LMS, ϵ-NLMS and RLS Simulations

Rishi Nandha V - EE21B111

EE6110 - Adaptive Signal Processing

October 11, 2023

1 Introduction

Least Mean Square (**LMS**), ϵ-Normalized Least Mean Square (ϵ**-NLMS**) and Recursive Least Mean Square (**RLS**) are closely related variants of the Least Mean Square Method used in adaptively deriving the parameters $\mathbf{w}$ of a filter that estimates $\mathbf{d}$ with given observations $\mathbf{u}$.

LMS uses the cost function $\mathbf{E}|e(i)|^2$ where $\mathbf{e} = \mathbf{d} - \mathbf{uw}$ and uses Stochastic Gradient Descent to update the weights using the relation:

$$\mathbf{w}_{i+1} = \mathbf{w}_i + \mu(\mathbf{u}_i^* \mathbf{e}(i))$$

ϵ**-NLMS** uses an additional normalization factor while updating weights, so that relation becomes:

$$\mathbf{w}_{i+1} = \mathbf{w}_i + \left(\frac{\mu}{\epsilon + ||\mathbf{u}_i||^2}\right)(\mathbf{u}_i^* \mathbf{e}(i))$$

RLS uses a Cost Function which is a Weighted Squared Error and a slight change in the Normalization Factor. The net effect of that can be represented with a factor $\mathbf{P}$ which is a matrix instead of μ.

$$\mathbf{P}_i = \frac{1}{\lambda}\left(\mathbf{P}_{i-1} - \frac{\mathbf{P}_{i-1}\mathbf{u}_i^*\mathbf{u}_i\mathbf{P}_{i-1}}{\lambda + \mathbf{u}_i\mathbf{P}_{i-1}\mathbf{u}_i^*}\right), \quad \mathbf{P}_{-1} = \frac{1}{\epsilon}I$$

$$\mathbf{w}_{i+1} = \mathbf{w}_i + \mathbf{P}_i(\mathbf{u}_i^* \mathbf{e}(i))$$

For the first simulation, *Numpy* was used in Python. For the second simulation which had a much larger number of operations, *PyTorch* with *CUDA & CuDNN accleration* was used. The *Python* codes have been attached as Minted Code Blocks.

2 Learning Curves of LMS, ϵ-NLMS & RLS

2.1 Tasks & Methodology

1. $D(z) = C(z) \cdot U(z) + N(z)$ where $C(z) = 1 + 0.5z^{-1} - z^{-2} + 2z^{-3}$, $\{u(i)\}$ is a white input sequence of variance 1 and $\{n(i)\}$ is a white noise of variance 0.01
2. 300 Independent Sets of Data were generated with 600 $\{d(i), u(i)\}$ Data-points each using the above conditions.
3. We aim to build a filter that behaves like $C(z)$, assuming we do not know the coefficients beforehand. So our filter is a 4-tap filter. In $e_i = d_i - u_i w_i$, u is a row vector of 4 causal terms and w is a column vector of 4 weights.
4. We use LMS with $\mu = 0.01$, RLS with $\epsilon = 0.001; \lambda = 0.995$ and ϵ-NLMS with $\epsilon = 0.001; \mu = 0.2$ (separately) and obtain $|e(i)|^2$ after each weights updation. Weights are updated once per data-point.
5. We average the results from the 300 Independent Experiments and plot the ensemble-averaged learning curve of a

2.2 Data Generation

The below code was used to generate the data required and store it. Here `d` and `u` are nested arrays with 300 arrays with each sub array having 600 data points

```
import numpy as np
import pickle

u = np.random.normal(loc=0, scale = 1, size = 300*(600) + 3)

n = np.random.normal(loc=0, scale = 0.1, size = 300*(600))

d = n + u[3:] + 0.5*u[2:-1] - u[1:-2] + 2*u[:-3]

u = u[3:].reshape((-1,600))
d = d.reshape((-1,600))

with open('expt1.bin','wb') as f:
        pickle.dump((d,u),f)
```

2.3 LMS

```
import numpy as np
import pickle as p

# Each Row is a seperate experiment
d,u = p.load(open('expt1.bin','rb'))
u = np.concatenate((np.zeros((300,3)),u), axis = 1)

# Weights to right are the coefficients of lower powers of z^-1
w = np.zeros((300,4))

# Each row is a seperate experiement's error
e = np.zeros((300,600))

# Learning Rate
mu = 0.01

for i in range(600):
        # w is the same as w^H
        # u[:, i:i+4] is the same as u
        # dpred same as uw
        dpred = np.sum(w * u[:,i:i+4],axis=1,keepdims=True)
        e[:,i:i+1] = d[:,i:i+1] - dpred
        # so here u*e[:,i:i+4] does the same as (u^H)*e(i)
        # since numpy multiplication is pointwise multiplication
        w = w + mu*u[:,i:i+4]*e[:,i:i+1]

ensemble_e = np.sum((e*e), axis = 0, keepdims=True)/300

with open('LMS_expt1.bin', 'wb') as f:
        p.dump(ensemble_e, f)
```

2.4 ϵ-NLMS

NLMS only has the two lines below different from LMS:

```
    unorm = np.sum(u[:,i:i+4] * u[:,i:i+4],axis=1,keepdims=True)

    w = w + (mu*u[:,i:i+4]*e[:,i:i+1])/(epsilon + unorm)
```

2.5 RLS

Below is the complete code for RLS:

```
d,u = p.load(open('expt1.bin','rb'))
u = np.concatenate((np.zeros((300,3)),u), axis = 1)

e = np.zeros((300,600))

Lambda = 0.995

for exp in range(300):

        P = np.array([[1000,0,0,0],
        [0,1000,0,0],
        [0,0,1000,0],
        [0,0,0,1000]
        ])

        w = np.array([0.5,0.5,0.5,0.5]).reshape((-1,1))

        d_exp = d[exp]
        u_exp = u[exp]

        for i in range(600):

                u_mat = np.array(u_exp[i:i+4]).reshape((1,4))

                d_mat = d_exp[i].reshape((1,1))

                e[exp,i] = (d_mat - (u_mat @ w))[0,0]

                A = P @ np.transpose(u_mat)

                P = (1/Lambda)*(P - (A @(u_mat @ P))/(Lambda + (u_mat @ A)))

                w = w + P@(np.transpose(u_mat) @ (d_mat - (u_mat @
 ↪w)))

ensemble_e = np.sum((e*e), axis = 0, keepdims=True)/300
```

2.6 Results

Below is a plot showing Ensemble Averaged Learning Curves from the three algorithms.

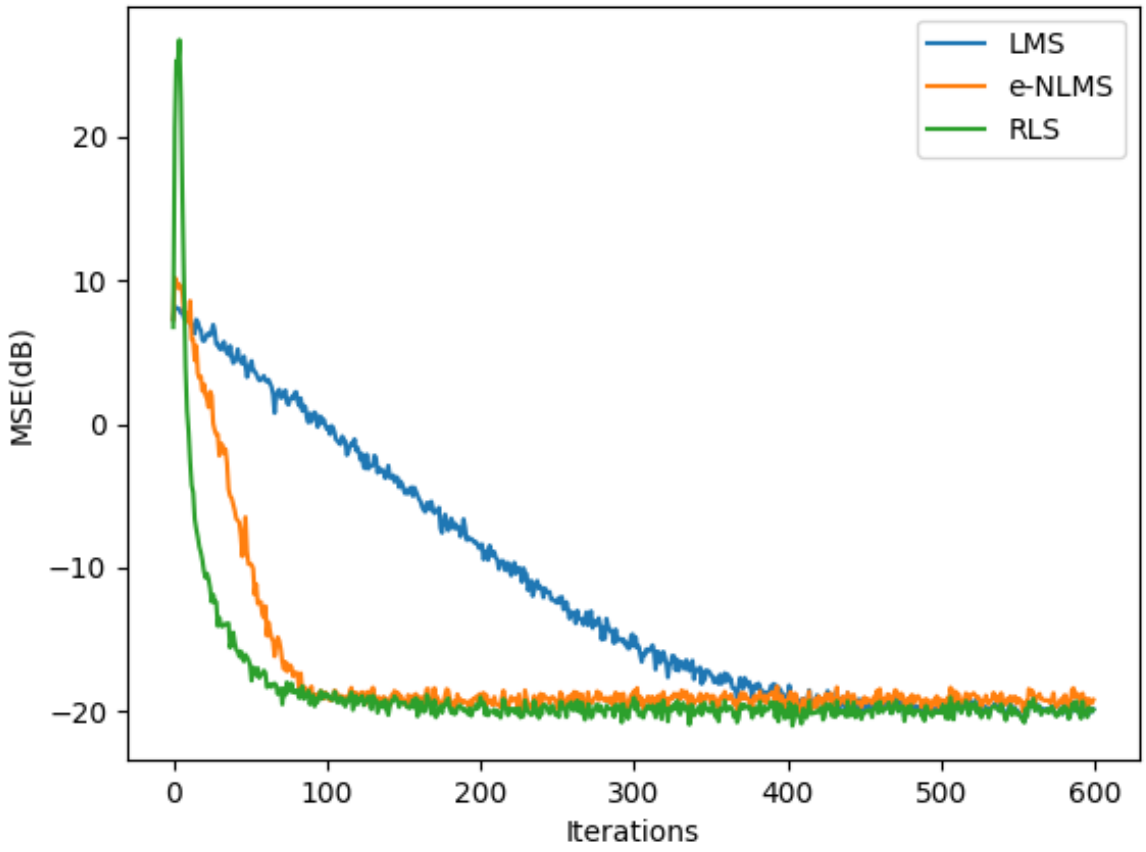

Figure 1: Ensemble-Averaged Learning Curves of LMS, RLS and ϵ-NLMS

2.7 Inferences

We can see that RLS converges faster than e-NLMS which converges much faster than LMS. But also note that LMS has a stabler behaviour as seen by how e-NLMS and RLS have a slight increase in error because it starts decreasing. In general, we can conclude that LMS can be done for any general data-set with lesser number of operations per iteration but **RLS will be much faster for any well behaved data-set**

3 Performance of LMS & the Step-Size of SGD

3.1 Tasks & Methodology

1. Excess Mean Squared Error is the excess error bound that the any algorithm saturates to from the minimum possible (which is the sum of variances of noise terms). We need to theoretically approximate this against step-size and compare it against simulation. EMSE can be approximated by the two following ways:

$$\zeta^{\text{LMS}}_{\text{Small }\mu} = \frac{\mu\sigma_v^2 \text{Tr}(R_u)}{2}$$

$$\zeta^{\text{LMS}}_{\text{Separation}} = \frac{\mu\sigma_v^2 \text{Tr}(R_u)}{2-\mu\text{Tr}(R_u)}$$

2. LMS Filter is defined with 10-Tap, trained with 4×10^5 Iterations and is done separately for a 100-Experiment Ensemble.

3. We have to generate two different Data-Sets described as follows:

   - Gaussian Regressors without Shift Structure: $\{\mathbf{u}_i\}$ is a set of vectors of 10 elements which are jointly Gaussian distributed with Covariance Matrix having eigenvalue spread of 5. $\{\mathbf{d}_i\}$ is generated by multiplying a vector of weights $\mathbf{w}$ to $\mathbf{u}_i$ and adding a Gaussian noise with $\sigma_v^2 = 0.001$

     To generate this we take, $u_i = Ag_i$ where g_i is a vector of 10 independent realizations of a normal distribution. This implies that $R_u = AA^T$ which must be chosen such that $\frac{\lambda_{\max}}{\lambda_{\min}} = 5$. For simplicity we choose the below A and use it for all the 100 Experiments in the Ensemble.

$$A = \begin{bmatrix} 1.7875 & -0.0598 & -0.1732 & -0.1297 & -0.0605 & -0.0515 & -0.1893 & 0.1579 & 0.0809 & -0.1704 \\ -0.0598 & 1.5689 & 0.0366 & 0.0039 & 0.0069 & -0.1182 & 0.0752 & 0.0848 & 0.1368 & -0.0087 \\ -0.1732 & 0.0366 & 1.8306 & -0.0134 & -0.1879 & -0.071 & -0.0453 & 0.0226 & -0.0407 & 0.0041 \\ -0.1297 & 0.0039 & -0.0134 & 1.5362 & 0.1064 & -0.0016 & -0.0852 & -0.0253 & -0.204 & 0.0384 \\ -0.0605 & 0.0069 & -0.1879 & 0.1064 & 1.7305 & -0.1806 & 0.0367 & 0.0874 & 0.2624 & -0.209 \\ -0.0515 & -0.1182 & -0.071 & -0.0016 & -0.1806 & 1.5696 & 0.0229 & 0.0041 & 0.2937 & 0.1439 \\ -0.1893 & 0.0752 & -0.0453 & -0.0852 & 0.0367 & 0.0229 & 1.5314 & -0.031 & 0.092 & -0.1965 \\ 0.1579 & 0.0848 & 0.0226 & -0.0253 & 0.0874 & 0.0041 & -0.031 & 1.6906 & 0.0107 & -0.0945 \\ 0.0809 & 0.1368 & -0.0407 & -0.204 & 0.2624 & 0.2937 & 0.092 & 0.0107 & 1.7288 & 0.1335 \\ -0.1704 & -0.0087 & 0.0041 & 0.0384 & -0.209 & 0.1439 & -0.1965 & -0.0945 & 0.1335 & 1.7872 \end{bmatrix}$$

   - Correlated Gaussian Input with Shift Structure: A Gaussian random sequence $\{\mathbf{g}_i\}$ is fed into a transfer function $A(z) = \frac{\sqrt{1-a^2}}{1-az^{-1}}$ with $a = 0.8$ to obtain the correlated data sequence $\{\mathbf{u}_i\}$. $\{\mathbf{d}_i\}$ is generated by multiplying a vector of weights $\mathbf{w}$ to $\mathbf{u}_i$ and adding a Gaussian noise with $\sigma_v^2 = 0.001$

     We use the convolution function in numpy & use the impulse response of the filter $a = \{0.6, (0.6)(0.8), (0.6)(0.8)^2 ...\}$ to generate this data. It's worth noting that R_{u_i} for this is a matrix defined as $\{a^{|i-j|}$; for valid i,j$\}$

4. We average the loss in the last 5000 iterations and estimate this to be our experimental saturation error. We use $\zeta_{\text{EMSE}} + J_{\min} = MSE_{\text{saturation}}$

3.2 Gaussian Regressors without Shift Structure

3.2.1 Data Generation

This below code generates 100 binary files holding the data we need for our simulation.

```
A = np.array([[1.7875 , -0.0598 , -0.1732 , -0.1297 , -0.0605 , -0.0515 , -
 ↪0.1893 , 0.1579 , 0.0809 , -0.1704],[-0.0598 , 1.5689 , 0.0366 , 0.0039 ,␣
 ↪0.0069 , -0.1182 , 0.0752 , 0.0848 , 0.1368 , -0.0087],[-0.1732 , 0.0366␣
 ↪, 1.8306 , -0.0134 , -0.1879 , -0.071 , -0.0453 , 0.0226 , -0.0407 , 0.
 ↪0041],[-0.1297 , 0.0039 , -0.0134 , 1.5362 , 0.1064 , -0.0016 , -0.0852 ,␣
 ↪-0.0253 , -0.204 , 0.0384],[-0.0605 , 0.0069 , -0.1879 , 0.1064 , 1.7305␣
 ↪, -0.1806 , 0.0367 , 0.0874 , 0.2624 , -0.209],[-0.0515 , -0.1182 , -0.
 ↪071 , -0.0016 , -0.1806 , 1.5696 , 0.0229 , 0.0041 , 0.2937 , 0.1439],[-0.
 ↪1893 , 0.0752 , -0.0453 , -0.0852 , 0.0367 , 0.0229 , 1.5314 , -0.031 , 0.
 ↪092 , -0.1965],[0.1579 , 0.0848 , 0.0226 , -0.0253 , 0.0874 , 0.0041 , -0.
 ↪031 , 1.6906 , 0.0107 , -0.0945],[
0.0809 , 0.1368 , -0.0407 , -0.204 , 0.2624 , 0.2937 , 0.092 , 0.0107␣
 ↪, 1.7288 , 0.1335],[-0.1704 , -0.0087 , 0.0041 , 0.0384 , -0.209 , 0.
 ↪1439 , -0.1965 , -0.0945 , 0.1335 , 1.7872]]).reshape((1,10,10)) + np.
 ↪zeros((400000, 10, 10))

w = np.array([1.7, -0.5, 1, -0.2, 6, 0.4, 4.5, -1.1, 1.3, -0.2]).
 ↪reshape((1,10,1))+np.zeros((400000,10,1))

for i in range(100):
        with open("expt2_data/expt2_"+str(i)+".bin", "wb") as f:
                g = np.random.normal(0, 1, size=(400000,1,10))
                u = np.matmul(g, A).reshape((400000,1,10))
                v = np.random.normal(0,np.sqrt(0.001), size=(400000, 1, 1))
                d = np.matmul(u, w) + v
                pack = (d , u)
                pickle.dump(pack, f)
```

While it's intuitive to run a loop for 100 times to go through each data-set. It is more efficient if we combine these 100 into 1 data-set and do tensor operations on it. The reason for this is even if the number of operations is the same, python loops are much slower than tensor operations especially since these operations are hardware accelerated in PyTorch. Thus we pack the data-set using the below code:

```
D, U = torch.reshape(torch.tensor([]),(400000,0,1,1)),torch.reshape(torch.
 ↪tensor([]),(400000,0,1,10))
for i in range(100):
        d,u = pickle.load(open("expt2_data/expt2_"+str(i)+".bin","rb"))
U = torch.cat((U, torch.reshape(torch.from_numpy(u),(400000,1,1,10))),1)
D = torch.cat((D, torch.reshape(torch.from_numpy(d),(400000,1,1,1))),1)
```

3.2.2 LMS with PyTorch Acceleration

We first import all the required modules.

```
import torch
import pickle
import numpy as np
from torch.utils.data import TensorDataset, DataLoader
from torch import nn
#from matplotlib import pyplot as plt
import pandas as pd
```

Then we open or create the required CSV file to store our results of experimental $MSE_{\text{saturation}}$.

```
csvpath = 'expt2results.csv'
try:
        df = pd.read_csv(csvpath, header=0, index_col=0)
        print("CSV Exists")
except:
        l = [str(i) for i in [0.0001, 0.0002, 0.0003, 0.0004, 0.0005, 0.
 ↪0006, 0.0007, 0.0008, 0.0009, 0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.
 ↪007, 0.008, 0.009, 0.01]]
        l = l[::-1]
        df = pd.DataFrame(columns=l)
        for _ in range(100):
                df = df.append(pd.Series(), ignore_index=True)
        df.to_csv(csvpath, index=True, header=True)
        print("Creating new CSV")
```

Next, we initiate our model's class in PyTorch and setup some variables useful for GPU Accelerating our code. Notice how we are using the model to initiate 1000 weights and then reshaping it the way we need it. `torch.matmul()` does matrix multiplication appropriately as we require.

```
no_of_i = 400000

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)

class MyCustomModel(nn.Module):
        def __init__(self, input_size, output_size):
                super(MyCustomModel, self).__init__()
                self.linear = nn.Linear(input_size, output_size)

        def forward(self, x):
                w = self.linear.weight.type(x.dtype)
                return torch.matmul(x,torch.reshape(w,(1,100,10,1)))
```

Now we defined some functions essential for us to perform SGD and track the error at each iteration. Notice here that the output of `mu_grad()` is reshaped such that it can be directed added to the weights of the model further.

```
def mu_grad(u_i, d_i, pred_i, mu):
        u_i = u_i.to(device)
        d_i = d_i.to(device)
        pred_i=pred_i.to(device)
        mu = mu.to(device)
        grad = mu*(torch.transpose(u_i,-1, -2) @ (d_i - pred_i))
        return torch.reshape(grad, (100,10))

def mse(t1,t2):
        t1, t2=t1.to(device), t2.to(device)
        return torch.reshape((t1-t2)*(t1-t2),[100,])
```

We proceed to initiate all of our variables and move them to the GPU at the start of the loop which iterates through different step sizes. Notice that at the start we are checking if we have already simulated this particular step size by checking against the CSV. This helps us checkpoint our code.

Also notice here that the model is given parameters of having an input size of 10 and output size of 100. This is just to get 1000 weights in total, it doesn't matter for anything more because we are using a custom `forward()` function and never using any attributes of the `nn.Linear()` class.

```
D,U = pickle.load(open("expt2_i_data.bin","rb"))
D = D.to(device)
U = U.to(device)

for mu in [0.0001, 0.0002, 0.0003, 0.0004, 0.0005, 0.0006, 0.0007, 0.0008,␣
 ↪0.0009, 0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008, 0.009, 0.
 ↪01]:
        col = Map[str(mu)] - 1
        if not(pd.isna(df.iloc[99, col])):
                print("Value exists already. Skipping iteration")
                continue

        mu_tensor = torch.reshape(torch.from_numpy(np.array([mu,])),(1,1))
        mu_tensor.to(device)
        error_last_5000 = torch.zeros(100)
        error_last_5000 = error_last_5000.to(device)

        dataset = TensorDataset(U, D)
        dataloader = DataLoader(dataset, 1, shuffle=False)

        model=MyCustomModel(input_size=10, output_size=100)
        model.to(device)
```

We proceed with making predictions, calculating loss, calculating gradient and updating the model. Finally we add the error to `error_last_5000` when we are in the final 5000 iterations. This is must later be divided by 5000 and stored in the CSV.

```
        iteration = torch.tensor([0])
        iteration=iteration.to(device)

        for u_i, d_i in dataloader:
                iteration=iteration+1
                u_i = u_i.to(device)
                d_i = d_i.to(device)

                preds = model(u_i)
                preds = preds.to(device)

                loss = mse(preds, d_i)
                loss = loss.to(device)

                grads = mu_grad(u_i, d_i, preds, mu_tensor)
                grads = grads.to(device)

                with torch.no_grad():
                        w = next(model.parameters())
                        w += grads

                if (iteration.item()>=no_of_i-5000):
                        error_last_5000 += loss
                        if (iteration.item()==no_of_i):
                                break
```

Like discussed above we then store the $MSE_{\text{Saturation}}$ into our CSV. We also add some verbose for our feedback.

```
                if (iteration.item()%500 == 0):
                        i=(iteration.item()*100)/no_of_i
                        print(("="*int(i//5))+("_"*int(20 - (i//5))), mu, i,␣
↪end="\r")

        df.iloc[:, col] = (error_last_5000.detach().cpu().numpy()/5000)
        df.to_csv(csvpath, index=True, header=True)
        print("="*20, mu, "Done")
```

Note that enclosing the entire loop with the context of `with torch.no_grad():` speeds up the algorithm too.

3.2.3 Results

The run-time of the program through all possible values of step size was about 50 minutes after complete vectorization and GPU acceleration with PyTorch. The (estimated) run-time with vanilla Python was about 12 hours and with complete vectorization using NumPy was about 2.5 hours. Below is what the obtained CSV of Saturation Errors.

CSV Exists

	0.01	0.009	0.008	0.007	0.006	0.005	0.004	0.003	0.002	0.001	0.0009	0.0008	0.0007	0.0006	0.0005	0.0004	0.0003	0.0002	0.0001
0	0.001178	0.001157	0.001136	0.001117	0.001098	0.001080	0.001062	0.001045	0.001029	0.001012	0.001011	0.001009	0.001007	0.001006	0.001004	0.001003	0.001001	0.001000	0.000999
1	0.001194	0.001171	0.001150	0.001129	0.001110	0.001091	0.001074	0.001057	0.001041	0.001026	0.001024	0.001023	0.001021	0.001020	0.001018	0.001017	0.001015	0.001014	0.001012
2	0.001186	0.001165	0.001144	0.001125	0.001107	0.001090	0.001073	0.001057	0.001042	0.001026	0.001025	0.001023	0.001022	0.001020	0.001019	0.001017	0.001016	0.001014	0.001012
3	0.001183	0.001161	0.001139	0.001119	0.001099	0.001079	0.001060	0.001042	0.001025	0.001008	0.001006	0.001005	0.001003	0.001001	0.001000	0.000998	0.000996	0.000995	0.000993
4	0.001164	0.001143	0.001124	0.001105	0.001087	0.001070	0.001053	0.001038	0.001024	0.001009	0.001008	0.001007	0.001005	0.001004	0.001002	0.001001	0.000999	0.000998	0.000997
..	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
95	0.001182	0.001160	0.001140	0.001120	0.001100	0.001082	0.001063	0.001045	0.001028	0.001011	0.001009	0.001008	0.001006	0.001005	0.001003	0.001002	0.001000	0.000999	0.000997
96	0.001150	0.001131	0.001112	0.001095	0.001078	0.001061	0.001046	0.001030	0.001016	0.001002	0.001000	0.000999	0.000997	0.000996	0.000995	0.000993	0.000992	0.000991	0.000989
97	0.001196	0.001173	0.001152	0.001131	0.001111	0.001091	0.001073	0.001055	0.001038	0.001021	0.001020	0.001018	0.001016	0.001015	0.001013	0.001012	0.001011	0.001009	0.001008
98	0.001173	0.001152	0.001133	0.001114	0.001096	0.001078	0.001061	0.001044	0.001028	0.001011	0.001010	0.001008	0.001007	0.001005	0.001004	0.001002	0.001001	0.000999	0.000998
99	0.001180	0.001158	0.001137	0.001116	0.001096	0.001078	0.001060	0.001043	0.001028	0.001013	0.001012	0.001010	0.001009	0.001007	0.001006	0.001004	0.001003	0.001001	0.001000

Figure 2: Expt2_Data.CSV

Now we compare these with our theoretical estimates.

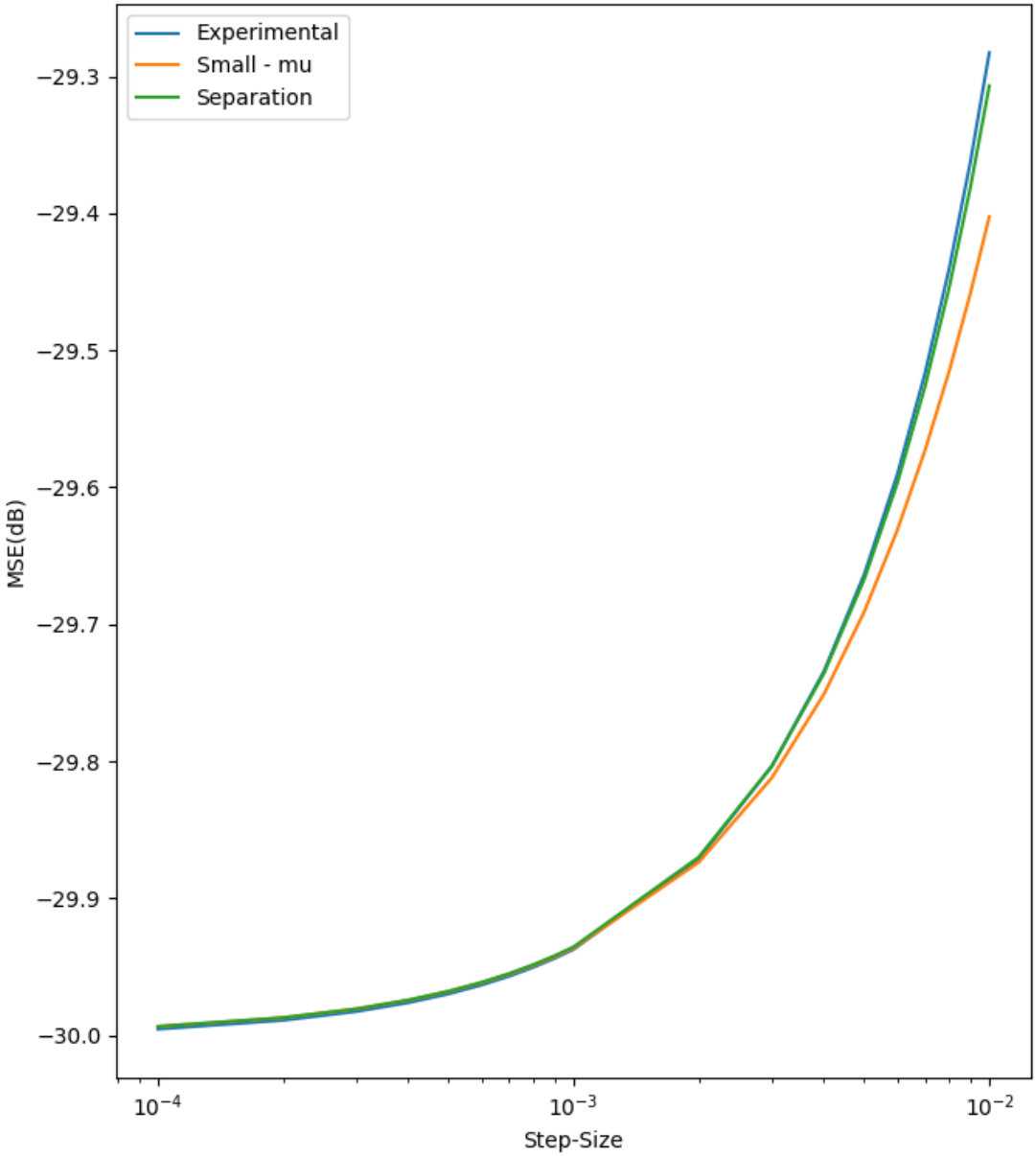

Figure 3: Experimental & Theoretically Estimated Saturation Errors of LMS

3.3 Correlated Gaussian Input with Shift Structure

3.3.1 Data Generation

Similar to the previous simulation, we generate data into binary files for checkpointing the data generation.

```
w = np.array([1.7, -0.5, 1, -0.2, 6, 0.4, 4.5, -1.1, 1.3, -0.2])

a = (0.6 * (0.8**np.arange(0,400000,1)))

U = np.array([]).reshape((0,400009))

for i in range(100):
        with open("expt3_data/expt3_"+str(i)+".bin", "wb") as f:
                g = np.random.normal(0, 1, size=(400000))

                u = np.convolve(a, g)[:400009]
                v = np.random.normal(0,np.sqrt(0.001), size=(400000))

                d = np.convolve(u,w,mode='valid') + v

                pack = (d,u)
                pickle.dump(pack,f)

                print(i, d.shape, u.shape, pack)
```

We then combine them into a single dataset, of shape that is compatible with how our pytorch model can deal with it.

```
D, U = torch.reshape(torch.tensor([]),(0,400000)),torch.reshape(torch.
 ↪tensor([]),(0,400009))
for i in range(100):
        d,u = pickle.load(open("expt3_data/expt3_"+str(i)+".bin","rb"))
        print(d.shape, u.shape, d, u)
        U = torch.cat((U, torch.reshape(torch.from_numpy(u),(1,400009))),0)
        D = torch.cat((D, torch.reshape(torch.from_numpy(d),(1,400000))),0)
        print(D.shape, U.shape, D, U)
pickle.dump((D,U), open("Expt3_data.bin","wb"))
```

3.3.2 LMS with PyTorch Acceleration

The LMS Implementation closely follows the previous Simulation's. Below is how the indices of variables in the program matches up with our familiar notation we've been following for LMS:

$$\begin{bmatrix} pred_i \end{bmatrix} = \begin{bmatrix} u[:,i-9] & u[:,i-8] & \dots & u[:,i] \end{bmatrix} \begin{bmatrix} w[9] \\ w[8] \\ \dots \\ w[0] \end{bmatrix}$$

$$\begin{bmatrix} w[9] \\ w[8] \\ \dots \\ w[0] \end{bmatrix}_{\text{new}} = \begin{bmatrix} w[9] \\ w[8] \\ \dots \\ w[0] \end{bmatrix}_{\text{old}} + \mu(d_i - pred_i) \begin{bmatrix} u[:,i-9] \\ u[:,i-8] \\ \dots \\ u[:,i] \end{bmatrix}$$

Below is the code to define a Shifting Structure Data-Set. This helps us retain rest of the code and still work with a shifting structure. This returns tensors of shape `(batch_size, 100, 10)` and `(batch_size, 100)` respectively

```
class CustomDataset(Dataset):
    def __init__(self, Input, Output):
        self.Input = Input
        self.Output = Output

    def __len__(self):
        return 400000

    def __getitem__(self, idx):
        start_idx = idx
        end_idx = idx + 10
        ret = torch.Tensor(self.Input[:,start_idx:end_idx]).cuda(), torch.
↪Tensor(self.Output[:,idx]).cuda()
        return ret
```

In fact, we can further modify the shapes of the tensors returned here itself so as to make no changes in the rest of our code (instead of having to go change the `forward()` of the model and the SGD Updating Step)

```
        def __getitem__(self, idx):
        start_idx = idx
        end_idx = idx + 10
        u_i = torch.reshape(self.Input[:,start_idx:end_idx],(100,1,10)).
↪cuda()
        d_i = torch.reshape(self.Output[:,idx],(100,1,1)).cuda()
        return u_i, d_i
```

3.3.3 Results

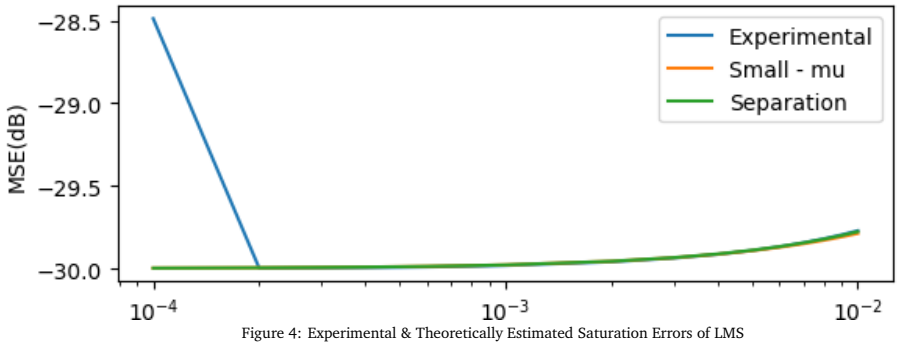

Figure 4: Experimental & Theoretically Estimated Saturation Errors of LMS

Notice here how the first data-point is not matching with the theoretical prediction. We can only infer that the number of iterations is not sufficient for the model to converge and saturate. We can roughly say that if the actual weights are farther from the weights the model is initialized to and if the step size is smaller, the number of iterations required is larger

To get a better visualization of the data-points that actually converge and saturate, we can plot the graph without first data-point.

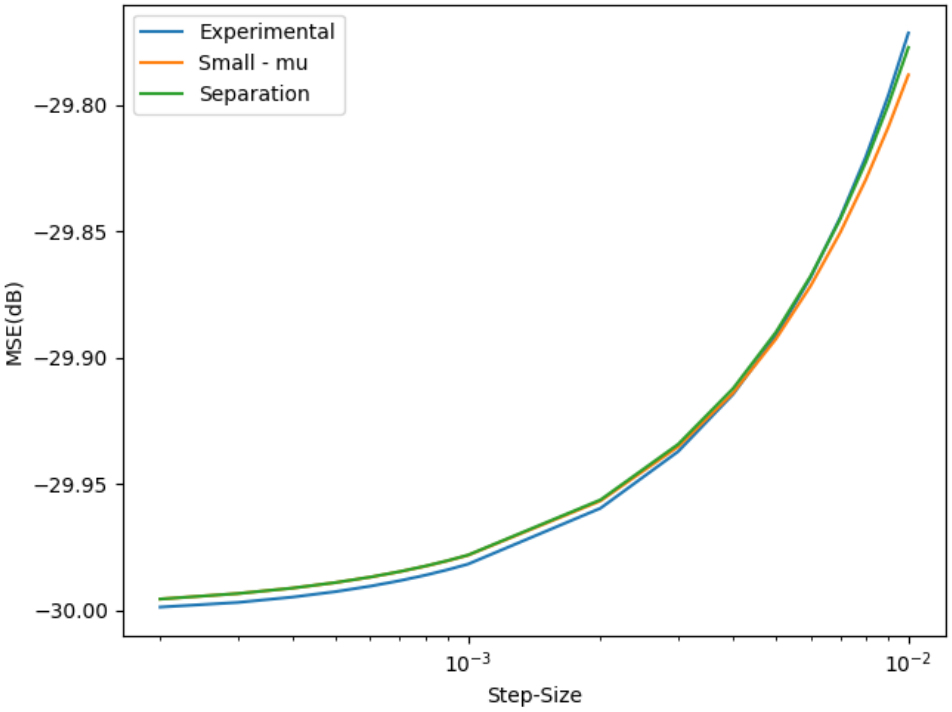

Figure 5: Experimental & Theoretically Estimated Saturation Errors of LMS

3.4 Inferences

We can observe that for both the data-sets, both the approximations match fairly well for small values of μ. For larger value the Experimental error starts to increase more than the small-μ approximation, but matches fairly well against the Separation approximation