Neural Networks for AI

Lab 1

Fernanda Marana (s3902064)
Floris Cornel (s2724685)

May 2, 2019

1 Introduction

2 Implementing a TLU in Matlab

The Listings below show the codes implemented for a TLU using two inputs and a single output that can learn to perform logical operations such as an AND or NAND function.

```
% Fernanda Marana
% Floris Cornel
```

Listing 1: names.m

Listing1 refers to the a file with the name of the group's members written down.

```
function [output] = examples()
    output = [
        0 0
        0 1
        1 0
        1 1
    ];
end
```

Listing 2: examples.m

Listing2 shows a nullary (0 arguments) function named examples (in file examples.m) that returns a matrix with 4 rows and 2 columns.

```
function [output] = goalAND()
    output = [0; 0; 0; 1];
end
```

Listing 3: goalAND.m

Listing3 represents a nullary function (goalAND) that returns a column vector that gives the desired output corresponding to the inputs of examples.

```
function [bias] = initB()
    bias = rand(1);
end
```

Listing 4: initB.m

```
function [weight] = initW()
    weight = rand(1);
end
```

Listing 5: initW.m

Listing 4 and Listing 5 return respectively the initial weight matrix and the initial threshold, both with random values between 0 and 1.

```
function [weightedSum] = summedInput(x, weights, threshold)
    weightedSum = sum([x -1] .* [weights threshold]);
end
```

Listing 6: summedInput.m

Listing 6 is the implementation of a ternary function which h takes as arguments an input vector (x), the weight matrix (W), and the bias (b), and returns the weighted sum of the current input (x). The bias should be also applied to the weighted sum.

```
function [output] = stepA(input)
    output = (input >= 0);
end
```

Listing 7: stepA.m

Listing 7 shows a unary threshold-function and since the bias is applied in summedInput, the threshold is just 0. Additionally the output obtains the correct value.

```
function [error] = pError(t, o)
    error = (t - o);
end
```

Listing 8: pError.m

Listing 8 is a binary function that given (in this order) a target (t) and an output (o) returns the network's error for this a pattern (test case).

```
function [dw] = deltaW(learn_rate, error, weights)
    dw = learn_rate * weights * error;
end
```

Listing 9: deltaW.m

Listing9 is a ternary fugiven (in this order) a learning rate, an error (for a specific pattern) and an input vector, returns the delta for the weight matrix.

```
function [db] = deltaB(learn_rate, error)
    db = learn_rate * -1 * error;
end
```

Listing 10: deltaB.m

Listing10 represents the function deltaB that given (in this order) a learning rate and an error (for a specific pattern), returns the delta for the bias.

```
function [newW] = upW(weights, deltaW)
    newW = weights + deltaW;
end
```

Listing 11: upW.m

```
function [newB] = upB(bias, deltaB)
    newB = bias + deltaB;
end
```

Listing 12: upB.m

Listing11and Listing 12 represent the functions upW and upB that, given (in this order) the previous values (of weights and bias respectively) and the deltas (of weights and bias respectively) returns the updated weights and threshold (respectively).

```
function [weights, threshold, error] = netStep(learn_rate, weights, threshold, x, t)

        % Initialize weighted sum of inputs
        summed_input = summedInput(x, weights, threshold);

        % Subtract threshold from weighted sum
        % - This is done in the summedInput function

        % Compute output
        output = stepA(summed_input);

        % Compute error
        error = pError(t, output);

        % Compute perceptron rule
        delta_weights = deltaW(learn_rate, error, weights);
        delta_threshold = deltaB(learn_rate, error);

        % Update weights and threshold
        weights = upW(weights, delta_weights);
        threshold = upB(threshold, delta_threshold);

end
```

Listing 13: netStep.m

Listing11 is the function netSep that puts everything together. Given (in order) a learning rate, a current weight matrix, a current bias, an input vector, a target (for that input) will return three values (in order) which are the updated weights, the updated bias, the epoch's error.

3 Experimenting with a TLU

Figure 1 and Figure 2 represent a common case in which the learning rate is 0.1 and the input is 0 or 1. Based on this example it is possible to compare and discuss some differences between the changes done later in this section. As we can observe on Figure 1, the weights may be higher or lower than the threshold but after some epochs they tend to be lower. On Figure 1, it is possible to see that a point in time where the weight 2 is turned into a smaller value than the threshold. This is the same point that the error line converges to zero on Figure 2.

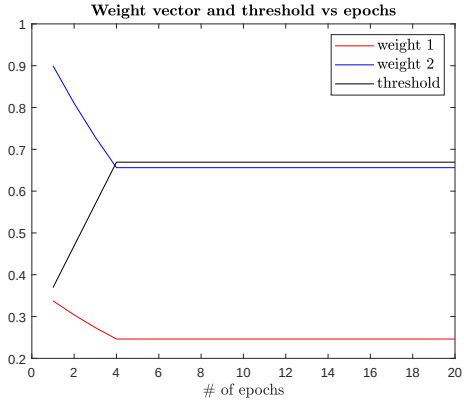

Figure 1: Weights and threshold.

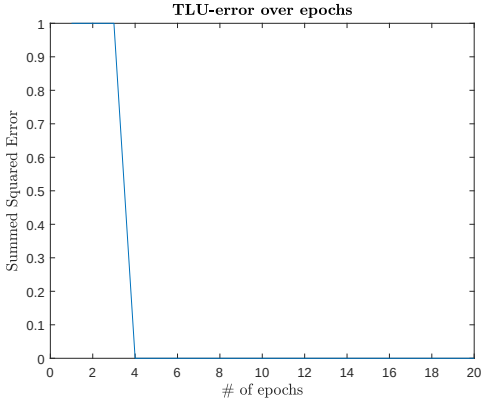

Figure 2: Summed Square Error.

At some cases it is valid to notice that the error does not decrease each epoch. This happens because as seen on Listing 8, the error is only the difference between the target and the output. Thus, when the target exceeds the output, the error becomes positive and the adjustment of the weights can distant itself from the zero at some epochs. Due to this problem, it is more interesting to work on purely positive quantity by using a summed squared error. With this, the signal does not become a problem and the error turns into a function that to be minimized depends on its variables in a smooth,continuous fashion. It is important to see that the initialization of weights and the initial threshold is both with random values between 0 and 1. So in some cases, it may take more or less epochs to reach an error of 0.

On Figure 3 and Figure 4, there are some observable changes as a result of the increase on the learning rate from 0.1 to 0.6. The inputs tend to adjust more than the threshold that does not appear to have much changes. By having in mind that these are just examples of multiple tests and that comparison were made between them, it is impossible to conclude if the higher the learning rate the better without thinking about the criteria. A high learning rate can take less time because it makes the delta for the weights and the delta for the bias increase and thus converge faster. However the low learning rate seems more reliable although its optimization process does take some time. If weight changes are too big the optimizer can overshoot the minimum and make the loss worse. So, as a result, having a low learning rate seems better.

Learning rate:

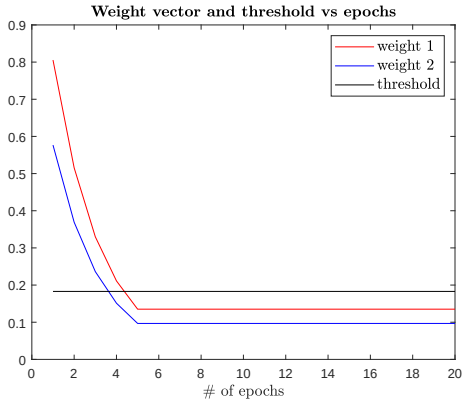

Figure 3: Weights and threshold.

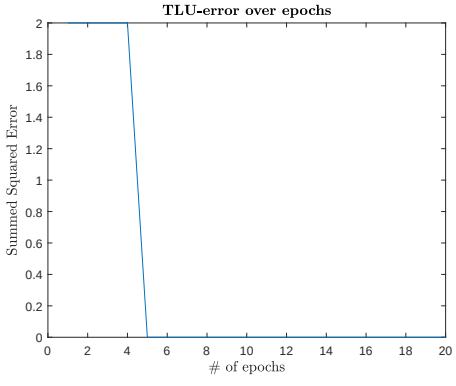

Figure 4: Summed Square Error.

Figures 5 and 6 show the changes made when the input is set to 0.2 or 0.8 instead of the binary value 0 or 1. It is possible to notice that the TLU is capable of learning the AND-function because

its error converges to zero. That happens because we encounter an important feature of artificial neural networks: no matter the type of value format is in the input as long as the function is linear separable it is possible for the TLU to learn it. 0.8 and 0.2:

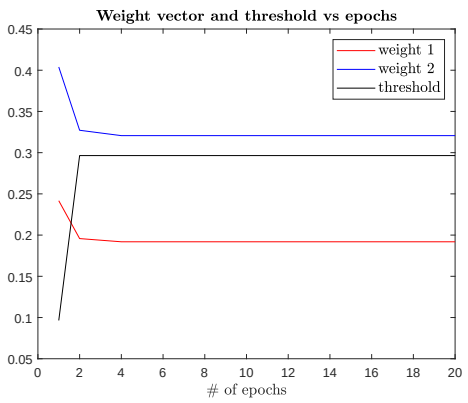

Figure 5: Weights and threshold.

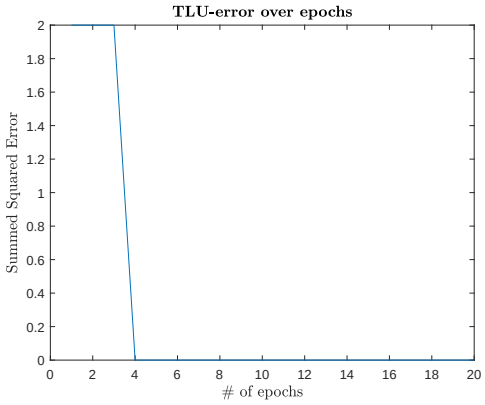

Figure 6: Summed Square Error.

Figure 7 and 8, shows now the result of a NAND-fucntion that works as well. It is possible to see that the weight values get lower and lower while the weigh stops reducing at some point. The group was not able to get an example where the threshold has become negative. NAND:

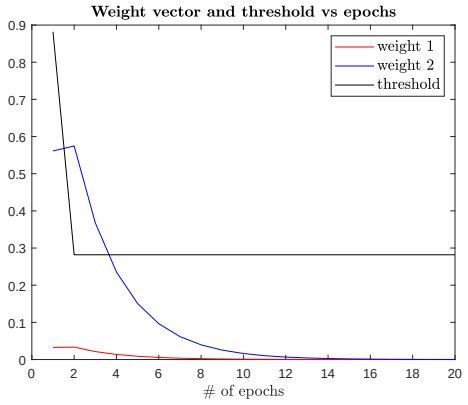

Figure 7: Weights and threshold.

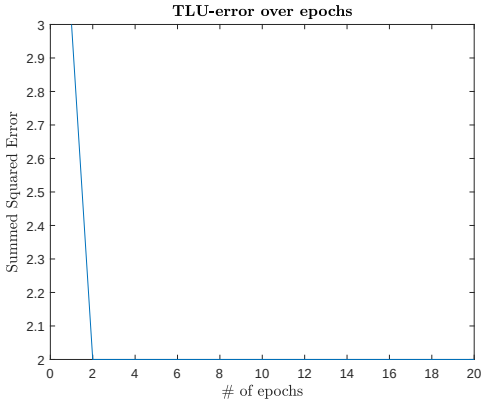

Figure 8: Summed Square Error.

4 XOR-rule

On Figures 9 and 10 it is shown the results of changing the goal vector such that the TLU learns a XOR-function. Since the XOR-rule is not linear separable the error does not converge to zero.

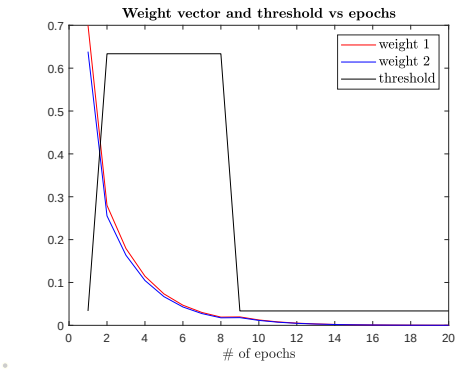

Figure 9: Weights and threshold.

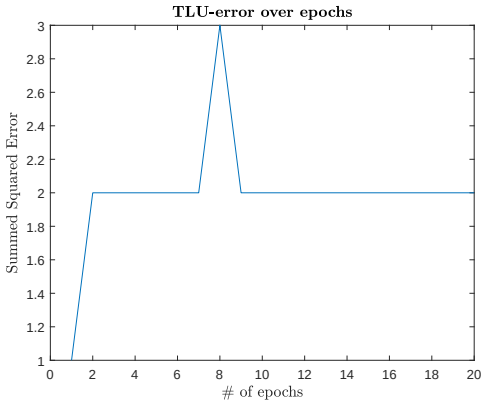

Figure 10: Summed Square Error.

5 Code extensions

The code below (Listing14) shows the group implementation of a TLU that stops learning after all examples are classified correctly and plots the graphs.

```
% TLU implementation
% Fernanda Marana (s3902064) & Floris Cornel (s2724685)

% Parameters
learn_rate = 0.1;    % the learning rate
n_epochs = 100;      % the number of epochs we want to train

% Define the inputs
examplesInput = examples();

% Define the corresponding target outputs
goal = goalAND();

% Initialize the weights and the threshold
weights = [initW() initW()];
threshold = initB();

% Preallocate vectors for efficiency. They are used to log your data
% The 'h' is for history
h_error = zeros(n_epochs,1);
h_weights = zeros(n_epochs,2);
h_threshold = zeros(n_epochs,1);

% Store number of examples and number of inputs per example
n_examples = size(examplesInput,1);     % The number of input patterns
n_inputs = size(examplesInput,2);       % The number of inputs

for epoch = 1:n_epochs
    epoch_error = zeros(n_examples,1);

    h_weights(epoch,:) = weights;
    h_threshold(epoch) = threshold;

```

```
    for pattern = 1:n_examples

        [weights, threshold, error] = netStep(learn_rate, weights, threshold,
    examplesInput(pattern, :), goal(pattern));

      % Store squared error
        epoch_error(pattern) = error.^2;

    end

    h_error(epoch) = sum(epoch_error);
    if (h_error(epoch) == 0)
        break;
    end
end

% Plot functions
figure(1);
plot(h_error(1:epoch))
title('\textbf{TLU-error over epochs}', 'interpreter', 'latex', 'fontsize', 12);
xlabel('\# of epochs', 'interpreter', 'latex', 'fontsize', 12)
ylabel('Summed Squared Error', 'interpreter', 'latex', 'fontsize', 12)

figure(2);
plot(1:epoch,h_weights(1:epoch,1),'r-','DisplayName','weight 1')
hold on
plot(1:epoch,h_weights(1:epoch,2),'b-','DisplayName','weight 2')
plot(1:epoch,h_threshold(1:epoch),'k-','DisplayName','threshold')
xlabel('\# of epochs', 'interpreter', 'latex', 'fontsize', 12)
title('\textbf{Weight vector and threshold vs epochs}', 'interpreter', 'latex', '
    fontsize', 12);
h = legend('location','NorthEast');
set(h, 'interpreter', 'latex', 'fontsize', 12);
hold off
```

Listing 14: tlu.m