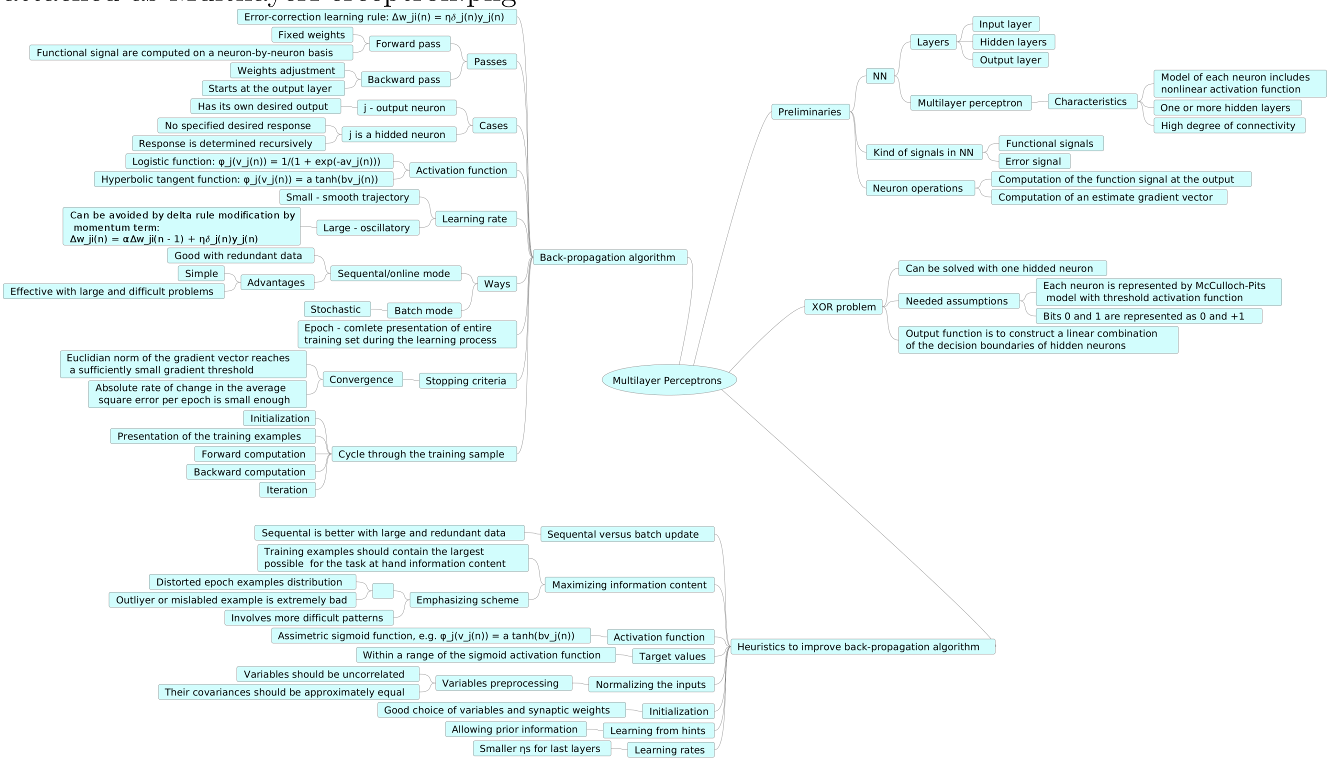
Neural Networks
- Homework 5 -

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. **??** (MLP) with sigmoid activation function on XOR data that is shown in Fig.**??**.

a. Experiments with initial weights

Figure 2: 2-2-1 multi layered perceptron

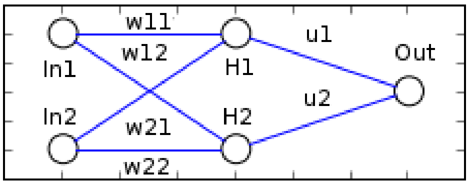
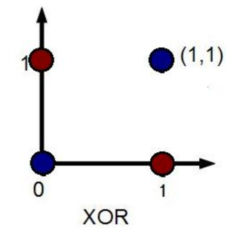


Figure 3: XOR values, blue - 0, red - 1



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 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)
```

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 then it learns more slow.

We have created a net of point from (0,0) to (1,1) and applied the perceptron to these points. The result is in Fig.