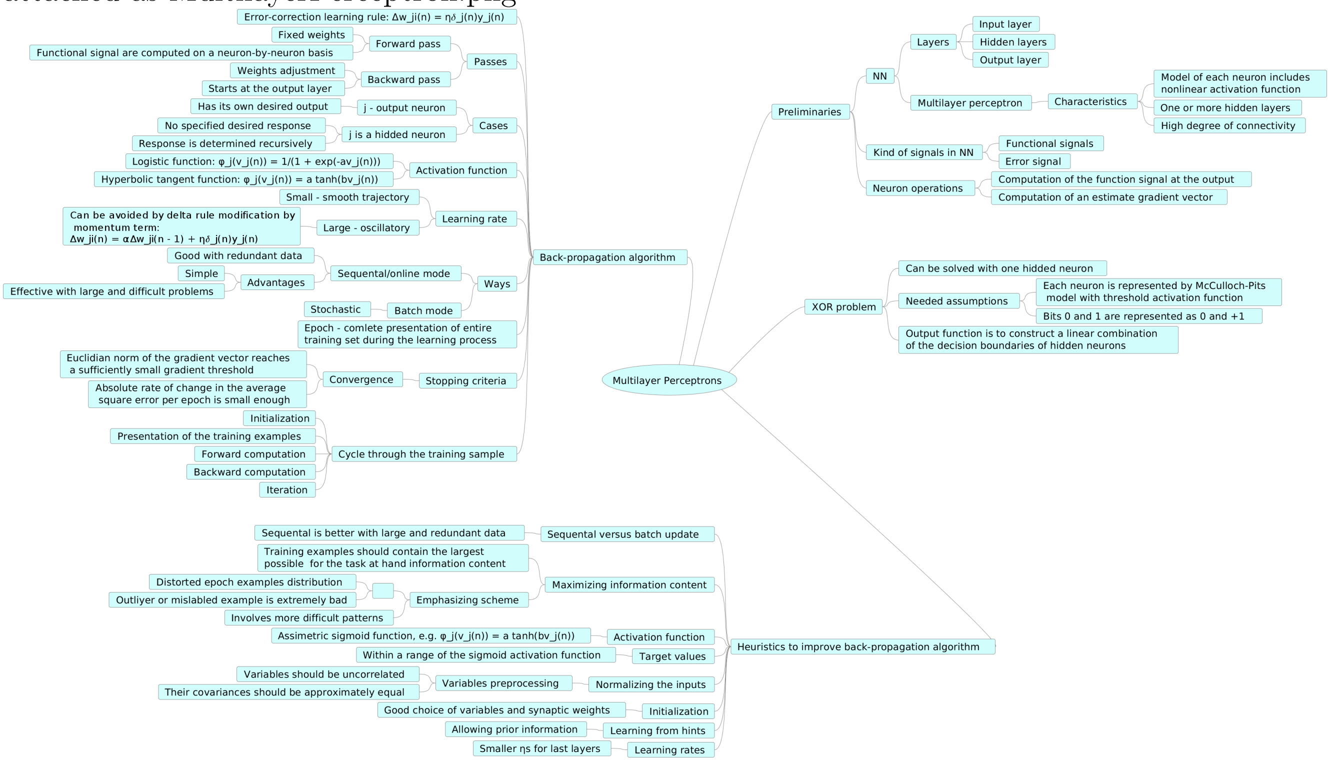
Neural Networks
- Homework 5 -

Petr Lukin, Evgeniya Ovchinnikova

Lecture date: 31 October 2016

1 Mind map

Figure 1: Mind map. Chapter 4 (first part) from Haykins book. A zoomed version is attached as MultilayerPerceptron.png



2 Exercises

2.1 Exercise 2

For this task you have to program the back-propogation (BP) for multi layered perceptron (MLP). Design your implementation for general NN with arbitrary many hidden layers. The test case is as follows: 2-2-1 multi layered perceptron depicted in Fig. 2 (MLP) with sigmoid activation function on XOR data that is shown in Fig.3.

a. Experiments with initial weights

Figure 2: 2-2-1 multi layered perceptron

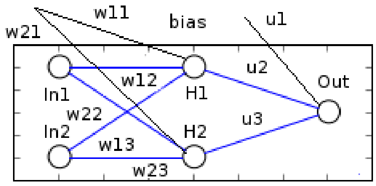
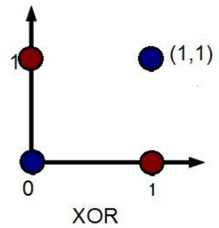


Figure 3: XOR values, blue - 0, red - 1



i. Train the network with zero initial weights i.e. $w_{ij} = 0$.

ii. Train with random initial weights

Compare and comment on the convergence.

b. Experiment with different learning rates e.g. 0.1, 0.3, 0.5, 0.9.

Compare the convergence and plot some resulting surfaces. You are not allowed to use any neural network toolbox for this solution.

NB: If you fail to implement the general case in order to get the full points it is sufficient to implement only the use case (2-2-1 MLP)

Solution:

We've used the following rule for weights update:

$\Delta w_{ij}(n) = \alpha \Delta w_{ij}(n-1) + (1-\alpha)\eta\delta_j o_i$,
where i, j are indices of neurons corresponding to the weights, δs are errors, o - output of the neuron, η - learning rate and α - a small constant.
$w_{ij}(n) = w_{ij}(n-1) + \Delta w_{ij}(n)$

It is important to mention that we've added a bias, because without it the results of learning we not satisfying.

We've used the following Python code:

```

import numpy as np
import matplotlib.pyplot as plt
```

```
%matplotlib inline

#XOR input and output, constants
bias = -1
x = np.array([[0.0,0.0,bias],[0.0,1.0,bias],[1.0,0.0,bias
    ],[1.0,1.0,bias]])
y = np.array([0.0,1.0,1.0,0.0])
eta1 = 0.1
eta2 = 0.3
eta3 = 0.5
eta4 = 0.9
alpha = 0.05

number_of_steps = 10000

#sigmoid function
def f(x):
    return 1/(1 + np.exp(-x))

def perceptron(w1, w2, u, eta):
    dw1 = np.array([0,0,0])
    dw2 = np.array([0,0,0])
    du = np.array([0,0,0])
    err = 111
    iter = 0

    while(abs(err)>0.02): # for zero weights - "for steps in
        range(number_of_steps)"
        out = []
        iter += 1
        for i in range(len(y)):
            o_in1 = f(np.dot(x[i], w1))
            o_in2 = f(np.dot(x[i], w2))
            intermediate_out = np.array([o_in1, o_in2, bias])
            o_out = f(np.dot(intermediate_out, u))

            delta_out = o_out * (y[i] - o_out) * (1 -o_out)
            err = y[i] - o_out
            delta_h1 = o_in1 * (1 - o_in1) * u[0] * delta_out
            delta_h2 = o_in2 * (1 - o_in2) * u[1] * delta_out
            for j in range(3):
                u[j] = alpha*du[j] + u[j] + (1 - alpha)*eta *
                    delta_out * intermediate_out[j]
                w1[j] = alpha*dw1[j] + w1[j] + (1 - alpha)*eta *
                    delta_h1 * x[i][j]
                w2[j] = alpha*dw2[j] + w2[j] + (1 - alpha)*eta *
                    delta_h2 * x[i][j]
                du[j] = alpha*du[j] + (1 - alpha)*eta *
```

```
                        delta_out * intermediate_out[j]
                dw1[j] = alpha*dw1[j] + (1 - alpha)*eta * delta_h1 * x[i][j]
                dw2[j] = alpha*dw2[j] + (1 - alpha)*eta * delta_h2 * x[i][j]
            out.append(o_out)
    output = ["%.3f" % element for element in out]
    print "iterations"
    print iter
    return output

# random initial weights, 3 - because of the bias

w1_rnd = np.array([np.random.random() for i in range(3)])
w2_rnd = np.array([np.random.random() for i in range(3)])
u_rnd = np.array([np.random.random() for i in range(3)])

print "eta1"
print perceptron(w1_rnd, w2_rnd, u_rnd, eta1)
print "eta2"
print perceptron(w1_rnd, w2_rnd, u_rnd, eta2)
print "eta3"
print perceptron(w1_rnd, w2_rnd, u_rnd, eta3)
print "eta4"
print perceptron(w1_rnd, w2_rnd, u_rnd, eta4)

# zero initial weights
w1_zero = np.array([0,0,0])
w2_zero = np.array([0,0,0])
u_zero = np.array([0,0,0])

print "eta1"
print perceptron(w1_zero, w2_zero, u_zero, eta1)
print "eta2"
print perceptron(w1_zero, w2_zero, u_zero, eta2)
print "eta3"
print perceptron(w1_zero, w2_zero, u_zero, eta3)
print "eta4"
print perceptron(w1_zero, w2_zero, u_zero, eta4)
```

The expected outputs for the following inputs [[0,0][0,1][1,0][1,1]] are [0, 1 ,1 ,0].
For zero initial weights we've obtained the following (with a fixed and large number of steps):

```
eta1
['0.500', '0.500', '0.500', '0.500']
eta2
['0.500', '0.500', '0.500', '0.500']
eta3
```

```
['0.500', '0.500', '0.500', '0.500']
eta4
['0.500', '0.500', '0.500', '0.500']
```

So, we can conclude that the network cannot be taught using zero initial weights. So, it can't converge.

For random weights we used ηs 0.1 (eta1), 0.3 (eta2), 0.5 (eta3), 0.9 (eta4) and 1.5. We obtained the following:

```
eta1
iterations
39710
['0.022', '0.981', '0.981', '0.020']
eta2
iterations
9
['0.022', '0.981', '0.981', '0.020']
eta3
iterations
6
['0.022', '0.981', '0.981', '0.020']
eta4
iterations
6
['0.022', '0.981', '0.981', '0.020']
eta4 = 1
iterations
7
['0.022', '0.981', '0.981', '0.020']
```

We can see that with increase of η the learning is better and converges faster till certain value of η (around 0.8-0.9 here), but after it learns more slow.

We have created a net of points from (0,0) to (1,1) and applied the perceptron to these points. For that we've modified the code in the following way:

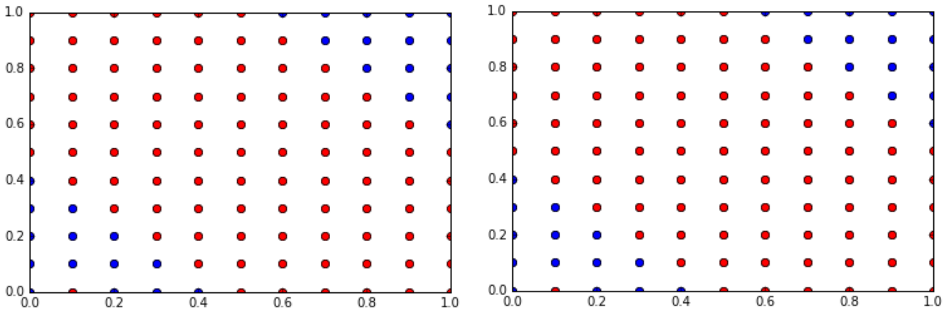
```
def classify(x,w1,w2,u):
    x = np.append(x,-1)
    o_in1 = f(np.dot(x, w1))
    o_in2 = f(np.dot(x, w2))
    intermediate_out = np.array([o_in1, o_in2, -1])
    o_out = f(np.dot(intermediate_out, u))
    return o_out

from matplotlib import pyplot
from math import cos, sin, atan

%matplotlib inline

```

```
points_to_classify = np.zeros([11,11,2])

for i in range(11):
    for j in range(11):
        points_to_classify[i][j][0] = i / 10.0
        points_to_classify[i][j][1] = j / 10.0

points_to_draw = np.zeros([11,11])


# random initial weights, 3 - because of the bias

w1_rnd = np.array([np.random.random() for i in range(3)])
w2_rnd = np.array([np.random.random() for i in range(3)])
u_rnd = np.array([np.random.random() for i in range(3)])

print "eta1"
print perceptron(w1_rnd, w2_rnd, u_rnd, eta1)[0]
weights = perceptron(w1_rnd, w2_rnd, u_rnd, eta1)[1]
for i in range(11):
    for j in range(11):
        if classify(points_to_classify[i][j],weights[0],weights[1],weights[2]) > 0.5:
            ones_arr = np.column_stack((ones_arr, points_to_classify[i][j]))
        else:
            zeros_arr = np.column_stack((zeros_arr, points_to_classify[i][j]))
plt.plot(zeros_arr[0], zeros_arr[1], 'bo', ones_arr[0], ones_arr[1], 'ro')
plt.show()

print "weights"
print weights

print "eta3"
weights = perceptron(w1_rnd, w2_rnd, u_rnd, eta3)[1]
for i in range(10):
    for j in range(10):
        if classify(points_to_classify[i][j],weights[0],weights[1],weights[2]) > 0.5:
            ones_arr = np.column_stack((ones_arr, points_to_classify[i][j]))
        else:
            zeros_arr = np.column_stack((zeros_arr, points_to_classify[i][j]))
plt.plot(zeros_arr[0], zeros_arr[1], 'bo', ones_arr[0], ones_arr[1], 'ro')
```

```
plt.show()

print "weights"
print weights
```

The results are depicted in Fig. 4. We've plotted two point net classifications: for $\eta = 0.1$ and $\eta = 0.5$ with the following weights: [[w11 w12 w13][w21 w22 w23][u1 u2 u3]] = [[6.3716883 6.36724135 2.81390949] [4.52244586 4.52120091 6.93846284] [9.31095822 -9.98278592 4.30383744]] for $\eta = 0.1$ and [[6.37216319 6.36771883 2.81420294] [4.52300285 4.52175749 6.93930633] [9.31245678 -9.98430441 4.30460043]] for $\eta = 0.5$.

Figure 4: Classified points net, blue - 0, red - 1



2.2 Exercise 3

Investigate the use of back-propagation learning using a sigmoidal nonlinearity to achieve one-toone mappings, as described here:

1. $F(x) = 1/x1 <= x <= 100$
2. $F(x) = log10(x)1 <= x <= 10$
3. $F(x) = exp(-x)1 <= x <= 10$
4. $F(x) = sin(x)0 <= x <= pi/2$

For each mapping, do the following:

- Set up two sets of data, one for network training, and the other for testing.
- Use the training data set to compute the synaptic weights of the network, assumed to have a single hidden layer.
- Evaluate the computation accuracy of the network by using the test data. Use a single hidden layer but with a variable number of hidden neurons. Investigate how the network performance is affected by varying the size of the hidden layer.

You can use any neural network toolbox (MATLAB or python or ...) for solving this problem or you can use your own implementation of MLP from previous question.

2.3 Solution

MLP backpropagation algorithm was implemented in MATLAB and has 2 functions:

- MLP evaluation;
- MLP backpropagation.

The algorithm works for MLP with any number of hidden layers and sigmoid activation function.

MLP evaluation:

```
function [ out ] = fmlp( input , w )

nlayers = length(w);
out = cell(nlayers ,1);
squash = @(v) exp(v)./(1+exp(v));

for j=1:nlayers
    out{j} = w{j}*input;
    for i=1:length(out{j})
        out{j}(i) = squash(out{j}(i));
    end
    input = out{j};

end
end
```

Training MLP by backpropagation.

```
function [ w] = trainMLP( structure ,x,t,winit,eta )
%Init weights
w = cell(length(structure)-1,1);
delta = cell(length(structure)-1,1);
dw = cell(length(structure)-1,1);

if winit==0
    for i =2:length(structure)
        w{i-1} = zeros(structure(i),structure(i-1));
    end
else
    for i =2:length(structure)
        w{i-1} = 0.5*rand(structure(i),structure(i-1));
    end
end
% Iterate training examples
for i=1:length(x)
    o = fmlp(x(:,i),w);

    %Delta calculation
    for k =1:structure(end)
      delta{end}(k) = -2*o{end}(k)*(1-o{end}(k))*(t(k,i)-o{end}(k));
    end
    for j=length(delta)-1:-1:1
        for k =1:structure(j)
          delta{j}(k) = 2*o{j}(k)*(1-o{j}(k))*sum(delta{j+1}*w{j}(:,k));
        end
    end
    %dw calculation
    for j=1:length(w)
        if j==1
            dw{j} = -eta*bsxfun(@times,delta{j},x(:,i));
        else
            dw{j} = -eta*bsxfun(@times,delta{j},o{j-1});
        end
        %Weight update
        w{j} = w{j}+transpose(dw{j});
    end

end

end
```

In the exercise, SISO functions were modelled with a single hidden layer. Initial weights were taken as uniformly distributed numbers in $[0, 1]$. Training and testing sets were created to verify MLP performance. Matlab script with experiment setup:

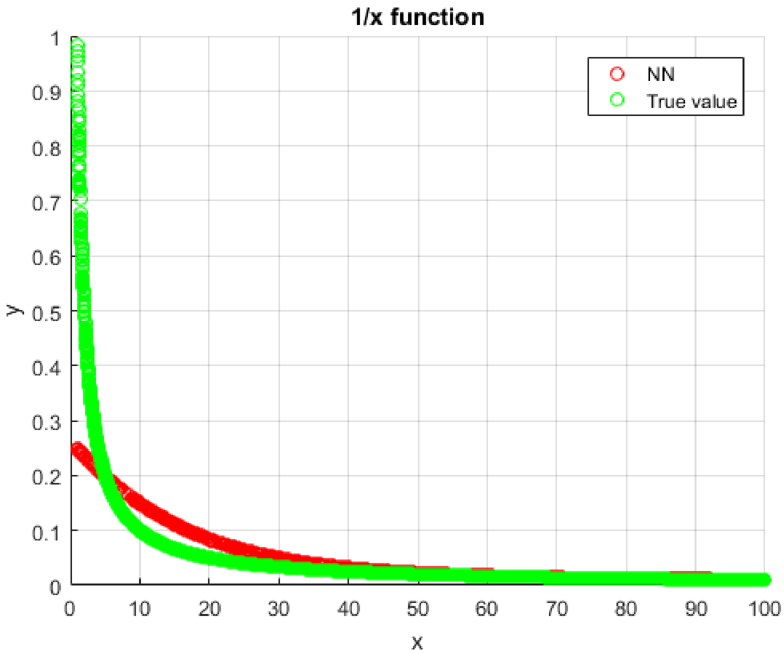
```
%% Exercise 3 Neural networks.
%Authors P.Lukin, I. Vishniakou, E. Ovchinnikova
clc;
clear all;
close all;

% %1/x test

structure = [1,5,1];
x = 1+99*rand(1,10000);
t = 1./x;

xtest = 1+99*rand(1,1000);
ttest = 1./xtest;

eta = 0.5;
w = trainMLP(structure,x,t,1,eta );
%Error
for i =1:length(xtest)
     yi = fmlp(xtest(:,i),w);
     y(i) = yi{end};
end
err = sum(abs(y-ttest))/length(xtest)
figure(1)
hold on
plot(xtest,y,'ro')
plot(x,t,'go')
grid on
xlabel('x')
ylabel('y')
legend('NN','True value')
title('1/x function')
hold off


% log10 test

structure = [1,5,1];
x = 1+9*rand(1,10000);
t = log10(x);

xtest = 1+9*rand(1,1000);
ttest = log10(xtest);

eta = 0.5;
w = trainMLP(structure,x,t,1,eta );
```

```
%Error
for i =1:length(xtest)
yi = fmlp(xtest(:,i),w);
y(i) = yi{end};
end
err = sum(abs(y-ttest))/length(xtest)

figure(2)
hold on
plot(xtest,y,'ro')
plot(x,t,'go')
grid on
xlabel('x')
ylabel('y')
legend('NN','True value')
title('log(10) function')
hold off


%exp(-x) test

structure = [1,5,1];
x = 1+9*rand(1,10000);
t = exp(-x);

xtest = 1+9*rand(1,1000);
ttest = exp(-xtest);

eta = 0.5;
w = trainMLP(structure,x,t,1,eta );
% Error
for i =1:length(xtest)
yi = fmlp(xtest(:,i),w);
y(i) = yi{end};
end
err = sum(abs(y-ttest))/length(xtest)
figure(3)
hold on
plot(xtest,y,'ro')
plot(x,t,'go')
grid on
xlabel('x')
ylabel('y')
legend('NN','True value')
title('exp(-x) function')
hold off

% sin(x) test
```

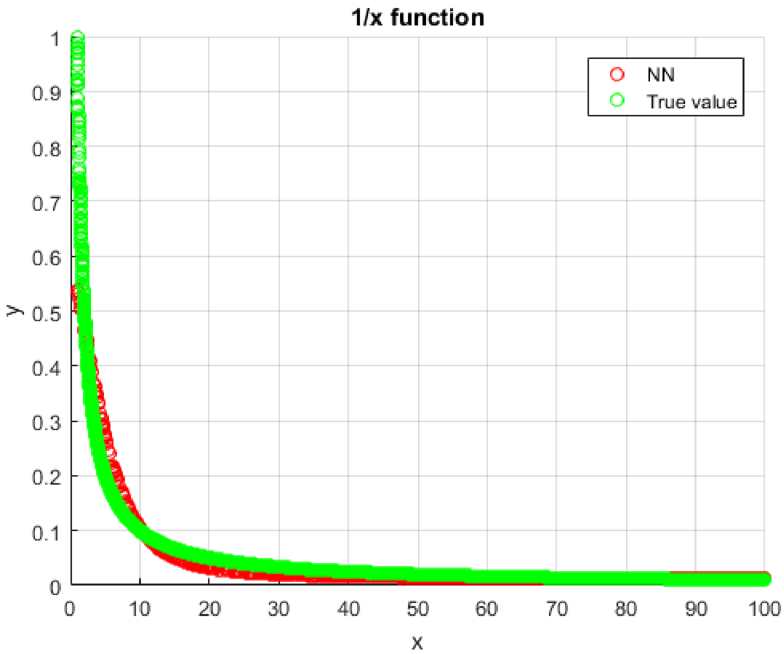
```
structure = [1,10,1];
x = pi/2*rand(1,10000);
t = sin(x);

xtest = pi/2*rand(1,1000);
ttest = sin(xtest);

eta = 0.5;
w = trainMLP(structure,x,t,1,eta );
%Error
for i =1:length(xtest)
yi = fmlp(xtest(:,i),w);
y(i) = yi{end};
end
err = sum(abs(y-ttest))/length(xtest)

figure(4)
hold on
plot(xtest,y,'ro')
plot(x,t,'go')
grid on
xlabel('x')
ylabel('y')
legend('NN','True value')
title('sin(x) function')
hold off
```

Results of the experiments:

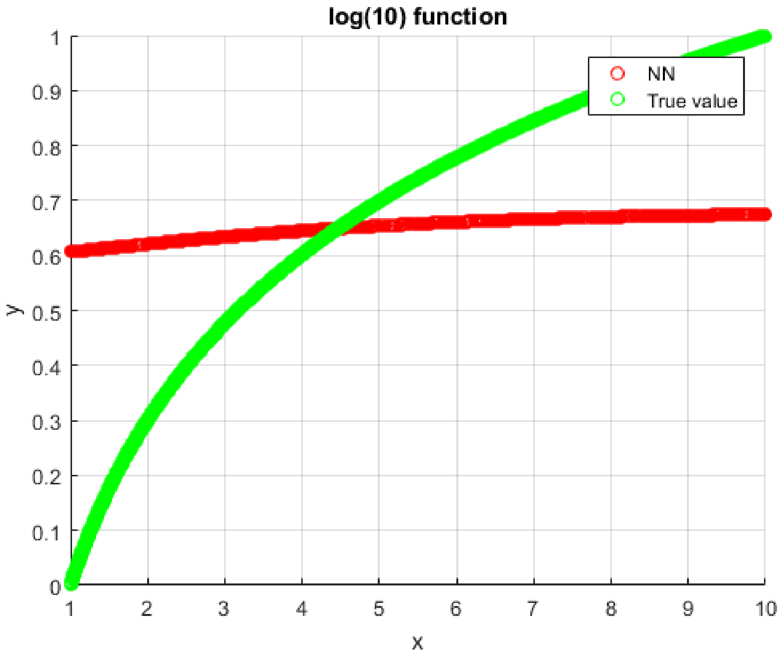
1. $1/x$ function with 2 neurons in hidden layer.



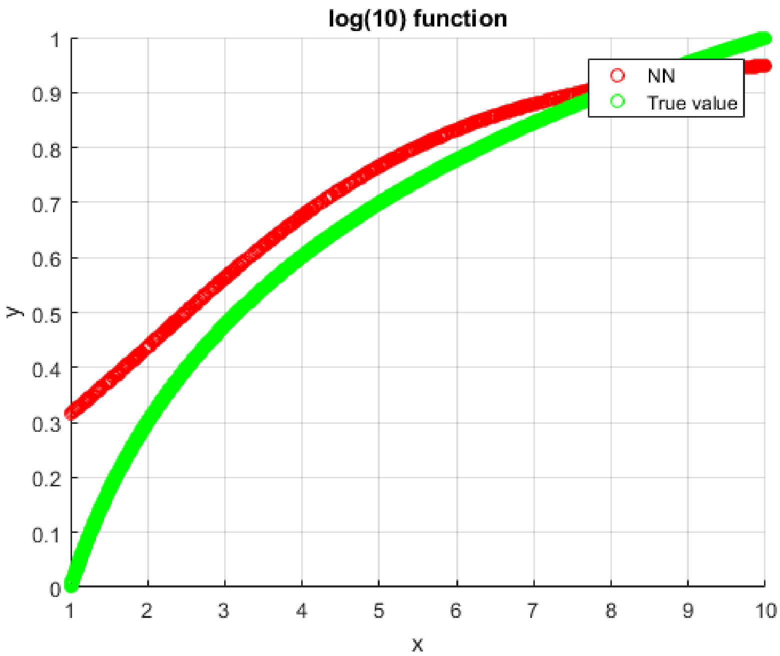
2. $1/x$ function with 5 neurons in hidden layer.



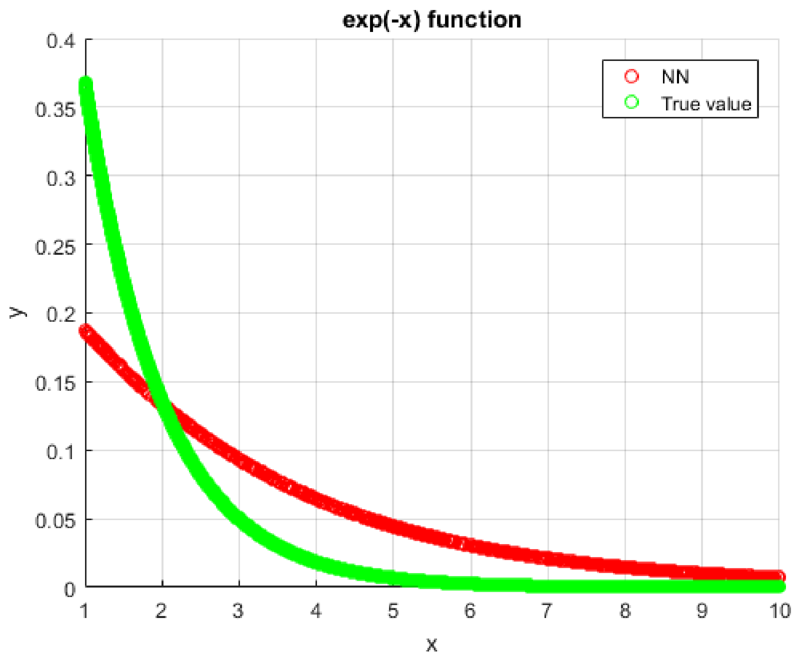
3. log_{10} function with 2 neurons in hidden layer.



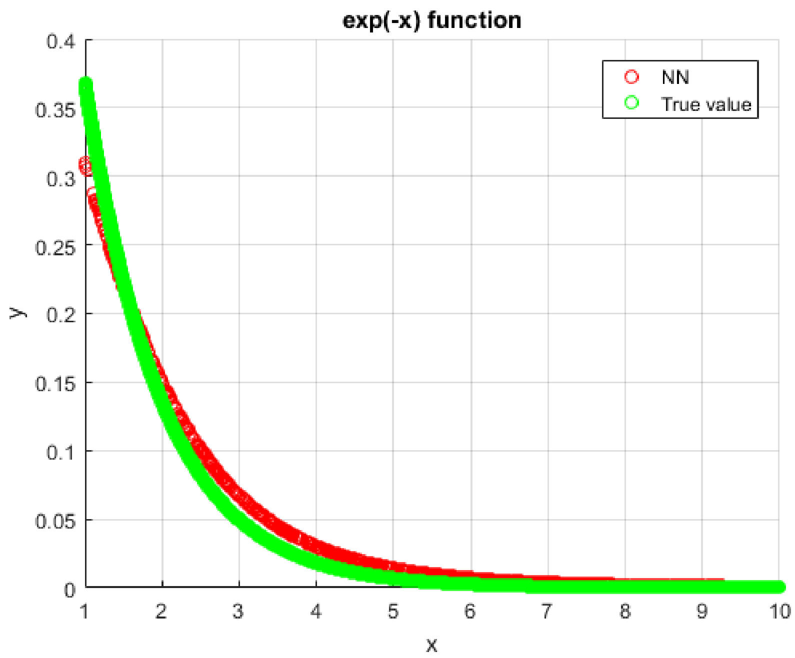
4. log_{10} function with 5 neurons in hidden layer.



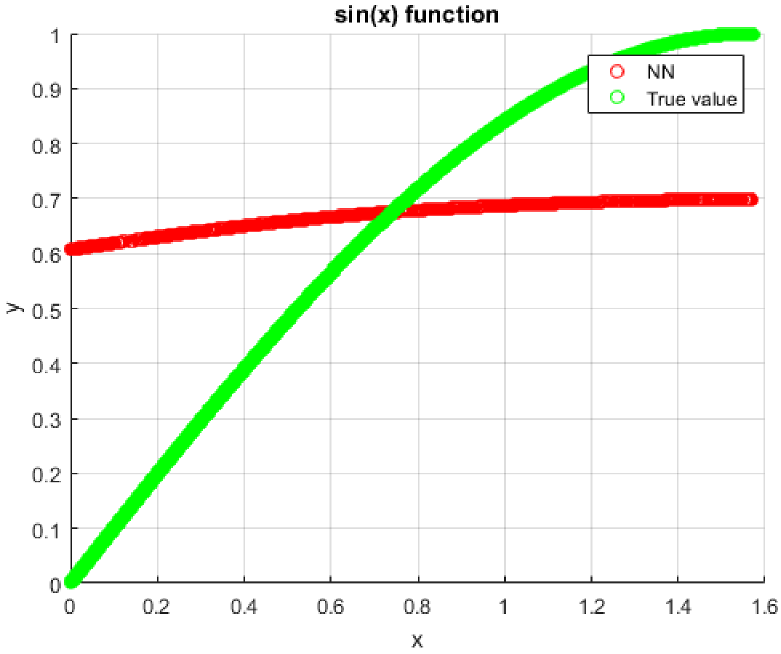
5. $\exp(-x)$ function with 2 neurons in hidden layer.



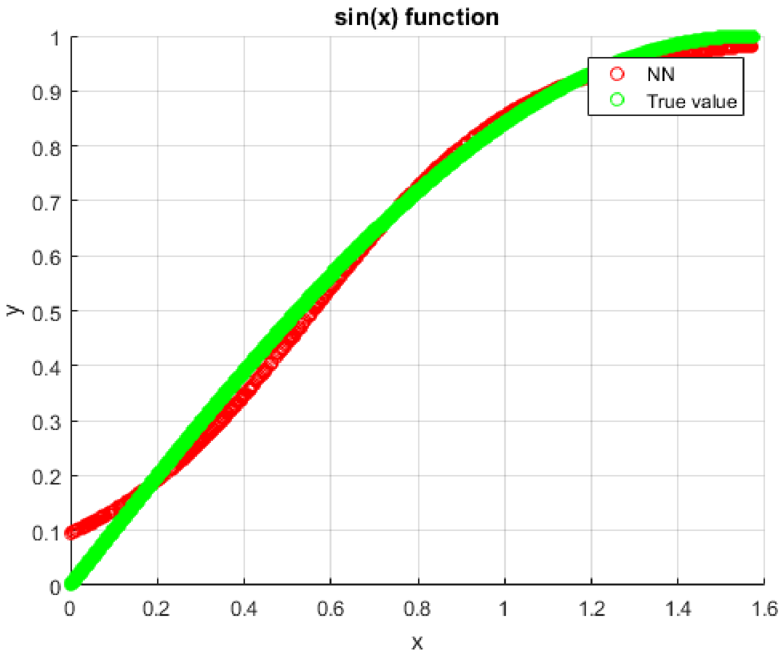
6. $\exp(-x)$ function with 5 neurons in hidden layer.



7. $\sin(x)$ function with 2 neurons in hidden layer.



8. $\sin(x)$ function with 5 neurons in hidden layer.



It can be seen, that extra hidden layers increase precision of the MLP. However, MLP is still sensitive to initial weights and parameter η.