Samoilă George Tiberiu
Gr. 321AA

Tema Optimizări

Clasificarea binară a pacienților cu diabet

Introducere

În acest proiect am antrenat o rețea neuronală superficială pentru rezolvarea unei sarcini de clasificare binară. Setul de date ales este [Pima Indians Diabetes], ce conține informații medicale despre pacienți.
Am implementat două metode de optimizare de ordinul I – Gradient Descent (GD) și Stochastic Gradient Descent (SGD) – pentru a minimiza funcția de pierdere și a îmbunătăți performanța rețelei.

Detalii Despre Baza de Date Utilizată

Setul de date "Pima Indians Diabetes Database" conține următoarele caracteristici pentru fiecare pacient:

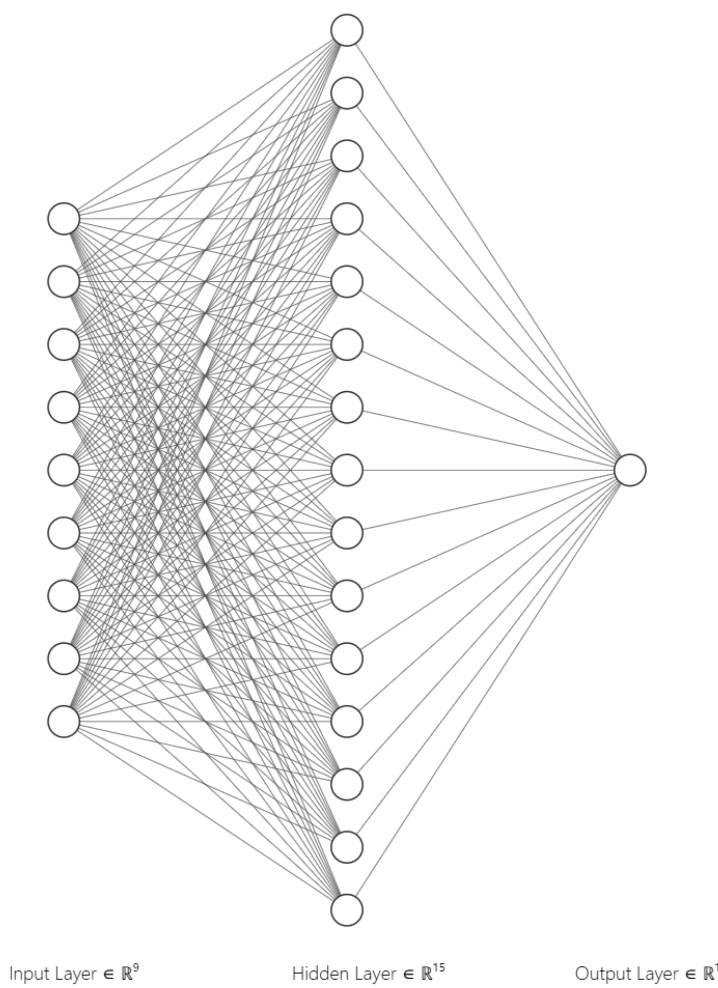
1. Număr sarcini anterioare (Pregnancies) — numeric (ex: 0, 1, 5, 10).
2. Glicemie (Glucose) — numeric: nivelul glicemiei la două ore după un test de glucoză.
3. Presiune arterială (BloodPressure) — numeric: presiunea diastolică (mm Hg).
4. Grosimea pliului cutanat (SkinThickness) — numeric: grosimea pielii de pe triceps (mm).
5. Nivelul de insulină (Insulin) — numeric: concentrația de insulină serică (mu U/ml).
6. Indice de masă corporală (BMI) — numeric: calculat ca greutate în kg / (înălțime în m)^2.
7. Funcție pedigree diabet (DiabetesPedigreeFunction) — numeric: estimarea riscului ereditar de diabet.
8. Vârstă (Age) — numeric: vârsta pacientului (ani).

Target (variabila de predicție):

- Outcome — binar:
    - 1 = Pacient diagnosticat cu diabet,
    - 0 = Pacient fără diagnostic de diabet.

Arhitectura Rețelei

Rețeaua neuronală utilizată are un singur strat ascuns cu 15 neuroni și funcția de activare ASU definită ca $g(z) = z\sin(z)$ cu derivata $g'(z) = \sin(z) + z\cos(z)$. Stratul de intrare are 8 neuroni pentru date si unul este pentru bias.
Ieșirea rețelei este activată printr-o funcție sigmoid pentru a obține o probabilitate între 0 și 1.



Funcția de Cost și Optimizare

Pentru clasificare, am folosit funcția de cost Entropie Încrucișată Binara (Binary Cross-Entropy).
Scopul optimizării este minimizarea pierderii prin actualizarea greutăților rețelei în direcția gradientului negativ.

Metode de Optimizare

Gradient Descent (GD)

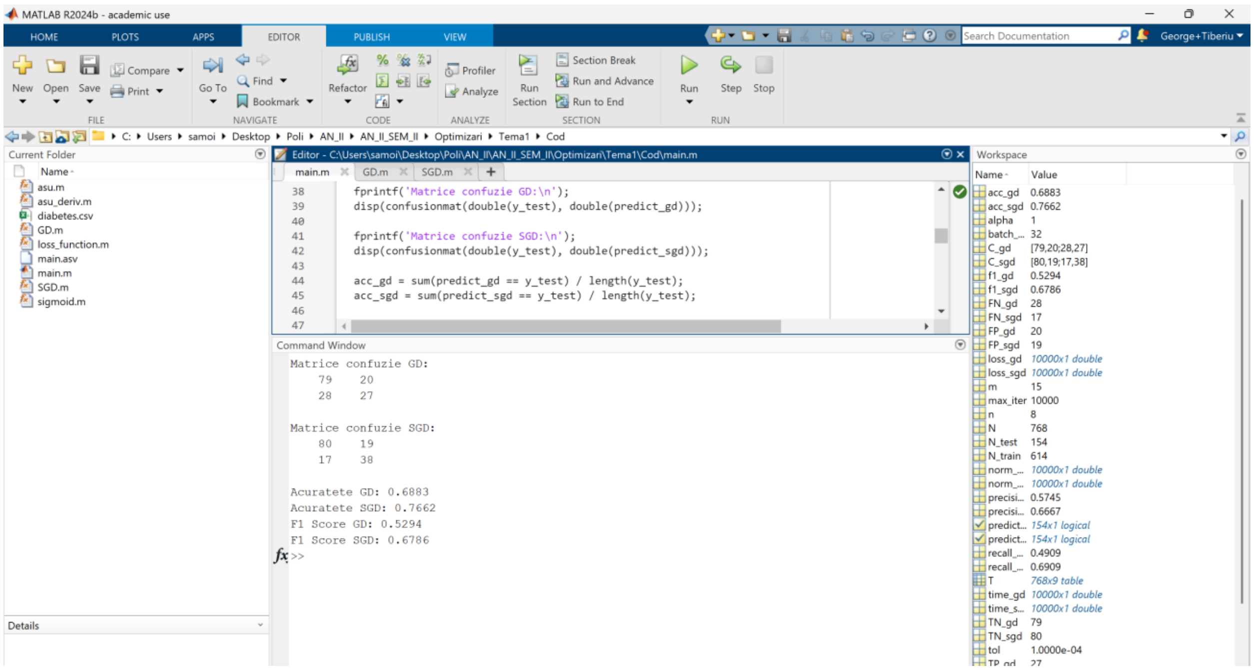
Gradient Descent actualizează greutățile folosind întreg setul de antrenare la fiecare pas.
Rata de învățare aleasă a fost $\alpha=0.01$, iar numărul maxim de iterații a fost 1000.
Oprirea antrenării se face fie după atingerea numărului maxim de iterații, fie dacă norma gradientului scade sub o toleranță 1e-8.

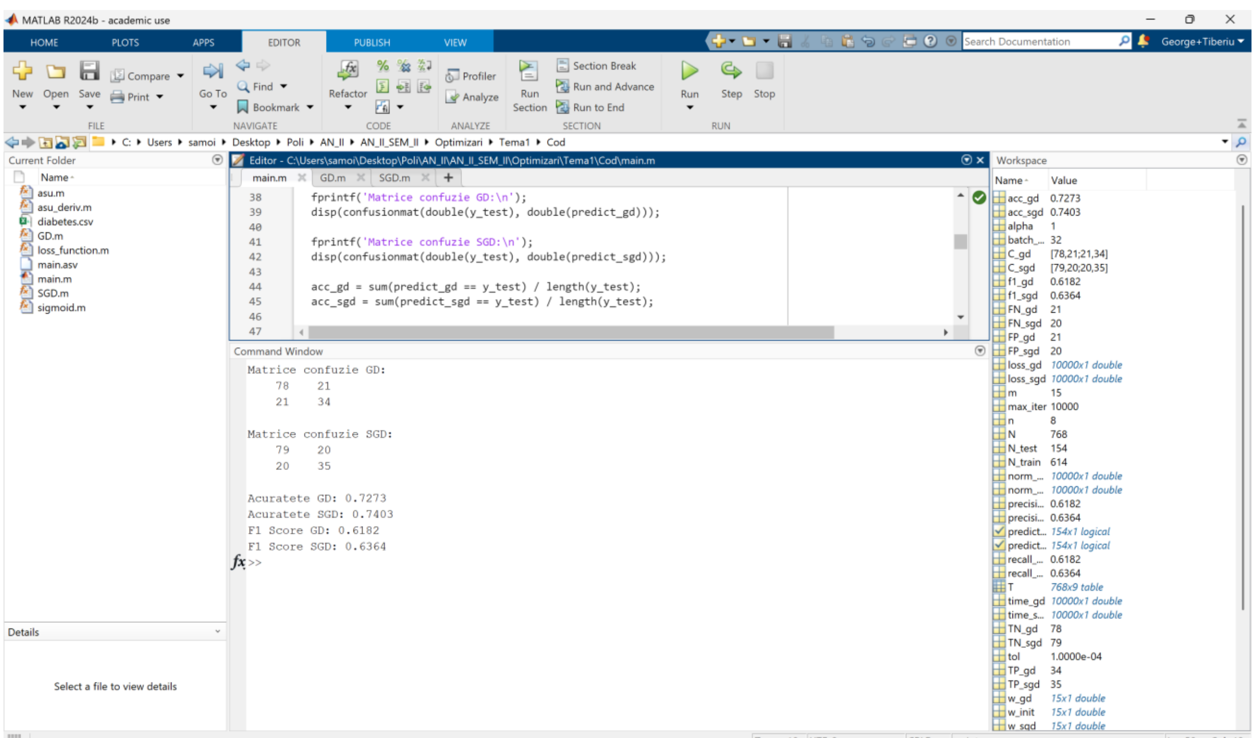
Stochastic Gradient Descent (SGD)

Metoda gradient stochastic a fost implementată similar cu metoda gradient, dar în loc să folosim toate exemplele de antrenare la fiecare iterație, am ales un număr mic de exemple aleatoare. Acest lucru reduce costul computațional și poate accelera convergența algoritmului. Am afișat, de asemenea, rezultatele sub formă de grafice și am testat modelul pe datele de testare.
Această metodă introduce variații aleatoare în procesul de învățare și poate accelera convergența.
Și în acest caz, am folosit toleranța pe norma gradientului pentru oprirea anticipată.

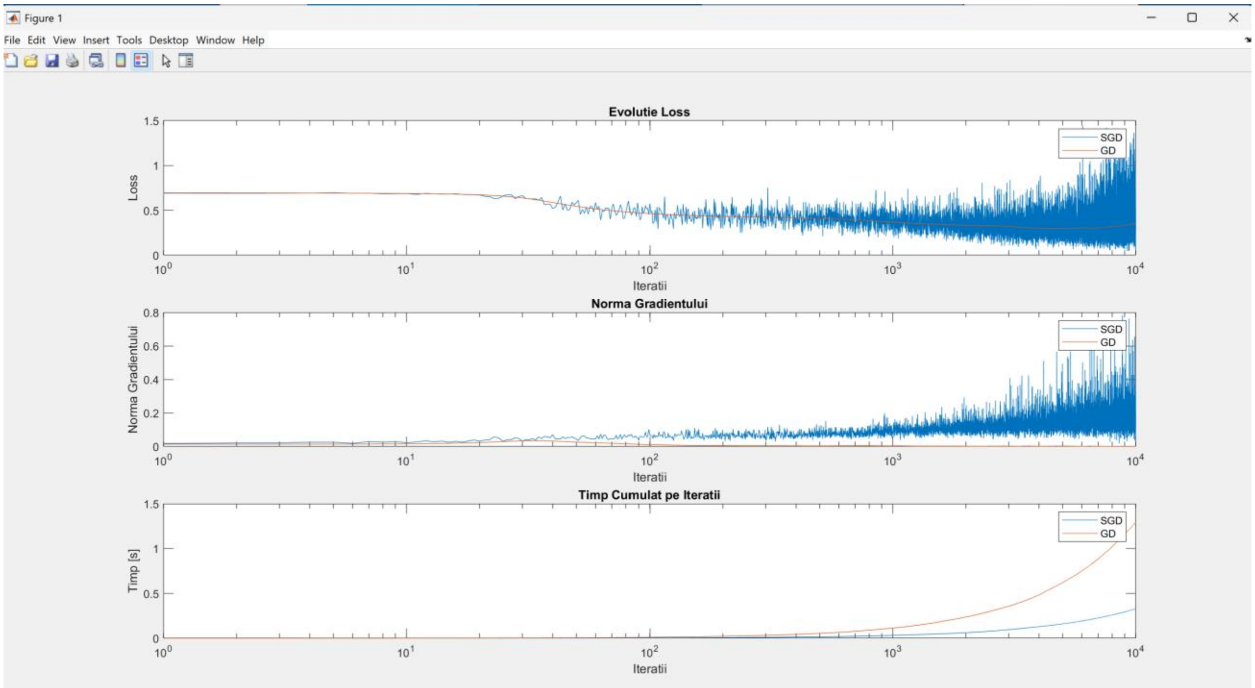
Rezultate

Am antrenat rețeaua folosind atât GD, cât și SGD și am evaluat performanța pe setul de testare.
Exemple de rulare:





Am reprezentat grafic evoluția funcției loss și a normei gradientului și a vitezei algoritmilor.



Comentarii

Se observă că niciuna dintre metodele testate nu oferă o predicție perfectă, ceea ce este de așteptat având în vedere complexitatea setului de date și natura problemei.
Atât metoda Gradient Descent (GD), cât și metoda Stochastic Gradient Descent (SGD) se pot bloca în minime locale sau în puncte saddle.

Datorită caracterului său stocastic, metoda SGD a obținut rezultate ușor superioare, fiind capabilă să evite mai eficient unele capcane locale prin variațiile introduse de mini-batch-uri.

Din punct de vedere al convergenței, metoda GD oferă o traiectorie mai stabilă și un control mai bun asupra evoluției funcției loss, însă SGD permite utilizarea unui număr mai mare de pași de optimizare și o adaptare mai rapidă în anumite regiuni ale funcției de eroare.

Alegerea metodei optime depinde astfel de raportul dorit între stabilitatea convergenței și viteza de antrenare.

Anexă

Codul principal – main

```
close all; clear; clc;

%% Incarcare si preprocesare date
T = readtable('diabetes.csv');
X = table2array(T(:,1:end-1));
y = table2array(T(:,end));

[N, n] = size(X);
X = (X - mean(X)) ./ std(X);%normalizare
Xbar = [X, ones(N,1)];

N_train = round(0.8 * N);
N_test = N - N_train;

X_train = Xbar(1:N_train,:);
y_train = y(1:N_train);
X_test = Xbar(N_train+1:end,:);
y_test = y(N_train+1:end);

%% Parametri
m = 15;
max_iter = 10000;
alpha = 1;
batch_size = 32;
tol = 1e-8;

Xw_init = randn(n+1, m)*0.1;
w_init = randn(m, 1)*0.1;

%% Antrenare modele
[Xw_gd, w_gd, loss_gd, norm_grad_gd, time_gd] = GD(X_train, y_train, Xw_init, w_init, alpha,
max_iter);
[Xw_sgd, w_sgd, loss_sgd, norm_grad_sgd, time_sgd] = SGD(X_train, y_train, Xw_init, w_init,
alpha, max_iter, batch_size);

%% Evaluare
predict_gd = sigmoid(asu(X_test * Xw_gd) * w_gd) >= 0.5;
predict_sgd = sigmoid(asu(X_test * Xw_sgd) * w_sgd) >= 0.5;

fprintf('Matrice confuzie GD:\n');
disp(confusionmat(double(y_test), double(predict_gd)));

fprintf('Matrice confuzie SGD:\n');
disp(confusionmat(double(y_test), double(predict_sgd)));

acc_gd = sum(predict_gd == y_test) / length(y_test);
acc_sgd = sum(predict_sgd == y_test) / length(y_test);

fprintf('Acuratete GD: %.4f\n', acc_gd);
fprintf('Acuratete SGD: %.4f\n', acc_sgd);

%% Rezultate
figure;
subplot(3,1,1);
semilogx(loss_sgd);
hold on;
semilogx(loss_gd);
legend('SGD','GD');
title('Evolutie Loss');
xlabel('Iteratii');
```

```
ylabel('Loss');


subplot(3,1,2);
semilogx(norm_grad_sgd);
hold on;
semilogx(norm_grad_gd);
legend('SGD','GD');
title('Norma Gradientului');
xlabel('Iteratii');
ylabel('Norma Gradientului');


subplot(3,1,3);
semilogx(cumsum(time_sgd));
hold on;
semilogx(cumsum(time_gd));
legend('SGD','GD');
title('Timp Cumulat pe Iteratii');
xlabel('Iteratii');
ylabel('Timp [s]');


C_gd = confusionmat(double(y_test), double(predict_gd));
TN_gd = C_gd(1,1);
FP_gd = C_gd(1,2);
FN_gd = C_gd(2,1);
TP_gd = C_gd(2,2);

precision_gd = TP_gd / (TP_gd + FP_gd);
recall_gd = TP_gd / (TP_gd + FN_gd);
f1_gd = 2 * (precision_gd * recall_gd) / (precision_gd + recall_gd);

fprintf('F1 Score GD: %.4f\n', f1_gd);


C_sgd = confusionmat(double(y_test), double(predict_sgd));
TN_sgd = C_sgd(1,1);
FP_sgd = C_sgd(1,2);
FN_sgd = C_sgd(2,1);
TP_sgd = C_sgd(2,2);

precision_sgd = TP_sgd / (TP_sgd + FP_sgd);
recall_sgd = TP_sgd / (TP_sgd + FN_sgd);
f1_sgd = 2 * (precision_sgd * recall_sgd) / (precision_sgd + recall_sgd);

fprintf('F1 Score SGD: %.4f\n', f1_sgd);
```

Funcția GD

```
function [Xw, w, loss, norm_grad, time_vec] = GD(X, y, Xw_init, w_init, alpha, max_iter)
    Xw = Xw_init;
    w = w_init;
    loss = zeros(max_iter, 1);
    norm_grad = zeros(max_iter, 1);
    time_vec = zeros(max_iter, 1);

    tol = 1e-8; % toleranta pe norma gradientului

    for iter = 1:max_iter
        tic;

        % Forward
        Z = X * Xw;
```

```
        G = asu(Z);
        y_pred = sigmoid(G*w);

        % Loss
        loss(iter) = loss_function(y, y_pred);

        % Backward
        delta = (y_pred - y) .* y_pred .* (1 - y_pred);
        dG = asu_deriv(Z);

        grad_w = G' * delta / length(y);
        grad_Xw = (X' * (delta * w' .* dG)) / length(y);

        % Norm gradient
        norm_grad(iter) = norm([grad_Xw(:); grad_w]);

        % Conditie de oprire
        if norm_grad(iter) < tol
            loss = loss(1:iter);
            norm_grad = norm_grad(1:iter);
            time_vec = time_vec(1:iter);
            break;
        end

        % Update parametri
        w = w - alpha * grad_w;
        Xw = Xw - alpha * grad_Xw;

        time_vec(iter) = toc;
    end
end
```

Funcția SGD

```
function [Xw, w, loss, norm_grad, time_vec] = SGD(X, y, Xw_init, w_init, alpha, max_iter,
batch_size)
    Xw = Xw_init;
    w = w_init;
    loss = zeros(max_iter, 1);
    norm_grad = zeros(max_iter, 1);
    time_vec = zeros(max_iter, 1);

    tol = 1e-8; % toleranta pe norma gradientului

    for iter = 1:max_iter
        tic;

        % Batch random
        idx_batch = randsample(length(y), batch_size);
        X_batch = X(idx_batch,:);
        y_batch = y(idx_batch);

        % Forward
        Z = X_batch * Xw;
        G = asu(Z);
        y_pred = sigmoid(G*w);

        % Loss
        loss(iter) = loss_function(y_batch, y_pred);

        % Backward
        delta = (y_pred - y_batch) .* y_pred .* (1 - y_pred);
        dG = asu_deriv(Z);
```

```
        grad_w = G' * delta / batch_size;
        grad_Xw = (X_batch' * (delta * w' .* dG)) / batch_size;

        % Norm gradient
        norm_grad(iter) = norm([grad_Xw(:); grad_w]);

        % Conditie de oprire
        if norm_grad(iter) < tol
            loss = loss(1:iter);
            norm_grad = norm_grad(1:iter);
            time_vec = time_vec(1:iter);
            break;
        end

        % Update parametri
        w = w - alpha * grad_w;
        Xw = Xw - alpha * grad_Xw;

        time_vec(iter) = toc;
    end
end
```

Funcția loss_function

```
function L = loss_function(e, y_pred)
    L = -(1/length(e)) * sum(e.*log(y_pred + 1e-8) + (1-e).*log(1 - y_pred + 1e-8));
end
```

Funcția asu

```
function g = asu(z)
    g = z .* sin(z);
end
```

Funcția asu_deriv

```
function dg = asu_deriv(z)
    dg = sin(z) + z .* cos(z);
end
```

Funcția sigmoid

```
function y = sigmoid(z)
    y = 1 ./ (1 + exp(-z));
end
```