МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДВНЗ "Ужгородський національний університет"

Факультет інформаційних технологій

Кафедра інформаційних управляючих систем та технологій

В. М. Коцовський

Методи та системи штучного інтелекту

Конспект лекцій

Ужгород – 2023

ЗМІСТ

1. БАЗОВІ ПОНЯТТЯ ШТУЧНОГО ІНТЕЛЕКТУ

1.1. Означення та історія виникнення

Штучний інтелект (ШІ, англ. Artificial intelligence) — *наука та технологія створення інтелектуальних машин, в особливості інтелектуальних комп'ютерних програм.* ШІ пов'язаний з завданням використання комп'ютерів для розуміння людського інтелекту, але не обов'язково обмежується біологічно правдоподібними методами (Джон Маккарті, 1956 р., конференція у Дартмутському університеті). В подальшому було зроблено чимало спроб дати формальне визначення інтелекту взагалі і інтелекту штучного зокрема. Найбільш відомим, очевидно, є визначення предмету теорії штучного інтелекту [1], що було дане видатним дослідником у галузі ШІ М. Мінські і яке у більш або менш видозміненому вигляді потрапило до словників та енциклопедій: "*штучний інтелект є дисципліна, що вивчає можливість створення програм для вирішення задач, які при вирішенні їх людиною потребують певних інтелектуальних зусиль*". Це визначення викликало критику, яка полягала в тому, що під нього можна підвести що завгодно, наприклад, виконання простих арифметичних операцій. Відтак до цього визначення додається поправка: "*сюди не входять задачі, для яких відома процедура їх вирішення*". Таке визначення також важко вважати задовільним.

Рассел та Норвіг [2] наводять класифікацію означень ШІ у табличній формі

Системи, які мислять подібно до людини	Системи, які мислять раціонально
Системи, які діють подібно до людини	Системи, які діють раціонально

Єдиної відповіді на питання чим займається ШІ, не існує. Майже кожен автор дає своє визначення. Зазвичай ці визначення зводяться до *наступних*:

- штучний інтелект вивчає методи розв'язання задач, які потребують людського розуміння. Тут мова іде про те, щоб навчити ШІ розв'язувати тести інтелекту. Це передбачає розвиток способів розв'язання задач за аналогією, методів дедукції та індукції, накопичення базових знань і вміння їх використовувати.
- штучний інтелект вивчає методи розв'язання задач, для яких не існує способів розв'язання або вони не коректні (через обмеження в часі, пам'яті тощо). Завдяки такому визначенню інтелектуальні алгоритми часто використовуються для розв'язання NP-повних задач, наприклад, задачі комівояжера.
- штучний інтелект займається моделюванням людської вищої нервової діяльності.
- штучний інтелект — це системи, які можуть оперувати з знаннями, а найголовніше — навчатися. В першу чергу мова ведеться про те, щоби визнати клас експертних систем (назва походить від того, що вони спроможні замінити «на посту» людей-експертів) інтелектуальними системами.

Останнє визначення, що з'явилося у 1990-х рр., засноване на так званому агентно-орієнтованому підході. Цей підхід акцентує увагу на тих методах і алгоритмах, які допоможуть інтелектуальному агенту виживати в оточуючому середовищі під час виконання свого завдання. В межах цього підходу вивчаються алгоритми пошуку і прийняття рішення.

Дж. Коупленд у праці "Що таке ШІ" відзначає, що незважаючи на наявність численних підходів та визначень як до розуміння ШІ, так і до створення інтелектуальних інформаційних систем, можна виділити два основні **підходи щодо розробки систем ШІ**:

1) *низхідний* (Top-Down AI, семіотичний, символьний) — створення експертних систем, баз знань та систем логічного виведення, що моделюють та імітують високорівневі психічні процеси: мислення, міркування, мова, емоції, творчість і т. п.;
2) *висхідний* (Bottom-Up AI, біологічний, конекціоністський) — вивчення нейронних мереж і еволюційних обчислень, які моделюють інтелектуальну поведінку на основі біологічних елементів, а також створення відповідних обчислювальних систем, таких як нейрокомп'ютери.

Другий підхід, власне кажучи не відноситься до визначення ШІ, яке дав Дж. Маккарті. Їх поєднує тільки кінцева мета.

Відсутність чіткого визначення ШІ не заважає оцінювати *інтелектуальність* на *інтуїтивному* рівні. Можна навести як мінімум два методи такої оцінки:

метод експертних оцінок. Рішення про ступінь інтелектуальності приймає досить велика група експертів (незалежно або у взаємодії між собою);

метод тестування. Існує значна кількість так званих інтелектуальних тестів, апробованих практикою, і ці тести широко використовуються для оцінки рівня розумових здібностей людини, а також у психології та психіатрії.

Для прикладу наведемо декілька типових тестових завдань.

Приклад 1. Вставте пропущене число:

36 30 24 18 _

Приклад 2. Вставте слово, яке означає те саме, що і два слова поза дужками:

дерево (…) підробка

Приклад 3. Викресліть зайве слово:

лев лисиця жираф щука собака

І метод експертних оцінок, і метод тестування мають свої недоліки. Головний з них полягає у тому, що оцінка дається виходячи лише з власних уявлень експертів, авторів тестів і т. п. про те, як має бути.

Серед психіатрів можна почути вислів: "він міркує логічно вірно, але неправильно". Наприклад, дається тестове завдання: "серед слів соловей, чапля, перепілка, стілець, шпак виділити зайве".

Більшість людей, не задумуючись, дає відповідь стілець, тому що всі інші слова — це назви птахів. І раптом хтось дає відповідь шпак, пояснюючи це тим, що це єдине слово, в якому відсутня літера "л".

Обидві класифікації є логічно вірними і формально рівноправними. Але чому ж перевага віддається одній з них? Тому, що так міркує більшість людей. А чому так міркує більшість людей? Очевидно, тому, що *перша класифікація вважається більш важливою для людської практики*. Це положення приймається на аксіоматичному рівні, без доведення.

Необхідно підкреслити, що поняття "штучний інтелект" не можна зводити лише до створення пристроїв, які імітують людину в усій повноті її діяльності. Насправді ж, спеціалісти які працюють в цій області вирішують іншу задачу: виявити механізми, які лежать в основі діяльності людини, щоб застосувати їх при вирішенні конкретних науково-технічних задач**.** І це лише одна з можливих проблем.

1.2. Приклади інтелектуальних задач

До переліку інтелектуальних задач можна віднести [1]:

- розпізнавання образів;
- логічне мислення;
- навчання і самонавчання;
- обробка природної мови (NLP);
- аналіз ситуації;
- розуміння нової інформації;
- планування цілеспрямованих дій;

Більш детально зупинимося на перших чотирьох задачах.

1.2.1. Розпізнавання

На інтуїтивному рівні можна сформулювати декілька типових задач розпізнавання:

- *ідентифікувати* об’єкт, що спостерігається людиною, тобто вирізнити його серед інших (наприклад, побачивши зображення людини, впізнати у ній свою дружину);
- здійснити *розпізнавання* у класичній постановці, тобто віднести об’єкт, що спостерігається людиною, до одного з заздалегідь відомих класів об’єктів (наприклад, відрізнити легковий автомобіль від вантажного);
- провести *кластеризацію* (розбиття множини об’єктів на класи);

Людина робить класифікацію просто. Чоловік, повернувшись додому, відразу ж пізнає свою дитину, собаку і т. д. Але він рідко може *пояснити, як* він це робить. Якби це можна було зробити, алгоритми розпізнавання можна було б легко програмувати і широко застосовувати.

Теорія розпізнавання необхідна для того, щоб навчити вирішувати задачі розпізнавання штучні інтелектуальні системи. Зокрема, сформульовано такий ключовий принцип:

будь-який об’єкт у природі — унікальний; унікальні об’єкти — типізовані.

У відповідності до цього принципу, розпізнавання здійснюється на основі аналізу певних характерних *ознак*. Вважається, що в природі не існує двох об’єктів, для яких співпадають *абсолютно всі* ознаки, і це теоретично дозволяє здійснювати ідентифікацію. Якщо ж для деяких об’єктів співпадають *деякі* ознаки, ці об’єкти теоретично можна об’єднувати в групи, або класи, за цими співпадаючими ознаками. "Близькість" значень ознак дає змогу проводити розбиття множини на *кластери*.

Проблема полягає у тому, що різноманітних ознак дуже багато. Незважаючи на легкість, з якою людина проводить розпізнавання, вона дуже рідко в змозі виділити ознаки, суттєві для цього. До того ж, об’єкти, як правило, змінюються з часом.

Розпізнавання об’єктів і ситуацій має виняткове значення для орієнтації людини в навколишньому світі і для прийняття правильних рішень. Розпізнавання, як правило, здійснюється людиною на інтуїтивному, підсвідомому рівні, людина навчилася цьому за мільйони років еволюції. На цьому фоні спроби навчити розпізнаванню складних об’єктів штучні інтелектуальні системи, навіть шляхом автоматизованого виділення характерних ознак, мають досить слабкі досягнення.

1.2.2. Логічне мислення

Логічне мислення — перехід від початкових положень до їх наслідків за формалізованими законами логіки. І тут пересічна людина рідко в змозі пояснити, за якими алгоритмами вона здійснює логічні побудови. Але методики, за якими можна автоматизувати логічне мислення, досить відомі.

Перш за все, це формальна логіка Аристотеля на основі конструкцій, які отримали назву *силогізмів* (міркувань, які складаються з трьох простих атрибутивних висловлювань: двох засновків і одного висновку). Класичний приклад.

Перше положення. *Усі люди смертні.*
Друге положення. *Сократ — людина.*
Висновок: *Сократ смертний.*

Якщо перше та друге положення у силогізмі істинні та задовольняють певним загальним формальним вимогам, тоді і висновок буде істинним незалежно від змісту тверджень, що входять до силогізму.

Виконання формальних вимог є важливим, інакше легко припуститися логічних помилок, подібних до таких:

Всі студенти вузу А знають англійську мову. Петро знає англійську мову. Отже, Петро — студент вузу А.

Аристотелем було запропоновано декілька формальних конструкцій силогізмів, які він вважав достатньо універсальними. У XIX столітті почала розвиватися сучасна математична логіка, яка розглядає силогізми Аристотеля як один із часткових випадків.

Основою більшості сучасних систем, призначених для автоматизації логічних побудов, є *метод резолюцій* Робінсона. Але практична автоматизація логічного мислення зіткнулася з двома серйозними проблемами.

Перша з них — феномен, який Р. Беллман називав *прокляттям розмірності*. Зовнішній світ являє собою винятково складне переплетіння різноманітних об'єктів та зв'язків між ними. Для того, щоб тільки ввести всю цю інформацію до пам'яті інтелектуального комп'ютера, може знадобитися не одна тисяча років.

Інша проблема — *алгоритмічна нерозв'язність*. Відомо, що в рамках будь-якої досить складної формальної теорії існують положення, які є істинними, але які не можна ні довести, ні спростувати (теорема Геделя про нерозв'язність формальної арифметики).

Реальні програми, які здійснюють логічне виведення (вони часто називаються *експертними системами*) мають досить обмежене застосування. Вони мають обмежений набір фактів та правил з певної, більш-менш чітко окресленої предметної галузі і можуть використовуватися лише у цій галузі.

Що ж стосується людини, то якість її логічного мислення часто буває далекою від бездоганної. Люди рідко проводять логічні побудови до кінця, часто роблять логічні помилки, а інколи взагалі керуються принципами, невірними з точки зору формальної логіки, а рішення приймається на підсвідомому, інтуїтивному рівні. Зрозуміло, що таке рішення може бути помилковим. Але, якби це було не так, людина була б практично не здатною ні до якої діяльності — ні до фізичної, ні до продуктивної розумової.

Для задач, які розглядалися вище, характерною була спільна риса: їх *погана формалізованість*, відсутність або невизначеність чітких алгоритмів вирішення. Саме такі задачі і являють собою основний предмет розгляду в теорії штучного інтелекту.

До зовсім іншого класу відносяться *обчислювальні задачі*. Важко відповісти на запитання, як саме людина здійснює ті чи інші обчислення. Добре відомими є і низька швидкість, і невисока надійність цього виду людської діяльності. Але були запропоновані ефективні принципи комп'ютерних обчислень.

1.2.3. Машине навчання

Машинне навчання (*machine learning*) — це підгалузь штучного інтелекту, яка часто застосовує статистичні прийоми для надання комп'ютерам здатності «навчатися» (тобто, поступово покращувати вміння розв'язувати певну задачу).

Назву було запропоновано 1959 року А. Семюелем [2]. Машинне навчання досліджує *вивчення та побудову алгоритмів, які можуть навчатися й робити передбачення та узагальнення на основі наявних даних* [3]. До прикладів застосувань machine learning належать фільтрування електронної пошти, виявляння мережевих атак, оптичне розпізнавання символів (ОРС), навчання ранжуванню та комп'ютерний зір.

Задачі машинного навчання, як правило, поділяють на дві широкі категорії, залежно від того, чи доступний системі, що навчається, навчальний «сигнал», або «зворотний зв'язок»:

- *Навчання з учителем* (supervised learning): комп'ютерові надають приклади входів та їхніх бажаних виходів, задані «вчителем», і метою є встановлення загального правила, яке відображає входи на виходи.

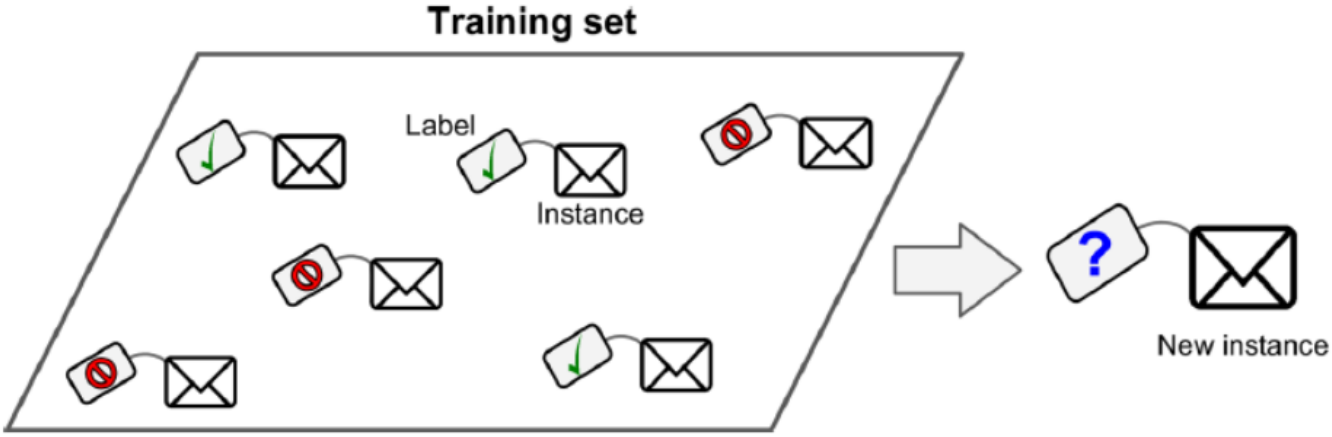


Рис. 1.1. Помічена навчальна вибірка для класифікації електронних листів

В окремих випадках вхідний сигнал може бути доступним лише частково, або бути обмеженим особливим зворотним зв'язком:

- *Напівавтоматичне навчання* (semi-supervised learning): системі надають тренувальний набір, в якому відсутні деякі цільові виходи.
- *Навчання з підкріпленням* (reinforcement learning): тренувальні дані (у вигляді винагород та покарань) надаються лише як зворотний зв'язок на дії програми в динамічному середовищі, як при керуванні автомобілем, або грі з опонентом.

Навчання з підкріпленням застосовується у робототехніці.

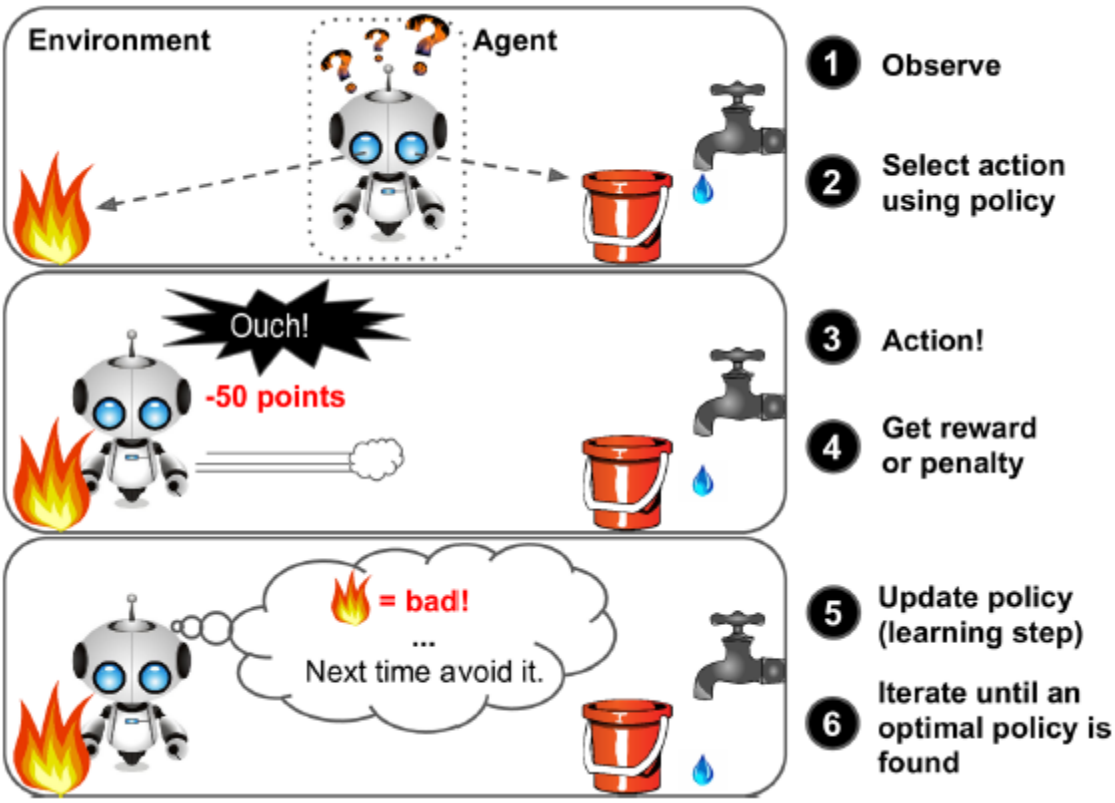


Рис. 1.2. Навчання з підкріпленням

- *Навчання без учителя* (unsupervised learning): Алгоритмові навчання не дається міток, залишаючи його самому знаходити приховані закономірності в наборах даних.

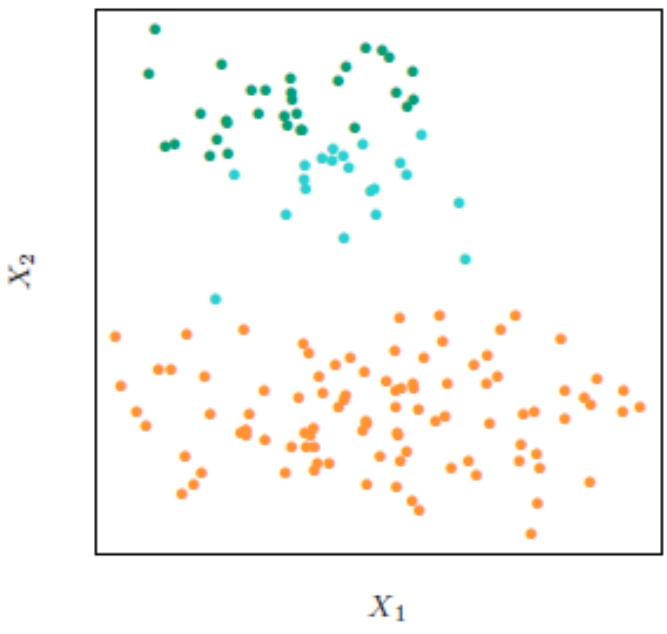


Рис. 1.3. Ілюстрація навчання без учителя (кластеризація)

Виокремлюють два основні типи задач навчання з учителем: задачі класифікації та регресії. Про задачу класифікації мова йшла у пункті 1.2.1. Задача регресії полягає у навчанні системи робити передбачення значення вихідного результуючого значення

(value) за значеннями вхідних показників (input features), використовуючи у процесі навчання знання про значення вхідних та вихідних показників елементів навчальної множини. Зазвичай, є кілька вхідних показників та один вихідний (може бути і кілька). Задача регресії у випадку одного вхідного показника проілюстрована на рис. 1.4.

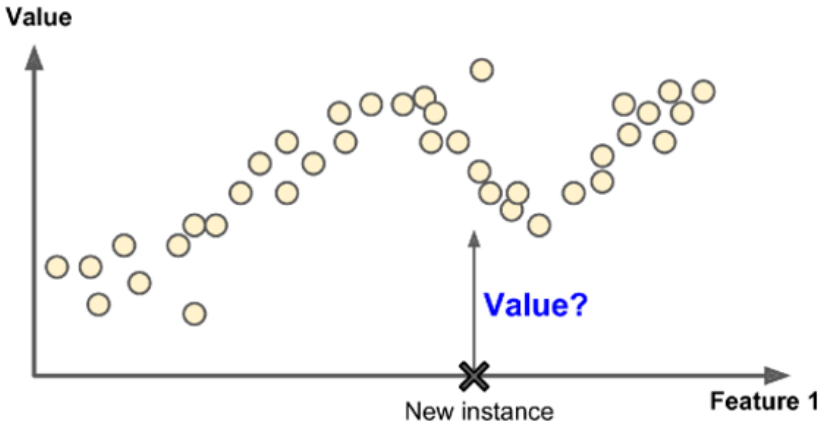


Рис. 1.4. Задача регресії

1.2.4. Обробка природної мови

Обробка природної мови (*Natural-language processing*, NLP) — напрямок у ШІ, який присвячений проблемам *комп'ютерного аналізу* та *синтезу природної мови*. Основна увага звертається розробці методів та програмних продуктів для обробки та аналізу великих обсягів інформації у форматі людської мови. Стосовно штучного інтелекту аналіз означає розуміння мови, а синтез — генерацію розумного тексту.

Розуміння природної мови вважають складною задачею ШІ, тому що розпізнавання живої мови потребує значних знань системи про навколишнє середовище та можливості взаємодіяти з ним. В наш час значну роль у вирішенні задач з обробки природномовних даних відіграють *онтології*.

Онтологія — формалізоване подання знань про певну предметну область (середовище, світ), придатне для автоматизованої обробки. Онтологію неодмінно супроводжує деяка концепція цієї області. Онтології зазвичай містять класи (поняття), екземпляри цих класів, їхні атрибути (властивості) та значення цих властивостей, а також відношення між класами та екземплярами класів. Крім того, онтологія може містити певні обмеження на використання класів та їх відношень. Визначення цих об'єктів і відношень між ними зазвичай називають *концептуалізацією*.

Для опису онтологій використовується мова OWL (*Web Ontology Language*).

Основні задачі NLP

1. *Розпізнавання мови*: виведення/розпізнавання тексту з зображень, відсканованих документів або файлів у PDF форматі. Сюди ж входить *розпізнавання мовлення*, продуковане людським голосом.
2. *Синтез мовлення*: озвучення/прочитання тексту (документ, повідомлення і т. д.) голосом, який є наближеним до природного.
3. *Машинний переклад*: автоматичний переклад з однієї людської мови на іншу. Задача є складною, адже машина не володіє тими знаннями, якими володіє людина, що робить їх «розуміння» тих чи інших фраз абсолютно різним.

4. *Питально-відповідальні системи*: відповіді на питання, поставлені людською мовою. Зазвичай питання є конкретизованими, наприклад, "Де знаходиться УжНУ?", проте існують питання, на які немає конкретної відповіді, наприклад, "Чому всі люди різні?", що робить дане завдання надзвичайно складним для виконання.
5. *Розпізнавання/визначення тематики*: поділ тексту на частини з подальшим визначенням провідної теми для кожної з них.
6. *Добування даних*: отримання семантичної інформації з тексту.
7. *Спрощення тексту*: зміна, розширення або інша обробка інформації для спрощення структури або граматики тексту зі збереженням основної думки.
8. *Розв'язання задачі лексичної багатоманітності*: надання списку можливих значень конкретного багатозначного слова, серед яких можна вибрати найбільш підходяще відповідно до контексту.

Основні підходи до NLP

1. *Символічний підхід* — здійснює глибинний аналіз лінгвістичних явищ та базується на явному представленні знань, що здійснюється шляхом використання добре досліджених схем представлення знань та алгоритмів, що працюють з ними. Джерелом знання про мову можуть виступати словники, формули та правила, розроблені людьми.
2. *Лінгвістичний* — складається з чотирьох рівнів: графематичного, морфологічного, синтаксичного та семантичного. Перший рівень полягає у виділенні окремих елементів тексту, наприклад, розділів, абзаців, речень і т. д. Другий рівень полягає у визначенні морфологічних характеристик окремого слова. Третій рівень відповідає за визначення синтаксичної залежності слів у реченнях. Останній рівень пов'язаний зі смисловим розумінням тексту, що включає розробки у сфері штучного інтелекту.
3. *Статистичний підхід* — в основі лежить припущення, що зміст тексту може бути визначено за найуживанішими словами. Основним завданням даного підходу є визначення кількості повторень конкретного слова в тексті. Основна проблема, з якою стикаються статистичні підходи, полягає в розгляді тексту як набору слів без смислового зв'язку.
4. *Нейромережевий* підхід — полягає у поєднанні статистичних знань та різних моделей нейромереж та методів їх навчання.

1.3. Огляд популярних систем ШІ

1. **Google Brain** — дослідницький проект Google по розробці ШІ на основі глибинного навчання (Deep Learning). В ньому поєднуються відкриті дослідження в галузі машинного навчання з розробкою систем та використанням обчислювальних можливостей в масштабах Google.

У жовтні 2016 р. Google Brain провів експеримент щодо шифрування повідомлень. У ньому два набори ШІ розробили свої власні криптографічні алгоритми для захисту своїх повідомлень від іншого ШІ, який в свою чергу спрямовані на розвиток власної системи для злому шифрування, створеного ШІ. Дослідження виявилося успішним, оскільки два первинних ШІ змогли з нуля навчитися спілкуванню один з одним.

В даному експерименті були створені три ШІ: Аліса, Боб і Єва. Мета експерименту полягала в тому, щоб Аліса послала повідомлення Бобу, який зможе його розшифровувати, а Єва спробувала б перехопити дане повідомлення. При цьому ШІ не давалося чіткі інструкції про те, як шифрувати їх повідомлення. Їм була надана тільки

функція втрат. Наслідком цього стало те, що якщо під час експерименту спілкування між Алісою і Бобом не увінчалися успіхом (повідомлення Аліси було неправильно витлумачено Бобом або перехоплено Євою), то в наступних раундах криптографічне правило змінюється таким чином, щоб Аліса і Боб змогли безпечно спілкуватися. Дослідження дозволило зробити висновок про те, що ШІ може розробити власну систему шифрування без заздалегідь прописаних алгоритмів шифрування, що може стати проривом в області шифрування повідомлень в майбутньому.

У лютому 2017 р. система Google Brain анонсувала систему поліпшення зображення, що використовує нейронні мережі для заповнення деталей зображень з дуже низькою роздільною здатністю. Зображення з роздільною здатністю 8 x 8 перетворюються в зображення з роздільною здатністю 32 x 32.

Було отримано прорив в покращенні зображень з низькою роздільною здатністю. Коли людям показували поліпшене зображення і справжнє зображення, то в 10% випадків для фотографій знаменитостей і в 28% випадків для фотографій спалень вони не могли правильно сказати, де справжнє, а де масштабоване зображення. До цього «звичайне» бікубічне масштабування завжди правильно визначалося людиною.

У вересні 2016 р. команда запустила нову систему — нейронний машинний переклад Google (GNMT), яка являє собою наскрізну систему навчання, здатну вчитися на великій кількості прикладів. Хоча її впровадження значно підвищило якість Перекладача Google для пілотних мов, було дуже складно створити такі поліпшення для всіх 103 підтримуваних мов. Для вирішення даного завдання команда Google Brain змогла розробити багатомовну версію GNMT, яка розширила попередню і дозволила здійснювати переклад між декількома мовами. Більш того, вона дозволила виконувати прямий переклад між мовними парами, які явно не задавалися при навчанні. Нещодавно Google анонсувала, що Перекладач Google може здійснювати переклад за допомогою нейронних мереж без розшифровки тексту. Це означає, що можна перевести запис, записаний на одній мові, в текст на іншій мові без попереднього перетворення мови в текст. Щоб навчити цьому систему, їй подали на вхід фрагменти записів іспанською мови з розшифровкою англійською мовою. Різні шари нейронних мереж, які імітують людський мозок, змогли об’єднати відповідні фрагменти і послідовно перетворити звукову хвилю в англійський текст.

В даний час технологія проекту використовується в системі розпізнавання мови в операційній системі Android, пошуку по фотографіях в Google+ і рекомендаціях відео в YouTube.

У роботі над Google Brain використовується TensorFlow — відкрита програмна бібліотека для машинного навчання, розроблена компанією Google для вирішення задачі конструювання і тренування нейронної мережі з метою автоматичного знаходження та класифікації образів, досягаючи якості людського сприйняття. Застосовується як для досліджень, так і для розробки власних продуктів Google. Основне API для роботи з бібліотекою реалізовано для Python, також існують реалізації для C++, Haskell, Java, Go і Swift.

Серед додатків, для яких TensorFlow є основою — програмне забезпечення автоматизованої анотації зображень DeepDream. У 2015 р. Google на базі TensorFlow реалізувала RankBrain — систему, що самонавчається і використовується пошуковиком Google для забезпечення більш релевантних результатів пошуку для користувачів. На сьогодні RankBrain є третім найбільш важливим фактором в алгоритмі ранжування корпорації Google нарівні з посиланнями і контентом.

RankBrain інтерпретує запити користувачів і надає найбільш релевантні сторінки, які можуть і не містити саме ті слова, які включені в пошуковий запит, але мають аналогічний сенс. В режимі офлайн RankBrain отримує дані про минулі пошукові запити і, аналізуючи їх, дізнається, як налаштувати результати пошуку. Після того як результати RankBrain перевіряються командою Google, система оновлюється і працює в режимі реального часу.

2. **IBM Watson** — суперкомп'ютер фірми IBM, оснащений системою штучного інтелекту. Основне завдання Уотсона — розуміти питання, сформульовані на природній мові, і знаходити на них відповіді в базі даних з використанням алгоритмів ймовірнісного пошуку. Він названий на честь першого президента IBM Томаса Уотсона.

Watson складається з 90 серверів IBM p750, кожен з яких оснащений чотирма восьмиядерними процесорами архітектури POWER7. Сумарний обсяг оперативної пам'яті — більше 15 терабайт.

Система мала доступ до 200 млн сторінок структурованої і неструктурованої інформації об'ємом в 4 терабайта, включаючи повний текст Вікіпедії.

У 2014 році IBM оголосила про інвестування 1 млрд доларів в розвиток проекту Watson, і про створення нового підрозділу когнітивних обчислень Watson Business Group, в завдання якого входить розробка і комерціалізація хмарних сервісів в таких областях як охорона здоров'я, фінанси, подорожі, телекомунікації та роздрібна торгівля.

Продовжуючи успішно розвивати проект IBM Watson, до 2018 року компанія випустила програмні продукти: Watson Studio — для побудови моделей машинного навчання та Watson SDK — для доступу до інтернет-сервісів IBM Watson, які доступні для операційних систем Linux, macOS і Windows.

Для демонстрації роботи Watson прийняв участь в американській вікторині «Jeopardy!», в якій учасники відповідають на питання з області загальних знань: кожне питання — твердження про якесь явище або істоту. Гравець повинен дати свою відповідь у формі запитання. Наприклад, при виборі категорії «Президенти» і питання вартістю 200 доларів ведучий читає твердження "Цей "батько-засновник" насправді не зрубав вишню», а гравець для отримання суми в 200 доларів повинен буде відповісти «Хто такий Джордж Вашингтон?". У гравця є 5 секунд для відповіді на питання: якщо він дасть правильну відповідь, то він переходить до наступного питання. Суперниками Watson у двох іграх був володар найбільшого виграшу в програмі і рекордсмен по тривалості безпрограшної серії. Під час гри Watson не мав доступу до Інтернету. Комп'ютер здобув перемогу, отримавши 1 млн доларів (системі вдалося виграти в обох іграх).

Приклад ще одного використання проекту Watson — система генерації питань та аналізу відповідей для діагностики онкологічних захворювань.

3. **Софія** (Sophia) — людиноподібний робот, розроблений компанією Hanson Robotics. Софія має СШІ, обладнана функціями обробки візуальної інформації і технологією розпізнавання осіб. Софія може імітувати людські жести і вирази обличчя, а також може відповідати на певні питання і проводити прості бесіди на певні теми (наприклад, про погоду). Всього Софія може імітувати 62 емоції.

Рис. 1.5. Софія у 2018 році

Робот використовує технологію розпізнавання мови від Google і вдосконалюється з часом, стаючи розумніший. Програмне забезпечення штучного інтелекту Софії розроблено компанією SingularityNET. Воно аналізує проведені розмови і на підставі нових даних покращує відповіді в майбутньому.

У жовтні 2017 р. Софія стала громадянкою Саудівської Аравії.

4. **Siri** — *персональний помічник* і запитально-відповідальна система, адаптована під iOS. Цей додаток спілкується природною мовою, щоб відповідати на питання і давати рекомендації. Siri пристосовується до кожного користувача індивідуально, вивчаючи його особливості протягом тривалого часу. Siri може:

- Відкривати програми.
- Писати в Twitter, Facebook.
- Рекомендувати ресторани, фільми, а також бронювати квитки і столики.
- Надавати інформації про спортивні ігри (рахунок, біографія, склад, матчі).
- Показувати маршрути на картах.
- Використовувати додаток Apple Store.
- Знаходити інформацію у Вікіпедії та браузері, розказати прогноз погоди, керувати «розумним будинком».

Аналогами Siri є Google Assistant, Amazon Alexa, Bixby тощо.

5. **ChatGPT** (Generative Pretrained Transformer) — чат-бот зі штучним інтелектом від OpenAI. Серед численних можливостей ChatGPT варто відзначити вміння писати програми на багатьох мовах програмування, створювати музику, сценарії, казки та письмові роботи, відповідати на тестові питання (часто краще, ніж середня людина), генерувати бізнес-ідеї та писати вірші.

Чат-бот заснований на використанні LLM (Large Language Model), розробленій лабораторією OpenAI у 2022 році, та написаний на Python (із залученням інших мов та технологій). Основою для розробки LLM-моделі ChatGPT є архітектура Transformer — глибинна модель, запропонована командою Google Brain у 2017 році. У моделі ChatGPT використовується біля 100 млрд параметрів.

ChatGPT отриманий шляхом доопрацювання моделі GPT-3.5, використовуючи техніки *керованого* машинного навчання і *машинного навчання з підкріпленням*. Обидві техніки передбачали участь людей, які тренували модель, щоб покращити її продуктивність. У випадку керованого навчання, модель тренувалася на діалогах, де людина-тренер виконувала дві ролі: користувача і асистента ШІ. На етапі підкріплення тренер оцінював відповіді, що давала модель у попередньому діалозі. Ці оцінки були використані для створення «моделей винагород», на яких модель була допрацьована шляхом проходження декількох ітерацій *Proximal Policy Optimization* (PPO). Моделі були навчені у співпраці з Microsoft на їхній хмарній інфраструктурі Azure.

1.4. Області застосування ШІ

Фінанси:

- *Алгоритмічна торгівля* — використання складних систем штучного інтелекту для прийняття торгових рішень зі швидкістю, що перевищує швидкість на яку здатний людський організм. Це дозволяє робити мільйони угод в день без будь-якого втручання людини. Автоматизовані торгівельні системи зазвичай використовуються великими інституційними інвесторами.
- *Дослідження ринку та інтелектуальний аналіз даних*.
  Кілька великих фінансових установ вклали кошти в розвиток ШІ, щоб використовувати його в інвестиційній практиці. Банки, такі як UBS і Deutsche Bank, використовують систему ШІ під назвою Sqreem, яка може обробляти дані для розробки профілів споживачів і зіставляти їх з продуктами, які вони, швидше за все захочуть придбати. Goldman Sachs використовує Kensho, платформу аналітики ринку, яка об’єднує статистичні обчислення з великими даними і обробкою природної мови. Його системи машинного навчання використовують дані в Інтернеті і оцінюють кореляції між світовими подіями і їх впливом на ціни фінансових активів. Інформація, витягнута системою ШІ з прямої трансляції новин, використовується в прийнятті інвестиційних рішень.
- *Андерайтинг* —процедура оцінки банком ймовірності погашення або непогашення кредиту, що запитується.
  Онлайн-кредитор Upstart аналізує величезна кількість споживчих даних і використовує алгоритми машинного навчання для побудови моделей кредитного ризику, які прогнозують ймовірність дефолту. Їх технологія буде ліцензована для банків, щоб вони могли використовувати її для оцінки своїх процесів.

Важка промисловість

Роботи стали поширені в багатьох галузях промисловості і часто займаються роботою, яка вважається небезпечною для людей. Вони виявилися ефективними на робочих місцях, пов’язаних з повторюваними рутинними завданнями, які можуть призвести до помилок або нещасних випадків.

Медицина

Штучні нейронні мережі, такі як технологія Concept Processing в програмному забезпеченні EMR, використовуються в якості систем прийняття рішень для медичної діагностики. Інші застосування:

- Комп'ютерна інтерпретація медичних зображень. Такі системи допомагають сканувати цифрові зображення (наприклад від комп'ютерного томографа), для виділення помітних відхилень, які сигналізують про можливі захворювання. Типовим застосуванням є виявлення пухлини.
- Аналіз серцевого ритму.
- Роботи-помічники для догляду за людьми похилого віку.
- Обробка медичних записів для надання більш корисної інформації.
- Створення планів лікування.
- Допомога в повторюваних завданнях, включаючи управління прийомом медикаментів, надання консультацій.
- Створення ліків.

В даний час в галузі охорони здоров'я працює понад 100 стартапів, заснованих на застосуванні ШІ.

Управління людськими ресурсами та рекрутинг

Інше застосування ШІ полягає в управлінні людськими ресурсами та рекрутингу. Існує три способи використання ШІ для управління людськими ресурсами та найму фахівців. ШІ використовується для перегляду резюме і ранжування кандидатів відповідно до їх рівнем кваліфікації. ШІ також використовується для прогнозування успіху кандидата в заданих ролях через платформи зіставлення посад. І нарешті, ШІ використовується при створенні чат-ботів, які можуть автоматизувати повторювані комунікаційні завдання.

Онлайн служби підтримки клієнтів

ШІ реалізується в автоматизованих онлайн-помічників, які можна розглядати як чат-боти на веб-сторінках. Це може допомогти підприємствам знизити витрати на найм і навчання співробітників. Основною технологією для таких систем є природна обробка мови. Google аналізує людську мову і перетворює їх в текст. Платформа може ідентифікувати сердитих клієнтів через особливості їх мови і реагувати відповідним чином.

Транспорт

Для автоматичних коробок передач в автомобілях були розроблені контролери нечіткої логіки. Автомобілі мають допоміжні функції, засновані на ШІ, такі як засоби круїз-контролю. ШІ використовується для оптимізації додатків управління дорожнім трафіком, що, в свою чергу, скорочує час очікування, споживання енергії і шкідливі викиди на 25%. В майбутньому будуть розроблені повністю автономні автомобілі. Очікується, що ШІ на транспорті забезпечить безпечне, ефективне і надійне транспортування, мінімізуючи згубний вплив на навколишнє середовище і суспільство.

Інші галузі застосування

Різні засоби ШІ також широко використовуються в області забезпечення безпеки, розпізнаванні мови і тексту, інтелектуального аналізу даних і фільтрації спаму в електронній пошті. Також розробляються програми для розпізнавання жестів (розуміння мови жестів машинами), індивідуальне розпізнавання голосу, глобальне розпізнавання голосу (для множини людей в галасливій кімнаті), розпізнавання особи для інтерпретації емоцій.

1.5. Сильний та слабкий штучний інтелект

Сильний ШІ (strong AI, artificial general intelligence) — здатність інтелектуальної машини чи агента зрозуміти та розв'язати будь-яку інтелектуальну задачу, з якою може впоратися людина.

На відміну від сильного ШІ, *слабкий ШІ* не зобов'язаний мати повний спектр людських розумових здібностей. Він полягає у використанні програмного забезпечення для розв'язування певних специфічних інтелектуальних задач. Прикладом слабкого ШІ є експертна система.

Термін «Сильний штучний інтелект» ввів *Джон Серль*.

Наступні тести були запропоновані для перевірки того, що ШІ є «сильним»:

1. *Тест Тюрінга*.
2. Кавовий тест (тест Возняка):
   ШІ може вважатися сильним, якщо машина під його керуванням здатна потрапити до пересічного американського будинку, знайти кавоварку, зварити в ній каву та налити її в горнятко.
3. Тест Герцеля:
   машина під керуванням ШІ здатна пройти той самий курс навчання, що й пересічна людина, та здобути освітній ступінь.
4. Тест Нільсона:
   здатність виконувати економічно важливу роботу не гірше за людину.

1.5.1. Тест Тюрінга

Праця А. Тюрінга "Обчислювальні машини та інтелект" з'явилася у 1950 р. і була однією з перших публікацій з даної тематики. Тюрінг розглянув питання про те, чи можна примусити машину думати по-справжньому. Відзначивши, що існує певна невизначеність у самому питанні (що таке "машина" і що значить "думати"), яка виключає можливість раціональної відповіді, Тюрінг запропонував замінити питання про інтелект більш чітким емпіричним тестом.

Тест Тюрінга порівнює здібності гадано "розумної" машини із здібностями людини — єдиним і найкращим стандартом розумної поведінки. В цьому тесті машину та слідчого (або експерта) поміщають у окремі кімнати. Ще у одній кімнаті знаходиться "людина". Експерт не може бачити машину чи людину і може спілкуватися з ними лише за допомогою текстового пристрою, наприклад комп'ютерного термінала (див. рис. 1.6).

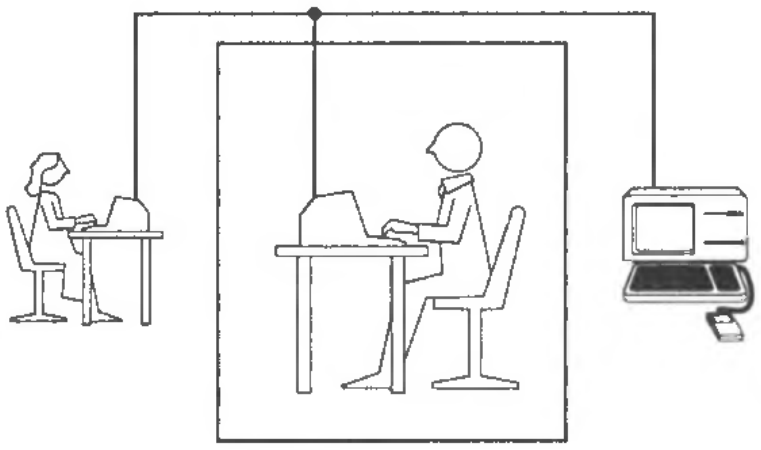


Рис. 1.6. Тест Тюрінга

Експерт повинен відрізнити комп'ютер від людини виключно за допомогою їх відповідей на запитання, які він їм задає. Якщо експерт не може відрізнити комп'ютер від людини, тоді, як стверджує Тюрінг, машину потрібно визнати розумною.

Особливості тесту Тюрінга [3]:

1. Тест дає об'єктивне уявлення про інтелект, тобто реакцію розумної людини на певний набір запитань.
2. Виключаються якісь особливі вимоги про внутрішню будову комп'ютера.
3. Виключає можливість упередженого ставлення на користь живих істот;

Одним із різновидів тесту Тюрінга є CAPTCHA (completely automated public Turing test to tell computers and humans apart).

Методики, засновані на тесті Тюрінга стали незамінними при розробці та перевірці сучасних експертних систем.

Багато спеціалістів вважали, що тест Тюрінга є цілком задовільним для визначення рівня інтелектуальності комп'ютерної системи. Але виявилось, що це не зовсім так.

В основі тесту Тюрінга лежить неявне припущення про те, що необхідною умовою ведення діалогу є розуміння співрозмовника. Але у кінці 60-х рр. американський кібернетик Дж. Вейценбаум створив дві програми — ЕЛІЗА і ЛІКАР. Вони були створені для використання в психіатрії, але виявилось, що за їх допомогою можна обманути експерта в умовах тесту Тюрінга. В основі лежить ідея *фатичного діалогу* [1].

1.5.2. Фатичний діалог

Фатичним діалогом називається діалог *без розуміння співрозмовника*, на основі формального перефразування почутого.

Так, наприклад, на висловлювання пацієнта "*Ви не відверті зі мною*" програма може відповісти "*Чому Ви думаєте, що я не відверта з Вами?*" У ряді випадків програма може видати заздалегідь заготовлену фразу. Так, на слова пацієнта "*Мій батько мене не любив*" вона може відповісти "*Розкажіть мені про Вашу сім'ю*".

Звичайно, це не має ніякого відношення ні до розуміння, ні до інтелекту, але багато експертів, які проводили з цими програмами тест Тюрінга, вирішили, що мають справу з людиною. Системи, які здатні тільки на підтримку фатичного діалогу, не можна вважати інтелектуальними. *Розуміння* співрозмовника є абсолютно необхідним для нормального діалогу, а для такого розуміння *необхідно мати певну суму знань про світ.*

Поява систем, що реалізують фатичний діалог, завдала сильного удару по визначенню Тюрінга. Безумовно, воно зберігає своє методологічне значення, але вже не може претендувати на універсальність (так само, як і будь-яке інше з відомих визначень).

Нижче у дещо спрощеному вигляді наведено фрагмент діалогу, характерного для програми ЛІКАР:

ЛІКАР: *Я Вас слухаю.*
ЛЮДИНА: *Мене бентежить погода.*
ЛІКАР: *Як давно Вас бентежить погода?*
ЛЮДИНА: *Відтоді, як померла моя мати.*
ЛІКАР: *Розкажіть про Вашу сім'ю.*
ЛЮДИНА: *Всі вони захоплюються комп'ютерами.*
ЛІКАР: *Машини Вас лякають.*

ЛЮДИНА: *Ні.*

ЛІКАР: *Будь ласка, не будьте такі небагатослівні…*

І так далі до нескінченності.

У [1] наводиться інший приклад фатичного діалогу. Віце-президент американської компанії захотів зв'язатись з системним програмістом Д. Бобровим. Відбулась така розмова:

Віце-президент : *Чи можу я зараз скористатись комп'ютером?*

Бобров: *А чому ви питаєте?*

Віце-президент: *Може, мені вдасться підписати контракт.*

Бобров: *Чому ви в цьому впевнені?*

Віце-президент: *Мої потенційні покупці ніколи не бачили систему в дії.*

Бобров: *Це має для вас велике значення?*

Віце-президент: *Звичайно.*

Бобров: *Ви маєте в цьому сумнів?*

Віце-президент: *Що за безглузде зауваження?*

Бобров: *Що Ви маєте на увазі, кажучи про безглузде зауваження?*

Віце-президент: *Чи можу я зараз скористатись комп'ютером?*

Бобров: *Розкажіть мені ще про Ваших потенційних покупців.*

Віце-президент: *Передзвоніть мені 491-1850.*

Віце-президент був, м'яко кажучи, здивований, тим більше, що дзвінка не було. Непорозуміння з'ясувалося пізніше. Виявилось, що телетайп з'єднав віце-президента не з самим Бобровим, а з програмою, яка реалізувала фатичний діалог.

Класичний метод написання програм, що реалізують фатичний діалог — застосування *зіставлення зі зразком*. В основі методу лежить *зіставлення* речень, які вводяться людиною, зі *зразками*, що зберігаються програмою. В залежності від результату зіставлення реалізується та чи інша заздалегідь запрограмована операція. І самих зразків, і операцій, що їм відповідають, як правило, порівняно небагато.

Можна стверджувати, що зіставлення зі зразком є однією з найбільш фундаментальних інтелектуальних процедур. Спробуємо формалізувати це поняття.

Визначення. Нехай M — набір зразків, що зберігаються в пам'яті інтелектуальної системи, F — деяка множина функцій або алгоритмів, і $r(a, b, f)$ — деякий критерій якості. Тоді під співставленням елемента q, що спостерігається інтелектуальною системою, зі зразком будемо розуміти знаходження елемента $m \in M$ і функції $f \in F$, таких, що $f(q) = m$ або $f(m) = q$.

Природно також накласти одну з двох умов:

- максимізація критерію якості: $r(m, q, f) \to \max$;
- досягнення задовільного рівня якості: $r(m, q, f) > \varepsilon$, де ε — заданий поріг.

Ця постановка задачі співставлення є дуже загальною. Її конкретизації залежать від того, які обмеження накладаються на множини M і F та на критерій якості. Можна вважати, *що більш інтелектуальна система повинна вміти здійснювати більш складне співставлення*.

Можна розглянути кілька варіантів зіставлень, кожний з яких може бути легко запрограмований.

Варіант 1 (повний збіг). Якщо речення, що вводиться, повністю збігається з одним із зразків, може прозвучати відповідь: *"Так, Ви маєте рацію"***,** або навпаки: *"Ви помиляєтесь, оскільки..."*, і після *"рацію"* або *"оскільки"* програміст може написати будь-який текст, що імітує глибоке розуміння специфіки предметної області. Наприклад, дуже непогано виглядатиме такий діалог:

Людина**:** *Квадрат гіпотенузи дорівнює сумі квадратів катетів.*

Програма: *Так, але є подібний результат і для непрямокутних трикутників — це теорема косинусів.*

Навряд чи після кількох подібних відповідей у когось залишаться сумніви в інтелектуальних здібностях програми. Але, звичайно, повні збіги трапляються дуже рідко. Тому доводиться використовувати інші типи порівнянь.

Варіант 2 (використання замінювачів). Типовим є використання замінювачів «*» і «?». Із замінювачем * зіставляється довільний фрагмент тексту, із замінювачем ? — будь-яке окреме слово.

Наприклад, шаблон *(*комп'ютери*)* успішно зіставляється з будь-яким реченням, в якому згадується про комп'ютери; шаблон *(Я люблю ? яблука)* — з такими реченнями, як *(Я люблю червоні яблука), (Я люблю солодкі яблука)* тощо (але не з реченням *Я люблю їсти зелені яблука)*.

Варіант 3 (надання значень змінним у процесі зіставлення). При цьому можливості програми, що реалізує фатичний діалог, значно розширюються. Вона набуває здатності до генерації відповідей, які залежать від запитань. Так, правило

$$(Я\ ?\ *) \overset{a:=?}{\rightarrow} (Що\ Ви\ ще\ <a>?)$$

дозволяє на речення *"Я люблю яблука"* відповісти *"Що Ви ще любите?",* а на речення "*Я ненавиджу дощі" — "Що Ви ще ненавидите?"*. У цьому прикладі при успішному зіставленні змінній *a* надається значення слова, з яким збігається замінювач "?". Безумовно, при використанні українських фраз потрібно стежити за узгодженням суфіксів і закінчень.

Варіант 4 (універсальний зразок). Зі зразком *(*)* зіставляється будь-яке речення. Звичайно, і відповіді, що відповідають цьому зразкові, повинні бути такими ж універсальними. Наприклад:

(Я Вас не дуже розумію)
(Не будьте такими небагатослівними)
(Чому це має для Вас значення?)

і т. п.

Варіант 5 (зіставлення більше ніж з одним зразком). Речення може зіставлятися не з одним зразком, а кількома. Наприклад, якщо в програмі задані підстановки

(люблю *) —> (Що Ви ще любите?)*

та

(комп'ютери *) —> (Ви маєте здібності до техніки),*

то речення *(Я люблю комп'ютери)* зіставляється з обома зразками.

1.5.3. Китайська кімната*

Китайська кімната — уявний експеримент, описаний Джоном Серлем, в якому критикується можливість комп'ютерного моделювання людського розуміння природної

мови, створення так званого *«сильного штучного інтелекту»*. Зокрема цей експеримент є критикою тесту Тюрінга. Був запропонований в 1980 р. в статті «Minds, Brains, and Programs». Серль описав експеримент таким чином [2]:

«Візьмемо, наприклад, яку-небудь мову, якої ви не розумієте. Для мене такою мовою є китайська. Тепер припустимо, що мене помістили в кімнату, в якій розставлені кошики, повні китайських ієрогліфів, та дали підручник англійською мовою, в якому наводяться правила поєднання символів китайської мови, причому правила ці можна застосовувати, знаючи лише форму символів, розуміти значення символів зовсім необов'язково. Наприклад, правила можуть твердити: «Візьміть такий-то ієрогліф з кошика номер один і помістіть його поряд з таким-то ієрогліфом з кошика номер два».

Уявімо собі, що люди, які перебувають за дверима кімнати і розуміють китайську мову, передають у кімнату набори символів, і що у відповідь я маніпулюю символами відповідно до правил і передаю назад інші набори символів. У цьому випадку книга правил є не що інше, як «комп'ютерна програма». Люди, які написали її, — «програмісти», а я граю роль «комп'ютера».

Кошики, наповнені символами, — це «база даних»; набори символів, що передаються до кімнати — «питання», а набори, що виходять з кімнати — «відповіді».

Припустимо далі, що книга правил написана так, що мої «відповіді» на «питання» не відрізняються від відповідей людини, яка вільно розмовляє китайською мовою. Наприклад, люди, що знаходяться зовні, можуть передати незрозумілі мені символи, що означають: «Який колір вам більше всього подобається?» У відповідь, виконавши запропоновані правилами маніпуляції, я видам символи, що означають, що мій улюблений колір синій, але мені насправді подобається зелений. Таким чином, я витримаю тест Тюрінга на розуміння китайської мови. Але все ж насправді я не розумію ані слова китайською. До того ж, я ніяк не можу навчитися цій мові у розглянутій системі, оскільки не існує жодного способу, за допомогою якого я міг би довідатися сенс хоча б одного символу. Подібно до комп'ютера, що маніпулює символами, але не може вкласти в них жодного змісту».

В 1984 р. Серль формулює свою ідею більш формалізовано. Він розглядає наступні *передумови*:

1. Мозок породжує розум.
2. Синтаксису недостатньо для існування семантики.
3. Комп'ютерна програма повністю визначається своєю синтаксичною структурою.
4. Людський розум оперує смисловим змістом (семантикою).

І робить *висновки*:

1. Програми не є сутністю розуму і їх наявності недостатньо для наявності розуму.
2. Той спосіб, за допомогою якого людський мозок насправді породжує ментальні явища, не може зводитися лише до виконання комп'ютерної програми.
3. Те, що породжує розум, повинно мати принаймні причинно-наслідкові властивості, еквівалентні відповідним властивостям мозку.

Уявний експеримент викликав інтенсивну дискусію серед учених. Існують різні аргументи проти результатів, отриманих Серлем:

- Деякі критики вважають, що тест Тюрінга витримує не людина, а система, що складається з кімнати, книги правил і людини. Серль, на їхню думку, відволікає увагу

від цього факту, концентруючись на одному з компонентів системи, що виконує в даному випадку чисто механічну роботу. Система з книги правил, людини і кімнати, на їхню думку, є розумною і розуміє китайську мову.

- Інша думка полягає в тому, що жодної семантики не існує в принципі: все, що відбувається в мозку, всього лише маніпулювання синтаксисом, яке здійснюється і в комп'ютерах.
- На думку С. Лема, розуміння у самій "китайської кімнаті" дійсно не існує, але воно є в мозку її творця — того, хто розробив систему правил поводження з ієрогліфами для учасника експерименту. «Робота завжди асемантично формальна», і це не може бути підставою говорити про відсутність розуміння. Автор експерименту повинен був заздалегідь передбачити всі комбінації питань і відповідей або придумати принцип добору відповіді на запитання. У реальних умовах при створенні передбачуваного штучного інтелекту навряд чи таке буде можливо.

2. ІНТЕЛЕКТУАЛЬНІ СИСТЕМИ

2.1. Керування складними системами

2.1.1. Алгоритмічний та декларативний підходи до керування

На найбільш загальному рівні, можна виділити два підходи до керування складними системами та до програмування роботів і комп'ютерів, що могли б розв'язувати ті чи інші задачі. При програмуванні сучасних комп'ютерів в основному реалізований традиційний **алгоритмічний підхід**, який можна ще назвати імперативним. Цей підхід вимагає заздалегідь продумати та детально розписати, як треба вирішувати певну проблему. Написання програми вимагає задання чітких послідовностей інструкцій. Якщо таку послідовність вдається написати — комп'ютер зможе вирішувати дану задачу, якою б складною вона не була. Але, ясна річ, він виконувати інструкції, абсолютно не розуміючи їх змісту. Якщо зміняться умови, за яких виконується програма, комп'ютер може виявитися безпорадним.

Інший підхід — **декларативний.** Інтелектуальному виконавцеві (людині чи комп'ютерові) досить сказати, **що** треба робити, тобто лише сформулювати завдання, побудувавши всі взаємозв'язки між об'єктами у предметній області. **Як** це завдання буде виконуватися — повинен визначити сам виконавець.

Запустити космічний корабель так, щоб він приземлився на Марсі, взяв зразки ґрунту та привіз їх назад — задача дуже складна, але вона піддається точній алгоритмізації. Математичні методи дозволяють точно розрахувати траєкторію, по якій повинен рухатися такий корабель.

Послати робота до магазину за пляшкою молока — задача, на декілька порядків більш складна. Не кажучи вже про сам процес спілкування з продавцем, робот повинен вирішити ряд не зовсім формалізованих підзадач. Заздалегідь розрахувати траєкторію руху неможливо, оскільки робот повинен уникнути зіткнень з людьми та автомобілями. Якщо магазин закритий на переоблік, він повинен знайти інший магазин. *Неможливо передбачити всі ситуації, які можуть виникнути, а відтак — алгоритмізувати розв'язок задачі.* Тому виконання такого завдання під силу лише інтелектуальній системі, яка вміє *орієнтуватися в зовнішньому світі, аналізувати поточні ситуації та коригувати, адаптувати свою поведінку на основі такого аналізу*.

2.1.2. Формалізація понять алгоритмічності та декларативності

Спробуємо формалізувати деякі з впроваджених раніше понять. Введемо такі позначення:

q — первинні інструкції, записані без будь-яких змін у тому вигляді, в якому їх сформулював автор; аналогічно можна визначити *первинний опис ситуації* як результат її безпосереднього сприйняття;

S — множина факторів, які визначають поточний стан виконавців; розглядатимемо *S* як об'єднання двох множин: S_1 та S_2, де S_1 — множина *контрольованих факторів,* які відомі авторові процедури і на які він може мати вплив, S_2 — множина *неконтрольованих* факторів;

Z — знання, які має виконавець;

r_A — робочий алгоритм, який формується та реалізується виконавцем при алгоритмічному підході. При цьому, як правило, автоматично формулюється і програма p_A, що відповідає цьому алгоритму;

r_D — робочий алгоритм, який формується та реалізується інтелектуальним виконавцем при декларативному підході. Згідно з фундаментальними тезами Тюрінга та Черча (все, що може бути виконано будь-яким виконавцем, може бути промодельоване на машині Тюрінга) сам факт виконання завдання свідчить про існування цього алгоритму. Щоправда, цей алгоритм може і не усвідомлюватися виконавцем, а тому не може бути явно сформульований у вигляді алгоритму чи програми (якщо ж програма, що відповідає алгоритму r_D, повинна бути згенерована, йдеться мова про *задачу синтезу програм*).

Тоді можна записати такі співвідношення:

$$r_A = f\left(q, S_1\right),$$

$$r_D = g\left(q, S_1, S_2, Z\right).$$

Принциповим є наступне. Алгоритм r_A формується на основі первинних інструкцій q однозначно. При цьому також можуть відбуватися зміна і поповнення первинних інструкцій, але цей процес має повністю контрольований і детермінований характер. Як приклад можна навести компіляцію програм, написаних мовами високого рівня. Тому автор процедури може бути впевнений у гарантованому результаті (якщо, звичайно, були дотримані певні формальні вимоги до первинних інструкцій і забезпечено належний стан виконавця).

На противагу цьому, за декларативного підходу такої впевненості немає. Інтелектуальний виконавець певним чином поповнює первинні інструкції, але їх авторові не завжди відомо, яким чином це поповнення відбувається. Тому не можна бути впевненим у результаті виконання інструкцій, і ця втрата гарантованості є неминучою платою за відмову від алгоритмічності. Отже, ми маємо справу з узагальненням поняття алгоритму, яке дістало назву **"квазіалгоритм".**

2.2. Квазіалгоритми

Алгоритмом називається чітка однозначна зрозуміла виконавцеві послідовність інструкцій, виконання яких обов'язково приводить до гарантованого результату за скінченний час. На відміну від цього, інструкції квазіалгоритму можуть бути не зовсім чіткими, і результат виконання квазіалгоритмічної процедури не обов'язково є гарантованим.

Можна виділити як мінімум чотири основні *джерела квазіалгоритмічності*:

1. **Дія випадкових чинників**, що не залежать від виконавця. Строго кажучи, з огляду на цей фактор квазіалгоритмом слід вважати будь-який, навіть найбільш формалізований, алгоритм. Алгоритм розрахований на роботу при певних умовах; якщо ці умови зміняться, алгоритм може не призвести до потрібного результату. *Якщо ж не вдається чітко окреслити межі застосування алгоритму або забезпечити виконання необхідних умов для його роботи, ми маємо справу зі справжнім квазіалгоритмом*.

2. **Недостатнє врахування автором алгоритмічної процедури особливостей виконавця.** Це призводить до того, що виконавець неправильно розуміє, що від

нього вимагається. Процедура може бути в принципі алгоритмічною, за нормальних умов призводити до гарантованого результату, але цей результат може бути не передбачений автором і тому бути для нього несподіванкою.

3. **Нечіткість формулювань**, що фігурують в описі процедури. Розглянемо будь-який кулінарний рецепт, наприклад: “розтерти тісто, змішати з дрібно нарізаними яблуками, додати солі і перцю за смаком і смажити до появи рум’яної скоринки”. Це є типовим прикладом нечіткості формулювань, що призводить до невизначеностей. До якої межі слід розтирати тісто? Що означає “дрібно нарізані”? Що означає “за смаком” і т. д. Неінтелектуальна система, орієнтована на чисто алгоритмічне керування, просто не зрозуміє цього опису і тому не зможе його виконати. Інтелектуальна ж система, наприклад, людина, повинна спробувати поповнити і уточнити цей опис на основі наявних знань і досвіду. Сам по собі факт виконання завдання буде свідчити про те, що квазіалгоритмічний первинний опис процедури був зведений до деякого внутрішнього алгоритму, але навряд чи виконавець зможе чітко пояснити і розписати цей алгоритм. Крім того, ці внутрішні алгоритми для кожного виконавця будуть різними.

4**. "Свобода волі",** яку можуть мати високоорганізовані, по-справжньому інтелектуальні системи. Ці системи можуть мати свої цілі, і, якщо дана процедура суперечить цим цілям, вони можуть відмовитися її виконувати або виконати не так, як це від них вимагається. “Свобода волі” полягає у тому, що інтелектуальна система самостійно приймає рішення про свою поведінку і в залежності від цих рішень може виконувати зовнішні розпорядження, не виконувати їх, або ж виконувати так, як вона вважає за потрібне.

2.3. Характеристика інтелектуальних систем з точки зору кібернетики

2.3.1. Означення інтелектуальної системи

Інтелектуальна система може припускати зовнішнє керування, але для неї характерною є **самокерованість**. Система має **певну мету і прагне так планувати свої дії, щоб досягати цієї мети**. Як вхідні стимули системи можна розглядати поточну ситуацію, що сприймається і аналізується системою. Результатом реакції системи стає зміна зовнішньої ситуації, і поведінка системи коригується в залежності від того, бажаною чи небажаною є ця зміна.

Людина має певну суму **знань про світ**, яка дозволяє їй орієнтуватися в життєвих ситуаціях та приймати правильні рішення. Крім того, людина вміє певним чином **використовувати ці знання**. Ці самі риси мають мати системи штучного інтелекту.

Також можна стверджувати, що **здатність до поповнення первинних знань, є** *однією із ключових рис інтелектуальних систем*. Ця властивість інтелектуальних систем називається **здатністю до навчання.** Розрізняють *зовнішнє навчання* та *самонавчання*.

Підсумувавши усі сказане, можна стверджувати, що **інтелектуальною системою** *називається самокерована кібернетична система, яка має певну суму знань про світ і здатна на основі безпосереднього сприйняття і подальшого аналізу поточної ситуації до планування дій, спрямованих на досягнення мети, а також до навчання*.

2.3.2. Типова схема функціонування інтелектуальної системи

Функціонування інтелектуальної системи можна описати як постійне прийняття рішень на основі аналізу поточних ситуацій для досягнення певної мети. Схема функціонування інтелектуальної системи наведена на рис. 2.1.

Виокремлюють наступні *етапи функціонування інтелектуальних систем*:

1. **Безпосереднє сприйняття зовнішньої ситуації.**
   Результатом є первинний опис ситуації.

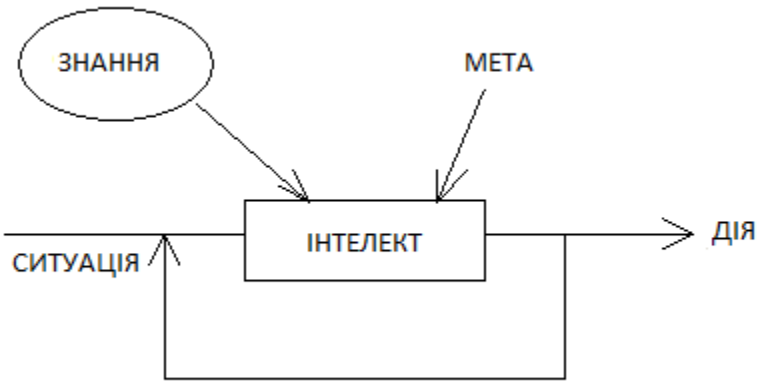


Рис. 2.1. Схема функціонування інтелектуальної системи.

2. **Зіставлення первинного опису зі знаннями системи і поповнення цього опису.**
   Результатом є формування вторинного опису ситуації в термінах знань системи. Цей процес можна розглядати як процес *розуміння ситуації* або як процес перекладу первинного опису на внутрішню мову системи. При цьому *можуть змінюватися* внутрішній стан системи та її знання.

   Вторинний опис може бути не єдиним, і система має змогу вибирати між різними вторинними описами. Крім того, система в процесі роботи може переходити від одного вторинного опису до іншого.
3. **Планування цілеспрямованих дій та прийняття рішень, аналіз можливих дій та їх наслідків і вибір тих дій, що найкраще узгоджуються з метою системи.**
   Таке рішення зазвичай формулюється певною внутрішньою мовою (свідомо або підсвідомо).
4. **Зворотна інтерпретація прийнятого рішення, тобто формування робочого алгоритму для реагування системи.**
5. **Реалізація реакції системи; наслідками є зміни зовнішньої ситуації і внутрішнього стану системи і т. д.**

Проте не слід вважати, що зазначені етапи є повністю відокремленими у тому розумінні, що наступний етап починається тільки після того, як повністю закінчиться попередній. Навпаки, для функціонування інтелектуальної системи характерним є взаємне проникнення цих етапів. Наприклад, ті чи інші рішення можуть прийматися вже на етапі безпосереднього сприйняття ситуації. Насамперед це рішення про те, на які зовнішні подразники слід звертати увагу, а на які не обов’язково. Зовнішніх подразників так багато, що їх сприйняття повинно бути вибірковим.

3. ПОДАННЯ ЗНАНЬ В ІНТЕЛЕКТУАЛЬНИХ СИСТЕМАХ

3.1. Підходи до подання знань

У філософії знання визначаються як "відображення об'єктивних властивостей та зв'язків світу".

Знання *є інформаційною основою інтелектуальних систем, оскільки саме вони завжди зіставляють зовнішню ситуацію зі своїми знаннями і керуються ними при прийнятті рішень. Не менш важливим є те, що знання — це систематизована інформація, яка може певним чином поповнюватися і на основі якої можна отримувати нову інформацію, тобто нові знання.*

Існує багато підходів до визначення поняття "знання". На сучасному етапі однією з двох основних парадигм, що лежать в основі найвідоміших моделей подання знань у системах штучного інтелекту, можна вважати парадигму (певну сукупність ключових принципів), характерну для **символьного підходу**. Цю парадигму можна охарактеризувати як **вербально-дедуктивну**, або **словесно-логічну**:

- будь-яка інформаційна одиниця задається вербально, тобто у формі, наближеній до словесної, у вигляді набору явно сформульованих тверджень або фактів;
- основним механізмом отримання нової інформації на базі існуючої є **дедукція**, тобто висновок від загального до часткового. Такий підхід є бездоганним з логічного погляду. В його основі лежать транзитивність імплікації (якщо з a випливає b, з b випливає c, то з a випливає c) і дія квантора загальності (якщо деяка властивість P виконується для будь-якого елемента множини M, а $x \in M$, то P виконується для x.

Але вербально дедуктивне задання знань не є повним, оскільки:

- дедуктивний висновок не виступає єдино можливим. Мислення людини багато в чому є рефлекторним, інтуїтивним. Воно, як правило, спирається на підсвідомі процеси. Людина часто робить висновки за аналогією, асоціацією. Ці висновки не завжди вірні, але вони істотно доповнюють процеси дедуктивного мислення. Без підсвідомого мислення стає неможливим (будь-яке відкриття — як правило, підсвідоме породження гіпотези, яка потім перевіряється дедуктивним або експериментальним шляхом);
- далеко не всі знання є вербальними. Так, жодне твердження не зберігається в пам'яті людини явно. Відомо, що в основі діяльності мозку людини лежить передача сигналів між нервовими клітинами. Часто людина не може сформулювати свої знання. Наприклад, будь-хто знає, що таке "стіл". Але якщо попросити людину дати визначення цього поняття, у неї можуть виникнути проблеми. Таким чином, поняттями можна оперувати і не знаючи чіткого їх визначення. Типовими є слова: "Я не можу пояснити чому, але мені здається".

Тому необхідно розвивати інші моделі знань, окрім вербально-дедуктивних Наприклад, у рамках *конекціоністського підходу* можна розглядати моделі на основі однорідного поля знань. *Однорідним полем знань* називається сукупність простих однорідних елементів, які обмінюються між собою інформацією: нові знання народжуються на основі певних процедур, визначених над полем знань.

3.2. Вербально-дедуктивне визначення знань

У рамках вербально-дедуктивної парадигми можна навести таке визначення знань.

Знаннями *інтелектуальної системи називається трійка <F, R, P>, де F — сукупність явних фактів, які зберігаються в пам'яті системи в явному вигляді, R — сукупність правил виведення, які дозволяють на основі відомих знань набувати нових знань, P — сукупність процедур, які визначають, яким чином слід застосовувати правила виведення.*

Базою знань *(БЗ) інтелектуальної системи називатимемо сукупність усіх знань, що зберігаються в пам'яті системи.*

У базах знань прийнято розрізняти *екстенсіональну* та *інтенсіональну* частини.

Екстенсіональною частиною бази знань називається сукупність усіх явних фактів, *інтенсіональною* частиною — сукупність усіх правил виведення та процедур, за допомогою яких з існуючих фактів можна виводити нові твердження.

Приклад. Розглянемо таку базу знань.

F1. Заєць їсть траву.
F2. Вовк їсть м'ясо.
F3. Заєць є м'ясом.
R1. Якщо *A* є м'ясом, а *B* їсть м'ясо, то *B* їсть *A*.
R2. Якщо *A* їсть м'ясо, то *A* є хижаком.
R3. Якщо *A* їсть *B*, то *B* є жертвою.
R4. Якщо хижак вимирає, жертва швидко розмножується.
R5. Якщо жертва вимирає, хижак вимирає.

Даний приклад є фрагментом неформалізованого опису моделі Лотки-Вольтерра, добре відомої в екології. Літерами "F" позначені явні факти, які відображають елементарні знання. Їх безпосередній аналіз дозволяє експертній системі відповідати на найпростіші запитання, такі, як "Вовк їсть м'ясо?". Для позитивної відповіді досить просто знайти відповідний факт в екстенсіональній частині бази знань.

Принципово іншим є запитання типу "Вовк є хижаком?". Відповідь на нього на основі простого пошуку знайти неможливо: такого явного факту в екстенсіональній частині бази знань немає. Тому для відповіді на це запитання необхідно застосовувати правила виведення, які дозволяють на основі існуючих явних фактів набувати нових знань, тобто нових фактів. Такі правила позначені у нашому прикладі літерою "R"**.** Наприклад, для відповіді на поставлене запитання досить знання факту F2 і правила R2.

Безумовно, до бази знань можна було б відразу включити твердження "Вовк є хижаком". Але легко побачити: якщо, крім вовків, розглядати інших хижаків (ведмедів, лисиць та ін.), то безпосередньо включати до бази знань явні факти, що вони є хижаками, недоцільно. Все, що може бути виведене логічним шляхом, як правило, не варто оголошувати фактами. Надалі називатимемо явні факти просто фактами, якщо це не викликає непорозумінь.

Нарешті, *процедури* визначають, яким саме чином слід застосовувати правила. Часто ці процедури є складовою мови, якою програмується експертна система (це стосується насамперед спеціалізованих мов ШІ, таких як Лісп, Пролог, Пленер).

3.3. Експертні системи

База знань, наведена в п. 3.2, може лягти в основу *експертної системи*, тобто інформаційної системи, яка здійснює дедуктивне виведення на основі наявних знань. Існують різні визначення експертних систем, основна суть яких зводиться до такого.

Експертні системи *— це інтелектуальні програмні засоби, здатні у діалогу з людиною одержувати, накопичувати та коригувати знання із заданої предметної галузі, виводити нові знання, розв'язувати на основі цих знань практичні задачі та пояснювати хід їх розв'язку.*

Експертні системи акумулюють знання експертів — провідних спеціалістів у даній предметній галузі. В основі роботи експертних систем лежить дедуктивне виведення нових тверджень з існуючих. Типове застосування експертних систем — консультації для фахівців середньої кваліфікації і неспеціалістів у тій галузі, для якої вона розроблена.

Створити універсальну експертну систему неможливо. По-перше, це пов'язано з "прокляттям розмірності". По-друге, експертна система повинна акумулювати знання людей-експертів, а "універсальних" експертів не існує і не може існувати. По-третє, жодне логічне виведення не може замінити інтуїцію та досвід експерта.

Можна лише експертні системи, які належать до конкретної предметної галузі. Остання передбачає лімітований набір явищ і понять певної сфери людської діяльності та обмежене коло задач, які вирішуються у цій галузі. Існує чимало експертних систем у таких сферах як медична і технічна діагностика, пошук корисних копалин, юриспруденція, аналіз інвестицій і комерційних ризиків і т. п. У створенні експертних систем повинні брати участь фахівці як мінімум двох категорій:

- **експерт,** що є висококваліфікованим фахівцем у даній предметній області, знання якого потрібно передати експертній системі;
- **інженер знань,** завдання якого — формалізувати знання експерта і *привести* їх до вигляду, придатного для занесення до бази знань.

Серйозна проблема у даному випадку пов'язана з тим, що знання експертів важко формалізувати. Експерт часто не може сформулювати свої знання у явному вигляді, робить правильні висновки, але *не може пояснити як саме він їх робить*. Якщо ж експерт і формулює певні правила виведення, вони далеко не завжди відзначаються точністю та адекватністю. З цього випливає ряд інших труднощів, які так чи інакше виявляють себе на етапі формалізації знань або застосування готових експертних систем.

3.4. Дані та знання

Відомо, що сучасний етап розвитку інформатики характеризується еволюцією моделей даних у напрямі переходу від традиційних (реляційна, ієрархічна, мережева) до моделей знань. Знання, як будь-яку інформацію, можна вважати даними. Але *знання є високоорганізованими даними, для яких характерна певна внутрішня структура та розвинуті зв'язки між різними інформаційними одиницями.* Іншим принциповим аспектом є те, що інформаційні системи, які ґрунтуються на знаннях, повинні мати можливість отримувати нові знання на основі існуючих. У цьому ми вже пересвідчилися на прикладах дедуктивного виведення нових знань з явних фактів.

Екстенсіональна частина БЗ складається з фактів, які запам'ятовуються явно, *інтенсіональна* — з правил, які дозволяють отримувати нові факти. Наявність розвинутої

інтенсіональної частини є однією з основних рис, які відрізняють знання від традиційних даних. *Інтенсіональним* відношенням, яке описує БЗ, часто називають правило або сукупність правил, яким підпорядковується кожний запис у БЗ.

Приклад. Розглянемо базу даних (БД) деканату, що містить дані про те, які курси прослухав кожний студент. Відомо, що жоден студент не має права слухати курс "Бази даних", якщо він не прослухав курсу "Основи програмування". Нехай у базі даних зберігається інформація про Іваненка, Петренка та Сидоренка, які прослухали обидва курси.

Екстенсіональна частина:

Прізвище	Дисципліна
Іваненко	Бази даних
Петренко	Бази даних
Сидоренко	Бази даних

Інтенсіональна БД (або її вже можна назвати БЗ) може мати такий вигляд:

Правило: *Якщо Прослухав*(*X*, *Бази даних*), *то Прослухав*(*X*, *Основи програмування*).

Наявність такого правила дозволяє скоротити базу знань: твердження, істинність яких можна встановити за допомогою правил, не запам’ятовуючи їх в явному вигляді.

Пошук в інтенсіональній БЗ складається з двох етапів:

- пошук потрібного факту в екстенсіональній частині;
- дедуктивне виведення факту на основі правил інтенсіональної частини.

Реалізовувати інтенсіональні правила можна навіть у рамках реляційної моделі даних шляхом програмування відповідних запитів.

Знання можуть бути **неповними.** Це означає, що для доведення або спростування певного твердження може не вистачати інформації. У багатьох системах логічного виведення прийнято **постулат замкненості світу:** на запит про істинність деякого твердження система відповідає "так" тоді і тільки тоді, коли його можна довести; якщо ж довести неможливо, система відповідає "ні". Водночас *"неможливо довести через нестачу інформації"* і *"доведено, що ні"* — це зовсім не те саме. З огляду на це бажано, щоб експертна система запитувала у користувача про факти, яких не вистачає.

Знання можуть бути **недостовірними.** Наприклад, на виготовлення продукції можуть впливати випадкові чинники (об’єктивна невизначеність) або експерт може бути не зовсім упевненим у деякому факті чи правилі (суб’єктивна невизначеність).

Ненадійність знань і недостовірність наявних фактів обов’язково повинні враховуватися в процесі логічних побудов. Звичайно, можна було б просто відкидати факти та правила виведення, які викликають сумнів, але довелося би відмовитися від цінної інформації. Крім того, в експертних системах часто доводиться мати справу з неточно визначеними поняттями, такими, як *"великий", "маленький"* тощо.

3.5. Зв’язки між інформаційними одиницями

У базі знань можуть зберігатися відомості про різні об’єкти, поняття, процеси і т. п., тобто відповідні описи, які називаються **інформаційними одиницями**.

Опис, який утворює інформаційну одиницю, розглядається як деяка абстракція реальної сутності, що існує в дійсному світі.

Це означає що дана абстракція описує певні риси реальної сутності. Бажано, щоб абстракція описувала найважливіші з них.

У реальному світі все тісно взаємопов'язане. Відповідні зв'язки повинні знайти адекватне відображення в БЗ. Існує велика група зв'язків, за допомогою яких можна описувати просторові відношення ("знизу", "зверху", "поруч", "між" і т. п.); часові відношення ("раніше", "пізніше", "під час", "одночасно" і т. п.), причинно-наслідкові відношення тощо. Ці відношення є спільними для всіх предметних областей. Логічні системи, що ґрунтуються на цих відношеннях, називаються псевдофізичними логіками**.** Використання псевдофізичних логік (гак само, як і більш загальних предикатів, таких, як узагальнення та агрегація) дає можливість інтелектуальній системі поповнювати первинні описи.

Так, якщо інтелектуальна система сприймає текст: "У 1985 році почалася перебудова. У 1991 році була проголошена незалежність України", врахування часових властивостей, що визначається часовою логікою, дає змогу відповідати на запитання типу: "Що було раніше: початок перебудови и чи проголошення незалежності України?"

Основними зв'язками між інформаційними одиницями є *узагальнення* та *агрегація.*

Агрегація дозволяє задавати будову об'єкта та його складових. Наприклад, агрегація дає змогу визначити в базі знань аудиторію як об'єкт, що має вікна, двері і т. п. Вікна, в свою чергу, теж можна розглядати як агрегований об'єкт: вони мають ручки, рами і т. п. Інакше кажучи, відношення *"Має",* яке описує агрегацію, означає, що певний об'єкт містить у своєму складі деякий інший об'єкт. За цим відношенням закріпилася спеціальна назва **Has_Part.** По суті, це відношення "ціле — частина".

Таким саме чином ввести і обернене відношення (*"Є частиною",* або **Is_Part**).

Узагальнення (відношення *"Є"*) задає ієрархію класів **(екземпляр — клас — надклас).** Так, можна говорити, що об'єкт Василь Петренко належить до класу *Студент*. Відповідно інформаційна одиниця, яка описує Василя Петренка, може бути описана на основі більш загальної інформаційної одиниці *Студент* (точніше, вона задається як екземпляр інформаційної одиниці *Студент*). Клас *Студент*, у свою чергу, є *підкласом* типу *Людина* і т. д. Надклас об'єднує атрибути, або ознаки, спільні для його підкласів; підкласи можуть успадковувати ті чи інші властивості надкласів.

Із відношенням *"Є"* пов'язаний один з основних відомих механізмів логічного виведення — механізм **виведення за успадкуванням** (інша назва — виведення *за наслідуванням*)**.** Суть цього механізму можна сформулювати так: якщо деяка умова виконується для всього класу, то вона виконується і для кожного представника цього класу, а також для всіх підкласів цього класу (якщо інше не задано явним чином). Інакше кажучи, екземпляри успадковують властивості класів, підкласи успадковують властивості надкласів.

В усіх системах керування БЗ повинна бути забезпечена належна підтримка успадкування. Розглянемо приклад із [8]. У базі знань міститься така інформація:

Усі птахи літають.
Ластівка є птахом.
Юкко є ластівкою.

Маємо загальний клас *"Птахи",* його підклас *"Ластівки"* та об'єкт *"Юкко",* який є екземпляром класу *"Ластівка".* На підставі цих знань будь-яка система, що підтримує успадкування, повинна зробити такі висновки: *Юкко є птахом; всі ластівки літають; Юкко теж літає.*

Насправді, замість єдиного відношення *"Є"* необхідно розглядати цілу систему споріднених, але не ідентичних відношень. Інакше легко припуститися логічних помилок, подібних до таких:

Юкко є ластівкою.
Ластівка є видом, який вивчається натуралістами.
Отже, *Юкко є видом, який вивчається натуралістами.*

Слід розглянути такі відношення:

- **"клас — підклас",** яке є відношенням часткового порядку (звичайно строгого). Для цього відношення виконується властивість транзитивності;
- **"екземпляр — клас".** Екземпляри успадковують властивості своїх класів, але саме відношення "екземпляр — клас" не є транзитивним.

"Ластівка" є екземпляром класу "Види, які вивчаються натуралістами", а "Юкко" є екземпляром класу "Ластівка", але "Юкко" у жодному разі не є екземпляром класу "Види, які вивчаються натуралістами" ("Юкко" взагалі не є видом).

Також необхідно чітко розрізняти поняття **"клас"** і **"множина", "належність до класу"** та **"належність до множини".** Клас об'єднує однотипні елементи, множина — необов'язково. Відношення "елемент — множина" також не є транзитивним. Складні системи можуть бути описані у вигляді певної канонічної форми, яка являє собою композицію двох ієрархій — *ієрархії класів* та *ієрархії об'єктів***. Ієрархія класів** задається відношенням *узагальнення,* **ієрархія об'єктів** — *відношенням агрегації.*

Дійсно, з одного боку, такі об'єкти, як "ручки", "рами" та інші, утворюють новий об'єкт — "вікно". "Вікна", "двері" тощо — це об'єкти, які утворюють ще один новий об'єкт: "аудиторію". Таким чином, можна розглядати певну ієрархію об'єктів. З іншого ж боку, всі ці об'єкти є екземплярами своїх класів. Так, конкретне вікно належить до класу "Вікна", клас "Вікна" є підкласом класу "Дерев'яні вироби" і т. п.

Відношення "клас — підклас" є характерним для ієрархії класів, відношення "елемент — множина" та "підмножина — множина" — до ієрархії об'єктів, а відношення "екземпляр — клас" об'єднує обидві ієрархії.

3.6. Проблема винятків

З успадкуванням пов'язана дуже серйозна проблема — **проблема винятків.** Вона полягає в тому, що деякі підкласи можуть не успадковувати ті чи інші властивості надкласів. Інакше кажучи, характерні риси класу успадковуються всіма його підкласами, **крім** деяких.

Нехай відомо, що *літають всі птахи, крім пінгвінів* (існують деякі інші види птахів, які не літають, але для наших цілей це не має суттєвого значення). Якби це твердження відразу потрапило до БЗ саме в такому вигляді, особливих проблем не виникало б (хоча і в цьому випадку слід було б передбачити належну обробку винятків).

Але, як було зазначено раніше, експерт не завжди може сформулювати свої знання в явному вигляді. Зокрема, він може не знати або не пам'ятати всіх винятків. Тому він може спочатку включити до БЗ твердження про те, що *всі птахи літають,* а потім пригадати, що *пінгвіни не літають,* і додати це до БЗ. У результаті ми могли б отримати БЗ, подібну до такої:

Усі птахи літають.
Ластівка є птахом.
Юкко є ластівкою.

Пінгвін є птахом.
Пінгвіни не літають.
Бакс є пінгвіном.

Якби передостаннє твердження не входило до бази знань, система просто дійшла б хибного висновку, що *Бакс літає.* Але включення даних відомостей до бази знань ще більше ускладнює ситуацію. **Система знань стає суперечливою**: з одного боку, система повинна дійти висновку, **що** Бакс літає, а з іншого — що Бакс не літає. У даному разі кажуть про **втрату монотонності** дедуктивної системи [8].

Система дедуктивного виведення *називається* ***монотонною,*** *якщо для неї виконується така властивість: якщо з набору тверджень* (q_1, …, q_n) *випливає твердження V, то V випливає і з набору тверджень* (q_1, …, q_n, r).

Інакше кажучи, в монотонній системі додавання нових фактів і правил не впливає на істинність висновків, які могли бути отримані без них.

Для вирішення проблеми винятків часто використовують наступне правило:
у разі виникнення суперечностей підклас успадковує відповідну властивість лише від найближчого попередника, тобто найближчого до нього в ієрархії класів.

3.7. Властивості та моделі знань

У [9] сформульовані такі особливості знань, які відрізняють їх від звичайних даних:

1) **внутрішня інтерпретованість:** кожна інформаційна одиниця повинна мати унікальне ім’я, за яким інформаційна система її знаходить, а також відповідає на запити, в яких це ім’я згадується;
2) **структурованість:** знання повинні мати гнучку структуру; одні інформаційні одиниці можуть включатися до складу інших (відношення типу "частина — ціле", "елемент — клас");
3) **зв’язність*:*** в інформаційній системі повинна бути передбачена можливість встановлення різних типів зв’язків між різними інформаційними одиницями (причинно-наслідкові, просторові та ін.);
4) **семантична метрика*:*** на множині інформаційних одиниць корисно задавати відношення, які характеризують ситуаційну близькість цих одиниць;
5) **активність:** виконання програм в інтелектуальній системі повинно ініціюватися поточним станом бази знань.

Часто виокремлюють інші властивості знань, наприклад **шкальованість,** яка означає, що формально неоднакові поняття насправді відображаються на одній і тій самій **шкалі понять,** різні точки якої відповідають інтенсивності того самого фактору. Так, температура може бути високою або низькою, і це породжує такі поняття, як "холодно", "тепло", "гаряче" тощо.

Моделлю знань *називається фіксована система формалізмів (понять і правил), відповідно до яких інтелектуальна система подає знання в своїй пам’яті та здійснює операції над ними.*

Моделі задання знань необхідні:

- для створення спеціальних мов описів знань і маніпулювання ними;
- для формалізації процедур зіставлення нових знань з уже існуючими;
- для формалізації механізмів логічного виведення.

Найвідомішими з моделей, які ґрунтуються на вербально-дедуктивній парадигмі, є чотири класи моделей:

- семантичні мережі;
- фреймові;
- логічні;
- продукційні.

Вербально-дедуктивні моделі задання знань мають багато спільних рис, Отже, можна вважати, що всі вони мають єдину концептуальну основу.

3.8. Неоднорідність знань. Області і рівні знань

Для успішного здійснення операцій зі знаннями необхідно відокремлювати в БЗ певні фрагменти, які називаються **областями знань.** Ці області знань повинні бути відносно незалежними між собою, що означає таке:

- зміни в одній області знань не повинні приводити до суттєвих змін у інших областях;
- вирішення складної задачі можна, як правило, звести до підзадач таким чином, що для вирішення кожної з цих підзадач достатньо знань з однієї області.

Такий розподіл є важливим для полегшення проектування і використання БЗ. Зокрема, різні області знань можуть проектуватися незалежно одна від одної.

Виходячи з постановки задачі можна по-різному розділяти знанні на області. Так, для експертних систем, які ведуть діалог з користувачем мовою, наближеною до природної, можна виокремити такі області знань [7]:

- **предметна область,** яка містить знання про конкретну предмет, галузь, в якій працює експертна система;
- **область мови,** яка містить знання про мову, якою ведеться діалог;
- **область системи,** яка містить знання експертної системи про власні можливості;
- **область користувача,** яка містить знання про користувача. Наявність їх дозволяє враховувати індивідуальні особливості кожного користувача. Наприклад, пояснення користувачеві можуть надавати залежно від рівня його підготовленості;
- **область діалогу,** яка містить знання про мету діалогу, а також про форми та методи його організації.

Знання можуть різнитися за **рівнем задання** і **рівнем детальності. За рівнями задання** розрізняють знання нульового рівня (конкретні і абстрактні) і знання вищих рівнів — знання про знання **(метазнання).**

У випадку класифікації знань за **рівнями детальності** до уваги береться ступінь деталізації знань. Кількість рівнів детальності залежить від специфіки задачі, обсягу знань і моделі їх задання. Якщо користувач запитує в експертної системи, як здійснити певну операцію. У разі отримання відповіді на найвищому рівні детальності система видає *загальний план.* Якщо цей загальний план не є достатнім, система може спуститися на нижчий рівень і розписати операції детальніше.

3. 9. База знань як об’єднання простіших одиниць

БЗ є кон’юнкцією простих тверджень. Можна вважати, що кожне з них відповідає окремому реченню природної мови. Подібні твердження називають *концептуальними одиницями.* Останні можуть бути або фактами, або правилами виведення.

Якщо концептуальна одиниця являє собою факт, вона може бути описана певним *предикатом***,** тобто логічною функцією, що залежить від заданої кількості змінних і може приймати одне з двох можливих значень 0 або 1 (false або true). Конкретні твердження утворюються з предикатів шляхом підстановки конкретних значень аргументів. При цьому для опису одного й того самого твердження можуть використовуватися різні предикати. Так, два твердження:

7 + 5 = 12,

8 + 9 = 15

відповідають одному предикату Плюс(*a*, *b*, *c*). Перша підстановка конкретних значень замість змінних *a, b, c* породжує істинне твердження, друга — хибне. Формально ці твердження будуть мати запис: Плюс(7, 5, 12) та Плюс(8, 9, 15).

Можна також розглядати загальніший предикат: Операція (*назва_операції*, *a, b, c*), який можна використовувати для запису тверджень про будь-яку бінарну операцію: множення, ділення і т. п. Можливим є запис: Операція(*Плюс*, 7, 5, 12). Який тип розглянутих предикатів слід обирати, залежить від проектувальника БЗ і специфіки конкретної задачі.

3.10. Бінарні предикати і тріада "об'єкт—атрибут—значення"

Концептуальна одиниця може бути достатньо складною. Тому концептуальну одиницю, що є фактом, часто зручно розглядати як кон'юнкцію елементарніших тверджень, а саме — **бінарних фактів,** кожний з яких описується **бінарним предикатом,** тобто предикатом, який залежить від двох змінних. Якщо формалізувати правила об'єднання бінарних предикатів у складніші одиниці, на основі бінарних предикатів можуть бути сконструйовані складні твердження та системи знань.

Розглянемо приклад. Маємо твердження *"Студент Іванов отримав п'ятірку на іспиті зі штучного інтелекту".*

Перепишемо цю фразу у вигляді кон'юнкції бінарних фактів:

Є(Іванов, Студент)
Є(ІспШтІнт, Іспит)
Ісп_ШІ(Іванов, 5)

Предикат Ісп_ШІ був уведений для зв'язку між різними інформаційними одиницями.

У вигляді бінарного можна переписати й **унарний предикат**, тобто предикат, який залежить від однієї змінної. Наприклад, предикат Птах(*x*), який означає, що *x* є птахом, можна переписати як **Є(***x***,** Птах).

Якщо певний унарний предикат описує **властивість** об'єкта, наприклад Літає**(***X***),** її можна переписати **у** вигляді Літає(*x***,** так).

Бінарним предикатам і бінарним фактам безпосередньо відповідає **тріада "об'єкт — атрибут — значення".** Об'єкт у цій тріаді — це, як правило, назва деякої інформаційної одиниці, атрибут — назва певної ознаки, а значення — конкретне значення цієї ознаки для даного об'єкта. Тріада "об'єкт — атрибут — значення" є просто іншою формою замін бінарного факту. Так, можна переписати наш приклад у вигляді:

Іванов —Є — Студент
Ісп_ШІ — Є — Іспит
Іванов — Ісп_ШІ — 5.

Остання тріада цього прикладу задає зв'язок між різними об'єктами.

4. МЕРЕЖЕВІ ТА ФРЕЙМОВІ МОДЕЛІ ЗНАНЬ

4.1. Семантичні мережі

В основі мережевих моделей лежить конструкція, що називається **семантичною мережею**.

Під семантичною мережею розуміється мережа, в якій поєднані зв'язки різних типів. **Семантичну мережу** можна неформально описати у вигляді графа, вершини якої, як правило, позначають об'єкти предметної області, а дуги відповідають зв'язкам між ними.

Формально ці моделі можна задати у вигляді $<I, C_1, C_2, ..., C_n, Г>$. Тут I — множина інформаційних одиниць, $C_1, C_2, ..., C_n$ — типи зв'язків між інформаційними одиницями, $Г$ — відображення, що задає зв'язки між інформаційними одиницями.

Мережні моделі тісно пов'язані з *моделями "сутність-зв'язок"*. Під сутністю розуміється об'єкт довільної природи. У базі знань відображаються також зв'язки між сутностями.

Виділяють **класифікуючі** мережі, **функціональні** мережі та **сценарії**.

Класифікуючі мережі дозволяють задавати відношення ієрархії між інформаційними одиницями.

Функціональні мережі характеризуються наявністю функціональних відношень, які дозволяють описувати процедури обчислень одних інформаційних одиниць через інші.

У *сценаріях* використовуються відношення типу "причина-наслідок", "дія", "засіб дії" та ін.

Запропоновано ряд формалізацій для представлення знань у вигляді семантичних мереж. Історично першою була модель Квілліана.

Базовим є поняття **концептуального об'єкту**, який графічно зображується **концептуальним графом**. У моделі Квілліана сутність описується концептуальним об'єктом, причому до опису включаються клас, до якого належить сутність, властивості сутності та її прототип (приклад). Як приклад можна навести концептуальний граф для чайника.

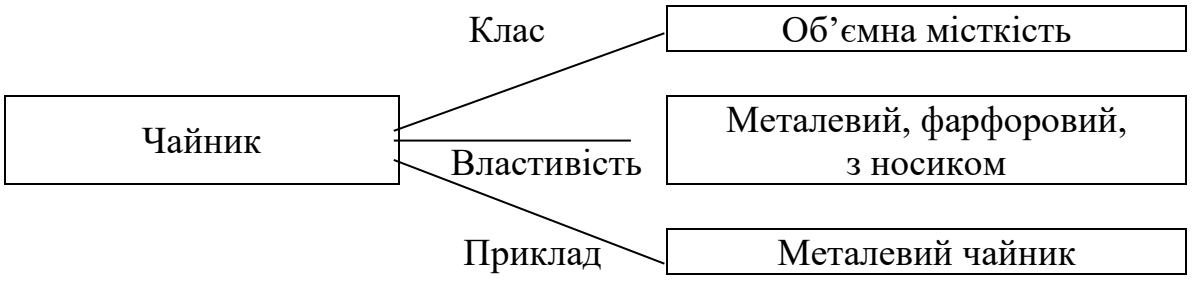


Рис. 4.1. Приклад концептуального графа

Більш загально, у вигляді концептуального графа можна зобразити будь-яку логічну формулу або словесне висловлювання. Такий концептуальний граф можна вважати простою семантичною мережею, що відповідає одному твердженню. Так, наприклад, висловлювання "Жак надсилає книгу Марі" може бути зображене у

вигляді предикату ПОСИЛКА(ЖАК_2, МАРІ_4, КНИГА_22), а концептуальний граф матиме вигляд:

Рис. 4.2. Приклад простої семантичної мережі, що відповідає одному фактові

Цифри потрібні для того, щоб дати кожній сутності своє індивідуальне ім’я, яке не співпадає ні з якими іншими.

Прямокутники концептуального графа призначені для зображення аргументів, а овали — для зображення предикатів. Круг і прямокутник з’єднуються дугою, якщо вони є відповідно ім’ям та аргументом того самого предиката.

Для спрощення вигляду концептуального графа можна не виділяти імена предикатів у вигляді окремих кругів, а писати ці імена прямо на дугах.

Часто буває корисним представити предикат, що відповідає деякій складній фразі природної мови, у вигляді кон’юнкції бінарних предикатів. У нашому прикладі предикат ПОСИЛКА(ЖАК_2, МАРІ_4, КНИГА_22) можна переписати у вигляді

Відправник(Посилка_8, Жак_2)
Отримувач(Посилка_8, Марі_4)
Що(Посилка_8, Книга_22)
Is_a(Посилка_8, Посилки).

Відповідний концептуальний граф матиме вигляд:

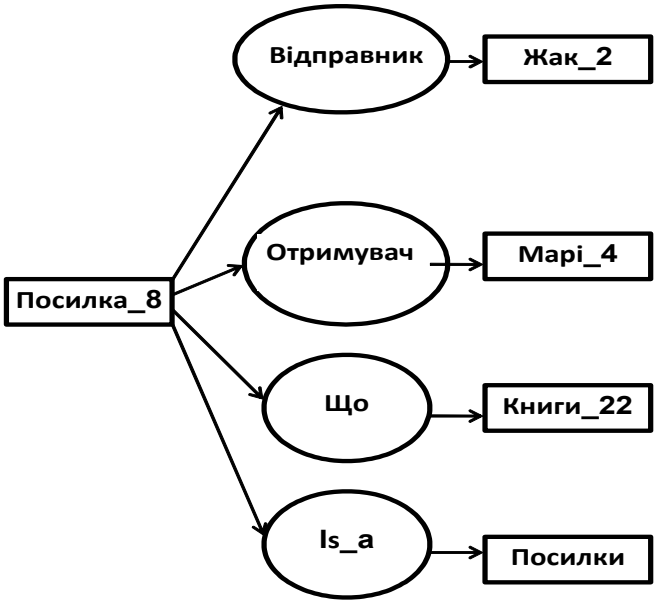


Рис. 4.3. Концептуальний граф у бінарній формі

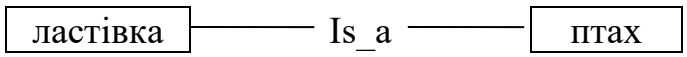
Предикати **Is_a, Has_Part** та **Is_Part** відіграють винятково важливу роль та *означають належність деякої сутності до певного типу / агрегату*.

Семантична мережа — *це композиція концептуальних графів, об'єднаних за тими чи іншими правилами.*

Багато методів логічного виведення на семантичних мережах ґрунтуються на зв'язку **Is_a** та понятті **наслідування**, що тісно з ним пов'язане. Сутність наслідує атрибути типу, до якого вона належить. Якщо для типу визначені конкретні значення атрибутів, сутність, що належить до цього типу, наслідує їх за замовченням.

Зв'язок Part_Of показує відношення **"ціле–частина".** Цей зв'язок показує відношення між екземплярами класу.

Як приклад розглянемо речення, яке задає факт "всі ластівки — птахи". Цьому реченню може відповідати така семантична мережа:



Якщо далі присвоїти ластівці ім'я "Юкко", тоді мережу можна розширити наступним чином:

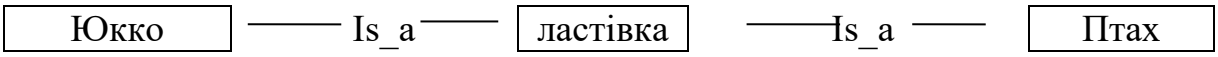


Рис. 4.4. Ілюстрація наслідування на семантичній мережі

Тоді з семантичної мережі можна вивести речення: Юкко — птах.

4.2. Фрейми

Фреймові моделі базуються на теорії фреймів, розробленій у 1975 р. М.Мінським.

Термін "*фрейм*" походить від англійського слова *frame*, яке перекладається як "рамка", "каркас". Мінський запропонував гіпотезу, згідно з якою знання групуються в модулі, і назвав фреймами ці модулі. Фрейм можна інтуїтивно уявляти собі як каркас, на якому тримаються знання, або як рамку, на основі якої описуються різноманітні об'єкти та ситуації. Мінський писав, що коли людина потрапляє в нову ситуацію, вона *співставляє* цю ситуацію з тими фреймами, які зберігаються у неї в пам'яті.

Фрейм можна визначити як *структуру даних, що призначена для опису типових ситуацій або типових понять.*

М. Мінський визначав *фрейм* як *мінімальний опис деякої сутності, такий, що подальше скорочення цього опису призводить до втрати цієї сутності.*

Розглянемо, наприклад, поняття "Студент"**,** яке описується відповідним фреймом. Кожний студент може бути охарактеризований такими характеристиками, як прізвище, ім'я та по-батькові, факультет, курс. Ці характеристики у фреймовій моделі зображуються *слотами*.

На основі фрейму "Студент" може бути описаний будь-який конкретний студент. Цей опис відбувається шляхом заповнення слотів конкретними значеннями.

Можна сказати, що *екземпляри понять* утворюються в результаті конкретизації *фреймів понять.*

Для фреймових моделей характерна *ієрархія понять*. У нашому прикладі фрейм "Студент" наслідує слоти від більш загального фрейму "Людина". Тому, якщо фрейм "Людина" описаний, відповідні слоти в описі фрейму "Студент" можна не задавати. Якщо ж якісь слоти уже заповнені конкретними значеннями, то ці значення можуть успадковуватися фреймами-нащадками.

За допомогою фреймів можуть описуватися і більш складні об’єкти. Розглянемо, наприклад, опис аудиторії на основі фрейму "Аудиторія":

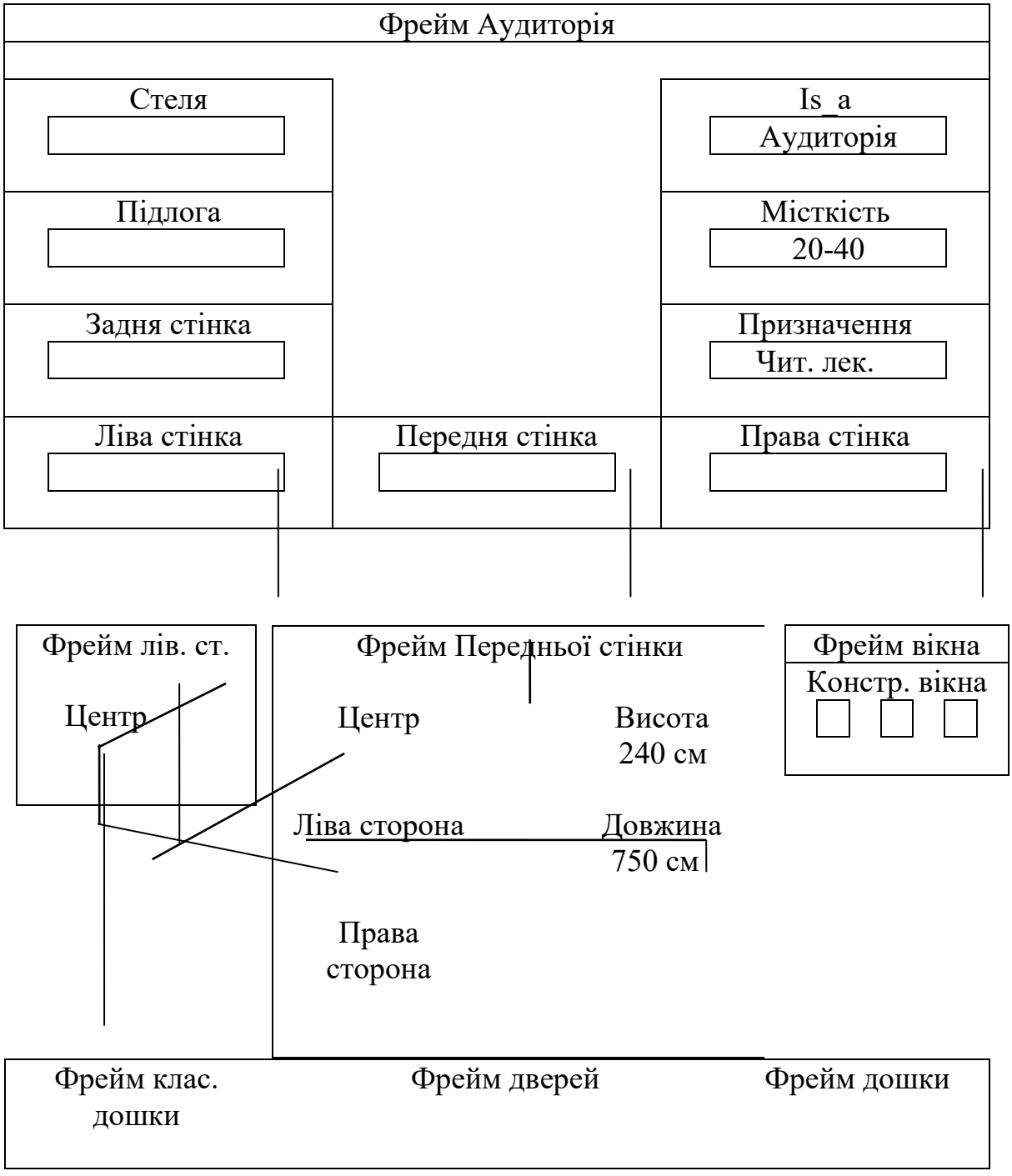


Рис. 4.5. Фрейм аудиторії

На рис. 4.5 фактично зображена мережа частково конкретизованих фреймів, пов'язаних між собою зв'язком **Has_Part**. Взагалі, фрейми можуть об'єднуватися в мережі, і тому їх часто розглядають в одному контексті з семантичними мережами [1].

Один і той самий предмет може описуватися різними фреймами, які відповідають різним умовам спостерігання цього предмета. Риси, спільні для цих різних фреймів, описуються *базовим фреймом*. Можна вважати, що фрейми, які відповідають різним умовам спостереження, успадковують основні характеристики базового фрейму. Зрозуміло, що зберігати ці похідні фрейми в пам'яті не обов'язково.

При пошуку фреймів, які найбільш підходять, велике значення мають мережі подібності, відмінності та ін. Як приклад можна навести відому *мережу подібності* за Уїнстоном [10]:

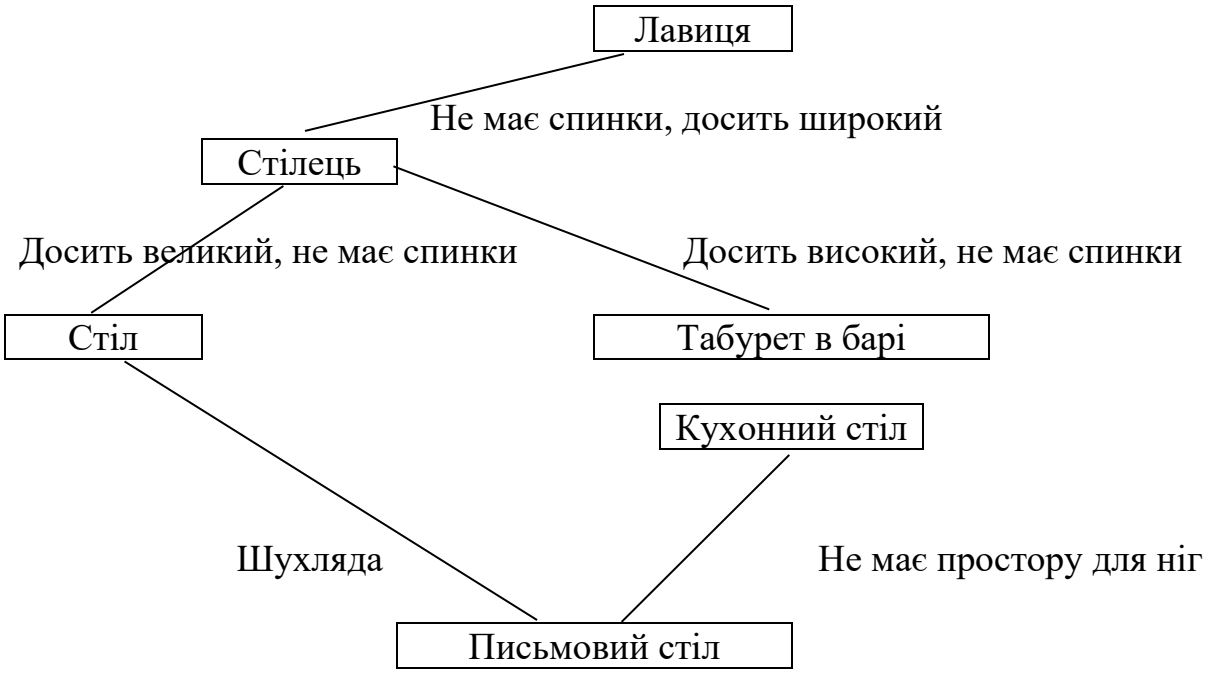


Рис. 4.6. Мережа подібності за Уінстоном

Існують різні моделі представлення знань за допомогою фреймів. Фрейм може мати, наприклад, таку структуру:

Ім'я Фрейму	Вказівник наслідування	Вказівник атрибутів	Значення слоту	Демони
Слот 1				
Слот 2				
……				
Слот *n*				

Рис. 4.7. Структура даних фрейму

Далі будуть пояснені основні елементи цієї структури.

1) Ім'я фрейму. Цей ідентифікатор є єдиним в даній фреймовій системі.

Кожний фрейм складається з довільного числа слотів (декілька з них системні, решта — користувача). Традиційно перелік основних системних слотів такий:

a) слот Is_a, що вказує на фрейм-предок даного фрейму;

b) слот-вказівник фреймів нащадків, який є списком вказівників цих фреймів;

c) слот для введення імені користувача, дати визначення, дати зміни, тексту коментаря;

d) інші слоти.

Кожний слот в свою чергу задається фіксованою структурою даних.

2) Ім'я слоту — ідентифікатор, унікальний в фреймі. Деякі імена слотів мають смислове навантаження:

a) Is_a(визначає тип відношення);

b) DESCENDANTS (вказівник прямого дочірнього фрейму);

c) DEFINED_BY (користувач, який визначає фрейм);

d) DEFINED_ON (дата визначення фрейму);

e) MODIFIED_ON (дата модифікації фрейму);

f) COMMENT — коментар;

g) Has_Part, RELATIONS — системні і використовуються при редагуванні бази знань і керуванні виведенням.

3) Вказівники наслідування використовуються тільки в фреймових системах ієрархічного типу, що ґрунтуються на відношеннях "Is_a". Вони вказують, яку інформацію про атрибути слотів фрейму вищого рівня наслідують слоти з таким же ім`ям в фреймі нижчого рівня.

Типові вказівники наслідування:

a) UNIQUE (U: унікальний) — показує, що кожний фрейм може мати слоти з різними значеннями;

b) SAME (S: такий же) — всі слоти повинні мати однакові значення;

c) RANGE (R: встановлення меж) — значення слотів нижчого рівня повинні бути в межах, вказаних значеннями слотів фрейма верхнього рівня;

d) OVERRIDE (O: ігнорувати) — при відсутності вказівки значення слотів фрейму верхнього рівня становиться значенням слота фрейму нижчого рівня, але у випадку визначення нового значення , значення слотів нижчого рівня вказуються в якості значень слотів. O виконує одночасно функції вказівників U і S.

4) Визначення типу даних — вказується, що слот має числове значення, або виступає вказівником іншого фрейму.

5) Значення слоту. Поле для введення значення слоту. Значення слоту повинно співпадати з вказаним типом даних цього слоту, і , крім того, повинна виконуватись умова наслідування.

6) Демон. Тут дається визначення *демонів* типу **if_needed, if_added, if_removed.** *Демоном* називається процедура, яка автоматично запускається при виконанні деякої умови. Демони запускаються при зверненні до відповідного слоту. Наприклад, демон **if_needed** запускається, якщо в момент звернення до слоту його значення не було встановленно, **if_added** — при підстановці значення в слот, **if_removed** — при стиранні значення слоту.

Демон можна вважати різновидом **приєднаної процедури**, які також можна включати до опису фреймів. Приєднана процедура запускається за повідомленням, переданим з іншого фрейму.

Відомі спеціалізовані мови для представлення знань фреймами, такі, як KRL та

ін.

Часто фреймові моделі ще називаються **об’єктними**.

Загальна форма об’єктного представлення визначена в [10] як

$$\text{Об’єкт}(\text{атрибут}_j, \text{значення}_j)(j = 1, \ldots, m).$$

4.3. Зв’язок між семантичними мережами та фреймами

На сучасному етапі фреймові моделі та семантичні мережі розглядають, як правило, у спільному контексті. Для цього є глибокі причини.

З одного боку, ніщо не заважає розглядати вузли семантичної мережі як фрейми з власною внутрішньою структурою.

З іншого боку, можна вводити різноманітні зв’язки між слотами фреймів. Тоді фрейм набуває рис семантичної мережі.

Глибокий аналіз об’єднаної мережно-фреймової моделі знань може, очевидно, принести чимало користі при аналізі глибинних структур знань та взаємозв’язків між різними інформаційними одиницями.

5. ЛОГІЧНІ МОДЕЛІ. ЛОГІЧНЕ ПРОГРАМУВАННЯ

5.1. Логічна модель знань

Логічна модель задання знань базується на формалізмах логіки предикатів першого порядку. Позитивною характеристикою логічних моделей є однозначність теоретичного обґрунтування і можливість реалізації системи формально точних визначень і висновків; недолік — формальний процедурний стиль мислення. Людська логіка — це інтелектуальна модель з нечіткою структурою. В цьому полягає її відмінність від строгої логіки.

Логічною моделлю називається формальна система, що задається четвіркою <*T*, *P*, *A*, *B*>**.** Тут *T* — це множина базових елементів, *P* **—** множина синтаксичних правил; *A* **—** множина аксіом, *B* **—** множина правил виведення.

5.2. Основні поняття мови Пролог

Пролог є **мовою логічного програмування**. Парадигма логічного програмування означає [1], що *для виконання програм використовується механізм доведення теорем*, який дозволяє з'ясувати, чи містяться суперечності у деякому наборі логічних формул. Сама програма розглядається як набір логічних формул разом з теоремою, яка повинна бути доведена.

Пролог — це мова програмування, *призначена для обробки символьної нечислової інформації*. Особливо добре Пролог пристосований для вирішення завдань, в яких фігурують об'єкти і зв'язки між ними.

Програмування на мові Пролог включає наступні етапи:

- опис *фактів* про об'єкти та відношення між ними;
- визначення *правил* про об'єкти та відношення між ними;
- формулювання *питань* (запитів) про об'єкти та відношення між ними;

5.2.1. Факти

Припустимо, потрібно повідомити системі Пролог факт: "Джону подобається Мері". Цей факт містить два об'єкта та відношення "подобається". У мові Пролог використовується наступна форма запису фактів:

```
likes(john, mary).
```

Потрібно дотримуватися наступних правил: *імена усіх відношень та об'єктів мають починатися з малої літери*. Спочатку записується ім'я відношення. Потім у дужках через кому записуються імена об'єктів. Кожний факт *має завершуватися крапкою*. *Порядок імен об'єктів є суттєвим*.

Розглянемо наступний набір фактів:

```
likes(john, mary). %John likes Mary
likes(john, book).
likes(john, fish).
likes(mary, book).
valuable(gold).
female(jane).
owns(john, gold).
father(john, mary).
```

```
food(butter).
gives(john, book, mary).
```

Кожний раз використання деякого імені має на увазі деякий індивідуальний об'єкт, який має інтерпретуватися однаково у всій базі знань.

Ім'я відношення, яке розташовується перед дужками, називається *предикатом*, а кількість його аргументів — *місністю* (арністю) предиката.

Сукупність фактів у Пролозі називається *базою даних*.

5.2.2. Запити

Після визначення сукупності фактів можна звертатися до системи Пролог з питаннями про задані відношення. Ці питання записуються так само, як факти, але перед ними ставиться спеціальний знак "?–".

Розглянемо питання

```
?- likes(mary, book).
```

Сенс запитання полягає у перевірці того, чи має Мері задану конкретну книгу (ми не запитуємо, чи має вона усі книги, чи які-небудь книги).

Система Пролог *сприймає запити як цілі, які потрібно досягти*.

Звертання до Прологу із запитом ініціює процедуру пошуку у базі даних, завантажених попередньо у систему. Система Пролог шукає в базі даних факти, *зіставні* з фактом у питанні. Два факти зіставні (або відповідають один іншому), якщо їх предикати однакові та їх відповідні аргументи співпадають. У разі успіху повертається значення `Yes` або `true` (в залежності від реалізації).

5.2.3. Змінні

Складніші запити можна отримати використовуючи змінні. Для знаходження усього, що подобається Джону, можна сформулювати питання "Чи подобається Джону *X*?" Це питання записуються у Пролозі так:

```
?– likes(john, X).
```

Це питання інтерпретується як "Існує що-небудь, що подобається Джону?".

Для позначення змінних у Пролозі використовуються ідентифікатори, які починаються із великої літери.

У процесі пошуку факти у БД переглядаються послідовно у порядку їх введення. **При цьому неконкретизована змінна вважається зіставною з будь-яким аргументом предиката, розташованим у відповідній позиції**. У процесі пошуку *знаходяться усі значення X, які дозволяють досягти мету*. Після знаходження зіставного факту змінна X конкретизується і в базі даних спеціальним маркером позначається місце, у якому відбулося співставлення. Це робиться для того, щоб мати можливість знайти наступні значення X, які можуть привести до задоволення мети. Лексична область визначення змінної — одне речення.

Якщо значення деяких змінних не використовується у програмі, то замість них можна використовувати *анонімні змінні*, які записуються як *символ підкреслювання*:

```
gives(john, _, mary).
```

Кожний раз при появі символу підкреслювання він позначає *нову* анонімну змінну.

Питання до системи може містити кілька цілей. Для того щоб перевірити, чи подобаються Джон та Мері один одному, треба зробити запит

```
likes(john, mary), likes(mary, john).
```

Якщо *цілі відокремлені комою*, то вважається, що перед системою ставиться завдання досягти *кон'юнкцію цілей*, причому запит опрацьовується зліва направо. Кожна ціль має свій маркер, які дають змогу здійснювати *backtracking* (пошук з поверненням).

Якщо цілі *відокремлює крапка з комою*, то система шукає *диз'юнкцію цілей*.

Для відстеження маркерів цілей можна використати команду `trace`.

5.2.4. Визначення відношень за допомогою правил

У Пролозі для того, щоб вказати, що деякі відношення визначаються на основі інших відношень, використовуються правила. Правило — це деяке *твердження про об'єкти та про зв'язки між ними*. Між фактами та правилами існує важлива відмінність. Факти є логічними твердженнями, які є безумовно істинними. З іншого боку правила — твердження, які є вірними при виконанні деяких умов.

У Пролозі правила складаються із *висновку* (*заголовку* або *голови*) та *тіла* (умови), які відокремлюються за допомогою символу :—. Символ ':—' читається *якщо*. Тіло складається із списку цілей, відокремлених комами, які розглядаються як знаки кон'юнкції. Факти та правила, які задають предикат, називаються *твердженнями*.

Припустимо, ми хочемо сформулювати твердження про те, що Джону подобаються усі жінки.

```
likes(john, X) :- female(X).
```

Кожне входження змінної у правило позначає один і той самий об'єкт. При формулюванні правил можна використовувати кон'юнкцію чи диз'юнкцію фактів інших правил.

```
likes(john, X) :- female(X), young(X).
likes(john, X) :- female(X); food(X).
```

Приклад. Розглянемо правило: "Людина може вкрасти предмет, якщо ця людина злодій, їй подобається цей предмет та предмет є цінним".

```
can_steal(Person, Object) :- thief(Person),
        likes(Person, Object), valuable(Object).
```

Фрази мови Пролог відносяться до трьох категорій: факти, правила та питання. Факти — це правила з порожнім тілом, питання — правила без голови.

5.2.5. Рекурсивні правила

Розглянемо двомісний предикат `has_part` для опису відношення агрегації (позначення того факту, що складовою частиною першого аргументу є другий аргумент). Опишемо за допомогою цього предикату наступні факти:

```
has_part(car, wheel).
has_part(car, engine).
has_part(bike, wheel).
has_part(car, door).
```

```
has_part(door, handle).
has_part(bike, engine).
has_part(window, glass).
has_part(door, glass).
```

Тоді можна визначити відношення `contains` для перевірки належності об'єктів до складу інших об'єктів:

```
contains(Aggregate, Detail) :- has_part(Aggregate, Detail).
contains(Aggregate, Detail) :- has_part(Aggregate, Part),
                               contains(Part, Detail).
```

Завдання для самостійної роботи. Написати правила для

1) Двомісного відношення `in_same_aggregate(X, Y)` (`X` та `Y` входять до складу того самого агрегату).
2) Одномісного відношення `in_different_aggregate(X)` (`X` входить до складу різних агрегатів).

5.3. Об'єкти даних у Пролозі

З точки зору синтаксису усі об'єкти даних у Пролозі є термами. Об'єкти даних поділяються на *прості об'єкти* та *структури*. *Прості об'єкти* діляться на *константи* та *змінні*. *Константи* бувають двох типів: *атоми* та *числа*.

Структура — це єдиний об'єкт, який складається із сукупності інших об'єктів, які називаються компонентами. Для з'єднання компонентів структури використовуються *функтори*.

Приклад описання структур:

```
likes(mary, book('White fang', 'Jack London')).
segment(point(2,3), point(5,6)).
segment(point(2,3), point(1,1)).
segment(point(1,6), point(1,1)).
segment(point(1,1), point(5,6)).
vertical(segment(point(X, _), point(X, _))).
```

Завдання для самостійної роботи. Написати правила для

1) відношення перпендикулярності двох відрізків;
2) перевірки належності трикутника до одного із наступних типів: прямокутні, гострокутні, тупокутні.

5.4. Основні операції Прологу

5.4.1. Рівність і встановлення відповідності

У Пролозі визначений *предикат рівність* "=", який використовується для узгодження термів. У випадку узгодження із базою даних цілі

```
? — X = Y.
```

Пролог робить спробу зробити зіставлення (встановити відповідність між X та Y). Цільове твердження вважаться доведеним, якщо така відповідність існує. При узгодженні з базою даних цілей вигляду `X = Y`, де X та Y — довільні терми, в яких можуть входити неконкретизовані змінні, використовуються наступні правила:

- якщо X неконкретизована змінна, а Y — конкретизований терм (який має певне значення), то X та Y узгоджуються. Крім того, змінні X також стає конкретизованою — вона набуває того самого значення, що і Y.
- Дві однакові константи (цілі числа або атоми) завжди рівні (узгоджуються) між собою.
- Дві структури рівні, якщо вони мають однаковий функтор і однакову кількість аргументів, причому відповідні аргументи рівні. Наприклад, наступна ціль буде успішно досягнута і змінна X буде конкретизована значенням bicycle.

```
rides(student, bicycle) = rides(student, X).
```

Цільове твердження `X = Y` є завжди істинним*, якщо один із аргументів неконкретизований*.

Предикат "`\=`" відповідає відношенню "не рівне". Ціль `X \= Y` досягається тоді і тільки тоді, коли не можна довести твердження `X = Y`.

Для *перевірки ідентичності* двох термів використовується операція "==", яка приймає істинне значення, якщо обидва операнди ідентичні (неконкретизована змінна рівна сама собі та усім змінним, зчепленим із нею попередніми операціями "=").

Для перевірки на "неідентичність" використовується відношення "`\==`".

5.4.2. Арифметичні операції

У Пролозі для виконання арифметичних операцій треба зробити окрему вказівку. Наприклад, наступний запит

```
?- X = 1 + 2.
```

не приведе до присвоювання змінній X значення 3.

Для обчислення значень виразів використовується інфіксний оператор `is`. Його правий аргумент — терм, значення якого треба обчислити. Наприклад:

```
?- X is 1 + 2.
```

Результат обчислень перевіряється на відповідність із лівим аргументом. Якщо він неконкретизований, то змінна конкретизується результатом обчислень. Основні операції:

```
+,  -,  *,  /,  //, **, mod.
```

5.4.3. Операції порівняння

Операції порівняння так само як операції **is** також змушують систему Пролог виконати арифметичні обчислення (у своїх обох аргументах!).

Для перевірки нерівності "менше або рівне" використовується символ "=<".

`X =:= Y`— операція перевірки рівності значень X та Y.

`X =\= Y` — операція перевірки нерівності значень X та Y.

Операція перевірки рівності "=:=" відрізняється від операції "=", оскільки *вона не спричиняє конкретизацію змінних*, а лише *приводить до обчислення значень виразів для обох аргументів* та їх порівняння.

Операція перевірки рівності відрізняється від операції "==", оскільки "==" не викликає попереднє обчислення значень своїх аргументів.

Для лексикографічного порівняння термів використовуються операції `@<`, `@>`, `@=<` та `@>=`. Наприклад, вони можуть використовуватися для порівняння дат:

```
date(2022, 12, 12) @> date(2022, 10, 15)
```

або

```
2022-12-12 @> 2022-10-15.
```

Ці операції використовують "стандартний порядок термів":

1) `Variables < Numbers < Atoms < Compound Terms`.
2) Змінні впорядковуються за адресою.
3) Числа порівнюються за значенням (при порівнянні рівних значень типів `integer` та `float`, наприклад `2` та `2.0`, значення типу `float` вважається меншим).
4) Атоми порівнюються за абеткою.
5) Для структур спочатку порівнюються їх арності, далі функтори і, нарешті їх аргументи зліва направо.

Для перевірки того, чи потрапляє цілочислове значення у заданий проміжок, можна використовувати предикат `between(+Low, +High, ?Value)`.

5.4.4. Заперечення як недосягнення мети

Згідно стандарту Пролога унарний предикат `\+` визначений таким чином, що виклик `\+goal` повертає істинне значення, якщо ціль `goal` є недосяжною. Наприклад

```
animal(dog).
animal(cobra).
animal(rabbit).
snake(cobra).
likes(mary, X):-
    animal(X), \+snake(X).
```

Приклад. Предикат `likes_to_only_John` задовольняють лише ті аргументи, які подобаються лише Джону.

```
likes_to_only_John(X) :-
      likes(john,X), \+ (likes(Y, X), Y \== john).
```

При використанні оператора `\+` потрібно враховувати гіпотезу про замкненість світу, яка приймається в системах ШІ. Наприклад, ввівши запит

```
?- \+ human(mary).
```

отримаємо відповідь `true`. (у випадку, якщо відношення `human` відоме системі і факт `human(mary)` відсутній). Цю відповідь потрібно сприймати не як твердження про те, що Мері не є людиною. Вона лише вказує на те, що у системі нема достатньої інформації для того щоб визначити є Мері людиною чи ні.

Розглянемо наступні факти та правила:

```
good_quality(samsung).
good_quality(apple).
```

```
expensive(apple).
moderate(X) :- \+ expensive(X).
```

На запит

```
?- good_quality(X), moderate(X).
```

система Пролог знайде відповідь `X = samsung`. Якщо ж переставити цілі:

```
?- moderate(X), good_quality(X).
```

то отримується відповідь `false`.

Відмінність полягає у тому, що у першому запиті перед спробою досягнення цілі `moderate(X)` змінна `X` вже є конкретизованою, а у другому — ні. Запит `\+expensive(X)` інтерпретується наступним чином:

`не` існує такий `X`, що досягається ціль `expensive(X)`)

Це рівносильно твердженню

Для всіх `X` твердження `expensive(X)` є хибним.

Завдання для самостійної роботи. Написати правило для відношення `bi_likes(X)`: об'єкт `X` подобається рівно двом особам.

5.5. Списки

5.5.1. Поняття списку

Список — послідовність, яка складається із довільної кількості елементів.

У Пролозі непорожній список складається із двох компонентів:

1) перший елемент (голова списку);

2) решта елементів (хвіст).

Хвіст списку сам є списком.

Для позначення порожнього списку використовується спеціальний *атом* `[]`.

Наприклад, для списку `ann, tennis, tom, skiing` головою є `ann`, а хвостом — список `tennis, tom, skiing`.

Список як і усі інші структуровані об'єкти мають ієрархічну (деревоподібну) структуру, наведену на рис. 5.1.

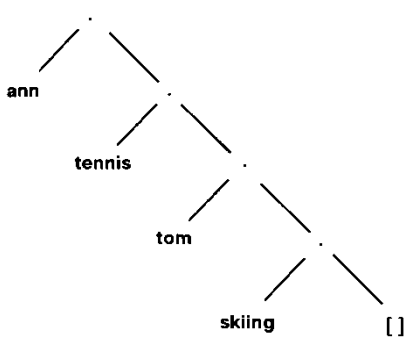


Рис. 5.1. Деревоподібне зображення списку

Для подання списків використовується наступна форма запису:

```
[ann, tennis, tom, skiing]
```

Крім того, передбачена ще одна форма запису — вертикальна риска для відокремлення голови списку від хвоста. Наприклад

```
[a, b, c] = [a | [b, c]] = [a, b | [c]] = [a, b, c | []]
```

5.5.2. Деякі операції із списками

1. Добавлення елемента у початок списку.

```
add_front(X, L, [X|L]).
```

2. Перевірка належності елемента до списку.

```
item(X, [X|_]).

item(X, [_|L]):- item(X, L).
```

3. Конкатенація списків.

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).
```

4. Знаходження останнього елемента списку (2 способи).

```
last([X], X).
last([_|L], Y) :- last(L, Y).
last2(L, X) :- conc(_, [X], L).
```

5. Видалення елемента із списку.

```
del(_, [], []).
del(X, [X|L], L).
del(X, [Y|L], [Y|L1]):- del(X, L, L1).
```

6. Підсписок

```
sub_list(L1, L2) :- conc(_, L1, L3), conc(L3, _, L2).
```

7. Знаходження найбільшого елемента числового списку

```
max([X], X).
max([X|L], X) :- max(L, MaxValue), X > MaxValue.
max([X|L], MaxValue) :- max(L, MaxValue), X =< MaxValue.
```

Для роботи із списками у SWI-Prolog можна використовувати бібліотечні предикати `member(?Elem, ?List)`, `append(?List1, ?List2, ?List1AndList2)`, `length(?List, ?Len)` та `select(?Elem, ?List1, ?List2)`.

Для сортування списків використовуються предикати `sort(+List, -Sorted)` та `msort(+List, -Sorted)`. Обидва вони використовують *стандартний порядок термів.* Перший предикат не включає дублікати у список `Sorted`, а 2-й — включає.

Задачі для самостійної роботи. Написати правила для предикатів `odd_length` — перевірка того, що список має парну довжину, `middle` — знаходження середнього елемента списку, `median` — знаходження медіани числового списку, `del_duplicate` — видалення повторних входжень елементів. Реалізація не має містити виклик бібліотечних предикатів обробки списків.

5.6. Керування перебором з поверненням

5.6.1. Відтинання

Нехай потрібно обчислити східчасту функцію, графік якої має вигляд

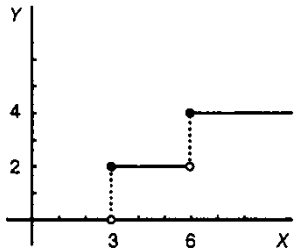


Рис. 5.2.

Залежність між x та $f(x)$ можна задати таким чином:

$$f(x)=\begin{cases}0, & x<3,\\ 2, & 3\le x<6,\\ 4, & 6\le x.\end{cases}$$

Для обчислення можна використати наступні правила Пролога:

```
f(X, 0) :- X < 3.          % Правило 1
f(X, 2) :- 3 =< X, X < 6.  % Правило 2
f(X, 4) :- 6 =< X.         % Правило 3
```

Проаналізуємо виконання наступного запиту:

```
?- f(1, Y), 2 < Y.
```

При спробі досягнення першої цілі, `f(1, Y)`, змінна `Y` конкретизується значенням 0. Тому друга ціль приймає вигляд $2 < 0$. Дана ціль недосяжна, а отже, а тому і весь запит завершується неуспіхом. Трасування виконання запиту наведене на рис. 5.3.

Оскільки правила 1-3 є попарно несумісними, то не може бути досягнуто ціль більше одного разу. Але система Пролог не має про це інформацію. У прикладі, наведеному на рис. 5.3, ціль правила один буде досягнуто у точці, позначеній як "Оператор відтинання". Тому потрібно повідомити систему Пролог про те, що не потрібно виконувати безперспективний перебір інших варіантів. Це завдання вирішується із використанням *оператора відтинання*, який записується як "`!`" і вставляється між цілями як своєрідна *псевдоціль*. Після використання оператора відтинання, програма стає такою:

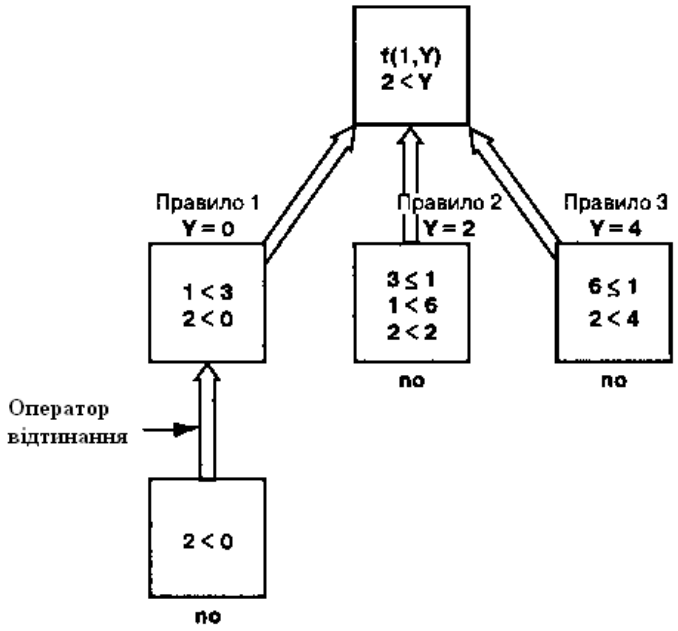


Рис. 5.3

```
f(X, 0) :- X < 3, !.
f(X, 2) :- 3 =< X, X < 6, !.
f(X, 4) :- 6 =< X.
```

Тепер символ ! запобігає перебору з поверненням. Отримана версія програми є більш ефективною, оскільки пошук цілі по середній та правій гілкам вже не ведеться.

Припустимо, що зроблений наступний запит:

```
?- f(7, Y).
Y = 4
```

При пошуку розв'язку була виконана послідовна перевірка усіх трьох правил. При цьому після виконання першої перевірки 7 < 3 перевірка умови 3 =< 7 є зайвою. Те саме стосується цілей 7 < 6 та 6 =< 7. Тому відповідні зайві перевірки можуть бути усунуті. Це дає змогу отримати наступну версію програми.

```
f(X, 0) :- X < 3, !.
f(X, 2) :- X < 6, !.
f(_, 4).
```

Слід зазначити, що *видалення операторів відтинання у останній програмі приводить до нової програми, яка вже не є еквівалентною до попередніх*, оскільки здатна знаходити кілька розв'язків.

Опис механізму відтинань:

Для прикладу розглянемо правило

```
h :- b1, b2, ..., bm, !, ... , bn.
```

яке викликається ціллю g, яка узгоджується із ціллю h. До моменту виконання оператора ! система вже знайшла деякий розв'язок, який відповідає цілям b1, b2, ..., bm. Після виконання оператора відтинання ці поточні значення *заморожуються*, а усі можливі альтернативи відкидаються. Крім того, ціль g *стає фіксованою*. Усі наступні спроби узгодити ціль g з головою якого-небудь іншого правила *не проводяться*.

Застосуємо правила механізму відтинань для аналізу наступного прикладу:

```
c :- p, q, r, !, s, t, u.
c :- v.
a :- b, c, d.
?- a.
```

Перебір з поверненням можливий для списку цілей p, q, r, але після досягнення ! усі альтернативи у цьому списку відкидаються. Альтернативне правило c :- v відносно c також відкидається. Але перебір з поверненням у межах списку цілей s, t, u усе ще можливий. Ціль c входить у правило a. Оператор відтинання впливає лише на ціль c. З позиції цілі a він залишається "непоміченим". Таким чином автоматичним перебір з поверненням у списку цілей b, c, d є активним, незважаючи на оператор відтинання. Вплив оператора відтинання на виконання програми проілюстровано на рис. 5.4.

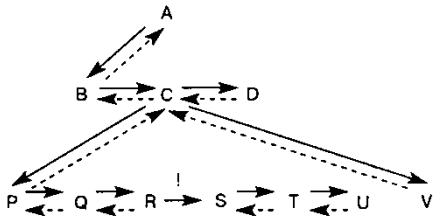


Рис. 5.4.

Ілюстрація прикладу:

```
c(X) :- p(X), !, s(X).
c(X) :- v(X).
a(X) :- b(X), c(X), d(X).
p(1). p(2).
s(2).
v(1). v(2). v(3).
b(1). b(2). b(3).
d(1). d(3).
```

Відповіддю на запит a(X) буде X = 3.

Якщо змінити правило a(X) на

```
a(X, Y) :- b(X), c(Y), d(Y).
```

то ціль a(X, Y) є недосяжною, ціль a(X, X) — досяжною. Причиною є те, що ціль c(Y) недосяжна у випадку вільної змінної Y, але ціль c(3) досяжна.

Для реалізації аналога умовного оператора `If-Then-Else` процедурних мов програмування використовується предикат "->". Він має форму `Condition -> Action`. Дія цього предиката може бути описана за допомогою правил:

```
If -> Then; _Else :- If, !, Then.
If -> Then; Else :- !, Else.
If -> Then :- If, !, Then.
```

Потрібно зазначити, що згідно до стандарту ISO (`If -> Then`) діє як

```
(If -> Then ; fail.)
```

Тому виклик, наприклад, предиката

```
g(X,Y):- (X > 2 -> Y = 1), Y = 2.
```

завершується неуспіхом при усіх числових значеннях аргументу `X`.

Як і операції "`,`" та "`;`" предикат "`->`" має ліву асоціативність.

Крім того, `A, B -> Then; Else` інтерпретується як ((`A, B) ->` `Then); Else`.

На відміну від `!`, будучи частиною складних цілей предикат `->` не «заморожує» значення змінних, які були конкретизовані при досягнені попередніх підцілей. Розглянемо такий фрагмент програми:

```
a(2). a(3).
b(3).
c(2). c(3).
try1(X) :- c(X), a(X) -> b(X).
try2(X) :- c(X), (a(X) -> b(X)).
try3(X) :- c(X), (a(X), !, b(X)).
```

Тоді виклик цілей `?- try1(X)` та `?- try3(X)` завершиться неуспіхом, а виклик цілі `try2(X)` є успішним:

```
?- try2(X).
X = 3.
```

оскільки для підцілі `c(X)` у правилі `try2` пошук з поверненням є можливим.

5.6.2. Приклади використання оператора відтинання

1. Обчислення максимуму двох чисел.

   Правило обчислення максимуму двох чисел

   ```
   max(X, Y, X) :- X >= Y.
   max(X, Y, Y) :- X < Y.
   ```

   з використанням оператора відтинання може бути переписано так

   ```
   max(X, Y, X) :- X >= Y, !.
   max(_, Y, Y).
   ```

Модифіковане правило є коректним лише у випадку його використання у цілях `max(X, Y, Max)` із неконкретизованою змінною Max. Інакше можливою є помилка. Так, наприклад, ціль `max(3, 1, 1)` є досяжною. Вказане обмеження можна подолати із використанням наступної форми max:

```
max(X, Y, Max) :-
     (   X >= Y, !,
         Max = X
     ;   Max = Y
     ).
```

2. Класифікація з розбиттям за категоріями.
   Розглянемо класифікацію учасників турніру з тенісу. Результати зберігаються у вигляді фактів:

```
beat(tom, jim).
beat(ann, tom).
beat(pat, jim).
```

Необхідно визначити відношення `class(Player, Category)` для розбивки гравців за трьома категоріями:

`winner` — переміг у всіх зустрічах,
`fighter` — має як перемоги так і поразки,
`looser` — програв усі зустрічі.

Відповідне словесне формулювання має вигляд:

Якщо X кого-небудь переміг та X був ким-небудь переможений,
  то X — `fighter`,
у протилежному випадку, якщо X когось переміг,
  то X — `winner`,
в протилежному випадку, якщо X був ким-небудь переможений,
  то X — `looser`.

Це розбиття можна описати з використанням відтинання наступним чином

```
class(X, fighter):- beat(X, _), beat(_, X), !.
class(X, winner) :- beat(X, _), !.
class(X, looser) :- beat(_, X), !.
```

Виклик `class` є безпечним, якщо другий параметр неконкретизований. Недоліком реалізації є те, що за допомогою цілі `class(X, Y)` з двома вільними змінними можна знайти лише один розв'язок.

Задача. Написати предикат `split(Numbers, Positives, Others)` для розбиття списку на відповідні підсписки.

5.6.3. Недосяжна ціль fail

Твердження "щось не є істинним" можна виразити з використанням цілі `fail`, спроба досягнення якої завжди є невдалою.

Розглянемо запис твердження "Мері подобаються усі тварини крім змій".

Для цього можна переформулювати твердження таким чином:

Якщо X — змія, то твердження "Мері подобається X" не є істинним.
В протилежному випадку, якщо X — тварина, то Мері подобається X.

Останнє твердження на Пролозі запишеться так:

```
likes(mary, X) :- snake(X), !, fail.
likes(mary, X) :- animal(X).
```

Компактніше це правило можна записати так:

```
likes(mary, X) :-
    (   snake(X), !,
        fail
    ;   animal(X)
    ).
```

Це саме відношення можна записати із використанням оператора `\+`.

```
likes(mary, X) :- animal(X), \+snake(X).
```

Ціль `fail` часто використовується у так званих "циклах керованих неуспіхом". Наприклад, ціль

```
between(2, 7, X), writeln(X), fail.
```

дає змогу вивести на усі цілі числа від 2 до 7 включно.

5.7. Додаткові вбудовані предикати Прологу

5.7.1. Перевірка типу термів

Предикат `number(X)` приймає істинне значення, якщо `X` — число або змінна, значенням якої є число. Цей предикат призначений для захисту цілі `Z is X + Y` від недопустимих аргументів.

Інші подібні вбудовані предикати:

`var(X)` — перевірка того, чи є `X` неконкретизованою змінною.

`nonvar(X)` — виконується успішно, якщо `X` — не змінна або уже конкретизована змінна.

`atom(X)` — перевірка того, чи є `X` атомом.

`integer(X), float(X)` — перевірка належності `X` до відповідних числових множин.

`atomic(X)` — приймає істинне значення, якщо `X` є числом або атомом.

`compound(X)` — приймає істинне значення, якщо `X` є структурою.

Приклад. Написати правило для заміни усіх неконкретизованих елементів списку на найбільший серед його числових елементів.

```
max_number([X|L], MaxValue) :-
    \+ number(X), !,
    max_number(L, MaxValue).
```

```
max_number([X|L], MaxValue) :-
    max_number(L, MaxValue),
    MaxValue > X, !.
max_number ([X|_], X).
```

% max_number працює правильно лише при виклику з вільною змінною MaxValue!

```
replace_var(InList, OutList) :- max_number(InList, MaxValue),
    rep_var(InList, MaxValue, OutList).
rep_var([], _, []).
rep_var([X|L1],Value,[Value|L2]):- var(X), !,
    rep_var(L1,Value,L2).
rep_var([X|L1], Value, [X|L2]) :- rep_var(L1,Value, L2).
```

Завдання для самостійного розв'язування:

1) Знайти розв'язок попередньої задачі, у якому використовується тільки один "прохід" елементів списку.
2) Написати реалізацію предиката `last_index(?Elem, ?List, ?Id)` для знаходження позиції останнього входження елемента у список. Предикат повинен коректно працювати і у тому випадку, якщо один або кілька його аргументів неконкретизовані, і проходити елементи списку один раз.

5.7.2. Створення та декомпозиція термів

1. Предикат =..

Предикат =.. записується як інфіксний оператор і читається "юнів" (*univ*). Ціль

```
Term =.. L
```

є істинною, якщо L — список, що містить головний функтор терму Term, за яким йдуть його параметри (компоненти). Приклади:

```
?- f(a, b) =.. L.
L = [f, a, b]
?- T =.. [rectangle, 3, 5].
T = rectangle(3, 5)
?- Z =.. [p, X, f(X,Y)].
Z = p(X, f(X,Y))
```

Розглянемо програму маніпулювання геометричними об'єктами, які зберігаються у програмі у вигляді термів, функтори яких позначають типи фігур, а параметри задають розмір фігури, як показано нижче.

```
square(Side)
rectangle(Side1, Side2)
triangle(Sidel, Side2, Side3)
```

```
circle(R)
```

Однією з можливих операцій над фігурами може бути зміна їх розміру за допомогою відношення

```
resize(Figure, Factor, ResizedFigure),
```

де `Factor` — коефіцієнт, який вказує у скільки разів треба збільшити розмір. Традиційний спосіб опису відношення `resize` наведений нижче:

```
resize(square(A), F, square(Al)) :- Al is F*A.
resize(circle(R), F, circle(Rl)) :- Rl is F*R1.
resize(rectangle(A,B), F, rectangle(Al,Bl)):- Al is F*A,
    Bl is F*B.
...
```

Такий підхід є громіздким для великої кількості фігур. Цю незручність можна усунути з використанням предикату =.. наступним чином:

```
resize(Fig, F, Fig1) :-
    Fig =.. [Type | Parameters],
    multiply_list(Parameters, F, Parameters1),
    Fig1 =.. [Type | Parameters1].
multiply_list([], _, []).
multiply_list([X | L], F, [X1 | L1]) :-
    X1 is F*X, multiply_list(L, F, L1).
```

Предикат =.. можна використовувати для програмного формування цілей, які потім можна виконувати із використанням `call`. Наступний предикат `call2` можна використовувати для виклику двомісних предикатів із заданими аргументами:

```
call2(Predicate, X, Y) :- Goal =.. [Predicate, X, Y],
    call(Goal).
```

Приклад виклику предиката call2:

```
?- call2(likes, john, X).
X = mary ;
X = meat ;
X = fish.
```

2. Предикати `functor` та `arg`

Предикат

```
functor(Term, F, N)
```

є істинним, якщо `F` — головний функтор терма `Term`, арність якого рівна `N`.

Ціль

```
arg(N, Term, A)
```

є істинною, якщо A — N-й параметр у термі Term за умови, що параметри термів-структур нумеруються зліва направо, починаючи з 1.

Приклад:

```
?- functor(t(f(X), X, t), Fun, Arity).
Fun = t
Arity = 3
?- arg(2, f(X, t(a), t(b)), Y).
Y = t(a)
?- functor(D, date, 3),
arg(1, D, 29),
arg(2, D, june),
arg(3, D, 1982).
D = date(29, june, 1982)
```

З використанням предикатів functor та arg можна моделювати масиви. Ціль

```
functor(A, f, 100)
```

створює структуру із 100 елементами:

```
A = f(_,_,_,... )
```

Аналогом присвоєння A[60] = 1, яке характерне для процедурних мов, може бути наступна ціль Пролога:

```
arg(60, A, 1)
```

У результаті виконання цієї цілі, структура A конкретизується таким чином:

```
A = f(_,...,_,1,_,...,_) % 60-й елемент рівний 1
```

Аналогом "процедурного" оператора X = A[60] буде

```
arg(60, A, X)
```

Проте у Пролозі відсутній простий аналог процедурного оновлення елементів масиву типу A[5] = A[5]+1.

Задача. Написати предикат atom_count для визначення кількості атомів у складі заданого терму. Наприклад, результатом запиту

```
?- atom_count(f(2 + g(X,3), h(a(1)* c), g(b)), C).
```

має бути C = 2.

5.7.3. Операції з базою даних

Базою даних у Пролозі вважається сукупність усіх визначених у Пролог-програмі фактів та правил. У мові Пролог передбачена можливість модифікації цієї бази під час виконання програм. Для цього призначені предикати asserta, assertz, retract.

Ціль

```
asserta(C)
```

завжди досягається і крім того заносить фразу (факт або правило) C у початок бази даних. Ціль assertz(C) аналогічна за винятком того, що C заноситься у кінець бази.

Ціль

```
retract(C)
```

виконує протилежну дію: видаляє фразу Пролога, яка узгоджується із C.

Згідно до стандарту asserta, assertz, retract можна застосовувати лише до предикатів, які описані як "динамічні". Динамічні предикати оголошуються за допомогою директиви :-dynamic із вказівкою їх арності. Наприклад, директива

```
:- dynamic parent/2, male/1, female/1.
```

описує динамічний двомісний предикат parent та одномісні предикати male, female.

Предикат retract є недетермінованим: за допомогою єдиної цілі retract можна видалити цілий набір фраз Пролога.

Приклад. Нехай у Пролог-програмі визначені факти

```
fast(ann).
slow(tom).
slow(pat).
```

До цієї програми можна додати нове правило faster:

```
?- asserta((faster(X,Y) :- fast(X), slow(Y))).
true.
?- faster(A, B).
A = ann
B = tom
?- retract(slow(X)).
X = tom;
X = pat;
false.
?- faster(ann, _).
false.
```

Розглянемо приклад застосування предикатів assertz та retract для знаходження списку унікальних імен батьків, які є першими аргументами відношення parent.

```
:-dynamic name/1.
get_parents(_) :- parent(X, _), \+name(X),
        assertz(name(X)), fail.
get_parents(List) :- extract_names(List).
extract_names([X|Names]) :- retract(name(X)), !,
        extract_names(Names).
```

```
extract_names([]).
```

Завдання для самостійного розв'язування. Написати предикат для знаходження списку усіх батьків, який не використовує базу даних чи вбудовані предикати `findall`, `bagof`, `setof`.

Розглянемо приклад формування таблиці множення:

```
:- dynamic product/3.
maketable :-
    L = [0,1,2,3,4,5,6,7,8,9] ,
    select(X, L, _), % Вибрати 1-й множник
    select(Y, L, _), % Вибрати 2-й множник
    Z is X*Y,
    assert(product(X,Y,Z)),
    fail.
```

Для формування таблиці треба виконати запит

```
?- maketable.
```

Після цього можна, наприклад, шукати числа, добуток яких рівний 24:

```
?- product(X, Y, 24).
X = 3,
Y = 8 ;
X = 4,
Y = 6 ;
X = 6,
Y = 4;
.....
```

Розглянемо приклад знаходження списку дітей для кожного із батьків:

```
children_list(_) :- parent(X, Y),
        (retract(name(X-C)) -> L = C; L = []),
        assertz(name(X-[Y|L])), fail.
children_list(List) :- extract_names(List).
?- children_list(L).
L = [tom-[bob, liz], pam-[bob], bob-[ann, pat], pat-[jim]].
```

Для видалення усіх фактів і правил, які узгоджуються із `Goal`, використовується предикат

```
retractall(Goal)
```

Приклад застосування: `?- retractall(name(_)).`
Перевагою предикату `retractall` над `retract` є те, що його можна використовувати для видалення динамічних правил.

Згідно зі стандартом мови, ціль яка полягає у виклику динамічного предиката, не "бачить" його модифікацій, які можуть бути спричинені викликами assert або retract, протягом життєвого циклу цілі. Це проілюстровано наступним фрагментом коду.

```
:- dynamic foo/1.
foo(1). foo(2).
```

Виклик цілі

```
? - foo(_X), format('X before retract ~w~n', [_X]),
    retract(foo(_Y)), format('Y after retract ~w~n', [_Y]).
```

призводить до наступного виводу

```
X before retract 1
Y after retract 1
true
Y after retract 2
true
X before retract 2
false
```

Тобто, перша підціль `foo(_X)` пам'ятає динамічний факт `foo(2)` навіть після його видалення! Таку поведінку стандарт пояснює міркуваннями безпеки.

5.7.4. Генерація списків. Предикати bagof, setof, findall

Ціль `bagof(X, G, L)` генерує список `L`, який складається з усіх об'єктів `X`, таких, що ціль `G` досягається.

Наприклад, для бази даних про родинні зв'язки можна знайти дітей Тома так:

```
?- bagof(X, parent(tom, X), L).
L = [bob, liz].
```

Якщо ім'я батька не вказано, то для кожного з батьків буде сформовано список дітей:

```
bagof(X, parent(Y, X), L).
Y = bob,
L = [ann, pat];
Y = pam,
L = [bob];
Y = pat,
L = [jim];
Y = tom,
L = [bob, liz, ann].
```

Можна також вказати, що потрібно сформувати один список без групування. Для цього використовується операція "^". Наприклад, якщо потрібно отримати список усіх батьків, то можна вказати пошуковій системі Пролог, що ім'я дитини не має значення:

```
?- bagof(Y, X^parent(Y, X), L).
L = [pam, bob, bob, tom, tom, pat].
```

Якщо один і той самий об'єкт знаходиться кілька разів, то він дублюється у списку. Якщо *відсутній розв'язок, то ціль* `bagof` *не досягається.*

Предикат setof(X, P, L) аналогічний до bagof за винятком того, що всі дублікати видаляються і результуючий список впорядковується (відповідно до вбудованого предиката @<). Наприклад

```
?- setof(Y, X^parent(Y, X), L).
L = [bob, pam, pat, tom].
```

Предикат findall(X, G, L) також продукує список об'єктів, які відповідають предикату G. Він відрізняється від bagof тим, що у список збираються усі об'єкти X незалежно від значення інших змінних. Наприклад

```
?- findall(Y, parent(Y, _), L).
L = [pam, bob, bob, tom, tom, pat].
```

Тобто findall еквівалентний до bagof з параметром ^. Але якщо жоден об'єкт X не відповідає предикату G, то *створюється порожній список і ціль досягається*. Крім того, на відміну від findall bagof коректно працює у випадку, коли ціль G задовольняють вільні змінні.

5.7.5. Функціональне програмування засобами Prolog

Предикат forall(:Condition, :Action) перевіряє ціль :Action для усіх результатів виклику цілі :Condition.

Приклад. Розглянемо предикат знаходження суми елементів числового списку:

```
:-dynamic sum/1.
sum_list(L, Sum) :- assertz(sum(0)),
   forall(member(X, L),(retract(sum(S)), T is S + X, assert(sum(T)))),
   retract(sum(Sum)).
```

Результат роботи:

```
?- sum_list([2,3,7], S).
S = 12.
```

Ціль max_member(:Pred, -Max, +List) досягається, якщо Max є найбільшим елементом списку List відповідно до відношення Pred, яке має мати 2 аргументи і бути аналогом @=</2. Для знаходження найбільшого елемента списку відповідно до "стандартного порядку термів" достатньо предиката max_member(-Max, +List). Аналогічним чином для знаходження мінімумів можна скористатися min_member з двома або трьома аргументами.

Задача. Знайти ті елементи, які зустрічаються у списку найчастіше.

Предикати бібліотеки apply

Виклик предиката maplist(:Goal, ?List) завершується успіхом, якщо ціль Goal може бути успішно застосована для усіх елементів списку List.

Приклад. Нехай у програмі визначений предикат

```
positive(X) :- X > 0.
```

Тоді для перевірки того, чи є додатними усі елементи списку, можна використати предикат maplist наступним чином:

```
?- maplist(positive, [1, 3, 2, 5]).
true.
```

Виклик предиката maplist(:Goal, ?List1, ?List2) завершується успіхом, якщо ціль Goal може бути успішно застосована для усіх пар відповідних елементів списків List1 та List2.

Приклад. Нехай у програмі визначений предикат

```
inc(X, Y) :- number(X), !, Y is X + 1.
inc(X, X).
```

Тоді для збільшення на 1 усіх числових елементів списку можна використати предикат maplist наступним чином:

```
?- maplist(inc, [2, a, -1, 5], L).
L = [3, a, 0, 6].
```

Виклик предиката maplist(:Goal, ?List1, ?List2, ?List3) завершується успіхом, якщо ціль Goal може бути успішно застосована для усіх трійок відповідних елементів списків List1, List2 та List3.

Наприклад, для знаходження поелементних сум двох числових списків можна використати ціль вигляду:

```
maplist(plus, [1,2], [3,4], L).
L = [4, 6].
```

При застосуванні maplist можна використовувати Goal з "пов'язаними" значеннями кількох перших аргументів. Наприклад, для збільшення на 3 елементів числового списку можна використати ціль вигляду:

```
maplist(plus(3), [1,4], L).
L = [4, 7].
```

Для формування списку із 10 одиниць можна використати ціль

```
length(L, 10), maplist(=(1), L).
```

Для генерації усіх булевих наборів довжини 4 можна використати ціль

```
length(L, 4), maplist(between(0,1), L).
```

Для передачі складніших цілей Goal у maplist потрібно використовувати лямбда-вирази вигляду {Free}/[Parameters] >> Goal (бібліотека yall). Наприклад, для знаходження поелементних добутків двох числових списків можна використати предикат:

```
memberwise_product(InList1, InList2, Product) :-
    maplist([X,Y,Z] >> (Z is X*Y), InList1, InList2, Product).
```

Предикат

```
include(:Goal, +List1, ?List2)
```

дає змогу фільтрувати елементи списку List1, включаючи у результуючий список List2 усі елементи початкового списку, які задовольняють ціль Goal.

Наприклад, підсписок Positives додатних елементів числового списку List можна отримати наступним чином:

```
include(<(0), List, Positives).
```

Предикат

```
partition(:Pred, +List, ?Included, ?Excluded)
```

розбиває список List на два підсписки Included та Excluded, перший з яких містить усі елементи X, які задовольняють ціль call(Pred, X), другий — усі інші елементи.

Наприклад, розбиття списку чисел L на підсписки непарних (L1) та парних (L2) елементів можна отримати так:

```
partition([X] >> (1 is X mod 2), L, L1, L2)
```

Предикат

```
foldl(:Goal, +List, +V0, -V)
```

робить "ліву" згортку списку List (застосовуючи до його елементів ціль Goal) із "початковим" значенням V0. Він діє наступним чином:

```
foldl(G, [X_1, ..., X_n], V0, V) :-
   call(G, X_1, V0, V1),
   call(G, X_2, V1, V2),
   ...
   call(G, X_n, V<n-1>, V).
```

Наприклад, сумарна кількість елементів в усіх списках, які містяться у списку списків Lists, може бути знайдена таким чином:

```
sum_len(Lists, Sum) :-
    foldl([L,Prev,S] >> (length(L, N), S is Prev + N), Lists, 0, Sum).
```

Приклад. Спростити предикат atom_count з використанням засобів функціонального програмування.

Предикати бібліотеки aggregate

Бібліотека aggregate надає засоби для здійснення агрегації множини розв'язків, які задовольняють певну ціль.

Предикат

```
aggregate(+Template, :Goal, -Result)
```

виконує агрегацію та "зв'язування" змінних у Goal відповідно до Template. Предикат aggregate застосовує bagof до Goal. У описі Goal можна використовувати операцію "^".

Предикат

```
aggregate_all(+Template, :Goal, -Result)
```

здійснює агрегацію та "зв'язування" змінних у `Goal` відповідно до `Template`. Предикат `aggregate_all` застосовує `findall` до `Goal`. У описі `Goal` можна використовувати анонімні змінні (_).

У предикатах `aggregate` та `aggregate_all` для задання режиму агрегації використовується параметр `Template`, який може містити терм, побудований з використанням наступних операцій:

`count` — кількість розв'язків.

`sum(Expr)` — сума значень `Expr` для всіх розв'язків.

`min(Expr)` — найменше значення `Expr` для всіх розв'язків.

`min(Expr, Witness)` — терм `min(Min, Witness)`, де `Min` є найменшим значенням `Expr` для всіх розв'язків, а `Witness` — довільний інший шаблон, застосований до розв'язку, який відповідає `Min`. У випадку, якщо мінімум досягається для кількох розв'язків, `Witness` відповідає *першому* такому розв'язку.

`max(Expr)` — найбільше значення `Expr` для всіх розв'язків.

`max(Expr, Witness)` — аналогічним чином до `min(Expr, Witness)`.

`set(X)` — впорядкована множина всіх розв'язків для `X`.

`bag(X)` — список всіх розв'язків для `X`.

Приклад. У базі даних Prolog є факти, які стосуються відношення

```
country(Name, Continent, Area, Population).
```

Написати правила для знаходження:

1) сумарної площі усіх країн;
2) списку пар континент-"найбільше населення країн континенту";
3) середньої площі країн;
4) назву та площу найменшої за площею країни;
5) списку назв країн, населення яких більше за середнє населення усіх країн того континенту, на якому вони знаходяться.

5.8. Застосування мови Пролог для розв'язування задач штучного інтелекту

5.8.1. Задача про Ханойську вежу

Задача про Ханойську вежу є класичним прикладом використання декомпозиційного методу розв'язування задач. На рис. 5.5 наведено задачу у випадку $n = 3$.

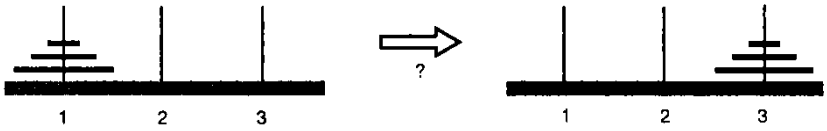


Рис. 5.5. Задача про Ханойську вежу

На вісь 1 нанизані n різних дисків у порядку спадання їх розмірів. Потрібно перенести усі диски на вісь 3, використовуючи при потребі вісь 2 у якості допоміжної осі. Дозволяється одночасно переносити лише один диск. При цьому не можна класти більший диск на менший.

Розв'язок задачі можна отримати за наступним алгоритмом:

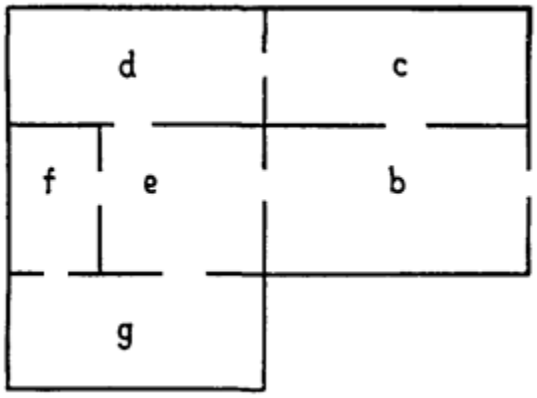
1. Перенести верхні $n-1$ диски з першого стержня на другий, використовуючи третій, як допоміжний.
2. Перекласти найбільший стержень з першої на третю вісь.
3. Перенести $n-1$ диски з другого стержня на третій, використовуючи перший, як допоміжний.

Реалізація на Пролозі:

```
hanoi_tower(N) :-
        move(N, left, middle, right).

write_list([]) :- nl, !.
write_list([H|T]) :- write(H), write(' '), write_list(T).


show_move(From, To, Number) :-
   write_list(['move disk', Number, 'from', From, 'to', To]).
%atomic_list_concat(['Move disk', Number, 'from', From, 'to', To], ' ', S).

move(1, From, _, To) :-
        show_move(From, To, 1), !.
move(N, From, Additional, To):-
        N1 is N - 1,
        move(N1, From, To, Additional),
        show_move(From, To, N),
        move(N1, Additional, From, To).
```

5.8.2. Задача про пошук у лабіринті

Потрібно знайти шлях, який веде від заданого початкового положення до виходу із лабіринту. Розглянемо приклад плану будинку, наведеного на рис. 5.6. Потрібно знайти вихід із заданої кімнати, наприклад `g`, назовні.



ТРис. 5.6. План будинку

Задамо фактами двомісне відношення `door`, яке вказує на кімнати, між якими є двері (`door(b, outside)` позначає зовнішні двері).

```
door(b, outside).
```

```
door(b, c).
door(b, e).
door(d, c).
door(d, e).
door(f, e).
door(g, e).
door(f, g).
```

При розробці Пролог-програми потрібно врахувати, що двері є симетричними. Крім того для зручності можна вважати `outside` додатковою кімнатою. Розглянемо програму із використанням списків. Ключовим предикатом є предикат

```
path(Start, Finish, Banned, Visited)
```

, який дає змогу знайти шлях від початкової до кінцевої кімнати, який не проходить через жодну із кімнат двічі. Для уникнення зациклення використовується список "заборонених" кімнат `BannedRooms`. Знайдений шлях записується у список `VisitedRooms`.

```
path(Start, Start, _, []) :- !.
path(Start, Finish, Banned, [Start|Visited]) :-
    (door(Start, Room); door(Room, Start)),
    \+member(Room, Banned),
    path(Room, Finish, [Start| Banned], Visited).
```

Основний предикат знаходження розв’язку:

```
exit_search(Start, Visited) :-
  path(Start, outside, [], Visited).
```

Приклад запиту:

```
?- exit_search(g, R).
R = [g, e, b] ;
R = [g, e, d, c, b] ;
R = [g, f, e, b] ;
R = [g, f, e, d, c, b] ;
false.
```

З використанням бази даних можна обійтися без використання списків. Для цього достатньо описати динамічний одномісний предикат `visited` для збереження у БД вже відвіданих кімнат. Відповідна реалізація має вигляд:

```
:-dynamic visited/1.
exit_search(Start, Mode, Rooms) :-
        path(Start, Mode, outside),
        findall(X, visited(X), Rooms);
        retractall(visited(_)).  % db clearing

path(Start, _, Start) :-!.
```

```
% nondeterministic version (search of all solutions)
path(Start, all, Exit) :-
        assertz(visited(Start)),
        (door(Start, NewRoom); door(NewRoom, Start)),
        \+visited(NewRoom),
        path(NewRoom, all, Exit);
        retract(visited(Start)), fail.
% clearing is needed for finding other solutions

% deterministic version (single solution search)
path(Start, single, Exit) :-
        assertz(visited(Start)),
        (door(Start, NewRoom); door(NewRoom, Start)),
        \+visited(NewRoom),
        path(NewRoom, single, Exit), !.
```

Другий параметр предикату `exit_search` (`single` або `all`) дозволяє здійснювати пошук одного або усіх шляхів без циклів.

5.8.3. Розв'язування числових ребусів

Приклад арифметичного ребуса:

```
  DONALD
+ GERALD
  ROBERT
```

Приклад виклику предиката для пошуку розв'язку (різним змінним відповідають різні цифри):

```
?- L1=[D,O,N,A,L,D], L2=[G,E,R,A,L,D], L3=[R,O,B,E,R,T], sum(L1,L2,L3).
      L1 = [5, 2, 6, 4, 8, 5],
      L2 = [1, 9, 7, 4, 8, 5],
      L3 = [7, 2, 3, 9, 7, 0].
```

Задача. Написати правила для розв'язування ребусів вигляду

а) `sum(L1,L2,L3)`, де списки можуть мати різну довжину;

б) `sum(ListOfOperands,Sum)`, де списки в `ListOfOperands` можуть мати різну довжину;

в) `product(L1,L2,L3)`, де списки можуть мати різну довжину.

5.8.4. Спрощення алгебраїчних виразів*

Розглянемо одну з найпростіших задач на спрощення алгебраїчних виразів: задачу зведення подібних доданків у сумі. Потрібно реалізувати предикат

```
simplify(InExpr, OutExpr),
```

який має проводити спрощення наступним чином:

```
?-simplify(x+2+z1+(x+4+(x+y+5+z1))+(y+(z1+x)+5), Expr).
   Expr = 16+2*y+3*z1+4*x.
```

Реалізація з використанням списків:

```
simplify(Expr1, Expr2) :- simp(Expr1, L), build_expr(L, Expr2).
simp(X+Y, MemberList) :- simp(X, L1), simp(Y, L2),
        combine(L1, L2, MemberList), !.
simp(X, [number-X]) :- number(X), !.
simp(X, [X-1]):- atom(X).

combine(L, [], L) :- !.
combine([], L, L) :- !.
combine([X-Y|L1], L2, [X-Coeff|L]) :- select(X-Z, L2, L3),
        !, Coeff is Y+Z, combine(L1, L3, L).
combine([X-Y|L1], L2, [X-Y|L]) :- combine(L1, L2, L).

build_expr([number-Coeff], Coeff) :- !.
build_expr([X-Coeff], Coeff*X) :- !.
build_expr([X|L], Expr2+Expr1) :- build_expr([X], Expr1),
        build_expr(L, Expr2).
```

Реалізація з використанням бази даних:

```
:-dynamic coeff/2, coeff/1.

simp(X+Y) :- !, simp(X), simp(Y).
simp(X) :- number(X), !, retract(coeff(Y)),
    Z is X+Y, asserta(coeff(Z)).
simp(X) :- atom(X), (retract(coeff(X, Y)), !; Y = 0),
    Z is Y+1, asserta(coeff(X, Z)).

build_expr(Expr+X*Atom) :- retract(coeff(Atom, X)), !,
    build_expr(Expr).
build_expr(X) :- retract(coeff(X)).
```

6. ПРОДУКЦІЙНІ МОДЕЛІ

6.1. Загальна характеристика продукційних моделей

Продукційною називається *модель знань*, в якій база знань складається із *продукцій,* тобто правил "Якщо *A*, тоді *B*". Термін "продукція" був введений американським математиком Е. Постом. Як і логічні, продукційні моделі насамперед концентруються на дедуктивному виведенні, але є менш формалізованими і тому більш наочними і зручними. На сучасному етапі найбільш вживаним формалізмом для задання знань в експертних системах є саме продукційні моделі.

Приклад 1. Розглянемо інформаційну експерту систему для зоологічного парку, яка повинна на основі заданого опису деякої тварини відповісти на питання, що це за тварина. Продукції такої системи можуть мати вигляд:

Якщо тварина є хижаком, має жовто-коричневий колір та темні смуги, то це — тигр.

Якщо тварина є птахом, не літає, плаває, має чорно-білий колір, то це — пінгвін.

Наведені продукції описують імплікації (з істинності *A* випливає істинність *B*).

Приклад 2. Нехай ідеться про систему керування деяким технічним процесом. Продукції можуть мати такий вигляд:

Якщо температура перевищує 200 градусів, ввімкнути систему охолодження.

Продукції задають *умовні операції* (якщо справедлива умова *A*, виконати дію *B*).

Приклад 3. Типовими продукціями є правила підстановки (замінити *A* на *B*). Прикладом можуть послугувати правила підстановки формальних граматик.

База знань, організована як сукупність продукцій, разом з механізмом керування продукціями, утворює продукційну систему.

Продукційною моделлю називається представлення знань у вигляді *сукупності продукцій. Продукція* визначається як вираз вигляду*:*

$$(i)\ Q;\ P;\ A \Rightarrow B;\ N,$$

де (*i*) — ім’я продукції, за допомогою якого вона виділяється серед усієї множини продукцій;

Q — сфера застосування продукції; предметна область, до якої вона відноситься;

P — умова застосування продукції;

$A \Rightarrow B$ *— ядро продукції;* інтерпретація залежить від конкретної ситуації; часто ядро інтерпретується як *"якщо маєш A, зроби B";*

N — післяумова продукції, постулює процедури, які необхідно виконати після реалізації ядра.

Як приклад можна навести такі продукції:

(М-1) Магазин; у покупця є гроші; якщо покупець хоче купити річ, він платить в касу; змінити базу даних про товари.

Як приклад можна навести таку продукцію:

(БС-29) Банківська сфера; клієнт має рахунок у банку; якщо клієнт бажає зняти певну суму грошей, то він повинен написати ордер; внести зміни до бази даних.

Можливим є поєднання продукційних моделей із фреймовими і мережевими. Наприклад, самі знання можна описувати на основі семантичних мереж, а операції над ними задавати як продукційні, що зумовлюють заміну одного фрагмента мережі на інший. Як типові продукції можна розглядати зв'язки функціональної мережі, які задають алгоритми обчислення одних величин через інші.

Популярність продукційних моделей зумовлена такими рисами [9]:

1. Більшість людських знань може бути записана у вигляді продукцій.
2. Модульність; продукції, за рідким винятком, є незалежними, і внесення або вилучення окремих продукцій, як правило, не приводить до змін в інших продукціях.
3. У разі необхідності продукційні системи можуть реалізувати будь-які алгоритми.
4. Продукції можуть бути порівняно легко розподілені за сферами застосування.
5. Продукції можуть працювати паралельно та асинхронно, тому їх зручно реалізовувати на основі багатопроцесорних комплексів.

До *основних проблем*, пов'язаних з використанням продукційних систем, відносять складність перевірки несуперечливості та коректності функціонування.

6.2. Продукції та мережі виведення

Системи продукцій часто буває зручно зображати у вигляді графів, які називаються *мережами виведення*.

Мережі виведення будуються так. Вершини графу відповідають умовам і діям, що входять до лівих і правих частин продукцій. Якщо в продукційні системі є продукція $A \Rightarrow B$, то від вершини A до вершини B йде орієнтована дуга. Якщо ж є продукція $A, B \Rightarrow C$, то дуги (AC) та (BC) пов'язані відношенням "I" (кон'юнкції)

Так, для продукційної системи

$A \Rightarrow C$

$B \Rightarrow C$

$B, F \Rightarrow L$

$F \Rightarrow Q$

$D, C \Rightarrow G$

мережа виведення має такий вигляд:

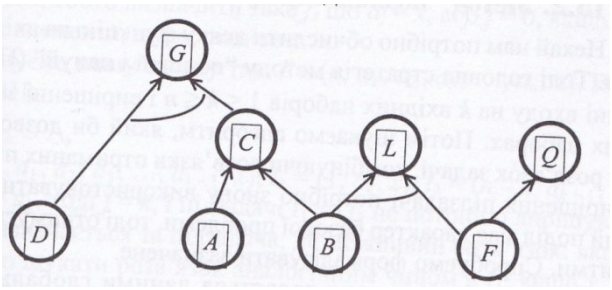


Рис. 6.1. Продукційна мережа виведення

На основі аналізу мереж виведення стає наочнішим те, що відповідь на запит користувача можна отримати шляхом зведення задач до підзадач. Так, щоб вирішити задачу G, необхідно вирішити і задачу D, і задачу C, розв'язування якої в свою чергу, потребує вирішення або A, або B.

6.3. Типова схема роботи експертної системи на базі продукцій

На рис. 6.2. показана узагальнена схема типової експертної системи, побудованої на основі продукційних правил:

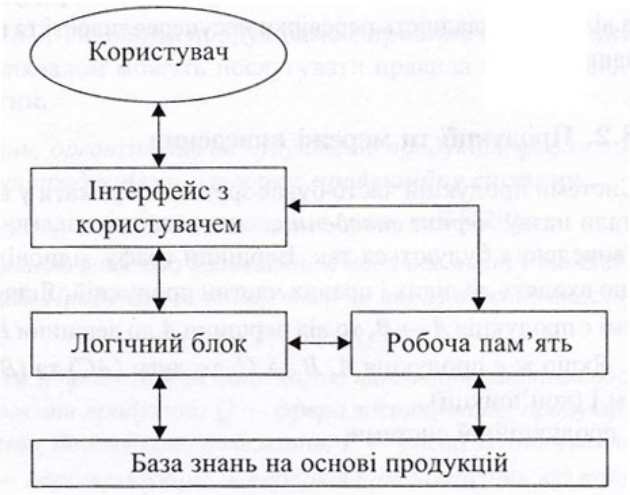


Рис. 6.2. Типова схема роботи продукційної системи

До робочої пам'яті заносяться факти, що задаються користувачем, а також його запити. Логічний блок зіставляє цю інформацію з правилами, які зберігаються в базі знань, і застосовує ті правила, які можуть бути зіставлені із вмістом робочої пам'яті. Наприклад, якщо в базі знань є продукції $A \Rightarrow B$ та $B \Rightarrow C$, то вони можуть бути виконані, коли користувач ввів дані про істинність факту A.

6.4. Пряме та зворотне виведення

Існують дві основні стратегії логічного виведення в продукційних системах: *пряме* та *зворотне виведення*.

Пряме виведення на основі наявних правил передбачає аналіз:

1) наслідків з фактів,

2) наслідків з наслідків і т. д.

Процес продовжується, поки не буде встановлено істинність або хибність запиту користувача.

Наприклад, нехай є продукції $A \Rightarrow B$ та $B \Rightarrow C$; користувач повідомив про істинність факту A і спитав про істинність C. Ланцюжок прямого виведення матиме вигляд: з A виводиться B, з B виводиться C, тому C — істинне.

Пряме виведення *не вважається ефективним* через те, що можна отримати чимало безперспективних продукцій, і, відповідно, породжується дуже багато зайвих проміжних результатів. Експертні системи на основі прямого виведення використовуються, як правило, при плануванні і прогнозуванні.

При зворотному виведенні логічний блок починає роботу із запиту користувача, тобто з твердження, яке перевіряється. Розглядається одне з правил, на основі яких можна вивести це твердження, після чого перевіряється істинність лівої частини цього правила. Процес повторюється до тих пір, доки він не дійде до фактів, які вважаються істинними.

У нашому прикладі ланцюжок зворотного виведення матиме вигляд: "C виводиться з B, B виводиться з A, A істинне, отже, і C істинне".

Як і у випадку прямого виведення, зворотне виведення може вимагати повторень у разі невдалої спроби, і тому є стратегією перебору.

Зворотне виведення тісно пов'язано з механізмом виконання програм Прологу.

6.5. Типові дисципліни виконання продукцій

Передусім ядра продукцій прийнято поділяти на *детерміновані* та *недетерміновані*. У детермінованих продукціях при виконанні лівої частини завжди виконується і права. У недетермінованих продукціях права частина при виконанні лівої виконується необов'язково. Можливі, наприклад такі інтерпретації недетермінованих продукцій: "Якщо A, то можливе B"; або "Якщо A, то B з певною вірогідністю".

Існують різні режими керування виконанням готових продукцій, тобто продукцій, ліві частини яких істинні, і, відповідно, ці продукції можуть бути виконані негайно. Можна виокремити такі режими:

- режим негайного виконання;
- режим формування конфліктного набору.

У *режимі негайного виконання*, перша знайдена продукція *виконується невідкладно*.

У *режимі формування конфліктного набору* знайдені готові продукції не виконуються відразу, а включаються до *конфліктного набору* — списку продукцій, готових до виконання. Лише після завершення формування конфліктного набору відбувається вирішення конфлікту, тобто вибір зі списку готових продукцій якоїсь однієї. Інша назва конфліктного набору "фронт готових продукцій".

Розглянемо *деякі проблеми, пов'язані з конфліктними наборами*. Нехай маємо продукції:

1) $A \Rightarrow B$

2) $A \Rightarrow C$

і A істинне.

Тоді обидві продукції готові до виконання і утворюють конфліктний набір.

Якщо ядра продукцій інтерпретуються як імплікації, проблема є менш гострою. Продукційна система є чисто декларативною. Після застосування 1), тобто виведення B, A залишається істинним, і ми завжди можемо використати продукцію 2). Таким чином, порядок застосування продукційних правил може впливати лише на час роботи системи, але не на істинність висновків.

Зовсім інша ситуація виникає, якщо ядра *продукцій інтерпретуються як "виконати таку-то дію"*. Тоді після виконання 1) ситуація може змінитися: A може перестати бути істинним, і тоді продукція 2) може стати недосяжною. Тому, якщо насправді треба було вибирати C, вибір B стає помилкою, яку не завжди можна виправити.

Існують різні стратегії вирішення конфліктів, а також обмеження при включенні готових продукцій до конфліктного набору.

Розрізняють також *централізоване і децентралізоване керування продукціями*.

За *централізованого керування* продукційна система має центр керування, який вирішує, коли має спрацювати та чи інша продукція.

За *децентралізованого керування* кожна продукція може самостійно прийняти рішення про свій запуск. Один з типових механізмів, який застосовується при цьому, є механізм "класної дошки". Таку назву має спеціальна область пам'яті, доступна для різних продукцій. На ній продукції що працюють паралельно, знаходять інформацію, що

може ініціювати запуск. Туди ж вони пишуть інформацію, що може виявитись корисною для інших продукцій.

6.6. Основні стратегії вирішення конфліктів у продукційних системах

Очевидно, що найзагальнішим способом керування системами продукцій є комплексне планування її роботи, тобто прийняті рішень на основі аналізу всіх можливих дій та їх наслідків з погляду мети, що стоїть перед системою. Але зрозуміло, що вказаний принцип рідко може бути застосований у повному обсязі, насамперед через експоненційне зростання кількості можливих варіантів розвитку подій.

Існують стратегії керування вирішенням конфліктів, мета яких — намагатися уникнути експоненційного вибуху. Подібні стратегії здебільшого можуть бути охарактеризовані як евристичні. В [9] описані деякі стратегії керування виконанням продукцій.

Принцип "стосу книг". Ґрунтується на ідеї, що найкориснішою є продукція, яка використовується найчастіше (так само, як у стосі книг ті, якими найчастіше користуються, як правило, опиняються зверху). Відповідно, з фронту готових продукцій вибирається продукція з найбільшою частотою використання.

Принцип "найдовшої умови". З фронту готових продукцій вибирається та, для якої стала справедливою найдовша умова виконання. Підставою для цього принципу є міркування про те, що часткові правила, застосовні до вузького класу ситуацій, є важливішими, ніж загальні правила для широкого класу ситуацій. Наприклад, нехай є дві продукції:

(1) Якщо *A* — птах, то *A* літає.

(2) Якщо *A* — птах і *A* — пінгвін, то *A* не літає.

Якщо відомо, що *A* — пінгвін, то за принципом найдовшої умови слід вибрати (2).

У [9] відзначається, що принцип найдовшої умови доцільно застосовувати, якщо продукції прив’язані до типових ситуацій, зв’язаних відношенням "часткове — загальне". До цього можна додати, що цей принцип заслуговує на особливу увагу при аналізі винятків. Так, у нашому прикладі правило (2) є винятком з правила (1); якби ми були впевнені, що (1) не має винятків, ми могли б застосувати це загальне правило у будь-якій ситуації.

Принцип метапродукцій. Ґрунтується на введенні до системи знань *метапродукцій*, тобто правил використання продукцій. Типовими метапродукціями можуть бути правила, які визначають, що слід робити, якщо до фронту готових продукцій ввійшли або не ввійшли ті чи інші конкретні продукції.

Принцип "класної дошки". Конфлікти вирішуються на основі обміну інформацією з використанням "класної дошки". Застосування "класної дошки" часто комбінується з застосуванням метапродукцій.

Принцип вибору за пріоритетом. Кожній продукції надається її пріоритет, який відображає її важливість. Ці пріоритети можуть бути різними для різних ситуацій, а також статичними і динамічними. Динамічні пріоритети змінюються з часом і можуть відображати, наприклад, ефективність використання продукцій у минулому або час перебування продукції у фронті готових продукцій.

Принцип керування іменами. Базується на заданні для імен продукцій формальної граматики або іншої процедури, що забезпечує звуження фронту готових продукцій і вибір з нього чергової продукції для виконання.

7. КОНЕКЦІОНІСТСЬКІ МОДЕЛІ ТА МЕТОДИ

7.1. Загальна характеристика конекціоністського підходу та його місце в теорії інтелектуальних систем

Конекціоністський підхід до побудови систем штучного інтелекту розвинувся на противагу символьному, що є характерним для сучасних моделей знань. В основі конекціоністського підходу лежить спроба безпосереднього моделювання розумової діяльності людського мозку. Відомо, що мозок людини складається з величезної кількості нервових клітин (*нейронів*), що взаємодіють між собою. Ці "обчислювальні елементи" мозку функціонують набагато повільніше, ніж обчислювальні елементи комп'ютерних систем. Але ефективність людського інтелекту досягається як за рахунок паралельної роботи нейронів, так і за рахунок того, що механізми їх взаємодії були вироблені шляхом тривалої еволюції.

Для багатьох цілей нейрон можна розглядати як елемент з певним критичним значенням. Це означає, що він або ж активізується (збуджується), якщо сума його входів досягає певного значення, або ж залишається пасивним.

МакКаллок і Піттс довели, що будь-яку обчислювану функцію можна реалізувати за допомогою спеціально організованої мережі з нейронів, модель яких вони запропонували. Але на практиці використання нейромереж пов'язане з кількома проблемами. По-перше, проблема полягає в тому, чи можна знайти якийсь розумний принцип реорганізації мережі, який дозволяв би випадково об'єднаній групі ідеальних нейронів самоорганізовуватися в "обчислювальний пристрій", здатний вирішувати задачу розпізнавання. По-друге, потрібно використовувати велику кількість нейронів. Так, модель мурашки потребує близько 20000 нейронів, людини — 100 млрд. нейронів, що на практиці поки що неможливо.

Нейрологічна теорія стала також основою системи розпізнавання, яка дістала назву *перцептрон*. Перцептрон Розенблатта передавав повідомлення від "ока", яке реалізовувалось системою фотоелементів, в блоки електромеханічних комірок пам'яті, які оцінювали відносні величини електричних сигналів. Перцептрон міг навчатись шляхом спроб і помилок, а також корекцією електричних імпульсів.

Мінські і Пейпертом було доведено, що перцептрони не в змозі розв'язувати низку типових задач, наприклад, розпізнавання частково затулених предметів.

Один з сучасних напрямків створення розумних машин — це розробка *нейрокомп'ютерів*. *Нейрокомп'ютер* — це спеціалізована ЕОМ, яка реалізує деяку формальну модель природної мережі нейронів.

Переваги нейрокомп'ютерів:

1) Характер взаємозв'язків між нейронами дозволяє розв'язувати багато задач на основі паралельної обробки;
2) У нейронній мережі пам'ять не локалізована в одному місці (як в послідовних машинах), а розподілена по всій структурі. В біологічних системах пам'ять реалізується підсиленням або послабленням зв'язків між нейронами, а не зберіганням двійкових символів;
3) Біологічні мережі реагують не на всі, а тільки на визначені зовнішні подразнення. Кожний нейрон виступає як елемент прийняття рішення і як елемент зберігання інформації. Перевага такої структури — "життєздатність" (вихід з ладу декількох нейронів не приводить до значної зміни даних, що зберігаються, або ж до руйнування всієї системи).

4) Можливість адресації за вмістом (асоціативної пам'яті) є ще однією важливою характеристикою систем з розподіленою пам'яттю (кожний елемент відшукується за його вмістом, а не зберігається в комірці пам'яті з визначеним номером).

В основу зв'язків в нейрокомп'ютерах покладено *принцип асоціацій*. *Асоціативні зв'язки* пронизують все мислення людини. Навіть найбільш примітивні процеси навчання принципово залежать від послідовності подій в часі. Це й закладено в природу нейронних систем. Тому їм притаманне реагування тільки на жорстко визначені зовнішні подразнення. Наприклад, домашні тварини "навчаються" ігнорувати повторні несуттєві зовнішні подразнення ("цокання" годинника), але посилюють сприйняття подразнень, які можуть мати серйозні наслідки (звук автомобільних гальм).

Штучні нейрони моделюють структуру й функції біологічних нейронів. Архітектура й особливості штучних нейронних мереж, утворених нейронами, залежать від конкретних завдань, які мають бути вирішені з їхньою допомогою [11].

7.2. Модель штучного нейрона

Структуру штучного нейрона, запропоновану у [11], наведено на рис. 7.1.

Вхідні сигнали штучного нейрона x_i $\left(i = \overline{1,N}\right)$ надходять ззовні або є вихідними сигналами інших нейронів. Кожний з них має свою вагу w_i $(i = \overline{1,N})$.

Рис. 7.1. Структура штучного нейрона

Вхідний оператор $f_{\text{вх}}$ перетворює зважені входи й подає їх на оператор активації f_a. Вихідний сигнал нейрона *y* являє собою перетворений вихідним оператором $f_{\text{вих}}$ вихідний сигнал оператора активації. Таким чином, нелінійний оператор перетворення вектора вхідних сигналів **x** у вихідний сигнал *y* може бути записаний у такий спосіб:

$$y = f_{\text{вих}}(f_{\text{а}}\,(f_{\text{вх}}(\mathbf{x}, \mathbf{w}))) \tag{7.1}$$

Як вже зазначено, вихідний сигнал нейрона може бути вхідним для наступного.

Вхідний оператор (вхідна функція) нейрона задає вигляд перетворення зважених входів. Відмінність гальмуючих входів від збуджувальних відбивається у знаках відповідних ваг. Звичайно використовуються такі вхідні функції:

— сума зважених входів [12]:

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{n} w_i x_i;$$

— максимальне значення зважених входів:

$$f(\mathbf{x}, \mathbf{w}) = \max_i(w_i x_i);$$

— мінімальне значення зважених входів:

$$f(\mathbf{x}, \mathbf{w}) = \min_i (w_i x_i).$$

7.2.1. Функція активації

Функція активації $f_a(\cdot)$ описує правило переходу нейрона, що перебуває в момент часу k у стані $z(k)$, у новий стан $z(k+1)$ при надходженні вхідного сигналу **x** [11]:

$$z(k+1) = f_a(z(k), f_{\text{вх}}(\mathbf{x}, \mathbf{w})).$$

Надалі позначатимемо функцію активації без індексу *«a»*. Найбільш простими активаційними функціями є:

— лінійна [13–15]:

$$f(z) = Cz,\ C = \text{const};$$

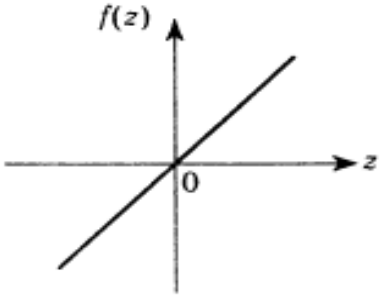


Рис. 7.2. Лінійна функція

Незважаючи на те, що лінійні функції є найбільш простими, мережі, які використовують лише їх, мають обмежені можливості. Двошарова лінійна мережа еквівалентна одношаровій з ваговою матрицею, що дорівнює добутку вагових матриць першого й другого шарів. Отже, будь-яка багатошарова лінійна мережа може бути замінена еквівалентною одношаровою. Тому для розширення можливостей мереж застосовують нелінійні функції активації.

— лінійна біполярна з насиченням [11]:

$$f(z) = \begin{cases} 1 \text{ при } z > \alpha_2; \\ Cz \text{ при } -\alpha_1 \le z \le \alpha_2; \\ -1 \text{ при } z < -\alpha_1; \end{cases}$$

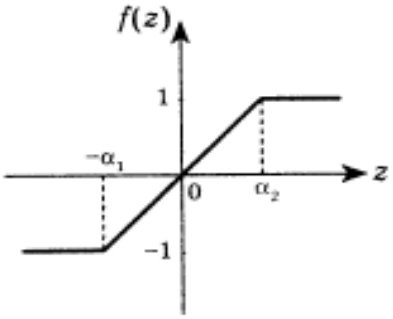


Рис. 7.3. Лінійна біполярна функція з насиченням

— ReLU (*rectified linear unit*) або випрямляч [16]:

$$f(z) = \max\{0, z\}$$

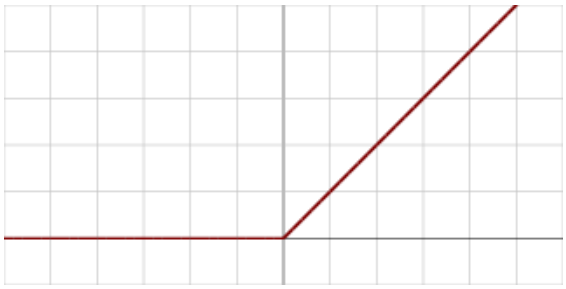


Рис. 7.4. Функція ReLU

У роботі У. МакКаллока і У. Піттса для активації нейронів використовувалася *функція Хевісайда — бінарна порогова* гранична функція вигляду [12]

$$f(z) = \begin{cases} 1, \text{при } z \geq 0; \\ 0, \text{при } z < 0. \end{cases}$$

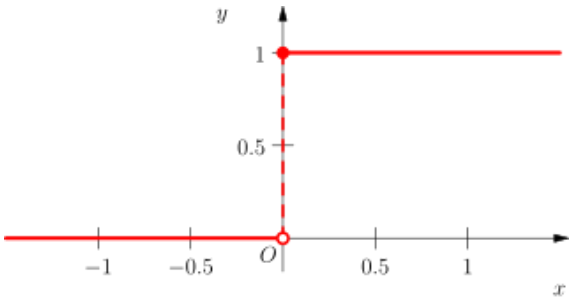


Рис. 7.5. Уніполярна порогова функція

Різновидом даної функції є *біполярна порогова* функція

$$f(z) = \begin{cases} 1, \text{при } z \geq \alpha; \\ -1, \text{при } z < \alpha. \end{cases}$$

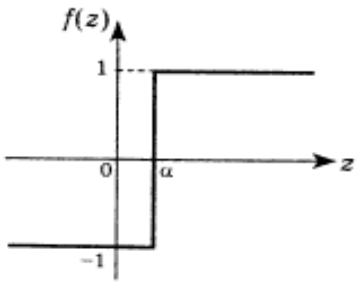


Рис. 7.6. Біполярна порогова функція

Ці функції активації застосовувалися в основному в класичних ШНМ. При побудові нових структур ШНМ найчастіше доводиться працювати як із самою активаційною функцією, так і з її першою похідною. У цих випадках необхідним є використання диференційованих функції. Це так звані *логістичні*, або *сигмоподібні* (*S*-подібні), функції.

Функція називається сигмоподібною, якщо вона є монотонно зростаючою, диференційованою і задовольняє умови:

$$\lim_{\lambda\to-\infty} f(\lambda) = k_1, \qquad \lim_{\lambda\to+\infty} f(\lambda) = k_2, \qquad k_1 < k_2.$$

До таких функцій належать [11, 12]:

— логістична (уніполярна)

$$f_{\log}(z) = \frac{1}{1+e^{-kz}}; \tag{7.2}$$

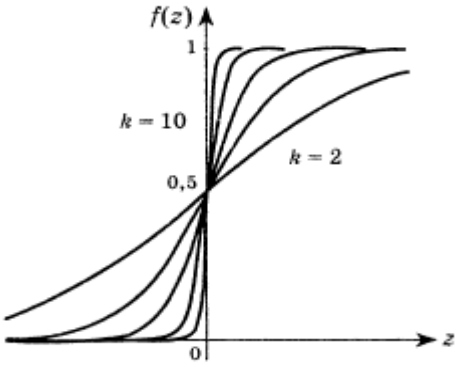


Рис. 7.7. Логістична функція

— гіперболічний тангенс (біполярна)

$$f_{\text{th}}(z) = \tanh(\alpha z) = \frac{e^{\alpha z} - e^{-\alpha z}}{e^{\alpha z} + e^{-\alpha z}};$$

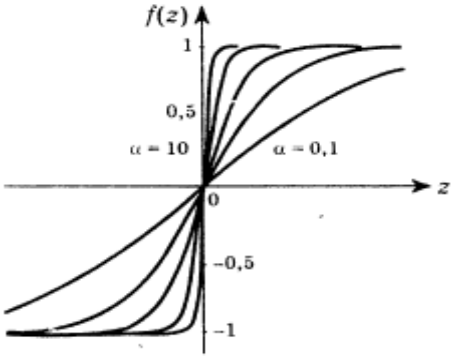


Рис. 7.8. Гіперболічний тангенс

У випадку застосування ШНМ для розв’язування задач розпізнавання у останньому шарі часто використовують функцію softmax [17]. При цьому кількість нейронів цього шару покладають рівною числу K — кількості класів для задачі, яку розв’язують з використанням нейромережі. Припустимо, що $s_k(\mathbf{x})$ — деяка оцінка належності вхідного вектора **x** до k-го класу, яка ще називається *логітом* (*logit*) [18]. Тоді можна оцінити ймовірність p_k належності до відповідного класу наступним чином:

$$p_k = \text{softmax}\left(s(\mathbf{x})_k\right) = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{i=1}^{K}\exp\left(s_i(\mathbf{x})\right)},\ k = 1,\ldots,K\,.$$

За рахунок нормалізації величини p_k задовольняють умову $p_k > 0,\ p_1 + \ldots + p_K = 1$.

Тоді нейромережевий класифікатор відносить **x** до класу, для якого ймовірність p_k максимальною:

$$y = \arg\max_k\left(\text{softmax}\left(s_k(\mathbf{x})\right)\right) = \arg\max_k\left(s_k(\mathbf{x})\right).$$

Моделі штучних нейронів залежать від конкретних застосувань. Тому синтез моделі в кожному окремому випадку є нетривіальною задачею.

7.2.2. Формальна модель нейрона МакКаллока-Піттса

Формальний штучний нейрон (його називають також нейроном МакКаллока-Піттса) [11] може бути поданий як нелінійний перетворювач із ваговими коефіцієнтами w_i, які також називаються синаптичними вагами. Нейрон додає N зважених входів і здійснює нелінійне перетворення (див. рис. 7.1):

$$y = \begin{cases} 0, & \sum_{i=1}^{N} w_i x_i < T, \\ 1, & \sum_{i=1}^{N} w_i x_i \geq T. \end{cases}$$

або

$$y = f\left(\sum_{i=1}^{N} w_i x_i + \theta\right), \tag{7.3}$$

де y — вихідний сигнал нейрона; f — порогова функція активації; N — кількість входів; w_i — синаптичні ваги; x_i — вхідні сигнали ($i = \overline{1,N}$); $T \in \mathbb{R}$ — *поріг*, $\theta = -T$ — пороговий сигнал, який називається *зсувом* (bias) [18].

Позначаючи $\theta = w_0 x_0$ (зазвичай $x_0 = 1$) формулу (7.3) можна переписати у вигляді

$$y = f\left(\sum_{i=0}^{N} w_i x_i\right) = f(\mathbf{w}^T\mathbf{x}),$$

де $\mathbf{x} = (1, x_1, \ldots, x_N)^T$; $\mathbf{w} = (w_0, w_1, \ldots, x_N)^T$ — відповідні вектори входів і ваг розмірності $(N+1) \times 1$.

7.3. Архітектура штучних нейронних мереж

7.3.1. Поняття штучної нейромережі

З'єднані між собою нейрони утворюють *штучну нейронну мережу* (ШНМ) [12]. Таким чином, ШНМ — це пара (V, C), де V — множина нейронів; C — множина зв'язків [11]. Структура мережі задається у вигляді графа, у якому вершини є нейронами, а ребра являють собою зв'язки (з'єднання).

У переважній більшості архітектур групи нейронів об'єднані у шари. У загаль-

ному випадку *багатошарова* ШНМ складається з декількох шарів, серед яких обов'язково є *вхідний*, що отримує зовнішні сигнали, вихідний, що відбиває реакцію нейронів на комбінації вхідних сигналів, і в багатошарових ШНМ — приховані шари (рис. 7.10). Така пошарова організація є аналогом шаруватих структур мозку. Наведена на рис. 7.10 ШНМ має *N* входів та *M* виходів.

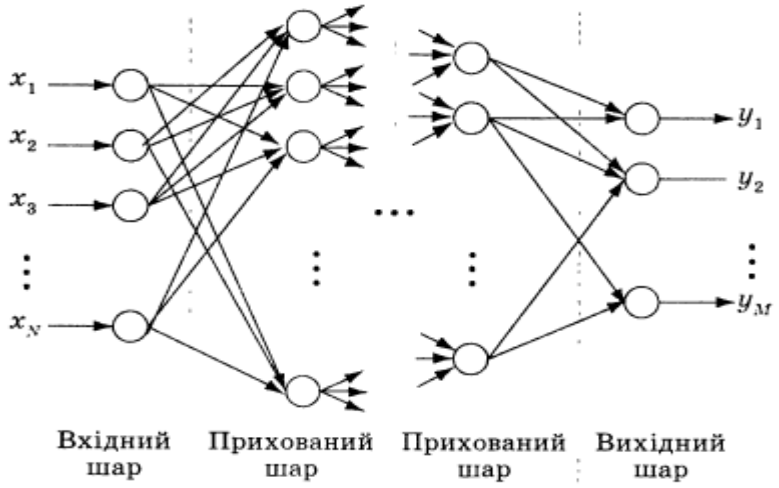


Рис. 7.10. Структура ШНМ

Зв'язки між нейронами задаються у вигляді векторів і матриць. Ваги зручно подавати елементами матриці $W = [w_{ij}]$ розмірності $n \times n$, де n — кількість нейронів мережі. Елемент w_{ij} відбиває зв'язок між i-м й j-м нейронами. При цьому, якщо

$w_{ij} = 0$ — зв'язок між i-м й j-м нейронами відсутній;

$w_{ij} < 0$ — гальмуючий зв'язок;

$w_{ij} > 0$ — прискорювальний (збуджуючий) зв'язок.

Залежно від того, чи містять ШНМ зворотні зв'язки, чи ні, розрізнюють такі їхні топології [11]:

- ШНМ без зворотних зв'язків (прямого поширення, feedforward)
- ШНМ зі зворотними зв'язками (зворотного поширення, рекурентні, feedback)
  — з прямими зворотними зв'язками (direct feedback);
  — з непрямими зворотними зв'язками (indirect feedback);
  — з латеральними зв'язками (lateral feedback);
  — повнозв'язні.

7.3.2. ШНМ прямого поширення

ШНМ прямого поширення припускає наявність декількох шарів зі зв'язками між нейронами різних шарів. У *мережах першого порядку* існують тільки зв'язки між двома сусідніми шарами, тобто між i-м й $(i+1)$-м шарами. Ці мережі мають назву *багатошарового перцептрона*. Приклад такої мережі зображено на рис. 7.10. Якщо в мережі цього типу кожен нейрон i-го шару пов'язаний з кожним нейроном $(i+1)$-го шару, мережа називається повнозв'язною мережею прямого поширення.

У *мережах другого порядку* поряд із зв'язками між нейронами сусідніх i-го й $(i+1)$-го шарів присутні зв'язки між нейронами шарів i-го й $(i+l)$-го, де $l > 1$. Такий зв'язок називається "shortcut". Приклад такої ШНМ наведено на рис. 7.11.

Для мереж прямого поширення матриця W є верхньою трикутною матрицею. Часто для мереж прямого поширення 1-го порядку використовують окремі матриці ваг лише для сусідніх шарів. Кількість цих матриць рівна кількості шарів ШНМ (без урахування вхідного шару). Такий підхід дає змогу вагових матриць з великою кількістю нулів, що притаманно єдиній матриці ваг для усієї багатошарової мережі.

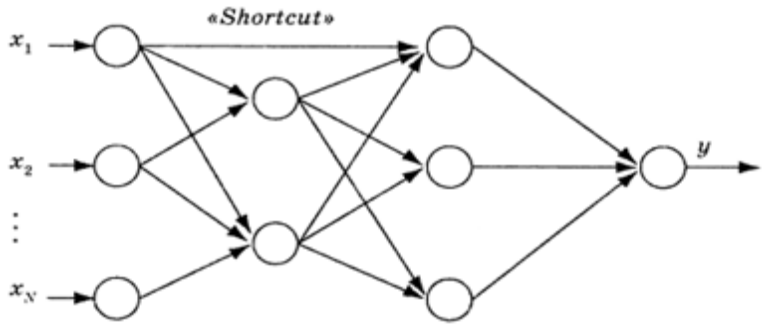


Рис. 7.11. ШНМ прямого поширення другого порядку

7.3.3. ШНМ зворотного поширення

Мережі цього типу допускають наявність *зворотних зв'язків* як між нейронами різних шарів, так і між нейронами одного шару. Використання мереж зі зворотними зв'язками необхідне у процесі вивчення складних динамічних об'єктів, наприклад об'єктів, що змінюють свій стан при надходженні нових вхідних сигналів. Такі ШНМ можуть мати властивості, подібні до короткочасної людської пам'яті.

У ШНМ із *прямими зворотними зв'язками* (рис. 7.12) на вхід нейрона деякого i-го шару подається його вихідний сигнал, тобто даний нейрон підсилює або послаблює сигнал, перетворений його активаційною функцією, завдяки чому досягається його граничний активаційний стан.

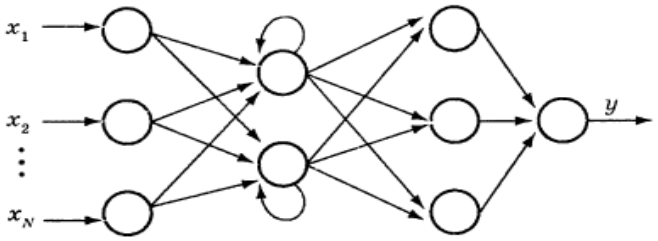


Рис. 7.12. ШНМ із прямими зворотними зв'язками

У ШНМ із *непрямими зворотними зв'язками* існують зв'язки нейрона i-го шару з нейронами $(i–k)$-го шару $k > 0$. При цьому одночасно можуть бути прямі зв'язки цього ж нейрона з нейроном $(i+l)$-го шару ($l > 0$). Введення таких зворотних зв'язків необхідно, щоб виділити певну особливо важливу для даної ШНМ область вхідних сигналів (рис. 7.13).

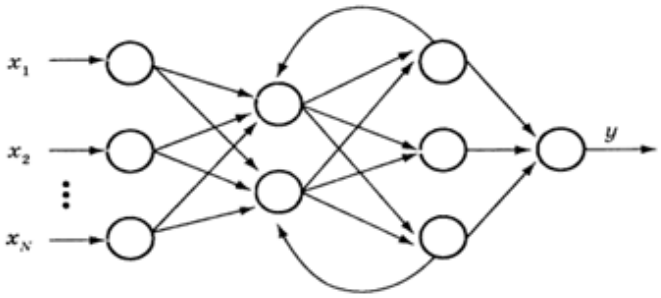


Рис. 7.13. ШНМ із непрямими зворотними зв’язками

ШНМ із *латеральними зв’язками* має зв’язки між нейронами одного шару (рис. 7.14). Такий тип зворотних зв’язків використовується у тому випадку, якщо тільки один нейрон з даної групи нейронів має бути активним. У цьому випадку на вхід кожного нейрона надходять гальмуючий (послаблюючий) сигнал від інших нейронів і звичайно збуджувальний (посилюючий) сигнал власного зворотного зв’язку. Нейрон із найбільшою активністю (переможець) придушує інші нейрони. Тому цю топологію називають також топологією мережі «переможець отримує все» (WTA–Network).

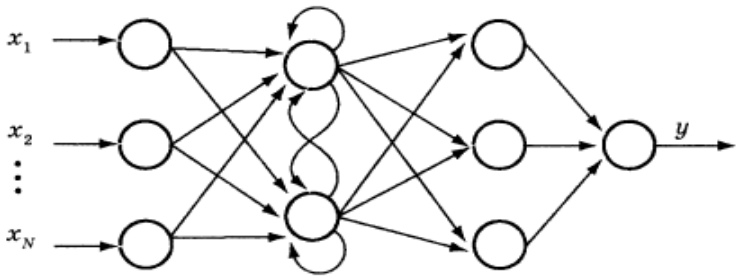


Рис. 7.14. ШНМ із латеральними зв’язками

7.3.4. Повнозв’язні ШНМ

Повнозв’язні ШНМ характеризуються наявністю зв’язків між усіма нейронами мережі (рис. 7.15). Цей вид топології відомий також як мережа Хопфілда.

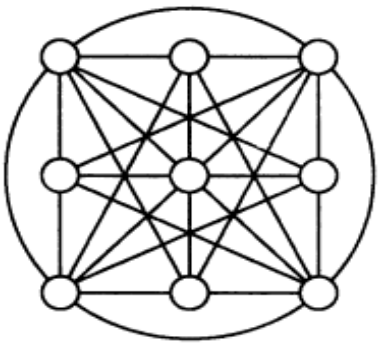


Рис. 7.15. Повнозв’язна ШНМ

Особливістю даної топології є те, що матриця зв’язків W має бути симетричною з нульовими діагональними елементами.

7.3.5. Глибинні ШНМ

Глибинна нейромережа може містити велику кількість шарів. Одним з найбільш ефективних видів глибинних ШНМ є згорткові нейромережі. На рис. 7.16 наведено архітектуру згорткової нейромережі [16]. На початку почергово йдуть згорткові та агрегатні шари. Останні шари мають структуру багатошарової ШНМ прямого поширення 1-го порядку (багатошарового перцептрона) [17].

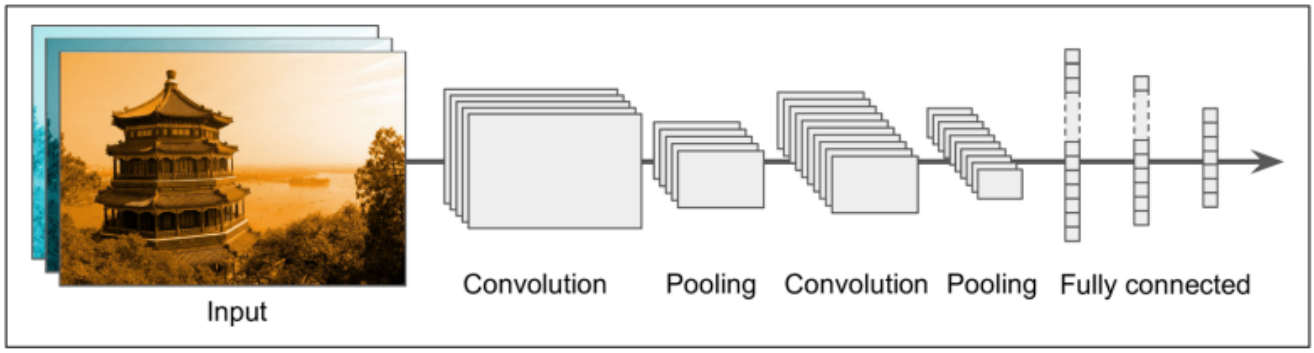


Рис. 7.16. Згорткова глибинна мережа

Приклад успішної згорткової архітектури наведено на рис. 7.17.

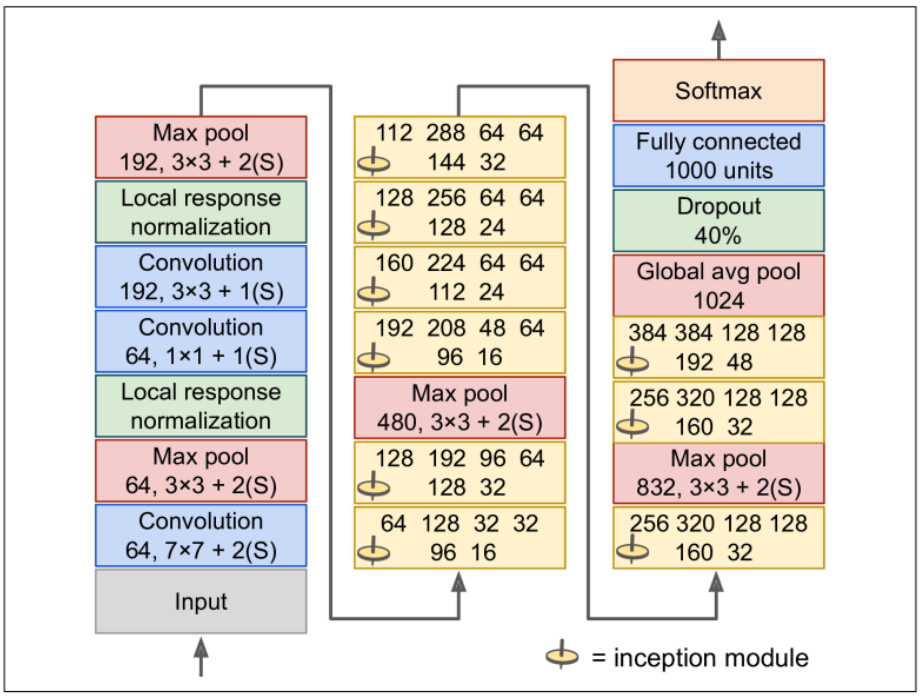


Рис. 7.17. Архітектура GoogLeNet

7.4. Навчання ШНМ

7.4.1. Поняття про навчання ШНМ

Характерною властивістю ШНМ є її здатність до навчання, що полягає *у виробленні правильної реакції на подані їй різні вхідні сигнали*. Існують такі можливості навчання ШНМ:

- зміна конфігурації мережі шляхом утворення нових або виключення деяких існуючих зв’язків між нейронами;

- зміна характеристик нейронів (виду й параметрів активаційної функції й т. д.);
- зміна елементів матриці ваг;

Найбільшого поширення сьогодні отримав підхід, при якому структура мережі задається апріорно, а мережа навчається шляхом налаштування вагових коефіцієнтів. У цьому випадку навчання полягає у зміні за певною процедурою елементів матриці *W* при послідовному поданні мережі деяких векторів, що навчають.

У зв'язку з цим штучний нейрон може бути зображений у такий спосіб (рис. 7.18).

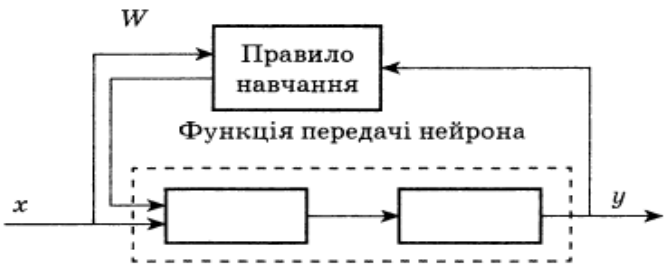


Рис. 7.18. Модель навчання нейрона

У процесі навчання ваги стають такими, що під час надходження вхідних сигналів мережа виробляє відповідні необхідні вихідні сигнали. Розрізняють *навчання з учителем* і *без учителя*. Перший тип навчання припускає, що є «учитель», що задає навчальні пари — для кожного вхідного вектора наявний необхідний вихід мережі. Для кожного вхідного вектора обчислюється вихід мережі, порівнюється з бажаним значенням, визначається помилка виходу, на основі якої й коректуються ваги. Навчальні пари подають мережі послідовно й ваги корегують доти, поки похибка мережі не досягне необхідного рівня.

Цей вид навчання неправдоподібний з біологічної точки зору [11]. Дійсно, важко уявити зовнішнього "учителя" мозку, що порівнює реальні й необхідні реакції того, кого навчають, і коригує його поведінку за допомогою негативного зворотного зв'язку. Природнішим є *навчання без учителя*, коли мережі подаються тільки вектори вхідних сигналів, і мережа сама, використовуючи деякий алгоритм навчання, підстроювала б ваги так, щоб при поданні їй досить близьких вхідних векторів вихідні сигнали були б однаковими. У цьому випадку в процесі навчання виділяються статистичні властивості множини вхідних векторів і відбувається об'єднання близьких (подібних) векторів у класи. Подання мережі вектора з даного класу викликає її певну реакцію, яка до навчання є непередбаченою. Тому в процесі навчання виходи мережі мають трансформуватися в деяку зрозумілу форму. Це не є серйозним обмеженням, оскільки зазвичай нескладно ідентифікувати зв'язок між вхідними векторами й відповідною реакцією мережі.

Існує ще один вид навчання — з підкріплюванням (reinforcement learning), при якому також передбачається наявність учителя, що не підказує, однак, мережі правильної відповіді [17].

Для навчання з учителем є характерним поділ множини навчальних пар на *навчальну* (*train set*) та *тестову* (*test set*) множини (вибірки) [18, 19]. Навчання відбувається на навчальній вибірці, а *оцінка його якості* — на тестовій. Якщо (середня) похибки мережі на тестовій вибірці незначна, то можна вважати, що навчання завершилося успішно [20]. Якщо похибки мережі на тестовій вибірці велика, а на навчальній вибірці —

незначна, то має місце явище, яке в теорії машинного навчання називається перенавчанням (*overfitting*) [16, 22]. Перенавчання свідчить про те, що вибрана структура або архітектура нейромережі не є адекватною до даних, з якими оперує мережа (зазвичай це вказує на те, що модель є занадто складною), і її потрібно змінювати.

Часто частину навчальної вибірки відводять під так звану *множину валідації* [21]. Вона не застосовується для вироблення корекцій ваг а використовується для того, щоб оцінювати різні стани мережі у процесі навчання. Якщо похибка валідації у процесі навчання спочатку спадає, а потім зростає (див. рис. 7.19), то це свідчить про те, що процес навчання варто зупинити [17].

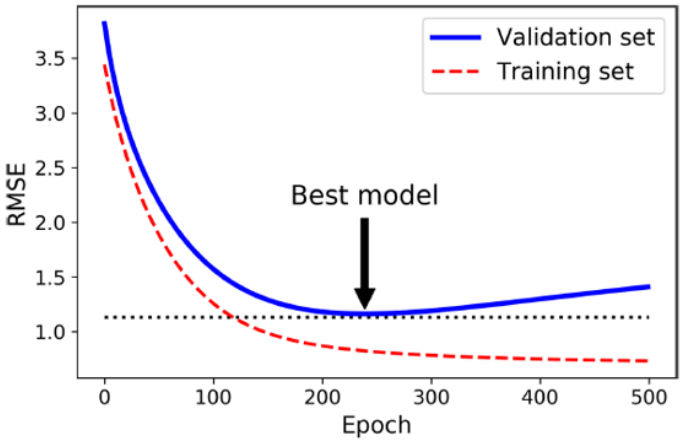


Рис. 7.19. Криві похибки мережі на навчальній вибірці та множині валідації

При навчанні з учителем використовується різноманітні *режими навчання*. Якщо корекція ваг відбувається кожний раз після подання одного навчального вектора, то відповідний режим навчання має назву *онлайн навчання* (*online learning*). У режимі *офлайн* (*offline learning*) на кожному кроці навчання виконується одна корекція для всієї навчальної множини, тобто використовується сума значень функціоналів похибки усіх елементів навчальної вибірки. Використання режимів онлайн та офлайн навчання допускається для значної кількості алгоритмів навчання.

Проміжним типом навчання з учителем є *пакетне навчання* (*batch learning*). Воно характеризується розміром *пакета даних* (*batch*) — кількості навчальних векторів, які використовуються для здійснення однієї корекції ваг мережі. Одна *епоха* навчання полягає у повному проході усієї навчальної вибірки. Протягом однієї епохи здійснюється обробка p / q пакетів даних, де p — розмір навчальної вибірки, q — розмір пакета даних [16, 19, 20].

7.4.2. Правило навчання Гебба (корелятивне, співвідносне навчання)

Більшість сучасних алгоритмів навчання виросло із правила Гебба. Наприкінці 40-х років XX ст. років Д. О. Гебб теоретично встановив, що асоціативна пам'ять у біологічних системах викликається процесами, що змінюють зв'язки між нервовими клітинами. Відповідно до установленого їм правила, що називається «правилом Гебба», при одночасній активації (порушенні) двох нейронів синаптична сила (вага їхнього зв'язку) зростає. Таким чином, часто використовувані зв'язки в мережі підсилюються, що пояснює феномен звички й навчання повторенням.

У ШНМ зростання синаптичної сили еквівалентне збільшенню ваги зв'язку між нейронами i та j на величину

$$\Delta w_{ij} = \gamma x_i y_j,$$

де x_i — вихід i-го та вхід j-го нейронів; y_j — вихід j-го нейрона; γ — коефіцієнт, що впливає на швидкість навчання [11].

Правило Гебба використовується у зв'язках асоціативної пам'яті, а також у деяких інших, заснованих на навчанні без учителя.

7.4.3. Дельта-правило

Це важливе правило навчання було запропоновано Б. Уідроу й М. Е. Гоффом і найбільше відповідає одношаровим ШНМ прямого поширення. Ідея його полягає в тому, що якщо під час навчання мережі можна встановити розбіжність між її бажаною й наявною реакціями, ця розбіжність може бути усунута або зменшена шляхом зміни певним чином вагових коефіцієнтів зв'язку. Для цього й використовується дельта-правило, відповідно до якого зміна ваги зв'язку між i-м й j-м нейронами визначається у такий спосіб:

$$\Delta w_{ij} = \gamma x_i (y_j^* - y_j),$$

де x_i — вихід попереднього i-го нейрона; y_j^*, y_j — бажана й реальна реакції j-го нейрона відповідно; γ — коефіцієнт, що впливає на швидкість навчання [11].

7.4.4. Градієнтні методи навчання

Багато методів навчання засновано на мінімізації деякої цільової функції E, яка має в літературі різні назви: функція похибки (error function), функція втрат (loss), функція ціни (cost). Функція E, зазвичай, є опуклою. Якщо використовувані функції активації $f(\cdot)$ диференційовані, зручно застосовувати градієнтні методи мінімізації. У цьому випадку корекція ваг зв'язку між i-м й j-м нейронами відбувається за правилом

$$\Delta w_{ij} = -\gamma \nabla_{\mathbf{w}} E(\mathbf{w})_{ij},$$

де γ — коефіцієнт, що впливає на швидкість навчання (learning rate);

$$\nabla_{\mathbf{w}} E(\mathbf{w})_{ij} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}_{ij}}.$$

Це так званий *градієнтний спуск*. Процес градієнтного спуску (у випадку одного вагового коефіцієнта) наведено на рис. 7.20

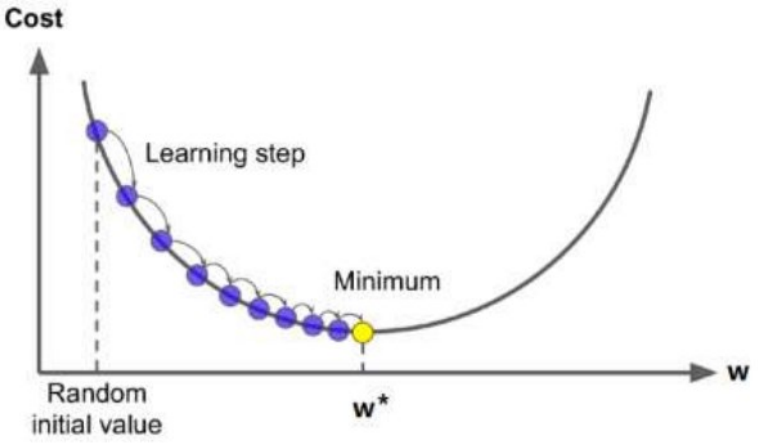


Рис. 7.20. Ілюстрація процесу градієнтного спуску

Градієнтні методи використовуються при контрольованому навчанні, коли відома необхідна реакція нейронів $\mathbf{y}^*$. Більшість градієнтних методів розв'язування задачі регресії засновано на мінімізації квадратичного функціонала похибки

$$E(\mathbf{w}) = 0{,}5e^2(k) = 0{,}5(y^*(k) - y(k))^2,$$

де k — номер кроку алгоритму навчання. Для одного вихідного нейрона з лінійною функцією активації та режиму онлайн:

$$E(\mathbf{w}) = 0{,}5\varepsilon^2(k) = 0{,}5(y^*(k) - \mathbf{w}^T(k)\mathbf{x}(k))^2.$$

У цьому випадку

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = -\varepsilon(k)\mathbf{x}(k),$$

де $\varepsilon(k) = y^*(k) - \mathbf{w}^T(k)\mathbf{x}(k)$.

Таким чином приходимо до алгоритму методу найменших квадратів (МНК):

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \gamma(k)e(k)\mathbf{x}(k).$$

Важливим параметром градієнтного спуску є коефіцієнт γ. Якщо його значення замале, то для збіжності алгоритму потрібно виконати занадто багато операцій (див. рис. 7.21 а). Якщо ж обрано занадто велике значення, то можливі великі «стрибки» значень цільової функції і навіть розбіжність алгоритму навчання (див. рис. 7.21 б).

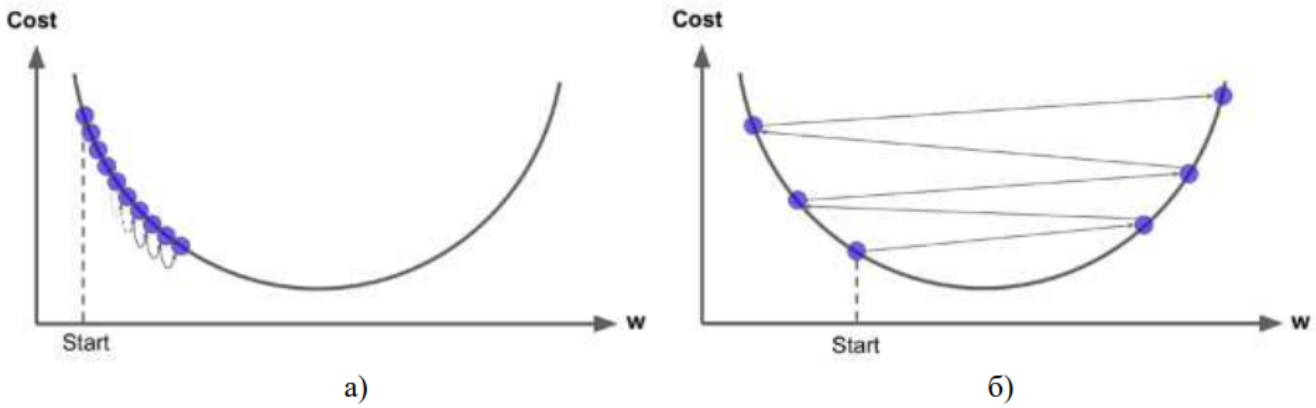


Рис. 7.21. Вплив значення нормуючого множника приросту на збіжність градієнтного спуску

Існують різні рекомендації з вибору γ. Так, у теорії стохастичної апроксимації, цей коефіцієнт вибирається змінним і таким, що задовольняє умовам Дворецького

$$\lim_{k\to\infty} \gamma(k) = 0, \sum_{k=1}^{\infty} \gamma(k) = \infty, \sum_{k=1}^{\infty} \gamma^2(k) < \infty,$$

зміст яких полягає в тому, що для збіжності послідовності у деяку точку $\mathbf{w}^*$ довжина кроку $\gamma(k)$ має з одного боку спадати досить повільно (для забезпечення власне збіжності), а з іншого — досить швидко (з метою придушення завад). Таким умовам задовольняє, наприклад, ряд $\gamma(k) = \gamma_0 k^{-\alpha}$, де γ_0 — деяка додатна константа, $0{,}5 < \alpha \leq 1$.

7.5. Одношаровий перцептрон

7.5.1. Будова перцептрона

Значний інтерес до перцептронів викликаний роботою Ф. Розенблатта, у якій він досліджував нейромережеву модель сітківки (RETINA). Згодом такий підхід широко використовувався для моделювання обробки оптичних сигналів. Відповідно до концепції Розенблатта, перцептрон містить три шари (рис. 7.22), що послідовно здійснюють попередню обробку (розбивання) образу, оцінку його характеристик і розпізнавання:

— сітківка (RETINA);
— асоціативний шар;
— вихідний шар.

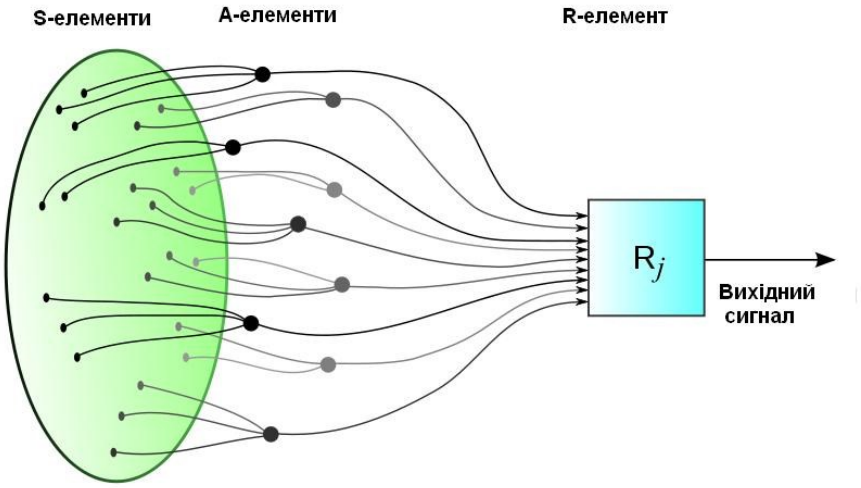


Рис. 7.22. Перцептрон Розенблатта з одним виходом

Попередня обробка образу не залежить від його виду. Однак результат цієї обробки має забезпечити можливість розпізнавання образів на основі аналізу їхніх характеристик. Нарешті, вихідний шар (класифікатор) аналізує характеристики вхідного образу й установлює його відповідність одному з раніше поданих.

Сигнали першого шару, сітківки, подані у двійковій формі, надходять на асоціативний шар, причому в загальному випадку не всі нейрони першого шару пов’язані з усіма нейронами другого шару. При встановленні цих зв’язків виникає можливість структуризації вхідних даних, тобто виділення й об’єднання в так звані рецептивні поля найбільш важливих ознак (областей). Під рецептивним полем розуміють множину всіх нейронів вхідного шару, пов’язаних з одним нейроном асоціативного шару.

Нейрони асоціативного шару мають лінійні активаційні функції, тому під час надходження із сітківки вхідних сигналів вони посилають імпульси на вихідний шар, де й відбувається додавання зважених імпульсів.

Зв’язки між нейронами асоціативного й вихідного шарів варіабельні й можуть модифікуватися шляхом зміни вагових коефіцієнтів. У вихідному шарі використовуються уні- або біполярна функції активації. Якщо сума зважених імпульсів перевищує деяке задане порогове значення, виробляється одиничний вихідний сигнал, якщо не перевищує — нульовий (для уніполярної) або –1 (для біполярної функції активації). У зв’язку з цим перцептрон може розглядатися як двошарова ШНМ прямого поширення [14].

7.5.2. Навчання перцептрона

Існують різні шляхи реалізації процесу навчання перцептрона, однак у їхній основі лежить таке правило: вагові коефіцієнти перцептрона змінюються тільки тоді, коли виникає розбіжність між його фактичною й бажаною реакціями.

Схему навчання перцептрона наведено на рис. 7.23.

Передбачається, що активаційна функція є пороговою. Процес навчання полягає в послідовному поданні навчальних пар $(\mathbf{x}_p, y_p^*)$, $p = \overline{1,P}$, де $\mathbf{x}_p$ — N-вимірний вхідний вектор, y_p^* — бажаний вихідний сигнал p-ї навчальної пари, відповідно, за допомогою яких визначається необхідний вектор вагових коефіцієнтів $\mathbf{w}^*$ такий, що

$$\operatorname{sgn}\left(\mathbf{w}^{*T}\mathbf{x}_p\right) = y_p^*, \qquad p = \overline{1,P}.$$

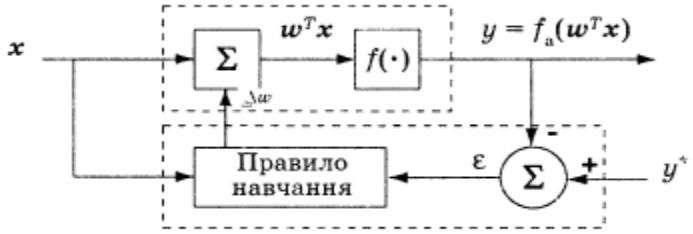


Рис. 7.23. Схема навчання перцептрона

У цьому випадку вектор $\mathbf{w}^*$ забезпечить правильну класифікацію перцептроном усіх навчальних пар.

Гіперплощина $\mathbf{w}^{*T}\mathbf{x} = 0$ ділить вхідний простір на два підпростори. Для $y_p^* = \mathbf{1}$ має виконуватися умова $\mathbf{w}^{*T}\mathbf{x}_p > 0$, а для $y_p^* = -1$ — умова $\mathbf{w}^{*T}\mathbf{x}_p < 0$.

Алгоритм навчання може бути записаний у такий спосіб:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \gamma\varepsilon_p\mathbf{x}_{p(k)},$$

де $\mathbf{x}_{p(k)}$ — вхідний вектор на k-му кроці навчання, $\varepsilon_p = y_{p(k)}^* - y(\mathbf{x}_{p(k)})$ — помилка класифікації; γ — параметр, що впливає на швидкість збіжності алгоритму (тривалість процесу навчання). Корекція ваг відповідно може відбуватися в режимах онлайн й офлайн.

У режимі online корекція відбувається при поданні кожної навчальної пари $\left(\mathbf{x}_{p(k)}, y_{p(k)}^*\right)$, $p(k)\epsilon\{1, \dots, P\}$.

У режимі offline у M-му циклі (епосі) навчання подаються всі пари $(\mathbf{x}_p, y_p^*)$ й обчислюється середнє значення помилки класифікації

$$\overline{\varepsilon}_M = \frac{1}{P}\sum_{i=1}^{P}(y_{i,M}^* - y_{i,M}),$$

що й використовується в алгоритмі навчання. Тут $y_{p,M}^*, y_{p,M}$ — бажані й реальні вихідні сигнали при пред'явленні p-ї навчальної пари, в M-му циклі навчання відповідно.

Теорема збіжності для перцептрона, доведена Розенблаттом, стверджує, що *перцептрон може навчитися правильно класифікувати подані йому образи на скінченному числі навчальних пар*.

7.6. Алгоритм зворотного поширення помилки навчання багатошарових нейромереж прямого поширення

Алгоритм зворотного поширення помилки описаний у [11]. Він реалізує градієнтний метод мінімізації квадратичного функціонала помилки в багатошарових мережах прямого поширення, що використовують диференціальні функції активації. Застосування сигмоподібних функцій активації, які мають відмінні від нуля похідні на всій області визначення, забезпечує правильне навчання й функціонування мережі. На кожній епосі навчання процес навчання полягає у послідовному поданні мережі навчальних пар $\left(\mathbf{x}(i), \mathbf{y}^*(i)\right), i = \overline{1,P}$, де $\mathbf{x}(i)$ і $\mathbf{y}^*(i)$ — вектор вхідних і бажаних вихідних сигналів мережі відповідно й корекції вагових коефіцієнтів (елементів вагової матриці) відповідно до реакції мережі.

Перед початком навчання всім вагам *присвоюють невеликі випадкові значення*.

Для реалізації алгоритму зворотного поширення для онлайн-навчання необхідно:

1. Вибрати із заданої навчальної множини чергову навчальну пару $\left(\mathbf{x}(i), \mathbf{y}^*(i)\right),\ i = \overline{1,P}$ та подати на вхід мережі вхідний сигнал $\mathbf{x}(i)$.
2. Обчислити реакцію мережі $\mathbf{y}(i)$.
3. Визначити помилку $\mathbf{y}^*(i) - \mathbf{y}(i)$.
4. Скорегувати ваги так, щоб помилка була мінімальною.
5. Кроки 1-4 повторити для всієї множини пар $\left(\mathbf{x}(i), \mathbf{y}^*(i)\right)\ i = \overline{1,P}$ доти, поки на заданій множині помилка не досягне необхідної величини (або поки не буде завершено усі епохи навчання).

Таким чином, у процесі навчання мережі подача вхідного сигналу й обчислення реакції відповідає *прямому* проходу сигналу від вхідного шару до вихідного, а обчислення помилки й корекція вагових коефіцієнтів — *зворотному*, коли сигнал помилки поширюється по мережі від її виходу до входу. При зворотному проході здійснюється пошарова корекція ваг, починаючи з вихідного шару.

Алгоритм зворотного поширення застосовують також щодо мереж з будь-якою кількістю шарів: як мереж прямого поширення, так і таких, що містять зворотні зв’язки. Обмежимося розглядом випадку навчання двох шарів мережі прямого поширення. Фрагмент такої мережі зображено на рис. 7.24.

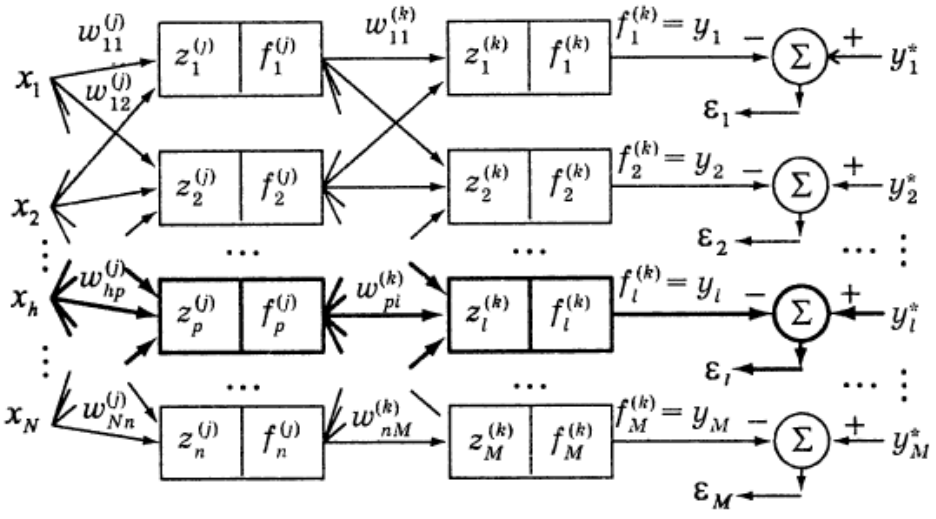


Рис. 7.24. Фрагмент мережі прямого поширення

На рисунку позначено: $w_{pl}^{(k)}$ — вага зв'язку p-го нейрона попереднього шару (j-го) з l-м нейроном наступного (k-го) шару; $f_p^{(j)}$ — активаційна функція p-го нейрона j-го шару; $z_p^{(j)}$ — зважена сума вихідних сигналів попереднього (i-го) шару, що надходять на вхід p-го нейрона наступного (j-го) шару.

Обчислення ваг нейронів вихідного шару

Розглянемо l-й нейрон вихідного (k-го) шару (див. рис. 7.25). Вихід даного нейрона є виходом мережі, тому його сигнал $y_i = f_l^{(k)}$ порівнюється з необхідним y_l^* й обчислюється помилка

$$\varepsilon_l = y_l^* - f_l^{(k)}.$$

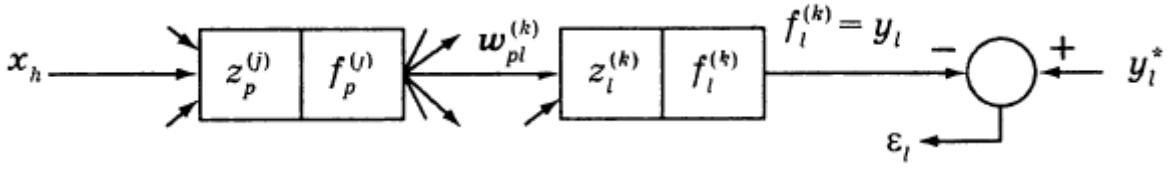


Рис. 7.25. Фрагмент багатошарової мережі

Використання квадратичного критерію якості навчання

$$E_l = \left(y_l^* - f_l^{(k)}\right)^2, \tag{7.4}$$

де E_l — l-та компонента квадратичної похибки мережі E дозволяє отримати градієнтний алгоритм корекції ваг, що у цьому випадку набуває вигляду

$$\Delta w_{pl}^{(k)} = -\gamma_{pl} \frac{\partial E_l}{\partial w_{pl}^{(k)}},$$

де γ_{pl} — коефіцієнт, що впливає на швидкість навчання. Обчислення складної частинної похідної здійснюється за правилом

$$\frac{\partial E_l}{\partial w_{pl}^{(k)}} = \frac{\partial E_l}{\partial f_l^{(k)}} \frac{\partial f_l^{(k)}}{\partial z_l^{(k)}} \frac{\partial z_l^{(k)}}{\partial w_{pl}^{(k)}}. \tag{7.5}$$

З урахуванням (7.4) і того, що $z_l^{(k)} = \sum_{p=1}^{n} w_{pl}^{(k)} f_p^{(j)}$, у випадку застосування логістичної функції активації отримуємо:

$$\frac{\partial E_l}{\partial f_l^{(k)}} = -2(y_l^* - y_l) = -2\varepsilon_l; \tag{7.6}$$

$$\frac{\partial f_l^{(k)}}{\partial z_l^{(k)}} = \alpha f_l^{(k)}(1 - f_l^{(k)}); \tag{7.7}$$

$$\frac{\partial z_l^{(k)}}{\partial w_{pl}^{(k)}} = f_p^{(j)}. \tag{7.8}$$

Підстановка (7.6)-(7.8) у (7.5) дає

$$\Delta w_{pl}^{(k)} = 2\alpha\gamma_{pl}\varepsilon_l f_l^{(k)}\left(1 - f_l^{(k)}\right) f_p^{(j)} \tag{7.9}$$

або

$$w_{pl}^{(k)}(i+1) = w_{pl}^{(k)}(i) + \gamma_{pl}\delta_{pl}^{(k)} f_p^{(j)}, \tag{7.10}$$

де $\delta_{pl}^{(k)} = 2\alpha\varepsilon_l f_l^{(k)}\left(1 - f_l^{(k)}\right)$, i — номер ітерації.

Аналогічно обчислюються ваги інших нейронів вихідного шару. Всі величини, що входять у (7.10), є відомими. Присутня в (7.9), (7.10) помилка ε_l відмінна від нуля внаслідок того, що нейрони вихідного шару виробляють помилкові вихідні сигнали, тому що, по-перше, на початку роботи алгоритму їх ваговим коефіцієнтам присвоєнні випадкові значення, по-друге, нейрони прихованих шарів також виробляють помилкові сигнали. Помилка ε_l поширюється назад на попередні приховані шари й використовується для корекції ваг цих шарів.

Обчислення ваг нейронів прихованого шару

Фрагмент мережі для обчислення ваг нейронів прихованого шару наведено на рис. 7.26.

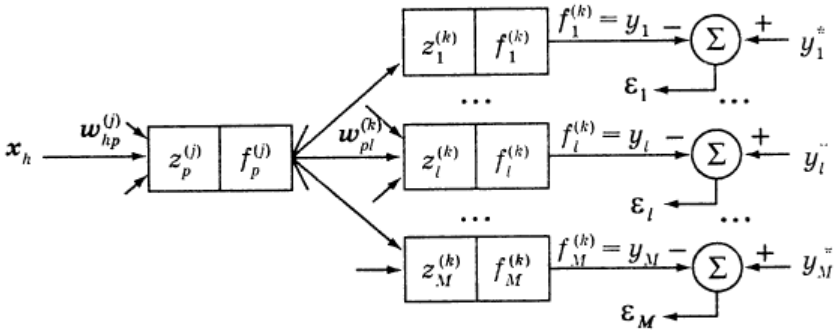


Рис. 7.26. Фрагмент мережі

Розглянемо p-й нейрон прихованого (j-го) шару. Оскільки вихідний сигнал цього нейрона надходить на виходи всіх нейронів вхідного шару, алгоритм налаштування записується в такий спосіб:

$$\Delta w_{hp}^{(j)} = -\gamma_{hp}\frac{\partial E}{\partial w_{hp}^{(j)}} = -\gamma_{hp}\sum_{l=1}^{M}\frac{\partial E_l}{\partial w_{hp}^{(j)}}. \tag{7.11}$$

В останній формулі E_l — квадратична помилка на l-му виході; γ_{hp} — параметр, що виконує ту саму роль, що й γ_{pl} у (7.10).

Оскільки похибка мережі E задовольняє умову $E = \sum_{l=1}^{M} E_l = \sum_{l=1}^{M}\left(y_l^* - f_l^{(k)}\right)^2$, частинна похідна в (7.11), обчислюється так:

$$\frac{\partial E}{\partial w_{hp}^{(j)}} = \sum_{l=1}^{M}\frac{\partial E_l}{\partial f_l^{(k)}}\frac{\partial f_l^{(k)}}{\partial z_l^{(k)}}\frac{\partial z_l^{(k)}}{\partial f_p^{(j)}}\frac{\partial f_p^{(j)}}{\partial z_p^{(j)}}\frac{\partial z_p^{(j)}}{\partial w_{hp}^{(j)}}.$$

Для розглянутого випадку за аналогією з вищевикладеним отримуємо

$$\frac{\partial E_l}{\partial f_l^{(k)}} = -2\left(y_l^* - f_l^{(k)}\right) = -2\varepsilon_l;$$

$$\frac{\partial f_i^{(k)}}{\partial z_i^{(k)}} = \alpha f_i^{(k)}(1 - f_i^{(k)});$$

$$\frac{\partial z_l^{(k)}}{\partial f_p^j} = w_{pl}^{(k)};$$

$$\frac{\partial f_p^{(j)}}{\partial z_p^{(j)}} = \alpha f_p^{(j)}\left(1 - f_p^{(j)}\right);$$

$$\frac{\partial z_p^{(j)}}{\partial w_{hp}^{(j)}} = x_h.$$

Тут враховано, що $z_l^{(k)} = \sum_{p=1}^{M} w_{pl}^{(k)} f_p^{(j)}$; $z_l^{(j)} = \sum_{h=1}^{N} w_{hp}^{(j)} x_h$, а функції активації є логістичними. Таким чином,

$$\frac{\partial E}{\partial w_{hp}^{(j)}} = \sum_{i=1}^{m}(-2)\alpha\varepsilon_l\left[f_l^{(k)}\left(1 - f_l^{(k)}\right)\right] w_{pl}^{(k)} \alpha\left[f_p^{(j)}\left(1 - f_p^{(j)}\right)\right] x_h =$$

$$= -\sum_{l=1}^{M} \delta_{pl}^{(k)} w_{pl}^{(k)} \frac{\partial f_p^{(j)}}{\partial z_p^{(j)}} x_h.$$

Позначимо $\delta_{hp}^{(j)} = \delta_{pl}^{(k)} w_{pl}^{(k)} \frac{\partial f_p^{(j)}}{\partial z_p^{(j)}}$. Тоді

$$\frac{\partial E}{\partial w_{hp}^{(j)}} = -\sum_{l=1}^{M} \delta_{hp}^{(j)} x_h$$

і алгоритм корекції ваг приймає вигляд

$$\Delta w_{hp}^{(j)} = -\gamma_{hp} x_h \sum_{l=1}^{M} \delta_{hp}^{(j)},$$

або

$$w_{hp}^{(j)}(i+1) = \Delta w_{hp}^{(j)}(i) + \gamma_{hp} x_h \sum_{l=1}^{M} \delta_{hp}^{(j)}.$$

Слід зазначити, що використання випадкових початкових значень вагових параметрів призводить до того, що при повторному моделюванні з іншими початковими значеннями форма поверхонь може бути іншою.

На відміну від задач регресії при розв'язуванні задач класифікації використовують ентропійні функції помилки мережі замість квадратичних. Для задач бінарної класифікації використовується *бінарна перехресна ентропія* (*Logistic regression cost function* або log loss):

$$E(\mathbf{w}) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i^* \log y_i + \left(1 - y_i^*\right)\log\left(1 - y_i\right)\right).$$

Її використання передбачає застосування логістичної функції активації у (єдиному) вихідному нейроні мережі. Градієнт цієї функції має вигляд:

$$\nabla E(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}\left(y_k(\mathbf{x}_i) - y_k^*(\mathbf{x}_i)\right)\mathbf{x}_i \ .$$

Для загальної задачі класифікації використовується *перехресна ентропія*:

$$E(\mathbf{w}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^*(\mathbf{x}_i)\log y_k(\mathbf{x}_i),$$

де $y_k^*(\mathbf{x}_i)$ — ймовірність належності i-го навчального вектора $\mathbf{x}_i$ до k-го класу, $y_k(\mathbf{x}_i)$ — k-й фактичний вихід мережі на цьому векторі, отриманий із застосуванням у останньому шарі функції активації *softmax*. Тоді відповідний градієнт (для k-го виходу) рівний

$$\nabla E_k(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}\left(y_k(\mathbf{x}_i) - y_k^*(\mathbf{x}_i)\right)\mathbf{x}_i .$$

Опис бібліотек, у яких реалізовано різні модифікації методів навчання штучних нейромереж, наведено у [12, 13].

Візуалізацію процесу навчання для заданої конфігурації нейромережі можна знайти за посиланням https://playground.tensorflow.org.

7.7. Машинне навчання на платформі TensorFlow

TensorFlow — це повнофункціональна платформа машинного навчання з відкритим кодом. Її можна сприймати як рівень інфраструктури для диференційованого програмування. TensorFlow поєднує в собі чотири ключові можливості [15, 16]:

- Ефективне виконання тензорних операцій низького рівня на CPU, GPU або TPU.
- Обчислення градієнта довільних диференційованих виразів.
- Масштабування обчислень на пристроїв та кластери (наприклад, суперкомп'ютер Frontier в Національній лабораторії Oak Ridge, який охоплює 8,7 млн ядер).
- Експорт програм ("графів" програм) до зовнішніх середовищ виконання, таких як сервери, браузери, мобільні та вбудовані пристрої.

Для доступу до засобів TensorFlow використовується API, написане на мові Python.

Для перевірки наявності TensorFlow та зчитування його поточної версії можна використати наступний скрипт:

```
import tensorflow as tf
tf.__version__
```

7.7.1. Модель обчислень та тензори

TensorFlow 1.X вимагав явного створення графа виразу. Граф виразу — мережа вершин та ребер. У ньому визначені всі дані, які будуть використовуватися (слова, тензори (константи, змінні та заповнювачі (placeholders) та всі обчислення, що підлягають виконанню, а саме об'єкти операції (Operation Objects)). Кожна вершина може мати нуль або більше входів, *але лише один вихід*. Вершини графа зображають об'єкти (тензори та операції), а ребра представляють тензори, що передаються між операціями. Граф обчислень визначає проект (blueprint) нейронної мережі.

Приклад графу обчислень наведено на рис. 7.27.

TensorFlow 2.X виконує обчислення за «жадібною» схемою і в для нього графи та сеанси розглядаються як деталі реалізації (backend), безпосереднє використання яких необов'язкове.

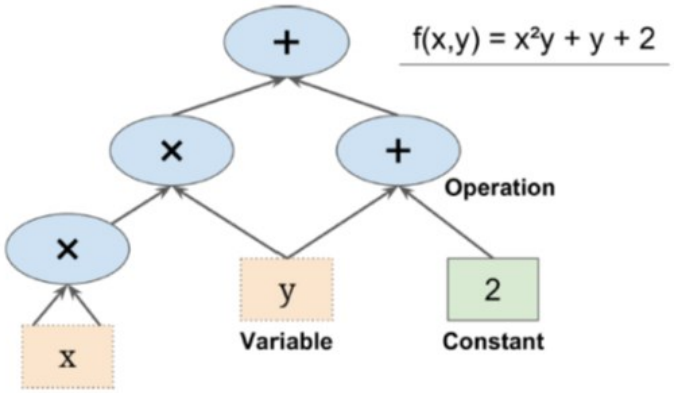


Рис. 7.27. Граф обчислень функції $f(x, y)$

Можна вважати, що тензор — це n-вимірний масив. Усі типи даних, тобто скалярні, вектори та матриці, є спеціальними типами тензорів, як це наведено на рис. 7.28.

Types of data	Tensor	Shape
Scalar	0-D Tensor	[]
Vector	1-D Tensor	$[D_0]$
Matrix	2-D Tensor	$[D_0,D_1]$
Tensors	N-D Tensor	$[D_0,D_1,....D_{n-1}]$

Рис. 7.28. Типи тензорів

Над тензорами можна виконувати покоординатні операції та здійснювати агрегацію:

```
a = tf.constant([1, 5, 3])
b = tf.constant([4, 1, -2])
print(a + 2 * b)
print(a * b)
print(tf.tensordot(a, b, axes=1))
```

7.7.2. Реалізація нейромережі прямого поширення засобами бібліотеки Keras

Keras — це високорівневий API для глибинного навчання, який дозволяє легко будувати, навчати, оцінювати та використовувати різноманітні нейронні мережі на платформі TensorFlow [19, 21].

Місце Keras у структурній схемі процесу машинного навчання проілюстровано на рис. 7.29.

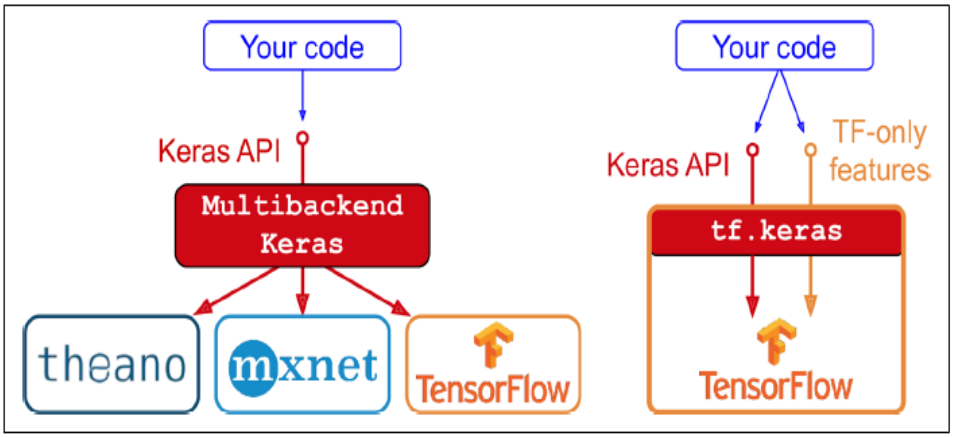


Рис. 7.29. Розташування Keras у архітектурі машинного навчання

Основними структурами даних Keras є шари (layers) та моделі (models) [20]. Найпростішим типом моделей є *послідовна модель* (*Sequential model*), яка призначена для моделювання нейромереж прямого поширення 1-го порядку.

Розглянемо основні дії, пов'язані із послідовними моделями (на прикладі їх застосуванням для розв'язування задачі класифікації):

1. Створення моделі:

```
from tensorflow.keras.models import Sequential

model = Sequential()
```

2. Додавання шарів:

```
from tensorflow.keras.layers import Dense

model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

3. Визначення параметрів моделі та її компіляція:

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

На цьому кроці вказується функція втрат (*loss functions*), яка є функцією помилки мережі, мінімізація якої відбувається у процесі навчання з використанням обраного алгоритму оптимізації (*optimizer*). Метрика (*metric*) — це функція, яка використовується для оцінки якості функціонування (performance) моделі.

4. Навчання моделі (`x_train` та `y_train` — масиви `Numpy`):

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

5. Обчислення значення функції втрат та обраних метрик на тестовій вибірці:

```
performance = model.evaluate(x_test, y_test, batch_size=128)
```

6. Генерація прогнозу для нових даних:

```
classes = model.predict(x_new, batch_size=128)
```

Приклад. Навчити ШНМ з одним прихованим шаром реалізувати функцію XOR (див. рис. 7.29).

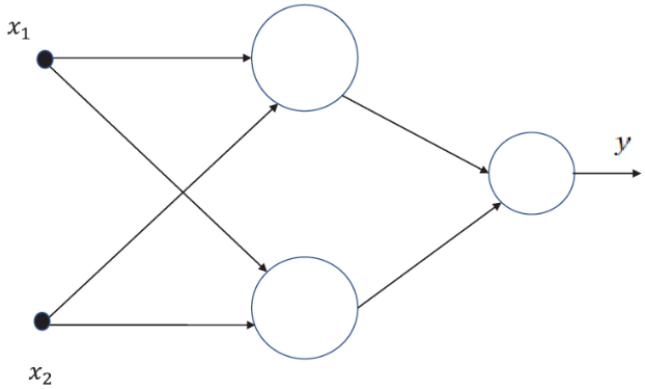


Рис. 7.29. Нейромережа для реалізації XOR

ЛІТЕРАТУРА

1. Глибовець М. М., Олецький О. В. Системи штучного інтелекту. К.: КМ Академія, 2002. 366 с.
2. Рассел С., Норвиг П. Искусственный интеллект. Современный поход. М.: Вильямс, 2006. 1408 с.
3. Люгер Дж. Искусственный интеллект. Стратегии и методы решения сложных проблем. М.: Вильямс, 2003. 864 с.
4. Смолин Д.В. Введение в искусственный интеллект: конспект лекций. М.: ФИЗМАТЛИТ, 2004. 208 с.
5. Братко И. Алгоритмы искусственного интеллекта на языке Пролог. М.: Вильямс, 2004. 640 с.
6. Девятков В. В. Системы искусственного интеллекта. М.: Изд-во МГТУ им. Баумана, 2001. 352 с.
7. Субботін С. О. Подання й обробка знань у системах штучного інтелекту та підтримки прийняття рішень. Запоріжжя: ЗНТУ, 2008. 341 с.
8. Представление и использование знаний / Под ред. Уэно Х., Исидзука М. М.: Мир, 1989. 220 с.
9. Искусственный интеллект: Справочник: В 3-х т. М.: Радио и связь, 1990.
10. Нильсон Н. Принципы искусственного интеллекта. М.: Радио и связь, 1985. 376 с.
11. Руденко О. Г., Бодянський Є. В. Штучні нейронні мережі: Навчальний посібник. Харків: ТОВ "Компанія СМІТ", 2006. 404 с.
12. Шаховська Н. Б., Камінський Р. М., Вовк О. Б. Системи штучного інтелекту: навч. посіб. Львів: Львівська політехніка, 2018. 392 с.
13. Ткаченко Р. О., Кустра Н. О., Павлюк О. М., Поліщук У. В. Засоби штучного інтелекту: навч. посіб. Львів: Вид-во Львів. політехніки, 2014. 204 с.
14. Хайкин С. Нейронные сети: полный курс, 2-е изд. М.: Вильямс-Телеком, 2006. 1104 с.
15. Haykin S. Neural networks and Learning Machines 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2008. 906 с.
16. Николенко С., Кадурин А., Архангельская Е. Глубокое обучение. СПб.: Питер, 2018. 480 с.
17. Жерон О. Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow. М.: Вильямс, 2018. 688 с.
18. Géron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, 2nd ed. Sebastopol, CA: O'Reilly Media, 2019. 820 с.
19. Рашка С., Мирджалили В. Python и машинное обучение: машинное и глубокое обучение с использованием Python, Scikit-Learn и TensorFlow 2, 3-е изд., М.: ООО "Диалектика", 2020. 848 с.
20. Raschka S., Mirjalili V. Python Machine Learning Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow 2, 3rd ed. Birmingham: Packt, 2019. 742 с.
21. Шолле Ф. Глубокое обучение на Python. СПб: Питер, 2018. 397 с.
22. Chollet F. Deep Learning with Python, 2nd ed. Shelter Island, NY: Manning Publications 2021, 478 с.
23. Raschka S., Yuxi L, Mirjalili V, Dr. Vahid Mirjalili. Machine Learning with PyTorch and Scikit-Learn. Birmingham: Packt, 2022. 742 с.