1 Представлення знань у вигляді байесовської мережі

Повний **спільний розподіл ймовірностей** дозволяє отримувати відповіді на будь-які запити в проблемній області, що розглядається. Але по мірі збільшення кількості змінних, цей розподіл може значно збільшувати свій обсяг, що в подальшому ускладнює або навіть унеможливлює подальші обчислення. Використання властивості **незалежності** та умовної незалежності між змінними, дозволяє значно скоротити кількість окремих ймовірностей, які мають бути розглянуті для визначення повного спільного розподілу.

Структура даних, яка дає можливість представляти подібні залежності між змінними, називається **байесовською мережею.** Байесовські мережі дозволяють представляти практично будь-який повний спільний розподіл ймовірностей.

Байесовська мережа — це **орієнтовний граф**, в якому кожна вершина має інформацію про **значення ймовірності**. Мережа має наступні властивості:

1. Кожній **вершині** графа відповідає ймовірнісна змінна, яка може мати дискретне (*True*, *False*) або неперервне значення;

2. Направлені **ребра** з’єднують пари вершин. Стрілка показує **напрям** цього зв’язку. Наприклад, якщо стрілка направлена від вершини *X* до вершини

Y, тоді вершина *X* називається **батьківською** вершиною для вершини *Y*. Граф не маю направлених **циклів.** З цього слідую, що він є **орієнтованим ациклічним графом** (*DAG — Directed Acyclic Graph*);

3. Кожна вершина X_i характеризується пов'язаною із нею інформацією про **значення ймовірності** $\theta(X_i|Батьківські(X_i))$, яка кількісно оцінює вплив батьківських вершин на дану вершину, використовуючи кінцеву кількість параметрів.

Топологія мережі (*множина вершин та їх ребер*) визначає умовну незалежність зв'язків, що присутні в конкретній проблемній області. Ребра між вершинами та стрілки показують **напрям впливу** подій одна на іншу. До прикладу, ребро у напряму від вершини *X* до вершини *Y* означатиме, що *X* прямо впливає на *Y*. З цього слідує, що будь-які вершини **"причини"** завжди мають бути батьківськими по відношення до вершин **"наслідків"**. Подібні відношення між вершинами встановлюються експертом в даній конкретній проблемній області, які як правило набагато простіше визначити, а ніж значення самих ймовірностей.

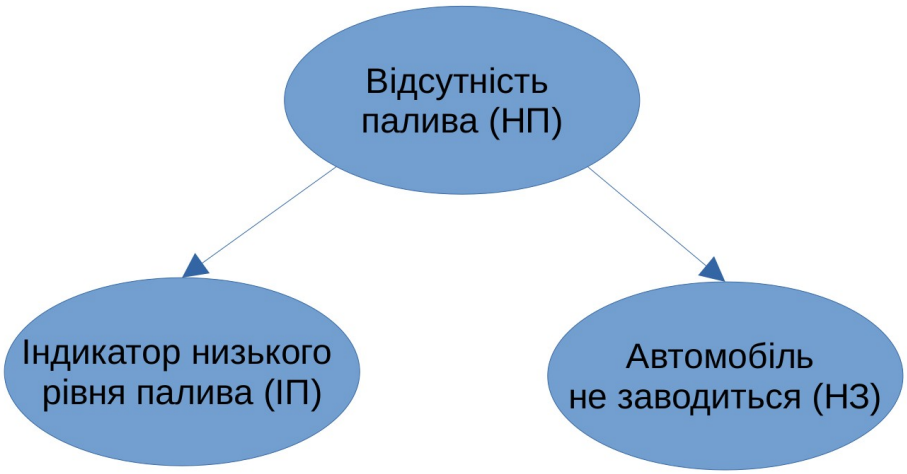


Рисунок 9.1 — Приклад простої байесовської мережі

Коли топологія байесовської мережі буде визначена, наступним кроком залишиться задати **локальну** інформацію про **значення ймовірностей** для кожної змінної у формі **розподілу умовних ймовірностей**, враховуючи

батьківські змінні. **Повний спільний розподіл** ймовірностей для всіх змінних визначається **топологією** самої мережі та **локальною ймовірнісною інформацією**.

Таблиця 9.1 — Локальна ймовірнісна інформація для певної змінної мережі

Автомобіль не заводиться (НЗ)			
ПЕ	НП	Т	F
F	F	0,02	0,98
F	T	0,999	0,001
T	F	0,95	0,05
T	T	0,999	0,001

Для ілюстрації зазначених визначень розглянемо просту байесовську мережу, що матиме структуру із прикладів про діагностику автомобіля та погоду з минулої теми.

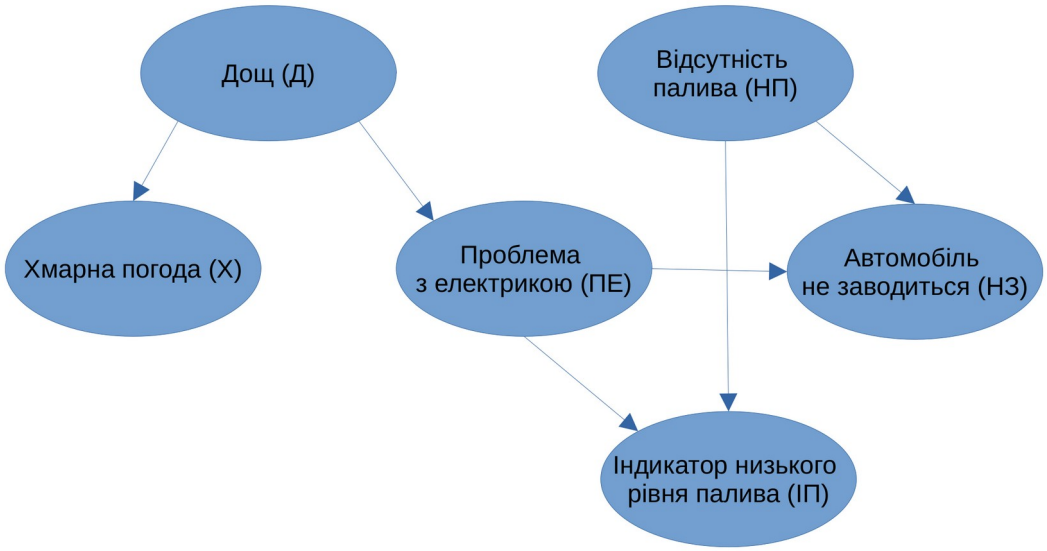


Рисунок 9.2 — Байесовська мережа для прикладів про діагностику автомобіля та погоду

Цей приклад проблемної області складатиметься із **6 змінних**, що показані у вигляді вершин. Ребрами виражається **причинно-наслідковий** зв’язок між зазначеними змінними. В даній мережі змінні “Хмарна погода” та “Проблема з електрикою” є **умовно незалежними**, якщо задана змінна “Дощ”. Формально умовна незалежність цих двох змінних позначається **відсутністю** зв’язку між цими двома вершинами “Хмарна погода” та “Проблема з електрикою”. Також із цієї мережі можна побачити, що змінні “Відсутність палива” є безпосередньою **причиною** змінних “Автомобіль не заводиться” та “Індикатор низького рівня палива”, які є **наслідком** для даної батьківської вершини. Аналогічно, змінна “Дощ” є причиною “Проблеми з електрикою”, що в подальшому створює непрямий причинно-наслідковий вплив на змінні “Індикатор низького рівня палива” та “Автомобіль не заводиться”. В той же час, змінні “Хмарна погода”, “Індикатор низького рівня палива” та “Автомобіль не заводиться” не мають прямого причинного зв’язку.

9.2 Синтаксис та семантика байесовських мереж

Синтаксис байесовської мережі складається із орієнтованого ациклічного графу із деякою **локальною ймовірнісною інформацією**.

Семантика байесовської мережі визначає, як її синтаксис відповідає **спільному розподілу ймовірностей** по змінним мережі.

Припустимо, байесовську мережу, що включає n змінних, X_1, ..., X_n. Типовий запис спільного розподілу ймовірностей для цих змінних буде мати наступний вигляд — $P(x_1,\dots,x_n)$. Семантика байесовської мережі визначає кожний запис в спільному розподілі за наступною формулою:

$$P(x_1,\dots,x_n)=\prod_{i=1}^{n}\theta(x_i|\textit{батьківські}(X_i)) \quad , \qquad (9.1)$$

де “ $\textit{батьківські}(X_i)$ ” позначає конкретні значення батьківських змінних по відношенню до поточної змінної, які з’являються в x_1, ..., x_n. З цього слідує,

що кожний елемент в **спільному розподілі** представлено у вигляді добутку відповідних елементів **локальних умовних розподілів** в байесовській мережі.

Подібний **повний спільний розподіл** можна застосовувати для отримання відповідей на будь-які запити у певній проблемній області. Якщо байесовська мережа є представленням спільного розподілу, тоді вона може бути використана для отримання відповіді на запити, шляхом **сумування** всіх відповідних значень ймовірностей **спільного розподілу**, кожне з яких розраховується шляхом **множення** ймовірностей із **локальних умовних розподілів**.

Окремо слід відмітити, що елементи **локального умовного розподілу** $\theta(x_i|батьківські(X_i))$ по факту є **умовними ймовірностями** $P(x_i|батьківські(X_i))$ **,** які слідують зі спільного розподілу. Умовні ймовірності можуть бути розраховані, використовуючи наступну формулу:

$$P(x_i|батьківські(X_i))=\frac{P(x_i,батьківські(X_i))}{P(батьківські(X_i))}=\frac{\sum_y P(x_i,батьківські(X_i),y)}{\sum_{x_i',y} P(x_i',батьківські(X_i),y)} \quad , \quad (9.2)$$

де y представляє значення всіх змінних, які відмінні від X_i і його батьківських вершин. Виходячи із тотожності **локального умовного розподілу** та **умовних ймовірностей**, формула **спільного розподілу** видозміниться у наступний вигляд:

$$P(x_1,\ldots,x_n)=\prod_{i=1}^{n} P(x_i|батьківські(X_i)) \quad (9.3)$$

Це означає, що коли оцінюються значення для **локальних умовних розподілів**, вони мають представляти собою фактичні **умовні ймовірності** для змінних при заданих батьківських змінних. Важливим фактом для надійності та простоти визначення мережі є те, що кожен параметр байесовської мережі має точний сенс в термінах лише невеликої множини змінних.

9.3 Побудова байесовських мереж

Наступним етапом після визначення значень ймовірностей байесовської мережі є побудова самої мережі таким чином, щоб результуючий спільний розподіл був адекватним представленням певної проблемної області. Попереднім рівнянням про умовний розподіл можна керуватись для задання топології мережі. Наприклад, можна переписати рівняння, використовуючи правило множення ймовірностей. В результаті отримаємо наступне рівняння:

$$P(x_1,\dots,x_n)=P(x_n|x_{n-1},\dots,x_1)\cdot P(x_{n-1},\dots,x_1) \quad (9.4)$$

Повторюючи цей процес перетворення кожної наступної спільної ймовірності в умовну ймовірність, в кінцевому результаті буде отримане наступне рівняння. Ця тотожність називається **ланцюговим правилом** (*chain rule*).

$$P(x_1,\dots,x_n)=P(x_n|x_{n-1},\dots,x_1)\cdot P(x_{n-1}|x_{n-2},\dots,x_1)\dots P(x_2|x_1)\cdot P(x_1)=\prod_{i=1}^{n} P(x_i|x_{i-1},\dots,x_1) \quad (9.5)$$

Побудована байесовська мережа буде правильним представленням проблемної області, тільки якщо кожна вершина в ній буде умовно незалежною від попередніх. Цю умову можна задовільнити, притримуючись наступних правил:

1. **Вершини.** Спочатку визначається множина змінних, які необхідні для моделювання проблемної області. Ці змінні далі впорядковуються таким чином, щоб змінні “причини” **передували** змінним “наслідкам”. При такому впорядкуванні результуюча мережа буде більш компактною.

2. **Ребра**. (1) Для кожної *i*-вершини обирається мінімальна кількість батьківських вершин. (2) Для кожної батьківської вершини додається ребро зі стрілкою у напрямі *i*-вершини, що задає **причинно-наслідковий** зв’язок.

3. **Таблиця ймовірностей.** Для всіх вершин задається таблиця умовних ймовірностей $P(X_i|\textit{батьківські}(X_i))$

В загальному вигляді, множина батьківських вершин для *i*-вершини повинна включати всі ті вершини, які безпосередньо впливають на дану *i*-вершину.

Оскільки, кожна вершина в мережі поєднується тільки із попередніми вершинами, це виключає можливість появи циклів. Ще однією важливою властивістю байесовської мережі є те, що вона не містить надлишкових значень ймовірностей. Цей факт виключає можливість появи несумісностей.

9.4 Приклади ймовірнісних запитів та розрахунки

Розглянемо наступні чотири запити:

1. яка ймовірність, що не має палива (НП), якщо автомобіль не завівся (НЗ)? (рис. 9.5)

2. яка ймовірність, що присутня проблема з електрикою (ПЕ), якщо автомобіль не завівся (НЗ)? (рис. 9.6)

3. яка ймовірність, що пройшов дощ (Д), якщо автомобіль **не завівся** (НЗ)? (це не прогноз погоди по автомобілю :) (рис. 9.7)

4. яка ймовірність, що автомобіль **не заведеться** (НЗ), якщо пройшов дощ (Д)? (рис. 9.8)

Рисунок 9.3 — Байесовська мережа для прикладу розрахунків ймовірностей запитів

$$P(НЗ, НП, ПЕ, Д) = P(НЗ \mid НП, ПЕ) \cdot P(ПЕ \mid Д) \cdot P(НП) \cdot P(Д)$$

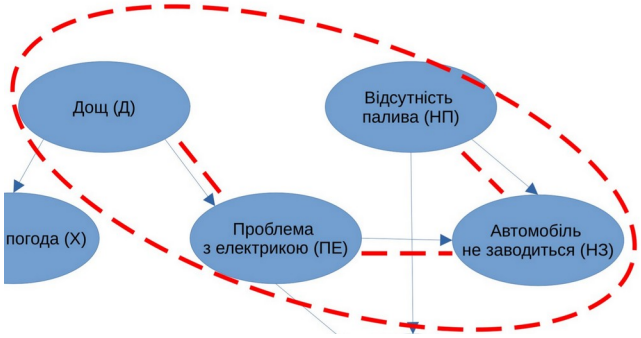


Рисунок 9.4 — Змінні, що входять у повний спільний розподіл ймовірностей

$$P(НП \mid НЗ) = \frac{\sum_{Д,\ ПЕ\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д)}{\sum_{НП,\ ПЕ,\ Д\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д)} = \frac{0{,}00999}{0{,}05188} = 0{,}1925$$

$$\sum_{Д,\ ПЕ\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д) = (0{,}999 \cdot 0{,}98 \cdot 0{,}01 \cdot 0{,}8)_{FF} + (0{,}999 \cdot 0{,}02 \cdot 0{,}01 \cdot 0{,}8)_{FT} +$$
$$+ (0{,}999 \cdot 0{,}96 \cdot 0{,}01 \cdot 0{,}2)_{TF} + (0{,}999 \cdot 0{,}04 \cdot 0{,}01 \cdot 0{,}2)_{TT} = 0{,}00999$$

$$\sum_{НП,\ ПЕ,\ Д\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д) = (0{,}02 \cdot 0{,}98 \cdot 0{,}99 \cdot 0{,}8)_{FFF} + (0{,}02 \cdot 0{,}96 \cdot 0{,}99 \cdot 0{,}2)_{FFT} +$$
$$+ (0{,}95 \cdot 0{,}02 \cdot 0{,}99 \cdot 0{,}8)_{FTF} + (0{,}95 \cdot 0{,}04 \cdot 0{,}99 \cdot 0{,}2)_{FTT} +$$
$$+ (0{,}999 \cdot 0{,}98 \cdot 0{,}01 \cdot 0{,}8)_{TFF} + (0{,}999 \cdot 0{,}96 \cdot 0{,}01 \cdot 0{,}2)_{TFT} +$$
$$+ (0{,}95 \cdot 0{,}02 \cdot 0{,}01 \cdot 0{,}8)_{TTF} + (0{,}999 \cdot 0{,}04 \cdot 0{,}01 * 0{,}2)_{TTT} +$$
$$= 0{,}05188$$

Автомобіль не заводиться (НЗ)			
ПЕ	НП	T	F
F	F	0,02	0,98
F	T	0,999	0,001
T	F	0,95	0,05
T	T	0,999	0,001

Проблема з електрикою (ПЕ)		
Дощ	T	F
T	0,04	0,96
F	0,02	0,98

Відсутність палива (НП)	
T	F
0,01	0,99

Дощ (Д)	
T	F
0,2	0,8

Рисунок 9.5 — Розрахунок ймовірності для 1-го запиту

$$P(ПЕ \mid НЗ) = \frac{\sum_{НП,\ Д\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д)}{\sum_{НП,\ ПЕ,\ Д\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д)} = \frac{0{,}02281}{0{,}05188} = 0{,}4397$$

$$\sum_{НП,\ Д\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д) = (0{,}95 \cdot 0{,}02 \cdot 0{,}99 \cdot 0{,}8)_{FF} + (0{,}95 \cdot 0{,}04 \cdot 0{,}99 \cdot 0{,}2)_{FT} +$$
$$+ (0{,}999 \cdot 0{,}02 \cdot 0{,}01 \cdot 0{,}8)_{TF} + (0{,}999 \cdot 0{,}04 \cdot 0{,}01 \cdot 0{,}2)_{TT} = 0{,}02281$$

Автомобіль не заводиться (НЗ)			
ПЕ	НП	T	F
F	F	0,02	0,98
F	T	0,999	0,001
T	F	0,95	0,05
T	T	0,999	0,001

Проблема з електрикою (ПЕ)		
Дощ	T	F
T	0,04	0,96
F	0,02	0,98

Відсутність палива (НП)	
T	F
0,01	0,99

Дощ (Д)	
T	F
0,2	0,8

Рисунок 9.6 — Розрахунок ймовірності для 2-го запиту

$$P(Д \mid НЗ) = \frac{\sum_{НП,\ ПЕ\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д)}{\sum_{НП,\ ПЕ,\ Д\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д)} = \frac{0{,}01332}{0{,}05188} = 0{,}2567$$

$$\sum_{НП,\ ПЕ\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д) = (0{,}02 \cdot 0{,}96 \cdot 0{,}99 \cdot 0{,}2)_{FF} + (0{,}95 \cdot 0{,}04 \cdot 0{,}99 \cdot 0{,}2)_{FT} +$$
$$+ (0{,}999 \cdot 0{,}96 \cdot 0{,}01 \cdot 0{,}2)_{TF} + (0{,}999 \cdot 0{,}04 \cdot 0{,}01 \cdot 0{,}2)_{TT} = 0{,}01332$$

Автомобіль не заводиться (НЗ)			
ПЕ	НП	T	F
F	F	0,02	0,98
F	T	0,999	0,001
T	F	0,95	0,05
T	T	0,999	0,001

Проблема з електрикою (ПЕ)		
Дощ	T	F
T	0,04	0,96
F	0,02	0,98

Відсутність палива (НП)	
T	F
0,01	0,99

Дощ (Д)	
T	F
0,2	0,8

Ймовірність того, що пройшов дощ, якщо автомобіль не завівся дорівнює 25,6%

Рисунок 9.7 — Розрахунок ймовірності для 3-го запиту

$$P(НЗ \mid Д) = \frac{\sum_{НП,\ ПЕ\ є\ \{T,\ F\}} P(НЗ, НП, ПЕ, Д)}{P(Д)} = \frac{0{,}01332}{0{,}2} = 0{,}067$$

Автомобіль не заводиться (НЗ)			
ПЕ	НП	T	F
F	F	0,02	0,98
F	T	0,999	0,001
T	F	0,95	0,05
T	T	0,999	0,001

Проблема з електрикою (ПЕ)		
Дощ	T	F
T	0,04	0,96
F	0,02	0,98

Відсутність палива (НП)	
T	F
0,01	0,99

Дощ (Д)	
T	F
0,2	0,8

Ймовірність того, що автомобіль не заведеться, якщо пройшов дощ дорівнює 6,7%

Рисунок 9.8 — Розрахунок ймовірності для 4-го запиту

ТЕМА 10
НАВЧАННЯ НА ОСНОВІ СПОСТЕРЕЖЕННЯ

В даній розглянемо агентів, які здатні **покращувати** свою поведінку за рахунок ретельного вивчення накопиченого раніше досвіду та власних прогнозів.

Агент може **навчатись**, якщо він покращує свою продуктивність в результаті **спостереження** за навколишнім середовищем. Якщо говорити про людину, то вона постійно навчається на власному досвіді. Говорячи про навчання комп’ютерного агента, це процес називають **машинним навчанням**. В ході даного процесу комп’ютер опрацьовує вхідні дані та будує на їх основі певну модель. Потім ця модель використовується в якості гіпотези про навколишнє середовище, а також як частина програмного забезпечення, яке здатне вирішувати поставлені задачі.

Можуть постати логічні питання: “навіщо потрібно, щоб машина або комп’ютерний агент, мав можливість навчатись?”, “хіба не простіше запрограмувати всі необхідні функції та поведінку в програмному коді, в процесі розробки такого агента?”. На ці питання можна дати дві відповіді:

1. **По-перше**, розробник подібного комп’ютерного інтелектуального агента або системи не може **передбачити** всі ситуації, які можуть трапитись із цією системою в майбутньому в процесі вирішення, поставлених перед нею задач. Наприклад, інтелектуальний агент — робот, задача якого знаходити шлях в приміщенні, повинен вивчати схему кожного нового приміщення, щоб ефективно в ньому діяти. Або, програма для прогнозування цін повинна мати можливість адаптуватись до різних умов на ринку.

2. **По-друге**, іноді розробник в найпростішому випадку **не має** точного **представлення** про те, як можна запрограмувати бажану поведінку інтелектуальної системи. Подібна проблема справедлива для природних задач з якими стикається людина. Більшість із ним ми вирішуємо на **підсвідомості**, через що ці процеси доволі **важко перенести** у вигляд чітких алгоритмів для

комп’ютерних програм. Людина не витрачає багато зусиль, щоб розпізнавати обличчя, розуміти природну мову, читати і багато чого іншого. Але для **комп’ютера** це може стати досить **складною** задачею, як з точки зору програмування її вирішення так, і з точки зору використання обчислювальних ресурсів на її вирішення. Тому рішенням, яке добре себе зарекомендувало в подібних випадках, стало застосування алгоритмів **машинного навчання**.

10.1 Машинне навчання

Будь-який компонент програми інтелектуального агента може бути вдосконалений за рахунок використання машинного навчання. Покращення та методи, які можуть бути застосовані для цього залежать від:

1. **Компоненту**, який має бути вдосконалений;
2. **Первинні знання**, якими володіє агент;
3. **Представлення знань**, яке використовує агент, і **аспекти середовища**, які впливають на модель, що будує агент;
3. Наявних **даних** та **механізмів** для зворотного зв’язку.

Попередніми темами вже розглядались деякі конструкції агентів до компонентів яких відносились:

1. Засоби прямого відображення **умов** поточного стану в **дії**;
2. Засоби формування **логічного висновку** про релевантні аспекти середовища, базуючись на послідовності **результатів сприйняття**;
3. Інформація про **зміни** в навколишньому середовищі та **результати** можливих дій, які агент може виконати;
4. Інформація про **корисність**, яка вказує наскільки бажаним є певний стан середовища;
5. Інформація про **ціну** дії;
6. **Мета**, яка визначає найбільш бажані стани;
7. Генератор проблем, критик та компонент навчання, які дозволяють **покращувати** систему.

Кожен із цих компонентів можна навчати. Для прикладу розглянемо агента, який керує автомобілем — автопілот. Цього агента можна доповнити можливістю навчатись, спостерігаючи за людиною-інструктором. Кожен раз, коли інструктор виконує певну дію, припустимо “гальмує”, агент може вивчати чергове правило “умова-дія” відносно того, коли слід гальмувати. В такому випадку навчається 1-й компонент. Переглядаючи множину зображень із камери, на яких, як завчасно відомо, показані “автобуси”, агент може навчитись розпізнавати автобуси (2-й компонент). Спробувавши виконати дії та спостерігаючи за їх результатом, наприклад, керування на слизькій дорозі, агент може вивчати результати своїх дій (3-й компонент). Отримуючи відгуки від пасажирів, яким не сподобалась поїздка через певні причини, агент зможе навчити відповідний компонент для покращення загальної функції корисності (4-й компонент).

Технології машинного навчання вже стали стандартним елементом програмної інженерії. Коли створюються системи програмного забезпечення, які не обов’язково пов’язані зі штучним інтелектом, потенційно компоненти цього ПЗ можуть бути покращенні за рахунок застосування машинного навчання.

В цілому **навчання** компонентів **інтелектуального агента** передбачає, що спочатку він має **невеликий** обсяг попередніх знань. Агент починає цей процес майже з нуля і вчиться лише на основі даних, які до нього надходять. На противагу існує інший варіант навчання, який називається **трансферним навчанням** (*transfer learning*) або перенесенням знань. При такому варіанті накопичені знання з однієї проблемної області **передаються** в нову проблемну область. Це дозволяє пришвидшити процес навчання і також зменшує кількість даних, які потрібні для навчання.

10.2 Структура даних та задачі навчання

Зазвичай для навчання застосовуються вхідні дані у **розгорнутому представлені**, у вигляді вектору значень атрибутів. Також можливі ситуації, коли на вхід системи будуть поступати дані будь-якої іншої структури такої, як **атомарної** або **реляційної**.

Якщо вихідні дані — результат обирається із завчасно відомої множини значень, наприклад, назва категорій або *true/false*, то така задача називається **класифікацією**. В разі, якщо результат представляється у вигляді числового значення, тоді така задача навчання називається **регресією**.

10.3 Типи навчання

Існує **три типи** зворотного зв’язку, що пов’язані із вхідними даними, які визначають основні типи навчання:

1. При **навчанні із вчителем** (*supervised learning*) агент **спостерігає** за парами “вхідних” та “вихідних” даних (стимул-реакція) і знаходить функцію, яка відображає вхідні дані на вихідні. Наприклад, вхідними даними можуть бути зображення, кожне з яких супроводжується певним вихідним визначенням: “будівля”, “дерева”, “тварини”, ... Таке визначення називається **міткою** (*label*) або назвою класу. Агент знаходить функцію, яка для кожного нового зображення **передбачає** для нього відповідну мітку.

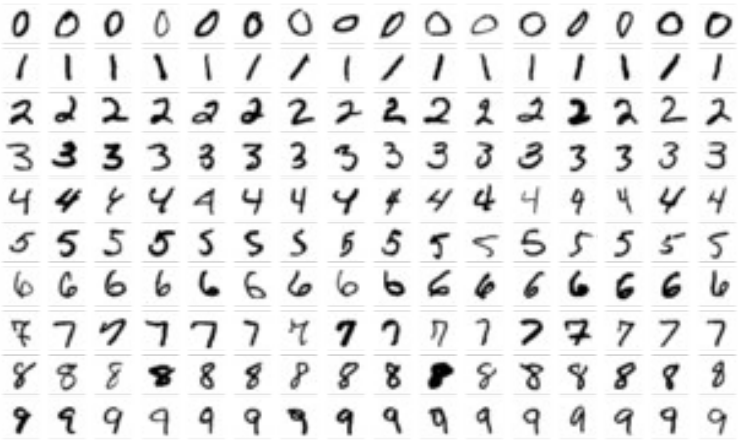


Рисунок 10.1 — Ілюстрація набору даних *MNIST*

2. При **навчанні без вчителя** (*unsupervised learning*) агент вивчає із вхідних даних певні шаблони або **закономірності** без якогось явного зворотного зв’язку. Найбільш розповсюдженою задачею навчання без вчителя — є **кластеризація.** Під цією задачею розуміється виявлення у вхідних прикладах потенційно корисних закономірностей. Наприклад, коли агенту надаються мільйони знімків із Інтернету, система комп’ютерного зору може ідентифікувати великий кластер або групу схожих за своїм вмістом зображень, які людина ідентифікувала би як зображення із “собаками”.

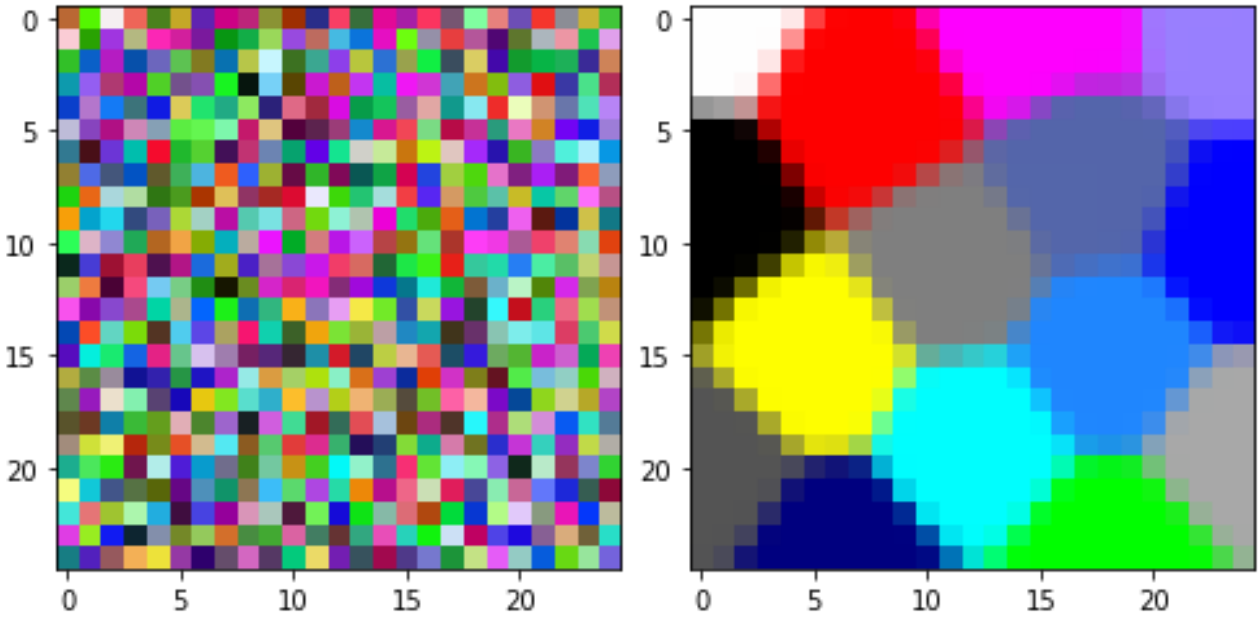


Рисунок 10.2 — Групування кольорів у кластери

3. При **навчанні з підкріпленням** (*reinforcement learning*) агент вчиться на серії підкріплень у вигляді **нагород** або **штрафів**. Наприклад агенту, який грає в шахи, в кінці гри говорять, що він виграв — “нагорода” або програв — “штраф”. Потім на агента покладається задача визначити дії, виконання яких в найбільшій мірі призвели до отримання відповідного результату, і змінити їх таким чином, щоб в подальшому мати можливість отримати більшу винагороду.

10.4 Навчання із вчителем

Формально задача навчання із вчителем формулюється наступним чином. При наявності **набору даних** для навчання у вигляді вхідних та вихідних значень (іншими словами це називається **прецедентами**):

$$(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N) \tag{10.1}$$

які згенеровано невідомою функцією $y = f(x)$ потрібно знайти функцію *h*, яка **апроксимує** істинну функцію *f*.

Функція *h* називається **гіпотезою** про середовище. Вона обирається із простору гіпотез η можливих функцій. Вихідне значення y_i називається істинним або **еталонним** значенням (*ground truth*). Воно представляє собою правильну відповідь, яку модель повинна надати у відповідь на запит x_i .

10.5 Простір гіпотез

Тепер розглянемо яким чином обрати простір гіпотез — невідомої функції. При першому варіанті можна спиратись на попередні **знання** про процес, який згенерував дані. Якщо такі знання відсутні, тоді можна виконати попередній **аналіз** даних: дослідити їх за допомогою статистичних тестів та візуалізації — графіки, гістограми, діаграми розсіювання. Це дозволить отримати деяке представлення про дані і зробити висновки про те, який простір гіпотез може підійти краще. Другий варіант передбачає **перевірку** декількох варіантів просторів гіпотез з метою оцінки, який підійде краще під конкретні дані.

Обрання кращої гіпотези передбачає підбір функції *h* , яка буде такою, що для кожного стимулу x_i в навчальному наборі вона буде видавати реакцію $h(x_i) = y_i$. У випадку неперервної області значень результатів не слід очікувати точного співпадіння реакції із істинним значенням. В якості найкращої гіпотези тут слід шукати таку функцію, для якої кожне значення $h(x_i)$ буде достатньо **наближеним** до y_i .

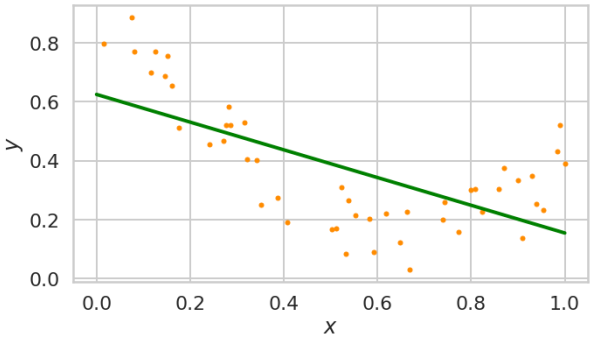


Рисунок 10.3 — Знаходження гіпотези для значень навчального набору даних

10.6 Ступінь істинності гіпотези

Ступінь істинності гіпотези визначається не тим, наскільки точно вона працює на **навчальному** наборі, а наскільки добре вона працює на даних, які до цього ще **не зустрічались**. Цей показник можна оцінити за допомогою другого набору пар (x_1, y_1) , які прийнято називати **тестувальним набором**. При цьому можна стверджувати, що функція *h* добре **узагальнює** дані, в разі якщо вона точно передбачає відгуки із тестового набору.

Для демонстрації, що функція *h* , яка була підібрана алгоритмом навчання, залежить від простору гіпотез і заданого навчального набору даних, розглянемо невеликий приклад. Даними є значення від невеликого періоду функції синуса із випадково доданим шумом. Нижче на трьох графіках представлені найбільш підходящі гіпотези із певного простору гіпотез.

1. Простір гіпотез — **прямої лінії** (рис. 10.4), а саме функція виду:

$$h(x_i) = \omega_1 \cdot x + \omega_0 \qquad (10.2)$$

Як можна побачити для такого варіанту відсутня лінія, яка представляє узгоджену гіпотезу із даною множиною точок.

2. Простір гіпотез — **кусково-лінійної функції** (рис. 10.5), де кожен лінійний сегмент поєднує дві суміжні точки. Подібні функції завжди є узгодженими.

3. Простір гіпотез — **поліноми** 7-ї степені (рис. 10.6):

$$h(x_i) = \sum_{i=0}^{7} \omega_i \cdot x^i \qquad (10.3)$$

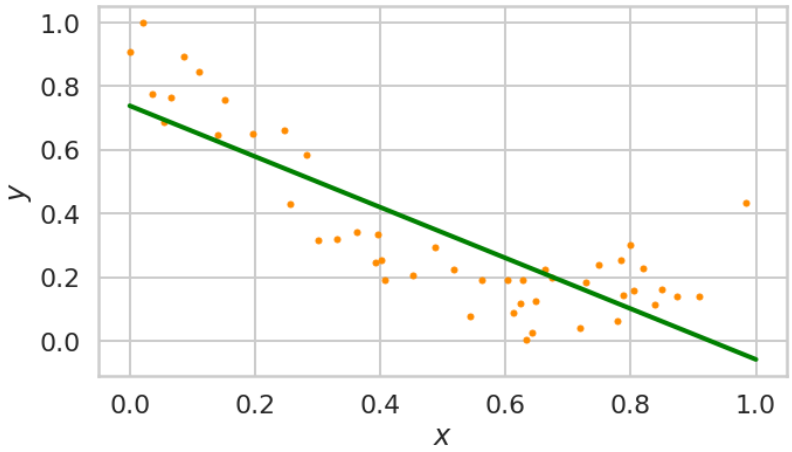


Рисунок 10.4 — Простір гіпотез прямої лінії

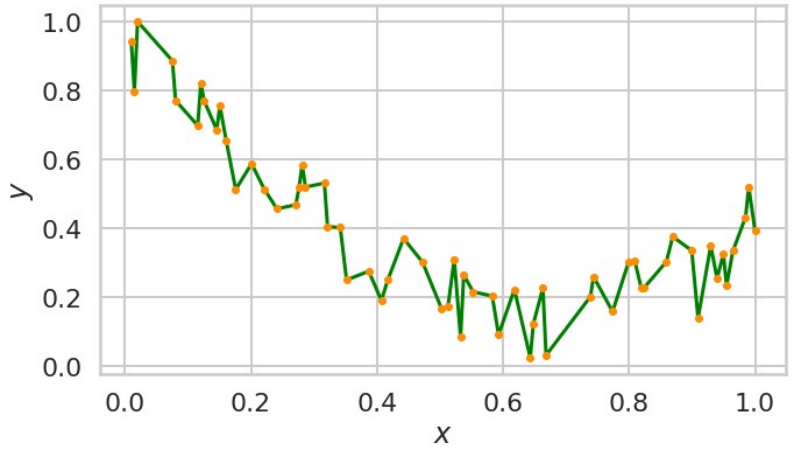


Рисунок 10.5 — Простір гіпотез — кусково-лінійні функції

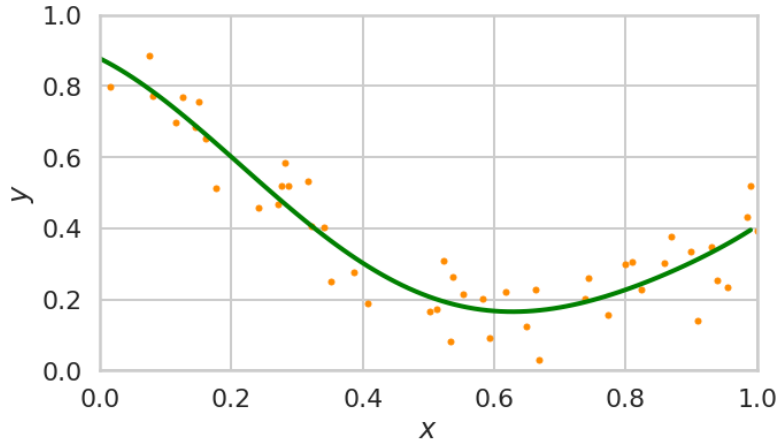


Рисунок 10.6 — Простір гіпотез — поліноми

10.7 Аналіз простору гіпотез

Одним із способів **аналізу** простору гіпотез є оцінка зміщення, яке воно накладає, незалежно від навчального набору даних, та дисперсію, яку воно створює.

Під **зміщенням** (*bias*) розуміється тенденція обраної гіпотези давати відхилення від очікуваного значення при усередненні по різним навчальним наборам. Зміщення часто є результатом обмежень, які накладаються простором гіпотез. Говорять, що гіпотеза проявляє ознаки **недостатнього навчання** (*underfitting*) в разі, якщо вона не дозволяє виявити певну закономірність в існуючих даних.

Під **дисперсією** (*variance*) розуміють кількість змін в гіпотезі у зв'язку із розкидом значень в навчальному наборі. Говорять, що функція є **перенавченою** (*overfitting*), якщо вона приділяє занадто багато уваги конкретним значенням даних в навчальному наборі і як результат погано працює із раніше невідомими значеннями даних.

10.7.1 Приклад зміщення

Наприклад, простір гіпотез лінійної функції дає **сильне зміщення**, оскільки воно допускає лише функції, представлені прямими лініями. Якщо в даних є шаблони або закономірності, які відрізняються від загального нахилу прямої, лінійна функція не зможе представити таку закономірність. З іншого боку кусково-лінійні функції має **мале зміщення**, оскільки її форма повністю визначається самими даними.

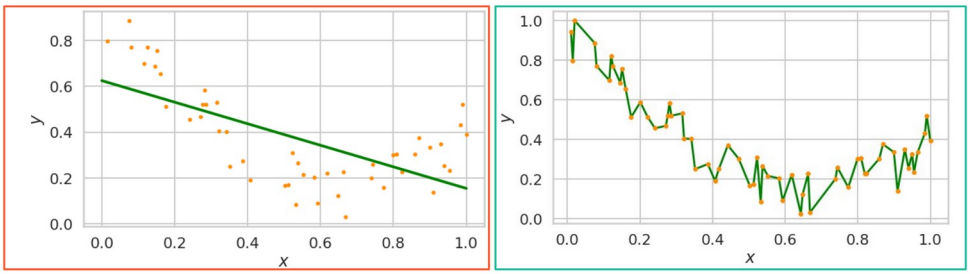


Рисунок 10.7 — Приклад сильного (лівий) та малого (правий) зміщення

10.7.2 Приклад дисперсії

Для прикладу ці варіанти графіків (рис. 10.8) представляються різними наборами даних, які були сформовані із інших значень тієї самої функції. Ці набори вийшли трохи різними. Для функцій лінійна та поліном 7-ї степені із певною регуляризацією ці відмінності призводять до невеликої різниці в гіпотезах. Подібна ситуація називається **низькою дисперсією**. Однак для іншого поліному 25-ї степені це призвело до суттєвих відмінностей, хоча це є одна і та сама функція. Це випадок **високої дисперсії**.

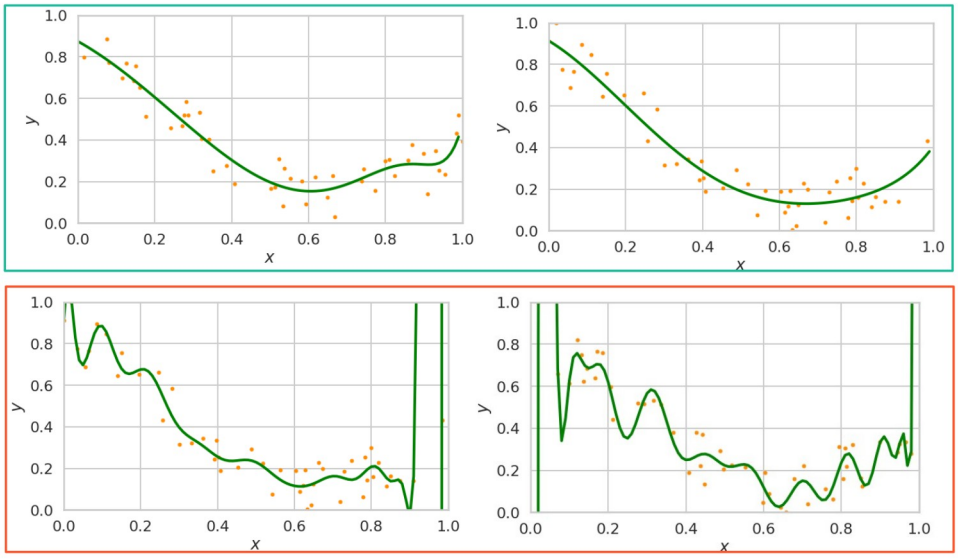


Рисунок 10.8 — Приклад низької (зверху) та високої (знизу) дисперсії

В ході навчання необхідно знаходити **компроміс** між зміщенням та дисперсією, тобто робити вибір між більш складними гіпотезами із невеликим зміщенням, що забезпечують хорошу відповідність між даними навчального набору, та більш простими гіпотезами із низькою дисперсією, які забезпечують краще узагальнення.

10.8 Обрання моделі та оптимізація

В машинному навчанні головною задачею є обрання такої гіпотези, яка буде оптимально відповідати майбутнім прикладам. Під “майбутніми прикладами” розуміються інші дані, які не присутні в навчальному наборі, але для них можна зробити припущення про їх **подібність** до навчальних даних. Подібне припущення називається **стаціонарністю**. Ніяке навчання не матиме сенсу в разі, якщо в подальшому дані на яких буде працювати обрана краща гіпотеза будуть неподібними.

Також визначення “оптимальної відповідності” має під собою розуміння того, що певна гіпотеза буде мінімізувати частоту появи помилкових відповідей. Тобто доля випадків, коли $h(x)=y$ для пари (x,y) повинна бути більшою.

Частоту появи помилок можна оцінити перевіряючи відповіді, які дає гіпотеза, на **тестовому наборі**. Логічно, що буде доволі мало інформативності в разі перевірки гіпотези на наборі на якому вона навчалась, тобто на навчальному наборі. Найбільш простий спосіб, щоб вирішити цю проблему, це розділити всі наявні дані для навчання на два набори: **навчальний** та **тестувальний**. Якщо передбачається обирати тільки одну найкращу гіпотезу-модель, тоді такий підхід матиме гарний результат. Але часто необхідно порівнювати декілька різних гіпотез при різних значеннях параметрів навчання. Такі параметри навчання називаються **гіперпараметрами**. В такій ситуації можуть виникати ситуація, коли процес навчання в цілому, враховуючи самого розробника, неодноразово будуть використовувати цей **тестувальний набір**. Це може призвести до **неадекватності** отриманих оцінок, результати яких в певному непрямому відношенні будуть враховувати особливості цього набору для тестування. Щоб уникнути такої ситуації, необхідно відмовитись від використання тестувального набору в процесі навчання і використовувати його лише для перевірки **фінальної** гіпотези. Отже з цього випливає необхідність у додатковому третьому наборі даних. В цілому отримаємо наступні поділи даних:

1. **Навчальний набір** (*training set*), призначений для навчання моделей-кандидатів;

2. **Перевірочний набір** (*validation set*) або набір для розробки (*dev set*), призначений для оцінок моделей кандидатів і обрання найкращої;

3. **Тестувальний набір** (*testing set*), призначений для проведення фінальної об’єктивної оцінки найкращої моделі.

Однак часто виникає ситуація коли не вистачає даних, щоб сформувати три окремих набори даних. Для вирішення цієї проблеми застосовують метод під назвою **“перехресна-валідація”** (*k-fold cross validation*). Ідея методу полягає в тому, щоб використовувати кожен приклад даних в наборі за двома призначеннями, як навчальний, так і як перевірочний, але не в один і той самий момент часу. Для цього спочатку дані розбиваються на *k* рівних **підмножин**. Після виконується *k* **раундів** навчання, при чому на кожному рунді *1/k* даних резервується для перевірки, а інша для навчання. Вважається, що середня оцінка для тестових наборів за *k* раундів буде мати більшу інформативність, а ніж одна оцінка. Зазвичай використовують значення *k* рівним 5 або 10, що дає можливість отримати доволі непогану оцінку моделі.

10.8.1 Втрати

До цього зазначалось, що в процесі навчання потрібно обирати моделі, які матимуть **меншу частоту** появи помилкових рішень. Але в деяких задачах помилка не завжди дозволяє якісно оцінити навчану модель. Для прикладу розглянемо задачу класифікації спаму в пошті. Доволі логічним є те, що **гірше** класифікувати листи, які не є спамом, як спам. Це може призвести до втрати потенційно важливого повідомлення. Напротивагу, класифікація спаму, як не спам, може нести за собою **менші** проблеми. В цілому до скриньки випадково потрапить непотрібне рекламне повідомлення. З цього слідує, що класифікатор із частотою помилки 1%, у якого майже всі помилки пов’язані із невірною класифікацією спаму як не спам, буде кращім, а ніж класифікатор із частотою помилок 0,5%, але де більшість помилок є класифікація не спаму, як спам.

У попередніх темах розглядалось, що для інтелектуальної системи або агента потрібно максимізувати очікувану корисність. Однак в машинному навчанні цей аспект традиційно виражається у від’ємному ракурсі, а саме: **мінімізації функції втрат**. Функція втрат $L(x, y, \hat{y})$ визначається як кількість втраченої корисності за рахунок видачі прогнозу $h(x)=\hat{y}$, коли в дійсності правильна відповідь була $f(x)=y$.

Загалом в машинному навчанні найчастіше використовують функції:

1. **Абсолютної оцінки втрат**;

$$L_1=\frac{\sum_{i=0}^{n}|y_i-\hat{y}_i|}{n} \tag{10.4}$$

2. **Квадратична оцінка втрат**;

$$L_2=\frac{\sum_{i=0}^{n}(y-\hat{y})^2}{n} \tag{10.5}$$

3. **Крос ентропія**.

$$L_{CE}=-\sum_{i=0}^{n} y_i \log(\hat{y}_i) \tag{10.6}$$

ТЕМА 11
АЛГОРИТМИ МАШИННОГО НАВЧАННЯ

11.1 Дерева рішень

Дерева рішень є засобом представлення функцій, які відображають вектор значень атрибутів на єдине значення — рішення. Алгоритм дерева рішень приходить до рішення за рахунок виконання серії перевірок умов, починаючи від кореня і переміщуючись по відповідній гілці, поки не буде досягнуто листове значення — рішення. Кожна внутрішня вершина дерева відповідає перевірці значення одного із вхідних атрибутів. Дерева рішень дозволяють представляти будь-яку булеву функцію. Для забезпечення ефективного знаходження простого, сумісного дерева рішення застосовують евристичний метод **прирощування інформації**. В цілому задача навчання дерева рішень полягає у побудові такого дерева, яке буде мати найменший розмір і дозволить однозначно визначати рішення, яке відповідає певним значенням вхідних атрибутів. Для цього використовуються певні функції евристики, які дозволяють визначати найбільш важливі атрибути для формування менших дерев.

Багато алгоритмів машинного навчання вже реалізовано у вигляді **бібліотек** для різних мов програмування. Наприклад, для мови *Python* є бібліотека для машинного навчання *scikit-learn* в якій реалізовані популярні алгоритми. Застосувавши її можна виконати побудову оптимального дерева рішень для потрібного набору даних.

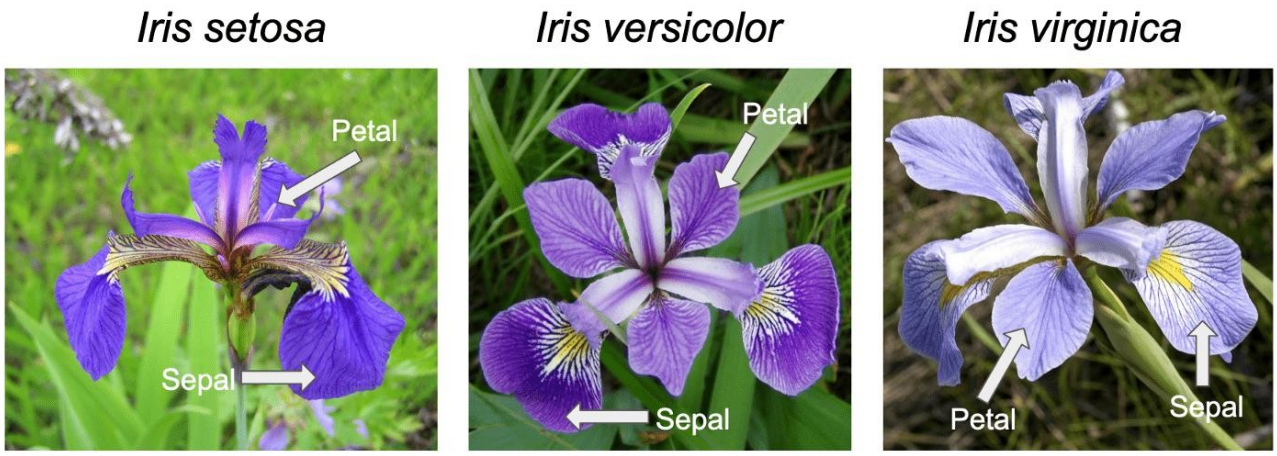


Рисунок 11.1 — Задача класифікації квітів ірисів “*The Iris Dataset*” за допомогою дерева рішень

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	flower
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

Рисунок 11.2 — Приклад набору даних для задачі класифікації квітів ірисів

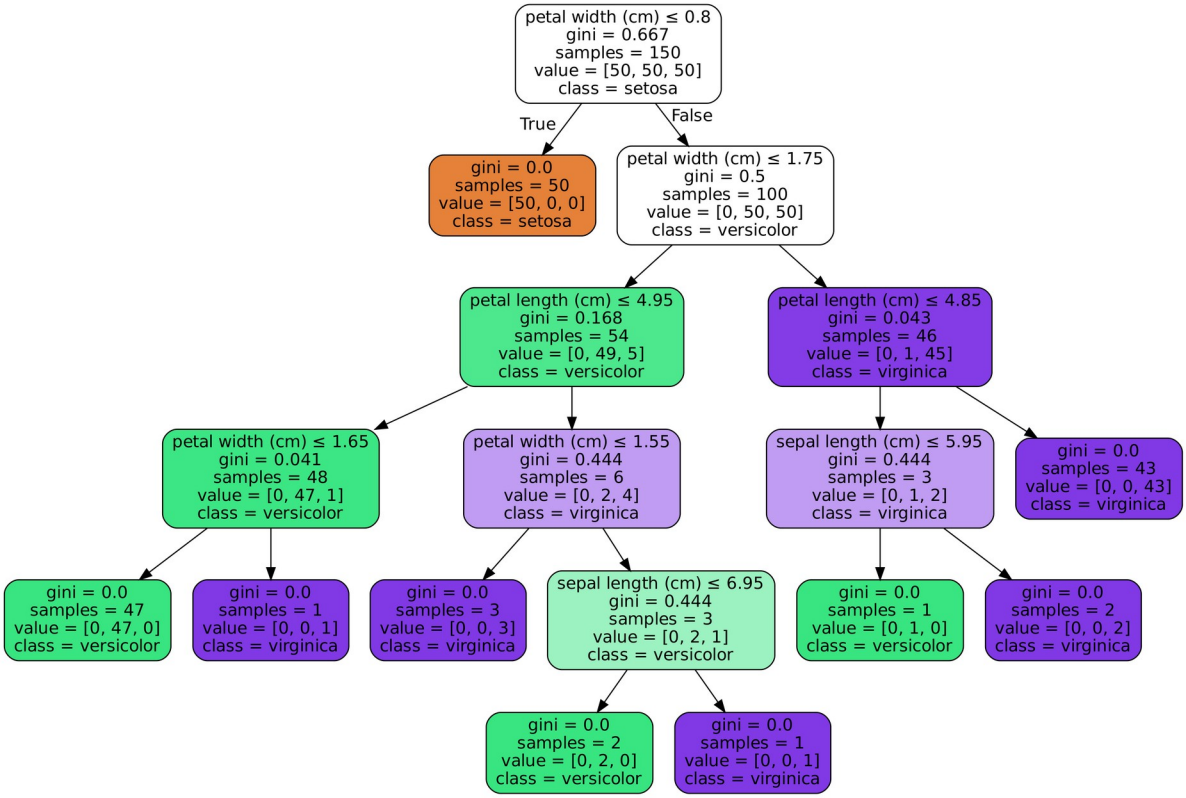


Рисунок 11.3 — Дерево рішень для задачі класифікації квітів ірисів

11.2 Лінійна регресія та класифікація

Одним із просторів гіпотез, який часто використовується у машинному навчанні є клас **лінійних функцій** із неперервним значенням вхідних параметрів. Найпростішим випадком навчання для даного класу функцій є **одномірна лінійна регресія**.

Проста лінійна функція — пряма лінія із вхідними параметрами x і вихідним значенням y має наступний вигляд:

$$y = w_1 \cdot x + w_0, \tag{11.1}$$

де w_1 та w_0 є числовими коефіцієнтами, які навчаються. Ці коефіцієнти називаються **ваговими** (*weights*), що в цілому дозволяє змінювати вихідне значення y за допомогою зміни одного або іншого коефіцієнту. В результаті, маємо вектор ваг (w_0, w_1). Цей вектор дозволяє записати гіпотезу у вигляді лінійної функції, що матиме дані вагові коефіцієнти:

$$h_w(x) = w_1 \cdot x + w_0, \tag{11.2}$$

Навчальний набір для навчання функції представляється у вигляді множини точок $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, які представляють певну невідому функцію.

Задача лінійної регресії полягає у знаходженні гіпотези h_w , яка буде найкращим чином відповідати заданим даним. Щоб "підігнати" відповідну пряму лінію, потрібно буде знайти такі значення вагових коефіцієнтів (w_0, w_1), які **мінімізуватимуть** емпіричні втрати $L(h_w)$. Традиційно для цього застосовують **квадратичну функцію втрат** L_2 , яка сумується по всім прикладам навчального набору.

Для знаходження потрібних значень вагових коефіцієнтів використовуються наступні формули:

Функція втрат:

$$L(h_w) = \sum_{j=1}^{N} L_2(y_j, h_w(x_j)) = \sum_{j=1}^{N} (y_j - h_w(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1 \cdot x_j + w_0))^2 \qquad (11.3)$$

Часткові похідні:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 \cdot x_j + w_0))^2 = 0 \qquad (11.4)$$

$$\frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (w_1 \cdot x_j + w_0))^2 = 0 \qquad (11.5)$$

Рішення часткових похідних:

$$w_1 = \frac{N \cdot (\sum x_j \cdot y_j) - (\sum x_j) \cdot (\sum y_j)}{N \cdot (\sum x_j^2) - (\sum x_j)^2} \qquad (11.6)$$

$$w_0 = \frac{(\sum y_j - w_1 \cdot (\sum x_j))}{N} \qquad (11.7)$$

Для ілюстрації лінійної регресії розглянемо приклад невідомої лінійної функції (рис. 11.4, верхній) для якої задана певна множина вхідних та вихідних значень (жовті точки). Застосувавши наведені формули, можна знайти оптимальні значення вагових коефіцієнтів. В результаті, після лінійної регресії отримаємо функцію:

$$h_w = 1{,}150939 \cdot x - 0{,}071306 \qquad (11.8)$$

Багато форм навчання включають в себе корегування значень вагових коефіцієнтів з метою мінімізації втрат. Для одномірної лінійної регресії простір,

що визначається вагами w_1 та w_0 , буде тривимірним графіком, який матиме опуклу форму (рис. 11.4, нижній). Ця характеристика буде справедливою для будь-якої задачі лінійної регресії із квадратичною функцією втрат, і це свідчить про відсутність локальних мінімумів. Отже для одномірної лінійної регресії, щоб підігнати лінію під дані, достатньо застосувати зазначені вище формули.

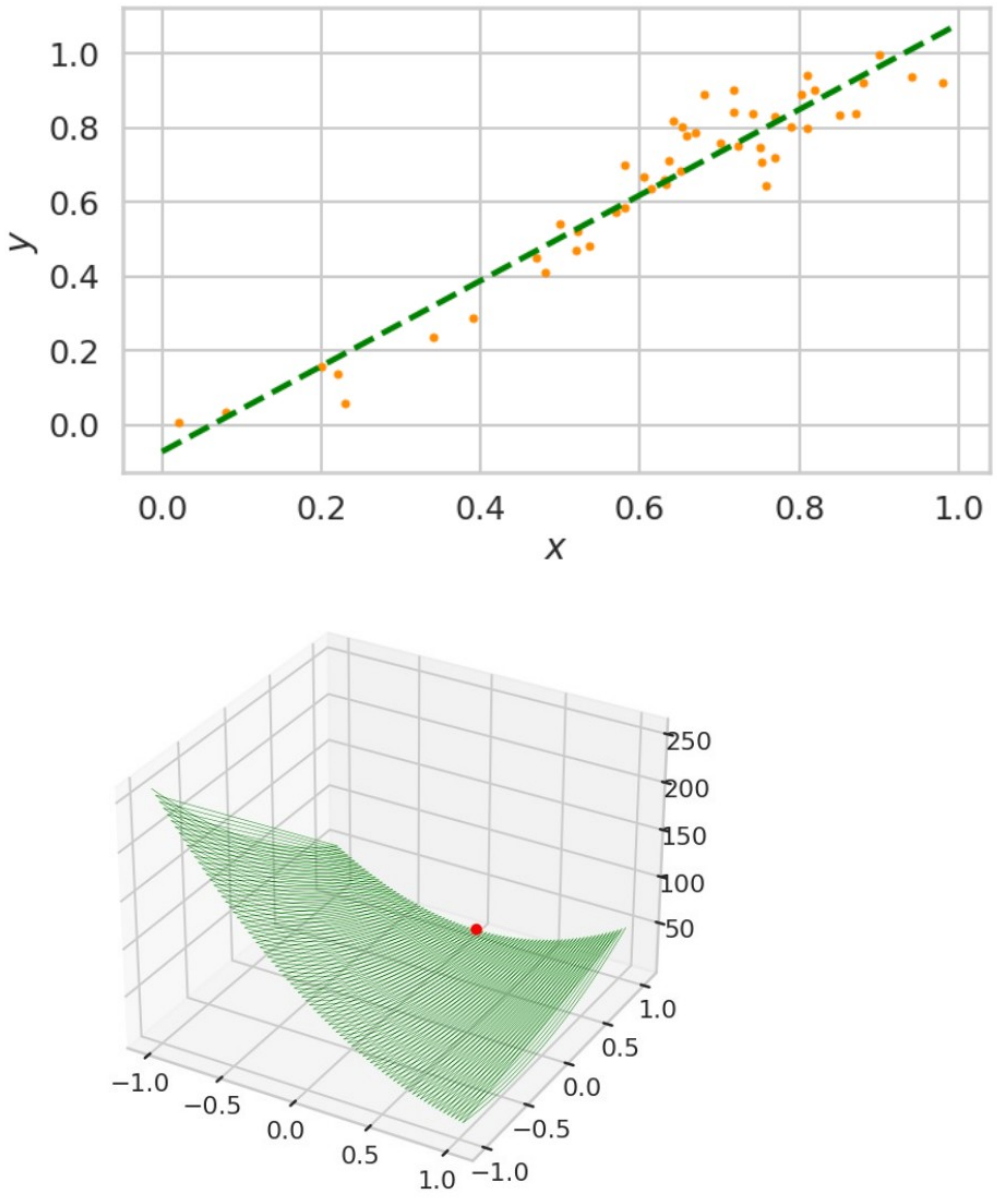


Рисунок 11.4 — Приклад невідомої лінійної функції (зверху) та простір вагових коефіцієнтів для одномірної лінійної регресії (знизу)

11.2.1 Градієнтний спуск

Одновимірна лінійна модель має зручну властивість при якій не складно знайти оптимальне рішення для якого часткова похідна дорівнюватиме нулю. Однак така ситуація трапляється не завжди, тому потрібно мати метод мінімізації втрат, який не вимагає пошуку нульових значень похідної, і який можна буде застосовувати для будь-якої функції втрат.

Пошук в неперервному просторі вагових коефіцієнтів можна виконувати шляхом **поступової зміни** параметрів. Як і до цього основна ідея полягає у мінімізації втрат. Такий метод називається **градієнтним спуском**. Для початку обирається будь-яка початкова точка в просторі вагових коефіцієнтів. Потім розраховується **оцінка** градієнта і точка **зсувається** трохи вниз у напрямі найбільш крутого спуску. Дана дія повторюється до тих пір, поки у просторі ваг не буде досягнуто точки із мінімальними значеннями функції втрат. Загалом формула для розрахунку має такий вигляд:

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} L(w) \tag{11.9}$$

Параметр α застосовується в якості **розміру кроку** при спробі мінімізувати втрати. В задачах машинного навчання цей параметр називається **швидкістю навчання** (*learning rate*). При різних підходах в навчанні цей параметр може представлятись у вигляді константи або зменшуватись із плином часу навчання.

Для випадку одновимірної регресії застосовуються наступні формули для розрахунків градієнтів вагових коефіцієнтів:

$$\frac{\partial}{\partial w_0} L(w) = -2 \cdot (y - h_w(x)) \tag{11.10}$$

$$\frac{\partial}{\partial w_1} L(w) = -2 \cdot (y - h_w(x)) \cdot x \tag{11.11}$$

В результаті підставивши дані формули у загальну формулу і замінивши “2” на невизначену швидкість навчання, матимемо:

$$w_0 \leftarrow w_0 - \alpha \cdot (y - h_w(x)) \tag{11.12}$$

$$w_1 \leftarrow w_1 - \alpha \cdot (y - h_w(x)) \cdot x \tag{11.13}$$

Ці зміни можна інтерпретувати наступним чином, якщо $h_w(x) > y$ необхідно трохи зменшити ваговий коефіцієнт w_0 . Також зменшуємо w_1 , якщо стимул x був додатним, інакше w_1 слід збільшити.

Ця формула застосовується в разі навчального набору із **єдиним** прикладом. В більш загальному вигляді цієї формули для навчального набору із ***N*** **прикладів** буде мінімізуватись сума індивідуальних втрат для кожного прикладу:

$$w_0 \leftarrow w_0 - \alpha \cdot \sum_j (y - h_w(x_j)) \tag{11.14}$$

$$w_1 \leftarrow w_1 - \alpha \cdot \sum_j (y - h_w(x_j)) \cdot x_j \tag{11.15}$$

Ці формули складають правило навчання **методом градієнтного спуску** для одномірної лінійної регресії. Поверхня графіку втрат має опуклу форму. Це означає, що в даному випадку не має локальних мінімумів, в яких міг би застрягнути алгоритм. Отже, матимемо гарантовану збіжність алгоритму мінімізації до глобального мінімуму, при умові, що параметр швидкості навчання не було встановлено у занадто велике значення. Подібне велике значення може привести до нескінченного перестрибування алгоритму через точку глобального мінімуму.

Зазначений процес може бути достатньо повільним, оскільки на кожному етапі необхідно просумувати результати всіх N прикладів навчального набору, і таких етапів може бути дуже багато. Проблема може ускладнюватись в разі, якщо обсяг даних із N прикладів перевищує обсяги пам’яті обчислювальної системи. Кожен етап розрахунків на якому розглядаються всі приклади навчального набору називається **епохою** (*epoch*).

11.2.2 Стохастичний градієнтний спуск

Більш швидкий варіант розглянутого методу називається **стохастичним градієнтним спуском** (*SGD*). При такому підході обирається невелика кількість навчальних прикладів і виконується оновлення вагових коефіцієнтів згідно

розглянутих раніше виразів. Наразі часто застосовують варіант, коли на кожному етапі градієнтного спуску обирається **невелика порція** (*minibatch*) розміром *m* прикладів із навчального набору розміром *N*.

Для деяких архітектур *CPU* та *GPU* можна обирати таке значення *m* , яке дозволить використати переваги використання **розпаралелювання** та **векторних операцій**, що дозволяє виконувати обчислення *m* майже так само швидко як обчислення одного прикладу. З цієї точки зору можна розглядати параметр *m* в якості ще одного **гіперпараметру**, який можна адаптувати під кожну конкретну задачу машинного навчання.

Збіжність методу стохастичним градієнтним спуском із *minibatch* строго не гарантується, оскільки може трапитись ситуація практично нескінченого блуканням навколо мінімуму. Щоб уникати такої проблеми можна змінювати гіперпараметр **швидкості навчання** або виконувати **ранню зупинку** алгоритму навчання, якщо значення навчання не покращуються більше певної межі.

Стохастичний градієнтний спуск широко застосовується для моделей окрім лінійної регресії. До таких моделей можна віднести і **нейронні мережі**. Навіть коли поверхня функції втрат має неопуклу форму, цей підхід у навчанні показав свою ефективність при пошуку локальних мінімумів, які є достатньо близькими до глобальних мінімумів.

11.2.3 Множинна лінійна регресія

Застосування розглянутих методів можна легко розширити на задачі множинної лінійної регресії, в яких кожен приклад x_j представляється ***n*-елементним вектором**. В подібних випадках простір гіпотез представляється у вигляді функцій виду:

$$h_w(x_j) = w_0 + \sum_i w_i \cdot x_{j,i} \qquad (11.16)$$

Тут w_0 є вільним членом який можна замінити на константу, наприклад, “1”. Це спрощує формулу до:

$$h_w(x_j) = \sum_i w_i \cdot x_{j,i} \qquad (11.17)$$

Загалом множинна лінійна регресія не складніша, а ніж одномірна регресія. Метод градієнтного спуску дозволяє досягти єдиного мінімуму функції втрат. Відповідне рівняння для оновлення кожного вагового коефіцієнту матиме вигляд:

$$w_j \leftarrow w_j - \alpha \cdot \sum_j (y - h_w(x_j)) \cdot x_{j,i} \qquad (11.18)$$

11.2.4 Регуляризація

У випадку із одномірною лінійною регресією не було необхідності хвилюватись про **перенавчання.** Однак для множинної лінійної регресії в багатомірному просторі можлива ситуація, що деякі нерелевантні розмірності будуть випадково прийматись за корисні і це буде призводити до перенавчання моделі. Щоб уникнути перенавчання, загальним правилом є застосування **регуляризації** множинних лінійних функцій. Під регуляризацією розуміється накладання **штрафу** на більш складну гіпотезу і спроба віднайти більш регулярну функцію. В ході цього процесу відбувається пошук гіпотези, яка точно буде мінімізувати зважену суму емпіричних втрат та складність гіпотези. Це представляється наступною формулою **регуляризаційної функції**:

$$Cost(h) = L(h) + \lambda \cdot Complexity(h) \ , \qquad (11.19)$$

де λ — є гіперпараметром (додатне число) , яке слугую в якості коефіцієнта перерахунку між **втратами** та **складністю** гіпотези. Якщо це значення обрано правильно, тоді емпіричні втрати більш простої функції будуть добре **урівноважуватись** із тенденцією складної функції до перенавчання. Тобто цей параметр буде керувати тим, як сильно потрібно віддавати перевагу простішим функціям. Для лінійних функцій застосовується сімейство функцій регуляризації, в який **складність** розраховується як функція від **вагових коефіцієнтів.**

$$Complexity(h_w) = L_q(w) = \sum_i |w_i|^q \ , \qquad (11.20)$$

де при $q = 1$ має місце регуляризація L_1 — мінімізація суми **абсолютних** значень, а при $q = 2$ — регуляризація L_2 , що мінімізує суму **квадратів**. Обрання потрібної функції залежить від задачі, що вирішується. Для прикладу, регуляризація L_1 має корисну тенденцію до отримання **розріджених моделей** (*sparse model*). В даному випадку деяка множина вагових коефіцієнтів встановлюються в “нульове” значення, фактично оголошуючи певні атрибути x повністю **нерелевантними**. Гіпотези, в яких відкидаються багато атрибутів, зазвичай мають меншу схильність до перенавчання.

11.2.5 Лінійна класифікація із порогом

Окрім задач регресії, лінійні функції можна також використовувати для задач класифікації. Наприклад, маючи певні точки даних, які відповідають двом класам, можна знайти таку гіпотезу h, що буде приймати на вхід точки даних (x_1, x_2) та повертати значення “0” для об’єктів одного класу і “1” — для другого класу. В даному випадку знайдена лінія буде **границею рішення**, що буде розділяти об’єкти двох класів. Ця границі називається **лінійним роздільником**. Дані, які допускають таке розділення на класи, називаються **лінійно роздільними**. Дані над цією лінією відповідатимуть одному класу, під лінією — іншому (рис. 11.5). Для того, щоб отримувати відповідні значення “0” або “1”, можна застосовувати певну **порогову функцію** (*threshold*). Ця функція і буде повертати потрібне значення класу “1” в разі, якщо значення лінійної функції більше 0, і “0” — якщо менше. Щоб знайти потрібний вектор вагових коефіцієнтів, можна було б застосовувати формули, які вже розглядались, або застосувати градієнтний спуск. Але є одне обмеження, а саме те, що істинні значення y та значення шуканої гіпотези $h_w(x)$ можуть бути лише “0” або “1”. Це призводить до того, що неможливо застосувати жоден із розглянутих способів, оскільки градієнт у просторі вагових коефіцієнтів буде майже всюди нульовим, окрім точок де функція $h_w(x)$ дорівнює нулю. В цих точках градієнт буде невизначеним.

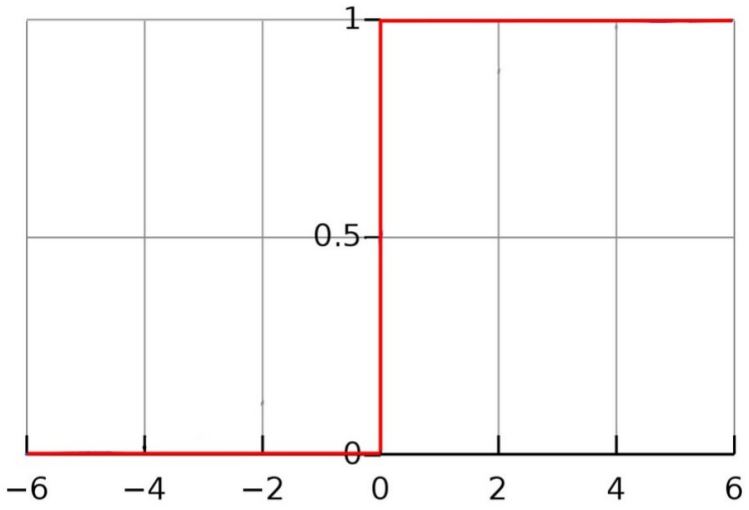


Рисунок 11.5 — Бінарна порогова функція

Для оновлення вагових коефіцієнтів в такому випадку є просте правило, яке буде збігатись у рішення — **лінійного роздільника**. В такому випадку формула оновлення вагових коефіцієнтів залишається такою самою, але виконуються три варіанти дій:

1. Якщо вихідне значення є правильним $y = h_w(x)$, тоді вагові коефіцієнти **не змінюються**;

2. Якщо істинне значення $y = 1$, а $h_w(x) = 0$, тоді вагові коефіцієнти w_i потрібно **збільшити**, якщо значення відповідного вхідного атрибуту x_i є **додатнім**, і **зменшити** — якщо **від'ємне**;

3. Якщо істинне значення $y = 0$, а $h_w(x) = 1$, тоді вагові коефіцієнти w_i потрібно **зменшити**, якщо значення відповідного вхідного атрибуту x_i є **додатнім**, і **збільшити** — якщо **від'ємне**;

Дане правило навчання називається правилом **навчання перцептрону**. При такому навчанні використовується по одному прикладу даних за один раз, обраного випадковим чином, аналогічно до **стохастичного градієнтного спуску**.

11.2.6 Лінійна класифікація із логістичною регресією

Напротивагу попередньому способу є інший спосіб створити лінійний класифікатор, який буде мати **кращі прогнози** для значень близьких до границі рішень. Головною проблемою попереднього прикладу є обмежені значення — лише “0” або “1”. Цю проблему можна значною мірою усунути шляхом використання більш **м’якої порогової функції** — тобто застосувати апроксимацію жорсткого порогу неперервною диференційованою функцією. Для цієї задачі часто застосовується **логістична функція** також відома як сигмоїда (*sigmoid*):

$$h_w(x) = \frac{1}{1 + e^{-w \cdot x}} \tag{11.21}$$

Тепер вихідні значення лежать в діапазоні [0, 1] і їх можна інтерпретувати як **ймовірність** приналежності певної точки до класу “1”. Така гіпотеза буде формувати більш м’які границю у просторі вхідних значень і видавати ймовірність “0,5” для будь-яких вхідних даних, які лежатимуть на граничній області. Інші значення будуть поступово наближатись до “0” або “1” по мірі віддалення від границі.

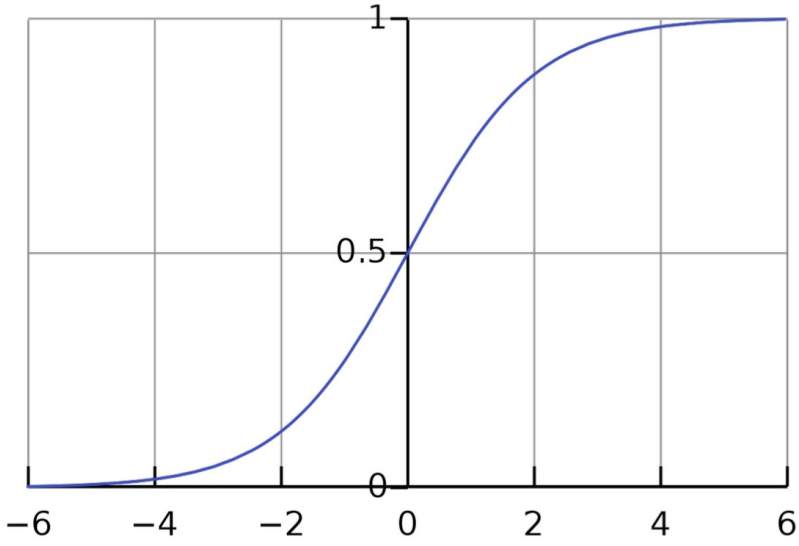


Рисунок 11.6 — Логістична порогова функція (сигмоїда)

11.2.7 Логістична регресія

Підбір вагових коефіцієнтів для моделі з метою мінімізації втрат в наборі даних в такому випадку називатиметься **логістичною регресією**. Процедура оновлення вагових коефіцієнтів є аналогічною до лінійної регресії із застосуванням градієнтного спуску, за винятком іншої формули похідної для сигмоїди. В загальному вигляді оновлення вагових коефіцієнтів відбуватиметься за наступною формулою:

$$w_i \leftarrow w_i + \alpha(y - h_w(x)) \times h_w(x)(1 - h_w(x)) \times x_i \qquad (11.22)$$

В цілому логістична регресія має повільнішу збіжність, але веде себе більш передбачувано. На зашумлених даних логістична регресія має кращу збіжність. Ці переваги, як правило, справедливі для реальних задач, тому логістична регресія стала одним із найбільш популярних методів **класифікації** для задач в різних проблемних областях.

11.3 Ансамблевий метод машинного навчання

Розглянуті методи машинного навчання використовували єдину гіпотезу для отримання прогнозів. Ідея ансамблевого навчання полягає у виборі деякого **набору** або ансамблю **гіпотез** h_1, h_2, ..., h_n і комбінуванні прогнозів шляхом **усереднення**, **голосування** або **іншого рівня** машинного навчання. В такому випадку окремі гіпотези називаються **базовими моделями** (*base models*), а їх комбінація — **ансамблем моделей** (*ensemble model*).

Застосовувати подібний підхід можна по декільком причинам:

1. По-перше, для того, щоб **зменшити зміщення**. Простір гіпотез базової моделі може бути занадто обмеженим, що призведе до сильного зміщення і поганого узагальнення вхідних даних. Ансамбль може виявитись більш виразним і в результаті матиме менше зміщення, а ніж базова модель. Наприклад, **трьома лінійними класифікаторами** можна успішно представити

трикутну область, яку не можливо представити одним лінійним класифікатором.

2. По-друге, застосування ансамблю дає можливість **зменшити дисперсію**. Для прикладу можна розглянути ансамбль із п’яти бінарних класифікаторів, результати яких комбінуються методом більшості голосів. Щоб ансамбль невірно класифікував певний новий приклад, щонайменше “три” з “п’яти” класифікаторів мають невірно його класифікувати. Ця ситуація має меншу ймовірність, а ніж невірна класифікація одного єдиного класифікатору.

Створюються подібні ансамблі декількома способами:

1. **Беггінг (*bootstrap aggregating*).** Подібний спосіб полягає у **генерації *K* навчальних наборів** шляхом випадкової вибірки *N* прикладів із початкового набору. При чому обраний приклад залишається у початковому наборі і може повторно обиратись. Далі виконується запуск певного алгоритму машинного навчання на *K* згенерованих наборах для отримання *K* різних гіпотез. В подальшому прогнози цих гіпотез об’єднуються у остаточний результат.

2. **Стекінг (*stacked generalization*).** При даному способі об’єднуються декілька базових моделей різних класів, які навчаються на одних і тих самих даних. Прогнози цих окремих базових моделей додаються до перевірочного набору. Далі цей перевірочний набір використовується для навчання нової ансамблевої моделі, яка використовує прогнози від попередніх базових моделей для формування фінального прогнозу.

3. **Бустінг (*boosting*)**. Найбільш популярний ансамблевий метод при якому застосовуються **зважені навчальні набори**. Під зваженим розуміється те, що приклади, які були класифіковані невірно отримують більший ваговий коефіцієнт, що призводить до частішого використання даного прикладу при подальшому навчані. В цілому ідея методу полягає у тому, що наступні базові моделі, які створюються, мають краще працювати на прикладах, які невірно класифікували попередні моделі. Тому наступні навчальні дані будуть містити більше прикладів, які складніше класифікувати.

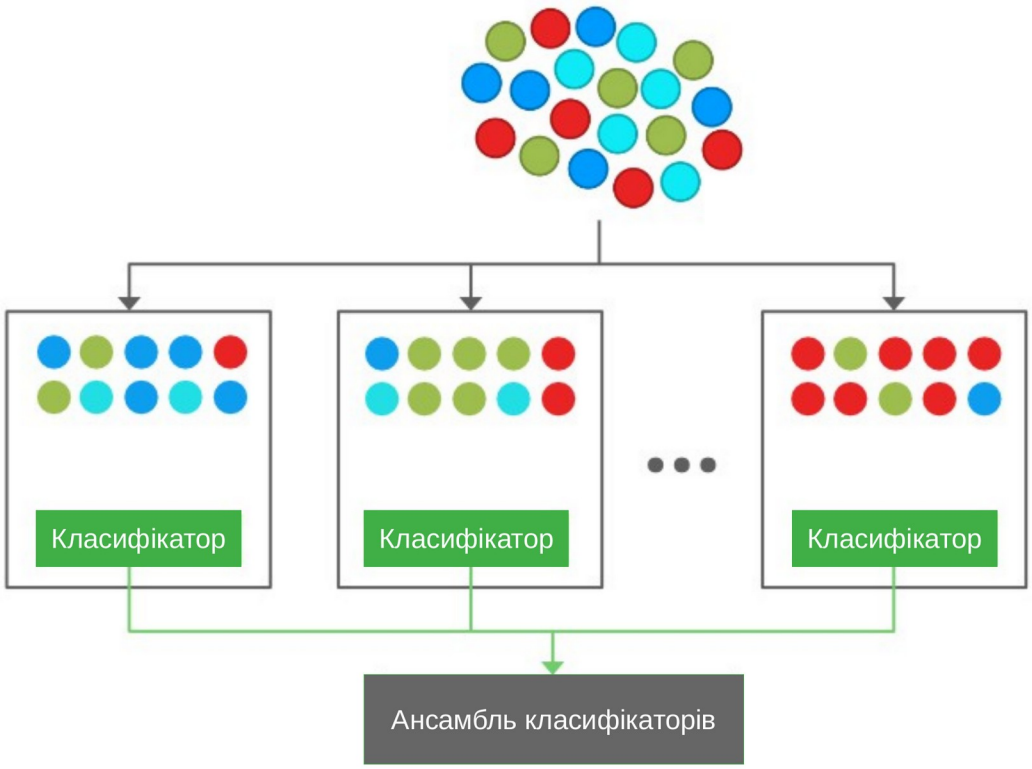


Рисунок 11.7 — Метод беггінгу для створення ансамблевої моделі

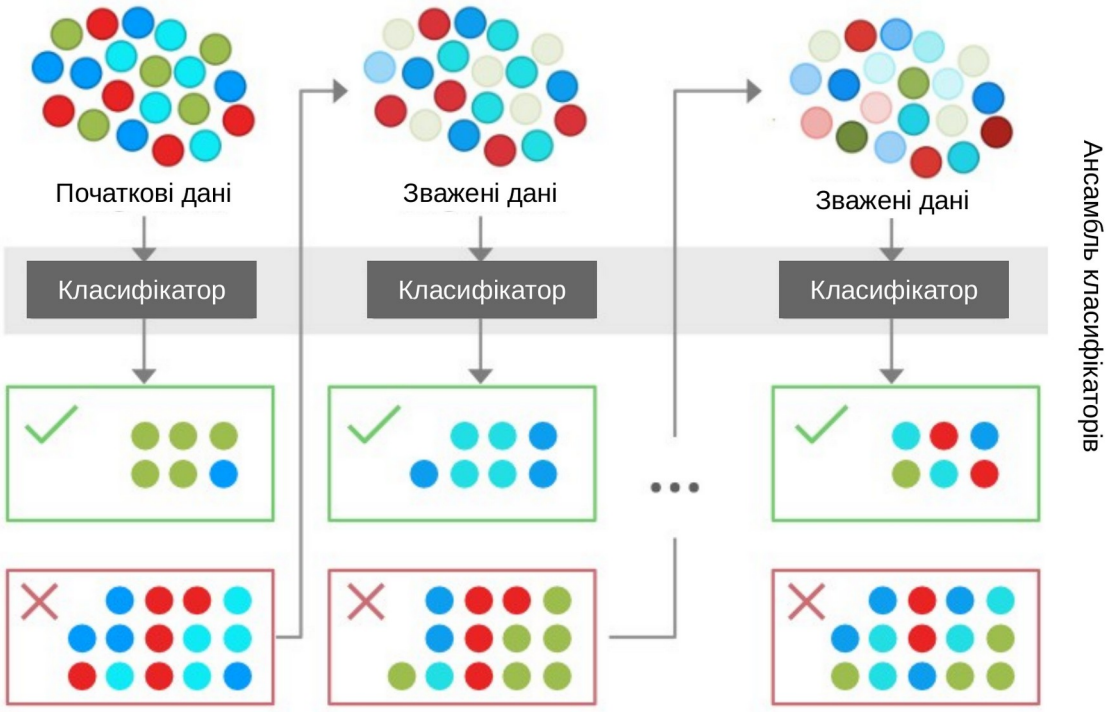


Рисунок 11.8 — Метод бустінгу для створення ансамблевої моделі

11.4 Дані для навчання

В цілому якість навчання будь-якого алгоритму навчання, що використовує навчальні набори залежить від **якості** та **обсягу** прикладів. Найбільш складною задачею в цьому є створення достатньо великих наборів даних для навчання. Незважаючи, що наразі є велика кількість загальнодоступних навчальних наборів даних, таких як: *ImageNet*, *CIFAR*, *MNIST* та інші вузькоспеціалізовані набори, можлива ситуація, коли потрібно буде застосовувати **власні дані** для навчання. Достатньо ймовірною ситуацією буде їх невеликий обсяг. Для того, щоб подолати дану проблему можна застосовувати метод **доповнення даних** (*data augmentation*). Наприклад, для набору даних зображень можна створювати декілька різних версій кожного зображення за рахунок їх обертання, перетворення, кадрування, масштабування, зміни яскравості, балансу кольору, додавання шуму, тощо. Як можна побачити, застосувавши лише ці наведені способи можна щонайменше у 8 разів **збільшити** обсяг навчального набору. Слід відмітити, що мітка модифікованого зображення при цьому залишається незмінною.

Також може виникати ситуація **незбалансованості класів**. Це ситуація, коли у навчальному наборі переважають приклади певного класу. Подібна ситуація призводитиме до того, що навчана модель буде погано узагальнювати дані і для більшості нових вхідних даних видаватиме прогноз, який відповідає мітці, що переважала. Уникнути цієї проблеми можна виконуючи **зменшувальну вибірку** (*undersampling*) — зменшуючи кількість прикладів із переважаючого класу до кількості прикладів класу меншого за обсягом. Також можна використовувати **збільшуючу вибірку** (*over-sampling*) — дублюючи приклади для класу із меншою кількістю даних.

ТЕМА 12
ГЛИБОКЕ НАВЧАННЯ

Термін глибоке навчання охоплює широкий клас методів машинного навчання, в яких гіпотези приймають форму **складних алгебраїчних схем** із регульованою силою зв’язків. Слово “глибокий” пов’язане із тим фактом, що ці схеми, як правило, організовані у вигляді **декількох шарів**. Це означає, що шлях виконання обчислень від входів до виходів включає в себе багато проміжних **етапів**. На теперішній час глибоке навчання є підходом, який широко застосовується в **застосунках** візуального розпізнавання об’єктів, машинного перекладу, розпізнавання природної мови, синтезу мови та зображень. Також цей підхід відграє важливу роль у застосунках, що включають навчання з підкріпленням.

Глибоке навчання бере свій початок у ранніх роботах 1943 р., в яких Уоррен Мак-Каллок та Уолтер Піттс описували спробу змоделювати мережу нейронів людського мозку за допомогою обчислювальних схем. З цієї причини мережі, які навчаються методами глибокого навчання, часто називають **нейронними мережами**, хоча їх **подібність** до реальних клітин мозку — нейронів і їх структур є доволі **поверхневою**.

Глибоке навчання має значні **переваги** порівняно із деякими методами машинного навчання. Особливо це помітно для **багатомірних даних**, таких як зображення. Наприклад, хоча такі методи, як **лінійна та логістична регресія**, дозволяють опрацьовувати велику кількість вхідних змінних, **обчислень**, що виконуються на шляху від входу до виходу, є **доволі мало**. До цих обчислень можна віднести множення на єдиний ваговий коефіцієнт і додавання результату у сукупний вихід. Більше того, різні вхідні змінні вносять незалежний вклад у вихідне значення і не взаємодіють одна з одною (рис. 12.1, лівий). Цей факт суттєво обмежує потужність таких моделей виражати складні об’єкти та закономірності. Вони можуть представляти лише лінійні функцій та границі у вхідному просторі, в той час як більшість концепцій у реальному

навколишньому середовищі є набагато складніші. З іншої сторони, **дерева рішень** допускають довгі шляхи обчислень, які можуть залежати від багатьох вхідних змінних, але тільки в межах відносно невеликого діапазону можливих векторів вхідних значень (рис. 12.1, середній). Якщо подібне дерево рішень буде мати довгі шляхи для значного діапазону вхідних даних, тоді воно безумовно буде експоненційно великим відносно кількості вхідних змінних. Основна ідея **глибокого навчання** полягає у навчанні схем таких чином, щоб шляхи обчислень були довгими, що дозволяє всім вхідним змінним складним чином взаємодіяти між собою (рис. 12.1, правий). Подібні моделі виявляються достатньо потужними, щоб мати можливість відображати складні реальні дані для багатьох проблемних областей.

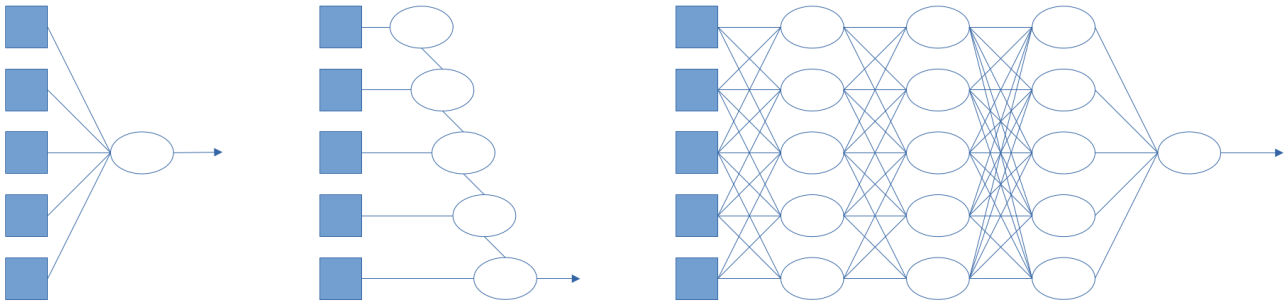


Рисунок 12.1 — Різновиди взаємозв’язків змінних/вершин

12.1 Прості мережі із прямим зв’язком

Мережа із прямим зв’язком має з’єднання, які працюють тільки в одному напрямі. Подібна мережа утворює **орієнтований ациклічний граф** із вказаними вхідними та вихідними вершинами. В кожній вершині обчислюється деяка функція від вхідних значень цієї вершини і отриманий результат передається у дочірню вершину в цій мережі. Інформація проходить через мережу від вхідних вершин до вихідних, при цьому **петлі** та **цикли** відсутні. Напротивагу, існує інший тип мереж — **рекурентні мережі**, в яких проміжні дані із внутрішніх вершин або шарів передаються у зворотному напрямі до

попередніх вершин/шарів. В такому випадку подібні **зворотні зв’язки** утворюють динамічну систему, яка має внутрішні стани або пам’ять.

Логічні схеми, що реалізують булеві функції, є прикладом мереж із прямим зв’язком. В логічних схемах значення вхідних сигналів обмежені двома значеннями “0” та “1”. Кожна вершина, в такому випадку, реалізує просту логічну функцію відносно значень від своїх вхідних вершин і на вихід видає або “0”, або “1”.

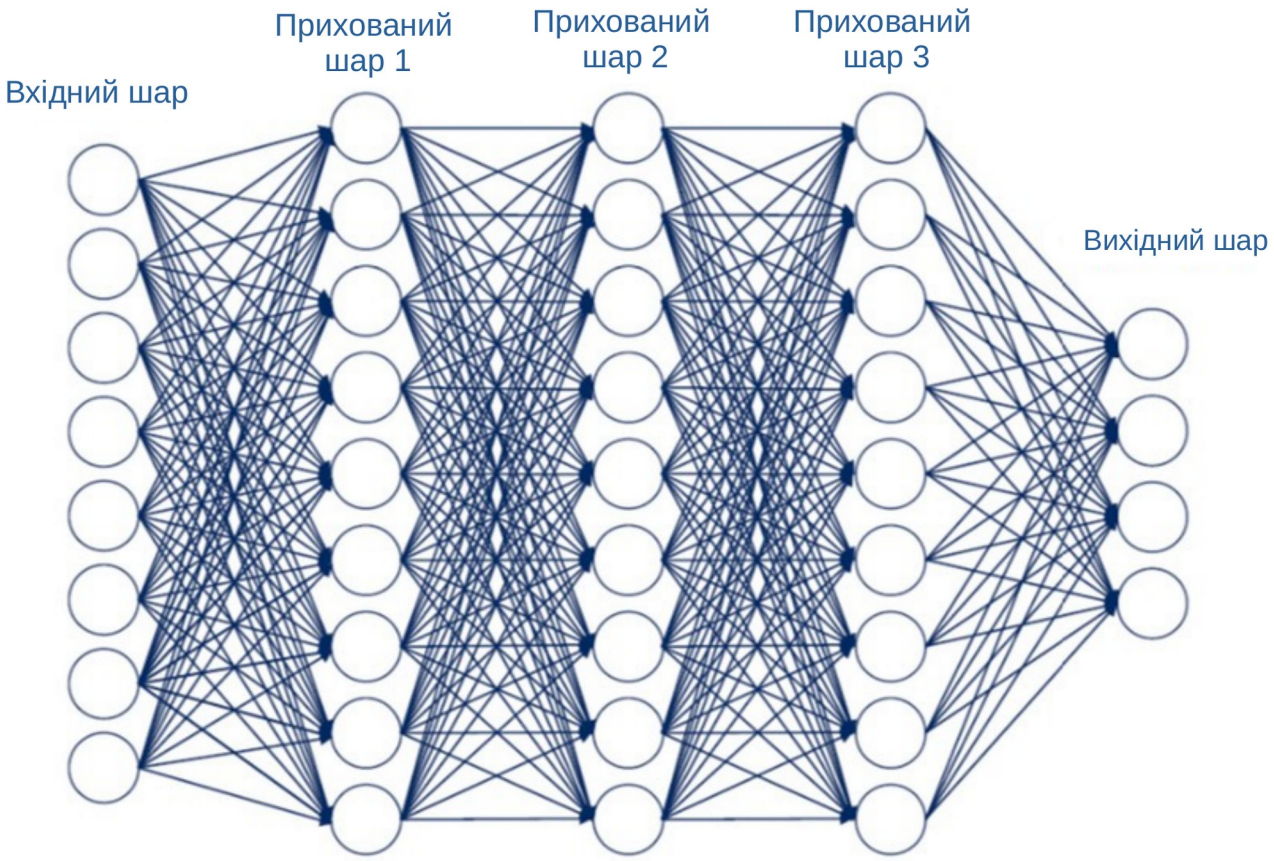


Рисунок 12.2 — Приклад простої мережі із прямим зв’язком

В **нейронних мережах** вхідні значення, як правило, представляються **неперервними**. Вершини приймають їх на вхід і передають на вихід також неперервні значення. Частина із вхідних значень вершин є параметрами мережі, які **навчаються**, щоб налаштувати значення цих параметрів таким чином, щоб вся мережа в цілому відповідала даним **навчального набору**.

12.2 Мережа як складна функція

Кожна **вершина** в мережі називається **елементом** (*unit*). Елемент спочатку розраховує зважену суму вхідних значень від попередніх вершин, а потім до отриманого значення застосовується деяка **нелінійна функція** для отримання вихідного результату. Це можна представити у наступній формулі:

$$a_j = g_j \sum_i w_{i,j} \cdot a_i \qquad (12.1)$$

де a_j позначає вихід елемента j; $w_{i,j}$ являються ваговим коефіцієнтом для зв'язку від елемента i до елемента j; g_j — нелінійна **функція активації**, пов'язана із елементом j.

Той факт, що функція активації є нелінійною являється важливим фактором, оскільки це дозволяє достатньо великим мережам елементів представляти довільні функції, не обмежуючись лише лінійними функціями. Широко застосовуються множина із різних функції активації в залежності від класу задачі. Найбільш розповсюдженими з них є:

1. Логістична функція — **сигмоїда**, яка також застосовується у логістичній регресії.

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad (12.2)$$

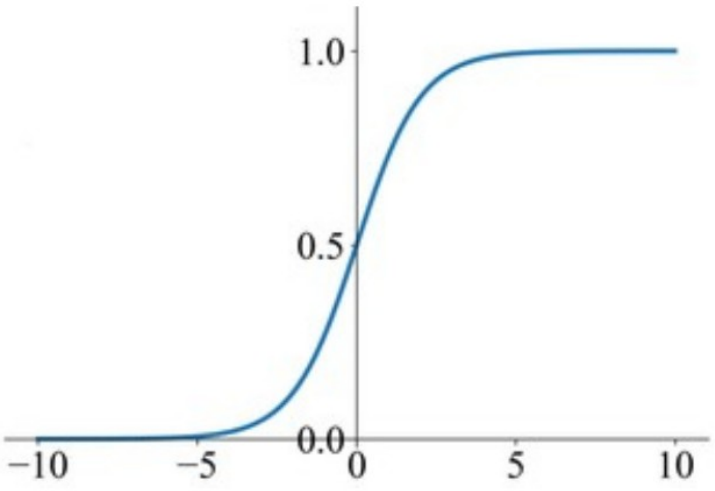


Рисунок 12.3 — Функція активації сигмоїда

2. Функція **ReLU** — назва якої є абревіатурою від випрямленого лінійного елемента (*rectified linear unit*).

$$ReLU(x)=max(0,x) \tag{12.3}$$

3. Функція **softplus —** згладжена версія *ReLU*.

$$softplus(x)=\log(1+e^{x}) \tag{12.4}$$

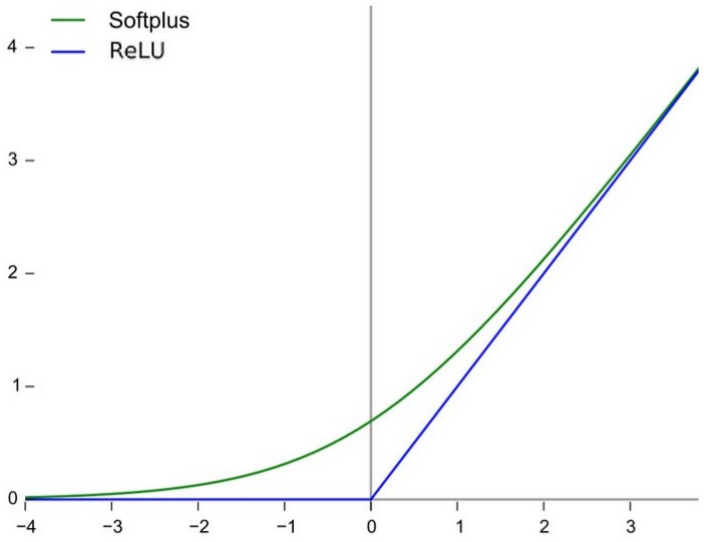


Рисунок 12.4 — Функції активації *ReLU* та *Sofplus*

4. Функція гіперболічного тангенсу — **tanh**. Діапазон значень цієї функції лежить в межах [-1, 1] і є зменшеною та зміщеною версією сигмоїди.

$$\tanh(x)=\frac{e^{2x}-1}{e^{2x}+1} \tag{12.5}$$

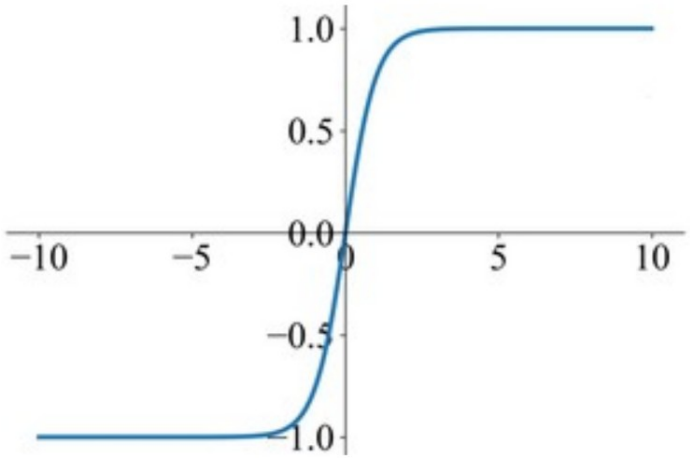


Рисунок 12.5 — Функція активації *Tanh*

Об’єднання декількох елементів в мережу створює складну функцію, яку можна розглядати як **композицію** алгебраїчних виразів, представлених окремими елементами. Наприклад, приведена мережа (рис. 12.6) представляє деяку функцію $h_w(x)$, параметризовану ваговими коефіцієнтами w , яка відображає вхідний вектор x з двома елементами в скалярне вихідне значення $\hat{y}$.

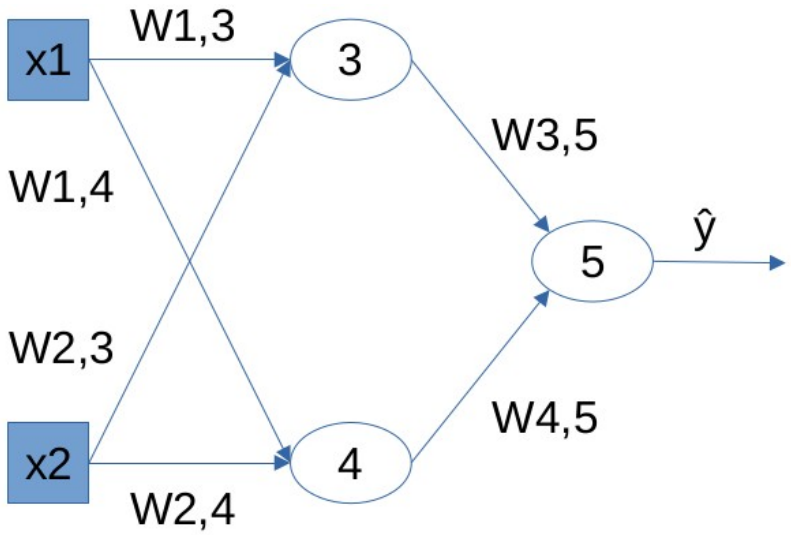


Рисунок 12.6 — Приклад простої мережі для невідомої функції

Внутрішня структура цієї функції відображає структуру мережі. В даному випадку її можна записати у вигляді наступного виразу:

$$\hat{y} = g_5(w_{0,5} + w_{3,5} a_3 + w_{4,5} a_4) = g_5(w_{0,5} + w_{3,5} g_3(w_{0,3} + w_{1,3} x_1 + w_{2,3} x_2) + w_{4,5} g_4(w_{0,4} + w_{1,4} x_1 + w 2{,}4\, x_2)) \quad (12.6)$$

Таким чином отримаємо вихідне значення $\hat{y}$, виражене як функція $h_w(x)$ від вхідних значень та вагових коефіцієнтів.

Більш розгорнутим представленням мережі є граф обчислень або **граф потоку даних**. Він являє собою схему, в якій кожна вершина представляє елементарні обчислення.

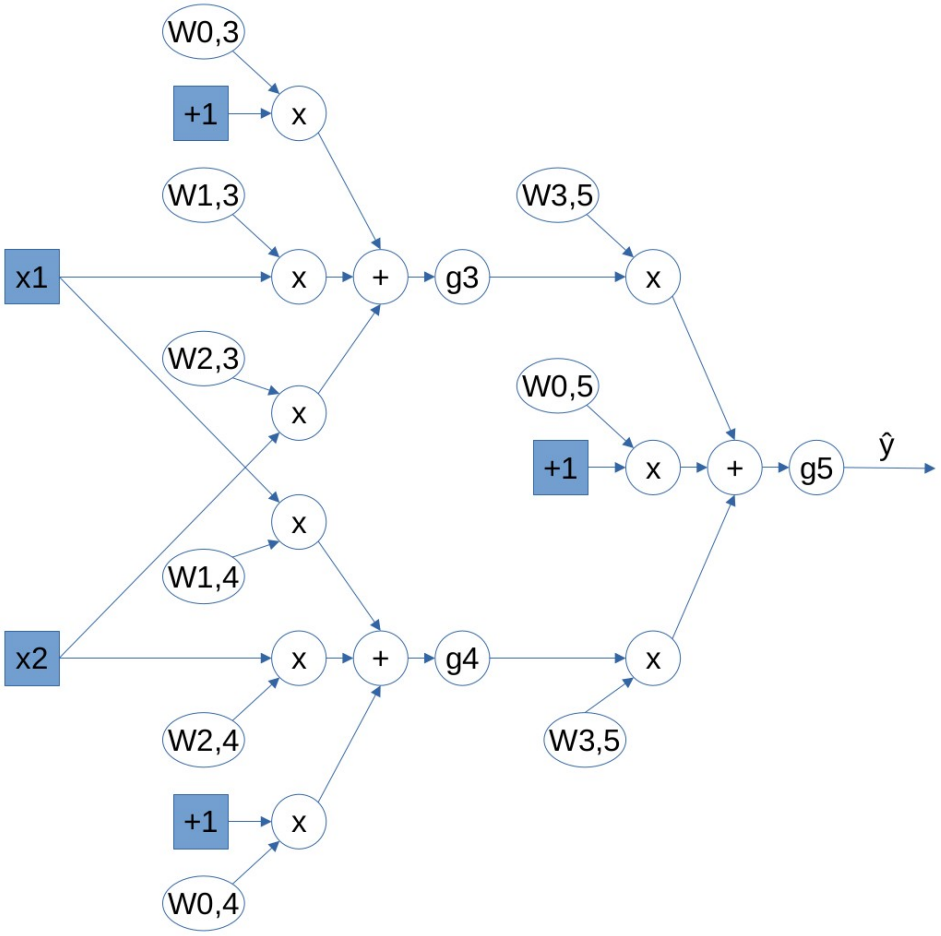


Рисунок 12.7 — Приклад графу потоку даних

Тут можна побачити як кожен ваговий коефіцієнт впливає на результуюче значення у відповідних вершинах. Ці вагові **коефіцієнти підбираються** в ході навчання таким чином, щоб вихідний сигнал $\hat{y}$ більш точно відповідав істинному значенню y в навчальному наборі. Кожен ваговий коефіцієнт визначає в якій мірі наступна вершина в графі буде враховувати певне значення від попередньої вершини.

В цілому обчислення в нейронних мережах представляються **матричними операціями**. Для позначення матриці **вагових коефіцієнтів** використовують позначення W та верхній індекс, що позначає до якого шару відносяться відповідні значення. Наприклад, $W^{(1)}$ позначає вагові коефіцієнти для першого шару. Аналогічно, $g^{(1)}$ та $g^{(2)}$ позначають **функції активації** в першому та другому шарах. Враховуючи зазначені позначення можна записати всю мережу у вигляді наступного рівняння:

$$h_w(x) = g^{(2)}(W^{(2)} g^{(1)}(W^{(1)} x)) \quad (12.7)$$

Даний запис повністю відповідає показаному графу обчислень, але тут все показано у вигляді ланцюжків матриць вагових коефіцієнтів, які подаються на кожному шарі, та матриць із значеннями функцій активації.

Слід відмітити, що мережа, представлена даним графом, є **повно зв'язною.** Це означає, що кожна вершина в кожному шарі мережі пов’язана із кожною вершиною у наступному шарі. В цілому **зв’язність** є важливої характеристикою нейронної мережі, яка впливає на ефективність навчання та продуктивність роботи.

12.3 Градієнти та навчання

Минулої теми вже розглядався підхід до **навчання із вчителем**, який засновувався на методі **градієнтного спуску,** коли розраховувався градієнт функції втрат відносно вагових коефіцієнтів і потім значення вагових коефіцієнтів змінювалось згідно цього градієнту, щоб **зменшити втрати**. Такий самий підхід можна застосовувати і для навчання вагових коефіцієнтів у графах обчислень, які представляють певну мережу. В цілому процес обчислень градієнтів буде аналогічним для **вихідних шарів**, які формують вихідне значення мережі, і трохи відрізнятись для **прихованих шарів**, які напряму не пов’язані із вихідними шарами. Для розрахунків втрат у нейронних мережах також часто застосовують **квадратичну функцію втрат** L_2 .

Як приклад можна розглянути розрахунок деяких градієнтів в прикладі мережі, показаному раніше. В розрахунках застосовується **ланцюгове правило:**

$$\frac{\partial}{\partial x} g(f(x)) = g'(f(x)) \frac{\partial}{\partial x} f(x) \quad (12.8)$$

Для **вагового коефіцієнту** $w_{3,5}$ формула градієнту буде мати наступний вигляд:

$$\frac{\partial}{\partial w_{3,5}} L_2(h_w) = \frac{\partial}{\partial w_{3,5}} (y - \hat{y})^2 = -2(y - \hat{y}) \frac{\partial \hat{y}}{\partial w_{3,5}} = -2(y - \hat{y}) \frac{\partial}{\partial w_{3,5}} g_5(\text{in}_5) = \quad (12.9)$$

$$= -2(y - \hat{y}) g'_5(\text{in}_5) \frac{\partial}{\partial w_{3,5}} \text{in}_5 = -2(y - \hat{y}) g'_5(\text{in}_5) \frac{\partial}{\partial w_{3,5}} (w_{0,5} + w_{3,5} a_3 + w_{4,5} a_4) = -2(y - \hat{y}) g'_5(\text{in}_5) a_3$$

Трохи складнішими будуть розрахунки для **коефіцієнту $w_{1,3}$** , де формула градієнту буде мати наступний вигляд:

$$\frac{\partial}{\partial w_{1,3}} L_2(h_w) = -2(y-\hat{y}) g'_5(\text{in}_5) \frac{\partial}{\partial w_{1,3}} (w_{0,5} + w_{3,5} a_3 + w_{4,5} a_4) = -2(y-\hat{y}) g'_5(\text{in}_5) w_{3,5} \frac{\partial}{\partial w_{1,3}} a_3 = \ldots$$

$$\ldots = -2(y-\hat{y}) g'_5(\text{in}_5) w_{3,5} g'_3(\text{in}_3) \frac{\partial}{\partial w_{1,3}} (w_{0,3} + w_{1,3} x_1 + w_{2,3} x_2) = -2(y-\hat{y}) g'_5(\text{in}_5) w_{3,5} g'_3(\text{in}_3) x_1$$

(12.10)

Якщо визначити **“помилку сприйняття”** у точці, де елемент 5 отримує свій вхід у даному вигляді $\Delta_5 = 2(\hat{y}-y) g'_5(\text{in}_5)$, тоді отримаємо градієнт відносно $w_{3,5}$ як $\Delta_5 a_3$. Якщо аналогічно визначити $\Delta_3 = \Delta_5 w_{3,5} g'_3(\text{in}_3)$, тоді градієнт відносно $w_{1,3}$ буде просто $\Delta_3 x_1$. З цього слідує, що “помилка сприйняття” на вході елемента “3” — це “помилка сприйняття” на вході елементу “5”, помножена згідно інформації вздовж шляху від елемента “5” до елемента “3”. Подібне явище носить загальний характер і називається **зворотнім розповсюдженням помилки** (*backpropagation*), при якому помилка на виході передається по мережі у зворотному напрямі для оновлення значень вагових коефіцієнтів.

Іншою важливою характеристикою цих виразів градієнтів є те, що вони включають локальні похідні $g`$. Ці похідні завжди додатні, але їх значення можуть бути близькими до нуля, у випадку функцій активації сигмоїди, гіперболічного тангенсу, або рівними нулю у випадку *ReLU*. Це може статись в разі, якщо для вхідних даних певний *j* елемент мережі потрапляє у плоску робочу область. Якщо відповідна похідна буде малою або рівною нулю, то це призведе до того, що зміна вагових коефіцієнтів, які ведуть до цього елементу, незначним чином вплине на вихідне значення цього елементу. В результаті глибокі мережі із великою кількістю шарів можуть мати проблему **зникаючих градієнтів** (*vanishing gradient*), коли сигнали про помилку повністю зникають при зворотному розповсюджені по мережі.

12.4 Автоматичне диференціювання

Вище було показано приклад розрахунків градієнтів для прикладу дуже невеликої мережі. Для такої мережі ці розрахунки були виражені за допомогою нескладних формул, які дозволяли розрахувати значення градієнтів для вагових коефіцієнтів, передаючи інформацію від вихідних елементів в зворотному напрямі. Подібні операції будуть справедливі і для будь-якого іншого графа мережі із прямим зв’язком. Зазначені формули можуть виглядати складно для мереж більшого розміру, і як виявляється не має потреби повторювати всі приведені у рівняннях дії для кожної нової мережевої структури. Всі ці градієнти можуть бути розраховані методом **автоматичного диференціювання,** коли дані правила розрахунків систематично застосовуються для розрахунків градієнтів в будь-якій мережі. Насправді метод зворотного розповсюдження помилки в **глибокому навчанні** — це просте застосування зворотного режиму диференціювання. Всі сучасні **програмні бібліотеки** для глибокого навчання надають засоби автоматичного диференціювання, тому користувач має можливість вільно експериментувати із різноманітними структурами нейронних мереж, функціями активації, функціями втрат і формами композиції, не переймаючись за необхідність виконувати безліч розрахунків, щоб вивести алгоритм навчання, тобто знайти градієнти, для чергової експериментальної структури мережі. Всі ці можливості дають змогу зосередитись на розробці нейронної мережі під певну поставлену задачу і в цілому подібний підхід називається **наскрізним навчанням** (*end-to-end learning*). Застосовуючи даний підхід складна обчислювальна система, наприклад для машинного перекладу, може бути побудована із декількох підсистем, які будуть навчатись. Далі ці системи навчаються наскрізним чином на навчальних парах “вхід-вихід”. При такому підході розробнику необхідна лише **концептуальна ідея** про те, як в цілому має працювати та бути структурована система, і не має необхідності завчасно знати, що повинна виконувати кожна підсистема або як маркувати її входи та виходи.

12.5 Обчислення для глибокого навчанням

Наразі вже було розглянуто основні ідеї глибокого навчання, а саме представлення гіпотези у вигляді графу обчислень із можливістю змінювати значення вагових коефіцієнтів за допомогою розрахунків градієнтів функції втрат відносно цих вагових коефіцієнтів з метою досягнення відповідності вихідних сигналів із істинними значеннями навчального набору. Тепер давайте розглянемо яким чином **закодувати** вхідні та вихідні дані *x* та *y* із навчального набору у множину значень вхідних та вихідних елементів мережі.

12.5.1 Вхідне кодування

Вхідні та вихідні вершини графу обчислень є фактично вершинами, які прямо пов’язані із вхідними та вихідними даними навчального набору. Кодування вхідних даних, як правило, є простою задачею у випадку із **розгорнутим представленням**, коли кожний навчальний приклад містить *n* значень атрибутів вхідних даних.

Якщо атрибути є **логічними**, тоді буде *n* вхідних вершин, які зазвичай матимуть значення “0” або “1” для відповідних значень “*False*” або “*True*”.

Іншим варіантом є **числові атрибути**, значення яких можуть бути цілочисельними або дійсними. Їх значення можуть записуватись в абсолютному вигляді, або масштабуватись чи **нормалізуватись** для приведення до певного діапазону значень.

Зображення, як тип даних, не зовсім відповідають розгорнутому представленню даних, хоча *RGB*-зображення розміром *X* на *У* пікселів доволі зручно розглядати як масив із 3*ХУ* цілочисельних атрибутів, зазвичай зі значеннями в діапазоні [0, ..., 255]. Але розглядаючи дані у вигляді зображень, слід враховувати, що кожна “трійка” атрибутів представляє **окремий піксель**. Також особливо важливим фактом є присутність **взаємозв’язку** між суміжними пікселями, яка в певній мірі також має своє значення. Можна представляти цей

“взаємозв’язок” у вигляді атрибутів, які знаходяться поруч, але в цілому цей зв’язок пікселів буде повністю втрачатись, в разі використання повно зв’язних шарів в мережі. На практиці нейронні мережі, які працюють із даними у вигляді зображень, мають структуру подібну до масивів, що дозволяє відображати семантику суміжності пікселів.

Атрибути, що відображають **категорії** або класи із більш ніж двома значеннями, зазвичай кодуються у вигляді **бінарного вектору** (*one-hot vector*) із кількістю елементів, що відповідає загальній кількості категорій. В ньому кожне значення “1” або “0” показує приналежність до відповідного класу. Можна було б кодувати категорію у вигляді єдиного числа у діапазоні [0, k], але оскільки мережа представляє сукупність неперервних функцій, вона безумовно враховувала би суміжність певних числових значень для різних категорій, наприклад “3” та “4”, але для даної задачі ця суміжність чисельних значень не має ніякого семантичного сенсу у приналежності певного прикладу до однієї чи іншої категорії.

12.5.2 Вихідні шари та функції втрат

На вихідній стороні мережі проблема кодування значень, які видає мережа, у реальні значення із навчального набору в більшій мірі подібна до вхідного кодування. Наприклад, якщо мережа повинна визначати приналежність об’єкту до певної категорії із чотирьох можливих, тоді можна застосовувати кодування категорій у вектор із **чотирма бітами**. Але такий спосіб кодування не підійде у випадку наявності вектору значень вихідних даних y, оскільки його потрібно певним чином порівнювати із значенням прогнозу $\hat{y}$, який видає мережа, оцінювати втрати та адаптувати вагові коефіцієнти.

До цього вже застосовувалась **квадратична функція втрат,** яка спрощувала обчислення. В більшості застосунків глибокого навчання вихідне значення мережі частіше інтерпретується як **ймовірність** і в якості функції

втрат використовується **від'ємна логарифмічна правдоподібність.** При навчанні із застосуванням даної функції буде шукатись вектор вагових коефіцієнтів *w,* який мінімізуватиме суму від'ємних логарифмічних правдоподібностей для *N* прикладів:

$$w^* = \underset{w}{argmin} - \sum_{j=1}^{N} \log P_w(y_j | x_j) \qquad (12.11)$$

В літературі по глибокому навчанню подібна мінімізація називається **мінімізація перехресної ентропії** (*cross entropy loss*). Щоб мінімізувати від'ємну логарифмічну ймовірність необхідно мати можливість інтерпретувати вихідні дані як ймовірності. Наприклад, якщо мережа має один вихідний елемент із функцією активації типу **сигмоїда** і мережа навчалась для **булевої класифікації**, тоді можна інтерпретувати вихідне значення у вигляді ймовірності приналежності прикладу до класу.

У випадку багатокласової класифікації, коли програма класифікатор розпізнає сотні категорій об'єктів, потрібна мережа із вихідним шаром у вигляді **категоріального розподілу**. Тобто, щоб дати *d* можливих відповідей, потрібно *d* вихідних вершин, що представлятимуть ймовірність із загальною сумою в "1". Щоб досягти цього, зазвичай застосовують шар під назвою ***softmax***, який видає вектор із *d* значень при заданому векторі вхідних значень "*in*" для цього шару. Кожен елемент вихідного вектору розраховується за формулою:

$$softmax(\text{in})_k = \frac{e^{\text{in}_k}}{\sum_{k'}^{d} e^{\text{in}_{k'}}} \qquad (12.12)$$

Прикладом вихідного результату для подібного шару може бути вектор (0,946; 0,047; 0,006; 0,001), де більшу ймовірність має елемент, що відповідає приналежності до 1-го класу.

Також можуть застосовуватись інші типи вихідних шарів таких, як **лінійні вихідні шари** (лінійної регресії), **шари змішаної щільності**, тощо.

12.5.3 Приховані шари

В процесі навчання нейронній мережі передаються багато вхідних значень *x* та відповідних вихідних значень *y*. При обробці вхідного вектору нейронна мережа виконує декілька **проміжних обчислень**, перед тим як отримати вихідне значення. Обчислення значень на кожному проміжному шарі мережі можна розглядати як різні представлення для вхідного вектору даних. Кожен шар мережі перетворює створене на попередньому шарі **представлення** з метою створення нового представлення. Поєднання всіх цих представлень дозволяє успішно перетворювати вхідне значення у вихідне. Одна гіпотеза відносно того, чому глибоке навчання добре працює, полягає у тому, що складні наскрізні перетворення, що відображають вхідні дані на вихідні, наприклад для певного зображення, де-компонуються множиною шарів мережі у структуру із багатьох простих перетворень, кожне з яких доволі легко можна навчити шляхом процедури локального оновлення.

Рисунок 12.8 — Ілюстрація мережі із прихованими шарами

В процесі формування даних внутрішніх проміжних перетворень глибокі нейронні мережі часто виявляють значущі представлення для даних, які в подальшому дозволяють краще відображати вхідні та вихідні дані навчального набору. Наприклад, мережа, яка навчається розпізнавати складні об’єкти на зображеннях, може сформувати внутрішні шари, що будуть виявляти **корисні**

ознаки (*features*) на зображеннях. Цими ознаками можуть бути певні краї, кути, прості фігури, риси обличчя, очі, та інші ознаки які притаманні зображенням у конкретній проблемній області. Але також глибокі нейронні мережі здатні формувати набагато складніші представлення у внутрішніх шарах, значення яких будуть абсолютно неочевидні для людини, хоча результат роботи мережі буде коректним.

Приховані шари нейронних мереж зазвичай менш різноманітні, порівнюючи із вхідними та вихідними. Протягом перших 25 років досліджень (до 2010) в області **багатошарових нейронних мереж** для внутрішніх вершин шарів мережі широко застосовували функції активації сигмоїду та гіперболічного тангенсу. Пізніше, після 2010, більш популярними стали функції активації *ReLU* та *softplus.* Однією із причин цього стало припущення, що вони дозволяють уникнути проблем із **зникаючими градієнтами**.

ТЕМА 13
СПЕЦІАЛІЗОВАНІ НЕЙРОННІ МЕРЕЖІ

13.1 Згорткові нейронні мережі

Згорткові мережі (*convolutional neural networks — CNN*) окремий вид нейронних мереж, які застосовуються для обробки даних, що представляють **зображення**. Попередньої теми вже згадувався той факт, що зображення не слід розглядати як простий вектор вхідних значень пікселів. Це справедливо тому, що інформація про **суміжність** пікселів тут відіграє ключове значення. Якщо будувати звичайну мережу із **повно зв'язними шарами** та обробляти за її допомогою вхідні дані у вигляді зображень, тоді обсяги прихованих шарів можуть бути доволі **великими**. Наприклад, візьмемо зображення з n пікселями і 1-й прихований шар із n елементів для якого ці пікселі зображення є вхідними даними. Оскільки вхідний та 1-й прихований шари є повно зв'язними, це означає, що необхідно n^2 вагових коефіцієнтів. Для типового мегапіксельного *RGB* зображення це означає 9 трлн. **вагових коефіцієнтів** лише для **вхідної частини** мережі. Настільки великий простір параметрів потребуватиме відповідної великої кількості зображень для навчання, а також колосальних обсягів обчислювальних ресурсів для виконання алгоритму навчання.

Описаний факт наштовхує на думку, що перший прихований шар мережі слід будувати іншим чином, щоб **кожний** прихований елемент отримував на вхід лише дані із певної **невеликої локальної області** зображення. Такий підхід дозволяє вирішити описану проблему із великою кількістю параметрів. В цілому застосувавши ці зміни можна значно скоротити кількість вагових коефіцієнтів. Припустимо, що кожна локальна область має $l << n$ пікселів, тоді тепер знадобиться всього $ln << n^2$ вагових коефіцієнтів.

Ще однією перевагою даного підходу є використання інформації про **суміжність пікселів** зображення. Це є важливою властивістю зображення, коли

в одній локальній області зображення певний невеликий об’єкт виглядатиме так само, якби цей або подібний об’єкт знаходився б в іншій локальній області зображення. Іншими словами можна очікувати, що дані, що представляються зображеннями, будуть проявляти приблизну **просторову незмінність** (*spatial invariance*), принаймні в малих та середніх масштабах. Однак подібна просторова незмінність є справедливою лише в **певних масштабах**, поза якими вона може не зберігатись.

Локальної просторової незмінності можна досягти шляхом обмеження вагових коефіцієнтів *l*, які з’єднують локальну область зображення із елементом у прихованому шарі, таким чином, щоб вони були **однаковими** для кожного прихованого елементу. Тобто, для прихованих елементи *i* та *j*, вагові коефіцієнти $w_{1,i}$, ..., $w_{l,i}$ будуть такими же, як $w_{1,j}$, ..., $w_{l,j}$. Це перетворює приховані елементи у свого роду **детектори ознак** (*features*), які виявляють одну і ту саму функцію, де б вона не знаходилась на зображенні. Як правило, потрібно, щоб **перший прихований шар**, виявляв **багато ознак**, а не лише одну. Щоб цього досягти, для кожної локальної області зображення створюють *d* прихованих елементів із *d* **різних наборів** вагових коефіцієнтів. В результаті потрібно *dl* вагових коефіцієнтів, що по-перше набагато менше за n^2, а по-друге не залежить від розміру зображення *n*. Таким чином, враховуючи певні попередні знання, а саме, знання про **суміжність пікселів** та **просторову незмінність**, можна розробити моделі мереж, які матимуть набагато менше параметрів і будуть здатні навчатись набагато швидше.

13.1.1 Операція згортки

Згортковими нейронними мережами називаються мережі, які містять **просторово локальні з’єднання** і мають структуру вагових коефіцієнтів, що **повторюється** для всіх елементів в кожному шарі (*convolutional layer*). Ця структура вагових коефіцієнтів, яка повторюється, називається **ядром** (*kernel*), а

процес застосування ядра до пікселів зображення або до чергових шарів, називається **згорткою** (*convolution*).

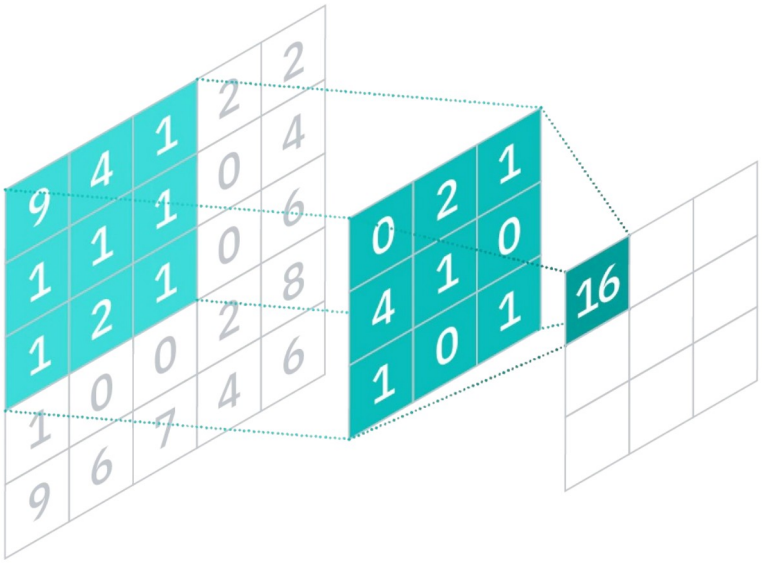


Рисунок 13.1 — Приклад обчислень у згортковому шарі

Поняття ядра та згортки краще всього можна проілюструвати на одновимірному зображені (рис. 13.1). Припустимо, що маємо зображення із n пікселів у якості вектору x, та ядро k розміром l. Операцію **згортки** $z = x * k$ можна представити наступним чином:

$$z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2} \tag{13.1}$$

За цією формулою, для кожної вхідної позиції “i” розраховується **скалярний добуток** ядра k і фрагменту x з центром в x_i та шириною l.

Окремо при згортці використовується поняття **кроку** (*stride*), коли пікселі, на яких центрується **ядро,** обираються через певну відстань. Також при застосуванні величини кроку **не кратній** розміру зображення, може виникати ситуація, коли ядро згортки буде потрапляти за межі зображення. В такому випадку виконується доповнення зображенням додатковими пікселями по краям шляхом заповнення нових пікселів “нулями” або копіюванням крайніх пікселів. Ця операція називається **доповненням** (*padding*) і дозволяю застосовувати ядро при згортку з кроком рівно n/s разів. Всього може бути d ядер, а не тільки

одне. В результаті такої згортки з кроком “1” виходів буде в *d* разів більше. Це означає, що двомірний вхідний масив перетворюється в тривимірний масив прихованих елементів, де **третя розмірність** має розмір *d*.

13.1.2 Поле сприйняття

Первинна ідея побудови згорткових нейронних мереж засновувалась на моделях **зорової кори**, які пропонувались у **нейробіології**. В цих моделях **поле сприйняття** (*receptive field*) нейрону являється тою частиною загального поля вхідних сигналів із сенсорів вводу, яка може впливати на активацію даного нейрону. В **згорткових мережах** поле сприйняття елементів **першого** прихованого шару є **малим** і дорівнює просто розміру ядра, однак на більш **глибоких** шарах воно може бути набагато **більшим**. Для прикладу із одновимірним зображенням, елемент на *m* прихованому шарі буде мати поле сприйняття розміром $(l-1)m+1$ — що означає лінійне збільшення розміру відносно *m*. Для двовимірних зображень збільшення розміру поля сприйняття збільшується квадратично.

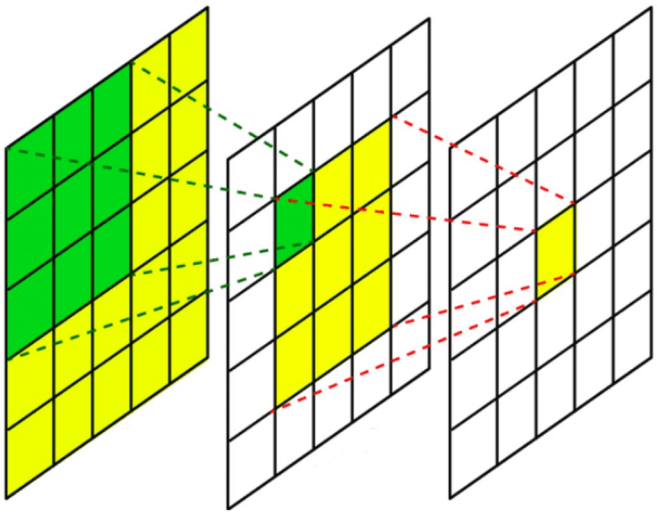


Рисунок 13.2 — Ілюстрація поля сприйняття

13.2 Об’єднання та субдискретизація

В нейронній мережі **шар об’єднання** (*pooling layer*) **узагальнює** деяку множину суміжних елементів із попереднього шару, замінюючи їх **єдиним значенням**. Шар об’єднання працює як згортковий шар із розміром ядра l та кроком s, але операція що застосовується є фіксованою і **не навчається**. Як правило, із шаром об’єднання **не пов’язана** яка-небудь функція активації. Найбільш розповсюдженими формами об’єднання є:

1. **Об’єднання за середнім** (*average pooling*), при якому обчислюється середнє значення для l входів. Ця операція ідентична згортці із одноманітним вектором ядра $k=[1/l,...,1/l]$. Якщо задати $l=s$, тоді матимемо ефект зниження розміру зображення. Буде відбуватись його **субдискретизація** (*downsample*) — ущільнення із коефіцієнтом s. В результаті, об’єкт розміром $10s$ пікселів після ущільнення буде займати всього 10 пікселів. Один і той самий **класифікатор**, який був навчаний розрізняти певний об’єкт на зображені 10 пікселів, при використанні об’єднання за середнім, тепер зможе розрізняти подібний об’єкт на **ущільнених** зображеннях, навіть якщо цей об’єкт на початковому зображенні був занадто великим для успішного розпізнавання. Застосування шару об’єднання за середнім спрощує багаторівневе розпізнавання і також дозволяє зменшити кількість вагових коефіцієнтів, що необхідні в наступних шарах. Це дозволяє **зменшувати** обчислювальні витрати і **пришвидшувати** навчання.

2. **Об’єднання за максимальним** (*max pooling*) із l вхідних значень обирає одне максимальне значення. Цей метод також може застосовуватись просто для **субдискретизації**, але при цьому він буде мати трохи іншу семантику. Об’єднання за максимальним діє як свого роду **логічне розділення**. Із певної множини вхідних значень буде обиратись одне максимальне і це свідчитиме про те, що десь у вхідних даних, які складають **поле сприйняття** елемента, присутня певна **важлива ознака**.

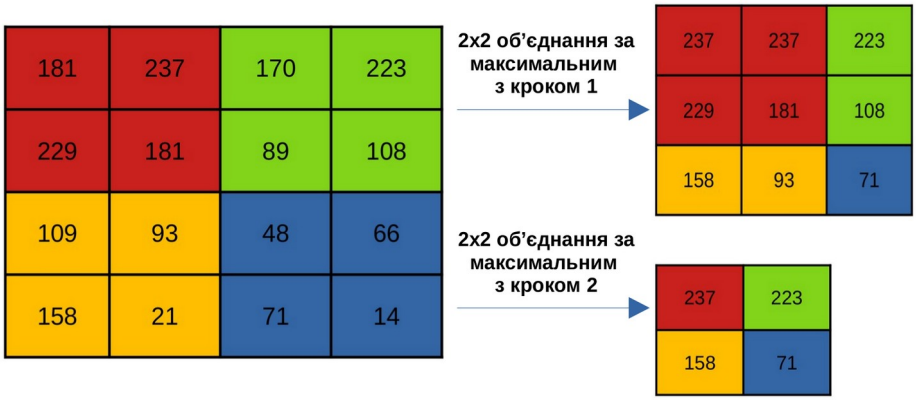


Рисунок 13.3 — Приклад об’єднання за максимальним (*max pooling*)

13.3 Шар *Softmax*

Якщо метою є **класифікація зображення** по одній із *c* категорій, тоді останнім шаром мережі застосовується шар *softmax* із *c* вихідних елементів. Розмір 1-х згорткових шарів відповідає вхідному розміру зображення, але до середини мережі розмір шарів суттєво зменшується. Для **зменшення** шарів мережі можна використовувати **згорткові шари** та **шари об’єднання** із застосуванням кроку більшого за “1”. Зменшити розмір шару можна також за рахунок **повно зв’язних шарів** з меншою кількістю елементів ніж у попередньому шарі. В згорткових нейронних мережах часто застосовують декілька повно зв’язних шарів, що передують кінцевому **шару *softmax***.

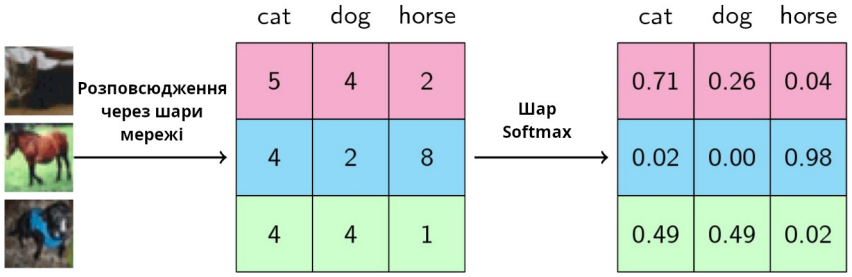


Рисунок 13.4 — Приклад шару *softmax* для класифікації

13.4 Залишкові мережі

Застосування залишкових мереж (*residual networks*) — це досить популярний і успішний підхід при побудові **дуже глибоких мереж,** який дозволяє уникнути проблеми **зникаючих градієнтів**.

Типові моделі глибокого навчання використовують шари, які вивчають нове представлення в *i*-му шарі, повністю заміщаючи ним представлення в шарі *i-1*. Це можна виразити у наступному вигляді:

$$z^{(i)} = f(z^{(i-1)}) = g^{(i)}(W^{(i)} z^{(i-1)}) \qquad (13.2)$$

Оскільки кожен шар **повністю замінює** представлення попереднього шару, всі шари повинні вивчати “робити” щось корисне. В такому випадку кожен шар повинен принаймні зберігати релевантну інформацію, яка міститься на попередньому шарі. Наприклад, якщо вагові коефіцієнти в такій мережі для будь якого *i*-го шару встановити в “0”, вся мережа перестане функціонувати. Якщо додатково встановити в “0” всі вагові коефіцієнти шару *i-1*, тоді вся мережа взагалі **втратить можливість навчатись**: *i*-й шар перестане навчатись, оскільки він не буде отримувати ніякої інформації про зміни на його входах від шару *i-1*, а в свою чергу шар *i-1* не зможе навчатись через те, що **градієнти** при зворотному розповсюджені помилки з *i*-го шару завжди будуть рівними “0”.

Основна ідея **залишкових мереж** полягає у тому, що кожен шар повинен **порушувати** представлення, отримане із попереднього шару, а **не змінювати** його повністю. Цей процес можна представити наступною формулою:

$$z^{(i)} = g_r^{(i)}(z^{(i-1)} + g^{(i)}(W^{(i)} z^{(i-1)})) = g_r^{(i)}(z^{(i-1)} + f(z^{(i-1)})) \quad , \qquad (13.3)$$

де вектор g_r представляє функцію **активації** для **залишкового шару**.

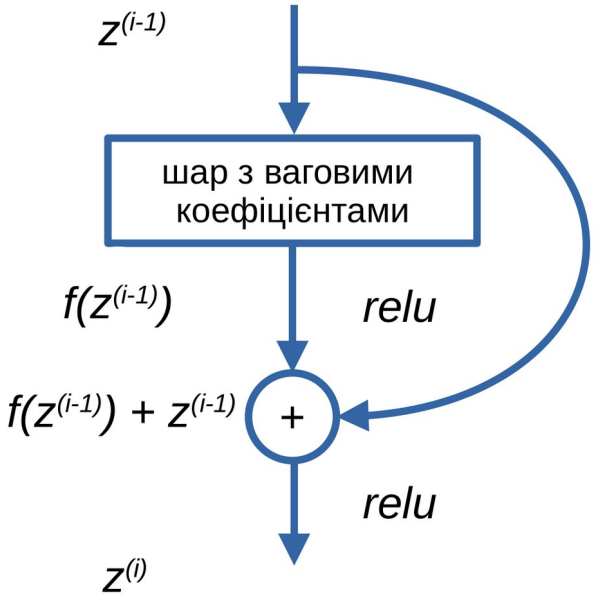


Рисунок 13.5 — Приклад залишкового шару/блоку

Тут під *f* розуміється **залишковий блок** (*residual*), який порушує звичний процес передачі інформації від шару *i-1* до шару *i*. Якщо встановити вагові коефіцієнти *i*-го шару в “0”, тоді рівняння спрощується до:

$$z^{(i)} = g_r^{(i)}(z^{(i-1)}) \tag{13.4}$$

Тепер якщо припустити, що залишковий шар g_r складається із функції активації *ReLU* і в $z^{(i-1)}$ також застосовувався *ReLU* для вхідних даних $z^{(i-1)} = ReLU(\text{in}^{(i-1)})$, враховуючи що $ReLU(ReLU(x)) = ReLU(x)$, матимемо:

$$z^{(i)} = g_r(z^{(i-1)}) = ReLU(z^{(i-1)}) = ReLU(ReLU(\text{in}^{(i-1)})) = ReLU(\text{in}^{(i-1)}) = z^{(i-1)} \tag{13.5}$$

Зазначену формулу можна пояснити тим, що в залишкових мережах, із функцією активації *ReLU*, шар з нульовими ваговими коефіцієнтами просто **пропускає** вхідні дані без змін. Подальша частина мережі функціонує таким чином, ніби даного шару в ній не існує. **Звичайні мережі** повинні були б навчатись розповсюджувати інформацію в подібній ситуації і були б схильні до відмов, в той час як **залишкові мережі** по своїй природі здатні надалі розповсюджувати інформації в таких ситуаціях.

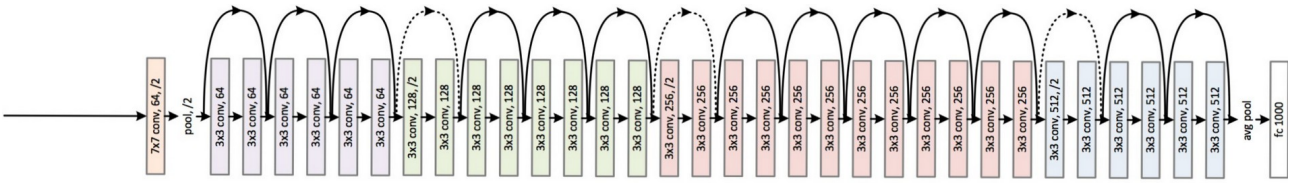


Рисунок 13.6 — Архітектура мережі *ResNet* 34

Залишкові мережі часто застосовуються із згортковими шарами в застосунках для роботи із візуальними даними, але їх застосування не обмежується лише цим. Вони є універсальним інструментом, який робить глибокі мережі більш надійними і дає можливість дослідникам експериментувати із більш складними і різнорідними мережевими конструкціями. Наразі існують залишкові мережі із 100 і більше шарами, наприклад, мережі сімейства *ResNet*.

13.5 Покращення можливостей моделі узагальнювати дані

Загалом можливість мережі **узагальнювати** дані можна в значній мірі покращити при використанні первинних **знань** про вхідні дані і використанні правильної **структури** та типу мережі. Але також можна додатково модифікувати алгоритм навчання, наприклад, стохастичний градієнтний спуск із порцією даних таким чином, щоб вводити в нього певну **регуляризацію**.

13.5.1 Метод скорочення вагових коефіцієнтів

Аналогічно до машинного навчання, в моделях глибокого навчання також застосовується регуляризація, що обмежує складність моделі та покращує її здатність узагальнювати дані. В контексті нейронних мереж такий підхід називається **скороченням вагових коефіцієнтів** (*weight decay*).

Його ідея полягає у додавані певного штрафу до функції втрат, яка використовується при навчанні мережі:

$$Cost = \lambda \sum_{i,j} W_{i,j}^2 + Loss \qquad (13.6)$$

де λ є гіперпараметром, який визначає величину цього штрафу, і зазвичай розраховується як сума по всім ваговим коефіцієнтам. Чим більше значення цього гіперпараметру обирається, тим швидше будуть змінюватись вагові коефіцієнти. Зазвичай в методі скорочення вагових коефіцієнтів застосовується значення $\lambda = 10^{-4}$.

13.5.2 Метод виключення

Іншим способом, який покращує узагальнення даних і характеризується зменшенням помилки на тестовому наборі даних, є ускладнення досягнення відповідності виводу моделі навчальному набору. Цей метод називається **виключенням** (*dropout*). В даному методі на кожній ітерації навчання мережі застосовується етап навчання шляхом зворотного розповсюдження помилки в **новій версії** мережі, яка створюється шляхом **деактивації** обраної випадковим чином підмножини елементів шарів оригінальної мережі. Фактично цей метод є дуже простим наближенням до навчання великого **ансамблю** різних нейронних мереж.

Більш конкретно можна розглянути приклад мережі із використанням стохастичного градієнтного спуску із порцією даних (*minibatch*) розміром *m*. Для кожної чергової порції даних алгоритм виключення із ймовірністю *p* множить вихід кожного елементу певного шару мережі на коефіцієнт *1/p*, а з ймовірністю *1-p* вихід відповідного елементу встановлюється рівним “0”. В методі виключення зазвичай застосовується ймовірність 0,5 для елементів **прихованих шарів**, і 0,8 — для **вхідних** елементів. В результаті застосування даної процедури отримується **розріджена мережа**, яка буде містити в два рази менше елементів ніж початкова мережа до якої застосовується алгоритм навчання. Процес повторюється до тих пір, поки навчання не буде завершене. Під час **тестування** модель запускається без виключень — використовується початкова версія із повною кількістю елементів в кожному шарі.

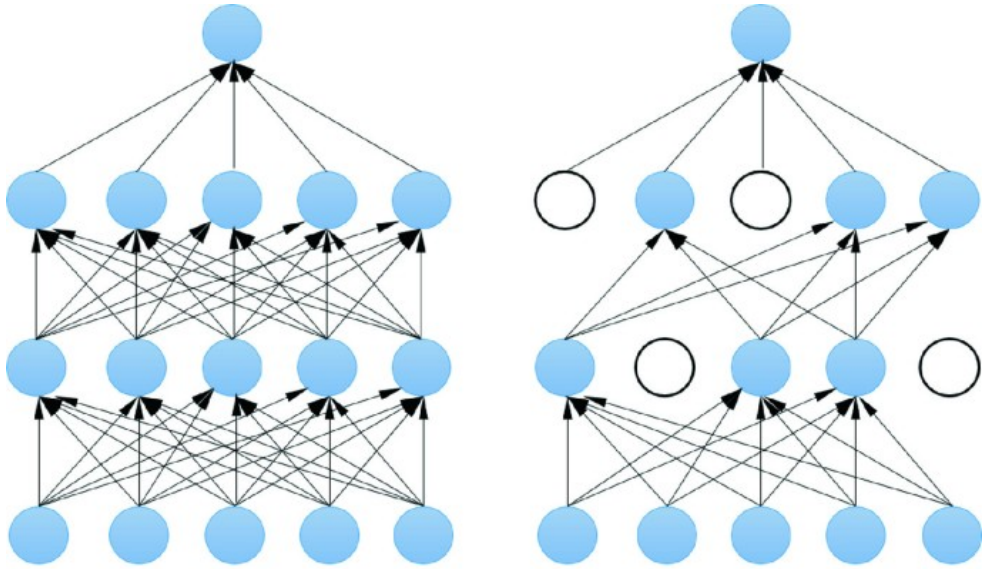


Рисунок 13.7 — Приклад мережі “до” (лівий) та “після” застосування методу виключення (правий)

Застосування методу виключення дозволяє **покращити** модель, а саме:

1. **Введення шуму** (вимкнення елементів) під час навчання мережі повинно покращити стійкість моделі до зашумлених даних;

2. Приховані елементи, які навчають методом виключення, навчаються не тільки бути **корисними** в цих шарах, але і бути **сумісними** із багатьма іншими можливими наборами елементів в інших прихованих шарах, які можуть бути, а можуть і не бути присутніми у моделі мережі;

3. Виключення, яке застосовується для більш пізніх шарів глибокої мережі, “змушує” мережу звертати увагу на більшу кількість абстрактних особливостей кожного прикладу даних, а **не концентруватись** тільки на одній особливості, ігноруючи інші. Наприклад, при класифікації зображення модель може пов’язати приналежність певного прикладу до конкретного класу при наявності в ньому певного об’єкту — властивості. При цьому, в разі якщо на іншому побідному прикладі ця властивість буде не чітко видна, мережа може не вірно класифікувати цей приклад. А із методом виключення подібна мережа зможе знайти **більше** інших характерних властивостей для того ж самого прикладу об’єкту.

В цілому застосування методу виключення покращує можливості моделі узагальнювати дані, але й також ускладнює досягнення високої точності відповідності моделі навчальному набору даних. Зазвичай для цього застосовують **більші** моделі, які навчаються на **більшій** кількості ітерацій.

13.6 Рекурентні нейронні мережі

Рекурентні нейронні мережі (*recurrent neural network — RNN*) відрізняються від мереж із прямим зв’язком тим, що допускають наявність **циклів** в графі обчислень. В подібних випадках цикл має **затримку**, що дозволяє елементу приймати в якості вхідних даних значення, обчислені на базі власних вхідних даних, що були отримані на більш ранніх етапах розрахунків. Такий підхід дозволяє рекурентним мережам мати внутрішній стан або **пам’ять**. Це означає, що вхідні дані, отримані на ранніх часових етапах впливають на відповідь рекурентної мережі для поточних вхідних даних.

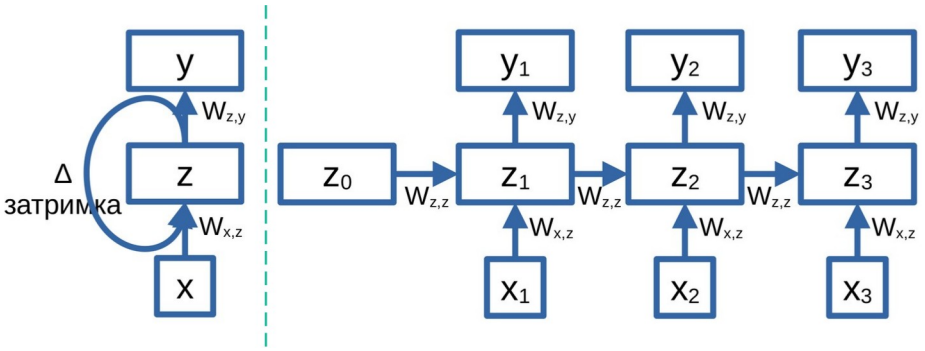


Рисунок 13.8 — Приклад рекурентної мережі

Рекурентні мережі застосовуються в якості інструменту аналізу послідовних даних, коли робиться припущення, що новий **вхідний вектор** x_t поступає на кожному часовому етапі. Подібна мережа використовує **прихований стан** z_t для фіксації інформації зі всіх попередніх входів. Процес оновлення прихованого стану рекурентної мережі можна представити наступною формулою:

$$z_t = f_w(z_{t-1}, x_t) \ , \qquad (13.7)$$

де f_w - деяка параметризована функція.

Після навчання ця функція представляє собою деякий **стаціонарний процес** — по суті певну динаміку, представлену функцією f_w , яка залишається незмінною для всіх часових етапів. В результаті цього **рекурентні мережі** для **послідовних даних** мають кращу продуктивність, а ніж мережі із прямим зв’язком. В дійсності, якщо спробувати використати мережу із прямим зв’язком для аналізу послідовних даних, фіксований розмір вхідного шару змусить мережу опрацьовувати ці вхідні дані окремими порціями фіксованого розміру. Це означатиме, що звичайна мережа не зможе виявити важливі залежності в даних на достатньо великих відстанях.

Особливістю розрахунків градієнтів в рекурентних мережах є те, що вираз для градієнтів є **рекурсивним**. Внесок в градієнт від часового етапу *t* розраховується із використанням вкладу від часового етапу *t-1*. Для правильно впорядкованих обчислень, загальний час виконання обчислень по розрахунку градієнту буде лінійно пропорціональним розміру мережі. Даний алгоритм розрахунків градієнтів називається **зворотне розповсюдження помилки у часі** (*back-propagation through time*) і як правило **реалізований** у програмних бібліотеках для систем глибокого навчання.

13.6.1 Рекурентні нейронні мережі із довгою короткотривалою пам’яттю

Для забезпечення зберігання інформації протягом багатьох часових етапів були розроблені спеціалізовані архітектури рекурентних мереж. Одна з них називається мережею із **довгою короткочасною пам’яттю** (*long short-term memory — LSTM*). В архітектурі *LSTM* компонент довготривалої пам’яті називається **елементом пам’яті** — *c*. По суті його зміст копіюється від одного часового етапу в інший. Нова інформація потрапляє у пам’ять шляхом **додавання оновлень**, за рахунок чого вирази градієнту не накопичується мультиплікативно з плином часу. Архітектура *LSTM* також включає в себе

вентилі (*gating unit*), які представляють собою вектори, що керують потоком інформації в мережі шляхом поелементного добутку відповідного **інформаційного вектору**. В *LSTM* передбачено декілька типів вентилів:

1. **Вентиль забування *f*** визначає чи **запам'ятовується** кожне значення елементу пам'яті — копіюється на наступний етап, чи **забувається** — скидається в нуль;

2. **Вхідний вентиль *i*** визначає чи буде кожне значення елементу пам'яті **адаптивно оновлюватись** новою інформацією із вхідного вектору на поточному часовому етапі.

3. **Вихідний вентиль *o*** визначає чи передається кожне значення елементу пам'яті в довгу короткотривалу пам'ять, яка відіграє роль **прихованого стану** в базовій архітектурі рекурентної мережі.

$$f_t = \sigma(W_{x,f} x_t + W_{z,f} z_{t-1}) \tag{13.8}$$

$$i_t = \sigma(W_{x,i} x_t + W_{z,i} z_{t-1}) \tag{13.9}$$

$$o_t = \sigma(W_{x,o} x_t + W_{z,o} z_{t-1}) \tag{13.10}$$

$$c_t = c_{t-1} \odot f_t + i_t \odot \tanh(W_{x,c} x_t + W_{z,c} z_{t-1}) \tag{13.11}$$

$$z_t = \tanh(c_t) \odot o_t \tag{13.12}$$

В *LSTM* під "вентилем" розуміється реалізація певної "м'якої" булевої функції, при якій значення вектору елементу пам'яті будуть частково забуті в разі, якщо відповідні значення вектору вентиля забування будуть малими, але не рівними нулю. Значення вентильних елементів завжди знаходяться в діапазоні [0, 1].

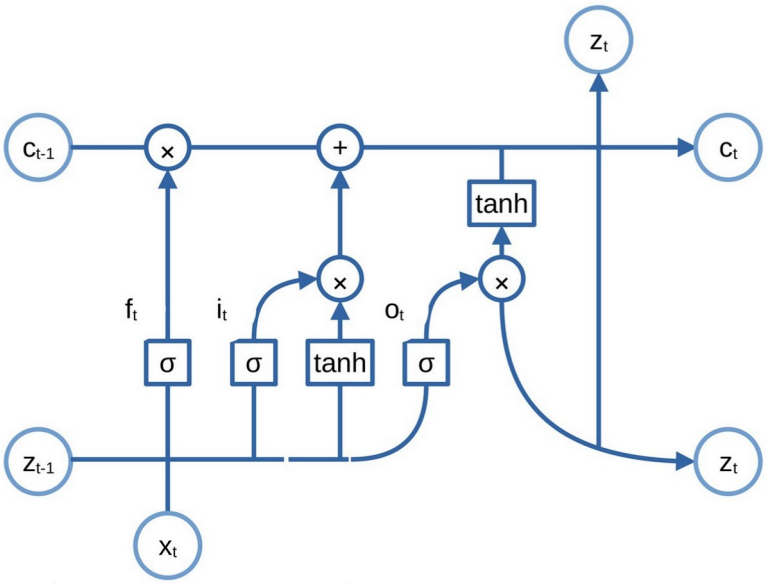


Рисунок 13.9 — Приклад мережі *LSTM*

Архітектура *LSTM* була однією з перших застосованих на практиці форм рекурентних нейронних мереж. Системи на її основі демонструють високу продуктивність на широкому переліку задач, що включають розпізнавання **природної мови** та розпізнавання **рукописного тексту**.

ТЕМА 14
ЗАСТОСУНКИ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ

14.1 Складності при навчанні із вчителем

Попередніми темами розглядались системи глибокого навчання, які будувались за принципом **навчання із вчителем.** В такому підході, кожний навчальний приклад був пов’язаний із міткою, що відповідала певному значенню цільової функції. Хоча подібні системи, які використовують навчальні дані, можуть досягати високих показників точності на тестових наборах даних, часто виникає ситуація, що для успішного навчання подібні системи потребують дуже **великої кількості** розмічених даних. Якщо порівнювати це із виконанням **задачі розпізнавання** людиною, то для успішного розпізнавання певного об’єкту їй буде достатньо переглянути всього декілька зображень з цим об’єктом. Після чого людина без особливих зусиль зможе розпізнавати аналогічні об’єкти в широкому діапазоні навколишніх середовищ, ракурсів, відстаней і т.д. Очевидно, що в підході глибокого навчання ще є багато аспектів, які можуть бути покращенні. В дійсності, в поточному підході навчання із вчителем можуть мати місце випадки, при яких стане **неможливим вирішення** деяких задач, через неможливість надання необхідної кількості розмічених даних, що відображатимуть всі можливі представлення необхідних об’єктів. Більше того, створення достатньо великих наборів розмічених даних, в результаті потребуватиме колосальних витрат людського часу для їх розмітки.

Рисунок 14.1 — Приклад навчального прикладу для задачі локалізації (лівий) та сегментації (правий)

Через ці причини існує великий інтерес до інших підходів у навчанні, які **зменшують залежність** процесу навчання від **розмічених даних**. До подібних підходів відносяться **навчання без вчителя** (*unsupervised learning*), перенесення навчання або **трансферне навчання** (*transfer learning*), **навчання з частковим залученням вчителя** або часткове навчання (*semisupervised learning*).

Алгоритми навчання без вчителя проводять навчання виключно на нерозмічених вхідних даних, які зазвичай доступні у достатній кількості, а ніж розмічені приклади. Алгоритми навчання без вчителя створюють **породжуючі моделі**, які здатні **генерувати реалістичні** тексти, зображення, аудіо та відео замість простого передбачення мітки для подібних вхідних даних.

Алгоритми трансферного навчання потребують невеликої кількості розмічених даних. При цьому вони здатні додатково покращувати свою ефективність шляхом використання навчальних даних з інших **суміжних задач**, таким чином роблячи можливим використання **додаткових** джерел даних.

Алгоритми навчання із частковим залученням вчителя також потребують невеликої кількості розмічених даних, але в подальшому вони здатні покращувати свою продуктивність за рахунок використання **нерозмічених** прикладів.

14.2 Навчання без вчителя

Всі алгоритми **навчання із вчителем** мають одну загальну рису, а саме: при заданому навчальному наборі входів x і відповідних виходів $y=f(x)$ знайти в ході навчання таку функцію h , яка буде добре апроксимувати цільову функцію f. Напротивагу алгоритми **навчання без вчителя** на вхід отримують навчальний набір, що містить лише нерозмічені приклади вхідних даних x. В цілому дані алгоритми переслідують дві основні мети:

1. вивчення **нових представлень**, наприклад нових ознак на зображеннях, що полегшать ідентифікацію об’єктів на них;

2. вивчення **породжуючих моделей,** зазвичай у формі розподілу ймовірностей, на базі яких можуть бути згенеровані нові вибірки даних.

14.2.1 Автокодувальник

Більшість алгоритмів глибокого навчання без вчителя засновані на ідеї **автокодувальника** (*autoencoder*). Автокодувальник — це модель, яка складається із двох частин:

1. **кодування**, яке відображає дані із вектору *x* в представлення $\hat{z}$;

2. **декодування**, яке відображає дані представлення $\hat{z}$ в дані спостереження *x*.

В загальному вигляді **кодувальник** (*encoder*) — це просто параметризована функція *f*, а **декодувальник** (*decoder*) — параметризована функція *g*. Модель **навчається** таким чином, щоб результат кодування даних був зворотним результату декодування — $x \approx g(f(x))$. Функції f та g можуть бути простими лінійними моделями, які параметризовані одиничною матрицею, або вони можуть бути представлені глибокою нейронною мережею.

Найпростішим варіантом є лінійний автокодувальник, коли функції f та g являються лінійними із спільною матрицею вагових коефіцієнтів. Це можна представити наступними формулами:

$$\hat{z} = f(x) = Wx \tag{14.1}$$

$$x = g(\hat{z}) = W^T \hat{z} \tag{14.2}$$

Одним із способів навчання подібної моделі є **мінімізація квадратичної помилки**:

$$L_2 = \sum_j \left\| x_j - g(f(x_j)) \right\|^2 \tag{14.3}$$

Ідея полягає у навчанні матриці вагових коефіцієнтів *W* таким чином, щоб низькорозмірний вектор $\hat{z}$ зберігав як можна більше інформації, щоб забезпечити можливість **відновлення** багатомірних даних *x*. Більш складні види **породжуючих моделей** використовують складніші види автокодувальників, одним з яких є **варіаційний автокодувальник** (*variational autoencoder — VAE*).

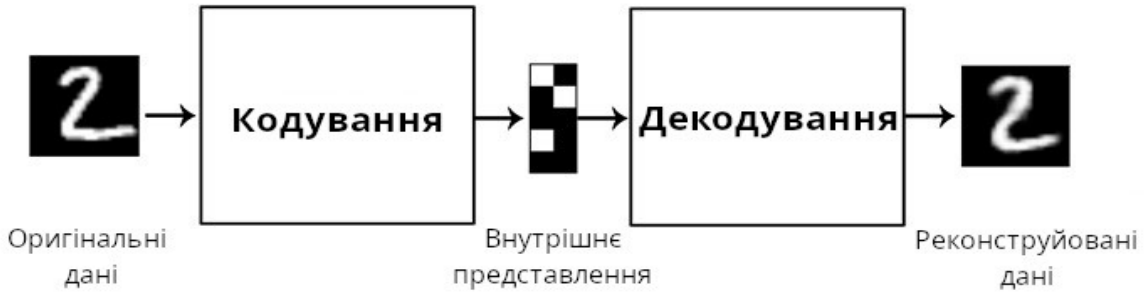


Рисунок 14.2 — Принцип роботи автокодувальника

14.2.2 Генеративно-змагальні мережі (*generative adversarial network — GAN*)

Мережа *GAN* фактично є двома мережами, які об’єднані з метою створення породжуючої системи. Одною частиною такої системи є **мережа генератор**, що відображає значення вектору *z* на вектор *x*. Типова схема генератора отримує на вхід випадкові дані *z* “шум”, що можуть представлятись, наприклад, рівномірним розподілом, а потім пропускає ці дані через глибоку нейронну мережу для отримання вихідних даних *x*. Інша мережа — **дискримінатор** представляє собою класифікатор, який навчаний розпізнавати дані *x* від генератора як **нереальні** та приклади із навчального набору як **реальні**. Генератор та дискримінатор **почергово** навчаються. При цьому генератор **вчиться** створювати такі дані, які дискримінатор розпізнаватиме як **реальні**. В свою чергу дискримінатор вчиться як можна краще розпізнавати дані, створені генератором, як **нереальні**. Загальний процес навчання мереж *GAN* полягає у досягнені **рівноваги**, коли генератор буде створювати ідеальні дані, щоб дискримінатор не міг виявити підробку вхідних даних. В такому випадку точність розпізнавання дискримінатора має складатиме 50% - він висуватиме **випадкові здогадки** про реальність чи нереальність згенерованих даних. Системи *GAN* особливо добре працюють у застосунках генерації зображень. Наприклад, генеративно-змагальні мережі можуть створювати фотореалістичні зображення людей, сцен для комп’ютерних ігор, кадрів фільмів і т.д.

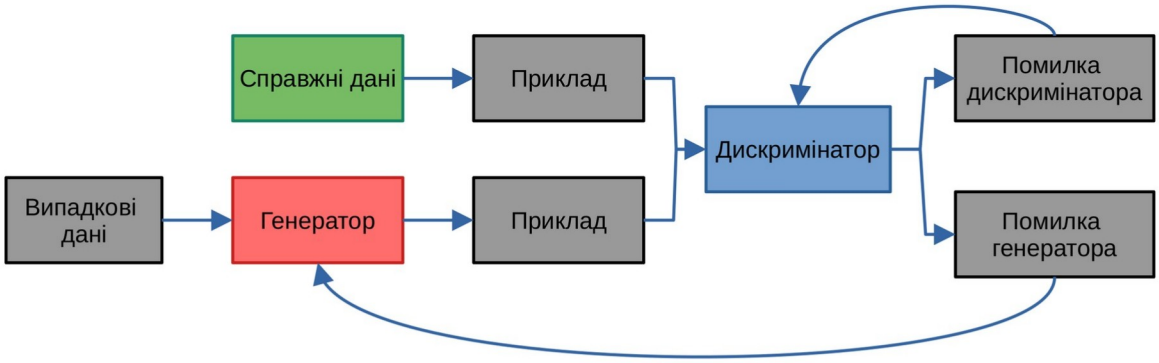


Рисунок 14.3 — Принцип роботи мережі *GAN*

14.2.3 Трансферне навчання

При **трансферному навчанні** досвід, який був отриманий для однієї задачі навчання, **допомагає** агенту краще навчатись в інших задачах. На прикладі із людиною це можна проілюструвати тим, що пілот, який навчився керувати певним типом пасажирського літака, набагато швидше зможе освоїти новий тип літака, маючи свій досвід, а ніж людина, яка навчається льотній справі з нуля.

Для нейронних мереж процес навчання полягає у **корегуванні вагових коефіцієнтів** для максимального наближення результату обробки даних до навчальних прикладів. В цьому контексті, підхід трансферного навчання полягає у **копіюванні** вагових коефіцієнтів, отриманих при навчанні для **задачі А**, в нейронну мережу, яка навчатиметься вирішувати **задачу Б** замість ініціалізації вагових коефіцієнтів випадковим чином. Далі ці скопійовані вагові коефіцієнти піддаються процесу **додаткового навчання** (*tuning* або *fine tuning*) вже для задачі Б. При такому процесі можна використовувати трохи меншу швидкість навчання, залежно від того, наскільки подібними були задачі А та Б, і скільки навчальних даних використовувалось в першій задачі. Важливою особливістю даного підходу в навчанні є те, що тепер для навчання моделі в задачі Б знадобиться набагато менше навчальних даних, і вони тепер можуть бути більш **вузькоспеціалізованими** в представленні саме об’єктів

характерних для другої задачі Б. А розпізнавання моделлю більш загальних ознак об’єктів забезпечується перенесеним **досвідом** із першої задачі.

Окремо слід відмітити, що для застосування трансферного навчання знадобиться певний **аналіз** експертом подібності задачі. Очевидно, що вагові коефіцієнти мережі навчаної **перекладати тексти** принесуть мало користі для мережі, що розпізнаватиме **друкований текст** із зображень. Також, оскільки вагові коефіцієнти пов’язані із відповідними шарами нейронної мережі, для їх успішного копіювання між двома задачами знадобиться використання **ідентичних мережевих архітектур**.

Однією з ключових причин популярності трансферного навчання стала доступність високоякісних **навчаних моделей** нейронних мереж для різних класів задач. Наприклад, можна завантажити попередньо навчану модель візуального розпізнавання об’єктів, таку як *ResNet-50*, що була навчана на наборі даних *ImageNet*. Це дозволить зекономити тижні роботи в разі навчання подібної мережі з нуля. Далі можна змінити параметри моделі і надати їй додаткові зображення з мітками об’єктів, що характерні вже для власної конкретної задачі. Дуже часто при використанні трансферного навчання є необхідність **заморозити** (*freeze layer*) декілька перших шарів попередньо навчаної мережі, оскільки ці шари працюють в якості **детекторів ознак**, що будуть корисними для новоствореної моделі. Під час навчання на прикладах для нової задачі буде дозволено змінювати вагові коефіцієнти лише більш **глибших шарів**. Це шари, які розпізнають більш **специфічні** ознаки характерні для проблемної області.

Загалом трансферне навчання дає можливість моделі адаптуватись під нові дані та умови застосування.

14.3 Глибоке навчання в комп’ютерному зорі

Глибоке навчання успішно застосовується в багатьох проблемних областях штучного інтелекту. **Комп’ютерний зір** можна назвати областю на яку

найбільше вплинуло глибоке навчання. Незважаючи на те, що глибокі згорткові мережі використовувались для таких задач, як розпізнавання рукописного тексту, ще на початку 90-х років минулого сторіччя, лише у 2012 році, завдяки успіху **глибокої мережі *AlexNet*** в конкурсі *ImageNet*, глибоке навчання змогло привернути до себе увагу спільноти в області штучного інтелекту.

Конкурс ImageNet представляв собою задачу машинного навчання із вчителем на базі навчального набору із 1 200 000 зображень, які відносяться до 1000 різних категорій. Мережа ***AlexNet*** на той час досягла рівня помилки у 15,3%, в той час як інші тодішні системи мали помилку у 25%. Мережа *AlexNet* включала п’ять **згорткових шарів** (*conv*), які поєднувались **шарами об’єднання** за максимальним (*maxpooling*), після яких слідувало три **повно зв’язних шари**. В якості функції активації використовувалась функція *ReLU*. Загалом мережа містила **60 млн.** вагових коефіцієнтів, а для прискорення розрахунків в процесі навчання використовувались **графічні прискорювачі**.

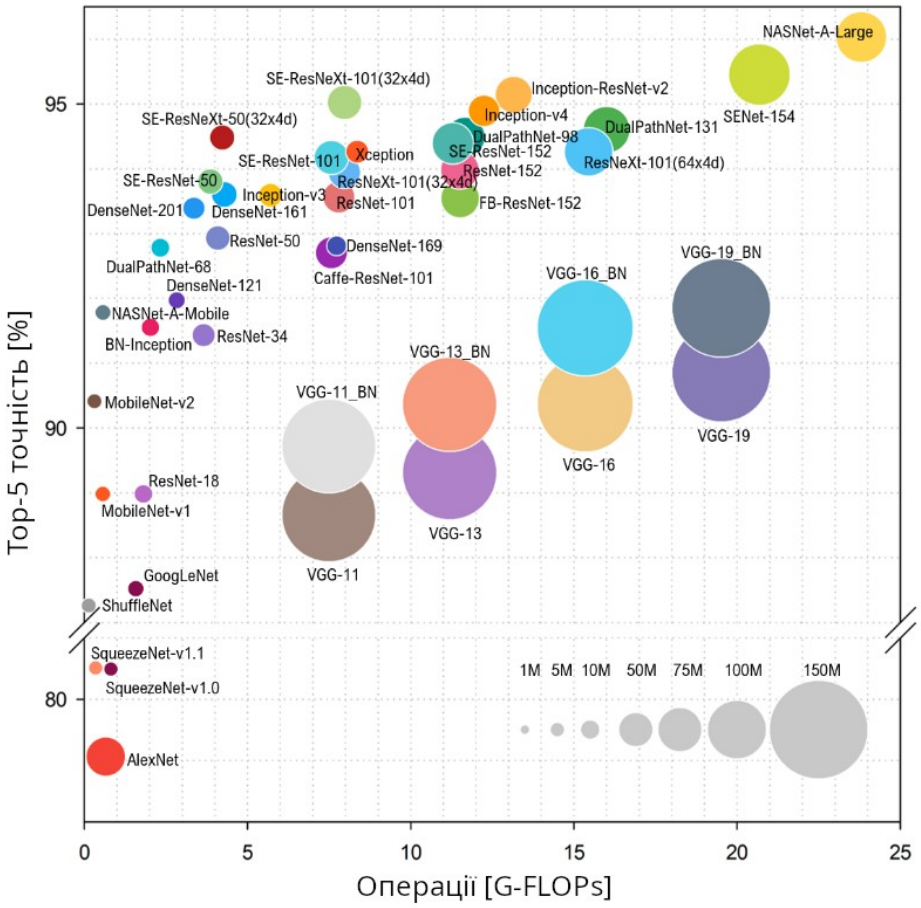


Рисунок 14.4 — Точність та складність сучасних архітектур глибоких нейронних мереж

З 2012 року завдяки покращенням в архітектурах нейронних мереж, методів навчання, та обчислювальних ресурсів, показники рівня помилок в задачах розпізнавання зменшився до 2%, що є менше за рівень помилок самої людини — приблизно 5%. Згорткові нейронні мережі стали застосовуватись для вирішення широкого кола задачі комп’ютерного зору, від розробок безпілотних автомобілів, систем безпеки, медицини, аналізі рухів людини, тощо.

14.3.1 Задачі комп’ютерного зору

Задачі комп’ютерного зору можна розділити у декілька загальних категорій: **класифікація**, **детекція** та **сегментація**.

Класифікація, про яку вже згадувалось, є найпростішим класом задач, коли відбувається розпізнавання об’єкту на зображені і його віднесення до певного класу із переліку можливих класів.

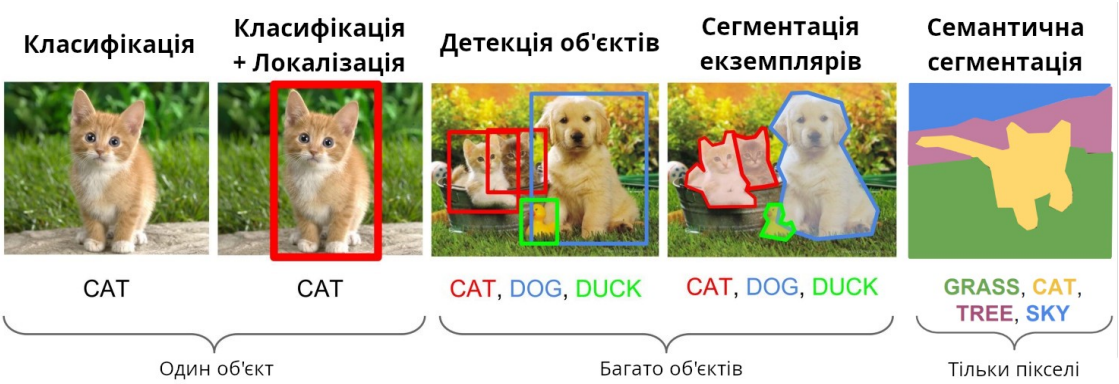


Рисунок 14.5 — Приклади задач комп’ютерного зору

14.3.2 Детекція об’єктів на зображенні

Наступною за деталізацією результату є задача **детекції** або виявлення об’єкту. На відміну від класифікатора, який просто видає приналежність зображення із об’єктом до певного класу, детектор **знаходить** даний об’єкт або множину об’єктів на зображенні, видає інформацію до якого класу вони належать, а також **показує** де на зображенні знаходиться кожний об’єкт. Місце знаходження кожного об’єкта відображається за допомогою **обмежувальної**

рамки (*bounding box*). Детектор об’єктів можна побудувати, розглядаючи велике зображення через невелике прямокутне вікно, що переміщується (*sliding window*). В кожній позиції вікна за допомогою класифікатора виявляються об’єкти, які в ньому присутні. Після цього обираються результати класифікації із високими оцінками, а всі інші відкидаються. В кінці отримуються набори об’єктів з їх позиціями.

Мережа, призначена для виявлення сегментів з об’єктами, називається **мережею регіональних пропозицій** (*regional proposal network — RPN*). Створення детектору об’єктів починається із кодування великої кількості обмежувальних рамок у вигляді карт фіксованого розміру. Далі створюється мережа, яка здатна прогнозувати присутність об’єкту у кожній рамці. Для кодування рамок використовується значення їх **центральних точок** на зображенні. Ці рамки можуть знаходитись у різних точках зображення, але не має потреби розглядати всі можливі точки на зображенні. Натомість центральні точки обираються із **кроком** у певну кількість пікселів. Для кожної центральної точки розглядаються декілька можливих рамок, які називають **якірними рамками** (*anchor boxes*). Такі якірні рамки можуть характеризувати рамки різних типів, наприклад, малі, середні, великі, високі, широкі, з різними пропорціями, тощо. Наступним кроком обираються рамки, які мають кращі **показники можливого вмісту** в них об’єкту, і рамки з достатньо високим показником проходять перевірку класифікатором. В результаті отримуємо велику кількість рамок, що містять об’єкти, при чому багато з цих рамок представляють один і той самий об’єкт (рис. 14.6, лівий). Для відображення лише потрібних рамок, які мають кращу оцінку, множина рамок проходить через алгоритм **немаксимального видалення** (*non-maximum suppression — NMS*).

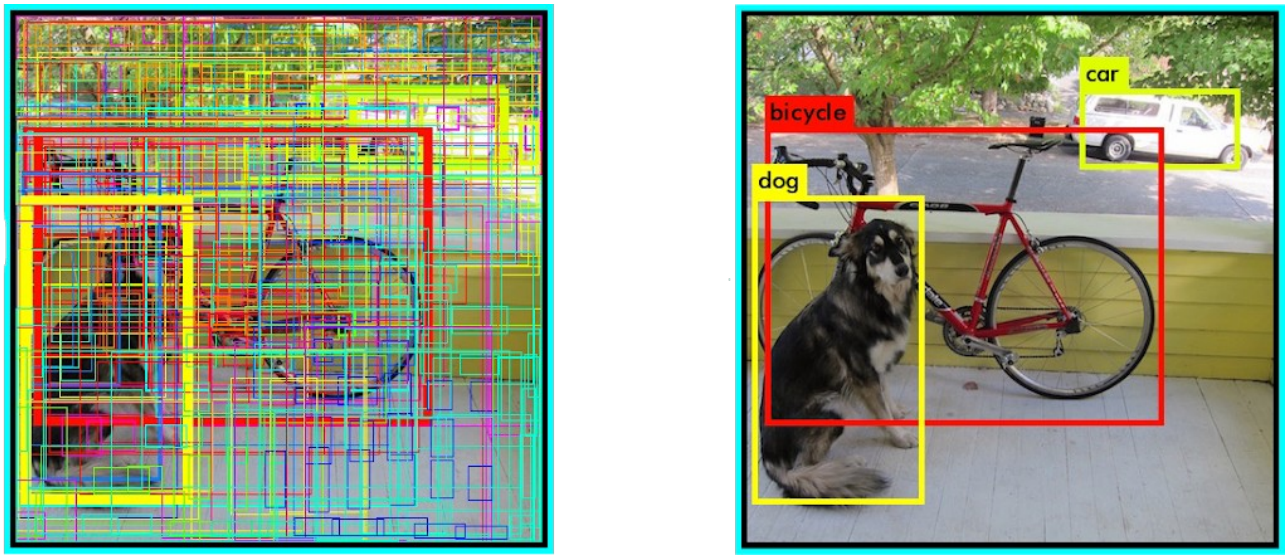


Рисунок 14.6 — Результат “до” (лівий) та “після” (правий) застосування алгоритму немаксимального видалення

Цей алгоритм сортує всі рамки із залишає лише ті, оцінка яких перевищує певний поріг. Далі обирається рамка з найбільшою оцінкою вмісту об’єкту, після чого із списку видаляються всі інші рамки, що в значній мірі перетинаються із обраною рамкою (рис. 14.6, правий). Рамки, які залишились містять потрібні об’єкти, але їх позиція може залишатись неточною. Завершальним етапом є **покращення положення** та розміру рамки, щоб точно обмежувати об’єкт, за рахунок використання представлення ознак, розрахованих класифікатором. Цей етап називається **регресією обмежувальної рамки** і дозволяє обрізати обмежувальну рамку до правильних розмірів.

Популярною метрикою для оцінки точності локалізації є **відношення площ перетину та об’єднання** (*intersection over union — IoU*). Для її розрахунку спочатку знаходиться область перетину отриманої рамки із еталоном (*ground truth*) і потім розраховується її площа. Після знайдена площа перетину ділиться на загальну площу двох об’єднаних рамок — знайденої та еталонної (рис. 14.7). Значення даної метрики дає **відсоток перекриття** обох рамок. Якщо рамки повністю перекриваються, це означає найкращу точність знайденої обмежувальної рамки, і навпаки.

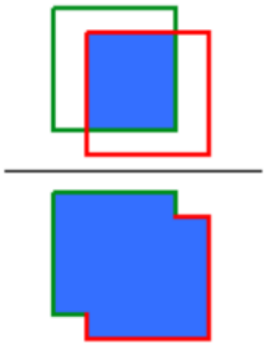


Рисунок 14.7 — Відношення площ перетину та об’єднання

Навчальні дані для створення **детекторів** також трохи відрізняються. Окрім мітки класу до якого належить об’єкт, кожен приклад тепер повинен містити інформацію про **обмежувальну рамку** цього об’єкта. Зазвичай це включає **координати** центральної або кутової точки рамки та її **розмір**.

14.3.3 Сегментація зображень

Сегментація — це процес розбиття зображення на **групи** пікселів, що мають **схожі характеристики**. Основна ідея полягає у тому, що кожен піксель зображення може бути пов’язаний з певними візуальними атрибутами такими, як колір, яскравість, текстура, тощо. В межах одного об’єкту або однієї частини об’єкту ці атрибути будуть змінюватись відносно мало, тоді як при перетині границь між двома об’єктами зазвичай відбувається суттєва та різка зміна цих атрибутів. Ця задача є набагато складнішою, порівнюючи із класифікацією та детекцією, оскільки в ній потрібно вказувати не приблизну позицію об’єкту як обмежувальна рамка при детекції, а **точні границі** кожного об’єкту. Часто для покращення результатів в даних задачах мережа доповнюється можливістю виокремлювати інформацію про **контекст зображення,** що покращує передбачення об’єктів. Наприклад, якщо мережа бачить, що велика область зображення відображає воду, то більш ймовірно і логічно, що об’єкт який знаходиться в цій області буде човном, а не автомобілем.

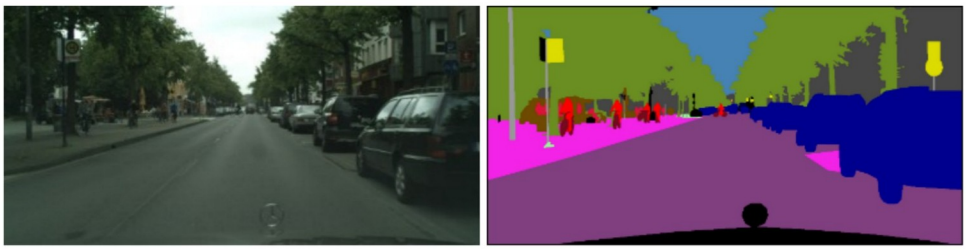


Рисунок 14.8 — Зображення “до” (лівий) та “після” (правий) сегментації

14.4 Глибоке навчання в обробці природної мови

Глибоке навчання також сильно вплинуло на застосунки обробки природної мови (*natural language processing — NLP*) таких, як машинний переклад та розпізнавання мови. Переваги глибокого навчання для таких систем включають можливість використання **наскрізного навчання** (*end-to-end learning*), автоматичного генерування **внутрішнього представлення** для значень слів, можливості взаємозамінності навчаних **кодувальників** та **декодувальників**. Маючи розмічені набори даних, скажімо для перекладу речень, подібні системи глибокого навчання можуть навчатись виконувати переклад, шляхом вивчення певної функції відображення речення на одній мові в іншу мову. При цьому класичний **конвеєрний підхід** у побудові систем перекладу потребував кропіткої людської роботи по підготовці даних, їх аналізу, переведені у представлення в іншій мові, виражене у мовній моделі. В результаті застосування нових підходів системи глибокого навчання для перекладу досягли 60% зменшення помилок перекладів, порівнюючи із конвеєрними системами. З 2020 року точність систем машинного перекладу на певних мовних парах наблизилась до точності перекладу людиною. Важливою особливістю застосування глибокого навчання для обробки мови полягає у багаторазовому представленні слів як векторів в багатовимірному просторі. Дані вектори отримуються в ході навчання мережі на великих обсягах тексту і містять інформацію лексичного контексту — де використовуються певні слова. Слова за схожими значеннями та контекстом, в якому вони використовуються, розташовуються ближче один до одного у векторному просторі, що дозволяє

мережі ефективно узагальнювати слова по власним внутрішнім категоріям без необхідності визначення цих категорій людиною.

Значних покращень результатів в задачах *NLP* було досягнуто із використанням **архітектури трансформер**, в якій застосовується механізм **самоуваги**, що здатен моделювати віддалений контекст без послідовних залежностей.

14.5 Спеціалізовані обчислювальні архітектури для глибоких нейронних мереж

Поточні успіхи глибокого навчання та глибоких нейронних мереж, тісно пов’язані із прогресом у сфері **високопродуктивних** обчислень та **спеціалізованих** обчислювальних архітектур. Дані архітектури, у порівняні із архітектурою графічних прискорювачів *GPU*, мають спеціальні обчислювальні модулі, що забезпечують більшу ефективність та продуктивність обробки даних глибокими нейронними мережами. Їхній паралелізм та архітектурні особливості дозволяють виконувати обчислення більш ефективно, враховуючи зростання складності глибокого навчання, збільшення розміру наборів даних, моделей глибоких нейронних мереж, кількості ознак, класів та категорій об’єктів. При цьому зростає потреба у більшій точності розпізнавання, швидкості навчання і опрацювання, що тісно пов’язана із вимогами більших обчислювальних потужностей, обсягів пам’яті та пропускної здатності пристрою. Протягом останнього десятиріччя обробка даних глибокими нейронними мережами була невід’ємно пов’язана із використанням графічних обчислювальних пристроїв (*GPU)*. Однак нещодавно почала з’являтись велика кількість нових архітектур спеціалізованих інтегральних схем *ASIC*, що включають **архітектури тензорних процесорів** (*tensor cores*) від *NVIDIA* і (*tensor processing units*) від *Google*, які оптимізовані під обчислювальні задачі нейронних мереж.

Для ефективної обробки інформації сучасними глибокими нейронними мережами на даних обчислювальних інфраструктурах потрібно враховувати

аспекти роботи як самих нейронних мереж, так і спеціалізованих обчислювальних архітектур. Задачі глибокого навчання здебільшого вирішуються за рахунок застосування прискорювачів таких, як *GPU*. Сутність цієї архітектури полягає у внутрішній можливості ефективно використовувати паралелізм даних та обчислень. Через це, використання подібної архітектури стало стандартом для задач глибокого навчання. Але враховуючи **постійне збільшення вимог** до обчислювальних потужностей для глибоких нейронних мереж наразі активно починають використовуватись спеціалізовані архітектури **тензорних прискорювачів**.

Рисунок 14.9 — Тензорні прискорювачі *Google Coral*

14.5.1 Тензорні ядра

Головною складовою спеціалізованих обчислювальних архітектур є тензорні ядра. Це апаратний компонент для прискорення математичних операцій із матрицями, який був вперше представлений компанією ***NVIDIA*** в архітектурі *Volta*. Тензорні ядра забезпечують роботу із матрицями розмірності *4 x 4 x 4* і забезпечують операцію $D = A * B + C$, де A, B, C, D є матрицями розмірності *4 x 4*. В комбінації із програмною бібліотекою *TensorRT*, тензорні ядра дозволяють підвищити продуктивність обробки даних глибокими нейронними мережами. Тензорні ядра разом із *TensorRT* дозволяють

використовувати **зменшену розрядність** представлення вагових коефіцієнтів та параметрів мережі. Наприклад, зменшення розрядності представлення до *FP16* дозволяє зменшити загальні витрати пам’яті для зберігання моделі мережі та кількості операцій читання/запису у порівнянні із *FP32*/*FP64*. Це покращує можливості використання великих моделей глибоких мереж на апаратурі з обмеженими ресурсами і також дозволяє пришвидшити арифметичні операції. Додатково, *TensorRT* може автоматично використовувати тензорні ядра архітектури *Volta* у *Tesla V100* для виконання операції прогнозування – *inference*, використовуючи розрядність *FP16*. Як результат, це дозволяє значно зменшити час виконання обчислень. Наприклад, модель *ResNet-50* на *GPU V100 + TensorRT + TCs* (*FP16*) в 4 рази швидше аналогічної моделі (*FP32*).

Також ***Google*** розробила **тензорні процесорні пристрої** (*TPU)*, які дозволяють підвищити ефективність та швидкість обчислень для глибоких нейронних мереж. *TPU* містить *256 x 256* арифметико-логічних пристроїв, які здатні виконувати 65,536 операцій множення та суми над 8-и бітними **цілими числами** при кожному циклі. *TPU* працює на частоті 700 *MHz* і у заявлених тестах дає пришвидшення операцій глибоких нейронних мереж до 30 разів і до 80 разів більшу енергоефективність у порівняні із *CPU* та *GPU*. Це стає можливим за рахунок того, що *TPU* спеціально призначений для виконання операцій глибоких нейронних мереж. Наразі доступні декілька варіацій *TPU* від *Google*. До них входять спеціальні портативні прискорювачі *Coral Edge TPU USB Stick* та *Coral TPU Dev board*, а також хмарні варіанти інфраструктур із прискорювачами: *Cloud TPU v2* (180 *teraflops*, 64 *GB High Bandwidth Memory* (*HBM*)), *Cloud TPU v3* (420 *teraflops*, 128 *GB HBM*), *Cloud TPU v2 Pod Alpha* (11.5 *petaflops*, 4 *TB HBM*, *2-D toroidal mesh network*).

Прискорювачі *Edge TPU* — це спеціалізована інтегральна схема *ASIC*, яка підтримує лише цілочисельні операції із розрядністю 8/16 та працює з моделями ***TensorFlow Lite***.

Компанія *Intel* також випустила декілька модифікацій пристроїв для обробки графічної інформації – *VPU*, що базуються на системі на чипі (*SoC)*

низької потужності, спеціально призначеній для прискорення застосунків **комп’ютерного зору**. Наприклад, *Intel Movidius Myriad X* містить декілька процесорів та обчислювальних блоків, оптимізованих для забезпечення високого паралелізму операцій глибоких нейронних мереж. *VPU* доступні у різних конфігураціях таких, як *USB* версія *Intel Neural Compute Sticks – NCS* із 4 Гб вбудованої *RAM*. Подібний пристрій може підключатись до *USB* порта звичайного комп’ютера і виконувати роль допоміжного процесора для збільшення швидкості обробки даних моделями глибокий нейронних мереж.

СПИСОК ЛІТЕРАТУРИ

Базова:

1. S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Fourth Edition. - Pearson Series in Artifical Intelligence, 2021. URL: https://zoo.cs.yale.edu/classes/cs470/materials/aima2010.pdf

2. Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. - MIT Press book, 2016. URL: https://www.deeplearningbook.org/

3. Shai Shalev-Shwartz and Shai Ben-David. Understanding Machine Learning From Theory To Algorithms. - Cambridge University Press, 2014. URL: https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf

Додаткова:

1. Python code for the book Artificial Intelligence: A Modern Approach. URL: https://github.com/hzhang7/Russel-Norvig

2. SWI-Prolog — Comprehensive Prolog compiler and development environment. URL: https://github.com/SWI-Prolog

3. scikit-learn — Machine Learning in Python. URL: https://scikit-learn.org/stable/index.html

4. Machine Learning Google Developers. URL: https://developers.google.com/machine-learning

5. Deep Learning for Computer Vision: The Abridged Guide. URL: https://www.run.ai/guides/deep-learning-for-computer-vision

6. Revolutionizing the World of Vision AI. URL: https://www.ultralytics.com/

7. YOLO: Real-Time Object Detection Explained. URL: https://www.v7labs.com/blog/yolo-object-detection

8. Image Segmentation: Architectures, Losses, Datasets, and Frameworks. URL: https://neptune.ai/blog/image-segmentation

9. PyTorch — an optimized tensor library for deep learning using GPUs and CPUs. URL: https://pytorch.org/tutorials/

10. TensorFlow Probability - library for probabilistic reasoning and statistical analysis. URL: https://www.tensorflow.org/probability

11. Introduction to OpenCV Object Tracker. URL: https://docs.opencv.org/4.4.0/d2/d0a/tutorial_introduction_to_tracker.html