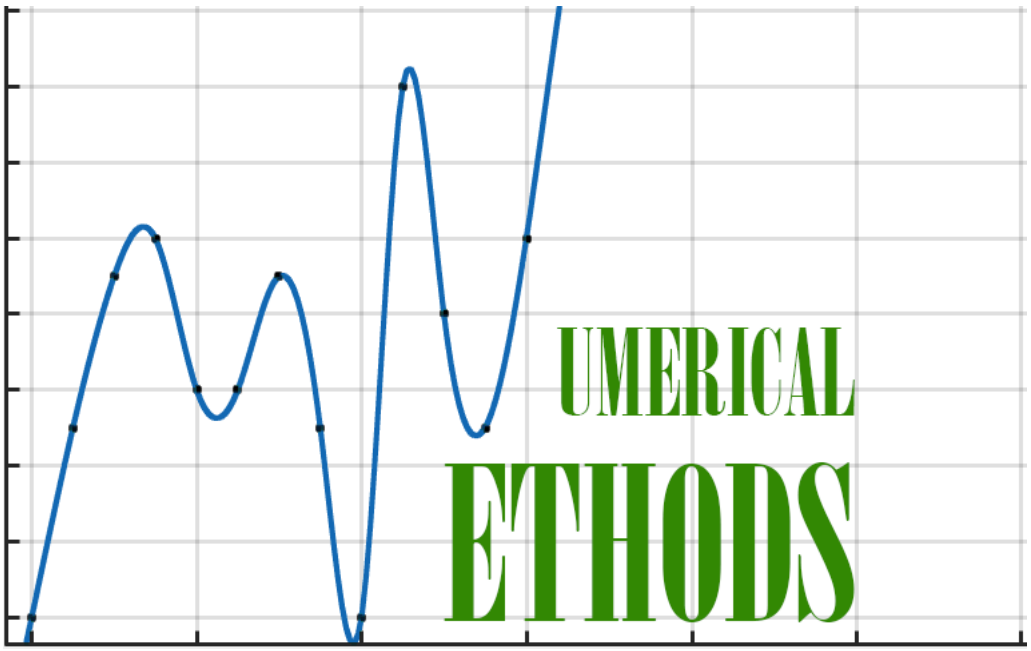
– Metode Numerice –

Tema 1: *Metode Numerice Matriceale*

Publicare: 03.04.2023 · **Deadline soft:** 30.04.2023 23:55 · **Deadline hard:** 04.05.2023 23:55



Responsabili: Alexandru Buzea, Cătălin Rîpanu

Editare, checker și testare: Valentin Vintilă

Corectare: Alexandru Buzea, Alexandru Mihai, Cătălin Rîpanu, Matei Ceaușu, Taisia Coconu, Valentin Vintilă, Vlad Negoiță

Contents

Obiectivele temei de casă

Prima temă de casă la Metode Numerice vizează următoarele obiective:

- Familiarizarea cu mediul de programare **GNU Octave** și facilitățile oferite de acesta;
- Folosirea matricelor și a sistemelor de ecuații liniare pentru a modela probleme reale, întâlnite în viața de zi cu zi, precum lanțurile Markov;
- Introducerea în învățarea supervizată[1].

Changelog

- **03.04.2023:** S-a publicat tema 1, *momentan fără checker*.
- **04.04.2023:** S-au adăugat mici clarificări pentru funcțiile și prezentarea task-ului 2.
- **06.04.2023:** A fost modificată precizarea din enunț cu privire la valorile pe care le poate lua **start_position** (task 1, heuristic greedy).
- **07.04.2023:** S-a adăugat în mod explicit o funcție nouă, ajutătoare, pentru a parsa fișierul .csv și testa funcția `gradient descent` (task 2).
- **08.04.2023:** A fost modificat sistemul de ecuații ce precede ecuația (2) (1.3.3) pentru a corespunde cu forma matriceală.
- **17.04.2023:** S-a adăugat un exemplu de (fișier .csv)

[1] en. *Supervised ML*

1 Markov is coming ... *(40p)*

1.1 Enunț

După ce a obținut note foarte bune la materiile de pe semestrul întâi (în mod special la programare și la algebră liniară), Mihai s-a decis să își ocupe timpul cu o problemă interesantă, pentru care își propune să găsească o soluție cât mai performantă.

Fiind pasionat de tehnologie, el și-a cumpărat un robotel pe care îl poate programa după cum dorește. În timpul liber, a mai construit și un mic labirint pe care intenționa să testeze roboțelul.

Acum, Mihai dorește să plaseze roboțelul undeva în labirint și să-l programeze astfel încât să aleagă, la fiecare pas, cea mai bună direcție pentru a reuși să evadeze. Roboțelul se consideră evadat dacă găsește una din ieșirile consancrate ale labirintului.

Pentru că Mihai nu este încă familiarizat cu algoritmii avansați de căutare (căci aceștia se învață abia în anul 2), își propune să plece de la o problemă mai simplă, iar apoi să implementeze un algoritm simplu, astfel: Având la dispoziție un labirint și o poziție de plecare, care este probabilitatea ca robotul meu **să ajungă într-o zonă de câștig**, dacă la fiecare pas el **alege o direcție aleatoare de deplasare** dintre cele disponibile? De asemenea, cum aș putea folosi probabilitățile determinate anterior pentru determinarea unei căi pentru robot prin labirintul meu, într-o manieră **mai eficientă** decât o căutare exhaustivă ?

1.2 O explicație vizuală

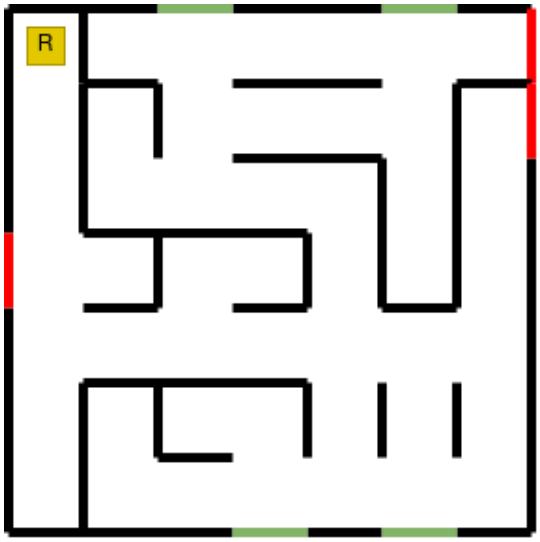


Figura 1: Exemplu de labirint

Să presupunem labirintul de mai sus, foarte simplu reprezentat sub forma unei matrice pătratice, în care poziția de plecare a roboțelului nostru este colțul din stânga–sus. Considerăm că punctul de pornire are coordonatele $(1, 1)$. Robotul va alege la fiecare pas să se mute într-o nouă celulă a labirintului pe care nu a vizitat-o anterior, putând să se deplaseze câte o pătrățică în sus, în jos, la stânga sau la dreapta, însă nu are voie să meargă pe diagonală.

Spre exemplu, din poziția de start, acesta va vizita celula de coordonate $(2, 1)$, apoi celulele de coordonate

$(3, 1)$ și $(4, 1)$, iar apoi va alege între cele 2 celule adiacente pe cea care are probabilitatea mai mare de a ajunge la o ieșire câștigătoare din labirintul nostru. Ieșirile în cadrul problemei studiate sunt de 2 tipuri:

- **Ieșiri care duc la câștig.** Ele sunt ieșirile marcate cu verde pe figura de mai sus, și se suprapun tot timpul cu limita superioară, respectiv cu cea inferioară ale labirintului. În momentul în care robotul alege să iasă din labirint pe una dintre aceste ieșiri, se poate spune că acesta a câștigat (probabilitatea de câștig este 1).
- **Ieșiri care duc la pierderea jocului.** Ele sunt ieșirile marcate cu roșu pe figura de mai sus, și se suprapun tot timpul cu limitele laterale (stânga / dreapta) ale labirintului. În momentul în care robotul alege să iasă din labirint prin una dintre aceste ieșiri, se poate spune că acesta a pierdut jocul (probabilitatea de câștig în această stare este 0).

În interiorul labirintului, pot exista și pereți care să nu permită trecerea roboțelului între 2 celule adiacente din punct de vedere spațial. Spre exemplu, în figura de mai sus, nu se poate merge din celula $(1, 1)$ direct în celula $(1, 2)$. Astfel, în orice moment de timp, robotul va avea un număr de cel mult 4 alegeri corespunzătoare direcțiilor în care se poate deplasa, din care acesta o poate alege complet aleator (cu probabilitate egală) pe oricare dintre ele.

1.3 Referințe teoretice

Pentru modelarea situației prezentate vom folosi lanțuri de probabilități, cunoscute sub denumirea de **lanțuri Markov**. Lanțurile Markov sunt deosebit de utile în teoria probabilităților, având aplicații în domenii precum economie, fiabilitatea sistemelor dinamice, respectiv în algoritmii de inteligență artificială. **Google Page Rank** este, de asemenea, o formă modificată a unui lanț Markov.

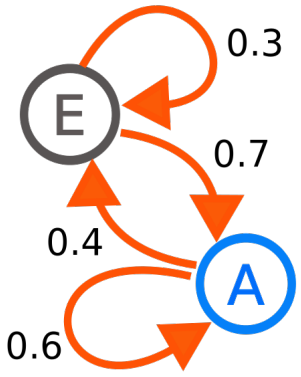


Figura 2: Ilustrație simplă a unui lanț Markov

Structura de date de bază în cadrul lanțurilor Markov este graful orientat, în care nodurile reprezintă stările, iar fiecare muchie existentă între două stări reprezintă o probabilitate nenulă de trecere de la o stare inițială la o stare finală. Evident, pentru orice stare, suma tuturor probabilităților de tranziție este egală cu unitatea. Matematic, acest lucru se poate reprezenta astfel:

$$\sum_{j=1}^{n} p_{ij} = 1, \forall i \in \overline{1, n} \tag{1}$$

Putem astfel să aplicăm o idee asemănătoare și în problema noastră. Vom asocia fiecărei celule a labirintului câte o stare și vom numerota stările, începând cu colțul din stânga-sus, ca în figura de mai jos:

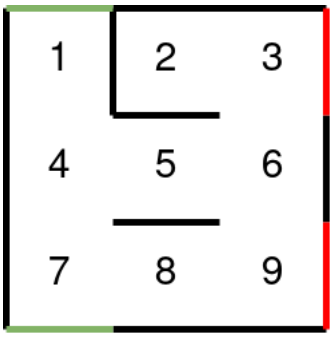


Figura 3: Exemplu de numerotare a unui labirint de dimensiune 3×3

În afară de stările corespunzătoare amplasării robotului într-una dintre celulele labirintului, vom mai avea nevoie de două stări suplimentare:

- **Starea WIN.** Aceasata va fi starea în care putem considera că am câștigat, din care nu mai putem ieși ulterior;
- **Starea LOSE.** Aceasta va reprezenta starea în care putem considera că am pierdut.

Odată adăugate și acestea în graful nostru orienat, discutăm de următorul lanț Markov:

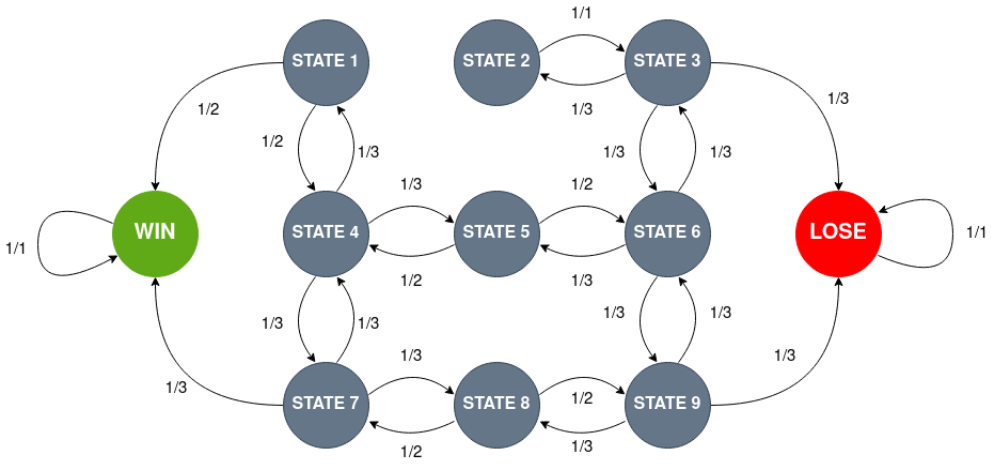


Figura 4: Lanțul Markov asociat exemplului de labirint din fig. 5

Pentru acest lanț Markov (care, din punct de vedere abstract, reprezintă un graf orientat în care fiecare muchie are o anumită greutate, egală cu probabilitatea descrisă mai sus), avem nevoie de o caracterizare concretă (cu alte cuvinte, de un mod de stocare) care să ne ajute în reținerea efectivă a lanțului. În continuare, vom prezenta câteva caracterizări posibile.

1.3.1 Matricea de adiacență

Matricea de adiacență a unui graf orientat, asemănătoare cu conceptul de matrice de adiacență a unui graf neorientat, se poate defini prin următoarea relație:

$$A = (A_{ij})_{i,j\in\overline{1,n}} \in \{0,1\}^{n\times n}, \text{ unde } A_{ij} = \begin{cases} 1, & \text{dacă există o tranziție din starea } i \text{ în starea } j \\ 0, & \text{altfel} \end{cases}$$

În situația grafului din fig. 4, reprezentat prin 11 stări (9 stări corespunzătoare celulelor labirintului, respectiv 2 stări suplimentare, WIN și LOSE, **în această ordine**), matricea de adiacență A a grafului este din $\{0,1\}^{11\times 11}$ și are următoarea formă:

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Se observă că sectorul $A(1:9, 1:9)$[2] este simetric, întrucât pereții **nu** sunt unidirecționali (dacă tranziția de la starea i la starea j este posibilă, atunci și tranziția inversă este posibilă).

1.3.2 Matricea legăturilor

Matricea legăturilor reprezintă o formă mai potentă a matricei de adiacență, fiind foarte asemănătoare cu aceasta – singura diferență este dată de semnificația elementelor ce o populează. În cazul matricei de legături, elementele sunt chiar probabilitățile de tranziție de la o stare la alta în lanțul Markov. Folosind notația de probabilitate p_{ij} introdusă anterior (ec. 1), ea se definește astfel:

$$L = (p_{ij})_{i,j\in\overline{1,n}} \in [0,1]^{n\times n} \Leftrightarrow L_{ij} = \begin{cases} p_{ij}, & 0 < p_{ij} \leq 1 \\ 0, & \text{altfel} \end{cases}$$

Pentru exemplul de lanț din fig. 4, matricea legăturilor este următoarea:

$$L = \begin{bmatrix} 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 & 0 & 0 & 0 & 1/3 \\ 1/3 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 & 0 \\ 0 & 0 & 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

[2] Am utilizat notația standard din Octave, adică $A(1:9, 1:9)$ se referă la submatricea formată din intersecția primelor 9 linii cu primele 9 coloane.

Observăm că matricea este o matrice stochastică pe linii[3]., ceea ce îi aduce o mulțime de proprietăți interesante, discutate deja în cadrul cursului de Metode Numerice.

1.3.3 Sistem de ecuații liniare

Pe lângă abordările anterioare, putem ține minte lanțul Markov și ca pe o formulare ce se pretează pe un alt stil de problemă: rezolvarea unui sistem de ecuații liniare.

Mai exact, să considera un vector $\boldsymbol{p} \in \mathbb{R}^{mn}$ reprezentând probabilitățile de câștig pentru fiecare celulă din labirint, unde $m \in \mathbb{N}^*$ și $n \in \mathbb{N}^*$ sunt dimensiunile labirintului; spre exemplu, în cazul fig. 5, $\boldsymbol{p} \in \mathbb{R}^9$. Popularea acestui vector se face respectând numerotarea stărilor descrisă anterior.

Să experimentăm puțin cu cazul în care robotelul nostru se află în starea 1. Atunci, el poate efectua următoarele tranziții / deplasări valide prin labirint:

- Poate trece în **starea 4**, cu probabilitatea ½;
- Poate trece în **starea WIN** și să câștige, tot cu probabilitatea ½.

Astfel, starea 1 va fi caracterizată prin următoarea ecuație:

$$\begin{aligned} p_1 &= \frac{1}{2} \cdot p_4 + \frac{1}{2} \cdot p_{\text{WIN}}, \text{ dar } p_{\text{WIN}} = 1 \\ \Rightarrow p_1 &= \frac{1}{2} \cdot p_4 + \frac{1}{2} \end{aligned}$$

În mod similar, se pot scrie ecuații pentru toate stările:

$$\begin{cases} p_1 &= \frac{1}{2} \cdot p_4 + \frac{1}{2} \\ p_2 &= p_3 \\ p_3 &= \frac{1}{3} \cdot p_2 + \frac{1}{3} \cdot p_6 \\ p_4 &= \frac{1}{3} \cdot p_1 + \frac{1}{3} \cdot p_5 + \frac{1}{3} \cdot p_7 \\ p_5 &= \frac{1}{2} \cdot p_4 + \frac{1}{2} \cdot p_6 \\ p_6 &= \frac{1}{3} \cdot p_3 + \frac{1}{3} \cdot p_5 + \frac{1}{3} \cdot p_9 \\ p_7 &= \frac{1}{3} \cdot p_4 + \frac{1}{3} \cdot p_8 + \frac{1}{3} \\ p_8 &= \frac{1}{2} \cdot p_7 + \frac{1}{2} \cdot p_9 \\ p_9 &= \frac{1}{3} \cdot p_6 + \frac{1}{3} \cdot p_8 \end{cases}$$

După cum ne-am obișnuit, putem trece acest sistem în forma sa matriceală:

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \end{bmatrix} + \begin{bmatrix} 1/2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1/3 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

[3]Suma elementelor de pe fiecare linie este egală cu 1

În sistemul anterior, vectorul termenilor liberi (evidențiat cu albastru) provine din acele elemente care inițial erau combinații liniare de p_{WIN}.

Am făcut toată această prelucrare în ec. 2 pentru a putea scrie următorul produs:

$$\boldsymbol{p} = G\boldsymbol{p} + \boldsymbol{c}$$

Această formă se pretează perfect metodei Jacobi de rezolvare în care identificăm G și $\boldsymbol{c}$ drept matricea, respectiv vectorul de iterație. De aceea, vom opta pentru această metodă iterativă pentru a soluționa sistemul. În cazul nostru particular (fig. 5), raza spectrală a matricei G are valoarea $\rho(G) \approx 0.85192$, ceea ce înseamnă că Jacobi va converge.

Pentru cei curioși, soluția este:

$$\begin{cases} p_1 \approx 0.84615 \\ p_2 \approx 0.15385 \\ p_3 \approx 0.15385 \\ p_4 \approx 0.69231 \\ p_5 \approx 0.50000 \\ p_6 \approx 0.30769 \\ p_7 \approx 0.73077 \\ p_8 \approx 0.50000 \\ p_9 \approx 0.26923 \end{cases}$$

1.3.4 Algoritm euristic de căutare

Rezultatele de mai sus reflectă o intuiție evidentă: stările care sunt mai *„apropiate”* de **starea WIN** au o probabilitate mai mare de câștig, iar cele care sunt apropiate de **starea LOSE** au o probabilitate mai mică. Acesta este motivul pentru care am putea gândi un algoritm de căutare **euristic** cu ajutorul căruia robotul ar putea ajunge din poziția inițială la una din stările câștigătoare.

Un algoritm de căutare euristic este un algoritm care nu ne furnizează o soluție optimă (în cazul nostru, un drum minim) pentru toate cazurile posibile, însă are avantajul de a fi foarte rapid în comparație cu algoritmii de căutare exhaustivi (clasici).

Apelăm la un algoritm greedy simplu, bazat de DFS, care are pseudocodul de pe următoarea pagină.

În pseudocodul de mai jos, parametrii funcției de căutare sunt următorii:

- ***start_position***, reprezentând poziția de start a robotului în codificarea utilizată până la acest moment (un indice de la 1 la $n \cdot m$ inclusiv, unde n și m sunt dimensiunile labirintului);
- ***probabilities***, prin care se înțelege vectorul probabilităților fiecărei stări în parte, de lungime $n \cdot m + 2$ (acesta este de fapt vectorul extins al probabilităților, spre deosebire de cel calculat anterior folosind metoda iterativă – conține și probabilitățile asociate stărilor WIN, respectiv LOSE);
- ***adjacency_matrix***, matricea de adiacență a labirintului propus, în maniera în care aceasta a fost descrisă mai devreme.

Algoritmul întoarce un vector de indecși reprezentativi celuleleor / stărilor labirintului, urmând ca aceștia să fie traduși ulterior în perechi linie–coloană pentru a putea fi interpretați mai ușor.

Algoritm euristic

```
1:  procedure HEURISTIC_GREEDY(start_position, probabilities, adjacency_matrix)
2:      path ← [start_position]
3:      visited[start_position] ← True
4:      while path is not empty do
5:          position ← top() / last element of the path vector
6:          if position is the WIN state then
7:              return path
8:          if position has no unvisited neighbours then
9:              erase position from the end of the path
10:         neigh ← the unvisited neighbour (with greatest probability to reach WIN) of the current position
11:         visited[neigh] ← True
12:         path ← [path, neigh]
13:     return path (since there is no path to the WIN state)
```

În figura de mai jos, am evidențiat drumul pe care îl va alege robotul, luând în considerare probabilitățile calculate anterior (vom porni în acest exemplu din starea / celula 2).

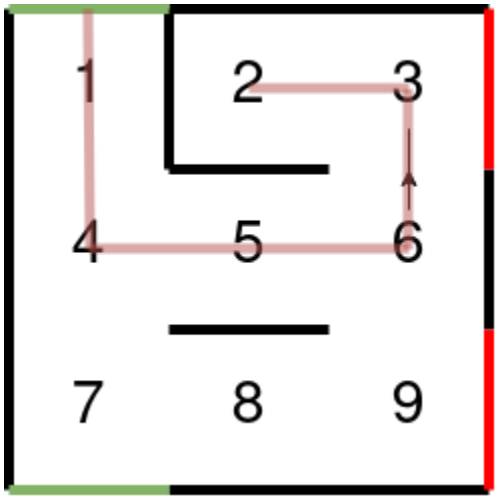


Figura 5: Exemplu de drum obținut folosind algoritmul euristic

1.3.5 Codificarea labirintului

Pentru a prelucra labirintul sub forma unor date de intrare, este necesară o reprezentare a labirintului într-o formă condensată. Astfel, Mihai se inspiră dintr-un algoritm pe care l-a găsit într-un alt context, cel al graficii pe calculator, numit algoritmul Cohen-Sutherland.

Ideea preluată din algoritmul original este de a codifica binar zidurile ce separă celule adiacente spațial: labirintul nostru poate fi stocat drept o matrice cu $m \times n$ intrări, numere întregi reprezentate pe 4 biți de forma $\overline{b_3 b_2 b_1 b_0}_{(2)}$, unde fiecare bit activ (setat pe 1) reprezintă o posibilă direcție de deplasare obturată de un perete al labirintului. În cazul nostru, ne însușim următoarea codificare:

- Bitul b_3 setat pe 1 indică un **zid la nordul celulei**;
- Bitul b_2 setat pe 1 indică un **zid la sudul celulei**;

- Bitul b_1 setat pe 1 indică un **zid la estul celulei**;
- Bitul b_0 setat pe 1 indică un **zid la vestul celulei**;

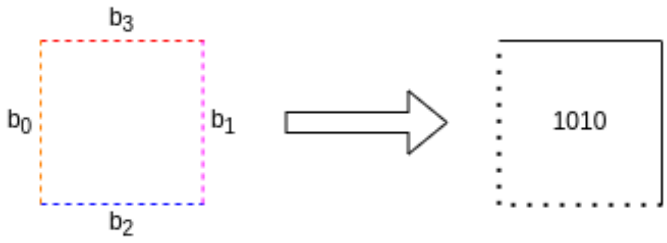


Figura 6: Reprezentarea direcțiilor în algoritmul Cohen-Sutherland, alături de un exemplu

În situația labirintul nostru, codificarea ar fi:

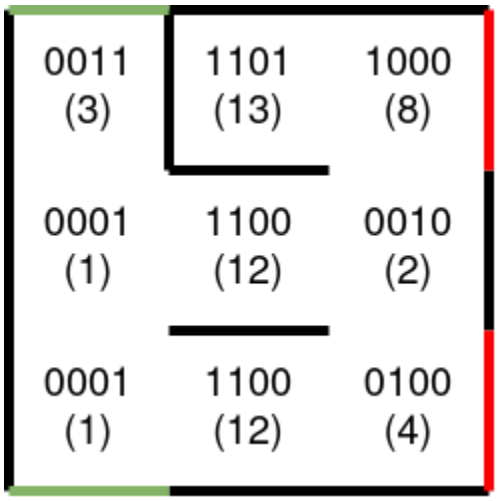


Figura 7: Codificarea pereților labirintului pentru exemplul dat

Este foarte important să observați faptul că pereții sunt bidirecționali (anume că, deși o codificare aleatoare ar permite tranziții unidirecționale între stări, noi vom trata exclusiv cazul pereților care blochează tranzițiile în labirint în ambele sensuri între oricare stări adiacente).

1.4 Cerințe

În urma parcurgerii materialului teoretic furnizat anterior, sunteți pregătiți să implementați următoarele funcții în Matlab:

- `function [Labyrinth] = parse_labyrinth(file_path)`

  Funcția `parse_labyrinth` va primi o cale relativă către un fișier text unde se află reprezentarea codificată a labirintului, așa cum a fost descrisă în secțiunea de teorie dedicată.

  Formatul fișierului de intrare va fi următorul:

```
m n
l_11 l_12 l_13 ... l_1n
l_21 l_22 l_23 ... l_2n
l_31 l_32 l_33 ... l_3n
...
l_m1 l_m2 l_m3 ... l_mn
```

- `function [Adj] = get_adjacency_matrix(Labyrinth)`

  Funcția `get_adjacency_matrix` va primi matricea codificărilor rezultată după pasul anterior și va întoarce matricea de adiacență a grafului / lanțului Markov.

- `function [Link] = get_link_matrix(Labyrinth)`

  Funcția `get_link_matrix` va primi matricea codificărilor unui labirint valid și va returna matricea legăturilor asociată labirintului dat.

- `function [G, c] = get_Jacobi_parameters(Link)`

  Funcția `get_Jacobi_parameters` va primi matricea legăturilor obținută anterior și va returna matricea de iterație și vectorul de iterație pentru metoda Jacobi.

- `function [x, err, steps] = perform_iterative(G, c, x0, tol, max_steps)`

  Funcția `perform_iterative` va primi matricea și vectorul de iterație, o aproximație inițială pentru soluția sistemului, o toleranță (eroare relativă maxim acceptabilă pentru soluția aproximativă a sistemului, între doi pași consecutivi) și un număr maxim de pași pentru execuția algoritmului.

- `function [path] = heuristic_greedy(start_position, probabilities, Adj)`

  Funcția `heuristic_greedy` va primi o poziție de start (un index al unei celule / stări din intervalul $\overline{1, mn}$), vectorul extins al probabilităților (incluzând cele două probabilități pentru stările WIN și LOSE) și matricea de adiacență a lanțului Markov.

  Va returna apoi o cale validă către starea de WIN. Se garantează că labirintul (și, implicit, graful asociat) este conex, și deci va exista întotdeauna o cale de câștig validă.

- `function [decoded_path] = decode_path(path, lines, cols)`

  Funcția `decode_path` va primi o cale validă (sub forma unui vector coloană) și dimesniunile labirintului și va returna un vector de perechi (matrice cu două coloane), fiecare pereche reprezentând linia și coloana celulei cu codificarea dată.

1.4.1 Restricții și precizări

Înainte să vă apucați de lucru, ar fi bine să luați aminte că:

- Labirintul **NU** este neapărăt pătratic (numărul de coloane nu trebuie să coincidă cu numărul de linii);
- Se garantează faptul că labirintul este conex și că există mereu câte o cale către ieșirea / starea de WIN și către cea de LOSE;

- Observați că matricele de adiacență și de legătură sunt matrice mari, dar rare. Este **OBLIGATORIE** reținerea acestor matrice și a matricelor derivate (precum matricea de iterație sau vectorul de iterație) sub forma unor **matrice rare**. Octave vă oferă posibilitatea stocării matricelor rare într-o manieră mult mai eficientă decât cea convențională, anume prin stocarea elementelor nenule și a pozițiilor acestora. De asemenea, există funcții specializate pentru lucrul cu matrice rare, pe care vă încurajăm să le descoperiți aici.

Pentru restricțiile general valabile, verificați sfârșitul acestui document.

2 Linear Regression *(40p)*

2.1 Enunț

Având o pasiune profundă și pentru **Metode Numerice**, Mihai este interesat atât de Învățarea Automată[4], cât și de Inteligența Artificială[5], și i-ar plăcea să le exploreze mai în detaliu (atât cât poate). Curios din fire, acesta începe să citească despre cum poate să proiecteze un model de învățare automată care să se antreneze pe baza unui set de date existent ce are o anumită dimensiune.

Cu ajutorul unui algoritm de Învățare Automată Supervizată[6], numit în literatura de specialitate **Linear Regression**, Mihai dorește să înțeleagă mai multe despre manipularea **predicțiilor** și a **erorilor**[7] ce pot să apară în prelucrarea computațională.

În esență, **Linear Regression** poate fi interpretat geometric drept o dreaptă (la ALGAED ați întâlnit noțiunea de *dreaptă de regresie*) care **minimizează** radicalul sumei pătratelor distanțelor punctelor (datelor) ce fac parte dintr-o mulțime de interes[8].

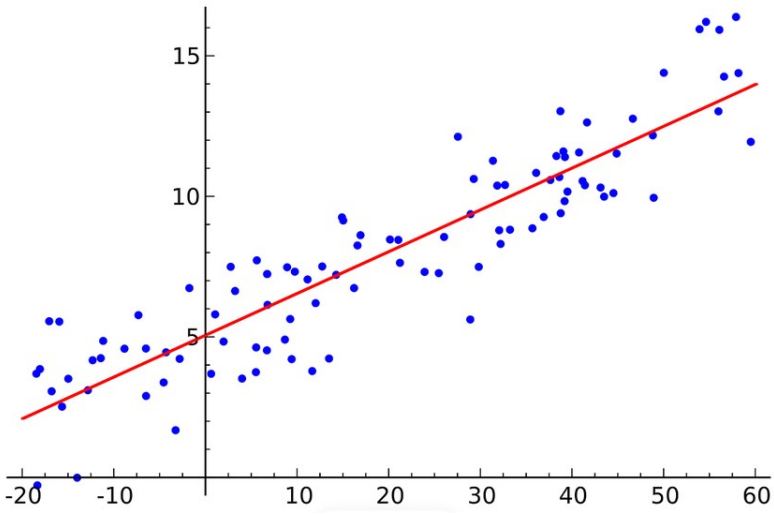


Figura 8: Reprezentarea grafică a unei drepte de regresie ce trece printr-un set de date.

Din punct de vedere funcțional, **Linear Regression** se ocupă de **micșorarea**, până într-o anumită limită, a **funcției de cost** și a **pierderii** (aceste concepte vor fi detaliate în paragrafele ce urmează). Evident, există mai multe tipuri de **Linear Regression** (precum regresia simplă, regresia multiplă și cea logistică).

În urma cercetărilor sale, Mihai se hotărăște să folosească **Multiple Linear Regression** pentru a putea face **predicții** cu privire la prețul apartamentelor din zona sa, întrucât nu mai dorește să locuiască cu ai lui, vrând să își manifeste independența față de ei.

O astfel de predicție poate fi scrisă sub forma unei funcții $h_{\boldsymbol{\theta}} : \mathbb{R}^n \to \mathbb{R}$, cu $\boldsymbol{\theta} \in \mathbb{R}^{n+1}$, funcție ce se poate

[4] en. *Machine Learning*
[5] en. *Artificial Intelligence*
[6] en. *Supervised Machine Learning*
[7] Conceptele de *bias* și *variance*
[8] Acest fenomen este cunoscut și drept *aproximare în sensul celor mai mici pătrate* și va fi studiat în cadrul metodelor numerice funcționale (a doua parte a materiei).

defini după cum urmează:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n + \varepsilon$$

În scrierea anterioară, am folosit următoarele notații:

- $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ reprezintă valoarea **prezisă** pentru funcționalitățile $(x_1, x_2, \ldots, x_n)$ (acestea se mai numesc și **predictori** sau **features**);
- $\theta_1, \ldots, \theta_n \in \mathbb{R}$ reprezintă coeficienții specifici modelului de învățare automată (aceștia mai poartă denumirea de **weights**);
- $\theta_0 \in \mathbb{R}$ reprezintă valoarea lui $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ atunci când toți predictorii sunt 0, adică $\boldsymbol{x} = \mathbf{0}$ (în literatură poartă numele de **intercept**);
- $\varepsilon \in \mathbb{R}$ este eroarea (diferența **în modul**) dintre valoarea prezisă și cea actuală a lui $h_{\boldsymbol{\theta}}(\boldsymbol{x})$.

Ei bine, acești coeficienți $\theta_0, \theta_1, \ldots, \theta_n$ ce formează $\boldsymbol{\theta}$ descriu cât de capabil este un model de învățare automată pentru a face predicții cât mai bune (apropiate de realitate) după primirea de date noi, ce nu au mai fost *văzute* de către acesta. Putem așadar să definim **funcția de cost**, o funcție ce returnează eroarea dintre valoarea actuală și cea prezisă, și să încercăm să o **minimizăm**.

Funcția de cost $J : \mathbb{R}^{n+1} \to \mathbb{R}$ va avea următoarea scriere:

$$J(\boldsymbol{\theta}) = J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} \left[h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) - y^{(i)} \right]^2$$

Am utilizat următoarele notații în scrierea de mai sus:

- $\theta_1, \ldots, \theta_n \in \mathbb{R}$ reprezintă coeficienții specifici modelului, la fel ca mai sus;
- $m \in \mathbb{N}^*$ reprezintă numărul de antrenamente[9];
- $\boldsymbol{x}^{(i)}$ reprezintă intrările pentru antrenamentele de ordin $i \in \mathbb{N}$[10], ceea ce înseamnă că $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$ este ipoteza (valoarea prezisă) pentru antrenamentul cu indexul i;
- $y^{(i)}$ reprezintă ieșirile pentru antrenamentele de ordin $i \in \mathbb{N}$.

NU confundați notația $\gamma^{(i)}$ cu ridicarea la putere sau cu derivarea! Facem referire strict la indexul (numărul) iterației curente.

2.1.1 Algoritmi de optimizare

Pe parcursul studiului său, Mihai a mai descoperit și faptul că există anumiți algoritmi de optimizare pentru a determina coeficienții modelului, și anume **metoda gradientului descendent**[11], respectiv **Normal Equation**.

Metoda gradientului descendent reprezintă o modalitate generală pentru optimizarea funcțiilor convexe (în cazul nostru, o vom aplica funcției de cost), ce poate determina minimul local al funcției de interes. Metoda utilizează o tehnică iterativă.

Având în vedere că funcția de cost $J(\boldsymbol{\theta})$ are un **minim global unic**, putem spune că orice minim local este, de asemenea, un minim global; cu alte cuvinte, funcția de cost este **convexă**, iar acest lucru ne garantează faptul că orice metodă de optimizare va converge către minimul global al funcției de cost.

Această metodă își efectuează pașii în funcție de gradientul funcției de cost și de valoarea aleasă pentru rata de învățare, notată la noi cu $\alpha \in \mathbb{R}$.

[9] en. *training samples*
[10] en. *i^{th} training example*
[11] en. *Gradient Descent*

Amintim că prin gradientul funcției de cost înțelegem vectorul fomat din derivatele parțiale în raport cu $\theta_1, \ldots, \theta_n$. Cu alte cuvinte:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial \theta_1}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial J}{\partial \theta_n}(\boldsymbol{\theta}) \end{bmatrix}, \text{ unde } \frac{\partial J}{\partial \theta_j}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left[h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) - y^{(i)} \right] \cdot \boldsymbol{x}_j^{(i)}, \forall j \in \overline{1, n}$$

Transformarea pe care această metodă o propune este dată de relația:

$$\theta_j := \theta_j - \alpha \cdot \frac{\partial J}{\partial \theta_j}(\boldsymbol{\theta}), \forall j \in \overline{1, n}$$

Normal Equation reprezintă o metodă care implică o ecuație directă pentru a determina coeficienții $\theta_1, \ldots, \theta_n \in \mathbb{R}$ specifici modelului de interes. Această tehnică este utilă în situația în care lucrăm cu seturi restrânse (mici) de date. Se cristalizează următoarea ecuație:

$$\boldsymbol{\theta} = (X^T X)^{-1} X^T Y$$

unde:

- $X \in \mathbb{R}^{m \times n}$ reprezintă matricea ce stochează **m** vectori linie $\boldsymbol{x}^{(i)}$, $i \in 1, 2, \ldots, m$, fiecare vector linie având **n** valori specifici predictorilor.
- $Y \in \mathbb{R}^{m \times 1}$ reprezintă vectorul **coloană** ce reține **m** valori **actuale**.
- $\boldsymbol{\theta} \in \mathbb{R}^{n \times 1}$ reprezintă vectorul **coloană** ce reține **n** coeficienți $\theta_1, \ldots, \theta_n \in \mathbb{R}$ specifici modelului de învățare automată.

O problemă vizibilă cu acest algoritm este că determinarea inversei unei matrice implică un cost computațional ridicat pentru seturi mari de date. Pentru a mitiga această dificultate, vom folosi o altă metodă *(prezentată la curs)* pentru a rezolva sistemul, anume **metoda gradientului conjugat**[12].

Reamintim algoritmul în cauză:

Conjugate Gradient Method

procedure CONJUGATE_GRADIENT(A, b, x_0, tol, max_iter)
  $r^{(0)} \leftarrow b - Ax^{(0)}$
  $v^{(1)} \leftarrow r^{(0)}$
  $x \leftarrow x_0$
  $tol_{squared} \leftarrow tol^2$
  $k \leftarrow 1$
  while $k < max_{iter}$ and ${r^{(\text{k-1})}}^T r^{(\text{k-1})} > tol_{squared}$ **do**
    $t_k \leftarrow \frac{{r^{(\text{k-1})}}^T r^{(\text{k - 1})}}{{v^{\text{k}}}^T A v^{\text{k}}}$
    $x^{(\text{k})} \leftarrow x^{(\text{k-1})} + t_k v^{(\text{k})}$
    $r^{(\text{k})} \leftarrow r^{(\text{k-1})} - t_k A v^{(\text{k})}$
    $s_k \leftarrow \frac{{r^{(\text{k})}}^T r^{(\text{k})}}{{r^{(\text{k-1})}}^T r^{(\text{k-1})}}$
    $v^{(\text{k+1})} \leftarrow r^{(\text{k})} + s_k v^{(\text{k})}$
    $k \leftarrow k + 1$
  return x

[12] **NU** uitați faptul că această metodă necesită ca matricea sistemului să fie **pozitiv definită**.

2.1.2 Regularizare

În domeniul Învățării Automate, **regularizarea** reprezintă o metodă ce poate fi aplicată unui model de învățare automată astfel încât acesta să devină mult mai **general**, adică să aibă eroarea de **variață** cât mai mică după introducerea de **noi date** în urma antrenamentului său.

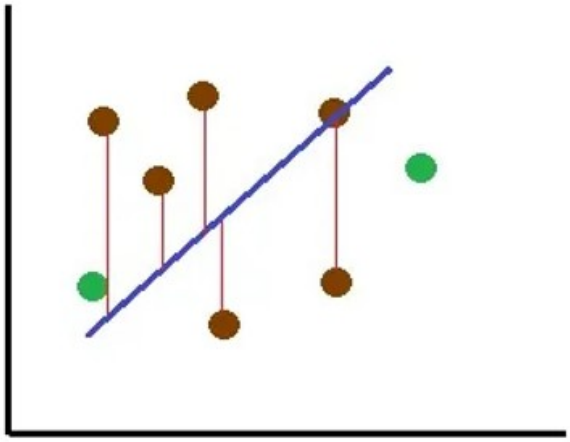


Figura 9: Reprezentarea grafică a unei drepte de regresie ce trece printr-un set de date de antrenament (punctele verzi) și printr-un set de date de testare (punctele maro). Se poate observa eroarea de variață (radicalul sumei pătratelor distanțelor punctelor maro)

Având în vedere cele menționate, Mihai este interesat în două tehnici de regularizare, **Regularizarea L1**, respectiv **Regularizarea L2**.

Regularizarea L2, denumită și **Ridge Regression**, se referă la a găsi o dreaptă de regresie care să treacă optim prin punctele care definesc setul de date de testare, introducând, însă, o mică eroare de bias. Cu alte cuvinte, dreapta găsită nu va minimiza pe deplin radicalul sumei pătratelor distanțelor punctelor din setul de date de antrenament.

În esență, această metodă se axează pe micșorarea coeficienților $\theta_0, \theta_1, \ldots, \theta_n \in \mathbb{R}$ astfel încât aceștia să fie apropiați de 0, efectul fiind **slăbirea** dependenței dintre $y^{(i)}$ și anumiți $x_1, x_2, \ldots, x_n$ din $\boldsymbol{x}^{(i)}$, adică ieșirea de ordin $i \in \mathbb{N}$ **va depinde mai puțin de predictori**.

Funcția regularizată de cost $J_{\boldsymbol{L2}} : \mathbb{R}^{n+1} \to \mathbb{R}$ va avea următoarea scriere:

$$J_{\boldsymbol{L2}}(\boldsymbol{\theta}) = J_{\boldsymbol{L2}}(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} \left[h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) - y^{(i)} \right]^2 + \lambda \sum_{j=1}^{n} \theta_j^2$$

Unde:

- $\lambda \sum_{j=1}^{n} \theta_j^2$ reprezintă termenul specific regularizării L2;
- $\lambda \in \mathbb{R}_+$ este parametrul care controlează **puterea regularizării**, acesta se poate determina folosind tehnica cross-validation, însă noi îl vom oferi la partea de implementare.

Regularizarea L1, denumită și **Lasso Regression**, este similară cu regularizarea L2, cu excepția faptului că anumiți $\theta_0, \theta_1, \ldots, \theta_n \in \mathbb{R}$ pot fi chiar 0, adică **se poate elimina definitiv** dependența ieșirii de ordin $i \in \mathbb{N}$ de anumiți **predictori**. Scopul rămâne același, și anume micșorarea complexității modelului de învățare automată.

Funcția regularizată de cost $J_{\boldsymbol{L1}} : \mathbb{R}^{n+1} \to \mathbb{R}$ va adopta următoarea scriere:

$$J_{\boldsymbol{L1}}(\boldsymbol{\theta}) = J_{\boldsymbol{L1}}(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} - h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) \right]^2 + \lambda \left\| \boldsymbol{\theta} \right\|_1$$

Unde:

- $\left\| \boldsymbol{\theta} \right\|_1$ reprezintă norma L1 a coeficienților modelului, adică $\left\| \boldsymbol{\theta} \right\|_1 = |\theta_0| + |\theta_1| + \cdots + |\theta_n|$.
- $\lambda \in \mathbb{R}_+$ este **parametrul** care controlează regularizarea.

2.1.3 Format CSV

Pentru realizarea funcției care implică metoda **gradientului descendent**, veți avea la dispoziție setul de date de antrenare în format CSV.

Pentru a exemplifica, ilustrăm tabelar primele 24 de intrări (doar 9 coloane din cele 13) din setul de date propus:

Price	Area	Bedrooms	Bathrooms	Stories	Mainroad	Guestroom	Basement	Hot water
13300000	7420	4	2	3	yes	no	no	no
12250000	8960	4	4	4	yes	no	no	no
12250000	9960	3	2	2	yes	no	yes	no
12215000	7500	4	2	2	yes	no	yes	no
11410000	7420	4	1	2	yes	yes	yes	no
10850000	7500	3	3	1	yes	no	yes	no
10150000	8580	4	3	4	yes	no	no	no
10150000	16200	5	3	2	yes	no	no	no
9870000	8100	4	1	2	yes	yes	yes	no
9800000	5750	3	2	4	yes	yes	no	no
9800000	13200	3	1	2	yes	no	yes	no
9681000	6000	4	3	2	yes	yes	yes	yes
9310000	6550	4	2	2	yes	no	no	no
9240000	3500	4	2	2	yes	no	no	yes
9240000	7800	3	2	2	yes	no	no	no
9100000	6000	4	1	2	yes	no	yes	no
9100000	6600	4	2	2	yes	yes	yes	no
8960000	8500	3	2	4	yes	no	no	no
8890000	4600	3	2	2	yes	yes	no	no
8855000	6420	3	2	2	yes	no	no	no
8750000	4320	3	1	2	yes	no	yes	yes
8680000	7155	3	2	1	yes	yes	yes	no
8645000	8050	3	1	1	yes	yes	yes	no

În acest caz, ieșirea (variabila y) reprezintă coloana **Price**, iar **predictorii** $x_1, x_2, \ldots, x_{12}$ sunt toate celelalte coloane.

2.2 Cerințe

Având în vedere expunerea suportului teoretic și problema ce se dorește a fi rezolvată, aveți de implementat următoarele funcții:

- `function [Y, InitialMatrix] = parse_data_set_file(file_path)`

  Funcția `parse_data_set_file` va primi o cale relativă către un fișier text unde se află datele pentru un set oarecare.

  Formatul fișierului de intrare va fi acesta:

```
m n
Y_11 x_11 x_12 x_13 ... x_1n
Y_21 x_21 x_22 x_23 ... x_2n
Y_31 x_31 x_32 x_33 ... x_3n
...
Y_m1 x_m1 x_m2 x_m3 ... x_mn
```

  În acest caz, n este numărul de predictori, iar m se refera la numărul vectorilor de predictori $x_1, x_2, \ldots, x_n$ și la dimensiunea vectorului Y de ieșire. *InitialMatrix* reprezintă o matrice cu tipuri de date **distincte**!, adică stochează atât tipuri numerice, cât și string-uri. Pentru a gestiona acest lucru, puteți folosi tipul **Cell** din **Octave**.

- `function [FeatureMatrix] = prepare_for_regression(InitialMatrix)`

  Funcția *prepare_for_regression* modelează matricea anterioară astfel încât să conțină doar tipuri **numerice**. Cu alte cuvinte, fiecare poziție din matrice ce conține string-ul 'yes' se înlocuiește cu tipul numeric (numărul) 1, iar fiecare poziție ce conține string-ul 'no' se înlocuiește cu tipul numeric (numărul) 0. Pentru pozițiile ce au aceste valori *'semi-furnished'*, *'unfurnished'* sau *'furnished'*, acestea se vor **descompune** în două poziții cu valori numerice de 0 și 1.
  Fie următoarele cazuri:
  - Dacă poziția are valoarea *'semi-furnished'*, atunci se va descompune în două poziții cu valorile 1 și 0.
  - Dacă poziția are valoarea *'unfurnished'*, atunci se va descompune în două poziții cu valorile 0 și 1.
  - Dacă poziția are valoarea *'furnished'*, atunci se va descompune în două poziții cu valorile 0 și 0.

  Exemplu:

```
no  0 yes semi-furnished
no  2 no  semi-furnished
yes 1 yes unfurnished
yes 2 no  furnished
yes 2 no  furnished
yes 1 yes semi-furnished
no  2 no  semi-furnished
```

  Înlocuind toate string-urile cu tipuri numerice, matricea de mai sus se va transforma în:

```
0 0 1 1 0
0 2 0 1 0
1 1 1 0 1
1 2 0 0 0
1 2 0 0 0
1 1 1 1 0
0 2 0 1 0
```

După înlocuirea tuturor pozițiilor cu valori numerice, rezultatul obținut trebuie salvat în variabila de ieșire *FeatureMatrix*. Se observă că s-a mai adăugat o coloană, prin urmare s-a mărit numărul de predictori cu 1.

- `function [Error] = linear_regression_cost_function(Theta, Y, FeatureMatrix)`

  Funcția *linear_ regression_ cost_ function* implementează funcția de cost, așa cum a fost descrisă în secțiunea teoretică, folosind cei doi vectori și o matrice:
  - **Theta**, care reprezintă un vector **coloană** format din coeficienții $\theta_1, \ldots, \theta_n \in \mathbb{R}$.
  - **FeatureMatrix**, care reprezintă o matrice ce reține valorile unor predictori (adică o linie **i** din această matrice reprezintă $\boldsymbol{x}^{(i)}$ descris în suportul teoretic).
  - **Y**, care reprezintă un vector **coloană** ce conține **valorile actuale**, adică **ieșirile** ce au un anumit ordin.

  Pentru simplificarea implementării, puteți omite termenul care indică eroarea din cadrul funcției $h_{\boldsymbol{\theta}}(\boldsymbol{x})$, iar θ_0 îl puteți considera **0**.
  Se garantează faptul că dimensiunile argumentelor sunt **compatibile** pentru a prelucra funcția de cost.

- `function [InitialMatrix, Y] = parse_csv_file(file_path)`

  Funcția `parse_csv_file` va primi o cale relativă către fișierul .csv unde se află datele pentru setul **propus**.
  Formatul (parțial) al acestui fișier se află la pagina 18.
  Există funcții Octave pentru a parsa, cu ușurință, astfel de fișiere.

- `function [Theta] = gradient_descent(FeatureMatrix, Y, n, m, alpha, iter)`

  Funcția *gradient_ descent* calculează, folosind tehnica **gradientului descendent**, coeficienții $\theta_1, \ldots, \theta_n \in \mathbb{R}$ după efectuarea celor *iter* pași. Ca mai sus, $\theta_0 = 0$ (îl considerăm 0).
  De asemenea, vectorul de predictori $\boldsymbol{x}^{(i)}$ reprezintă linia **i** din matricea *FeatureMatrix*.
  Considerați această aproximație inițitală: $\theta_1 = 0, \theta_2 = 0, \ldots, \theta_n = 0$.
  Această funcție **se va testa** folosind setul de date din fișierul .csv (există funcții Octave pentru a parsa, cu ușurință, astfel de fișiere).

- `function [Theta] = normal_equation(FeaturesMatrix, Y, tol, iter)`

  Funcția *normal_ equation* calculează, cu ajutorul metodei **gradientului conjugat**, coeficienții $\theta_1, \ldots, \theta_n \in \mathbb{R}$. De asemenea, $\theta_0 = 0$.
  Această funcție trebuie să returneze un vector *Theta* cu toți coeficienții calculați.
  Dacă matricea *sistemului* nu este **pozitiv definită**, atunci *Theta* o să stocheze doar valori de 0 și o să fie returnat direct. Se garantează faptul că *iter* va fi ales în mod corespunzător.

- `function [Error] = lasso_regression_cost_function(Theta, Y, FeMatrix, lambda)`

Funcția *lasso_ regression_ cost_ function* implementează funcția de cost, așa cum a fost descrisă în secțiunea teoretică, folosind cei doi vectori, o matrice și un scalar de la intrare:

- **Theta**, care reprezintă un vector **coloană** format din coeficienții $\theta_1, \ldots, \theta_n \in \mathbb{R}$.
- **FeMatrix**, care reprezintă o matrice ce reține valorile unor predictori (adică o linie **i** din această matrice reprezintă $\boldsymbol{x}^{(i)}$ descris în suportul teoretic).
- **Y**, care reprezintă un vector **coloană** ce conține **valorile actuale**, adică **ieșirile** ce au un anumit ordin.
- λ, care reprezintă parametrul ce controlează **regularizarea**.

Pentru simplificarea implementării, puteți omite termenul care indică eroarea din cadrul funcției $h_{\boldsymbol{\theta}}(\boldsymbol{x})$, iar θ_0 îl puteți considera **0**.
Se garantează faptul că dimensiunile argumentelor sunt **compatibile** pentru a prelucra funcția de cost.

- `function [Error] = ridge_regression_cost_function(Theta, Y, FeMatrix, lambda)`

Funcția *ridge_ regression_ cost_ function* implementează funcția de cost, așa cum a fost descrisă în secțiunea teoretică, folosind cei doi vectori, o matrice și un scalar de la intrare:

- **Theta**, care reprezintă un vector **coloană** format din coeficienții $\theta_1, \ldots, \theta_n \in \mathbb{R}$.
- **FeMatrix**, care reprezintă o matrice ce reține valorile unor predictori (adică o linie **i** din această matrice reprezintă $\boldsymbol{x}^{(i)}$ descris în suportul teoretic).
- **Y**, care reprezintă un vector **coloană** ce conține **valorile actuale**, adică **ieșirile** ce au un anumit ordin.
- λ, care reprezintă parametrul ce controlează **regularizarea**.

Pentru simplificarea implementării, puteți omite termenul care indică eroarea din cadrul funcției $h_{\boldsymbol{\theta}}(\boldsymbol{x})$, iar θ_0 îl puteți considera **0**.
Se garantează faptul că dimensiunile argumentelor sunt **compatibile** pentru a prelucra funcția de cost.

3 MNIST 101 *(40p)*

3.1 Enunț

Trecând prin periplul său prin algoritmi numerici și predicție cu ajutorul regresiei liniare, Mihai face un ultim pas pentru o introducere completă în învățarea supervizată. Foarte captivat de regresia liniară, el se întreabă cum ar putea adapta algoritmul și metodele de optimizare deja cunoscute pentru regresia liniară pentru a le putea folosi și la alt gen de probleme, cum ar fi problemele de **clasificare**.

Astfel, task-ul pe care și-l propune este să clasifice poze conținând cifre zecimale scrise de mână (de la 0 la 9) folosind un model de clasificare potrivit. Pentru că este vorba despre o problemă de clasificare în mai multe clase, clasificatorul ales de Mihai este o mică rețea neurală care are un strat de input cu 400 de unități neuronale (valorile pixelilor unei poze de dimensiune 20×20), un strat de output cu 10 unități (câte una pentru fiecare clasă) și un strat ascuns, cu un număr intermediar de unități neuronale (25 de unități), folosit pentru a crește complexitatea și deci și performanța modelului de clasificare.

Figura 10: Câteva exemple din dataset-ul MNIST

3.2 Referințe teoretice

3.2.1 Adaptare a regresiei liniare: regresia logistică

Principiul de bază prin care se realizau predicții cu ajutorul regresiei liniare era faptul că rezultatul dorit reprezenta o combinație liniară a unui set de parametri dați (*features* - ați întâlnit deja câteva exemple de features în cadrul celei de-a doua părți a temei). Astfel, un model similar poate fi folosit și pentru clasificarea datelor primite într-un număr finit de clase.

Să luăm mai întâi o problemă foarte simplă de clasificare în două clase. O problemă de clasificare cu două clase are drept date de intrare un vector de parametri (la fel ca la regresia liniară), împreună cu un rezultat, reprezentat de un label (*o etichetă*). În cazul clasificării binare, acest label poate avea valorile $y \in \{0, 1\}$. Încă din acest pas observăm ineficiența aplicării regresiei liniare pentru o problemă de clasificare: regresia liniară poate da drept rezultat (valoare prezisă) orice număr real (pozitiv sau negativ), un rezultat nepotrivit pentru predicția noastră în doar 2 clase.

Din acest motiv, avem nevoie de o metodă (o *neliniaritate*) prin care să mapăm rezultatul obținut în urma combinației liniare în intervalul $[0, 1]$. Acesta este motivul pentru care, în loc de binecunoscuta ipoteză

$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x}$ să utilizăm o ipoteză modificată, de forma

$$h_{\boldsymbol{\theta}} = \sigma\left(\boldsymbol{\theta}^T \boldsymbol{x}\right)$$

unde $\sigma : \mathbb{R} \to [0, 1]$ este neliniaritatea amintită anterior. De obicei, se alege funcția *sigmoid* pentru maparea dorită, adică funcția:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

În figura de mai jos, aveți graficul funcției sigmoid, în care se evidențiază rolul acesteia de a mapa orice rezultat real în intervalul $[0, 1]$:

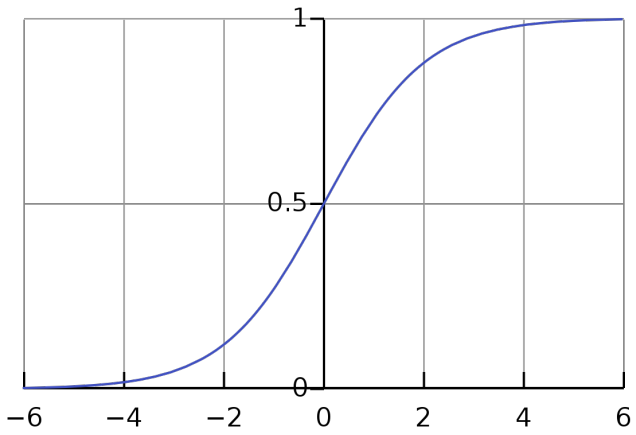


Figura 11: Graficul funcției sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$

Având acum o nouă formă a ipotezei noastre, trebuie să redefinim și funcția de cost (*loss function*) care va trebui să fie optimizată, întrucât eroarea pătratică (în termeni de normă 2 – cele mai mici pătrate) este insuficientă. Ne dorim ca un model neantrenat să aibă o eroare mare dacă valoarea ipotezei de regresie diferă semnificativ față de valoarea efectivă a clasei în care un exemplu este încadrat. De aceea, s-a introdus o nouă funcție de cost, al cărei rol este să evidențieze acest caz extrem, numită *cross-entropy*:

$$\text{cost}_i = -y^{(i)} \cdot \log\left[h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right] - (1 - y^{(i)}) \cdot \log\left[1 - h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right]$$

Funcția de cost (pe toate exemplele de training) devine:

$$J(\boldsymbol{\theta}) = \frac{1}{m} \cdot \sum_{i=1}^{m} \text{cost}_i = \frac{1}{m} \cdot \sum_{i=1}^{m} \left\{-y^{(i)} \cdot \log\left[h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right] - (1 - y^{(i)}) \cdot \log\left[1 - h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right]\right\}$$

Pentru acest caz, putem aplica tehnicile de optimizare a funcției de cost cunoscute deja din secțiunea anterioară a temei (**Gradient Descent** și o formă modificată de Gradient Conjugat), obținând un model cu performanțe foarte bune pentru task-uri simple de clasificare.

3.2.2 Neajunsurile regresiei logistice

Regresia logistică este o tehnică de învățare supervizată foarte bună atunci când avem de-a face cu probleme simple de clasificare (numărul de features este mic). Cu toate acestea, are unele neajunsuri, dintre care merită menționate următoarele:

- Regresia logistică clasică nu se poate extinde ușor la mai mult de 2 clase. Extinderea problemei de clasificare necesită câte un model particular pentru fiecare clasă introdusă (*one vs all classification*);
- Regresia logistică nu scalează la probleme de clasificare mai complexe, cum ar fi probleme specifice din zona de Computer Vision (identificarea obiectelor și procesarea imaginilor). Pentru astfel de probleme, este necesară utilizarea unor clasificatori mai complecși.

3.2.3 Extinderea de la regresia logistică la o rețea neurală. Perceptronul

Regresia logistică poate fi privită ca o rețea, așa cum este prezentat în figura de mai jos. În rețeaua dată, componentele noastre sunt reprezentate de:

- **Nodurile rețelei**, numite și **neuroni**;
- **Legăturile** între nodurile rețelei. Acestea semnifică contribuția (cu o anumită pondere[13]) a respectivului perceptron pentru calcularea valorii unui neuron din următorul strat;
- **Funcția de activare**, care reprezintă o neliniaritate. Cele mai uzuale funcții de activare sunt sigmoid (prezentat anterior), Rectified Linear Unit (ReLU) și tangenta hiperbolică. În cazul nostru, vom analiza strict cazul în care funcția de activare este sigmoid;
- Pentru modelul de mai jos, avem un anumit număr de unități neuronale de intrare și o singură unitate de ieșire, corespunzătoare clasei din care va face parte.

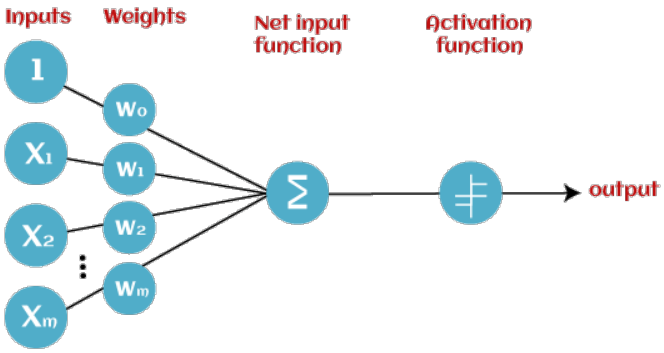


Figura 12: Perceptronul

Acest design poate fi extins prin includerea unor unități neuronale intermediare, care să formeze un **strat ascuns**[14] și prin mărirea numărului de unități neuronale de ieșire, corespunzător numărului de clase ale clasificatorului nostru. Astfel, am obținut o rețea neuronală conectată[15].

O rețea neuronală poate avea oricâte straturi ascunse, însă în cadrul acestui task noi vom folosi o arhitectură care are un singur strat ascuns, ca în figura următoare.

[13] en. *weight*
[14] en. *hidden layer*
[15] en. *fully-connected neural network*

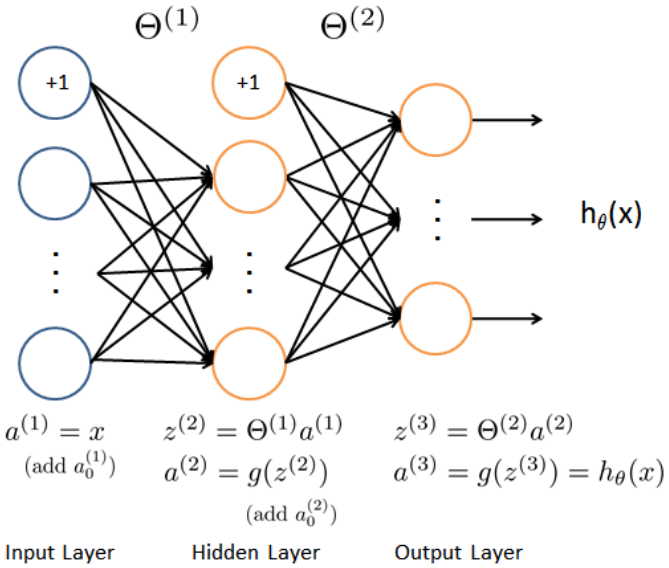


Figura 13: Arhitectura rețelei neurale folosite

Mărimile ce apar într-o rețea neurală descrisă de arhitectura de mai sus sunt:

- **Cele trei straturi existente** (denumite în literatura de specialitate *input layer*, *hidden layer* și *output layer*) au dimensiunile $s_1, s_2, s_3 \in \mathbb{N}$. Dimensiunea unui layer este reprezentată de numărul de neuroni din acel strat;
- **Numărul de clase finale** (care este egal cu numărul de neuroni din stratul de output) se notează cu $K \in \mathbb{N}$;
- Fiecare neuron dintr-un strat este caracterizat de o mărime, numită **activare**. Activările pentru layer-ul de input sunt chiar datele de intrare în rețeaua neurală (în cazul nostru, vor fi 400 de pixeli ai unor poze 20×20). Pentru layer-ul de output, activările sunt chiar predicțiile noastre (layer-ul de output va avea 10 unități neuronale). Pentru layer-ul intermediar (hidden) și pentru cel de output, activările vor fi determinate în funcție de toate activările neuronilor din layer-ul anterior (de aici și noțiunea de *fully-connected*);
- Pentru trecerea de la un layer la altul, vom utiliza o serie de parametri care alcătuiesc două matrice, $\Theta^{(1)} \in \mathbb{R}^{s_2 \times (s_1+1)}$ și $\Theta^{(2)} \in \mathbb{R}^{s_3 \times (s_2+1)}$.

3.2.4 Predicție. Forward propagation

Predicția clasei din care face parte un anumit exemplu este un procedeu efectuat atât în etapa de antrenare a modelului, cât și în etapa de testare (după antrenare). În cazul rețelei neurale, procedeul prin care se realizează determinarea activărilor neuronilor din layer-ul intermediar și determinarea predicțiilor finale se numește *forward propagation*. Acest procedeu are următorii pași:

- Fie $\left(\boldsymbol{x}^{(i)}, y^{(}i)\right)$ un exemplu din dataset-ul de antrenare, unde $\boldsymbol{x}^{(i)}$ reprezintă datele de intrare în rețeaua neurală și $y^{(i)}$ reprezintă clasa din care face parte exemplul dat;
- Se construiește vectorul activărilor neuronilor din layer-ul de input din datele de intrare, la care se adaugă o unitate (*bias*):

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} 1 \\ \boldsymbol{x}^{(i)} \end{bmatrix}$$

- Se aplică prima transformare liniară, dată de matricea $\Theta^{(1)}$, și se obține un vector $\boldsymbol{z}^{(2)}$ al rezultatelor intermediare (pre-activări).
$$\boldsymbol{z}^{(2)} = \Theta^{(1)} \cdot \boldsymbol{a}^{(1)}$$
- Se aplică funcția de activare (în cazul nostru, funcția sigmoid – ec. 3). Funcția de activare va fi implementată vectorizat, rezultatul aplicării acesteia pe un tablou fiind aplicarea funcției *sigmoid* pe fiecare element din acel tablou.
$$\boldsymbol{a}^{(2)} = \sigma\left(\boldsymbol{z}^{(2)}\right)$$
- Se adaugă o unitate la începutul vectorului activărilor (pentru *bias*):
$$\boldsymbol{a}^{(2)} = \begin{bmatrix} 1 \\ \boldsymbol{a}^{(2)} \end{bmatrix}$$
- Se aplică și cea de-a doua transformare liniară, astfel:
$$\boldsymbol{z}^{(3)} = \Theta^{(2)} \cdot \boldsymbol{a}^{(2)}$$
- Se aplică, din nou, funcția de activare (sigmoid):
$$\boldsymbol{a}^{(3)} = \sigma\left(\boldsymbol{z}^{(3)}\right)$$
- Aflându-ne în contextul ultimului layer, nu mai adăugăm unitatea pentru *bias*, iar activările obținute vor reprezenta predicțiile noastre pentru clasele propuse.

În cazul concret al task-ului nostru, vom obține un vector de 10 predicții, fiecare element reprezentând o predicție (similaritate) a exemplului dat cu una dintre cele 10 clase disponibile.

3.2.5 Determinarea gradienților. Backpropagation

La fel cum am observat la regresia liniară, orice model de învățare are nevoie de o modalitate prin care să își optimizeze (în acest caz, minimizeze) funcția de cost prin ajustarea parametrilor săi.

Pentru început, să scriem funcția de cost pentru o rețea neurală. Această funcție de cost reprezintă o generalizare a funcției de cost pentru regresie logistică și se bazează, de asemenea, pe **cross entropy**.

Înainte de a vă furniza formula, clarificăm că:

- $\Theta^{(1)}$ este o **matrice**, $\Theta^{(1)} \in \mathbb{R}^{s_2 \times (s_1+1)}$;
- $\Theta^{(2)}$ este o **matrice**, $\Theta^{(2)} \in \mathbb{R}^{s_3 \times (s_2+1}$;
- $\boldsymbol{\theta}$ este un **vector**, $\boldsymbol{\theta} \in \mathbb{R}^{s_2 \cdot (s_1+1) + s_3 \cdot (s_2+1)}$, și reprezintă vectorul care conține toate elementele din cele 2 matrice, în mod desfășurat.

$$J(\theta) = \frac{1}{m} \cdot \sum_{i=1}^{m} \text{cost}_i = \frac{1}{m} \cdot \sum_{i=1}^{m} \sum_{k=1}^{K} \left\{ -y_k^{(i)} \cdot \log\left[h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)_k\right] - \left(1 - y_k^{(i)}\right) \cdot \log\left[1 - h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)_k\right] \right\}$$
$$+ \frac{\lambda}{2m} \left[\sum_{j=2}^{s_1+1} \sum_{k=1}^{s_2} \left(\Theta_{k,j}^{(1)}\right)^2 + \sum_{j=2}^{s_2+1} \sum_{k=1}^{s_3} \left(\Theta_{k,j}^{(2)}\right)^2 \right]$$

A se observa că, la fel ca la regresie liniară, nu am regularizat și ponderile corespunzătoare activărilor constante (*biases*).

Dacă la regresie liniară ajustarea parametrilor se putea realiza cu ajutorul gradienților determinați analitic (sub forma derivatelor parțiale ale funcției de cost), în cazul rețelelor neurale acest lucru nu mai este posibil direct, întrucât o expresie analitică a gradienților este foarte dificil de obținut.

Cu toate acestea, putem folosi un algoritm cu ajutorul căruia să determinăm gradienții pentru fiecare parametru al modelului. Acest algoritm se numește **backpropagation**.

Algoritmul de backpropagation se bazează pe soluționarea gradienților prin intermediul determinării **erorilor de activare**. Concret, să presupunem că tocmai am realizat *forward propagation* pentru a determina predicțiile exemplului de antrenament curent pentru fiecare clasă. Atunci, putem defini eroarea care a apărut în activarea din layer-ul l, în neuronul de indice k, notată cu $\delta_k^{(l)}$. De asemenea, vom mai păstra matricele $\Delta^{(1)}$ și $\Delta^{(2)}$, de aceleași dimensiuni cu matricele $\Theta^{(1)}$ și $\Theta^{(2)}$, în care vom acumula gradienții parametrilor rețelei.

Pe scurt, pașii pentru algoritmul de *backpropagation* sunt:

- Determinăm eroarea în layer-ul de output:
$$\delta^{(3)} = a^{(3)} - y^{(i)}$$
- Putem acumula gradienții pentru parametrii care fac trecerea de la layer-ul intermediar la layer-ul de output, folosind formula:
$$\Delta^{(2)} = \Delta^{(2)} + \delta^{(3)} \cdot (a^{(2)})^T$$
- Pentru determinarea erorii în layer-ul intermediar, folosim formula:
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* \sigma'(z^{(2)})$$
unde σ' este derivata funcției de activare *sigmoid*, avem adevărată relația $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$. De asemenea, operatorul $.*$ reprezintă produsul Hadamard a două tablouri (produsul *elementwise*).
- Eliminăm prima componentă din $\delta^{(2)}$ (aceasta este componenta pentru bias, acolo unde nu are sens să calculăm o eroare în valoarea activării).
- Acumulăm gradienții și pentru parametrii care fac trecerea de la layer-ul de input la cel intermediar:
$$\Delta^{(1)} = \Delta^{(1)} + \delta^{(2)} \cdot (a^{(1)})^T$$
- După ce am realizat acumularea gradienților pentru toate exemplele de antrenament, putem împărți la numărul exemplelor de antrenament:
$$\frac{\partial J(\Theta)}{\partial \theta_{ij}^{(l)}} = \frac{1}{m} \cdot \Delta_{ij}^{(l)}$$
- La final, putem adăuga și termenul corespunzător regularizării (numai pentru $j > 1$, pentru $j = 1$ formula anterioară rămâne valabilă):
$$\frac{\partial J(\Theta)}{\partial \theta_{ij}^{(l)}} = \frac{1}{m} \cdot \Delta_{ij}^{(l)} + \frac{\lambda}{m} \cdot \Theta_{ij}^{(l)}$$

Având la dispoziție acum gradienții și valoarea funcției de cost, putem realiza optimizarea funcției de cost prin Gradient Descent sau Gradient Conjugat.

3.2.6 Inițializarea parametrilor

În cazul rețelelor neurale, inițializarea parametrilor (elementelor matricelor) cu valori nule nu este posibilă (avem simetrie și vom obține o simetrie în ceea ce privește clasificarea exemplului nostru în diversele

clase disponibile). De asemenea, inițializarea cu zero a parametrilor duce la anularea gradienților (rețeaua neurală este incapabilă să învețe), o problemă care în Deep Learning se numește Vanishing Gradient Problem. Puteți citi mai multe despre această problemă aici.
Soluția este inițializarea parametrilor cu valori aleatoare, din intervalul $(-\epsilon, \epsilon)$. Empiric, s-a constatat că o valoare potrivită pentru ϵ este dată de următoarea formulă:

$$\epsilon_0 = \frac{\sqrt{6}}{\sqrt{L_{prev} + L_{next}}}$$

3.3 Cerințe

Având în vedere referințele teoretice, aveți de implementat următoarele funcții:

- `function [X, y] = load_dataset(path)`
  Funcția *load_dataset* primește o cale relativă la un fișier .mat și încarcă în memorie acel fișier, returnând matricea care conține exemplele folosite pentru training și pentru test. Liniile matricei X vor reprezenta exemplele de date. Pentru fiecare linie, se va adăuga la început o coloană care să reprezinte termenul de *bias* (o coloană numai cu valori de 1).

- `function [x_train, y_train, X_test, y_test] = split_dataset(X, y, percent)`
  Funcția *split_dataset* primește un dataset, așa cum a fost el returnat de funcția anterioară (training examples, împreună cu labels) și împarte setul de date în 2 seturi: un set de training și un set de test, ambele reprezentate printr-o matrice de features și un vector de clase. Împărțirea pe cele 2 seturi se va face astfel: se amestecă exemplele, iar apoi o fracțiune egală cu parametrul *percent* din exemplele date în dataset va fi adăugată în setul de training (valorile de retur X_train și y_train), iar restul exemplelor vor fi plasate în setul de test.

- `function [matrix] = initialize_weights(L_prev, L_next)`
  Funcția *initialize_weights* primește dimensiunile (numărul de neuroni) celor 2 straturi între care se aplică transformarea liniară și întoarce o matrice cu elemente aleatoare din intervalul $(-\epsilon, \epsilon)$, conform precizărilor din referințe.

- `function [grad, J] = cost_function(params, X, y, lambda, input_layer_size, hidden_layer_size, output_layer_size)`
  Funcția *cost_function* primește următorii parametri:
  - *params* reprezintă un vector coloană care conține toate valorile ponderilor (*weights*) din matricele $\Theta^{(1)}$ și $\Theta^{(2)}$. Cu alte cuvinte, folosind elementele din acest vector și dimensiunile straturilor putem construi matricele pentru transformările liniare. Hint: **reshape**.
  - X reprezintă mulțimea exemplelor de training, fără labels asociate (*feature matrix*).
  - y reprezintă label-urile asociate exemplelor de mai sus.
  - *input_layer_size* reprezintă dimensiunea stratului de input.
  - *hidden_layer_size* reprezintă dimensiunea stratului intermediar/ascuns.
  - *output_layer_size* reprezintă dimensiunea stratului final (care este egal cu numărul de clase).

  Funcția returnează un vector de aceeași dimensiune cu parametrul *params*, obținut prin desfășurarea (*unrolling*) matricelor în care calculăm gradienții după aplicarea algoritmului de *backpropagation*, și J, care reprezintă funcția de cost pentru valoarea momentană a parametrilor.

- `function [classes] = predict_classes(X, weights, input_layer_size, hidden_layer_size, output_layer_size)`
  Funcția primește un set de exemple de test și vectorul pentru weights, precum și dimensiunile layerelor rețelei și întoarce un vector cu toate predicțiile pentru exemplele date din setul de test.
  Hint: **forward propagation**

3.3.1 Restricții și precizări

- În dataset-ul folosit, pentru clasa corespunzătoare cifrei 0 s-a folosit label-ul 10, tocmai pentru ca label-urile să fie conforme cu indexarea din GNU Octave.

4 Regulament

Regulamentul general al temelor se găsește pe platforma Moodle (Temele de casă). Vă rugăm să îl citiți integral înainte de continua cu regulile specifice acestei teme.

4.1 Arhivă

Soluția temei se va trimite ca o arhivă **zip**. Numele arhivei trebuie să fie de forma **Grupă_NumePrenume_TemaX.zip** - exemplu: `311CA_Alexandru-Mihai-IulianBuzea_Tema1.zip`.

Vom reveni cu detalii referitoare la conținutul arhivei de îndată ce vom publica și checkerul, împreună cu posibilitatea de a submite tema.

Numele şi extensiile fişierelor trimise **NU** trebuie să conţină spaţii sau majuscule, cu exceptia fișierului README (care este are nume scris cu majuscule și nu are extensie).

Nerespectarea oricărei reguli din secțiunea **Arhivă** aduce un punctaj **NUL** pe temă.

4.2 Punctaj

Distribuirea punctajului:

- Markov is coming: 40p
- Problema 2: 40p
- Problema 3: 40p
- Modularizare, claritatea codului și explicațiile din README: 10p

ATENȚIE! Punctajul maxim pe temă este 100p. Acesta reprezintă 1p din nota finală la această materie. La această temă se pot obține până la 130p (există un bonus de 30p), adică un punctaj maxim de 1.3p din nota finală.

4.2.1 Reguli și precizări

- Punctajul pe teste este cel acordat de scriptul/scripturile de *check*, rulat(e) pe VMChecker. Echipa de corectare își rezervă dreptul de a depuncta pentru orice încercare de a trece testele fraudulos (de exemplu prin hardcodare).
- Punctajul pe calitatea explicațiilor și a codului se acordă în mai multe etape:
  - Codul sursă trebuie să fie însoțit de un fișier README care trebuie să conțină informațiile utile pentru înțelegerea funcționalității, modului de implementare și utilizare a soluțiilor cerute. Acesta evaluează, de asemenea, abilitatea voastră de a documenta complet și concis programele pe care le produceți și va fi evaluat de către echipa de asistenți. In funcție de calitatea documentației, se vor aplica depunctări sau bonusuri.
  - La corectarea manuală se va acorda un bonus de maximum 10 puncte pentru modularizare, claritatea explicațiilor din README și coding style.
  - Deprinderea de a scrie cod sursă de calitate, este un obiectiv important al materiei. Sursele greu de înteles, modularizate neadecvat sau care prezintă hardcodări care pot afecta semnificativ mentenabilitatea programului cerut, pot fi depunctate adițional.

  - În această etapă se pot aplica depunctări mai mari de 30p.

  - Deși nu impunem un anumit standard de coding style, ne așteptăm să întâlnim cod lizibil, documentat corespunzător. Erorile grave de coding style (cod ilizibil, variabile denumite nesugestiv, hardcodarea sau lipsa comentariilor de orice fel) vor fi depunctate corespunzător.

4.3 Alte precizări

- Implementarea se va face in limbajul **GNU Octave**, iar tema va fi compilată şi testată **DOAR** într-un mediu **LINUX**. Nerespectarea acestor reguli aduce un punctaj **NUL**.

- Tema trebuie trimisă sub forma unei arhive pe site-ul cursului curs.upb.ro. Vom reveni cu detalii referitoare la modalitatea de corectare ulterior publicării checkerului.

- Tema poate fi submisă de oricâte ori fără depunctări până la deadline. Mai multe detalii se găsesc în regulamentul de pe ocw.

- Ultima temă submisă pe vmchecker poate fi rulată de către responsabili de mai multe ori în vederea verificării faptului că nu aveți buguri în sursă. Vă recomandăm să verificați **local** tema de mai multe ori pentru a verifica că punctajul este mereu același, apoi să încărcați tema.

- Temele vor fi testate antiplagiat. Este interzisă publicarea pe forum și în orice spațiu public (GitHub) a întregului cod sau a unor porțiuni din cod care reprezintă soluții la task-urile propuse. Nu este permisă colaborarea de orice fel în vederea realizării temelor de casă. Soluțiile care nu respectă aceste criterii vor fi punctate cu 0 (zero) puncte.

- Preluarea de cod din resurse publice este permisă doar în contextul menționării sursei în comentarii **și** în README, cu excepția resurselor care reprezintă rezolvări directe ale task-urilor din temă. Soluțiile care nu respectă aceste criterii vor fi punctate cu 0 (zero) puncte.

- Este interzisă folosirea ChatGPT sau a oricărei formă de inteligență artificială în rezolvarea temei de casă. Soluțiile care nu respectă acest criteriu vor fi punctate cu 0 (zero) puncte.