Adaptive Signal Processing Project report
Kernel Based Identification of Hammerstein Model for Nonlinear Echo cancellation

Saurabh Vaishampayan:EP17B028

December 2020

Contents

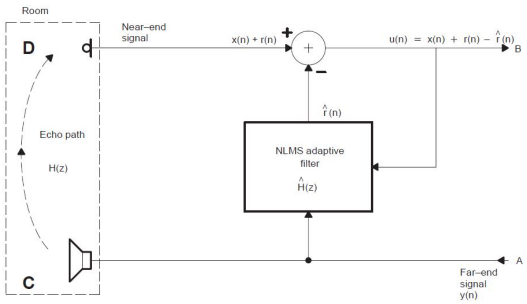
1 Abstract

This report is based on the following paper by St Vaerenbergh: Kernel Based Identification of Hammerstein Model for Nonlinear Acoustic Echo Cancellation[1]

We will be looking at the problem of acoustic echo cancellation and the need for nonlinear echo cancellation due to practical reasons. We briefly state the Hammerstein model and its features. Then we motivate using Kernel methods for modelling instantaneous nonlinearity. We look at the properties of cost function briefly and gradients for each of the parameters. We later dive into an offline identification method for the model and implement it in python, and give our observations. We then cover the online time update method covered in the paper. However the method given in the method only deals with static nonlinearity and time varying linear part. **As an extension of the paper based on concepts covered in class we propose a time update for nonlinearity as well.** We later summarise our findings.

2 Introduction to the problem

In a general auditorium or room setting, the broadcasted acoustic signal undergoes some distortion as it propagates through the room. The sound can be thought of as travelling through multiple paths and undergoing phase change, time lags and interference, We wish to correct for this distortion. The simplest technique usually is to estimate a linear model for the distortion and then subtract his from the received signal to obtain an estimate of the original far end audio. This is portrayed as a figure below, reference [2]



However one of the practical difficulties faced is that the audio undergoes nonlinear distortion as it is being transmitted through a loudspeaker, especially if the chosen speaker is of lower quality. In such a case, a linear estimate of the channel will not only be off from the actual channel coefficients, but also the end result will be off due to the nonlinearity. Keeping this in mind, the authors of the paper proposed a method to correct for this effect.

The model used is called Hammerstein model. It consists of two parts: an instantaneous nonlinearity followed by a linear time invariant filter.

$$y[n] = h[n] * f(x[n]) \tag{1}$$

Here $*$ represents convolution. f is an instantaneous nonlinearity. One assumes that input is sent to speakers where it undergoes nonlinear distortion followed by convolution with room impulse response.

One can clearly see that the model is time invariant, though not linear. The solution hence consists of two parts: Identify the nonlinearity and identify the channel. The method used for identifying the nonlinearity is based on the Kernel trick, which we explore in detail in the next section.

3 Kernel trick

We briefly take a dive into the problem of regression. The task is to approximate the input output data using some parmeters(to be estimated).

$$y_i \approx w_0\phi_0(x_i) + w_1\phi_1(x_i) + w_2\phi_2(x_i)... + w_d\phi_d(x_i)$$

with i^{th} input output data points being x_i, y_i respectively.

Here ϕ_i are basis functions. The goal is to find optimal weight vector to have as good as an approximation as possible. In matrix form

$$y \approx \phi w$$

Here y, w are column vectors and ϕ is an $n \times d$ matrix, with d being number of basis functions and n being number of data points. For a wide matrix($d > n$), the least squares solution is given by:

$$w = \phi^T(\phi\phi^T)^{-1}y$$

But since we are only interested in prediction for a given input, we do not need to explicitly calculate ϕ matrices and then find w for prediction, which is costly. Instead one employs a Kernel trick, where a relevant kernel is chosen such that

$$K(u, v) =< \phi(u), \phi(v) >$$

ie the kernel represents inner product of two functions in infinite dimensional Hilbert space. One must choose a suitable Kernel instead as part of the model and use it for making predictions. Popular kernels are gaussian($K(u, v) = e^{\frac{-||u-v||^2}{\sigma^2}}$, polynomial($K(u, v) = (1 + u^T v)^k$).

The Kernel method can be though of as doing interpolations about the given training points.

The least squares objective becomes

$$min_\alpha ||y - K\alpha||^2$$

whose solution is

$$\alpha = K^{-1}y$$

Here $K_{ij} = K(x_i, x_j)$.

Instead of interpolating about training points, one can choose a predetermined fixed set of interpolation points(x_s), called supports. In this case the optimal solution is given by:

$$\alpha = (K^T K + cK_s)^{-1}K^T y \tag{2}$$

Here $K_{ij} = K(x_i, x_{s_j})$, $K_{s_{ij}} = K(x_{s_i}, x_{s_j})$, with x_i being training points and x_{s_j} being support points. c is regularisation parameter.

4 The cost function

The model assumed is:

$$d[n] = h[n] * f(x[n]) \tag{3}$$

We assume access to some training points $\{x[n], d[n]\}_{n=1}^{N}$. Finding h, f is done using Linear model, Kernel methods respectively. We use a least squares cost function:

$$min_{h,\alpha} ||d - h * K\alpha||^2$$

Here $*$ represents convolution. h is stacked as a tap delay line toeplitz matrix. The cost functions for α and h individually are given by

$$J_\alpha = ||d - K_h\alpha||^2 + c_\alpha \alpha^T K_s \alpha \tag{4}$$

$$J_h = ||d - K_\alpha h||^2 + c_h h^T h \tag{5}$$

Here $K_h = h * K$, with h stacked as columns in tap delay line toeplitz form, $K_\alpha = K\alpha$.
One can clearly see that the cost is individually convex in h and α but is not jointly convex. Simple gradient descent is not guaranteed to give global minimum, and we need to choose initialisation points properly.

5 Batch algorithm for offline identification

5.1 Summary of the algorithm

To determine linear coefficients as well as Kernel coefficients, the paper proposes an iterative update model:

1. Initialise $\hat{h} = (X^T X + c_h I)^{-1} X^T d$
2. while J not converged do:

   $\hat{\alpha} = (K^T K + c_\alpha K_s)^{-1} K^T d$

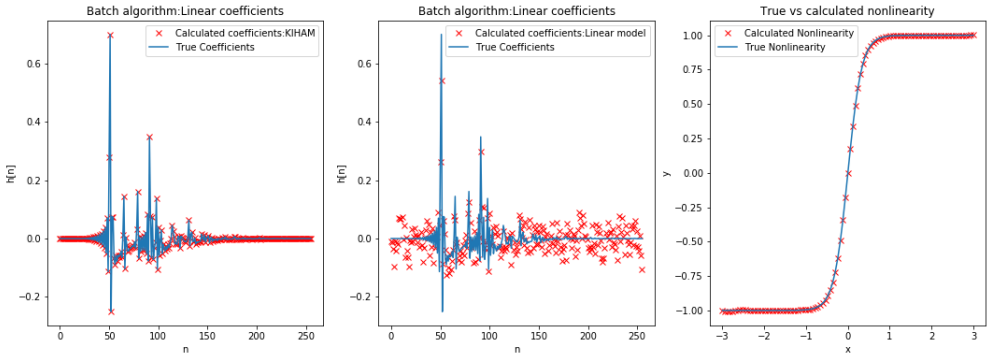
   $\hat{h} = (K_\alpha^T K_\alpha + c_h I)^{-1} K_\alpha^T d$ end while
3. Return $\hat{\alpha}, \hat{h}$

5.2 Implementation specifications in Python

- The nonlinearity chosen was $tanh(3x)$.
- Coefficients for Room Impulse Response were taken from [3].
- Number of training samples were 2048, with gaussian input with variance as 1 and noise as 30dB.
- Regularisation parameters for h and α were 1.0, 0.001 respectively.
- Support points were 0.3 apart from -3 to 3 and σ for gaussian kernel was 0.6

5.3 Results

Jmin for Kernel based Identification: 0.00014955080031167907
Jmin for linear model: 0.4803247339355206
Figures



5.4 Observations and conclusions

From the minimum cost functions obtained as well as plots of observed vs actual coefficients one can clearly see that purely linear model does a poor job at estimation, while the Kernel based method does a near perfect job. From the plotted nonlinearity also one can clearly see that the calculated nonlinearity matches the actual one.

6 Online update for linear part with static nonlinearity

6.1 Summary of the algorithm

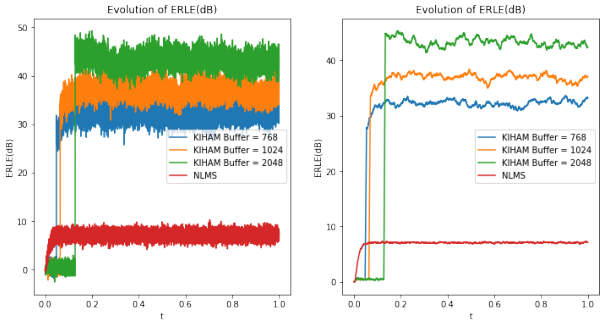
The motivation from this arises from the fact that the room impulse response is prone to rapid changes in time while the nonlinearity usually changes slowly. Just a small change in the source location and direction can result in a change in the room impulse response, while change in nonlinearity usually happens in slow timescales.
The method for online update in the paper consists of the following steps:

- Choose a predetermined buffer of some length.
- Until the buffer fills, update the linear coefficients by NLMS.
- After the buffer has fillled, calculate nonlinearity as well as linear model estimates.
- Now time update the linear coefficients via NLMS

6.2 Implementation specifications in Python

- The nonlinearity chosen was $tanh(x)$.
- Coefficients for Room Impulse Response were taken from [3].
- 3 buffer lengths were chosen for the initialisation part: 768, 1024, 2048.
- Number of iterations were 16000, since the above impulse response was sampled at 16KHz, ie a time of 1s
- Gaussian input with variance as 1 and noise as 30dB.
- Regularisation parameters for h and α were 1.0, 0.001 respectively.
- Support points were 0.3 apart from -3 to 3 and σ for gaussian kernel was 0.6

6.3 Results



6.4 Conclusions

We observe that vis a vis the simple linear model NLMS update, all of the graphs for this method perform better. This adds on to the previous section where the achieved cost was very low for Kernel based vs purely linear. We are not updating the nonlinearity(as per the paper), after it is estimated once the buffer is filled, so it is natural that the algorithms with bigger buffer size gives better performance.

7 Extra effort: Online update of nonlinearity

7.1 Description of method

This part is extra work on my behalf as an extension of the work done in the paper and the concepts covered in class.

Here we update coefficients for both the kernel and the linear part in real time. Since the cost is a least squares one, the gradient and update are of the same form as any standard quadratic cost minimisation, but with more complexities due to presence of two parts to optimise over. Here since we are dealing with real vectors, we use transpose instead of conjugate

We write the full mathematical form for the cost again below:

$J = ||d - h * K\alpha||^2 + c_\alpha \alpha^T K_s \alpha + c_h h^T h$

Writing the gradients:

$(\nabla_\alpha J)^T = (K_h^T K_h + c_\alpha K_s)\alpha - K_h^T d$

$(\nabla_h J)^T = (K_\alpha^T K_\alpha + c_h I)h - K_\alpha^T d$

The actual variable to estimate is h and α stacked. The gradient wrt this variable is just the gradients of each stacked. **Finding the Hessian however is no trivial task**

The hessian is given by:

$$\nabla^2 J = \begin{pmatrix} \nabla_\alpha^2 J & S \\ S^T & \nabla_\alpha^2 J \end{pmatrix}$$

Finding the matrix S, which corresponds to cross-terms, is not trivial. **However while Newton's method requires inverse of Hessian times gradient transpose as descent direction, using any other positive semidefinite matrix will also be a valid descent direction, though the cost will not decrease as quickly as Newton's method. We note that the matrix**

$$A = \begin{pmatrix} \nabla_\alpha^2 J & 0 \\ 0 & \nabla_\alpha^2 J \end{pmatrix}^{-1} = \begin{pmatrix} (\nabla_\alpha^2 J)^{-1} & 0 \\ 0 & (\nabla_\alpha^2 J)^{-1} \end{pmatrix}$$

$(\nabla_\alpha^2 J) = (K_h^T K_h + c_\alpha K_s), (\nabla_h^2 J) = (K_\alpha^T K_\alpha + c_h I)$

is also positive definite. So assuming this we derive NLMS-like expressions for update. The true NLMS update equations would have made use of the correct Hessian expressions, but we use a different positive definite matrix which give us expressions that look like NLMS updates in each of h and α individually

Also, because we have added a regularisation parameter, the update equations are similar to leaky-LMS/leaky NLMS as discussed in the textbook by Sayed[4]

LMS-like update equation:

$$h_{i+1} = h_i + \mu(K_{\alpha_i}^T[d_i - K_{\alpha_i}h_i] - c_h h_i)$$

$$\alpha_{i+1} = \alpha_i + \mu(K_{h_i}^T[d_i - K_{h_i}\alpha_i] - c_\alpha \alpha_i)$$

Here both $K_{\alpha_i}^T$ and $K_{h_i}^T$ are column vectors, not matrices, since we are only dealing with one data point at a time.

NLMS like update equation:

$$h_{i+1} = h_i + \mu \frac{(K_{\alpha_i}^T[d_i - K_{\alpha_i}h_i] - c_h h_i)}{c_h + K_{\alpha_i}K_{\alpha_i}^T}$$

$$\alpha_{i+1} = \alpha_i + \frac{\mu}{c_\alpha}(K_s^{-1})(I - \frac{K_{h_i}^T K_{h_i} K_s^{-1}}{c_\alpha + K_{h_i}K_s^{-1}K_{h_i}^T})(K_{h_i}^T[d_i - K_{h_i}\alpha_i] - c_\alpha \alpha_i)$$

Due to lower computational complexity we implement LMS update for nonlinearity and NLMS update for linear part in the code.

An important caveat here of course is that the cost function is convex for each of the parameters individually, but not jointly, so gradient descent cannot guarantee a global minimum.

7.2 Python implementation specifications

- The nonlinearity chosen was $tanh(x)$.
- Coefficients for Room Impulse Response were taken from [3].
- Length of buffer was 2048. An initial guess is obtained after buffer is filled, after which update is done online.
- Gaussian input with variance as 1 and noise as 30dB.
- Regularisation parameters for h and α were 1.0, 0.001 respectively.
- Support points were 0.3 apart from -3 to 3 and σ for gaussian kernel was 0.6

7.3 Results

7.4 Conclusions

We observe that the new method gives a better performance by around 4.3dB over the method proposed in the paper. This is because we are updating our estimate for nonlinearity as well online, while the paper does not update its estimate. Because it's estimate is based on a finite size noisy data, there is room for improvement, which our new algorithm accounts for by updating. Due to lesser computational complexity, we only used LMS update for nonlinearity. One can also use NLMS like updates as given above, but one needs either higher computation or updates can be done after every few iterations instead of after every iteration.

8 Summary

In summary we have looked at performing system identification using the Hammerstein model. We estimated the linear and nonlinear part by using a least squares objective function, with a kernel based approach for the nonlinear function. This can be later used for echo cancellation by subtracting the estimate of the distortion. We looked at 3 methods: Offline method for identification, Online method where linear part is in time update form and an extra section where we saw time update for linear as well as nonlinear part.

9 References

[1] S. Vaerenbergh, L Azpiceuta Ruiz ,KERNEL-BASED IDENTIFICATION OF HAMMERSTEIN SYSTEMS FOR NONLINEAR ACOUSTIC ECHO-CANCELLATION, International Conference on Acoustics, Speech, and Signal Processing 2014 IEEE.
[2] David Qi, Acoustic Echo Cancellation, Application Report Texas Instruments, 1996
[3] Nils Werner Room Impulse Response Generator, https://pypi.org/project/rir-generator/
[4] Ali H Sayed, Adaptive Filter Theory, Wiley an Sons, 2008

10 Appendix: Python code(Jupyter)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import toeplitz
```

10.1 Batch Identification Algorithm(Offline)

10.1.1 Function definitions

[2]:
```
def calculate_kernel_matrix(x,y,hyperparam,kernel_type):
    '''
    kernel_type:'linear' or 'gaussian' or 'polynomial'
    hyperparam: sigma for gaussian, degree for polynomial
    '''
    if kernel_type=='linear':
        K = np.outer(x,y)
    if kernel_type=='gaussian':
        K = np.exp(-0.5*((x[:,None]-y[None,:])**2)/(hyperparam**2))
    if kernel_type=='polynomial':
        K = (1+np.outer(x,y))**hyperparam
    return K
```

[3]:
```
def find_ls_h(x,L,d,reg_param_channel):
    col1 = x
    row1 = np.zeros(L)
    X = toeplitz(col1,row1)

    h_opt = (np.linalg.inv(X.T@X+reg_param_channel*np.eye(L)))@(X.T@d)

    return h_opt
```

[4]:
```
def find_kernel_coeff(K,h,Ks,d,reg_param_kernel):
    H = toeplitz(np.concatenate((h,np.zeros(len(K)-len(h)))),np.zeros(len(K)))
    K_h = H@K
    inv = np.linalg.inv(K_h.T@K_h+reg_param_kernel*Ks)
    b = K_h.T@d
    alpha = inv@b
    return alpha
```

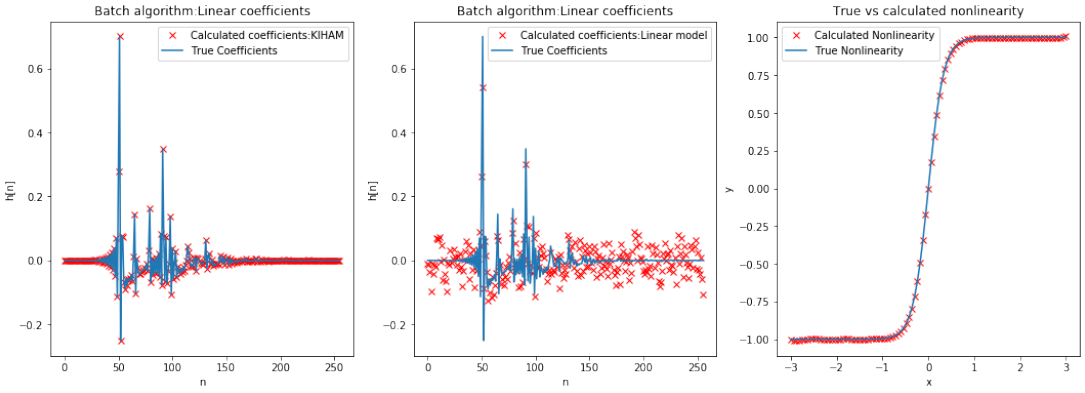
[5]:
```
def batch_identification_kiham(x,d,h_init,channel_dict,kernel_dict,convergence_dict):

    L = channel_dict['Length']
    reg_param_channel = channel_dict['Regularisation']

    M = kernel_dict['Support Number']
    spac = kernel_dict['Support Spacing']
    reg_param_kernel = kernel_dict['Regularisation']
    kernel_type = kernel_dict['Type']
    kernel_hyperparam = kernel_dict['Hyperparameter']

    max_iter = convergence_dict['Maximum Iterations']
    err_tol = convergence_dict['Error Tolerance']

    x_s = np.arange(-0.5*M,0.5*M,spac)
```

```
    K = calculate_kernel_matrix(x,x_s,kernel_hyperparam,kernel_type)
    Ks = calculate_kernel_matrix(x_s,x_s,kernel_hyperparam,kernel_type)

    d_norm = np.linalg.norm(d)

    # Initialise h[n]

    h = h_init/np.linalg.norm(h_init)
    h_linear = h.copy()

    n_iter = 0

    J_frac = d_norm**2

    while ((n_iter<=max_iter) and (J_frac>=err_tol)):
        alpha = find_kernel_coeff(K,h,Ks,d,reg_param_kernel)
        f_x = K@alpha
        h = find_ls_h(f_x,L,d,reg_param_channel)
        h = h/np.linalg.norm(h)

        H = toeplitz(np.concatenate((h,np.zeros(len(x)-len(h)))),np.zeros(len(x)))

        n_iter+=1
        J_frac = (np.linalg.norm(d-H@K@alpha))**2/np.linalg.norm(d)

    return h,alpha,n_iter,J_frac,x_s,h_linear
```

[6]:
```
n_iter = 1024
x = np.random.randn(n_iter)
f_x = np.tanh(3*x)

h_actual = np.loadtxt(open("rir1.csv"), delimiter=",")

d = np.convolve(h_actual,f_x)[0:n_iter]+0.001*np.random.randn(n_iter)

M = 50
spac = 0.2
c_h = 1.0
c_a = 0.01
sigma = 0.4

L = 256
k_type = 'gaussian'

max_iter = 50
err_tol = 1e-4

channel_att = {'Length':L,'Regularisation':c_h}
kernel_att = {'Support Number':M,'Support Spacing':spac,
              'Regularisation':c_a,'Type':k_type,'Hyperparameter':sigma}
conv_att = {'Maximum Iterations':max_iter,'Error Tolerance':err_tol}

h_init = find_ls_h(x,L,d,c_h)

h,alpha,el_iter,J_frac,x_s,h_linear =␣
↪batch_identification_kiham(x,d,h_init,channel_att,kernel_att,conv_att)
```

[7]:
```
x_ = np.linspace(-3,3,101)
y = np.tanh(3*x_)

K_new = calculate_kernel_matrix(x_,x_s,sigma,'gaussian')

y_pred = K_new@alpha
```

[26]:
```
fig, ax = plt.subplots(1,3,figsize=(18,6))

ax[0].plot(h,'rx',label='Calculated coefficients:KIHAM')
ax[0].plot(h_actual,label='True Coefficients')
ax[0].set_xlabel('n')
ax[0].set_ylabel('h[n]')
ax[0].set_title('Batch algorithm:Linear coefficients')
ax[0].legend()

ax[1].plot(h_linear,'rx',label='Calculated coefficients:Linear model')
ax[1].plot(h_actual,label='True Coefficients')
ax[1].set_xlabel('n')
ax[1].set_ylabel('h[n]')
ax[1].set_title('Batch algorithm:Linear coefficients')
ax[1].legend()

ax[2].plot(x_,y_pred,'rx',label='Calculated Nonlinearity')
ax[2].plot(x_,y,label='True Nonlinearity')
ax[2].set_xlabel('x')
ax[2].set_ylabel('y')
ax[2].set_title('True vs calculated nonlinearity')
ax[2].legend()

#plt.savefig('batch.png')
plt.show()
```



[9]:
```
J_frac
```

[9]:
```
0.00014955080031167907
```

[10]:
```
d_linear_pred = np.convolve(h_linear,x)[0:n_iter]
J_linear = np.linalg.norm(d-d_linear_pred)/np.linalg.norm(d)
J_linear
```

[10]:
```
0.4803247339355206
```

10.2 Online learning: Static nonlinearity and time varying channel

Uptil the input length fills a buffer of a predetermined length, one performs NLMS to obtain estimates for channel coefficients. Then one uses the data in the buffer to obtain estimates for kernel coefficients and channel coefficients. Then one continues updating the coefficients for the linear path by NLMS

[17]:
```
Nb = [768,1024,2048]
n_iter = 16000
n_ensemble = 100

M = 50
spac = 0.2
c_h = 10.0
c_a = 0.01
sigma = 0.4

L = 256
k_type = 'gaussian'

x_s = np.arange(-0.5*M,0.5*M,spac)
Ks = calculate_kernel_matrix(x_s,x_s,sigma,'gaussian')

max_iter = 50
err_tol = 1e-4

channel_att = {'Length':L,'Regularisation':c_h}
kernel_att = {'Support Number':M,'Support Spacing':spac,
              'Regularisation':c_a,'Type':k_type,'Hyperparameter':sigma}
conv_att = {'Maximum Iterations':max_iter,'Error Tolerance':err_tol}

mu = 0.2

x_iter = np.zeros(L)
x_temp = np.zeros(L)
h_nlms = np.random.randn(L)
erle = np.zeros((3,n_ensemble,n_iter))
f_x_iter = np.zeros(L)


for k in range(3):
    for j in range(n_ensemble):
        x_iter = np.zeros(L)
        x_train = np.zeros(Nb[k])
        d_train = np.zeros(Nb[k])
        h_nlms = np.zeros(L)
        for i in range(Nb[k]):
            x_iter[1:] = x_iter[0:-1].copy()
            x_iter[0] = np.random.randn(1)
            x_train[i] = x_iter[0].copy()

            d_iter = np.tanh(x_iter)@h_actual+0.0001*np.random.randn(1)
            d_train[i] = d_iter.copy()

            pred_temp = np.sum(x_iter*h_nlms)
            err_temp = d_iter-pred_temp

            erle[k][j][i] = 10*np.log10(d_iter**2/err_temp**2)
```

```
            h_nlms_new = h_nlms+mu*(err_temp*x_iter-c_h*h_nlms)/(c_h+np.linalg.
↪norm(x_iter**2))

            h_nlms = h_nlms_new.copy()

        h_nlms = h_nlms/np.linalg.norm(h_nlms)

        #h_init = find_ls_h(x_train,L,d_train,c_h)

        h_nlms,alpha,el_iter,J_frac,x_s,h_linear =␣
↪batch_identification_kiham(x_train,d_train,h_nlms,channel_att,kernel_att,conv_att)

        f_x_iter = calculate_kernel_matrix(x_iter,x_s,sigma,'gaussian')@alpha
        f_x_temp = np.zeros(L)

        for i in range(Nb[k],n_iter):
            x_iter[1:] = x_iter[0:-1].copy()
            x_iter[0] = np.random.randn(1)

            d_iter = np.sum(np.tanh(x_iter)*h_actual)+0.0001*np.random.randn(1)

            f_x_new = calculate_kernel_matrix(np.
↪array([x_iter[0]]),x_s,sigma,'gaussian')[0]@alpha

            f_x_iter[1:] = f_x_iter[0:-1].copy()
            f_x_iter[0] = f_x_new.copy()

            pred_temp = np.sum(f_x_iter*h_nlms)
            err_temp = d_iter-pred_temp

            erle[k][j][i] = 10*np.log10(d_iter**2/err_temp**2)

            h_nlms_new = h_nlms+mu*(err_temp*f_x_iter)/(c_h+np.linalg.
↪norm(f_x_iter**2))

            h_nlms = h_nlms_new.copy()
```

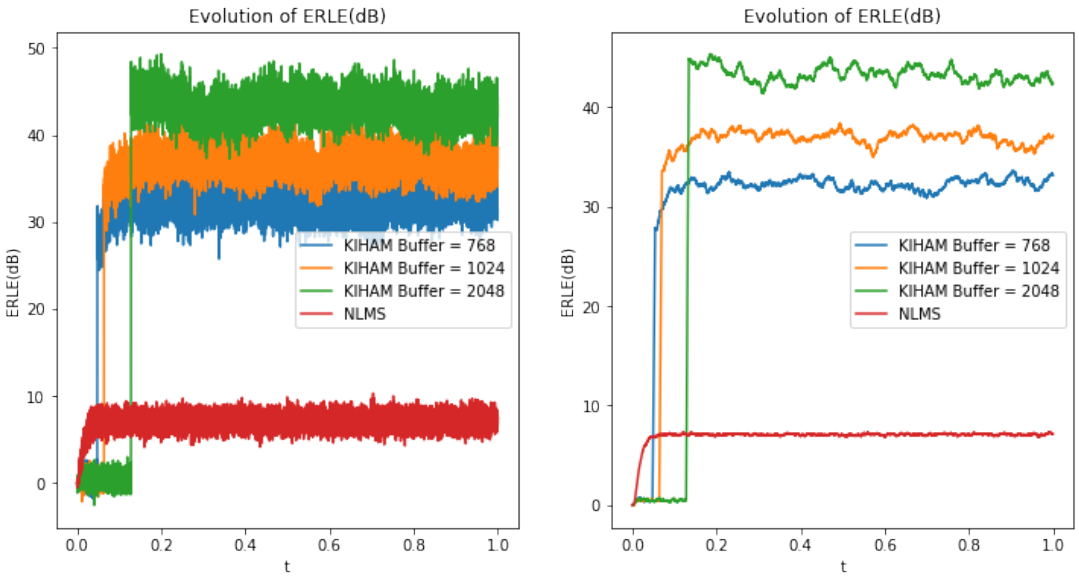
[18]:
```
erle = np.mean(erle,axis=1)
```

[19]:
```
n_ensemble = 300
erle_linear = np.zeros((n_ensemble,n_iter))

for j in range(n_ensemble):
    x_iter = np.zeros(L)
    h_nlms = np.zeros(L)
    for i in range(n_iter):
        x_iter[1:] = x_iter[0:-1].copy()
        x_iter[0] = np.random.randn(1)

        d_iter = np.sum(np.tanh(x_iter)*h_actual)+0.0001*np.random.randn(1)

        pred_temp = np.sum(x_iter*h_nlms)
        err_temp = d_iter-pred_temp

        erle_linear[j][i] = 10*np.log10(d_iter**2/err_temp**2)

        h_nlms_new = h_nlms+mu*(err_temp*x_iter)/(c_h+np.linalg.norm(x_iter**2))
```

```
        h_nlms = h_nlms_new.copy()
```

[20]:
```
erle_linear = np.mean(erle_linear,axis=0)
```

[27]:
```
t = np.linspace(0,1,n_iter)

mva = 0.01*np.ones(100)
b1 = np.convolve(erle[0],mva)[0:len(t)]
b2 = np.convolve(erle[1],mva)[0:len(t)]
b3 = np.convolve(erle[2],mva)[0:len(t)]
b4 = np.convolve(erle_linear,mva)[0:len(t)]

fig, ax = plt.subplots(1,2,figsize=(12,6))

ax[0].plot(t,erle[0],label='KIHAM Buffer = 768')
ax[0].plot(t,erle[1],label='KIHAM Buffer = 1024')
ax[0].plot(t,erle[2],label='KIHAM Buffer = 2048')
ax[0].plot(t,erle_linear,label='NLMS')
ax[0].set_xlabel('t')
ax[0].set_ylabel('ERLE(dB)')
ax[0].set_title('Evolution of ERLE(dB)')
ax[0].legend()

ax[1].plot(t,b1,label='KIHAM Buffer = 768')
ax[1].plot(t,b2,label='KIHAM Buffer = 1024')
ax[1].plot(t,b3,label='KIHAM Buffer = 2048')
ax[1].plot(t,b4,label='NLMS')
ax[1].set_xlabel('t')
ax[1].set_ylabel('ERLE(dB)')
ax[1].set_title('Evolution of ERLE(dB)')
ax[1].legend()

#plt.savefig('fig2.png')

plt.show()
```



10.3 Online learning: Time varying nonlinearity and channel, LMS like algorithm and NLMS-like algorithm (Extra)

This part is extra work on my behalf as an extension of the work done in the paper and the concepts covered in class.

Here we update coefficients for both the kernel and the linear part in real time. Since the cost is a least squares one, the gradient and update are of the same form as any standard quadratic cost minimisation, but with more complexities due to presence of two parts to optimise over. Here since we are dealing with real vectors, we use transpose instead of conjugate

We write the full mathematical form for the cost again below:

$J = ||d - h * K\alpha||^2 + c_\alpha \alpha^T K_s \alpha + c_h h^T h$

Writing the gradients:

$(\nabla_\alpha J)^T = (K_h^T K_h + c_\alpha K_s)\alpha - K_h^T d$

$(\nabla_h J)^T = (K_\alpha^T K_\alpha + c_h I)h - K_\alpha^T d$

The actual variable to estimate is h and α stacked. The gradient wrt this variable is just the gradients of each stacked. **Finding the Hessian however is no trivial task**

The hessian is given by:

$$\nabla^2 J = \begin{pmatrix} \nabla_\alpha^2 J & S \\ S^T & \nabla_\alpha^2 J \end{pmatrix}$$

Finding the matrix S, which corresponds to cross-terms, is not trivial. **However while Newton's method requires inverse of Hessian times gradient transpose as descent direction, using any other positive semidefinite matrix will also be a valid descent direction, though the cost will not decrease as quickly as Newton's method. We note that the matrix**

$$A = \begin{pmatrix} \nabla_\alpha^2 J & 0 \\ 0 & \nabla_\alpha^2 J \end{pmatrix}^{-1} = \begin{pmatrix} (\nabla_\alpha^2 J)^{-1} & 0 \\ 0 & (\nabla_\alpha^2 J)^{-1} \end{pmatrix}$$

$(\nabla_\alpha^2 J) = (K_h^T K_h + c_\alpha K_s), (\nabla_h^2 J) = (K_\alpha^T K_\alpha + c_h I)$

is also positive definite. So assuming this we derive NLMS-like expressions for update. The true NLMS update equations would have made use of the correct Hessian expressions, but we use a different positive definite matrix which give us expressions that look like NLMS updates in each of h and α individually

Also, because we have added a regularisation parameter, the update equations are similar to leaky-LMS/leaky NLMS as discussed in the textbook by Sayed.

LMS-like update equation:

$h_{i+1} = h_i + \mu(K_{\alpha_i}^T[d_i - K_{\alpha_i} h_i] - c_h h_i)$

$\alpha_{i+1} = \alpha_i + \mu(K_{h_i}^T[d_i - K_{h_i}\alpha_i] - c_\alpha \alpha_i)$

Here both $K_{\alpha_i}^T$ and $K_{h_i}^T$ are column vectors, not matrices, since we are only dealing with one data point at a time.

NLMS like update equation:

$h_{i+1} = h_i + \mu \frac{(K_{\alpha_i}^T[d_i - K_{\alpha_i} h_i] - c_h h_i)}{c_h + K_{\alpha_i} K_{\alpha_i}^T}$

$\alpha_{i+1} = \alpha_i + \frac{\mu}{c_\alpha}(K_s^{-1})(I - \frac{K_{h_i}^T K_{h_i} K_s^{-1}}{c_\alpha + K_{h_i} K_s^{-1} K_{h_i}^T})(K_{h_i}^T[d_i - K_{h_i}\alpha_i] - c_\alpha \alpha_i)$

Due to lower computational complexity we implement LMS update for nonlinearity and NLMS update for linear part in the code.

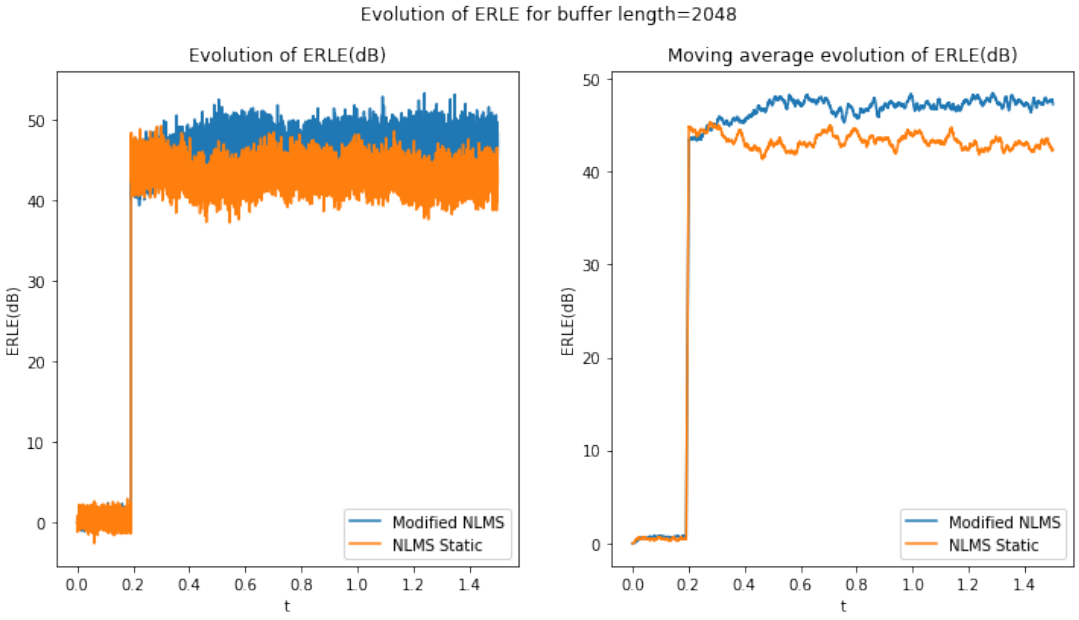
[22]:
```
Nb = [2048]
n_iter = 16000
n_ensemble = 100

M = 50
spac = 0.2
c_h = 10.0
c_a = 0.1
sigma = 0.4

L = 256
k_type = 'gaussian'

x_s = np.arange(-0.5*M,0.5*M,spac)
Ks = calculate_kernel_matrix(x_s,x_s,sigma,'gaussian')

max_iter = 50
err_tol = 1e-4

channel_att = {'Length':L,'Regularisation':c_h}
kernel_att = {'Support Number':M,'Support Spacing':spac,
              'Regularisation':c_a,'Type':k_type,'Hyperparameter':sigma}
conv_att = {'Maximum Iterations':max_iter,'Error Tolerance':err_tol}

mu = 0.02

x_iter = np.zeros(L)
x_temp = np.zeros(L)
h_nlms = np.random.randn(L)
erle_new = np.zeros((3,n_ensemble,n_iter))
f_x_iter = np.zeros(L)


for k in range(1):
    for j in range(n_ensemble):
        x_iter = np.zeros(L)
        x_train = np.zeros(Nb[k])
        d_train = np.zeros(Nb[k])
        h_nlms = np.zeros(L)
        del alpha
        for i in range(Nb[k]):
            x_iter[1:] = x_iter[0:-1].copy()
            x_iter[0] = np.random.randn(1)
            x_train[i] = x_iter[0].copy()

            d_iter = np.tanh(x_iter)@h_actual+0.0001*np.random.randn(1)
            d_train[i] = d_iter.copy()

            pred_temp = np.sum(x_iter*h_nlms)
            err_temp = d_iter-pred_temp

            erle_new[k][j][i] = 10*np.log10(d_iter**2/err_temp**2)

            h_nlms_new = h_nlms+mu*(err_temp*x_iter-c_h*h_nlms)/(c_h+np.linalg.
 ↪norm(x_iter**2))

            h_nlms = h_nlms_new.copy()
```

```
        h_nlms = h_nlms/np.linalg.norm(h_nlms)

        #h_init = find_ls_h(x_train,L,d_train,c_h)

        h_nlms,alpha,el_iter,J_frac,x_s,h_linear =␣
↪batch_identification_kiham(x_train,d_train,h_nlms,channel_att,kernel_att,conv_att)
        #print(J_frac)

        f_x_iter = calculate_kernel_matrix(x_iter,x_s,sigma,'gaussian')@alpha
        f_x_temp = np.zeros(L)

        for i in range(Nb[k],n_iter):
            x_iter[1:] = x_iter[0:-1].copy()
            x_iter[0] = np.random.randn(1)

            d_iter = np.sum(np.tanh(x_iter)*h_actual)+0.0001*np.random.randn(1)

            f_x_new = calculate_kernel_matrix(np.
↪array([x_iter[0]]),x_s,sigma,'gaussian')[0]@alpha

            f_x_iter[1:] = f_x_iter[0:-1].copy()
            f_x_iter[0] = f_x_new.copy()

            pred_temp = np.sum(f_x_iter*h_nlms)
            err_temp = d_iter-pred_temp

            erle_new[k][j][i] = 10*np.log10(d_iter**2/err_temp**2)

            Khi = h_nlms[0]*calculate_kernel_matrix(np.
↪array([x_iter[0]]),x_s,sigma,'gaussian')[0]

            #temp = np.eye(len(Ks))-(np.outer(Khi,Khi)@Ks_inv)/
↪(c_a+Khi@(Ks_inv@Khi))

            alpha_new = alpha+mu*((d_iter-Khi@alpha)*Khi)

            h_nlms_new = h_nlms+mu*(err_temp*f_x_iter)/(c_h+np.linalg.
↪norm(f_x_iter**2))

            h_nlms = h_nlms_new.copy()

            alpha = alpha_new.copy()
```

[23]:
```
erle_new = np.mean(erle_new,axis=1)
erle_old = erle[-1]
```

[28]:
```
t = np.linspace(0,1.5,n_iter)
#plt.plot(t,erle_lms_new,label='Modified LMS')
fig, ax = plt.subplots(1,2,figsize=(12,6))

mva = 0.01*np.ones(100)
a1 = np.convolve(mva,erle_new[0])[0:len(t)]
a2 = np.convolve(mva,erle_old)[0:len(t)]

ax[0].plot(t,erle_new[0],label='Modified NLMS')
ax[0].plot(t,erle_old,label='NLMS Static')
ax[0].set_xlabel('t')
```

```
ax[0].set_ylabel('ERLE(dB)')
ax[0].set_title('Evolution of ERLE(dB)')
ax[0].legend()

ax[1].plot(t,a1,label='Modified NLMS')
ax[1].plot(t,a2,label='NLMS Static')
ax[1].set_xlabel('t')
ax[1].set_ylabel('ERLE(dB)')
ax[1].set_title('Moving average evolution of ERLE(dB)')
ax[1].legend()

plt.suptitle('Evolution of ERLE for buffer length=2048')
#plt.savefig('fig3.png')
plt.show()

#plt.savefig('fig3.png')
```



[ ]: