



UNIVERSITATEA DIN BUCUREŞTI



FACULTATEA DE
MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA MATEMATICĂ - INFORMATICĂ

Lucrare de licență

REȚELE NEURALE ȘI APLICAȚII ÎN ANALIZA IMAGINILOR MEDICALE

Absolvent

Dragomir Elena Alexandra

Coordonator științific

Conf. Dr. Iulian Cîmpean

București, iunie 2022

Rezumat

Inteligenta artificială a cunoscut o creștere deosebită în ultimii ani, în special în domeniul învățării automate și al învățării profunde. Acestea se realizează cu ajutorul unor structuri ce își propun să imite creierul uman din procesul învățării, numite rețele neurale. Matematic, ele pot fi vizualizate ca o compunere de funcții pentru care ne propunem să găsim parametrii care minimizează eroarea dintre rezultatul real și cel prezis. În cadrul lucrării vom demonstra că metodele prin care rețeaua ”învăță” funcționează, ajungându-se din punct de vedere teoretic la o convergență, în anumite cazuri particulare. De asemenea, vom determina și impactul pe care aceste tehnici îl au în domeniul medical printr-o aplicație de detecție a pneumoniei.

Abstract

Artificial intelligence has grown exponentially in recent years, with special focus on machine learning and deep learning. This is possible with the help of structures that aim to imitate the human brain in the learning process, called neural networks. Mathematically, they can be viewed as a composition of functions for which we aim to find parameters that minimize the error between the actual and the predicted result. We will prove in this paper that the methods by which the network ”learns” work, reaching theoretically a convergence in certain particular cases. We will also analyze the impact that these techniques have in the medical field through a pneumonia detection application.

Cuprins

1	Introducere	5
2	Rețele Neurale Artificiale	6
2.1	Arhitectura Rețelelor Neurale	6
2.1.1	Modele liniare si limitări	6
2.1.2	Structură	11
2.1.3	Funcția de activare	12
1)	<i>Functia sigmoidală Heaviside.</i>	13
2)	<i>Functia sigmoidală logistică.</i>	13
3)	<i>Functia Softmax.</i>	14
4)	<i>Functia ReLu (Rectified Linear Unit).</i>	14
2.2	Gradient Descent	15
2.3	Stochastic Gradient Descent (SGD)	25
2.4	Backpropagation	31
2.5	Antrenarea MNIST folosind ANN	35
3	Rețele Neurale Convoluționale	37
3.1	Arhitectură	37
3.2	Convoluție	39
3.3	Filtre	41
3.4	Pooling	42
4	Aplicații	43
4.1	Rețele Neurale Convoluționale în analizarea imaginilor medicale	43
4.1.1	AlexNet	44
4.1.2	ResNet18	46
4.1.3	DenseNet201	47

4.2	Class Activation Map	49
5	Anexe	52
5.1	Regresia Liniară	52
5.2	Antrenare MNIST folosind ANN	53
5.3	Pneumonia Detection	57
	Bibliografie	64

1 Introducere

Domeniul inteligenței artificiale este în continuă expansiune în ultimii ani iar o atenție mare se pune îndeosebi pe partea de învățare automată (*machine learning*) și o subramură a sa numită învățare adâncă (*deep learning*). Aceasta constă în găsirea unor funcții matematice care să mapeze un input de un output. Dacă în programarea tradițională folosim date împreună cu reguli dinainte știute pentru a afla un rezultat, în cazul învățării automate reguliile trebuie deduse în urma unor date de input și rezultate dorite. În deep learning se folosesc rețelele neurale pentru a extrage aceste reguli. Acestea pot fi văzute ca o compunere de funcții care, prin felul lor de a fi aplicate pe rezultate deja existente, se pot vizualiza sub forma unor straturi ce sugerează și ideea de ”adâncime”. [2]

În această lucrare ne propunem să prezentăm baza matematică a rețelelor neurale și felul în care acestea au evoluat spre structuri mai complexe care sunt folosite în probleme de clasificare a imaginilor, precum și aplicații ale acestora. Lucrarea este formată din trei părți și anume:

În primul capitol vom vorbi despre rețelele neurale artificiale, structura lor și felul în care informația se propagă prin ele, urmărind în mare parte abordarea din [15] și [25]. Algoritmii care stau la baza învățării sunt Gradient Descent și Backpropagation, întrucât, cu ajutorul lor, rețeaua reușește să se îmbunătățească la fiecare pas, ajungând mai aproape de rezultatul dorit. La finalul capitolului este prezentată și o aplicație de clasificare a unor imagini cu cifre scrise de mână din setul de date MNIST [6].

În al doilea capitol se face trecerea către un nou fel de rețele neurale, cele convolutionale, și motivele folosirii lor în majoritatea problemelor de clasificare de imagini. Aici intervin alte tipuri de operații și straturi cu noi roluri.

În ultimul capitol prezentăm o aplicație a rețelelor neurale convolutionale în analiza imaginilor medicale, mai precis în detecția pneumoniei. Codul a fost realizat de către autoarea lucrării pornind de la tutorialul ”Deep Learning with PyTorch for Medical Image Analysis” împreună cu o contribuție personală la antrenările realizate.

2 Rețele Neurale Artificiale

Rețelele neurale artificiale (ANN) sunt structuri care simulează activitatea creierului uman din procesul învățării. Aceasta este antrenată la început folosind un set de date care modelează rețeaua în aşa fel încât aceasta să poată ulterior recunoaște modelele deja întâlnite pentru a prezice rezultatul dorit.

2.1 Arhitectura Rețelelor Neurale

2.1.1 Modele liniare și limitări

Un model elementar de rețea neurală este regresia liniară. Dacă avem o dimensiune, datele de input vor fi de forma $(x_i, y_i) \in \mathbb{R}^2$, $i = \overline{1, n}$ unde fiecare y_i îl estimăm ca fiind de forma $\hat{y}_i = \alpha x_i + \beta$. Scopul rețelei neurale în acest caz este să estimeze α și β cât mai exact pentru ca mai apoi, având la dispoziție un x , să putem estima pe baza modelului învățat, că output-ul va fi aproximativ $\hat{y} = \alpha x + \beta$.

Rețeaua neurală în acest caz va arăta astfel:

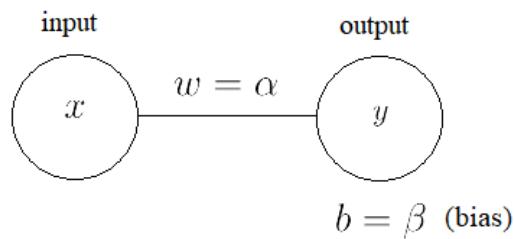


Figure 2.1.1: Rețea neurală a unui model liniar în \mathbb{R}

Se consideră ca funcție de activare (subsecțiunea 2.1.3) funcția identitate, $S(x) = x$, $x \in \mathbb{R}$ și rețeaua neurală se scrie sub forma:

$$\hat{y} = \Phi(x) = S(w x + b) = \alpha x + \beta$$

Asemănător, putem extinde problema în d dimensiuni și avem pentru fiecare intrare de date:

$$\hat{y}_i = w_1 x_i^1 + w_2 x_i^2 + \dots + w_d x_i^d + b, i = \overline{1, n}$$

unde cu \hat{y}_i am notat estimarea valorii reale y_i la fiecare pas. Putem scrie relația de mai sus compact, ca produs scalar:

$$\hat{y}_i = w^T \cdot x_i + b,$$

unde $x_i = (x_i^1, \dots, x_i^d)$, $w = (w_1, \dots, w_n) \in \mathbb{R}^d$ și $b \in \mathbb{R}$.

Pentru toate intrările putem scrie problema matricial

$$\hat{y} = X w + \bar{b} \quad (2.1.1)$$

unde $X \in \mathbb{R}^{n \times d}$, $y, \bar{b} \in \mathbb{R}^n$, sau:

$$\begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} x_1^1 \dots x_1^d \\ \vdots \\ x_n^1 \dots x_n^d \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix} + \begin{bmatrix} b \\ \vdots \\ b \end{bmatrix}$$

De data aceasta, rețeaua neurală pentru fiecare intrare i va arăta astfel:

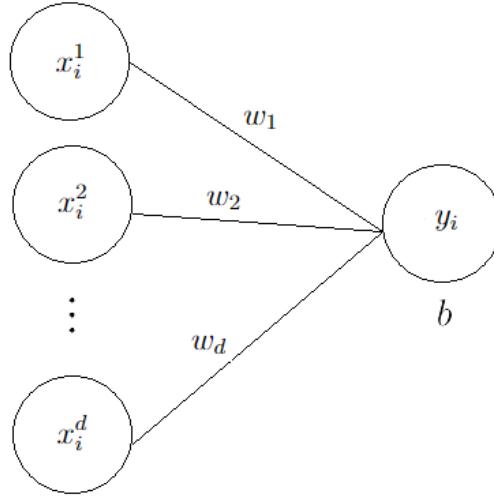


Figure 2.1.2: Rețea neurală a unui model liniar în \mathbb{R}^d

Se consideră ca funcție de activare (subsecțiunea 2.1.3), ca mai devreme, funcția identitate,

$S(x) = x$, $x \in \mathbb{R}^d$ și rețeaua neurală se scrie sub forma:

$$\Phi(x) = S(wx + b)$$

Prin regresie liniară vrem să gasim cele mai bune aproximări ale vectorilor w, \bar{b} din ecuația (2.1.1) astfel încât, dacă avem un nou set de date cu aceeași distribuție ca X , să putem prezice rezultatul corect cu o eroare cât mai mică.

Un set real de date este rareori sub o formă liniară exactă. Prin urmare, vrem să gasim o formă liniară cât mai apropiată de valorile reale. Pentru a obține aceste aproximări cât mai bune ale vectorilor w și \bar{b} , avem nevoie să minimizăm distanța dintre valorile estimate și cele reale. Vom numi funcția egală cu această distanță, funcția loss. Cea mai folosită funcție loss este eroarea patratică [29]:

$$l_i(w, b) = \frac{1}{2} (\hat{y}_i - y_i)^2 \quad (2.1.2)$$

Alte funcții loss pe care le vom folosi mai târziu sunt cross entropy și binary cross entropy. Cross entropy este definită pentru o clasificare în c clase, unde vom nota cu $\hat{y}_i(t)$ al t -lea output al rețelei pentru input-ul x_i iar $y_i(t)$ va fi 1 dacă x_i este repartizat în clasa t și 0 altfel [3]. Funcția va fi de forma:

$$l_i(w, b) = - \sum_{t=1}^c y_i(t) \cdot \log(\hat{y}_i(t)) \quad (2.1.3)$$

Binary cross entropy este cazul particular $c = 2$ al funcției de mai sus pentru clasificarea în două clase. În continuare vom reveni la eroarea patratică ca funcție loss.

Un exemplu de estimare a valorilor lui y_i ca regresie liniară cu distanțele dintre estimare și valoarea reală se găsește în figura 2.1.3

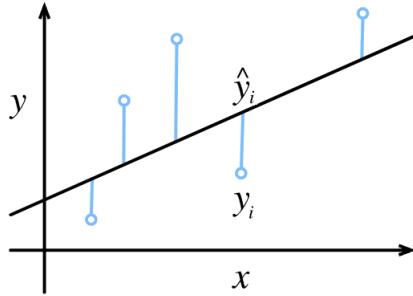


Figure 2.1.3: Exemplu regresie liniară (imagine preluata din [29])

Pentru a măsura cât de bună este estimarea pe întregul set de date, facem o medie a funcțiilor loss pentru fiecare intrare [29]:

$$L(w, b) = \frac{1}{n} \sum_{i=1}^n l_i(w, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (w^T x_i + b - y_i)^2 \quad (2.1.4)$$

În timpul antrenării modelului, vrem să găsim parametrii (w^*, b^*) care minimizează pierderea totală $L(w, b)$:

$$w^*, b^* = \underset{w,b}{\operatorname{argmin}} L(w, b) \quad (2.1.5)$$

Problema de minimizare se va rezolva prin metoda numită Gradient Descent, pe care o vom prezenta în subsecțiunea (2.2).

Limitări. Dacă setul de date inițial nu se poate aproxima natural sub forma unei drepte, vom obține erori mari la predicție iar dreapta dată nu va reflecta realitatea. Un exemplu de astfel de date se poate observa în figura 2.1.4.

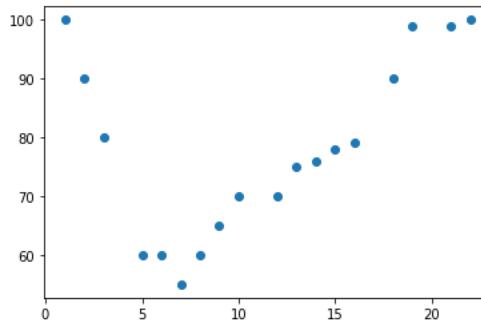


Figure 2.1.4: Exemplu set de date

Dacă încercăm să aproximăm acest set de date cu o dreaptă, vom obține figura 2.1.5 care se poate observa că nu reprezintă o estimare exactă. Mărind gradul polinomului de aproximare, putem obține estimări din ce în ce mai exacte (2.1.6). Acest aspect este dat de forma funcției de activare, care trebuie să aibă și forme neliniare pentru a identifica și modele mai complexe.

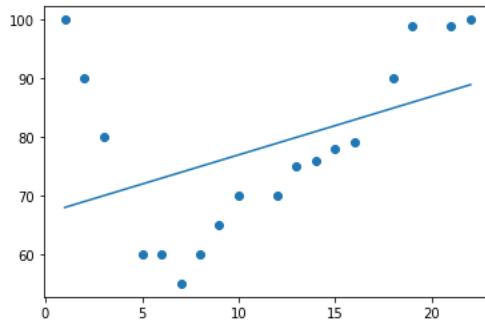


Figure 2.1.5: Aproximare cu polinom de grad 1

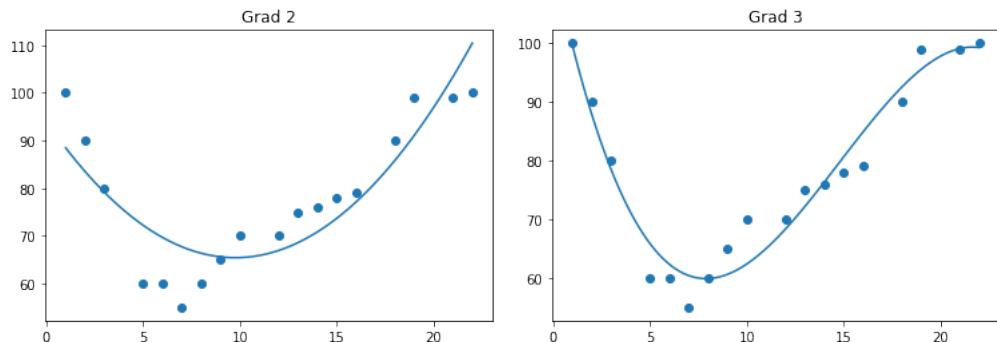


Figure 2.1.6: Aproximare cu polinoame de grad superior

Codul sursă pentru graficele de aproximare de mai sus se găsește în anexa 5.1.

2.1.2 Structură

Rețelele neurale, asemenea sistemului nervos uman, sunt alcătuite din neuroni. Prima abstracțizare a acestora a fost făcută în 1943 de către Warren McCulloch și Walter Pitts. [31]

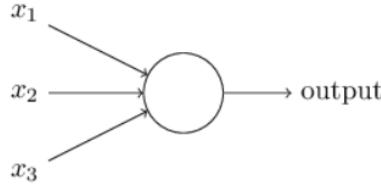


Figure 2.1.7: Neuron (imagine preluata din [15])

Majoritatea rețelilor neurale sunt formate din mai multe straturi de neuroni. Straturile poziționate între cel de input și output se numesc straturi ascunse. Rețelele cu mai multe straturi ascunse se mai numesc și rețele neurale adânci.

Un neuron preia mai multe input-uri x_1, x_2, \dots care transmit, cu ponderi diferite, mai multe output-uri pe fiecare strat, ce vor fi folosite la rândul lor ca input-uri pentru următoarele straturi.

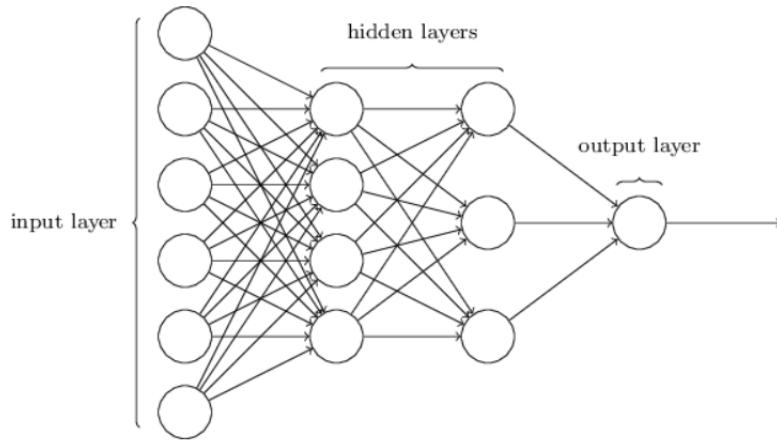


Figure 2.1.8: Exemplu rețea neurală adâncă (imagine preluata din [15])

În continuare vom folosi notații asemănătoare cu cele din tutorialul de "Deep Learning" de la Stanford [32; 25]. Pentru ponderi vom folosi notația $w_{ij}^{(l)}$, unde i reprezintă numărul nodului din stratul $l + 1$ iar j numărul nodului de pe stratul l . Asemănător, bias-ul va fi $b_i^{(l)}$, unde i este numărul nodului de pe stratul $l + 1$ iar al i -lea neuron de pe stratul l va fi $x_i^{(l)}$.

Vom nota output-urile cu $h_i^{(l)}$, unde i reprezintă numărul nodului de pe stratul l al retelei:

$$h_i^{(l+1)} = \begin{cases} f\left(\sum_{j=1}^n w_{ij}^{(l)} x_j + b_i^{(l)}\right) & , \text{dacă } l = 1 \\ f\left(\sum_{j=1}^n w_{ij}^{(l)} h_j^{(l)} + b_i^{(l)}\right) & , \text{dacă } l \geq 2 \end{cases}$$

unde f se numește funcție de activare (despre care vom detalia în secțiunea următoare), care determină felul în care informația va ajunge sau nu la output.

Pentru simplitate, vom nota și cu $z_i^{(l+1)}$ output-ul nodului i de pe stratul l al retelei, înainte de a trece prin funcția de activare, adică:

$$z_i^{(l+1)} = \begin{cases} \sum_{j=1}^n w_{ij}^{(l)} x_j + b_i^{(l)} & , \text{dacă } l = 1 \\ \sum_{j=1}^n w_{ij}^{(l)} h_j^{(l)} + b_i^{(l)} & , \text{dacă } l \geq 2 \end{cases}$$

Vom avea deci că $h_i^{(l)} = f(z_i^{(l)})$.

2.1.3 Funcția de activare

Funcțiile de activare sunt folosite în rețelele neurale pentru a transmite input-ul output-ului, care este folosit la rândul lui ca un nou input pentru stratul următor (în cazul rețelelor cu mai multe straturi). Acestea sunt aplicate produsului scalar $w \cdot x$ la care se adună bias-ul.

Așa cum am văzut și în subsecțiunea 2.1.1, funcția de activare are un rol vital în ceea ce privește nivelul de complexitate care poate fi antrenat de către rețea neurală, motive pentru care se folosesc mai mult funcții de activare non-liniare. Altfel, rețea neurală ar funcționa ca o regresie liniară (cum am văzut în 2.1.1) și s-ar propaga aceleași erori pe parcurs. În cazul liniar rețea neurală se va putea adapta doar modificăriilor liniare ale input-ului iar caracteristicile non-liniare nu vor putea fi modelate corect [22].

Rolul funcției de activare depinde de poziția sa în rețea neurală. Astfel, cele folosite după straturile ascunse au rolul de a transforma maparea liniară învățată anterior într-o formă non-liniară folosită pentru propagare, iar cele folosite în stratul output-ului sunt utile pentru a formula predictii [16].

În continuare vom da exemple de funcții sigmoidale, Softmax și funcții de activare semi-liniare ReLu (*Rectified Linear Unit*).

1) Funcția sigmoidală Heaviside. Este o funcție treaptă, nederivabilă, care funcționează ca un activator. Pentru numere pozitive, aceasta ia valoarea 0, iar pentru numere negative ia valoarea 1.

$$H(x) = \begin{cases} 1 & , x > 0 \\ 0 & , x \leq 0 \end{cases}$$



Figure 2.1.9: Funcția sigmoidală Heaviside

2) Funcția sigmoidală logistică. Este o funcție non-liniară, derivabilă și este dată de relația:

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

În general, este de preferat să se folosească funcția sigmoidală ca funcție de activare pentru stratul de output pentru a prezice o probabilitate.

Se poate observa din figura 2.1.10 că pentru un a mare, funcția sigmoidală logistică tinde către o variantă continuă și derivabilă a funcției Heaviside. Așadar, are și un rol de regularizare pentru aceasta.

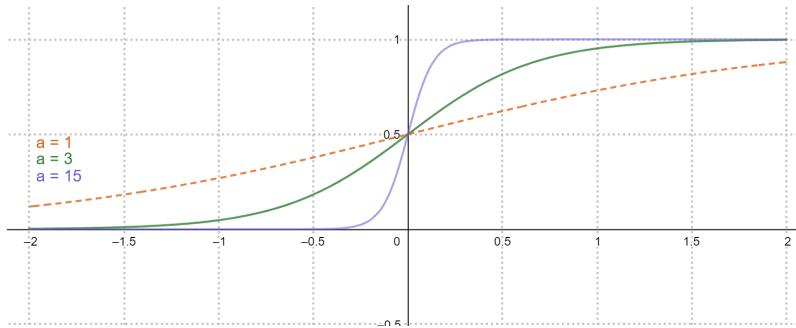


Figure 2.1.10: Functii sigmoidale în funcție de a

3) Funcția Softmax. Este folosită când avem nevoie de un output sub formă de probabilitate pentru mai multe clase de output-uri. Fiecare valoare este cuprinsă între 0 și 1 iar suma probabilităților este 1. Funcția returnează probabilitatea fiecărei clase iar cea dorită va avea probabilitatea cea mai mare. Având output-urile x_1, x_2, \dots, x_n , valoarea funcției în fiecare x_i va fi:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Principala diferență dintre funcția sigmoidală și Softmax este dimensiunea output-ului. În timp ce funcția sigmoidală este folosită în clasificările binare, Softmax este folosită pentru clasificări de mai multe variabile [16].

4) Funcția ReLu (Rectified Linear Unit). Funcția de activare semi-liniară ReLu este cea mai folosită funcție de activare pentru probleme de deep learning datorită formei sale simple de procesat [20]. Matematic, ea este de forma:

$$f(x) = \max(x, 0) = \begin{cases} x & , \text{dacă } x > 0 \\ 0 & , \text{altfel} \end{cases}$$

Fiind o funcție aproape liniară, ea păstrează proprietățile modelelor liniare ușor de optimizat [16].

Rețelele neurale au o proprietate de universalitate, în sensul că orice funcție poate fi reprezentată printr-o rețea neurală cu un singur strat ascuns. Matematic, acest lucru este dat de Teorema de aproximare universală și este demonstrat de George Cybenko în [4].

Teorema de aproximare universală - George Cybenko (1989) [4; 13]

Teorema 2.1. Fie σ o funcție sigmoidală:

$$\sigma(x) = \begin{cases} 1 & , \text{pentru } x \rightarrow +\infty \\ 0 & , \text{pentru } x \rightarrow -\infty \end{cases}$$

Fie multimea:

$$\mathcal{A} = \left\{ g : [0, 1]^n \rightarrow \mathbb{R} : \exists N \geq 1, w_j^1 \in \mathbb{R}^n, w_j^2 \in \mathbb{R}, b_j \in \mathbb{R}, j = \overline{1, n} \text{ a.i.} \right. \\ \left. g(x) = \sum_{j=1}^N w_j^2 \sigma((w_j^1)^T x + b_j) \right\}$$

Atunci $\mathcal{A} \subseteq C([0, 1]^n)$ cu densitate. Mai precis, $\forall f \in C([0, 1]^n)$ și $\epsilon > 0$, $\exists g \in \mathcal{A}$ pentru care:

$$|g(x) - f(x)| < \epsilon, \text{ pentru orice } x \in [0, 1]^n$$

2.2 Gradient Descent

Așa cum am menționat anterior, scopul unei rețele neurale este de a prezice un output y având un input x , după o antrenare a rețelei cu un set de n date $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$. Antrenarea rețelei neurale constă în actualizarea valorilor ponderilor w și bias-ului b astfel încât să minimizăm eroarea. Aceasta se face folosind metoda gradientului descendente [25]. Un caz simplu al acesteia putem observa în figura (2.2.1).

Pentru funcția eroare (sau loss) vom alege, ca mai devreme (2.1.4), eroarea pătratică. Pentru un singur input (x_i, y_i) aceasta va fi:

$$J(w, b, x_i, y_i) = \frac{1}{2} \|y_i - \hat{y}_i\|^2$$

unde \hat{y}_i reprezinta predicția rețelei neurale. Pentru un set de n input-uri eroarea va fi [25]:

$$J(w, b) = \frac{1}{n} \sum_{i=0}^n \frac{1}{2} \|y_i - \hat{y}_i\|^2 \tag{2.2.1}$$

$$= \frac{1}{n} \sum_{i=0}^n J(w, b, x_i, y_i) \tag{2.2.2}$$

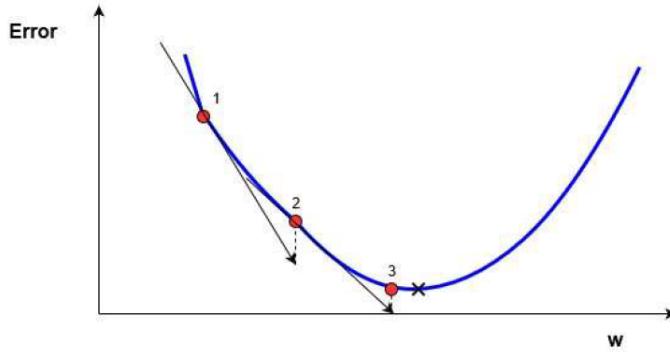


Figure 2.2.1: Problema gradientului descendent într-o dimensiune (imagine preluată din [25], pag 17)

Ideea principală a algoritmului este de a pleca cu un punct arbitrar din domeniul de definiție al erorii care să se deplaseze de-a lungul gradientului până ajunge (dacă este posibil) la un punct staționar. Considerăm mai întâi cazul general al problemei gradientului descendent, după care particularizăm pentru cazul minimizării erorii $J(w, b)$.

Dacă avem funcția derivabilă f și punctul $x^{(0)}$ arbitrar în interiorul domeniului lui f atunci, la fiecare pas $k \geq 0$, deplasăm iterativ punctul $x^{(k)}$ printr-o deplasare în direcția $\Delta x^{(k)}$ cu un pas α_k (valoare pe care o numim *rată de învățare*) către punctul $x^{(k+1)} = x^{(k)} + \alpha_k \Delta x^{(k)}$. Cum în metoda gradientului sensul deplasării este opus gradientului în acel punct, vom avea $\Delta x = -\nabla f(x)$ [23]. Deci, iterația va fi de forma:

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla f(x^{(k)}) \quad (2.2.3)$$

Scopul este să găsim x^* astfel încât:

$$x^* = \arg \min_{x \in \mathbb{R}^d} f(x)$$

Pentru a demonstra convergența unei iterații de tipul (2.2.3) avem nevoie să definim următoarele noțiuni, urmărind ideile din [7]. Reamintim:

Definiția 2.2. Fie $f : \mathbb{R}^d \rightarrow \mathbb{R}$ o funcție de două ori diferențialabilă. Spunem că f este convexă

dacă:

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y), \forall x, y \in \mathbb{R}^d, t \in [0, 1]$$

Definiția 2.3. O funcție diferențiabilă f spunem că este L -regulată dacă gradientul său este continuu Lipschitz, adică:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \quad (2.2.4)$$

Ne uităm la câteva exemple de funcții L -regulate:

i) Pe \mathbb{R} definim f astfel încât $f(x) = \alpha x^2$ și $\alpha \in \mathbb{R}^*$. Avem că $f'(x) = 2\alpha x$, de unde:

$$|f'(x) - f'(y)| = 2|\alpha| |x - y|, \forall x, y \in \mathbb{R}.$$

Dacă luăm $L = 2|\alpha|$ obținem, conform definiției (2.3), că f este un exemplu de funcție L -regulată.

ii) Generalizând, putem defini $f : \mathbb{R} \rightarrow \mathbb{R}$ cu $f(x) = \alpha x^2 + \beta x + \gamma$, $\alpha, \beta, \gamma \in \mathbb{R}^*$. Avem $f'(x) = 2\alpha x + \beta$, de unde obținem din nou:

$$|f'(x) - f'(y)| = 2|\alpha| |x - y|, \forall x, y \in \mathbb{R}.$$

Deci f este $2|\alpha|$ -regulată.

iii) Observăm că pentru o funcție polinomială de grad 3 nu se mai pastrează L -regularitatea.

Dacă avem $f(x) = \alpha x^3$, $\alpha \in \mathbb{R}^*$, atunci $f'(x) = 3\alpha x^2$, de unde:

$$|f'(x) - f'(y)| = 3|\alpha| |x - y| |x + y|, \forall x, y \in \mathbb{R}.$$

Dacă ar exista $L \in \mathbb{R}^*$ care să satisfacă condițiile definiției (2.3), atunci acesta trebuie să verifice $L \geq 3|\alpha| |x + y|$, $\forall x, y \in \mathbb{R}$, contradicție întrucât $|x + y|$ nu poate fi mărginit superior.

iv) Pe \mathbb{R}^d , o clasa standard de exemple este data de:

$$f(x) = \|Ax - b\|^2, A \in \mathcal{M}_{d \times d}, x, b \in \mathbb{R}^d.$$

Obținem succesiv, folosind [11]:

$$\begin{aligned}
f(x) &= \|Ax - b\|^2 = (Ax - b)^T(Ax - b) \\
&= (Ax)^T Ax - (Ax)^T b - b^T Ax + b^T b \\
&= x^T A^T Ax - 2b^T Ax + b^T b \\
&= x^T A^T Ax - 2(A^T b)^T x + b^T b
\end{aligned}$$

Folosind $\frac{\partial c^T x}{\partial x} = c$ și $\frac{\partial x^T B x}{\partial x} = (B + B^T)x$ vom avea, pentru $c = A^T b$ și $B = A^T A$, că:

$$\begin{aligned}
\nabla_x f(x) &= (A^T A + (A^T A)^T)x - 2A^T b \\
&= 2A^T Ax - 2A^T b
\end{aligned}$$

Deci, avem că:

$$\nabla_x f(x) = 2A^T(Ax - b) \quad (2.2.5)$$

Așadar, putem scrie:

$$\begin{aligned}
\|\nabla f(x) - \nabla f(y)\| &= 2\|A^T(Ax - b - Ay + b)\| \\
&= 2\|A^T A(x - y)\| \\
&\leq 2\|A^T A\| \|x - y\|, \forall x, y \in \mathbb{R}^d
\end{aligned}$$

Putem concluziona deci că f este L -regulată, unde $L = 2\|A^T A\|$.

Consecințe directe ale acestei definiții sunt date de următoarele leme:

Lema 2.4. *Fie $f : \mathbb{R}^d \rightarrow \mathbb{R}$ o funcție de două ori diferențiabilă. Dacă f este L -regulată atunci:*

$$\langle \nabla^2 f(x)v, v \rangle \leq L\|v\|^2, \quad \forall x, y \in \mathbb{R}^d \quad (2.2.6)$$

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2}\|y - x\|^2 \quad (2.2.7)$$

Demonstrație. Dacă f este de două ori diferențiabilă putem scrie:

$$\nabla f(x) - \nabla f(x + \alpha v) = \int_0^\alpha \nabla^2 f(x + tv)v dt$$

Luăm produsul scalar cu v și obținem:

$$\begin{aligned}
\int_0^\alpha \langle \nabla^2 f(x + tv)v, v \rangle dt &= \langle \nabla f(x) - \nabla f(x + \alpha v), v \rangle \\
&\leq \|\nabla f(x) - \nabla f(x + \alpha v)\| \|v\| \\
&\stackrel{(2.2.4)}{\leq} L\|\alpha v\| \|v\| \\
&= L\alpha \|v\|^2
\end{aligned}$$

Împărțim ambii membri prin α și facem limita când $\alpha \xrightarrow[\infty]{} 0$:

$$\lim_{\alpha \rightarrow 0} \frac{1}{\alpha} \int_0^\alpha \langle \nabla^2 f(x + tv)v, v \rangle dt = \langle \nabla^2 f(x)v, v \rangle \leq L\|v\|$$

Pentru a două relație dezvoltăm Taylor și obținem:

$$\begin{aligned}
f(y) - f(x) &= \int_{t=0}^1 \langle \nabla f((1-t)x + ty), y - x \rangle dt \\
&= \langle \nabla f(x), y - x \rangle + \int_{t=0}^1 \langle \nabla f((1-t)x + ty) - \nabla f(x), y - x \rangle dt \\
&\leq \langle \nabla f(x), y - x \rangle + \int_{t=0}^1 \|\nabla f((1-t)x + ty) - \nabla f(x)\| \cdot \|y - x\| dt \\
&\stackrel{(2.2.4)}{\leq} \langle \nabla f(x), y - x \rangle + \int_{t=0}^1 L\|(1-t)x + ty - x\| \cdot \|y - x\| dt \\
&= \langle \nabla f(x), y - x \rangle + L \int_{t=0}^1 t\|y - x\|^2 dt \\
&= \langle \nabla f(x), y - x \rangle + \frac{L}{2}\|y - x\|^2
\end{aligned}$$

□

Lema 2.5. Dacă f este o funcție L -regulată atunci:

$$f\left(x - \frac{1}{L}\nabla f(x)\right) - f(x) \leq -\frac{1}{2L}\|\nabla f(x)\|^2 \quad (2.2.8)$$

și

$$f(x^*) - f(x) \leq -\frac{1}{2L}\|\nabla f(x)\|^2, \quad \forall x \in \mathbb{R}^d, \quad (2.2.9)$$

unde reamintim că x^* este dat de relația $x^* = \arg \min_{x \in \mathbb{R}^d} f(x)$.

Demonstrație. Introducem în (2.2.7) în loc de y pe $x - \frac{1}{L}\nabla f(x)$ și obținem:

$$\begin{aligned}
f(x - \frac{1}{L}\nabla f(x)) &\leq f(x) - \langle \nabla f(x), x - \frac{1}{L}\nabla f(x) - x \rangle + \frac{L}{2} \left\| x - \frac{1}{L}\nabla f(x) - x \right\|^2 \\
&= f(x) - \frac{1}{L} \langle \nabla f(x), \nabla f(x) \rangle + \frac{L}{2} \frac{1}{L^2} \|\nabla f(x)\|^2 \\
&= f(x) - \frac{1}{L} \|\nabla f(x)\|^2 + \frac{1}{2L} \|\nabla f(x)\|^2 \\
&= f(x) - \frac{1}{2L} \|\nabla f(x)\|^2
\end{aligned}$$

de unde obținem relația (2.2.8). Pentru demonstrația relației (2.2.9) folosim faptul că $f(x^*) \leq f(y)$, $\forall y \in \mathbb{R}^d$, în particular $f(x^*) \leq f(x - \frac{1}{L}\nabla f(x))$, în inegalitatea obținută anterior:

$$f(x^*) - f(x) \leq f(x - \frac{1}{L}\nabla f(x)) - f(x) \leq -\frac{1}{2L} \|\nabla f(x)\|^2$$

de unde concluzia.

□

Putem îmbunătății noțiunea de convexitate definind funcția tare convexă:

Definiția 2.6. O funcție se numește μ -tare convexă, unde $\mu > 0$, dacă:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2} \|y - x\|^2, \quad \forall x, y \in \mathbb{R}^d. \quad (2.2.10)$$

Lema 2.7. Fie f o funcție diferențiabilă. Următoarea afirmație este echivalentă cu f μ -tare convexă:

$$\langle \nabla^2 f(x)v, v \rangle \geq \mu \|v\|^2 \quad \forall x, v \in \mathbb{R}^d. \quad (2.2.11)$$

Avem în continuare exemple de funcții μ -tari convexe:

- i) Pe \mathbb{R} definim f prin $f(x) = \alpha x^2$, $\alpha > 0$. Avem că $\nabla^2 f(x) = f''(x) = 2\alpha$ iar $\langle \nabla^2 f(x)v, v \rangle = f''(x)v^2 = 2\alpha v^2$. Luând $\mu = 2\alpha$ vom avea din (2.2.11) că f e μ -tare convexă.
- ii) Pe \mathbb{R}^d definim f prin $f(x) = \|Ax - b\|^2$, $A \in \mathcal{M}_{d \times d}$ inversabilă, $x, b \in \mathbb{R}^d$. Folosind (2.2.5) obținem:

$$\nabla^2 f(x) = 2A^T A$$

Pentru ca f să fie μ -tare convexă trebuie să găsim μ astfel încât $\langle \nabla^2 f(x)v, v \rangle \geq \mu \|v\|^2 \forall x, v \in \mathbb{R}^d$, adică $\langle 2A^T A v, v \rangle \geq \mu \|v\|^2 \forall x, v \in \mathbb{R}^d$. Cum $A^T A$ este simetrică și (strict) pozitiv definită, rezultă că ea este ortogonal diagonalizabilă, adică există $U, D \in \mathcal{M}_{d \times d}$ cu U ortogonală și D diagonală cu intrări pozitive, astfel încât $A^T A = UDU^T$. Deci avem că

$$\begin{aligned}\langle 2A^T A v, v \rangle &= 2\langle UDU^T v, v \rangle \\ &= 2\langle DU^T v, U^T v \rangle \\ &= 2\langle Du, u \rangle\end{aligned}$$

unde cu u am notat $U^T v \in \mathcal{M}_{d \times 1}$. Cum D diagonală, rezultă că există $\lambda_1, \lambda_2, \dots, \lambda_d \in \mathbb{R}_+^*$ astfel încât:

$$D = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \lambda_d \end{bmatrix}$$

Și dacă avem $u = [u_1 \ u_2 \ \dots \ u_d]^T$, atunci:

$$Du = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \lambda_d \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_d \end{bmatrix} = \begin{bmatrix} \lambda_1 u_1 \\ \lambda_2 u_2 \\ \vdots \\ \lambda_d u_d \end{bmatrix}$$

Deci,

$$\langle Du, u \rangle = \sum_{i=1}^d \lambda_i u_i^2 \geq \min_{1 \leq i \leq d} \lambda_i \sum_{i=1}^d u_i^2 = \min_{1 \leq i \leq d} \lambda_i \|u\|^2$$

Dar, cum $u = U^T v$ rezultă că $\|u\|^2 = \|U^T v\|^2 = \langle U^T v, U^T v \rangle = \langle v, UU^T v \rangle = \langle v, I_d v \rangle = \|v\|^2$. Așadar,

$$\langle \nabla^2 f(x)v, v \rangle = \langle 2A^T A v, v \rangle = 2\langle Du, u \rangle \geq 2 \min_{1 \leq i \leq d} \lambda_i \|v\|^2$$

Deci, dacă luăm $\mu = 2 \min_{1 \leq i \leq d} \lambda_i > 0$ avem că f va fi μ -tare convexă.

Contra-exemplu. Noțiunea de tare convexitate se poate extinde și pentru $\mu = 0$, obținând fix proprietatea de convexitate clasică, însă această proprietate mai slabă nu este suficientă pentru

a asigura convergența iterației (2.2.3). Luăm, de exemplu, funcția $f : \mathbb{R} \rightarrow \mathbb{R}_+$ definită prin $f(x) = |x|$. Observăm că aceasta este 0-tare convexă și nu există $\mu > 0$ astfel încât f să fie μ -tare convexă.

Iterația (2.2.3) o putem scrie, echivalent, sub forma:

$$x_{n+1} = x_n - \alpha f'(x)$$

Alegem un punct de plecare, x_0 , în intervalul $(0, \alpha)$ și vom avea că $x_1 = x_0 - \alpha$, de unde $x_1 \in (-\alpha, 0)$. Cum $x_1 < 0$ rezultă că $f'(x) = -1$, deci $x_2 = x_1 + \alpha = x_0 - \alpha + \alpha = x_0$. Continuând, vom avea că $x_{2k} = x_0$, $\forall k \in \mathbb{N}$ iar $x_{2k+1} = x_1$, $\forall k \in \mathbb{N}$.

Așadar, sirul $(x_n)_n$ va oscila la infinit dacă are un punct în intervalul $(-\alpha, \alpha)$, deci nu va converge spre un punct de minim.

Propoziția 2.8. *Fie f funcție L regulată și μ -tare convexă. Atunci avem că $\mu \leq L$.*

Demonstrație. Din f L -regulată avem:

$$f(x^*) - f(x) \leq -\frac{1}{2L} \|\nabla f(x)\|^2, \quad \forall x \in \mathbb{R}^d \quad (2.2.12)$$

iar din f μ -tare convexă:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2} \|y - x\|^2, \quad \forall x, y \in \mathbb{R}^d. \quad (2.2.13)$$

Luăm $y = x^*$ și avem că:

$$f(x^*) \geq f(x) + \langle \nabla f(x), x^* - x \rangle + \frac{\mu}{2} \|x^* - x\|^2 \quad (2.2.14)$$

adică

$$f(x^*) - f(x) - \langle \nabla f(x), x^* - x \rangle \geq \frac{\mu}{2} \|x^* - x\|^2 \quad (2.2.15)$$

Folosim (2.2.12) în (2.2.15) și obținem:

$$-\frac{1}{2L} \|\nabla f(x)\|^2 - \langle \nabla f(x), x^* - x \rangle \geq f(x^*) - f(x) - \langle \nabla f(x), x^* - x \rangle \geq \frac{\mu}{2} \|x^* - x\|^2 \quad (2.2.16)$$

Scriem membrul stâng al egalității ca:

$$-\frac{1}{2L} \left(\|\nabla f(x)\|^2 + 2L\langle \nabla f(x), x^* - x \rangle + \|L(x^* - x)\|^2 \right) + \frac{1}{2L} \|L(x^* - x)\|^2$$

ce este echivalent cu:

$$-\frac{1}{2L} \|\nabla f(x) + L(x^* - x)\|^2 + \frac{1}{2L} \|L(x^* - x)\|^2 \quad (2.2.17)$$

Introducem în inegalitatea (2.2.16) și avem următoarea succesiune de inegalități:

$$\begin{aligned} -\frac{1}{2L} \|\nabla f(x) + L(x^* - x)\|^2 + \frac{1}{2L} \|L(x^* - x)\|^2 &\geq \frac{\mu}{2} \|x^* - x\|^2 \iff \\ -\frac{1}{2L} \|\nabla f(x) + L(x^* - x)\|^2 + \frac{L^2}{2L} \|x^* - x\|^2 &\geq \frac{\mu}{2} \|x^* - x\|^2 \iff \\ -\frac{1}{2L} \|\nabla f(x) + L(x^* - x)\|^2 &\geq \frac{\mu}{2} \|x^* - x\|^2 - \frac{L}{2} \|x^* - x\|^2 \iff \\ -\frac{1}{2L} \|\nabla f(x) + L(x^* - x)\|^2 &\geq \frac{\|x^* - x\|^2}{2} (\mu - L) \end{aligned}$$

Cum membrul stâng este negativ, vom avea că și cel drept va fi, de unde rezultă că $\mu - L \leq 0$ de unde concluzia. \square

Acum putem enunța o teoremă de convergență valabilă pentru funcțiile L regulate și μ -tari convexe:

Teorema 2.9. Fie f funcție L regulată și μ -tare convexă. Pentru un punct initial $x_0 \in \mathbb{R}^d$ și pentru rata de învățare $\alpha \in \mathbb{R}$ astfel încât $\frac{1}{L} \geq \alpha > 0$ considerăm sirul $(x_k)_{k \geq 0}$ definit iterativ prin:

$$x^{(k+1)} = x^{(k)} - \alpha \nabla f(x^{(k)}). \quad (2.2.18)$$

Dacă $x^* = \arg \min_{x \in \mathbb{R}^d} f(x)$, atunci:

$$\|x^{(k+1)} - x^*\|^2 \leq (1 - \alpha\mu)^{k+1} \|x^{(0)} - x^*\|^2 \quad (2.2.19)$$

Demonstratie. Din iterată (2.2.18) avem că:

$$\begin{aligned} \|x^{(k+1)} - x^*\|^2 &= \|x^{(k)} - x^* - \alpha \nabla f(x^{(k)})\|^2 \\ &= \|x^{(k)} - x^*\|^2 - 2\alpha \langle \nabla f(x^{(k)}), x^{(k)} - x^* \rangle + \alpha^2 \|\nabla f(x^{(k)})\|^2 \end{aligned}$$

Dar din (2.2.10) avem că:

$$\langle \nabla f(x^{(k)}), x^* - x^{(k)} \rangle \leq f(x^*) - f(x^{(k)}) - \frac{\mu}{2} \|x^* - x^{(k)}\|^2$$

de unde:

$$\begin{aligned} -2\alpha \langle \nabla f(x^{(k)}), x^{(k)} - x^* \rangle &= 2\alpha \langle \nabla f(x^{(k)}), x^* - x^{(k)} \rangle \\ &\leq 2\alpha(f(x^*) - f(x^{(k)})) - \alpha\mu \|x^* - x^{(k)}\|^2 \\ &= -2\alpha(f(x^{(k)}) - f(x^*)) - \alpha\mu \|x^{(k)} - x^*\|^2 \end{aligned}$$

Din (2.2.9) avem că:

$$-\frac{1}{2L} \|\nabla f(x^{(k)})\|^2 \geq f(x^*) - f(x^{(k)})$$

de unde:

$$\|\nabla f(x^{(k)})\|^2 \leq 2L(f(x^{(k)}) - f(x^*))$$

Deci, vom avea că:

$$\begin{aligned} \|x^{(k+1)} - x^*\|^2 &= \|x^{(k)} - x^*\|^2 - 2\alpha \langle \nabla f(x^{(k)}), x^{(k)} - x^* \rangle + \alpha^2 \|\nabla f(x^{(k)})\|^2 \\ &\leq (1 - \alpha\mu) \|x^{(k)} - x^*\|^2 - 2\alpha(f(x^{(k)}) - f(x^*)) + 2\alpha^2 L(f(x^{(k)}) - f(x^*)) \\ &\leq (1 - \alpha\mu) \|x^{(k)} - x^*\|^2 - 2\alpha(1 - \alpha L)(f(x^{(k)}) - f(x^*)) \end{aligned}$$

Din ipoteză avem că $\frac{1}{L} \geq \alpha$, deci $-2\alpha(1 - \alpha L) \leq 0$ de unde putem afirma că

$$\|x^{(k+1)} - x^*\|^2 \leq (1 - \alpha\mu) \|x^{(k)} - x^*\|^2 \quad (2.2.20)$$

Cum $\alpha \leq \frac{1}{L} \leq \frac{1}{\mu}$ (din (2.8)) avem că paranteza va fi mai mare sau egală cu 0, deci putem înlocui succesiv k cu $k - 1, k - 2, \dots, 1, 0$ și vom obține recurența (2.2.19).

□

Așadar, sirul va converge către punctul de minim local x^* care va avea $\nabla f(x^*) = 0$. Algoritmul va continua până când se verifică un criteriu de oprire. Putem considera că și criteriu $|f(x^{(k)})| \leq \epsilon$, unde $\epsilon > 0$ este fixat.

Revenim acum la minimizarea functiei loss, $J(w, b)$. Actualizăm ponderile $w_{ij}^{(l)}$ și bias-urile $b_i^{(l)}$ astfel:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial}{\partial w_{ij}^{(l)}} J(w, b) \quad (2.2.21)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(w, b) \quad (2.2.22)$$

2.3 Stochastic Gradient Descent (SGD)

Calculul gradientului $\nabla J(w, b)$ revine la calcul a n gradiente $\nabla J(w, b, x_i, y_i)$, iar din ecuația (2.2.2) avem că:

$$\nabla J(w, b) = \frac{1}{n} \sum_{i=1}^n \nabla J(w, b, x_i, y_i)$$

Când avem un set mare de date, calculul a n gradiente poate fi costisitor din punct de vedere computațional, motiv pentru care se alege deseori metoda gradientului stochastic (sau SGD - Stochastic Gradient Descent). Pentru aceasta, setul de date este amestecat aleator și împărțit în loturi (batch-uri) pentru care se calculează gradientul.

Se alege un mini lot de $m \leq n$ date, fie acestea $\{(x_{k_1}, y_{k_1}), (x_{k_2}, y_{k_2}), \dots, (x_{k_m}, y_{k_m})\}$ cu $\{k_1, k_2, \dots, k_m\} \subset \{1, 2, \dots, n\}$, pentru care se va calcula gradientul. Se actualizează astfel, succesiv, w, b folosind de fiecare dată media pentru căte un lot de m date. La finalul acestor $\frac{n}{m}$ actualizări spunem că s-a desfășurat o epocă a antrenării. La începutul fiecărei epoci datele sunt amestecate din nou aleator. În general, vom folosi mai multe epoci pentru a ajunge la o acuratețe bună a modelului. Prezentăm în continuare o schiță a algoritmului pentru un set de date de dimensiune n și o dimensiune a batch-ului de m . Precizăm faptul că vom nota cu $\nabla J(w, b)$ gradientul $\nabla_{w,b} J(w, b)$ calculat în raport cu w sau b .

Algoritm SGD: Considerăm datele $(x_1, y_1), \dots, (x_n, y_n)$, unde x_i este input-ul rețelei, y_i este rezultatul dorit iar α_k rata de învățare de la pasul k , pentru $i = \overline{1, n}$. Pentru simplitatea scrierii vom nota în acest algoritm cu $w(k)$ ponderea $w_{tj}^{(l)}(k)$ a neuronului j de pe stratul l în neuronul t de pe stratul $l + 1$ la parcurgerea k și cu $b(k)$ bias-ul $b_t^{(l)}$ al neuronului t de pe stratul $(l + 1)$.

Epoca 1: Amestecăm și renumerotăm datele. Alegem $w(0), b(0)$ aleator. Facem actualizările pe un strat l după t, j , păstrând notația amintită mai devreme, astfel:

$$\begin{aligned}(w(1), b(1)) &= (w(0), b(0)) - \alpha_1 \cdot \frac{1}{m} \sum_{i=1}^m \nabla J(w, b, x_i, y_i) \\(w(2), b(2)) &= (w(1), b(1)) - \alpha_2 \cdot \frac{1}{m} \sum_{i=m+1}^{2m} \nabla J(w, b, x_i, y_i) \\&\dots \\(w(k), b(k)) &= (w(k-1), b(k-1)) - \alpha_k \cdot \frac{1}{m} \sum_{i=n-m+1}^n \nabla J(w, b, x_i, y_i)\end{aligned}$$

unde $k = \frac{n}{m}$.

Epoca 2: Amestecăm și renumerotăm datele. Facem actualizările pe un strat l după t, j , de la ultimul moment din epoca precedentă, astfel:

$$\begin{aligned}(w(k+1), b(k+1)) &= (w(k), b(k)) - \alpha_{k+1} \cdot \frac{1}{m} \sum_{i=1}^m \nabla J(w, b, x_i, y_i) \\(w(k+2), b(k+2)) &= (w(k+1), b(k+1)) - \alpha_{k+2} \cdot \frac{1}{m} \sum_{i=m+1}^{2m} \nabla J(w, b, x_i, y_i) \\&\dots \\(w(2k), b(2k)) &= (w(2k-1), b(2k-1)) - \alpha_{2k} \cdot \frac{1}{m} \sum_{i=n-m+1}^n \nabla J(w, b, x_i, y_i),\end{aligned}$$

Se continuă astfel și cu restul epocilor, până ajungem la convergență.

Remarcă: i) $J(w, b)$ poate fi văzut ca o medie: dacă considerăm variabila aleatoare cu distribuția:

$$(X, Y) \sim \begin{pmatrix} (x_1, y_1) & \dots & (x_n, y_n) \\ \frac{1}{n} & \dots & \frac{1}{n} \end{pmatrix}, \quad (2.3.1)$$

unde $(x_1, y_1), \dots, (x_n, y_n)$ sunt datele de input, atunci:

$$J(w, b) = \mathbb{E}[J(w, b, X, Y)]$$

ii) La fel, gradientul $\nabla J(w, b)$ se poate scrie ca $\mathbb{E}[\nabla J(w, b, X, Y)]$ întrucât $\nabla J(w, b) = \nabla \mathbb{E}[J(w, b, X, Y)] = \mathbb{E}[\nabla J(w, b, X, Y)]$.

Propoziția 2.10. Fie $(X_1, Y_1), \dots, (X_n, Y_n), \dots$ variabile aleatoare, independente și identic distribuite cu (X, Y) și presupunem că

$$\mathbb{E}[|\nabla J(w, b, X, Y)|] < \infty. \quad (2.3.2)$$

Atunci:

$$\frac{1}{k} \sum_{i=1}^k \nabla J(w, b, X_i, Y_i) \xrightarrow[k]{a.s.} \nabla J(w, b)$$

Demonstrație. Din (2.3.2) avem:

$$\nabla J(w, b) = \mathbb{E}[\nabla J(w, b, X, Y)].$$

Mai departe, avem că $\nabla J(w, b, X_1, Y_1), \dots, \nabla J(w, b, X_n, Y_n), \dots$ sunt copii independente și identic distribuite ale lui $\nabla J(w, b, X, Y)$. Din Legea Numerelor Mari rezultă că:

$$\nabla J(w, b) = \mathbb{E}[\nabla J(w, b, X, Y)] = \lim_{k \rightarrow \infty} \frac{1}{k} \sum_{i=1}^k \nabla J(w, b, X_i, Y_i)$$

□

Pentru demonstrația convergenței metodei Stochastic Gradient Descent urmărim ideile din [26] și vom folosi batch-uri de dimensiune 1.

Scopul nostru este să minimizăm $\mathbb{E}[J(w, b, X, Y)] = J(w, b)$. Acest lucru îl facem fixând, la fiecare iterație n , o rată de învățare α_n , o pereche de variabile aleatoare (X_n, Y_n) independentă și identic distribuită cu (X, Y) (definită în (2.3.1)) și actualizând variabilele w, b la pasul $n + 1$ astfel:

$$(w_{n+1}, b_{n+1}) = (w_n, b_n) - \alpha_n \nabla J(w_n, b_n, X_n, Y_n), \quad (2.3.3)$$

unde cu w_n, b_n am notat matricele w, b la momentul n .

Pentru a simplifica notațiile, vom folosi $\theta = (w, b)$ și $Z = (X, Y)$. Așadar, $J(w, b, X_n, Y_n)$ devine $J(\theta, Z)$ și, pentru a nu se confunda cu $J(w, b)$ (care, în urma notatiilor, devine $J(\theta)$), vom scrie $\mathcal{J}(\theta)$ ca fiind $\mathbb{E}[J(w, b, X, Y)] = \mathbb{E}[J(\theta, Z)]$. Astfel, problema revine la a minimiza $\mathcal{J}(\theta)$ și recurența (2.3.3) devine:

$$\theta_{n+1} = \theta_n - \alpha_n \nabla J(\theta_n, Z_n) \quad (2.3.4)$$

Pentru a demonstra convergența metodei avem nevoie de următoarele ipoteze pentru J și \mathcal{J} :

i) Gradientul lui J este mărginit, adică $\exists C < \infty$ astfel încât:

$$\|\nabla_{\theta} J(\theta, Z)\|^2 \leq C, \quad \forall \theta, Z \quad (2.3.5)$$

ii) \mathcal{J} este μ -tare convexă cu $\mu > 0$, adică, conform definiției (2.2.10):

$$\mathcal{J}(\theta) \geq \mathcal{J}(\eta) + \langle \nabla \mathcal{J}(\eta), \theta - \eta \rangle + \frac{\mu}{2} \|\eta - \theta\|^2, \quad \forall \theta, \eta \quad (2.3.6)$$

Teorema 2.11. Presupunem că $J(\cdot, z)$ este diferențiabilă pentru orice z și că J și \mathcal{J} satisfac ipotezele (2.3.5) respectiv (2.3.6). Atunci:

1. Funcția \mathcal{J} are un unic punct de minim, θ^* ;
2. Pentru fiecare $n \geq 0$ definim:

$$d_n = \mathbb{E} [\|\theta_n - \theta^*\|^2]. \quad (2.3.7)$$

Atunci:

$$d_{n+1} \leq (1 - \alpha_n \mu) d_n + \alpha_n^2 C \quad (2.3.8)$$

3. Pentru oricare $\epsilon > 0$ $\exists \alpha > 0$ astfel încât, dacă $\alpha_n = \alpha$, avem:

$$\overline{\lim}_{n \rightarrow \infty} \mathbb{E} [\|\theta_{n+1} - \theta^*\|^2] \leq \epsilon \quad (2.3.9)$$

4. Dacă considerăm sirul α_n astfel încât:

$$\alpha_n \rightarrow 0 \text{ și } \sum_{n \geq 1} \alpha_n = \infty \quad (2.3.10)$$

Atunci $d_n \rightarrow 0$, de unde $\lim_{n \rightarrow \infty} \theta_n = \theta^*$, unde convergența este cea L^2 de variabile aleatoare.

Demonstratie. 1. Existența și unicitatea punctului de minim θ^* sunt garantate de presupunerea convexității tari a funcției \mathcal{J} , folosind [1].

2. Avem că:

$$d_{n+1} = \mathbb{E} \left[\|\theta_{n+1} - \theta^*\|^2 \right] = \mathbb{E} \left[\|\theta_n - \theta^* - \alpha_n \nabla J(\theta_n, Z_n)\|^2 \right] \quad (2.3.11)$$

$$= \mathbb{E} \left[\|\theta_n - \theta^*\|^2 \right] + \alpha_n^2 \mathbb{E} \left[\|\nabla J(\theta_n, Z_n)\|^2 \right] - 2\alpha_n \mathbb{E} \left[\langle \theta_n - \theta^*, \nabla J(\theta_n, Z_n) \rangle \right] \quad (2.3.12)$$

Am definit mai devreme $\mathcal{J}(\theta) = \mathbb{E}[J(\theta, Z)]$ de unde:

$$\begin{aligned} \mathbb{E}[\nabla \mathcal{J}(\theta)] &= \mathbb{E}[\nabla \mathbb{E}[J(\theta, Z)]] = \mathbb{E}[\mathbb{E}[\nabla J(\theta, Z)]] \\ &= \mathbb{E}[\nabla J(\theta, Z)] \end{aligned}$$

Deci, putem spune că:

$$\begin{aligned} \mathbb{E} \left[\langle \theta_n - \theta^*, \nabla J(\theta_n, Z_n) \rangle \right] &= \mathbb{E} \left[\langle \theta_n - \theta^*, \nabla \mathcal{J}(\theta_n) \rangle \right] \\ &= -\mathbb{E} \left[\langle \theta^* - \theta_n, \nabla \mathcal{J}(\theta_n) \rangle \right] \\ &\stackrel{(2.3.6)}{\geq} \mathbb{E} \left[\mathcal{J}(\theta_n) - \mathcal{J}(\theta^*) + \frac{\mu}{2} \|\theta_n - \theta^*\|^2 \right] \end{aligned}$$

Și cum θ^* este punctul de minim al funcției \mathcal{J} rezultă că $\mathcal{J}(\theta_n) - \mathcal{J}(\theta^*) \geq 0$. Deci:

$$\mathbb{E} \left[\mathcal{J}(\theta_n) - \mathcal{J}(\theta^*) + \frac{\mu}{2} \|\theta_n - \theta^*\|^2 \right] \geq \frac{\mu}{2} \mathbb{E} \left[\|\theta_n - \theta^*\|^2 \right]. \quad (2.3.13)$$

Folosind inegalitatea (2.3.13) în (2.3.12) obținem:

$$\begin{aligned} d_{n+1} &\leq \mathbb{E} \left[\|\theta_n - \theta^*\|^2 \right] + \alpha_n^2 \mathbb{E} \left[\|\nabla J(\theta_n, Z_n)\|^2 \right] - 2\alpha_n \frac{\mu}{2} \mathbb{E} \left[\|\theta_n - \theta^*\|^2 \right] \\ &\stackrel{(2.3.5)}{\leq} d_n + \alpha_n^2 C - 2\alpha_n \frac{\mu}{2} d_n = (1 - \alpha_n \mu) d_n + \alpha_n^2 C \end{aligned}$$

3. Considerăm $\alpha_n = \alpha$ în (2.3.8) și scădem $\alpha \frac{C}{\mu}$ în ambii membrii:

$$d_{n+1} - \alpha \frac{C}{\mu} \leq (1 - \alpha \mu)(d_n - \alpha \frac{C}{\mu}) \quad (2.3.14)$$

Dacă alegem un $\alpha < \frac{1}{\mu}$ vom avea că $1 - \alpha \mu > 0$ și, dacă aplicăm funcția parte pozitivă ($x \rightarrow x_+$), care este crescătoare, obținem:

$$\left(d_{n+1} - \alpha \frac{C}{\mu} \right)_+ \leq (1 - \alpha \mu) \left(d_n - \alpha \frac{C}{\mu} \right)_+$$

unde, dacă iterăm după $k \geq 1$, rezultă:

$$\left(d_{n+k} - \alpha \frac{C}{\mu} \right)_+ \leq (1 - \alpha \mu)^k \left(d_n - \alpha \frac{C}{\mu} \right)_+$$

Dacă luăm $k \rightarrow \infty$ avem $\overline{\lim}_{k \rightarrow \infty} \left(d_k - \alpha \frac{C}{\mu} \right)_+ = 0$ de unde rezultă propoziția (2.3.9), unde $\alpha < \frac{1}{\mu}$ și $\alpha < \epsilon \frac{\mu}{C}$.

4. Scădem în ambii membrii din (2.3.8) un $\epsilon > 0$ și obținem:

$$\begin{aligned} d_{n+1} - \epsilon &\leq (1 - \alpha_n \mu) d_n - \epsilon + \epsilon \alpha_n \mu - \epsilon \alpha_n \mu + \alpha_n^2 C \\ &= (1 - \alpha_n \mu)(d_n - \epsilon) + \alpha_n(\alpha_n C - \mu \epsilon) \end{aligned}$$

Cum $\alpha_n \rightarrow 0$ rezultă că $\exists n_0$ astfel încât pentru $n \geq n_0$ vom avea $\alpha_n < \frac{\mu \epsilon}{C}$ de unde $\alpha_n(\alpha_n C - \mu \epsilon) \leq 0$. Așadar, pentru $n \geq n_0$ inegalitatea anterioară devine:

$$d_{n+1} - \epsilon \leq (1 - \alpha_n \mu)(d_n - \epsilon)$$

Alegem un n_1 pentru care $\forall n \geq n_1 \alpha_n < \frac{1}{\mu}$ și aplicăm asupra ultimei inegalități, ca mai devreme, funcția parte pozitivă. Rezultă că pentru $n \geq \max\{n_0, n_1\}$ avem:

$$(d_{n+1} - \epsilon)_+ \leq (1 - \alpha_n \mu)(d_n - \epsilon)_+$$

unde, dacă iterăm după $k \geq 1$, rezultă:

$$(d_{n+k} - \epsilon)_+ \leq \prod_{l=n}^{n+k-1} (1 - \alpha_l \mu)(d_l - \epsilon)_+ \quad (2.3.15)$$

Demonstrăm că $\lim_{k \rightarrow \infty} \prod_{l=n}^{n+k-1} (1 - \alpha_l \mu) = 0$. Stim că $\ln(1 - x) \leq -x$ pentru orice $x \in (0, 1)$.

Cum $0 < \alpha_n < \frac{1}{\mu}$ rezultă că putem scrie:

$$0 \leq \prod_{l=n}^{n+k-1} (1 - \alpha_l \mu) = e^{\sum_{l=n}^{n+k-1} \ln(1 - \alpha_l \mu)} \leq e^{\sum_{l=n}^{n+k-1} (-\alpha_l \mu)} \xrightarrow[k \rightarrow \infty]{(2.3.10)} e^{-\infty} = 0 \quad (2.3.16)$$

Folosind (2.3.16) în (2.3.15) obținem că:

$$\lim_{k \rightarrow \infty} (d_k - \epsilon)_+ = 0, \text{ unde } \epsilon \text{ a fost ales aleator.}$$

Așadar, putem afirma că $d_n \xrightarrow{n \rightarrow \infty} 0$, de unde $\lim_{n \rightarrow \infty} \theta_n = \theta^*$, unde convergența este cea L^2 de

variabile aleatoare.

□

Am văzut aşadar în această teoremă că metoda Stochastic Gradient Descent converge la un punct de minim pentru θ având presupunerile inițiale legate de mărginirea gradientului funcției loss pentru fiecare pereche de date și μ -tare convexitatea funcției loss totale. În demonstrație s-a pornit de la iterația

$$\theta_{n+1} = \theta_n - \alpha_n \nabla J(\theta_n, Z_n),$$

adică cu batch-uri de dimensiune 1. În general, trebuie testate diferite valori pentru mărimea batch-ului pentru a vedea când se obțin rezultatele dorite mai repede, însă pentru valori extreme (foarte mici sau mari) se pot obține de multe ori rezultate instabile.

2.4 Backpropagation

Prin algoritmul de backpropagation vrem să calculăm derivatele parțiale $\partial J / \partial w_{ij}^{(l)}$ și $\partial J / \partial b_i^{(l)}$ pentru a putea actualiza ponderile $w_{ij}^{(l)}$, respectiv biasurile $b_i^{(l)}$. Acest algoritm ne ajută să determinăm cum se modifică rezultatul funcției cost la o modificare a ponderilor și bias-urilor. [15]

Introducem notația $\delta_j^{(l)}$ pe care o numim eroarea neuronului j în stratul l și o definim astfel:

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}}$$

Pentru a vectoriza calculele ulterioare, definim produsul Hadamard. [14]

Definiția 2.12. Pentru două matrici A și B de aceeași dimensiune $m \times n$ avem că produsul Hadamard al lui A și B este definit prin:

$$[A \circ B]_{ij} = [A]_{ij} [B]_{ij}, \quad 1 \leq i \leq m, 1 \leq j \leq n$$

Vom enunță în continuare o serie de rezultate referitoare la eroarea $\delta_j^{(l)}$, urmărind [15]:

Propoziția 2.13. Dacă L este stratul de output atunci componentele erorii $\delta_j^{(L)}$ pe acest strat sunt date de:

$$\delta_j^{(L)} = \frac{\partial J}{\partial h_j^{(L)}} f'(z_j^{(L)}) \quad (2.4.1)$$

sau, în formă matriceală, de:

$$\delta^{(L)} = \nabla_h J \circ f'(z^{(L)}) \quad (2.4.2)$$

Demonstrație. Avem din definiție că

$$\delta_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$$

Folosind regula lanțului putem exprima derivata parțială de mai sus astfel:

$$\delta_j^{(L)} = \sum_k \frac{\partial J}{\partial h_k^{(L)}} \frac{\partial h_k^{(L)}}{\partial z_j^{(L)}}$$

Cum $h_k^{(L)} = f(z_k^{(L)})$ vom avea că derivatele parțiale unde $k \neq j$ vor fi 0, deci rămâne:

$$\delta_j^{(L)} = \frac{\partial J}{\partial h_j^{(L)}} \frac{\partial h_j^{(L)}}{\partial z_j^{(L)}}$$

Întrucât $h_j^{(L)} = f(z_j^{(L)})$ putem scrie al doilea termen ca $f'(z_j^{(L)})$ de unde rezultă concluzia (2.4.1).

□

În Backpropagation vrem să vedem cum se propagă eroare de la ultimul strat până la input. Următoarea propozitie ne oferă o relație de recurență între erorile de pe două straturi consecutive.

Propoziția 2.14. Relația dintre eroarea de pe un strat l în funcție de eroarea de pe stratul $l + 1$ este dată de:

$$\delta^{(l)} = ((w^{(l)})^T \delta^{(l+1)}) \circ f'(z^{(l)}) \quad (2.4.3)$$

Demonstrație. Exprimăm $\delta_j^{(l)}$ în funcție de $\delta_k^{(l+1)}$ astfel:

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} \quad (2.4.4)$$

$$= \sum_k \frac{\partial J}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \quad (2.4.5)$$

$$= \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \quad (2.4.6)$$

Avem că:

$$\begin{aligned} z_k^{(l+1)} &= \sum_s w_{ks}^{(l)} h_s^{(l)} + b_k^{(l)} \\ &= \sum_s w_{ks}^{(l)} f(z_s^{(l)}) + b_k^{(l)} \end{aligned}$$

Derivând în funcție de $z_j^{(l)}$, unde $w_{kj}^{(l)} f(z_j^{(l)})$ este un termen din suma de mai sus, obținem:

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = w_{kj}^{(l)} f'(z_j^{(l)}) \quad (2.4.7)$$

Înlocuim (2.4.7) în (2.4.6) și avem că:

$$\delta_j^{(l)} = \sum_k w_{kj}^{(l)} \delta_k^{(l+1)} f'(z_j^{(l)})$$

ce reprezintă (2.4.3) scris pe componente. \square

Propoziția 2.15.

$$\frac{\partial J}{\partial b_j^{(l)}} = \delta_j^{(l+1)} \quad (2.4.8)$$

Demonstrație. Folosind regula lanțului, putem scrie pe $\frac{\partial J}{\partial b_j^{(l)}}$ ca:

$$\begin{aligned} \frac{\partial J}{\partial b_j^{(l)}} &= \sum_k \frac{\partial J}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial b_j^{(l)}} \\ &= \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial b_j^{(l)}} \end{aligned}$$

Și cum:

$$z_k^{(l+1)} = \sum_s w_{ks}^{(l)} h_s^{(l)} + b_k^{(l)}$$

rezultă că

$$\frac{\partial z_k^{(l+1)}}{\partial b_j^{(l)}} = \begin{cases} 0 & , \text{dacă } k \neq j \\ 1 & , \text{dacă } k = j \end{cases}$$

Deci,

$$\frac{\partial J}{\partial b_j^{(l)}} = \delta_j^{(l+1)}$$

□

Propoziția 2.16.

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = h_j^{(l)} \delta_i^{(l+1)} \quad (2.4.9)$$

Demonstrație. Procedăm ca la demonstrațiile propozițiilor 2.14 și 2.15. Avem mai întâi, din regula lanțului că:

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \sum_k \frac{\partial J}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial w_{ij}^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial w_{ij}^{(l)}} \quad (2.4.10)$$

Stim că:

$$z_k^{(l+1)} = \sum_s w_{ks}^{(l)} h_s^{(l)} + b_k^{(l)}$$

Derivăm în raport cu $w_{ij}^{(l)}$ și avem:

$$\frac{\partial z_k^{(l+1)}}{\partial w_{ij}^{(l)}} = \begin{cases} h_j^{(l)} & , \text{dacă } k = i \\ 0 & , \text{dacă } k \neq i \end{cases} \quad (2.4.11)$$

Înlocuind (2.4.11) în (2.4.10) obținem:

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \delta_i^{(l+1)} h_j^{(l)}$$

□

Așadar, din propozițiile (2.13)-(2.16) putem determina $\partial J / \partial w_{ij}^{(l)}$ și $\partial J / \partial b_i^{(l)}$ recurrent de la ultimul strat la primul și înlocui în problema gradientului descendente.

2.5 Antrenarea MNIST folosind ANN

O categorie de probleme în care rețelele neurale artificiale (ANN) sunt foarte folosite este cea a problemelor de clasificare de imagini.

Vom folosi în continuare setul de date MNIST (*Modified National Institute of Standards and Technology*) [6] care conține 70.000 de imagini de dimensiune 28x28, cu cifre scrise de mână, dintre care 60.000 vor fi folosite pentru antrenare iar 10.000 pentru testarea modelului. Au fost folosite pentru antrenare batch-uri de 500 iar pentru testare batch-uri de 100. Imaginele sunt normalizate, centrate și sunt de tipul celor din figura 2.5.1. Ele sunt mai întâi vectorizate, astfel ajungând de la o matrice 28x28, la un vector de lungime 784 care va fi transmis ca input în rețea. Output-ul este reprezentat de clasa fiecărei imagini, adică o cifră de la 0 la 9.



Figure 2.5.1: Exemple imagini din setul de date MNIST (preluat din [12])

Am folosit pentru antrenare două arhitecturi propuse de Yann LeCun în [12].

- O rețea neurală cu un strat ascuns, de 300 unități, funcțiile de activare ReLU și Softmax și funcția loss Cross Entropy rezultă într-o acuratețe maximă de 98.17% (în epoca 28) după 50 de epoci de antrenare. Graficele pentru funcția loss din timpul antrenării și testării, cât și evoluția acurateții la antrenare și testare se găsesc în figura 2.5.2.

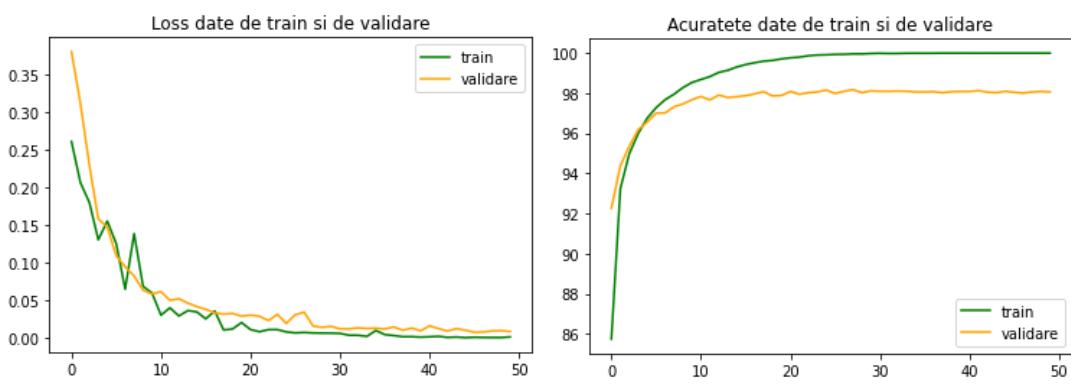


Figure 2.5.2: Analiză loss și acuratețe rețea cu un strat ascuns

- ii) O rețea neurală cu două straturi ascunse, de 300 și respectiv 100 de unități, funcția de activare ReLU după stratul de input și cel ascuns și funcția loss Cross Entropy rezultă într-o acuratețe maximă de 98.18% (în epoca 23) după 50 de epoci de antrenare. Graficele pentru funcția loss din timpul antrenării și testării cât și evoluția acurateții la antrenare și testare se găsesc în figura 2.5.3.

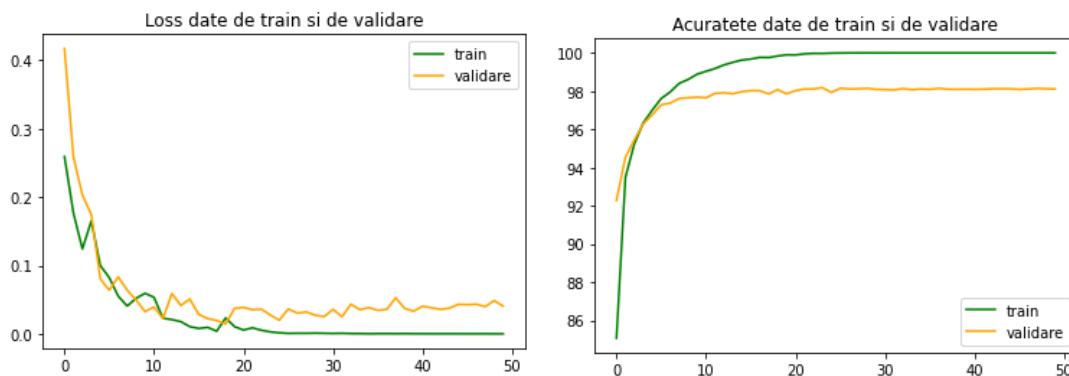


Figure 2.5.3: Analiză loss și acuratețe rețea cu două straturi ascunse

Codul sursă se găsește în anexa 5.2.

3 Rețele Neurale Convoluționale

Așa cum am văzut în secțiunea anterioară, putem folosi rețele neurale pentru clasificarea imaginilor. În cazul setului de date MNIST, am avut imagini de dimensiunea 28x28, care au putut fi vectorizate pentru a avea un input unidimensional, de 784. În cazul unor imagini de rezoluție mai mare, de exemplu 1024x1024 (imagini folosite în secțiunea 4), dacă am aplica același procedeu, am obține un input de dimensiune 1.048.576. Însă, o rețea de dimensiuni mai mari precum aceasta este mai predispusă la overfitting [12; 24], motiv pentru care au început să se folosească rețelele neurale convoluționale. Overfitting-ul se întâmplă când o rețea nu mai poate învăța eficient; are mai multe cauze, printre care: o rețea cu prea mulți parametri sau un timp prea mare de antrenare care face rețeaua incapabilă să distingă generalități din datele de test [17].

În continuare vom folosi prescurtarea CNN (*Convolutional Neural Networks*) pentru a ne referi la rețelele neurale convoluționale și ANN (*Artificial Neural Networks*) pentru rețelele neurale artificiale prezentate în secțiunea anterioară.

3.1 Arhitectură

CNN sunt formate din straturi convoluționale, straturi de pooling și straturi fully-connected, precum cele din ANN. Vom vorbi în continuare de rețelele folosite pentru clasificări de imagini. Ele primesc ca input o imagine sub formă de tensor cu 3 canale, câte unul pentru fiecare culoare RGB (*Red, Green, Blue*) sau 1 canal în cazul imaginilor alb-negru, pe care se găsesc valorile pixelilor corespunzători fiecărei culori.

Așa cum este detaliat și în [15], avem că în CNN, în straturile convoluționale, neuronii sunt legați printr-o pondere doar de o secțiune mică din stratul anterior, spre deosebire de ANN unde fiecare neuron era legat de toți neuronii din stratul anterior. De exemplu, în figura 3.1.1 putem observa outputul unei secțiuni 5x5, numită câmp receptiv, dintr-o imagine din setul MNIST după trecerea printr-un strat convoluțional.

Se parurge astfel toată matricea, reducând dimensiunea imaginii în acest caz la 24x24, din

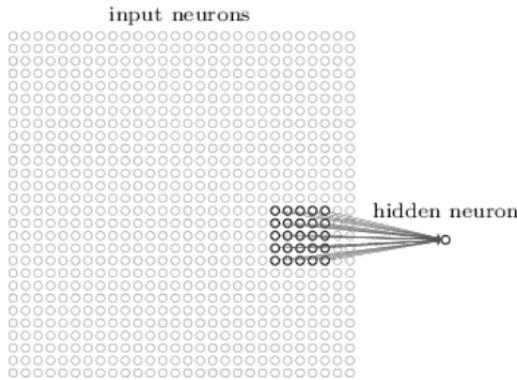


Figure 3.1.1: Legătura dintre două straturi consecutive (preluat din [15])

28×28 . Pentru a determina valorile neuronilor de output din această parcurgere se face operația de conoluție (pe care o vom detalia mai târziu) dintre matricea formată din secțiunea selectată de pe stratul de input și o matrice numită filtru, care are ponderile ce trebuie învățate prin antrenarea CNN, la care se aplică o funcție de activare (spre exemplu ReLU sau sigmoid, pe care le-am folosit și în secțiunile anterioare).

Filtrele sunt comune pentru fiecare neuron obținut din câmpul receptiv de pe stratul anterior și, prin antrenare, acestea vor învăța să distingă anumite caracteristici dintr-o imagine, relevante pentru rezultatul final.

La fiecare strat conoluțional putem avea mai multe filtre ce sunt aplicate matricilor din input. Cum filtrele sunt comune, observăm o scădere considerabilă a numărului de parametrii ce vor trebui învățați. De exemplu, pentru imaginile din setul MNIST avem dimensiunile 28×28 , deci 784 neuroni de input. La prima antrenare din subsecțiunea 2.5 aveam 300 de neuroni pe stratul ascuns cu 300 bias-uri, fiecare fiind legat de toți 784 anteriori, adică în total 235.500 de parametrii. În cazul conoluțional putem avea, de exemplu, 20 filtre de dimensiunea 5×5 , adică 25 parametrii plus bias-ul care este comun (arhitectura LeNet5 [12]), deci în total $20 \times 26 = 520$ parametrii, de 450 de ori mai puțin. Acest lucru contribuie la reducerea timpului total de antrenare și va scăda șansa de overfit.

În continuare vom prezenta abordarea din [28] legată de operațiile folosite în CNN.

3.2 Convoluție

Din punct de vedere matematic, conoluția este o operație care preia două funcții, fie acestea f și g , și returnează o a treia funcție, $f * g$, care exprimă modul în care forma uneia este modificată de celalaltă. Termenul de conoluție se referă atât la rezultat, cât și la operație. Aceasta este definită ca integrala produsului dintre cele două funcții, pe tot domeniul de definiție, după ce una este inversată și deplasată [36]. Dacă avem $f : \mathbb{R} \rightarrow \mathbb{R}$ mărginită și $g : \mathbb{R} \rightarrow \mathbb{R}_+$ intergrabilă cu $\int_{-\infty}^{+\infty} g(x)dx = 1$ atunci definim operația de conoluție între f și g astfel:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(x)g(t+x)dx$$

Conoluția $(f * g)(t)$ poate fi interpretată și ca fiind $\mathbb{E}[f(t + X)]$, unde $X \sim g$, acest lucru sugerându-ne faptul că rezultatul este o regularizare a graficului lui f , ținând cont de graficul lui g .

Pentru a ilustra mai bine operația de conoluție în cazul rețelelor neurale, vom urmări exemplul din figura 3.2.1.

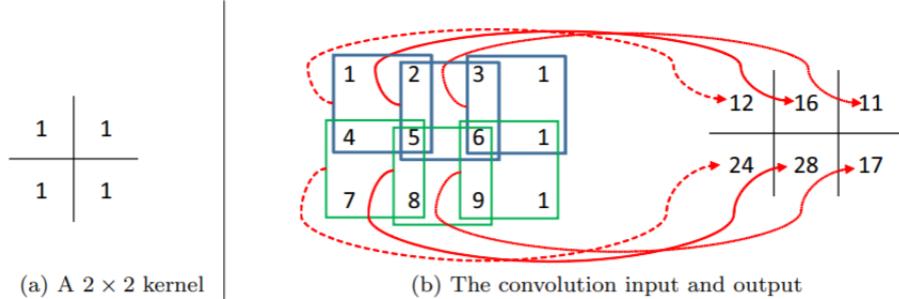


Figure 3.2.1: Operația de conoluție folosind un filtru 2×2 și o imagine 3×4 (preluat din [28])

Conoluția constă în suprapunerea, peste imaginea de input, a filtrului ales (numit și kernel) și însumarea produselor pe componente din secțiunea respectivă. Spre exemplu, pe prima poziție (când avem filtrul peste colțul din stânga sus) vom obține $1 \times 1 + 1 \times 2 + 1 \times 4 + 1 \times 5 = 12$. În continuare mutăm filtru cu un pixel mai jos iar următoarea valoare va fi $1 \times 4 + 1 \times 5 + 1 \times 7 + 1 \times 8 = 24$. Se continuă până când filtrul va fi așezat în colțul din dreapta jos. În acest exemplu am mutat la fiecare pas filtrul cu câte un pixel, însă putem folosi și alte valori. Această variabilă poartă denumirea de *stride*.

Pentru tensorii de ordin 3 se procedează asemănător. Dacă am avea, de exemplu, pe stratul l un tensor de dimensiune $H^l \times W^l \times D^l$ atunci și filtrul va fi un tensor de ordin 3 cu același număr de canale, $H \times W \times D^l$. Ca mai devreme, se suprapune filtrul acesta peste input, începând cu colțul din stânga sus și se calculează suma celor $H \cdot W \cdot D^l$ produse. Apoi mutăm filtrul de sus în jos, de la stânga la dreapta. Output-ul într-un caz de acest tip, dacă avem un singur filtru, va fi un tensor de dimensiunea $(H^l - H + 1) \times (W^l - W + 1) \times 1$.

De obicei, la un strat convecțional avem nevoie de mai multe filtre ce vor trebui învățate. Astfel, dacă avem ca mai devreme pe stratul l un tensor de dimensiune $H^l \times W^l \times D^l$ și filtrele vor fi într-un tensor de ordin 4 de dimensiune $H \times W \times D^l \times D$, unde D reprezintă numărul de filtre, output-ul va fi de forma $(H^l - H + 1) \times (W^l - W + 1) \times D$.

În unele cazuri vrem ca output-ul să fie de aceeași dimensiune cu input-ul în urma aplicării convecției. Pentru a face acest lucru putem aplica operația de *padding* (sau inserare) prin care se adaugă linii sau coloane de valoare 0 care nu vor afecta rezultatul final. De exemplu, dacă adăugam la cazul anterior $\left[\frac{H-1}{2}\right]$ linii deasupra primei linii și $\left[\frac{H}{2}\right]$ linii sub ultima linie, $\left[\frac{W-1}{2}\right]$ coloane în stânga primei coloane și $\left[\frac{W}{2}\right]$ coloane în dreapta ultimei, vom obține în urma convecției un output de dimensiune $H \times W \times D$, unde $[.]$ reprezintă funcția partea întreagă. [28]

Tensorul obținut în urma operației de convecție poartă denumirea și de hartă de activare ("*activation map*").

3.3 Filtre

Vom folosi site-ul [33] pentru a vedea cum diverse filtre transformă o imagine prin operația de convoluție:

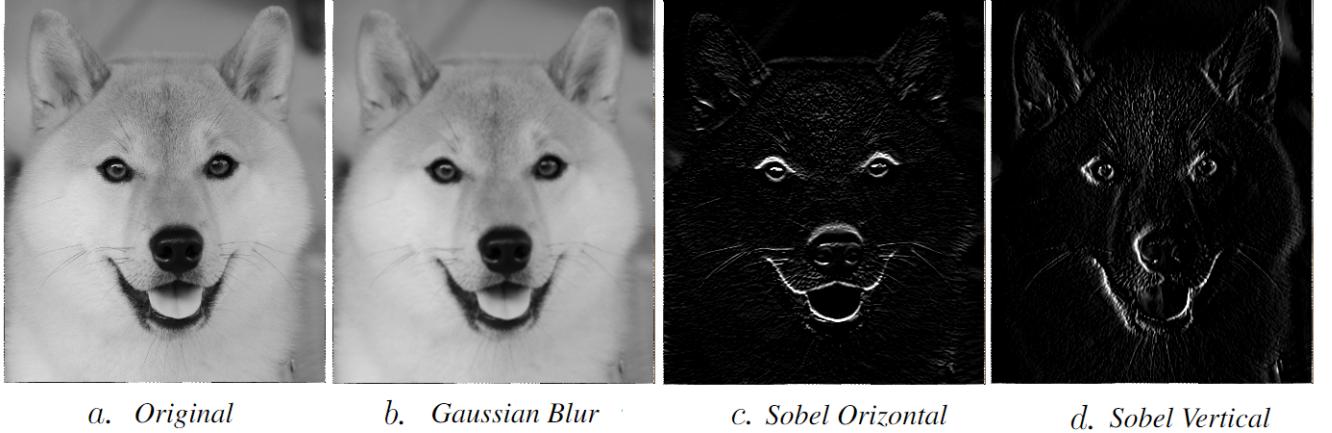


Figure 3.3.1: Diferite filtre aplicate pe o imagine

În figura 3.3.1 avem o poză alb-negru (*a*) și rezultatele convoluției acesteia aplicând trei filtre, (*b*, *c*, *d*). Pentru *b* am folosit un filtru 3×3 de estompare Gaussiană, adică de forma $G = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$, pentru a reduce nivelul de zgomot și a netezi imaginea, ignorând detaliile neglijabile.

În figura *c* este folosit un filtru cu matricea $K = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$. Vrem să vedem ce se întâmplă când pe poziția (x, y) se află o margine orizontală (adică valorile pixelilor de pe pozițiile $(x, y - 1)$ și $(x, y + 1)$ diferă printr-o cantitate mare). Presupunem că avem următoarea matrice, unde *a* și *c* sunt foarte depărtate:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \begin{bmatrix} a & a & a \\ b & b & b \\ c & c & c \end{bmatrix} = a + 2a + a - c - 2c - c = 4(a - c), \text{ care este o valoare mare.}$$

Așadar, punctele de acest tip vor avea o magnitudine mare, așa cum putem observa și în figură, ce duce la evidențierea marginilor orizontale.

Asemănător, folosind filtrul $K^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ se vor accentua marginile verticale, așa cum se observă și în figura *d*. Aceste două matrici, K și K^T , se numesc și operatori Sobel.

Adăugarea unui bias la aceste operații convolutionale poate face rezultatul pozitiv pe margini orizontale (verticale) într-o anumite direcție și negativ în rest. Dacă aceste tipuri de straturi sunt urmate de unul unde se aplică funcția de activare ReLU, rezultatul acestora va putea detecta dacă un anumit fel de margine este prezentă sau nu.

Dacă înlocuim filtrele Sobel cu altele rezultate din timpul antrenării, acestea pot învăța să activeze margini în unghiuri diferite și, ulterior, prin alăturarea mai multor straturi convolutionale, pot detecta grupări de margini care compun o anumită formă sau obiect.

3.4 Pooling

Așa cum am văzut în subsecțiunea 3.2, straturile de conveție pot reduce dimensionalitatea input-ului, dar nu cu foarte mult având în vedere că filtrele sunt, în general, destul de mici (2×2 , 3×3 , 5×5 sunt cele mai folosite). Pentru acest lucru se folosesc straturile de pooling, acestea având ca scop o reducere graduală a input-ului și implicit numărul de parametri și complexitatea computațională, fără să afecteze acuratețea.

Straturile de pooling acționează asupra hărților de activare din input și le micșorează prin aplicarea unei funcții (de obicei "MAX" sau "AVG") pe secțiuni de o dimensiune dată din acesta. Dacă alegem, de exemplu, un strat de max pooling cu un kernel de dimensiune 2×2 , acesta se va deplasa câte 2 pixeli la fiecare pas și va selecta fiecare valoare maximă din secțiunea curentă. Astfel, se reduce dimensiunea cu 25%, dar păstrând în același timp numărul de canale neschimbăt.[17]

4 Aplicații

4.1 Rețele Neurale Convoluționale în analizarea imaginilor medicale

În continuare prezentăm o aplicație a rețelelor neurale convoluționale în detecția unor boli precum pneumonia, având la dispoziție pentru antrenare un set de radiografii din care modelul învăță caracteristicile importante în identificarea acesteia. Aplicația a fost dezvoltată urmărind tutorialul "Deep Learning with PyTorch for Medical Image Analysis" ținut de Jose Portilla et all [18].

Am extras din setul de date ChestX-ray8 [27; 34] radiografiile care au fost etichetate cu pneumonie sau fără, adică 26.684. În setul de date radiografiile sunt etichetate cu 1 pentru pacienții pozitivi cu pneumonia și 0 altfel. În cazul pozitiv sunt date și coordonatele dreptunghiului în care este prezentă pneumonia pe radiografie, însă nu vom folosi această informație în continuare. În partea de procesare a datelor am normalizat imaginile, aducând valorile pixelilor din intervalul $[0, 255]$ în $[0, 1]$, am micșorat dimensiunea lor din 1024×1024 în 224×224 pentru a reduce complexitatea modelului.

S-a constatat că prelucrări ale imaginilor înainte de antrenare (precum rotiri, translații, scalări, normalizări) duc la rezultate mai bune ale acesteia și ajută modelul să identifice și cazuri mai generale. Aceste prelucrări poartă denumirea de *data augmentation*. În figura 4.1.1 sunt exemple de imagini din setul de date înainte și după aceste transformări.

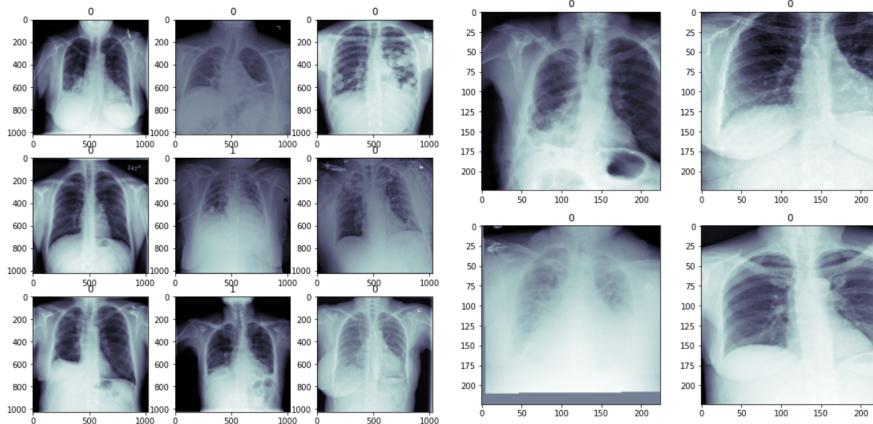


Figure 4.1.1: Exemple de imagini din setul de date înainte și după transformări

Am împărțit cele 26.684 radiografi în 24.000 imagini pentru antrenare (dintre care 18.593 negativi și 5407 pozitivi) și 2684 pentru testare. Întrucât setul de date este destul de dezechilibrat, fapt ce poate duce la o tendință în a prezice că o imagine este negativă, vom amplifica ponderea imaginilor pozitive cu un factor de 3 (deoarece sunt de aproximativ 3 ori mai multe imagini negative decât pozitive).

În general, CNN se comportă mai bine la seturi mari de date iar, când nu avem la dispoziție aşa ceva, putem apela la o tehnică numită transfer de învățare (*transfer learning*) unde, în loc să inițializăm rețea cu parametrii aleatori la început, se vor folosi parametrii învățați dintr-o antrenare pe un set mare de date. De cele mai multe ori se folosesc antrenările de pe setul de date ImageNet [5]. Acesta cuprinde 1.2 milioane de imagini, împărțite în 1000 clase.

Vom prezenta pe scurt arhitecturile folosite pentru antrenări cât și rezultatele obținute la fiecare dintre ele, și anume: AlexNet [10], ResNet18 [8] și DenseNet201 [9] întrucât s-au dovedit a fi eficiente la astfel de clasificări în lucrări de specialitate precum [19]. Antrenările au fost făcute pe platforma Google Colab.

4.1.1 AlexNet

Este formată din cinci straturi covoluționale, cu trei straturi de pooling, două straturi complet conectate și un strat Softmax pentru prezicerea output-ului sub formă de probabilitate. Dimensiunea inputut-ului este de $227 \times 227 \times 3$ (3-ul provine de la numărul de canale pentru culoare, în cazul nostru am modificat la 1 întrucât avem imagini alb-negru), iar output-ul avea inițial o dimensiune de 1000 (de la clasificarea pentru ImageNet) însă a fost modificat cu 1 (deoarece vrem un rezultat sub formă de probabilitate ca pacientul să aibă pneumonie). Modelul are în total 57 milioane de parametrii ce trebuie învățați. O vizualizare mai detaliată a arhitecturii se poate observa în figura 4.1.2.

Pentru antrenare am folosit batch-uri de dimensiune 16, iar ca funcție loss Binary Cross Entropy. După 35 de epoci de antrenare obținem următorii indicatori:

- Acuratețe: 76,97% $\left(\frac{\# \text{ imagini prezise corect}}{\# \text{ total de imagini}} \right)$

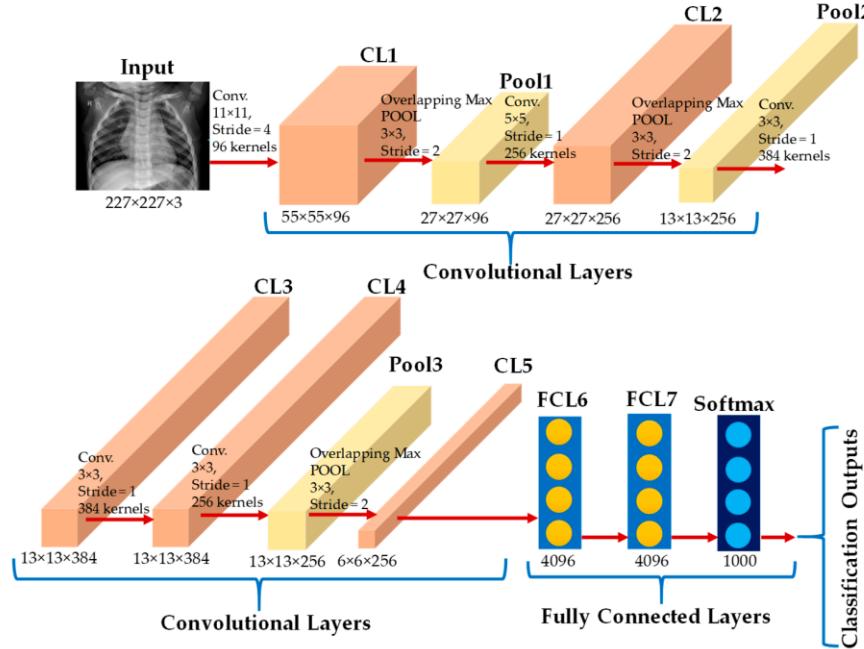


Figure 4.1.2: Arhitectura AlexNet (preluat din [19])

- Precizie: $49,38\% \left(\frac{\# \text{ corect pozitive}}{\# \text{ corect pozitive} + \# \text{ fals pozitive}} \right)$
- Recall: $85,95\% \left(\frac{\# \text{ corect pozitive}}{\# \text{ corect pozitive} + \# \text{ fals negative}} \right)$
- Scor F1: $62,72\% \left(2 \cdot \frac{\text{precizie} \cdot \text{recall}}{\text{precizie} + \text{recall}} \right)$
- Matricea de confuzie: $\begin{pmatrix} 1546 & 533 \\ 85 & 520 \end{pmatrix} \begin{pmatrix} \# \text{ corect pozitive} & \# \text{ fals pozitive} \\ \# \text{ fals negative} & \# \text{ corect negative} \end{pmatrix}$

Acești indicatori menționați sunt utili în funcție de ce ne dorim să realizeze modelul nostru.

Observăm că, deși precizia are o valoare mică, recall-ul este destul de mare, 85,97%. Aceasta ne spune mai exact că modelul reușește să detecteze majoritatea imaginilor în care pneumonia e prezentă, cu costul mai multor imagini fals diagnosticate pozitiv. În cazul analizei imaginilor medicale vrem să îmbunătățim în special acest indicator.

4.1.2 ResNet18

Conceptul de rețea neurală reziduală a apărut pentru prima dată într-un articol publicat de He K et all [8] în cadrul competiției ILSVRC 2015. Autorii au propus un tip de arhitectură ce facilitează o antrenare pe mai multe straturi și are rezultate progresiv mai bune adăugând straturi în plus. Până atunci s-a evitat acest lucru deoarece au început să apară probleme la propagarea gradientului în straturile initiale. Ideea de bază a acestor rețele este în adăugarea unor conexiuni omise (*skipped connections*) cu ajutorul cărora rețeaua va trebui să învețe doar un reziduu, ce e mai eficient computațional. Modelul are în total 11.2 milioane de parametrii ce trebuie învățați. În figura 4.1.3 putem vedea mai bine straturile propriu-zise.

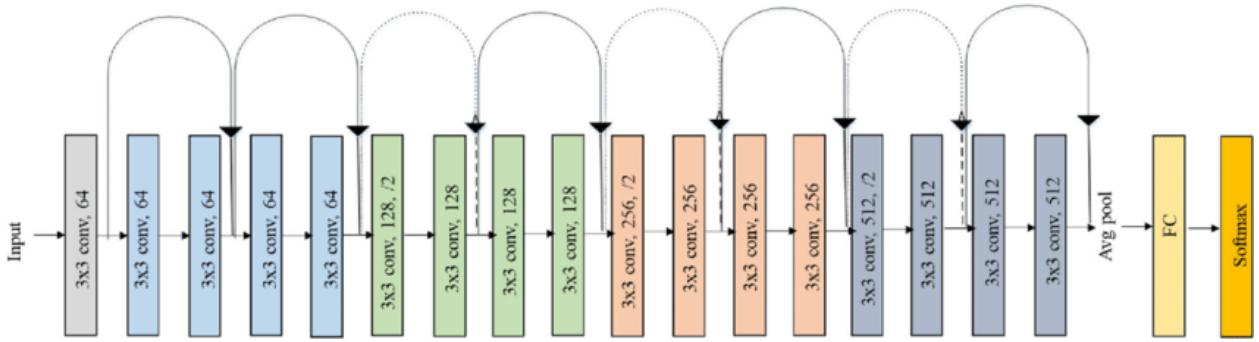


Figure 4.1.3: Arhitectura ResNet18 (preluat din [21])

Pentru antrenare am folosit batch-uri de dimensiune 64, iar ca funcție loss Binary Cross Entropy. După 35 de epoci de antrenare obținem indicatorii optimi din punct de vedere al recall-ului în epoca 21:

- Acuratețe: 74,10%
- Precizie: 46,18%
- Recall: 90,08%
- Scor F1: 61,06%
- Matricea de confuzie: $\begin{pmatrix} 1444 & 635 \\ 60 & 545 \end{pmatrix}$

4.1.3 DenseNet201

O rețea neurală convecțională densă conectează între ele și pentru fiecare strat input-ul este reprezentat de toate hărțile de activare obținute anterior. Acest lucru ajută la propagarea mai puternică a caracteristicilor și încurajează refolosirea hărților de activare, lucru ce micșorează numărul total de parametrii [9]. În figura 4.1.4 putem observa un model de rețea densă cu 5 straturi per bloc.

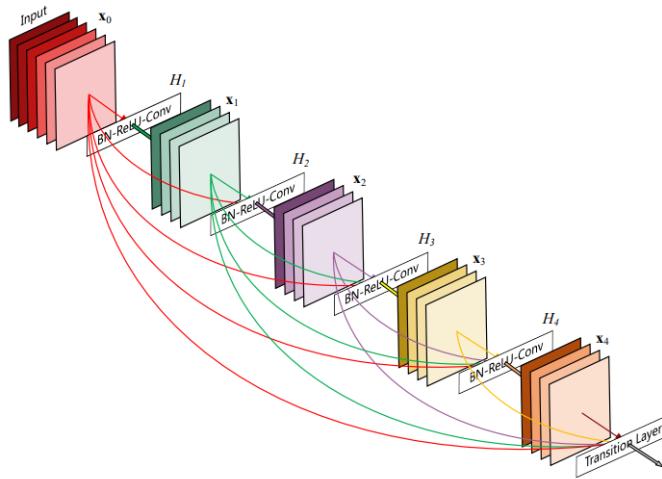


Figure 4.1.4: O rețea densă cu 5 straturi per bloc (preluat din [9])

În antrenare vom folosi DenseNet201 ce are 201 straturi grupate în 4 blocuri dense, 3 blocuri de tranziție și un strat de clasificare, aşa cum vedem și în tabelul din figura 4.1.5. Acest model are în total 18.1 milioane de parametrii ce trebuie învățați.

Întrucât rețeaua a fost mai complexă și în urma încercării de a antrena cu tot setul de date a fost atinsă limita de memorie, am redus setul de date de antrenare la 10.000, setul de testare la 2000 și numărul de epoci la 10. Batch-urile au fost de dimensiune 16, iar ca funcție loss am ales Binary Cross Entropy. După 10 epoci de antrenare obținem indicatorii:

- Acuratețe: 72,14%
- Precizie: 40,23%
- Recall: 85,27%

- Scor F1: 51,42%
- Matricea de confuzie: $\begin{pmatrix} 1107 & 499 \\ 58 & 336 \end{pmatrix}$

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112		7 × 7 conv, stride 2		
Pooling	56 × 56		3 × 3 max pool, stride 2		
Dense Block (1)	56 × 56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56 × 56		1 × 1 conv		
	28 × 28		2 × 2 average pool, stride 2		
Dense Block (2)	28 × 28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28 × 28		1 × 1 conv		
	14 × 14		2 × 2 average pool, stride 2		
Dense Block (3)	14 × 14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14 × 14		1 × 1 conv		
	7 × 7		2 × 2 average pool, stride 2		
Dense Block (4)	7 × 7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1 × 1		7 × 7 global average pool		
			1000D fully-connected, softmax		

Figure 4.1.5: Arhitecturi DenseNet (preluat din [9])

4.2 Class Activation Map

Vom prezenta în această secțiune o metodă prin care putem evidenția zonele din imagini ce au determinat clasificarea sa într-o anumită clasă, în cazul nostru să vedem unde este localizată pneumonia, urmărind ca referință articolul [30]. Pentru asta avem nevoie să prelucrăm o arhitectură de rețea pe care am făcut antrenarea pentru a returna o hartă de activare în care să fie evidențiate aceste caracteristici. Vom alege ResNet18 deoarece ultimul strat convolutional este urmat de un strat de *global average pooling*, care returnează media spațială a hărților de activare returnate de ultimul strat convolutional, și de un strat complet conectat ce returnează clasa imaginii. Făcând produsul scalar al acestor valori cu ponderile învățate în ultimul strat complet conectat obținem output-ul dorit. Dacă în schimb facem produsul scalar al valorilor obținute din ultimul strat convolutional (înainte de mediere) cu ponderile ultimului strat, vom obține harta de activare a unei clase (*class activation map*). Acest lucru este evidențiat în figura 4.2.1.

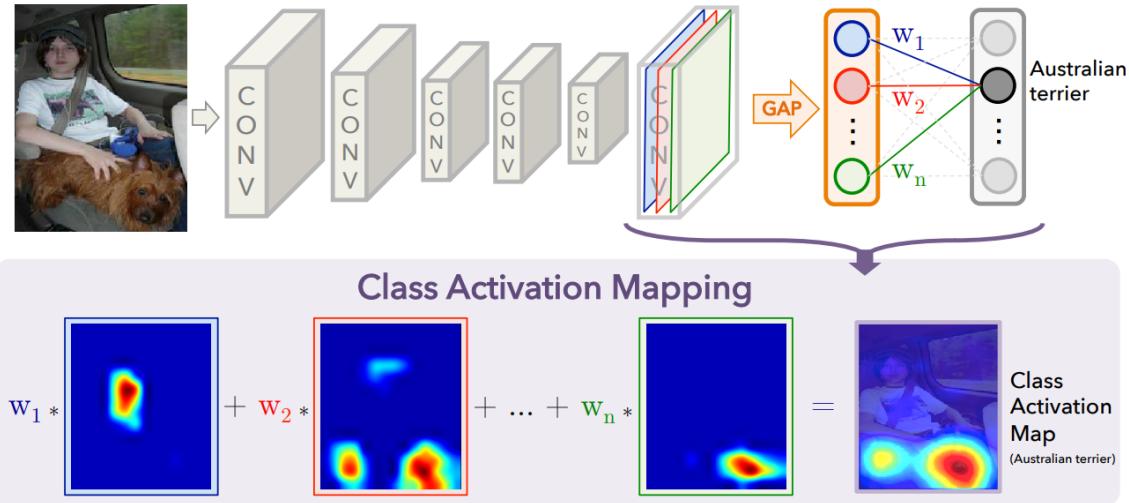


Figure 4.2.1: Exemplu Class Activation Map (CAM) (preluat din [30])

Explicăm acum această metodă din punct de vedere formal, folosind ca funcție de activare pentru output Softmax pentru o clasificare de imagini în m clase. Având o imagine dată, să presupunem că $f_k(x, y)$ este a k -a hartă de activare returnată de ultimul strat convolutional, în punctul spatial (x, y) . Așadar, pentru această hartă rezultatul aplicării stratului de global average pooling va fi $F_k = \sum_{x,y} f_k(x, y)$. Deci, input-ul pentru funcția softmax pentru o clasă c va fi $S_c = \sum_k w_k^c F_k$,

unde w_k^c este ponderea asociată clasei c pentru a k -a hartă din stratul precedent. Cu alte cuvinte, w_k^c indică importanța lui F_k pentru clasa c . Ouput-ul în urma aplicării softmax va fi:

$$P_c = \frac{e^{S_c}}{\sum_c e^{S_c}}.$$

De precizat că în acest caz o să considerăm bias-ul ca fiind 0. Dacă înlocuim $F_k = \sum_{x,y} f_k(x, y)$ în scrierea lui S_c obținem:

$$S_c = \sum_k w_k^c \sum_{x,y} f_k(x, y) = \sum_{x,y} \sum_k w_k^c f_k(x, y).$$

Definim M_c ca fiind harta de activare a clasei c , unde fiecare valoare spațială este dată de:

$$M_c(x, y) = \sum_k w_k^c f_k(x, y).$$

Așadar, $S_c = \sum_{x,y} M_c(x, y)$ de unde rezultă că $M_c(x, y)$ indică direct importanța activării valorii de pe poziția (x, y) , ce duce la clasificarea imaginii în clasa c .

Intuitiv, ne așteptăm ca fiecare hartă returnată de straturile convolutionale să fie activată de un anumit model vizual pe care îl detectează în câmpul receptiv. Așadar, f_k reprezintă harta prezenței acestui model iar harta de activare a unei clase este suma ponderată a prezentelor acestor modele în diferite locații spațiale. Printr-o supraeașantionare a acestor hărți la dimensiunea imaginii de input, vom putea identifica pe ea regiunile cele mai relevante pentru o clasă anume.

În cazul aplicației noastre pentru detecția pneumoniei avem o singură clasă de output care ne dă probabilitatea ca un pacient să aibă pneumonie sau nu. Așadar, vom avea o singură hartă de activare a clasei pentru fiecare imagine, care va evidenția caracteristicile ce au dus la clasificarea imaginii în una dintre cele două categorii (pozitivă sau negativă). În figura 4.2.2 sunt câteva exemple de imagini din setul de date și hărțile de activare ale clasei obținute în urma antrenării folosind ResNet18 pentru fiecare din ele.

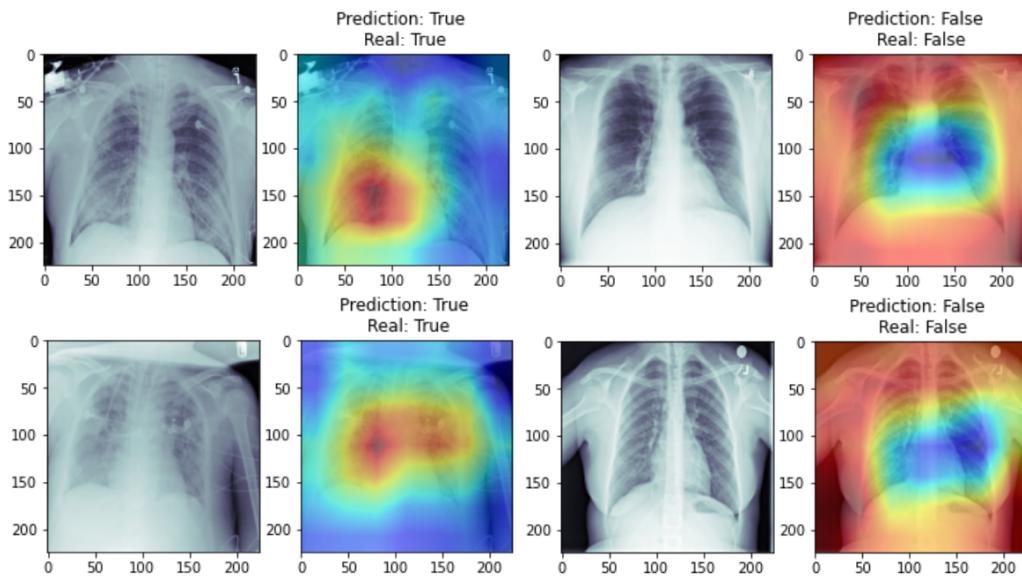


Figure 4.2.2: Exemple CAM cu localizarea pneumoniei

Codul sursă pentru antrenările efectuate pe aceste trei modele, cât și rezultatele experimentelor cu hărțile de activare ale claselor se găsesc în anexa 5.3.

5 Anexe

5.1 Regresia Liniară

Codul sursă al regresiei liniare din secțiunea 2.1.1, folosind [35]:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
5 y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]
6
7 mymodel1 = np.poly1d(np.polyfit(x, y, 1))
8 mymodel2 = np.poly1d(np.polyfit(x, y, 2))
9 mymodel3 = np.poly1d(np.polyfit(x, y, 3))
10
11 myline = np.linspace(1, 22, 100)
12
13 fig = plt.figure()
14 ax = fig.add_subplot()
15 ax.set_title("Grad 1")
16 plt.scatter(x, y)
17 plt.plot(myline, mymodel1(myline))
18 plt.show()
19
20 fig = plt.figure()
21 ax = fig.add_subplot()
22 ax.set_title("Grad 2")
23 plt.scatter(x, y)
24 plt.plot(myline, mymodel2(myline))
25 plt.show()
26
27 fig = plt.figure()
28 ax = fig.add_subplot()
29 ax.set_title("Grad 3")
30 plt.scatter(x, y)
31 plt.plot(myline, mymodel3(myline))
32 plt.show()
```

5.2 Antrenare MNIST folosind ANN

În continuare prezentăm codul sursă de la experimentele făcute pe setul de date MNIST folosind ANN, rulat pe platforma Google Colab, iar detaliile antrenării se găsesc pe GitHub [37]:

```
1 # Importam librariile necesare
2 import torch
3 import torch.nn as nn
4 import torch.optim as optim
5 import torch.utils.data as data
6 import torch.nn.functional as F
7 from torch.utils.data import DataLoader
8 from torchvision import datasets, transforms
9 import numpy as np
10 import pandas as pd
11 from sklearn.metrics import confusion_matrix
12 import matplotlib.pyplot as plt
13 from IPython import display as dspl
14 import time
15 import torchvision

1 transform = transforms.ToTensor()
2 train_data = datasets.MNIST(root='../Data', train=True, download=True, transform=transform)
3 test_data = datasets.MNIST(root='../Data', train=False, download=True, transform=transform)

1 batch_size_train = 500
2 batch_size_test = 100
3
4 train_loader = data.DataLoader(train_data, batch_size=batch_size_train, shuffle=True, num_workers
   =2)
5 test_loader = data.DataLoader(test_data, batch_size=batch_size_test, shuffle=False)

1 class OneHidden(nn.Module):
2     def __init__(self):
3         super(OneHidden, self).__init__()
4         self.fc1 = nn.Linear(784, 300)
5         self.fc2 = nn.Linear(300, 10)
6
7         def forward(self, x: torch.Tensor):
8             x = F.relu(self.fc1(x))
9             x = F.relu(self.fc2(x))
10            return F.log_softmax(x, dim=1)
```

```

1 # functia cu ajutorul careia se face antrenarea
2 def train_fn(epochs: int, train_loader: data.DataLoader, test_loader: data.DataLoader, net: nn.
3             Module, loss_fn: nn.Module, optimizer: optim.Optimizer):
4
5
6     for e in range(epochs):
7         total = 0
8         train_corr = 0
9
10        for images, labels in train_loader:
11            images = images.to(device)
12            labels = labels.to(device)
13            out = net(images.view(batch_size_train, -1))
14            loss = loss_fn(out, labels)
15            loss.backward()
16            optimizer.step()
17            optimizer.zero_grad()
18            total += len(images)
19            predicted = torch.max(out.data, 1)[1]
20            nr_corr_batch = (predicted == labels).sum()
21            train_corr += nr_corr_batch
22
23        acc_train = (train_corr / total) * 100
24        train_accs.append((e, acc_train))
25        train_losses.append((e, loss.item()))
26        total = 0
27        test_corr = 0
28        with torch.no_grad():
29            for test_images, test_labels in test_loader:
30                test_images = test_images.to(device)
31                test_labels = test_labels.to(device)
32                total += len(test_images)
33                y_val = net(test_images.view(batch_size_test, -1))
34                out_class = torch.max(y_val.data, 1)[1]
35                test_corr += (out_class == test_labels).sum()
36
37                loss_test = loss_fn(y_val, test_labels)
38                acc_test = (test_corr / total) * 100
39                test_accs.append((e, acc_test))
40                test_losses.append((e, loss_test.item()))
41
42
43    print(f'epoch: {e:2}  loss_train/test: {loss.item():10.8f}/{loss_test.item():10.8f}

```

```

accuracy_train/test: {acc_train:7.3f}% / {acc_test:7.3f}%'
```

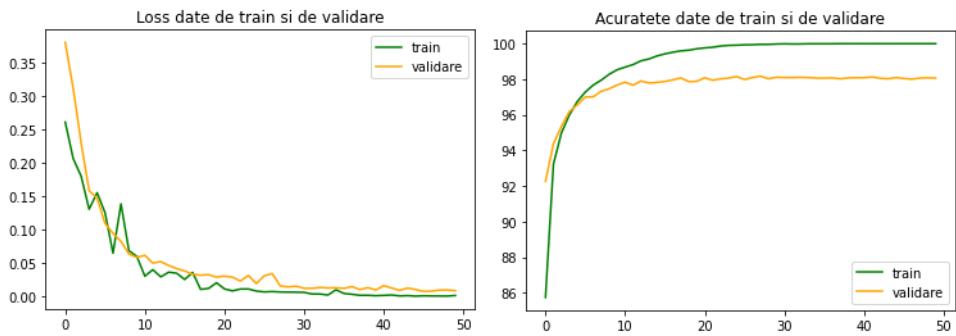
- 1 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11 train_fn(epochs, train_loader, test_loader, model, loss_fn, optimizer)

```

# Graficele cu loss ul si acuratetea
import matplotlib.pyplot as plt
plt.figure()

x_val_train_loss = [x[0] for x in train_losses]
y_val_train_loss = [x[1] for x in train_losses]
plt.plot(x_val_train_loss, y_val_train_loss, color = 'green', label = 'train')
x_val_valid_loss = [x[0] for x in test_losses]
y_val_valid_loss = [x[1] for x in test_losses]
plt.plot(x_val_valid_loss, y_val_valid_loss, color = 'orange', label = 'validare')
plt.title("Loss date de train si de validare")
plt.legend()
plt.figure()

x_val_train_acc = [x[0] for x in train_accs]
y_val_train_acc = [torch.Tensor.cpu(x[1]) for x in train_accs]
plt.plot(x_val_train_acc, y_val_train_acc, color = 'green', label = 'train')
from torch import Tensor
x_val_valid_acc = [x[0] for x in test_accs]
y_val_valid_acc = [torch.Tensor.cpu(x[1]) for x in test_accs]
plt.plot(x_val_valid_acc, y_val_valid_acc, color = 'orange', label = 'validare')
plt.title("Acuratete date de train si de validare")
plt.legend()
```



```

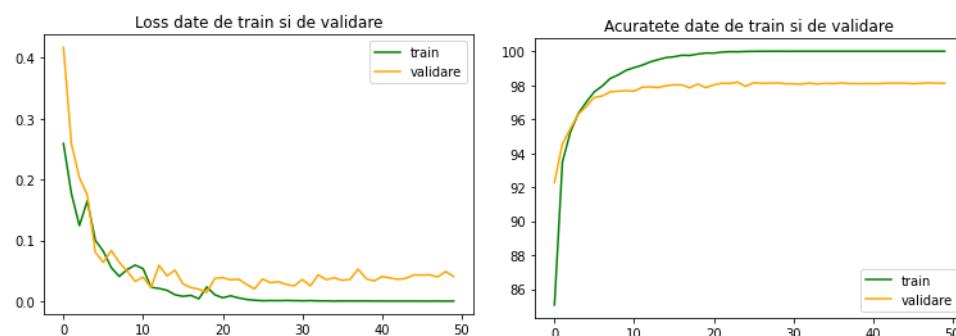
1 class TwoHidden(nn.Module):
2     def __init__(self):
3         super(TwoHidden, self).__init__()
4         self.fc1 = nn.Linear(784, 300)
5         self.fc2 = nn.Linear(300, 100)
6         self.fc3 = nn.Linear(100, 10)
7
8     def forward(self, x: torch.Tensor):
9         x = F.relu(self.fc1(x))
10        x = F.relu(self.fc2(x))
11        x = self.fc3(x)
12
13        return x

```

```

1 epochs = 50
2 model = TwoHidden()
3 model.to(device)
4 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
5 optimizer.zero_grad()
6 loss_fn = nn.CrossEntropyLoss()
7 train_fn(epochs, train_loader, test_loader, model, loss_fn, optimizer)

```



5.3 Pneumonia Detection

Preprocesarea datelor

```
1 from pathlib import Path
2 import pydicom
3 import numpy as np
4 import cv2
5 import pandas as pd
6 import matplotlib.pyplot as plt
7 from tqdm.notebook import tqdm
8 import torch
9 import torchvision
10 from torchvision import transforms
11 import torchmetrics
12 import pytorch_lightning as pl
13 from pytorch_lightning.callbacks import ModelCheckpoint
14 from pytorch_lightning.loggers import TensorBoardLogger
15 from tqdm.notebook import tqdm

1 # Descarcarea setului de date de pe Kaggle
2 !kaggle competitions download -c rsna-pneumonia-detection-challenge

1 labels = pd.read_csv("stage_2_train_labels.csv")
2 labels = labels.drop_duplicates("patientId")

1 ROOT_PATH = Path("stage_2_train_images/")
2 SAVE_PATH = Path("Processed")

1 # Impartirea imaginilor in foldere, normalizare si resize
2 from unicodedata import normalize
3 sums, sums_squared = 0, 0
4
5 for c, patient_id in enumerate(tqdm(labels.patientId)):
6     patient_id = labels.patientId.iloc[c]
7     dcm_path = ROOT_PATH / patient_id
8     dcm_path = dcm_path.with_suffix(".dcm")
9     dcm = pydicom.read_file(dcm_path).pixel_array / 255 # normalizare
10    dcm_array = cv2.resize(dcm, (224, 224)).astype(np.float16)
11    label = labels.Target.iloc[c]
12    train_or_val = "train" if c < 24000 else "val" # impartirea in antrenare si testare
13
14    current_save_path = SAVE_PATH / train_or_val / str(label)
```

```

15     current_save_path.mkdir(parents = True, exist_ok = True)
16     np.save(current_save_path/patient_id, dcm_array)
17
18     normalizer = 224*224
19
20     if train_or_val == "train":
21         sums += np.sum(dcm_array) / normalizer
22         sums_squared += (dcm_array ** 2).sum() / normalizer

```

```

1 mean = sums / 24000
2 std = np.sqrt((sums_squared / 24000) - mean**2)
3 mean, std

```

```

1 (0.49039623525191567, 0.2479507326197431)

```

Data Augmentation

```

1 def load_file(path):
2     return np.load(path).astype(np.float32)

```

```

1 train_transforms = transforms.Compose([
2
3             transforms.ToTensor(),
4             transforms.Normalize(mean, std),
5             transforms.RandomAffine(degrees = (-5, 5), translate = (0,
6                 0.05), scale = (0.9, 1.1)),
7             transforms.RandomResizedCrop((224, 224), scale = (0.35, 1))
8         ])
9
10 val_transforms = transforms.Compose([
11
12             transforms.ToTensor(),
13             transforms.Normalize(mean, std),
14         ])

```

```

1 train_dataset = torchvision.datasets.DatasetFolder("Processed/train", loader = load_file,
2                                                 extensions = "npy", transform = train_transforms)
3 val_dataset = torchvision.datasets.DatasetFolder("Processed/val", loader = load_file, extensions =
4                                                 "npy", transform = val_transforms)

```

În continuare prezentăm antrenarea rețelei ResNet18 și rezultatele pentru cele trei arhitecturi, codurile sursă complete găsindu-se pe GitHub [37].

• RESNET18

```
1 batch_size = 64
2 num_workers = 2
3 train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, num_workers =
    num_workers, shuffle = True)
4 val_loader = torch.utils.data.DataLoader(val_dataset, batch_size = batch_size, num_workers =
    num_workers, shuffle = False)

1 class PneumoniaModel(pl.LightningModule):
2
3     def __init__(self, weight = 1):
4         super().__init__()
5
6         self.model = torchvision.models.resnet18()
7         self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
8                                         bias=False)
9         self.model.fc = torch.nn.Linear(in_features=512, out_features=1, bias=True)
10
11         self.optimizer = torch.optim.Adam(self.model.parameters(), lr=1e-4)
12         self.loss_fn = torch.nn.BCEWithLogitsLoss(pos_weight = torch.tensor([weight]))
13
14         self.train_acc = torchmetrics.Accuracy()
15         self.val_acc = torchmetrics.Accuracy()
16         self.val_f1 = torchmetrics.F1Score()
17         self.val_recall = torchmetrics.Recall()
18
19     def forward(self, data):
20         pred = self.model(data)
21         return pred
22
23     def training_step(self, batch, batch_idx):
24         x_ray, label = batch
25         label = label.float()
26         pred = self(x_ray)[:,0]
27         loss = self.loss_fn(pred, label)
28
29         self.log("Train Loss", loss)
30         self.log("Step Train ACC", self.train_acc(torch.sigmoid(pred), label.int()))
```

```

30
31     return loss
32
33 def training_epoch_end(self, outs):
34     self.log("Train ACC", self.train_acc.compute())
35
36 def validation_step(self, batch, batch_idx):
37     x_ray, label = batch
38     label = label.float()
39     pred = self(x_ray)[:,0]
40     loss = self.loss_fn(pred, label)
41
42     self.log("Val Loss", loss)
43     self.log("Step Val ACC", self.val_acc(torch.sigmoid(pred), label.int()))
44     self.log("Step Val Recall", self.val_recall(torch.sigmoid(pred), label.int()))
45     self.log("Step Val F1Score", self.val_f1(torch.sigmoid(pred), label.int()))
46
47     return loss
48
49 def validation_epoch_end(self, outs):
50     self.log("Val ACC", self.val_acc.compute())
51     self.log("Val Recall", self.val_f1.compute())
52     self.log("Val F1Score", self.val_recall.compute())
53
54 def configure_optimizers(self):
55     return [self.optimizer]

```

```
1 model = PneumoniaModel(weight = 3)
```

```

1 # salvarea ponderilor intermedii
2 checkpoint_callback1 = ModelCheckpoint(
3     monitor='Val Recall',
4     dirpath='./weights',
5     save_top_k=15,
6     mode='max'
7 )
```

```

1 gpus = 1
2 trainer = pl.Trainer(gpus=gpus, logger=TensorBoardLogger(save_dir='./logs'), log_every_n_steps=1,
3                         callbacks=checkpoint_callback1, max_epochs=35)
```

```

1 # antrenarea
2 trainer.fit(model, train_loader, val_loader)
```

Testarea acurateții:

```
1 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
2
3 model = PneumoniaModel.load_from_checkpoint("/content/weights/epoch=20-step=7933.ckpt")
4 model.eval()
5 model.to(device);

1 preds = []
2 labels = []
3 with torch.no_grad():
4     for data, label in tqdm(val_dataset):
5         data = data.to(device).float().unsqueeze(0)
6         pred = torch.sigmoid(model(data)[0].cpu())
7         preds.append(pred)
8         labels.append(label)
9     preds = torch.tensor(preds)
10    labels = torch.tensor(labels).int()

1 acc = torchmetrics.Accuracy()(preds, labels)
2 precision = torchmetrics.Precision()(preds, labels)
3 recall = torchmetrics.Recall()(preds, labels)
4 F1score = torchmetrics.F1Score()(preds, labels)
5 cm = torchmetrics.ConfusionMatrix(num_classes=2)(preds, labels)
6
7 print(f"Val Accuracy: {acc}")
8 print(f"Val Precision: {precision}")
9 print(f"Val Recall: {recall}")
10 print(f"Val F1Score: {F1score}")
11 print(f"Confusion Matrix:\n {cm}")

1 Val Accuracy: 0.7410581111907959
2 Val Precision: 0.4618644118309021
3 Val Recall: 0.9008264541625977
4 Val F1Score: 0.6106442213058472
5 Confusion Matrix:
6 tensor([[1444,  635],
7         [   60,  545]])
```

• DENSENET201

```
1 acc = torchmetrics.Accuracy()(preds, labels)
2 precision = torchmetrics.Precision()(preds, labels)
3 recall = torchmetrics.Recall()(preds, labels)
4 F1score = torchmetrics.F1Score()(preds, labels)
5 cm = torchmetrics.ConfusionMatrix(num_classes=2)(preds, labels)
6
7 print(f"Val Accuracy: {acc}")
8 print(f"Val Precision: {precision}")
9 print(f"Val Recall: {recall}")
10 print(f"Val F1Score: {F1score}")
11 print(f"Confusion Matrix:\n {cm}")
```

```
1 Val Accuracy: 0.7214999794960022
2 Val Precision: 0.40239521861076355
3 Val Recall: 0.8527919054031372
4 Val F1Score: 0.5467860188
5 Confusion Matrix:
6 tensor([[1107,  499],
7         [   58,  336]])
```

• ALEXNET

```
1 acc = torchmetrics.Accuracy()(preds, labels)
2 precision = torchmetrics.Precision()(preds, labels)
3 recall = torchmetrics.Recall()(preds, labels)
4 F1score = torchmetrics.F1Score()(preds, labels)
5 cm = torchmetrics.ConfusionMatrix(num_classes=2)(preds, labels)
6
7 print(f"Val Accuracy: {acc}")
8 print(f"Val Precision: {precision}")
9 print(f"Val Recall: {recall}")
10 print(f"Val F1Score: {F1score}")
11 print(f"Confusion Matrix:\n {cm}")
```

```
1 Val Accuracy: 0.7697466611862183
2 Val Precision: 0.4938271641731262
3 Val Recall: 0.8595041036605835
4 Val F1Score: 0.6272616982460022
5 Confusion Matrix:
6 tensor([[1546,  533],
7         [   85,  520]])
```

CAM (Class Activation Maps)

Definim funcțiile folosite în generarea hărților de activare iar codul complet se găsește pe GitHub [37].

```
1 class PneumoniaModelCAM(pl.LightningModule):
2     def __init__(self, weight = 1):
3         super().__init__()
4         self.model = torchvision.models.resnet18()
5         self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
6             bias=False)
7         self.model.fc = torch.nn.Linear(in_features=512, out_features=1,bias=True)
8         self.feature_map = torch.nn.Sequential(*list(self.model.children())[:-2])
9
9
9     def forward(self, data):
10        feature_map = self.feature_map(data)
11        avg_pool_output = torch.nn.functional.adaptive_avg_pool2d(input = feature_map, output_size =
12            (1,1))
13        avg_output_flattened = torch.flatten(avg_pool_output)
14        pred = self.model.fc(avg_output_flattened)
15
15        return pred, feature_map
```

```
1 def CAM(model, img):
2     with torch.no_grad():
3         pred, features = model(img.unsqueeze(0))
4         features = features.reshape((512, 49)) # [1, 512, 7, 7]
5         weight_params = list(model.model.fc.parameters())[0] # fara bias
6         weight = weight_params[0].detach()
7         cam = torch.matmul(weight, features)
8         cam_img = cam.reshape(7,7).cpu()
9
9         return cam_img, torch.sigmoid(pred)
```

```
1 def visualize(val, act, pred):
2     img = val[0][0]
3     act = transforms.functional.resize(act.unsqueeze(0), (224, 224))[0]
4     fig, axis = plt.subplots(1, 2)
5     axis[0].imshow(img, cmap = "bone")
6     axis[1].imshow(img, cmap = "bone")
7     axis[1].imshow(act, alpha=0.5, cmap="jet")
8     plt.title(f"Prediction: {(pred > 0.5).item()} \n Real: {(val[1]==1)}")
9     plt.show()
```

Bibliografie

- [1] Ahmadi, A.A., 2018, Theory of convex functions. Lecture Notes Princeton, https://www.princeton.edu/aaa/Public/Teaching/ORF523/ORF523_Lec7.pdf
- [2] Barla, N., 2022. A Gentle Introduction to Deep Learning - the ELIS Way <https://www.v7labs.com/blog/deep-learning-guide> (accesat pe 14.06.2022)
- [3] Brownlee J., 2020. A Gentle Introduction to Cross-Entropy for Machine Learning <https://machinelearningmastery.com/cross-entropy-for-machine-learning/> (accesat pe 14.06.2022)
- [4] Cybenko, G., 1989. Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems, 2(4), pp.303-314.
- [5] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K. and Fei-Fei, L., 2009, June. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition (pp. 248-255). Ieee.
- [6] Deng, L., 2012. The mnist database of handwritten digit images for machine learning research [best of the web]. IEEE signal processing magazine, 29(6), pp.141-142.
- [7] Gower, R.M., 2018. Convergence theorems for gradient descent. Lecture notes for Statistical Optimization.
- [8] He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).
- [9] Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K.Q., 2017. Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

- [10] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.
- [11] Lambers, J., 2010. Gradients of Inner Products. Lecture Notes, <https://www.math.usm.edu/lambers/mat610/sum11/lecture8.pdf>
- [12] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), pp.2278-2324.
- [13] Lörke, A., Schneider, F., Heck, J., 2019. Cybenko's Theorem and the capability of a neural network as function approximator.
https://www.mathematik.uni-wuerzburg.de/fileadmin/10040900/2019/Seminar_Artificial_Neural_Network_24_9_.pdf
- [14] Million, E., 2007. The hadamard product. Course Notes, 3(6).
<http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf>
- [15] Nielsen, M.A., 2015. Neural networks and deep learning (Vol. 25). San Francisco, CA, USA: Determination press.
- [16] Nwankpa, C., Ijomah, W., Gachagan, A. and Marshall, S., 2018. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*.
- [17] O'Shea, K. and Nash, R., 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*.
- [18] Portilla, J., Fruh, M., Gatidis, S., Hepp, T., 2021. Deep Learning with PyTorch for Medical Image Analysis:
<https://www.udemy.com/course/deep-learning-with-pytorch-for-medical-image-analysis/>
- [19] Rahman, T., Chowdhury, M.E., Khandakar, A., Islam, K.R., Islam, K.F., Mahbub, Z.B., Kadir, M.A. and Kashem, S., 2020. Transfer learning with deep convolutional neural network (CNN) for pneumonia detection using chest X-ray. *Applied Sciences*, 10(9), p.3233.

- [20] Ramachandran, P., Zoph, B. and Le, Q.V., 2017. Searching for activation functions. arXiv preprint arXiv:1710.05941.
- [21] Ramzan, F., Khan, M.U.G., Rehmat, A., Iqbal, S., Saba, T., Rehman, A. and Mehmood, Z., 2020. A deep learning approach for automated diagnosis and multi-class classification of Alzheimer's disease stages using resting-state fMRI and residual neural networks. *Journal of medical systems*, 44(2), pp.1-16.
- [22] Sharma, S., Sharma, S. and Athaiya, A., 2017. Activation functions in neural networks. towards data science, 6(12), pp.310-316.
- [23] Singer, Y., "Advanced Optimization", Lecture notes (2016) https://people.seas.harvard.edu/~yaron/AM221-S16/lecture_notes/AM221_lecture9.pdf
- [24] Tetko, I.V., Livingstone, D.J. and Luik, A.I., 1995. Neural network studies. 1. Comparison of overfitting and overtraining. *Journal of chemical information and computer sciences*, 35(5), pp.826-833.
- [25] Thomas, A., 2017. An introduction to neural networks for beginners (pp. 14-15). Technical report in Adventures in Machine Learning.
- [26] Turinici, G., 2021. The convergence of the Stochastic Gradient Descent (SGD): a self-contained proof. arXiv preprint arXiv:2103.14350.
- [27] Wang, X., Peng, Y., Lu, L., Lu, Z., Bagheri, M. and Summers, R.M., 2017. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2097-2106).
- [28] Wu, J., 2017. Introduction to convolutional neural networks. National Key Lab for Novel Software Technology. Nanjing University. China, 5(23), p.495.

- [29] Zhang, A., Lipton, Z.C., Li, M. and Smola, A.J., 2021. Dive into deep learning. arXiv preprint arXiv:2106.11342.
- [30] Zhou, B., Khosla, A., Lapedriza, A., Oliva, A. and Torralba, A., 2016. Learning deep features for discriminative localization. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2921-2929).
- [31] <https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html> (accesat pe 29.11.2021)
- [32] Stanford Deep Learning tutorial: <http://deeplearning.stanford.edu/tutorial/> (accesat pe 15.06.2022)
- [33] <https://setosa.io/ev/image-kernels/> (accesat pe 10.06.2022)
- [34] <https://www.kaggle.com/competitions/rsna-pneumonia-detection-challenge/data> (accesat pe 10.06.2022)
- [35] https://www.w3schools.com/python/python_ml_polynomial_regression.asp (accesat pe 10.06.2022)
- [36] <https://en.wikipedia.org/wiki/Convolution> (accesat pe 14.06.2022)
- [37] <https://github.com/alexandra-dragomir/Licenta> (accesat pe 15.06.2022)