ESIGELEC

École supérieure d'ingénieurs en génie électrique

Master's in Electronic EmbeddedSystems (EES2020)

SubmittedBy

Mr. Sandeep Anand
Mr. Madhu Ameneni
Mr. Khangesh Matte

Laboratory Report corresponding to the subject of Smart sensors taught by
Professor. Dr. Maciej ZAWODNIOK

ROUEN – 2020

LAB 1.

Objective: Understand how aNon-linear adaptive pulse coded modulation-based compression (NADPCMC) works, using the software Octave or Mathlab, and how the parameters can influence the estimates and the error.

Development: For this assignment, we used the code below, provided by the lecturer and we commented all the code to make sure that each programming line understanding is correct. We also plot the receiver part, changing the code to verify the final process.

The first block from line 29 to 24 of the Matlab/Octave script is to set the signal and its characteristics, which in this case is a sinusoidal signal whose amplitude is A=127 (predefined by the Engineer andcorresponding to 8 bits),with maximum error of 20% , number of samples equal to 31.

The second block initializes the loop variables ,the Parameter alpha is calculated by (1/|phi|^2) with alpha_max (maximum value of alpha) corresponding to (1/A^2) and kv_max (maximum value of the gain) calculated in function of alpha and the amplitude A.We then defined the number of transmission bits of our actual sample, we calculated its minimum and maximum quantization errors.The first sample Y(k) is then initialized, without errors and the next samples y(k+1) are thus initialized with quantized errors and processed through if conditional statementsand sent.The last block is for plotting the signal and ending the function.

During this labs, the values analyzed more closely are: kv, alpha, scale, A and tx_bytes.

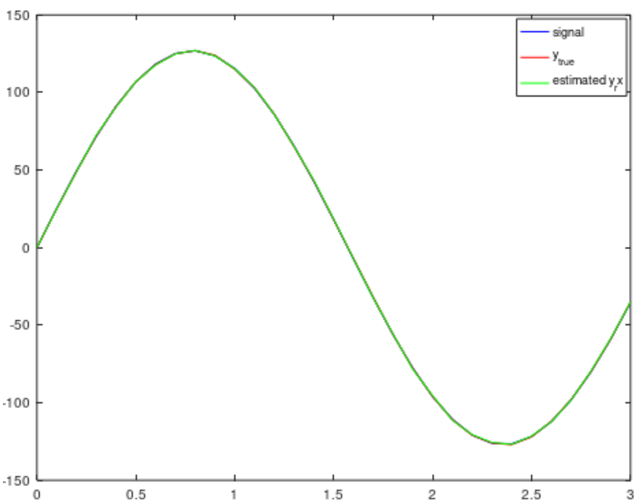
```
    A=127;
    twenty_percent = 0.2 * (2*A)
    t=[0:0.1:3];
    max_k=size(t, 2)
scale1=2.0;
    s1=A*sin(t*scale1);


    alpha=0.0005
    alpha_max = 1/A^2
    kv_max = 1/ sqrt(1/(1-alpha*A^2))
 kv = 0.0016

    tx_bits = 5;
    TX_MAX = 2^(tx_bits-1)-1; TX_MIN = -(2^(tx_bits-1));
    y_true = round (s1);

    phi = y_true;
    alpha_max_from_data = 1/ (max(phi)^2)

    y(1) = y_true(1);
    y_true(max_k+1) = y_true(max_k);
theta(1) = 1;
    e(1)=0 ;
    e_tx(1)=0;

)
for k=1:max_k

        y(k+1) = theta(k)*phi(k) + kv*e_tx(k);
e(k+1) = y_true(k+1) - y(k+1);
```

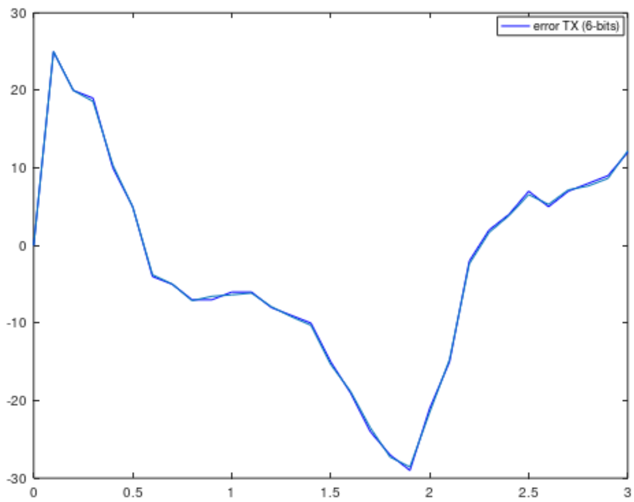
```
        e_tx(k+1) = round (e(k+1));
if e_tx(k+1)>TX_MAX
            e_tx(k+1) = TX_MAX;
end
if e_tx(k+1)<TX_MIN
            e_tx(k+1) = TX_MIN;
end
        phi(k+1)=y(k+1)+e_tx(k+1); % past value (from previous sample - needed on both
sides)
% Update/adapt theta coefficient to improve estimation in the future:
        theta(k+1) = theta(k) + alpha*phi(k)*e_tx(k+1);
end

%(assume first sample is sent exactly)

y_rx = y + e_tx;

%Plot results
figure(2)
plot(t, s1(1:max_k), 'b', t, y_true(1:max_k), 'g', t, y(1:max_k), 'r' );
legend show
figure(3)
leg_info_1 = sprintf('g', tx_bits)
plot(t, e_tx(1:max_k), leg_info_1, t, e(1:max_k))
legend show
figure(4)
plot(t, s1(1:max_k), 'g', t, y_rx(1:max_k), 'r' );
legend show
```

Interpretation:

1. Implement NADPCMC scheme in Matlab. Parametrize the initialization vector size and number of bits used for encoding. Show result in terms of error in reproduction of the data at the received AND the amount of data needed to transmit.

-The firstanalysis was done using the values of alpha: 0.00005, kv = 0.16, tx_bits = 5, scale = 2.0.

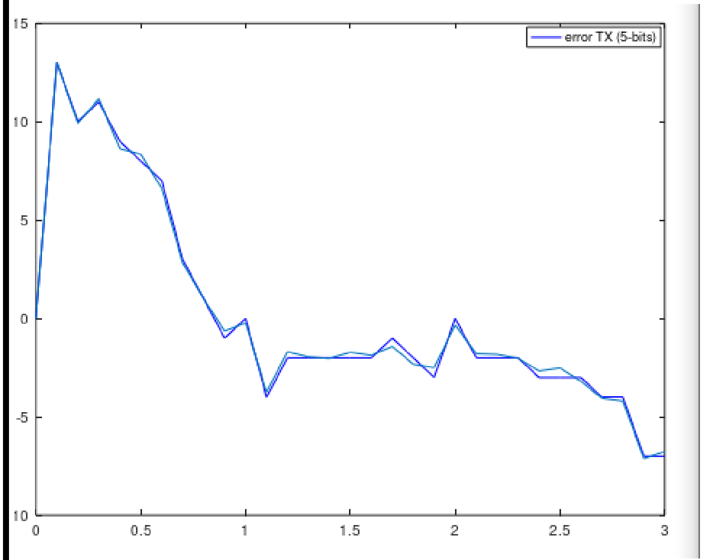


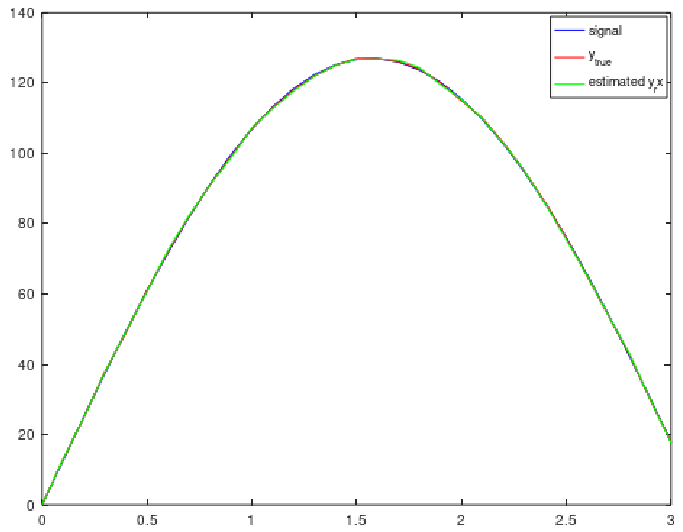


.

2.Create a slowly varying sensor data (e.g. slow sinusoid). Evaluate the performance (reconstruction error) with different values for length of initialization vector and size of error value transmitted (number of bits).

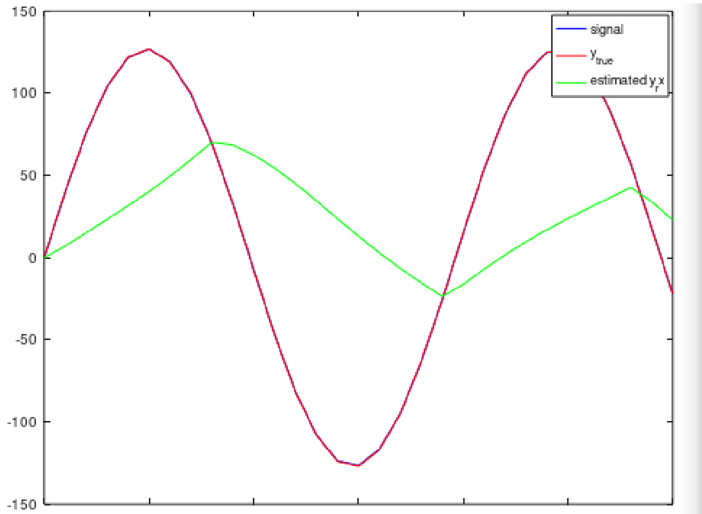
alpha = 0.00005, kv = 0.16, tx_bits = 5, scale = 1.0, and A = 127. The first graph obtained was:

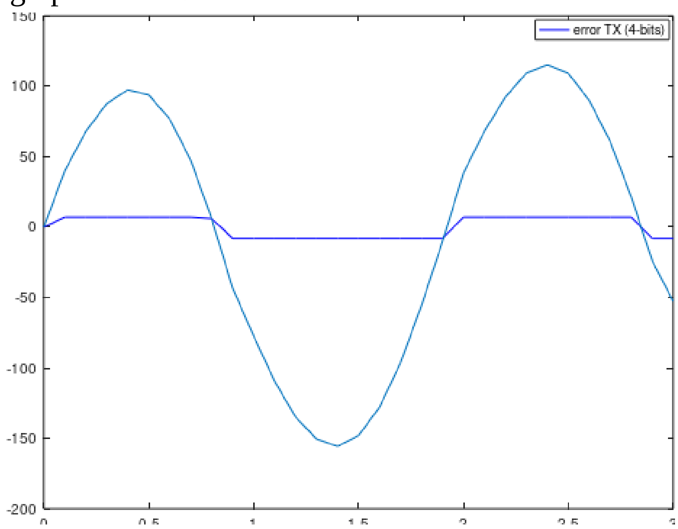




- After changing the scale for 3.2, we can see that the gap between the signal and the estimatorreduces and the period of the signal is larger.The sensor is be more accurate.

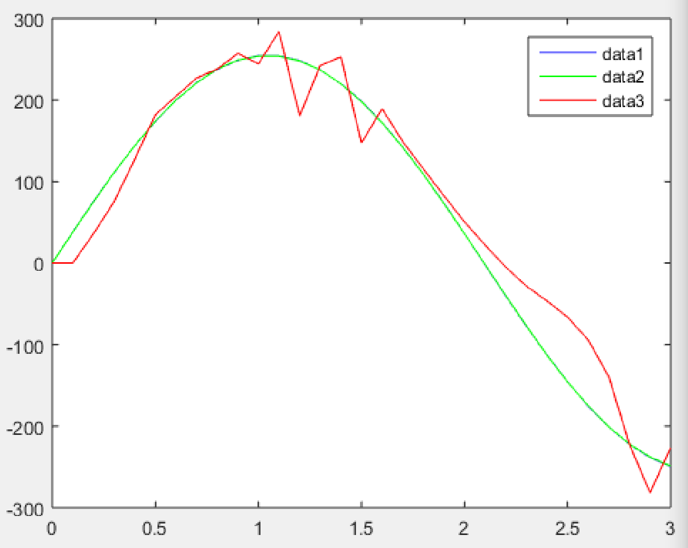
- Changing the number of bits to 4 t we obtained the graphs below:

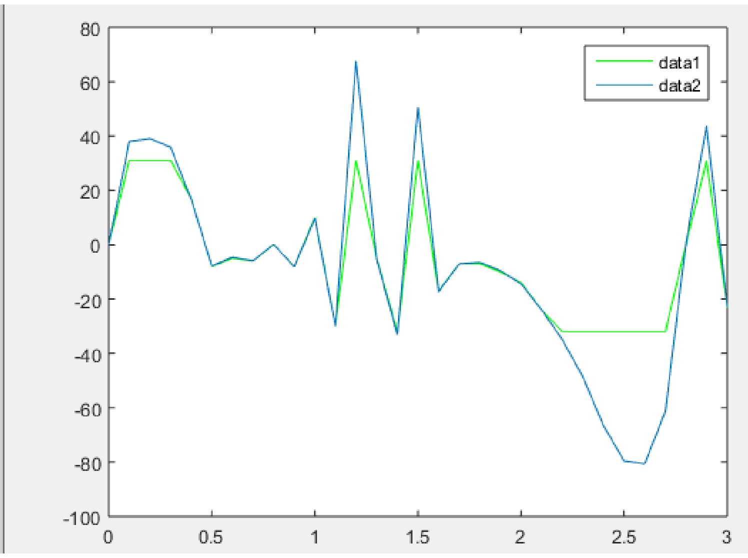




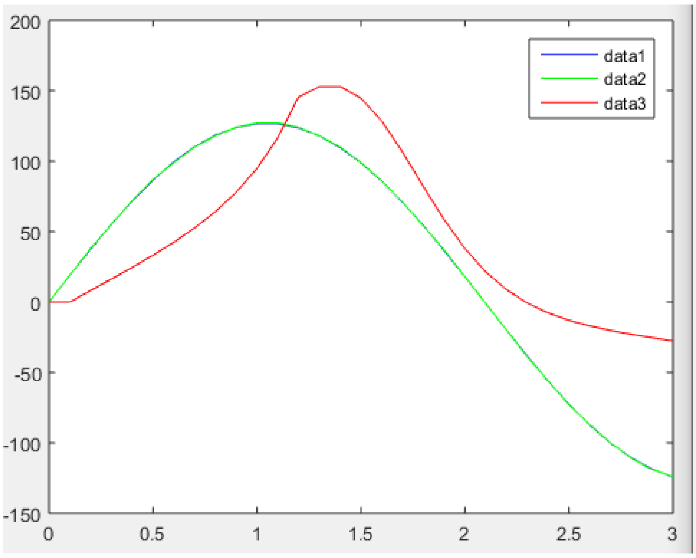
- The maximum value of the error happens in the first moment, once we assume theta value to start the process. Once we change the tx_bits, the estimate value starts to distance from the signal. It happens because the error value exceeds the value that the number of transmitted bits can represent, and this cause the failure of the system.

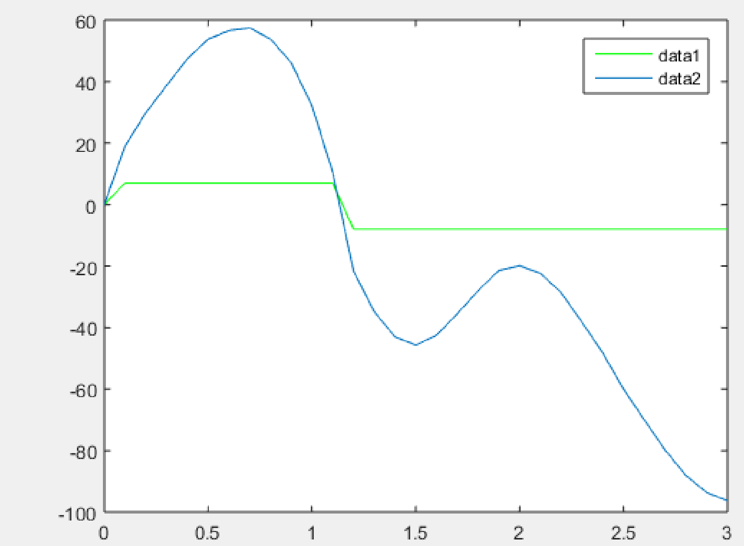
- Using a length of 255 the error get bigger, and if the value of tx_bits is not big enough to reproduce the error, we will have a saturation:





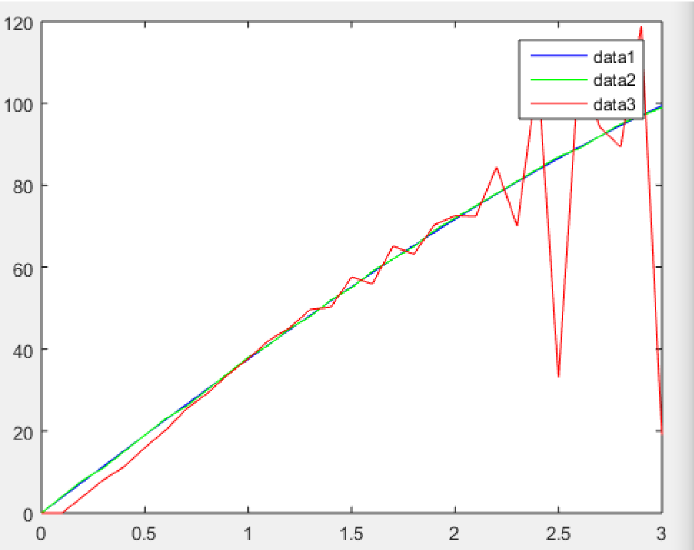
- It means that if we reduce the length we may obtain a less error and we can reduce the tx_bits without having a saturation, as show the graphs below using A = 64.

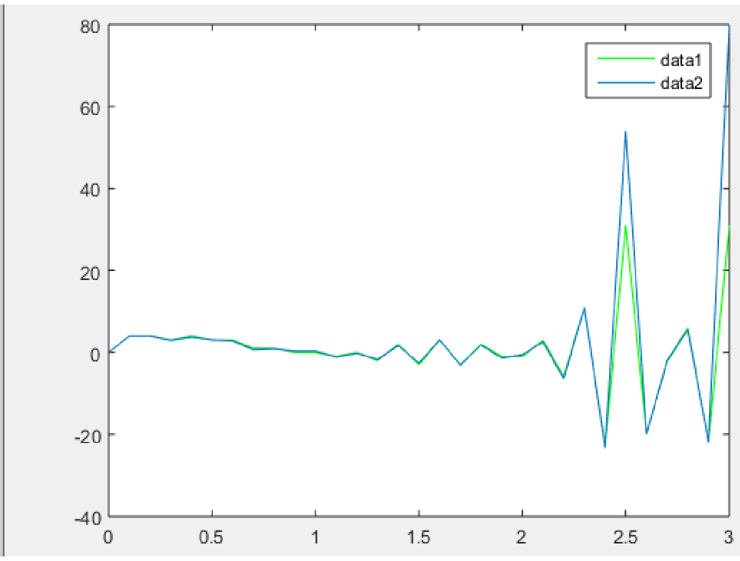




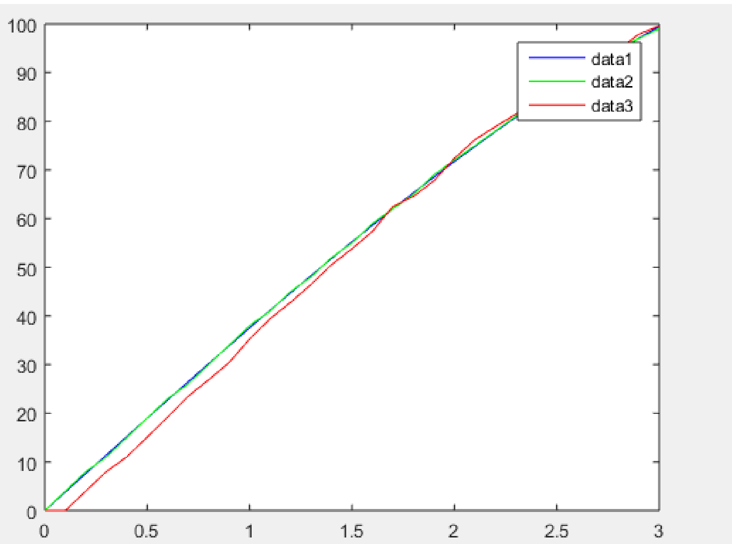
3. Create increasingly faster-changing sensor data, until the error of the sensor data reproduction becomes larger then 20%.

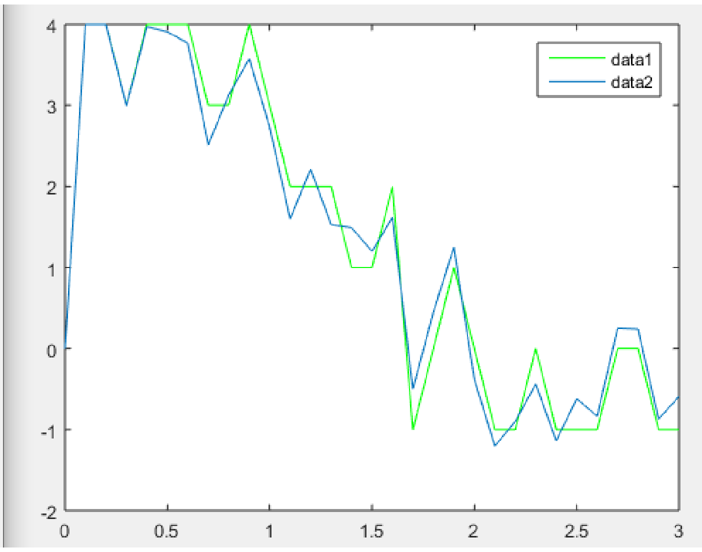
-To create a faster-changing sensor data we need to change the scale and with this, increase the frequency of the signal. Still using kv = 0.001, alpha = 0.0005, tx_bits = 6 but now using scale = 0.3:



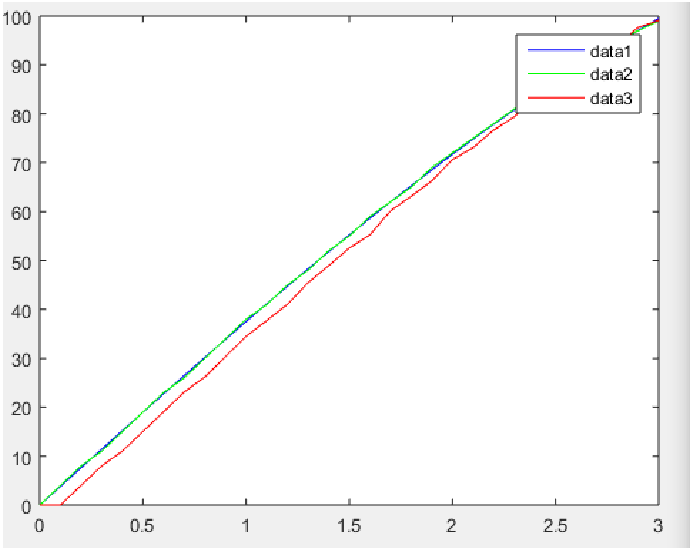


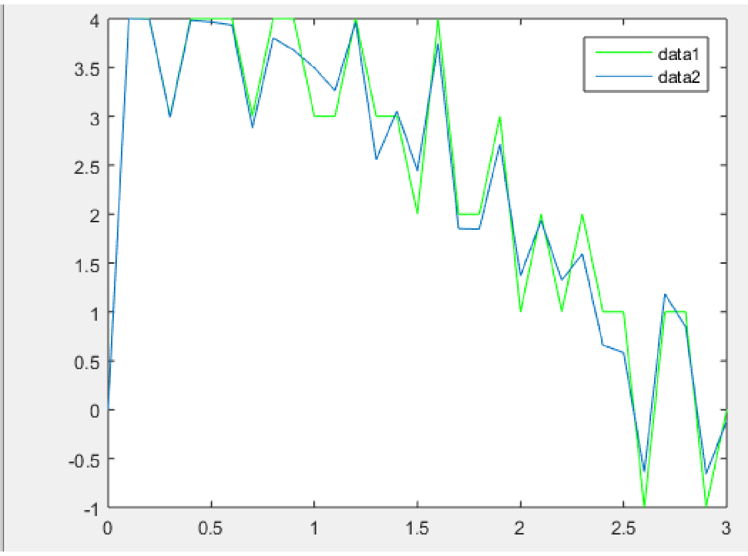
-With the graphs we can see that the increase of the scale make the estimate value less accurate, and the measures start to distance from the estimate value. Increase the value of tx_bits this time will make the results get worse and decrease the value of kv will not have significant impacts. To correct this, we can change alpha to a lower value (in this case, 0.0005), with this, we can be more accurate as the next graph shows:



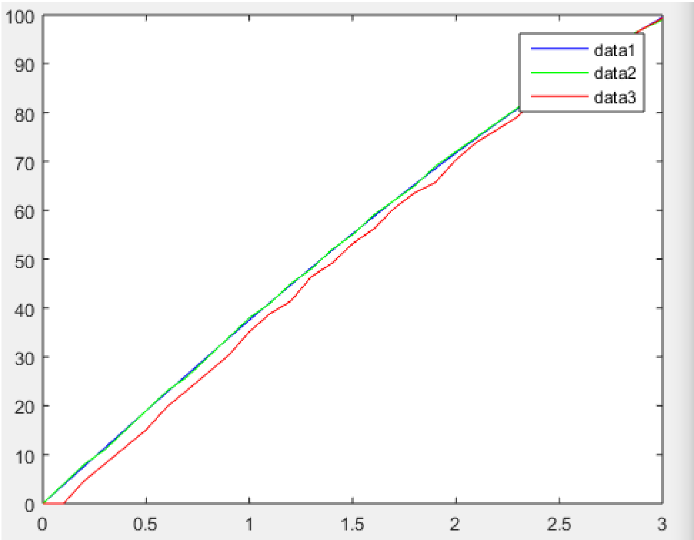


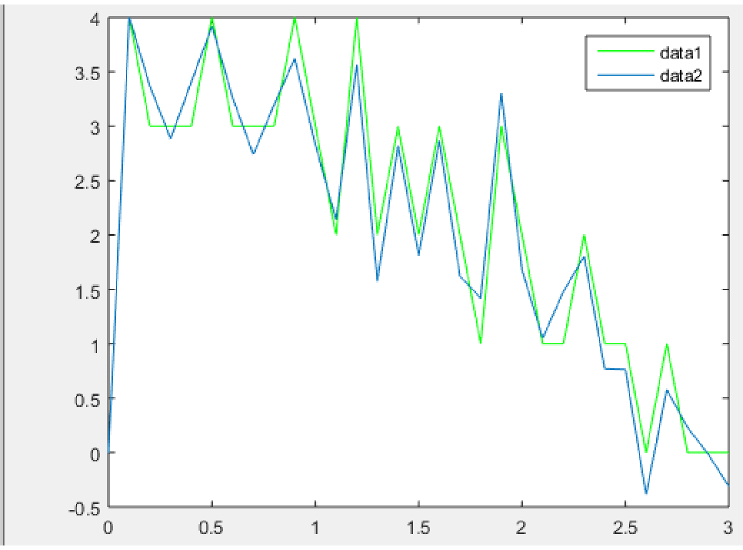
-But more this value get lower, more time will be needed to have an accurate measure and more time the error graph have a high value, as the graph shows when we changed the alpha to 0.00001:



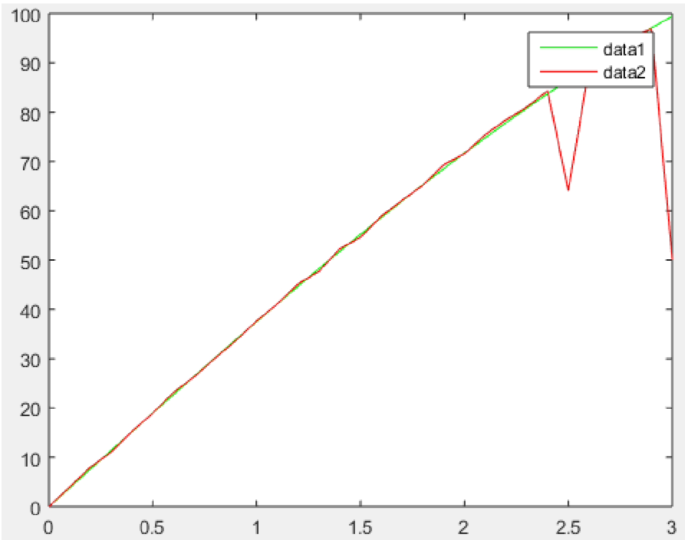


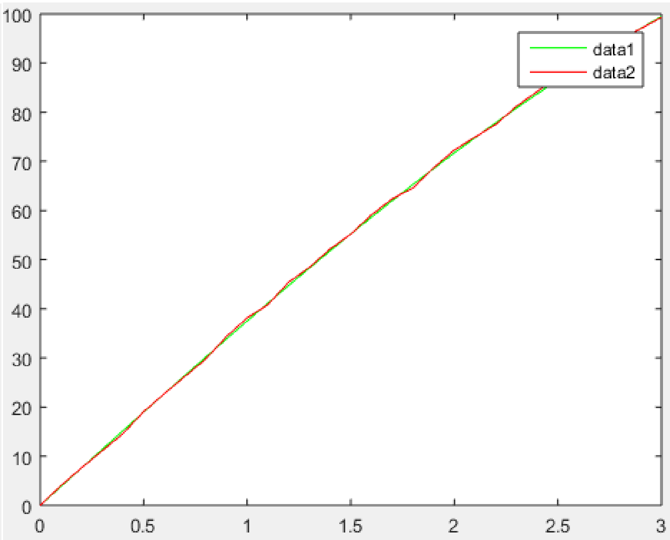
- To correct this, we can increase the value of kv, from 0.001 to 0.159 . The importance of it is to correct the error in advance and it can be noticed on the next graphs:





-With this corrections we can change the scale for any value without a divergent system and appropriating the values to the lower error as possible, without have to change in tx_bits. Receiver side before and after the correction:





LAB 3. Introduction to ML/AI with PyTorch

Task 1. Run sample code to get familiarized with the PyTorch.

Explanation: The sample program "Task1.py" is been executed on the anaconda terminal.

Note A : According to our analysis the input data provided are the student study hour and the sleeping hours and the output data is students marks obtained from the exams.

As shown in figure 1: the predicted output is 0.92 for the Input scaled data : ([1.3333, 0.8889]) with loss of -0.067742.

Output :

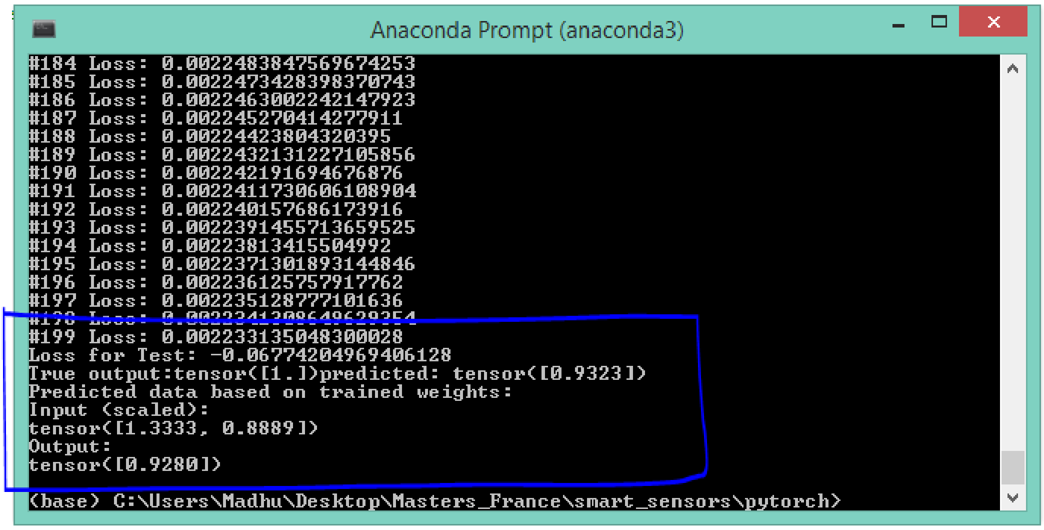


Figure : 1

Note: Task1.py is attached to the document.

Task 2. Modify the example PyTorch script – each time rerun script with training and analyze result.

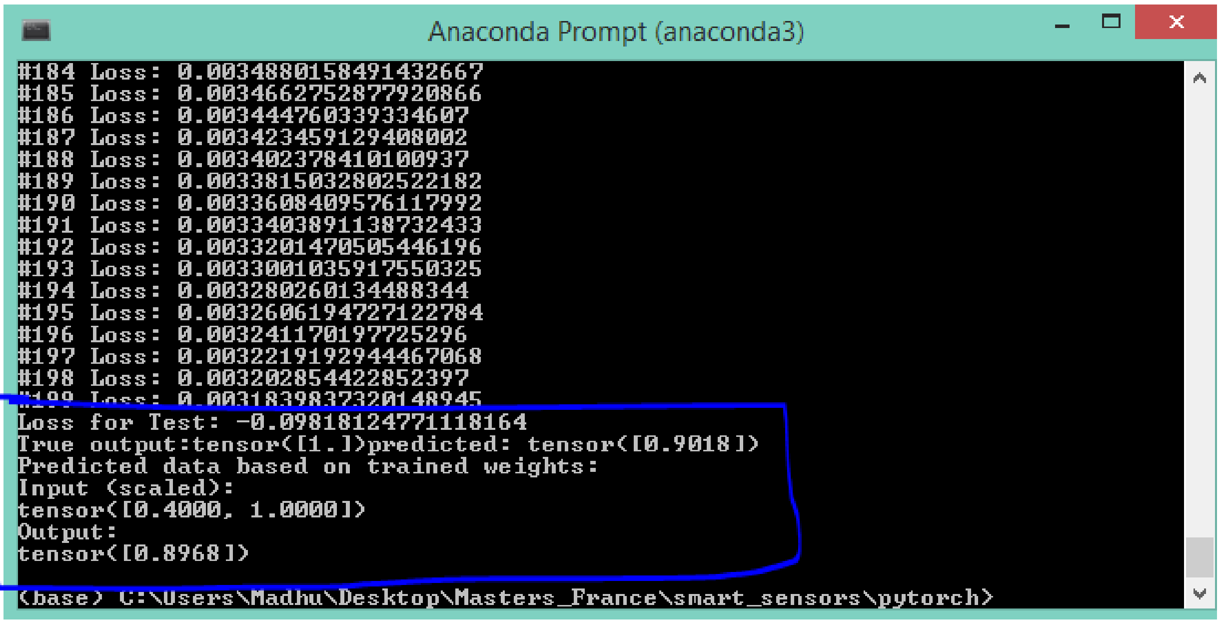
A. Modify size of input and output data.

Note : 2 ways the size of Input and Output is been done which has been explained as below.

Explanation 1: The input array size is been increased to 5 * 2 matrix and Output has 5*1.File "" is been executed on the terminal.

As shown in fig : 2.1(a) the predicted output is 0.8968 for the Input scaled data : ([0.4000,1.000]) with loss of -0.098.

Output :



```
Anaconda Prompt (anaconda3)
#184 Loss: 0.0034880158491432667
#185 Loss: 0.0034662752877920866
#186 Loss: 0.003444760339334607
#187 Loss: 0.003423459129408002
#188 Loss: 0.003402378410100937
#189 Loss: 0.0033815032802522182
#190 Loss: 0.0033608409576117992
#191 Loss: 0.0033403891138732433
#192 Loss: 0.0033201470505446196
#193 Loss: 0.0033001035917550325
#194 Loss: 0.003280260134488344
#195 Loss: 0.0032606194727122784
#196 Loss: 0.003241170197725296
#197 Loss: 0.0032219192944467068
#198 Loss: 0.003202854422852397
#199 Loss: 0.0031839837320148945
Loss for Test: -0.09818124771118164
True output:tensor([1.])predicted: tensor([0.9018])
Predicted data based on trained weights:
Input (scaled):
tensor([0.4000, 1.0000])
Output:
tensor([0.8968])
(base) C:\Users\Madhu\Desktop\Masters_France\smart_sensors\pytorch>
```

Fig: 2.1(a)

Note: TaskA2.1.py is attached to the document and Assuming same as Note A.

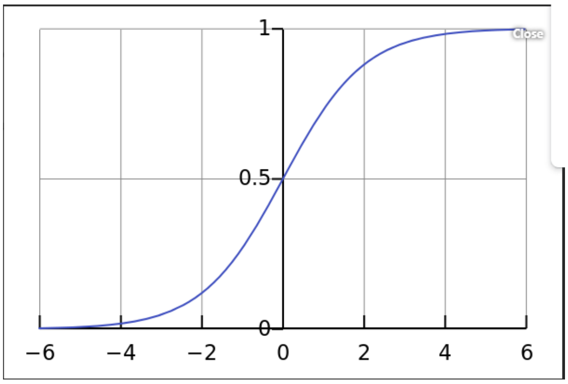
Explanation 2: A new characteristics is been added to the input size as games and output size as percentile.

As shown in fig : 2.1(b) the predicted output is [0.9367,0.5185] for the Input scaled data : ([0.6667,1.000,1.0000]) with the loss of [-0.0672,-0.0458] and the activation used is **Sigmoid.**

Sigmoid : The output is defined by following equation:

y(x)=11+e−x

and it looks like following **figure:**



It is differentiable, non-linear.
Pros: it is differentiable, non-linear, produces non-binary activations and it is bounded between (0,1).
Cons: vanishing gradients. Yes, the values are between 0 & 1 and it will be zero when value of the activation reaches 0 or 1 (the horizontal part of the curve).

Output :

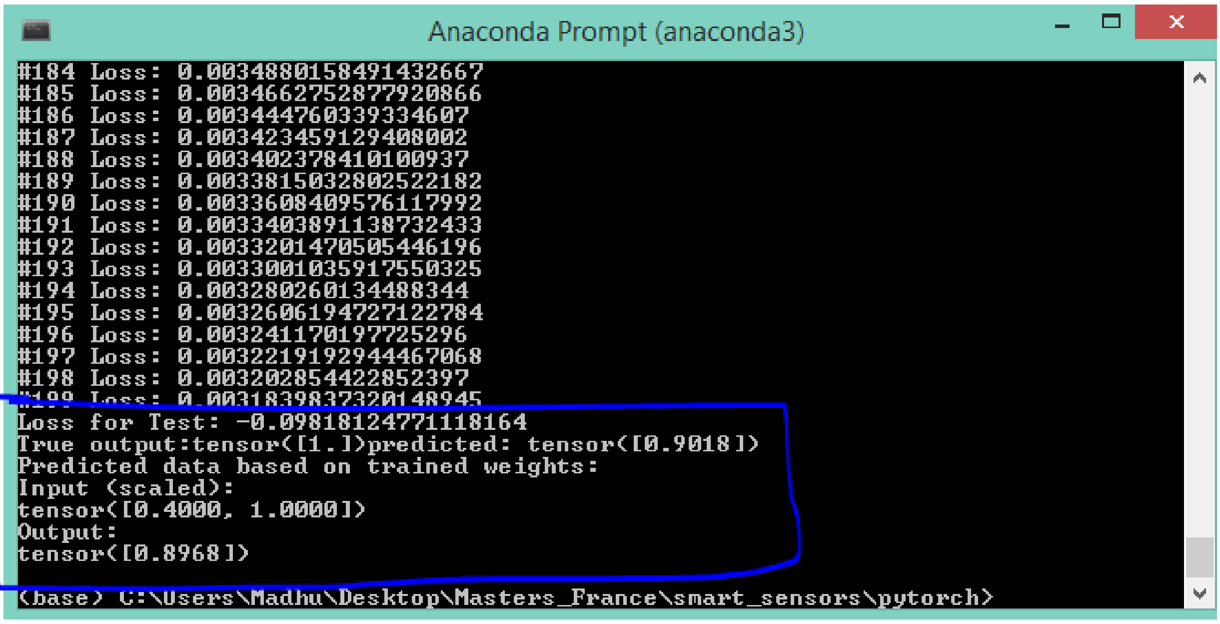


Fig: 2.1(b)

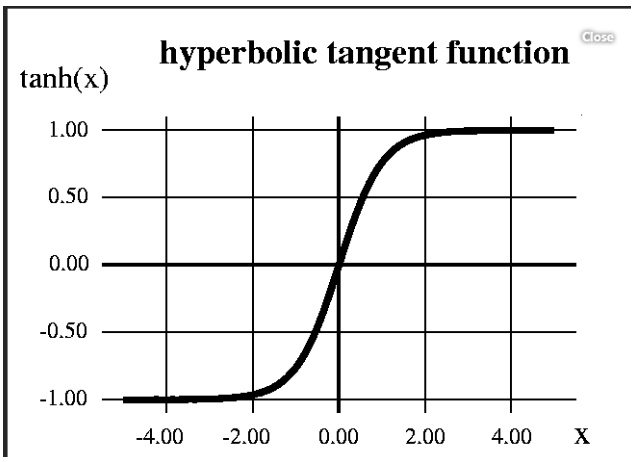
Note: TaskA2.2.py is attached to the document.

As shown in fig : 2.1(b) the predicted output is [[0.8968] for the Input scaled data : ([([0.4000,1.000]) with the loss of [-0.0981] and the activation used is **Tanh.**

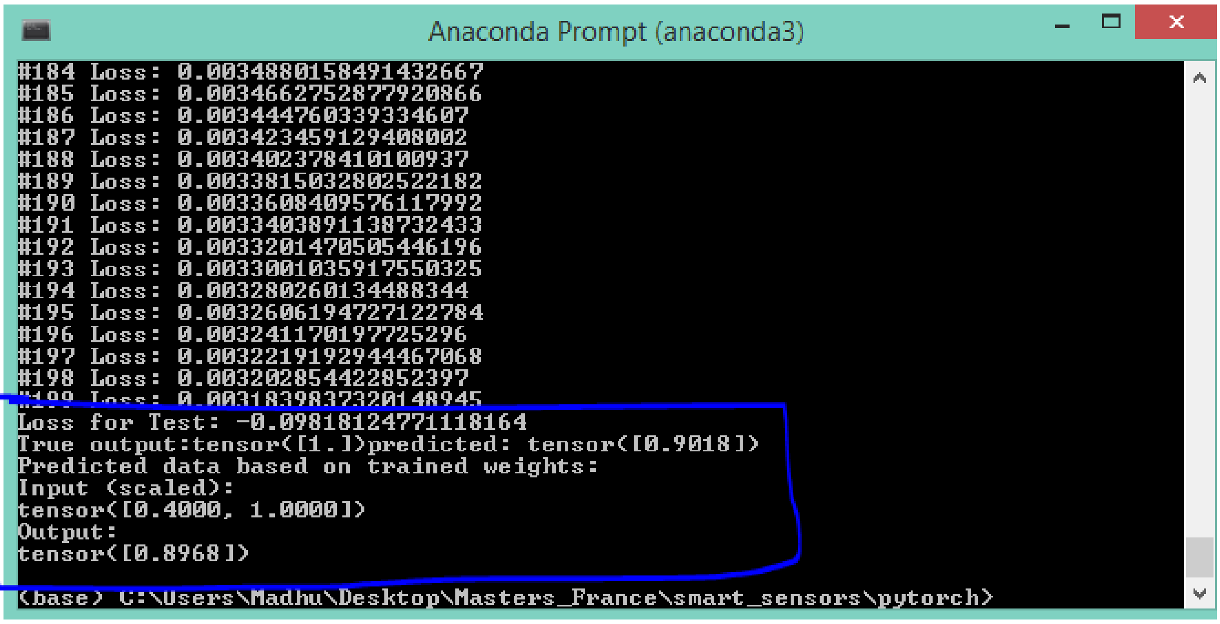
tanh: these are the two basic and former functions used in neutral networks. They have several advantage for a small network and many disadvantage for large or deep networks. The activation is following given input x:x:

$$\boxed{y(x) = \frac{1-e^{-2x}}{1+e^{-2x}}}$$

and it looks like following **figure:**



Output :



```
Anaconda Prompt (anaconda3)
#184 Loss: 0.0034880158491432667
#185 Loss: 0.0034662752877920866
#186 Loss: 0.003444760339334607
#187 Loss: 0.003423459129408002
#188 Loss: 0.003402378410100937
#189 Loss: 0.0033815032802522182
#190 Loss: 0.0033608409576117992
#191 Loss: 0.0033403891138732433
#192 Loss: 0.0033201470505446196
#193 Loss: 0.0033001035917550325
#194 Loss: 0.003280260134488344
#195 Loss: 0.0032606194727122784
#196 Loss: 0.003241170197725296
#197 Loss: 0.0032219192944467068
#198 Loss: 0.003202854422852397
#199 Loss: 0.0031839837320148945
Loss for Test: -0.09818124771118164
True output:tensor([1.])predicted: tensor([0.9018])
Predicted data based on trained weights:
Input (scaled):
tensor([0.4000, 1.0000])
Output:
tensor([0.8968])

(base) C:\Users\Madhu\Desktop\Masters_France\smart_sensors\pytorch>
```

Fig: 2.1(c)

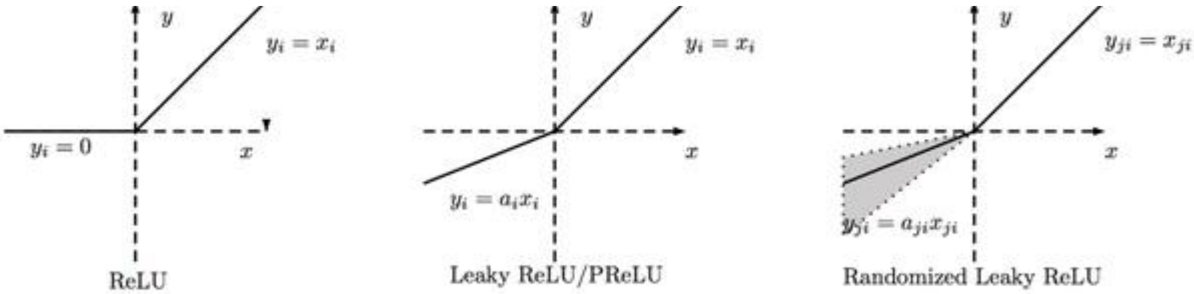
B. Choose different activation function or number of neurons.

Explanation : 2 activation function is been used for the “Task1.py” they are ReLu and Tanh.

As shown in fig : 2.2(a) the predicted output is [0.] for the Input scaled data : [1.3333,0.8889]) with the loss of [-1.0] and the activation used is **Relu.**

Relu ;

It is very popular from the recent advances in deep learning research. The function is simply y(x)=max(0,x).y(x)=max(0,x).



Pros: as it is clear from the figure that any variant of Relu doesn’t have a flat curve, it avoids vanishing gradient problem.

Cons: it is not differentiable at 0 and may result in exploding gradients.

To resolve problems with ReLu, leaky relu was proposed which is differentiable at 0 (as shown in the figure).

Output :

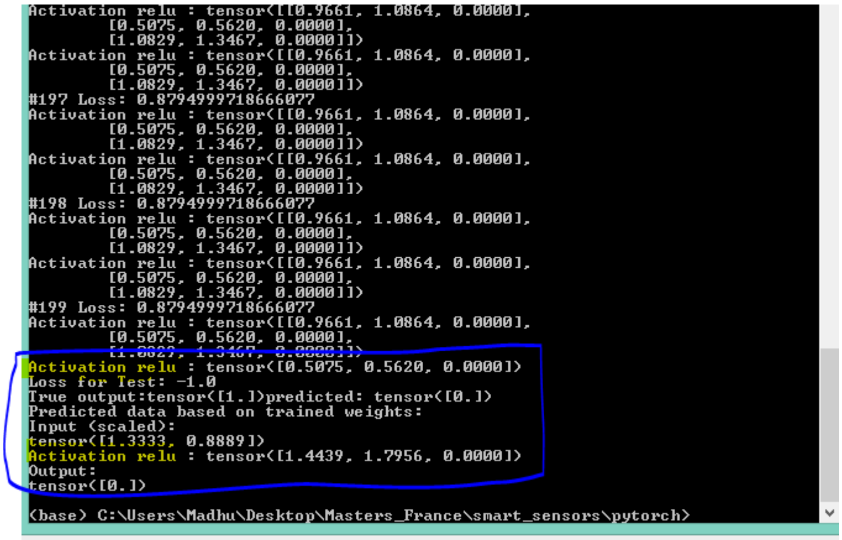


Fig 2.2(a)

Note: TaskB2.1.py is attached to the document.

As shown in fig : 2.2(b) the predicted output is [0.] for the Input scaled data : ([1.333,0.8889]) with the loss of [-1.0] and the activation used is **TanH.**

The Tanh loss is very less when compare to Relu and Sigmoid.

Output :

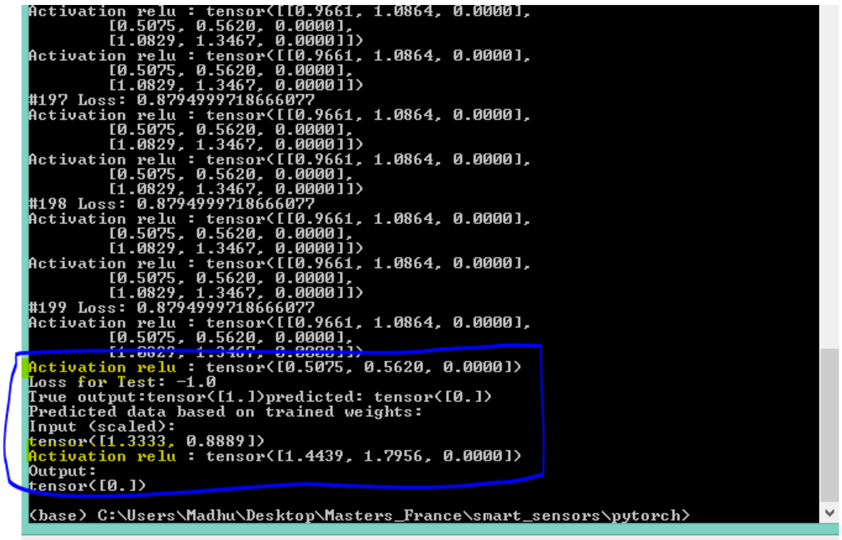


Fig 2.2(b)

Note: TaskB2.2.py is attached to the document.

Task 3. Process provided vector data using NN of your design and evaluate performance on subset of training and test data.

Explanation : A new network is been designed based on the vector data provided a downsampling is been done in the sequential function with Tanh activation function and linear function is used as shown below.

```
self.seq = torch.nn.Sequential(torch.nn.Linear(self.inputSize, self.hiddenSize), \
                   torch.nn.Tanh(), \
                   torch.nn.Linear(self.hiddenSize, 8000),\
                   torch.nn.Tanh(), \
                   torch.nn.Linear(8000, 6000), \
                   torch.nn.Tanh(), \
                   torch.nn.Linear(6000, 4000), \
                   torch.nn.Tanh(), \
                   torch.nn.Linear(4000, self.outputSize),)
```

And the below graph of fig 3.1 represents the blue line as the final value of Y and the orange value as the predicted data.

Output :

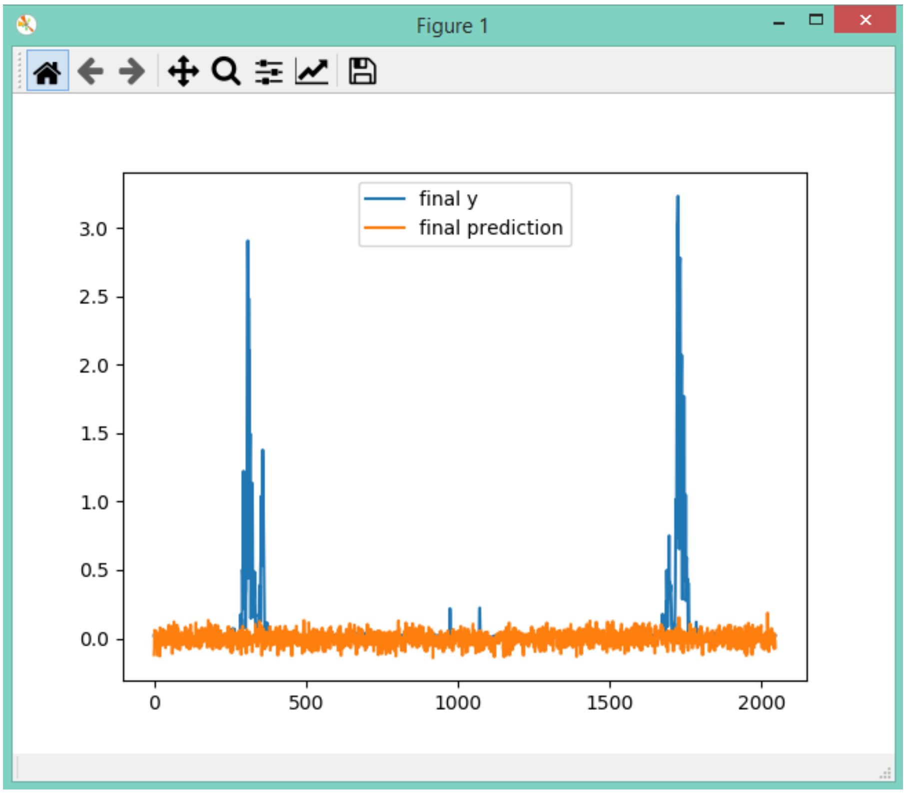


Fig 3.1

Note: Task3.py is attached to the document and **as my system is hanging a lot the iteration is been limited to 100.**

As shown in fig : 3.2 the predicted output is [0.0.0211,0.0098,0.0078] for the Input scaled data : ([([-0.1201,-0.0709,0.0098,0.0078]) with the loss of [0.08245344460010529]**.**

Output :

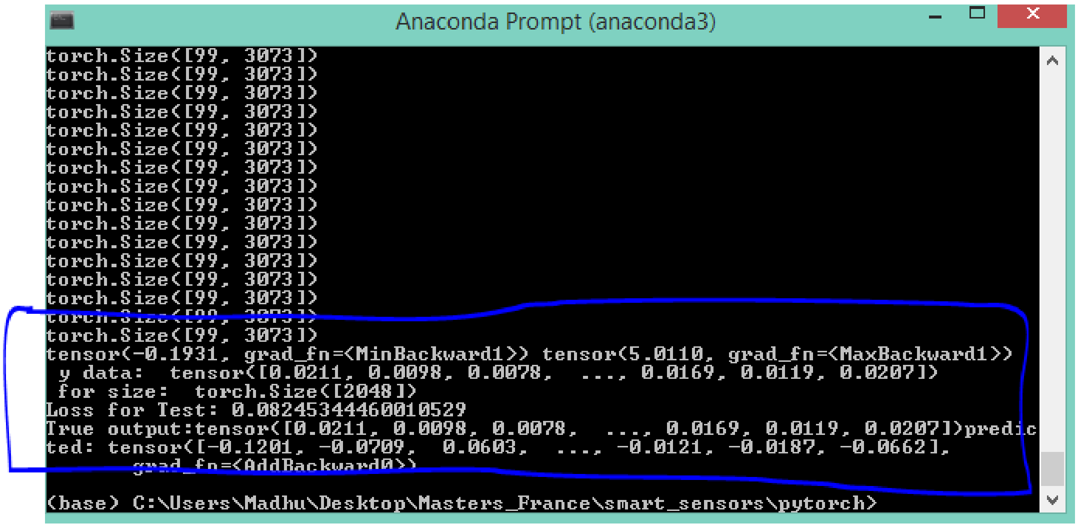


Fig 3.2

Task 4. Tweak some NN parameters and identify if it helped or not with the NN performance.

Parameter 2:

Explanation : A new network is been designed based on the vector data provided a down sampling is been done in the sequential function with Tanh activation and linear function and a Momentum and lr optimizer is used as shown below.

```
self.seq = torch.nn.Sequential(torch.nn.Linear(self.inputSize, self.hiddenSize), \
                   torch.nn.Tanh(), \
                   torch.nn.Linear(self.hiddenSize, 8000),\
                   torch.nn.Tanh(), \
                   torch.nn.Linear(8000, 6000), \
                   torch.nn.Tanh(), \
                   torch.nn.Linear(6000, 4000), \
                   torch.nn.Tanh(), \
                   torch.nn.Linear(4000, self.outputSize),)
        print("madhu" , self.seq)
        self.criterion = torch.nn.MSELoss(reduction='mean')
        #self.optimizer = torch.optim.SGD(self.seq.parameters(), lr=1e-4)
        # Another optimizer setup - with MOMENTUM for non-trivial data/relationship
        self.optimizer = torch.optim.SGD(self.seq.parameters(), lr=1e-4, momentum=0.9)
```

And the below graph of fig 3.3 represents the blue line as the final value of **Y which is the ground truth value** and the orange value as the predicted data.

Momentum: it is considered to be a alpha constant value which is **0.9** and **ls = 1e-4.This will help in preventing oscillation of data when the weights are updated during training.**

Output :

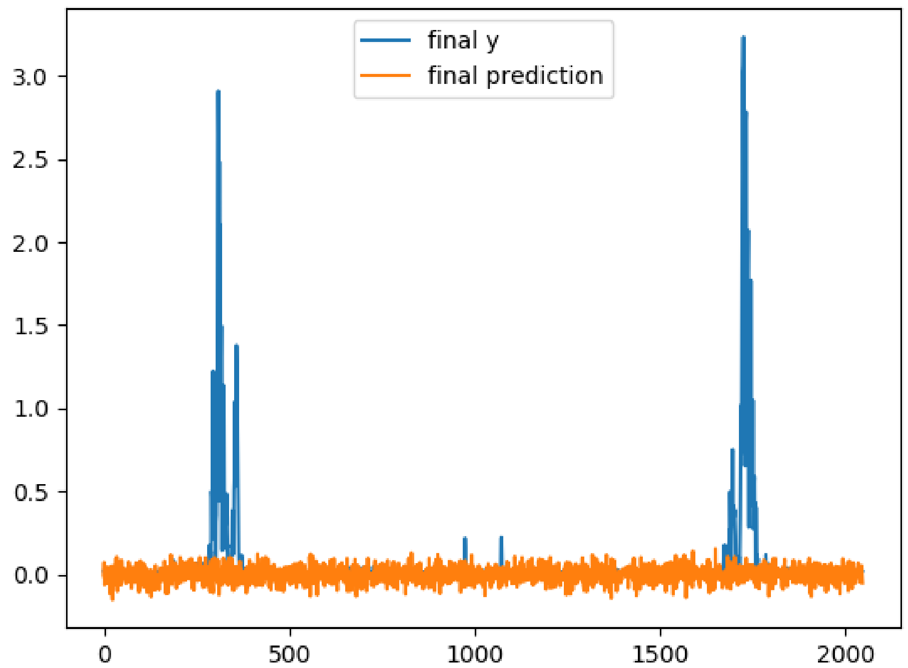
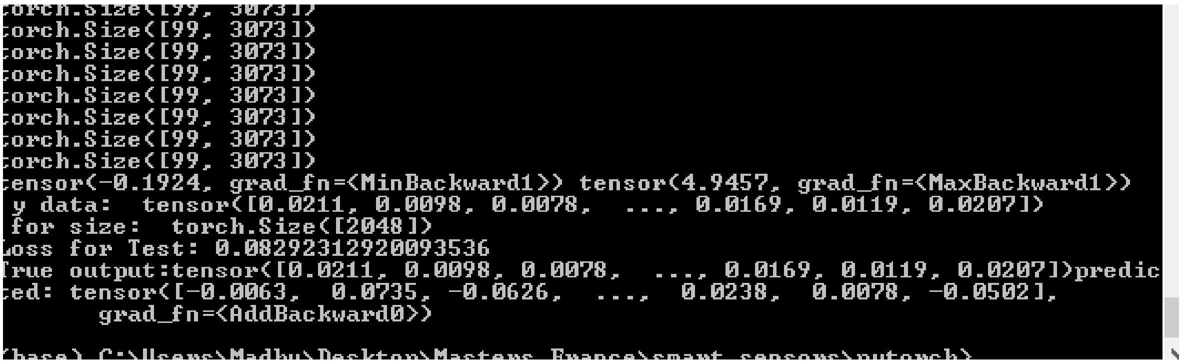


Fig 3.3

Note: Task4.py is attached to the document and **as my system is hanging a lot the iteration is been limited to 100.**

As shown in fig : 3.4 the predicted output is [0.0211,0.0098,0.0078] for the Input scaled data : ([0.0211,0.0098,0.0078]) with the loss of [0.082923129200]**.**

Output :



```
torch.Size([99, 3073])
torch.Size([99, 3073])
torch.Size([99, 3073])
torch.Size([99, 3073])
torch.Size([99, 3073])
torch.Size([99, 3073])
torch.Size([99, 3073])
torch.Size([99, 3073])
tensor(-0.1924, grad_fn=<MinBackward1>) tensor(4.9457, grad_fn=<MaxBackward1>)
 y data:  tensor([0.0211, 0.0098, 0.0078,   ..., 0.0169, 0.0119, 0.0207])
 for size:  torch.Size([2048])
Loss for Test: 0.08292312920093536
True output:tensor([0.0211, 0.0098, 0.0078,   ..., 0.0169, 0.0119, 0.0207])predic
ted: tensor([-0.0063,  0.0735, -0.0626,   ...,  0.0238,  0.0078, -0.0502],
       grad_fn=<AddBackward0>)

(base) C:\Users\Madhu\Desktop\Masters France\smart sensors\pytorch>
```

Fig 3.4

Parameter 2:

Explanation : A new network is been designed based on the vector data provided a downsampling is been done in the sequential function with Sigmoid activation and linear function and a SGD optimizer is used as shown below.

```
self.seq = torch.nn.Sequential(torch.nn.Linear(self.inputSize, self.hiddenSize), \
                    torch.nn.Sigmoid(), \
                    torch.nn.Linear(self.hiddenSize, 8000),\
```

```
                    torch.nn.Sigmoid(), \
                    torch.nn.Linear(8000, 6000), \
                    torch.nn.Sigmoid(), \
                    torch.nn.Linear(6000, 4000), \
                    torch.nn.Sigmoid(), \
                    torch.nn.Linear(4000, self.outputSize),)
        print("madhu" , self.seq)
        self.criterion = torch.nn.MSELoss(reduction='mean')
        self.optimizer = torch.optim.SGD(self.seq.parameters(), lr=1e-4)
```

And the below graph of fig 3.4 represents the blue line as the final value of **Y which is the ground truth value** and the orange value as the predicted data.

Output :

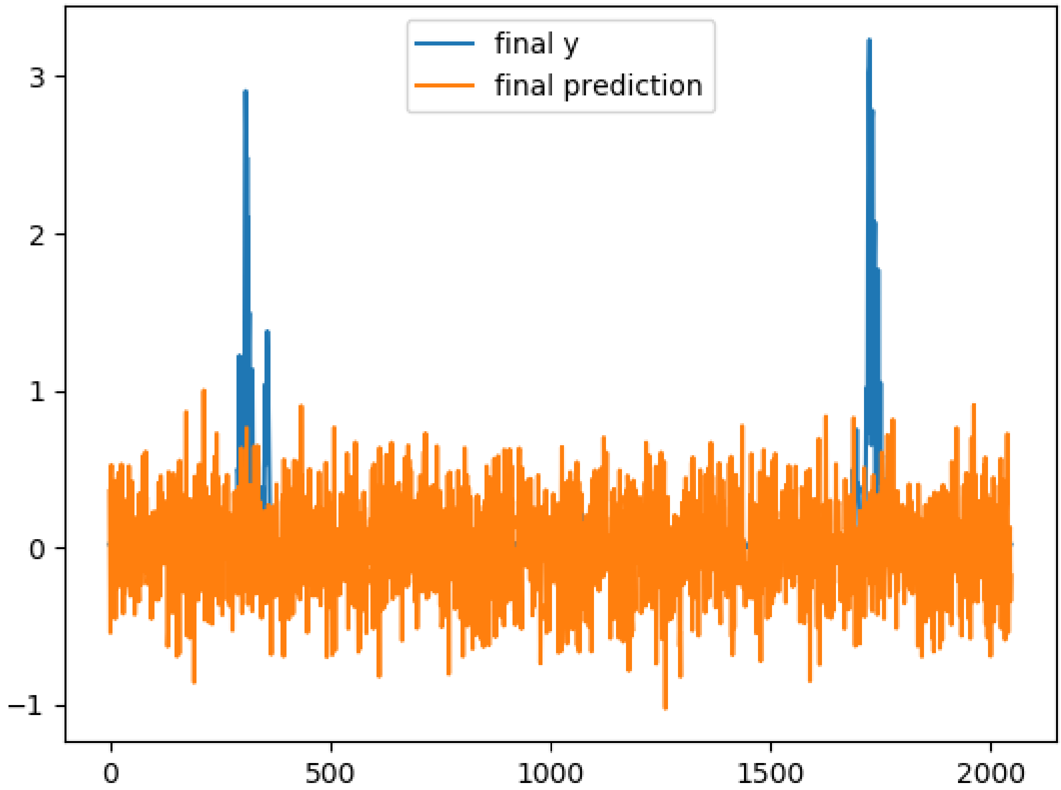


Fig 3.4

Note: Task4.1.py is attached to the document and **as my system is hanging a lot the iteration is been limited to 100.**

As shown in fig : 3.5 the predicted output is [0.0211,0.0098,0.0078] for the Input scaled data : ([0.0211,0.0098,0.0078]) with the loss of [0.1636360287666]**.**

Output :

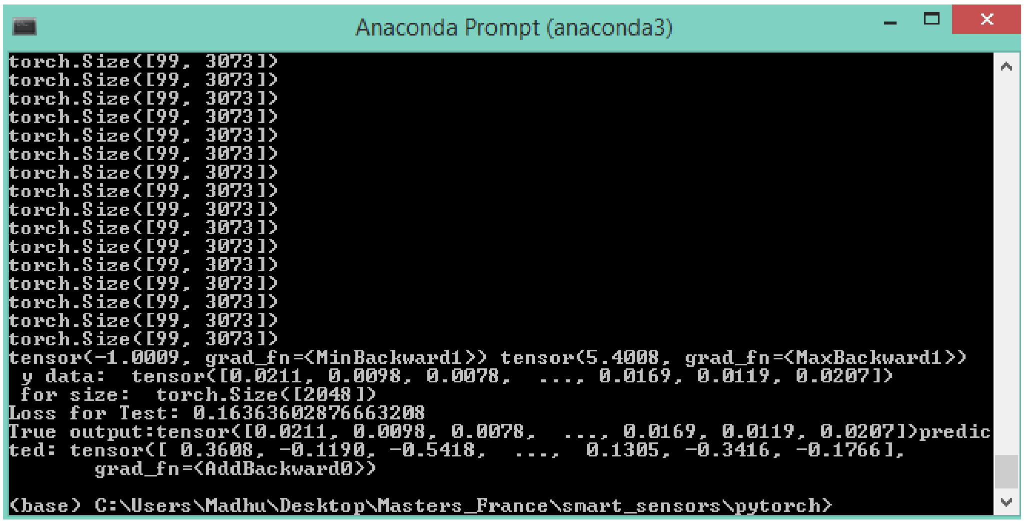


Fig 3.5

Note : Due to unknown dimension Maxpool was not used for down sampling the input data.

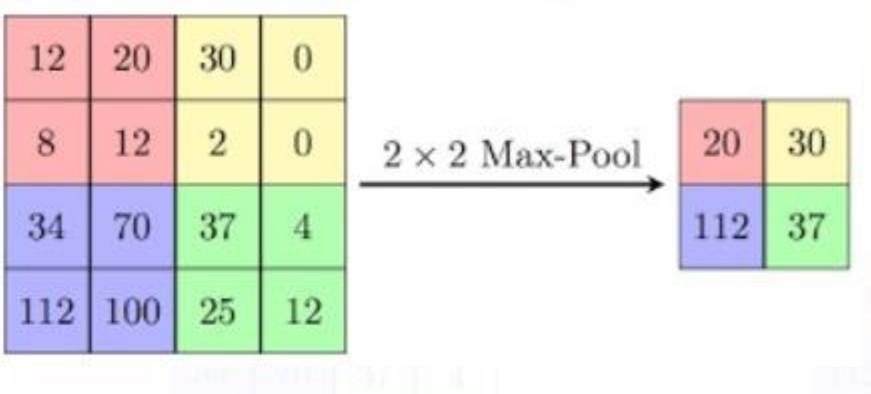


Fig 4.3 (Maxpooling)

Task 5. Plot and tabulate the results, then discuss them and provide conclusions.

The graph is been shown below based on the different parameters used.

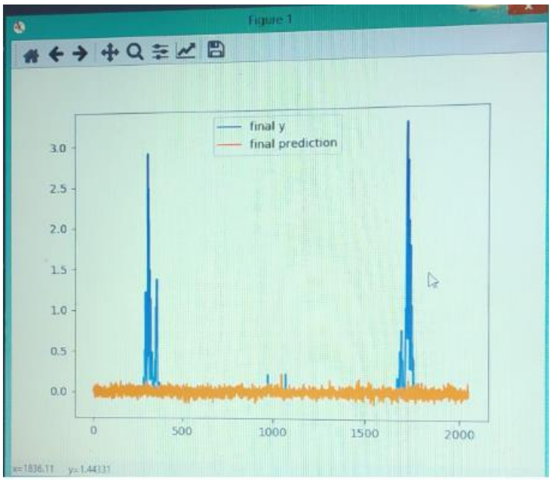


Fig 5.1

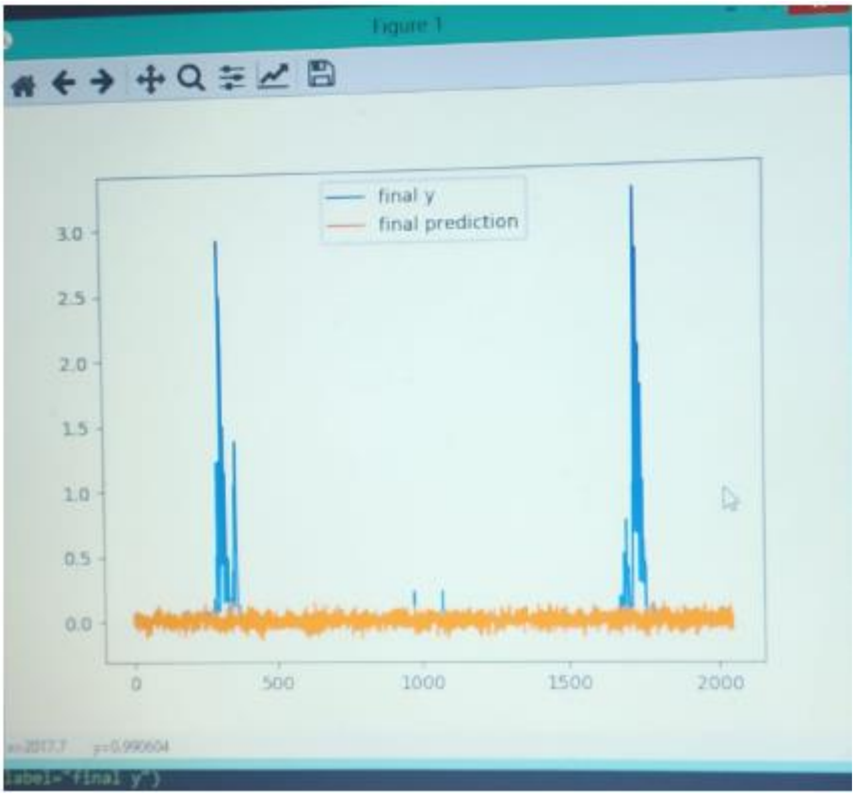


Fig 5.2

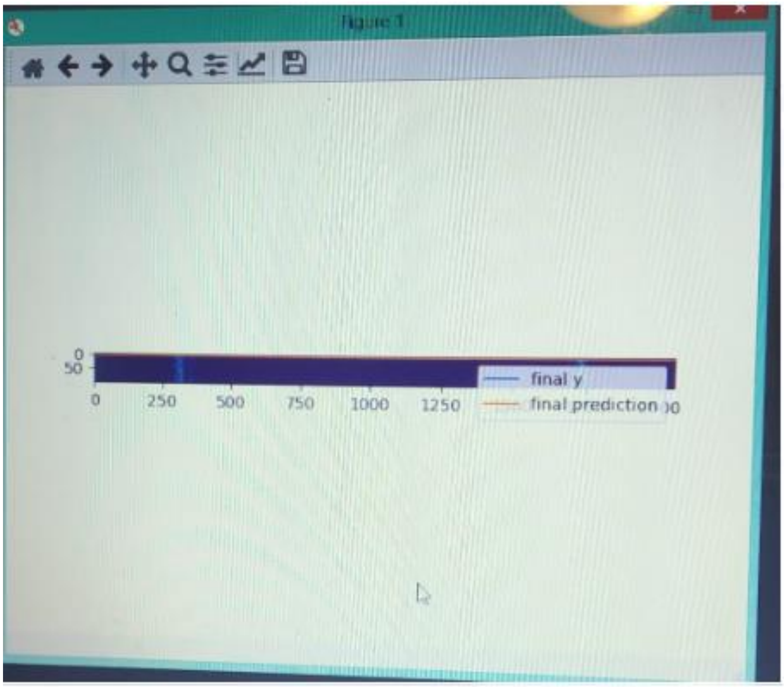


Fig 5.3

For the **figure 5.3** the literation was only 2, for **fig : 5.2** iterartion is 50 and for **Fig : 5.1** iteration is 100.

As more the data, then more the iteration is needed and the loss value will be less and the precision will be more. Based on the Application, size of the input data and the output prediction, the Activation and optimizer should be selected.