Neural Networks
Lab 3

Juan Jose Mendez Torrero (s3542416)
Sharif Hamed (s2562677)

May 17, 2018

1 Implementing an MLP in Matlab

1. In order to compute the activation of the hidden layer, we have to compute $a_h = \Sigma w_{h_i} * x_i$. Hence, we set the activation as :

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

Listing 1: Hidden layers activation

2. If we want to compute the output of the hidden layer, we have to change to the file *sigmoid.m*. It looks like Listing 2.

```
%this functions calculates the sigmoid
function [output] = sigmoid(x)
    % Define the sigmoid function here
    [output] = 1.0 ./ (1.0 + exp(-x));
end
```

Listing 2: Sigmoid function

Now, the function of the hidden output is set to $\sigma(x) = \frac{1}{1+e^{-x}}$.

3. To compute the activation of the output layer, we set it as Listing 3 shows.

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

Listing 3: Output layers activation

4. In order to compute the output function, we have implemented Listing 4.

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

Listing 4: Output function

5. In Listing 5 we can observe how is computed the derivative of the output function.

```
function [output] = d_output_function(x)
  aux=output_function(x);
    output = aux .* (1 - aux);
end
```

Listing 5: Derivative of the output function

6. in order to calculate the local gradient of the output layer, we have set the error of the output layer. Then, the local gradient is computed as Listing 6 illustrates.

```
% Compute the error on the output
output_error = goal(pattern)-output;
% Compute local gradient of output layer
local_gradient_output = d_sigmoid(output_activation).*output_error;
```

Listing 6: Output layers local gradient

7. In order to compute the local gradient in the hidden layer, we have to calculate the error on this layer. Then, as before, the local gradient is computed as Listing 7 shows.

```
% Compute the error on the hidden layer (backpropagate)
for i=1:size(w_output,1);
  aux=0;
  for j=1:size(w_output,2);
    aux = aux + (local_gradient_output * w_output(i,j));
    end
    hidden_error(i) = aux ;
end
% Compute local gradient of hidden layer
local_gradient_hidden = d_sigmoid(hidden_activation).*hidden_error;
```

Listing 7: Hidden layers local gradient

8. Here, we have compute the delta rule for the weights.

```
% 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;
```

Listing 8: Hidden layers local gradient

9. Now we update the weight as Listing 9 shows.

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

Listing 9: Hidden layers local gradient

10. In order to implement the stop criterion, we have implemented Listing 10.

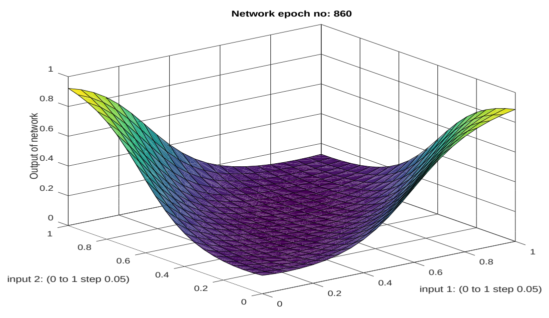
```
% Implement a stop criterion here
min_error=0.01;
if h_error(epoch) < min_error
    stop_criterium=1;
end
```

Listing 10: Hidden layers local gradient

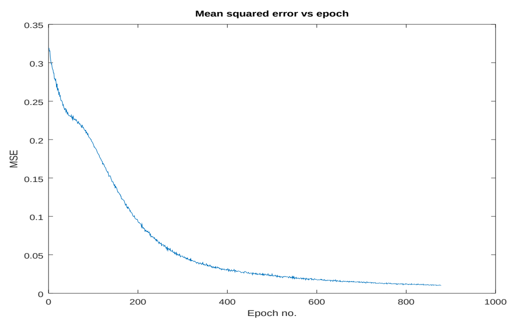
2 Testing the MLP

1. It is not guaranteed that the MLP will learn the XOR function. This is due to the number of neuron. In this case, we have used only 2 hidden neuron, therefore, is more complicated to reach the output with a small number of hidden neurons.

2. This time, we have set the number of hidden neurons to 20. That means, that the MLP will be able to learn the XOR function. The number of epoch is not always the same, that is because the weights are set randomly each test, but with the number of test that we have done, we can see that it takes around 380 and 420 epochs.
3. Setting the noise level to 0,5 what we have obtained is to have a unstable MLP, that has large spikes during the epochs. Hence, it is not guaranteed that the MLP is going to learn the XOR function.
4. By setting the noise level to 0.2 and the weight spread to 5, we have obtained a faster MLP that needs a few number of epochs to reach the solution. That is because the changes that the delta rule produces are bigger than before. Therefore, it produces large spikes when it is reaching the solution.
5. The two different solution that we can get with this setting are exposed in Figure 1 and Figure 2.

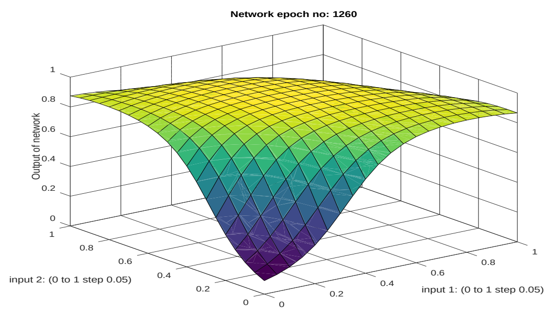


(a) Plot

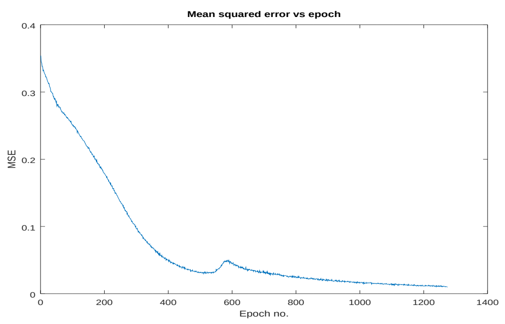


(b) Error

Figure 1: Solution A



(a) Plot



(b) Error

Figure 2: Solution B

6. It usually have the shape of Figure 1. As we can see, the MSE does not get higher values than the previous epochs.

3 Another function

For this exercise, we have set the for loop and the stop criteria the same way as before. The difference between now and before, is that, now, the output function has been changed into:

```
function [output] = output_function(x)
    % set output here
    output=x;
end
```

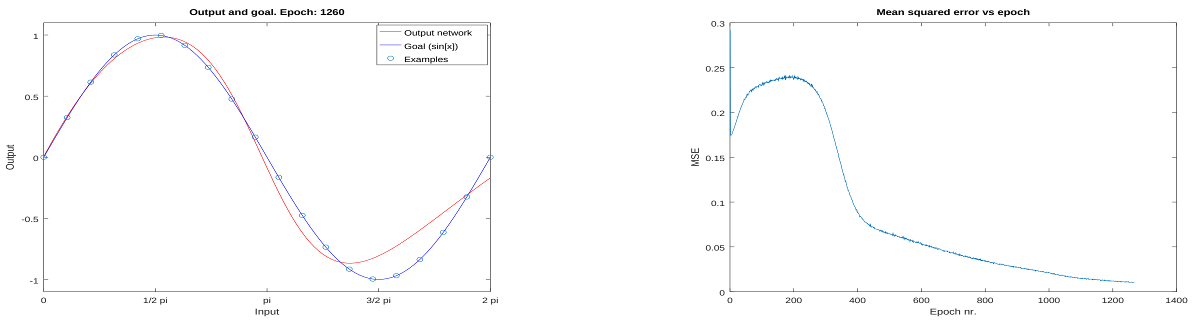
Listing 11: Output layers local gradient

As we know, the derivative of the output function is constant, hence, we have to change the derivative function as well.

```
function [output] = d_output_function(x)
    output = 1;
end
```

Listing 12: Derivative output layers local gradient

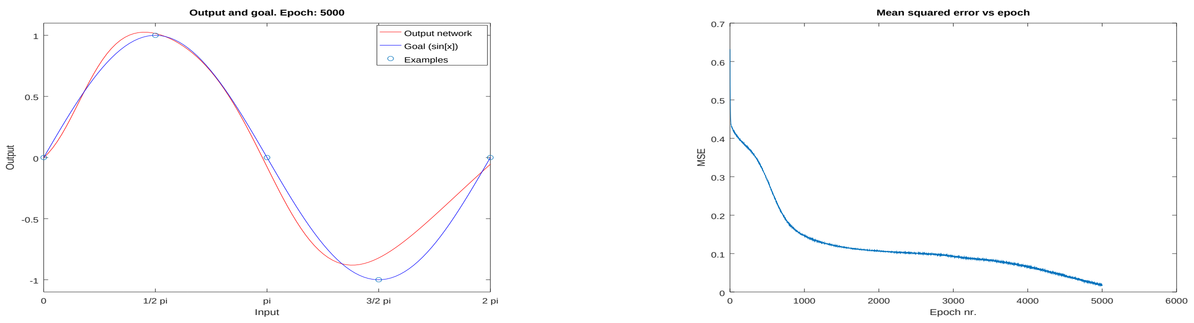
1. Yes, As we can see in Figure 3, the MLP is capable of learning the sinus function.



(a) Plot (b) Error

Figure 3: MLP learning a sinus function.

2. In Figure 4 we can observe that the MLP is still able to learn the sinus function, but in a slower way than before. The reason is the feature of generalization. This means that the neural network is able to learn from input that it has never seen before. This is because the neural network extracts statistical regularities from the training set and then it classifies the unknown pattern with one which it has seen before.



(a) Plot (b) Error

Figure 4: MLP learning a sinus function.

3. We can see in Figure 5 that the domain of the neural network is [-1,6]. If the input is outside of this domain, as we can observe, the neural network takes values that do not correspond with the target.

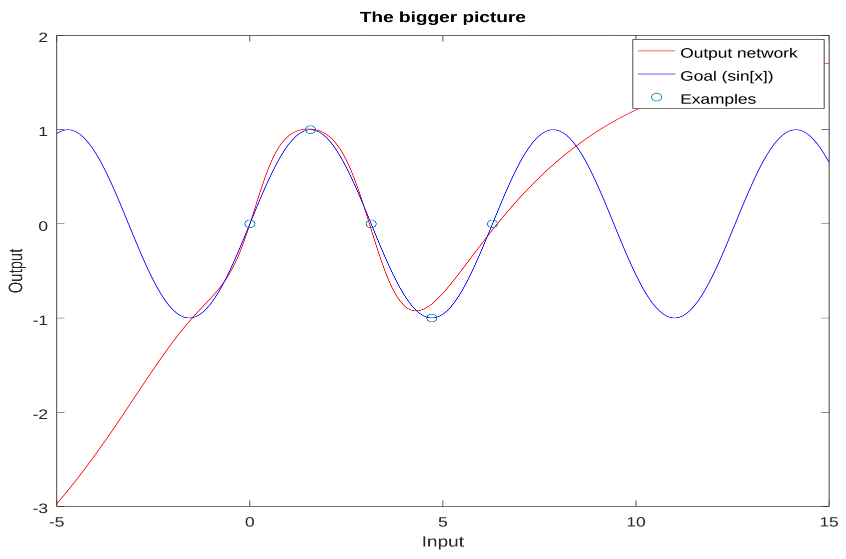
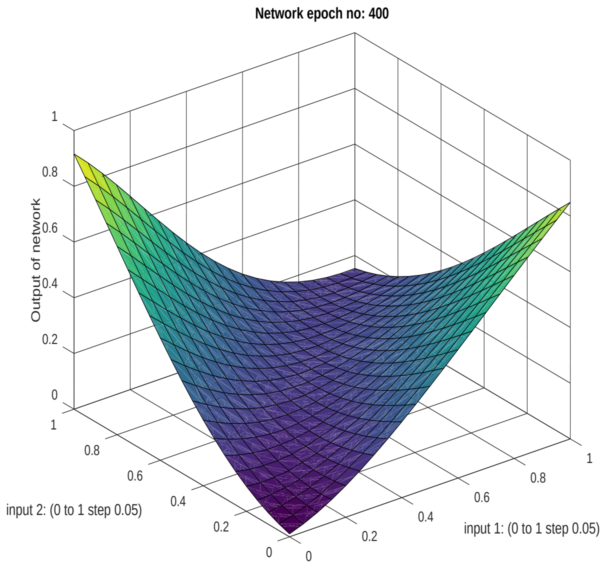


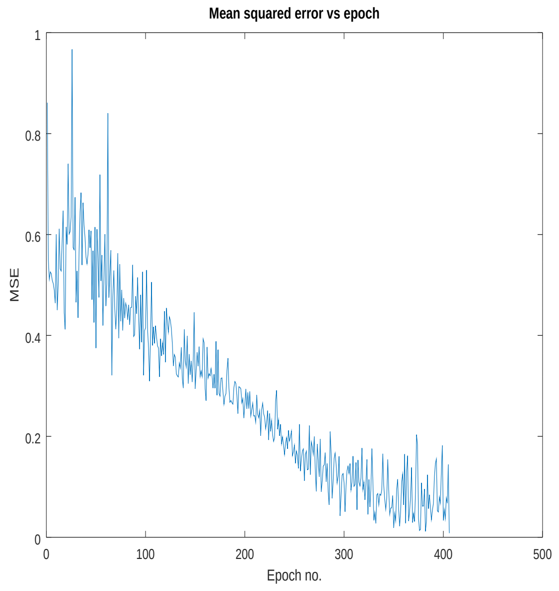
Figure 5: Big plot

4. After various tries, we have concluded that the neural network needs, at least, 4 neurons to learn the sinus function. It also works with 3, but not as good as with 4 neurons.

5. After changing the output function, we can see in Figure 6 that the MLP is still able to learn the XOR function. This does not change anything because we are still using the sigmoid function to calculate the local gradient of the hidden layer.



(a) Plot



(b) Error

Figure 6: XOR with $f(x) = x$