Νευρο-ασαφής Έλεγχος

9ο Εξάμηνο 2023 – 2024

Assignment 1 – Solutions

Ζαρίφης Στέλιος – el20435
Email: el20435@mail.ntua.gr

Contents

1 Σύγκλιση μεθόδων Gradient Descend και Newton

Gradient Descent Το Gradient Descent είναι ένας επαναληπτικός αλγόριθμος βελτιστοποίησης που βασίζεται στην ελαχιστοποίηση μιας συνάρτησης. Δεδομένης μιας διαφορίσιμης objective συνάρτησης $f(x)$, ο αλγόριθμος ανανεώνει τις παραμέτρους x προς την αντίθετη κατεύθυνση της κλίσης της f ως προς x. Ο κανόνας ανανέωσης είναι:

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$

όπου x_k το τρέχον διάνυσμα παραμέτρων, $\nabla f(x_k)$ το gradient της f στο x_k, και α το learning rate.

Newton's Method Η μέθοδος Newton είναι επίσης μια επαναληπτική τεχνική βελτιστοποίησης που επιδιώκει να υπολογίσει τις ρίζες μιας συνάρτησης. Για προβλήματα βελτιστοποίησης, χρησιμοποιείται στην εύρεση ελαχίστου μιας διαφορίσιμης συνάρτησης. Ο κανόνας ενημέρωσης είναι:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

όπου x_k η τρέχουσα παράμετρος, $f'(x_k)$ η πρώτη παράγωγος και $f''(x_k)$ η δεύτερη παράγωγος της αντικειμενικής συνάρτησης $f(x)$ στο x_k.

1.1 Μέθοδος Gradient Descend

Η συνάρτηση μας, μαζί με τις παραγώγους της είναι:

$$f(x_1, x_2) = x_1^2 + (x_2 - 1)^2 + (x_1 - x_2)^4$$

$$\nabla f(x) = \begin{bmatrix} 2x_1 + 4(x_1 - x_2)^3 \\ 2(x_2 - 1) + 4(x_2 - x_1)^3 \end{bmatrix}$$

$$\nabla^2 f = \begin{bmatrix} 2 + 12(x_1 - x_2)^2 & -12(x_1 - x_2)^2 \\ -12(x_1 - x_2)^2 & 2 + 12(x_1 - x_2)^2 \end{bmatrix}$$

Εφαρμόζουμε τη μέθοδο Gradient Descend

```
% Random vector with std = 1
x = randn(2, 1);
N = 30;
step_size = 0.01;
x_plot = x;

% Gradient Descent for N steps
for i = 1:N
  grad = [2 * x(1) + 4 * (x(1) - x(2))^3;
  2 * (x(2) - 1) + 4 * (x(2) - x(1))^3];
  x_new = x - step_size * grad;
  x_plot = [x_plot, x_new];
  Val(i) = x(1)^2 + (x(2) - 1)^2 + (x(1) - x(2))^4;
  x = x_new;
end

figure
plot(x_plot, '-x')

```

```
figure
plot(x_plot', '-x')

figure
plot(x_plot(1,:), x_plot(2,:), '-x')
```

Σχεδιάζουμε με 3 τρόπους την εξέλιξη των 2 συντεταγμένων:

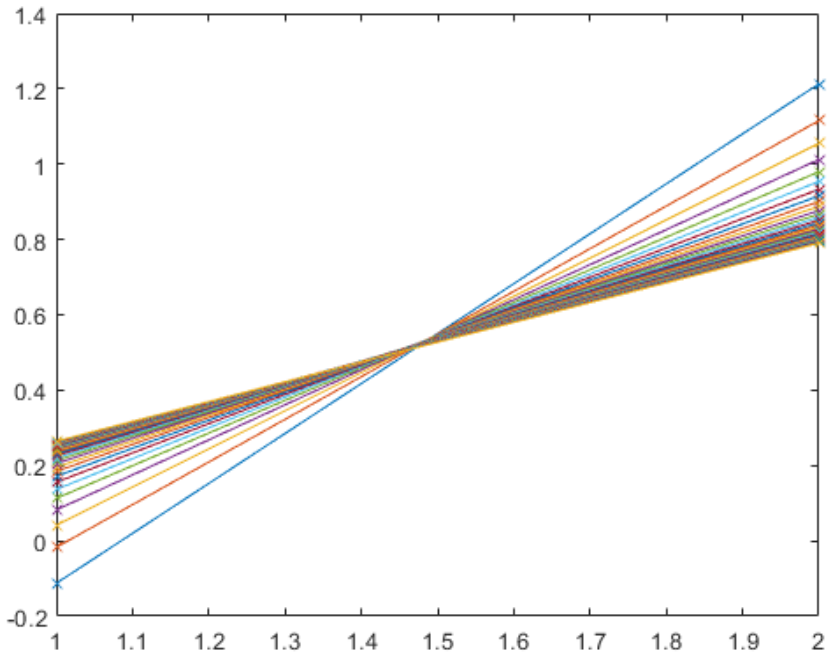


Figure 1: Plot $[x_1, x_2]$

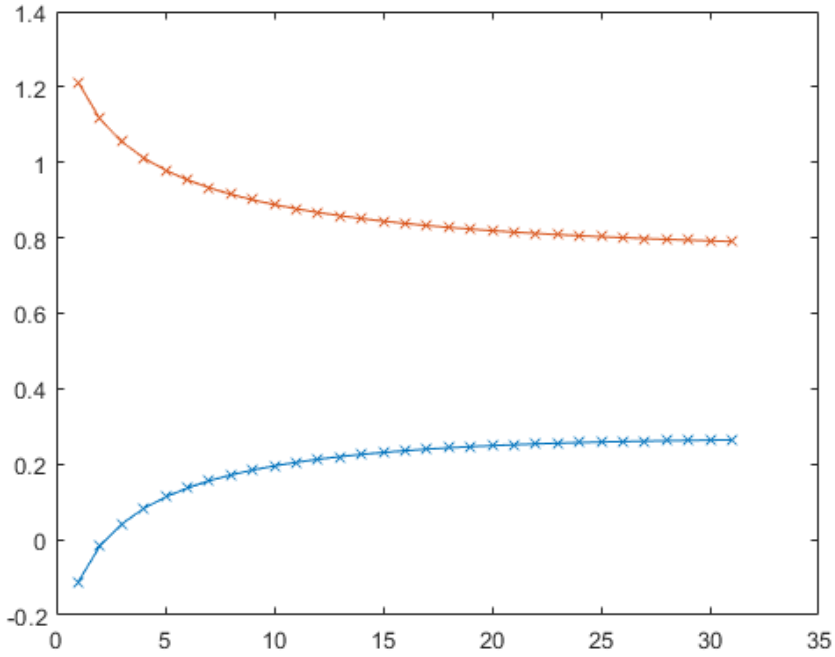


Figure 2: Plot $[x_1, x_2]^T$

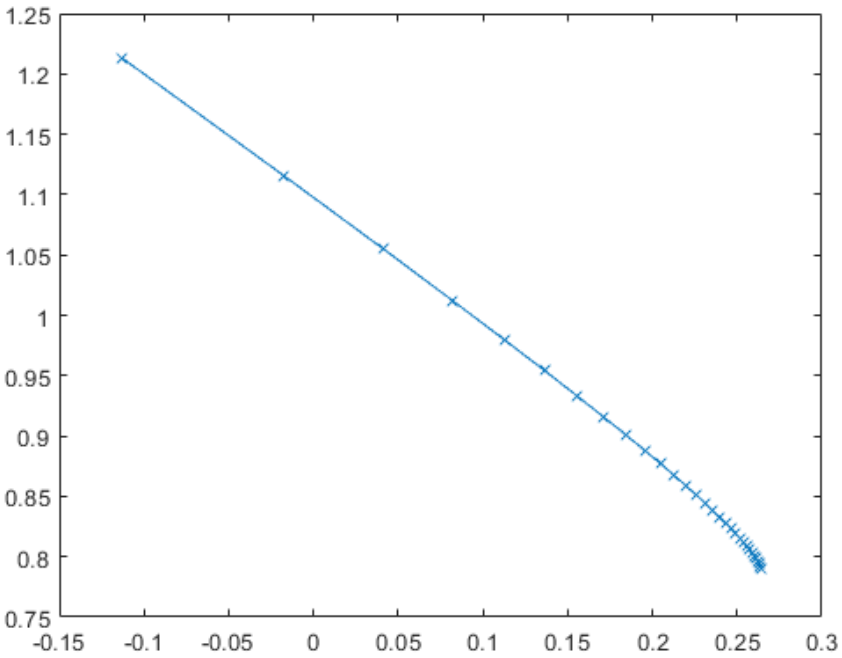


Figure 3: Plot the points (x_1, x_2)

Παρατηρούμε ότι τα σημεία (x_1, x_2) κινούνται σχεδόν πάνω σε ευθεία γραμμή. Αυτό όμως εξαρτάται από τις αρχικές συνθήκες.

Ακόμα, σχεδιάζουμε πώς εξελίσσεται η τιμή της συνάρτησης κατά τη διάρκεια του Gradient Descend και μια

συνηθισμένη οπτικοποίηση του ρυθμού σύγκλισης, που είναι ο λογάριθμος της διαφοράς της τελευταίας τιμής της συνάρτησής f από την ίδια τη συνάρτηση rate $= \log(f(x_1, x_2) - f(x_1^{final}, x_2^{final}))$.

```
figure
plot(Val)

% Convergence Rate
figure
plot(log(Val - Val(end)))
```

Και βλέπουμε πως η σύγκλιση είναι σχεδόν εκθετική

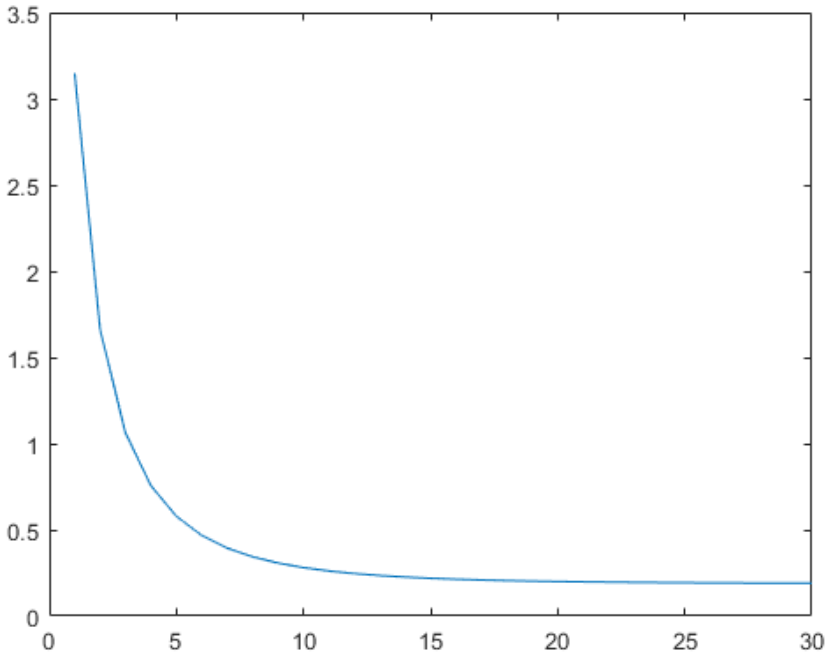


Figure 4: Convergence Visualization

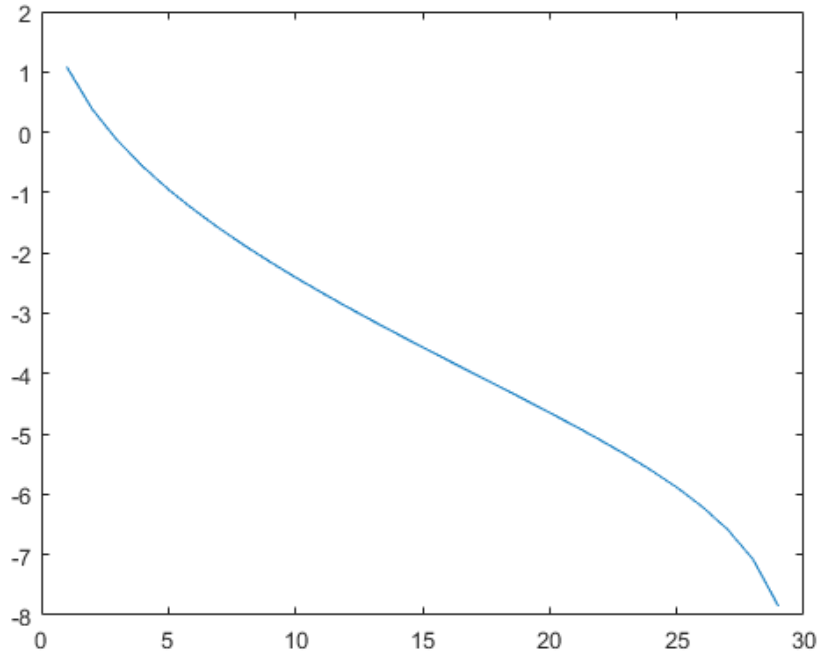


Figure 5: Convergence Rate Visualization

1.2 Μέθοδος Newton

Στο ίδιο πρόβλημα εφαρμόζουμε τη μέθοδο Newton.

```
% Newton's Method
x_n = randn(2, 1);
x_plot_newton = x_n;
step_size = 1;
for i = 1:N
  grad = [2 * x_n(1) + 4 * (x_n(1) - x_n(2))^3;
  2 * (x_n(2) - 1) + 4 * (x_n(2) - x_n(1))^3];
  x_newton = x_n - step_size * inv(hessian(x_n)) * grad;
  x_plot_newton = [x_plot_newton, x_newton];
  Val_newton(i) = x_n(1)^2 + (x_n(2) - 1)^2 + (x_n(1) - x_n(2))^4;
  x_n = x_newton;
end

% Plotting the results
figure;
plot(x_plot_newton, '-o');
title("Parameter Evolution with Gradient Descent and Newton's Method");

figure;
plot(x_plot', '-x', 'DisplayName', 'Gradient Descent');
hold on;
plot(x_plot_newton', '-o', 'DisplayName', "Newton's Method");
title("Parameter Evolution with Gradient Descent and Newton's Method (Separate
   Axes)");
legend;
```

```
25
26 figure;
27 plot(x_plot(1,:), x_plot(2,:), '-x', 'DisplayName', 'Gradient Descent');
28 hold on;
29 plot(x_plot_newton(1,:), x_plot_newton(2,:), '-o', 'DisplayName', "Newton's
      Method");
30 title('Parameter Trajectory in 2D Space');
31 legend;
32
33 figure;
34 plot(Val, 'DisplayName', 'Gradient Descent');
35 hold on;
36 plot(Val_newton, 'DisplayName', "Newton's Method");
37 title('Objective Function Value');
38 legend;
39
40 % Logarithm of the decrease in objective function value
41 figure;
42 plot(log(Val - Val(end)), 'DisplayName', 'Gradient Descent');
43 hold on;
44 plot(log(Val_newton - Val_newton(end)), 'DisplayName', "Newton's Method");
45 title('Logarithm of Objective Function Decrease');
46 legend;
47
48 function H = hessian(x)
49   H = [2 + 12*(x(1) - x(2))^2, -12*(x(1) - x(2))^2;
50   -12*(x(1) - x(2))^2, 2 + 12*(x(2) - x(1))^2];
51 end
```

Και σχεδιάζουμε τα αντίστοιχα γραφήματα για να συγκρίνουμε τις μεθόδους

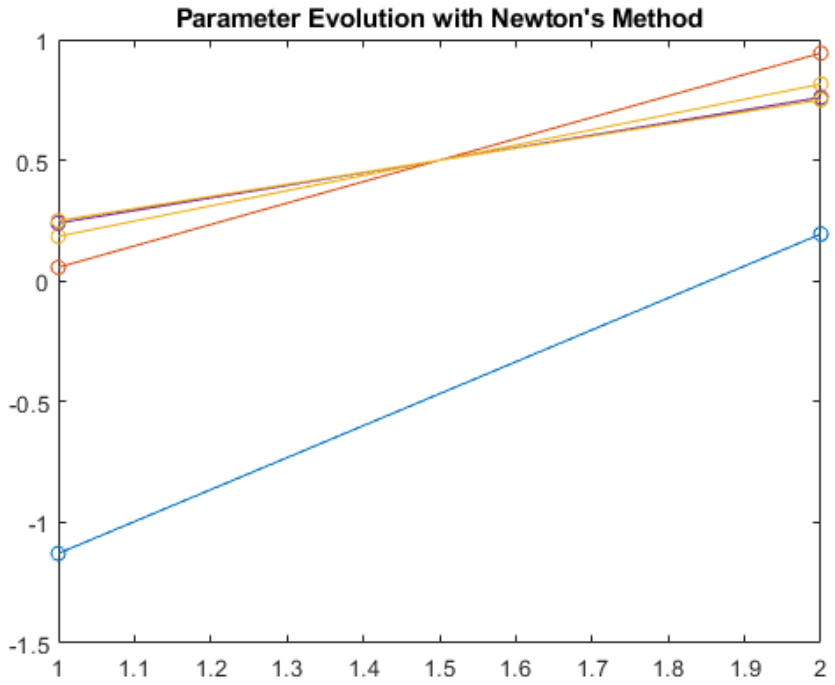


Figure 6: Plot $[x_1, x_2]$

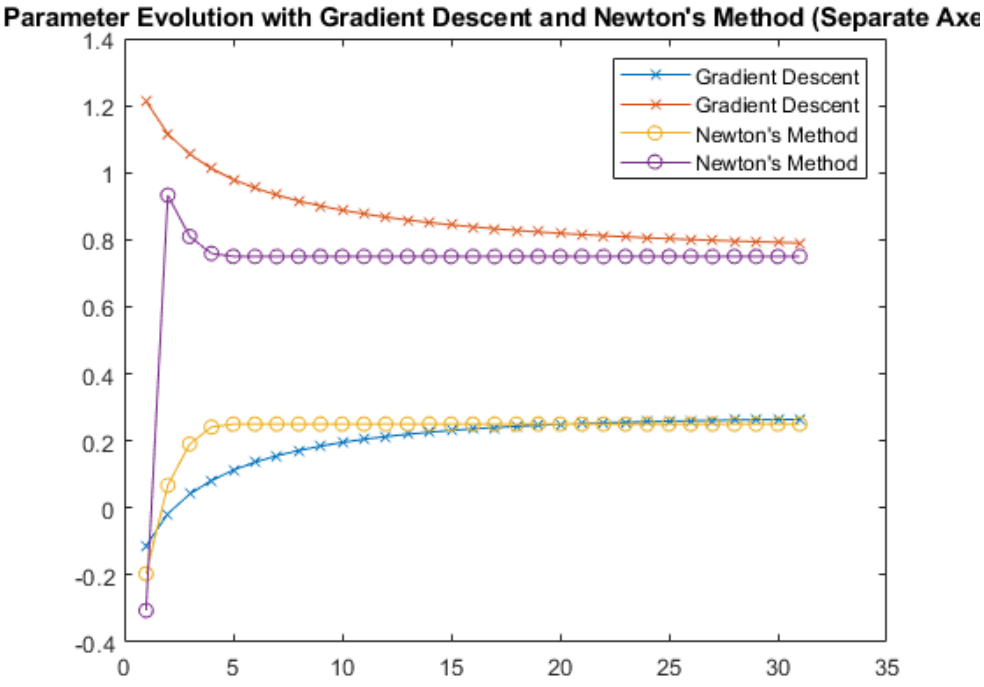


Figure 7: Plot $[x_1, x_2]^T$ for Gradient Descent and Newton's Methods

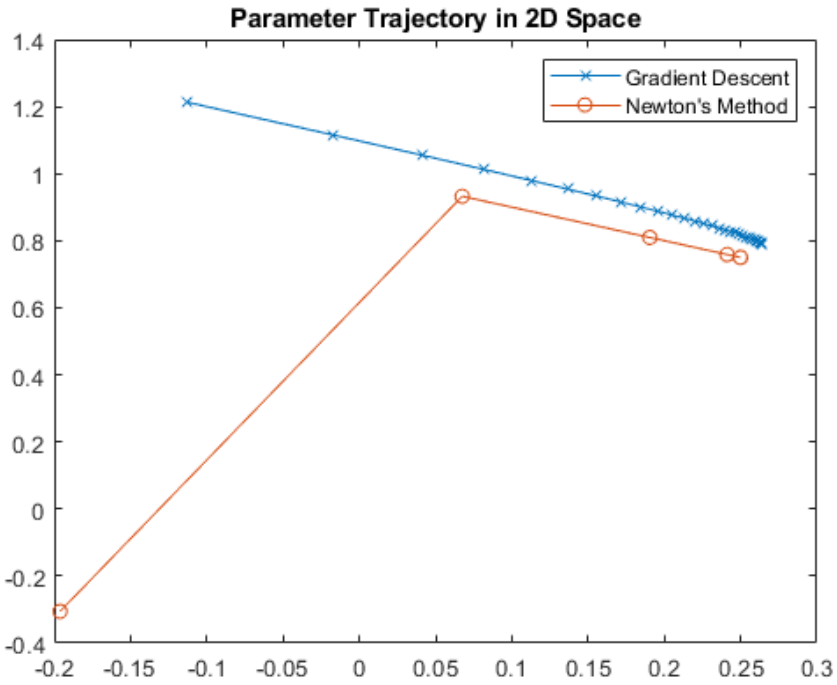


Figure 8: Gradient Descent and Newton's Methods points (x_1, x_2)

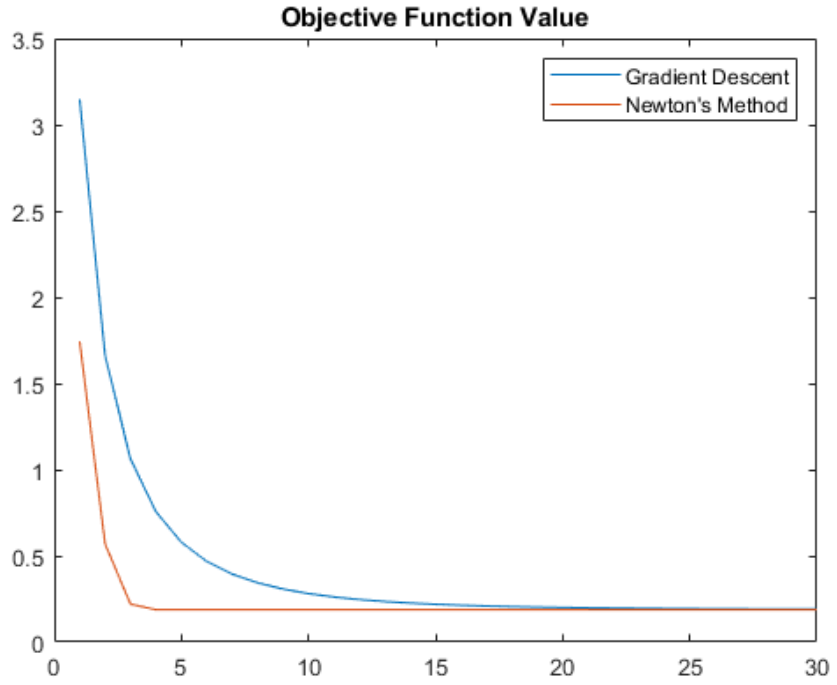


Figure 9: Convergence Visualization

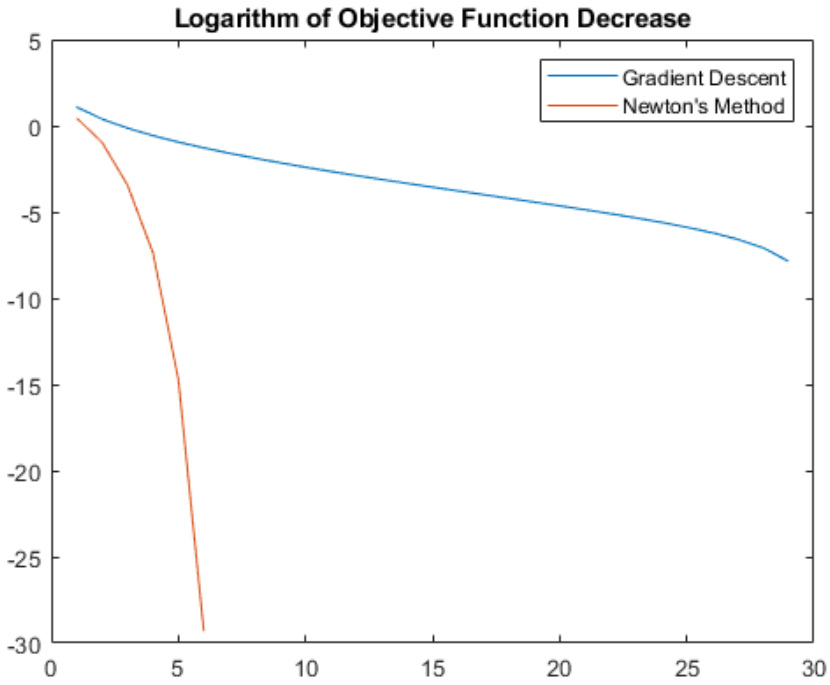


Figure 10: Convergence Rates Comparison

Μπορούμε να παρατηρήσουμε ότι η μέθοδος Newton οδηγεί σε σημαντικά ταχύτερη σύγκλιση. Αυτό συμβαίνει διότι, σε αντίθεση με τη μέθοδο Gradient Descend, το βήμα που κάνουμε κατά φορά αντίθετη του gradient είναι μεταβλητό. Μάλιστα, αφού χρησιμοποιούμε τον αντίστροφο Εσσιανό πίνακα, το βήμα μικραίνει όσο πιο απότομα μεταβάλλεται το gradient και μεγαλώνει όσο πιο ομαλό είναι. Με αυτόν τον έξυπνο τρόπο, η μέθοδος αυτόματα καταλαβαίνει ότι αν το gradient είναι απότομο, το τοπικό ελάχιστο είναι κατά πάσα πιθανότητα πολύ κοντά, ενώ όσο πιο ομαλό είναι, το τοπικό ελάχιστο πιθανώς βρίσκεται μακριά.

2 Ταχύτητα Σύγκλισης Gradient Descend και Condition Number

Η συνάρτηση που θα εξετάσουμε, μαζί με την παράγωγό της, είναι

$$f(x_1, x_2) = Ax_1^2 + \frac{1}{A}x_2^2$$
$$\nabla f(x) = \begin{bmatrix} 2Ax_1 \\ 2x_2/A \end{bmatrix} = \begin{bmatrix} 2A & 0 \\ 0 & 2/A \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$x^+ = x - \alpha \begin{bmatrix} 2A & 0 \\ 0 & 2/A \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1-2\alpha A & 0 \\ 0 & 2\alpha/A \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Για να έχουμε ευστάθεια, πρέπει

$$|1 - 2\alpha A| < 1 \Rightarrow \begin{cases} \alpha < 1/A \\ \alpha < A \end{cases}$$

Επιλέγουμε $A = 10$ και $\alpha = 0.04$, οπότε

$$x^+ = \begin{bmatrix} 0.2 & 0 \\ 0 & 1 - 0.008 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.2 & 0 \\ 0 & 0.992 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Εξετάζουμε τι συμβαίνει ύστερα από 20 και 100 βήματα:

```
a = 0.04;
ALPHA = 10;

fprintf("For a = 0.04 and A = 10, matrix A is:\n")
A = [1 - a*a*ALPHA 0; 0 1 - 2*a/ALPHA]
% Calculate and display the condition number
condition_number = cond(A);
fprintf('Condition number of the matrix A: %f\n', condition_number);
fprintf('After 20 steps:\n')
A^20
fprintf('After 100 steps:\n')
A^100

%%
a = 0.4;
ALPHA = 0.5;

fprintf("For a = 0.4 and A = 0.5, matrix A is:\n")
A = [1 - a*a*ALPHA 0; 0 1 - 2*a/ALPHA]
% Calculate and display the condition number
condition_number = cond(A);
fprintf('Condition number of the matrix A: %f\n', condition_number);
fprintf('After 20 steps:\n')
A^20
fprintf('After 100 steps:\n')
A^100
fprintf('The convergence is slower\n')
fprintf('Bigger condition number leads to slower convergence\n');
```

Και το αποτέλεσμα φαίνεται εδώ:

```
For a = 0.04 and A = 10, matrix A is:

A =

 0.9840         0
 0    0.9920

Condition number of the matrix A: 1.008130
After 20 steps:

ans =

 0.7243         0
 0    0.8516

After 100 steps:

ans =

 0.1993         0
 0    0.4479

For a = 0.4 and A = 0.5, matrix A is:

A =

 0.9200         0
 0   -0.6000

Condition number of the matrix A: 1.533333
After 20 steps:

ans =

 0.1887         0
 0    0.0000

After 100 steps:

ans =

 1.0e-03 *

 0.2392         0
 0    0.0000

The convergence is slower
Bigger condition number leads to slower convergence
```

[Συμπεράσματα]

Εφαρμόζουμε τη μέθοδο της ορμής με μια αυθαίρετη αρχική μαντεψιά:

$$v_{k+1} = a_1 v_k - a_2 \nabla f(x_k), a_1 < 1 \text{for damping}$$
$$x_{k+1} = x_k + v_{k+1}$$

```
% Apply the method of momentum
fprintf('Applying the method of momentum:\n');
a = 0.04;
ALPHA = 10;
% Some arbitrary initial guess
x = [.6; -.4];

% Momentum parameter
alpha1 = 0.1;
alpha2 = 0.1;
v = 0;

for i = 1:200
  grad = [2 * ALPHA * x(1);
  2 * x(2) / ALPHA];
  v = alpha1 * v - alpha2 * grad;
  % Update with momentum
  x = x + v;

  % Display the result at every iteration
  if mod(i, 10) == 0
    fprintf('Iteration %d: x = [%f, %f]\n', i, x(1), x(2));
  end
  plot_x1(i) = x(1);
  plot_x2(i) = x(2);
end

plot(plot_x2, plot_x1)
```

Και παρατηρούμε σύγκλιση:

```
Applying the method of momentum:
Iteration 10: x = [0.059861, -0.320140]
Iteration 20: x = [0.004395, -0.255559]
Iteration 30: x = [0.000323, -0.204006]
Iteration 40: x = [0.000024, -0.162852]
Iteration 50: x = [0.000002, -0.130000]
Iteration 60: x = [0.000000, -0.103776]
Iteration 70: x = [0.000000, -0.082841]
Iteration 80: x = [0.000000, -0.066130]
Iteration 90: x = [0.000000, -0.052790]
Iteration 100: x = [0.000000, -0.042141]
Iteration 110: x = [0.000000, -0.033640]
Iteration 120: x = [0.000000, -0.026854]
Iteration 130: x = [0.000000, -0.021436]
```

```
Iteration 140: x = [0.000000, -0.017112]
Iteration 150: x = [0.000000, -0.013660]
Iteration 160: x = [0.000000, -0.010905]
Iteration 170: x = [0.000000, -0.008705]
Iteration 180: x = [0.000000, -0.006949]
Iteration 190: x = [0.000000, -0.005547]
Iteration 200: x = [0.000000, -0.004428]
```

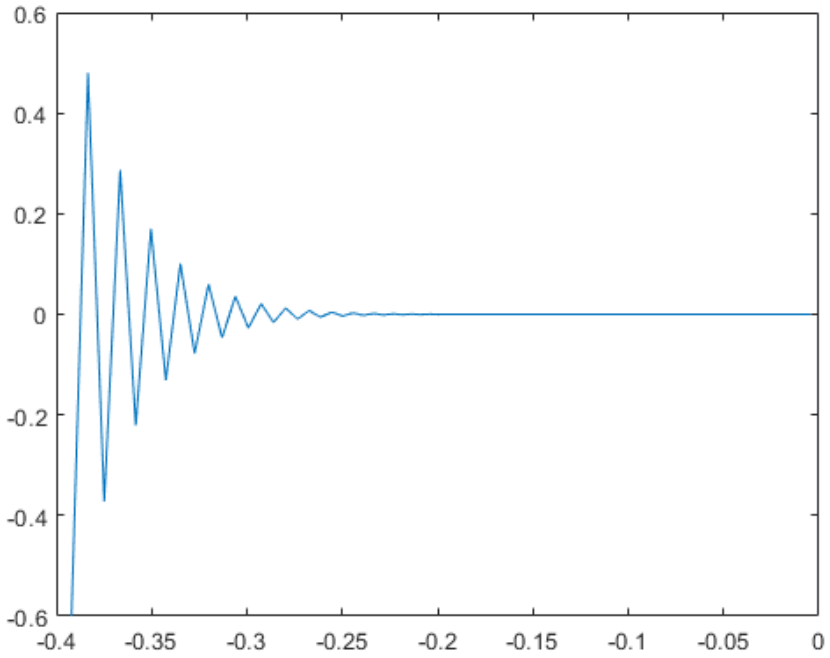


Figure 11: Momentum Method Convergence

3 Condition Number Συναρτήσει της Διάστασης Πίνακα

Με ένα απλό script MATLAB θα εξετάσουμε διαισθητικά τα χαρακτηριστικά του condition number ενός τυχαίου ελαχίστου.

```
rng(0);
N = 4;
A = rand(N, N);

Q = A*A';
fprintf('Eigenvalues of Q = A*A.T:\n')
eig(Q)

fprintf('N = 4  -> Condition Number of Q:\n')
condQ = max(eig(Q))/min(eig(Q))

N = 20;
A = rand(N, N);

Q = A*A';
fprintf('N = 20 -> Condition Number of Q:\n')
condQ = max(eig(Q))/min(eig(Q))
```

Από το αποτέλεσμα παρατηρούμε ότι για τυχαίο πίνακα, το condition number εκτινάσσεται καθώς αυξάνουμε τις διαστάσεις:

```
Eigenvalues of Q = A*A.T:

ans =

 0.0009
 0.1441
 0.7379
 6.8648

N = 4  -> Condition Number of Q:

condQ =

 7.3312e+03

N = 20 -> Condition Number of Q:

condQ =

 2.1871e+05
```

Παρατήρηση: Φαίνεται ξεκάθαρα πλέον η σημασία της χρήσης της ορμής στα προβλήματα βελτιστοποίησης.

4 Gradient Descend σε Κυρτή - μη Παραγωγίσιμη Συνάρτηση

Έστω η συνάρτηση $f(x_1, x_2) = \max(x_1 + x_2, 0.9x_1 - 1.1x_2 + 1, -0.8x_1 + 1.2x_2 - 1, 2 - 1.1x_1 - 0.9x_2)$. Τη σχεδιάζουμε:

```
% Define the function
f = @(x1, x2) max(x1 + x2, max(0.9*x1 - 1.1*x2 + 1, max(-0.8*x1 + 1.2*x2 - 1, 2
    - 1.1*x1 - 0.9*x2)));

% Create a grid of points
[x1, x2] = meshgrid(linspace(-5, 5, 50), linspace(-5, 5, 50));

% Evaluate the function at each point
z = f(x1, x2);

% Plot the surface
figure;
surf(x1, x2, z);
title('Surface Plot of f(x_1, x_2)');
xlabel('x_1');
ylabel('x_2');
zlabel('f(x_1, x_2)');
```

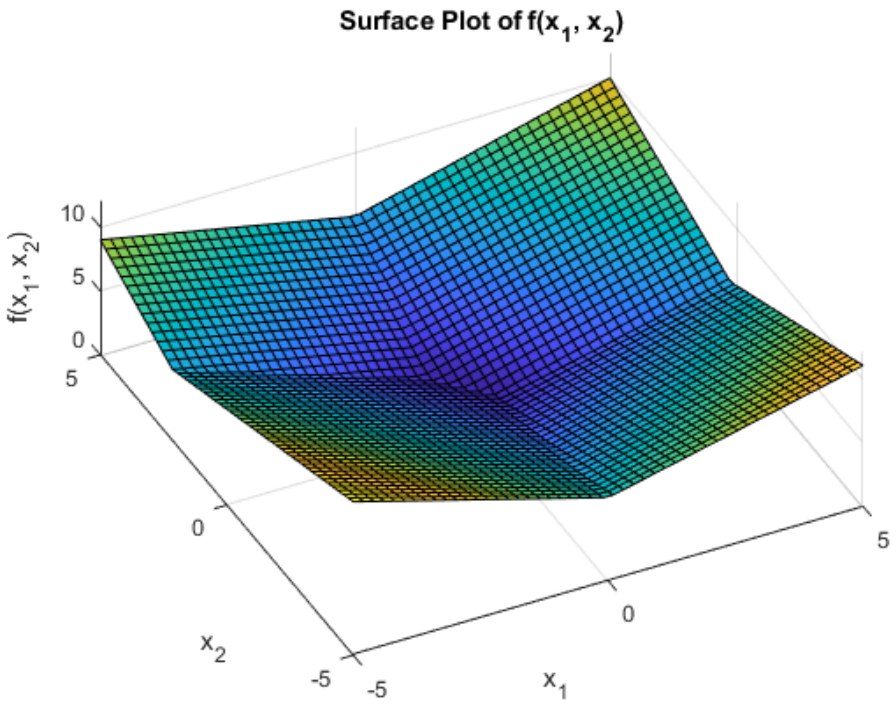


Figure 12: Function Surface

Με τον ακόλουθο κώδικα, εκτελούμε Gradient Descend από ένα αυθαίρετο σημείο και προβάλουμε την πορεία των εκτιμήσεων ως το ελάχιστο:

```
% Define the function
f = @(x1, x2) max(x1 + x2, max(0.9*x1 - 1.1*x2 + 1, max(-0.8*x1 + 1.2*x2 - 1, 2
    - 1.1*x1 - 0.9*x2)));
```

```
3
4  % Create a grid of points
5  [x1, x2] = meshgrid(linspace(-5, 5, 50), linspace(-5, 5, 50));
6
7  % Evaluate the function at each point
8  z = f(x1, x2);
9
10 % Plot the surface
11 figure;
12 surf(x1, x2, z);
13 title('Surface Plot of f(x_1, x_2)');
14 xlabel('x_1');
15 ylabel('x_2');
16 zlabel('f(x_1, x_2)');
17
18 % grad_f = @(x1, x2) [1 - 0.9 * (x2 < x1 + x2) - 0.8 * (x2 < 1.1 * x1 - 1), 1 -
      1.1 * (x1 + x2 < 0) + 1.2 * (1.1 * x1 - 1 < x2) - 0.9 * (2 - 1.1*x1 - 0.9*x2
      < x2)];
19
20 % Gradient Descent parameters
21 learning_rate = 0.1;
22 iterations = 50;
23
24 % Initial point
25 x = [5; 3];
26 % Initialize points matrix
27 points = zeros(2, iterations + 1);
28 points(:, 1) = x;
29
30 % Gradient Descent
31 for i = 1:iterations
32   gradient = grad_f(x(1), x(2));
33   x = x - learning_rate * gradient;
34   points(:, i + 1) = x;
35   if f(x(1), x(2)) == 0
36     break;
37   end
38 end
39
40 % Plot the function
41 figure;
42 surf(x1, x2, z);
43 hold on;
44
45 % Plot the gradient descent path
46 plot3(points(1, :), points(2, :), f(points(1, :), points(2, :)), '-o', 'Color',
      'r', 'LineWidth', 2);
47
48 title('Gradient Descent on f(x_1, x_2)');
49 xlabel('x_1');
50 ylabel('x_2');
51 zlabel('f(x_1, x_2)');
```

```
legend('Function', 'Gradient Descent Path');

function gradient = grad_f(x1, x2)
  gradient = zeros(2, 1);

    if x1 + x2 >= max([0.9 * x1 - 1.1 * x2 + 1, -0.8 * x1 + 1.2 * x2 - 1, 2 -
   1.1 * x1 - 0.9 * x2])
    gradient(1) = 1;
    gradient(2) = 1;
  elseif 0.9 * x1 - 1.1 * x2 + 1 >= max([x1 + x2, -0.8 * x1 + 1.2 * x2 - 1, 2 -
   1.1 * x1 - 0.9 * x2])
    gradient(1) = 0.9;
    gradient(2) = -1.1;
  elseif -0.8 * x1 + 1.2 * x2 - 1 >= max([x1 + x2, 0.9 * x1 - 1.1 * x2 + 1, 2 -
   1.1 * x1 - 0.9 * x2])
    gradient(1) = -0.8;
    gradient(2) = 1.2;
  else
    gradient(1) = -1.1;
    gradient(2) = -0.9;
  end
end
```

Και έχουμε

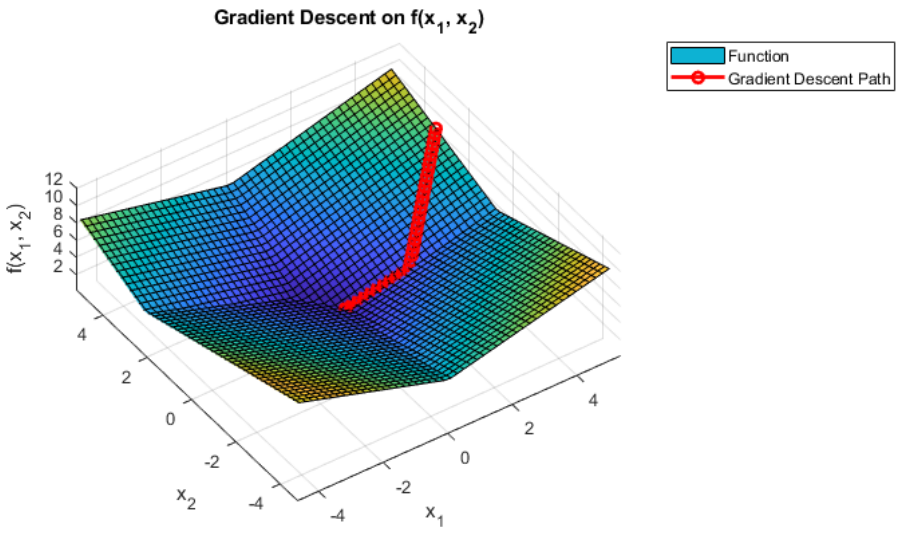


Figure 13: Gradient Descend Visualization

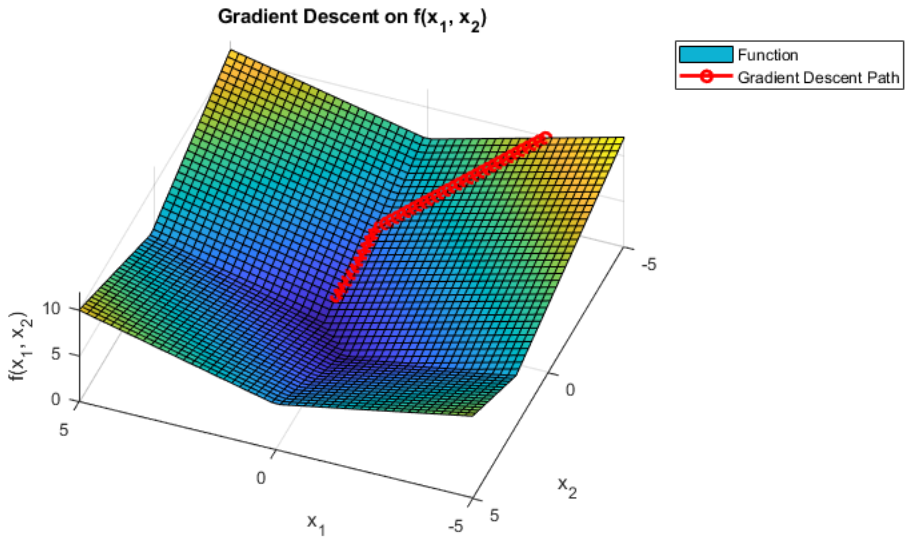


Figure 14: Gradient Descend Visualization from different initial conditions

4.1 Stochastic Gradient Descend

Θεωρούμε τη συνάρτηση

$$f(x_1, x_2) = \frac{1}{4}\sum_{n=1}^{4}[(x_1 - a_i)^2 + (x_2 - b_i)^2]$$

Τη σχεδιάζουμε:

```
% Define the function
f = @(x1, x2, a, b) 1/4 * ((x1 - a(1)).^2 + (x2 - b(1)).^2 + (x1 - a(2)).^2 + (
    x2 - b(2)).^2) + (x1 - a(3)).^2 + (x2 - b(3)).^2 + (x1 - a(4)).^2 + (x2 - b
    (4)).^2;

% Create a grid of points
[x1, x2] = meshgrid(linspace(-5, 5, 50), linspace(-5, 5, 50));

% Evaluate the function at each point
z = f(x1, x2, [1; 2; 3; 4], [2; -1; 3; 0]);

% Plot the surface
figure;
surf(x1, x2, z);
title('Surface Plot of f(x_1, x_2)');
xlabel('x_1');
ylabel('x_2');
zlabel('f(x_1, x_2)');
```

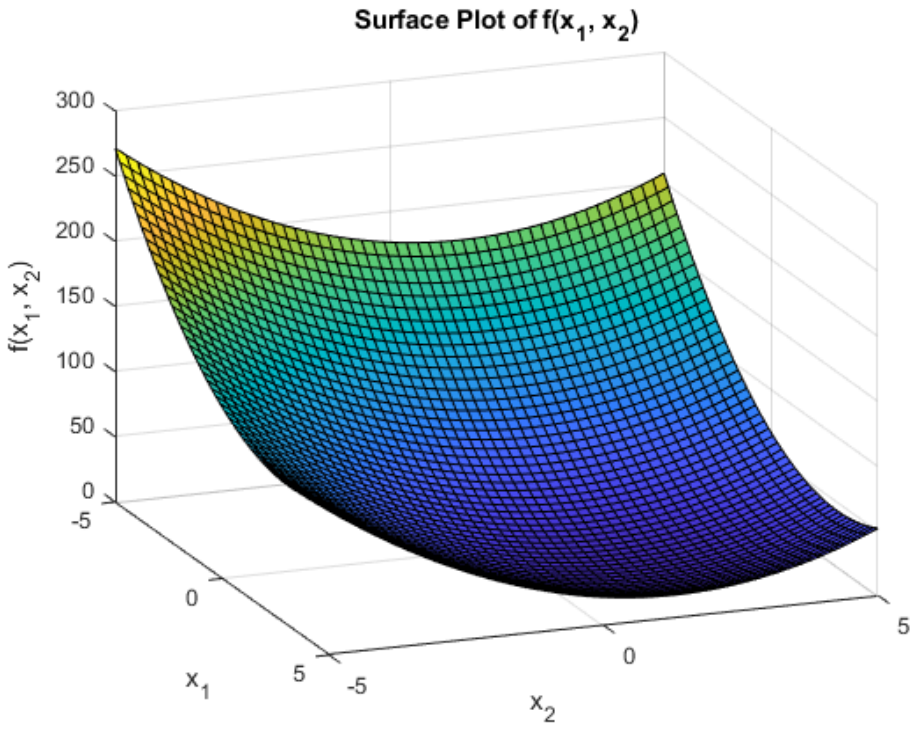


Figure 15: Function Surface

Θα εφαρμόσουμε Stochastic Gradient Descend όπου σε κάθε βήμα θα θεωρούμε γνωστό μόνο ένα από τα 4 gradients $d_i = \frac{1}{2}[(x_1 - a_i) + (x_2 - b_i)]^T$

```
% Gradient Descent parameters
learning_rate = 0.1;
iterations = 50;

% Parameters for the function
a = [1; 2; 3; 4];
b = [2; -1; 3; 0];

% Initial point
x = [-3; -5];
% Initialize points matrix
points = zeros(2, iterations + 1);
points(:, 1) = x;

% Stochastic Gradient Descent
for i = 1:iterations
% Choose a random index
idx = randi(4);

% Compute stochastic gradient
gradient = 0.5 * [(x(1) - a(idx)); (x(2) - b(idx))];

% Update x
x = x - learning_rate * gradient;

% Save the point
```

```
points(:, i + 1) = x;
end

% Plot the function
figure;
surf(x1, x2, z);
hold on;

% Plot the gradient descent path
plot3(points(1, :), points(2, :), f(points(1, :), points(2, :), a, b), '-o', '
    Color', 'r', 'LineWidth', 2);

title('Gradient Descent on f(x_1, x_2)');
xlabel('x_1');
ylabel('x_2');
zlabel('f(x_1, x_2)');
legend('Function', 'Gradient Descent Path');
```

Και βλέπουμε την πορεία των εκτιμήσεων. Παρατηρούμε ότι ο αλγόριθμος είναι robust, αφού ακόμα και με άγνωστα τα 3 από τα 4 gradients κάθε φορά, συγκλίνει:

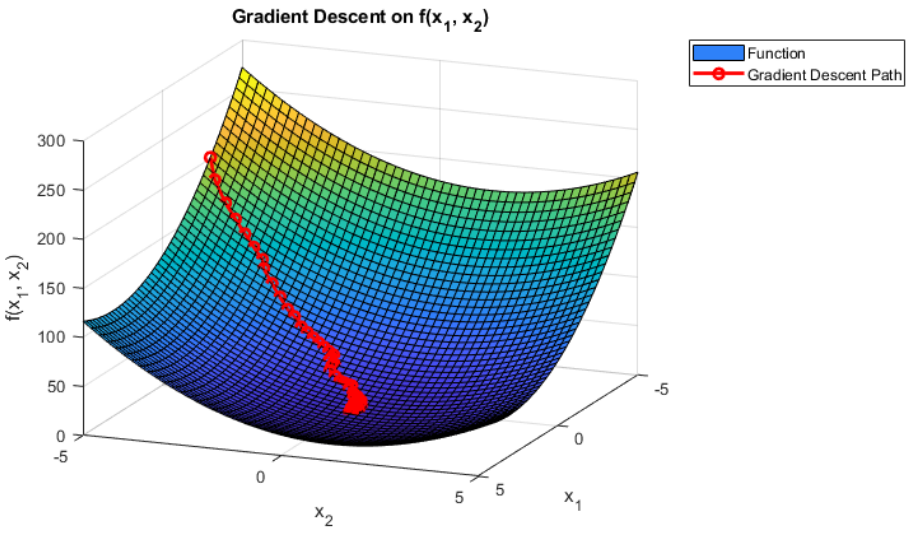


Figure 16: Gradient Descend Visualization

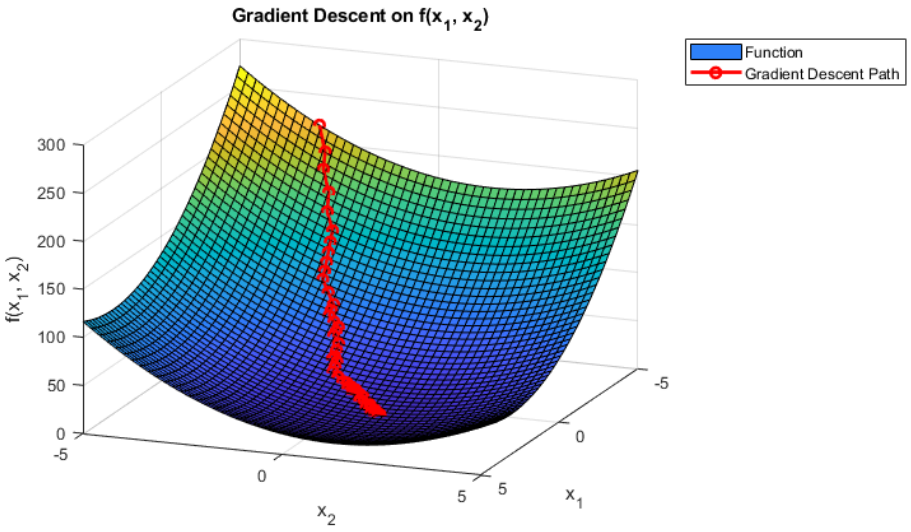


Figure 17: Gradient Descend Visualization from different initial conditions

5 Προσέγγιση Συνάρτησης με Απλό Νευρωνικό Δίκτυο

Θα χρησιμοποιήσουμε τη βιβλιοθήκη keras για να εκπαιδεύσουμε ένα απλό νευρωνικό δίκτυο με ένα κρυφό στρώμα που θα προσεγγίζει τη συνάρτηση

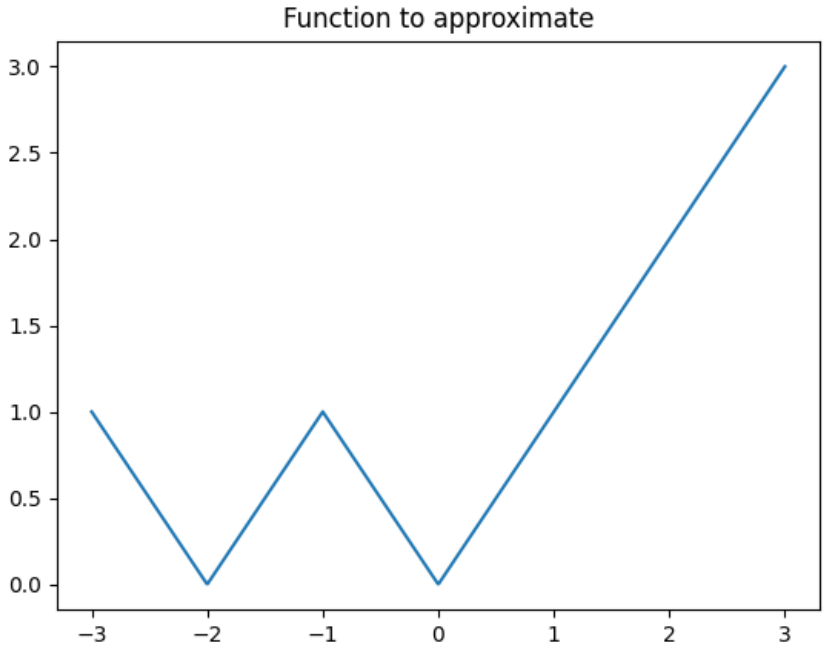


Figure 18: Function To Be Approximated

Κάνουμε import τις απαραίτητες βιβλιοθήκες και ορίζουμε τη συνάρτηση που θα προσεγγίσουμε:

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from numpy import asarray
from matplotlib import pyplot
import numpy as np
from keras.optimizers import Adam

# Define the piecewise function
def piecewise_function(x):
    if x <= -2:
        return -x - 2
    elif -2 < x <= -1:
        return x + 2
    elif -1 < x <= 0:
        return -x
    else:
        return x
```

Δημιουργούμε dataset 1000 σημείων σε διάστημα $[-3, 3]$:

```
# Create the dataset
x = asarray(np.linspace(-3, 3, 1000))
y = asarray([piecewise_function(i) for i in x])
print(x.min(), x.max(), y.min(), y.max())

# Reshape arrays into rows and columns
x = x.reshape((len(x), 1))
y = y.reshape((len(y), 1))

pyplot.plot(x, y)
pyplot.title('Function to approximate')
pyplot.show()
```

Μια καλή πρακτική είναι να κάνουμε scaling στα δεδομένα μας. Βοηθά στην ταχύτερη σύγκλιση του μονέλου. Παρόλο που το dataset μας είναι πολύ απλό και δεν είναι απαραίτητο, κάνουμε scaling:

```
# Separate scaling for the input and output variables
scale_x = MinMaxScaler()
x = scale_x.fit_transform(x)
scale_y = MinMaxScaler()
y = scale_y.fit_transform(y)
print(x.min(), x.max(), y.min(), y.max())
```

Ορίζουμε τώρα το μοντέλο και το εκπαιδεύουμε. Αυθαίρετα επιλέγουμε το κρυφό στρώμα να έχει αρχικά λίγους νευρώνες (10). Όσο περισσότερους νευρώνες χρησιμοποιούμε, τόσο πιο σύνθετες συναρτήσεις μπορούμε να περιγράψουμε. Βέβαια, σε περίπτωση που το dataset έχει θόρυβο, η μεγάλη πολυπλοκότητα του μοντέλου θα οδηγήσει σε overfitting, δηλαδή το μοντέλο είναι τόσο σύνθετο που έχει τη δυνατότητα να μάθει το θόρυβο και θα το κάνει (επειδή τον θεωρεί μέρος της συνάρτησης που επιδιώκει να περιγράψει - δεν μπορεί αλλιώς, άλλωστε). Το αποτέλεσμα θα είναι να έχει κακή γενίκευση. Στην περίπτωσή μας όμως, έχουμε ένα τέλειο dataset. Χρησιμοποιούμε MSE Loss για criterion και Adam optimizer με learning rate 0.001. Ακόμα, επιλέγουμε activation function τη ReLU.

```
# Design the neural network model
model = Sequential()
model.add(Dense(100, input_dim=1, activation='relu', kernel_initializer='
    he_uniform'))
model.add(Dense(1))

# Define the loss function and optimization algorithm
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))

# Fit the model on the training dataset
history = model.fit(x, y, epochs=500, batch_size=10, verbose=1)
```

Αφού εκπαιδεύσουμε το μοντέλο, κάνουμε προβλέψεις για τα σημεία $[-3, 3]$:

```
# Make predictions for the input data
yhat = model.predict(x)

# Inverse transforms
x_plot = scale_x.inverse_transform(x)
```

```
y_plot = scale_y.inverse_transform(y)
yhat_plot = scale_y.inverse_transform(yhat)

# Report model error
print('MSE: %.3f' % mean_squared_error(y_plot, yhat_plot))
```

Τέλος, σχεδιάζουμε τα αποτελέσματα:

```
# Plot training loss
pyplot.plot(history.history['loss'])
pyplot.title('Model Training Loss')
pyplot.xlabel('Epoch')
pyplot.ylabel('Loss')
pyplot.show()

# Plot x vs y
pyplot.scatter(x_plot, y_plot, label='Actual')

# Plot x vs yhat
pyplot.scatter(x_plot, yhat_plot, label='Predicted', s=2)

pyplot.title('Model Approximation')
pyplot.xlabel('Input Variable (x)')
pyplot.ylabel('Output Variable (y)')
pyplot.legend()
pyplot.show()
```

Για 10 νευρώνες στο κρυφό στρώμα έχουμε:

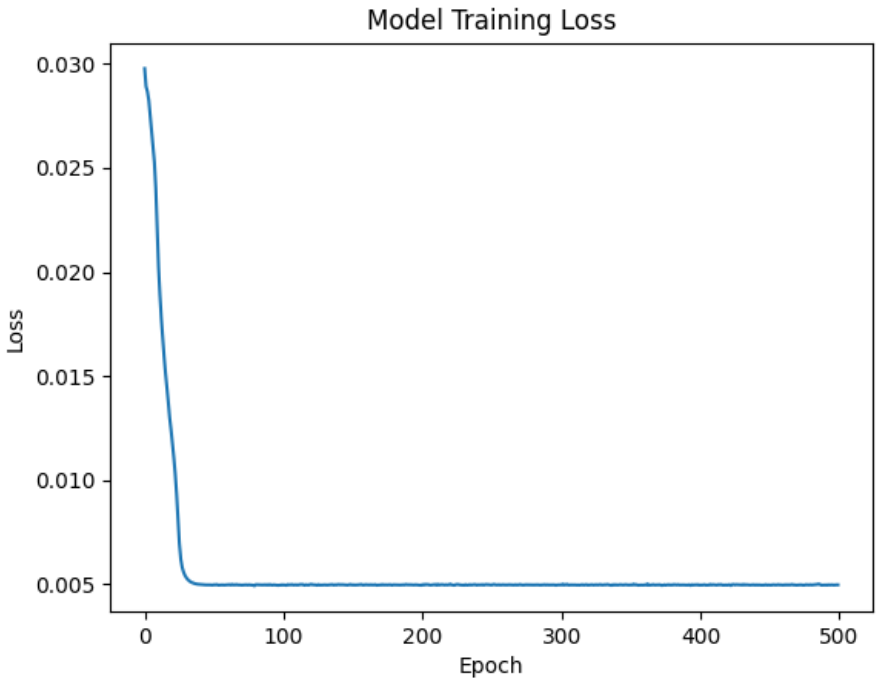


Figure 19: Training Loss for 10 Hidden Neurons

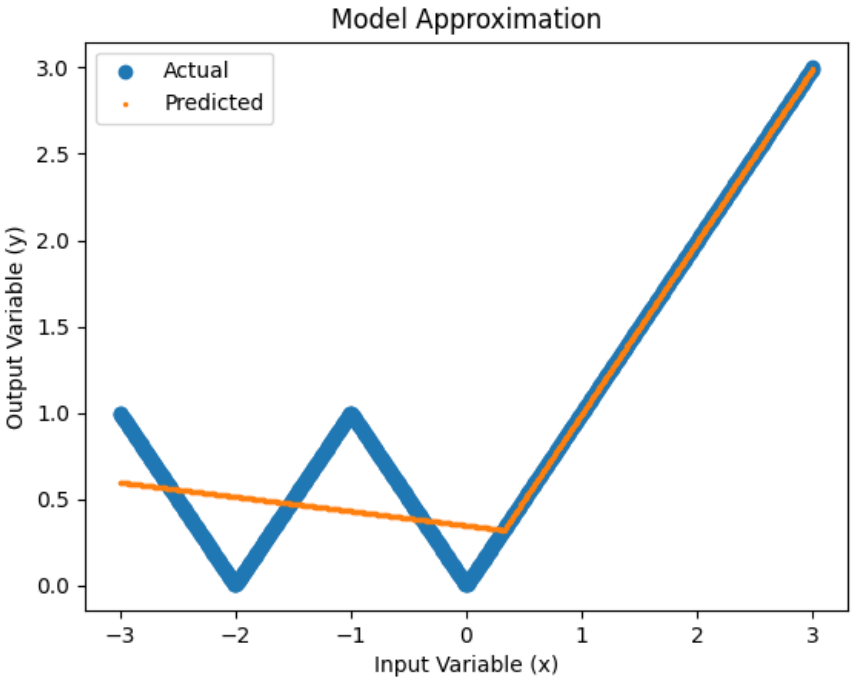


Figure 20: Approximation for 10 Hidden Neurons

Βλέπουμε ότι το νευρωνικό δίκτυο δεν καταφέρνει να μάθει το dataset. Δοκιμάζουμε με 100 νευρώνες. Βλέπουμε το log της εκπαίδευσης εδώ. Παρατηρούμε ότι το MSE σταδιακά πέφτει στο 0.

```
Epoch 1/500
100/100 [==============================] - 0s 706us/step - loss: 0.3501
Epoch 2/500
100/100 [==============================] - 0s 622us/step - loss: 0.0327
Epoch 3/500
100/100 [==============================] - 0s 589us/step - loss: 0.0302
Epoch 4/500
100/100 [==============================] - 0s 559us/step - loss: 0.0289
Epoch 5/500
100/100 [==============================] - 0s 605us/step - loss: 0.0284
Epoch 6/500
100/100 [==============================] - 0s 606us/step - loss: 0.0280
Epoch 7/500
100/100 [==============================] - 0s 606us/step - loss: 0.0282
Epoch 8/500
100/100 [==============================] - 0s 569us/step - loss: 0.0278
Epoch 9/500
100/100 [==============================] - 0s 608us/step - loss: 0.0273
Epoch 10/500
100/100 [==============================] - 0s 606us/step - loss: 0.0274
...
Epoch 491/500
100/100 [==============================] - 0s 522us/step - loss: 6.5585e-08
Epoch 492/500
```

```
100/100 [==============================] - 0s 607us/step - loss: 6.4020e-07
Epoch 493/500
100/100 [==============================] - 0s 643us/step - loss: 9.8022e-07
Epoch 494/500
100/100 [==============================] - 0s 648us/step - loss: 1.7729e-05
Epoch 495/500
100/100 [==============================] - 0s 524us/step - loss: 1.1735e-04
Epoch 496/500
100/100 [==============================] - 0s 609us/step - loss: 7.9281e-08
Epoch 497/500
100/100 [==============================] - 0s 615us/step - loss: 1.4296e-08
Epoch 498/500
100/100 [==============================] - 0s 615us/step - loss: 1.6921e-08
Epoch 499/500
100/100 [==============================] - 0s 598us/step - loss: 1.5826e-08
Epoch 500/500
100/100 [==============================] - 0s 584us/step - loss: 1.9087e-08
32/32 [==============================] - 0s 635us/step
MSE: 0.000
```

Και έχουμε:

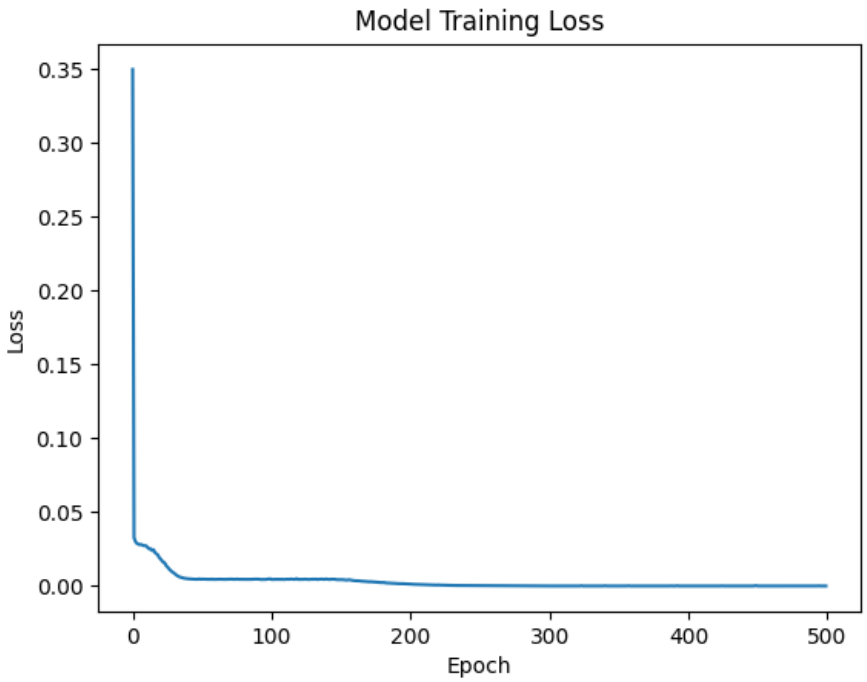


Figure 21: Training Loss for 10 Hidden Neurons

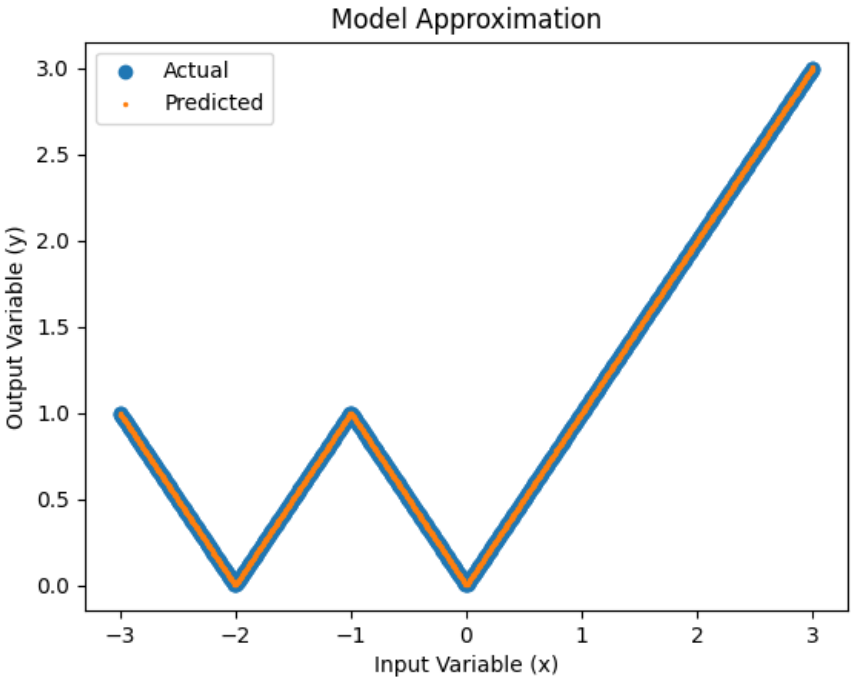


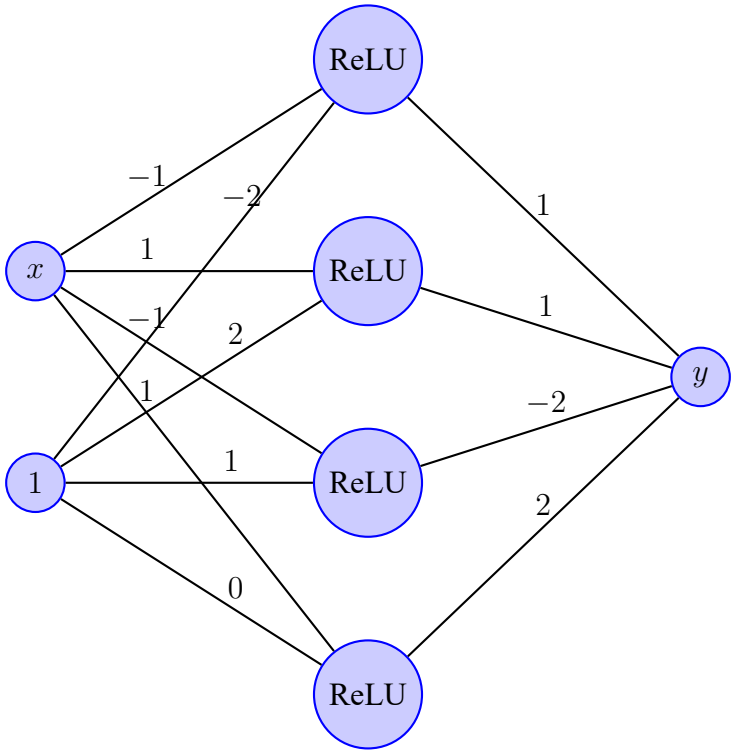
Figure 22: Approximation for 100 Hidden Neurons

Το νευρωνικό δίκτυο πλέον προσεγγίζει τέλεια τη συνάρτηση!

- Μετά από αυτήν την ανάλυση συνειδητοποίησα ότι δεν έπρεπε να φτιάξουμε το δίκτυο με κώδικα -
Πιο θεωρητικά, λοιπόν, η συνάρτηση που θέλουμε να προσεγγίσουμε αποτελείται από 4 κλάδους, όπου κάθε ένας είναι μια γραμμική συνάρτηση. Μπορούμε να γράψουμε

$$f(x) = \max(-x-2,0) + \max(x+2,0) - 2\max(x+1,0) + 2\max(x,0) =$$
$$= \begin{cases} (-x-2)+(0)+(0)+(0) = -x-2, x \leq -2 \\ (0)+(x+2)+(0)+(0) = x+2, -2 < x \leq -1 \\ (0)+(x+2)-2(x+1)+(0) = -x, -1 < x \leq 0 \\ (0)+(x+2)-2(x+1)+2x = x, 0 < x \end{cases}$$

Από τα παραπάνω, έχουμε το εξής νευρωνικό δίκτυο:


ReLU
ReLU
ReLU
ReLU
x
1
y
−1
−2
1
1
−1
2
1
−2
1
2
0

6 Εξοικείωση με τα Νευρωνικά Δίκτυα - Επεξεργασία Notebook

Το notebook είναι ένα tutorial σε TensorFlow για την ταξινόμηση εικόνων με χρήση νευρωνικών δικτύων. Το dataset που χρησιμοποιείται είναι το Fashion MNIST, που περιέχει 70.000 grayscale εικόνες, κατηγοριοποιημένες σε 10 κλάσεις.
Γίνεται μια προεπεξεργασία των δεδομένων μετασχηματίζοντας τις τιμές των pixels σε εύρος $[0, 1]$. Το μοντέλο που δοκιμάζεται έχει δύο στρώματα, χρησιμοποιεί optimizer Adam, activation function ReLU και κριτήριο crossentropy loss. Στο τελευταίο layer έχουμε ένα softmax layer. Αυτό το επίπεδο μετατρέπει τις προβλέψεις (logits) σε πιθανότητες μέσω της συνάρτησης softmax:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Έτσι, το μοντέλο μαθαίνει να παράγει μια κατανομή πιθανότητας πάνω στα labels και όχι να εκτιμά ένα one-hot διάνυσμα που είναι μια πολύ πιο γενική διαδικασία (άρα το accuracy είναι υψηλότερο έτσι). Ένα παράδειγμα όπου η σωστή πρόβλεψη είναι η κλάση 4 φαίνεται εδώ. Βλέπουμε ότι έχει αποδοθεί μεγαλύτερη πιθανότητα στην 4η θέση:

$$\mathbf{z} = \begin{bmatrix} -2.5 \\ 1.0 \\ 0.5 \\ 3.0 \\ 2.0 \\ -1.0 \\ 0.2 \\ -0.5 \\ 1.5 \\ -1.2 \end{bmatrix} \Rightarrow \text{Softmax}(\mathbf{z}) = \begin{bmatrix} \frac{e^{-2.5}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{1.0}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{0.5}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{3.0}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{2.0}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{-1.0}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{0.2}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{-0.5}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{1.5}}{\sum_{i=1}^{10} e^{z_i}} \\ \frac{e^{-1.2}}{\sum_{i=1}^{10} e^{z_i}} \end{bmatrix} = \begin{bmatrix} 0.002 \\ 0.070 \\ 0.042 \\ 0.516 \\ 0.190 \\ 0.009 \\ 0.031 \\ 0.016 \\ 0.115 \\ 0.008 \end{bmatrix}$$

Και βλέπουμε πως αυτό το παράδειγμα οδηγεί στην εκτίμηση μιας κατανομής πιθανότητας πάνω στις κλάσεις:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import make_interp_spline

# Given probabilities
probabilities = np.array([0.00211003, 0.06987444, 0.04238099, 0.51630616,
    0.18993842, 0.00945648, 0.03139661, 0.0155911, 0.11520348,
   0.00774231])

```

```
# Create a vector for x-axis (assuming indices as x values)
x_values = np.arange(len(probabilities))

# Plot the bar graph
plt.bar(x_values, probabilities, color='blue', alpha=0.7, label='
    Probabilities')

# Smooth curve using spline interpolation
x_smooth = np.linspace(x_values.min(), x_values.max(), 100)
spl = make_interp_spline(x_values, probabilities, k=2)
smooth_curve = spl(x_smooth)

# Plot the smooth curve
plt.plot(x_smooth, smooth_curve, color='red', linestyle='-', linewidth=2,
     label='Smooth Curve')

# Customize the plot
plt.xlabel('Class Index')
plt.ylabel('Probability')
plt.title('Probability Distribution with Smooth Curve')
plt.legend()

# Show the plot
plt.show()
```

Και έχουμε το ιστόγραμμα με τις πιθανότητες και μια προσέγγιση με σπλήνες για να μοιάζει συνεχής η κατανομή:

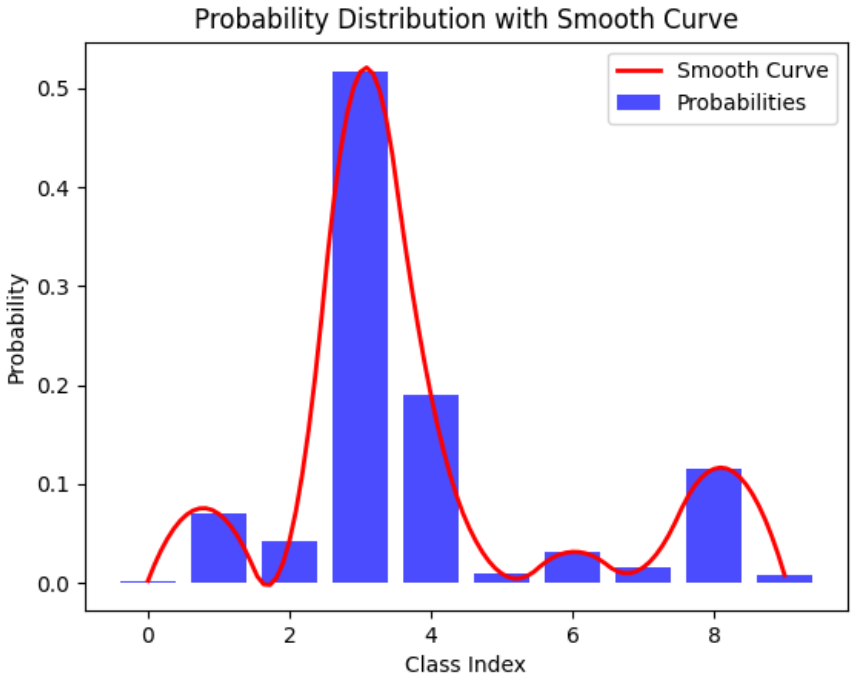


Figure 23: Probability Distribution

Γίνεται η εκπαίδευση του μοντέλου προσαρμόζοντας τα βάρη του στο train dataset. Ύστερα, η αξιολόγηση

συμβαίνει στο test dataset και είναι μια καλή εικόνα της πραγματικής απόδοσης του μοντέλου αφού δοκιμάζεται σε δεδομένα που δεν έχει δει ποτέ. Έτσι μπορούμε να καταλάβουμε κατά πόσο το μοντέλο έμαθε τα γενικά χαρακτηριστικά του dataset και μπορεί να γενικεύσει σε unseen data ή απομνημόνευσε το train dataset και δε γενικεύει (μεγάλη διαφορά train - test accuracy συνεπάγεται overfitting). Σε αυτό το σημείο θα εξετάσουμε πως η πολυπλοκότητα του μοντέλου επηρρεάζει την ικανότητα γενίκευσης.

Παρουσιάζουμε μερικές δοκιμές:

```
====================================
1 hidden layer - 2 neurons:
TRAIN - loss: 0.7578 - accuracy: 0.7328
TEST  - loss: 0.7704 - accuracy: 0.7314
====================================
1 hidden layer - 4 neurons:
TRAIN - loss: 0.5130 - accuracy: 0.8207
TEST  - loss: 0.5601 - accuracy: 0.8050
====================================
1 hidden layer - 16 neurons:
TRAIN - loss: 0.3506 - accuracy: 0.8771
TEST  - loss: 0.4167 - accuracy: 0.8544
====================================
1 hidden layer - 128 neurons:
TRAIN - loss: 0.2366 - accuracy: 0.9116
TEST  - loss: 0.3428 - accuracy: 0.8793
====================================
1 hidden layer - 1024 neurons:
TRAIN - loss: 0.2213 - accuracy: 0.9164
TEST  - loss: 0.3481 - accuracy: 0.8778
====================================
10 hidden layers - 128 neurons each:
TRAIN - loss: 0.2853 - accuracy: 0.8970
TEST  - loss: 0.3569 - accuracy: 0.8794
====================================
```

Παρατηρούμε:

- Ακόμα και με λίγους νευρώνες (π.χ. 16), πετυχαίνουμε αρκετά υψηλό accuracy. Αυτό σημαίνει ότι το dataset είναι απλό.
- Με πολύ λίγους νευρώνες (π.χ. 2 ή 4), το μοντέλο είναι πολύ απλό και δεν μπορεί να μάθει έστω και λίγο σύνθετες δομές, δηλαδή το accuracy μειώνεται σημαντικά.
- Με πάρα πολλούς νευρώνες (π.χ. 1024), το μοντέλο είναι πολύ σύνθετο και όπως εξηγήσαμε, απομνημονεύει το train dataset (accuracy 0.92) και δυσκολεύεται να γενικεύσει σε unseen data του test dataset (accuracy 0.88). Ακόμα βέβαια δε φαίνεται η επίδραση του overfitting, διότι και το dataset είναι πολύ απλό και μεγάλο (απαιτούνται πάρα πολλοί νευρώνες για να απομνημονεύσει το μοντέλο 60000 εικόνες)
- Ακόμα δοκιμάσαμε πολλά κρυφά επίπεδα (10), δηλαδή υψηλή πολυπλοκότητα όπως με τους 1024 νευρώνες. Πάλι παρατηρήσαμε υψηλότερο train accuracy από test accuracy που είναι ένδειξη overfitting, όμως για μια ακόμα φορά, η απλότητα του dataset και το μεγάλο μέγεθός του δεν επιτρέπουν να φανεί καλά η επίδραση της μεγάλης πολυπλοκότητας στην ικανότητα γενίκευσης του μοντέλου