**Chivu Iulian**

**432A**

**Proiect2 – Clasificator forme geometrice cu MLP**

**A) Introducere asupra temei abordate**

Tema abordata consta in realizarea unui clasificator de forme geometrice cu ajutorul unei retele neuronale artificiale(RNA). Acesta fiind primul proiect cu RNA am ales sa implementez arhitectura: MLP (Multilayer perceptron), aceasta fiind considerata cea mai simpla retea neuronala, mai ales daca avem un singur strat ascuns.

Dupa partea de invatare a retelei acest clasificator va primi la intrare o poza (28x28 pixeli) in care se afla o figura desenata cu negru pe fundal alb si va da la iesire un raspuns cu ce figura geometrica se afla in imaginea respectiva. Pentru inceput o sa proiectez reteaua sa determine daca se afla sau nu triunghiuri in poza.

**B) Teoria asupra metodelor folosite**

**a)Creare baza de date + procesare imagine**

Primul pas pe care l-am facut in acest proiect a fost sa imi creez baza de date. Deoarece initial am decis ca reteaua sa detecteze daca se afla sau nu triunghiuri intr-o poza de 28X28 pixeli, am pregatit setul de antrenament. Acesta consta in 132 de poze in care sunt desenate triunghiuri de diferite marimi si in diferite pozitii si poze in care sunt desenate alte forme geometrice diferite de triughiuri.

Urmatorul pas a fost procesarea imaginii. Folosind libraria opencv am reusit sa citesc imaginea si sa obtin 3 matrici de pixeli (RGB), care contin valori intregi intre 0 si 255. Cele 3 matrici fiind similare doarece folosesc imagini alb negru, am decis sa merg mai departe doar cu matricea verde si am normalizat valorile (am impartit fiecare valoare a pixelului cu 255) pentru a usura munca retelei.

Am modificat forma matricei intr-un vector de dimenisune 784. Inputul in retea va fi o matrice X care este alcatuita din 132 de coloane de astfeli de vectori, deoarece am 132 de exemple pentru antrenarea retelei. deci X va fi o matrice de 784 de linii si 132 de coloane. Iar iesirea Y consta intr-un vector cu 132 de valori (0 sau 1) 0-nu se afla triunghiuri in poza, 1-se afla triunghiuri in poza.

**b)Teorie retele neuronale(MLP):**

Multilayer perceptron este o retea neuronala care contine cel putin doua straturi: un strat ascuns si un strat de iesire, aceasta arhitectura fiind cea mai simpla. Fiecare strat contine un numar de neuroni, straturile ascunse au un numar arbitrar de neuroni, iar stratul de iesire de exemplu in cazul meu are un singur neuron care da la iesire probabilitatea ca in poza sa se afle un triunghi.

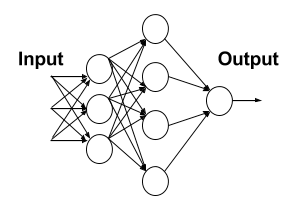


Fig1 : Multilayer Perceptron cu 2 straturi ascunse si un strat de iesire

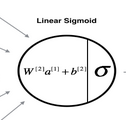
Fiecare neuron contine ponderi: weights = w si biases = b si o functie de activare care sunt folosite in algoritmul de invatare **Logistic Regression**:

1)Z = w\*X + b ; unde X reprezinta intrarea in stratul curent.

2)A = sigmoid(Z) ; deoarece ne dorim ca iesirea sa reprezinte o probabilitate.

w este initializat random dupa o distributie normala

b este initializt cu 0

Fig2: Reprezentare neuron

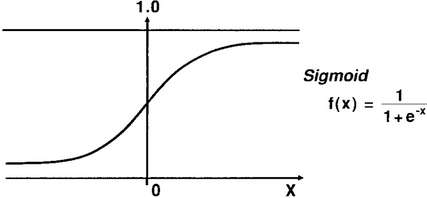


Fig3: Functiia de activare sigmoid

Deoarece la intrare avem o matrice X[784][132], considerand ca reteau are strucutra din Fig1, atunci fiecare din cei 3 neuroni de pe stratul 1 vor avea un vector w de dimensiunea numarului de intrari in retea = 784, o valoare b si functia de acctivare sigmoid.

In codul realizat de mine am folosit tehnica de vectorizare pentru a scapa de implementarea retelei folosind o structura complexa de lista in linsta. Astfel am putut implementa retaua folosind doar o lista dublu inlatuita.

Tehnica de vectorizare consta in gruparea valorilor w, b din cele 3 noduri intr-o matrice respectiv vector. In cazul nostru pentru stratul numarul 1 vom avea o matrice de ponderi w1 de dimensiuni [numarul de noduri de pe stratul curent] [numarul de noduri de pe stratul anterior, respectiv intrare in retea] => w1 [3][784], iar b1 devine un vector de dimensiune [3] (numarul de noduri de pe stratul curent). Iesirea din strat = A dupa inmultirea matricilor va fi de dimensiunea [3][132].

Algoritmul de invatare al retelei se poate impartii in doua parti:

**-etapa forward**

**-etapa backward**

**Etapa forward**:

Aceasta etapa se bazeaza pe calcularea erorii(Cost function) intre Y – setul de iesire pentru antrenare si A – iesirea din ultimul strat din retea folosind algoritmul Logistic Regression.

Primul pas este in a definii functia care ne va da diferenta dintre cele doua seturi de valori. Trebui sa fim atenti in aceasta decizie doarece in etapa backwards o sa folosim algoritmul gradientului descendent, algoritm care poate ramane blocat intr-un minim local. Astfel trebuie aleasa o functie convexa, cu un singur minim:



J(w,b) - Cost function; unde: m = numarul de exemple pentru antrenarea retelei = 132

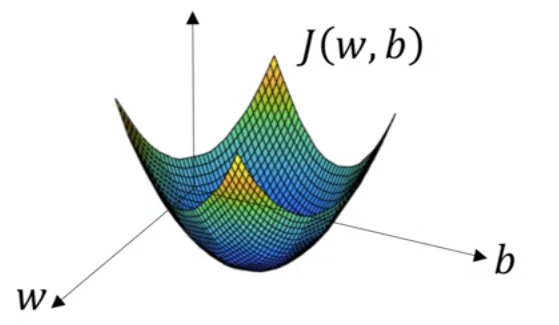


Fig4: Cost function plot

**Etapa backward:**

Aceasta etapa se bazeaza pe optimizarea parametrilor w si b astfel incat eraorea(Cost function) sa fie cat mai mica prin folosirea algoritmului **gradientului descendent**:

Metoda gradientului descendent este o metoda iterativa care porneste dintr-un punct de start aleator si care deplaseaza acest punct in directia data de derivata cu semnul schimbat deoarece vrem sa gasim punctul de minim al functiei, iar derivata functiei ne arata directia de crestere a functiei. Pozitia punctului se actualizeaza astfel : x2 = x1 – alpha \* f '(x1) .

Unde: **alpha** reprezinta pasul, acest parametru trebuie ales cu grija deoarece daca acesta este prea mare putem sarii peste punctul de minim, iar daca acesta este prea mic, algoritmul va parcurge mai multi pasi deci este nevoie de un timp mare de procesare pentru a gasi minimul. Ideal alpha se alege variabil ( o valoare mare la inceput, care scade odata cu numarul de pasi).

Astfel folosind acest algoritm w si b sunt actualizati astfel:

**w = w – alpha \* dw**

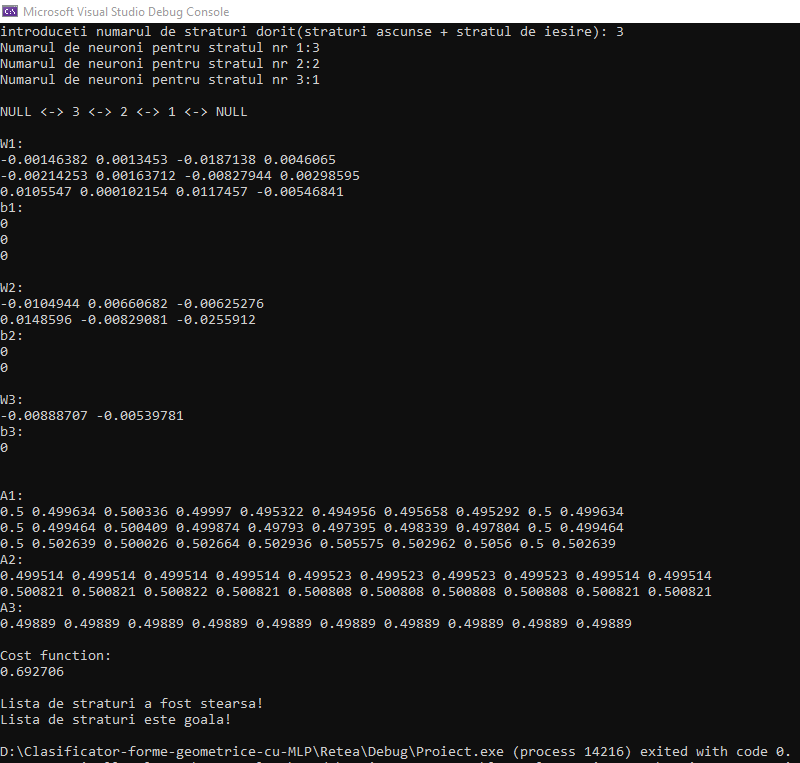
**b = b – alpha \* db**

unde: dw si db reprezinta derivatele partiale a functiei J(w,b) (Cost function) in raport cu w respectiv b;

Algoritmul de invatare al retelei repeta aceste doua etape principale avand conditia de stop fiind sa se opreasca dupa un anumit numar de pasi sau sa se opreasca daca eroarea ajunge sa fie mai mica decat o toleranta impusa.

**C)Rezultate si concluzii:**

Am reusit implementarea structurii retelei folosind o lista dublu inlatuita si notiunile de vectorizare a parametrilor invatate in cursul Neural Network and Deep Learning de Andrew Ng de pe platforma Coursera. De asemena am reusit implementare etapei forward si calcularea Cost function, verificand la fieacare pas ca dimensiunile variabilelor sa fie corecte, insa nu am mai reusit in timpul acordat sa termin de implementat etapa backward.

Fig5: Rezultatele unei rulari a codului de verificare a parametrilor

**D) Codul strict pentru implementarea proprie**

#include<iostream>

#include<random>

using namespace std;

class Strat {

double\*\* w; //weights

double\* b; //biases

int neuroni; //stocheaza nr. de neuroni de pe stratul curent

double\*\* Z; //Z = w\*X + b (logistic regression); A = sigmoid(Z);

Strat\* next;

Strat\* prev;

public:

Strat() {

w = NULL;

b = NULL;

neuroni = 0;

Z = NULL;

next = NULL;

prev = NULL;

}

double sigmoid(double Z) {

return (double)1 / (1 + exp(-Z));

}

Strat\* creare(Strat\* prim, int dim\_vect\_intrare) {

Strat\* nou, \* tmp;

int n;

default\_random\_engine generator;

normal\_distribution<double> distribution(0, 0.01); //distributie normala(Gauss) medie=0 si dispersie=0.01

if (prim != NULL) {

cout << "Lista de straturi nu este vida!" << endl;

return prim;

}

cout << "introduceti numarul de straturi dorit(straturi ascunse + stratul de iesire): ";

cin >> n;

for (int i = 0; i < n; i++) {

//creare si citirea unui nod individual:

nou = new Strat;

if (!nou) {

cout << "Memoria nu a putut fi alocata" << endl;

return NULL;

}

cout << "Numarul de neuroni pentru stratul nr " << i + 1 << ":";

cin >> nou->neuroni;

//alocare biases

nou->b = new double[nou->neuroni];

//initializare biases

for (int j = 0; j < nou->neuroni; j++) {

nou->b[j] = 0;

}

//caz particular pentru primul strat(weights):

if (prim == NULL) {

//alocare weights

nou->w = new double\* [nou->neuroni];

for (int j = 0; j < nou->neuroni; ++j) {

nou->w[j] = new double[dim\_vect\_intrare];

}

//initializare weights

for (int j = 0; j < nou->neuroni; j++) {

for (int k = 0; k < dim\_vect\_intrare; k++) {

nou->w[j][k] = distribution(generator);

}

}

}

//in cazul celorlalte straturi(weights):

else {

tmp = prim;

while (tmp->next != NULL) tmp = tmp->next;

//alocare weights

nou->w = new double\* [nou->neuroni];

for (int j = 0; j < nou->neuroni; ++j) {

nou->w[j] = new double[tmp->neuroni];

}

//initializare weights

for (int j = 0; j < nou->neuroni; j++) {

for (int k = 0; k < tmp->neuroni; k++) {

nou->w[j][k] = distribution(generator);

}

}

}

nou->next = NULL;

nou->prev = NULL;

//caz particular pentru primul strat:

if (prim == NULL) {

prim = nou;

}

//in cazul celorlalte straturi:

else {

tmp = prim;

while (tmp->next != NULL) tmp = tmp->next;

tmp->next = nou;

nou->prev = tmp;

}

}

return prim;

}

void afisare(Strat\* prim) {

Strat\* tmp;

tmp = prim;

if (prim == NULL) {

cout << "Lista de straturi este goala!" << endl;

}

else {

cout << "NULL <-> ";

while (tmp != NULL) {

cout << tmp->neuroni << " <-> ";

tmp = tmp->next;

}

cout << "NULL" << endl;

}

}

void afisare\_w\_b(Strat\* prim, int dim\_vect\_intrare) {

Strat\* tmp;

tmp = prim;

int counter = 1;

if (prim == NULL) {

cout << "Lista de straturi este goala!" << endl;

}

else {

while (tmp != NULL) {

cout << "W" << counter << ":" << endl;

//caz particular primul strat

if (tmp == prim) {

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < dim\_vect\_intrare; j++) {

cout << tmp->w[i][j] << " ";

}

cout << endl;

}

}

//in cazul celorlalte straturi

else {

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < tmp->prev->neuroni; j++) {

cout << tmp->w[i][j] << " ";

}

cout << endl;

}

}

cout << "b" << counter << ":" << endl;

counter++;

for (int i = 0; i < tmp->neuroni; i++) {

cout << tmp->b[i] << endl;

}

cout << endl;

tmp = tmp->next;

}

}

}

Strat\* stergere(Strat\* prim) {

Strat\* tmp;

tmp = prim;

if (prim == NULL) {

cout << "Lista de straturi este vida!" << endl;

return NULL;

}

while (tmp != NULL) {

tmp = tmp->next;

//stergere weights

for (int i = 0; i < prim->neuroni; i++) {

delete[]prim->w[i];

}

delete[]prim->w;

//stergere biases

delete[]prim->b;

//stergere Z

for (int i = 0; i < prim->neuroni; i++) {

delete[]prim->Z[i];

}

delete[]prim->Z;

//stergere strat

delete prim;

prim = tmp;

}

cout << "Lista de straturi a fost stearsa!" << endl;

return NULL;

}

Strat\* forward(Strat\* prim, int n, int m, double\* X) {

/\*for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

cout << \*((X + i \* m) + j) << " ";

}

cout << endl;

}\*/ //test afisare corecta X

Strat\* tmp;

tmp = prim;

if (prim == NULL) {

cout << "Lista de straturi este goala!" << endl;

}

else {

while (tmp != NULL) {

//alocare spatiu Z

tmp->Z = new double\* [tmp->neuroni];

for (int i = 0; i < tmp->neuroni; ++i) {

tmp->Z[i] = new double[m];

}

//calcul Z

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < m; j++) {

tmp->Z[i][j] = 0;

for (int k = 0; k < tmp->neuroni; k++) {

//caz particular primul strat

if (prim == tmp) {

tmp->Z[i][j] += tmp->w[i][k] \* (\*((X + k \* m) + j));

}

//pentru restul de noduri

else {

tmp->Z[i][j] += tmp->w[i][k] \* tmp->prev->Z[k][j];

}

}

}

}

//calcul A

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < m; j++) {

tmp->Z[i][j] = tmp->sigmoid(tmp->Z[i][j]);

}

}

tmp = tmp->next;

}

}

return prim;

}

void afisare\_A(Strat\* prim, int m) {

Strat\* tmp;

tmp = prim;

int counter = 1;

if (prim == NULL) {

cout << "Lista de straturi este goala!" << endl;

}

else {

while (tmp != NULL) {

cout << "A" << counter << ":" << endl;

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < m; j++) {

cout << tmp->Z[i][j] << " ";

}

cout << endl;

}

counter++;

tmp = tmp->next;

}

}

}

double cost\_function(Strat\* prim, double\* Y, int m) {

Strat\* tmp;

tmp = prim;

double \*lnA = new double[m];

double \*lnA\_1 = new double[m];

double sum=0;

if (prim == NULL) {

cout << "Lista de straturi este goala!" << endl;

return 0;

}

else {

while (tmp->next != NULL) {

tmp = tmp->next;// pentru a ajunge pe ultimul strat

}

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < m; j++) {

lnA[j] = log(tmp->Z[i][j]);

lnA\_1[j] = log(1 - tmp->Z[i][j]);

}

}

for (int i = 0; i < m; i++) {

lnA[i] = lnA[i] \* Y[i];

lnA\_1[i] = lnA\_1[i] \* (1 - Y[i]);

}

for (int i = 0; i < m; i++) {

lnA[i] = lnA[i] + lnA\_1[i];

}

for (int i = 0; i < m; i++) {

sum += lnA[i];

}

delete[]lnA;

delete[]lnA\_1;

return ((double)-1/m) \* sum;

}

}

Strat\* backward(Strat\* prim, double\* X, double\* Y, int m, int n) {

double\*\* dZ = NULL;

double\*\* dw = NULL;

double\* db = NULL;

double\*\* dA = NULL;

Strat\* tmp;

tmp = prim;

if (prim == NULL) {

cout << "Lista de straturi este goala!" << endl;

}

else {

while (tmp->next != NULL) {

tmp = tmp->next;// pentru a ajunge pe ultimul strat

}

while (tmp->prev != NULL) {

//alocare spatiu derivate

dZ = new double\* [tmp->neuroni];

for (int i = 0; i < tmp->neuroni; ++i) {

dZ[i] = new double[m];

}

dw = new double\* [tmp->neuroni];

for (int i = 0; i < tmp->neuroni; ++i) {

if (tmp == prim) {

dw[i] = new double[n];

}

else {

dw[i] = new double[tmp->prev->neuroni];

}

}

db = new double[tmp->neuroni];

if (tmp == prim) {

dA = new double\* [n];

for (int i = 0; i < n; ++i) {

dA[i] = new double[m];

}

}

else {

dA = new double\* [tmp->prev->neuroni];

for (int i = 0; i < tmp->prev->neuroni; ++i) {

dA[i] = new double[m];

}

}

//calcul derivate

if (tmp->next == NULL) { //caz particular ultimul strat

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < m; j++) {

dZ[i][j] = Z[i][j] - Y[j];

}

}

}

else {

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < m; j++) {

dZ[i][j] = Z[i][j] - dA[i][j];

}

}

}

if (tmp == prim) { //caz particular primul strat

for (int i = 0; i < tmp->neuroni; i++) {

for (int j = 0; j < n; j++) {

for (int k = 0; k < tmp->neuroni; i++) {

//dw[i][j] += dZ[i][k] + (\*((X + k \* m) + j));

}

}

}

}

tmp = tmp->prev;

}

}

return prim;

}

};

int main() {

double X[4][10] = {

{0,1,0,1,0,1,0,1,0,1},

{0,0,1,1,0,0,1,1,0,0},

{0,0,0,0,1,1,1,1,0,0},

{0,0,0,0,0,0,0,0,1,1},

}; //intrarea in retea(trainig set) 784(28x28) -> pixeli, 100 -> fotografi(learning set)

double Y[10] = { 0,0,0,1,0,1,1,0,0,1 }; //training set expected output

int dim\_vect\_intrare = sizeof(X) / sizeof(X[0]); //nr. de pixeli la intrarea in retea (784 normal, dar pentru verificare am folosit 4)

int m = sizeof(X[0]) / sizeof(X[0][0]); //m = numarul de exemple pentru antrenarea retelei(nr de poze)

double Cost=0; //erroare medie pentru toate exemplele de antrenare

Strat\* prim = NULL; //pointer catre primul strat din retea

//creare retea + afisare functionare corecta

prim = prim->creare(prim, dim\_vect\_intrare);

cout << endl;

prim->afisare(prim);

cout << endl;

//afisare ponderi

prim->afisare\_w\_b(prim, dim\_vect\_intrare);

cout << endl;

//etapa forward

prim->forward(prim, dim\_vect\_intrare, m, (double\*)X);

prim->afisare\_A(prim, m);

cout << endl;

//calcul cost

Cost = prim->cost\_function(prim, Y, m);

cout << "Cost function:\n" << Cost << endl; cout << endl;

//etapa backward

prim->backward(prim, (double\*)X, Y, m, dim\_vect\_intrare);

//dezalocare memorie

prim = prim->stergere(prim);

prim->afisare(prim);

return 0; }