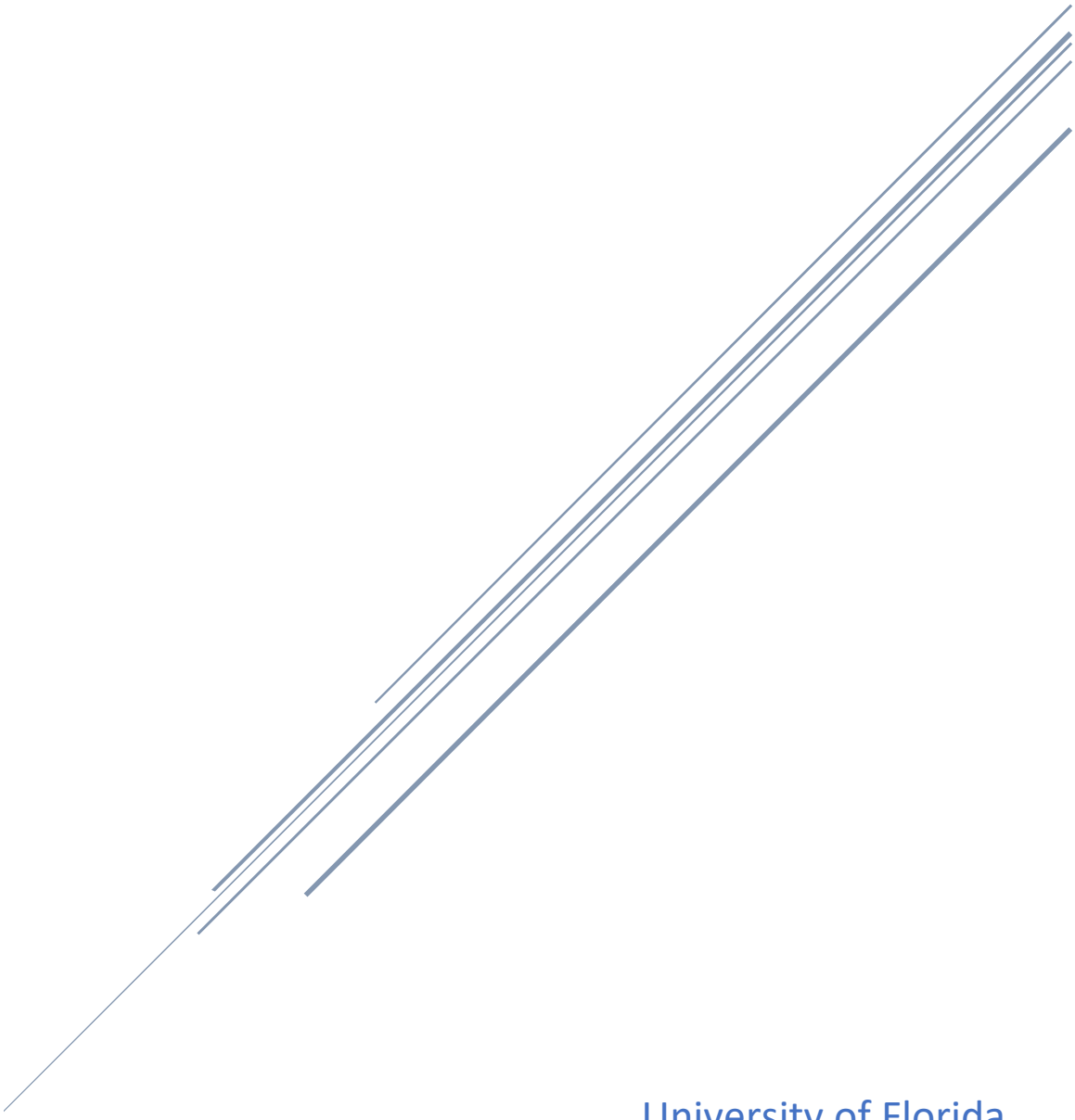
CAP 6615 – NEURAL NETWORKS

Programming Assignment 1 -- Single-Layer Perceptron

Group Member:

Fugang Deng, Wei He, Hang Ye, Zihang Huang, Jianan He, Weijia Sun



University of Florida
CAP 6615

CAP 6615 - Neural Networks - Programming Assignment 1 -- Single-Layer Perceptron

Group Member: Fugang Deng, Wei He, Hang Ye, Zihang Huang, Jianan He, Weijia Sun

Spring Semester 2022

28 Jan 2022

---

1. Network parameters

Parameter name	Value
learning rate	0.001
weights	256*256
Input dimension	16*16
Output dimension	16*16
epoch	600
Batch size	10
Standard	0.001, 0.002, 0.003, 0.005, 0.01, 0.02, 0.03, 0.05 and 0.1

Table 1 Parameters

To run the code successfully, we would like to present the external library we used in python, please make sure you install all the necessary library before running the code.

Library Name
torch
PIL
numpy
matplotlib

Table 2 Necessary Library

2. Python code for your SLP

```
# Step 2
# define a neural network
class Perceptron(nn.Module):
    def __init__(self, input_size, num_classes):
        super(Perceptron, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)
        self.activate = nn.Sigmoid()

    def forward(self, x):
        res = self.linear(x)
        res = self.activate(res)
        return res
```

Figure 1 Python code

It is very easy and simple to build a single layer perceptron due to the powerful external python library. In this project, the single layer perceptron contains a fully connected layer and also a activation function named “Sigmoid”. Below the function “__init__()”, the “forward()” is to call the function in the Module class.

3. Training set configuration

1. Training set without Gaussian Nosie

As requirement, we make every character into a 16x16-pixel image without grey value and present them in a 2*5 array format figure, including training image and output image both.

Figure 2 is the input image before training, and *figure 3* is the output image after training. All these training sets do not add the noise.

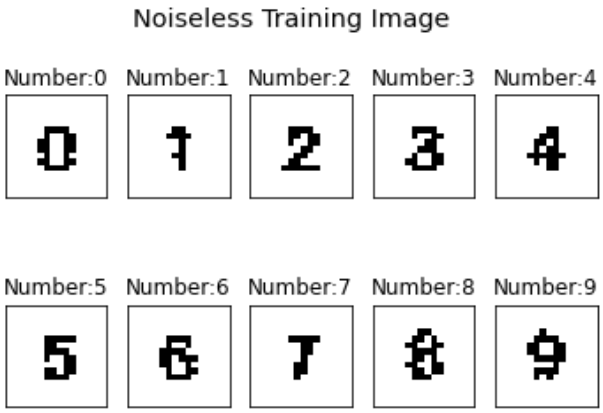


Figure 2 Image before training

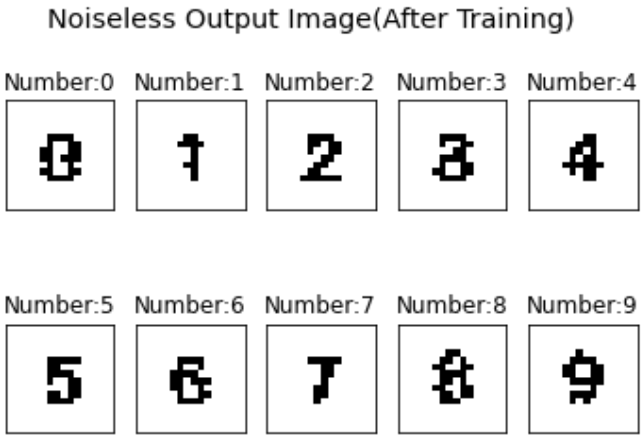


Figure 3 Image after training

From above training output image, we can see there is slightly difference from the input image(noiseless), especially the number “0” and number “3”.

2. Training set with Gaussian Noise

As the requirement, we should add the Gaussian Noise into the training set with a given array of standard deviation. To fully observe the differences between the image number with different level of noise, we plot them all together in one plot. See as *figure 4*,

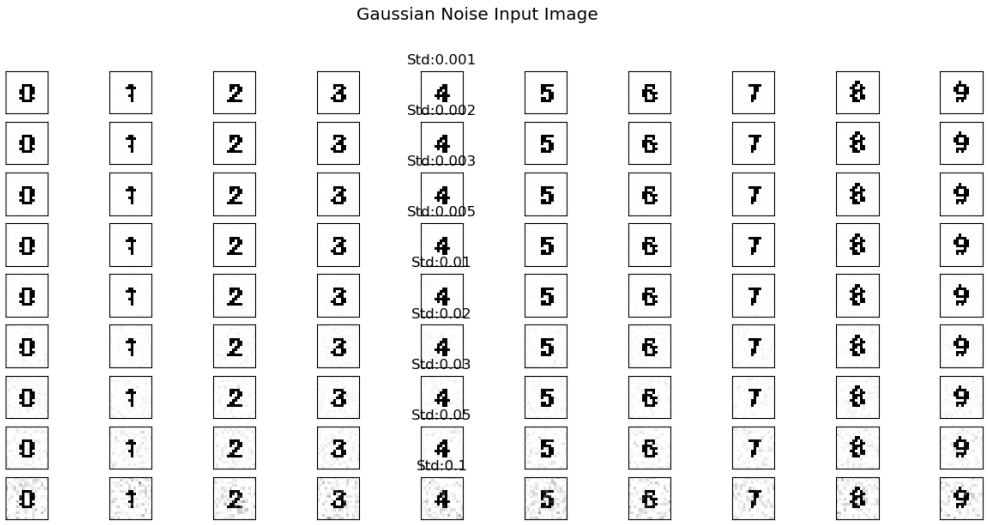


Figure 4 Noise Image Input

Above the is the input image with Gaussian Noise and standard deviation of (0.001, 0.002, 0.003, 0.005, 0.01, 0.02, 0.03, 0.05, and 0.1). We can see that the background has been added some gray dots. (To see a clearer image, please run the code in Jupyter notebook Cell [44])

Then we use this noise-corrupted image to train our SLP, then we get the result as following:

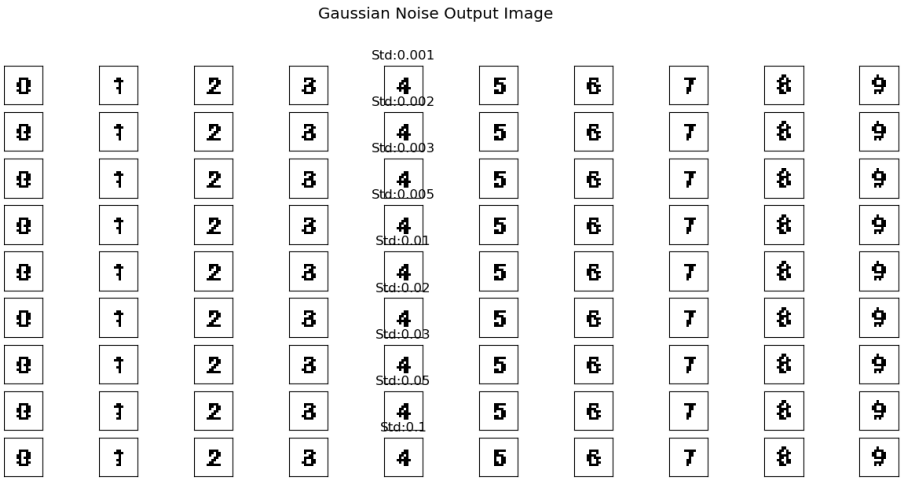


Figure 5 Output Image

As you can see from the *Figure 5*, the result shows

4. SLP output results for noiseless input in terms of *Fh* & *Ffa*

Figure 6 is the table of *Fh* and *Ffa* of noiseless input. We can see that the *Ffa* would be very little. We also present them as scatter plot, see *figure 7*.

Image(Number)	Fh	Ffa
0	1.0	0.00433
1	1.0	0.00211
2	1.0	0.00141
3	1.0	0.00213
4	1.0	0.0017
5	1.0	0.00142
6	1.0	0.00122
7	1.0	0.00106
8	1.0	0.00095
9	1.0	0.00086

Figure 6 Noiseless Output Fh&Ffa

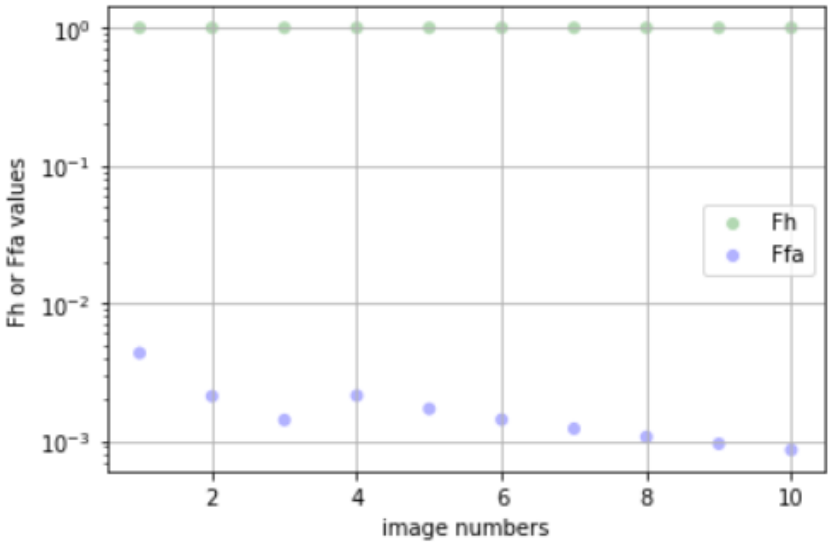


Figure 7 Ffa&fh scatter plot(noiseless)

We can see the Fh is 1 for all image numbers. This is wired a little bit, but it still makes senses because we remove all the grey value, only left is black and white so that such feature would be

very easy to learn. We examine and compare the image, it turns out that every black pixel is in the correct place. So the Fh should be all "1".

5. Pseudocode and **Python code** for algorithms to compute *Fh* & *Ffa*

Algorithm 1 Calculate_Fh

Input:

1: $Input\_dataset, Output\_dataset$: grey scale value of image

Output: Fh_array

```
2: Fh_denominator, Fh_numerator = 0
3: for i in size(Input_dataset) do
4:     for  j in size(output_dataset) do
5:         if input_dataset[j][i] represent black then
6:             Fh_denominator = Fh_denominator + 1
7:             if output_dataset[j][i] represent black then
8:                 Fh_numerator = Fh_numerator + 1
9:             end if
10:        end if
11:    end for
12:    Fh = Fh_numerator / Fh_denominator
13:    Fh_array[j] = Fh
14: end for
15: return Fh_aray
```

Figure 8 Pseudocode of Computing Fh

Algorithm 2 Calculate_Ffa

Input:

$Input\_dataset, Output\_dataset$: grey scale value of image

Output: Ffa_array

$Ffa\_denominator, Ffa\_numerator = 0$
for j in $(Input\_dataset)$ **do**
  for i in $size(output\_dataset)$ **do**
    if $input\_dataset[j][i]$ represent white **then**
      $Ffa\_denominator$ += 1
      if $output\_dataset[j][i]$ represent black & $input\_dataset[j][i]$ represent white **then**
        $Ffa\_numerator$ += 1
      end if
    end if
  end for
  $Ffa\_array[j] = Ffa\_numerator$ / $Ffa\_denominator$
end for
return Ffa_aray

Figure 9 Pseudocode of Computing Ffa

```
# Step 4b
# Calculate Fh
def calculateFh(input_dataset, output_dataset):
    x, y = input_dataset.shape
    Fh_denominator = 0     # denominator of Fh
    Fh_numerator = 0       # numerator of Fh
    Fh_array = np.zeros([x])
    for j in range(x):
        for i in range(y):
            if input_dataset[j][i] == 0:
                Fh_denominator = Fh_denominator + 1
                if output_dataset[j][i] == 0:
                    Fh_numerator = Fh_numerator + 1
        Fh = Fh_numerator / Fh_denominator
        Fh_array[j] = Fh
    return Fh_array
```

Figure 10 Python code for Fh

```python
# Calculate Ffa
def calculateFfa(input_dataset, output_dataset):
    x, y = input_dataset.shape
    Ffa_denominator = 0    # Ffa分母
    Ffa_numerator = 0      # Ffa分子
    Ffa_array = np.zeros([x])
    for j in range(x):
        for i in range(y):
            if input_dataset[j][i] == 1:
                Ffa_denominator = Ffa_denominator + 1
            if output_dataset[j][i] == 0 and input_dataset[j][i] == 1:
                Ffa_numerator = Ffa_numerator + 1
        Ffa = Ffa_numerator / Ffa_denominator
        Ffa_array[j] = Ffa
    return Ffa_array
```

Figure 11 Python code for Ffa

6. SLP output results for noise-corrupted input as table and graph of *Fh* and *Ffa*

Number	Stdev=0.001	Stdev=0.002	Stdev=0.003	Stdev=0.005	Stdev=0.01	Stdev=0.02	Stdev=0.03	Stdev=0.05	Stdev=0.1
0	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043
1	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0042	fh:1.0 ffa:0.0042
2	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0028	fh:1.0 ffa:0.0028
3	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0032	fh:1.0 ffa:0.0032
4	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0026	fh:1.0 ffa:0.0026
5	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0007	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0007	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0028
6	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0006	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0006	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0018	fh:1.0 ffa:0.0024
7	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0005	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0005	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0016	fh:1.0 ffa:0.0021
8	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0005	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0005	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0019
9	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0004	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0004	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0013	fh:1.0 ffa:0.0017

Figure 12 Table of noise Fh and Ffa

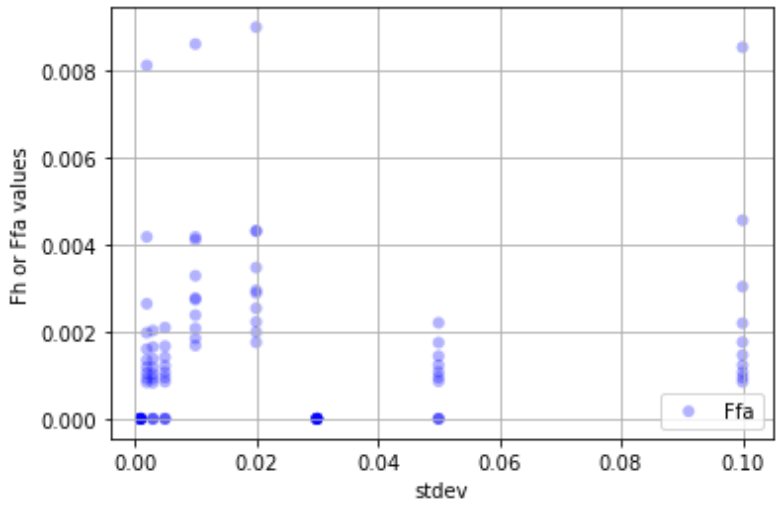


Figure 13 Scatter Plot of ffa

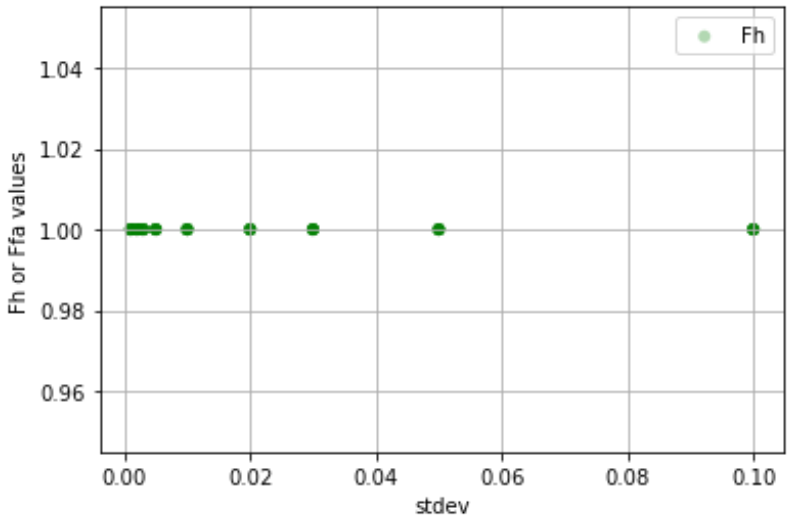


Figure 14 Scatter Plot of fh

7. Discussion

To fully discussion why the perceptron works, we would like to briefly introduce the workflow at first.

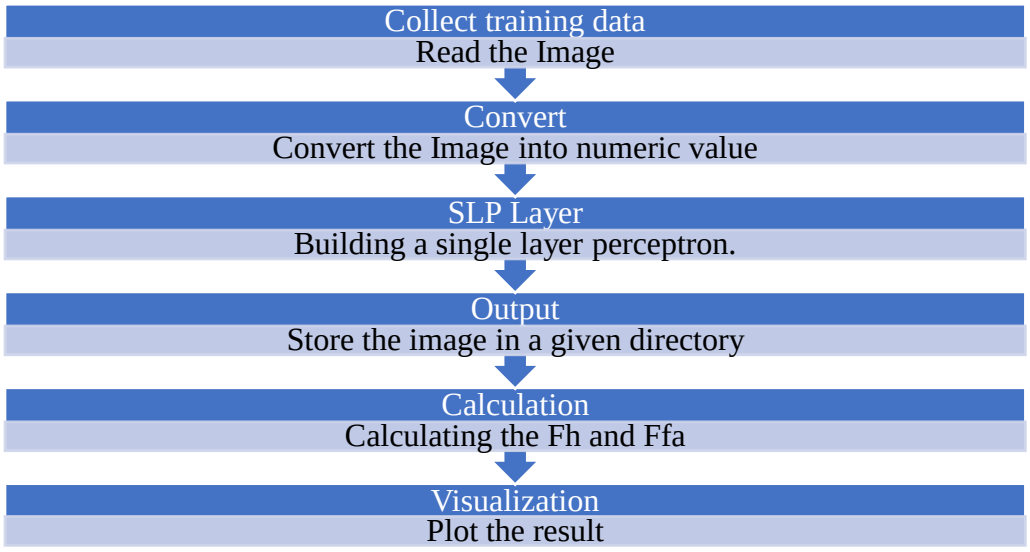


Figure 15 Workflow

In neural network, all the images must be converted into a matrix so that the computer can process with it. In this project, we use an external library called "Dataset" from Pytorch. After we convert the image into a matrix, we can start computing and move into the next layer.

In the next layer, we build a single layer perceptron and use sigmoid function to activate. Now, we can get represent out SLP in a mathematical form as below:

$$o_j = sigmoid(\sum_{i=1}^{n} W_i X_i + b)$$

Where $Sigmoid()$ is our activation function, and o_j is the output, W is the weight matrix, and X is the input matrix (namely the image value matrix in this project), and b is the bias.

And once we got the output, we now can convert the matrix into an image by the same way we did in the convert layer using Dataset library. And what the rest part is calculating the Fh and Ffa and the visualization.

1. **Solution for "Sigmoid ()" fail to converge.**

We try to use different activation function in our SLP, such as Tanh, ReLU, LogSigmoid, Hardsigmoid. Only Sigmoid () has excellent performance (very low mean error) for noiseless

image. But there is serious problem of Sigmoid (), that is the Sigmoid would fail to converge when training with noise-corrupted image.

```
Training dataset with noise standard deviation 0.001
[0/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[10/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[20/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[30/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[40/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[50/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[60/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[70/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[80/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[90/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[100/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[110/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[120/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[130/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[140/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[150/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[160/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
[170/600] Loss: 0.2561 MAE: 0.4919 Mean Error: 0.4017 STD: 0.3080
```

Figure 16 Fail to Converge

For further improvement, we can try other different activation function to see if the Loss would be converged.

To improve the performance of the SLP, we were trying different kinds of the activation function to see if the accuracy would be better. But it turns out the “Sigmoid” has the best performance. But we can still change the “Learning rate”, “Epoch” and “Batch_size” to compare the result.

8. Appendix

For extra credit, we built a SMNN. The workflow would basically the same with the SLP. However, we have a different result here. We would like to firstly present the result and add some descriptions, and then for the python code of SMNN (similar with *section 2* above), there will be a screen shot of the main code and we will discuss how it works. As the final part of this appendix, we would likely to discuss how we can improve its performance in the future.

1. **Results presentation and Description(Noiseless)**

Same as the above section, we would present the Input image and output image without noise in SMNN to get a better comparison. *See as figure18.*

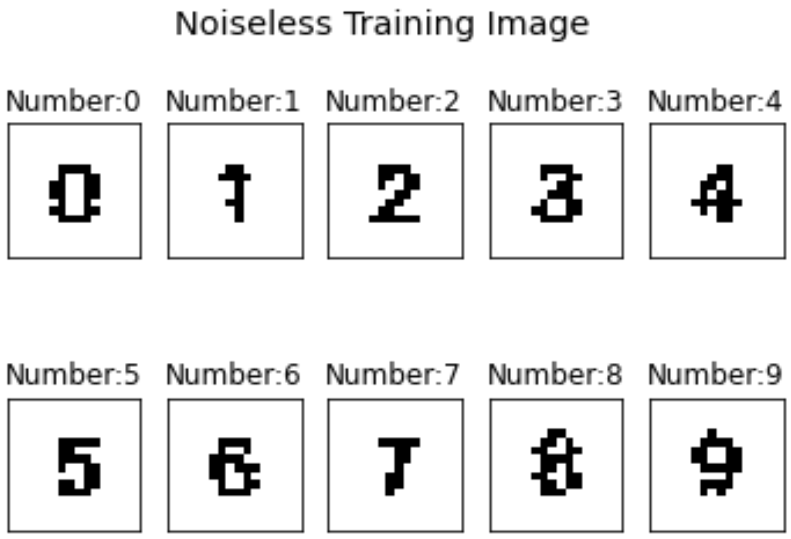


Figure 17 Noiseless Input Image(SMNN)

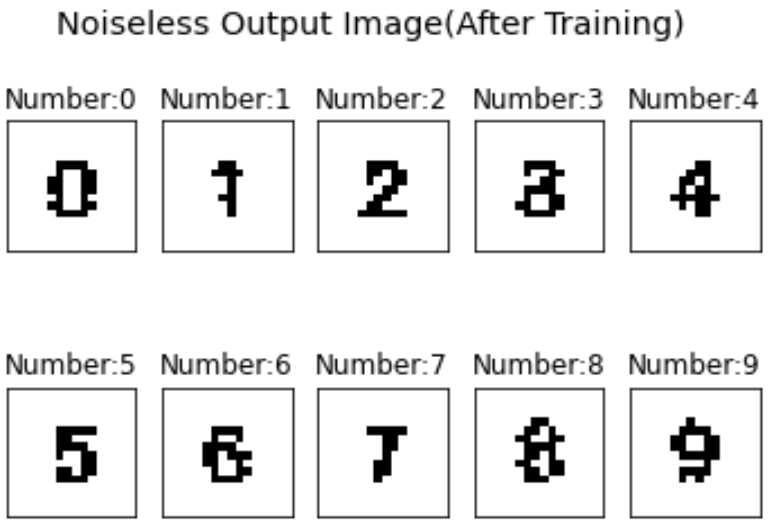


Figure 18 Noiseless output image (SMNN)

We can see above output images have a better result because we add multiple fully connected layer. But there is still a slightly difference between the input and output, you can see it from the number “3”. To gain tabular forms and graphical forms of *Fh* and *Ffa,* see as figures below.

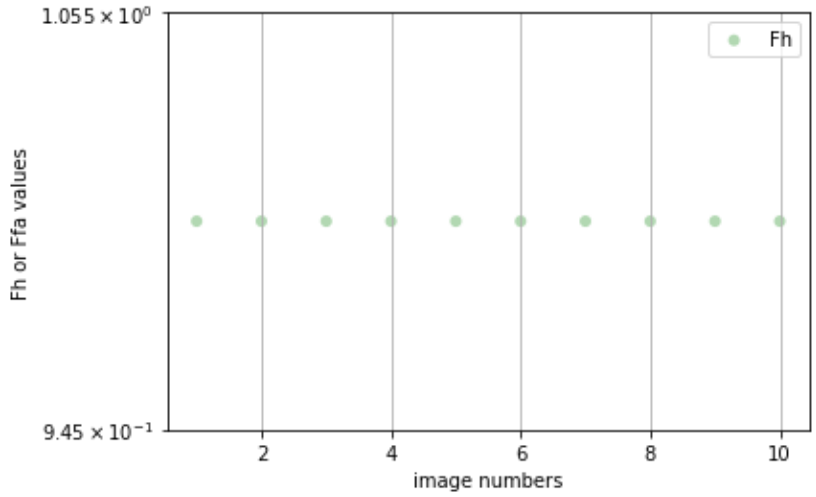


Figure 19 Fh Noiseless(SMNN)

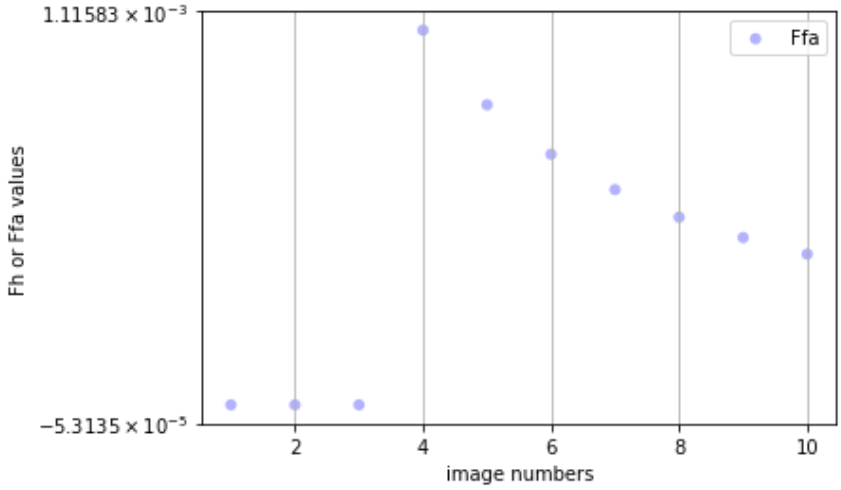


Figure 20 Ffa Noiseless (SMNN)

Image(Number)	Fh	Ffa
0	1.0	0.0
1	1.0	0.0
2	1.0	0.0
3	1.0	0.00106
4	1.0	0.00085
5	1.0	0.00071
6	1.0	0.00061
7	1.0	0.00053
8	1.0	0.00047
9	1.0	0.00043

*Figure 21 Tabular Form of Fh*Ffa(SMNN)*

From the data visualization above, we can see that the Fh is always "1", this means the black pixels are all in the correct place. We fully examine that from the *figure 19* and *figure 18*, it shows that all the black pixels are exactly in the place where they supposed to be. However, the white pixels have the same situation, so we can see the Ffa is different for every image numbers.

2. Results presentation and Description with Gaussian Noise

In this procedure, we add the Gaussian noise into the image and then train those noise-corrupted images, same as SLP before. We would present the results as followings,

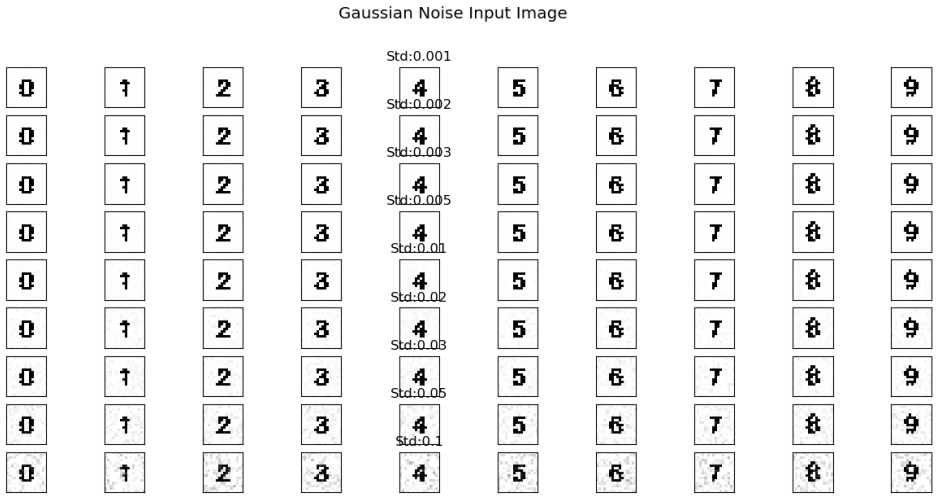


Figure 22 Noise Input Image(SMNN)

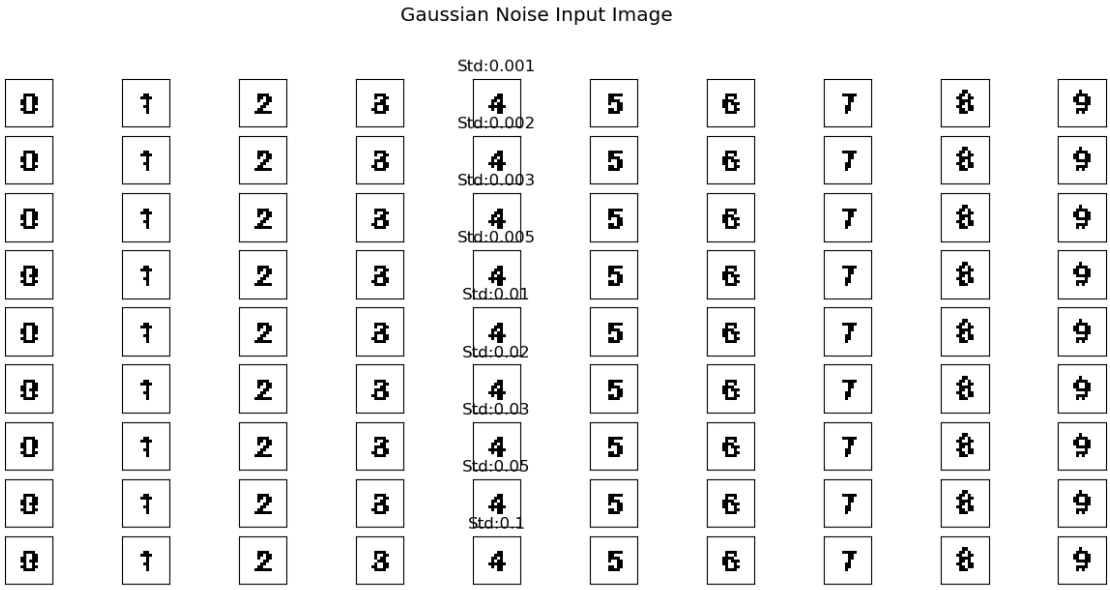


Figure 23 Noise output Image (SMNN)

Above is the input image and output image with noise, probably you cannot see a very clear image here, so if you want to see a clearer image, you can run the code. But just for clarification here, there is a lot of grey dots in the input image, and however, in the output image, there is not any grey dots. For the tabular form and scatter plot for *Fh* and *Ffa*, you can see the *figure 24 to figure 25*.

Number	Stdev=0.001	Stdev=0.002	Stdev=0.003	Stdev=0.005	Stdev=0.01	Stdev=0.02	Stdev=0.03	Stdev=0.05	Stdev=0.1
0	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043	fh:1.0 ffa:0.0043
1	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0042
2	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0028
3	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0021	fh:1.0 ffa:0.0032
4	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0017	fh:1.0 ffa:0.0026
5	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0007	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0014	fh:1.0 ffa:0.0021
6	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0006	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0012	fh:1.0 ffa:0.0018
7	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0005	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0011	fh:1.0 ffa:0.0016
8	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0005	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0014
9	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0004	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0009	fh:1.0 ffa:0.0013

Figure 24 Tabular form of Fh and Ffa(SMNN)

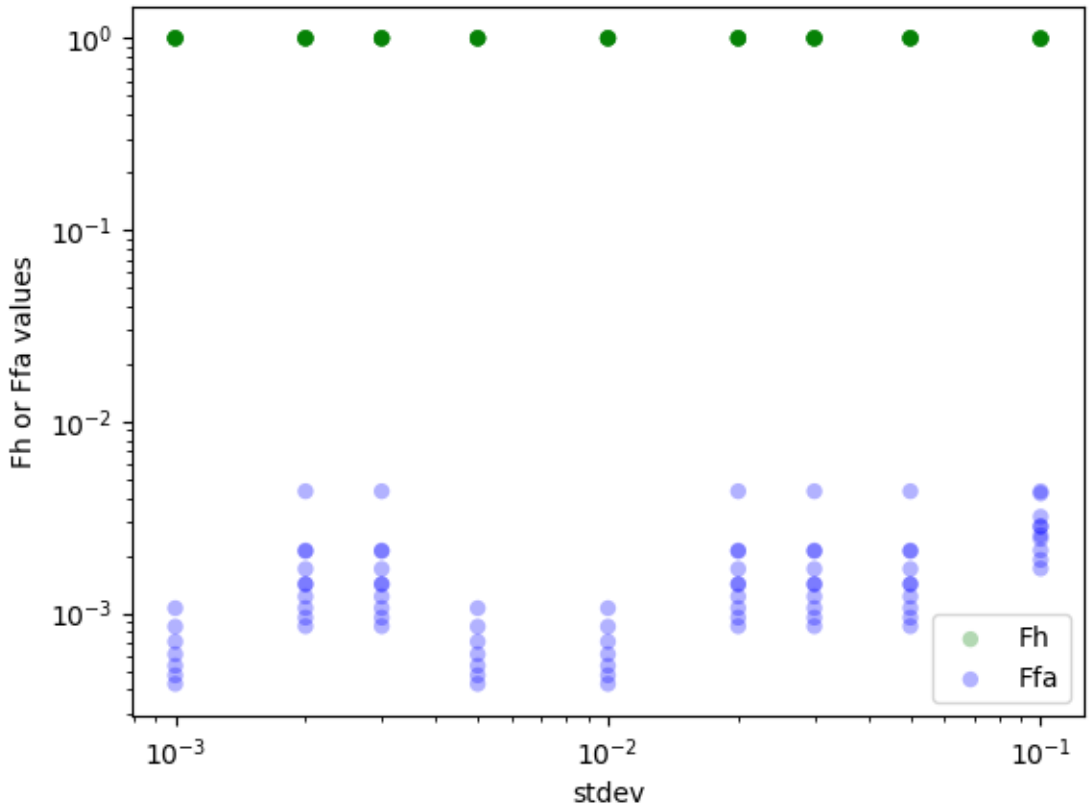


Figure 25 Scatter plot of Fh and Ffa(SMNN)

3. Python code of SMNN

```
# Step 2
# define a neural network
class Perceptron(nn.Module):

    def __init__(self, input_size, num_hidden, num_classes):
        super(Perceptron, self).__init__()
        self.inputlayer = nn.Linear(input_size, num_hidden)
        self.hiddenlayer = nn.Linear(num_hidden, num_hidden)
        self.outputlayer = nn.Linear(num_hidden, num_classes)
        self.activate = nn.Sigmoid()

    def forward(self, x):
        res = self.inputlayer(x)
        res = self.activate(res)
        res = self.hiddenlayer(x)
        res = self.activate(res)
        res = self.outputlayer(x)
        res = self.activate(res)
        return res
```

Figure 26 Python Code of SMNN

I will briefly descript the code above. As you can see, we add more layers in this part. It is basically the same with the SLP but with extra lines of code. So, it is very simple to build multiple layers neural network by Pytorch library.

4. Discussion

We use different way to improve the performance of SMNN. But the sigmoid() always have a better performance here. We were figuring out why this happens and trying to find the mathematical reason behind the Sigmoid function. Thus, we conclude that Sigmoid has a better performance because the output range is limited in Sigmoid function, so the data is not easily diverged during the transfer process. Of course, there are corresponding disadvantages, that is, the gradient is too small when converge.

We can try to set different Learning rate, batch size and also threshold to compare with performance in the future.