CSS 485

Jun Hyung Park

Jan 13, 2020

1. Consider a single-input neuron on with a bias. We would like the output to be 0 for inputs less than 3 and 1 for inputs greater than or equal to 3

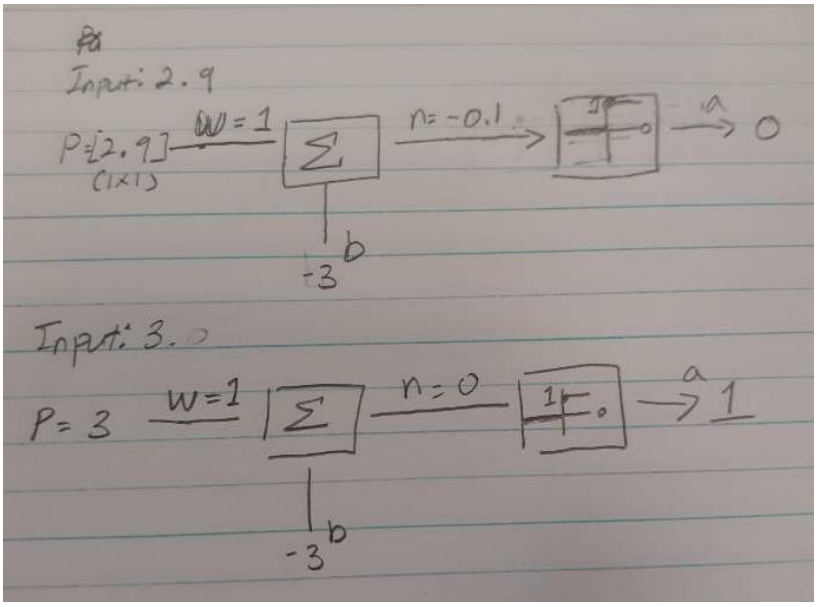
a) What kind of transfer function is required?

hardlim

b) What weight and bias would you suggest? Is your bias in any way related to the weight? If yes, how?
w = 1 b = -3

Bias is related to weight by time 3.
I would approach as w = x then b = -3x
for instance, if w = 1, then b = -3; w = -5 then b = 15.

c) Summarize your network by naming the transfer function and stating the bias and the weight. Draw a diagram of the network.



d) Write a Mathlab function that implements your transfer function (or use the appropriate transfer function from NND). Then write a neuron function uses that transfer function to implement your neuron design. Make sure you follow the convention of putting each Mathlab function in a.m file of the name same. Verify using mathlab that it performs as expected. Include both function and demonstration output in your document.

Mathlab function

```
format compact

myInput = 3.0;
fprintf("My Output: %d\n", a(myInput))

% NND transfer function, hardlim(n)
function n = a(input_)
    weight = 1;
    bias = -3;
    n_value = (weight * input_) + bias;
    fprintf("My Input: %d\n", input_)
    n = func(n_value);
end

function transferFunc = func(n_value)
        transferFunc = hardlim(n_value);
end
```

Output

My Input: 3
My Output: 1

My Input: 2.999900e+00
My Output: 0

2) Modify your single-input neuron function so that it takes multiple inputs and computes the scalar output for that neuron. This function should take three arguments: an input vector, a weight vector, and bias (also scalar). This function should call the same transfer functions as the single-input neuron on function from question 1. Implement this multi-input neuron function two ways:

a) Use a for loop to iterate over the inputs and weights to compute their inner product

```
neuron(myInput, myWeight, myBias)

function transferFunc = func(n_value)
        transferFunc = hardlim(n_value);
end

% modifiy the function
% inner product using loop
function result_a = neuron(input_, weight_, bias_)
    % if the lenght of input and weight is different
    % in vector inner product, it is not possible to get scalar
    if (length(input_) ~= length(weight_))
        disp("Number of Element should be same to conduct" + ...
```

```
            " inner product" + ...
            " exiting the Function")
        return
    end
    productResult = 0;
    for i = 1:length(input_)
        productResult = productResult + (input_(i) * weight_(i));
    end
    % Add bias to productResult to get summation
    summation = productResult + bias_;
    result_a = func(summation);
end
```

b) Use the fact that Matlab can directly multiply vectors, plus the Matlab transpose (‘) operator, to compute the inner product in a single mathematical expression

```
neuronTranspose(myInput, myWeight, myBias)

function transferFunc = func(n_value)
        transferFunc = hardlim(n_value);
end

function result_a2 = neuronTranspose(input_, weight_, bias_)
    % if the length of input and weight is differnt
    % in vector inner product, it is not possible to get scalar
    if (length(input_) ~= length(weight_))
        disp("Number of Element should be same to conduct" + ...
            " inner product" + ...
            " exiting the Function")
        return
    end
    summation = (weight_ * input_') + bias_;
    result_a2 = func(summation)
end
```

3) Implement a layer function that implements a single layer of multiple neurons of the type from question 2. This function should take three arguments: an input vector, a weight matrix, and a bias vector. The function should return a single value: a vector of a neuron outputs. In other words, this is a function which can be used to compute the output of multiple neurons. This function should call the same transfer functions as the single-neuron function from question 1. Implement this layer function two ways:

a) use a for loop to iterate over the neurons in the layer, computing each neuron’s output and appending its output to the output vector (thus assembling the layer’s output vector sequentially).

```
% test data
sprintf("OR   AND")
inputV1 = [1 0];
weightM = [1 1; 1 1];
biasV = [-1 -2];
% end of test data
layer(inputV1, weightM, biasV)
```

```
function transferFunc = func(n_value)
        transferFunc = hardlim(n_value);
end

function summation = layer(inputV_, weightM_, biasV_ )
    [weightRows, weightCols] = size(weightM_);
    [inputRows, inputCols] = size(inputV_);
    transposeBias = biasV_';
    % create a variable that hold summation value
    % size will be weightRows, inputRows
    result = zeros(weightRows, inputRows);
    for i = 1:weightRows
        for j = 1:weightCols
            result(i) = (result(i) + (weightM_(i,j) * inputV_(j)));
        end
        result(i) = result(i) + transposeBias(i);
    end
    % convert column vector into row vector
    summation = func(result');
end
```

Result

"OR AND"

inputV1 =

1 0

ans =

1 0

"OR AND"

inputV1 =

0 1

ans =

1 0

ans =

"OR AND"

inputV1 =

1 1

ans =

1 1

ans =

"OR AND"

inputV1 =

0 0

ans =

0 0

b) use Matlab matrix/vector mathematical operations to compute the outputs of all of the neurons in the layer in parallel, using no loop

```
% test data
inputV1 = [1 0];
weightM = [1 1; 1 1];
biasV = [-1 -2];
% end of test data
layerMathlabOperation(inputV1, weightM, biasV)

function transferFunc = func(n_value)
        transferFunc = hardlim(n_value);
end

function summation = layerMathlabOperation(inputV_, weightM_, biasV_)
    n = weightM_ * inputV_' + biasV_';
    % use ' to convert to vector looks better
    summation = func(n');
end
```

Result

"OR AND"

inputV1 =

1 0

ans =

1 0

"OR AND"

inputV1 =

0 1

ans =

1 0

```
ans =

   "OR   AND"

inputV1 =

    1     1

ans =

    1     1

ans =

   "OR   AND"

inputV1 =

    0     0

ans =

    0     0
```

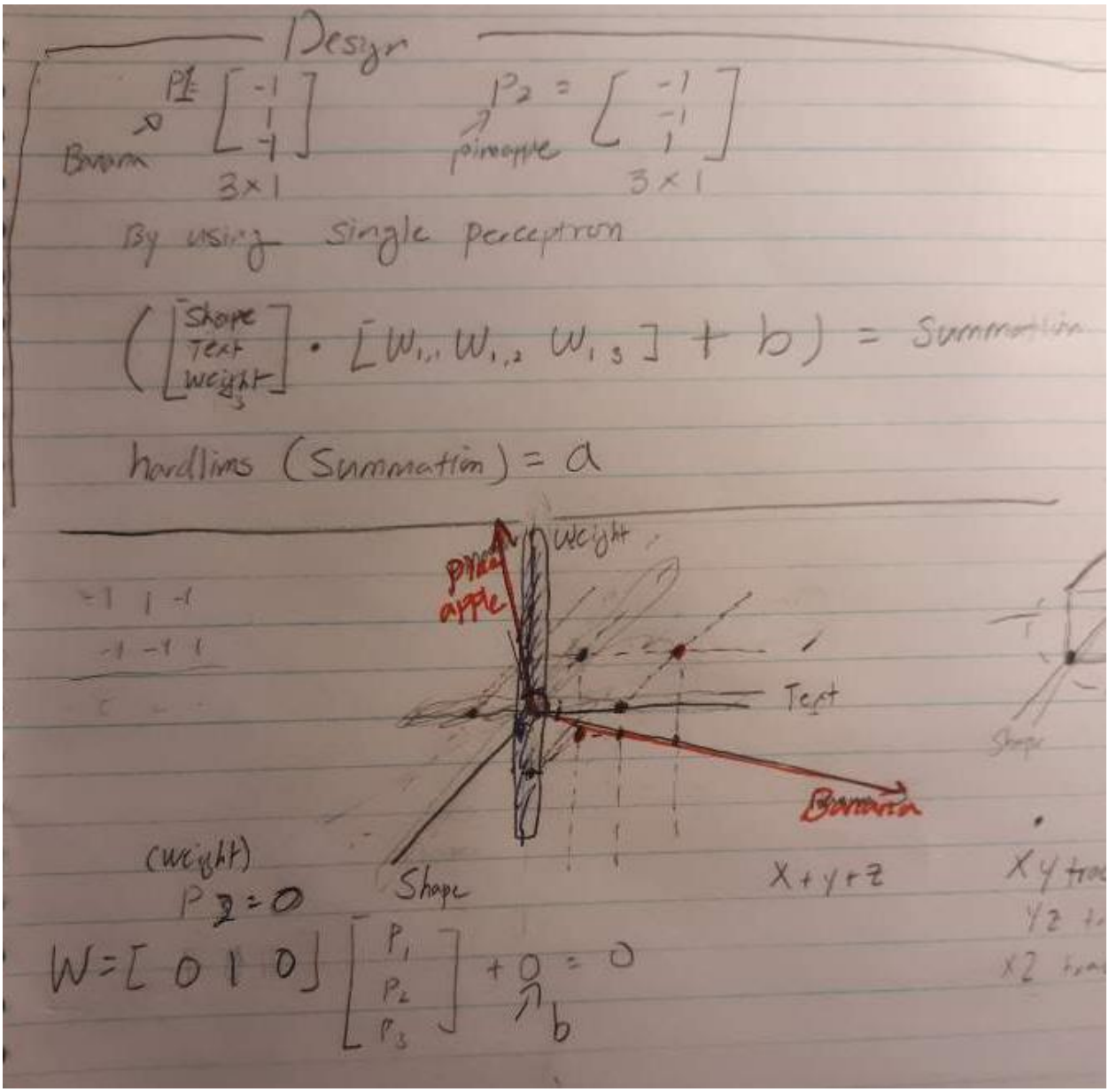
4) Chapter 3 of the textbook designed three different neural networks to distinguish between apples and oranges, based on three sensor measurements (shape, texture, and weight). Suppose that we want to distinguish between bananas and pineapples:

P1 = [-1; 1; -1] (Banana)

P2 = [-1; -1; 1] (Pineapple)

Design a perceptron to recognize these patterns. Explain what choices you needed to make. Present your weights and biases and use your Matlab code from 2 or 3 (with the transfer function called changed, of course, since this uses a different one) to illustrate that this network works. Include this demonstration output in your document.

Design:



Design Reason:

Since there are only two categories exist (Banana and Pineapple), as book proposed single neuron perceptron is fine.

To get the correct weight and bias, I had created prototype vectors (pineapple and banana) and find the linear boundary.
Based on the decision boundary I have, my weight is [0 1 0] and bias is 0

Function used is hardlims (Symmetric hard-limit transfer function) just as book proposed.

Mathlab:

```
% testing data for banana and pineapple
```

```
bananaP1 = [-1 1 -1]
pineappleP2 = [-1 -1 1]
decisionWeight = [0 1 0]
bias = 0
layerMathlabOperation(bananaP1, decisionWeight, bias)
layerMathlabOperation(pineappleP2, decisionWeight, bias)
% end of testing

function summation = layerMathlabOperation(inputV_, weightM_, biasV_)
    n = weightM_ * inputV_' + biasV_';
    % use ' to convert to vector looks better
    summation = changedFunc(n);
end

% this function is modified version of hardlims
% instead hardlims return 1 if x >= 0, display Banana
% instead hardlims return -1 if x < 0, display Pineapple
function transferFunction = changedFunc(n_value)
    if n_value >= 0
        transferFunction = "Banana";
    else
        transferFunction = "pineapple";
    end
end
```

Output

bananaP1 =

-1 1 -1

decisionWeight =

0 1 0

bias =

0

ans =

"Banana"

pineappleP2 =

-1 -1 1

decisionWeight =

0 1 0

```
bias =
     0
ans =
    "pineapple"
```