

**UNIVERSITATEA "AUREL VLAICU" DIN ARAD**

**FACULTATEA DE ȘTIINȚE EXACTE**

**DOMENIUL: INFORMATICĂ**

**PROGRAMUL DE STUDIU: INFORMATICĂ**

FORMA DE ÎNVĂŢĂMÂNT CU FRECVENȚĂ

**LUCRARE DE LICENȚĂ**

ÎNDRUMĂTOR ŞTIINŢIFIC:

Lector univ. dr. – ANDREI GABOR

Asist. univ. drd. ing. – ANTONIO M. F. LUPUTI

ABSOLVENT:

MOTOC A. DAN

**ARAD**

**Iunie 2024**



**UNIVERSITATEA "AUREL VLAICU" DIN ARAD**

**FACULTATEA DE ȘTIINȚE EXACTE**

**DOMENIUL: INFORMATICĂ**

**PROGRAMUL DE STUDIU: INFORMATICĂ**

FORMA DE ÎNVĂŢĂMÂNT CU FRECVENȚĂ

**DEZVOLTAREA REȚELELOR NEURONALE ÎN PROCESAREA LIMBAJULUI NATURAL**

ÎNDRUMĂTOR ŞTIINŢIFIC:

Lector univ. dr. – ANDREI GABOR

Asist. univ. drd. ing. – ANTONIO M. F. LUPUTI

ABSOLVENT:

MOTOC A. DAN

**ARAD**

**Iunie 2024**

**UNIVERSITATEA "AUREL VLAICU" DIN ARAD**

**FACULTATEA DE ȘTIINȚE EXACTE DOMENIUL: INFORMATICĂ**

**PROGRAMUL DE STUDIU: INFORMATICĂ**

Nr. din 12 IUNIE 2024

APROBAT

DECAN

Conf. dr. MARIUS TOMESCU

VIZAT

Îndrumător ştiinţific

Lector univ. ANDREI GABOR

**DATE PERSONALE ALE CANDIDATULUI**

**1. Date privind identitatea persoanei**

Numele: Motoc

Numele anterior: -

Prenumele: Dan-Adrian

**2. Sexul: (M/F):** M

**3. Data şi locul naşterii:**

Ziua / luna / anul: 13/10/1994

Locul (localitate, judeţ) Arad, Jud. Arad

**4. Prenumele părinţilor:**

Tata: Mihai

Mama: Daniela

**5. Domiciliul permanent:** Str. Banu Maracine, nr. 1, loc. Arad, cod postal 310150, telefon: 0720562043, e-mail: robnewton90@gmail.com

**6. Sunt absolventă promoţia:** Iunie 2024

**7. Forma de învăţământ pe care am absolvit-o este: (cu frecvenţă, cu frecvenţă redusă, ID), cu taxă/fără taxă:** zi, cu taxă;

**8. Locul de muncă (dacă e cazul):** SCJUBH

**9. Solicit înscrierea la examenul de LICENȚĂ,** Sesiunea IUNIE anul 2024**.**

**10. Lucrarea de LICENTĂ pe care o susţin are următorul titlu:** Dezvoltarea rețelelor neuronale în procesarea limbajului natural

**11. Îndrumător ştiinţific:** Lector univ. dr. Andrei Gabor

**12. Menţionez că susţin examenul de finalizare a studiilor pentru prima oară şi declar pe propria-mi răspundere că am luat la cunoştinţă de prevederile art. 143 din Legea 1/2011. Declar că prezenta lucrare nu este realizată prin mijloace frauduloase, fiind conştient de faptul că, dacă se dovedeşte contrariul, diploma obţinută prin fraudă îmi poate fi anulată, conform art. 146 din Legea 1/2011.**

**SEMNĂTURA,**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

REFERAT PRIVIND LUCRAREA DE LICENŢĂ A

ABSOLVENTULUI Motoc Dan-Adrian

DOMENIUL INFORMATICĂ

PROGRAMUL DE STUDIU INFORMATICĂ

PROMOŢIA 2024

**1. Titlul lucrării**: Dezvoltarea Reţelelor Neuronale în Procesarea Limbajului Natural

**2. Structura lucrării** Lucrarea are 47 pagini şi este structurată pe 3 capitole

Introducere

Capitolul 1 Gradient Descent: Fundamente și Implicații în Învățarea Automată

Capitolul 2 Implementarea Rețelei Neuronale Multi-Layer Perceptron (MLP) fără Utilizarea de Biblioteci Externe

Capitolul 3 Evoluția și Impactul Modelării Lingvistice Neuronale în Prelucrarea Limbajului Natural

**3. Aprecieri asupra conţinutului lucrării de licenţă, organizare logică, mod de abordare, complexitate, actualitate, deficienţe.**

Lucrarea de licență este foarte bine organizată și oferă o perspectivă didactică asupra unui subiect relevant în multe domenii ale vieții. Modul de prezentare este captivant și îndeamnă cititorul să continue lectura, având o abordare ludică. Introducerea este esențială, oferind o înțelegere clară a potențialului metodelor aplicate direct printr-un exemplu. Această manieră de a introduce tehnicile teoretice detaliate în capitolele 2 și 3 stimulează interesul pentru acest domeniu nou și inovator, fiind în același timp îmbogățită de rezultate teoretice elegante în anumite secțiuni.

În ceea ce privește complexitatea, lucrarea tratează teme dificile pentru studenții de licență în informatică. Înțelegerea metodelor și algoritmilor utilizați este probabil cea mai importantă și mai puțin vizibilă parte a lucrării, esențială pentru obținerea rezultatelor prezentate în ultimul capitol. Fără această înțelegere, rezultatele nu ar fi fost posibile..

**4. Aprecieri asupra lucrării** (se va menţiona: numărul titlurilor bibliografice consultate, frecvenţa notelor de subsol, calitatea şi actualitatea surselor consultate; modul în care absolventul a prelucrat informaţiile din sursele bibliografice, contribuţii originale)

Lucrarea este redactată conform indicaţiilor metodologice. Ea include 26 de referințe bibliografice, toate publicații științifice. Deoarece subiectul este extrem de actual, multe dintre referințe sunt recente. Aceste surse bibliografice au fost fie stabilite în prealabil de către coordonator, fie căutate și descoperite de candidat.

**5. Concluzii** (valoarea lucrării elaborate de absolvent, relevanţa studiului întreprins, competenţele absolventului, consecvenţa şi seriozitatea de care a dat dovadă absolventul pe parcursul documentării şi elaborării lucrării)

În elaborarea lucrării de licenţă și a dezvoltării aplicației aferente, candidatul a dat dovadă de seriozitate și curiozitate. Această curiozitate împreună cu o forță de muncă puternică a făcut ca planul iniţial pentru licenţă să fie atins cu succes. Candidatul a dat dovadă de consecvenţă și rigoare pe parcursul redactării manuscrisului, precum și calități tehnice de remarcat. Am fost personal surprins de implicarea şi puterea de muncă de care a dat dovadă pe tot parcursul elaborării lucrării de licență şi a punerii în practică a algoritmilor analizați.

6. Redactarea lucrării respectă normele de redactare.

7. Nu există suspiciuni de realizare prin fraudă a prezentei lucrări.

8. Procentul de similitudine din Raportul de similitudine al lucrării este CS1: 0.32%, CS2: 0.00%, BDL: 0.00%, CIT: 3.28%, mai mic de 25%.

9. Consider că lucrarea îndeplineşte condiţiile pentru susţinere în sesiunea de Examen de licenţă din cadrul Programului de studii Informatică din cadrul Facultății de Științe Exacte,

Arad, Îndrumător ştiinţific,

Data, 12/06/2024 Lect. univ. dr. Andrei Gabor

Cuprins

[Introducere 1](#_Toc163118730)

[1. Gradient Descent: Fundamente și Implicații în Învățarea Automată 5](#_Toc163118731)

[1.1. Funcția de Cost 5](#_Toc163118732)

[1.1.1. Fundamentele Teoretice ale Funcție de Cost 5](#_Toc163118733)

[1.1.2. Graficul 2D al Funcției de Cost - Eroare Medie Pătratică 6](#_Toc163118734)

[1.1.3. Graficul 3D al Funcției de Cost - Eroare Medie Pătratică 7](#_Toc163118735)

[1.2. Calculul Gradientului 8](#_Toc163118736)

[1.2.1. Fundamentele Teoretice ale Calculului Gradientului 8](#_Toc163118737)

[1.2.2. Plotarea Traseului Gradientului în Python 10](#_Toc163118738)

[1.3. Variante ale Gradient Descent 11](#_Toc163118739)

[1.3.1. Batch Gradient Descent 11](#_Toc163118740)

[1.3.2. Stochastic și Mini-Batch Gradient Descent 12](#_Toc163118741)

[1.3.3. Optimizări Adaptative în Gradient Descent 13](#_Toc163118742)

[2. Implementarea Rețelei Neuronale Multi-Layer Perceptron (MLP) fără Utilizarea de Biblioteci Externe 16](#_Toc163118743)

[2.1. Clasa Valoare 16](#_Toc163118744)

[2.1.1. Componente ale Clasei Valoare 16](#_Toc163118745)

[2.1.2. Funcționalități Cheie 17](#_Toc163118746)

[2.1.3. Utilizare în Antrenarea Rețelelor Neurale 20](#_Toc163118747)

[2.2. Propagarea printr-un Singur Neuron 20](#_Toc163118748)

[2.2.1. Etape 20](#_Toc163118749)

[2.2.2. Funcția de activare - tangenta hiperbolică (tanh) 21](#_Toc163118750)

[2.2.3. Perceptronul lui Rosenblatt 23](#_Toc163118751)

[2.3. Arhitectura Rețelelor Neuronale: Neuron, Layer, MLP 24](#_Toc163118752)

[2.3.1. Clasa neuron 24](#_Toc163118753)

[2.3.2. Stratul Neuronal 25](#_Toc163118754)

[2.3.3. Construcția și Optimizarea Perceptronului Multistrat (MLP) 27](#_Toc163118755)

[2.3.4. Antrenarea MLP 29](#_Toc163118756)

[3. Evoluția și Impactul Modelării Lingvistice Neuronale în Prelucrarea Limbajului Natural 31](#_Toc163118757)

[3.1. Fundamentele Modelării Lingvistice 32](#_Toc163118758)

[3.1.1. Evoluția Istorică și Contextuală, de la Modelele Bigram la Rețele Neuronale Complexe 32](#_Toc163118759)

[3.1.2. MLP în Procesarea Limbajului Natural 33](#_Toc163118760)

[3.1.3. RNN și Transformeri în Procesarea Limbajului Natural 34](#_Toc163118761)

[3.2. Modelarea și Generarea de Text cu Bigrame folosind Rețele Neuronale Profunde în Prelucrarea Limbajului Natural 34](#_Toc163118762)

[3.2.1. Explorarea Datelor și Preprocesarea 35](#_Toc163118763)

[3.2.2. Vizualizarea Frecvenței Bigramelor Folosind Tensori Bidimensionali 35](#_Toc163118764)

[3.2.3. Popularea Tensorului N, Normalizarea datelor 38](#_Toc163118765)

[Bibliografie 40](#_Toc163118766)

# Introducere

***De la Concepte Simple la Aplicații Sofisticate: Evoluția Rețelelor Neurale în Procesarea Limbajului Natural***

Dezvoltarea rețelelor neurale în procesarea limbajului natural (PLN) reprezintă unul dintre cele mai fascinante progrese în domeniul inteligenței artificiale, schimbând radical modul în care mașinile interpretează și generează limbaj uman. Această introducere va urmări să explice, într-o manieră coerentă și gradată, principiile de bază ale rețelelor neurale și impactul lor semnificativ în PLN, evidențiind cum această tehnologie evoluează de la concepte simple la aplicații complexe și sofisticate.

La început, este esențial să înțelegem conceptul de rețea neurală și rostul său în simularea proceselor cognitive umane. Inspirându-se din biologia neuronală, rețelele neurale sunt sisteme de calcul care imită modul în care neuronii din creierul uman interacționează. Acestea sunt compuse din noduri (sau "neuroni artificiali") interconectate, care procesează informațiile prin transmiterea semnalelor între straturi [1]. Această structură de bază permite rețelelor neurale să învețe din exemple, ajustând conexiunile între neuroni în funcție de erorile observate, un proces cunoscut sub numele de "învățare supervizată".

Învățarea profundă, o ramură specifică a rețelelor neurale, introduce conceptul de "straturi ascunse" care permit modelului să învețe reprezentări la niveluri multiple de abstractizare [1]. Această capacitate de a construi reprezentări complexe ale datelor face rețelele neurale profund apte pentru sarcini de PLN, cum ar fi traducerea automată sau recunoașterea vorbirii, unde nuanțele și contextul limbajului trebuie înțelese cu precizie [2].

Un aspect central în antrenarea rețelelor neurale îl constituie optimizarea, adică procesul prin care modelul își ajustează parametrii pentru a minimiza o funcție de cost. În acest context, algoritmul de optimizare a gradientului descendent joacă un rol cheie, facilitând ajustarea ponderilor rețelei pentru a îmbunătăți performanța modelului pe seturi de date de antrenament [3]. Variante ale acestui algoritm, cum ar fi gradientul descendent stochastic (SGD) [4] și Adam [5], au fost dezvoltate pentru a spori eficiența și convergența în antrenarea modelelor complexe.

Odată cu creșterea volumului de date și a complexității modelelor, eficiența computatională devine crucială. Lucrări precum cele ale lui LeCun et al. [6] au introdus tehnici de optimizare specifice, precum metodele eficiente de backpropagation, care reduc timpul necesar pentru antrenarea rețelelor neurale, făcând posibilă procesarea rapidă a limbajului natural la scară largă.

Pe măsură ce tehnologia avansează, rețelele neurale continuă să joace un rol fundamental în îmbunătățirea sistemelor de PLN, permițând dezvoltarea de aplicații din ce în ce mai sofisticate și intuitive. Această evoluție nu numai că deschide noi orizonturi în interacțiunea om-mașină, dar și subliniază importanța cercetării continue în domeniul învățării automate și al inteligenței artificiale.

***Bazele Rețelelor Neurale în Procesarea Limbajului Natural: Înțelegerea Arhitecturii și Principiilor Fundamentale***

1. Arhitectura și Principiile Rețelelor Neurale

Inițial, rețelele neurale au fost inspirate de structura neuronală a creierului uman, având la bază un ansamblu de unități de procesare - neuroni artificiali - care simulează funcționarea sinaptică prin intermediul ponderilor ajustabile [1]. Această fundamentare biologică a oferit o bază pentru dezvoltarea algoritmilor capabili să învețe din date, prin adaptarea ponderilor în funcție de erorile generate de predicții inexacte. Este esențial să recunoaștem că această capacitate de auto-îmbunătățire a rețelelor neurale a constituit piatra de temelie pentru avansurile în PLN, facilitând interpretarea semanticii și sintaxei limbajului uman la un nivel de complexitate neegalat anterior.

2. Învățarea Profundă și Reprezentarea Ierarhică a Datelor

Învățarea profundă, caracterizată prin introducerea de straturi ascunse, a revoluționat modul în care rețelele neurale pot modela relațiile complexe din date [2]. Aceste straturi intermediare permit modelului să construiască reprezentări la diferite grade de abstractizare, un aspect crucial în procesarea eficientă a limbajului, unde semnificația poate depinde de contextul extins sau de subtilitățile lingvistice. Capacitatea de a extrage și de a învăța caracteristici semnificative din secvențe de text lungi subliniază importanța profundă a învățării profunde în dezvoltarea sistemelor de PLN capabile să gestioneze sarcini de o complexitate remarcabilă, cum ar fi generarea de text coerent și relevant contextual.

3. Optimizarea și Evoluția Algoritmilor de Învățare

Un pilon central în antrenarea eficientă a rețelelor neurale este reprezentat de optimizarea algoritmică, în special prin utilizarea variantelor de gradient descendent [3][5][6]. Acești algoritmi, prin ajustarea iterativă a ponderilor rețelei pentru minimizarea unei funcții de cost, constituie fundamentul mecanismului de învățare. În contextul PLN, optimizarea fină a acestor modele este esențială pentru a gestiona ambiguitățile și variațiile lingvistice, asigurând că rețelele neurale pot interpreta și genera limbaj într-un mod cât mai apropiat de înțelegerea umană. Discuțiile academice se concentrează adesea pe compararea eficacității diferitelor strategii de optimizare, subliniind importanța selecției algoritmului adecvat în funcție de specificul sarcinii de PLN abordate.

4. Provocările Scalabilității și Eficienței Computationale

Cu expansiunea seturilor de date și a complexității modelelor, eficiența computatională devine o provocare semnificativă. Implementările avansate de backpropagation [4], care reduc timpul necesar pentru antrenarea rețelelor neurale, aduc contribuții valoroase la gestionarea acestor provocări. Aceste metode optimizează utilizarea resurselor computaționale, permițând antrenarea modelelor de dimensiuni mari pe seturi de date extinse, o cerință frecventă în aplicațiile de PLN.

5. Impactul și Aplicațiile în Sfera PLN

Progresele în rețelele neurale au catalizat dezvoltarea unei game largi de aplicații PLN, de la sisteme de traducere automată la platforme de analiză a sentimentelor și asistenți virtuali. Aceste inovații subliniază rolul transformațional al rețelelor neurale în facilitarea unei interacțiuni om-mașină mai naturale și mai eficiente, deschizând noi direcții de explorare în inteligența artificială.

***Evoluția Arhitecturilor Rețelelor Neurale în PLN: De la Rețelele Neuronale Simple la Transformeri și GPT-3***

Explorarea evoluției arhitecturilor rețelelor neurale în procesarea limbajului natural (PLN) ne conduce printr-o călătorie istorică fascinantă, marcând progresele semnificative realizate în acest domeniu. Această secțiune își propune să facă o trecere în revistă a acestor dezvoltări, punctând momente cheie și contribuții semnificative care au modelat peisajul actual al PLN.

Începând cu rețelele neurale simple din anii '80 și '90, care se bazau pe straturi dense și algoritmi de backpropagation pentru a învăța relații simple în date [7], am asistat la o transformare profundă a capabilităților și complexității modelelor de PLN. Aceste modele inițiale, deși limitate în capacitatea de a procesa secvențe lungi de text sau de a capta dependențe temporale, au pus bazele pentru inovațiile ulterioare.

Evoluția către rețelele neurale convoluționale (CNNs) a reprezentat un pas major înainte, adaptând o arhitectură care a avut succes inițial în procesarea imaginilor pentru a trata date textuale. Aplicarea CNNs în PLN, demonstrată în lucrări precum cele ale lui Kim [8], a arătat că modelele pot învăța reprezentări eficiente ale textului prin extragerea de caracteristici locale – cum ar fi grupuri de cuvinte sau fraze – care sunt relevante pentru sarcini precum clasificarea textului.

Rețelele neurale recurente (RNNs), în special variantele lor avansate, cum ar fi LSTM (Long Short-Term Memory) și GRU (Gated Recurrent Units), au marcat următorul val de inovație [9]. Aceste modele au fost proiectate pentru a aborda limitările CNNs și ale RNNs tradiționale în manipularea dependențelor de lungă durată, esențiale în înțelegerea textului. RNNs și-au dovedit eficacitatea într-o gamă largă de aplicații PLN, de la generarea de text la traducerea automată, prin capacitatea lor de a procesa secvențe de date în mod secvențial, păstrând un „context” al informațiilor procesate anterior.

Introducerea arhitecturilor de tip Attention și, mai apoi, a Transformerilor a reprezentat probabil cea mai semnificativă inovație în PLN din ultimii ani [10]. Arhitectura Transformer, introdusă de Vaswani et al., a revoluționat înțelegerea și generarea limbajului, oferind un mecanism mai eficient și mai flexibil pentru modelarea dependențelor între cuvinte, indiferent de distanța lor în text. Această abordare a permis dezvoltarea modelelor de tip BERT (Bidirectional Encoder Representations from Transformers) [11], GPT (Generative Pre-trained Transformer) [12], și multe altele, care au stabilit noi standarde de performanță într-o serie vastă de sarcini PLN.

Ultimele tendințe în arhitecturile de rețele neurale pentru PLN se concentrează pe creșterea eficienței computaționale, scalabilitatea și capacitatea de a învăța din cantități din ce în ce mai mari de date. Modele precum GPT-3 [12] demonstrează potențialul de a genera text care este indiscernabil de cel uman, deschizând noi posibilități pentru aplicații creative și analitice în PLN.

# Gradient Descent: Fundamente și Implicații în Învățarea Automată

## Funcția de Cost

### Fundamentele Teoretice ale Funcție de Cost

Funcția de cost servește ca un indicator al performanței modelului, măsurând discrepanța dintre predicțiile modelului și valorile reale. Această discrepanță, sau eroare, orientează ajustarea parametrilor modelului în direcția optimă. "Minimizarea funcției de cost este echivalentă cu îmbunătățirea acurateței modelului nostru"¹, ceea ce reprezintă obiectivul principal în optimizarea modelelor de învățare automată.

O funcție de cost comună pentru problemele de regresie este eroarea medie pătratică:

(1.1)

este vectorul de parametri ai modelului, inclusiv interceptul cât și coeficienții pentru fiecare caracteristică

interceptul, cunoscut și sub numele de termen liber sau bias în contextul regresiei liniare, reprezintă valoarea de ieșire a funcției atunci când toate caracteristicile de intrare (x) sunt egale cu zero

m este numărul de exemple de antrenament,

este vectorul caracteristicilor pentru exemplul de antrenament i

este eticheta reală a exemplului de antrenament i

este funcția ipoteză, definită ca

(1.2)

în regresia liniară.

Împărțirea cu 2m în loc de m este o convenție utilizată uneori pentru a simplifica derivata funcției de cost atunci când se calculează gradientul, făcând calculele matematice ulterioare puțin mai directe în contextul optimizării prin metoda gradient descent.

### Graficul 2D al Funcției de Cost - Eroare Medie Pătratică

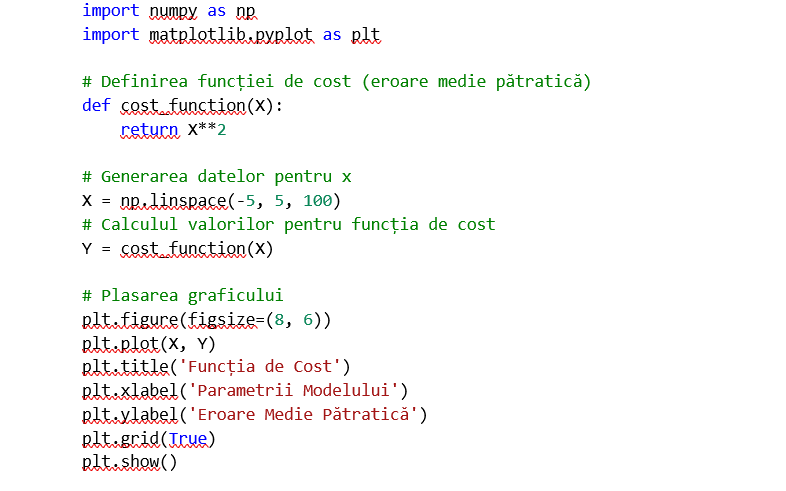


Fig. 1: Generarea 2D a Funcției de Cost în Python

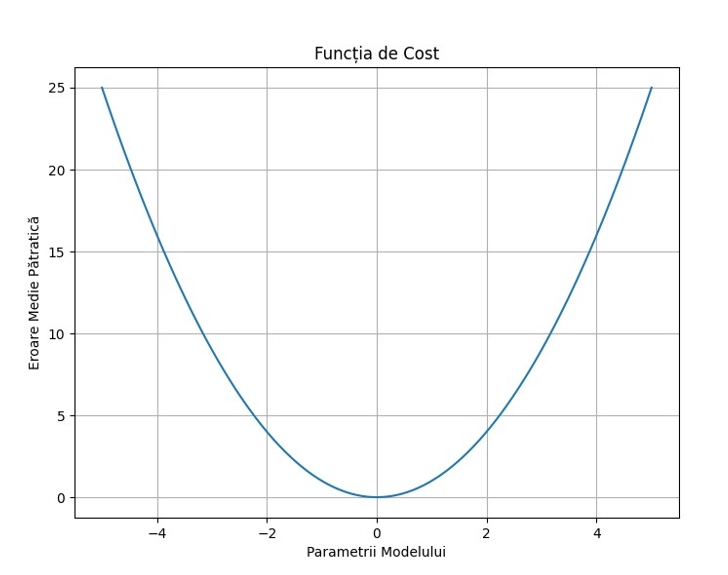
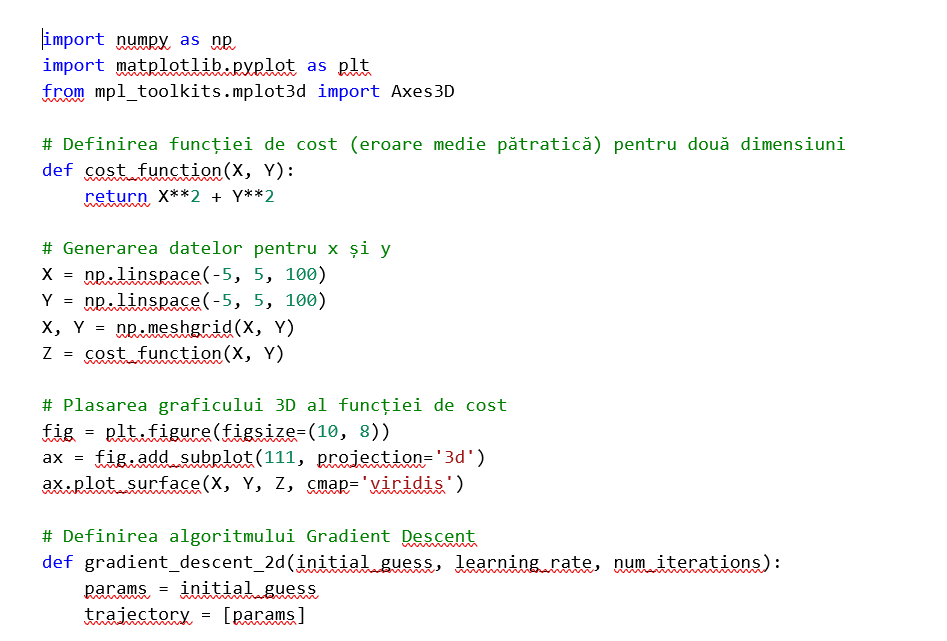
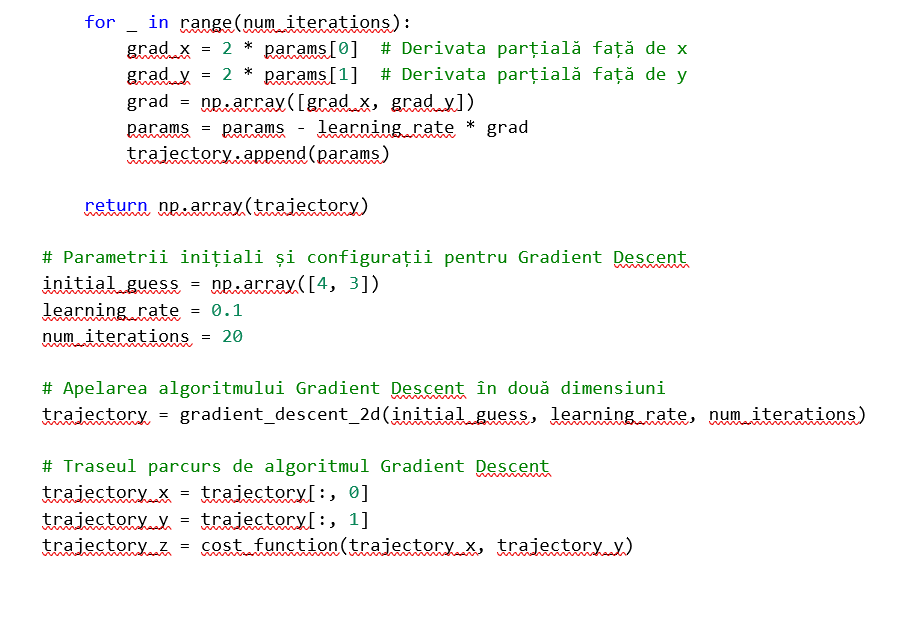


Fig. 2 Vizualizarea 2D a Funcției de Cost

### Graficul 3D al Funcției de Cost - Eroare Medie Pătratică





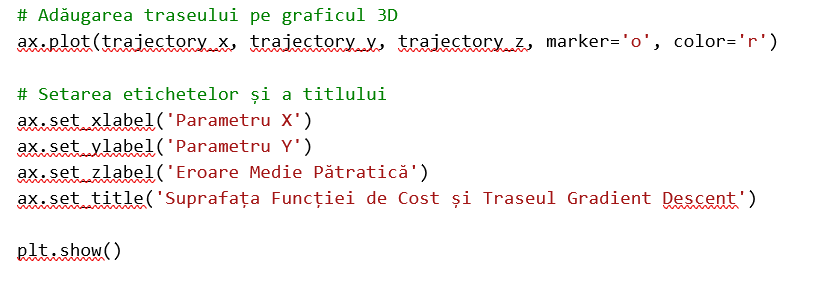


Fig. 3 Generarea 3D a Funcției de Cost în Python

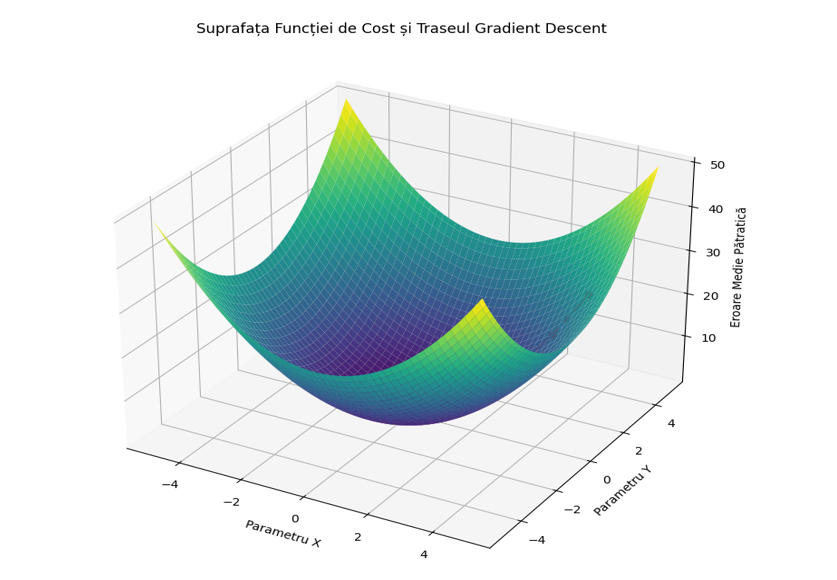


Figura 1.4: Vizualizarea 3D a Funcției de Cost

## Calculul Gradientului

### Fundamentele Teoretice ale Calculului Gradientului

Direcția și magnitudinea ajustărilor necesare parametrilor modelului sunt dictate de gradientul funcției de cost. "Identificarea direcției corecte pentru ajustarea parametrilor este esențială; calculul gradientului ne oferă această direcție"². Prin urmare, calculul eficient al gradientului este un pas crucial în iterația eficace către minimizarea erorii modelului.

***Gradientul funcției de cost pentru regresia liniară cu MSE este calculat astfel***:

(1.3)

unde:

reprezintă gradientul funcției de cost în raport cu parametrii modelului .

este numărul de exemple din setul de antrenament.

este matricea caracteristicilor, unde fiecare linie corespunde unui exemplu de antrenament și fiecare coloană corespunde unei caracteristici.

este vectorul de parametri ai modelului, inclusiv interceptul cât și coeficienții pentru fiecare caracteristică

reprezintă produsul matriceal dintre matricea caracteristicilor și vectorul parametrilor

este vectorul cu valorile reale (etichetele) corespunzătoare fiecărui exemplu de antrenament.

***Regula de actualizare:*** poate fi exprimată pentru fiecare parametru astfel:

(1.4)

reprezintă parametrul j al modelului, care poate fi fie interceptul , fie unul dintre coeficienții asociați caracteristicilor

este rata de învățare, un scalar pozitiv care controlează mărimea pasului efectuat în direcția opusă gradientului.

m este numărul total de exemple de antrenament în setul de date.

dată de

(1.5)

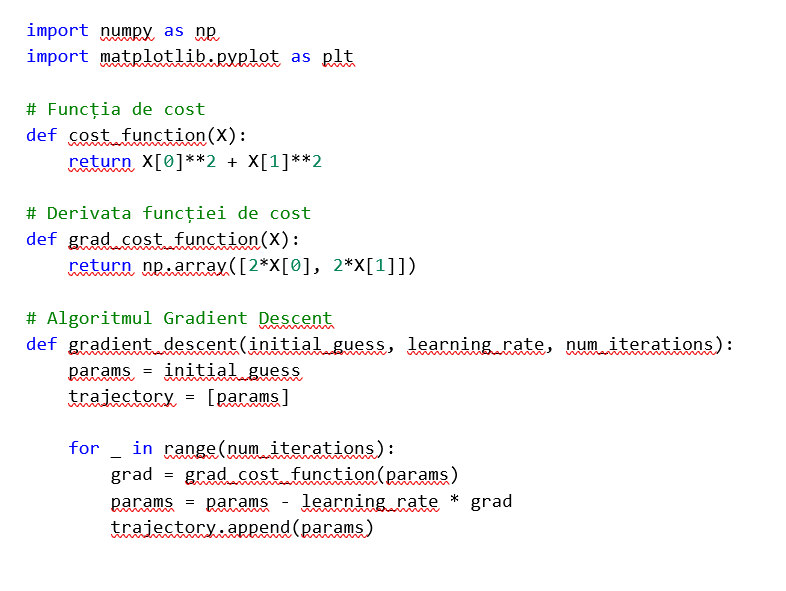
este eticheta reală a exemplului de antrenament i

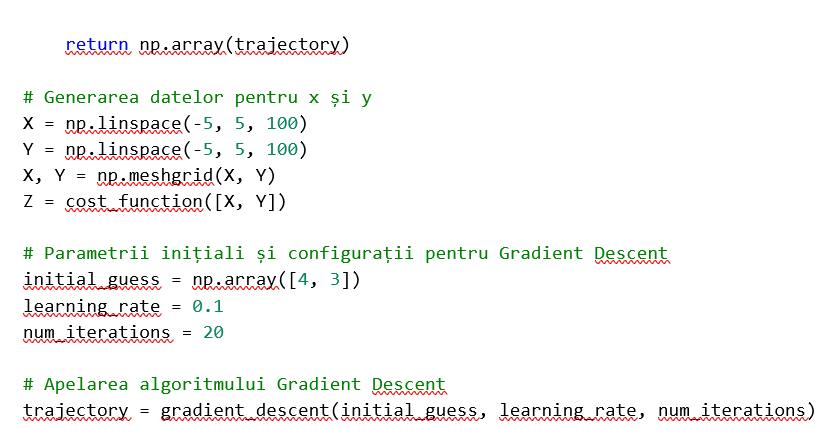
este valoarea caracteristicii j pentru exemplul de antrenament i

***Ratele de Învățare și Impactul lor asupra Convergenței***

Un aspect critic în implementarea algoritmului Gradient Descent este selecția ratei de învățare. Această rată determină mărimea pasului efectuat la fiecare iterație în direcția negativă a gradientului. "Echilibrarea ratei de învățare este esențială pentru a asigura convergența eficientă a modelului"[3]. O rată de învățare nepotrivită poate împiedica algoritmul să ajungă la soluția optimă într-un mod eficient.

### Plotarea Traseului Gradientului în Python





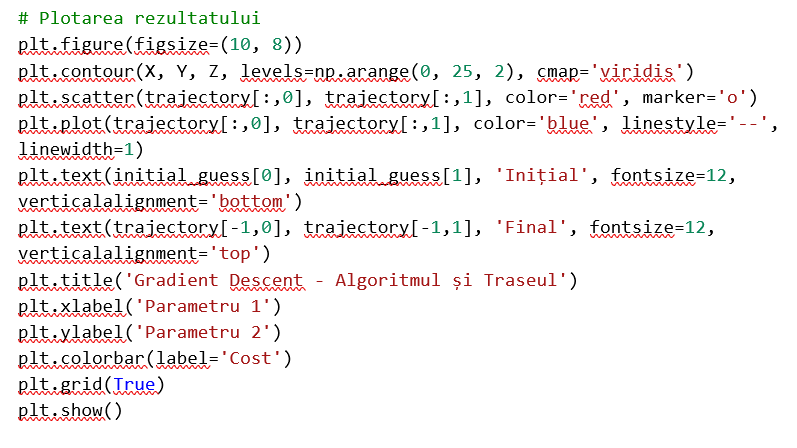


Figura 1.5: Codul Python de Generare a Traseului de Coborâre al Gradientului

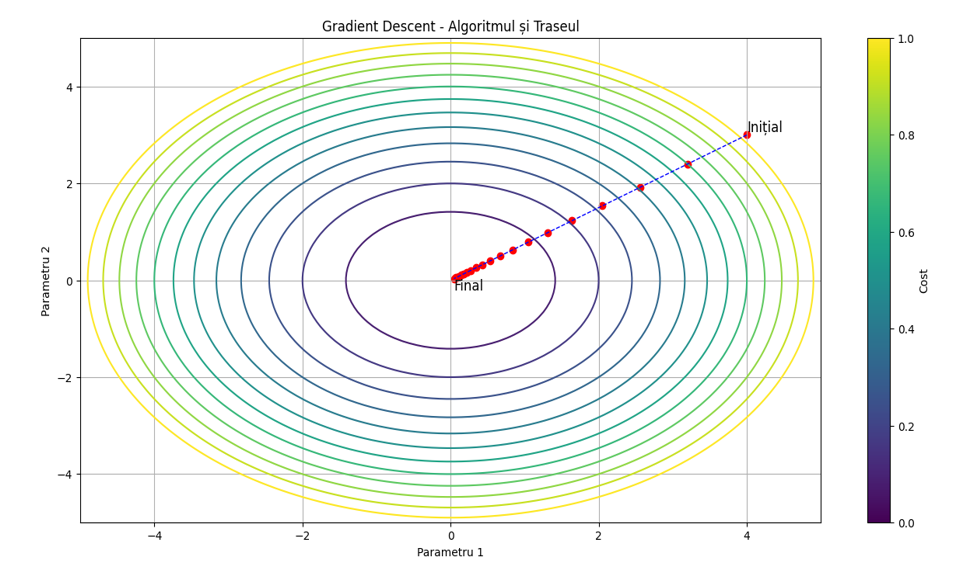


Figura 1.6: Traseului de Coborâre al Gradientului

## Variante ale Gradient Descent

### Batch Gradient Descent

Batch Gradient Descent este recunoscut pentru avantajul său semnificativ în termeni de stabilitate în procesul de optimizare, asigurând direcții de ajustare precise prin calculul gradientului pe baza întregului set de date la fiecare pas de iterație. Această metodologie, deși avantajoasă pentru precizia ajustărilor, se confruntă cu limitări majore legate de eficiența computatională, în special când este aplicată pe seturi de date de dimensiuni mari. Așa cum este evidențiat în literatura de specialitate, "costul computațional poate deveni prohibitiv pe seturi mari de date"⁴, datorită necesității de a procesa întregul volum de date în fiecare ciclu de antrenament, ceea ce poate încetini considerabil procesul de învățare.

Pentru a adresa această provocare și a îmbunătăți eficiența optimizării, cercetările în domeniu au explorat diverse strategii, inclusiv metode alternative de calcul al gradientului care să permită o convergență mai rapidă și să reducă sarcina computațională. Printre acestea, strategii precum Stochastic Gradient Descent (SGD) și Mini-batch Gradient Descent oferă un compromis între acuratețea calculului gradientului și eficiența computatională, permițând actualizări mai frecvente ale parametrilor cu o fracțiune din costul computațional asociat metodei Batch Gradient Descent.

În contextul acestor dezvoltări, lucrări recente aduc în discuție metode inovative și tehnici de optimizare avansate care își propun să optimizeze mai departe balanța dintre acuratețe și eficiență în procesele de antrenare ale modelelor de învățare automată. Aceste contribuții sunt esențiale pentru avansarea înțelegerii și aplicabilității metodelor de optimizare în diverse scenarii de învățare automată.

### Stochastic și Mini-Batch Gradient Descent

Prin contrast cu Batch Gradient Descent, Stochastic Gradient Descent (SGD) și Mini-Batch Gradient Descent reprezintă metode alternative care oferă avantaje semnificative în termeni de eficiență computatională. Aceste abordări, prin procesarea individuală a fiecărui exemplu (SGD) sau a unor mini-batch-uri de date, nu numai că accelerează procesul de convergență, dar, conform literaturii de specialitate, "îmbunătățesc eficiența computatională și pot ajuta la evitarea minimelor locale nedorite" [5]. Actualizările frecvente ale parametrilor, caracteristice acestor metode, facilitează o ajustare mai dinamică și reactivă a modelului, ceea ce poate contribui la explorarea mai eficientă a spațiului de căutare pentru optimizare.

Deși Stochastic și Mini-Batch Gradient Descent sunt eficiente în accelerarea convergenței și în reducerea riscului de a rămâne blocat în minime locale, aceste metode introduc o varianță mai mare în actualizările parametrilor. Acest aspect poate duce la o traiectorie de optimizare mai zgomotoasă, care, pe de o parte, favorizează evadarea din minimelor locale, dar, pe de altă parte, necesită ajustări fine ale ratei de învățare pentru a asigura stabilitatea convergenței [13].

Pentru a aborda aceste provocări, cercetătorii au propus diverse tehnici de adaptare a ratei de învățare și algoritmi avansați de optimizare. De exemplu, algoritmi precum Adam [14] și RMSprop [15] sunt concepuți pentru a ajusta automat rata de învățare pe parcursul procesului de antrenament, îmbunătățind stabilitatea și eficiența convergenței, chiar și în prezența varianței ridicate introduse de SGD și Mini-Batch Gradient Descent.

În plus, cercetările recente subliniază importanța tehnicilor de regularizare și a strategiilor de early stopping pentru a preveni overfitting-ul și pentru a îmbunătăți generalizabilitatea modelelor antrenate cu aceste metode de optimizare [16]. Prin implementarea acestor tehnici suplimentare, practicienii pot exploata avantajele SGD și Mini-Batch Gradient Descent, maximizând în același timp performanța și robustețea modelelor de învățare automată.

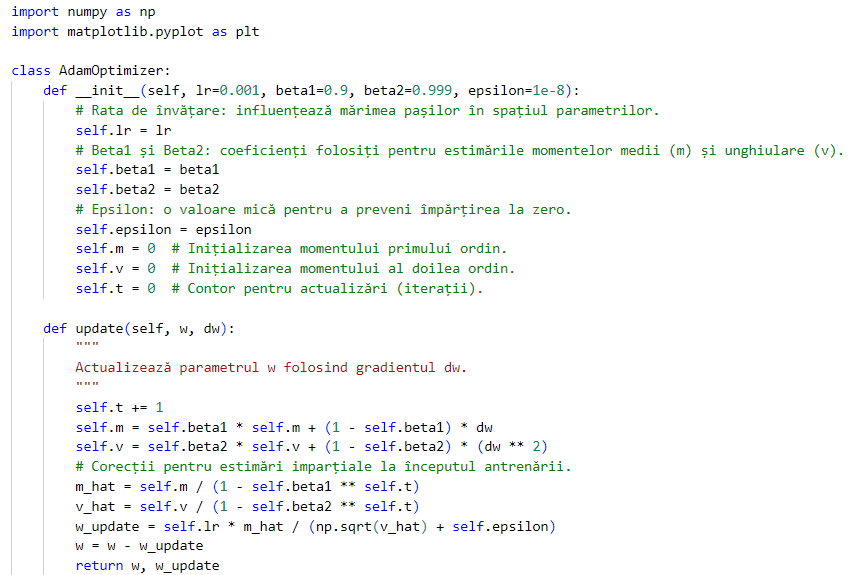
### Optimizări Adaptative în Gradient Descent

Inovațiile în domeniul optimizării algoritmilor de învățare automată, precum algoritmul Adam, marchează o evoluție semnificativă în abordarea procesului de antrenare a modelelor. Adam, un acronim pentru "Adaptive Moment Estimation", se distinge prin capacitatea sa de a realiza ajustări adaptative ale ratei de învățare pentru fiecare parametru în parte. Conform literaturii de specialitate, "Prin ajustarea dinamică a ratei de învățare, Adam optimizează convergența, abordând provocările legate de alegerea statică a ratei de învățare" [6]. Această caracteristică permite algoritmului Adam să navigheze mai eficient prin peisajul complex al funcțiilor de cost, facilitând o convergență rapidă și stabilă, chiar și în scenarii cu varianță mare a gradientului.

Algoritmul Adam și-a dovedit eficacitatea într-o gamă largă de aplicații, de la rețele neurale profunde la sisteme de procesare a limbajului natural și recunoaștere a imaginilor, devenind rapid una dintre opțiunile preferate pentru optimizarea în învățarea automată. "Eficiența sa în diferite contexte de antrenament demonstrează flexibilitatea și capacitatea de adaptare a Adam, stabilindu-l ca un standard de facto în optimizarea algoritmilor de învățare" [17].

Mai mult, cercetările recente explorează extensii și îmbunătățiri ale lui Adam, cum ar fi AdamW [18], care integrează tehnici de regularizare a ponderilor în schema de optimizare a lui Adam, adresând unele dintre criticile legate de gestionarea regularizării în contextul ajustărilor adaptative ale ratei de învățare. "AdamW îmbunătățește generalizabilitatea modelelor antrenate prin gestionarea mai eficientă a regularizării, oferind o bază solidă pentru antrenarea robustă a rețelelor neurale" [18].

Pe lângă aceste avansuri, comunitatea științifică continuă să investigheze alternative la Adam, cum ar fi algoritmi bazati pe tehnici de optimizare secundară sau abordări noi de ajustare adaptativă a ratei de învățare, care să ofere îmbunătățiri în termeni de eficiență și stabilitate a convergenței. "Explorarea continuă a spațiului de optimizare și dezvoltarea de noi metode reprezintă un domeniu activ de cercetare, cu potențialul de a descoperi tehnici încă și mai eficiente pentru antrenarea modelelor de învățare automată" [19].



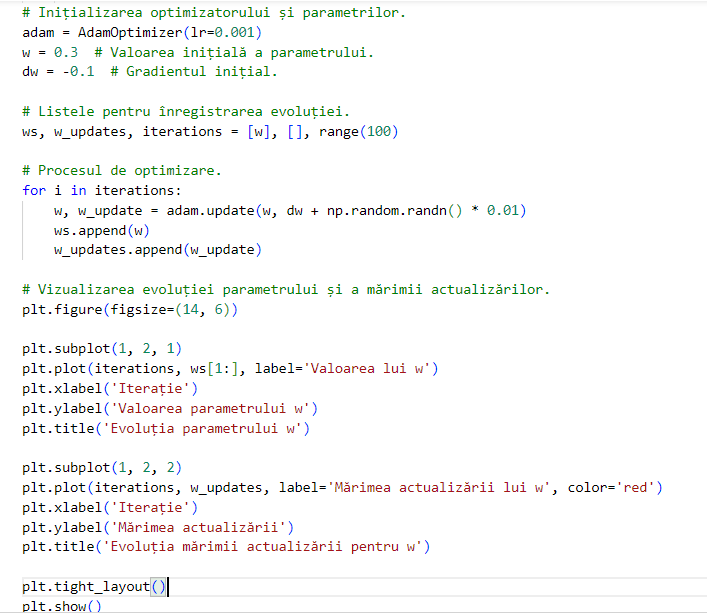


Figura 1.7: Generarea și Vizualizarea Optimizării cu Algoritmul Adam

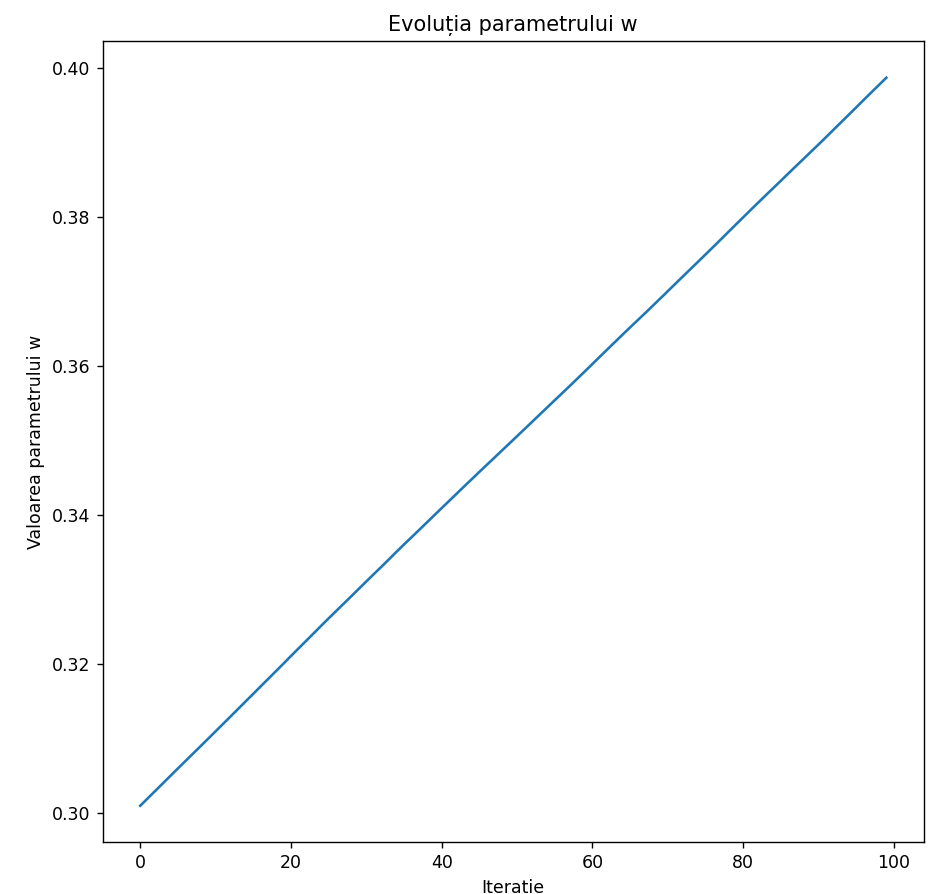


Figura 1.8: Evoluția Parametrului w în Optimizarea cu Adam

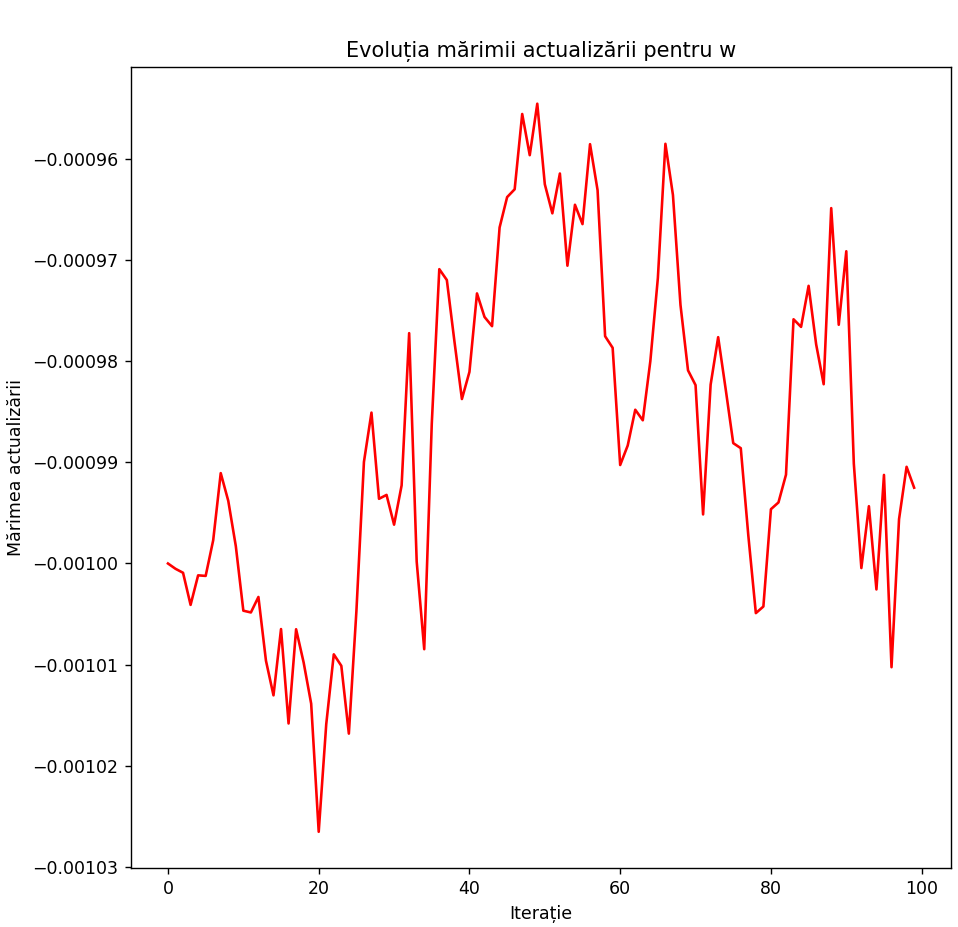


Figura 1.9: Evoluția Mărimii Actualizării pentru Parametrul w în Optimizarea cu Adam

# Implementarea Rețelei Neuronale Multi-Layer Perceptron (MLP) fără Utilizarea de Biblioteci Externe

Clasa Valoare este un element central în construcția și funcționarea rețelelor neurale definite manual, jucând rolul unei unități de bază pentru stocarea datelor și a gradientelor asociate într-un graf de calcul. Scopul său principal este de a permite calculul diferențiat automat, adică de a calcula gradientul fiecărei operații efectuate asupra datelor, ceea ce este esențial în optimizarea algoritmilor de învățare automată prin metode de gradient.

## Clasa Valoare

### Componente ale Clasei Valoare

data: Acest atribut stochează valoarea propriu-zisă a instanței Valoare. În contextul rețelelor neurale, aceasta poate fi valoarea unei intrări, valoarea unei ponderi, bias-ul unui neuron sau orice altă valoare intermediară sau finală generată în timpul propagării înainte (forward pass) a rețelei.

grad: Gradientul este derivata parțială a unei funcții de pierdere în raport cu valoarea stocată în data. În timpul propagării înapoi (backward pass), acest gradient este calculat pentru fiecare Valoare din graf, permițând ajustarea parametrilor modelului (de exemplu, ponderile și bias-urile) pentru a minimiza funcția de pierdere.

\_inapoi: Acest atribut este o funcție (inițializată ca o funcție lambda care nu face nimic) care implementează logica specifică de propagare înapoi pentru operația care a creat instanța respectivă. Când se apelează metoda inapoi(), funcția \_inapoi este executată pentru a calcula gradientul operației respective și a-l propaga înapoi prin graf.

\_precedent: Este un set de instanțe Valoare din care instanța curentă a fost derivată direct prin operații matematice. Această legătură permite construirea grafului de calcul, urmărind cum fiecare valoare este generată din valorile precedente.

\_op: Un șir de caractere care etichetează operația efectuată pentru a genera valoarea curentă (de exemplu, '+', '\*', 'tanh'). Acesta este folosit pentru scopuri de depanare și vizualizare, permițând identificarea tipului de operație care a produs o anumită valoare în graf.

eticheta: O etichetă opțională care poate fi utilizată pentru a identifica mai ușor nodurile în graf, utilă în depanare și în vizualizarea graficului de calcul.

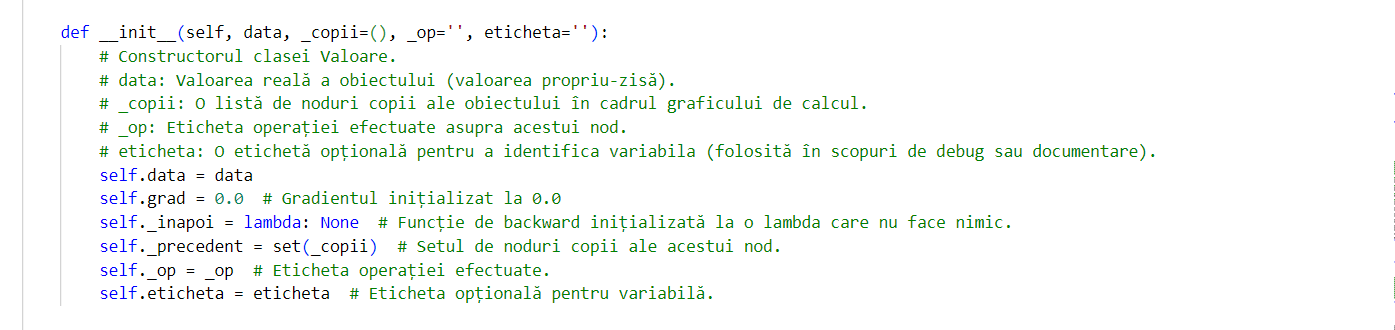
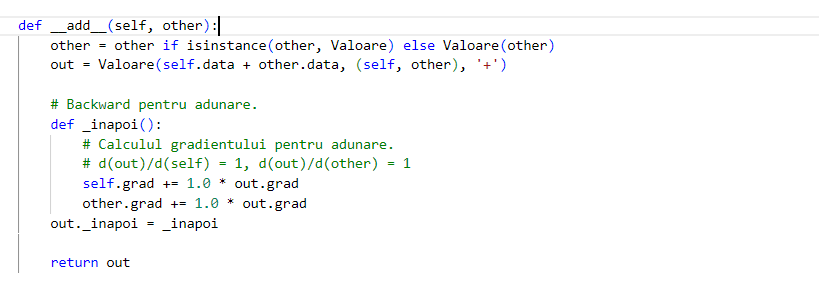
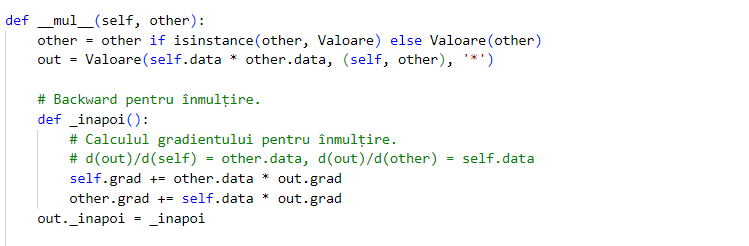


Figura 2.1: Constructorul Clasei Valoare

### Funcționalități Cheie

* Supraîncărcarea operatorilor: Clasa Valoare implementează metode speciale (\_\_add\_\_, \_\_mul\_\_, \_\_sub\_\_, \_\_truediv\_\_, \_\_pow\_\_, etc.) care permit utilizarea operatorilor matematici obișnuiți (+, \*, -, /, \*\*) în operații cu instanțele Valoare. Aceste metode creează noi instanțe Valoare care reprezintă rezultatul operației și configurează funcția \_inapoi corespunzătoare pentru fiecare din aceste operații noi.





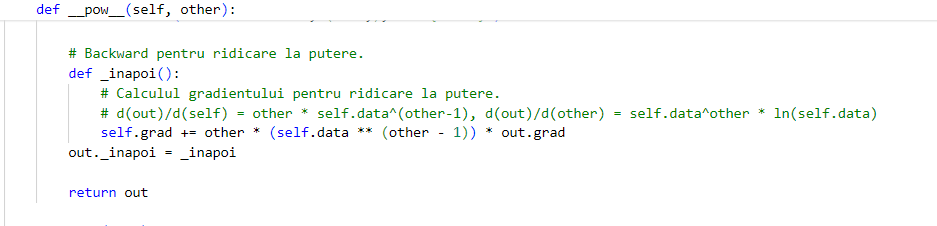


Figura 2.2: Metodele Clasei Valoare pentru Operatiile Matematice Standard și Formulele de Derivare pentru Calcularea Gradientului

* Metode de activare: Implementează funcții de activare cum ar fi tanh și exponențiala (exp), esențiale în definirea comportamentului neuronilor în rețelele neurale. Aceste metode permit aplicarea funcțiilor de activare direct pe instanțe Valoare și setează mecanismul de propagare înapoi specific fiecărei funcții.

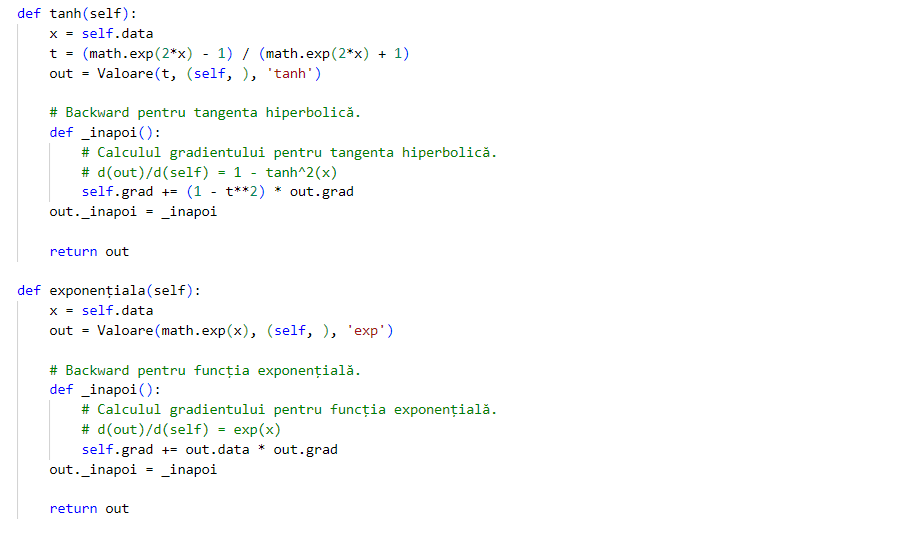


Figura 2.3: Metodele de activare

Backpropagation:

Metoda inapoi() din clasa Valoare este esențială în implementarea mecanismului de propagare înapoi (backpropagation) într-un graf de calcul. Propagarea înapoi este un proces prin care gradientul funcției de pierdere este transmis înapoi prin rețea pentru a calcula gradientul fiecărei variabile (sau parametru) în raport cu funcția de pierdere. Această metodă permite ajustarea parametrilor rețelei în direcția care reduce eroarea de predicție. Să detaliem pas cu pas funcționalitatea metodei inapoi():

1. Construirea Listei Topologice

Pasul inițial în procesul de propagare înapoi este construirea unei liste topologice a nodurilor din graf. O listă topologică a unui graf direcționat este o ordonare secvențială a nodurilor sale astfel încât pentru fiecare muchie direcționată de la nodul u la nodul v, u apare înainte de v în listă.

Funcția construieste\_topo(v): Această funcție recursivă vizitează nodurile graficului începând de la un nod dat v, marcându-le ca vizitate pentru a evita ciclurile și adăugându-le în lista topo în ordinea corectă. Recursivitatea asigură că toate nodurile anterioare (dependențele) unui nod sunt adăugate în listă înainte de nodul însuși, respectând astfel dependențele de calcul.

2. Inițierea Gradientului

După construirea listei topologice, gradientul nodului de ieșire (de obicei, valoarea pierderii) este inițiat la 1.0. Acest lucru reflectă faptul că derivata oricărei variabile în raport cu ea însăși este 1. Inițierea gradientului la 1.0 este punctul de start pentru propagarea gradientului înapoi prin graf.

3. Propagarea Gradientelor Înapoi

În acest pas, se parcurge lista topologică în ordine inversă, începând de la nodurile de ieșire (sau de la nodul de pierdere) către nodurile de intrare. Pentru fiecare nod în această ordine:

Se apelează metoda \_inapoi() specifică fiecărui nod. Această metodă calculează gradientul operației reprezentate de nod în raport cu fiecare dintre intrările sale, folosind regula lanțului. De exemplu, pentru o operație de adunare, metoda \_inapoi() va adăuga gradientul nodului curent la gradientele nodurilor sursă, deoarece derivata unei sume în raport cu oricare dintre termenii săi este 1.

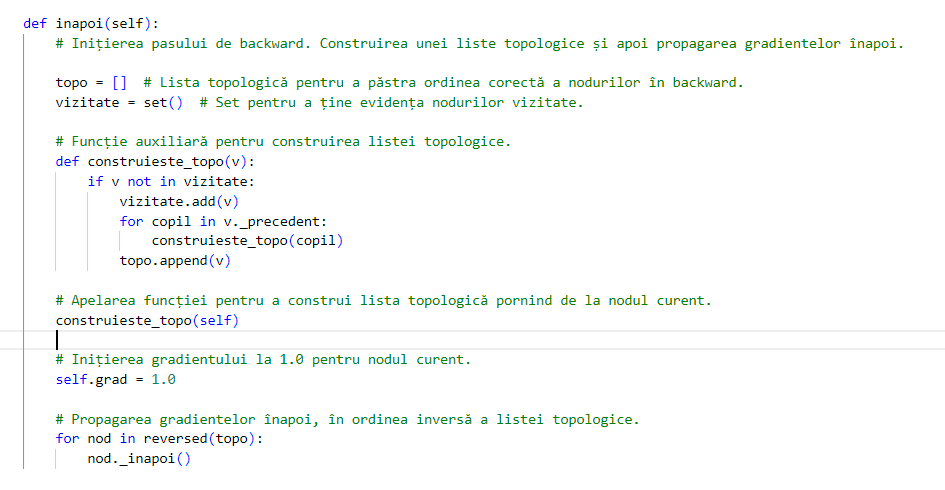


Figura 2.4: Implementarea Metodei de Backpropagation

### Utilizare în Antrenarea Rețelelor Neurale

Clasa Valoare, prin capacitatea sa de a reprezenta datele și de a calcula și propaga gradientele, este un instrument fundamental în construirea manuală a rețelelor neurale, permițând antrenarea acestora prin metode de optimizare bazate pe gradient, cum ar fi coborârea pe gradient (gradient descent). Prin utilizarea acestei clase, dezvoltatorii pot avea un control fin asupra fluxului de date și al calculului gradientului în rețea, facilitând experimentarea și înțelegerea profundă a mecanismelor de învățare automată.

## Propagarea printr-un Singur Neuron

### Etape

***Inițializare:*** În codul nostru, avem două intrări (x1 și x2), greutăți asociate (w1 și w2) și un bias (b). Acestea sunt folosite pentru a calcula o ieșire prin intermediul unei formule și apoi aplicând funcția tangenta hiperbolică (tanh) la rezultat.

***Propagarea Înainte (Forward Pass):*** Calculăm ieșirea neuronului prin înmulțirea fiecărei intrări cu greutatea sa, adunând rezultatele și adăugând bias-ul, urmat de aplicarea funcției tanh pe suma totală pentru a obține ieșirea (o).

***Inițierea Propagării Înapoi (Backpropagation):*** Propagarea înapoi începe cu setarea gradientului ieșirii la 1. Aceasta indică faptul că ne interesează cum afectează fiecare parametru direct ieșirea finală.

***Calculul Gradientilor:*** În această etapă, ne concentrăm pe calcularea modului în care fiecare greutate (w1, w2) și bias (b) influențează ieșirea și, implicit, eroarea. Folosim regulile de calcul diferențial, precum derivata funcției tanh, pentru a determina aceste influențe: Pentru fiecare greutate și bias, calculăm cât de mult trebuie ajustate pentru a minimiza eroarea. Acest lucru este făcut utilizând informațiile despre cum fiecare contribuie la calculul ieșirii.

***Rezultatul Final:*** După finalizarea propagării înapoi (executată prin o.inapoi()), fiecare parametru (w1, w2, b) va avea un gradient asociat. Acești gradienți ne indică direcția în care trebuie ajustați parametrii pentru a reduce eroarea în procesul de antrenament.



Figura 2.5: Propagarea printr-un singur neuron. Funcția de activare tanh

### Funcția de activare - tangenta hiperbolică (tanh)

Funcția tangenta hiperbolică (tanh) este o funcție de activare des utilizată în rețelele neurale. Aceasta este definită matematic ca:

(2.1)

Unde e este baza logaritmului natural, iar x este valoarea de intrare pentru funcție.

**Proprietăți:**

***Domeniu de valori:*** Ieșirea funcției tanh este întotdeauna între -1 și 1. Aceasta înseamnă că normalizează ieșirea neuronului, ceea ce poate ajuta în stabilizarea și accelerarea convergenței în timpul antrenării rețelei neurale.

***Simetrică față de origine:*** Funcția tanh este simetrică față de origine, ceea ce înseamnă că pentru orice valoare pozitivă a lui x, există o valoare negativă a lui x care produce o ieșire de magnitudine egală dar semn opus. Acest lucru permite funcției să gestioneze datele negative mai eficient decât alte funcții de activare, cum ar fi funcția sigmoidă.

Funcția tanh este folosită ca o ***funcție de activare*** pentru neuroni într-o rețea neurală. În contextul codului prezentat, aplicăm funcția tanh la suma ponderată a intrărilor și bias-ului unui neuron. Aceasta are rolul de a introduce non-linearitatea în model, permițând rețelei să învețe relații complexe între intrări și ieșiri.

***Derivata funcției tanh*** este importantă în procesul de backpropagation, deoarece ne spune cum să ajustăm greutățile și bias-ul în raport cu eroarea calculată. Derivata este dată de:

(2.2)

Această derivată are câteva proprietăți utile:

Este întotdeauna între 0 și 1, ceea ce menține gradientul într-un interval gestionabil, reducând riscul de explozie a gradientului în timpul antrenării.

Permite calculul eficient al gradientului necesar pentru ajustarea parametrilor rețelei.

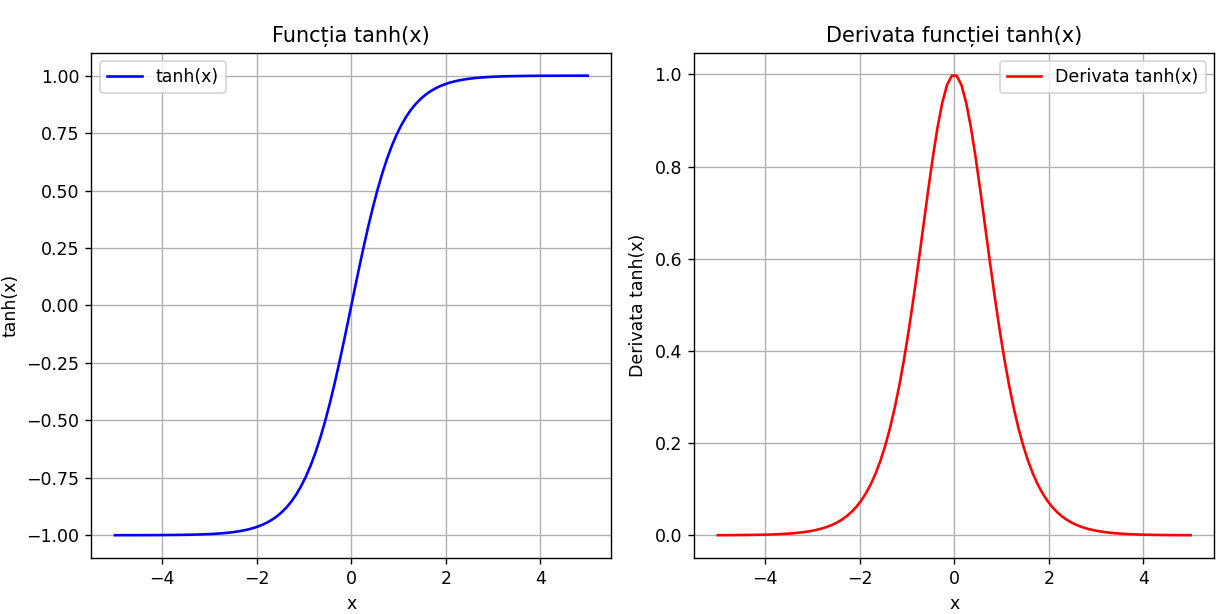


Figura 2.6: Graficele funcției tanh și a derivatei acesteia

### Perceptronul lui Rosenblatt

***Originea Perceptronului:***

În 1957, Frank Rosenblatt a introdus conceptul de perceptron, considerat una dintre primele încercări de a modela procesul de gândire umană prin intermediul mașinilor. Perceptronul lui Rosenblatt a fost un pas semnificativ în dezvoltarea inteligenței artificiale și învățării automate, prezentând o arhitectură simplă capabilă să efectueze clasificări binare. În lucrarea sa seminală, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain" (Rosenblatt, 1958), el descrie fundamentul teoretic pentru acest model, subliniind capacitatea sa de a învăța și de a face predicții bazate pe datele de intrare.

***De la Perceptron la Rețele Neuronale Moderne:***

Perceptronul lui Rosenblatt a fost conceput să simuleze un neuron biologic, având intrări (sinapse), greutăți (forta sinaptică), o funcție de activare (procesul de decizie al neuronului) și o ieșire (impulsul trimis de neuron). Această structură simplă reflectă principiile de bază ale rețelelor neuronale moderne, în care multiplele straturi de neuroni artificiali colaborează pentru a procesa și a învăța din date complexe.

***Funcția de Activare Tanh: O Legătură între Trecut și Prezent***

În codul analizat anterior, utilizăm funcția de activare tanh pentru a introduce non-linearitate în model, o evoluție față de funcția de activare inițială a perceptronului, care era o funcție treaptă. Acest avans subliniază progresul în înțelegerea și modelarea proceselor neuronale complexe. Funcția tanh, cu ieșirea sa variind între -1 și 1, oferă o normalizare naturală și o distribuție centrată a activărilor, îmbunătățind astfel procesul de antrenare al rețelelor neuronale profunde prin atenuarea problemelor precum dispariția gradientului.

***Propagarea Înapoi: De la Teorie la Practică***

Metoda de backpropagation, esențială pentru antrenarea rețelelor neuronale moderne, nu era prezentă în modelul original al perceptronului. Această tehnică, care calculează gradientul funcției de eroare în raport cu toți parametrii rețelei (greutăți și bias-uri), permite ajustarea fină a modelului în direcția minimizării erorii. Astfel, backpropagation reprezintă o piatră de temelie în evoluția rețelelor neuronale, facilitând tranziția de la modele simple, lineare, la sisteme complexe, capabile să înțeleagă și să proceseze limbajul natural, imagini și multe alte tipuri de date structurate și nestructurate.

Reflectând asupra moștenirii lăsate de Rosenblatt, este evident că principiile fundamentale stabilite de perceptronul său continuă să influențeze dezvoltarea rețelelor neuronale moderne. Integrarea funcțiilor de activare avansate, cum ar fi tanh, și implementarea tehnicilor de optimizare, precum backpropagation, demonstrează cum viziunea inițială a evoluat în sisteme complexe de învățare automată. Această evoluție subliniază importanța continuării cercetării și inovării în domeniul inteligenței artificiale, inspirându-ne să construim pe fundația solidă a pionierilor precum Rosenblatt și să explorăm noi orizonturi în modelarea proceselor cognitive umane.

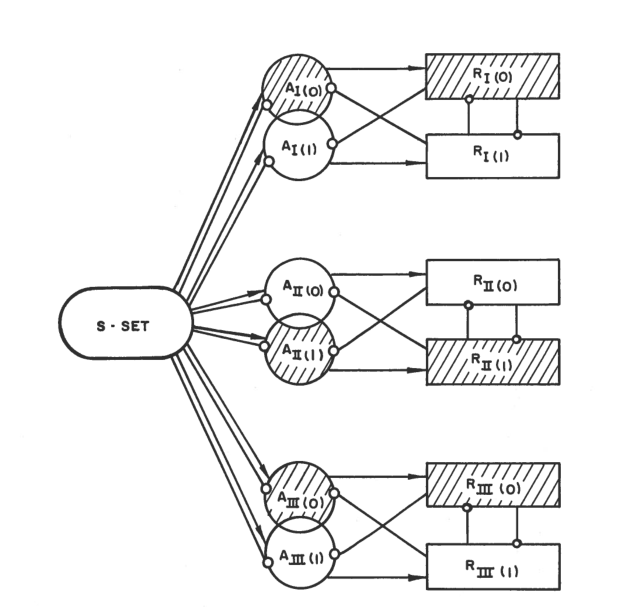


Figura 2.7: Organizarea unui Perceptron cu Trei Seturi de Raspunsuri Binare, Frank Rosenblatt 1957

## Arhitectura Rețelelor Neuronale: Neuron, Layer, MLP

### Clasa neuron

În centrul rețelelor neuronale stă conceptul de neuron artificial, o unitate de calcul inspirată de neurobiologia umană. Conform Goodfellow, Bengio și Courville (2016)[1], un neuron primește intrări, le înmulțește cu greutăți, adună un bias și aplică o funcție de activare la suma acestora pentru a produce o ieșire. Codul prezentat definește clasa Neuron, care encapsulează această logică fundamentală. Fiecare neuron inițializează greutățile și bias-ul cu valori aleatorii, reflectând necesitatea inițializării în rețelele neuronale discutată de LeCun et al. (1998)[4]. Funcția de activare tanh, utilizată în această implementare, este esențială pentru introducerea non-linearității, permițând modelului să învețe relații complexe între intrări și ieșiri.

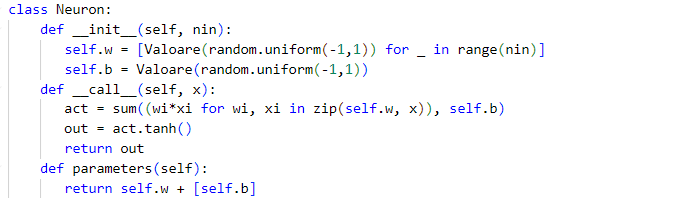


Figura 2.8: Definirea Clasei Neuron

***Inițializarea:*** Constructorul clasei Neuron primește un parametru, nin, care indică numărul de intrări (sau caracteristici) pe care neuronul le va procesa. În interiorul constructorului, greutățile (self.w) și bias-ul (self.b) sunt inițializate aleatoriu în intervalul [−1,1], urmând recomandările generale pentru inițializarea parametrilor în rețelele neuronale. Inițializarea aleatorie este crucială pentru a sparge simetria și a permite rețelei să învețe diverse funcții.

***Apelarea:*** Metoda „magică” de apelare transformă instanța clasei Neuron într-un obiect apelabil, permițând utilizarea instanței ca pe o funcție. Aceasta primește un set de intrări x și calculează activarea neuronului. În primul rând, se efectuează suma ponderată a intrărilor și greutăților, la care se adaugă bias-ul. Rezultatul acestei sume ponderate este apoi transmis prin funcția de activare tanh, producând ieșirea neuronului. Acest proces este esența calculului efectuat de un neuron: combinarea liniară a intrărilor urmată de aplicarea non-linearității.

***Parametrii (parameters):*** Metoda parameters oferă acces la parametrii neuronului (greutățile și bias-ul) sub formă de listă. Aceasta este utilă pentru procesele de antrenare și optimizare, permițând algoritmilor de ajustare a parametrilor să acceseze și să modifice greutățile și bias-ul neuronului.

***Implicații și Utilizări:*** Implementarea clasei Neuron ilustrează principiile fundamentale ale calculului neuronal și pune bazele pentru construcția de rețele neuronale mai complexe. Prin agregarea mai multor instanțe ale clasei Neuron în structuri mai mari, precum straturi de neuroni și rețele întregi, putem modela funcții matematice complexe capabile să aproximeze o gamă largă de relații între datele de intrare și ieșire. Această modularitate și flexibilitate fac din rețelele neuronale o unealtă puternică în domeniul inteligenței artificiale și învățării automate, cu aplicații variind de la recunoașterea imaginilor și a vorbirii până la generarea de text și dincolo de acestea.

### Stratul Neuronal

Extinderea de la un singur neuron la un strat de neuroni este un pas crucial în construcția rețelelor neuronale profunde. Clasa Layer, așa cum este definită în cod, reprezintă acest concept prin agregarea mai multor neuroni care operează în paralel. Această arhitectură permite rețelei să extragă și să combine caracteristici la diferite niveluri de abstracție. LeCun, Bengio și Hinton (2015)[2] subliniază importanța straturilor multiple în învățarea reprezentărilor de date ierarhice, o proprietate fundamentală a rețelelor neuronale profunde.

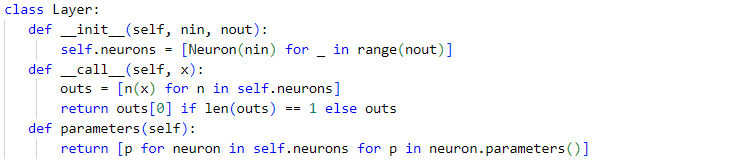


Figura 2.9: Definirea Clasei Layer

Clasa Layer reprezintă un strat dintr-o rețea neurală, oferind o abstracție pentru gestionarea unui grup de neuroni care lucrează împreună pentru a procesa informații. Să detaliem fiecare parte a implementării și rolul acesteia în contextul unei rețele neuronale:

Constructorul - Inițializarea Straturilor: Constructorul clasei Layer primește două argumente: nin, numărul de intrări pentru fiecare neuron din strat, și nout, numărul de neuroni din strat. Această structură permite definirea dimensiunii și complexității stratului în funcție de arhitectura rețelei neuronale dorită.

Crearea Neuronilor: În interiorul constructorului, se creează o listă de neuroni (self.neurons), fiecare inițializat cu numărul de intrări specificat (nin). Numărul de neuroni creat este egal cu nout, asigurându-se astfel că stratul are dimensiunea dorită. Această inițializare colectivă ilustrează modularitatea și scalabilitatea rețelelor neuronale, permițând construcția de straturi adaptate la diverse sarcini de procesare.

Metoda Apelabilă: Procesarea Intrărilor: Metoda permite ca instanța Layer să fie utilizată ca o funcție, care primește un set de intrări x și calculează ieșirile pentru întregul strat. Acest comportament facilitează integrarea ușoară a straturilor în arhitectura rețelei, permitând datele să fie propagate prin rețea într-o manieră clară și concisă.

Calculul Ieșirilor: Pentru fiecare neuron din strat, se calculează ieșirea pe baza intrărilor x. Rezultatul este o listă de ieșiri (outs), câte una pentru fiecare neuron. Dacă stratul conține un singur neuron, se returnează direct ieșirea acestuia; altfel, se returnează lista de ieșiri. Aceasta reflectă flexibilitatea straturilor de a gestiona atât situații în care este necesară o singură ieșire, cât și pe cele în care sunt necesare multiple ieșiri paralele.

Accesarea Parametrilor (parameters): Metoda parameters extrage și returnează o listă a tuturor parametrilor (greutăți și bias-uri) din strat. Aceasta este realizată prin agregarea parametrilor fiecărui neuron din self.neurons, demonstrând modul în care rețelele neuronale pot fi ajustate și optimizate la nivel granular.

Utilizare în Antrenare: Accesul la parametrii stratului este crucial pentru procesele de antrenare și optimizare, deoarece permite ajustarea fină a greutăților și bias-urilor în funcție de feedback-ul primit prin algoritmi de învățare, cum ar fi backpropagation.

Clasa Layer encapsulează funcționalitatea esențială a unui strat dintr-o rețea neurală, oferind un cadru pentru definirea și manipularea grupurilor de neuroni. Prin abstractizarea gestionării multiplelor unități de calcul și facilitarea propagării datelor, această implementare subliniază principiile de modularitate și reutilizabilitate care stau la baza designului eficient al rețelelor neuronale. Astfel, clasa Layer constituie un bloc de construcție fundamental în dezvoltarea de arhitecturi neuronale complexe, permițând cercetătorilor și inginerilor să experimenteze și să inoveze în domeniul vast al inteligenței artificiale.

### Construcția și Optimizarea Perceptronului Multistrat (MLP)

Perceptronul Multistrat (MLP) constituie o piatră de temelie în domeniul rețelelor neuronale, oferind o structură capabilă să aproximeze orice funcție continuă, ceea ce îl face extrem de versatil pentru o gamă largă de probleme de învățare automată.

***Principii de Bază și Evoluție:***

MLP este o extensie a perceptronului simplu, introdus de Rosenblatt în anii '50, care a demonstrat capacitatea de a învăța decizii liniare. Cu toate acestea, limitările perceptronului simplu în reprezentarea funcțiilor non-lineare au condus la dezvoltarea MLP, care, prin adăugarea unuia sau mai multor straturi ascunse și utilizarea funcțiilor de activare non-lineare, poate captura complexitatea în datele de intrare. "Deep Learning" de Goodfellow, Bengio și Courville (2016)[1] explică în detaliu fundamentul teoretic al acestui aspect, subliniind capacitatea MLP de a modela complexitatea printr-o compoziție ierarhică de funcții.

***Importanța Funcțiilor de Activare***

Funcțiile de activare, cum ar fi sigmoid, tanh și ReLU, joacă un rol crucial în adăugarea non-linearității în MLP, permițând rețelei să învețe relații complexe. LeCun, Bengio și Hinton (2015)[2] discută despre importanța alegerii adecvate a funcției de activare pentru îmbunătățirea eficienței antrenării și a performanței modelului.

***Optimizarea și Antrenarea MLP***

Antrenarea MLP implică ajustarea ponderilor și bias-urilor prin algoritmi de optimizare, cum ar fi SGD (Stochastic Gradient Descent) și variațiile sale, inclusiv Adam, descrise de Kingma și Ba (2014)[14]. Acești algoritmi utilizează backpropagation, o metodă de calcul a gradientului funcției de pierdere în raport cu fiecare parametru, pentru a actualiza ponderile în direcția care minimizează eroarea. "Efficient BackProp" de LeCun et al. (1998)[4] oferă o analiză a tehnicilor de optimizare a procesului de antrenare, evidențiind strategii pentru îmbunătățirea convergenței.

MLP reprezintă o avansare semnificativă față de modelele inițiale de învățare automată, oferind o structură capabilă să captureze și să modeleze complexitatea într-o varietate de seturi de date. Implementarea sa, de la principiile teoretice la codul practic, reflectă o evoluție continuă în domeniul inteligenței artificiale, deschizând drumul către dezvoltarea de modele și mai sofisticate și eficiente.

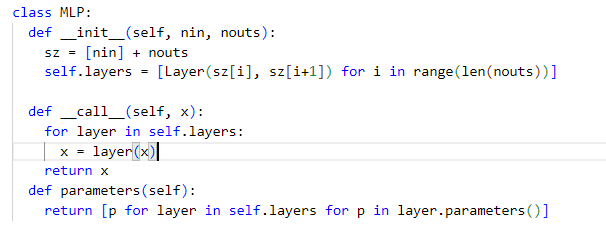


Figura 2.10: Definirea Clasei MLP

Această secțiune de cod definește clasa MLP (Multilayer Perceptron), care reprezintă o arhitectură de bază pentru rețelele neuronale profunde. Să examinăm fiecare parte a codului și funcționalitatea sa specifică:

Inițializarea MLP: Constructorul clasei MLP primește doi parametri: nin, numărul de intrări ale rețelei, și nouts, o listă care definește numărul de neuroni în fiecare strat ascuns. Acest design permite crearea unei rețele cu o arhitectură flexibilă și adaptabilă la diferite tipuri de probleme de învățare automată.

Construirea Straturilor: În interiorul constructorului, se construiește lista self.layers, care conține obiecte de tip Layer. Fiecare Layer este inițializat cu dimensiunea corespunzătoare, derivată din lista sz, care este o concatenare între numărul de intrări și dimensiunile straturilor ascunse specificate în nouts. Aceasta asigură că fiecare strat este conectat corect cu următorul, într-o manieră secvențială.

Procesarea Intrărilor: Metoda \_\_call\_\_ permite ca instanța MLP să fie utilizată ca o funcție, care primește un set de intrări x și le propagă prin rețea. Intrările sunt transmise secvențial prin fiecare strat, fiind procesate de neuroni conform ponderilor și bias-urilor lor, precum și funcției de activare specificate în Layer.

Generarea Ieșirii: Pe măsură ce datele sunt propagate prin straturi, ieșirea fiecărui strat devine intrarea pentru stratul următor, până când sunt procesate de ultimul strat. Ieșirea finală a rețelei este returnată ca rezultat al apelului funcției.

Extragerea Parametrilor: Metoda parameters colectează și returnează toți parametrii rețelei, inclusiv greutățile și bias-urile din fiecare strat. Această funcționalitate este vitală pentru procesul de antrenare și optimizare, permițând ajustarea parametrilor rețelei în funcție de feedback-ul primit prin algoritmi de învățare supervizată, cum ar fi backpropagation.

Implementarea MLP în acest mod evidențiază modularitatea și scalabilitatea rețelelor neuronale. Prin utilizarea obiectelor Layer pentru a construi rețeaua, codul rămâne clar, concis și ușor de extins sau modificat pentru a experimenta cu diferite arhitecturi. Aceasta facilitează cercetarea și dezvoltarea în domeniul inteligenței artificiale, oferind un cadru robust pentru explorarea și îmbunătățirea continuă a modelelor de învățare automată.

### Antrenarea MLP

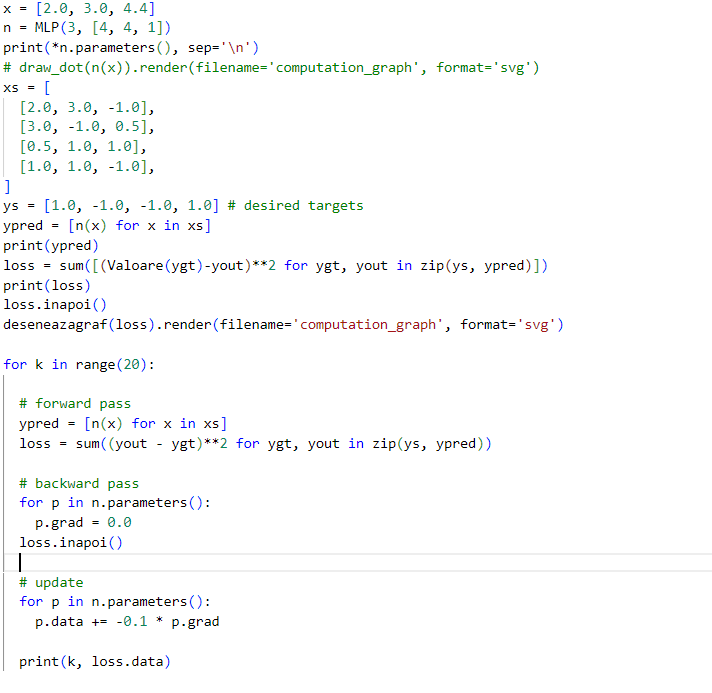


Figura 2.11: Antrenarea modelului

Inițializarea Modelului MLP: Modelul MLP este inițializat cu un număr specific de intrări (nin = 3) și o structură definită de straturi ([4, 4, 1]), ceea ce înseamnă că modelul va avea două straturi ascunse cu câte 4 neuroni fiecare și un strat de ieșire cu un singur neuron. Fiecare Layer în MLP este construit folosind clasa Neuron, care inițializează greutățile și bias-ul aleatoriu pentru fiecare neuron. Această inițializare aleatorie este crucială pentru a permite rețelei să înceapă procesul de învățare, asigurând că simetria inițială este spartă și că fiecare neuron învață un set diferit de caracteristici.

Propagarea Înainte (Forward Pass): În cadrul metodei \_\_call\_\_, intrările x sunt procesate secvențial prin straturile rețelei. Pentru fiecare strat, intrările sunt înmulțite cu greutățile neuronilor, se adaugă bias-ul și se aplică funcția de activare tanh. Această secvență de operații transformă datele de intrare într-un mod non-linear, permițând rețelei să învețe reprezentări complexe ale datelor. Propagarea înainte se încheie odată ce toate straturile au procesat datele, producând o ieșire finală a rețelei.

Calculul Funcției de Pierdere: Funcția de pierdere MSE este utilizată pentru a evalua performanța modelului comparând predicțiile acestuia (ypred) cu valorile reale (ys). MSE calculează media pătratelor diferențelor între predicții și valorile reale, oferind o măsură a erorii modelului pe setul de date de antrenament. Această funcție de pierdere este sensibilă la abaterile mari, punând un accent mai mare pe erorile mari, ceea ce face ca modelul să fie penalizat mai puternic pentru acestea.

Backward Pass și Actualizarea Parametrilor: După calculul pierderii, se inițiază backward pass-ul prin apelarea metodei loss.inapoi(). Această metodă calculează gradientul funcției de pierdere în raport cu fiecare parametru al modelului (greutăți și bias-uri), utilizând derivata funcției de activare și regula lanțului pentru a propaga gradientul înapoi prin rețea. Acest proces determină cum trebuie ajustați fiecare parametru pentru a reduce eroarea modelului.

Odată calculați gradientii, parametrii rețelei sunt actualizați în direcția negativă a gradientului, cu scopul de a minimiza funcția de pierdere. Rata de învățare specificată (de exemplu, 0.1) controlează mărimea pașilor făcuți în spațiul parametrilor la fiecare actualizare, echilibrând între rapiditatea de convergență și riscul de a "sări" peste minimele globale sau locale ale funcției de pierdere.

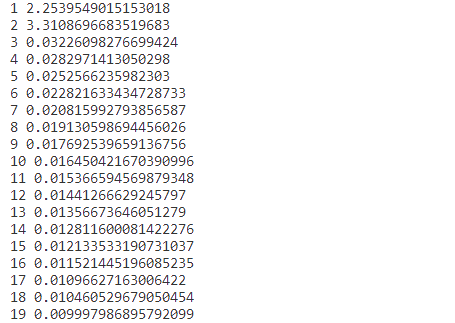


Figura 2.12: Scăderea funcției loss

Aceste valori reprezintă pierderea modelului MLP la fiecare iterație a procesului de antrenare pentru 20 de epoci. Observăm cum pierderea (loss) scade de la o iterație la alta, ceea ce indică faptul că modelul își îmbunătățește treptat performanța în a prezice corect ieșirile bazate pe intrările date. Acest proces de minimizare a pierderii este esența învățării supervizate în rețelele neuronale.

***Analiză și Interpretare***

Îmbunătățirea Rapidă Inițială: La primele iterații, se observă o reducere semnificativă a pierderii. Aceasta este tipică pentru faza inițială a antrenării, când modelul, pornind de la parametri aleatorii, face ajustări mari care conduc la îmbunătățiri rapide.

Diminuarea Graduală a Pierderii: Pe măsură ce antrenarea progresează, ratele de îmbunătățire ale modelului devin mai mici. Aceasta indică faptul că modelul începe să se stabilizeze și să se apropie de un minim al funcției de pierdere.

Convergența Modelului: Reducerea treptată și stabilizarea valorilor pierderii sugerează că modelul converge către o soluție optimă (sau cel puțin local optimă) pentru setul de date de antrenament. Convergența este un indicator că modelul și-a ajustat ponderile într-o manieră care îi permite să facă predicții mai precise.

Analiza Fină a Performanței: Deși reducerea pierderii este un semn pozitiv, este esențial să se evalueze modelul și pe baza altor metrici, cum ar fi acuratețea, pentru a înțelege mai bine performanța sa pe date nevăzute. În plus, este important să se verifice existența supraajustării (overfitting), situație în care modelul performează bine pe datele de antrenament, dar nu generalizează eficient pe date noi.

***Concluzii***

Scăderea consistentă a funcției de pierdere în timpul antrenării indică faptul că procesul de antrenare funcționează corect, optimizând parametrii modelului pentru a minimiza eroarea de predicție. Cu toate acestea, evaluarea finală a modelului trebuie să includă teste pe un set de date de validare independent pentru a asigura că modelul a învățat relații generalizabile, nu doar particularitățile setului de antrenament. Aceste rezultate oferă o bază solidă pentru iterații ulterioare de îmbunătățire și ajustare fină a modelului.

# Evoluția și Impactul Modelării Lingvistice Neuronale în Prelucrarea Limbajului Natural

## Fundamentele Modelării Lingvistice

Modelarea lingvistică, fundamentul teoretic și aplicativ al prelucrării limbajului natural, se ocupă cu analiza probabilistică și predictivă a limbii. Aceasta vizează construirea de modele matematice capabile să anticipeze probabilitățile secvențelor de cuvinte sau simboluri, deschizând calea către o înțelegere profundă și generarea de limbaj de către sisteme informatice.

Jurafsky și Martin definesc acest domeniu ca fiind procesul de creare a modelelor matematice pentru a înțelege și genera limbajul uman pe baza probabilităților [20]. Acest proces nu se limitează doar la identificarea secvențelor de cuvinte frecvente, ci include și interpretarea contextelor și semnificațiilor subtile ale limbajului.

Modelele n-gram, fundamentale în acest context, evaluează probabilitatea apariției unui cuvânt în funcție de prezența sa după o secvență de n−1 cuvinte. Chelba și colaboratorii săi ilustrează importanța acestor modele în PLN, subliniind că acestea oferă o bază probabilistică pentru predicția secvențelor de cuvinte [21]. Totuși, modelele n-gram se confruntă cu limitări, precum gestionarea datelor rare și dependențele pe termen lung.

Cercetările recente în PLN au migrat către utilizarea rețelelor neuronale și a modelelor de deep learning pentru a aborda aceste provocări. Rețelele neuronale permit modelarea relațiilor complexe din datele lingvistice, ajungând la un nivel de profunzime neexplorat anterior [1]. Prin adoptarea reprezentărilor distribuite și arhitecturilor avansate, cum ar fi RNN și transformatorii, modelele actuale prezic secvențe textuale cu o acuratețe înaltă.

### Evoluția Istorică și Contextuală, de la Modelele Bigram la Rețele Neuronale Complexe

Modelele lingvistice inițiale, fundamentate pe paradigme statistice, au reprezentat piatra de temelie a prelucrării automate a limbajului. Aceste modele, în special cele bazate pe n-grame, au furnizat o metodologie pentru analiza și predicția structurilor lingvistice prin calculul probabilităților secvențelor de cuvinte. Jurafsky și Martin subliniază că, deși modelele n-gram au constituit un progres semnificativ în înțelegerea limbajului, ele sunt limitate de spațiul dimensional vast și de dispersia datelor, ceea ce le afectează capacitatea de extrapolare și generalizare [20].

***Modelele bigram*** reprezintă una dintre cele mai timpurii și fundamentale abordări în modelarea limbajului natural, bazându-se pe presupunerea că probabilitatea apariției unui caracter sau a unui cuvânt poate fi estimată prin considerarea imediat predecesorului său. Această abordare simplă, dar eficace, pune bazele pentru înțelegerea relațiilor dintre unitățile lingvistice și a fost esențială în dezvoltarea ulterioară a modelelor de limbaj mai complexe.

Un principiu de bază al modelelor bigram este descris astfel: fiecare element dintr-o secvență este considerat a fi dependent doar de elementul precedent, ceea ce simplifică semnificativ calculul probabilităților pentru lanțuri lungi de text. Această presupunere de independență condiționată permite modelarea limbajului cu un set de date relativ mic și cu calcule simplificate, facilitând înțelegerea fenomenelor lingvistice fără necesitatea unor resurse computaționale extinse.

În lucrarea sa fundamentală, Markov (1913) a introdus conceptul de lanțuri, care stau la baza modelelor bigram: "Aplicarea lanțurilor Markov în procesarea limbajului a deschis calea către modelarea statistică a limbajului, permițând estimări bazate pe probabilitate pentru secvențe de cuvinte sau caractere" [26]. Această metodologie a fost adoptată și extinsă în numeroase aplicații ale procesării limbajului natural, demonstrând utilitatea sa în înțelegerea structurii și dinamicii limbajului.

Deși modelele bigram sunt limitate prin natura lor simplistă și prin presupunerea că dependențele lingvistice se extind doar la elementul imediat precedent, ele au pavat drumul pentru dezvoltări ulterioare în modelarea limbajului. În ciuda limitărilor, modelele bigram rămân un instrument valoros în studiul inițial al dependențelor lingvistice și au influențat dezvoltarea modelelor de limbaj mai sofisticate.

În căutarea soluțiilor la aceste limitări, cercetătorii au început să exploreze alte modele statistice și probabilistice, cum ar fi modelele Markov ascunse (HMM) și modelele de spațiu vectorial pentru reprezentarea semantică a cuvintelor. Aceste abordări au permis o înțelegere mai nuanțată a limbajului, oferind totodată o flexibilitate mai mare în modelarea dependențelor pe termen lung între cuvinte.

Cu toate acestea, chiar și aceste modele avansate au întâmpinat dificultăți în captarea complexităților și subtilităților limbajului uman, cum ar fi ambiguitatea, polisemia și relațiile sintactice și semantice profunde dintre cuvinte. Această conștientizare a condus la dezvoltarea și adoptarea tehnologiilor de deep learning în PLN, marcând o schimbare paradigmatică în domeniu.

### MLP în Procesarea Limbajului Natural

După modelele bigram, care au pus bazele înțelegerii secvențelor de caractere sau cuvinte printr-o abordare relativ simplistă, evoluția tehnologică în procesarea limbajului natural a făcut un salt semnificativ cu introducerea perceptronilor multi-strat (MLP). Aceste modele au deschis calea către analiza și modelarea relațiilor mult mai complexe dintre elementele limbajului, marcând o etapă crucială în dezvoltarea inteligenței artificiale și a înțelegerii profunde a structurilor limbajului natural.

"Arhitectura MLP, prin straturile sale ascunse, a oferit un cadru robust pentru înțelegerea și modelarea complexităților inerente ale limbajului, un progres semnificativ față de modelele anterioare" [2]. Această capacitate de a învăța reprezentări la diferite niveluri de abstracție a transformat fundamental modul în care sistemele informatice pot interpreta și genera limbaj natural.

Lucrarea seminală a lui Bengio et al. a demonstrat eficacitatea acestor modele în captarea dependențelor pe termen lung și a contextului amplu al cuvintelor în cadrul secvențelor de text, deschizând calea către explorarea unor metode mai sofisticate de procesare a limbajului și explicând în detaliu mecanismele prin care MLP-urile și alte modele de rețele neuronale pot învăța reprezentări complexe ale datelor. "Această abordare a deschis noi perspective în înțelegerea limbajului, permițând dezvoltarea unor sisteme capabile să interpreteze și să genereze text cu o acuratețe remarcabilă. Prin introducerea și aplicarea MLP-urilor în modelarea limbajului, s-a deschis calea către explorarea unor metode mai sofisticate de procesare a limbajului" [1].

### RNN și Transformeri în Procesarea Limbajului Natural

Introducerea rețelelor neuronale profunde, și în special a arhitecturilor recurente (RNN) și ulterior a transformatorilor, a reprezentat un punct de cotitură în modelarea lingvistică. Mikolov și colaboratorii săi, prin lucrarea lor privind reprezentările dense ale cuvintelor, au demonstrat capacitatea rețelelor neuronale de a captura nuanțe semantice complexe și relații între cuvinte cu o acuratețe fără precedent [22]. Aceasta a deschis calea spre dezvoltarea de modele lingvistice capabile să înțeleagă și să genereze limbaj natural într-un mod mult mai umanoid.

Mai departe, Vaswani și echipa sa au introdus arhitectura Transformer, care a revoluționat PLN-ul prin îmbunătățirea capacității de a modela dependențele pe termen lung și de a procesa secvențe de text într-un mod paralel, eficientizând în același timp învățarea reprezentărilor lingvistice. "Aceasta a permis modelelor să gestioneze dependențe pe termen lung și să îmbunătățească semnificativ eficiența în procesarea limbajului" [23].

Această inovație a stat la baza dezvoltării ulterioare a modelelor de limbaj de dimensiuni mari, cum ar fi BERT și GPT, care au demonstrat performanțe remarcabile în diverse sarcini de PLN, stabilind noi standarde de excelență în domeniu[24][25].

Pe măsură ce PLN-ul continuă să evolueze, este evident că modelele lingvistice se îndreaptă către o înțelegere tot mai profundă și mai holistă a limbajului uman, depășind treptat limitele care caracterizau abordările inițiale bazate pe statistici și n-grame.

## Modelarea și Generarea de Text cu Bigrame folosind Rețele Neuronale Profunde în Prelucrarea Limbajului Natural

### Explorarea Datelor și Preprocesarea

Numele incluse în setul de date names.txt, ca exemplu, sunt cele mai comune 32.000 de nume luate de pe ssa.gov pentru anul 2018. Arată astfel:

emma

olivia

ava

isabella

sophia

charlotte

...

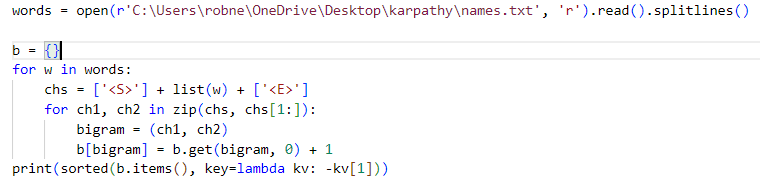


Figura 3.1: Calcularea frecvenței bigramelor din fișierul 'names.txt' în Python

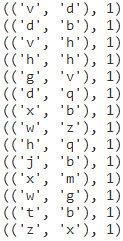
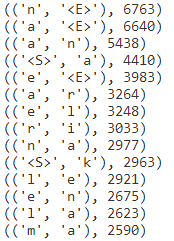


Figura 3.2: Primele si ultimele bigrame ca frecvență

### Vizualizarea Frecvenței Bigramelor Folosind Tensori Bidimensionali

* Definiție: Un tensor în PyTorch este similar cu un array multidimensional, asemănător cu array-urile NumPy. Poate fi un scalar (tensor de rang 0), un vector (tensor de rang 1), o matrice (tensor de rang 2) sau o structură multidimensională de ordin superior.
* Creare: Tensorii pot fi creați folosind diferite metode, cum ar fi specificarea directă a datelor, generarea aleatoare sau conversia din alte structuri de date, precum liste sau array-uri NumPy.
* Accesul la elemente: Elementele unui tensor pot fi accesate și modificate folosind indecși similari cu cei din NumPy. De asemenea, PyTorch oferă metode pentru lucrul cu sub-tensori și pentru manipularea formei unui tensor.
* Operații cu tensori: PyTorch oferă o gamă largă de funcții și operații matematice pentru lucrul cu tensori, inclusiv adunare, scădere, înmulțire, împărțire, operări de însumare și redus (sumă, produs, minim, maxim etc.), funcții trigonometrice, funcții de activare pentru rețele neurale, operări de reducere (cum ar fi media sau deviația standard) și multe altele.

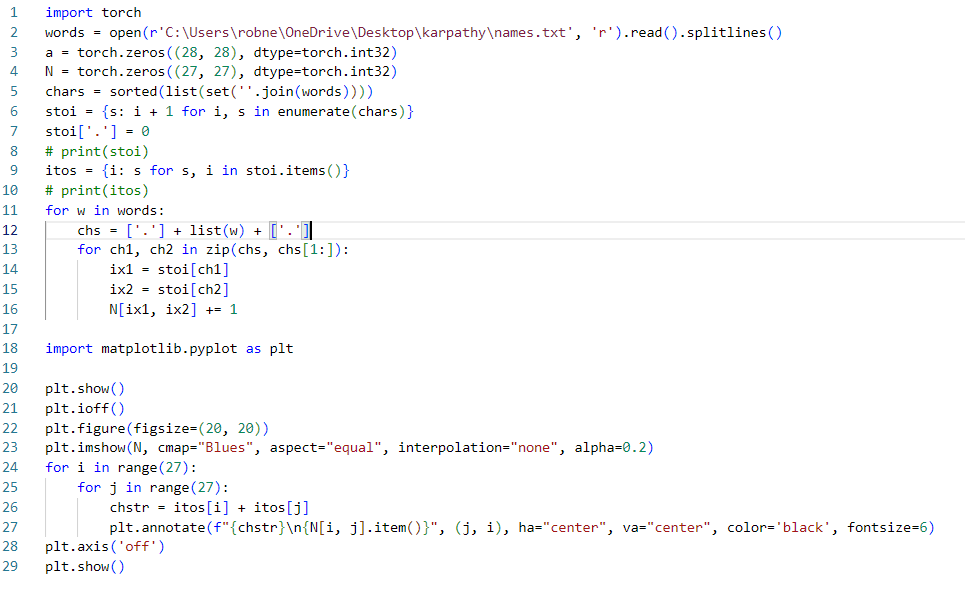


Figura 3.3: Generarea matricei de frecvență

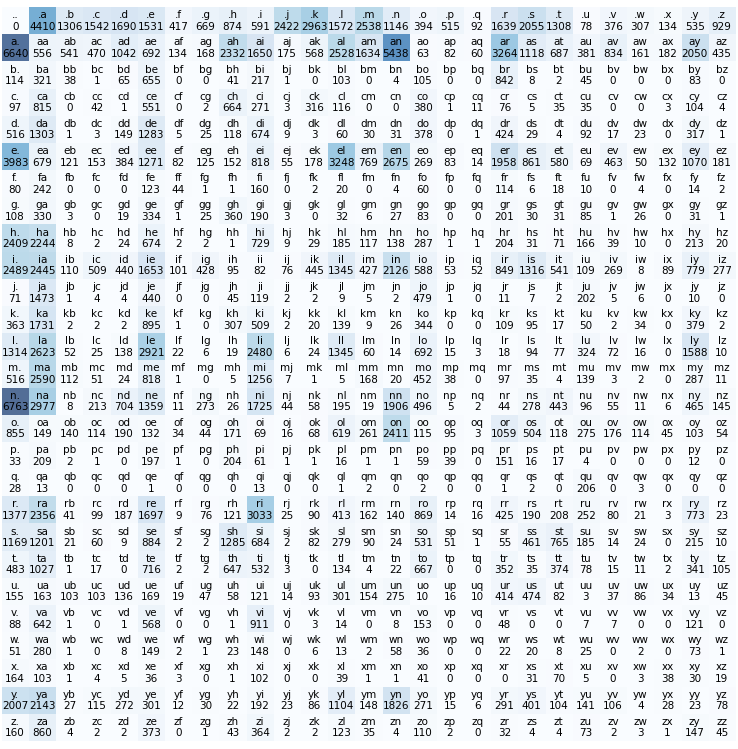


Figura 3.4: Graficul matricei de frecvență

***Explicații pentru grafic:***

* Axa X și Y: Acestea reprezintă caracterele individuale din setul de caractere. Deoarece sunt 27 de caractere distincte (fără caracterele majuscule și caracterele speciale), axele X și Y variază de la 0 la 26.
* Culori și intensitate: Intensitatea culorii indică frecvența perechilor de caractere. Culoarea albastră mai închisă indică o frecvență mai mare, în timp ce culorile albastre mai deschise indică o frecvență mai mică.
* Etichetele: Fiecare celulă conține două caractere care formează bigramul și frecvența bigramului în setul de cuvinte. Aceste etichete sunt afișate în interiorul fiecărei celule pentru a arăta perechea de caractere și numărul lor de apariții.

### Popularea Tensorului N, Normalizarea datelor

***Etape în popularea si maparea tensorului:***

* Inițializarea matricei N cu dimensiunile (27, 27), unde fiecare intrare va stoca un număr întreg reprezentând numărul de tranziții între perechi de caractere. Dimensiunea 27 se referă la cele 26 de litere din alfabet și un caracter special, cum ar fi punctul.
* Se extrag caracterele unice din cuvintele din variabila words, se sortează și se construiește un dicționar stoi (String to Integer) care mapează fiecare caracter la un index întreg. Deoarece a fost specificat că punctul ('.') va avea indexul 0, acesta este adăugat explicit în dicționar.
* Se creează un dicționar invers, itos (Integer to String), care mapează fiecare index întreg înapoi la caracterul corespunzător.
* Se parcurg cuvintele din lista words:
* Fiecare cuvânt este extins cu un punct la început și un punct la sfârșit.
* Se iterează prin fiecare pereche de caractere consecutive în cuvânt și se actualizează matricea N pentru a reflecta tranziția.





Figura 3.5: Primul rând al tensorului N

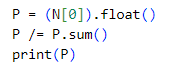
 

Figura 3.6: Normalizarea datelor, prin conversie la probabilități

***Torch.multinomial***

Funcția torch.multinomial(input, num\_samples, replacement=False, \*, generator=None) este o funcție din biblioteca PyTorch care efectuează eșantionarea multinomială, adică extrage un număr specificat de eșantioane dintr-o distribuție multinomială definită de vectorul de probabilități furnizat.

* input: Este un tensor ce conține probabilitățile asociate fiecărei categorii
* num\_samples: Numărul de eșantioane care trebuie extrase din distribuția multinomială definită de input.
* replacement: Un indicator boolean care specifică dacă eșantioanele trebuie să fie extrase cu înlocuire sau fără înlocuire. Dacă este True, atunci eșantioanele sunt extrase cu înlocuire, ceea ce înseamnă că aceeași categorie poate fi aleasă de mai multe ori. Dacă este False, eșantioanele sunt extrase fără înlocuire, ceea ce înseamnă că o dată ce o categorie este aleasă, ea nu mai poate fi aleasă din nou.
* generator: Generatorul de numere aleatoare utilizat pentru a genera eșantioanele. Dacă este None, atunci este utilizat un generator implicit. Acesta este utilizat pentru a controla comportamentul generării de numere aleatoare în funcțiile PyTorch care necesită un generator de numere aleatoare pentru a asigura reproducibilitatea și consistența rezultatelor.Generatorul de numere aleatoare poate fi specificat explicit în funcțiile care utilizează eșantionarea sau alte operații care implică alegeri aleatoare, permițând astfel controlul asupra reproducibilității rezultatelor.

Funcția returnează un tensor care conține indicele categoriilor eșantionate.

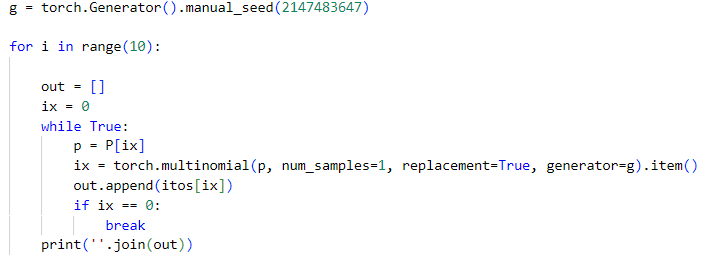


Figura 3.7: Codul Python de generare a “numelor”

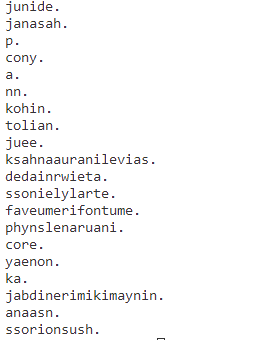


Figura 3.8: Generări

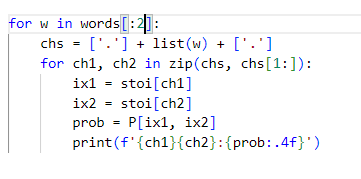
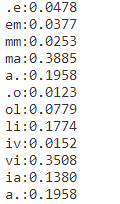
 

Figura 3.8: Probabilitatea de apariție, conform modelului a perechilor de caractere in primele 2 nume

# Bibliografie

[1] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

[2] LeCun, Y., Bengio, Y., & Hinton, G. (2015). "Deep learning." Nature, 521(7553), 436-444.

[3] Ruder, S. (2016). "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747.

[4] LeCun, Y., Bottou, L., Orr, G.B., & Müller, K.R. (1998). "Efficient BackProp." In: Orr, G.B., Müller K.R. (eds) Neural Networks: Tricks of the Trade. Springer, Berlin, Heidelberg.

[5] Bottou, L. (2010). "Large-scale machine learning with stochastic gradient descent." Proceedings of COMPSTAT'2010.

[6] Kingma, D.P., & Ba, J. (2014). "Adam: A Method for Stochastic Optimization." arXiv preprint arXiv:1412.6980.

[7] Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). "Learning Representations by Back-propagating Errors."

[8] Kim, Y. (2014). "Convolutional Neural Networks for Sentence Classification."

[9] Hochreiter, S., & Schmidhuber, J. (1997). "Long Short-Term Memory."

[10] Vaswani, A., et al. (2017). "Attention Is All You Need."

[11] Devlin, J., et al. (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding."

[12] Radford, A., et al. (2020). "Language Models are Few-Shot Learners."

[13] Robbins, H., & Monro, S. (1951). "A Stochastic Approximation Method." The Annals of Mathematical Statistics.

[14] Kingma, D.P., & Ba, J. (2014). "Adam: A Method for Stochastic Optimization." arXiv preprint arXiv:1412.6980.

[15] Tieleman, T., & Hinton, G. (2012). "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning.

[16] Prechelt, L. (1998). "Early Stopping — But When?" In: Orr, G.B., Müller K.R. (eds) Neural Networks: Tricks of the Trade. Springer, Berlin, Heidelberg

[17] Reddi, S.J., Kale, S., & Kumar, S. (2018). "On the Convergence of Adam and Beyond." ICLR 2018.

[18] Loshchilov, I., & Hutter, F. (2017). "Fixing Weight Decay Regularization in Adam." arXiv preprint arXiv:1711.05101.

[19] Zhou, B., et al. (2020). "Towards Understanding the Dynamics of Adam." arXiv preprint arXiv:2006.02341.

[20] Jurafsky, D., & Martin, J. H. (2020). "Speech and Language Processing". Prentice Hall.

[21] Chelba, C., et al. (2000). "A Large Vocabulary Continuous Speech Recognition System". IEEE Transactions on Speech and Audio Processing.

[22] Mikolov, T., et al. (2013). "Efficient Estimation of Word Representations in Vector Space". arXiv preprint arXiv:1301.3781.

[23] Vaswani, A., et al. (2017). "Attention Is All You Need". Advances in Neural Information Processing Systems.

[24] Devlin, J., et al. (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." arXiv preprint arXiv:1810.04805.

[25] Brown, T., et al. (2020). "Generative Pre-trained Transformer 3 (GPT-3)." arXiv preprint arXiv:2005.14165.

[26] Markov, A.A. (1913). "O introducere în lanțurile Markov și aplicarea lor", Lucrări ale Societății Matematice Ruse.

