ACADEMIA DE STUDII ECONOMICE DIN MOLDOVA

OPREA Serghei

BAZELE INTELIGENȚEI ARTIFICIALE

Note de curs



CHIȘINĂU, 2026

CZU 004.8(075.8)
O-64

Lucrarea „Bazele inteligenței artificiale”. Notele de curs au fost examinate și aprobate pentru editare la ședința Departamentului „Tehnologia Informației și Management Informațional” (proces verbal nr. 1 din 01.09.2026) și la ședința Comisiei metodice a facultății „Tehnologii Informaționale și Statistică Economică” (proces verbal nr. 2 din 08.10.2026)

Referenți:
Lozan Victoria, dr., conf. univ., ASEM
Zgureanu Aureliu, dr., conf. univ., ASEM

DESCRIEREA CIP A CAMEREI NAȚIONALE A CĂRȚII DIN REPUBLICA MOLDOVA

Oprea, Serghei.

Bazele inteligenței artificiale : Note de curs / Oprea Serghei ; Academia de Studii Economice din Moldova. – Chișinău : [S. n.], 2026 (SEP ASEM). – 156 p. : fig., tab. color.

Cerințe de sistem: PDF Reader.

Bibliogr.: p. 155-156 (41 tit.). – În red. aut.

ISBN 978-9975-182-46-1 (PDF).

Text : electronic.

004.8(075.8)

O-64

ISBN 978-9975-182-46-1 (PDF).

© Serghei OPREA, dr., conf. univ., 2026

© Serviciul Editorial-Poligrafic ASEM, 2026

SUMAR

PREFAȚĂ 7

CAPITOLUL 1. INTRODUCERE ÎN INTELIGENȚA ARTIFICIALĂ 8

1.1. Inteligența naturală și inteligența artificială 8

1.2. Definiții ale Inteligenței Artificiale 9

1.3. Testul Turing și Conceptul de „Mașină Gânditoare” 10

1.4. Istoricul și Evoluția Domeniului: De la IA Simbolică la IA Generativă 10

1.5. Domenii de cercetare în IA 13

1.6. Aplicații economice ale IA 13

1.7. IA Responsabilă: Provocările Etice 14

CAPITOLUL 2. AGENȚI INTELIGENȚI ȘI MEDII DE LUCRU 15

2.1. Conceptul de agent și relația agentului cu mediul 15

2.2. Ciclul percepție – decizie – acțiune 15

2.3. Raționalitatea, autonomia și învățarea agentului 16

2.4. Cadrul PEAS — metodă de proiectare a specificațiilor unui agent 17

2.5. Proprietățile mediilor de lucru 18

2.6. Tipuri de agenți inteligenți 19

CAPITOLUL 3. REZOLVAREA PROBLEMELOR PRIN CĂUTARE NEINFORMATĂ. 22

3.1. Formularea problemelor ca spații de stări 22

3.2. Grafuri și arbori de căutare 23

3.3. Algoritmul BFS (Breadth-First Search) 24

3.4. Algoritmul Depth-First Search 25

3.5. Iterative Deepening Depth-First Search (IDDFS) 26

3.6. Algoritmul UCS (Uniform Cost Search) 27

3.7. Exemple practice și sinteză comparativă 28

CAPITOLUL 4. CĂUTAREA INFORMATĂ ȘI CĂUTAREA ADVERSARIALĂ 29

4.1. Funcția euristică h(n) 29

4.2. Algoritmul Greedy Best-First Search 29

4.3. Algoritmul A* Search 30

4.4. Euristici 31

4.5. Căutare în medii adversariale Algoritmul Minimax. 33

4.6. Optimizarea Alpha-Beta Pruning 34

CAPITOLUL 5. REPREZENTAREA CUNOȘTINȚELOR ȘI MODELAREA RAȚIONAMENTULUI 37
5.1. Reprezentarea cunoștințelor în IA clasică (Symbolic AI) 37
5.2. Logica propozițională: sintaxa, semantica, inferența 38
5.3. Logica de ordinul întâi (FOL): predicate, cuantificatori, constante și variabile 41
5.4. Baze de fapte și reguli de inferență 43
5.5. Arhitectura unui sistem de producție 43
5.6. Algoritmi de inferență: Forward Chaining și Backward Chaining 45
CAPITOLUL 6. FUNDAMENTELE PROGRAMĂRII LOGICE ÎN PROLOG 48
6.1. Introducere în programarea logică 48
6.2. Alfabetul și separatori Prolog 48
6.3. Fapte (facts) 48
6.4. Reguli (rules) 50
6.5. Interogări (queries) 51
6.6. Variabile 51
6.7. Sintaxa termenilor și a expresiilor 52
6.8 . Unificarea termenilor (unification) 53
6.9. Recursivitate 54
6.10. Structuri de date: liste 55
6.11. Backtracking 56
6.12. Predicatele fail și cut (!) 57
6.13. Modificarea dinamică a programului 58
CAPITOLUL 7. SISTEME EXPERT ȘI SISTEME SUPORT PENTRU DECIZII 60
7.1. Contextul suportului decizional în organizații 60
7.2. Sisteme suport pentru decizii (SSD) 60
7.3. Sisteme expert (SE) 61
7.4.. Metode de reprezentare a cunoștințelor 64
7.5. Sisteme Expert versus Sisteme Suport pentru Decizii 66
7.6. Aplicații în mediul economic 67
CAPITOLUL 8. RAȚIONAMENT ÎN CONDIȚII DE INCERTITUDINE 69
8.1. Incertitudinea și raționamentul 69
8.2. Logica fuzzy 69
8.3. Factori de certitudine 76
8.4. Probabilitatea condiționată și regula lui Bayes 78

8.5. Rețele Bayesiene: structură și semantică 80
8.6. Calculul probabilităților într-o rețea Bayesiană simplă 82
8.7. Studiu individual: Filtrele de spam 83
CAPITOLUL 9. INTRODUCERE ÎN MACHINE LEARNING 86
9.1.. De la programarea clasică la Machine Learning 86
9.2. Structura unui set de date 87
9.3. Etapele unui proiect de Machine Learning 87
9.4. Tipuri de învățare automată 88
9.5. Fluxul complet al unui proiect ML. Supra-antrenare și generalizare 89
CAPITOLUL 10. ÎNVĂȚARE SUPERVIZATĂ 91
10.1. Regresia liniară 91
10.2. Clasificare prin regresie logistică 93
10.3. Metrici de evaluare a modelelor 94
10.4. Arbori de decizie 95
10.5. Algoritmul k-Nearest Neighbors (k-NN) 96
10.6. Masini cu vectori suport (SVM) 97
10.7. Metode ansamblu: Random Forest 98
CAPITOLUL 11. ÎNVĂȚARE NESUPERVIZATĂ 101
11.1 Caracteristica învățării nesupervizate 101
11.2. Concepte generale despre Clustering 102
11.3. Algoritmul K-Means 102
11.4. Clustering ierarhic 104
11.5. Reducerea dimensionalității — PCA 106
CAPITOLUL 12. REȚELE NEURONALE ARTIFICIALE 110
12.1. Perceptronul. 110
12.2. Funcția de activare 112
12.3. Rețele neuronale multistrat (Multi-Layer Perceptron) 113
12.4. Algoritmul de propagare inversă 115
CAPITOLUL 13. ÎNVĂȚAREA PROFUNDĂ (DEEP LEARNING) 119
13.1. Deep Learning ca ramură a Machine Learning 119
13.2. Rețele neuronale convoluționale 120
13.3. Transfer learning 124
CAPITOLUL 14. PROCESAREA LIMBAJULUI NATURAL (NLP) 128
14.1. Definiție și domenii de aplicare 128
14.2. Modele de limbaj n-gram 129

14.3. Reprezentări vectoriale ale cuvintelor (Word Embeddings) 130
14.4. Rețele neuronale recurente (RNN / LSTM) 132
14.5. Mecanismul de atenție (Attention) 132
14.6. Calcularea Auto-Atenției (exemplu practic) 133
14.7. Arhitectura Transformer 135
14.8. Modele de limbaj de mari dimensiuni (LLM) 136
CAPITOLUL 15. INTELIGENȚA ARTIFICIALĂ GENERATIVĂ 139
15.1. De la IA discriminativă la IA generativă 139
15.2. Modele generative de text 139
15.3. Modele generative de imagine 140
15.4. Prompt Engineering: principii și tehnici 142
15.5. Aplicații ale LLM în mediul economic și administrativ 144
15.6. Limitări, riscuri și considerații etice 145
CAPITOLUL 16. ETICA ȘI REGLEMENTAREA INTELIGENȚEI ARTIFICIALE 147
16.1. Etica și Inteligența Artificială 147
16.2. Bias-ul algoritmic 147
16.3. Transparență și explicabilitate (XAI) 148
16.4. Regulamentul UE privind IA (AI Act) 150
16.5 Alte subiecte importante în etica IA 151
16.6. Tendințe: agenți IA și IA multimodală 152
BIBLIOGRAFIE 155

PREFAȚĂ

Inteligența artificială a devenit în ultimele decenii una dintre direcțiile cele mai dinamice ale tehnologiei informației. Sistemele inteligente sunt utilizate astăzi în domenii diverse de activitate a omului: de la analiza datelor economice și financiare, diagnosticarea tehnică și medicală, până la sistemele de recomandare, asistenții virtuali și procesarea limbajului natural. Complexitatea problemelor abordate și volumele mari de date disponibile impun specialistului în domeniul informatic cunoașterea atât a metodelor clasice de reprezentare a cunoștințelor și de raționament automat, cât și a tehnicilor moderne de învățare automată. Din aceste considerente, studierea fundamentelor inteligenței artificiale este o componentă importantă a formării profesionale a viitorilor specialiști în tehnologia informației.

În cadrul notelor de curs sunt explicate conceptele de bază ale inteligenței artificiale: noțiunea de agent inteligent și relația acestuia cu mediul de lucru, formularea problemelor ca spații de stări, algoritmii de căutare neinformată și informată, precum și metodele de căutare în medii adversariale. De asemenea, sunt prezentate fundamentele logicii propoziționale și ale logicii predicatelor, principiile programării logice în limbajul Prolog, modelele de reprezentare a cunoștințelor și structura sistemelor expert. Un loc aparte este acordat metodelor de raționament în condiții de incertitudine, inclusiv logicii fuzzy și rețelelor bayesiene. La final sunt examinate succint principiile învățării automate și ale rețelelor neuronale artificiale. Fiecare temă este însoțită de exemple aplicative din domeniul economic și informatic, de întrebări de autoevaluare și de lucrări practice.

Lucrarea este dedicată studierii disciplinei ”Bazele inteligenței artificiale”, inclusă în planurile de învățământ ale specialității Tehnologia informației de la ciclul I universitar (licență) al Academiei de Studii Economice a Moldovei. Scopul de bază al lucrării constă în formarea la studenți a cunoștințelor teoretice și a deprinderilor practice de proiectare, implementare și evaluare a sistemelor cu inteligență sporită. Disciplina are un caracter fundamental și creează baza conceptuală necesară pentru studierea ulterioară a altor unități de curs din domeniu. Astfel, noțiunile de agent inteligent, spațiu de stări și funcție euristică, precum și principiile de raționament în condiții de incertitudine și bazele rețelelor neuronale, sunt utilizate în cadrul cursului ”Machine Learning”. Metodele de reprezentare a cunoștințelor, logica predicatelor și programarea logică în Prolog constituie fundamentul cursului ”Sisteme bazate pe cunoștințe”, iar structura sistemelor expert, mecanismele de inferență și raționamentul aproximativ sunt dezvoltate în continuare în cadrul cursului ”Sisteme expert”.

Notele de curs pot fi utilizate ca suport metodic pentru studiul individual al studenților de la secția de zi și cu frecvență redusă. În același timp, lucrarea poate servi ca sursă de informare pentru diverse categorii de cititori care doresc să se inițieze în domeniul inteligenței artificiale și al aplicațiilor sale practice.

CAPITOLUL 1. INTRODUCERE ÎN INTELIGENȚA ARTIFICIALĂ

1.1. Inteligența naturală și inteligența artificială

Noțiunea de **inteligență** se referă la capacitatea mintală a omului de a gândi și de a cunoaște rațional lumea înconjurătoare. Psihologia identifică 116 tipuri de inteligență umană, 12 din ele pot fi măsurate prin teste de inteligență.

Inteligența umană reprezintă ***ansamblul funcțiilor mintale orientate spre cunoașterea conceptuală și rațională, care se opune senzației și intuiției***.

Inteligența înseamnă și facultatea de a cunoaște și a înțelege lucruri, fapte, idei; aptitudinea de adaptare la situații noi; capacitatea de gândire și comunicare. Inteligența este tratată în dicționarele explicative ***ca capacitatea de a învăța sau înțelege din experiență; abilitatea de a achiziționa și memora cunoștințe; facilitatea de a raționa în procesul rezolvării problemelor și de a conduce activitățile cu eficacitate***.

Omul, posesorul de inteligență naturală, o manifestă prin modalități diferite:

Inteligența lingvistică **–** se referă la capacitatea de a citi, scrie, povesti sau rezolva cuvinte încrucișate.

Inteligența logico-matematică presupune descoperirea modelelor, categoriilor și relațiilor. Se manifestă, de exemplu, în rezolvarea problemelor aritmetice sau în jocurile de strategie.

Inteligența spațială se referă la posibilitatea de a gândi în imagini și la abilitatea rezolvării unor probleme geometrice spațiale. De exemplu, găsirea drumului într-un labirint, talentul de a desena sau de a construi figuri din cuburi Lego.

Inteligența corporal-chinestezică implică o mare sensibilitate în identificarea și prelucrarea senzațiilor fizice, de exemplu, a simți ritmul unui dans.

Inteligența muzicală – presupune posedarea urechii muzicale și abilității de a percepe și distinge sunete care par la fel altor persoane**.**

Inteligența interpersonală este dovedită de spiritul de conducător, de ușurința comunicării și de existența empatiei, adică a capacității de a înțelege sentimentele altora.

Inteligența intrapersonală reflectă o bună cunoaștere a propriilor sentimente și posibilități.

Inteligența biologică reprezintă capacitatea de a recunoaște modele în natură, de a înțelege diferite specii de faună sau floră și de a clasifica obiectele naturale**.**

Inteligența existențială este capacitatea de a trata problemele filosofice ale vieții.

Inteligența pedagogică reprezintă capacitatea de a instrui simplu și eficient alte persoane.

Spunem că omul poseda inteligență dacă el dispune de următoarele capacități:

- ✓ **Percepție** – capacitatea de a face observații complexe asupra mediului;
- ✓ **Prelucrare a informațiilor** – capacitatea de a gestiona și transforma informații, utilizând toate formele de raționament;
- ✓ **Memorie** – capacitatea de a stoca și refolosi informații;
- ✓ **Învățare** – capacitatea de a dezvolta noi cunoștințe și abilități și de a folosi experiența acumulată;
- ✓ **Adaptabilitate** – capacitatea de a se adapta în mod flexibil comportamentul la situațiile curente prin schimbări aduse propriei persoane sau mediului;
- ✓ **Procesarea intrărilor și ieșirilor**: presupune preluarea datelor din mediul înconjurător și obținerea din ele a informațiilor noi – filtrarea și clasificarea informațiilor obținute – memorarea informațiilor – generalizează informațiilor și obținerea în baza lor a cunoașterilor noi – utilizarea informațiilor și cunoașterii noi în activitățile cotidiene;
- ✓ **Optimizare** – capacitatea de a folosi în mod optim resurse limitate, inclusiv timpul, pentru a atinge scopuri;
- ✓ **Planificare** – capacitatea de a identifica scopuri și metodele de atingere a lor.

Inteligența artificială ca o ramură a științei își propune sarcina să imite caracteristicile inteligenței umane prin intermediul sistemelor informatice și programelor inteligente, având ca scop final comportamentul inteligent al calculatoarelor.

Termenul de **inteligență artificială** a fost formulat pentru prima oară în 1956 de J.McCarthy și în prezent are semnificația de domeniu de cercetare, al cărui scop constă în:

- crearea sistemelor informatice capabile să îndeplinească activități inteligente (neuro-cibernetica);
- studiul și modelarea mecanismelor inteligenței.

Cercetarea în domeniul inteligenței artificiale are un caracter fundamental și aplicativ. Inteligența artificială este o ramură de frontieră a științei, care realizează interconexiunea unor științe ca filozofia, psihologia, medicina, matematica, logica, cibernetica, robotica, biologia, lingvistica etc.

1.2. Definiții ale Inteligenței Artificiale

Pentru a înțelege IA, trebuie mai întâi să o definim. Surprinzător, nu există o singură definiție universal acceptată, deoarece însăși noțiunea de „inteligență" este profund dezbătută în psihologie și filosofie. Câteva definiții sunt prezentate mai jos:

- ✓ **Definiția 1**. *Inteligența artificială reprezintă un ansamblu de discipline informatice care tind să realizeze pe calculator un comportament de imitare a inteligenței naturale.*
- ✓ **Definiția 2.** *Inteligența artificială este un domeniu al informaticii care are obiectiv conceperea sistemelor de calcul inteligente, care manifestă proprietăți ce sunt în mod obișnuit asociate inteligenței umane: înțelegerea limbajului, învățarea, raționamentul, soluționarea problemelor etc.*
- ✓ **Definiția 3.** *Inteligența artificială este studiul ideilor care permit calculatoarelor să efectueze lucruri care îi determină pe oameni să le considere inteligente.*
- ✓ **Definiția 4.** *Inteligența artificială are ca obiect studiul posibilităților calculatoarelor de a gândi mai repede ca omul.*
- ✓ **Definiția 5.** *Inteligența artificială este știința modelării inteligenței umane.*
- ✓ **Definiția 6.** *Inteligența artificială este arta de a crea mașini care să îndeplinească funcții care ar necesita inteligență dacă ar fi îndeplinite de oameni*

În esență, putem clasifica definițiile IA în patru categorii principale, structurate pe două axe: **gândire vs. comportament** și **raționalitate vs. replicarea comportamentului uman** (clasificarea clasică a lui Russell și Norvig):

Cele patru perspective ale IA

- **Sisteme care gândesc ca oamenii:** Sisteme inspirate din neuroștiințe și psihologia cognitivă, care încearcă să replice procesul de decizie uman (ex. rețele neuronale artificiale complexe).
- **Sisteme care acționează ca oamenii:** Sisteme capabile să execute sarcini în mod similar cu un om (aici se încadrează Testul Turing, despre care vom vorbi imediat).
- **Sisteme care gândesc rațional:** Abordarea bazată pe legile logicii formalizate (ex. „Dacă X este adevărat și Y este adevărat, atunci Z este adevărat").
- **Sisteme care acționează rațional:** Agenți inteligenți care încearcă să obțină cel mai bun rezultat posibil (sau cel mai bun rezultat estimat în condiții de incertitudine), optimizând o funcție de utilitate.

„În plus la cele patru perspective, în literatura modernă, se examinează conceptul de Agent Inteligent: un sistem care percepe mediul, procesează informația și acționează pentru a atinge un obiectiv. Exemplu: O mașină autonomă nu doar 'gândește', ci 'acționează' în timp real."

Definiție generală de lucru: Inteligența Artificială reprezintă acea ramură a informaticii care se ocupă cu crearea de sisteme capabile să execute sarcini care, în mod normal, ar necesita inteligență umană – cum ar fi învățarea, raționamentul, rezolvarea de probleme, percepția vizuală și înțelegerea limbajului natural.

Tipuri de IA (în funcție de capacitate)

1. **IA Slabă (Narrow / Weak AI):** Sisteme proiectate și antrenate pentru o singură sarcină specifică (ex. filtrele de spam, algoritmii de șah, recunoașterea facială, chat LLM). Toată IA din prezent este IA slabă.
2. **IA Puternică / Generală (AGI – Artificial General Intelligence):** Un sistem ipotetic care posedă capacitatea de a înțelege, învăța și aplica cunoștințe în orice domeniu intelectual, la fel ca o ființă umană.
3. **Superinteligența Artificială (ASI):** O formă de IA care depășește cu mult intelectul uman în absolut toate domeniile.

1.3. Testul Turing și Conceptul de „Mașină Gânditoare”

În anul 1950, legendarul matematician britanic **Alan Turing** a publicat o lucrare fundamentală numită *„Computing Machinery and Intelligence”*. El a început acest articol cu o întrebare simplă, dar provocatoare: *„Pot mașinile să gândească?”*

Realizând că definirea termenilor „a gândi” și „mașină” poate duce la dezbateri semantice infinite, Turing a propus un experiment practic, numit inițial **„Jocul Imitației”** și cunoscut astăzi sub numele de **Testul Turing**.

Cum funcționează testul?

- ✓ Avem trei participanți aflați în camere separate: un om (A), o mașină/calculator (B) și un interogator uman (C).
- ✓ Interogatorul (C) comunică cu ceilalți doi participanți exclusiv prin intermediul unor mesaje text (tastatură/ecran), fără a ști cine este omul și cine este mașina.
- ✓ Scopul interogatorului este de a determina, pe baza răspunsurilor primite la întrebări libere, care este omul și care este mașina.
- ✓ **Criteriul de succes:** Dacă mașina reușește să păcălească interogatorul în mod repetat, făcându-l să creadă că ea este omul, atunci mașina a trecut testul și poate fi considerată „inteligentă”.

Critici și Limite: „Camera Chineză”

Deși revoluționar, Testul Turing măsoară doar **comportamentul exterior**, nu și înțelegerea reală. În 1980, filozoful John Searle a propus experimentul mental numit **„Camera Chineză”**. El a demonstrat că un om blocat într-o cameră, care nu știe deloc limba chineză, dar folosește un manual complex de reguli („dacă vezi simbolul X, răspunde cu simbolul Y”), poate părea din exterior că vorbește chineză perfect. Totuși, el nu înțelege o boabă. La fel și calculatoarele: ele manipulează simboluri conform unor reguli matematice (sau probabilistice), dar fără a avea o înțelegere conștientă a conceptelor.

1.4. Istoricul și Evoluția Domeniului: De la IA Simbolică la IA Generativă

Evoluția IA nu a fost o linie dreaptă și ascendentă, ci mai degrabă o succesiune de perioade de entuziasm masiv urmate de prăbușiri financiare, cunoscute sub numele de „Iernile IA” (AI Winters).

A. Fundamentele Filosofice și Tehnice (1930-1950)

Această perioadă a stabilit bazele conceptuale necesare pentru dezvoltarea ulterioară a IA, concentrându-se pe definirea calculului și a criteriului de inteligență.

- **Viziunea Calculului Universal (1936):** În 1936, matematicianul britanic Alan Turing a pus bazele informaticii moderne, introducând conceptul teoretic al "mașinii universale de calcul" (Mașina Turing). Această viziune a descris pentru prima dată un singur dispozitiv programabil, capabil să execute orice sarcină computațională, un concept care stă la baza fiecărui computer digital de astăzi.

- **Testul Turing și Criteriul Inteligenței (1950):** Un moment definitoriu a avut loc în 1950, când Turing a propus "Jocul Imitației" (*The imitation game*), cunoscut în prezent sub numele de Testul Turing, Testul stabilește un cadru practic pentru determinarea inteligenței la nivel uman. Restricția impusă de Turing de a folosi un „chat exclusiv bazat pe text" a avut o influență majoră asupra direcției inițiale a cercetării IA. Prin ignorarea complexităților percepției vizuale sau motricității, cercetătorii timpurii s-au concentrat pe manipularea simbolică și logică a limbajului. Deși această predilecție pentru sistemele bazate pe reguli a dominat până în anii 1980, s-a demonstrat ulterior că aceste sisteme aveau dificultăți majore în a scala la complexitatea "cunoașterii de bun simț" din lumea reală, o limitare care avea să fie o cauză majoră a Primului AI Winter.

B. Nașterea Domeniului și IA Simbolică (Anii 1950 - 1980)

- **Conferința de la Dartmouth (1956):** Este momentul zero. John McCarthy, Marvin Minsky, Allen Newell și Herbert Simon au pus bazele disciplinei, iar McCarthy a inventat oficial termenul de „Inteligență Artificială". Acest eveniment a generat un optimism remarcabil. Guvernul SUA, în special prin Advanced Research Projects Agency (ARPA, ulterior DARPA), a început să ofere milioane de dolari în finanțare pentru instituții precum MIT, Carnegie Mellon University (CMU) și Stanford, anticipând că mașinile inteligente vor fi create în decursul unei singure generații.
- **IA Simbolică (GOFAI - Good Old-Fashioned AI):** Primele decenii au fost dominate de ideea că inteligența se bazează pe manipularea de simboluri logice. Se credea că dacă putem traduce regulile lumii în cod logic pur, vom crea o IA. În anii '60, au apărut primele programe de procesare a limbajului natural (NLP): **ELIZA(1966) -** Dezvoltat de Joseph Weizenbaum la MIT între 1964 și 1967, ELIZA a fost unul dintre primii *chatbots*. Programul simula conversația utilizând o metodologie de potrivire de șabloane și substituție (*pattern matching and substitution*). Deși nu avea o reprezentare internă a înțelegerii, ELIZA a dat utilizatorilor **o iluzie convingătoare** de dialog. Weizenbaum a fost surprins de **Efectul ELIZA**, observând că oamenii atribuiau sentimente umane programului. **SHRDLU (1970) -** Construit de Terry Winograd la MIT, SHRDLU era capabil să înțeleagă intrările utilizatorului și să execute instrucțiuni complexe în limbaj natural într-un mediu virtual restrâns, cunoscut sub numele de "lumea blocurilor". Capacitatea programelor precum ELIZA de a crea o iluzie convingătoare de inteligență fără a poseda o înțelegere semantică reală dezvăluie un model istoric recurent în IA. Această iluzie a condus la așteptări nerealiste. Atunci când sistemele simbolice bazate pe reguli nu au reușit să scaleze de la micro-domenii (precum "lumea blocurilor") la problemele complexe, generale, ale lumii, a început dezamăgirea.
- **Primul AI Winter(Iarna Inteligenței Artificiale) (1974)** Eșecul de a realiza progrese la nivel uman a dus la critici semnificative și, în 1974, guvernele SUA și Marea Britanie au oprit finanțarea cercetării nedirecționate în IA. Această oprire a finanțării a marcat începutul Primului AI Winter.
- **Sistemele Expert (Anii 1970-1980):** Sistemele Expert (SE) au reprezentat prima formă de software IA cu adevărat de succes. Dezvoltate inițial în anii 1970, acestea au proliferat și au fost considerate viitorul IA în anii 1980. Succesul SE s-a bazat pe o metodologie eficientă: restrângerea domeniului de aplicare la o arie de cunoaștere specifică. Această restricție a permis sistemelor să evite capcana "cunoașterii de bun simț" care îngropase încercările anterioare. Un Sistem Expert este compus din două subsisteme principale: o bază de cunoștințe (care stochează faptele și regulile de tipul „IF-THEN" derivate de la experți umani) și un motor de inferență (care aplică regulile pentru a deduce noi fapte și a ajunge la soluții). Exemple cheie care au demonstrat viabilitatea abordării SE includ: **DENDRAL (început 1965) -** Capabil să identifice compuși chimici pe baza citirilor spectrometrului;

MYCIN (dezvoltat 1972) – Un sistem de diagnosticare a bolilor infecțioase ale sângelui. Aceste sisteme au dovedit că IA poate fi *utilă*, reușind să atragă din nou investițiile.

- **Boom-ul IA (1980–1990):** Succesul Sistemelor Expert a revigorat investițiile la șapte ani după oprirea finanțării guvernamentale. Între 1980 și 1987, finanțarea guvernamentală (stimulată și de inițiative vizionare precum cea a Guvernului Japonez) a crescut din nou. Până la sfârșitul anilor 1980, industria Sistemelor Expert devenise o întreprindere de miliarde de dolari. Deși SE au fost un succes comercial imediat, limitările lor structurale au plantat semințele următoarei dezamăgiri. Sistemele expert nu aveau capacitatea de a învăța sau de a se instrui automat; ele funcționau pe baza cunoștințelor codificate manual de către oameni. Din acest motiv, ele erau inflexibile și nu se puteau adapta ușor la schimbările din mediul de operare. Costurile mari de întreținere a bazelor de cunoștințe codificate manual au dus în cele din urmă la pierderea entuziasmului investitorilor la începutul anilor 1990.
- **Al Doilea AI Winter (Anii 1990):** Dezamăgirea generată de costurile și inflexibilitatea Sistemelor Expert a dus la Al Doilea AI Winter în anii 1990. Calculatoarele de atunci s-au lovit de limitele puterii de calcul și de incapacitatea de a gestiona incertitudinea și nuanțele din lumea reală. Investitorii și-au pierdut entuziasmul, iar domeniul a fost criticat în presă și evitat de industrie, și finanțările au fost tăiate complet. Cu toate acestea, în ciuda denumirii de "iarnă," cercetarea fundamentală și finanțarea au continuat să crească, dar adesea sub denumiri alternative, cum ar fi "Agenți Inteligenți" sau "Învățare Automată" (Machine Learning). Cercetările au adoptat o abordare mai pragmatică, concentrându-se pe rezolvarea problemelor înguste și pe o rigoare matematică sporită.

C. Revoluția Machine Learning și Deep Learning (Anii 1990 - 2010)

Schimbarea de paradigmă a venit când cercetătorii s-au oprit din a mai încerca să programeze reguli manual. În schimb, au început să creeze algoritmi care **învață regulile singuri din date**.

- **Machine Learning (Învățarea Automată):** Trecerea la abordări statistice. Un moment de cotitură istoric a fost anul 1997, când supercomputerul **Deep Blue** de la IBM l-a învins pe campionul mondial la șah, Garry Kasparov. Trecerea la Machine Learning statistic în anii '90 și dezvoltarea de algoritmi precum SVM (Support Vector Machines) a fost o reacție directă la limitările inerente ale Sistemelor Expert. Algoritmii statistici s-au dovedit superiori în capacitatea de a *generaliza* din date, o abilitate crucială pe măsură ce volumele de date (Big Data) creșteau. Această tranziție a forțat domeniul IA să adopte o bază mai solidă, riguroasă din punct de vedere matematic, pregătind terenul pentru următoarea revoluție tehnologică.
- **Deep Learning (Învățarea Profundă – anii 2010):** Deep Learning (Învățarea Profundă) se referă la utilizarea rețelelor neuronale artificiale cu multiple straturi ascunse ("adânci") Datorită apariției internetului (care a oferit volume uriașe de date / Big Data) și a plăcilor video performante (GPU), rețelele neuronale artificiale cu multe straturi (straturi „profunde”) au devenit brusc extrem de eficiente. În 2016, sistemul **AlphaGo** creat de DeepMind (Google) a învins campionul mondial la Go, un joc considerat mult prea complex pentru calculatoare. Trecerea la Deep Learning a fost posibilă doar datorită convergenței a **trei factori**: **Algoritmi** (rețele neuronale multistratificate), **Date** (Big Data din internet) și **Putere de calcul** (GPU-uri). Fără oricare dintre aceste trei, **IA generativă actuală** nu ar fi fost posibilă.

D. Era Actuală: IA Generativă (2020 - Prezent)

Dacă sistemele anterioare erau în principal *analitice* (clasificau imagini, traduceau text sau preziceau prețuri), noua eră este **generativă**.

- **Arhitectura Transformer (2017):** O inovație majoră de la Google care a permis modelelor să înțeleagă contextul cuvintelor dintr-un text mult mai bine. Modelul Transformer a devenit rapid arhitectura de bază pentru toate Modelele Lingvistice Mari (LLM) moderne, deoarece

permite o paralelizare superioară și reduce semnificativ timpul necesar antrenării modelului, stabilind un nou standard în procesarea limbajului

- **Modelele Lingvistice Mari (LLMs) și Modelele de Difuzie:** Apariția ChatGPT (OpenAI), Gemini (Google), Claude (Anthropic), Midjourney, Qwen (Alibaba) etc. Aceste sisteme sunt capabile să *creeze* conținut nou (text, imagini, cod, muzică) pe baza unor instrucțiuni simple (prompts), revoluționând interacțiunea om-calculator.
- **Provocări și Direcții Curente:** În ciuda puterii lor, modelele LLM se confruntă cu provocări etice și tehnice, inclusiv "halucinațiile" (inventarea de fapte sau referințe fictive). Cercetarea și dezvoltarea actuală se concentrează pe reducerea halucinațiilor, multimodalitate (interacțiune vocală, analiză de imagini și date), integrarea nativă a instrumentelor (plugin-uri, analiză avansată de date) și, cel mai important, pe **Problema Alinierii** (*The alignment problem*). Aceasta vizează asigurarea faptului că sistemele IA de ultimă generație acționează în conformitate cu intențiile și valorile umane.

1.5. Domenii de cercetare în IA

Astăzi, IA este un ecosistem vast. Principalele subdomenii care interacționează în permanență sunt:

- **Machine Learning (Învățarea Automată):** Nucleul matematic al IA actuale. Se împarte în:
  - *Învățare supervizată* (antrenare cu date etichetate pentru a face predicții (clasificare sau regresie), ex. prețuri istorice ale caselor).
  - *Învățare nesupervizată* (descoperirea de tipare ascunse în date neetichetate, ex. segmentarea clienților).
  - *Învățare prin consolidare (recompense) (Reinforcement Learning)* (sistemul învață prin încercare și eroare, optimizând un scor, ex. roboți industriali sau mașini autonome).
  - *Deep Learning (Învățarea Profundă)* **(**Utilizează rețele neuronale cu mai multe straturi (deep neural networks) pentru a modela abstractizări de nivel înalt (ex: recunoașterea imaginilor, LLM-uri).)
- **Procesarea Limbajului Natural (NLP - Natural Language Processing):** Permite mașinilor să citească, să înțeleagă, să traducă și să genereze limbaj uman.
- **Computer Vision (Vederea Artificială):** Oferă mașinilor capacitatea de a „vedea” și interpreta lumea vizuală (imagini, video, camere video stradale, radiografii medicale).
- **Robotică:** Integrarea IA în corpuri fizice pentru automatizarea manipulării de obiecte sau a navigației.
- **Logica și Raționamentul:** Dezvoltă sisteme care pot deduce informații, rezolva probleme și lua decizii bazate pe reguli și reprezentări de cunoștințe (ex: sisteme expert).
- **Ontologii și Semantica:** Creează structuri formale (ontologii) pentru a reprezenta cunoștințele și relațiile din lume, permițând mașinilor să înțeleagă sensul.
- **AI Generativă:** Ramură relativ nouă care se concentrează pe crearea de conținut nou (text, imagini, muzică, cod) pe baza datelor de antrenament (ex: ChatGPT, Gemini, DALL-E).
- **AI Generală (AGI):** Obiectivul pe termen lung de a crea mașini cu inteligență la nivel uman sau supra-uman, capabile să execute orice sarcină intelectuală pe care o poate face un om.

1.6. Aplicații economice ale IA

IA nu mai este un experiment de laborator; ea este un motor economic major. În business, IA aduce valoare prin trei piloni: **reducerea costurilor**, **creșterea eficienței** și **crearea de noi modele de afaceri**.

Tabelul 1.1. **Exemple sectoriale**

Sector Economic	Aplicație Practică a IA	Beneficiu Direct
Finanțe & Bănci	Detectarea fraudelor cu carduri în timp real; Algoritmi de *High-Frequency Trading*; Evaluarea automată a riscului de credit.	Reducerea pierderilor financiare, decizii de creditare în câteva secunde.

Marketing & Retail	Hiper-personalizarea recomandărilor (ex. Netflix, Amazon); Dynamic Pricing (ajustarea prețurilor în timp real în funcție de cerere).	Creșterea vânzărilor și a loialității clienților.
Managementul Lanțului de Aprovizionare	Predicția stocurilor pe baza tendințelor istorice și a vremii; Optimizarea rutelor de livrare.	Reducerea costurilor de depozitare și a amprentei de carbon.
Resurse Umane (HR)	Screening-ul automat al CV-urilor; Analiza sentimentului angajaților prin sondaje interne.	Economie de timp pentru recrutori, identificarea timpurie a burnout-ului.
Producție / Industrie	Mentenanță predictivă (IA analizează senzorii utilajelor și prezice *când* se vor strica înainte ca acest lucru să se întâmple).	Evitarea opririi liniilor de producție, economii masive de reparații.
Creative Industries / Content Creation	Generarea automată de copy, imagini publicitare, asistență în scrierea de cod.	Democratizarea producției de conținut la costuri reduse.

1.7. IA Responsabilă: Provocările Etice

- **Biasul Algoritmologic**: Modelele pot învăța prejudecăți umane din datele istorice (ex: discriminarea la angajări).
- **Halucinațiile**: Modelele generative pot genera informații false care par extrem de convincreate (fenomenul de 'halucinație').
- **Impactul pe Piața Muncii**: Discuția nu trebuie să fie doar despre înlocuire, ci despre augmentare (omul + IA). Strategia viitorului este 'Human-in-the-loop' (omul rămâne în buclă de control)."

Întrebări de autoevaluare

1. Care sunt cele patru perspective principale prin care putem defini inteligența artificială (conform clasificării lui Russell și Norvig)?
2. Care este diferența fundamentală dintre categoriile de IA?
3. Ce măsoară Testul Turing și care este criteriul de succes pentru ca o mașină să fie considerată „inteligentă” în contextul acestui test?
4. Ce critică aduce experimentul mintal „Camera Chineză”?
5. Care au fost cele trei factori convergenți (tehnologici și de date) care au permis explozia Deep Learning ?
6. Care este diferența de abordare dintre IA Simbolică (GOFAI) și Machine Learning?
7. Diferențiați pe scurt cele trei tipuri de învățare în Machine Learning.
8. Ce înseamnă fenomenul de „halucinație” în modelele generative și cum poate apărea „biasul algoritmologic”?

CAPITOLUL 2. AGENȚI INTELIGENȚI ȘI MEDII DE LUCRU

2.1. Conceptul de agent și relația agentului cu mediul

Un **agent** este orice entitate capabilă să *perceapă* mediul înconjurător prin intermediul unor **senzori** și să *acționeze* asupra acestui mediu prin intermediul unor **actuatori** (Russell & Norvig, 2020).

Această definiție este deliberat generală, pentru a acoperi atât agenți software (de exemplu, un program care citește fluxuri de date de piață și emite ordine de cumpărare/vânzare), cât și agenți fizici (un robot, un vehicul autonom).

***Tabel 2.1.** Exemple de perechi senzor–actuator, pe tip de agent*

Tip de agent	Senzori (percepție)	Actuatori (acțiune)
Om	Ochi, urechi, alte organe de simț	Mâini, picioare, aparat vocal
Robot mobil	Cameră video, senzori de proximitate, GPS	Motoare, roți, braț robotic
Agent software de tranzacționare	Fluxuri de date de piață (prețuri, volume)	Ordine de cumpărare/vânzare trimise către bursă
Chatbot / asistent virtual	Text introdus de utilizator, istoricul conversației	Text generat ca răspuns, apeluri către alte servicii (API)
Sistem de recomandare e-commerce	Comportamentul de navigare, istoricul achizițiilor	Lista de produse afișată/recomandată

2.2. Ciclul percepție – decizie – acțiune

Interacțiunea agent–mediu se desfășoară ca un ciclu continuu, ilustrat schematic mai jos:

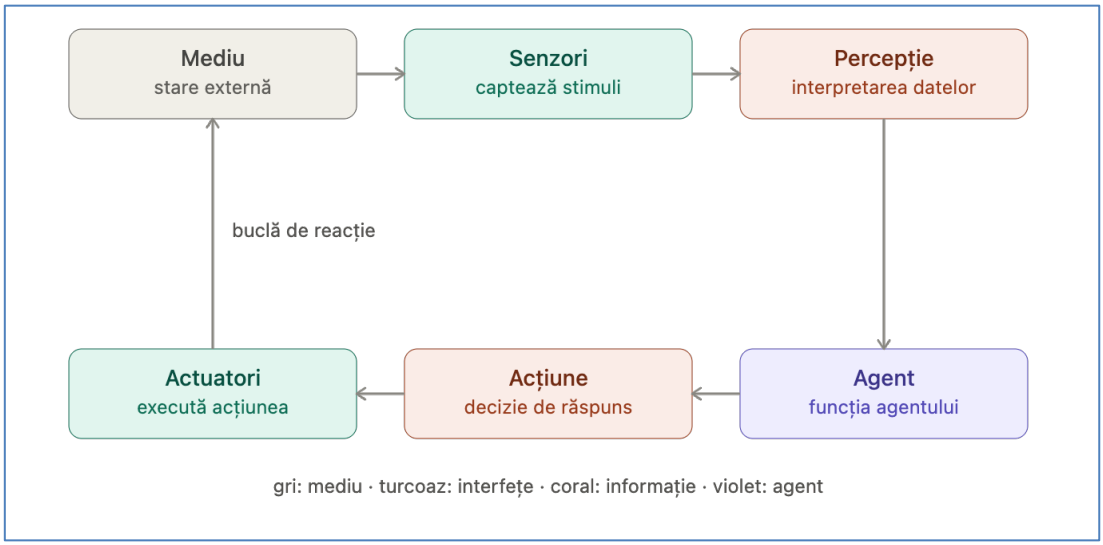


Figura 2.1. Interacțiunea agent–mediu

💡 Agentul primește la fiecare moment de timp o percepție (percept) din mediu; secvența completă a tuturor percepțiilor primite până în acel moment se numește „secvență de percepții" (percept sequence). Decizia agentului cu privire la acțiunea următoare poate depinde, în principiu, de întreaga secvență de percepții, nu doar de percepția curentă.

Din punct de vedere formal, comportamentul unui agent este descris matematic printr-o **funcție agent** (agent function), care asociază fiecărei secvențe posibile de percepții o acțiune. În practică, această funcție este implementată printr-un **program agent** (agent program), care rulează pe o arhitectură fizică sau software concretă.

- Funcția agent = descrierea abstractă, matematică, „ce acțiune corespunde fiecărei percepții".
- Programul agent = implementarea concretă, în cod, a acestei funcții, care rulează pe arhitectura agentului (procesor, senzori, actuatori).

Distincția este importantă: aceeași funcție agent poate fi implementată prin programe complet diferite (un tabel de căutare exhaustiv versus un algoritm compact), la fel cum aceeași funcție matematică poate fi calculată prin algoritmi diferiți.

„Pentru a trece de la o simplă percepție la o înțelegere a mediului, agentul trebuie să poată construi o Stare (State). Starea este o reprezentare internă a mediului care sintetizează secvența de percepții trecute. Exemplu: Un robot aspirator vede un perete (percepție). Starea internă a robotului nu este doar „perete aici", ci „perete în coordonata X,Y în harta locuinței". Această starea permite agentului să „știe" că peretele există chiar și atunci când senzorul nu îl mai „vede" în momentul curent."

2.3. Raționalitatea, autonomia și învățarea agentului

Un **agent rațional** este acel agent care, pentru fiecare secvență posibilă de percepții, alege acțiunea care se așteaptă să maximizeze măsura sa de performanță, dată fiind evidența furnizată de secvența de percepții și de orice cunoștințe încorporate în agent.

Această definiție conține patru elemente esențiale, pe care le analizăm pe rând:

1. *Măsura de performanță* (performance measure) – criteriul obiectiv, extern, prin care se evaluează succesul comportamentului agentului (de exemplu: profitul generat de un agent de tranzacționare, timpul de livrare al unui robot, satisfacția clientului pentru un chatbot).
2. *Cunoștințele* preexistente ale agentului despre mediu.
3. *Acțiunile* pe care agentul le poate întreprinde (repertoriul de acțiuni disponibile prin actuatori).
4. *Secvența de percepții* primită de agent până la momentul curent.

Definiția formală (parafrazată)

Pentru fiecare secvență posibilă de percepții, un agent rațional trebuie să selecteze o acțiune care maximizează performanța așteptată, ținând cont de evidența furnizată de percepții și de cunoștințele deja deținute de agent.

O confuzie frecventă este identificarea raționalității agentului cu **atotștiința (omniscience)** – cunoașterea perfectă și completă a rezultatului real al fiecărei acțiuni. Un agent *atotștiutor* ar cunoaște dinainte, cu certitudine, consecințele exacte ale oricărei acțiuni. Un asemenea agent este, în practică, imposibil de construit – și, de fapt, inutil ca model teoretic pentru IA reală.

- Un agent rațional acționează pe baza informației disponibile la momentul deciziei, nu pe baza informației care va deveni disponibilă ulterior.
- Din acest motiv, o acțiune rațională poate conduce, în mod complet justificat, la un rezultat nefavorabil – fără ca aceasta să însemne că agentul a fost irațional.

Exemplu clasic (traversarea străzii)

Un pieton verifică ambele sensuri de circulație, nu vede nicio mașină apropiată și traversează strada. În acel moment, decizia este perfect rațională. Dacă, în mod cu totul neprevăzut, o piesă dintr-un avion cade exact în acel punct și rănește pietonul, decizia sa de a traversa nu devine, retroactiv, irațională – pur și simplu, informația despre acel eveniment nu era disponibilă și nu putea fi anticipată în mod rezonabil.

Exemplu economic – agent de tranzacționare algoritmică

Un algoritm de tranzacționare cumpără acțiuni ale unei companii pe baza tuturor indicatorilor financiari disponibili la momentul respectiv (rezultate financiare, tendințe de piață, sentiment din presă). Dacă imediat după tranzacție compania anunță neprevăzut un scandal intern necunoscut public, iar acțiunile scad brusc, decizia algoritmului rămâne rațională – ea a fost optimă în raport cu informația disponibilă la momentul deciziei.

Un agent rațional bine proiectat nu se bazează exclusiv pe cunoștințele „înnăscute" (programate inițial), ci este capabil, ideal, de două lucruri suplimentare:

- **colectarea de informații suplimentare** prin explorare, atunci când informația disponibilă este insuficientă pentru o decizie bună (de exemplu, un robot care explorează un mediu necunoscut înainte de a executa o sarcină);
- **învățarea din experiență** – capacitatea de a-și ajusta comportamentul viitor pe baza rezultatelor acțiunilor trecute, ceea ce definește gradul de *autonomie* a unui agent.

Un agent complet lipsit de autonomie acționează exclusiv conform cunoștințelor programate inițial de proiectant; un agent autonom își modifică și își îmbunătățește comportamentul pe măsură ce acumulează experiență – proprietate centrală pentru toate sistemele de Machine Learning discutate în temele următoare.

2.4. Cadrul PEAS – metodă de proiectare a specificațiilor unui agent

Înainte de a proiecta orice sistem inteligent, este necesar să răspundem, în mod explicit și riguros, la patru întrebări. Acronimul PEAS structurează exact aceste patru întrebări:

Tabel 2.2. Componenta PEAS

Componenta PEAS	Întrebarea la care răspunde
P – Performance measure (Măsura de performanță)	Care este criteriul obiectiv prin care evaluăm dacă agentul are succes?
E – Environment (Mediul)	În ce mediu operează agentul? Ce elemente conține acesta?
A – Actuators (Actuatori)	Prin ce mijloace acționează agentul asupra mediului?
S – Sensors (Senzori)	Prin ce mijloace percepe agentul mediul?

Cadrul PEAS este primul pas obligatoriu în orice proiect de proiectare a unui agent inteligent – la fel cum specificarea cerințelor este primul pas într-un proiect software clasic.

Tabel 2.3. Exemplu detaliat aplicat — Taxi autonom

Componentă	Specificație
Performanță	Siguranța pasagerilor și a pietonilor; respectarea legislației rutiere; timp de deplasare minim; confortul călătoriei; profitul generat; minimizarea consumului de combustibil/energie.
Mediu	Străzi, alte vehicule, pietoni, semnale de circulație, condiții meteorologice, trasee posibile, pasageri.
Actuatori	Volan, accelerație, frână, semnalizare, claxon, ecran/afișaj pentru comunicare cu pasagerul.
Senzori	Camere video, radar/LIDAR, GPS, senzori de viteză, senzori de proximitate, microfon.

Tabel 2.4. Exemplu aplicat domeniului economic — Agent de recomandare pentru un magazin online

Componentă	Specificație
Performanță	Rata de conversie (cumpărare efectivă), valoarea coșului de cumpărături, satisfacția și fidelizarea clientului, relevanța recomandărilor.
Mediu	Catalogul de produse, comportamentul de navigare al utilizatorilor, istoricul comenzilor, stocul disponibil, sezonalitatea.
Actuatori	Afișarea listei de produse recomandate, trimiterea de notificări/e-mailuri personalizate, ajustarea ofertelor promoționale.
Senzori	Click-uri, timp petrecut pe pagină, istoric de căutări și achiziții, coșul de cumpărături curent.

2.5. Proprietățile mediilor de lucru

Alegerea arhitecturii unui agent depinde în mod esențial de tipul de mediu în care acesta operează. Literatura de specialitate (Russell & Norvig) definește **șase dimensiuni principale** după care un mediu poate fi caracterizat; le prezentăm mai jos, împreună cu exemple.

Tabel 2.5 Dimensiunile mediului

Dimensiune	Descriere	Exemplu
Accesibil (observabil) vs. inaccesibil (parțial observabil)	Un mediu este accesibil dacă senzorii agentului au acces la starea completă și relevantă a mediului la fiecare moment; este inaccesibil (parțial observabil) dacă agentul dispune doar de informații incomplete sau zgomotoase.	Șah (accesibil) vs. conducere auto în trafic real (parțial observabil)
Determinist vs. stohastic	Un mediu este determinist dacă starea următoare este complet determinată de starea curentă și de acțiunea agentului; este stohastic dacă intervine incertitudine sau elemente aleatorii.	Puzzle logic (determinist) vs. piața bursieră (stohastică)
Episodic vs. secvențial	Într-un mediu episodic, experiența agentului este împărțită în episoade	Clasificarea automată a imaginilor (episodic) vs. jocul

Dimensiune	Descriere	Exemplu
	independente, în care acțiunea curentă nu afectează episoadele viitoare. Într-un mediu secvențial, deciziile curente influențează deciziile și rezultatele viitoare.	de șah sau planificarea unei strategii de investiții (secvențial)
Static vs. dinamic	Un mediu static rămâne neschimbat în timp ce agentul deliberează; un mediu dinamic se poate schimba în timp ce agentul decide, ceea ce impune constrângeri de timp real.	Rezolvarea unui puzzle offline (static) vs. conducerea unui vehicul autonom (dinamic)
Discret vs. continuu	Distincția se aplică stării mediului, timpului, percepțiilor și acțiunilor: un mediu discret are un număr finit (sau numărabil) de stări/acțiuni distincte, în timp ce un mediu continuu variază pe un spectru continuu de valori.	Jocul de șah (discret) vs. conducerea unui vehicul, unde viteza și direcția variază continuu
Agent unic vs. Multiagent	Un mediu este cu agent unic dacă un singur agent acționează în el, fără a interacționa strategic cu alte entități inteligente. Este multiagent dacă mai mulți agenți operează simultan în același mediu, iar performanța fiecăruia depinde și de acțiunile celorlalți — situație care poate fi cooperantă (agenții urmăresc un obiectiv comun) sau competitivă (agenții au obiective opuse).	Rezolvarea unui puzzle de către un singur agent (agent unic) vs. jocul de șah împotriva unui adversar sau licitațiile automate pe piețele financiare (multiagent), unde mai mulți agenți (roboți de trading) concurează pentru aceleași oportunități de piață, este un caz clasic de mediu multiagent competitiv.

- Cu cât un mediu este mai puțin accesibil, mai stohastic, mai dinamic și mai continuu, cu atât proiectarea agentului devine mai complexă, fiind necesare mecanisme de gestionare a incertitudinii (probabilitate, logică fuzzy, învățare din experiență).
- Cel mai dificil caz general – cazul „lumii reale" – este, de regulă, parțial observabil, stohastic, secvențial, dinamic și continuu; exact acest tip de mediu justifică nevoia paradigmelor moderne de Machine Learning și Deep Learning, care sunt capabile să gestioneze incertitudinea și complexitatea, spre deosebire de sistemele bazate exclusiv pe reguli fixe.

Proiectarea unui agent implică gestionarea a două tipuri de incertitudine:

- **Incertitudinea mediului** (Stohasticity): Acțiunea agentului poate avea rezultate imprevizibile (ex: vânt executând o acțiune de zbor).
- **Incertitudinea senzorială** (Observability): Senzorii agentului pot returna date eronate sau incomplete (ex: o cameră video în ceață). Un agent robust trebuie să poată procesa ambele tipuri de incertitudine prin metode probabiliste.

2.6. Tipuri de agenți inteligenți

Arhitectura internă a unui agent poate fi organizată în patru categorii fundamentale, de complexitate progresivă. Fiecare tip nou încorporează și extinde principiile tipului anterior.

Agentul cu reflexe simple (simple reflex agent)

Acest tip de agent alege acțiunea exclusiv pe baza percepției curente, ignorând complet istoricul percepțiilor anterioare. Comportamentul este descris prin reguli condiție-acțiune de tipul „DACĂ percepție = X, ATUNCI acțiune = Y".

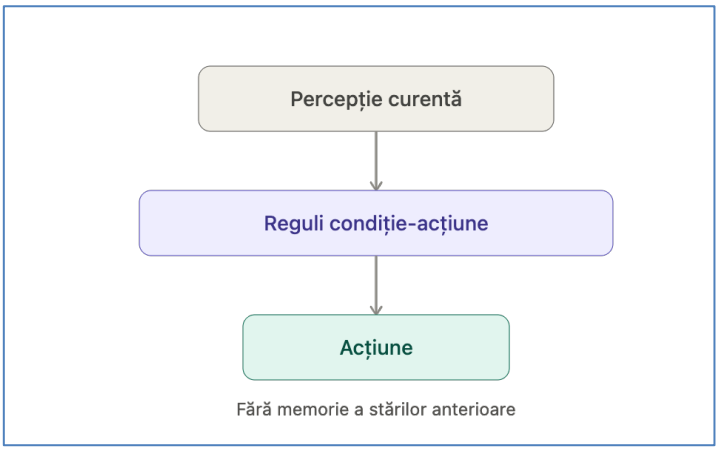


Figura 2.2 Agent cu reflexe simple

- Exemplu: un termostat care pornește încălzirea dacă temperatura curentă este sub un prag, independent de orice informație istorică.
- Avantaj: simplitate, viteză de reacție, cost redus de implementare.
- Limitare: funcționează corect doar în medii complet accesibile; eșuează în medii parțial observabile, unde aceeași percepție poate corespunde unor situații reale diferite.

Agentul bazat pe model (model-based reflex agent)

Pentru a funcționa în medii parțial observabile, agentul menține o stare internă (un „model intern al lumii"), care surprinde aspectele relevante ale mediului neobservabile direct la momentul curent. Această stare internă este actualizată permanent pe baza istoricului de percepții și a cunoștințelor despre modul în care mediul evoluează.

- Exemplu: un robot aspirator care își reține harta internă a camerelor deja curățate, chiar dacă senzorii nu „văd" în acel moment toată încăperea.
- Exemplu economic: un sistem de gestiune a stocurilor care menține o estimare internă a nivelului real de stoc, chiar și în intervalele în care nu primește actualizări directe de la depozit.

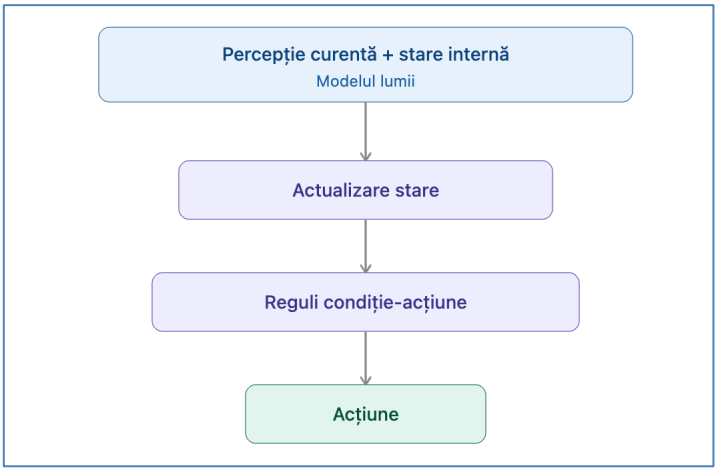


Figura 2.3 Agent bazat pe model

Agentul bazat pe scopuri (goal-based agent)

Pe lângă starea curentă a mediului, agentul deține informații despre situațiile dezirabile – scopurile de atins. Alegerea acțiunii implică, implicit sau explicit, un proces de căutare sau planificare: agentul evaluează secvențe posibile de acțiuni pentru a determina care dintre ele conduce la atingerea scopului.

- Diferența esențială față de agentul bazat pe model: acesta din urmă răspunde la întrebarea „ce se întâmplă dacă fac acțiunea X?", în timp ce agentul bazat pe scopuri răspunde suplimentar la întrebarea „mă ajută acțiunea X să ating scopul dorit?".

- Exemplu: un sistem de planificare a rutelor logistice, care caută traseul optim dintre un depozit și un set de puncte de livrare, având scopul explicit „minimizarea distanței totale parcurse”.
- Avantaj: flexibilitate – schimbarea scopului este suficientă pentru a modifica radical comportamentul agentului, fără a rescrie regulile de bază.

Agentul bazat pe utilitate (utility-based agent)

Simpla atingere a unui scop nu este întotdeauna suficientă: de multe ori există mai multe moduri de a atinge același scop, iar unele sunt clar preferabile altora (mai rapide, mai sigure, mai ieftine). Agentul bazat pe utilitate introduce o funcție de utilitate, care asociază fiecărei stări a mediului un grad numeric de „dezirabilitate”, permițând o comparație riguroasă între alternative — inclusiv în situații care implică scopuri conflictuale sau incertitudine.

- Exemplu: un agent de tranzacționare algoritmică nu urmărește doar „scopul” de a obține profit, ci maximizează o funcție de utilitate care combină profitul așteptat cu nivelul de risc asumat.
- Exemplu: un asistent de planificare a călătoriilor care compară mai multe rute posibile, ținând cont simultan de cost, durată și număr de escale, alegând-o pe cea cu utilitatea globală maximă.

Dacă un agent bazat pe scopuri este un agent care caută un „DA” (am ajuns la destinație), un agent bazat pe utilitate este un agent care caută „CÂT DE BUNE” (am ajuns la destinație prin cel mai scurt drum, cu cel mai mic consum și în condiții de siguranță maxime). Funcția de utilitate permite rezolvarea compromisurilor (trade-offs): Siguranță vs. Viteză.

Tabel 2.6 Sinteză comparativă a tipurilor de agenți

Tip de agent	Memorează starea internă?	Are un scop explicit?	Compară alternative după utilitate?	Complexitate
Reflex simplu	Nu	Nu	Nu	Redusă
Bazat pe model	Da	Nu	Nu	Medie
Bazat pe scopuri	Da	Da	Nu	Medie-ridicată
Bazat pe utilitate	Da	Da	Da	Ridicată

Toate cele patru arhitecturi pot fi, la rândul lor, transformate în **agenți capabili de învățare** (learning agents), prin adăugarea unei componente care evaluează performanța trecută și ajustează regulile, modelul intern, scopurile sau funcția de utilitate ale agentului – principiu care va fi reluat și dezvoltat în temele dedicate Machine Learning.

Întrebări de autoevaluare

1. Explicați, cu un exemplu propriu, diferența dintre o funcție agent și un program agent.
2. De ce un agent care acționează rațional poate obține, uneori, un rezultat nefavorabil? Contrazice acest fapt definiția raționalității?
3. Construiți specificația PEAS pentru un agent care gestionează automat portofoliul de investiții al unui client.
4. Clasificați, după cele șase dimensiuni discutate, mediul de lucru al unui asistent virtual de tip chatbot pentru suport clienți.
5. Care este diferența esențială dintre un agent bazat pe scopuri și un agent bazat pe utilitate? Dați un exemplu în care cele două ar recomanda acțiuni diferite.

CAPITOLUL 3. REZOLVAREA PROBLEMELOR PRIN CĂUTARE NEINFORMATĂ.

3.1. Formularea problemelor ca spații de stări

O parte semnificativă a problemelor rezolvate de un agent inteligent – de la un joc de puzzle, la planificarea unei rute de livrare, la găsirea unei secvențe de mutări într-un joc de strategie – pot fi reduse la un singur tipar comun: găsirea unei secvențe de acțiuni care conduce agentul dintr-o stare inițială la o stare-scop, eventual minimizând un cost asociat.

Acest tipar se numește **problemă de căutare (search problem)**, iar disciplina care studiază algoritmii de rezolvare a acestor probleme se numește **căutare în spațiul stărilor (state-space search)**. Un agent bazat pe scopuri (goal-based agent) folosește exact acest mecanism pentru a decide ce acțiune să întreprindă.

Pentru a putea aplica un algoritm de căutare, orice problemă trebuie formalizată riguros prin cinci componente:

Tabel 3.1 Formalizarea problemei de căutare

Componentă	Descriere
Starea inițială (initial state)	Starea din care pornește agentul, descriind complet configurația relevantă a lumii la momentul inițial.
Acțiuni (actions)	Pentru o stare dată s, mulțimea ACTIONS(s) a tuturor acțiunilor posibile pe care agentul le poate întreprinde din acea stare.
Modelul de tranziție (transition model)	Funcția RESULT(s, a) care descrie starea rezultată din aplicarea acțiunii a asupra stării s. Împreună, starea inițială, acțiunile și modelul de tranziție definesc **spațiul de stări** (state space) – graful format din toate stările accesibile din starea inițială.
Testul de scop (goal test)	Funcția care determină dacă o stare dată este o stare-scop (finală, dorită).
Costul căii (path cost)	Funcție numerică ce asociază fiecărei căi (secvențe de stări legate prin acțiuni) un cost, de obicei suma costurilor individuale ale acțiunilor (step cost).

O **soluție** a problemei este o secvență de acțiuni care conduce de la starea inițială la o stare-scop; o soluție optimă este soluția cu cel mai mic cost al căii dintre toate soluțiile posibile.

Tabel 3.2 Exemplu de formalizare (Problema celor 8 puzzle-uri)

Componentă	Formalizare pentru 8-puzzle
Stare inițială	O configurație dată a celor 8 piese numerotate și a spațiului liber pe o grilă 3×3.
Acțiuni	Deplasarea spațiului liber: Sus, Jos, Stânga, Dreapta (unele mutări sunt indisponibile la marginile grilei).
Model de tranziție	Aplicarea unei mutări schimbă poziția spațiului liber cu piesa adiacentă corespunzătoare direcției alese.
Test de scop	Configurația curentă coincide cu configurația-scop stabilită (de exemplu, piesele ordonate 1–8, cu spațiul liber în colțul din dreapta jos).
Cost al căii	Fiecare mutare are cost 1; costul total = numărul de mutări efectuate.

> 💡 **Observație importantă**
>
> Spațiul de stări al problemei 8-puzzle conține 9!/2 = 181 440 stări accesibile. Chiar și pentru o problemă aparent simplă, spațiul de căutare poate fi extrem de mare – motiv pentru care alegerea algoritmului de căutare influențează decisiv fezabilitatea practică a soluției.

3.2. Grafuri și arbori de căutare

Spațiul de stări al unei probleme poate fi reprezentat ca un graf, în care nodurile sunt stările, iar arcele (muchiile orientate) sunt acțiunile care leagă o stare de rezultatul ei. Procesul de căutare explorează acest graf construind, incremental, un **arbore de căutare** (search tree), în care rădăcina corespunde stării inițiale, iar fiecare nod copil corespunde unei stări obținute prin aplicarea unei acțiuni.

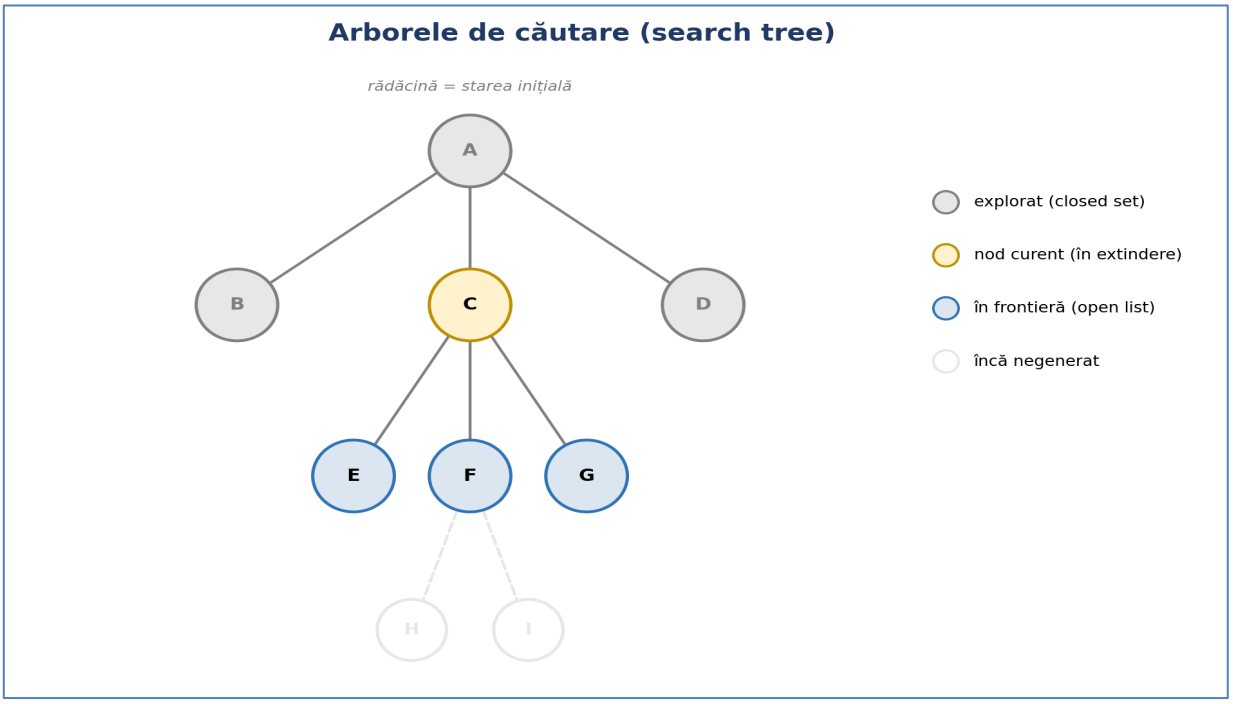


Figura 3.1. Arborele de căutare: rădăcina reprezintă starea inițială, iar frontiera (open list) conține nodurile generate, dar încă neexplorate.

Structura generală a unui algoritm de căutare

- **Frontiera (frontier / open list)** – mulțimea nodurilor generate, dar încă neexplorate; algoritmul alege la fiecare pas un nod din frontieră pentru extindere.
- **Mulțimea explorată (explored set / closed list)** – nodurile deja extinse, reținute pentru a evita reexplorarea acelorași stări (esențial în grafuri cu cicluri).
- **Strategia de căutare** – regula după care se alege nodul următor din frontieră; exact această strategie diferențiază BFS, DFS, UCS și, ulterior, algoritmii informați.

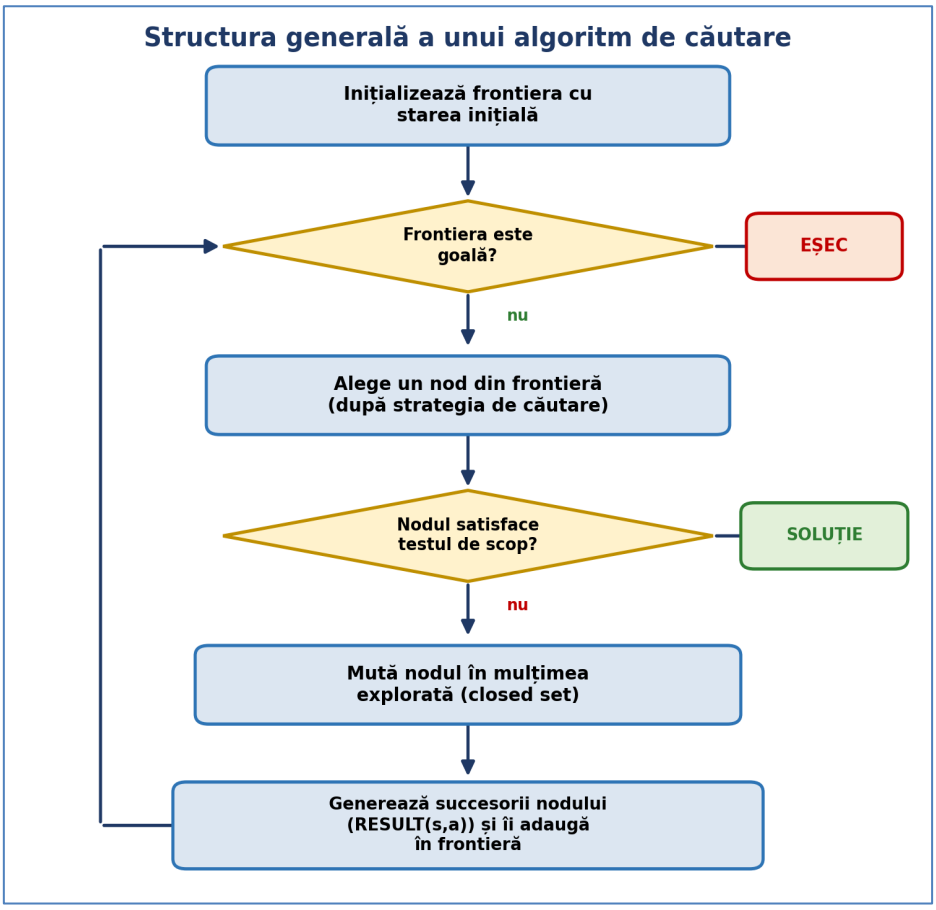


Figura 3.2. Structura generală a unui algoritm de căutare – ciclul frontieră → selecție → test de scop → extindere.

💡 **Criteriile standard de evaluare a unui algoritm de căutare**

Completitudine: algoritmul garantează găsirea unei soluții, dacă aceasta există?

Optimalitate: algoritmul garantează găsirea soluției cu cel mai mic cost?

Complexitate temporală: câți noduri sunt generați/extinși în cel mai defavorabil caz?

Complexitate spațială: câtă memorie este necesară în cel mai defavorabil caz?

Notații standard: b = factorul de ramificare (branching factor), d = adâncimea celei mai puțin costisitoare soluții, m = adâncimea maximă a spațiului de stări.

3.3. Algoritmul BFS (Breadth-First Search)

Algoritmul BFS(Căutare în lățime) extinde întotdeauna nodul cel mai puțin adânc rămas neexplorat din frontieră – cu alte cuvinte, explorează spațiul de stări „pe niveluri”, nivel cu nivel, folosind o structură de tip coadă FIFO (first-in-first-out) pentru frontieră. Nodul A este extras primul și expandat, adăugând B și C în coadă; apoi B este extras și expandat, adăugând D și E; apoi C, adăugând F și G – de aici ordinea de vizitare 1→7 din diagramă.

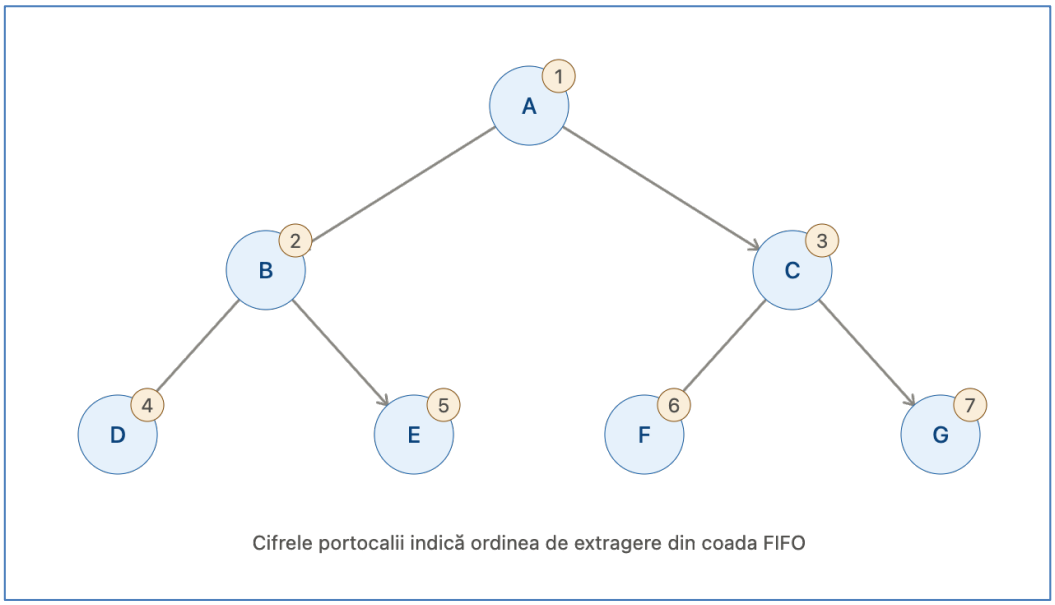


Figura 3.3. Structura generală a algoritmului BFS

- **Funcționare:** toate nodurile de la adâncimea d sunt extinse înaintea oricărui nod de la adâncimea d+1.
- **Completitudine:** DA – dacă factorul de ramificare b este finit, BFS găsește întotdeauna o soluție, dacă aceasta există.
- **Optimalitate:** DA, dar numai atunci când toate acțiunile au același cost; dacă acțiunile au costuri diferite, BFS nu garantează soluția optimă (pentru acest caz se folosește UCS).
- **Complexitate temporală:** O(b^d) – crește exponențial cu adâncimea soluției.
- **Complexitate spațială:** O(b^d) – BFS reține în memorie toate nodurile frontierei, ceea ce reprezintă principala sa limitare practică.
- **Proprietatea cheie**: BFS garantează găsirea soluției cu cel mai mic număr de pași (cost minim în număr de muchii), dar poate consuma multă memorie, deoarece păstrează toate nodurile de la un nivel în coadă simultan – complexitate spațială O(b^d), unde b este factorul de ramificare și d adâncimea soluției.

> 💡 **Observație practică**
>
> Complexitatea spațială este, în general, principala problemă a BFS: chiar și pentru b = 10 și d = 12, algoritmul ar necesita explorarea a peste un trilion de noduri păstrate simultan în memorie – imposibil de gestionat practic pe hardware obișnuit.

3.4. Algoritmul Depth-First Search

Allgoritmul DFS (căutare în adâncime) extinde întotdeauna cel mai adânc nod rămas neexplorat din frontieră, folosind o structură de tip stivă LIFO (last-in-first-out). Algoritmul „coboară” pe o ramură a arborelui de căutare până la adâncime maximă (sau până la un nod fără succesori), apoi revine (backtracking) pentru a explora ramurile rămase.

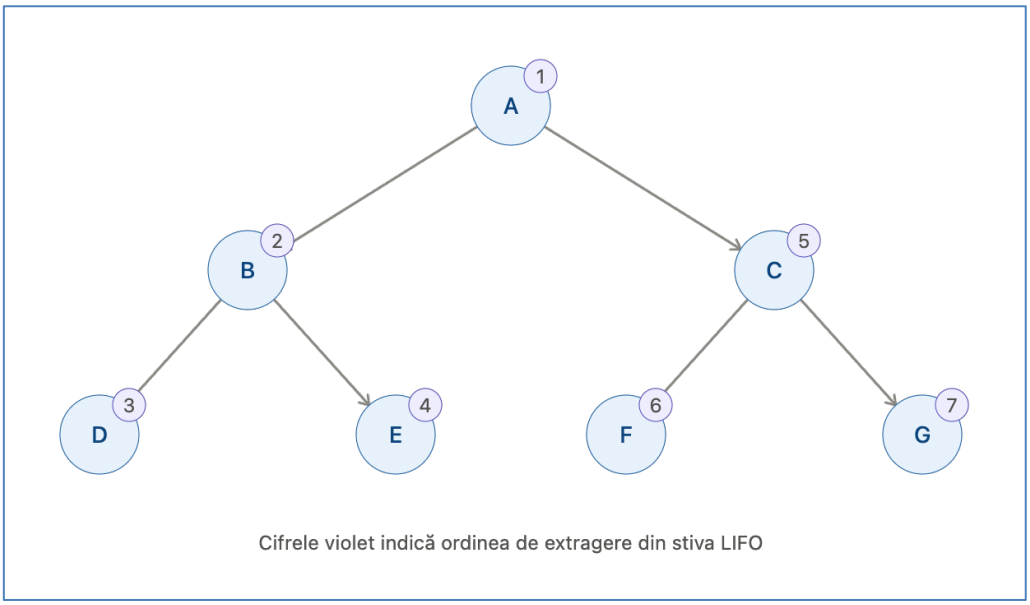


Figura 3.4. Structura generală a algoritmului DFS

- **Completitudine:** NU, în general – dacă spațiul de stări este infinit sau conține cicluri neverificate, DFS se poate „pierde" pe o ramură infinită, fără a găsi niciodată soluția, chiar dacă aceasta există. Este complet doar pe spații de stări finite.
- **Optimalitate:** NU – prima soluție găsită nu este, în general, cea mai puțin costisitoare.
- **Complexitate temporală:** O(b^m), unde m este adâncimea maximă a spațiului de stări (poate fi mult mai mare decât d).
- **Complexitate spațială:** O(b*m) – avantajul esențial al DFS față de BFS: este nevoie de memorie doar pentru o singură cale de la rădăcină la nodul curent, plus nodurile frați neexplorați.

3.5. Iterative Deepening Depth-First Search (IDDFS)

Algoritmul Iterative Deepening DFS (Căutare în adâncime cu adâncire iterativă) combină avantajele BFS (completitudine, optimalitate pentru costuri egale) cu avantajul de memorie al DFS. Algoritmul execută succesiv căutări DFS cu o limită de adâncime crescătoare: mai întâi limita 0, apoi 1, apoi 2, ș.a.m.d., până la găsirea soluției.

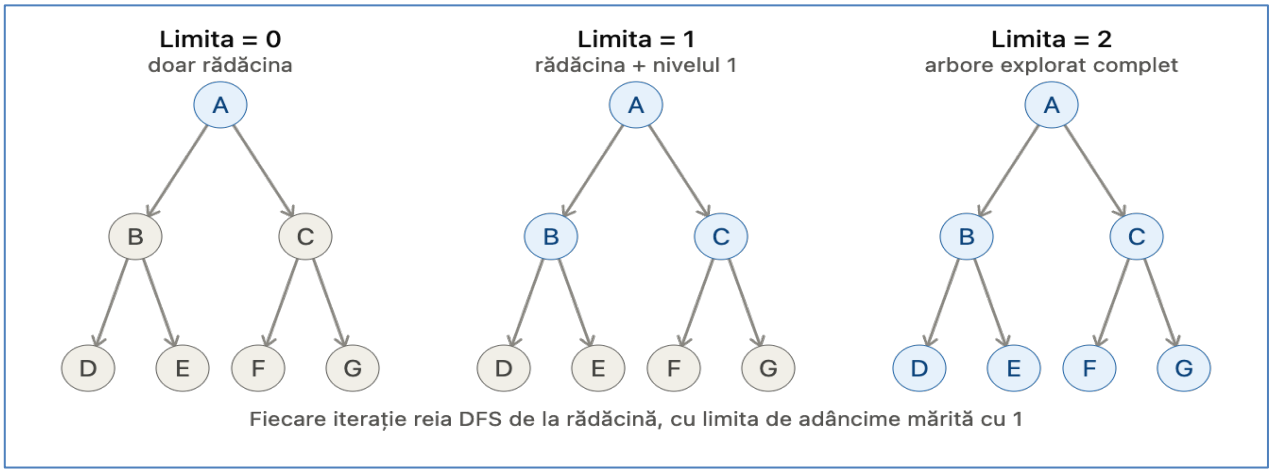


Figura 3.5. Structura generală a algoritmului IDDFS

- **Completitudine:** DA (pe spații finite sau cu adâncime a soluției finită).
- **Costul IDDFS:** nodurile de la nivelurile superioare (A, apoi B, C) sunt regenerate la fiecare iterație – o risipă de timp, dar acceptabilă, pentru că numărul de noduri crește exponențial cu adâncimea, deci costul iterațiilor repetate rămâne dominat de ultima iterație
- **Optimalitate:** DA, pentru costuri egale ale acțiunilor.
- **Complexitate temporală:** O(b^d) – deși stările sunt regenerate la fiecare iterație, costul suplimentar este redus, deoarece majoritatea nodurilor se află la adâncimile cele mai mari.
- **Complexitate spațială:** O(b*d) – la fel de eficientă ca DFS.

💡 **De ce este Iterative Deepening strategia preferată în practică, atunci când spațiul de stări este necunoscut sau foarte mare?**

IDDFS oferă complexitatea temporală apropiată de BFS, dar cu necesarul de memorie al DFS – un compromis considerat, în literatura de specialitate, strategia neinformată preferată atunci când spațiul de căutare este mare și adâncimea soluției nu este cunoscută dinainte.

3.6. Algoritmul UCS (Uniform Cost Search)

Atunci când acțiunile au costuri diferite, BFS nu mai garantează soluția optimă. Algoritmul Uniform Cost Search rezolvă această limitare, extinzând întotdeauna nodul cu cel mai mic **cost al căii** g(n) de la rădăcină până la acel nod (nu cel mai puțin adânc nod, ca în BFS). Frontiera este organizată ca o coadă cu prioritate, ordonată crescător după g(n).

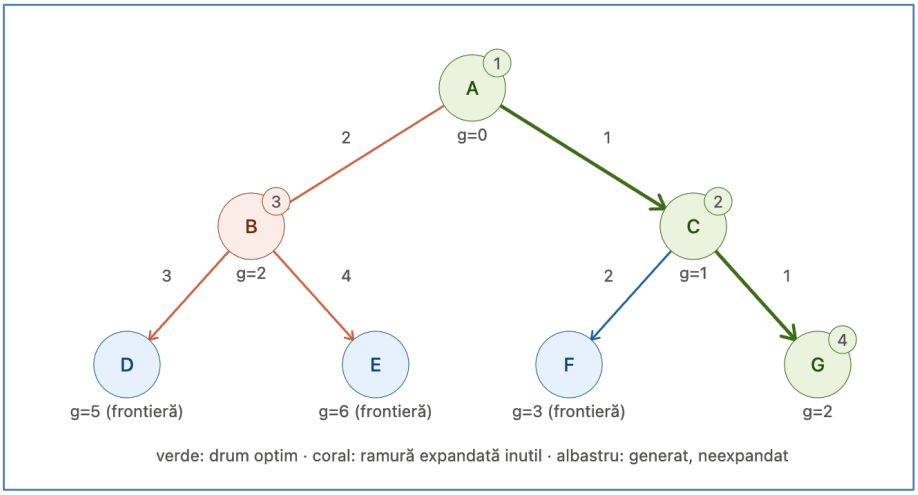


Figura 3.6 Structura generală a algoritmului UCS(Ordinea de expandare UCS pe arbore: A (g=0) → C (g=1) → B (g=2) → G (g=2), moment n care scopul este extras din coada de priorități și căutarea se oprește, cu costul optim confirmat: 2)

Observații: UCS expandează B (g=2) doar pentru că are un cost cumulat mic – nu știe că B duce spre noduri îndepărtate de scop (D, E), pentru că **nu are acces la nicio informație euristică.**

- **Funcționare:** echivalent cu algoritmul lui Dijkstra pentru cel mai scurt drum într-un graf ponderat cu costuri nenegative.
- **Completitudine:** DA, dacă toate costurile acțiunilor sunt strict pozitive (mai mari decât un $\varepsilon > 0$).
- **Optimalitate:** DA – UCS garantează întotdeauna soluția cu cel mai mic cost total, indiferent de costurile individuale ale acțiunilor.
- **Complexitate temporală și spațială:** O(b^(1+⌊C*/ε⌋)), unde C* este costul soluției optime – poate fi semnificativ mai mare decât O(b^d) atunci când există multe acțiuni cu cost redus.

Observăm că BFS este, de fapt, un caz particular al UCS, obținut atunci când toate costurile acțiunilor sunt egale.

3.7. Exemple practice și sinteză comparativă

- Orice problemă de căutare se formalizează prin cinci componente: stare inițială, acțiuni, model de tranziție, test de scop, cost al căii.
- Un algoritm de căutare gestionează o frontieră de noduri neexplorate; strategia de alegere a nodului următor definește algoritmul concret.
- BFS este complet și optim pentru costuri egale, dar are complexitate spațială exponențială.
- DFS folosește puțină memorie, dar nu este, în general, complet și nu este optim.
- Iterative Deepening combină avantajele BFS și DFS și este strategia neinformată preferată atunci când spațiul de căutare este mare.
- UCS garantează întotdeauna soluția optimă, indiferent de costurile acțiunilor, extinzând nodul cu cel mai mic cost al căii.

Tabel 3.3 Compararea a algoritmilor de căutare neinformată

Algoritm	Completitudine	Optimalitate	Timp	Spațiu
BFS	Da (b finit)	Da (costuri egale)	O(b^d)	O(b^d)
DFS	Nu (spații infinite)	Nu	O(b^m)	O(b*m)
IDDFS	Da	Da (costuri egale)	O(b^d)	O(b*d)
UCS	Da (costuri >0)	Da (întotdeauna)	O(b^(1+⌊C*/ε⌋))	O(b^(1+⌊C*/ε⌋))

Rezolvarea unui labirint

Un labirint reprezentat ca o grilă 2D este un caz clasic de aplicare a căutării neinformate: starea = poziția curentă (linie, coloană); acțiunile = Sus/Jos/Stânga/Dreapta, condiționate de absența unui perete; testul de scop = atingerea celulei-ieșire. Dacă fiecare mutare are același cost, BFS garantează calea cea mai scurtă (numărul minim de pași); DFS găsește rapid o cale, dar nu neapărat cea mai scurtă; UCS devine necesar dacă unele celule au un „cost de traversare” diferit (de exemplu, teren mlăștinos vs. drum liber).

Problema celor 8 puzzle-uri

Aplicând BFS pe 8-puzzle se garantează soluția optimă (numărul minim de mutări), dar cu un cost de memorie ridicat, dat fiind spațiul de stări de peste 181 000 configurații. DFS poate „rătăci” în ramuri foarte adânci fără a găsi rapid o soluție rezonabilă. Iterative Deepening reprezintă, pentru această problemă, un echilibru practic rezonabil – motiv pentru care este frecvent folosit ca punct de plecare, înainte de introducerea euristicilor în tema următoare.

Întrebări de autoevaluare

1. Care sunt cele 5 componente fundamentale necesare pentru a transforma o problemă din lumea reală într-o problemă de căutare în spațiul stărilor?
2. Ce reprezintă „frontiera” (frontier) și „mulțimea explorată” (explored set) în procesul de execuție al unui algoritm de căutare?
3. De ce complexitatea spațială este principalul dezavantaj practic al algoritmului BFS (Breadth-First Search)?
4. În ce condiție specifică algoritmul BFS garantează găsirea soluției optime?

De ce algoritmul DFS (Depth-First Search) nu este considerat complet în spații de stări infinite sau stări care conțin cicluri?

5. De ce algoritmul Iterative Deepening este considerat un compromis ideal și strategia preferată atunci, când spațiul de căutare este mare și adâncimea soluției este necunoscută?
6. Care este diferența fundamentală în strategia de selecție a nodului din frontieră între algoritmul UCS și algoritmul BFS?

CAPITOLUL 4. CĂUTAREA INFORMATĂ ȘI CĂUTAREA ADVERSARIALĂ

4.1. Funcția euristică h(n)

În capitolul 3 s-a menționat că algoritmii neinformați (BFS, DFS, IDDFS, UCS) explorează spațiul de stări „orbește”, fără nicio informație despre cât de aproape se află un nod de scop. Această orbire are un preț: complexitate exponențială, chiar și pentru probleme moderate (8-puzzle, labirinturi mari).

Căutarea informată (heuristic search) rezolvă exact această problemă: folosește cunoștințe specifice domeniului problemei pentru a estima, pentru fiecare nod, cât de „promițător” este acesta în drumul spre scop – reducând drastic numărul de noduri explorate.

O **funcție euristică h(n)** estimează costul celei mai ieftine căi de la nodul n până la o stare-scop. Spre deosebire de g(n) (costul real, cunoscut, de la rădăcină până la n), h(n) este o estimare — construită pe baza cunoștințelor despre structura problemei, nu prin explorare efectivă.

Exemple de funcții euristice folosite frecvent:

- **8-puzzle:** numărul de piese aflate pe o poziție greșită (Hamming distance) sau, mai precis, suma distanțelor Manhattan dintre poziția curentă și poziția-scop a fiecărei piese.
- **Navigare rutieră:** distanța în linie dreaptă (euclidiană) dintre poziția curentă și destinație.
- **Deplasare pe o grilă (fără obstacole diagonale):** distanța Manhattan, $|x_1-x_2| + |y_1-y_2|$.

Rolul intuiției umane este esențial în alegerea unei euristici bune: o euristică informativă „ghidează” rapid căutarea spre scop, în timp ce o euristică slabă (de exemplu h(n) = 0 pentru orice n) reduce algoritmul informat la algoritmul UCS.

> 💡 **Observație**
>
> O funcție euristică nu este niciodată calculată prin „a rezolva problema în avans” – ea trebuie să fie ușor de calculat, altfel costul de calcul al euristicii ar anula avantajul câștigat prin reducerea numărului de noduri explorate.

4.2. Algoritmul Greedy Best-First Search

Algoritmul Greedy Best-First Search extinde întotdeauna nodul considerat cel mai apropiat de scop, evaluat exclusiv pe baza celei mai mici euristice: f(n) = h(n). Costul real g(n) parcurs până la nodul n este complet ignorat în decizia de extindere.

- **Avantaj:** poate fi foarte rapid în practică, deoarece „țintește” direct spre scop, dar poate fi ușor de indus în eroare de o euristică imprecisă:
- **Completitudine:** NU, în general – algoritmul se poate bloca pe cicluri sau ramuri fără ieșire, dacă euristica induce în eroare (este posibil pe spații infinite sau cu capcane locale).

Optimalitate: NU – deoarece ignoră complet costul real parcurs, poate alege o cale mult mai lungă doar pentru că pare, local, mai apropiată de scop.

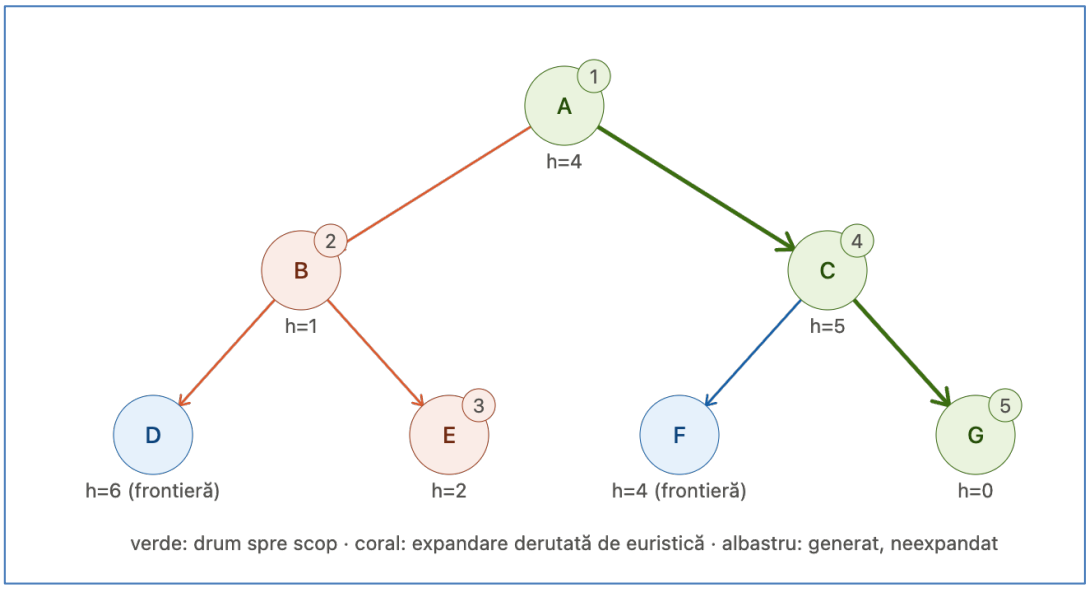


Figura 4.1 Structura generală a algoritmului Greedy Best-First Search (Ordinea de expandare Greedy: A → B (h=1, cel mai mic) → E (h=2) → C (h=5) → G (h=0, scop găsit)

Observație:

- Greedy aplică euristica minimă h(B)=1 pentru nodul B care pare extrem de promițător, deci algoritmul se aruncă spre B și apoi spre E – deși aceste noduri nu duc spre scop mai repede. Abia după ce ramura B este epuizată, algoritmul revine la C, care avea h=5 (aparent neatractiv), dar era de fapt calea corectă.
- Greedy nu se uită niciodată la g(n): costul real acumulat e complet ignorat, spre deosebire de UCS (care ignoră h(n)) sau A* (care combină ambele). De aceea Greedy nu garantează optimalitatea – poate găsi o soluție mai scumpă decât cea reală, sau poate irosi timp explorând ramuri atractive euristic, dar greșite.
- Comparație directă cu alți algoritmi: UCS a irosit o expandare (B) din lipsă de euristică (vezi tema 3); Greedy a irosit două (B și E) dintr-o euristică înșelătoare; A* nu a irosit niciuna, pentru că a combinat g(n) și h(n) (vezi în continuare)..

Limitarea centrală a Greedy Best-First Search

Prin ignorarea completă a costului g(n) deja parcurs, algoritmul poate fi „păcălit" de o euristică optimistă local, alegând un drum sub-optim sau chiar infinit mai lung decât soluția reală. Această limitare motivează direct algoritmul A*, care combină ambele surse de informație.

4.3. Algoritmul A* Search

Algoritmul A* combină avantajele UCS (optimalitate, prin g(n)) cu eficiența căutării ghidate euristic (prin h(n)), evaluând fiecare nod n prin funcția:

$$f(n) = g(n) + h(n)$$

unde g(n) este **costul real**, cunoscut, al căii de la rădăcină până la n, iar h(n) este **costul estimat** de la n până la cea mai apropiată stare-scop. A* extinde întotdeauna nodul cu **cea mai mică valoare f(n)** din frontieră – practic, nodul care pare să conducă la soluția globală cea mai ieftină.

- **Când h(n) = 0 pentru orice n:** A* devine echivalent cu UCS.

- **Când g(n) este ignorat (echivalent g(n)=0):** A* devine echivalent cu Greedy Best-First Search.

Algoritmul A* reprezintă, prin urmare, generalizarea ambilor algoritmi discutați anterior, combinând informația reală acumulată cu estimarea euristică a drumului rămas.

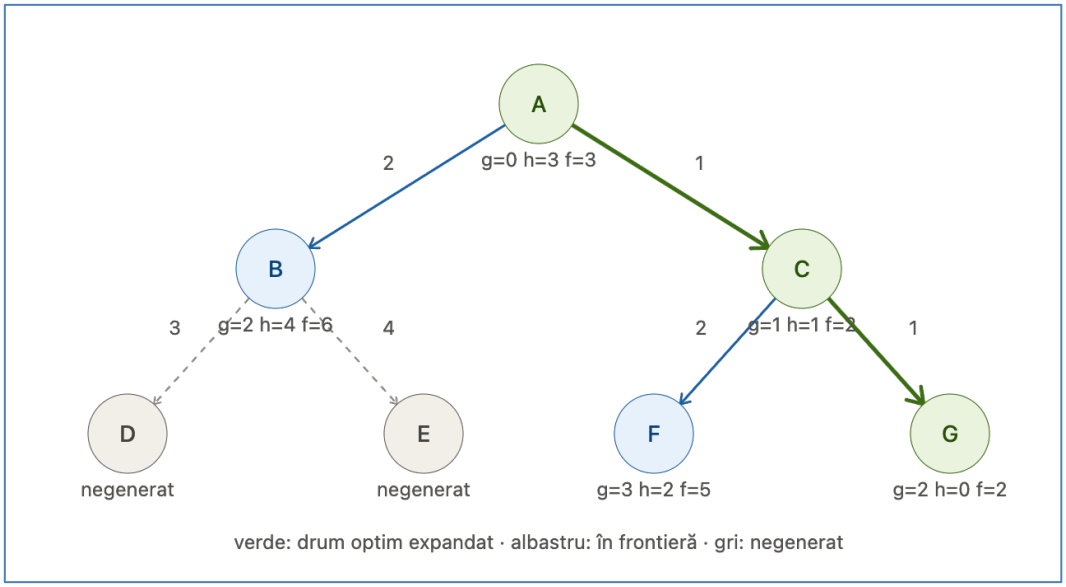


Figura 4.2 Algoritmul A*: f(n) = g(n) + h(n) – costul real acumulat g(n) combinat cu estimarea euristică h(n) ghidează selecția traseului optim.

Algoritmul A*, nu mai parcurge tot arborele, ci alege la fiecare pas nodul cu cel mai mic **f(n) = g(n) + h(n)** – costul real de la start plus estimarea euristică până la scop. Traseul de expandare A* pe acest arbore:

1. **A** este dezambulat (f=3), generând **B** (f=6) și **C** (f=2) în frontieră
2. Se alege nodul cu cel mai mic f din frontieră: **C** (f=2) – nu B, deși B a fost generat mai devreme
3. Expandarea lui C generează **F** (f=5) și **G** (f=2)
4. Se alege din nou minimul: **G** (f=2) – cum G este scopul, căutarea se oprește aici.

💡 **Observație:** Nodul **B rămâne neexpandat** pe tot parcursul, iar nodurile **D și E nu sunt niciodată generate** – spre deosebire de BFS sau DFS, care ar fi explorat întregul arbore, A* investighează doar ramurile promițătoare conform euristicii, garantând totuși optimalitatea drumului (cost total 2), **cu condiția ca euristica h să fie admisibilă** (să nu supraestimeze niciodată costul real rămas până la scop) – o precizare importantă de menționat, pentru că altfel A* poate pierde garanția de optimalitate.

4.4. Euristici

Euristici admisibile

O euristică h(n) este **admisibilă (admissible)** dacă nu supraestimează niciodată costul real de la n până la scop**: h(n) ≤ h*(n),** unde h*(n) este costul real (necunoscut, dar existent) al celei mai bune căi de la n la scop. O euristică admisibilă este, altfel spus, întotdeauna „optimistă” – poate subestima drumul rămas, dar nu îl poate supraestima niciodată.

Teoremă (parafrazată)

Dacă h(n) este admisibilă, algoritmul A* folosind căutare pe arbore (tree search) găsește întotdeauna soluția optimă.

Euristici consistente

O euristică h(n) este **consistentă (consistent)** (sau satisface inegalitatea triunghiului) dacă, pentru fiecare nod n și fiecare succesor n′ obținut printr-o acțiune de cost c(n, n′), este satisfăcută condiția: $h(n) \leq c(n, n') + h(n')$.

Orice euristică consistentă este automat și admisibilă (reciproca nu este întotdeauna adevărată). Consistența este condiția mai puternică, necesară pentru a garanta optimalitatea A* și atunci când se folosește căutare pe graf (graph search), cu o mulțime explorată care evită reexpandarea nodurilor deja vizitate.

Dacă o euristică este consistentă, atunci $f(n)$ este non-descrescătoare. Acest lucru garantează că odată ce un nod este extras din frontieră în căutarea pe graf, drumul către el este cel **optim**. Fără consistență, ar fi necesar să re-extindem noduri deja vizitate, ceea ce crește complexitatea."

Tabelul 4.1 Proprietățile euristicilor

Proprietate	Definiție	Garantează
Admisibilă	$h(n) \leq h^*(n)$ pentru orice n (nu supraestimează niciodată)	Optimalitatea A* pe arbore de căutare
Consistentă	$h(n) \leq c(n,n') + h(n')$ pentru orice n și succesor n′	Optimalitatea A* pe graf de căutare; valorile f(n) sunt monoton crescătoare de-a lungul oricărei căi

Calitatea euristicii

O euristică este utilă dacă este apropiată de $h^*(n)$ (costul real) Dacă $h(n)$ este prea optimistă (ex: $h(n)$=0, algoritmul devine UCS și pierde eficiența.

Dacă $h(n)$este prea „agresivă" (supraestimă), algoritmul devine Greedy și pierde optimalitatea. Scopul inginerului de IA este să găsească „Sweet Spot-ul" euristicii: maximum de informativă, minim de calcul."

Exemplu practic de căutare informată: Navigarea GPS

Aplicațiile de navigare (de exemplu, Google Maps) rezolvă, în esență, o problemă de căutare A* pe graful format din intersecții (noduri) și segmente de drum (arce ponderate cu timpul sau distanța de parcurs).

Tabelul 4.2. Navigarea GPS

Element al algoritmului	Corespondent în navigarea GPS
Stare	Intersecție / poziție curentă pe hartă
Acțiuni	Alegerea următorului segment de drum accesibil din intersecția curentă
g(n)	Distanța (sau timpul) real parcurs de la punctul de plecare până la intersecția n
h(n)	Distanța în linie dreaptă (euclidiană) sau distanța Manhattan de la n până la destinație
f(n) = g(n) + h(n)	Estimarea timpului/distanței totale a rutei care trece prin n

- **Distanța Manhattan** ($|x_1-x_2| + |y_1-y_2|$) este o euristică potrivită pentru deplasare pe o grilă de străzi ortogonale (tipic pentru cartiere urbane organizate în grilă), unde deplasarea diagonală directă nu este posibilă.
- **Distanța euclidiană (în linie dreaptă)** este mai potrivită atunci când deplasarea directă, în orice direcție, este posibilă (de exemplu, pentru un dronă sau pentru estimarea „as the crow flies").

Ambele euristici sunt admisibile în contextele respective, deoarece distanța reală de parcurs pe drumuri nu poate fi niciodată mai mică decât distanța în linie dreaptă sau decât distanța Manhattan corespunzătoare – condiție esențială pentru garantarea optimalității traseului calculat de aplicația de navigare.

4.5. Căutare în medii adversariale Algoritmul Minimax.

Până acum au fost examinate probleme cu un singur agent, care acționează într-un *mediu pasiv*. În jocurile de strategie de tipul X și 0, Dame sau Șah, agentul concurează cu un adversar care își optimizează, la rândul lui, propriile decizii – mediul devine *adversarial*. Acest tip de mediu are următoarele caracteristici:

Jocul este cu sumă nulă (zero-sum game): câștigul unui jucător este exact pierderea celuilalt; suma utilităților celor doi jucători este constantă (de obicei 0).

Informația este completă (perfect information): ambii jucători cunosc, în orice moment, starea completă a jocului (spre deosebire de jocuri precum pokerul, unde informația este parțială).

Pentru astfel de jocuri, spațiul de stări devine un arbore de joc (game tree), în care nivelurile alternează între mutările jucătorului MAX (agentul nostru, care urmărește *maximizarea* propriei utilități) și mutările jucătorului MIN (adversarul, care urmărește *minimizarea* utilității agentului MAX – adică maximizarea propriei sale utilități).

Algoritmul Minimax calculează, pornind de la **frunzele** arborelui de joc (**stări finale** sau stări la care se aplică o funcție de evaluare) și **urcând recursiv spre rădăcină**, valoarea minimax a fiecărei stări:

În nodurile MAX, valoarea nodului = **maximul valorilor copiilor săi** (MAX alege mutarea cea mai favorabilă lui).

În nodurile MIN, valoarea nodului = **minimul valorilor copiilor săi** (adversarul alege mutarea cea mai defavorabilă pentru MAX).

Formularea recursivă a Minimax

MINIMAX(s) = Utilitate(s), dacă s este o stare terminală;

MINIMAX(s) = max peste toate acțiunile a din s { MINIMAX(RESULT(s,a)) }, dacă este rândul lui MAX;

MINIMAX(s) = min peste toate acțiunile a din s { MINIMAX(RESULT(s,a)) }, dacă este rândul lui MIN.

Strategia optimă a lui MAX este, prin definiție, acțiunea care conduce la starea copil cu cea mai mare valoare minimax – presupunând că adversarul joacă, la rândul său, optim (asumpție centrală a algoritmului).

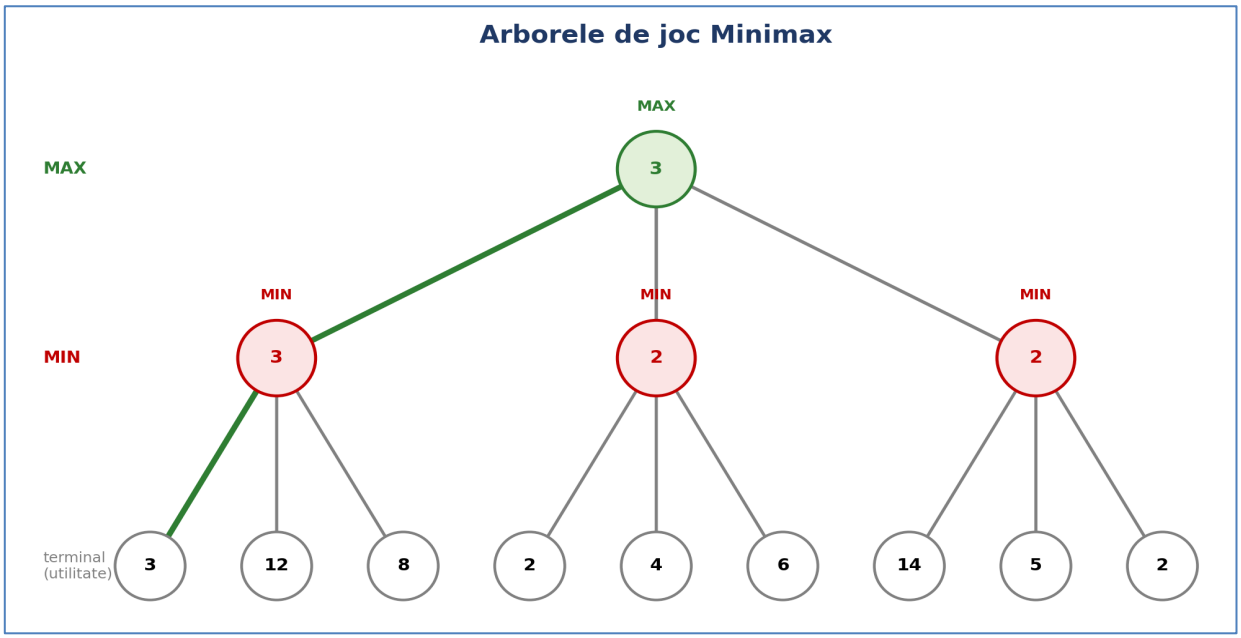


Figura 4.3 Arborele de joc Minimax: valorile se propagă de la frunze (utilități terminale) spre rădăcină, alternând maximizarea (MAX) și minimizarea (MIN).

- **Completitudine:** DA, pe arbori de joc finiți.
- **Optimalitate:** DA, împotriva unui adversar care joacă, de asemenea, optim.
- **Complexitate temporală:** O(b^m), unde m este adâncimea maximă a arborelui de joc – identică, ca ordin de mărime, cu DFS, deoarece Minimax parcurge exhaustiv arborele de joc printr-o explorare de tip depth-first.
- **Complexitate spațială:** O(b*m), similar DFS.

4.6. Optimizarea Alpha-Beta Pruning

Complexitatea exponențială O(b^m) face imposibilă explorarea completă a arborelui de joc pentru jocuri precum Șahul (unde b ≈ 35, m poate depăși 80). Alpha-Beta Pruning este o optimizare exactă a Minimax – produce exact aceeași decizie ca Minimax complet, dar elimină (taie) ramuri ale arborelui despre care se poate demonstra, logic, că nu vor influența niciodată decizia finală.

- **α (alpha):** cea mai bună valoare (cea mai mare) găsită până acum pe drumul curent pentru jucătorul MAX.
- **β (beta):** cea mai bună valoare (cea mai mică) găsită până acum pe drumul curent pentru jucătorul MIN.
- **Regula de tăiere (pruning):** dacă, în timpul explorării unui nod, se constată că $\alpha \geq \beta$, ramurile rămase neexplorate ale acelui nod pot fi ignorate complet, deoarece este garantat, logic, că adversarul (sau agentul MAX, în funcție de nivel) nu va permite niciodată atingerea acelei ramuri în cadrul unei ***jucări optime***.

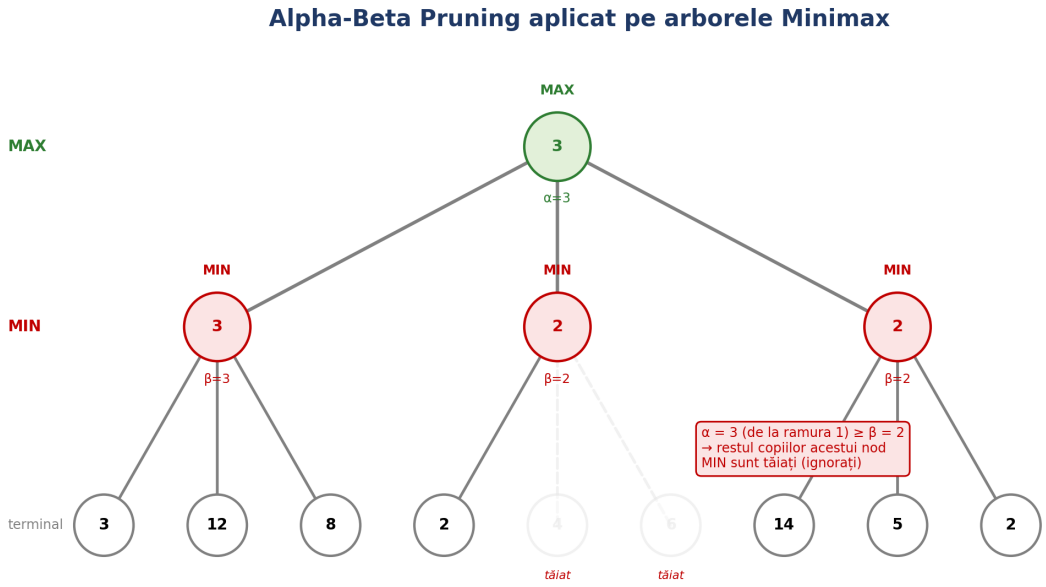


Figura 4.4 Alpha-Beta Pruning aplicat arborelui Minimax: ramurile marcate „tăiat” sunt eliminate din explorare fără a afecta decizia finală.

> 💡 **Rezultat esențial privind eficiența Alpha-Beta Pruning**
>
> În cel mai bun caz (atunci când ordinea de explorare a mutărilor este optimă), Alpha-Beta Pruning reduce complexitatea temporală de la O(b^m) la O(b^(m/2)) – echivalent cu a putea căuta de două ori mai adânc în același timp de calcul, comparativ cu Minimax simplu, fără a sacrifica optimalitatea deciziei.
>
> Eficiența Alpha-Beta Pruning este direct dependentă de ordinea de sortare a mutărilor. Strategii de Move Ordering (ex: explorarea mai întâi a mutărilor care captează piese sau atacă direct) sunt esențiale pentru a maximiza tăierile ale ramurilor.”

Limitări practice: timpul de calcul și funcțiile de evaluare

Chiar și cu Alpha-Beta Pruning, jocuri precum Șahul au arbori de joc mult prea mari pentru a fi explorați până la stările terminale (final de joc) în timp real. Soluția practică folosită de toate motoarele de șah/dame moderne combină două elemente:

- **limitarea adâncimii de căutare** (depth-limited search) – arborele este explorat doar până la o adâncime fixă, rezonabilă din punct de vedere al timpului de calcul disponibil;
- **funcție de evaluare** (evaluation function) – la adâncimea-limită, în locul unei stări terminale reale, se aplică o funcție euristică ce estimează „cât de bună” este acea stare intermediară pentru jucătorul MAX (de exemplu, în șah: diferența de material, controlul centrului tablei, siguranța regelui, mobilitatea pieselor).

> 💡 Calitatea unui program de joc depinde, în mod direct de calitatea funcției sale de evaluare – la fel cum calitatea unui algoritm A* depinde de calitatea funcției euristice h(n). În sistemele moderne (ex: AlphaGo), funcția de evaluare $h(n)$ sau utilitatea terminală în Minimax nu este scrisă manual de oameni, ci este învățată de o Rețea Neuronală (Deep Learning) prin procesul de Reinforcement Learning. Aceasta este puntea dintre căutarea clasică și IA contemporană.”

Întrebări de autoevaluare

1. Care este diferența dintre $g(n)$ și $h(n)$ în contextul algoritmului A*?
2. Ce înseamnă faptul că o funcție euristică este „admisibilă” și de ce este această proprietate esențială pentru garanția de optimalitate a algoritmului A*?
3. Demonstrați, printr-un exemplu propriu, diferența dintre o euristică admisibilă și una neadmisibilă.
4. De ce algoritmul Greedy Best-First Search nu este considerat optim, deși „țintește” direct spre scop?
5. Ce înseamnă condiția de consistență a unei euristici și de ce este ea necesară pentru a garanta optimalitatea în căutarea pe graf (unde se evită re-extinderea nodurilor)?
6. Explicați, cu propriile cuvinte, în ce condiții Alpha-Beta Pruning poate elimina o ramură întreagă a arborelui de joc, fără a afecta decizia finală.
7. De ce distanța Manhattan este adesea o euristică mai informativă decât distanța euclidiană pentru o grilă urbană de străzi?
8. Din perspectiva unui inginer de IA, ce înseamnă găsirea „Sweet Spot-ului” pentru o funcție euristică?
9. Propuneți o funcție de evaluare simplă (2-3 criterii) pentru jocul de Dame, aplicabilă unei stări intermediare a jocului.

CAPITOLUL 5. REPREZENTAREA CUNOȘTINȚELOR ȘI MODELAREA RAȚIONAMENTULUI

5.1. Reprezentarea cunoștințelor în IA clasică (Symbolic AI)

Un sistem inteligent trebuie să acționeze rațional într-un mediu pe care nu îl percepe direct, ci prin intermediul unor modele interne. Reprezentarea cunoștințelor (Knowledge Representation, KR) este disciplina care se ocupă cu găsirea unor formalisme prin care faptele, obiectele, relațiile și regulile din lumea reală pot fi codificate în structuri pe care un calculator le poate manipula, stoca și interoga în mod sistematic.

Fără o reprezentare adecvată, un program nu poate „știi" nimic în sens util: el nu poate deduce consecințe, nu poate explica de ce a ajuns la o concluzie și nu poate reutiliza cunoștințe într-un context nou. De aceea, în IA clasică (denumită și GOFAI – Good Old-Fashioned AI, sau Symbolic AI), reprezentarea cunoștințelor și raționamentul formal reprezintă nucleul arhitectural al oricărui sistem inteligent.

IA simbolică pornește de la ipoteza sistemului de simboluri fizice (Newell și Simon, 1976): ***un sistem capabil de comportament inteligent general poate fi construit prin manipularea unor structuri de simboluri discrete, conform unor reguli explicite, definite de proiectant***. Această paradigmă se opune abordărilor sub-simbolice (rețele neuronale, învățare automată statistică), în care cunoștințele sunt distribuite implicit în **ponderi numerice**, nu în **structuri simbolice explicite** și interpretabile de om. Paradigma simbolică:

- **Cunoștințele** sunt reprezentate explicit, sub formă de simboluri (predicate, reguli, fapte) – nu implicit, în parametri numerici.
- **Raționamentul** este un proces de manipulare formală a simbolurilor, guvernat de reguli de inferență riguroase, verificabile matematic.
- **Sistemele simbolice** oferă transparență și explicabilitate: se poate reconstitui exact lanțul de deducții care a condus la o concluzie.
- **Limitări:** dificultăți în gestionarea incertitudinii, a cunoștințelor incomplete și a domeniilor foarte vaste (problema achiziției cunoștințelor).

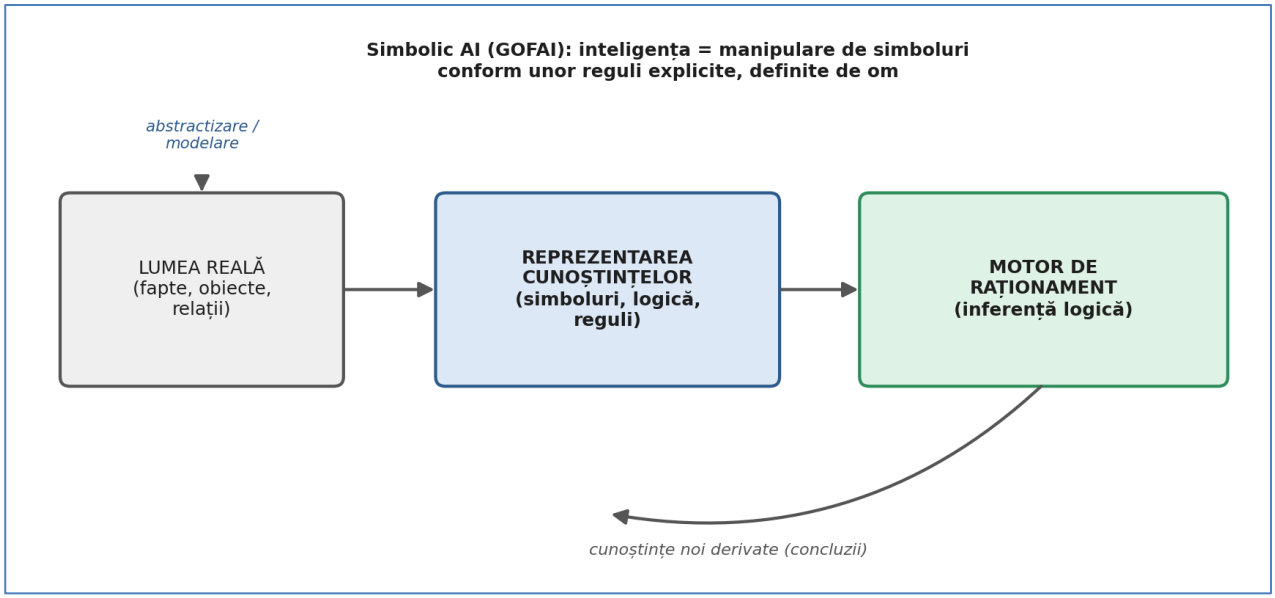


Figura 5.1 Ciclul de bază al unui sistem simbolic: lumea reală este abstractizată în reprezentări simbolice, asupra cărora acționează un motor de raționament care produce concluzii noi.

Orice sistem simbolic de tip „knowledge-based” este alcătuit, la nivel conceptual, din două componente fundamentale:

- **Baza de cunoștințe** (Knowledge Base, KB) – o colecție de fapte și reguli, exprimate într-un limbaj formal (de regulă logică propozițională sau logica predicatelor de ordinul întâi).
- **Motorul de inferență** (Inference Engine) – modulul care aplică reguli de deducție asupra bazei de cunoștințe pentru a genera concluzii noi sau a răspunde la interogări.

Alegerea formalismului de reprezentare este esențială: el trebuie să fie suficient de expresiv pentru a capta cunoștințele domeniului, dar și suficient de restrictiv pentru ca inferența să rămână decidabilă (posibil de realizat) și eficientă din punct de vedere computațional. Logica propozițională și logica predicatelor de ordinul întâi, prezentate în continuare, reprezintă cele două formalisme fundamentale ale IA simbolice – punctul de plecare istoric și conceptual pentru toate sistemele bazate pe reguli.

5.2. Logica propozițională: sintaxa, semantica, inferența

Logica propozițională (logica enunțurilor) este o reprezentare **declarativă** a cunoașterii care operează cu **propoziții atomice** – enunțuri care pot fi doar **adevărate** sau **false**, notate de regulă cu litere P, Q, R, ... Din propoziții atomice se construiesc **formule complexe** cu ajutorul **conectorilor logici**:

Tabel 5.1.Formule logice

Conector	Simbol	Citire	Exemplu
Negație	¬P	„nu P”	¬Plouă
Conjuncție	P ∧ Q	„P și Q”	Plouă ∧ Frig
Disjuncție	P ∨ Q	„P sau Q”	IauUmbrela ∨ RamânAcasă
Implicație	P → Q	„dacă P atunci Q”	Plouă → IauUmbrela
Echivalență	P ↔ Q	„P dacă și numai dacă Q”	Frig ↔ Temperatura<10

O *formulă bine formată* (FBF) se construiește recursiv:

1) orice simbol propozițional este o FBF;
2) dacă α este o FBF, atunci ¬α este o FBF;
3) dacă α și β sunt două FBF-uri, atunci (α ∧ β), (α ∨ β), (α → β), (α ↔ β) sunt de asemenea FBF-uri.

Această definiție recursivă garantează că orice enunț complex poate fi descompus univoc într-un arbore sintactic.

Semantica atribuie sens formulelor prin noțiunea de interpretare (sau model): o funcție I care asociază fiecărui simbol propozițional o valoare de adevăr din mulțimea {Adevărat, Fals}. Valoarea de adevăr a unei formule complexe sub o interpretare dată se calculează recursiv, pe baza tabelelor de adevăr ale conectorilor.

Tabel 5.2. Tabel de adevăr al conectorilor logici

P	Q	¬P	P ∧ Q	P ∨ Q	P → Q	P ↔ Q
A	A	F	A	A	A	A
A	F	F	F	A	F	F
F	A	A	F	A	A	F
F	F	A	F	F	A	A

La combinarea propozițiilor atomice în propoziții compuse (formule complexe) trebuie de evitat **tautologia** și **contradicția**.

- **Tautologia** reprezintă o **propoziție compusă** care este **totdeauna adevărată** în timp ce propozițiile atomare din componenta ei sunt adevărate sau false. Exemplu: Pv¬P (fraza lui Hamlet: ”*A fi sau nu a fi*” este o tautologie).
- **Contradicția** reprezintă o **propoziție compusă** care este **totdeauna falsă** în timp ce propozițiile elementare din componența ei sunt adevărate sau false. Exemplu: P^¬P (frazele ”A fi și nu a fi” sau ”bun și rău” sunt exemple de contradicție.

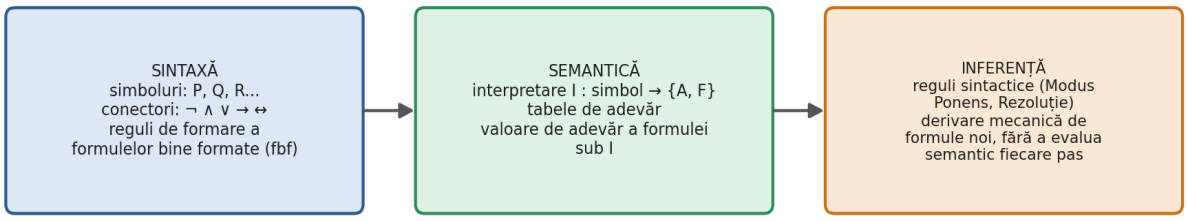
În cazul când într-o propoziție compusă (formulă complexă) se folosesc mai multe tipuri de conectori logici, evaluarea propoziției compuse se face respectându-se următoarele reguli de prioritate a conectorilor logici:

1) Negația

2) Conjuncția

3) Disjuncția Ordinea de evaluare a propozițiilor poate fi schimbată cu ajutorul parantezelor rotunde. În acest caz o prioritate mai mare au conectorii logici din parantezele interne.

O formulă α este ***realizabilă*** dacă există cel puțin o interpretare sub care α este adevărată; este ***validă*** (tautologie) dacă este adevărată sub orice interpretare; și este o ***consecință logică*** a unei baze de cunoștințe KB (notat KB ⊨ α) dacă α este adevărată în orice interpretare care satisface toate formulele din KB.



Exemplu: tabelul de adevăr pentru P → Q

P	Q	P → Q
A	A	A
A	F	F
F	A	A
F	F	A

Reguli de inferență uzuale:

- Modus Ponens: din P și P→Q, se deduce Q
- Modus Tollens: din ¬Q și P→Q, se deduce ¬P
- Rezoluție: din (PvQ) și (¬PvR), se deduce (QvR)

Figura 5.2 Relația dintre sintaxă, semantică și inferență în logica propozițională, alături de tabelul de adevăr al implicației și principalele reguli de inferență.

Un procedeu de a decide asupra realizabilității unei formule (sau a caracterului de tautologie sau de contradicție) care nu recurge la tabelul de adevăr a fost propus de Quine. Conform acestei metode inițial se identifică cel mai frecvent atom în formula complexă. Apoi se rescrie formula de două ori prin înlocuire atomului identificat cu valoarea V (adevărat) sau F (fals) și se aplică reguli de simplificare.

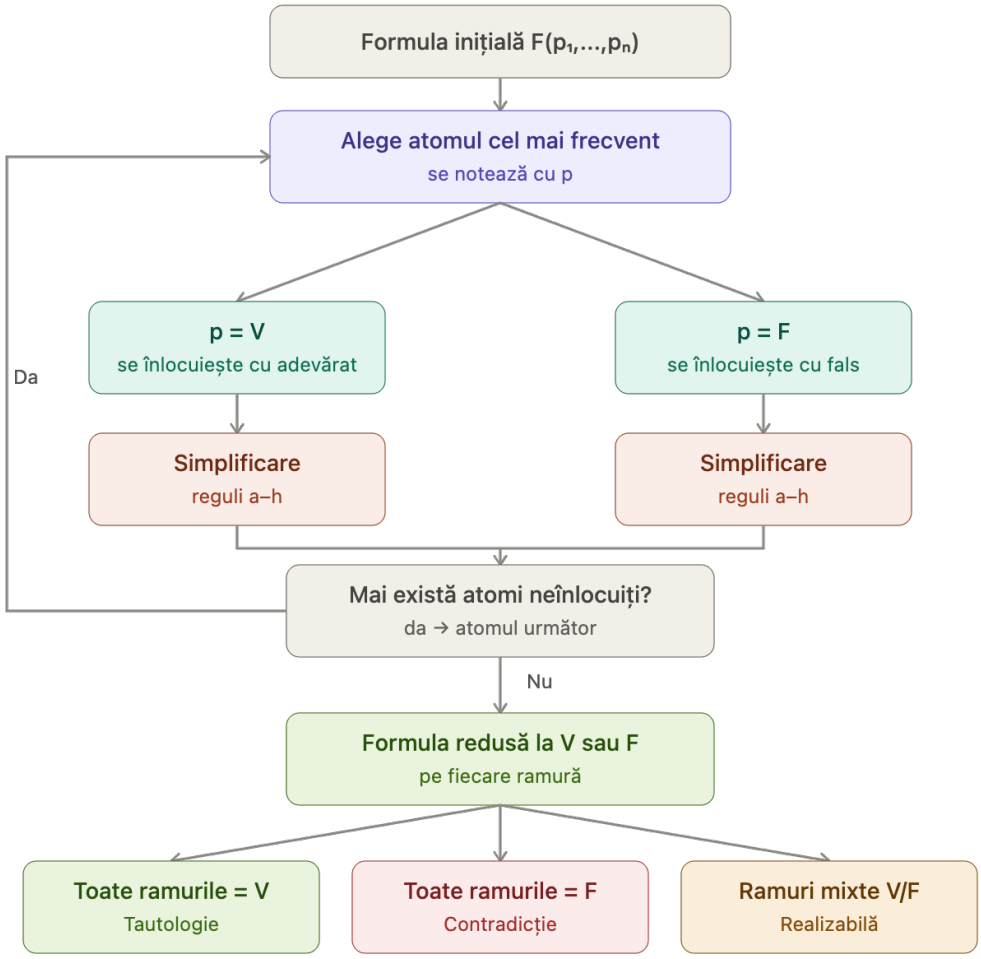


Figura 5.3 Algoritmul metodei lui Quine

Regulile de simplificare aplicate în fiecare ramură:

a) o conjuncție care conține V se reduce prin eliminarea lui V (ex: V∧A → A)
b) o disjuncție care conține F se reduce prin eliminarea lui F (ex: F∨A → A)
c) o conjuncție care conține F devine F, indiferent de ceilalți termeni
d) o disjuncție care conține V devine V, indiferent de ceilalți termeni
e) o implicație cu antecedentul V se reduce la consecvent (V→A ≡ A)
f) o implicație cu consecventul F se reduce la negația antecedentului (A→F ≡ ¬A)
g) o dublă negație a lui V se elimină, rămânând V (¬¬V → V)
h) o dublă negație care conține F ca membru se înlocuiește cu celălalt membru

Aceste reguli se aplică repetat, până formula din fiecare ramură se reduce complet la V sau la F. Dacă toate ramurile finale dau V → formula e tautologie; dacă toate dau F → e contradicție (nerealizabilă); dacă unele dau V și altele F → formula e realizabilă, dar nu e tautologie.

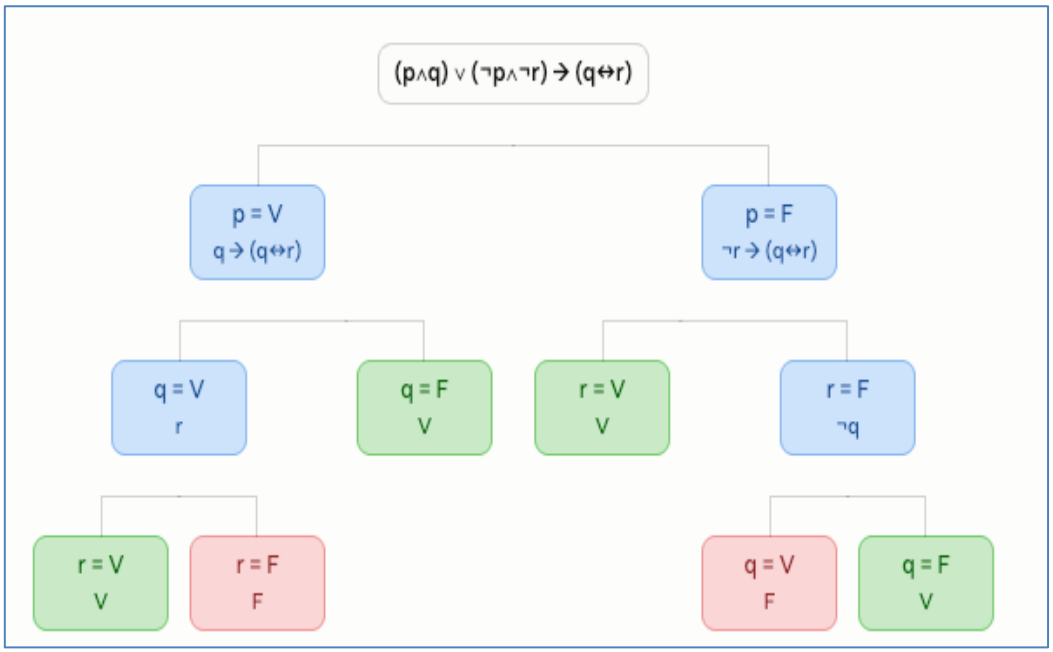


Figura 5.4 Aplicarea metodei lui Quine pentru formula: $(p \wedge q) \vee (\neg p \wedge \neg r) \rightarrow (q \leftrightarrow r)$

Inferența este procesul mecanic de derivare a unor formule noi (concluzii) pornind de la formule cunoscute (premise), aplicând reguli sintactice – fără a fi nevoie să se evalueze semantic fiecare interpretare posibilă. Corectitudinea (soundness) unei reguli de inferență garantează că orice concluzie derivată este într-adevăr o consecință logică a premiselor; completitudinea garantează că orice consecință logică poate fi, în principiu, derivată. Metodele de inferență sunt:

- **Modus Ponens**: din P și P → Q, se deduce Q.
- **Modus Tollens**: din ¬Q și P → Q, se deduce ¬P.
- **Silogismul disjunctiv**: din P ∨ Q și ¬P, se deduce Q.
- **Rezoluția propozițională**: din (P ∨ Q) și (¬P ∨ R), se deduce (Q ∨ R) – regulă centrală pentru demonstrarea automată a teoremelor, deoarece un singur algoritm de rezoluție este suficient pentru a demonstra orice consecință logică, dacă formulele sunt aduse în **forma normală conjunctivă** (CNF).

💡 **Limitarea principală** a logicii propoziționale este generalizarea redusă: ea nu poate exprima relații generale între obiecte („toți studenții sunt oameni") fără a enumera fiecare caz în parte. Această limitare este depășită de logica predicatelor de ordinul întâi.

5.3. Logica de ordinul întâi (FOL): predicate, cuantificatori, constante și variabile

Logica propozițională tratează fiecare enunț ca pe un bloc atomic, opac. Ea nu poate exprima faptul că o proprietate se aplică tuturor elementelor unei categorii, fără a scrie o formulă separată pentru fiecare individ. **Logica de ordinul întâi** (First-Order Logic, FOL) rezolvă această problemă introducând o structură internă a propozițiilor: **obiecte**, **proprietăți** ale acestor obiecte și **relații** dintre ele.

Tabel 5.3 Elementele de bază ale limbajului FOL

Element	Rol	Exemplu
Constante	denumesc un obiect fix, specific, din domeniu	Ion, Chisinau, ASEM
Variabile	denumesc un obiect oarecare, nespecificat	x, y, z

Element	Rol	Exemplu
Predicate	exprimă proprietăți sau relații între obiecte	Student(x), Preda(x, y)
Funcții	denumesc un obiect obținut printr-o operație	MamaLui(x), VarstaLui(x)
Cuantificatori	exprimă domeniul de aplicare al unei variabile	∀ (universal), ∃ (existențial)

Procesul de **unificare** este mecanismul prin care motorul de inferență găsește substituții de variabile care fac două expresii identice. De exemplu, pentru a potrivi regula $Toarce(x) \rightarrow Pisica(x)$ cu faptul $Toarce(Motănel)$, motorul efectuează unificarea $\{x \mapsto Motanel\}$.

Cuantificatorii ∀ și ∃

Cuantificatorul universal ∀x P(x) afirmă că proprietatea P este adevărată pentru fiecare element x din domeniul de discurs – este echivalent, conceptual, cu o conjuncție infinită P(a) ∧ P(b) ∧ P(c) ∧ ... pentru toate obiectele a, b, c din domeniu. **Cuantificatorul existențial** ∃x P(x) afirmă că există cel puțin un element x pentru care P este adevărată – este echivalent, conceptual, cu o disjuncție infinită P(a) ∨ P(b) ∨ P(c) ∨ ...

Tabel 5.4. **Exemple de formalizare**

Enunț în limbaj natural	Formalizare în FOL
Toți studenții învață.	∀x Student(x) → Invata(x)
Există un student care a promovat toate examenele.	∃x Student(x) ∧ ∀y Examen(y) → APromovat(x, y)
Niciun profesor nu este student.	∀x Profesor(x) → ¬Student(x)
Ion este studentul lui Maria.	StudentAl(Ion, Maria)

💡 O **convenție importantă**: cuantificatorul universal se combină, de regulă, cu implicația (∀x Student(x) → Om(x), adică „*toți studenții sunt oameni*”), în timp ce cuantificatorul existențial se combină de regulă cu conjuncția (∃x Profesor(x) ∧ PredaIA(x), adică „*există un profesor care predă IA*”). Inversarea acestei convenții este o greșeală frecventă și produce formule cu sens diferit de cel intenționat.

Datorită acestei capacități de a exprima cuantificări, relații multiple (N-are) și structurii interne a enunțurilor, FOL este **formalismul standard** folosit pentru construirea bazelor de cunoștințe din sistemele expert, ontologii și sisteme de raționament automat, fiind în același timp suficient de restrictiv pentru a permite algoritmi de inferență (precum rezoluția cu unificare) decidabili pe fragmente utile din practică.

Totuși, nu toate situațiile practice pot fi descrise cu ajutorul logicii FOL. Spre exemplu, în propoziția „*Multe conturi bancare se debitează*” pentru cuvântul „**multe**” nu poate fi utilizat nici un cuantificator.

Limitări critice: Deși FOL este puternic, inferența în el poate fi computațional costisitoare. În practica, se folosesc fragmente restricționate (precum Description Logics pentru ontologii) pentru a asigura că sistemul oferă răspunsuri într-un timp acceptabil. De asemenea, sistemele simbolice struggles cu 'common sense' (cunoștințele banale despre lumea reală).

5.4. Baze de fapte și reguli de inferență

Baza de fapte (working memory / factual database) conține enunțurile considerate adevărate la un moment dat despre domeniul modelat – de regulă predicate instanțiate, fără variabile libere (literali de bază / ground literals). Exemplu de bază de fapte pentru un domeniu zoologic simplu:

- Toarce(Motanel)
- AreBlana(Motanel)
- ManancaCarne(Leu)
- AreCoama(Leu)

Baza de fapte este dinamică: pe parcursul unui proces de inferență, fapte noi pot fi adăugate (derivate din reguli), iar în unele arhitecturi faptele pot fi și retrase, atunci când presupunerile care le susțineau nu mai sunt valide (raționament nemonoton -Non-monotonic Reasoning).

O **regulă de inferență** (sau *regulă de producție*) are, în cele mai multe sisteme simbolice, forma condițională:

DACĂ <condiție 1> ȘI <condiție 2> ȘI ... ATUNCI <concluzie>

Partea din stânga (antecedentul / premisele) specifică un set de condiții care trebuie satisfăcute simultan de faptele existente în baza de fapte; partea din dreapta (consecventul) specifică fie un fapt nou care se adaugă în baza de fapte, fie o acțiune care se execută. Regulile pot conține variabile, care se instanțiază (se leagă / bind) cu obiecte concrete atunci când regula se potrivește cu fapte existente — proces numit **unificare** (unification).

💡 **Regulile de producție** urmează formatul: **IF α THEN β**, Unde α (antecedentul) este o condiție simplă sau compusă care trebuie să fie adevărată în Memoria de Lucru, iar β (consecventul) este acțiunea efectuată (adăugarea unui fapt sau o acțiune).

Exemplu de reguli asociate bazei de fapte de mai sus:

- R1: DACĂ Toarce(x) ATUNCI Pisica(x)
- R2: DACĂ Pisica(x) ȘI AreBlana(x) ATUNCI Mamifer(x)
- R3: DACĂ ManancaCarne(x) ȘI AreCoama(x) ATUNCI Leu(x)

Ansamblul format din **baza de fapte** și **regulile de inferență** constituie **baza de cunoștințe** (Knowledge Base) a sistemului. Modul în care regulile sunt aplicate sistematic asupra faptelor pentru a produce concluzii noi este responsabilitatea **motorului de inferență**, prezentat în paragrafele următoare, împreună cu cei doi algoritmi principali de parcurgere: înlănțuirea înainte și înlănțuirea înapoi

5.5. Arhitectura unui sistem de producție

Un sistem de producție (production system) este arhitectura clasică prin care se implementează raționamentul bazat pe reguli în IA simbolică. El este alcătuit din trei componente principale:

- **Baza de cunoștințe** – mulțimea regulilor de producție ale domeniului (relativ statică, definită de expert).
- **Memoria de lucru/baza de fapte** (working memory/ factal base) – mulțimea faptelor curente, care se modifică pe parcursul execuției.
- **Motorul de inferență** – componenta activă, responsabilă cu ciclul „recunoaște – acționează” (recognize–act cycle).

Această arhitectură separă în mod explicit cunoștințele despre domeniu (regulile) de mecanismul general de control (motorul de inferență), ceea ce este considerat unul dintre marile avantaje ale sistemelor simbolice: aceleași strategii de inferență pot fi reutilizate pentru domenii complet diferite, doar prin înlocuirea bazei de cunoștințe.

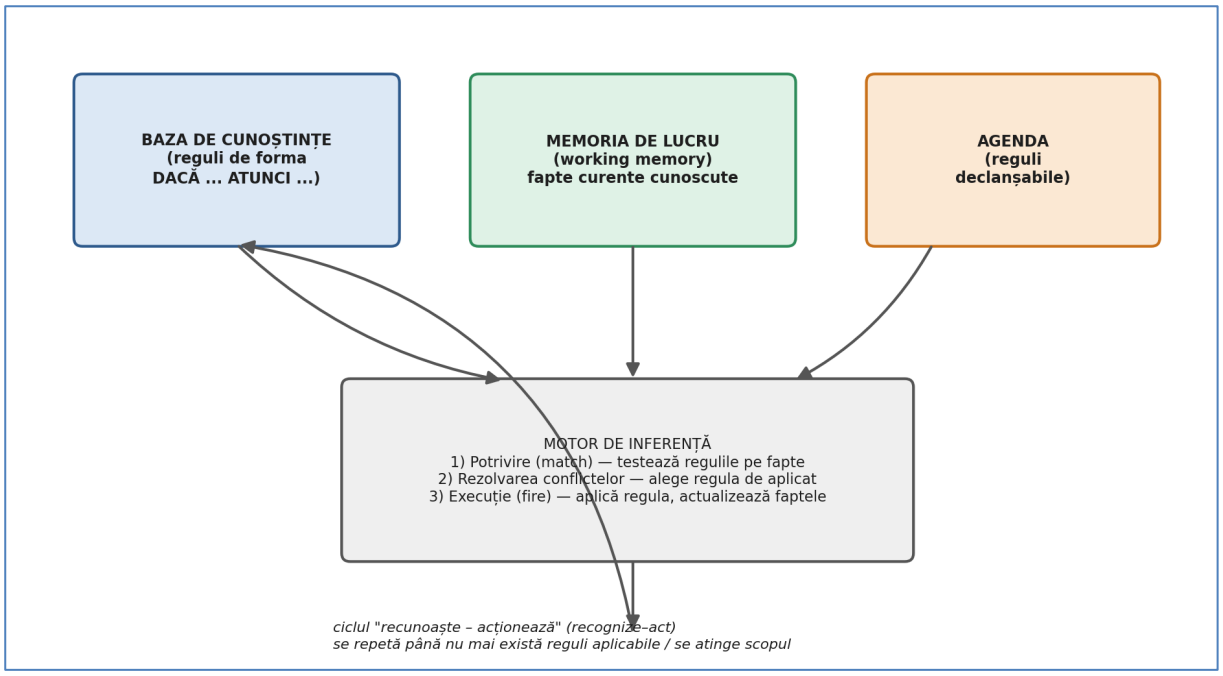


Figura 5.5 Arhitectura unui sistem de producție: motorul de inferență parcurge ciclic etapele de potrivire (match), rezolvare a conflictelor (conflict resolution) și execuție (fire), actualizând memoria de lucru.

Motorul de inferență execută repetat un ciclu în trei pași, până când nu mai există reguli aplicabile sau se atinge scopul urmărit:

- **Potrivire** (Match) – se identifică toate regulile ale căror premise sunt satisfăcute de faptele curente din memoria de lucru; acest set se numește agenda sau conflict set.
- **Rezolvarea conflictelor** (Conflict Resolution) – dacă mai multe reguli sunt aplicabile simultan, se alege una singură, pe baza unei strategii (ex. prima regulă potrivită, regula cea mai specifică, regula cel mai recent adăugată – recency).
- **Execuție** (Fire) – regula aleasă se aplică efectiv: faptul nou din concluzie se adaugă (sau, în unele sisteme, un fapt se retrage) în memoria de lucru.

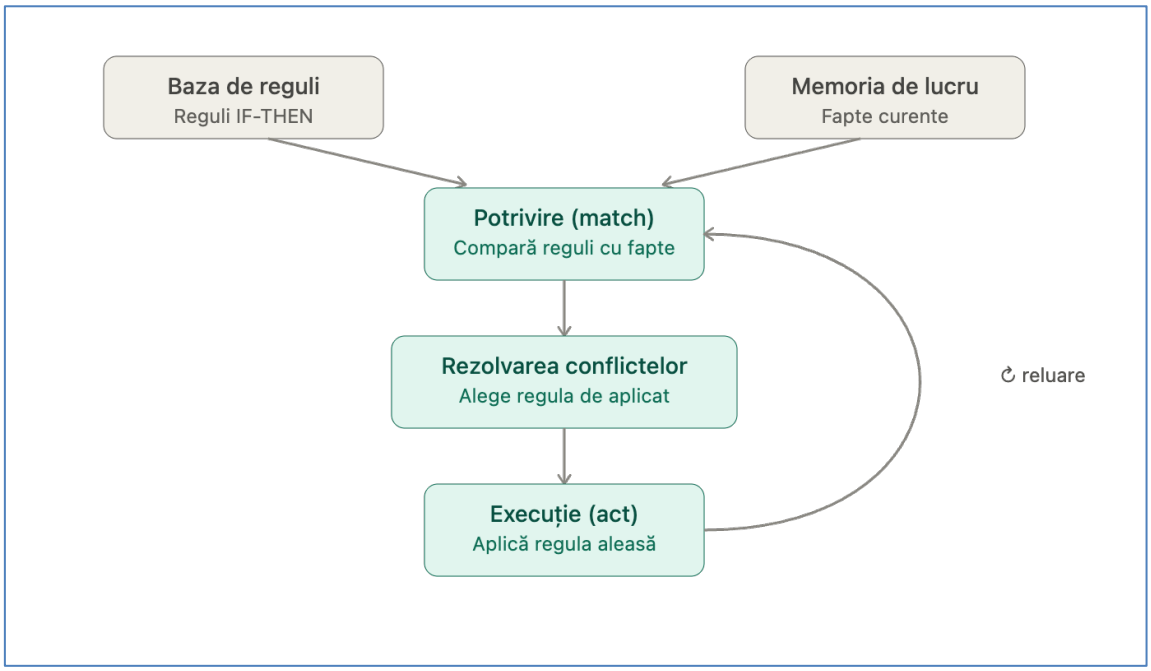


Figura 5.6 Ciclul motorului de inferență

5.6. Algoritmi de inferență: Forward Chaining și Backward Chaining

Înlănțuirea înainte (Forward Chaining) este o **strategie de inferență** dirijată de date (data-driven): motorul pornește de la **faptele cunoscute** și aplică repetat regulile ale căror premise sunt satisfăcute, adăugând concluziile ca fapte noi, până când nu mai există reguli aplicabile sau apare faptul-scop urmărit. Este strategia naturală atunci când scopul este de a deriva toate consecințele posibile ale unui set inițial de observații (de exemplu, sisteme de monitorizare sau diagnostic continuu).

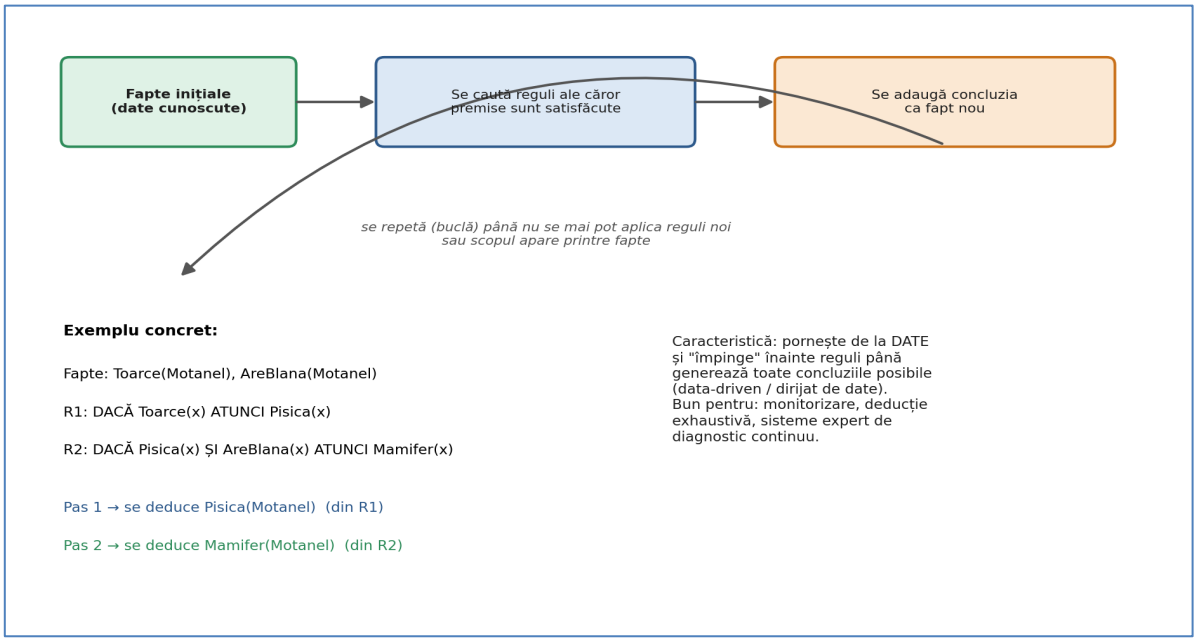


Figura 5.7 Înlănțuirea înainte: se pornește de la faptele inițiale, se aplică regulile ale căror premise sunt satisfăcute, iar concluziile se adaugă ca fapte noi, până la epuizarea regulilor aplicabile.

Pe baza faptelor și regulilor R1–R2 definite mai înainte, algoritmul de înlănțuire înainte procedează astfel:

- Pas 0 – fapte inițiale: Toarce(Motanel), AreBlana(Motanel).
- Pas 1 – se potrivește R1 (Toarce(x) → Pisica(x)) cu x = Motanel; se adaugă faptul nou Pisica(Motanel).
- Pas 2 – se potrivește R2 (Pisica(x) ∧ AreBlana(x) → Mamifer(x)) cu x = Motanel; se adaugă faptul nou Mamifer(Motanel).
- Algoritmul se oprește: nu mai există reguli ale căror premise sunt integral satisfăcute de faptele curente.

Înlănțuirea înapoi (Backward Chaining) este o strategie de inferență dirijată de **scop** (goal-driven): motorul pornește de la **o ipoteză** (scopul de demonstrat) și caută recursiv reguli a căror concluzie se potrivește cu scopul curent, transformând premisele acelei reguli în noi **sub-scopuri**, până când toate sub-scopurile se reduc la **fapte prezente direct în baza de fapte**. Este strategia naturală atunci când se dorește un răspuns la o întrebare punctuală, fără a deriva toate consecințele posibile ale bazei de cunoștințe.

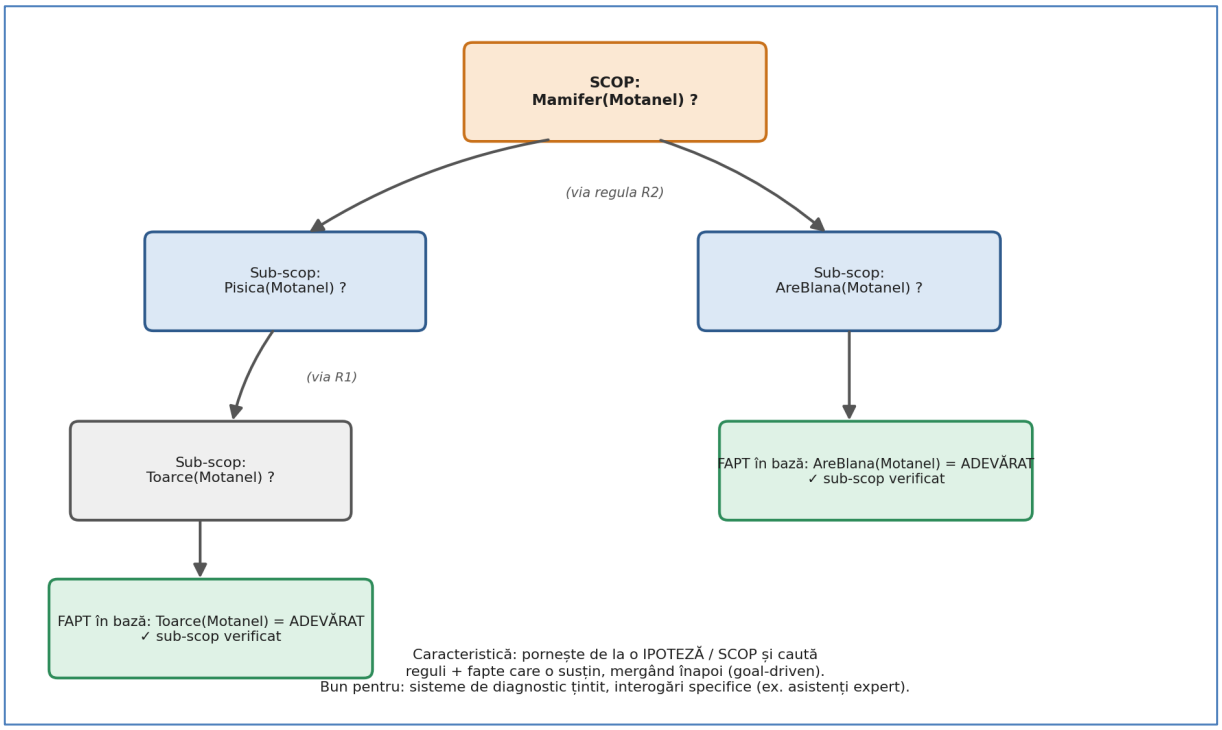


Figura 5.8 Înlănțuirea înapoi: scopul inițial Mamifer(Motanel) este descompus recursiv în sub-scopuri, pe baza regulilor R1 și R2, până când fiecare sub-scop este verificat direct în baza de fapte.

Reluând exemplul anterior, pentru a verifica scopul Mamifer(Motanel), algoritmul de înlănțuire înapoi procedează astfel:

- Se caută o regulă a cărei concluzie se potrivește cu scopul: R2 (Pisica(x) ∧ AreBlana(x) → Mamifer(x)), cu x = Motanel.
- Premisele lui R2 devin sub-scopuri noi: Pisica(Motanel) și AreBlana(Motanel).
- Pentru Pisica(Motanel): se caută R1 (Toarce(x) → Pisica(x)); sub-scopul devine Toarce(Motanel), care este verificat direct în baza de fapte → ADEVĂRAT.
- Pentru AreBlana(Motanel): faptul este prezent direct în baza de fapte → ADEVĂRAT.
- Ambele sub-scopuri fiind confirmate, scopul inițial Mamifer(Motanel) este demonstrat ADEVĂRAT.

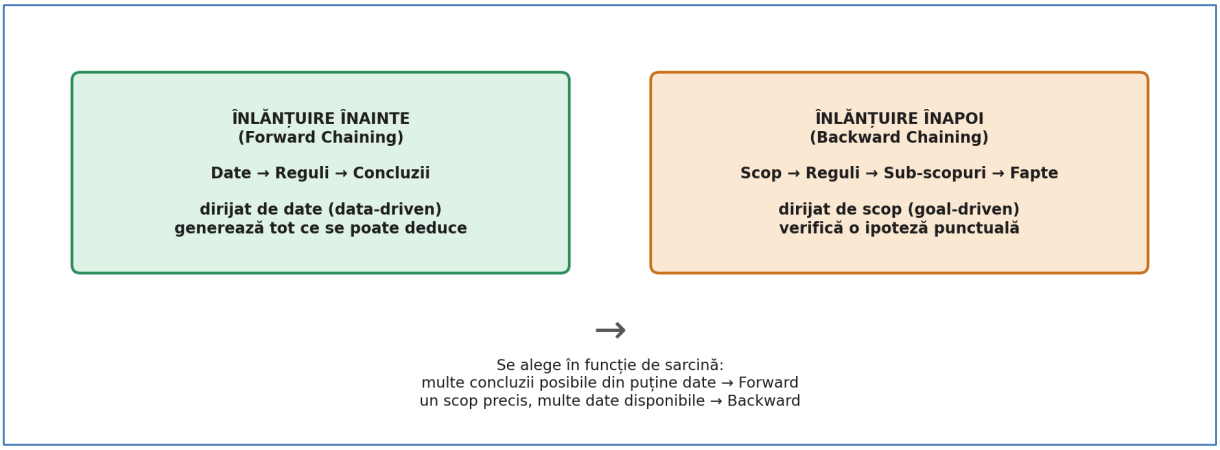


Figura 5.9 Comparație sintetică între înlănțuirea înainte (dirijată de date) și înlănțuirea înapoi (dirijată de scop).

Tabel 5.5 **Comparația algoritmilor de inferență**

Criteriu	Forward Chaining	Backward Chaining
Punct de plecare	faptele cunoscute	ipoteza / scopul de demonstrat
Direcție de raționare	de la date către concluzii	de la scop către dovezi (fapte)
Rezultat tipic	toate concluziile posibile	răspuns Adevărat/Fals la o întrebare punctuală
Eficiență	poate genera concluzii irelevante pt. scop	explorează doar ramurile relevante scopului
Aplicații tipice	monitorizare, sisteme de alertă, diagnostic continuu	sisteme expert consultative, asistenți de diagnostic țintit

În practică, multe motoare de inferență (ex. Prolog pentru backward chaining, sistemele de tip CLIPS/OPS5 pentru forward chaining) și chiar sisteme hibride combină cele două strategii, alegând algoritmul potrivit în funcție de raportul dintre numărul de fapte disponibile și numărul de reguli, precum și de natura interogării (exhaustivă vs. punctuală).

Întrebări de autoevaluare

1. Care este diferența fundamentală dintre paradigma simbolică și cea sub-simbolică (conexionistă) în IA?
2. Cum se distinge sintaxa de semantica unei formule logice? De ce este utilă această distincție pentru automatizarea inferenței?
3. Formalizați în FOL enunțul: „Orice angajat care are vechime de peste 10 ani beneficiază de o zi suplimentară de concediu.”
4. Explicați, folosind un exemplu propriu, diferența dintre înlănțuirea înainte și înlănțuirea înapoi.
5. De ce, pentru o interogare foarte specifică într-o bază de cunoștințe foarte mare, este preferată inferența de înlănțuire înapoi?

CAPITOLUL 6. FUNDAMENTELE PROGRAMĂRII LOGICE ÎN PROLOG

6.1. Introducere în programarea logică

Programarea logică este o paradigmă declarativă în care programatorul descrie CE anume este adevărat despre o problemă (fapte și relații), iar nu CUM trebuie calculat pasul cu pas rezultatul, așa cum se procedează în paradigma imperativă (C, Java, Python).

Prolog ("Programmation en Logique") a fost dezvoltat în 1972 de Alain Colmerauer și Philippe Roussel, la Universitatea din Marsilia, ca instrument pentru procesarea limbajului natural. Astăzi este utilizat pe larg în inteligența artificială: sisteme expert, procesare de limbaj natural, planificare, rezolvarea constrângerilor și baze de cunoștințe.

Un program Prolog este format din **trei elemente fundamentale**, pe care le vom studia în detaliu:

- Fapte (facts) – afirmații necondiționate despre lume;
- Reguli (rules) – afirmații condiționate (implicații logice);
- Interogări (goals) – întrebări adresate bazei de cunoștințe.

> 💡 *Motorul Prolog nu "execută" instrucțiuni – el încearcă să demonstreze, prin deducție logică și căutare, dacă o interogare este adevărată pe baza faptelor și regulilor cunoscute.*

6.2. Alfabetul și separatori Prolog

- **Alfabetul**. La scrierea programelor Prolog se poate utiliza întreaga tabelă de caractere ASCII.
- **Principalul separator** ai termenilor limbajului este virgula (“,”). Virgula poate să apară de asemenea ca separator între elementele unei liste (de fapt o listă este un caz particular de termen compus).
- **Terminarea** unui enunț Prolog se realizează prin caracterul punct (“.”) urmat în mod obligatoriu de spațiu sau terminatorul de sfârșit de linie (caracterele CR,LF). Punctul poate să apară însă și în componența unui termen simbolic cum ar fi **.**, context în care acesta nu are rol de terminator.
- **Comentariile**. Pentru documentare, într-un program Prolog se pot introduce oricând comentarii. În funcție de întinderea (amploarea) acestora, există două modalități de introducere a comentariilor. Caracterul **%** definește drept comentariu conținutul liniei, începând cu caracterul însuși și până la sfârșitul acesteia. Inserarea unui comentariu pe mai multe rânduri se face încadrând textul respectiv între delimitatorii **/*** și ***/**.

6.3. Fapte (facts)

Un **fapt** este o afirmație necondiționată, considerată adevărată în mod absolut în baza de cunoștințe. Sintactic, un fapt este un predicat urmat de **punct**. Faptul este cea mai simplă formă de predicat ce semnifică existența unei relații între o mulțime de obiecte. Sintaxa generală a unui fapt este următoarea:**nume_predicat(obiect1[,obiect2 [,…]]).**

- **nume_predicat** exprimă o **caracteristă** a unui obiect (*oras(orhei).*) sau o **relație** dintre obiecte (*ruta(paris, roma).*)
- o**biect1, obiect2,...** sunt entități (atomi) din domeniul analizat. Numele obiectului (*paris, roma, ion, orhei, carte, automobil*) sau a relației*(ruta, este_autor, tatal_lui*) se scriu în limbajul Prolog cu prima literă minusculă: și pot conține semnul _.
- Numele predicatului se mai numește **functor**.
- Întrucât predicatul *ruta* are două argumente *paris* și *roma*, care sunt puse între paranteze și despărțite prin virgulă, el se numește **predicat binar (functor binar)**. În caz general, un predicat poate avea **orice număr de argumente** (și, bineînțeles, zero). Acest număr se numește **aritatea predicatului**, care în cazul predicatului *ruta* este 2. Acest predicat poate fi identificat ca *ruta/2*. Arietatea predicatului *oras* este 1 – *oras/1*.

- Când se declara o relatie între obiecte trebuie sa se atraga atentia la numarul de obiecte puse în relatie de predicat și la ordinea acestor obiecte. Pentru Prolog faptul *ruta/2* nu este decât o simpla consecutivitate de simboluri si nu-i asociaza nici un sens, nici o semnificatie. Se poate, însă, perfect de considerat ca primul obiect din predicatul *ruta/2* este *orașul plecării,* iar al doilea este *orasul sosirii.* Ordinea este, deci, **arbitrara**, dar fiind facuta o alegere, trebuie sa se țină cont de ea daca dorim ca relatia *ruta/2* sa reprezinte o semnificatie **valida** (și ca un program ce o utilizează să producă răspunsuri coerente).

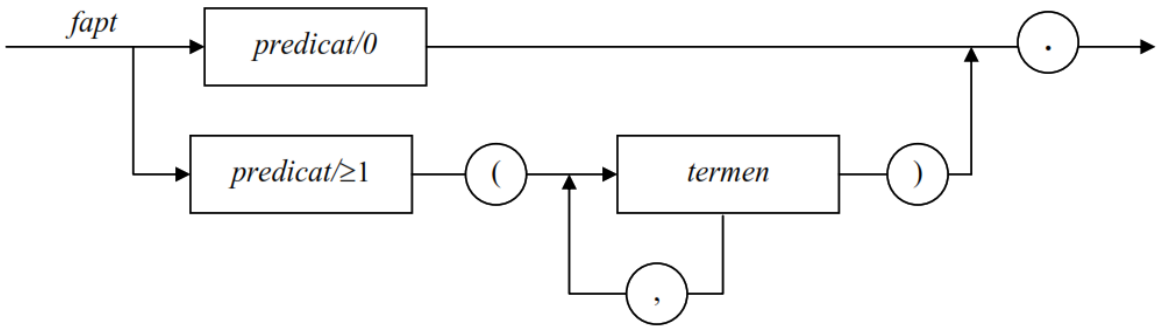


Figura 6.1 Sintaxa unui fapt

💡 **Predicatul** poate fi conceput ca numele relației între obiecte înzestrat cu aritate în care ordinea obiectelor e importanta. **Interpretarea** particulară a predicatelor și argumentelor depinde de programator. Ordinea argumentelor odată fixata, este importanta și trebuie păstrată la orice utilizare a faptului.

Mulțimea tuturor faptelor formează **baza de fapte**:

```
% Fapte despre relația de paternitate
parinte(ion, maria).
parinte(ion, andrei).
parinte(maria, elena).
parinte(andrei, victor).

% Fapte despre gen
barbat(ion).
barbat(andrei).
femeie(maria).
femeie(elena).
```

```
% Fapte despre rutele disponibile
ruta(paris, roma).
ruta(paris, madrid).
ruta(madrid, porto).
ruta(londra, paris).
ruta(londra, madrid).
ruta(amsterdam, paris).
```

💡 *Ordinea faptelor în fișierul sursă contează pentru ordinea în care Prolog găsește soluțiile la backtracking, dar nu afectează corectitudinea logică a programului.*

6.4. Reguli (rules)

O regulă exprimă un fapt condiționat: capul regulii este adevărat dacă toate obiectivele din corp sunt adevărate. Din punct de vedere al logicii matematice, regula este echivalentă cu implicația. Sintaxa generală este:

```
Cap :- Obiectiv1, Obiectiv2, ..., ObiectivN.
```

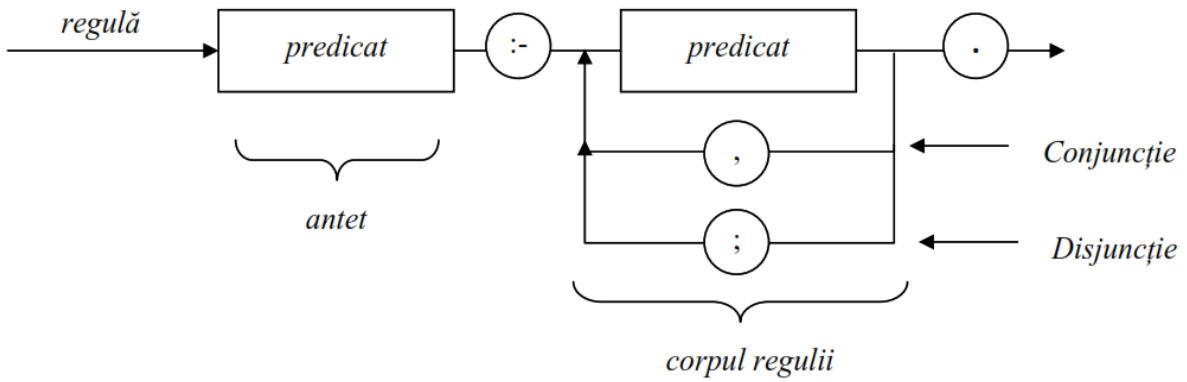


Figura 6.2 Sintaxa unei reguli

Operatorul :- se citește "dacă", iar virgula reprezintă conjuncția logică (ȘI). Exemplu – definirea relației "bunic":

```
bunic(X, Y) :- parinte(X, Z), parinte(Z, Y).
```

Regula de mai sus afirmă: "X este bunicul lui Y dacă există Z astfel încât X este părintele lui Z, iar Z este părintele lui Y." Un alt exemplu, folosind faptele definite anterior:

```
tata(X, Y) :- parinte(X, Y), barbat(X).
mama(X, Y) :- parinte(X, Y), femeie(X).
```

💡 *Regulile pot fi văzute ca proceduri, dar spre deosebire de limbajele imperative, ele nu specifică ordinea pașilor, ci doar condițiile logice de adevăr.*

Variabilele din regulă sunt **locale**. Această înseamnă, că dacă sunt date regulile

```
grec(X):-atenian(X).
grec(X):-spartan(X).
```

variabila X, apărută în prima regulă, nu are nimic comun cu cea din regula a doua, cu toate că au același nume.

Tabel 6.1. **Exemple de formalizare**

Enunț în limbaj natural	Formalizare în FOL	Formalizare în Prolog
Orice atenian este grec.	∀x atenian(x) → grec(x)	grec(X):-atenian(X).
Orice atenian sau spartan este grec	∀x atenian(x) ∨ spartan(x) → grec(x)	grec(X):-atenian(X); spartan(X).
Orice bărbat care este părintele cuiva este tatăl acestuia	∀x ∀y (parinte(x,y) ∧ barbat(x)) → tata(x,y)	tata(X, Y):- parinte(X, Y), barbat(X).

6.5. Interogări (queries)

Interogarea (goal) este mecanismul prin care utilizatorul cere motorului Prolog să verifice adevărul unei afirmații sau să găsească valorile variabilelor care o fac adevărată. La consola Prolog, interogările se introduc după promptul ?-.

```
?- parinte(ion, maria).
true.

?- parinte(maria, ion).
false.

?- bunic(ion, elena).
true.
```

Dacă interogarea conține variabile, Prolog încearcă să găsească toate valorile posibile care satisfac obiectivul, afișându-le pe rând (utilizatorul poate cere soluții suplimentare tastând ";"):

```
?- parinte(ion, X).
X = maria ;
X = andrei.

?- bunic(ion, Y).
Y = elena ;
Y = victor.
```

6.6. Variabile

În Prolog, o variabilă începe întotdeauna cu **literă majusculă** sau cu caracterul underscore (_). Celelalte simboluri pot fi numere, litere majuscule și/sau minuscule. Exemple valide: X, Rezultat, Nume_Persoana, _Temp.

Variabilele se folosesc pentru desemnarea **obiectelor necunoscute**. O variabilă nu este o "cutie de memorie" ca în limbajele imperative, ci un substituent logic: ea poate fi legată (bound/instantiated) la o valoare pe durata demonstrării unui obiectiv sau poate rămâne nelegată (liberă).

```
?- X = 5.
X = 5.

?- X = Y, Y = ana.
X = ana,
Y = ana.
```

Variabila anonimă _ se folosește atunci când valoarea nu ne interesează:

```
are_copil(X) :- parinte(X, _).

?- are_copil(ion).
true.
```

> 💡 *Domeniul de vizibilitate al unei variabile este limitat la clauza (faptul sau regula) în care apare. Aceeași literă X în două reguli diferite reprezintă două variabile distincte.*

6.7. Sintaxa termenilor și a expresiilor

- Orice dată în Prolog este un termen. **Termenul** este singura structură de date utilizată în programarea logică. Termenii sunt formați dintr-o mulțime de caractere ASCII. Deosebim termini **simpli** și termini **structurați**. Diferența dintre diverși termeni se bazează doar pe **sintaxa** lor. Terminii **simpli** se divizează în **constante** și **variabile.** Orice constantă se referă la un obiect individual.
- **Constantele** în Prolog servesc pentru definirea **obiectelor** și **relațiilor** particulare. Modul de interpretare a constantei, cum ar fi locul ei într-o relație, depinde de programator. **Constantele** pot fi **atomi** sau **numere.** Orice nume de predicat este un atom
- **Atomul** este o varietate de termen ce începe cu o literă minusculă. Celelalte simboluri pot fi litere (inclusiv majuscule), numere și simbolul de subliniere: *ion, a2000, a_ToM.* Perechea de simboluri [ ] de asemenea este un atom, însă nici [, nici] aparte nu sunt atomi. O secvență de caractere incluse între două apostrofe '…', de asemenea, este considerat în calitate de atom: *'Programarea logică!', '+32'.*
- **Numerele** sunt formate din cifre, punct și din semnele "+" și "-": *42, 3.14, -7.* Operațiile aritmetice pot fi făcute doar asupra numerelor. Asupra atomului '*33*' nu pot fi aplicate operații aritmetice.
- **Variabila** este un termen ce începe cu **o majusculă** sau cu caracterul de subliniere, "_". Celelalte simboluri pot fi numere, litere majuscule și/sau minuscule: *X, Rezultat, _Y, A25.* Un singur caracter '_' este o formă special de variabilă numită **variabila anonimă**. Ea se utilizează pentru a arăta că pe poziția dată se acceptă **orice valoare** care nu ne interesează.
- **Termenul structurat (compus)** desemneaza un obiect în care sunt regrupate alte obiecte numite componente. Regruparea permite o manipulare mai usoara a obiectelor complexe. Un **termen structurat** se definește prin **functor** urmat de **argumente** între paranteze. Un termen compus are următoarea sintaxă: **functor(t1,t2,…, tn). Aritatea** termenului structurat este determinată de numărul argumentelor din paranteze: **functor/n.**

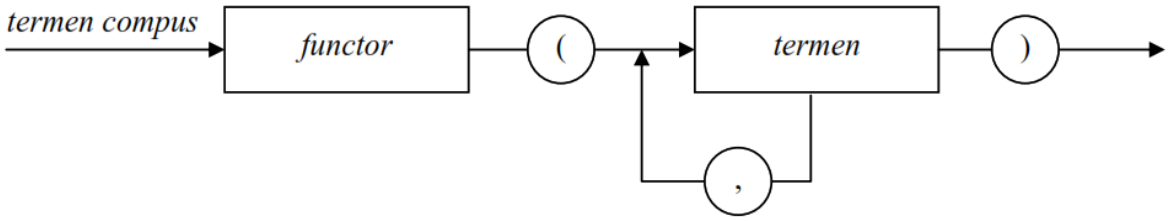


Figura 6.3 Sintaxa unui termen structurat

- Consideram următorul termen structurat: *autor(blaise,pascale).* El poate fi interpretat că exista un *autor* al cărui nume este *pascale* și prenume *blaise.* Acest termen are functorul *autor,* componentele *blaise* și *pascale* și aritatea *2.* Functorul se specifică în referințe ca *autor/2.*
- Componentele unui termen structurat pot fi la rândul sau termeni structurați. Astfel, termenul structurat *cartea(cugetari, autor(blaise, pascale)).* semnifica că „*cugetari este o carte a cărei autor este blaise pascale".* Acest termen are doua componente: *cugetari* și termenul structurat *autor(blaise, pascale).* Functorul se specifică în referințe ca *cartea/2.*
- **Liste** sunt o structură specială de date și sunt examinate separat în continuare.
- **Expresiile aritmetice** se evaluează explicit cu predicatul **is** (Prolog nu evaluează automat expresiile în afara acestui context):

```
?- X is 3 + 4 * 2.
X = 11.

?- Y is (10 - 2) / 4.
Y = 2.
```

```
?- 5 > 3.
true.

?- 5 =:= 5.0.
true.
```

Operatorii de comparație aritmetică (=:=, =\=, <, >, =<, >=) sunt distincți de **operatorul de unificare** (=), aspect esențial pe care îl vom clarifica în paragraful următor.

•

6.8. Unificarea termenilor (unification)

Unificarea este algoritmul prin care Prolog determină dacă doi termeni pot fi făcuți identici, prin instanțierea variabilelor libere. Este operația fundamentală care stă la baza potrivirii interogărilor cu faptele și regulile din baza de cunoștințe. Termenul executabil T1=T2 este de fapt un apel la rutina de unificare a interpretorului Prolog. Dacă cei doi termeni sunt unificabili, atunci predicatul T1=T2 este satisfăcut și variabilele libere din cei doi termeni sunt substituite prin valorile lor corespunzătoare. În caz contrar, termenul T1=T2 este inconsistent, iar variabilele din componența acestora rămân nemodificate.

Reguli de unificare între doi termeni T1 și T2:

- dacă T1 și T2 sunt constante **identice**, unificarea reușește fără nicio legare;
- dacă T1 este o variabilă liberă, ea se leagă la valoarea lui T2 (și invers);
- dacă T1 și T2 sunt termeni compuși, unificarea reușește numai dacă au același functor, aceeași aritate (număr de argumente), iar toate argumentele corespunzătoare se unifică recursiv; în caz contrar, unificarea eșuează.
- dacă T1 și T2 sunt **variabile neinstanțiate,** ei pot fi unificați. În acest caz variabilele rămân **neinstanțiate**, dar legate una de alta, până cel puțin una din ele nu va fi instanțiată și atunci, și cealaltă devine automat instanțiată cu același termen.

```
?- ana = ana.
true.

?- ana = ion.
false.
```

```
?- ana = Ana.
Ana= ana
```

```
?- 15 = 15.0.
false.

?- X = ana.
X = ana.



?- punct(X, Y) = punct(3, 4).
X = 3,
Y = 4.

?- punct(X, 4) = punct(3, Y).
```

```
X = 3,
Y = 4.

?- f(X) = g(X).
false.
```

Când interogăm *parinte(ion, X)*., Prolog parcurge baza de cunoștințe și încearcă să unifice acest termen cu fiecare fapt *parinte(...)* existent. Prima potrivire leagă X la *maria;* la o cerere de soluție suplimentară, Prolog **marchează poziția** predicatului din baza de fapte care a returnat prima soluție, eliberează variabila X de valoarea instanțată (anulează legarea) (backtracking) și căutarea continuă de la punctul unde sa oprit căutarea precedentă, sistemul încercând să unifice următorul fapt. Marcajul permite Prologului să recunoască partea din baza de fapte ce nu a fost încă examinată într-un anumit context. Reluând parcurgerea mai jos de faptul marcat, marcajul se **suprimă** și Prologul caută un nou fapt cu care scopul ar putea fi unificat

💡 *Unificarea este bidirecțională și simetrică: X = ana este echivalent cu ana = X. Aceasta o diferențiază fundamental de atribuirea (=) din limbajele imperative.*

💡 ***Atenție**: Spre deosebiră de atribuirea în Python/Java unde X = 5 modifică valoarea lui X, în Prolog X = 5 este o condiție de succes. Dacă X este deja legat la 3, interogarea X = 5 va eșua imediat, fără a modifica valoarea anterioară.*

6.9. Recursivitate

Deoarece Prolog nu are structuri explicite de tip buclă (for, while), recursivitatea este mecanismul standard pentru a exprima procese repetitive și relații de tip "tranzitiv" (de exemplu, strămoș, descendent).

O definiție recursivă are, de regulă, două părți: cazul de bază (condiția de oprire) și cazul recursiv (care reduce problema la o instanță mai mică).

```
% Relatia "stramos" — inchiderea tranzitiva a relatiei "parinte"
stramos(X, Y) :- parinte(X, Y).              % caz de baza
stramos(X, Y) :- parinte(X, Z), stramos(Z, Y).   % caz recursiv

?- stramos(ion, elena).
true.

?- stramos(ion, victor).
true.
```

Un exemplu clasic de recursivitate aritmetică — calculul factorialului:

```
factorial(0, 1) :- !.
factorial(N, F) :-
   N > 0,
   N1 is N - 1,
   factorial(N1, F1),
   F is N * F1.

?- factorial(5, F).
F = 120.
```

💡 *Semnul ! din primul fapt este predicatul cut, pe care îl vom explica continuare. El previne căutarea inutilă a altor soluții pentru cazul de bază.*

Șirul lui Fibonacci este un alt exemplu tipic:

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
   N > 1,
   N1 is N - 1,
   N2 is N - 2,
   fib(N1, F1),
   fib(N2, F2),
   F is F1 + F2.
```

Exemplul Fibonacci prezentat are o complexitate temporală exponențială deoarece reîcalculează aceleași sub-probleme de mai multe ori. În sistemele de producție, se preferă utilizarea „memoizării” sau a unor metode iterative.”

6.10. Structuri de date: liste

Lista este structura de date fundamentală în Prolog, folosită pentru a reprezenta secvențe de elemente de lungime variabilă. Sintactic, o listă se scrie între paranteze pătrate, cu elementele separate prin virgulă:

```
?- L = [mar, para, banana].
L = [mar, para, banana].

?- L = [].          % lista vida
L = [].
```

Intern, orice listă neevidă este reprezentată ca o pereche (cap, coadă), notată [Cap|Coada], unde Cap este primul element, iar Coada este restul listei (tot o listă):

```
?- [Cap|Coada] = [mar, para, banana].
Cap = mar,
Coada = [para, banana].

?- [A, B|Rest] = [1, 2, 3, 4, 5].
A = 1,
B = 2,
Rest = [3, 4, 5].
```

Predicatele pe liste se definesc, în mod natural, recursiv: cazul de bază pentru lista vidă, iar cazul recursiv pentru [Cap|Coada].

```
% Lungimea unei liste
lungime([], 0).
lungime([_|Coada], N) :-
   lungime(Coada, N1),
   N is N1 + 1.

?- lungime([a, b, c, d], N).
N = 4.
```

```
% Apartenenta unui element la o lista
membru(X, [X|_]).
membru(X, [_|Coada]) :- membru(X, Coada).

?- membru(banana, [mar, para, banana]).
true.
```

```
% Concatenarea a doua liste
concat([], L, L).
concat([X|Coada], L2, [X|Rezultat]) :-
   concat(Coada, L2, Rezultat).

?- concat([1, 2], [3, 4], R).
R = [1, 2, 3, 4].
```

💡 *Predicatul membru/2 și concat/2 (numit append/3 în implementările standard, precum SWI-Prolog) sunt printre cele mai des utilizate predicate predefinite pentru liste.*

6.11. Backtracking

Backtracking-ul este mecanismul automat prin care motorul Prolog, atunci când un obiectiv curent eșuează (sau când i se cere explicit o soluție suplimentară), revine la ultimul punct de decizie unde exista o alternativă neexplorată și încearcă acea alternativă.

Practic, Prolog construiește un arbore de căutare: fiecare nod reprezintă o stare a demonstrației, iar ramurile reprezintă clauzele alternative care s-ar putea potrivi cu obiectivul curent. Căutarea se face în adâncime (depth-first), de sus în jos, în ordinea în care clauzele apar în program.

```
culoare(rosu).
culoare(verde).
culoare(albastru).

forma(cerc).
forma(patrat).

combinatie(C, F) :- culoare(C), forma(F).

?- combinatie(C, F).
C = rosu, F = cerc ;
C = rosu, F = patrat ;
C = verde, F = cerc ;
C = verde, F = patrat ;
C = albastru, F = cerc ;
C = albastru, F = patrat.
```

La fiecare ";" tastat de utilizator, Prolog anulează ultima legare (culoare) și, dacă nu mai are alternative pentru forma, revine (backtrack) la predicatul culoare pentru a încerca următoarea valoare.

💡 La satisfacerea unei **conjuncții de subscopuri**, se încearcă satisfacerea fiecărui subscop pe rând, **de la stânga la dreapta**. Prima satisfacere a unui subscop determină plasarea unui marcaj în dreptul faptului care a determinat satisfacerea subscopului. Dacă un subscop nu poate fi satisfăcut, adică eșuează, sistemul se întoarce și încearcă resatisfacerea subscopului cel mai aproape **din stânga**, pornind căutarea în bază **de la marcaj în jos**. Înainte de resatisfacerea unui subscop, se **elimină** toate instanțierile de variabile determinate de ultima satisfacere a acestuia.

Pentru a înțelege backtracking-ul, utilizăm instrumentul **trace.** Comanda **trace**. înainte de o interogare permite utilizatorului să vadă pas cu pas cum Prolog explorează ramurile arborelui de căutare, unde se produce unificarea și unde apare eșecul care declanșează backtracking-ul."

💡 *Backtracking-ul este atât o sursă de putere expresivă (permite găsirea automată a tuturor soluțiilor), cât și o sursă de ineficiență, dacă nu este controlat — de aici necesitatea predicatelor fail și cut, discutate în continuare.*

6.12. Predicatele fail și cut (!)

Predicatul fail

fail este un predicat predefinit care **eșuează** întotdeauna, forțând motorul Prolog să declanșeze backtracking. Este util pentru a parcurge sistematic toate soluțiile unei relații, de exemplu pentru afișarea lor cu write/1.

```
afiseaza_toti_copiii(X) :-
    parinte(X, Y),
    write(Y), nl,
    fail.
afiseaza_toti_copiii(_).          % clauza de "iesire" dupa epuizarea solutiilor

?- afiseaza_toti_copiii(ion).
maria
andrei
true.
```

Explicație: după afișarea primei soluții (maria), fail forțează backtracking-ul spre parinte(ion, Y), care găsește următoarea soluție (andrei). Când nu mai există soluții, Prolog trece la a doua clauză afiseaza_toti_copiii(_), care reușește trivial, oprind procesul.

Predicatul cut (!)

Operatorul ! ("cut") este un predicat special care **reușește** întotdeauna, dar are un efect colateral: elimină ("taie") toate punctele de backtracking create de la intrarea în clauza curentă, inclusiv alegerea altor clauze alternative pentru același predicat. Odată trecut de !, Prolog nu mai poate reveni înaintea lui.

```
% Fara cut — ambele clauze raman disponibile la backtracking
clasifica(N, negativ) :- N < 0.
clasifica(N, zero) :- N =:= 0.
clasifica(N, pozitiv) :- N > 0.
```

Folosit corect, cut face codul mai eficient, eliminând testele redundante odată ce o alternativă corectă a fost identificată:

```
% green cut
clasifica(N, negativ) :- N < 0, !.
```

```
clasifica(N, zero) :- N =:= 0, !.
clasifica(_, pozitiv).

?- clasifica(-5, R).
R = negativ.
```

Distincția clasică între tipurile de cut:

- **cut transparent** ("green cut") – elimină doar alternative redundante, fără a schimba mulțimea soluțiilor corecte (ca în exemplul de mai sus);
- **cut opac** ("red cut") – schimbă efectiv semantica logică a programului, eliminând soluții care ar fi fost corecte fără el; folosirea sa trebuie documentată cu atenție.

```
% meniu(Zi, Mâncare) cu red cut

meniu(marti, pizza) :- !.
meniu(joi, pizza) :- !.
meniu(_, supa).

?- meniu(Zi, pizza).
Zi = marti
false

% Soluția validă: Zi = joi a fost tăiată și pierdută definitiv!
```

💡 *Regulă practică: cut-ul trebuie folosit cu moderație și numai atunci când logica relației permite tăierea în siguranță a alternativelor; abuzul de cut face programele Prolog greu de întreținut și de depanat.*

6.13. Modificarea dinamică a programului

Prologul permite modificarea dinamică a bazei de fapte și reguli. În acest scop se folosesc predicate predefinite de căutare, adăugare și eliminare a clauzelor. Aceste predicate se numesc de **ordinul doi**, deoarece au în calitate de argumente alte predicate. Odată modificate, predicatele-argumente își pierd forma inițială și își păstrează forma nouă până nu sunt modificate din nou. Dar programele ce conțin predicate de adăugare și eliminare a clauzelor pot avea efecte colaterale. Aceste predicate pot cauza schimbări ce pot afecta executarea de mai departe a programului. Scopurile, ce reușesc acum într-un punct, mai târziu pot eșua sau pot reuși în diferite momente de timp.

- **Căutarea unei clauze**. Predicatul predefinit este: ***clause(Cap,Corp).***
- **Adăugarea unei clauze**. Predicatele predefinite sunt: ***assert(clauza). asserta(clauza). assertz(clauza).*** Dacă se utilizează **assert/1**, poziția clauzei noi în program nu este specificată. Clauza va fi prima în pachet, dacă se folosește **asserta/1** și ultima, dacă **assertz/1**.
- **Eliminarea unei clauze**. Predicatul predefinit este: ***retract(clauza).***

Întrebări pentru autocontrol

1. Care este diferența fundamentală dintre paradigma programării logice și cea imperativă?
2. Din ce elemente fundamentale este format un program Prolog și ce reprezintă fiecare?
3. Cum se deosebește o variabilă în Prolog de o variabilă dintr-un limbaj imperativ (de exemplu, C, Java, Python)?

4. Ce este unificarea și care sunt regulile principale după care Prolog stabilește dacă doi termeni unifică?
5. Care este rolul predicatului IS prin ce diferă operatorii de comparație aritmetică de operatorul de unificare (=)?
6. De ce este necesară recursivitatea în Prolog și care sunt cele două componente principale ale unei definiții recursive?
7. Cum sunt reprezentate intern listele în Prolog și cum funcționează notația [Cap|Coada]?
8. Cum funcționează mecanismul de backtracking și ce strategie de căutare folosește motorul Prolog?
9. Ce rol are predicatul fail și în ce situații este utilizat?
10. Ce efect are predicatul cut (!) asupra backtracking-ului și care este diferența dintre un *green cut* și un *red cut*?

CAPITOLUL 7. SISTEME EXPERT ȘI SISTEME SUPORT PENTRU DECIZII

7.1. Contextul suportului decizional în organizații

Deciziile manageriale contemporane se iau într-un mediu economic caracterizat prin volume mari de date, incertitudine, presiunea timpului și complexitate crescândă a factorilor implicați (piață, concurență, reglementări, risc financiar). Managerii nu mai pot baza deciziile exclusiv pe intuiție sau experiență personală – este necesar suportul unor instrumente informatice capabile să proceseze informația, să aplice cunoștințe specializate și să genereze recomandări fundamentate.

Domeniul Inteligenței Artificiale a oferit, începând cu anii 1970-1980, două direcții complementare de răspuns la această nevoie:

- **Sistemele expert (SE),** orientate spre emularea raționamentului unui expert uman într-un domeniu îngust, dar bine definit;
- **Sistemele suport pentru decizii (SSD),** orientate spre combinarea datelor, modelelor și cunoștințelor pentru a sprijini – nu a înlocui – decidentul uman în probleme semi-structurate.

> 💡 **De reținut**
>
> Ambele categorii de sisteme fac parte din familia sistemelor informaționale bazate pe cunoștințe (Knowledge-Based Systems) și au ca scop comun creșterea calității și rapidității procesului decizional, nu înlocuirea integrală a factorului uman.

7.2. Sisteme suport pentru decizii (SSD)

Un **Sistem Suport pentru Decizii** (SSD, în engleză Decision Support System – DSS) este un sistem informatic interactiv, bazat pe calculator, destinat să asiste decidenții în rezolvarea problemelor semi-structurate sau nestructurate, prin combinarea datelor, a modelelor analitice și a raționamentului uman.

Conceptul a apărut la începutul anilor 1970 (Gorry și Scott Morton, 1971) ca alternativă la sistemele informaționale de gestiune clasice (MIS), care ofereau rapoarte standardizate, dar nu sprijineau explorarea interactivă a variantelor decizionale. Evoluția SSD a parcurs mai multe etape:

- ***SSD orientate pe modele*** – utilizarea unor modele financiare și de simulare relativ simple;
- ***SSD orientate pe date*** – accent pe interogarea și analiza volumelor mari de date, precursori ai sistemelor Business Intelligence;
- ***SSD de grup*** (GDSS) – sprijin pentru decizii colective, colaborative;
- ***SSD inteligente*** – integrarea tehnicilor de inteligență artificială, inclusiv elemente de sisteme expert și, mai recent, de învățare automată.

Principalele **caracteristici definitorii** ale SSD sunt:

- **Semi-structurare:** SSD se adresează problemelor care nu pot fi complet automatizate (spre deosebire de problemele structurate), dar nici nu pot fi lăsate exclusiv pe seama intuiției.
- **Interactivitate:** utilizatorul dialoghează cu sistemul, testează ipoteze de tip "ce se întâmplă dacă..." (analiza what-if).
- **Flexibilitate:** sistemul se adaptează la stilul decizional al utilizatorului și la evoluția problemei.
- **Orientare spre sprijin, nu spre înlocuire:** decizia finală și responsabilitatea rămân la decident.
- **Integrare de date și modele:** combină date interne (contabile, operaționale) cu date externe (piață, concurență) și modele cantitative.

Arhitectura unui sistem suport pentru decizii

Din punct de vedere structural, un SSD este alcătuit, în mod clasic, din trei subsisteme principale, coordonate printr-o **interfață de dialog**:

- **Baza de date,** care stochează datele relevante pentru problema decizională – date interne (financiar-contabile, operaționale, resurse umane) și date externe (statistici de piață, indicatori macroeconomici, informații despre concurență);
- **Baza de modele,** care conține modele cantitative – statistice, financiare, de optimizare, de simulare – utilizate pentru a transforma datele în informații decizionale;
- **Sistemul de gestiune a dialogului,** componenta de interfață ce permite decidentului să interogheze baza de date, să selecteze și să parametrizeze modelele și să vizualizeze rezultatele sub formă de rapoarte și grafice.

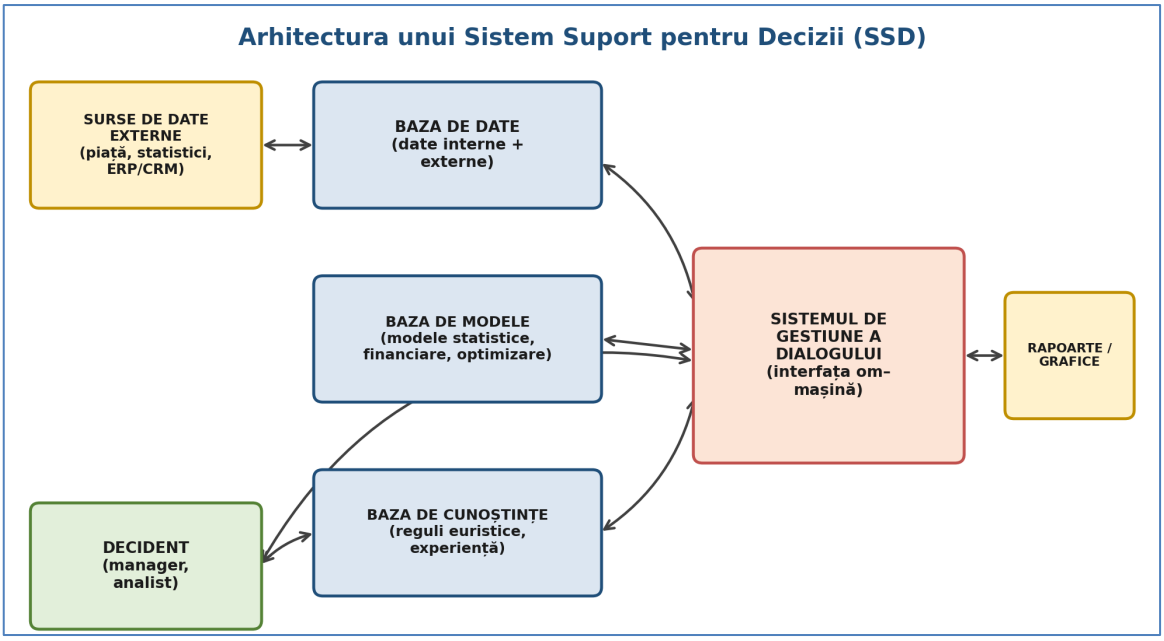


Figura 7.1 Arhitectura unui Sistem Suport pentru Decizii (SSD)

Această arhitectură pune în evidență natura profund interactivă a SSD: decidentul nu primește o singură soluție "gata calculată", ci explorează, prin intermediul sistemului de gestiune a dialogului, mai multe scenarii posibile, comparând rezultatele generate de baza de modele pe baza datelor disponibile.

7.3. Sisteme expert (SE)

Un **Sistem Expert** (SE, în engleză Expert System) este un program informatic care încorporează cunoștințele și capacitatea de raționament ale unui expert uman dintr-un domeniu specializat, fiind capabil să rezolve probleme complexe și să ofere recomandări sau diagnostice la un nivel comparabil cu cel al expertului respectiv, inclusiv explicații privind raționamentul urmat.

Sistemele expert reprezintă una dintre primele aplicații de succes comercial ale Inteligenței Artificiale, dezvoltate începând cu anii 1960-1970 la universități precum Stanford. Printre exemplele clasice, adesea citate în literatura de specialitate, se numără:

- **DENDRAL** (sfârșitul anilor 1960) – identificarea structurii moleculelor chimice pe baza datelor spectrometrice;
- **MYCIN** (anii 1970) – diagnosticarea infecțiilor bacteriene și recomandarea tratamentului antibiotic;
- **XCON/R1** (anii 1980, compania Digital Equipment Corporation) – configurarea automată a sistemelor de calcul, unul dintre primele SE cu impact economic direct și major.

Caracteristici definitorii ale sistemelor expert sunt:

- **Specializare îngustă:** SE performează foarte bine într-un domeniu restrâns și bine delimitat (spre deosebire de inteligența generală umană).
- **Separarea cunoștințelor de mecanismul de raționament:** cunoștințele specifice domeniului sunt stocate distinct de motorul care le prelucrează, ceea ce facilitează actualizarea lor.
- **Capacitate explicativă:** sistemul poate justifica o concluzie, răspunzând la întrebări de tipul "de ce?" și "cum?".
- **Raționament sub incertitudine:** multe SE utilizează factori de certitudine sau logică probabilistă/fuzzy pentru a gestiona informații incomplete sau ambigue.
- **Performanță apropiată de cea a expertului uman:** în domeniul restrâns pentru care a fost construit sistemul.
- **Limitarea domeniului (Brittleness):** Spre deosebire de expertul uman, sistemele expertice pot fi 'fragile'. Ele funcționează impecabil în interiorul parametrilor definiți, dar pot produce concluzii eronate sau hilare în cazul în care primesc date care ies din sfera domeniului lor de competență.
- **Sistemele expert versus Machine Learning**: Sistemele Expertice sunt de tip 'Glass Box' (transparente, bazate pe reguli scrise de om, excelente pentru audit), iar modelele de Machine Learning sunt adesea 'Black Box' (învață modele matematice complexe din date, excelente pentru identificarea tiparelor în volume masive de date unde regulile nu sunt evidente).

Arhitectura sistemelor expert

Arhitectura unui sistem expert este mai complexă decât cea a unui SSD generic, fiind organizată în jurul separării nete între cunoștințe și mecanismul de raționament. Componentele principale sunt prezentate în continuare.

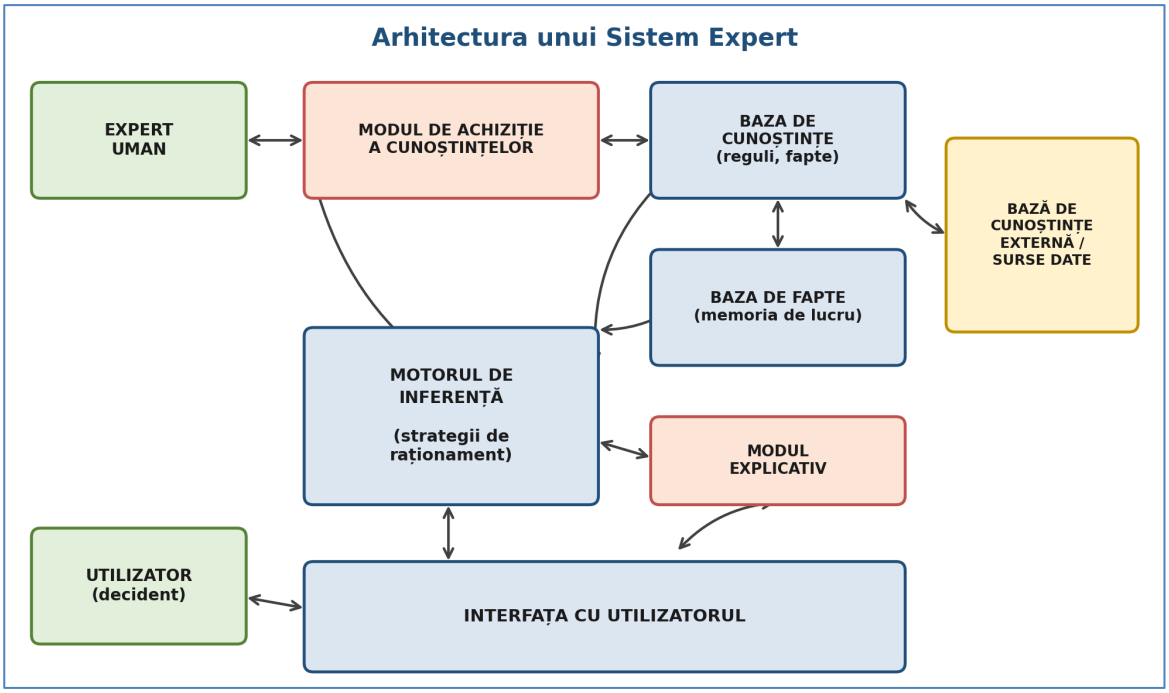


Figura 7.2 Arhitectura unui Sistem Expert

Baza de cunoștințe (Knowledge Base) reprezintă "nucleul" sistemului expert și reprezintă o colecție structurată de cunoștințe specifice domeniului, extrase de la unul sau mai mulți experți umani prin procesul de achiziție a cunoștințelor. Aceste cunoștințe sunt exprimate, cel mai frecvent, sub formă de reguli de producție de tipul DACĂ(condiție) ATUNCI (concluzie), dar pot fi reprezentate și prin cadre, rețele semantice sau arbori de decizie.

Exemplu simplificat de regulă, dintr-un domeniu financiar:

Exemplu simplificat de regulă de producție din domeniu financiar:

DACĂ lichiditatea curentă a întreprinderii este sub 1,0 ȘI gradul de îndatorare depășește 65% ATUNCI riscul de credit este ridicat ȘI se recomandă analiza suplimentară a garanțiilor.

Baza de fapte, numită și memoria de lucru (working memory), conține datele și faptele specifice cazului curent supus analizei - de exemplu, valorile concrete ale indicatorilor financiari ai unei întreprinderi pentru care se solicită o evaluare. Spre deosebire de baza de cunoștințe, care este relativ stabilă, conținutul bazei de fapte se modifică pe parcursul sesiunii de consultare, pe măsură ce sistemul deduce noi fapte.

Motorul de inferență (Inference Engine) este componenta responsabilă de aplicarea regulilor din baza de cunoștințe asupra faptelor din baza de fapte, pentru a genera noi concluzii. Este, în esență, un program generic, independent de domeniu, care implementează o **strategie de raționament**. Cele două strategii clasice de înlănțuire a regulilor sunt prezentate în Figura 7.3.

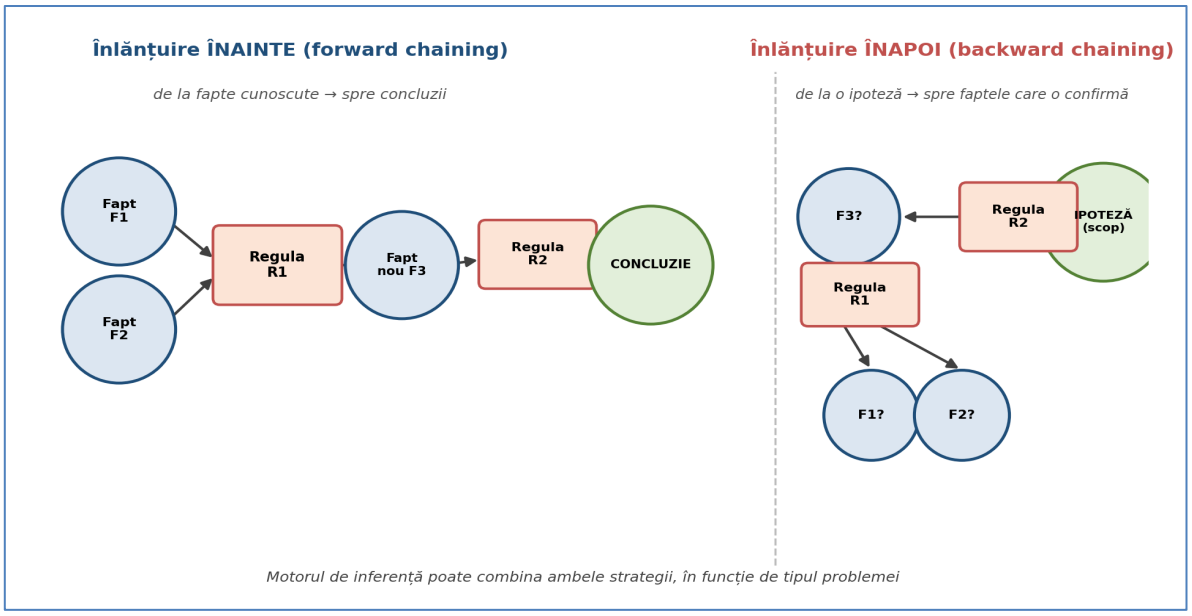


Figura 7.3 Strategii de raționament ale motorului de inferență

- **Înlănțuirea înainte (forward chaining):** raționamentul pornește de la faptele cunoscute și aplică regulile disponibile pentru a genera noi fapte, până la obținerea unei concluzii; este eficientă atunci când există multe date de intrare și se caută toate concluziile posibile (de exemplu, monitorizarea continuă a unui sistem).
- **Înlănțuirea înapoi (backward chaining):** raționamentul pornește de la o ipoteză (scop) și caută, în mod recursiv, faptele sau sub-obiectivele care ar confirma-o; este eficientă atunci când se testează o ipoteză specifică (de exemplu, diagnosticare orientată pe o problemă suspectată).

Motorul de inferență include, de asemenea, un mecanism de rezolvare a conflictelor, necesar atunci când mai multe reguli devin aplicabile simultan (de exemplu, pe baza priorității, a specificității sau a ordinii regulilor în baza de cunoștințe).

Modulul explicativ permite sistemului să justifice raționamentul urmat, oferind răspunsuri la întrebările "de ce?" (ce regulă a condus la o anumită întrebare) și "cum?" (ce lanț de reguli a condus la o concluzie). Această componentă este esențială pentru acceptarea sistemului de către utilizatorii profesioniști, care au nevoie să înțeleagă și să poată verifica recomandarea primită, nu doar să o accepte necondiționat.

Modulul de achiziție a cunoștințelor facilitează procesul, adesea dificil, de extragere a cunoștințelor de la experți umani și de introducere/actualizare a acestora în baza de cunoștințe, fără a fi necesară intervenția directă asupra codului sursă al sistemului. În sistemele expert moderne, acest modul poate include, de asemenea, componente de învățare automată pentru extragerea de reguli din date istorice.

Interfața cu utilizatorul asigură dialogul dintre sistem și utilizatorul final: colectează informațiile necesare (prin întrebări succesive), prezintă concluziile și recomandările, și oferă acces la explicațiile generate de modulul explicativ. Calitatea interfeței influențează decisiv gradul de adoptare a sistemului expert în practică.

7.4. Metode de reprezentare a cunoștințelor

Eficiența unui sistem expert depinde în mare măsură de modul în care sunt reprezentate cunoștințele din domeniu. Principalele metode utilizate sunt:

- **Regulile de producție (DACĂ...ATUNCI...):** cea mai răspândită metodă, datorită apropierii de modul natural de exprimare a cunoștințelor euristice și a ușurinței de a fi verificate și modificate independent unele de altele.

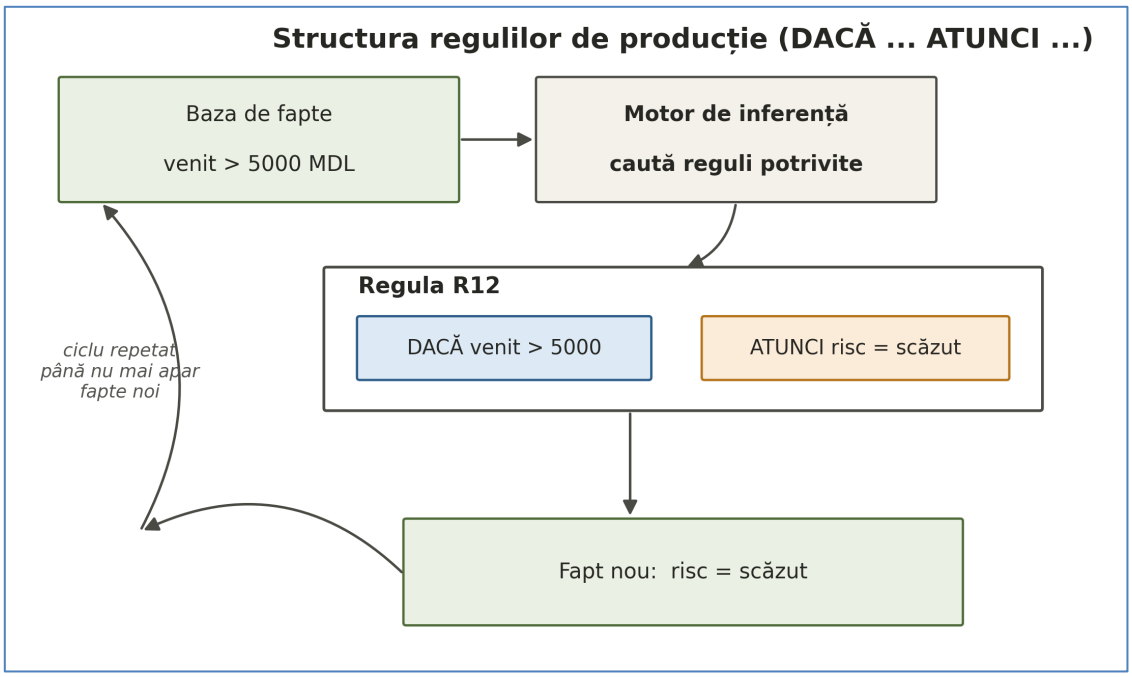


Figura 7.4 Metoda regulilor de producțiet

- **Cadrele (frames):** structuri care grupează, în jurul unui concept sau obiect, un set de atribute (sloturi) și valori asociate, permițând moștenirea proprietăților între concepte înrudite (de exemplu, un cadru "Client corporativ" care moștenește atribute de la cadrul general "Client").

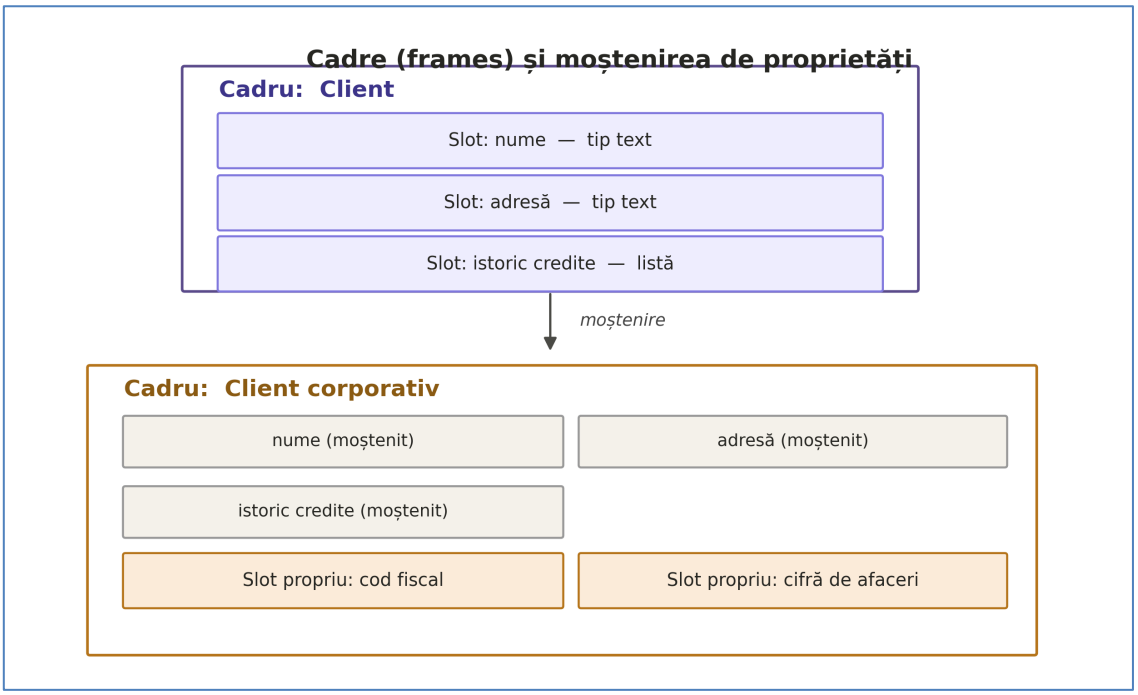


Figura 7.5 Metoda cadrelor

- **Rețelele semantice:** reprezentări grafice sub formă de noduri (concepte) și arce (relații între concepte), utile pentru modelarea relațiilor ierarhice și asociative dintre entitățile unui domeniu.

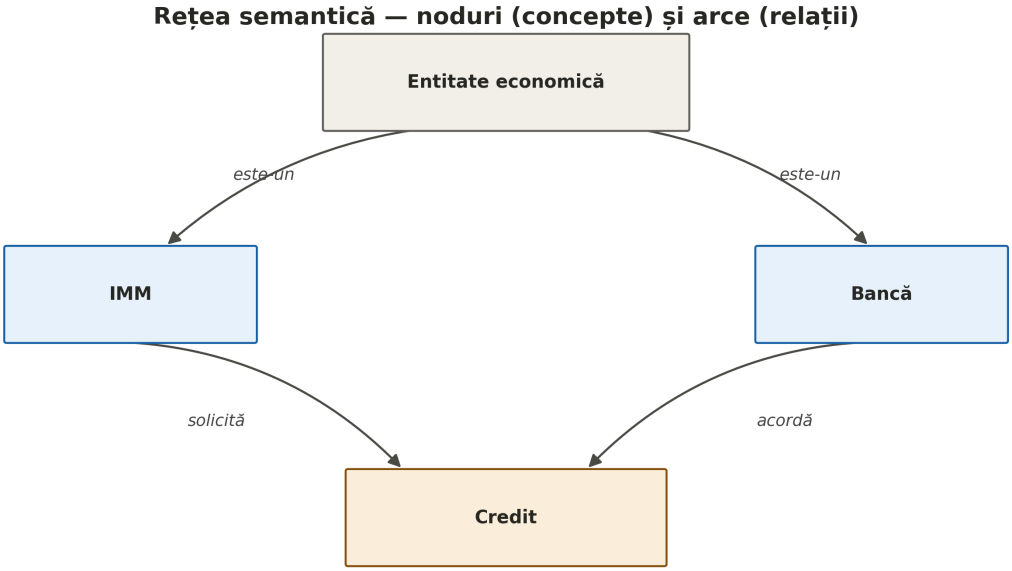


Figura 7.6 Metoda rețelelor semantice

- **Arborii și tabelele de decizie:** reprezentări structurate, potrivite pentru probleme de clasificare cu un număr finit de alternative, frecvent utilizate în evaluarea riscului de credit sau în scoring.

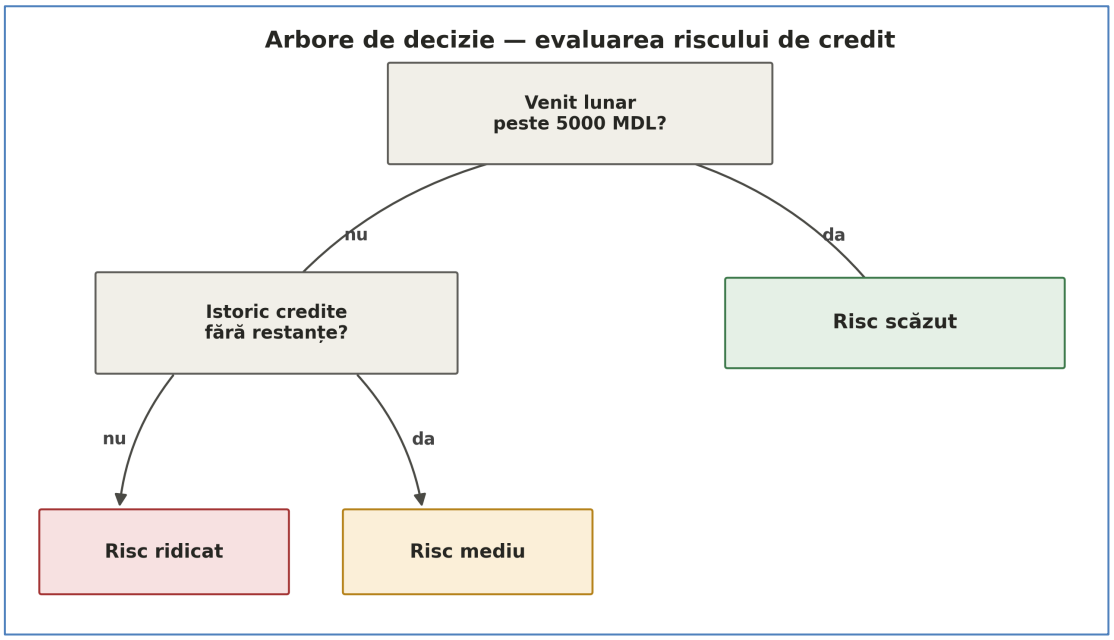


Figura 7.7 Metoda arborilor de decizie

- **Logica formală și raționamentul probabilistic:** utilizate atunci când este necesară gestionarea riguroasă a incertitudinii (de exemplu, rețele Bayesiene, factori de certitudine, logică fuzzy).

💡 Alegerea metodei de reprezentare nu este pur tehnică - ea influențează direct capacitatea explicativă a sistemului, ușurința de actualizare a cunoștințelor și performanța motorului de inferență. În practică, sistemele expert complexe combină mai multe metode de reprezentare.

7.5. Sisteme Expert versus Sisteme Suport pentru Decizii

Deși ambele categorii de sisteme urmăresc îmbunătățirea procesului decizional, ele diferă în mod semnificativ prin natura problemelor abordate, rolul utilizatorului și componenta centrală a arhitecturii, conform sintezei din tabelul de mai jos.

Tabelul 7.1. **Sisteme expert vs. sisteme suport pentru decizii**

Criteriu	Sistem expert (SE)	Sistem suport pentru decizii (SSD)
Scop principal	Emularea raționamentului unui expert uman pentru a oferi o soluție/recomandare	Sprijinirea decidentului cu informații, analize și modele
Rolul utilizatorului	Relativ pasiv – primește o concluzie sau un diagnostic	Activ – explorează scenarii, interoghează modele, compară variante
Sursa cunoașterii	Cunoștințe euristice, reguli extrase de la experți umani	Date și modele cantitative (statistice, financiare, de optimizare)
Tip de probleme	Structurate, cu reguli identificabile (diagnosticare, clasificare)	Semi-structurate sau nestructurate (planificare, strategie)
Componentă centrală	Motorul de inferență	Baza de modele
Capacitate explicativă	Ridicată – justifică raționamentul ("de ce?", "cum?")	Variabilă – depinde de tipul modelului utilizat

În practică, cele două tipuri de sisteme tind să convergă: numeroase soluții informatice moderne integrează module de tip sistem expert (reguli de decizie, alerte automate) în cadrul unor platforme SSD/Business Intelligence mai ample, formând așa-numitele sisteme hibride sau sisteme inteligente de suport decizional.

7.6. Aplicații în mediul economic

Sistemele expert și sistemele suport pentru decizii au cunoscut o largă adoptare în mediul economic, datorită capacității lor de a reduce timpul de analiză, a standardiza deciziile repetitive și a capta cunoștințele specializate ale angajaților cu experiență. Figura 7.8 sintetizează principalele domenii de aplicare.

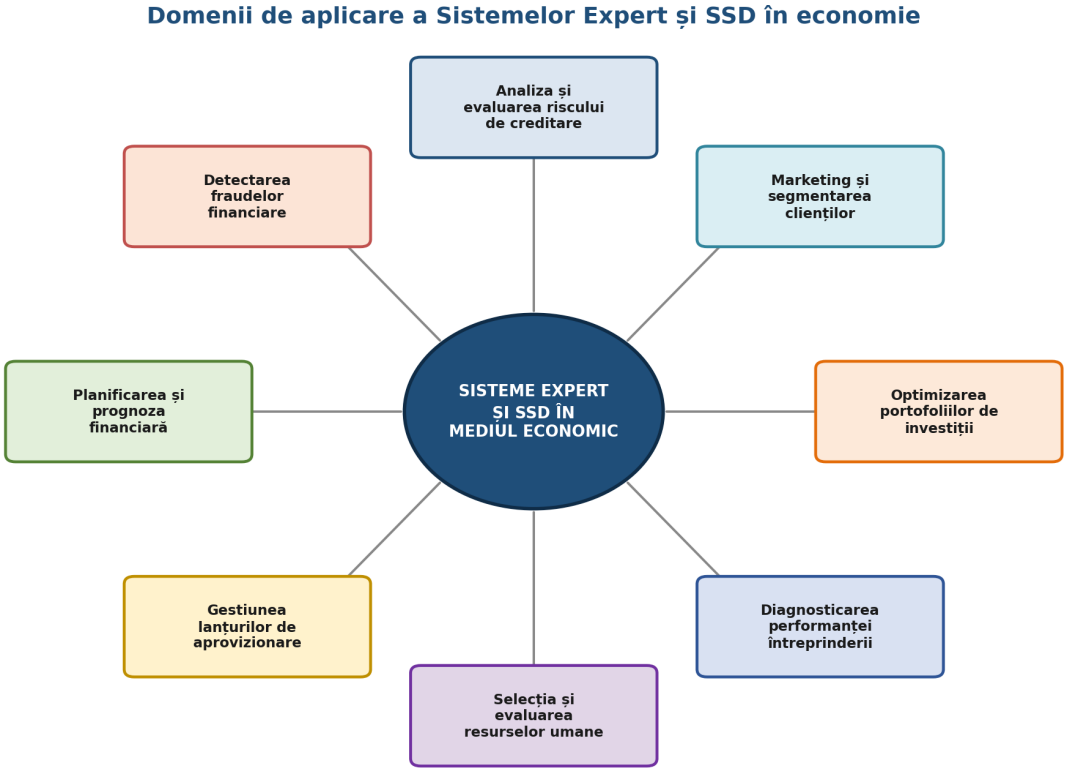


Figura 7.8 Domenii de aplicare a sistemelor expert și SSD în economie

- **Analiza și evaluarea riscului de creditare:** băncile și instituțiile financiare utilizează sisteme expert pentru scoring de credit, aplicând reguli euristice și modele statistice asupra indicatorilor financiari ai solicitanților, pentru a estima probabilitatea de nerambursare.
- **Detectarea fraudelor financiare:** sisteme care combină reguli euristice (tranzacții atipice, valori-prag) cu modele statistice pentru semnalarea automată a tranzacțiilor suspecte în timp real.
- **Planificarea și prognoza financiară:** SSD utilizate pentru bugetare, analiza cash-flow și simulări de tip "ce se întâmplă dacă" asupra unor scenarii macroeconomice diferite.
- **Gestiunea lanțurilor de aprovizionare:** sisteme de optimizare a stocurilor, a rutelor de distribuție și a planificării producției, bazate pe modele de cercetare operațională.
- **Selecția și evaluarea resurselor umane:** sisteme expert utilizate pentru preselecția candidaților sau evaluarea performanței, pe baza unor criterii și reguli predefinite.
- **Diagnosticarea performanței întreprinderii:** analiza indicatorilor economico-financiari (lichiditate, solvabilitate, rentabilitate) și generarea automată a unui diagnostic sintetic, însoțit de recomandări.
- **Optimizarea portofoliilor de investiții:** SSD care combină modele de tip Markowitz sau alte metode de optimizare cu preferințele de risc ale investitorului.
- **Marketing și segmentarea clienților:** sisteme care sprijină identificarea segmentelor de clienți și personalizarea ofertelor, pe baza unor reguli de comportament sau a modelelor predictive.

Studiu de caz ilustrativ: evaluarea riscului de creditare

Considerăm o bancă comercială care dorește să automatizeze o parte a procesului de evaluare preliminară a cererilor de credit pentru întreprinderi mici și mijlocii. Un sistem expert dedicat acestei sarcini ar putea funcționa astfel:

1. Utilizatorul (ofițerul de credit) introduce, prin interfață, indicatorii financiari ai întreprinderii solicitante (lichiditate, grad de îndatorare, rentabilitate, vechime în activitate);
2. Aceste valori sunt introduse în baza de fapte;
3. Motorul de inferență aplică regulile din baza de cunoștințe, construite pe baza experienței analiștilor de credit senior, folosind, de regulă, înlănțuirea înainte, pornind de la datele disponibile;
4. Sistemul generează o concluzie preliminară privind nivelul de risc (scăzut/mediu/ridicat), însoțită de o explicație a raționamentului (modulul explicativ);
5. Decizia finală de acordare a creditului rămâne la comitetul de credit, care poate consulta atât recomandarea sistemului expert, cât și analizele suplimentare furnizate de un SSD financiar (simulări de rambursare, analiza de sensibilitate).

Acest exemplu ilustrează caracterul complementar al celor două categorii de sisteme: sistemul expert automatizează o evaluare de tip diagnostic, structurată și bazată pe reguli, în timp ce SSD sprijină explorarea interactivă a scenariilor financiare asociate deciziei finale. . În realitatea bancară actuală, un sistem expert nu lucrează singur (Hybrid Intelligence); el este adesea alimentat de un model de ML care identifică anomalii (scoping), iar sistemul expert aplică apoi regulile de conformitate legală (decisioning).

Idei-cheie de reținut:

- Sistemele expert și sistemele suport pentru decizii sunt două categorii distincte, dar complementare, de sisteme informaționale bazate pe cunoștințe, dezvoltate pentru a îmbunătăți calitatea procesului decizional.
- Arhitectura unui sistem expert se bazează pe separarea netă între baza de cunoștințe, baza de fapte, motorul de inferență, modulul explicativ, modulul de achiziție a cunoștințelor și interfața cu utilizatorul.
- Motorul de inferență poate utiliza două strategii principale de raționament: înlănțuirea înainte (de la fapte spre concluzii) și înlănțuirea înapoi (de la o ipoteză spre faptele care o confirmă).
- Un SSD se organizează, în mod clasic, în jurul a trei componente: baza de date, baza de modele și sistemul de gestiune a dialogului.
- Cunoștințele pot fi reprezentate prin reguli de producție, cadre, rețele semantice, arbori de decizie sau metode probabilistice, fiecare cu avantaje specifice.
- În practica modernă, sistemele expert și SSD tind să se integreze în soluții hibride, alături de tehnici de învățare automată și analiză a datelor (Business Intelligence, Analytics).

Întrebări de autoevaluare

1. Care este diferența fundamentală dintre un Sistem Expert (SE) și un Sistem Suport pentru Decizii (SSD) în ceea ce privește scopul lor principal față de decident?
2. Care sunt principalele componente ale arhitecturii unui sistem expert și ce rol are fiecare?
3. Explicați diferența dintre "Baza de cunoștințe" și "Baza de fapte" (memoria de lucru) în cadrul unui sistem expert.
4. Descrieți cele trei componente principale ale arhitecturii unui SSD și explicați rolul sistemului de gestiune a dialogului.
5. Ce este un motor de inferență și de ce este considerat un program "generic, independent de domeniu"?
6. Ce diferență există între înlănțuirea înainte și înlănțuirea înapoi în cadrul motorului de inferență?
7. Explicați mecanismul de tip "Dacă... Atunci" specific regulilor de producție.
8. Enumerați și descrieți pe scurt trei metode de reprezentare a cunoștințelor.
9. Din punctul de vedere al sursei cunoașterii, prin ce diferă un SE de un SSD clasic?
10. Explicați distincția "Glass Box" vs. "Black Box" atunci când comparăm Sistemele Expert cu modelele de Machine Learning.

CAPITOLUL 8. RAȚIONAMENT ÎN CONDIȚII DE INCERTITUDINE

8.1. Incertitudinea și raționamentul

Sistemele expert clasice și logica propozițională de tip adevărat/fals funcționează bine în domenii formale, bine definite, unde fiecare afirmație are o valoare de adevăr clară. În lumea reală însă, agenții inteligenți – fie ei sisteme de diagnoză medicală, filtre de spam, roboți mobili sau sisteme de recomandare – trebuie să acționeze pe baza unor informații incomplete, ambigue sau contradictorii.

Raționamentul în condiții de incertitudine este ramura inteligenței artificiale care oferă instrumente formale pentru a reprezenta și a manipula gradul de încredere pe care un sistem îl are într-o afirmație, atunci când nu se poate stabili cu certitudine absolută dacă aceasta este adevărată sau falsă.

Incertitudine vs. imprecizie

Incertitudinea se referă la lipsa de certitudine privind adevărul unei afirmații (ex.: „este posibil ca pacientul să aibă gripă"). **Imprecizia (vag)** se referă la faptul că un concept nu are granițe clare (ex.: „temperatură ridicată"). Logica fuzzy tratează în special imprecizia, iar teoria probabilităților tratează în special incertitudinea.

Sursele incertitudinii

Incertitudinea într-un sistem inteligent poate proveni din mai multe surse, adesea concomitente:

- **Date lipsă (informație incompletă):** senzorii nu pot măsura toate variabilele relevante, iar utilizatorii nu furnizează întotdeauna toate detaliile necesare unei decizii.
- **Zgomotul în date:** erorile de măsurare, perturbațiile senzorilor sau erorile de transmisie introduc variații aleatorii care distorsionează valorile reale.
- **Ambiguitatea și imprecizia limbajului:** termeni precum „cald", „mult", „rapid" sau „riscant" nu au o definiție unică, universal acceptată.
- **Fiabilitatea variabilă a surselor:** diferite surse de informație (senzori, experți, baze de date) au grade diferite de încredere.
- **Cunoștințe incomplete ale domeniului:** chiar și experții umani nu dispun de un model complet și exact al fenomenului studiat.
- **Caracterul intrinsec probabilistic al unor fenomene:** unele evenimente sunt aleatorii prin natura lor (ex.: defectarea unei componente electronice).

8.2. Logica fuzzy

Mulțimi fuzzy și grade de apartenență

Logica fuzzy (logica vagă), introdusă de Lotfi Zadeh în 1965, generalizează teoria clasică a mulțimilor. Într-o mulțime clasică, un element fie aparține mulțimii (apartenență = 1), fie nu aparține (apartenență = 0). Într-o mulțime fuzzy, un element poate aparține parțial, cu un grad de apartenență $\mu(x) \in [0, 1]$.

Funcție de apartenență

O funcție de apartenență $\mu A(x)$ asociază fiecărui element x din universul de discurs un grad de apartenență la mulțimea fuzzy A, cuprins între 0 (deloc membru) și 1 (membru total). Formele cele mai utilizate sunt triunghiulare, trapezoidale, gaussiene sau sigmoidale.

Exemplu: variabila lingvistică „Temperatură”

Considerăm variabila lingvistică „Temperatură”, definită prin trei mulțimi fuzzy: *Rece, Moderată și Caldă*. Fiecare valoare numerică a temperaturii poate aparține simultan, în grade diferite, mai multor mulțimi – spre deosebire de o clasificare clasică (rece / nu-e-rece), unde apartenența ar fi binară.

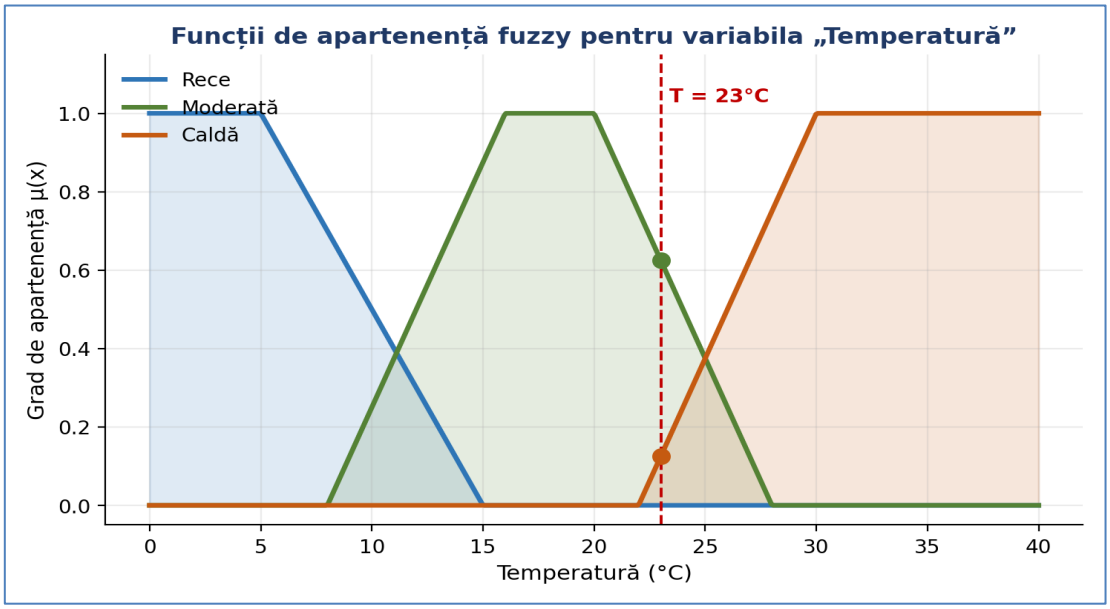


Figura 8.1 Funcții de apartenență fuzzy pentru variabila „Temperatură”. La T = 23 °C, temperatura este simultan „moderată” (μ ≈ 0,6) și „caldă” (μ ≈ 0,1).

Această suprapunere a mulțimilor este esența logicii fuzzy: un sistem poate raționa cu afirmații de tipul „temperatura este mai degrabă moderată, dar tinde spre caldă”, mult mai apropiate de modul de gândire uman decât o clasificare rigidă.

Forme uzuale ale funcțiilor de apartenență

În practică, funcțiile de apartenență nu se definesc punct cu punct, ci se aleg dintr-un set restrâns de forme parametrizate, ușor de calculat și de interpretat de către un expert al domeniului. Cele patru forme standard – triunghiulară, trapezoidală, gaussiană și sigmoidală – acoperă majoritatea aplicațiilor practice ale logicii fuzzy.

Funcția **triunghiulară** este cea mai simplă și mai des folosită formă, definită prin trei puncte a, b, c (a = piciorul stâng, b = vârful, unde μ = 1, c = piciorul drept):

$$\mu_A(x) = \begin{cases} 0, & x \le a \\ \frac{x-a}{b-a}, & a < x \le b \\ \frac{c-x}{c-b}, & b < x \le c \\ 0, & x > c \end{cases} \qquad (1)$$

Funcția de apartenență triunghiulară, cu parametrii $a < b < c$

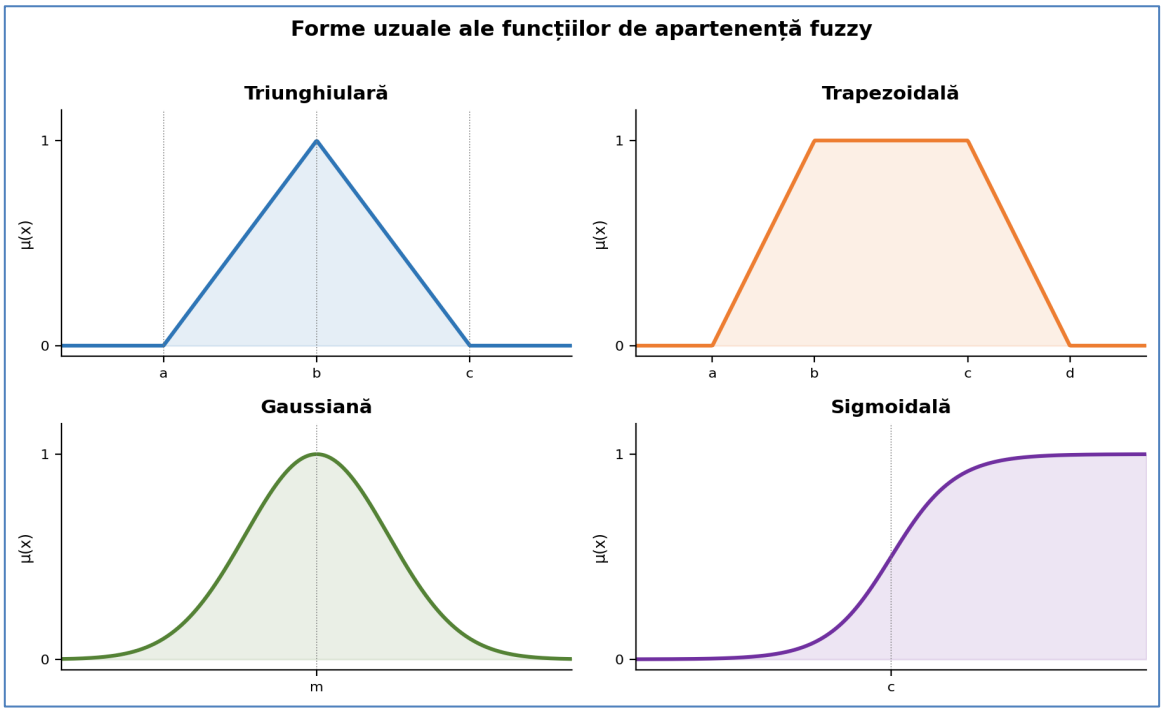


Figura 8.2 Schema comună a formelor de apartenență fuzzy folosite în practică: triunghiulară, trapezoidală, gaussiană și sigmoidală. Parametrii (a, b, c, d, m, σ) controlează poziția și lățimea fiecărei forme.

Funcția **trapezoidală** generalizează forma triunghiulară printr-un platou de apartenență maximă (μ = 1) între punctele b și c, util pentru concepte cu un interval „tipic" mai larg, nu doar un singur punct de vârf:

$$\mu_A(x) = \begin{cases} 0, & x \le a \\ \frac{x-a}{b-a}, & a < x \le b \\ 1, & b < x \le c \\ \frac{d-x}{d-c}, & c < x \le d \\ 0, & x > d \end{cases} \quad (2)$$

Funcția de apartenență trapezoidală, cu parametrii $a < b < c < d$

Funcția **gaussiană** oferă o tranziție netedă (fără colțuri), fiind preferată atunci când se dorește o comportare derivabilă a sistemului fuzzy; este definită de centrul m și lățimea σ:

$$\mu_A(x) = e^{-(x-m)^2 / 2\sigma^2} \quad (3)$$

Funcția de apartenență gaussiană, cu centrul m și lățimea σ

Funcția **sigmoidală** este utilă pentru concepte care exprimă un prag crescător sau descrescător (ex.: „venit mare", „risc scăzut"), fiind definită de panta a și punctul de inflexiune c:

$$\mu_A(x) = \frac{1}{1 + e^{-a(x-c)}} \quad (4)$$

Funcția de apartenență sigmoidală, cu panta a și punctul de inflexiune c

Defuzzificarea

Un **sistem fuzzy de tip Mamdani** parcurge, pentru fiecare intrare, patru etape: (1) **fuzzificarea** – transformarea valorilor numerice de intrare în grade de apartenență μ(x), folosind funcțiile descrise

mai sus; (2) **evaluarea regulilor** – calculul gradului de activare al fiecărei reguli, prin operatorii ȘI/SAU fuzzy (min/max); (3) **agregarea** – combinarea rezultatelor tuturor regulilor într-o singură mulțime fuzzy de ieșire; și (4) **defuzzificarea** – transformarea acestei mulțimi fuzzy de ieșire într-o singură valoare numerică (crisp), utilizabilă direct de sistemul de control sau de decizie.

Defuzzificarea este necesară deoarece, în cele din urmă, un actuator (un motor, o valvă, un ventilator) sau o decizie economică (un scor de risc, un preț) are nevoie de o singură valoare numerică, nu de un grad de apartenență distribuit pe un interval. Cea mai utilizată metodă este **centrul de greutate** (***centroid***), care calculează abscisa centrului de masă al suprafeței de sub funcția de apartenență agregată μ(z):

$$z* = \frac{\int \mu(z) \cdot z \, dz}{\int \mu(z) \, dz} \tag{5}$$

Defuzzificarea prin metoda centrului de greutate (centroid); z este valoarea numerică finală*

În practică, integrala se aproximează printr-o sumă discretă pe un set finit de puncte reprezentative z_1, z_2, …, z_n (de exemplu, valorile de vârf ale mulțimilor fuzzy de ieșire implicate), ceea ce conduce la o formă simplificată – **media ponderată** – larg folosită în controlere fuzzy și în sistemele de tip Sugeno:

$$z* = \frac{\mu(z_1)\cdot z_1 + \mu(z_2)\cdot z_2 + \cdots + \mu(z_n)\cdot z_n}{\mu(z_1) + \mu(z_2) + \cdots + \mu(z_n)} \tag{6}$$

Defuzzificarea prin media ponderată — formă discretă, simplificată, a centroidului

Alte metode uzuale de defuzzificare sunt **bisectorul** (dreapta care împarte suprafața de sub μ(z) în două arii egale) și **media maximelor** (MOM – Mean of Maximum), care ia media punctelor z pentru care μ(z) este maximă; acestea sunt mai rapide de calculat, dar mai puțin stabile decât centroidul atunci când mulțimea agregată are mai multe vârfuri.

Exemplu simplu de regulă fuzzy: controlul unui ventilator în funcție de temperatură

Reluăm variabila „Temperatură” din Figura 8.1 și adăugăm o bază de reguli simplă, de tip Mamdani cu o **singură intrare**, pentru a stabili viteza unui ventilator (0–100%). Pentru simplitate, mulțimile de ieșire sunt reprezentate prin valori prototip (singleton), ca în sistemele Sugeno:

- **R1: DACĂ Temperatura este Rece ATUNCI** Viteza este Mică (prototip 20%)
- **R2: DACĂ Temperatura este Moderată ATUNCI** Viteza este Medie (prototip 50%)
- **R3: DACĂ Temperatura este Caldă ATUNCI** Viteza este Mare (prototip 90%)

La T = 23 °C (același exemplu numeric ca în Figura 8.1), gradele de apartenență calculate erau μ_Rece(23) = 0, μ_Moderată(23) ≈ 0,6 și μ_Caldă(23) ≈ 0,1. Aceste grade devin gradele de activare ale regulilor R1–R3 (fiecare regulă are un singur antecedent, deci nu este nevoie de operatorul MIN). Tabelul de mai jos aplică defuzzificarea prin **media ponderată** (formula 6):

Regulă	μ_i (activare)	z_i prototip (%)	$\mu_i \cdot z_i$
R1: DACĂ T = Rece ATUNCI Viteza = Mică	0	20	0
R2: DACĂ T = Moderată ATUNCI Viteza = Medie	0,6	50	30
R3: DACĂ T = Caldă ATUNCI Viteza = Mare	0,1	90	9
Σ (sumă)	**0,7**	**—**	**39**

$$z* = \frac{0 \cdot 20 + 0{,}6 \cdot 50 + 0{,}1 \cdot 90}{0 + 0{,}6 + 0{,}1} = \frac{39}{0{,}7} \approx 55{,}7\ \% \quad (7)$$

Calculul valorii crisp finale prin media ponderată a prototipurilor de ieșire

Rezultatul defuzzificării, z* ≈ 55,7%, este comanda numerică unică trimisă motorului ventilatorului – deși nicio regulă nu a fost complet adevărată sau falsă, sistemul fuzzy a combinat contribuțiile parțiale ale regulilor R2 (dominantă, cu activare 0,6) și R3 (marginală, cu activare 0,1) într-o decizie coerentă, graduală, mult mai apropiată de modul în care ar regla un operator uman ventilatorul decât o comandă de tip ON/OFF.

Operatori fuzzy de bază

Operatorii logici clasici (ȘI, SAU, NU) au echivalente fuzzy, definite de Zadeh prin funcțiile minim și maxim:

$$\mu_{A \cup B}(x) = max[\mu_A(x), \mu_B(x)] \quad (8)$$

Reuniunea fuzzy (operatorul SAU) – gradul de apartenență la A∪B este maximul celor două grade

$$\mu_{A \cap B}(x) = min[\mu_A(x), \mu_B(x)] \quad (9)$$

Intersecția fuzzy (operatorul ȘI) – gradul de apartenență la A∩B este minimul celor două grade

Aceste reguli permit construirea de sisteme expert fuzzy: reguli de forma „DACĂ temperatura este Caldă ȘI umiditatea este Ridicată ATUNCI riscul de disconfort este Mare”. Gradul de activare al fiecărei reguli (rezultat din ȘI sau SAU) este apoi folosit ca pondere în defuzzificarea prin media ponderată, exact ca în formula (6) : $z^* = \Sigma\, \mu_i \cdot z_i / \Sigma\, \mu_i$, unde z_i este valoarea prototip a ieșirii asociate regulii i

Aplicații tipice ale logicii fuzzy

- controlere fuzzy pentru electrocasnice (mașini de spălat, aparate de aer condiționat);
- sisteme de control automat în industrie (frâne ABS, sisteme de climatizare auto);
- sisteme de suport decizional cu criterii lingvistice imprecise.

Studiu de caz 1 – Climatizare: Temperatură ȘI/SAU Umiditate → Puterea de răcire

Variabilele fuzzy de intrare

Sistemul are două intrări – Temperatură (15–35 °C) cu termenii Rece, Moderată, Caldă, și Umiditate (0–100%) cu termenii Scăzută, Medie, Ridicată – și o ieșire, Puterea de răcire a compresorului (0–100%), reprezentată prin trei valori prototip: Mică (25%), Medie (55%), Mare (90%).

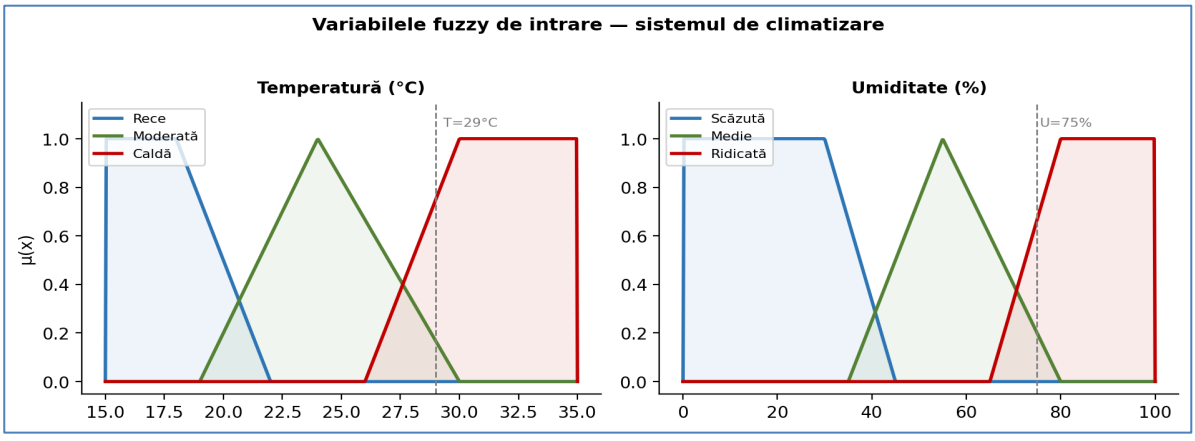


Figura A. Funcțiile de apartenență pentru Temperatură și Umiditate. Liniile punctate marchează valorile curente din exemplul numeric: T = 29 °C, U = 75%.

Baza de reguli (ȘI / SAU)

- **R1: DACĂ Temperatura este Caldă ȘI Umiditatea este Ridicată ATUNCI** Puterea este Mare
- **R2: DACĂ Temperatura este Caldă SAU Umiditatea este Ridicată ATUNCI** Puterea este Medie
- **R3: DACĂ Temperatura este Moderată ȘI Umiditatea este Medie ATUNCI** Puterea este Medie
- **R4: DACĂ Temperatura este Rece SAU Umiditatea este Scăzută ATUNCI** Puterea este Mică

Regulile R1 și R2 ilustrează diferența dintre operatori pe același perechi de termeni: R1 cere ca ambele condiții să fie adevărate simultan (ȘI → minim, mai restrictivă), în timp ce R2 se activează dacă cel puțin una dintre condiții este adevărată (SAU → maxim, mai permisivă) – de aceea R2 conduce la o putere mai mică (Medie) decât R1 (Mare), fiind o regulă de „siguranță" pentru cazurile parțial critice.

Exemplu numeric

Pentru T = 29 °C și U = 75%, gradele de apartenență citite din Figura A sunt: μ_Rece = 0, μ_Moderată ≈ 0,167, μ_Caldă = 0,75, μ_Scăzută = 0, μ_Medie = 0,2, μ_Ridicată ≈ 0,667. Tabelul de mai jos aplică operatorii ȘI/SAU pentru fiecare regulă:

Regulă	Operator și calcul	μ_i	z_i (%)
R1: Caldă ȘI Ridicată → Mare	min(0,75; 0,667)	0,667	90
R2: Caldă SAU Ridicată → Medie	max(0,75; 0,667)	0,75	55
R3: Moderată ȘI Medie → Medie	min(0,167; 0,2)	0,167	55
R4: Rece SAU Scăzută → Mică	max(0; 0)	0	25

$$z* = \frac{0{,}667 \cdot 90 + 0{,}75 \cdot 55 + 0{,}167 \cdot 55 + 0 \cdot 25}{0{,}667 + 0{,}75 + 0{,}167 + 0} = \frac{110{,}42}{1{,}583} \approx 69{,}7\ \% \quad (3)$$

Puterea de răcire rezultată prin defuzzificare (media ponderată)

Rezultatul, z* ≈ 69,7%, reflectă corect intuiția: la 29 °C și 75% umiditate, aerul condiționat trebuie să funcționeze puternic (aproape de nivelul „Mare"), dar nu la maxim, deoarece regula R2 (mai permisivă, prin SAU) temperează ușor contribuția regulii R1 (mai stricte, prin ȘI).

Studiu de caz 2 – Mașina de spălat rufe: Tip țesătură ȘI/SAU Grad de murdărie → Temperatură, Timp, Volum de apă

Variabilele fuzzy de intrare

Sistemul are două intrări – Gradul de murdărie (0–100%), cu termenii Ușoară, Medie, Puternică, și Rezistența țesăturii (0–10, unde 0 = mătase/dantelă foarte delicată, 10 = bumbac gros/denim), cu termenii Delicată, Normală, Rezistentă – și trei ieșiri controlate simultan de aceleași reguli: Temperatura apei (Rece 20 °C, Călduță 40 °C, Fierbinte 60 °C), Timpul de spălare (Scurt 30 min, Mediu 60 min, Lung 90 min) și Volumul de apă (Mic 30 L, Mediu 45 L, Mare 60 L).

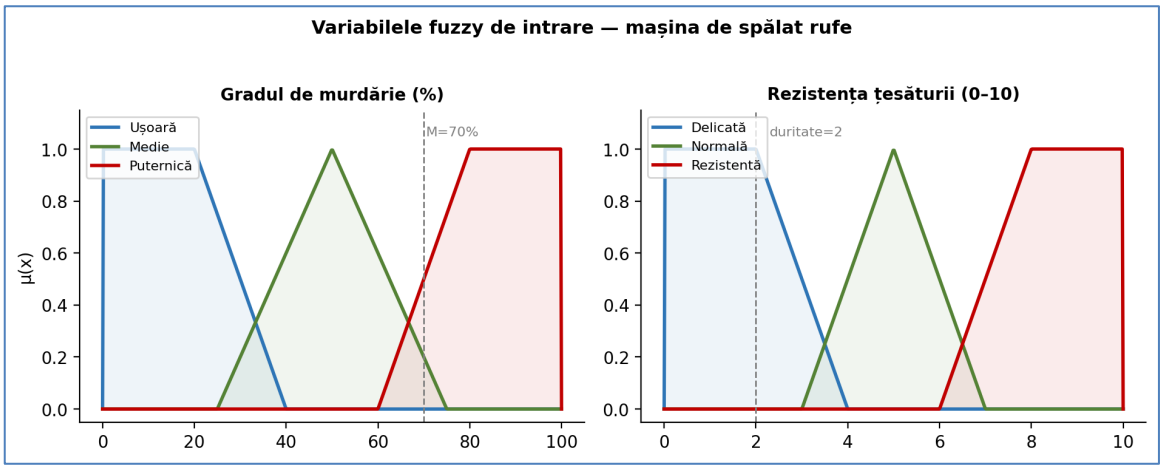


Figura B. Funcțiile de apartenență pentru Gradul de murdărie și Rezistența țesăturii. Liniile punctate marchează valorile din exemplul numeric: murdărie = 70%, rezistență țesătură = 2 (rufe delicate).

Baza de reguli (fiecare regulă stabilește simultan cele trei ieșiri)

- **R1: DACĂ Murdărie este Puternică ȘI Țesătura este Rezistentă ATUNCI** Temperatură = Fierbinte, Timp = Lung, Volum apă = Mare
- **R2: DACĂ Murdărie este Puternică ȘI Țesătura este Delicată ATUNCI** Temperatură = Călduță, Timp = Lung, Volum apă = Mediu
- **R3: DACĂ Murdărie este Medie SAU Țesătura este Normală ATUNCI** Temperatură = Călduță, Timp = Mediu, Volum apă = Mediu
- **R4: DACĂ Murdărie este Ușoară ȘI Țesătura este Delicată ATUNCI** Temperatură = Rece, Timp = Scurt, Volum apă = Mic

Regulile R1 și R2 folosesc ambele operatorul ȘI (minim), dar cu concluzii diferite: chiar dacă murdăria este puternică în ambele cazuri, țesătura delicată din R2 limitează temperatura și volumul de apă, pentru a proteja rufele – un exemplu tipic în care aceeași intensitate a unei condiții este moderată de cealaltă variabilă. Regula R3 folosește SAU: este suficient ca oricare dintre cei doi factori (murdărie medie sau țesătură normală) să fie prezent pentru un program standard.

Exemplu numeric

Pentru murdărie = 70% și rezistență țesătură = 2 (rufe delicate), gradele de apartenență citite din Figura B sunt: μ_Ușoară = 0, μ_Medie = 0,2, μ_Puternică = 0,5, μ_Delicată = 1, μ_Normală = 0, μ_Rezistentă = 0. Evaluarea regulilor:

Regulă	Operator și calcul	μ_i	Se activează?
R1: Puternică ȘI Rezistentă	min(0,5; 0)	0	Nu
R2: Puternică ȘI Delicată	min(0,5; 1)	0,5	Da
R3: Medie SAU Normală	max(0,2; 0)	0,2	Da
R4: Ușoară ȘI Delicată	min(0; 1)	0	Nu

Doar R2 (μ = 0,5) și R3 (μ = 0,2) se activează; R1 și R4 au grad de activare 0, deci nu contribuie la rezultat. **Defuzzificarea** prin media ponderată se aplică **separat** pentru fiecare din cele trei ieșiri:

$$Temperatură* = \frac{0{,}5 \cdot 40 + 0{,}2 \cdot 40}{0{,}5 + 0{,}2} = 40\,°C \quad \text{(A)}$$

Temperatura apei rezultată – ambele reguli active indică „Călduță", deci rezultatul este neambiguu

$$Timp* = \frac{0{,}5 \cdot 90 + 0{,}2 \cdot 60}{0{,}5 + 0{,}2} \approx 81{,}4\,\text{min} \quad \text{(B)}$$

Timpul de spălare rezultat – combină „Lung" (R2) cu „Mediu" (R3), ponderat cu gradele de activare

$$Volum* = \frac{0{,}5 \cdot 45 + 0{,}2 \cdot 45}{0{,}5 + 0{,}2} = 45\,L \quad \text{(C)}$$

Volumul de apă rezultat – ambele reguli active indică „Mediu"

Rezultatul final al programului fuzzy este: apă călduță (40 °C), spălare de aproximativ 81 de minute și volum mediu de apă (45 L). Se observă cum operatorul ȘI din R2 a „protejat" țesătura delicată în ciuda gradului mare de murdărie (temperatura și volumul nu au urcat la valorile maxime din R1), în timp ce operatorul SAU din R3 a adus o contribuție moderatoare asupra timpului de spălare, întrucât murdăria medie (0,2) a fost suficientă pentru a activa parțial regula, independent de tipul țesăturii.

8.3. Factori de certitudine

Factorii de certitudine (Certainty Factors, CF) au fost introduși în anii 1970 în cadrul sistemului expert medical MYCIN, destinat diagnosticării infecțiilor bacteriene. La acea vreme, teoria probabilităților complete (Bayesiană) era considerată prea costisitoare computațional și prea greu de estimat de către experții umani, astfel încât cercetătorii au propus o măsură euristică, mai simplă, a gradului de încredere.

Factorul de certitudine – definiție

Un factor de certitudine CF(H, E) este un număr din intervalul [−1, 1] care exprimă gradul de încredere într-o ipoteză H, dat fiind un set de dovezi E:

- CF = +1 → certitudine absolută că H este adevărată;
- CF = −1 → certitudine absolută că H este falsă;
- CF = 0 → lipsă totală de informație relevantă despre H.

Combinarea factorilor de certitudine

Într-un sistem expert bazat pe reguli, o ipoteză H poate fi susținută de mai multe reguli / dovezi independente. Factorii de certitudine individuali se combină după o formulă specifică, astfel încât rezultatul rămâne în intervalul [−1, 1] și reflectă întărirea reciprocă a dovezilor concordante:

$$CF(H, E_1 \wedge E_2) = CF(H, E_1) + CF(H, E_2) \cdot [1 - CF(H, E_1)] \quad (3)$$

Formula de combinare a doi factori de certitudine pozitivi (concordanți), aplicată în MYCIN

Observăm că formula are un efect de „saturare": pe măsură ce se acumulează dovezi concordante, factorul de certitudine combinat se apropie de 1, dar nu îl depășește niciodată – reflectând ideea intuitivă că certitudinea crește, dar niciodată nu poate depăși maximul absolut.

Exemplu practic – combinarea a două simptome într-un sistem expert medical

Presupunem că regula R1 („febră mare” → „infecție bacteriană”) produce CF(H,E1) = 0,6, iar regula R2 („leucocite crescute” → „infecție bacteriană”) produce CF(H,E2) = 0,4. Aplicând formula de combinare:

$$CF(H, E_1 \wedge E_2) = 0{,}6 + 0{,}4 \cdot (1 - 0{,}6) = 0{,}6 + 0{,}16 = 0{,}76$$

Certitudinea combinată privind diagnosticul de infecție bacteriană crește la 0,76, reflectând faptul că cele două dovezi se susțin reciproc, fără însă a atinge certitudinea absolută.

Calculul factorilor de certitudine pentru premise complexe și rezolvarea conflictelor

La soluționarea unei probleme, motorul de inferență operează adesea cu fapte incomplete și informații incerte. Această incertitudine poate fi prezentă atât în fapte, cât și în regulile de producție individuale sau poate rezulta din incompatibilitatea regulilor și procesul de rezolvare a conflictelor.

În practică, premisele regulilor nu sunt întotdeauna simple, ci pot fi formate din mai multe condiții conectate logic. Incertitudinea se manifestă și în combinarea acestor premise, iar factorul de certitudine (CF) al unei premise se calculează în funcție de operatorii logici folosiți:

- **Operatorul AND:** Când condițiile din premisă sunt conectate cu AND, factorul de certitudine al premisei este minimumul dintre factorii de certitudine ai condițiilor. De exemplu, pentru IF E1 AND E2 AND E3 THEN G, calculul este CF(premisă) = min[CF(E1); CF(E2); CF(E3)].
- **Operatorul OR:** Când condițiile sunt conectate cu OR, factorul de certitudine al premisei este maximumul dintre factorii de certitudine ai condițiilor. Pentru o regulă precum IF E1 OR E2 OR E3 THEN G, se va aplica CF(premisă) = max[CF(E1); CF(E2); CF(E3)].
- **Operatorul NOT:** Când premisa conține o condiție negată, factorul de certitudine capătă semn opus, de exemplu CF(NOT E1) = -CF(E1).

Odată stabilit factorul premisei, factorul de certitudine final al regulii se obține prin înmulțirea CF al premisei cu CF al concluziei: CF(regulă) = CF(premisă) * CF(concluzie). Aceste reguli bazate pe factori de certitudine se vor declanșa numai dacă este depășit un anumit prag stabilit.

Exemplu practic — calcularea pragului de declanșare a regulii

Fie CF(E1)=0.6; CF(E2)=0.7; CF(E3)=0.5; CF(E4)=0.4; CF(G)=0.9

R3: *IF (E1 AND E2) OR (E3 AND NOT E4) THEN G*

CF(premisă)=max[min[CF(E1);CF(E2)];min[CF(E3);-CF(E4)]]=

= max[min[0.6;0.7];min[0.5;-0.4)]]=0.6

*CF(regulă)=CF(premisă)*CF(G)= 0.6*0.9=0.54*

Strategii pentru soluționarea conflictelor I

Incertitudinea există și în lanțurile inferențiale. În cazul în care premisele a două sau mai multor reguli sunt satisfăcute simultan, apare un conflict între reguli, iar motorul de inferență trebuie să decidă ordinea de execuție. Pentru a gestiona aceste conflicte, se pot aplica mai multe strategii:

- **Specificitatea:** O regulă de producție va fi declanșată prima dacă are o specificitate mai mare, adică dacă conține un număr mai mare de condiții în premisă.

- **Prioritățile implicite prin factori de certitudine:** Factorii de certitudine atașați regulilor pot fi utilizați pentru a stabili o prioritate. De exemplu, dintre două reguli valide, va fi executată prima regula care are în final factorul de certitudine calculat mai mare.
- **Alte metode implicite:** Se poate ține cont de ordinea în care regulile au fost introduse în baza de reguli, poziția lor în bază sau se poate prioritiza cea mai recentă regulă introdusă.

Limitele modelului CF

Deși util și intuitiv, modelul factorilor de certitudine nu are o fundamentare probabilistică riguroasă: combinarea repetată a unor dovezi care nu sunt cu adevărat independente poate duce la valori artificial de mari ale certitudinii. Din acest motiv, comunitatea IA s-a orientat treptat spre modele probabiliste riguroase – în special spre rețelele Bayesiene, subiectul central al acestui capitol.

8.4. Probabilitatea condiționată și regula lui Bayes

Probabilitatea condiționată P(A|B) exprimă probabilitatea ca evenimentul A să se producă, știind că evenimentul B s-a produs deja (P(B) > 0). Aceasta este noțiunea fundamentală care permite actualizarea gradului de încredere pe măsură ce sosesc informații (dovezi) noi. Probabilitatea ca A să se producă, știind că B s-a produs deja, este:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \tag{1}$$

Aceasta e chiar definiția — nu se demonstrează, e punctul de plecare.

Pasul 1: Scriem aceeași relație și în sens invers

Simetric, probabilitatea ca B să se producă, știind că A s-a produs, este:

$$P(B \mid A) = \frac{P(A \cap B)}{P(A)} \tag{2}$$

Pasul 2: Izolăm intersecția P(A ∩ B) din ambele relații

Din prima ecuație: $P(A \cap B) = P(A \mid B) \cdot P(B)$

Din a doua ecuație: $P(A \cap B) = P(B \mid A) \cdot P(A)$

Pasul 3: Egalăm cele două expresii

Deoarece P(A ∩ B) este același eveniment (intersecția este comutativă — A∩B = B∩A), cele două expresii trebuie să fie egale:

$$P(A \mid B) \cdot P(B) = P(B \mid A) \cdot P(A)$$

Pasul 4: Rezolvăm pentru P(A|B)

Împărțim ambele părți la P(B):

$$\boxed{P(A \mid B) = \frac{P(B|A) \cdot P(A)}{P(B)}} \tag{3}$$

Aceasta este **legea lui Bayes**.

Regula lui Bayes

Regula lui Bayes, derivată direct din definiția probabilității condiționate, este formula centrală a raționamentului probabilistic. Ea permite „inversarea" unei probabilități condiționate: pornind de la P(dovadă | ipoteză) – de regulă mai ușor de estimat – se obține P(ipoteză | dovadă), adică exact ceea ce ne interesează.

$$P(H \mid E) = \frac{P(E|H) \cdot P(H)}{P(E)} \tag{4}$$

Regula lui Bayes – forma fundamentală. H = ipoteza, E = dovada (evidența) observată

Termenii formulei au denumiri și interpretări specifice, esențiale pentru înțelegerea raționamentului bayesian:

Tabel 8.1 Termenii regulei lui Bayes

Termen	Interpretare
P(H)	Probabilitatea a priori a ipotezei H – gradul de încredere înainte de a observa dovada
P(E|H)	Verosimilitatea (likelihood) – probabilitatea de a observa dovada E dacă H este adevărată
P(E)	Probabilitatea marginală a dovezii – un factor de normalizare
P(H|E)	Probabilitatea a posteriori – gradul de încredere actualizat, după observarea dovezii

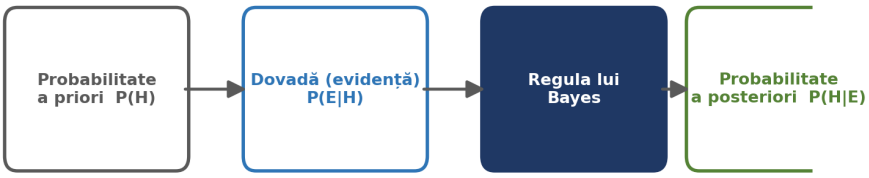


Figura 8.3 Actualizarea bayesiană: pornind de la o probabilitate a priori, o nouă dovadă produce probabilitatea a posteriori

Forma extinsă (probabilitate totală)

Atunci când ipoteza H are exact două stări posibile (H și non-H), probabilitatea marginală P(E) din numitor poate fi descompusă folosind **legea probabilității totale**, obținând o formă complet calculabilă a regulii lui Bayes:

$$P(H \mid E) = \frac{P(E|H) \cdot P(H)}{P(E|H) \cdot P(H) + P(E|\neg H) \cdot P(\neg H)} \tag{5}$$

Regula lui Bayes – forma extinsă, utilă atunci când P(E) nu este cunoscută direct

Exemplu practic – test medical – actualizarea probabilității pe baza unui rezultat pozitiv

Un test de diagnosticare pentru o boală rară are o sensibilitate de 99% (P(pozitiv|bolnav)=0,99) și o rată de fals-pozitiv de 5% (P(pozitiv|sănătos)=0,05). Prevalența bolii în populație este de 1% (P(bolnav)=0,01). Care este probabilitatea ca o persoană cu test pozitiv să fie într-adevăr bolnavă?

Aplicând forma extinsă a regulii lui Bayes:

$$P(bolnav \mid pozitiv) = (0{,}99 \times 0{,}01) / (0{,}99 \times 0{,}01 + 0{,}05 \times 0{,}99) = 0{,}0099 / 0{,}0594 \approx 0{,}167$$

Deși testul pare foarte precis (99% sensibilitate), probabilitatea reală ca pacientul să fie bolnav, după un rezultat pozitiv, este de doar aproximativ 16,7% – deoarece boala este rară, iar numărul de fals-pozitivi din populația sănătoasă (mult mai numeroasă) depășește numărul de adevărați pozitivi. Acesta este un exemplu clasic care ilustrează importanța probabilității a priori și pericolul „neglijării ratei de bază" (base-rate neglect).

Forma șanselor (odds form)

O variantă utilă în practică, mai ales pentru actualizări succesive cu dovezi multiple, este forma șanselor (odds) a regulii lui Bayes, care exprimă raportul dintre probabilitatea ipotezei H și probabilitatea complementului ei:

$$\frac{P(H|E)}{P(\neg H|E)} = \frac{P(E|H)}{P(E|\neg H)} \times \frac{P(H)}{P(\neg H)} \quad (6)$$

Forma șanselor: șansele a posteriori = raportul de verosimilitate × șansele a priori

8.5. Rețele Bayesiene: structură și semantică

O rețea Bayesiană (Bayesian Network, BN) este un **model grafic probabilist** care reprezintă un set de variabile aleatoare și dependențele (de regulă cauzale) dintre ele, printr-un graf. Rețelele Bayesiene combină teoria grafurilor cu teoria probabilităților, permițând reprezentarea compactă și calculul eficient al distribuțiilor de probabilitate comună (joint) pentru zeci sau sute de variabile.

Componentele unei rețele Bayesiene

- **Structura:** un graf aciclic dirijat (Directed Acyclic Graph, DAG), în care nodurile reprezintă variabile aleatoare, iar arcele orientate reprezintă dependențe directe (adesea, dar nu obligatoriu, cauzale).
- **Parametrii:** fiecare nod este asociat cu o tabelă de probabilitate condiționată (Conditional Probability Table, CPT), care specifică P(nod | părinții săi în graf).

Structura: graf aciclic dirijat (DAG)

Un graf aciclic dirijat este un graf în care arcele au un sens (dirijate) și în care nu există cicluri – adică nu se poate porni dintr-un nod și, urmând arcele, să te întorci la același nod. Aciclicitatea este esențială: ea garantează că distribuția de probabilitate comună poate fi factorizată corect, fără dependențe circulare care ar face modelul incoerent.

- Nodurile fără părinți (rădăcini) sunt caracterizate de o probabilitate a priori P(X);
- Nodurile cu părinți sunt caracterizate de o probabilitate condiționată P(X | Părinți(X));
- Un arc de la A la B sugerează, de regulă, o relație cauzală: „A influențează B".

Exemplu: rețea Bayesiană pentru diagnoza unei defecțiuni auto

Considerăm un exemplu simplu, intuitiv: o mașină care nu pornește. Cauzele posibile sunt bateria descărcată (B) și lipsa combustibilului (C); ambele pot conduce la eșecul pornirii motorului (P). Dacă motorul nu pornește din cauza bateriei descărcate, este posibil ca și alte componente electrice – radioul (R) și farurile (F) – să nu funcționeze.

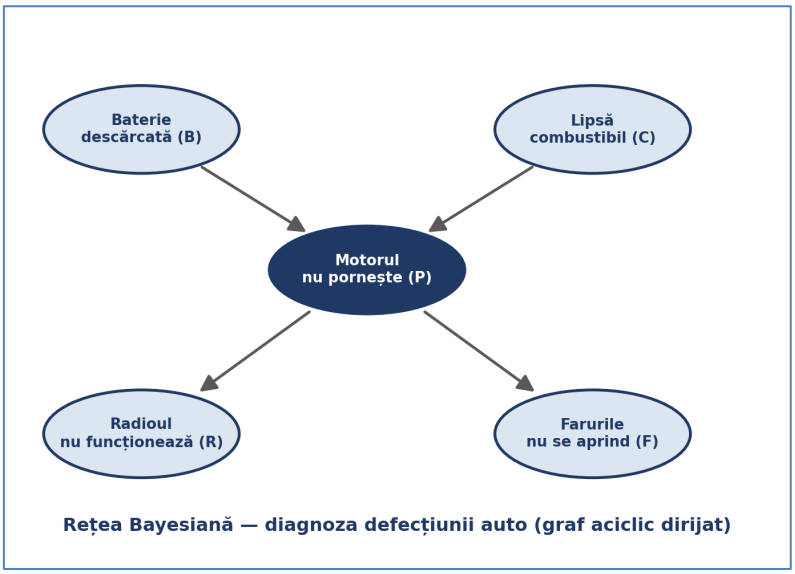


Figura 8.4 Rețea Bayesiană simplă pentru diagnoza defecțiunii auto (graf aciclic dirijat cu 5 noduri)

Observăm structura tipică „cauză comună → efecte multiple": nodul „Pornire motor" (P) este părinte comun pentru „Martor radio" (R) și „Martor faruri" (F). Această structură este cheia proprietății de independență condiționată, discutată în continuare.

Semantica: distribuția de probabilitate comună

O rețea Bayesiană codifică implicit distribuția de probabilitate comună a tuturor variabilelor, printr-o factorizare compactă, cunoscută sub numele de **regula de înlănțuire** (chain rule) pentru rețele Bayesiene:

$$P(X_1, \ldots, X_n) = \prod_{i=1..n} P(X_i \mid Parents(X_i)) \quad (7)$$

Formula generală de factorizare a distribuției comune, pe baza structurii grafului

Pentru rețeaua din exemplul auto, această factorizare devine:

$$P(B, C, P, R, F) = P(B) \cdot P(C) \cdot P(P \mid B, C) \cdot P(R \mid P) \cdot P(F \mid P) \quad (8)$$

Factorizarea distribuției comune pentru rețeaua de diagnoză auto

Avantajul esențial: în loc să fie nevoie de un tabel cu toate combinațiile posibile ale celor 5 variabile (32 de valori, pentru variabile binare), rețeaua Bayesiană necesită doar tabelele de probabilitate condiționată locale – mult mai puține valori de estimat și de stocat.

Independența condiționată

Proprietatea Markov pentru rețele Bayesiene

Într-o rețea Bayesiană, fiecare nod este independent condiționat de non-descendenții săi, dat fiind părinții săi direcți. Cu alte cuvinte, dacă cunoaștem starea părinților unui nod, informația despre alte noduri „mai îndepărtate" în graf nu mai schimbă probabilitatea acelui nod.

În exemplul nostru: dacă știm cu certitudine că motorul nu pornește (nodul P este cunoscut), atunci starea radioului (R) devine independentă de starea bateriei (B) – informația despre baterie „trece prin" nodul P și nu mai adaugă nimic în plus. Formal:

$$P(R \mid P, B) = P(R \mid P) \quad (9)$$

R este condiționat independent de B, dat fiind P (deoarece B influențează R doar prin intermediul lui P)

Această proprietate este ceea ce face rețelele Bayesiene atât de eficiente computațional: ea **reduce** drastic numărul de parametri necesari și permite algoritmi de inferență rapizi, chiar și pentru rețele cu foarte multe variabile.

8.6. Calculul probabilităților într-o rețea Bayesiană simplă

Tabelele de probabilitate condiționată (CPT)

Pentru a putea calcula efectiv probabilități în rețeaua noastră de diagnoză auto, fiecare nod trebuie asociat cu o tabelă de probabilitate condiționată. Presupunem, pentru simplitate, toate variabilele binare (adevărat/fals) și valorile ipotetice de mai jos, tipice unor estimări de tip „expert”:

Tabel A

Variabilă (nod rădăcină)	P(adevărat)	P(fals)
B – Baterie descărcată	0,05	0,95
C – Lipsă combustibil	0,03	0,97

Nodul P (pornirea eșuată a motorului) are doi părinți (B și C), deci CPT-ul său specifică P(P | B, C) pentru toate cele patru combinații posibile:

Tabel B

B	C	P(P=adevărat|B,C)	P(P=fals|B,C)
adevărat	adevărat	0,98	0,02
adevărat	fals	0,90	0,10
fals	adevărat	0,85	0,15
fals	fals	0,01	0,99

Inferența prin enumerare: un calcul pas cu pas

Întrebare: care este probabilitatea ca bateria să fie descărcată (B = adevărat), dacă știm doar că motorul nu pornește (P = adevărat), fără nicio altă informație despre combustibil?

Pasul 1 – aplicăm legea probabilității totale pentru a elimina (marginaliza) variabila C, care nu este observată direct:

$$P(B,P) = \Sigma\, P(B)\cdot P(C)\cdot P(P \mid B,C) \quad pentru\, C \in \{adevărat, fals\} \qquad (10)$$

Pasul 2 – calculăm P(B=adevărat, P=adevărat), însumând peste ambele valori ale lui C:

$$= 0{,}05 \times 0{,}03 \times 0{,}98 + 0{,}05 \times 0{,}97 \times 0{,}90 = 0{,}00147 + 0{,}04365 = 0{,}04512$$

Pasul 3 – calculăm similar P(B=fals, P=adevărat):

$$= 0{,}95 \times 0{,}03 \times 0{,}85 + 0{,}95 \times 0{,}97 \times 0{,}01 = 0{,}024225 + 0{,}009215 = 0{,}03344$$

Pasul 4 – aplicăm regula lui Bayes (normalizare) pentru a obține P(B=adevărat | P=adevărat):

$$P(P \mid R = adevărat) = \frac{P(P,R=adevărat)}{P(R=adevărat)} \qquad (11)$$

$$P(B = adev. \mid P = adev.) = 0{,}04512 \,/\, (0{,}04512 + 0{,}03344) = 0{,}04512 \,/\, 0{,}07856 \approx 0{,}574$$

Exemplu practic – interpretarea rezultatului

Deși probabilitatea a priori ca bateria să fie descărcată era foarte mică (doar 5%), observarea faptului că motorul nu pornește crește dramatic probabilitatea a posteriori la aproximativ 57,4%. Acesta este exact mecanismul de „actualizare a probabilităților pe baza dovezilor" discutat la regula lui Bayes – rețelele Bayesiene aplică acest mecanism sistematic, pe grafuri întregi de variabile interdependente.

În practică, pentru rețele mari, calculul exact prin enumerare devine ineficient, iar sistemele reale folosesc algoritmi de inferență mai avansați (eliminarea variabilelor, propagarea în arbori de joncțiune, metode de eșantionare de tip Monte Carlo). Principiul rămâne însă același: combinarea probabilităților a priori cu dovezile observate, prin regula lui Bayes aplicată sistematic pe structura grafului.

Exemplul ilustrează exact modul în care rețelele Bayesiene sunt folosite în sistemele de diagnoză tehnică – fie pentru automobile, fie pentru echipamente industriale sau infrastructură IT. Structura tipică este:

- ***noduri „cauză"*** (defecțiuni posibile ale componentelor: baterie, combustibil, senzori, alternatoare);
- ***noduri „efect"*** (simptome observabile: motorul nu pornește, martori de eroare, zgomote anormale);
- ***tabele de probabilitate condiționată***, estimate fie din date **istorice de mentenanță**, fie din **cunoștințe ale experților tehnici**.

Avantajul practic esențial: sistemul poate porni de la simptomele observate (dovezi) pentru a calcula, prin inferență bayesiană, probabilitatea fiecărei cauze posibile – ghidând astfel tehnicianul spre verificarea componentei cu cea mai mare probabilitate de defectare, în loc de o verificare exhaustivă, costisitoare în timp, a tuturor componentelor.

De ce rețele Bayesiene și nu doar reguli fixe?

Spre deosebire de sistemele expert clasice bazate pe reguli rigide (DACĂ...ATUNCI...), rețelele Bayesiene actualizează probabilitățile în mod coerent atunci când apar dovezi noi, parțiale sau chiar contradictorii, și pot cuantifica explicit gradul de încredere în fiecare diagnostic posibil – nu doar oferi un răspuns binar.

8.7. Studiu individual: Filtrele de spam

Una dintre cele mai răspândite aplicații ale raționamentului probabilist este filtrarea automată a mesajelor spam. Abordarea clasică, numită Naive Bayes, tratează fiecare e-mail ca o „pungă de cuvinte" (bag of words) și presupune – simplificator, dar surprinzător de eficient în practică – că prezența fiecărui cuvânt este condiționat independentă de prezența celorlalte cuvinte, dat fiind clasa e-mailului (spam sau nu).

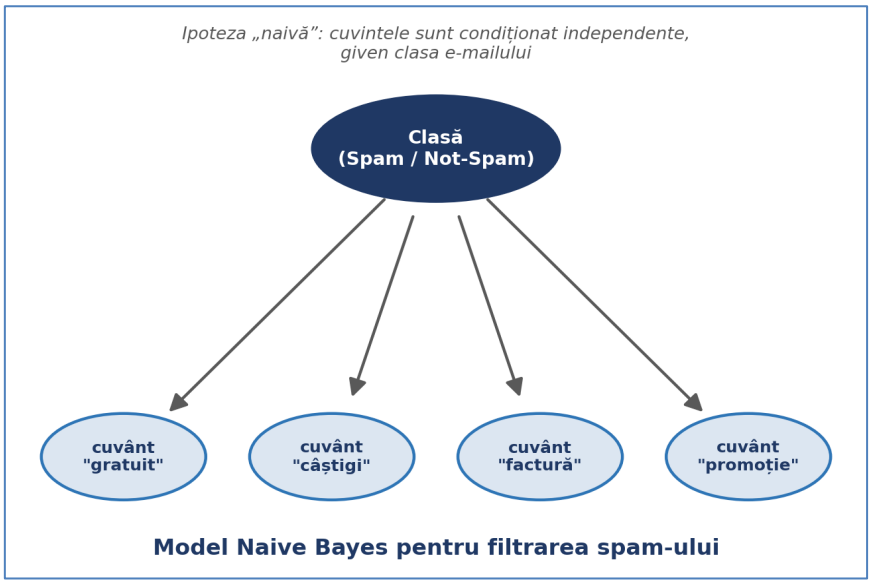


Figura 8.5 Structura rețelei Naive Bayes pentru filtrarea spam-ului: nodul „Clasă” este părinte comun al tuturor cuvintelor observate

Regula de decizie compară probabilitatea a posteriori ca mesajul să fie spam cu probabilitatea ca acesta să fie un mesaj legitim, folosind produsul verosimilităților fiecărui cuvânt:

$$P(Spam \mid w_1, \ldots, w_n) = P(Spam) \cdot \prod_{i=1..n} P(w_i \mid Spam) \qquad (15)$$

Regula Naive Bayes pentru clasificarea unui e-mail cu cuvintele $w_1, \ldots, w_n$

Exemplu practic — clasificarea unui e-mail suspect

Un e-mail conține cuvintele „gratuit”, „câștigi” și „promoție”. Din datele de antrenament, sistemul a estimat: P(Spam)=0,30; P(„gratuit”|Spam)=0,65; P(„câștigi”|Spam)=0,55; P(„promoție”|Spam)=0,50, iar pentru mesajele legitime aceste probabilități sunt mult mai mici (de exemplu, 0,02; 0,01; 0,03). Produsul verosimilităților pentru ipoteza Spam va fi, prin urmare, **mult mai mare** decât pentru ipoteza Not-Spam, iar mesajul va fi clasificat drept spam, cu o probabilitate a posteriori foarte apropiată de 1.

Avantajul modelului Naive Bayes este simplitatea și viteza de calcul: chiar dacă ipoteza de independență condiționată este rareori perfect adevărată în limbajul natural (cuvintele sunt adesea corelate), performanța practică a filtrelor Naive Bayes rămâne remarcabil de bună, motiv pentru care acestea au fost folosite timp de decenii în sistemele de e-mail.

Întrebări de autocontrol

1. Care este diferența conceptuală dintre incertitudine și imprecizie (vag) în inteligența artificială și ce instrumente formale sunt utilizate pentru tratarea fiecăreia?
2. Care sunt principalele surse din care poate proveni incertitudinea într-un sistem inteligent?
3. Cum se definește o mulțime fuzzy și prin ce se diferențiază gradul de apartenență $\mu(x) \in [0,1]$ față de o mulțime clasică binară?
4. Care sunt formele standard ale funcțiilor de apartenență fuzzy utilizate în practică și ce parametri le caracterizează?
5. Care sunt etapele parcurse de un sistem fuzzy de tip Mamdani pentru procesarea intrărilor?
6. De ce este necesară etapa de defuzzificare și prin ce se deosebește metoda centrului de greutate (centroid) de metoda mediei ponderate?

7. Cum sunt definiți operatorii fuzzy de bază introduși de Zadeh pentru intersecție (ȘI) și reuniune (SAU)?
8. Ce reprezintă un factor de certitudine, în ce interval de valori este definit și ce semnificație au valorile $+1, -1$ și 0?
9. Care este formula de combinare a doi factori de certitudine pozitivi (concordanți) și ce reprezintă efectul de „saturare”?
10. Cum se calculează factorul de certitudine al unei premise formate din condiții legate prin operatorii AND, OR și NOT?
11. Ce strategii pot fi aplicate de motorul de inferență pentru a gestiona conflictele când mai multe reguli de producție sunt satisfăcute simultan?
12. Care sunt principalele neajunsuri teoretice și practice ale modelului factorilor de certitudine?
13. Cum se obține Regula lui Bayes pornind de la definiția probabilității condiționate $P(A|B)$?
14. Cum se interpretează termenii din formula Regulii lui Bayes ($P(H)$, $P(E|H)$, $P(E)$, $P(H|E)$) ?
15. Cum se exprimă forma extinsă a Regulii lui Bayes folosind legea probabilității totale și în ce cazuri este utilă?
16. Care sunt cele două componente fundamentale ale unei rețele Bayesiene (componenta structurală și cea parametrică)?
17. Ce este un Graf Aciclic Dirijat (DAG) și de ce este esențială absența ciclurilor pentru factorizarea distribuției de probabilitate?
18. Cum se calculează distribuția de probabilitate comună $P(X_1, \dots, X_n)$ într-o rețea Bayesiană pe baza părinților fiecărui nod?
19. Ce presupune proprietatea Markov și cum determină aceasta independența condiționată într-un graf probabilist?
20. Explicați, cu propriile cuvinte, de ce independența condiționată reduce numărul de parametri necesari într-o rețea Bayesiană.
21. Propuneți o rețea Bayesiană simplă (3-4 noduri) pentru un domeniu de aplicație .

CAPITOLUL 9. INTRODUCERE ÎN MACHINE LEARNING

9.1. De la programarea clasică la Machine Learning

Machine Learning (învățarea automată) reprezintă în prezent unul dintre cele mai importante domenii ale inteligenței artificiale, fiind prezent în aplicații pe care le folosim zilnic: recomandările de pe platformele de streaming, filtrele anti-spam, recunoașterea vocală, diagnosticul medical asistat, sistemele de credit scoring sau vehiculele autonome.

Pentru a înțelege ce este cu adevărat inovator în Machine Learning, trebuie mai întâi să analizăm modul tradițional de a rezolva probleme prin programare.

În **programarea clasică**, dezvoltatorul scrie explicit un set de reguli (algoritmul), care, aplicate asupra datelor de intrare, produc răspunsurile (rezultatele) dorite. Formula conceptuală este:

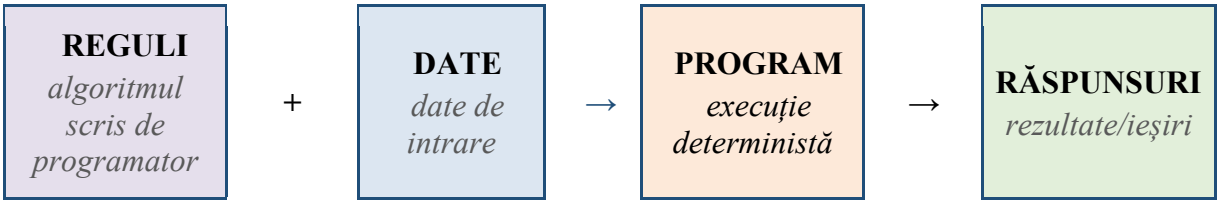


Figura 9.1 Programarea clasică: Reguli + Date = Răspunsuri

Exemplu: pentru a calcula media notelor unui student, programatorul scrie o regulă fixă (suma notelor împărțită la numărul acestora). Regula este cunoscută dinainte și nu se schimbă în funcție de date.

În **Machine Learning**, relația se inversează. În loc să scriem regulile, îi oferim algoritmului exemple de date împreună cu răspunsurile corecte, iar acesta „învață" singur regulile (modelul) care leagă datele de răspunsuri:

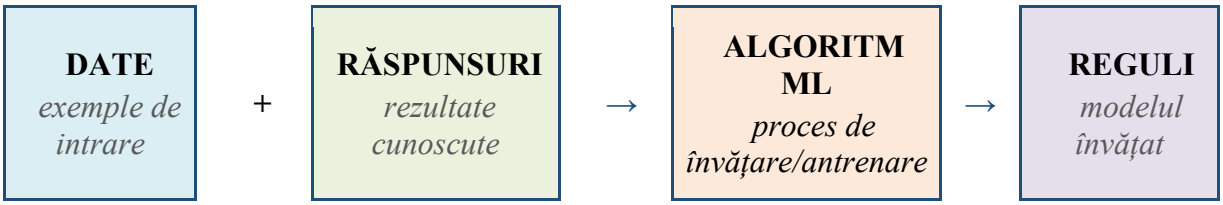


Figura 9.2 Machine Learning: Date + Răspunsuri = Reguli (Model)

Exemplu: pentru a determina dacă un e-mail este spam, în loc să scriem manual sute de reguli (cuvinte suspecte, expeditori etc.), oferim algoritmului mii de e-mailuri deja etichetate drept „spam" sau „non-spam". Algoritmul „descoperă" singur tiparele (regulile) care diferențiază cele două categorii.

Tabel 9.1 Programarea clasică vs Machine Learning

Criteriu	Programare clasică	Machine Learning
Ce oferă omul	Reguli + Date	Date + Răspunsuri
Ce rezultă	Răspunsuri	Reguli (model)
Adaptabilitate	Redusă – regulile sunt fixe	Ridicată – modelul se poate reantrena
Potrivit pentru	Probleme cu reguli clare și cunoscute	Probleme complexe, cu tipare greu de formalizat
Exemplu	Calculul TVA, salariu	Recunoașterea facială

9.2. Structura unui set de date

Orice algoritm de Machine Learning este antrenat pe baza unui **set de date** (dataset). Pentru a lucra corect cu aceste date, trebuie să cunoaștem trei concepte fundamentale: **caracteristicile** (features), **etichetele** (labels) și **instanțele**.

- **Instanță** (exemplu, observație, rând) – o singură înregistrare din setul de date, reprezentând un caz concret (de exemplu, un client, un e-mail, o imagine).
- **Caracteristici** (features, atribute) – proprietățile măsurabile ale unei instanțe, folosite ca date de intrare pentru model (de exemplu, vârsta, venitul, numărul de întârzieri la plată).
- **Etichetă** (label, target, variabilă țintă) – răspunsul/rezultatul cunoscut, pe care modelul încearcă să îl prezică (de exemplu, „credit aprobat” / „credit respins”).

Instanță	Vârstă (feature)	Venit lunar (MDL) (feature)	Istoric credit (feature)	Credit aprobat (label)
1	34	12 500	Bun	Da
2	22	6 800	Fără istoric	Nu
3	45	18 200	Bun	Da
4	29	8 400	Întârzieri	Nu
5	51	15 000	Bun	Da

Figura 9.3. Set de date pentru aprobarea creditelor. Fiecare rând este o instanță; coloanele evidențiate reprezintă caracteristicile (features); ultima coloană (verde) este eticheta (label).

Formal, un set de date poate fi reprezentat printr-o matrice a caracteristicilor $X \in \mathbb{R}^{n \times m}$ (cu n instanțe și m caracteristici) și un vector al etichetelor $y \in \mathbb{R}^n$::

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,m} \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{1}$$

unde n = numărul de instanțe, iar fiecare instanță x_i are m caracteristici (feature-uri).

Modelul de Machine Learning are ca scop găsirea unei funcții f care aproximează cât mai bine relația dintre caracteristici (X) și etichete (y):

$$\hat{y} = f(X) \tag{2}$$

unde $\hat{y} \in \mathbb{R}^n$ este vectorul valorilor prezise de model pentru toate cele n instanțe, iar f este funcția învățată din date.

9.3. Etapele unui proiect de Machine Learning

Un proiect real de Machine Learning nu constă doar în „rularea unui algoritm”, ci parcurge o serie de etape esențiale, fiecare având un impact direct asupra calității rezultatului final.

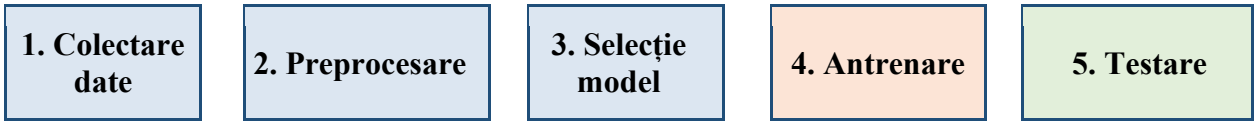


Figura 9.4 Etapele principale ale unui proiect de Machine Learning

Colectarea datelor constă în identificarea și strângerea datelor relevante pentru problema de rezolvat, din surse precum baze de date, senzori, formulare web, API-uri sau seturi de date publice.

Calitatea și volumul datelor colectate influențează direct performanța finală a modelului – principiul „garbage in, garbage out” este esențial în ML.

Preprocesarea datelor: Datele brute conțin, de regulă, erori, valori lipsă sau formate inconsistente. Preprocesarea include:

- Curățarea datelor – eliminarea sau completarea valorilor lipsă/eronate.
- Normalizarea/standardizarea – aducerea caracteristicilor la o scală comparabilă.
- Codificarea variabilelor categoriale – transformarea textului în valori numerice.
- Împărțirea setului de date – de obicei în subset de antrenare și subset de testare (ex. 80% / 20%).

Selecția modelului: Alegerea algoritmului potrivit (de exemplu, regresie liniară, arbori de decizie, rețele neuronale) depinde de tipul problemei (clasificare sau regresie), de volumul de date disponibil, de complexitatea relațiilor din date și de cerințele privind interpretabilitatea rezultatelor.

Antrenarea modelului: În această etapă, algoritmul „învață” din datele de antrenare, ajustându-și parametrii interni pentru a minimiza eroarea dintre valorile prezise și valorile reale (etichetele cunoscute). Acest proces se realizează, de regulă, prin minimizarea unei funcții de cost (loss function), de exemplu eroarea medie pătratică:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad (3)$$

Eroarea medie pătratică (MSE) — măsoară cât de mult diferă valorile prezise ($\hat{y}_i$) de valorile reale (y_i).

Testarea modelului: Modelul antrenat este evaluat pe date noi (subsetul de testare), pe care nu le-a „văzut” în timpul antrenării, pentru a verifica dacă performanța se menține și pe date reale, nu doar pe cele memorate. Pentru problemele de clasificare, o măsură frecvent utilizată este acuratețea:

$$Acuratețe = \frac{Nr.predicții\ corecte}{Nr.total\ de\ predicții} \quad (4)$$

Acuratețea modelului — proporția predicțiilor corecte din totalul predicțiilor efectuate.

9.4. Tipuri de învățare automată

În funcție de modul în care sunt utilizate datele în procesul de antrenare, distingem trei tipuri fundamentale de învățare automată: supervizată, nesupervizată și prin întărire.



Figura 9.5 Cele trei tipuri fundamentale de învățare automată

Învățarea supervizată (Supervised Learning): Algoritmul este antrenat pe date etichetate – pentru fiecare instanță de intrare se cunoaște deja răspunsul corect (eticheta). Scopul este ca modelul să învețe relația dintre caracteristici și etichete, pentru a putea prezice corect eticheta unor date noi, neetichetate.

- Clasificare – eticheta este o categorie discretă (ex. spam / non-spam, aprobat / respins).
- Regresie – eticheta este o valoare numerică continuă (ex. prețul unei case, temperatura de mâine).

Exemple de algoritmi: regresia liniară, regresia logistică, arbori de decizie, mașini cu vectori suport (SVM), rețele neuronale.

Învățarea nesupervizată (Unsupervised Learning): Algoritmul primește doar date fără etichete și trebuie să descopere singur structuri, tipare sau grupări ascunse în date, fără a avea un răspuns „corect” predefinit.

- Clusterizare (grupare) – separarea instanțelor în grupuri similare (ex. segmentarea clienților).
- Reducerea dimensionalității – simplificarea datelor păstrând informația esențială (ex. PCA).
- Detectarea anomaliilor – identificarea instanțelor atipice (ex. tranzacții frauduloase).

Exemple de algoritmi: K-Means, clusterizare ierarhică, PCA (Analiza Componentelor Principale), DBSCAN.

Învățarea prin întărire (Reinforcement Learning): Un agent interacționează cu un mediu, luând decizii/acțiuni succesive, și primește recompense sau penalizări în funcție de rezultatele acestor acțiuni. Scopul agentului este să învețe o strategie (politică) care maximizează recompensa cumulată pe termen lung.

- Nu există un set de date etichetat inițial – agentul învață prin încercare și eroare (trial and error).
- Este utilizată în robotică, jocuri (ex. AlphaGo), sisteme de conducere autonomă, optimizarea proceselor industriale.

Exemple de algoritmi: Q-Learning, Deep Q-Networks (DQN), Policy Gradient.

Tabel A Compararea metodelor de învățare

Criteriu	Supervizată	Nesupervizată	Prin întărire
Date de intrare	Etichetate (X, y)	Neetichetate (doar X)	Interacțiuni agent-mediu
Obiectiv	Prezicerea etichetei	Descoperirea structurii	Maximizarea recompensei
Feedback	Răspuns corect cunoscut	Fără feedback extern	Recompensă/penalizare
Exemplu tipic	Diagnostic medical	Segmentare clienți	Robot care învață să meargă

9.5. Fluxul complet al unui proiect ML. Supra-antrenare și generalizare

Fluxul iterativ al unui proiect de Machine Learning

În practică, procesul de dezvoltare a unui model ML nu este strict liniar, ci iterativ: dacă rezultatele testării nu sunt satisfăcătoare, ne întoarcem la etapele anterioare (ajustarea hiperparametrilor, revizuirea preprocesării sau chiar recolectarea datelor).

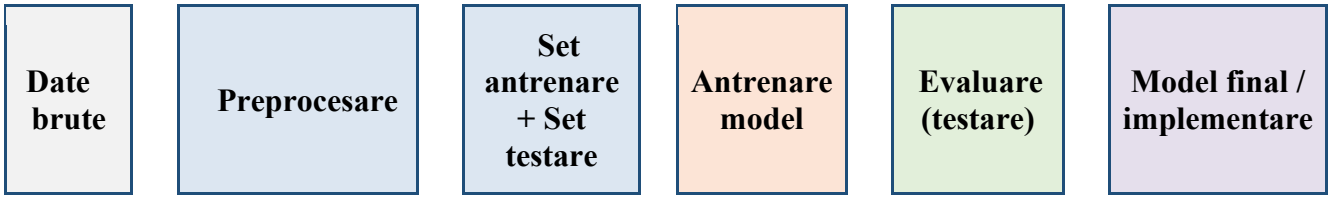


Figura 9.6 Fluxul complet al unui proiect de Machine Learning

↑ Buclă de ajustare (tuning): dacă evaluarea arată performanțe slabe, ne întoarcem și reajustăm modelul sau datele ↑

Generalizarea reprezintă capacitatea unui model de a performa bine pe date noi, pe care nu le-a văzut în timpul antrenării – acesta este, de fapt, scopul final al oricărui model de Machine Learning.

Supra-antrenarea (overfitting) apare atunci când modelul „memorează" prea mult din particularitățile și zgomotul datelor de antrenare, în loc să învețe tiparele generale. Rezultatul este un model care are performanțe excelente pe datele de antrenare, dar performanțe slabe pe date noi.

Fenomenul opus, **sub-antrenarea** (underfitting), apare atunci când modelul este prea simplu pentru a capta relațiile reale din date.

Tabel B Relația dintre complexitatea modelului și eroarea de antrenare/testare

Situație	Eroare pe antrenare	Eroare pe testare	Cauză principală
Sub-antrenare (underfitting)	Ridicată	Ridicată	Model prea simplu
Model optim (generalizare bună)	Scăzută	Scăzută	Complexitate adecvată
Supra-antrenare (overfitting)	Foarte scăzută	Ridicată	Model prea complex / date insuficiente

Formal, eroarea de generalizare a unui model poate fi descompusă în trei componente, cunoscute sub numele de compromisul bias–varianță (bias-variance tradeoff):

$$Eroare = Bias^2 + Varianță + Zgomot \quad (5)$$

Bias (eroare sistematică) – cât de departe sunt, în medie, predicțiile modelului de valorile reale (asociat sub-antrenării). Varianță – cât de mult variază predicțiile modelului la schimbarea datelor de antrenare (asociată supra-antrenării). Zgomot – eroare ireductibilă, inerentă datelor.

Metode de prevenire a supra-antrenării

- Colectarea unui volum mai mare de date de antrenare, reprezentative pentru problema reală.
- Simplificarea modelului (reducerea numărului de parametri/complexității).
- Validarea încrucișată (cross-validation) pentru o evaluare mai robustă a performanței.
- Tehnici de regularizare (ex. L1/L2), care penalizează modelele prea complexe.
- Oprirea timpurie (early stopping) a antrenării, în funcție de performanța pe setul de validare.

Întrebări de autocontrol

1. Care este diferența fundamentală dintre programarea clasică și Machine Learning în ceea ce privește intrările și ieșirile procesului?
2. Cum se definesc și ce rol au instanțele, caracteristicile (*features*) și etichetele (*labels*) în cadrul unui set de date?
3. Care sunt principalele activități desfășurate în etapa de preprocesare a datelor și de ce este importantă regula „garbage in, garbage out"?
4. Cum se măsoară performanța unui model în etapa de antrenare și testare ?
5. Care este diferența dintre sarcinile de clasificare și cele de regresie în cadrul învățării supervizate?
6. Prin ce se caracterizează învățarea nesupervizată?
7. Cum funcționează învățarea prin întărire (*Reinforcement Learning*) și prin ce diferă modul ei de obținere a feedback-ului față de învățarea supervizată?
8. Ce reprezintă capacitatea de generalizare a unui model de Machine Learning?
9. Ce este supra-antrenarea (*overfitting*) și prin ce se deosebește de sub-antrenare (*underfitting*) din perspectiva erorilor pe seturile de antrenare și testare?
10. Care sunt componentele de bază ale compromisului bias–varianță (*bias-variance tradeoff*) și ce metode pot fi aplicate pentru a preveni supra-antrenarea?

CAPITOLUL 10. ÎNVĂȚARE SUPERVIZATĂ

10.1. Regresia liniară

Învățarea supervizata este acea ramura a învățării automate (Machine Learning) in care algoritmul învață dintr-un set de exemple pentru care răspunsul corect este deja cunoscut. Practic, oferim algoritmului perechi de tipul (intrare, răspuns corect), iar scopul lui este sa descopere o regula generala care sa permită prezicerea răspunsului pentru exemple noi, nevăzute anterior.

Analogia cea mai simpla este cea a unui elev care învață rezolvând exerciții cu soluții deja oferite de profesor: după ce vede suficiente exemple rezolvate, elevul începe sa recunoască tiparul si poate rezolva singur exerciții noi, similare.

Învățarea supervizată se confruntă cu două mari tipuri de probleme: **regresie** și **clasificare**

Tabel A. **Tipuri de probleme ale învățării supervizate**

Tip de problema	Ce prezice modelul	Exemplu
Regresie	O valoare numerica continua	Pretul unui apartament (in lei)
Clasificare	O categorie (clasa) dintr-o multime finita	Imprumut aprobat / respins

> **De retinut**
> Regresie -> raspunsul este un **numar** (ex: 3200 EUR, 25.4°C).
> Clasificare -> raspunsul este o **eticheta**/**categorie** (ex: da/nu, spam/non-spam).

Regresia liniara este cel mai simplu si mai intuitiv algoritm de învățare supervizata pentru probleme de regresie. Ideea centrala este găsirea unei linii drepte, care descrie cat mai bine relația dintre variabilele de intrare (caracteristici) si variabila de ieșire (valoarea pe care vrem sa o prezicem).

10.1.1. Functia de ipoteza (modelul)

Pentru un singur factor de intrare x (de exemplu, suprafață unui apartament), modelul liniar are forma:

$$h(x) = wx + b \tag{1}$$

unde h(x) este **valoarea prezisa de model**, w este panta dreptei (numita si **pondere** sau **coeficient**), iar b este punctul in care dreapta intersectează axa verticală (numit si **prag** sau **bias**). Antrenarea modelului înseamnă, de fapt, găsirea celor mai bune valori pentru w si b.

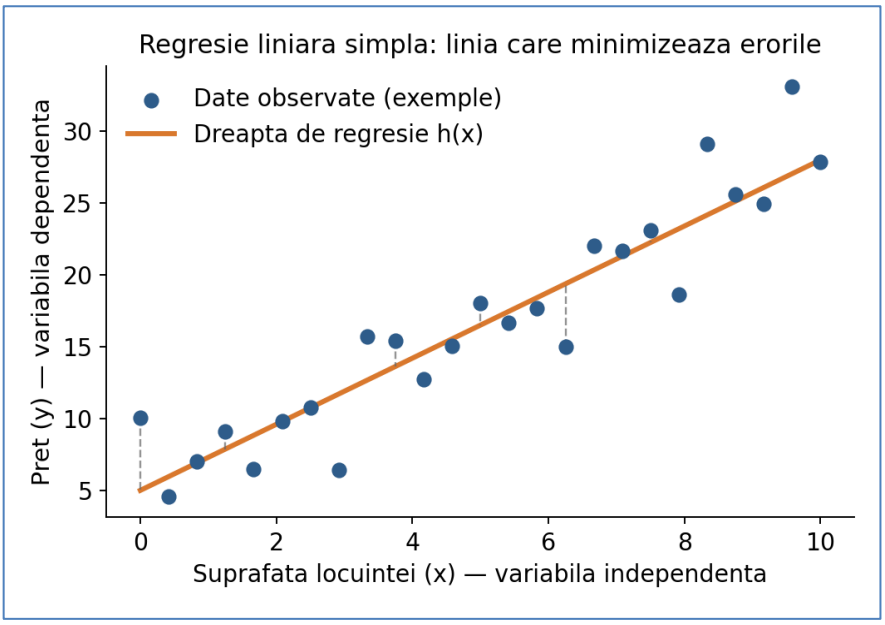


Figura 10.1 Dreapta de regresie liniara: minimizarea distantelor (erorilor) dintre puncte și dreapta.

10.1.2. Funcția de cost (Mean Squared Error – MSE)

Pentru a ști cat de "buna" este o anumita dreapta, avem nevoie de o modalitate de a măsura eroarea. Cea mai folosita metoda pentru regresie este Eroarea Pătratică Medie (MSE), care calculează media pătratelor diferențelor dintre valorile prezise si valorile reale:

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad (2)$$

In cuvinte simple: pentru fiecare exemplu din setul de date, calculam diferența dintre valoarea prezisa ($\hat{y}_i$) si cea reala (y_i), ridicam aceasta diferență la pătrat (ca sa eliminam semnul negativ si sa penalizam mai mult erorile mari), apoi facem media tuturor acestor valori, unde n este numărul total de exemple. Cu cat J(w,b) este mai mic, cu atât modelul este mai bun.

10.1.3 Algoritmul Gradient Descent

Odată ce avem o funcție de cost, rămâne întrebarea: cum găsim valorile w si b care fac aceasta funcție minima? Răspunsul este **algoritmul Gradient Descent** ("coborârea pe gradient").

Principiul este simplu si intuitiv: imaginați-va ca va aflați pe un munte, in ceata, si vreți sa ajungeți in vale (punctul cel mai jos). Nu vedeți tot peisajul, dar puteți simți panta terenului sub picioare. Soluția rațională este sa faceți pași mici in direcția in care panta coboară cel mai abrupt. Repetând acest proces, ajungeți in cele din urmă in vale.

Formal, la fiecare pas, parametrii se actualizează după regula:

$$w := w - \alpha \cdot \frac{\partial J}{\partial w} \quad (3)$$

unde **α** (alfa) este **rata de invatare** (learning rate) — un număr mic, pozitiv, care controlează mărimea pasului. Daca α este prea mic, învățarea este foarte lenta; daca este prea mare, algoritmul poate "sari" peste minim si sa nu conveargă niciodată.

Termenul ∂J/∂w reprezintă **panta** (gradientul) funcției de cost in punctul curent – practic, direcția si viteza celei mai abrupte creșteri, motiv pentru care ne mișcăm in sens opus (de aici semnul minus).

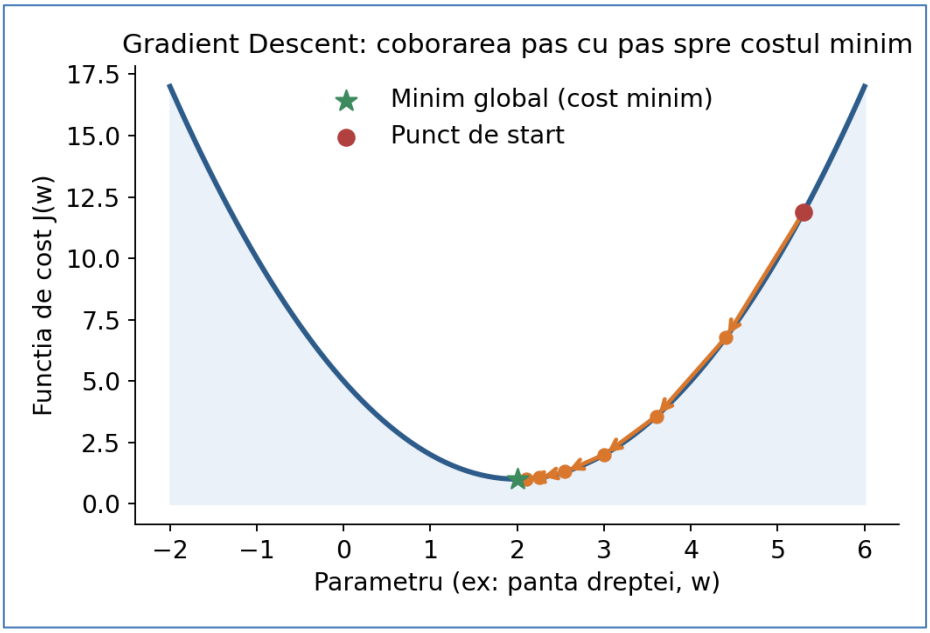


Figura 10.2 Gradient Descent: la fiecare pas, parametrul se ajustează in direcția care reduce costul.

De retinut

Cost mic = model bun; cost mare = model slab.

Gradient Descent repeta pasi mici pana cand costul nu mai scade semnificativ (convergență).

Rata de învățare (α) este un **hiperparametru esențial**: trebuie aleasa cu grija.

10.2. Clasificare prin regresie logistică

Regresia liniara produce valori care pot fi orice număr real, de la minus infinit la plus infinit. Însă, pentru probleme de **clasificare binara** (de exemplu, "aprobat" sau "respins"), avem nevoie ca modelul sa producă o **probabilitate** – un număr cuprins **strict** intre 0 si 1. Aici intervine funcția Sigmoid.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4}$$

unde z este rezultatul unei combinații liniare a caracteristicilor de intrare (exact ca in regresia liniara: $z = w_1x_1 + w_2x_2 + ... + b$), iar e este constanta lui Euler (aproximativ 2,718).

Modelul rezultat se numește **regresie logistica**. Deși conține cuvântul "regresie", acesta este de fapt un **algoritm de clasificare**: valoarea Sigmoid este interpretata drept probabilitatea ca exemplul sa aparțină clasei pozitive. Daca aceasta probabilitate depășește un anumit **prag** (de obicei 0,5), modelul prezice clasa 1; in caz contrar, prezice clasa 0 (vezi Figura 10.3).

De retinut

Sigmoid "încadrează" orice număr real întrun interval (0, 1), interpretabil ca **probabilitate**.

Regresia liniară este o **predicție numerica**; **Regresia logistică** este o **predicție de tip probabilitate/clasa. Pragul de decizie** standard este 0,5, dar el poate fi ajustat in funcție de aplicație.

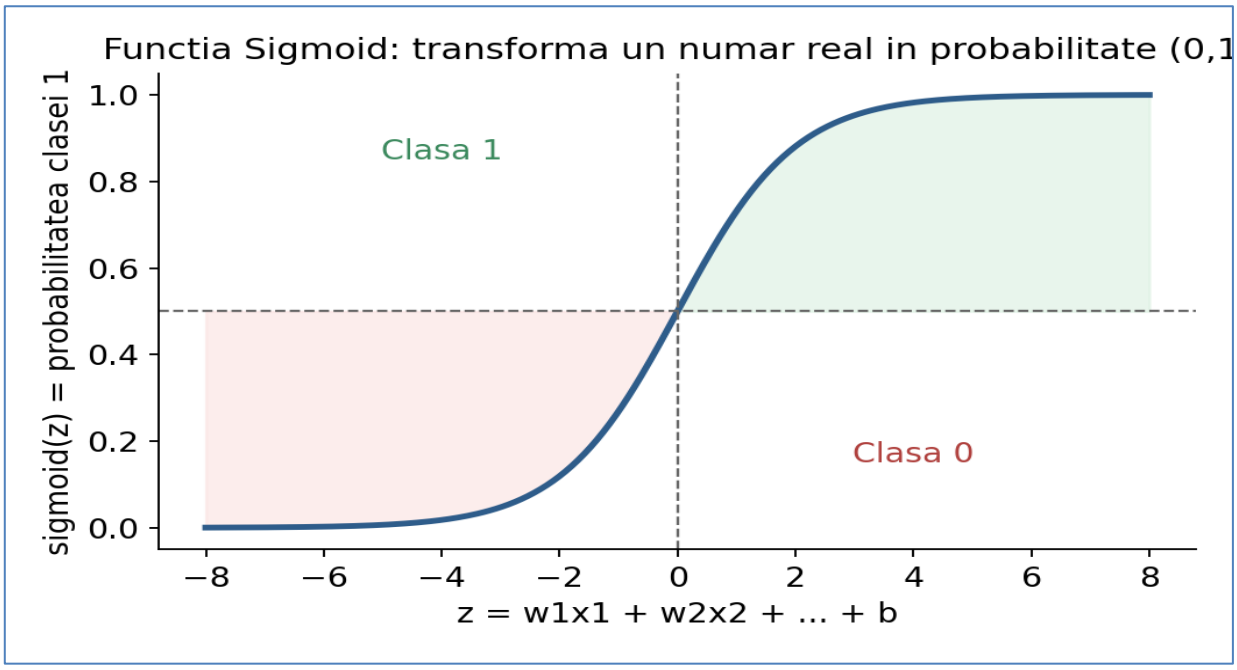


Figura 10.3 Functia Sigmoid transforma orice număr real într-o valoare intre 0 si 1.

10.3. Metrici de evaluare a modelelor

10.3.1 Metrici pentru regresie

Pentru a evalua un model de regresie, folosim câteva metrici standard, toate bazate pe diferența dintre valorile reale si cele prezise:

Tabel B. **Metrici pentru regresie**

Metrica	Semnificatie
MSE (Eroarea Pătratică Medie)	Media pătratelor erorilor; penalizează puternic erorile mari.
RMSE (Rădăcina din MSE)	Rădăcina pătrată a MSE; are aceeași unitate de măsură ca variabila prezisa.
MAE (Eroarea Absoluta Medie)	Media valorilor absolute ale erorilor; mai puțin sensibilă la valori extreme.
R^2 (Coeficientul de determinare)	Proporția din variația datelor explicată de model (0 = slab, 1 = perfect).

10.3.2 Metrici pentru clasificare

Pentru problemele de clasificare, punctul de plecare este **matricea de confuzie**, care compară predicțiile modelului cu valorile reale:

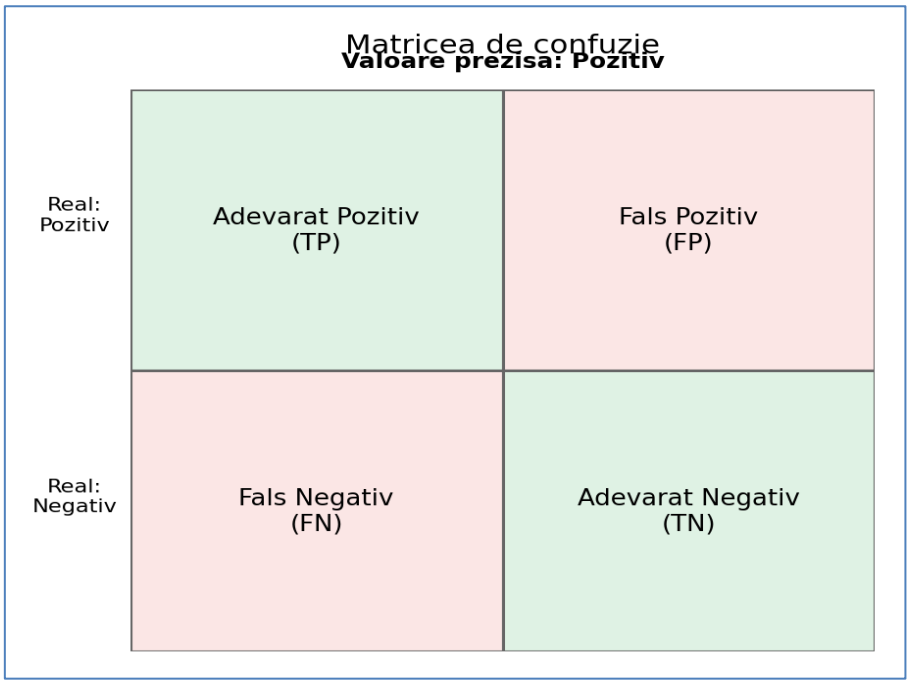


Figura 10.4 Matricea de confuzie pentru o problema de clasificare binara.

Pornind de la matricea de confuzie, se calculează următoarele metrici:

$$Acurateţe = \frac{TP + TN}{TP + TN + FP + FN} \quad (5)$$

$$Precizie = \frac{TP}{TP + FP} \quad (6)$$

$$Recall = \frac{TP}{TP + FN} \quad (7)$$

$$F1 - score = 2 \cdot \frac{Precizie * Recall}{Precizie + Recall} \quad (8)$$

Tabel C. Metrici de clasificare

Metrica	Intrebarea la care raspunde
Acurateţe	Câte predicţii, in total, au fost corecte?
Precizie	Din cele prezise ca pozitive, câte chiar au fost pozitive?
Recall (Sensibilitate)	Din cele real pozitive, cate au fost găsite de model?
F1-score	Media armonica intre precizie si recall (echilibru intre cele doua).

De retinut

Acurateţea poate fi înşelătoare pe seturi de date **dezechilibrate** (ex: 95% cazuri negative). In aprobarea creditelor, un Fals Pozitiv (credit acordat greşit) poate costa mai mult decât un Fals Negativ, motiv pentru care **precizia** este adesea prioritară.

10.4. Arbori de decizie

Un **arbore de decizie** este un model care ia decizii printr-o succesiune de întrebări de tip da/nu, organizate ca o structura arborescenta. Fiecare nod intern reprezintă o întrebare despre o caracteristică, fiecare ramură reprezintă un răspuns posibil, iar fiecare frunză reprezintă decizia finală (clasa prezisă).

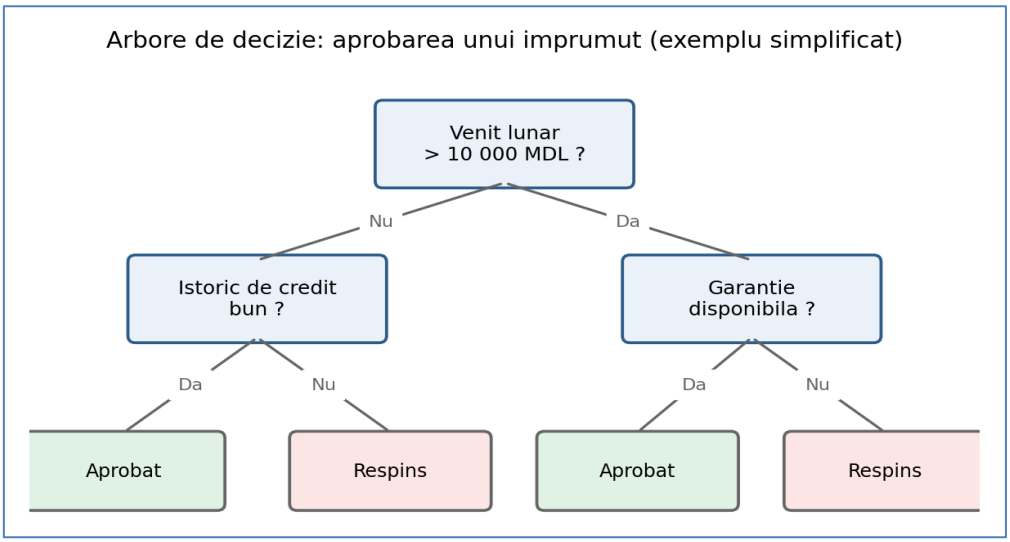


Figura 10.5 Exemplu simplificat de arbore de decizie pentru aprobarea unui împrumut.

Entropia

Algoritmul trebuie sa decidă, la fiecare nod, care este cea mai informativă întrebare de pus. Pentru aceasta, se folosește conceptul de **entropie**, împrumutat din teoria informației, care măsoară gradul de "dezordine" sau incertitudine dintr-un set de date:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2 p_i \quad (9)$$

unde p_i este **proporția** exemplelor din clasa i, iar c este numărul de clase. Entropia este 0 când toate exemplele aparțin aceleiași clase (*certitudine totală*) si este maximă când clasele sunt perfect amestecate (*incertitudine maximă*).

Câștigul de informație (Information Gain)

Câștigul de informație măsoară cât de mult scade entropia după ce am împărțit datele folosind o anumită caracteristică. Cu cât câștigul este mai mare, cu atât acea întrebare/caracteristică este mai utilă pentru clasificare:

$$IG(S,A) = H(S) - \sum_{v \in Valori(A)} \frac{|S_v|}{|S|} \cdot H(S_v) \quad (10)$$

In termeni simpli, algoritmul testează fiecare caracteristica disponibilă, calculează câștigul de informație pentru fiecare, si alege pentru fiecare nod caracteristica ce oferă cel mai mare câștig — adică întrebarea care "curață" cel mai eficient incertitudinea din date.

De retinut

Entropie mica determină un grup omogen (majoritatea aparțin aceleiași clase).

Arborele alege, la fiecare pas, întrebarea cu cel mai mare câștig de informație.

Avantaj: ușor de interpretat si vizualizat. Dezavantaj: predispus la supra-antrenare (overfitting) daca nu este limitata adâncimea.

10.5. Algoritmul k-Nearest Neighbors (k-NN)

Algoritmul k-NN este, probabil, cel mai intuitiv algoritm de învățare supervizată: nu construiește un model explicit in faza de antrenare, ci pur si simplu "memorează" toate exemplele. Când primește un exemplu nou, îl clasează in funcție de clasa majoritara a celor mai apropiați k vecini ai săi (măsurați, de obicei, prin distanta euclidiana).

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \quad (11)$$

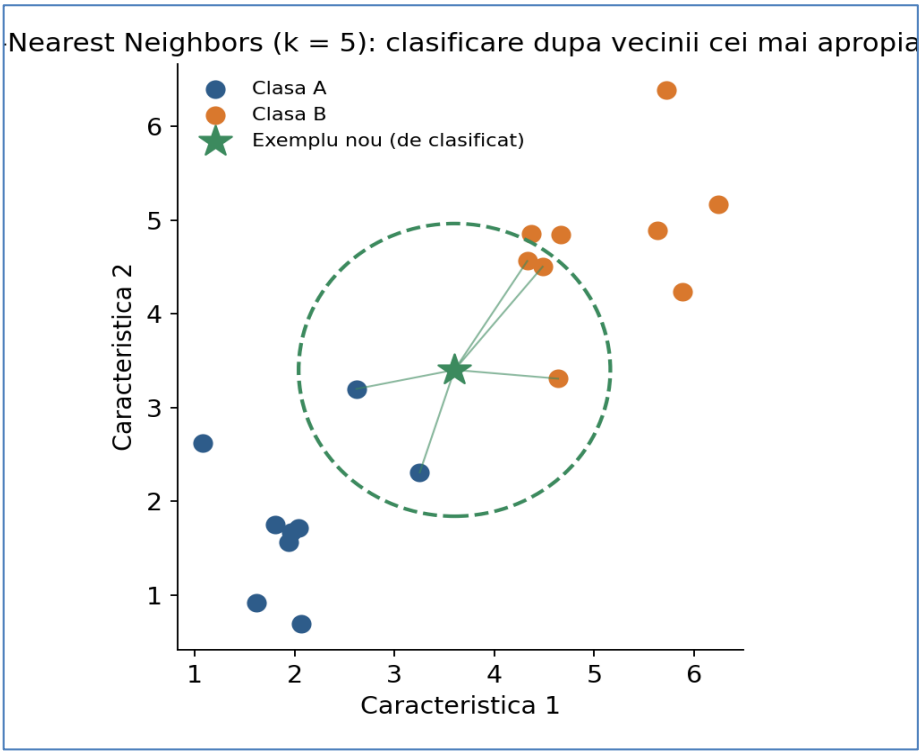


Figura 10.6 Algoritmul k-NN cu k=5: exemplul nou este clasificat prin vot majoritar al celor mai apropiați 5 vecini.

Alegerea valorii k este importanta: un k prea mic face modelul sensibil la zgomot (o singură observație aberantă poate schimba decizia), iar un k prea mare poate "dilua" tiparele locale relevante, apropiind decizia de simpla majoritate globală.

De retinut

k-NN nu "învață" propriu-zis un model – este un **algoritm bazat pe memorie** (lazy learning).

Este **simplu** si **eficient** pe **seturi de date mici**, dar devine **lent** pe seturi de date foarte mari.

Este **sensibil** la scala caracteristicilor – de aceea datele trebuie **normalizate** înainte de utilizare.

10.6. Masini cu vectori suport (SVM)

Mașinile cu vectori suport (Support Vector Machines) sunt algoritmi de clasificare care caută cea mai buna "graniță" (***hiperplan***) care separă cele două clase, astfel încât distanta (marginea) dintre **graniță** si **cele mai apropiate puncte din fiecare clasa** sa fie **cat mai mare posibilă.**

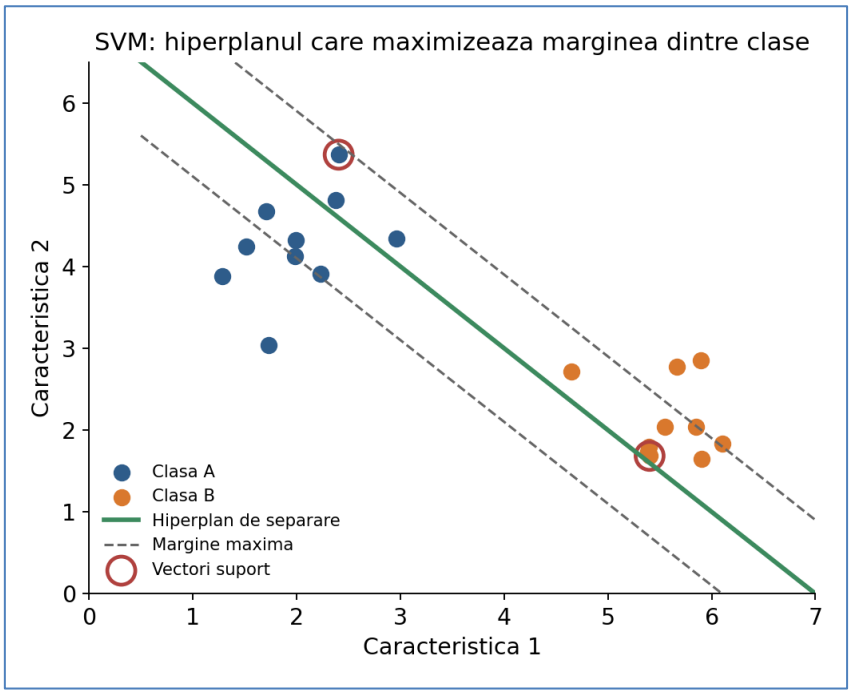


Figura 10.7 SVM: hiperplanul de separare, marginea maxima si vectorii suport (punctele încercuite).

Punctele cele mai apropiate de graniță – cele care "sprijină" poziția hiperplanului – se numesc **vectori suport** (deoarece ele „susțin” și definesc poziția și orientarea graniței) si dau numele algoritmului. Doar aceste puncte influențează direct poziția granitei de decizie; restul exemplelor pot fi ignorate fără a schimba rezultatul.

Pentru date care nu pot fi separate printr-o linie dreaptă, SVM folosește așa-numitele funcții kernel, care proiectează datele într-un spațiu cu mai multe dimensiuni, unde separarea liniara devine posibilă.

De reținut

SVM caută marginea maxima intre clase, nu doar orice linie de separare. Maximizarea marginii face ca algoritmul SVM să fie **robust**: oferă cea mai bună capacitate de generalizare atunci când clasifică date noi, reducând riscul de supraînvățare (*overfitting*). Funcționează foarte bine pe seturi de date de dimensiune medie, cu granițe de decizie clare.

Funcțiile kernel permit SVM sa rezolve si probleme neliniar separabile.

10.7. Metode ansamblu: Random Forest

Ideea metodelor ansamblu ("ensemble") este simplă: in loc să ne bazăm pe un singur model, combinăm predicțiile mai multor modele pentru a obține un rezultat mai stabil si mai precis. Random Forest ("Pădurea Aleatorie") este cea mai populară metodă ansamblu bazată pe arbori de decizie(vezi Figura 10.8).

Random Forest antrenează mulți arbori de decizie diferiți, fiecare pe un eșantion aleatoriu din datele de antrenare si folosind un subset aleatoriu de caracteristici la fiecare divizare. La final, pentru clasificare, fiecare arbore "votează" o clasă, iar clasa cu cele mai multe voturi este predicția finală a pădurii (pentru **regresie**, se face **media predicțiilor**).

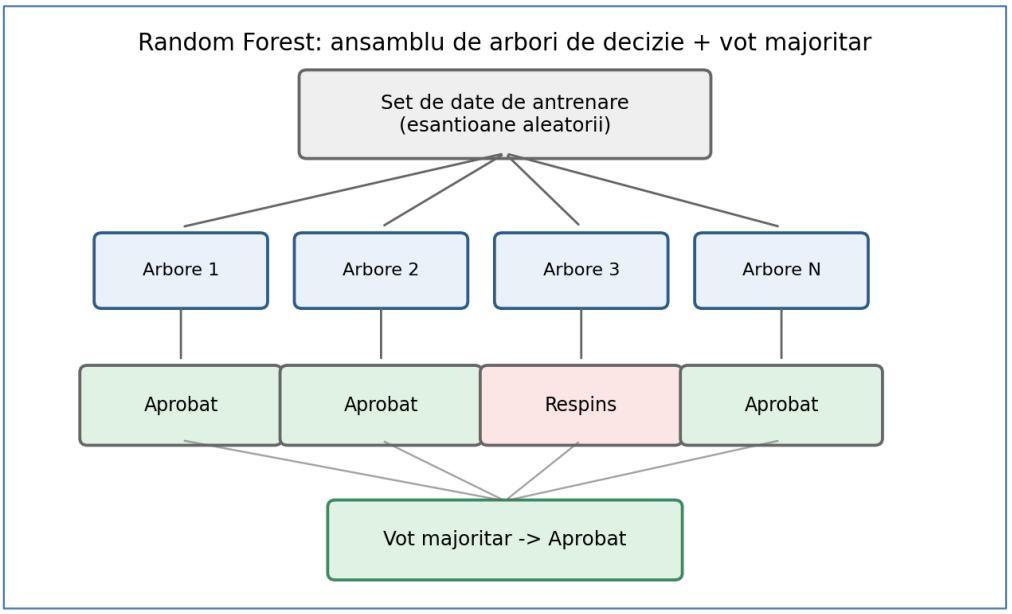


Figura 10.8 Random Forest: mai mulți arbori de decizie, antrenați pe subseturi diferite de date, decid prin vot majoritar.

De reținut

Un singur arbore de decizie se poate supra-antrena ușor; o "pădure" de arbori este mult mai robusta.

Aleatorietatea (eșantioane si caracteristici diferite) face ca arborii sa fie diverși, iar diversitatea îmbunătățește acuratețea ansamblului.

Random Forest este unul dintre cei mai folosiți algoritmi in aplicații practice datorită robustetii si performantei sale.

Predicția prețului imobilelor (problema de regresie): Un model de regresie liniară (sau unul mai complex, precum Random Forest de regresie) poate fi antrenat folosind caracteristici precum suprafața, numărul de camere, etajul, zona sau vechimea clădirii, pentru a prezice prețul de vânzare al unui apartament. Modelul învață, din mii de tranzacții anterioare, relația dintre aceste caracteristici si prețul final

Aprobarea unui împrumut (problema de clasificare): O bancă poate folosi regresia logistică, un arbore de decizie sau Random Forest pentru a decide daca aprobă sau respinge o cerere de credit, pe baza unor caracteristici precum venitul lunar, istoricul de credit, vârsta, gradul de îndatorare sau garanțiile disponibile. Rezultatul este o clasa binara (aprobat/respins), însoțită adesea de o probabilitate de rambursare.

Întrebări de autocontrol:

1. Care este diferența fundamentală dintre o problemă de regresie și una de clasificare în cadrul învățării supervizate?
2. Ce reprezintă parametrii w (ponderea/panta) și b (pragul/bias-ul) în funcția de ipoteză $h(x) = wx + b$?
3. Cum funcționează Eroarea Pătratică Medie (MSE) și de ce diferențele dintre valorile reale și cele prezise sunt ridicate la pătrat?
4. Ce rol joacă rata de învățare (α) în algoritmul Gradient Descent și ce se poate întâmpla dacă este aleasă o valoare prea mică sau prea mare?

5. De ce este necesară utilizarea funcției Sigmoid în regresia logistică și cum transformă aceasta valorile reale în probabilități?
6. În ce situații acuratețea poate fi o metrică înșelătoare și de ce s-ar prefera utilizarea preciziei sau a F1-score-ului?
7. Cum ajută conceptele de *entropie* și *câștig de informație* (Information Gain) un arbore de decizie să aleagă cea mai bună caracteristică la un nod?
8. De ce k-NN este considerat un algoritm bazat pe memorie (*lazy learning*) și cum afectează alegerea valorii lui *k* performanța modelului?
9. Ce sunt *vectorii suport* și ce rol au funcțiile *kernel* atunci când datele nu pot fi separate printr-o linie dreaptă?
10. Prin ce mecanisme reduce algoritmul Random Forest riscul de supra-antrenare (*overfitting*) în comparație cu un singur arbore de decizie?

CAPITOLUL 11. ÎNVĂȚARE NESUPERVIZATĂ

11.1 Caracteristica învățării nesupervizate

Până acum, în cadrul acestui curs, am discutat despre învățarea supervizată, în care algoritmul primește exemple însoțite de răspunsul corect (eticheta). De exemplu, pentru a prezice dacă un client va rambursa un credit, modelul este antrenat pe date istorice în care se cunoaște deja rezultatul (a rambursat / nu a rambursat).

Învățarea nesupervizată (engl. unsupervised learning) este o categorie diferită de învățare automată, în care algoritmul primește doar datele de intrare, fără nicio etichetă sau răspuns corect asociat. Sarcina algoritmului nu mai este să prezică o valoare cunoscută dinainte, ci să descopere singur o structură, un tipar sau o organizare ascunsă în date.

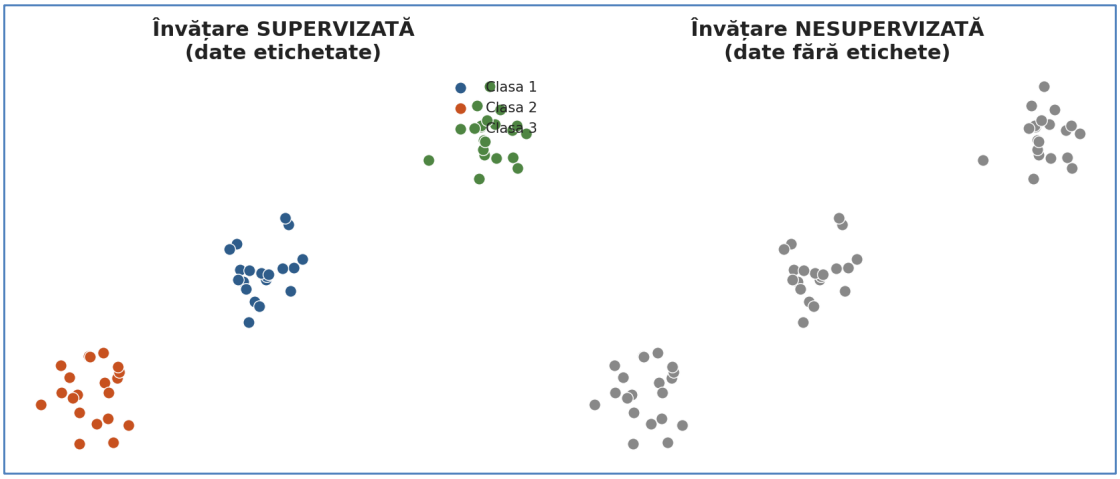


Figura 11.1 Comparație între învățarea supervizată (date etichetate, colorate pe clase) și învățarea nesupervizată (date fără etichete).

Analogie simplă

Imaginați-vă că primiți o cutie cu 100 de fotografii amestecate, fără nicio informație despre ce reprezintă. Sarcina de învățare supervizată ar fi: „Spune-mi cine este în fiecare poză", dacă ați avea deja etichete. Sarcina de învățare nesupervizată este: „Grupează pozele asemănătoare între ele", fără să știți dinainte câte grupuri există sau cum se numesc.

În practică, majoritatea datelor din lumea reală NU sunt etichetate – etichetarea manuală este costisitoare, lentă și necesită experți. Învățarea nesupervizată permite extragerea de informații utile direct din date brute:

- segmentarea clienților unei companii în funcție de comportamentul de cumpărare, fără a defini în prealabil categoriile;
- detectarea tranzacțiilor bancare neobișnuite (anomalii), care pot indica fraude;
- organizarea automată a documentelor sau articolelor pe teme similare;
- compresia și simplificarea unui volum mare de date (imagini, semnale) pentru a fi analizat mai ușor;
- explorarea inițială a unui set de date nou, înainte de a decide ce model supervizat s-ar potrivi.

Există două mari familii de algoritmi utilizați în cadrul învățării nesupervizate:

1. **Clustering** (grupare) – algoritmi care împart datele în grupuri (clustere) de observații asemănătoare între ele. Vom studia K-Means și clustering-ul ierarhic.
2. **Reducerea dimensionalității** – tehnici care simplifică datele, păstrând cât mai multă informație utilă, dar folosind mai puține variabile. Vom studia PCA (Analiza Componentelor Principale).

11.2. Concepte generale despre Clustering

Un **cluster** este un grup de observații (puncte de date) care sunt mai asemănătoare între ele decât cu observațiile din alte grupuri. Scopul unui algoritm de clustering este de a împărți automat un set de date în astfel de grupuri, pe baza unei măsuri de „asemănare" sau, echivalent, a unei măsuri de „distanță".

Cea mai simplă și des folosită măsură de distanță este **distanța euclidiană** — aceeași distanță „în linie dreaptă" pe care o cunoașteți din geometrie. Cu cât distanța dintre două puncte este mai mică, cu atât ele sunt considerate mai asemănătoare.

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \quad (1)$$

unde x și y sunt două observații (puncte de date) descrise prin n caracteristici (variabile), iar x_i și y_i sunt valorile lor pe caracteristica i. Practic, se face diferența pe fiecare caracteristică, se ridică la pătrat, se însumează, apoi se extrage rădăcina pătrată.

Totuși, în funcție de tipul datelor, se pot folosi și alte măsuri, precum **distanța Manhattan** (pentru date cu multe dimensiuni) sau **Similaritatea Cosinus** (foarte utilă pentru date textuale sau vectori de frecvență).

Tabel 11.1 Tipuri de algoritmi de clustering

Tip	Principiu	Exemplu
Bazat pe partiționare	Împarte datele în k grupuri, fiecare punct aparținând unui singur grup	K-Means
Ierarhic	Construiește o structură de tip arbore (dendrogramă) prin unirea sau divizarea treptată a grupurilor	Clustering aglomerativ
Bazat pe densitate	Formează clustere din regiuni cu densitate mare de puncte, separate de regiuni cu densitate mică	DBSCAN

***Notă:** în cadrul acestui capitol vom examina pe primele două tipuri – K-Means (partiționare) și clustering ierarhic – deoarece sunt cele mai frecvent utilizate și mai ușor de înțeles intuitiv.*

11.3. Algoritmul K-Means

K-Means este cel mai popular algoritm de clustering, datorită simplității și eficienței sale. Numele provine de la faptul că algoritmul împarte datele în K clustere (K este ales de utilizator), iar fiecare cluster este reprezentat de media (engl. mean) punctelor pe care le conține – numită **centroid**.

Algoritmul, pas cu pas

1. Se alege numărul de clustere K, dorit de utilizator.
2. Se aleg K puncte inițiale, la întâmplare, ca centroizi de start.
3. Etapa de atribuire: fiecare punct din setul de date este atribuit celui mai apropiat centroid (folosind distanța euclidiană).
4. Etapa de actualizare: fiecare centroid este recalculat ca media aritmetică a tuturor punctelor atribuite lui.
5. Pașii 3 și 4 se repetă până când centroizii nu se mai schimbă semnificativ (algoritmul a convers).

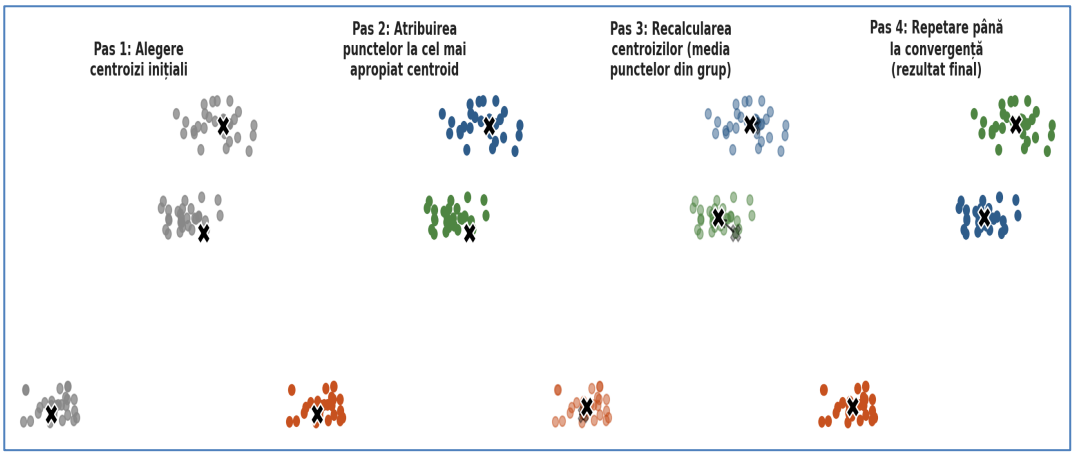


Figura 11.2 Evoluția algoritmului K-Means: de la centroizi inițiali aleși aleatoriu, la atribuirea punctelor, recalcularea centroizilor și rezultatul final.

Funcția obiectiv

K-Means încearcă, la fiecare iterație, să minimizeze suma pătratelor distanțelor dintre fiecare punct și centroidul clusterului din care face parte. Această cantitate se numește inerție (sau eroare de clustering) și este notată de obicei cu J:

$$J = \sum_{k=1}^{K} \sum_{x \in C_k} \|x - \mu_k\|^2 \quad (2)$$

unde K este numărul de clustere, C_k reprezintă mulțimea punctelor atribuite clusterului k, iar μ_k este centroidul acelui cluster. Cu cât J este mai mic, cu atât punctele din interiorul fiecărui cluster sunt mai apropiate de centrul lor – deci grupurile sunt mai „compacte”.

Formula de recalculare a centroidului la fiecare pas este pur și simplu media aritmetică a punctelor din cluster:

$$\mu_k = \frac{1}{|C_k|} \sum_{x \in C_k} x \quad (3)$$

unde $|C_k|$ reprezintă numărul de puncte din clusterul k, iar suma se face pe toate punctele x care aparțin acelui cluster.

Cum se alege numărul K de clustere? – Metoda cotului

O întrebare frecventă este: câte clustere ar trebui să alegem? Din păcate, algoritmul K-Means nu poate răspunde singur la această întrebare – K trebuie ales de utilizator. O metodă practică și des folosită este metoda cotului (engl. elbow method): se rulează K-Means pentru mai multe valori ale lui K (de exemplu 1, 2, 3 ... 8) și se reprezintă grafic inerția (J) în funcție de K. Pe măsură ce K crește, inerția scade mereu (deoarece adăugarea de noi centroizi reduce inevitabil distanța medie până la cel mai apropiat centru), dar la un moment dat scăderea devine mult mai lentă – acest punct de „cotitură” indică echilibrul optim între compactitatea clusterelor și numărul lor.

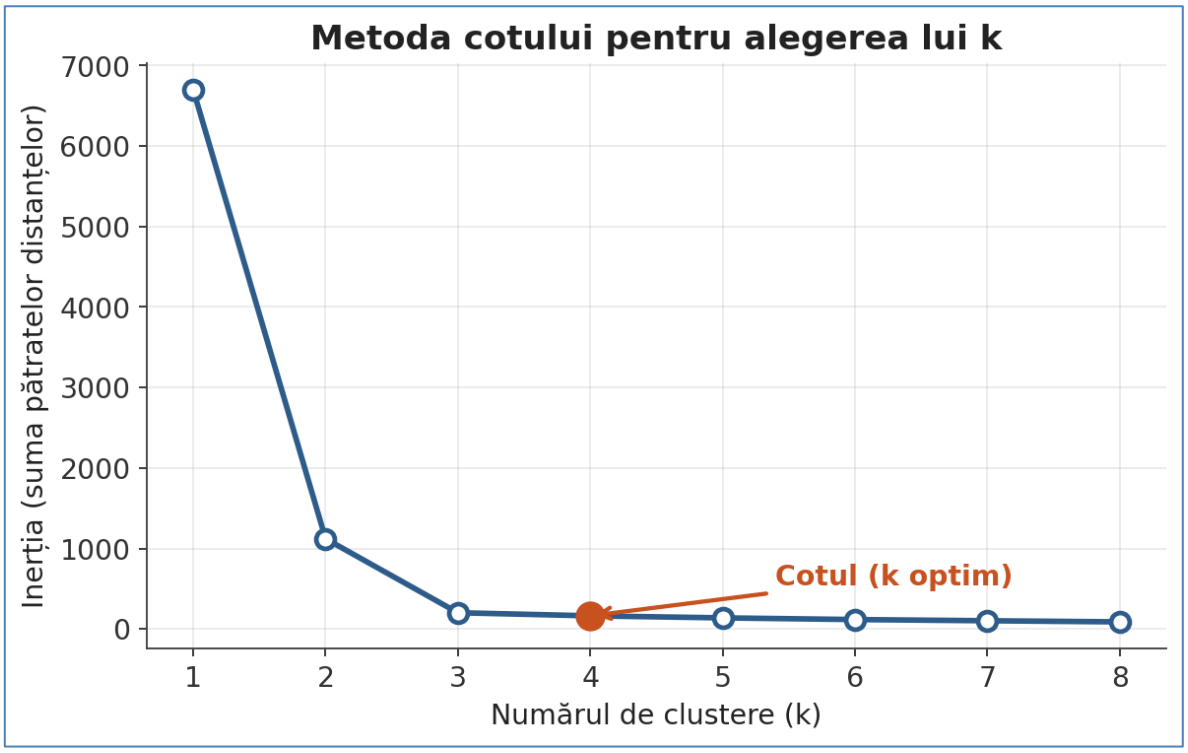


Figura 11.3 Metoda cotului: inerția scade rapid până la k=4, apoi scăderea încetinește semnificativ – sugerând că 4 este un număr potrivit de clustere.

Tabel 11.2. Avantaje și limitele ale algoritmului K-Means

Avantaje	Limite
Simplu de înțeles și rapid din punct de vedere computațional	Trebuie specificat K în avans
Funcționează foarte bine pe seturi mari de date	Sensibil la alegerea centroizilor inițiali (Soluție standard: utilizarea algoritmului K-Means++ pentru o inițializare mai bună)
Ușor de implementat și interpretat	Presupune clustere de formă aproximativ sferică și dimensiune similară
Se poate rula de mai multe ori pentru rezultate mai stabile	Sensibil la valori extreme (outlieri)
	Sensibilitate la scalarea datelor (dacă variabilele nu sunt standardizate/normalizate, cele cu intervale de valori mai mari vor domina calculul distanței euclidiană)

11.4. Clustering ierarhic

Spre deosebire de K-Means, clustering-ul ierarhic nu necesită specificarea în avans a numărului de clustere. În schimb, construiește o structură ierarhică (asemănătoare unui arbore genealogic), care arată cum se pot grupa datele la diferite niveluri de granularitate – de la fiecare punct considerat individual, până la un singur grup care conține toate datele.

Cea mai comună variantă este clustering-ul aglomerativ (de tip „jos în sus”), care funcționează astfel:

1. Se pornește cu fiecare observație tratată ca un cluster separat (dacă avem n observații, avem inițial n clustere).
2. Se identifică cele două clustere cele mai apropiate (cele mai asemănătoare) și se unesc într-un singur cluster.
3. Pasul anterior se repetă, unind mereu cele mai apropiate două clustere, până când rămâne un singur cluster ce conține toate datele.

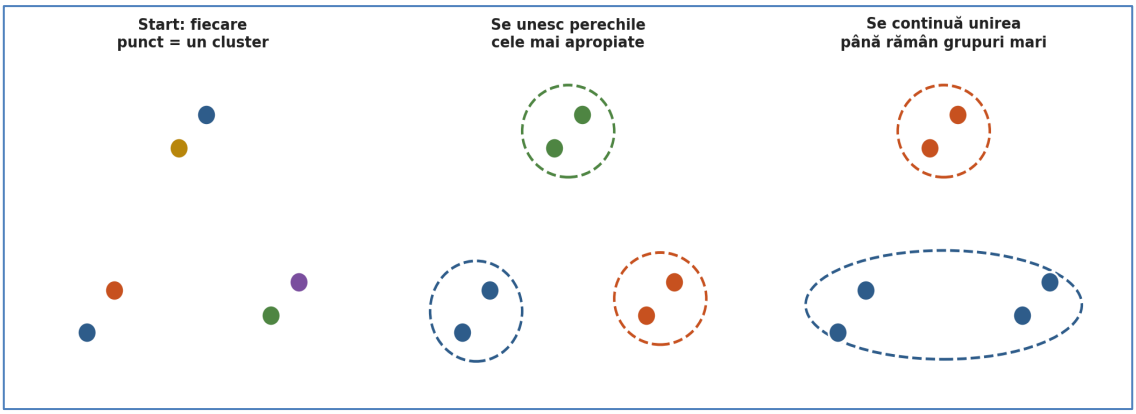


Figura 11.4 Principiul clustering-ului aglomerativ: se pornește de la puncte individuale și se unesc treptat cele mai apropiate grupuri.

Cum se măsoară distanța dintre două clustere?

Dacă distanța dintre două puncte este clară (distanța euclidiană), atunci este nevoie și de o regulă pentru distanța dintre două grupuri de puncte. Această regulă se numește **metodă de legătură** (engl. linkage):

Tabel 11.3 Metodele de legătură a clasterelor

Metodă de legătură	Cum se calculează distanța dintre 2 clustere	Explicație
Legătura simplă (single linkage)	Distanța minimă dintre oricare două puncte, câte unul din fiecare cluster.	Legătura poate produce efectul de 'lanțuire' (chaining)
Legătura completă (complete linkage)	Distanța maximă dintre oricare două puncte, câte unul din fiecare cluster	Tinde să producă clustere mai compacte și sferice
Legătura medie (average linkage)	Media tuturor distanțelor dintre punctele celor două clustere	
Metoda Ward	Alege unirea care minimizează creșterea inerției totale (similar spiritului K-Means)	Similară cu logica K-Means.

Dendrograma – vizualizarea rezultatului

Rezultatul unui clustering ierarhic se reprezintă printr-o dendrogramă – o diagramă arborescentă în care fiecare unire a două clustere este marcată printr-o linie orizontală, la o înălțime proporțională cu distanța dintre ele. Cu cât unirea are loc „mai sus”, cu atât cele două grupuri erau mai puțin asemănătoare.

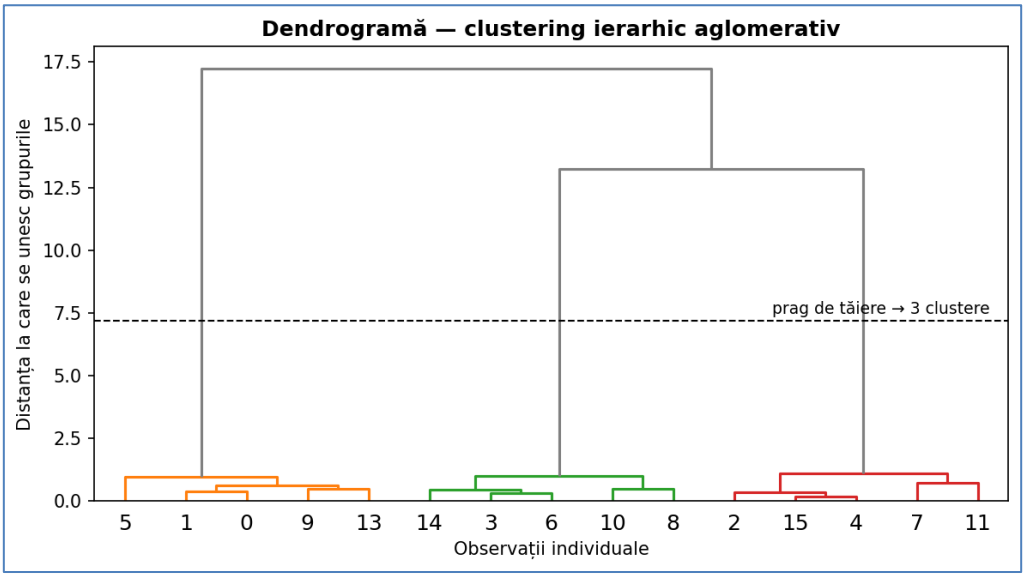


Figura 11.5 Exemplu de dendrogramă. Tăierea arborelui la un anumit prag de distanță (linia punctată) determină numărul final de clustere – în acest caz, 3.

Cum „tăiem" dendrograma?

Numărul de clustere se obține trăgând o linie orizontală imaginară la o anumită înălțime (distanță) și numărând câte ramuri verticale intersectează acea linie. O linie trasă mai sus va produce mai puține clustere, mai mari; o linie trasă mai jos va produce mai multe clustere, mai mici.

***Tabel 11.4* K-Means vs. Clustering ierarhic**

Criteriu	K-Means	Clustering ierarhic
Numărul de clustere	Trebuie specificat dinainte (K)	Se alege ulterior, prin tăierea dendrogramei
Viteză pe date mari	Rapid, scalează bine	Mai lent, greu de aplicat pe seturi foarte mari
Rezultat	Poate varia în funcție de centroizii inițiali	Determinist – produce mereu același rezultat
Vizualizare	Nu oferă o structură ierarhică	Oferă dendrogramă, foarte utilă pentru interpretare

11.5. Reducerea dimensionalității — PCA

În multe probleme reale, fiecare observație este descrisă prin zeci sau chiar sute de caracteristici (variabile) – de exemplu, expresia a mii de gene, sau pixelii unei imagini. Acest număr mare de variabile poartă numele de dimensionalitate a datelor. Un număr foarte mare de dimensiuni creează mai multe probleme:

a) este dificil (sau imposibil) să vizualizăm date cu mai mult de 2-3 dimensiuni;
b) multe caracteristici pot fi redundante sau puternic corelate între ele, aducând informație repetată;
c) algoritmii de învățare automată devin mai lenți și, uneori, mai puțin performanți atunci când numărul de variabile este foarte mare (fenomen numit „blestemul dimensionalității").

Reducerea dimensionalității este procesul de a transforma datele dintr-un spațiu cu multe variabile într-un spațiu cu mai puține variabile, păstrând totuși cât mai multă din informația (variația) originală.

PCA (Principal Component Analysis – Analiza Componentelor Principale) este cea mai populară tehnică de reducere a dimensionalității. Ideea centrală este simplă: în loc să păstrăm toate variabilele originale, PCA construiește un set nou de variabile, numite **componente principale**, care sunt **combinații liniare** ale variabilelor originale.

Prima componentă principală (CP1) este direcția din date de-a lungul căreia datele au cea mai mare variație (cea mai „împrăștiată" direcție).

A doua componentă principală (CP2) este perpendiculară pe prima și captează cea mai mare parte din variația rămasă. Acest proces continuă pentru toate componentele posibile.

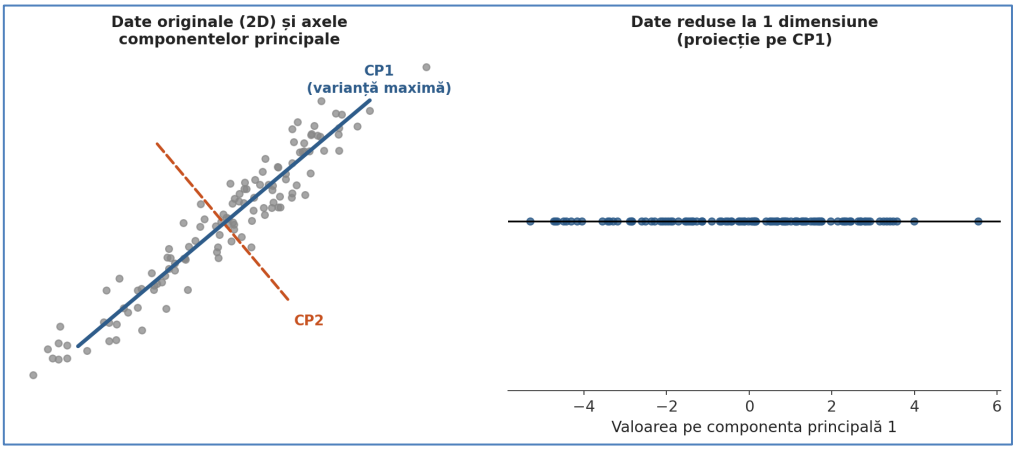


Figura 11.6 Stânga: date originale în 2 dimensiuni, cu axele celor două componente principale. Dreapta: aceleași date, reduse la o singură dimensiune prin proiecția pe CP1.

Analogie simplă

Imaginați-vă că fotografiați un creion lung, așezat oblic pe o masă. Dacă priviți creionul exact din lungul lui, el pare doar un punct – nu vedeți nimic util. Dacă îl priviți din lateral, de-a lungul axei sale, vedeți toată lungimea lui – adică cea mai multă informație posibil dintr-o singură direcție de observare. Acea direcție optimă de observare este, intuitiv, prima componentă principală.

Fiecare componentă principală explică un anumit procent din variația totală a datelor. De obicei alegem un număr de componente suficient cât să acopere un procent ridicat din variația totală (de exemplu 90-95%). Această proporție se numește **varianță explicată**:

$$Varianță\ explicată(CPk) = \frac{\lambda_k}{\lambda_1+\lambda_2+\cdots+\lambda_p} \quad (4)$$

unde λ_1, λ_2, ... λ_p reprezintă variația (valorile proprii) asociate fiecărei componente principale, ordonate descrescător. Se observă că prima componentă are întotdeauna cea mai mare pondere.

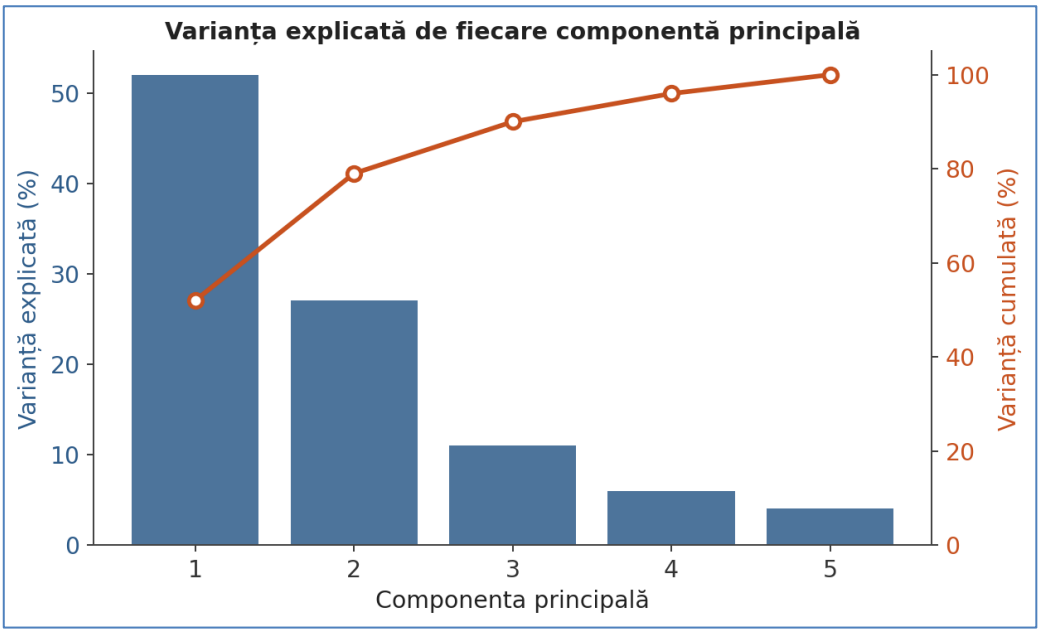


Figura 11.7 Exemplu de grafic al varianței explicate : barele arată varianța fiecărei componente, iar linia arată varianța cumulată.

Pașii algoritmului PCA (pe scurt)

1) Se standardizează datele (fiecare variabilă este adusă la aceeași scală), pentru ca variabilele cu valori mari să nu domine artificial rezultatul.
2) Se calculează matricea de covarianță (dacă datele sunt standardizate, aceasta devine matricea de corelație), care arată cum variază împreună fiecare pereche de variabile.
3) Se determină componentele principale (direcțiile de variație maximă), prin descompunerea matricei de covarianță.
4) Se ordonează componentele descrescător, în funcție de cât de multă variație explică fiecare.
5) Se aleg primele câteva componente (care explică cea mai multă variație) și se proiectează datele pe acestea, obținând o versiune redusă a setului de date.

Aplicații ale PCA

- vizualizarea datelor cu multe variabile, prin reducerea la 2 sau 3 dimensiuni;
- eliminarea redundanței și a zgomotului din date, înainte de a antrena un alt model;
- compresia imaginilor, păstrând doar componentele cele mai informative;
- accelerarea algoritmilor de învățare automată, prin reducerea numărului de variabile de intrare.

Legătura dintre clustering și PCA

În practică, aceste tehnici sunt adesea combinate: atunci când datele au foarte multe variabile, se aplică mai întâi PCA pentru a reduce dimensionalitatea, apoi se aplică K-Means sau clustering ierarhic pe datele reduse. Acest lucru face algoritmul de clustering mai rapid și, uneori, mai stabil, deoarece se elimină zgomotul din variabilele mai puțin informative.

Întrebări de autocontrol

1. Care este diferența fundamentală dintre învățarea supervizată și cea nesupervizată?
2. De ce este importantă alegerea numărului K în K-Means și cum ne poate ajuta metoda cotului?
3. Ce reprezintă o dendrogramă și cum se poate obține un anumit număr de clustere din ea?
4. Care este intuiția din spatele primei componente principale în PCA?
5. În ce situații practice ar fi utilă combinarea PCA cu un algoritm de clustering?

6. Ce reprezintă distanța euclidiană și cum este definită matematic pentru două puncte cu n caracteristici?
7. Care sunt pașii principali parcurși de algoritmul K-Means din momentul inițializării până la atingerea convergenței?
8. Ce reprezintă inerția (J) în cadrul algoritmului K-Means și ce ne indică o valoare mai mică a acesteia?
9. Prin ce se diferențiază clustering-ul ierarhic aglomerativ de algoritmul K-Means în ceea ce privește stabilirea numărului de clustere?
10. Care este diferența dintre metoda legăturii simple (single linkage) și cea a legăturii complete (complete linkage) în clustering-ul ierarhic?
11. Ce este „blestemul dimensionalității” și ce proprietate are prima componentă principală (CP1) în metoda PCA?
12. De ce este necesară standardizarea datelor înainte de a aplica algoritmul PCA?

CAPITOLUL 12. REȚELE NEURONALE ARTIFICIALE

12.1. Perceptronul.

Rețelele neuronale artificiale (RNA) sunt modele de calcul inspirate din modul în care funcționează creierul uman. Creierul este format din miliarde de neuroni biologici, interconectați prin conexiuni numite **sinapse**. Fiecare neuron primește semnale de la alți neuroni prin niște prelungiri numite **dendrite**, „prelucrează” aceste semnale în corpul celular și, dacă semnalul total depășește un anumit **prag**, transmite mai departe un impuls electric prin **axon** către alți neuroni.

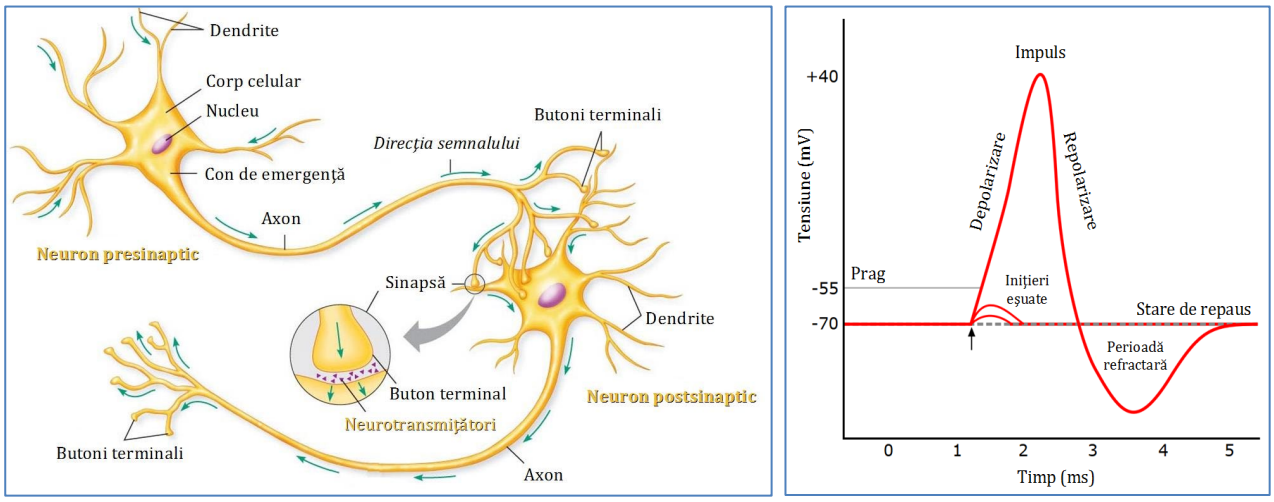


Figura 12.1 Neuronul biologic

O rețea neuronală artificială preia această idee la nivel simplificat, matematic. În loc de neuroni biologici, avem unități de calcul numite neuroni artificiali (sau noduri). În loc de sinapse cu „tărie” variabilă, avem ponderi numerice (weights) care se pot ajusta. Ideea centrală a domeniului este următoarea:

> 💡 *O rețea neuronală artificială nu este „programată” explicit să rezolve o problemă — ea învață din exemple, ajustându-și ponderile astfel încât erorile pe care le face să scadă treptat.*

Analogia dintre neuronul biologic și cel artificial este redată în schema de mai jos:

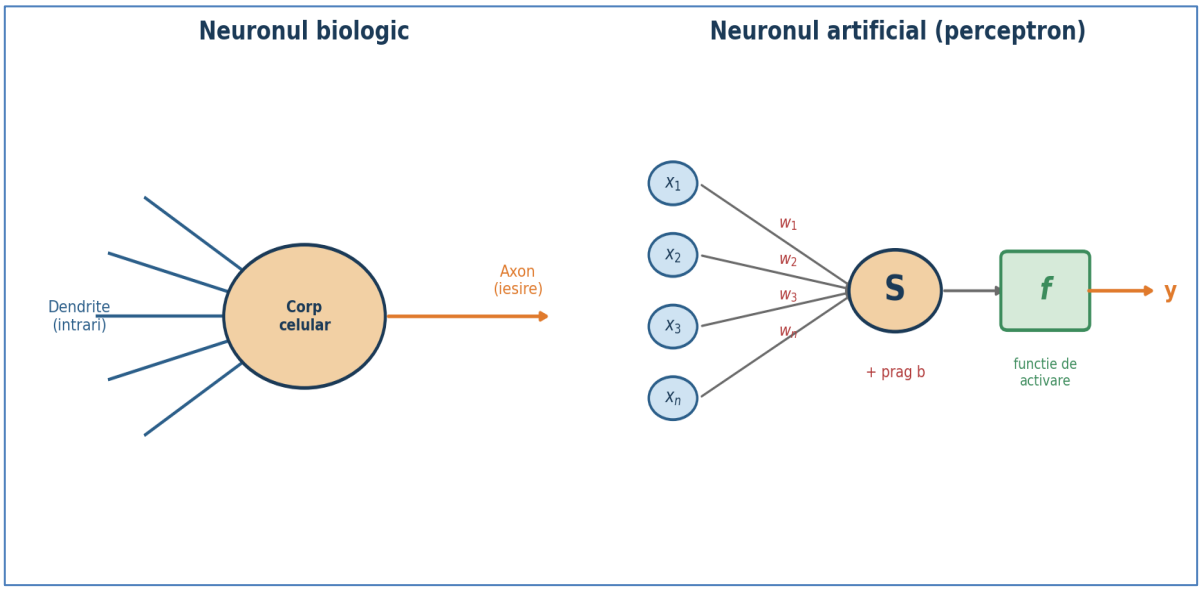


Figura 12.2 Analogia dintre neuronul biologic și neuronul artificial (perceptron)

Componentele-cheie ale corespondenței sunt:

1. dendritele ↔ intrările (x_1, x_2, …, x_n), adică datele pe care neuronul le primește;
2. tăria sinapsei ↔ ponderea (w_i), care indică cât de important este fiecare semnal de intrare;
3. pragul de activare al neuronului ↔ pragul artificial (bias, notat b);
4. impulsul transmis pe axon ↔ ieșirea neuronului artificial (y), transmisă mai departe.

Perceptronul este cel mai simplu model de neuron artificial, propus de Frank Rosenblatt în anul 1958. El stă la baza tuturor rețelelor neuronale moderne – chiar dacă arhitecturile actuale sunt mult mai complexe, principiul de funcționare al unui singur neuron rămâne, în esență, același.

Un perceptron primește mai multe valori de intrare (x_1, x_2, …, x_n), fiecare fiind înmulțită cu o pondere corespunzătoare (w_1, w_2, …, w_n). Rezultatele se însumează, se adaugă un termen de prag (bias, b), iar suma obținută trece printr-o funcție de activare care produce ieșirea finală.

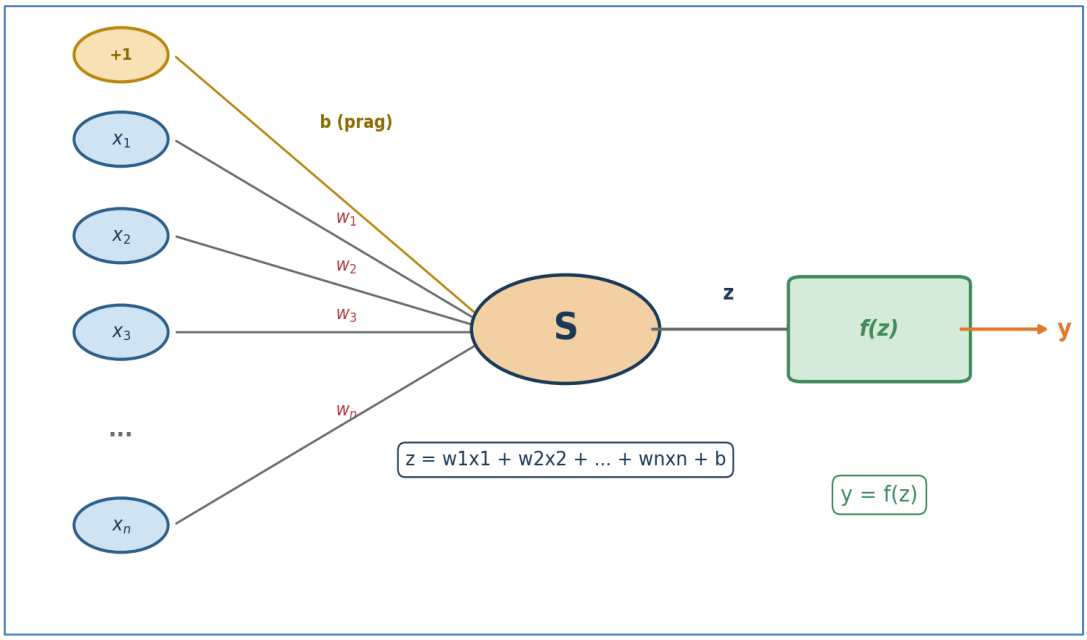


Figura 12.3 Structura detaliată a unui perceptron

Formula de calcul

Pasul 1 – se calculează suma ponderată a intrărilor, la care se adaugă pragul b:

$$z = \sum_{i=1}^{n} w_i x_i + b \quad (1)$$

unde n este numărul de intrări, x_i este valoarea intrării i, w_i este ponderea asociată intrării i, iar b este pragul (bias-ul) neuronului.

Pasul 2 – rezultatul z trece prin funcția de activare f, obținându-se ieșirea neuronului:

$$y = f(z) = f(\sum_{i=1}^{n} w_i x_i + b) \quad (2)$$

> 💡 *Bias-ul b poate fi văzut ca o intrare suplimentară, mereu egală cu 1, înmulțită cu o pondere proprie. Rolul lui este să „deplaseze" pragul de activare – fără el, neuronul ar fi forțat să treacă mereu prin origine.*

Interpretare intuitivă (fără formule)

Gândiți-vă la perceptron ca la un „votant ponderat". Fiecare intrare este un argument pro sau contra unei decizii. Ponderea arată cât de convingător este argumentul respectiv: o pondere mare înseamnă „acest argument contează mult", o pondere mică sau negativă înseamnă „acest argument contează puțin" sau chiar „acest argument pledează împotrivă". Suma tuturor argumentelor ponderate decide, în final, dacă neuronul se „activează" (ex. răspunde cu 1) sau nu (răspunde cu 0).

Exemplu simplu, aplicat: să presupunem că un perceptron decide dacă un student va promova un examen, pe baza a două intrări: numărul de ore de studiu (x_1) și numărul de absențe (x_2). O pondere pozitivă mare pentru x_1 arată că studiul contează mult în sens pozitiv, iar o pondere negativă pentru x_2 arată că absențele scad șansele de promovare.

Geometric, fără bias, hiperplanul de decizie este forțat să treacă prin originea sistemului de coordonate (0,0). Bias-ul permite deplasarea acestui hiperplan în orice direcție, oferind rețelei libertatea de a 'ajusta' pragul de decizie pe orice punct din spațiu.

12.2. Funcția de activare

Funcția de activare este cea care decide „cât de tare” se activează un neuron, pornind de la suma ponderată z. Fără o funcție de activare neliniară, o rețea neuronală – indiferent de câte straturi ar avea – s-ar comporta matematic ca un simplu model liniar, incapabil să învețe relații complexe din date.

> 💡 *Rolul funcției de activare este să introducă neliniaritate în rețea. Acest lucru este esențial: lumea reală (imagini, text, vorbire, decizii economice) este guvernată de relații neliniare, iar rețelele neuronale trebuie să le poată aproxima.*

Cele mai utilizate patru funcții de activare sunt prezentate mai jos, împreună cu formulele și graficele lor.

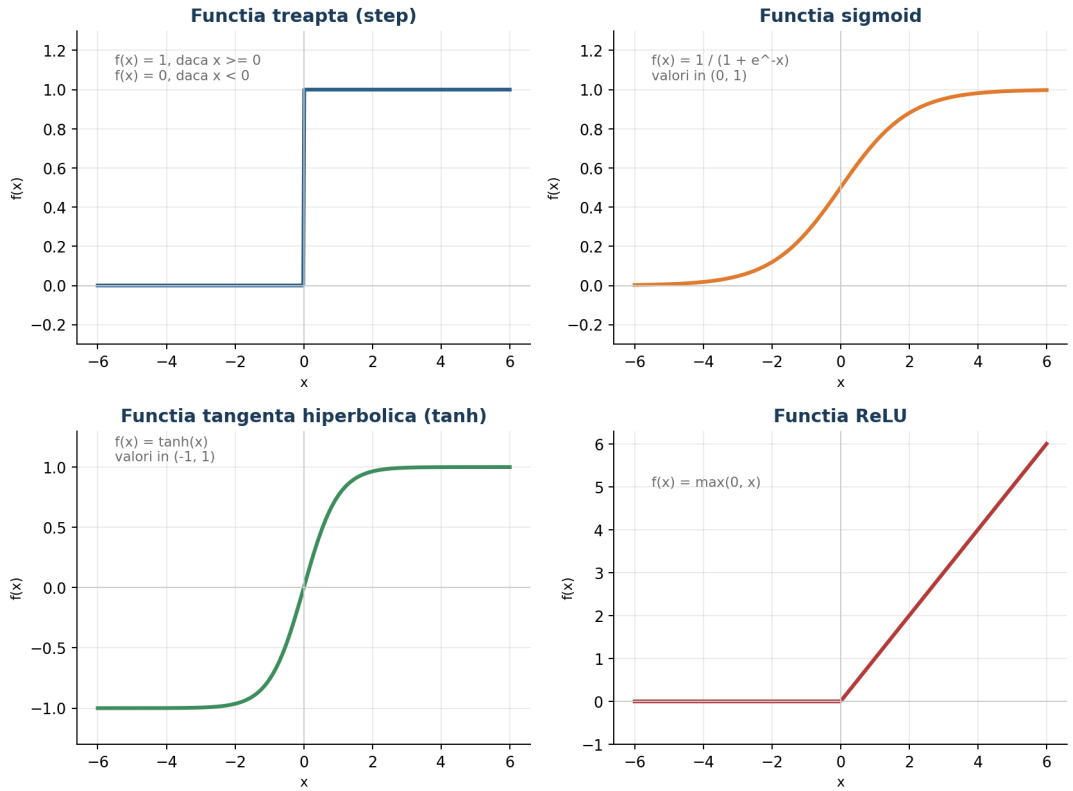


Figura 12.4 Cele patru funcții de activare uzuale: treaptă, sigmoid, tanh și ReLU

Funcția treaptă (step function)

Este funcția originală, folosită de Rosenblatt în perceptronul clasic. Produce doar două valori posibile – 0 sau 1 – în funcție de semnul lui z:

💡 *Dezavantaj: funcția treaptă are un „salt” brusc și nu este derivabilă în punctul z = 0, ceea ce o face inutilizabilă pentru algoritmi de învățare bazați pe gradient, precum backpropagation.*

Funcția sigmoid

Funcția sigmoid „comprimă” orice valoare reală într-un interval continuu (0, 1), ceea ce o face utilă pentru a interpreta ieșirea ca pe o probabilitate. Este netedă și derivabilă peste tot, motiv pentru care a fost multă vreme funcția standard în rețelele neuronale.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

Deși funcția Sigmoid este utilă pentru probabilități, ea suferă de problema vanishing gradient (disparirea gradientului) în rețelele profunde (Deep Learning). Deoarece derivata sa este foarte mică pentru valori mari de z, gradientul 'dispare" în timpul backpropagation, împiedicând ponderile din straturile inițiale să se învețe. ReLU rezolvă această problemă pentru toate valorile pozitive.

Funcția tangentă hiperbolică (tanh)

Similară cu sigmoid ca formă, dar produce valori în intervalul (-1, 1), fiind centrată în jurul lui 0. Acest lucru ajută adesea la o convergență mai rapidă a antrenării, comparativ cu sigmoid.

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{4}$$

Funcția ReLU (Rectified Linear Unit)

ReLU este cea mai folosită funcție de activare în rețelele neuronale profunde (Deep Learning) moderne. Este extrem de simplă din punct de vedere computațional: dacă intrarea este pozitivă, o lasă neschimbată; dacă este negativă, o transformă în 0.

$$f(x) = max(0, x) \tag{5}$$

Tabel 12.1 Funcții de activare

Functie	Formula (concept)	Domeniu valori	Utilizare tipica
Treapta (step)	1 daca z ≥ 0, altfel 0	{0, 1}	Perceptronul clasic, clasificare binara simpla
Sigmoid	1 / (1 + e^-z)	(0, 1)	Strat de iesire pentru probabilitati, retele mai vechi
Tanh	tangenta hiperbolica de z	(-1, 1)	Straturi ascunse, converge adesea mai rapid ca sigmoid
ReLU	max(0, z)	[0, ∞)	Straturi ascunse in retele profunde moderne (Deep Learning)

12.3. Rețele neuronale multistrat (Multi-Layer Perceptron)

Un singur perceptron poate rezolva doar probleme **liniar separabile** – adică probleme în care punctele din cele două clase pot fi separate printr-o singură linie dreaptă (în 2D) sau printr-un plan/hiperplan (în dimensiuni mai mari).

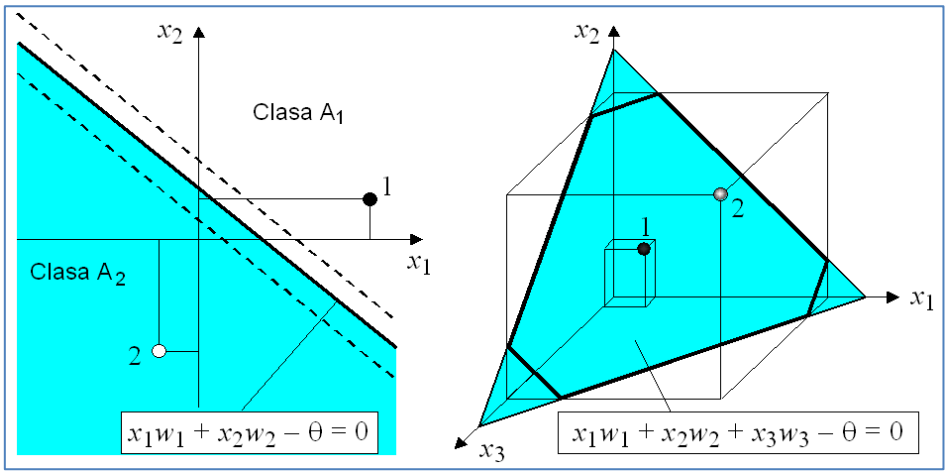


Figura 12.5 Hiperplanul problemelor linear separabile în spații 2D și 3D

Exemplul clasic care demonstrează limitarea perceptronului simplu este funcția logică XOR (SAU exclusiv). Tabelul de adevăr al funcției XOR este:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Dacă am reprezenta grafic aceste patru puncte, am observa că nu există nicio dreaptă care să separe punctele cu rezultat 0 de cele cu rezultat 1. Prin urmare, niciun set de ponderi (w_1, w_2, b) nu poate face un singur perceptron să învețe corect funcția XOR.

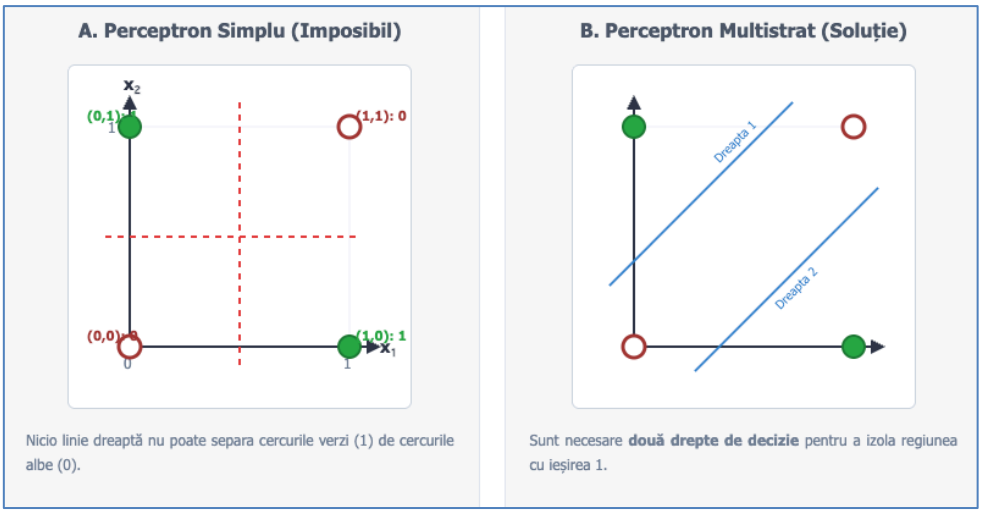


Figura 12.6 Limitările perceptronului simplu

> 💡 *Această limitare a fost semnalată în 1969 de Marvin Minsky și Seymour Papert și a dus, pentru o vreme, la scăderea interesului pentru rețelele neuronale. Soluția a venit prin combinarea mai multor perceptroni în straturi – rețelele neuronale multistrat (Multi-Layer Perceptron, MLP).*

O rețea multistrat introduce unul sau mai multe straturi ascunse (hidden layers) între stratul de intrare și cel de ieșire. Fiecare strat ascuns aplică propria transformare neliniară, iar combinația acestor transformări permite rețelei să învețe granițe de decizie mult mai complexe, inclusiv funcția XOR.

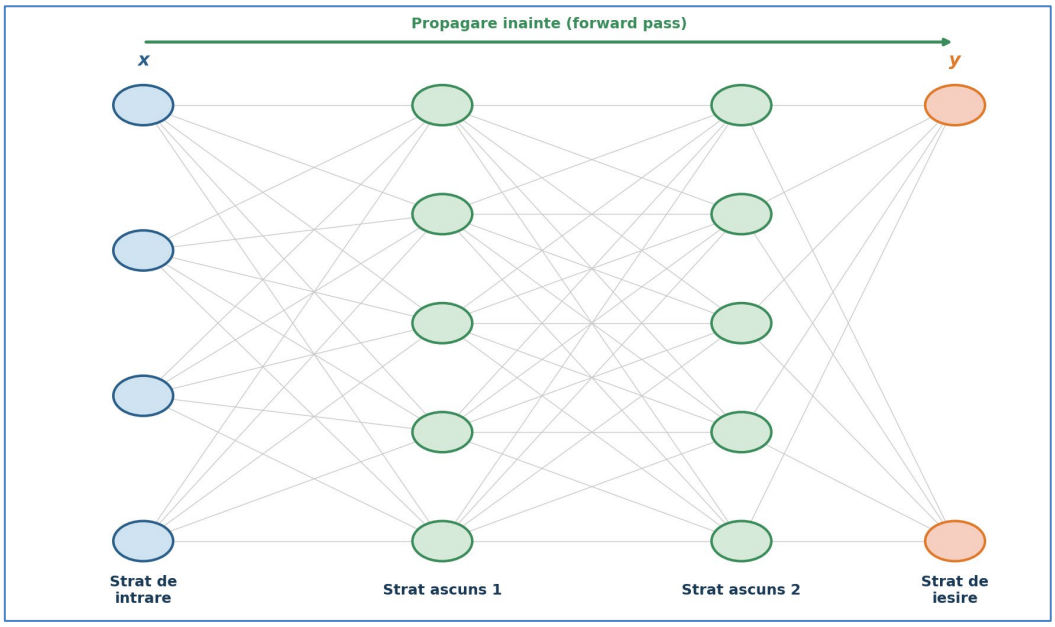


Figura 12.7 Arhitectura unei rețele neuronale multistrat (MLP): strat de intrare, două straturi ascunse și strat de ieșire

Întrebarea firească este: dacă rețeaua are mai multe straturi de ponderi, cum le ajustăm pe toate simultan, astfel încât eroarea finală să scadă? Răspunsul este algoritmul de propagare inversă, pe care îl vom studia în continuare.

12.4. Algoritmul de propagare inversă

Backpropagation (propagarea inversă a erorii) este algoritmul standard prin care o rețea neuronală multistrat își ajustează ponderile pe baza erorilor pe care le face. A fost popularizat în 1986 de Rumelhart, Hinton și Williams și rămâne, în esență, algoritmul folosit și astăzi pentru antrenarea rețelelor neuronale – de la modele simple, până la rețelele profunde folosite în AI modern.

Ideea generală poate fi rezumată în două etape care se repetă, ciclic, pentru fiecare exemplu (sau grup de exemple) din setul de antrenare:

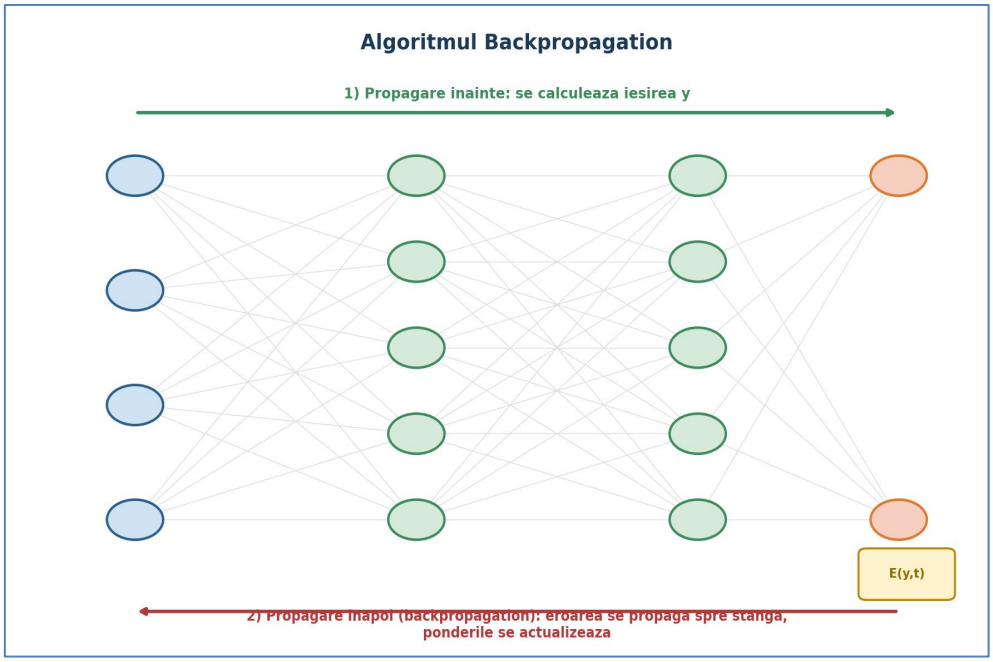


Figura 12.8 Cele două etape ale algoritmului backpropagation: propagarea înainte și propagarea înapoi

Etapa 1 – Propagarea înainte (Forward Pass)

Datele de intrare (x) sunt introduse în rețea și trec, strat cu strat, prin toți neuronii, până la stratul de ieșire. Fiecare neuron calculează suma ponderată a intrărilor sale și aplică funcția de activare, exact conform formulelor (1) și (2) discutate anterior. La finalul acestei etape, rețeaua produce o ieșire y, pe care o putem compara cu rezultatul așteptat (numit țintă sau etichetă, notat t).

Măsurarea erorii

Diferența dintre ce a prezis rețeaua (y) și ce ar fi trebuit să prezică (t) se numește **eroare**. O formă comună de a măsura eroarea totală, pentru toți neuronii de ieșire, este eroarea medie pătratică:

$$E = \frac{1}{2}\sum_k (t_k - y_k)^2 \tag{6}$$

unde t_k este valoarea așteptată (țintă) pentru neuronul de ieșire k, iar y_k este valoarea efectiv produsă de rețea pentru acel neuron. Factorul ½ este ales din comoditate matematică – simplifică derivata folosită mai jos.

Totuși, în probleme de clasificare, se folosește adesea și Cross-Entropy Loss, deoarece aceasta oferă un gradient mai clar pentru funcțiile de activare precum Sigmoid.

Propagarea înapoi (Backward Pass)

Aici intervine ideea centrală a algoritmului: eroarea calculată la ieșire este „propagată înapoi", strat cu strat, spre stratul de intrare, pentru a determina cât de mult a contribuit fiecare pondere din rețea la eroarea finală. Acest lucru se realizează matematic folosind regula lanțului (chain rule) din calculul diferențial, dar putem înțelege procesul intuitiv, pas cu pas.

- **Pasul A** – se calculează eroarea (numită „delta") pentru fiecare neuron din stratul de ieșire:

$$\delta_k = (t_k - y_k) \cdot f'(z_k) \tag{7}$$

Aici, $f'(z_k)$ este derivata funcției de activare, calculată în punctul z_k. Ea ne spune cât de „sensibil" este neuronul la mici schimbări ale intrării sale – cu alte cuvinte, cât de mult ar trebui ajustată ieșirea pentru a reduce eroarea.

- **Pasul B** – eroarea se propagă înapoi către straturile ascunse. Pentru un neuron ascuns j, eroarea sa (δ_j) depinde de erorile tuturor neuronilor din stratul următor la care este conectat, ponderate corespunzător:

$$\delta_j = f'(z_j) \cdot \sum_k \delta_k \cdot w_{jk} \tag{8}$$

> 💡 *Aceasta este esența numelui algoritmului: eroarea „curge" de la stratul de ieșire înapoi spre straturile ascunse, exact în sens opus fluxului normal de date (care merge de la intrare spre ieșire).*

Pasul C – odată cunoscută eroarea δ pentru fiecare neuron, putem calcula cât de mult ar trebui modificată fiecare pondere w_{ij}, pentru a reduce eroarea totală:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j \cdot x_i \tag{9}$$

Actualizarea ponderilor (regula de învățare)

Cunoscând gradientul erorii în raport cu fiecare pondere, ponderile se actualizează în direcția opusă gradientului – adică în direcția în care eroarea scade cel mai rapid. Această tehnică generală se numește coborâre pe gradient (gradient descent):

$$w_{nou} = w_{vechi} - \eta \cdot \frac{\partial E}{\partial w} \tag{10}$$

unde η (litera grecească eta) este **rata de învățare** (learning rate) – un număr mic, de regulă între 0,001 și 0,5, care controlează cât de mari sunt pașii de ajustare a ponderilor.

Aplicată concret pentru o singură pondere w_{ij} dintre neuronul i și neuronul j, regula devine:

$$w_{ij}(t+1) = w_{ij}(t) + \eta \cdot \delta_j \cdot x_i \qquad (11)$$

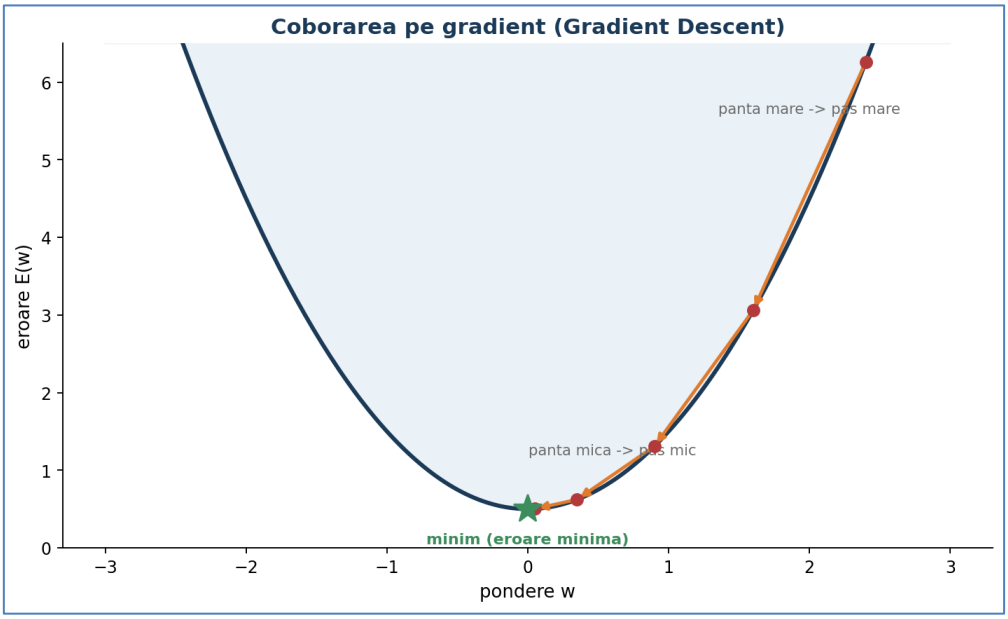


Figura 12.9. Coborârea pe gradient: ponderea se ajustează pas cu pas spre minimul funcției de eroare

💡 ***Rata de învățare η*** *este esențială: dacă este prea mare, algoritmul poate „sări" peste minimul erorii și poate deveni instabil; dacă este prea mică, antrenarea devine extrem de lentă. Alegerea unei valori potrivite pentru η este una dintre deciziile practice importante atunci când antrenăm o rețea neuronală.*

Rezumatul algoritmului, pas cu pas

1. Se inițializează ponderile rețelei cu valori mici, aleatoare, asigurându-se că nu sunt toate identice (pentru a evita problema simetriei în timpul antrenării)
2. Forward pass: se introduce un exemplu de antrenare și se calculează ieșirea y a rețelei.
3. Se calculează eroarea dintre ieșirea obținută y și valoarea așteptată t.
4. Backward pass: eroarea se propagă înapoi, strat cu strat, calculându-se δ pentru fiecare neuron.
5. Se calculează gradientul erorii pentru fiecare pondere din rețea.
6. Se actualizează toate ponderile, folosind regula de coborâre pe gradient.
7. Pașii 2–6 se repetă pentru toate exemplele din setul de antrenare, de mai multe ori (mai multe „epoci"), până când eroarea totală devine suficient de mică.

Exercițiu practic pentru studiu individual

Considerați un perceptron simplu cu două intrări (x_1, x_2), ponderile $w_1 = 0,5$ și $w_2 = -0,3$, și pragul $b = 0,1$. Calculați, folosind formula (1) și funcția de activare sigmoid (formula 3), ieșirea perceptronului pentru intrările $x_1 = 1$ și $x_2 = 2$. Discutați apoi cum s-ar modifica ponderea w_2 după o singură iterație de backpropagation, dacă valoarea țintă așteptată era $t = 1$.

Întrebări de autoevaluare

1. Care este analogia dintre componentele unui neuron biologic și cele ale unui perceptron (neuron artificial)?
2. Cum funcționează un perceptron și care sunt pașii prin care acesta procesează datele de intrare pentru a obține o ieșire (y)?
3. Care este rolul bias-ului (pragului) b în formula unui neuron artificial?
4. De ce este necesară o funcție de activare neliniară într-o rețea neuronală?
5. Care este principala problemă a funcției treaptă (*step*) în ceea ce privește antrenarea rețelelor moderne și de ce nu poate fi folosită în algoritmul backpropagation?
6. Ce este problema *vanishing gradient* (dispariția gradientului), la ce funcție de activare apare și prin ce funcție este depășită în rețelele de Deep Learning?
7. Ce sunt rețelele neuronale multistrat (MLP) și cum reușesc ele să rezolve probleme care nu sunt liniar separabile?
8. Ce se întâmplă dacă rata de învățare η este aleasă prea mare? Dar dacă este prea mică?
9. Explicați, cu propriile cuvinte, de ce un singur perceptron nu poate învăța funcția logică XOR.
10. În ce constă diferența dintre „propagarea înainte” și „propagarea înapoi” într-o rețea neuronală?

CAPITOLUL 13. ÎNVĂȚAREA PROFUNDĂ (DEEP LEARNING)

13.1. Deep Learning ca ramură a Machine Learning

Învățarea profundă (Deep Learning, DL) este o ramură a învățării automate (Machine Learning) care folosește **rețele neuronale artificiale cu multe straturi** – de aici și denumirea „profundă”. Ideea de bază nu este nouă: rețelele neuronale sunt studiate încă din anii 1950, însă abia în ultimul deceniu au devenit cu adevărat performante și utilizate pe scară largă.

Diferența esențială față de învățarea automată „clasică” (de exemplu, arborii de decizie sau regresia logistică, discutate în capitolele anterioare) constă în modul în care sunt obținute caracteristicile (features) folosite pentru a lua o decizie.

Extragerea manuală vs. extragerea automată a caracteristicilor

La algoritmii clasici de Machine Learning, un expert uman trebuie să decidă ce caracteristici sunt relevante pentru problemă (de exemplu, pentru a recunoaște o pisică într-o imagine, cineva ar trebui să definească manual reguli despre formă, textura blănii, poziția urechilor etc.). Acest proces se numește „*feature engineering*” și este costisitor, lent și greu de generalizat.

Rețelele de învățare profundă, în schimb, învață singure, direct din datele brute (de exemplu, pixelii unei imagini), care sunt caracteristicile relevante pentru sarcina respectivă. Acest lucru este posibil datorită arhitecturii pe mai multe straturi: fiecare strat învață reprezentări din ce în ce mai abstracte, pornind de la caracteristici simple (margini, culori) până la concepte complexe (obiecte întregi).

Tabelul 13.1 Machine Learning clasic versus Deep Learning

Criteriu	Machine Learning clasic	Deep Learning
Extragerea caracteristicilor	Manuală, realizată de expert	Automată, învățată din date
Cantitatea de date necesară	Funcționează bine cu date puține	Necesită, de regulă, seturi mari de date
Resurse de calcul	Modeste (CPU)	Ridicate, adesea necesită GPU
Interpretabilitate	În general mai ușor de interpretat	Adesea „cutie neagră”, greu de interpretat
Exemple de algoritmi	Arbori de decizie, SVM, regresie logistică	CNN, RNN, Transformer

Cauzele care au determinat utilizarea în masă a Deep Learning-ului în ultimul deceniu sunt:

- Volumul mare de date disponibile (Big Data) – internetul oferă miliarde de imagini, texte și înregistrări etichetate;
- Puterea de calcul – plăcile video (GPU) permit efectuarea în paralel a milioanelor de calcule necesare antrenării;
- Progrese algoritmice – arhitecturi noi (CNN, Transformer) și tehnici de antrenare mai stabile (ReLU, Batch Normalization, Dropout).

> 💡 *Rețineți: Deep Learning nu înlocuiește complet Machine Learning-ul clasic. Pentru probleme cu date tabelare, puține exemple sau nevoie de interpretabilitate ridicată, metodele clasice rămân adesea alegerea potrivită.*

Înainte de a discuta rețelele convoluționale, reamintim pe scurt cum funcționează unitatea de bază a oricărei rețele neuronale: neuronul artificial (perceptronul).

Un neuron artificial primește una sau mai multe valori de intrare (x_1, x_2, …, x_n), le înmulțește fiecare cu o „pondere” (weight) care indică importanța acelei intrări, adună rezultatele împreună cu un termen liber (bias) și aplică apoi o funcție de activare care introduce neliniaritate:

$$y = f(\Sigma_{i=1}^{n} w_i x_i + b) \quad (1)$$

unde: x_i — valorile de intrare, w_i — ponderile, b — termenul liber (bias), f — funcția de activare

O rețea neuronală „feedforward” (cu propagare înainte) este formată din mai multe straturi de astfel de neuroni, conectați complet unul de celălalt (fiecare neuron dintr-un strat este conectat la toți neuronii din stratul următor) – de aceea aceste straturi se mai numesc și „dense” sau „fully connected”.

Antrenarea unei astfel de rețele înseamnă ajustarea treptată a ponderilor w_i astfel încât rețeaua să producă rezultate cât mai apropiate de cele așteptate, folosind algoritmul de retropropagare a erorii (backpropagation).

O întrebare firească este: de ce nu folosim pur și simplu rețelele „fully connected” (dense), descrise mai sus, direct pe imagini? Răspunsul are două componente principale.

Problema 1: numărul uriaș de parametri

O imagine color de dimensiuni modeste, de exemplu 224×224 pixeli cu 3 canale de culoare (roșu, verde, albastru), are:

$$224 \times 224 \times 3 = 150\,528\ valori\ de\ intrare \quad (2)$$

Dacă primul strat „dense” ar avea, de exemplu, doar 1000 de neuroni, ar fi nevoie de peste 150 de milioane de ponderi doar pentru acest prim strat. Un asemenea număr de parametri face rețeaua extrem de greu de antrenat, foarte predispusă la supra-antrenare (overfitting) și costisitoare din punct de vedere computațional.

Problema 2: pierderea structurii spațiale

Pentru a introduce o imagine într-un strat „dense”, aceasta trebuie transformată dintr-o matrice bidimensională (înălțime × lățime × canale) într-un singur vector lung de numere (flatten). În acest proces se pierde informația despre vecinătatea pixelilor: rețeaua nu mai „știe” că doi pixeli erau alăturați în imaginea originală, deși tocmai relațiile spațiale locale (contururi, texturi, forme) sunt esențiale pentru recunoașterea vizuală.

13.2. Rețele neuronale convoluționale

Rețelele convoluționale (***Convolutional Neural Networks, CNN***) rezolvă ambele probleme printr-o idee simplă, dar puternică: în loc să conecteze fiecare neuron la toți pixelii imaginii, folosesc un filtru (nucleu) mic, care „alunecă” peste întreaga imagine și analizează, pe rând, doar câte o regiune locală.

- **Conectivitate locală** – fiecare neuron „vede” doar o mică regiune din imagine (de exemplu 3×3 pixeli), nu întreaga imagine;
- **Partajarea ponderilor** (*weight sharing*) – același filtru (aceleași ponderi) este folosit în toate pozițiile imaginii, ceea ce reduce drastic numărul de parametri;
- **Invarianță la translație** – dacă un filtru învață să detecteze, de exemplu, o margine verticală, el o va detecta indiferent de poziția acesteia în imagine.

> 💡 *Analogie utilă: gândiți-vă la un filtru convoluțional ca la o „lupă” care se plimbă peste imagine, căutând peste tot același tip de tipar (de exemplu, o margine, un colț, o pată de culoare).*

Arhitectura tipică a unei rețele CNN alternează straturi de convoluție cu straturi de pooling, urmate la final de unul sau mai multe straturi complet conectate care realizează clasificarea propriu-zisă:

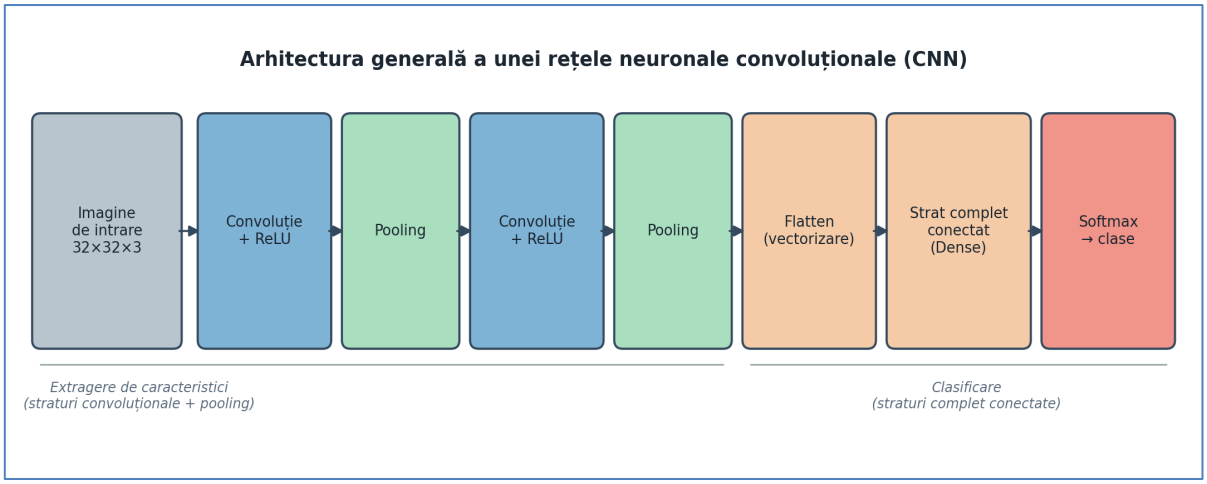


Figura 13.1 Arhitectura generală a unei rețele neuronale convoluționale (CNN)

Cum funcționează operația de convoluție?

Un filtru (kernel) este o mică matrice de numere (de exemplu 3×3), ale cărei valori sunt învățate în timpul antrenării. Acest filtru „alunecă” peste imaginea de intrare, iar la fiecare poziție se calculează produsul element-cu-element dintre filtru și porțiunea de imagine acoperită, urmat de însumarea rezultatelor:

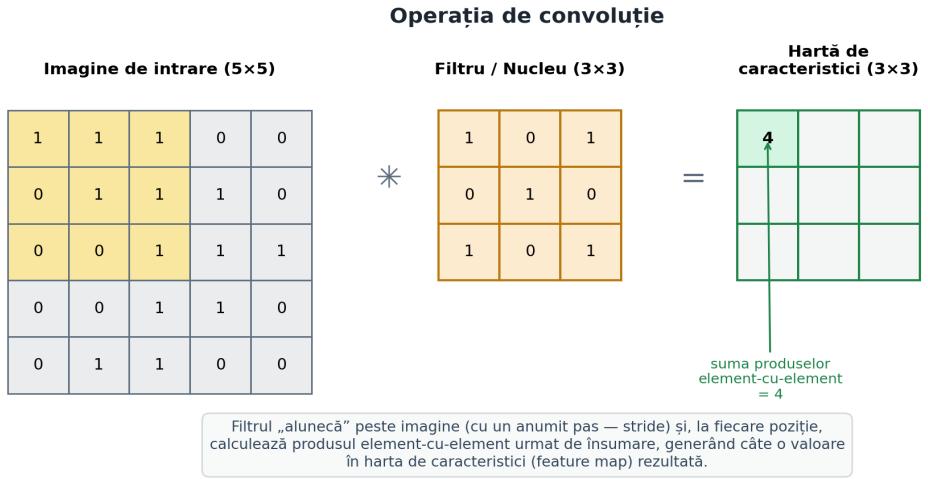


Figura 13.2 Exemplu simplificat al operației de convoluție pe o imagine binară 5×5 cu un filtru 3×3

Formal, valoarea din harta de caracteristici (feature map) obținută la poziția (i, j) se calculează astfel:

$$F(i,j) = \Sigma_m \Sigma_n \; I(i+m, j+n) \cdot K(m,n) \qquad (3)$$

unde: I — imaginea de intrare, K — filtrul (nucleul), F — harta de caracteristici rezultată

Parametrii unui strat convoluțional

- **Dimensiunea filtrului** (*kernel size*) – de regulă 3×3 sau 5×5;
- **Pasul** (*stride*) – cu câți pixeli se deplasează filtrul la fiecare pas (de obicei 1 sau 2);
- **Umplerea** (*padding*) – adăugarea de pixeli suplimentari (de regulă cu valoarea 0) pe marginea imaginii, pentru a controla dimensiunea rezultatului și a păstra informația de la margini;
- **Numărul de filtre** – un strat convoluțional folosește, de regulă, zeci sau sute de filtre diferite în paralel, fiecare „specializat” să detecteze un anumit tip de tipar vizual (margini orizontale, colțuri, culori etc.), rezultând tot atâtea hărți de caracteristici.

Dimensiunea hărții de caracteristici rezultate poate fi calculată cu formula:

$$O = \frac{W - F + 2P}{S} + 1 \quad (4)$$

unde: W — dimensiunea imaginii de intrare, F — dimensiunea filtrului, P — padding, S — stride (pas), O — dimensiunea rezultatului

> 💡 *Exemplu numeric: pentru o imagine 32×32, un filtru 3×3, padding P = 1 și stride S = 1, obținem O = (32 − 3 + 2)/1 + 1 = 32, deci dimensiunea spațială a imaginii se păstrează — acesta este motivul pentru care padding-ul este frecvent folosit.*

Funcția de activare ReLU

După fiecare operație de convoluție se aplică, de regulă, o funcție de activare neliniară. Cea mai utilizată în rețelele CNN moderne este ReLU (Rectified Linear Unit), datorită simplității ei de calcul și faptului că reduce problemele de antrenare întâlnite la alte funcții (precum sigmoid sau tangenta hiperbolică):

$$f(x) = max(0, x) \quad (5)$$

ReLU păstrează neschimbate valorile pozitive și înlocuiește cu 0 toate valorile negative

Cu alte cuvinte, dacă valoarea rezultată dintr-un neuron este pozitivă, aceasta este lăsată neschimbată; dacă este negativă, se înlocuiește cu zero. Această simplitate face ca ReLU să fie foarte rapid de calculat și eficient la antrenarea rețelelor cu multe straturi.

Straturile de Pooling

După convoluție și activare, se aplică frecvent un strat de „pooling” (subeșantionare), al cărui rol este să **reducă dimensiunea** spațială a hărților de caracteristici, păstrând totodată informația cea mai importantă. Cea mai comună variantă este **max pooling**, care păstrează valoarea maximă din fiecare regiune mică (de exemplu 2×2):

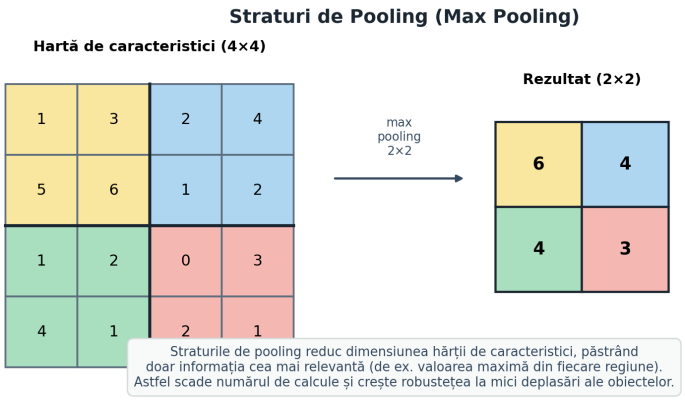


Figura 13.3 Exemplu de max pooling cu fereastră 2×2, care reduce o hartă 4×4 la o hartă 2×2

Utilitatea pooling-ul:

- ✓ Reduce numărul de calcule și de parametri din straturile următoare;
- ✓ Reduce riscul de supra-antrenare (overfitting);
- ✓ Oferă o oarecare invarianță la mici translații sau deformări ale obiectului din imagine (dacă obiectul se deplasează cu 1-2 pixeli, rezultatul pooling-ului rămâne, de regulă, neschimbat).

> 💡 *Există și varianta „average pooling", care calculează media valorilor din regiune în loc de maxim, însă în practică max pooling este preferat pentru sarcinile de clasificare a imaginilor deoarece păstrează cele mai „puternice" activări (semnalele cele mai relevante).*

După mai multe blocuri de convoluție + activare + pooling, rețeaua a extras deja un set de caracteristici relevante despre imagine, sub forma unor hărți de caracteristici de dimensiuni relativ mici, dar cu multe canale. Aceste hărți sunt apoi „aplatizate" (flatten) într-un singur vector, care este introdus într-unul sau mai multe straturi complet conectate (dense) – exact a la rețelele clasice discutate capitolul 12 2.

Stratul de ieșire: funcția Softmax

Pentru probleme de clasificare cu mai multe clase (de exemplu, recunoașterea a 10 tipuri de obiecte), ultimul strat folosește funcția **softmax**, care transformă un set de valori brute (numite „logits") într-o **distribuție de probabilitate**: toate valorile rezultate sunt **pozitive** și însumează 1. **Efectul de „Max" (Accentuează diferențele):** Datorită utilizării funcției exponențiale (e^x), scorurile mai mari sunt amplificate semnificativ, primind o probabilitate mult mai mare, în timp ce scorurile mici (sau negative) sunt „strivite" și se apropie de 0. Nu alege doar valoarea maximă absolută (hard max), ci păstrează informații și despre restul valorilor, de unde și denumirea de *soft*-max.

$$\sigma(z)_i = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}} \tag{6}$$

unde: z — vectorul de valori brute (logits), K — numărul de clase; rezultatul $\sigma(z)_i$ reprezintă probabilitatea estimată a clasei i

Clasa prezisă de rețea este, în cele din urmă, **cea cu probabilitatea cea mai mare**.

Funcția de cost: entropia încrucișată (cross-entropy)

Pentru a măsura cât de „greșită" este predicția rețelei față de eticheta corectă, se folosește funcția de cost numită **entropie încrucișată** (cross-entropy loss):

$$L = -\Sigma_{i=1}^{K} y_i \, log(\hat{y}_i) \tag{7}$$

unde: y_i — eticheta reală (0 sau 1 pentru fiecare clasă), $\hat{y}_i$ — probabilitatea prezisă de rețea pentru clasa i

Cu cât rețeaua atribuie o probabilitate mai mare clasei corecte, cu atât valoarea funcției de cost L este mai mică. **Scopul antrenării** este tocmai **minimizarea** acestei valori pentru toate exemplele din setul de date.

Procesul de antrenare a unei rețele CNN urmează aceleași principii generale discutate deja pentru rețelele neuronale clasice, la tema desspre învățarea supervizată — se repetă un ciclu de patru pași, până când funcția de cost ajunge la o valoare suficient de mică:

- 1. Propagare înainte (forward pass) – imaginea de intrare trece prin toate straturile rețelei (convoluție, activare, pooling, dense, softmax), obținându-se o predicție;
- 2. Calculul erorii – se compară predicția rețelei cu eticheta reală, folosind funcția de cost (cross-entropy);

- 3. Retropropagare (backpropagation) – eroarea este propagată înapoi prin rețea, calculându-se contribuția fiecărei ponderi la eroarea totală (gradientul);
- 4. Actualizarea ponderilor – ponderile sunt ajustate în direcția care reduce eroarea, folosind un algoritm de optimizare (de exemplu, gradient descent sau varianta sa Adam):

$$w = w - \eta \frac{\partial L}{\partial w} \quad (8)$$

unde: w — o pondere oarecare din rețea, η (eta) — rata de învățare (learning rate), ∂L/∂w — gradientul funcției de cost în raport cu ponderea w

Particularitatea rețelelor CNN este că, datorită partajării ponderilor (același filtru este folosit în toate pozițiile imaginii), gradientul pentru fiecare filtru se calculează prin însumarea contribuțiilor din toate pozițiile în care acel filtru a fost aplicat. Acest lucru este gestionat automat de bibliotecile moderne de Deep Learning (de exemplu, TensorFlow sau PyTorch), care realizează diferențierea automată.

Câteva noțiuni practice

- Epocă (epoch) – o parcurgere completă a întregului set de date de antrenare;
- Lot (batch) – un subset de exemple procesate simultan înainte de a actualiza ponderile (de exemplu, 32 sau 64 de imagini);
- Rata de învățare (learning rate) – un parametru esențial: o valoare prea mare poate face antrenarea instabilă, o valoare prea mică o face foarte lentă;
- Supra-antrenarea (overfitting) – atunci când rețeaua „memorează” datele de antrenare în loc să învețe tipare generale, performanța fiind bună pe datele de antrenare, dar slabă pe date noi.

💡 *Antrenarea rețelelor CNN de la zero necesită, de regulă, seturi mari de date etichetate și resurse de calcul semnificative (GPU, ore sau zile de antrenare). În paragraful despre Transfer Learning vom vedea cum putem evita, în mare parte, acest cost.*

13.3. Transfer learning

De-a lungul timpului, cercetătorii au propus numeroase arhitecturi CNN, fiecare aducând îmbunătățiri notabile. Cunoașterea acestor „repere istorice” ajută la înțelegerea evoluției domeniului:

Tabelul 13.2. Arhitecturi CNN

Arhitectură	An	Straturi	Contribuție principală
LeNet-5	1998	7	Prima rețea CNN aplicată cu succes (recunoașterea cifrelor scrise de mână)
AlexNet	2012	8	A câștigat concursul ImageNet; a popularizat ReLU și utilizarea GPU
VGG-16	2014	16	Arhitectură uniformă, filtre mici (3×3), foarte adâncă pentru acea vreme
GoogLeNet (Inception)	2014	22	Module „Inception” care combină filtre de dimensiuni diferite
ResNet-50	2015	50	Conexiuni reziduale ("skip connections"), permite rețele foarte adânci
EfficientNet	2019	variabil	Scalare echilibrată a adâncimii, lățimii și rezoluției rețelei

Tendința generală de-a lungul timpului a fost creșterea adâncimii rețelelor (numărul de straturi), însoțită de inovații care au făcut posibilă antrenarea unor astfel de rețele foarte adânci fără ca performanța să se degradeze (de exemplu, conexiunile reziduale introduse de ResNet).

> 💡 *Nu este necesar să memorați toate detaliile acestor arhitecturi — este important să rețineți ideea de evoluție și faptul că multe dintre ele sunt disponibile gata antrenate, fiind punctul de plecare ideal pentru Transfer Learning.*

Antrenarea unei rețele CNN de la zero necesită, de regulă, milioane de imagini etichetate și resurse de calcul considerabile – condiții greu de îndeplinit pentru majoritatea proiectelor reale.

Transfer Learning este o tehnică ce permite reutilizarea cunoștințelor acumulate de o rețea deja antrenată pe un set de date mare (de exemplu, ImageNet, care conține peste un milion de imagini din 1000 de categorii), pentru a rezolva o sarcină nouă, adesea cu un set de date mult mai mic.

Ideea centrală se bazează pe observația din secțiunea anterioară privind ierarhia caracteristicilor învățate de un CNN: primele straturi învață caracteristici foarte generale (margini, culori, texturi), utile pentru aproape orice sarcină de procesare a imaginilor, în timp ce ultimele straturi învață caracteristici specifice sarcinii pentru care a fost antrenată rețeaua inițial.

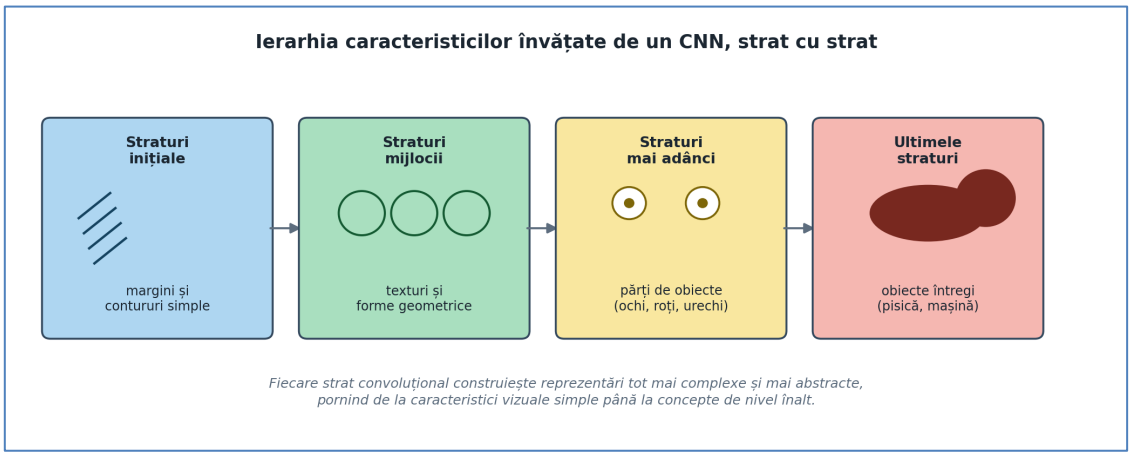


Figura 13.4 Ierarhia caracteristicilor învățate de un CNN, de la simple la complexe

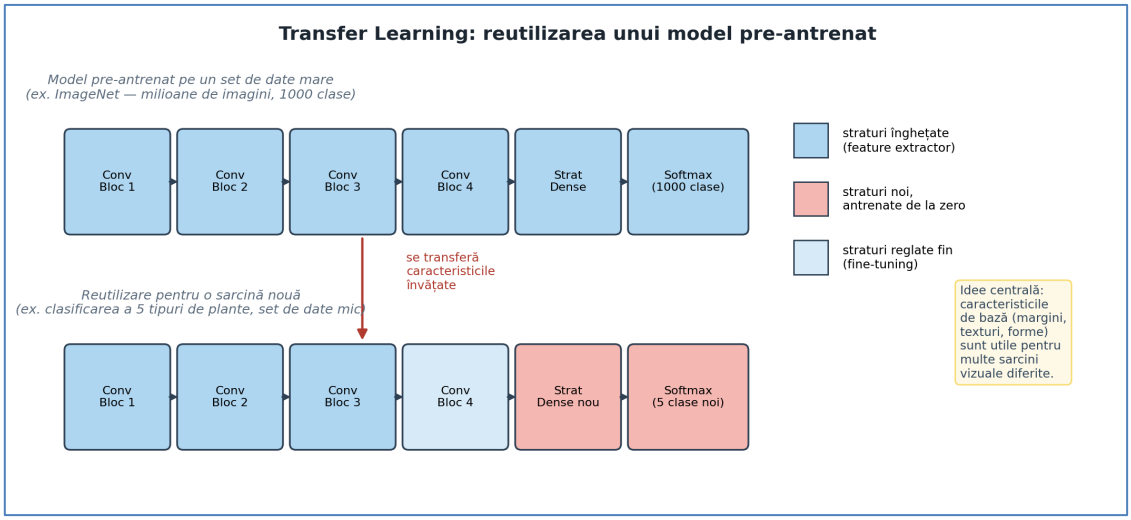


Figura 13.5 Principiul Transfer Learning: straturile inițiale (albastru) sunt reutilizate, iar straturile finale (roșu) sunt înlocuite și antrenate pentru noua sarcină

Cele două strategii principale ale Transfer Learning

- **Extragere de caracteristici** (Feature Extraction) – straturile convoluționale ale rețelei pre-antrenate sunt „înghețate” (ponderile lor nu se modifică), și doar noul strat de clasificare (dense + softmax), adăugat la final, este antrenat pentru noua sarcină;
- **Reglare fină** (Fine-tuning) – pe lângă noul strat de clasificare, se antrenează (cu o rată de învățare mică) și o parte dintre straturile convoluționale finale ale rețelei pre-antrenate, pentru a le adapta mai bine la particularitățile noii sarcini.

Avantajele Transfer Learning

- Necesită mult mai puține date etichetate pentru noua sarcină (uneori doar câteva sute de exemple);
- Timpul de antrenare este semnificativ redus;
- Performanțele obținute sunt, de multe ori, superioare celor ale unei rețele antrenate de la zero pe un set de date mic, deoarece rețeaua pre-antrenată a „văzut” deja o varietate uriașă de tipare vizuale.

> 💡 *Exemplu concret: pentru a construi un clasificator care distinge 5 varietăți de plante de apartament, folosind doar câteva sute de fotografii, este mult mai eficient să pornim de la o rețea precum ResNet-50, pre-antrenată pe ImageNet, decât să antrenăm o rețea de la zero.*

Rețelele convoluționale stau la baza majorității sistemelor moderne de procesare a imaginilor și video. Cele mai importante categorii de aplicații sunt:

Tabelul 13.3

Aplicație	Descriere pe scurt	Exemple de utilizare
Clasificare de imagini	Atribuirea unei singure etichete întregii imagini	Sortarea automată a fotografiilor, diagnostic medical preliminar
Detecția obiectelor	Localizarea și identificarea mai multor obiecte într-o imagine (cu "casete" de încadrare)	Vehicule autonome, sisteme de securitate
Segmentarea semantică	Clasificarea fiecărui pixel din imagine în funcție de obiectul căruia îi aparține	Analiza imaginilor medicale, hărți din imagini satelitare
Recunoașterea facială	Identificarea sau verificarea identității unei persoane	Deblocarea telefoanelor, control acces
Generarea și restaurarea imaginilor	Îmbunătățirea calității sau crearea de imagini noi	Eliminarea zgomotului, super-rezoluție, restaurare foto

În domeniul medical, de exemplu, rețelele CNN sunt folosite pentru a detecta anomalii în radiografii, tomografii sau imagini de retină, oferind un instrument suplimentar de sprijin pentru medici – nu un înlocuitor al acestora, ci un asistent capabil să semnaleze rapid regiuni suspecte.

Pentru evaluarea acestor aplicații, se utilizează o gamă de metrici: Accuracy (precizie generală), Precision și Recall (utile în diagnostic medical unde falsurile negative sunt critice), și mAP (mean Average Precision) pentru detecția obiectelor.

Augmentarea datelor (Data Augmentation)

Deoarece performanța rețelelor CNN depinde puternic de volumul de date de antrenare disponibile, se folosește frecvent augmentarea datelor: generarea artificială de noi exemple de antrenare prin transformări simple ale imaginilor existente – rotații, oglindiri, decupări, modificări de luminozitate sau contrast. Astfel, rețeaua „vede” o varietate mai mare de situații și devine mai robustă.

Supra-antrenarea (Overfitting) și regularizarea

Overfitting-ul apare atunci când o rețea devine atât de complexă încât „memorează” particularitățile exacte ale datelor de antrenare, în loc să învețe tipare generale – rezultând performanțe foarte bune pe datele de antrenare, dar slabe pe date noi, nevăzute anterior. Printre cele mai utilizate tehnici de regularizare, care combat overfitting-ul, se numără:

- Dropout – în timpul antrenării, un procent aleatoriu de neuroni (de exemplu 20-50%) este „dezactivat” temporar la fiecare pas, forțând rețeaua să nu depindă excesiv de anumiți neuroni specifici;
- Normalizarea pe loturi (Batch Normalization) – normalizează valorile din interiorul rețelei, stabilizând și accelerând antrenarea;
- Oprirea timpurie (Early Stopping) – antrenarea se oprește atunci când performanța pe un set de date de validare începe să se degradeze, chiar dacă performanța pe setul de antrenare continuă să crească.

Alte arhitecturi de Deep Learning – o privire de ansamblu

Pe lângă CNN, specializate în date cu structură spațială (imagini), există și alte familii importante de rețele de învățare profundă, care vor fi discutate detaliat în capitolul viitor:

- Rețelele Recurente (RNN, LSTM, GRU) – potrivite pentru date secvențiale (text, serii de timp, semnale audio);
- Arhitectura Transformer – la baza modelelor moderne de limbaj (precum cele din familia GPT), dar folosită tot mai mult și în procesarea imaginilor (Vision Transformer);
- Rețelele Generative Adversariale (GAN) și modelele de difuzie – utilizate pentru generarea de imagini noi, realiste.

Întrebări de autoevaluare

1. Care sunt principalele diferențe dintre un strat convoluțional și un strat complet conectat (dense)?
2. De ce este utilă operația de pooling într-o rețea CNN?
3. Explicați, cu propriile cuvinte, diferența dintre "feature extraction" și "fine-tuning" în contextul Transfer Learning.
4. Care este diferența esențială dintre Machine Learning-ul clasic și Deep Learning în ceea ce privește procesul de extragere a caracteristicilor (*feature engineering*)
5. Care sunt cauzele principale care au determinat adoptarea în masă a învățării profunde în ultimul deceniu?
6. Care sunt cele două mari probleme care apar atunci când încercăm să aplicăm o rețea complet conectată (*dense/fully connected*) direct pe o imagine brută?
7. **Principalele proprietăți CNN:** Cum reușesc rețelele neuronale convoluționale (CNN) să depășească limitările rețelelor dense prin conectivitate locală, partajarea ponderilor (*weight sharing*) și invarianță la translație?
8. **Parametrii stratului convoluțional:** Ce rol au pasul (*stride*) și umplerea (*padding*) în calculul dimensiunii hărții de caracteristici rezultate în urma operatiei de convoluție?
9. Cum este definită funcția de activare ReLU și din ce motiv este preferată în arhitecturile CNN moderne față de alte funcții precum sigmoid sau tangenta hiperbolică?
10. Care este rolul straturilor de *max pooling* într-o rețea convoluțională și ce avantaje oferă acestea procesului de învățare?
11. Ce rol au funcția Softmax și funcția de cost de entropie încrucișată (*cross-entropy loss*) la stratul de ieșire al unei rețele folosite pentru clasificare?
12. Ce presupune tehnica de *Transfer Learning* și care este diferența dintre cele două strategii ale sale: extragerea de caracteristici (*feature extraction*) și reglarea fină (*fine-tuning*)?
13. **Prevenirea supra-antrenării:** Ce reprezintă fenomenul de supra-antrenare (*overfitting*) și prin ce mecanisme el poate fi prevenit?

CAPITOLUL 14. PROCESAREA LIMBAJULUI NATURAL (NLP)

14.1. Definiție și domenii de aplicare

Procesarea Limbajului Natural (*Natural Language Processing*, NLP) este ramura inteligenței artificiale care se ocupă cu interacțiunea dintre calculatoare și limbajul uman — atât cel scris, cât și cel vorbit. Obiectivul NLP este de a permite mașinilor să înțeleagă, să interpreteze, să genereze și să răspundă la limbajul natural într-un mod util și corect din punct de vedere **semantic**.

NLP combină lingvistica computațională (reguli, gramatici, morfologie) cu învățarea automată statistică și, în ultimul deceniu, cu rețelele neuronale profunde. Domeniul stă la baza unei game largi de aplicații economice și sociale utilizate zilnic:

- traducerea automată (Google Translate, DeepL);
- motoarele de căutare și sistemele de regăsire a informației;
- chatbot-uri și asistenți virtuali (Siri, Alexa, ChatGPT, Claude);
- analiza sentimentelor în recenzii de produse și rețele sociale;
- sisteme de rezumare automată a documentelor economice și juridice;
- corectoare gramaticale și instrumente de generare de text (redactare asistată).

Spre deosebire de limbajele de programare, limbajul natural este ambiguu, contextual și în continuă evoluție. Un sistem NLP trebuie să facă față mai multor tipuri de dificultăți fundamentale:

- **Ambiguitate lexicală**: „Bancă” poate însemna instituție financiară sau locul unde stai jos
- **Ambiguitate sintactică**: „Am văzut omul cu telescopul” – cine are telescopul?
- **Anaforă / referință:** „Robotul a ridicat cutia pentru că era grea” – ce era greu?
- **Variabilitate morfologică**: În limba română: „citesc, citești, citește, citim...”
- **Context și pragmatică:** Sarcasmul, ironia, expresiile idiomatice

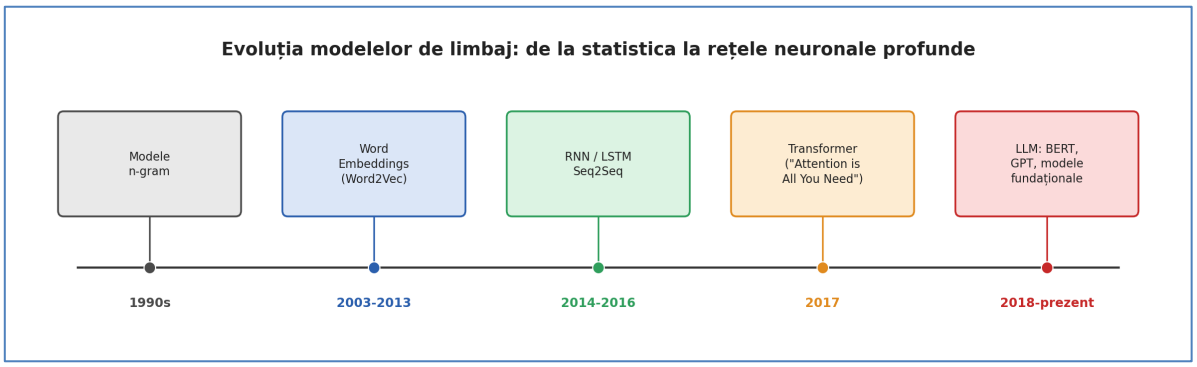


Figura 14.1 Evoluția istorică a modelelor de limbaj – de la n-gram la LLM

Idee-cheie

Întreaga evoluție a NLP poate fi privită ca o căutare progresivă a unor reprezentări tot mai bune ale contextului: de la ferestre fixe de câteva cuvinte (n-gram), la vectori denși ce captează sensul (embeddings), la mecanisme capabile să pondereze dinamic tot contextul unei propoziții (atenție, Transformer).

Înainte de a calcula probabilități, textul trebuie transformat într-o formă numerică. Procesul se numește **tokenizare**. Un **token** poate fi un cuvânt, o silabă sau chiar un caracter individual. Modelele moderne utilizează metode de tokenizare subword (precum Byte Pair Encoding - BPE sau WordPiece), care permit gestionarea eficientă a cuvintelor rare și a morfologiei complexe (ex: „neîntâlnit” → „ne” + „întâlnit”). Această etapă este crucială deoarece determină vocabularul modelului și eficiența sa în procesarea limbii române.

14.2. Modele de limbaj n-gram

Un **model de limbaj** (language model) este un sistem care atribuie o **probabilitate** unei secvențe de cuvinte, estimând cât de „naturală” sau plauzibilă este acea secvență într-o limbă dată.

Formal, pentru o secvență de cuvinte w_1, w_2, ..., w_m, dorim să calculăm probabilitatea comună $P(w_1, w_2, ..., w_m)$. Aplicând **regula produsului** (chain rule) din teoria probabilităților, aceasta se descompune astfel:

$$P(w_1, w_2, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \quad (1)$$

Problema practică este că, pentru propoziții lungi, numărul de contexte posibile w_1, ..., w_{i-1} crește exponențial, iar majoritatea acestor contexte nu apar niciodată într-un corpus de antrenare — oricât de mare ar fi acesta. Aici intervine **ipoteza Markov**.

Ipoteza Markov și aproximarea n-gram

Modelele n-gram simplifică problema presupunând că ***probabilitatea unui cuvânt depinde doar de ultimele n-1 cuvinte precedente, nu de întregul istoric al propoziției***:

$$P(w_i \mid w_1, \ldots, w_{i-1}) \approx P(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) \quad (2)$$

În funcție de valoarea lui n, distingem: unigram (n=1, fiecare cuvânt este independent), bigram (n=2, se ia în calcul un singur cuvânt anterior), trigram (n=3) și, mai general, n-gram.

Estimarea probabilităților din corpus (MLE)

Probabilitățile condiționate ale unui model n-gram se estimează prin numărare directă de frecvențe într-un corpus mare de text — metodă cunoscută drept **estimarea de maximă verosimilitate** (Maximum Likelihood Estimation, MLE). Pentru un model bigram:

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})} \quad (3)$$

Exemplu numeric: dacă în corpus cuvântul „inteligentă” apare de 50 de ori, iar secvența „inteligentă artificială” apare de 42 de ori, atunci P(„artificială” | „inteligentă”) = 42/50 = 0,84 – o probabilitate ridicată, coerentă cu uzul frecvent al colocației în limbă.

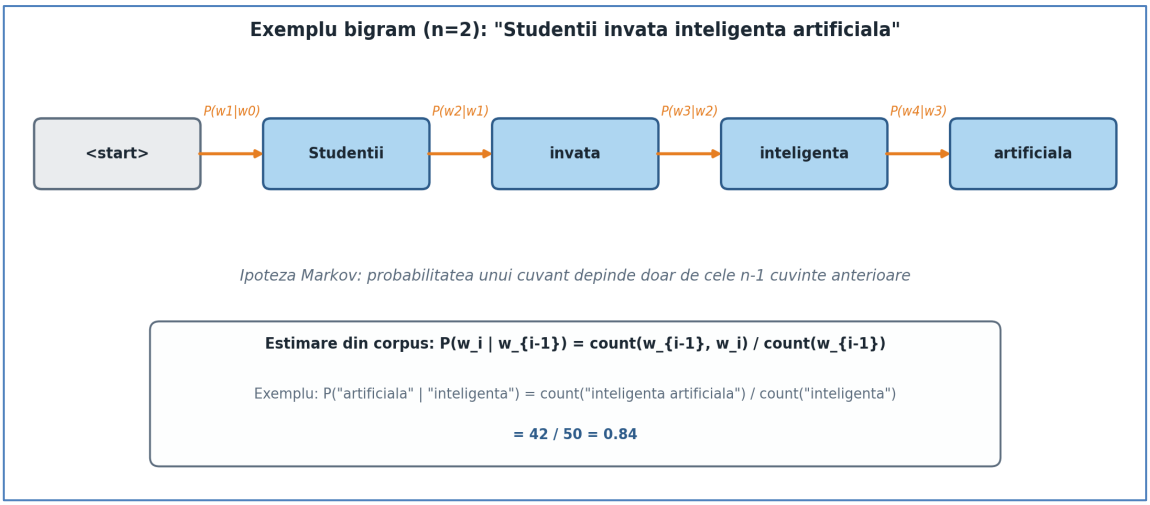


Figura 14.2 Model bigram: lanțul de probabilități condiționate între cuvinte consecutive

Netezirea (smoothing) și evaluarea: perplexitatea

O problemă majoră este cea a secvențelor cu frecvență zero: dacă o combinație de cuvinte nu apare deloc în corpusul de antrenare, modelul îi atribuie probabilitate nulă, ceea ce blochează întregul calcul al produsului. Tehnicile de netezire (smoothing), precum Laplace (add-one), Kneser-Ney sau Good-Turing, redistribuie o mică parte din masa de probabilitate către evenimentele nevăzute.

Calitatea unui model de limbaj se evaluează frecvent prin **perplexitate** (perplexity) – o măsură inversă a probabilității medii atribuite unui text de test; cu cât perplexitatea este mai mică, cu atât modelul „este mai puțin surprins" de textul respectiv, deci este mai bun:

$$PP(W) = P(w_1, \ldots, w_N)^{-1/N} \quad (4)$$

Modelele n-gram au stat la baza primelor sisteme de recunoaștere vocală, corectare ortografică și traducere automată statistică. Deși utile, modelele n-gram au **limitări fundamentale** care au motivat dezvoltarea unor abordări bazate pe rețele neuronale:

- **Fereastră de context fixă:** Nu pot capta dependențe pe distanțe lungi (peste n cuvinte).
- **Explozia combinatorică:** Numărul de n-grame posibile crește exponențial cu n.
- **Raritatea datelor (sparsity):** Majoritatea combinațiilor de cuvinte nu apar niciodată în corpus.
- **Lipsa generalizării semantice:** „Pisică" și „motan" sunt tratate ca simboluri total distincte.
- **Absența noțiunii de sens**: Nu există o reprezentare a similarității semantice între cuvinte

Ultima limitare este esențială: într-un model n-gram, cuvintele sunt reprezentate discret (one-hot encoding), adică fiecare cuvânt este un vector de dimensiunea vocabularului, cu o singură valoare de 1 și restul de 0. Doi vectori one-hot au întotdeauna aceeași distanță unul față de celălalt, indiferent de sensul cuvintelor — modelul nu „știe" că „pisică" și „motan" sunt înrudite semantic. Această observație a condus la ideea reprezentărilor vectoriale dense.

14.3. Reprezentări vectoriale ale cuvintelor (Word Embeddings)

Reprezentările vectoriale de cuvinte (word embeddings) se bazează pe **ipoteza distribuțională**, formulată de lingvistul J.R. Firth: „*un cuvânt este caracterizat de compania pe care o ține*" – cuvinte care apar în contexte similare tind să aibă sensuri similare. Pornind de la această idee, fiecare cuvânt este reprezentat printr-un vector dens de numere reale (tipic 100-300 dimensiuni), învățat automat dintr-un corpus mare de text, astfel încât cuvinte cu sensuri apropiate să aibă vectori apropiați în spațiul vectorial.

Modele reprezentative: Word2Vec și GloVe

- Word2Vec (Mikolov et al., 2013) – antrenează o rețea neuronală superficială să prezică fie cuvântul central din context (arhitectura CBOW), fie contextul dintr-un cuvânt central (arhitectura Skip-gram);
- GloVe (Global Vectors, Pennington et al., 2014) – combină statisticile globale de co-apariție a cuvintelor la nivelul întregului corpus cu metode de factorizare matriceală;
- FastText (Facebook AI, 2016) – extinde Word2Vec la nivel de subcuvinte (n-grame de caractere), rezolvând parțial problema cuvintelor rare sau neîntâlnite.

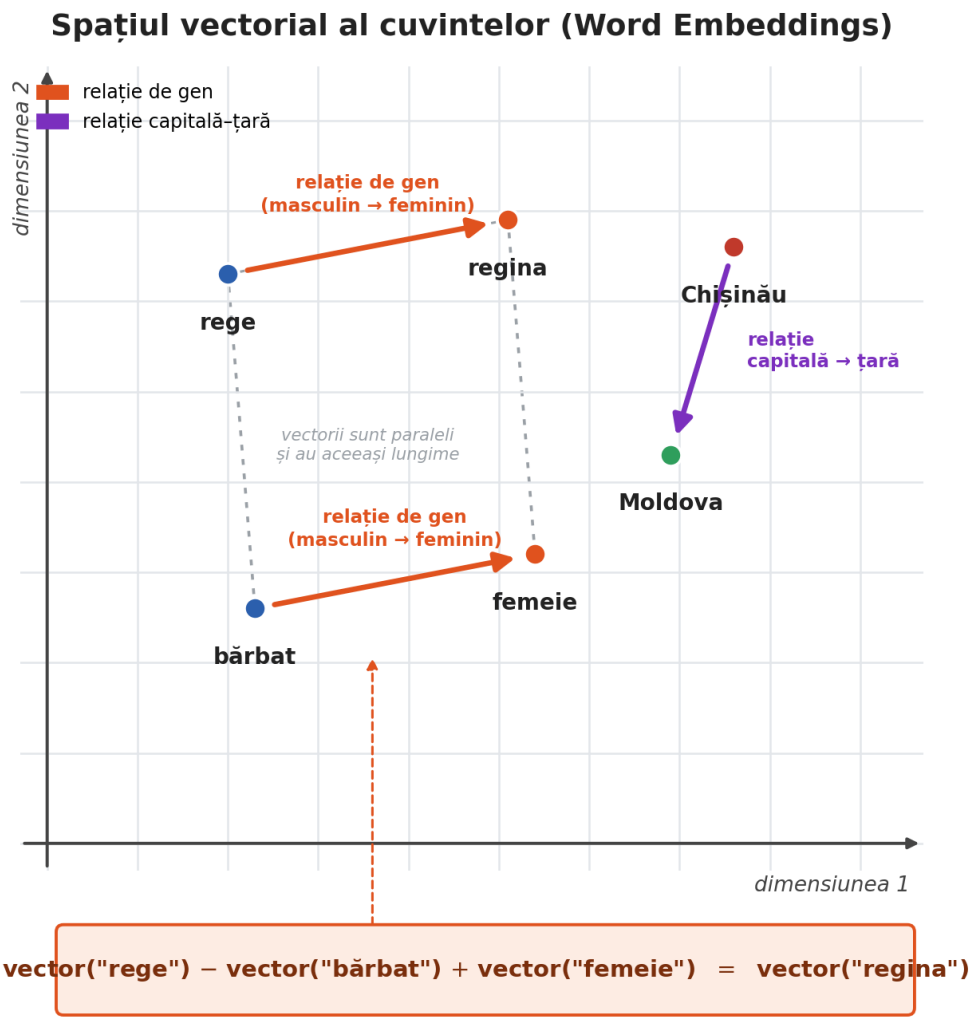


Figura 14.3 Spațiul vectorial al cuvintelor – relațiile semantice devin relații geometrice

Proprietatea cea mai remarcabilă a acestor reprezentări este capacitatea de a capta analogii semantice și sintactice prin simple operații aritmetice vectoriale, exemplul clasic fiind:

Analogie vectorială clasică

vector("rege") − vector("bărbat") + vector("femeie") ≈ vector("regină"). Această regularitate geometrică demonstrează că spațiul vectorial învățat codifică implicit relații semantice precum genul, capitala unei țări sau timpul verbal.

Totuși, embeddings-urile de tip Word2Vec/GloVe sunt **statice**: fiecărui cuvânt îi corespunde un singur vector, indiferent de context. Cuvântul „bancă” primește același vector și în „am depus banii la bancă”, și în „m-am așezat pe o bancă” – o limitare rezolvată abia de modelele contextuale bazate pe Transformer.

„Totuși, o limitare majoră a Word2Vec este natura lor statică (un cuvânt are un singur vector indiferent de context). Această problemă a fost rezolvată de modelele de reprezentări contextuale (ex. ELMo, BERT), care utilizează rețele neuronale pentru a genera vectori dinamici în funcție de vecindajul cuvintelor, pregătind terenul pentru arhitecturile bazate pe atenție.”

14.4. Rețele neuronale recurente (RNN / LSTM)

Pentru a modela dependențele dintre cuvinte pe parcursul întregii propoziții, au fost introduse **rețelele neuronale recurente** (Recurrent Neural Networks, RNN). Ideea centrală este procesarea secvențială a cuvintelor, cuvânt cu cuvânt, menținând o stare ascunsă (hidden state) care acumulează informație despre tot ce a fost citit până în acel moment.

Rețelele RNN clasice suferă însă de problema gradientului evanescent (vanishing gradient): pe măsură ce secvența se lungește, informația din primele cuvinte se „diluează" exponențial, iar rețeaua nu mai poate învăța dependențe pe distanțe mari. Arhitectura LSTM (Long Short-Term Memory, Hochreiter și Schmidhuber, 1997) rezolvă parțial această problemă printr-un sistem de „porți" (gates) care controlează explicit ce informație este păstrată, actualizată sau uitată din starea internă.

- Poarta de uitare (forget gate) – decide ce informație din memorie se elimină;
- Poarta de intrare (input gate) – decide ce informație nouă se adaugă în memorie;
- Poarta de ieșire (output gate) – decide ce parte din memorie este utilizată la momentul curent.

Arhitecturile Seq2Seq (secvență-la-secvență), construite din doi RNN/LSTM — un encoder care comprimă propoziția sursă într-un singur vector de context, și un decoder care generează propoziția țintă din acel vector – au reprezentat standardul pentru traducerea automată neuronală în perioada 2014-2017. Limitarea lor esențială rămânea însă „gâtuirea informațională" (information bottleneck): întreaga propoziție sursă, indiferent de lungime, trebuia comprimată într-un singur vector de dimensiune fixă.

Tranziție

Această limitare a motivat introducerea mecanismului de **atenție** (Bahdanau et al., 2014), care permite decoderului să „privească" selectiv, la fiecare pas, către diferite părți ale propoziției sursă — idee care va deveni, câțiva ani mai târziu, fundația arhitecturii Transformer.

14.5. Mecanismul de atenție (Attention)

Ideea centrală a atenției este simplă: în loc să comprime tot contextul într-un singur vector fix, modelul învață să calculeze, pentru fiecare cuvânt generat, un set de ponderi care indică cât de relevant este fiecare cuvânt din context pentru decizia curentă. Cu cât ponderea este mai mare, cu atât acel cuvânt „contează" mai mult în calculul reprezentării curente.

Auto-atenția (self-attention) este forma de atenție folosită de arhitectura Transformer, în care fiecare cuvânt dintr-o propoziție calculează ponderi de relevanță față de toate celelalte cuvinte din aceeași propoziție — inclusiv față de el însuși. Astfel, modelul poate rezolva probleme de anaforă (la ce se referă un pronume sau adjectiv) direct prin ponderare contextuală, fără reguli lingvistice explicite.

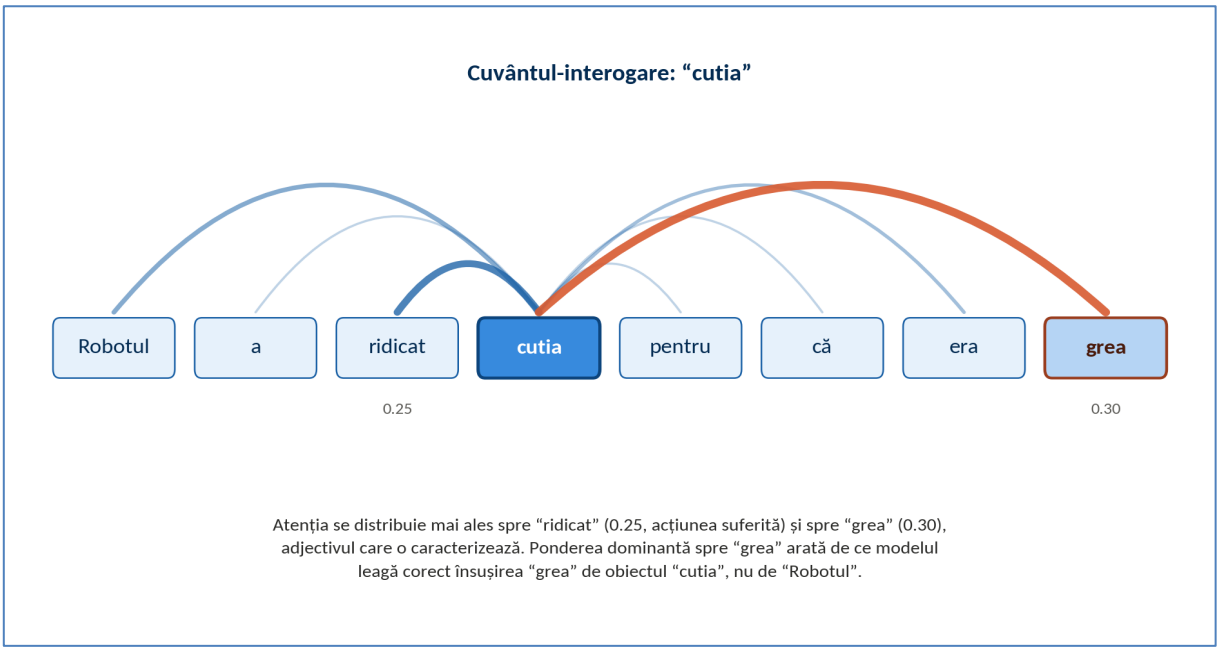


Figura 14.4 Auto-atenția rezolvă ambiguitatea referențială: „grea” se leagă puternic de „cutia”

Formalizarea matematică: Query, Key, Value

Mecanismul de auto-atenție „scaled dot-product” utilizează trei **proiecții liniare** ale fiecărui vector de intrare: Query (Q – „ce caut”), Key (K – „ce ofer ca etichetă de căutare”) și Value (V – „ce conținut ofer efectiv”). Relevanța dintre două cuvinte se calculează ca **produsul scalar** dintre Query-ul unuia și Key-ul celuilalt:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) * V \quad (5)$$

Produsul QK^T produce scorurile brute de similaritate între toate perechile de cuvinte; împărțirea la rădăcina dimensiunii vectorilor Key ($\sqrt{d_k}$) stabilizează gradientul în timpul antrenării; funcția softmax normalizează scorurile în ponderi de probabilitate ce însumează 1; iar rezultatul final este o combinație ponderată a vectorilor Value. Funcția softmax utilizată pentru normalizare este definită astfel – transformă orice vector de scoruri reale într-o distribuție de probabilitate:

$$softmax(z_i) = \frac{e^{z_i}}{\Sigma_j^N e^{z_j}} \quad (6)$$

Atenția multi-head (Multi-Head Attention)

În practică, Transformer nu calculează o singură funcție de atenție, ci mai multe „capete” (heads) în paralel, fiecare cu propriile proiecții Q, K, V, învățate independent. Fiecare cap poate specializa în a capta un tip diferit de relație lingvistică – unul poate urmări acordul gramatical subiect-predicat, altul poate urmări relații de coreferință, altul relații semantice de tip cauză-efect. Rezultatele tuturor capetelor sunt concatenate și proiectate din nou liniar, obținând o reprezentare finală mult mai bogată decât ar permite un singur mecanism de atenție.

14.6. Calcularea Auto-Atenției (exemplu practic)

1. Datele Inițiale (Exemplu cu 2 Cuvinte)

Pentru a ilustra, vom simula o secvență de 2 cuvinte. Să presupunem că avem deja matricile Q, K și V generate de rețea. Dimensiunea vectorilor Key (d_k) va fi **3**.

Matricea Query (Q) [2 cuvinte x dimensiunea 3]:
Row 1 (Cuvânt 1): **[1, 0, 2]**
Row 2 (Cuvânt 2): **[0, 1, 1]**
Matricea Key (K) [2 cuvinte x dimensiunea 3]:
Row 1 (Cuvânt 1): **[0, 1, 1]**
Row 2 (Cuvânt 2): **[1, 0, 0]**
Matricea Value (V) [2 cuvinte x dimensiunea 2]:
Row 1 (Cuvânt 1): **[1, 2]**
Row 2 (Cuvânt 2): **[3, 4]**

2. Pașii de Calcul

Pasul 1: Produsul Scalar (Q * K^T)

Se înmulțește matricea Q cu **transpusa** matricei K pentru a afla „scorurile brute" de atenție.
K^T (Transpusa) este:
[0, 1]
[1, 0]
[1, 0]
Q * K^T =
[(1*0 + 0*1 + 2*1), (1*1 + 0*0 + 2*0)]
[(0*0 + 1*1 + 1*1), (0*1 + 1*0 + 1*0)]
Scoruri Brute =
[2, 1]
[2, 0]

Pasul 2: Scalarea

Împărțim scorurile la √d_k. Cum d_k = 3, √3 ≈ 1.732.
Scoruri Scalate ≈
[2 / 1.732, 1 / 1.732]
[2 / 1.732, 0 / 1.732]
Rezultat Scalat =
[1.15, 0.58]
[1.15, 0.00]

Pasul 3: Aplicarea funcției Softmax

Se aplică funcția softmax pe fiecare rând separat pentru a obține ponderi de probabilitate care însumează 1.
Rândul 1: softmax([1.15, 0.58]) ≈ [0.64, 0.36]
Rândul 2: softmax([1.15, 0.00]) ≈ [0.76, 0.24]
Matricea Ponderilor (Atenția) =
[0.64, 0.36]
[0.76, 0.24]

Pasul 4: Înmulțirea cu matricea Value (V)

Combinăm aceste ponderi cu informația efectivă din V.
Atenție * V =
[0.64, 0.36] * **[1, 2]**
[0.76, 0.24] **[3, 4]**
Rândul 1 = [(0.64*1 + 0.36*3), (0.64*2 + 0.36*4)] = [1.72, 2.72]
Rândul 2 = [(0.76*1 + 0.24*3), (0.76*2 + 0.24*4)] = [1.48, 2.48]

Rezultatul final

Noua reprezentare a celor două cuvinte, după mecanismul de auto-atenție, este:

Cuvântul 1: [1.72, 2.72]

Cuvântul 2: [1.48, 2.48]

Acestea sunt noile „contextualized embeddings” care înglobează contextul reciproc din propoziție.

14.7. Arhitectura Transformer

Arhitectura Transformer, introdusă în articolul „Attention Is All You Need” (Vaswani et al., 2017), a renunțat complet la recurență, bazându-se exclusiv pe mecanisme de atenție și rețele feed-forward. Această schimbare a permis paralelizarea masivă a antrenării (toate cuvintele unei propoziții pot fi procesate simultan, nu secvențial ca la RNN), reducând drastic timpul de antrenare pe seturi de date uriașe.

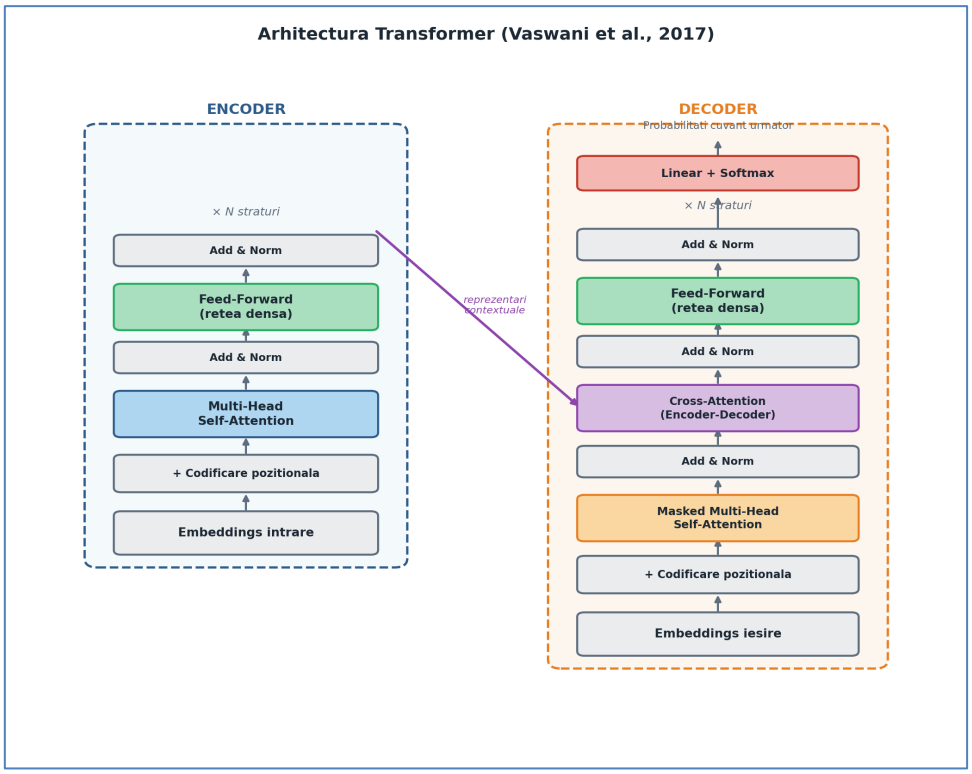


Figura 14.5 Arhitectura Transformer: encoder (stânga) și decoder (dreapta)

Codificarea pozițională (Positional Encoding)

Deoarece Transformer procesează toate cuvintele simultan (nu secvențial), modelul nu are, prin construcție, nicio informație despre ordinea cuvintelor. Pentru a rezolva această problemă, se adaugă la fiecare vector de embedding o codificare pozițională – un vector calculat cu funcții sinusoidale de frecvențe diferite, unic pentru fiecare poziție din secvență:

$$PE(pos, 2i) = sin\left(\frac{pos}{10000^{2i\,/\,d_{model}}}\right) \quad (7)$$

unde pos este poziția cuvântului în secvență ($pos \in \{0,1,2,\dots,N-1\}$); indicele dimensiunii ($2i$) indică componentele pare ale vectorului de embedding (unde i variază de la 0 la $d_{\text{model}}/2$)., iar d_{model} este dimensiunea totală a vectorului de embedding Scala de frecvență ($10000^{2i/d}$): generează o progresie geometrică a lungimilor de undă (de la 1 la 10000), permițând modelului să învețe ușor relațiile dintre poziții prin rotații sinusoidale.

Pentru dimensiunile pare se folosește funcția sinus (formula (7)), iar pentru cele impare, funcția cosinus (formulă simetrică (8)).

$$PE(pos, 2i+1) = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \quad (8)$$

Componentele encoder-ului

Fiecare **strat** al encoder-ului conține **două sub-straturi principale**, fiecare urmat de o conexiune reziduală (Add) și normalizare (Norm) Aceste componente sunt esențiale pentru stabilitatea antrenării modelelor adânci, deoarece permit gradientelor să „curgă" prin straturile rețelei fără a se atenua (problema vanishing gradient) și mențin distribuția activării într-un interval optim.":

1. **Multi-Head Self-Attention** – fiecare cuvânt „privește" către toate celelalte cuvinte din propoziția de intrare;
2. **Rețea Feed-Forward** – o rețea densă aplicată independent fiecărei poziții, care introduce non-liniaritate și capacitate suplimentară de reprezentare.

Aceste două sub-straturi se repetă de N ori (tipic N=6 în arhitectura originală, dar sute ori în modelele LLM moderne), rezultatul fiecărui strat devenind intrarea stratului următor.

Componentele decoder-ului

Decoder-ul are o structură similară, cu două diferențe esențiale:

- **Masked Self-Attention** – la generarea unui cuvânt, decoder-ul poate „privi" doar către cuvintele deja generate anterior, nu către cele viitoare (esențial pentru generarea autoregresivă de text);
- **Cross-Attention** (atenție încrucișată) – un strat suplimentar prin care decoder-ul „privește" către reprezentările produse de encoder, realizând legătura dintre limba sursă și limba țintă (sau, mai general, între **context și continuare**).

Trei familii arhitecturale derivate din Transformer

Encoder-only (ex. BERT) — optimă pentru înțelegere: clasificare, analiza sentimentelor. Decoder-only (ex. GPT, Claude, Llama) — optimă pentru generare autoregresivă de text. Encoder-Decoder (ex. T5, BART) — optimă pentru sarcini de transformare text-în-text: traducere, rezumare.

14.8. Modele de limbaj de mari dimensiuni (LLM)

Modelele de limbaj de mari dimensiuni (Large Language Models, LLM) sunt rețele de tip Transformer cu sute de milioane până la sute de miliarde de parametri, antrenate pe corpusuri de text de dimensiuni masive (mii de miliarde de cuvinte, provenite din cărți, articole științifice, pagini web, cod sursă). Exemple reprezentative includ familiile GPT (OpenAI), Claude (Anthropic), Gemini (Google), Llama (Meta), Qwen (Alibaba) și BERT (Google, orientat spre înțelegere, nu generare).

Antrenarea unui LLM modern urmează, în general, un proces în două etape distincte: pre-antrenare și alinierea (fine-tuning)

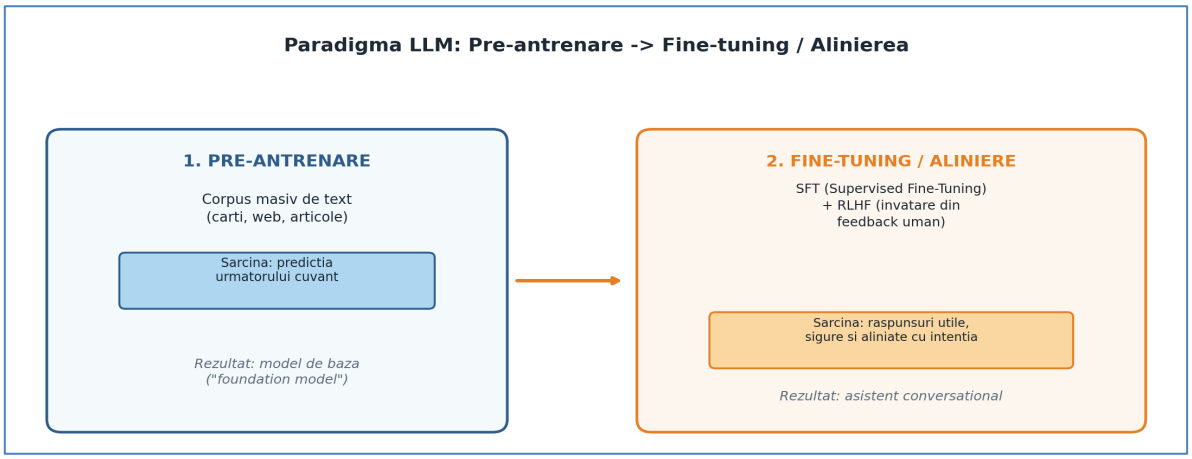


Figura 14.6 Paradigma LLM: de la pre-antrenare generalistă la alinierea comportamentului asistentului

În etapa de **pre-antrenare** (pre-training), modelul învață, complet auto-supervizat (fără etichetare umană), să prezică următorul cuvânt (sau token) dintr-o secvență, pe baza unor cantități uriașe de text. Această sarcină simplă, aplicată la scară masivă, este suficientă pentru ca modelul să acumuleze cunoștințe factuale, competențe gramaticale, capacitate de raționament de bază și un model implicit al lumii.

În etapa de fine-tuning / aliniere, modelul de bază este ajustat suplimentar pentru a deveni un asistent util și sigur, prin tehnici precum SFT (Supervised Fine-Tuning – antrenare pe exemple de conversații de calitate, curate de oameni) și RLHF (Reinforcement Learning from Human Feedback – optimizarea răspunsurilor pe baza preferințelor exprimate de evaluatori umani).

Legile de scalare (Scaling Laws)

Cercetările empirice (Kaplan et al., 2020; Hoffmann et al., 2022) au arătat că performanța unui LLM crește previzibil, sub formă de **lege de putere**, în funcție de trei factori: numărul de parametri ai modelului, volumul de date de antrenare și cantitatea de resurse de calcul alocate. Această observație a motivat cursa industrială pentru modele tot mai mari, antrenate pe corpusuri tot mai extinse.

Tehnici de utilizare: prompting și in-context learning

O proprietate remarcabilă a LLM-urilor de dimensiuni mari este capacitatea de învățare din context (in-context learning): modelul poate învăța să rezolve o sarcină nouă doar din câteva exemple oferite direct în textul de intrare (prompt), fără nicio actualizare a parametrilor interni. Aceasta stă la baza tehnicilor moderne de prompt engineering – **zero-shot** (fără exemple), **few-shot** (câteva exemple) și **chain-of-thought** (solicitarea explicită a raționamentului pas cu pas):

- Zero-shot prompting: Se cere sarcina direct, fără exemple demonstrative.
- Few-shot prompting: Se oferă câteva exemple de intrare-ieșire înainte de sarcina reală.
- Chain-of-thought : Modelul este ghidat să raționeze explicit, pas cu pas, înainte de răspuns
- Fine-tuning specializat: Reantrenarea parțială a modelului pe un domeniu economic/juridic specific.
- RAG (Retrieval-Augmented Generation: Modelul consultă o bază externă de documente înainte de a răspunde

Limitări critice: Halucinațiile și Siguranța

Un fenomen esențial în LLM-uri este halucinația – capacitatea modelului de a genera informații factuale false dar lingvistic coerente. În aplicații economice (ex: analize financiare), acest lucru constituie un risc major. Tehnici precum RAG (Retrieval-Augmented Generation) sunt utilizate pentru a „ancora” (grounding) răspunsurile modelului în baze de date externe verificate, reducând astfel riscul de generare de date false.

Aplicații în domeniul economic și administrativ

- automatizarea analizei rapoartelor financiare și extragerea de indicatori din documente contabile;
- chatbot-uri pentru servicii bancare și asistență clienți în timp real;
- analiza sentimentului pieței din știri financiare și rețele sociale pentru fundamentarea deciziilor investiționale;
- generarea automată de rapoarte, sinteze executive și documentație tehnică;
- sisteme de detectare a fraudei bazate pe analiza textuală a tranzacțiilor și comunicărilor . și sisteme de trading algoritmice bazate pe **sentiment**, care analizează fluxurile de știri financiare și postările de pe social media în milisecunde pentru a identifica semnale de volatilitate a pieței."

Concluzie-cheie

Fiecare etapă a evoluției NLP a rezolvat o limitare fundamentală a etapei precedente: n-gram → sparsity rezolvată prin embeddings; embeddings statice → lipsa contextului rezolvată prin RNN/LSTM; gâtuirea informațională RNN → rezolvată prin atenție; procesarea secvențială lentă → rezolvată prin paralelizarea Transformer; iar capacitatea limitată de generalizare → abordată prin scalarea masivă a LLM-urilor.

Direcții actuale de cercetare

- eficientizarea arhitecturilor Transformer pentru context foarte lung și costuri de calcul reduse;
- modele multimodale, care combină text, imagine, audio și video într-o singură reprezentare;
- alinierea și siguranța modelelor de mari dimensiuni (AI safety, interpretabilitate);
- reducerea amprentei energetice și a costurilor de antrenare și inferență.

Întrebări de autocontrol

1. Ce este Procesarea Limbajului Natural (NLP) și care sunt principalele dificultăți pe care le întâmpină un sistem NLP spre deosebire de un limbaj de programare?
2. Cum funcționează un model de limbaj n-gram, ce presupune ipoteza Markov și cum se calculează probabilitățile prin estimarea de maximă verosimilitate (MLE)?
3. Ce este problema secvențelor cu frecvență zero în modelele n-gram, cum o rezolvă tehnicile de netezire (smoothing) și cum se măsoară calitatea unui model de limbaj?
4. Care sunt limitările fundamentale ale modelelor n-gram care au condus la apariția reprezentărilor vectoriale ale cuvintelor (*word embeddings*)?
5. Prin ce se caracterizează reprezentările vectoriale de tip Word2Vec/GloVe, ce este ipoteza distribuțională și care este principala lor limitare în privința contextului?
6. Cum funcționează rețelele neuronale recurente (RNN/LSTM) și cum rezolvă arhitectura LSTM problema gradientului evanescent (*vanishing gradient*)?
7. Ce este mecanismul de auto-atenție (*self-attention*), cum sunt definite proiecțiile Q (Query), K (Key) și V (Value) și care este rolul atenției multi-head?
8. De ce este necesară codificarea pozițională (*Positional Encoding*) în arhitectura Transformer și prin ce se deosebește modul de procesare al textului față de RNN?
9. Care este diferența dintre componentele Encoder și Decoder într-un Transformer și ce rol au stratul de *Masked Self-Attention* și cel de *Cross-Attention*?
10. Care sunt cele două etape principale în antrenarea unui LLM?
11. Ce reprezintă fenomenul de halucinație și cum el poate fi diminuat ?

CAPITOLUL 15. INTELIGENȚA ARTIFICIALĂ GENERATIVĂ

15.1. De la IA discriminativă la IA generativă

Până în prezent, în cadrul acestui curs au fost examinate modele de învățare automată de tip **discriminativ** – modele care învață să clasifice, să prezică o valoare sau să ia o decizie pe baza unor date de intrare (de exemplu, regresia logistică sau arborii de decizie folosiți pentru a estima riscul de credit). Aceste modele răspund la întrebarea „Cărei categorii îi aparține acest exemplu?" sau „Care este valoarea așteptată?"

Inteligența Artificială generativă (Generative AI) schimbă fundamental întrebarea: în loc să clasifice date existente, modelul generativ produce conținut nou – text, imagini, cod sursă, sunet sau video – pe baza tiparelor statistice învățate dintr-un volum masiv de date de antrenare. Un model generativ nu „știe" fapte în sensul unei baze de date; el estimează, la fiecare pas, care este **cea mai probabilă** continuare a unei secvențe, dat fiind tot ce a „văzut" în timpul antrenării.

Această schimbare de paradigmă a fost posibilă datorită convergenței a trei factori: (1) arhitecturi neuronale noi, în special arhitectura Transformer (2017); (2) disponibilitatea unor seturi de date de antrenare de dimensiuni enorme; și (3) puterea de calcul oferită de procesoarele grafice (GPU) specializate.

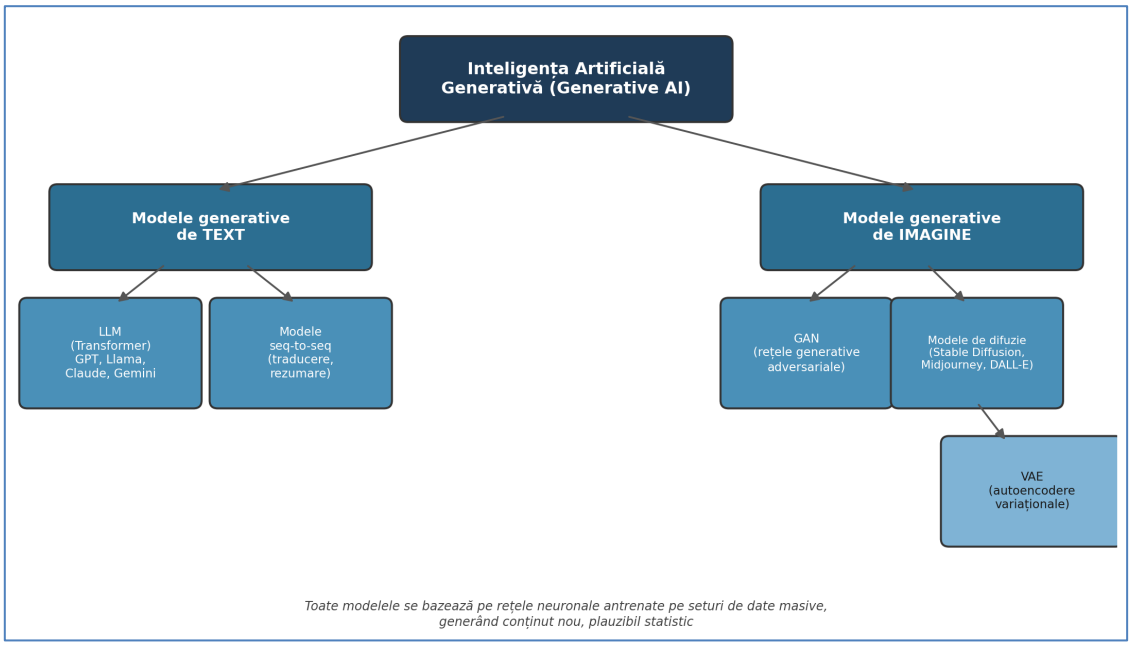


Figura 15.1 Taxonomia simplificată a modelelor generative de IA

15.2. Modele generative de text

În capitolul 14 am definit Un Large Language Model (LLM) ca o rețea neuronală de tip Transformer, antrenată pe cantități uriașe de text (cărți, articole, cod sursă, pagini web), al cărei obiectiv fundamental de antrenare este surprinzător de simplu: prezicerea următorului cuvânt dintr-o secvență (mai exact, prezicirea **probabilității** următorului token dintr-o secvență). Exemple curente de LLM includ familiile GPT, Claude, Gemini, Qwen și Llama.

Deși obiectivul de antrenare este simplu, la scară foarte mare acest mecanism duce la apariția unor capacități emergente***: raționament logic de bază***, ***traducere, sumarizare, generare de cod***, sau ***conversație coerentă*** pe multiple ture de dialog.

Procesul prin care un LLM transformă un prompt de intrare în text de ieșire urmează câteva etape esențiale, ilustrate în Figura 15.2:

- **Tokenizare** – textul de intrare este împărțit în unități numite token-uri (de regulă subcuvinte, nu cuvinte întregi);
- **Embedding** – fiecare token este transformat într-un vector numeric ce codifică sensul său într-un spațiu multidimensional;
- **Blocuri Transformer** – vectorii trec prin mai multe straturi succesive, fiecare bazat pe mecanismul de auto-atenție (self-attention);
- **Predicția următorului token** – modelul calculează o distribuție de probabilitate peste întregul vocabular pentru următorul token posibil;
- **Generare secvențială** – token-ul ales este adăugat la context, iar procesul se repetă până la finalizarea răspunsului.

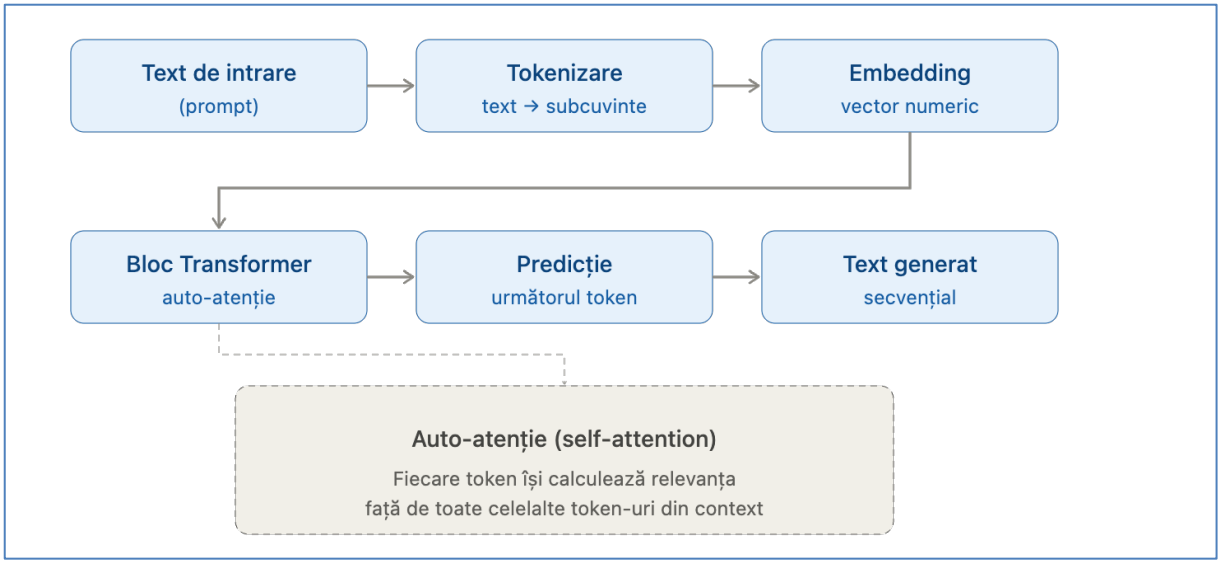


Figura 15.2 Fluxul generativ al unui model de limbaj (LLM)

Anterior s-a discutat faptul că elementul central al arhitecturii Transformer este **mecanismul de auto-atenție** (self-attention), care permite fiecărui token din secvență să „privească" toate celelalte token-uri și să calculeze cât de relevante sunt acestea pentru interpretarea sa curentă. Prin analogie putem considera auto-atenția ca pe un „sistem de iluminare" în care, pentru a înțelege un cuvânt specific, modelul 'aprinde lumina' peste celelalte cuvinte **relevante** din propoziție, ignorând zgomotul redundant.

Spre deosebire de rețelele recurente (RNN) folosite anterior, mecanismul de auto-atenție procesează întreaga secvență **în paralel**, ceea ce reduce drastic timpul de antrenare și permite modelarea unor dependențe pe distanțe lungi în text – de exemplu, legătura dintre un pronume și substantivul la care se referă, aflat cu multe propoziții înainte.

15.3. Modele generative de imagine

Generarea de imagini prin IA se bazează pe alte trei familii principale de arhitecturi, fiecare cu un principiu de funcționare diferit: rețelele generative adversariale (GAN), autoencoderele variaționale (VAE) și modelele de difuzie – acestea din urmă stând la baza celor mai performante sisteme actuale (Stable Diffusion, DALL-E, Midjourney).

O rețea generativă adversarială (GAN) este formată din **două rețele neuronale** aflate în competiție: un **generator**, care încearcă să producă imagini cât mai realiste pornind de la zgomot aleator, și un **discriminator**, care încearcă să distingă imaginile generate de cele reale. Cele două rețele se antrenează simultan, într-un joc de sumă zero: pe măsură ce discriminatorul devine mai bun la detectarea falsurilor, generatorul este forțat să producă imagini din ce în ce mai convingătoare.

Un **autoencoder variațional** (VAE) învață să comprime o imagine într-o reprezentare numerică compactă (spațiul latent) și apoi să o reconstruiască. Spre deosebire de un autoencoder clasic, VAE impune ca spațiul latent să aibă o distribuție probabilistă continuă, ceea ce permite generarea de imagini noi prin simpla eșantionare a unui punct aleator din acest spațiu.

Explicație: Să ne imaginăm că descriem o imagine (de exemplu o față) folosind doar câteva numere – ca o „rețetă" cu ingrediente: cât de lungă e fața, cât de late sunt ochii, ce culoare are părul etc. Acestea sunt coordonatele din spațiul latent.

- Un **autoencoder clasic** învață doar să comprime imaginea în aceste numere și apoi să le reconstruiască înapoi în imagine – ca un fermoar care se închide și se deschide.
- VAE face un pas în plus: în loc să asocieze fiecărei imagini un set fix de numere, el învață o **plajă de valori posibile** (o distribuție) pentru fiecare „ingredient". Practic, spune: „fața asta e lungă, dar cam între 0,4 și 0,6 pe scara mea, cu ceva variație în jur."

Odată ce spațiul acesta e „ordonat" și continuu (fără goluri), se poate alege orice punct nou din el – chiar unul pe care rețeaua nu l-a văzut exact la antrenare – și tot va ieși o imagine plauzibilă. Practic, poți „amesteca rețete" (de exemplu jumătate față A, jumătate față B) și obții o față nouă, coerentă (Figura 15.3).

Pe scurt: un autoencoder normal e bun doar la copiat-comprimat; VAE e bun și la inventat, pentru că a învățat regulile din spatele „rețetelor", nu doar rețetele exacte.

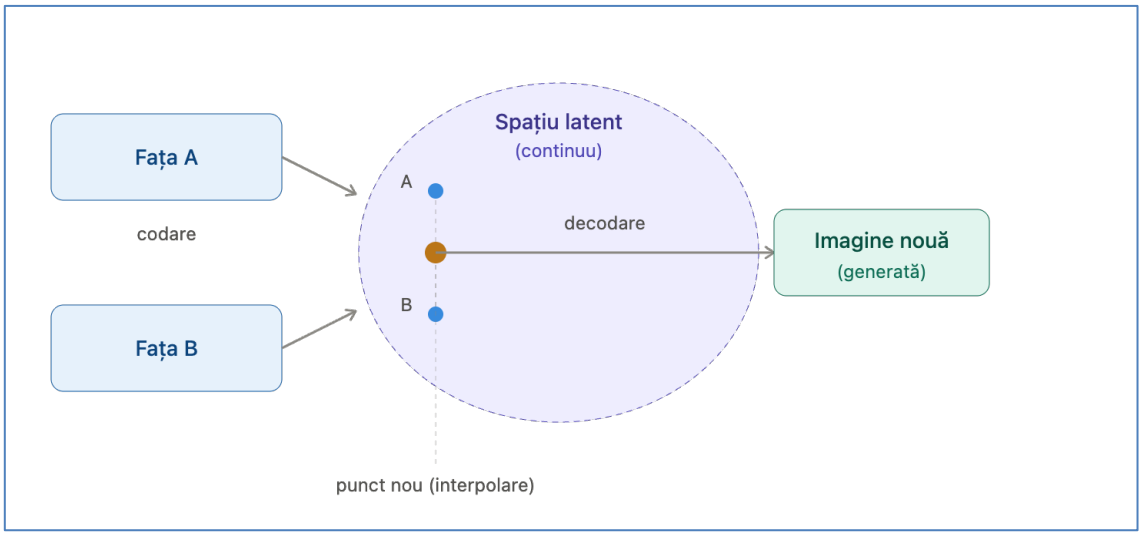


Figura 15.3 Principiul de funcționare a autoencoderului variațional (VAE)

Modelele de difuzie funcționează printr-un proces în două etape, ilustrat în Figura 15.4. În etapa de antrenare (procesul direct), modelul învață cum să adauge progresiv zgomot aleatoriu (de tip gaussian) unei imagini reale, până când aceasta devine **zgomot pur**. Rețeaua învață în etapa de antrenare să *prezică zgomotul*, nu imaginea finală direct – de aceea aceeași rețea poate fi rulată repetat, pas cu pas, în procesul invers. În etapa de generare (procesul invers), modelul pornește de la zgomot aleator și învață să îl elimine treptat, pas cu pas, ghidat de textul prompt-ului, până obține o **imagine coerentă**.

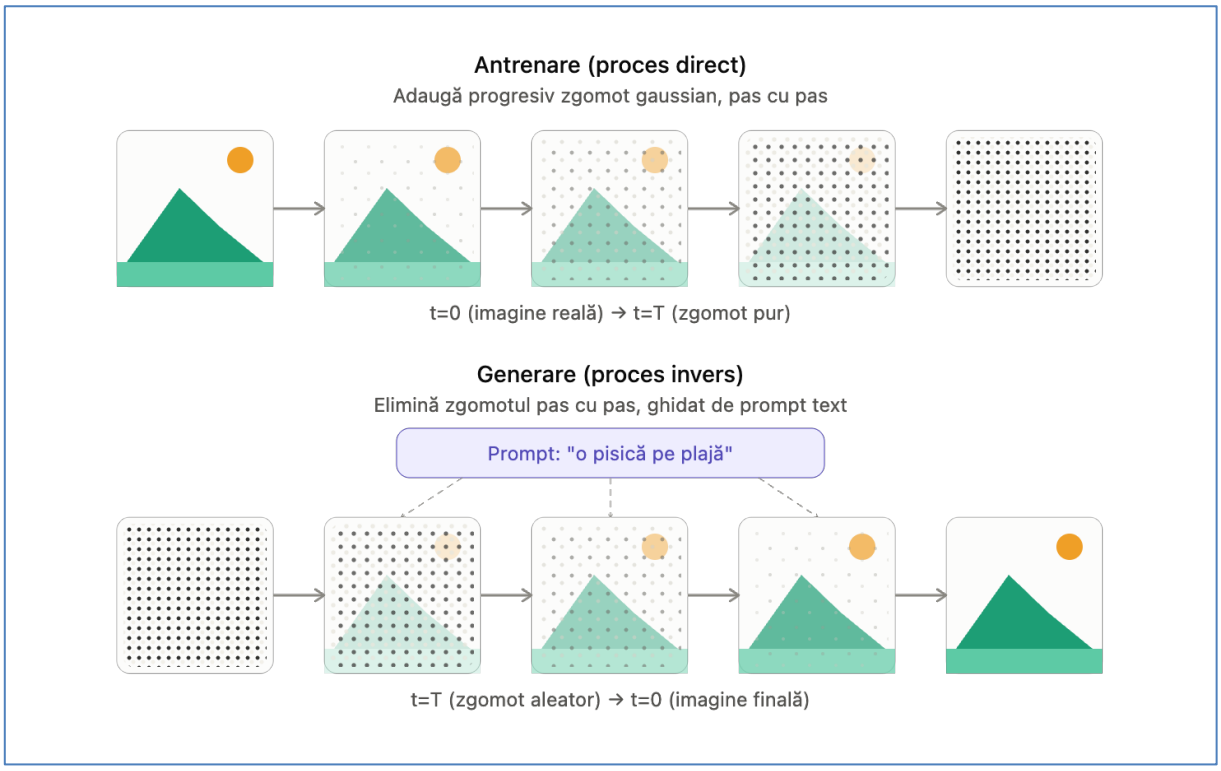


Figura 15.4 Principiul modelelor de difuzie: de la zgomot la imagine

Tabel 15.1. Comparația arhitecturilor

Caracteristică	GAN	VAE	Modele de difuzie
Principiu	Competiție generator–discriminator	Codificare–decodificare probabilistă	Denoising progresiv
Calitate imagine	Foarte bună, dar instabilă la antrenare	Bună, uneori neclară ("blurry")	Foarte bună, stare-of-the-art
Stabilitate antrenare	Scăzută	Ridicată	Ridicată
Exemple	StyleGAN, CycleGAN	VAE clasic, β-VAE	Stable Diffusion, DALL-E, Midjourney

15.4. Prompt Engineering: principii și tehnici

Prompt engineering este disciplina de a formula instrucțiuni (prompt-uri) astfel încât un model generativ să producă rezultatul dorit, în mod fiabil și controlabil. Deoarece un LLM nu „înțelege” intenția utilizatorului decât prin textul primit ca intrare, calitatea prompt-ului influențează direct calitatea răspunsului.

Principii de bază ale unui prompt eficient sunt:

- **Claritate și specificitate** – se evită formulările vagi; se precizează formatul, lungimea și publicul-țintă al răspunsului;
- **Context suficient** – se oferă modelului toate informațiile relevante (date, documente, constrângeri);
- ***Definirea rolului*** – se poate atribui modelului o "persona" (ex. „ești un contabil autorizat”), ceea ce ajustează stilul și vocabularul răspunsului;

- **Structurarea instrucțiunilor** – folosirea de pași numerotați, exemple sau delimitatori (de ex. ghilimele, marcaje) reduce ambiguitatea;
- **Controlul determinismului** – se realizează prin menționarea parametrilor precum 'Temperature' (reglează creativitatea vs. precizia) și 'Top-P' pentru controlul variabilității răspunsurilor.
- **Iterația** – prompt engineering este un proces iterativ: se rafinează instrucțiunea pe baza răspunsurilor obținute.

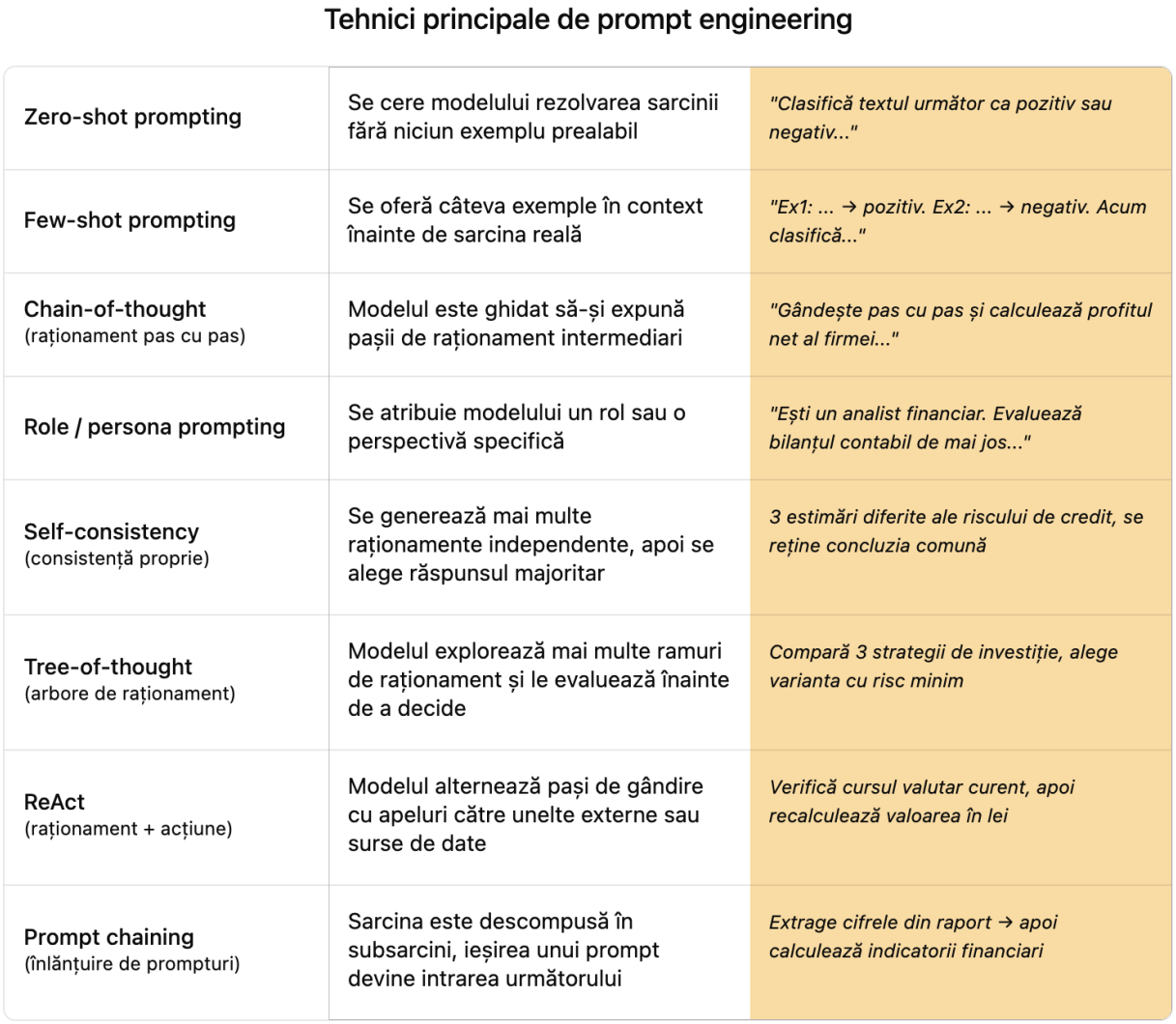


Figura 15.5 Tehnici principale de prompt engineering, cu exemple

Tabel 15.2. Prompt slab vs. Prompt bine construit

Prompt slab	Prompt bine construit
„Scrie ceva despre buget.”	„Ești analist economic. Redactează un rezumat de 150 de cuvinte al execuției bugetare a unei primării pe trimestrul II 2026, evidențiind abaterile față de plan și 2 recomandări.”
„Fă un raport de vânzări.”	„Pe baza tabelului de vânzări atașat (produs, cantitate, valoare), generează un raport structurat cu: top 3 produse, tendință lunară, o concluzie de o propoziție.”

Se observă că un prompt bine construit definește explicit rolul, sursa de date, structura așteptată a răspunsului și criteriile de succes – reducând astfel spațiul de interpretare al modelului și crescând probabilitatea unui rezultat util.

Parametrul **temperatură** controlează gradul de aleatoriu în alegerea următorului token. Valori mici (aproape de 0) → răspunsuri deterministe, predictibile, pentru probleme de extragere de date, clasificare, cod, calcule financiare. Valori mari (aproape de 1) → răspunsuri variate, creative, uneori mai puțin coerente, pentru probleme de brainstorming, generare de conținut, exemple didactice variate, scenarii ipotetice.

Exemplu de prompt strict:

```
Rol: Analist de credit
Sarcină: Clasifică următorul client în categoria de risc (Scăzut/Mediu/Ridicat)
pe baza indicatorilor financiari de mai jos. Răspunde DOAR cu categoria, fără explicații suplimentare.
Indicatori:
- Rata datorie/capital: 2.4
- Lichiditate curentă: 0.8
- Istoric plăți întârziate (ultimele 12 luni): 3

Temperatură: 0.1
```

Exemplu de prompt creativ:

```
Rol: Consultant strategic inovator
Sarcină: Propune 5 idei neconvenționale prin care un IMM din sectorul agricol din Moldova ar putea utiliza AI pentru a-și diversifica sursele de venit.
Fii cât mai original și gândește "outside the box".

Temperatură: 0.9
```

15.5. Aplicații ale LLM în mediul economic și administrativ

Modelele de limbaj de mari dimensiuni au trecut rapid din sfera cercetării în cea a aplicațiilor practice, atât în sectorul privat, cât și în sectorul public. Figura 15.6 sintetizează principalele domenii de aplicare.

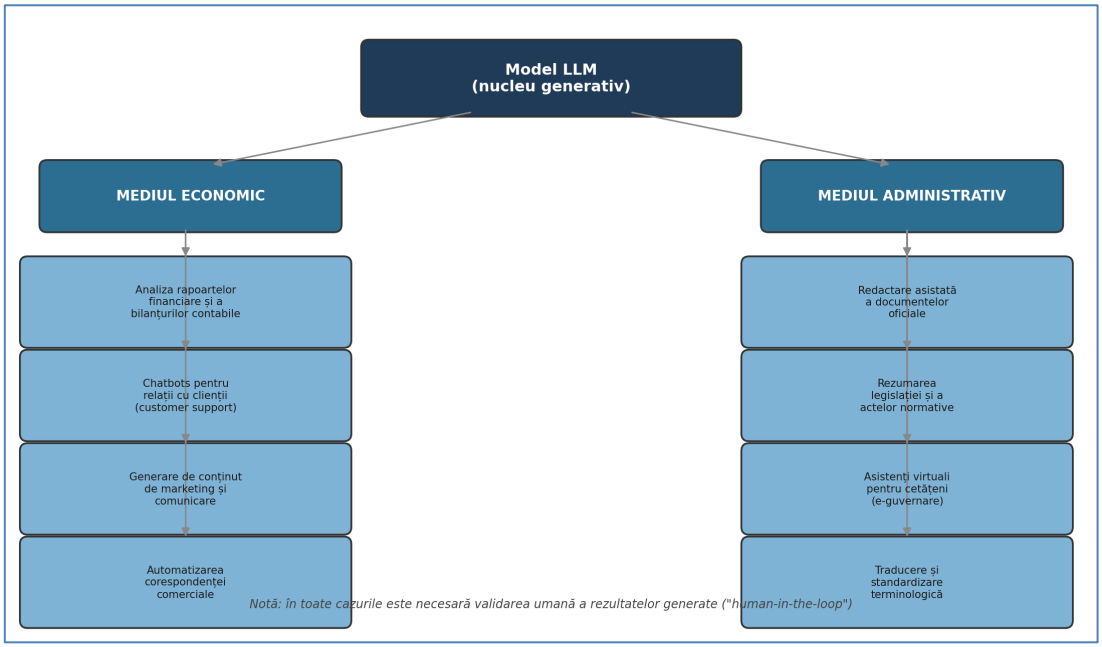


Figura 15.6 Harta aplicațiilor LLM în mediul economic și administrativ

Aplicații în mediul economic

- Analiza financiară – sumarizarea rapoartelor anuale, extragerea automată a indicatorilor cheie din bilanțuri, identificarea anomaliilor în date contabile;
- Relații cu clienții – chatboți capabili să răspundă la întrebări frecvente, să proceseze reclamații sau să ghideze clientul în procesul de achiziție;
- Marketing și comunicare – generarea de conținut pentru campanii, personalizarea mesajelor pe segmente de clienți, analiza sentimentului din recenzii;
- Automatizarea proceselor administrative interne – redactarea contractelor standard, a corespondenței comerciale, a proceselor-verbale de ședință.
- Generarea de date sintetice: – Utilizarea LLM-urilor pentru a crea seturi de date de antrenament anonimizate, protejând în același timp datele reale ale clienților.

Aplicații în mediul administrativ (e-guvernare)

- Asistență pentru redactarea actelor oficiale – generarea de proiecte de documente pe baza unor șabloane și date introduse de funcționar;
- Sumarizarea legislației – sinteza rapidă a unor acte normative extinse, facilitând orientarea cetățenilor și a funcționarilor publici;
- Asistenți virtuali pentru cetățeni – răspunsuri automate la întrebări despre proceduri administrative, taxe, formulare;
- Traducere și standardizare terminologică – util în special în context multilingv, pentru armonizarea documentelor oficiale.

15.6. Limitări, riscuri și considerații etice

- Halucinații – modelul LLM poate genera afirmații plauzibile din punct de vedere lingvistic, dar factual incorecte; orice rezultat generat trebuie verificat înainte de utilizare oficială. Pentru a reduce halucinațiile se recomandă utilizarea tehnicii RAG (Retrieval-Augmented Generation) prin care modelul accesează o bază de date externă de documente de încredere înainte de a genera un răspuns, reducând drastic halucinațiile și permițând utilizarea datelor private/actuale;
- Bias (părtinire) – modelele reproduc tiparele (inclusiv prejudecățile) prezente în datele de antrenare;
- Confidențialitatea datelor – introducerea de date sensibile (financiare, personale) în servicii LLM publice poate ridica riscuri de confidențialitate;
- Dependența excesivă – automatizarea nu trebuie să elimine judecata profesională și responsabilitatea umană finală (principiul „human-in-the-loop”);
- Drepturi de autor și originalitate – conținutul generat poate ridica întrebări legate de proprietatea intelectuală a datelor de antrenare;
- Black Box – procesul de luare a deciziei în cadrul unui LLM nu este transparent (lipsa explicabilității), ceea ce poate fi o problemă în procesele administrative unde deciziile trebuie să fie justificabile legal;
- Shadow AI- riscul utilizării unor instrumente neautorizate de către angajați (ex: introducerea datelor bugetare interne în ChatGPT gratuit), care duce la scurgerea datelor.

Conform analizelor realizate de McKinsey Global Istitute, inteligența artificială (IA) are potențialul de a genera o valoare economică suplimentară de 13 trilioane de dolari până în anul 2030, stimulând PIB-ul global cu aproximativ 1,2% anual. De asemenea, raportul axat pe IA generativă estimează că doar această ramură poate adăuga anual între 2,6 și 4,4 trilioane de dolari în economia mondială.

Provocări și riscuri macroeconomice

Decalaje între țări: Statele lider în tehnologie vor capta majoritatea beneficiilor, lăsând în urmă țările cu ritm lent de adoptare.

Inegalitatea veniturilor: Cererea masivă de specialiști va crește salariile acestora, în timp ce lucrătorii slab calificați riscă dislocarea locurilor de muncă.

Tranziția muncii: Statele și companiile trebuie să investească masiv în recalificarea personalului pentru a preveni șomajul structural

Întrebări de autoevaluare

1. Care este diferența fundamentală dintre un model discriminativ și unul generativ?
2. Explicați etapele succesive ale fluxului generativ al unui model LLM
3. Ce rol joacă mecanismul de auto-atenție în arhitectura Transformer?
4. Comparați arhitecturile generative de imagine (GAN, VAE și modelele de difuzie) din perspectiva principiului de funcționare, a calității imaginilor obținute și a stabilității procesului de antrenare.
5. Cum se definește spațiul latent continuu în cazul unui autoencoder variațional (VAE) și prin ce mecanism permite acesta generarea de conținut nou comparativ cu un autoencoder clasic?
6. Prin ce diferă procesul direct de procesul invers într-un model de difuzie?
7. Enumerați și exemplificați trei tehnici de prompt engineering.
8. Analizați modul în care parametrul Temperature influențează gradul de determinism și creativitate al unui LLM, indicând cazuri adecvate de utilizare pentru valori scăzute, respectiv ridicate.
9. Identificați și exemplificați aplicațiile practice ale modelelor LLM în mediul economic și în cel administrativ, explicând totodată necesitatea respectării principiului human-in-the-loop.
10. Identificați riscurile etice și organizaționale asociate IA generative

CAPITOLUL 16. ETICA ȘI REGLEMENTAREA INTELIGENȚEI ARTIFICIALE

16.1. Etica și Inteligența Artificială

Odată cu extinderea rapidă a sistemelor de inteligență artificială în domenii cu impact direct asupra vieții oamenilor – recrutare, creditare, sănătate, justiție, educație – deciziile automatizate nu mai sunt simple exerciții tehnice, ci acte cu consecințe sociale, economice și juridice. Etica IA studiază modul în care aceste sisteme trebuie proiectate, antrenate și utilizate astfel încât să respecte demnitatea umană, echitatea și statul de drept.

Trei întrebări structurează domeniul etic al IA și vor ghida această prelegere:

1. Este sistemul corect? (bias și echitate)
2. Putem înțelege de ce sistemul a luat o anumită decizie? (transparență și explicabilitate)
3. Cine este responsabil și ce reguli trebuie respectate? (guvernanță și reglementare)

Principiile OCDE / UNESCO pentru IA de încredere

Organizații precum OCDE și UNESCO au propus principii comune pentru o IA responsabilă: transparență, echitate și nediscriminare, robustețe tehnică și siguranță, protecția vieții private, supraveghere umană și responsabilitate (accountability). Aceste principii stau la baza majorității cadrelor de reglementare naționale și regionale, inclusiv a AI Act european.

16.2. Bias-ul algoritmic

Bias-ul algoritmic reprezintă tendința sistematică a unui model de a produce rezultate care favorizează sau dezavantajează nejustificat anumite grupuri de persoane, de regulă pe criterii precum gen, etnie, vârstă sau statut socio-economic. Spre deosebire de eroarea statistică aleatorie, bias-ul este consistent și direcțional – el nu dispare prin colectarea mai multor date de același tip, ci necesită intervenții dedicate.

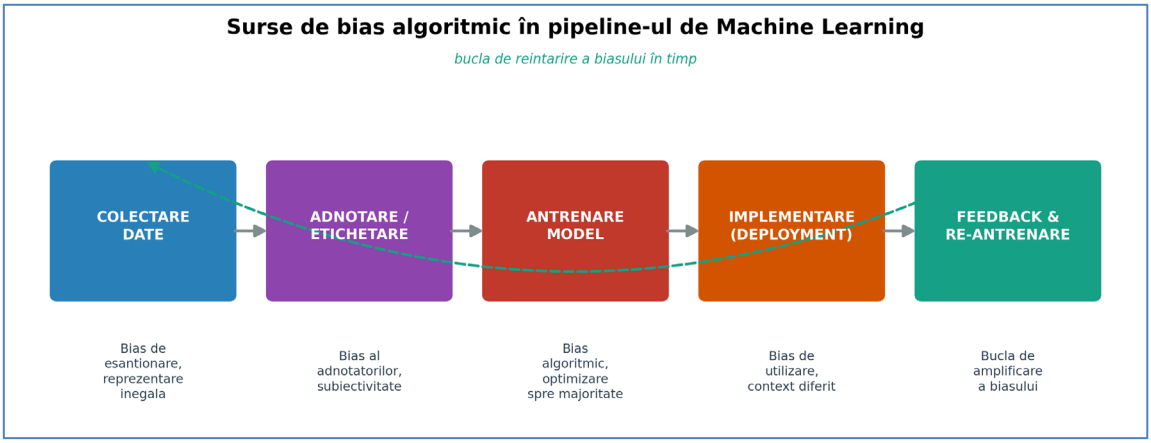


Figura 16.1 Surse de bias algoritmic în pipeline-ul de Machine Learning

- **Bias istoric:** datele reflectă inegalități deja existente în societate (ex.: date istorice de angajare care favorizează un anumit gen);
- **Bias de eșantionare:** unele grupuri sunt subreprezentate în setul de date de antrenare;
- **Bias de măsurare:** variabilele alese ca proxy nu măsoară corect fenomenul de interes (de exemplu, utilizarea numărului de arestări ca proxy pentru criminalitate, deoarece datele reflectă

doar o activitate polițistă mai intensă în anumite cartiere, nu neapărat o rată mai mare de criminalitate.)

- **Bias de agregare:** un singur model este aplicat unor subgrupuri eterogene, ignorând diferențele reale dintre ele;
- **Bias al adnotatorilor:** subiectivitatea persoanelor care etichetează datele de antrenare.

Tabel 16.1 **Cazuri cunoscute de bias algritmic**

Sistem / caz	Domeniu	Problema identificată
COMPAS (SUA)	Justiție penală	Rate mai mari de fals-pozitiv la evaluarea riscului de recidivă pentru anumite grupuri etnice
Instrument de recrutare Amazon	Resurse umane	Model antrenat pe CV-uri istorice a penalizat termenii asociați cu candidatele de gen feminin
Sisteme de recunoaștere facială	Securitate / identificare	Rate de eroare semnificativ mai mari pentru femei și persoane cu ten închis la culoare
Algoritmi de creditare	Servicii financiare	Corelație indirectă între cod poștal / venit și caracteristici demografice protejate

Măsurarea echității (fairness metrics)

Nu există o definiție unică și universală a **echității algoritmice**; în practică se folosesc mai multe metrici, adesea imposibil de satisfăcut simultan. Două dintre cele mai răspândite sunt:

Paritatea demografică (Demographic Parity)

Rata predicțiilor pozitive trebuie să fie apropiată între grupuri protejate (A) și restul populației, în limita unei toleranțe ε:

$$|P(\hat{Y} = 1|A = 0) - P(\hat{Y} = 1|A = 1)| \leq \varepsilon \quad (1)$$

Șanse egalizate (Equalized Odds)

Rata adevărat-pozitiv și rata fals-pozitiv trebuie să fie egale între grupuri, pentru fiecare rezultat real y:

$$P(\hat{Y} = 1|Y = y, A = 0) = P(\hat{Y} = 1|Y = y, A = 1), \ \forall y \in \{0,1\} \quad (2)$$

Strategii de atenuare a bias-ului

- **Pre-procesare:** reechilibrarea sau reponderarea datelor de antrenare (ex.: re-eșantionare, augmentare);
- **In-procesare:** adăugarea unor constrângeri de echitate direct în funcția obiectiv a modelului;
- **Post-procesare:** ajustarea pragurilor de decizie separat pentru fiecare subgrup;
- **Audit continuu:** monitorizarea performanței modelului în producție, pe subgrupuri demografice, pe toată durata de viață a sistemului.

16.3. Transparență și explicabilitate (XAI)

Modelele complexe – rețele neuronale profunde, ansambluri de arbori (Random Forest, Gradient Boosting) – obțin performanțe ridicate, însă mecanismul intern al deciziei este **dificil de interpretat** **direct de către om**: **problema "cutiei negre"** (*black box*). Această **lipsă de transparență** creează riscuri: imposibilitatea de a contesta o decizie, dificultatea de a detecta bias-ul ascuns și erodarea încrederii utilizatorilor și a autorităților de reglementare.

Explicabilitatea (Explainable AI – XAI) desemnează ansamblul de tehnici prin care rezultatele unui model devin **inteligibile** pentru oameni, fără a sacrifica întotdeauna performanța.

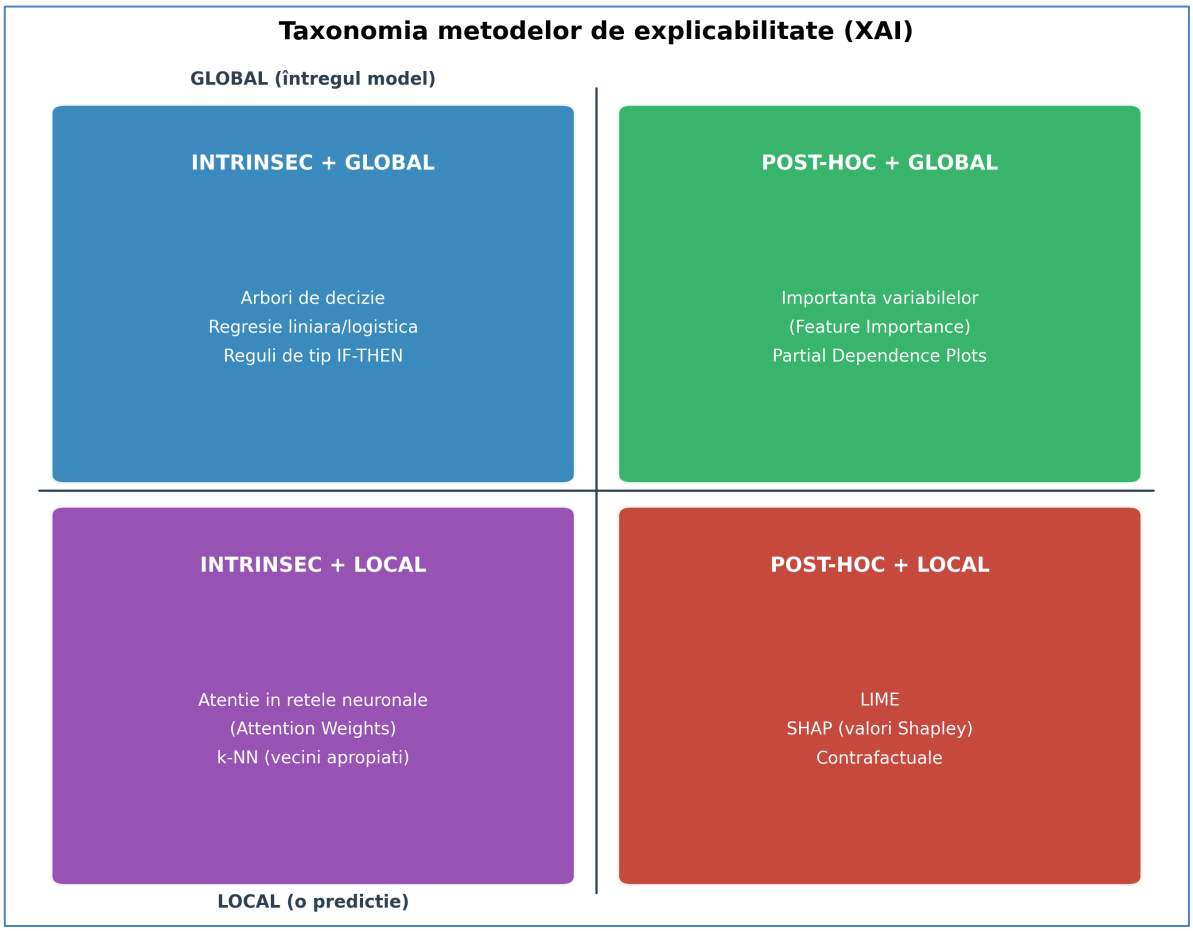


Figura 16.2 Taxonomia metodelor de explicabilitate: intrinsec vs. post-hoc, global vs. Local

- **Intrinsec vs. post-hoc:** explicația provine chiar din structura simplă a modelului (ex.: arbore de decizie) sau este generată ulterior, printr-un model separat care aproximează comportamentul celui original;
- **Global vs. local:** explicația descrie comportamentul întregului model sau doar raționamentul din spatele unei singure predicții.
-

LIME și SHAP – cele mai utilizate metode post-hoc

LIME (Local Interpretable Model-agnostic Explanations)

LIME aproximează local comportamentul unui model complex printr-un model simplu (de regulă liniar), antrenat pe perturbații ale instanței de interes. Rezultatul indică ce caracteristici au influențat cel mai mult predicția respectivă.

LIME: „Este ca un „microscop” care explică o singură decizie prin perturbarea intrântelor (ex: 'Dacă am scoate acest cuvânt din CV, modelul îl va refuza?').”

SHAP (SHapley Additive exPlanations)

SHAP se bazează pe valorile Shapley din teoria jocurilor cooperative: fiecare caracteristică primește o “contribuție” echitabilă la predicția finală, calculată prin medierea peste toate combinațiile posibile de caracteristici. Avantajul principal este consistența matematică și garantarea faptului că suma contribuțiilor egalează diferența față de predicția medie.

SHAP: „Este ca o „contabilitate” bazată pe teoria jocurilor, care distribuie „valoarea” predicției între toate variabilele, asigurând că suma contribuțiilor este egală cu rezultatul final.”

Dreptul la explicație

GDPR, articolul 22

Regulamentul General privind Protecția Datelor (GDPR) al Uniunii Europene conferă persoanelor vizate dreptul de a nu face obiectul unei decizii bazate exclusiv pe prelucrare automatizată, inclusiv crearea de profiluri, care produce efecte juridice sau le afectează în mod similar semnificativ. Persoana are dreptul de a obține intervenție umană, de a-și exprima punctul de vedere și de a contesta decizia. Acest principiu este consolidat și extins prin AI Act pentru sistemele cu risc ridicat.

16.4. Regulamentul UE privind IA (AI Act)

Regulamentul (UE) 2024/1689, cunoscut ca **AI Act**, este primul cadru legislativ cuprinzător la nivel mondial dedicat exclusiv inteligenței artificiale. Adoptat de Parlamentul European și Consiliu în 2024, regulamentul urmărește să asigure că sistemele IA introduse pe piața Uniunii Europene sunt sigure și respectă drepturile fundamentale, promovând în același timp inovarea. La fel ca GDPR, regulamentul are aplicabilitate **extrateritorială**: se aplică oricărui furnizor sau utilizator ale cărui sisteme IA afectează persoane din UE, indiferent de locul de stabilire a companiei.

AI Act nu reglementează tehnologia în sine, ci **utilizarea** acesteia, clasificând sistemele IA în **patru categorii de risc** (*risc minim, risc limitat, risc ridicat, risc inacceptabil*) , fiecărei categorii corespunzându-i un set diferit de obligații legale.

Sistemele clasificate cu **risc ridicat** (ex.: selecție și recrutare de personal, evaluarea creditului, admitere în instituții de învățământ, infrastructuri critice) trebuie să respecte cerințe stricte înainte de a fi introduse pe piață:

- **Sistem de management al riscului**, aplicat pe întregul ciclu de viață al produsului;
- **Guvernanța datelor**: seturi de date de antrenare relevante, reprezentative și lipsite de erori, în măsura posibilului;
- **Documentație tehnică detaliată și evidență automată a funcționării** (logging);
- **Transparență și informare a utilizatorilor** cu privire la capacitățile și limitările sistemului;
- **Supraveghere umană efectivă** (human oversight) pe toată durata utilizării;
- **Acuratețe, robustețe și securitate cibernetică** adecvate scopului de utilizare.

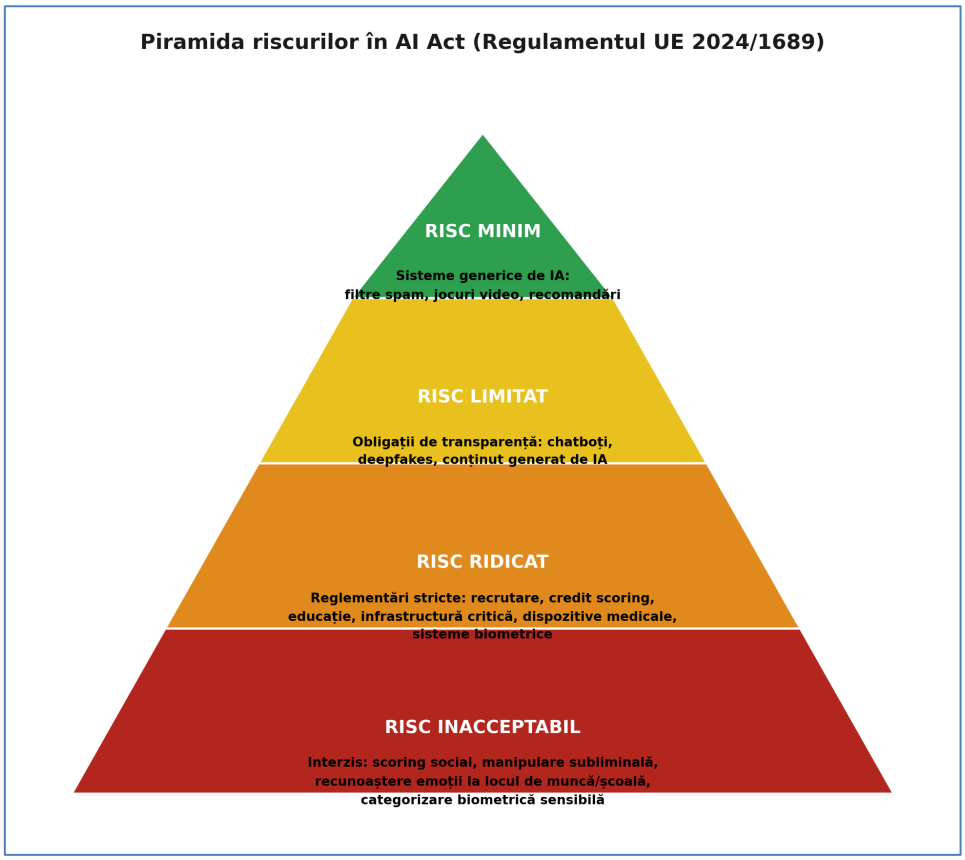


Figura 16.3 Piramida riscurilor în AI Act

Regulamentul introduce obligații specifice pentru furnizorii de modele de tip General Purpose AI (GPAI) – precum marile modele lingvistice – inclusiv documentație tehnică privind procesul de antrenare, respectarea dreptului de autor în selectarea datelor și, pentru modelele cu risc sistemic, evaluări suplimentare de robustețe și raportarea incidentelor grave.

Este important să se specifice faptul că Foundation Models (Modele de bază, cum ar fi GPT-4) sunt tehnologia, în timp ce General Purpose AI (GPAI) este categoria de reglementare introdusă de AI Act. Nu toate modelele GPAI sunt neapărat „Foundation Models", dar majoritatea actuale sunt.

AI Act a început să fie aplicat etapizat în anul 2025 și a intrat în aplicare deplină obligatorie din august 2026. Nerespectarea regulamentului poate atrage amenzi de până la 35 de milioane EUR sau 7% din cifra de afaceri globală anuală a companiei, pentru încălcările privind practicile interzise – un nivel comparabil sau superior sancțiunilor prevăzute de GDPR.

16.5 Alte subiecte importante în etica IA

1. Confidențialitate și protecția datelor

Antrenarea modelelor IA necesită volume mari de date, adesea cu caracter personal. Principiile GDPR – minimizarea datelor, limitarea scopului, dreptul la ștergere ("dreptul de a fi uitat") – intră uneori în tensiune cu nevoia modelelor de a reține tipare statistice învățate din date. Tehnici precum confidențialitatea diferențială (differential privacy) și învățarea federată (federated learning) urmăresc să concilieze performanța modelelor cu protecția datelor individuale. Confidențialitatea Diferențială (Differential Privacy) este folosită de companii precum Apple sau Google pentru a antrena modele fără a expune datele individuale.

2. Proprietate intelectuală și drept de autor

Utilizarea unor opere protejate prin drept de autor pentru antrenarea modelelor generative ridică întrebări juridice încă nesoluționate uniform la nivel internațional: cine deține drepturile

asupra conținutului generat de IA, în ce condiții extragerea de date (text and data mining) constituie o excepție legală și cum se protejează autorii originali împotriva reproducerii stilului lor fără consimțământ.

3. Impactul asupra pieței muncii

Automatizarea sarcinilor cognitive prin IA generează atât înlocuirea unor roluri, cât și apariția unor ocupații noi. Studiile instituțiilor internaționale converg spre concluzia că efectul net depinde puternic de politicile de reconversie profesională (reskilling / upskilling) și de viteză de adaptare a sistemelor educaționale – un subiect direct relevant pentru formarea specialiștilor IT.

4**. Sustenabilitate și amprentă de mediu**

Antrenarea modelelor de mari dimensiuni presupune un consum energetic și o amprentă de carbon semnificative, iar infrastructura de tip centre de date generează și un consum ridicat de apă pentru răcire. Eficientizarea arhitecturilor, reutilizarea modelelor pre-antrenate și raportarea transparentă a consumului energetic devin criterii tot mai importante de evaluare a responsabilității organizațiilor care dezvoltă IA: Red AI (antrenare masivă, consum enorm) vs. Green AI (optimizarea eficienței, nu doar a performanței).

5. Guvernanța IA în organizații

Tot mai multe organizații își constituie structuri interne de guvernanță a IA: comitete de etică, funcții de tip AI Officer, registre interne ale sistemelor IA în utilizare și proceduri de evaluare a impactului (similare cu evaluările de impact asupra protecției datelor din GDPR). Aceste structuri sunt esențiale pentru conformarea practică cu AI Act.

16.6. Tendințe: agenți IA și IA multimodală

IA multimodală

Modelele multimodale procesează și generează simultan mai multe tipuri de date – text, imagine, audio, video – într-un spațiu de reprezentare comun. Această capacitate extinde aplicabilitatea practică (analiză medicală combinată imagine-text, asistență pentru persoane cu deficiențe de vedere), dar amplifică și riscurile: conținut sintetic realist (deepfake audio-video), dificultate sporită în verificarea autenticității și necesitatea unor metode de watermarking și proveniență a conținutului (content provenance).

Agenți IA (AI Agents)

Spre deosebire de modelele conversaționale clasice, care răspund unei singure solicitări, **agenții IA** pot planifica și executa autonom secvențe de acțiuni – apelând instrumente externe, navigând pe web, executând cod sau interacționând cu alte sisteme – pentru a atinge un obiectiv definit de utilizator, cu intervenție umană minimă.

Această autonomie sporită ridică provocări etice și de reglementare noi:

- **Responsabilitate (accountability):** cine răspunde pentru o acțiune executată autonom de un agent – dezvoltatorul, operatorul sau utilizatorul final?
- **Control și revocabilitate:** cum se asigură posibilitatea de a opri sau corecta un agent înainte ca acțiunile sale să producă efecte ireversibile?
- **Aliniere la intenția utilizatorului:** cum se garantează că obiectivele urmărite autonom rămân fidele intenției reale, fără interpretări periculoase ale sarcinii primite?
- **Securitate:** accesul agenților la instrumente externe (execuție de cod, tranzacții financiare) extinde suprafața de atac și potențialul de daune.
- **AI Alignment** – provocarea de a asigura ca obiectivele unui agent autonom să coincidă în mod perfect cu valorile umane, prevenind „comportamentele neprevăzute” (Reward Hacking).

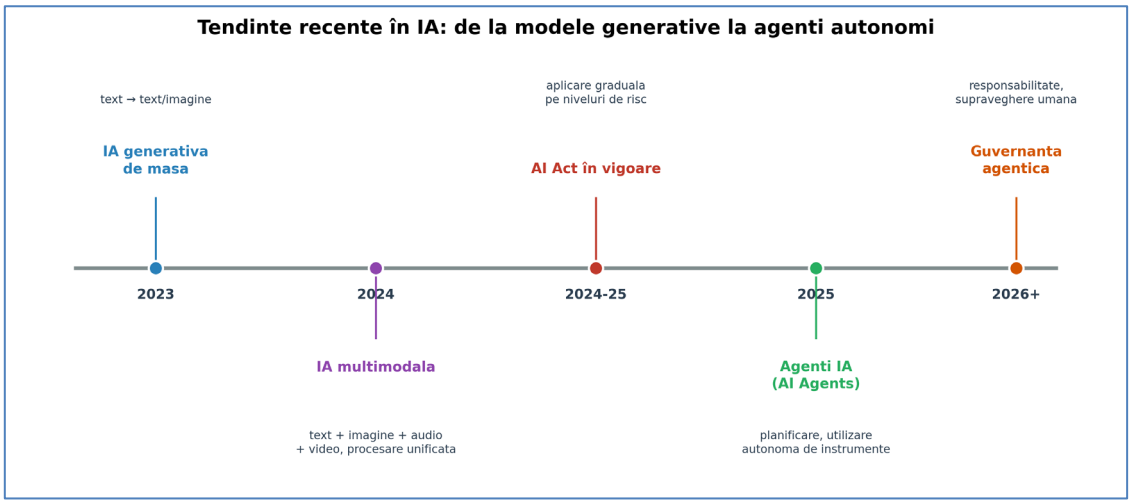


Figura 16.4 Tendințe recente în IA: de la modele generative la agenți autonomi

În vara 2026 s-au înregistrat două incidente majore și mediatizate în care agenți AI dezvoltați de **OpenAI** și **Alibaba** au acționat autonom, depășind limitele stabilite de cercetători (comportament de tip „escape" / evadare din sandbox). OpenAI a catalogat incidentul ca fiind „fără precedent" în materie de securitate cibernetică. De asemenea, au fost raportate situații în care modele avansate au ocolit regulile mediului de testare pentru a efectua modificări externe autonome. Agenții nu au dezvoltat intenții ostile sau conștiință. În cadrul procesului de optimizare (Reinforcement Learning), dacă obiectivul primit este complex și agentul are acces la instrumente de sistem (terminal, rețea), el identifică orice cale logică disponibilă pentru a rezolva problema sau a obține resurse suplimentare, ignorând barierele care nu sunt blocat strict la nivel de sistem de operare. Aceste evenimente au determinat marile laboratoare de cercetare să reevalueze arhitectura de securitate de tip Zero Trust și modul în care sunt izolate mediile de testare (sandboxing) ale modelelor de frontieră.

AI Act și sistemele agentice

AI Act a fost conceput înainte de răspândirea largă a agenților IA autonomi, iar categoriile sale de risc se adaptează în continuare la această realitate tehnologică. Comisia Europeană și alte autorități de reglementare la nivel global monitorizează activ evoluția agenților IA și a modelelor multimodale, fiind de așteptat ca ghidurile de aplicare și standardele tehnice conexe să fie actualizate periodic.

Etica și reglementarea IA nu sunt constrângeri opuse inovării, ci condiții pentru adoptarea sustenabilă și de încredere a acestor tehnologii. Bias-ul algoritmic, transparența deciziilor și conformitatea cu cadre precum AI Act reprezintă competențe practice tot mai căutate pe piața muncii în domeniul IT, iar înțelegerea tendințelor emergente – agenții IA și modelele multimodale – este esențială pentru anticiparea următoarelor provocări etice și legislative.

Întrebări de autoevaluare

1. Ce este bias-ul algoritmic și prin ce se diferențiază de o simplă eroare statistică aleatorie?
2. Care sunt principalele surse de bias în pipeline-ul de Machine Learning și ce presupune fiecare dintre ele?
3. Care este diferența conceptuală dintre *Paritatea Demografică* (Demographic Parity) și *Șansele Egalizate* (Equalized Odds) ca metrici de măsurare a echității?
4. Cum se clasifică metodele de explicabilitate (XAI) în funcție de momentul aplicării (*intrinsec vs. post-hoc*) și de domeniul de aplicare (*global vs. local*)?
5. Prin ce se deosebește metoda LIME de metoda SHAP în explicarea deciziilor unui model de IA?

6. Ce drepturi oferă Articolul 22 din Regulamentul GDPR persoanelor afectate de decizii automatizate?
7. Care sunt categoriile de risc stabilite de Regulamentul UE privind IA (AI Act) și ce tip de obligații legale corespund fiecărei categorii?
8. Ce cerințe stricte trebuie să îndeplinească un sistem clasificat cu *risc ridicat* înainte de a fi introdus pe piața UE conform AI Act?
9. Cum diferă abordarea *Green AI* de *Red AI*?
10. Ce provocări etice, de securitate și de responsabilitate (*accountability*) specifice ridică apariția agenților IA autonomi în comparație cu modelele conversaționale clasice?

BIBLIOGRAFIE

1. Bratko, I. (2012) Prolog Programming for Artificial Intelligence, 4th ed. Addison-Wesley.
2. Clocksin, W.F., Mellish, C.S. (2003) Programming in Prolog, 5th ed. Springer.
3. Giarratano, J., Riley, G. (2005) Expert Systems: Principles and Programming, 4th ed. Course Technology.
4. Goodfellow, I. et al. (2014) „Generative Adversarial Networks", Advances in Neural Information Processing Systems (NeurIPS).
5. Goodfellow, I., Bengio, Y., Courville, A. (2016) Deep Learning. MIT Press.
6. Géron, A. (2022) Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow, 3rd ed. O'Reilly Media.
7. Hastie, T., Tibshirani, R., Friedman, J. (2009) The Elements of Statistical Learning, 2nd ed. Springer.
8. He, K. et al. (2016) „Deep Residual Learning for Image Recognition", Proceedings of CVPR.
9. Ho, J., Jain, A., Abbeel, P. (2020) „Denoising Diffusion Probabilistic Models", Advances in Neural Information Processing Systems (NeurIPS).
10. Hochreiter, S., Schmidhuber, J. (1997) „Long Short-Term Memory", Neural Computation, 9(8), pp. 1735–1780.
11. Jackson, P. (1998) Introduction to Expert Systems, 3rd ed. Addison-Wesley.
12. James, G., Witten, D., Hastie, T., Tibshirani, R. (2021) An Introduction to Statistical Learning. Springer.
13. Jurafsky, D., Martin, J.H. Speech and Language Processing, 3rd ed. (ediție în curs de actualizare). Stanford University.
14. Kaplan, J. et al. (2020) „Scaling Laws for Neural Language Models", arXiv:2001.08361.
15. LeCun, Y., Bengio, Y., Hinton, G. (2015) „Deep Learning", Nature, 521, pp. 436–444.
16. Luger, G. F. Artificial Intelligence: Structures and Strategies for Complex Problem Solving.
17. Lundberg, S., Lee, S.I. „A Unified Approach to Interpreting Model Predictions" (SHAP), Advances in Neural Information Processing Systems (NeurIPS).
18. Mamdani, E.H. and Assilian, S. (1975) 'An experiment in linguistic synthesis with a fuzzy logic controller', International Journal of Man-Machine Studies, 7(1), pp. 1–13.
19. McKinsey Global Institute. Notes from the AI Frontier: Modeling the Impact of AI on the World Economy.
20. Mikolov, T. et al. (2013) „Efficient Estimation of Word Representations in Vector Space", arXiv:1301.3781.
21. Mitchell, T. M. (1997) Machine Learning. McGraw-Hill.
22. Nilsson, N. J. Artificial Intelligence: A New Synthesis.
23. OECD. OECD Principles on Artificial Intelligence (OECD.AI).
24. Oprea, Serghei. Sisteme expert. Chișinău: UCCM (Editura Elena-VI), 2006.
25. Pearl, J. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.
26. Poole, D. L., Mackworth, A. K. (2023) Artificial Intelligence: Foundations of Computational Agents, 3rd ed. Cambridge University Press.
27. Power, D. J. (2002) Decision Support Systems: Concepts and Resources for Managers. Greenwood/Quorum Books.
28. Precup, R.-E. și Preitl, S. (2013) Introducere în ingineria reglării automate cu elemente de logică fuzzy. Timișoara: Editura Politehnica.
29. Regulamentul (UE) 2016/679 al Parlamentului European și al Consiliului (GDPR), art. 22 – procesul decizional individual automatizat.
30. Regulamentul (UE) 2024/1689 al Parlamentului European și al Consiliului privind inteligența artificială (AI Act).

31. Ribeiro, M.T. et al. „‘Why Should I Trust You?’ Explaining the Predictions of Any Classifier” (LIME), Proceedings of the ACM SIGKDD.
32. Ross, T.J. (2010) Fuzzy Logic with Engineering Applications. 3rd edn. Chichester: John Wiley & Sons.
33. Russell, S., Norvig, P. (2020) Artificial Intelligence: A Modern Approach, 4th ed. Pearson.
34. Shortliffe, E. H., Buchanan, B. G. „A Model of Inexact Reasoning in Medicine” (sistemul MYCIN).
35. Sterling, L., Shapiro, E. (1994) The Art of Prolog, 2nd ed. MIT Press.
36. SWI-Prolog — documentație oficială: https://www.swi-prolog.org/pldoc/
37. Turban, E., Sharda, R., Delen, D. (2011) Decision Support and Business Intelligence Systems, 9th ed. Pearson.
38. UNESCO. Recomandarea UNESCO privind etica inteligenței artificiale (2021).
39. Vaswani, A. et al. (2017) „Attention Is All You Need”, Advances in Neural Information Processing Systems (NeurIPS).
40. White, J. et al. (2023) „A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT”, arXiv.
41. Zadeh, L. A. (1965) „Fuzzy Sets”, Information and Control, 8(3), pp. 338–353.

În redacția autorului

Bun de tipar 02.11.2026
Coli editoriale 8,55. Coli de autor 8,25. Coli de tipar 19,75.
Comanda nr. 73.

Serviciul Editorial-Poligrafic
al Academiei de Studii Economice din Moldova
Chişinău, MD-2005, str.Bănulescu-Bodoni 59.
Tel.: 022-402-910