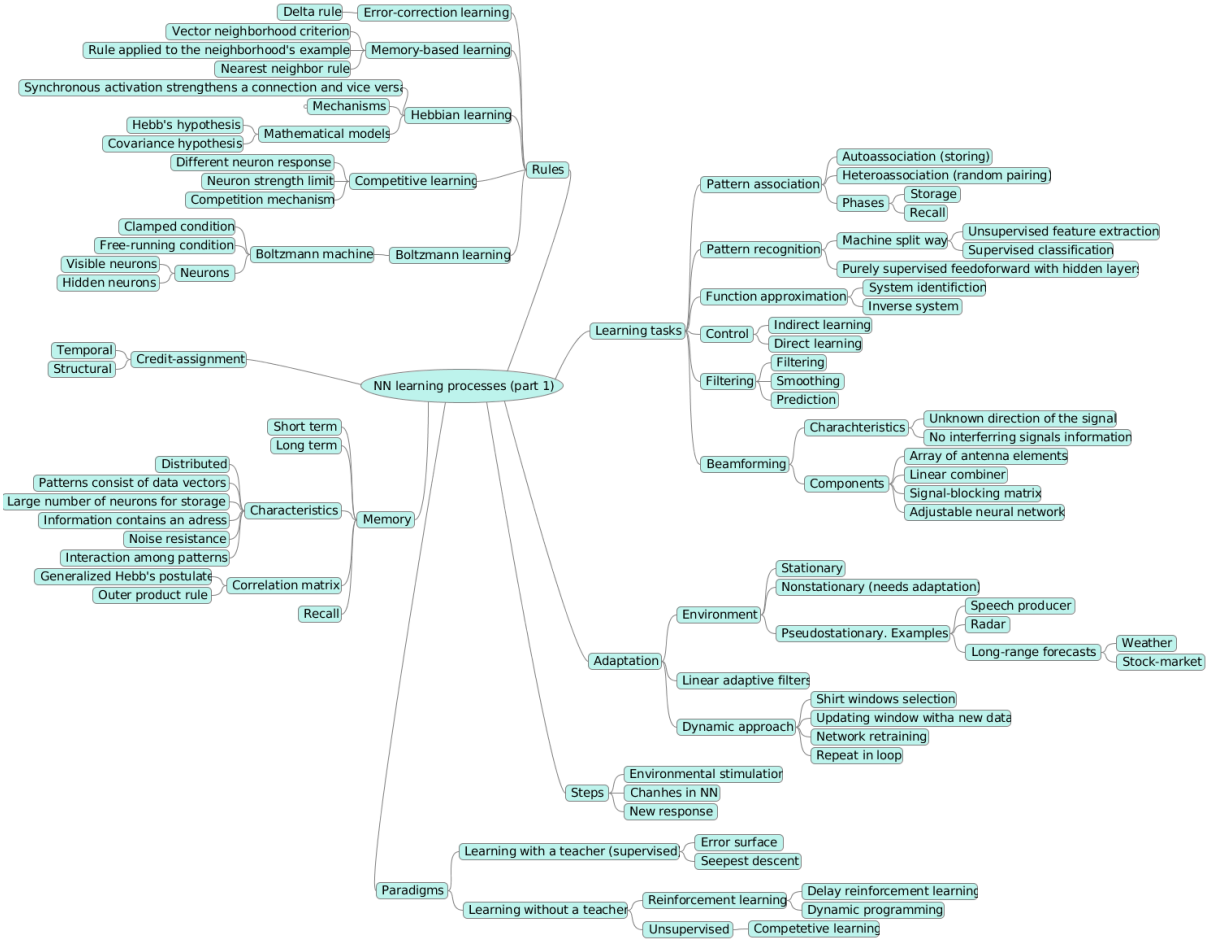
Neural Networks
- Homework 2 -

Petr Lukin, Ivan Vishniakou, Evgeniya Ovchinnikova

Lecture date: 10 October 2016

1 Mind map

Figure 1: S. Haykin, Neural Networks, chapter 2 (before 2.13). Mind map (a zoomed version of the map is attached as mindmap.png file).



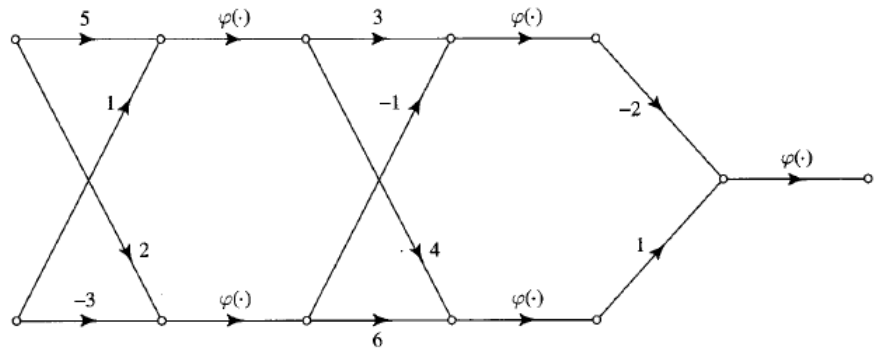
2 Exercises

2.1 Exercise 1.13

Do the problem 1.13 (Network architecture) from the previous weeks assignment. This time use MATLAB or Pythons (sympy) symbolic toolbox. Finally assume the network presented in fig P1.13 is a binary-classifier, please depict how the input space (R2) is classified on a 2D graph using different colors.

This exercise was solved in matlab. The input-output mapping can be made for a network with structure $2-2-..-2-1$ with different weights and squashing functions: linear, sigmoid. Output layer also can have heaviside squashing function.

The network itself has a following structure:



Here is the code of the hidden layer output implementation.

```
function [ out ] = hiddenlayer( input,weights,squash_type )

if strcmp(squash_type,'sigmoid')
    squash = @(v) exp(v)./(1+exp(v));
else
    squash = @(v) v;
end
out = [squash(dot(weights(1:2),input)),squash(dot(weights(3:4),
   input))];

end
```

The output layer.

```
function [ out ] = outputlayer( input,weights,squash_type )

if strcmp(squash_type,'sigmoid')
    squash = @(v) exp(v)./(1+exp(v));
else
    squash = @(v) v;
end
if strcmp(squash_type,'heaviside')
    squash = @(v) heaviside(v);
end
out = squash(dot(weights,input));
```

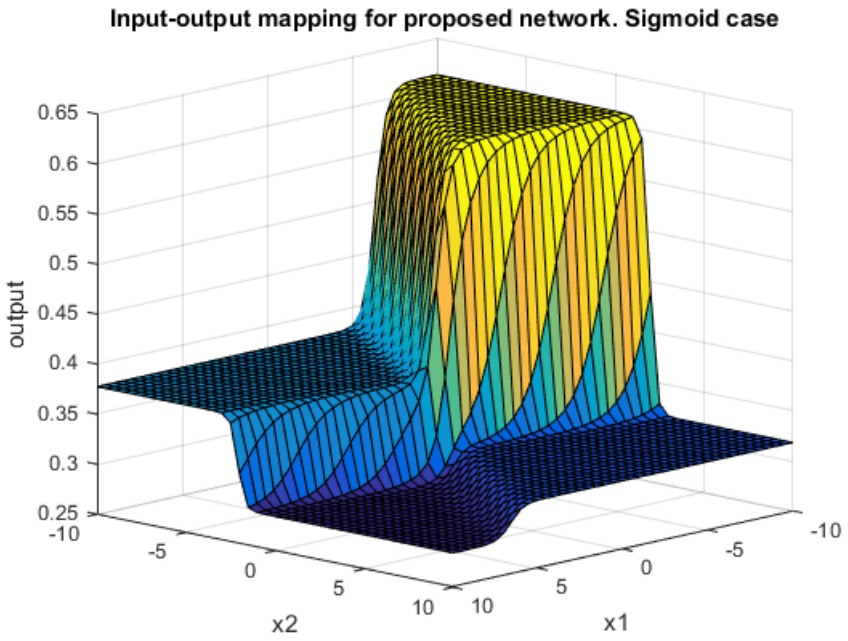
```

end
```

Mapping calculation and plotting.

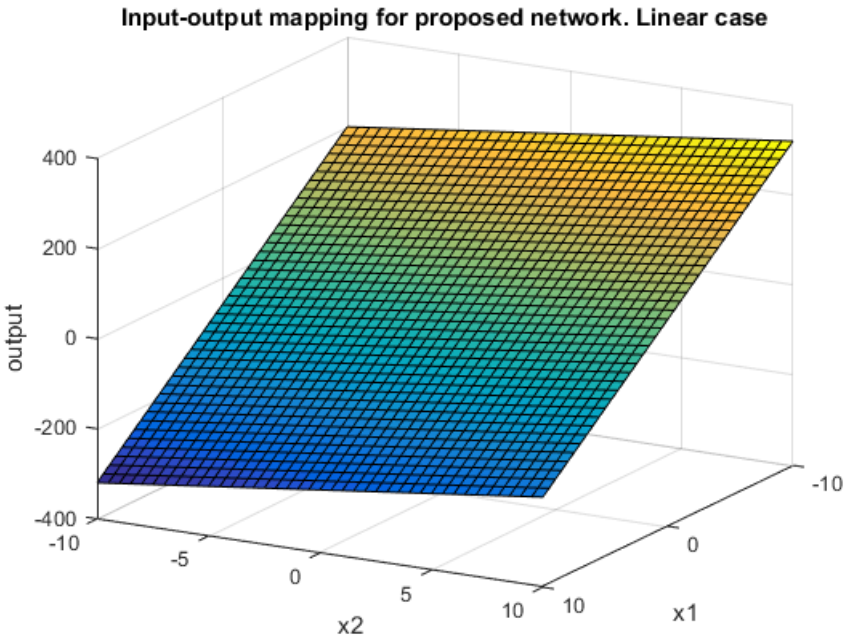
```
%% Exercise 1.13 from Haukin, Neural networks.
%Authors P.Lukin, I. Vishniakou, E. Ovchinnikova
clear all
close all

%a) Sigmoid function case
mapping = @(x1,x2) outputlayer(hiddenlayer(hiddenlayer([x1,x2],[5,1,2,-3],'sigmoid'),[3,-1,4,6],'sigmoid'),[-2,1],'sigmoid');
x1 = -10:0.5:10;
x2 = -10:0.5:10;
mappingPlot = zeros(length(x1),length(x2));
for i=1:length(x1)
    for j =1:length(x2)
        mappingPlot(i,j) = mapping(x1(i),x2(j));
    end
end
figure(1)
surf(x1,x2,mappingPlot)
xlabel('x1')
ylabel('x2')
zlabel('output')
title('Input-output mapping for proposed network. Sigmoid case')

%b) Linear function case
mapping = @(x1,x2) outputlayer(hiddenlayer(hiddenlayer([x1,x2],[5,1,2,-3],'linear'),[3,-1,4,6],'linear'),[-2,1],'linear');
for i=1:length(x1)
    for j =1:length(x2)
        mappingPlot(i,j) = mapping(x1(i),x2(j));
    end
end
figure(2)
surf(x1,x2,mappingPlot)
xlabel('x1')
ylabel('x2')
zlabel('output')
title('Input-output mapping for proposed network. Linear case')

%Binary classifier
mapping = @(x1,x2) outputlayer(hiddenlayer(hiddenlayer([x1,x2],[5,1,2,-3],'linear'),[3,-1,4,6],'linear'),[-2,1],'heaviside');
for i=1:length(x1)
```

```
40      for j =1:length(x2)
41          mappingPlot(i,j) = mapping(x1(i),x2(j));
42      end
43  end
44  figure(3)
45  surf(x1,x2,mappingPlot)
46  xlabel('x1')
47  ylabel('x2')
48  zlabel('output')
49  title('Input-output mapping for proposed network. Binary
      classification')
```

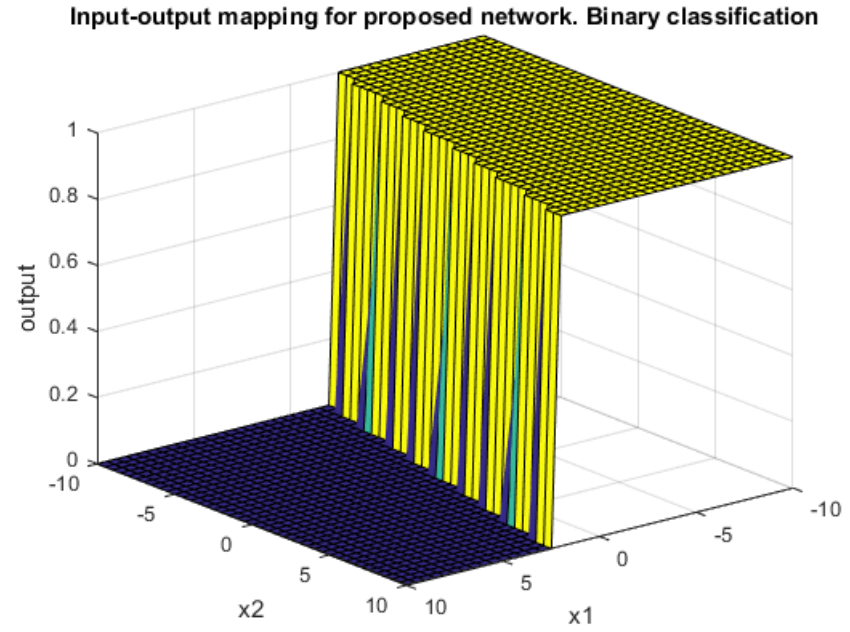
Plots of the input-output mappings.
Sigmoid squashing function:



Linear squashing function:



Binary classifier:



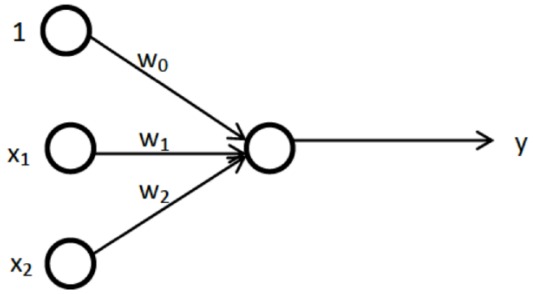
2.2 Exercise 4

Derive the Delta rule for two ADALINE neural networks depicted in Fig. 2, Fig. 3.
Solution:

1 neural network.

Input: $\vec{x} = (x_0, x_1, x_2)$, where $x_0 = 1$
Weights: $\vec{w} = (w_0, w_1, w_2)$

Figure 2: Neural network 1.



Output: $y = \vec{x}\vec{w} = w_0 + x_1 w_1 + x_2 w_2$

$e = d - y = d - (w_0 + x_1 w_1 + x_2 w_2)$, where d is a desired value.

$E(w) = \frac{1}{2} e^2$

Each i^{th} weight with an should be moved in the direction $\frac{\partial E}{\partial w_i}$:

$\frac{\partial E}{\partial w_i} = x_i(d - (w_0 + x_1 w_1 + x_2 w_2))$

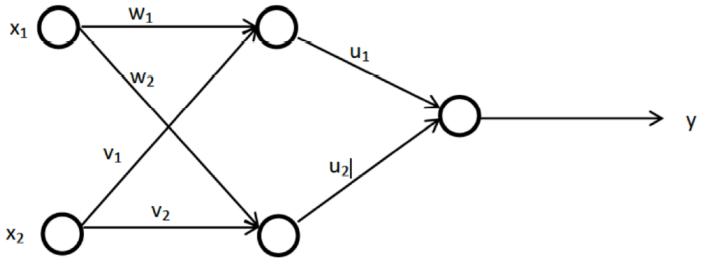
So, each step will change each weight by the following value:
$\Delta w_i = \eta x_i(d - (w_0 + x_1 w_1 + x_2 w_2))$,
where η is a learning rate that should be chosen according to the specific of the certain case. It can be either a constant (0.5, 1, etc.) or variable, or adaptive.

Therefore the delta rule will take the following form:

$w_i^{new} = w_i^{old} + \eta x_i(d - (w_0 + x_1 w_1 + x_2 w_2))$

Neural network 2.

Figure 3: Neural network 2.



Input: $\vec{x} = (x_1, x_2)$
Weights: $\vec{w} = (w_1, w_2)$, $\vec{v} = (v_1, v_2)$, $\vec{u} = (u_1, u_2)$
Activation function is linear, so:
Output: $y = u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2)$

$e = d - y = d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$.

Each i^{th} weight with an should be moved in the direction $\frac{\partial E}{\partial weight_i}$:

$\frac{\partial E}{\partial v_i} = x_1 u_i (d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$

$\frac{\partial E}{\partial w_i} = x_2 u_i (d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$

$\frac{\partial E}{\partial u_i} = (x_1 w_i + x_2 v_i)(d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$

So, each step will change each weight by the following value:

$\Delta w_i = \eta_w x_1 u_i (d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2)),$
$\Delta v_i = \eta_v x_2 u_i (d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$
$\Delta u_i = \eta_u (x_1 w_i + x_2 v_i)(d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$

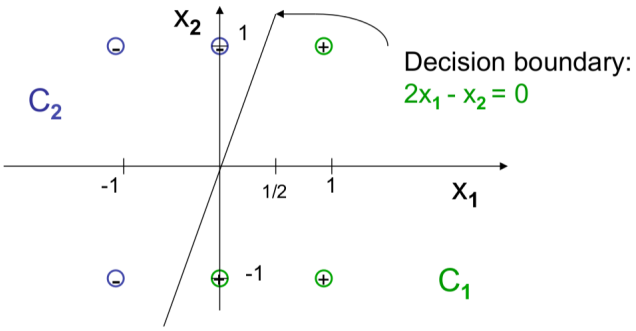
Therefore the delta rule will take the following form:

$w_i^{new} = w_i^{old} + \eta x_i (d - (w_0 + x_1 w_1 + x_2 w_2))$
$v_i^{new} = v_i^{old} + \eta_v x_2 u_i (d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$
$u_i^{new} = u_i^{old} + \eta_u (x_1 w_i + x_2 v_i)(d - (u_1(x_1 w_1 + x_2 v_1) + u_2(x_1 w_2 + x_2 v_2))$

2.3 New Classification Example (now with bias)

The original data was linearly separable without bias:

Figure 4: Original data and the decision line



To have the example actually use the bias the data can be modified: offset by (1,0). In this case the decision line would have to not go through the origin but the classes would remain linearly separable. To validate the provided percepton learning algorithm the bias is added to the inputs and they are introduced to the perceptron.

C1	C2
(2, 1)	(0, 1)
(2, -1)	(0, -1)
(1, -1)	(1, 1)

If 4 epoch the algorithm converges to the line $2x_1 - x_2 = 2$, the classes are separated, bias is nonzero. The learning iterations on the adjusted patterns with added biases would be as shown in tables 1-4.

Table 1: Epoch 1

Adjusted pattern	**Weight applied**	**w(n) x(n)**	**Update?**	**New weight**
(1, 2, 1)	(1, 0, 0)	1	No	(1, 0, 0)
(1, 2, -1)	(1, 0, 0)	1	No	(1, 0, 0)
(1, 1, -1)	(1, 0, 0)	1	No	(1, 0, 0)
(-1, 0, -1)	(1, 0, 0)	-1	Yes	(0, 0, -1)
(-1, 0, 1)	(0, 0, -1)	-1	Yes	(-1, 0, 0)
(-1, -1, -1)	(-1, 0, 0)	1	No	(-1, 0, 0)

Table 2: Epoch 2

Adjusted pattern	**Weight applied**	**w(n) x(n)**	**Update?**	**New weight**
(1, 2, 1)	(-1, 0, 0)	-1	Yes	(0, 2, 1)
(1, 2, -1)	(0, 2, 1)	3	No	(0, 2, 1)
(1, 1, -1)	(0, 2, 1)	1	No	(0, 2, 1)
(-1, 0, -1)	(0, 2, 1)	-1	Yes	(-1, 2, 0)
(-1, 0, 1)	(-1, 2, 0)	1	No	(-1, 2, 0)
(-1, -1, -1)	(-1, 2, 0)	-1	Yes	(-2, 1, -1)

Table 3: Epoch 3

Adjusted pattern	**Weight applied**	**w(n) x(n)**	**Update?**	**New weight**
(1, 2, 1)	(-2, 1, -1)	-1	Yes	(-1, 3, 0)
(1, 2, -1)	(-1, 3, 0)	5	No	(-1, 3, 0)
(1, 1, -1)	(-1, 3, 0)	2	No	(-1, 3, 0)
(-1, 0, -1)	(-1, 3, 0)	1	No	(-1, 3, 0)
(-1, 0, 1)	(-1, 3, 0)	1	No	(-1, 3, 0)
(-1, -1, -1)	(-1, 3, 0)	-2	Yes	(-2, 2, -1)

Table 4: Epoch 4

Adjusted pattern	**Weight applied**	**w(n) x(n)**	**Update?**	**New weight**
(1, 2, 1)	(-2, 2, -1)	1	No	(-2, 2, -1)
(1, 2, -1)	(-2, 2, -1)	3	No	(-2, 2, -1)
(1, 1, -1)	(-2, 2, -1)	1	No	(-2, 2, -1)
(-1, 0, -1)	(-2, 2, -1)	3	No	(-2, 2, -1)
(-1, 0, 1)	(-2, 2, -1)	1	No	(-2, 2, -1)
(-1, -1, -1)	(-2, 2, -1)	1	No	(-2, 2, -1)