ECE 629: Introduction to Neural Networks
Homework 2

September 3, 2018

Julian Büchel

1

1.1 Gradient vector of $E(w)$

$\nabla E(w) = \begin{pmatrix} \frac{\partial E(w)}{\partial w_1} \\ \frac{\partial E(w)}{\partial w_2} \end{pmatrix}$ With $E(w) = 0.4[(2-w_1)^2 + 3(w_1-w_2)^2]$ this yields

$$\nabla E(w) = \begin{pmatrix} 3.2w_1 - 2.4w_2 - 1.6 \\ -2.4(w_1 - w_2) \end{pmatrix}$$

1.2 Optimal vector W_0

Setting $\nabla E(w) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ yields $w_1 = w_2$ from the second equation and inserting this into the first equation gives us

$$3.2w_1 - 2.4w_1 = 1.6 \rightarrow w_1 = 2 \rightarrow w_2 = w_1 = 2$$

2 Two sweeps

```
x = [1 2 1 -1;1 1 -2 1]'; %Include leading 1 for b
d = [1 -1];
w = [0.2 0.4 -0.1 0.3]; %Inlcude b (0.2) as the first entry
alpha = 0.2;
delta = zeros(1,size(w,2));
g = @(h) 2./(1+exp(-h))-1; %Definition of bipolar sigmoid
dg = @(h) 0.5*(1+g(h))*(1-g(h)); %Derivative of bipolar sigmoid
for j=1:2
    for i=1:size(x,2)
        %Compute delta w(i)
```

```
        h = w*x(:,i); %Compute the weighted sum
        y = g(h); %Compute the output of the network
        delta = (alpha*((d(i)-y)*dg(h)*x(:,i)))'; %Compute delta w
        w = w + delta %Update the weight vectors
        abs(y-d(i));
    end
end
y = g(w*x)
```

The first entry in the weight vector is b.

1. Sweep weight vectors:

$$w = 0.2649\ 0.5297\ -0.0351\ 0.2351]$$

$$w = [0.1524\ 0.4172\ 0.1898\ 0.1227]$$

2. Sweep weight vectors:

$$w = [0.1920\ 0.4965\ 0.2294\ 0.0831]$$

$$w = [0.0793\ 0.3838\ 0.4549\ -0.0297]$$

3 Range of learning rate for LMS

According to the LMS algorithm we update the weights according to

$$w(n+1) = w(n) + \alpha[d(n) - w(n)^T x(n)]x(n)$$

$$= w(n) + \alpha d(n)x(n) - x(n)x(n)^T w(n)$$

$$= \alpha(d(n)x(n)) + [I - \alpha x(n)x(n)^T]w(n)$$

The expected weight $w(n+1)$ is:

$$E(w(n+1)) = (I - \alpha R_x)E(w(n)) + \alpha r_{xd}$$

with

$$R_x = E(x(n)x(n)^T)$$

and

$$r_{xd} = E(x(n)d(n))$$

Since the optimal weight vector satisfies the Wiener-Hopf equation $R_x w_0 = r_{xd}$ we get

$$E(w(n+1)) = (I - \alpha QAQ^T)E(w(n)) + \alpha QAQ^T w_0$$

with $R_x = QAQ^T$ (Q orthogonal so $Q^T = Q^{-1}$). Multiplying by Q^T from the left yields:

$$Q^T E(w(n+1)) = (I - \alpha A)Q^T E(w(n)) + \alpha AQ^T w_0)\ (1)$$

Substituting $k(n) = Q^T(E(w(n)) - w_0)$ gives

$$E(w(n)) = Qk(n) + w_0$$

inserting into (1) gives:

$$k(n+1) = (I - \alpha A)k(n)$$

. Looking at the single values in the vector we get

$$k_k(n+1) = (1 - \alpha\lambda_k)k_k(n)$$

we can write this recursively starting from $k_k(0)$ as

$$k_k(n) = (1 - \alpha\lambda_k)^n k_k(0)$$

and in order for this to not diverge we need $|1 - \alpha\lambda_k| < 1$ and this leads us to the conclusion that $0 < \alpha < 2/\lambda_{max}$.

3.1 Show that for normalized input: $0 < \alpha < 2$ for convergence

We build on top of the previous exercise, by proving that the Eigenvalues all lie within the unit circle, so that $0 < \alpha < 2$ holds for convergence. Let us call

$$B = E(\frac{x(n)x(n)^T}{||x||^2})$$

We previously had $R_x = E(x(n)x(n)^T)$. Now let us assume that $[I - \alpha B]$ has Eigenvalue β and Eigenvector b. We the know that $b^T[I - \alpha B]b = \beta b^T b$ from the definition of Eigenvectors.
From the definition of B and the Cauchy Schwarty inequality (holds for x normalized), we then conclude that

$$0 < b^T Bb < b^T b$$

In order to prove convergence for $0 < \alpha < 2$ we now assume $0 < \alpha < 2$. With $0 < \alpha < 2$, $0 < b^T Bb < b^T b$ and $\alpha b^T Bb = (1 - \beta)b^T b$ we know that

$$0 < 1 - \beta < \alpha < 2$$

so we can conclude that $|\beta| < 1$. So the Eigenvalue lies within the unit circle. We can then conclude that from the previous exercise, we have convergence for $0 < \alpha < 2/\lambda_{max}$, which yields convergence for $0 < \alpha < 2$,because all Eigenvalues are in the unit circle.

3.2 Derive recursion formula for the weights

We begin by writing the error function w.r.t. $w_0, w_1, ..., w_{p-1}$:

$$\epsilon(w_0..w_{p-1}) = \sum_{i=1}^{n} e(i)^2 = \sum_{k=1}^{p-1} w_k(n)u(i-k)$$

Which can be rewritten as:

$$w(n) = (\sum_{i=1}^{n} u(i)u(i)^T)^{-1}(\sum_{i=1}^{n} u(i)d(i)) = \sigma(n)^{-1}\theta(n)$$

with $\sigma(n) = \sum_{i=1}^{n} u(i)u(i)^T$ and $\theta(n) = \sum_{i=1}^{n} u(i)d(i)$
So $w(n) = \sigma(n)^{-1}\theta(n)$. In recursion, we base our computatoon on the previous mcomputation $w(n-1)$:

$$w(n-1) = \sigma(n-1)^{-1}\theta(n-1)$$

We rewrite the $\sigma(n)$ and $\theta(n)$ in terms of $\sigma(n-1)$ and $\theta(n-1)$:

$$\sigma(n) = \sum_{i=1}^{n-1} u(i)u(i)^T + u(n)u(n)^T = \sigma(n-1) + u(n)u(n)^T$$

$$\theta(n) = \sum_{i=1}^{n-1} u(i)d(i) + u(n)d(n) = \theta(n-1) + u(n)d(n)$$

Because for a matrix A and B (p.s.d) we have that $A = B^{-1} + CD^{-1}C^T$ the inverse $A^{-1} = B - BC(D + C^TBC)^{-1}C^TB$ we see that $\sigma(n) = \sigma(n-1) + u(n)u(n)^T$, which corresponds to $D^{-1} = I, C = u(n), B^{-1} = \sigma(n-1)$ has the following inverse:

$$\sigma(n)^{-1} = \sigma(n-1)^{-1}\frac{\sigma(n-1)^{-1}u(n)u(n)^T\sigma(n-1)^{-1}}{1 + u(n)^T\sigma(n-1)^{-1}u(n)}$$

with $P(n) = \sigma(n)^{-1}$ we have that

$$k(n) = \frac{P(n-1)u(n)}{1 + u(n)^T P(n-1)u(n)} = P(n)u(n)$$

We get $P(n) = P(n-1) - k(n)u(n)^T P(n-1)$. By constantly inserting and expanding the forumla $w(n) = \sigma(n)^{-1}\theta(n)$ that

$$w(n) = w(n-1) + P(n)u(n)\alpha(n) = w(n-1) + k(n)\alpha(n)$$

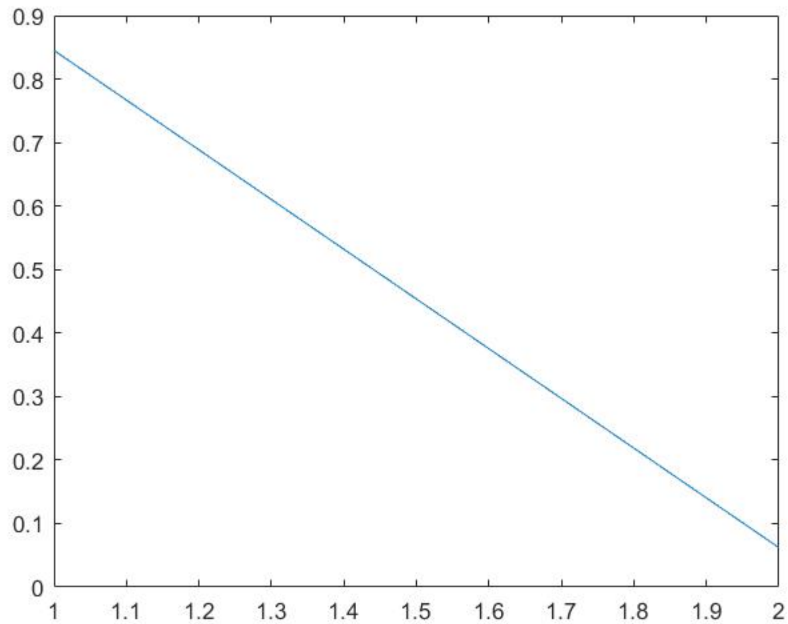
with $\alpha(n) = d(n) - u(n)^T w(n-1)$ which is the recursion formula.

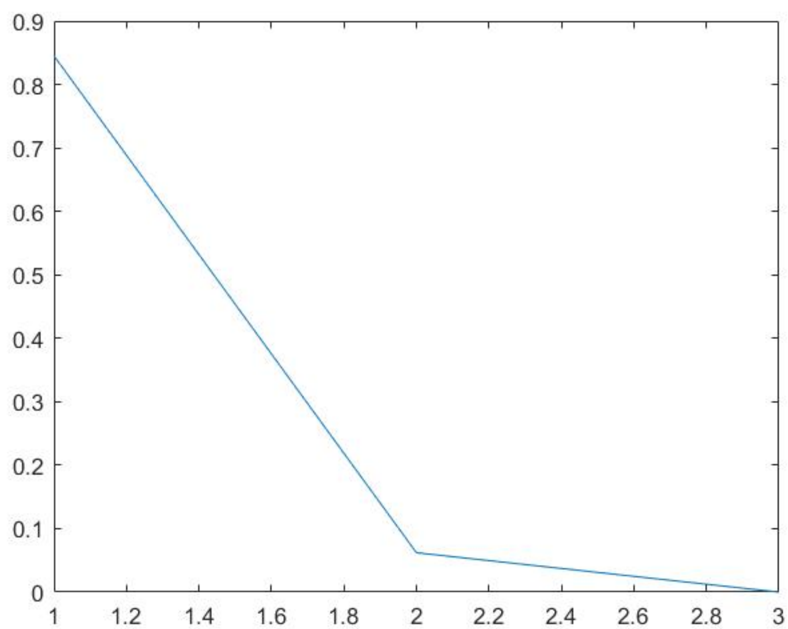
4 Matlab

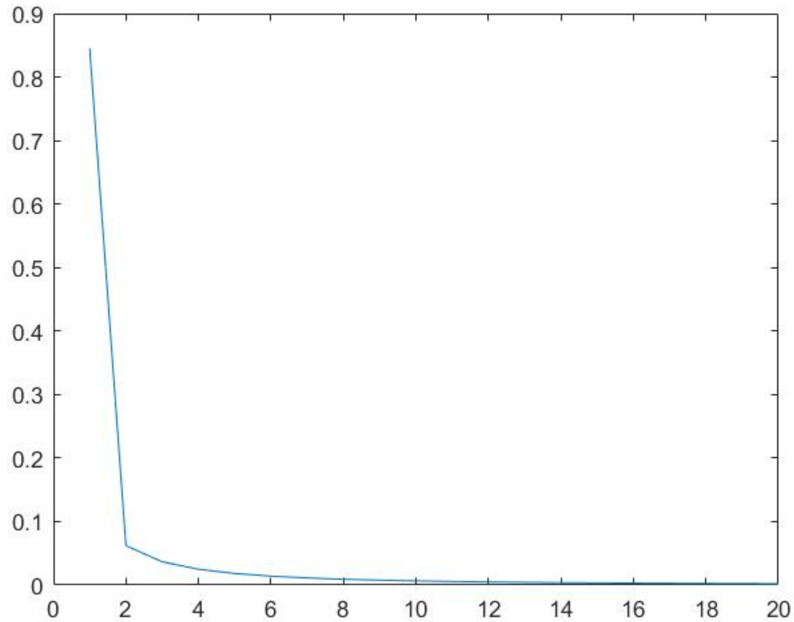
4.1 Run the program

The output is: $dw = [-0.1752\ 0.3504\ \ -0.1752]$ $db = 0.1752$ $w = 0.1800\ 0.7522\ \ -0.1673$ $b = 0.502$ and $y = 0.9449\ \ -0.7571$

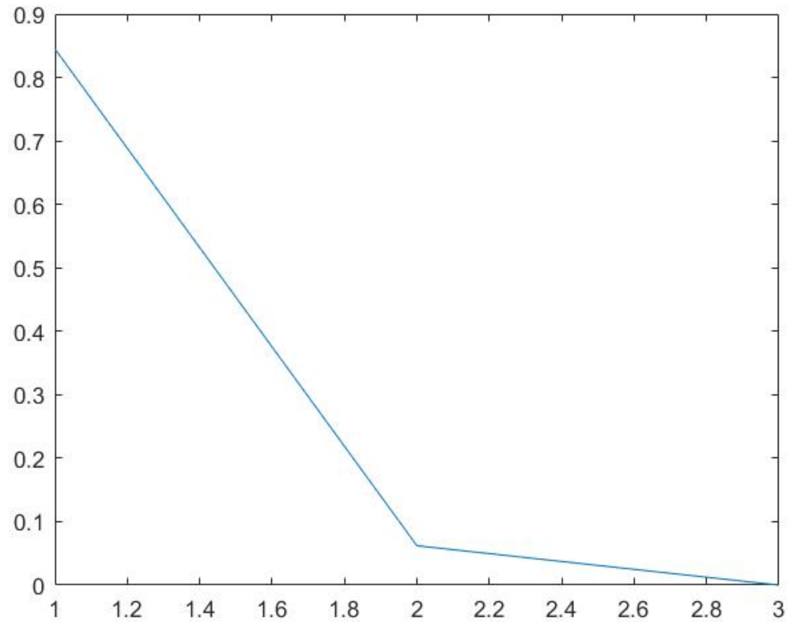
4.2 Plots for different N (x-axis)



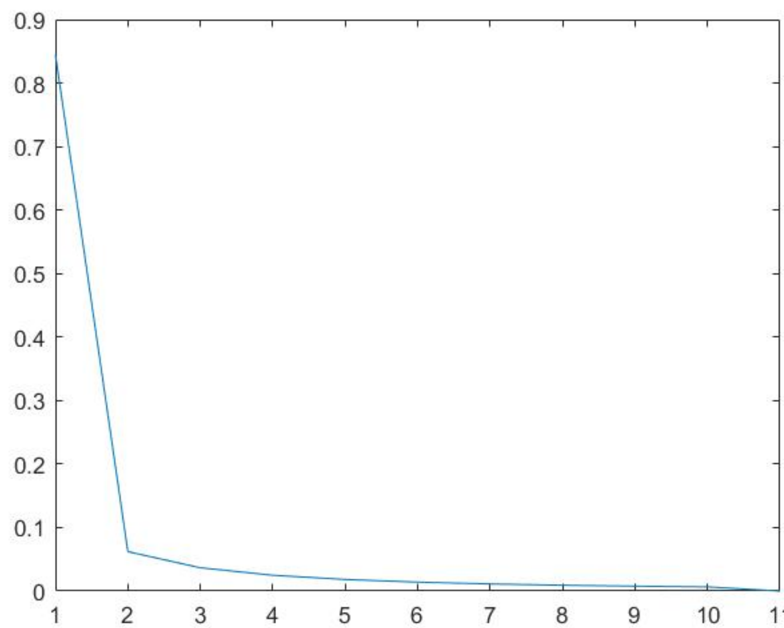



0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 2 4 6 8 10 12 14 16 18 20

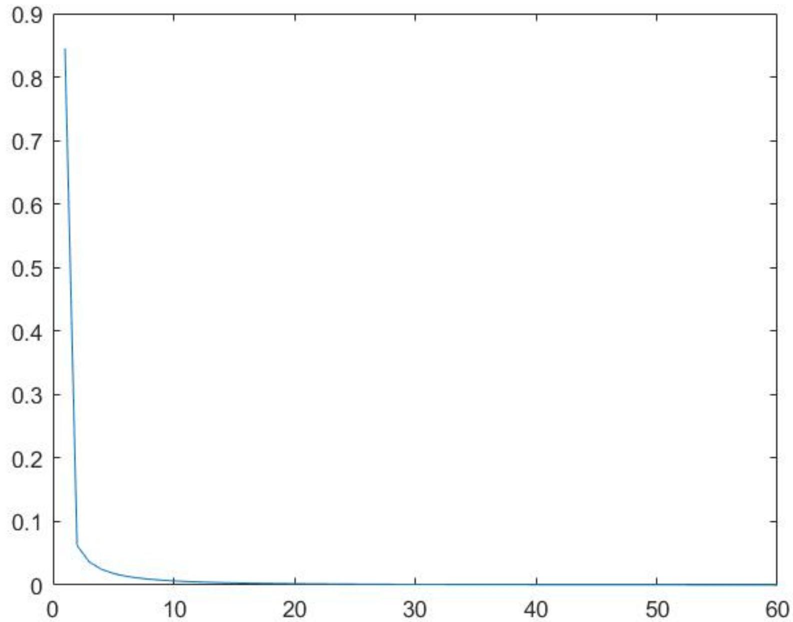
4.3 Different tolerance



Tolerance: 0.1



Tolerance: 0.001



Tolerance:0.000001

5 Translated to Python

```
In [59]: import numpy as np
         import matplotlib.pyplot as plt
```

```
In [53]: x = np.array([[2,1],
                       [1,-2],
                       [-1,1]])
```

```
In [54]: d = np.array([1,-1])
         w = np.array([0.4, -0.1, 0.3])
         b = 0.2
         a = 0.2
         n = 20
         f_x = np.zeros([1,n])
         last = 0
         tol = 0.0001
         stopped_at = n
```

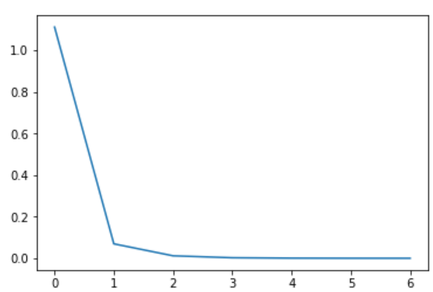
```
In [55]: x[:,j].transpose()

Out[55]: array([ 1, -2,  1])
```

```
In [56]: for i in range(0,n-1):
             for j in range(0,len(x)-1):
                 y = np.tanh(w.dot(x[:,1]) + b)
                 z = (d[j] - y)*(1-np.tanh(y)**2)
                 dw = a*z*(x[:,j].transpose())
                 db = -a*z;
                 w = w + dw;
                 b = b + db;

             y = np.tanh(np.matmul(w,x) + b);
             curr = np.linalg.norm(y-d)**2

             if abs(last-curr) < tol:
                 print(abs(last-curr))
                 stopped_at = i
                 print(stopped_at)
                 break
             last = curr
             f_x[:,i] = np.linalg.norm(y-d)**2

         2.91210769351e-05
         7
```

```
In [57]: print(w)

         [ 2.65585196  1.88836163 -1.34418737]
```

```
In [58]: print(y)

         [ 0.99999966 -0.99681798]
```

```
In [71]: t = np.arange(stopped_at)
         plt.plot(t,np.transpose(f_x[:,0:stopped_at]))
         plt.show()
```



The program are not the same. The python version is doing better. (Why?).

6 LMS Algorithm

6.1 Commented Matlab code

```
clear all
close all
hold off
                                                    %channel system order
sysorder = 5 ;
                                                    %Number of system points
N=2000;
inp = randn(N,1);
n = randn(N,1);
[b,a] = butter(2,0.25);                             %Create a low pass filter of second order with cutoff frequency 0.25
Gz = tf(b,a,.1);                                    %Create discrete-time transfer function with undetermined sample time
                                                    %and numerator b and a, which are the transfer function coefficients of the
                                                    %2nd order butterworth filter with cutoff freq 0.25
                                                    % if you use ldiv this will give h :filter weights to be
h=  [0.0976;                                        %This is the actual filter that we are trying to recreate
    0.2873;
    0.3360;
    0.2210;
    0.0964;];
y = lsim(Gz,inp);                                   %This simulates the time response of the system Gz given random input inp
                                                    %add some noise
n = n * std(y)/(10*std(n));
d = y + n;
totallength=size(d,1);
                                                    %Take 60 points for training
N=60 ;
                                                    %begin of algorithm
w = zeros ( sysorder  , 1 ) ;
for n = sysorder : N
        u = inp(n:-1:n-sysorder+1) ;
    y(n)= w' * u;                                   %Compute y(n) with the weights and the input
```

```
    e(n) = d(n) - y(n) ;                                    %Compute the error: d(n) is the true input
                                                            %Start with big mu for speeding the convergence then slow down
                                                            %to reach the correct weights
    if n < 20
        mu=0.32;
    else
        mu=0.15;
    end
                                                            %The update corresponds to the LMS update rule, which can be derived
                                                            %using the gradient of the cost function and the approximation of the
                                                            %expectation function E{e(n)x(n)} which is approximated by the average
                                                            %over n points. In the lms we use n=1, so just e(n)x(n)
        w = w + mu * u * e(n) ;                                 %Update the weights w.r.t. the old weight,learning rate mu, input u, and error e(n)
end
                                                            %Check of results on
                                                            %the rest of the data
for n =  N+1 : totallength
        u = inp(n:-1:n-sysorder+1);
    y(n) = w' * u ;                                         %Compute the results with the obtained weight vector
    e(n) = d(n) - y(n) ;                                    %Compute the error
end
hold on
plot(d)
plot(y,'r');                                                %Plot the obtained output vs. the true output d
title('System output') ;
xlabel('Samples')
ylabel('True and estimated output')
figure
semilogy((abs(e))) ;                                        %Create semi log scale plot (only y axis)
```

```
title('Error curve') ;
xlabel('Samples')
ylabel('Error value')
figure
                                                    %Plot the weights h and w and
                                                        let the user see the
                                                        difference
plot(h, 'k+')
hold on
plot(w, 'r*')
legend('Actual weights','Estimated weights')
title('Comparison of the actual weights and the estimated weights') ;
axis([0 6 0.05 0.35])
```

7 Python code

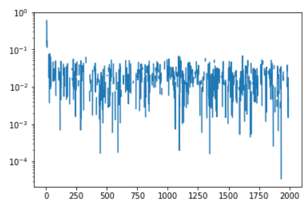
In [251]:
```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
```

In [252]:
```
sysorder = 5
N=2000
h = np.array([0.0976,0.2873,0.3360,0.2210,0.0964])
inp = np.transpose(np.random.rand(1,N))
n = np.random.rand(1,N)
b,a = signal.butter(2,0.25,'low')
Gz = signal.TransferFunction(b,a,dt=0.1)
t_out, y = signal.dlsim(Gz,inp) #Important to use discrete simulation
y = np.concatenate(y)
n = np.concatenate(n * np.std(y)/(10*np.std(n)))
d = y + n
totallength = np.size(d)
N = 60
```

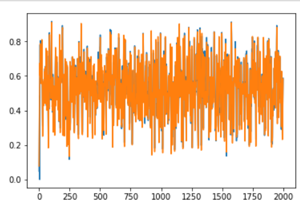
In [256]:
```
w = np.zeros([1,sysorder])
e = np.zeros([1,totallength])
for n in range(sysorder, N):
    index = np.linspace(n,n-sysorder+1,sysorder,dtype='int') #start stop number=stop
    u = np.transpose(np.concatenate(inp[index]))
    y[n] = w.dot(u)

    e[0,n] = d[n] - y[n]
    if n < 20:
        mu = 0.32
    else: mu = 0.15

    w = w + mu*u*e[0,n]
```

In [258]:
```
for n in range(N+1,totallength):
    index = np.linspace(n,n-sysorder+1,sysorder,dtype='int') #start stop number=stop
    u = np.transpose(np.concatenate(inp[index]))
    y[n] = w.dot(u)
    e[0,n] = d[n] - y[n]
```

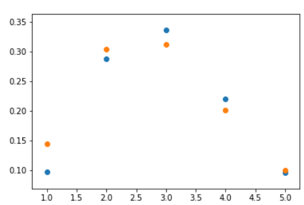
In [259]:
```
t = np.arange(totallength)
plt.plot(t,np.transpose(e[:,0:totallength]))
plt.yscale('log')
plt.show()
```



In [260]:
```
plt.plot(t,y)
plt.plot(t,np.transpose(d))
plt.show()
```



In [271]:
```
x = np.linspace(1,np.size(h),np.size(h))
plt.scatter(x, h)
plt.scatter(x, w)
plt.show()
```



In [ ]:

7.1 References

[1] On the Convergence Behavior of the LMS and the Normalized LMS Algorithms, D.Slock, Trans. on Signal Processing 1993

[2] Convergence and Performance Analysis of the Normalized LMS Algorithm with Uncorrelated Gaussian Data, Trans. of Information Theory 1988

[3] Recursive Least Squares Estimation, http://www.cs.tut.fi/ tabus/course/ASP/LectureNew10.pdf

[4] Convergence Behavior of NLMS Algorithm for Gaussian Inputs: Solutions Using Generalized Abelian Integral Functions and Step Size Selection, Journal of Signal Processing Systems