Lab2 Report

Multi-Layer Perceptron

Eduardo Delgado Coloma Bier (s3065979)
Mikel Orbea Sotil (s3075001)

March 3, 2016

1 Theory Questions

a) The credit assignment problem is the process of determining which weights are responsible for the output of the network. In other words, which weights are to blame and need to be changed when the output is not what we want? Or, when things go well, which weights are responsible and shouldn't be changed?

b) The error in a hidden layer is determined by the output of the hidden layer and the goal.

c) The sigmoid function is defined by:

$$\sigma(a) = \frac{1}{1 + exp[\frac{-(a-\theta)}{\rho}]} \tag{1}$$

θ and ρ are constants, while a is the variable given by the activation (summed weights $\times$ input). While θ is responsible for when the curve starts going up (the larger the θ, the longer it takes), ρ is responsible for how smoothly it does so. With a low ρ one can expect a very steep curve. The derivative of the sigmoid function is:

$$\sigma'(a) = \frac{1}{\rho} \cdot \sigma \cdot (1 - \sigma) \tag{2}$$

d) If we initialize the weights with a very high value, chances are our activation will have a high value as well. This means that we would be on the right part of the sigmoid function, where the derivative is close to 0, which is something we don't want because the weights would be adjusted really slowly.

e) One criteria we can use to choose when to stop learning is choosing a minimum error ϵ where we would stop learning if the error is smaller than ϵ. However, choosing the ϵ could be tricky and completely arbitrary. Another possibility would be to stop as soon as the error starts getting bigger. This method, however is not ideal since if we reach a local minimum of the function, we'd probably stop there instead of the actual optimal solution. A third stop criteria would be to keep track of the error and if it" never lower than the last minimal error you found for a certain (big) number of epochs, than you rollback to that minimal error you found. Choosing the amount of epochs, though, can also be tricky and lead to sub optimal solutions.

f) Aside from changing the learning rate, one can speed the learning of a network by choosing appropriate initial weights, neither too high or too low. Both too high and too low values for the weights would lead to σ' values close to zero, which in turn would make every step taken by the network really small.

g) To verify that the network is generalizing for a set of training data we use cross-validation. This method consists on setting aside part of the training set to be used as a validation set after the training. If the error in the validation set starts increasing, then the network is starting to overfit and should be stopped. A problem with this method is that sometimes the error in the validation set only grows temporarily.

h) Overfitting happens when a network trains for too long with a set of data and ends up incorporating too many details from that learning set. When that happens, the network works really well for the learning set, but when different inputs (not in the training set) are given to it, they are not well classified.

i) Network pruning is the deletion of nodes within the network that are considered not important for the network to work. The network is therefore smaller and simpler, resulting in a better performance overall. The basic idea of pruning method is: start with a large enough network, train it with a set of data, determine the importance of the weights, remove the least important weight, retrain the network and repeat that until you get a reasonably sized network. The difference in each pruning method is basically choosing when and which node to cut. One way of doing it is always removing the weight with the lowest value. Another possibility is removing the connection with the smallest contribution (to the network) variance.

2 An MLP on paper

a) Neural network

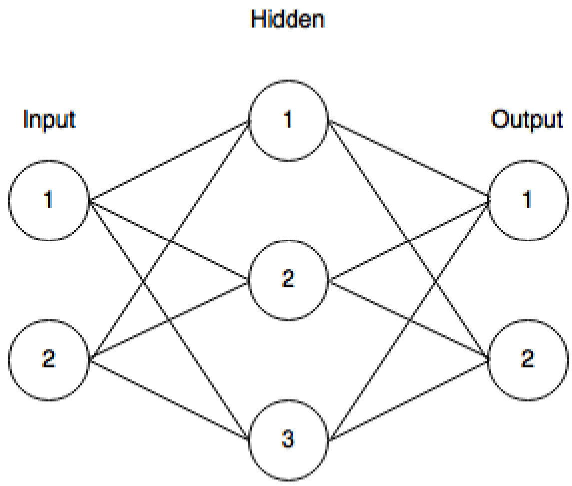

Figure 1: Neural networks with 2 input neurons, 3 hidden neurons and 2 output neurons

b) That is considered a two layer network because the input layer is not really considered a layer as it does no computation whatsoever.

c) W^h is a 2×3 matrix, while W^o is a 3×2 matrix.

d) The weights on the same column all go to the same neuron.

e) Colored weights

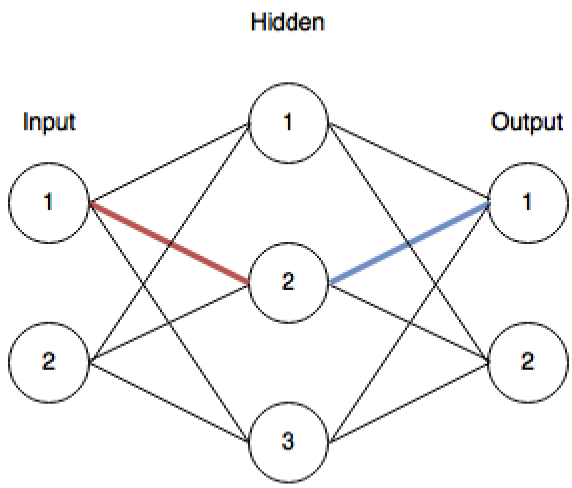

Figure 2: Neural networks with highlighted weights: w^h_{12} in red and w^o_{21} in blue

f) With the augmented input layer, W^h is a 3×3 matrix, while W^o is a 3×2 matrix.

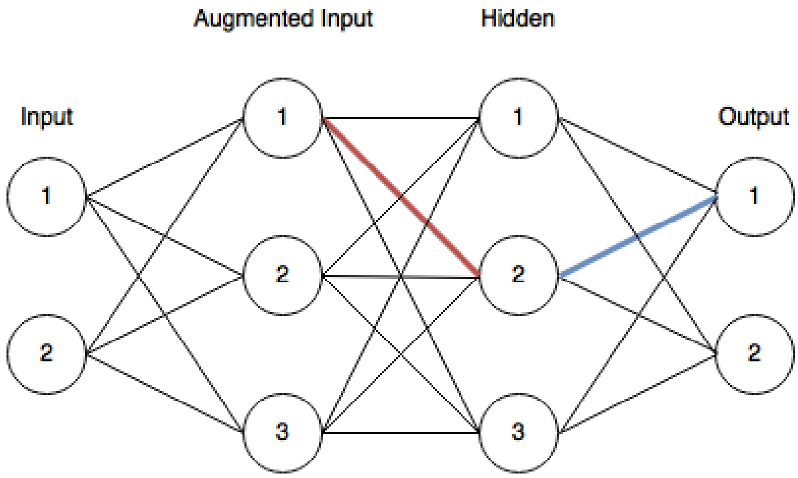

Figure 3: Neural network with augmented input layer

g) The values of this table can be found on the first row of W^h

input neuron	weights to first hidden neuron
1	0.3
2	0.6
3	0.5

Table 1: Connections from the augmented input neurons to the first hidden neuron

h) Given the input vector $\vec{x} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}^T$ and the weight vector $\vec{w} = \begin{pmatrix} 0.3 \\ 0.6 \\ 0.5 \end{pmatrix}$, we can easily calculate the activation using the following equation:

$$a = \vec{x} \cdot \vec{w} = 1 \cdot 0.3 + 0 \cdot 0.6 + (-1) \cdot 0.5 = -0.2 \tag{3}$$

i) By multiplying $\vec{x} \cdot W^h$ we end up with an array $\vec{a}$ whose i-th is exactly $\vec{x} \cdot Wh_i$, where W_i^h is the i-th column of the W^h matrix. This means that the array calculated by the multiplication holds the activation for each neuron.

j) To calculate the activation of the output layer, we simply need to multiply the output vector of the hidden layer y^h, a 1×3 array, by the weights of the output layer, W^o, a 3×2 matrix, similar to what we did to calculate the activation of the hidden layer. Therefore, we get:

$$\vec{a}^o = y^h \cdot W^o \tag{4}$$

Notice that the dimensions match and that, as expected, we end up with 2 outputs.

3 Implementing an MLP in Matlab

1. **MLP**

```
% mlp.m Implementation of the Multi-Layer Perceptron

clear all
close all

examples = [0 0;1 0;0 1;1 1];
goal = [0.01 0.99 0.99 0.01]';

min_error = 0.01;

% Boolean for plotting the animation
plot_animation = true;

% Parameters for the network
learn_rate = 0.2;               % learning rate
max_epoch = 5000;              % maximum number of epochs

mean_weight = 0;
weight_spread = 5;

n_input = size(examples,2);
n_hidden = 20;
n_output = size(goal,2);

% Noise level at the input
noise_level = 0.05;

% Activation of the bias node
bias_value = -1;


% Initializing the weights
w_hidden = rand(n_input + 1, n_hidden) .* weight_spread - weight_spread/2 + mean_weight;
w_output = rand(n_hidden, n_output) .* weight_spread - weight_spread/2 + mean_weight;

% Start training
stop_criterium = 0;
epoch = 0;

while ~stop_criterium
    epoch = epoch + 1;

    % Add noise to the input data.
    noise = randn(size(examples)) .* noise_level;
    input_data = examples + noise;

    % Append bias to input data
    input_data(:,n_input+1) = ones(size(examples,1),1) .* bias_value;
```

```

    epoch_error = 0;
    epoch_delta_hidden = 0;
    epoch_delta_output = 0;

    % FROM HEREON YOU NEED TO MODIFY THE CODE!
    for pattern = 1:size(input_data,1)

        % Compute the activation in the hidden layer
        hidden_activation = input_data(pattern, :) * w_hidden;

        % Compute the output of the hidden layer (don't modify this)
        hidden_output = sigmoid(hidden_activation);

        % Compute the activation of the output neurons
        output_activation = hidden_output * w_output;

        % Compute the output
        output = output_function(output_activation);

        % Compute the error on the output
        err = goal(pattern, :)' - output;
        output_error = 0.5 * err;

        % Compute local gradient of output layer
        local_gradient_output = d_sigmoid(output_activation) .* (err);

        % Compute the error on the hidden layer (backpropagate)
        hidden_error = 0.5 * (goal(pattern, :)' - hidden_output);

        % Compute local gradient of hidden layer
        local_gradient_hidden = d_output_function(hidden_activation) .* (w_output *
     local_gradient_output')';

        % Compute the delta rule for the output
        delta_output = learn_rate .*  hidden_output' * local_gradient_output;

        % Compute the delta rule for the hidden units;
        delta_hidden = learn_rate .* input_data(pattern, :)' *
    local_gradient_hidden;

        % Update the weight matrices
        w_hidden = w_hidden + delta_hidden;
        w_output = w_output + delta_output;

        % Store data
        epoch_error = epoch_error + (output_error).^2;
        epoch_delta_output = epoch_delta_output + sum(sum(abs(delta_output)));
        epoch_delta_hidden = epoch_delta_hidden + sum(sum(abs(delta_hidden)));
    end

    % Log data
    h_error(epoch) = epoch_error / size(input_data,1);
```

```
    log_delta_output(epoch) = epoch_delta_output;
    log_delta_hidden(epoch) = epoch_delta_hidden;

    % Check whether maximum number of epochs is reached
    if epoch > max_epoch
        stop_criterium = 1;
    end

    % Implement a stop criterion here
    if epoch_error < min_error
        stop_criterium = 1;
    end

    % Plot the animation
    if and((mod(epoch,20)==0),(plot_animation))
        emp_output = zeros(21,21);
        figure(1)
        for x1 = 1:21
            for x2 =  1:21
                hidden_act = sigmoid([(x1/20 - 0.05) (x2/20 -0.05) bias_value] *
    w_hidden);
                emp_output(x1,x2) = output_function(hidden_act * w_output);
            end
        end
        surf(0:0.05:1,0:0.05:1,emp_output)
        title(['Network epoch no: ' num2str(epoch)]);
        xlabel('input 1: (0 to 1 step 0.05)')
        ylabel('input 2: (0 to 1 step 0.05)')
        zlabel('Output of network')
        zlim([0 1])
    end

end

% Plotting the error
figure(2)
plot(1:epoch,h_error)
title('Mean squared error vs epoch');
xlabel('Epoch no.');
ylabel('MSE');

% Add additional plot functions here (optional)
```

Listing 1: mlp.m

2. Sigmoid Function

```
%this functions calculates the sigmoid
function [output] = sigmoid(x)
    output = 1 ./ (1 + exp(-x));
end
```

Listing 2: sigmoid.m

3. Output Function

```
function [output] = output_function(x)
    output = sigmoid(x);
end
```

Listing 3: outputfunction.m

4. Derivative of the Sigmoid Function

```
%this functions calculates the differential of the sigmoid
function [output] = d_sigmoid(x)
    temp = sigmoid(x);
    output = temp .* (1 - temp);
end
```

Listing 4: dsigmoid.m

5. Derivative of the Output Function

```
%this functions calculates the differential of the output function
function [output] = d_output_function(x)
    output = d_sigmoid(x);
end
```

Listing 5: doutputfunction.m

6. MLP Sine

```
clear all
close all

% The number of examples taken from the function
n_examples = 5;

examples = (0:2*pi/(n_examples-1):2*pi)';
goal = sin(examples);

% Boolean for plotting animation
plot_animation = true;
plot_bigger_picture = false;

% Parameters for the network
learn_rate = 0.05;                 % learning rate
max_epoch = 5000;               % maximum number of epochs


mean_weight = 0;
weight_spread = 1;

n_input = size(examples,2);
n_hidden = 20;
n_output = size(goal,2);

% Noise level at input
noise_level = 0.05;

```

```
bias_value = -1;

% Initializing the weights
w_hidden = rand(n_input + 1, n_hidden) .* weight_spread - weight_spread/2 +
    mean_weight;
w_output = rand(n_hidden, n_output) .* weight_spread - weight_spread/2 +
    mean_weight;

% Start training
stop_criterium = 0;
epoch = 0;

min_error = 0.01;

while ~stop_criterium
    epoch = epoch + 1;

    % Add noise to the input
    noise = randn(size(examples)) .* noise_level;
    input_data = examples + noise;

    % Append bias
    input_data(:,n_input+1) = ones(size(examples,1),1) .* bias_value;

    epoch_error = 0;
    epoch_delta_hidden = 0;
    epoch_delta_output = 0;
    for pattern = 1:size(input_data,1)

        % Compute the activation in the hidden layer
        hidden_activation = input_data(pattern, :) * w_hidden;

        % Compute the output of the hidden layer (don't modify this)
        hidden_output = sigmoid(hidden_activation);

        % Compute the activation of the output neurons
        output_activation = hidden_output * w_output;

        % Compute the output
        output = output_function(output_activation);

        % Compute the error on the output
        err = goal(pattern, :)' - output;
        output_error = 0.5 * err;

        % Compute local gradient of output layer
        local_gradient_output = err;

        % Compute the error on the hidden layer (backpropagate)
        hidden_error = 0.5 * (goal(pattern, :)' - hidden_output);

        % Compute local gradient of hidden layer
        local_gradient_hidden = d_output_function(hidden_activation) .* (w_output *
```

```
 local_gradient_output')';

        % Compute the delta rule for the output
        delta_output = learn_rate .*  hidden_output' * local_gradient_output;

        % Compute the delta rule for the hidden units;
        delta_hidden = learn_rate .* input_data(pattern, :)' *
    local_gradient_hidden;

        % Update the weight matrices
        w_hidden = w_hidden + delta_hidden;
        w_output = w_output + delta_output;

        % Store data
        epoch_error = epoch_error + (output_error).^2;
        epoch_delta_output = epoch_delta_output + sum(sum(abs(delta_output)));
        epoch_delta_hidden = epoch_delta_hidden + sum(sum(abs(delta_hidden)));
    end

    h_error(epoch) = epoch_error / size(input_data,1);
    log_delta_output(epoch) = epoch_delta_output;
    log_delta_hidden(epoch) = epoch_delta_hidden;

    if epoch > max_epoch
        stop_criterium = 1;
    end

    % Add your stop criterion here
    if epoch_error < min_error
        stop_criterium = 1;
    end

    % Plot the animation
    if and((mod(epoch,20)==0),(plot_animation))
        %out = zeros(21,1);
        nPoints = 100;
        input = linspace(0, 2 * pi, nPoints);
        for x=1:nPoints
            h_out = sigmoid([input(x) bias_value] * w_hidden);
            out(x) = output_function(h_out * w_output);
        end
        figure(1)
        plot(input,out,'r-','DisplayName','Output netwerk')
        hold on
        plot(input,sin(input),'b-','DisplayName','Goal (sin[x])')
        hold on
        scatter(examples, goal, 'DisplayName', 'Voorbeelden')
        hold on
        title(['Output and goal. Epoch: ' num2str(epoch)]);
        xlim([0 2*pi])
        ylim([-1.1 1.1])
        set(gca,'XTick',0:pi/2:2*pi)
        set(gca,'XTickLabel',{'0','1/2 pi','pi','3/2 pi ','2 pi'})
```

```
        xlabel('Input')
        ylabel('Output')
        legend('location','NorthEast')
        hold off
    end

end


% Plot error
figure(2)
plot(1:epoch,h_error)
title('Mean squared error vs epoch');
xlabel('Epoch nr.');
ylabel('MSE');

%Plot the bigger picture
if plot_bigger_picture
    figure(3)
    in_raw = (-5:0.1:15)';
    in_raw = horzcat(in_raw,(bias_value*ones(size(in_raw))));
    h_big = sigmoid(in_raw * w_hidden);
    o_big = output_function(h_big * w_output);

    plot(-5:0.1:15,o_big,'r-','DisplayName','Output network')
    hold on
    plot(-5:0.1:15,sin(-5:0.1:15),'b-','DisplayName','Goal (sin[x])')
    hold on
    scatter(examples, sin(examples), 'DisplayName', 'Examples');
    hold off
    xlabel('Input')
    ylabel('Output')
    legend('location','NorthEast')
    title('The bigger picture')
end
```

Listing 6: mlpsinus.m

4 Testing the MLP

a) It is not guaranteed that the network will always find a solution. That happens because the number of nodes in the hidden layer is too small, which causes the network to have too few weights to play around with in the hidden layer. This makes it so that the network sometimes doesn't converge to a solution.

b) About 2500 epochs in average are needed to find a solution with these settings.

c) The network is unable to find a solution because the noise level is now too high. The noise level makes it harder for the inputs to be correctly classified because the inputs end up being too altered to be identified.

d) The number of epochs needed to reach the solution is greatly decreased (370, down from 2500), so the network is much more efficient. That happens because now the weights are not so low anymore, which causes the local gradient of the hidden layer to be larger. That impacts the

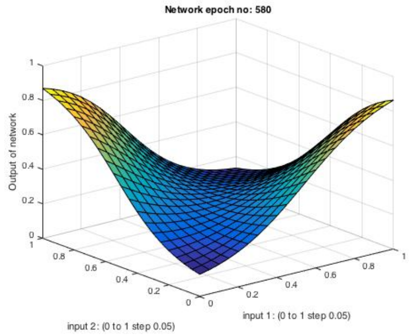

Figure 4: Solution 1

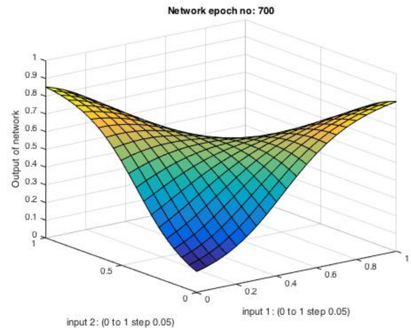

Figure 5: Solution 2

delta rule for the hidden layer directly, making it larger, which means larger steps (at least in the first few epochs). Therefore, the error decreases faster, and fewer epochs are needed to reach a reasonable weight.

e) The two different solutions to the XOR problem are as shown above. Notice that the output of the center part of Solution 1 (Figure 4) is (close to) 0, while on Solution 2 (Figure 5) it is (close to) 1. That behavior is due to the fact that the points in the center, where x_1 and x_2 have similar values get undefined behavior for XOR, since the network doesn't know if they are representing a 0 or a 1. The network therefore has to make a "decision" regarding how they'll be treated.

f) The graph of the error is usually a (noisy) decreasing exponential function. Since the delta rule is a linear function, the weights are changed linearly too. That means that the non-squared error also changes linearly, which makes the squared error decrease exponentially.

5 Another function

a) Yes, although not perfectly, the neural network is able to learn the sine function, as shown in Figure 6

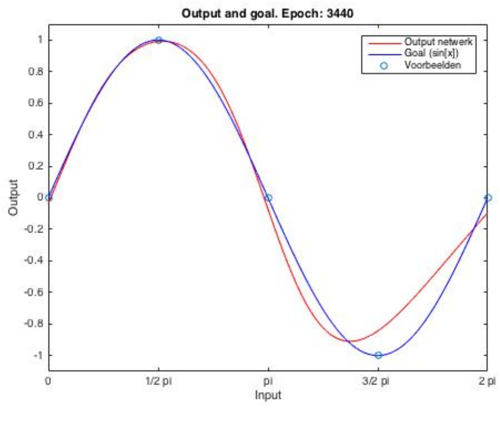

Figure 6: The network's learned function vs the sine function

b) The network is still able to learn with fewer examples. This phenomenon corresponds to the fact that a neural network reaches a generalized solution that is independent of the input.

c) The domain of the network is determined by the input set. If an input is outside of the domain, the network doesn't really know what to do with it, which leads to a wrong classification of that input. It's interesting to see that when the input is still close to the domain, the network is still able to give acceptable results, but as it gets further and further away from the domain, the results are terrible.

d) At least 3 neurons are required to learn the sine function.

e) The XOR learning network still works. The reason for that is that the sigmoid function, which was originally used as the output function for the XOR, does the same thing as the identity function in terms of making a continuous function that can be derived.