Machine Learning (CS696)

Fall 2017

Assignment 4

Submitted by

Ashok Kumar Shrestha

Assignment 4

In this assignment you will design and train a backpropagation network to learn the exclusive-OR function.

1. Use one hidden layer with 4 hidden units. Plot the error as a function of the number of iterations. Remember the initial weights.

Answer:

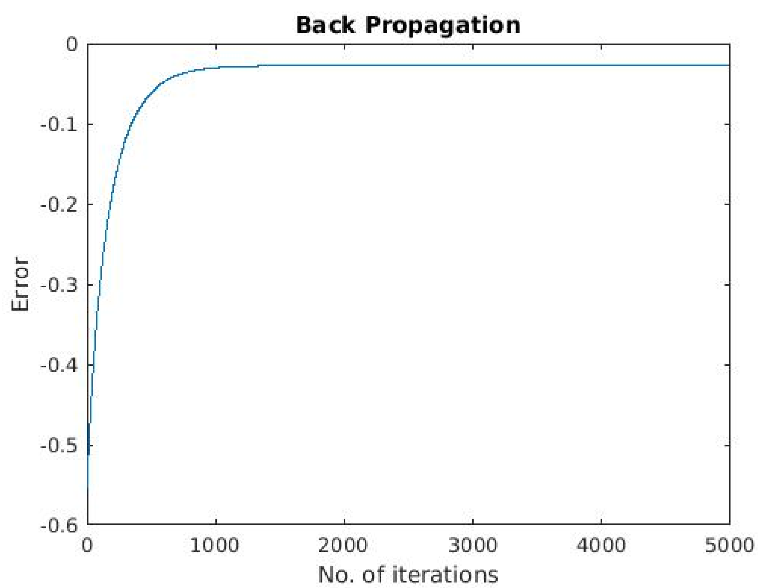
Initial values:

w = [0.4047 0.1099 0.1177 0.3594]

v = [0.3909 0.1987 -0.4695 0.0000
-0.1658 -0.3022 0.2441 -0.0201]

x0 = 0.2500
z0 = 0.2500
alpha = 0.01

Here, with 4 hidden units and one hidden layer convergence happens around 1400 iterations as shown in the figure.



Source Code: Code in MatLab.

```
%Assignment 4.a: Back propagation Algorithm

clear;
clc;

main();

%Squashing function
function sq_val = squashing(z)
   sq_val = 1./(1+exp(-z));
```

```
end

function main()
    %Initial variables
    x = [0 0; 0 1; 1 0; 1 1]; %Input
    t = [0, 1, 1, 0];   %Ouput XOR
    n = size(x);
    m = length(t);
    alpha = 0.01;  %Learning rate
    iterations = 5000;   %no. of iterations

    x0 = 0.25;  %Input Bias
    z0 = 0.25;  %Hidden layer Bias

    r0 = -0.5;  %random variable lower limit
    r1 = 0.5;   %random variable upper limit

    v = (r1 - r0) * rand(2,4) + r0;
    w = (r1 - r0) * rand(4,1) + r0;

    %Iterating the operations

    for i = 1:iterations

        z_in = sum(x * v) + x0;
        z = squashing(z_in);

        y_in = sum(z .* w) + z0;
        y = squashing(y_in);
        error = t - y;

        f_dash = y' * (1 - y);
        delta = error * f_dash;    %weight correction term
        delta_weight = alpha * delta' * z;
        delta_weight_bias = alpha * delta;
        w = w + delta_weight;
        z0 = z0 + delta_weight_bias;
        err_graph(i) = sum(error);

        f_dash_output = z * (1 - z)';
        delta_output = sum(delta * w) * f_dash_output;
        delta_v = alpha * delta_output * x;
        delta_v_bias = alpha * delta_output;
        v = v + delta_v';
        x0 = x0 + delta_v_bias;
    end

    plot(0:iterations-1, err_graph(1:iterations));
    title('Back Propagation');
```

```
    ylabel('Error');
    xlabel('No. of iterations');

end
```

2. Verify that momentum term indeed improves convergence (μ = 0.5). Use the initial weights remembered in (1).

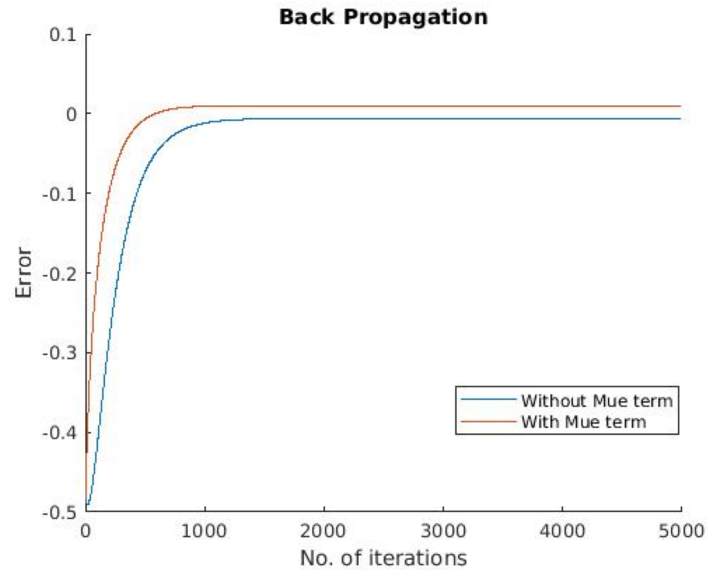
Answer:
Initial values:

w = [0.4047 0.1099 0.1177 0.3594]

v = [0.3909 0.1987 -0.4695 0.0000
-0.1658 -0.3022 0.2441 -0.0201]

x0 = 0.2500
z0 = 0.2500
μ = 0.5
alpha = 0.01

For same initial values, without momentum term, convergence happens around 1400 iterations whereas, with the momentum term, convergence occurs around 800 iterations as shown in figure below. This shows that momentum term indeed improves convergences.



Source Code: Code in MatLab.

```
%Assignment 4 a vs b: Back propagation Algorithm

clear;
clc;

main();

%Squashing function
function sq_val = squashing(z)
    sq_val = 1./(1+exp(-z));
end


function main()
    %Initial variables
    x = [0 0; 0 1; 1 0; 1 1]; %Input
    t = [0, 1, 1, 0];   %Ouput XOR
    n = size(x);
    m = length(t);
```

```
    alpha = 0.01;  %Learning rate
    iterations = 5000;   %no. of iterations
    mue = 0.5;  %momentum term

    x0 = 0.25;  %Input Bias
    z0 = 0.25;  %Hidden layer Bias

    r0 = -0.5;  %random variable lower limit
    r1 = 0.5;   %random variable upper limit

    v = (r1 - r0) * rand(2,4) + r0;
    w = (r1 - r0) * rand(4,1) + r0;

    old_w = w;
    old_v = v;
    old_x0 = x0;
    old_z0 = z0;

    %Iterating the operations without momentum term

    for i = 1:iterations
            z_in = sum(x * v) + x0;
        z = squashing(z_in);

        y_in = sum(z .* w) + z0;
        y = squashing(y_in);
        error = t - y;

        f_dash = y' * (1 - y);
        delta = error * f_dash;    %weight correction term
        delta_weight = alpha * delta' * z;
        delta_weight_bias = alpha * delta;
        w = w + delta_weight;
        z0 = z0 + delta_weight_bias;
        err_graph(i) = sum(error);

        f_dash_output = z_in .* (1 - z_in);
        delta_output = sum(delta .* w) .* f_dash_output;
        delta_v = alpha * delta_output * x;
        delta_v_bias = alpha * delta_output;
        v = v + delta_v';
        x0 = x0 + delta_v_bias;
end

   %Initializing the old values
    w = old_w;
    v = old_v;
    x0 = old_x0;
    z0 = old_z0;
```

```
    %Iterating the operations with momentum term

    for i = 1:iterations

        z_in = sum(x * v) + x0;
        z = squashing(z_in);

        y_in = sum(z .* w) + z0;
        y = squashing(y_in);
        error = t - y;

        f_dash = y' * (1 - y);
        delta = error * f_dash;    %weight correction term
        delta_weight = alpha * delta' * z;
        delta_weight_bias = alpha * delta;
        momentum_w = mue * (w - old_w);
        old_w = w;
        w = w + delta_weight + momentum_w;

        momentum_z0 = mue * (z0 - old_z0);
        old_z0 = z0;
        z0 = z0 + delta_weight_bias + momentum_z0;
        err_graph1(i) = sum(error);

        f_dash_output = z * (1 - z)';
        delta_output = sum(delta * w) * f_dash_output;
        delta_v = alpha * delta_output * x;
        delta_v_bias = alpha * delta_output;
        momentum_v = mue * (v - old_v);
        old_v = v;
        v = v + delta_v' + momentum_v;

        momentum_x0 = mue * (x0 - old_x0);
        old_x0 = x0;
        x0 = x0 + delta_v_bias + momentum_x0;
    end

    plot(0:iterations-1, err_graph(1:iterations), 0:iterations-1, err_graph1(1:iterations));
    title('Back Propagation');
    ylabel('Error');
    xlabel('No. of iterations');
    legend('Without Mue term','With Mue term')

end
```

3. Repeat (1) using bipolar sigmoid and bipolar representation for exclusive-OR function. The network should converge faster.

Answer:

Initial values:

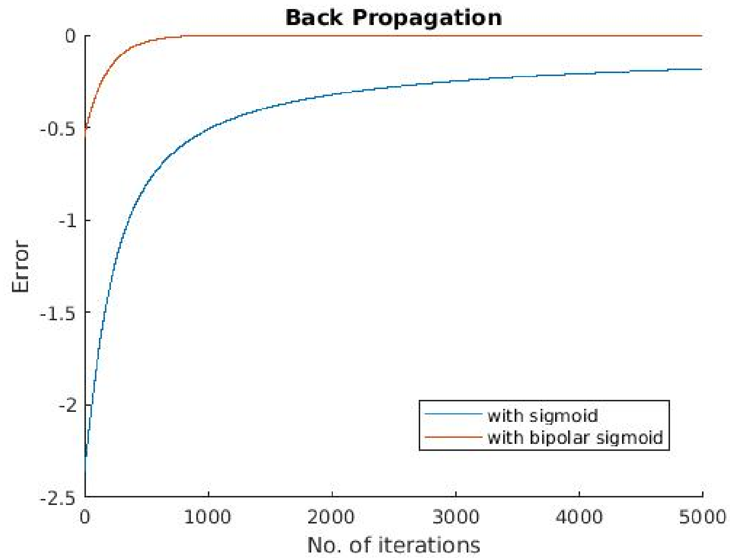
w = [0.1451   0.0523   -0.2819   0.2724]

v = [-0.4739   -0.0694   0.2624   0.1800
0.4547   0.4616   -0.4927   0.2060]

x = [-1 -1; -1 1; 1 -1; 1 1]; %Bipolar Input

t = [-1, 1, 1, -1];   %Bipolar Ouput XOR

x0 =   0.2500
z0 =   0.2500
alpha = 0.005
sigma = 1



For same initial values, with normal sigmoid function, convergence happens around 5000 iterations whereas, with bipolar sigmoid function, convergence occurs around 700 iterations as shown in figure below. This shows that with bipolar sigmoid network converges faster.

Source Code: Code in MatLab.

```
%Assignment 4. c: Back propagation Algorithm

clear;
clc;

main();

%Squashing function
function sq_val = squashing(z)
   sq_val = (1 - exp(-z))./(1 + exp(-z));
end

function main()
   %Initial variables
   x = [-1 -1; -1 1; 1 -1; 1 1]; %Input
   t = [-1, 1, 1, -1];   %Ouput XOR
   n = size(x);
   m = length(t);
   alpha = 0.005;  %Learning rate
   iterations = 5000;   %no. of iterations
   sigma = 1;

   x0 = 0.25;  %Input Bias
```

```
    z0 = 0.25;  %Hidden layer Bias

    r0 = -0.5;  %random variable lower limit
    r1 = 0.5;   %random variable upper limit

    v = (r1 - r0) * rand(2,4) + r0;
    w = (r1 - r0) * rand(4,1) + r0;


    %Iterating the operations

    for i = 1:iterations

        z_in = sum(x * v) + x0;
        z = squashing(z_in);

        y_in = sum(z .* w) + z0;
        y = squashing(y_in);
        error = t - y;

        f_dash = sigma/2 * (1 - y) * (1 + y)';
        delta = error * f_dash;    %weight correction term
        delta_weight = alpha * delta' * z;
        delta_weight_bias = alpha * delta;
        w = w + delta_weight;
        z0 = z0 + delta_weight_bias;
        err_graph(i) = sum(error);

        f_dash_output = sigma/2 * (1 - z) .* (1 + z);
        delta_output = sum(delta * w) * f_dash_output;
        delta_v = alpha * delta_output * x;
        delta_v_bias = alpha * delta_output;
        v = v + delta_v';
        x0 = x0 + delta_v_bias;
    end

    plot(0:iterations-1, err_graph(1:iterations));
    title('Back Propagation');
    ylabel('Error');
    xlabel('No. of iterations');

end
```

4. Verify that Nguyen-Widrow approach of assigning initial weights improves convergence.

Answer:

Initial values:

x0 =   0.2500
z0 =   0.2500
alpha = 0.005

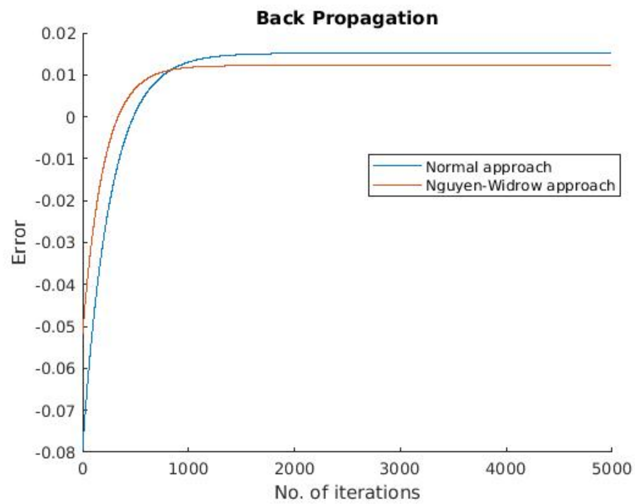
v = [0.4809   0.3008   0.0975   0.4437
    -0.2134   0.3961   0.3840   0.0492]

w = [0.2284   0.0768   -0.4741   -0.0535]

% Nguyen-Widrow approach of assigning initial weights

v = [0.9026   0.5646   0.1831   0.8329
    -0.4005   0.7435   0.7208   0.0923]

x0 =   0.4096



For same initial values, without Nguyen-Widrow approach, convergence happens around 1800 iterations whereas, with the Nguyen-Widrow approach of assigning initial weights, convergence occurs around 1200 iterations as shown in figure below. This shows that Nguyen-Widrow approach of assigning initial weights indeed improves convergences.

<u>Source Code:</u> Code in MatLab.

```
%Assignement 4.d: Backpropagation Algorithm

clear;
clc;

main();

%Squashing function
function sq_val = squashing(z)
    sq_val = 1./(1+exp(-z));
end


function main()
    %Initial variables
    x = [0 0; 0 1; 1 0; 1 1]; %Input
    t = [0, 1, 1, 0];   %Ouput XOR
    n = size(x);
    m = length(t);
    alpha = 0.005;  %Learning rate
    iterations = 5000;   %no. of iterations

    x0 = 0.25;  %Input Bias
    z0 = 0.25;  %Hidden layer Bias

    n = 2;
```

```
    h = 4;  %no. of hidden units

    beta = 0.7 * power(h,1/n);

    r0 = -0.5;  %random variable lower limit
    r1 = 0.5;   %random variable upper limit

    v = (r1 - r0) * rand(2,4) + r0;
    w = (r1 - r0) * rand(4,1) + r0;

    v = beta * v / norm(v);
    x0 = (beta+beta) * rand() - beta;

    %Iterating the operations

    for i = 1:iterations

        z_in = sum(x * v) + x0;
        z = squashing(z_in);

        y_in = sum(z .* w) + z0;
        y = squashing(y_in);
        error = t - y;

        f_dash = y' * (1 - y);
        delta = error * f_dash;    %weight correction term
        delta_weight = alpha * delta' * z;
        delta_weight_bias = alpha * delta;
        w = w + delta_weight;
        z0 = z0 + delta_weight_bias;
        err_graph(i) = sum(error);

        f_dash_output = z * (1 - z)';
        delta_output = sum(delta * w) * f_dash_output;
        delta_v = alpha * delta_output * x;
        delta_v_bias = alpha * delta_output;
        v = v + delta_v';
        x0 = x0 + delta_v_bias;
    end

    plot(0:iterations-1, err_graph(1:iterations));
    title('Back Propagation');
    ylabel('Error');
    xlabel('No. of iterations');

end
```

5. Use two hidden layers (3 units in the first hidden layer and 2 units in the second hidden layer).

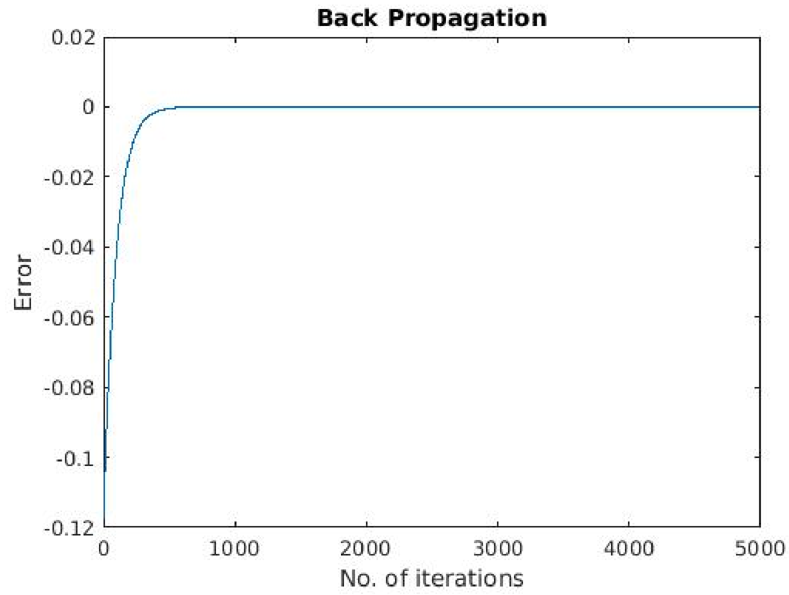
Answer:
Initial values:

x0 = 0.25; %Input Bias
h0 = 0.25; %Hidden layer 1 Bias
z0 = 0.25; %Hidden layer 2 Bias
alpha = 0.01

v = [0.4137   0.0989   0.3997
      0.0583   -0.3511   -0.0496]

h = [-0.2943   0.3825
        0.3997   -0.2150
        0.2626   0.1732]

w = [0.1643   -0.3772]



Here, with two hidden layers, the network converges faster as shown in the figure.

Source Code: Code in MatLab.

```
%Assignement 4.e: Backpropagation Algorithm with 2 hidden layers

clear;
clc;

main();

%Squashing function
function sq_val = squashing(z)
   sq_val = 1./(1+exp(-z));
end


function main()
   %Initial variables
   x = [0 0; 0 1; 1 0; 1 1]; %Input
   t = [0, 1, 1, 0];   %Ouput XOR
   n = size(x);
   m = length(t);
   alpha = 0.01;  %Learning rate
   iterations = 5000;   %no. of iterations

   x0 = 0.25;  %Input Bias
   h0 = 0.25;  %Hidden layer 1 Bias
   z0 = 0.25;  %Hidden layer 2 Bias

   r0 = -0.5;  %random variable lower limit
   r1 = 0.5;   %random variable upper limit
```

```
    v = (r1 - r0) * rand(2,3) + r0;
    h = (r1 - r0) * rand(3,2) + r0;
    w = (r1 - r0) * rand(2,1) + r0;

    %Iterating the operations

    for i = 1:iterations

        z_in = (x * v) + x0;
        z = squashing(z_in);

        h_in = (z * h) + h0;
        hh = squashing(h_in);

        y_in = ((hh * w) + z0)';
        y = squashing(y_in);
        error = t - y;

        f_dash = y' * (1 - y);
        delta = error * f_dash;    %weight correction term
        delta_weight = alpha * delta * hh;
        delta_weight_bias = alpha * delta;
        w = w + delta_weight';
        z0 = z0 + delta_weight_bias';
        err_graph(i) = sum(error);

        f_dash1 = hh * (1 - hh)';
        delta1 = sum(delta .* w) * f_dash1;  %weight correction term hidden layer
        delta_weight1 = alpha * delta1 * z;
        delta_weight_bias1 = alpha * delta1;
        h = h + delta_weight1';
        h0 = h0 + delta_weight_bias1';

        f_dash_output = z * (1 - z)';
        delta_output = (delta1' * sum(h'))' * f_dash_output;
        delta_v = alpha * delta_output * x;
        delta_v_bias = alpha * sum(delta_output);
        v = v + delta_v';
        x0 = x0 + delta_v_bias';
    end

    plot(0:iterations-1, err_graph(1:iterations));
    title('Back Propagation');
    ylabel('Error');
    xlabel('No. of iterations');

end
```